MSP430™ microcontrollers offer a number of peripherals that, when configured properly, can be used to perform a capacitance measurement. The purpose of the capacitive touch library is to create a single interface that can be integrated with the peripheral sets found in the Value Line, 2xx, 5xx, FR4xx, and FR59xx product families. This document explains the capacitive touch library’s configuration and use.

The software library described in this document can be downloaded from http://www.ti.com/tool/capsenselibrary.

Contents

1 Introduction ................................................................................................................... 4
2 Methods ....................................................................................................................... 5
3 Configuration ............................................................................................................... 8
4 Resources ................................................................................................................... 41
5 API Calls ................................................................................................................... 43
6 Establishing Measurement Parameters ........................................................................ 50

Appendix A  CTS_Layer.c and CTS_Layer.h Detailed Description ........................................ 55

List of Figures

1 Relaxation Oscillator Measurement ............................................................................. 5
2 Resistor-Capacitor Time Constant Measurement (RC) .................................................. 6
3 RC Schematic .............................................................................................................. 7
4 Fast Scan RO Measurement ......................................................................................... 7
5 Elements ..................................................................................................................... 9
6 Element Measurement Parameters: Buttons Example .................................................. 10
7 CBCTL0 Register ........................................................................................................ 11
8 RO_PinOsc Schematic .................................................................................................. 11
9 PinOsc Port Parameters ............................................................................................... 12
10 Capacitive Touch I/O Implementation ......................................................................... 12
11 CAPSIOxCTL Register ............................................................................................... 13
12 RC IO Parameters ....................................................................................................... 14
13 Sensor Example ......................................................................................................... 15
14 RO_COMPB Port Parameters ....................................................................................... 16
15 Port Parameters for RO_COMPAp_TAx Implementation ............................................. 17
16 fRO_COMPAp_SW_TAx General Description ............................................................ 18
17 RO_COMPAp_TAx Schematic Description ................................................................ 20
18 Timing Parameters: Example WDTp ......................................................................... 22
19 RO_COMPB Schematic .............................................................................................. 23
20 RO_PinOsc_TA0 Timing Parameter ............................................................................. 28
21 RC Timing Parameters ............................................................................................... 32
22 fRO_CSIQ_TA2_TA3 Timing Parameters ................................................................... 33
23 fRO_PinOsc_TA0_TA1 Timing Parameters ................................................................ 34
24 fRO_PinOsc_TA0 Timing Parameters ....................................................................... 35
25 fRO_COMPB_TA1_TA0 Timing Parameters ................................................................. 36
<table>
<thead>
<tr>
<th>No.</th>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Status/Baseline Control Register</td>
<td>55</td>
</tr>
<tr>
<td>27</td>
<td>Tracking Settings Against Direction of Interest</td>
<td>57</td>
</tr>
<tr>
<td>28</td>
<td>Tracking Settings in Direction of Interest</td>
<td>57</td>
</tr>
<tr>
<td>29</td>
<td>Example HAL Definitions</td>
<td>59</td>
</tr>
</tbody>
</table>
1 Introduction

The Capacitive Touch Software Library provides code that performs capacitance measurements on the MSP430 microcontroller platform using several combinations and permutations of peripherals. To simplify the use of capacitive touch sensing on the MSP430, the library provides a configuration structure that is specific to the implementation and API calls that are specific to the application.

The main purpose of this document is to explain the API calls and the configuration of the library. However, it is also important to have a basic understanding of the measurement implementations and how peripheral resources are used before using the library.

Section 2 describes the relaxation oscillator (RO), time constant (RC), and fast RO methods with the MSP430.

Section 3 describes the library configuration.

Section 4 describes peripheral resources.

Section 5 describes the APIs and the levels or degrees of abstraction with those APIs. The higher levels of abstraction provide standard controls for faster and easier development while the lower levels allow for customization and unique controls.

The associated code (1) is intended to be a starting point for developing capacitive touch and other capacitive measurement solutions. The library supports a wide variety of features and functions, all of which may not be required for a specific application. The source code is provided, and customers are encouraged to remove sections of code that are not used after creating a working application. Additionally, interrupt service routines (ISRs) are part of the source code, which may need additional code to support other applications. This is a common adjustment needed with WDT ISR. Again, as applications allow for shared ISR functionality, customers are encouraged to update the source code provided to support capacitive touch functionality in addition to other tasks.

(1) The software library described in this document can be downloaded from http://www.ti.com/tool/capsenselibrary.

<table>
<thead>
<tr>
<th>Devices</th>
<th>HALs</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2xx2</td>
<td>RO_PINOSC_TA0_WDTp</td>
<td>Section 3.2.2.3.1</td>
</tr>
<tr>
<td>G2xx3</td>
<td>RO_PINOSC_TA0_WDTp</td>
<td>Section 3.2.2.3.1</td>
</tr>
<tr>
<td>G2xx5</td>
<td>RO_PINOSC_TA0_WDTp</td>
<td>Section 3.2.2.3.1</td>
</tr>
<tr>
<td>G2xx5</td>
<td>RO_PINOSC_TA1_WDTp</td>
<td>Section 3.2.2.3.1</td>
</tr>
<tr>
<td>G2xx5</td>
<td>RO_PINOSC_TA1_WDTp</td>
<td>Section 3.2.2.3.1</td>
</tr>
<tr>
<td>F51xx</td>
<td>RO_COMPB_TA0_WDTA</td>
<td>Section 3.2.2.2.1</td>
</tr>
<tr>
<td>F52xx</td>
<td>RO_COMPB_TA1_WDTA</td>
<td>Section 3.2.2.2.1</td>
</tr>
<tr>
<td>F52xx</td>
<td>RO_COMPB_TA1_WDTA</td>
<td>Section 3.2.2.2.1</td>
</tr>
<tr>
<td>FR4xx</td>
<td>RO_CSIO_TA0_TA1</td>
<td>Section 3.2.2.5.1</td>
</tr>
<tr>
<td>FR55xx</td>
<td>RO_CSIO_TA0_RTC</td>
<td>Section 3.2.2.5.1</td>
</tr>
<tr>
<td>FR55xx</td>
<td>RO_CSIO_TA0_RTC</td>
<td>Section 3.2.2.5.1</td>
</tr>
<tr>
<td>FR58xx and FR59xx</td>
<td>RO_CSIO_TA2_WDTA</td>
<td>Section 3.2.2.5.1</td>
</tr>
<tr>
<td>FR58xx and FR59xx</td>
<td>RO_CSIO_TA2_WDTA</td>
<td>Section 3.2.2.5.1</td>
</tr>
</tbody>
</table>

MSP430 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.
2 Methods

For the methods described in this document, the fundamental principle is that two independent time bases are compared. One time base is fixed and the other is variable as a function of the capacitance. As long as the capacitance does not change, the relationship between the two remains constant. Timers are used to measure the two time bases, and the relationship is monitored in software. When the relationship changes, the software must decide if the change was an increase or decrease in the capacitance and if the change was significant enough to be considered a 'touch'.

The three methods discussed in this document are the relaxation oscillator (RO), the resistor capacitor (RC), and the fast relaxation oscillator (fRO).

2.1 Relaxation Oscillator (RO)

The RO method counts the number of relaxation oscillator cycles within a fixed period (gate time), as shown in Figure 1. The variable time base, TimeBase 1, is connected to a relaxation oscillator circuit. The relaxation oscillator can be realized with a comparator, the PinOsc feature found in the Value Line product family, or the Capacitive Touch I/O found in the MSP430FR58xx and MSP430FR59xx family. The fixed time base, TimeBase 2, is connected to an internal MSP430 system clock such as the DCO or REFO. The gate time is a number of the system clock oscillations.

The frequency of the relaxation oscillator is function of the capacitance of the circuit. As the capacitance increases (C1 to C2), the relationship between the TimeBase 1 and TimeBase 2 changes. The number of relaxation oscillator cycles counted within the gate time decreases. As the capacitance decreases (C2 to C1), the number of cycles within the gate time increases. Capacitance and counts have an inverse relationship in the RO method.

(1) The RO method is described in more detail in PCB-Based Capacitive Touch Sensing With MSP430 (SLAA363).
The naming convention for the RO method in the library identifies the relaxation oscillator mechanism, the timer that is used to measure or count oscillations, and the timer that is used to define the gate period.

### Table 2. Relaxation Oscillator Naming Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>RO Mechanism</th>
<th>Counter</th>
<th>Gate Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO_XXX_YYY_ZZZ</td>
<td>XXX</td>
<td>YYY</td>
<td>ZZZ</td>
</tr>
<tr>
<td>RO_COMPAp_TA0_WDTp</td>
<td>Comparator_A+</td>
<td>TimerA0</td>
<td>Watchdog timer (interval mode)</td>
</tr>
<tr>
<td>RO_Pinosc_TA0</td>
<td>Pin Oscillator</td>
<td>TimerA0</td>
<td>'n' ACLK periods(^{(1)})</td>
</tr>
<tr>
<td>RO_CSIO_TA2_TA3</td>
<td>Capacitive-Touch</td>
<td>TimerA2</td>
<td>TimerA3</td>
</tr>
<tr>
<td></td>
<td>Enabled IO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^{(1)}\) The RO_PINOSC_TA0 is a special case that takes advantage of the internal connection between ACLK and the TimerA0 capture input. The user has the choice of simply dividing the ACLK in the application layer (by 2, 4, 8, and setting \(n\) to 1) or by entering a number of ACLK cycles, or both.

### 2.2 Resistor-Capacitor Time Constant Measurement (RC)\(^{(1)}\)

The RC method is the reciprocal of the RO method. As shown in Figure 2, the gate time is now variable, and the oscillator being counted is fixed. The fixed time base, TimeBase 1, is connected to an internal MSP430 oscillator, such as the DCO. The variable time base, TimeBase 2, is connected to a capacitor and resistor and the time it takes to charge and discharge the capacitor a number of times through the resistor is now the gate time. The RC method can be realized with any MSP430.

**Figure 2. Resistor-Capacitor Time Constant Measurement (RC)**

The gate time is a function of the capacitance of the circuit. As the capacitance increases (C1 to C2), the relationship between the two time bases changes. The number of system clock cycles counted within the gate time increases. As the capacitance decreases (C2 to C1), the number of cycles within the gate time decreases. For the RC method the relationship between capacitance and counts is direct.

This method uses a single timing resource (TIMER_A0) to measure the time it takes the capacitance to charge and discharge 'n' times. To measure both charge and discharge, two IO are needed, as shown in Figure 3. This is reflected in the function name within the library: RC_PAIR.

\(^{(1)}\) The RO method is described in more detail in *PCB-Based Capacitive Touch Sensing With MSP430* (SLAA363).
2.3 Fast Scan Relaxation Oscillator (fRO)

The fRO method is similar to the RC method except that the variable gate period is created with a relaxation oscillator instead of the charge and discharge time. And as shown in Figure 4, the fixed time base, TimeBase 1, is connected to an internal MSP430 oscillator, such as the DCO. The variable time base, TimeBase 2, is connected to the relaxation oscillator.

The gate time is a function of the capacitance of the circuit. As the capacitance increases (C1 to C2), the relationship between TimeBase 1 and TimeBase 2 changes. The number of system clock cycles counted within the gate time increases. As the capacitance decreases (C2 to C1), the number of cycles within the gate time decreases. For the fRO method the relationship between capacitance and counts is direct just like the RC method. The fast relaxation oscillator (fRO) method is intended to bridge the gap between the RC and RO methods. The fRO method provides the fast scan rates of the RC method and the improved sensitivity of the RO method.

The naming convention for the fRO method in the library identifies the relaxation oscillator mechanism, the timer used to define the gate period (as a function of the variable relaxation oscillator frequency), and the timer used to count the fixed oscillation frequency.

<table>
<thead>
<tr>
<th>Name</th>
<th>RO Mechanism</th>
<th>Gate Period</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>fRO_XXX_YYY_ZZZ</td>
<td>XXX</td>
<td>YYY</td>
<td>ZZZ</td>
</tr>
<tr>
<td>fRO_COMPB_TA1_SW</td>
<td>Comparator B</td>
<td>TimerA1</td>
<td>Software Loop</td>
</tr>
<tr>
<td>fRO_PINOSC_TA0_TA1</td>
<td>Pin Oscillator</td>
<td>TimerA0</td>
<td>TimerA1</td>
</tr>
</tbody>
</table>
As the name implies, the purpose of the fRO method is to provide fast scan rates; faster than the RO method with similar sensitivity (1). With the RO method, sensitivity is a function of the gate time. Increasing the gate time increases sensitivity. The negative consequence of increasing gate time is decreased scan rates: more time is spent during a single measurement. The sensitivity of the fRO method can also be improved with increased gate time, however, the sensitivity can also be improved by improving the system clock resolution (higher speed).

In comparison to the RO method the fRO method provides similar sensitivity (change in counts) in a shorter gate time. The theoretical exercise found in shows that similar sensitivity can be achieved with the fRO method in less time. (1) In this context, sensitivity is a measure of how small a change in capacitance can be resolved.

### Table 4. Comparison of fRO and RO Measurement Times

<table>
<thead>
<tr>
<th></th>
<th>RO_PinOsc_TA0_WDTp</th>
<th>fRO_PinOsc_TAx_TA1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gate Time</strong></td>
<td>(1-MHz SMCLK, WDT, 512)</td>
<td>(45 RO cycles)</td>
</tr>
<tr>
<td><strong>Counter</strong></td>
<td>(RO)</td>
<td>(SMCLK)</td>
</tr>
<tr>
<td><strong>Counts</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Touched</td>
<td>512 µs</td>
<td>45/1 MHz = 45 µs</td>
</tr>
<tr>
<td>Untouched</td>
<td>512 µs</td>
<td>45/1.1 MHz = 41 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>51</td>
<td>48</td>
</tr>
</tbody>
</table>

#### 3 Configuration

There are two main files that serve as the means to configure the library: structure.c and structure.h. structure.c includes definitions of the elements and the sensors (groups of elements). The structure.h file makes the definitions in structure.c visible to the other portions of the library and also uses precompiler definitions to enable functions and limit code size.

#### 3.1 Element Definition

A capacitive measurement element is a singular structure, whose capacitance represents an event; for example, a touch, a change in humidity, or a change in dielectric. An element can be used individually (for example as a button) or combined with other elements to create a keypad, wheel, or slider as shown in Figure 5.
The element definitions belong to one of two categories; port definition or measurement parameter. The ports definitions include digital I/O peripheral registers, comparator peripheral registers, and bit definitions. The measurement parameters include the threshold and maximum signal response (maxResponse) of an element. The port definition can be completed by simply reading the schematic while the measurement parameters require testing (see Section 6). Establishing the correct measurement parameters calibrates the elements.

The element definition found in the configuration file structure.c uses a designated initializer list. This allows members to be initialized in any order and also enhances the readability of the element being initialized. This feature requires the GCC language extension found in Code Composer Studio (CCS). C99 is the default dialect found in IAR and, therefore, the default settings can be used.

3.1.1 Common Element Variables

InputBits is one common definition that can represent the bit y in the GPIO definition Px.y, the comparator input mux for either COMP_A+ or COMP_B solutions, or the capacitive touch IO port and bit selection in the capacitive touch IO control register.

threshold defines the limit or threshold that the change in capacitance must exceed before an event (typically a touch) is declared.

maxResponse is the maximum response expected from an element within a sensor and only used in sensors with multiple elements: slider, wheel, and buttons (1). The purpose of the maxResponse parameter is to normalize the capacitive measurement to a percentage, where the threshold represents 0% and the maxResponse represents 100%. This percentage is used to identify the dominant element within the sensor if multiple elements have threshold crossings.

(1) The buttons abstraction is a sensor made from two or more elements. The button abstraction is a sensor made from one element.
Figure 6 shows the relative relationship between threshold and maxResponse in a buttons application. First, the maxResponse value is only needed for the buttons, wheel, and slider APIs. As mentioned in the definition the maxResponse is used to determine the dominant element. Second, the maxResponse represent a heavy touch where the finger flattens, by pressing heavily, to create a larger surface area. By contrast a light or normal touch creates a much smaller surface area. The threshold value represents a change in capacitance that is smaller than even the lightest touch but still significantly larger than any noise that might appear as a change in capacitance.

The threshold and maxResponse variables are limited to unsigned 16-bit integers (0 to 65535). These values are further limited by the following when a multi-element abstraction is used (buttons, slider, or wheel): maxResponse – threshold < 655. Section 6 provides more detail on establishing the measurement parameters.

3.1.1.1 Comparator_A+ (COMP_A+)

Implementations using the COMP_A+ peripheral to create a relaxation oscillator use the same element structure format.

*InputBits* identify the bits, P2CA1, P2CA2, and P2CA3, within the CACTL2 register which represent the negative terminal input of the comparator. This input is connected directly to the electrode. The input for the reference is defined in the sensor section.

```c
Const struct Element element_name = {
  .inputBits = P2CA2, // CA2
  .threshold = 100,
  .maxResponse = 200
};
```
3.1.1.2 Comparator_B (COMP_B)

InputBits identify the CBIMSEL bits in the CBCTL0 register.

Figure 7. CBCTL0 Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CBIMEN</td>
<td>RW</td>
<td>0h</td>
<td>Channel input enable for the V– terminal of the comparator. 0b = Selected analog input channel for V– terminal is disabled. 1b = Selected analog input channel for V– terminal is enabled.</td>
</tr>
<tr>
<td>14-12</td>
<td>Reserved</td>
<td>R</td>
<td>0h</td>
<td>Reserved. Always reads as 0.</td>
</tr>
<tr>
<td>11-8</td>
<td>CBIMSEL</td>
<td>RW</td>
<td>0h</td>
<td>Channel input selected for the V– terminal of the comparator if CBIMEN is set to 1.</td>
</tr>
<tr>
<td>7</td>
<td>CBIPEN</td>
<td>RW</td>
<td>0h</td>
<td>Channel input enable for the V+ terminal of the comparator. 0b = Selected analog input channel for V+ terminal is disabled. 1b = Selected analog input channel for V+ terminal is enabled.</td>
</tr>
<tr>
<td>6-4</td>
<td>Reserved</td>
<td>R</td>
<td>0h</td>
<td>Reserved. Always reads as 0.</td>
</tr>
<tr>
<td>3-0</td>
<td>CBIPSEL</td>
<td>RW</td>
<td>0h</td>
<td>Channel input selected for the V+ terminal of the comparator if CBIPEN is set to 1.</td>
</tr>
</tbody>
</table>

```
const struct Element element_name = {
  .inputBits = CBIMSEL_2, // CB2
  .threshold = 100,
  .maxResponse = 200
};
```

3.1.1.3 Pin Oscillator (PinOsc)

The Pin Oscillator (PinOsc) implementation of the relaxation oscillator replaces the comparator and reference circuitry with the Schmitt trigger input found in the digital I/O and an internal inverter. The PinOsc feedback path to the RC filter is accomplished with the integrated resistor, as shown in Figure 8.
*inputPxselRegister and *inputPxsel2Register identify the appropriate registers that need to be configured for the pin oscillator method. These registers in conjunction with inputBits configure the element as shown in Figure 9 and the following code snippet.

```c
const struct Element right = {
    .inputPxselRegister = (uint8_t *)&P2SEL,
    .inputPxsel2Register = (uint8_t *)&P2SEL2,
    .inputBits = BIT3,
    .maxResponse = 400,
    .threshold = 50
};
```

### 3.1.1.4 Capacitive Touch I/O (CSIO)

The capacitive touch IO (CSIO) implementation of the relaxation oscillator is the same as the PinOsc; however, the interface for controlling the digital I/O is different.

```c
P.x.y
element0->inputBits = CAPSIPOSEL_x + CAPSIPISEL_y
```

**Figure 10. Capacitive Touch I/O Implementation**

The relaxation oscillator is accomplished with internal circuitry and the port parameter only includes the inputBits which represent the capacitive touch IO port and pin select bits within the control register (CAPSIOxCTL). The port and pin select bits are defined at the element level while the control register is defined at the sensor level: inputCapsiOctlRegister.
**InputBits** identify the port and pin select bits in the CAPSIOxCTL register.

![Figure 11. CAPSIOxCTL Register](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>Reserved</td>
<td>R</td>
<td>0h</td>
<td>Reserved. Always reads 0.</td>
</tr>
<tr>
<td>9</td>
<td>CAPSIO</td>
<td>R</td>
<td>0h</td>
<td>Capacitive Touch I/O state. Reports the current state of the selected Capacitive Touch I/O. Reads 0, if Capacitive Touch I/O disabled. 0b = Current state 0 or Capacitive Touch I/O is disabled 1b = Current state 1</td>
</tr>
<tr>
<td>8</td>
<td>CAPSIOEN</td>
<td>RW</td>
<td>0h</td>
<td>Capacitive Touch I/O enable 0b = All Capacitive Touch I/Os are disabled. Signal toward timers is 0. 1b = Selected Capacitive Touch I/O is enabled</td>
</tr>
<tr>
<td>7-4</td>
<td>CAPSIOPOSELx</td>
<td>RW</td>
<td>0h</td>
<td>Capacitive Touch I/O port select. Selects port Px. Selecting a port pin that is not available on the device in use gives unpredictable results. 0000b = Px = PJ 0001b = Px = P1 0010b = Px = P2 ... 1110b = Px = P14 1111b = Px = P15</td>
</tr>
<tr>
<td>3-1</td>
<td>CAPSIOPISELx</td>
<td>RW</td>
<td>0h</td>
<td>Capacitive Touch I/O pin select. Selects the pin within selected port Px (see CAPSIOPOSELx). Selecting a port pin that is not available on the device in use gives unpredictable results. 000b = Px.0 001b = Px.1 010b = Px.2 ... 110b = Px.6 111b = Px.7</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td>R</td>
<td>0h</td>
<td>Reserved. Always reads 0.</td>
</tr>
</tbody>
</table>

```c
const struct Element element_name = {
    .inputBits = CAPSIOPOSEL0+CAPSIOPOSEL1+CAPSIOPISEL2, // P3.4
    .threshold = 100,
    .maxResponse = 200
};
```
3.1.2 Element Variables Specific to RC

The RC method is comprised of two GPIO. One is the input and the other is the reference. The configuration requires the pertinent register addresses for a given port as well as the bit definition.

`inputPxdirRegister`, `inputPxoutRegister`, and `inputPxinRegister` identify the port direction, output, and input addresses. These registers in conjunction with `inputBits` configure the input portion of the element definition.

`referencePxdirRegister` and `referencePxoutRegister` identify the port direction and output addresses. These registers in conjunction with `referenceBits` configure the reference portion.

One feature of this description is that one pin can have two different functions. That is the pin can be a reference in one element definition and an input in another element definition. Figure 12 and the following code snippet show the reference and input definitions.

```
Figure 12. RC IO Parameters

//RC P2.0 input, P2.1 Reference
const struct Element element1 = {
    .inputPxinRegister = (uint8_t *)&P2IN,
    .inputPxoutRegister = (uint8_t *)&P2OUT,
    .inputPxdirRegister = (uint8_t *)&P2DIR,
    .inputBits = BIT0,
    .referencePxoutRegister = (uint8_t *)&P2OUT,
    .referencePxdirRegister = (uint8_t *)&P2DIR,
    .referenceBits = BIT1,
    .threshold = 100,
    .maxResponse = 200
};

//RC P2.1 input, P2.0 Reference
const struct Element element2 = {
    .inputPxinRegister = (uint8_t *)&P2IN,
    .inputPxoutRegister = (uint8_t *)&P2OUT,
    .inputPxdirRegister = (uint8_t *)&P2DIR,
    .inputBits = BIT1,
    .referencePxoutRegister = (uint8_t *)&P2OUT,
    .referencePxdirRegister = (uint8_t *)&P2DIR,
    .referenceBits = BIT0,
    .threshold = 120,
    .maxResponse = 250
};
```
3.2 Sensor Definition

The sensor can simply be a single element, a group of independent elements like a keypad, or a group of elements functioning as one sensor like a wheel or slider. The sensor definition includes all of the applicable elements, the implementation that is used to measure the capacitance of all the elements, and the peripheral addresses and bit settings for the given mechanism. In the case of wheels and sliders, the sensor definition also defines the number of points or positions along with the slider, as shown in Figure 13, and the sensitivity of the sensor.

![Figure 13. Sensor Example](image)

This section is divided into three sub-sections: port variables, common variables, and variables specific to sliders and wheels. The port variables are only applicable to the CSIO and comparator implementations of capacitive touch sensing. These implementations require either additional GPIO or register settings to configure the MSP430. The common variables are variables that are common in name but different in meaning dependent upon the method. This sub-section describes through each implementation, providing code examples. The wheels and sliders sub-section describes the variables associated with wheel and slider sensor.

The sensor definition found in the configuration file structure.c uses a designated initializer list. This allows members to be initialized in any order and also enhances the readability of the sensor being initialized. This feature requires the GCC language extension found in Code Composer Studio (CCS). C99 is the default dialect found in IAR and therefore the default settings can be used.

3.2.1 Port Variables

The port variables within the sensor definition are only applicable to the COMP_A+, COMP_B, and CSIO implementations.

3.2.1.1 CSIO Port Variable

The port and pin select bits are defined at the element level while the control register is defined at the sensor level: inputCapsioclRegister.

The inputCapsioclRegister is different from the other parameters found in the library because the association between the CAPSIOxCTL register and timer is device dependent. It is important that the control register selected works with the HAL also indicated within the sensor level definition. For example if CAPSIO0CTL is associated with TimerA2, then the inputCapsioclRegister must point to CAPSIO0CTL when the RO_CSIO_TA2_WDTA HAL definition is used.
3.2.1.2 COMP_B Port Variables

The implementation of the COMP_B solution requires an alternative set of parameters for the sensor port variables, shown in Figure 14.

![Figure 14. RO_COMPB Port Parameters](image)

`cboutTxRegister`, `cboutTxRegister` identify the port direction address and port selection address. The variable `cboutTxBits` define the bits that are to be set or reset within the direction and selection register to select the CBOUTx output and TxCLK input function of the port. Note that on 5xx family devices, these ports share the same IO.

Some MSP430 devices allow the digital I/O to be driven from a different voltage rail (DVIO) than the voltage rail (VCC) that the Comp_B peripheral uses. If the comparator output, CBOUT, uses DVIO and DVIO is different from VCC, then the input high and input low values set within the library need to be updated. These values are set in the Comp_B control register 2, CBCTL2, found in the CTS_HAL.c file.

`cbpdBits` is used to disable the Digital I/O on the port pins also used as the comparator inputs. This is applied to the COMP_B control register CBCTL3. The bit CBPDy, in CBCTL3, disables the port of the comparator channel ‘y’ (that is, CBPDy disables CBy and not Px.y)
3.2.1.3 COMP_A+ Port Variables

In the element structure the comparator input is defined for each element. At the sensor level, as shown in Figure 15, the comparator reference input is defined as well as the reference port, comparator output, and timer input.

**Figure 15. Port Parameters for RO_COMPAp_TAx Implementation**

`caoutDirRegister` and `caoutSelRegister` identify the port direction address and port selection address. The variable `caoutBits` define the bits that are to be set or reset within the direction and selection register to select the CAOUT output function of the port. Some devices also use the PxSEL2 register to define the CAOUT use case. In the case of these devices the value `caoutSel2Register` must also be defined (either as P1SEL2 or P2SEL2).

`txclkDirRegister`, `txclkSelRegister` identify the port direction address and port selection address. The variable `txclkBits` define the bits that are to be set or reset within the direction and selection register to select the TxCLK input function of the port. Some devices also use the PxSEL2 register to define the TxCLK use case. In the case of these devices, the value `txclkSel2Register` must also be defined (either as P1SEL2 or P2SEL2).

The `refPxDirRegister`, `refPxOutRegister`, and `refBits` variables define the pullup portion of the external reference circuit shown in Figure 15. These bits provide the mechanism to turn on and off the reference for power savings. `refPxDirRegister`, `refPxOutRegister` identify the port direction address and port output address. The variable `refBits` define the bits that are to be set or reset within the direction and selection register to enable the reference circuit.

`refCactl2Bits` indicates which positive input of COMP_A+ is connected to the voltage reference. The reference should only be applied to the positive input via CA0, CA1, or CA2. This is represented as P2CA0, P2CA4, and P2CA0+P2CA4, respectively.

`capdBits` is used to define the IO which make up the sensor. This is applied to the COMP_A+ control register CAPD. This value is the logical ‘OR’ of all the bit definitions for each input and the reference input (that is, the ‘y’ of Px.y and NOT the ‘y’ in CAy).

A slight variation of the COMP_A+ implementation is the fRO_COMPAp_SW_TAx implementation which uses a software timer to create the gate time and the timer peripheral as the frequency counter. Because this implementation does not use the timer to count the number relaxations oscillations the physical connection is no longer needed as shown in Figure 16.
The port parameters are the same as those described above with the exception of `txclkDirRegister`, `txclkSelRegister`, and `txclkBits`, which are not used.

### 3.2.2 Common Sensor Variables

`numElements` identifies the number of elements that are within the sensor.

`baseOffset` is a cumulative count of the number of elements that are defined in the application. There is a baseline value stored in RAM for each element.

```c
//IN structure.h FILE
#define TOTAL_NUMBER_OF_ELEMENTS 8

//IN CTS_Layer.c FILE
uint16_t baseCnt[TOTAL_NUMBER_OF_ELEMENTS];
```

<table>
<thead>
<tr>
<th>Sensor</th>
<th>.baseOffset</th>
<th>Element</th>
<th>RAM Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slider0</td>
<td>0</td>
<td>element0</td>
<td>baseCnt[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element1</td>
<td>baseCnt[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element2</td>
<td>baseCnt[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element3</td>
<td>baseCnt[3]</td>
</tr>
<tr>
<td>Slider1</td>
<td>4</td>
<td>element4</td>
<td>baseCnt[4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element5</td>
<td>baseCnt[5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element6</td>
<td>baseCnt[6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element7</td>
<td>baseCnt[7]</td>
</tr>
</tbody>
</table>

`arrayPtr` identifies all of the elements associated with a sensor. In the case of the wheel and slider the order of the array is important because it is assumed that the order represents the physical order of the elements.

`measGateSource` is used to define either the gate timer source or the measurement clock source depending upon the method (halDefinition). In RC and fRO methods, measGateSource defines the measurement clock source. In the RO implementation measGateSource identifies the gate timer source.
sourceScale is used to further divide down the source (either the measurement source or the gate source depending upon the context). This only applies to timer peripherals and not the watchdog timer peripherals.

accumulationCycles is used to define the gate time. Typically accumulationCycles is the number of oscillations of the gate time, but when the watchdog timer is used as the gate peripheral the accumulationCycles represents the bit settings in the watchdog timer control register.

halDefinition identifies which measurement implementation is being used for the sensor. Table 8 lists the different implementations currently supported.

### Table 8. halDefinition Descriptions

<table>
<thead>
<tr>
<th>halDefinition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO_COMPAp_TA0_WDTp</td>
<td>Relaxation oscillator implemented with COMP_A+ peripheral. The gate time is fixed and defined by the</td>
</tr>
<tr>
<td>RO_COMPAp_TA1_WDTp</td>
<td>WDT+ peripheral set to interval mode. The capacitance is represented by the number of RO cycles</td>
</tr>
<tr>
<td></td>
<td>counted by TimerA0 or TimerA1 during the fixed gate time (see Section 3.2.2.1).</td>
</tr>
<tr>
<td>RO_PINOSC_TA0_WDTp</td>
<td>Relaxation oscillator implemented with PinOsc. TimerA0 is used to measure frequency of oscillator, and</td>
</tr>
<tr>
<td></td>
<td>the WDTp is used to set the gate time (see Section 3.2.2.3).</td>
</tr>
<tr>
<td>RO_PINOSC_TA1_WDTp</td>
<td>Relaxation oscillator implemented with PinOsc. TimerA1 is used to measure frequency of oscillator, and</td>
</tr>
<tr>
<td></td>
<td>the WDTp is used to set the gate time (see Section 3.2.2.3).</td>
</tr>
<tr>
<td>RO_PINOSC_TA0</td>
<td>Relaxation oscillator implemented with PinOsc. TimerA0 is used to measure frequency of oscillator, and</td>
</tr>
<tr>
<td></td>
<td>the ACLK source is used to set the gate time (see Section 3.2.2.4).</td>
</tr>
<tr>
<td>RO_PINOSC_TA0_TA1</td>
<td>Relaxation oscillator implemented with PinOsc. TimerA0 is used to measure frequency of oscillator, and</td>
</tr>
<tr>
<td></td>
<td>TimerA1 is used to set the gate time (see Section 3.2.2.3).</td>
</tr>
<tr>
<td>RO_PINOSC_TA0_TB0</td>
<td>Relaxation oscillator implemented with PinOsc. TimerA1 is used to measure frequency of oscillator, and</td>
</tr>
<tr>
<td></td>
<td>TimerB0 is used to set the gate time (see Section 3.2.2.3).</td>
</tr>
<tr>
<td>RO_COMPB_TA0_WDTA</td>
<td>Relaxation oscillator implemented with COMP_B peripheral, TimerA0, TimerA1, or TimerB0 is used to</td>
</tr>
<tr>
<td>RO_COMPB_TA1_WDTA</td>
<td>measure frequency of oscillator, and the WDTA peripheral is used to set the gate time (see Section</td>
</tr>
<tr>
<td>RO_COMPB_TB0_WDTA</td>
<td>3.2.2.2).</td>
</tr>
<tr>
<td>RO_COMPB_TA1_TA0</td>
<td>Relaxation oscillator implemented with COMP_B peripheral, TimerA1 is used to measure frequency of</td>
</tr>
<tr>
<td></td>
<td>oscillator, and TimerA0 is used to set the gate time (see Section 3.2.2.2).</td>
</tr>
<tr>
<td>RO_CSIO_TA2_WDTA</td>
<td>Relaxation oscillator implemented with capacitive touch IO peripheral, TimerA2 is used to measure</td>
</tr>
<tr>
<td></td>
<td>frequency of oscillator, and the WDTA peripheral is used to set the gate time (see Section 3.2.2.5).</td>
</tr>
<tr>
<td>RO_CSIO_TA2_TA3</td>
<td>Relaxation oscillator implemented with capacitive touch IO peripheral, TimerA2 is used to measure</td>
</tr>
<tr>
<td></td>
<td>frequency of oscillator, and TimerA3 is used to set the gate time (see Section 3.2.2.5).</td>
</tr>
<tr>
<td>RO_CSIO_TA0_WDTA</td>
<td>Relaxation oscillator implemented with capacitive touch IO peripheral, TimerA0 is used to measure</td>
</tr>
<tr>
<td></td>
<td>frequency of oscillator, and WDTA peripheral is used to set the gate time (see Section 3.2.2.5).</td>
</tr>
<tr>
<td>RO_CSIO_TA0_TA1</td>
<td>Relaxation oscillator implemented with capacitive touch IO peripheral, TimerA0 is used to measure</td>
</tr>
<tr>
<td></td>
<td>frequency of oscillator, and TimerA1 is used to set the gate time (see Section 3.2.2.5).</td>
</tr>
<tr>
<td>RO_CSIO_TA0_RTC</td>
<td>Relaxation oscillator implemented with capacitive touch IO peripheral, TimerA0 is used to measure</td>
</tr>
<tr>
<td></td>
<td>frequency of oscillator, and RTC peripheral is used to set the gate time (see Section 3.2.2.5).</td>
</tr>
<tr>
<td>RC_PAIR_TA0</td>
<td>Measure RC time constant with TimerA0. The gate time is variable and changes with the charge and</td>
</tr>
<tr>
<td></td>
<td>discharge time. A software loop is used to establish the number of charge and discharge cycles that</td>
</tr>
<tr>
<td></td>
<td>define the gate time. The capacitance is represented by the number of TimerA0 counts within the gate</td>
</tr>
<tr>
<td></td>
<td>time. Typically TA0 is sourced from a high frequency clock (SMCLK) for improved sensitivity. The</td>
</tr>
<tr>
<td></td>
<td>capacitive element is charged and discharge with the other IO defined in the pair (see Section 3.2.2.6).</td>
</tr>
<tr>
<td>fRO_COMPAp_TA0_SW</td>
<td>Fast Scan Relaxation oscillator implemented with COMP_A+ peripheral. The gate time is variable and</td>
</tr>
<tr>
<td>fRO_COMPAp_TA1_SW</td>
<td>changes with the period of the relaxation oscillator. The TimerA0 or TimerA1 is used to establish the</td>
</tr>
<tr>
<td></td>
<td>number of oscillations that define the gate time. The capacitance is represented by the number of</td>
</tr>
<tr>
<td></td>
<td>software loops counted within the gate time (see Section 3.2.2.13).</td>
</tr>
<tr>
<td>fRO_COMPAp_SW_TA0</td>
<td>Fast Scan Relaxation oscillator implemented with COMP_A+ peripheral. A software (SW) loop is used to</td>
</tr>
<tr>
<td></td>
<td>establish the number of oscillations that define the gate time. The capacitance is represented by the</td>
</tr>
<tr>
<td></td>
<td>number of timer counts in TimerA0 for the gate time. Typically TA0 is sourced from a high frequency</td>
</tr>
<tr>
<td></td>
<td>clock (SMCLK) for improved sensitivity (see Section 3.2.2.14).</td>
</tr>
<tr>
<td>fRO_PINOSC_TA0_SW</td>
<td>Measure the time with TimerA0. Relaxation oscillator implemented with PinOsc, several oscillations are</td>
</tr>
<tr>
<td></td>
<td>counted in SW to establish the gate time (see Section 3.2.2.10).</td>
</tr>
<tr>
<td>fRO_PINOSC_TA0_TA1</td>
<td>Fast Scan Relaxation oscillator implemented with PinOsc. The TimerA0 is used to establish the number</td>
</tr>
<tr>
<td></td>
<td>of oscillations that define the gate time. The capacitance is represented by the number of TimerA1</td>
</tr>
<tr>
<td></td>
<td>counts within the gate time. Typically TA1 is sourced from a high frequency clock (SMCLK) (see</td>
</tr>
<tr>
<td></td>
<td>Section 3.2.2.9).</td>
</tr>
</tbody>
</table>
Table 8. halDefinition Descriptions (continued)

<table>
<thead>
<tr>
<th>halDefinition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fRO_COMPB_TA0_SW</td>
<td>Fast Scan Relaxation oscillator implemented with the COMP_B peripheral. The TimerA0 or TimerA1 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of software loops counted within the gate time (see Section 3.2.2.12).</td>
</tr>
<tr>
<td>fRO_COMPB_TA1_SW</td>
<td>Fast Scan Relaxation oscillator implemented with COMP_B. The TimerA1 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of TimerA1 counts within the gate time. Typically TA1 is sourced from a high frequency clock (SMCLK) (see Section 3.2.2.11).</td>
</tr>
<tr>
<td>fRO_COMPB_TA1_TA0</td>
<td>Fast Scan Relaxation oscillator implemented with the COMP_B peripheral. The TimerA0 or TimerA1 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of software loops counted within the gate time (see Section 3.2.2.12).</td>
</tr>
<tr>
<td>fRO_CSIO_TA2_TA3</td>
<td>Fast Scan Relaxation oscillator implemented with COMP_B. The TimerA1 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of TimerA1 counts within the gate time. Typically TA1 is sourced from a high frequency clock (SMCLK) (see Section 3.2.2.7).</td>
</tr>
<tr>
<td>fRO_CSIO_TA0_TA1</td>
<td>Fast Scan Relaxation oscillator implemented with the capacitive touch IO. The TimerA0 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of TimerA3 counts within the gate time. Typically TA3 is sourced from a high frequency clock (SMCLK) (see Section 3.2.2.8).</td>
</tr>
<tr>
<td>fRO_CSIO_TA0_SW</td>
<td>Fast Scan Relaxation oscillator implemented with the capacitive touch IO. The TimerA0 is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of software loops counted within the gate time (see Section 3.2.2.15).</td>
</tr>
</tbody>
</table>

3.2.2.1 RO_COMPAp_TAx_WDTp

The relaxation oscillator is comprised of the COMP_A+ module, a reference, and RC filter. The reference is connected to the non-inverting input of COMP_A+ (via the input mux) while the RC filter is connected to the inverting input (also via an input mux). The reference is a voltage divider made up of one connection to a GPIO and the other connections to ground and the comparator output.

One way to measure the capacitance is to route the oscillator (via CAOUT) to a timer input (TAxCLK). Two different HAL definitions are provided depending upon which clock is available; RO_COMPAp_TA0_WDTp and RO_COMPAp_TA1_WDTp.

![Figure 17. RO_COMPAp_TAx Schematic Description](image)

The two timing parameters define the WDTp interval that is the gate time for the RO_COMPAp_TAx_WDTp implementation.
measureGateSource indicates the WDTp source: SMCLK or ACLK. This parameter is equivalent to the 'Watchdog timer+ clock source select' bits in the Watchdog Timer+ register.

### Table 9. Watchdog Timer Source Select Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATE_WDTp_ACLK</td>
<td>0x0004</td>
<td>ACLK</td>
</tr>
<tr>
<td>GATE_WDTp_SMCLK</td>
<td>0x0000</td>
<td>SMCLK</td>
</tr>
</tbody>
</table>

accumulationCycles is used to define the WDTp interval in the RO_COMPAp_TAx_WDTp implementation. This is equivalent to the interval select bits in the Watchdog Timer+ register.

### Table 10. Watchdog Timer+ Interval Select Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Interval (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTp_GATE_32768</td>
<td>0x0000</td>
<td>32768/source</td>
</tr>
<tr>
<td>WDTp_GATE_8192</td>
<td>0x0001</td>
<td>8192/source</td>
</tr>
<tr>
<td>WDTp_GATE_512</td>
<td>0x0002</td>
<td>512/source</td>
</tr>
<tr>
<td>WDTp_GATE_64</td>
<td>0x0003</td>
<td>64/source</td>
</tr>
</tbody>
</table>
Figure 18 shows how the common sensor parameters measGateSource and accumulationCycles are used to select the gate time.

Figure 18. Timing Parameters: Example WDTp
The following example describes a sensor made up of four elements, and each element is measured with the RO method for a period of 512/SMCLK.

```c
const struct Sensor slider = {
    .halDefinition = RO_COMPBp_TA0_WDTp,
    .numElements = 4,
    .baseOffset = 0,
    .points = 80,
    .sensorThreshold = 50,
    // Pointer to elements
    .arrayPtr[0] = &element0, // point to first element
    .arrayPtr[1] = &element1,
    .arrayPtr[2] = &element2,
    .arrayPtr[3] = &element3,

    // Reference Information
    // CAOUT is P1.7
    // TACLK is P1.0
    .caoutDirRegister = (uint8_t *)&P1DIR, // PxDIR
    .caoutSelRegister = (uint8_t *)&P1SEL, // PxSEL
    .txclkDirRegister = (uint8_t *)&P1DIR, // PxDIR
    .txclkSelRegister = (uint8_t *)&P1SEL, // PxSEL
    .caoutBits = BIT7, // BITy
    .txclkBits = BIT0,
    // Reference is on P1.6
    .refPxoutRegister = (uint8_t *)&P1OUT,
    .refPxdirRegister = (uint8_t *)&P1DIR,
    .refBits = BIT6, // BIT6
    .refCactl2Bits = P2CA4, // CACTL2-> P2CA4, CA1
    .capdBits = (BIT1+BIT2+BIT3+BIT4+BIT5),

    // Timer Information
    .measGateSource= GATE_WDTp_SMCLK, // 0->SMCLK, 1-> ACLK
    .accumulationCycles = WDTp_GATE_512 // 512
};
```

### 3.2.2.2 RO_COMPB_yyy_zzz

The relaxation oscillator solution with COMP_B is the same in function as the COMP_A+ solution. The COMP_B peripheral solution is different in implementation, integrating the reference circuitry and connection to the TimerA or TimerB peripheral, as shown in Figure 19.

![Figure 19. RO_COMPB Schematic](image-url)
### 3.2.2.2.1 RO_COMPB_TAx_WDTA, RO_COMPB_TBx_WDTA

The two timing parameters define the WDTA interval that is the gate time for the implementation. The WDTA module provides four different source settings and eight watchdog timer intervals.

`measureGateSource` indicates the WDTA source: SMCLK, ACLK, VLO, or XCLK. This parameter is equivalent to the Watchdog timer clock source select bits in the Watchdog Timer Control register (WDTCTL).

#### Table 11. Watchdog Source Select Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATE_WDTA_SMCLK</td>
<td>0x0000</td>
<td>SMCLK</td>
</tr>
<tr>
<td>GATE_WDTA_ACLK</td>
<td>0x0020</td>
<td>ACLK</td>
</tr>
<tr>
<td>GATE_WDTA_VLO</td>
<td>0x0040</td>
<td>VLO</td>
</tr>
<tr>
<td>GATE_WDTA_XCLK</td>
<td>0x0060</td>
<td>XCLK</td>
</tr>
</tbody>
</table>

`accumulationCycles` is used to define the WDTA interval. This is equivalent to the interval select bits in the Watchdog Timer Control Register (WDTCTL).

#### Table 12. Watchdog Interval Select Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Interval (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTA_GATE_2G</td>
<td>0x0000</td>
<td>2G/source</td>
</tr>
<tr>
<td>WDTA_GATE_128M</td>
<td>0x0001</td>
<td>128M/source</td>
</tr>
<tr>
<td>WDTA_GATE_8192K</td>
<td>0x0002</td>
<td>8192k/source</td>
</tr>
<tr>
<td>WDTA_GATE_512K</td>
<td>0x0003</td>
<td>512k/source</td>
</tr>
<tr>
<td>WDTA_GATE_32768</td>
<td>0x0004</td>
<td>32768/source</td>
</tr>
<tr>
<td>WDTA_GATE_8192</td>
<td>0x0005</td>
<td>8192/source</td>
</tr>
<tr>
<td>WDTA_GATE_512</td>
<td>0x0006</td>
<td>512/source</td>
</tr>
<tr>
<td>WDTA_GATE_64</td>
<td>0x0007</td>
<td>64/source</td>
</tr>
</tbody>
</table>

The following example describes a sensor made up of four elements and each element is measured with the RO method for a period of 512000/SMCLK.

```c
const struct Sensor sliderA = {
    .halDefinition = RO_COMPB_TA0_WDTA,  
    .numElements = 4, 
    .baseOffset = 0, 
    .points = 64, 
    .sensorThreshold = 5, 
    .cbpdBits = (BITC+BITD+BITE+BIFT), 
    // Pointer to elements 
    .arrayPtr[0] = &element0, 
    .arrayPtr[1] = &element4, 
    .arrayPtr[2] = &element8, 
    .arrayPtr[3] = &elementC, 
    .cboutTAxDirRegister = (uint8_t *)&P3DIR, // PxDIR
    .cboutTAxSelRegister = (uint8_t *)&P3SEL, // PxSEL
    .cboutTAxBits = BIT4, // P3.4 
    // Timer Information
    .measGateSource= GATE_WDTA_SMCLK,  
    .accumulationCycles= WDTA_GATE_512K  
};
```

Different members within the 5xx family provide an internal connection between CBOUT and a timer. The description and parameters are the same for timers TA0, TA1, and TB0, with the exception of the HAL definition name.
The timing parameters define the timer interval that is the gate time for the implementation. The timer
module provides four different source settings, a divider to scale the input source, and the interval, which
is a 16-bit integer loaded into the timer compare register.

measureGateSource indicates the timer source: TxCLK, ACLK, SMCLK, or INCLK. This parameter is
equivalent to the timer source select bits (TxSSEL) in the timer control register (TxCTL).

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_TxCLK</td>
<td>0x0000</td>
<td>TxCLK</td>
</tr>
<tr>
<td>TIMER_ACLK</td>
<td>0x0100</td>
<td>ACLK</td>
</tr>
<tr>
<td>TIMER_SMCLK</td>
<td>0x0200</td>
<td>SMCLK</td>
</tr>
<tr>
<td>TIMER_INCLK</td>
<td>0x0300</td>
<td>INCLK or inverted TxCLK</td>
</tr>
</tbody>
</table>

sourceScale is used to divide the timer source. This parameter is equivalent to the input divider select bits
(ID) found in the timer control register (TxCTL).

accumulationCycles is used to define the timer interval. This value is loaded into the capture/compare
register (TxCCR0).

The following example describes a sensor made up of four elements and each element is measured with
the RO method for a period of 50/ACLK.

```c
const struct Sensor keypad =
{
    .halDefinition = RO_COMPB_TA1_TA0,
    .numElements = 5,
    .baseOffset = 0,
    .cbpdBits = 0x001F,  // CB0,CB1,CB2,CB3,CB4
    .arrayPtr[0] = &element0,  // point to first element
    .arrayPtr[1] = &element1,
    .arrayPtr[2] = &element2,
    .arrayPtr[3] = &element3,
    .arrayPtr[4] = &element4,
    .cboutTAxBits = BIT6,  // P1.6
    .cboutTAxDirRegister = (uint8_t *)&P1DIR,  // PxDIR
    .cboutTAxSelRegister = (uint8_t *)&P1SEL,  // PxSEL
    .measGateSource = TIMER_ACLK,  // ACLK
    .sourceScale = TIMER_SOURCE_DIV_0, // ACLK/1
    // Timer Information
    /* 50 ACLK/1 cycles or 50*1/32Khz = 1.5ms */
    .accumulationCycles = 50
};
```
3.2.2.3 RO_PinOsc_TAx_zzz

3.2.2.3.1 RO_PinOsc_TA0_WDTp

The common timing parameters, measGateSource and accumulationCycles, are the same as the COMP_A+ implementation (see Section 3.2.2.1).

The following sensor definition describes a sensor made up of one element that is measured with the RO method for a period of 8192/SMCLK.

```c
const struct Sensor middle_button = {
  .halDefinition = RO_PINOSC_TA0_WDTp,
  .numElements = 1,
  .baseOffset = 4,
  // Pointer to elements
  .arrayPtr[0] = &middle_element, // point to first element
  // Timer Information
  .measGateSource = GATE_WDTp_SMCLK, // 0->SMCLK, 1-> ACLK
  .accumulationCycles = WDTp_GATE_8192 // 8192
  // .accumulationCycles = WDTp_GATE_32768 //32768
  // .accumulationCycles = WDTp_GATE_512 //512
  // .accumulationCycles = WDTp_GATE_64 //64
};
```

3.2.2.3.2 RO_PinOsc_TA0_TA1

The two timing parameters define the timer interval that is the gate time for the implementation. The timer module provides four different source settings, a divider to scale the input source, and the interval, which is a 16-bit integer loaded into the timer compare register.

`measureGateSource` indicates the timer source: TxCLK, ACLK, SMCLK, or INCLK. This parameter is equivalent to the timer source select bits (TxSSEL) in the timer control register (TxCTL).

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_TxCLK</td>
<td>0x0000</td>
<td>TxCLK</td>
</tr>
<tr>
<td>TIMER_ACLK</td>
<td>0x0100</td>
<td>ACLK</td>
</tr>
<tr>
<td>TIMER_SMCLK</td>
<td>0x0200</td>
<td>SMCLK</td>
</tr>
<tr>
<td>TIMER_INCLK</td>
<td>0x0300</td>
<td>INCLK or inverted TxCLK</td>
</tr>
</tbody>
</table>

`sourceScale` is used to divide the timer source. This parameter is equivalent to the input divider select bits (ID) found in the timer control register (TxCTL).

`accumulationCycles` is used to define the timer interval. This value is loaded into the capture/compare register (TxCCR0).
The following example describes a sensor made up of four elements and each element is measured with the RO method for a period of 11000/ACLK.

```c
const struct Sensor proximity =
{
    /* Gate source is INCLK by default */
    .hallDefinition = RO_PINOSC_TA0_TA1,
    .numElements = 1,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &eProx,
    /*
    * Timer Information
    * Gate Source is SMCLK
    */
    .measGateSource = TIMER_SMCLK,
    /* Gate Source divided by 1 */
    .sourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 11000 // Number of TimerA0 (RO) cycles
};
```

### 3.2.2.4 RO_PinOsc_TA0

An alternative implementation of the RO_PinOsc, with select MSP430 devices (1), is to use the internal ACLK connection to the timer capture input. The gate time is the number of capture events (equivalent to ACLK cycles), while the frequency counter is still the peripheral TimerA0 sourced from the relaxation oscillator. Because the capture interrupt represents a single oscillation several interrupts are counted with a software loop to create the equivalent gate time. Unlike the WDT method where the measurement is done in low power mode, this software loop method consumes more power because the CPU stays in Active Mode.

(1) See the data sheet to determine if the Timer capture input supports this connection to ACLK.
As shown in Figure 20, the only timing parameter definition for the RO_PinOsc_TA0 implementation is the number of ACLK cycles: sensor->accumulationCycles.

Figure 20. RO_PinOsc_TA0 Timing Parameter

The following sensor definition describes a sensor made up of one element that is measured with the RO method for a period of 100/ACLK.

```c
const struct Sensor volume = {
  .halDefinition = RO_PINOSC_TA0,
  .numElements = 2,
  .baseOffset = 0,
  // Pointer to elements
  .arrayPtr[0] = &up, // point to first element
  .arrayPtr[1] = &down, //
  // Timer Information
  .accumulationCycles= 100 // 100 ACLK cycles
};
```
3.2.2.5  RO_CSIO_yyy_zzz

3.2.2.5.1  RO_CSIO_TA2_WDTA

The common timing parameters, measGateSource and accumulationCycles, are the same as the COMP_B implementation (see Section 3.2.2.2).

The following sensor definition describes a sensor made up of 6 elements, where each element is measured with the RO method for a period of 64/SMCLK.

```c
const struct Sensor keyPad =
{
    .halDefinition = RO_CSIO_TA2_WDTA,
    .inputCapsioctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = GATE_WDTA_SMCLK,
    .accumulationCycles = WDTA_GATE_64
};
```

3.2.2.5.2  RO_CSIO_TA0_WDTA

The common timing parameters, measGateSource and accumulationCycles, are the same as the COMPB implementation (see Section 3.2.2.2).

The following sensor definition describes a sensor made up of 6 elements, where each element is measured with the RO method for a period of 64/SMCLK.

```c
const struct Sensor keyPad =
{
    .halDefinition = RO_CSIO_TA0_WDTA,
    .inputCapsioctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = GATE_WDTA_SMCLK,
    .accumulationCycles = WDTA_GATE_64
};
```
The timing parameters define the timer interval that is the gate time for the implementation. The timer module provides four different source settings, a divider to scale the input source, and the interval, which is a 16-bit integer loaded into the timer compare register.

`measureGateSource` indicates the timer source: TxCLK, ACLK, SMCLK, or INCLK. This parameter is equivalent to the timer source select bits (TxSSEL) in the timer control register (TxCTL).

| Table 15. TimerA Source Select Definitions |
|-------------------------------|---|-------------------------------|
| Definition | Value | Source             |
| TIMER_TxCLK      | 0x0000 | TxCLK |
| TIMER_ACLK        | 0x0100 | ACLK |
| TIMER_SMCLK       | 0x0200 | SMCLK |
| TIMER_INCLK       | 0x0300 | INCLK or inverted TxCLK |

`sSourceScale` is used to divide the timer source. This parameter is equivalent to the input divider select bits (ID) found in the timer control register (TxCTL).

`accumulationCycles` is used to define the timer interval. This value is loaded into the capture/compare register (TxCRR0).

The following example describes a sensor made up of 6 elements and each element is measured with the RO method for a period of 400/SMCLK.

```c
const struct Sensor keyPad =
{
    .halDefinition = RO_CSIO_TA2_TA3,
    .inputCapsioctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = TIMER_SMCLK,
    .sSourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 400
};
```
3.2.2.5.4 **RO_CSIO_TA0_TA1**

The common timing parameters, measGateSource and accumulationCycles, are the same as the RO_CSIO_TA2_TA3 implementation.

The following example describes a sensor made up of six elements and each element is measured with the RO method for a period of 400/SMLK.

```c
const struct Sensor keyPad = {
    .halDefinition = RO_CSIO_TA0_TA1,
    .inputCapsoctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = TIMER_SMCLK,
    .sourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 400
};
```

3.2.2.5.5 **RO_CSIO_TA0_RTC**

The timing parameters define the RTC interval which is the gate time for the implementation. The RTC module provides 3 different source settings, a divider to scale the input source, and the interval which is a 16 bit integer loaded into the RTC modulo register.

*measureGateSource* indicates the timer source: SMCLK, VLOCLK or XT1CLK. This parameter is equivalent to the real-time clock source select bits (RTCSS) in the RTC control register (RTCCTL).

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATE_RTC_SMCLK</td>
<td>0x1000</td>
<td>SMCLK</td>
</tr>
<tr>
<td>GATE_RTC_XCLK</td>
<td>0x2000</td>
<td>XT1CLK</td>
</tr>
<tr>
<td>GATE_RTC_VLO</td>
<td>0x3000</td>
<td>VLOCLK</td>
</tr>
</tbody>
</table>

*sourceScale* is used to divide the timer source. This parameter is equivalent to the real-time clock pre-divider select bits (RTCPSS) found in the RTC control register (RTCCTL).

*accumulationCycles* is used to define the timer interval. This value is loaded into the RTC modulo register (RTCMOD).
The following example describes a sensor made up of six elements and each element is measured with the RO method for a period of 400/SMCLK.

```c
const struct Sensor keyPad =
{    .halDefinition = RO_CSIO_TA0_RTC,
    .inputCapsioctlRegister = (uint16_t *) &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = GATE_RTC_SMCLK,
    .sourceScale = RTC_DIV_10,
    .accumulationCycles = 40
};
```

### 3.2.2.6 RC_PAIR_TA0

The RC method uses the timer peripheral to measure the charge and discharge time of the RC circuit. This measurement can be increased (in time and in counts) by accumulating several charge and discharge cycles as shown in Figure 21. The number of cycles is defined in the parameter: sensor->accumulation cycles.

![Figure 21. RC Timing Parameters](image)

The following sensor definition describes a sensor made up of two elements that are measured with the RC method. The gate time for each element is four charge and discharge cycles.

```c
const struct Sensor scroll =
{    .halDefinition = RC_PAIR_TA0,
    .numElements = 2,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &left, // point to first element
    .arrayPtr[1] = &right, //
    // Timer Information
    .accumulationCycles= 4 // 4 charge/discharge cycles
};
```
3.2.2.7 fRO_CSIO_TA2_TA3

The timing source for gate timer, TIMERA2, is part of the HAL definition. The source for the gate timer can be scaled with the `sourceScale` parameter to increase the gate time, but it is recommended to use the `/1` setting. The gate time is a function of the scale and the number of oscillations shown as ‘n’ in Figure 22. The number of oscillations is defined by the parameter `accumulationCycles`.

The timing parameter `measGateSource` is used to select the clock source for the frequency counter, TIMERA3. Typically this source is SMCLK, because it represents the highest resolution clock. In the following code snippet the gate time for each element is 50 relaxation cycles. The number of SMCLK cycles within that gate time represents the capacitance.

```c
const struct Sensor keyPad = {
    .halDefinition = fRO_CSIO_TA2_TA3,
    .inputCapsioctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = TIMER_SMCLK,
    .sourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 50
};
```

Figure 22. fRO_CSIO_TA2_TA3 Timing Parameters
3.2.2.8 fRO_CSIO_TA0_TA1

The common timing parameters, measGateSource and accumulationCycles, are the same as the fRO_CSIO_TA2_TA3 implementation.

In the following code snippet, the gate time for each element is 50 relaxation cycles. The number of SMCLK cycles within that gate time represents the capacitance.

```c
const struct Sensor keyPad = {
    .halDefinition = fRO_CSIO_TA0_TA1,
    .inputCapsioctlRegister = (uint16_t *)&CAPSIO0CTL,
    .numElements = 6,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &key0,
    .arrayPtr[1] = &key1,
    .arrayPtr[2] = &key2,
    .arrayPtr[3] = &key3,
    .arrayPtr[4] = &key4,
    .arrayPtr[5] = &key5,
    .measGateSource = TIMER_SMCLK,
    .sourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 50
};
```

3.2.2.9 fRO_PinOsc_TA0_TA1

The timing source for gate timer, TIMERA0, is part of the HAL definition. The source for the gate timer can be scaled with the `sourceScale` parameter to increase the gate time, but it is recommended to use the /1 setting. The gate time is a function of the scale and the number of oscillations shown as 'n' in Figure 23. The number of oscillations is defined by the parameter `accumulationCycles`.

![Diagram of fRO_PinOsc_TA0_TA1 Timing Parameters](Image)

**Figure 23. fRO_PinOsc_TA0_TA1 Timing Parameters**
The timing parameter `measGateSource` is used to select the clock source for the frequency counter, `TIMERA1`. Typically this source is SMCLK, because it represents the highest resolution clock. In the following code snippet the number of SMCLK cycles counted during the gate time of 11K relaxation cycles represents the capacitance.

```c
const struct Sensor proximity = {
    /* Gate source is INCLK by default */
    .halDefinition = fRO_PINOSC_TA0_TA1,
    .numElements = 1,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &eProx,
    // Timer Information
    /* Measurement Source is SMCLK */
    .measGateSource = TIMER_SMCLK,
    /* Gate Source divided by 1 */
    .sourceScale = TIMER_SOURCE_DIV_0,
    .accumulationCycles = 11000 // Number of RO cycles
};
```

### 3.2.2.10 fRO_PinOsc_TA0_SW

In the case of the fRO_PinOsc_TA0_SW implementation, the variable gate timer is created with the relaxation oscillator and the peripheral TimerAx. The frequency counter is a software loop with a frequency of MCLK/10.

The timing sources are part of the HAL definition therefore the only parameter to define is the number of oscillations for the gate time. This number, shown as 'n' in Figure 24, is defined by the variable `accumulationCycles` at the sensor level: sensor->accumulationCycles.
3.2.2.11 fRO_COMPB_TA1_TA0

The COMP_B implementation is found in the 5xx and 6xx MSP430 families of devices. These families can support clocks speeds up to 25 MHz, further improving the sensitivity demonstrated in Table 4.

The timing source for gate timer, TIMERA1, is part of the HAL definition. The source for the gate timer can be scaled with the `sourceScale` parameter to increase the gate time, but it is recommended to use the /1 setting. The gate time is a function of the scale and the number of oscillations shown as 'n' in Figure 25. The number of oscillations is defined by the parameter `accumulationCycles`.

![Figure 25. fRO_COMPB_TA1_TA0 Timing Parameters](image)
The timing parameter \textit{measGateSource} is used to select the clock source for the frequency counter, TIMERA0. Typically this source is SMCLK, because it represents the highest resolution clock. In the following code snippet the number of SMCLK cycles counted during the gate time of 10 relaxation cycles (per element) represents the capacitance.

```c
const struct Sensor keypad = {
    .halDefinition = fRO_COMPB_TA1_TA0,
    .numElements = 5,
    .baseOffset = 0,
    .cbpdBits = 0x001F, //CB0,CB1,CB2,CB3,CB4
    .arrayPtr[0] = &element0, // point to first element
    .arrayPtr[1] = &element1,
    .arrayPtr[2] = &element2,
    .arrayPtr[3] = &element3,
    .arrayPtr[4] = &element4,
    .cboutTAXdirRegister = (uint8_t *)&P1DIR, // PxDIR
    .cboutTAXselRegister = (uint8_t *)&P1SEL, // PxSEL
    .cboutTAXBits = BIT6, // P1.6
    .measGateSource = TIMER_SMCLK, // SMCLK
    .sourceScale = TIMER_SOURCE_DIV_0, // RO/1
    .accumulationCycles = 10
};
```

### 3.2.2.12 \texttt{fRO\_COMPB\_TAX\_SW}

The \texttt{fRO\_COMPB\_TAX\_SW} implementation has the same hardware description as the \texttt{RO\_COMPB\_TAX\_WDTA} implementation (see Section 3.2.2.2). As already mentioned the key difference between the RO and fRO methods is that the frequency counter and gate timer inputs are switched. The gate timer now is a function of the capacitance being measured and the frequency counter is fed by a fixed frequency (a system clock). In the case of the \texttt{fRO\_COMPB\_TAX\_SW} implementation, the variable gate timer is created with the relaxation oscillator and the peripheral TimerAx. The gate time is a software loop with a frequency of MCLK/10.

The fast RO method is typically used in devices that have multiple timers available, so that the frequency counter function is performed with another hardware timer instead of with a software loop. This not only decreases power consumption (running in LPM0 instead of in active mode), but is also removes the 10x factor associated with the software.

There is only one timing parameter, \texttt{accumulationCycles}. Figure 26 shows how the parameter \texttt{accumulationCycles} is used to accumulate multiple relaxation oscillator cycles in the \texttt{fRO\_COMPB\_TAX\_SW} method.
3.2.2.13 \textit{fRO\_COMP\_TAx\_SW}

The \textit{fRO\_COMP\_TAx\_SW} implementations have the same hardware description as the \textit{RO\_COMP\_TAx\_WDTp} implementations (see Figure 17). As already mentioned the key difference between the RO and \textit{fRO} methods is that the frequency counter and gate timer inputs are switched. The gate timer now is a function of the capacitance being measured and the frequency counter is fed by a fixed frequency (a system clock). In the case of the \textit{fRO\_COMP\_TAx\_SW} implementation, the variable gate timer is created with the relaxation oscillator and the peripheral TimerAx. The gate time is a software loop with a frequency of MCLK/10.

The fast RO method is typically used in devices that have multiple timers available, so that the frequency counter function is performed with another timer peripheral instead of with a software loop. This not only decreases power consumption (running in LPM0 instead of in active mode), but is also removes the 10x factor associated with the software.
There is only one timing parameter; accumulationCycles. Figure 27 shows how the parameter accumulationCycles is used to accumulate multiple relaxation oscillator cycles in the fRO_COMPAp_TA0_SW method.

**Figure 27. fRO_COMPAp_TA0_SW Timing Parameters**

### 3.2.2.14 fRO_COMPAp_SW_TAx

There are three timing parameters; measureGateSource, sourceScale, and accumulationCycles. The gate timer which defines the measurement period is defined by accumulationCycles. The software loop counts the relaxation oscillator cycles until accumulationCycles is reached and at this time reads the timer, TAx. The measureGateSource and sourceScale configure the TAx peripheral. Specifically these parameters define the source (typically SMCLK) and the timer divider (typically divide by 1).

measureGateSource identifies the clock source for TAx. This parameter is the equivalent to the TASSELx bits found in the TimerA control register TACTL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Timer Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_TxCLK</td>
<td>0x0000</td>
<td>TxCLK</td>
</tr>
<tr>
<td>TIMER_ACLK</td>
<td>0x0100</td>
<td>ACLK</td>
</tr>
<tr>
<td>TIMER_SMCLK</td>
<td>0x0200</td>
<td>SMCLK</td>
</tr>
<tr>
<td>TIMER_INCLK</td>
<td>0x0300</td>
<td>INCLK</td>
</tr>
</tbody>
</table>
sourceScale is used to divide the timer source. This is equivalent to the input divider bits (IDx) found in the TimerA control register TACTL.

Table 18. sourceScale Definitions for fRO_xxxx_SW_Txx

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Source Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_SOURCE_DIV_0</td>
<td>0x0000</td>
<td>/1</td>
</tr>
<tr>
<td>TIMER_SOURCE_DIV_1</td>
<td>0x0040</td>
<td>/2</td>
</tr>
<tr>
<td>TIMER_SOURCE_DIV_2</td>
<td>0x0080</td>
<td>/4</td>
</tr>
<tr>
<td>TIMER_SOURCE_DIV_3</td>
<td>0x00C0</td>
<td>/8</td>
</tr>
</tbody>
</table>

accumulationCycles defines the number of relaxation oscillator cycles per gate period. In the fRO_COMPAp_SW_TA0 method the counting of relaxation oscillator cycles is done with a software polling loop that looks for a comparator interrupt flag to indicate that an oscillation has occurred.

3.2.2.15 fRO_CSIO_TA0_SW

In the case of the fRO_CSIO_TA0_SW implementation, the variable gate timer is created with the relaxation oscillator and the peripheral TimerA0. The frequency counter is a software loop with a frequency of MCLK/10.

The timing sources are part of the hal definition therefore the only parameter to define is the number of oscillations for the gate time. This number, shown as ‘n’ in Figure 28, is defined by the variable accumulationCycles at the sensor level: sensor->accumulationCycles.

![Diagram of fRO_CSIO_TA0_SW Timing Parameters](image)
3.2.3 Slider and Wheel Specific Sensor Variables

The following definitions are only required with the use of API functions TI_CAPT_Wheel and TI_CAPT_Slider.

To include the wheel or slider API within the library the following definitions need to be made in the structure.h file:

```c
// Are wheel or slider representations used?
#define SLIDER
#define WHEEL

// Illegal slider position. This value is returned
// when no touch on the wheel or slider is detected.
#define ILLEGAL_SLIDER_WHEEL_POSITION 0xFFFF
```

In the structure.c file the sensor definitions for `points` and `sensorThreshold` need to be added.

The variable `points` is used to define the number of points along a slider or wheel.

`sensorThreshold` defines the cumulative response required by the sensor to declare a valid touch. The intent of this variable is to distinguish a genuine interaction with the sensor from an unintentional interaction that may activate only one element.

The wheel or slider `sensorThreshold` is compared with the response of the dominant element and its neighbors (summation of x-1, x, and x+1). The endpoints of a slider are a special case which requires a comparison of only the end element (the dominant element) and the one neighbor. If the response exceeds the `sensorThreshold` then a valid use case of the sensor has been validated and the position is calculated. If no valid use case is detected, then the `ILLEGAL_SLIDER_WHEEL_POSITION`, defined in the structure.h file is returned.

4 Resources

Depending upon the configuration this library can consume several different resources making them completely unavailable to the main application or only unavailable during actual measurement cycles. Resources that are completely unavailable are typically the digital I/O and allocated memory resources.

The library does perform a simple context save of all the registers used to minimize the need for resetting peripherals. It should be noted that the context save is not extensive and a good practice is to clear IFG flags before enabling interrupts. (1)

4.1 Time

The API calls found in the library are 'blocking' calls and do not return the CPU to the application until after the measurement is complete. The dominant factor on how long the CPU is unavailable is the gate time. This time can either be a number of cycles from a fixed (system clock) or variable (relaxation oscillator) clock source. Table 19 shows some example gate times for various capacitance measurement methods and settings. It is important to note that sensitivity is directly related to the gate time. Shortening the gate time results in a decrease in sensitivity. In the fRO method, the sensitivity can be increased (while keeping the shorter gate time) by increasing the fixed system clock frequency as described in Section 2.3.

(1) An explicit example of this is with TimerA3 where the Library does not use all three capture and control registers however, the CCIFG may be set when the timer is used.
Table 19. Gate Time Examples

<table>
<thead>
<tr>
<th>Method</th>
<th>Gate Time Source (.measGateSource)</th>
<th>Interval Definition (.accumulationCycles)</th>
<th>Time (ms)</th>
</tr>
</thead>
</table>
| RO_XXX_YYY_WDTp  
RO_XXX_YYY_WDTa | ACLK = VLO = 12 kHz  
SMCLK = 2 MHz  
SMCLK = 1 MHz | 64 (WDTp_GATE_64)  
8192 (WDTp_GATE_8192)  
512 (WDTp_GATE_512) | 5.33  
4.1  
0.512 |
| RO_PINOSC_TA0 | ACLK = VLO = 12 kHz | 100 | 8.3 |
| RC_PAIR_XXX | Rise+Fall (untouched) ≈ 1.4 µs  
Rise+Fall (touched) ≈ 1.6 µs | 20  
20 | 0.028 + TBD(1)  
0.032 + TBD |
| fRO_COMPx_YYY_SW | RO(untouched) = 700 kHz  
RO(touched) = 600 kHz | 4000  
4000 | 5.71  
6.67 |
| fRO_PINOSC_YYY_SW | RO(untouched) = 1.2 MHz  
RO(touched) = 1 MHz | 4000  
4000 | 3.334  
6.67 |
| fRO_COMPx_SW_YYY | RO(untouched) = 700 kHz  
RO(touched) = 600 kHz | 500  
500 | 0.714  
0.833 |

(1) This additional time is the overhead associated with using software to setup the charge and discharge over several cycles.

4.2 Memory: Flash and RAM

The amount of code space consumed by the library is directly a function of the number of elements, the number of sensors, the measurement method, and the level of abstraction. Table 20 shows an example of how the code size increases with higher levels of abstraction.

Table 20. Example Flash Resource Allocation

<table>
<thead>
<tr>
<th>API Calls</th>
<th>Library (bytes)</th>
<th>Configuration Structure: Six Elements, Three Sensors (RO_PinOsc_TA0_WDTp)</th>
<th>Comments</th>
</tr>
</thead>
</table>
| TL_CAPT_Raw | 376 (0x178)  
TI_CAPT_Init_Baseline  
TI_CAPT_Update_Baseline  
TI_CAPT_Custom | 114 (0x72) | Optimization level 0 (CCSv5, TI v4.1.2) |
| TI_CAPT_Init_Baseline  
TI_CAPT_Update_Baseline  
TI_CAPT_Custom | 1814 (0x716)  
114 (0x72) | Optimization level 0 (CCSv5, TI v4.1.2) |
| TI_CAPT_Init_Baseline  
TI_CAPT_Update_Baseline  
TI_CAPT_Custom  
TI_CAPT_Button  
TI_CAPT_Wheel | 2478 (0x9AE)  
120 (0x7B) | Optimization level 0 (CCSv5, TI v4.1.2) |

RAM can be allocated statically to maintain the baseline tracking feature. The amount of RAM needed is a function of the total number of elements: 2 bytes per element. The library uses the TOTAL_NUMBER_OF_ELEMENTS definition to indicate that RAM needs to be allocated for the baseline tracking and how much. When using only the TL_CAPT_RAW API TOTAL_NUMBER_OF_ELEMENTS should not be defined and therefore no RAM resources are consumed.

RAM can also be allocated statically or dynamically to perform the measurements to determine a change in capacitance (TI_CAPT_Custom and sensor abstractions). In the event that the RAM is allocated statically, the definition RAM_FOR_FLASH must be made in the structure.h file. The amount of RAM space allocated is dependent upon the largest number of elements per sensor, 2 bytes per element.

At the cost of additional FLASH space, this RAM can be allocated dynamically from a HEAP. To allocate the RAM dynamically the RAM_FOR_FLASH definition must be omitted. The HEAP size needs to be set (in the IDE) to 2 bytes plus 2 bytes per number of elements in the largest sensor.
4.3 **System Clocks**

The library does not make any adjustments to the system clocks (MCLK, SMCLK, or ACLK) and uses them as defined in the application layer for capacitance measurements. It is important to understand the clock usage of the library in the context of the application. For example, if the capacitance measurement time is set with the watchdog timer interval to 8192/SMCLK, then changing the frequency of SMCLK in the application also changes the measurement time during the capacitance measurement. If the clock source for the capacitance measurement is changed in an application, then it is important to re-initialize the baseline tracking accordingly.

4.4 **Peripherals**

Different combinations of peripherals can be used to measure changes in capacitance. While these peripherals are not available to the application during a capacitance measurement, most of the peripherals can be shared or multiplexed in time with other applications or functions.

4.4.1 **TimerA and TimerB**

The timer peripheral is reconfigured and initialized with every measurement and, therefore, can be used for other functions when a capacitance measurement is not taking place.

4.4.2 **Real-Time Timer**

The real-time timer can be used as the gate timer for capacity measurement when not used as the real-time clock for the system. The overflow of RTC can be used to trigger capture action on TimerA in some devices which makes it very convenient in capacity measuring.

4.4.3 **Watchdog Timer**

The Watchdog timer ISR is defined within the library and included with a precompiler directive. This definition prohibits using the WDT peripheral as a true watchdog. However, the source code is provided and the ISR can be modified to support the scheduling of the WDT for different tasks along with capacitive touch at different times within an application.

4.4.4 **Comparator_A and Comparator_B**

The comparator peripheral is reconfigured and initialized with every measurement and therefore can be used for other tasks within an application when a capacitance measurement is not taking place. It is not recommended to connect other inputs to the capacitive sensor element because this might interfere with the capacitance measurement.

4.4.5 **Digital I/O**

It is not recommended to share or multiplex functions on I/O pins that are used for capacitance measurements.

5 **API Calls**

The library provides three different layers of abstraction. The lowest level of abstraction is the TI_CAPT_Raw API function call. This function call measures the appropriate sensor and provides the raw capacitance measurement to the application layer. The TI_CAPT_RAW function is the most powerful in that it allows the most flexibility in interpretation and application of the capacitance measurement.

The next level of abstraction is the TI_CAPT_Custom API function call. This API calls the TI_CAPT_Raw function and also includes a baseline tracking algorithm. The TI_CAPT_Custom API provides, to the application layer, the magnitude of change of the measured capacitance from the baseline capacitance.

Changes are only provided to the application layer if the change is in the direction of interest. The direction of interest can be either an increase in capacitance or a decrease in capacitance. The direction of interest can be set with the TI_CAPT_Update.DOI API, and the default setting is an increase in capacitance. If the change in capacitance is against the direction of interest this information is used by the baseline tracking but not provided to the application layer.
The TI_CAPT_Custom API is intended for use with custom sensor designs and applications that are not supported by the other APIs. In these custom applications the baseline tracking can still be used but the interpretation of the change in capacitance must be handled in the application code.

The level of abstraction above the TI_CAPT_Custom API, includes the sensor representation of a button, group of buttons, a wheel, and a slider. These APIs are TI_CAPT_Button, TI_CAPT_Buttons, TI_CAPT_Wheel, and TI_CAPT_Slider, respectively. These APIs include the interpretation and application of the TI_CAPT_Custom function.

Table 21 lists the APIs of the capacitive touch library. In addition to the APIs just described, there are APIs to adjust the baseline tracking.

Table 21. Capacitive Touch Library APIs

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t TI_CAPT_Button(Sensor *); — Capacitance Measurement</td>
<td>45</td>
</tr>
<tr>
<td>element * TI_CAPT_Buttons(Sensor *); — Capacitance Measurement</td>
<td>45</td>
</tr>
<tr>
<td>uint16_t TI_CAPT_Slider(Sensor *); — Capacitance Measurement</td>
<td>46</td>
</tr>
<tr>
<td>uint16_t TI_CAPT_Wheel(Sensor *); — Capacitance Measurement</td>
<td>47</td>
</tr>
<tr>
<td>void TI_CAPT_Custom(Sensor *, uint16_t *); — Capacitance Measurement</td>
<td>48</td>
</tr>
<tr>
<td>void TI_CAPT_Raw(Sensor*, uint16_t*); — Capacitance Measurement</td>
<td>48</td>
</tr>
<tr>
<td>void TI_CAPT_Init_Baseline(Sensor *); — Baseline Tracking</td>
<td>48</td>
</tr>
<tr>
<td>void TI_CAPT_Update_Baseline(Sensor *, uint8_t); — Baseline Tracking</td>
<td>49</td>
</tr>
<tr>
<td>void TI_CAPT_Reset_Tracking (void); — Baseline Tracking</td>
<td>49</td>
</tr>
<tr>
<td>void TI_CAPT_Update_Tracking_DOI (uint8_t); — Baseline Tracking</td>
<td>49</td>
</tr>
<tr>
<td>void TI_CAPT_Update_Tracking_Rate (uint8_t); — Baseline Tracking</td>
<td>50</td>
</tr>
</tbody>
</table>
### uint8_t TI_CAPT_Button(Sensor *)

**Capacitance Measurement**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Pointer to Sensor that defines a button.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outputs</td>
<td>0/1,</td>
</tr>
<tr>
<td>Function</td>
<td>Measure the button. A 0 means that the change in capacitance is less than or equal to the threshold set in the element and 1 means that the change in capacitance has exceeded the threshold.</td>
</tr>
</tbody>
</table>

### element * TI_CAPT_Buttons(Sensor *)

**Capacitance Measurement**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Pointer to Sensor that defines group of elements where each element represents a button.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outputs</td>
<td>Pointer to an element or 0.</td>
</tr>
<tr>
<td>Function</td>
<td>Measure the sensor (buttons) and determine which button, if any, is being touched. This function returns the pointer to the element that exceeds its threshold by the largest margin (% of the maxResponse - threshold). If no button exceeds its threshold, then this function returns a 0.</td>
</tr>
</tbody>
</table>
**uint16_t TI_CAPT_Slider(Sensor *);**  
*Capacitance Measurement*

**Inputs**  
Pointer to Sensor that defines group of elements which makeup slider.

**Outputs**  
Location on the slider; ILLEGAL_SLIDER_WHEEL_POSITION-> No touch;  
0-max -> touch at location where max is defined in the Sensor as the variable points.

**Function**  
Measures the elements within the sensor. This function returns either an invalid number to indicate that no touch was measured or a valid number representing the touch position along the slider.

```c
const struct Sensor group = {
    .numElements = 5,
    .baseOffset = 0,
    // Pointer to elements
    .arrayPtr[0] = &element0,  // point to first element
    .arrayPtr[1] = &element1,
    .arrayPtr[2] = &element2,
    .arrayPtr[1] = &element3,
    .arrayPtr[2] = &element4,
    .points = 100,
    .sensorThreshold = 50
};
```

As shown in Figure 29, the order of the elements within the Sensor should represent the order of the elements along the slider. The first element identified within the sensor position represents the lowest value: the outer edge of the first element in the Sensor array is position 0. The last element represents the largest value: the outer edge of the last element in the array represents the resolution number found in the Sensor (points).

![Figure 29. Slider Example](image)
uint16_t TI_CAPT_Wheel(Sensor *);  — Capacitance Measurement

Inputs
Pointer to Sensor that defines group of elements which makeup wheel.

Outputs
Location on the slider, ILLEGAL_SLIDER_WHEEL_POSITION-> No touch
0-max -> touch at location where max is defined by Sensor structure definition 'points'

Function
Measure the elements within the sensor. This function returns either an invalid number to indicate that no touch was measured or a valid number representing the position along the wheel.

The order of the elements within the Sensor structure should represent the order of the elements around the wheel. The first element identified within the sensor position represents the lowest value: the outer edge of the first element in the Sensor array is position 0. The last element represents the largest value: the outer edge of the last element in the array represents the point where the value wraps around.

Figure 30. Wheel Example

If no element exceeds its threshold (set in the element structure), then this function returns the value ILLEGAL_SLIDER_WHEEL_POSITION, which is defined in the structure.h file.
void TI_CAPT_Custom(Sensor *, uint16_t *); — Capacitance Measurement

**Inputs**
Pointer to Sensor that defines group of elements which makeup a custom interface and the pointer to the array that is updated with the results of the measurement.

**Outputs**
None

**Function**
Measure the change in capacitance relative to the baseline (capacitance history) for each element within the sensor.
The order of the elements within the Sensor structure can be arbitrary but must be understood between the application and configuration. The first element in the array corresponds to the first element within the Sensor structure.

![Figure 31. Custom Slider Example](image)

This function requires the application to allocate an array which this function can fill when called. This type of API is useful when the function of the sensor needs to be controlled within the application layer but the measurement and baseline tracking can still be managed by the library.

void TI_CAPT_Raw(Sensor*, uint16_t*); — Capacitance Measurement

**Inputs**
Pointer to Sensor that defines group of elements which makeup a custom interface and the pointer to the array that is updated with the results of the measurement.

**Outputs**
None

**Function**
Measure the capacitance of each element within the Sensor. This function updates the input array with the timer representation of capacitance.
The order of the elements within the Sensor structure is arbitrary and must be managed by the application and configuration. The first element in the array being passed corresponds to the first element within the Sensor structure.
This function requires that the application allocate an array which this function can fill when called. This type of API is useful when the function of the sensor and baseline tracking needs to be controlled within the application layer but the measurement can still be managed by the library.

void TI_CAPT_Init_Baseline(Sensor *); — Baseline Tracking

**Inputs**
Pointer to Sensor.

**Outputs**
None

**Function**
Measure the sensor and directly place measured values into the associated baseline variables. Using this function loads the measurements into the RAM space for each element within the sensor. Various functions automatically average the current measurement with the existing baseline function and may cause erroneous performance until the tracking algorithm reaches a steady-state value representative of the environment. It is recommended to use this API when the state of the RAM space is unknown, such as on power up.
void TI_CAPT_Update_Baseline(Sensor *, uint8_t);  

Inputs
Pointer to Sensor and the number of measurements to average with baseline.

Outputs
None

Function
Average baseline with number of measurements defined in input. The purpose of this function is to take measurements solely for updating baseline value for each element within the sensor.

void TI_CAPT_Reset_Tracking (void);  

Inputs
None

Outputs
None

Function
Reset the baseline tracking so that the direction of interest is an increase in capacitance. Also reset the tracking rates so that the baseline tracks changes in the direction of interest at the slow setting (01) and changes in capacitance against the direction of interest at the fast setting (00). I.e. track decreases in capacitance at the fast setting and increases in capacitance at the slow setting.

void TI_CAPT_Update_Tracking_DOI (uint8_t);  

Inputs
The direction of interest.

Outputs
None

Function
If the input is true (non-zero), then the direction of interest is an increase in capacitance. If the input is 0x00, then the direction of interest is a decrease in capacitance. In most applications the direction of interest is an increase in capacitance, because the introduction of an object within a field causes an increase in capacitance. In some situations it is beneficial to identify when an object is present and then change the direction of interest to detect when the object is removed. This is typically useful in applications where the object is stationary for long periods of time.
void TI_CAPT_Update_Tracking_Rate (uint8_t);

--- Baseline Tracking

**Inputs**
The rate of how quickly the baseline adjusts to changes in capacitance that are in the direction of interest and against the direction of interest.

**Outputs**
None

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Tracking Rate in Direction of Interest</th>
<th>Tracking Rate Against Direction of Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000b</td>
<td>Very Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>0001 0000b</td>
<td>Slow (Default)</td>
<td>Fast (Default)</td>
</tr>
<tr>
<td>0010 0000b</td>
<td>Medium</td>
<td>Fast</td>
</tr>
<tr>
<td>0011 0000b</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>0000 0000b</td>
<td>Very Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>0100 0000b</td>
<td>Very Slow</td>
<td>Medium</td>
</tr>
<tr>
<td>1000 0000b</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>1100 0000b</td>
<td>Very Slow</td>
<td>Very Slow</td>
</tr>
</tbody>
</table>

**Function**
Update the tracking rates per Table 22.

6 Establishing Measurement Parameters

The measurement parameters, maxResponse, threshold, and sensorThreshold are impacted by timing parameters selected within the sensor definition. Calibration is an iterative process where the sensor parameters are changed to provide the appropriate response before the measurement parameters are selected.

6.1 Measurement Functions

The TI_CAPT_Raw function does not use any of the measurement parameters and can be used to establish a threshold for the TI_CAPT_Custom function. The TI_CAPT_Custom requires a threshold parameter to disable the baseline tracking when one or more elements within a sensor exceed the threshold. It is important to note that with the raw function, an increase in capacitance is represented by an increase in counts with the RC and fRO methods and by a decrease in counts with the RO method.

```c
#include "CTS_Layer.h"

unsigned int raw_data[4];

void main(void)
{
...
while(1)
{
    TI_CAPT_Raw(&group,raw_data);
    __no_operation();   // set breakpoint here
}
}
```
Establishing Measurement Parameters

Table 23. Example Raw Results With RO Method

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>394</td>
<td>435</td>
<td>426</td>
<td>367</td>
</tr>
<tr>
<td>0 (light)</td>
<td>257</td>
<td>424</td>
<td>427</td>
<td>369</td>
</tr>
<tr>
<td>0 (normal)</td>
<td>137</td>
<td>410</td>
<td>428</td>
<td>371</td>
</tr>
<tr>
<td>0 (heavy)</td>
<td>110</td>
<td>304</td>
<td>420</td>
<td>371</td>
</tr>
<tr>
<td>None</td>
<td>390</td>
<td>435</td>
<td>426</td>
<td>367</td>
</tr>
<tr>
<td>1 (light)</td>
<td>367</td>
<td>223</td>
<td>408</td>
<td>367</td>
</tr>
<tr>
<td>1 (normal)</td>
<td>361</td>
<td>165</td>
<td>401</td>
<td>366</td>
</tr>
<tr>
<td>1 (heavy)</td>
<td>226</td>
<td>117</td>
<td>332</td>
<td>365</td>
</tr>
<tr>
<td>None</td>
<td>389</td>
<td>435</td>
<td>425</td>
<td>368</td>
</tr>
<tr>
<td>2 (normal)</td>
<td>382</td>
<td>349</td>
<td>146</td>
<td>341</td>
</tr>
<tr>
<td>None</td>
<td>390</td>
<td>435</td>
<td>426</td>
<td>267</td>
</tr>
<tr>
<td>3 (normal)</td>
<td>388</td>
<td>421</td>
<td>255</td>
<td>111</td>
</tr>
</tbody>
</table>

<sup>(1)</sup> The difference between a light, normal, and heavy press is the surface area. In applications with a finger, as more pressure is applied the end of the finger flattens creating a larger surface area.

From Table 23, the threshold for elements 0 and 1 can be established from the difference between the interaction and no interaction. A good rule of thumb is half the difference. For example, in this configuration to ensure detection of a light touch on sensors 0 and 1, the thresholds are \((394-257)/2\) and \((435-223)/2\), respectively.

### 6.2 Button and Buttons

Defining the threshold value for the TI_CAPT_Button and TI_CAPT_Buttons abstractions can be done with the either the TI_CAPT_Raw or TI_CAPT_Custom functions. The TI_CAPT_Custom function is preferred simply because the information is provided as a magnitude of change instead of a raw value. The TI_CAPT_Custom function measures the magnitude of change from the baseline that is being tracked by the library. The magnitude of change is only returned for the direction of interest. Changes in the opposite direction are represented by a 0. In the following code example, the direction of interest is an increasing capacitance. See the TI_CAPT_Update_Tracking.DOI API for a description on changing the direction of interest.

```c
#include "CTS_Layer.h"

// threshold set to '0' in structure.c
unsigned int delta_data[4];

void main(void)
{
    TI_CAPT_Init_Baseline(&group);
    TI_CAPT_Update_Baseline(&group,30);

    while(1)
    {
        TI_CAPT_Custom(&group,delta_data);
        __no_operation(); // set breakpoint here
    }
}
```
Establishing Measurement Parameters

Table 24. Example Change in Capacitance Results With RO Method

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 (light)</td>
<td>130</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 (normal)</td>
<td>188</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 (heavy)</td>
<td>287</td>
<td>71</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 (light)</td>
<td>14</td>
<td>205</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>1 (normal)</td>
<td>30</td>
<td>288</td>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>1 (heavy)</td>
<td>222</td>
<td>328</td>
<td>91</td>
<td>2</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 (normal)</td>
<td>3</td>
<td>49</td>
<td>292</td>
<td>24</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 (normal)</td>
<td>0</td>
<td>5</td>
<td>52</td>
<td>243</td>
</tr>
</tbody>
</table>

The threshold calculation from Table 24 is similar to that shown with Table 23, 130/2 and 205/2. APIs that use an array of elements, like the TI_CAPT_Buttons API, require the definition of the maxResponse parameter in addition to the threshold. With multiple elements within a sensor, the maxResponse is used to normalize the response of each element and identify which element has the dominant response. The purpose of the normalization is to account for possible differences in element performance. As an example, the maxResponse is simply the result from the heavy interaction. Keep in mind the relationship between threshold and maxResponse as described in Section 3.1.1.

6.3 Sensor Arrays: Wheels and Sliders

With the wheel and slider APIs, the threshold and maxResponse measurement parameters take on slightly different meanings. The threshold represents the minimum response expected as the interaction first 'slides' into the element area. The maxResponse represents the maximum return as the interaction slides across the element. This is typically found to be the center of the element which has the largest area overlap between the element and interaction. The following example describes how to use the custom function to measure the performance of a slider and determine the values for the threshold and maxResponse variables.
Establishing Measurement Parameters

Figure 32. Measurement Example of a Four-Element Sensor

Ideally the geometry of the electrodes results in equivalent non-zero responses for a, c, d, f, h, and k. More importantly, the response should be greater than 10% of the corresponding maximum return, b, e, g, or j.

Table 25. Measurement Example of a Four-Element Sensor

<table>
<thead>
<tr>
<th>Element</th>
<th>threshold</th>
<th>maxResponse&lt;sup&gt;(1)&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>(a + d) / 2</td>
<td>b or (threshold +655), whichever is smaller</td>
</tr>
<tr>
<td>e1</td>
<td>(c + f) / 2</td>
<td>e or (threshold +655), whichever is smaller</td>
</tr>
<tr>
<td>e2</td>
<td>(d + h) / 2</td>
<td>g or (threshold +655), whichever is smaller</td>
</tr>
<tr>
<td>e3</td>
<td>(f + k) / 2</td>
<td>j or (threshold +655), whichever is smaller</td>
</tr>
</tbody>
</table>

<sup>(1)</sup> In some geometry the value within maxResponse is not truly the largest return from the electrode but the return recorded at the center of the electrode. The important criteria are that the neighbors (for a slider or wheel) have equal returns.
If the design prohibits meeting these criteria, then one should consider using the TI_CAPT_Custom function and performing the position calculations within the application layer. If the TI_CAPT_Custom function is used then only the threshold value is required as mentioned earlier.

Wheels and sliders also require a third measurement parameter that is part of the sensor structure, sensorThreshold. As described in Figure 33, the sensorThreshold defines the valid area of the wheel or slider. A good starting value for a wheel is 15. Decreasing this value increases the area, but the accuracy is a function of how closely the interaction is to the center line. Conversely, increasing this value reduces the area of interaction more closely to the center line of the wheel. The slider is a special case of the wheel in that the endpoints present a discontinuity in the sensorThreshold calculation. For sliders it is recommended to set the sensorThreshold to 0, otherwise the ends of the slider can be truncated.

Figure 33. Valid Slider Locations as a Function of the Sensor Threshold
The functions found in CTS_Layer.c and CTS_Layer.h can be divided up into four groups: Status/Baseline Control Register, baseline tracking, measurement functions, and sensor abstraction.

### A.1 Status/Baseline Control Register

A status control register, shown in Figure 34, is provided within the capacitive touch layer and serves as an interface between the application and the library for controlling the baseline tracking. The EVNT and PAST_EVNT fields are not intended to be used by the application but are used within the capacitive touch layer to perform various functions. The DOI, TRADOI, and TRIDOI fields are intended to be accessed by the application and the following APIs are provided for doing so: TI_CAPT_Update_Tracking.DOI() and TI_CAPT_Update_Tracking_Rate().

```c
uint16 ctsStatusReg = 0;
```

#### Figure 34. Status/Baseline Control Register (RAM)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td>7-6</td>
<td>TRADOI</td>
<td>Tracking rate against direction of interest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = Very slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = Slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Fast</td>
</tr>
<tr>
<td>5-4</td>
<td>TRIDOI</td>
<td>Tracking rate in direction of interest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = Very slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = Slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Fast</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>PAST_EVNT</td>
<td>Past event. An event occurred on the previous scan.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No event occurred on the previous scan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = An event occurred on the previous scan</td>
</tr>
<tr>
<td>1</td>
<td>DOI</td>
<td>Direction of interest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Decreasing capacitance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Increasing capacitance</td>
</tr>
<tr>
<td>0</td>
<td>EVNT</td>
<td>Event. One of the elements in the group has detected a threshold crossing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No event has occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = An event has occurred</td>
</tr>
</tbody>
</table>
A.2 Baseline Tracking

Baseline tracking is one of the main features of the capacitive touch layer. The baseline capacitance is followed or tracked to account for any environmental changes that impact the mechanism used to make the capacitance measurement. This includes but is not limited to \( V_{CC} \), temperature, and humidity.

A.2.1 Direction of Interest

The representation of an increase in capacitance is an increase in counts for the RC and FastRO methods while a decrease in counts in the RO method. The purpose of identifying a direction of interest is to establish if the application is looking for an increase or decrease in capacitance. In most human interface applications the direction of interest is an increase in capacitance. The presence of a finger or touch increases the capacitance of an element. Increases in capacitance can also be caused by environmental factors but the assumption is that these changes are relatively slow in comparison to the interaction with a person. Changes in capacitance that are in the direction of interest but are not large enough in magnitude to exceed the threshold may be changes due to the environment. This requires an update in the base capacitance. To insure that these changes are not due to a slow moving object, it is recommended to make adjustments in the direction of interest very slowly. The tradeoff in choosing the adjustment rate is accounting for slow moving objects and rapid environmental changes.

Capacitance changes that are against the direction of interest typically represent only a change in the environment. Because the change can be attributed to the environment without any ambiguity the baseline can be adjusted more dramatically to account for the shift.

A.2.2 Examples of Direction of Interest

An application needs to detect when a block of wood is in place and then removed. The block is typically left in place for several days. The direction of interest is an increase in capacitance to identify when the block is in place and then the direction of interest is changed to a decrease in capacitance to identify when the block has been removed. Once the block is in place any additional increase in capacitance is treated as a change against the direction of interest and the baseline is updated accordingly. In the same way, after the block is removed, if there is a decrease in capacitance this is treated as a change against the direction of interest.

A.2.3 Updating the Baseline Capacitance

Figure 35 shows how baseline updates occur when the change in capacitance is against the direction of interest or under certain conditions when the change of capacitance is in the direction of interest.

A change in capacitance that is in the direction of interest must meet two criteria before it is applied to the baseline tracking. First it must be less than the threshold and second no event within the sensor can occur. A change in capacitance that exceeds the threshold indicates an event. When an event occurs it is possible that the other elements within the sensor are excited even if only by a very small amount. The past event flag, PAST_EVNT, indicates that one of the elements within a sensor has experienced a threshold crossing. Therefore if the PAST_EVNT flag is true within a sensor it is important to suspend baseline updates in the direction of interest.

When the change is against the direction of interest it is limited in magnitude to the threshold/2 before being applied to the baseline.

Table 27 and Table 28 show how the current capacitance measurement and the baseline are weighted to determine the new baseline. The default setting of the library are the 'fast' setting for changes against the direction of interest and 'slow' for changes in the direction of interest.
Change in capacitance in Direction of Interest

If Change in capacitance < threshold:

- Capacitance Change is in direction of interest but does not exceed threshold and there is no recent threshold event: Update Baseline per TRADOI setting
- Capacitance Change opposite of direction of interest: Update Baseline per TRADOI setting

Set Event Flag and Past event flag in Status Register

Limit Change in Capacitance to threshold/2

Figure 35. Base Capacitance Update

Table 27. Tracking Settings Against Direction of Interest

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>RO (meas &lt; base)</th>
<th>RC, Fast RO (meas &gt; base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Fast</td>
<td>base = meas/2 + base/2</td>
<td>base = meas/2 + base/2</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
<td>base = meas/4 + 3*base/4</td>
<td>base = meas/4 + 3*base/4</td>
</tr>
<tr>
<td>1</td>
<td>Slow</td>
<td>base = meas/64 + 63*base/64</td>
<td>base = meas/64 + 63*base/64</td>
</tr>
<tr>
<td>0</td>
<td>Very Slow (Default)</td>
<td>base = meas/128 + 127*(base/128)</td>
<td>base = meas/128 + 127*(base/128)</td>
</tr>
</tbody>
</table>

Table 28. Tracking Settings in Direction of Interest

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>RO (meas &lt; base)</th>
<th>RC, Fast RO (meas &gt; base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Fast</td>
<td>base = meas/2 + base/2</td>
<td>base = meas/2 + base/2</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
<td>base = meas/4 + 3*base/4</td>
<td>base = meas/4 + 3*base/4</td>
</tr>
<tr>
<td>1</td>
<td>Slow (Default)</td>
<td>base = base – 2</td>
<td>base = base + 2</td>
</tr>
<tr>
<td>0</td>
<td>Very Slow</td>
<td>base = base – 1</td>
<td>base = base + 1</td>
</tr>
</tbody>
</table>
A.3 Measurement Functions

A.3.1 Delta Measurement + Base Capacitance Tracking: Custom API Call

The 'custom' API measures the change in capacitance for the elements of a given structure. The inputs for the custom API function are the pointer to the sensor and a pointer to the first element of the array in which the capacitance change is recorded. The custom API call measures the capacitance of each element with the 'raw' function.

A.3.1.1 Delta Calculation

The HAL definition (hal_definition) and the direction of interest determine the delta calculation. The hal_definitions are arranged so that all values less than 64 are methods who's count values directly relate to the change in capacitance (that is, an increase in counts means an increase in capacitance) when the hal_definition greater than 64 relate inversely (that is, an increase in capacitance results in a decrease in counts). With the RC and Fast Scan RO methods an increase in capacitance is indicated by an increase in counts. Conversely with the RO method, an increase in capacitance is indicated by a decrease in counts.

The delta calculation performed within the custom API results in either a 0 or non-zero value. The non-zero value is simply the difference between the measured capacitance and the baseline capacitance of a given element. A 0 value indicates that the change in capacitance is opposite (against) the direction of interest.

![Figure 36. Delta Measurement Block Diagram](image-url)
A.3.2 Raw Capacitance Measurement: Raw API Call

The single purpose of the RAW measurement function is to call the appropriate HAL function based upon the user configuration. This function updates the RAM variables provided within the function call, which is used by the higher level calling function.

The ‘Raw’ feature calls the appropriate HAL function. The HAL definition for a group of elements is found in the sensor structure.

sensor0.halDefinition

The hal_definition represents a combination of MSP430 peripherals to accomplish the cap touch function.

Table 29. Example HAL Definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Number</th>
<th>Measurement Hardware</th>
<th>Measurement Timer</th>
<th>Gate Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC_PAIR_TA0</td>
<td>RC</td>
<td>1</td>
<td>Digital I/O</td>
<td>TimerA0</td>
<td>SW</td>
</tr>
<tr>
<td>fRO_PINOSC_TA0_SW</td>
<td>fRO</td>
<td>25</td>
<td>Digital I/O</td>
<td>SW</td>
<td>TimerA0</td>
</tr>
<tr>
<td>fRO_COMPB_TA1_TA0</td>
<td>fRO</td>
<td>31</td>
<td>COMP_B</td>
<td>TimerA0</td>
<td>TimerA1</td>
</tr>
<tr>
<td>RO_COMPAp_TA0_WDTp</td>
<td>RO</td>
<td>64</td>
<td>COMP_A+</td>
<td>TimerA0</td>
<td>WDTp</td>
</tr>
<tr>
<td>RO_PINOSC_TA0_WDTp</td>
<td>RO</td>
<td>65</td>
<td>Digital I/O</td>
<td>TimerA0</td>
<td>WDTp</td>
</tr>
</tbody>
</table>

A.4 Sensor Abstractions

A.4.1 Button, Buttons

uint8_t TI_CAPT_Button(Sensor *);

Inputs: pointer to Sensor that defines button

Outputs: 0/1,

Function: Measure the button. A 0 means that the change in capacitance is less than or equal to the threshold set in the Sensor and 1 means that the change in capacitance has exceeded the threshold.

Figure 37. Single Button Algorithm
Element * TI_CAPT.Buttons(Sensor *);

Inputs: pointer to Sensor that defines group of elements where each element represents a button

Outputs: pointer to an element structure

Function: This function return is the structure pointer to the element that exceeds its threshold by the
largest margin: normalized to (maxResponse - threshold value) for each element. If no button exceeds its
threshold (set in the element structure), then this function returns a 0 or ‘Null Pointer’.

![Diagram](image)

**Figure 38. Array of Buttons Algorithm**

### A.4.2 Slider and Wheel

A wheel or slider is a sensor type consisting of an array of elements. The sensor is divided into a number
of points defined by the user. The orientation of the array (first to last) is left to the interpretation of the
application. From the perspective of the library the first element within the array definition corresponds to
the 0 value on the slider and the last element corresponds to the number of points defined by the user.

The algorithm for the slider and wheel functions is shown in **Figure 39**. The TI_CAPT_Custom function is
used to measure the change in capacitance for each element defined in the sensor. This function also
updates the baseline tracking and the event flag status (see Section A.3.1)

An additional detection mechanism is provided at the sensor level for wheels and sliders. The event flag is
the means to determine a threshold crossing at the element level while the sensorThreshold variable
provides a threshold at the sensor level. The intent of this mechanism is to distinguish a genuine
interaction with the sensor from an unintentional interaction that may activate only one element.

Finally the slider and wheel functions calculate the position where the interaction (touch) takes place. The
slider and wheel require four types of configuration parameters to present a location. These four
parameters are the number of resolvable points, the sensor level threshold, the element level threshold for
each element, and the maximum response for each element.
Start

Update delta array: custom function

Event Flag

Y

index = dominant_element_find();

Sum Response of dominant Element and neighbors

Sum_Response > sensorThreshold

End: Return ILLEGAL_SLIDER_WHEEL_POSITION

position = index*(points/numElements) + points/numElements/2

Y

index == 0

Y

Special Case Beginning

Y

index == num_elements -1

Special Case End

position += deltaCnt[index+1]*(points/numElements)/100

position -= deltaCnt[index-1]*(points/numElements)/100

End: Return position

Figure 39. Slider and Wheel Algorithm
A.4.2.1 Slider Detection

The slider sensorThreshold is a compared with the response of the dominant element and its neighbors. As shown in Figure 40, The endpoints are a special case which requires a comparison of only the end element (the dominant element) and the one neighbor.

![Diagram of Slider Threshold Detection](image)

As the interaction moves (slides) beyond the center of the last element the contribution from the neighbor goes to 0 and the threshold of the sensor is only a function of the last element. Therefore the sensorThreshold defines how far the finger can deviate from the center position of the ends and still be a part of the slider. As long as the sensor response exceeds the sensorThreshold, the position is calculated.

A.4.2.2 Slider Position

Calculation of the slider position is predicated upon passing the sensor threshold criteria. If the criterion is not met, then the function simply returns a predefined value to indicate no interaction was detected. When there is a valid interaction, the function determines the position from the response of the dominant element and its nearest neighbors. The dominant element function determines the middle element for establishing a ‘base’ position while one or two neighboring elements are used to pull or weight the final position. In the example in Figure 41, the slider has 64 positions and there are four elements in the slider array.
In the special case where the dominant element is either the beginning or end element of the slider, then only the nearest neighbor is used to weight or influence the position.

Figure 41. Slider and Wheel Process Middle Algorithm

Figure 42. Slider Algorithm: Beginning of Slider
A.4.2.3 Wheel Detection

The wheel sensorThreshold is a compared with the response of the dominant element and its neighbors (summation of x-1, x, and x+1). The endpoints are a special case which requires the 'wrap around' to be accounted for, see Figure 44. Once the normalized responses of these three elements are added then the value is compared with the sensorThreshold. If the threshold is exceeded, then the function continues on to calculate the position.
A.4.2.4  Wheel Position

As previously mentioned the wheel is simply a special case of the slider. Additional handling needs to be put in place to account for the 'wrap around' from the end of the array back to the beginning. Figure 45 and Figure 46 show the algorithm for calculating the position when the dominant element is the beginning and end elements of the array,

```
START

position += deltaCnt[1]*(points/numElements)/100

position -= deltaCnt[numElements-1]*(points/numElements)/100

position < 0

Y

position += (points+1)

END
```

Figure 45. Wheel Algorithm: Beginning
Figure 46. Wheel Algorithm: Ending
### A.5 Dominant Element Identification

The identification of a threshold crossing actually takes place in the base capacitance update function (see Section A.3.1). When a threshold crossing event has occurred then the following is used to determine the dominant element within the sensor structure and scale the response to a range from 0 to 100. A zero indicates that the response is equal to the threshold, and a value of 100 indicates a response equal to the maxResponse.

\[
\text{percentDelta} = 0
\]

\[
\text{deltaCnt}[i] > \text{threshold}
\]

\[
i = 0
\]

\[
i < \text{num} \_\text{sensors}
\]

\[
i++
\]

\[
\text{percentDelta} = \frac{100 \times (\text{delta}[i] \pm \text{threshold})}{\text{maxResponse} \pm \text{threshold}}
\]

\[
\text{deltaCnt}[i] = \text{maxResponse}
\]

\[
\text{deltaCnt}[i] \geq \text{percentDelta}
\]

\[
\text{percentDelta} = \text{deltaCnt}[i];
\]

\[
\text{dominantElement} = i;
\]

\[
\text{End:}
\]

\[
\text{Return dominantElement}
\]

---

**Figure 47. Dominant Element Identification Function**
## Revision History

### Changes from August 10, 2014 to August 20, 2015

<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added missing cross reference to Table 19</td>
<td>41</td>
</tr>
<tr>
<td>Added missing cross reference to Table 20</td>
<td>42</td>
</tr>
<tr>
<td>Added missing cross reference to Table 23</td>
<td>51</td>
</tr>
<tr>
<td>Added missing cross references to Table 24 and Table 23</td>
<td>52</td>
</tr>
<tr>
<td>Updated Figure 34 and Table 26, Status/Baseline Control Register (RAM)</td>
<td>55</td>
</tr>
</tbody>
</table>

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.
**IMPORTANT NOTICE**

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that TI is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer’s risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio</td>
<td>Automotive and Transportation</td>
</tr>
<tr>
<td>Amplifiers</td>
<td>Communications and Telecom</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Computers and Peripherals</td>
</tr>
<tr>
<td>DLP® Products</td>
<td>Consumer Electronics</td>
</tr>
<tr>
<td>DSP</td>
<td>Energy and Lighting</td>
</tr>
<tr>
<td>Clocks and Timers</td>
<td>Industrial</td>
</tr>
<tr>
<td>Interface</td>
<td>Medical</td>
</tr>
<tr>
<td>Logic</td>
<td>Security</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Space, Avionics and Defense</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Video and Imaging</td>
</tr>
<tr>
<td>RFID</td>
<td></td>
</tr>
<tr>
<td>OMAP Applications Processors</td>
<td></td>
</tr>
<tr>
<td>Wireless Connectivity</td>
<td></td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2015, Texas Instruments Incorporated