



# CapTIvate™ Technology Guide

## Users Guide

# Contents

<b>1</b>	<b>Main Page</b>	<b>1</b>
<b>2</b>	<b>Getting Started</b>	<b>2</b>
<b>3</b>	<b>Introduction</b>	<b>3</b>
<b>4</b>	<b>Capacitive Sensing Basics</b>	<b>5</b>
4.1	Self Capacitance . . . . .	6
4.1.1	Self Capacitance Parasitics . . . . .	6
4.1.2	Parasitics and the E-Field . . . . .	7
4.1.3	E-Field Propagation . . . . .	7
4.1.4	Self Capacitance Summary . . . . .	8
4.2	Mutual Capacitance . . . . .	8
4.2.1	Mutual Capacitance Parasitics . . . . .	9
4.2.2	Tightly Coupled E-Fields . . . . .	9
4.2.3	Overlay Material Requirement . . . . .	10
4.2.4	Mutual Capacitance Sensitivity . . . . .	10
4.2.5	Mutual Capacitance Summary . . . . .	10
4.3	True Matrix Capability . . . . .	10
<b>5</b>	<b>Technology</b>	<b>12</b>
5.1	Charge Transfer Technology . . . . .	12
5.1.1	Charge and Transfer Phases in Self Mode . . . . .	13
5.1.2	Mutual Mode Charge and Transfer Phases . . . . .	13
5.2	The CapTIvate™ Peripheral . . . . .	13
5.2.1	Capacitance Measurement . . . . .	14
5.2.1.1	IO Mux . . . . .	15
5.2.2	Auxillary Digital Functions . . . . .	17
5.2.2.1	Low Power Operation . . . . .	17
5.2.2.2	Conversion Timing . . . . .	23
5.2.2.3	EMC . . . . .	23
5.2.2.4	Reference Capacitors . . . . .	24

---

<b>6 Design Guide</b>	<b>25</b>
6.1 Introduction	25
6.2 Starting a New Design	25
6.3 Design Process	26
6.4 Best Practices	28
6.4.1 Mechanicals	29
6.4.2 Common Layout Considerations	33
6.4.2.1 Routing	34
6.4.2.2 Electrode Material	36
6.4.2.3 Spacing Between Electrodes	38
6.4.2.4 Shapes	38
6.4.2.5 Crosstalk	38
6.4.2.6 Adjacent Capacitive Touch Signals	38
6.4.2.7 Digital Signals	39
6.4.2.8 LEDs/LED Backlighting	39
6.4.2.9 Ground Planes	41
6.4.2.10 Separation	42
6.4.2.11 Pour	42
6.4.2.12 Mutual Capacitance Traces	44
6.5 Buttons	45
6.5.1 Self Capacitive Buttons	45
6.5.1.1 Self Capacitive Button Shapes	46
6.5.2 Mutual Capacitive Buttons	48
6.5.2.1 Mutual Capacitive Button Shapes	49
6.6 Sliders and Wheels	53
6.6.1 Slider and Wheel Resolution	53
6.6.2 Self Capacitive Sensor Shapes	53
6.6.2.1 Self Capacitive Slider	53
6.6.2.2 Self Capacitive Wheel	54
6.6.3 Mutual Capacitive Sensor Shapes	54
6.6.3.1 Mutual Capacitive Slider	55
6.6.3.2 Mutual Capacitive Wheel	55
6.7 Proximity	57
6.7.1 Proximity Shapes	57
6.8 Ultra Low Power	63
6.8.1 Introduction	63
6.8.2 Expectations	63
6.8.3 Optimization Steps	64
6.8.3.1 Selecting a Low Frequency Clock	65
6.8.3.2 Optimizing the Application Scan Period	65

---

6.8.3.3	Optimizing for the Smallest Parasitic Capacitance . . . . .	66
6.8.3.4	Optimizing for the Shortest Measurement Time . . . . .	66
6.8.3.5	Using the Wake-on-Proximity State Machine Effectively . . . . .	67
6.8.4	Walking Through the Optimization Process . . . . .	67
6.8.5	A Note about Mutual Capacitance and Bias Current . . . . .	70
6.9	Moisture . . . . .	70
6.9.1	Introduction . . . . .	70
6.9.2	Expectations . . . . .	70
6.9.3	Moisture Physics . . . . .	70
6.9.4	Moisture Problems . . . . .	71
6.9.5	Moisture Mitigation Techniques . . . . .	71
6.9.5.1	Moisture Tolerance Applications . . . . .	71
6.9.5.2	Spill Rejection Applications . . . . .	72
6.10	Noise Immunity . . . . .	72
6.10.1	Introduction . . . . .	72
6.10.1.1	How to Use This Section . . . . .	72
6.10.1.2	Additional Resources . . . . .	72
6.10.1.3	Aggressor-Victim Philosophy . . . . .	73
6.10.2	The Three Sided Approach . . . . .	73
6.10.2.1	Introduction to Electromagnetic Compatibility (EMC) Standards . . . . .	74
6.10.2.2	Signal-to-Noise Ratio . . . . .	75
6.10.3	Types of Noise . . . . .	75
6.10.3.1	Differential Mode Supply Rail Noise . . . . .	75
6.10.3.2	Electrostatic Discharge (ESD) . . . . .	76
6.10.3.3	Electrical Fast Transient (EFT) . . . . .	77
6.10.3.4	RF Susceptibility (Conducted and Radiated) . . . . .	78
6.10.4	Noise Mitigation Steps . . . . .	88
6.10.4.1	Self versus Mutual Capacitance for Noise Immunity . . . . .	88
6.10.4.2	Noise Immunity Hardware Check List . . . . .	88
6.10.4.3	Tuning Check List . . . . .	92
6.11	References . . . . .	93
7	<b>Design Center GUI</b> . . . . .	94
7.1	Introduction . . . . .	94
7.1.1	Key features . . . . .	95
7.1.2	Hardware and software requirements . . . . .	95
7.1.2.1	Additional Linux requirements . . . . .	95
7.1.3	Licensing . . . . .	95
7.2	CapTlivate™ Design Center Quick start . . . . .	95
7.2.1	Start the Design Center . . . . .	96

---

7.2.2	Place the sensors . . . . .	96
7.2.3	Place the MSP430 controller . . . . .	105
7.2.4	Connect sensors to MSP430 capacitive touch I/O ports . . . . .	105
7.2.5	Generate source code . . . . .	107
7.2.6	Load and run the MSP430FR26xx/25xx generated firmware project using CCS or IAR . . . . .	108
7.2.7	Communicating with the target . . . . .	108
7.3	CapTivate™ Design Center Users guide . . . . .	113
7.3.1	Design Center GUI panels . . . . .	113
7.3.1.1	Menu bar . . . . .	115
7.3.1.2	Object Selection Tab . . . . .	115
7.3.1.3	Miscellaneous Selection Tab . . . . .	115
7.3.1.4	Design Canvas . . . . .	117
7.3.2	Menu Descriptions . . . . .	119
7.3.2.1	File . . . . .	119
7.3.2.2	Edit . . . . .	120
7.3.2.3	Options . . . . .	121
7.3.2.4	Communications . . . . .	122
7.3.2.5	Help . . . . .	123
7.3.2.6	Topics . . . . .	123
7.3.2.7	About . . . . .	123
7.3.3	Controller Properties . . . . .	124
7.3.3.1	Device configuration . . . . .	124
7.3.3.2	Enable target communications and target MCU communications interface . . . . .	125
7.3.3.3	Select Compile Time options . . . . .	125
7.3.3.4	Mapping sensor ports to controller pins . . . . .	126
7.3.3.5	Making manual connections graphically . . . . .	129
7.3.3.6	Source Code Generation . . . . .	130
7.3.3.7	Controller target data displays . . . . .	130
7.3.3.8	Controller Conversion Control parameters . . . . .	131
7.3.3.9	Controller Scan Time Estimator . . . . .	131
7.3.4	Sensor Properties . . . . .	132
7.3.4.1	Position Display . . . . .	134
7.3.4.2	Sensor Configuration . . . . .	135
7.3.4.3	Enable Target Communications from Sensor Properties . . . . .	140
7.3.4.4	Sensor target data displays . . . . .	140
7.3.4.5	Tuning Sensor performance . . . . .	140
7.3.5	Displaying Target Data . . . . .	141
7.3.5.1	Channel Bar Chart . . . . .	142
7.3.5.2	Oscilloscope Plot . . . . .	143
7.3.5.3	Channel Table . . . . .	144

---

7.3.6	SNR Measurements . . . . .	144
7.4	Loading and running generated projects . . . . .	148
7.4.1	Importing/opening projects . . . . .	148
7.4.1.1	Code Composer Studio . . . . .	148
7.4.1.2	IAR Embedded Workbench . . . . .	151
7.4.2	Enable target communications in the Design Center . . . . .	155
<b>8</b>	<b>Device Family</b>	<b>156</b>
8.1	Standard Devices . . . . .	156
8.2	Small Form Factor (Die Size) Devices . . . . .	156
<b>9</b>	<b>Software Library</b>	<b>158</b>
9.1	Introduction . . . . .	158
9.1.1	Using This Chapter . . . . .	158
9.1.2	Device and Tools Support . . . . .	159
9.1.3	Delivery Mechanism . . . . .	159
9.1.4	Change Control . . . . .	159
9.2	Overview . . . . .	160
9.2.1	Programming Model . . . . .	160
9.2.2	Organization and Architecture . . . . .	166
9.3	Getting Started . . . . .	169
9.3.1	Starting from Scratch with the Starter Project . . . . .	169
9.3.2	Adding CapTlivate™ to an Existing Project . . . . .	178
9.4	How-To . . . . .	187
9.4.1	Use the Top Level API . . . . .	187
9.4.2	Register a Callback Function . . . . .	189
9.4.3	Access Element State Data . . . . .	190
9.4.4	Access the Dominant Button . . . . .	192
9.4.5	Access Slider or Wheel Position Data . . . . .	194
9.4.6	Access Element Measurement Data . . . . .	196
9.4.7	Update Sensors Independently . . . . .	197
9.4.8	Update a Sensor's Raw Data Only . . . . .	199
9.4.9	Enable an IO as a Shield . . . . .	199
9.4.10	Create a Custom EMC Configuration . . . . .	200
9.4.11	Stream Unformatted Data to the Design Center GUI . . . . .	201
9.5	Technical Details . . . . .	202
9.5.1	Devices with CapTlivate™ Software in ROM . . . . .	202
9.5.2	MSP430 CPUX Memory Model . . . . .	203
9.6	Base Module . . . . .	203
9.6.1	HAL . . . . .	203
9.6.2	Touch . . . . .	204

---

9.6.2.1	Sensor Update Routines . . . . .	204
9.6.2.2	Calibration Algorithms . . . . .	206
9.6.2.3	Environmental Drift Algorithms . . . . .	206
9.6.3	ISR . . . . .	207
9.6.4	Type Definitions . . . . .	207
9.7	Advanced Module . . . . .	207
9.7.1	Manager . . . . .	208
9.7.2	Buttons . . . . .	208
9.7.3	Sliders and Wheels . . . . .	208
9.7.4	EMC . . . . .	208
9.7.4.1	EMC Module Background . . . . .	208
9.7.4.2	Using the EMC Module . . . . .	209
9.7.4.3	EMC Module Algorithms . . . . .	214
9.8	Communications Module . . . . .	216
9.8.1	Background . . . . .	216
9.8.2	Overview . . . . .	216
9.8.3	Interface Layer . . . . .	217
9.8.3.1	Using the Communications Module: Initializing the Interface . . . . .	217
9.8.3.2	Using the Communications Module: Handling Incoming Data . . . . .	217
9.8.3.3	Using the Communications Module: Writing Out Data . . . . .	217
9.8.3.4	Compile-Time Configuration . . . . .	218
9.8.4	Protocol Layer . . . . .	219
9.8.4.1	Introduction to Packet Types . . . . .	219
9.8.4.2	Format: Sensor Packets . . . . .	221
9.8.4.3	Format: Cycle Packets . . . . .	222
9.8.4.4	Format: Trackpad Packets . . . . .	223
9.8.4.5	Format: General Purpose Packets . . . . .	224
9.8.4.6	Format: Parameter Packets . . . . .	224
9.8.5	UART Driver . . . . .	230
9.8.5.1	Purpose of the Driver . . . . .	230
9.8.5.2	Driver Features . . . . .	231
9.8.5.3	Driver Overview . . . . .	231
9.8.5.4	Compile-time Driver Configuration Options . . . . .	232
9.8.5.5	Run-time Driver Configuration Options . . . . .	233
9.8.5.6	Using the Driver: Opening and Closing the Driver . . . . .	234
9.8.5.7	Using the Driver: Transmitting Data . . . . .	235
9.8.5.8	Using the Driver: Receiving Data . . . . .	238
9.8.5.9	Using the Driver: Error Handling . . . . .	240
9.8.5.10	Using the Driver: Example Application . . . . .	240
9.8.6	I2C Slave Driver . . . . .	242

---

9.8.6.1	Purpose of the Driver . . . . .	242
9.8.6.2	Driver Features . . . . .	242
9.8.6.3	Driver Overview . . . . .	243
9.8.6.4	Compile-time Driver Configuration Options . . . . .	245
9.8.6.5	Run-time Driver Configuration Options . . . . .	248
9.8.6.6	Using the Driver: Opening and Closing the Driver . . . . .	249
9.8.6.7	Using the Driver: Transmitting Data (I2C Read Operation) . . . . .	250
9.8.6.8	Using the Driver: Receiving Data (I2C Write Operation) . . . . .	250
9.8.6.9	Using the Driver: Error Handling . . . . .	251
9.8.6.10	Using the Driver: Example Application . . . . .	251
9.9	Benchmarks . . . . .	253
9.9.1	Memory Requirements . . . . .	253
9.9.2	Execution Times . . . . .	255
<b>10</b>	<b>MSP-CAPT-FR2633 Development Kit</b> . . . . .	<b>257</b>
10.1	Introduction . . . . .	257
10.1.1	Overview . . . . .	257
10.1.2	Key Features . . . . .	257
10.2	Getting Started with the MCU Development Kit . . . . .	259
10.2.1	Pre-Work . . . . .	259
10.2.2	Running an Example Project . . . . .	260
10.2.3	Example Project Locations . . . . .	261
10.3	Sensor Panel Demonstrations . . . . .	261
10.3.1	CAPTIVATE-BSWP Demonstration (Out of Box Experience) . . . . .	261
10.3.1.1	CAPTIVATE-BSWP Bonus Software Projects . . . . .	262
10.3.2	CAPTIVATE-PHONE Demonstration . . . . .	264
10.3.3	CAPTIVATE-PROXIMITY Demonstration . . . . .	268
10.4	Hardware . . . . .	270
10.4.1	CAPTIVATE-FR2633 Processor PCB Overview . . . . .	270
10.4.1.1	Features . . . . .	272
10.4.2	CAPTIVATE-PGMR Programmer PCB Overview . . . . .	276
10.4.2.1	Features . . . . .	277
10.4.3	CAPTIVATE-ISO (Communications Isolation PCB) . . . . .	279
10.4.3.1	Features . . . . .	281
10.4.4	CAPTIVATE-BSWP . . . . .	282
10.4.5	CAPTIVATE-PHONE . . . . .	286
10.4.6	CAPTIVATE-PROXIMITY . . . . .	292
10.4.7	48-pin Male Sensor PCB Connector Information . . . . .	293
10.5	Using the HID Bridge . . . . .	295
10.5.1	HID Bridge Interfaces . . . . .	295

---

10.5.2	HID Bridge Configuration Command Set . . . . .	296
10.5.3	HID Bridge Modes of Operation . . . . .	297
10.5.3.1	Operating Mode: Packet Mode . . . . .	297
10.5.3.2	Operating Mode: Raw Mode . . . . .	298
<b>11</b>	<b>Workshop (Getting Started)</b>	<b>299</b>
11.1	Hardware Setup . . . . .	301
11.2	Important note about wake on proximity behavior . . . . .	302
11.2.1	Out of Box Experience . . . . .	305
11.3	Creating a new sensor design project . . . . .	309
11.3.1	LAB #1 . . . . .	309
11.3.2	Part 1 - Creating a New Design . . . . .	309
11.3.3	Part 2 - Tuning Sensors . . . . .	319
11.3.4	Part 3 - Generate Updated Sensor Configuration . . . . .	330
11.4	Experiments with Low Power . . . . .	331
11.4.1	LAB #2 . . . . .	331
11.4.2	Part 1 - Measure Proximity Sensor Low Power current . . . . .	335
11.4.3	Part 2 - Effects of Sensor Sensitivity on Low Power current . . . . .	335
11.4.4	Part 3 - Effects of Sensor Scan Rate on Low Power current . . . . .	336
11.5	Exploring the CapTlivate™ Touch Library . . . . .	337
11.5.1	LAB #3 . . . . .	337
11.5.2	Part 1 - CapTlivate™ Touch Library Overview . . . . .	339
11.5.3	Part 2 - Explore CCS project structure and details . . . . .	340
11.5.4	Part 3 - Creating and using callbacks . . . . .	340
11.5.5	Part 4 - Determining sensor state and position . . . . .	342
<b>12</b>	<b>FAQ</b>	<b>344</b>
12.1	Design Kit . . . . .	344
12.1.1	Does the kit arrive pre-programmed? . . . . .	344
12.1.2	How do I verify that the EVM is working? . . . . .	344
12.1.3	How can I increase the sensitivity or range of the proximity sensor in the out-of-box experience demo? . . . . .	344
12.2	CapTlivate™ Technology Design Center . . . . .	345
12.2.1	Can I download code from the Design Center? . . . . .	345
12.2.2	What does the 'No connected HID devices' pop-up window mean? . . . . .	345
12.2.3	Why is the target data display slow? . . . . .	345
12.2.4	When do I need to generate a project from the design center and load onto the target? . . . . .	345
12.2.5	What does updating a project from the design center do? . . . . .	345
12.2.6	Where can I find the example demo project source code files? . . . . .	345
12.2.7	Why is the CapTlivate™ Design Center window not visible when I launch the GUI? . . . . .	345
12.3	CapTlivate™ Technology . . . . .	346

---

12.3.1 Why does sensitivity decrease as I increase the Conversion Gain? . . . . .	346
12.3.2 Why is the scan rate slower than I specify? . . . . .	346
<b>13 Glossary</b>	<b>347</b>
13.1 Automatic Power Down . . . . .	347
13.2 Wake On Proximity Mode . . . . .	347
13.3 Bias Current . . . . .	348
13.4 Button Group Sensor . . . . .	348
13.5 Target Communications Configuration . . . . .	348
13.6 Channel . . . . .	349
13.7 Conversion Count . . . . .	349
13.8 Conversion Gain . . . . .	351
13.9 Count . . . . .	353
13.10Count Filter . . . . .	353
13.11Debounce . . . . .	355
13.12Delta . . . . .	357
13.13Desired Resolution . . . . .	358
13.14Electrode . . . . .	358
13.15Element . . . . .	358
13.16Engineering Parameters . . . . .	358
13.17Error Threshold . . . . .	359
13.18Frequency Divider . . . . .	360
13.19Idle State . . . . .	361
13.20LTA Filter . . . . .	363
13.21Modulation Enable . . . . .	366
13.22Mutual Capacitance . . . . .	366
13.23Negative Touch Threshold . . . . .	366
13.24Noise Immunity . . . . .	368
13.25Parallel Sense Block . . . . .	368
13.26Phase Lengths . . . . .	369
13.27Position Filter . . . . .	370
13.28Proximity Sensor . . . . .	371
13.29Proximity Threshold . . . . .	371
13.30Run Time Recalibration . . . . .	373
13.31Sample Capacitor Discharge . . . . .	374
13.32Self Capacitance . . . . .	374
13.33Sensor Port . . . . .	375
13.34Sensor Timeout Threshold . . . . .	375
13.35Slider Sensor . . . . .	375
13.36Slider Trim . . . . .	375

---

13.37	Sync Parameters	377
13.38	System Report Rate	377
13.39	Time Cycle	378
13.40	Time Estimation	378
13.41	Touch Threshold	379
13.42	Trace	381
13.43	Wake On Touch	382
13.44	Wheel Sensor	382
<b>14</b>	<b>Disclaimer</b>	<b>383</b>

# **Chapter 1**

## **Main Page**

Copyright © 2015-2016 Texas Instruments Incorporated. All rights reserved.

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments  
13532 N. Central Expressway  
Dallas, TX 75243  
[www.ti.com](http://www.ti.com)



## Chapter 2

# Getting Started

### Welcome

Welcome to the CapTlve™ Technology Guide!

CapTlve™ is a capacitive user interface design ecosystem that takes the next big step in design process evolution. It brings together a capacitive measurement technology, a design GUI, a capacitive touch software library, and a hardware development platform. This document is the single source of information about how to use that ecosystem.

### Starting Out

If you are a beginner at capacitive sensing, start by learning about [capacitive sensing basics](#). If you have a CapTlve™ Development Kit, you can quickly get started with hardware by going through the [out-of-box experience](#). Then, you'll want to check out the [technology](#) chapter, which introduces the main principles behind the CapTlve™ technology.

### Becoming an Expert

Once you've got the basics down, become a CapTlve™ expert by completing workshops for [creating a new sensor design](#), [experimenting with ultra-low-power](#), and [using the CapTlve™ Touch Library](#). Then explore the other sensor panels that come with the [MSP-CAPT-FR2633 Development Kit](#).

### Kicking off a Design

If you are going to be starting a new design, be sure to visit the [design](#) chapter, which will guide you through making design decisions and tradeoffs. You can then reference the [CapTlve™ Design Center quick start guide](#) for help starting a new project. The [CCS/IAR project quick start guide](#) explains how to get firmware projects up and running in a development environment such as TI's Code Composer Studio or IAR Embedded Workbench.

### Using this Guide

If you would like more information about this guide, the [introduction chapter](#) provides a brief introduction to every chapter.

# Chapter 3

## Introduction

Each chapter in the CapTlivate™ Technology Guide is introduced briefly below. Check out the [getting started](#) section if you are new to CapTlivate™.

### Capacitive Sensing Basics

The [Capacitive Sensing Basics](#) chapter provides a fundamental overview of self and mutual capacitance.

### Technology

The [Technology](#) chapter describes the operating principles of the CapTlivate™ peripheral.

### Design Guide

The [Design Guide](#) chapter introduces the flow for beginning a new design, and walks through the trade-offs of designing for different requirements, such as low power, noise immunity, or moisture.

### CapTlivate™ Design Center

Read the [Design Center GUI](#) chapter to get started with the rapid development tool that accelerates capacitive touch designs for CapTlivate™ Technology enabled MSP devices. By helping guide the product developer through the capacitive touch development process, the CapTlivate™ Design Center can simplify and accelerate any touch design through the use of innovative user graphical interfaces, wizards and controls.

### Device Family

The [Device](#) chapter introduces the MSP MCUs that have the CapTlivate™ technology.

### Software Touch Library

The [CapTlivate™ Capacitive Touch Software Library](#) is a comprehensive collection of functions providing touch, communications and sensor management features. Touch functions range from advanced sensor processing for buttons, sliders and wheels to "bare-metal" functions that provide direct access to the CapTlivate™ peripheral.

---

## MCU Development Kit

The [MSP-CAPT-FR2633 MCU Development Kit](#) is an easy to use evaluation platform for the MSP430FR2633 MCU. It contains everything needed to start developing on this MCU platform, including on-board emulation for programming, debugging and EnergyTrace™ measurements.

## Workshop

The [workshop](#) is based around the [MSP-CAPT-FR2633 MCU Development Kit](#), and contains the out-of-box experience as well as several labs to introduce the technology hands-on.

## FAQ

The [FAQ](#) contains answers to the questions that get asked the most.

## For More Information

For more information on the CapTIvate™ technology, software, and available evaluation kits, see the [CapTIvate™ Design Center download page](#). For technical support, E-mail us directly at [captivate@list.ti.com](mailto:captivate@list.ti.com)

## Chapter 4

# Capacitive Sensing Basics

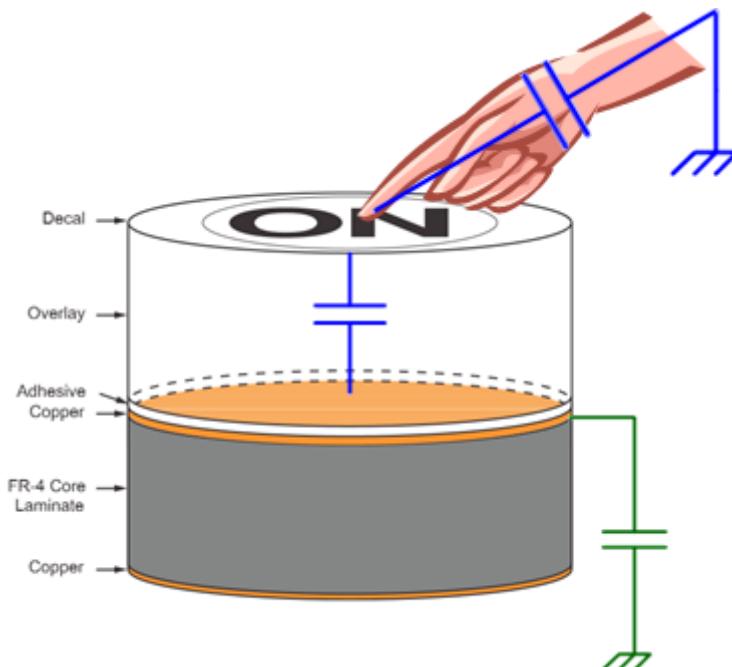
This chapter provides a fundamental overview of self and mutual capacitance and capacitive sensing techniques.

Capacitive sensing is performing a measurement to detect a capacitive change to a sensor element. A sensor element can be any conductive material (copper PCB plane, a wire, etc.) The change can be due to human interaction, such as a finger, ear, or hand. This is often termed "capacitive touch" or "proximity." However, capacitive sensing is not limited to human interaction. Other objects or materials can change the capacitance of a sensor. This could be an organic or inorganic material. Think metals or liquids. In either case, the interpretation of the change in capacitance is what defines an application.

Capacitance is a measure of an object's ability to store electric charge. Any two conductive materials that are separated by an insulator will exhibit capacitance, if they are close enough together.

A similar system can be created in which the second capacitor is actually a person interacting with the system

1. The copper electrode is constructed on FR4 PCB material, and has a free-space coupling capacitance to earth ground.
2. A user touches the top of the overlay. The overlay is an insulator, and acts as a dielectric between the user and the copper electrode. Additional capacitance is added to the electrode.



## 4.1 Self Capacitance

This method of measuring changes in capacitance with respect to earth ground is commonly referred to as self capacitance measurement. Sometimes it is also referred to as surface capacitance. In a parallel-plate model, the electrode defines one plate of the capacitor, with the other plate being ground (for  $C_{\text{electrode}}$ ) or the user's finger (for  $C_{\text{touch}}$ ). A touch causes the capacitance of the electrode to increase. (Typically 1-10pF)

When describing the various capacitances found in a capacitive touch solution, an equivalent circuit model can be helpful in visualizing the source of the different capacitances as well as the effect of each capacitance. Figure 2 is an example of an equivalent circuit for a single self-capacitance button.

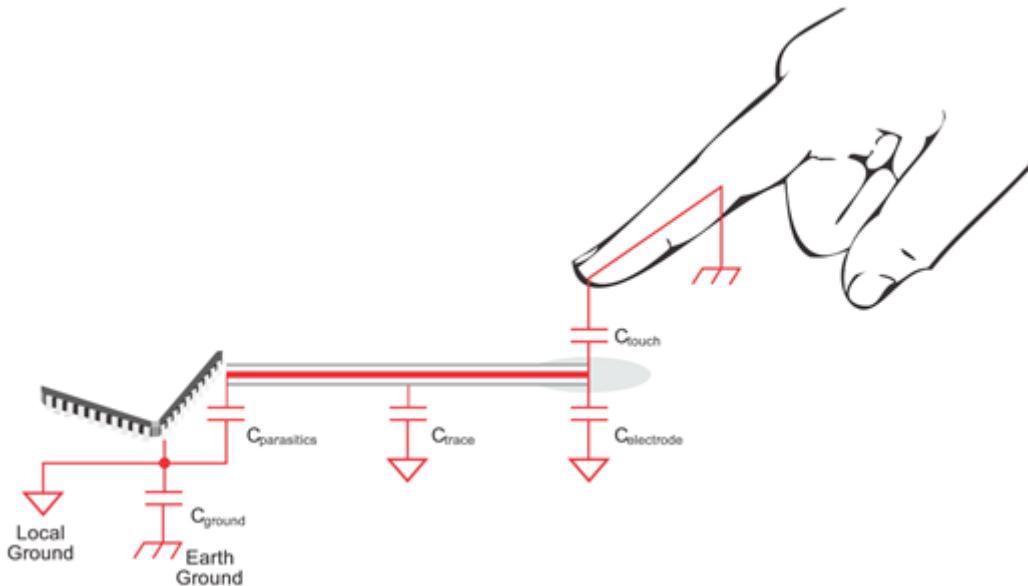


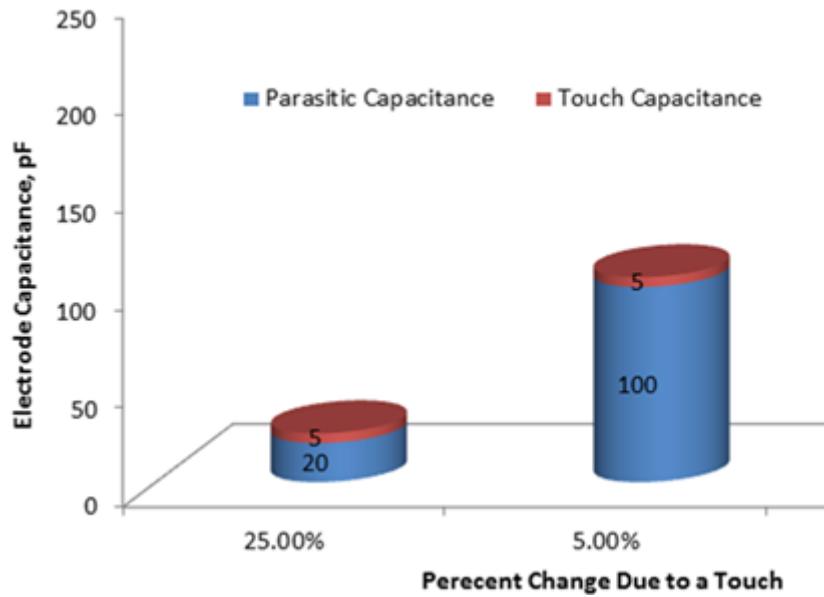
Figure 4.1: Equivalent Circuit

Five different capacitances are shown in figure below.  $C_{\text{ground}}$  is the capacitance between the local device under test (DUT) ground and earth ground. In some applications, local and earth ground are connected when the DUT uses mains power, but typically the local ground is capacitive coupled back to earth ground.  $C_{\text{trace}}$  and  $C_{\text{electrode}}$  is the capacitance between the trace and electrode structures back to the local ground. This capacitance is most directly affected by surrounding structures, typically ground pours, that are either on the same layer or on adjacent layers. Not shown is the capacitance between the trace and electrode structures and earth ground. These capacitances are not without merit; however, for simplicity and in the context of this document, the design guidelines are given with the principle of affecting the local capacitance (that is, separation between local ground and the traces and electrodes). The capacitance  $C_{\text{parasitics}}$  is a combination of the internal parasitic capacitance of the microcontroller and any components within the circuit. This capacitance is also referenced to local ground. The touch capacitance,  $C_{\text{touch}}$ , is the parallel plate capacitance formed between the touch interaction and the electrode. In the example of a touch, as the finger presses against the overlay, the flattened surface of the finger forms the upper plate and the electrode forms the lower plate. The capacitance is a function of the area of the two plates, the distance between them, and the dielectric of the material that separates them.

### 4.1.1 Self Capacitance Parasitics

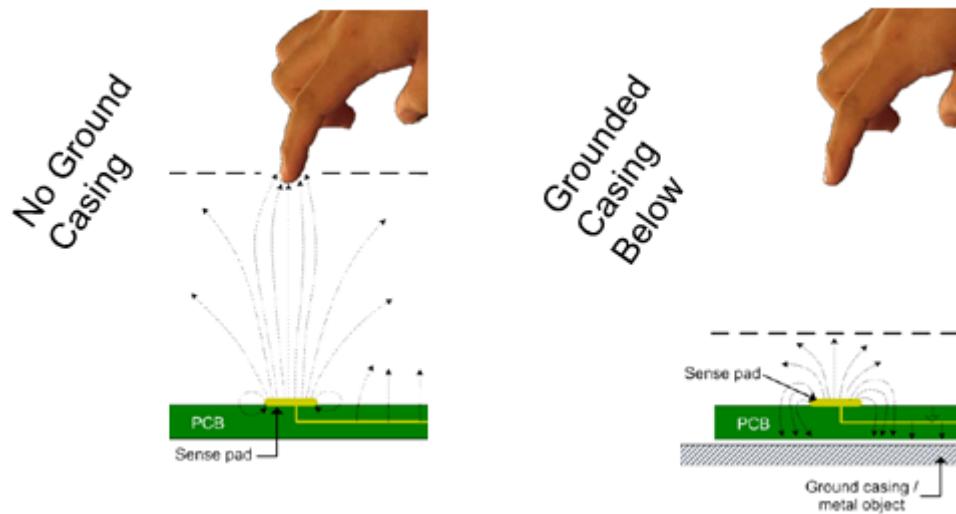
In addition to any PCB traces, wires, connectors, etc., there will be some parasitic capacitances to ground as well as to earth ground. Parasitic capacitances have the effect of decreasing the effect of a user's touch in the system. This is a result of the fact that we are measuring change in capacitance.

- If we assume a change in capacitance due to a touch of 5pF, then we get a 25% increase in capacitance if our parasitic capacitance is only 20pF.
- If our parasitic capacitance is 100pF, then a touch only causes a 5% increase in capacitance. As a result, the change is more difficult to measure.



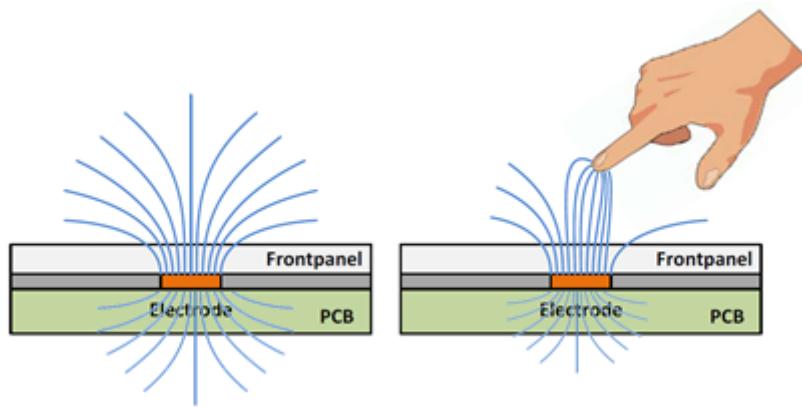
#### 4.1.2 Parasitics and the E-Field

The effect of parasitic capacitance goes beyond just a reduction in percent change. From a physics perspective, having ground structures in close proximity of the electrode will cause the E-field lines projected out from the electrode to concentrate between the electrode and ground, rather than penetrating up through the overlay into the area of interaction.

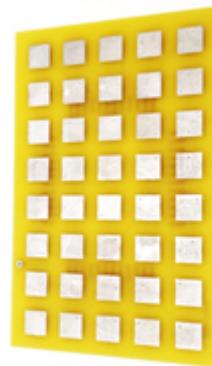


#### 4.1.3 E-Field Propagation

Self capacitance electrodes project E-field lines out 360 degrees from the electrode. This means that the electrode can be interacted with from the bottom side of a PCB as well as through the overlay material. Ground can be used as a shielding mechanism, as can a voltage-follower driven shield.



The 360 degree sphere projected in a self-capacitance system also places a limitation on how dense buttons can be packed next to each other. If the keys are placed too close together, it will be very easy for the user to incorrectly trigger adjacent keys. The limit on how close keys can be to each other is usually dependent upon the size of the keys and the thickness of the overlay material.



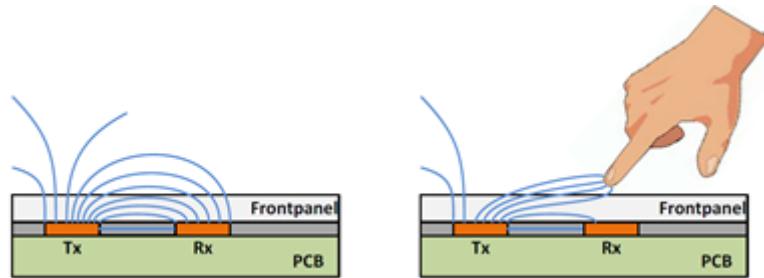
#### 4.1.4 Self Capacitance Summary

- Self capacitance is a capacitive sensing measurement method in which a single electrode's capacitance with respect to earth ground is measured.
- A typical user's touch adds between 1pF to 10pF to the electrode.
- Parasitic capacitances to ground have the effect of de-sensitizing a user's touch.
- Self capacitance electrodes project E-field lines out 360 degrees, and can be interacted with on both sides (unless ground shielding is utilized).

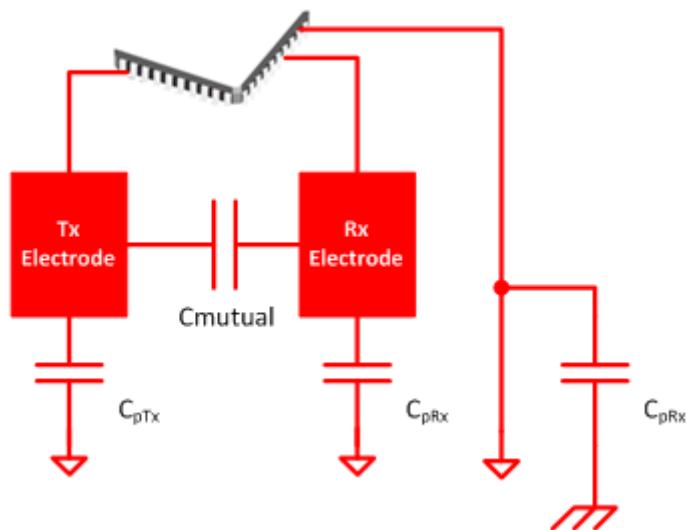
## 4.2 Mutual Capacitance

Mutual capacitance involves measuring a change in capacitance just like self-capacitance, with one big difference: we define both plates of the capacitor, instead of utilizing earth ground as the second plate. Note: Mutual capacitance may also be referred to as projected capacitance.

Mutual capacitance electrodes actually consist of two separate electrode structures, and they require two pins from the microcontroller- a transmit electrode, and a receive electrode. When a user touches an area on the panel where a Tx meets an Rx, the mutual capacitance between those Tx and Rx electrodes is reduced. This is because the user's interaction has the effect of disturbing the electric field propagation between the two electrodes. Users are coupled to earth ground, and the human body is a conductor. Placing a finger in between two mutual capacitance electrodes has roughly the same effect as placing ground between them- it reduces electric field



coupling between them, which reduces the capacitance. Typical changes in mutual capacitance due to a touch are small- usually less than 1pF.



#### 4.2.1 Mutual Capacitance Parasitics

Just like in self capacitance systems, there are still parasitic capacitances! There are now two kinds of parasitic capacitance, mutual and ground.

- Parasitic mutual capacitance. Anywhere that a Tx trace comes near an Rx trace, the two will have a parasitic mutual capacitance. This is similar to parasitic ground capacitance in self cap.
- Parasitics to ground ( $C_{pTx}$ ,  $C_{pRx}$ ). In a mutual capacitance network, the Tx and Rx electrodes will still have capacitance to circuit ground and earth ground. The difference is that now that parasitic capacitance does not affect our measurement. It does, however, still affect our ability to drive the electrodes at high speeds - since we are charging and discharging that capacitance.

#### 4.2.2 Tightly Coupled E-Fields

Because mutual capacitance electrodes are defined by two conductors (both plates of the capacitor), the electric field that a user can interact with will be tightly defined between the two conductors. This is a drastic difference from self-cap, where the field of interaction is projected outward in all directions. This can have many design benefits. Keypads can have closely grouped keys without worry of cross-coupling when a user isn't centered directly on one key. It's also easier to route guard channels and proximity sensors between keys.

### 4.2.3 Overlay Material Requirement

Overlay material is required for mutual capacitance touch panels. It provides an area for the electric field between Rx and Tx electrodes to propagate where a user can interact with them. Without an overlay, making contact with mutual capacitance electrodes shorts the Rx and Tx together, and also adds significant parasitic capacitance to ground.

Since the overlay serves the purpose of projecting out the mutual E-field, it is also important that mutual capacitance electrodes be shaped according to the thickness of the overlay material in order to optimize sensitivity. If the Rx and Tx electrodes are too close together on a design with a thick overlay, very little field will be penetrating up to the top of the overlay.



### 4.2.4 Mutual Capacitance Sensitivity

Because the change due to a touch on a mutual-cap electrode is smaller than that of a self-cap electrode, the measurements tend to be noisier than self capacitance measurements. It is difficult to construct large sliders and wheels from mutual capacitance electrodes, unless many electrodes are used. As a result, mutual capacitance electrodes lend themselves best to button matrices and more advanced sensors that have small node sizes but high electrode density.

It's important to remember that we are subtracting capacitance in a mutual-cap system. This means that we need to have some mutual capacitance to start with. However, we want that mutual capacitance to be in one specific place: the area of user interaction.

### 4.2.5 Mutual Capacitance Summary

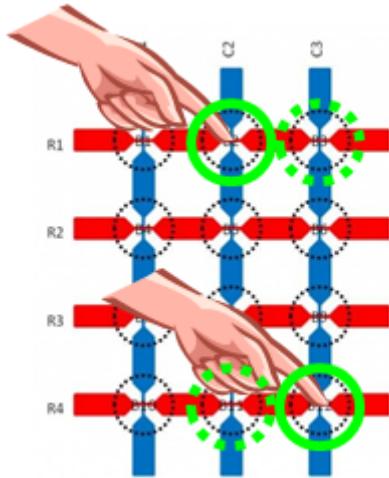
- Mutual capacitance is a capacitive sensing method where changes in the capacitance between two electrodes are measured.
- A user's touch disrupts the field between the two electrodes, reducing the coupling between them and removing mutual capacitance.
- A typical user's touch adds or removes less than 1pF of mutual capacitance from the electrode.
- Parasitic capacitances to ground do not affect the measurement, but do affect our ability to drive the electrodes.
- Mutual capacitance E-fields are tightly coupled between the Tx and Rx electrodes, allowing high-density panels.

## 4.3 True Matrix Capability

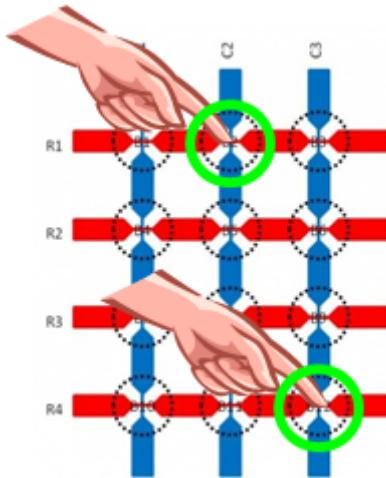
One method of creating a large keypad from a small number of pins is through the use of a electrode matrix. A matrix can be realized with self capacitance as well. Let's look at an example. The diagram to the right shows a capacitive touch matrix consisting of 4 row electrodes and 3 column electrodes. This allows for 12 keys to be derived from 7 pins. In a self capacitance system, each row and each column is scanned as an independent

---

electrode. Individual keys are derived from aggregating the data from row and column electrodes to determine the position of a touch. Drawback: In a self capacitance matrix, multi-touch is not achievable due to ghosting effects.



In a mutual capacitance approach, the columns could be treated as Tx electrodes, and the rows could be treated as Rx electrodes. Every row/column intersection point is then a unique Tx/Rx combination, and a unique mutual capacitance. Since each node is a unique capacitance that is being measured, full multi-touch is possible.



Because each Rx/Tx intersection is a unique node, a 16 pin device could support up to 64 individual buttons in an 8 X 8 matrix configuration. Only 16 pins would need to be routed on the panel, which is very achievable on a two-layer PCB. An equivalent self capacitance design would require 64 pins to realize the same panel. Having distinct nodes also improves the reliability of the system, as each node (Rx/Tx intersection) is tracked separately in software (as opposed to aggregating row and column measurements to calculate the likely key press, as would be required with self capacitance).

## Chapter 5

# Technology

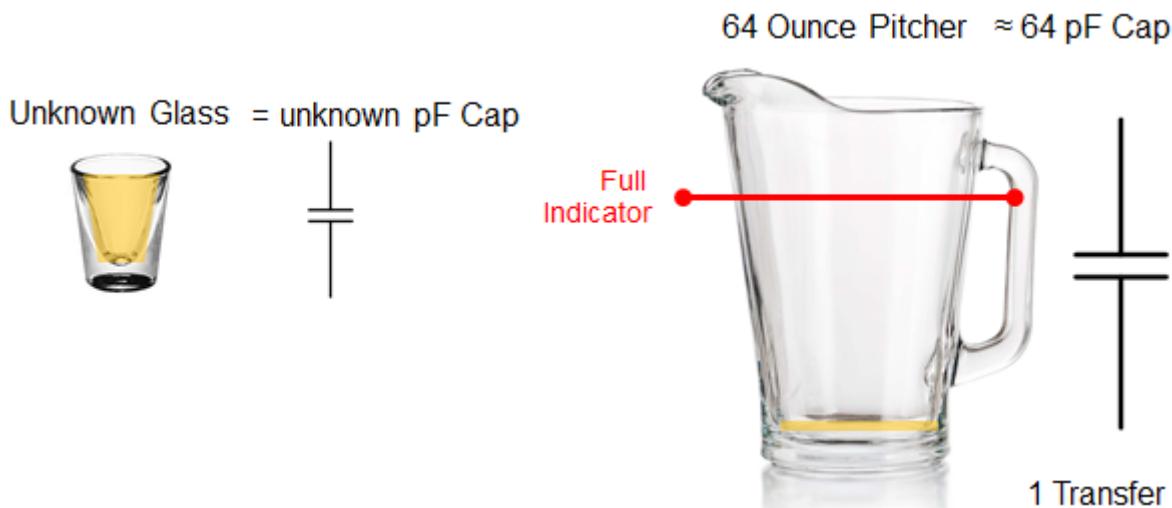
Welcome to the technology chapter. This chapter is a detailed overview of the CapTlve™ peripheral. It covers the principles of measurement and the measurement signal chain that makes everything possible. In addition, the low power features are discussed in conjunction with the wake-on-proximity state machine.

This section describes the CapTlve™ peripheral and the intended use. This section is broken into analog and digital sections. The analog section describes the measurement technology while the digital section describes the features surrounding the technology to support sensing applications. Throughout this section the peripheral inputs and outputs are described in the context of the CapTlve™ library structures.

Before discussing the peripheral a short introduction to charge-transfer as a capacitance measurement technology is presented.

### 5.1 Charge Transfer Technology

Charge transfer is an effective way to measure a change in capacitance based upon a fixed capacitance. By means of simple analogy, charge and capacitance are represented by a liquid and a container, as shown in Figure 1. The smaller container is the variable capacitance while the larger container is the fixed capacitance.



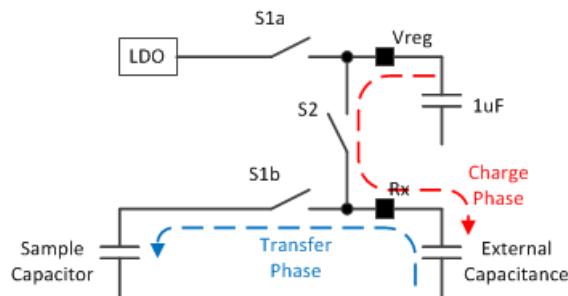
The smaller container is filled (charged) and then emptied (transferred) into the larger container. The number of times it takes to fill the larger container is representative of the volume (capacitance) of the smaller container. If the number of times it takes to fill the larger container changes, then the volume of the smaller container has changed. In most capacitive touch systems, the interest is not in the absolute capacitance but in the change in capacitance. That is when a touch or other interaction occurs, the capacitance of the smaller container changes and consequently the number of times it takes to charge and empty the smaller capacitance into the larger changes. It

is this change that is used to determine if a touch occurred.

The CapTlivate™technology allows for two different types of external capacitance to be measured. These two types are called self and mutual capacitance and are described in the next two sections.

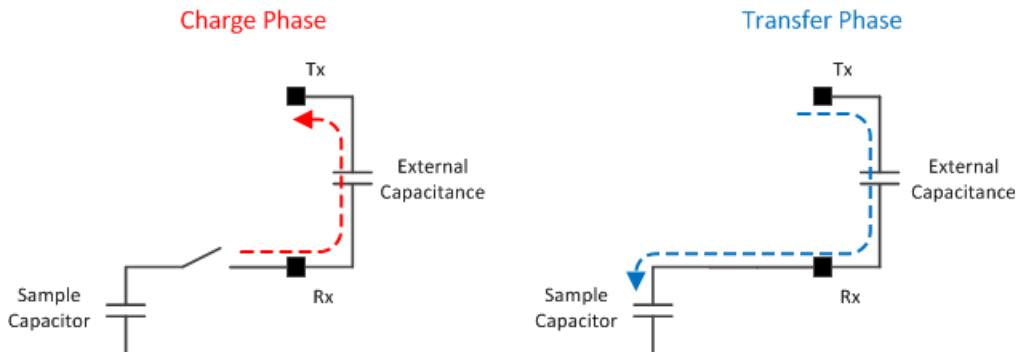
### 5.1.1 Charge and Transfer Phases in Self Mode

Self capacitance, as explained in the Capacitive Sensing Basics section, is the capacitance relative to earth ground. In order to measure the self capacitance with the CapTlivate™charge transfer implementation, charge is transferred between three difference capacitors, as shown in Figure 2. First, the charge stored on the Vreg capacitor (recommended value of 1uF), is used to charge the external unknown capacitance during the charge phase. Second, the charge from the external capacitance is transferred to an internal sampling capacitor. During this transfer phase when charge is moved from the external capacitor to the sample capacitor the Vreg capacitor is refilled with charge by the LDO. These charge and transfer phases are repeated until the voltage on the internal sampling capacitor changes by the desired amount. As will be discussed in a later chapter, this voltage can be changed to allow for a wide range of external capacitances.



### 5.1.2 Mutual Mode Charge and Transfer Phases

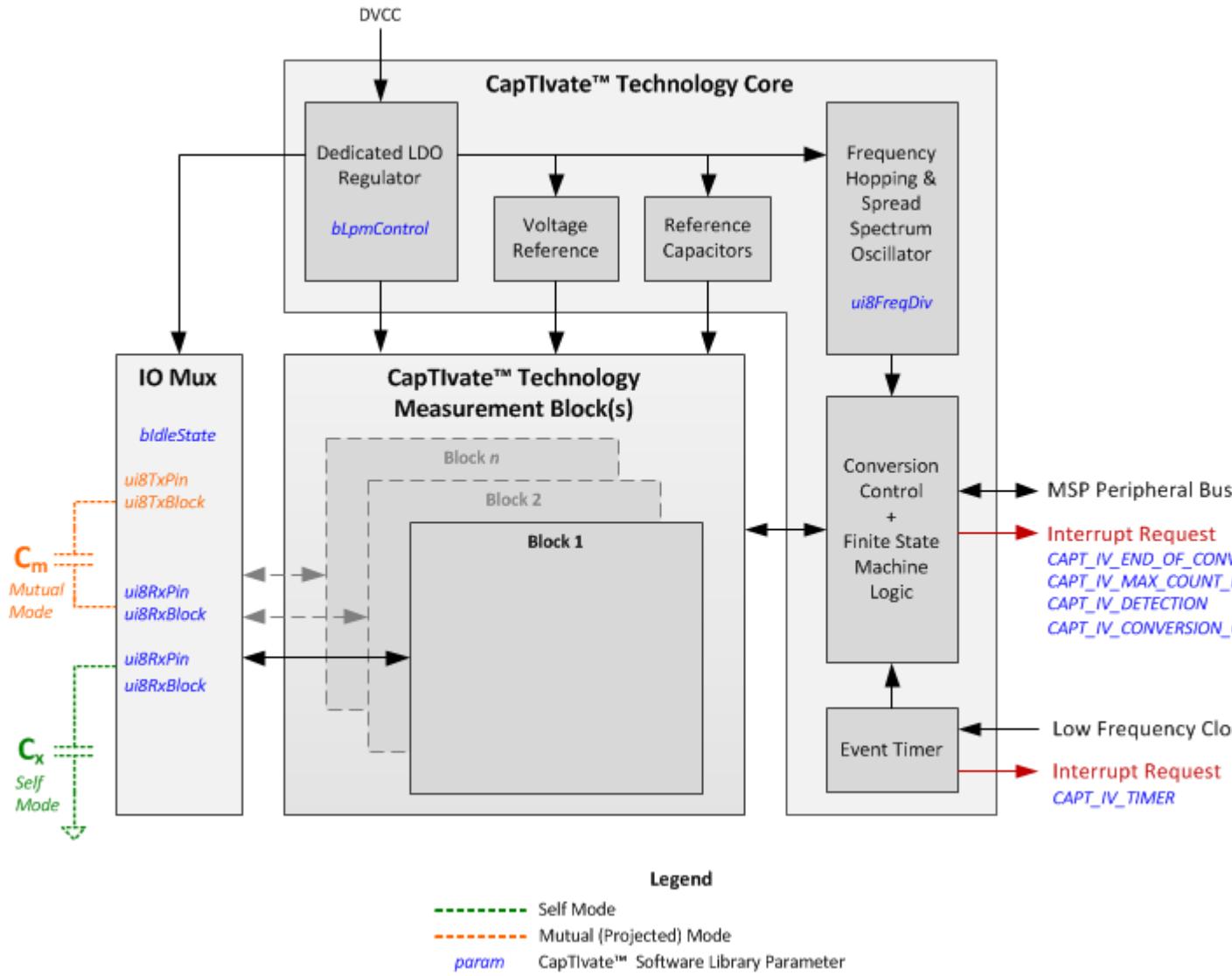
Mutual capacitive mode is the measurement of capacitance between two electrodes; typically a receive (Rx) electrode and a transmit (Tx) electrode. During the charge phase a sample-and-hold circuit drives the Rx node while the Tx node is pulled to ground. In the transfer phase the Tx node is pulled high (driven by Vreg similar to the charge state in self capacitive mode) and the charge across the capacitance is transferred to the sample capacitor.



## 5.2 The CapTlivate™ Peripheral

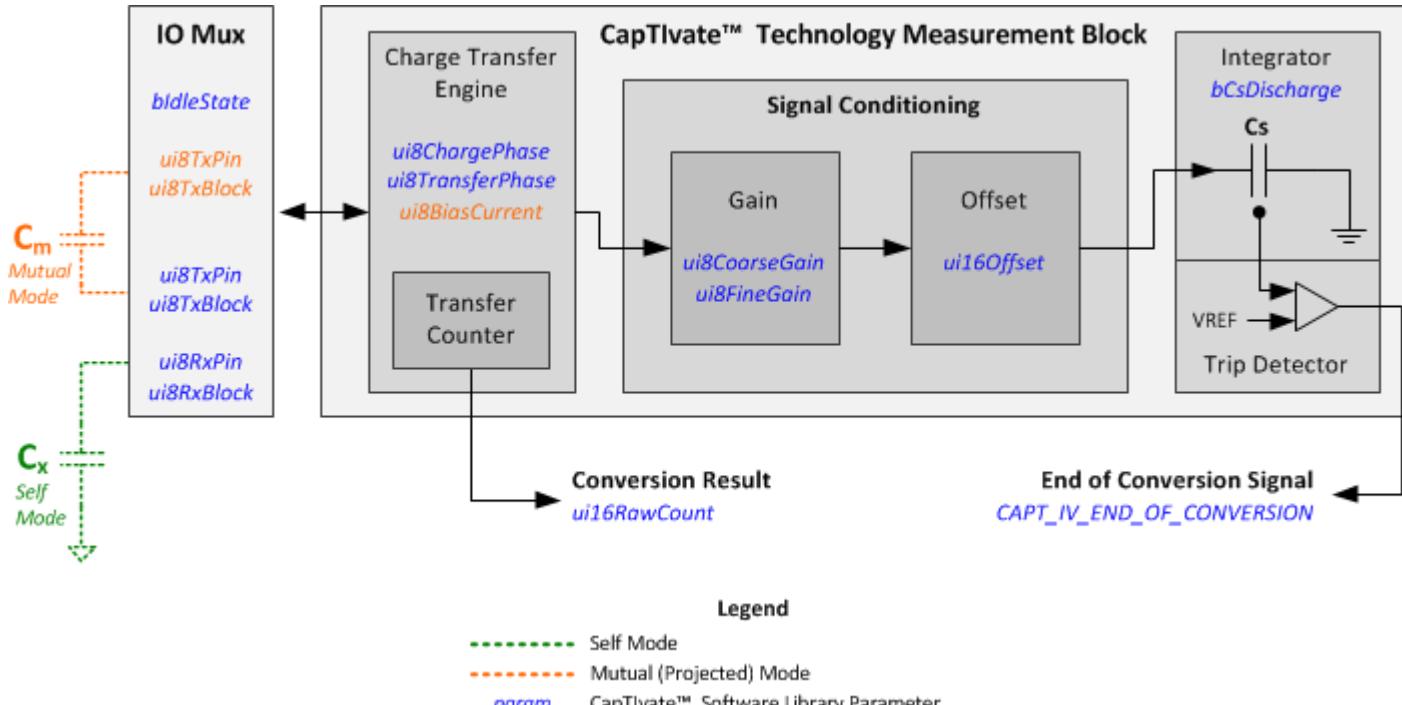
The CapTlivate™peripheral can be described in terms of two main themes or functions: the capacitance measurement and the complimentary digital functions. In Figure 4, the capacitance measurement function is illustrated in the CapTlivate™Measurement blocks and IO mux while the complimentary digital functions are illustrated by the CapTlivate™Core. Depending upon the device configuration, there can be up to 12 measurement blocks and up to 8 measurement inputs per block. The number of blocks will determine the number of measurements that can be made in parallel while the number of pins per block will determine the actual number of

IO available for a capacitive touch measurement. So for example, a device with 4 blocks and 4 pins per block can measure 4 elements in parallel and provides 16 CapTlivate™IO. The complimentary digital functions are not directly related to the capacitance touch measurement but enable low power operation, electromagnetic compatibility (EMC) control, and other features.



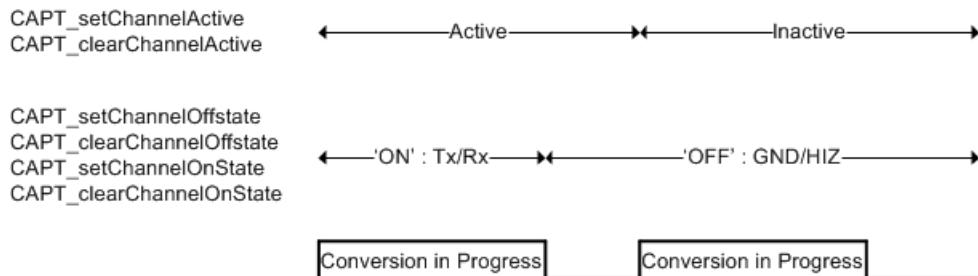
### 5.2.1 Capacitance Measurement

Figure 5 shows the IO Mux control and one instantiation of the CapTlivate™ Measurement Block. The IO mux performs the necessary switching to support mutual and self capacitive mode measurements as well as configuring unused pins as either floating (High-Z) or active low (GND). Within the CapTlivate™ measurement block the technology can be further divided down into sub-blocks to provide a more complete picture and understanding behind the library parameters.



#### 5.2.1.1 IO Mux

As with most analog peripherals, the IO control can be switched over from the GPIO to the analog functionality by enabling the CapTlivate™function for that IO (refer to the specific datasheet for a description of the IO). In addition to enabling the IO, the CapTlivate™functionality must be defined when the IO is active, when the IO is on (receiver or transmitter), and when the IO is off (ground or high-z). Figure 6 shows the control that is applied to an enabled channel with respect to the conversion.



The `CAPT_initUI()` API found in `CAPT_Manager.h` applies the appropriate settings to the IO based upon the sensor settings `tSensor.bldleState` and the element definitions `tElement.ui8RxBlock`, `tElement.ui8RxPin`, `tElement.ui8TxBlock`, and `tElement.ui8TxPin`. It is recommended to use these settings provided through the design center, although additional APIs (like the ones in Figure 6) are provided for custom applications and more unique IO control.

The CapTlivate™Design Center provides the ability to easily configure and develop simple self capacitive mode and mutual capacitive mode structures: self capacitive mode consists of one Rx channel while mutual capacitive mode consists of one Rx and one Tx channel. The modularity of the APIs allows for multiple Tx in the case of mutual capacitive mode or multiple Rx in self capacitive mode. This provides a unique way to unify multiple structures into a single new element. Typically this new element is used as a wake-up mechanism, before transitioning to more traditional separate measurements.

The APIs shown in Figure 6 as well as the `CAPT_forceSensorIO` API enable a more custom implementation of the IO. Use of these functions, should be done with caution and always in the context of a sensor definition. These APIs do not explicitly configure the mode, ie self or mutual capacitive mode, and therefore it is always important to utilize these parameters within the context of having applied the sensor parameters (`CAPT_applySensorParams`).

---

## Charge Transfer Engine

As previously discussed, the CapTlivate™technology uses charge transfer to measure relative changes in capacitance. Maintain consistency in measurements is important so that the only change being introduced in the system is the change in capacitance due to a touch or proximity effect. One potential cause for an inconsistent measurement is an incomplete charge and/or transfer phase. This is directly related to the size of the capacitive load being driven. The larger the capacitance, the longer it takes to fully charge and discharge that capacitance. If the charge or transfer phase is too short for the given capacitive load there is a potential for any deviation in the timing to result in a change in the amount of charge being placed on or transferred from the capacitance being measured. Given the exponential nature of the charge and discharge waveforms it is important to control the charge and transfer phase lengths. The control of the charge and transfer phases, and ultimately the conversion rate, is provided in the configuration structure and also can be controlled via the CapTlivate™Design Center.

As shown in the Capacitive Sensing Basics section, in self capacitive mode the parasitic capacitance is in parallel with the electrode capacitance being measured and therefore the charge and phase lengths are adjusted to ensure complete charge and transfer of charge to all capacitances. In mutual capacitive mode; however, there is a distinction between the parasitics to ground and the parasitics associated with the Rx and Tx lines which are in parallel with the mutual capacitance between the Rx and Tx electrodes actually being measured. The phase lengths are adjusted to compensate for the mutual capacitance between the Rx and Tx electrodes and any mutual parasitic capacitance between Rx and Tx lines (this can occur when Rx and Tx are in close proximity to each other when routed on the PCB). This control does not compensate for the parasitic capacitances to ground (primarily on the Rx line) which can have a negative impact on the measurement. In order to compensate for this capacitance an additional control mechanism is in place to strengthen the sample-and-hold amplifier when the parasitic capacitance to ground is large. Similar to the phase control, this compensation is also found in the CapTlivate™ Design Center and the sensor configuration of the CapTlivate™Touch Library. Table 1 shows the applicable charge transfer control mechanisms and the associated CapTlivate™Design Center tuning parameter and CapTlivate™Touch library structure parameter name. These parameters are applied to the peripheral via the HAL API CAPT\_applySensorParams.

Mode	CapTlivate Design Center	Configuration Parameter
Self and Mutual	Charge_Hold_Phase_Length	tSensor.ui8ChargeLength
	Transfer_Sample_Phase_length	tSensor.ui8TransferLength
	Frequency Divider	tSensor.ui8FreqDiv
Mutual Only	Bias Current	tSensor.ui8BiasControl

Also shown in Table 1 is the frequency divider. The charge and transfer phase lengths are a function of the divided CapTlivate™oscillator. Therefore it is possible to increase the phase lengths with both the frequency divider and the individual control of the charge and transfer phase lengths.

## Signal Conditioning (Sensitivity Related Functions)

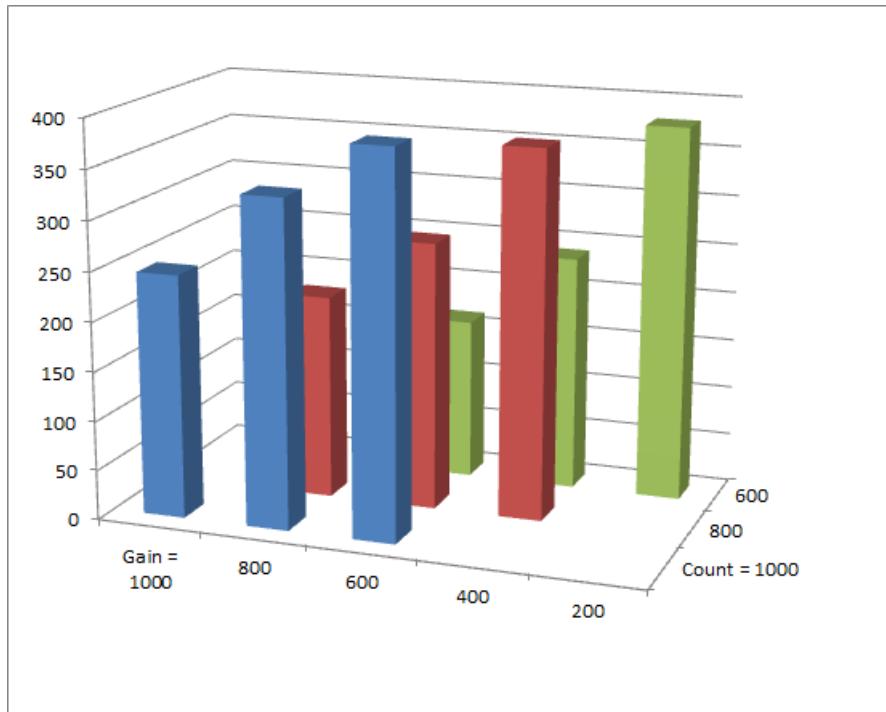
Once reliable charge transfers have been established the main issue can be addressed: discerning (small) changes in capacitance. This issue becomes much more challenging as the relative change in capacitance is small when compared to the capacitance being measured. In order to provide sensitivity over a large range of capacitances the CapTlivate™peripheral provides gain and offset functions. These functions are described as the tElement.ui8CoarseGain, tElement.ui8FineGain, and tElement.ui16Offset which are all unique to each element definition in the CapTlivate™ library. These components are automatically tuned for each element in order to achieve the conversion gain and conversion count defined within the sensor.

The conversion count, firstly, describes the conversion time. This number is the number of charge transfers that occur for a non-touch measurement. For example if the charge and transfers phases are each 250ns, then a conversion count of 200 counts would result in a conversion time of ~100us. Secondly, the conversion time is directly proportional to the sensitivity of the system. As the conversion count is increased the relative change in the conversion result, due to touch (increase in capacitance), will increase.

The conversion gain is always less than or equal to the conversion count. The difference between the gain and the count is proportional to the sensitivity of the system and is directly related to the offset compensation. When the conversion gain is equal to the conversion count, no offset compensation is applied and the sensitivity is solely related to the gain. As the conversion gain decreases the amount of offset compensation is increased. By

increasing the offset compensation, although the absolute change in capacitance is the same, the relative change in capacitance increases.

Figure 7 shows the response of the system to an increase in capacitance ( $\sim 0.1\text{pF}$ ) and how this response can be amplified by changing the conversion gain and conversion count settings. The internal reference capacitor is used to generate the change in capacitance: moving from setting 3 (5.0pF typical) to setting 5 (5.1pF typical).



Mode	CapTlivate Design Center	Configuration Parameter
Self and Mutual	Conversion Count	tSensor.ui16ConversionCount
	Conversion Gain	tSensor.ui16ConversionGain
	Sample Capacitor Discharge	tSensor.bCsDischarge

Table 2 shows the Conversion Count and Gain parameters already discussed and a third parameter, bCsDischarge, which can be used to extend the sensitivity over a greater range of capacitance by setting the value to 0 or 0V. The default setting is 1, or 0.5V, and recommended for most applications.

## 5.2.2 Auxillary Digital Functions

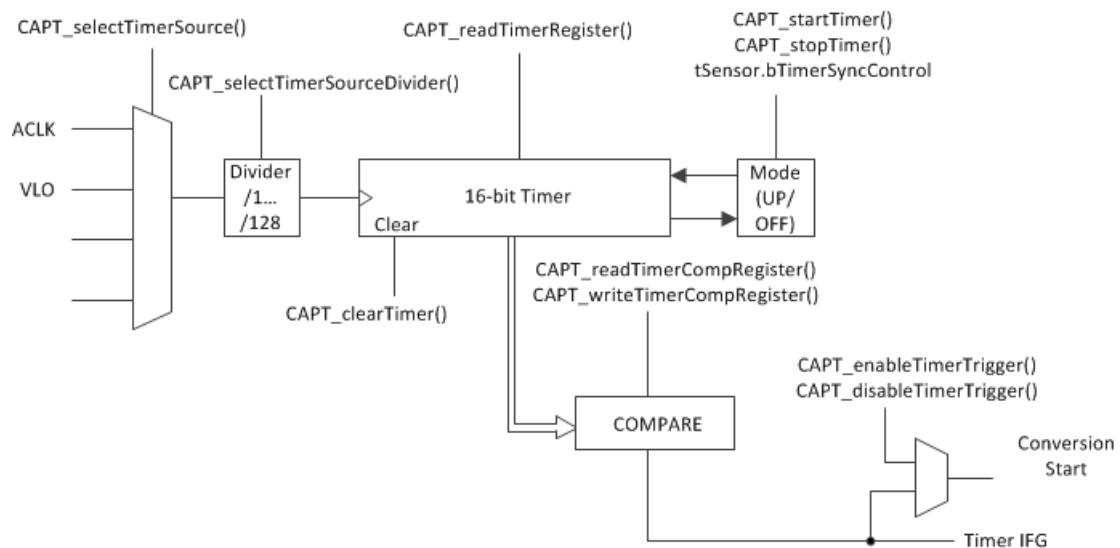
In addition to the CapTlivate™capacitive measurement technology, the CapTlivate™ peripheral is composed of several other helper functions to automate and improve the performance as well as provide additional features specifically related to electro-magnetic compatibility (EMC).

### 5.2.2.1 Low Power Operation

The CapTlivate™peripheral is designed to make single or multiple measurements without any CPU interaction or other peripheral interaction (typically a timer). This is made possible with an integrated timer and finite state machine.

#### 5.2.2.1.1 CapTlivate™Timer

The integrated timer is a standard MSP430 Timer\_A implementation without the capture feature.



The primary purpose for the timer is to periodically trigger conversion events. The interrupt vector is available and therefore the timer is available to the system as a general purpose timer. Table 3, Table 4, and Table 5 show the interrupt related CapTlivate™ registers and specifically the timer associated bits.

Bit	Field	Type	Reset	Description
15-9	Reserved	R	0b	Reserved. Always Read 0
8	CAPMAXIE	RW	0b	CapTlivate maximum count interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled
7-4	Reserved	R	0b	Reserved. Always Read 0
3	CAPCNTIE	RW	0b	CapTlivate conversion counter interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled
2	CAPTIE	RW	0b	CapTlivate timer interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled
1	CAPDTCIE	RW	0b	CapTlivate Detection interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled
0	EOCIE	RW	0b	End of Conversion interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled When enabled, an interrupt is called when EOCIFG = 1; that is, at the end of each conversion. EOCIFG must be cleared by software before exiting the interrupt service routine.

Bit	Field	Type	Reset	Description
15-9	Reserved	R	0b	Reserved. Always Read 0
8	CAPMAXIFG	RW	0b	CapTlivate maximum count interrupt flag 0b = No Interrupt pending 1b = Interrupt pending
7-4	Reserved	R	0b	Reserved. Always Read 0
3	CAPCNTIFG	RW	0b	Specific number of conversions has been reached 0b = No Interrupt pending 1b = Interrupt pending
2	CAPTIIFG	RW	0b	CapTlivate timer interrupt flag 0b = No Interrupt pending 1b = Interrupt pending
1	CAPDTCIFG	RW	0b	CapTlivate Detection interrupt flag 0b = No interrupt pending 1b = Interrupt pending
0	EOCIFG	RW	0b	End of Conversion interrupt flag 0b = No end of conversion has occurred: no interrupt pending 1b = End of conversion has occurred: interrupt pending This bit is set by hardware when each of the enabled CRx channels has finished converting and its results are ready. This bit is cleared by hardware when a conversion is launched (when CIPF becomes 1) or when CAPPWR = 0. If EOCITEN = 1, the CapTlivate interrupt occurs when EOCIFG transitions to 1. EOCIFG must be cleared by the software before exiting the interrupt service routine.

Bit	Field	Type	Reset	Description
15-0	CAPIVx	R	0b	CapTlivate Interrupt vector value. It generates a value that can be used as an address offset for fast interrupt service routine handling. 0000h = No interrupt pending 0002h = Interrupt source: End of Conversion interrupt, Flag = EOCIFG 0004h = Interrupt source: Detection interrupt, Flag = CAPDTCIFG 0006h = Interrupt source: CapTlivate Timer interrupt, Flag = CAPCNTRIFG 0008h = Interrupt source: CapTlivate counter interrupt, Flag = CAPCNTRIFG 000Ah = Interrupt source: max count value reached, Flag = CAPMAXIFG 000Ch to FFFEh = Reserved Read will clear highest priority interrupt. Write will clear all pending interrupts.

#### 5.2.2.1.2 Finite State Machine (FSM)

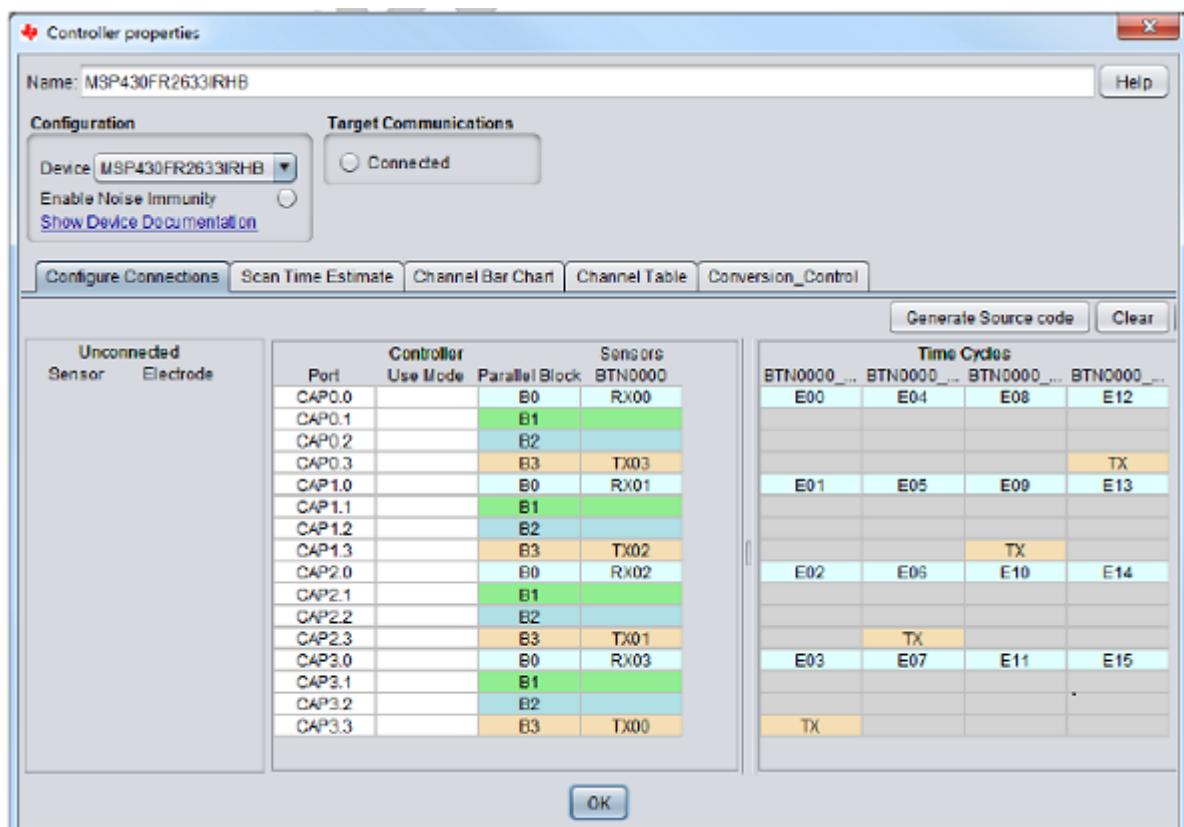
The finite state machine provides several functions to lessen the burden upon the CPU. As mentioned, in conjunction with the timer, the FSM can automate the measurement process so that no CPU interaction is needed until a specified (interrupt) event occurs. This section is divided into four levels of utilization of the FSM: FSM OFF, FSM MANUAL, and FSM AUTO.

## FSM OFF

The maximum count error detection is the only FSM logic that remains enabled when the FSM is bypassed (Library API: CAPT\_bypassFSM()). The maximum count detection logic, defined as tSensor.ui16ErrorThreshold in the CapTlivate™Library, places a digital limit on the conversion result and consequently a limit on the maximum conversion time. This is particularly helpful to help prevent shorts from negatively impacting the entire system, and provides a mechanism to quickly identify faulty keys. In the event that the maximum count error is exceeded the conversion result is set to 0 and the CAPMAXIFG flag is set, see Table 4.

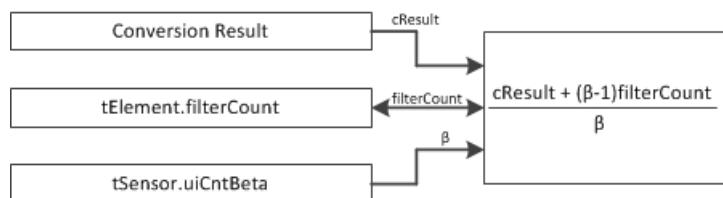
## FSM MANUAL

The FSM MANUAL mode enables the FSM but uses the CPU to move the configuration information from the CapTlivate™library configuration to the peripheral and then the results from the peripheral to the CapTlivate™library configuration. This mode is typically used in a multi-cycle sensor implementation. Figure 9, shows a classic example of a 16-key keypad divided into 4 4-element cycles. The CPU loads the information for each cycle, performs the measurement, saves the results of the measurement, and then proceeds to the next cycle. During the measurement phase the CPU can enter a low power mode or perform other activities.



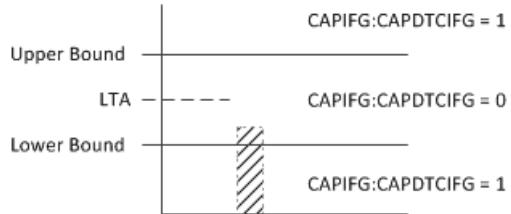
In addition to the maximum count error detection logic, detection logic is also added. This detection logic is predicated upon the current filter count value and the upper and lower thresholds.

The FSM provides a simple low pass filter, as shown in Figure 10, to weight the new filtered count result as a function of the previous filter count and the current conversion result. The weighting is controlled by the input tSensor.ui8CntBeta.



The updated filter count is then compared to the upper and lower bounds. If the filtered count value is within the

bounds then no detection occurs. Once the filter count value exceeds the bounds the detection IFG is set (CAPIFG:CAPDTCTIFG, see Table 4).



The upper and lower bounds are always relative to the long term average (LTA) and the defined in the negative touch and proximity thresholds. It is important to briefly note that the direction of interest is used to interpret the upper and lower bounds. Direction of interest indicates which direction of change (up or down) will be flagged as a proximity event verses a negative touch.

<b>tSensor. DirectionOfInterest</b>	<b>Upper Bound</b>	<b>Lower Bound</b>
eDOIDown	tElement.LTA.ui16Natural + tSensor.ui16NegativeTouchThreshold	tElement.LTA.ui16Natural - tSensor.ui16ProxThreshold
eDOIUp	tElement.LTA.ui16Natural + tSensor.ui16ProxThreshold	tElement.LTA.ui16Natural - tSensor.ui16NegativeTouchThreshold

Once the detection logic has completed, the FSM takes on the task of updating the long term average (LTA). This value has other names; baseline tracking, environmental compensation, etc, but the important concept is that the LTA represents the undisturbed (untouched) system capacitance. Similar to the filter count the LTA is the weighted average of the previous (old) LTA with the new filter count value. The inputs to the LTA calculation are the weighting, tSensor.ui8LTABeta, and the old LTA, tElement.LTA.ui16Natural. Note: The filter count and the LTA are floating point values and the FSM takes care of the floating point math. Only the natural portion is described here since this is what is used for the detection logic.

In addition to the filter, the FSM provides mechanisms to halt or suspend the LTA update. This can be an important feature to prevent other touch events from falsely influencing the LTA for a neighboring or surrounding element.

<b>tSensor</b>			<b>tElement</b>	<b>Result</b>
<b>.bSensorHalt</b>	<b>.bPTSensorHalt</b>	<b>.bPTSensorHalt</b>	<b>.bPTSensorHalt</b>	
False	True	True	True	Default setting; halt the element LTA if the Sensor or the element is in detect,prox, or touch state
True	NA	NA	NA	Halt LTA for all elements in Sensor
False	NA	NA	True	Halt LTA for Element
False	False	True	False	Halt LTA for element if element is in prox or touch state
False	True	False	False	Halt LTA for element if Sensor is in prox or touch state
False	False	NA	True	Halt LTA for Element

#### FSM Auto mode

The automatic mode of the FSM is simply the timer triggered conversion, using the CapTlivate™timer described in the earlier section. Similar to the manual mode, the CPU may be used to move data between the CapTlivate™library structures and the CapTlivate™peripheral. Configuring a sensor for timer triggered conversions can be done by setting the tSensor.bTimerSyncControl bit or using the CAPT\_enableTimerTrigMeasurement API. Typically the CAPIFG:EOCIFG or associated interrupt, is used in the manual mode to indicate that the measurement is finished and it is time for the CPU to move data to and from the configuration structures and process the cycle results while waiting for or even during the next cycle measurement. This flow is illustrated in Figure 12 and Figure 13.

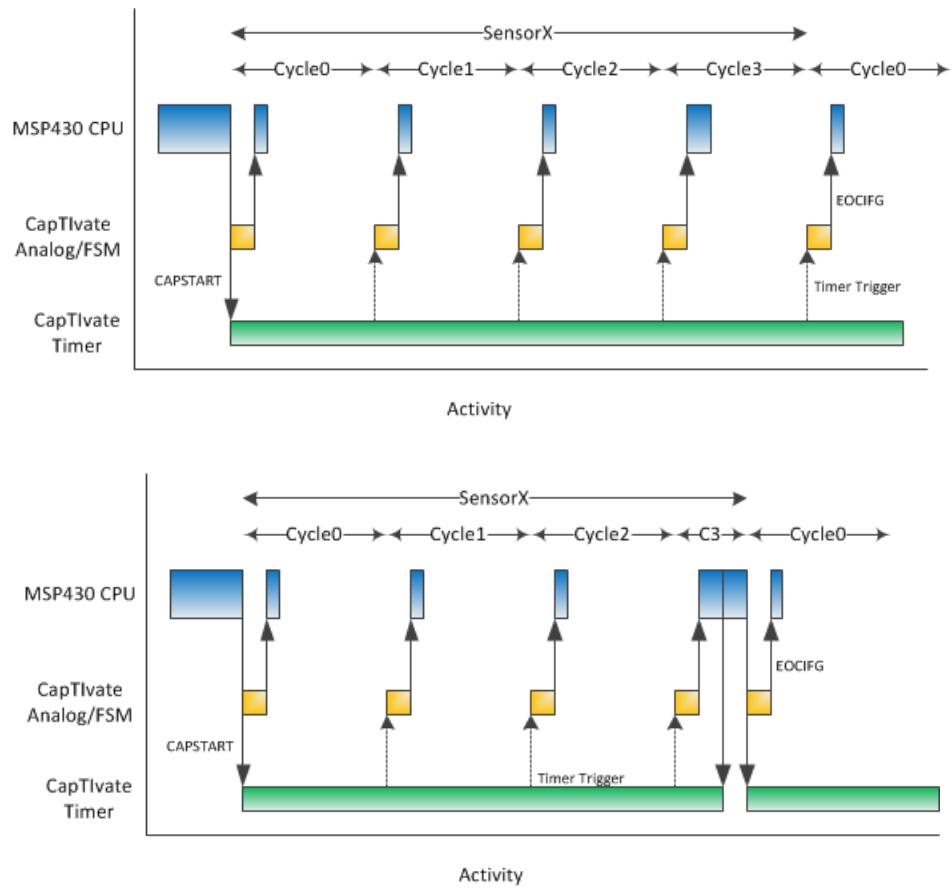
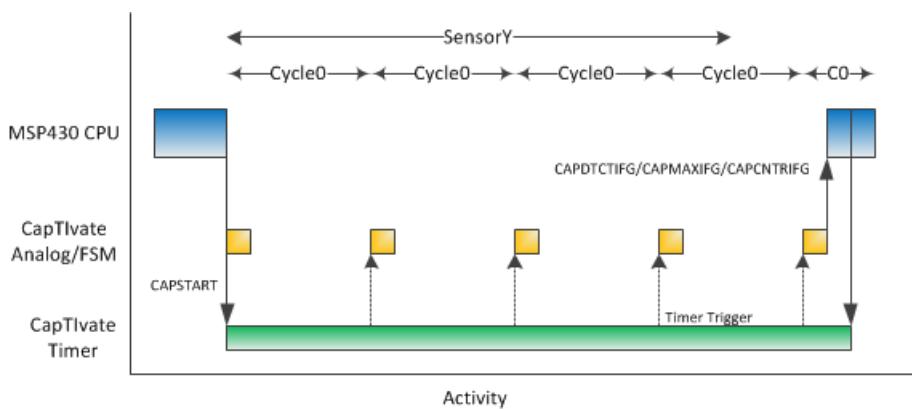


Figure 13 is different, illustrating an event of some kind (error, detection, or diagnostic) that requires the CPU to; stop the timer triggered sequence of events, perform some additional processing, and restarting the timer sequence.

#### 5.2.2.1.3 Wake on Touch Operation (WOT)

The wake on touch operation (WOT) is a special case of the FSM auto mode where the sensor only has one cycle and no CPU operation is required to load new cycle related values. Specifically the LTA and filter count values are updated by the FSM and are available within the peripheral for successive measurements. As shown in Figure 14, cycle 0 is repeatedly measured without CPU interaction. Additionally, in Figure 13, the EOCIFG is used to trigger post processing of the cycle and/or sensor at which time different event flags are read and acted upon. During WOT these events are reconfigured to generate an interrupt and request CPU interaction. The specific interrupts used are the detection interrupt (CAPDTCIEN), the maximum counter error interrupt (CAPMAXIEN), and the conversion counter interrupt (CAPCNTRIEN), see Table 3.



The detection and maximum count events have been described in previous chapters. The conversion counter interrupt is a unique counter which counts the number of conversions and generates an interrupt when a certain

---

number (of conversions) is reached. This provides a defined wake-up period specifically for diagnostic purposes to ensure that the LTA and filter count values are within an expected range or perform self-tests.

API	Brief Description
CAPT_selectCCounterInterval(n)	Select the Conversion Count Interval: $2^{(4+n)}$ , n=0-7
CAPT_clearCCounter()	Set Counter to 0
CAPT_startCCounter()	Enable/Start Counter
CAPT_stopCCounter()	Disable/Stop Counter

### 5.2.2.2 Conversion Timing

The start of a capacitance to digital conversion can be triggered from software, the CapTlivate™timer, the external SYNC pin, or a combination.

A conversion can be initiated from a number of APIs. However, when the actual conversion takes place is a function of the sensor settings. For example, when the ui8InputSyncControl field in the sensor structure is set, the conversion will not take place until the appropriate edge is seen on the SYNC input pin.

Implementation	tSensor.ui8InputSyncControl	tSensor.bTimerSyncControl	Comments
Firmware Trigger	0	False	
Timer Trigger	0	True	First Conversion is synchronous with the firmware. Subsequent conversions are synchronous with timer
SYNC Trigger	1 or 2	False	Rising(1) or Falling(2) Edge
Combination Timer and SYNC	1 or 2	True	Conversion(s) are triggered by SYNC event that follows timer event

### 5.2.2.3 EMC

The CapTlivate™peripheral provides several features to support EMC in terms of emissions and susceptibility.

#### 5.2.2.3.1 Emissions

The spread-spectrum feature of the charge-transfer frequency sweeps the frequency linearly about the center frequency. This spreads the amount of energy radiated in the frequency and thus reduces the peak power radiated at the center frequency. This feature is enabled by simply setting the Boolean tSensor.bModEnable in the sensor configuration.

It is recommended, when modulation is enabled, to make the conversion count a multiple of 40. This helps to ensure that the number of samples (charge-transfer cycles) below the center frequency are equivalent to the number of samples taken above the center frequency

#### 5.2.2.3.2 Susceptibility

Susceptibility can be reduced by measures in both the time and frequency domains. In the time domain, a SYNC input is provided to trigger the conversion event. This is helpful in extremely noisy AC systems when the best time to do the conversion is during the zero-crossing of the AC waveform. In the frequency domain, a frequency hopping feature is provided to move the charge-transfer frequency out of the band where noise is present. The CapTlivate™peripheral includes a dedicated oscillator which can be configured to provide 16Mhz, 14.7Mhz, 13.1Mhz, or 11.2Mhz. These frequencies can be further divided down by the tSensor.ui8FreqDiv parameter as discussed in an earlier section. Automatic control over the oscillator is provided within the noise immunity features of the CapTlivate™library.

---

#### 5.2.2.4 Reference Capacitors

The CapTlVate™ peripheral also comes equipped with a set of reference or self-test capacitors. These capacitors vary in size and can be measured in parallel with an external load or independent of the external connection as a self-test mechanism. There is only one reference test bank of capacitors, so only one block can be tested at a time. Applying the reference capacitor is done with the MAP\_CAPT\_enableRefCap() API. Similar to the channel IO control APIs this API must be called within the context of a sensor and the sensor parameters must have been applied (MAP\_CAPT\_apply\_SensorParams). The reference capacitor size is dependent upon the mode (mutual or self) as defined by the sensor.

capSize	Self Mode (typical, pF)	Mutual Mode (typical, pF)
0	1	0.1
1	1	0.5
2	5	0.1
3	5	0.5
4	1.1	0.1
5	1.5	0.1
6	5.1	0.1
7	5.5	0.1

Removing the reference capacitor from the measurement circuit is done by simply calling the API, MAP\_CAPT\_disableRefCap().

# Chapter 6

## Design Guide

### 6.1 Introduction

Capacitive touch detection is sometimes considered more art than science. This often results in multiple design iterations before the optimum performance is achieved. There are, however, good design practices for circuit layout and principles of materials that need to be understood to keep the number of iterations to a minimum.

Good sensor design are the foundation for a successful touch product.



The purpose of this design guide is to provide guidance for the design and layout of capacitive touch sensors so that they can achieve maximum performance. By achieving maximum performance in the hardware, the CapTlivate™ capacitive touch software library can perform the capacitive touch measurements consuming the lowest power. Tuning guides, along with the CapTlivate™ Design Center are used to help tune the performance of the capacitive touch application.

### 6.2 Starting a New Design

Starting any new capacitive touch design can be a challenging task. This section will guide you through this process and help determine what is important for your design and end application.

#### Identify Sensors

First we want to identify the types of sensors that you are considering for your application and provide some basic design guidelines.

- [Best Practices](#)
- [Buttons](#)

- Sliders and Wheels
- Proximity

### Identify Care Abouts

Next it is important to identify any special requirements for the end application, such as low-power, noise immunity, moisture, etc.

- Ultra Low Power
- Moisture
- Noise

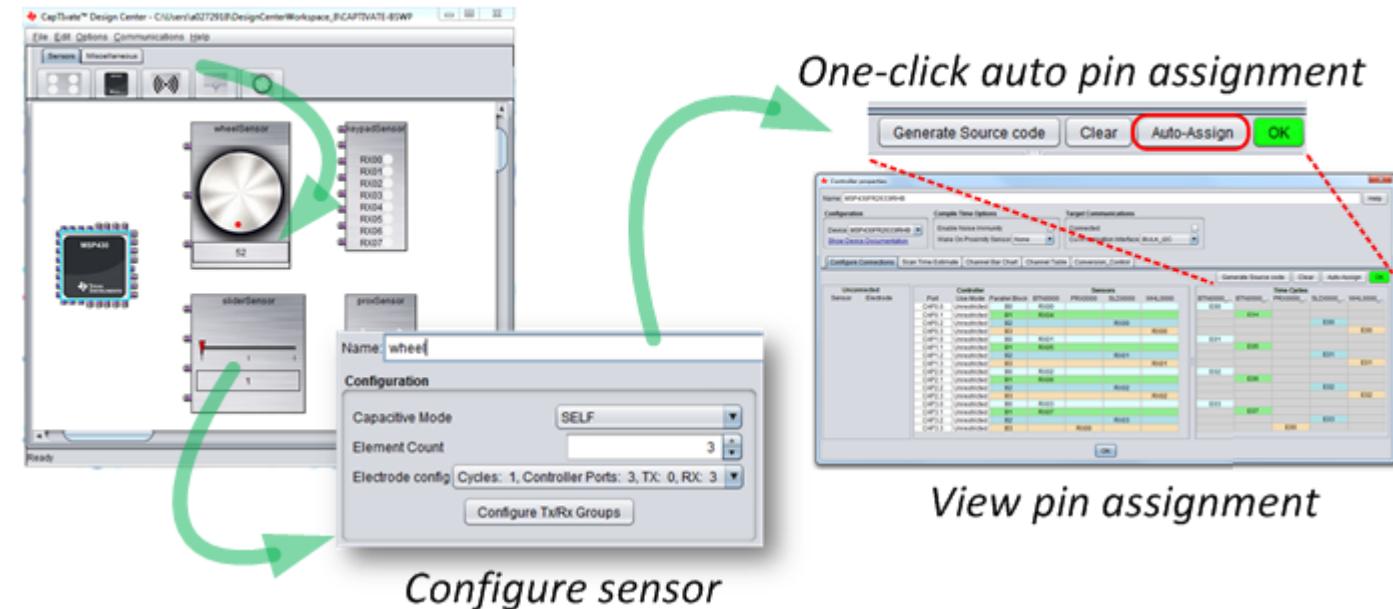
## 6.3 Design Process

Creating a reliable capacitive touch design can be a time consuming process that starts with designing sensors, laying out a PCB, writing lots of code and finally performing an iterative sensor tuning process to achieve the desired performance. The CapTlivate™ Technology sensor design process *accelerates* the capacitive touch development cycle through automation, helping get your product to market faster.

### Creating a new sensor design

We begin the process by creating a new capacitive touch design using the CapTlivate™ Design Center. With a few simple drag-n-drop inputs and configuration selections, the CapTlivate™ Design Center will automatically determine the optimal pin assignments between the target MCU and the sensors (CapTlivate™ Technology peripheral can measure four channels in parallel). Alternatively, connections can be manually assigned or reserved for those applications with specific routing requirements.

### Drag-n-drop sensors

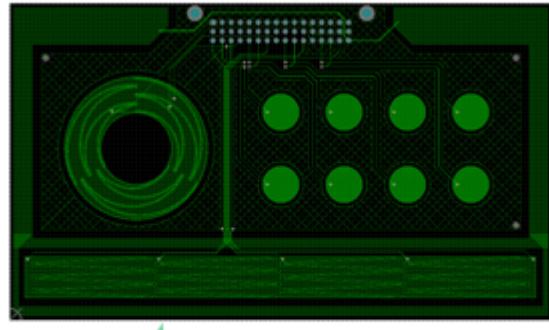


### Layout PCB

Once the pin assignments have been made, the MCU pin to sensor electrode connection information is exported to a .csv file and can be used during the PCB layout and design.

## MCU and sensor connection output for PCB layout

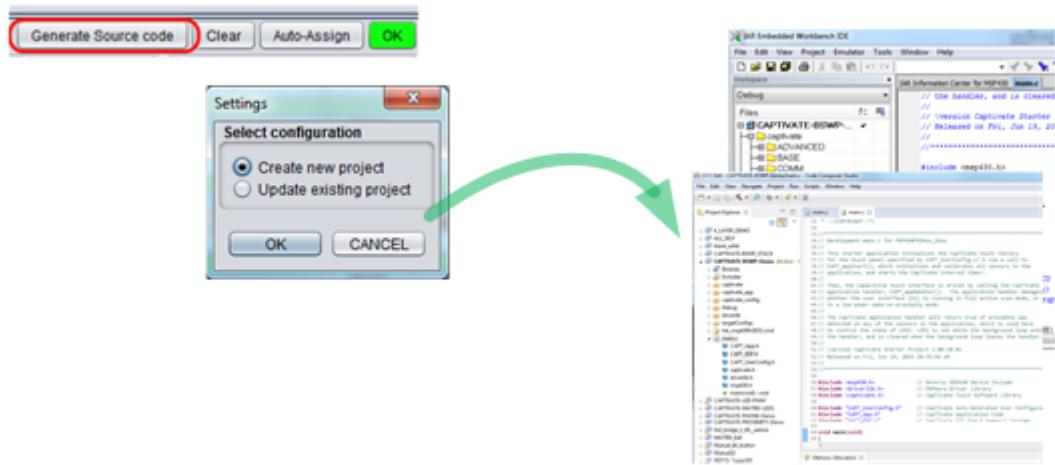
	A	B	C	D	E
1	Sensor	Electrode	MCU Pin	Package Pin	
2	keypadSensor	RX00	CAP0.0	12	
3	keypadSensor	RX04	CAP0.1	13	
4	sliderSensor	RX00	CAP0.2	14	
5	wheelSensor	RX00	CAP0.3	15	
6	keypadSensor	RX01	CAP1.0	16	
7	keypadSensor	RX05	CAP1.1	17	
8	sliderSensor	RX01	CAP1.2	18	
9	wheelSensor	RX01	CAP1.3	19	
10	keypadSensor	RX02	CAP2.0	21	
11	keypadSensor	RX06	CAP2.1	22	
12	sliderSensor	RX02	CAP2.2	23	
13	wheelSensor	RX02	CAP2.3	24	
14	keypadSensor	RX09	CAP3.0	25	
15	keypadSensor	RX07	CAP3.1	26	
16	sliderSensor	RX03	CAP3.2	27	
17	proximitySensor	RX00	CAP3.3	28	



## Generate configuration and starter project files

Now we are ready to create a custom starter code project based on our sensor design. The CapTivate™ Design Center creates a fully working capacitive touch application that can be imported into Texas Instruments Code Composer Studio (CCS) or IAR Embedded Workbench. In addition to the application, the CapTivate™ Software Library and MSP430 driver-lib are automatically included with the project, as portions of the libraries are needed during the development cycle.

## Create MSP430 starter code project and sensor configuration file



MSP430 Code Composer Studio and IAR generated code projects

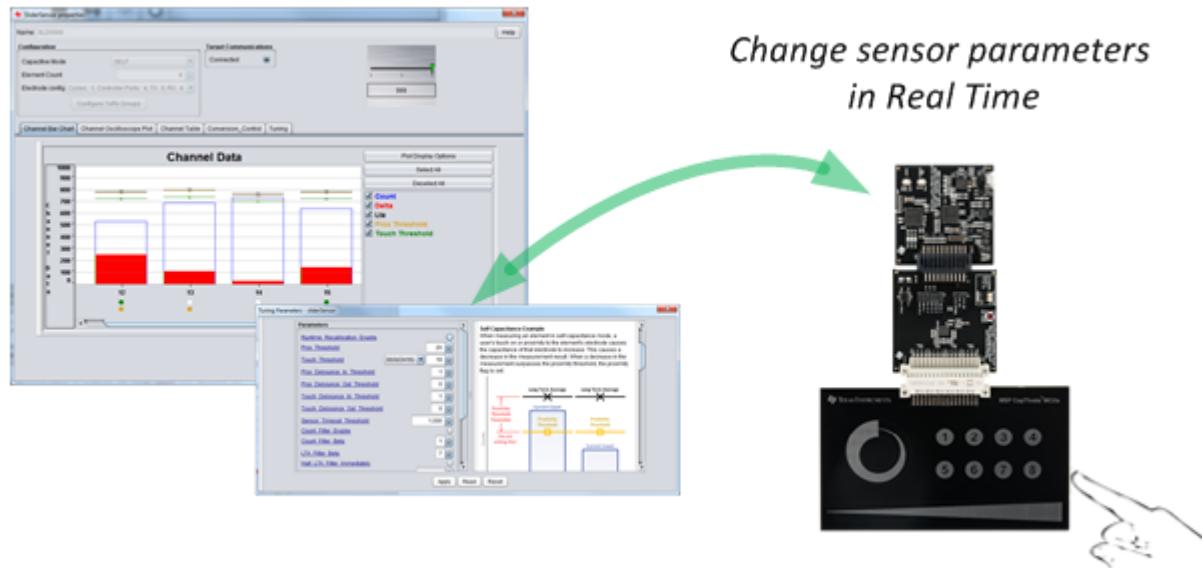
## Program target MCU

Import the generated CapTivate™ project into Code Composer Studio or IAR IDE, compile and program the target MCU. The specific steps to perform the programming operation depend on the IDE, however, once the target MCU is programmed, you are ready to start running your first capacitive touch application without having to write a *single*

line of code!

Didn't get it right the first time? Changes to a sensor design is simple. We use the CapTlve™ Design Center to modify our sensor design project, then generate an updated configuration file. Simply copy the new configuration file and replace the existing one in the CCS or IAR application project and rebuild the application. That's it!

**Tune Sensors** Our PCB is ready to go so we can now begin the process of tuning the sensors for the desired touch and feel. With the MCU programmed with our application and running, the sensors are scanned and real-time data is transmitted to the CapTlve™ Design Center using the high-speed HID serial bridge MCU on the CAPTIVATE-PGMR PCB. The CapTlve™ Design Center has several different views to display and monitor the sensor response and allow quick modifications in real-time to the sensor's parameters, providing instant feedback.



**Generate final configuration** Once our desired touch and feel is achieved, the final sensor configuration can be generated and then programmed into the target MCU. That's it! Using the steps outlined above, capacitive sensor design time can be dramatically reduced.

Congratulations! You are done with your design and ready to go to market.

## 6.4 Best Practices

In this section we will cover both mechanical as well as common layout practices.

Capacitive touch detection is a type of analog-to-digital converter (ADC), specifically a capacitance-to-digital converter. As with most ADCs, the terms of interest are resolution, signal-to-noise ratio (SNR), and linearity, in the specific cases of wheels and sliders. Throughout this document, the design guidance helps to maximize signal, minimize noise, and address when these two goals are at odds.

As mentioned above, the basis of capacitive touch detection is the ability to measure a change in capacitance. This change in capacitance is the signal that the capacitive touch solution identifies. The term sensitivity is often used to describe the signal strength a more sensitive solution has a stronger signal.

Sensitivity is measured in capacitance per counts. In the context of capacitive touch detection, the magnitude of change introduced by a touch is on the order of picofarads or hundreds of femtofarads. It is not uncommon for a solution to see a touch introduce 1 pF of change and be measured as 300 counts. And while the sensitivity might be 3.3 fF/count, this should not be considered until the noise is factored in. Refer to the section on [Noise](#) below.

Sensitivity can be controlled within firmware, but the goal is to provide good sensitivity with the hardware, so that the lowest sensitivity settings can be used in firmware, which provides the lowest-power solution. Referring to the [Equivalent circuit](#) for a self capacitive sensor,  $C_{\text{touch}}$  must be maximized, while the other four capacitances must be minimized. The capacitances of the trace and electrode are approximated as parallel plate capacitances. The following equation serves as the basis for layout recommendations.

---

$$C = \epsilon_r \times \epsilon_0 \times A/d \quad (1)$$

The dielectric constant ( $\epsilon_r$ ), area (A), and distance (d) are described throughout this document with the intent of positively influencing the capacitance for a touch system.

Although parasitic capacitance is presented separately, it is a part of the sensitivity and signal. As already mentioned, the capacitance of interest is the relative change in capacitance. The change in capacitance is based upon the touch interaction, but this change is perceived relative to the parasitic capacitance of the system. The parasitic capacitance is also called the steady-state capacitance or baseline capacitance. If the introduced change is 100 fF, then the sensitivity, which is the relative change in capacitance, can be increased by decreasing the parasitic capacitance.

The capacitances  $C_{\text{trace}}$ ,  $C_{\text{electrode}}$ , and  $C_{\text{parasitics}}$  are generically referred to as parasitic capacitance.

#### 6.4.1 Mechanics

The mechanics are the mechanical characteristics of the design. The mechanics include the overlay material, ink on top of the overlay, any adhesives used to bond the electrode to the overlay or enclosure, and any transition materials used to remove air gaps between the electrode and the overlay. Mechanics also include the types of materials used for the electrodes. The mechanics affect both the signal and the parasitic capacitance.

The goal of this section is threefold:

1. To understand the benefits in terms of both aesthetics and robustness
2. To understand how the materials on top of the electrode and the electrode material itself influence the layout of the electrodes
3. To avoid mistakes in the mechanics that are detrimental to the electrical performance

#### Typical Stackup

The following figure shows a typical stackup for a capacitive touch solution. One of the main goals of this stackup is to reduce (or eliminate, if possible) any low-dielectric (air) gaps between the electrode and the area where the touch takes place. The capacitance associated with the stackup has a very strong effect on the signal (change in capacitance from a touch). The signal is directly proportional to the dielectric of the materials. If possible, high-dielectric materials should be used, but at a minimum the stackup should eliminate any air gaps.

- Air gaps can also contain moisture, which can influence performance or even damage the stackup as temperatures vary and the contents of the gap expand and contract.

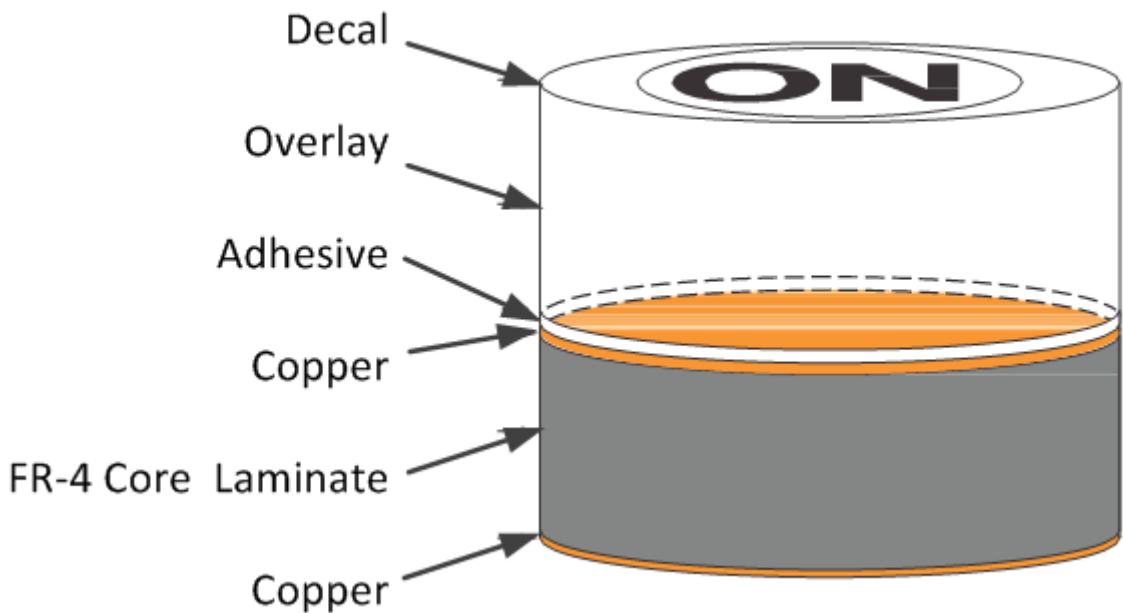


Figure 6.1: Typical Material Stackup

Another critical attribute of the stackup is that it should be non-conductive. This is not usually a problem with the overlay material but can be overlooked when choosing adhesives, labels, or inks. Popular adhesives for capacitive touch solutions include 200MP products from 3M™ such as 467MP and 468MP.

#### Overlay

The capacitance of the stackup is a superposition of all of the material, but the overlay is often the dominant material. The type and dimension of the overlay material that is used is determined by the desired aesthetic and from the required amount of protection. A common requirement is rugged (scratch or puncture resistant) and yet lightweight.

The following figure shows the relationship between the thickness of the overlay and the sensitivity of the circuit. From the parallel plate capacitance equation, the capacitance is inversely proportional to the material thickness ( $C \sim 1/d$ ).

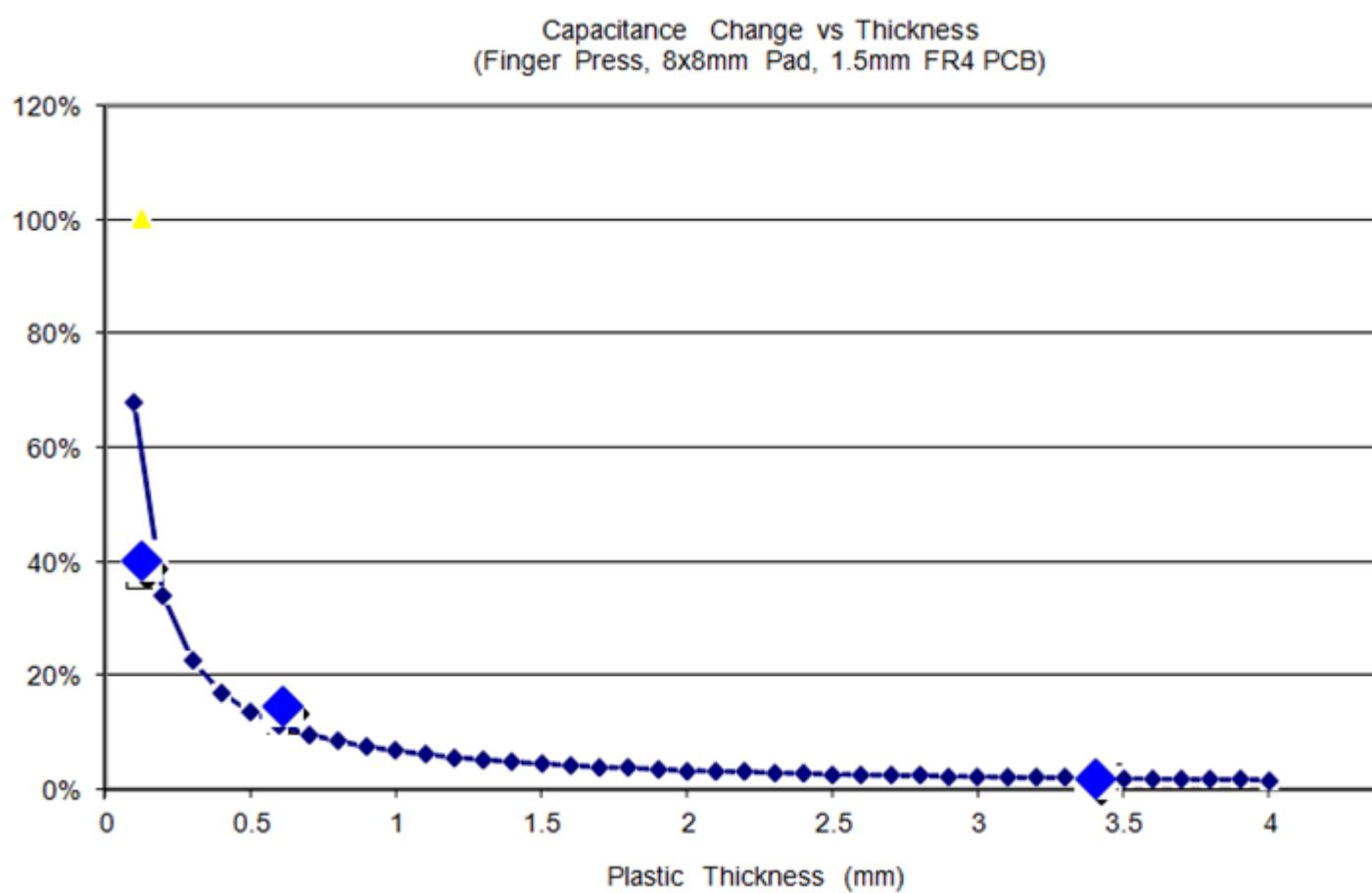


Figure 6.2: Sensitivity vs Thickness

The thickness and dielectric of the material influence the electrode design. The electrode area is a function of the area of interaction (a fingertip or the palm of the hand) while the spacing (to adjacent electrodes or ground fill) is related to the thickness of the overlay. For example, with a 2-mm overlay that has a dielectric of 3, the spacing should be approximately 1 mm (half of the thickness). Using a higher dielectric material (for example,  $\epsilon_r = 6$ ) the thickness could be doubled while maintaining the same level of performance. The following table shows dielectric values for various materials used as overlays.

Table: Material Dielectric and Breakdown Voltage

Material	Dielectric Constant ( $\epsilon_r$ ) (1)	Breakdown Voltage (V/mm)
Air	1.0	3300 (STP)
FR-4	4.8	20000
Glass	7.6 to 8.0	7900
Gorilla® Glass	7.2 to 7.6	See Manufacturer (2)
Polycarbonate	2.9 to 3.0	16000
Acrylic	2.8	13000
ABS	2.4 to 4.1	16000

(1) Relative permittivity

(2) <http://www.corninggorillaglass.com>

The table above also includes the breakdown voltage for different overlay materials. This should be considered when designed for ESD protection. ESD solutions should be system solutions, and any additional components should complement the protection provided by the overlay.

### Electrode and Trace Materials

The performance is affected by the conductive materials that are used for the electrode and for the trace between the electrode and the microcontroller. Most applications use copper on a PCB, and copper has a resistivity of  $1.7 \times 10^{-6}$  Ohm-cm (3). As the resistivity of the conductor increases, the ability to move charge to and from the electrode decreases. This has the same effect as an increase in parasitic capacitance. This increase in resistivity, like an increase in parasitic capacitance, reduces the system sensitivity. The table below shows the resistivity for materials that are commonly used in touch applications.

(3) Resistivity is given in Ohm-cm so that resistance is equal to the resistivity times the length divided by the cross-sectional area :  $R = \rho \times L / A$ .

Table: Resistivity of Materials

Material	Resistivity, $\rho$ (Ohm-cm)
Copper	$1.68 \times 10^{-6}$
Silver	$1.59 \times 10^{-6}$
Tin	$1.09 \times 10^{-5}$
Indium Tin Oxide	$1.05 \times 10^{-3}$ (1)

(1) This resistivity is for a film thickness of 270 nm. Typically, vendors provide sheet resistance instead of resistivity for ITO, which is on the order of 10 to 100 Ohms per square.

When using high-resistivity materials, generally the recommendation is to increase the area of traces to reduce the resistance (at the cost of capacitance). ITO solutions provide lower sensitivity, which must be compensated for in the capacitance measurement algorithm by longer measurement times.

### Other Situations

Not all applications fit into the typical category, and this section describes two special cases. The first is intentional air gaps that are greater than 2 mm between the electrode and the overlay material, and the second is the use of gloves.

## Gaps

In some applications, components are on the same layer as the electrode. This prevents the overlay from being directly applied to the electrode. A common example of this is when an LCD is mounted near the electrode (see diagram below). Another scenario is when the overlay material is not a uniform surface and, therefore, the electrode cannot make direct contact with the overlay.

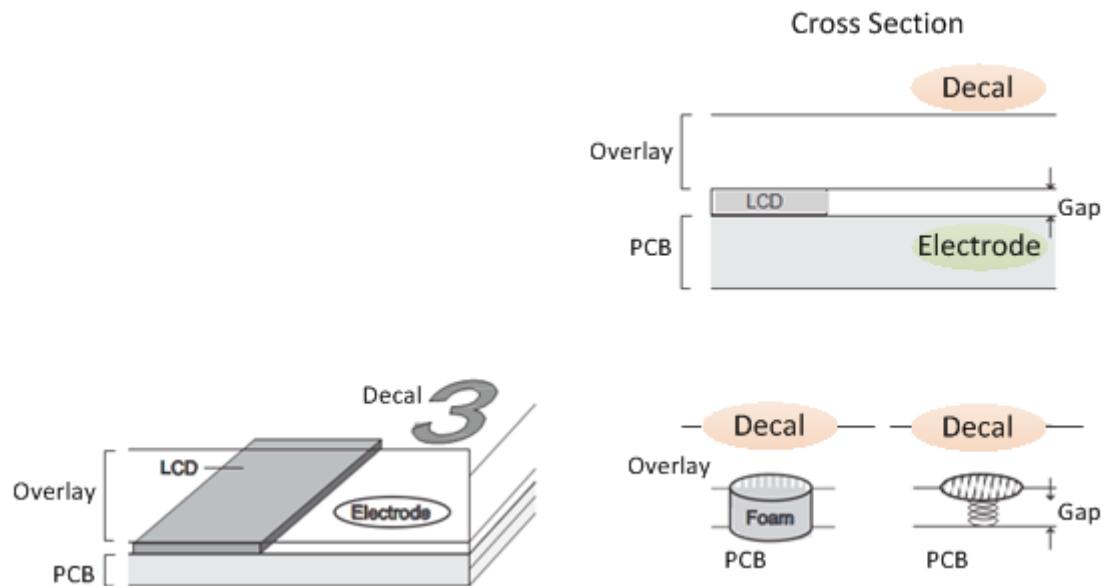


Figure 6.3: Intentional Gaps Between Electrode and Overlay

In either case, the gap must be filled or bridged with a non-conductive filler (typically adhesive) or a conductive extension. When the gap is in excess of 2 mm, then a conductive extension, either foam or metal, should be used. The metal or foam must be malleable to conform to the shape of the surfaces and prevent the formation of gaps. As shown in the Figure above, the area created by the foam or metal in contact with the overlay is now the area that influences the capacitance.

## Gloves

Gloves are simply another layer of medium between the electrode and the finger, and the same principles of thickness and dielectric apply. The challenges with glove applications include the ability to support both gloved and ungloved hands as well as the variation in the types of gloves the application might require. Typical leather or plastic gloves have a dielectric constant in the range of 2 to 4, and fabric gloves and gloves with insulation can have a dielectric constant less than 2.

### 6.4.2 Common Layout Considerations

After the mechanics are understood, the electrodes can be sized and designed to provide the most signal. Independent of the mechanics, the layout design is affected by the distance between the microcontroller and the electrodes, the PCB stackup (for example, one layer, two layer, or four layer), and other electrical circuits on the PCB.

The first item to consider relates to the schematic and the placement of any external components that are associated with the capacitive touch solution. A typical example is ESD protection components such as a series current limiting resistor. In all cases, the components should be kept as close as possible to the microcontroller. As the components move farther away from the microcontroller, the increased area correlates to an increased risk of noise or ESD conducting into the device.

#### 6.4.2.1 Routing

The parasitic capacitance of the trace comprises several major and minor capacitance contributors. For simplicity,  $C_{\text{trace}}$  represents the major capacitances formed between the trace line and the ground pour on the bottom side and surrounding it. The top and cross-sectional views of a typical two layer PCB are shown in Figure 6. The capacitance,  $C_{\text{trace}}$ , is determined by the trace line width ( $W$ ), dielectric thickness ( $H$ ), trace thickness ( $T$ ), and the relative permittivity of the PCB material ( $\epsilon_r$ ).

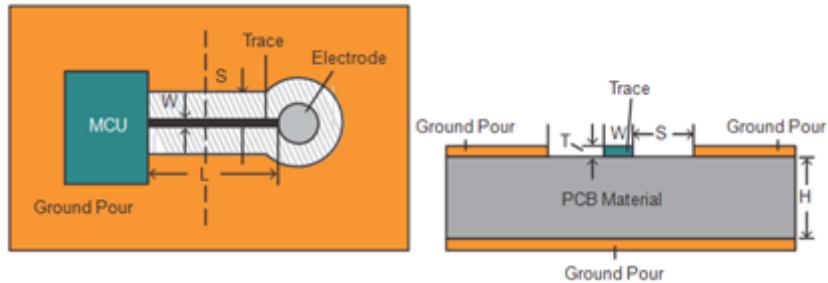


Figure 6.4: Top and Cross-Sectional Views of Trace Line in PCB

The capacitance per unit length of the trace is an important concept to emphasize the need for short traces. Increasing the distance of the trace increases the parasitic capacitance associated with the trace. Increasing the trace length can also increase susceptibility to noise. Therefore, the trace routing between the microcontroller and the electrode should be kept as short as possible. This is not always possible, so it is important to understand the increase in parasitic capacitance associated with the trace routing.

#### Capacitance per Unit Length

The capacitance per unit length should be kept as small as possible to minimize the parasitic capacitance ( $C_{\text{trace}}$ ) and ultimately maximize sensitivity. As previously mentioned, the dominant capacitance in  $C_{\text{trace}}$  is the parallel plate capacitance between the trace and the surrounding ground pour. The ability to reduce this capacitance is a direct function of the PCB manufacturing capabilities. Tighter tolerances and smaller minimum dimensions (trace width and separation) allow for thinner traces and larger separation, which result in lower capacitance per unit length. These manufacturing capabilities typically come at a higher cost.

The table below shows how the capacitance per unit length changes with the variation of different dimensions. These values are taken from [Reference Section](#) in the Coplanar Waveguide Analysis/Synthesis Calculators.

Table: Calculating Results of Capacitance per Unit Length

<b>W (mm)</b>	<b>S (mm)</b>	<b>T (mm)</b>	<b>H (mm)</b>	<b><math>\epsilon_r</math></b>	<b>C (pF/cm)</b>
0.152	0.152	0.036	1.6	4.6	0.633
0.152	0.254	0.036	1.6	4.6	0.555
0.152	0.381	0.036	1.6	4.6	0.496
0.203	0.152	0.036	1.6	4.6	0.692
0.203	0.254	0.036	1.6	4.6	0.602
0.203	0.381	0.036	1.6	4.6	0.543
0.254	0.152	0.036	1.6	4.6	0.740
0.254	0.254	0.036	1.6	4.6	0.641
0.254	0.381	0.036	1.6	4.6	0.578
0.254	0.152	0.036	2.54	4.6	0.736
0.254	0.254	0.036	2.54	4.6	0.637
0.254	0.381	0.036	2.54	4.6	0.566

The table above shows that increasing the space,  $S$ , between the trace line and the ground is an effective way to reduce the parasitic capacitance. However, increasing the separation can have negative effects that need to be understood. One effect is simply increased board space. Increasing the dimensions can lead to larger PCBs and higher cost. Another effect is related to noise. The larger separation makes traces more sensitive to touch events (touching the trace instead of the electrode) and more susceptible to radiated emissions.

In practice, the designer should choose a balanced  $S$  value, and a value of 1/8 of the overlay thickness is typically

acceptable. Additionally, a hatched ground is commonly used instead of solid fill near the trace lines to reduce the area and consequently the parasitic capacitance.

The table above also shows that as H gets smaller, the parasitic capacitance increases, which results in a decrease in sensitivity. In most applications, a two-layer PCB is used, and a standard FR4 PCB with the thickness of 1 mm to 1.6 mm is recommended. If a multilayer board is used, it is recommended to keep the H as large as possible. With complex multilayer boards (more than six layers), it is important to recognize that the absence of copper in the internal layers can cause issues and the height, H, between the top layer and the bottom may be smaller than predicted.

The figure below shows an example of a four-layer PCB, which may be desired in applications with limited board space. To reduce parasitic capacitance, the ground pour is usually placed on the lowest layer underneath the trace to provide the largest H. If it is difficult to achieve the maximum height, then a narrower W, a larger S, or even a hatched ground fill are alternatives to reducing the parasitic capacitance.

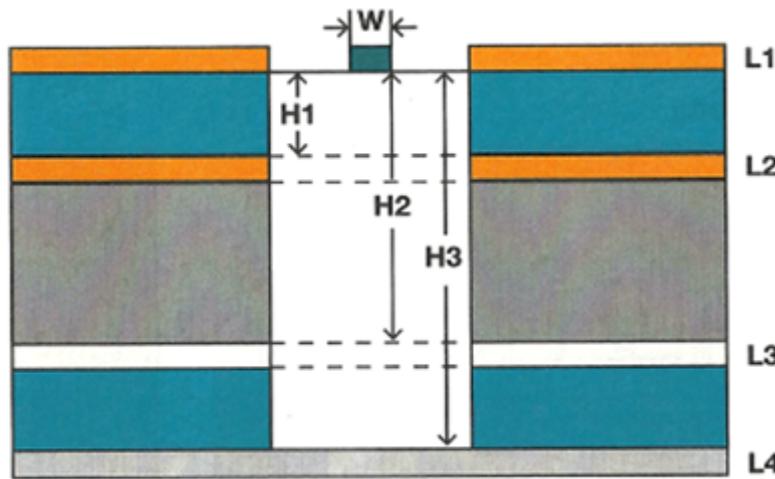


Figure 6.5: Trace Without Copper Pouring Underneath in Multilayer PCB

To determine the maximum trace length from the capacitance per unit length, some additional information is needed. The following example is used to calculate the maximum distance for a trace with a capacitance per unit length of 0.58 pF/cm. The capacitance introduced by a finger,  $C_{\text{touch}}$ , is assumed to be approximately 1 pF. To make sure that the capacitance induced by a finger is large relative to the parasitic capacitance, the total parasitic capacitance ( $C_{\text{parasitics}}$ ,  $C_{\text{trace}}$ , and  $C_{\text{electrode}}$ ) should be in the range of 10 pF to 20 pF (the change is at least a 5% change). Assuming that the  $C_{\text{electrode}}$  is approximately 3 pF, the  $C_{\text{parasitics}}$  is 5 pF, and the capacitance per unit length is 0.58 pF/cm, so the trace line length L should be no longer than 210 mm. If the electrode is larger (for a proximity application), the capacitance itself is larger (assume approximately 8 pF for this example), so the maximum trace line length L is reduced to 120 mm.

Generally speaking, the trace width W should be as thin as the PCB technology allows, because a short and narrow trace line is preferred. The trace thickness T and the relative permittivity of the material  $\epsilon_r$  also have significant influence on capacitance per unit length, but they are determined by PCB manufacture process and are usually difficult to change.

#### Connectors

Some designs require the electrode to be off-board, and consequently a connector is used to transition the trace from the PCB to a cable or to another PCB. Connectors are generally not desired because like any component there is an associated parasitic capacitance with connector PCB footprint and structure.

In terms of parasitic capacitance and sensitivity, the connector is treated as a parasitic capacitance reducing the sensitivity of the solution. Because the connector is treated as a lumped capacitance, the placement is irrelevant to the sensitivity. However, the placement is important with respect to noise.

The figure below shows that if the aggressor (noise source, N1) is located on the PCB, then the preferred placement of the connector is near the MSP430 microcontroller. The parasitic capacitance associated with the

connector shunts high-frequency noise ( $Z = 1/j\omega C$ ) and increases the noise immunity of the circuit. Conversely, if the aggressor is located off-board (noise source, N2), then the connector should be placed farther from the microcontroller and closer to the electrode. This arrangement minimizes the off-board trace and cable length (which acts like an antenna). When multiple aggressors exist as is shown in the figure below, it is recommended to reduce the effect of the radiators that most closely match the operating frequency of the touch detection circuit.

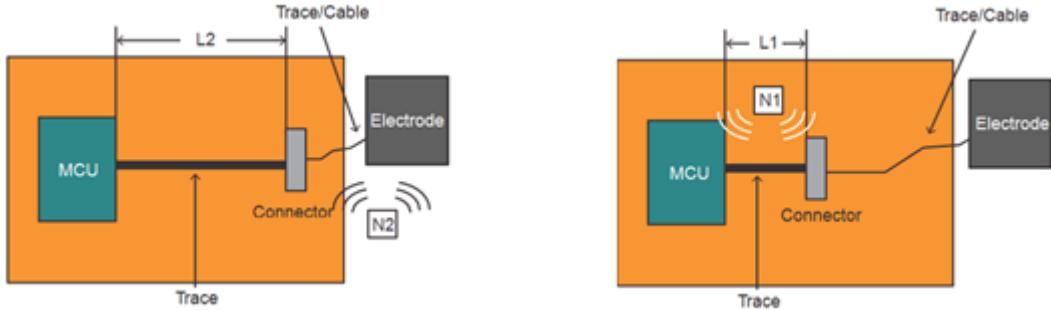


Figure 6.6: Top View of Connector and Noise Source

When deciding the position of the connector, the designer must balance the length of the trace and the cable while considering parameters like the background noise.

#### Routing Material

The routing is typically done with copper, but other materials like silver and ITO can be used. Silver is similar to copper and similar performance can be expected (if the thickness of the materials are also equivalent). ITO is very different from copper, and the difference in resistivity degrades the sensitivity of the solution. Lowering the impedance of the trace (by increasing the width) should be a high priority in the design, even though this comes at the cost of increased parasitic capacitance and reduced noise immunity. Ultimately, the use of materials like ITO requires more processing by the MSP430 microcontroller to adjust to lower sensitivity and increased noise.

#### 6.4.2.2 Electrode Material

As discussed in the [Routing section](#), conductivity becomes an issue in more resistive materials like ITO. Although the transparency of ITO is very good, the resistivity is high when compared to materials like silver and copper. Typically the physical dimensions prohibit increasing the area of the ITO electrodes, and therefore any degradation in sensitivity must be compensated for in the firmware. This typically results in slightly longer measurement times and consequently increased power consumption.

#### Electrode Design

The electrode design must accomplish two goals. First, the design must provide sufficient signal (change in capacitance with interaction). The design must project the e-field up and out so that the appropriate level of sensitivity is achieved at the desired distance. Understanding the stackup, thickness and dielectric, the electrode can be sized and shaped to provide the maximum signal. Second, the electrode design needs to have a minimal parasitic capacitance.

In the following sections the shape and area of the electrode are discussed with the intent of maximizing the signal for different implementations (buttons, sliders, and wheels). The basis for controlling the parasitic capacitance is common to different sensor implementations and is discussed here.

The figure below shows an example PCB cross-section and the important parameters that influence the parasitic capacitance. The height, width, and separation have a direct effect on the parasitic capacitance of the electrode,  $C_{\text{electrode}}$ . The fundamental parameter area is not shown in the figure below because this has a direct effect on both the touch capacitance ( $C_{\text{touch}}$ ) and the parasitic capacitance ( $C_{\text{electrode}}$ ). This section describes how changes to the height and separation can minimize the parasitic capacitance. The following sections describe how changes to the area can maximize  $C_{\text{touch}}$ .

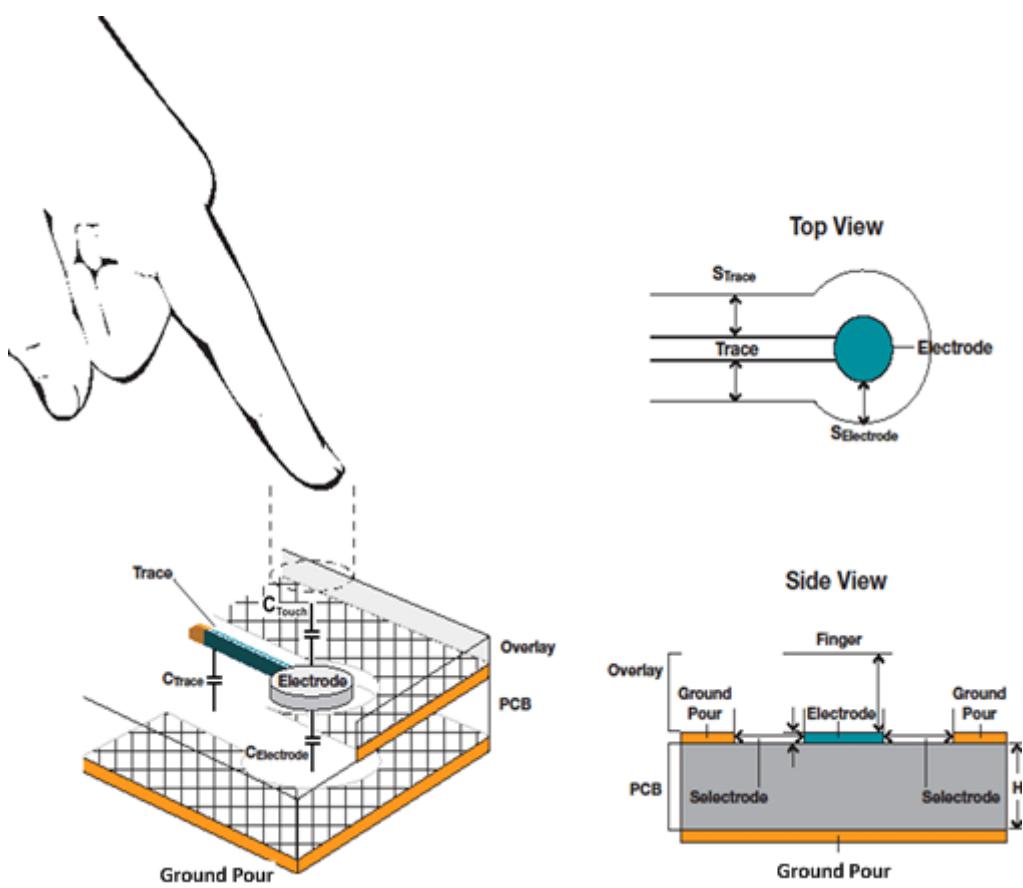


Figure 6.7: Capacitance of the Electrode

---

The separation ( $S_{\text{electrode}}$ ) is directly related to the height of the overlay as described in the [Mechanical section](#). The height is a function of the PCB and is not a parameter that can be easily controlled. The separation is typically the parameter with the most flexibility.

Table: Baseline Electrode Capacitance

Area(mm <sup>2</sup> )	Height(mm)	Description	Separation(mm)	Capacitance(pF)
10x10, FR-4 (Er = 4.4)	1.572	2 Layer PCB, L2	0.508	3.3
10x10, FR-4 (Er = 4.4)	1.572	2 Layer PCB, L2	1.02	3.2
10x10, FR-4 (Er = 4.4)	1.572	2 Layer PCB, L2	1.52	3.1
10x10, FR-4 (Er = 4.4)	1.30	4 Layer PCB, L2	0.508	3.8
10x10, FR-4 (Er = 4.4)	1.57	4 Layer PCB, L3	0.508	3.3

The table above shows that increasing the height (the distance between the electrode and the reference plane) decreases the capacitance. By decreasing the parasitic capacitance, the relative change in capacitance caused by a touch event is increased. For example, if the change in capacitance associated with a touch is 0.5 pF, the relative change is greater when the base capacitance is 11 pF instead of 12 pF, (5.5% instead of 4.2%).

#### 6.4.2.3 Spacing Between Electrodes

By default, the CapTlve™ Software Library drives non-actively scanned electrodes to ground allowing neighboring electrodes to be treated as an extension of the ground pour. Therefore, the spacing between the electrodes follows the same rules for spacing from ground. The goal is provide enough spacing so that the e-field propagates up and through the overlay material. A minimum spacing of one-half the laminate thickness has been found to provide sufficient signal (sensitivity).

#### 6.4.2.4 Shapes

The capacitance of the electrode is a function of area, but the shape is important to consider, because the shape can influence the area. An important detail of designing the electrode shape is not to design shapes that have low surface area. The area of each electrode must provide the maximum  $C_{\text{touch}}$ , which in turn produces the most signal (the change in capacitance) when a touch event occurs.

#### 6.4.2.5 Crosstalk

In the Connectors section above, the aggressor is assumed to be a point source of noise. In this section, the aggressors are different signals that are routed in close proximity to the capacitive touch sensor trace. These aggressors can be another capacitance sensor trace or non-capacitance sensor lines. Examples of non-capacitance sensor lines include digital signals, analog signals, and high-current signals used to drive LEDs.

#### 6.4.2.6 Adjacent Capacitive Touch Signals

Capacitive sensor traces influence neighboring capacitive touch sensor traces. The space between capacitance sensor trace lines,  $Scs$ , should be kept as a safe distance (see figure below).

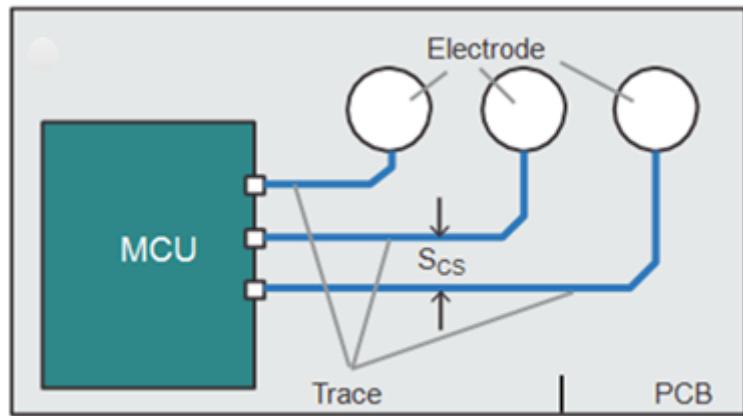


Figure 6.8: Top View of Different Traces

#### 6.4.2.7 Digital Signals

Digital signals are typically PWM signals or communications like I<sub>2</sub>C or SPI. Unlike the capacitive traces, these signals can act as aggressors and can be active during a capacitance measurement. It is recommended to keep these types of signals at least 4 mm away from the capacitive touch trace. If the digital signal and the capacitive touch trace must cross, then it is recommended to keep the crossing at a 90 degree angle.

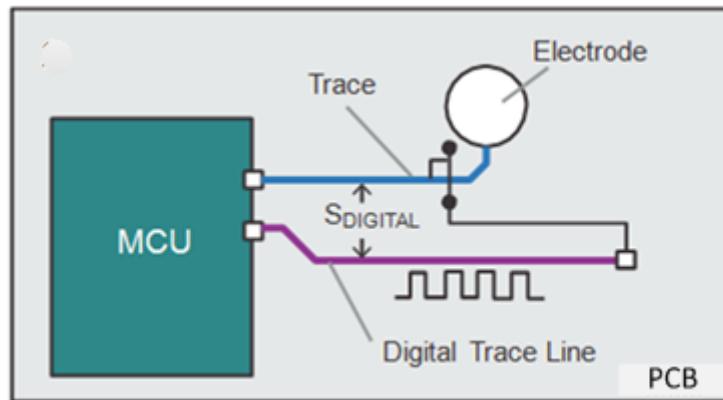


Figure 6.9: Top View of Different Traces

#### 6.4.2.8 LEDs/LED Backlighting

Signals used to drive LEDs (unless the LEDs require high-strength drivers) are similar to other digital signals. As with digital signals, a distance of at least 4 mm is strongly recommended for  $S_{LED}$  as shown in the figure below.

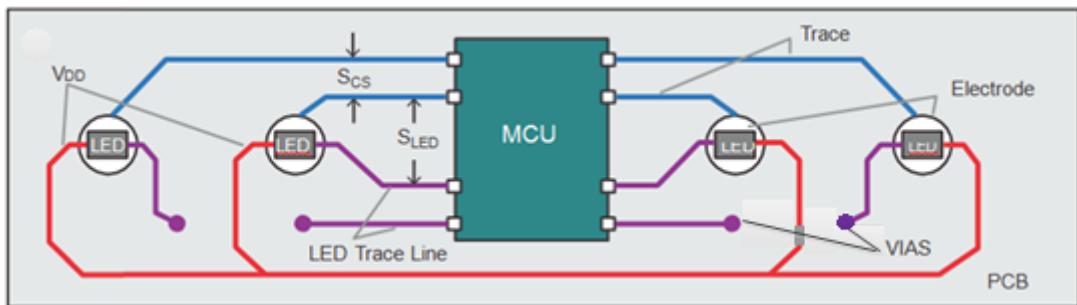


Figure 6.10: Top View of Different Traces

As a general rule, LEDs should be driven and the use of high-impedance states should not be used to control the LED. The use of a high impedance to prevent an LED from conducting can result in a significant difference between the on-state and off-state capacitances. This change in capacitance may be detected by the touch solution and treated as a change in the system capacitance, or even worse as a false detection. If the use of high-impedance control of the LED is unavoidable, a discrete capacitor (typically 1 nF is acceptable) in parallel with the LED is recommended.

- Avoid tri-stating the LED as the LED capacitance will change when turned on and off
- If LED must be tri-stated, add small bypass capacitor to control change in LED capacitance
- Capacitor does not have to be physically near LED, just in the circuit

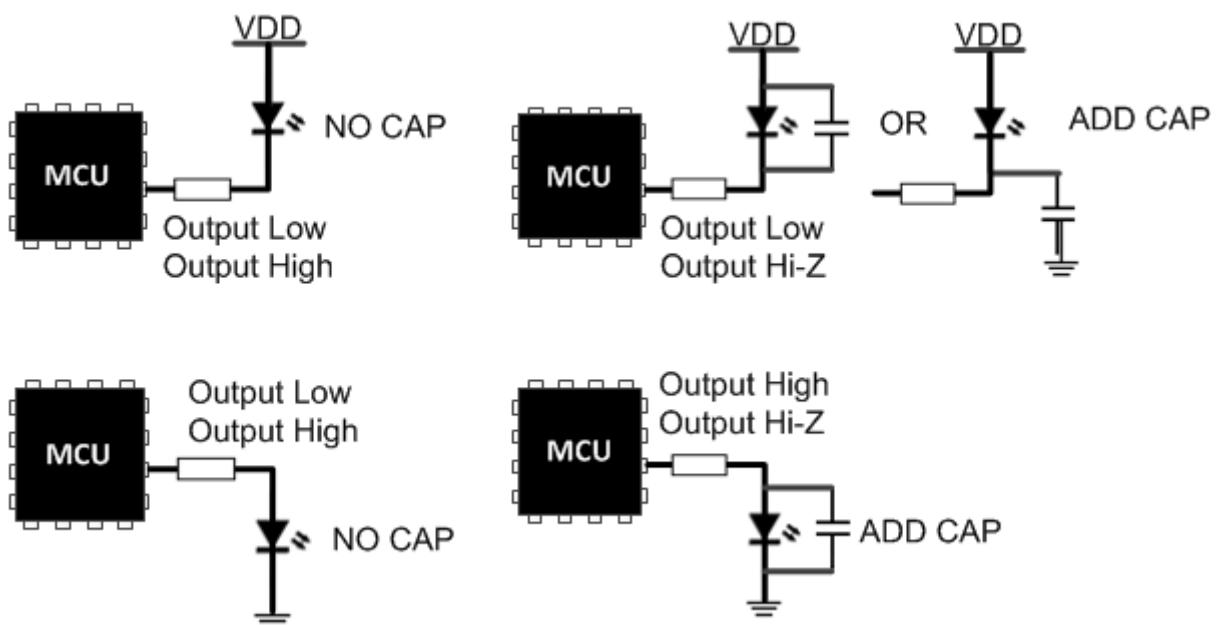


Figure 6.11: Recommended methods to drive LEDs

LED BackLighting can be done easily using a back lighting LED on the opposite side of the PCB

- Keep hole as small as possible to eliminate possible dead spots in sensor
- LEDs may require bypass capacitor (see above)

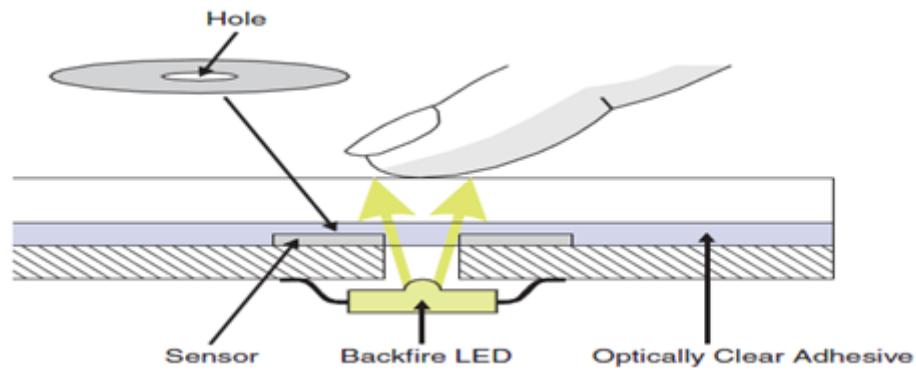


Figure 6.12: Backlighting LEDs

As an example of an LED backlighting technique, the following wheel design has an LED mounted on the backside of the PCB and illuminates through a small hole. In parallel to the LED, a small 1nF capacitor is used to reject changes in capacitance when the LED is switched between ground and high-Z. Also note the hatched ground on both the top and bottom layers to help with noise immunity.

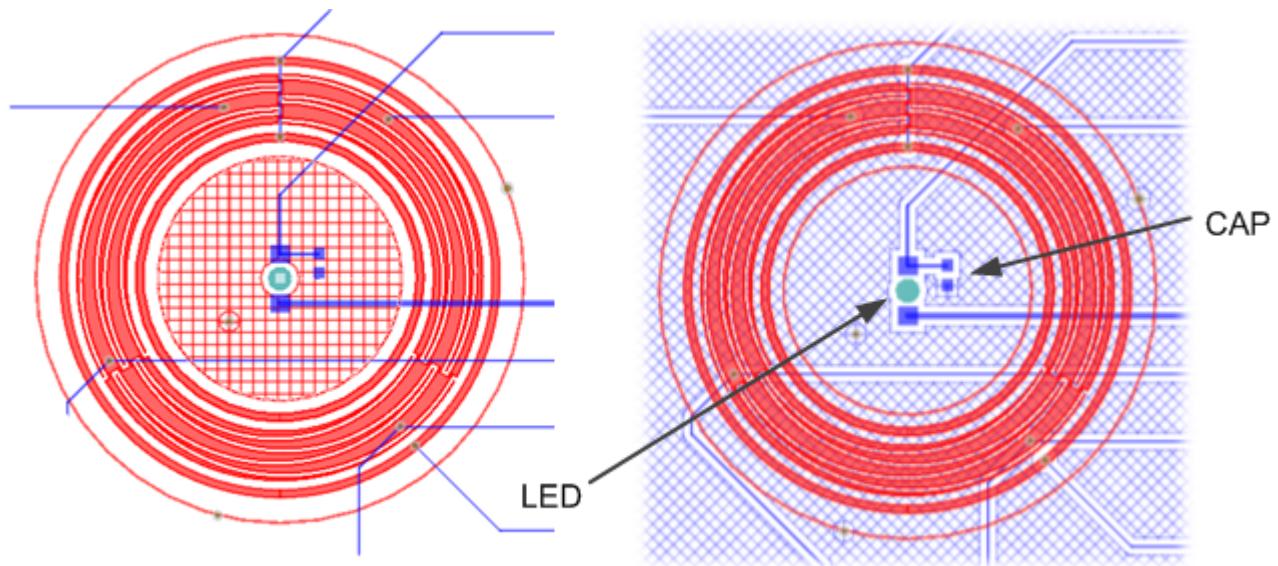


Figure 6.13: LED BackLight Example

#### 6.4.2.9 Ground Planes

Surrounding ground planes affect the sensitivity of the electrode. This is seen primarily as an increase in base capacitance as the separation from the ground decreases and the area of the ground pour increases. This section looks at the placement of surrounding ground pours and the fill (percent hatch) of those pours.

---

Planes and pours near the electrode and trace must be connected to a potential and cannot be left floating or in a high impedance state. Such structures serve as a mechanism for noise coupling and are strongly discouraged.

#### 6.4.2.10 Separation

Ground planes, both coplanar and on neighboring PCB layers, reduce noise. This is the same principle discussed in the [Routing section](#). The ground or guard structures are placed as close as possible to reduce noise but also kept far enough away to minimize parasitic capacitance. This separation is a function of the thickness of the materials (overlay, adhesive, etc.) on top of the trace. As mentioned in the [Routing section](#), a good rule is one-eighth the thickness for separation between traces and ground. The separation between electrodes and ground should be at least one-half the thickness.

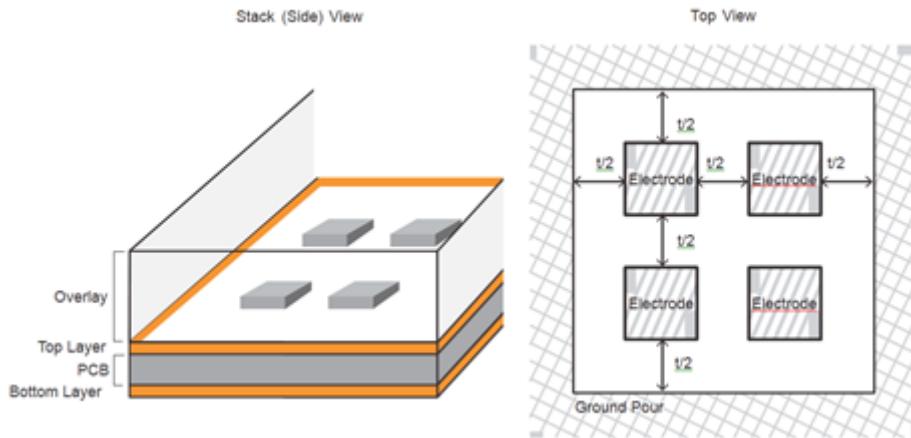


Figure 6.14: Ground Separation

As shown in the figure above, the separation between the electrode and the coplanar ground pour should be at least one half the thickness of the overlay material. Any unused electrode should be held at a logic low level ( $V_{ss}$  potential) and not allowed to float. In this way the spacing between electrodes is the same as the ground and electrode distance.

#### 6.4.2.11 Pour

The use of a hatched pour instead of a solid ground pour is a good design practice. This reduces the area and consequently the parasitic capacitance associated with both  $C_{trace}$  and  $C_{electrode}$ . Typically, a 25% fill hatch is sufficient, but this percentage can be increased or decreased to improve noise immunity or sensitivity, respectively.

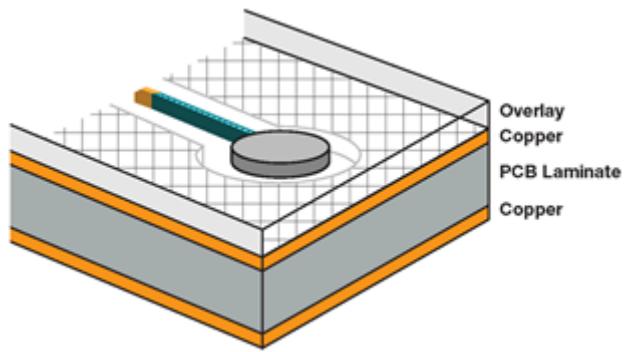


Figure 6.15: Example Hatched Ground Fill

#### Using Poly Cut Out to control Ground Pour around Sensors

When designing a button, slider or wheel of any size or shape, a flexible method to control the distance between any sensor and the ground pour is to provide a poly cut-out region around the sensor as shown below.

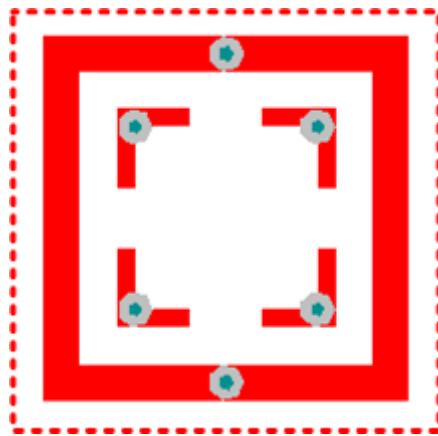


Figure 6.16: Define Poly Cut Out Region

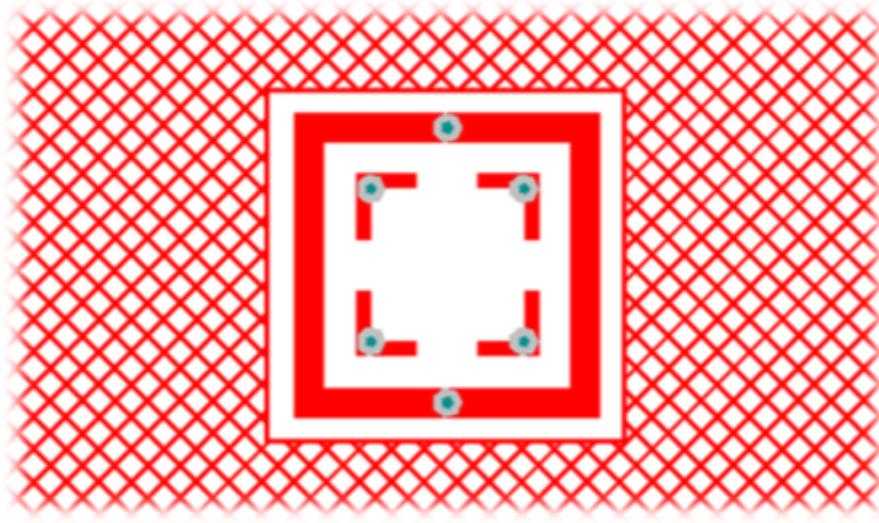


Figure 6.17: Ground Pour Spacing Controlled

#### 6.4.2.12 Mutual Capacitance Traces

Routing mutual capacitive traces has special considerations compared to self capacitive traces. Specifically, the TX and RX signals on the same layer should be close only in those areas where coupling through a finger touch is expected by design, such as the case with button, slider or wheel elements (electrodes). Anywhere else that TX and RX traces on the same layer are routed close can be susceptible to a "ghost" touch should a finger touch in that area.

##### Traces

- Avoid routing TX lines near RX lines when possible
  - Within a finger spacing, they will create a button
  - If unable to avoid on same layer, put ground trace in between (adds parasitic capacitance)
  - If TX needs to cross RX, make crossing perpendicular (minimize surface area intersection)
  - Avoid running TX on one layer of PCB parallel to RX on other layer of PCB
- Route TX lines next to other TX lines
- Route RX lines next to other RX lines
- Keep nearby ground away from TX and RX traces by 1/2 panel thickness (reduces parasitic capacitance)

The diagram below shows an 8-button keypad using (2) TX and (4) RX traces. In this layout the TX and RX traces are on the same layer. The TX traces are routed close to each other as are the RX traces. The TX traces are routed away from the RX traces. In this example there is one RX that must be routed near a TX trace. A ground trace/fill is used to separate the RX and TX traces.

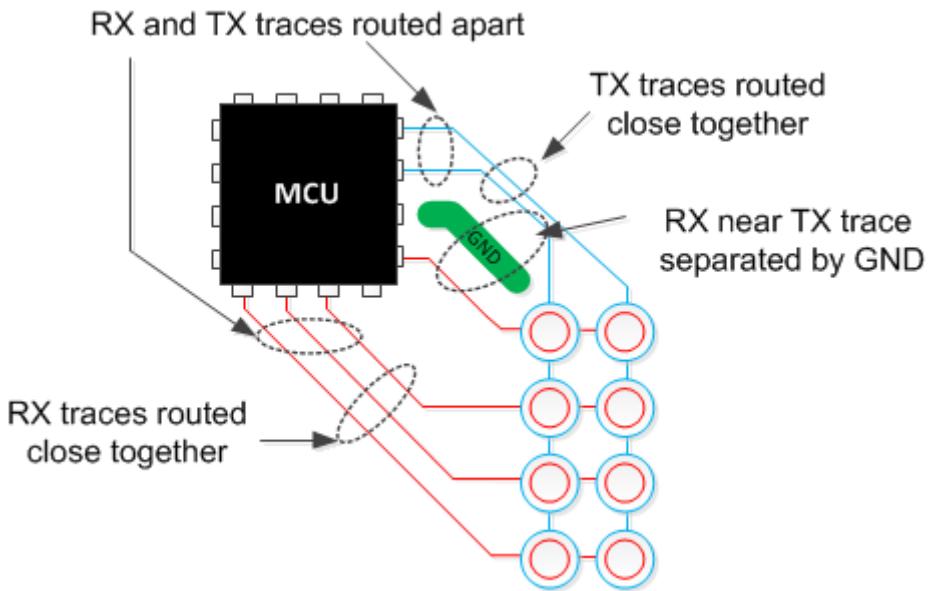


Figure 6.18: Routing mutual capacitance sensors

#### Electrodes

- For noise suppression, use 25% hatched ground on:
  - Bottom layer of PCB and/or
  - On top layer of PCB keeping ground 1/2 panel thickness away from electrode

## 6.5 Buttons

### 6.5.1 Self Capacitive Buttons

A self capacitive button sensor is a single electrode. Self capacitive buttons are simple to layout and each button is assigned to only one pin on the MCU. Self capacitive buttons will provide greater sensitivity as compared to a mutual capacitive button, but are more influenced by parasitic capacitances to ground.

Parameter	Guidance
Radiation Pattern	Between Electrode and Ground
Size	Equivalent to interaction
Shape	Various: typically round or square
Spacing	0.5 x Overlay minimum thickness

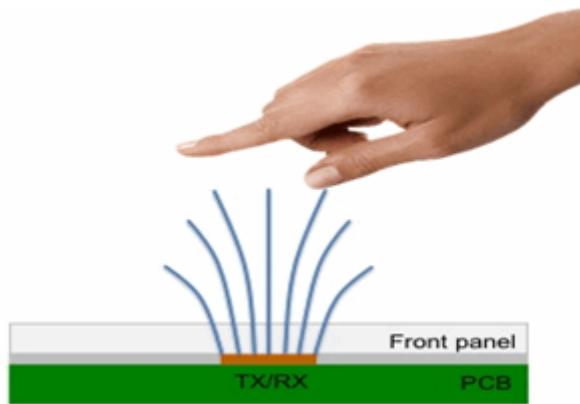


Figure 6.19: Example Self Capacitive Button Designs

#### 6.5.1.1 Self Capacitive Button Shapes

The electrode shape is typically rectangular or round with common sizes of 10mm and 12mm. Ultimately, the size will depend on the required touch area. A good design practice is to keep the size of the button as small as possible, which minimizes the capacitance and will help with the following:

- Reduce susceptibility to noise
- Improve sensitivity
- Lower power operation due to smaller capacitance and reduced electrode scan time

In the diagram below, an example silkscreen button outline pattern is shown.



Figure 6.20: Example Self Capacitive Button Designs

The goal of the button area is to provide sufficient signal when the user touches the overlay above the button electrode. Typically a nonconductive decal or ink is used to identify the touch area above the electrode. The relationship between the decal and the electrode can be varied so that contact with the outer edge of the decal registers a touch. Conversely, the electrode could be small to ensure that the button is activated only when the center of the decal is touched. The two figures below show how the effective touch area is a function of the electrode size and the size of the finger.

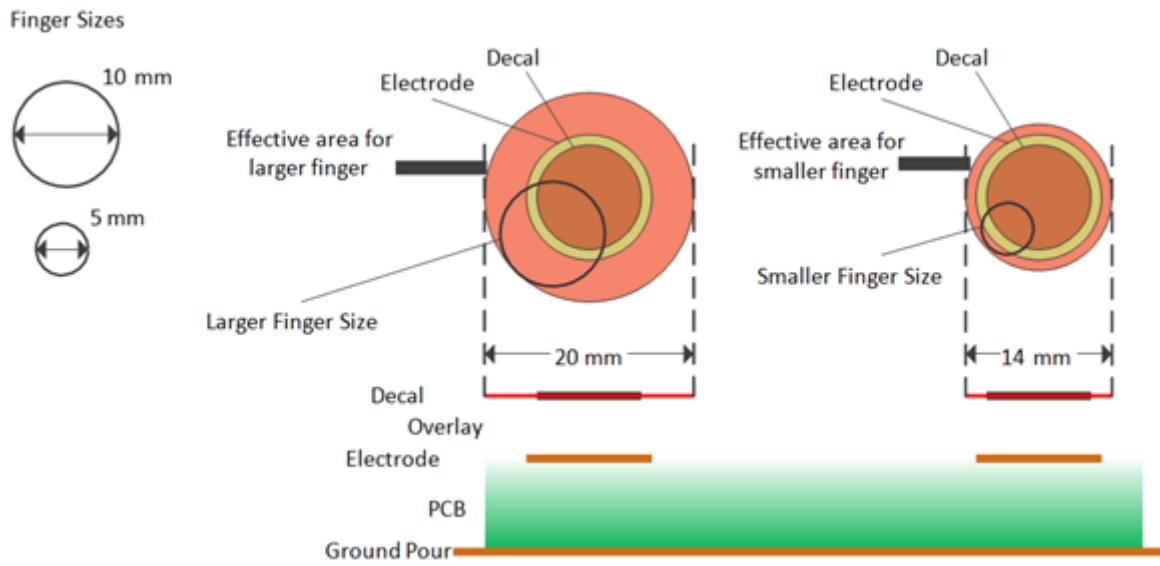


Figure 6.21: Effective Area Example for Electrodes Larger Than Decal

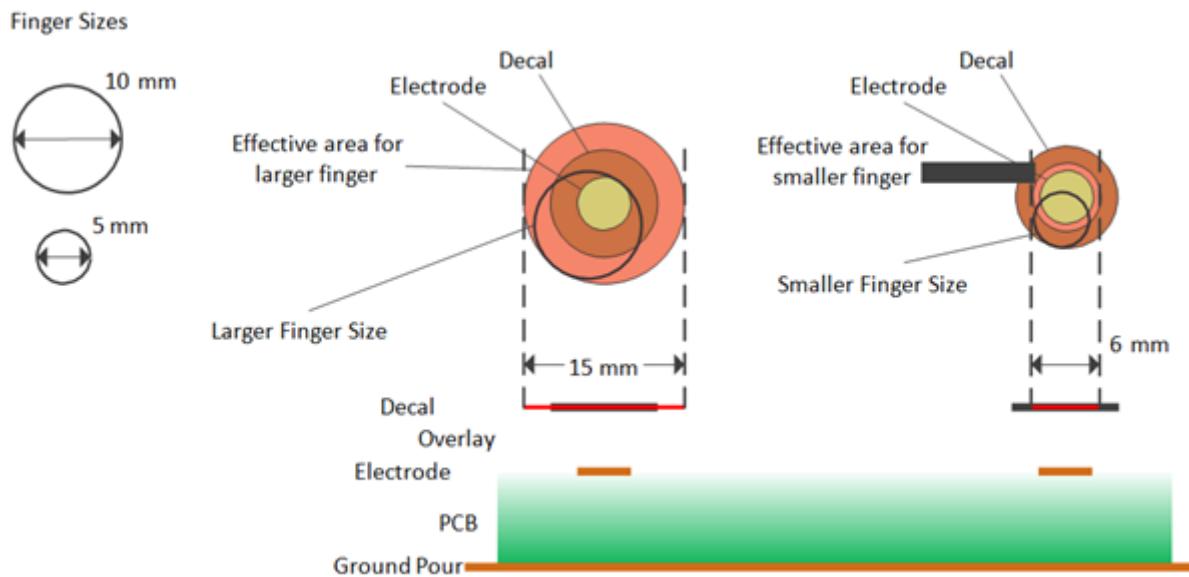


Figure 6.22: Effective Area Example for Electrode Smaller Than Decal

One common mistake is to make the electrode the same shape as the icons printed (in nonconductive ink) on the overlay. As shown in the figure below, this can lead to electrodes with odd shapes that create discontinuities and reduce surface area.

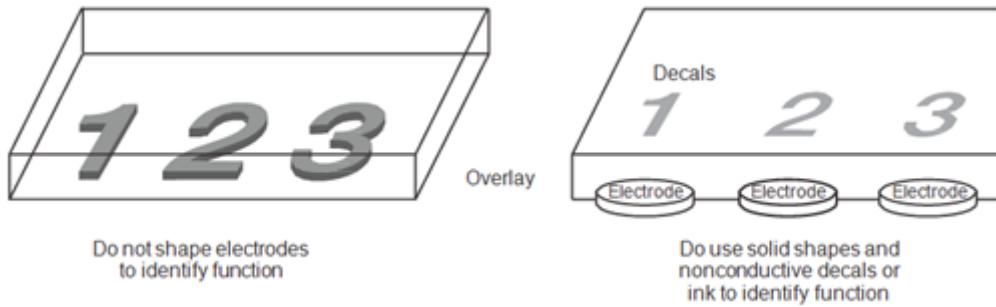


Figure 6.23: Button Shape Examples, Dos and Don'ts

As the distance of the overlay increases, the effective area decreases. Therefore, it is important to keep the button electrode diameter at least three times the laminate thickness.

### 6.5.2 Mutual Capacitive Buttons

A mutual or "projected" capacitive button sensor requires two electrodes, one as a TX and the other for RX. Mutual capacitive buttons are not as sensitive as self capacitive buttons, however, it is possible to pack the electrodes closer together with a low risk of cross talk between neighboring electrodes. With mutual capacitive electrodes, multi-touch is possible by multiplexing the channels. Multiplexing creates up to 64 electrode pairs or "buttons" that can be scanned using only 16 CapTivate™ I/O pins.

Parameter	Guidance
Radiation Pattern	Between TX and RX and ground
Size	Equivalent to interaction
Shape	Various: recommend square or shape with corners
Spacing	0.5 x Overlay minimum

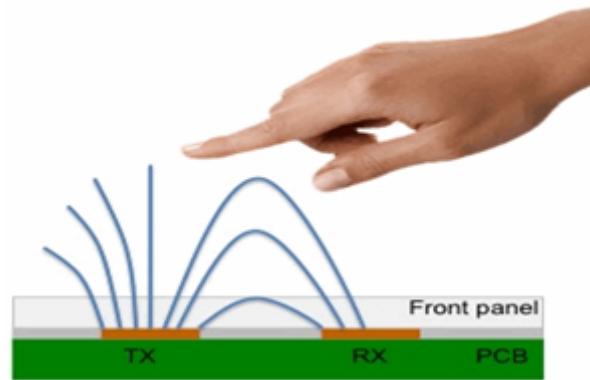


Figure 6.24: Example Mutual Capacitive Button Designs

---

### 6.5.2.1 Mutual Capacitive Button Shapes

The electrode shape is typically rectangular with common sizes being 10mm and smaller. Ultimately, the size will depend on the required touch area. In the diagram below, the TX and RX electrodes are identified and a suggested silkscreen button outline pattern is shown. The position of the vias on the TX and RX electrodes provide flexible signal connection points when routing traces.

- Simple Square Electrode
  - Easiest to layout
  - Highest sensitivity
  - TX on outside, RX on inside
  - For single layer designs, can open TX on one side and feed RX trace through (preferably not in a corner)
  - Better sensitivity if traces on two layers

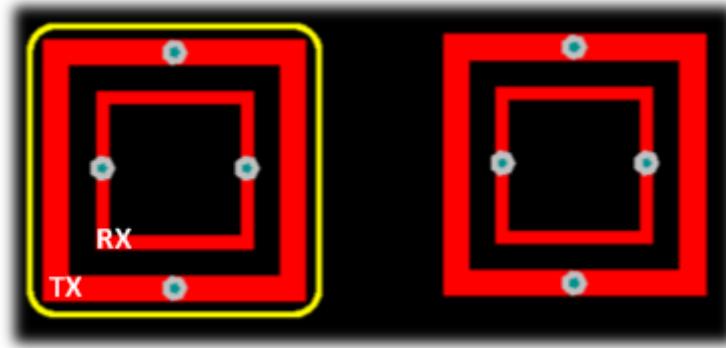


Figure 6.25: Example 10mm Mutual Capacitive Button Design)

A good design practice is to keep the size of the button as small as possible, which minimizes the capacitance and will help with the following:

- Reduce susceptibility to noise
- Improve sensitivity
- Lower power operation due to smaller capacitance and reduced electrode scan time

The dimensions shown on a 10mm x 10mm example button are suitable for overlay thickness up to 2mm.

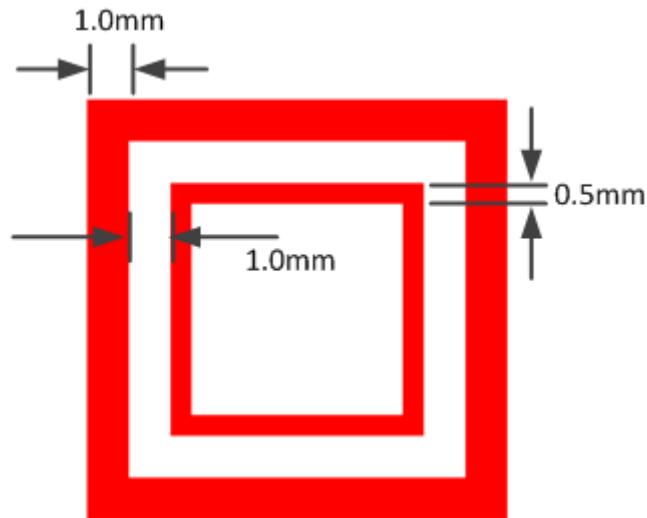


Figure 6.26: 10mm Mutual Capacitive Button Dimensions

An alternative button design, which can provide better sensitivity and is slightly more difficult to create, forces the e-field lines to be concentrated in the four RX corners.

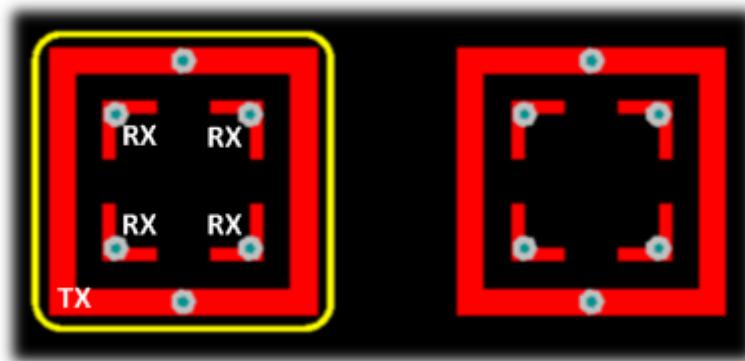


Figure 6.27: Alternative Mutual Capacitive Button Design

The dimensions shown for this 10mm x 10mm example button are suitable for overlay thickness up to 2mm.

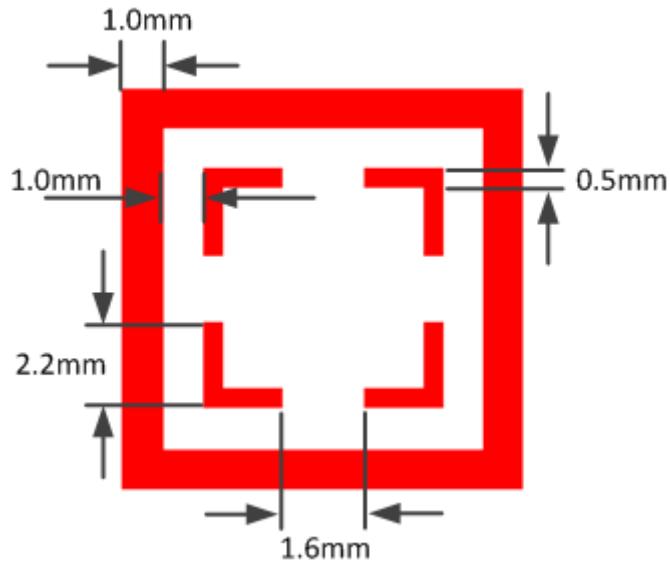


Figure 6.28: 10mm Mutual Capacitive Button Dimensions

To accommodate the individual RX connections in this design, vias on each RX electrode allow the four electrodes to be connected on the bottom PCB layer as shown. The four vias provide convenient connection points when sharing an RX channel with neighboring button sensors or to simplify overall trace routing.

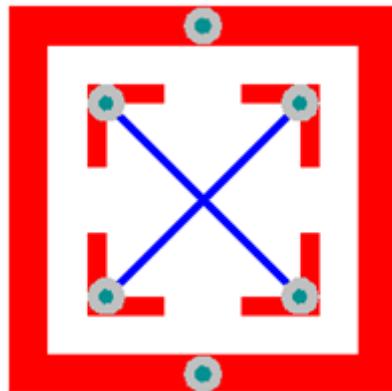


Figure 6.29: RX Electrodes Bottom Layer Connections

#### Mutual Capacitive TX / RX Electrode Spacing Considerations

For thicker overlays, the spacing between the TX and RX is related to the thickness of the overlay and should be maintained at  $0.5 \times$  overlay thickness. The reason is the coupling that occurs between the outer TX electrode and the inner RX electrode is affected by the separation distance between these electrodes. In the diagram below, TX and RX traces that have larger spacing between them will build a field that extends further out compared to TX and RX traces that are closer together. With TX and RX traces close together, it may not be possible to work reliably

---

with a thick overlay. So to accommodate a thicker overlay, the design requires the TX and RX traces to be moved further apart. There may be some design limitations that restrict the overall button size, so experimentation maybe required to meet the TX and RX spacing recommendations.



Figure 6.30: Example Mutual Capacitive Button Designs

As mentioned earlier, mutual capacitive electrodes can be multiplexed with other mutual capacitive electrodes. This means that more than one button can share a common signal. In the 8-button example below, the design of the button and placement of vias allow for easy routing to the neighboring buttons. Depending on the orientation of the routing signals, the buttons could be rotated if needed. Taking advantage of CapTlve™ Technology hardware feature allowing up to four channels to be scanned in parallel, the top four RX channels are measured while TX\_0 is driven. The sequence is then repeated for the bottom row. Scanning four channels in parallel reduces the device's overall power by reducing the scan time by 4x and it should become apparent that the eight buttons were measured with only six CapTlve™ I/O pins.

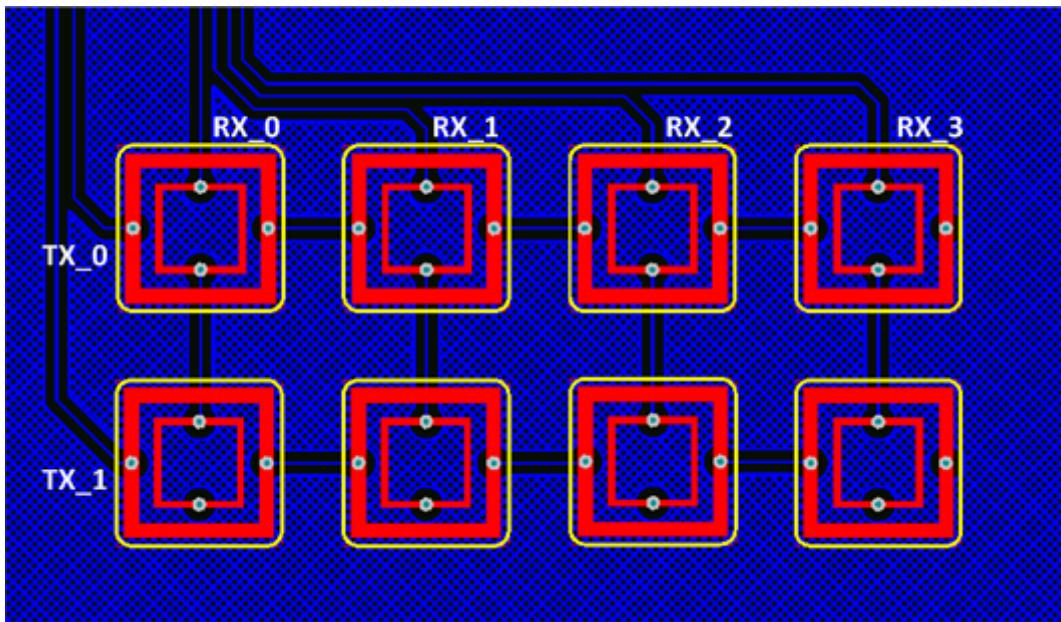


Figure 6.31: 8-Button Multiplexed Mutual Capacitive Example Design

---

## 6.6 Sliders and Wheels

Sliders and wheels are multi-electrode sensors. Sliders and wheels with as few as three or four electrodes can provide excellent performance. The CapTlivate™ Software Library supports both sliders and wheels, ranging from 3 to 12 electrodes. While it is certainly possible to use as many electrodes, the layout becomes more difficult, requires more MCU RX pins and generally does not improve the sensor's performance. A powerful feature of CapTlivate™ Technology is the ability to scan 4 electrodes in parallel. Optimizing a sensor to take advantage of this hardware feature for a slider or wheel provides the best power and measurement time efficiency.

For both sliders and wheels, the area of the electrode is not as critical as the percentage of coverage across multiple electrodes. As shown in the examples below, interdigitated slider and wheel designs provide the most efficient and optimal coupling, but can be complicated to create. Simpler designs are possible, but require experimentation.

For reference, several self and mutual capacitive slider and wheel sensors examples are illustrated throughout this section and are the same sensors used on the CAPTIVATE-BSWP and CAPTIVATE-PHONE demo PCBs.

### 6.6.1 Slider and Wheel Resolution

CapTlivate™ Technology's increased sensitivity combined with the CapTlivate™ Software Library provide exceptional linearity and accuracy for slider and wheel resolutions well beyond 10-bit of resolution. By following the design guidelines and examples that follow in this section, designing a slider or wheel with great performance is easy.

#### A Note About Resolution vs. Number of Positions

For sliders and wheels, the number of discrete positions = the resolution, but because 0 is included as a position, the reported range of positions covers from 0 to (resolution - 1). As an example, if a slider or wheel has a resolution of 1000, the sensor will report 1000 positions, from 0 to 999.

### 6.6.2 Self Capacitive Sensor Shapes

In general, self capacitive sensors are also easy to design and route on a PCB. Due to the nature of self-capacitance, self capacitive sliders and wheels have higher sensitivity compared to mutual capacitive sliders and wheels. For that reason they should be considered as a first choice when designing an application. However, self capacitive sensors require one CapTlivate™ I/O pin for each electrode.

#### 6.6.2.1 Self Capacitive Slider

As mentioned above, a self capacitive slider with great performance can be designed with only three or four electrodes depending the size of the sensor. In fact, a 30cm slider using only four electrodes has been successfully demonstrated with superior linearity and resolution. A basic slider design below uses four RX electrodes. Each electrode is interdigitated and the two end electrodes are electrically connected.

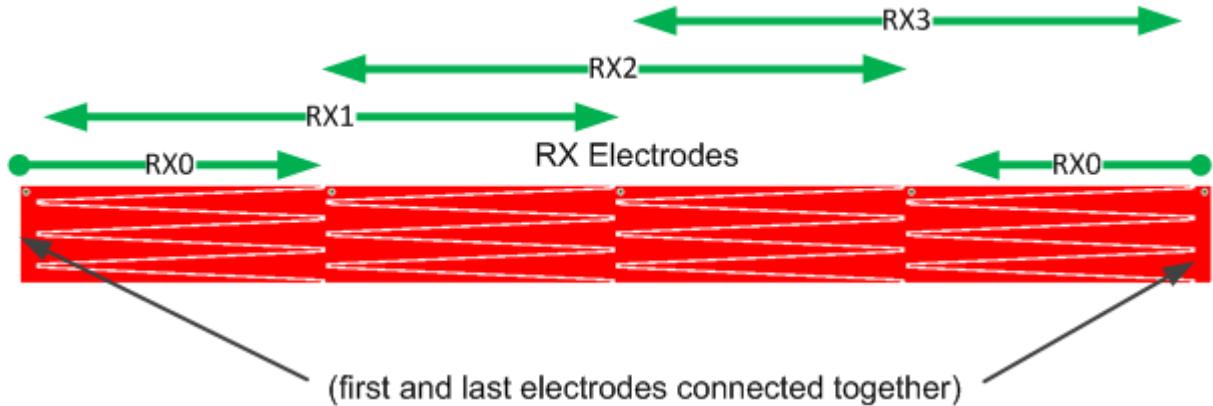


Figure 6.32: Example 4-Element Slider Design

#### 6.6.2.2 Self Capacitive Wheel

A wheel is basically a slider design with both ends wrapped around and connected together. With CapTlivate™ Technology, a self capacitive wheel only needs three elements to provide exceptional linearity and resolution. A basic wheel design below uses three RX channels. Each electrode is interdigitated.

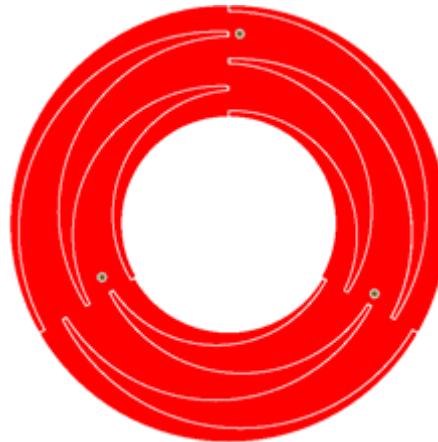


Figure 6.33: Example 3-Element Wheel Design

#### 6.6.3 Mutual Capacitive Sensor Shapes

Mutual capacitance has a unique feature that allows sensors to be multiplexed. Mutual capacitive sliders and wheels can take advantage of this feature by sharing several RX channels with one or more sensors. Because of this, it is possible to have up to 64 electrodes using 16 CapTlivate™ I/O pins. As mentioned earlier, mutual capacitive sensor doesn't have the same sensitivity as a self capacitive slider or wheel. However, with CapTlivate™ Technology, mutual capacitive sliders and wheels will provide the same performance as self capacitive sensors.

#### 6.6.3.1 Mutual Capacitive Slider

The following diagram illustrates a four element slider. The outer TX electrode traces surround the interdigitated RX electrodes and are connected together at both ends of the slider. This simple slider design uses four RX and one TX channels.



Figure 6.34: 4 Element Slider Design

- TX on outside (track on top and bottom)
- RX on inside as triangles
- Spacing from TX to RX is 1/2 the overlay thickness
- 1st and last RX triangle connected to each other
- Spacing between TX and RX ~0.5mm

#### 6.6.3.2 Mutual Capacitive Wheel

The diagram below illustrates a basic 3 element wheel design. The wheel has two TX circle electrodes surrounding the interdigitated RX electrodes. The wheel is basically a slider design with both ends wrapped around and connected together. This wheel design uses three RX and one TX channels.

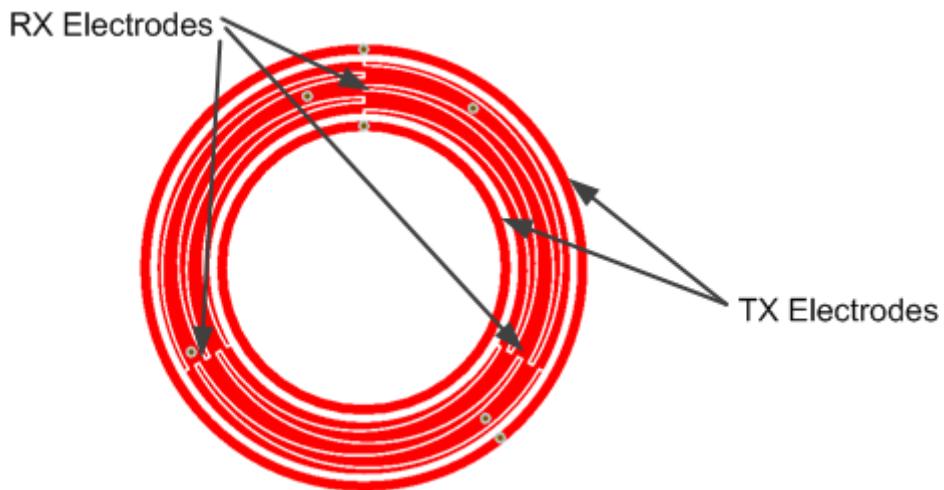


Figure 6.35: 3 Element Wheel Design

- TX Circles on outside and inside of wheel (Need to connect inside and outside together on backside of board)
- RX as interdigitated patterns (similar to Self Capacitance Wheel)
- Spacing between TX and RX ~1/2 overlay thickness
- Ground hatch added inside wheel to provide noise stability. Also ok to have hatching on backside of board. (be careful of total ground loading)

- Possible to include LED/LED backlight in the center
- Difficult to draw

As a variation to the design above, the open portion of the wheel on the top PCB layer can be filled with either a solid or hatched ground to improve noise immunity. Providing an additional hatched ground on the bottom layer of the PCB will also help with noise. Using a solid ground on the bottom layer may present too much capacitive loading, so be careful.

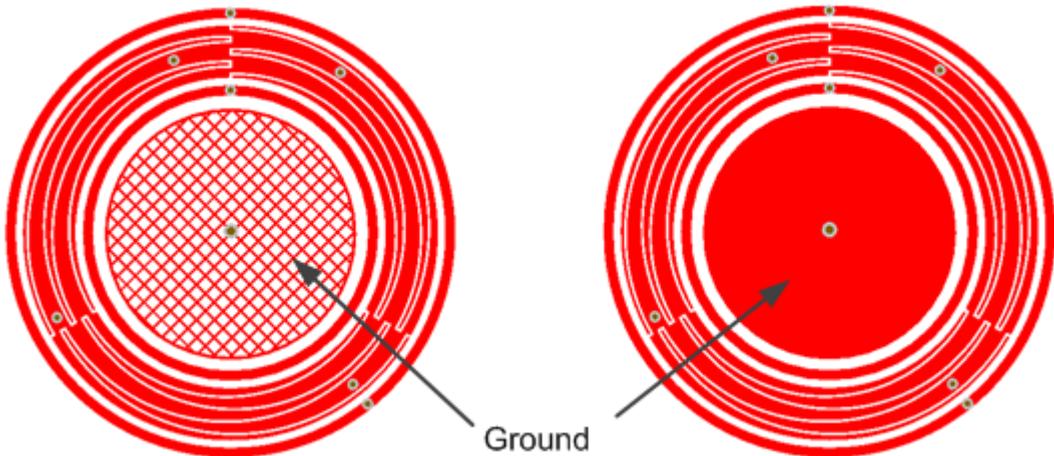


Figure 6.36: 3 Element Wheel Design with Center Grounded

#### Mutual Capacitive Wheel with Center Button

A great feature to add is a separate "button" sensor inside the wheel sensor, if size permits. The design is the same as the basic wheel but a ground separation trace is added between the wheel electrodes and button electrodes. This helps prevent cross talk between the two sensors, should a finger wander slightly off the wheel sensor. This wheel design with center button uses three RX channels and one TX channel for the wheel, one RX and TX channel for the button.

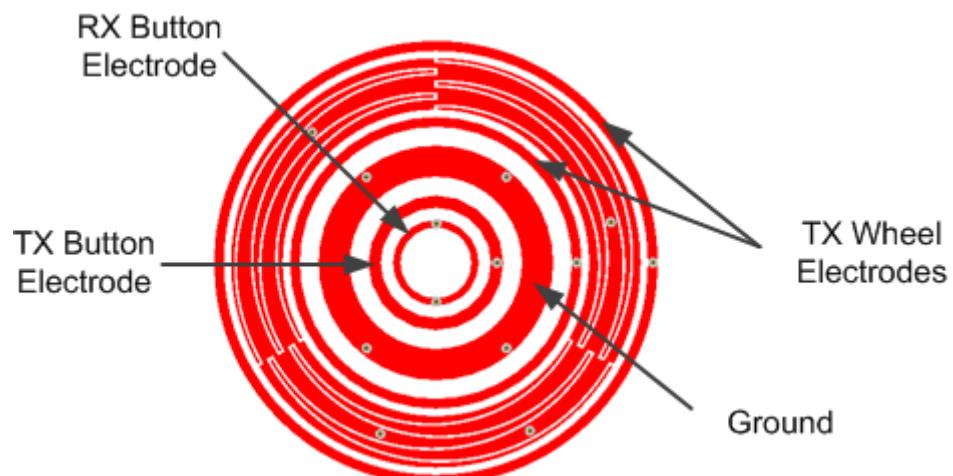


Figure 6.37: 3 Element Wheel Design with Center Button Sensor

---

### Mutual Capacitive Wheel with Center LEDs

Similar to the wheel design with a button in the center, it is simple to add a backlit LED as illustrated in the next diagram. This wheel design with center LED uses three RX channels and one TX channel for the wheel. The LED is driven separately. For additional information about LEDs, refer to [LEDs and Backlighting](#).

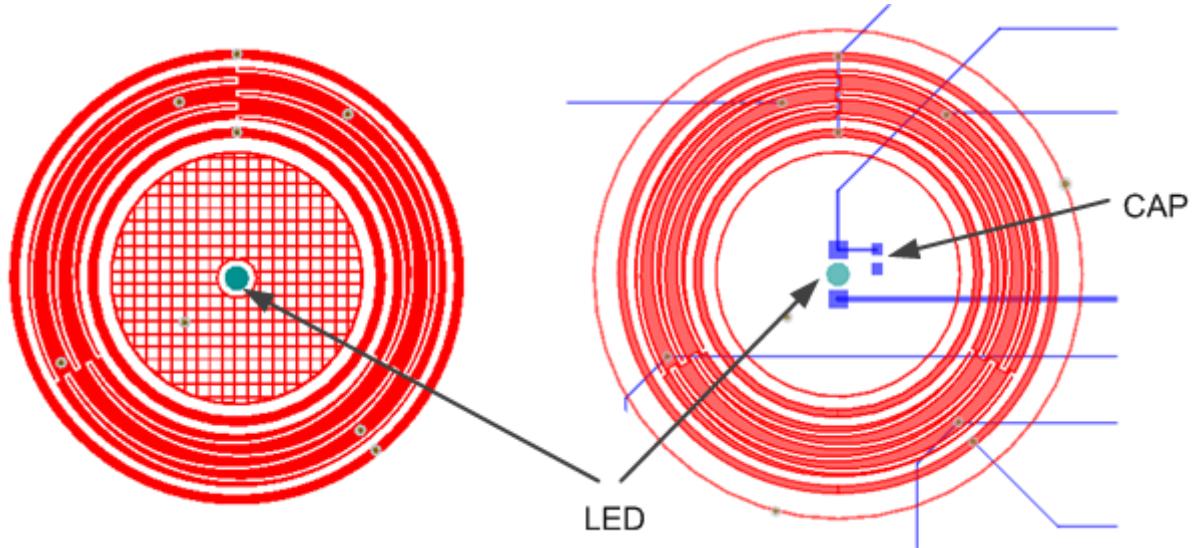


Figure 6.38: 3 Element Wheel Design with Center LED

## 6.7 Proximity

Proximity sensors are electrodes designed to detect a hand or other conductive object at some distance using greater sensitivity compared to buttons, sliders or wheels. For this reason, proximity sensors are self capacitive and can have one or more electrodes.

### 6.7.1 Proximity Shapes

Proximity electrodes can be any shape and for most applications the electrode size is limited by the end product dimensions. As an example, the CAPTIVATE-BSWP Demo PCB with its proximity sensor highlighted in RED is shown below. As mentioned above, proximity sensors require a higher degree of sensitivity than buttons, and this higher sensitivity can be achieved by:

- Increasing the area
- Separation from other conductors.

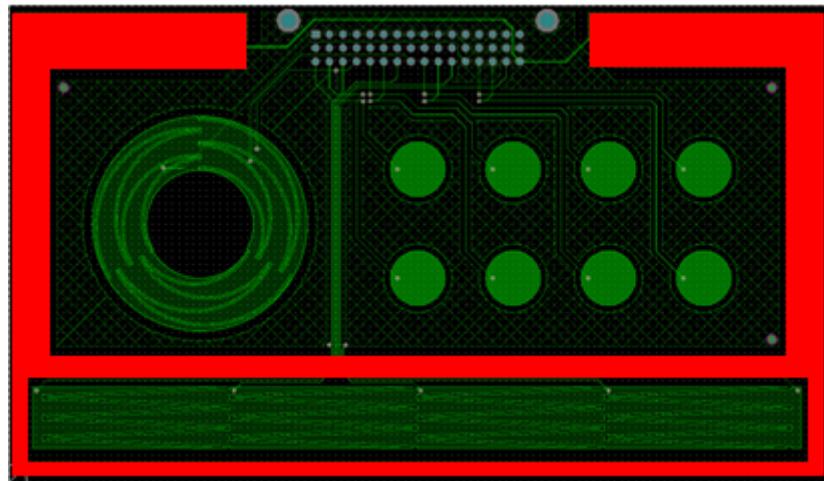


Figure 6.39: CAPTIVATE-BSWP Demo PCB Proximity Sensor

Proximity sensors typically have a larger area in order to detect large surfaces (palm of the hand) at larger distances. When considering the size of the proximity electrode, keep in mind that as size increases, so does parasitic capacitance. And with an increase in capacitance, there is an increase in measurement times which can directly translate into higher power consumption. So there needs to be a careful balance between size, sensitivity and power.

### Proximity Ground and Shielding

In a perfect design, without any parasitic capacitances, a proximity sensor's e-field can extend a great distance, providing incredible sensitivity. The field lines will flow from the electrode to any nearby ground potential and can include a nearby hand or earth ground.

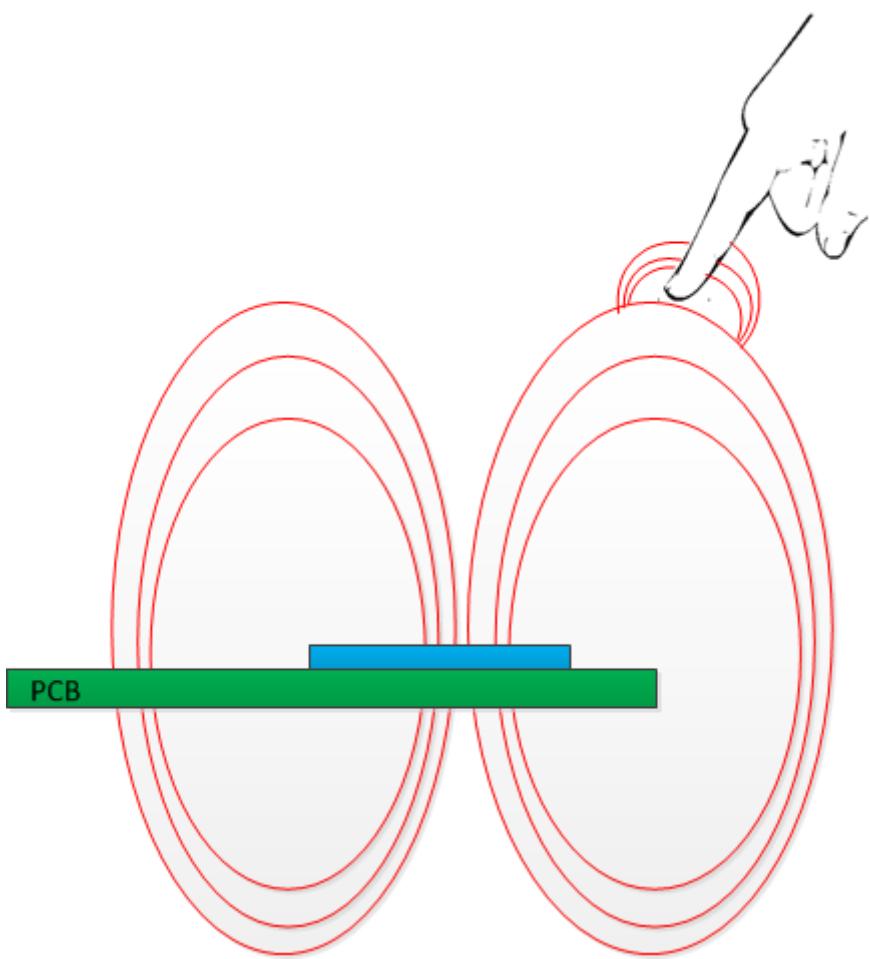


Figure 6.40: PCB with Minimal Parasitic Capacitances

---

Unfortunately, it is almost impossible to avoid having some ground potential on a PCB, so some charge is stolen away from the proximity sensor, reducing the sensor's sensitivity and detection distance.

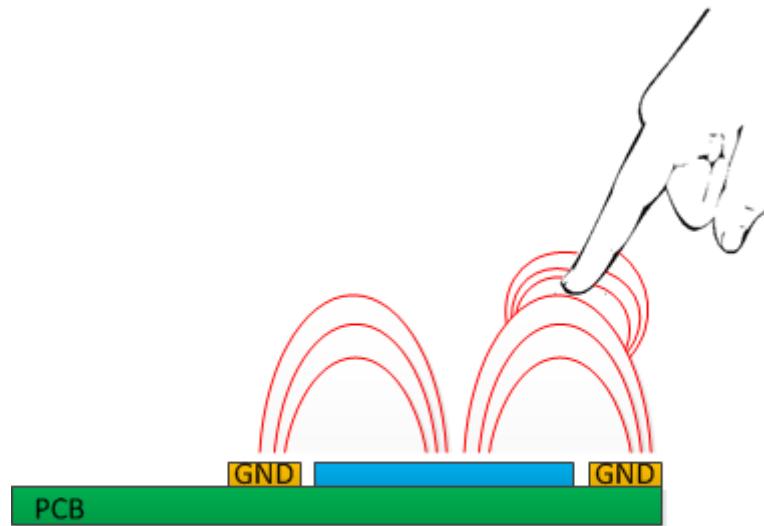


Figure 6.41: PCB with Typical Parasitic Capacitances

One must be careful to avoid designs with excessive ground potentials. In some applications, where electrical noise is an issue, ground planes on top and bottom layers are in place to help shield circuits from potential noise sources. However, these very same ground planes that protect against noise now provide a very easy path to steal even more charge away from the proximity sensor, making proximity detection very difficult or even impossible.

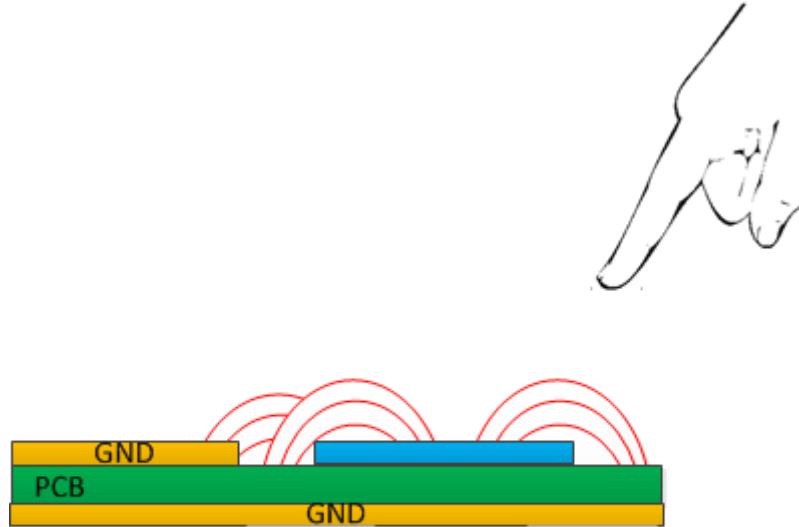


Figure 6.42: PCB with Large Parasitic Capacitances

One method to control the amount of parasitic capacitance to nearby ground is through the use of a driven shield. By replacing the dedicated ground plane with a plane driven by an additional CapTlve™ I/O pin, the plane can be grounded through the I/O pin when the proximity sensor is not active. When driving the proximity sensor, the shield is also driven at the same potential. Any remaining parasitic capacitance to other ground potentials will steal charge from the shield and with minimal impact to proximity sensor's sensitivity. This minimizes the parasitic capacitance and restores the proximity sensor's sensitivity.

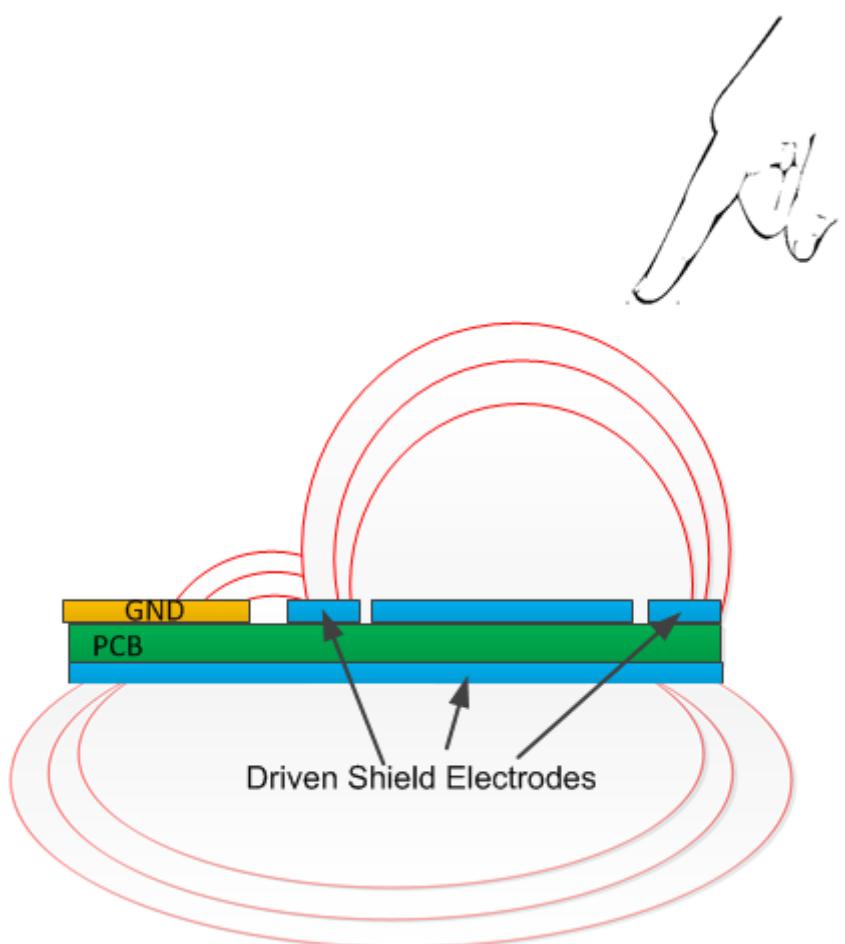


Figure 6.43: PCB with Large Parasitic Capacitances

---

## 6.8 Ultra Low Power

The MSP430 CapTlve™ peripheral is capable of enabling user interface designs with extremely low power consumption. This is possible because the CapTlve™ peripheral includes a processing state machine that is capable of performing the following functions from LPM3 without any CPU interaction whatsoever:

1. Wake up the CapTlve measurement blocks from deep sleep (LPM3) at a periodic interval
2. Start a conversion automatically (1 electrode per measurement block, or 4 electrodes on the MSP430FR2633)
3. Finish the conversion
4. Perform IIR noise filtering (adjustable strength)
5. Perform a threshold crossing detection (adjustable window comparator)
6. If a threshold was crossed, wake up the CPU
7. Otherwise, perform environmental drift compensation (adjustable strength)
8. Shut down the CapTlve measurement blocks and wait until the next interval, then go back to #1

This capability combined with the MSP430 FRAM low power architecture enables designs with average current in the single digit microamps.

### 6.8.1 Introduction

Designing for low power consumption involves optimizing the hardware and the tuning to achieve 5 basic things:

1. The lowest **base LPM3 current** possible
2. The slowest **UI refresh rate** possible (software design)
3. The smallest **parasitic capacitance** possible (hardware design)
4. The shortest **measurement time** possible (hardware and software design)
5. The most effective use of the wake-on-proximity state machine

In short, achieving the lowest possible power consumption is all about optimizing the duty cycle of the application - **minimizing the amount of time that the CPU and high performance analog are awake, and maximizing the amount of time spent in deep sleep.**

### 6.8.2 Expectations

Below are several test cases that have been bench verified with the MSP430FR2633 on the CAPTIVATE-FR2633 processor module.

Sensor Configuration	Mode	Method	Scan Rate	Base Scan Time	LF Clock	I-avg	I-avg / Electrode	Years on AAAs	Years on a CR2032

1 Proximity Sensor	Wake-on-Proximity	Self	8 Hz	420us	Crystal	5uA	5uA	16	3.6
1 Proximity Sensor	Active (CPU)	Self	8 Hz	420us	Crystal	7.3uA	7.3uA	10.9	2.5
1 Button	Wake-on-Proximity	Self	8 Hz	145us	Crystal	3.4uA	3.4uA	23.5	5.3
1 Button	Active (CPU)	Self	8 Hz	145us	Crystal	6.0uA	6.0uA	13.3	3.0
4 Buttons	Wake-on-Proximity	Self	8 Hz	120us	Crystal	3.0uA	750nA	26	6.0
4 Buttons	Wake-on-Proximity	Self	8 Hz	145us	Crystal	3.8uA	0.9uA	21	4.7
4 Buttons	Wake-on-Proximity	Self	30 Hz	145us	Crystal	9.4uA	2.4uA	8.5	1.9
4 Buttons	Active (CPU)	Self	8 Hz	145us	Crystal	8.7uA	2.2uA	9.2	2.1
8 Buttons	Active (CPU)	Self	8 Hz	145us	Crystal	14.5uA	1.8uA	5.5	1.2
8 Buttons	Active (CPU)	Self	15 Hz	145us	Crystal	25.9uA	3.2uA	3.0	0.7
8 Buttons	Active (CPU)	Self	30 Hz	145us	Crystal	50.1uA	6.3uA	1.5	0.3
64 Buttons	Active (CPU)	Mutual	8 Hz	145us	Crystal	109.2uA	1.7uA	0.7	0.2
64 Buttons	Active (CPU)	Mutual	15 Hz	145us	Crystal	203.5uA	3.0uA	0.35	0.08

Note: These measurements were captured using a high resolution source-meter with UART/I2C communications disabled. Battery life was approximated assuming 1000mAh AAA's and 225mAh CR2032 cells with 70% effective usable life.

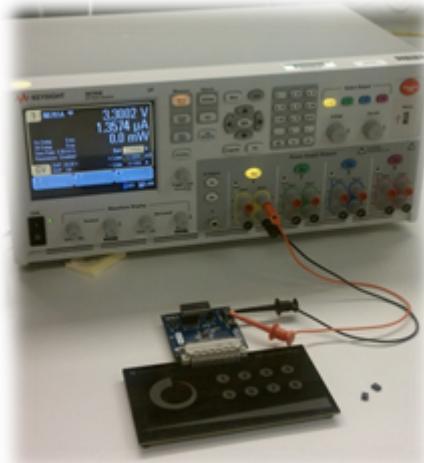


Figure 6.44: Power Characterization Bench

Note how the wake-on-proximity state machine is able to increase the battery life by over a year on a CR2032 coin cell for the proximity sensor, and over 2 years for a basic button.

### 6.8.3 Optimization Steps

The following sections discuss how to optimize the design to achieve the 5 goals introduced above.

### 6.8.3.1 Selecting a Low Frequency Clock

The lowest power consumption achievable is limited by the deep sleep current consumption of the MCU. From there, power consumption will rise with the complexity of the panel and rate at which the panel is scanned. In deep sleep (low power mode 3), since it is not possible to achieve a lower average current than the LPM3 current, the low frequency clock that is selected for the application must be considered. Below are the low frequency clock options available for the MSP430FR2633 MCU.

LF Clock Source	Typical LPM3 Base Current	Typical Frequency	Typical Relative Accuracy	Typical Relative Cost
Crystal	1.2uA	32768 Hz	Best	External Component (Highest)
REFO	16uA	32768 Hz	Good	Internal (Lowest)
VLO	1uA	10000 Hz	Poor	Internal (Lowest)

- If budget and PCB space constraints allow for it, an external 32 kHz crystal provides the best combination of accuracy and low power.
- If accuracy is still required but an external crystal cannot be used, then the REFO will provide a decent 32 kHz time base at the expense of higher power consumption.
- If an external crystal cannot be used and accuracy is not application critical, than the VLO provides the lowest possible power.

### Using the VLO in Wake-on-Proximity Mode

The CAPT\_App layer of the [Starter Project](#) contains a provision for using the VLO as the clock source to the CapTlivate timer. This may be desirable for applications that require very low power but cannot use a crystal for some reason. When the VLO is used as the clock source for the CapTlivate timer, it is possible to run CapTlivate's wake-on-proximity mode while in LPM4, rather than the typical LPM3. When in LPM4, ACLK is turned off completely. This means that if the REFO was the ACLK source, the REFO will also be shut down in LPM4, lowering average power consumption. To enable the VLO+LPM4 combination when in wake-on-proximity mode, define CAPT\_WOP\_VLO\_LPM4 at the top of CAPT\_App.c. This will cause the application low power mode to switch to LPM4 when in the wake-on-proximity state. It will also set up the CapTlivate timer to run off of the VLO when in the wake-on-proximity state. Once the application wakes back up and the UI switches to active mode, ACLK and LPM3 will again be used to ensure timing accuracy.

While the VLO can provide low power consumption without the use of a crystal, it is not as accurate as a crystal. Designers should be aware that the VLO clock frequency can drift considerably with voltage and temperature. See the device specific datasheet for more details. The variance in the VLO clock frequency can show up as a change in the response time of the application. For example, if the frequency drifts low, CapTlivate touch sensors may take longer than usual to respond. Likewise, if the frequency drifts high, sensors may respond faster than usual. In the second case, the power consumption may also be higher than expected because the sensors are being measured more frequently than intended. It is possible to improve the usage of the VLO by implementing a calibration routine that trims the scan interval setting by comparing the VLO to a more accurate clock (such as the REFO).

### 6.8.3.2 Optimizing the Application Scan Period

The application [scan period](#) is the amount of time between each sampling of the user interface. For example, if the period is set to 100ms, then the user interface is sampled every 100ms, or at a rate of 10Hz. If the period is set to 10ms, then the user interface is sampled at a rate of 100Hz. The scan period has a direct effect on the power consumption, as it controls the duty cycle of the application. When a lower period (higher rate) is selected, the CapTlivate analog and possibly the CPU are waking up much more frequently- consuming more power. The design tradeoff here is response time versus power consumption. Scanning less often reduces the power consumption, but also increases the response time of the user interface. The CapTlivate™ starter project implements an active mode scan rate and a wake-on-proximity scan rate. This allows for a longer response time to be used when no one is touching the panel, and a shorter response time to be used when someone is touching the panel. The [CAPTIVATE-BSPW](#) demo panel runs at 10 Hz (100ms response time) when in wake-on-proximity mode, and at 30 Hz (33ms response time) when in active mode.

### 6.8.3.3 Optimizing for the Smallest Parasitic Capacitance

When a sensor has a low [parasitic capacitance](#), it is more sensitive to touch (all else being equal). This means that the sensor may be scanned at a lower resolution to obtain the same resolution and SNR. A lower resolution conversion means a lower measurement time. Battery powered applications typically do not have the same noise immunity requirements as line powered applications, so solid ground fills that limit sensitivity are generally not required. In addition, when sensors have low parasitic capacitance the conversion frequency may be increased while still ensuring good charge transfer.

### 6.8.3.4 Optimizing for the Shortest Measurement Time

The measurement time for each sensor should be optimized to be as small as possible. When the CapTlivate™ analog IP is operational, it draws several 100uA of current. The goal should be to minimize the measurement time so that the analog is only enabled for a brief period of time.

In the CapTlivate™ Design Center, the measurement time is a function of the following factors:

1. Frequency Divider (Conversion clock frequency selection)
2. Phase Lengths
3. Conversion Count (Base number of charge transfers without a touch)

The measurement time may be calculated manually per the following formula. It is also [calculated automatically](#) in the CapTlivate™ Design Center.

$$EQN\ 1 \quad ChargeTransferPeriod[f] = \frac{FrequencyDivider}{CAPOS C[f]} (ChargeHoldPhaseLength + Transfer)$$

$$EQN\ 2 \quad CyclePeriod[f] = ChargeTransferPeriod[f] * ConversionCount + \left( \frac{CAPOS C[f]}{STABLETIME} \right)$$

$$EQN\ 3 \quad SensorPeriod = NrOfCycles * \begin{cases} Noise = true & \sum_{f=0}^3 CyclePeriod[f] \\ else & CyclePeriod[0] \end{cases}$$

Figure 6.45: Scan Time Calculation

### Constants

- STABLETIME=320
- CAPOS C[0]=16000000
- CAPOS C[1]=14700000
- CAPOS C[2]=13100000
- CAPOS C[3]=11200000

If the sensors have fairly low parasitic capacitance, a conversion frequency of 2MHz for self capacitance and 4MHz for mutual capacitance is a reasonable setting. With phase lengths of 1, this is achieved by setting the frequency divider to /4 and /2, respectively.

---

The [conversion count](#) controls the resolution of the measurement. Decreasing this value decreases the overall resolution of the measurement (in terms of counts per picofarad). To optimize the resolution, decrease this value until the minimum acceptable delta due to a touch is reached.

#### 6.8.3.5 Using the Wake-on-Proximity State Machine Effectively

The wake-on-proximity state machine allows for CPU-less measurement and processing for 1 element per measurement block (4 elements total with the MSP430FR2633). Most sensing panels are actually touched <1% of the total time that they are running. 99% or more of the time, they are waiting for a user. The state machine provides a mechanism to handle the latter 99% without the need to involve the CPU, reducing the average current. A sensor may be [selected as a wake-on-proximity sensor](#) in the CapTlivate™ Design Center. When a sensor is selected, the starter project that is generated by the CapTlivate™ Design Center will transition into a wake-on-proximity mode whenever the inactivity timeout counter expires. In wake-on-proximity mode, the first cycle of the sensor that was selected for wake-on-proximity will be automatically measured by the state machine, and the CPU will not receive an interrupt until one of the following occurs:

1. A proximity or negative touch threshold crossing occurs
2. A conversion counter interrupt occurs

The wake-on-proximity feature is best used with longer-range proximity sensors, such as the one on the [CAPTIVATE-BSPW](#) demo panel. However, if an MSP430FR2633 design has 4 buttons or less, it is possible to use the wake-on-proximity state machine functionality to wake up on any of the four buttons!

### 6.8.4 Walking Through the Optimization Process

In this example, the CAPTIVATE-BSPW panel will be used as a piece of test hardware to look at how to optimize an imaginary application that has 4 buttons. From start to finish, going through this exercise will transform the average current of the solution from 100's of microamps to single digit microamps. To run the completed example, check out the [UltraLowPower\\_4Button](#) example project.

#### Step 1: Creating a New Project

To start, a new CapTlivate™ Design Center project will be created that has a button group sensor with 4 buttons. The buttons will be on CAP0.0, CAP1.0, CAP2.0, and CAP3.0. This mapping allows for parallel scanning, which is important for power optimization. All of the default tuning parameters will be kept initially:

Parameter	Value	Optimized?
COMM Peripheral	UART	No
Active Scan Rate	33	No
Wake-on-Proximity Scan Rate	N/A	No
Conversion Count	500	No
Frequency Divider	f/4	Yes

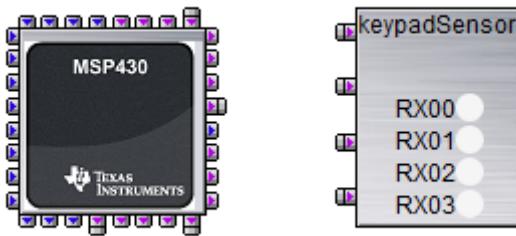


Figure 6.46: Design Center Configuration

Out of all the default configuration values, only the frequency divider is optimized (f/4 with phase lengths of 1 for a conversion frequency of 2 MHz). This yields a starting, un-optimized current of approximately **310 uA-avg**. We can do better than this! Note that LED1 and LED2 of the CAPTIVATE-FR2633 processor module must be disconnected, or the measured power will be higher.

#### Step 2: Switch from UART to I2C

UART requires that SMCLK be active for the baud rate generator to work. By switching to I2C for data transfer, we are able to go to LPM3 instead. This is possible because as a slave, the MSP does not need to provide a clock for I2C to operate.

Parameter	Value	Optimized?
COMM Peripheral	Bulk-I2C (Slave)	Yes
Active Scan Rate	33	No
Wake-on-Proximity Scan Rate	N/A	No
Conversion Count	500	No
Frequency Divider	f/4	Yes

Making this change has lowered the current to **68 uA-avg**. We can still do better! Note that the Design Center must be connected to see these power numbers, otherwise the I2C requests will time out, using more power.

#### Step 3: Optimize the Conversion Count

The project came with a default conversion count of 500. This provides a delta due to touch of about 80 on the 4 buttons. This is more resolution than is needed. Lowering the conversion count to 200 and the conversion gain to 100 (to boost sensitivity) will cut the measurement time more than in half, and still provide a delta of about 30 counts. 30 counts is significantly less, but for button detection in a battery powered application, it is adequate. The default touch thresholds still work as well.

Parameter	Value	Optimized?
COMM Peripheral	Bulk-I2C (Slave)	Yes
Active Scan Rate	33	No
Wake-on-Proximity Scan Rate	N/A	No
Conversion Count	200	Yes
Frequency Divider	f/4	Yes

These changes lower the current to **38 uA-avg**. We are getting there.

#### Step 4: Adjusting the Active Mode Scan Period

The default scan period of 33ms provides good response time, but it uses power. Adjusting this down to 83ms (12 Hz) will reduce the power, but not so much that it compromises the ability to pick up a press.

Parameter	Value	Optimized?
COMM Peripheral	Bulk-I2C (Slave)	Yes
Active Scan Rate	83	Yes
Wake-on-Proximity Scan Rate	N/A	No
Conversion Count	200	Yes
Frequency Divider	f/4	Yes

This change lowered the current to **26 uA-avg**.

#### Step 5: Enabling and Adjusting the Wake-on-Proximity Mode

Now to add the final touches, let's enable the wake-on-proximity state machine functionality to manage states. We only have 4 buttons, with one from each measurement block, so we can measure all 4 buttons and test for proximity without any CPU involvement.

Parameter	Value	Optimized?
COMM Peripheral	Bulk-I2C (Slave)	Yes
Active Scan Rate	83	Yes
Wake-on-Proximity Scan Rate	125	Yes
Conversion Count	200	Yes
Frequency Divider	f/4	Yes

The capture below shows the conversion control tab of the controller customizer, which contains most of the settings that we have been adjusting to optimize for low power.

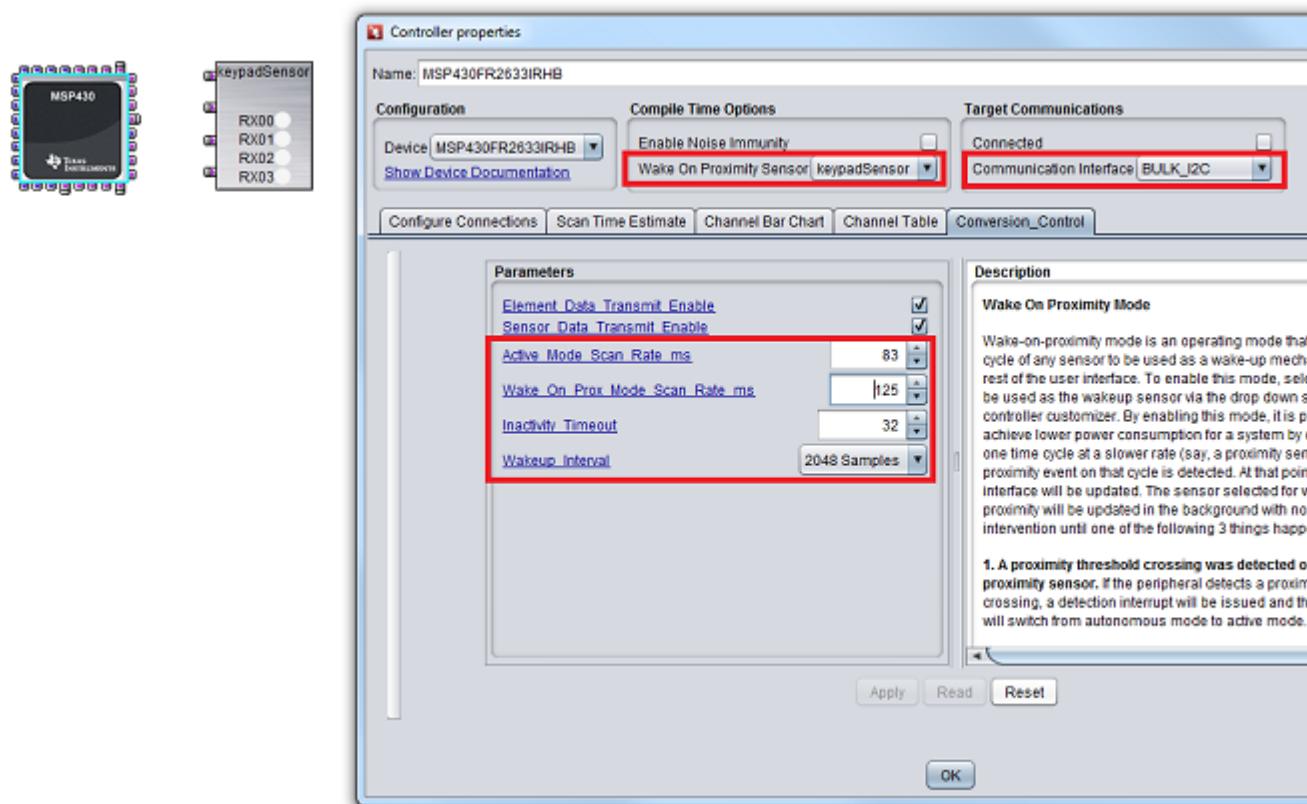


Figure 6.47: Design Center Configuration

This change lowered the current to **3 uA-avg**, or about 750nA per button when no one is touching the panel. As soon as a user touches a button, the power consumption goes up to 23-26uA-avg with I2C reporting. Once the touch is removed, the UI goes back into wake-on-proximity mode after 32 samples and is back to the 3 uA-avg.

---

### 6.8.5 A Note about Mutual Capacitance and Bias Current

Mutual capacitance sensors have an extra tuning parameter: the bias current. During a mutual capacitance measurement, a sample-and-hold amplifier cancels out the effects of the parasitic / stray self capacitance on the receive electrode. This sample-and-hold amplifier has an adjustable bias current capability, which allows for the drive strength of the amplifier to be adjusted to suite the application.

For information on bias current settings how to adjust the bias current, see the [bias current guide](#). Note that the bias current control is only available in [advanced mode](#).

A larger bias current setting should be used when the parasitic capacitance to ground of the Rx electrode is large. The default value is the highest possible bias current, which works for the most applications but requires the highest power consumption. During low power optimization, it is desirable to lower the bias current if the layout allows for it. Most button designs with minimal parasitic capacitance work very well with the lowest bias current setting.

#### Bias Current Optimization Procedure

To optimize the bias current setting for a low power application, begin by setting the bias current to the lowest available setting. Test each electrode in the sensor for sensitivity to touch. One effect of setting the bias current too low is that a touch on a sensor will cause a decrease in counts rather than an increase, because the system cannot accurately compensate for the parasitic self capacitance of the receive (Rx) electrode. If this occurs, continue increasing the bias current until the behavior goes away. Note that the bias current performance also varies with the conversion frequency. Higher conversion frequencies will require higher bias current settings for designs with more parasitic capacitance to ground.

## 6.9 Moisture

This section discusses special considerations for designing products that will come into contact with moisture.

### 6.9.1 Introduction

The performance of a capacitive touch panel can be affected by moisture spray and build-up, as well as liquid spills. In addition, self capacitance and mutual capacitance exhibit different phenomena depending on the nature of the moisture. For example, small water droplets on a self capacitance electrode may have little effect, but a water droplet on a mutual capacitance electrode may have the effect of improving the mutual coupling and creating a change in the measurement counts that goes the opposite direction of a touch. Interestingly still, a water droplet or spill on a self or a mutual capacitance electrode could cause a change in measurement counts in the direction of (rather than against) a touch, if the droplet or spill is also in contact with a nearby ground plane, power plane, or grounded electrode that is not being measured. This section will introduce the physics behind the effects of moisture, as well as some basic best practices to add a level of moisture resistance and detection to an application.

### 6.9.2 Expectations

With careful attention to detail, it is possible to create designs that can reject small amounts of moisture. Through the use of guard channels, it is also possible to create designs that can detect spills and lock out the rest of the interface when a spill is detected.

### 6.9.3 Moisture Physics

Moisture build-ups and liquid spills that occur on panels in the real world affect capacitive sensing electrodes because the liquid on the panel exhibits electrical conductivity. Water is a polarized molecule, and in addition to its polarization, it is rarely pure and typically contains ions in solution. The amount of ions dissolved directly effects

---

the conductivity of the moisture. Distilled water will appear to have much less of an effect than salt water or sweat, for example.

This conductive property means that moisture can appear to have a similar effect as placing a conductor (such as a piece of metal) on a panel. It's easy to understand how that will cause problems! Making things worse is the fact that moisture is unpredictable in multiple aspects:

1. Its actual conductivity is extremely variable
2. The shape and size of droplets and spills are unknown and constantly changing
3. Changes tend to be fast- making them look more like a touch and less like an environmental change

#### 6.9.4 Moisture Problems

At best, build-up of moisture occurs slowly over time and is tracked via environmental filtering. At worst, it happens unpredictably and quickly and triggers a false detection.

#### 6.9.5 Moisture Mitigation Techniques

Because of the unpredictable nature of moisture, it is necessary to divide mitigation techniques into two types of applications: **moisture tolerance** and **spill rejection**.

Application Type	Full Touch Detection	Spill Detection	Guard Channel Required	Typical Mechanical Mounting
Moisture Tolerance	Yes, typically	No	No	Vertical
Spill Detection	No, sensors are masked during a spill	Yes	Typically required	Horizontal

##### 6.9.5.1 Moisture Tolerance Applications

Moisture tolerant applications include those which are able to operate with touch detection in the presence of steam, mist, and spray. Common applications include exterior/interior security panels, E-locks, thermostats, and car access keypads. Whether or not moisture tolerance is feasible for an application depends on the mechanical design and environment. Typically, the following must be true:

1. The touch sensing panel is installed perpendicular to the earth (vertically or at some angle) such that any moisture that accumulates on the panel will drip away on its own due to gravity
2. There is no opportunity for moisture/fluids to "pool" or "flood" the sensing panel

##### Design Tips for Moisture Tolerance

When designing a moisture tolerant application:

1. Provide as much spacing as possible between buttons
2. Provide significant spacing between buttons and nearby ground planes
3. Route all electrode connection traces on the PCB layer furthest from the surface
4. Set sensor **idle states** to High-Z (floating) so that nearby sensors don't provide coupling points that can cause false detections
5. If possible, use a non-conductive enclosure for the product

---

### 6.9.5.2 Spill Rejection Applications

Spill rejection applications include those which are able to detect when a fluid spill has occurred on the sensing panel. Common applications are white goods such as a cooking surface control panel. The goal of spill rejection is not to enable full touch detection (which is extremely difficult when moisture or fluids cover multiple keys and/or ground), but simply to detect the presence of a spill and lock out the keypad until the spill is cleared by a user. This is most commonly achieved with the use of a guard channel, whose purpose is to detect large objects as well as spills.

#### Design Tips for Spill Detection

When designing a spill detection application:

1. Implement a guard channel to detect when a large unknown object or some kind of spill comes into contact with the sensing panel
2. In software, utilize the guard channel to mask reporting of touches on other keys when a spill is detected.

The [CAPTIVATE-PHONE](#) demonstration panel has a guard channel that is used for proximity sensing, palm rejection, and basic moisture rejection.

## 6.10 Noise Immunity

This section discusses how to design for electromagnetic compatibility.

### 6.10.1 Introduction

Capacitive touch sensing involves the measurement of very small changes in capacitance of a sensing electrode. These changes are often on the order of a picofarad or less. Because the quantity being measured is so small, capacitive touch circuits that are going to be used in noisy environments must be designed with noise immunity in mind from the start. Often, potential noise sources may not even be known at the time of design. This chapter is dedicated to providing the necessary guidance to enable immunity to various types of noise. It introduces the most common noise threats to a capacitive touch system. For each threat, it discusses how to reduce the susceptibility of a capacitive touch circuit to that threat when using MSP CapTlve™ MCUs.

#### 6.10.1.1 How to Use This Section

If you are designing a system that will exist in a high noise environment, it is best to read and understand this guide before going through the process of schematic capture and layout design. It is strongly recommended that the [Capacitive Sensing Basics](#) chapter be read before reading this chapter.

The [introduction](#) section of the noise immunity guide establishes a knowledge base that will be used as a backdrop for the rest of the discussion. After the [introduction](#), the following section will discuss several different [noise problems](#) (RF susceptibility, for example). Each problem will be introduced so that the phenomenon is understood. Finally, the last section will discuss the [noise mitigation steps](#) to apply to your design to provide robustness to all of the noise types discussed. The noise mitigation steps will be presented in the context of the [Three Sided Approach](#). The [noise mitigation steps](#) section is the most important section, as it provides a check list of considerations to apply when designing for noise immunity. If you don't read the entire noise immunity guide, be sure to read the [noise mitigation steps](#).

#### 6.10.1.2 Additional Resources

In addition to the design documentation here, there are several TI Designs and literature available that demonstrate how to achieve noise immunity:

- Enabling noise-tolerant capacitive-touch HMIs with MSP CapTIVate technology
- Noise Tolerant Capacitive Touch HMI Reference Design TIDM-CAPTOUCHEMCREF
- Capacitive Touch Thermostat UI with MSP MCUs Featuring CapTIVate Technology TIDM-CAPTIVATE-THERMOSTAT-UI

#### 6.10.1.3 Aggressor-Victim Philosophy

An aggressor-victim approach is taken to understand noise in a system. The diagram below illustrates this concept.



Figure 6.48: Aggressor-Victim Diagram

The aggressor is the noise source, which generates interference that affects the victim through the coupling medium. Generally speaking, for noise to couple from the aggressor to affect the victim, the following must be true:

- The noise must be at a frequency at which the victim is vulnerable
- The noise must have enough amplitude to affect the victim
- The noise must be present at a time when the victim is vulnerable to the noise

#### Aggressor

The aggressor varies with every application. Common aggressors are poorly designed power supplies with high common-mode and differential-mode emissions, long cables and/or traces that act as antennas to radio frequency (RF) signals, and high-current switched inductive loads that share their power source with the victim at some level. In some cases, the viewpoint is such that the capacitive touch circuit is the aggressor.

#### Coupling Medium

The medium through which the noise travels from the aggressor to the victim is often misunderstood. Common mediums include the VCC rail, the VSS rail, the capacitive coupling between traces (in the case of crosstalk), and even users of the capacitive sensing interface themselves.

#### Victim

During most of the discussion in this chapter, the capacitive touch circuit is the victim. However, there are times where the capacitive touch circuit becomes the aggressor, at which time a neighboring subsystem may become the victim.

#### 6.10.2 The Three Sided Approach

Based on this understanding, it becomes clear that to reduce the effects of the aggressor on the victim, it is necessary to fulfil one or more of the following objectives:

- Change the characteristics of the noise emitted by the aggressor to either change its frequency or reduce its amplitude to a level that does not affect the victim
- Reduce the ability of the noise to couple from the aggressor to the victim by modifying the coupling medium
- Reduce the susceptibility of the victim to the noise emitted by the aggressor

---

In some cases, it is possible to reduce the noise emitted by the aggressor (per the first bullet above). This is always the best course of action. Reducing the noise simplifies PCB layout and software considerations, and reduces the potential for other issues in the system. This is typically accomplished by designing robust power supplies and IO interfaces that do not generate their own noise nor pass on noise from the outside world. However, many times this is not possible because the aggressor may be outside the control of the system being designed. In that case, a solution must be obtained by **reducing the coupling medium** and **reducing the victim's susceptibility**.

Along these lines, to achieve a high level of noise immunity in a capacitive touch design it is necessary to apply the relevant CapTlve™ peripheral features, hardware design principles, and signal processing tools. The basic tools available are listed below.

### 1. CapTlve™ Peripheral EMC Features

- Integrator-based charge transfer method
- Offset subtraction signal chain block
- Frequency hopping
- Spread-spectrum clocking
- Synchronization input pin to align conversions with AC mains zero-crossing events

### 2. Hardware Design Principles

- Grounding techniques
- Filtering techniques
- Protection passives
- Noise sensing channel

### 3. Signal Processing Tools

- Multi-frequency processing (MFP) algorithm
- Dynamic threshold adjustment (DTA) algorithm
- IIR noise filtering
- De-bounce

The general theme of this chapter is how these three main toolboxes (the **CapTlve™ peripheral**, **hardware design principles**, and **signal processing**) may be applied together to solve several noise problems at the same time.

#### 6.10.2.1 Introduction to Electromagnetic Compatibility (EMC) Standards

International standards exist to allow for consistent, repeatable electromagnetic compatibility testing to be applied to different products. The primary international standard concerning system level robustness in the area of EMC is the IEC 61000-4 family. The tests most relevant to capacitive touch circuits are introduced below, and will be addressed in detail later in this chapter.

1. [IEC 61000-4-2 Electrostatic Discharge \(ESD\)](#) describes testing for electrostatic discharge immunity.
2. [IEC 61000-4-3 Radiated RF Noise](#) describes testing for immunity to radiated RF noise between 80 MHz and 1 GHz.
3. [IEC 61000-4-4 Electrical Fast Transient/Burst \(EFT/B\)](#) describes testing for immunity to high voltage fast transients.
4. **IEC 61000-4-5 Surge** Describes immunity to power surge energy levels. This is not discussed here as it is considered a system-level problem that is the responsibility of the power supply.
5. [IEC 61000-4-6 Conducted RF Noise](#) describes testing for immunity to conducted RF noise between 150kHz and 80 MHz.
6. [IEC 61000-4-16 LF Conducted Noise](#) describes testing for immunity to conducted disturbances in the frequency range of 0 Hz to 150 kHz.

---

### 6.10.2.2 Signal-to-Noise Ratio

The signal is the change in capacitance that results in a meaningful change in counts. Noise, on the other hand, is any disturbance that does not change the capacitance but does change the counts. Most often these disturbances are the result of power supply switching noise, electrostatic discharges (ESD), electrical fast transients (EFT), radiated RF noise, or some other type of electrical noise that couples into the system.

SNR is a system-level specification and needs to be tested at the system level. In achieving an acceptable level of SNR, the competing requirements for noise and signal must be addressed. The most common example of competing requirements is seen with ground shields. Ground shields help reduce susceptibility to radiated and conducted interference, and the closer these structures are to the circuit being protected, the more effective they are. However, moving these structures closer to the capacitive touch circuit has an unwanted effect of increasing the parasitic capacitance and consequently lowering the sensitivity.

In addition to managing the competing goals of sensitivity and noise immunity, these goals need to be managed within the overall requirements of the end product. This can mean trying to achieve a solution in a limited space, under a large aesthetic housing, or in the presence of other electrical activity (noise).

### 6.10.3 Types of Noise

This section introduces the major types of noise that a capacitive sensing system may encounter. For each topic, the problem, MCU solution, and required design actions are discussed.

#### 6.10.3.1 Differential Mode Supply Rail Noise

Differential-mode supply rail noise is any noise seen at the DVCC supply rail of the capacitive touch MCU with respect to the DVSS (circuit common) supply rail. Some touch solutions on the market require a dedicated LDO just for the capacitive touch circuit to reduce these issues.

##### 6.10.3.1.1 Problem

Some of the problems associated with a direct reference to DVCC include:

1. Changes in sensitivity as DVCC drifts down as a battery supply discharges
2. Vulnerability to false detects from ripple voltage on the DVCC supply rail
3. Increased vulnerability to EFT spikes on the DVCC supply rail

##### 6.10.3.1.2 CapTivate™ MCU Solution

To address these issues, the CapTivate™ peripheral has low drop-out regulator that is dedicated to the capacitive sensing analog circuitry. This regulator provides decoupling from the supply rail (DVCC) as well as digital noise on-chip. Issue 1 above is resolved immediately, as the capacitive touch analog circuitry will always run at a constant, internally regulated voltage that does not drift with DVCC. The regulator is functional across the DVCC supply voltage range of the MCU (see the device datasheet for details). Issues 2 and 3 above are dramatically reduced as a result of the power supply rejection ratio of the dedicated regulator. While a dedicated regulator is not required, it does provide a compounded PSRR in very noisy systems and is still recommended.

##### 6.10.3.1.3 Design Actions Required

In general, no specific action is required- the on-chip regulator provides the benefits described above.

#### 1. CapTivate™ Peripheral EMC Features

- Dedicated CapTivate™ LDO provides a constant voltage across the DVCC input range, and provides a level of PSRR against ripple voltages

#### 2. Hardware Design Principles

- 
- Follow all recommendations for DVCC decoupling capacitance (per the device-specific datasheet).
  - Follow all recommendations for the dedicated voltage regulator (VREG) tank capacitor (per the device specific datasheet).
  - Place capacitors as close to the IC as possible.

### 3. Signal Processing Tools

- [De-bounce](#) may be applied as necessary if noise spikes are of significant amplitude to be seen in the measurement.

#### 6.10.3.2 Electrostatic Discharge (ESD)

Electrostatic discharge, if not properly managed, can disrupt a sample or even damage the MCU permanently. Fortunately, many tools exist to prevent this.

##### 6.10.3.2.1 Problem

An electrostatic discharge involves a sudden flow of current between two charged objects. The flow of current may be caused by the breakdown of a dielectric material (such as plastic or even air), or direct contact. The voltage differential in a discharge may be on the order of several kilovolts (kV) or more. An electrostatic discharge applied to a device pin can permanently damage the device. Since capacitive touch panels by definition are interacted with by users or other objects that may be electrically charged, ESD deserves consideration during the design process.

##### 6.10.3.2.2 CapTivate™ MCU Solution

The primary line of defense against ESD is always the touch panel overlay material. The overlay provides electrical isolation between the conductive electrode and any user or outside object that might interact with it. The ability of the overlay material to defend against ESD damage is dependent upon two main factors- the thickness of the material and its breakdown voltage, which is usually specified in terms of breakdown per unit of thickness. It is important to note that even if a discharge breaks down the dielectric material, the stress observed at the electrode (and possibly the device IO) will be less than the original stress on the overlay itself (since the breakdown accounts for some of the energy). Different materials will have different breakdown specifications. Generally speaking, plastics such as acrylic or polycarbonate have a higher breakdown voltage than materials such as glass. However, glass has a higher dielectric constant, which allows for better sensitivity through thicker overlays, allowing for both materials to be used easily.

Care should be taken to account for all places in a system that a discharge could slip through (say around the bezel of a product or through a ventilation gap). This is generally of more concern with products that have plastic enclosures than with those that have metallic enclosures that can be tied to an ESD ground or safety ground.

If it is not possible to obtain the needed protection mechanically, the next best alternative is to employ the use of series resistors on electrode pins, followed by low-capacitance TVS diode clamps if needed.

##### 6.10.3.2.3 Design Actions Required

###### 1. CapTivate™ Peripheral EMC Features

- Internal protection ESD clamps on the IO provide some chip-level protection during device handling and product assembly, but this should not be solely relied on for system level ESD protection.

###### 2. Hardware Design Principles

- Populate a 470-1k ohm resistor in series with the electrode to provide ESD protection, with a protection clamp such as a TVS diode placed between the electrode and ground (return) on the electrode side of the resistor.
- Follow recommended [hardware best practices](#) for noise immunity

###### 3. Signal Processing Tools

- [De-bounce](#) of at least one sample is recommended to reduce the possibility of a false detection due to a discharge on an IO during a conversion. ESD transients are fast, and in theory do not affect multiple samples in a periodic way (like RF interference, for example).

### 6.10.3.3 Electrical Fast Transient (EFT)

Electrical fast transients are common in industrial environments as well as within commercial products that contain switching inductive loads. These transients are similar in nature to ESD, but exhibit a lower magnitude and longer event length. When high-current inductive loads are switched on and off, high voltage transients may appear on power supply lines. Transients may also appear on signal lines if a system is connected to another system that is generating noise. The transients are a result of the inductance in the load that attempts to maintain the flow of current, and in so doing generates large voltages.

The IEC test simulates these threats by injecting bursts repetitively into the supply or signal lines of a system, in differential or common mode (though common mode is the most typical).

#### 6.10.3.3.1 Expectations

High voltage transients are one of the issues that must be considered ahead of time when designing the system. They can wreak havoc on simple analog circuits like the MCU reset circuit. However, with proper design techniques it is possible to realize designs with immunity to 4kV per the IEC 61000-4-4 test specification.

#### 6.10.3.3.2 Problem

Fast transients can create effects similar to that of conducted noise, but generally only for brief periods of time as the duration of a transient is often quite short (on the order of 50 nanoseconds). This short burst time often means that not every capacitive touch sample will be affected by a burst. The figure below is data captured from a self capacitance button exposed to 1.5kV EFT bursts that were 50ns each. The 50ns transients were repeated at a 5 kHz burst rate for 15ms. That pattern was then repeated every 300ms, per the IEC test specification. The different effects observed on the electrode can be attributed to the amount of time a sample may have overlapped with a 15ms burst window. There is about a 5% chance of a sample overlapping with a burst, due to the 300ms test period. It is important to note, however, that real world threats will likely not follow this pattern! The IEC test should be used as a tool to understand the implications of EFT on a system- but real world testing in-application is just as if not more valuable.

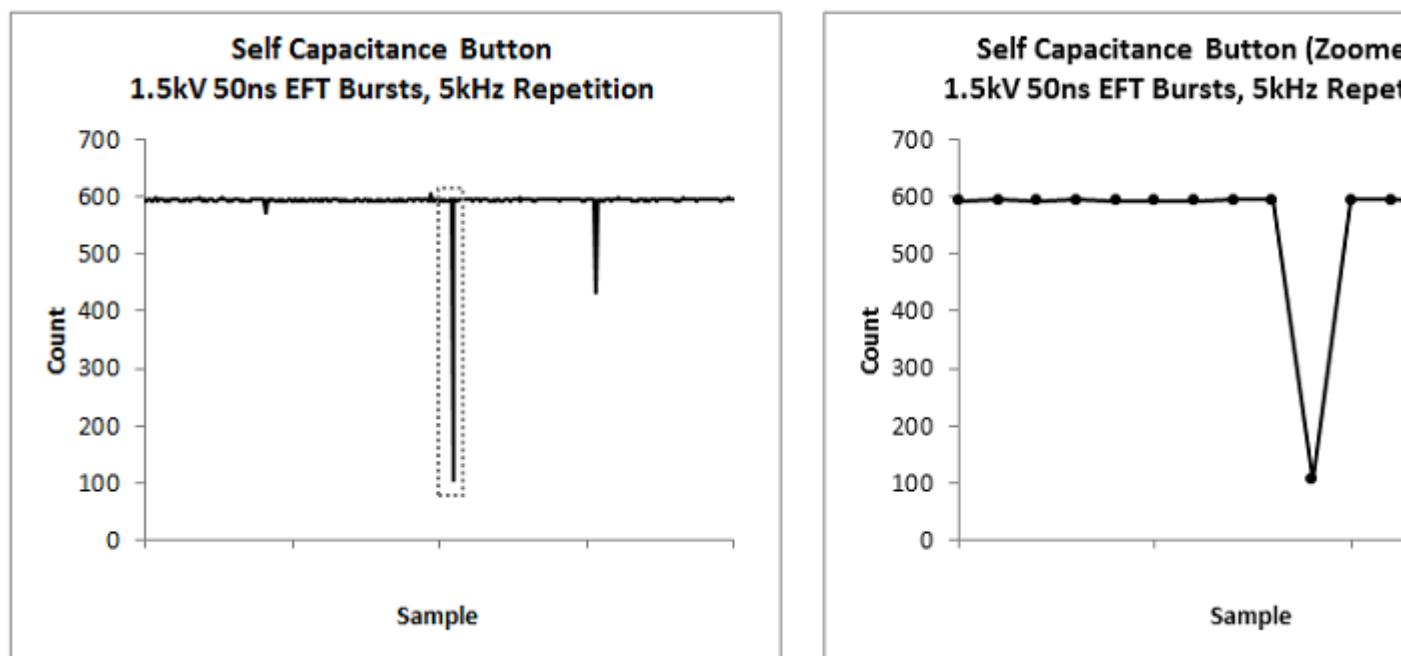


Figure 6.49: Self Capacitance Button 1.5kV EFT Test

#### 6.10.3.3.3 CapTivate™ MCU Solution

The primary line of defense against EFT must be hardware design. The best place to stop transients is at the power supply before they have a chance to effect different components of a product. Chances are good that the

capacitive sensing MCU will not be the only component that is vulnerable to fast transients. Supplementing a good power supply, software should be EMC hardened. This includes de-bouncing the device reset pin, as well as implementing a watchdog timer. If EFT is a concern in a safety application, memory integrity self-tests should also be applied periodically for robustness. Heavy de-bounce of the capacitive touch sensor is also extremely valuable-almost more so than IIR filtering in an EFT environment. Take the figure below as an example. The EFT burst only affected a single sample- something a basic de-bounce can easily handle. However, the magnitude of the pulse was significant enough to pull down the IIR filter for more than one sample, overwhelming a de-bounce level of 1 and requiring a de-bounce level of 2+ or a weaker filter coefficient.

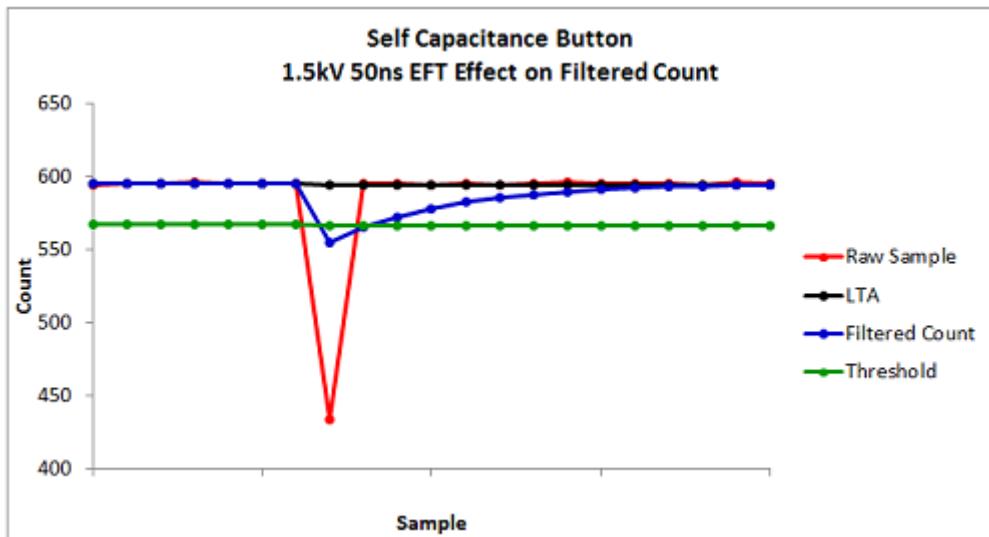


Figure 6.50: Self Capacitance Button 1.5kV EFT Test - Filter Ineffectiveness

#### 6.10.3.3.4 Design Actions Required

##### 1. CapTivate™ Peripheral EMC Features

- Integrator-based charge transfer sensing technology

##### 2. Hardware Design Principles

- If possible, design a robust power supply for the system that blocks high frequency transients
- Follow the same layout techniques as presented in the RF immunity section

##### 3. Signal Processing Tools

- De-bounce
- Re-purpose the reset pin as a non-maskable interrupt (NMI) to de-bounce potential IC reset conditions.

#### 6.10.3.4 RF Susceptibility (Conducted and Radiated)

Capacitive touch circuits are inherently vulnerable to RF interference, since it can lead to injected noise currents into transmit (Tx) and receive (Rx) sensing IOs. By applying a combination of peripheral features, hardware techniques, and signal processing, it is possible to realize capacitive touch designs that are robust in the presence of RF interference.

RF interference will be broken down into two categories (conducted and radiated) based upon the frequency of the interference. The distinction between the two is important, as it defines the nature of the coupling medium between the aggressor and the victim. The breakdown is as follows:

- Conducted RF Interference
  - 10's of kHz to 10's of MHz

- Long wavelengths
- PCB traces are not efficient antennas
- Noise is generally assumed to be coupled into the system via the power supply or through IO cables
- Cables are of sufficient length to be efficient antennas
- Generally of the most concern for capacitive touch circuits since the frequency range overlaps with the charge transfer frequency range
- Radiated RF Interference
  - 100's of MHz to GHz
  - Short wavelengths
  - PCB traces can be efficient antennas
  - Generally of less concern for capacitive touch circuits

#### 6.10.3.4.1 Expectations

In general, self capacitance sensors will allow for a higher level of noise immunity to be achieved than mutual capacitance sensors. This is a result of the fact that self-capacitance measurements are looking for larger changes (a few pF), whereas mutual capacitance measurements are looking for small changes (less than 1pF). In addition, to realize a mutual capacitance measurement the self-capacitance effect of the Rx must be eliminated from the measurement. This is achieved through the use of a sample-and-hold amplifier which compensates for the receiver (Rx) to ground capacitance to remove it from the measurement. If strong enough noise currents exist, this sample and hold amplifier can become overwhelmed to the point where it is not able to stabilize the Rx. When this happens, a self-capacitance effect (an overall decrease in counts) is observed. For this reason, self capacitance is the recommended method for high levels of immunity above 3Vrms. At or below 3 Vrms (the commercial product test level), mutual capacitance is a very viable sensing method with many advantages, such as higher key density.

The following table breaks down what kind of immunity can be expected when testing a system to the IEC 61000-4-6 specification.

**Class A Immunity Table (Fully Functional in Presence of Noise)**

Sensing Method	Sensor Type	Achievable IEC 61000-4-6(Conducted) Level
Self	Buttons	Level 3 (10V RMS)
Self	Sliders/Wheels	Level 3 (10V RMS)
Self	Proximity	Level 1 (1V RMS)
Mutual	Buttons	Level 2 (3V RMS)
Mutual	Sliders/Wheels	Level 1 (1V RMS)
Mutual	Proximity	Not Recommended

#### 6.10.3.4.2 Conducted RF Noise Test System

Conducted noise immunity is most reliably tested with the coupling/decoupling network method as specified in IEC 61000-4-6. A block diagram of the test configuration is shown below.

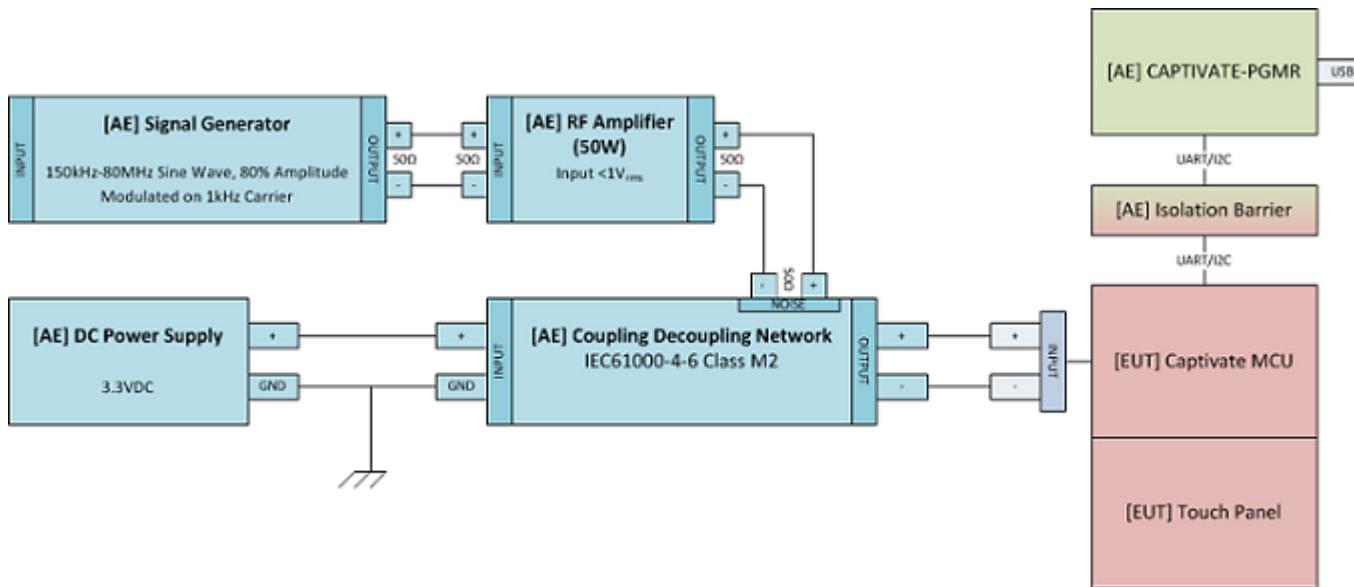


Figure 6.51: Conducted RF Test System

The frequency range tested is typically 150 kHz to 80 MHz, amplitude modulated at 80% depth on a 1 kHz carrier frequency to simulate real threats.

#### 6.10.3.4.3 Problem

The problem with RF interference is that it leads to injected currents that can couple into receive (Rx) electrodes. Typically, the currents are common mode in nature with respect to some reference (usually earth ground). The adverse effects observed on capacitive touch circuits are different for self capacitance than for mutual capacitance, so the two will be treated separately in the following section.

##### Self Capacitance

Self capacitance sensors experience several different effects in the presence of noise depending upon the following factors:

1. Frequency of the noise.

- If the noise is centered directly on the conversion frequency or its lower harmonics (+/- 50 kHz), the measurement data will be corrupted positively and negatively, as shown below. This occurs whether or not a touch is present, although the effect is amplified when a touch is present (as shown).

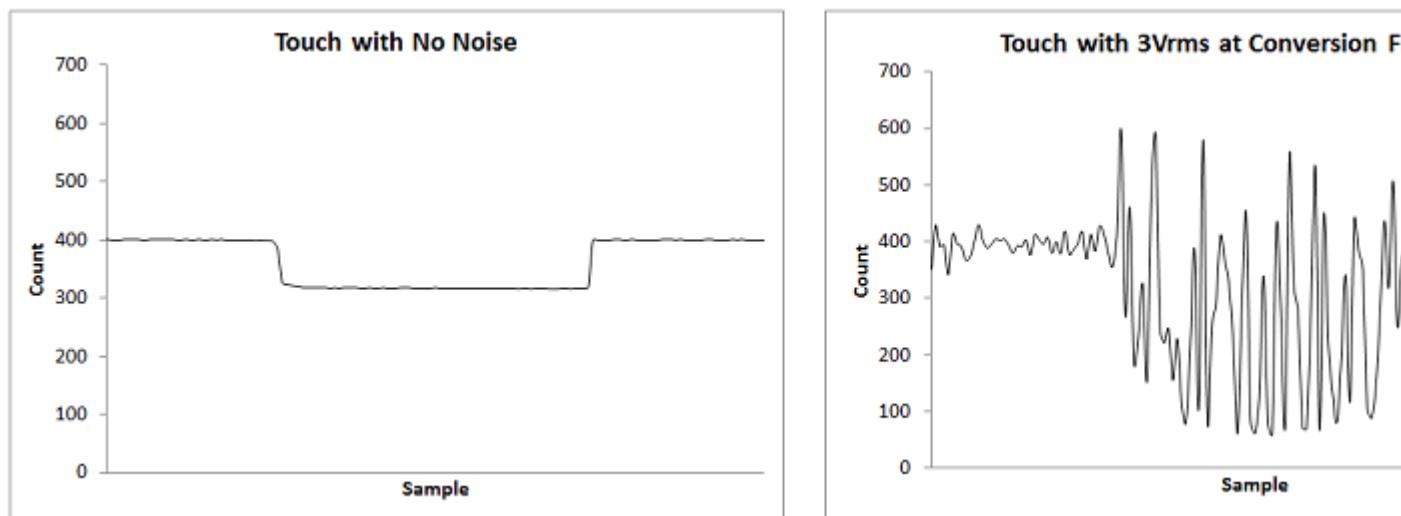


Figure 6.52: Self Capacitance Button with No Noise and 3Vrms Conducted Noise at Conversion Frequency

- If the noise is nearby the conversion frequency, but not directly aligned with it or a low harmonic, the measurement will not be corrupted but it will experience an increase in sensitivity to touch with a time-varying noise component.

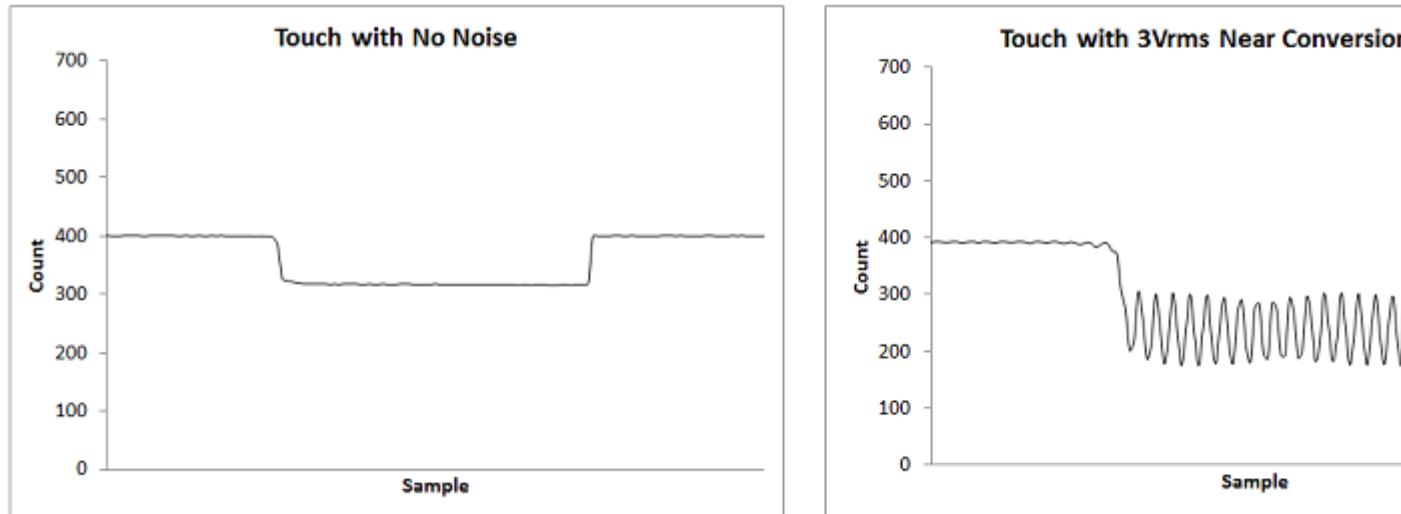


Figure 6.53: Self Capacitance Button with No Noise and 3Vrms Conducted Noise at Conversion Frequency

Note that the measurement is stable, and the noise only occurs in one direction. Self capacitance sensors will exhibit susceptibility in this way beginning just below their charge transfer frequency and going out past it with diminishing noise farther out in frequency.

- Amplitude of the noise. As the noise amplitude increases, the average delta due to a touch increases. This is a result of injected current into the Rx. This effect shows up as an increase in sensitivity to touch. For example, a button may trigger with a few mm of proximity rather than a full touch.

#### Mutual Capacitance

Mutual capacitance sensors exhibit corruption of data if noise is present at the conversion frequency or its lower harmonics. The susceptibility band is typically 100 kHz around the conversion frequency. The effect can be seen in the diagram below.

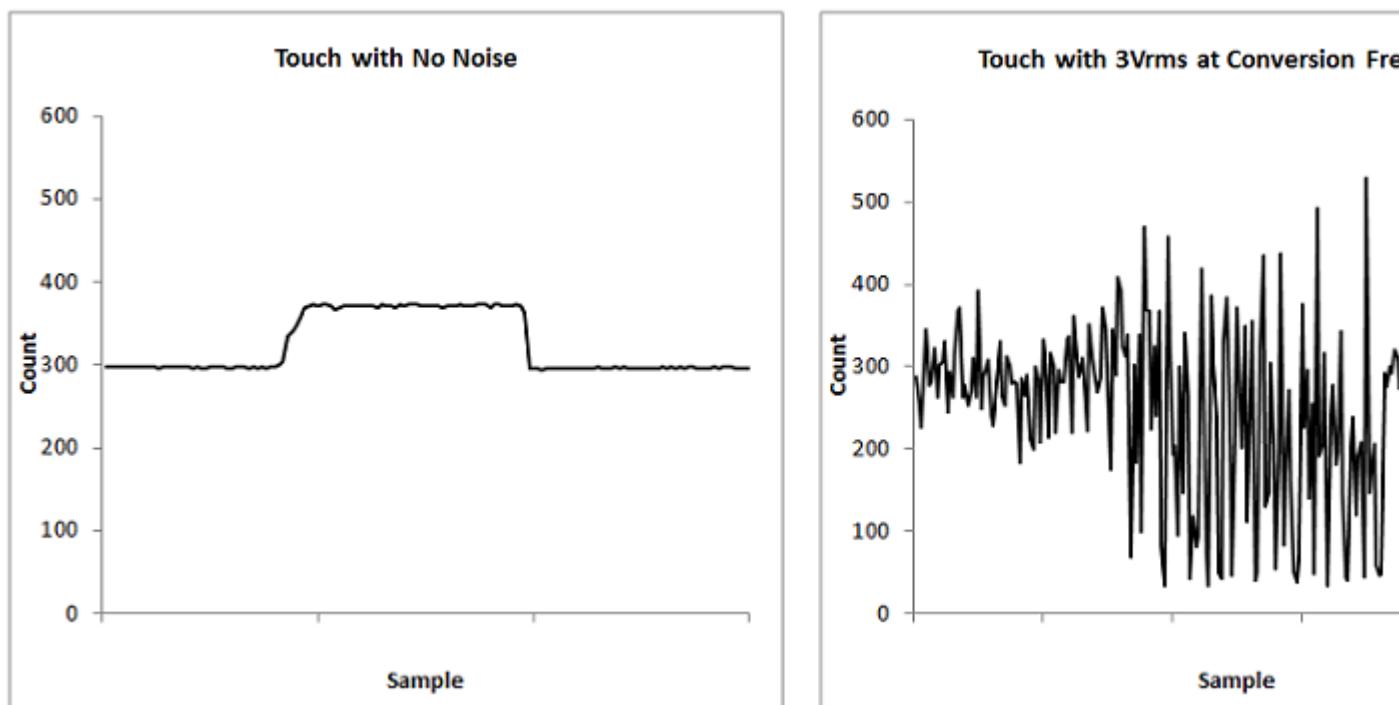


Figure 6.54: Mutual Capacitance Button with No Noise and 3Vrms Conducted Noise

Notice how the data captured is completely unusable at this point. If the recommended layout practices are not followed, the result will be a widening of the susceptibility band in frequency beyond the typical value of  $\sim 100$  kHz. This will limit the performance of the frequency hopping solution described next.

#### 6.10.3.4.4 CapTivate™ MCU Solution

Several methods are employed to improve immunity to conducted and radiated RF interference:

- Liberal use of ground structures in the PCB layout to reduce the fringe free-space coupling of the electrode and its trace back to earth ground and the user.
- Frequency hopping is applied to prevent noise in a given frequency band from corrupting the measurement by aggregating the data from 4 different conversion frequencies with the multi frequency processing (MFP) algorithm
- In mutual mode, filter capacitors are placed between Rx lines and circuit common to narrow the susceptibility band in frequency, improving the benefit of frequency hopping
- In self mode, spread spectrum clock modulation and the dynamic threshold adjustment (DTA) algorithm are applied to compensate for the increased sensitivity to touch
- [IIR filtering](#) and [de-bounce](#) may be used to add additional robustness

#### Self Capacitance

To achieve self-capacitance immunity, it is imperative that dense grounding structures be applied to limit fringing E-field lines. This has the effect of stabilizing the electrode. Then the use of frequency hopping and spread-spectrum clocking improves the data to a level where it may be processed by an algorithm, as shown below. In the plot below, noise is being injected at F3. Notice how F3 looks the same as the plot shown above in the problem section- it has positive and negative noise. However, F0-F2 are clean when no touch is present, and only exhibit the additional sensitivity in the negative direction when the button is touched. Applying spread-spectrum clock modulation reduces the effect of the noise at F3 to the point where it looks similar to F0-F2.

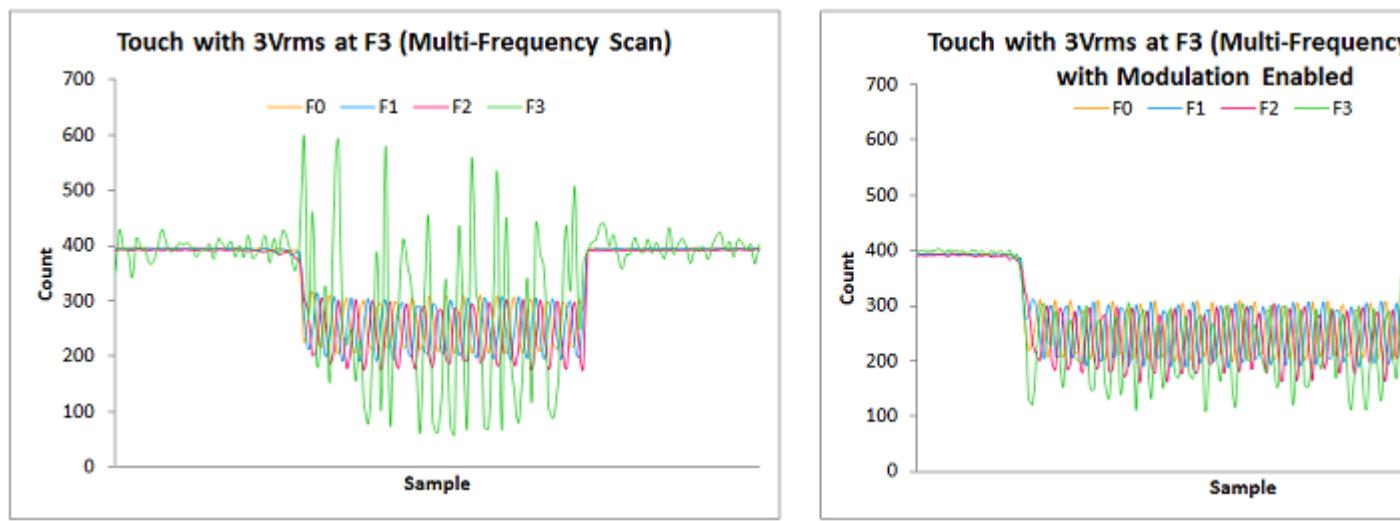


Figure 6.55: Self Capacitance Button with 3Vrms Conducted Noise, 4 Frequency Raw Data Without and With Modulation

Taking that data set and applying the multi-frequency processing algorithm results in a fairly clean signal that may be filtered via the addition of the IIR count filter, as shown below. Notice that the signal still has an increased delta due to the injected noise current.

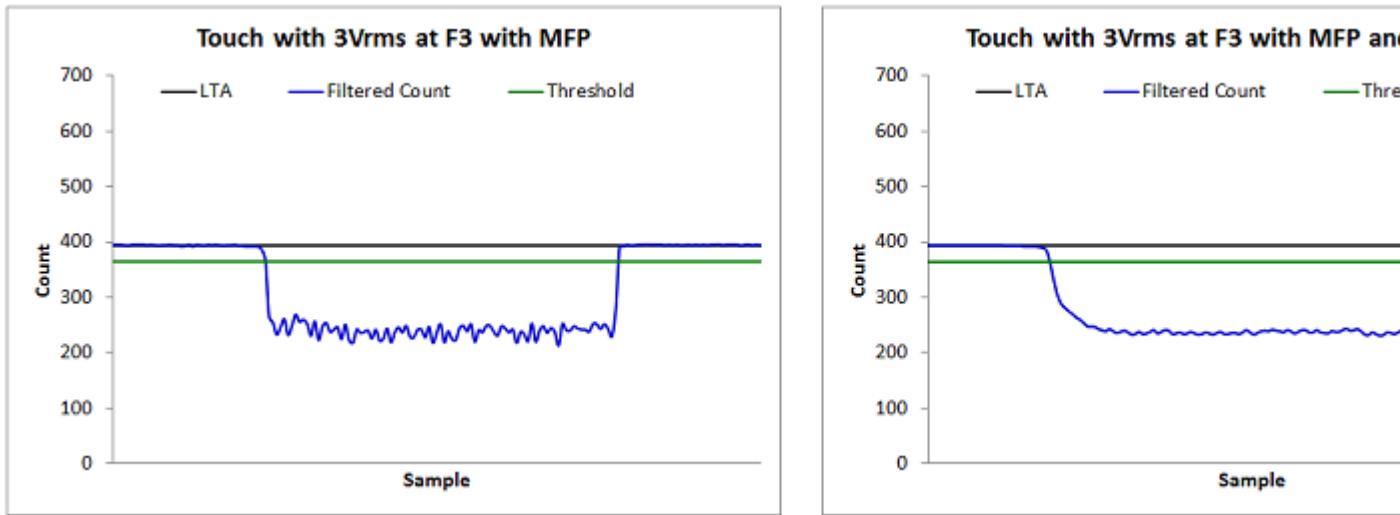


Figure 6.56: Self Capacitance Button with 3Vrms Conducted Noise, MFP Data Without and With IIR Filtering

Now that a clean signal has been obtained, the noise level that was measured in the original raw data is used to compute a threshold adjustment factor via the dynamic threshold adjustment algorithm (DTA). This provides compensation for the slight gain in sensitivity due to noise in the system. The noise level is also reported, should it be of interest to the application. Below is a comparison of a no-noise touch and a touch in the presence of 3Vrms conducted noise at the base conversion frequency.

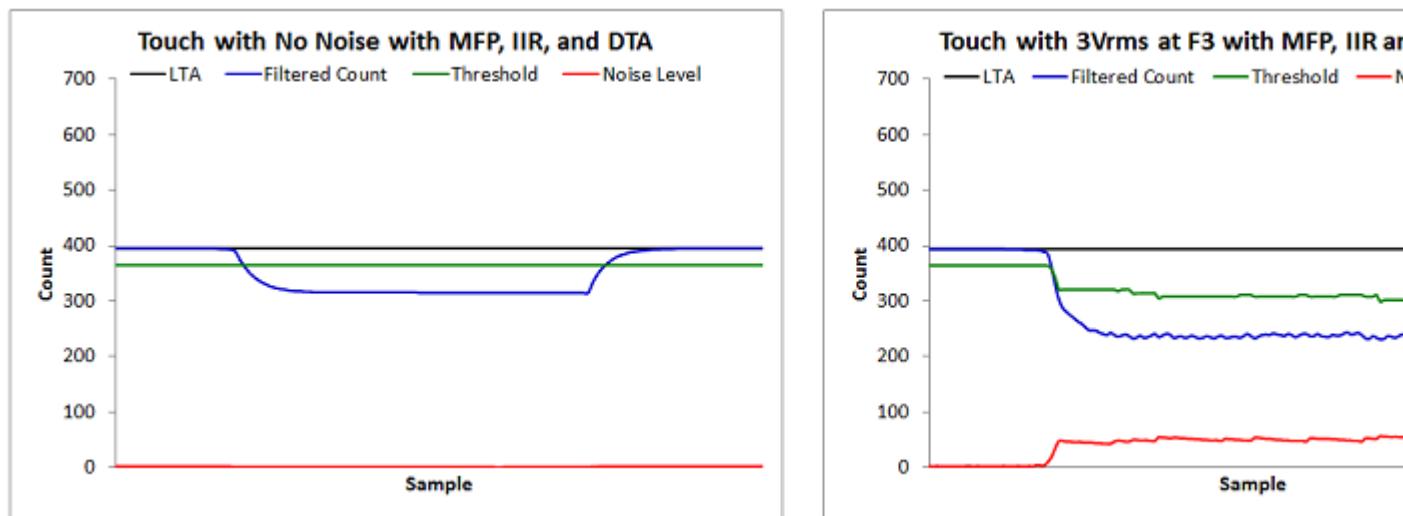


Figure 6.57: Self Capacitance Button with No Noise and 3Vrms Conducted Noise, with MFP, IIR, and DTA

The application of all these design principles results in a button that dynamically responds to changes in the environment, compensating for noise. These techniques also work quite well on slider and wheel sensors.

#### Mutual Capacitance

As shown below, the application of frequency hopping and filter capacitors in mutual mode produces usable measurements at the auxiliary frequencies (left plot). The multi-frequency processing (MFP) algorithm may then resolve the data set into a usable count value that is fed into the standard element and sensor processing algorithms.

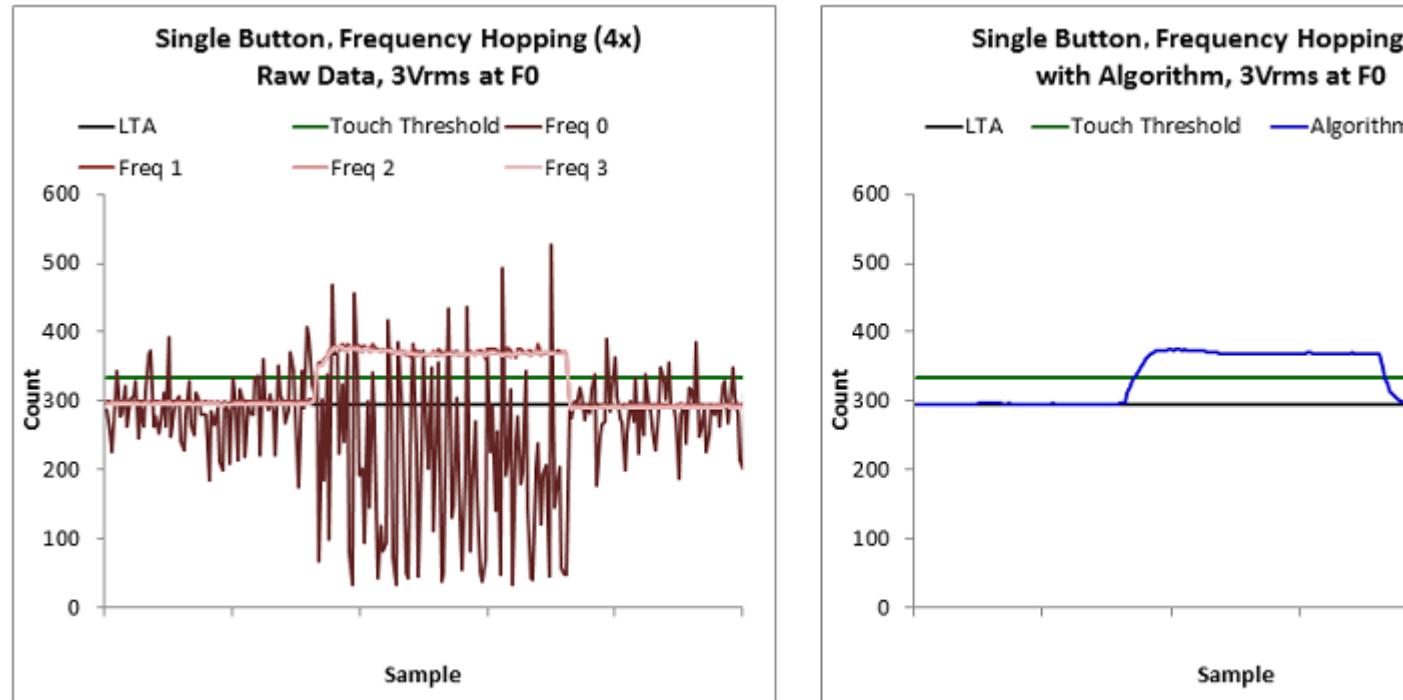


Figure 6.58: Mutual Capacitance Button with 3Vrms Conducted Noise, Raw Data and Processed Data

If the solution is looked at across a frequency sweep, the benefit of the frequency hopping approach becomes evident. This data set corresponds to a button that was not touched during the test.

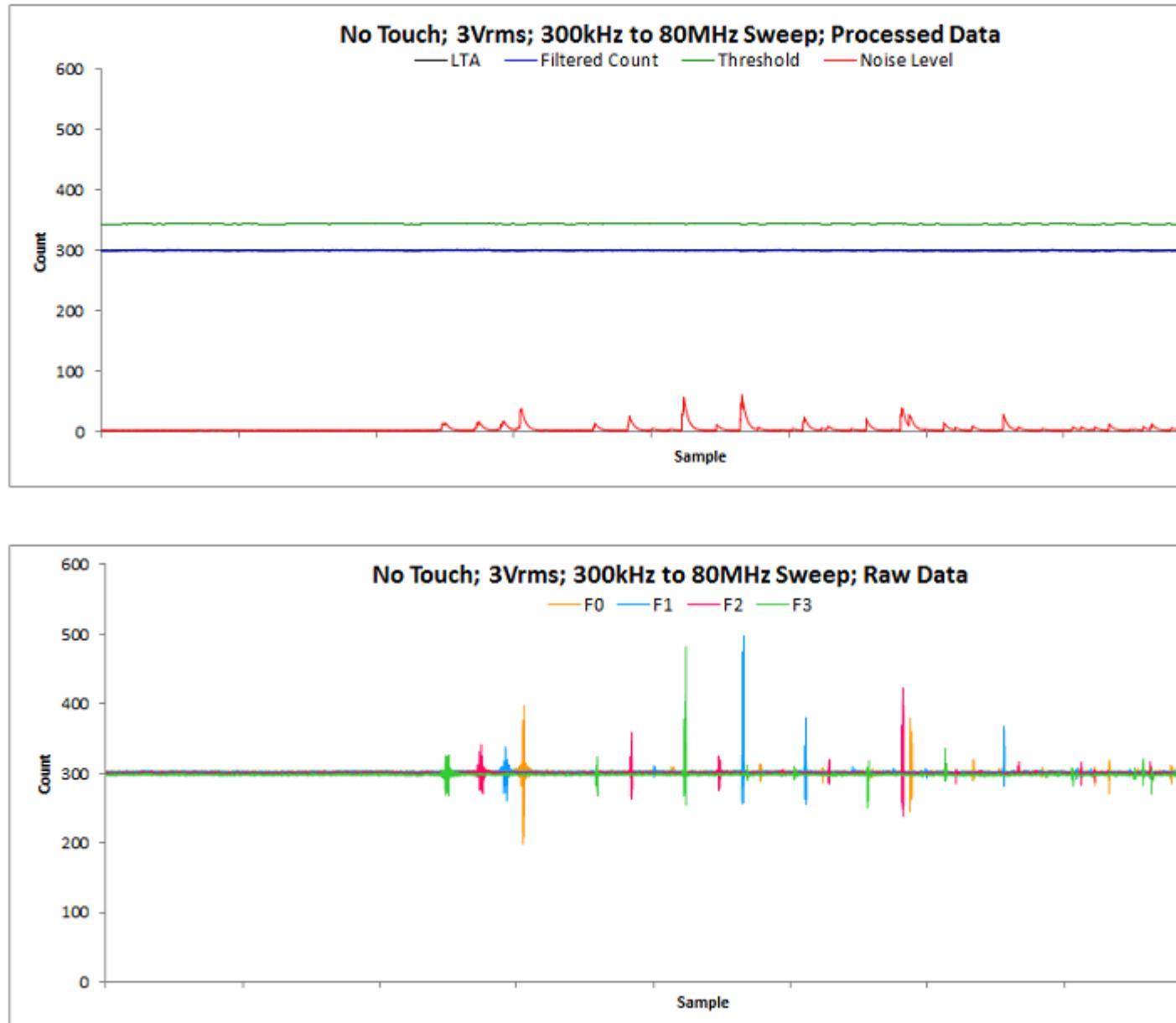


Figure 6.59: Mutual Capacitance Button with 3Vrms Conducted Noise, No-Touch Noise Sweep

Note how the noise bands are very narrow, and are easily overcome by the frequency hopping algorithm. When touched, the susceptibility band increases and the amplitude of the noise increases. However, usable data is still obtained.

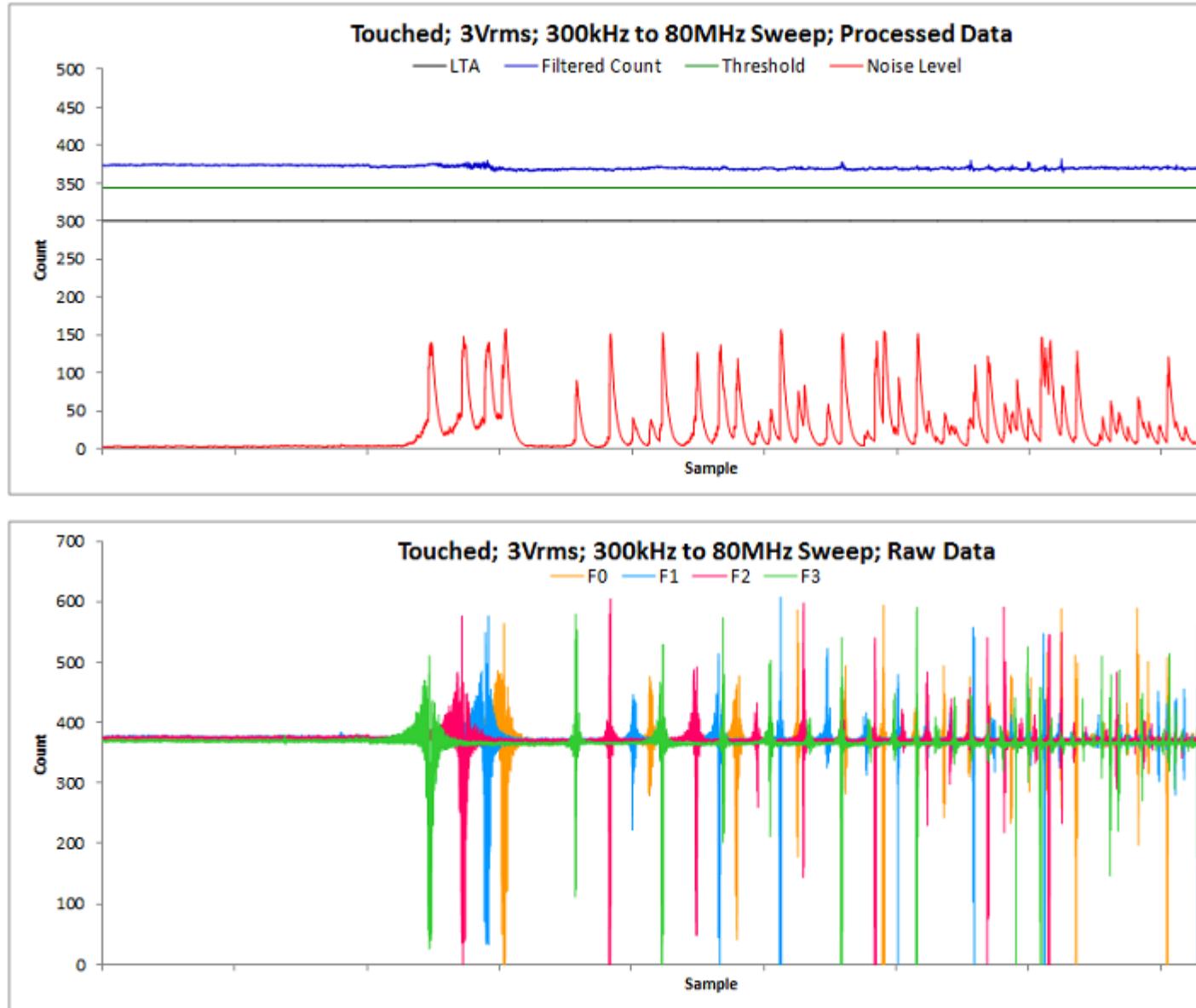


Figure 6.60: Mutual Capacitance Button with 3Vrms Conducted Noise, Touched Noise Sweep

In both the untouched and touched diagrams, it is easy to pick out the fundamental frequencies at the left, and their corresponding harmonics. It may become obvious from reviewing the above diagrams that noise may affect more than one frequency at a time. In the event that this occurs, the frequency hopping approach is still able to handle most situations. The diagram below shows noise affecting two frequencies at the same time.

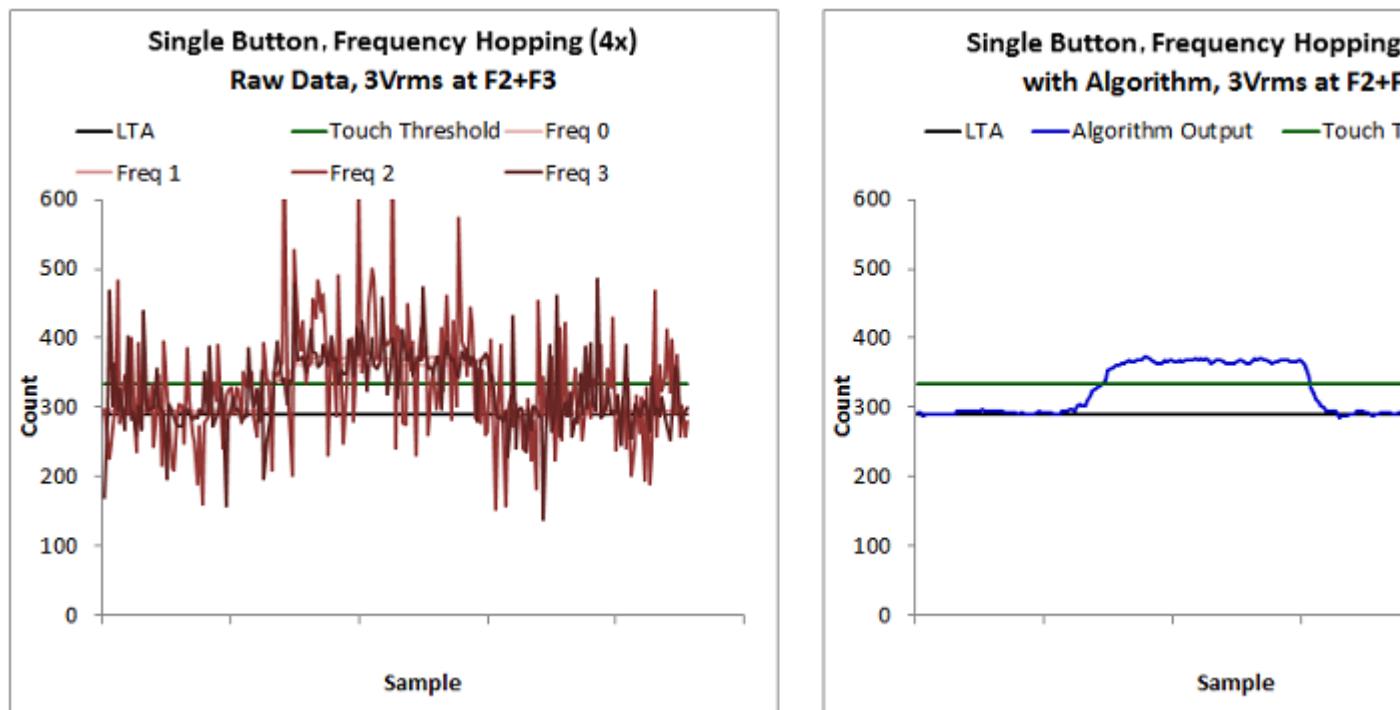


Figure 6.61: Mutual Capacitance Button 3Vrms Conducted Noise

In the event that the noise become too great for the algorithm to determine what is happening, it is possible to use the noise threshold and noise status bits of the elements to alert the application and lock out a safety critical function.

#### 6.10.3.4.5 Design Actions Required

Designing for immunity to RF interference does require hardware and software considerations. Be sure to review the final section of this guide, [Noise Mitigation Steps](#), for implementation details.

##### 1. CapTivate™ Peripheral EMC Features

- Integrator-based charge transfer method
- Offset subtraction signal chain block
- Frequency hopping
- Spread-spectrum clocking

##### 2. Hardware Design Principles

- Implement conducted noise immunity layout best practices, taking into account the thickness and dielectric of the overlay material
- Eliminate or limit use of off-board connectors that carry capacitive sensing lines

##### 3. Signal Processing Tools

- CapTivate™ EMC Multi Frequency Processing (MFP) algorithm
- CapTivate™ EMC Dynamic Threshold Adjustment (DTA) algorithm
- IIR Count Filtering
- De-bounce

---

## 6.10.4 Noise Mitigation Steps

One of the nice things about designing for EMC is that many of the techniques used to improve a design for one type of noise also tend to improve the design for other types of noise as well. This section steps through noise mitigation steps that should be applied to all designs that require noise immunity. It is best to think of this section as a design check list. All of the items discussed here should be applied when designing a product to have robust noise immunity.

1. [Determine](#) whether the design will be using self or mutual capacitance (or both).
2. Before beginning a PCB layout, review the [hardware checklist](#).
3. When configuring a project in the CapTlve™ Design Center, review the [tuning checklist](#).

### 6.10.4.1 Self versus Mutual Capacitance for Noise Immunity

Self capacitance is the recommended measurement method for designs that require sliders and wheels, as well as designs that need more than the commercial product level of 3Vrms conducted noise immunity. Mutual capacitance is recommended for designs that have higher key densities (16-32 buttons) and only require the commercial product immunity levels.

### 6.10.4.2 Noise Immunity Hardware Check List

Review this check list before starting a PCB layout.

1. Follow [electrode design best practices](#) for noise immunity.
2. Utilize dense [return \(ground\) plane structures](#) to limit fringe E-field lines.
3. For mutual (projected) capacitance sensors, populate a 68pF ceramic capacitor between each Rx and circuit return (ground). This capacitor should be placed close to the MCU. It is also a good idea to allow a provision to populate a capacitor between each Tx and circuit return, should it be needed during qualification. The TI noise immunity reference designs do not require the Tx capacitor.
4. Avoid routing capacitive sensing lines (Rx's or Tx's) through board-to-board connectors or cables. Connectors can be significant noise receptors.
5. Optimize the overlay thickness as much as possible. The thicker the overlay material, the higher the chance of a touch on one key affecting a nearby key. With thinner overlays, electrodes will be more susceptible to ESD. The sweet spot is 1.5mm to 4mm for a typical PC/ABS/acrylic overlay.
6. Use good power supply design principles to limit noise at the entry point. If possible, implement a common mode choke coil with a high impedance to common-mode noise in the range of 100 kHz to 100 MHz.

#### 6.10.4.2.1 Noise Immunity Electrode Design Best Practices

When designing for noise immunity, there are several best practices to apply to electrode geometry. Self capacitance and mutual capacitance will be discussed separately.

##### Self Capacitance

In the case of self capacitance, 3Vrms immunity is usually attainable without significant overhaul to the electrode design principles discussed in the [button design section](#).

The following principles should be applied:

1. Limit the size of the electrodes where possible. Do not oversize electrodes beyond what is needed to achieve the desired sensitivity with the overlay that will be used with the product.
2. Avoid air gaps in the overlay stackup

- 
3. Keep traces from the MCU to the electrode as short as possible
  4. If possible, do not place electrodes on the edges of the PCB, as this limits the effectiveness of [ground shielding](#)
  5. If the design has a thin overlay stackup (<3mm), a special electrode design may be used which has a ground plane in the center of the button. Placing a ground plane in the center of a button has two effects: additional stability for the electrode, and a path for noise currents into the ground plane and not into the electrode during a touch

Below is a comparison of a standard button layout with the special button layout discussed in principle #5. The special button layout is really only suitable for designs with overlay stackups less than 3mm. Above 3mm, the sensitivity trade-off becomes too great.

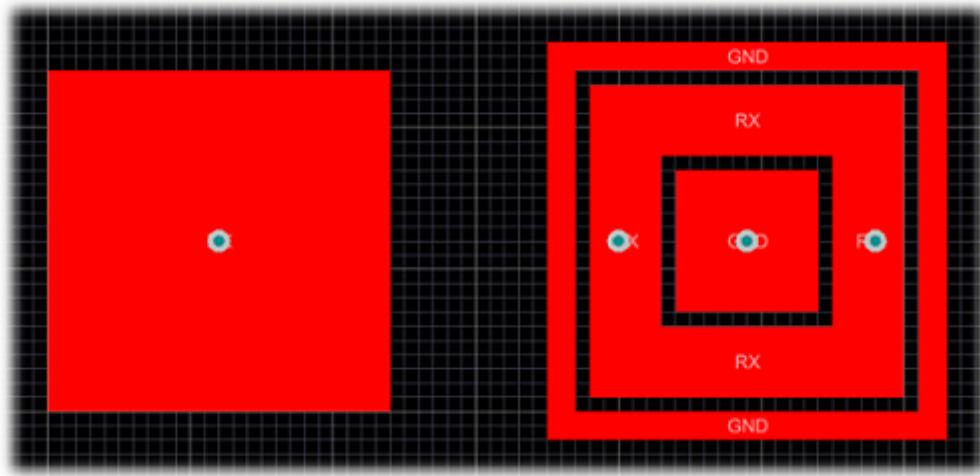


Figure 6.62: Self Buttons (Grid = 0.5mm)

#### Mutual Capacitance

Mutual capacitance requires a specific geometry to achieve immunity. The layout below depicts the recommended geometry. The goals are as follows:

1. Limit the size of the Rx
2. Overlay stackup air gaps are not permitted for mutual capacitance
3. Use cutouts in the Rx as shown to improve sensitivity
4. Place a small ground fill in the center of the electrode to add a path for noise currents into the ground plane and not into the electrode during a touch
5. Route all Rx signals on the bottom layer if possible to limit their exposure to a touch
6. Keep the number of Tx lines per Rx as low as possible. This limits the vulnerability of the receiver to noise.

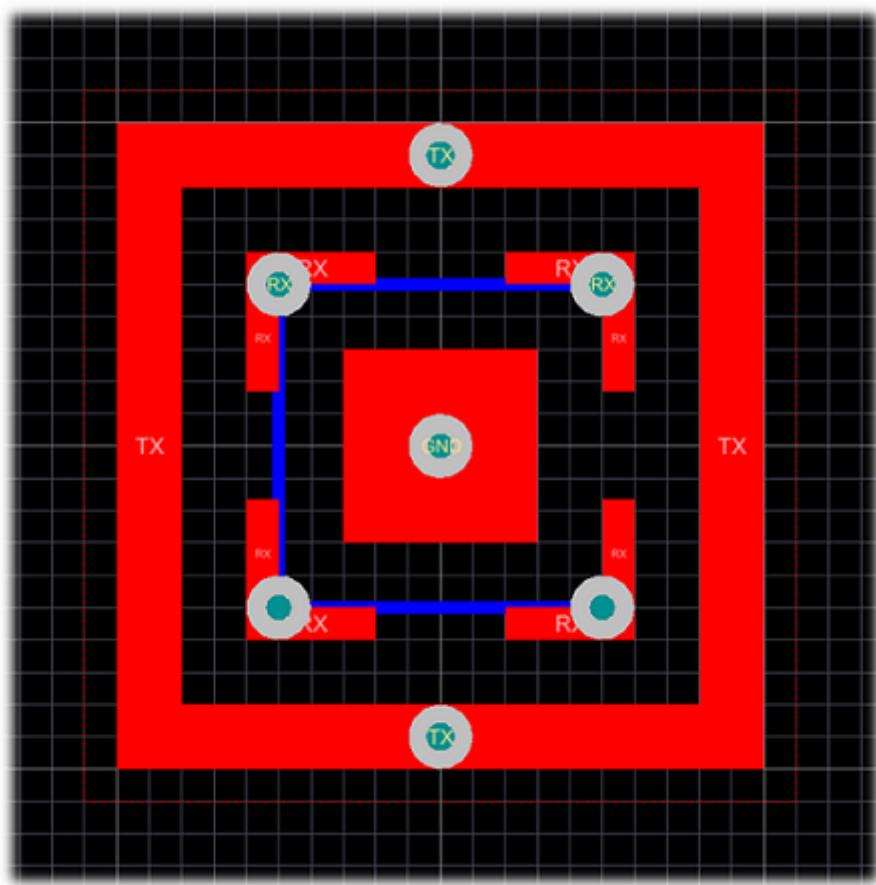


Figure 6.63: Mutual Buttons (Grid = 0.5mm)

The geometry is as follows:

Parameter	Value
Overall Geometry	10mm x 10mm
Plane Keepout	0.5mm from overall geometry on top layer, flooded on bottom layer
Tx Thickness	1mm
Rx Thickness	0.5mm
Inner Ground Fill	3mm x 3mm
Tx to Rx Spacing	1mm
Rx Wing Length	2mm

This electrode geometry has been proven with overlays between 1mm and 3mm of thickness.

#### 6.10.4.2.2 Noise Immunity Electrode Shielding Best Practices

Proper ground shielding of electrodes is required to achieve noise immunity. This applies to both self and mutual capacitance designs.

##### Self Capacitance Shielding

For self capacitance designs, utilize a solid ground or power plane on the top layer of the PCB, and a medium-density hatched power or ground plane on the bottom layer of the PCB. As an example, if the CAPTIVATE-BSWP panel (which was designed for low power battery operation) was modified to attain noise immunity, the following changes would be made:

1. Removal of the proximity sensor
2. Addition of a solid ground/power plane on the top layer
3. Expansion of the hatched ground/power plane on the bottom layer to the size of the PCB

Below are before and after images of the top and bottom layers, applying the principles described above:

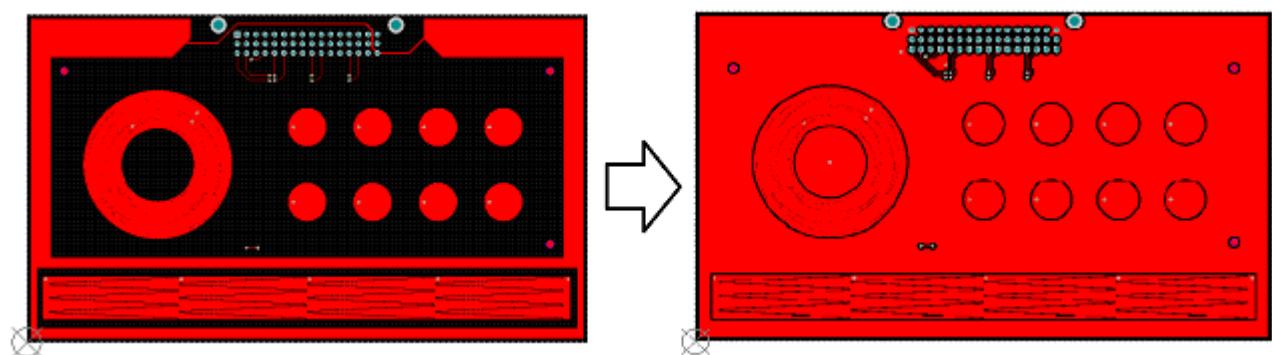


Figure 6.64: CAPTIVATE-BSWP Modified for Noise Immunity (Top Layer)

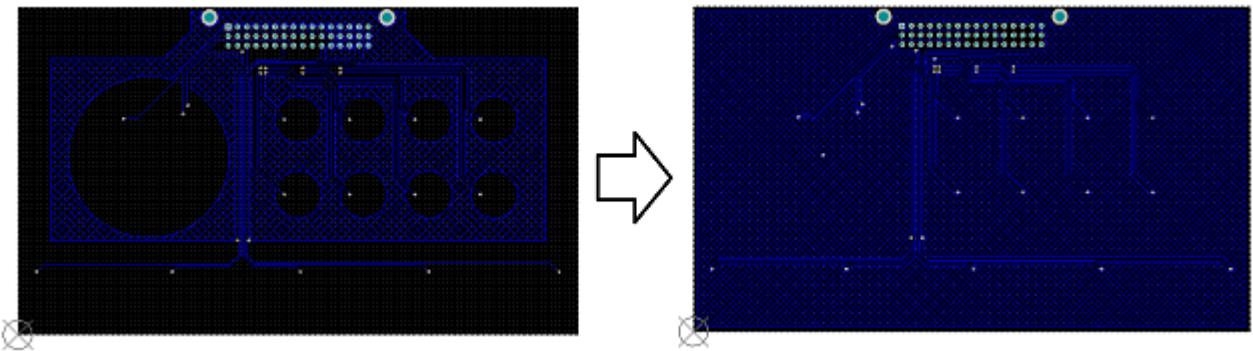


Figure 6.65: CAPTIVATE-BSWP Modified for Noise Immunity (Bottom Layer)

The addition of these dense ground structures serves to limit fringing E-field lines as much as possible, stabilizing the electrodes. Normally the addition of this much parasitic capacitance is considered bad practice, but in the case of noise immunity the trade-off is extremely beneficial. In addition, the gain and offset subtraction capability of the CapTivate™ technology makes it possible to maintain decent sensitivity even with the ground shield structures.

#### Mutual Capacitance Shielding

For mutual capacitance designs, utilize solid ground or power planes on the top and bottom layer of the PCB. Mutual receive electrodes (Rx's) are extremely sensitive (on the order of  $<1\text{mV}$ ). As a result, they are very vulnerable to noise currents. It is necessary to keep the Rx tracks as short as possible and as well-shielded as possible.

The design below illustrates these concepts for a 4x2 8-button mutual capacitance matrix:

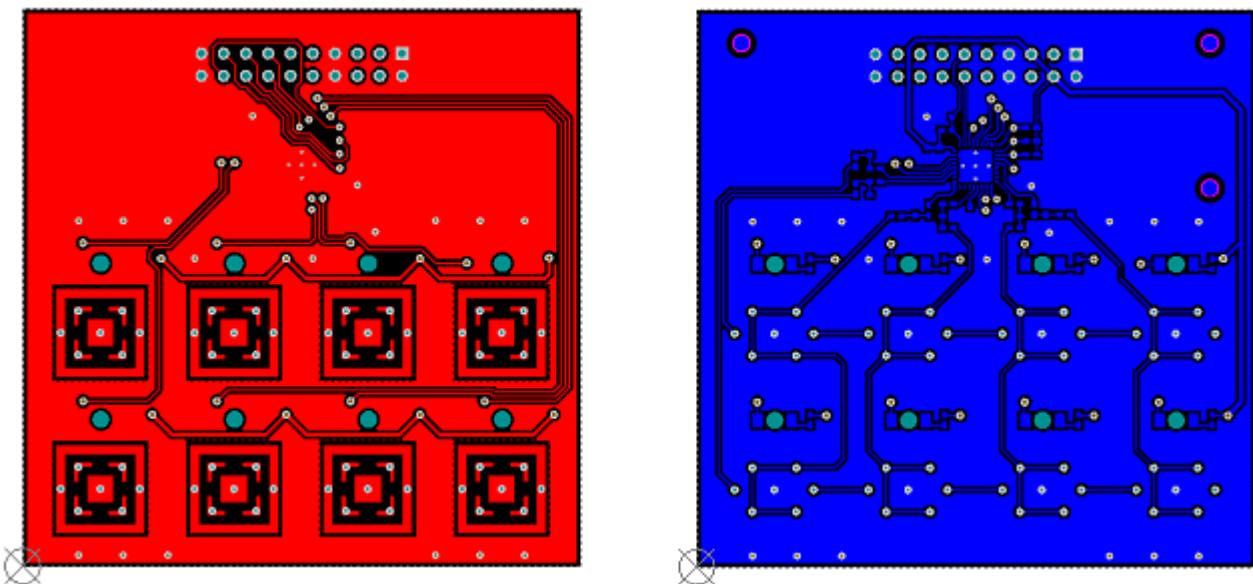


Figure 6.66: TIDM-CAPTIVATE-THERMOSTAT-UI (Top and Bottom Layer)

#### 6.10.4.3 Tuning Check List

Review this check list when tuning a design that requires noise immunity.

1. If RF immunity is required, enable [noise immunity mode](#) in the CapTivate™ Design Center. This will enable frequency hopping during measurement, and will increase the measurement time for each sensor by just over 4x.

- 
2. Set strong **de-bounce settings**. A value of 2 or greater is recommended.
  3. Set a strong **count filter beta**. A value of 2-3 for buttons and 1-2 for sliders/wheels is recommended.
  4. For self capacitance sensors, **enable modulation**. In self mode, the modulation improves the performance of the DTA (dynamic threshold adjustment) algorithm by spreading the conversion clock frequency and reducing noise spikes directly at the conversion frequencies and lower harmonics.
  5. For mutual (projected) capacitance sensors, **disable modulation**. In mutual mode, the modulation interferes with frequency hopping, as it widens the susceptibility bands.
  6. For mutual (projected) capacitance sensors, use a conversion frequency of 2 MHz or greater. If the **phase lengths** are both set to 1, a **frequency** divider of f/4 provides an effective conversion frequency of 2 MHz. Utilizing a faster conversion frequency increases the effectiveness of the frequency hopping mechanism by providing a wider spread between hop points.
  7. Set an error threshold for each sensor. For mutual capacitance, a good rule-of-thumb is to take the conversion count + 2-3x the maximum expected touch strength. For example, if a button has conversion count setting of 400, and a maximum expected delta due to touch of 100, than setting an error threshold of 600-700 would be appropriate. This serves to "cap" every conversion result to a maximum value. If noise in the system attempts to cause a conversion to continue beyond the error threshold, then the conversion will be halted. This prevents rogue samples from having extremely long sample times and effecting the response time / filtering of the system. For self capacitance, a good value is the conversion count \* 1.25. For example, a sensor with a conversion count of 800 would have an error threshold of 1000. This allows space for the runtime re-calibration algorithm to kick in (which happens if the LTA drifts outside +/- 1/8th of the conversion count, if enabled).
  8. For mutual (projected) capacitance sensors, set the **bias current** value to the highest setting.
  9. Become familiar with the **EMC module** configuration parameters, should you need to adjust them during development.

## 6.11 References

1. <http://wcalc.sourceforge.net/cgi-wcalc.html>

## Chapter 7

# Design Center GUI

### 7.1 Introduction

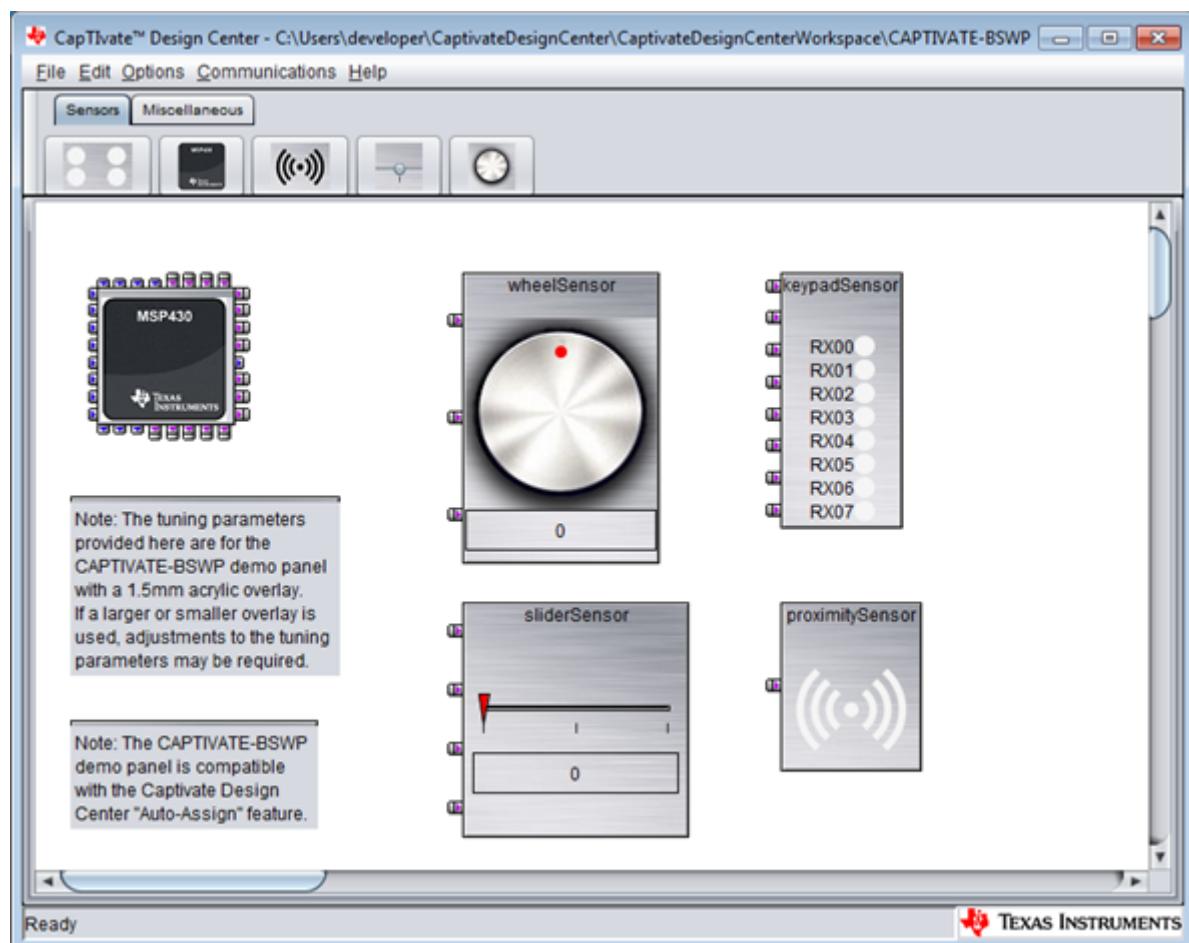


Figure 7.1: CapTlivate Design Center

The CapTlivate™ Design Center is a rapid development tool that accelerates capacitive touch designs for CapTlivate™ Technology enabled MSP430 devices. By helping guide the product developer through the capacitive touch development process, the CapTlivate™ Design Center can simplify and accelerate any touch design through the use of innovative user graphical interfaces, wizards and controls.

---

### 7.1.1 Key features

- Intuitive GUI tools for creating, configuring and defining the MSP430 connections for sensors
- Support for slider, wheel, button group, and proximity sensors
- Support mutual and self capacitive sensor types in the same design
- Automated generation of complete source code projects for CCS and IAR IDE's
- Real time target communication via a HID communication bridge. Enabling target communication allows users to:
  - View detailed sensor data
  - Configure and tune sensor performance
  - Perform SNR measurements

### 7.1.2 Hardware and software requirements

A PC with the following requirements is required to execute the CapTlivate™ Design Center:

- Windows 7 operating system with Java version 1.6+
- Apple OS X 10.10.1+ with Java version 1.7+
- Linux OS with Java version 1.6+

The source code projects generated by the tool support the following IDEs:

- `Texas Instruments Code Composer Studio (CCS)` version 6.1 or later
- `IAR Systems Embedded Workbench for MSP430` version 6.30 or later

#### 7.1.2.1 Additional Linux requirements

If you are not running the application using a root account, you may not have permissions to the HID device mounted by the MSP microcontroller.

To enable access perform the one-time setup step:

- Create the file `/etc/udev/rules.d/ti_hid.rules`.
- Add the lines below to the file and save the file.

```
1 # Allow open access to any usb device with the default Texas Instruments MSP
2 # Vendor Id of 0x2047
3 ATTRS{idVendor}=="2047", MODE=="0666"
```

### 7.1.3 Licensing

The CapTlivate™ Design Center is released under the BSD 3-clause license. See the license manifest in the installation directory for details.

## 7.2 CapTlivate™ Design Center Quick start

This section is intended to provide a very basic and quick overview of how to use the CapTlivate™ Design Center to define a typical capacitive touch design, generate source code, and communicate with the target. For additional detailed information about the features of the CapTlivate™ Design Center, refer to the [CapTlivate™ Design Center user's guide](#).

When ready to start your own design, check out the workshop guide section, [Creating a new sensor design project](#).

---

### 7.2.1 Start the Design Center

Double click on the desktop "CapTlivateDesignCenter" short cut to start the tool.



Figure 7.2: CapTlivate\texttrademark {} Design Center Shortcut

#### Create a new project

Use the File->Project New menu to create a new project

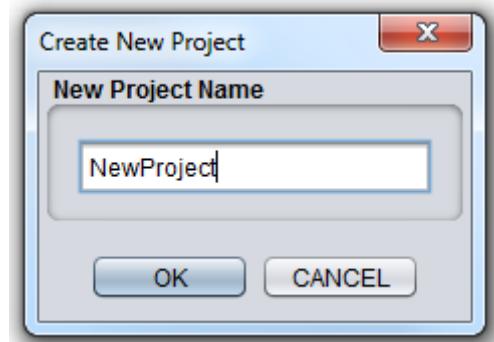


Figure 7.3: New Project Name

### 7.2.2 Place the sensors

Place a slider sensor in the design area

Select the slider by clicking on the sensor icon. Place the new object by clicking on the design canvas.

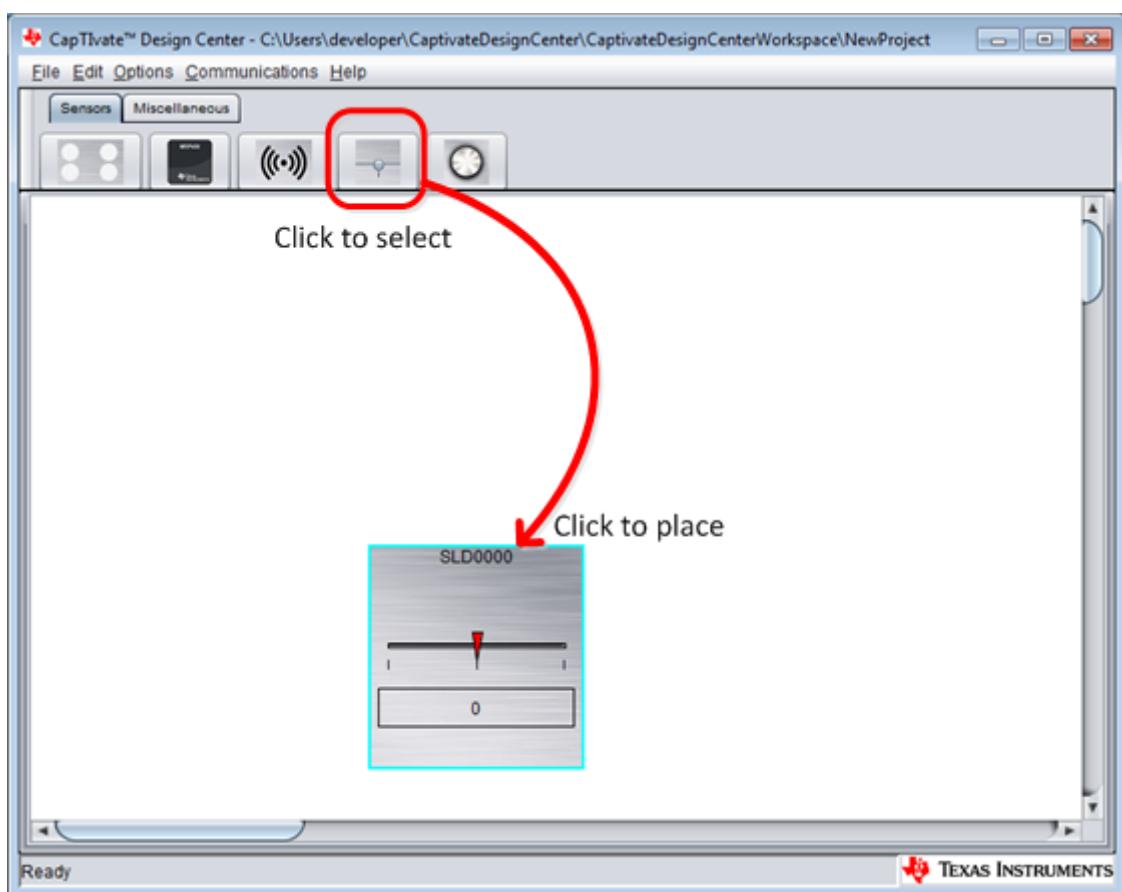


Figure 7.4: Placing a Slider Sensor

Display the sensor properties to configure it to be a self capacitance slider with 4 elements.

- Double-click on the slider object in the design area to display its properties.
- Configure the sensor to have 4 elements and close the properties dialog.

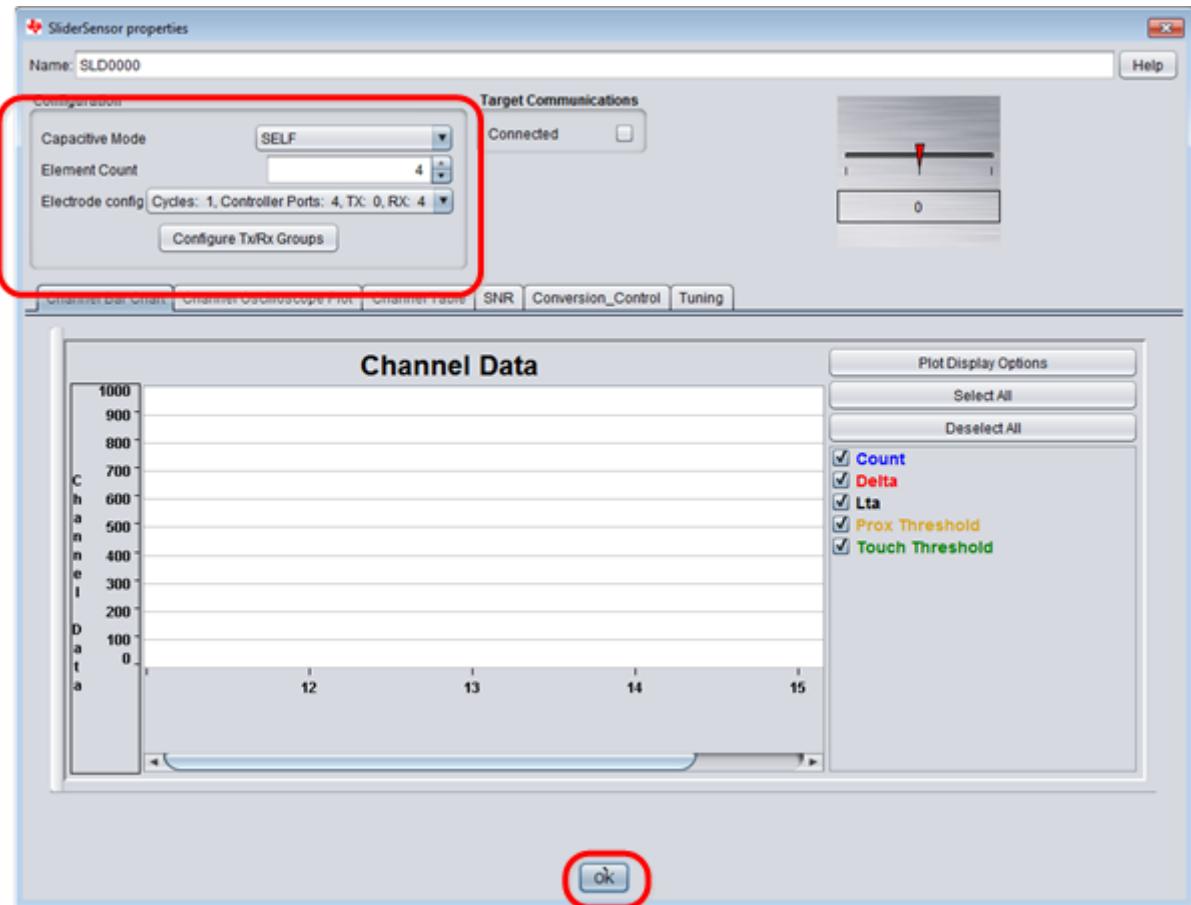


Figure 7.5: Modify Sensor Properties

Place a wheel sensor in the design area

Select the wheel by clicking on the sensor icon. Place the new object by clicking on the design canvas.

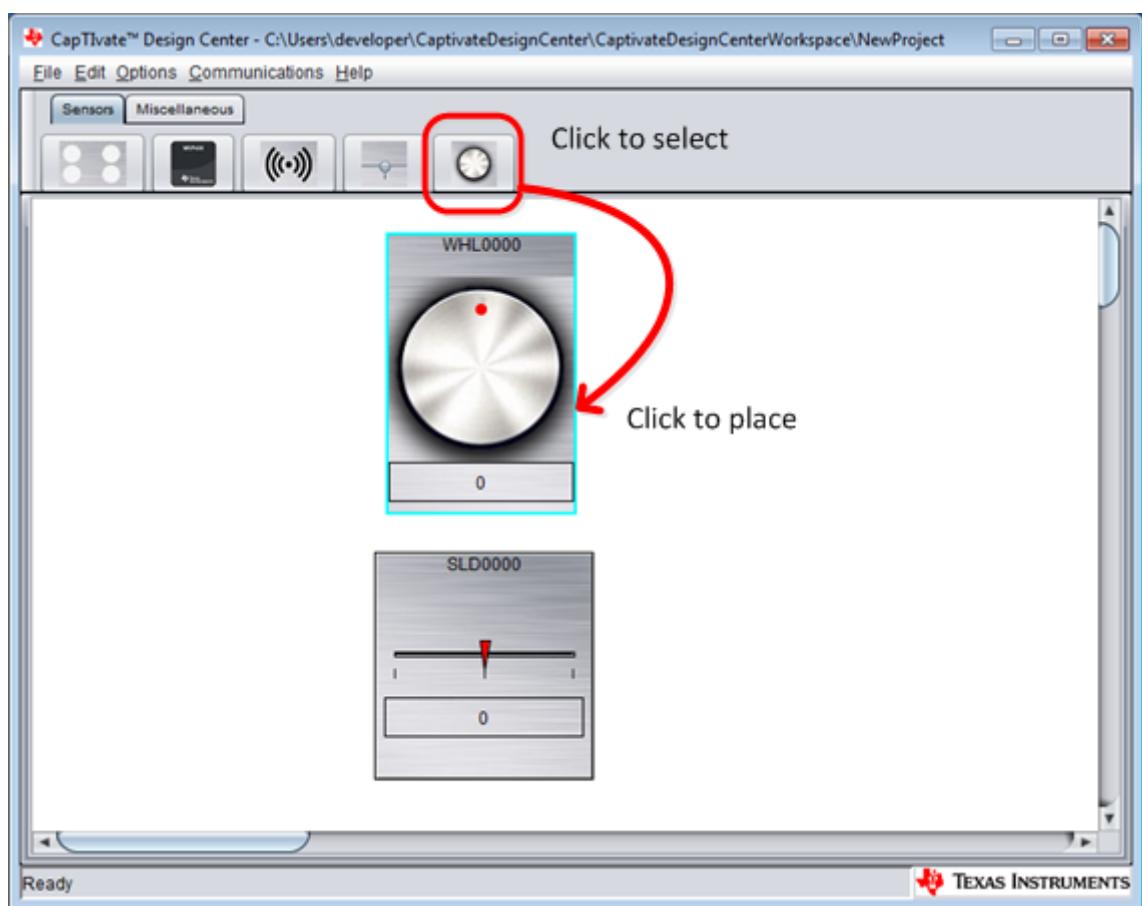


Figure 7.6: Placing a Wheel Sensor

Display the sensor properties to configure it to be a self capacitance wheel with 3 elements.

- Double-click on the wheel object in the design area to display its properties.
- Configure the sensor to have 3 elements and close the properties dialog.

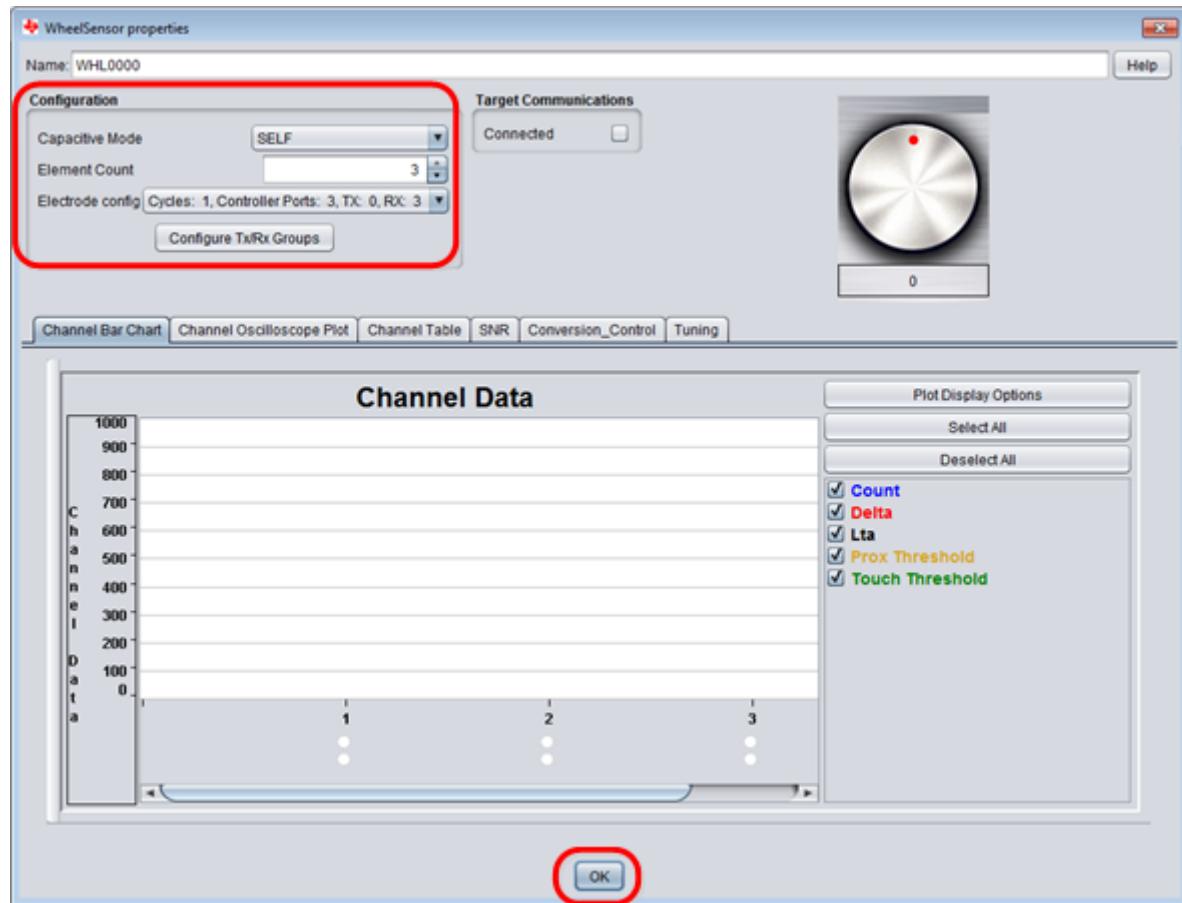


Figure 7.7: Modify Sensor Properties

Place a button group (keypad) sensor in the design area

Select the button group by clicking on the sensor icon. Place the new object by clicking on the design canvas.

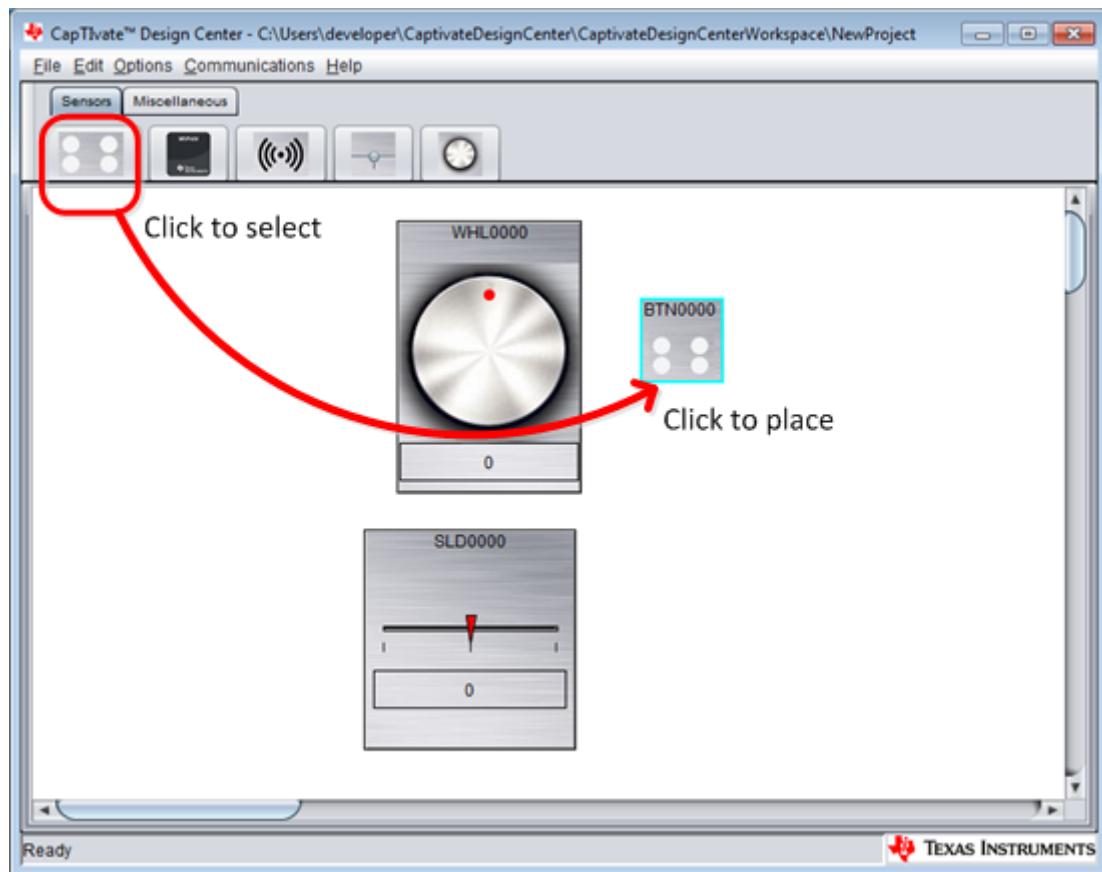


Figure 7.8: Placing Button Group Sensor

Display the sensor properties to configure it to be a self capacitance button group with 8 elements.

- Double-click on the button group object in the design area to display its properties.
- Configure the sensor to have 8 elements and close the properties dialog.

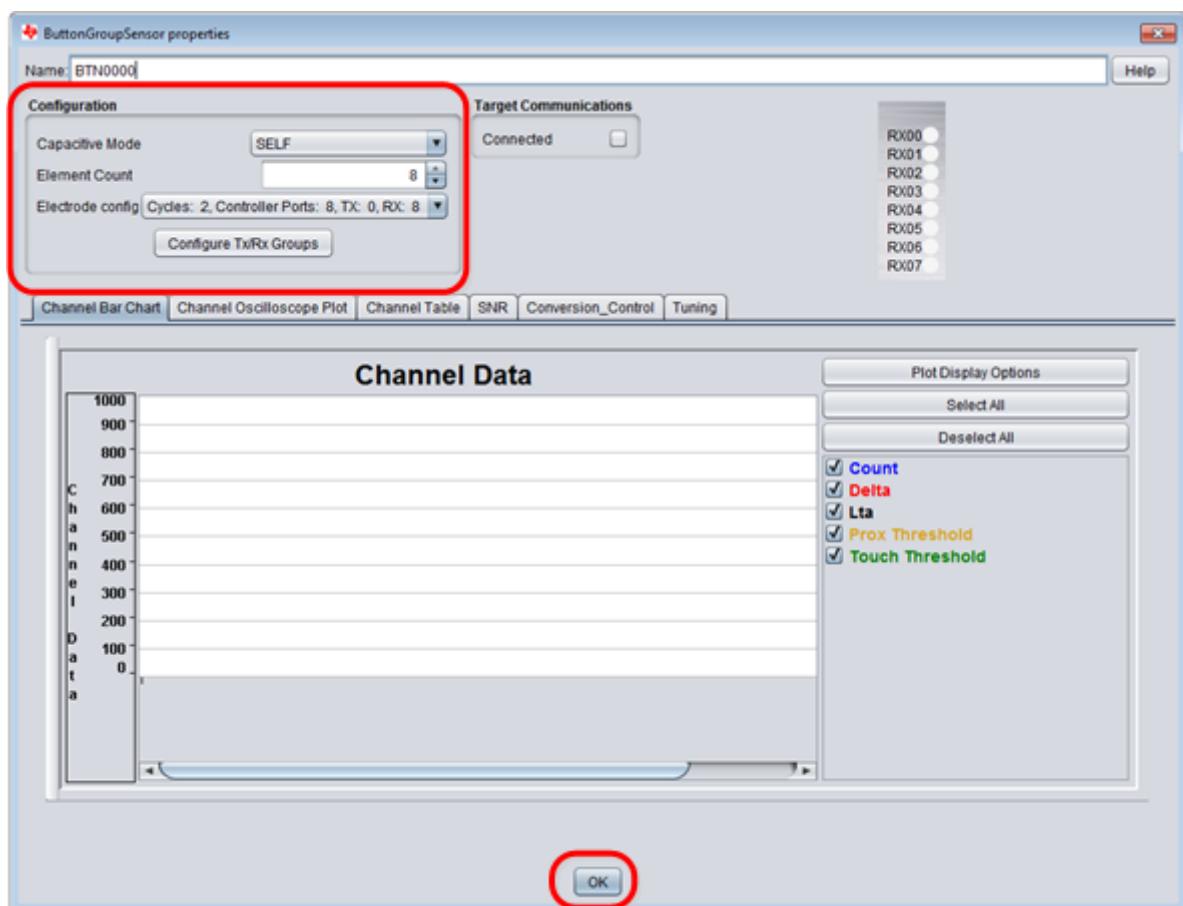


Figure 7.9: Modify Sensor Properties

Place a proximity sensor in the design area

Select the proximity sensor by clicking on the sensor icon. Place the new object by clicking on the design canvas.

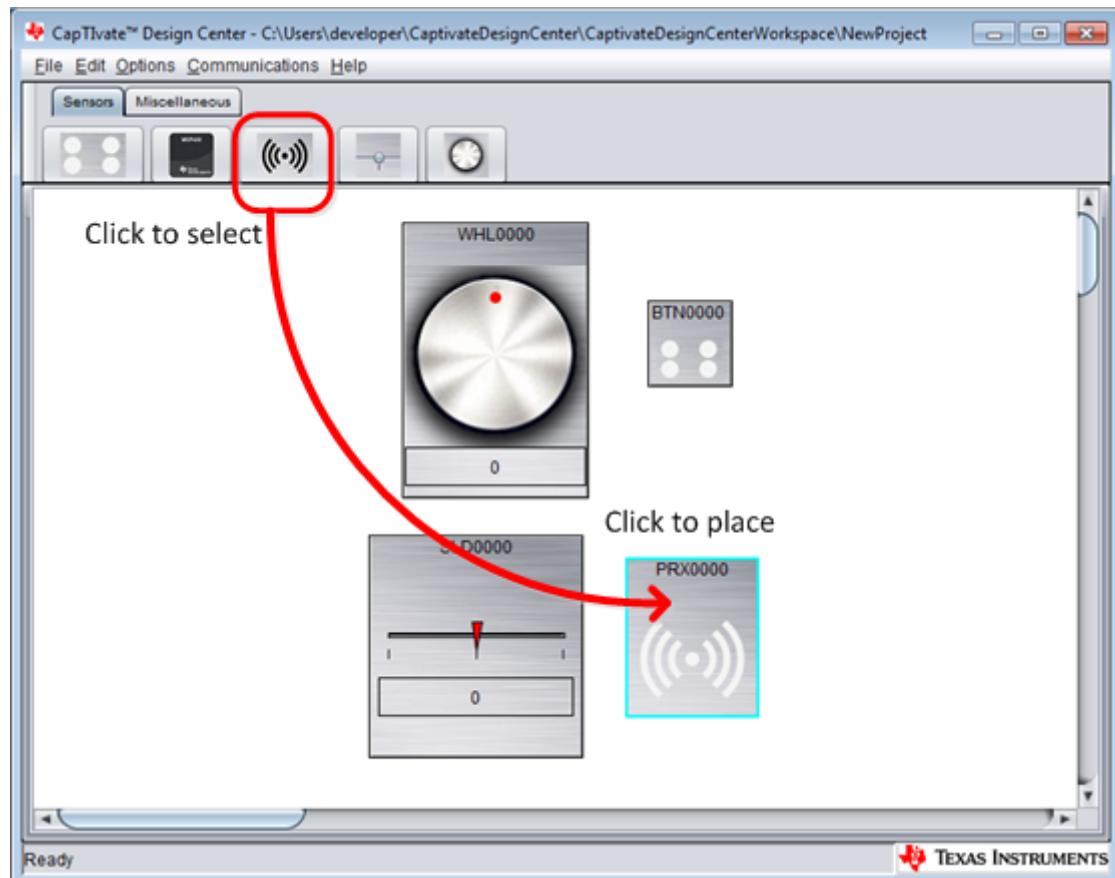


Figure 7.10: Place Proximity Sensor

Display the sensor properties to configure it to be a self capacitance proximity with 1 element.

- Double-click on the proximity object in the design area to display its properties.
- Configure the sensor to have 1 element and close the properties dialog.

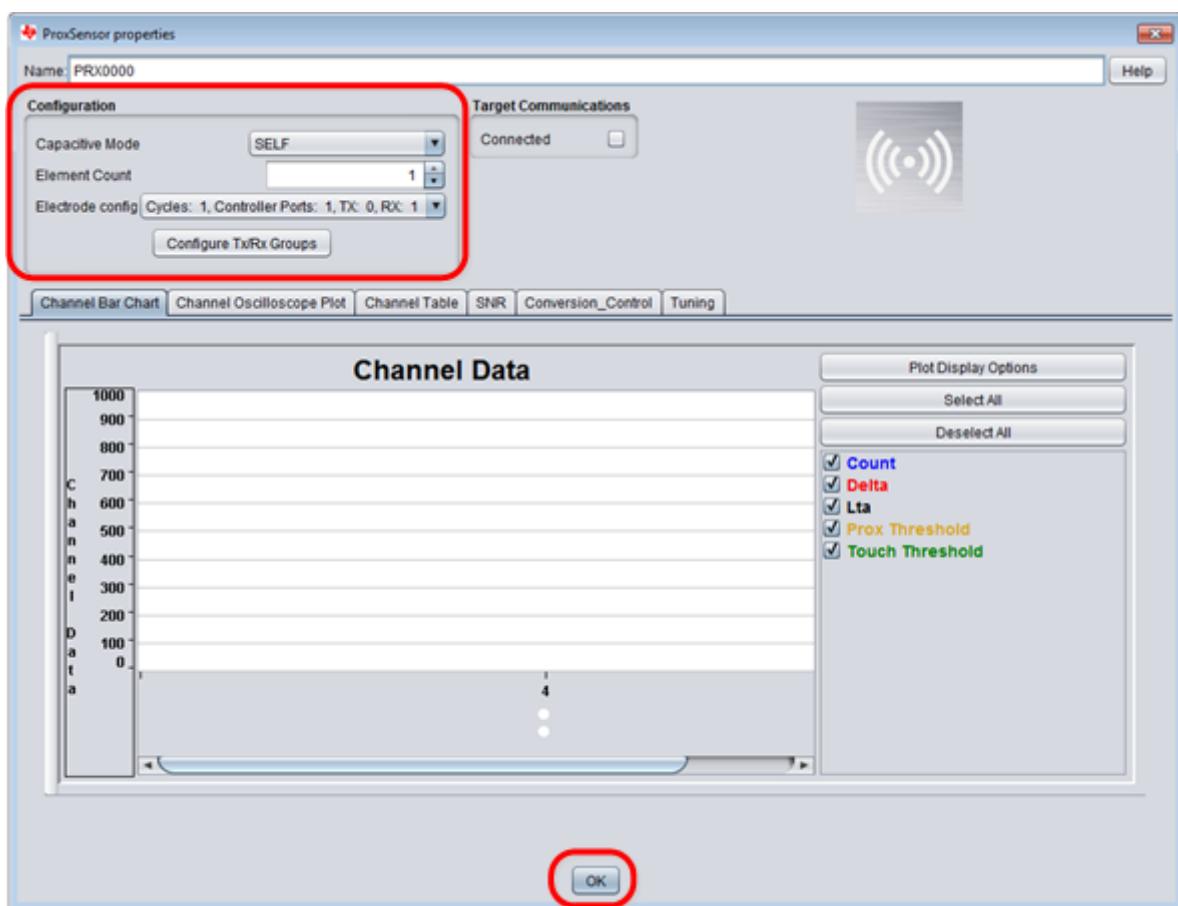


Figure 7.11: Modify Sensor Properties

### 7.2.3 Place the MSP430 controller

Select the MSP430 by clicking on the MCU icon. Place the new object by clicking on the design canvas.

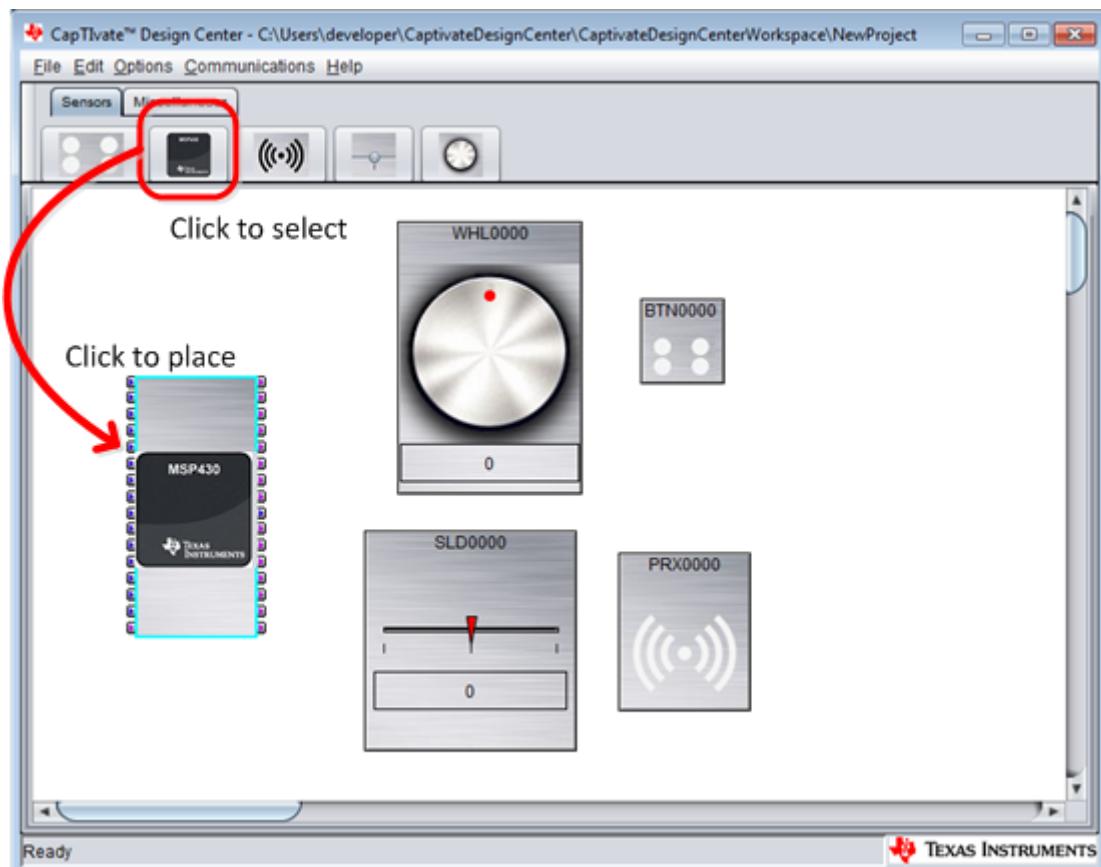


Figure 7.12: Place MCU

### 7.2.4 Connect sensors to MSP430 capacitive touch I/O ports

Double-click on the MSP430 controller object in the design area to display its properties.

- Configure the MSP430 controller as MSP430FR2633IRHB (32-pin QFN package)
- Note that the "Errors" LED is red, indicating that there are still unconnected sensor ports.

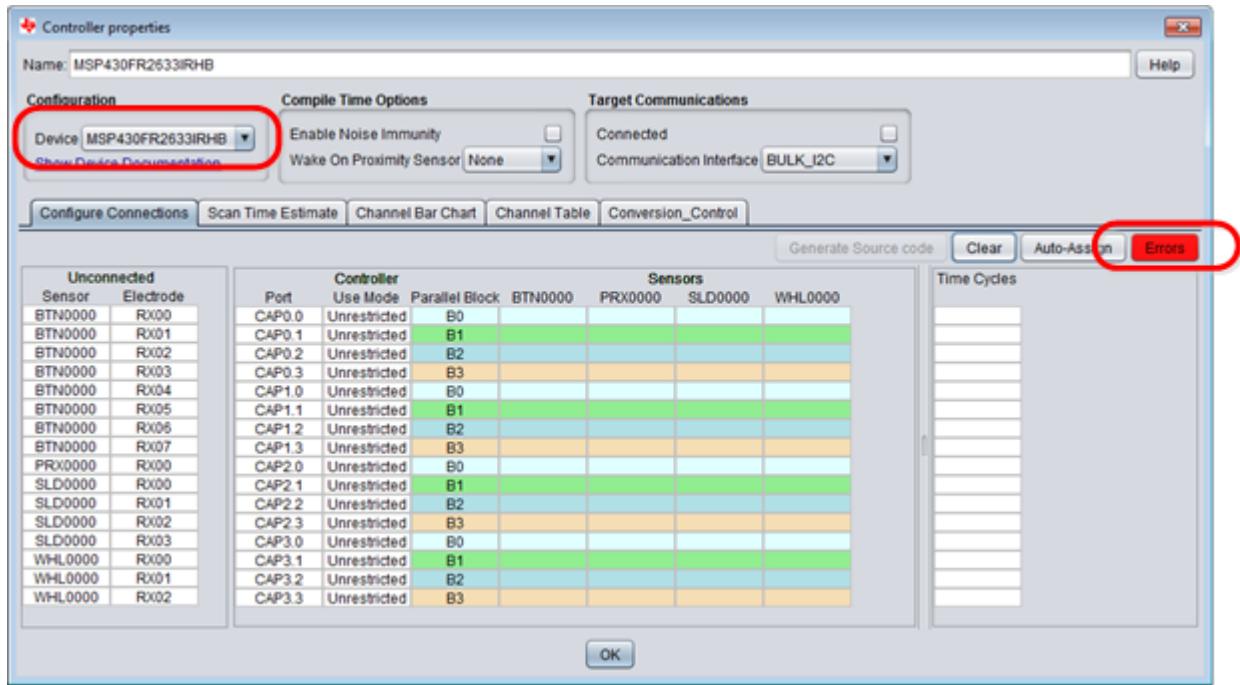


Figure 7.13: Display MCU Properties

Select the "Auto-Assign" button to automatically assign all the sensor ports to appropriate ports on the MSP430. Note that the "Errors" LED turns green and "OK", indicating that all sensor ports have been assigned to controller ports.

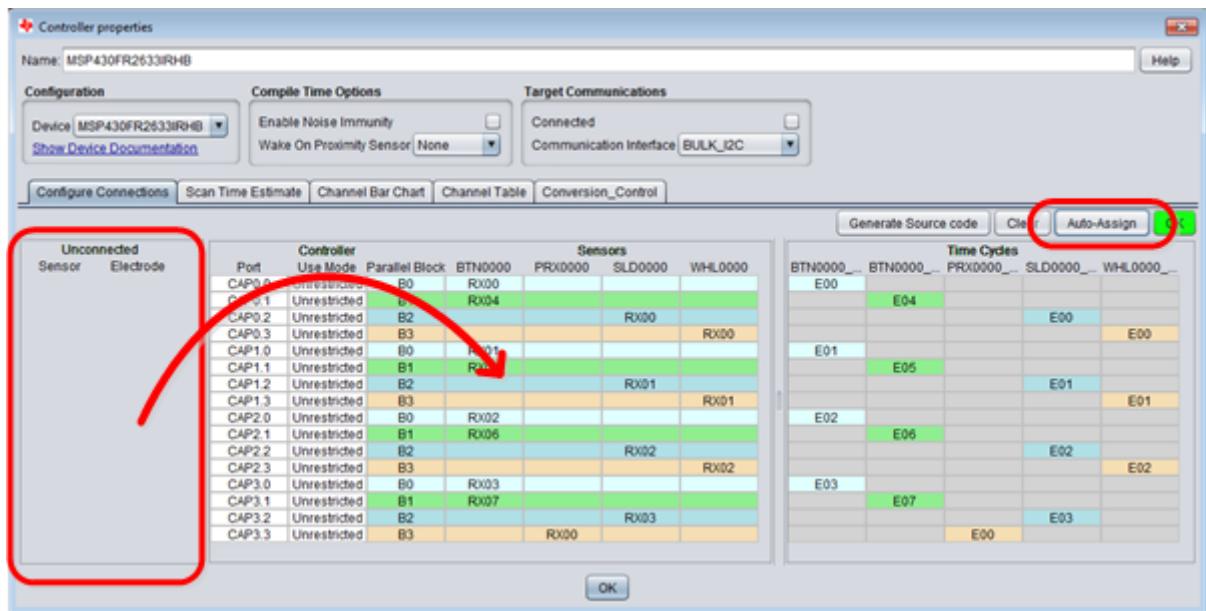


Figure 7.14: CapTIvate\texttrademark {} Auto-Assignment Feature

## 7.2.5 Generate source code

Select the "Generate Code" button on the MSP430 Controller properties dialog.

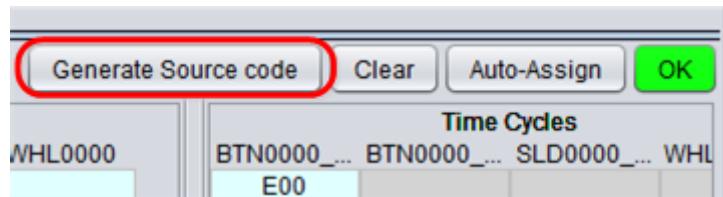


Figure 7.15: Generate Source Code Button

Select "Create new project" then "OK" in the dialog indicating that you want to create a new full project.

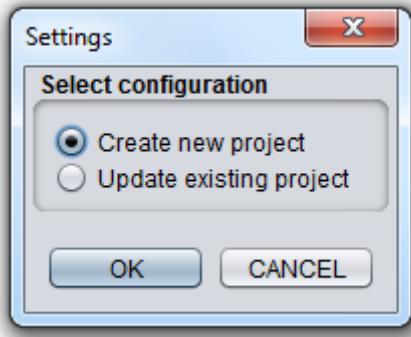


Figure 7.16: Select New Project

Select "OK" to keep the default location for the generated code.

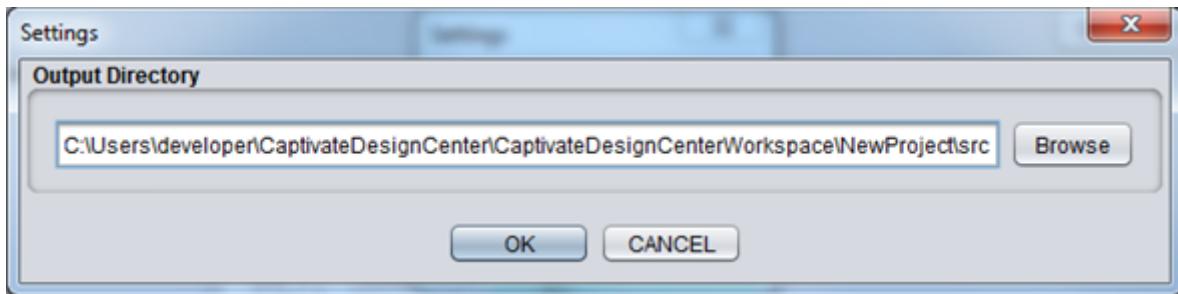


Figure 7.17: Use Default Location

## 7.2.6 Load and run the MSP430FR26xx/25xx generated firmware project using CCS or IAR

For information regarding importing/opening project in CCS or IAR and running the generated projects please refer to [Loading and Running Generated Projects](#)

## 7.2.7 Communicating with the target

This section shows some examples of how the data read from the target over HID is displayed.

Note that you will not see any valid sensor data unless your target board is connected to a sensor board that matches the configuration created in the Design Center GUI.

## Start HID communication with the target

Enable the HID communication by selecting the menu "Communications->Connect." Verify the HID device is connected by viewing the message in the main window lower left corner.



Figure 7.18: HID Connection Status

## Display sensor data

Double click on the slider object to display the properties dialog. The tabs are used to display graphs of the real-time sensor data, and also to enable reading and modifying the sensor's configuration and tuning parameters. The sensor "position" is also reflected in the display.

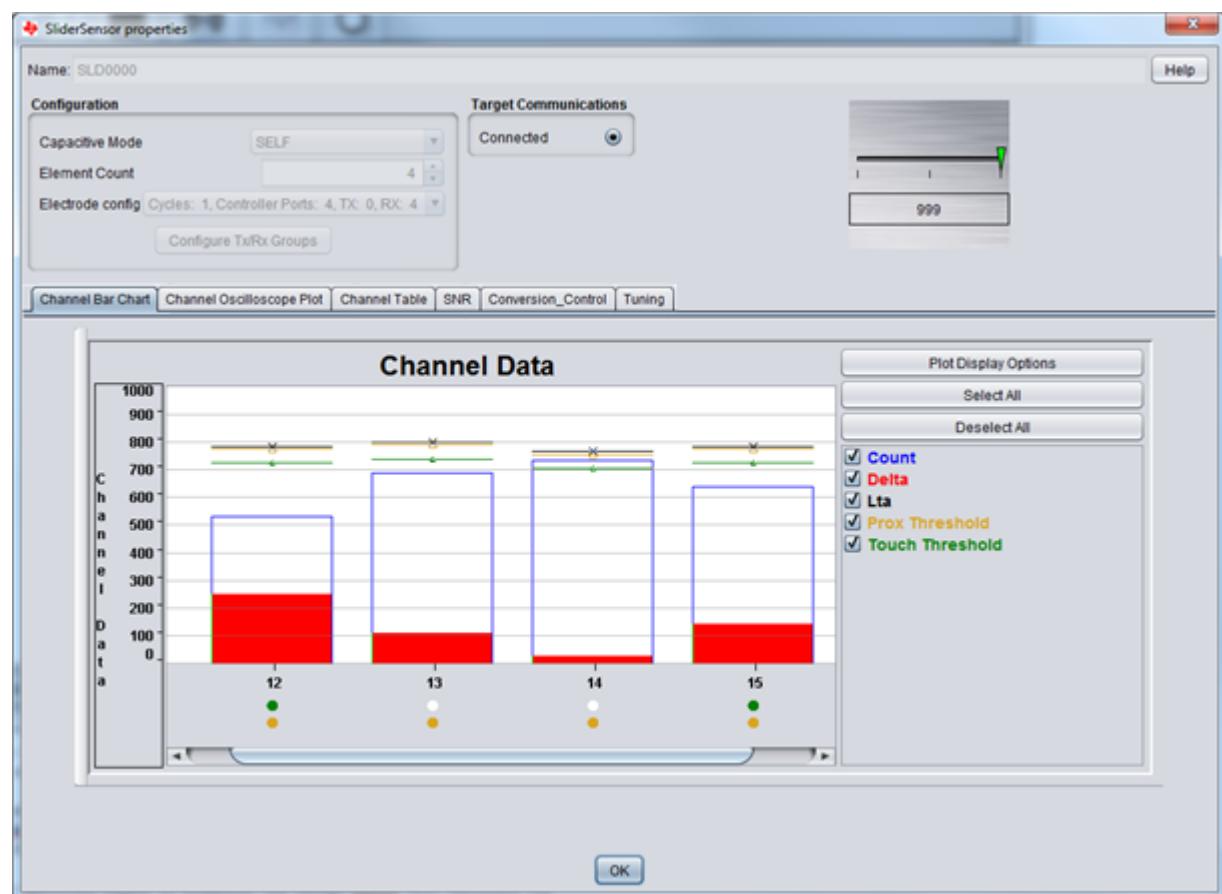


Figure 7.19: Channel Bar Chart

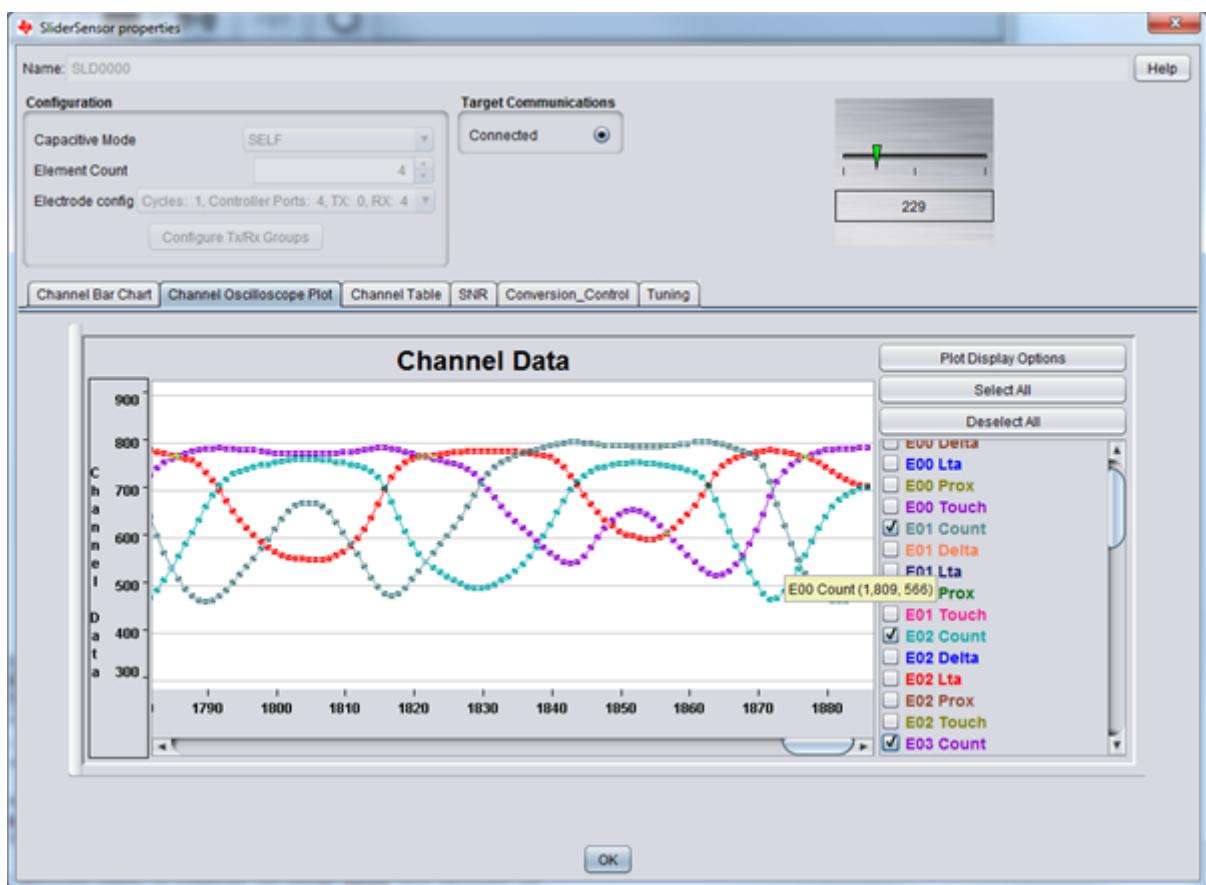


Figure 7.20: Oscilloscope Chart

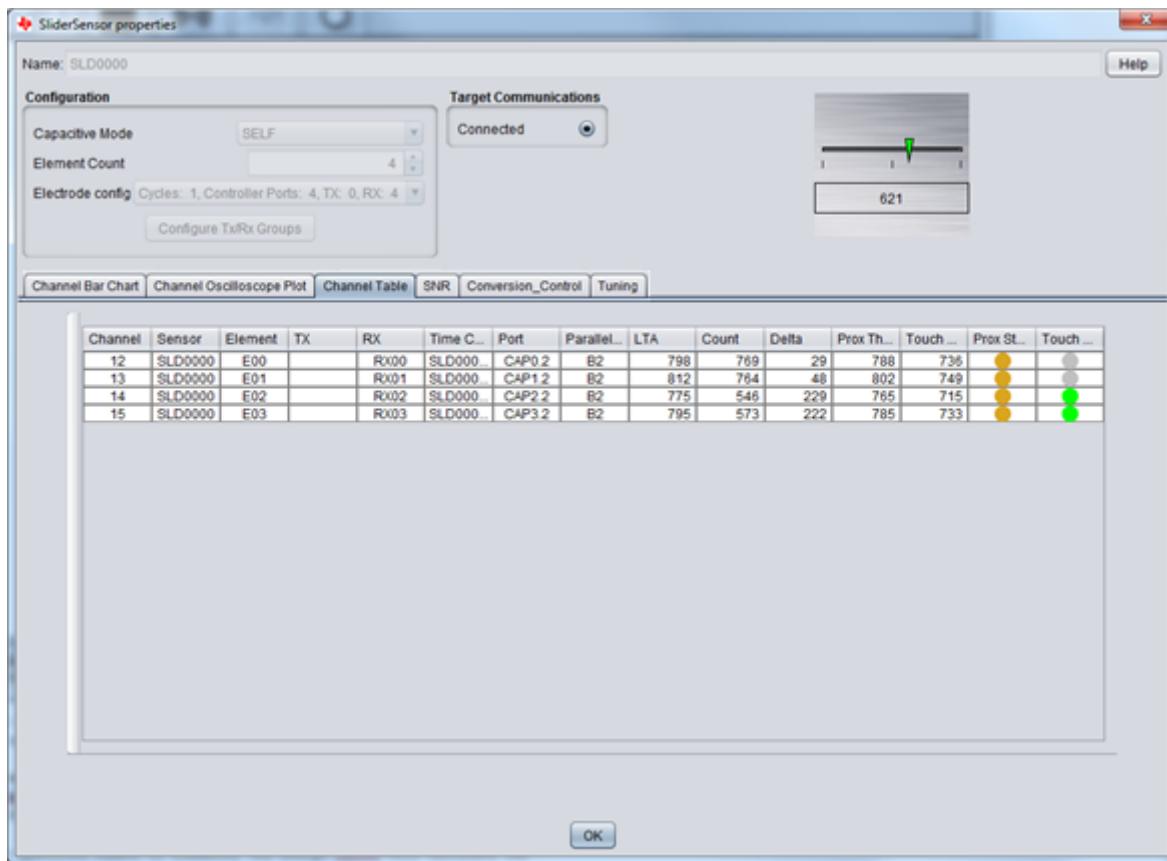


Figure 7.21: Channel Table

#### Tune sensor performance

Sensors can be configured in real-time using the Tuning panel. Modify the desired parameter then click "Apply". All parameters are described directly in the panel for ease of use. Sensor parameters can only be modified when the target is connected.

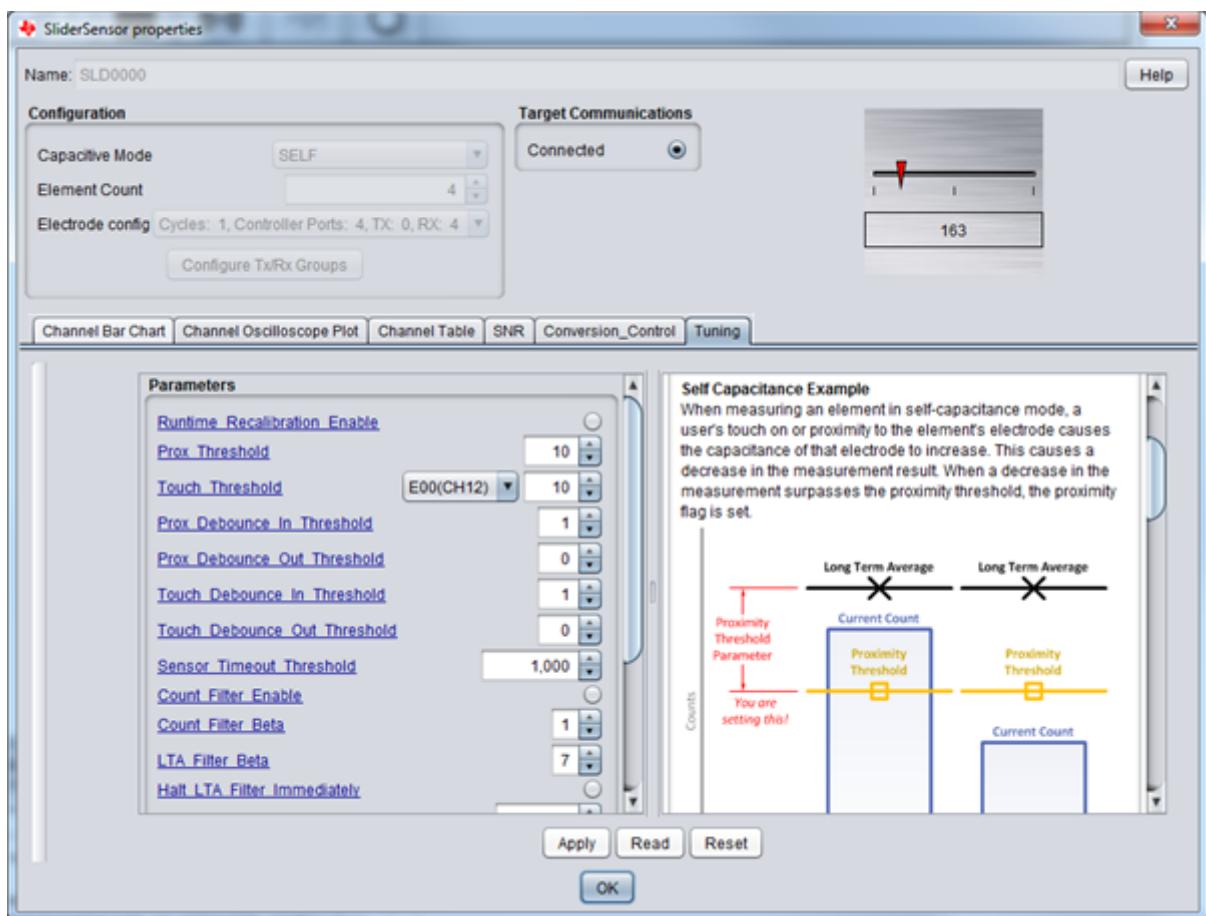


Figure 7.22: Sensor Tuning Parameters

It is possible to "un-dock" any of the tuning views to allow easier viewing of one or more views at one time. Click and drag the handle on the left of the view to position as needed. Closing an un-docked window returns it to the tab pane.

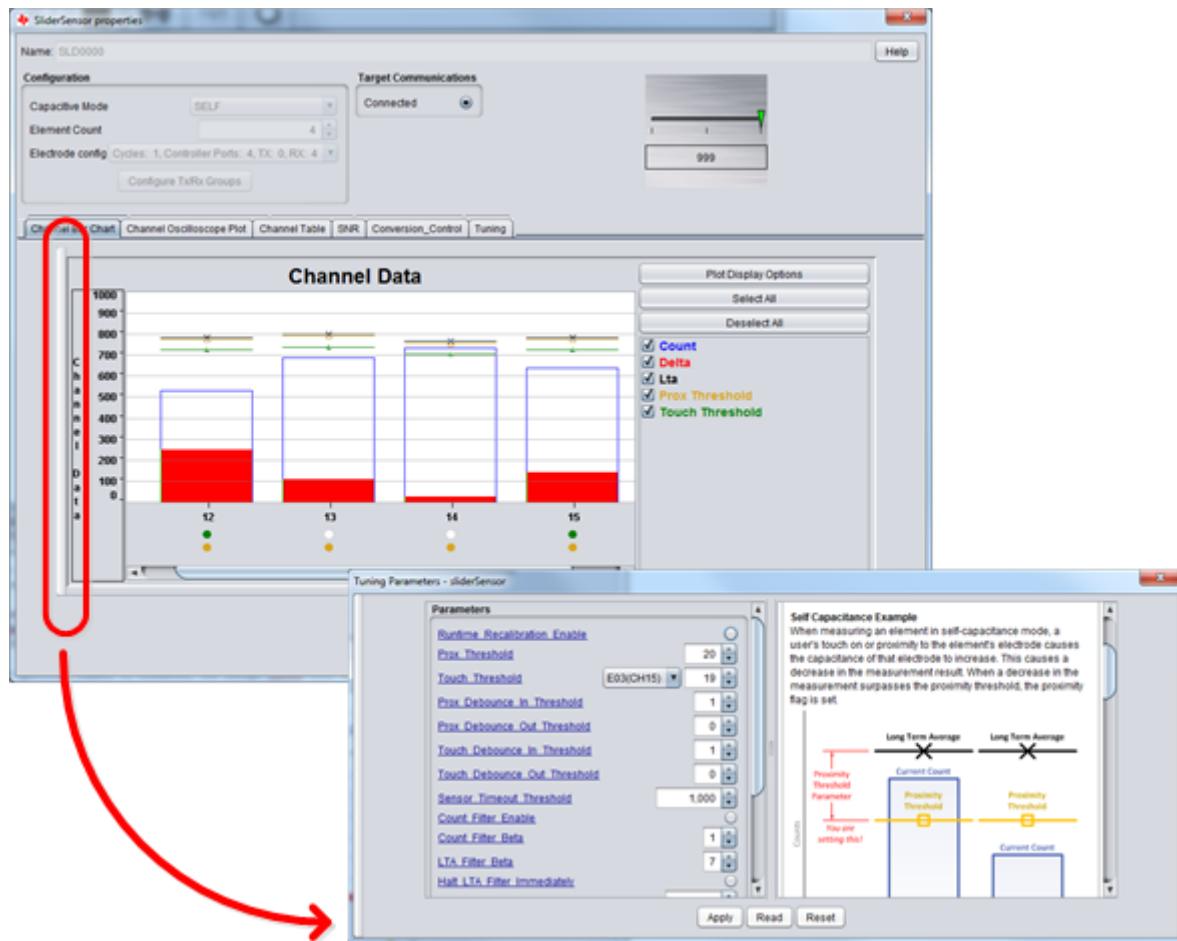


Figure 7.23: Un-docking views

For additional detailed information about the features of the CapTIVate™ Design Center, refer to the [CapTIVate™ Design Center user's guide](#).

When ready to start your own design, check out the workshop guide section, [Creating a new sensor design project](#).

## 7.3 CapTIVate™ Design Center Users guide

### 7.3.1 Design Center GUI panels

The Design Center GUI consists of three main panels; Menu bar, Object selection tab and Design canvas.

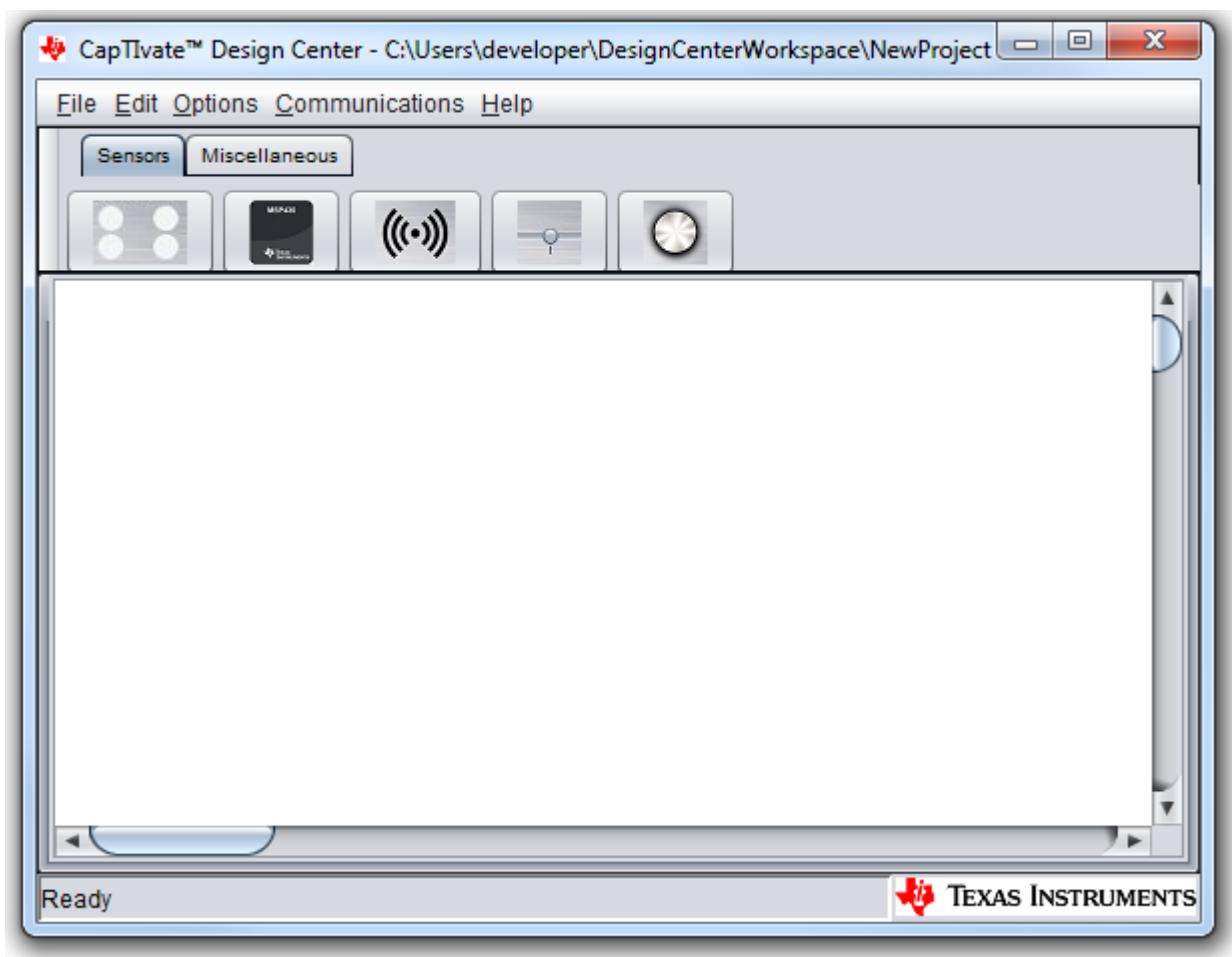


Figure 7.24: CapTivate™ Design Center GUI

### 7.3.1.1 Menu bar

The menu bar provides access to the GUI's features and controls.



Figure 7.25: Menu Bar

### 7.3.1.2 Object Selection Tab

The object selection tab displays objects can be selected for placement in the design canvas. Object types include:

- Sensors
- MSP430 controller
- Miscellaneous objects such as comments



Figure 7.26: Object Selection Bar

### 7.3.1.3 Miscellaneous Selection Tab

The Miscellaneous selection tab displays a comment object and user data logging object.

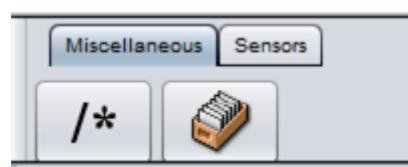


Figure 7.27: Object Selection Bar

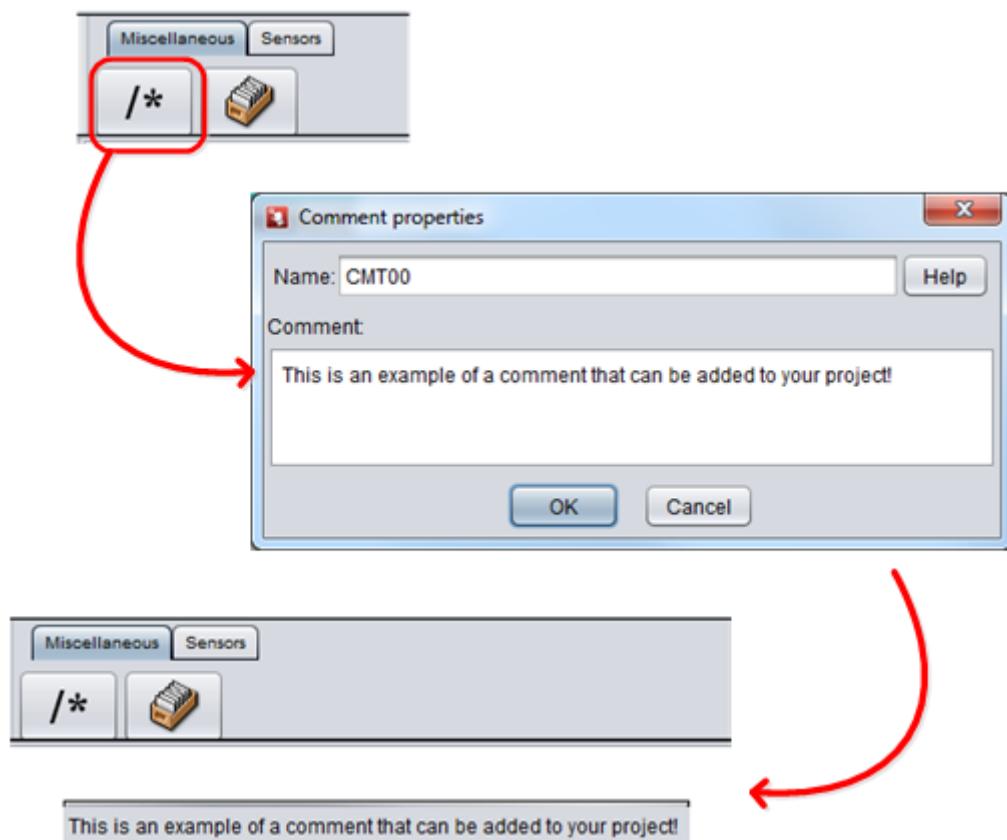


Figure 7.28: Adding a comment

#### Comment Object

The comment object is used to create user comments that can be added to the design project.

### User Data Logging Object

The user data logging object provides a means to log user defined data. For more information about how to use this feature with firmware running on the target MCU, refer to the [Software Library Chapter](#). User data is collected and stored in a `userDataLog.csv` file. Refer to [Communications Section](#) for details on the data enable/disable collection process.

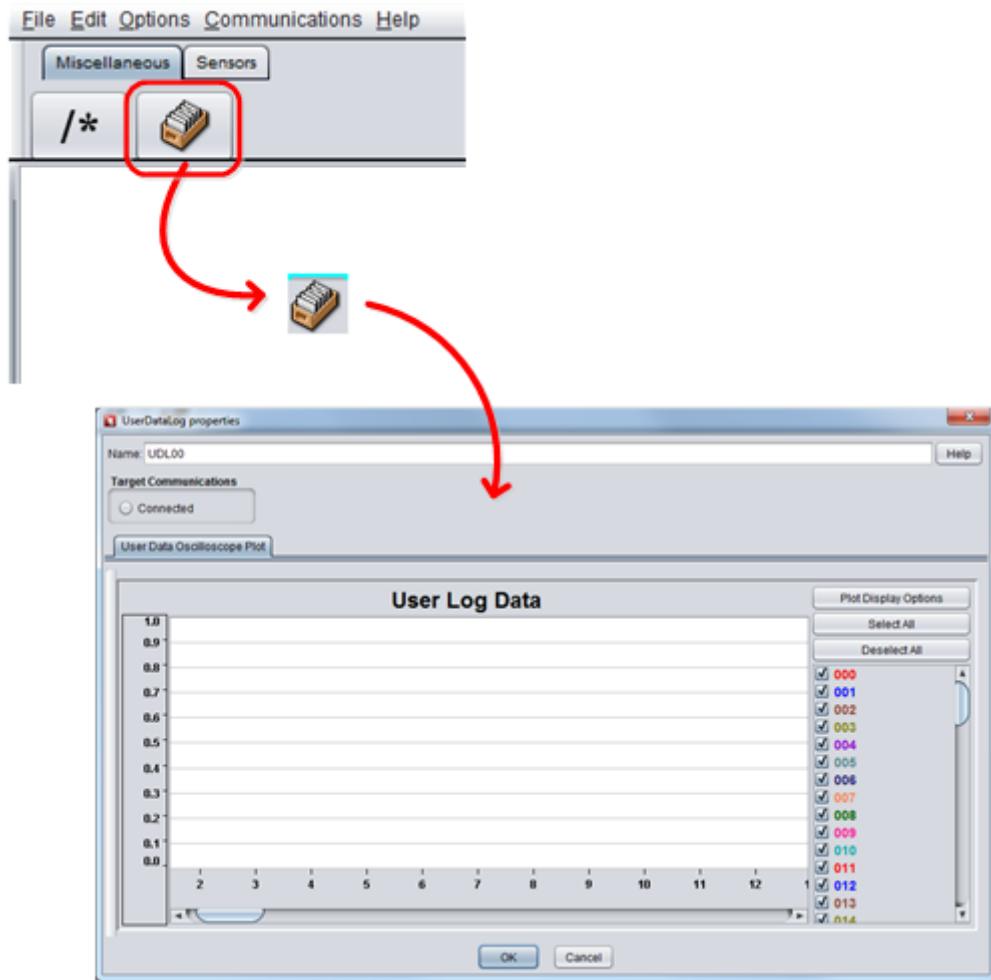


Figure 7.29: Adding a comment

#### 7.3.1.4 Design Canvas

To place an object in the design canvas:

- Select an object from the object tab with the left mouse button
- Position the cursor in the design area and place the object by clicking the left mouse button

Objects may be selected and moved anywhere on the canvas. Additional object operations can be displayed by right-clicking the mouse button on the object. To display the properties and configure the object, double-click using the left mouse button. To access the on-line help for an object type, select it and press the F1 key.

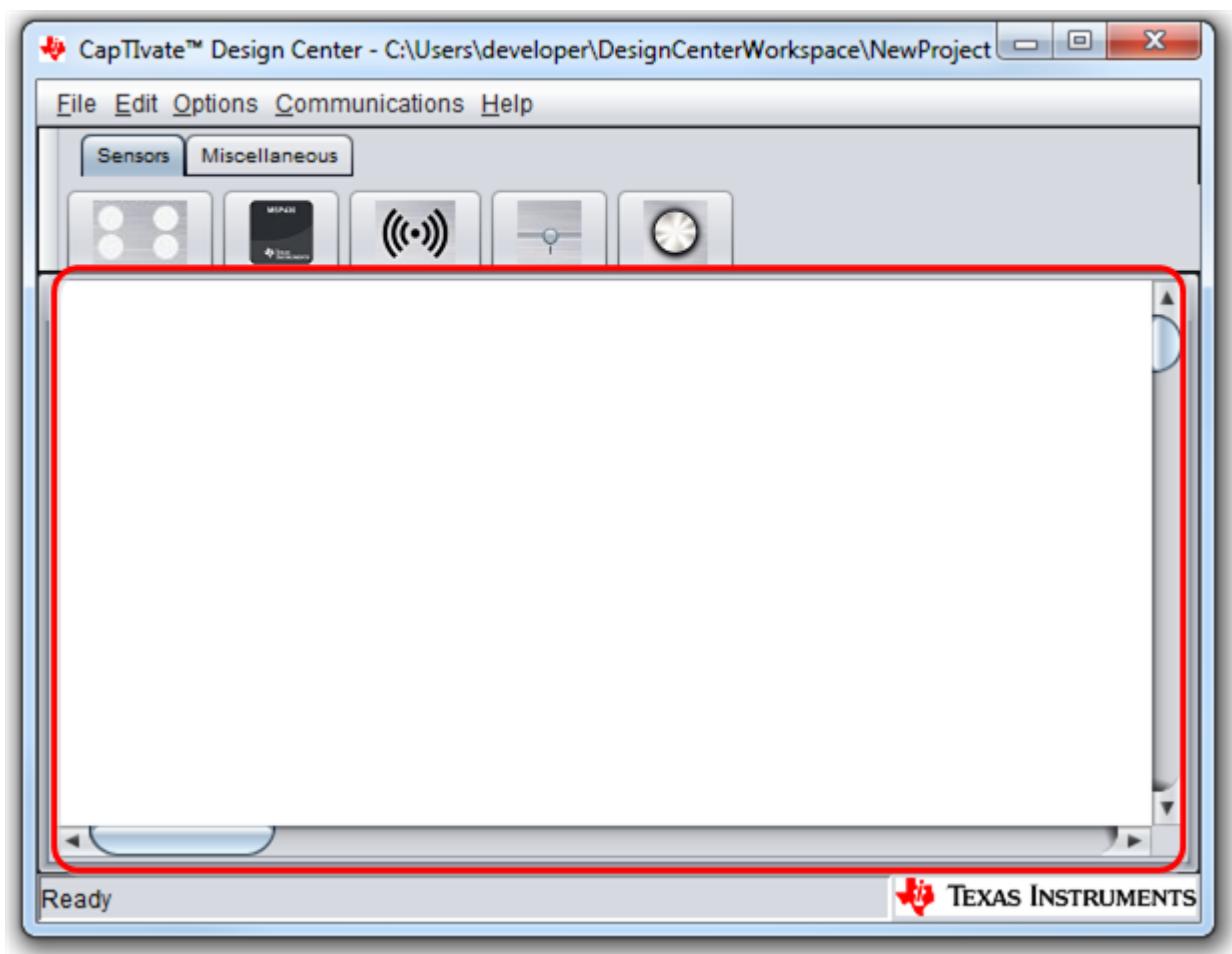


Figure 7.30: Design Canvas

---

## 7.3.2 Menu Descriptions

This menu item provides selections for projects, workspaces and printing.

### 7.3.2.1 File

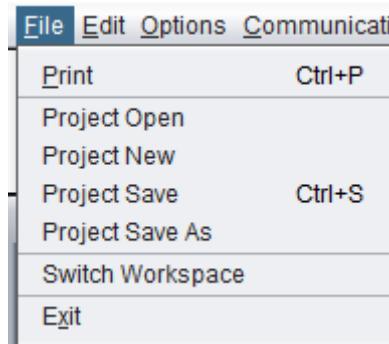


Figure 7.31: Menu - File

#### 7.3.2.1.1 Print

Allows the user to print the window.

#### 7.3.2.1.2 Project Open/Save/SaveAs

Allows the user to:

- Open a new or existing project in the current workspace
- Save the current project
- Save the current project as a different name

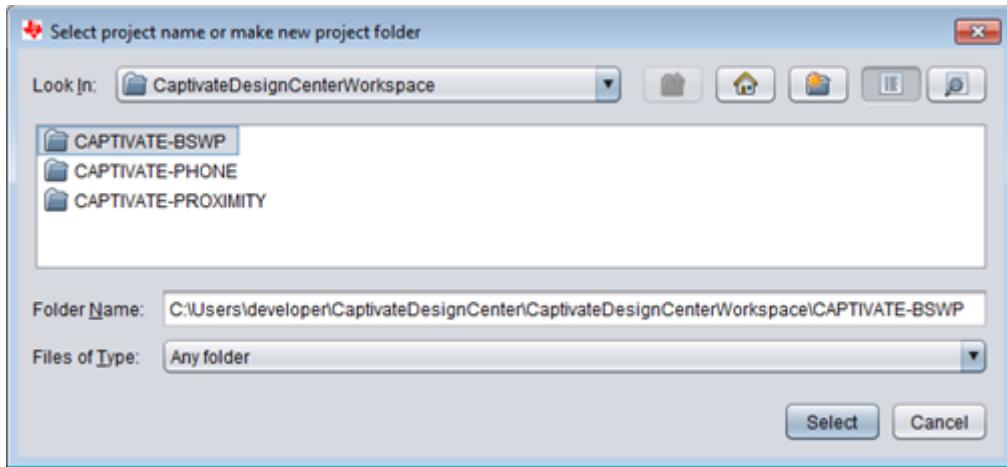


Figure 7.32: Menu-Project Save

#### 7.3.2.1.3 Switch Workspace

The design center stores projects within a workspace directory. This menu allows the user to change open a different workspace. This behavior is similar to the way TI Code Composer Studio uses workspaces and projects.

The default workspace path is USERPROFILE%/CaptivateDesignCenter/CaptivateDesignCenterWorkspace.

#### 7.3.2.1.4 Exit

Exit the CapTivate™ Design Center application.

#### 7.3.2.2 Edit

This menu item provides options for deleting or restoring one or more selected objects. For example, if the canvas is populated with multiple sensors, to clear the canvas simply select all, then cut (delete).

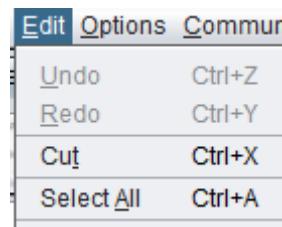


Figure 7.33: Menu-Edit

#### 7.3.2.2.1 Undo/Redo

Undo or redo a previous edit menu operation.

---

#### 7.3.2.2.2 Cut

Remove the selected object from the design

#### 7.3.2.2.3 Select All

Select all the objects in the design canvas

#### 7.3.2.3 Options

Allows the user to customize the display of the design canvas.

##### 7.3.2.3.1 Display

Modifies the canvas.

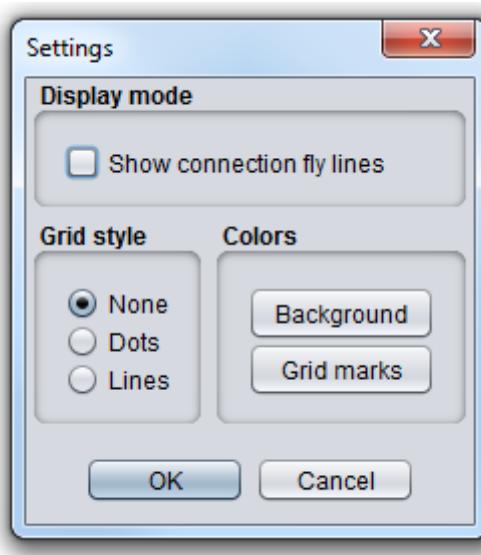


Figure 7.34: Menu-Options-Display

#### 7.3.2.3.2 Features

Changes the feature mode. In "Advanced" mode, more sensor tuning and configuration options are displayed.

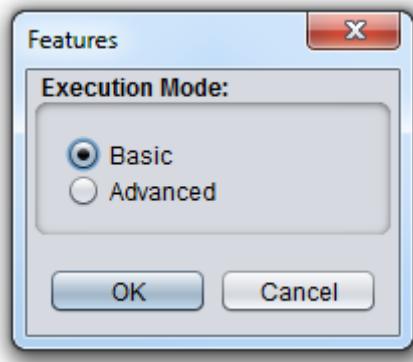


Figure 7.35: Menu-Options-Features

#### 7.3.2.4 Communications

Modifies the communications.

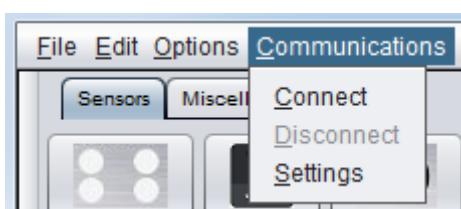


Figure 7.36: Menu-Communications

##### 7.3.2.4.1 Connect/Disconnect

Controls whether the Design Center [communicates with the target board over HID](#).

##### 7.3.2.4.2 Settings - HID

###### HID

Enables the user to change the HID connection parameters. **NOTE:** The parameters default to the correct values for TI's CapTIvate™ Technology EVMs.

##### 7.3.2.4.3 Settings - Logging

Enabling data logging will allow element, sensor and user data to be logged to \*.csv files automatically generated in the project's directory as shown.

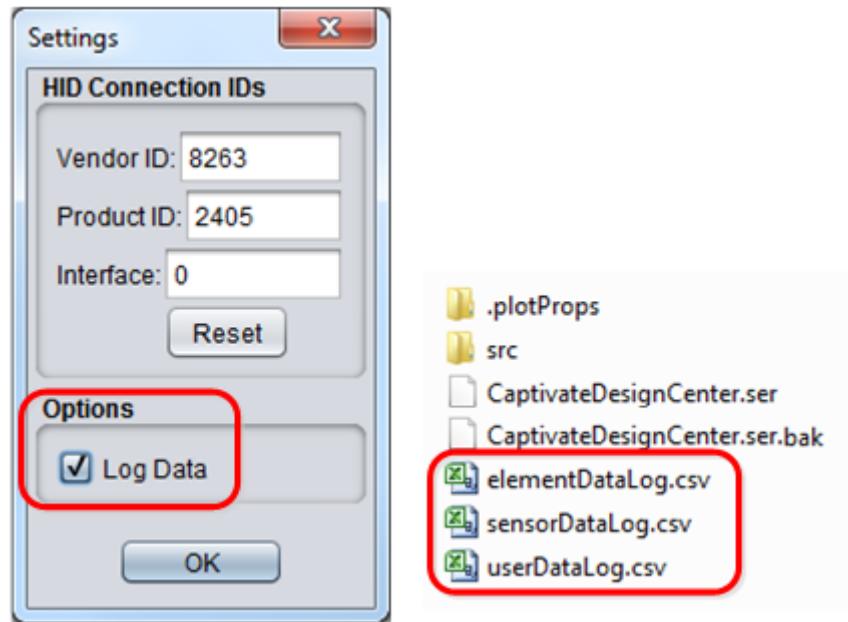


Figure 7.37: Enable Data Logging

After logging has been enabled, use the communications enable/disable to collect and append data to the log files.

**NOTE:** Disabling then re-enabling the Log Data feature will delete any previous data stored in the log files.

#### 7.3.2.5 Help

Access to the CapTlve™ Documentation and software version.

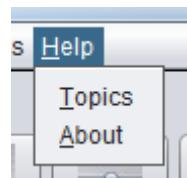


Figure 7.38: Menu-Help

#### 7.3.2.6 Topics

Displays the help documentation in a browser window.

#### 7.3.2.7 About

Displays software version information.

### 7.3.3 Controller Properties

Double clicking on the controller object will display the properties dialog.

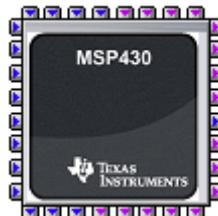


Figure 7.39: Controller

The properties dialog for a **controller** object allows the user to:

- configure the device
- map **controller** pins to sensor **elements**
- enable **target communications** and view sensor data

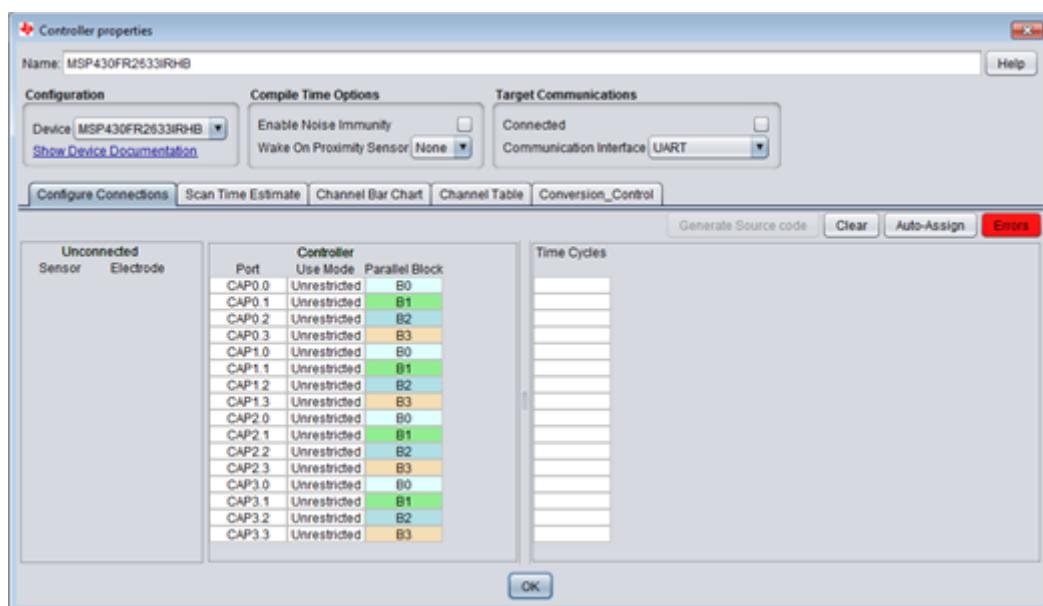


Figure 7.40: Controller Properties

#### 7.3.3.1 Device configuration

Select the desired MSP430 device with CapTIvate™ Technology from the pull-down list. When using the CapTIvate™ EVM, choose the MSP430FR2633IRHB.



Figure 7.41: Selecting Device

### 7.3.3.2 Enable target communications and target MCU communications interface

Enable/disable [target communications](#) using this button. This is equivalent to the [Communications->Connect/Disconnect](#) menu items. The Communication Interface provides UART and I2C selections for the HID Bridge MCU to target MCU communications interface. To change the current interface, select a new one and click the "Connected" button. The HID Bridge MCU on the CAPTIVATE-PGMR PCB will immediately start using the new interface. When generating source code, the generated project files will be updated to provide support for the new interface.



Figure 7.42: Selecting Communications

### 7.3.3.3 Select Compile Time options

Compile options can be selected to enable additional features provided in software. These features, when selected, are added to the code project generated by the CapTIvate™ Design Center.

- [Enable Noise Immunity](#) for improving noise immunity when operating in noisy environments.
- [Wake on Proximity](#) enables ultra-low power proximity detection using the hardware state machine (no CPU) and the selected sensor. The sensor list varies with the types and numbers of sensors in the project. Additional parameters, such as low power scan rate, inactivity timeout and others can be specified in the [Controller Conversion Control](#).

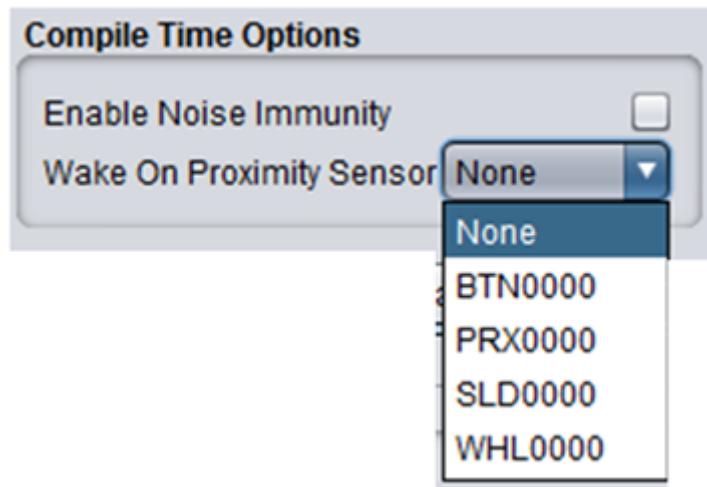


Figure 7.43: Compile Time Options

#### 7.3.3.4 Mapping sensor ports to controller pins

There are three ways to map the [sensor ports](#) to controller pins:

1. Allow the Design Center to automatically map all the connections using the "Auto-Assign" function on the controller "Configure Connections" tab
2. Manually map the connections using the connection table in the controller "Configure Connections" tab
3. Graphically connect the sensor and controller pins on the objects in the design canvas

##### 7.3.3.4.1 Auto-assign

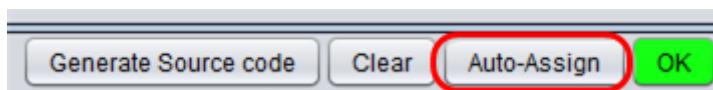


Figure 7.44: Auto-Assignment Button

If the "Auto-Assign" button is selected, the Design Center will attempt to optimally connect all of the *unconnected* the [sensor ports](#) to the controller. Note that any connections that already exist will not be changed. In this manner, it is possible to make some connections manually, and then auto-assign the rest.

This is a good option if there are no pre-existing restrictions on the mapping. For example, if the mapping is constrained due to board layout considerations, users should choose one of the other options.

If the assignment completes successfully, the "Errors" indicator will turn green. If the indicator is "red", errors still exist. Clicking on the indicator will display a dialog with a listing of the remaining errors.

##### 7.3.3.4.2 Manual mapping using the Configure Connections table

The screenshot shows the 'Configure Connections' tab of the Design Center GUI. The main area is a table titled 'Controller' with columns: Port, Use Mode, Parallel Block, and Sensors (BTN0000, PRX0000, SLD0000, WHL0000). To the left is a list of 'Unconnected' sensor/electrode pairs. To the right are two smaller tables: 'Time Cycles' and 'Errors'.

Sensor	Electrode	Port	Use Mode	Parallel Block	BTN0000	PRX0000	SLD0000	WHL0000
PRX0000	RX00	CAP0.0		B0	RX00			
SLD0000	RX00	CAP0.1		B1	RX04			
SLD0000	RX01	CAP0.2		B2				
SLD0000	RX02	CAP0.3		B3				
SLD0000	RX03	CAP1.0		B0	RX01			
WHL0000	RX00	CAP1.1		B1	RX05			
WHL0000	RX01	CAP1.2		B2				
WHL0000	RX02	CAP1.3		B3				
		CAP2.0		B0	RX02			
		CAP2.1		B1	RX06			
		CAP2.2		B2				
		CAP2.3		B3				
		CAP3.0		B0	RX03			
		CAP3.1		B1	RX07			
		CAP3.2		B2				
		CAP3.3		B3				

Figure 7.45: Manual Mapping using Table

The Configure Connection table enables detailed control for mapping the sensor ports to the controller pins. The table is split into 3 main parts:

- The "Unconnected" column lists all the sensor ports that are not assigned to a controller pin. The data in this column is read-only.
- The "Controller" columns display information on the CapTivate™ Technology enabled ports on the MSP430, any usage mode restrictions, and which of the ports belong to the same parallel sense block.

#### 7.3.3.4.3 Port column edits

The screenshot shows the 'Controller' table with a dropdown menu open over the 'Use Mode' column for CAP0.0. The menu options are: Available, Reserved, RX only, TX only, SELF only, and MUTUAL on.

Port	Controller	Parallel B
Port	Use Mode	Parallel B
CAP0.0		B0
CAP0.1		B1
CAP0.2		B2
CAP0.3		B3
CAP3.2		B2
CAP3.3		B3

Figure 7.46: Editing Port Column

---

Selecting a cell in the port column will bring up a pull-down menu that allows the user to control the usage mode of the port. The "Use Mode" column will be updated to display any restrictions made a controller pin.

- Available: Any sensor port may be assigned
- Reserved: No sensor port may be assigned to this pin
- RX only: Only sensor RX ports may be assigned to this pin
- TX only: Only sensor TX ports may be assigned to this pin
- SELF only: Only [self capacitance](#) sensor ports may be assigned to this pin
- MUTUAL only: Only [mutual capacitance](#) sensor ports may be assigned to this pin

#### 7.3.3.4.4 Parallel Block column edits

Controller		
Use Mode	Parallel Block	BTN
	B0	
	B1	R
	B2	
	B0	
	B1	R
	B2	
	B3	
	B0	R
	B1	R
	...	

Figure 7.47: Editing Parallel Blocks

Selecting a cell in the port column will bring up a pull-down menu that allows This column allows the user to modify which pins are assigned to each [parallel sense block](#).

- The "Sensors" section contains a column for each sensor in the design and is used to enable users to individually map each sensor port to the desired controller pin.

#### 7.3.3.4.5 Sensor column edits

Controller		Sensors		
Use Mode	Parallel Block	BTN0000	PRX0000	SLD
	B0	RX00		
	B1			
	B2			
	B3	RX01		
	B0	RX02		
	B1	RX04		
	B2	RX05		
	B3	RX06		
	B0			
	B1			

Figure 7.48: Editing Sensor Columns

Selecting a cell in a sensor column will display the list of unconnected ports for that sensor which are legally allowed to be assigned to that controller pin. As sensor ports are assigned, the Unconnected and Time Cycle tables are dynamically updated.

- The [Time Cycles](#) section displays which sensor ports will be scanned in parallel by the target. This data is read-only and is updated whenever changes to the sensor connections or configuration are detected.

#### 7.3.3.5 Making manual connections graphically

Connections between the controller and sensors can be made by clicking the port on the sensor and dragging the connection to the desired port on the controller.

**Note:** Be sure to display the connections first using the [Options->Display](#) menu item.

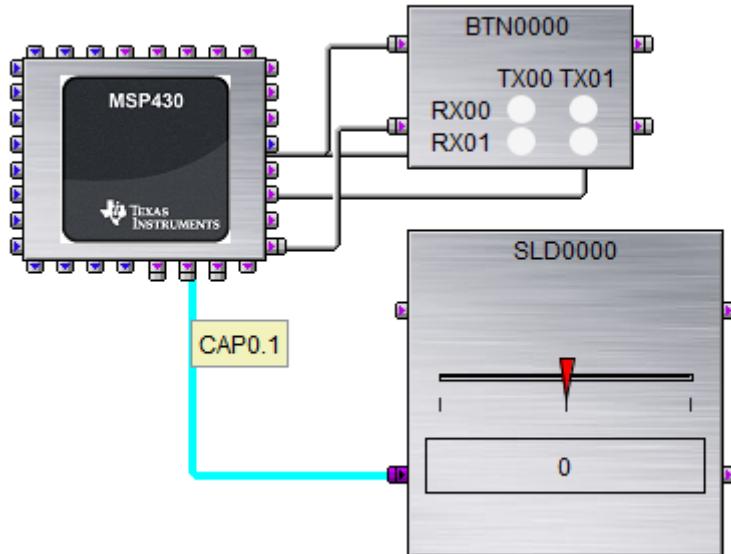


Figure 7.49: Graphical Connections

### 7.3.3.6 Source Code Generation

Source code for the design can be generated once all the sensor ports have been connected to the controller.



Figure 7.50: Generate Source Code Button

#### 7.3.3.6.1 Create new project

This option will create a complete project with the design specific definitions, support code, and CSS/IAR project files.

#### 7.3.3.6.2 Update existing project

This option will only update the design specific definition files. Use this option to update a project that has already been imported into an IDE.

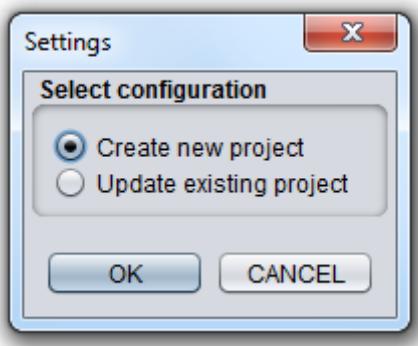


Figure 7.51: Select Output

### 7.3.3.7 Controller target data displays

#### 7.3.3.7.1 Controller Channel Bar Chart

See [Channel Bar Chart](#)

#### 7.3.3.7.2 Controller Channel Table

See [Channel Table](#)

### 7.3.3.8 Controller Conversion Control parameters

There are two groups of parameters that are controlled in this tab; one for data control, the other for wake on proximity. Clicking on any parameter will display a description for it in the "Description" viewing pane to the right of the parameter.

#### Element and Sensor data enables

These controls provide a method to minimize the amount of data sent from the target MCU to the CapTivate™ Design Center during real-time tuning. By default, these both should be "checked." For designs with large number of sensors or elements, the data transmission can be very large and potentially slow down the report rate during sensor tuning. Deselecting one or the other will help improve the throughput.

#### Wake on proximity parameters\*

When using wake on proximity for ultra-low power proximity detection, it is possible to select:

- Ultra-low power Wake on proximity scan rate is the rate at which the hardware state machine (no CPU) scans the selected proximity sensor/electrode. Note: up to 4 electrodes can be scanned in parallel.
- Active Mode scan rate is the rate at which the CPU scans all the sensors connected to the target MCU.
- Wake up interval sets a failsafe timeout used by the hardware state machine to force the CPU to wake up and run any background tasks periodically. The wakeup period is based on the "Wake on proximity scan rate" time the "N samples selected in drop down" value. This parameter is optional and can be disabled
- Inactivity timeout is the time the CPU continues to scan all the sensors after there is no proximity detection. At the end of the timeout, the MCU re-enters the ultra-low power wake on proximity mode. The timeout period is based on the "Active scan rate" times the "Inactivity Timeout" value.

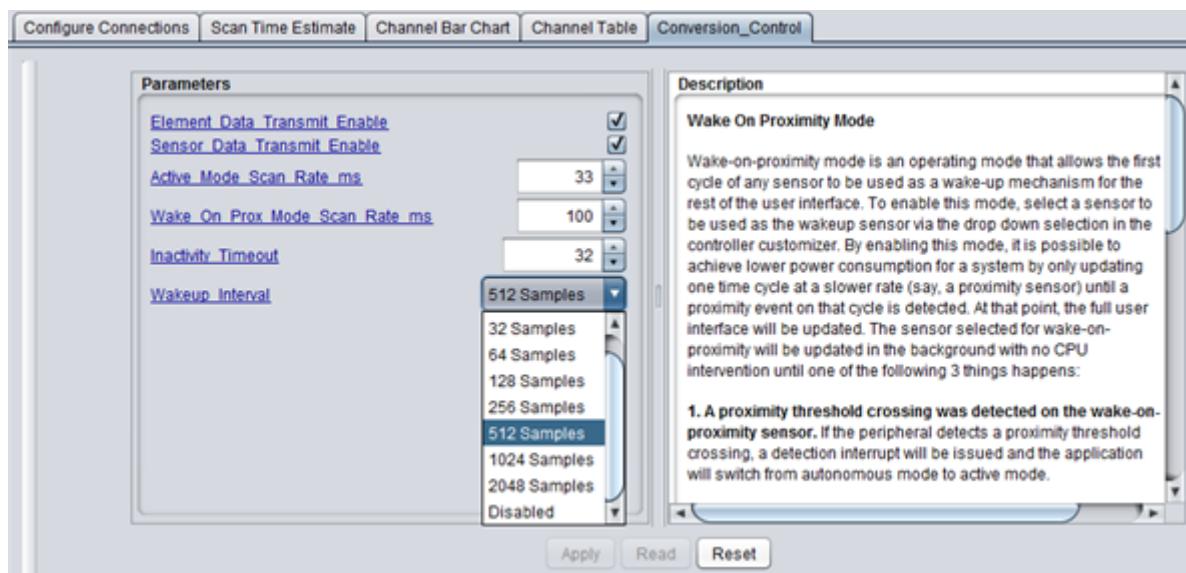


Figure 7.52: Wake On Proximity Parameters

### 7.3.3.9 Controller Scan Time Estimator

The scan time estimator displays the nominal amount of time it will take to measure each sensor, drawn with respect to the overall scan period. The overall scan period is controlled by the [System Report Rate](#) parameter.

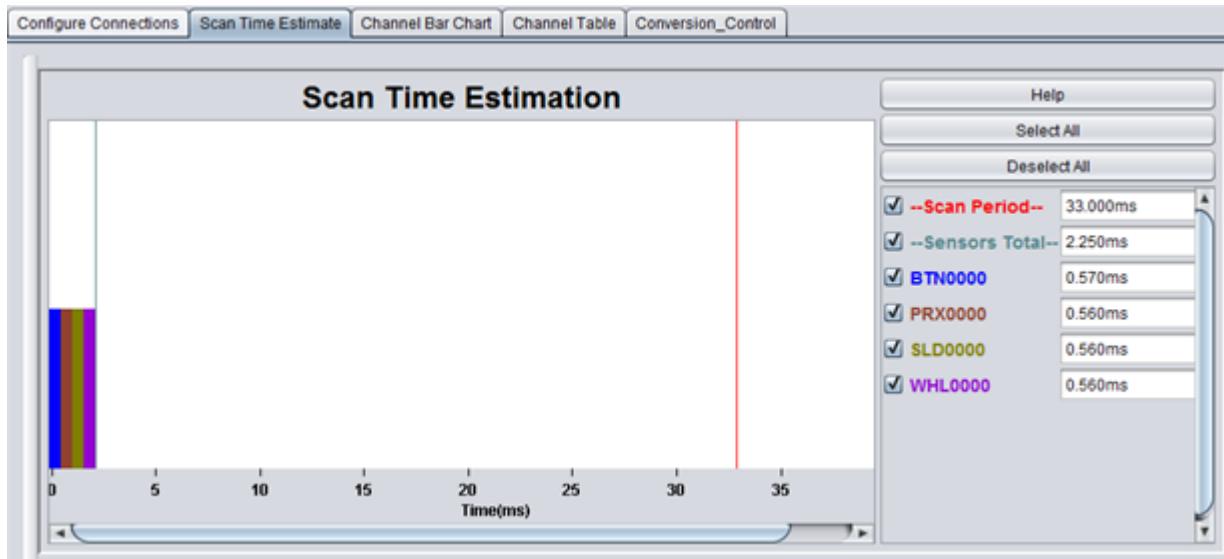


Figure 7.53: Scan Time Estimator View

### 7.3.4 Sensor Properties

Double clicking on a sensor object will display the properties dialog. The properties dialog for a sensor object allows the user to:

- configure the sensor
- tune the sensor
- enable [target communications](#) and view sensor data

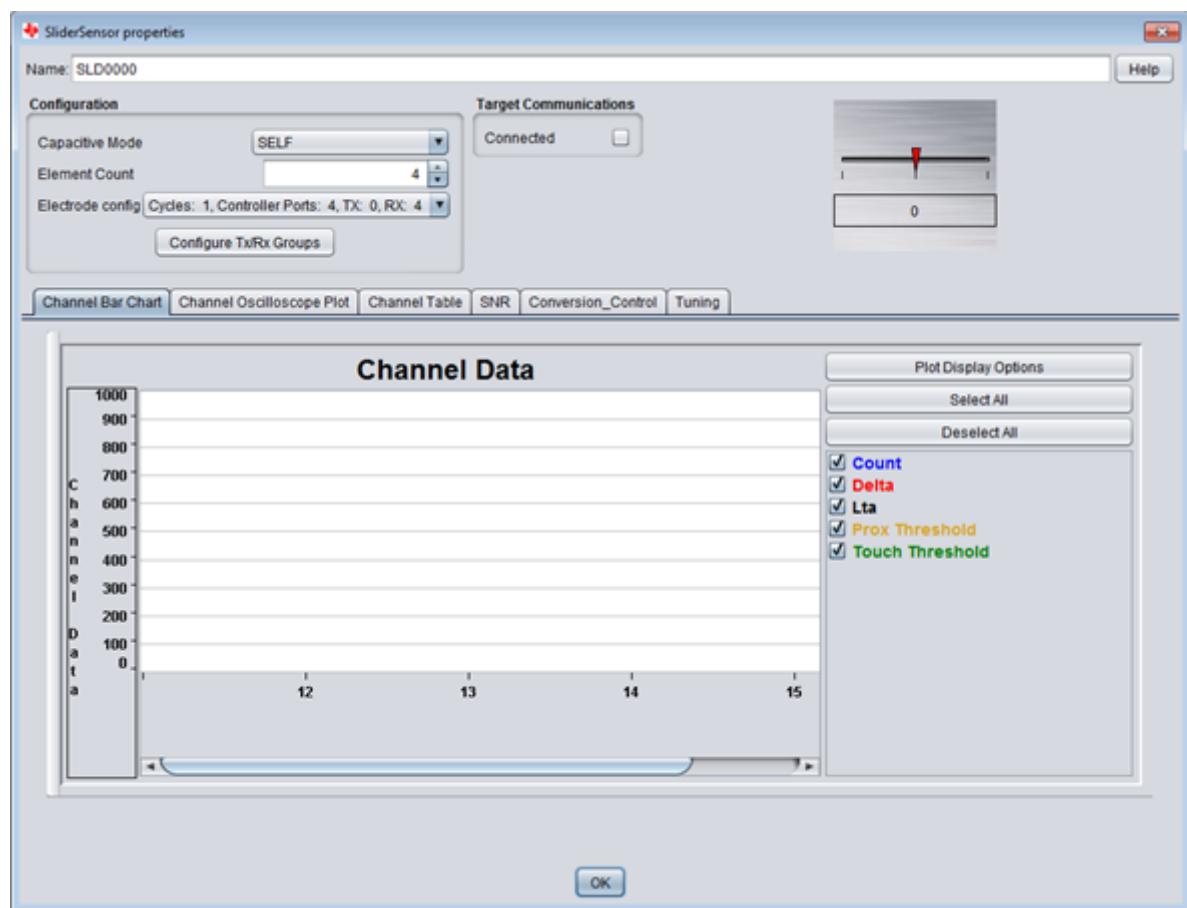


Figure 7.54: Sensor Properties View

#### 7.3.4.1 Position Display

Each properties dialog will display the "position" of the sensor. The rendering of this information is specific to the sensor type. The display is active whenever the Design Center [target communications](#) are active.

**Button Group Sensor** Each button in the group is displayed in a grid indexed by the sensor's TX/RX ports.

- White indicates the button is not being touched
- Yellow indicates that the [proximity threshold](#) has been reached
- Green indicates that the [touch threshold](#) has been reached and that a touch has been detected



Figure 7.55: Button Group Sensor

#### Slider Sensor

- Red indicates the slider is not being touched
- Green indicates that the [touch threshold](#) has been reached and that a touch has been detected
- The slider position is rendered on the display and the numeric value is also displayed below the GUI.

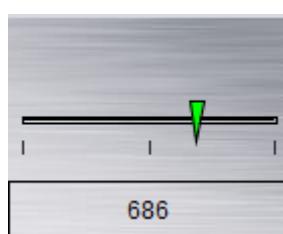


Figure 7.56: Slider Sensor

#### Proximity Sensor



Figure 7.57: Proximity Sensor

- White indicates the button is not being touched
- Yellow indicates that the [proximity threshold](#) has been reached
- Green indicates that the [touch threshold](#) has been reached and that a touch has been detected

#### Wheel Sensor

- Red indicates the wheel is not being touched
- Green indicates that the [touch threshold](#) has been reached and that a touch has been detected
- The wheel position is rendered on the display and the numeric value is also displayed below the GUI.



Figure 7.58: Wheel Sensor

#### 7.3.4.2 Sensor Configuration

The sensor configuration is specified through the properties dialog.

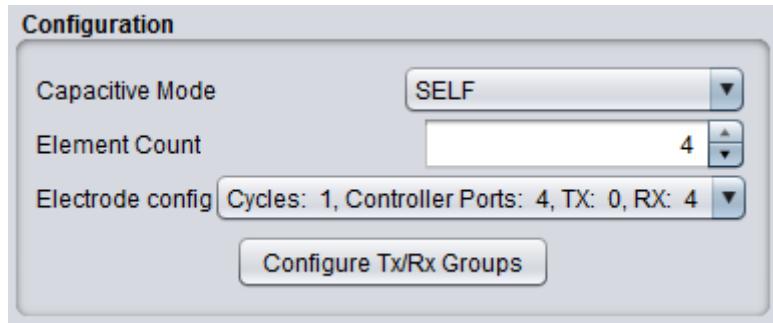


Figure 7.59: Sensor Configuration

#### 7.3.4.2.1 Capacitive Mode

Specify the capacitive mode type for the sensor. Choices are:

- Mutual
- Self

#### 7.3.4.2.2 Element Count

Specify the number of [elements](#) for sensor.

#### 7.3.4.2.3 Electrode Configuration

Based on the specified number of elements, the Design Center will calculate the possible [TX/RX port](#) configurations along with the number of [time cycles](#) that are required for each of the configurations. The Design Center will default to a configuration that takes the minimum number of [time cycles](#).

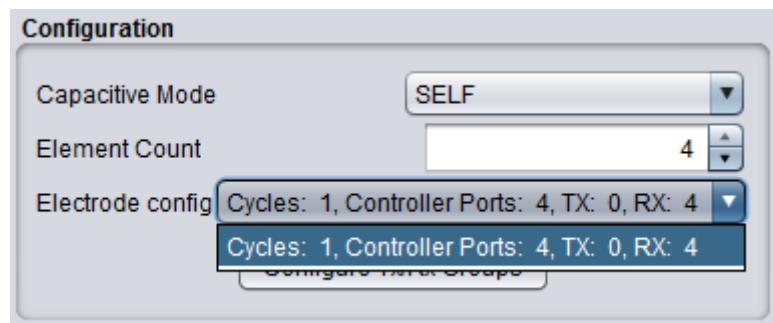


Figure 7.60: Electrode Configuration

#### 7.3.4.2.4 Configure TX/RX grouping

In some cases, it may be desired to change how the default assignment of the [sensor elements](#) to the TX/RX ports. Selecting the "Configure TX/RX Groups" will display a dialog where these mappings can be changed.

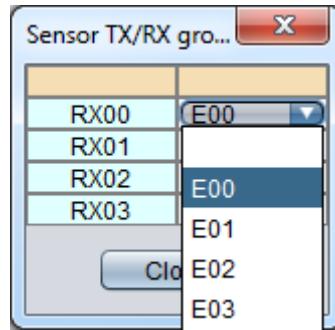


Figure 7.61: TX/RX Grouping

The example below shows a typical 4-element self capacitive slider with default TX/RX group assignment. Note how the Design Center assigns RX0..RX3 to one pin from each CapTlve™ block and to the sensor electrodes E00..E03. When designing the PCB it is important to follow these assignments generated by the Design Center. In the case of a slider or wheel, it is very important that RX0->E00, RX1->E01, RX2->E02, etc., else the algorithm will not work correctly.

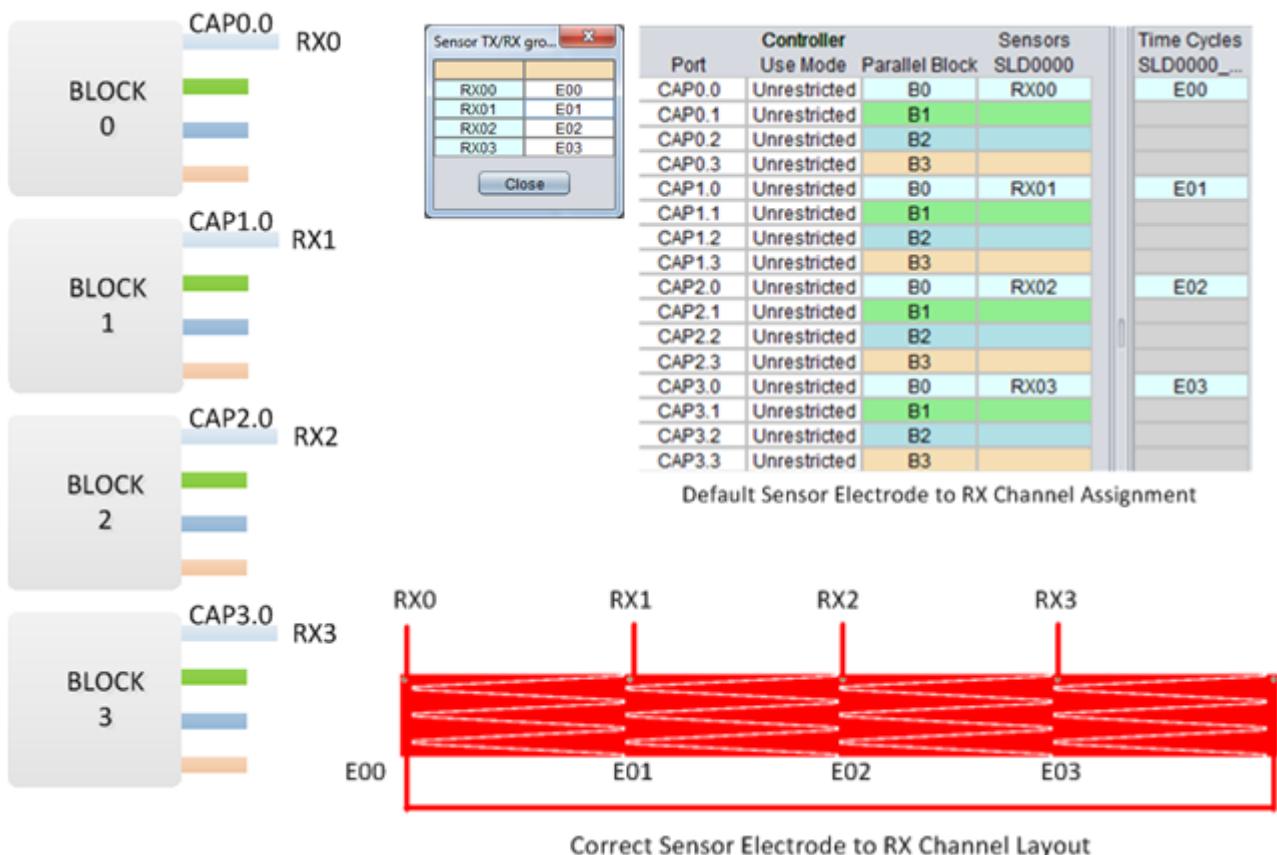


Figure 7.62: Typical TX/RX Sensor Group Assignment

There may be instances where a previous PCB design doesn't match the current Design Center configuration or the PCB routing was done incorrectly. Below is an example showing a 4-element slider with RX1 and RX2 swapped (RX1 assigned to slider element E02 and RX2 assigned to slider element E01).

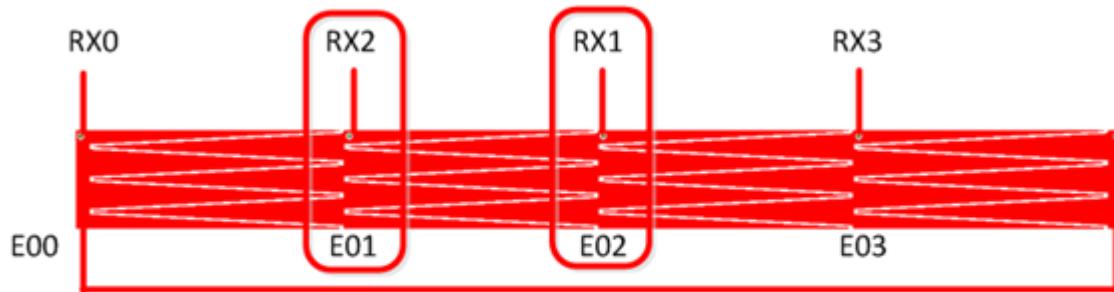


Figure 7.63: Incorrect TX/RX Sensor Group Assignment

The Design Center sensor TX/RX group assignments can be overridden to provide electrode assignments that match the PCB design. Below is an example showing how sensor electrode E02 is assigned to RX02 and E01 is assigned to RX01 using the Sensor TX/RX grouping tool. After the re-assignment the Controller properties view shows the new proper sensor electrode assignments and the software can now processes the slider in the correct order.

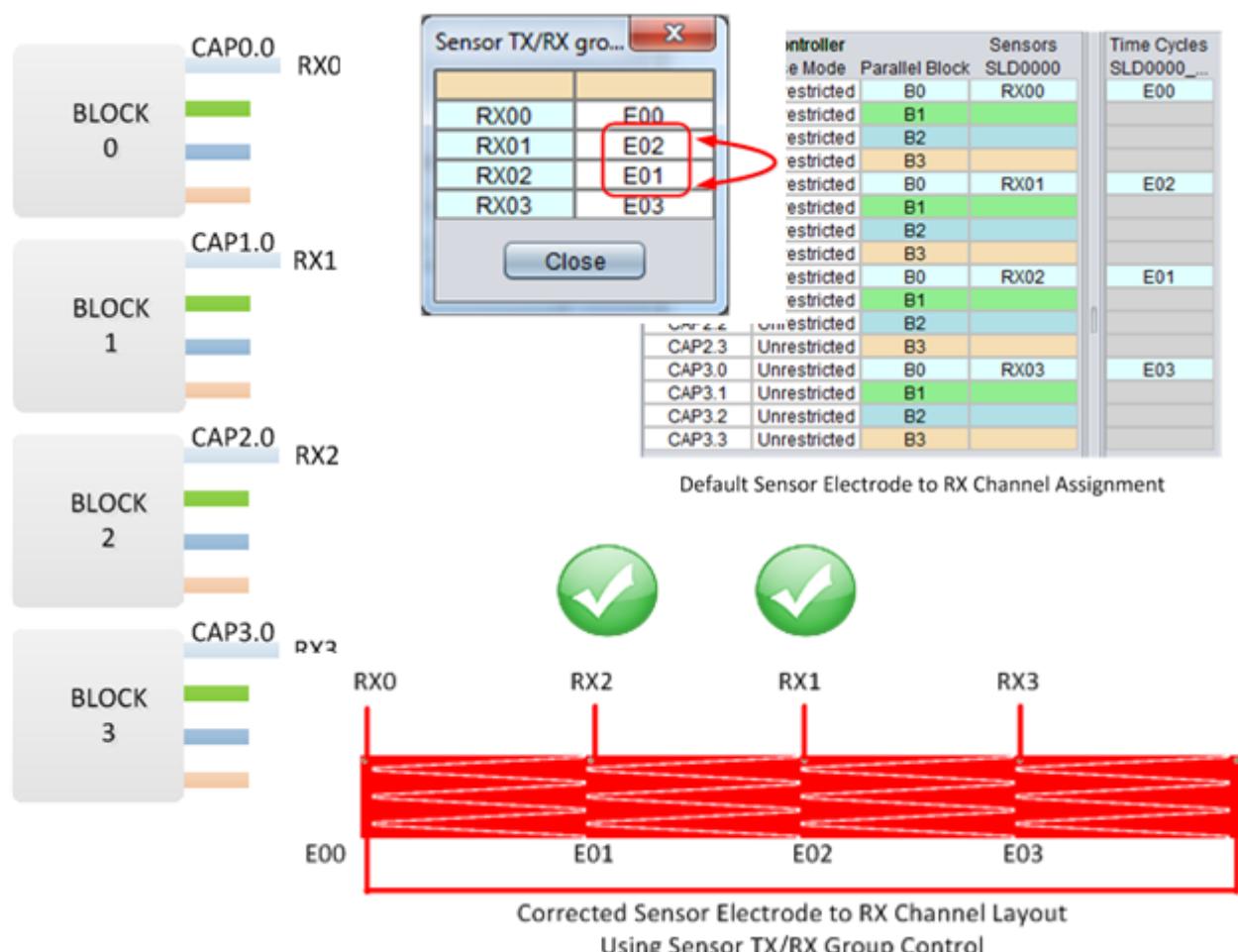


Figure 7.64: Corrected TX/RX Sensor Group Assignment

---

**Note**, this method provides a convenient way to reverse the behavior of either a slider or wheel if desired.

#### 7.3.4.3 Enable Target Communications from Sensor Properties

Enable/disable [target communications](#). This button is equivalent to the [Communications->Connect/Disconnect](#) menu items.



Figure 7.65: Communications Enable

#### 7.3.4.4 Sensor target data displays

##### 7.3.4.4.1 Sensor Channel Bar Chart

See [Channel Bar Chart](#)

##### 7.3.4.4.2 Sensor Oscilloscope Plot

See [Channel Table](#)

##### 7.3.4.4.3 Sensor Channel Table

See [Channel Table](#)

#### 7.3.4.5 Tuning Sensor performance

The sensor parameters are specified through the **Tuning** and **Conversion Control** tabs. Real-time reading and writing values to the target requires that [target communications](#) are enabled.

Parameter values are written to the appropriate structures whenever the source code is generated for the design. The values will take affect after the new source code is compiled and loaded to the target.

It is also possible to read and write values directly to the target in real-time when the [target communications](#) are enabled in the Design Center. This enables real-time tuning of sensor performance and can greatly reduce the time necessary to achieve the desired system behavior.

**Note:** that any values loaded in real time will revert to the original values when the target is reset unless new source code is generated, compiled and loaded to the target.

##### Tuning Parameters

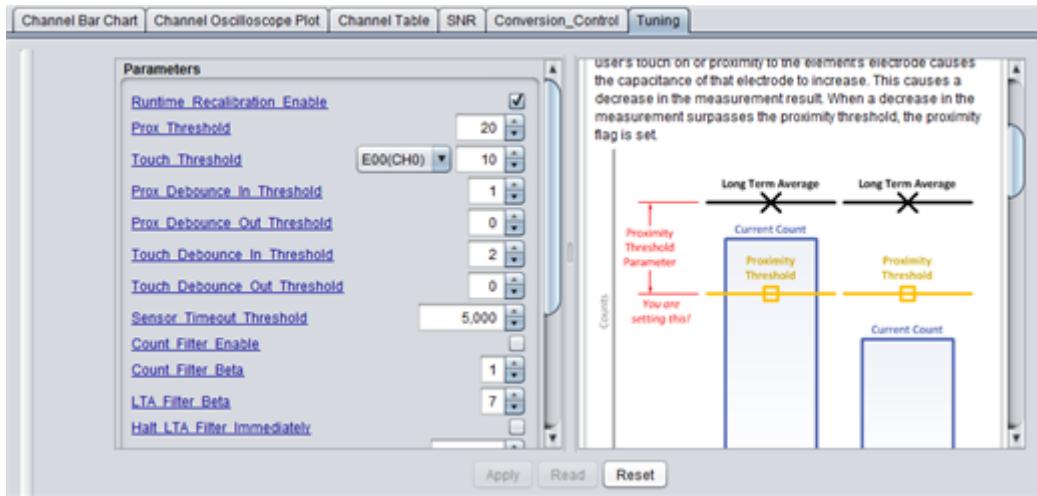


Figure 7.66: Tuning View

### Conversion Control Parameters

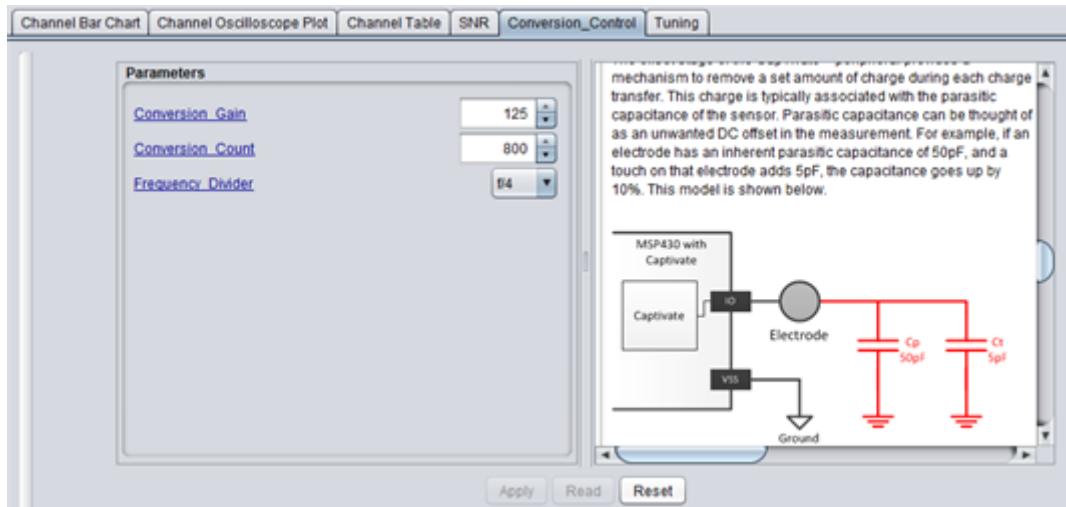


Figure 7.67: Conversion Control View

### 7.3.5 Displaying Target Data

The Design Center provides a rich set of tools to view the sensor data from the target.

- Channel Bar Chart provides a easy way to visualize the interaction between the values related to measurement counts and touch/proximity thresholds
- Oscilloscope Plot provides a way to look at historical traces of the sensor data measurements

- Channel Table provides a tabular rendering of the data present in the Channel Bar chart along with the sensor to controller mapping for each sensor element
- Zoom in/out/full zoom
  - To zoom in, click and drag down and to the right to define the zoom area
  - To zoom out, click and drag up and to the left to define the amount to zoom out
  - Click the right mouse button to do a full zoom out
- Plot Display options will display a dialog with several options for changing the plot display.

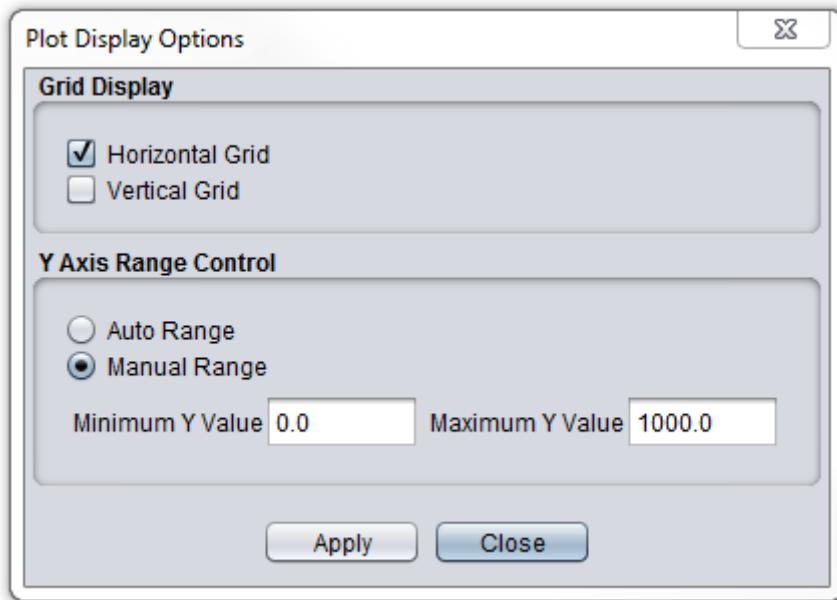


Figure 7.68: Plot Options

#### 7.3.5.1 Channel Bar Chart

Data displayed for each channel:

- Count
- Delta
- Proximity Threshold
- Touch Threshold



Figure 7.69: Channel Bar View

### 7.3.5.2 Oscilloscope Plot

Data displayed for each channel:

- Count
- Delta
- Proximity status - value of 100 indicates that the **Proximity Threshold** has been reached
- Touch status - value of 100 indicates that the **Touch Threshold** has been reached

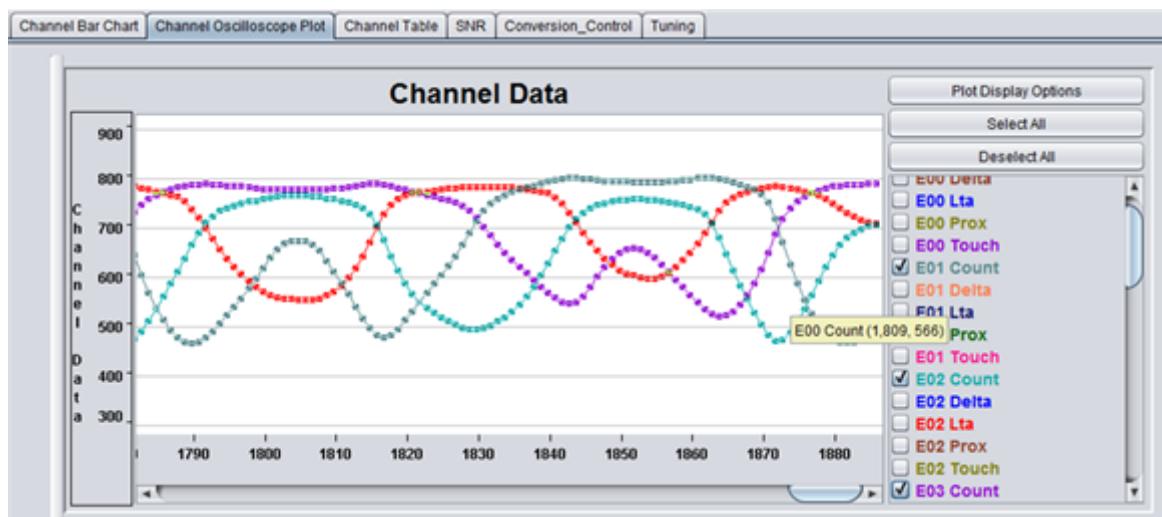


Figure 7.70: Oscilloscope View

### 7.3.5.3 Channel Table

- Data displayed for each [channel](#):
  - Connection data
    - \* [Channel](#)
    - \* Sensor name
    - \* [Element](#)
    - \* [TX](#)
    - \* [RX](#)
    - \* [Time Cycle](#)
    - \* [Controller Port](#)
    - \* [Parallel Sense Block](#)
  - Measurement data
    - \* [Count](#)
    - \* [Delta](#)
    - \* [Proximity Threshold](#)
    - \* [Touch Threshold](#)
    - \* Proximity status - yellow indicates that the [Proximity Threshold](#) has been reached
    - \* Touch status - green indicates that the [Touch Threshold](#) has been reached

Channel	Sensor	Element	TX	RX	Time C...	Port	Parallel...	LTA	Count	Delta	Prox Th...	Touch ...	Prox St...	Touch ...
12	SLD0000	E00		RX00	SLD000...	CAP0.2	B2	798	769	29	788	736	Yellow	Grey
13	SLD0000	E01		RX01	SLD000...	CAP1.2	B2	812	764	48	802	749	Yellow	Grey
14	SLD0000	E02		RX02	SLD000...	CAP2.2	B2	775	546	229	765	715	Yellow	Green
15	SLD0000	E03		RX03	SLD000...	CAP3.2	B2	795	573	222	785	733	Yellow	Green

Figure 7.71: Channel View

### 7.3.6 SNR Measurements

The CapTivate™ Design Center provides a built-in SNR measurement capability that makes it very easy to perform SNR data collection and calculations in real-time. The SNR measurement view is available on every sensor.

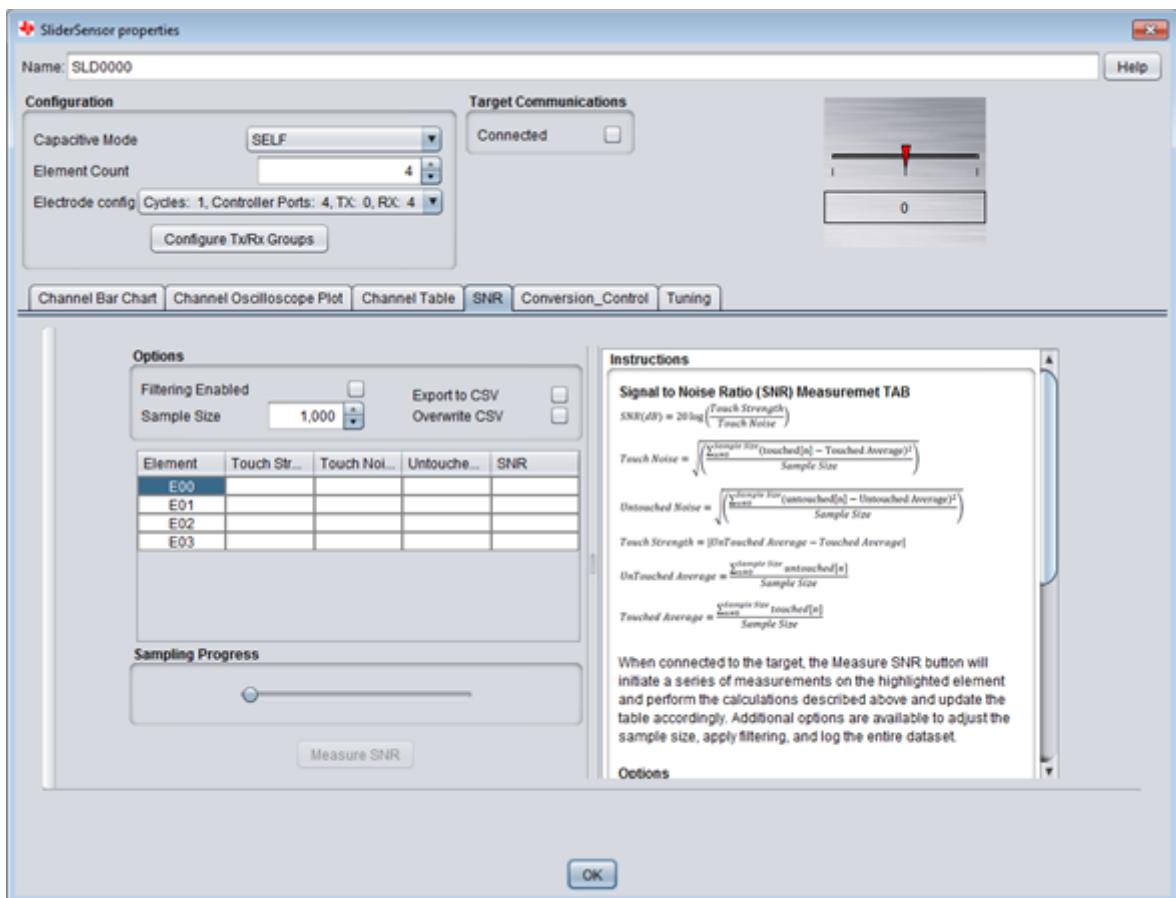


Figure 7.72: SNR View

## SNR Formula

$$SNR(dB) = 20 \log \left( \frac{\text{Touch Strength}}{\text{Touch Noise}} \right)$$

$$\text{Touch Noise} = \sqrt{\left( \frac{\sum_{n=0}^{\text{Sample Size}} (\text{touched}[n] - \text{Touched Average})^2}{\text{Sample Size}} \right)}$$

$$\text{Untouched Noise} = \sqrt{\left( \frac{\sum_{n=0}^{\text{Sample Size}} (\text{untouched}[n] - \text{Untouched Average})^2}{\text{Sample Size}} \right)}$$

$$\text{Touch Strength} = |\text{UnTouched Average} - \text{Touched Average}|$$

$$\text{UnTouched Average} = \frac{\sum_{n=0}^{\text{Sample Size}} \text{untouched}[n]}{\text{Sample Size}}$$

$$\text{Touched Average} = \frac{\sum_{n=0}^{\text{Sample Size}} \text{touched}[n]}{\text{Sample Size}}$$

Figure 7.73: SNR Formula

## SNR Options

The SNR options provide flexibility for the data collection and measurement process.

- Filtering Enabled can be used to control the data being collected by the target MCU and provide either raw or filtered data. Refer to the sensor's Tuning view to configure the filtering weight.
- Sample size determines the number of samples to be collected during the measurement process.
- Export to CSV and Overwrite CSV control how the sample data is to be stored.



Figure 7.74: SNR Options

### SNR Prompts

The SNR measurement process is automated, but does require interaction (touch) with the sensors. Only one electrode can be measured at a time. In the table under the SNR options, click to select the electrode to be measured, then click the "Measure SNR" button. The first prompt will appear. Touch and hold the target electrode and click the "OK" button. After the number of samples have been acquired, the second prompt will appear. Remove your finger from the electrode and click the "OK" button.

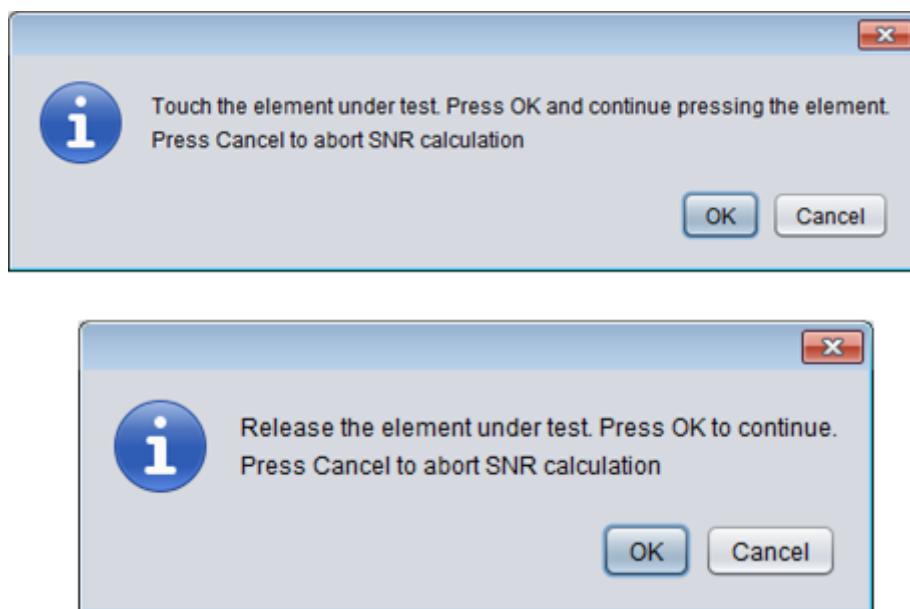


Figure 7.75: SNR Measurement Prompts

When the measurement sequence is complete, the SNR calculations are displayed in the selected electrode row.

Options				
Filtering Enabled				
Sample Size		1,000	Export to CSV	Overwrite CSV
Element	Touch Str...	Touch Noi...	Untouche...	SNR
E00	46.00	0.61	0.91	37.55
E01				
E02				
E03				
E04				
E05				
E06				
E07				

Figure 7.76: SNR Measurements

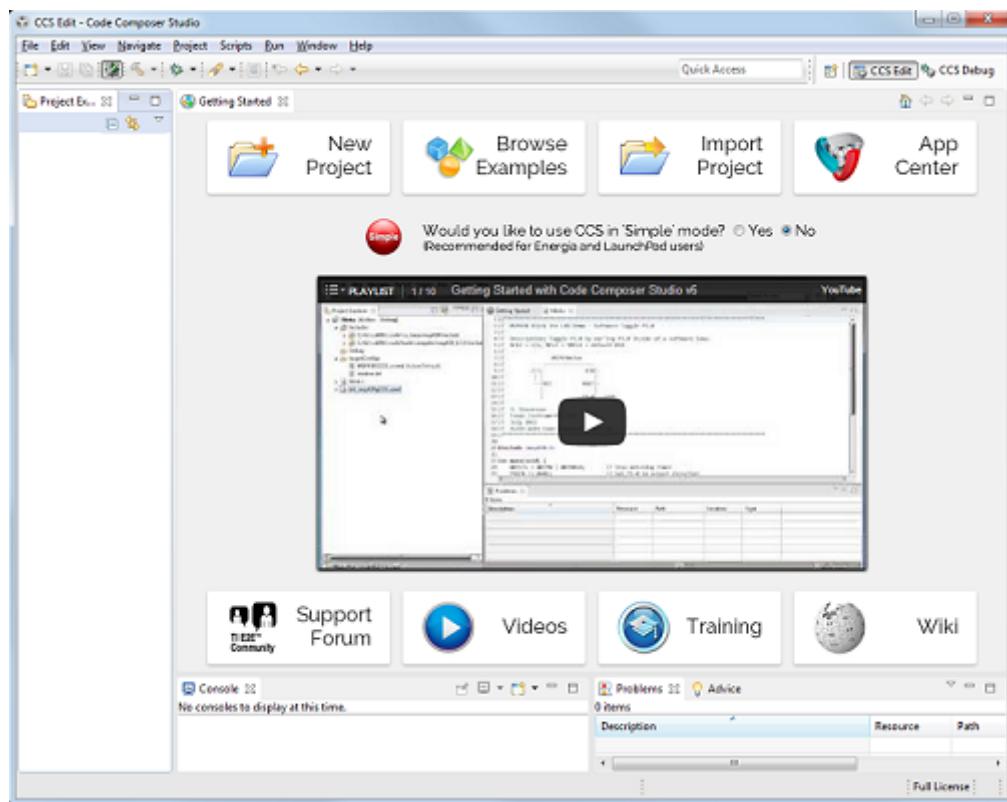
## 7.4 Loading and running generated projects

### 7.4.1 Importing/opening projects

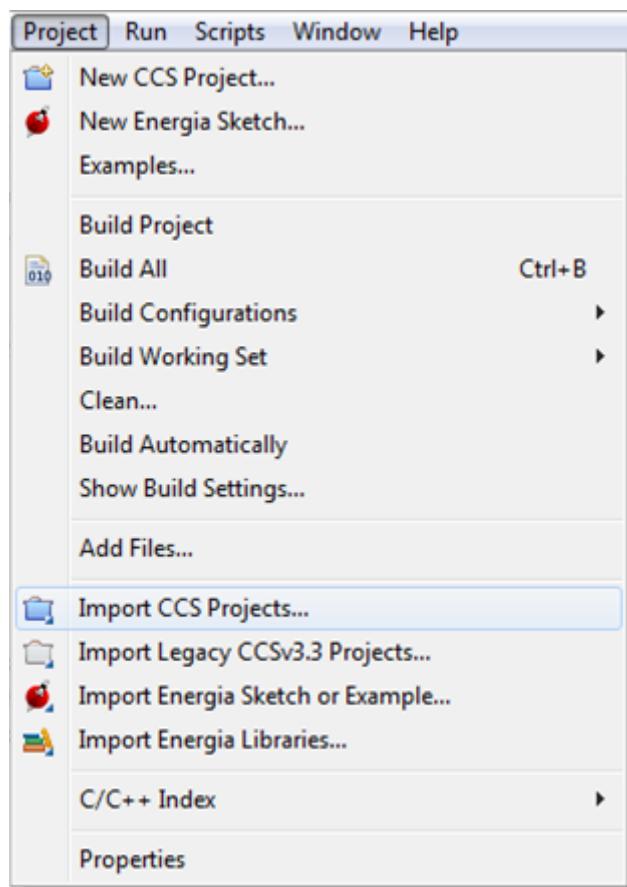
#### 7.4.1.1 Code Composer Studio

To import a CCS project generated by the Design Center, follow the instructions below.

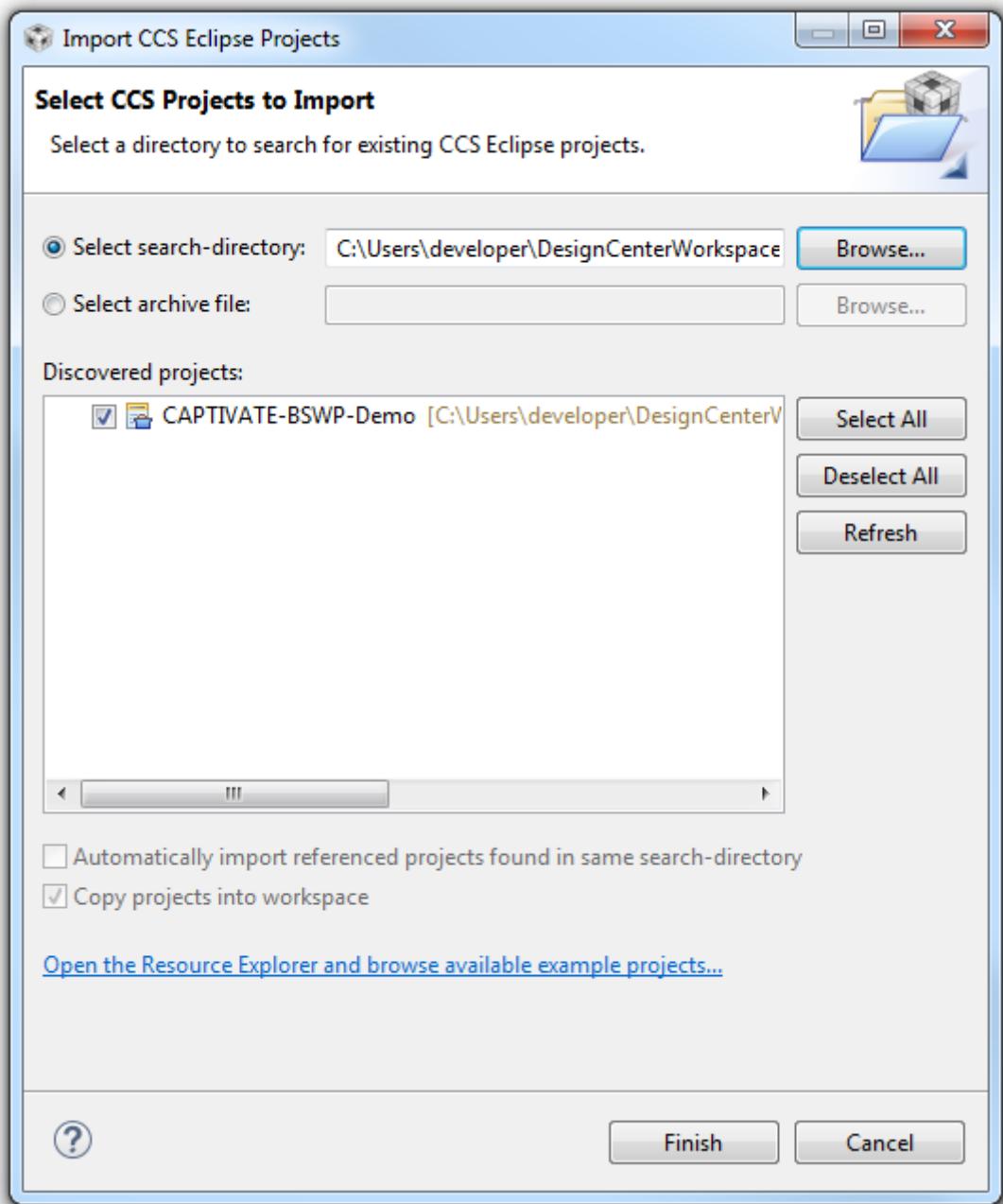
- Launch CCS



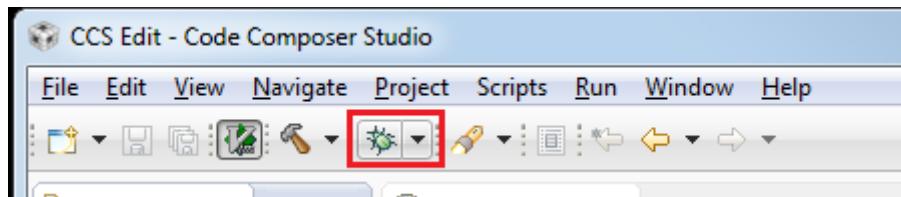
- Select "Project->Import CCS Projects"



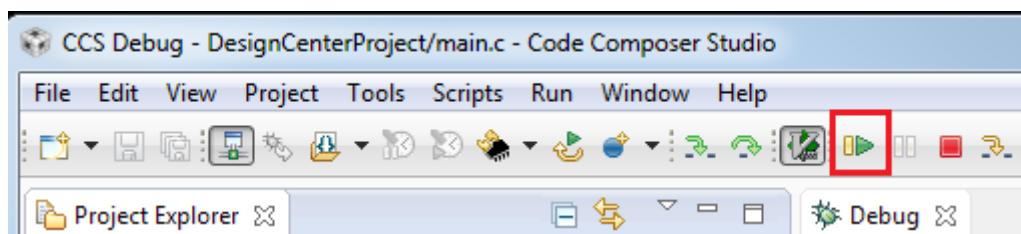
- Provide the location of the generated project source and select the project



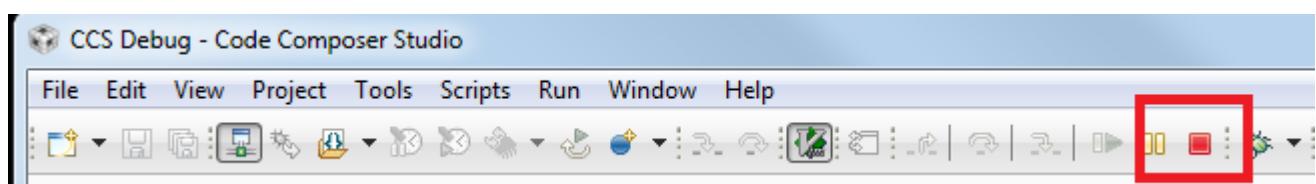
- Click "Finish"
- Compile and load the code onto the target MSP430
  - Click the "debug" button



- Click the "run" button



- Click the "stop" button

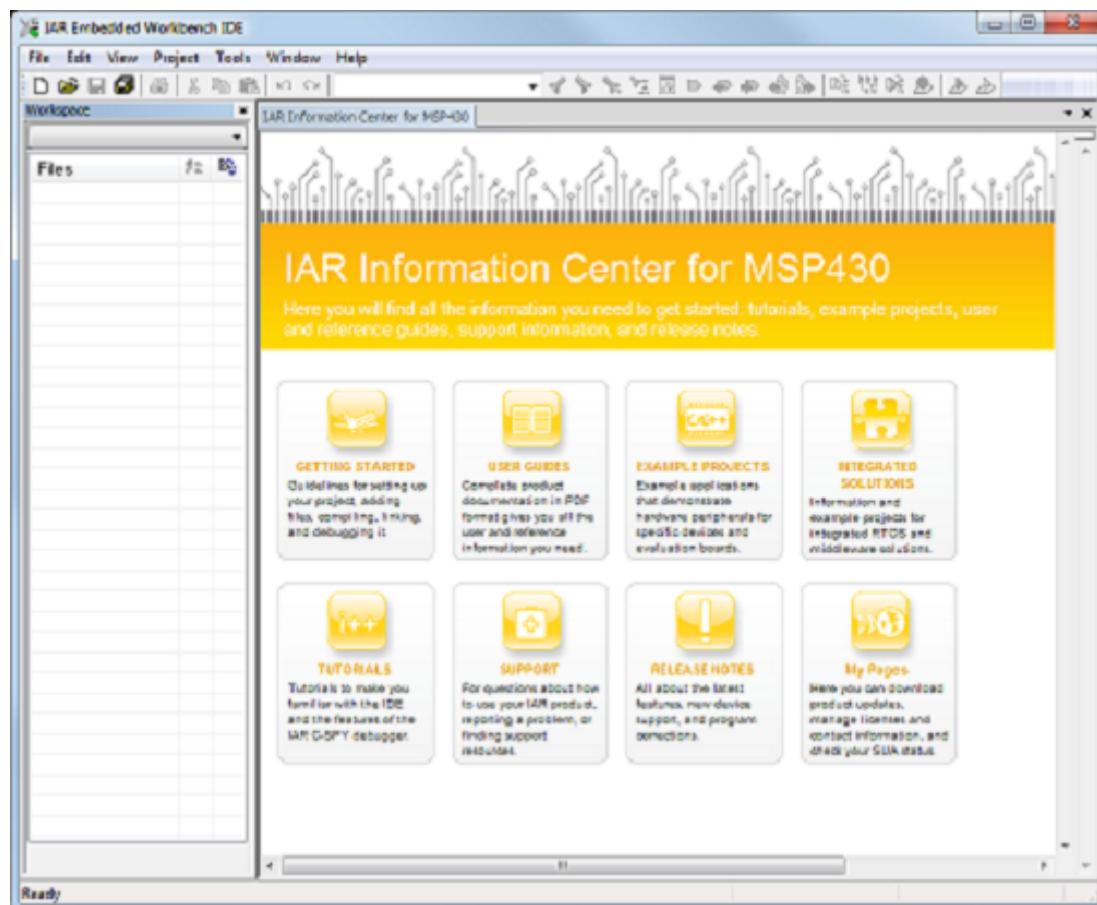


Click the "stop" button to stop the debugger.

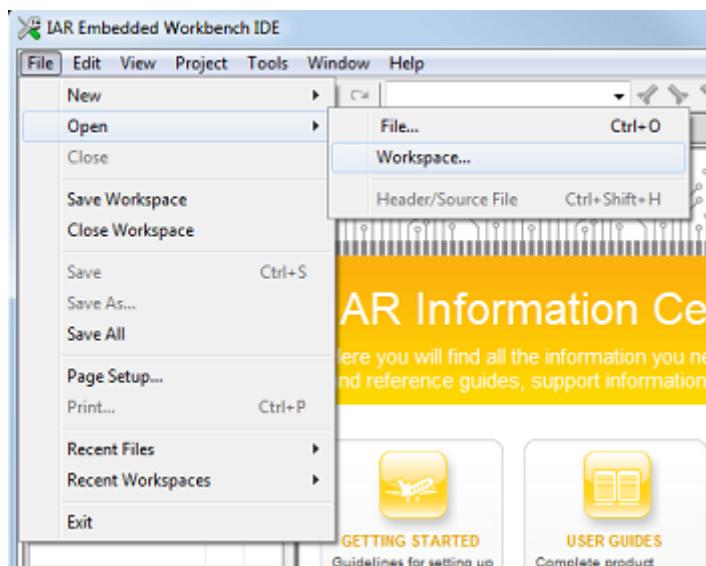
#### 7.4.1.2 IAR Embedded Workbench

To open an IAR project generated by the Design Center, follow the instructions below.

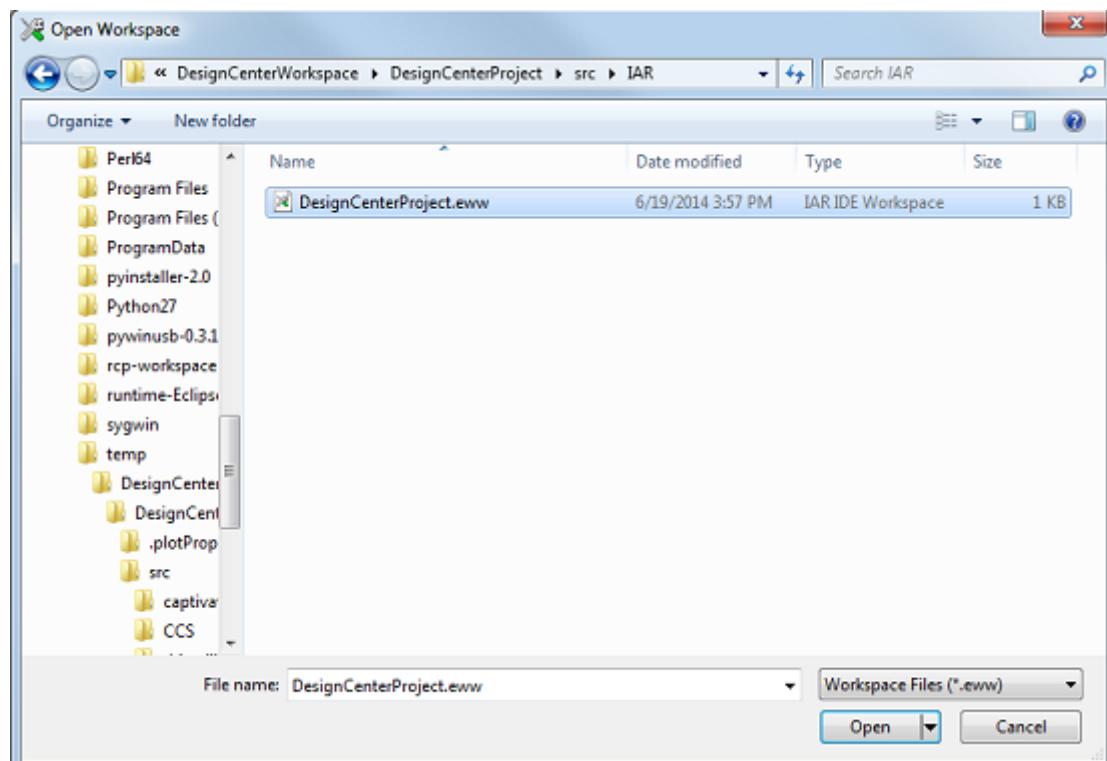
- Launch IAR Embedded Workbench



- Go to "File->Open Workspace" and navigate to the generated source directory

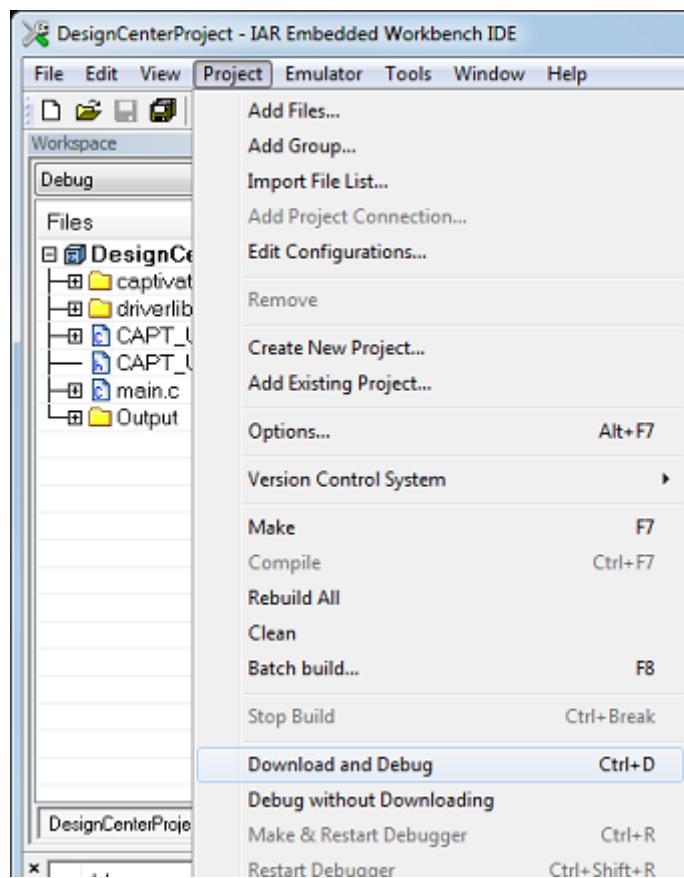


- Select the workspace file found under "IAR/<PROJECT\_NAME>.eww"

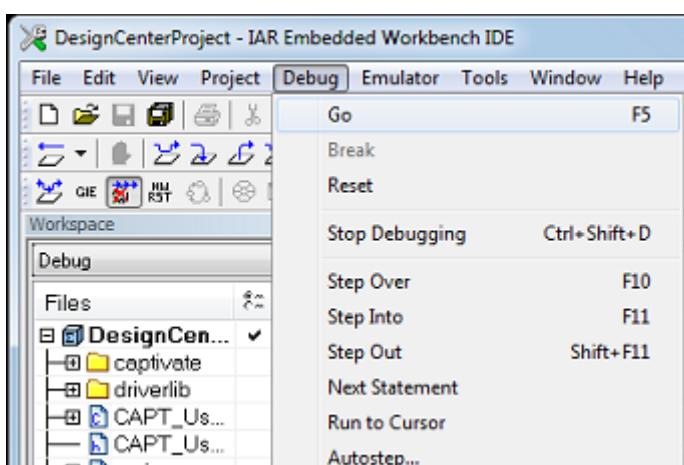


- Compile and load the code onto the target MSP430

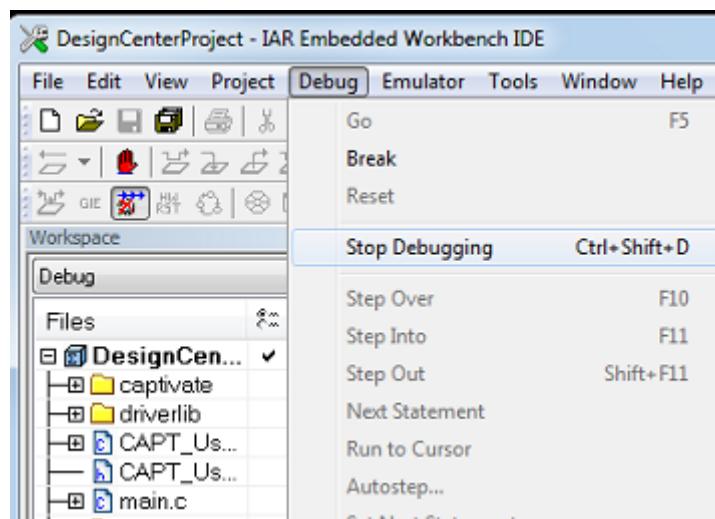
- Start debugging (Project->Download and Debug)



- Start execution (Debug->Go)

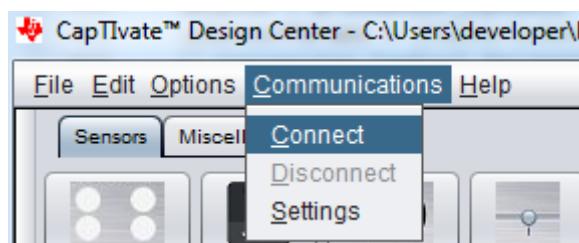


- Stop execution (Debug->Stop Debugging)



Click the "Stop Debugging" menu item to stop the debugger.

#### 7.4.2 Enable target communications in the Design Center



Once the code has been loaded onto the target, select the "Communications->Connect" menu in the Design Center to enable the HID communications.

# Chapter 8

## Device Family

CapTlivate™ is a touch sensing peripheral that is attached to a microcontroller. This chapter introduces the devices with CapTlivate™ technology on board.

The following device types are available:

- Standard devices
- Small form factor devices (Die Size / Chip Scale)

### 8.1 Standard Devices

Standard devices are listed in the table below. They are sorted by channel count and memory size in descending order.

Device Name	Packages	CapTlivate IO	CapTlivate Blocks	Typical Max CapTlivate Channels (Self/Mutual)	CPU	FRAM (kB)	SRAM (kB)	ROM (kB)	Product Folder
MSP430FR2230FN(RHB), 32-TSSOP(DA)		16	4	16/64	MSP430 CPUX	15.5	4	16	<a href="#">Link</a>
MSP430FR2230FN(RHB), 32-TSSOP(DA)		16	4	16/16	MSP430 CPUX	15.5	2	16	<a href="#">Link</a>
MSP430FR2632FN(RGE)		8	4	8/16	MSP430 CPUX	8.5	2	16	<a href="#">Link</a>
MSP430FR2532FN(RGE)		8	4	8/8	MSP430 CPUX	8.5	1	16	<a href="#">Link</a>

### 8.2 Small Form Factor (Die Size) Devices

Small form factor deviecs are listed in the table below. These are die size (chip scale) packages for space constrained designs.

Device Name	Packages	CapTivate IO	CapTivate Blocks	Typical Max CapTivate Channels (Self/Mutual)	CPU	FRAM (kB)	SRAM (kB)	ROM (kB)	Product Folder
MSP430FR2631	DSBGA(YQW)	8	4	8/16	MSP430 CPUX	15.5	4	16	Pre-release. Contact your local TI sales office.
MSP430FR2632	DSBGA(YQW)	8	4	8/16	MSP430 CPUX	8.5	2	16	Pre-release. Contact your local TI sales office.

# Chapter 9

## Software Library

### 9.1 Introduction

The CapTlivate™ Software Library is a collection of target software components designed to help shorten the development process when working with CapTlivate™ MCUs. The library is provided and supported by Texas Instruments and is delivered with the CapTlivate™ Design Center.

The library provides the following features:

1. **Hardware abstraction** of the CapTlivate peripheral features
2. **Processing** of button, slider, wheel, and proximity sensors with simple callback reporting when measurement and processing are complete
3. **User interface management** to enable a simple top level API that is easy to use
4. **Electromagnetic compatibility** features for improving noise immunity
5. **Communications** infrastructure for connecting a CapTlivate™ MCU to the CapTlivate™ Design Center during tuning or to a host processor in an application

These features provide the following main benefits:

1. Simplification of sensor configuration, measurement, processing and data communication
2. Faster application development cycles
3. Seamless integration with the CapTlivate™ Design Center development GUI
4. Reduced code footprint on devices with CapTlivate™ software in ROM

The library was designed and organized for capacitive user interface applications. However, it may also be used for other applications that require the ability to measure relative changes in capacitance.

#### 9.1.1 Using This Chapter

This chapter consists of the following main sections:

1. The [Overview](#) section introduces the programming model, organization, and architecture of the library.
2. The [Getting Started](#) section introduces how to get up and running with the starter project, as well as how to add CapTlivate™ to an existing software project.
3. The [How-To](#) section provides basic code snippets that demonstrate how to do basic things.

- 
4. The [Technical Details](#) section discusses advanced software implementation details.
  5. The [Base Module](#), [Advanced Module](#), and [Communications Module](#) sections each provide a detailed description of the respective software module for advanced users.

The [Getting Started](#) and [How-To](#) are the most helpful sections for new users that want to quickly begin developing applications.

### 9.1.2 Device and Tools Support

The CapTlivate™ Software Library can only be used with MSP devices that have CapTlivate™ technology.

#### Supported Devices

- MSP430FR26xx devices with CapTlivate™ technology
- MSP430FR25xx devices with CapTlivate™ technology

#### Supported Tool Chains

The following development tool chains are supported.

- TI Code Composer Studio
  - IDE version 6.1.0 or higher
  - MSP430 Emulator Tools 6.2.1.0 or higher
  - EABI (ELF) output format
- IAR Embedded Workbench
  - IDE version 6.3 or higher
  - MSP430 Emulator Tools 6.2.1.0 or higher

#### Programming Language

The software library is available in C. It follows C99 conventions and uses C99 primitives (uintX).

### 9.1.3 Delivery Mechanism

The CapTlivate™ Software Library and CapTlivate™ Design Center GUI have linked functionality. Features that exist in the software library are configurable via the Design Center, and data measured via the software library can be communicated back to the Design Center. Because of this, the software library and Design Center are always released together as one software download and installation. The Design Center is the sole point of access to the software library.

### 9.1.4 Change Control

Although they are delivered together, the CapTlivate Software Library has its own version tracking and change control. Every major library release comes with change control data in the Software Library API guide that describes any new features that have been added and any changes to existing functionality.

- [Access the software library change control document](#)
- [Access the software library API guide](#)

---

## 9.2 Overview

This section introduces the CapTlivate™ Software Library programming model, its organization and its architecture. It also discusses delivery of the library and version control.

### 9.2.1 Programming Model

The CapTlivate™ Software Library consists of several software *modules* and *sub-modules* that work together to provide various features and abstract complexity. The software library model will be introduced here in a "top-down" approach, starting from the highest point of abstraction and working downward to the lowest point.

#### Objects

The software library function calls operate on C structures which will be referred to in this section as objects. All of the main objects (C type definitions) for the software library are defined in the BASE module inside of the CAPT\_Type.h header file. See the [type definitions](#) section for more details.

#### Generic Capacitive Touch Application

Capacitive sensing applications involve the continual measurement and post-processing of one or more capacitive sensors. As such, an application typically has the following flow:

```
Initialize MCU
Initialize user interface
Calibrate user interface

Loop(Forever)
  If (Time to update = true)
    Then
      Update user interface
      Report user interface status
  End If
```

#### Top Level Object (User Interface Application)

The CapTlivate™ Software Library utilizes this basic application model as the framework for the top level API and top level object. The top level object in the software library is the **user interface application**, or **tCaptivateApplication**. Functions are provided for initializing a user interface, calibrating a user interface, and updating a user interface. These functions are implemented as CAPT\_initUI(), CAPT\_calibrateUI(), and CAPT\_updateUI(), respectively. An application can be created just by using these three functions. This is discussed in the [Use the Top Level API](#) section.

In order to realize the top level API, the top level object (**tCaptivateApplication**) holds information about the state of the user interface, how many sensors there are to update, where to find those sensors, and how often to update them.

#### Sensor Object

It then follows that the next object that is needed is a **sensor** object, or **tSensor**. A sensor object is an abstracted user interface control. The software library supports button group, slider, wheel, and proximity sensor types. Every sensor object contains the following information:

1. Information about which electrodes to measure and how to measure them in parallel (element objects and time cycle objects)
2. Information about how to configure the CapTlivate™ technology peripheral for measurement (conversion control parameters)
3. Information about how to interpret the data from the measurement (tuning parameters)

Various functions can operate on sensor objects directly. For example, it is possible to only update a particular sensor via a call to CAPT\_updateSensor() or CAPT\_updateSensorWithEMC(). The top level API function

---

CAPT\_updateUI() merely calls CAPT\_updateSensor() or CAPT\_updateSensorWithEMC() for each sensor in the UI application.

#### Time Cycle Object

The sensor object links to **time cycle** objects. A time cycle object is defined as **tCycle**. A time cycle is nothing more than a group of element objects that may be measured in parallel.

#### Element Object

Element objects are the lowest abstraction level, and can be thought of as the software representation of a single electrode, whether it is self or mutual capacitance. Each element contains the following types of information:

1. Information about the pin(s) the electrode is connected to
2. Any tuning parameters that are specific to the element (such as a touch threshold)
3. Any data associated with the element (such as its current sample or long term average)
4. Any status flags that are specific to the element (such as touch or proximity status).

#### Object Tree

As an example, a basic application with one sensor and 4 elements organized into two time cycles could be represented with the object tree diagram shown below.

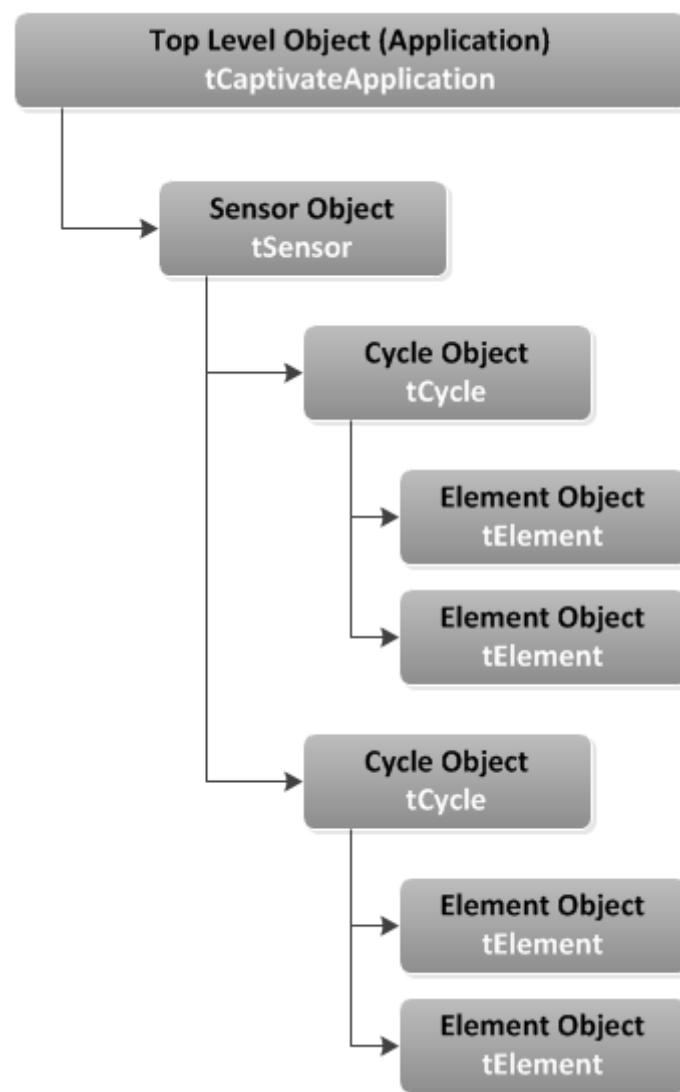


Figure 9.1: Example Application Object Tree

---

In software, this configuration would have the following structure:

### Element Definitions

```
// Sensor: keypad, Element: E00
uint16_t keypad_E00_RawCnts[CAPT_SELF_FREQ_CNT];
tCaptivateElementTuning keypad_E00_Tuning[CAPT_SELF_FREQ_CNT];
tElement keypad_E00 =
{
    .ui8RxPin = 0,
    .ui8RxBlock = 0,
    .ui8TouchThreshold = 10,
    .pRawCount = keypad_E00_RawCnts,
    .pTuning = keypad_E00_Tuning,
};

// Sensor: keypad, Element: E01
uint16_t keypad_E01_RawCnts[CAPT_SELF_FREQ_CNT];
tCaptivateElementTuning keypad_E01_Tuning[CAPT_SELF_FREQ_CNT];
tElement keypad_E01 =
{
    .ui8RxPin = 0,
    .ui8RxBlock = 1,
    .ui8TouchThreshold = 10,
    .pRawCount = keypad_E01_RawCnts,
    .pTuning = keypad_E01_Tuning,
};

// Sensor: keypad, Element: E02
uint16_t keypad_E02_RawCnts[CAPT_SELF_FREQ_CNT];
tCaptivateElementTuning keypad_E02_Tuning[CAPT_SELF_FREQ_CNT];
tElement keypad_E02 =
{
    .ui8RxPin = 1,
    .ui8RxBlock = 0,
    .ui8TouchThreshold = 10,
    .pRawCount = keypad_E02_RawCnts,
    .pTuning = keypad_E02_Tuning,
};

// Sensor: keypad, Element: E03
uint16_t keypad_E03_RawCnts[CAPT_SELF_FREQ_CNT];
tCaptivateElementTuning keypad_E03_Tuning[CAPT_SELF_FREQ_CNT];
tElement keypad_E03 =
{
    .ui8RxPin = 1,
    .ui8RxBlock = 1,
    .ui8TouchThreshold = 10,
    .pRawCount = keypad_E03_RawCnts,
    .pTuning = keypad_E03_Tuning,
};
```

Note that each element has three components:

1. An array for storing raw data after a measurement is complete (**keypad\_E0x\_RawCnts[]**)
2. An array for storing element tuning values (**keypad\_E0x\_Tuning[]**)
3. The element data object itself (**keypad\_E0x**)

The first component is the raw data array. Whenever an element is updated, the raw conversion results are populated in this array. Normally, the raw data array is an array of one value (**keypad\_E0x\_RawCnts[1]**). However, if noise immunity is enabled (EMC features), the raw data array may be larger to store conversion results from a multi-frequency conversion.

The second component is the element's tuning. Each element is calibrated with specific coarse gain, fine gain, and offset subtraction values. To understand what these parameters effect, check out the [CapTlVate peripheral](#) section of the [technology](#) chapter. Just like the raw data array, the tuning is stored in an array as well. If multi-frequency scanning is used to support noise immunity, a tuning is stored for each conversion frequency.

The final component is the element data object (**tElement**). This object stores the pin definition for the element. An electrode on CAPx.y would be mapped in this way, where 'x' is the CapTlVate™ measurement block the electrode is connected to, and 'y' is the pin on that block.

```
.ui8RxPin = y,
.ui8RxBlock = x,
```

---

In addition to the pin connection information, the element object also stores the touch threshold for this element. This specifies the level of interaction required to trigger a touch detection. Each element has its own touch threshold.

Finally, the element object is linked to the raw data and tuning arrays via pointers.

### Time Cycle Definitions

```
// Time Cycle: keypad_C00
tElement* keypad_C00_Elements[2] =
{
    &keypad_E00,
    &keypad_E01,
};

tCycle keypad_C00 =
{
    .ui8NrOfElements = 2,
    .pElements = keypad_C00_Elements,
};

// Time Cycle: keypad_C01
tElement* keypad_C01_Elements[2] =
{
    &keypad_E02,
    &keypad_E03,
};

tCycle keypad_C01 =
{
    .ui8NrOfElements = 2,
    .pElements = keypad_C01_Elements,
};
```

As discussed previously, time cycles are simply a collection of element objects that have the capability of being measured in parallel. Each time cycle is composed of two components:

1. An array of pointers to the elements in the cycle (**keypad\_C0x\_Elements[]**)
2. The cycle object itself (**keypad\_C0x**)

The array of element pointers provides the link to the element objects that belong to the cycle. The cycle object links to that array, and also defines how many elements are in the cycle.

### Sensor Definition

```
//Sensor: keypad
const tCycle* keypad_Cycles[2] =
{
    &keypad_C00,
    &keypad_C01,
};

tButtonSensorParams keypad_Params;
tSensor keypad =
{
    // Basic Properties
    .TypeOfSensor = eButtonGroup,
    .SensingMethod = eSelf,
    .DirectionOfInterest = eDOIDown,
    .pvCallback = NULL,
    .ui8NrOfCycles = 2,
    .pCycle = keypad_Cycles,
    .pSensorParams = (tGenericSensorParams*)&keypad_Params,
    // Conversion Control Parameters
    .ui16ConversionCount = 500,
    .ui16ConversionGain = 200,
    .ui8FreqDiv = 2,
    .ui8ChargeLength = 0,
    .ui8TransferLength = 0,
    .bModEnable = false,
    .ui8BiasControl = 3,
    .bCsDischarge = true,
    .bLpmControl = false,
    .ui8InputSyncControl = 0,
    .bTimerSyncControl = false,
    .bIdleState = true,
    // Tuning Parameters
    .ui16ProxThreshold = 10,
    .ui16NegativeTouchThreshold = 20,
```

---

```

.ui16ErrorThreshold = 8191,
.ui16TimeoutThreshold = 1000,
.ProxDbThreshold.DbIn = 1,
.ProxDbThreshold.DbOut = 0,
.TouchDbThreshold.DbIn = 1,
.TouchDbThreshold.DbOut = 0,
.bCountFilterEnable = true,
.ui8CntBeta = 1,
.bSensorHalt = false,
.bPTSensorHalt = true,
.bPTElementHalt = true,
.ui8LTABeta = 7,
.bReCalibrateEnable = true,
};
```

The sensor definition has three components:

1. An array of pointers to the cycles in the sensor (**keypad\_Cycles[]**)
2. The sensor type specific parameters (in this case, a button group) (**keypad\_Params**)
3. The generic sensor object itself (**keypad**)

The pointer to cycle array allows the sensor to find its child objects (cycles, and through the cycles, the elements). The sensor type specific parameters component stores parameters that are specific to a sensor type. For example, a button group, slider/wheel, and proximity sensor all have different parameter structures.

The remainder of the parameters in the sensor object provides the conversion control and tuning configuration, as set up in the CapTivate™ Design Center.

### UI Application Definition

```

// Application
tSensor* g_pCaptivateSensorArray[CAPT_SENSOR_COUNT] =
{
    &keypad,
};

tCaptivateApplication g_uiApp =
{
    .state = eUIActive,
    .pSensorList = &g_pCaptivateSensorArray[0],
    .ui8NrOfSensors = CAPT_SENSOR_COUNT,
    .ui8AppLPM = LPM0_bits,
    .bElementDataTxEnable = true,
    .bSensorDataTxEnable = true,
    .ui16ActiveModeScanPeriod = 33,
    .ui16WakeOnProxModeScanPeriod = 100,
    .ui16InactivityTimeout = 32,
    .ui8WakeInterval = 5,
};
```

The application definition has two components:

1. An array of pointers to the sensors in the application (**g\_pCaptivateSensorArray[]**)
2. The application object itself (**g\_uiApp**)

The array of pointers to sensors allows the top level API to find all of the sensors in the application through the application structure. Note that the application structure also defines the following items:

1. The low power mode to use during conversions (.ui8AppLPM)
2. Element and sensor data transmission enable/disable (.bElementDataTxEnable, .bSensorDataTxEnable)
3. The scan periods to use in active mode (.ui16ActiveModeScanPeriod)
4. A place holder for wake-on-proximity parameters (.ui16WakeOnProxModeScanPeriod, .ui16InactivityTimeout, .ui8WakeInterval)

---

## Accessing Measurement Results and Data

In general, the software library operates on the principle of refreshing data inside of objects, rather than returning results directly via a function call. For example, when a top level API call is made to a function like CAPT\_updateUI(), all sensors in the UI and each of those sensor's child elements will have their data structures refreshed. CAPT\_updateUI() does not provide any status information directly when returning to the application.

As such, it is the responsibility of the application to directly access the results of a measurement in the appropriate object data structure. A [callback function](#) may be registered with any sensor, that will be called whenever a sensor's data is refreshed.

### 9.2.2 Organization and Architecture

The CapTIVate™ Software Library is organized into three major modules by functionality: **BASE**, **ADVANCED**, and **COMM**. Each module has several sub-modules, or "layers". Some of those sub-modules are delivered as source code; others are delivered as object code.

#### BASE Module

The **BASE** module is the core of the software library. It contains the hardware abstraction layer, the touch layer, the interrupt service routine (ISR), and the type definitions for the library. The touch layer acts as a "hub," providing functions for calibrating, measuring, and processing sensors. For a detailed overview of the **BASE** module, see the [Base Module](#) section.

#### ADVANCED Module

The **ADVANCED** module provides several processing plug-ins to the **BASE** module. This includes button processing, slider and wheel processing, and EMC processing for noise immunity. It also contains the manager layer, which serves as the top level API for the library.

The basic software stack is shown below.

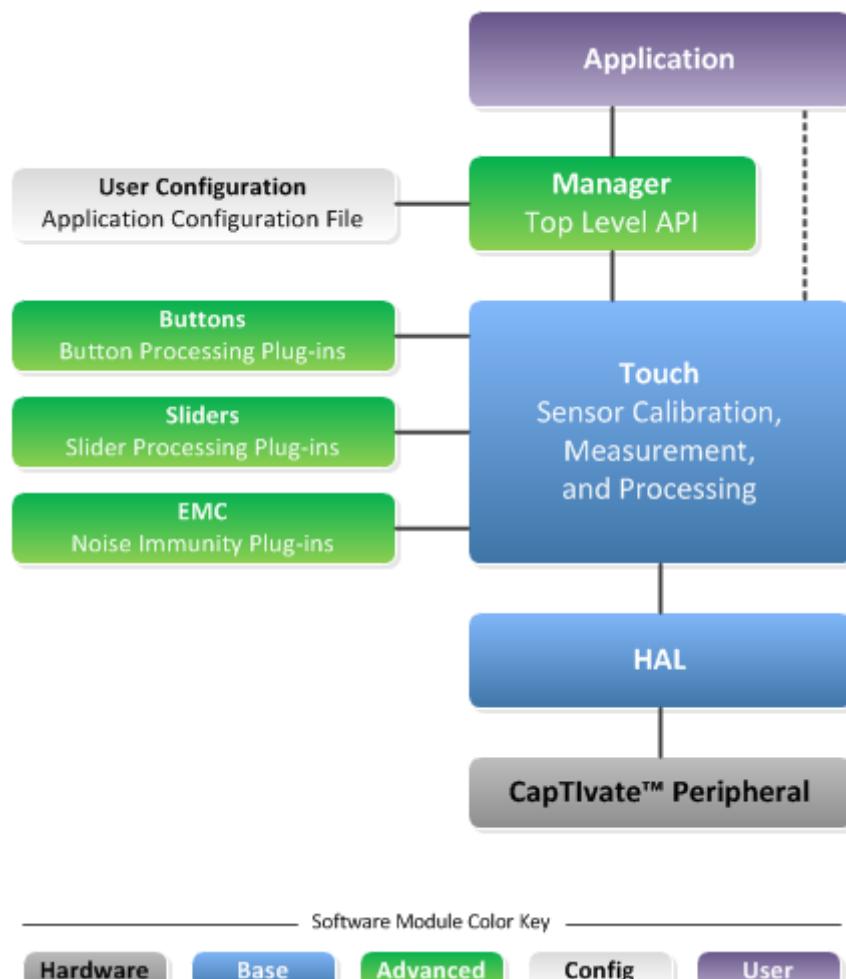


Figure 9.2: CapTivate™ Software Library Organization (Without COMMs)

As shown here, from the application space it is only necessary to call the top level API functions in the manager layer to have a functioning application. The touch layer handles pulling in the necessary plug-ins (button, slider/wheel, EMC), so there is no need to call these functions from the application space. For a detailed overview of the **ADVANCED** module, see the [Advanced Module](#) section.

### User Configuration

Every CapTivate™ application has a user configuration that defines all of the objects on the application. This includes the application, sensor, cycle, and element object definitions. This configuration is typically auto-generated by the CapTivate™ Design Center.

### COMM Module

The **COMM** module provides communication services to either a host processor or the CapTivate Design Center. It contains a top level interface layer, a protocol layer, serial drivers, and several data structures. The expanded software stack with the communications module is shown below. For a detailed overview of the **COMM** module, see the [Communications Module](#) section.

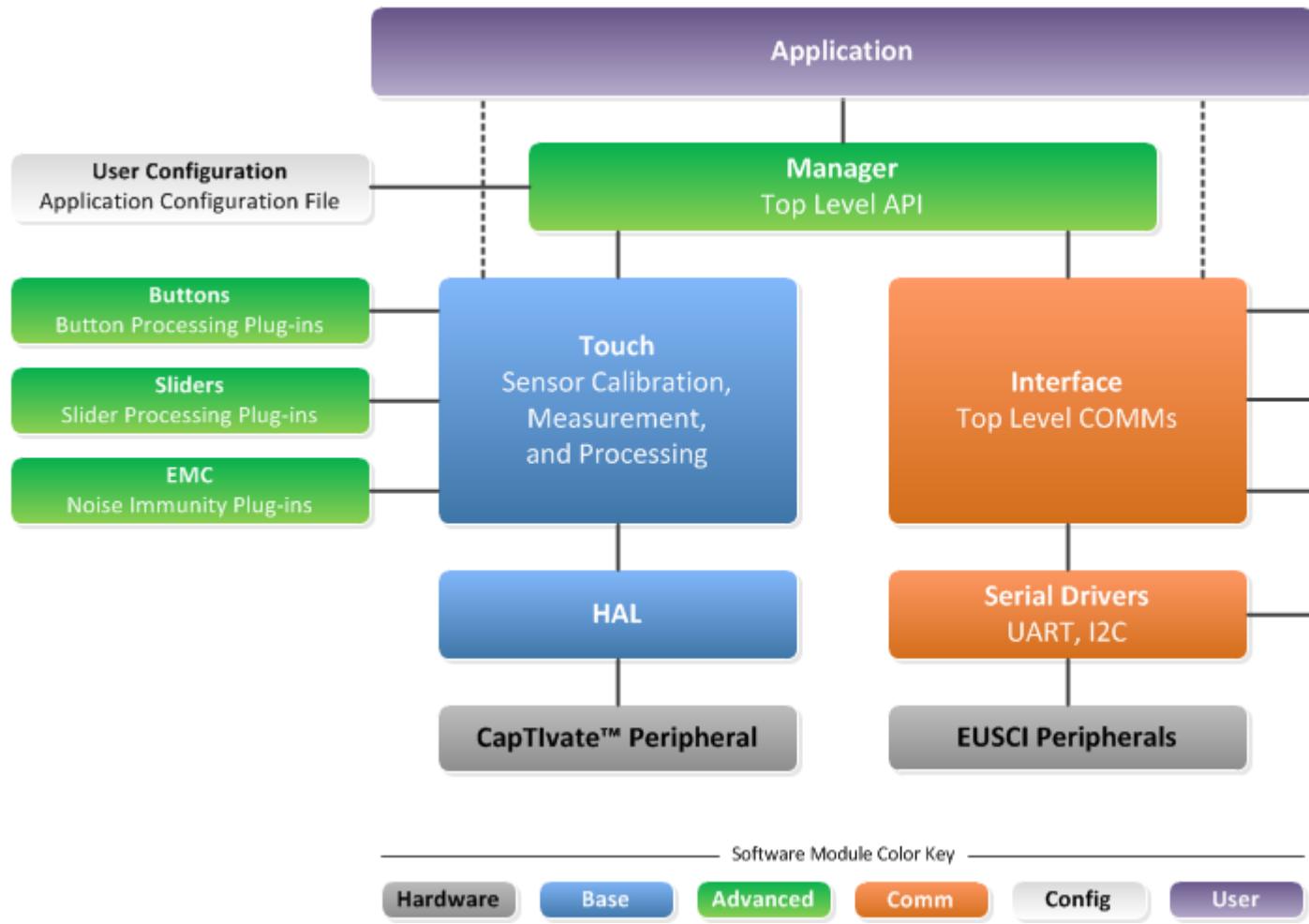


Figure 9.3: CapTivate™ Software Library Organization (With COMMs)

### Source Code Directory Structure

The library source code is organized into sub-directories by module (**BASE**, **ADVANCED**, or **COMM**). In addition to the HAL, Touch, and ISR components, the **BASE** directory contains the following files:

- 
- **captivate.lib** This is the CCS library archive. It contains all of the functions that are pre-compiled and delivered as object code for linking against the CCS compiler.
  - **captivate.r43** This is the IAR library archive. It contains all of the functions that are pre-compiled and delivered as object code for linking against the IAR compiler.
  - **lnt\_captivate.cmd** This linker command file tells the CCS linker about the CapTlve peripheral address space.
  - **rom\_captivate.h** This is the ROM function header file. It defines the ROM function calls based on the ROM function table.
  - **rom\_map\_captivate.h** This is the ROM map header file. It controls which ROM functions are valid for the version of the library that is being compiled. When making ROM calls, it is best to use the MAP\_\* convention. See the [ROM function overview](#) for details on how to call ROM functions.

## 9.3 Getting Started

### 9.3.1 Starting from Scratch with the Starter Project

The recommended way to get started with a CapTlve™ software library project is to generate a new starter software project with the CapTlve™ Design Center. To learn how to do this, step through the [new sensor project design workshop](#).

This section will focus on the features of the starter project itself. The software library starter project is a ready-to-go application that includes the following software components:

1. The CapTlve™ Software Library
2. The MSP Peripheral Driver Library (DriverLib), delivered as pre-compiled object code
3. A board support package, configured for the MSP-CAPT-FR2633 EVM
4. An example application with wake-on-proximity support built-in
5. An example main.c

The starter project contains everything that is needed to bring up the MCU, calibrate and run a capacitive sensing application per the specified user configuration, and communicate measurement data. It serves as a known-good starting point for new development and tuning. Once an application is tuned, features can be added and removed from the starter application to build toward a final production application.

#### Directory Structure

The directory structure of the starter project is shown below in CCS:

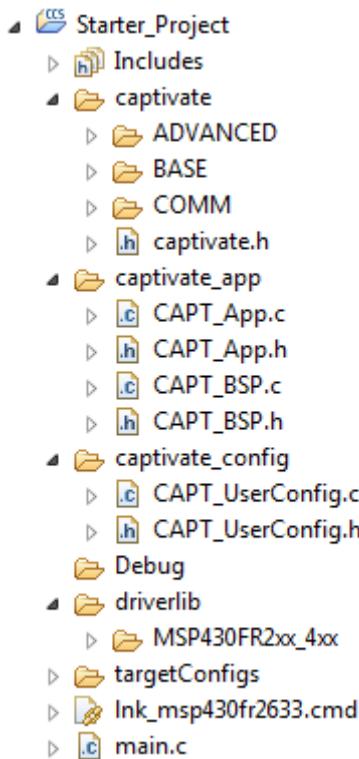


Figure 9.4: Starter Project Files

The *captive* directory contains the CapTIvate™ Software Library. This directory will likely not need to be edited during development.

The *captive\_app* directory contains the board support package (CAPT\_BSP) and the example wake-on-proximity application (CAPT\_APP). This directory contains starter files that should be modified to suit the needs of each individual application.

The *captive\_config* directory contains the automatically generated user configuration files that describe the capacitive sensing application as specified in the CapTIvate™ Design Center. The CAPT\_UserConfig.c and CAPT\_UserConfig.h files in this directory **should never be edited manually**, as changes made to these files are **overwritten if the Design Center is utilized to update the configuration**. For this reason, it is best to modify all configuration parameters from inside the Design Center.

The *driverlib* directory contains the MSP430 peripheral driver library. To lower compilation times, it is delivered as a pre-compiled library archive. It is possible to replace this DriverLib directory with the standard, open-source driver library, if desired.

## Main

The starter application main.c is shown below:

```

#include <msp430.h>                                // Generic MSP430 Device Include
#include "driverlib.h"                               // MSPWare Driver Library
#include "captivate.h"                             // CapTIvate Touch Software Library
#include "CAPT_App.h"                               // CapTIvate Application Code
#include "CAPT_BSP.h"                               // CapTIvate EVM Board Support Package

void main(void)
{
    //
    // Initialize the MCU
    // BSP_ConfigureMCU() sets up the device IO and clocking
    // The global interrupt enable is set to allow peripherals
    // to wake the MCU.
    //
    WDT_A_hold(WDT_A_BASE);
}

```

---

```

BSP_configureMCU();
__bis_SR_register(GIE);

//
// Start the CapTIVate application
//
CAPT_appStart();

//
// Background Loop
//
while(1)
{
    //
    // Run the captivate application handler.
    // Set LED1 while the app handler is running,
    // and set LED2 if proximity is detected
    // on any sensor.
    //
    LED1_ON;
    if(CAPT_appHandler() == true)
        LED2_ON;
    else
        LED2_OFF;
    LED1_OFF;

    //
    // This is a great place to add in any
    // background application code.
    //
    __no_operation();

    //
    // End of background loop iteration
    // Go to sleep if there is nothing left to do
    //
    CAPT_appSleep();
}

} // End background loop
} // End main()

```

The **main()** routine disables the watchdog timer, initializes the MCU via the board support package, and starts the capacitive sensing application. It then spends the rest of its time inside the application background loop. LED1 is toggled whenever the application handler is called, and LED2 is set whenever the **CAPT\_appHandler()** function returns true, indicating that any sensor has a proximity detection. After the app handler runs, a call to **CAPT\_appSleep()** will put the MCU to sleep if there are no pending flags that need to be serviced. The CapTIVate™ conversion timer interrupt will wake the application each time the user interface needs to be refreshed.

#### Board Support Package (CAPT\_BSP)

The board support package configures the MCU for operation with the following parameters:

- Watchdog timer is disabled
- 8 MHz DCO Frequency
- 8 MHz MCLK for zero wait state FRAM execution, sourced from the DCO
- 2 MHz SMCLK, sourced from the DCO
- 32 kHz ACLK, sourced from an external crystal (if connected) or the internal REFO
- USCI\_A0 and USCI\_B0 are muxed to pins to allow for communication

The port muxing is configured for the CAPTIVATE-FR2633 processor module. **The board support package should be ported to the platform and device used in each application!**

#### Example Application (CAPT\_App)

The example application demonstrates how to enable a generic capacitive sensing application with or without wake-on-proximity. It includes three functions:

- CAPT\_appStart()
- CAPT\_appHandler()
- CAPT\_appSleep()

#### CAPT\_appStart()

This function provides an example of the functions that need to be called to configure the CapTlivate™ Software Library for operation. It handles the following tasks:

1. Initializing and calibrating the capacitive sensing UI via the CapTlivate™ Software Library top level API calls
2. Configuring and starting the CapTlivate™ timer for periodic interrupts

As discussed in the [guide for using the top level API](#), it is necessary to call **CAPT\_initUI()** and **CAPT\_calibrateUI()** when starting a CapTlivate™ software library application. These functions configure the CapTlivate™ peripheral and calibrate all of the elements in the UI. If noise immunity (EMC) functionality is going to be used, it is also necessary to load an EMC configuration structure via a call to **CAPT\_loadEMCConfig()**. Note that in the actual example CAPT\_appStart() function, the **CAPT\_loadEMCConfig()** function is a compile-time include.

```
CAPT_initUI(&g_uiApp);
CAPT_loadEMCConfig(&g_EMConfig); // (Only needed if EMC features are enabled!)
CAPT_calibrateUI(&g_uiApp);
```

The integrated CapTlivate™ conversion timer is a periodic timer that can be used to generate an interrupt or directly trigger a conversion at a specified interval. The timer is configured via HAL function calls, as shown below. Note that these HAL functions are available in ROM on devices with CapTlivate™ software in ROM- hence the **MAP\_** calls. For more information on ROM software, see the [ROM](#) section.

```
MAP_CAPT_stopTimer();
MAP_CAPT_clearTimer();
MAP_CAPT_selectTimerSource(CAPT_TIMER_SRC_ACLK);
MAP_CAPT_selectTimerSourceDivider(CAPT_TIMER_CLKDIV__1);
MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ui16ActiveModeScanPeriod));
MAP_CAPT_startTimer();
MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);
```

These setup functions configure the timer to be sourced from ACLK (32 kHz in this starter project). An input divider of 1 is selected, and the compare register is set to the active mode scan period, converted to cycles. The macro **CAPT\_MS\_TO\_CYCLES** approximates the number of 32 kHz clock cycles needed to produce the desired scan rate by multiplying the saved value (in milliseconds) by 32 (via a 5x bit shift). The timer interrupt is enabled to start the application. When the timer interrupt is asserted by the timer, the CapTlivate™ peripheral interrupt handler will run. The interrupt handler is in the BASE layer of the CapTlivate™ library, and is available in source code for transparency.

```
#pragma vector=CAPTIVATE_VECTOR
__interrupt void CAPT_ISR(void)
{
    switch(__even_in_range(CAPT_getInterruptVector(), CAPT_IV_MAX_COUNT_ERROR))
    {
        // End of Conversion Interrupt
        case CAPT_IV_END_OF_CONVERSION:
            g_bEndOfConversionFlag = true;
            break;

        // Detection Interrupt
        case CAPT_IV_DETECTION:
            g_bDetectionFlag = true;
            break;

        // Timer Interrupt
        case CAPT_IV_TIMER:
            g_bConvTimerFlag = true;
            break;

        // Conversion Counter Interrupt
        case CAPT_IV_CONVERSION_COUNTER:
            g_bConvCounterFlag = true;
            break;
    }
}
```

---

```

        break;

    // Max Count Error Interrupt
    case CAPT_IV_MAX_COUNT_ERROR:
        g_bMaxCountErrorFlag = true;
        break;
    }
    __bic_SR_register_on_exit(LPM3_bits);
}

```

The interrupt handler will set the appropriate flag, and it always exits active. Therefore, the application merely needs to monitor the **g\_bConvTimerFlag** boolean value to know when the timer has tripped, meaning that it is time to update the user interface.

#### CAPT\_appHandler()

The application handler function must be periodically called from the application background loop, as shown in **main.c**. At first glance, the function appears quite complex- but really, all the application handler does is manage when the UI needs to be updated (in active mode), as well as manage the transitions between active mode and wake-on-proximity mode.

The application handler makes use of several convenience variables in the CapTlivate™ top level application object:

- The active mode scan period
- The wake on proximity mode scan period
- The inactivity time-out
- The wakeup interval

It may be configured at compile time for two different modes of operation:

1. Active mode only (No wake-on-proximity state handling). This is the reduced memory footprint option.
2. Active mode with wake-on-proximity management. This option requires more memory to handle the wake-on-proximity management.

The compile time mode is determined by settings in the user configuration header file (CAPT\_UserConfig.h in the *CAPT\_config* directory):

#### Active Mode Only Configuration

```
#define CAPT_WAKEONPROX_ENABLE (false)
#define CAPT_WAKEONPROX_SENSOR (none)
```

#### Active Mode with Wake-on-Proximity Configuration

```
#define CAPT_WAKEONPROX_ENABLE (true)
#define CAPT_WAKEONPROX_SENSOR (selected sensor)
```

These options can and should be configured through the CapTlivate™ Design Center. See the [Compile Time Options](#) section for details.

Active mode is characterized by the following behavior:

- The CapTlivate™ conversion timer is used in interrupt mode to wake the CPU at a specified interval
- When the **g\_bConvTimerFlag** is asserted, **CAPT\_updateUI()** is called to refresh all sensors in the UI under CPU control.

Wake-on-proximity mode is characterized by the following behavior:

- The CapTlve™ conversion timer is used in timer-triggered conversion mode to automatically start a conversion of a single time cycle at a specified interval without any CPU intervention
- When any element in the time cycle selected for wake-on-proximity has a proximity threshold crossing or negative touch threshold crossing, the detection flag (**g\_bDetectionFlag**) is asserted and the application switches to active mode
- If the conversion counter flag (**g\_bConvCounterFlag**) is asserted, the application switches to active mode

#### CAPT\_appHandler() Compiled for Active Mode Only

The diagram below describes the behaviour of the application handler when compiled for active mode support only.

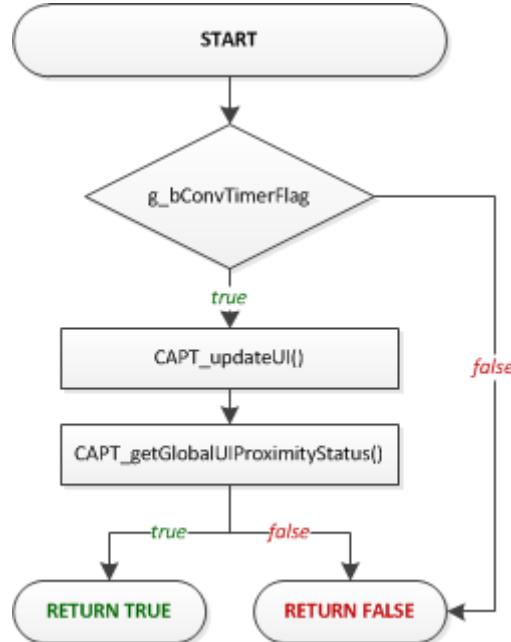


Figure 9.5: CAPT\_appHandler() with Active Mode Only Support

When compiled for active mode only, the function is pre-processed down to this basic set of functionality:

```

bool CAPT_appHandler(void)
{
    static bool bActivity = false;

    if (g_bConvTimerFlag == true)
    {
        //
        // Clear the conversion timer flag,
        // and update the UI
        //
        g_bConvTimerFlag = false;
        CAPT_updateUI(&g_uiApp);
        bActivity = CAPT_getGlobalUIProximityStatus(&g_uiApp);
    }

    return bActivity;
}
  
```

Since the CapTlve™ conversion timer was already configured by **CAPT\_appStart()**, all that is needed is to test the **g\_bConvTimerFlag**. If it is set, then the application handler clears it, updates the UI via **CAPT\_updateUI()**, and checks to see if any elements in the UI have a proximity detection. The function returns true if any element was in proximity, else false.

If communications are enabled, the function also checks for incoming packets as shown below:

```

bool CAPT_appHandler(void)
{
  
```

---

```

static bool bActivity = false;

if (g_bConvTimerFlag == true)
{
    //
    // Clear the conversion timer flag,
    // and update the UI
    //
    g_bConvTimerFlag = false;
    CAPT_updateUI(&g_uiApp);
    bActivity = CAPT_getGlobalUIProximityStatus(&g_uiApp);
}

//
// If communications are enabled, check for any incoming packets.
//
CAPT_checkForInboundPacket();

//
// Check to see if the packet requested a re-calibration.
// If wake-on-prox is enabled and the current application state
// is wake-on-prox, disable the wake-on-prox feature during the calibration
// and re-enable it after the calibration.
//
if (CAPT_checkForRecalibrationRequest() == true)
{
    CAPT_calibrateUI(&g_uiApp);
}

return bActivity;
}

```

A call is made to **CAPT\_checkForInboundPacket()** to see if any packets have been received that need processing. Some parameter packets (such as a sensor's conversion count) require that a re-calibration take place. The **CAPT\_checkForRecalibrationRequest()** checks to see if a re-calibration request is pending.

Note that if **g\_bConvTimerFlag** is false, the function is essentially non-blocking. This allows the background loop in **main()** to service other application tasks, while just periodically calling **CAPT\_appHandler()** to see if it is time to do something.

#### **CAPT\_appHandler() Compiled for Active Mode with Wake-on-Proximity Mode**

When compiled with wake-on-proximity mode enabled, the application handler manages transitions between active mode and wake-on-proximity mode. The application starts in active mode, and follows the flow shown below:

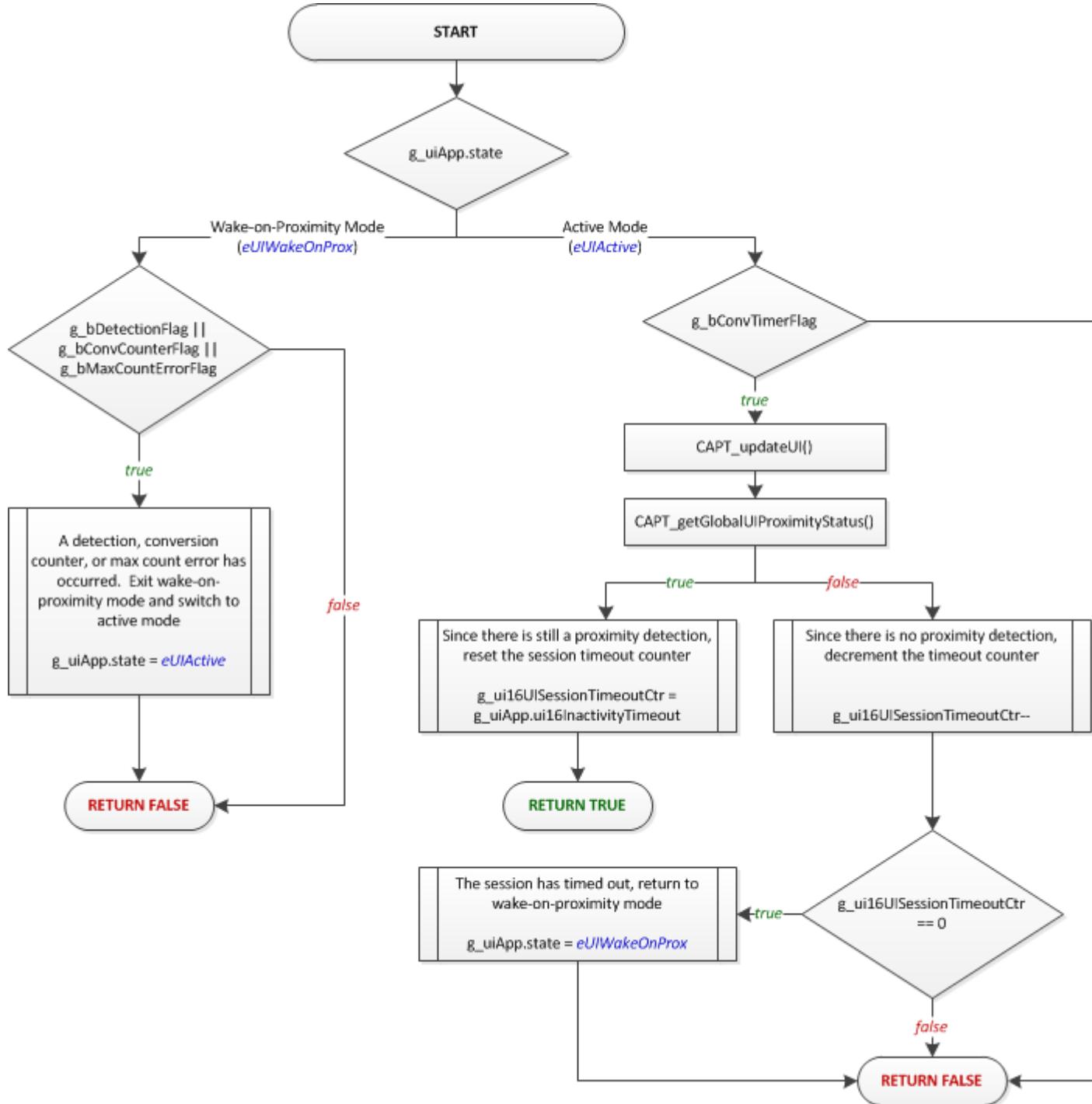


Figure 9.6: CAPT\_appHandler() with Wake-on-Proximity Support

---

The wake-on-proximity feature allows for one time cycle to be measured and processed autonomously with no CPU interaction until a detection, counter interrupt, or error condition occurs. This is useful for applications that have a power key or a proximity sensor that is used to wake up the rest of the system, since that sensor may be scanned with no CPU overhead. When a detection does occur, the handler switches operation to active mode and the entire user interface is scanned under CPU control.

The application remains in active mode under CPU control as long as at least one element is in proximity detect. Once all elements are clear of proximity, the session time-out counter begins counting down. This counter will keep the system in active mode for the specified number of samples before returning into wake-on-proximity mode. The timeout is specified in the application object via the **ui16InactivityTimeout** parameter.

In addition to waking on a detection, it is also possible to periodically wake up into active mode after a certain number of conversions have taken place in wake-on-proximity mode. This is useful to ensure that all the other sensors in the system have current long term averages (LTAs) to account for environmental drift. The conversion counter interrupt may be used to specify a wakeup period.

These wakeup sources are discussed further in the [wake-on-proximity](#) description.

```
bool CAPT_appHandler(void)
{
    static uint16_t g_ui16UISessionTimeoutCtr = 1;
    static bool bActivity = false;

    switch (g_uiApp.state)
    {
        case eUIActive:
            if (g_bConvTimerFlag == true)
            {
                //
                // Clear the conversion timer flag,
                // and update the UI
                //
                g_bConvTimerFlag = false;
                CAPT_updateUI(&g_uiApp);
                bActivity = CAPT_getGlobalUIProximityStatus(&g_uiApp);

                //
                // If autonomous mode is enabled, check to
                // see if autonomous mode should be entered.
                //
                if (bActivity == true)
                {
                    //
                    // If there is still a prox detection,
                    // reset the session timeout counter.
                    //
                    g_ui16UISessionTimeoutCtr = g_uiApp.ui16InactivityTimeout;
                }
                else if (--g_ui16UISessionTimeoutCtr == 0)
                {
                    //
                    // If the session has timed out,
                    // enter autonomous mode
                    //
                    g_uiApp.state = eUIWakeOnProx;
                    bActivity = false;

                    //
                    // Set the timer period for wake on touch interval
                    //
                    MAP_CAPT_disableISR(CAPT_TIMER_INTERRUPT);
                    MAP_CAPT_stopTimer();
                    MAP_CAPT_clearTimer();
                    MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ui16WakeOnProxModeScanPeriod));
                }
            };
            MAP_CAPT_startTimer();
            g_bConvTimerFlag = false;
            CAPT_startWakeOnProxMode(
                &CAPT_WAKEONPROX_SENSOR,
                0,
                g_uiApp.ui8WakeupInterval
            );
        }
        break;

        case eUIWakeOnProx:
            if (g_bDetectionFlag || g_bConvCounterFlag || g_bMaxCountErrorFlag)
            {
                //
                // If a detection, conversion counter, or max count error flag was set,
```

---

```

        // stop autonomous mode and reload an active session
        //
        CAPT_stopWakeOnProxMode(&CAPT_WAKEONPROX_SENSOR, 0);
        g_bDetectionFlag = false;
        g_bConvCounterFlag = false;
        g_bMaxCountErrorFlag = false;
        g_uiApp.state = eUIActive;
        g_ui16UISessionTimeoutCtr = g_uiApp.ul16InactivityTimeout;

        //
        // Set the timer period for normal scan interval
        //
        MAP_CAPT_disableISR(CAPT_TIMER_INTERRUPT);
        MAP_CAPT_stopTimer();
        MAP_CAPT_clearTimer();
        MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ul16ActiveModeScanPeriod));
        MAP_CAPT_startTimer();
        CAPT_clearIFG(CAPT_TIMER_INTERRUPT);
        MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);
    }

    break;
}

return bActivity;
}

```

Note that it is possible to set a different scan period for wake-on-proximity mode than the period used in active mode. This allows for slow scanning while waiting for proximity, and faster scanning when a user is detected. Scanning less often reduces the overall power consumption.

In addition, the clock source and low power mode used during wake-on-proximity mode may be set to the VLO and LPM4, respectively, for applications that require low power but cannot use a crystal. To enable this combination, define **CAPT\_WOP\_VLO\_LPM4** at the beginning of CAPT\_App.c. When this is defined, the application's low power mode will be set to LPM4 (ACLK off), and the VLO will be selected as the input clock source to the CapTlivate timer.

For more details on designing for low power, see the [low power design guide](#).

#### **CAPT\_appSleep()**

The application sleep function is a safety wrapper that ensures no flags are pending when the application transitions into a low power mode. Safety is achieved by disabling interrupts, testing the flags, and then entering a low power mode while simultaneously re-enabling interrupts. This sequence protects the application from any software race conditions that might occur, such as a flag being set after the flag was tested but before the device entered low power mode.

```

__bic_SR_register(GIE);
if (!(g_bConvTimerFlag || g_bDetectionFlag || g_bConvCounterFlag || g_bMaxCountErrorFlag))
{
    __bis_SR_register(g_uiApp.ui8AppLPM | GIE);
}
else
{
    __bis_SR_register(GIE);
}

```

#### **Moving Forward**

That's it! The example application in *captivate\_app* is meant to be just that- an example. Feel free to modify it to suit the needs of another application. For details on how to use the CapTlivate™ Software Library top level API, see [Using the Top Level API](#).

#### **9.3.2 Adding CapTlivate™ to an Existing Project**

If you have an existing application using an MSP MCU and would like to add CapTlivate™ capacitive touch sensing capability to that project, this guide will discuss the important steps to take to make the integration as seamless as possible.

This guide also provides details about how to set up a new project from scratch, if the CapTlivate™ starter project is not being used.

---

## Similar Devices

The CapTlivate™ MSP430FR26xx and MSP430FR25xx MCUs have a similar platform architecture as some other MSP430 FRAM devices. As such, porting an existing application from another FRAM device to an MSP430FR26xx or MSP430FR25xx MCU does not require significant effort. MCUs that are very compatible include the MSP430FR4133, MSP430FR2433, and MSP430FR2033.

## Porting Approach

The best way to begin CapTlivate™ software library development, regardless of whether the capacitive sensing functionality will be integrated with another project or not, is to generate a starter project with the CapTlivate™ Design Center. This starter project may then be used by itself to quickly bring up a capacitive sensing application and experiment with tuning. Once that process is complete, all that is needed is to bring over the necessary CapTlivate™ software components from the starter project into the existing software project.

The [previous section](#) discusses how to generate a starter project. The [new sensor project design workshop](#) also provides a step-by-step overview of the process. Once you have a starter project, you can extract the CapTlivate™ software library components from the starter project and integrate them into the existing project.

### Bringing over CapTlivate™ Software Components from a Starter Project to an Existing Project

As an example, this section will discuss bringing over the CapTlivate™ software components from a starter project to an existing software project. This example will be discussed in the context of TI's Code Composer Studio (CCS) IDE. The same principles apply to an IAR Embedded Workbench project. In this example, the existing software project will be an empty CCS project, as shown below:

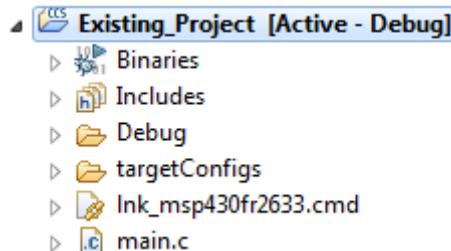


Figure 9.7: Empty Application

#### main.c

```
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    while(1)
    {
        __no_operation();
    }
}
```

To bring over CapTlivate™, the following steps are required:

1. Copy over the *captivate*, *captivate\_config*, and *driverlib* directories and all of their content from the starter project to the existing application
2. Configure the existing project's compiler settings for CapTlivate™
3. Configure the existing project's linker settings for CapTlivate™
4. Add top level API calls to the main application to initialize, calibrate, and begin updating the newly added capacitive sensing interface

---

5. [Optional] Optimize the clock system configuration for CapTlve™

**Step 1: Copying the Needed CapTlve™ Files**

- 1a. Copy the *captivate*, *captivate\_config*, and *driverlib* directories and all of their content from the starter project to the existing application
- 1b. Copy the *driverlib* directory and all of its contents from the starter project to the existing application

Below are the directories that need to be copied from the starter project:

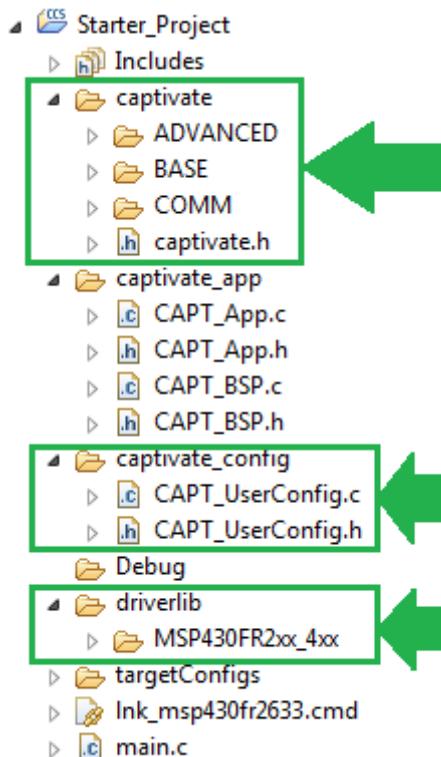


Figure 9.8: Starter Application Files To Copy

**Step 2: Configuring the Existing Project's Compiler Settings for CapTlve™**

- 2a. Add the *captivate*, *captivate/BASE*, *captivate/ADVANCED*, *captivate/COMM*, and *captivate\_config* project directories to the existing project's include search path, as shown below. Also add the DriverLib directory *driverlib/MSP430FR2x\_4xx*

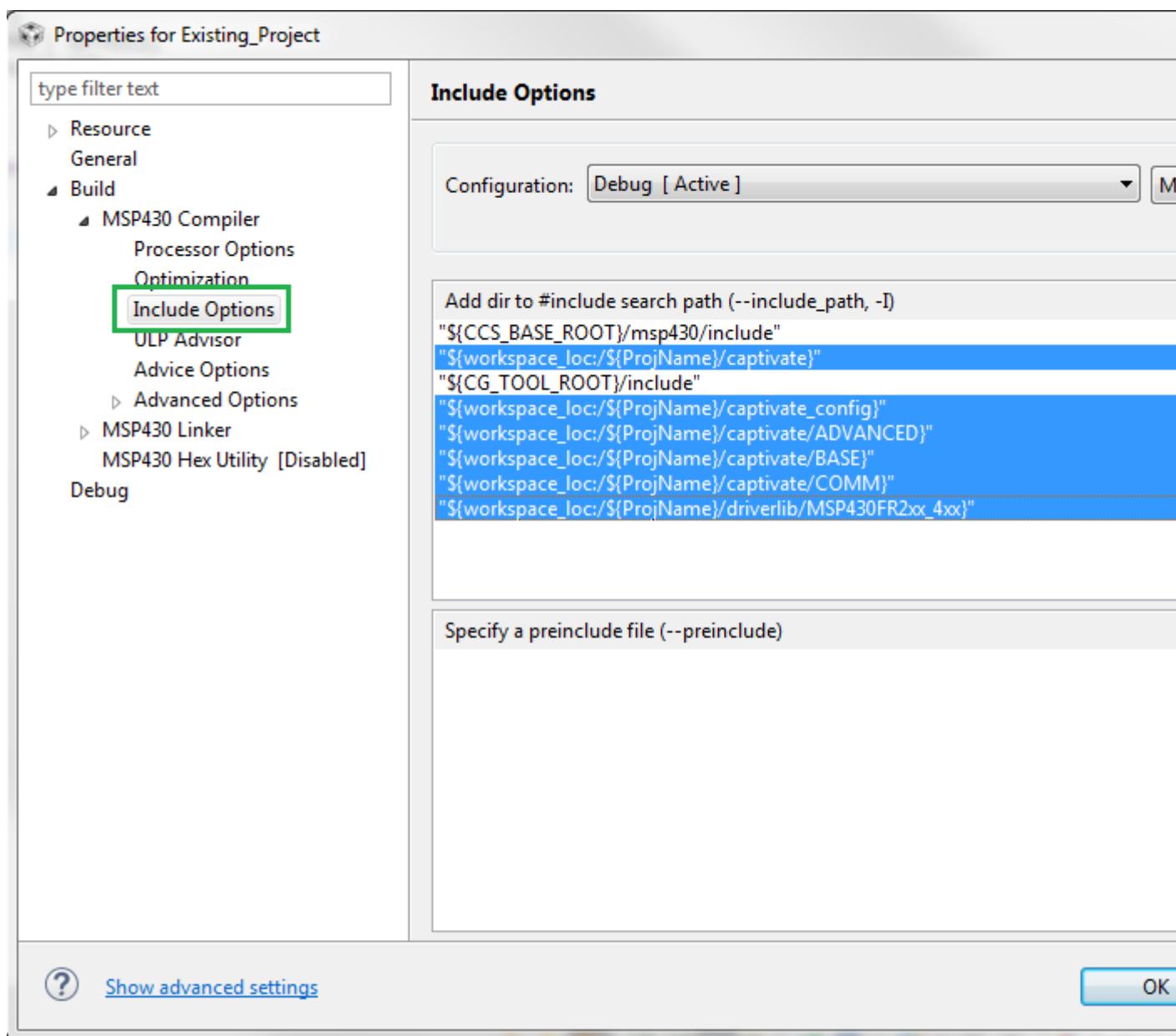


Figure 9.9: Compiler Include Settings

- 2b. Change the existing project's compiler processor options settings to use the small code, small data memory model. This is required for the project to be compatible with the ROM and object code library.

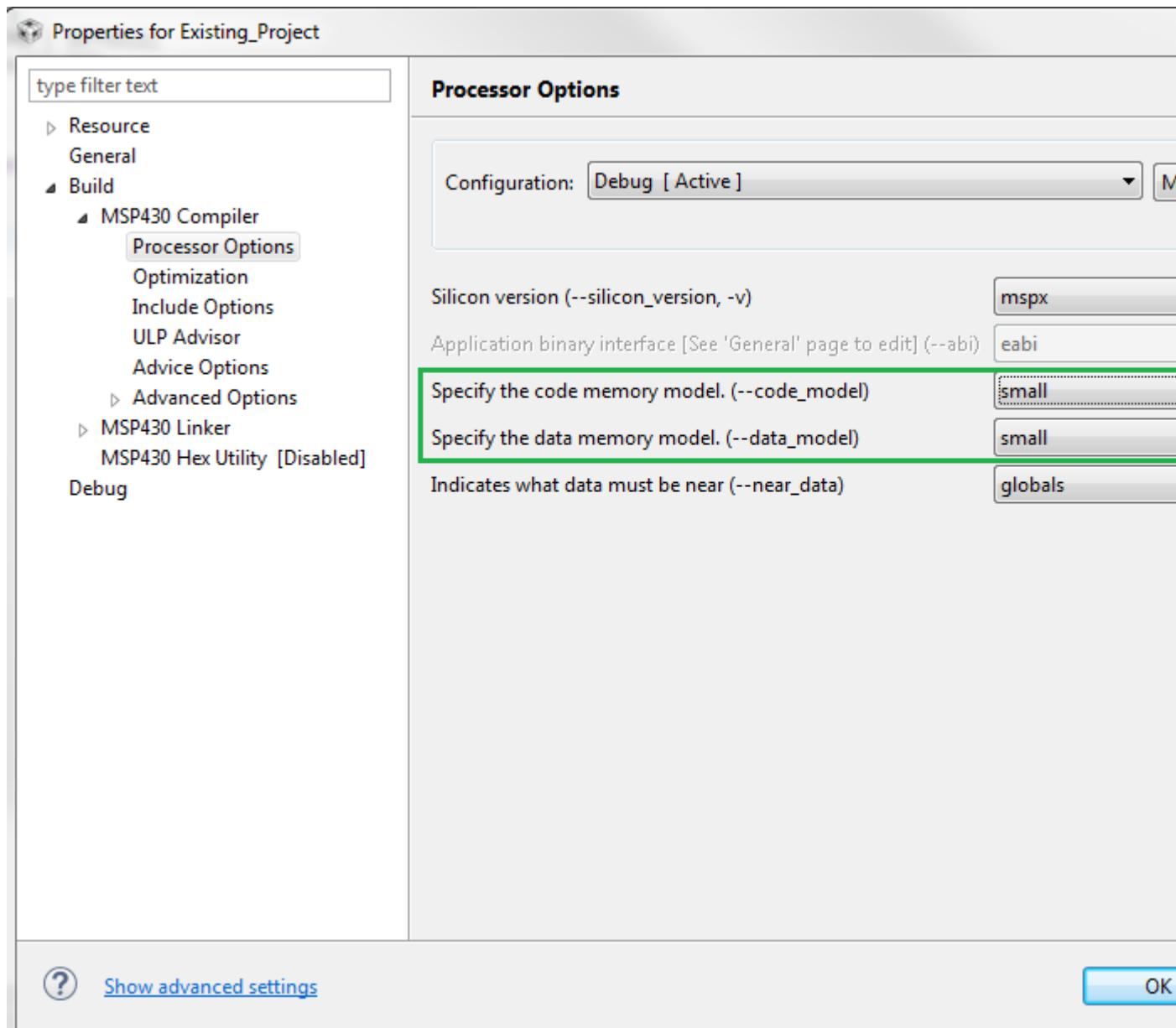


Figure 9.10: Compiler Processor Options

- 2c. Add the following definitions to the project's compiler predefined symbol list:
  - TARGET\_IS\_MSP430FR2633 (To include the CapTlivate™ ROM functions)
  - TARGET\_IS\_MSP430FR2XX\_4XX (To include the DriverLib ROM functions)

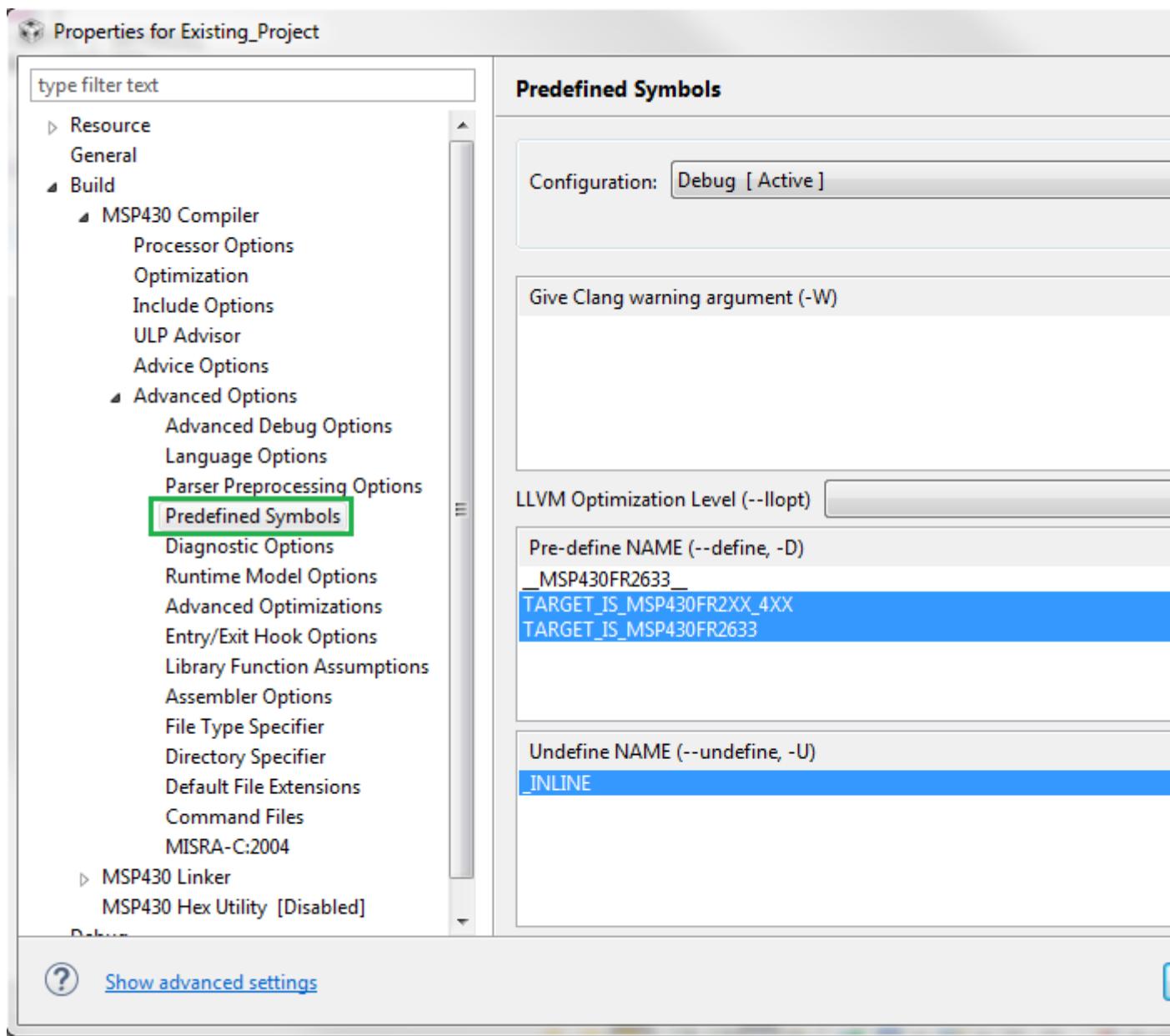


Figure 9.11: Compiler Pre-Defined Symbols

### Step 3: Configuring the Existing Project's Linker Settings for CapTlve™

- 3a. Add the CapTlve™ library archive to the linker's input. If you are using the pre-compiled DriverLib directory from the starter project, add the DriverLib library archive as well, as shown below.

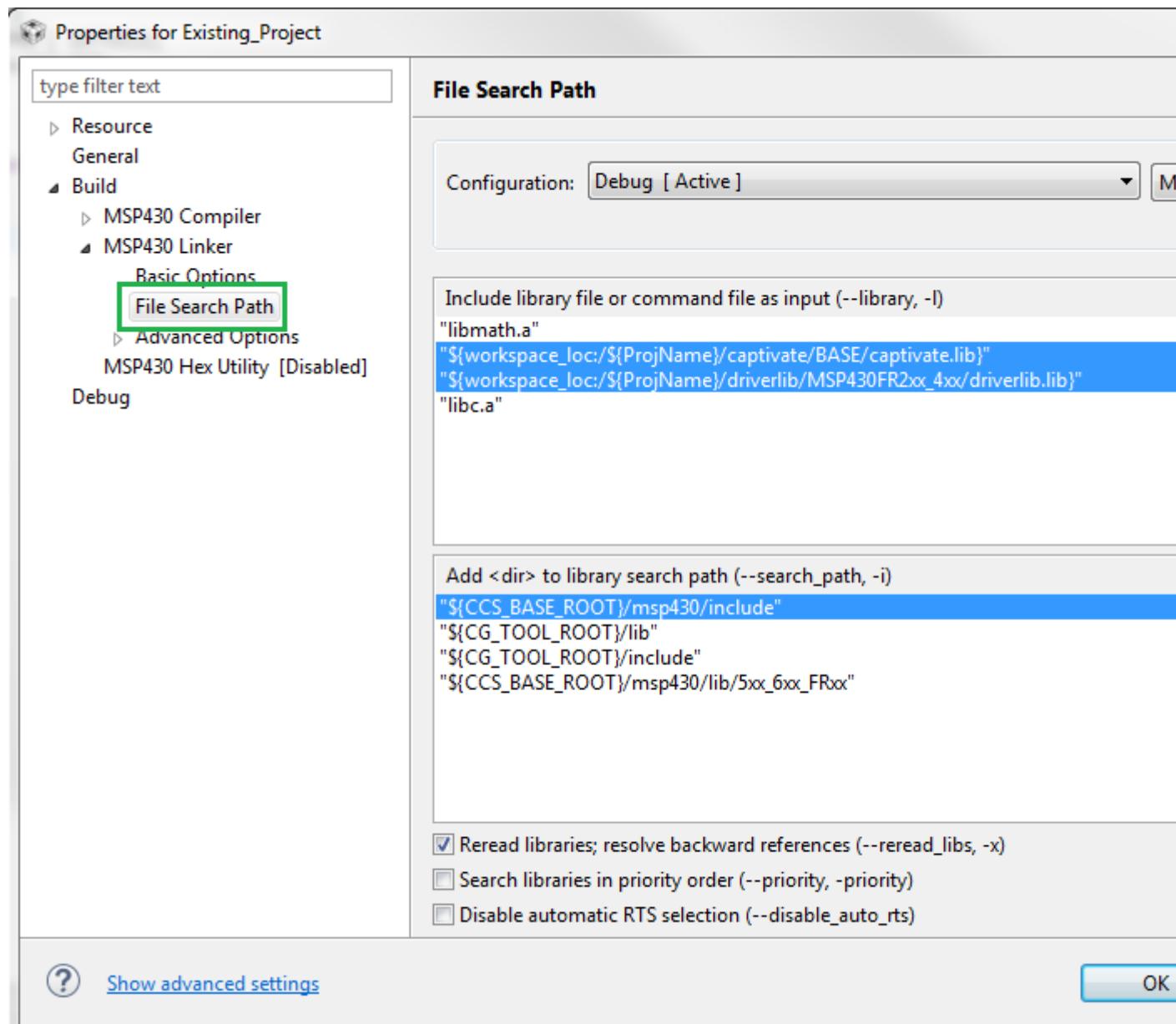


Figure 9.12: Linker Inputs

### Step 4: Add Top Level API Calls to Begin Using CapTlve™

- 4a. Insert calls to initialize and calibrate the user interface once at the beginning of the application. Be sure to also add the include statement for the CapTlve library.

```
// Step 4a:  
CAPT_initUI(&g_uiApp);  
CAPT_calibrateUI(&g_uiApp);
```

If noise immunity (EMC) features are going to be enabled for this design, it is also necessary to link the EMC configuration structure to the EMC module via a call to **CAPT\_loadEMCConfig()**. This must be done before calling **CAPT\_calibrateUI()**, so that the EMC configuration parameters are available during the calibration process.

---

```
// Step 4a with EMC:
CAPT_initUI(&g_uiApp);
CAPT_loadEMCConfig(&g_EMConfig);
CAPT_calibrateUI(&g_uiApp);
```

- 4b. Configure the CapTlve™ interval timer to periodically set the **g\_bConvTimerFlag** status flag.

```
// Step 4b:
MAP_CAPT_selectTimerSource(CAPT_TIMER_SRC_ACLK);
MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ul16ActiveModeScanPeriod));
MAP_CAPT_startTimer();
MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);
```

- 4c. Begin updating the UI! Add a test in the background loop to see if the conversion timer flag has been set. If it has, clear it and update the user interface via **CAPT\_updateUI()**.

### main.c

```
#include <msp430.h>
#include "captivate.h"

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    // Step 4a:
    CAPT_initUI(&g_uiApp);
    CAPT_calibrateUI(&g_uiApp);

    // Step 4b:
    MAP_CAPT_selectTimerSource(CAPT_TIMER_SRC_ACLK);
    MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ul16ActiveModeScanPeriod));
    MAP_CAPT_startTimer();
    MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);

    while(1)
    {
        // Step 4c:
        if (g_bConvTimerFlag == true)
        {
            g_bConvTimerFlag = false;
            CAPT_updateUI(&g_uiApp);
        }
    }
}
```

- 4d. Check the touch status of the sensor by testing its **bSensorTouch** flag. This flag will indicate if any element in the sensor has a touch detection. After **CAPT\_updateUI()** is called, all of the sensor and element objects in the UI will have updated status variables.

```
while(1)
{
    // Step 4c:
    if (g_bConvTimerFlag == true)
    {
        g_bConvTimerFlag = false;
        CAPT_updateUI(&g_uiApp);

        // Step 4d:
        if (keypad.bSensorTouch == true)
        {
            __no_operation();
        }
    }
}
```

### Step 5: [Optional] Optimizing the Clock System for CapTlve™

The default clock frequency for the DCO is approximately 1 MHz. The CapTlve™ Software Library runs most efficiently at 8 MHz. At 8 MHz, no memory access wait states are required, providing an efficient UA/MHz ratio. To increase the DCO clock frequency to 8 MHz, insert calls to DriverLib to configure the clock system as shown below. It is best to configure the clock system before any calls to the CapTlve™ Software Library.

---

```

#define MCLK_FREQ 8000000
#define FLLREF_FREQ 32768
#define FLL_RATIO (MCLK_FREQ / FLLREF_FREQ)

// Step 5:
CS_initClockSignal(CS_FLLREF, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
CS_initClockSignal(CS_ACLK, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
CS_initClockSignal(CS_MCLK, CS_DCOCCLKDIV_SELECT, CS_CLOCK_DIVIDER_1);
CS_initClockSignal(CS_SMCLK, CS_DCOCCLKDIV_SELECT, CS_CLOCK_DIVIDER_4);
CS_initFLLSettle((MCLK_FREQ/1000), FLL_RATIO);
while (CS_getFaultFlagStatus(CS_DCOFFG | CS_FLLULIFG))
{
    CS_clearFaultFlag(CS_DCOFFG | CS_FLLULIFG);
}

```

Note that SMCLK is configured to run at DCO/4, or 2 MHz. This is the frequency that the CapTIvate™ Software Library COMM module expects in order to generate UART baud rates correctly.

#### Step 6: [Optional] Muxing IO for Communications

To add the ability to communicate with the CapTIvate™ Design Center, it is necessary to configure the CapTIvate™ user configuration for a communication interface (UART or I2C) and mux the appropriate peripheral to device pins. See the appropriate device datasheet and device family user's guide for information on how to mux digital functions to device pins. The starter project also provides an example of how to mux the eUSCI\_A0 and eUSCI\_B0 peripherals on the MSP430FR2633.

#### Completed Example Application

Below is the completed example:

##### main.c

```

#include <msp430.h>
#include "captivate.h"
#include "driverlib.h"

#define MCLK_FREQ 8000000
#define FLLREF_FREQ 32768
#define FLL_RATIO (MCLK_FREQ / FLLREF_FREQ)

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    // Step 5:
    CS_initClockSignal(CS_FLLREF, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
    CS_initClockSignal(CS_ACLK, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
    CS_initClockSignal(CS_MCLK, CS_DCOCCLKDIV_SELECT, CS_CLOCK_DIVIDER_1);
    CS_initClockSignal(CS_SMCLK, CS_DCOCCLKDIV_SELECT, CS_CLOCK_DIVIDER_4);
    CS_initFLLSettle((MCLK_FREQ/1000), FLL_RATIO);
    while (CS_getFaultFlagStatus(CS_DCOFFG | CS_FLLULIFG))
    {
        CS_clearFaultFlag(CS_DCOFFG | CS_FLLULIFG);
    }

    // Step 4a:
    CAPT_initUI(&g_uiApp);
    CAPT_calibrateUI(&g_uiApp);

    // Step 4b:
    MAP_CAPT_selectTimerSource(CAPT_TIMER_SRC_ACLK);
    MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ul16ActiveModeScanPeriod));
    MAP_CAPT_startTimer();
    MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);

    while(1)
    {
        // Step 4c:
        if (g_bConvTimerFlag == true)
        {
            g_bConvTimerFlag = false;
            CAPT_updateUI(&g_uiApp);

            // Step 4d:
            if (keypad.bSensorTouch == true)
            {
                __no_operation();
            }
        }
    }
}

```

```
    }  
}
```

## 9.4 How-To

The how-to section of the CapTlivate™ Software Library chapter contains basic code snippets that demonstrate how to perform a simple task, such as measuring a sensor, checking the status of a sensor, or accessing raw data.

### 9.4.1 Use the Top Level API

The top level API of the CapTlivate™ Software Library provides a very simple, highly abstracted programming interface to the library. To get an application up and running, it is only necessary to have knowledge of three basic functions: **CAPT\_initUI()**, **CAPT\_calibrateUI()**, and **CAPT\_updateUI()**.

#### Scope

As introduced in the [programming model](#) section, the top level API functions operate solely on the top level application object, or *tCaptivateApplication*. This object contains all of the information that is needed to run the user interface. It contains the links to all of the sensors that are in the UI. It is important to understand that when a top level API function is used, all of the sensors that are associated with the application are affected. For example, calling **CAPT\_calibrateUI()** causes each sensor in the UI to be calibrated.

#### Open Source

The top level API functions are delivered as open source functions in the library, so that the software designer can understand how the functions work. The functions exist in the *CAPT\_Manager.c* and *CAPT\_Manager.h* files, which are a part of the **ADVANCED** module of the library.

#### Setting up an Application

When setting up an application, it is important to note that there are two types of functions in the top level API:

- Initialization functions (Run one time at start-up)
- Periodic functions (called periodically while the application is running to do something)

The initialization functions are **CAPT\_initUI()** and **CAPT\_calibrateUI()**. **CAPT\_updateUI()** is a periodic function. The initialization functions must be called at the beginning of the application. The behavior of each function is described below.

#### CAPT\_initUI()

The **CAPT\_initUI()** function is responsible for the following actions at start-up:

1. Power on the CapTlivate™ peripheral
2. Initialize the CapTlivate™ peripheral global settings
3. Configure each sensor's IO
4. Initialize each sensor
5. If communications are enabled, initialize the library's communication (COMM) module

Essentially, this function takes care of the one-time settings that do not change once the application is up and running. It should be called one time before any other CapTlivate™ library function is called.

---

### CAPT\_calibrateUI()

The CAPT\_calibrateUI() function is responsible for obtaining coarse gain, fine gain, and offset subtraction calibration values for every element in every sensor in the application. Once these calibration values are obtained via this function call, they will be applied every time an element is measured to provide the correct amount of gain and offset. For details on how gain and offset parameters work, see the [peripheral section](#) of the technology guide.

This top-level calibration function is also responsible for determining whether to use the standard calibration routine (**CAPT\_calibrateSensor()**) or the EMC calibration routine (**CAPT\_calibrateSensorWithEMC()**), depending on whether or not noise immunity is enabled. If noise immunity is enabled, the EMC calibration routine provides calibration values at multiple conversion frequencies and performs additional self-test functions.

This function must be called before any UI or sensor update functions are called- otherwise, no calibration values will be present!

In addition to the first call, in certain applications it may be desirable to force a re-calibration at run-time. One example of this scenario is a mobile device that experiences a "negative" or "reverse" touch. If a user is touching a sensor during power-up or reset, the sensor will be calibrated to the touched state rather than the un-touched state. When the user lets go of the sensor, the measurement will change rapidly *against* the expected direction of change. This can be interpreted as a reverse touch scenario, after which it may be desirable to re-calibrate the entire user interface to ensure a good starting point.

### CAPT\_updateUI()

The CAPT\_updateUI() function is responsible for the following actions:

1. Updating each sensor
2. Testing each sensor for a re-calibration condition
3. If communications are enabled, attempting transmission of element and/or sensor data via the COMM module

This top-level update function is also responsible for determining whether to use the standard update routine (**CAPT\_updateSensor()**) or the EMC update routine (**CAPT\_updateSensorWithEMC()**), depending on whether or not noise immunity is enabled.

This function should be called periodically to update all of the sensors in the system. After this function is called, the following values are updated:

- Raw count
- Filtered count
- Long term average (LTA)
- Detect and negative touch flags
- Touch and proximity flags
- Previous touch flags
- Dominant element ID (for a button group)
- Position (for a slider or wheel)
- Max count error flags and noise state flags
- De-bounce counters

After the values are updated via a call to **CAPT\_updateUI()**, any value can be checked by referencing it in the appropriate data structure.

---

## Simple Code Example

The example below demonstrates the structure of a typical application. For simplicity, this example application continuously measures the UI and does not go to sleep in between scans. To enable scheduled scanning, it is necessary to use a timer to trigger the update, such as the CapTlivate™ interval timer.

```
// Execute one-time setup functions
CAPT_initUI(&g_uiApp);
CAPT_calibrateUI(&g_uiApp);

while(1)
{
    // Continuously update the UI
    CAPT_updateUI(&g_uiApp);
}
```

### 9.4.2 Register a Callback Function

Callbacks provide a mechanism for the application to be notified when a sensor has been updated. The application must first register its "callback" function for each sensor before it can receive updates. When the callback is executed, the application can query the sensor's data structure to determine the status of the sensor.

The library function **CAPT\_registerCallback()** provides the registration.

#### Format

Sensor callback functions are passed a pointer to the calling sensor, and they must return void. An example skeleton callback function is shown below:

```
void my_button_callback(tSensor* pSensor)
{
    // DO SOMETHING ...
}
```

#### Example

This is what it would look like to register the callback function named "my\_button\_callback" to the sensor BTN0000.

```
MAP_CAPT_registerCallback(&BTN0000, &my_button_callback);
```

Once an application's callback function is registered, the callback is executed each time the corresponding sensor is scanned and processed, regardless if a proximity or touch detection has occurred. Inside the user callback, the application can perform any required sensor data and status post-processing. In a typical button application, this is where the application will check for a proximity, touch detection or slider/wheel position.

#### Example

Here is a typical callback example for a button that checks both when a touch is detected and a release.

```
void my_button_callback(tSensor* pSensor)
{
    if((pSensor->bSensorTouch == true) && (pSensor->bSensorPrevTouch == false))
    {
        // BUTTON PRESSED

        // DO SOMETHING ...
    }
    else if((pSensor->bSensorTouch == false) && (pSensor->bSensorPrevTouch == true))
    {
        // BUTTON RELEASED

        // DO SOMETHING ...
    }
}
```

Here is a callback example for a proximity sensor.

```
void my_proximity_callback(tSensor* pSensor)
{
```

---

```

    if(pSensor->bSensorProx == true)
    {
        // PROXIMITY DETECTED
        // DO SOMETHING ...
    }
}

```

In addition to proximity and touch status, sliders and wheels also provide position status. Here is a typical callback example for a slider or wheel that checks a sensor's position.

```

void my_slider_callback(tSensor* pSensor)
{
    uint16_t ui16Position;

    // FIRST CHECK IF THERE IS VALID TOUCH
    if(pSensor->bSensorTouch == true)
    {
        // THEN GET THE CURRENT TOUCH POSITION ON THE SLIDER/WHEEL
        ui16Position = (uint16_t)((tSliderSensorParams*)pSensor->pSensorParams)->SliderPosition.ui16Natural
        ;

        // DO SOMETHING WITH POSITION ...
    }
}

```

### 9.4.3 Access Element State Data

Each element in a sensor has a set of boolean state flags that indicate its status. The following flags are provided:

- Detection
- Negative touch detection
- Proximity
- Touch
- Built-in-self-test (BIST) status
- Noise status

When a sensor is updated (via **CAPT\_updateUI()**, **CAPT\_updateSensor()**, or **CAPT\_updateSensorWithEMC()**), these status flags are updated for every element within the sensor. There are multiple ways to retrieve the data for processing.

#### Accessing Element State Data Directly

It is possible to access element state data directly in an element's data structure. To do this, it is necessary to know the name of the variable in the element. An example is shown below that uses element 0 of a sensor named *keypad* to control some other function, such as illuminating an LED.

```

extern tElement keypad_E00;

void updateLED(void)
{
    if(keypad_E00.bTouch == true)
    {
        // ILLUMINATE LED
    }
    else
    {
        // TURN LED OFF
    }
}

```

---

## Accessing Element State Data Indirectly

Note that in the example above, it was necessary to forward declare keypad\_E00. It is also possible to "look up" E00 of the keypad sensor though the parent sensor structure, as shown below. All sensor structures are forward declared in the user configuration header file (*CAPT\_UserConfig.h*), and do not need to be re-declared. The element of interest is accessed via the cycle pointer array and the element pointer array of that cycle.

```
void updateLED(void)
{
    if(keypad.pCycle[0]->pElements[0]->bTouch == true)
    {
        // ILLUMINATE LED
    }
    else
    {
        // TURN LED OFF
    }
}
```

## Generating a Status Bit Field for all Elements in a Sensor

The above access methods shown are simple and do not require very much memory. However, they only provide data for one element. The library function **CAPT\_getElementStateBitField()** returns a bit field in which each element is represented with a bit position. The bit field supports up to 64 elements. The return type is a 64-bit unsigned integer, which may be casted down to the size that is needed for the application.

Elements are mapped to bit positions starting with the first element of the first cycle to the last element of the last cycle. For example, a sensor with two cycles and two elements in each cycle would have the following mapping:

- Return value bit 0 (0x01): Cycle 0 Element 0
- Return value bit 1 (0x02): Cycle 0 Element 1
- Return value bit 2 (0x04): Cycle 1 Element 0
- Return value bit 3 (0x08): Cycle 1 Element 1

This function may be used to query any of the element status flags. The example below tests the touch status flag. If element 0 and element 1 (BIT0 and BIT1), or 0x03, are touched, the LED would be illuminated.

```
void updateLED(void)
{
    uint8_t multiTouchState;
    multiTouchState = (uint8_t)CAPT_getElementStateBitField(&keypad, eTouchStatus);

    // If '0x03', or element 0 and element 1, are both in detect:
    if (multiTouchState & 0x03)
    {
        // ILLUMINATE LED
    }
    else
    {
        // LED OFF
    }
}
```

## Accessing a Sensor's Global Flags

Many of the flags that are available at the element level are also available as global flags at the sensor level. The global, sensor-level flags operate as a logical OR of all elements in the sensor. In other words, if any element's flag is set, the sensor's global flag is also set. It this way, it is possible to quickly test one flag to see if anything is happening with a sensor. Then, if something is, the element flags can be used to identify which element(s) threw the flag. In the example below, the LED would be illuminated if any element in the sensor was touched.

```
void sensorHandler(tSensor* pSensor)
{
    if(pSensor->bSensorTouch == true)
```

---

```

    {
        // ILLUMINATE LED
    }
    else
    {
        // TURN LED OFF
    }
}

```

In addition to the global sensor touch flag (*bSensorTouch*), there is also a global sensor previously touched flag (*bSensorPrevTouch*). This flag is set if *bSensorTouch* was set on the previous sample. This can be used as a mechanism to determine if a touch is new (meaning someone just touched the button) versus stale (meaning the touch on this sample is a continuation of a previously started touch). This allows for toggling between states, as shown below:

```

void sensorHandler(tSensor* pSensor)
{
    if((pSensor->bSensorTouch == true) && (pSensor->bSensorPrevTouch == false))
    {
        // TOGGLE A FUNCTION
    }
}

```

#### Element Status Flag Reference Table

The table below lists the available status flags that all elements have, and the name of the parameter to use when accessing it.

Description	Element Structure Parameter	CAPT_getElementStateBitField() Parameter
Touch Detection	.bTouch	eTouchStatus
Proximity Detection	.bProx	eProxStatus
Negative Touch Detect	.bNegativeTouch	eNegativeTouchStatus
Detect	.bDetect	eDetectStatus
Built-in-self-test Fail	.bBISTFail	eBISTStatus
Noise Detected	.bNoiseDetected	eNoiseStatus

The *detect* status is the un-debounced state of the *prox* status. When an element is in *detect* but not in *prox*, this means that the long term average tracking filter is disabled, but the state change into proximity detection is not yet complete because it is currently being de-bounced.

#### 9.4.4 Access the Dominant Button

Button group sensors output a dominant button ID that corresponds to the element with the highest delta response. This is useful for keypads which do not require multi-touch but would like to have some level of nearby key rejection. For example, if a user is touching in between to keys, the dominant key with the highest delta response will be reported.

When a button group sensor is updated (via **CAPT\_updateUI()**, **CAPT\_updateSensor()**, or **CAPT\_updateSensorWithEMC()**), the dominant element value is updated.

Elements are mapped to IDs starting with the first element of the first cycle to the last element of the last cycle. For example, a sensor with two cycles and two elements in each cycle would have the following mapping:

- 0: Cycle 0 Element 0
- 1: Cycle 0 Element 1
- 2: Cycle 1 Element 0
- 3: Cycle 1 Element 1

---

## Accessing the Dominant Button Directly

The example below shows how to directly access the value in the *tButtonSensorParams* structure. The LED is illuminated if a touch is present and the dominant key is the first element.

```
extern tButtonSensorParams keypadSensor_Params;

void updateLED(void)
{
    if (keypadSensor.bSensorTouch == true)
    {
        if (keypadSensor_Params.ul16DominantElement == 0x00)
        {
            // ILLUMINATE LED
        }
        else
        {
            // TURN OFF LED
        }
    }
    else
    {
        // TURN OFF LED
    }
}
```

## Accessing the Dominant Button Indirectly

Note that in the example above, it was necessary to forward declare *keypadSensor\_Params*. It is also possible to "look up" these parameter structures through the parent sensor structure, as shown below. All sensor structures are forward declared in the user configuration header file (*CAPT\_UserConfig.h*), and do not need to be re-declared. It is necessary to type-cast the parameter structure based on the type of sensor.

```
void updateLED(void)
{
    uint8_t dominantButton;

    if (keypadSensor.bSensorTouch == true)
    {
        dominantButton = ((tButtonSensorParams*) (keypadSensor.pSensorParams)) ->ul16DominantElement;
        if (dominantButton == 0x00)
        {
            // ILLUMINATE LED
        }
        else
        {
            // TURN OFF LED
        }
    }
    else
    {
        // TURN OFF LED
    }
}
```

## Accessing the Dominant Button with a Function Call

The final way to access the dominant button value is via a function call to **CAPT\_getDominantButton()** or **CAPT\_getDominantButtonAddr()**. The former function returns the ID of the dominant button, while the latter function returns the memory address (essentially a pointer to) the dominant element.

The example below demonstrates accessing the dominant button ID via a function call.

```
void updateLEDs(void)
{
    uint8_t dominantElement;

    if ((keypadSensor.bSensorTouch==true) && (keypadSensor.bSensorPrevTouch==false))
    {
        dominantElement = CAPT_getDominantButton(&keypadSensor);
        if (dominantElement == 0)
        {
            LED1_OFF;
            LED2_OFF;
```

```

    }
else if (dominantElement == 1)
{
    LED1_ON;
    LED2_OFF;
}
else if (dominantElement == 2)
{
    LED1_OFF;
    LED2_ON;
}
else if (dominantElement == 3)
{
    LED1_ON;
    LED2_ON;
}
}
}

```

The example below demonstrates processing of the dominant button based on a pointer to the dominant element.

```
extern tElement keypadSensor_E00;
extern tElement keypadSensor_E01;
extern tElement keypadSensor_E02;
extern tElement keypadSensor_E03;

void updateLEDs(tSensor *sensor)
{
    tElement* dominantElement;
    if ((keypadSensor.bSensorTouch==true) && (keypadSensor.bSensorPrevTouch==false))
    {
        dominantElement = CAPT_getDominantButtonAddr(&keypadSensor);
        if (dominantElement == &keypadSensor_E00)
        {
            LED1_OFF;
            LED2_OFF;
        }
        else if (dominantElement == &keypadSensor_E01)
        {
            LED1_ON;
            LED2_OFF;
        }
        else if (dominantElement == &keypadSensor_E02)
        {
            LED1_OFF;
            LED2_ON;
        }
        else if (dominantElement == &keypadSensor_E03)
        {
            LED1_ON;
            LED2_ON;
        }
    }
}
```

#### 9.4.5 Access Slider or Wheel Position Data

Slider and wheel sensors output a position in addition to touch and proximity status. The position value is available as a IQ16-style value, with 16 integer bits and 16 fractional bits. For almost all applications, the integer bits are the bits of interest, and the fractional bits are merely there to support filtering.

When a slider or wheel sensor is updated (via **CAPT\_updateUI()**, **CAPT\_updateSensor()**, or **CAPT\_updateSensorWithEMC()**), the position value is updated.

**NOTE:** When a slider or wheel sensor is not being touched, a value of 0xFFFF (UINT16\_MAX) is reported.

#### **Accessing the Slider or Wheel Position Directly**

The example below shows how to directly access the slider and wheel position parameters in the *tSliderSensorParams* and *tWheelSensorParams* structures, respectively. The application assigns the values to the speakerVolume and optionSelection variables, which are intended to represent functionality in an application.

```
uint16_t speakerVolume;
uint16_t optionSelection;
extern tSliderSensorParams volumeSlider Params;
```

---

```

extern tWheelSensorParams scrollWheel_Params;

void updateVolumeAndOptionSelection(void)
{
    // Set the speaker volume to the natural (integer) 16 bit slider position
    if (volumeSlider_Params.SliderPosition.uil6Natural != UINT16_MAX)
    {
        speakerVolume = volumeSlider_Params.SliderPosition.uil6Natural;
    }

    // Set the option selection to the natural (integer) 16 bit wheel position
    if (scrollWheel_Params.SliderPosition.uil6Natural != UINT16_MAX)
    {
        optionSelection = scrollWheel_Params.SliderPosition.uil6Natural;
    }
}

```

### Accessing the Slider or Wheel Position Indirectly

Note that in the example above, it was necessary to forward declare `volumeSlider_Params` and `scrollWheel_Params`. It is also possible to "look up" these parameter structures through the parent sensor structure, as shown below. All sensor structures are forward declared in the user configuration header file (`CAPT_UserConfig.h`), and do not need to be re-declared. It is necessary to type-cast the parameter structure based on the type of sensor.

```

uint16_t speakerVolume;
uint16_t optionSelection;

void updateVolumeAndOptionSelection(void)
{
    uint16_t position;

    // Set the speaker volume to the natural (integer) 16 bit slider position
    position = ((tSliderSensorParams*)(volumeSlider.pSensorParams))->SliderPosition.uil6Natural;
    if (position != UINT16_MAX)
    {
        speakerVolume = position;
    }

    // Set the option selection to the natural (integer) 16 bit wheel position
    position = ((tWheelSensorParams*)(scrollWheel.pSensorParams))->SliderPosition.uil6Natural;
    if (position != UINT16_MAX)
    {
        optionSelection = position;
    }
}

```

Note that for both slider and wheel parameter structures, the parameter for position is called `SliderPosition`. This is because both of these sensor types utilize the same processing algorithm.

### Accessing the Slider or Wheel Position with a Function Call

The final (and simplest) way to access slider or wheel position is via a function call to `CAPT_getSensorPosition()`. This function will return 0xFFFF (UINT16\_MAX) if no touch is present.

```

uint16_t speakerVolume;
uint16_t optionSelection;

void updateVolumeAndOptionSelection(void)
{
    uint16_t position;

    // Set the speaker volume to the natural (integer) 16 bit slider position
    position = CAPT_getSensorPosition(&volumeSlider);
    if (position != UINT16_MAX)
    {
        speakerVolume = position;
    }

    // Set the option selection to the natural (integer) 16 bit wheel position
    position = CAPT_getSensorPosition(&scrollWheel);
    if (position != UINT16_MAX)
    {
        optionSelection = position;
    }
}

```

---

Using the function call allows the software implementation to be clearer, at the penalty of function overhead.

#### 9.4.6 Access Element Measurement Data

Each element in a sensor has several variables that contain the current measurement data. The following values are provided:

- Filtered Count (The conversion result)
- Long Term Average (LTA) (The baseline reference)
- Raw Count(s) (The raw data sample before any processing is applied)
- Composite Raw Count (The composite data sample of a multi-frequency or oversampled conversion)
- Previous Composite Raw Count (The previous composite data sample of a multi-frequency or oversampled conversion)

When a sensor is updated (via **CAPT\_updateUI()**, **CAPT\_updateSensor()**, or **CAPT\_updateSensorWithEMC()**), these variables are updated for every element within the sensor. There are multiple ways to retrieve the data.

##### Accessing Element State Data Directly

It is possible to access element measurement data directly in an element's data structure. To do this, it is necessary to know the name of the variable in the element. An example is shown below that uses element 0 of a sensor named *keypad*.

```
extern tElement keypad_E00;

void myDataFunction(void)
{
    uint16_t data;

    // Place the filtered count value into the 'data' variable.
    // Note that the filtered count value is an IQ16 format variable with 16 integer bits and 16 fractional
    // bits.
    // Typically, only the integer bits are of interest.
    data = keypad_E00.filterCount.uil6Natural;
    __no_operation();

    // Place the long term average into the 'data' variable.
    // Note that the LTA is an IQ16 format variable with 16 integer bits and 16 fractional bits.
    // Typically, only the integer bits are of interest.
    data = keypad_E00.LTA.uil6Natural;
    __no_operation();

    // Place the raw count value into the 'data' variable.
    // Note that there may be either 1 or 4 raw count variables, depending on whether the conversion
    // type used is single or multi-frequency.
    data = keypad_E00.pRawCount[0];
    __no_operation();

    // If the conversion is multi-frequency, then keypad_E00.pRawCount[0] holds frequency 0's result,
    // and keypad_E00.pRawCount[3] holds frequency 3's result.
    // The raw composite result of all 4 frequencies is placed in the keypad_E00.uil6CompositeRawCount
    // variable.
    data = keypad_E00.uil6CompositeRawCount;
    __no_operation();
}
```

##### Accessing Element Measurement Data Indirectly

Note that in the example above, it was necessary to forward declare `keypad_E00`. It is also possible to "look up" `E00` of the `keypad` sensor through the parent sensor structure, as shown below. All sensor structures are forward declared in the user configuration header file (`CAPT_UserConfig.h`), and do not need to be re-declared. The element of interest is accessed via the cycle pointer array and the element pointer array of that cycle.

---

```

void myDataFunction(void)
{
    uint16_t data;

    // Place the filtered count value into the 'data' variable.
    // Note that the filtered count value is an IQ16 format variable with 16 integer bits and 16 fractional
    // bits.
    // Typically, only the integer bits are of interest.
    data = keypad.pCycle[0]->pElements[0]->filterCount.ul16Natural;
    __no_operation();
}

```

### 9.4.7 Update Sensors Independently

Up until this point, all discussion around measuring sensors has been via the top level API- specifically, the **CAPT\_updateUI()** call. When using the **CAPT\_updateUI()** function call, all sensors in the application are updated. For most applications, this is the desired operation. However, there are cases where it may be desired to update sensors individually or at different rates. This how-to explains the function calls that are used to individually update sensors. There are two function calls available for updating a sensor: **CAPT\_updateSensor()** and **CAPT\_updateSensorWithEMC()**.

#### **CAPT\_updateSensor()**

This is the standard sensor update function. After calling this function, the following values are updated for the passed sensor only:

- Raw counts
- Filtered counts
- Long term averages (LTAs)
- Detect and negative touch flags
- Touch and proximity flags
- Previous touch flags
- Dominant element ID (for a button group)
- Position (for a slider or wheel)
- Max count error flags and noise state flags
- De-bounce counters

Below is the syntax used to call the function. The parameters include a pointer to the sensor to update, and the low power mode bits to set during the conversion process. LPM0, LPM1, LPM2, and LPM3 may be used.

```
CAPT_updateSensor(&keypadSensor, LPM0_bits);
```

#### **CAPT\_updateSensorWithEMC()**

The EMC version of the sensor update call provides the same end functionality as the standard call, with the exception that EMC plug-ins from the advanced layer are applied. When this function is used, the EMC configuration structure defines the style of conversion to use. This may mean that multi-frequency scanning and/or oversampling is utilized.

The syntax for the EMC version is identical to the standard version, as shown below. This is to provide a standard call so that application code does not need to change significantly to accommodate switching to an EMC scanning mode. All EMC plug-in configuration is controlled by the EMC configuration structure.

```
CAPT_updateSensorWithEMC(&keypadSensor, LPM0_bits);
```

---

## Important Functionality to Handle

When the sensor update functions are used rather than the top level API, several other tasks need to be handled by the application, such as testing for a re-calibration condition or transmitting data.

### Testing for Re-Calibration

Over time, the long term average of a sensor may drift. To ensure that consistent sensitivity is always provided, the software library provides a mechanism to test to see if any element in a sensor has drifted outside of an acceptable boundary. This mechanism is the **CAPT\_testForRecalibration()** function. The **CAPT\_updateUI()** function takes care of handling this when the top level API is used, but if sensors are updated individually then this needs to be handled by the application.

The typical handling method is shown below:

```
// Update the individual sensor(s)
CAPT_updateSensor(&keypadSensor, LPM0_bits);

// Test for a re-calibration condition
if (CAPT_testForRecalibration(&keypadSensor) == true)
{
    // If a re-calibration is required, perform it now
    CAPT_calibrateSensor(&keypadSensor);
}
```

For details on how the re-calibration test works, see the [runtime re-calibration definition](#).

### Transmitting Sensor and Element Data

If communication via the COMM module is desired, it is necessary to add the calls to transmit the sensor and element data for this sensor via the COMM module.

```
//
// If the UART or Bulk I2C interface is enabled, write out element
// and sensor data.
//
#if ((CAPT_INTERFACE==__CAPT_UART_INTERFACE__) || \
     (CAPT_INTERFACE==__CAPT_BULKI2C_INTERFACE__))
CAPT_writeElementData(x);
CAPT_writeSensorData(x);
#endif
```

In this code example, 'x' represents the sensor's integer ID. This ID is the position of the sensor in the global sensor pointer array. This is the array that the COMM module uses to look up sensors.

For the CAPTIVATE-BSWP demo panel, the array looks like this:

```
tSensor* g_pCaptivateSensorArray[CAPT_SENSOR_COUNT] =
{
    &keypadSensor,
    &proximitySensor,
    &sliderSensor,
    &wheelSensor,
};
```

Thus, the ID of the keypad sensor would be 0.

### Full Implementation

Putting it all together, updating a sensor individually would have the following progression:

```
// Update the individual sensor(s)
CAPT_updateSensor(&keypadSensor, LPM0_bits);

// Test for a re-calibration condition
if (CAPT_testForRecalibration(&keypadSensor) == true)
{
    // If a re-calibration is required, perform it now
```

---

```

        CAPT_calibrateSensor(&keypadSensor);

    }

// If communications are enabled, transmit data now
#if ((CAPT_INTERFACE==__CAPT_UART_INTERFACE__) || \
     (CAPT_INTERFACE==__CAPT_BULKI2C_INTERFACE__))
CAPT_writeElementData(0);
CAPT_writeSensorData(0);
#endif

```

#### 9.4.8 Update a Sensor's Raw Data Only

For certain custom applications it may be desirable to only update a sensor's raw data after a conversion, bypassing all of the high-level processing. For applications that require this, the [CAPT\\_updateSensorRawCount\(\)](#) may be used directly. This function only updates raw count values for each element in the sensor. No processing is performed on the data, and the sensor callback function is not called upon completion of the update. The raw data update function takes two additional parameters that specify details about type of conversion. An example function call is shown below that updates the raw data for a sensor named *keypad*. For details on the conversion type and oversampling type parameters, see the [CAPT\\_updateSensorRawCount](#) overview.

```

CAPT_updateSensorRawCount(
    &keypadSensor,           // Pointer to the sensor to update
    eStandard,               // Conversion type
    eNoOversampling,         // Oversampling type
    LPM0_bits                // Low power mode to use
);

```

After the update is complete, results may be looked up in each element's data structure. The following values are updated by this function:

- Composite Raw Count (The raw result after the specied multi-frequency processing or oversampling)
- Previous Composite Raw Count (The composite raw result from the previous sample)
- Noise Level (if the conversion was a multi-frequency conversion, the spread between data at frequencies)
- Raw Count for Each Frequency (if the conversion was a multi-frequency conversion, else a single frequency)

Below is an example of measuring a sensor named *keypad* without frequency hopping and with an oversampling level of 2. The composite output is read and used to perform some unknown task.

```

extern tElement keypadSensor_E00;
uint16_t rawSample;

// Perform the update
CAPT_updateSensorRawCount(
    &keypadSensor,           // Pointer to the sensor to update
    eStandard,               // Conversion type
    e2xOversampling,         // Oversampling type
    LPM0_bits                // Low power mode to use
);

// Read out the data
rawSample = keypadSensor_E00.ul16CompositeRawCount;

// Do Something ...

```

#### 9.4.9 Enable an IO as a Shield

In self-capacitance mode, it is possible to enable an extra CapTlivate™ sensing IO to fire in phase with other IOs during a conversion. This provides a shielding effect that can reduce the parasitic capacitance of the sensing IOs.

In order to realize a shield IO, the following must be true:

1. The sensor of interest must be a self-capacitance sensor

2. The shield must be connected to an IO on a CapTlve™ block that is not shared with the sensor of interest. For example, if the sensor of interest is a proximity sensor on CAP0.0 (block 0, pin 0), the shield may be connected to any pin on CAP1.x, CAP2.x, or CAP3.x- but it may NOT be connected to any pin on CAP0.x, since the proximity sensor (the sensor of interest) is on that block. If a shield is enabled on the same block as the sensor of interest, the effect will be equivalent to a digital "OR" in which the shield and the sensor of interest appear to be connected.

To enable an IO as a shield, use the library function **CAPT\_enableShieldIO()** as shown below. This enables CAP1.0 to be a shield structure.

```
// Enable shield IO before calibration:  
CAPT_enableShieldIO(1, 0);  
  
// If needed, the shield can be disabled:  
CAPT_disableShieldIO(1, 0);
```

#### 9.4.10 Create a Custom EMC Configuration

When noise immunity is enabled for a design, or when any of the `*WithEMC()` function calls are used, the EMC processing plug-ins from the ADVANCED module are applied. The EMC processing plug-ins are configured via a data structure. A default configuration is provided in the user configuration file that works well for most applications. However, if some customization is needed the default structure in the user configuration file may be overridden and a new structure may be provided in the application.

While it is possible to directly edit the structure in the user configuration file, it is best to make the changes elsewhere as the user configuration file is auto-generated by the CapTlve™ Design Center, and any changes will be lost when an update is performed.

To add a custom EMC configuration, create a new EMC configuration data structure by copying the structure from the user configuration file and placing it in the application. It must be re-named with a unique name. The new structure may be placed in the `CAPT_App.c` file, if desired. Below is an example:

```
const tEMCConfig myCustomEMCConfig =  
{  
    // Conversion Style  
    .selfModeConversionStyle = eMultiFrequency,  
    .projModeConversionStyle = eMultiFrequencyWithOutlierRemoval,  
  
    // Oversampling Style  
    .selfModeOversamplingStyle = eNoOversampling,  
    .projModeOversamplingStyle = eNoOversampling,  
  
    // Jitter Filter Enable  
    .bJitterFilterEnable = true,  
  
    // Noise Thresholds and Calibration Noise Limits  
    .ui8NoiseThreshold = 20,  
    .ui16CalibrationNoiseLimit = 10,  
    .ui8CalibrationTestSampleSize = 8,  
  
    // Dynamic Threshold Adjustment Parameters  
    .bEnableDynamicThresholdAdjustment = true,  
    .ui8MaxRelThreshAdj = 76,  
    .ui8NoiseLevelFilterEntryThresh = 40,  
    .ui8NoiseLevelFilterExitThresh = 0,  
    .ui8NoiseLevelFilterDown = 6,  
    .ui8NoiseLevelFilterUp = 1,  
    .coeffA = _IQ31(0.0065),  
    .coeffB = _IQ31(0.050)  
};
```

The structure is not modified at runtime, and thus may be declared as a `const` object.

In the the **CAPT\_appStart()** function, there is a call to **CAPT\_loadEMCConfig()**. Replace the passed configuration structure with the custom configuration structure.

#### Original

```
//  
// Load the EMC configuration, if this design has
```

---

```

// noise immunity features enabled. This function call
// associates an EMC configuration with the EMC module.
//
#ifndef (CAPT_CONDUCTED_NOISE_IMMUNITY_ENABLE==true)
    CAPT_loadEMCConfig(&g_EMCCConfig);
#endif

```

## Modified

```

//
// Load the EMC configuration, if this design has
// noise immunity features enabled. This function call
// associates an EMC configuration with the EMC module.
//
#ifndef (CAPT_CONDUCTED_NOISE_IMMUNITY_ENABLE==true)
    CAPT_loadEMCConfig(&myCustomEMCConfig);
#endif

```

Now, the custom configuration may be modified to suit the needs of the application. For details on how to set the parameters in the EMC configuration structure, see the [EMC Module](#) section.

### 9.4.11 Stream Unformatted Data to the Design Center GUI

In addition to the element and sensor data streaming to the CapTivate™ Design Center customizer windows, a mechanism exists to stream miscellaneous user-defined data to a oscilloscope plot with logging capability.

- The data format is 16-bit unsigned integers.
- Up to 29 values may be streamed.

To stream data, insert a call to the **CAPT\_writeGeneralPurposeData()** function.

The function expects a pointer to an array of 16-bit unsigned integers, and a length value that specifies how many values there are, up to the maximum of 29.

```

#define BUFFER_SIZE (8)
uint16_t buffer[BUFFER_SIZE];

// Transmit the packet via the general purpose data mechanism
CAPT_writeGeneralPurposeData(&buffer[0], BUFFER_SIZE);

```

This mechanism is very helpful during development of noise immunity applications, as it allows for streaming of raw data and multi-frequency data. The code snippet below may be registered as a callback function for a sensor. It streams the data of the first element in the sensor with the following format:

1. LTA
2. Filtered Count
3. Abs Thresh
4. Noise Level
5. F0 Raw Value
6. F1 Raw Value
7. F2 Raw Value
8. F3 Raw Value

```

#define NOISETEST_PACKET_SIZE (8)

void NoiseTest_callbackHandler(tSensor* pSensor)
{
    static uint16_t packet[NOISETEST_PACKET_SIZE];
    tElement *element;

```

```

        uint16_t threshold;
        uint8_t i;
        uint8_t j;

        //
        // Initialize variables
        //
        i = 0;
        element = pSensor->pCycle[0]->pElements[0];

        //
        // Compute the touch threshold
        //
        threshold = element->ui8TouchThreshold;
        if (pSensor->SensingMethod == eSelf)
        {
            threshold += CAPT_computeRelNoiseComp();
            if (threshold > CAPT_getMaxRelThreshold())
            {
                threshold = CAPT_getMaxRelThreshold();
            }
        }
        threshold = CAPT_convertRelToAbs(element->LTA.ul16Natural, threshold);
        if (pSensor->DirectionOfInterest == eDOIDown)
        {
            threshold = element->LTA.ul16Natural - threshold;
        }
        else
        {
            threshold = element->LTA.ul16Natural + threshold;
        }

        //
        // Frame the packet
        //
        packet[i++] = element->LTA.ul16Natural;
        packet[i++] = element->filterCount.ul16Natural;
        packet[i++] = threshold;
        packet[i++] = element->ul16NoiseCount;
        for (j=0; j<4; j++)
        {
            packet[i++] = element->pRawCount[j];
        }

        //
        // Transmit the packet via the general purpose data mechanism
        //
        CAPT_writeGeneralPurposeData(&packet[0], NOISETEST_PACKET_SIZE);
    }
}

```

When communications are enabled, this data will appear in the CapTlivate™ Design Center's user data plot. To view the streaming data, add a user data log bean, as shown below:

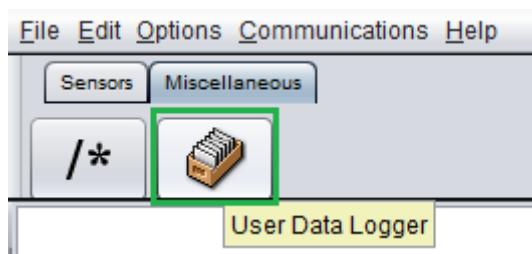


Figure 9.13: User Data Plot

## 9.5 Technical Details

The CapTlivate™ Software Library has certain technical features and requirements, described below.

### 9.5.1 Devices with CapTlivate™ Software in ROM

Some MSP devices have portions of the CapTlivate™ Software Library in ROM, reducing the amount of FRAM or flash memory required for an application. Refer to the device data sheet to determine if a given device has library

---

components in ROM.

#### Calling Functions in ROM

Calling library functions directly from ROM requires the use of two header files that are provided with the library, `rom_captivate.h` and `rom_map_captivate.h`. These header files provide a mapping between functions in the software library archive and the same function located in a device's ROM image. Use the function call technique described in the following example to simplify the procedure.

##### Example:

Assume the `captivate.lib` has been added to the project and header files `rom_captivate.h` and `map_rom_captivate.h` are included in the source file where the library function call will be made. To call the library function "foo()", it is recommended to make an implicit ROM call `MAP_foo()`. The compiler parses through the `rom_map_captivate.h` file to determine if the function resides in ROM, and if it does, will make the ROM call. If this function does not exist in ROM then the compiler will make a call to the pre-compiled library version.

It is possible to explicitly call the function "foo()" from the pre-compiled library using `foo()`. The linker will pull this function from the pre-compiled library during the link process, adding the function to FRAM program memory. To explicitly call the function "foo()" from ROM, use `ROM_foo()`. This will force the compiler to make a call to ROM, with no impact on FRAM program memory.

#### 9.5.2 MSP430 CPUX Memory Model

Smaller memory map devices with the TI MSP430 CPUX core (such as the MSP430FR26xx and MSP430FR25xx devices) have a total memory map that is less than 64kB. Such a memory map is accessible in its entirety with 16 bit pointers. In order to improve execution speed and reduce memory requirements, the ROM functions and pre-compiled library functions are compiled using the small code small data (SCSD) memory model. All CapTlivate™ software library projects must be compiled using the small code small data memory model in order to be compatible with the ROM functions and the pre-compiled library. Using the incorrect memory model will result in a linker error.

## 9.6 Base Module

The base module implements the core of the CapTlivate™ Software Library. It is responsible for providing the base feature set for initializing, calibrating, measuring and processing capacitive sensors.

The base module contains the following components:

- Hardware Abstraction Layer (HAL)
- Touch Layer
- Interrupt Service Routine (ISR)
- CapTlivate™ Software Library Type Definitions

#### 9.6.1 HAL

The hardware abstraction layer provides access to the CapTlivate™ peripheral. This includes functions for performing the following tasks:

- Configuring the CapTlivate™ peripheral periodic timer
- Handling CapTlivate™ peripheral interrupts
- Configuring IO for CapTlivate™

- 
- Configuring conversion settings
  - Loading and storing conversion results

## 9.6.2 Touch

The touch layer sits on top of the HAL layer and provides the sensor update routines, calibration algorithms, and basic sensor processing algorithms.

### 9.6.2.1 Sensor Update Routines

Several sensor update routines are available in the touch module:

1. **CAPT\_updateSensor()**
2. **CAPT\_updateSensorWithEMC()**
3. **CAPT\_updateSensorRawCount()**

#### 9.6.2.1.1 CAPT\_updateSensor()

This is the standard, fundamental sensor update routine that is used in most applications. Calling this function will immediately measure all of the elements within the sensor, and perform all of the standard signal processing, including:

1. Applying any IIR count filtering as configured in the sensor structure. This function utilizes the hardware accelerated IIR filtering in the CapTlivate™ finite state machine.
2. Testing of touch, proximity, and negative touch thresholds. This function utilizes the detection capability of the CapTlivate™ finite state machine to detect proximity and negative touch events.
3. Processing of the long term average (LTA) as configured in the sensor structure. This function utilizes the hardware accelerated IIR filtering in the CapTlivate™ finite state machine to process the LTA.
4. Application of touch/proximity de-bounce as configured in the sensor structure.
5. Testing for re-calibration, if so configured in the sensor structure.

The function takes two parameters:

- A pointer to the sensor structure to update
- The low power mode to use during the conversion

```
CAPT_updateSensor(&mySensor, LPM3_bits);
```

#### Low Power Mode

The low power mode may be LPM0 to LPM3. The function returns when all measurements and processing are complete.

#### 9.6.2.1.2 CAPT\_updateSensorWithEMC()

This function is identical to **CAPT\_updateSensor()**, with the addition of EMC processing components from the EMC module. See the [EMC module](#) for more details.

### 9.6.2.1.3 CAPT\_updateSensorRawCount()

This is a basic function that may be used if access to the raw conversion data if that is all that is desired. Using this function bypasses all of the higher-level processing. Filtered count, long term average, and status parameters are not maintained. Only the raw results are populated. However, the function does allow for some low-level signal processing algorithms to be applied.

The function takes 4 parameters:

- A pointer to the sensor structure to update
- A conversion style specification
- An oversampling specification
- The low power mode to use during the conversion

```
CAPT_updateSensorRawCount (
    &keypadSensor,           // Pointer to the sensor to update
    eStandard,              // Conversion type
    eNoOversampling,        // Oversampling type
    LPM0_bits               // Low power mode to use
);
```

#### Conversion Style

The conversion style control influences the type of conversion used to update the raw values. The output is stored in the \*.ui16CompositeRawCount\* parameter of each element.

Option	tRawConversionStyle Enumeration
Standard	eStandard
Multi-Frequency	eMultiFrequency
Multi-Frequency with Outlier Removal	eMultiFrequencyWithOutlierRemoval

- **Standard conversion**, in which each time cycle is measured once. This is the normal conversion style that should be used for most applications.
- **Multi-frequency conversion**, in which each time cycle is measured at 4 frequencies and the composite result is the average of the 4 frequencies.
- **Multi-frequency conversion with outlier removal**, in which each time cycle is measured at 4 frequencies and the composite result is the average of the 4 frequencies after the largest outlier is removed from the data set. This is useful for mutual capacitance sensors that require conducted noise immunity.

#### Oversampling

The oversampling style control allows for the addition of oversampling in binary steps to a conversion. The available options are listed below:

Option	tOversamplingStyle Enumeration
No Oversampling	eNoOversampling
Double	e2xOversampling
Quadruple	e4xOversampling
8x	e8xOversampling
16x	e16xOversampling
32x	e32xOversampling

When no oversampling is applied, each time cycle is sampled once and that value is used as the conversion result. When a level of oversampling is applied, each time cycle is sampled to the level of oversampling, and the results are averaged. This enables a basic averaging filter that helps with transient noise rejection.

Note that the measurement time increases 2x with each step.

---

## Low Power Mode

The low power mode may be LPM0 to LPM3. The function returns when all measurements and processing are complete.

### 9.6.2.2 Calibration Algorithms

Sensors are calibrated with a top-level calibration call to **CAPT\_calibrateSensor()** or **CAPT\_calibrateSensorWithEMC()**. These two functions in turn call the two low-level calibration routines:

- **CAPT\_calibrateGain()**, for establishing each element's coarse gain and fine gain based upon the *ui16ConversionGain\*\* parameter of the sensor being calibrated*
- **CAPT\_calibrateOffset()**, for establishing each element's offset subtraction based upon the previously calibrated coarse gain and fine gain values, and the sensor's *\*ui16ConversionCount* parameter.

Functionally, calibration process is a two-step process:

1. First, **CAPT\_calibrateGain()** is called. The coarse gain and fine gain is adjusted for each element in the sensor until the conversion result for each element is as close to the specified ui16ConversionGain parameter as possible.
2. Second, **CAPT\_calibrateOffset()** is called. The offset subtraction is increased until the conversion result for each element is as close to the specified ui16ConversionCount parameter as possible.

At the end of the calibration process, all elements should have conversion results that are normalized to the ui16ConversionCount parameter.

### 9.6.2.3 Environmental Drift Algorithms

Capacitive sensing measurement results will drift over time in response to environmental changes such as temperature and humidity. Humidity effects dielectric properties (specifically, the dielectric of air). Large temperature changes can affect on-chip circuitry by changing the resistance of a pathway in a circuit, or causing an oscillator frequency to drift. This is partially why capacitive sensing is a relative measurement and not an absolute measurement. A change in temperature, humidity, or both can appear to the system as a touch if not properly interpreted. In order to distinguish an environmental change from a touch, it is necessary to examine the rate of the change. A touch event occurs more quickly than a temperature drift in most applications, and the two changes may be distinguished from each other on that basis.

To ensure reliable operation, slow drift in a sensor's measurement result due to temperature or humidity is handled by the CapTlivate Software Library in 3 ways:

1. First, the [long-term-average \(LTA\)](#) tracks measurement drift associated with slow environmental changes via a slow-moving IIR filter.
2. Second, the [touch threshold](#) varies proportionally with the LTA, rather than as an absolute offset, to maintain sensitivity.
3. Third, if [runtime recalibration](#) is enabled then the system will re-calibrate if the LTA drifts outside of a window set at +/- 1/8th of the specified [conversion count](#). This re-normalizes the sensors to the specified [conversion count](#).

These three methods work together to ensure that the system behaves as designed across the lifetime of the product, even in different environments and climates.

### 9.6.3 ISR

The CapTlivate™ peripheral has a single interrupt vector with 5 possible interrupt sources. For details on the interrupts themselves, see the [auxiliary digital functions](#) section of the [technology](#) chapter.

The software library uses the peripheral interrupts to set global status flags. The ISR is designed to quickly determine the cause of an interrupt, set the appropriate status flag, and exit. Upon exit, any low power mode is cleared so that the CPU remains alive after the interrupt. This is the mechanism that is used to wake up the application.

Of the 5 interrupts that are available, 2 are used solely by the library and 3 are left up to the application.

The following flags are used by the library, and generally do not need to be tested in the application:

1. End of Conversion Interrupt (*CAPT\_END\_OF\_CONVERSION\_INTERRUPT*, *CAPT\_IV\_END\_OF\_CONVERSION*). This interrupt is triggered when a time cycle conversion is complete. The ISR sets the *g\_bEndOfConversionFlag*, which signals the library that the conversion is complete.
2. Max Count Error Interrupt (*CAPT\_MAX\_COUNT\_ERROR\_INTERRUPT*, *CAPT\_IV\_MAX\_COUNT\_ERROR*). This interrupt is triggered if a conversion exceeds the error threshold that was specified for a sensor. The ISR sets the *g\_bMaxCountErrorFlag*, which signals the library that the error limit was reached and the conversion has stopped.

The following flags are meant to be used by the application:

1. Conversion Timer Interrupt (*CAPT\_TIMER\_INTERRUPT*, *CAPT\_IV\_TIMER*). This interrupt is triggered when the CapTlivate™ interval timer has counted up to the compare register, indicating that it is time to trigger a conversion. The ISR sets the *g\_bConvTimerFlag*. It is expected that the application is configuring and monitoring this interrupt.
2. Conversion Counter Interrupt (*CAPT\_CONVERSION\_COUNTER\_INTERRUPT*, *CAPT\_IV\_CONVERSION\_COUNTER*). This interrupt is triggered when conversion counter has reached the conversion counter interrupt threshold. This mechanism allows for an interrupt to be thrown after a certain number of conversions have taken place. In a wake-on-proximity application, this can be used to periodically wake up the CPU to ensure that the application is proceeding as expected and that all values are within range. This interrupt is enabled by the **CAPT\_startWakeOnProxMode()** function, and is disabled by the **CAPT\_stopWakeOnProxMode()** function.
3. Detection Interrupt (*CAPT\_DETECTION\_INTERRUPT*, *CAPT\_IV\_DETECTION*). This interrupt is triggered when any element in a time cycle has its proximity or negative touch thresholds exceeded at the end of a conversion. This mechanism allows for the CPU to wake up due to a threshold crossing. This interrupt is enabled by the **CAPT\_startWakeOnProxMode()** function, and is disabled by the **CAPT\_stopWakeOnProxMode()** function.

### 9.6.4 Type Definitions

The type definitions file, *CAPT\_Type.h*, contains the definitions for all of the data structures that are used in the library. It is important that the data structures are not modified! The library functions in ROM as well as the pre-compiled library are dependent upon the data structure configuration being consistent.

## 9.7 Advanced Module

The advanced module serves two main purposes: It provides processing plug-ins to the base module, and it provides the top level API. The top level API is implemented by the manager module. Processing plug-ins include button processing, slider/wheel processing, and EMC processing.

---

## 9.7.1 Manager

The manager provides the top level API for the library. For details on how to use the top level API, see the [How to Use the Top Level API](#) section.

## 9.7.2 Buttons

The buttons processing plug-in is a dominant element computation. The dominant element computation compares the delta response from all elements within the sensor. The element with the highest delta response is reported as the dominant element. For details on how to use the dominant element feature, see the [How to Access the Dominant Button](#) section.

## 9.7.3 Sliders and Wheels

The slider processing plug-in provides a vector position computation to determine the location of a touch over a 1-dimensional array of elements. The same vector math is utilized for processing slider and wheel sensors. The slider is really just a special case of a wheel where the endpoints are disconnected.

### Supported Sizes

A slider or wheel sensor must be composed of at least 3 elements, but no more than 12 elements.

### Supported Resolution

The algorithm allows for up to 16 bits of resolution, although 5-10 bits is the typical use-case. Measurement results will be reported back from 0 to resolution-1.

### Slider Endpoint Trim

The slider algorithm allows for "endpoint trim" to ensure that the beginning position is true 0 and the end position is the resolution-1. For details on how the endpoint trim works, see the [trim help section](#).

## 9.7.4 EMC

The CapTlivate™ Software Library includes an EMC module in the advanced module. This module provides processing plug-ins to the touch layer to enhance robustness in the presence of noise. This section discusses how to configure that module. For a detailed noise immunity design guide, visit the [noise immunity](#) section of the design chapter.

### 9.7.4.1 EMC Module Background

While the CapTlivate™ peripheral provides a significant feature set for dealing with electromagnetic compatibility issues on its own, some amount of digital signal processing is still required to process the raw data into usable values. The EMC module in the CapTlivate™ Software Library fills that need by providing configurable algorithms to the base touch layer of the software library. It is not necessary to call EMC processing functions directly; rather, they are automatically called by the touch layer. The various EMC features are enabled, disabled, and configured via an EMC configuration structure.

The EMC module provides the following feature set:

1. **Multi Frequency Processing (MFP) Algorithm** for resolving a raw measurement set of 4 frequencies into a single, usable measurement result

- 
2. **Multi Frequency Calibration Algorithm** for ensuring that accurate, usable calibration values are obtained during the sensor calibration process, even in a noisy environment
  3. **Oversampling (Averaging) Filter** to improve SNR
  4. **Jitter Filter** to remove small 1-count glitches
  5. **Global Relative Noise Level Tracking** to keep a global value of the noise level observed on all self-capacitance elements in the system for use in dynamic threshold adjustment
  6. **Dynamic Threshold Adjustment (DTA) Algorithm** for calculating a threshold adjustment factor to compensate for increased sensitivity of self-capacitance sensors in the presence of noise

#### 9.7.4.2 Using the EMC Module

When using the CapTlivate™ Software Library, the EMC module functions are not called by the application. Rather, noise immunity is enabled for a user configuration at a top level. When noise immunity is enabled in the library for a design, the top level library functions for calibration and sensor measurement are replaced with EMC versions of the same functions.

##### 9.7.4.2.1 Enabling Noise Immunity (EMC) Features

It is best to enable noise immunity via the CapTlivate™ Design Center. The controller customizer has a compile-time option for [noise immunity](#). Selecting this option sets the **CAPT\_CONDUCTED\_NOISE\_IMMUNITY\_ENABLE** compile-time definition in the CAPT\_UserConfig.h file to **true** when source code is generated.

##### 9.7.4.2.2 Function Replacements with Noise Immunity Enabled

When the compile-time option is set, the manager layer will make calls to EMC versions of functions rather than the standard versions. Below is a mapping of which functions are replaced:

###### Top Level Functions

Description	Standard Function	EMC Function
Calibrate a Sensor	CAPT_calibrateSensor()	CAPT_calibrateSensorWithEMC()
Update a Sensor	CAPT_updateSensor()	CAPT_updateSensorWithEMC()

These top level functions are called by the application via abstractions in CAPT\_Manager.

###### Supporting Functions

Description	Standard Function	EMC Function
Process a Cycle	CAPT_processFSMCycle()	CAPT_processCycleWithEMC()
Update Prox/Touch Status	CAPT_updateProx, CAPT_updateTouch	CAPT_updateSelfElementProxTouchWithEMC(), CAPT_updateProjElementProxTouchWithEMC()

These functions are called inside the touch layer, and are not directly called by the application.

##### 9.7.4.2.3 EMC Module Configuration

The EMC Module is configured through the tEMCConfig structure. The EMC layer only reads from this structure, so the configuration may be kept in non-volatile read-only memory if desired. A default configuration is provided in the user configuration file, and is shown below:

```
const tEMCConfig g_EMCCConfig =
{
    // Conversion Style
    .selfModeConversionStyle = eMultiFrequency,
    .projModeConversionStyle = eMultiFrequencyWithOutlierRemoval,

    // Oversampling Style
    .selfModeOversamplingStyle = eNoOversampling,
```

---

```

.projModeOversamplingStyle = eNoOversampling,
// Jitter Filter Enable
.bJitterFilterEnable = true,
// Noise Thresholds and Calibration Noise Limits
.ui8NoiseThreshold = 20,
.ui16CalibrationNoiseLimit = 10,
.ui8CalibrationTestSampleSize = 8,
// Dynamic Threshold Adjustment Parameters
.bEnableDynamicThresholdAdjustment = true,
.ui8MaxRelThreshAdj = 76,
.ui8NoiseLevelFilterEntryThresh = 40,
.ui8NoiseLevelFilterExitThresh = 0,
.ui8NoiseLevelFilterDown = 6,
.ui8NoiseLevelFilterUp = 1,
.coeffA = _IQ31(0.0065),
.coeffB = _IQ31(0.050)
};


```

The default values were selected by bench characterization and have proven effective for several different sensing panels. However, for certain applications and/or certain noise environments, it may be necessary to adjust some of the parameters. To implement a custom configuration, simply create a new tEMCConfig structure with the desired values, and pass its address to CAPT\_loadEMCConfig() when the application is initialized at start-up. Note that the CapTlivate™ starter project makes this call in CAPT\_appStart() just before CAPT\_calibrateUI().

```
CAPT_loadEMCConfig(&g_EMConfig);
```

The configuration parameters can be grouped into 5 different categories:

- Conversion Style Control
- Oversampling Style Control
- Jitter Filter Control
- Noise Thresholds and Calibration Noise Limits
- Dynamic Threshold Adjustment Parameters

Each group will be discussed in detail below.

#### Conversion Style Control

```
// Conversion Style
.selfModeConversionStyle = eMultiFrequency,
.projModeConversionStyle = eMultiFrequencyWithOutlierRemoval,
```

The conversion style control influences the type of conversion used by EMC sensor update functions. Conversion style is specified separately for self and mutual capacitance sensors, enabling different algorithms to be applied to designs that have both self and mutual sensors. There are three possible conversion styles:

- **Standard conversion**, in which each time cycle is measured once.
- **Multi-frequency conversion**, in which each time cycle is measured at 4 frequencies and the composite result is the average of the 4 frequencies
- **Multi-frequency conversion with outlier removal**, in which each time cycle is measured at 4 frequencies and the composite result is the average of the 4 frequencies after the largest outlier is removed from the data set.

Conversion style is a data type that may be passed to the **CAPT\_updateSensorRawCount()** function, which is what the EMC sensor update functions use to measure sensors. The enumeration options are shown below.

Option	tRawConversionStyle Enumeration
Standard	eStandard
Multi-Frequency	eMultiFrequency
Multi-Frequency with Outlier Removal	eMultiFrequencyWithOutlierRemoval

For mutual (projected) capacitance sensors, the recommended style is multi-frequency with outlier removal. The narrow-band susceptibility of mutual capacitance sensors suites them well to this approach. If noise exists at one of the conversion frequencies, that outlying sample is removed and the composite result is re-calculated with the remaining values.

For self-capacitance sensors, the recommended style is multi-frequency if a low-value series impedance is used (<50k-ohm), and standard if a high-value series impedance is used (>50k-ohm). For the higher series impedance approach, a hardware low-pass filter is formed by the electrode/pin capacitance and the series impedance, attenuating noise. For the low-value series impedance approach, a multi-frequency conversion provides 4 data points from which a spread can be calculated and use as a noise level reference. That reference can then be used as an input to the dynamic threshold adjustment algorithm.

#### Oversampling Style Control

```
// Oversampling Style
.selfModeOversamplingStyle = eNoOversampling,
.projModeOversamplingStyle = eNoOversampling,
```

The oversampling style control allows for the addition of oversampling in binary steps to a conversion.

Oversampling style is a data type that may be passed to the **CAPT\_updateSensorRawCount()** function, which is what the EMC sensor update functions use to measure sensors.

Option	tOversamplingStyle Enumeration
No Oversampling	eNoOversampling
Double	e2xOversampling
Quadruple	e4xOversampling
8x	e8xOversampling
16x	e16xOversampling
32x	e32xOversampling

When no oversampling is applied, each time cycle is sampled once and that value is used as the conversion result. When a level of oversampling is applied, each time cycle is sampled to the level of oversampling, and the results are averaged. This enables a basic averaging filter that helps with transient noise rejection.

Note that this oversampling is in addition to the multi-frequency scanning. For example, a mutual capacitance sensor with a multi-frequency conversion style and a 4x oversampling style is actually measured 16 times per update- 4 frequencies per sample, and a 4x oversample.

For designs that have a smaller number of buttons, more oversampling can be applied, which improves the overall SNR. Oversampling style is specified separately for self and mutual capacitance sensors.

#### Jitter Filter Control

```
// Jitter Filter Enable
.bJitterFilterEnable = true,
```

A basic 1-level jitter filter may be applied when sensors are updated with EMC features enabled. The jitter filter has a simple control (on or off). The filter looks at each new sample and determines if it is greater or less than the previous sample. If it is greater, the new sample is decremented by a value of 1. if it is less, the new sample is incremented by 1. This reduces low-level jitter in measurements, improving SNR. It is recommended that the jitter filter be enabled in most applications.

#### Noise Level Thresholds and Calibration Noise Limits

```
// Noise Thresholds and Calibration Noise Limits
.ui8NoiseThreshold = 20,
.ui16CalibrationNoiseLimit = 10,
.ui8CalibrationTestSampleSize = 8,
```

These parameters primarily exist to enable alerting of the application to the fact that the amount of noise observed in the system is greater than a specified amount. In addition, they aid in calibration if noise is present during a

---

calibration.

#### Noise Threshold

The *ui8NoiseThreshold* parameter specifies the relative noise level beyond which an element's noise detected status flag should be set. This provides a mechanism to alert the application that there is a certain amount of noise present in the measurement. The parameter is specified as a *relative* value, not an absolute value. The value is defined as a percentage of the long term average (LTA), in which 0=0% and 128=100%. The absolute noise threshold would be calculated as the relative threshold multiplied by the LTA and divided by 128, as shown below.

$$\text{absolute} = \frac{\text{relative} * \text{reference}}{128}$$
$$\text{thresh}_{\text{abs}} = \frac{\text{thresh}_{\text{rel}} * \text{LTA}}{128}$$

Figure 9.14: Relative/Absolute Noise Threshold Conversion

Relative thresholds are used here so that they can be applied to an application that may have many sensors with different conversion count settings.

Note that this noise threshold only serves the purpose of setting the noise detected flag- it does not impact the processing of the library in any way.

#### Calibration in Noisy Environments

The *ui16CalibrationNoiseLimit* and *ui8CalibrationTestSampleSize* parameters are used to test the results of the calibration process when multi-frequency scanning is enabled. If the MCU powers up in an environment with noise at one of the conversion frequencies, it is possible that the calibration algorithm may produce invalid calibration values at that frequency. To determine if a calibration value may be corrupt, after the calibration process is complete each element is sampled *ui8CalibrationTestSampleSize* times. Out of that sample set, the peak-to-peak difference at each frequency in the set is compared with the *ui16CalibrationNoiseLimit*. If the peak-to-peak variation of the measurement results at a given conversion frequency is greater than the *ui16CalibrationNoiseLimit* parameter, that frequency's calibration values are marked as invalid. When this happens, they are replaced with the calibration values of the nearest valid conversion frequency.

There are 3 possible outcomes from the multi-frequency calibration test:

1. All 4 conversion frequencies provided data that was noise-free. Thus, they retain their original calibration values.
2. 1 to 3 conversion frequencies provided data that was noisy. the noisy frequencies have their calibration values replaced with the values from the nearest conversion frequency with valid, noise-free data.
3. All 4 conversion frequencies provided data that was noisy. In this scenario, there is no way to determine if the calibration values are valid or not. To alert the application, the sensor's *bCalibrationError* status flag and *bSensorNoiseState* status flag are set. Applications should test these flags after the calibration process to determine if a usable calibration solution was obtained. If a valid calibration solution was not obtained, the application should stall and re-attempt the calibration process until a valid calibration is obtained.

#### Dynamic Threshold Adjustment Parameters

```
// Dynamic Threshold Adjustment Parameters
.bEnableDynamicThresholdAdjustment = true,
.ui8MaxRelThreshAdj = 76,
.ui8NoiseLevelFilterEntryThresh = 40,
.ui8NoiseLevelFilterExitThresh = 0,
.ui8NoiseLevelFilterDown = 6,
.ui8NoiseLevelFilterUp = 1,
.coeffA = _IQ31(0.0065),
.coeffB = _IQ31(0.050)
```

The dynamic threshold adjustment (DTA) parameters enable and configure the DTA algorithm. The parameters are introduced below:

Member	Description	Default Value	Valid Values
bEnableDynamicThresholdAdjustment	Enable or disable dynamic threshold adjustment. Note that dynamic threshold adjustment only applies to self capacitance sensors in either case.	true	true, false
ui8NoiseLevelFilterEntryThreshold	If the noise level is increasing (low to high vector) and the new noise sample is below this value, the global and local value filters will be disabled to allow for rapid tracking. A value of '0' keeps the filters enabled at all times.	32	0-128
ui8NoiseLevelFilterExitThreshold	If the noise level is decreasing (high to low vector) and the new noise sample is below this value, the global and local value filters will be disabled to allow for rapid tracking. A value of '0' keeps the filters enabled at all times.	0	0-128
ui8NoiseLevelFilterDown	The filter beta applied to the global filtered noise value when the new noise sample is lower than the filtered noise value.	5	0-15
ui8NoiseLevelFilterUp	The filter beta applied to the global filtered noise value when the new noise sample is higher than the filtered noise value.	1	0-15
coeffA	The 'A' coefficient in the dynamic threshold adjustment algorithm calculation.	0.0065	0-0.999999999
coeffB	The 'B' coefficient in the dynamic threshold adjustment algorithm calculation.	0.0100	0-0.999999999

The *bEnableDynamicThresholdAdjustment* parameter enables and disables the DTA algorithm. The DTA algorithm only applies to self capacitance sensors. See the [DTA](#) section for an overview of how the DTA algorithm works.

#### 9.7.4.3 EMC Module Algorithms

#### 9.7.4.3.1 Multi Frequency Processing (MFP) Algorithm

The multi frequency processing algorithm is implemented in the **CAPT\_resolveMultiFreqSet()** function. When noise immunity is enabled, each element is measured at four different conversion frequencies to gather more data in the presence of noise. The algorithm is then applied to the four raw measurements. The output of the algorithm is a single, composite measurement and a noise level. The composite measurement is then used by the higher levels of the library just like a raw sample normally would. The noise level is used to update each element's filtered noise level.

#### 9.7.4.3.2 Dynamic Threshold Adjustment (DTA) Algorithm

The dynamic threshold adjustment (DTA) algorithm is implemented in the **CAPT\_computeRelativeNoiseComp()** function. The algorithm calculates threshold adjustments based on the amount of noise seen in a history of measurements. It relies on a filtered relative noise value as an input, and it calculates the corresponding threshold adjustment to be applied for proximity and touch detection. The threshold adjustment is calculated based on a polynomial model as shown below, where 'x' is the relative noise value and 'y' is the corresponding relative threshold adjustment. The polynomial allows for greater adjustment at higher noise levels.

$$thresh_{adj\_rel} = A * noise_{rel}^2 + B * noise_{rel}$$

Figure 9.15: DTA Formula

This formula with the default values provides the adjustment curve shown below. A linear ( $A=0$ ;  $B=0.5$ ) curve is also shown for reference.

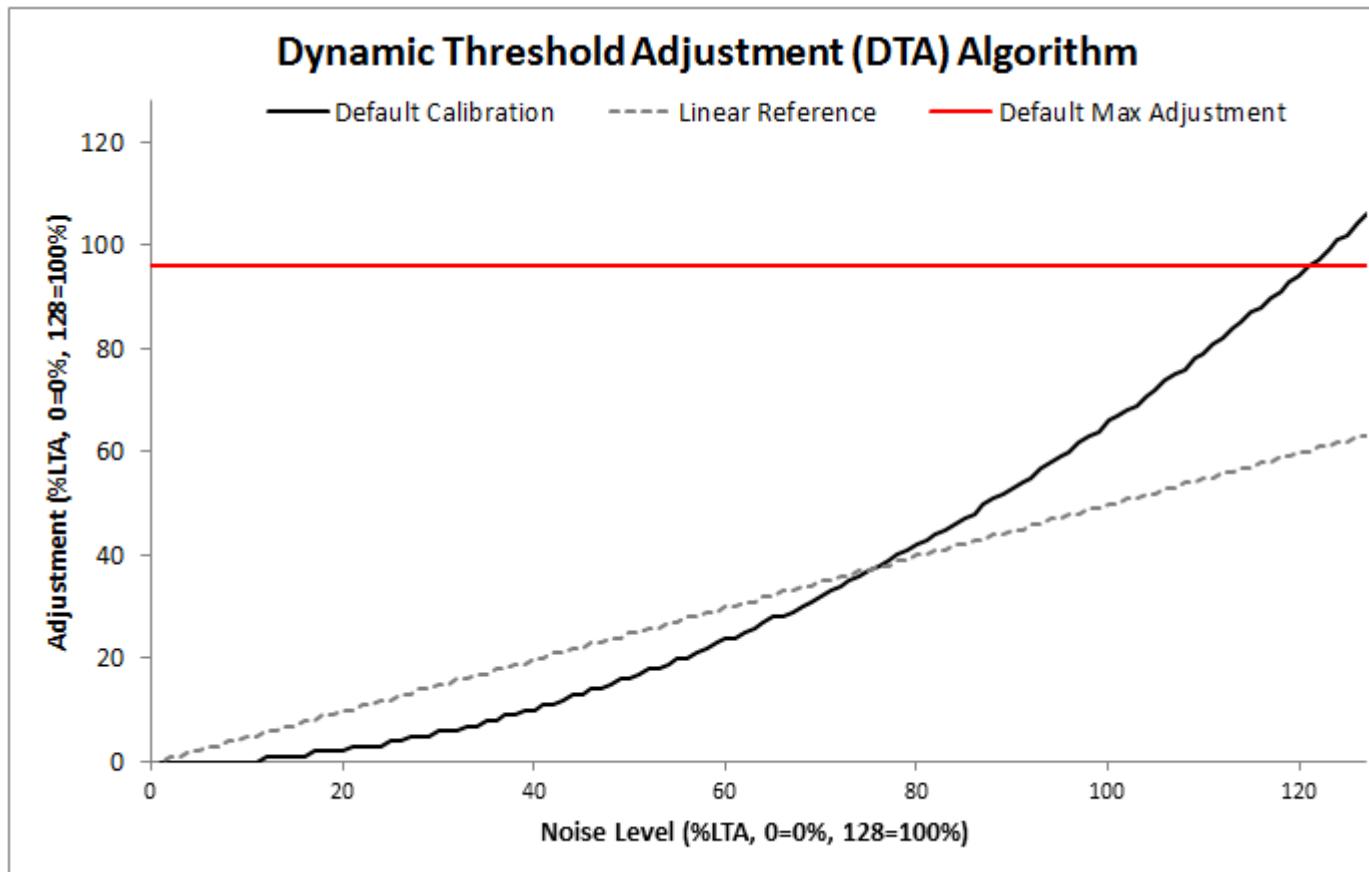


Figure 9.16: DTA Default Value Response Curve

---

## 9.8 Communications Module

The CapTlivate™ Touch Library includes a communications module for connecting CapTlivate™ MCUs to the outside world. This section discusses the architecture, features, and specification for that communications module, as well as how it may be used in a variety of applications from development to production.

This section assumes that the reader is familiar with the following:

- Capacitive sensing as a user interface technology
- The Captivate ecosystem and touch library
- Basic MSP microcontroller architecture

### 9.8.1 Background

Designing a capacitive touch interface is an iterative process. Every sensor in the system must be individually tuned and optimized to achieve the desired sensitivity and "feel." Having the ability to communicate in real-time between a PC GUI and the target MCU drastically reduces the amount of time needed to tune an interface, and can provide better tuning results as the features incorporated into the CapTlivate™ peripheral can quickly and easily be exercised to determine the best configuration.

Following the design and tuning phase, a capacitive touch microcontroller takes on one of two roles in a system. It may be a dedicated human-machine interface (HMI) that only serves to resolve a capacitive touch panel into usable information (like touch/no touch, or a slider position), or it may double as a host processor, integrating other functionality such as monitoring sensors or controlling other functions in the system. In the first case (the dedicated HMI case), the controller will almost always require a way to communicate the status of the interface to some other host, which may be another MCU or an MPU. This interface could be as simple as a GPIO that gets set when a button is pressed, or as complex as a full I2C protocol with addressable parameters.

### 9.8.2 Overview

The CapTlivate™ Software Library communications module provides a single solution to the two needs above. The communications module is a layered set of firmware with a simple top-level API, designed to link a CapTlivate™ MCU to the CapTlivate™ Design Center PC GUI or to a host processor of some kind via a standard, common serial interface.

In a capacitive touch application the MCU is responsible for measuring capacitive sensors, processing the measurement to interpret some kind of result, and transmitting that result either to the application locally or to a host processor. The communications layer provides the "transmission" part of the equation. The diagram below portrays how the communications module fits in with the rest of the firmware in a typical CapTlivate™ application.

The CapTlivate™ communications module is layered and contains several standalone features that are interlinked together. These features are introduced below. To use the communications module, it is only necessary to set up the configuration file and call the top-level APIs.

The communications module contains 4 layers. In order of decreasing abstraction, they are:

- [Interface Layer](#) [COMM/CAPT\_Interface.c/h]
  - The interface layer implements the top-level API.
- [Protocol Layer](#) [COMM/CAPT\_Protocol.c/h]
  - The protocol layer implements Captivate protocol packet generation and packet interpretation.
- Serial Driver Layer [COMM/Serial\_Drivers/\*.c/h]
  - The serial driver layer contains several interchangeable serial driver options.
  - One and only one option may be selected at any given time.

- Selection and configuration is handled in the configuration file.
- The serial drivers are built on top of the MSP430 DriverLib API.
- **UART** and **I2C Slave** drivers are provided.
- Data Structure Layer [COMM/CAPT\_ByteQueue.c/.h, COMM/CAPT\_PingPongBuffer.c/.h]
  - The data structure layer implements basic abstract data types, such as a FIFO queue and a ping pong buffer to aid serial communication.

### 9.8.3 Interface Layer

The CapTlivate™ interface layer is the top-level communication layer. It provides the top-level function calls that are used by the application, and serves to marry together the protocol layer (which handles packet generation and interpretation) with the serial driver (which actually moves the data in and out of the microcontroller). The functionality provided is covered in this section below. All application access to the communications module should be through the interface layer.

#### 9.8.3.1 Using the Communications Module: Initializing the Interface

The communications module must be initialized at startup by the application via a call to **CAPT\_initCommInterface()**. This top-level init function handles opening the selected serial driver, as well as initializing any queues/buffers that are needed for communication.

Description	Declaration
Init the Communications Module	extern void CAPT_initCommInterface(tCaptivateApplication *pApp)

#### 9.8.3.2 Using the Communications Module: Handling Incoming Data

Incoming raw data from a host is buffered by the serial driver to be serviced when the application is available to do so. The application must periodically call **CAPT\_checkForInboundPacket()** to check to see if any packets have been received from the host. This top-level function will check for packets in the receive queue of the serial driver, and if any packets are found, they will be processed according to their type. Typically, this function is called in a background loop when the application is available. Note that the serial drivers will exit active from sleep if a data is arriving from the host, which can be used as a mechanism to wake up the background loop to call this function.

The **CAPT\_checkForRecalibrationRequest()** function should also be called periodically to see if any of the packets received and handled by **CAPT\_checkForInboundPacket()** require the application to re-calibrate the user interface. An example of this would be if a packet was received that changed the conversion count, requiring a re-calibration of the sensors in the system.

Description	Declaration
Check for an Inbound Packet, and Process It	extern bool CAPT_checkForInboundPacket(void)
Check for a Re-calibration Request	extern bool CAPT_checkForRecalibrationRequest(void)

#### 9.8.3.3 Using the Communications Module: Writing Out Data

The interface layer provides 3 top-level constructs for transmitting data to the host. The three functions below handle generation of the appropriate packet, management of the transmit ping/pong buffers, and transmission over the serial interface. If the serial peripheral is available (not busy) these calls are non-blocking, and the packets that are generated are transmitted to the host via interrupt service routines in the serial driver.

Description	Declaration
Write Element Data	extern bool CAPT_writeElementData(uint8_t ui8SensorID)
Write Sensor Data	extern bool CAPT_writeSensorData(uint8_t ui8SensorID)
Write General Purpose Data	extern bool CAPT_writeGeneralPurposeData(uint16_t *pData, uint8_t ui8Cnt)

#### 9.8.3.4 Compile-Time Configuration

The compile-time configuration options are set in the CAPT\_CommConfig.h file. The available compile-time options are described below.

##### Interface Selection Definition

Parameter	File	Valid Values
CAPT_INTERFACE	CAPT_UserConfig.h	__CAPT_NO_INTERFACE__, CAPT_UART_INTERFACE, CAPT_BULKI2C_INTERFACE, CAPT_REGISTERI2C_INTERFACE

CAPT\_INTERFACE, unlike the remaining definitions, is located in the User Config file (CAPT\_UserConfig.h). It selects the interface that the communications module should be built for. If the communication module should be excluded from the build, then **CAPT\_NO\_INTERFACE** should be set. Otherwise, the desired communication mode should be set.

NOTE: This value is automatically populated in the CAPT\_UserConfig.h file by the Design Center during source code generation.

##### Transmit Buffer Size Definition

Parameter	File	Valid Values
CAPT_TRANSMIT_BUFFER_SIZE	CAPT_CommConfig.h	Unsigned Integer

CAPT\_TRANSMIT\_BUFFER\_SIZE defines the size of the transmit buffer. Note that 2x this size will be allocated, since ping-pong buffering is used. This size should also be at least 2x the size of the largest packet, to allow for byte stuffing.

##### Receive Queue Buffer Size Definition

Parameter	File	Valid Values
CAPT_QUEUE_BUFFER_SIZE	CAPT_CommConfig.h	Unsigned Integer

CAPT\_QUEUE\_BUFFER\_SIZE defines the size of the receive queue. This is the queue that the serial driver uses to buffer received data until the data is processed by a call to **CAPT\_checkForInboundPacket()**. If it seems like packets are being dropped, a good first step is to increase the size of this buffer.

##### I2C Slave Serial Driver Receive Buffer Size Definition

Parameter	File	Valid Values
CAPT_I2C_RECEIVE_BUFFER_SIZE	CAPT_CommConfig.h	Unsigned Integer

CAPT\_I2C\_RECEIVE\_BUFFER\_SIZE defines the size of the receive buffer used by the I2C Slave driver, if that driver is selected. This buffer size should be at least as large as the maximum length I2C bus write transaction that is expected.

##### I2C Slave Serial Driver Buffer Size in Register Mode Definition

Parameter	File	Valid Values
CAPT_I2C_REGISTER_RW_BUFFER_SIZE	CAPT_CommConfig.h	Unsigned Integer

CAPT\_I2C\_REGISTER\_RW\_BUFFER\_SIZE defines the size of the buffer used by the I2C Slave driver when the communication interface is configured in register I2C mode. This buffer size should be at least as large as the maximum length I2C bus transaction that is expected.

## 9.8.4 Protocol Layer

The CapTlivate™ protocol is a communications specification for sending capacitive touch specific data. It enables MSP430 Captivate-equipped microcontrollers to communicate with design, debug, and tuning tools on host PCs. In addition to this function, it can also provide a mechanism for interfacing a Captivate MCU to another host MCU or SoC in the context of a larger system. This guide discusses the details of the protocol itself: packet types and packet structure.

The Captivate protocol is a packet-based serial messaging protocol. It includes provisions for passing real-time capacitive measurement data from a Captivate target MCU to another processor, as well as provisions for tuning parameter read and write.

### Use Cases

The various use cases for the protocol are described below.

1. Capacitive Touch Development and Tuning Capacitive touch development and tuning involves looking at real-time data from a touch panel and adjusting software parameters to achieve the desired response and feel from the sensors on that panel. For example: tuning a capacitive button involves looking at the raw data coming back from the microcontroller about that button, and adjusting thresholds, de-bounce, and filters accordingly to create a robust user interface. Having the ability to adjust all of these software parameters in real-time while looking at sensor data, without re-compiling code, is extremely powerful and reduces development time. The Captivate protocol was designed with the Captivate Design Center specifically to meet this need.
2. Interface to Host Processor Most capacitive touch user interfaces involve a dedicated microcontroller driving the touch panel, which communicates up to a host processor of some kind. The flexibility of the Captivate protocol allows for it to be re-used as an interface to a host processor; it can stream touch status, proximity status, and slider/wheel position up to a host.
3. In-Field / In-System Debug Interface and Tuning Since the Captivate protocol supports reading and writing of capacitive touch tuning parameters, as well as the streaming of real-time data, it could potentially be utilized as a diagnostic tool in the field when coupled with the Captivate Design Center PC tool.

### 9.8.4.1 Introduction to Packet Types

The Captivate protocol supports five packet types: sensor packets, cycle packets, parameter packets, general purpose packets, and trackpad packets. In a capacitive touch system, there are two endpoints in the communication link: the target MCU itself, and the host. The host might be a PC tool or some kind of embedded processor. Sensor packets, cycle packets, general purpose packets, and trackpad packets carry information about the current state of the touch panel being driven by the target MCU. These packets are UNIDIRECTIONAL, and only travel from the target to the host. Parameter packets are BIDIRECTIONAL, and may travel from the target to the host or from the host to the target.

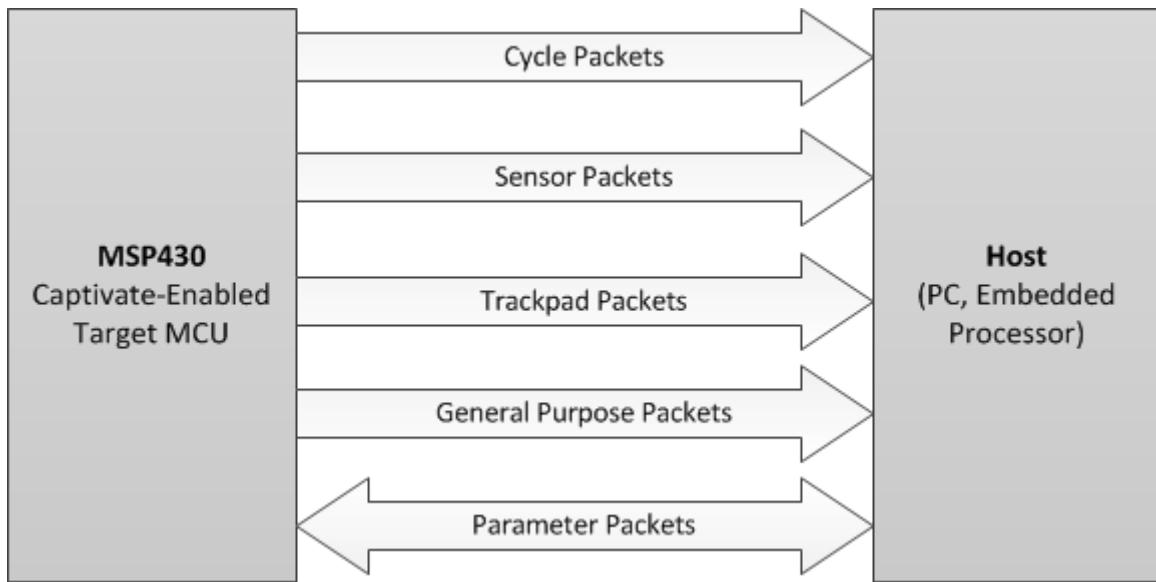


Figure 9.17: Packet Directionality

#### Sensor Packets

**Sensor packets** are unidirectional packets from the Captivate MCU to the host. They provide information about the current state of a sensor. Sensor state information includes things like dominant button, slider or wheel position, sensor global proximity state, and sensor global touch/previous touch state.

#### Cycle Packets

**Cycle packets** are unidirectional packets from the Captivate MCU to the host. They provide low level element information, such as element touch status, element proximity status, element count, and element long term average for all of the elements within a cycle. These packets are typically used in the tuning phase, where it is desirable to have real-time views of count and long term average for setting thresholds and tuning filters.

#### Trackpad Packets

**Trackpad packets** are unidirectional packets from trackpad MCUs to the host. They provide the X and Y coordinates of touches on the trackpad.

#### General Purpose Packets

**General purpose packets** are unidirectional packets from a Captivate MCU to the host. They serve as a generic container to send any information that can be formatted as a 16-bit unsigned integer. This channel can serve as a debug tool for sending any kind of information that doesn't fit into any of the other packet types. Up to 29 integers (58 bytes) may be sent in a single packet.

#### Parameter Packets

**Parameter packets** are bi-directional packets between a host and the Captivate MCU. **Parameter packets** allow for the host to adjust a tuning parameter on the target at runtime. For example, the touch threshold for an element or the resolution of a slider could be adjusted by sending the appropriate parameter command. Parameters can be read or written. Parameter reads and writes from a host to a target MCU always result in a read-back of the most current value (the value after the write, in the case of a write).

#### Transmission Rules

The packet types discussed above are transmitted via a serial interface of some kind between the target and the host. Full-duplex UART is the typical interface. To provide reliable and accurate packet transmission, a set of

transmission rules is applied to all packets when being transmitted. These rules are discussed in the [HID Bridge Packet Mode](#) section.

The following functions aid in applying transmission rules:

Description	Declaration
Stuff Sync Bytes in a Packet	extern uint16_t CAPT_stuffSyncBytes(uint8_t *pBuffer, uint16_t ui16Length)
Verify a Checksum	extern bool CAPT_verifyChecksum(const uint8_t *pBuffer, const uint16_t ui16Length, const uint16_t ui16Checksum)
Get a Checksum	extern uint16_t CAPT_getChecksum(const uint8_t *pBuffer, const uint16_t ui16Length)
Identify and Frame a Packet in a Receive Data Queue	extern bool CAPT_processReceivedData(tByteQueue *pReceiveQueue, tParameterPacket *pPacket, tTLProtocolProcessingVariables *pVariables)

#### 9.8.4.2 Format: Sensor Packets

Sensor packets have a fixed length of 6 bytes. There are two control bytes and four data payload bytes.

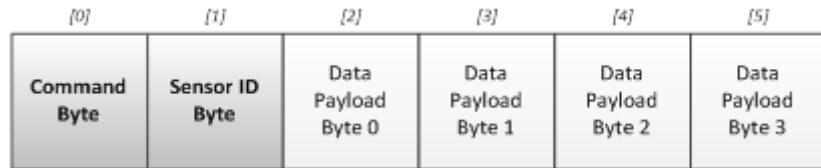


Figure 9.18: Sensor Packet Format

- 1. Command Byte [0]** The command byte for a sensor packet is always 00h.
- 2. Sensor ID Byte [1]** The sensor ID byte contains an unsigned 8-bit integer that specifies the ID of the sensor whose data is being transmitted. Sensor ID on the target side is typically established by the order of sensor pointers in the global sensor pointer array. Sensors are sorted alphabetically by the Captivate Design Center PC GUI.
- 3. Data Payload Bytes [2-5]** The data payload bytes contain the sensor data that is being sent. The information contained in the payload is dependent upon the sensor type. The table below describes the payload on a sensor type basis.

Sensor Type	Byte 0	Byte 1	Byte 2	Byte 3
Button Group	Dominant Element (256 elements max)	Previous Dominant Element (256 elements max)	Reserved	Sensor Status
Slider	Slider Position (Lower 8 bits of 16 bits)	Slider Position (Upper 8 bits of 16 bits)	Reserved	Sensor Status
Wheel	Wheel Position (Lower 8 bits of 16 bits)	Wheel Position (Upper 8 bits of 16 bits)	Reserved	Sensor Status
Proximity	Reserved	Reserved	Reserved	Sensor Status
Trackpad	Reserved	Reserved	Reserved	Sensor Status

The sensor status byte, included in button group, slider, and wheel sensor packets, provides additional data about the state of the sensor that is often meaningful. The status flags are all boolean flags, and are assigned to bit positions as follows:

Bit Mask (Position)	Sensor Status Flag
Bit 0 (01h)	Global Sensor Touch Flag
Bit 1 (02h)	Global Sensor Previous Touch Flag
Bit 2 (04h)	Global Sensor Proximity Flag
Bit 3 (08h)	Global Sensor Detect Flag (Prox Detect Pre-Debounce)
Bit 4 (10h)	Global Sensor Negative Touch Flag (Reverse Touch)
Bit 5 (20h)	Global Sensor Noise State
Bit 6 (40h)	Global Sensor Max Count Error Flag
Bit 7 (80h)	Global Sensor Calibration Error Flag

**NOTE:** For slider / wheel sensors, the 16 position bits in the data payload will contain the valid slider or wheel position when the global sensor touch flag is true on that sensor. If there is not a global touch detection, the 16 position bits will all be set high (0xFFFF for the slider/wheel position value).

**NOTE:** Reserved fields are still transmitted, but do not contain any meaningful data. Do not use or rely on any data transmitted in a reserved field.

Sensor packets are generated via a call into the protocol layer. The **CAPT\_getSensorPacket()** function looks up the sensor at index ui8SensorID in the array sensorArray, and stores the generated packet in the buffer space pointed to by pBuffer. The length of the packet is returned by the function.

Description	Declaration
Get a Sensor Packet	extern uint16_t CAPT_getSensorPacket(tSensor **sensorArray, uint8_t ui8SensorID, uint8_t *pBuffer)

#### 9.8.4.3 Format: Cycle Packets

Cycle packets have a variable length which is dependent upon the number of elements within the cycle. There are always 3 control bytes and 3 state bytes. In addition to those 6 bytes, there are 4 bytes per element in the cycle.

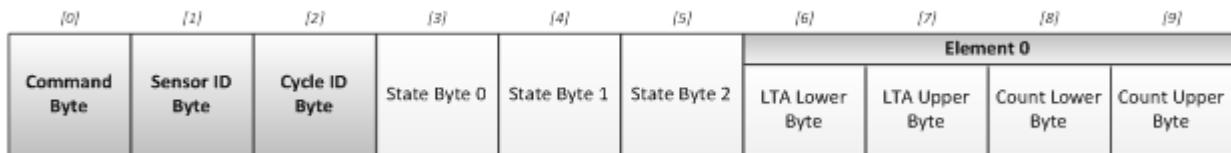


Figure 9.19: Cycle Packet Format

1. **Command Byte [0]** The command byte for a cycle packet is always 01h.
2. **Sensor ID Byte [1]** The sensor ID byte contains an unsigned 8-bit integer that specifies the ID of the sensor whose data is being transmitted. Sensor ID on the target side is typically established by the order of sensor pointers in the global sensor pointer array. Sensors are sorted alphabetically by the Captivate Design Center PC GUI.
3. **Cycle ID Byte [2]** The cycle ID byte contains an unsigned 8-bit integer that specifies the ID of the cycle whose data is being transmitted, relative to the sensor. For example, in a sensor with 3 cycles, the first cycle would have an ID of 0, the second, an ID of 1, and the third, an ID of 2. The cycle ID should correspond to the index of the cycle in the sensor's cycle pointer array.
4. **Cycle State Bytes [3-5]** The cycle state bytes specify the touch and proximity detection flags for each element in the cycle. Up to 12 elements per cycle are supported. Bytes 3 through 5 of the cycle packet comprise a 24 bit cycle state section, where the lower 12 bits represent the proximity state for each element in a bitwise fashion, and the upper 12 bits represent the touch state for each element in a bitwise fashion.
5. **Element LTA and Element Count Bytes [6-n], n=5+4(number of elements)** The remainder of the cycle packet is comprised of element LTA and count values. LTA and count are represented as 16 bit unsigned integers. As such, 32 bits are required for each element in the LTA/Count section. The packet shall increase in size in 32 bit (4 byte) increments for every additional element in the cycle, up to a maximum of 12.

---

elements. The LTA is sent first, and the count second. Per the protocol, when sending values greater than one byte, the transmission sequence is lowest order byte to highest order byte.

Cycle packets are generated via a call into the protocol layer. The **CAPT\_getCyclePacket()** function looks up the cycle at index ui8Cycle in the sensor at index ui8SensorID in the array sensorArray, and stores the generated packet in the buffer space pointed to by pBuffer. The length of the packet is returned by the function.

Description	Declaration
Get a Cycle Packet	extern uint16_t CAPT_getCyclePacket(tSensor **sensorArray, uint8_t ui8SensorID, uint8_t ui8Cycle, uint8_t *pBuffer);

#### 9.8.4.4 Format: Trackpad Packets

Trackpad packets have a variable length which is dependent upon the maximum number of simultaneous touches supported by the trackpad device. There are 3 control bytes. Following the control bytes, there is a trackpad gesture byte for indicating the detection of a gesture and what the gesture was. For each simultaneous touch the trackpad device supports, 4 additional payload bytes are added to the packet. The packet format can be seen below.

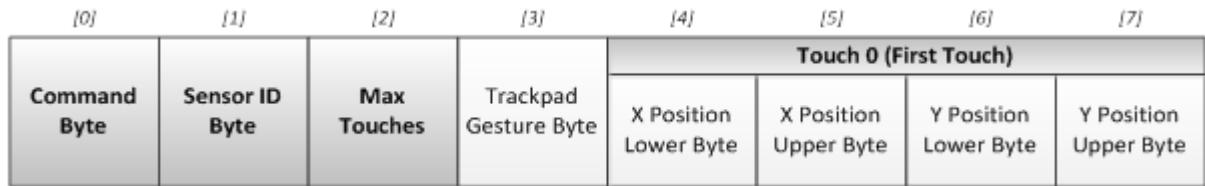


Figure 9.20: Trackpad Packet Format

- 1. Command Byte [0]** The command byte for trackpad packets is always 02h.
- 2. Sensor ID Byte [1]** The sensor ID byte contains an unsigned 8-bit integer that specifies the ID of the sensor whose data is being transmitted. Sensor ID on the target side is typically established by the order of sensor pointers in the global sensor pointer array. Sensors are sorted alphabetically by the Captivate Design Center PC GUI.
- 3. Max Touches [2]** The max touches byte contains an unsigned 8-bit integer that specifies the number of simultaneous touches supported by the trackpad device. This also provides insight into the length of the remainder of the packet. Following the max touches byte, there will be 4 bytes for each simultaneous touch supported by the trackpad device. Trackpad devices must support at least one touch.
- 4. Gesture Status [3]** The trackpad gesture byte indicates whether a gesture was just detected, and what that gesture was. The value/gesture pairs are listed below. If no gesture was detected, the gesture byte will be set to FFh.

**Value**	Gesture
00h	Wake on Proximity Detection
01h	Reserved
02h	Single Touch Single Tap
03h	Single Touch Double Tap
04h	Two Touch Single Tap
05h	Two Touch Double Tap
06h	Tap and Hold

07h	Two Touch Tap and Hold
08h	Swipe Left
09h	Swipe Right
0Ah	Swipe Up
0Bh	Swipe Down
0Ch-FEh	Reserved
FFh	No Gesture Detected

1. **Touch Coordinates [4-n], n=4+4(Max Touches)** The touch coordinates indicate the presence of and location of a touch on the trackpad. Touch locations are communicated via two 16-bit unsigned integers per touch: one for the coordinate on the X axis, and one for the coordinate on the Y axis. 16 bits is the maximum resolution in either the X or Y direction for the trackpad. Most applications will have a working resolution that is less than 16 bits per coordinate, but 16 bits are always sent regardless. If no touch is present, both the X and the Y coordinates will read as FFFFh.

**NOTE:** Trackpad packets only pertain to dedicated trackpad devices.

#### 9.8.4.5 Format: General Purpose Packets

General purpose packets are unique in that they simply serve as a data streaming mechanism for any data an application wishes to send. Data is sent as an array of 16-bit unsigned integers. General purpose packets have variable length that is dependent upon the number of integers being sent. There are always 2 control bytes. Following the control bytes is the data payload (up to 29 entries), with 2 bytes (16 bits) per entry.

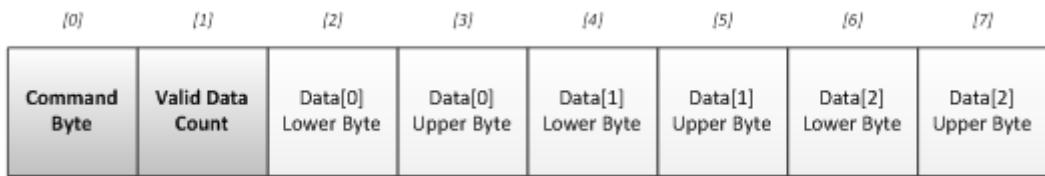


Figure 9.21: General Purpose Packet Format

- Command Byte [0]** The command byte for general purpose packets is always 10h.
- Valid Data Count Byte [1]** The valid data count byte indicates how many valid data entries exist in the packet. This also implies the length of the packet: each valid entry implies two bytes of payload (each entry is a 16-bit unsigned integer). Valid values for this field are 1-29.
- Data Payload [2-n]** The data payload section contains the generic data to be transmitted, formatted as unsigned 16-bit integers. Up to 29 payload items (58 bytes) are allowed.

General purpose packets are generated via a call into the protocol layer. The **CAPT\_getGeneralPurposePacket()** function generates a packet for the data array of length ui8Cnt pointed to by pData, and stores the generated packet in the buffer space pointed to by pBuffer.

Description	Declaration
Get a General Purpose Packet	<pre>extern uint16_t CAPT_getGeneralPurposePacket(uint16_t *pData, uint8_t ui8Cnt, uint8_t *pBuffer)</pre>

#### 9.8.4.6 Format: Parameter Packets

Parameter packets have a fixed length of 7 bytes. There are 3 control bytes and 4 data payload bytes. The packet format can be seen below.

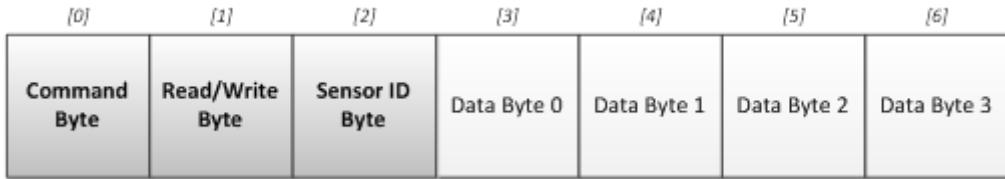


Figure 9.22: Parameter Packet Format

- Command Byte [0]** The command byte for parameter packets is variable, and indicates the ID of the parameter that is to be read or written to.
- Read/Write Byte [1]** The read/write byte is a 1-bit Boolean value that indicates whether the operation is a parameter read or a parameter write. 1 = write, 0 = read.
- Sensor ID Byte [2]** The sensor ID byte contains an unsigned 8-bit integer that specifies the ID of the sensor whose parameter is being read from or written to. Sensor ID on the target side is typically established by the order of sensor pointers in the global sensor pointer array. Sensors are sorted alphabetically by the Captivate Design Center PC tool. Note that parameters for controller commands (C-h commands) may not use the Sensor ID field.
- Data Bytes [3-6]** The data bytes contain the information that is to be read or written to. In some cases, 1 or more of the data bytes are used for further ID (such as cycle ID or element ID).

The protocol layer provides several functions for framing and interpreting parameter packets. Placing a call into **CAPT\_processReceivedData()** causes the protocol layer to search for potential packets in a datastream that has been queued up by a serial driver. Once a valid packed has been framed, the parameter may be accessed and updated/read via calls to **CAPT\_accessSensorParameter()** and **CAPT\_accessSpecialSensorParameter()**.

Description	Declaration
Access a Sensor Parameter	extern tTLPParameterAccessResult CAPT_accessSensorParameter(tSensor **sensorArray, tParameterPacket *pPacket)
Access a Special Sensor Parameter	extern tTLPParameterAccessResult CAPT_accessSpecialSensorParameter(tSensor **sensorArray, tParameterPacket *pPacket)
Identify and Frame a Packet in a Receive Data Queue	extern bool CAPT_processReceivedData(tByteQueue *pReceiveQueue, tParameterPacket *pPacket, tTLPProtocolProcessingVariables *pVariables)
Verify a Checksum	extern bool CAPT_verifyChecksum(const uint8_t *pBuffer, const uint16_t ui16Length, const uint16_t ui16Checksum)

The available parameters that can be adjusted through the use of parameter packets are listed below.

#### Sensor Parameters

Parameter Name	Byte 0: CMD	Byte 1: RW	Byte 2: ID	Byte 3	Byte 4	Byte 5	Byte 6	MCU Task	SW Containing Structure	SW Containing Variable
Conversion Gain	80	RW	Sensor ID	Don't Care	Don't Care	ATI Base Lower Byte	ATI Base Upper Byte	Re-Cal	tSensor	ui16ConversionGain

Conversion Count	81	RW	Sensor ID	Don't Care	Don't Care	ATI Target Lower Byte	ATI Target Upper Byte	Re-Cal	tSensor	ui16ConversionCount
Prox Threshold	82	RW	Sensor ID	Don't Care	Don't Care	Prox Threshold Lower Byte	Prox Threshold Upper Byte	NA	tSensor	ui16ProxThreshold
Prox Debounce-In Threshold	84	RW	Sensor ID	Don't Care	Don't Care	Prox Db In	Don't Care	NA	tSensor	ProxDThreshold.DbUp
Prox Debounce-Out Threshold	85	RW	Sensor ID	Don't Care	Don't Care	Prox Db Out	Don't Care	NA	tSensor	ProxDThreshold.DbDown
Touch Debounce-In Threshold	86	RW	Sensor ID	Don't Care	Don't Care	Touch Db In	Don't Care	NA	tSensor	TouchDbThreshold.DbUp
Touch Debounce-Out Threshold	87	RW	Sensor ID	Don't Care	Don't Care	Touch Db Out	Don't Care	NA	tSensor	TouchDbThreshold.DbDown
Sensor Timeout Threshold	88	RW	Sensor ID	Don't Care	Don't Care	Sensor Timeout Lower Byte	Sensor Timeout Upper Byte	NA	tSensor	ui16TimeoutThreshold
Count Filter Enable	89	RW	Sensor ID	Don't Care	Don't Care	Count Filter Enable bit	Don't Care	NA	tSensor	bCountFilterSelect
Count Filter Beta	8A	RW	Sensor ID	Don't Care	Don't Care	Count Filter Beta	Don't Care	NA	tSensor	ui8CntBeta
LTA Filter Beta	8B	RW	Sensor ID	Don't Care	Don't Care	LTA Filter Beta	Don't Care	NA	tSensor	ui8LTABeta
Halt LTA Filter Immediately	8C	RW	Sensor ID	Don't Care	Don't Care	LTA Filter Halt	Don't Care	NA	tSensor	bSensorHalt
Runtime Re-Calibration Enable	8D	RW	Sensor ID	Don't Care	Don't Care	Runtime Re-Cal Enable	Don't Care	NA	tSensor	bReCalibrateEnable
Force Re-Calibrate	8E	N/A	Sensor ID	Don't Care	Don't Care	Don't Care	Don't Care	Re-Cal	tSensor	N/A
Bias Current	8F	RW	Sensor ID	Don't Care	Don't Care	Bias Current	Don't Care	Re-Cal	tSensor	ui8BiasControl
Sample Capacitor Discharge	95	RW	Sensor ID	Don't Care	Don't Care	Cs Discharge	Don't Care	Re-Cal	tSensor	bCsDischarge
Modulation Enable	96	RW	Sensor ID	Don't Care	Don't Care	Mod Enable	Don't Care	Re-Cal	tSensor	bModEnable

Frequency Divider	97	RW	Sensor ID	Don't Care	Don't Care	Freq Div	Don't Care	Re-Cal	tSensor	ui8FreqDiv
Charge/Hold Phase Length	98	RW	Sensor ID	Don't Care	Don't Care	Charge Length	Don't Care	Re-Cal	tSensor	ui8ChargeLength
Transfer/Sample Phase Length	99	RW	Sensor ID	Don't Care	Don't Care	Transfer Length	Don't Care	Re-Cal	tSensor	ui8TransferLength
Error Threshold	9A	RW	Sensor ID	Don't Care	Don't Care	Error Threshold Lower Byte	Error Threshold Upper Byte	NA	tSensor	ui16ErrorThreshold
Negative Touch Threshold	9B	RW	Sensor ID	Don't Care	Don't Care	Negative Touch Threshold Lower Byte	Negative Touch Threshold Upper Byte	NA	tSensor	ui16NegativeTouchThreshold
Idle State	9C	RW	Sensor ID	Don't Care	Don't Care	Idle State	Don't Care	NA	tSensor	bIdleState
Input Sync	9D	RW	Sensor ID	Don't Care	Don't Care	Input Sync	Don't Care	NA	tSensor	ui8InputSyncControl
Timer Sync	9E	RW	Sensor ID	Don't Care	Don't Care	Timer Sync	Don't Care	NA	tSensor	bTimerSyncControl
Automatic Power-Down Enable	9F	RW	Sensor ID	Don't Care	Don't Care	Power Down Control	Don't Care	NA	tSensor	bLpmControl
Halt LTA on Sensor Prox or Touch	A0	RW	Sensor ID	Don't Care	Don't Care	Sensor Prox/Touch Halt	Don't Care	NA	tSensor	bPTSensorHalt
Halt LTA on Element Prox or Touch	A1	RW	Sensor ID	Don't Care	Don't Care	Element Prox/Touch Halt	Don't Care	NA	tSensor	bPTElementHalt

#### Element Parameters

Parameter Name	Byte 0: CMD	Byte 1: RW	Byte 2: ID	Byte 3	Byte 4	Byte 5	Byte 6	MCU Task	SW Containing Structure	SW Containing Variable
Touch Threshold	83	RW	Sensor ID	Cycle # (relative to sensor)	Lower 4 Bits: Element # (relative to cycle)	Touch Threshold	Don't Care	NA	tElement	ui8TouchThreshold [Element #]

Coarse Gain Ratio	A2	R	Sensor ID	Cycle # (relative to sensor)	[UPPER 4 Bits: Frequency #] [ LOWER 4 Bits: Element # relative to cycle]	Coarse Gain	Don't Care	NA	tCaptive	EleGainRatio	Coarse
Fine Gain Ratio	A3	R	Sensor ID	Cycle # (relative to sensor)	[UPPER 4 Bits: Frequency #] [ LOWER 4 Bits: Element # relative to cycle]	Fine Gain	Don't Care	NA	tCaptive	EleGainRatio	Fine
Parasitic Offset Scale	D0	R	Sensor ID	Cycle # (relative to sensor)	[UPPER 4 Bits: Frequency #] [ LOWER 4 Bits: Element # relative to cycle]	Offset Scale (2 bit selection)	Don't Care	NA	tCaptive	EleOffsetScale	Upper Byte
Parasitic Offset Level	D1	R	Sensor ID	Cycle # (relative to sensor)	[UPPER 4 Bits: Frequency #] [ LOWER 4 Bits: Element # relative to cycle]	Offset Level (8 bit selection)	Don't Care	NA	tCaptive	EleOffsetLevel	Lower Byte

### Slider/Wheel Parameters

Parameter Name	Byte 0: CMD	Byte 1: RW	Byte 2: ID	Byte 3	Byte 4	Byte 5	Byte 6	MCU Task	SW Containing Structure	SW Containing Variable
Slider/Wheel Position Filter Enable	90	RW	Sensor ID	Don't Care	Don't Care	Slider Filter Enable bit	Don't Care	NA	tSliderSensorData	SliderFilterEnable
Slider/Wheel Position Filter Beta	91	RW	Sensor ID	Don't Care	Don't Care	Slider Filter Beta	Don't Care	NA	tSliderSensorData	SliderBeta
Desired Slider/Wheel Resolution	92	RW	Sensor ID	Don't Care	Don't Care	Slider Resolution Lower Byte	Slider Resolution Upper Byte	NA	tSliderSensorData	SliderResolution
Slider Lower Trim	93	RW	Sensor ID	Don't Care	Don't Care	Slider Lower Trim Lower Byte	Slider Lower Trim Upper Byte	NA	tSliderSensorData	SliderLower
Slider Upper Trim	94	RW	Sensor ID	Don't Care	Don't Care	Slider Upper Trim Lower Byte	Slider Upper Trim Upper Byte	NA	tSliderSensorData	SliderUpper

### Controller/Management Parameters

Parameter Name	Byte 0: CMD	Byte 1: RW	Byte 2: ID	Byte 3	Byte 4	Byte 5	Byte 6	MCU Task	SW Containing Structure	SW Containing Variable
Element Data Transmit Enable	C0	RW	Don't Care	Don't Care	Don't Care	Element Transmit Enable	Don't Care	NA	tCaptivateApplicationDataTx	ApplDataTxEnable
Sensor Data Transmit Enable	C1	RW	Don't Care	Don't Care	Don't Care	Sensor Transmit Enable	Don't Care	NA	tCaptivateApplicationDataTx	ApplDataTxEnable
Active Mode Scan Rate (ms)	C2	RW	Don't Care	Don't Care	Don't Care	Active Report Period (ms) Lower Byte	Active Report Period (ms) Upper Byte	NA	tCaptivateApplicationModeScanPeriod	ApplModeScanPeriod

Wake-on-Prox Mode Scan Rate (ms)	C3	RW	Don't Care	Don't Care	Don't Care	WoP Report Period (ms) Lower Byte	WoP Report Period (ms) Upper Byte	NA	tCaptive AppWakeOnProxModeScan	AppWakeOnProxModeScan
Wakeup Interval	C4	RW	Don't Care	Don't Care	Don't Care	Wakeup Interval	Don't Care	NA	tCaptive AppWakeOnInterval	AppWakeOnInterval
Inactivity Timeout	C5	RW	Don't Care	Don't Care	Don't Care	Timeout Lower Byte	Timeout Upper Byte	NA	tCaptive AppInactivityTimeout	AppInactivityTimeout

## 9.8.5 UART Driver

The MSP430 eUSCI\_A UART Driver provides a simple UART API to MSP430 applications, enabling interrupt-driven transmit and receive operations as well as error handling. This section provides an overview of the driver's features, architecture, and API. This document assumes that the reader is familiar with the MSP430 MCU architecture, as well as embedded C programming concepts.

### Related Documents

This document should be used with the following additional supporting documentation. The MSP430 Driver Library API is not specific to this driver, and has its own documentation.

- MSP430 Driver Library (DriverLib) User's Guide
- The relevant MSP430 Family User's Guide

#### 9.8.5.1 Purpose of the Driver

The eUSCI\_A UART Driver enables developers to quickly get up and running with UART communication via an API that is similar to one used on a PC. The API provides simple functions such as `UART_openPort()` and `UART_transmitBuffer()`, and it allows the developer to register event handlers for received data and error conditions. To enable portability, the driver is built upon the MSP430 DriverLib register abstraction. This driver has been designed to work with MSP430 DriverLib build 1.90.00.00 and greater. Like any other serial interface, UART has benefits and drawbacks. Whether or not it is the best choice for an embedded interface depends on a number of factors. The benefits and drawbacks of UART are discussed below.

#### UART Benefits

- Ease of implementation. UART is one of the most common serial communication protocols around. While interfaces with higher speed and higher reliability exist, such as I2C and SPI, none match UART for ease of implementation. UART is easily interfaced to a host PC or another microcontroller, and as such it is well suited for use as a debug console.
- Point-to-point simplicity. Since only two nodes exist, the interface is not a bus with multiple devices attached. This removes the need for addressing or chip-select overhead.
- Full duplex communication. In this UART driver implementation, transmit and receive operations are decoupled and may occur concurrently. Each of the two connected nodes may consider itself a master, and may begin transmission to the other node at any time.

#### UART Drawbacks

- Critical bit timing. Because UART transmission is asynchronous (there is no bit clock), accurate timing must be guaranteed on both sides of the interface.

- 
- Bit rate limitation. Due to the critical bit timing above, UART is typically more limited in maximum bit clock than when compared with a synchronous interface, such as SPI or I2C.
  - Critical ISR timing. This driver does not employ the use of a DMA channel for moving data from transmit and receive buffers into RAM, as not all devices are equipped with a DMA engine. Instead of DMA, all data transfers are handled via interrupt service routines. This means that the CPU must be available to service receive interrupts at least every byte period. A byte period is defined as the transmission time for one byte, which is equal to 10 times the bit period. Note that the byte transmission time increases if parity or more than one stop bit is used. For example, if the bit clock is 250 kHz, the transmission time of one byte is 4us times 10 bits, or 40us. With an 8MHz CPU clock, there are only 320 instruction cycles available every 40us. If the CPU is not available to service receive interrupts, received bytes may be lost. The driver provides an error detection mechanism to alert the application if and when data is being lost.
  - Asynchronous overhead. UART as configured in this driver requires at least one start bit and one stop bit. As such, each 8-bit byte requires that 10 bits be sent. This overhead would not be present in a SPI interface, although I2C does impose an ACK/NACK bit as well as addressing overhead.

#### 9.8.5.2 Driver Features

The key features implemented in the UART driver are listed below.

- Full duplex bi-directional communication
- No flow control required
- Fully interrupt-driven
- Non-blocking transmit function
- Receive event callback
- Error event callback
- Compile-time selection of which eUSCI\_A instance to use

#### 9.8.5.3 Driver Overview

The UART driver is provided in source code. The driver consists of three source files:

- `UART_Definitions.h` (Configuration header file)
- `UART.h` (API Header file)
- `UART.c` (API Implementation file) The core driver is implemented in `UART.h` and `UART.c`. The `UART_Definitions.h` file allows the developer to adjust the driver's compile-time options.

The UART driver requires the following hardware resources:

- One eUSCI\_A peripheral instantiation
- Two device pins (`UCAxTXD` and `UCAxRXD`), where 'x' represents the selected eUSCI\_A instance

The UART driver API is composed of functions and data types. The sections below describe how to configure the UART driver and use it to perform transmit and receive operations. The UART driver employs a basic software state machine to manage operations. The three driver states and their interconnection are shown in the diagram below.

## Driver States

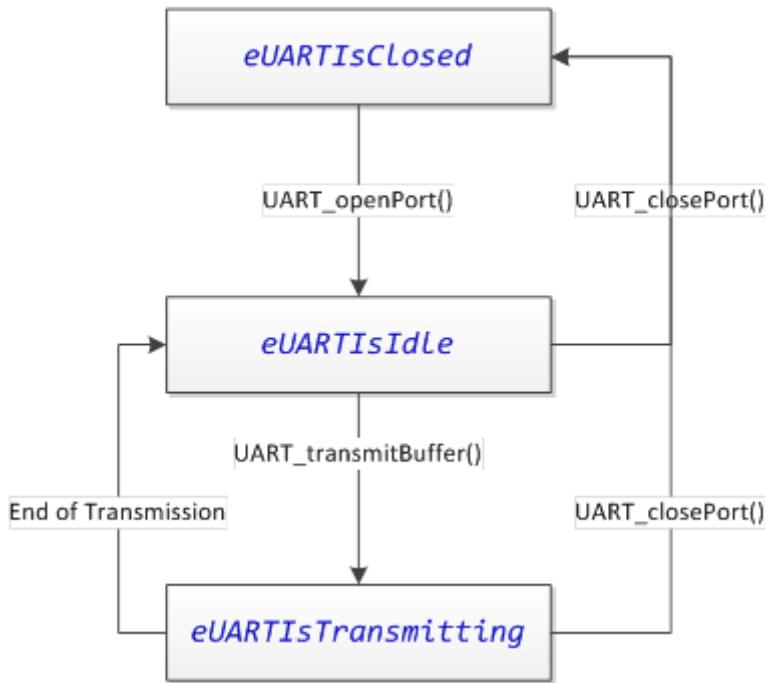


Figure 9.23: UART Driver State Machine

### 9.8.5.4 Compile-time Driver Configuration Options

The UART driver has compile-time options as well as run-time configurable options. Certain things must be known at compile-time, such as which eUSCI\_A peripheral instance is associated with the driver. Other options, such as baud rate, may be controlled at runtime.

The compile-time configuration options are set in the `UART_Definitions.h` file. The available compile-time options are described below.

#### UART Enable Definition

Parameter	File	Valid Values
<code>UART__ENABLE</code>	<code>UART_Definitions.h</code>	true, false

The `UART_ENABLE` compile-time option selects whether the UART Driver is enabled or disabled. This provides a mechanism to exclude the driver from the compilation process. To include the driver, define `UART__ENABLE` as true. Else, define it as false.

#### UART eUSCI\_A Peripheral Selection Definition

Parameter	File	Valid Values
<code>UART__EUSCI_A_PERIPHERAL</code>	<code>UART_Definitions.h</code>	<code>EUSCI_A0_BASE</code> , <code>EUSCI_A1_BASE</code>

The `UART_EUSCI_A_PERIPHERAL` compile-time option allows easy selection of which eUSCI\_A instance to associate with the UART driver. This provides flexibility during design if a pin-mux change is necessary. A valid base address must be provided. When a eUSCI\_A peripheral selection is made, the UART driver ISR address is linked to the appropriate eUSCI interrupt vector automatically. If an invalid address is selected, a compiler error is thrown.

#### UART Low Power Mode (LPMx)

Parameter	File	Valid Values
UART__LPMx_bits	UART_Definitions.h	0, LPM0_bits, LPM1_bits, LPM2_bits, LPM3_bits

UART communications is often performed in a low power mode. The UART LPM Mode configuration is used in three ways, as described below.

- The receive callback and error callback functions may trigger an active exit from a low power mode to wake the CPU for further action. If a callback function returns a Boolean true, the UART driver will wake the CPU after returning from the callback. It does this by clearing the status register bits indicated in the UART LPM Mode configuration upon exit from the UART ISR.
- A transmit operation will exit the low power mode specified by the UART LPM Mode configuration once the full transmission is complete, to allow the application to know that the transmission was completed.
- The transmit function is normally a non-blocking function. Once transmission begins, the function returns. However, in the event that a previous transmission was still in progress when the transmit function was called, there is an option to wait for the previous transmission to complete inside of a low power mode. If that option is selected, the low power mode specified by the UART LPM Mode configuration will be entered while waiting. The transmit function will know to begin transmission when the previous transmission exits active (per number 2 above).

#### 9.8.5.5 Run-time Driver Configuration Options

The UART driver has compile-time options as well as run-time configurable options. The runtime configuration options are specified by populating a tUARTPort structure in the application, and passing this structure to the driver when opening the driver. A tUARTPort structure is required when opening the UART driver. The tUARTPort structure must be available in memory whenever the driver is open. This is a result of the fact that the UART driver references this structure at runtime to find the callback functions for receive handling and error handling. However, the driver does not modify the data in the structure at any time, and as such, the structure may be placed in a read-only memory section (such as C const memory). These parameters are considered runtime adjustable because the parameters may be modified by the application if the UART driver is closed first, then re-opened. For example, the application may change the UART baud rate by closing the port, changing the baud rate options on the tUARTPort structure, and re-opening the port. Note that the .peripheralParameters member of the tUARTPort structure is a EUSCI\_A\_UART\_initParam structure from the MSP430 Driver Library.

Member	Description	Valid Values
bool (*pbReceiveCallback)(uint8_t)	pbReceiveCallback is a function pointer that may point to a receive event handler. If no receive handling is required, initialize this member to 0 (null).	Null, or a valid function address.
bool (*pbErrorCallback)(uint8_t)	pbErrorCallback is a function pointer that may point to an error event handler. If no error handling is required, initialize this member to 0 (null).	Null, or a valid function address.
.peripheralParameters.selectClockSource	This member specifies the clock source for the eUSCI_A peripheral.	EUSCI_A_UART_CLOCKSOURCE_SMCLK, EUSCI_A_UART_CLOCKSOURCE_ACLK
.peripheralParameters.clockPrescalar	This member specifies the eUSCI_A clock prescalar. This affects the baud rate.	0-65535
.peripheralParameters.firstModReg	This member specifies the eUSCI_A first stage modulation. This affects the baud rate.	0-15

.peripheralParameters.secondModReg	This member specifies the eUSCI_A second stage modulation. This affects the baud rate.	0-255
.peripheralParameters.parity	This member specifies the UART parity mode.	EUSCI_A_UART_NO_PARITY, EUSCI_A_UART_ODD_PARITY, EUSCI_A_UART_EVEN_PARITY
.peripheralParameters.msbOrLsbFirst	This member specifies the transmission bit order.	EUSCI_A_UART_LSB_FIRST, EUSCI_A_UART_MSB_FIRST
.peripheralParameters.numberofStopBits	This member specifies the number of stop bits.	EUSCI_A_UART_ONE_STOP_BIT, EUSCI_A_UART_TWO_STOP_BITS
.peripheralParameters uartMode	This member specifies the UART mode.	EUSCI_A_UART_MODE, EUSCI_A_UART_IDLE_LINE_MULTI_PROCESSOR, EUSCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR, EUSCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION
.peripheralParameters.oversampling	This member specifies whether UART oversampling is enabled.	EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION, EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION

#### 9.8.5.6 Using the Driver: Opening and Closing the Driver

Opening and closing of the UART driver is accomplished through the following function calls:

Description	Declaration
Open the UART Port	extern void <b>UART_openPort</b> (const tUARTPort *pPort)
Close the UART Port	extern void <b>UART_closePort</b> (void)

The UART driver is opened and initialized by a call to **UART\_openPort()**, which is passed a completed tUARTPort structure. The tUARTPort structure must be populated by the application. It may be placed in read-only memory, as the UART API does not modify the structure at any time; rather, it only references it. It is important that the structure be left in memory at the address given when **UART\_openPort()** is called, as the UART API will reference this structure to access callback functions when the port is open. If the memory must be freed, first close the UART port with a call to **UART\_closePort()**.

A call to **UART\_closePort()** will disable the UART port and its associated eUSCI\_A peripheral, halting all Rx/Tx interrupt activity. After the port is closed, the event handlers will no longer be called, and the tUARTPort structure memory may be released to the application.

#### Opening a Port

```

// g_myUartPort specifies the UART port configuration that is passed
// to the UART port driver during init.
// The UART configuration is 9600B8N1, sourced by a 4MHz SMCLK.
//
const tUARTPort g_myUartPort =
{
    .pbReceiveCallback = &receiveHandler,
    .pbErrorCallback = &errorHandler,
    .peripheralParameters.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_SMCLK,
    .peripheralParameters.clockPrescalar = 26,
    .peripheralParameters.firstModReg = 0,
    .peripheralParameters.secondModReg = 0xB6,
    .peripheralParameters.parity = EUSCI_A_UART_NO_PARITY,
    .peripheralParameters.msbOrLsbFirst = EUSCI_A_UART_LSB_FIRST,
    .peripheralParameters.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT,
    .peripheralParameters uartMode = EUSCI_A_UART_MODE,
    .peripheralParameters.oversampling = EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION
};

UART_openPort(&g_myUartPort);

```

---

#### **9.8.5.7 Using the Driver: Transmitting Data**

Data transmission is accomplished through the following function calls:

---

Description	Declaration
Transmit a Buffer	extern void <b>UART_transmitBuffer</b> (const uint8_t *pBuffer, uint16_t ui16Length)
Transmit a Byte	extern void <b>UART_transmitByteImmediately</b> (uint8_t ui8Data)
Get Port Status	extern uint8_t <b>UART_getPortStatus</b> (void)

Transmit operations are interrupt driven. To initiate a transmission after opening the UART port, make a call to **UART\_transmitBuffer()**. This function is simply passed a pointer to the buffer to transmit, and the length of the buffer to transmit (specified in bytes). If the eUSCI\_A peripheral is available, transmission will begin immediately and the function will return. If the peripheral is still busy sending a previous transmission, it will block (in a low power mode, if specified in **UART\_Definitions.h**) until the previous transmission is complete.

### Transmitting a Buffer

```
UART_transmitBuffer("\n\rHello World!\n\r", sizeof("\n\rHello World!\n\r"));
```

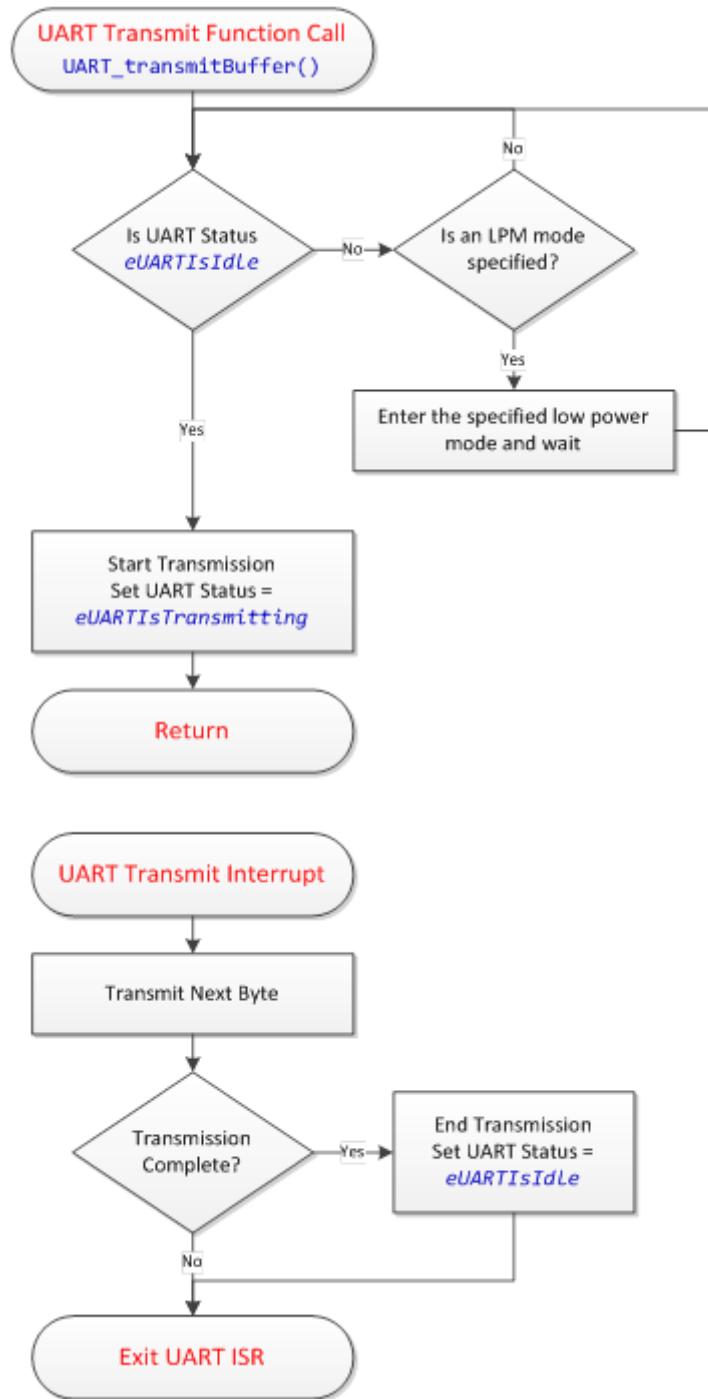


Figure 9.24: UART Transmit Flow Diagram

---

The transmit operation uses the buffer memory that was pointed to in the **UART\_transmitBuffer()** function - it does not perform a data copy in the name of efficiency. As such, that memory must not be modified during the transmission, or the transmission will be invalid. To detect when transmission has completed and the memory is again available, it is possible to check the UART port status via a call to **UART\_getPortStatus()**. If an application will be re-using buffer space, it is best to employ a ping-pong buffer strategy so that a new packet may be assembled while a previous packet is being sent.

A call to **UART\_getPortStatus()** returns one of the values enumerated by tUARTStates. The options are described below.

Port Status Option	Description
eUARTIsClosed	The UART driver is not currently open, and is neither receiving nor transmitting data.
eUARTIsIdle	The UART driver is currently open, but is not in the process of transmitting a buffer.
eUARTIsTransmitting	The UART driver is currently open, and is in the process of transmitting a buffer.

If only a single byte is going to be sent, it is possible to send it immediately via the **UART\_transmitByteImmediately()** function. This function sends a single byte as soon as the eUSCI\_A peripheral transmit buffer is available. It will block until the buffer is available. If a transmission started by a call to **UART\_transmitBuffer()** is in progress, this transfer may finish first as it is interrupt-driven. The **UART\_transmitByteImmediately()** function is mainly available to be used in terminal emulator applications, where sent ASCII characters are echoed back to be visible to the user.

### Transmitting a Single Byte

```
uint8_t ui8Data = 0x01;  
UART_transmitByteImmediately(ui8Data);
```

#### 9.8.5.8 Using the Driver: Receiving Data

Received data is passed to the application through the use of the receive callback. The UART driver will call a user-defined function every time a new byte is received from the UART peripheral, passing that byte to the receive callback. It is then up to the application to decide how to handle the data. It is recommended to use a ring-buffer FIFO queue to handle buffering incoming data. The data may then be extracted and processed in a background process. The receive event handler is called from the UART ISR in the UART driver, and as such, the event handler should be kept as short as possible. It is recommended to follow the same practice used to write ISR's when writing event handlers.

The receive callback function is linked to the driver by placing its address in the pbReceiveCallback member of the tUARTPort structure. If the application does not wish to listen for receive events, the receive callback pointer in the tUARTPort structure may be initialized to null. Receive operations are performed entirely out of the ISR. The receive operation ISR flow is shown below. [Errors](#) are checked for whenever a receive interrupt is serviced.

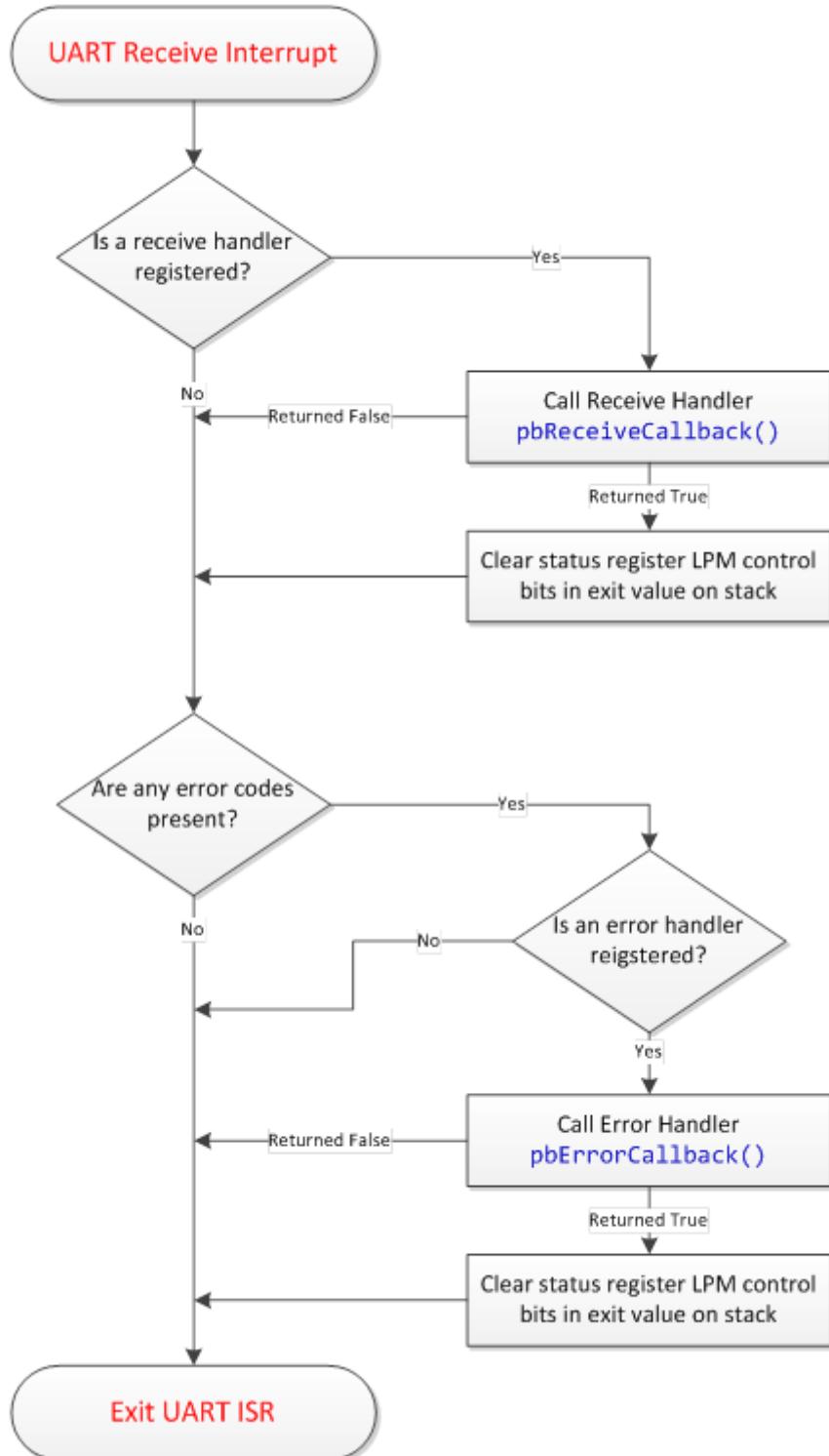


Figure 9.25: UART Receive Flow Diagram

---

### 9.8.5.9 Using the Driver: Error Handling

The UART driver employs basic UART error detection to alert the application when something has gone wrong. The two errors detected by the driver are:

1. UART Receive Buffer Overrun. This occurs if a new byte is placed into the eUSCI\_A UART receive buffer before a previous byte was read out. This occurs when there is not enough CPU bandwidth to read out data at the rate data is coming into the peripheral. To resolve this error during development, it is necessary to either: slow down the rate at which data is received, whether through delays between bytes or a lower baud rate, or to increase the CPU availability by shortening interrupt handlers or increasing the CPU clock frequency. The enumerated error flag is eUARTOverrunError.
2. UART Framing Error. This typically occurs if there is a baud rate mismatch between devices. The enumerated error flag is eUARTFramingError. The driver alerts the application via the error callback function. This works similar to the receive callback, except the error code is passed rather than the received byte. The error callback function is called from the UART ISR, and as such, it should be kept as short as possible. The error callback function is linked to the driver by placing its address in the pbErrorCallback member of the tUARTPort structure. If the application does not wish to listen for error events, the error callback pointer in the tUARTPort structure may be initialized to null.

### 9.8.5.10 Using the Driver: Example Application

This example listens for a ASCII line on eUSCI\_A0. When a line is received, it is echoed back. The baud rate is 9600-8N1. The max line size is 64 bytes.

```
#include "driverlib.h"
#include "UART.h"

// DCO_FREQ defines the MSP430 DCO frequency in Hz
// REFO_FREQ defines the REFO frequency in Hz
// RECEIVE_BUFFER_SIZE defines the size of the UART receive buffer.
// g_ui8ReceiveBuffer is the UART receive buffer. It is filled with
// incoming characters via the UART receive event handler.
// g_ui8ReceiveIndex is the UART receive buffer index. It defines the
// buffer index that the next received byte is placed into.
// g_bGotNewLine is a flag used by the UART receive event handler to
// signal the background loop (the main application thread) that a full
// ASCII line has been received in the buffer.
bool g_bGotNewLine = false;

// The UART receive handler is called by the UART port driver whenever a new
// character is received.
bool receiveHandler(uint8_t ui8Data)
{
    if (g_bGotNewLine == false)
    {
        if (g_ui8ReceiveIndex<RECEIVE_BUFFER_SIZE)
        {
            // It is safe to receive the new character into the buffer.
            g_ui8ReceiveBuffer[g_ui8ReceiveIndex] = ui8Data;
            g_ui8ReceiveIndex++;
        }
    }
}
```

```

        // If the data is a ASCII return, signal the background app
        // and exit awake.
        //
        g_ui8ReceiveBuffer[g_ui8ReceiveIndex++] = ui8Data;
        if (ui8Data=='\r')
        {
            g_bGotNewLine = true;
            return true;
        }
        else
        {
            UART_transmitByteImmediately(ui8Data);
        }
    }
    else
    {
        //
        // The buffer is full, return new line to print what
        // we have right now
        //
        g_bGotNewLine = true;
        return true;
    }
}
return false;
}

//
// The UART error handler is called by the UART port driver whenever an
// error on the UART port is detected.
// The application could handle errors here if desired (such as a UART buffer
// overrun or framing error).
//
//
bool errorHandler(uint8_t ui8Error)
{
    return false;
}

//
// g_myUartPort specifies the UART port configuration that is passed
// to the UART port driver during init.
// The UART configuration is 9600B8N1, sourced by SMCLK.
//
const tUARTPort g_myUartPort =
{
    .pbReceiveCallback = &receiveHandler,
    .pbErrorCallback = &errorHandler,
    .peripheralParameters.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_SMCLK,
    .peripheralParameters.clockPrescalar = 26,
    .peripheralParameters.firstModReg = 0,
    .peripheralParameters.secondModReg = 0xB6,
    .peripheralParameters.parity = EUSCI_A_UART_NO_PARITY,
    .peripheralParameters.msborLsbFirst = EUSCI_A_UART_LSB_FIRST,
    .peripheralParameters.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT,
    .peripheralParameters.uartMode = EUSCI_A_UART_MODE,
    .peripheralParameters.overSampling = EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION
};

//
// Application main()
//
void main(void)
{
    //
    // Set up the MCU with a 8MHz MCLK, 4MHz SMCLK, and ~32.768kHz ACLK.
    //
    WDT_A_hold(WDT_A_BASE);
    CS_clockSignalInit(CS_FLLREF, CS_REFCLK_SELECT, CS_CLOCK_DIVIDER_1);
    CS_initFLLSettle(DCO_FREQ/1000, DCO_FREQ/REFO_FREQ);
    CS_clockSignalInit(CS_MCLK, CS_DCCLKDIV_SELECT, CS_CLOCK_DIVIDER_1);
    CS_clockSignalInit(CS_SMCLK, CS_DCCLKDIV_SELECT, CS_CLOCK_DIVIDER_2);
    CS_clockSignalInit(CS_ACLK, CS_REFCLK_SELECT, CS_CLOCK_DIVIDER_1);
    __delay_cycles(8000000);

    //
    // Set P1.0 and P1.1 to UCA0TXD and UCA0RXD, respectively.
    //
    PM5CTL0 &= ~LOCKLPM5;
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P1, GPIO_PIN4, GPIO_PRIMARY_MODULE_FUNCTION);
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1, GPIO_PIN5, GPIO_PRIMARY_MODULE_FUNCTION);

    //
    // Enable masked interrupts.
    //
    __bis_SR_register(GIE);
}

```

---

```

// Open the UART port and send the welcome message!
//
UART_openPort(&g_myUartPort);
UART_transmitBuffer("\n\rHello World!\n\r", sizeof("\n\rHello World!\n\r"));
UART_transmitBuffer("Waiting for a new line... ", sizeof("Waiting for a new line... "));

//
// Application background loop
//
while(1)
{
    //
    // If a new line was received, echo it back!
    //
    if (g_bGot.NewLine == true)
    {
        UART_transmitBuffer("\n\rReceived: ", sizeof("\n\rReceived: "));
        UART_transmitBuffer(g_ui8ReceiveBuffer, g_ui8ReceiveIndex);
        UART_transmitBuffer("\n\r", sizeof("\n\r"));
        UART_transmitBuffer("Waiting for a new line: ", sizeof("Waiting for a new line: "));
        g_ui8ReceiveIndex = 0;
        g_bGot.NewLine = false;
    }

    //
    // Sleep in LPM0 (CPU off, DCO on).
    // The UART receive event handler will
    // wake up the background loop if a line is received.
    //
    LPM0;
}

```

## 9.8.6 I2C Slave Driver

The MSP430 eUSCI\_B I2C slave driver provides a simple I2C slave API to MSP430 applications, enabling interrupt-driven I2C read/write operations as well as bus timeout detection and driver error handling. This User's Guide provides an overview of the driver's features, architecture, and API. This document assumes that the reader is familiar with MSP430 MCU architecture, as well as embedded C programming concepts and basic I2C principles. Note: From this point forward, the I2C slave driver will simply be referred to as "the driver."

### Related Documents

This guide should be used with the following additional supporting documentation. The MSP430 Driver Library API is not specific to this driver, and has its own documentation.

- MSP430 Driver Library (DriverLib) User's Guide
- The relevant MSP430 Family User's Guide

#### 9.8.6.1 Purpose of the Driver

The driver enables quick development of MSP430 applications where the MSP430 itself is a slave device to some other master in a larger embedded system. This is a common application, as MSP430 microcontrollers are often a secondary MCU to a larger host MCU or host MPU.

The driver is structured to be flexible, enabling many different applications. It is capable of providing a register-file interface to a host processor, similar to a sensor or an EEPROM. It may also be used as a bulk-transfer interface to a host.

#### 9.8.6.2 Driver Features

The key features provided by the driver are listed below.

- Interrupt-driven I2C slave software state machine

- 
- I2C Write (master transmitter, slave receiver) event callback
  - I2C Read (master receiver, slave transmitter) buffer assignment
  - Support for I2C start/stop/start as well as I2C start/repeated-start sequences
  - I2C error/warning reporting via an error callback function
  - Operation in LPM3 between I2C interrupts when timeout feature is used
  - Operation in LPM4 between I2C interrupts when timeout feature is not used
  - Open drain slave request signal (to alert the master to service the slave)
  - Slave request timeout capability to prevent an application stall
  - I2C bus transaction timeout capability to prevent an application stall

#### 9.8.6.3 Driver Overview

The driver is provided in C source code. It consists of 3 base driver source files, plus an additional 3 source files related to the I2C timeout detection feature (which is a compile-time option).

Base Driver:

- I2CSlave\_Definitions.h (Configuration header file)
- I2CSlave.h (API header file)
- I2CSlave.c (API implementation file)

Function Timer (used for I2C timeout detection):

- FunctionTimer\_Definitions.h (Configuration header file)
- FunctionTimer.h (API header file)
- FunctionTimer.c (API source file)

The I2CSlave\_Definitions.h and FunctionTimer\_Definitions.h files contain the compile-time options for each component.

The driver requires the following MCU hardware resources:

Base Driver:

- One eUSCI\_B peripheral instantiation
- Two device pins (UCBxSDA and UCBxSCL), where 'x' represents the selected eUSCI\_B instance

Slave Request Feature (Optional):

- One additional device pin (Any digital IO for the slave request line feature).

Timeout Feature (Optional):

- One TIMER\_A peripheral instantiation with at least 2 capture-compare units (CCR0 and CCR1)

The driver operation is based upon a software state machine that keeps track of the current driver state. There are four possible states: closed, idle, read, and write. Since the driver implements an I2C slave, it is important to be clear on the naming conventions for read and write. An I2C bus write is a receive operation from the perspective of an I2C slave, as the bus master is writing to the slave. Similarly, an I2C bus read is a transmit operation from the perspective of the slave. The state machine implemented by the driver is depicted below.

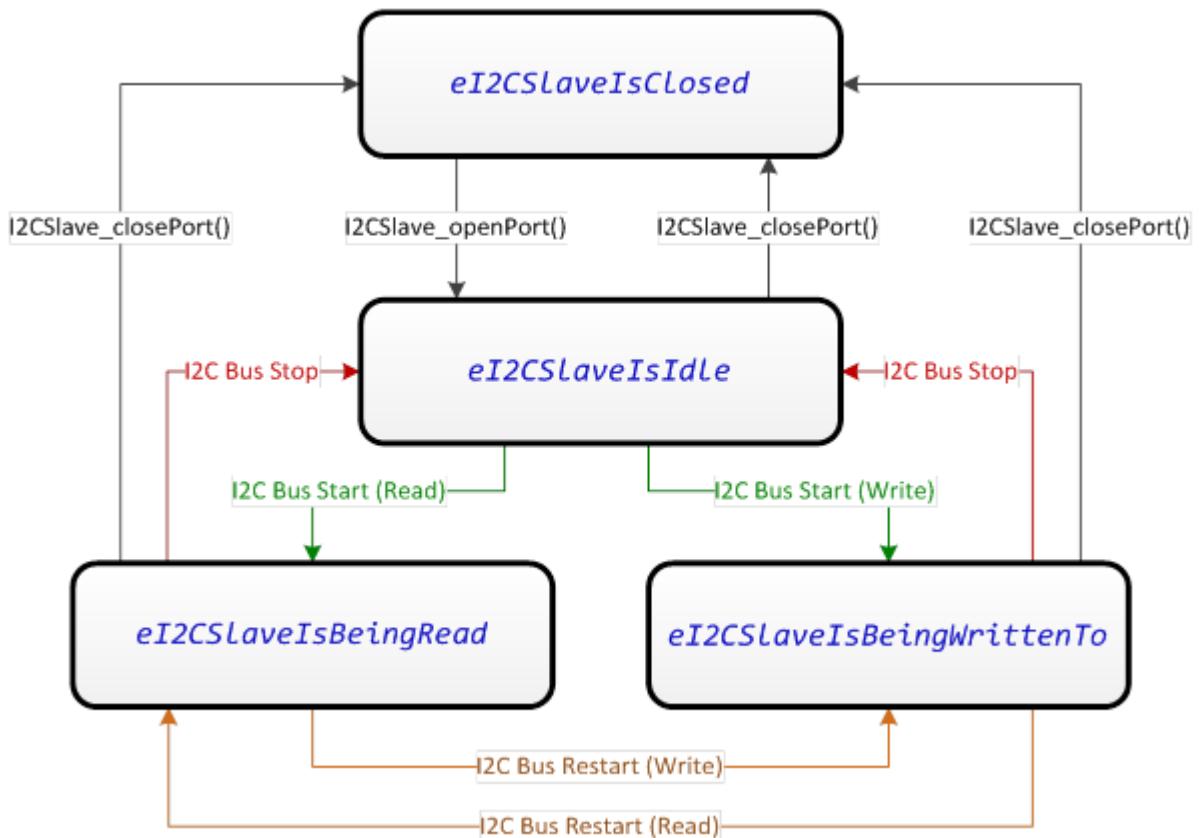


Figure 9.26: I2C Slave Driver State Machine

As shown, state changes between idle, read, and write are controlled solely by the I2C bus master. It is possible for the slave to close the driver at any time, however.

#### 9.8.6.3.1 Driver Overview: I2C Receive Operations (I2C Bus Write)

The driver enters the "write" state (`eI2CSlaveIsBeingWrittenTo`) whenever the bus master sends a start or restart condition to the driver's 7-bit I2C address with the R/\_W bit cleared. In this state, the driver is receiving data. Data is received into the buffer memory that was specified by the user when the driver was opened. If the bus master attempts to write data beyond the length of the receive buffer, the data is ignored and an error callback alerts the application. The write state is cleared when a stop condition or a restart condition is issued. When this happens, the driver calls a user-specified callback function for the received data to be handled.

#### 9.8.6.3.2 Driver Overview: I2C Transmit Operations (I2C Bus Read)

The driver enters the "read" state (`eI2CSlaveIsBeingRead`) whenever the bus master sends a start or restart condition to the driver's 7-bit I2C address with the R/\_W bit set. In this state, the driver is transmitting data to the master by loading data from the latest transmit buffer memory that was linked to the driver. Note that the buffer memory must have been pre-loaded before the read state was entered. If the bus master attempts to read data before any buffer was specified or if it attempts to read out more data than was made available in the buffer, the driver will clock out an invalid byte, which may be specified as a compile-time option. The read state is cleared when a stop condition or a restart condition is issued. When this happens, the driver wakes the CPU from any low power modes in case it was pending on the completion of a read operation.

#### 9.8.6.3.3 Driver Overview: Slave Request Feature

The driver provides a mechanism for alerting the bus master that it wishes to communicate. This is helpful in many applications, as the slave has no way to initiate communication on the I2C bus. The slave request feature is implemented as an open-drain request line. The request line may be any digital IO on the MCU. The request line should be pulled up to VCC via a pull-up resistor, just like an I2C bus line. The driver controls whether the request

---

line IO is left tri-stated (Hi-Z), or whether it is pulled to a logic low (sinking current to ground through the pullup resistor and the digital IO). The driver API contains a function for "pulling" the request line, which waits for a I2C bus response from the master (or a timeout) before returning.

#### 9.8.6.3.4 Driver Overview: Timeout Feature (Function Timer)

The driver provides a timeout mechanism for preventing application lock-ups. To enable timeouts in a low-power way, a dedicated hardware timer is used to set limits on how long specific driver operations may take. The operations that may have timeouts placed on them are:

1. **I2C Slave Request Timeout.** The slave request feature described in section 2.3.3 signals the bus master to communicate with the slave. That slave request API function is designed to block until the master starts communication (via a I2C bus read start condition). If the timeout feature is enabled, the slave request function will time out automatically if the master does not start a read within the specified timeout period.
2. **I2C Transaction Timeout.** The transaction timeout places a limit on how long any I2C bus transaction may take. An I2C transaction time is defined between a start or restart condition until a restart or stop condition is issued. The goal of the transaction timeout is to detect scenarios where the bus master went down in the middle of transmission, or an I2C bus disconnection.

The timeout feature is implemented by the Function Timer module, which is a completely independent software module. As transaction monitoring may be a part of the application layer (perhaps with a standard watchdog timer), the timeout feature may be excluded completely at compile time. The Function Timer module essentially calls a predefined function in the foreground after a defined delay in timer cycles. That function can then take actions such as cancelling a slave request or resetting the driver.

#### 9.8.6.4 Compile-time Driver Configuration Options

The compile-time configuration options are set in the I2CSlave\_Definitions.h and FunctionTimer\_Definitions.h files. These options are described below.

##### I2C Slave Enable Definition

Parameter	File	Valid Values
I2CSLAVE__ENABLE	I2CSlave_Definitions.h	true, false

The I2C Slave Enable compile-time option selects whether the driver is enabled or disabled. This provides a mechanism to exclude the driver from the compilation process. To include the driver, define I2CSlave\_\_ENABLE as true. Else, define it as false.

##### I2C eUSCI\_B Peripheral Selection Definition

Parameter	File	Valid Values
I2CSLAVE_EUSCI_B_PERIPHERAL	I2CSlave_Definitions.h	EUSCI_B0_BASE

The I2C eUSCI\_B peripheral selection allows easy selection of which eUSCI\_B instance to associate with the I2C driver. This provides flexibility during design if a pin-mux change is necessary. A valid base address must be provided.

##### I2C Slave Address Definition

Parameter	File	Valid Values
I2CSLAVE_ADDRESS	I2CSlave_Definitions.h	0x00 to 0x7F

The I2C slave address specifies the 7-bit I2C bus address associated with this device.

##### I2C Slave LPM Mode Definition

Parameter	File	Valid Values
I2CSLAVE__LPMx_bits	I2CSlave_Definitions.h	LPM0_bits, LPM1_bits, LPM2_bits, LPM3_bits, LPM4_bits

I2C communication may be performed in a low power mode. No clocks need to be enabled on the MSP430 to send or receive bytes on a eUSCI\_B peripheral when in I2C slave mode. This is a benefit of the bit clock being provided by the bus master. The I2C Slave LPM mode control is used in the following ways.

1. Receive callback functions may trigger an active exit from the I2C ISR after the completion of a receive event (triggered by either a re-start or a stop condition). If a receive callback returns true to the driver, the status register bits specified by the I2C slave LPM mode will be cleared when exiting.
2. A call to I2CSlave\_SetTransmitBuffer() will block if a previous read is requested or in progress. If a low power mode is specified via the I2C Slave LPM Mode definition, the block will take place in that low power mode—otherwise, it will block in active mode.

#### I2C Slave Invalid Byte Definition

Parameter	File	Valid Values
I2CSLAVE__INVALID_BYTE	I2CSlave_Definitions.h	0x00 to 0xFF

The invalid byte specifies the byte that is transmitted to the master (during an I2C read) if the master attempts to read beyond the length of the transmit buffer provided to the driver.

#### I2C Slave Timeout Enable Definition

Parameter	File	Valid Values
I2CSLAVE__TIMEOUT_ENABLE	I2CSlave_Definitions.h	true, false

The slave timeout enable controls whether the timeout feature of the driver is included. The timeout feature provides the ability to set a maximum amount of time a certain I2C slave task may take before it fails. The two tasks that may be monitored with a timeout is the I2C slave request and any I2C transaction. Note that enabling the timeout feature requires the inclusion of the FunctionTimer source files, a Timer\_A instance, and additional memory.

#### I2C Slave Request Timeout Cycles Definition

Parameter	File	Valid Values
I2CSLAVE__REQ_TIMEOUT_CYCLES	I2CSlave_Definitions.h	0x0000 - 0xFFFF

The I2C slave request timeout cycles specifies the timeout period for the I2C request timeout, in units of the function timer clock period. This is the amount of time the driver will wait after pulling the slave request line low before the transaction fails out. The bus master must respond to the slave request within this amount of time.

#### I2C Slave Transfer Timeout Cycles Definition

Parameter	File	Valid Values
I2CSLAVE__TXFR_TIMEOUT_CYCLES	I2CSlave_Definitions.h	0x0000 - 0xFFFF

The I2C slave transfer timeout cycles specifies the timeout period for any I2C transaction, in units of the function timer clock period. I2C transactions are timed from start condition to stop condition, or start condition to re-start condition.

#### I2C Slave Request Enable Definition

Parameter	File	Valid Values
I2CSLAVE__REQ_ENABLE	I2CSlave_Definitions.h	true, false

The request enable controls whether the I2C request line feature is included in the driver build. The I2C request line provides a mechanism for the slave to signal the master that it would like to communicate.

#### I2C Slave Request Line Definition

Parameter	File	Valid Values
I2CSLAVE_REQ_POUT	I2CSlave_Definitions.h	PxOUT
I2CSLAVE_REQ_PDIR	I2CSlave_Definitions.h	PxDIR
I2CSLAVE_REQ_MASK	I2CSlave_Definitions.h	BIT0 - BIT7

The I2C slave request line is defined by three values- the port output register, the port direction register, and the pin mask. These do not need to be defined if the slave request enable is set to false.

#### Function Timer Enable Definition

Parameter	File	Valid Values
FUNCTIONTIMER_ENABLE	FunctionTimer_Definitions.h	true, false

The function timer enable controls whether the function timer is included in the driver build. If the I2C slave timeout feature is not used, memory is conserved when the function timer is disabled (set to false).

#### Function Timer Peripheral Definition

Parameter	File	Valid Values
FUNCTIONTIMER_PERIPHERAL	FunctionTimer_Definitions.h	TIMER_A0_BASE, TIMER_A1_BASE, TIMER_A2_BASE, TIMER_A3_BASE

The function timer peripheral stores the base address of the Timer\_A instance that should be associated with the function timer. This instance must have at least two capture compare units (CCR0 and CCR1).

#### Function Timer Clock Source Definition

Parameter	File	Valid Values
FUNCTIONTIMER_CLOCK	FunctionTimer_Definitions.h	TASSEL_SMCLK, TASSEL_ACLK

The function timer clock determines the resolution of the function timer, as well as the maximum delay. Sources include SMCLK and ACLK. Note that if the clock source is changed, the effective function timer delay may change.

#### Function Timer Clock Divider Definition

Parameter	File	Valid Values
FUNCTIONTIMER_DIVIDER	FunctionTimer_Definitions.h	ID_1, ID_2, ID_4, ID_8

The function timer clock divider divides down the source clock (which was specified above). Note that if the clock divider is changed, the effective function timer delay may change.

#### Function Timer Extended Clock Divider Definition

Parameter	File	Valid Values
FUNCTIONTIMER_EXDIVIDER	FunctionTimer_Definitions.h	TAIDEX_0 to TAIDEX_7

The function timer extended divider allows an additional second divider to be added in series with the standard divider. Note that if the extended clock divider is changed, the effective function timer delay may change.

#### Function Timer LPM Clear Definition

Parameter	File	Valid Values
FUNCTIONTIMER_LPM_CLEAR	FunctionTimer_Definitions.h	LPM0_bits, LPM1_bits, LPM2_bits, LPM3_bits, LPM4_bits

The function timer LPM clear controls which LPM bits are cleared upon exit from a function called in the foreground by the function timer. This should be set to enable CPU wake-up after a timeout event. The function timer feature will never use these bits to enter into a low power mode- it will only use them to exit.

---

#### 9.8.6.5 Run-time Driver Configuration Options

The driver's runtime configuration options are specified by populating a `tI2CSlavePort` structure in the application, and passing this structure to the driver when opening the driver. The `tI2CSlavePort` structures are discussed below.

Member	Description	Valid Values
bool (*pbReceiveCallback)(uint16_t)	pbReceiveCallback is a function pointer that may point to a receive event handler in the application. If no receive handling is required, initialize this member to 0 (null).	Null, or a valid function address.
void (*pvErrorCallback)(uint8_t)	pvErrorCallback is a function pointer that may point to an error event handler in the application. If no error handling is required, initialize this member to 0 (null).	Null, or a valid function address
ui16ReceiveBufferSize	This member stores the size of the receive buffer pointed to by pReceiveBuffer.	0x00 to 0xFF
pReceiveBuffer	This member is a pointer to the I2C receive buffer.	A valid pointer
bSendReadLengthFirst	When set to true, this flag configures the driver to always load the length of the transmit buffer as the first data byte read by the bus master. This is useful when variable length bulk packets are being read out by the master, and the master needs to know how many bytes to read from the slave.	true, false

If the timeout feature is included in the driver build, a function timer run-time configuration also happens- but it is handled automatically by the driver when the I2C port is opened. The function timer runtime configuration structure (**tFunctionTimer**) is outlined below for completeness, but application does not need to know any of the function timer details.

Member	Description	Valid Values
ui16FunctionDelay_A	This member specifies the length of the delay (in timer clock cycles) before function A is called.	0x0000 to 0xFFFF
bool (*pbFunction_A)(void)	pbFunction_A is a function pointer to function A.	A valid function pointer, else null.
ui16FunctionDelay_B	This member specifies the length of the delay (in timer clock cycles) before function B is called.	0x0000 to 0xFFFF
bool (*pbFunction_B)(void)	pbFunction_B is a function pointer to function B.	A valid function pointer, else null.

#### 9.8.6.6 Using the Driver: Opening and Closing the Driver

Opening and closing of the I2C slave driver is accomplished through the following function calls:

Description	Declaration
Open the I2C Slave Port	extern void <b>I2CSlave_openPort</b> (const tI2CSlavePort *pPort)
Close the I2C Slave Port	extern void <b>I2CSlave_closePort</b> (void)

The driver is initialized and made ready for use by placing a call to **I2CSlave\_openPort()**, which is passed a completed tI2CSlavePort structure. The tI2CSlavePort structure must be populated by the application. This structure is not modified by the driver at any time, and as such, it may be placed in read-only memory such as a C const memory section. It is important that the structure be left in memory at the original address that is passed to **I2CSlave\_openPort()**, as the driver will reference this structure to access callback functions when the port is open. If the memory must be freed, first close the driver with a call to **I2CSlave\_closePort()**. A call to **I2CSlave\_closePort()** will disable the driver and its associated eUSCI\_B peripheral, halting all I2C slave activity. If the timeout feature was included in the build, **I2C\_closePort()** disables the function timer module that drives the

---

timeout feature. After the port is closed, the event handlers will no longer be called and the `ti2cSlave` structure memory may be released to the application.

### Open the I2C Slave Port

```
//  
// Open the I2C slave port  
/  
I2CSlave_openPort (&g_myI2CPort);
```

#### 9.8.6.7 Using the Driver: Transmitting Data (I2C Read Operation)

Data transmission is accomplished through the following function calls:

Description	Declaration
Set Transmit Buffer	extern void <b>I2CSlave_setTransmitBuffer</b> (uint8_t *pBuffer, uint16_t ui16Length)
Set Request Flag	extern void <b>I2CSlave_setRequestFlag</b> (void)
Get Port Status	extern uint8_t <b>I2CSlave_getPortStatus</b> (void)

An I2C slave cannot push data out onto the bus on its own- rather, a master must address the slave and clock out the data. Therefore, the data buffer must be made available to the driver before the bus master attempts to read it. The application may pass a pointer and a length (in bytes) to the `I2CSlave_setTransmitBuffer()` function. This function will block if a current transaction is in progress, and will return when the transmit buffer pointer and length have been updated. If no transmit buffer is provided and the bus master attempts to read from the slave, it will read out the invalid character (a compile-time option).

The `I2CSlave_setTransmitBuffer()` function does not perform a copy of the buffer- rather, it just stores its location and length. This is very valuable for "register" applications, where the application just updates the buffer, and `I2CSlave_setTransmitBuffer()` only needs to be called once during initialization. Repeated reads from the bus master will re-read the same buffer until `I2CSlave_setTransmitBuffer()` is called again. On the flipside, it is important that the application does not overwrite the transmit buffer space until transmission is complete.

If the slave request feature is enabled, the master may be signaled by calling the `I2CSlave_setRequestFlag()` function. This function immediately pulls the request line, then waits for the master to begin performing a read operation before it returns (or times out).

The driver state may be obtained by the application at any time by calling `I2CSlave_getPortStatus()`. This function returns the current I2C Slave state. The possible states are enumerated by `ti2cSlaveStates`. The possible enumerations are listed below.

Port Status Option	Description
eI2CSlavesClosed	The driver is closed. All functions besides <code>I2CSlave_openPort()</code> and <code>I2CSlave_getPortStatus()</code> are invalid.
eI2CSlavesIdle	The driver is open, but no I2C transactions are currently in progress.
eI2CSlavesBeingRead	The driver is open and the bus master is reading data. The driver loads data from the transmit buffer until all data has been sent, then it loads the invalid byte.
eI2CSlavesBeingWrittenTo	The driver is open and the bus master is writing data. The written data is placed into the receive buffer if there is space. At the end of this state, the receive callback in the application is called.

### Setting the I2C Slave Transmit Buffer

```
uint8_t g_ui8MemoryArray[MEMORY_SIZE] = {0};  
I2CSlave_setTransmitBuffer(g_ui8MemoryArray, MEMORY_SIZE);
```

#### 9.8.6.8 Using the Driver: Receiving Data (I2C Write Operation)

The bus master may start a write operation at any time. The driver will buffer incoming data into the receive buffer. When the transaction is complete, the application receive callback is called. The callback is passed the size of the

---

data received from the master (in bytes). Since it is called from the driver interrupt service routine, no other interrupts will be processed and the eUSCI\_B may be stretching the bus clock line until the driver returns from the receive callback. This provides the callback function the opportunity to process any received data and update the transmit buffer before the master may continue with communication.

#### 9.8.6.9 Using the Driver: Error Handling

The driver provides an error callback to alert the application if there is a problem with the driver. The error callback, when called, passes a value indicating the error that occurred. The possible errors are listed below.

Error Code	Description
eI2CSlaveTransmitRequestTimeout	This code indicates that a slave request to the bus master was not serviced within the timeout window, and the request timed out.
eI2CSlaveTransactionTimeout	This code indicates that an I2C transaction was taking longer than the timeout window, and the transaction timed out. This error also results in a driver reset.
eI2CSlaveReceiveBufferFull	This code indicates that the receive buffer was full during the last read transaction, and data from the bus master was lost.
eI2CSlaveWasReadBeyondTransmitBuffer	This code indicates that the bus master attempted to read data from the slave beyond the valid transmit buffer length (indicating the master was reading invalid bytes).

#### 9.8.6.10 Using the Driver: Example Application

This is an I2C Slave Driver 512B Embedded Memory IC example. This example project configures the MSP430FR2633 as a 512B RAM device with an I2C interface. The master may write up to 128 bytes to the 8kB memory at a time. The master may read from any memory address until the end of the memory section at any time. The read/write format is as follows: I2C-START / ADDRESS+RW / MEMORY ADDR UPPER BYTE / MEMORY ADDR LOWER BYTE / DATA.

```
#include "driverlib.h"
#include "I2CSlave.h"
#include "string.h"

// 
// MEMORY_SIZE defines the size of the memory array in bytes
//
#define MEMORY_SIZE (512)

// 
// g_ui8MemoryArray is the memory array.
//
uint8_t g_ui8MemoryArray[MEMORY_SIZE] = {0};

// 
// I2C_RECEIVE_BUFFER_SIZE defines the size of the I2C receive buffer
//
#define I2C_RECEIVE_BUFFER_SIZE 130

// 
// g_ui8I2CReceiveBuffer is the buffer space used for I2C receive ops.
//
uint8_t g_ui8I2CReceiveBuffer[I2C_RECEIVE_BUFFER_SIZE];

// 
// REQ_LOW_ADDR defines a macro to get the low address byte in the I2C
// receive buffer.
//
// REQ_HIGH_ADDR defines a macro to get the high address byte in the I2C
// receive buffer.
//
// REQ_FULL_ADDR defines a macro to access the full 13-bit address in
// the receive buffer
//
// REMAINING_MEMORY defines macro to get the remaining memory from the
// current address in the receive buffer to the end of memory.
```

---

```

// DATA_TO_WRITE defines a macro to get a pointer to the data to write
// in the I2C receive register
//
#define REQ_LOW_ADDR           (g_ui8I2CReceiveBuffer[1])
#define REQ_HIGH_ADDR          (g_ui8I2CReceiveBuffer[0])
#define REQ_FULL_ADDR ((uint16_t)REQ_LOW_ADDR | ((uint16_t)REQ_HIGH_ADDR << 8))
#define REMAINING_MEMORY        (MEMORY_SIZE-REQ_FULL_ADDR)
#define DATA_TO_WRITE           (&g_ui8I2CReceiveBuffer[2])

//
// The receive handler is called by the I2C port driver whenever a new
// packet was received.
//
bool receiveHandler(uint16_t ui16Length)
{
    //
    // Update the current memory location
    //
    uint8_t *pCurrMemoryLocation = &g_ui8MemoryArray[REQ_FULL_ADDR];
    I2CSlave_setTransmitBuffer(pCurrMemoryLocation, REMAINING_MEMORY);

    //
    // If there is data in the packet, write it into the memory
    //
    if (ui16Length > 2)
    {
        memcpy(pCurrMemoryLocation, DATA_TO_WRITE, (ui16Length-2));
    }

    //
    // No need to wake the CPU, return false to exit asleep
    //
    return false;
}

void errorHandler(uint8_t ui8Error)
{
    return;
}

//
// g_myI2CPort specifies the I2C Slave port configuration that is passed
// to I2CSlave_openPort().
//
const ti2cslaveport g_myI2CPort =
{
    .pbReceiveCallback = &receiveHandler,
    .pvErrorCallback = &errorHandler,
    .ui16ReceiveBufferSize = I2C_RECEIVE_BUFFER_SIZE,
    .pReceiveBuffer = g_ui8I2CReceiveBuffer,
    .bSendReadLengthFirst = false
};

void main(void)
{
    WDT_A_hold(WDT_A_BASE);
    PMM_unlockLPM5();

    //
    // Set P1.6 and P1.7 to UCB0SDA and UCB0SCL, respectively
    //
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P1, GPIO_PIN2, GPIO_SECONDARY_MODULE_FUNCTION);
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1, GPIO_PIN3, GPIO_SECONDARY_MODULE_FUNCTION);

    //
    // Enable masked interrupts
    //
    __bis_SR_register(GIE);

    //
    // Open the I2C slave port
    //
    I2CSlave_openPort(&g_myI2CPort);
    I2CSlave_setTransmitBuffer(g_ui8MemoryArray, MEMORY_SIZE);

    //
    // Application background loop
    //
    while(1)
    {
        //
        // Enter LPM3
        //
        LPM3;
    }
}

```

---

## 9.9 Benchmarks

This section contains benchmarks for the CapTlivate™ Software Library as used in different applications.

### 9.9.1 Memory Requirements

The CapTlivate Software Library is just one component of an application. At a minimum, an application will also require the following:

- MCU port muxing configuration (requires code space)
- MCU clock configuration (requires code space)
- CapTlivate configuration (requires code space)
- A main routine (requires code space)
- A call stack (requires data space)

Therefore, when selecting the correct device it is important to consider the memory needs for the entire application and not just for CapTlivate.

#### CODE (FRAM) and DATA (RAM) Requirements

The following table illustrates the typical memory requirements for several different use-cases of the CapTlivate Software Library to aid in selecting the correct device for an application.

For each of these examples, the following assumptions were taken:

- CapTlivate Software Library and Starter Project v1.04.00.00 (September 2016)
- A 256B call stack (CapTlivate examples default to a 256B call stack for application safety, this may be reduced)
- MCU initialization via the CAPT\_BSP module (configures GPIO and clocks)
- CCS Toolchain (Compiler 15.12.1.LTS)
- Optimization set to level 2 / size

Sizes are shown with and without CapTlivate Design Center UART COMM support enabled. Enabling UART communications adds 1664B of FRAM and 389B of RAM with the default CapTlivate COMM configuration.

Example Case	Sensor Configuration	FRAM (B)	RAM (B)	FRAM (B) w/COMM	RAM (B) w/COMM	Recommended IC	Comments
Basic touch switch	1 button	4200	358	5864	747	MSP430FR2532	
Basic touch switch with wake-on-prox	1 button	4634	361	6298	750	MSP430FR2532	When adding wake-on-touch, FRAM increases.

Basic touch switch with noise immunity	1 button	6594	384	8258	773	MSP430FR2532	When adding noise immunity, RAM and FRAM increase. Some optimizations are required when communications are enabled.
Numeric Keypad UI	12 buttons	4356	722	6020	1111	MSP430FR2532	Main increase when adding buttons is RAM.
Numeric Keypad UI with wake-on-prox	12 buttons, 1 prox	4850	799	6514	1188	MSP430FR2532	
Numeric Keypad UI with noise immunity	12 buttons	6750	946	8414	1335	MSP430FR2633	Some optimizations are required when communications are enabled.
32 button UI	32 buttons	4678	1392	6342	1781	MSP430FR2533	>16 buttons requires an FR2533 or FR2633
32 button UI with noise immunity	32 buttons	7074	1976	8738	2365	MSP430FR2633	
64 button UI	64 buttons	5178	2464	6842	2853	MSP430FR2633	
64 button UI with noise immunity	64 buttons	7570	3624	9234	4013	MSP430FR2633	

#### Stack Requirements

The worst-case stack usage for most applications running only the CapTlve Touch Library is approximately 100B, with 30B of additional space required for interrupt service routines. Therefore, applications should have a minimum stack size of 100B+30B=130B. If callback functions are used, these may potentially increase the stack size requirement. A good, conservative estimate that is safe for many applications is a 256B stack. Every application will have its own unique stack requirements, with interrupt service routines being the largest unknown factor. Therefore it is recommended that each designer perform a worst-case stack usage analysis to determine the required stack size.

## 9.9.2 Execution Times

It is often of interest to understand the amount of time required to set up a measurement, perform the measurement, and process the measurement results. This total time (setup+measure+process) determines the overall scan rate that is achievable for a system. This section is intended to provide general guidance on typical execution times for the setup+measure+process sequence.

### Top Level - CAPT\_updateUI()

**CAPT\_updateUI()** is the top level user-interface update application. It handles the setup+measure+process sequence for all sensors in the user interface, and will transmit data via a serial interface if configured to do so. This is the easiest function to use as it includes all of the required processing. Below are several examples of execution times for the setup+measure+process sequence.

Test conditions:

- One button, one time cycle, one sensor
- CAPTIVATE-BSPW demo panel
- 250 counts of measurement resolution at a 2 MHz conversion frequency for a 145us measurement time.
- CPU frequency is 8 MHz

Function	Total Time	Setup Time	Measurement Time	Processing Time	Notes
CAPT_updateUI(&g_uiApp);	0.723ms	0.388ms	0.145ms	0.190ms	LPM3 during conversion, bLpmControl=false
CAPT_updateUI(&g_uiApp);	0.632ms	0.297ms	0.145ms	0.190ms	LPM3 during conversion, bLpmControl=true
CAPT_updateUI(&g_uiApp);	0.626ms	0.297ms	0.145ms	0.184ms	LPM0 during conversion, bLpmControl=true

In the first case, the CapTlve peripheral was configured to turn off after each conversion. This means that before each conversion starts, there is an extra start-up time for the peripheral. Here that time measures to be approximately 0.1ms. Also note the difference between LPM3 and LPM0 in the second and third cases. It takes up to 0.010ms to wake up from LPM3 to active, whereas the LPM0 to active transition is less than 0.001ms.

### Sensor Level - CAPT\_updateSensor()

**CAPT\_updateSensor()** is the sensor-level update function. It handles the setup+measure+process sequence for a single sensor at time. It does not transmit data. This is an easy function to use if you want to control when a specific sensor is measured.

Test conditions:

- One button, one time cycle, one sensor
- CAPTIVATE-BSPW demo panel
- 250 counts of measurement resolution at a 2 MHz conversion frequency for a 145us measurement time.
- CPU frequency is 8 MHz
- LPM0 during conversion, bLpmControl=false

Function	Total Time	Setup Time	Measurement Time	Processing Time	Notes
CAPT_updateSensor(&BTN00, CPUOFF);	0.603ms	0.293ms	0.145ms	0.165ms	Full element and sensor processing performed
CAPT_updateSensorRawCount(&BTN00, eStandard, eNoOversampling, CPUOFF);	0.514ms	0.296ms	0.145ms	0.073ms	No element or sensor processing performed

#### Low Level - CAPT\_startConversionAndWaitUntilDone()

Low level functions may be used to extract raw data. This is the most time efficient way to measure if there is only one sensor, because the sensor's parameters do not need to be re-loaded each time it is measured. This type of low-level conversion is only valuable for systems that have only 1 time cycle and are only looking to extract raw data with no post-processing.

The sequence to set up the sensor for measurement is as follows (run once):

- MAP\_CAPT\_applySensorParams(&BTN00);
- MAP\_CAPT\_applySensorFreq(CAPT\_OSC\_FREQ\_DEFAULT, &BTN00);

The sequence to load the cycle, run, and extract the cycle's measurement data is as follows (run for each measurement):

- CAPT\_loadCycle(&BTN00, 0, CAPT\_OSC\_FREQ\_DEFAULT, false);
- CAPT\_startConversionAndWaitUntilDone(CPUOFF);
- CAPT\_unloadCycle(&BTN00, 0, CAPT\_OSC\_FREQ\_DEFAULT, false);

Function	Total Time	Setup Time	Measurement Time	Processing Time	Notes
Low Level	0.243ms	0.066	0.145	.032	3 function calls

# Chapter 10

## MSP-CAPT-FR2633 Development Kit

### 10.1 Introduction

Capacitive touch evaluation and rapid prototyping has never been faster and easier. Using this modular kit with the CapTIvate™ Design Center, you can evaluate the performance of MSP430FR2633 MCU using the capacitive touch demo boards, or develop your own PCB and experience the power of the CapTIvate™ Design Center and the ease of real-time sensor tuning, all without writing a single line of code. Free software development tools are also available, including TI's Eclipse-based Code Composer Studio™ (CCS) and IAR Embedded Workbench® IAR-KICKSTART. Both of these integrated development environments (IDEs) support EnergyTrace™ technology when paired with the MSP430FR2633 MCU Development Kit.

Read through the [overview](#) section to become familiar with all of the hardware in the kit, or jump to the [getting started](#) section to begin evaluation.

#### 10.1.1 Overview

The MSP-CAPT-FR2633 MCU Development Kit is an easy to use evaluation platform for the MSP430FR2633 microcontroller featuring CapTIvate™ Capacitive Touch Technology. It contains everything needed to start developing on the MSP430™ ultra-low-power (ULP) FRAM-based microcontroller (MCU) platform, including on-board emulation for programming, debugging and energy measurements. The MSP430FR2633 MCU features a new CapTIvate™ Capacitive Touch Technology, supporting self and mutual capacitive sensors, and embedded FRAM (ferroelectric random access memory). FRAM is a non-volatile memory known for its ultra-low power, high endurance, and high-speed write access.

#### 10.1.2 Key Features

- Modular Design
  - Different sensing panels may be attached through a common connector.
  - The programming/debug logic has been split from the target MCU, unlike a typical LaunchPad which has both components on one PCB. This allows for an isolation module to be inserted between the two for testing/tuning, or for the programming/debug module to be used for in-system product development.
  - The target MCU PCB features BoosterPack headers, allowing it to be used as a BoosterPack for another LaunchPad, such as the MSP-EXP432P401R LaunchPad.
- Application Oriented
  - The sensing panels included in the kit are designed to mimic real applications.
  - Evaluating a new application simply involves laying out a basic 1 or 2 layer PCB with a footprint for the sensor connector, or simply laying out electrodes with copper tape and wiring them into the connector.
- Easy Access to Data

- EnergyTrace power profiling enables capturing of power profiles without any measurement equipment.
- The HID-Bridge communications interface allows easy debug data transfer between the target and a PC over UART or I2C.

### What's Included

The following PCBs are included in the kit:

- **CAPTIVATE-FR2633** Target MCU Module
- **CAPTIVATE-PGMR** eZ-FET with EnergyTrace and HID Communication Bridge
- **CAPTIVATE-ISO** UART, I2C, and SBW Isolation Board
- **CAPTIVATE-BSWP** Self Capacitance Demo (Out-of-Box Experience)
- **CAPTIVATE-PHONE** Mutual Capacitance Demo with Haptics and Guard Channel
- **CAPTIVATE-PROXIMITY** Proximity Detection and Gesturing Demo

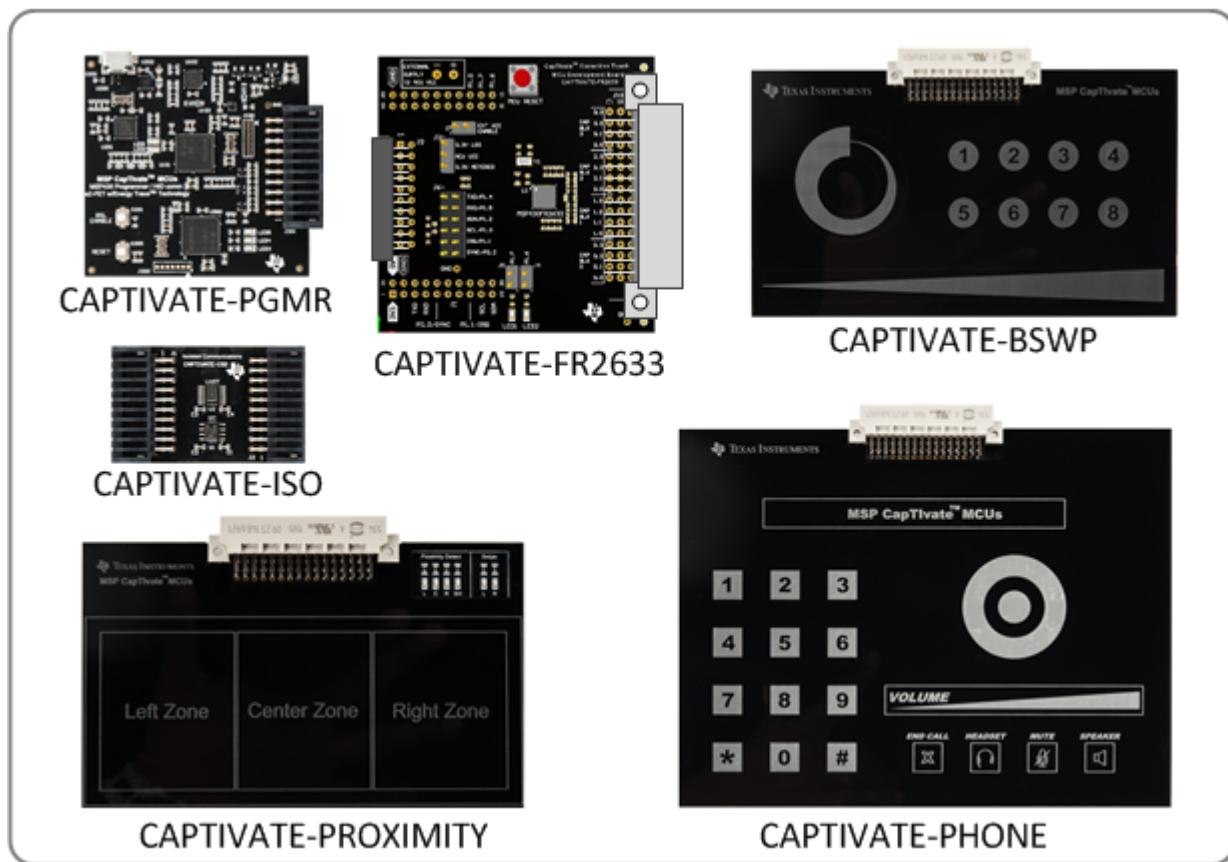


Figure 10.1: Kit Contents

### Additional CapTivate™ Demo Boards

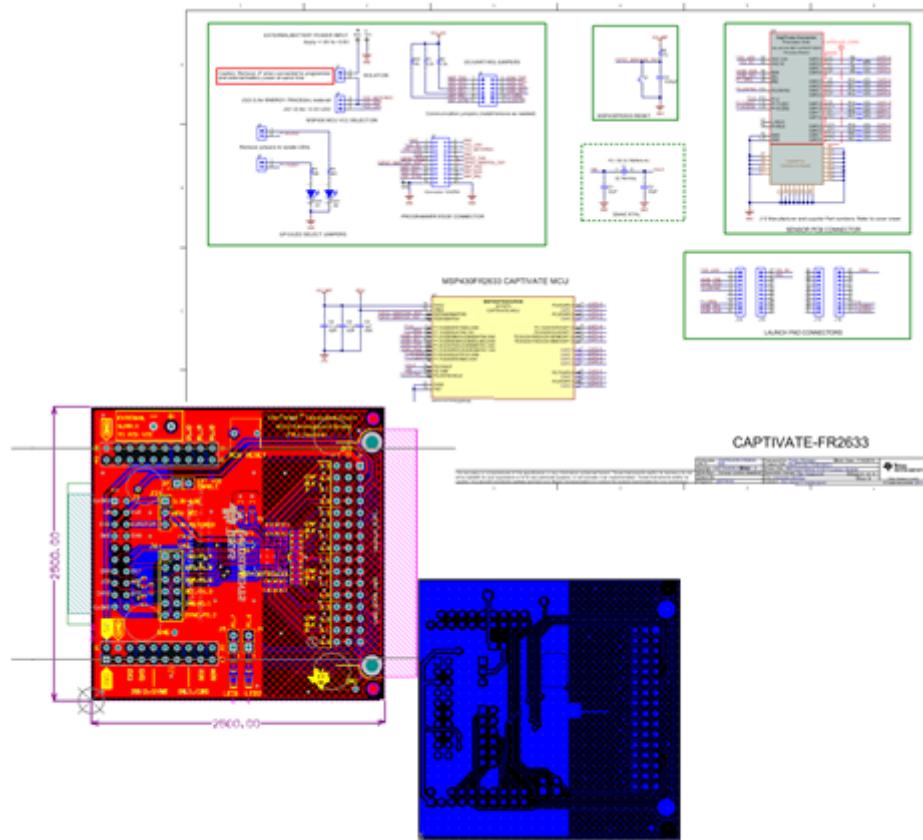
The following PCBs will be available separately, but are compatible with this kit:

### Software Examples

Complete CCS and IAR example projects for each demo panel are provided with the CapTivate™ Design Center installation. Example projects are also available with the installation of MSPWare.

## Design Files

Schematic and layout design files for all of the PCBs in the kit are available below.



- Download CapTIvate™ EVM Design Files Here:

## 10.2 Getting Started with the MCU Development Kit

This section details how to get started with the development kit.

### 10.2.1 Pre-Work

To get started with the kit, complete the following software and hardware steps:

#### Software Installation

1. Install the CapTIvate™ Design Center GUI on your PC, Linux, or MAC computer (note the [requirements](#)).
2. Install a development environment (IDE). TI's Code Composer Studio v6.1.0 and IAR Embedded Workbench v6.30 or greater are supported.

#### Hardware Setup

1. Connect the CAPTIVATE-FR2633 MCU module and CAPTIVATE-PGMR module together.
2. Connect the desired sensing panel to the CAPTIVATE-FR2633 module. The out-of-box experience uses the CAPTIVATE-BSPW panel.

3. Connect the micro-USB cable between the CAPTIVATE-PGMR programmer PCB and your computer
4. Verify that LED2 and LED5 (power good LED's) on the CAPTIVATE-PGMR module are lit, and that LED4 (HID-Bridge enumeration) is blinking.

Refer to the [FAQ](#) section for further troubleshooting tips.

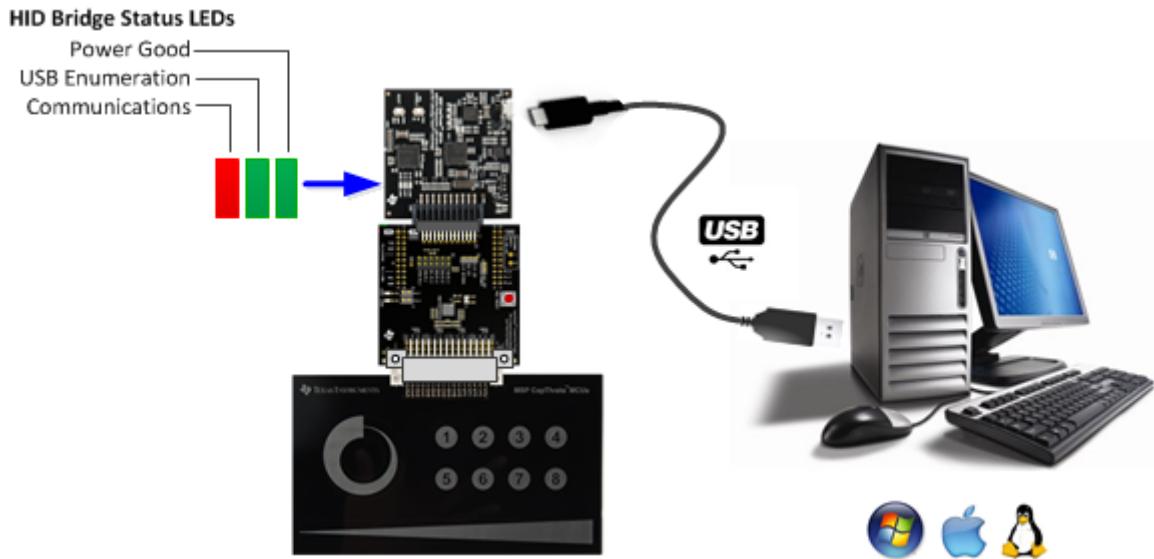


Figure 10.2: Typical Setup

## Out-of-Box Experience

Be sure that you have completed the [pre work](#). The CAPTIVATE-FR2633 module ships pre-programmed with the demo software for the CAPTIVATE-BSWP panel. It is not necessary to program the target device to begin evaluating with that panel. Check out the [workshop chapter](#) for the out-of-box experience to begin working with the kit!

### 10.2.2 Running an Example Project

Be sure that you have completed the [pre work](#). To run an example project, it is necessary to do the following:

1. If it is not already connected, connect the desired sensing panel (CAPTIVATE-BSWP, CAPTIVATE-PHONE, CAPTIVATE-PROXIMITY) to the CAPTIVATE-FR2633 module.
2. Import, build, and program the example software for the specific board that you are using onto the MSP430FR2633 target MCU on the CAPTIVATE-FR2633 module. Follow these steps to [load and run generated firmware projects](#). [Example project locations](#) are described below. **Note** that the CAPTIVATE-FR2633 module ships pre-programmed for the CAPTIVATE-BSWP sensing panel.
3. Start the CapTlve™ Design Center using the icon that was installed on your desktop.
4. Click *File -> Project Open* from the menu bar and select the desired project directory. [Click here for help with CapTlve™ Design Center](#). The project's [main window](#) is the design canvas and is pre-populated with the microcontroller and selected sensors matching the hardware for this demo.
5. [Enable Communications](#) in the CapTlve™ Design Center to begin the demonstration.
6. View the data for different sensors when you approach and touch the panel. Note: If the MCU is in wake-on-proximity mode, there may appear to be no activity. Bring your hand near or touch the board to activate the MCU.

### 10.2.3 Example Project Locations

During the CapTlve™ Design Center installation process, example projects and MSP430FR2633 firmware projects are placed in their own directory in the user's home directory under "CapTlveDesignCenter/CapTlveDesignCenterWorkspace". On Windows 7, this would be C:/Users/"username"/CapTlveDesignCenter/CapTlveDesignCenterWorkspace.

## 10.3 Sensor Panel Demonstrations

This section describes each of the sensor panel demonstrations. For details on the hardware for each panel, visit the [hardware](#) section.

### 10.3.1 CAPTIVATE-BSPW Demonstration (Out of Box Experience)

The CAPTIVATE-BSPW sensing panel demonstrates the low power and high sensitivity features of the CapTlve™ technology. The wake-on-proximity state machine is utilized to measure the proximity sensor in LPM3 until a user is close to the panel. When a user is detected, all of the sensors begin scanning at a faster rate. Data is communicated to the CapTlve™ Design Center via the bulk I2C interface.

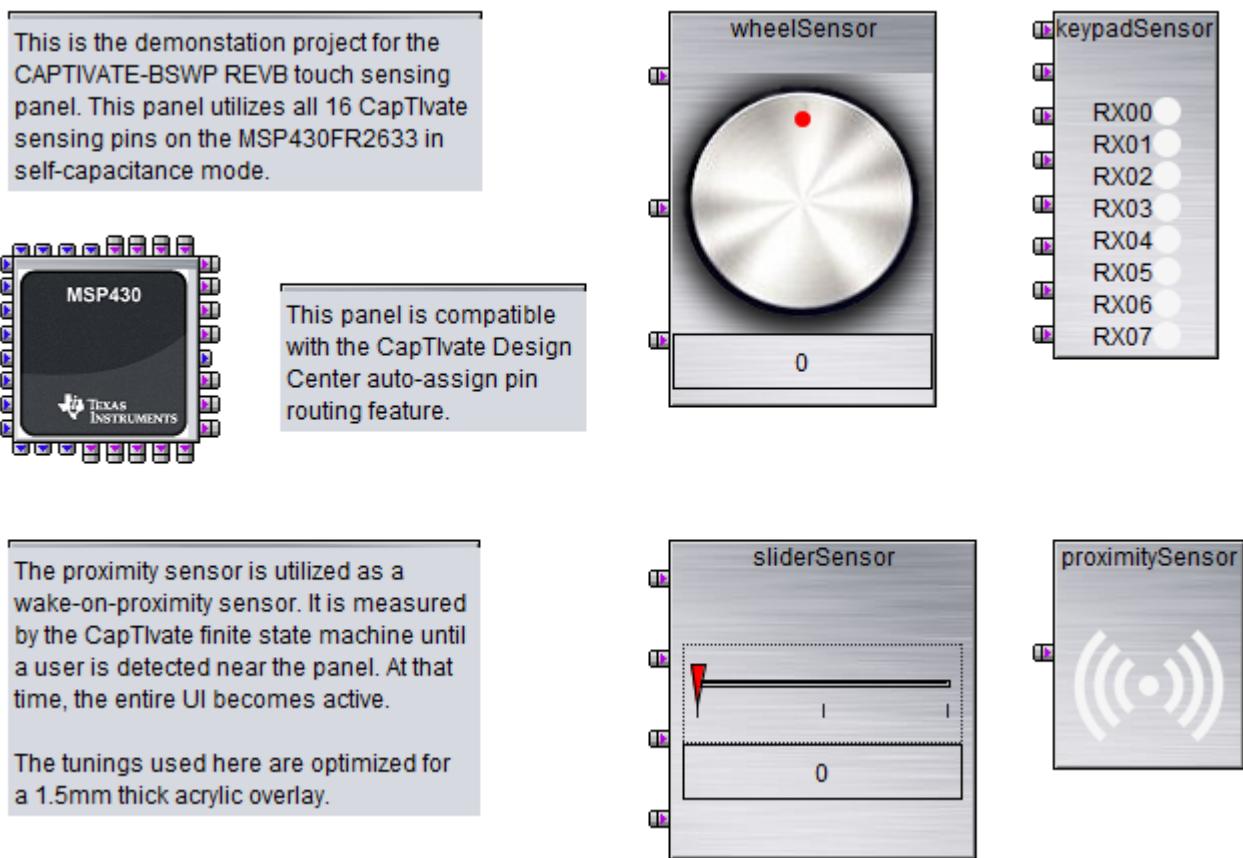


Figure 10.3: CAPTIVATE-BSPW Design Canvas

This panel is configured with the following settings:

- **A 33ms active mode scan period (30 Hz).** This provides a balance between response time and power consumption when a user is interacting with the panel. Scanning faster (at 20ms/50Hz for example) would provide a faster response time and also has a perceived performance benefit when working with sliders and

---

wheels. Scanning slower (at 50ms/20Hz for example) would provide lower power consumption, but a higher response time.

- **An inactivity timeout of 32 samples.** This means that the system will transition from active mode into wake-on-proximity mode after 32 samples without any interaction have passed. This works out to about a 1 second delay. This aggressive transition improves battery life.
- **A 100ms wake-on-proximity mode scan period (10 Hz).** This slower scan rate is designed to conserve power while still waking up the panel fast enough to be ready for an approaching user.
- **A wakeup interval of 512 samples.** This means that the system will wake from the low-power wake-on-proximity state every 512 samples (about every 50 seconds) even if there is not a proximity detection. This provides a mechanism to periodically update the environmental filters for the other sensors on the PCB.
- **A 2 MHz conversion frequency** for all sensors.
- **A total measurement time of 2ms** for all sensors.

To begin working with this panel, go through the steps for [running an example project](#) and open the CAPTIVATE-BSPW project.

For an overview of the hardware design, visit the [CAPTIVATE-BSPW hardware overview](#).

#### 10.3.1.1 CAPTIVATE-BSPW Bonus Software Projects

In addition to the out-of-box software project, the CAPTIVATE-BSPW panel is also used as the hardware platform for other examples that show off different features of the CapTlve peripheral and MSP430FR2633 MCU.

##### 10.3.1.1.1 Ultra Low Power Four Buttons Demonstration

This supplementary example project uses the first 4 buttons on the CAPTIVATE-BSPW panel only. It demonstrates how to achieve extremely low power consumption for a design that only contains 4 buttons.

This project is configured with the following settings:

- **An 83ms active mode scan period (12 Hz).**
- **An inactivity timeout of 32 samples.** This means that the system will transition from active mode into wake-on-proximity mode after 32 samples without any interaction have passed.
- **A 125ms wake-on-proximity mode scan period (8 Hz).** This slower scan rate is designed to optimize battery life.
- **A wakeup interval of 2048 samples.**
- **A 2 MHz conversion frequency** for all sensors.
- **A total measurement time of 260us** for all sensors.
- **An I-avg of 3uA when no one is touching the panel**

To begin working with this panel, go through the steps for [running an example project](#) and open the UltraLowPower\_FourButtons project.

For a step-by-step workshop of how to reproduce this project from scratch, see the [low power design example](#) section of the design guide.

---

#### 10.3.1.1.2 Code Size Optimized One Button Demonstration

This supplementary example project uses the first button on the CAPTIVATE-BSWP panel only. It demonstrates a bare-bones touch application that just controls the state of the two LEDs on the MCU module. When the button is pressed, LED1 will illuminate and stay lit the entire time the button is pressed. At the beginning of each new button press, LED2 will change its state.

This example serves as a reference for how to get down to a basic set of function calls. It achieves this by removing the CAPT\_App layer that comes with the other example projects, and replacing it with register-level initialization of the MCU and top-level API calls directly into the CapTivate library. This organization does not have support for the wake-on-proximity features, but it does have considerably reduced complexity as well as reduced memory requirements. This project easily fits into a small 8kB FRAM / 1kB RAM device variant.

Note: No communications module is included in the project build, so the only feedback will be visual via the LEDs!

The CapTivate setup code is very basic, as shown below:

```
//  
// Set up the CapTivate peripheral  
// and timer. The timer will periodically  
// set the g_bConvTimerFlag.  
//  
CAPT_initUI(&g_uiApp);  
CAPT_calibrateUI(&g_uiApp);  
MAP_CAPT_registerCallback(&button, &buttonHandler);  
MAP_CAPT_selectTimerSource(CAPT_TIMER_SRC_ACLK);  
MAP_CAPT_writeTimerCompRegister(CAPT_MS_TO_CYCLES(g_uiApp.ui16ActiveModeScanPeriod));  
MAP_CAPT_startTimer();  
MAP_CAPT_enableISR(CAPT_TIMER_INTERRUPT);
```

A simple background loop calls CAPT\_updateUI directly and manages the use of the conversion timer interrupt flag variable.

```
//  
// Start the application  
//  
while(1)  
{  
    if (g_bConvTimerFlag == true)  
    {  
        //  
        // If it is time to update the button,  
        // update it here with the generic library call.  
        //  
        g_bConvTimerFlag = false;  
        CAPT_updateUI(&g_uiApp);  
    }  
  
    //  
    // Go to sleep when finished.  
    // The CapTivate timer will wake the application from sleep.  
    //  
    __bis_SR_register(g_uiApp.ui8AppLPM | GIE);  
}
```

A basic callback sets the LED states whenever the button is updated:

```
void buttonHandler(tSensor* pSensor)  
{  
    //  
    // If the button is currently being touched,  
    // illuminate LED1.  
    //  
    if (pSensor->bSensorTouch==true)  
    {  
        P1OUT |= LED1;  
  
        //  
        // If a completely new touch was detected,  
        // toggle the state of LED2.  
        //  
        if (pSensor->bSensorPrevTouch==false)  
        {  
            P1OUT ^= LED2;  
        }  
    }  
    else
```

---

```
{  
    P1OUT &= ~LED1;  
}  
}
```

This project is configured with the following settings:

- **An 50ms active mode scan period (20 Hz).**
- **A 2 MHz conversion frequency** for all sensors.
- **A total measurement time of 260us** for the button.
- **An I-avg of 10uA**

To begin working with this panel, go through the steps for [running an example project](#) and open the CodeSizeOptimized\_OneButton project.

### 10.3.2 CAPTIVATE-PHONE Demonstration

The CAPTIVATE-PHONE sensing panel demonstrates the use of mutual capacitance to realize a high-density panel with many different sensor types using just 12 of the 16 CapTivate™ sensing pins. The panel is designed to mimic a typical office phone application that would have a 12-key numeric keypad, several mode buttons, and several selection sensors. The panel features haptic vibration feedback thanks to a TI DRV2605L haptic driver IC coupled with a Samsung linear resonant actuator (LRA). A guard channel technique is applied to reject palm/arm presses on buttons as well as minor liquid spills. Data is communicated back to the CapTivate™ Design Center via a UART interface.

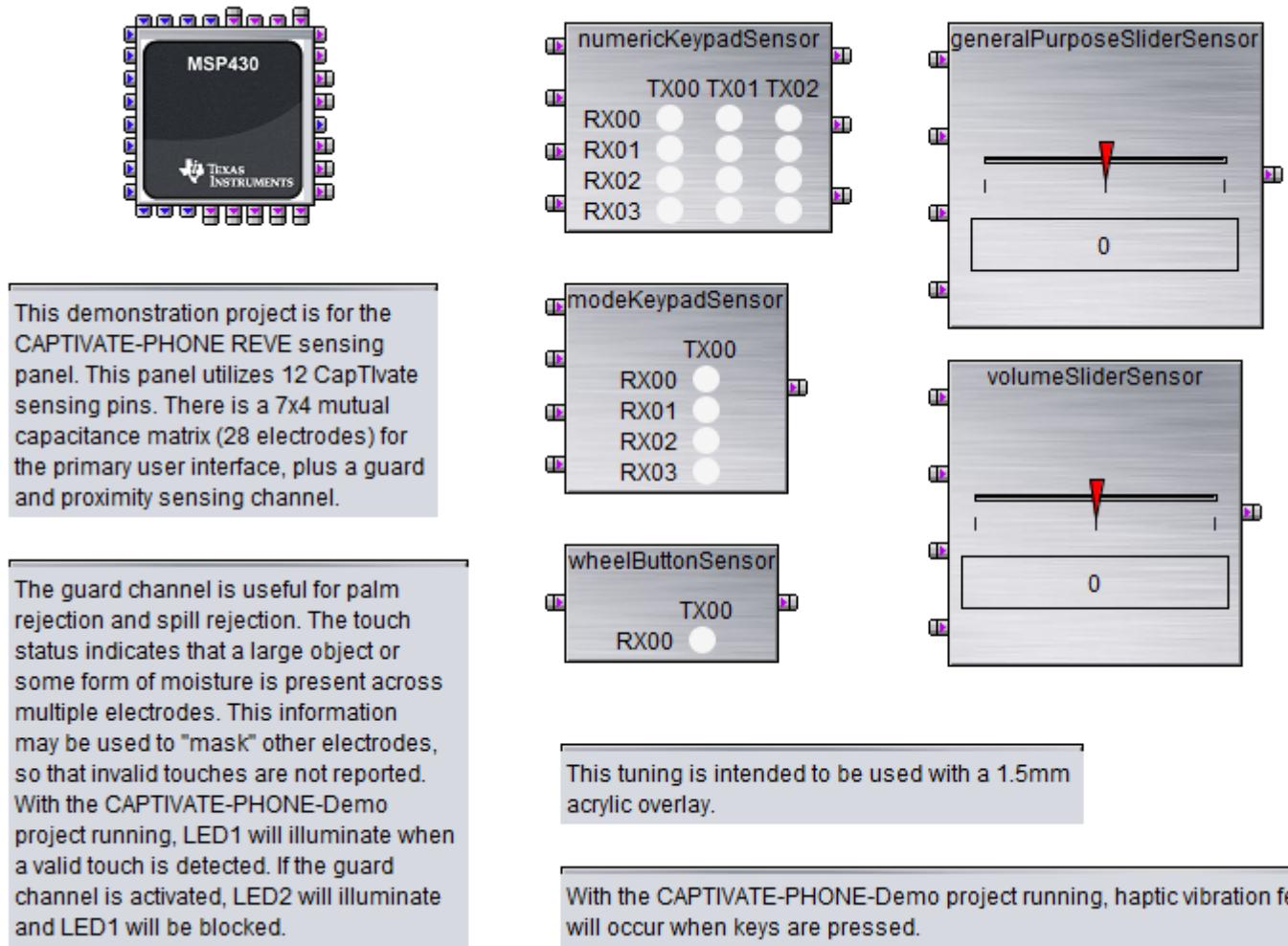


Figure 10.4: CAPTIVATE-PHONE Design Canvas

---

This panel is configured with the following settings:

- **A 33ms active mode scan period (30 Hz).** This provides a balance between response time and power consumption when a user is interacting with the panel. Scanning faster (at 20ms/50Hz for example) would provide a faster response time and also has a perceived performance benefit when working with sliders and wheels. Scanning slower (at 50ms/20Hz for example) would provide lower power consumption, but a higher response time.
- **A 4 MHz conversion clock rate** for the mutual capacitance matrix, as mutual capacitance sensors may be scanned at a higher frequency.
- **A 1 MHz conversion clock rate** on the self-capacitance guard channel, as there is significant ground loading on this sensor, requiring a slower frequency.
- **A total measurement time of <2.4ms** for all sensors.

### Haptic Feedback

This demonstration includes haptics to provide users with mechanical feedback that they did in fact touch a key. The most common and cost-effective actuators are the ERM (eccentric rotating mass) and LRA (linear resonant actuator). The LRA was chosen for this application because it provides a higher quality vibration feel than the ERM. For this reason, LRA's are more common with consumer products. However, there are some advantages to an ERM that are worth discussing here. LRAs have a limited lifetime of "clicks," and thus are not as suitable for long life cycle products as an ERM. Therefore, and ERM makes more sense than an LRA for long-life products such as industrial control panels.

With regard to the driver, The DRV2605L was selected as the driver IC for this demonstration for the following reasons:

- It has integrated ROM effect libraries that are pre-licensed from Immersion
- It supports both ERM and LRA haptic actuators
- It has a simple I2C register interface

The demonstration firmware includes an I2C master driver and a DRV26x driver for communicating with the DRV2605L. Setting up the DRV2605L with these modules is accomplished in the Demo\_init() function as follows:

```
//  
// Open the I2C Master driver, which the DRV26x driver will use to  
// communicate with the DRV2605L haptic driver IC via I2C.  
// Enable the haptic driver by setting P1.0, which is connected  
// to the DRV2605L ENABLE pin. Then, load the configuration for the  
// actuator, run an auto-calibration routine, set up for internal trigger  
// mode, and select the linear resonant actuator (LRA) effect library.  
//  
I2CMaster_open();  
P1OUT |= BIT0;  
DRV26x_reset();  
DRV26x_exitStandby();  
DRV26x_loadActuatorConfig(&DRV26x_actuator_DMJBRN1030);  
DRV26x_runAutoCalibration();  
DRV26x_setMode(DRV26x_mode_internalTrigger);  
DRV26x_selectEffectLibrary(DRV26x_lib_ROM_LRA);
```

The **callback** capability of the CapTlve™ Software Library is utilized to trigger playback of haptic events directly from the library. A sample callback function is shown below. Effects are fired on a "new" touch- if "touch" is true and "previous touch" is false. Note that the callback exits if the guard mask is active. This would be the case if the guard channel was in detect, and the touch on this sensor must be masked.

```
void Demo_numericKeypadHandler(tSensor* pSensor)  
{  
    //  
    // If the guard mask is activated, abort here and do not process  
    // any events.  
    //
```

---

```

    if (Demo_guardMaskActive == true)
    {
        return;
    }

    //
    // If the sensor has a new touch, fire a "strong click" effect.
    //
    if ((pSensor->bSensorTouch == true)
        && (pSensor->bSensorPrevTouch == false))
    {
        DRV26x_fireROMLibraryEffect(
            DRV26x_effect_strongClick_100P,
            true
        );
    }
}

```

Below is a mapping of the various DRV2605L haptic effects that are mapped to the different sensors:

Sensor	Element	Touchdown Effect	Continued Effect
numericKeypadSensor	All	Strong Click (100%)	None
modeKeypadSensor	Mute Button	Double Click (100%)	None
modeKeypadSensor	Speaker Button	Strong Click (100%)	None
modeKeypadSensor	Headset Button	Strong Click (100%)	None
modeKeypadSensor	End Call Button	Triple Click (100%)	None
wheelButtonSensor	Button	Pulsing Sharp 1 (100%)	None
scrollWheelSensor	Button	Buzz 1 (100%)	Soft Bump (100%)
generalPurposeSliderSensor	Button	Buzz 1 (100%)	Soft Bump (100%)
volumeSliderSensor	Button	Buzz 1 (100%)	Soft Bump (100%)
proxAndGuardSensor	Button	Smooth Hum 3 (30%)	None

### Guard Channel Integration

The guard channel is used as a detection mask for all other sensors in the system to provide a level of palm/arm rejection and spill rejection. The guard electrode wraps around the panel between all of the other sensors, and when it is not being measured, it serves as a grounded shield. The data from the guard channel is used to discern the case where a user accidentally puts their whole hand against the sensing panel. In that case, it would be very undesirable for all of the sensors to go into detect. It is also helpful in the event that the panel surface is being wiped down with a cloth or a liquid is spilled onto the panel.

Guard channel tuning requires the following considerations:

1. Tune the touch threshold to be sensitive enough to trigger a detect when a user is in between keys, but not when a user is correctly touching just one sensor.
2. Set the touch **de-bounce in** parameter to '0', and the **de-bounce out** parameter to the maximum of '15'. This causes the guard channel to engage immediately in a detect situation, and to remain in detect for 15 samples even after the user has cleared the threshold. This improves the robustness of the mask.
3. Set the touch **de-bounce in** of all other sensors to at least a value of '1', to ensure that the guard channel mask has one sample to kick in, preventing false touch detections.
4. Test multiple use-cases and approach angles to the panel to ensure that the guard channel touch detection flag is being set before the other sensors. This is fairly easy to discern with this demonstration because of the haptic feedback.

The two LEDs on the CAPTIVATE-FR2633 module indicate the status of the guard channel. When a valid touch is detected on an element, LED1 will illuminate. When the guard channel is active, LED2 will illuminate.

The example callback handler for the guard channel is shown below.

```

void Demo_guardChannelHandler(tSensor *pSensor)
{
    //
}

```

---

```

// If the guard channel is detecting a touch,
// set the guard mask active flag to mask all other
// touch processing.
//
if (pSensor->bSensorTouch == true)
{
    Demo_guardMaskActive = true;
    if (pSensor->bSensorPrevTouch == false)
    {
        DRV26x_fireROMLibraryEffect(
            DRV26x_effect_smoothHum3_30P,
            false
        );
    }
}

// If the guard channel is not detecting a touch,
// clear the guard mask active flag to allow standard
// touch processing.
//
else
{
    Demo_guardMaskActive = false;
}

```

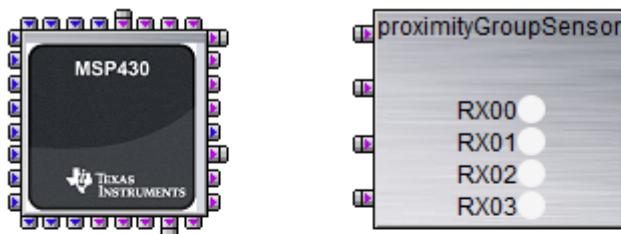
Note that the guard channel will not mask the reporting of touch and proximity events on other sensors to the CapTivate™ Design Center. Rather, the mask is application-level and is used to control the haptic effect playback and LED illumination. The data flowing back to the design center provides the true state of each sensor at all times.

To begin working with this panel, go through the steps for [running an example project](#) and open the CAPTIVATE-PHONE project.

For an overview of the hardware design, view the [CAPTIVATE-PHONE hardware overview](#).

### 10.3.3 CAPTIVATE-PROXIMITY Demonstration

This example demonstrates standard distance-detection proximity as well as proximity gesturing. In addition, it demonstrates the use of a driven shield sensor to reduce the parasitic capacitance of large electrodes, improving their sensitivity. The module has 4 total electrodes- 3 on the top layer, and one large electrode around the 3 on the top layer and across the entire bottom layer. The large electrode acts as a shield for parasitic capacitance, while the 3 primary electrodes on the top layer perform the sensing. 6 LED indicators on the top of the module indicate the status of the panel.



This is the demonstration project for the CAPTIVATE-PROXIMITY REVB panel. The design consists of 4 elements measured in parallel. Of the 4 elements, three are proximity interaction zones and the 4th is a driven shield (E1).

Proximity detection should occur 10-15cm above the panel. Waving a hand left or right 5-10cm above the panel will result in a swipe left or swipe right detection. The touch threshold is repurposed in this design as a "close proximity" threshold, which is used in the swipe determination algorithm.

Figure 10.5: CAPTIVATE-PROXIMITY Design Canvas

This panel is configured with the following settings:

- **A 5ms active mode scan period (~200 Hz).** This is a very high scan rate that is biased towards being able to detect quick gestures. Scanning slower (at 20ms/50Hz for example) would provide lower power consumption, but the algorithm would begin to miss states and would potentially miss valid swipe detections.
- **An inactivity timeout of 375 samples.** This means that the system will transition from active mode into wake-on-proximity mode after 375 samples without any interaction have passed. This works out to about a 1.5-2 second delay.
- **A 33ms wake-on-proximity mode scan period (30 Hz).** Note that this is significantly slower than the 5ms period used in active mode- but it is also faster than the 100ms period used in the CAPTIVATE-BSWP panel. 33ms provides a good balance between fast wake-up to detect gestures, and overall power consumption.
- **A wakeup interval of 512 samples.** This means that the system will wake from the low-power wake-on-proximity state every 512 samples (about every 50 seconds) even if there is not a proximity detection. This provides a mechanism to periodically update the environmental filters for the other sensors on the PCB.
- **A 2 MHz conversion frequency** for the proximity sensor.
- **A total measurement time of 760us** for the proximity sensor.

#### Proximity Detection

The 4 amber LEDs indicate prox detection on a given element (left, center, right, shield). Proximity is detected in the 5-10cm range, depending on the angle of approach.

#### Swipe Detection

The two green LEDs indicate whether or not a left or right swipe was detected above the panel. Waving your hand from left to right over the top of the panel triggers a right swipe, and waving from right to left triggers a left swipe. The LEDs are set after a valid swipe pattern is detected, and are cleared as soon as a new gesture is attempted by a new entry into the proximity field.

Gestures are detected through the use of a software state machine. All steps of the software state machine must be satisfied in order for a swipe to be detected.

Step	Right Swipe Detection	Left Swipe Detection
0	No Detection	No Detection
1	Left	Right
2	Left Center	Right Center
3	Left Center Right	Right Center Left
4	Center Right	Center Left
5	Right	Left

To begin working with this panel, go through the steps for [running an example project](#) and open the CAPTIVATE-PROXIMITY project.

For an overview of the hardware design, view the [CAPTIVATE-PROXIMITY hardware overview](#).

## 10.4 Hardware

This section provides detailed information about the hardware in the kit.

### 10.4.1 CAPTIVATE-FR2633 Processor PCB Overview

The CAPTIVATE-FR2633 PCB contains the MSP430FR2633 target microcontroller. Signals are brought in and out of the PCB in several different ways:

1. There is a 20-pin female debug connector on the top of the PCB. This connector provides power, SBW, UART, and I2C connectivity with the CAPTIVATE-PGMR module.
2. There is a 48-pin female sensor panel connector on the bottom of the PCB. All capacitive sensing IO are brought out to this connector, as well as power and I2C.
3. There is a 40-pin BoosterPack ecosystem header provision in the PCB, to allow the CAPTIVATE-FR2633 PCB to be a BoosterPack to another LaunchPad.
4. There is a 2-pin external power header on the right of the PCB, which provides a location to bring in battery power or some other external power source.

The connection routing is shown below:

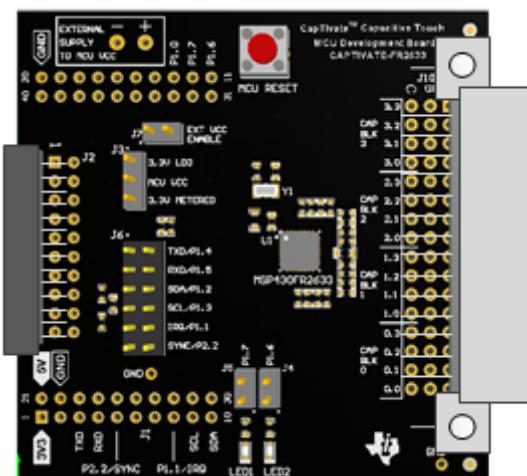


Figure 10.6: CAPTIVATE-FR2633 PCB

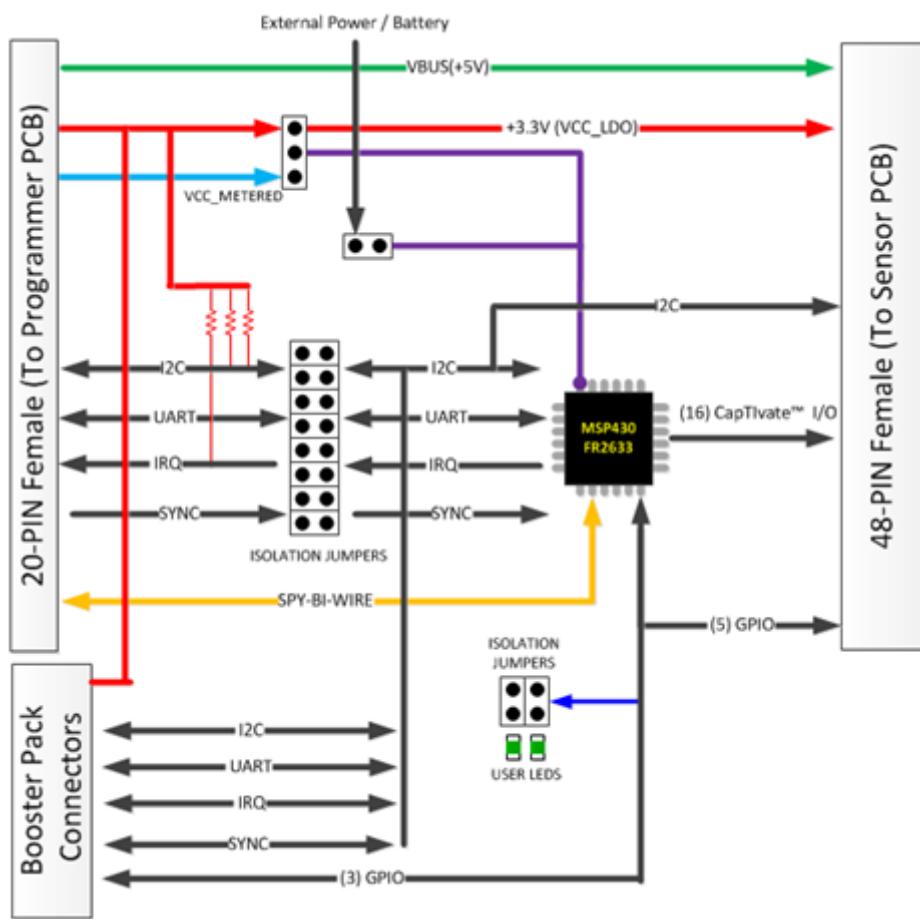


Figure 10.7: CAPTIVATE-FR2633 Signal Flow

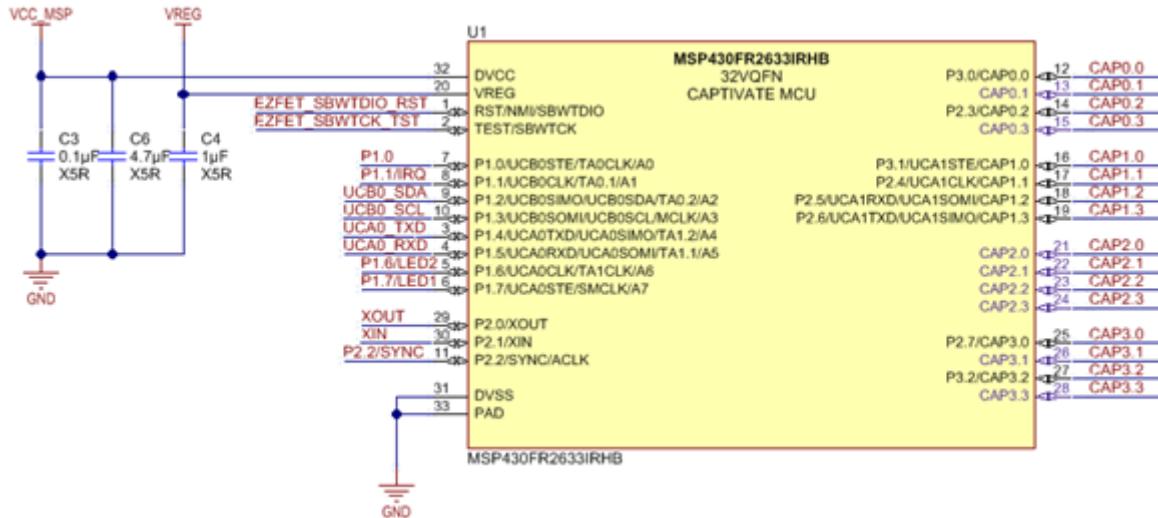


Figure 10.8: CAPTIVATE-FR2633 MCU Pinout

#### 10.4.1.1 Features

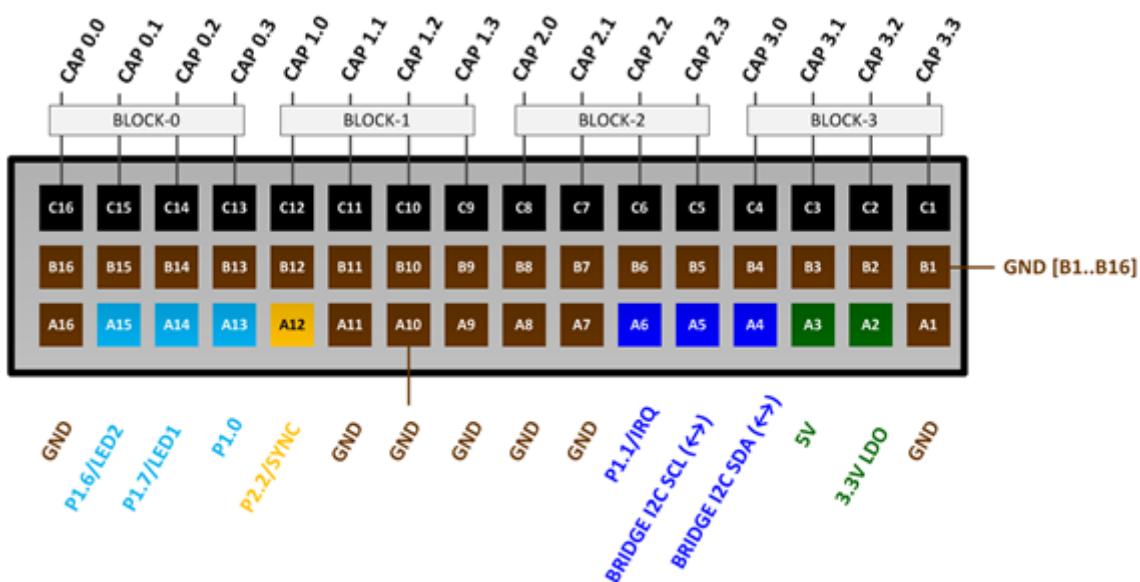
- MSP430FR2633 MCU with CapTlivate™ Capacitive Touch Technology
  - 15.5KB FRAM, 4KB RAM
  - 12KB ROM (BSL, CapTlivate™ Software library, HID communications and MSP430-driverlib I2C and UART drivers)
  - 16 CapTlivate™ I/O
- 48-pin Sensor PCB connector
  - 16 CapTlivate™ I/O
  - 5 GPIO
  - I2C
  - +3.3V LDO
  - +5V (VBUS)
  - Support for future CapTlivate™ devices with up to 32 CapTlivate™ I/O
- 20-pin programmer / power/ communications connector
  - Spy-Bi-Wire target MCU programming
  - Serial Communication with HID bridge
  - (2) +3.3V supply rails
  - +5V USB
- 40-pin BoosterPack footprint
  - Power, UART, I2C and 3 GPIO
- Selectable VCC power jumper J3
  - Selectable VCC source from programmer PCB
    - \* +3.3V EnergyTrace&trade for energy measurements
    - \* +3.3V LDO provides up to 250mA to target MCU and Sensor PCB connector
  - External supply or battery

## MSP430FR2633 MCU with CapTivate™ Capacitive Touch Technology

The MSP430FR2633 can operate from 1.8v to 3.6v, features up to 16-MHz system clock and 8-MHz FRAM access, 15.5KB of non-volatile FRAM and 4KB of data RAM. To help maximize the amount of FRAM code memory available to applications, the CapTivate™ Software Touch Library, HID communications and MSP430-Driverlib I2C and UART drivers are provided in the 12KB of ROM and accessible through a simple API. Refer to the [Software Library API](#) for further details.

The CapTivate™ capacitive touch technology supports self and mutual capacitive measurements on 16 I/O pins, with parallel scanning and up to 64 electrodes. This flexibility allows designers the freedom to mix self and mutual type buttons, sliders, wheels and proximity sensors into a single design.

### Sensor PCB Connector (48 pin Female)



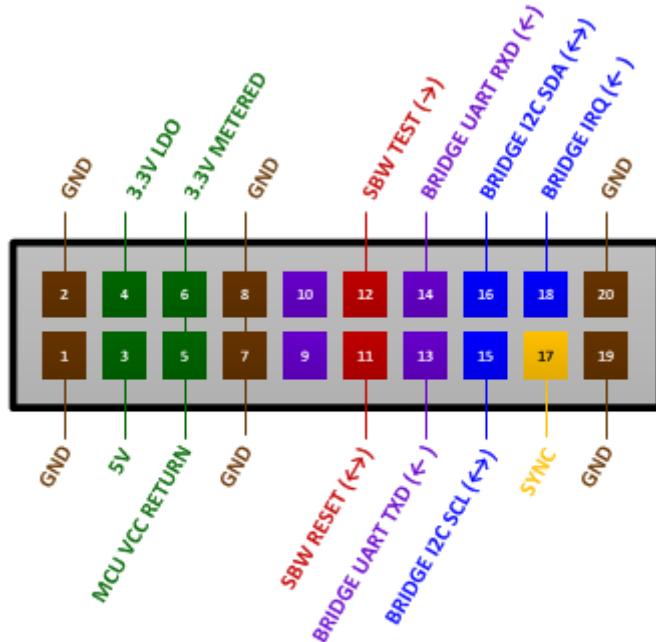
CAPTIVATE-FR2633 PCB 48-PIN Female Connector

Figure 10.9: Sensor Connector pinout

The female connector shown above makes available all of the MSP430FR2633 16 CapTivate™ I/O channels, I2C, +3.3V, +5V and several general purpose I/O. The middle row of this connector is reserved for future devices that support up to 32 CapTivate™ I/O channels. On this PCB, all of the pins on row-B are grounded.

For information on the 48-pin Male version of this connector, refer to the [Sensor PCB Connector](#).

### Programming/Power/Communications Connector (20 pin Female)



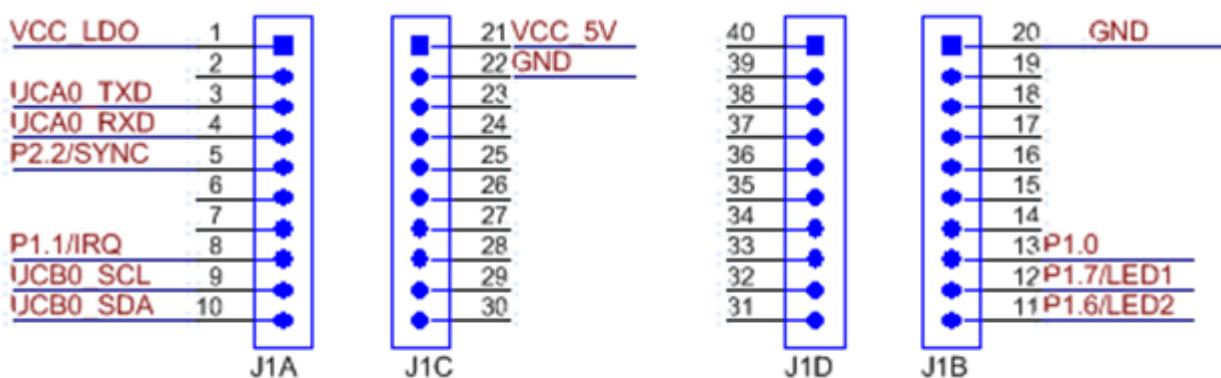
PCB 20-PIN Female Connector

Figure 10.10: Programmer Connector pinout

This female connector shown above is designed to connect with the CAPTIVATE-PGMR programmer PCB. This connector provides power, programming and communications between the two PCBs.

#### BoosterPack Connector Footprint

The CAPTIVATE-FR2633 provides a standard 40-pin BoosterPack connector footprint providing +3.3V, +5V, UART, I2C and 5 GPIO signals.



## LAUNCH PAD CONNECTORS

Figure 10.11: Programmer Connector pinout

Suggested BoosterPack header/connector

- SAMTEC DW-10-15-F-D-210, header, male
- SAMTEC SSQ-110-23-G-D, connector, female

## Power

The CAPTIVATE-FR2633 PCB has two power sources available through the 20-pin connector J2 when attached to the CAPTIVATE-PGMR PCB. One source is a +3.3V (VCC\_LDO) that provides power to the 48-pin connector J10 for devices such as LED drivers or Haptics drivers on a sensor PCB. The second source is the eZFET™ with EnergyTrace™ Technology DC/DC +3.3V output (VCC\_METERED).

### Target MCU power

Position the jumper J3 to pins(1-2) to select +3.3v LDO power source for the target MCU.

### Measuring target MCU power

EnergyTrace™ Technology makes it possible to measure the power consumed by the target MCU when using CCS or IAR IDE. Position the jumper J3 to pins(2-3) to select +3.3v METERED power source for the target MCU.

### Programming/Debug

The MSP430FR2633 MCU on this CAPTIVATE-FR2633 PCB is designed to be programmed and debugged through its Spy-Bi-Wire Interface. The full JTAG connection is not available on this PCB.

**Note** The eZFET™ back-channel UART feature is not supported on the CAPTIVATE-FR2633 PCB.

### Communication

The MSP430FR2633 MCU communicates with a dedicated USB HID Bridge MCU located on the CAPTIVATE-PGMR PCB using UART or I2C communication to send sensor data and status to the CapTlivate Design Center as part of the sensor design and tuning process. A compact communications protocol is provided as part of the CapTlivate™ software library along with UART and I2C drivers. Both are located in the MSP430FR2633 ROM to minimize the impact on the FRAM memory footprint. The communications protocol is described in the [HID Bridge Chapter](#).

When used with CapTlivate™ protocol, the UART operates in a full duplex mode using RX and TX pins, and the I2C operates as an I2C Slave using SDA and SCL pins with an additional pin P1.2/IRQ to generate interrupt requests.

Jumpers are provided to allow isolation between the MSP430FR2633 and the 20-pin Programming and Communications connector, if one or more of the I/O pins need to be re-purposed. Pull up resistors for I2C SDA, SCL and IRQ signals are provided on the "isolated" side of jumpers.

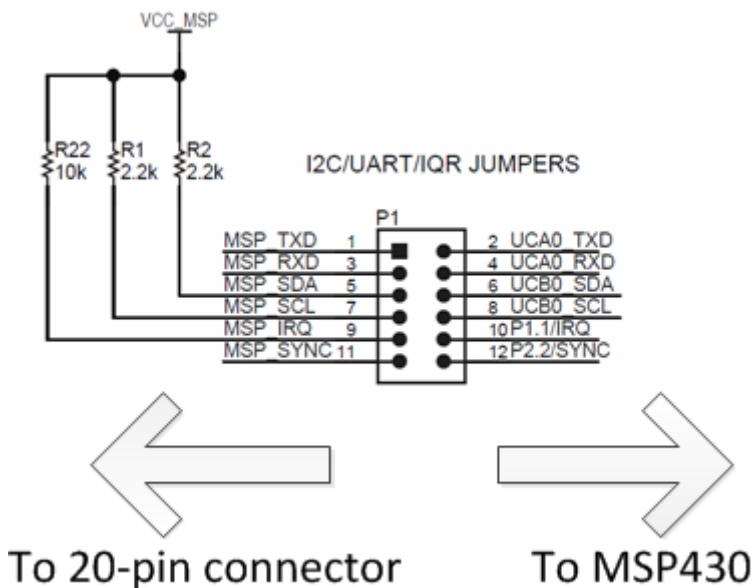


Figure 10.12: Communication Isolation Jumpers

Jumper	Description	Alternate Use
--------	-------------	---------------

TXD	MSP430FR2633 UART data out	P1.4
RXD	MSP430FR2633 UART data in	P1.5
SDA	MSP430FR2633 I2C data in/out	P1.2
SCL	MSP430FR2633 I2C clock input	P1.3
IRQ	MSP430FR2633 I2C slave Interrupt out	P1.1
SYNC	MSP430FR2633 CapTivate™ Sync input	P2.2

### Extended I2C Bus

The I2C signals are also present on the 48-pin sensor PCB connector. The I2C allows the MSP430FR2633 to control I2C slave devices, such as LED or Haptics drivers that might be installed on the sensor PCB.

### Power Selection

Both VCC\_LDO and VCC\_METERED are routed to a selection jumper J3 which provides power to the MSP430FR2633 MCU. For most applications the VCC\_LDO is recommended as it provides a very well-regulated output. When the VCC\_METERED output is selected, power consumed by the MCU can be measured using the EnergyTrace™ Feature in the CCS IDE.

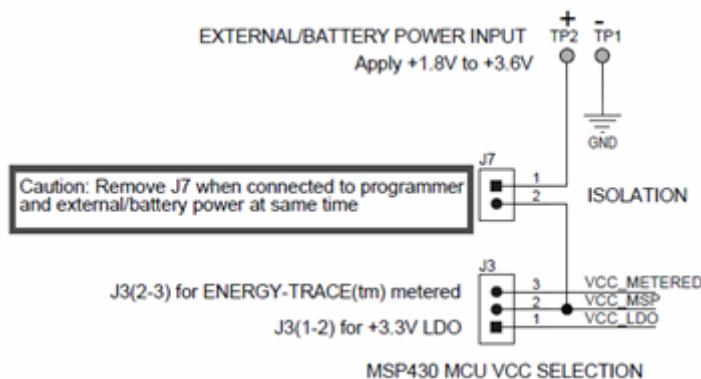


Figure 10.13: MSP430 VCC selection

### External / Battery Power

For battery powered applications, the MSP430FR2633 can be powered from an external 1.8V to 3.6V power source at the external power pads provided on the PCB. This allows the CAPTIVATE-FR2633 PCB and sensor PCB to be evaluated in an environment that is floating from any system or earth ground.

**Caution** When evaluating the MSP430FR2633 with an external power source, jumper J3 on the CAPTIVATE-FR2633 PCB should be completely removed. This prevents any conflicts between VCC\_LDO or VCC\_METERED from the CAPTIVATE-PGMR PCB and the external power source. However, as part of any normal development/debugging process the CAPTIVATE-PGMR and CAPTIVATE-FR2633 PCBs can be connected through the CAPTIVATE-ISO PCB. Using the CAPTIVATE-ISO PCB allows programming with debug as well as I2C and UART communications with the target MCU.

#### 10.4.2 CAPTIVATE-PGMR Programmer PCB Overview

The CAPTIVATE-PGMR PCB provides debug services. It includes an eZ-FET with EnergyTrace and a HID Bridge for bringing UART and I2C into a host PC.

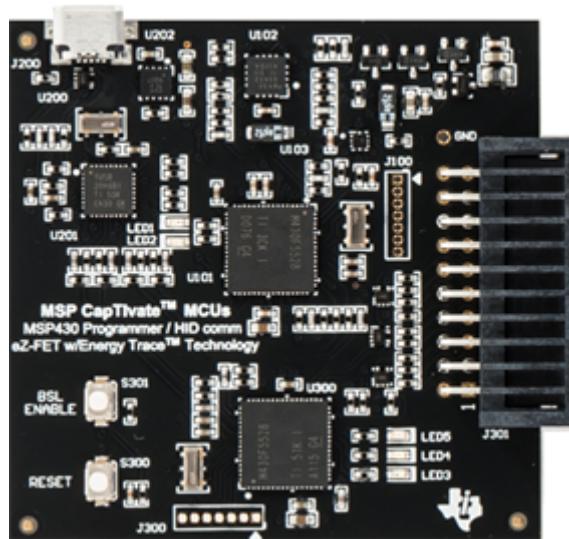
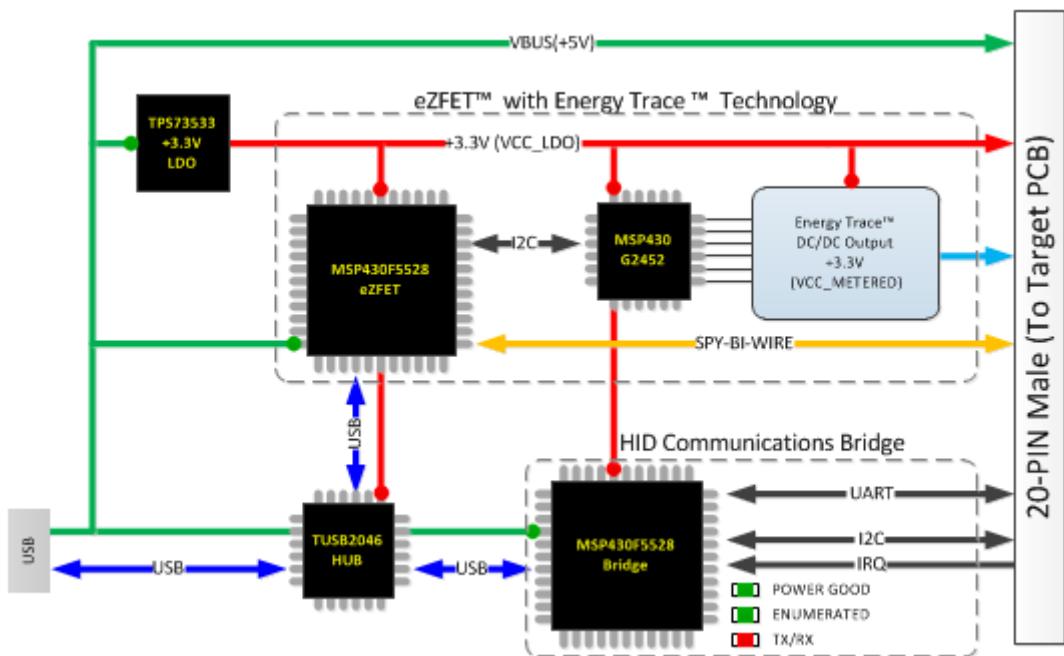


Figure 10.14: pcb



#### 10.4.2.1 Features

- eZFET™ with EnergyTrace™ Technology
  - Simple Spy-Bi-Wire target MCU programming
  - Can be used to program any MSP430
  - Separate +3.3 V outputs available for MCU
    - \* EnergyTrace™
    - \* Dedicated LDO
- 20-pin programmer / power/ communications connector
  - Spy-Bi-Wire Interface
  - UART and I2C Serial Communication with target

- (2) +3.3V supply rails
- +5V USB
- USB HID Serial Bridge
  - Provides interface between CapTivate™ Design Center and target MCU
  - Supports UART and I2C
  - HID - No drivers to install
  - Supports up to 250k baud
  - Easy update via BSL

Programming/Power/Communications Connector (20 pin Male)

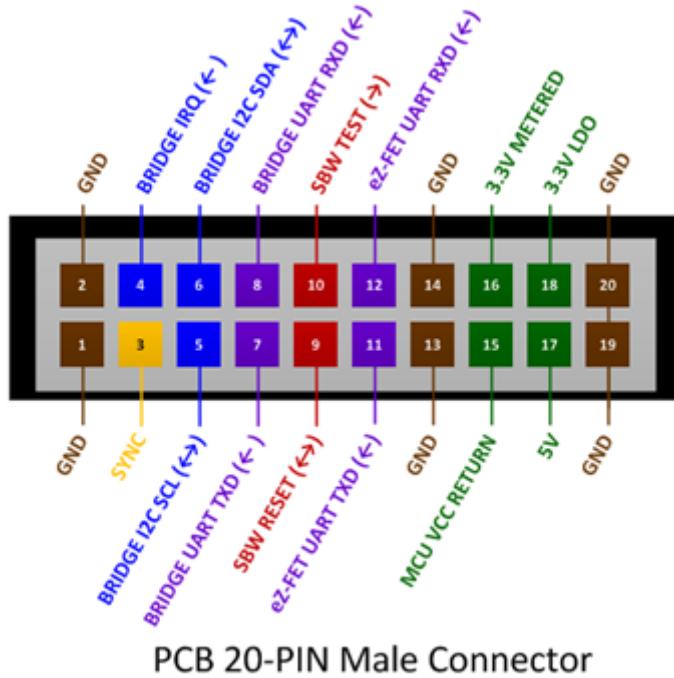


Figure 10.15: Programmer Connector pinout

The male connector shown above is designed to connect with the CAPTIVATE-FR2633 MCU PCB. This connector provides power, programming and communications between the two PCBs.

#### Power

Power to the CAPTIVATE-PGMR PCB is provided through the USB connector at approximately 5VDC. A TI TPS73533 +3.3V, 500mA LDO provides power for all the devices on the CAPTIVATE-PGMR. This +3.3V output is referred to as VCC\_LDO and is made available to the MSP430FR2633 target MCU on the CAPTIVATE-FR2633 PCB when attached. The eZFET™ with EnergyTrace™ Technology also provides a DC/DC +3.3V output. This output is referred to as VCC\_METERED and is available only to the target MCU for energy measurements. A jumper on the CAPTIVATE-FR2633 PCB selects between the two sources for the MSP430FR2633.

**Note:** The CAPTIVATE-PGMR is considered a low-power USB device and therefore should draw at most 100mA. The CAPTIVATE-PGMR PCB draws a nominal 60mA. This allows any target to draw up to 40mA.

#### Programming/Debug

The eZFET™ provides programming and debugging through its Spy-Bi-Wire Interface.

The eZFET™ back-channel UART feature is available on this PCB, however, the CAPTIVATE-FR2633 does not support this feature.

**Using Spy-Bi-Wire (SBW) programming with CAPTIVATE-ISO PCB** Refer to [JTAG and SBW Limitations](#)

---

#### Target Communication via the USB HID Serial Bridge

The CAPTIVATE-PGMR PCB features a HID-Bridge which enumerates as a USB HID device and does not require any drivers to be installed on the PC. It supports both I2C and UART interfaces and is factory programmed with a compact communications protocol for sending sensor data and status between the target MCU and the CapTlivate™; Design Center. For detailed information regarding the communications protocol, refer to [HID Bridge Chapter](#).

Key features include:

- Supports UART and I2C interfaces
- Factory programmed with CapTlivate™ Protocol
- No USB drivers needed
- Easy firmware updates using USB BSL
- For more information, see the [HID Bridge](#) section.

The MSP430F5528 Bridge MCU supports firmware updates using USB BSL. To update the existing firmware or load a user-defined firmware image, use the "Python Firmware Upgrade" utility provided with MSP430ware. Depending on where MSP430ware has been installed on your computer, navigate to the "...\\MSP430ware...\\usblib430\\Host\_USB\_Software\\Python\_Firmware\_Updater" directory.

- Place the MSP430F5528 into BSL mode
  - Press and hold RESET button (S300)
  - Press BSL button (S301)
  - Release RESET button
  - Release BSL button
- Launch the firmware upgrade utility
  - Double-click on the utility icon (the utility will scan the USB bus looking for a specific VID/PID combination of the MSP430F5528 in BSL mode).
  - If no device is found, repeat the steps above to place the MSP430F5528 into BSL mode and select "File>Rescan Bus" in the utility menu.
  - When ready, select "File>Open user firmware" from the utility menu to select the firmware image and begin the update process.

#### 10.4.3 CAPTIVATE-ISO (Communications Isolation PCB)

The CAPTIVATE-ISO PCB provides a way to maintain SBW, I2C, and UART communication when the CAPTIVATE-FR2633 target is powered from an external power source such as a battery or another system.

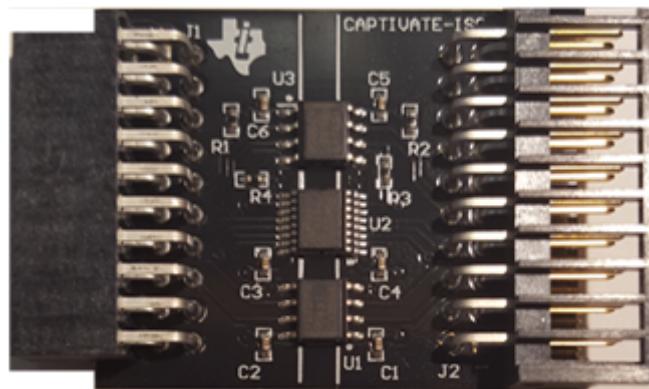


Figure 10.16: CAPTIVATE-ISO PCB

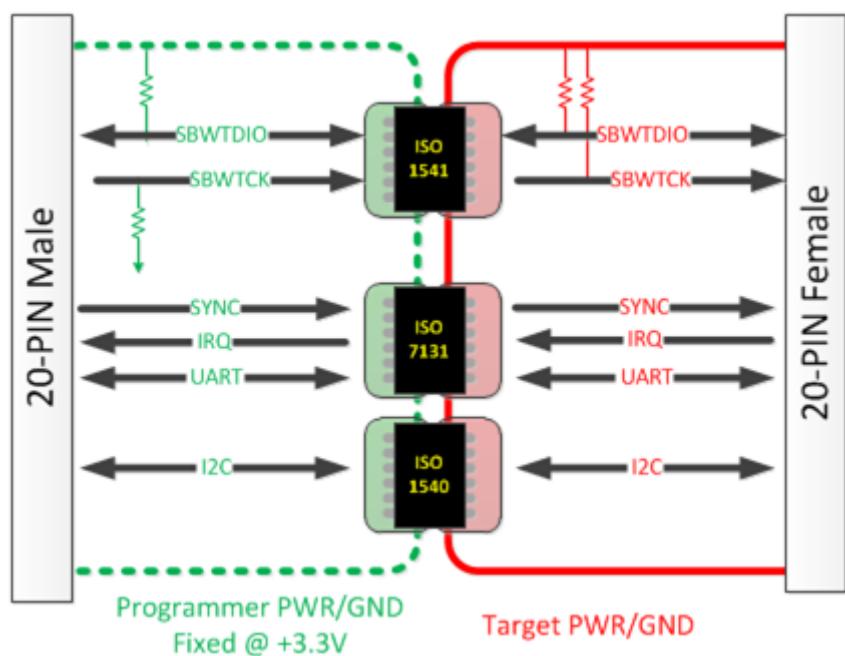


Figure 10.17: Block Diagram

#### 10.4.3.1 Features

- Provides galvanic isolation for SBW programming/debug, I2C and UART communications
  - TI low power digital isolators
    - \* ISO1541 for Spy-by-Wire
    - \* ISO7131 for UART
    - \* ISO1540 for I2C
  - No shared power or grounds
- Use when performing
  - Tuning battery powered applications
  - Conducted noise testing

#### Isolated JTAG Spy-Bi-Wire

The CAPTIVATE-ISO PCB provides isolated Spy-Bi-Wire programming and debugging. Due to the added delays in the SBW timing, it is recommended to use the default medium JTAG/SBW speed or slower. JTAG/SBW speed = FAST is not supported at this time.

#### Typical Isolated Setup

The diagram below illustrates a typical setup using an isolated power supply or battery to power the CAPTIVATE-FR2633 MCU target PCB.

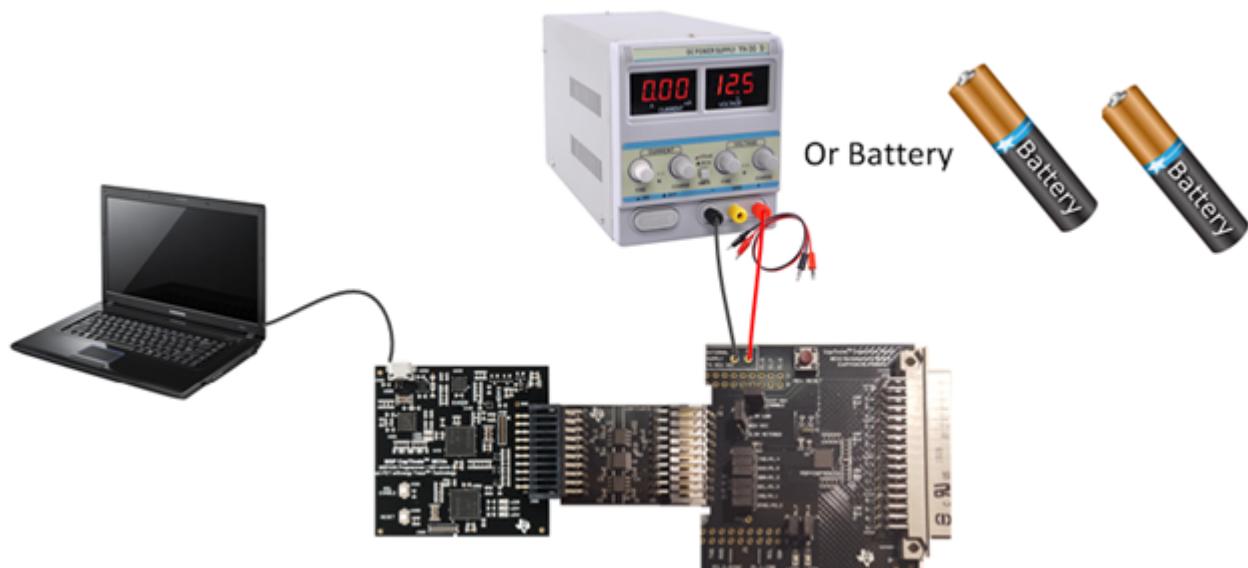


Figure 10.18: Typical Setup

**Important:** To guarantee proper operation, power the "Target" side of the CAPTIVATE-ISO PCB with an external power source that meets the VCC minimum operating voltages shown in the table below. These values are provided for convenience only. Please consult the corresponding device's datasheet for more information.

Isolator	Min Operating VCC
TI ISO1540	3.0
TI ISO1541	3.0

The maximum "Target" side voltage should not exceed the MSP430FR2633 VCC maximum operating voltage 3.6 VDC.

### Connectors

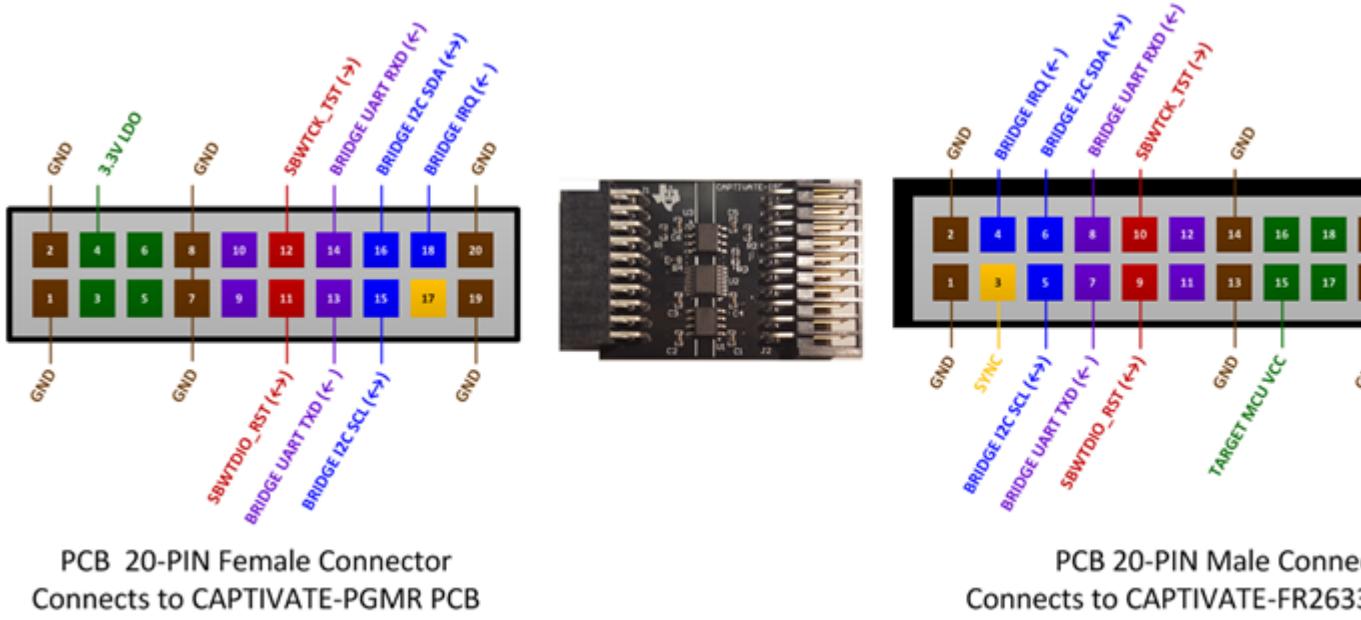


Figure 10.19: Connectors

#### 10.4.4 CAPTIVATE-BSPW

The CAPTIVATE-BSPW is a demonstration sensing panel with buttons, a slider, a wheel, and a proximity sensor. All of the sensors on this panel are self-capacitance sensors. The sensing panel serves the following purposes:

1. Demonstrates low power design principles for a battery powered application
2. Demonstrates the slider and wheel resolution that is achievable through the use of self capacitance
3. Serves as a reference layout for the recommended way to design a slider or wheel sensor
4. Exercises all 16 CapTivate™ sensing IOs on the MSP430FR2633
5. Demonstrates use of the CapTivate™ wake-on-proximity feature for low power consumption
6. Demonstrates use of the CapTivate™ Design Center Auto-Assign feature for automatic pin routing

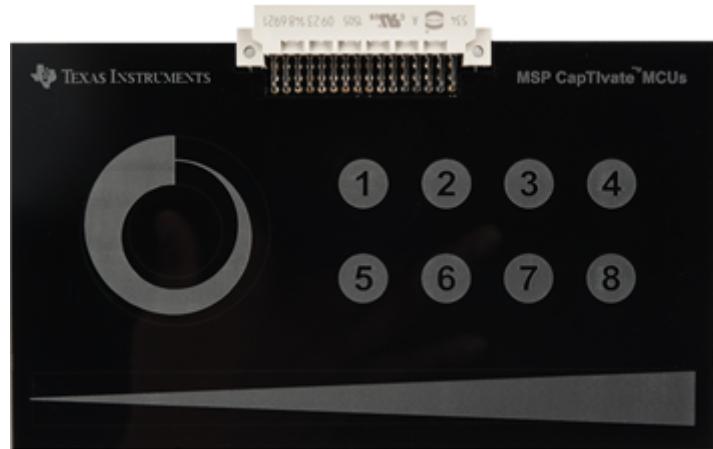


Figure 10.20: CAPTIVATE-BSWP

#### Sensor Design and Organization

The 16 CapTivate™ IOs are fully utilized by this panel. They are organized as shown in the diagram below.

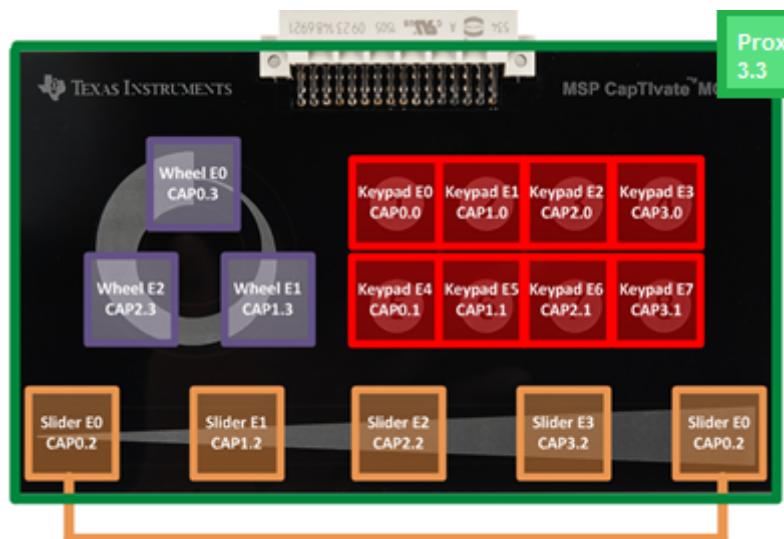


Figure 10.21: Electrode Mapping

#### Button Group Sensor with 8 Buttons

The button group sensor has 8 buttons, or "elements." They are connected to the following IO: CAP0.0, CAP1.0, CAP2.0, CAP3.0, CAP0.1, CAP1.1, CAP2.1, and CAP3.1. Note how the CapTivate™ Design Center auto-assign feature selected two pins from each measurement block. This enables parallel scanning in 2 groups of 4 elements. Basic 10mm diameter circular electrodes were used for the buttons. A ground hatch was utilized underneath the electrodes, but was hollowed out underneath them to improve sensitivity and consequently power consumption, which was a design goal for this panel. For more information on button layouts, see the [design guide](#).

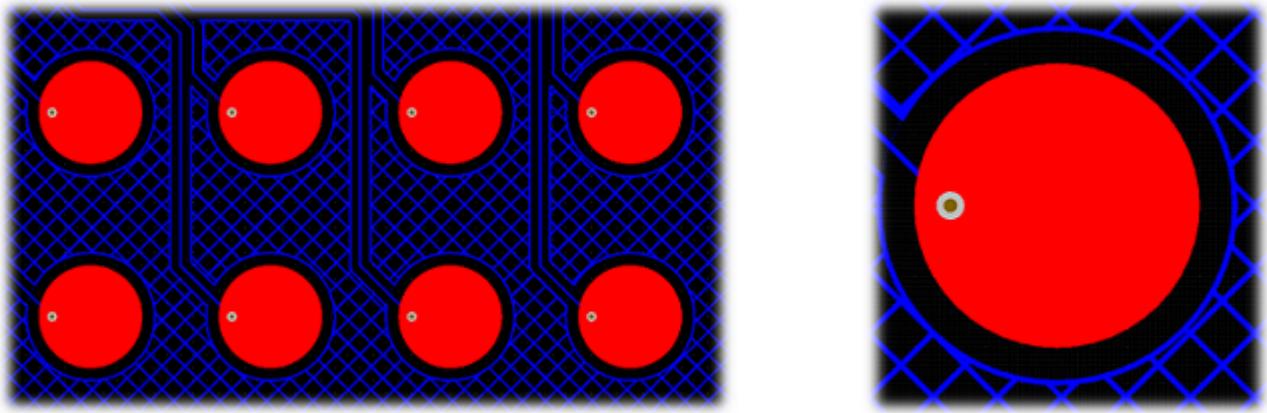


Figure 10.22: Button Layout

#### Slider Sensor with 4 Elements

The slider sensor has 4 elements. They are connected to the following IO: CAP0.2, CAP1.2, CAP2.2, and CAP3.2. Again, one pin was routed from each block to enable parallel scanning of the entire slider. This is very important for achieving the best possible linearity from a slider.

Linearity is defined as the linear accuracy of the reported position of a slider or wheel sensor. It's a measure of how accurately the slider reports back position. For example, a touch in the center of a slider with 1000 points of resolution should return a value in the range of 490-510, not 400 or 600. Additionally, the slider should maintain that accuracy throughout the range of the slider. An example of poor slider performance would be what is called "stair stepping", where the position reported moves quickly, then slowly, then quickly again, all while a touch is moving at a constant rate.

By scanning all 4 slider elements in parallel, three performance improvements are attained. First, because each electrode is driven at the same voltage, there will be minimal E-field between the electrodes. This reduces the parasitic capacitance of each electrode. If they had to be scanned sequentially, neighboring electrodes would have to be grounded during a scan- meaning they look like a large parasitic capacitance. By scanning them in parallel, it is possible to tightly pack the electrodes together, optimizing surface area coverage and sensitivity. Second, there is no latency in the scan. With sequential scanning, by the time the 4th electrode is scanned the touch may be in a different location than when the first scan was performed- distorting the raw data that is fed to the position algorithm and reducing the accuracy of the slider to time changing touches. Third, the slider has common mode noise rejection because of the fact that all electrodes are scanned together. If noise effects one channel, it will affect all channels evenly. This has the overall effect of ensuring that a stable position is reported even in the presence of noise.

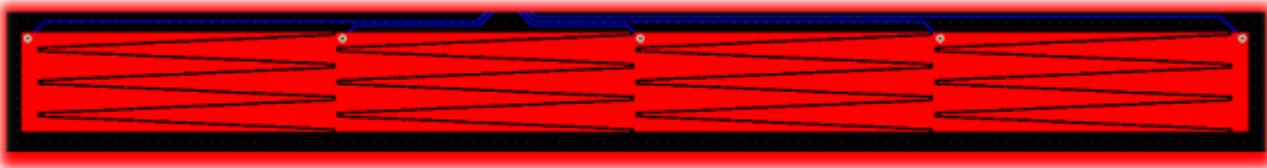


Figure 10.23: Slider Layout

From a layout perspective, the inter-digitation of each element must be as linear as possible to enable linear position tracking. The performance of the slider is only as good as the layout. One feature of this slider is that the outer element is actually split into two pieces, one on each end, connected together to the same pin. The easiest way to think about this configuration is to think of it as a wheel that was "chopped" at the top, and unrolled to be flat. This provides the most linear performance, because the algorithm is going to resolve the slider back to a unit circle.

The proximity sensor that surrounds the slider is grounded during the scan of the slider. This serves to limit the interaction area to the slider, as fringing E-field lines from the edges of the sensor will be reduced because of the grounded sensor nearby.

---

#### Wheel Sensor with 3 Elements

The wheel has 3 elements. They are connected to the following IO: CAP0.3, CAP1.3, and CAP2.3. The design principles for the wheel are the same as for the slider.

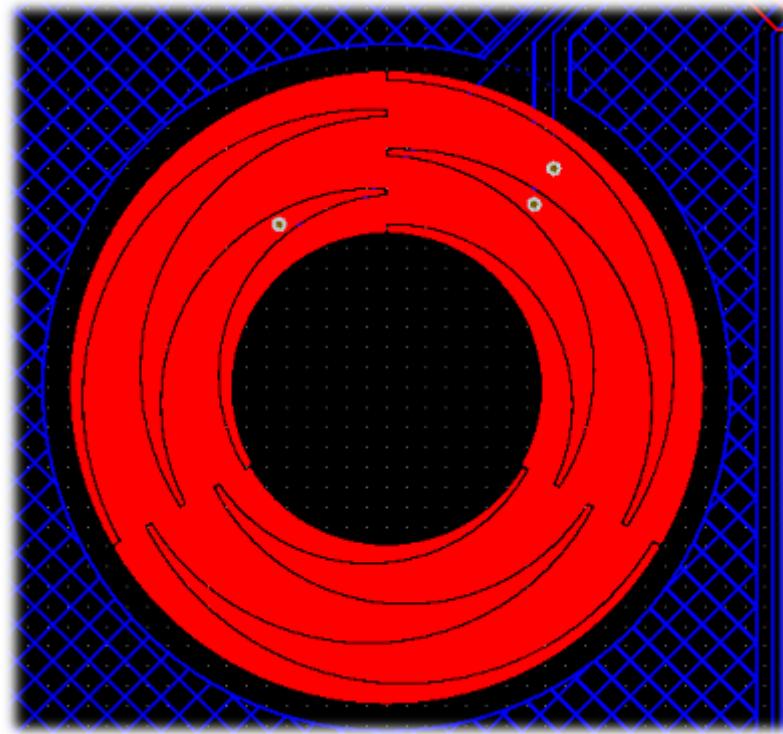


Figure 10.24: Wheel Layout

#### Proximity Sensor with 1 Element

The proximity has 1 element connected to CAP3.3. The proximity sensor serves as a wake-up sensor for the rest of the panel. It was designed to wrap around the existing sensors already on the panel. By providing surface area around the existing sensors, it can detect when a user is close to any given sensor and use that information to wake the MCU from LPM3.

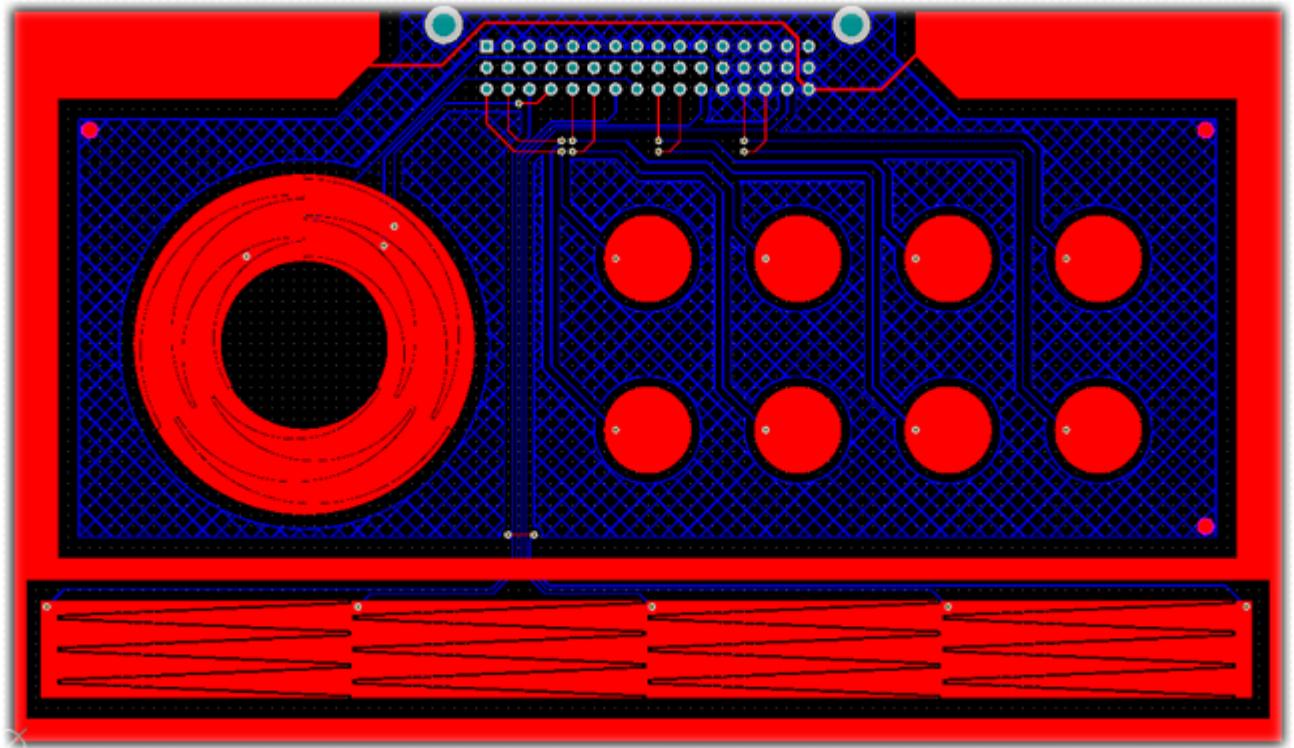


Figure 10.25: Proximity Layout

Note that there is no ground plane underneath of the proximity electrode. This was done intentionally to improve the sensitivity.

To explore the demonstration for this panel, check out the [CAPTIVATE-BSWP demonstration](#) section.

#### 10.4.5 CAPTIVATE-PHONE

The CAPTIVATE-PHONE is a demonstration sensing panel with 17 buttons, 2 sliders, a wheel, and a proximity/guard sensor. There is also a DRV2605L haptic driver IC with a Samsung linear resonant actuator (LRA) for vibrational feedback. The sensing panel serves the following purposes:

1. Demonstrates how to matrix mutual capacitance sensors for high density and low pin count
2. Serves as a reference layout for the recommended way to design a mutual capacitance slider or wheel sensor
3. Demonstrates a hybrid technology approach that has mutual and self capacitance in the same design
4. Demonstrates the re-purposing of a proximity sensor as a guard channel
5. Demonstrates a real-world application (desk phone interface)

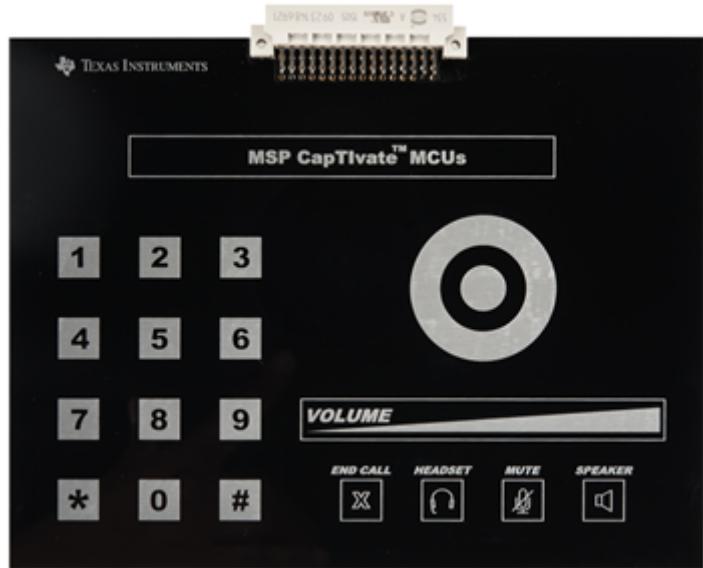


Figure 10.26: CAPTIVATE-PHONE

#### Sensor Design and Organization

Only 12 CapTIvate™ IOs are used for this demonstration, which has 29 elements! A mutual capacitance matrix made up of 4 Rx lines and 7 Tx lines (11 total pins) forms 28 elements. The 29th element is a self-capacitance guard channel and proximity combo sensor. The guard and proximity sensor is on CAP3.3. The 4 Rx lines are shared between all of the mutual capacitance sensors, and are connected to CAP0.0, CAP1.0, CAP2.0, and CAP3.0. Just like the CAPTIVATE-BSPW panel, selecting one receive line from each measurement block allows for efficient parallel scanning of 4 elements at a time.

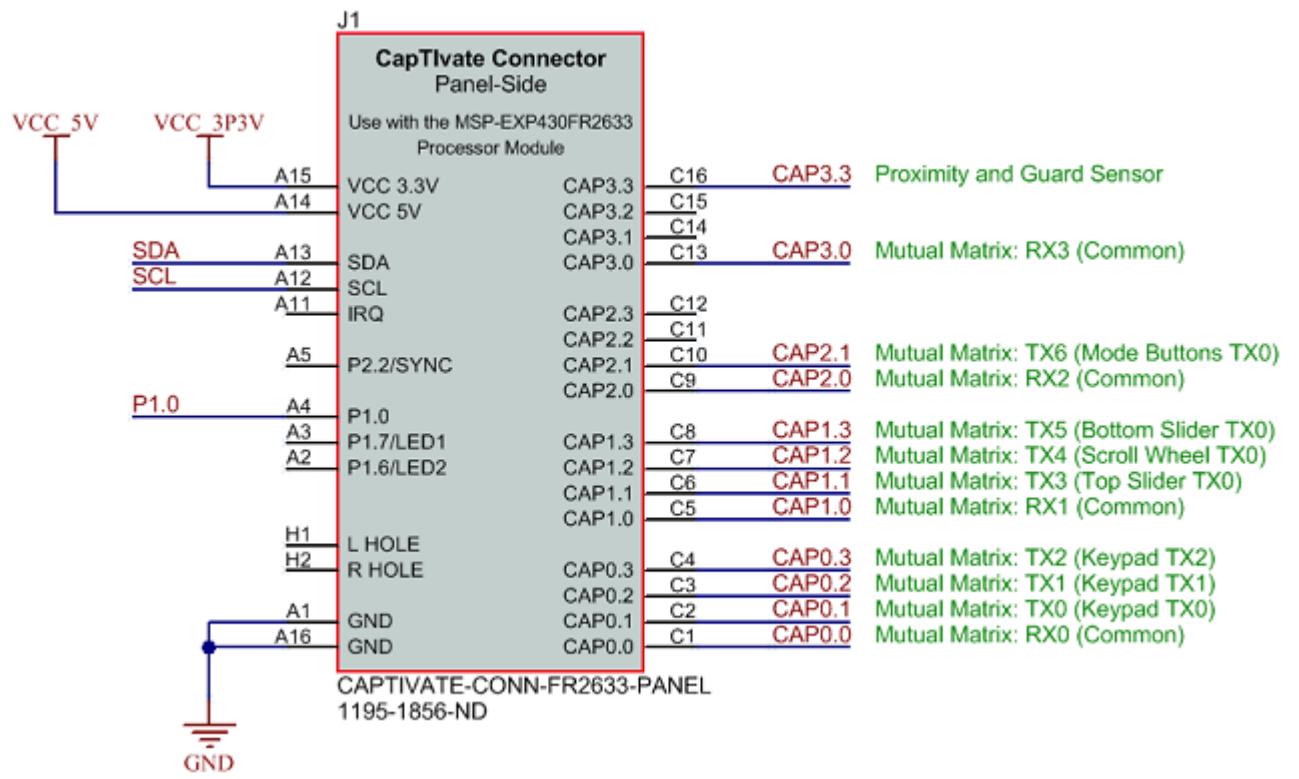


Figure 10.27: CAPTIVATE-PHONE

#### Numeric Keypad and Mode Keypad

The CAPTIVATE-PHONE demonstration utilizes the basic mutual capacitance button layout as shown in the [design guide](#). This geometry is easy to lay out and provides more than adequate sensitivity for this application. Note that the guard channel is routed between the buttons. This will serve as a ground shield while the buttons are being measured.

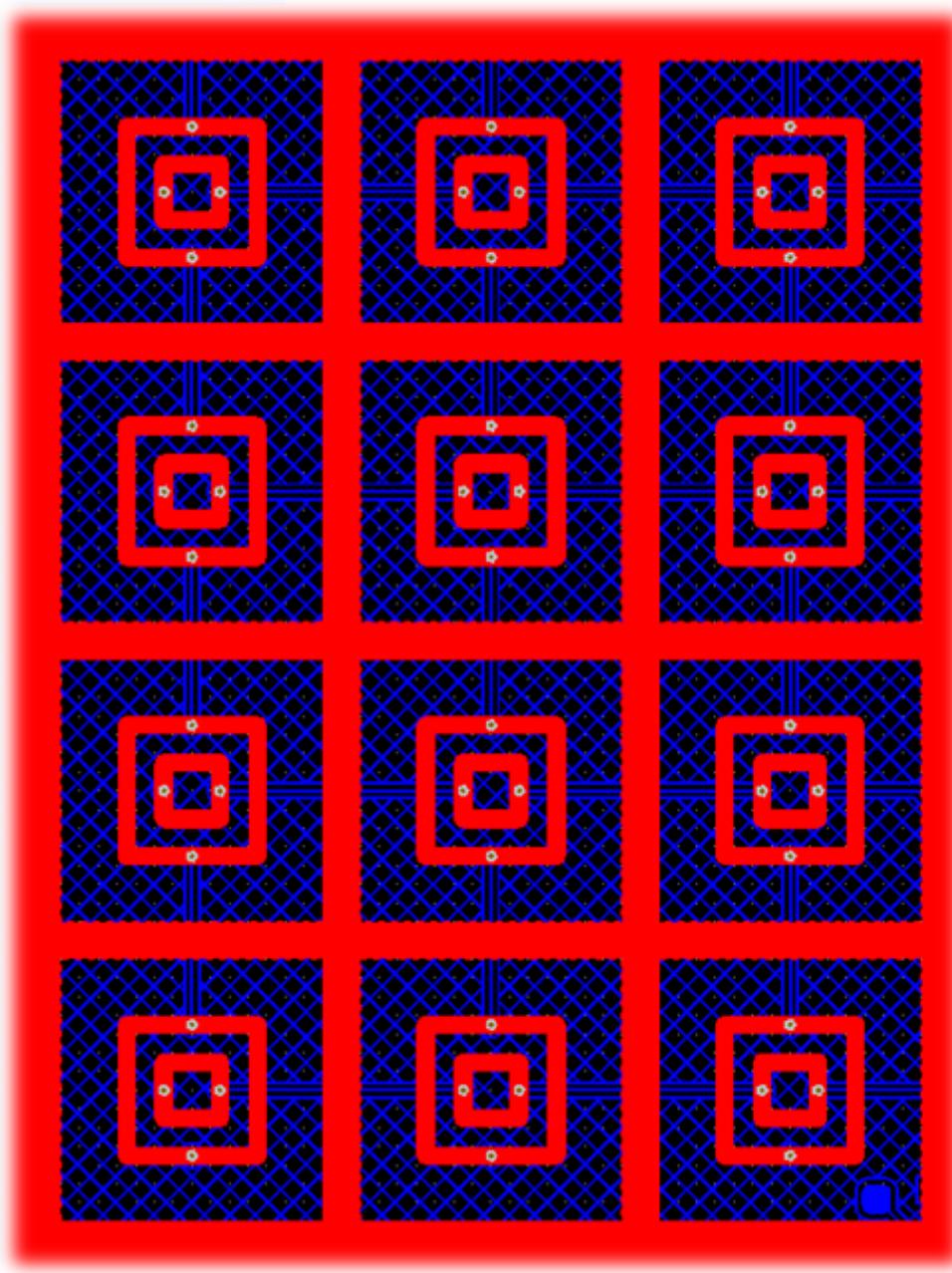


Figure 10.28: Keypad Buttons (Square)

---

### Wheel Button

The wheel selection button in the center of the wheel demonstrates how the geometry may also be radial with similar performance. Testing has revealed that E-field lines concentrate at the 90-degree corners of the square geometry, improving sensitivity. However, a radial geometry is still viable and is more applicable to the wheel button case because of the surrounding wheel sensor. Note that there is a ground fill between the wheel itself and the wheel button. This serves to provide shielding for the button, so that a touch on the edge of the wheel does not trigger the button.

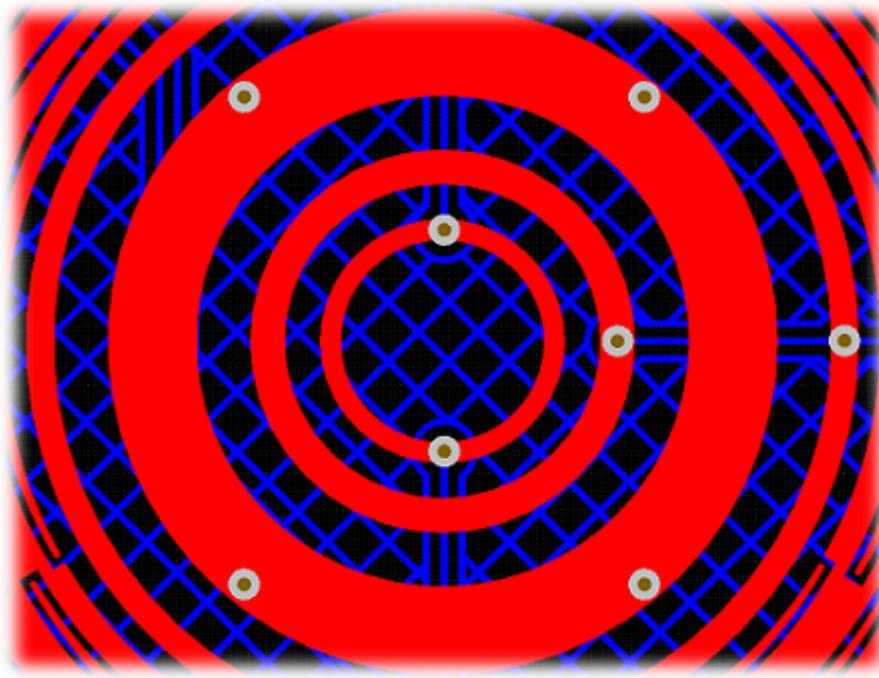


Figure 10.29: Wheel Button (Circular)

### Scroll Wheel

The scroll wheel implementation on this PCB is effectively a condensed self-capacitance wheel for the Rx electrodes, with an inner and outer Tx ring forming the mutual coupling. This topology works quite well for creating a mutual capacitance wheel.

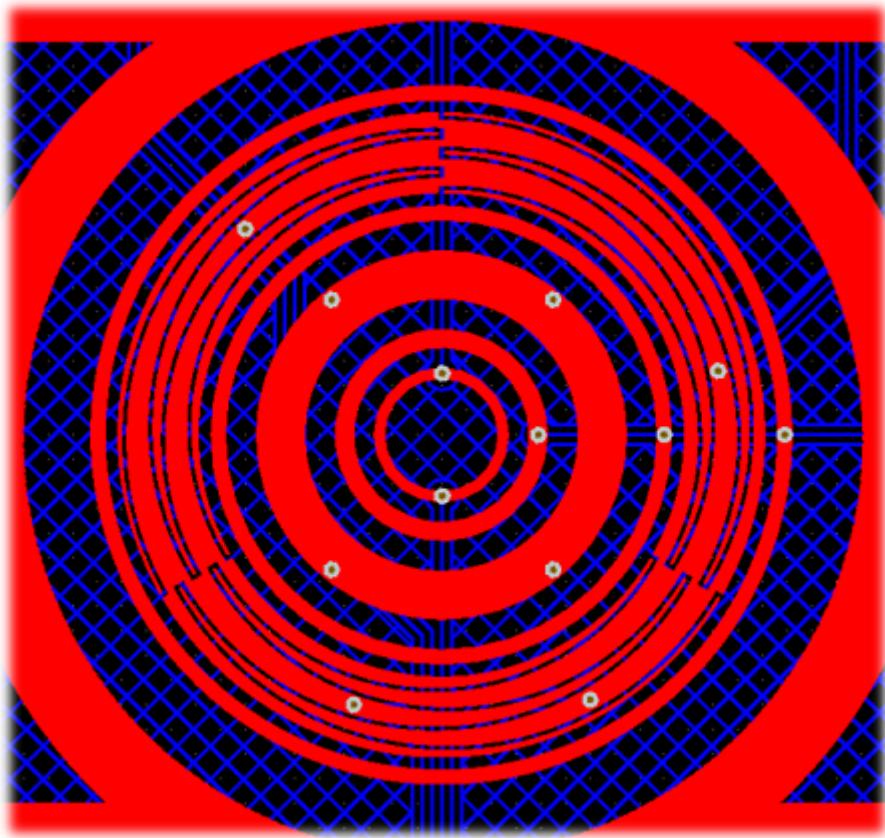


Figure 10.30: Scroll Wheel

With slider and wheel design it is extremely important to match the parasitic capacitances as best as possible to ensure the best linearity. In the case of mutual capacitance, this means parasitic *mutual* capacitance. This type of parasitic shows up wherever Rx's and Tx's are brought close together. Usually it is straightforward to minimize trace parastics, but it can be difficult on larger PCBs such as this, or PCBs that have connectors. Drastic differences in parasitic capacitance lead to differences in sensitivity between electrodes in the slider or wheel, which will negatively impact the linearity. The CapTIVate™ technology calibration routine will reduce these effects to a certain extent, but it is always best to start with a good layout. Be aware of where your Rx's and Tx's come close together, and minimize these areas as much as possible by crossing them at 90-degree angles and placing ground between them if they run parallel to each other. This will serve to improve sensitivity as well as linearity.

#### Slider Sensors (General Purpose and Volume Slider)

The CAPTIVATE-PHONE sensing panel has two mutual capacitance sliders of different lengths. Both have 4 elements each based on the 4 shared Rx electrodes. The same layout technique is applied to both sliders, and it is quite easy to implement in most layout packages. Just like the scroll wheel, the Rx electrodes are encapsulated by two Tx tracks. Rather than the complex inter-digitation of the wheel layouts and the self-capacitance slider, a simple triangle design is implemented for this PCB.

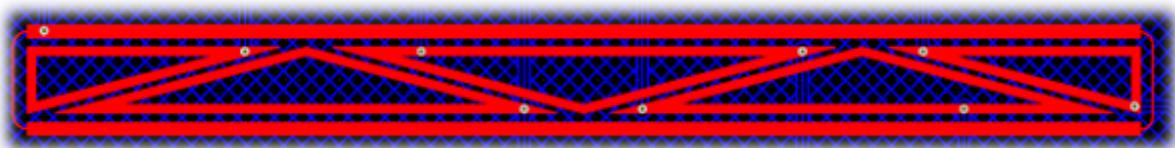


Figure 10.31: Slider Sensor

Note that just like the self-capacitance slider on the CAPTIVATE-BSPW panel, the end elements are "half" elements, and they are connected together as if they were one full size element. This technique provides the best

---

performance from the position algorithm. If a smaller slider is desired, it is acceptable to drop an inner element at implement a 3-element slider using the same approach.

#### Proximity and Guard Sensor

The proximity and guard sensor is simply a fill that flows in between all of the other sensor elements on the PCB. This is the only self-capacitance sensor on the panel.

#### Proximity Functionality

From a proximity perspective, the electrode does not offer large sensing distances because it is directly above the ground hatch on the bottom layer. This creates a large parasitic capacitance on the order of 200pF, and it also reduces the E-field penetration into the area above the panel. A distance of a few centimeters is attainable- but not 5-10cm.

#### Guard Channel Functionality

The guard channel functionality is essentially a re-purposing of the proximity sensor data. Because the proximity/guard electrode wraps around all of the sensors, it can be used as a detection mask for the other sensors. What this means is that when the guard channel reaches a certain level of interaction (the touch threshold), reporting of touches on other sensors is *masked* by the guard channel detection. This prevents other sensors from triggering if someone puts their whole palm down on the panel, or wants to wipe it down for cleaning purposes.

To explore the demonstration for this panel, check out the [CAPTIVATE-PHONE demonstration](#) section.

### 10.4.6 CAPTIVATE-PROXIMITY

The CAPTIVATE-PROXIMITY is a demonstration proximity sensing panel with 4 electrodes. There are also 6 status LED indicators on the PCB: 4 for proximity detection, and 2 for gesture detection. The sensing panel serves the following purposes:

1. Demonstrates how to implement proximity gesturing to improve the robustness of a proximity application such as a paper towel dispenser
2. Serves as a reference layout for the implementation of a driven shield electrode with a CapTivate™ IO to reduce parasitic capacitance
3. Demonstrates the use of muxed CapTivate™ IOs as active-low LED current sinks.

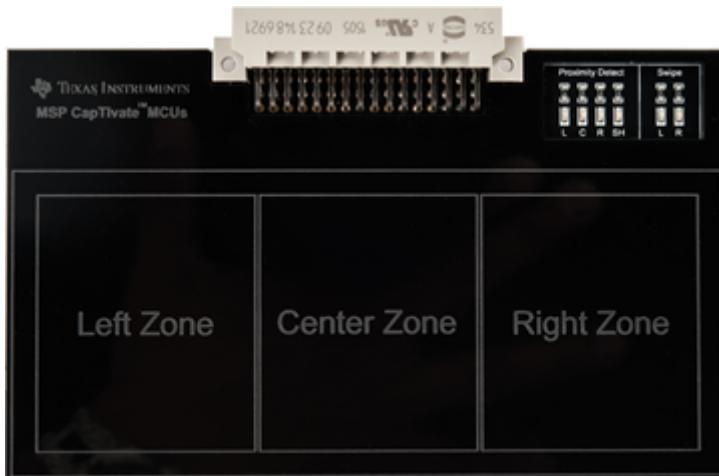


Figure 10.32: CAPTIVATE-PROXIMITY

---

## Sensor Design and Organization

The module's 4 electrodes are connected to CAP0.3, CAP1.3, CAP2.3, and CAP3.3. CAP1.3 is used for the shield electrode, which is a solid plane across the bottom layer of the PCB and a ring around the 3 zones on the top layer of the PCB. CAP0.3, CAP2.3, and CAP3.3 are used to drive the left, center, and right zones, respectively. These three IO's are all dedicated capacitive sensing IO's- they are not muxed with GPIO functionality. As a result, they are not connected to the DVCC IO rail internally. While they have similar sensitivity when compared with muxed IO's, in theory they will be less effected by any switching noise that may be present on that rail due to a PWM, I2C, or other digital signal. All 4 electrodes are measured in parallel (one per sensing block).

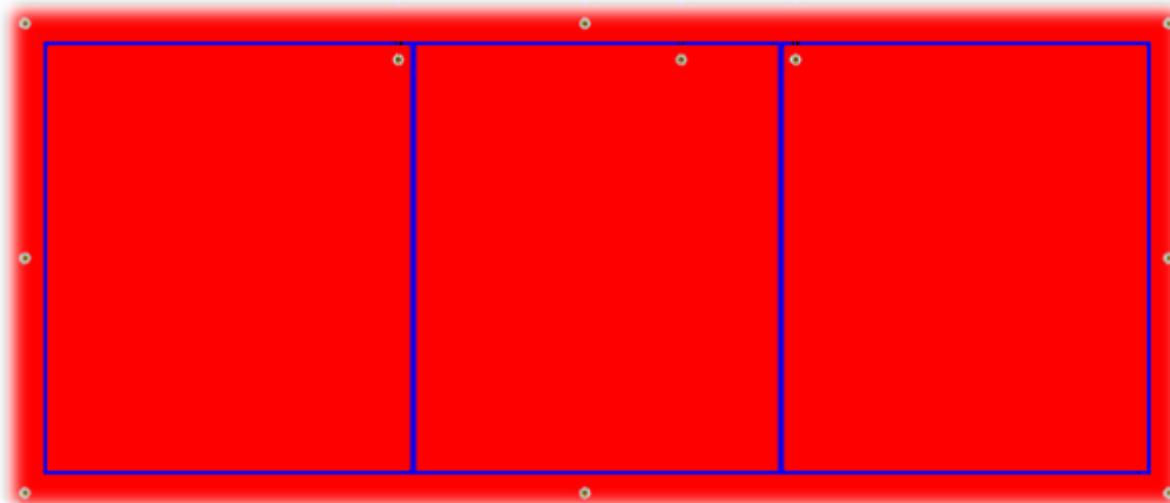


Figure 10.33: Proximity Sensors

## Shield Theory

Because all four electrodes are driven in parallel off of the same conversion clock, the potential difference between the various electrodes will be small. This means that the E-field between them (and thus, the result parasitic capacitance) will also appear quite small to the electrode. This parasitic reduction directly results in an improvement in overall sensitivity. It should be noted that the shield does not provide immunity to proximity interactions below the shield. If immunity to proximity on one side of an electrode is needed, a light (<20%) ground hatch is a better structure.

### 10.4.7 48-pin Male Sensor PCB Connector Information

All demo sensor PCBs use the same 48 pin male connector that is commonly available from various manufacturers and suppliers. The following mechanical information will be useful if designing your own PCB to connect to the CAPT-FR2633 PCB. Below is a list of manufacturer part numbers:

Manufacturer	Part No	Digikey No
FCI	86093487313H55ELF	609-4951-ND
Harting	09 23 148 6921	1195-1856-ND
Harting	09 23 148 2921	1195-1854-ND

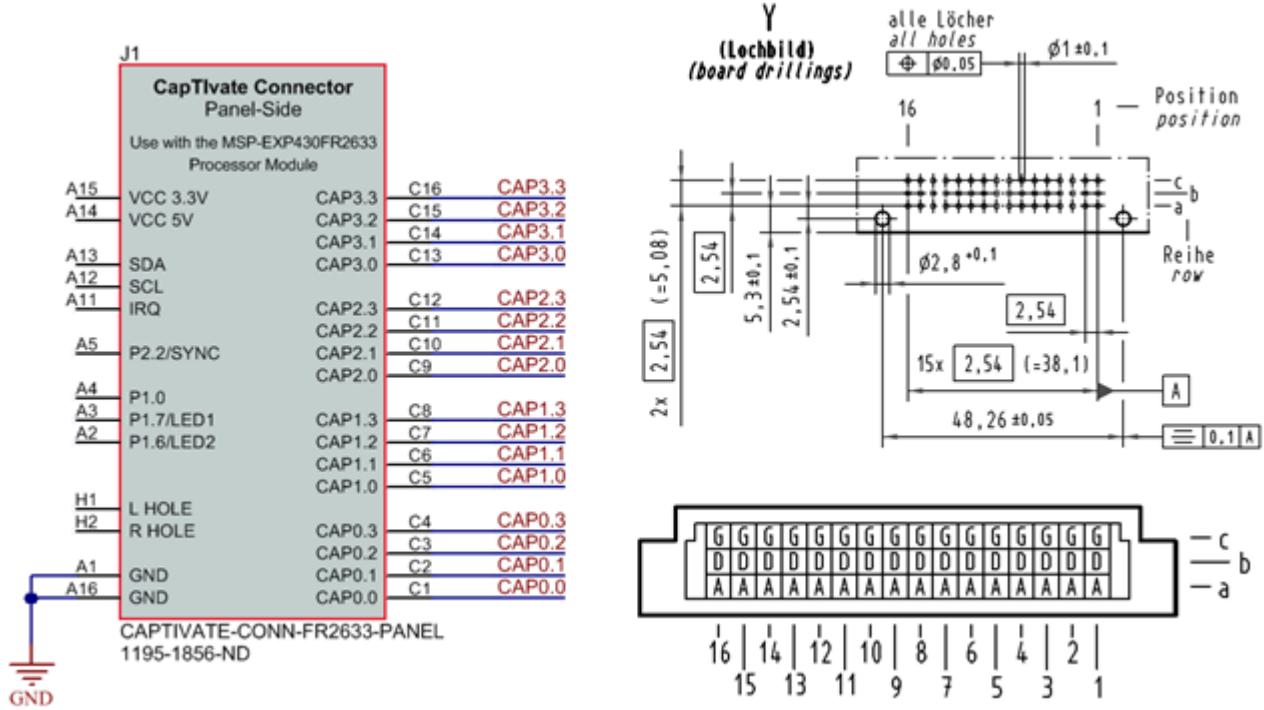


Figure 10.34: Sensor Connector pinout

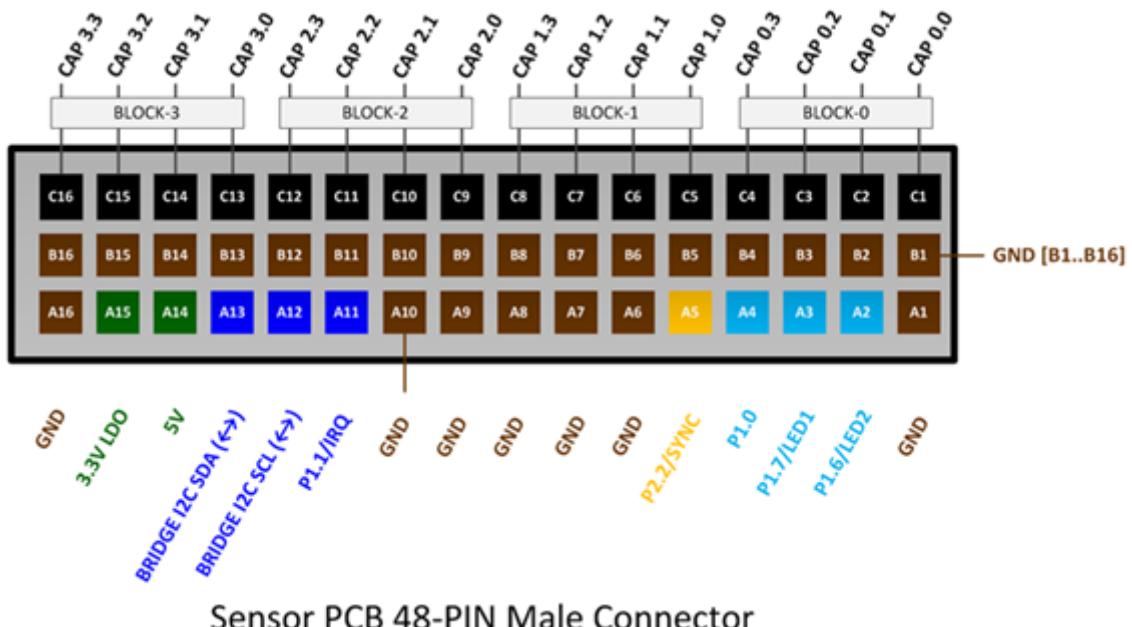


Figure 10.35: Sensor Connector pinout

## 10.5 Using the HID Bridge

The HID Bridge is a multi-purpose development tool that provides a way to interface platform software (such as a PC GUI) with embedded systems containing basic serial interfaces (such as UART or I2C). The HID Bridge is a firmware product designed to run on an MSP430F5xx MCU with USB support. Typical implementations (such as the CAPTIVATE-PGMR) utilize an MSP430F5528 MCU.

The HID Bridge enables bidirectional transfer of data from a serial interface to a host platform via the USB HID device class. HID provides a unique advantage over other competing interfaces. Unlike USB CDC devices, no COM port identification is necessary on the host and no drivers are required on most modern operating systems. In addition, HID devices are much better suited to hot-swapping (disconnection and re-connection). During embedded system development it is very common to disconnect and re-connect development tools; using HID makes this process seamless.

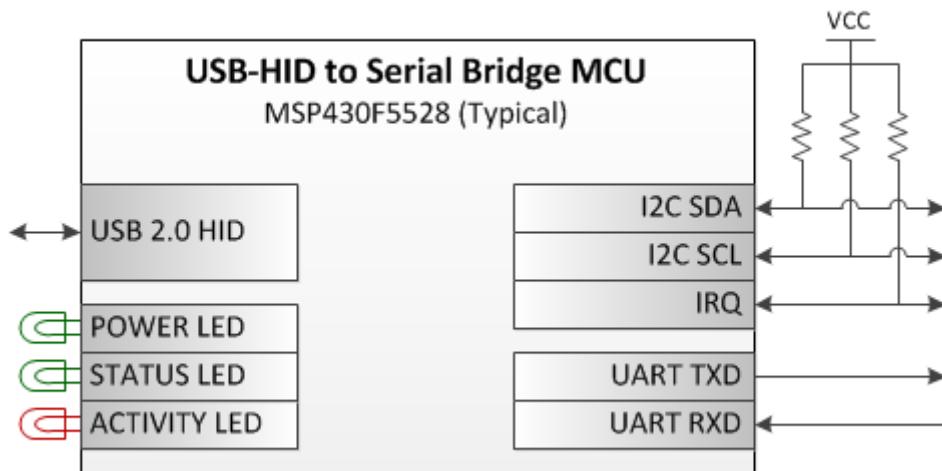


Figure 10.36: HID Bridge Diagram

### Supported Modes of Operation

The HID Bridge provides the following modes of operation:

- HID to Unformatted Raw UART (bi-directional full-duplex up to 250k baud)
- HID to Formatted UART (bi-directional full-duplex up to 250k baud, with intelligent packet identification and framing)
- HID to Unformatted Raw I2C Master (Up to 400kbps I2C master)
- HID to Formatted I2C Master (Up to 400kbps I2C master, with intelligent packet identification, framing and slave IRQ handling)

#### 10.5.1 HID Bridge Interfaces

##### Host Interface: USB HID Device-Class Implementation

The bridge enumerates on the USB host as a composite class human interface device (HID). Two HID devices enumerate:

- HID0, the data transfer interface

- Used for transmission of data between the target and the host
- Data is binary formatted
- HID1, the HID Bridge configuration console interface
  - Used for configuration of the HID Bridge itself via the [HID Bridge \(HB\) command set](#)
  - Commands are ASCII string formatted

#### Target Interface: UART Implementation

The HID Bridge target UART interface is a bi-directional, full-duplex, two-wire UART interface that supports several common baud rates.

The baud rates supported are: 250kBaud, 115.2kBaud, 38.4kBaud, 19.2kBaud, and 9.6kBaud. The HID Bridge, when configured in UART mode, is constantly listening to the target UART port and may receive data at any time, even while it is transmitting data to the target. The UART baud rate is configurable via the HB command set.

The UART implementation also supports a "byte delay" feature for slow targets. This feature will insert a delay (in ms) between the start-of-transmission of each byte. This can allow a fast (250kBaud) rate to be used, but the bytes will be spaced out to allow the target to have time to service its UART interrupt service routine. When the target is sending data, it can still send at the full 250kBaud rate- but it doesn't have to respond to a data stream at that rate, because bytes will be spaced. The UART delay is configurable via the HB command set.

#### Target Interface: I2C Implementation

When configured in I2C mode, the HID Bridge acts as an I2C bus master that communicates with an I2C slave device.

The HID Bridge initiates writes to the target by initiating a start/write condition to the slave address that is selected, writing the data, and issuing an I2C stop condition.

The slave can request a read by pulling the I2C IRQ line low until the HID Bridge reads out the data from the target. Once the HID Bridge issues a start/read condition to the slave, the slave must release the IRQ line. The first byte read by the master is expected to be a length field that indicates to the HID Bridge how many further bytes it should read from the slave. The master will then continue the read, reading out that many bytes, then issuing a stop condition.

#### 10.5.2 HID Bridge Configuration Command Set

The HID Bridge is configured by the host platform through the sending and receiving of HID Bridge "HB" commands over the configuration console HID interface. The available configuration commands are described below.

Command	Description	Arguments?	Valid Arguments (1)	Valid Arguments (2)
HB VERSION	Get the version string.	None	-	-
HB PLATFORM	Get the platform string.	None	-	-
HB UPGRADE	Enter the USB bootloader.	None	-	-
HB REBOOT	Reboot the HID Bridge.	None	-	-
HB MODE	Get/set the operating mode.	Format, Interface	RAW, PACKET	UART, I2C

HB SAVECONFIG	Save the active or default configuration to boot memory.	Config to save	ACTIVE, DEFAULT	-
HB UARTBAUD	Get/set the UART baud rate.	New baud rate	250000, 115200, 38400, 19200, 9600	-
HB UARTDELAY	Get/set the UART byte delay period.	New delay period	0 to 1000	-
HB I2CADDRESS	Get/set the target I2C slave address.	New address	0 to 127	-
HB I2CCLOCK	Get/set the target I2C clock freq.	New frequency	100000, 400000	-

For the commands that have no arguments, simply send "HB x" where x is the command. For commands with arguments, sending "HB x" where x is the command will echo back the current setting of that command, if supported. If valid parameters are passed (for example, "HB UARTBAUD 250000"), the parameter will be updated to the new value passed.

Note that the Captivate Design Center will automatically send configuration commands on this interface based on the current Design Center project that is open.

### 10.5.3 HID Bridge Modes of Operation

The HID bridge supports several different modes of operation as introduced above. The details behind how each mode behaves are described in this section. The operating mode may be set via the HID Bridge command set.

#### 10.5.3.1 Operating Mode: Packet Mode

Packet mode requires that data sent over the target interface be formatted in packets according to the following set of transmission rules. The HID Bridge will buffer all data received from the target. It will then identify and frame packets to be sent to the host. When sending packets to the host, each packet will be transmitted in its own HID report. The HID Bridge buffers all packets coming from the host, and applies the relevant transmission rules before sending the packet on to the target.

If any transactions are of the following type, they are subject to the transmission rules:

- Packets from the target to the HID Bridge in PACKET/UART Mode
- Packets from the HID Bridge to the target in PACKET/UART Mode
- Packets from the HID Bridge to the target in PACKET/I2C Mode

The transmission rules for the serial interface in packet mode are as follows:

- All packets will begin with a 3-byte header, consisting of:
  - [0] A SYNC byte at position 0, equal to 55h
  - [1] A BLANK byte at position 1, that is NOT equal to 55h. Typically , AAh is used.
  - [2] A LENGTH byte at position 2, that indicates the size of the payload and checksum in bytes (payload length + 2)
- All packets may contain a payload up to 60 bytes, which may contain any binary data
  - If the SYNC byte (55h) should occur in the payload section, it must be sent twice (byte stuffed) to distinguish it from a true SYNC byte. This repeated byte does NOT count towards the maximum payload size of 60 bytes.
- All packets will end with a 2-byte checksum.

- The checksum is calculated as the lower 16 bits of the summation of the payload section, less any stuffed bytes (repeated 55h bytes).
- The checksum is send lower byte first, upper byte second.

Per these rules, a packet would have the following format:

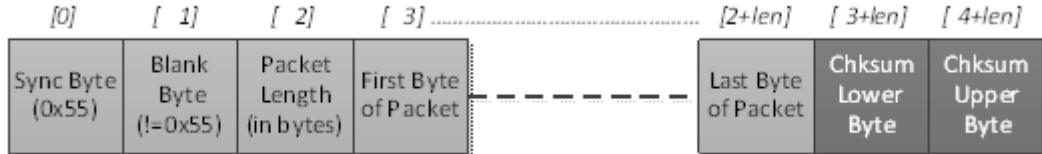


Figure 10.37: Serial Interface Packet with Transmission Rules Applied

All other packet types (the remaining types are listed below), only require that a checksum be appended to the payload.

- Packets from the target to the HID Bridge in PACKET/I2C Mode
- Packets from the HID Bridge to the host in any mode

Note that the HID Bridge strips out the transmission rule overhead (if applied) when sending the payload and checksum on to the host. The SYNC, BLANK, and LENGTH bytes are removed, and any stuffed bytes in the payload section are removed. The checksum is preserved and is sent on.

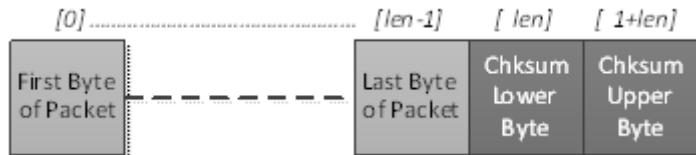


Figure 10.38: Base Packet without Transmission Rules

In all cases, when the HID Bridge is in PACKET mode, the checksum field must be valid or data will not be sent on. For example, if a packet is sent from a target to the HID Bridge, and the checksum is invalid, that packet will not be sent on to the host. Likewise, if the host sends a packet to the HID Bridge, and the checksum is invalid, that packet will not be sent on to the target.

#### 10.5.3.2 Operating Mode: Raw Mode

When operating in RAW mode, the HID Bridge basically acts as a communication buffer. No formatting is required. Any data received from the target is sent on to the host, and any data received from the host is sent on to the target. There is no HID frame alignment on the host side.

# Chapter 11

## Workshop (Getting Started)

### An Introduction to Capacitive Touch with MSP CapTlivate™ Microcontrollers

This workshop steps through how to get started with the CapTlivate™ ecosystem and the CapTlivate™ Capacitive Touch MCU Development Kit. The workshop consists of tool setup, an [out-of-box experience](#), and three labs. The labs all build on each other to teach capacitive touch tuning techniques, recommended workflow, low power optimization, and software development with the CapTlivate™ capacitive touch library.

### Workshop Summary

This workshop introduces the CapTlivate™ ecosystem through hands-on experimentation. The workshop is broken up into four sections which are described below.

1. Out-of-Box Experience
  - Explore the CapTlivate™ Capacitive Touch MCU Development Kit components
  - Experiment with the features and performance of the MSP430FR2633 CapTlivate™ MCU without writing or compiling any code
2. Lab 1: Using the CapTlivate™ Design Center to Create and Tune Sensors
  - Learn how to use the CapTlivate™ Design Center views
  - Learn the workflow between the CapTlivate™ Design Center and CCS
  - Learn basic capacitive touch tuning principles
3. Lab 2: Experimenting with Low Power Design Techniques
  - Learn how to use the wake-on-proximity feature
  - Learn how to tune for the lowest possible power consumption
  - Use EnergyTrace™ to measure the effects of tuning on power consumption
4. Lab 3: Using the CapTlivate™ Software Library
  - Learn about the sensor management features in the software library
  - Learn how to use callback functions to get proximity, touch, and position data

Upon completion of this workshop, the user will understand the basics of how to use the CapTlivate™ Design Center, the CapTlivate™ Touch Software Library, and the CapTlivate™ Capacitive Touch MCU Development Kit.

---

## Assumptions

This workshop assumes the following:

1. The user is reasonably familiar with MSP430 microcontroller architecture
2. The user is comfortable with Code Composer Studio v6 and its features, and has Code Composer Studio installed and licensed
3. The user is reasonably familiar with elementary capacitive touch principles

## Required Tools

The following tools are required to complete this workshop:

- CapTIvate™ Design Center PC GUI Tool
  - The CapTIvate™ Design Center is a PC GUI tool for configuring and tuning CapTIvate™ touch panels. An installer can be downloaded from its home page:  
[http://software-dl.ti.com/msp430/msp430\\_public\\_sw/mcu/msp430/CapTIvate\\_Design\\_Center/](http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/CapTIvate_Design_Center/)
  - Java run-time engine (JRE) version 1.7 or later is required.
- TI Code Composer Studio v6.10 or greater with MSP430 support and MSP430FR2633 service packs
  - Code Composer Studio can be downloaded from the following Wiki page:  
[http://processors.wiki.ti.com/index.php/Download\\_CCS](http://processors.wiki.ti.com/index.php/Download_CCS).

## CAPTIVATE-FR2633 Target MCU and CAPTIVATE-PGMR PCBs

The CAPTIVATE-FR2633 and CAPTIVATE-PGMR PCBs provide the MSP430FR2633 target MCU and programming/debug features, respectively.

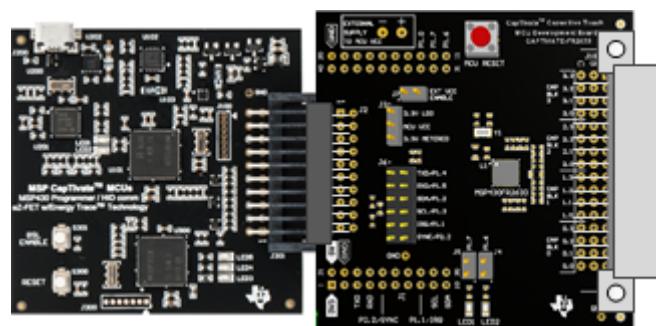


Figure 11.1: Programmer and Target PCBs

## CAPTIVATE-BSWP (Out-of-Box Experience Demo Panel):

The CAPTIVATE-BSWP is a capacitive touch evaluation panel that plugs into the EVM. It will be used to demonstrate a capacitive touch interface in this workshop. This panel demonstrates self-capacitance.

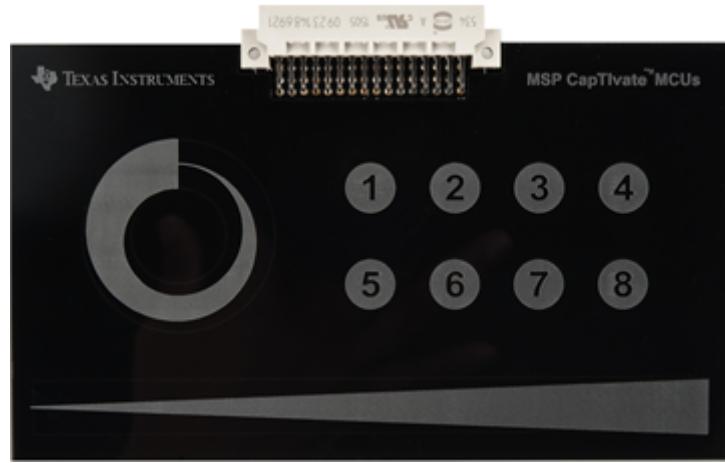


Figure 11.2: BSWP Demo Panel

For additional information about these and other demo PCBs provided in the CapTivate™ MCU Development Kit, click - [Kit](#).

## 11.1 Hardware Setup

Please perform the following:

- Connect the example sensor demonstration PCB to the CAPTIVATE-FR2633 MCU and CAPTIVATE-PGMR PCBs
- Connect the micro-USB cable between the CAPTIVATE-PGMR programmer PCB and your computer
- Verify the green LED (Power Good) is on and the green LED (USB Enumeration) is blinking.

Refer to the [FAQ](#) section for further troubleshooting tips.

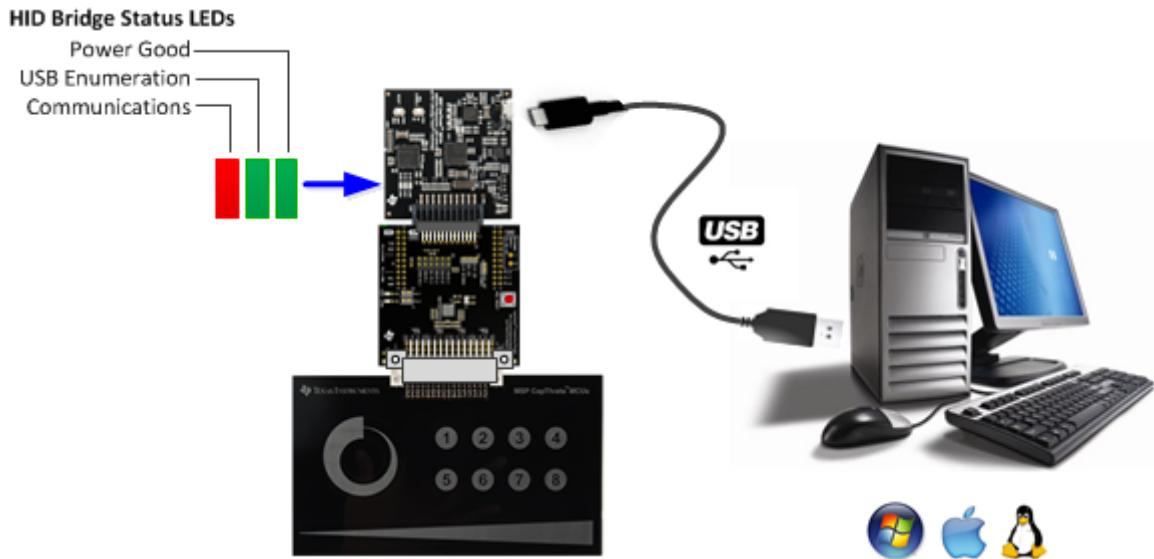


Figure 11.3: Typical Setup

## 11.2 Important note about wake on proximity behavior

To understand how the wake on proximity mode affects the out of box experience, the behavior is illustrated below.

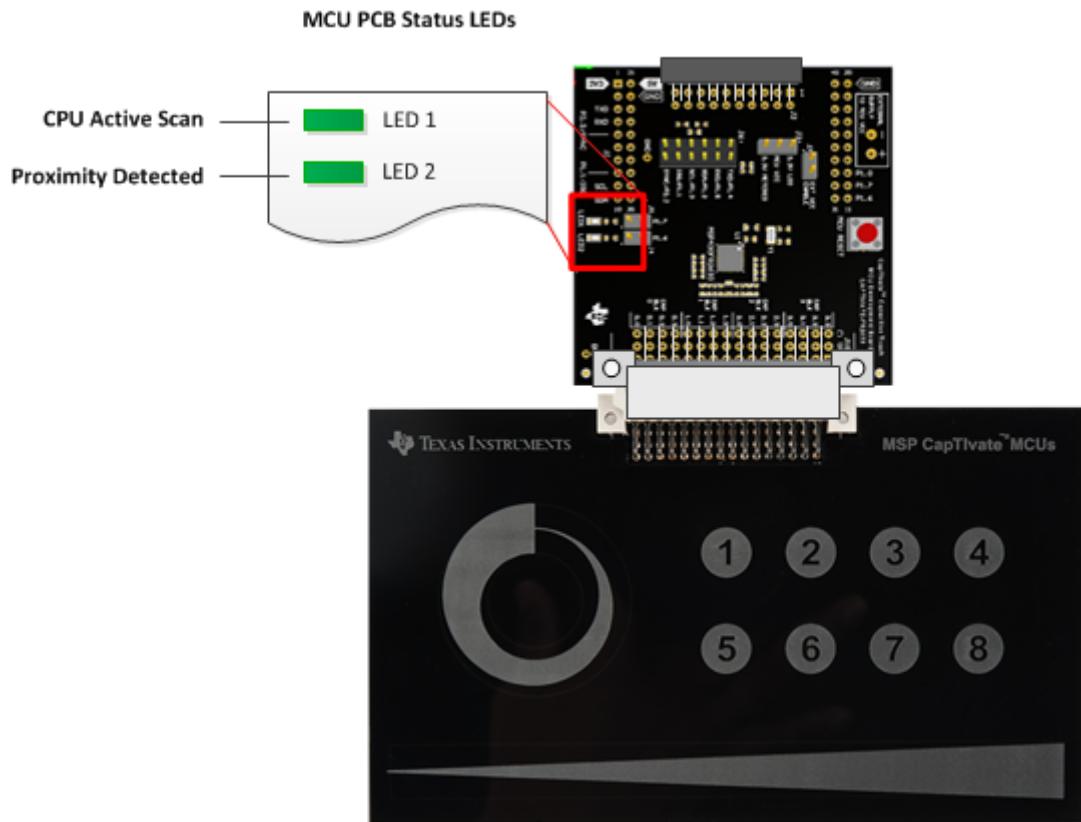


Figure 11.4: Out of box status LEDs

### Wake on proximity low power mode

The MCU board may appear to not be working during wake-on-proximity mode. There is no communications with the design center and both LEDs on the MCU board will be off. In this state the CPU is in low power mode 3 with the hardware state machine actively scanning only the proximity sensor at 10Hz until a proximity event is detected.

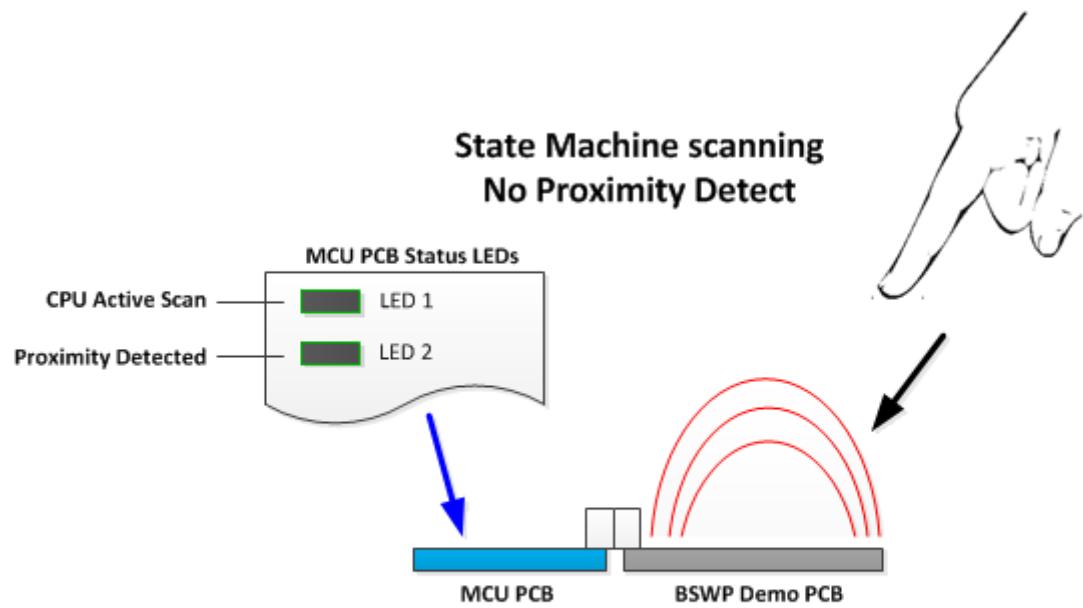


Figure 11.5: Wake on proximity mode

#### CPU scanning in active mode

Bringing your hand near the board will introduce a proximity condition causing the CPU to exit low power mode and resume actively scanning all the panel sensors at 30Hz as long as your hand is near or touching the panel. LED1 will blink at the rate the CPU is scanning the sensors. LED2 will be on while the proximity detect is true.

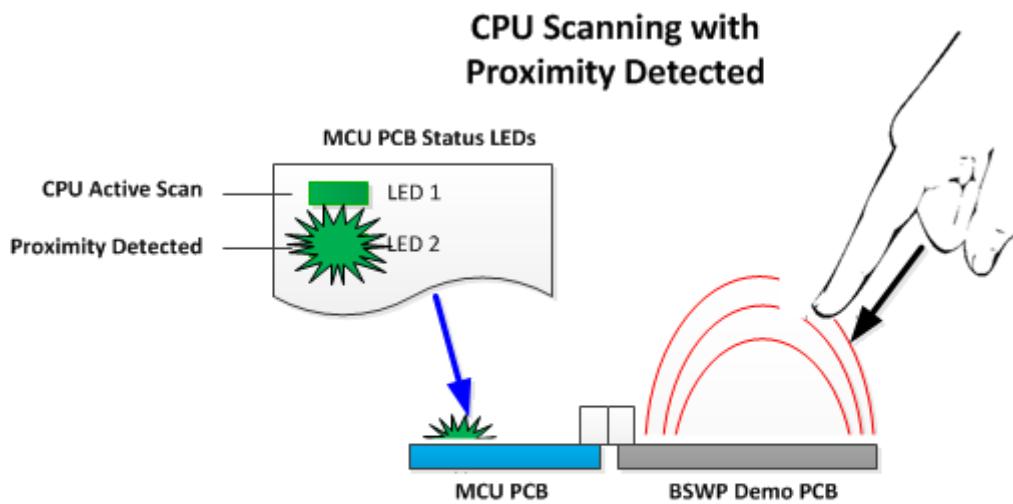


Figure 11.6: Active mode

#### Entering wake on proximity low power mode

After removing your hand away from the board, LED2 led will turn off, however, LED1 will remain on, indicating the panel is still being scanned for a short period by the CPU. After one second the CPU enters low-power mode 3, LED1 turns off and the hardware state machine is now actively scanning only the proximity sensor.

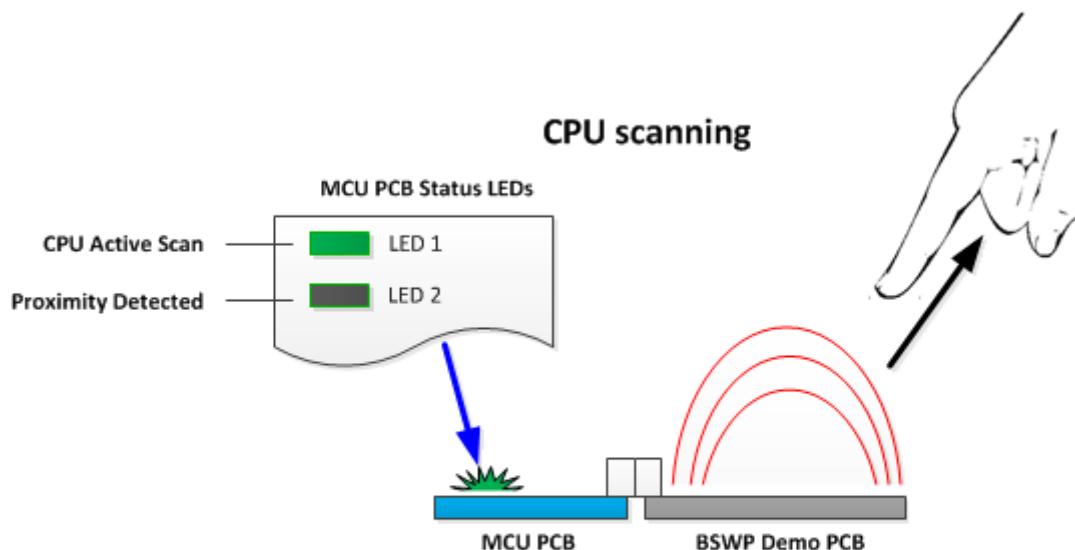


Figure 11.7: Entering sleep mode

## CAPTIVATE-BSWP Demonstration

### 11.2.1 Out of Box Experience

This demonstration uses the self capacitive BSWP (Button, Slider, Wheel and Proximity) Panel to explores the CapTlivate™ Design Center key features.

- Design and sensor tuning views
- Experiment with self capacitive button, slider, wheel and proximity

After completing this Out of Box Experience, continue with the workshop to learn more.

- How to create and tune sensors for desired touch and feel
- Low power sensor tuning considerations
- Using the CapTlivate™ Touch Library

**Note:** The CAPTIVATE-FR2633 MCU PCB comes pre-programmed from the factory with the CAPTIVATE-BSWP "Out of Box Experience" demonstration firmware. However, if the CAPTIVATE-FR2633 PCB needs to be re-programmed, [Reprogram Target MCU](#) before continuing.

### Starting the Demo

- Follow the Demo Hardware Setup instructions above.
- Start the CapTlivate™ Design Center using the icon that was installed on your desktop. Select "Project Open" and select the "CAPTIVATE-BSWP" project directory. [Need help with CapTlivate™ Design Center?](#)
- The project's [main window](#) is the design canvas and is pre-populated with the microcontroller and selected sensors matching the hardware for this demo.

- [Enable Communications](#) in the CapTlivate™ Design Center to begin the demonstration. Note: The MCU is in "wake on proximity" mode and there may appear to be no activity. Bring your hand near or touch the board to activate the MCU.

The Design Center canvas displays the current state of the sensors on the touch panel. Touch the various sensors on the panel, and observe the response on the canvas. The Design Center draws circular button indicators for button groups, and shows a slider, wheel and proximity depiction for slider, wheel and proximity sensors.

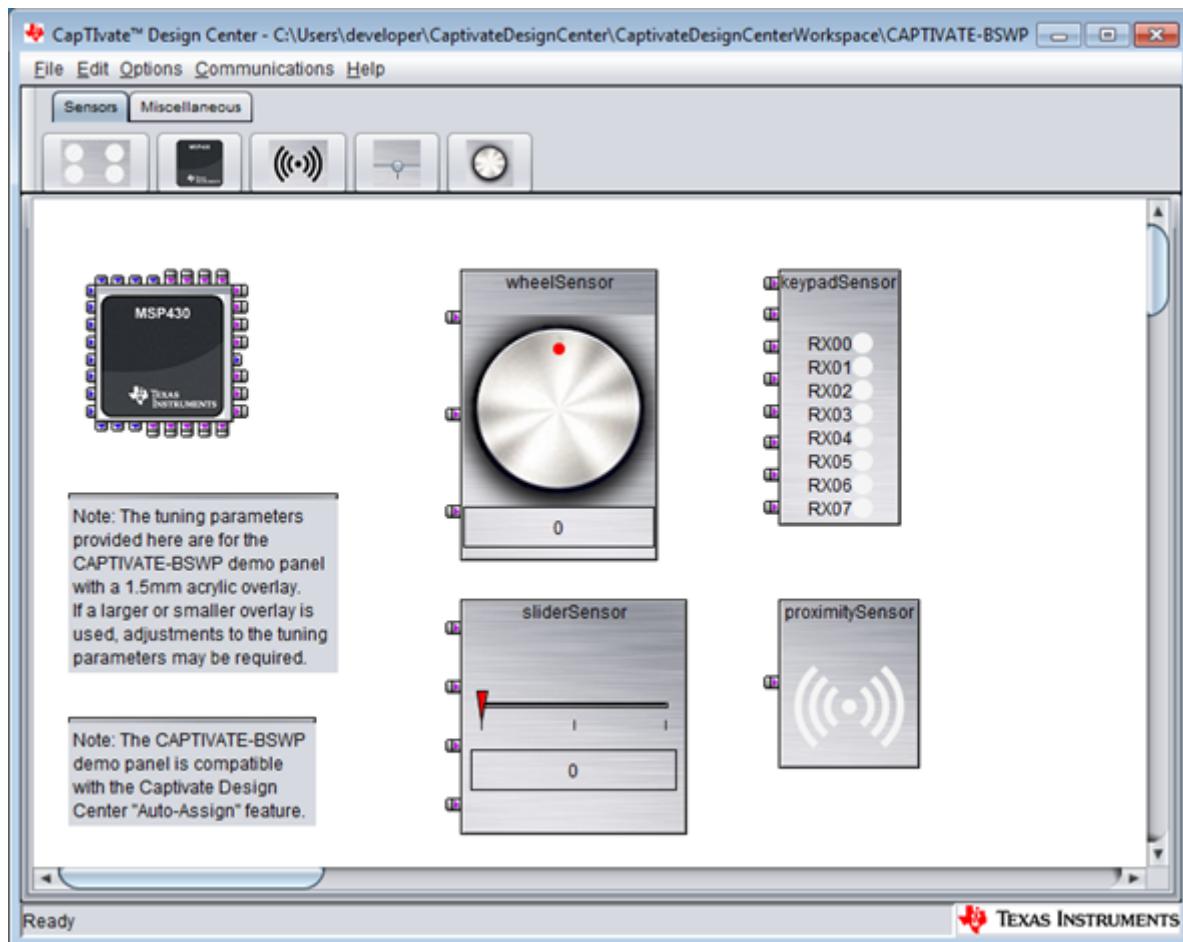


Figure 11.8: Design Center Canvas

The first thing to note is that there are indicators that flash different colors depending on the state of the element that they represent.

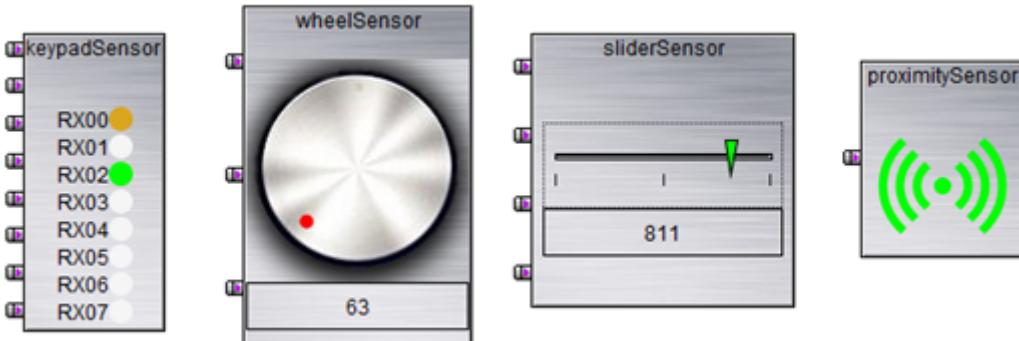


Figure 11.9: Sensor color states

Red on a slider or wheel sensor indicates that the slider or wheel is NOT being touched, and the needle or circle shows the last valid position. Green on a slider or wheel sensor indicates that the slider or wheel is being touched, and the needle or circle indicator shows the position. WHITE on a button group indicator or proximity sensor implies that there is no touch or proximity detection on the respective element. Green indicates that the element is being touched. Amber indicates that the element is in proximity detection. GREEN will always supersede AMBER, because when an element is in a touch detect state it is usually also in a proximity detect state. To see both the proximity detection LED and the touch detection LED at the same time, along with the raw data, let's open the controller customizer.

Double click on the MSP430 widget in the canvas view. The controller customizer window will open.



Figure 11.10: MSP430 Widget

The controller customizer window offers MCU configuration control as well as real-time viewing of element data from the entire panel at once. These different features are located on the different tabs of the customizer. The default tab for the controller customizer is the Configure Connections tab. This tab specifies the sensor-to-port mapping.

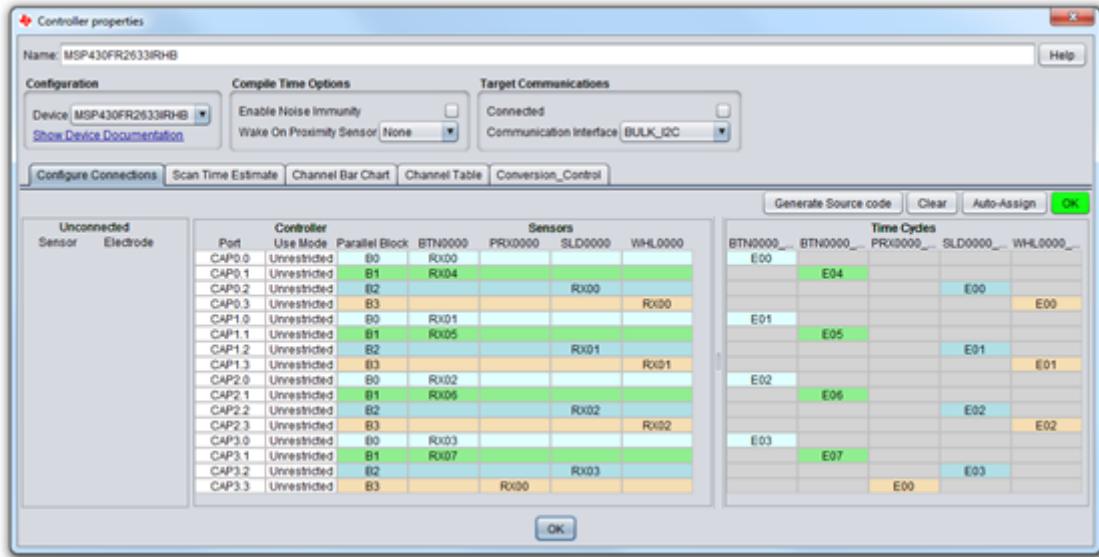


Figure 11.11: Controller Customizer View

Open the Channel Bar Chart tab. The Channel Bar Chart tab displays channel data for every element in the system. Proximity and touch indicators (Amber and Green, respectively) are shown below the bar for each channel. The channel's immediate count value is drawn as the blue bar in the plot for each channel. The green bar indicates the touch threshold, and the yellow bar indicates the proximity threshold. The LTA is drawn as a black bar, and usually appears hidden behind the count bar unless the element is being touched.

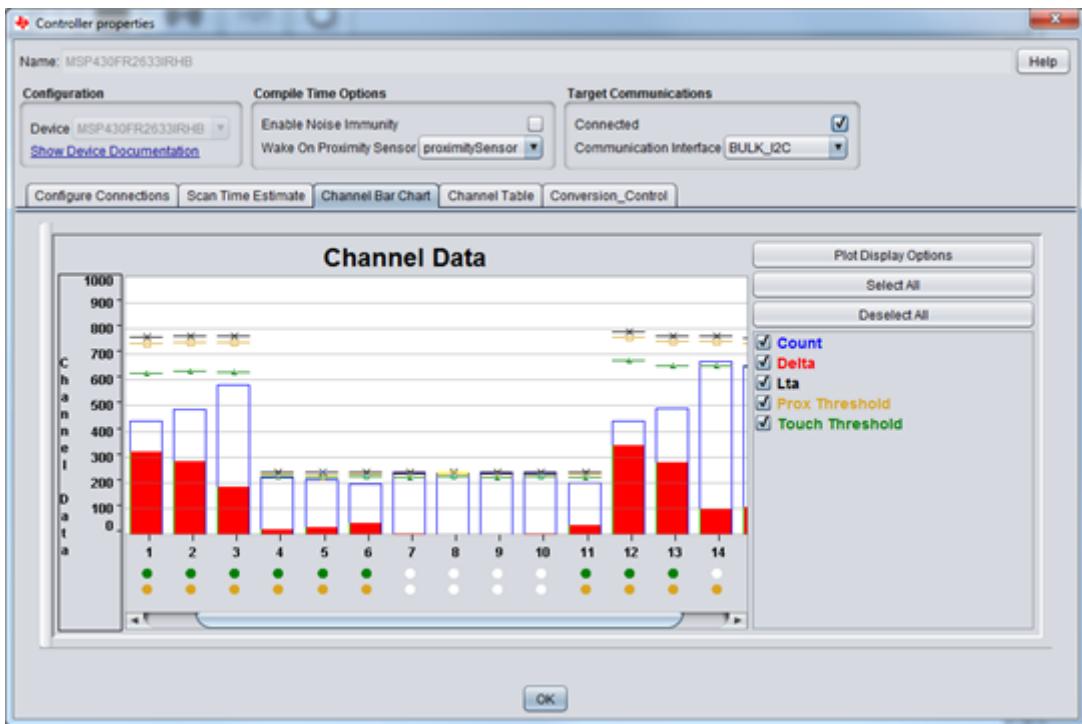


Figure 11.12: Controller Customizer View

This concludes the out-of-box demonstration. Feel free to continue to look through the Design Center views to see what's available.

Ready to start your first sensor design? Continue this workshop in the next section to learn how.

## 11.3 Creating a new sensor design project

### 11.3.1 LAB #1

In this section we go beyond the [Out of Box Experience](#) and will learn the following:

- How to create new sensor project using the CapTivate™ Design Center
- How to generate CCS/IAR target code
- How to tune sensors

### 11.3.2 Part 1 - Creating a New Design

Configuring the sensors is an important step. We want to take advantage of the target MCU's parallel scanning capability, and to do this we need to make optimal connections between the available CapTivate™ port pins and the electrodes in each sensor. Being able to scan electrodes in parallel can impact both the scan time and power consumption. In this section, we'll use the CapTivate™ Design Center's auto-assignment feature to generate an optimal pin assignment for the CAPTIVATE-BSWP panel.

The BSWP (Button, Slider, Wheel and Proximity) panel features several different self-capacitance sensors: a button group, a wheel, a slider and a proximity sensor. The diagram below shows the pin assignments and sensor layout.

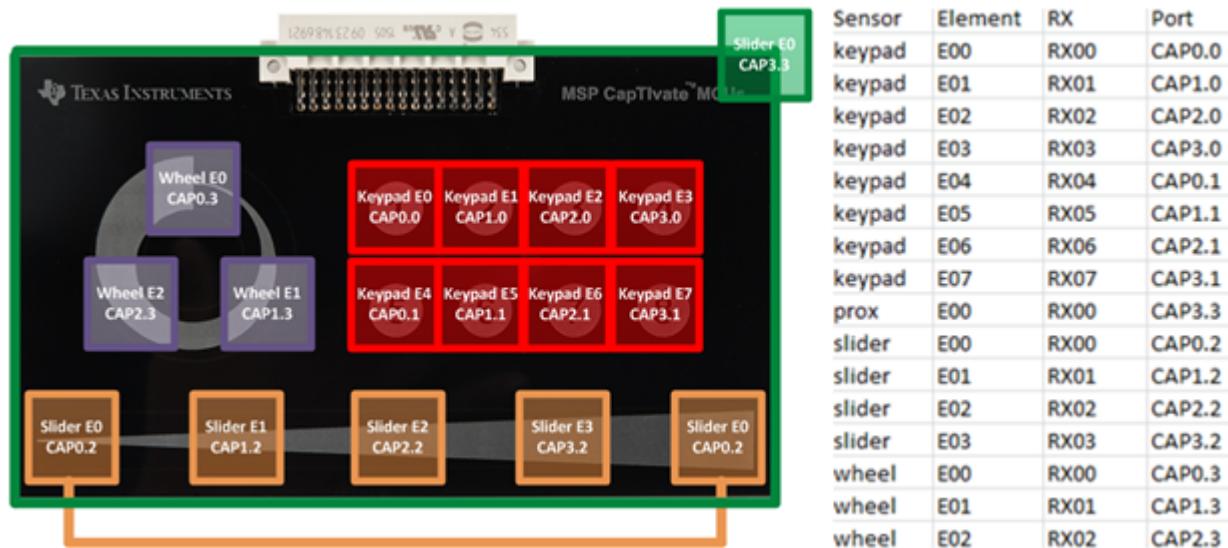


Figure 11.13: Demo Panel Electrode Connections

#### Create a new CapTivate™ Design Center project

Select File, Project New from the Design Center toolbar and provide "LAB1" for the project name. A blank workspace canvas should appear. If the Design Center asks if you would like to save any changes to the example BSWP project, select Discard, so that the example project can serve as a reference later.

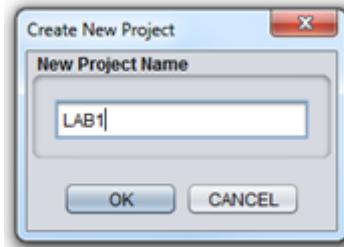


Figure 11.14: New Project Name

#### Instantiate Beans - Adding an MSP CapTivate™ Controller

Underneath the menu toolbar in the Design Center is the bean bar. The Design Center contains objects called beans that represent physical things in a capacitive touch system- a MCU, a button group, a slider, a wheel, etc.



Figure 11.15: Bean Objects Menu

All designs must have a controller. They may not have more than one controller (since we are designing a configuration for a single MCU). Let's add a controller to the workspace canvas. Select the MSP Controller bean

icon from the bean bar, and click on the canvas to place it. The canvas should now show a controller bean.

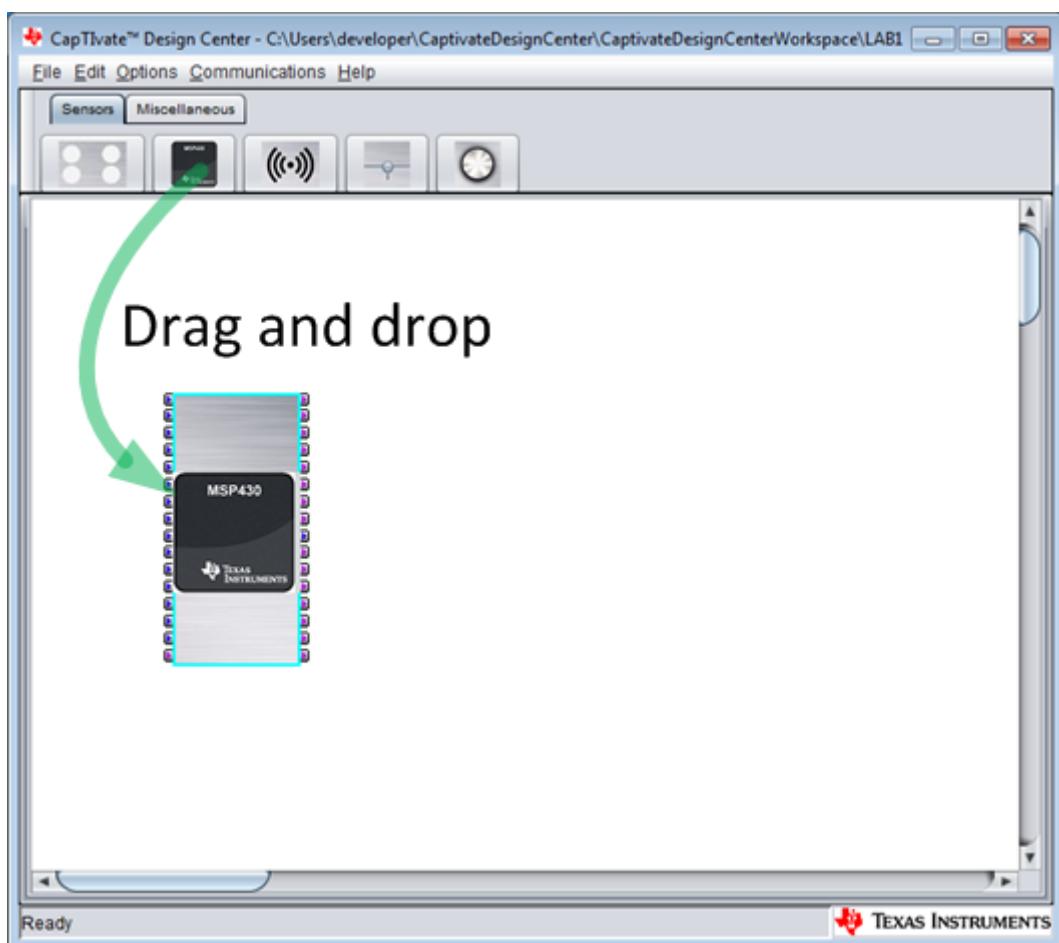


Figure 11.16: Add MCU Bean

Double click the MSP bean to open the controller customizer. In the controller properties view, select the MSP430FR2633IRHB (32-pin QFN) device from the configuration device drop down.

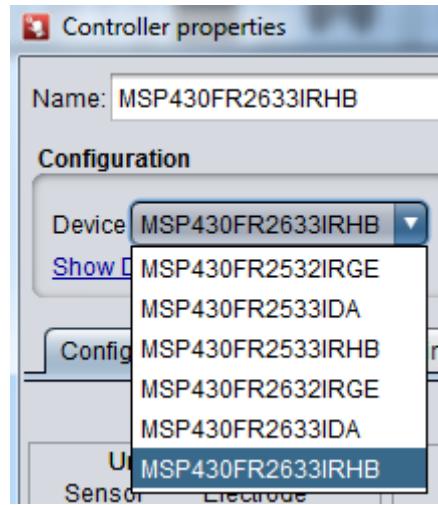


Figure 11.17: Select Device Variant

We will also need to configure the communications interface for communicating with the target MCU. We'll use I2C as it offers lower power consumption than the UART interface. In the controller customizer, select BULK\_I2C from the communications interface pull-down.



Figure 11.18: Select Communications Mode

Close the controller customizer. The bean will now change to represent the correct device and package that will be used in this workshop.

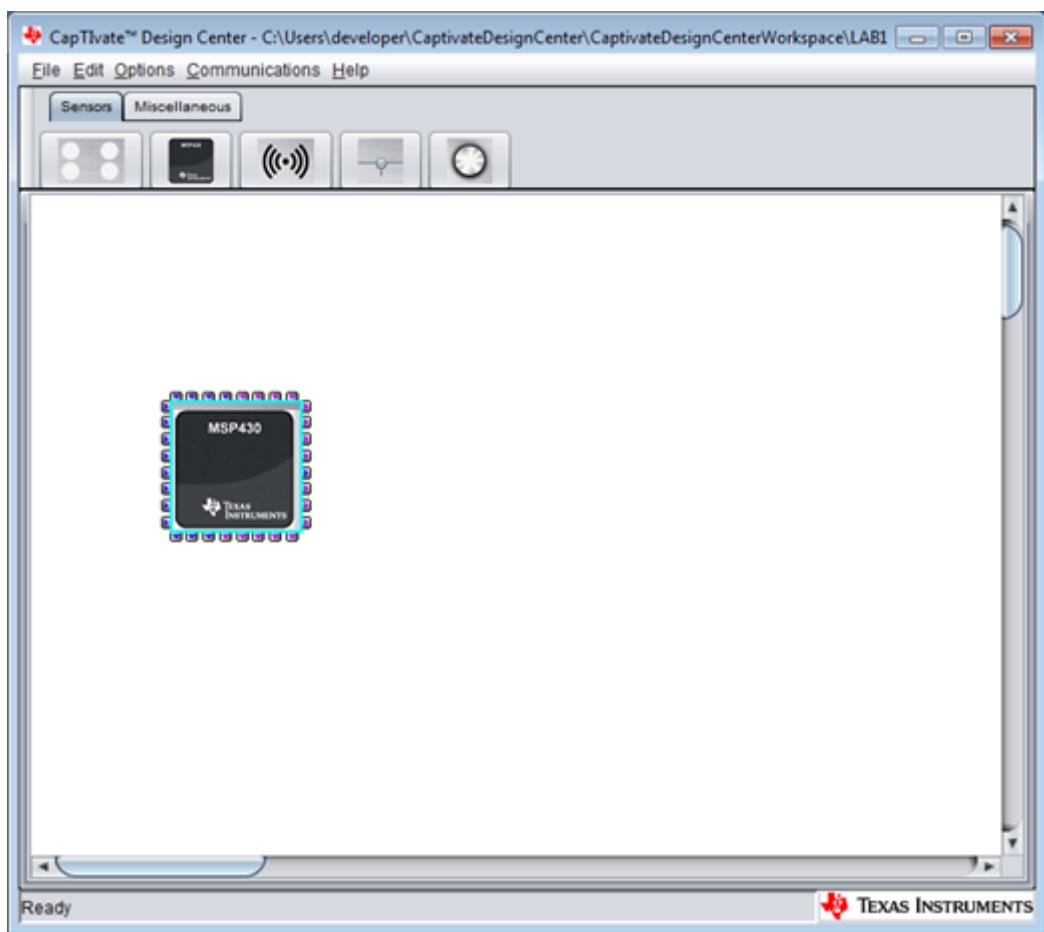


Figure 11.19: MCU Bean Configured

## Adding Sensors

Our panel has a button group (keypad with 8 buttons), a 4-element slider, a 3-element wheel and single element proximity sensor. Let's add a bean for each one of those sensors now. Just like the controller, the respective beans for each one of these sensor types can be added by clicking the bean icon, and dropping the bean onto the canvas. Now that we've added these sensors, the canvas should look similar to the image below.

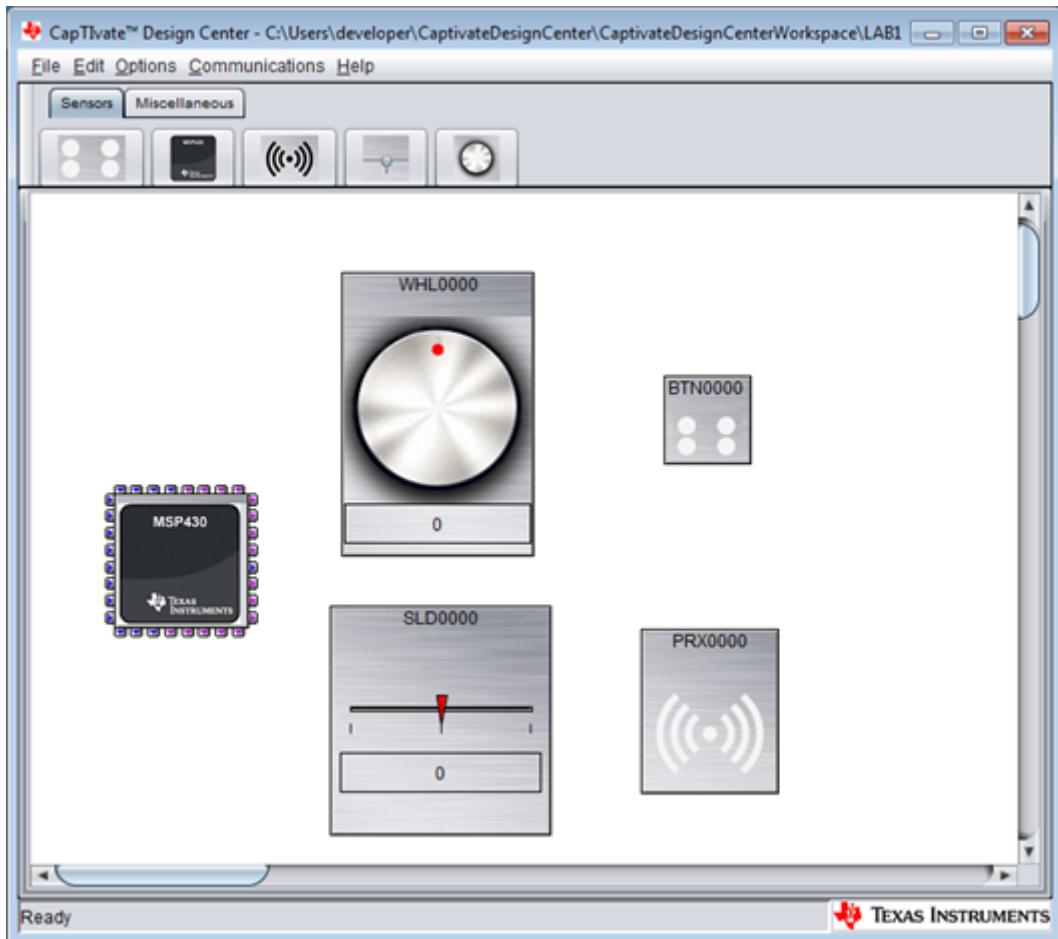


Figure 11.20: Add Sensor Beans

## Configuring Sensors

The canvas now contains all of the sensors in our system, plus the controller. Now we need to tell the Design Center more information about each of the sensors in our system. Let's start with the button group. Double-click the button group that we added to the canvas to open its customizer window. The customizer has a Configuration section at the top of the customizer. Since our panel is self-capacitance only, we'll want to specify SELF for the Capacitive Mode option. We have eight buttons that make up the keypad, so we will set the button count to 8. Notice that the Electrode Configuration drop-down will only show one option: 2 Cycles, 0 Tx Pin and 8 Rx Pin, since that is the only possibility for an eight-button, self-capacitance sensor. Your customizer should match the image shown here.

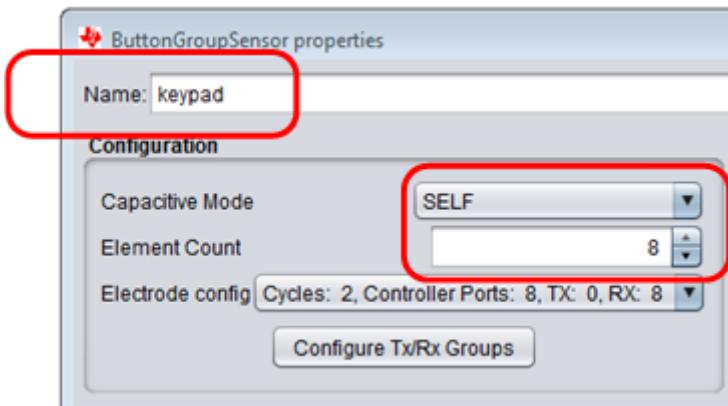


Figure 11.21: Add Sensor Beans

Let's also give the button group a creative name. The name is entered in the Name text box at the top of the customizer. If you remember back to the demo, it was named buttonGroupSensor. Here, it's named BTN0000, a unique default name provided by the Design Center. Let's name it keypad for this lab.

The same process is required for the slider, wheel and proximity sensors. Go ahead and setup the configuration for those sensors now to match the panel configuration (refer to the image at the beginning of this section for a reference). The Design Center assumes 3 elements for a wheel and 4 elements for a slider, which is exactly the configuration we have on this board. That means all we are really doing is naming the sensors and setting the capacitive sensing mode to SELF. Don't forget to give these sensors new names as shown in the diagram below. When you're finished, they should look like the following three images.

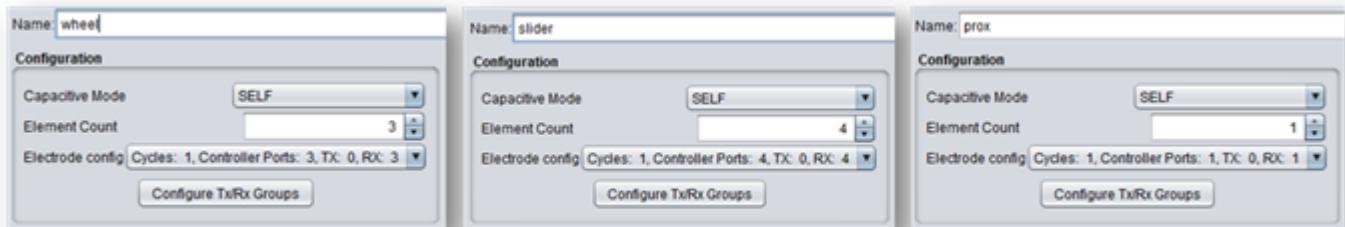


Figure 11.22: Sensors Configured

The canvas view should now show small connection points on the left side of the four sensors: 8 for the keypad, 4 for the slider, 3 for the wheel and 1 for the proximity sensor. Each connection point indicates a CapTIvate™ port that is required for that sensor. Next, we'll be mapping these connection points to actual ports on the MSP430FR2633 MCU that we selected to be the controller.

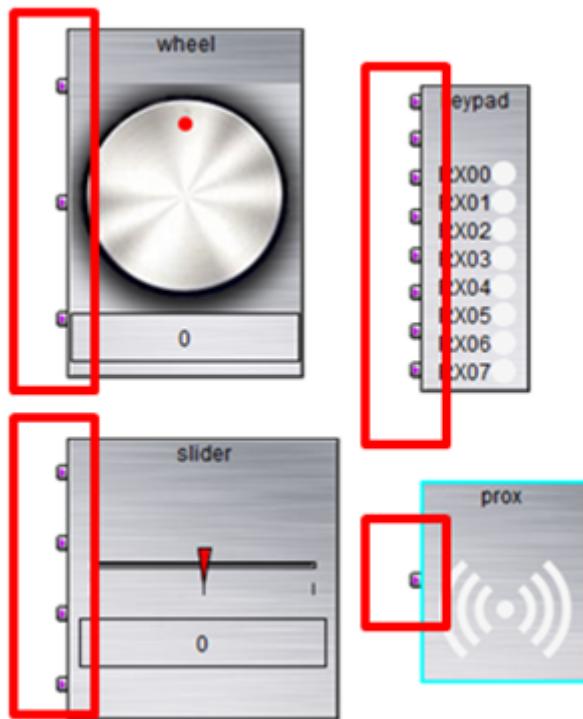


Figure 11.23: Unassigned sensor electrodes

#### Auto Assign Sensor Connections

As we pointed out at the beginning of this section, we are going to let the Design Center automatically map sensors to ports. This is the recommended way to map ports because the Design Center uses an algorithm to determine the most optimal configuration. Port assignment is done in the controller customizer. Open the controller customizer. This is the first bean we added- the one that looks like a device. The customizer opens with the Configure Connections tab selected, as shown below. The sensors we added in the previous step are listed in the "Unconnected Sensor Electrode" box at the far left of the tab, indicating that nothing has been connected yet. Notice that the Generate Source Code button is grayed out and unavailable. This is because a valid port mapping is required to generate source code. The middle box on the Configure Connections tab has one row for each available CapTivate™ port. The Use Mode column of the table has each pin to "Unrestricted" by default. This means that the Auto-Assign feature assumes it has access to all of the ports listed here. It's possible to reserve a port (say, if you wanted to use it for a SPI interface) by changing the "Unrestricted" option to "Reserved." The BSWP panel that we are designing now requires all 16 ports, so let's leave all of the ports as "Unrestricted."

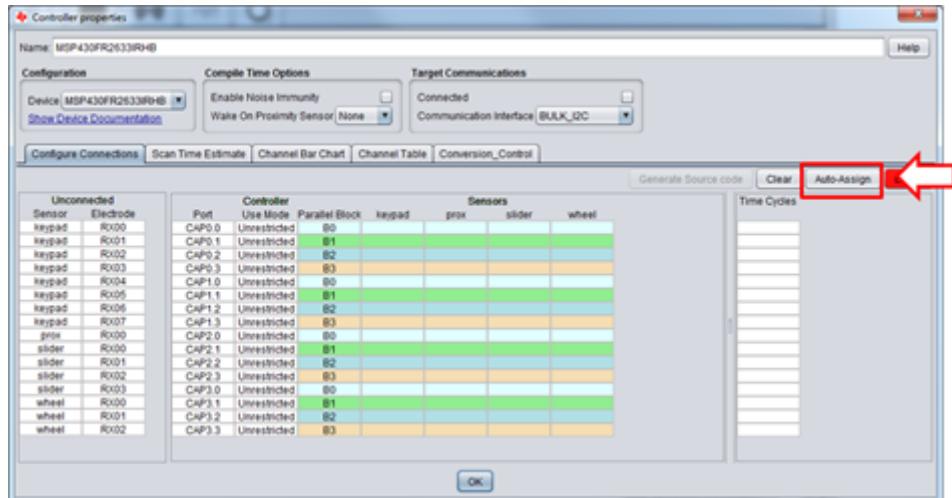


Figure 11.24: Auto Assign Feature

When complete, you should see a configuration as shown below. Notice how the indicator is GREEN and shows OK, indicating all the sensors have been connected to controller port pins with no conflicts. Also notice that all 16 CapTivate™ I/O pins have been used.

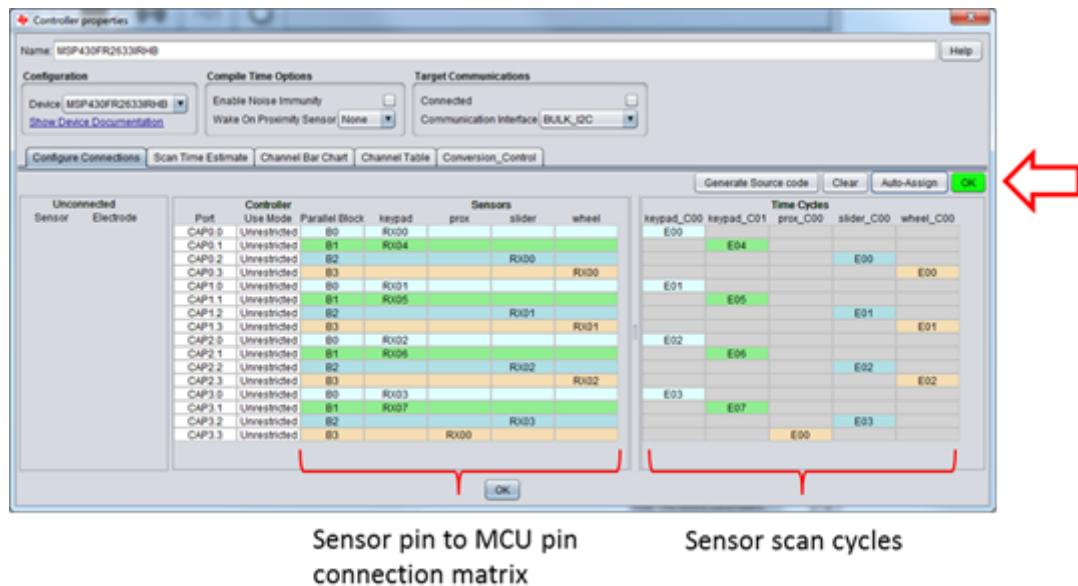


Figure 11.25: Auto Assign Successful

Now that the port mapping is known, the time cycles are calculated. The 3rd and final box on the "Configure Connections" tab shows a breakdown of the time cycles. A time cycle is a group of elements that will be measured in parallel (at the same time as each other). Notice that the keypad sensor will be measured in two time cycles, and the proximity, slider, and wheel sensor each require one time cycle each. If the device only supported sequential measurements, this panel would take 16 cycles to measure each sensor! With this device we are doing it in just 5, reducing measurement time and power consumption.

**NOTE:** We were able to use the Auto-Assign feature here to map the ports for a board that already exists. This is because the Auto-Assign feature was used to generate the original port mapping for the CAPTIVATE-BSWP panel.

Before we move on, we want to save our current Design Center project. Select "Project Save As" and change the name of project folder to LAB1 as shown below.

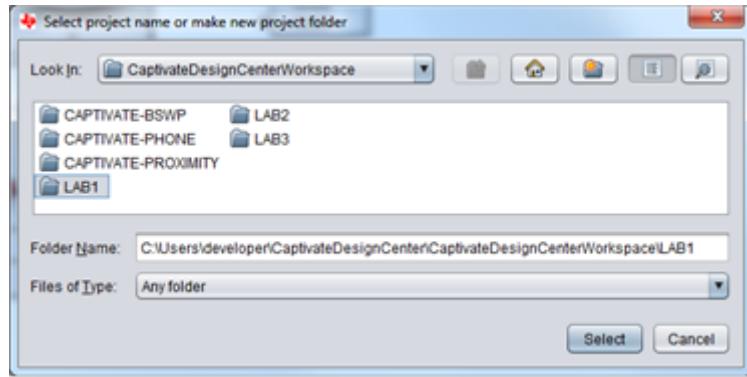


Figure 11.26: Save CapTlve\textrademark {} Design Center Project

#### Generate Starter Project Target MCU Code

Now that we've created a sensor configuration for our sensors, the next step is to generate a starter code project. A starter code project is a fully functional Code Composer Studio and IAR firmware project for the MSP430FR2633 that we can use to program our target and begin the tuning process. Generate starter code project. In the controller customizer, under the Configure Connections tab, select the Generate Source Code button.

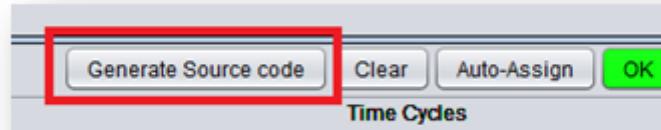


Figure 11.27: Generate Starter Code Project

Save starter code project. A dialog box will ask whether we want to create a new project or update an existing project. Select "Create new project", then click "OK" to accept the default location for our new starter project.

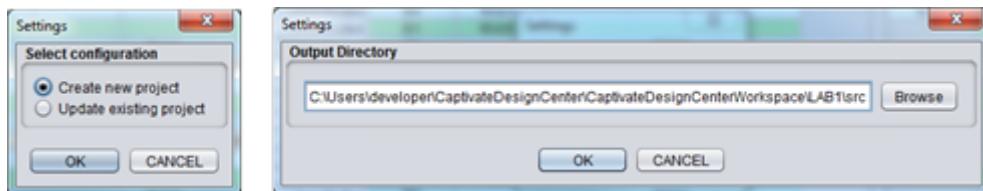


Figure 11.28: Save Starter Code Project

The Design Center will create the starter code project under your Design Center Workspace directory. In the next step, you will import a copy of this project into your CCS workspace directory.

#### Build Lab#1 Starter Project Code

Import starter code project into CCS. Now that we've generated our LAB1 starter project source code, import the new project to your CCS workspace. If you need a CCS "refresher", follow the steps in a later section CCS - Building and programming. After importing the project, click the debugger icon to build and download the code to the target.

Note: The very first time a CapTlve source code project is built in CCS, the compiler may issue the following message if the small code/ small data run-time library needs to be built. Depending on your PC, this may take

---

several minutes.

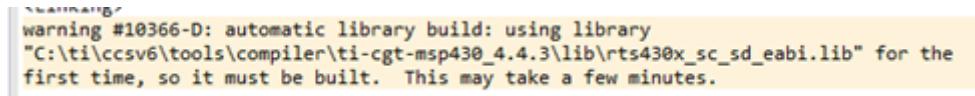


Figure 11.29: Small memory run-time library build message

Since we won't be using the debugger at this time, set the target running in "free run" mode or simply terminate the debug session. Press the reset button on the MCU PCB.

**Note: After a POR or RESET, the MCU requires a second or two to calibrate the sensors before communication begins.**

Now that we've programmed the target with our start project, let's view the real time data and start tuning our sensors! Communicate with the Design Center. In the Design Center, close all of the customizer windows so that only the main canvas is visible, showing all of our sensors. Select Communications, Connect from the Design Center menu bar. The sensor data that is being streamed from the target will now appear in the main view of the BSWP demo board.

### 11.3.3 Part 2 - Tuning Sensors

At this point, we have not yet specified what kind of tuning we want for each sensor. Each sensor is operating off of the default values that were assigned by the design center. These default values are good starting points, but there is room for improvement. That is the goal of Part 2- to improve upon the defaults and dial in the values that are best for this application. Let's look at the individual element data for the 16 channels in our system. Open the Controller customizer and select the Channel Bar Chart tab. All 16 channels will be showing their element data here. While touching the various sensors, the BLUE bars (absolute count measurement) will decrease since this is a self-capacitance panel. Solid RED bars indicate the delta between the long term average (LTA) and the current count. These will increase when an element is touched. Note the position of the default thresholds relative to the counts as you interact with the panel.

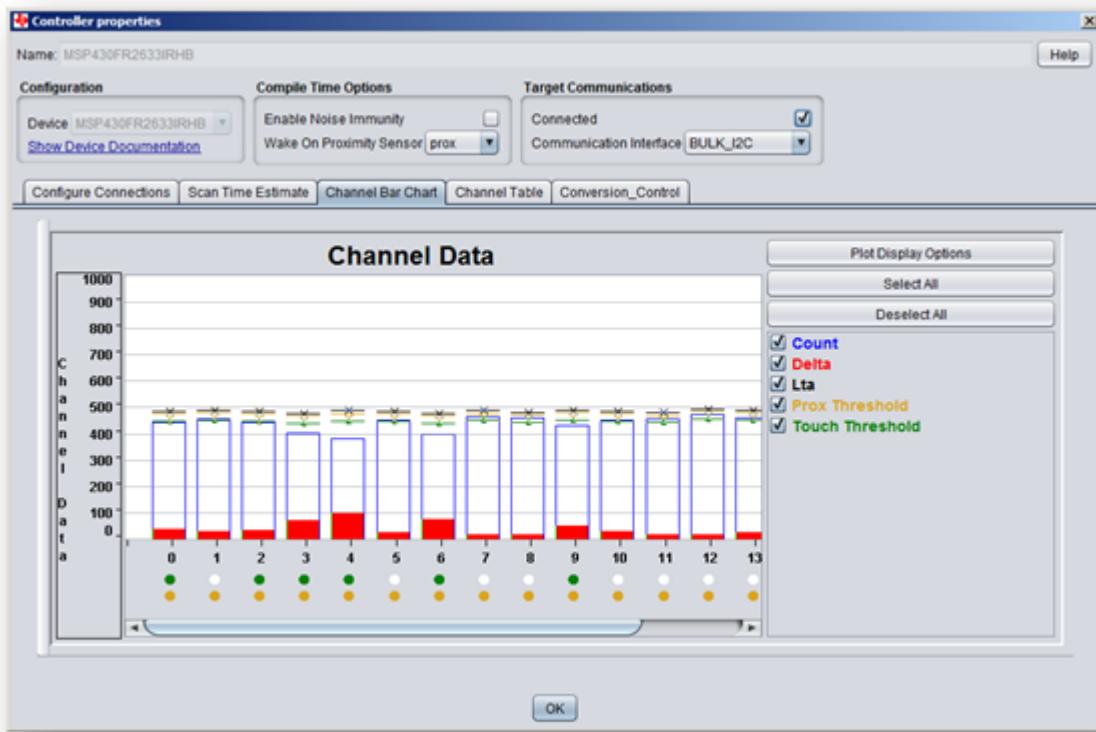


Figure 11.30: Channel Bar View

The design center assigns default proximity and touch thresholds for all sensors. These values will have to be fine-tuned for each element, since each element has a slightly different sensitivity and parasitic capacitance. You can examine these defaults by opening the CAPT\_UserConfig.c file in CCS. Search for ".ui8TouchThreshold" or ".ui16ProxThreshold." Notice that proximity threshold is assigned to the sensor while touch thresholds are assigned to each individual element in the sensor.

Note: the proximity threshold value is an absolute deviation from the LTA (long term average) whereas the touch threshold value is dynamic, expressed as a percentage of the LTA. To learn more, double click on a sensor, click on the Tuning Tab, then on the Proximity Threshold or Touch Threshold Parameters to bring up the integrated help topics.

Tuning is done on a sensor-by-sensor basis. Each sensor has a customizer window that is opened by double-clicking the sensor bean on the canvas.

#### Tuning a Button Group Sensor

Let's start with the button group sensor first.

**TIP:** To assist us in the sensor tuning process, any one of the sensor's tabs can be "un-docked" allowing us to view the real time data in one tab while modifying a tuning parameter in another tab.

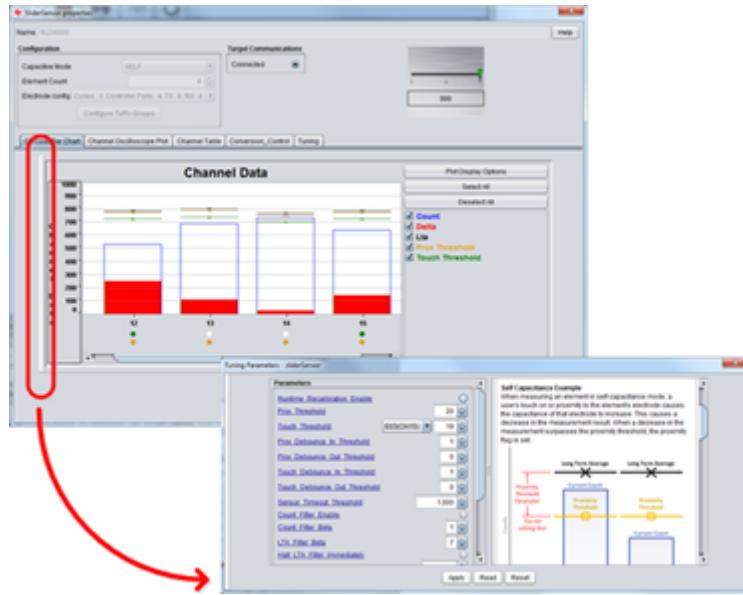


Figure 11.31: Undocking Tabs

#### Conversion Count and Conversion Gain

Let's experiment with Conversion Count and Conversion Gain. Close the controller customizer, and open the keypad sensor customizer by double clicking the keypad sensor on the canvas. In the customizer, we can see the same bar chart as we saw in the controller view, but here we only see 8 bars for the 8 elements in the keypad sensor. Touch button #1 on the panel while viewing the data in the Design Center. Notice that a touch produces around 75 counts of delta in the measurement - it goes from ~500 counts (also the default conversion count specification), down to ~425 when touched.

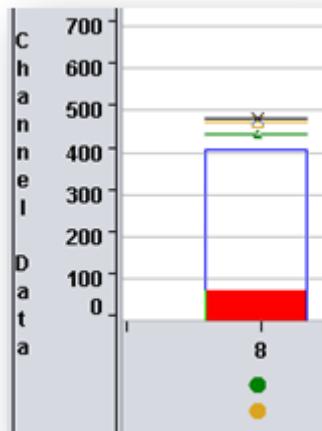


Figure 11.32: Close up Button Bar

It's nice to have 75 counts of delta, but do we really need that much information to determine a touch condition? The system is quite stable (only a few counts of noise), so the extra resolution isn't really necessary, and it comes at the cost of scan time and power consumption, which we will investigate in a later lab. Let's see if we can still get enough information out of the sensor to determine a touch condition if we bring the conversion count to 250, instead of the default 500.

Open the Conversion Control tab in the button sensor customizer.

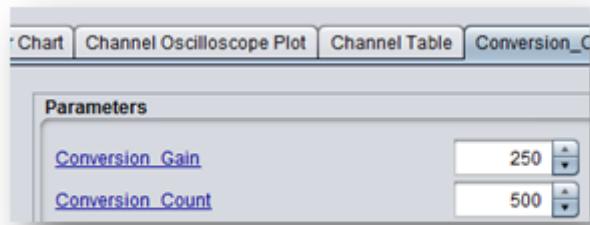


Figure 11.33: Conversion Control

If you did not change any of the default settings, your conversion control window should look like the one above. The Design Center has selected a Conversion Count of 500, and a Conversion Gain of 250. We've shown that this provides the adequate information we need to determine a touch, but let's dial the numbers back a bit. Let's try 250 for Conversion Count and 100 for Conversion Gain. Note that Conversion Gain is the "base" value that the MCU will try to achieve without using parasitic capacitance offset subtraction. The overall Conversion Count is the count we are trying to achieve after we've added offset subtraction.

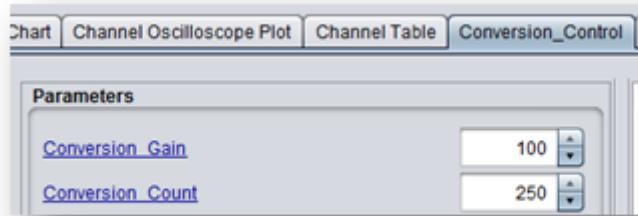


Figure 11.34: Conversion Control

After a tuning parameter is adjusted in the Design Center, the Apply button lights up yellow, reminding you that you have parameter changes that need to be applied to the target microcontroller. Let's press the Apply button now to submit the changes. Switching back to the Channel Bar Chart tab, the changes should be immediately visible. The baseline (no touch) measurement is now normalized at ~250 counts, and touching the button should decrease the counts to ~200, providing a delta of ~45-50 counts.

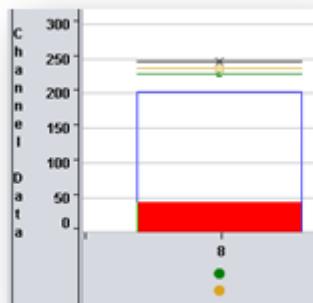


Figure 11.35: Close up Button Bar

## Proximity and Touch Thresholds

Now that we've got an appropriate measurement resolution, let's adjust the proximity threshold and touch threshold to appropriate values that provide the desired feel for the button. While pressing button #1 on the panel, look at the bar graph you will see the green bar represents the location of the touch threshold and the amber bar represents the proximity threshold. Since our blue bar (current count) is past both the proximity threshold (amber) and the touch threshold (green), both the proximity and touch light is illuminated.

Since a full touch provides  $\sim 50$  counts of delta, the default touch threshold makes the element a little too sensitive. We require a firm touch that is centered on the button. Let's edit the touch threshold to require a firmer touch.

The touch threshold defines the size of the delta on a given element that is required for that element to enter a touch state. Touch thresholds are specified as a percentage of the long term average (LTA). Because of this, the actual absolute value is calculated dynamically at runtime. In the Design Center, they are entered in units of 1/128 the long term average (LTA). By specifying touch thresholds as a percentage of the LTA, the threshold is kept proportional to the LTA even as the LTA drifts. The default touch threshold value is 10. Since we specified a conversion count of 250, our effective touch threshold is now  $250/128 * 10$ , or 19. Viewing the touch threshold in the Channel Table now shows the absolute threshold = 231, or 250-19.

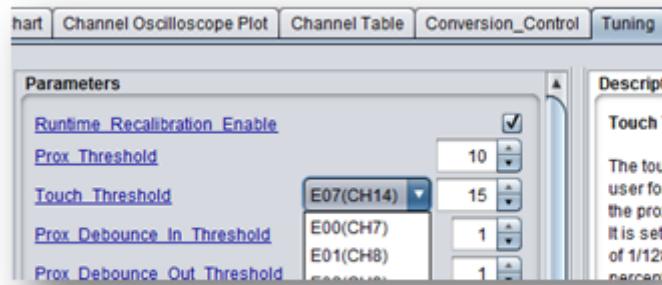


Figure 11.36: Proximity and Touch Thresholds

Let's increase the touch threshold to 15 for this system. A touch threshold of 15 would provide an effective touch threshold of  $250/128 * 15$ , or 29. This will require a firmer touch for a touch to be declared. The touch threshold is set in the Tuning tab of the sensor customizer, just like the Conversion Count and Conversion Gain. Since the keypad is made up of 8 elements numbered from E00 to E07, button #1 corresponds to electrode E00 (CH8), so we want to modify the touch threshold for all 8 elements (E00 to E07). These changes improve the feel of the buttons.

If you'd like to understand more about how touch thresholds work, read the Description box in the Design Center when editing one. Feel free to adjust the proximity threshold. Note that there is only one proximity threshold. This threshold applies to the entire sensor.

## Count Filtering

When looking at the bar graph, it's easy to see that the count value drops 50 counts almost instantly when the button is touched. Let's adjust the count filter to be more aggressive. In the Tuning tab, set the count filter beta value to 2 (the default is 1). This means that we will have a 25% weight for each new value, as opposed to a 50% weight- so the sensor should respond twice as slow. We might also see a little bit less noise on the system due to the increase in the filter.

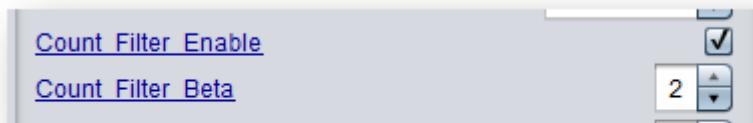


Figure 11.37: Count Filters

Don't forget to hit Apply after you've made the change. Now, looking at the bar graph, the system should show a slightly slower response. One neat way to look at filter responses is the Channel Oscilloscope Plot tab. This is an alternative view to the bar chart that shows history, and it's a nice way to look at responses to filter changes.

You may notice that there is a count filter beta value as well as a LTA filter beta value. The LTA filter beta value effects how fast the long-term-average (LTA) tracks changes in the base count when no one is touching the panel (no threshold crossings). The default filter beta for the long term average filter is 7. The available options for both the count filter beta and LTA filter beta are 0 through 7. The filter doubles in strength with each increase. As we've seen, moving from 1 to 2 doubles the response of the filter to a step input.

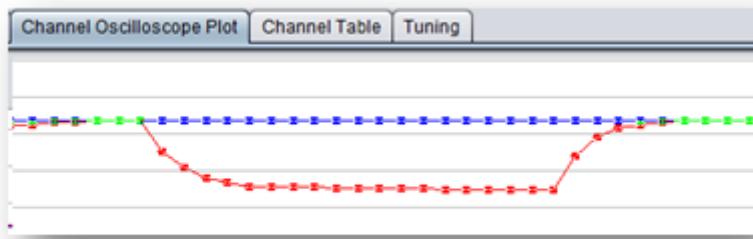


Figure 11.38: Oscilloscope Tab

By this point, the button should have a decent initial tuning, and we've learned a little bit about filter responses.

#### Tuning a Wheel Sensor

Let's take a look at another sensor type- the wheel sensor. Close the button sensor customizer, and open the wheel sensor customizer by double-clicking the wheel sensor bean on the canvas.



Figure 11.39: Wheel Sensor View

#### Wheel Conversion Count and Conversion Gain

The wheel sensor also has a default Conversion Count of 500, with a Conversion Gain of 250. You should see about 100 counts of delta as you run your finger across the wheel. Unlike the buttons that we tuned previously, the wheel uses each element's delta as an input to a position function to calculate where you are touching on the wheel. Thus, here it is beneficial to increase the conversion count to provide additional resolution. Let's set the Conversion Count to 800 and the Conversion Gain to 125.

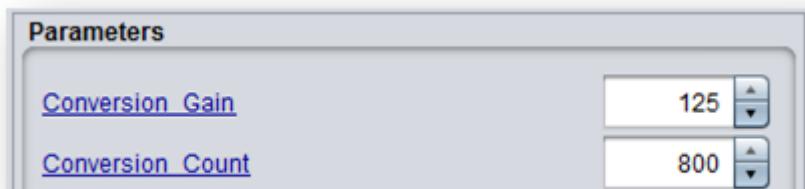


Figure 11.40: Conversion Control

This should increase the maximum delta on each element by about a factor of 3. This increased resolution will greatly benefit the position calculation.

#### Wheel Resolution

The default wheel position resolution is 1000 points, providing 0 to 999 positions. Most applications won't need this much resolution, so let's reduce it to a more usable value of 100. Change the desired resolution tuning parameter to 100 from the default of 1000. After applying the change, notice how the response of the wheel is still just as linear as it was at 1000 counts, but now the data is actually much more usable, since it's easy to dial in any specific number. The algorithm supports up to 16 bits of resolution.

## Wheel Sensor Position Filtering

The other knob we're going to play with is the position filter coefficient. The sensor filter can be adjusted in 255 different steps. 255 would represent a unity response (no filtering), and 1 would represent the strongest filter. Let's pick up the response time of the wheel even more by changing the wheel filter coefficient to 150 from the default of 100. The wheel response should be quite fast now, and there should still be enough filtering to provide a stable position report with little jitter. For fun, set the sensor filter to 25 and see how the wheel responds to over-filtering.



Figure 11.41: Filter Control

## Wheel Touch Thresholds

Since we increased the sensitivity significantly in the Conversion Count / Conversion Gain step, the wheel is going to be extremely sensitive to touch. Notice that if you hold a finger hovering over the wheel, the elements enter a touch state well before you actually touch on the panel. We'll update the values here just as we did with the button group sensor. A setting of 27-28 is much better than the default value of 10. As always, remember to update the touch threshold for all 3 elements.

## Tuning a Slider Sensor

Now that we've set up the wheel, let's turn our attention to the slider.

### Slider Conversion Count and Conversion Gain

The slider is just like the wheel when it comes to conversion count and conversion gain. Let's set the Conversion Count to 800 and the Conversion Gain to 125 for the slider as well to increase the sensitivity and provide more resolution to the position algorithm. Double-click the slider sensor to open its customizer and click on the Conversion Control tab.

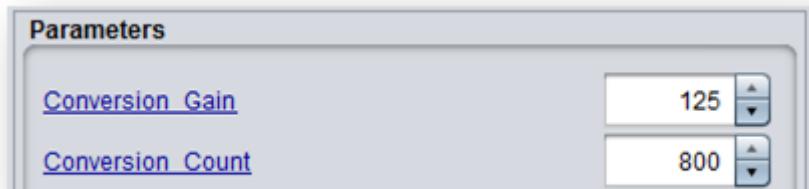


Figure 11.42: Conversion Control

## Slider Endpoint Trimming

The slider is very similar to the wheel- with the exception that it's chopped and has two endpoints. All of the same principles apply, but there will be two new parameters: Slider Lower Trim, and Slider Upper Trim.

Select the Tuning tab in the customizer. The Upper Trim and Lower Trim parameters are the last two parameters in the parameter window so use the scroll bar to view them.

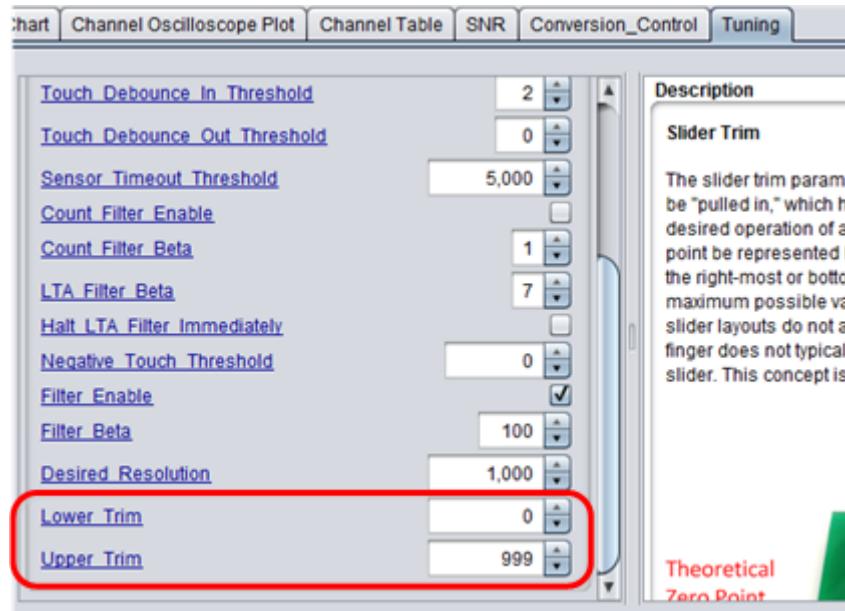


Figure 11.43: Slider Trim

While moving your finger up and down the slider, you may notice that the slider values only seem to go from about 40 up to 960, even though we've stated we want a resolution of 1000 (0 to 999 positions) for this sensor. The reason we never get to 0 or 999 is because we've essentially taken a wheel and cut it in half- and the end points never get "pulled" by their opposite elements. To counteract this, there is a mechanism to stretch the slider by trimming the edges. Currently, the lower trim value is set to 0, and the upper trim value is set to 999. This means that there is no stretching being applied. We can apply stretching by providing trim values to the slider to re-align our zero point and max point. To set the slider trim values, place your finger on the left-most part of the slider, and look at the response. It will likely be somewhere above zero, but less than 100. We want that position to be defined as zero, so that the user can fully get to 0 when using the slider. Whatever measurement you get when touching the left-most part of the slider, enter that in the Lower Trim parameter box in the Tuning tab. Then, place your finger in the right-most part of the slider. You will likely see a value between 900 and 999 (since we are going out to 1000 points of resolution). Whatever measurement you see, place that in the Upper Trim parameter box. Then Apply the changes to the controller, and review the results. For this slider, the values below work well with the other settings that have been specified.

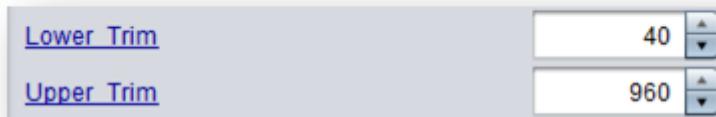


Figure 11.44: Selecting Slider Trim Values

That's it! Outside of the trim values, sliders behave exactly like wheels.

#### Slider Touch Thresholds

Since we increased the sensitivity significantly in the Conversion Count / Conversion Gain step, the slider is going to be extremely sensitive to touch. Notice that if you hold a finger hovering over the slider, the elements enter a touch state well before you actually touch on the panel. We'll update the values here just as we did with the button group sensor and wheel sensor. A setting of 23-25 is much better than the default value of 10. As always, remember to update the touch threshold for all 4 elements in the slider.

## Proximity Sensor Tuning

The proximity sensor has all of the same parameters as a button group, but there is only one element. Let's set up the proximity sensor at see what kind of proximity distances we can achieve.

### Proximity Conversion Count and Conversion Gain

The default Conversion Count of 500 and Conversion Gain of 250 do not provide very much sensitivity. With the default proximity threshold of 10, the sensor only yields about 1cm of detection distance. Let's turn up the sensitivity by setting the Conversion Count to 800 and the Conversion Gain to 100.

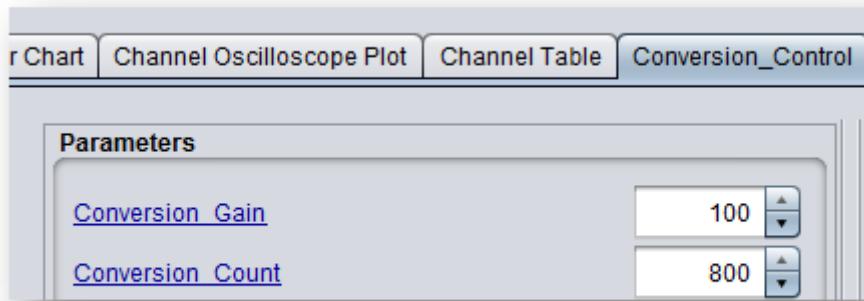


Figure 11.45: Selecting Conversion Control Values

Notice the improvement in sensitivity! We are now able to get over 5cm of detection distance.

### About Re-purposing the Touch Threshold

Since we are using the proximity threshold to know when someone is close to the panel, we could repurpose the touch threshold for another task. Since the proximity sensor layout on the CAPTIVATE-BSWP panel travels around the PCB, it could be used to detect if someone was leaning up against the panel, or putting their whole hand down on top of it. This kind of information could be used to block outputs of other keys. Imagine a security keypad application that is mounted on a wall. If someone leans up against it, it would be nice if we could detect that and prevent all of the other sensors from triggering at the same time. The touch threshold could be set to detect this kind of a scenario, while the proximity threshold is used to detect distance.

### Miscellaneous Sensor Parameters

Let's briefly re-visit a few generic sensor parameters that we have not yet addressed. The first is the sensor timeout threshold.

**Sensor Timeout** The sensor timeout threshold provides a mechanism for averting a "stuck key" scenario. Imagine a touch panel on thermostat on a wall. Let's say someone put a broom stick up against thermostat, and that showed up as a proximity detection. A proximity detection has the effect of halting our long-term-average. This means that the LTA is no longer going to track environmental drift. This affects the usability of the touch panel. The timeout feature allows for this to be averted by specifying, in samples, how long a sensor can be in detect (proximity or touch) before the interaction is declared invalid.

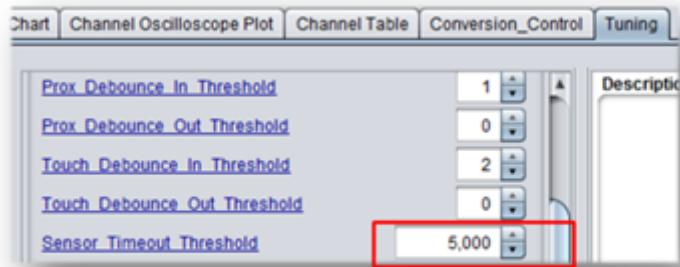


Figure 11.46: Selecting Slider Timeout Value

Typically when users are using sensors, they touch them briefly- not for 30 seconds or more. By setting a lower sensor timeout threshold, we can have the system "reset" if for some reason it is stuck in detect for a long time. We can simulate this tool by setting the timeout extremely short- say, 200 samples. After 200 samples in detect (< 10sec), the system will reset the touch flags and reseed the long term average. Feel free to try this out.

**Sensor De-Bounce** The touch library offers a built-in de-bounce mechanism for into and out of proximity and touch states. This is adjustable at a sensor level.



Figure 11.47: Selecting Slider Debounce Values

Default settings for De-Bounce is 0 (no de-bounce applied). By applying de-bounce to the system, you offer a way to ensure that there was actually a valid touch (and not a brush of a hand or a noise event). However, de-bounce introduces latency into the system.

Try out setting longer and longer de-bounce settings and observing the effects on the sensor performance.

### Scan Time Estimation

Based on the sensors we have configured for this project, a time profile or scan time estimation can be viewed showing the overall scan period, the time to scan each sensor and the combined time for all of the sensors. Note that the time to process the data and transmit the data to the Design Center is not accounted here, only the sensor measurement scan times. To view the scan time estimation, open the Controller Customizer and select the Scan Time Estimate tab.

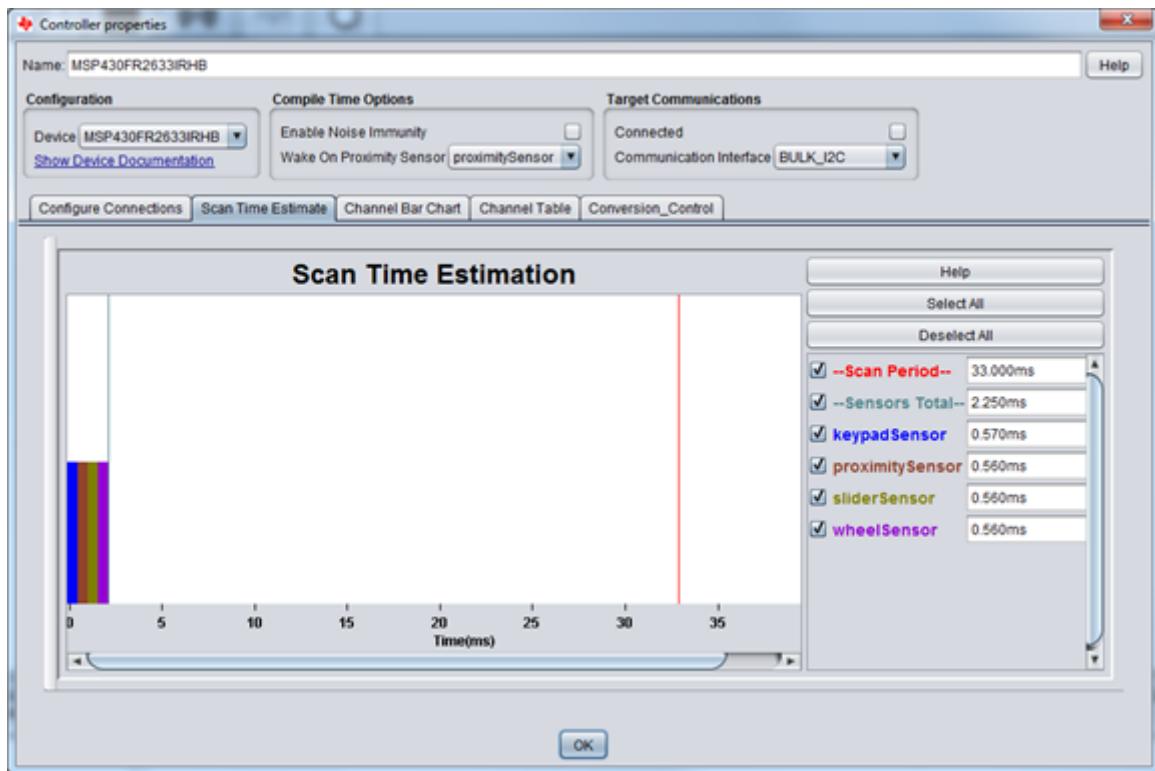


Figure 11.48: Scan Time Estimation View

#### 11.3.4 Part 3 - Generate Updated Sensor Configuration

Now that we've gone through and tuned our sensors, we want to apply those new tuning values to the source code. We don't need to re-generate a whole new project to do that- we can simply update the CAPT\_UserConfig.c and CAPT\_UserConfig.h files, thanks to the Update Code feature in the Design Center. First, let's save of an updated copy of the Design Center model (\*.ser), so that we have a known good copy of our tuning in Design Center format. Then, open the Controller customizer, and press the Generate Source Code button again. The configuration we want is "Update existing project."

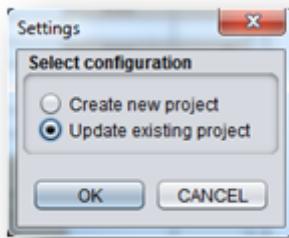


Figure 11.49: Select Update

This time, we want to select the root directory of the CCS project we created, which is in our CCS Workspace, not our working folder where we first generated the starter project.

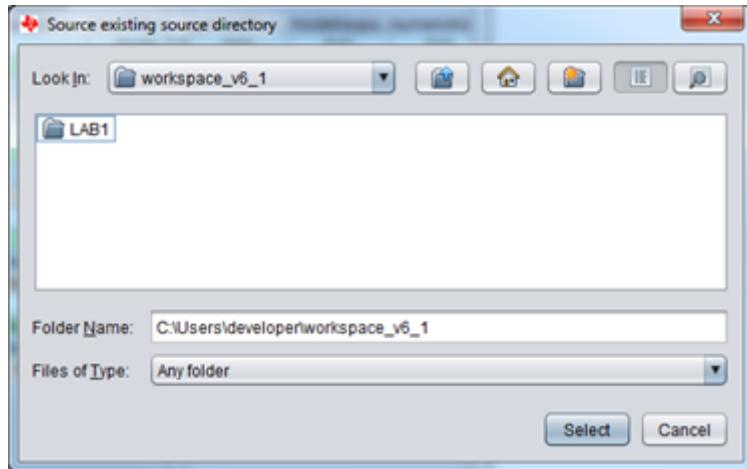


Figure 11.50: CCS Project Directory

You can now re-build the CCS project and re-run it, and all of the new tuning values will have taken effect.

**Congratulations!** You have successfully completed your first CapTivate™ sensor design. Next, let's learn how easy the low-power "wake on proximity" feature is to use and measure the current for our sensor design.

## 11.4 Experiments with Low Power

### 11.4.1 LAB #2

This is Lab #2 and we will experiment with the "Wake on Proximity" feature which can achieve extremely low-power by using the CapTivate™ hardware state machine to scan the proximity sensor while the CPU remains in low power mode 3. We will investigate how the sensor tuning and scan rates can directly impact the low power operation. We will be using the Energy Trace™ feature to measure the power during this lab.

#### Pre-requisite

We will continue to build on the work that was done in LAB1, so if you skipped that step, go back to [Creating a new sensor design project](#).

#### Goals

- Learn how to use EnergyTrace™ to make power measurements
- See how tuning sensors effects low-power

#### Hardware Setup

Continue to use the same hardware setup that was used in the previous lab, LAB1. On the MCU PCB, move Jumper J3 between MCU\_VCC and 3.3V METERED.



Figure 11.51: Jumper to 3.3V Metered

## Design Center Project

Before starting with this lab, we want to save our current CapTlivate™ Design Center project as LAB2. Select "Project Save As" and re-name the folder as "LAB2".

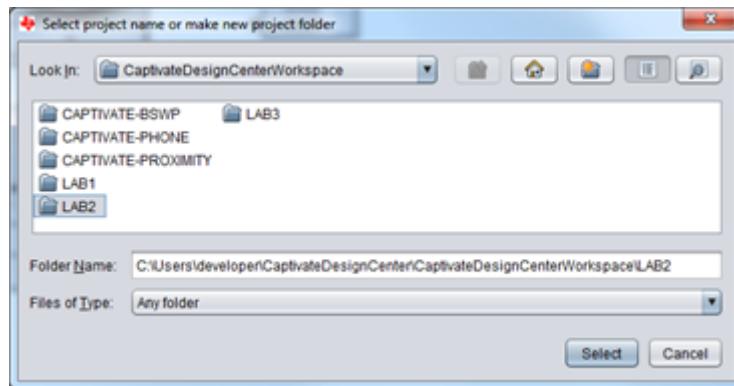


Figure 11.52: Save as LAB2

## Adding wake on touch functionality

The CapTlivate Design Center can enable certain compile time software features to be included in the project's starter code. One of these features is the "wake on proximity", also known as "wake on touch", which enables the CPU to remain in a low power mode while the CapTlivate™ hardware state machine scans for a proximity condition. To understand how this behavior works, click [here](#)

**Select the Wake on Proximity sensor.** The controller customizer Compile Time Options box lets you select a "Wake on Proximity Sensor" from a drop-down box populated with the sensors in the project. This selection chooses which sensor will be scanned during wake on proximity operation. For this lab select the *prox* sensor.

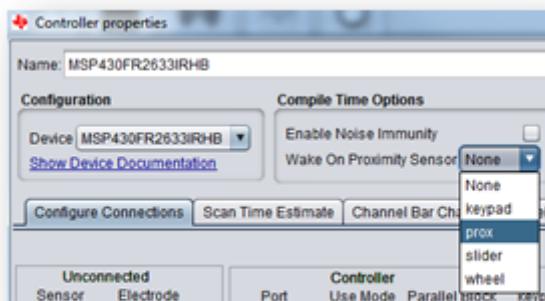


Figure 11.53: Selecting Wake on Proximity Sensor

## Modify wake on proximity parameters

In the controller's "Conversion Control" tab, several parameters define the CPU and wake on proximity operations.

The "Wakeup Interval" specifies how often the CPU should wake automatically, regardless of proximity or touch event. This is a built-in safety mechanism for the wake on proximity state machine. For this lab, disable the Wakeup Interval. This will keep the wake on proximity state machine active indefinitely, providing ample time to take our low power measurements.

## Generate LAB# 2 Starter Project Code

Generate starter code project. In the controller customizer, under the Configure Connections tab, select the Generate Source Code button. Select Create new project and use the default output directory.

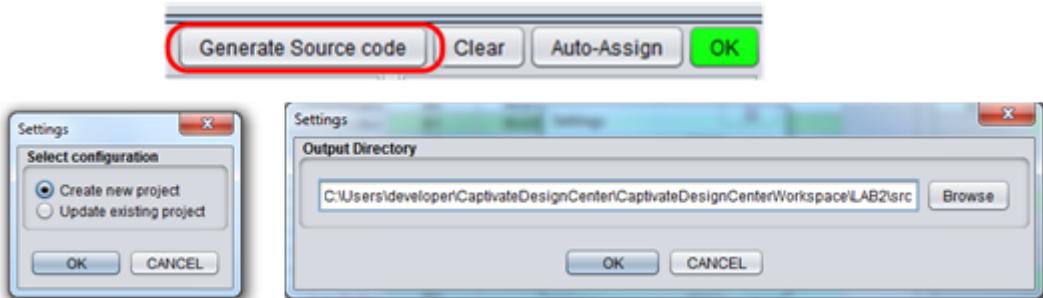


Figure 11.54: Create/Save starter code project

## Launching target MCU

Import starter code project to CCS. Now that we've generated our LAB2 source code, we need to import the new project to our CCS workspace. If you need a CCS "refresher", follow the steps in a later section CCS - Building and programming.

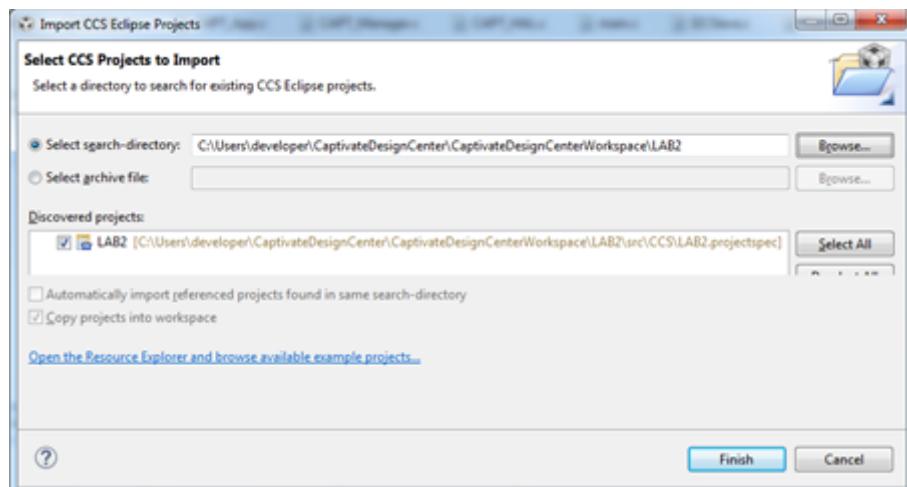


Figure 11.55: Import Project

After importing the project, click the debugger icon to build and download the code to the target. Since we won't be using the debugger at this time, set the target running in "free run" mode, Run>Free Run or terminate the debug session. Disabling the debugger disables the EEM module in the MCU, further reducing the current drawn by the MCU.

## Using EnergyTrace™

EnergyTrace™ Technology is built into the eZFET programmer on the CAPTIVATE-PGMR PCB. It provides a DC/DC regulated output and monitors the amount of energy a load (target MCU) consumes. We will use the EnergyTrace™ to measure the power and currents in the target MSP430FR2633.

---

**IMPORTANT: Although EnergyTrace™ Technology can measure very small currents with reasonable accuracy and provide a simple tool for low-power demonstrations, it is HIGHLY RECOMMENDED that low-power measurements should be made using a bench supply, high accuracy meter and the target isolated from the eZFET programmer.**

To enable energy trace, click the icon shown in the diagram below. If this icon does not appear on your menu bar, refer to the troubleshooting section at the end of this manual.

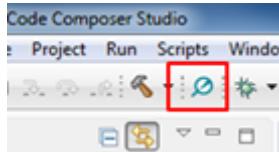


Figure 11.56: Enable EnergyTrace™

A new view tab should appear in your CCS that looks something like the following:

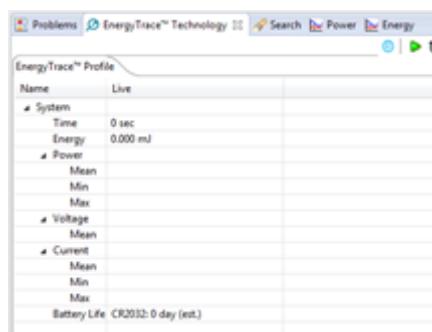


Figure 11.57: EnergyTrace™ Tab

Set the EnergyTrace™ capture length for 30 seconds. This should provide adequate time to capture the traces for this lab.

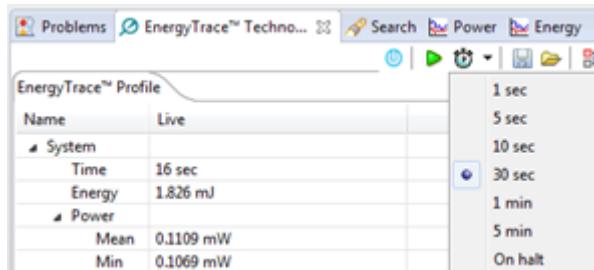


Figure 11.58: EnergyTrace™ Capture

The target MCU is running in the "free run" mode without the debugger. To reset the target MCU, press the RESET button on the MCU PCB, then wait for LED2 to blink once. Start an EnergyTrace™ capture by clicking the green "play" arrow when ready.

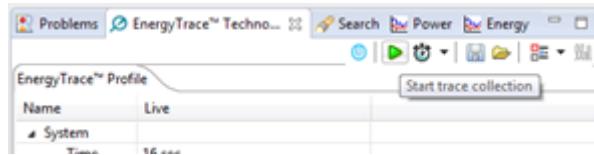


Figure 11.59: Start EnergyTrace\texttrademark {}

#### 11.4.2 Part 1 - Measure Proximity Sensor Low Power current

First, let's measure the wake on proximity average current to establish a baseline. Be sure to keep your hands clear of the PCB while performing this measurement. The EnergyTrace™ "mean" current measurement should < 10uA. This is the current required to measure the proximity sensor @ 10 times per second using the default conversion count of 800. If your measurement is much greater than 10uA, hit the RESET button, wait for LED2 to blink once then repeat the measurement. If the measurement is 0, make sure you have jumper J3 between MCU\_VCC and 3.3V METERED on the MCU PCB. Record your measurement.

#### 11.4.3 Part 2 - Effects of Sensor Sensitivity on Low Power current

Next we want to measure the effects a sensor's tuning can have on the device's current consumption. The BSWP demo board has a single element proximity sensor that runs the perimeter of the PCB. This proximity sensor is used by the "wake on proximity" state machine to detect when a hand is near the PCB. The proximity sensor is tuned just like a button, but with greater sensitivity. In LAB1, the proximity conversion count was set to 800, providing sufficient resolution and proximity detection out to a couple of centimeters. Let's see if we can increase the distance by increasing the resolution of the sensor.

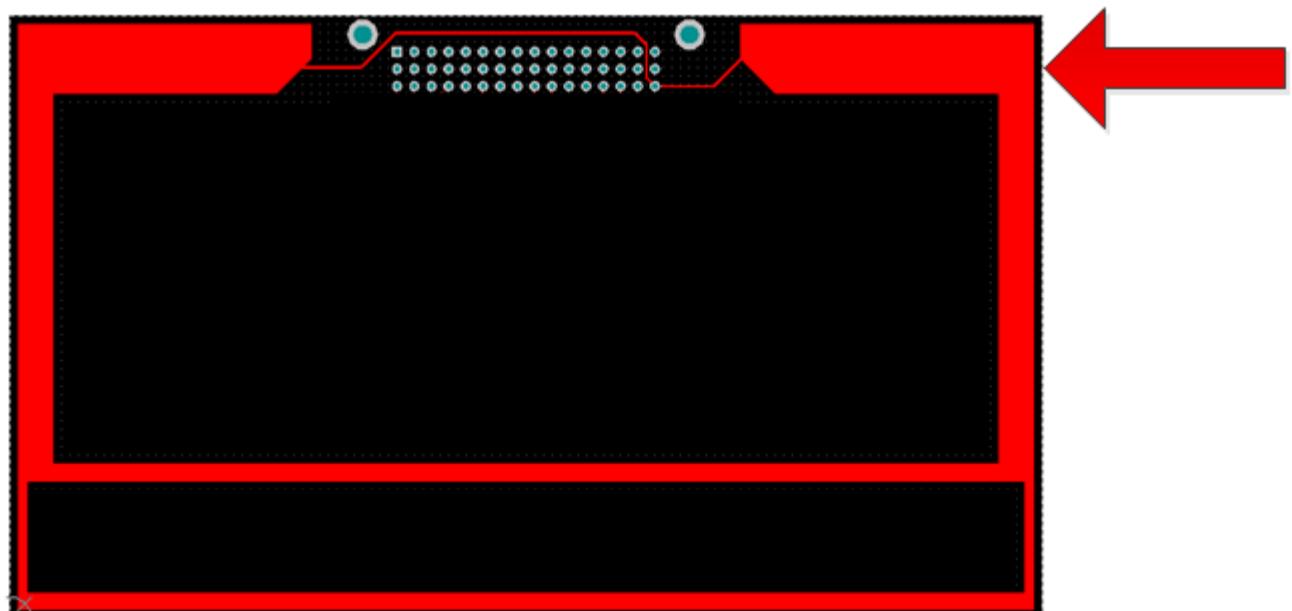


Figure 11.60: BSWP Proximity Traces

Let's start by increasing the conversion count to 1600. In the Design Center, double-click the proximity sensor to open its customizer, then click the Conversion Control tab. Update the conversion count to 1600, then click the *Apply* button. Click on the Channel Bar Chart to view the new conversion count data and confirm the LTA is near 1600. Verify as your hand approaches the PCB, the proximity detection now occurs sooner (further from the PCB). The proximity sensor is now more sensitive and able to detect proximity from a greater distance. If you are having trouble seeing the values at 1600, it may be necessary to adjust the plot range in order to see the channel bar data. Adjust the plot parameters to fit the new size of the channel bar using the Plot Display Options.

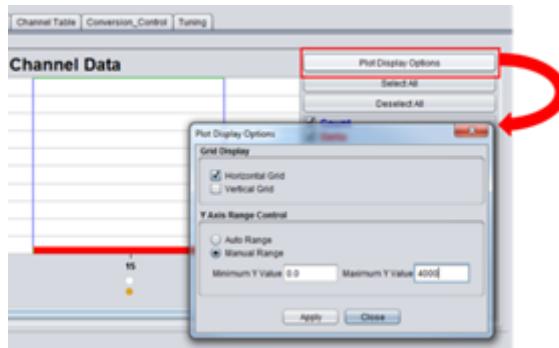


Figure 11.61: Plot options

You may also notice with this higher resolution it may be more difficult to see when the blue bar crosses the proximity thresholds. The design center has a "zoom" feature allowing an expanded view of the selected area.

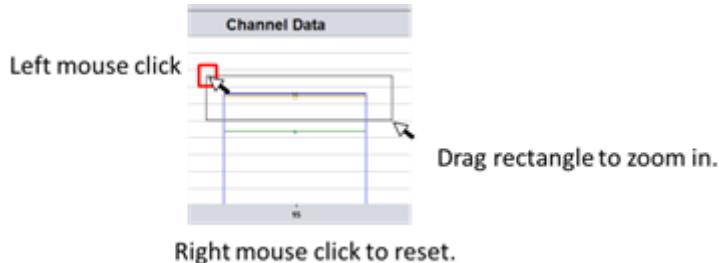


Figure 11.62: Zoom in

Now that we have increased the sensitivity, let's change the proximity threshold to prevent false detections. Select the Proximity sensor's Tuning tab and modify the sensor's Proximity Threshold and Negative Touch threshold to 30. Click on the Negative Touch parameter for additional information about this parameter in the Design Center description frame.

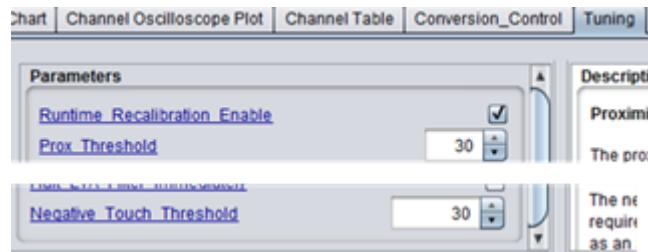


Figure 11.63: Proximity Thresholds

With the increase in sensitivity, we have effectively increased our proximity range. Let's see how this impacts our low-power current. **DO NOT** reset the MCU, as this will cause the conversion count reset to 800 and thresholds to reset to 15. Start EnergyTrace™ and examine the "mean" current. It should be about double from the previous baseline measurement. Record your measurement.

#### 11.4.4 Part 3 - Effects of Sensor Scan Rate on Low Power current

The device's low power current can also be affected by the rate at which a sensor is scanned. Let's reset everything back to default by pressing the MCU reset button. To confirm the parameters were reset to their original programmed values, click the proximity sensor Conversion control tab, then the READ button. This reads the current values in the target MCU and you should see the conversion count = 800. In the Design Center, double-click the controller to open its customizer and select the Conversion Control tab. Modify the Wake On

---

Proximity Mode Scan Rate to 50ms then click Apply. This effectively doubles the rate we scan the proximity sensor.

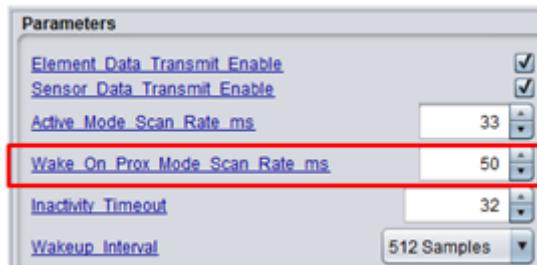


Figure 11.64: Proximity Scan Rate

**DO NOT** reset the MCU, as this will reset the scan rate to 100. Start EnergyTrace™ and examine the "mean" current. Again, you should see the current more than double from the baseline measurement.

## Summary

The demand for higher sensitivity, higher resolution sensing has a direct impact on the low-power current, as does the rate at which a sensor is scanned. So always consider the trade-offs when designing your sensors for the end application.

Congratulations! You have successfully experimented with the low-power "wake on proximity" feature and learned how sensor tuning and scan rate can directly impact your sensor design. Next, let's learn about the CapTlivate™ capacitive touch software library. Continue with this workshop in the next section.

## 11.5 Exploring the CapTlivate™ Touch Library

### 11.5.1 LAB #3

#### Pre-requisite

We will continue to build on the work that was done in LAB2, so if you skipped that step, go back to [Experiments with Low Power](#).

#### Goals

- Learn how the sensor manager is used to simplify sensor measurements
- Learn how to use callbacks to handle sensor events
- Learn how to acquire proximity, touch and position status as part of callback

#### Hardware Setup

Continue to use the same hardware setup that was used in the previous lab, LAB2. On the MCU PCB, move Jumper J3 between MCU\_VCC and 3.3V LDO and check the jumpers are in place for the two LEDs.

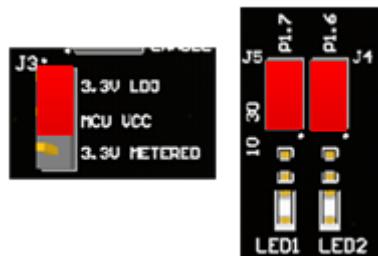


Figure 11.65: Jumpers

### Design Center Project

Before starting with this lab, we want to save our current CapTivate™ Design Center project as LAB3. Select "Project Save As" and re-name the folder as "LAB3".

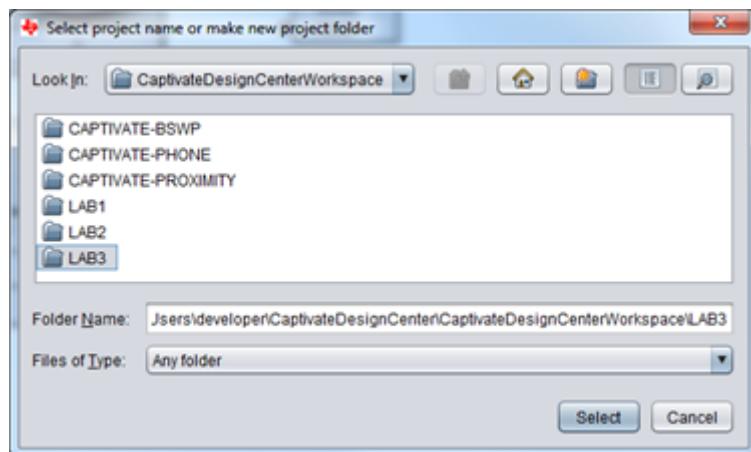


Figure 11.66: Save as LAB2

### Generate LAB# 3 Starter Code Project

Generate starter code project. In the controller customizer, under the Configure Connections tab, select the Generate Source Code button. Select Create new project and use the default output directory.

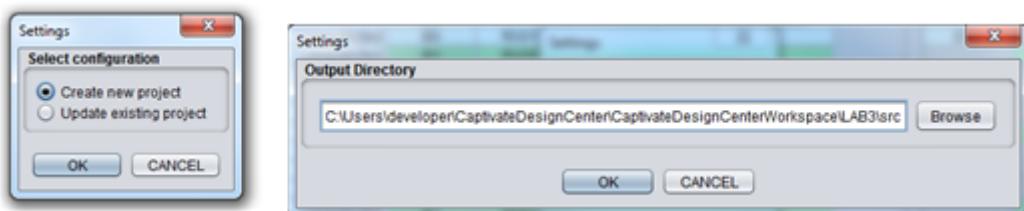


Figure 11.67: Save as LAB3

Select the Design Center communication interface. Double click on the controller to bring up the customizer view. In the Target Communications Select the Connected communications checkbox and verify the Target Communications interface is set to BULK\_I2C.

Import the starter code project into CCS as we did in previous labs. Build the LAB3 starter code project and program your target MCU.

### 11.5.2 Part 1 - CapTlivate™ Touch Library Overview

For this portion of the lab you are encouraged to follow along as we explore the CapTlivate™ capacitive touch library.

The software library is a comprehensive collection of functions or components for touch, communications and sensor management as represented by the software stack shown below. The primary "Touch" layers are Advanced, Touch (or "Base") and Communications. Functions range from advanced sensor processing to "bare-metal" functions allowing direct access to the CapTlivate™ peripheral.

Communication functions for I2C and UART serial drivers and a communications protocol enable the MSP430FR26xx\25xx microcontroller to send sensor data to the CapTlivate™ Design Center during the sensor development process.

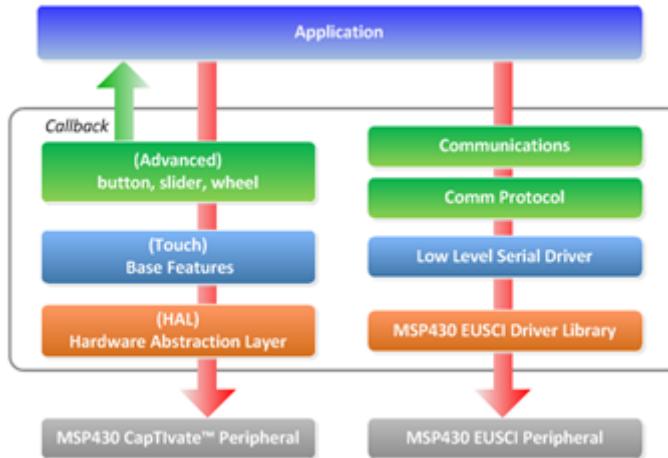


Figure 11.68: Slider callback extra credit

The majority of library components are provided pre-compiled in the MSP430FR2633 ROM. They are also provided as captivate .lib or captivate.r43 file format for CCS and IAR respectively. No source code is provided for these components. These components are accessed using the [CapTlivate™ API](#)

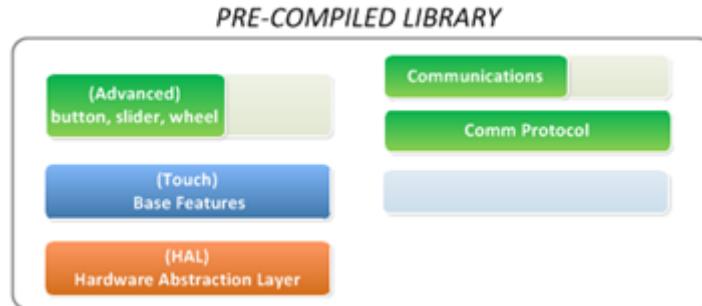


Figure 11.69: Slider callback extra credit

Other components are provided as source code that can be modified to suit the needs of the application. The library manager is one example. The library manager "CAPT\_Manager.c" provides three high-level functions to manage the primary sensor activities: CAPT\_initUI(), CAPT\_updateUI() and CAPT\_calibrateUI(). This greatly simplifies the application's interface with the library and can be used a framework for your application. The library manager is provided as source.

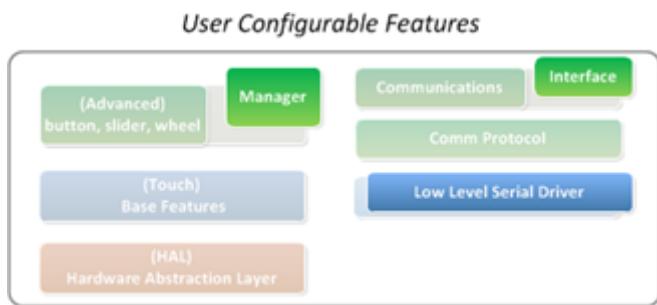


Figure 11.70: Slider callback extra credit

### 11.5.3 Part 2 - Explore CCS project structure and details

During this part of the lab you are encouraged to explore the CCS project and library features on your own. Things to look at are:

- CapTlivate™ Library are components located "captivate" directory
  - API header files for library
  - ROM header files
  - Pre-compiled library
  - Application level sensor manager
- CapTlivate™ configuration files are located "captivate\_config"directory
  - Configuration files generated by the Design Center
  - **DO NOT hand edit these files** as they get updated by the Design Center
- CapTlivate™ application framework files are located "captivate\_app"directory
  - Provides application level framework example for using the library
  - Board support files

### 11.5.4 Part 3 - Creating and using callbacks

The CapTlivate™ touch library uses callbacks to communicate with the user's application. Callbacks provide a mechanism for the application to be notified when a sensor has been updated. The application must first register its "callback" function for each sensor before it can receive updates. Once an application's callback function is registered, the callback is executed each time the corresponding sensor is scanned and processed, regardless if a proximity or touch detection has occurred. During the callback, the application can query the sensor's data structure to determine the status of the sensor. The illustration below shows an example sequence where the keypad is scanned and processed, then the keypad callback is executed, followed by sensor data and status communicated to the CapTlivate™ Design Center. This sequence continues for the other sensors then repeats based on the programmed scan rate.

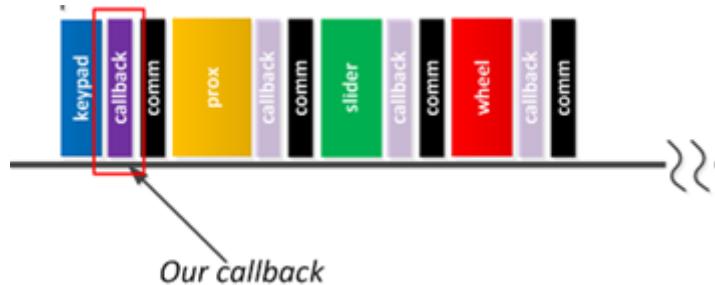


Figure 11.71: Callback after sensor processing

For this lab we are going to create a callback function that will turn on LED1 when any keypad button is pressed. The library function *CAPT\_registerCallback()* provides the callback registration.

Registering the application's callback function named "my\_keypad\_callback" to a keypad sensor uses the following syntax:

```
MAP_CAPT_registerCallback(&keypad, &my_keypad_callback);
```

#### LAB3 Action:

In our LAB3 code project, insert the function call described above just before the function call *CAPT\_appStart()*, around line 80, in main. This will allow our callback to be registered before the library starts.

Note: The names of the sensors created by the design center for our project are located in the *CAPT\_UserConfig.h* file and shown here for convenience. You can see the first sensor in this list is our keypad sensor.

```

101 //
102 // Captivate Sensor Prototypes
103 // These prototypes enable access to sensors
104 // from the application.
105 //
106 extern tSensor keypad;
107 extern tSensor prox;
108 extern tSensor slider;
109 extern tSensor wheel;
110

```

Next, inside our callback, the application can perform any processing on the sensor data and status, such as checking for proximity, touch detection or slider/wheel position. Here is our example that checks when any button in the keypad is touched and released, toggling an LED accordingly.

#### LAB3 Action:

Add the following callback code snippet somewhere before main():

```

void my_keypad_callback(tSensor* pSensor)
{
    if ((pSensor->bSensorTouch == true) && (pSensor->bSensorPrevTouch == false))
    {
        LED2_ON;
    }
    else if ((pSensor->bSensorTouch == false) && (pSensor->bSensorPrevTouch == true))
    {
        LED2_OFF;
    }
}

```

Then, in our callback we will turn on LED2 only when a "touch" is first detected, and turn off LED2 when a release is detected. This minimizes the time the CPU spends in the callback. We will need to make one additional change

---

to the original code to get this to work. We need to comment out in main() the lines of code shown below. This will allow our callback to control LED2.

**LAB3 Action:**

Modify the section of main() as shown below:

```
/*
LED1_ON;
if(CAPT_appHandler()==true)
    LED2_ON;
else
    LED2_OFF;
LED1_OFF;
*/
CAPT_appHandler();
```

Click the debugger icon to build and download the code to the target. Click the "play" icon to start the demo running. You should now see LED2 turn on when just the keypad buttons are touched.

#### 11.5.5 Part 4 - Determining sensor state and position

Sliders and wheels are multi-element sensors, typically with 3 or more electrodes. Like buttons, sliders and wheels can have proximity and touch states, but they also have a position. Position is determined from the individual element measurements. The CapTivate™ software library's algorithm determines the sensors position (place where finger is touching) based on the measurements from all the sensor's elements and provides the result in the sensor's structure.

In this section we are going to create a callback function for the slider that will turn on LED1 when the slider's position is greater than 500 and turn off LED when the position is less than or equal to 500. We chose 500 because from LAB1 the slider's resolution was set to 1000, which provides points from 0 to 999.

Registering the application's callback function named "my\_slider\_callback" to a slider sensor uses the following syntax:

```
MAP_CAPT_registerCallback(&slider, &my_slider_callback);
```

**LAB3 Action:**

In the LAB3 code project, insert this function call described above just before the function call CAPT\_appStart() in main().

Below is an example callback that reads the sensor's data structure and turns LED1 on or off depending on the value of the slider position.

**LAB3 Action:**

Add the following callback code snippet before the function main():

```
void my_slider_callback(tSensor* pSensor)
{
    tSliderSensorParams *pSliderParams;

    if (pSensor->bSensorTouch == true)
    {
        pSliderParams = (tSliderSensorParams*)(pSensor->pSensorParams);
        if (pSliderParams->SliderPosition.ui16Natural > 500)
        {
            LED1_ON;
        }
        else
        {
            LED1_OFF;
        }
    }
}
```

Click the debugger icon to build and download the code to the target. Click the "play" icon to start the demo running.

While viewing the slider in the design center, LED1 will turn on when your finger is greater than 500 and turn off below 500. What happens when you release your finger from the slider when greater than 500? LED1 remains on. That's because our code only changes the state of the led when the slider is being pressed.

What modifications to this callback will turn off LED1 if the slider is not being touched?

Here is one answer below. Try it to see if it works:

```
void my_slider_callback(tSensor* pSensor)
{
    tSliderSensorParams *pSliderParams;

    if (pSensor->bSensorTouch == true)
    {
        pSliderParams = (tSliderSensorParams*)(pSensor->pSensorParams);
        if (pSliderParams->SliderPosition.ui16Natural > 500)
        {
            LED1_ON;
        }
        else
        {
            LED1_OFF;
        }
    }
    else
    {
        LED1_OFF;
    }
}
```

Congratulations! This is the end of the CapTivate™ workshop. Visit the [Introduction](#) section to learn more about CapTivate™

# Chapter 12

## FAQ

### 12.1 Design Kit

#### 12.1.1 Does the kit arrive pre-programmed?

- Yes, the target is programmed to work with the CAPTIVATE\_BSWP panel and the design center loaded with the CAPTIVATE-BSWP project which is the [out-of-box experience](#).

#### 12.1.2 How do I verify that the EVM is working?

- Verify the green LED (Power Good) is on and the green LED (USB Enumeration) is blinking.
- Verify the red LED (LED4) on the programmer is blinking (indicates sensor data is being sent to the design center)
  - Note: After power is first applied, it may take a couple of seconds before the red LED begins to blink. During this time, the CapTlivate™ Software Library automatic sensor calibration is being performed.
  - If red LED is not blinking:
    - \* Verify all communications jumpers P1 are in place on the CAPTIVATE-FR2633 MCU PCB
    - \* Verify the MSP430FR2633 has been programmed with the correct demonstration firmware

#### 12.1.3 How can I increase the sensitivity or range of the proximity sensor in the out-of-box experience demo?

Link to [out-of-box experience](#)

- The default Conversion Control settings of the proximity sensor are [Conversion Count](#) = 800, [Conversion Gain](#) = 100, this the minimum possible value for the Conversion Gain, and the frequency divider is f/4. There a lot of different means available to increase the sensitivity of the proximity sensor, and the following descriptions are just a few examples.
- Increase the Conversion Gain settings of the proximity sensor.
  - First, in order to gain some more insight, the [Features](#) should be set to advanced.
  - One method of increasing the sensitivity is to apply more Parasitic offset. The parasitic offset level, shown in the [Conversion\\_Control](#) tab, has a maximum setting of 255. By increasing the Conversion Count value you will see the sensitivity and the parasitic offset level increase. Verify that Conversion Count value has not been increased beyond a value that results in the maximum parasitic offset level and that the noise is acceptable.
- Increase the Conversion Gain settings of the proximity sensor and then the Conversion Count.

- 
- As in the previous example, in order to gain some more insight, the [Features](#) should be set to advanced.
  - Another method of increasing the sensitivity is to apply more Parasitic offset in the context of more overall gain.
    - \* Increase the Conversion Gain until the coarse gain ratio is 5 or 6.
    - \* Increase the Conversion Count until the desired sensitivity or range is achieved. Verify that this uses both a lower parasitic offset level and that the noise is acceptable.

## 12.2 CapTlivate™ Technology Design Center

### 12.2.1 Can I download code from the Design Center?

- No, the Design Center does not download firmware to the target. The Design Center will generate source code, and projects, that can be imported into CCS or IAR. Once this code is loaded onto the target communications are available to adjust various parameters and monitor performance.

### 12.2.2 What does the 'No connected HID devices' pop-up window mean?

- If you are also running the target application in an IDE, ensure the IDE debugger is not paused
- Attempt to reset the HID connection.
  - Set the target connection status in the Design Center to "disconnected"
  - Unplug the USB cable from PC and reconnect.
  - Set the target connection status in the Design Center to "connected"

### 12.2.3 Why is the target data display slow?

- Attempting to display too many graphs in the GUI can cause the display refresh time to increase. Try closing some of the display windows.

### 12.2.4 When do I need to generate a project from the design center and load onto the target?

- You can generate a project for either CCS or IAR at any time, however, it is recommended to generate a project once after the sensors and the MCU have been placed as described in [Generate source code](#).

### 12.2.5 What does updating a project from the design center do?

- Updating a project from the design center updates the CAPT\_UserConfig.c and CAPT\_UserConfig.h files only.

### 12.2.6 Where can I find the example demo project source code files?

- The CapTlivate™ demo project source file locations are described [here](#).

### 12.2.7 Why is the CapTlivate™ Design Center window not visible when I launch the GUI?

- When running on Windows, it is possible that Windows will attempt to draw the CapTlivate™ Design Center display on a monitor/display that is no longer connected. This most commonly occurs in multi-monitor configurations or after a computer has been connected to a TV or projector. If the CapTlivate Design Center

---

appears in the Windows task tray, but does not appear on the screen when selected, try the following steps: Select the CapTlivate™ Design Center in the Windows task bar. Then, press and hold the Windows key followed by the left or right arrow key to snap the window onto the connected display. The CapTlivate™ Design Center window should appear.

## 12.3 CapTlivate™ Technology

### 12.3.1 Why does sensitivity decrease as I increase the Conversion Gain?

- Increasing the [Conversion Gain](#) without increasing the [Conversion Count](#) will effectively reduce the amount of parasitic offset compensation being applied. In other words, the change in capacitance will be relative to a larger portion of the base parasitic capacitance.

### 12.3.2 Why is the scan rate slower than I specify?

- There are many possible causes related to a slow scan rate, many of which are not related to the CapTlivate technology. One possible related issue is the selection of Conversion Count value that is not realizable and the Runtime Recalibration Enable is selected. This is easily seen in any of the data analysis tabs when the actual conversion result is less than 7/8th of the Conversion Count. In this state the library will automatically rerun the calibration routine after each measurement in an attempt to resolve the difference between the conversion result and the Conversion Count to 0. The increase in scan time is actually the addition of the recalibration with each measurement. Adjust the conversion time to a lower achievable level.

# Chapter 13

## Glossary

Brief descriptions of key terms.

### 13.1 Automatic Power Down

#### Automatic Power Down

The automatic power down feature, when enabled, prevents the CapTlve peripheral from shutting down into its low power state after the completion of a conversion. Normally (when disabled), The power-down occurs immediately following the completion of each time cycle's measurement. The next time a measurement is started, the peripheral will power back up. There will be a delay associated with the power-up.

#### Affected Software Parameters

The Frequency\_Divider parameter corresponds to the **bLpmControl** member of the **tSensor** type in the CapTlve Touch Library.

### 13.2 Wake On Proximity Mode

#### Wake On Proximity Mode

Wake-on-proximity mode is an operating mode that allows the first cycle of any sensor to be used as a wake-up mechanism for the rest of the user interface. To enable this mode, select a sensor to be used as the wakeup sensor via the drop down selection in the controller customizer. By enabling this mode, it is possible to achieve lower power consumption for a system by only updating one time cycle at a slower rate (say, a proximity sensor) until a proximity event on that cycle is detected. At that point, the full user interface will be updated. The sensor selected for wake-on-proximity will be updated in the background with no CPU intervention until one of the following 3 things happens:

1. **A proximity threshold crossing was detected on the wake-on-proximity sensor.** If the peripheral detects a proximity threshold crossing, a detection interrupt will be issued and the application will switch from autonomous mode to active mode.
2. **A negative touch threshold crossing was detected on the wake-on-proximity sensor.** If the peripheral detects a negative touch threshold crossing, a detection interrupt will be issued and the application will switch from wake-on-proximity mode to active mode.
3. **A conversion counter interrupt condition occurred.** The conversion counter interrupt provides a simple way to wake the application after a specified number of conversions has taken place (specified by the Wakeup\_Interval parameter). In systems with other sensors, it is advised to wake up the application periodically to update the long term averages (LTA's) of the other sensors in the system, so that environmental drift is accounted for.

#### Implementation Notes

---

The Wake\_On\_Prox\_Mode\_Scan\_Rate\_ms parameter specifies the interval at which the wake-on-proximity sensor is updated in hardware. To convert to samples per second (SPS), simply take 1000 divided by the specified report rate period. For example, a report rate (response time) of 50ms would equate to 20 samples per second (SPS).

The Inactivity\_Timeout parameter specifies the amount of time (in samples) to wait before switching from active mode to wake-on-proximity mode after all sensor proximity flags have cleared.

#### Affected Software Parameters

The Wake\_On\_Prox\_Mode\_Scan\_Rate\_ms parameter corresponds to the *ui16WakeOnProxModeScanPeriod* member of the *tCaptivateApplication* type in the CapTlve Touch Library.

The Inactivity\_Timeout parameter corresponds to the *ui16InactivityTimeout* member of the *tCaptivateApplication* type in the CapTlve Touch Library.

The Wakeup\_Interval parameter corresponds to the *ui8WakeupInterval* member of the *tCaptivateApplication* type in the CapTlve Touch Library.

## 13.3 Bias Current

### Bias Current

Bias current is only applicable to mutual capacitance sensors. When measuring mutual capacitance sensors, a sample and hold amplifier drives the receive electrode to remove any contributions of the parasitic capacitance to ground. This ensures that only the charge associated with the mutual capacitance is accumulated on the sample capacitor during the measurement. The bias current parameter provides the ability to optimize a design for power consumption if the system allows for it.

If power consumption is not a concern, the bias current may be defaulted to the highest value and left alone.

If power must be optimized, it is possible to begin reducing the bias current one level at a time while observing the impact on the measurement when touched. If there is too much parasitic capacitance to ground, the achievable deltas may be reduced and the measurement may dip below the LTA due to a touch, rather than rise above it. In this case, the parasitic capacitance is overwhelming the drive capability of the sample and hold amplifier and the bias current needs to be increased.

### Implementation Notes

For some systems with little parasitic capacitance to ground, varying the bias current may not have a large effect on performance. When adjusting the bias current, be sure to test all touch use-cases to verify functionality.

#### Affected Software Parameters

The Bias\_Current parameter corresponds to the *ui8BiasControl* member of the *tSensor* type in the CapTlve Touch Library.

## 13.4 Button Group Sensor

A button group sensor is a sensor with two or more buttons, such as a keypad.

## 13.5 Target Communications Configuration

### Target Communications Configuration

The CapTlve touch library communications module has the capability of sending element data and sensor data to the host PC. Element data consists of all the low level data for a sensor's elements: current measurement count, long term average, and touch and proximity status. Sensor data consists of aggregate data for the sensor. This includes items such as slider position, wheel position, and global sensor status flags.

---

Both data types can be sent, as well as one or the other or none. On larger panels, it may be desirable to disable transmission of element data to increase the maximum report rate.

#### Implementation Notes

The protocol used to communicate between the target Captivate MCU and the PC running the CapTlve Design Center is re-usable for communicating between the target and another embedded host processor or host MCU.

#### Affected Software Parameters

The Element\_Data\_Transmit\_Enable parameter corresponds to the **bElementDataTxEnable** member of the **tGlobalCaptivateSettings** type in the CapTlve Touch Library.

The Sensor\_Data\_Transmit\_Enable parameter corresponds to the **bSensorDataTxEnable** member of the **tGlobalCaptivateSettings** type in the CapTlve Touch Library.

## 13.6 Channel

A channel refers to the physical CapTlve™ I/O pin assigned as a RX electrode in self-capacitive mode and RX/TX pair in mutual capacitive mode.

## 13.7 Conversion Count

#### Conversion Gain and Conversion Count

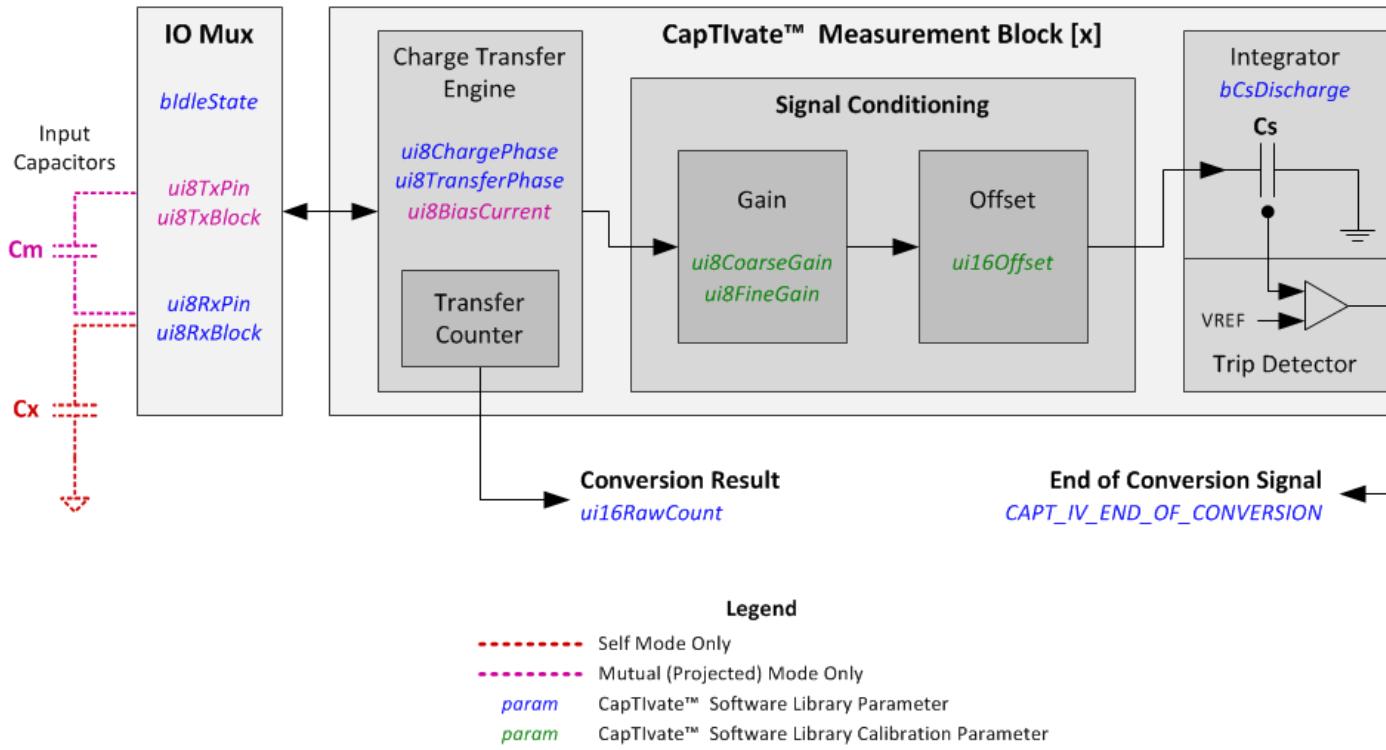
Conversion Gain and Conversion Count are the fundamental parameters used to establish the performance of the sensor. These parameters determine the resolution, sensitivity, and required conversion time. They are the inputs to the calibration algorithm, which identifies the correct coarse/fine gain ratios, offset scale, and offset level for each element at runtime.

#### Quick Reference

1. To increase resolution, increase the Conversion Count.
2. To increase sensitivity, increase the Conversion Count OR decrease the Conversion Gain.
3. The Conversion Count value establishes the conversion time, as it specifies the approximate number of charge transfers per conversion.
4. The Conversion Count value must be equal to or larger than the Conversion Gain. When the two values are equal, the minimum amount of offset subtraction is applied.

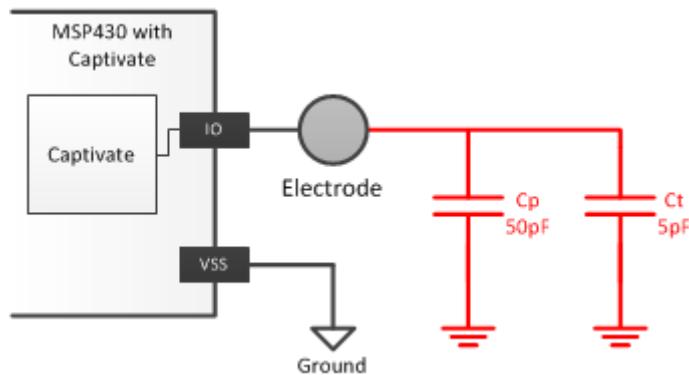
#### Background

The CapTlve peripheral has the ability to apply gain as well as offset to the capacitance being measured. When a conversion is started, the unknown external capacitance being measured is charged to a known voltage. The charge on that external capacitor is then transferred into a sample capacitor which is on-chip. The conversion result is simply the number of charge transfers from the external capacitor to the internal sample capacitor that are required to "fill up" the internal sample capacitor. The number of transfers needed to do so is representative of the capacitance of the external electrode being measured. The signal chain is shown below.



The gain stage of the CapTlivate peripheral provides the ability to scale the effective size of the unknown external capacitor relative to the internal sample capacitor. This serves two main purposes. First, it allows the peripheral to handle a wide range of capacitances. Second, it allows for the designer of the system to dial in a desired measurement resolution.

The offset stage of the CapTlivate peripheral provides a mechanism to remove a set amount of charge during each charge transfer. This charge is typically associated with the parasitic capacitance of the sensor. Parasitic capacitance can be thought of as an unwanted DC offset in the measurement. For example, if an electrode has an inherent parasitic capacitance of 50pF, and a touch on that electrode adds 5pF, the capacitance goes up by 10%. This model is shown below.



However, if charge associated with the parasitic capacitance was subtracted off such that the inherent parasitic capacitance of the electrode appeared as just 10pF, then a 5pF touch would increase the capacitance by 50%. This provides an increase in sensitivity to touch or proximity.

#### Conversion Gain Definition

The Conversion Gain parameter specifies the desired number of charge transfers in the conversion before offset subtraction is applied. The runtime calibration algorithm will adjust the gain stage for each element (making the external capacitor look larger or smaller relative to the internal sample capacitor) until the gain ratio is identified that gets closest to the number of charge transfers specified by the Conversion Gain parameter.

---

### Conversion Count Definition

The Conversion Count parameter specifies the desired number of charge transfers in the conversion after offset subtraction has been applied. Since offset subtraction removes charge from each charge transfer, whenever the offset subtraction amount is increased, more charge transfers will be required in order to fill up the internal sample capacitor. The runtime calibration algorithm will first identify the gain ratios based on the conversion gain parameter, specified above. Then, it will begin increasing the amount of offset subtraction until the number of charge transfers in the conversion gets as close to the Conversion Count parameter as possible.

### Relationship Between Conversion Count and Gain

The Conversion Count specifies the approximate measurement result (or "count"), which is equivalent to the number of charge transfers that are required to fill up the internal sample capacitor. If this value is held constant, reducing the Conversion Gain has the effect of increasing the amount of offset subtraction that is applied. This has the effect of increasing sensitivity to touch without increasing the conversion time.

Note that the Conversion Count value must be equal to or larger than the Conversion Gain. When the two values are equal, the minimum amount of offset subtraction is applied.

### Implementation Notes

A typical Conversion Gain value that works well for a variety of applications is 200. This value can be decreased to 100 to apply additional offset subtraction, or it may be increased to apply less offset subtraction.

A typical Conversion Count value that works well for a variety of applications is 500. Button group sensors will often not need this much resolution, and can have their Conversion Count reduced. Slider, Wheel, and Proximity sensors may require additional, and may need their Conversion Count increased accordingly to provide it.

### Range of Valid Values

The conversion gain may be set anywhere between 100 and 8191. The default is 200. The conversion count may be set anywhere between 100 and 8191. The default is 500.

### Affected Software Parameters

The Conversion\_Gain parameter corresponds to the ***ui16ConversionGain*** member of the ***tSensor*** type in the CapTlivate Touch Library.

The Conversion\_Count parameter corresponds to the ***ui16ConversionCount*** member of the ***tSensor*** type in the CapTlivate Touch Library.

## 13.8 Conversion Gain

### Conversion Gain and Conversion Count

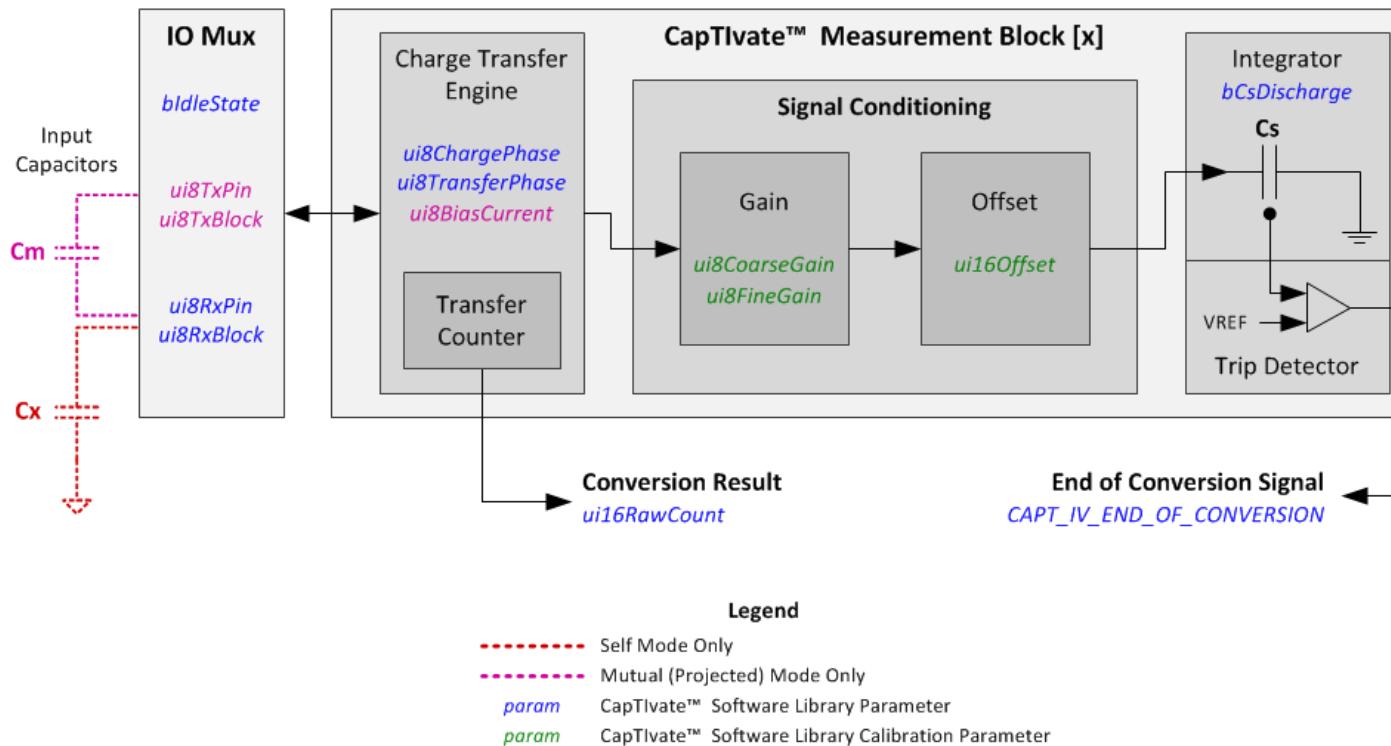
Conversion Gain and Conversion Count are the fundamental parameters used to establish the performance of the sensor. These parameters determine the resolution, sensitivity, and required conversion time. They are the inputs to the calibration algorithm, which identifies the correct coarse/fine gain ratios, offset scale, and offset level for each element at runtime.

#### Quick Reference

1. To increase resolution, increase the Conversion Count.
2. To increase sensitivity, increase the Conversion Count OR decrease the Conversion Gain.
3. The Conversion Count value establishes the conversion time, as it specifies the approximate number of charge transfers per conversion.
4. The Conversion Count value must be equal to or larger than the Conversion Gain. When the two values are equal, the minimum amount of offset subtraction is applied.

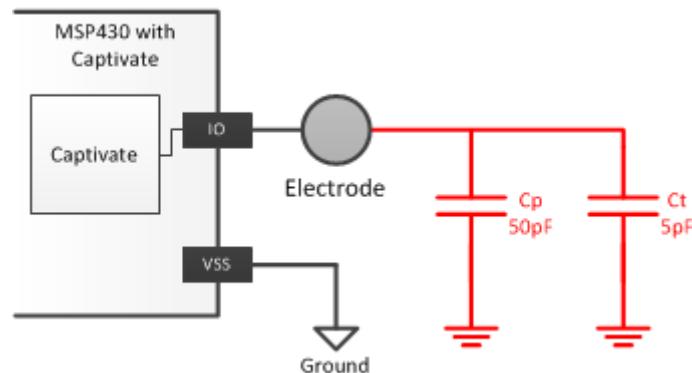
## Background

The CapTlivate peripheral has the ability to apply gain as well as offset to the capacitance being measured. When a conversion is started, the unknown external capacitance being measured is charged to a known voltage. The charge on that external capacitor is then transferred into a sample capacitor which is on-chip. The conversion result is simply the number of charge transfers from the external capacitor to the internal sample capacitor that are required to "fill up" the internal sample capacitor. The number of transfers needed to do so is representative of the capacitance of the external electrode being measured. The signal chain is shown below.



The gain stage of the CapTlivate peripheral provides the ability to scale the effective size of the unknown external capacitor relative to the internal sample capacitor. This serves two main purposes. First, it allows the peripheral to handle a wide range of capacitances. Second, it allows for the designer of the system to dial in a desired measurement resolution.

The offset stage of the CapTlivate peripheral provides a mechanism to remove a set amount of charge during each charge transfer. This charge is typically associated with the parasitic capacitance of the sensor. Parasitic capacitance can be thought of as an unwanted DC offset in the measurement. For example, if an electrode has an inherent parasitic capacitance of 50pF, and a touch on that electrode adds 5pF, the capacitance goes up by 10%. This model is shown below.



However, if charge associated with the parasitic capacitance was subtracted off such that the inherent parasitic capacitance of the electrode appeared as just 10pF, then a 5pF touch would increase the capacitance by 50%.

---

This provides an increase in sensitivity to touch or proximity.

### Conversion Gain Definition

The Conversion Gain parameter specifies the desired number of charge transfers in the conversion before offset subtraction is applied. The runtime calibration algorithm will adjust the gain stage for each element (making the external capacitor look larger or smaller relative to the internal sample capacitor) until the gain ratio is identified that gets closest to the number of charge transfers specified by the Conversion Gain parameter.

### Conversion Count Definition

The Conversion Count parameter specifies the desired number of charge transfers in the conversion after offset subtraction has been applied. Since offset subtraction removes charge from each charge transfer, whenever the offset subtraction amount is increased, more charge transfers will be required in order to fill up the internal sample capacitor. The runtime calibration algorithm will first identify the gain ratios based on the conversion gain parameter, specified above. Then, it will begin increasing the amount of offset subtraction until the number of charge transfers in the conversion gets as close to the Conversion Count parameter as possible.

### Relationship Between Conversion Count and Gain

The Conversion Count specifies the approximate measurement result (or "count"), which is equivalent to the number of charge transfers that are required to fill up the internal sample capacitor. If this value is held constant, reducing the Conversion Gain has the effect of increasing the amount of offset subtraction that is applied. This has the effect of increasing sensitivity to touch without increasing the conversion time.

Note that the Conversion Count value must be equal to or larger than the Conversion Gain. When the two values are equal, the minimum amount of offset subtraction is applied.

### Implementation Notes

A typical Conversion Gain value that works well for a variety of applications is 200. This value can be decreased to 100 to apply additional offset subtraction, or it may be increased to apply less offset subtraction.

A typical Conversion Count value that works well for a variety of applications is 500. Button group sensors will often not need this much resolution, and can have their Conversion Count reduced. Slider, Wheel, and Proximity sensors may require additional, and may need their Conversion Count increased accordingly to provide it.

### Range of Valid Values

The conversion gain may be set anywhere between 100 and 8191. The default is 200. The conversion count may be set anywhere between 100 and 8191. The default is 500.

### Affected Software Parameters

The Conversion\_Gain parameter corresponds to the ***ui16ConversionGain*** member of the ***tSensor*** type in the CapTlve Touch Library.

The Conversion\_Count parameter corresponds to the ***ui16ConversionCount*** member of the ***tSensor*** type in the CapTlve Touch Library.

## 13.9 Count

Count represents the measurement value for a given channel.

## 13.10 Count Filter

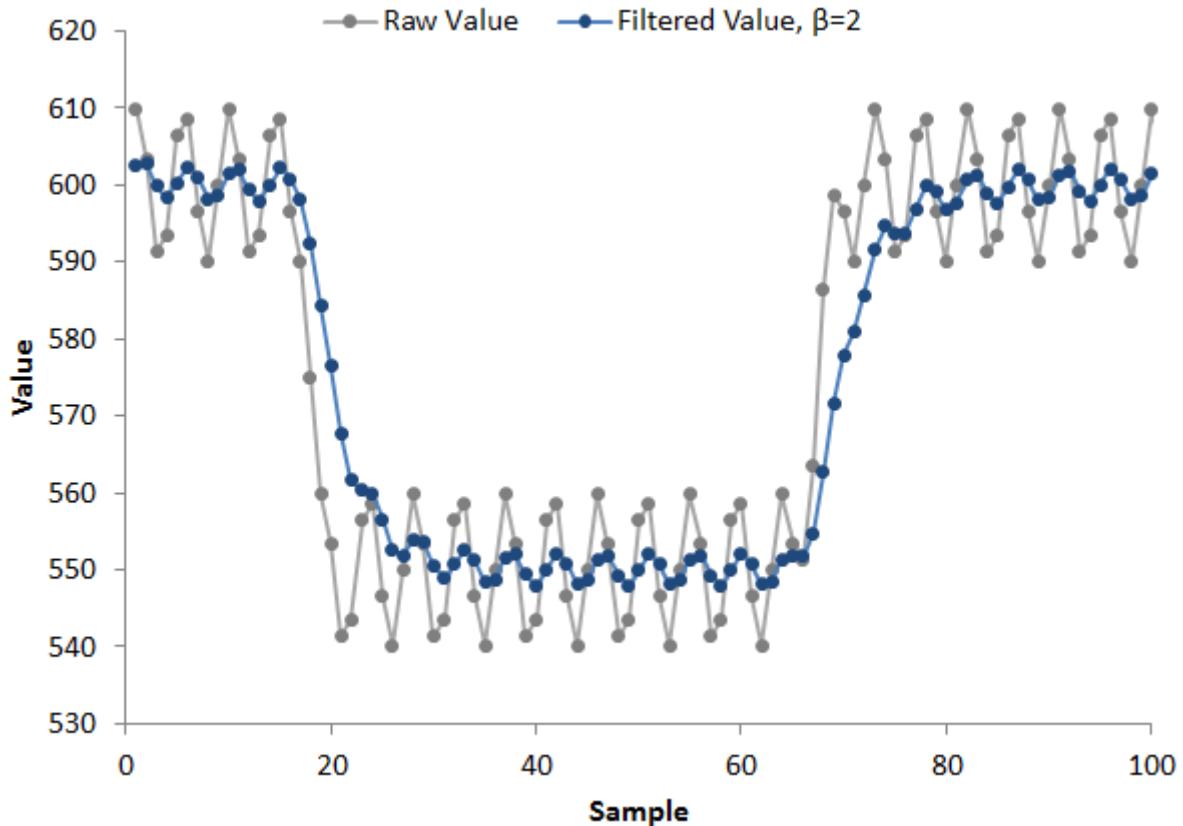
### Count Filter

The count filter provides a blocking mechanism for AC noise. It is a first-order IIR low-pass filter with 7 adjustable steps to control the filter strength. It may be enabled or disabled by toggling the Count\_Filter\_Enable parameter. The equation below defines the output of the filter. The previous filtered count value is combined with each new raw values according to this equation.

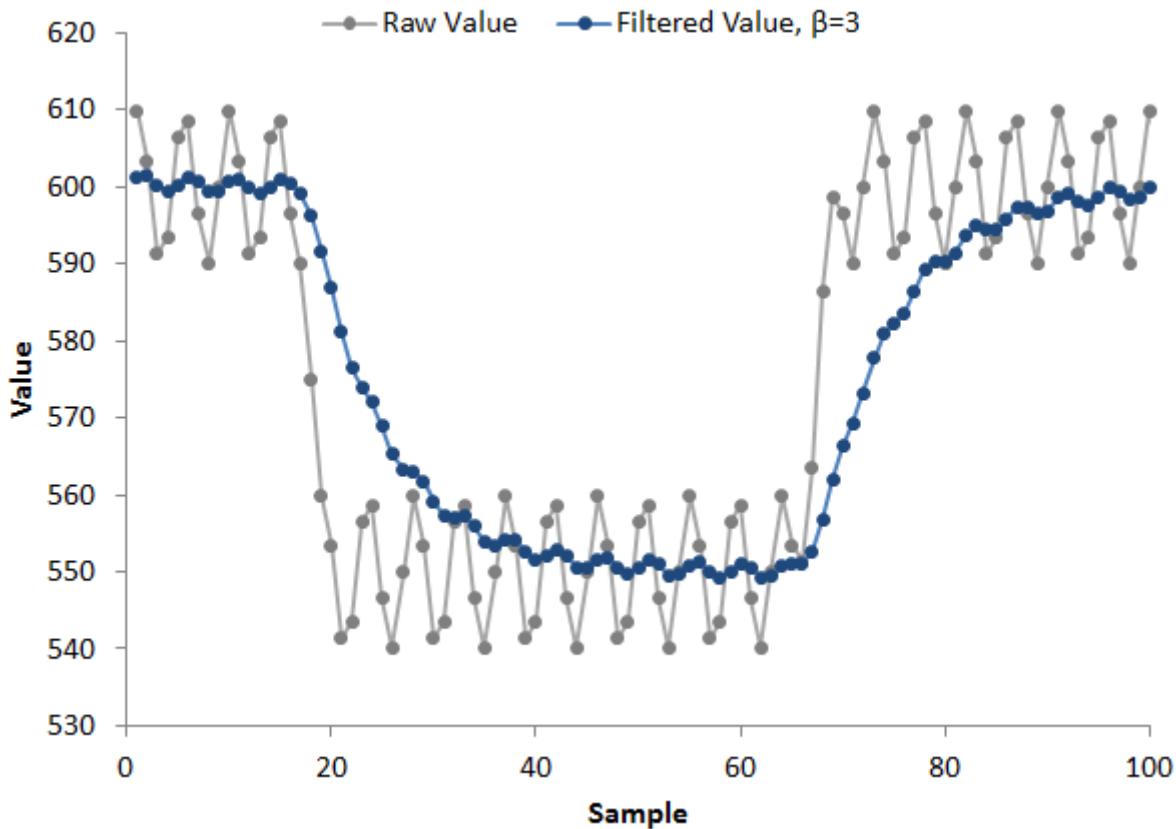
$$\text{NewFiltered} = \text{PrevFiltered} - \frac{\text{PrevFiltered}}{2^{\beta}} + \frac{\text{NewRaw}}{2^{\beta}}$$

---

If the count filter is enabled, its strength is controlled by the Count\_Filter\_Beta parameter. As the count filter beta value is increased, the attenuation of AC signals increases- but at the expense of DC response time. The two examples below illustrate this concept. A self-capacitance button in the presence of heavy AC noise is shown. The raw signal and the filtered signal are superimposed to show the effect of the filter. Notice how the DC component (a touch) may be passed, while the AC component is attenuated.



The second example below shows the use of a stronger count filter than the previous example. Note that while the noise attenuation is improved, there is now an increase in the response time to the desired DC signal.



#### Implementation Notes

A good starting point for implementing a count filter is a beta of 1. This is the weakest filter available, outside of full off. If heavy AC noise is present, increasing the beta may help stabilize the measurements. Note that increasing the beta will increase the response time of the system. It is also important to note that the filter value used is somewhat dependent upon the system scan rate. For example, a count filter beta of 1 at 50 Hz has similar AC-blocking characteristics as a count filter beta of 2 at 100 Hz, if the noise frequency is constant.

#### Range of Valid Values for the Count Filter Beta Parameter

The count filter beta may be set from 0 to 7, with zero being equivalent to off.

#### Affected Software Parameters

The Count\_Filter\_Enable parameter corresponds to the **bCountFilterEnable** member of the **tSensor** type in the CapTlivate Touch Library.

The Count\_Filter\_Beta parameter corresponds to the **ui8CntBeta** member of the **tSensor** type in the CapTlivate Touch Library.

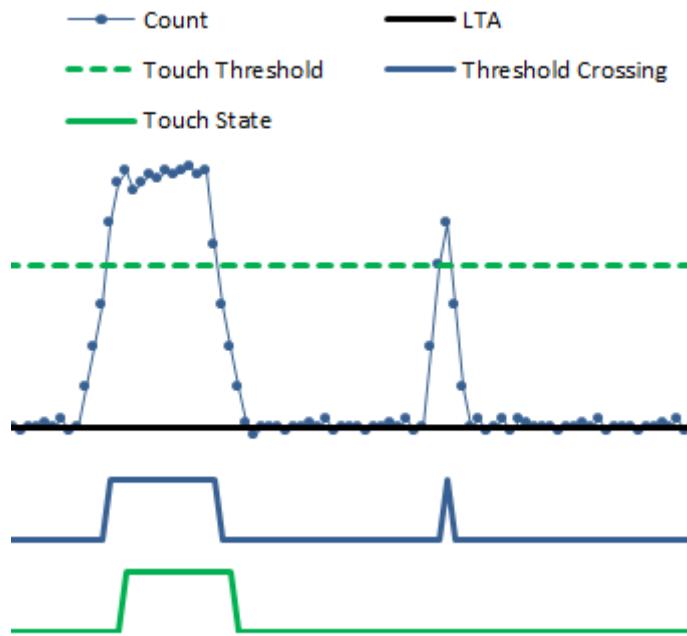
## 13.11 Debounce

#### De-Bounce

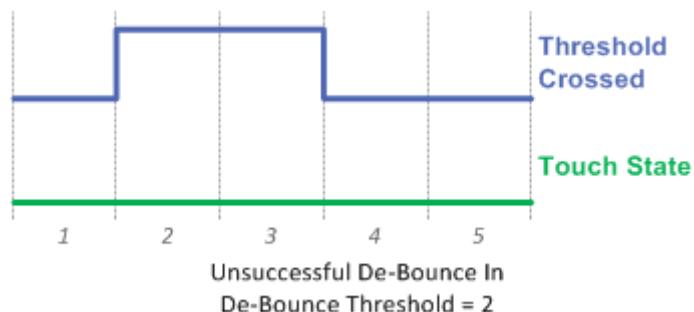
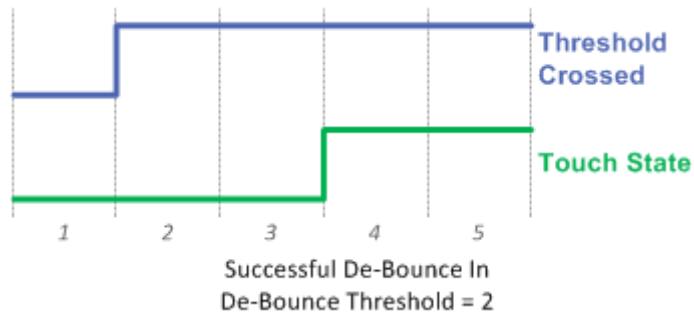
De-bouncing a capacitive touch sensor helps to ensure robust operation in noisy environments by providing control over how touch and proximity states are entered and exited. It works by requiring a sensor going through a state change (no-touch into touch, or touch into no-touch) to be in the new state for a certain number of samples before the state is actually updated in software. For example: with de-bounce switched off (set to 0), as soon as a touch threshold crossing occurs the touch state will be updated accordingly. If the de-bounce feature is switched on, and set to 1, a threshold crossing will not effect the state unless the measurement stays past the threshold for at least 1 (initial sample) + 1 (debounce sample) = 2 total samples.

Take the plot below as an example. The measurement results (over time for several samples) are plotted in the *Count* series. The *Threshold Crossing* logical series indicates when *Count* has passed the *Touch Threshold*. If no de-bounce is applied, then the *Touch State*, or our final output, will always be equal to the *Threshold Crossing*. In

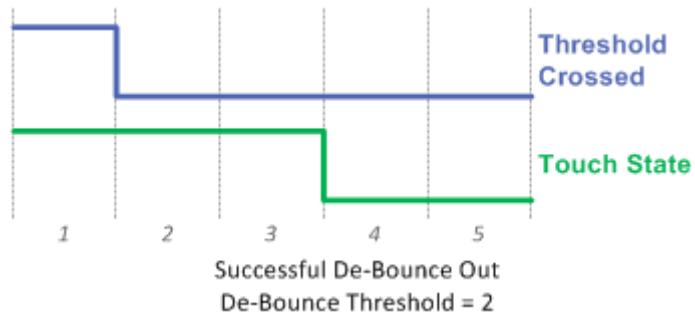
In this example, let's assume that the second spike is due to noise- not a touch. De-bounce can be used to suppress this event so that it does not effect the touch state. If a de-bounce threshold of 2 is set, the second spike (the noise spike) does not remain past the threshold long enough to trigger a touch state. Now that de-bounce is applied, the *Touch State* is no longer equal to the *Threshold Crossing*.



De-bounce may be applied to touch or proximity state detection. The thresholds may be set independently for touch and prox, as well as the rising edge (into a touch/prox state) and falling edge (out of a touch/prox state). Below is an example of a successful de-bounce (touch state entered) and an unsuccessful debounce (touch state not entered).



The falling edge feature (de-bounce out) is particularly useful when it is undesirable to detect a "new" touch if a user accidentally lets go of a button for a short period of time, and then returns it. The plots below demonstrate a successful exit of a touch state, and an unsuccessful exit (a temporary removal).



### Implementation Notes

De-bounce configurations are set on a sensor basis. All elements within a sensor will share the same de-bounce thresholds. There are four thresholds that may be set:

- Proximity De-Bounce In
- Proximity De-Bounce Out
- Touch De-Bounce In
- Touch De-Bounce Out

It is important to note that applying de-bounce to a sensor increases the response time of the sensor. For example, if a system is set up to measure at a 20 Hz rate (every 50 ms), a de-bounce of 1 increases the response time to a touch from 50ms to 100ms. This delay is in addition to any delay imposed by count filtering. If heavy de-bounce is to be used, it may be necessary to increase the sample rate.

### Range of Valid Values for De-bounce Thresholds

De-bounce thresholds may be set between 0 and 15. Zero is equivalent to no de-bounce.

### Affected Software Parameters

The `Prox_Debounce_In_Threshold` parameter corresponds to the `ProxDbThreshold.DbIn` member of the `tSensor` type in the CapTlivate Touch Library.

The `Prox_Debounce_Out_Threshold` parameter corresponds to the `ProxDbThreshold.DbOut` member of the `tSensor` type in the CapTlivate Touch Library.

The `Touch_Debounce_In_Threshold` parameter corresponds to the `TouchDbThreshold.DbIn` member of the `tSensor` type in the CapTlivate Touch Library.

The `Touch_Debounce_Out_Threshold` parameter corresponds to the `TouchDbThreshold.DbOut` member of the `tSensor` type in the CapTlivate Touch Library.

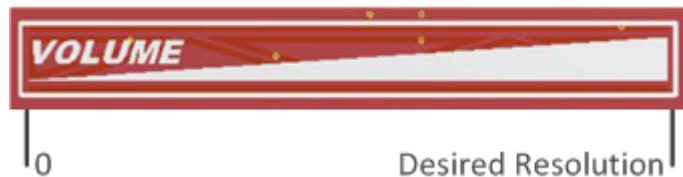
## 13.12 Delta

Delta typically refers to the change in a measurement due to an electrode's change in capacitance caused by a proximity or touch condition. When viewing a sensors output the CapTlivate™ Design Center to view theThis is typically seen in the CapTlivate™ Design Center sensor view during the sensor tuning process.

## 13.13 Desired Resolution

### Desired Resolution

The desired resolution parameter specifies the resolution of the position that is calculated by the slider and wheel processing algorithm. Slider and wheel sensors are one-dimensional sensors that report back a position as their output. When selecting a resolution, the value reported back for the position will be in the range of 0 to resolution-1. For example, if a resolution of 100 is selected, the values will range from 0 to 99 (100 total points of interest).



### Implementation Notes

While resolutions up to 16-bit are supported by the algorithm, most realistic use cases involve 8-bit resolution (256 points of interest along the slider or wheel).

### Range of Valid Values for the Desired Resolution Parameter

The desired resolution may be set from 3 to 65535.

### Affected Software Parameters

The Desired\_Resolution parameter corresponds to the ***ui16Resolution*** member of the ***tSliderSensorParams*** and ***tWheelSensorParams*** types in the CapTlivate Touch Library.

## 13.14 Electrode

An electrode is the physical conductive structure that a person interacts with. This structure is typically thought of as the copper on a printed circuit board (PCB), but can also be made of transparent materials such as indium tin oxide (ITO) or other conductive materials like silver.

## 13.15 Element

An element is associated with a sensor's electrodes. For example, if a slider has four electrodes, regardless if the sensor is a self-capacitive sensor or mutual capacitive, then each electrode is considered an element of the slider sensor. This is not as important for buttons however, for sliders and wheels software uses the element to electrode assignments to correctly process position and direction.

## 13.16 Engineering Parameters

### Engineering Parameters:

#### Coarse Gain, Fine Gain, and Offset Subtraction

The coarse gain ratio, fine gain ratio, and offset subtraction are engineering controls for advanced users during the development process. These parameters exist to provide visibility into the behaviour of the runtime calibration algorithm.

The primary controls for adjusting sensitivity and resolution are the Conversion\_Count and Conversion\_Gain parameters. At runtime, these two parameters are inputs to the calibration algorithm, which adjusts the peripheral control knobs: coarse gain, fine gain, and offset subtraction- to achieve the requested performance for each element in each sensor. The configuration for each element is stored in a structure that get associated with each element. It is important to note that these values are recalculated after every device reset, and may differ

---

depending on where a system is used. If the runtime re-calibration feature is enabled, any time an element's LTA drifts out of range due to environmental drift the sensor will be re-calibrated to regain the desired sensitivity. These parameters are provided for development. Note that any values set here will not persist after the target device is reset or the sensor is recalibrated.

#### **Coarse Gain Ratio**

The coarse gain ratio applies a coarse scaling factor to the relationship between the external capacitor being measured and the internal sample capacitor.

#### **Fine Gain Ratio**

The fine gain ratio applies a fine scaling factor to the relationship between the external capacitor being measured and the internal sample capacitor.

#### **Offset Subtraction**

The offset subtraction applies a capacitive DC offset, reducing the component of the measurement that is associated with parasitic capacitance.

#### **Affected Software Parameters**

The Coarse\_Gain\_Ratio parameter corresponds to the ***ui8CoarseGainRatio*** member of the ***tCaptivateElementTuning*** type in the CapTlivate Touch Library.

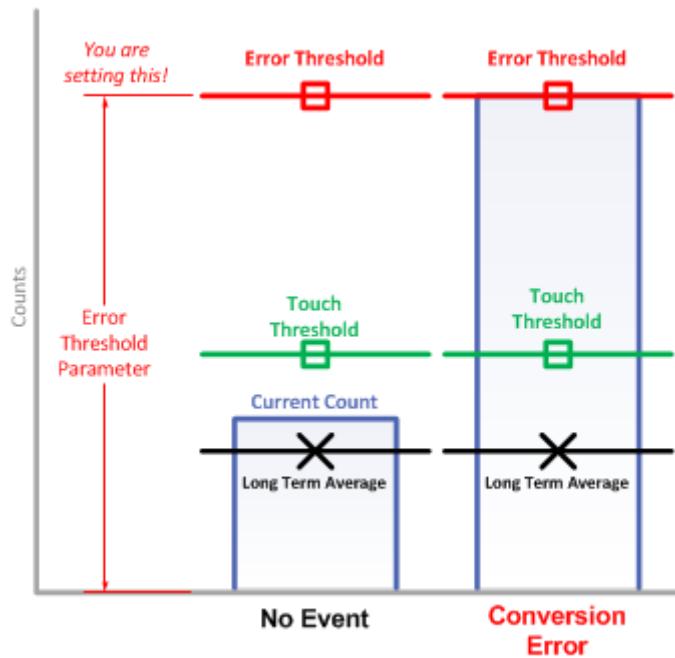
The Fine\_Gain\_Ratio parameter corresponds to the ***ui8FineGainRatio*** member of the ***tCaptivateElementTuning*** type in the CapTlivate Touch Library.

The Offset\_Subtraction parameter corresponds to the ***ui16OffsetTap*** member of the ***tCaptivateElementTuning*** type in the CapTlivate Touch Library.

## **13.17 Error Threshold**

#### **Error Threshold**

The error threshold specifies the maximum conversion result allowed for any element within a sensor. Error threshold crossings are detected in hardware. If during the conversion process one or more elements does not complete their conversion before the conversion result crosses the error threshold, the peripheral will terminate the conversion and the error condition will be flagged in software at a sensor level. This mechanism prevents a conversion from continuing forever in the event of an incorrect calibration or a hardware problem. Hardware problems may include, but are not limited to, a disconnected electrode or a shorted electrode. The figure below illustrates an error threshold crossing. Note that the conversion result never exceeds the error threshold, as the CapTlivate peripheral stops the conversion upon crossing of the error threshold. Unlike other thresholds, the error threshold is an absolute threshold, and is not relative to the LTA.



### Implementation Notes

Conversion results are 13-bit values. As such, the maximum conversion result as well as the maximum error threshold is 8191. This is the default error threshold, and is fine for most applications. However, the error threshold may be reduced here if desired. Lowering the error threshold limits how long the peripheral may continue trying to complete the conversion. It is up to the application to determine how to handle an error condition in software. Several steps may be taken to determine why the error condition occurred, such as a re-calibration or a self-test.

### Range of Valid Values for the Error Threshold Parameter

The error threshold may be set from 0 to 8191. Conversions are limited in hardware to a maximum result of 8191.

### Affected Software Parameters

The `Error_Threshold` parameter corresponds to the `ui16ErrorThreshold` member of the `tSensor` type in the CapTlve Touch Library.

If an error condition occurs at runtime, the `bMaxCountError` member of the affected `tSensor` instance will be set to alert the application that there was an error in the conversion.

## 13.18 Frequency Divider

### Conversion Frequency Divider

The conversion frequency divider allows for the conversion clock to be divided down from the base rate of 16 MHz. The conversion clock period must be long enough to ensure complete charge transfer phases. This can be verified on an oscilloscope by probing the electrode pins if a sensor is experiencing noise.

Divider	Base Frequency
/1	16 MHz
/2	8 MHz
/4	4 MHz
/8	2 MHz
/16	1 MHz
/32	500 kHz

---

/64	250 kHz
/128	125 kHz

#### Implementation Notes

The conversion clock divider works in series with the conversion charge and transfer phase lengths. The effective overall conversion rate is a function of the clock divider, the charge/hold phase length, and the transfer/sample phase length.

#### Affected Software Parameters

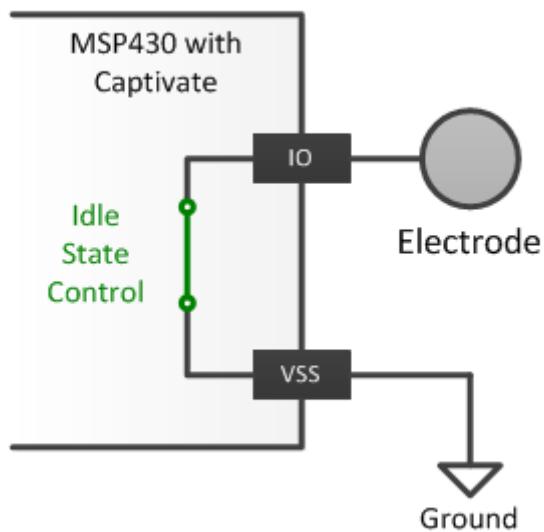
The Frequency\_Divider parameter corresponds to the *ui8FreqDiv* member of the *iSensor* type in the CapTlve Touch Library.

## 13.19 Idle State

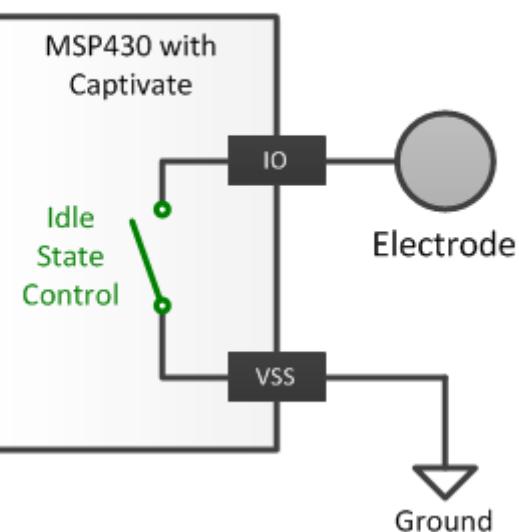
### Pin Idle State

The idle state specifies the drive state of a sensor's pins when the sensor is not actively being measured. Pins have the option of being driven low to ground (VSS), or left tri-stated (high-impedance).

### Grounded (VSS)



### High-Z (High Impedance)



---

### Implementation Notes

For most applications, pins should be grounded when not being measured. This is important especially if there are other sensors in the system. When sensor pins (and their connected electrodes) are left floating, they can capacitively couple with the electrodes of nearby sensors while those sensors are being measured. This can cause cross-triggering. For some applications it is desirable to float pins (set to High-Z) when not being measured to prevent coupling to ground. Water and moisture suppression would be an example of such a use case.

### Affected Software Parameters

The Idle\_State parameter corresponds to the **idleState** member of the **tSensor** type in the CapTlvate Touch Library.

## 13.20 LTA Filter

### Long Term Average (LTA) Filter

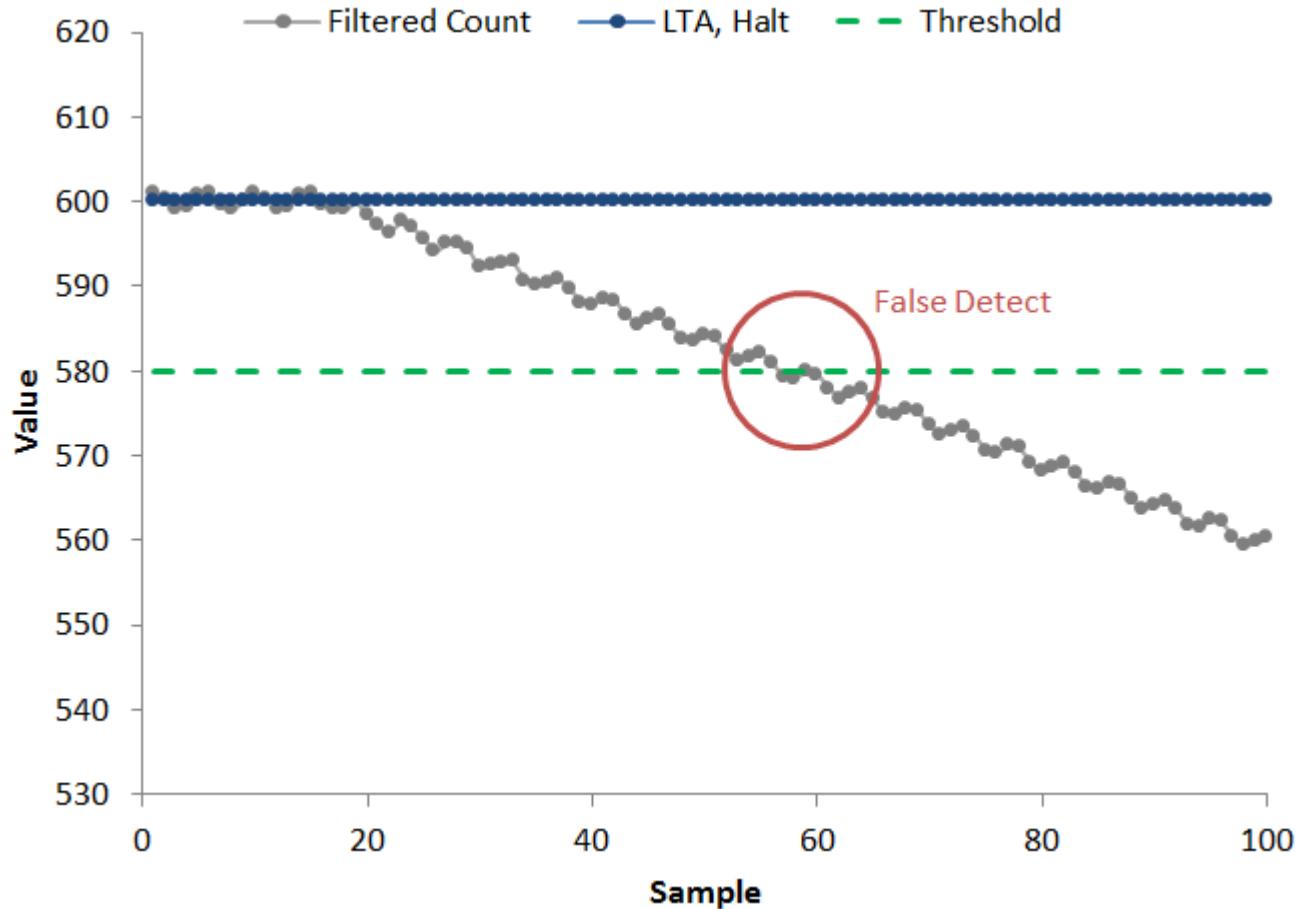
The long-term-average (LTA) filter allows the LTA for each element to compensate for environmental drift. Environmental drift may be caused by physical or electrical changes. Examples of physical changes include temperature, humidity, and the presence of nearby conductors. Examples of electrical changes include operating voltage and power source (battery versus grounded mains power).

The LTA filter is a first-order IIR low-pass filter with 7 adjustable steps to control the filter strength. The LTA filter is enabled by default. To disable it, set the Halt\_LTA\_Filter\_Immediately parameter.

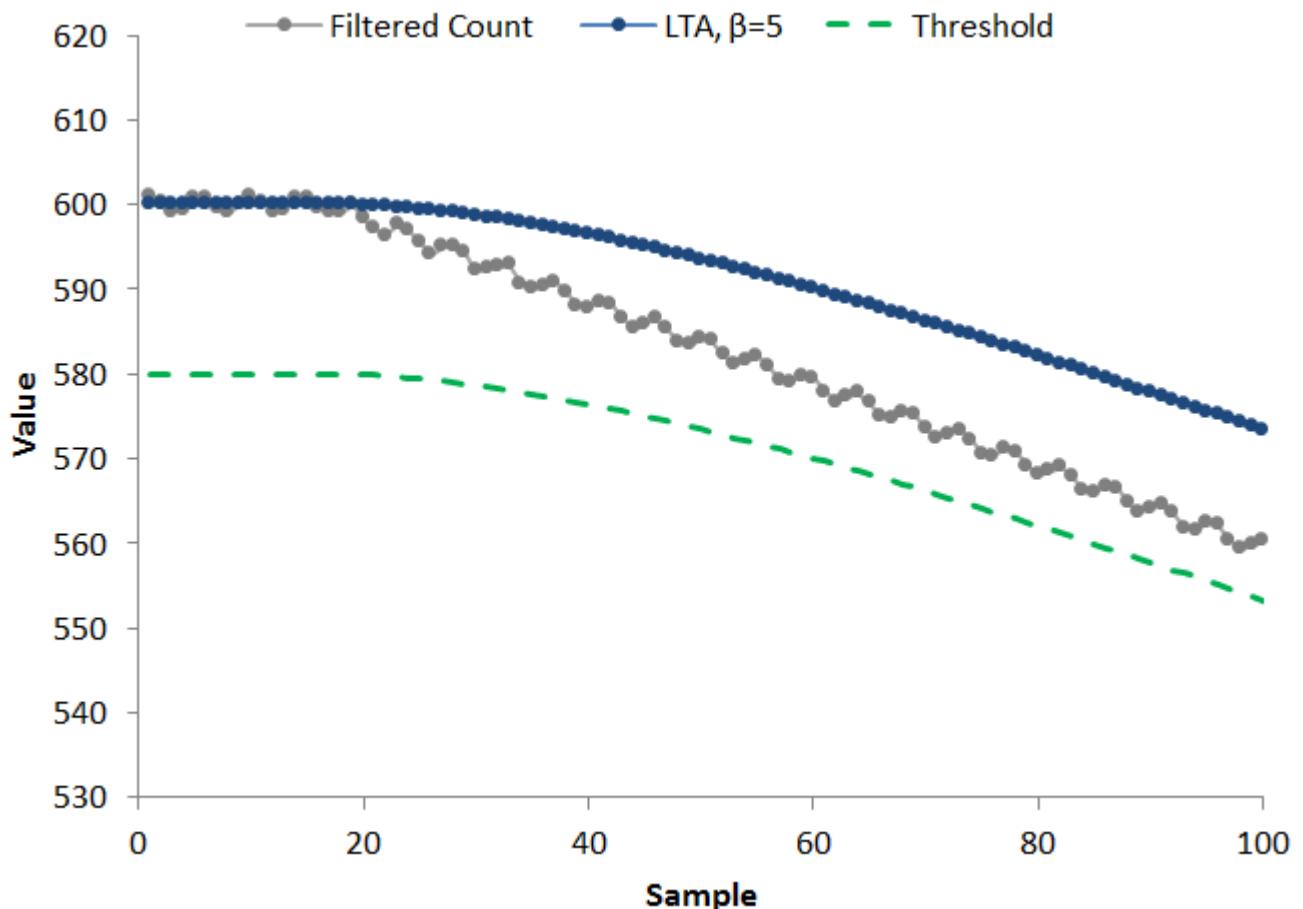
The filter mechanism used to maintain the long term average is identical to that used for the count filter- the difference is the filter strength (corner frequency) that is selected. The LTA filter typically uses a very strong beta (usually the maximum of 7). This is because environmental changes are typically slow when compared to a touch or proximity event. The LTA filter strength is set in the LTA\_Filter\_Beta parameter.

$$NewFiltered = PrevFiltered - \frac{PrevFiltered}{2^\beta} + \frac{NewRaw}{2^\beta}$$

The example below illustrates how the LTA filter works to adapt to environmental changes. Consider the data set below, where the filtered count slowly drifts downward. This could be due to a sudden environment change (moving from a climate-controlled room to an outdoor environment, for example). If the LTA filter is halted (as shown below), the filtered count will eventually dip below the touch threshold, and a false touch will be reported (as shown by the red circle).



This false touch may be avoided through the use of LTA filtering. An assumption is made that environmental changes are slow changes (relative to the speed of a touch). With the LTA filter active, the LTA tracks with the slow environmental change, and a false threshold crossing never occurs.



### Implementation Notes

A good starting point for implementing the LTA filter is a beta of 7. This is the strongest filter available, outside of an LTA halt. It is also important to note that the filter value used is dependent upon the system scan rate. For example, a LTA filter beta of 6 at 50 Hz has similar characteristics as one with a beta of 7 at 100 Hz.

The LTA Control parameters provide the ability to control when the LTA is stopped, or "halted." There are a variety of circumstances where it is necessary to halt an element's long term average. Halting the LTA essentially "locks" the LTA to its current value. When halted, the LTA is not updated with new conversion results. When an element crosses into a touch or proximity state, the LTA must be halted to prevent the LTA from tracking to the new proximity or touch state. If an element's LTA is not halted during a touch or proximity event, the touch and/or proximity event would be "tracked" out and would be considered the new baseline. By default, the LTA filter is halted automatically when a proximity or touch threshold is crossed for the element. Other halting options are available via the `Halt_LTA_On_Sensor_Prox_Or_Touch` and `Halt_LTA_On_Element_Prox_Or_Touch` parameters. If `Halt_LTA_On_Element_Prox_Or_Touch` is cleared, the LTA will continue to track even if an element is in detect. Setting `Halt_LTA_On_Sensor_Prox_Or_Touch` will halt the LTA of every element in the sensor if any one of them is in prox or touch detect. `Halt_LTA_On_Sensor_Prox_Or_Touch` overrides `Halt_LTA_On_Element_Prox_Or_Touch`.

### Affected Software Parameters

The `LTA_Filter_Beta` parameter corresponds to the `ui8LTabeta` member of the `tSensor` type in the CapTlivate Touch Library.

The `Halt_LTA_Filter_Immediately` parameter corresponds to the `bSensorHalt` member of the `tSensor` type in the CapTlivate Touch Library.

The `Halt_LTA_On_Element_Prox_Or_Touch` parameter corresponds to the `bPTElementHalt` member of the `tSensor` type in the CapTlivate Touch Library. This is set by default.

The `Halt_LTA_On_Sensor_Prox_Or_Touch` parameter corresponds to the `bPTSensorHalt` member of the `tSensor` type in the CapTlivate Touch Library.

---

## 13.21 Modulation Enable

### Oscillator Modulation Enable

The CapTlVate peripheral oscillator incorporates a spread-spectrum clocking feature for reducing electromagnetic radiation. When modulation is enabled, the CapTlVate oscillator will dither clock edges, spreading out clock energy across a wider frequency band. This has an effect on emissions as well as susceptibility.

### Implementation Notes

If nearby systems are effected by the capacitive touch signals being driven by the CapTlVate peripheral, enabling modulation will reduce the peak radiated energy by spreading out the energy any may help improve electromagnetic compatibility. For self-capacitance designs the require noise immunity, modulation should be enabled. For mutual-capacitance designs that require noise immunity, modulation should be disabled.

### Affected Software Parameters

The Modulation\_Enable parameter corresponds to the **bModEnable** member of the **tSensor** type in the CapTlVate Touch Library.

---

## 13.22 Mutual Capacitance

## 13.23 Negative Touch Threshold

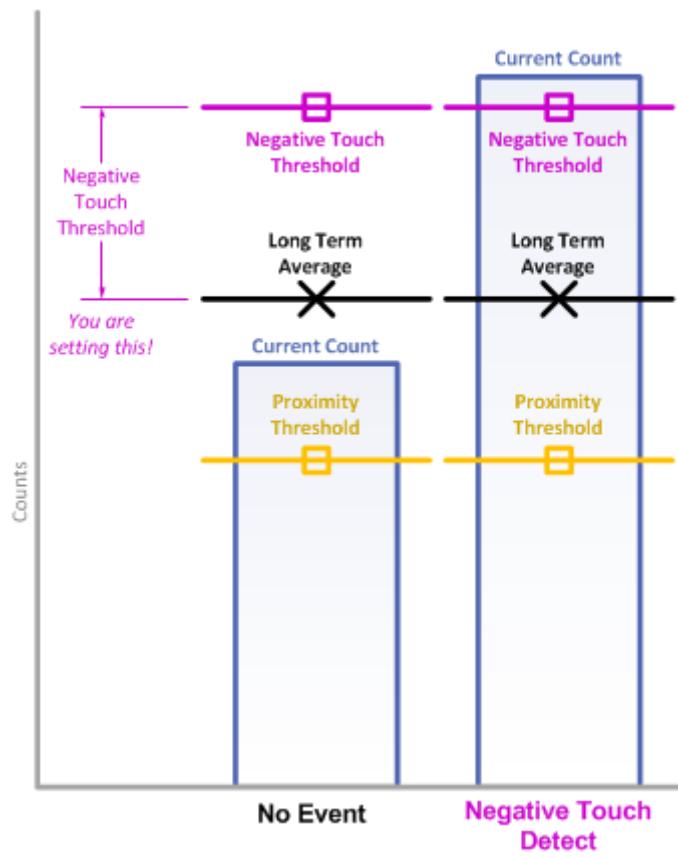
### Negative Touch Threshold

The negative touch threshold provides a mechanism for detecting when an element's measurement count adjusts quickly in the direction opposite of a touch. This scenario might happen during device power-up if a user has their hand covering an electrode during power-up calibration. The library makes the assumption at power-up that there is no user interaction. This may not always be the case. If the system is calibrated with a user touching a sensor, when they remove their hand, the measurements from that sensor's elements will quickly adjust in the opposite direction of a touch. Setting a negative touch threshold provides a way to quickly detect this event. Upon detection, the application may want to re-calibrate to ensure full sensitivity from the electrode immediately following the "negative" touch.

The negative touch threshold value sets the delta from the LTA required for a negative touch detection to be declared. It is set as an absolute deviation from the long term average. In other words, it specifies how large the delta must be against the direction of interest (against the direction of a touch) for a negative touch detection to be declared.

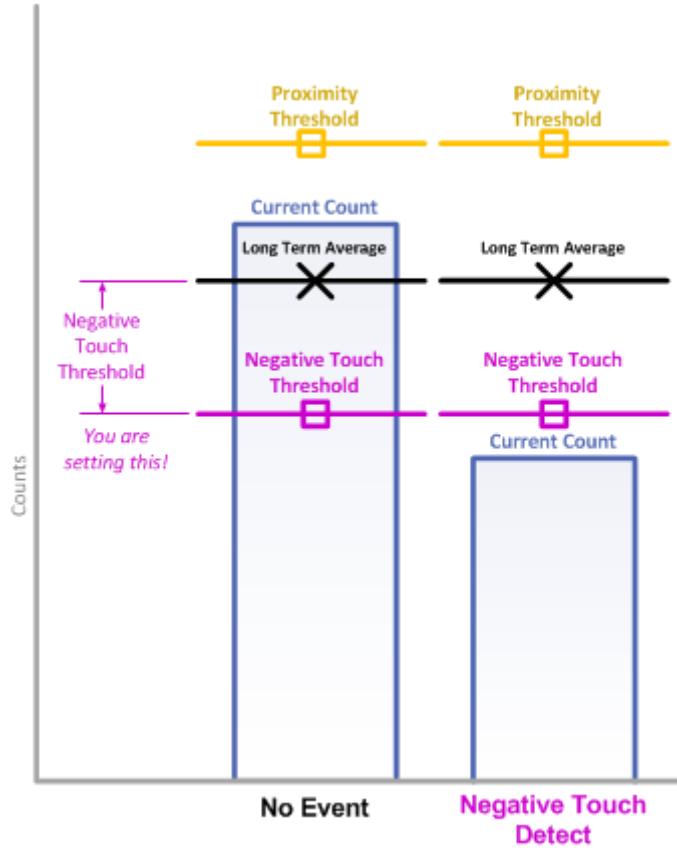
### Self Capacitance Example

When measuring an element in self-capacitance mode, a negative touch on an electrode causes the capacitance of that electrode to decrease. This causes an increase in the measurement result. When an increase in the measurement surpasses the negative touch threshold, the negative touch flags are set in the element and sensor structures.



### Mutual Capacitance Example

In mutual-capacitance mode, a negative touch on an electrode causes the mutual capacitance between the Rx and Tx to increase. This causes a decrease in the measurement result. When a decrease in the measurement surpasses the negative touch threshold, the negative touch flags are set.



The absolute negative touch threshold at any given time is defined by the following:

$$\begin{aligned} \text{NegativeTouchThresh}_{\text{absolute}} \\ = \begin{cases} \text{Self: } \text{LTA} + \text{NegativeTouchThresh}_{\text{programmed}} \\ \text{Mutual: } \text{LTA} - \text{NegativeTouchThresh}_{\text{programmed}} \end{cases} \end{aligned}$$

### Implementation Notes

One negative touch threshold is set and used for a whole sensor. This means that you only need to set one threshold per sensor, and all elements underneath that sensor will use that threshold. While only one threshold is set, each element still tracks and shows its negative touch status independently.

### Affected Software Parameters

The `Negative_Touch_Threshold` parameter corresponds to the `ui16NegativeTouchThreshold` member of the `tSensor` type in the CapTlivate Touch Library.

The `bNegativeTouch` member of the `tElement` type in the CapTlivate Touch Library is set when an element has a negative touch detection.

The `bSensorNegativeTouch` member of the `tSensor` type in the CapTlivate Touch Library is set when any element in that sensor experiences a negative touch detection.

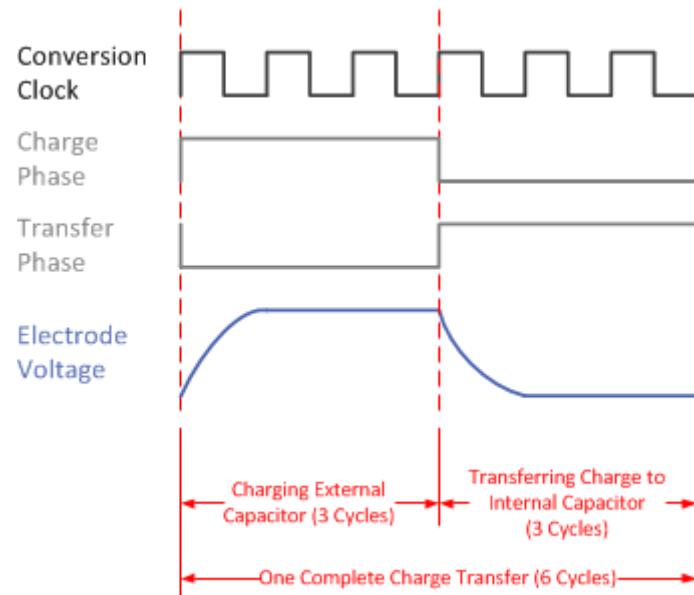
## 13.24 Noise Immunity

## 13.25 Parallel Sense Block

## 13.26 Phase Lengths

### Phase Lengths

The two phase length parameters (`Charge_Hold_Phase_Length` and `Transfer_Sample_Phase_Length`) control the length of the charge/hold and transfer/sample phases, respectively. The length is set in conversion clock periods. Thus, the total phase length is the conversion oscillator base frequency divided by the conversion clock divider (the `Frequency_Divider` parameter), multiplied by the phase length parameter. Each measurement, or "conversion," consists of a series of charge transfers. For a self capacitance sensor, a charge transfer is composed of a charge phase and a transfer phase. During the charge phase, the external capacitor being measured is charged to a known voltage. Then, during the transfer phase, that charge is moved into an on-chip tank capacitor (the sample capacitor). Charge transfers run until the sample capacitor is full. The output of the conversion is the number of charge transfers required to fill the sample capacitor. The diagram below illustrates the self capacitance case, with both phases set to 3.



The charge phase must be set long enough to ensure that the external electrode being measured is getting fully charged each cycle. The transfer phase must be set long enough to ensure that the charge is fully transferred to the sample capacitor. Larger external capacitances require longer phase lengths, as they take longer to charge and discharge.

Mutual capacitance works in a similar way. There is a hold phase and a sample phase. During the hold phase, the transmit electrode is grounded and the receive electrode is driven by the previous sampled voltage to remove any effects of parasitic capacitance to ground. During the sample phase, the transmit electrode is pulled up to a known voltage, reversing the polarity of the mutual capacitor and causing charge to be pushed through the mutual capacitance and into the internal sample capacitor.

### Implementation Notes

The effective phase lengths in units of time is also adjustable by increasing or decreasing the frequency divider that sources the conversion clock.

### Affected Software Parameters

The `Charge_Hold_Phase_Length` parameter corresponds to the `ui8ChargeLength` member of the `tSensor` type in the CapTlVate Touch Library.

The `Transfer_Sample_Phase_Length` parameter corresponds to the `ui8TransferLength` member of the `tSensor` type in the CapTlVate Touch Library.

## 13.27 Position Filter

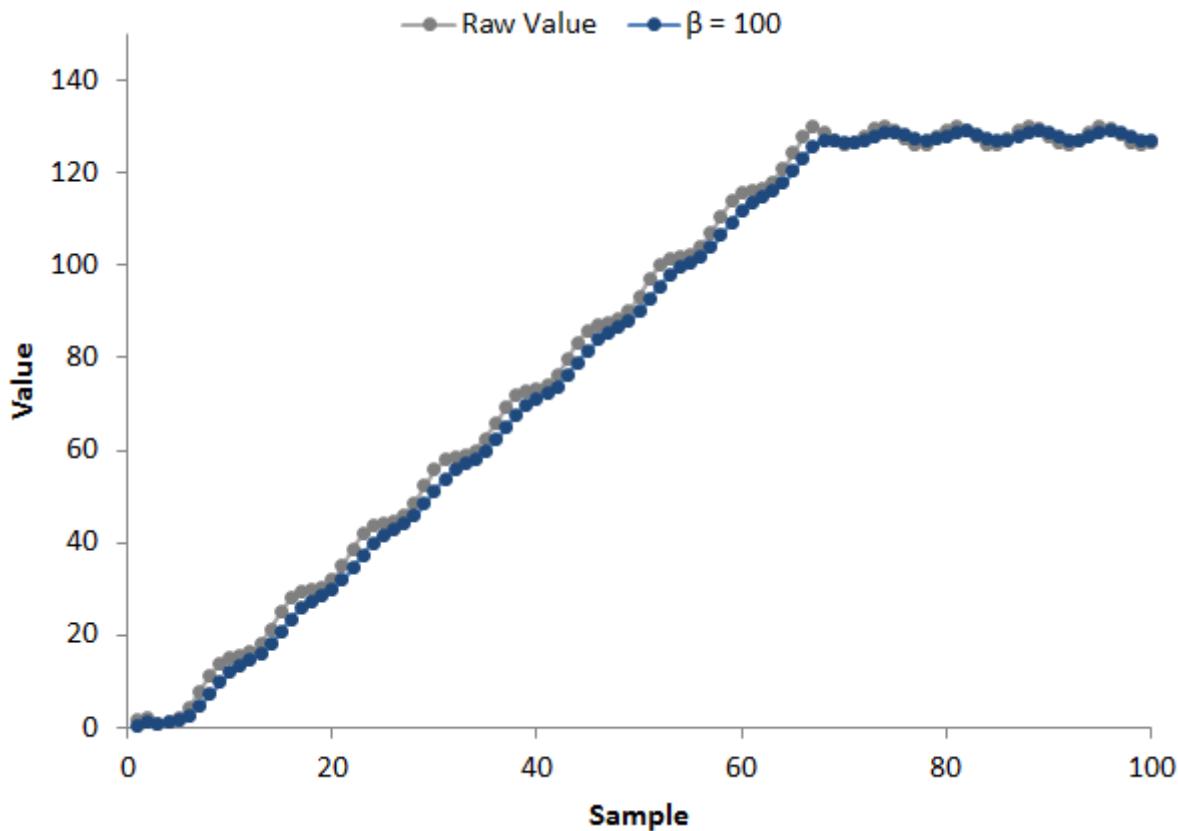
### Position Filter

The position filter provides output smoothing to positional sensors such as sliders, wheels, and trackpads. It is a first-order IIR low-pass filter with 255 adjustable steps to control the filter strength. It may be enabled or disabled by toggling the Position\_Filter\_Enable parameter. The filter characteristics are the same as the count filter and LTA filter, but the position filter allows for a greater level of adjustment- 255 different filter strengths, rather than the 7 allowed by the basic count and LTA filter. The position filter is best applied to sensors with resolutions higher than 16 points. Below 16 points, filtering is generally not required for stable selection of a position.

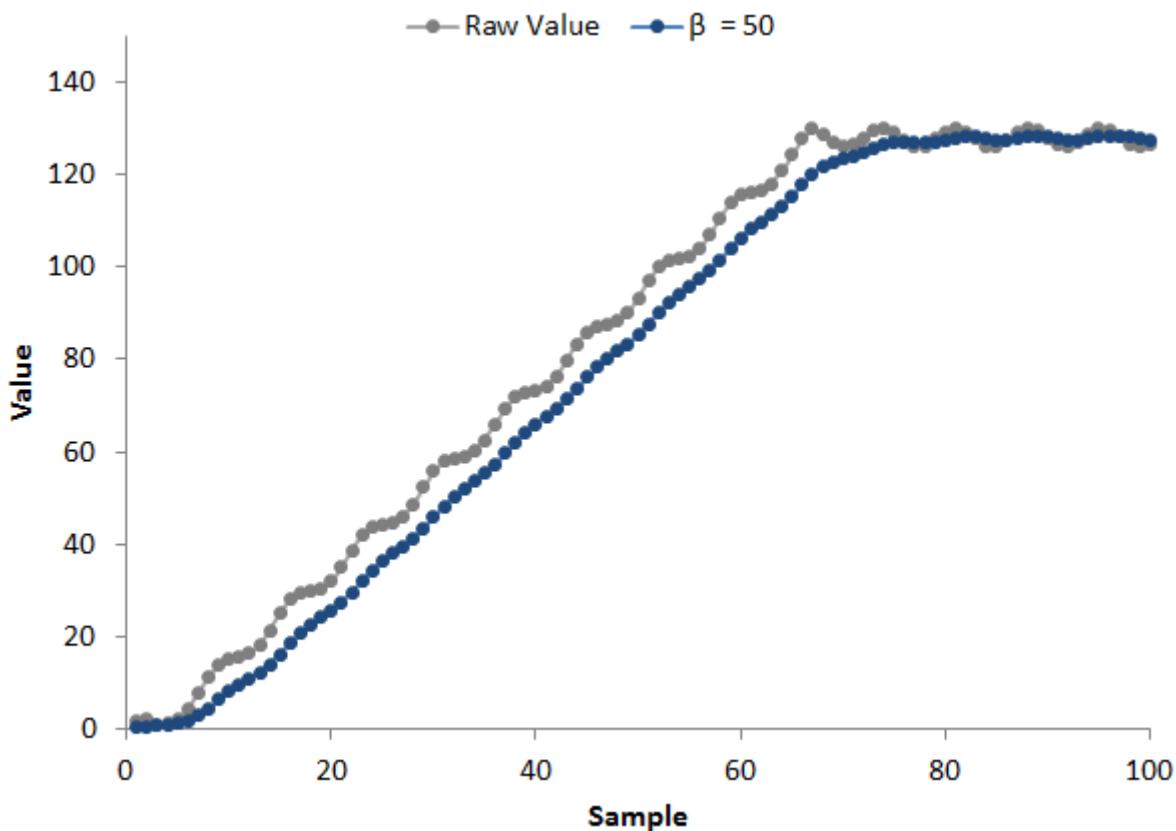
The equation below defines the output of the filter. The previous filtered position value is combined with each new raw position according to this equation.

$$\text{New Filtered Value} = \frac{(ui64Filtered)(256) + (ui64Raw - ui64Filtered)\beta}{256}$$

If the position filter is enabled, its strength is controlled by the Position\_Filter\_Beta parameter. As the beta value is decreased, the attenuation of AC signals increases, but at the expense of DC response time. The two examples below illustrate this concept. The position output of a 7-bit slider is shown, with the raw waveform displaying the samples captured while a user moved their finger from position 0 to position 128. The filtered signal is superimposed over the raw value to show the effect of "smoothing" the position output.



The second example below shows the use of a stronger filter than the previous example. In this example, almost all of the jitter from the raw measurement was removed in the filtering process. However, a small lag time was introduced. A small lag is typically far less noticeable to a user than jitter, so applying the stronger filter provides better perceived performance.



### Implementation Notes

A good starting point for implementing a position filter is a beta of 150. Note that decreasing the beta will increase the response time of the system. It is also important to note that the filter strength is also dependent upon the system scan rate. For example, a filter beta of 100 at 25 Hz has similar smoothing characteristics as a filter beta of 50 at 50 Hz. However, measuring the sensor at a higher report rate and strengthening the filter (by decreasing the beta) often provides better performance, as there is an oversampling effect. Note that the filtered value is reseeded immediately when a new touch is detected.

### Range of Valid Values for the Count Filter Beta Parameter

The position filter beta may be set from 0 to 255, with 255 being equivalent to off.

### Affected Software Parameters

The Position\_Filter\_Enable parameter corresponds to the **bSliderFilterEnable** member of the **tSliderSensorParams** and **tWheelSensorParams** types in the CapTlivate Touch Library.

The Position\_Filter\_Beta parameter corresponds to the **ui8SliderBeta** member of the **tSliderSensorParams** and **tWheelSensorParams** types in the CapTlivate Touch Library.

## 13.28 Proximity Sensor

Proximity sensors are typically self-capacitive electrodes with high sensitivity and can be any size or shape. Proximity sensors are used to detect the presence of an approaching finger or hand. A typical use of a proximity sensor is a "wake on proximity/touch", allowing the MCU to remain in a low power mode until a proximity event is detected. This is possible because the CapTlivate™ peripheral can perform the measurement, environmental tracking and proximity detection autonomously from the MCU.

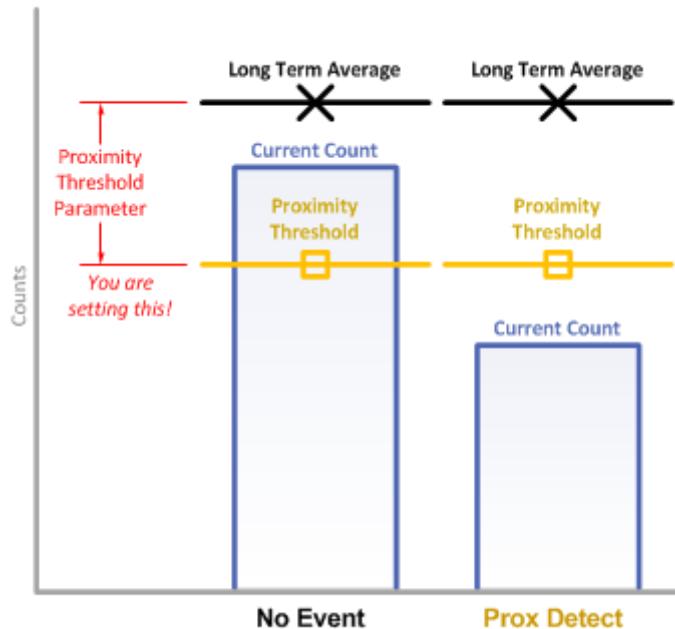
## 13.29 Proximity Threshold

### Proximity Threshold

The proximity threshold sets the level of interaction required by the user for a proximity detection to be declared for an element. The proximity threshold is set as an absolute deviation from the long term average. In other words, it specifies how large the delta must be in the direction of interest for a proximity state to be declared.

### Self Capacitance Example

When measuring an element in self-capacitance mode, a user's touch on or proximity to the element's electrode causes the capacitance of that electrode to increase. This causes a decrease in the measurement result. When a decrease in the measurement surpasses the proximity threshold, the proximity flag is set.



### Mutual Capacitance Example

In mutual-capacitance mode, a user's touch on or proximity to an element's electrode causes the mutual capacitance between the Rx and Tx to decrease. This causes an increase in the measurement result. When an increase in the measurement surpasses the proximity threshold, the proximity flag is set.



The absolute delta at any given time is defined by the following:

$$\text{ProxThresh}_{\text{absolute}} = \begin{cases} \text{Self: } LTA - \text{ProxThresh}_{\text{programmed}} \\ \text{Mutual: } LTA + \text{ProxThresh}_{\text{programmed}} \end{cases}$$

### Implementation Notes

One proximity threshold is set and used for a whole sensor. This means that you only need to set one proximity threshold per sensor, and all elements underneath that sensor will use that threshold. While only one threshold is set, each element still tracks and shows its proximity status independently. Note that the proximity detect flag is also dependent upon the de-bounce process. If de-bounce is used, the proximity status flag will not be set immediately after the threshold crossing. A global sensor proximity flag is made available at the sensor level. A proximity detect on any element will cause the sensor's global proximity detect flag to be set.

### Range of Valid Values for the Proximity Threshold Parameter

The proximity threshold may be set from 1 to 8191.

### Affected Software Parameters

The Prox\_Threshold parameter corresponds to the ***ui16ProxThreshold*** member of the ***tSensor*** type in the CapTivate Touch Library.

If a prox detect occurs at runtime, the ***bSensorProx*** member of the related ***tSensor*** instance will be set to true, alerting the application that an element on that sensor entered a prox state.

In addition, the ***bProx*** member of the related ***tElement*** instance will also be set to true, allowing the application to determine which specific element(s) entered into a prox state.

## 13.30 Run Time Recalibration

### Runtime Re-Calibration

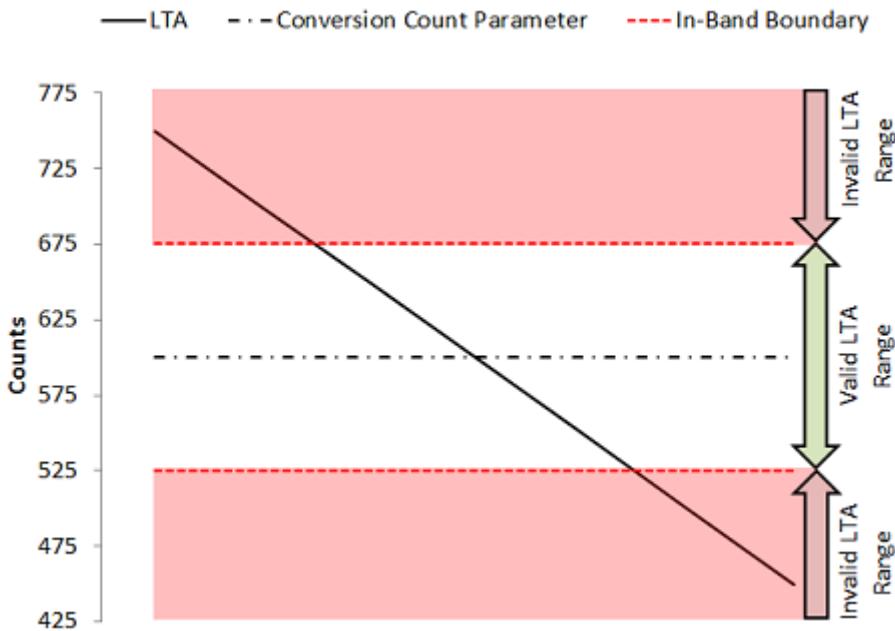
The runtime re-calibration feature provides a runtime check of each element in the sensor to ensure that it is within the resolution and sensitivity band that was specified. The conversion count and conversion gain parameters (in the *Conversion Control* section) specify the desired measurement resolution and sensitivity for the sensor. At device power-up, each element in the sensor is calibrated according to these values. However, environmental drift due to temperature, voltage, humidity, and other factors may cause elements within the sensor to slowly drift from their initial calibration target over time.

Run-time re-calibration, if enabled, checks each element after every measurement to ensure that the long term average (LTA) is within range of the specified conversion count. If the LTA happens to drift out of range, the sensor will be re-calibrated to bring the LTA back in line with the specified conversion count. The valid range is +/- 1/8th of the conversion count, as shown in the equation below.

$$LTA_{\text{ValidRange}} = \text{ConversionCount} \left( 1 \pm \frac{1}{8} \right)$$

The plot below shows the valid and invalid ranges for the LTA if the conversion count is specified as

1. If the LTA tracks into an invalid range, the sensor will be re-calibrated to return the LTA and count to the specified conversion count.



#### Implementation Notes

For most applications, runtime re-calibration should be enabled to ensure that the sensor is always being measured in the desired sensitivity band.

#### Affected Software Parameters

The Runtime\_Recalibration\_Enable parameter corresponds to the **bReCalibrateEnable** member of the **tSensor** type in the CapTlve Touch Library.

## 13.31 Sample Capacitor Discharge

### Sample Capacitor Discharge

The sample capacitor discharge parameter controls whether the sample capacitor is discharged to 0V (empty) or 0.5V (half full). A measurement, or "conversion," is achieved by charging the external capacitor being measured and transferring that charge into an internal sample capacitor multiple times until the internal sample capacitor is full. Having the sample capacitor on-chip reduces the overall system cost. However, on-chip capacitors are not linear when close to empty. By filling the sample capacitor from half full to full, rather than empty to full, a more stable measurement is achievable. The trade-off is that the effective size of the sample capacitor is divided in half. In the case of a very large electrode with a large amount of parasitic capacitance, it is possible to configure the sample capacitor to discharge to 0V rather than 0.5V to increase the sensing range of the system.

#### Implementation Notes

Sample capacitor discharge should be set to 0.5V (half full) for the majority of applications. Only when the extra range is needed for handling large external capacitors should the sample capacitor be discharged fully to 0V.

#### Affected Software Parameters

The Sample\_Capacitor\_Discharge parameter corresponds to the **bCsDischarge** member of the **tSensor** type in the CapTlve Touch Library.

## 13.32 Self Capacitance

Self capacitance refers to capacitance that is created between a single CapTlve™ I/O pin and any neighboring ground, such as circuit and earth grounds.

---

## 13.33 Sensor Port

## 13.34 Sensor Timeout Threshold

### Sensor Time-Out Threshold

The sensor time-out threshold specifies a maximum amount of time a sensor may be in a touch or proximity state before it is reset. The threshold is set in units of samples. The threshold can be related to a time-out period in seconds by using the formula below, where *Report Rate* represents the measurement frequency of the sensor (in Hz).

$$\text{timeout}_{\text{sec}} = \frac{\text{timeout}_{\text{thresh}}}{\text{ReportRate}}$$

The sensor time-out feature is useful for restoring the system in the event that a sensor gets stuck in proximity or touch detect. This may happen for a variety of reasons.

Here are a few examples:

1. A proximity sensor is included in a wall-mounted thermostat to detect when someone is approaching the thermostat touch panel. Someone decides to hang a picture above the thermostat on the wall, increasing the capacitance and causing the proximity sensor to get stuck in detect. If the sensor time-out feature was applied, after a period of time the proximity sensor would reset, clearing the proximity state and resetting the long-term-average to the new environment with the picture on the wall.
1. A phone keypad utilized capacitive touch buttons. A user sets a stack of paper on top of all the buttons, causing them to go into a touch state. After the time-out period expires, the touch state is cleared. When the papers are removed, the long-term-average quickly adjusts to the new value, since it appears as a change against the direction of interest (a negative touch).

### Implementation Notes

The best time-out threshold depends on the application. Buttons typically are not pressed for large lengths of time, and the time-out periods can be shorter. Slider and wheel controls may be in a touch or proximity state for an extended period of time by design, and as such they may require longer time-out periods.

If you are designing a system and the counts being read back are jumping unexpectedly when they are touched for a long period of time, be sure to check the sensor time-out value to ensure it is not too short. Note that the system report rate effects the actual length of time in seconds necessary for a time-out to be issued.

### Range of Valid Values for the Sensor Timeout Threshold Parameter

The sensor timeout threshold may be set from 0 to 65534 samples. Note that specifying a value of zero would result in a sensor that times out every sample, which is not a useful configuration. Setting a value of 65535 disables the timeout feature altogether.

### Affected Software Parameters

The Sensor\_Timeout\_Threshold parameter corresponds to the ***ui16TimeoutThreshold*** member of the ***tSensor*** type in the CapTlve Touch Library.

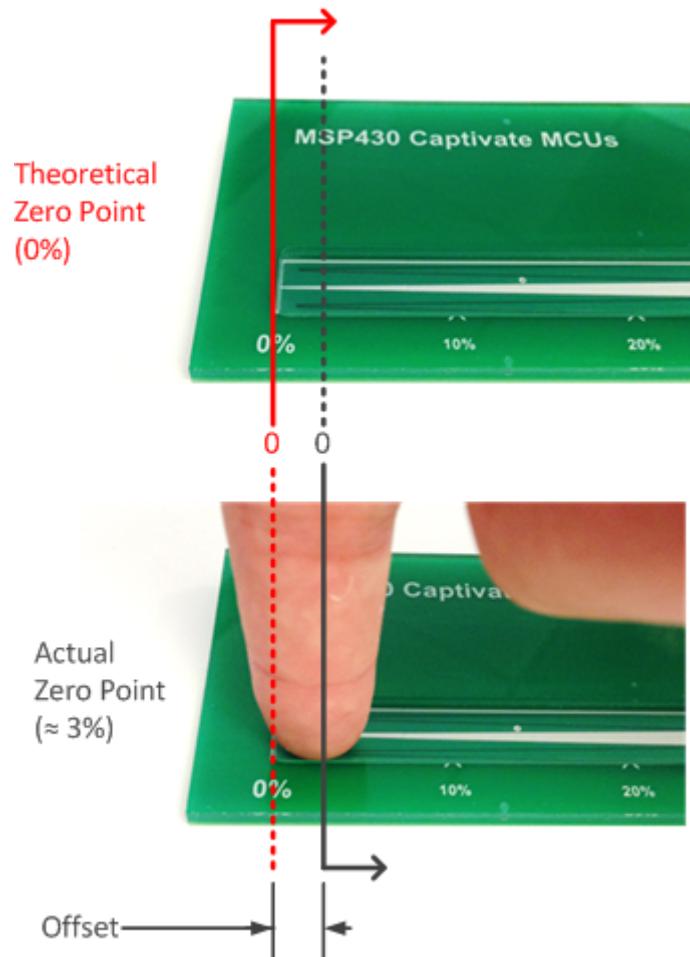
---

## 13.35 Slider Sensor

## 13.36 Slider Trim

### Slider Trim

The slider trim parameter allows for the end points of a slider to be "pulled in," which has the effect compressing the slider. The desired operation of a slider is to have the left-most or top-most point be represented by position lowest possible value (0), and the right-most or bottom-most point be represented by the maximum possible value (the resolution of the slider - 1). Most slider layouts do not allow for this, since the centroid of a user's finger does not typically line up with the exact endpoint of the slider. This concept is illustrated in the example slider below.



This slider has a guide channel in the overlay material that limits the area of interaction with the sensor to the surface area of the electrodes. Notice that while zero is geometrically defined as the end of the slider, the user's finger does not ever touch directly on this point, since the size of the finger provides an offset that is roughly the radius of the finger. In addition, the algorithm treats the slider as an imaginary circle (like a wheel) where the endpoints are next to each other. Since the other end of the slider (the far end) is not near the close end being touched, there is no neighboring electrode to pull the position towards zero.

The slider upper and lower trim parameters provide a way to compensate for both of these factors by scaling the slider to match the desired interaction area.

#### Implementation Notes

To tune these parameters, touch a slider on the left-most (or top-most) side, and if the slider position output is not zero set the Lower\_Trim parameter to the value observed. Then touch the slider on the right-most (or bottom-most) side, and if the slider position is not equal to the slider's resolution - 1, set the Upper\_Trim parameter to the value observed. Then review the slider again. It may take a few iterations to obtain the desired performance.

#### Range of Valid Values for Slider Trim Parameters

Slider trim values must be between 0 and Desired\_Resolution-1. In addition, the lower trim must be less than the upper trim. To disable slider trim, set the lower trim to 0 and the upper trim to the Desired\_Resolution.

#### Affected Software Parameters

The Lower\_Trim parameter corresponds to the ***ui16SliderLower*** member of the ***tSliderSensorParams*** type in the CapTlve Touch Library.

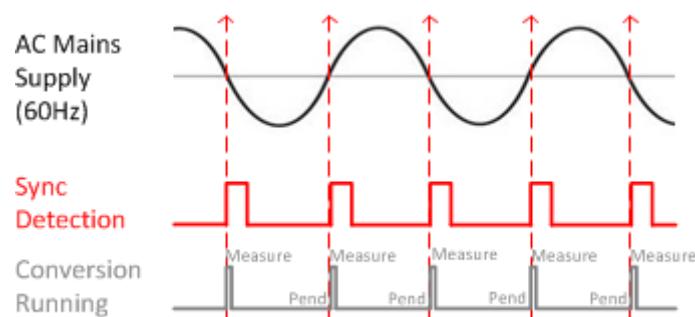
The Upper\_Trim parameter corresponds to the ***ui16SliderUpper*** member of the ***tSliderSensorParams*** type in the CapTlve Touch Library.

## 13.37 Sync Parameters

### Sync Parameters (Input and Timer)

The CapTlve peripheral provides a mechanism for pending the start of a conversion on a hardware event. Two different hardware events are available: an external edge on a digital input pin, or an internal timer trigger.

The first option, an external edge trigger, might be used for a variety of reasons. For example, a measurement could be delayed to be triggered exactly on a 60 Hz zero crossing detection in a noisy environment. Or, multiple CapTlve MCUs could be synchronized to a clock edge to begin measurement at the exact same time, minimizing ground loading on a large panel. The measurement may be triggered on either a rising edge or a falling edge by setting the Input\_Sync parameter.



The second option, an internal timer trigger, exists to provide wake-on-touch capability by allowing the CapTlve internal timer to be used to start measurements at periodic intervals without CPU intervention. The timer triggered conversion is enabled by toggling on the Timer\_Sync parameter.

#### Implementation Notes

Pending on an input sync means that each time cycle in a sensor will pend on its own individual sync event. For example, a sensor with 3 time cycles would take 3 sync edges to complete measurement. Be aware that if the sync event is slow (60 Hz, for example) the delay associated with waiting for the sync can become noticeable and effect the maximum achievable report rate.

#### Affected Software Parameters

The Input\_Sync parameter corresponds to the ***ui8InputSyncControl*** member of the ***tSensor*** type in the CapTlve Touch Library.

The Timer\_Sync parameter corresponds to the ***bTimerSyncControl*** member of the ***tSensor*** type in the CapTlve Touch Library.

## 13.38 System Report Rate

### Active Mode Scan Rate (in milliseconds)

The active mode scan rate specifies the period (in milliseconds) to refresh the user interface at when in active mode (as opposed to autonomous mode). To convert to samples per second (SPS), simply take 1000 divided by the specified report rate period. For example, a report rate (response time) of 50ms would equate to 20 samples per second (SPS).

#### Implementation Notes

This parameter specifies the desired period. If a period is specified that is shorter than the amount of time required to measure and process the panel, then the system will simply run at the fastest rate possible.

---

The period described here is closely tied to the overall response time of the system to a touch or proximity. Shorter values for Active\_Mode\_Scan\_rate\_ms lead to more frequent scanning of the user interface. This is the negative effect of increasing power consumption, but the net positive effect of increasing count filter and de-bounce performance by allowing the filter coefficients and de-bounce thresholds to be increased without sacrificing the response of the system to a touch.

#### Affected Software Parameters

The Active\_Mode\_Scan\_rate\_ms parameter corresponds to the **ui16ActiveModeScanPeriod** member of the **tCaptivateApplication** type in the CapTlve Touch Library.

## 13.39 Time Cycle

Time cycles refer to the measurement cycles performed by the CapTlve™ peripheral. Depending the number and type of sensors used, up to four electrodes can be measured in a single time cycle. For instance, a 12 button mutual capacitive key pad using three TX and four RX channels can be measured in only three time cycles.

## 13.40 Time Estimation

### Scan Time Estimator

The scan time estimator calculates the nominal conversion time for each sensor in the system. This provides insight into how long it takes to measure each sensor individually, as well as the user interface as a whole.

#### Using the Scan Time Estimator

The conversion time for a sensor may be used as a reference when optimizing a design for lowest power consumption. As shown in the formula below, the time it takes to run the conversion is a function of how the conversion control parameters are set up for each sensor. Decreasing the conversion count value decreases the conversion time, as does increasing the conversion clock frequency by adjusting the frequency divider and phase lengths. The conversion time has a direct impact on the amount of power used during the conversion (a longer conversion time implies a higher power consumption).

Knowing the overall conversion time for the user interface can be helpful in situations when the system report rate needs to be optimized. By comparing the *Scan Period* line on the plot to the *Sensors Total* line, it's possible to obtain an estimate of the amount of time available after all the sensors in the system are measured. As the System Report Rate parameter is decreased, the user interface will be measured more often, and the amount of available time will decrease. Note that the more often the user interface is measured, the higher the power consumption will be.

### Limitations

- The estimator only calculates the conversion time itself. It does not take into account the time needed by the software library to process the measurement and call the user callback. This processing time may become significant as the report rate goes up (this is equivalent to the scan period going down).
- The estimator does not take into account the amount of time needed to transmit the status of the user interface via the selected communication interface (UART or I2C). This timing is application dependent.
- The conversion time of a self capacitance sensor goes down when that sensor is interacted with. This is not accounted for scan time estimator.
- The conversion time of a mutual capacitance sensor goes up when that sensor is interacted with. This is not accounted for in the scan time estimator.

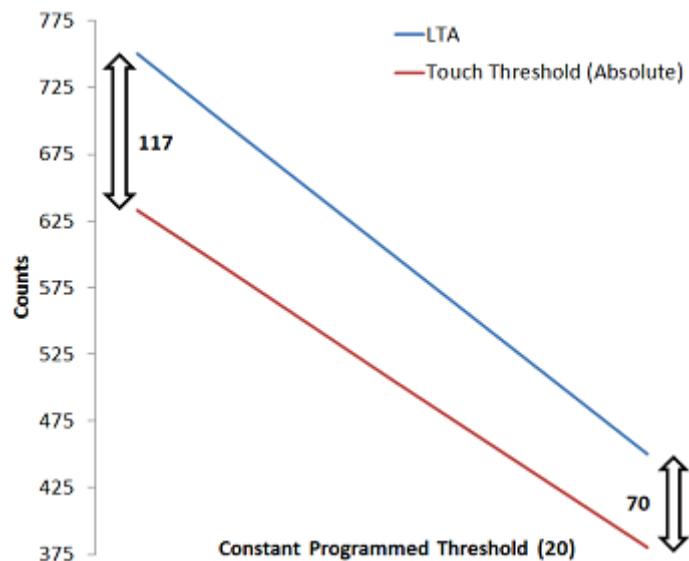
## 13.41 Touch Threshold

### Touch Threshold

The touch threshold sets the level of interaction required by the user for a touch detection to be declared for an element. Unlike the proximity threshold, the effective touch threshold is dynamic. It is set as a deviation from the long term average (LTA), in units of 1/128 of the LTA. By defining the touch threshold unit as a percentage of the LTA, it is possible to maintain a consistent sensitivity even if the LTA drifts due to a changing environment. The absolute touch threshold for a given sample is calculated at runtime, per the equation below:

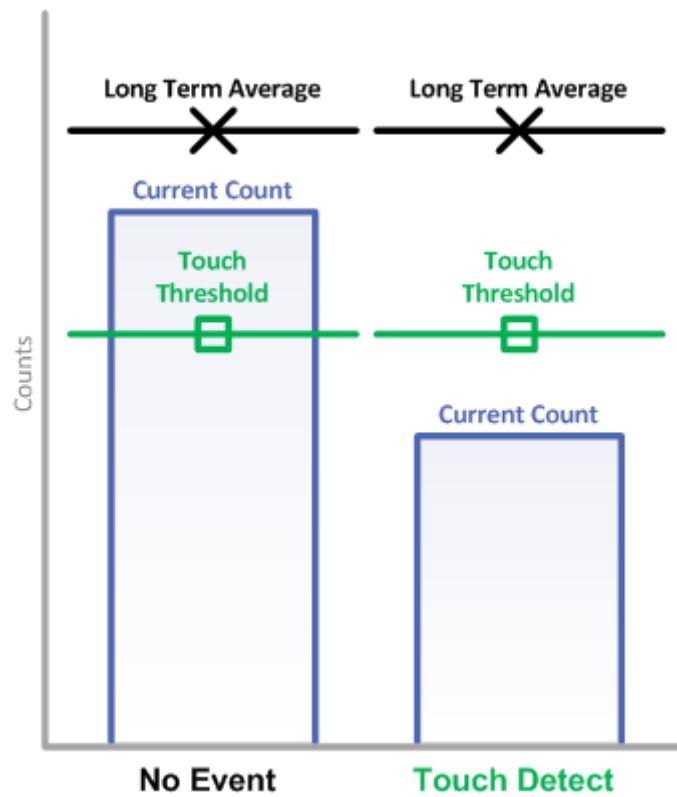
$$\begin{aligned} \text{TouchThresh}_{\text{absolute}} &= \\ &= \begin{cases} \text{Self: } \text{LTA} - \left( \frac{\text{LTA}}{128} \text{TouchThresh}_{\text{programmed}} \right) \\ \text{Mutual: } \text{LTA} + \left( \frac{\text{LTA}}{128} \text{TouchThresh}_{\text{programmed}} \right) \end{cases} \end{aligned}$$

For example, if an element has an LTA of 600, and a programmed touch threshold of 10, the effective delta required to enter a touch state is  $(600 * 10) / 128$ , or 46. The diagram below illustrates how the touch threshold is reduced as the LTA is reduced, if the threshold is held at a constant of 20.



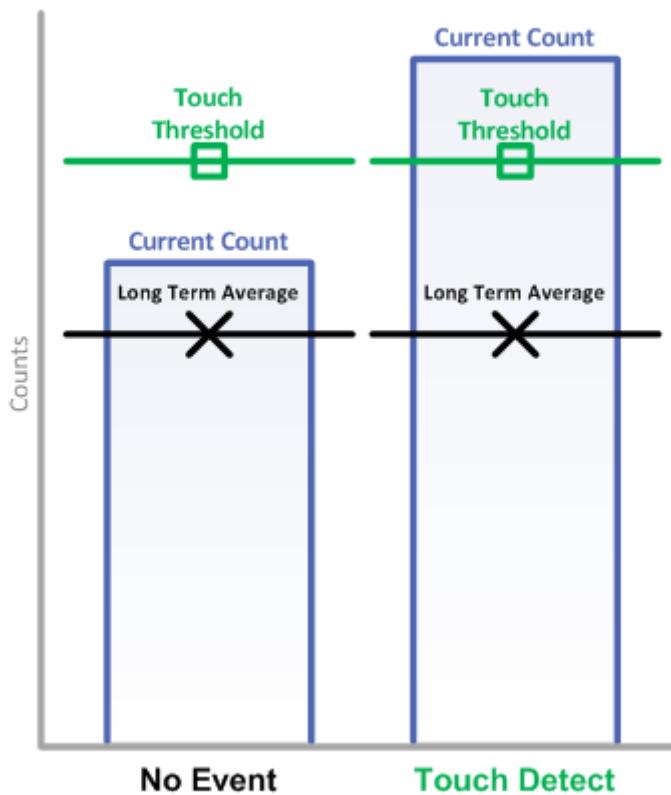
### Self Capacitance Example

When measuring an element in self-capacitance mode, a user's touch on the element's electrode causes the capacitance of that electrode to increase. This causes a decrease in the measurement result. When a decrease in the measurement surpasses the touch threshold, the touch flag is set.



### Mutual Capacitance Example

In mutual-capacitance mode, a user's touch on an element's electrode causes the mutual capacitance between the Rx and Tx to decrease. This causes an increase in the measurement result. When an increase in the measurement surpasses the touch threshold, the touch flag is set.



### Implementation Notes

Touch thresholds are specified individually for each element in a sensor. Each element has an individual touch detect flag that gets set whenever there is a touch detect for that element. Note that the touch detect flag is also dependent upon the de-bounce process. If de-bounce is used, the touch status flag will not be set immediately after the threshold crossing. A global touch flag for a sensor is also provided. A touch detect flag on any element will cause the sensor's global touch detect flag to be set.

### Range of Valid Values for the Touch Threshold Parameter

The proximity threshold may be set from 1 to 255.

### Affected Software Parameters

The Touch\_Threshold parameter corresponds to the ***ui8TouchThreshold*** member of the ***tElement*** type in the CapTlve Touch Library.

If a touch detect occurs at runtime, the ***bSensorTouch*** member of the related ***tSensor*** instance will be set to true, alerting the application that an element on that sensor entered a touch state.

In addition, the ***bTouch*** member of the related ***tElement*** instance will also be set to true, allowing the application to determine which specific element(s) entered into a touch state.

## 13.42 Trace

A trace is the conductive connection between the MSP430™ microcontroller and the electrode. Similar to the electrode, the trace is typically a copper trace on a PCB, but it could also be made of materials like ITO and silver. Connectors and cables between the microcontroller and electrode also affect performance and are described along with trace routing in Section 5. Capacitance is the ability of the electrode to store an electrical charge. In the context of capacitive touch detection, there are two common categories of capacitance: mutual capacitance and self capacitance. As the names imply, self capacitance refers to the capacitance of one electrode, while mutual

---

capacitance refers to the capacitance between two electrodes. Self capacitance is the topic of this document, and the concepts described here pertain primarily to self-capacitance solutions. An important concept within capacitive touch detection is baseline capacitance. This represents the steady-state no-interaction capacitance seen by the microcontroller. The baseline capacitance is the sum of the parasitic capacitances, which include the electrode, trace, and parasitic capacitances associated with the MSP430 pins, solder pads, and any discrete components associated with the circuit, for example, ESD current-limiting resistors. The baseline capacitance is important because sensitivity is a function of the relative change in capacitance. If the baseline capacitance is too large, then any change in capacitance caused by a touch or proximity event is very small and might not be distinguishable from the baseline.

### 13.43 Wake On Touch

### 13.44 Wheel Sensor

A wheel sensor is essentially a slider sensor with the two ends wrapped together.

## Chapter 14

### Disclaimer

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

---

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

You can obtain information on other Texas Instruments products and application solutions from [www.ti.com](http://www.ti.com).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Copyright © 2015-2016, Texas Instruments Incorporated