Capacitive Touch Library

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 2xx and 5xx MSP430 families and the peripheral sets within those families. This document explains the capacitive touch library's configuration and use.

The software library described in this document can be downloaded from http://focus.ti.com/docs/toolsw/folders/print/capsenselibrary.html.

Topic **Page**

1 Introduction

The Capacitive Touch Software Library is a flexible software base that quickly enables one of several different capacitive touch sensing algorithms on the MSP430 microcontroller platform. While each algorithm has its unique features, there are often a number of application-specific factors that would accommodate one implementation more easily than another. One of the main purposes of this library is to provide an interface with several MSP430 series and the different peripheral sets within those series.

The library provides several layers or degrees of abstraction. The higher levels of abstraction provide standard controls for faster and easier development while the lower levels allow for customization and unique controls.

To use the library, it is important to have a basic understanding of the measurement methods, how to configure the library for a particular method, how peripheral resources are used, and the API function calls.

The associated code[1] is intended to be a springboard for developing capacitive touch and other capacitive measurement solutions. The feature set accommodates a wide variety of applications, all of which may not be required for a specific application. The source code is provided and customers are encouraged after creating a working application to remove sections of code that are not used. Additionally, low-power features are not typically used in the library because of the need to define the ISR, which may conflict with an existing application definition. Again, as applications allow for shared ISR functionality, customers are encouraged to update the source code provided to make the most of the low-power capabilities of the MSP430.

2 Implementations

For the implementations described in this document, the fundamental principle is that two independent timing domains are compared. One domain is fixed, and the other is variable as a function of the capacitance.

2.1 Relaxation Oscillator (RO)

The relaxation oscillator method counts the number of relaxation oscillator cycles within a fixed period (gate time), as shown in Figure 1.

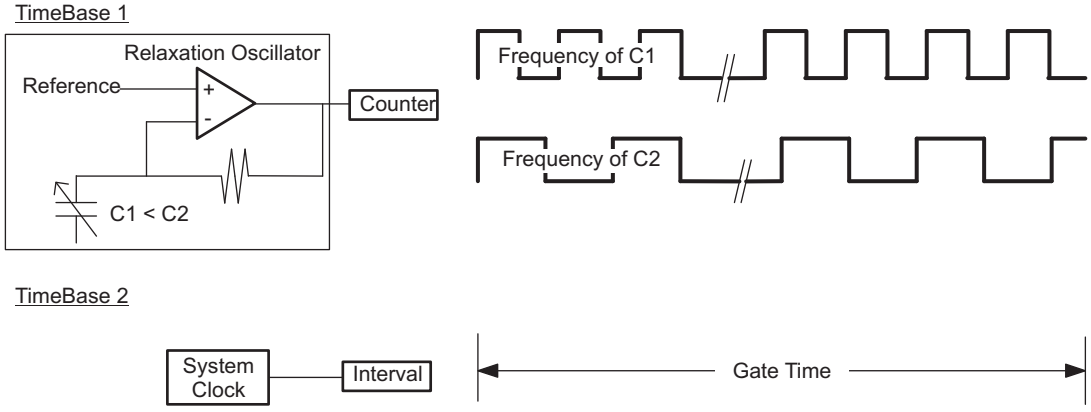


Figure 1. Relaxation Oscillator Measurement

The relaxation oscillator can be realized with a comparator or the PinOsc feature found on several MSP430 devices. The frequency of the oscillation is function of the resistance and capacitance of the circuit. The capacitance is the intended variable and increases with a touch. In the time domain the rise and fall time are increased, but in the frequency domain the frequency is reduced. With an increase in capacitance the number of relaxation oscillator cycles decreases within the fixed gate time.

(1) The software library described in this document can be downloaded from http://focus.ti.com/docs/toolsw/folders/print/capsenselibrary.html.

The naming convention for the RO method in the library identifies the relaxation oscillator mechanism, the timer used to measure or count oscillations, and the timer used to define the gate period (see Table 1).

Table 1. Relaxation Oscillator Naming Convention

Name	RO Mechanism	Counter	Gate Period
RO_XXX_YYY_ZZZ	XXX	YYY	ZZZ
RO_COMPAp_TA0_WDTp	Comparator A+	Timer_A0	Watchdog timer (interval mode)
RO_COMPB_TA1_WDTA	Comparator B	Timer_A1	Watchdog timer (interval mode)
RO_Pinosc_TA0(1)	Pin Oscillator	Timer_A0	'n' ACLK periods

(1) The RO_PINOSC_TA0 is a special case that takes advantage of the internal connection between ACLK and the Timer_A0 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)

The RC method is the reciprocal of the RO method. As shown in Figure 2, the gate time is now variable, a function of the capacitance, and the oscillator being counted is fixed.

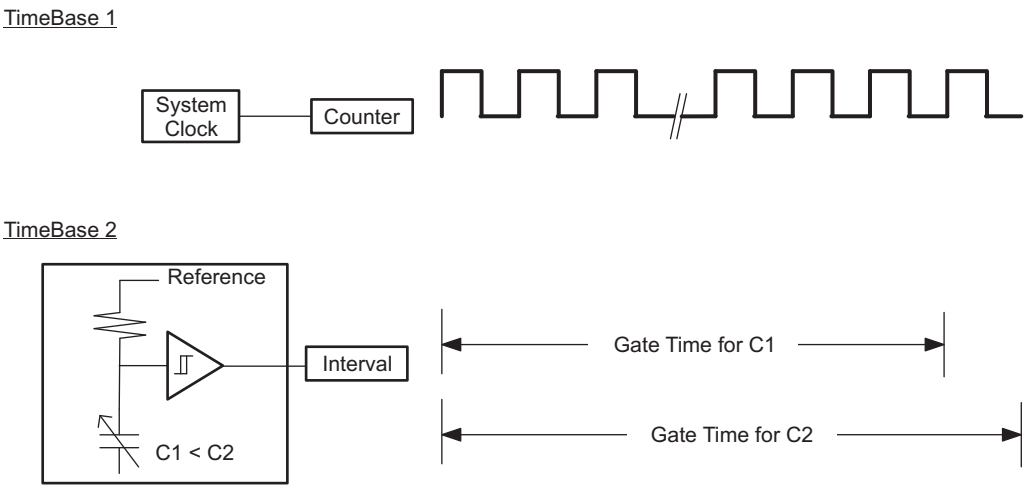


Figure 2. Resistor-Capacitor Time Constant Measurement (RC)

The RC method can be realized with any MSP430. The gate time is defined by how long it takes to charge and discharge the capacitance to the port V_{IT+} and V_{IT-} levels. During this variable gate time the number of fixed oscillator cycles is counted. An increase in capacitance (a touch) would result in an increase in the gate time and consequently an increase in the number of cycles counted. The naming convention is simple since only the timer that counts the fixed oscillation needs to be identified: RC_PAIR_TA0.

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 3, the oscillator frequency being counted is still fixed.

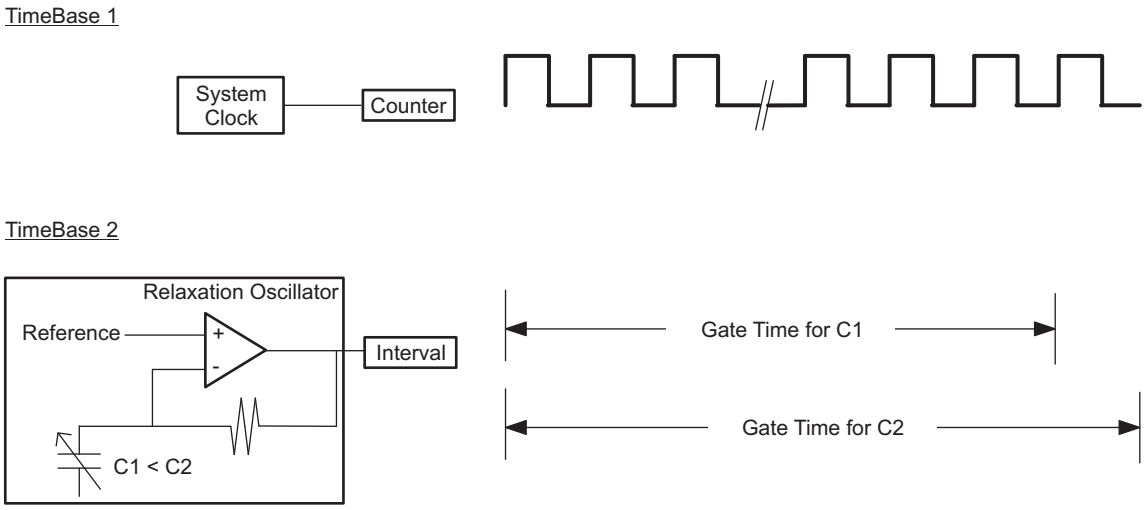


Figure 3. Fast Scan RO Measurement

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

Table 2. Relaxation Oscillator Naming Convention

Name	RO Mechanism	Counter	Gate Period
fRO_XXX_YYY_ZZZ	XXX	YYY	ZZZ
fRO_COMPB_TA1_SW	Comparator B	Timer_A1	Software loop
fRO_PINOSC_TA0_SW	Pin Oscillator	Timer_A0	Software loop

As the name implies, the purpose of the fRO method is provide fast scan rates; faster than the RO method with similar sensitivity (how small a capacitive change can be resolved). With the RO method, sensitivity is a function of the gate period—increasing the gate period increases sensitivity. The negative consequence of increasing gate time is decreased scan rates: more time is spent during a single measurement. To maintain the sensitivity of the fRO method, the fixed clock rate must provide enough resolution for the given changes in gate time. This typically means a much faster clock source and higher power consumption during the shorter gate time.

3 Configuration

There are two files that serve as the primary means to configure the library: structure.c and structure.h. structure.c includes all of the definitions of the elements and the sensors (groups of elements). structure.h 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 Definitions

A capacitive measurement element is a singular structure, whose capacitance represents an event: a touch, change in humidity, change in dielectric, etc. An element can be used individually, for example as a button, or combined with other elements to create sensor; keypad, wheel, or slider, shown in Figure 4.

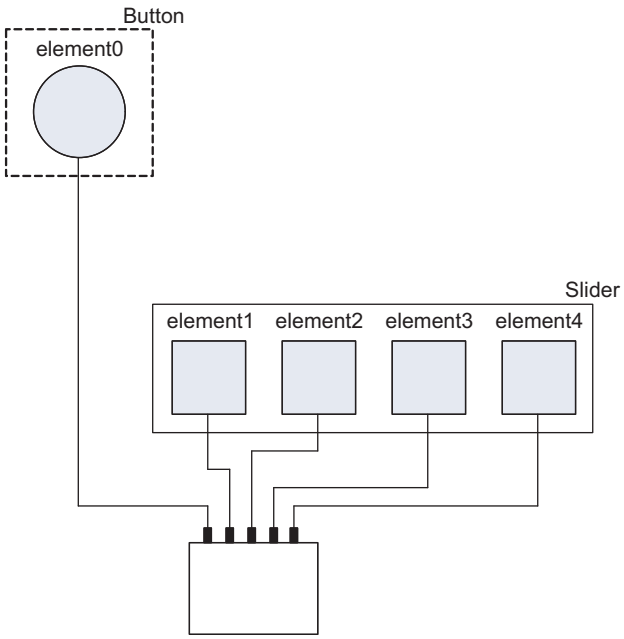


Figure 4. Elements

The element definitions belong to one of two categories; port definition or measurement parameter. The ports definitions include digital IO peripheral registers, comparator peripheral registers, and bit definitions. The measurement parameters include the threshold and maximum signal response (maxResponse) of an element for a given measurement implementation. The port definition can be completed by simply reading the schematic while the measurement implementation requires testing (see Section 6). Establishing the correct measurement parameters for a given measurement implementation calibrates the elements.

3.1.1 Common Definitions

InputBits is one common definition that can represents the bit y in the GPIO definition Px.y or the comparator input mux for either COMPA+ or COMPB solutions.

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 5 shows the relative relationship between measurement parameters, threshold and maxResponse, in a buttons application. 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.

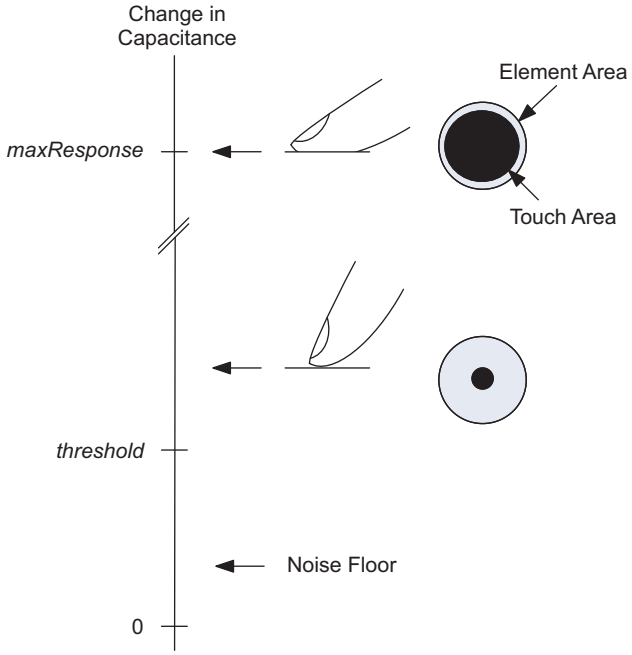


Figure 5. Element Measurement Parameters: Buttons Example

3.1.1.1 Comp_A+ Implementations

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

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

```
Const struct Element element_name = {
              .inputBits = P2CA2, // CA2
              .threshold = 100,
              .maxResponse = 200
};
```

3.1.1.2 Comp_B Implementations

InputBits identify the CBIMSEL bits in the CBCTL0 register.

15	14	13	12	11	10	9	8
CBIMEN	**Reserved**			**CBIMSEL**			
rw-0	r-0	r-0	r-0	rw-0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
CBIPEN	**Reserved**			**CBIPSEL**			
rw-0	r-0	r-0	r-0	rw-0	rw-0	rw-0	rw-0

CBIMEN	Bit 15	Channel input enable for the V– terminal of the comparator. 0 Selected analog input channel for V– terminal is disabled. 1 Selected analog input channel for V– terminal is enabled.
Reserved	Bits 14-12	Reserved
CBIMSEL	Bits 11-8	Channel input selected for the V– terminal of the comparator if CBIMEN is set to 1.
CBIPEN	Bit 7	Channel input enable for the V+ terminal of the comparator. 0 Selected analog input channel for V+ terminal is disabled. 1 Selected analog input channel for V+ terminal is enabled.
Reserved	Bits 6-4	Reserved
CBIPSEL	Bits 3-0	Channel input selected for the V+ terminal of the comparator if CBIPEN is set to 1.

Figure 6. Comp_B Control Register 0 (CBCTL0)

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

3.1.1.3 PinOsc Implementations

**inputPxselRregister* and **inputPxsel2Register* identify the appropriate registers that must be configured for the pin oscillator method. These registers, in conjunction with *inputBits*, configure the structure.

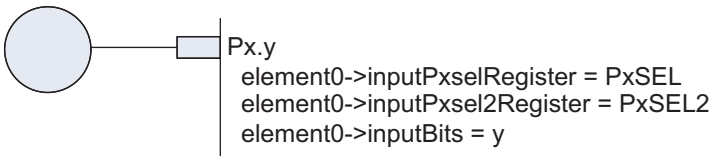


Figure 7. PinOsc Port Parameters

```
const struct Element right = {

              .inputPxselRegister = (uint8_t *)&P2SEL,
              .inputPxsel2Register = (uint8_t *)&P2SEL2,

              .inputBits = BIT3,
              .maxResponse = 400,
              .threshold = 50
};
```

3.1.1.4 RC Implementations

The RC implementation 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 address, and input address. These registers, in conjunction with *inputBits*, configure the input portion of the structure.

referencePxdirRegister and referencePxoutRegister identify the port direction and output address. These registers, in conjunction with *referenceBits*, configure the reference portion of the structure.

One feature of this description is that the ports can share two different functions. That is, the functions can 'flip' so that the reference becomes the input and the input becomes the reference to measure the other electrode connected to the reference input. This is shown in Figure 8 and the following code example.

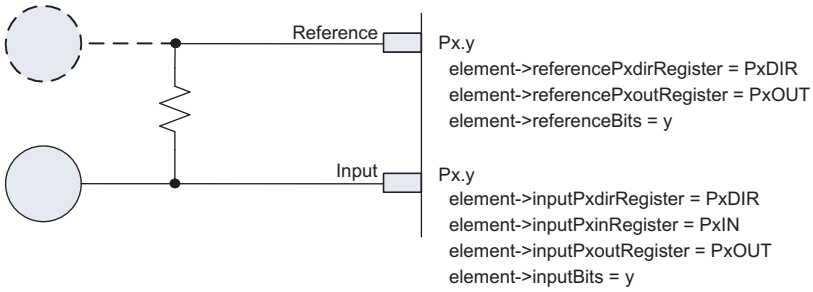


Figure 8. RC I/O Parameters

```
//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 Definitions

The sensor can be a group of independent elements such as a keypad, or it can be a group of elements functioning as one sensor such as a wheel or slider. The sensor definition includes all of the applicable elements, the common mechanism 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 (see Figure 9) and the sensitivity of the sensor.

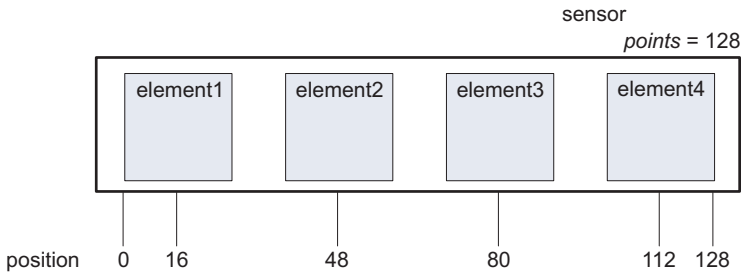


Figure 9. Sensor Example

3.2.1 Common Definitions

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.

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

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

Table 3. baseOffset Description

Sensor	baseOffset	Element	RAM Address
Slider0	0	element0	baseCnt[0]
		element1	baseCnt[1]
		element2	baseCnt[2]
		element3	baseCnt[3]
Slider1	4	element4	baseCnt[4]
		element5	baseCnt[5]
		element6	baseCnt[6]
		element7	baseCnt[7]

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 defines either the gate timer source or the measurement clock source depending upon the implementation (halDefinition). In RC and fRO implementations, 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 when the timer peripheral provides an input timer to the input clock source. This applies to timer peripherals only and not to the watchdog timer peripherals.

accumulationCycles defines the gate time for various implementations. Typically, the accumulationCycles represents the number of times a measurement is repeated but, in the case of a watchdog timer being 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 4 lists the different implementations currently supported.

Table 4. halDefinition Description

halDefinition	Description
RO_COMPAp_TA0_WDTp RO_COMPAp_TA1_WDTp	Relaxation oscillator implemented with COMPA+ peripheral. The gate time is fixed and defined by the WDT+ peripheral set to interval mode. The capacitance is represented by the number of RO cycles counted by Timer_A0/A1 during the fixed gate time (see Section 3.2.2.1).
RO_PINOSC_TA0_WDTp	Relaxation oscillator implemented with Digital IO peripheral(1), Timer_A0 is used to measure frequency of oscillator, and the WDTp is used to set the gate time (see Section 3.2.2.2).
RO_PINOSC_TA0	Relaxation oscillator implemented with Digital IO peripheral , Timer_A0 is used to measure frequency of oscillator, and the ACLK source is used to set the gate time (see Section 3.2.2.4).
RO_COMPB_TA0_WDTA RO_COMPB_TA1_WDTA	Relaxation oscillator implemented with COMPB peripheral, Timer_A0/A1 is used to measure frequency of oscillator, and the WDTA peripheral is used to set the gate time (see Section 3.2.2.2).
RC_PAIR_TA0	Measure RC time constant with Timer_A0. The gate time is variable and changes with the charge/discharge time. A software loop is used to establish the number of charge and discharge cycles that define the gate time. The capacitance is represented by the number of timer counts in Timer_A0 for the gate time. Typically TA0 is sourced from a high frequency clock (SMCLK) for improved sensitivity. The capacitive element is charged and discharge with the other IO defined in the pair (see Section 3.2.3.1).
fRO_COMPAp_TA0_SW fRO_COMPAp_TA1_SW	Fast Scan Relaxation oscillator implemented with COMPA+ peripheral. The gate time is variable and changes with the period of the relaxation oscillator. The Timer_A0/A1 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.
fRO_PINOSC_TA0_SW	Measure the time with Timer_A0, Relaxation oscillator implemented with Digital IO peripheral , several oscillations are counted in software to establish the gate time (see Section 3.2.4.2).
fRO_COMPB_TA0_SW fRO_COMPB_TA1_SW	Fast Scan Relaxation oscillator implemented with COMPB peripheral. The gate time is variable and changes with the period of the relaxation oscillator. The Timer_A0/A1 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.
fRO_COMPAp_SW_TA0	Fast Scan Relaxation oscillator implemented with COMPA+ peripheral. The gate time is variable and changes with the period of the relaxation oscillator. A software loop is used to establish the number of oscillations that define the gate time. The capacitance is represented by the number of timer counts in Timer_A0 for the gate time. Typically TA0 is sourced from a high frequency clock (SMCLK) for improved sensitivity.

(1) The digital I/O functionality described is only found on devices with the pin oscillator feature.

3.2.2 Definitions for the Relaxation Oscillator (RO) Method

3.2.2.1 RO_COMPAp_TAx_WDTp

The relaxation oscillator comprises the Comp_A+ module, a reference, and RC filter. The reference is connected to the noninverting 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. The 'C' in the RC filter is the capacitive element being measured.

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.

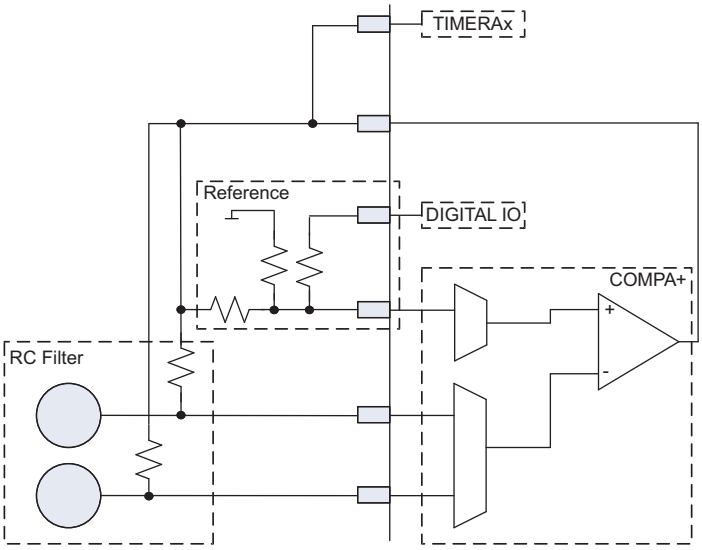


Figure 10. RO_COMPAp_TAx Schematic Description

3.2.2.1.1 Port Parameters

In the element structure, the comparator input is defined for each element. At the sensor level (see Figure 11), the comparator reference input is defined as well as the reference port, comparator output, and timer input.

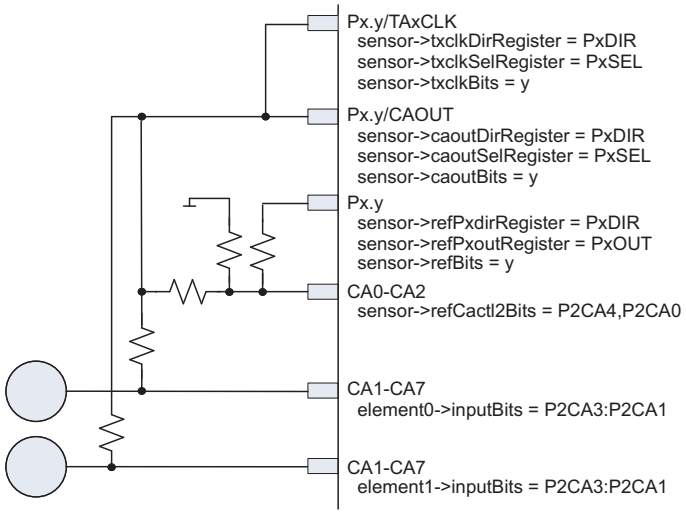


Figure 11. Port Parameters for RO_COMPAp_TAx Implementation

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

txclkDirRegister and *txclkSelRegister* identify the port direction address and port selection address. The variable *txclkBits* defines the bit(s) that are to be set/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 these devices, the value *txclkSel2Register* must also be defined (either as P1SEL2 or P2SEL2).

The *refPxDirRegister*, *refPxOutRegister*, and *refBit* variables define the pullup portion of the external reference circuit shown in Figure 11. These bits provide the mechanism to turn on and turn off the reference for power savings. *refPxDirRegister* and *refPxOutRegister* identify the port direction address and port output address. The variable *refBits* defines the bit(s) that are to be set/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 be applied only to the positive input via CA0, CA1, or CA2. This is represented as P2CA0, P2CA4, and P2CA0+P2CA4, respectively.

capdBits defines the I/Os that 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).

3.2.2.1.2 Timing Parameters

The two timing parameters define the WDTp interval which 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 5. Watchdog Timer Source Select Definitions

Definition	Value	Source
GATE_WDT_ACLK	0x0004	ACLK
GATE_WDT_SMCLK	0x0000	SMCLK

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 6. Watchdog Timer+ Interval Select Definitions

Definition	Value	Interval (s)
WDTp_GATE_32768	0x0000	32768/source
WDTp_GATE_8192	0x0001	8192/source
WDTp_GATE_512	0x0002	512/source
WDTp_GATE_64	0x0003	64/source

Figure 12 shows how the common sensor parameters measGateSource and accumulationCycles are used to select the gate time.

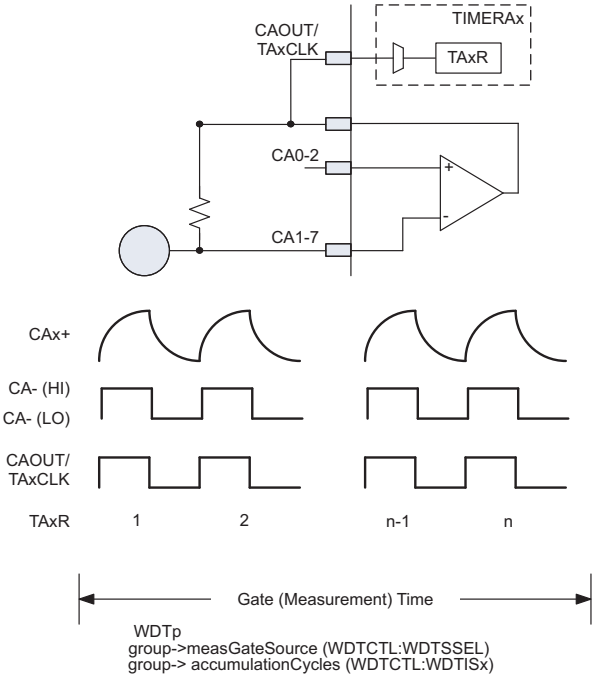


Figure 12. Timing Parameters: Example WDT+

The following example shows a sensor made of four elements. Each element is measured with the RO method for a period of 512/SMCLK.

```
const struct Sensor slider =
{
                  .halDefinition = RO_COMPAp_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,  // SxSEL
                  .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_TAx_WDTA

The RO_COMPB_TAx_WDTA definition 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 Timer_A peripheral, as shown in Figure 13.

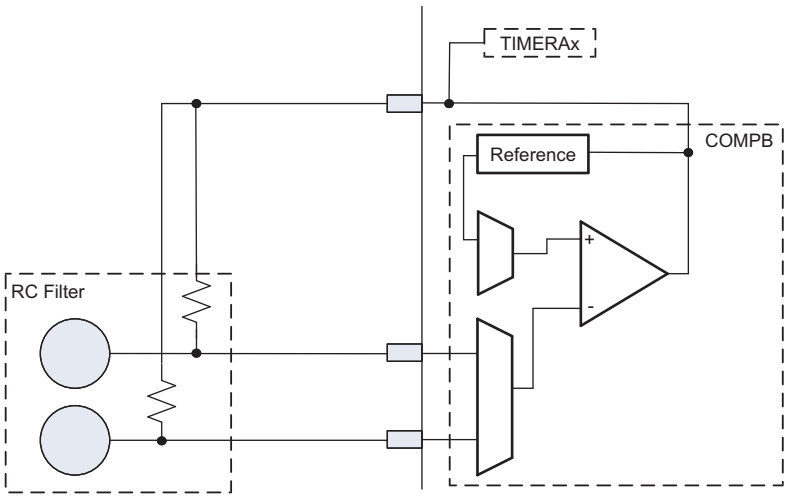


Figure 13. RO_COMPB Schematic

3.2.2.2.1 Port Parameters

The different implementation of the Comp_B solution requires an alternative set of parameters for the sensor definition, shown in Figure 14.

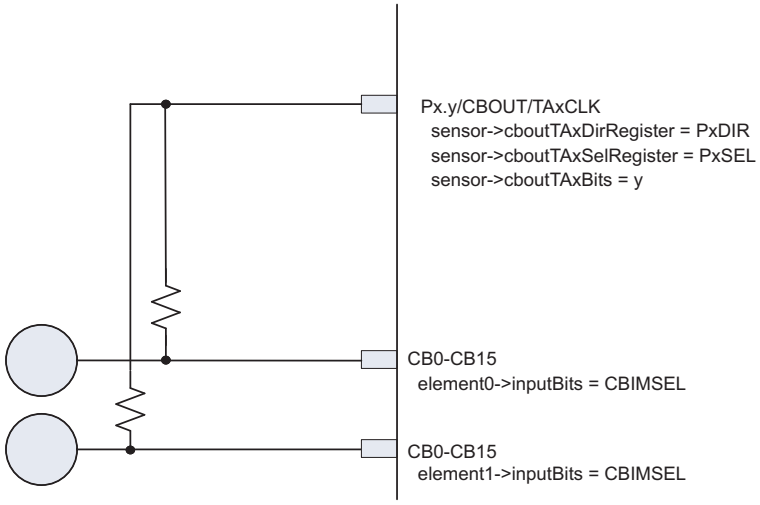


Figure 14. RO_COMPB Port Parameters

cboutTAxRegister and *cboutTAxRegister* identify the port direction address and port selection address. The variable *cboutTAxBits* defines the bit(s) that are to be set/reset within the direction and selection register to select the CBOUTx output and TxCLK input function of the port. Note that these ports share the same I/O on 5xx family devices.

cbpdBits disables the Digital I/O function on the port pins that are 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)

3.2.2.2.2 Timing Parameters

The two timing parameters define the WDTA interval which is the gate time for the RO_COMPB_TAx_WDTA 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 7. Watchdog Timer_A Source Select Definitions

Definition	Value	Source
GATE_WDTA_SMCLK	0x0000	SMCLK
GATE_WDTA_ACLK	0x0020	ACLK
GATE_WDTA_VLO	0x0040	VLO
GATE_WDTA_XCLK	0x0060	XCLK

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

Table 8. Watchdog Timer_A Interval Select Definitions

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

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.

```
const struct Sensor sliderA =
               {
                  .halDefinition = RO_COMPB_TA0_WDTA,
                  .numElements = 4,
                  .baseOffset = 0,
                  .cbpdBits = (BITC+BITD+BITE+BITF),
                  // Pointer to elements
                  .arrayPtr[0] = &element0,  // point to first element
                  .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 TA0 or between CBOUT and TA1 and, in some cases, both. The description and parameters are the same for both TA0 and TA1, with the exception of the HAL definition name.

3.2.2.3 RO_PINOSC_TA0_WDTp

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. This integrated resistor is the 'R' in the RC filter of Figure 15.

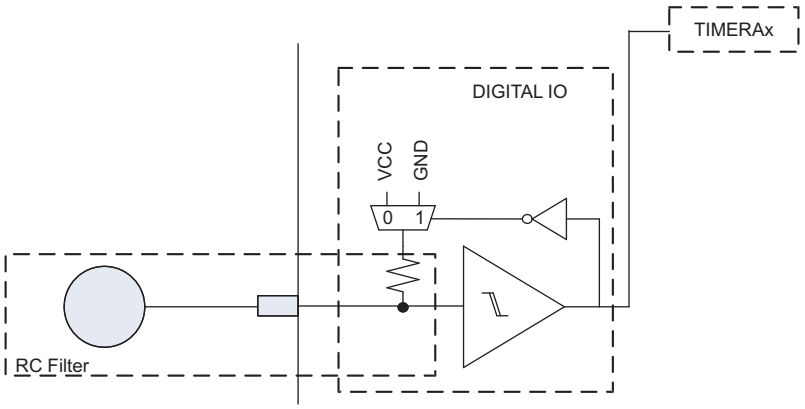


Figure 15. RO_PinOsc Schematic

3.2.2.3.1 Port Parameters

Because the relaxation oscillator is accomplished with internal circuitry the port parameters only include the port selection register (PxSEL), the port selection 2 register (PxSEL2), and the input bit definition. All of these parameters are defined at the element level and there are no pertinent port definitions at the sensor level.

3.2.2.3.2 Timing Parameters

The common timing parameters, measGateSource and accumulationCycles, are the same as the Comp_A+ implementation (see Section 3.2.2.1.2).

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.

```
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_WDT_SMCLK,     //  0->SMCLK, 1-> ACLK
                  //.accumulationCycles= WDTp_GATE_32768          //32768
                  .accumulationCycles= WDTp_GATE_8192             // 8192
                  //.accumulationCycles= WDTp_GATE_512            //512
                  //.accumulationCycles= WDTp_GATE_64             //64
               };
```

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 Timer_A0 sourced from the relaxation oscillator. Since 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 will consume more power due to the CPU staying in Active Mode.

3.2.2.5 Port Parameters

Because the relaxation oscillator is accomplished with internal circuitry the port parameters only include the port selection register (PxSEL), the port selection 2 register (PxSEL2), and the input bit definition. All of these parameters are defined at the element level and there are no pertinent port definitions at the sensor level.

3.2.2.6 Timing Parameters

As shown in Figure 16, the only timing parameter definition for the RO_PINOSC_TA0 implementation is the number of ACLK cycles: sensor->accumulationCycles.

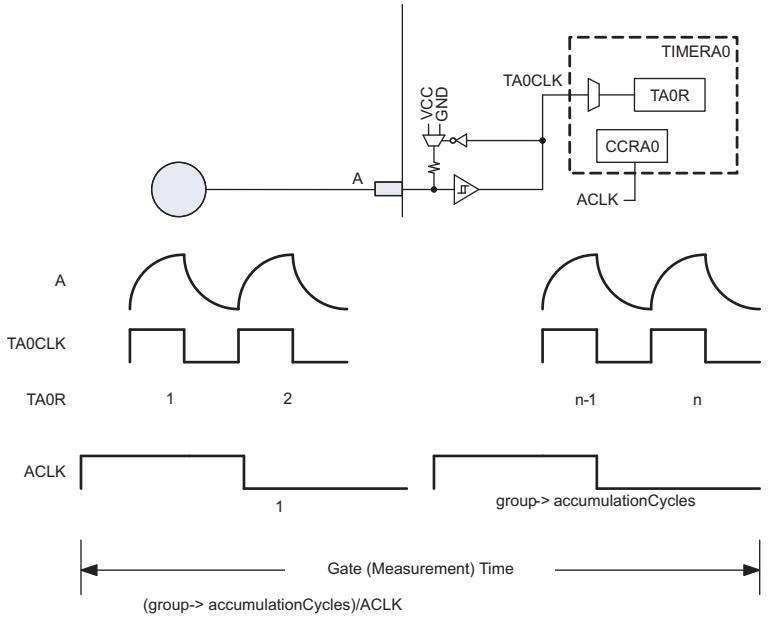


Figure 16. RO_PINOSC_TA0 Timing Parameter

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.

```
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
               };
```

[1] See the device-specific data sheet to determine if the Timer capture input supports this connection to ACLK.

3.2.3 Definitions for the RC Method

3.2.3.1 RC_PAIR_TA0

The RC method measures the RC-time constant, where R represents the resistor and C the capacitance of the electrode. This method uses a single timing resource (peripheral or software) to measure the time it takes the capacitance to charge and discharge 'n' times. In order to measure both charge and discharge, two IO are needed. This is reflected in the function name within the library: RC_PAIR.

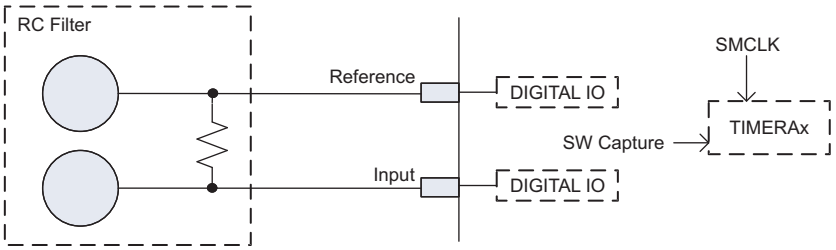


Figure 17. RC Schematic

3.2.3.1.1 Port Parameters

The port parameters which define the input and the reference are described at the element level and not the sensor level (see Section 3.1.1.4).

3.2.3.1.2 Timing Parameters

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/discharge cycles as shown in Figure 18. The number of cycles is defined in the parameter: sensor->accumulation cycles.

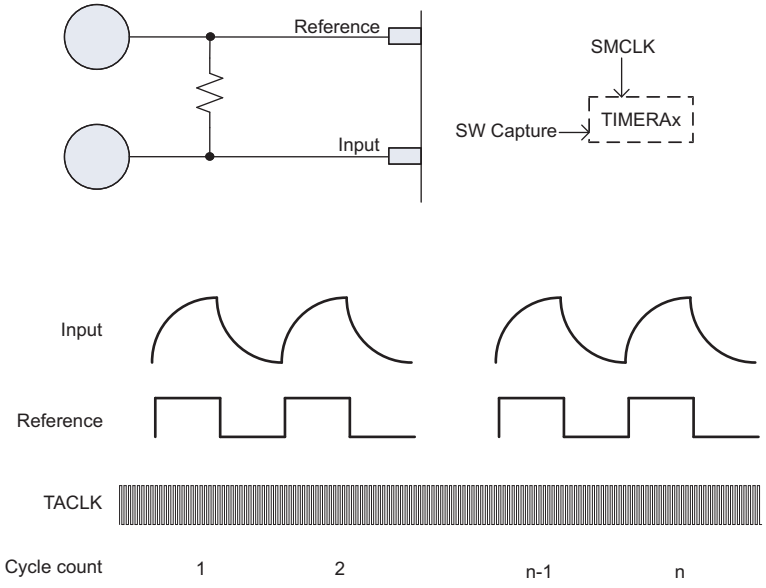


Figure 18. RC Timing Parameters

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/discharge cycles.

```
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
               };
```

3.2.4 Definitions for the Fast Scan Relaxation Oscillator (fRO) 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. In contrast to the RO method the gate time (gate time) is now variable, based upon the relaxation oscillator, instead of fixed. The variation in the gate time is a function of the capacitive element being measured. Additionally, with the RO method the frequency counter was variable, but is now fixed in the fRO implementation. The frequency counter can either be a software loop based upon MCLK or a timer based upon a system clock (typically SMCLK). Now as capacitance increases, the gate time increases and consequently the frequency counter number within that gate time increases.

3.2.4.1 fRO_COMPAp_TAx_SW

The fRO_COMPAp_TAx_SW implementations have the same hardware description as the RO_COMPAp_TAx_WDTp implementations (see Figure 10). 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 fRO_COMPAp_TAx_SW implementation, the variable gate timer is created with the relaxation oscillator and the peripheral Timer_Ax. The gate time is a software loop with a frequency of MCLK/10.

In comparison to the RO method, the fRO method provides similar sensitivity (change in counts) in a shorter measurement or gate time. The theoretical exercise found in Table 9 shows that similar sensitivity can be achieved with the fRO method in less time.

Table 9. Comparison of fRO and RO Measurement Times at 16 MHz

	RO_COMPAp_TAx_WDTp			fRO_COMPAp_TAx_SW		
	Gate Time: (1-MHz SMCLK, WDT, 512)	**Counter: (RO)**	**Counts**	**Gate Time: (160 RO cycles)**	**Counter: (MCLK/10)**	**Counts**
Touched	512 µs	600 kHz	307	150 / 600 kHz = 250 µs	16 MHz / 10	400
Untouched	512 µs	700 kHz	358	150 / 700 kHz = 214 µs	16 MHz / 10	342
			51			58

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

3.2.4.1.1 Port Parameters

The port parameters for the fRO_COMPAp_TAx_SW and the RO_COMPAp_TAx_WDTp implementations are the same (see Section 3.2.2.1 and Section A.1).

3.2.4.1.2 Timing Parameters

There is only one timing parameter; accumulationCycles. Figure 19 shows how the parameter accumulationCycles is used to accumulate multiple relaxation oscillator cycles in the fRO_COMPAp_TA0_SW method.

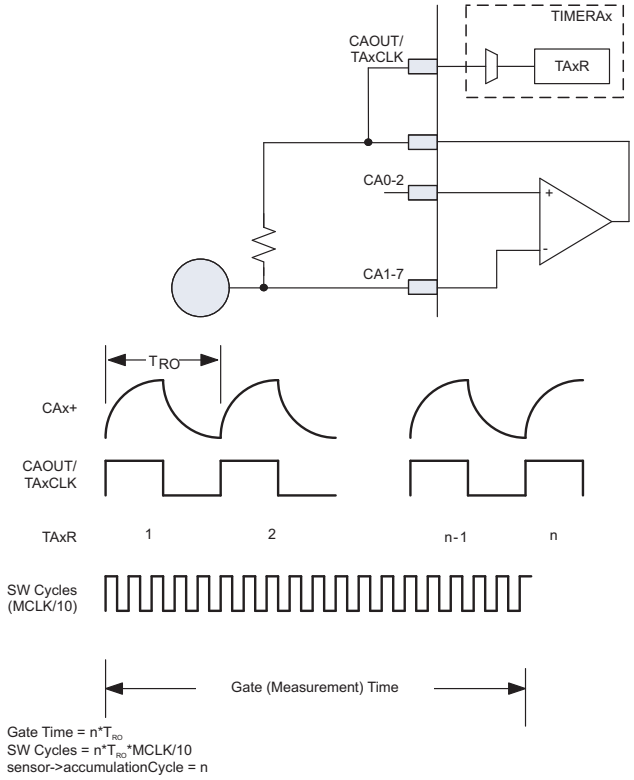


Figure 19. fRO_COMPAp_TA0_SW Timing Parameter

3.2.4.2 fRO_PINOSC_TA0_SW

The fRO_PINOSC_TA0_SW implementation has the same hardware description as the RO_PINOSC_TA0_WDTp implementations (see Section 3.2.2.3). 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 fRO_PINOSC_TA0_SW implementation, the variable gate timer is created with the relaxation oscillator and the peripheral Timer_Ax. The gate time is a software loop with a frequency of MCLK/10.

In comparison to the RO method the fRO method provides similar sensitivity (change in counts) in a shorter measurement or gate time. The theoretical exercise found in Table 7 shows that similar sensitivity can be achieved with the fRO method in less time.

Table 10. Comparison of fRO and RO PinOsc Measurement Times at 16 MHz

	RO_PINOSC_TA0_WDTp			fRO_PINOSC_TAx_SW		
	Gate Time: (1-MHz SMCLK, WDT, 512)	**Counter: (RO)**	**Counts**	**Gate Time: (160 RO cycles)**	**Counter: (MCLK/10)**	**Counts**
Touched	512 µs	1 MHz	512	320 / 1 MHz = 320 µs	16 MHz / 10	512
Untouched	512 µs	1.1 MHz	563	320 / 1.1 MHz = 290 µs	16 MHz / 10	465
			51			47

3.2.4.2.1 Port Parameters

Like the RO_PinOsc implementation there are no port parameters found in the sensor level description. The PxSEL and PxSEL2 definitions are found in the element level description.

3.2.4.2.2 Timing Parameters

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 20, is defined by the variable accumulationCycles at the sensor level: sensor->accumulationCycles.

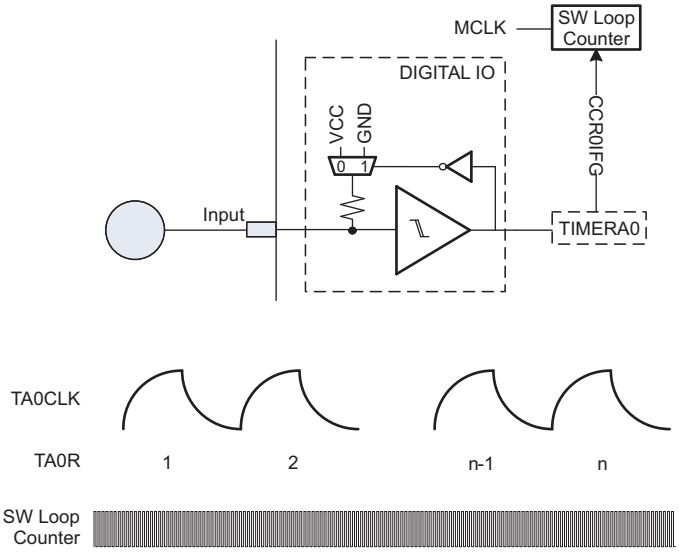


Figure 20. fRO_PINOSC_TA0 Timing Parameters

3.2.4.3 fRO_COMPB_TAx_SW

The fRO_COMPB_TAx_SW implementation has the same hardware description as the 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 fRO_COMPB_TAx_SW implementation, the variable gate timer is created with the relaxation oscillator and the peripheral Timer_Ax. The gate time is a software loop with a frequency of MCLK/10.

In comparison to the RO method the fRO method provides similar sensitivity (change in counts) in a shorter measurement or gate time. The theoretical exercise found in Table 11 shows that similar sensitivity can be achieved with the fRO method in less time.

Table 11. Comparison of fRO and RO Measurement Times at 25 MHz

	RO_COMPB_TAx_WDTA			fRO_COMPB_TAx_SW		
	Gate Time: (1-MHz SMCLK, WDTA, 512)	**Counter: (RO)**	**Counts**	**Gate Time: (160 RO cycles)**	**Counter: (MCLK/10)**	**Counts**
Touched	512 µs	600 kHz	307	80 / 600 kHz = 133 µs	25 MHz / 10	333
Untouched	512 µs	700 kHz	358	80 / 700 kHz = 114 µs	25 MHz / 10	285
			51			48

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

3.2.4.3.1 Port Parameters

The port parameters for the fRO_COMPAp_TAx_SW and the RO_COMPAp_TAx_WDTp implementations are the same (see Section 3.2.2.2).

3.2.4.3.2 Timing Parameters

There is only one timing parameter; accumulationCycles. Figure 21 shows how the parameter accumulationCycles is used to accumulate multiple relaxation oscillator cycles in the fRO_COMPB_TAx_SW method.

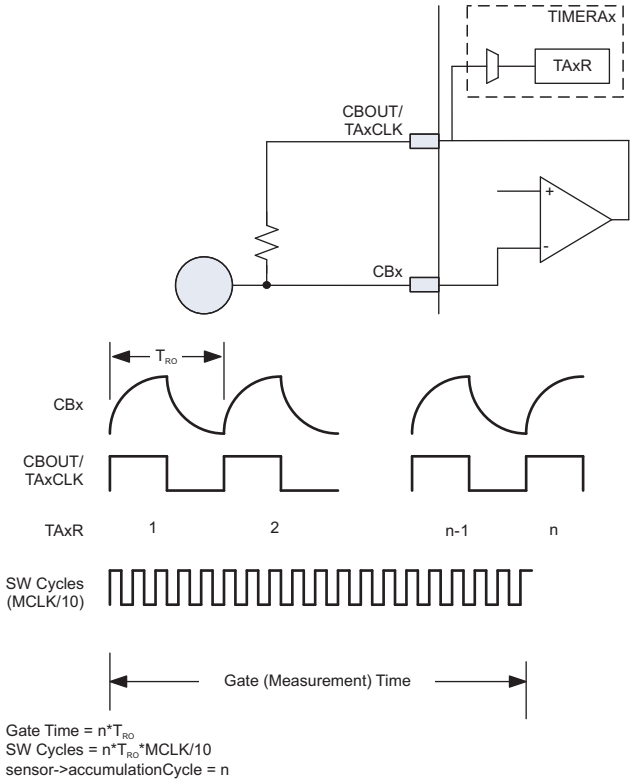


Figure 21. fR0_COMPB_TAx_SW Timing Parameter

3.2.4.4 fRO_COMPAp_SW_TAx

As the name indicates the fRO_COMPAp_SW_TAx implementation uses a software timer to create the gate time and the timer peripheral as the frequency counter, where the frequency source is a system clock. Since this implementation does not use the timer to count the number relaxations oscillations the physical connection is no longer needed as shown in Figure 22.

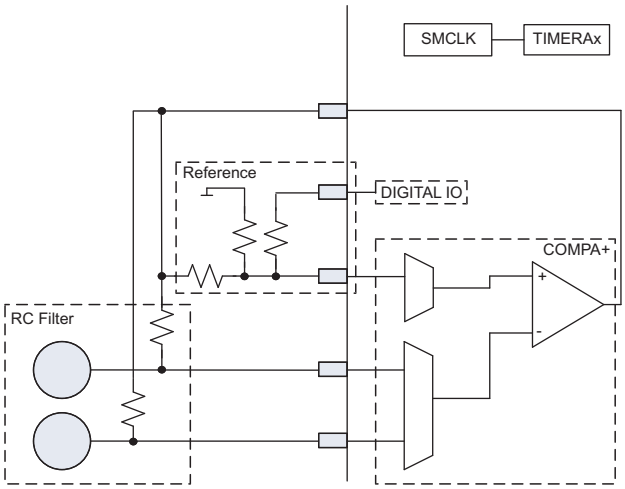


Figure 22. fRO_COMPAp_SW_TAx General Description

In comparison to the RO method the fRO method provides similar sensitivity (change in counts) in a shorter measurement or gate time. The theoretical exercise found in Table 12 shows that similar sensitivity can be achieved with the fRO method with a shorter gate time.

Table 12. Comparison of fRO and RO Measurement Times at 16 MHz

	RO_COMPAp_TAx_WDTA			fRO_COMPAp_SW_TAx		
	Gate Time: (1-MHz SMCLK, WDTp, 512)	Counter: (RO)	Counts	Gate Time: (160 RO cycles, MCLK is 16 MHz) (1)	Counter: (SMCLK)	Counts
Touched	512 µs	600 kHz	307	80 / 600 kHz = 133 µs	16 MHz	333
Untouched	512 µs	700 kHz	358	80 / 700 kHz = 114 µs	16 MHz	285
			51			48

(1) Having a slow MCLK results in errors in counting the number of relaxation oscillations. That is, if the software polling loop is too slow it may miss oscillation cycles resulting in a longer gate time than expected. The software loop is 14 cycles and, therefore, the MCLK frequency must be 14 times faster than the maximum relaxation oscillator frequency.

The fast RO method would typically be used in devices which have multiple timers available, so that the frequency counter function and gate timer would be accomplished with hardware. This would decrease power consumption (running in LPM0 instead of in active mode).

3.2.4.4.1 Port Parameters

The port parameters are similar to those described in Section 3.2.2.1, with the exception of txclkDirRegister, txclkSelRegister, and txclkBits, which are not used.

3.2.4.4.2 Timing Parameters

There are three timing parameters; measureGateSource, sourceScale, and accumulationCycles. The gate timer which defines the gate time 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 Timer_A control register TACTL.

Table 13. measureGateSource Definitions for fRO_xxxx_SW_Txx

Name	Definition	Clock Source
TIMER_TxCLK	0x0000	TxCLK
TIMER_ACLK	0x0100	ACLK
TIMER_SMCLK	0x0200	SMCLK
TIMER_INCLK	0x0300	INCLK

sourceScale is used to divide the timer source. This is equivalent to the input divider bits (IDx) found in the Timer_A control register TACTL.

Table 14. sourceScale Definitions for fRO_xxxx_SW_Txx

Name	Definition	Clock Source
TIMER_SOURCE_DIV_0	0x0000	TxCLK
TIMER_SOURCE_DIV_1	0x0040	ACLK
TIMER_SOURCE_DIV_2	0x0080	SMCLK
TIMER_SOURCE_DIV_3	0x00C0	INCLK

accumulationCycles defines the number of relaxation oscillator cycles per gate period. In the fRO_COMPAp_SW_TAx 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.

3.2.5 Slider and Wheel Specific Definitions

The following definitions are required only with the 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 structure.h:

```
/ 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 structure.c, the sensor definitions for *points* and *sensorThreshold* must be added.

The variable *points* defines 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 that requires a comparison of only the end element (the dominant element) and the one neighbor. If the response exceeds 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 ILLEGAL_SLIDER_WHEEL_POSITION (defined in structure.h) 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 watchdog timer peripheral, 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 (Measurement 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 capacitance measurement time. This time can either be a number of cycles from a fixed (system clock) or variable (relaxation oscillator) clock source.

Table 15 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 will result in a decrease in sensitivity. In the fRO implementation the sensitivity can be increased (while keeping the shorter gate time) by increasing the fixed frequency clock feeding the counter (see Section 3.2.4).

(1) An explicit example of this is with Timer_A3, where the library does not use all three capture and control registers; however, the CCIFG may be set when the timer is used.

Table 15. Measurement Time Examples

Method	Gate Time Source (.measGateSource)	Interval Definition (.accumulationCycles)	Time (ms)
RO_XXX_YYY_WDTp/A	ACLK = VLO ~ 12 kHz	64 (WDTp_GATE_64)	5.33
	SMCLK = 2 MHz	8192 (WDTp_GATE_8192)	4.1
	SMCLK = 1 MHz	512 (WDTp_GATE_512)	0.512
RO_PINOSC_TA0	ACLK = VLO ~ 12 kHz	100	8.3
RC_PAIR_XXX	Rise + Fall (untouched) ~ 1.4 µs	20	0.028 + TBD(1)
	Rise + Fall (touched) ~ 1.6 µs	20	0.032 + TBD
fRO_COMPx_YYY_SW	RO (untouched) ~ 700 kHz	4000	5.71
	RO (touched) ~ 600 kHz	4000	6.67
fRO_PINOSC_YYY_SW	RO (untouched) ~ 1.2 MHz	4000	3.33
	RO (touched) ~ 1 MHz	4000	4
fRO_COMPx_SW_YYY	RO (untouched) ~ 700 kHz	500	0.714
	RO (touched) ~ 600 kHz	500	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 16 shows an example of how the code size increases with higher levels of abstraction.

Table 16. Example Flash Resource Allocation

API Calls	Library (bytes)	Configuration Structure: Six Elements, Three Structures (RO_PINOSC_TA0_WDTp)	Comments
TI_CAPT_Raw	396 (0x18C)	114 (0x72)	Optimization level 0 (CCSv4, CGT v3.3.2)
TI_CAPT_Init_Baseline TI_CAPT_Update_Baseline TI_CAPT_Custom	1840 (0x730)	114 (0x72)	Optimization level 0 (CCSv4, CGT v3.3.2)
TI_CAPT_Init_Baseline TI_CAPT_Update_Baseline TI_CAPT_Custom TI_CAPT_Button TI_CAPT_Wheel	2828 (0xB0C)	120 (0x78)	Optimization level 0 (CCSv4, CGT v3.3.2)

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 TI_CAPT_RAW API, then TOTAL_NUMBER_OF_ELEMENTS should not be defined and, therefore, no RAM resources are consumed.

RAM can be allocated statically or dynamically to perform the measurements to determine a change in capacitance (TI_CAPT_Custom and sensor abstractions). If the RAM is allocated statically, the definition RAM_FOR_FLASH must be made in structure.h. 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 use of the library in the context of the application. For example, if the capacitance measurement time is set with 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 Timers: A, B, D

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 Watchdog Timer

The watchdog timer ISR cannot be used if the peripheral is already selected for use in the library.

4.4.3 Comparators: A, B

The comparator peripheral is reconfigured and initialized with every measurement and, therefore, can be used for other functions 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.

4.4.4 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. If the changes are against the direction of interest this information is used by the baseline tracking but not provided to the application layer. Several API function calls are provided to adjust the baseline tracking during run time. The TI_CAPT_Custom API is intended for use with ‘custom' element and sensor design. The baseline tracking can still be used but the interpretation and application of the change in capacitance must be handled in the application layer.

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. The associated APIs include the interpretation and application of the TI_CAPT_Custom function.

Table 17. API Functions

Category	API Function
Capacitance Measurement	uint8_t TI_CAPT_Button(const struct Sensor *);
Capacitance Measurement	const struct Element * TI_CAPT_Buttons(const struct Sensor *);
Capacitance Measurement	uint16_t TI_CAPT_Slider(const struct Sensor*);
Capacitance Measurement	uint16_t TI_CAPT_Wheel(const struct Sensor*);
Capacitance Measurement	void TI_CAPT_Custom(const struct Sensor *, uint16_t*);
Capacitance Measurement	void TI_CAPT_Raw(const struct Sensor*, uint16_t*);
Baseline Tracking	void TI_CAPT_Init_Baseline(const struct Sensor*)
Baseline Tracking	void TI_CAPT_Update_Baseline(const struct Sensor*,uint8_t)
Baseline Tracking	void TI_CAPT_Reset_Baseline_Tracking(void);
Baseline Tracking	void TI_CAPT_Update_Tracking_DOI(uint8);
Baseline Tracking	void TI_CAPT_Update_Tracking_Rate(uint8_t, uint8_t);

5.1 *uint8_t TI_CAPT_Button(Sensor *);*

Inputs: Pointer to Sensor, which defines the element that represents a 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 element and 1 means that the change in capacitance has exceeded the threshold.

5.2 *element * TI_CAPT_ Buttons(Sensor *);*

Inputs: Pointer to Sensor, which defines the group of elements in which each element represents a button

Outputs: pointer to an element or 0

Function: Measure the sensor (buttons) and determine which button, if any, is being touched. This function return is the structure pointer to the element that exceeds its threshold by the largest margin (% of the threshold value). If no button exceeds its threshold (set in the element structure), then this function returns a 0 or 'Null Pointer'.

5.3 uint16_t TI_CAPT_Slider(Sensor *);

Inputs: Pointer to Sensor, which defines the group of elements that form the slider

Outputs: location on the slider; ILLEGAL_SLIDER_WHEEL_POSITION-> No touch; 0-max -> touch at location where max is defined by Sensor structure.

Function: This function returns the position of the slider if touched and an illegal value if no touch was detected.

The order of the elements within the Sensor structure 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).

```
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
                };
```

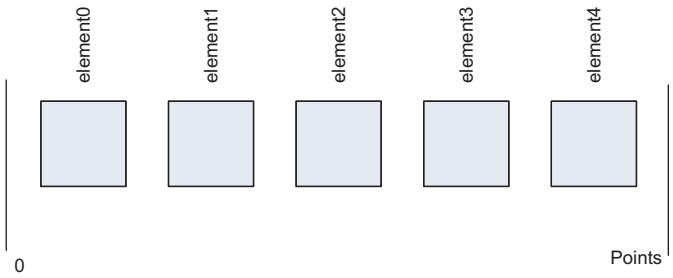


Figure 23. Slider Implementation

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 structure.h.

5.4 uint16_t TI_CAPT_Wheel(Sensor *);

```
uint16_t TI_CAPT_Wheel(Sensor *);
```

Inputs: Pointer to Sensor, which defines the group of elements that form the 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 value wraps around.

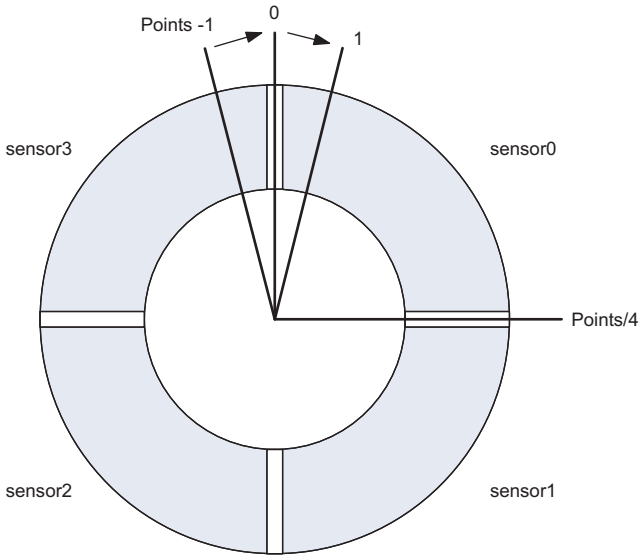


Figure 24. Wheel Example

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 structure.h.

5.5 *void TI_CAPT_Custom(Sensor *, uint16_t *);*

Inputs: Pointer to Sensor, which defines the group of elements that form 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.

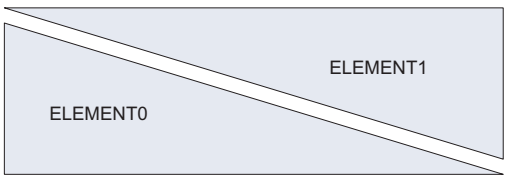


Figure 25. Custom Slider Example

This function requires that the application allocate an array that 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 lower level functions or measurement and baseline tracking can still be managed by the library.

5.6 void TI_CAPT_Raw(Sensor*, uint16_t*);

Inputs: Pointer to Sensor, which defines the group of elements that form 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 that 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.

5.7 void TI_CAPT_Init_Baseline(Sensor *);

Inputs: pointer to Sensor

Outputs: None.

Function: Measure the sensor and directly place measured values into the associated baseline variables. At the beginning of operation the values for the baseline, stored in RAM, may be in an unknown state. 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.

5.8 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.

5.9 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); that is, track decreases in capacitance at the fast setting and increases in capacitance at the slow setting.

5.10 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.

5.11 void TI_CAPT_Update_Tracking_Rate (uint8_t);

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.

Table 18. Update Tracking Rate Format

Input Value	Tracking Rate in Direction of Interest	Tracking Rate Against Direction of Interest
0000 0000b	Very Slow	Fast
0001 0000b	Slow (Default)	Fast (Default)
0010 0000b	Medium	Fast
0011 0000b	Fast	Fast
0000 0000b	Very Slow	Fast
0100 0000b	Very Slow	Medium
1000 0000b	Very Slow	Slow
1100 0000b	Very Slow	Very Slow

Outputs: None.

Function: Update the tracking rates per Table 18.

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 is represented by a decrease in counts with the RO method.

```
#include "CTS_Layer.h"

// Need to Allocate at least 10 bytes to HEAP in IDE
// 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
  }
}
```

Table 19. Example of Raw Results With RO Method

Active Element	raw_data[0]	raw_data[1]	raw_data[2]	raw_data[3]
None	394	435	426	367
0 (light)	257	424	427	369
0 (normal)	137	410	428	371
0 (heavy)	110	304	420	371
None	390	435	426	367
1 (light)(1)	367	223	408	367
1 (normal)	361	165	401	366
1 (heavy)	226	117	332	365
None	389	435	425	368
2 (normal)	382	349	146	341
None	390	435	426	267
3 (normal)	388	421	255	111

(1) 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 19, 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 would be 137/2 and 212/2, respectively.

6.2 Button and Buttons

Defining the threshold value for the TI_CAPT_Button and TI_CAPT_Buttons abstractions is done with the TI_CAPT_Custom function. 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.

```
#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
  }
}
```

Table 20. Example of Change in Capacitance Results With RO Method

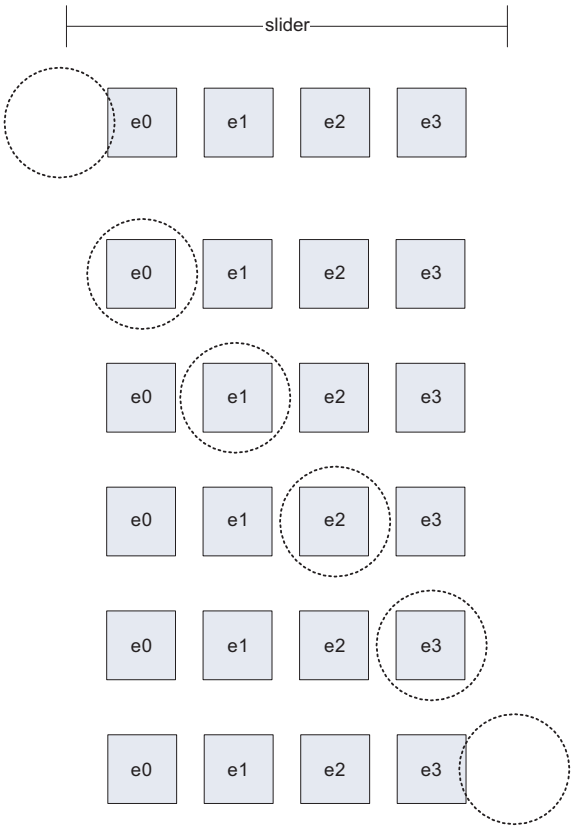
Active Element	delta_data[0]	delta_data[1]	delta_data[2]	delta_data[3]
None	0	0	0	0
0 (light)	130	11	0	0
0 (normal)	188	16	0	0
0 (heavy)	287	71	0	0
None	0	0	0	0
1 (light)	14	205	13	1
1 (normal)	30	288	35	2
1 (heavy)	222	328	91	2
None	0	0	0	0
2 (normal)	3	49	292	24
None	0	0	0	0
3 (normal)	0	5	52	243

The threshold calculation is similar to that shown with Table 19.

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. Using Table 20 as an example, the maxResponse would simply be 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 elements 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 that has the largest area overlap between the element and interaction. Figure 26 shows how to use the custom function to measure the performance of a slider and determine the values for the threshold and maxResponse variables.



TI_CAPTI_Custom(&slider, dCnt)

Element	Address	Value	Position
e0	dCnt[0]	a	0
e1	dCnt[1]	-	
e2	dCnt[2]	-	
e3	dCnt[3]	-	
e0	dCnt[0]	b	points/8
e1	dCnt[1]	c	
e2	dCnt[2]	-	
e3	dCnt[3]	-	
e0	dCnt[0]	d	3*points/8
e1	dCnt[1]	e	
e2	dCnt[2]	d	
e3	dCnt[3]	-	
e0	dCnt[0]	-	5*points/8
e1	dCnt[1]	f	
e2	dCnt[2]	g	
e3	dCnt[3]	f	
e0	dCnt[0]	-	7*points/8
e1	dCnt[1]	-	
e2	dCnt[2]	h	
e3	dCnt[3]	j	
e0	dCnt[0]	-	points
e1	dCnt[1]	-	
e2	dCnt[2]	-	
e3	dCnt[3]	k	

Figure 26. Measurement Example of a Four-Element Sensor

Ideally, the geometry of the electrodes would result 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 21. Measurement Example of a Four-Element Sensor

Element	Threshold	maxResponse (1)
e0	(a + d) / 2	b or (threshold + 655), whichever is smaller
e1	(c + f) / 2	e or (threshold + 655), whichever is smaller
e2	(d + h) / 2	g or (threshold + 655), whichever is smaller
e3	(f + k) / 2	j or (threshold + 655), whichever is smaller

(1) In some geometries 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 27, the sensorThreshold defines the valid area of the slider. A good starting value is 75. Decreasing this value increases the slider area at the expense of position accuracy. Conversely, increasing this value increases position accuracy, but the interaction must follow the center line of the wheel or slider much more closely.

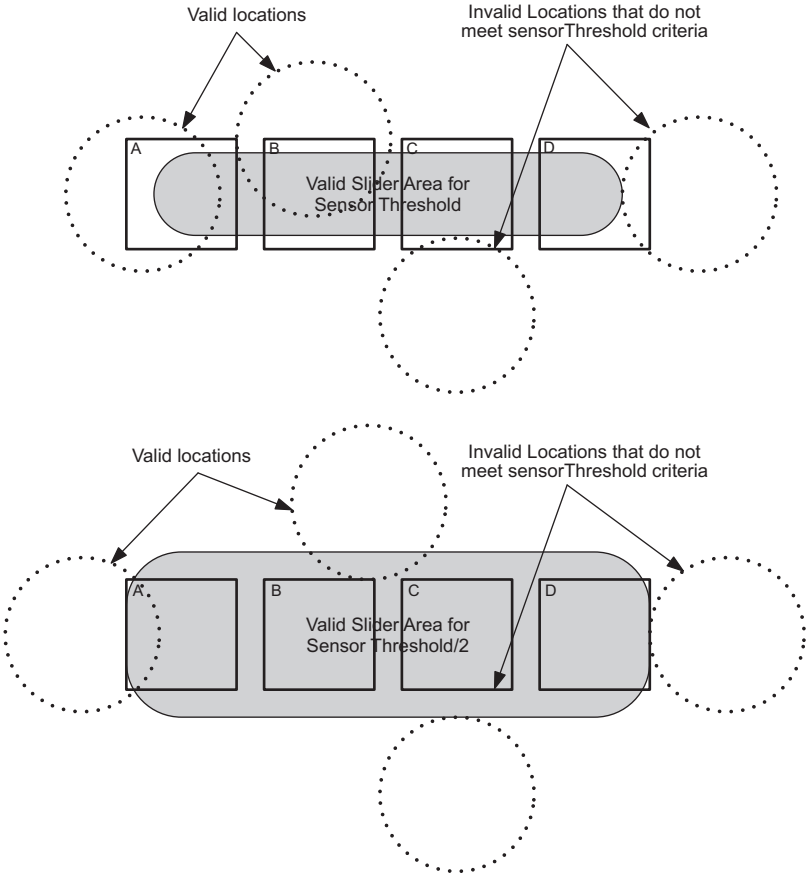


Figure 27. Valid Slider Locations as a Function of the Sensor Threshold

Appendix A Element and Sensor Definitions

A.1 RO_PINOSC_TA0_xx

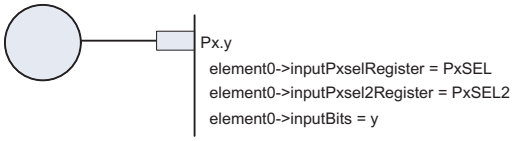


Figure 28. RO_PINOSC_TA0 Element and Sensor Definitions

A.1.1 RO_PINOSC_TA0_WDTp

```
const struct Element element0 = {
              .inputPxselRegister = (uint8_t *)&P2SEL,
              .inputPxsel2Register = (uint8_t *)&P2SEL2,
              .inputBits = BIT3,
              .maxResponse = 400,
              .threshold = 50
};

const struct Sensor group =
               {
                  .halDefinition = RO_PINOSC_TA0_WDTp,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,  // point to first element
                  // Timer Information
                  .measGateSource= GATE_WDTp_SMCLK,     //  0->SMCLK, 1-> ACLK
                  .accumulationCycles= WDTp_GATE_8192  // 8192 SMCLK cycles in gate time
               };
```

A.1.2 RO_PINOSC_TA0

```
const struct Element element0 = {
              .inputPxselRegister = (uint8_t *)&P2SEL,
              .inputPxsel2Register = (uint8_t *)&P2SEL2,
              .inputBits = BIT3,
              .maxResponse = 200,
              .threshold = 80
};

const struct Sensor group =
               {
                  .halDefinition = RO_PINOSC_TA0,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,  // point to first element
                  // Timer Information
                  .accumulationCycles = 20   //  Number of ACLK cycles in gate time
               };
```

A.1.3 fRO_PINOSC_TA0_xx

```
const struct Element element0 = {
              .inputPxselRegister = (uint8_t *)&P2SEL,
              .inputPxsel2Register = (uint8_t *)&P2SEL2,
              .inputBits = BIT3,
              .maxResponse = 200,
              .threshold = 80
};

const struct Sensor group =
               {
                  .halDefinition = fRO_PINOSC_TA0_SW,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,  // point to first element
                  // Timer Information
                  .accumulationCycles = 1000   //  Number of RO cycles in gate time
               };
```

A.2 xx_COMPAp_TAx_xx

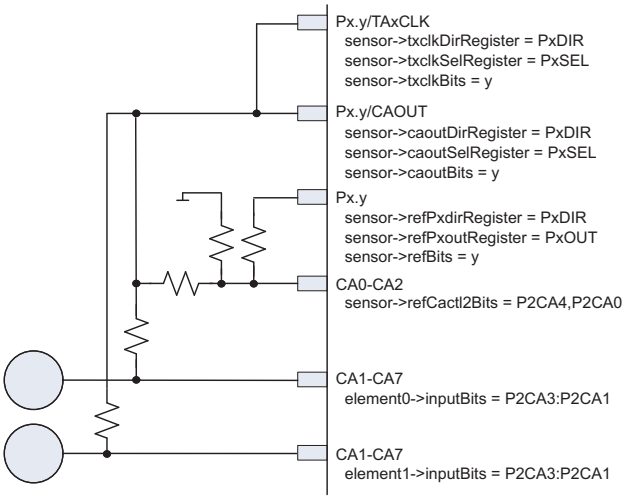


Figure 29. RO_COMPAp_TAx Element and Sensor Definitions

A.2.1 RO_COMPAp_TAx_xx

```
const struct Element element0 = {
              .inputBits =  P2CA2,  // CA2
              .maxResponse = 250,
              .threshold = 100
};

const struct Sensor group =
               {
                  .halDefinition = RO_COMPAp_TA0_WDTp,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,
                  // 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,  // SxSEL
                  .caoutBits = BIT7, // BITy
                  .txclkBits = BIT0,
                  .refPxoutRegister = (uint8_t *)&P1OUT,
                  .refPxdirRegister = (uint8_t *)&P1DIR,
                  .refBits = BIT6,           // BIT6
                  .capdBits = BIT1+BIT2,     // BIT1,2
                  .refCactl2Bits = P2CA4,    // CACTL2-> P2CA4 , CA1
                  // Timer Information
                  .measGateSource= GATE_WDTp_SMCLK,    //  0->SMCLK, 1-> ACLK
                  .accumulationCycles= WDTp_GATE_512   // gate time is 512 SMCLK cycles
               };
```

A.2.2 fRO_COMPAp_TAx_xx

```
const struct Element element0 = {
              .inputBits =  P2CA2,  // CA2
              .maxResponse = 230,
              .threshold = 80
};

const struct Sensor group =
               {
                  .halDefinition = fRO_COMPAp_TA0_SW,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,
                  // 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,  // SxSEL
                  .caoutBits = BIT7, // BITy
                  .txclkBits = BIT0,
                  .refPxoutRegister = (uint8_t *)&P1OUT,
                  .refPxdirRegister = (uint8_t *)&P1DIR,
                  .refBits = BIT6,           // BIT6
                  .capdBits = BIT1+BIT2,      // BIT1,2
                  .refCactl2Bits = P2CA4,   // CACTL2-> P2CA4 , CA1
                  // Timer Information
                  .accumulationCycles = 1000
               };
```

A.3 fRO_COMPAp_SW_xx

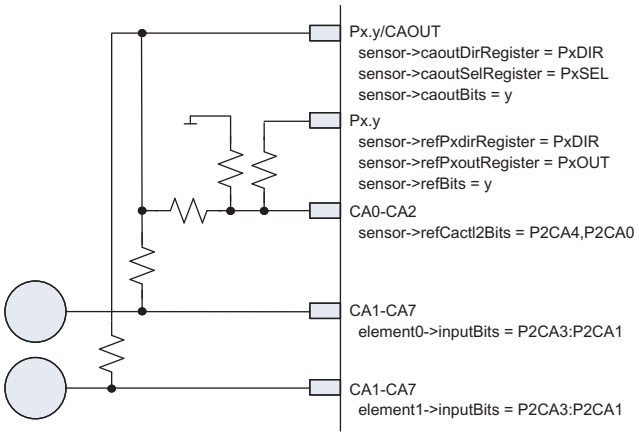


Figure 30. fRO_COMPAp_TAx Element and Sensor Definitions

```
const struct Element element0 = {
              .inputBits =  P2CA2,  // CA2
              .maxResponse = 180,
              .threshold = 60
};

const struct Sensor group =
               {
                  .halDefinition = fRO_COMPAp_SW_TA0,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,
                  // Reference Information
                  // CAOUT is P1.7
                  .caoutDirRegister = (uint8_t *)&P1DIR,  // PxDIR
                  .caoutSelRegister = (uint8_t *)&P1SEL,  // PxSEL
                  .caoutBits = BIT7, // BITy
                  .refPxoutRegister = (uint8_t *)&P1OUT,
                  .refPxdirRegister = (uint8_t *)&P1DIR,
                  .refBits = BIT6,           // BIT6
                  .capdBits = BIT1+BIT2,      // BIT1,2
                  .refCactl2Bits = P2CA4,   // CACTL2-> P2CA4 , CA1
                  // Timer Information
                  .accumulationCycles = 1000  // number of RO cycles in gate time
               };
```

A.4 xx_COMPB_TAx_xx

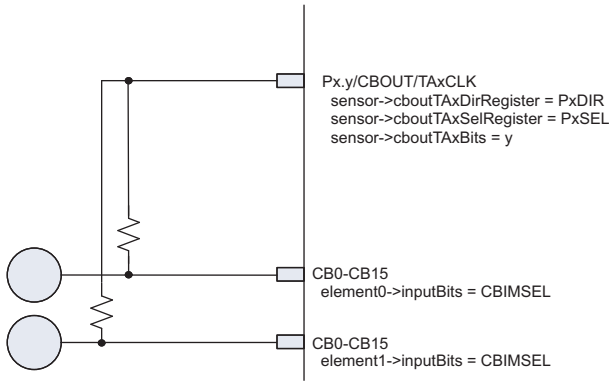


Figure 31. RO_COMPB_TAx Element and Sensor Definitions

A.4.1 RO_COMPB_TAx_xx

```
const struct Element element0 = {
             .inputBits = CBIMSEL_2,  // CB2
             .maxResponse = 310,
             .threshold = 240
};

const struct Sensor group =
              {
                 .halDefinition = RO_COMPB_TA0_WDTA,
                 .numElements = 1,
                 .baseOffset = 0,
                 .cbpdBits = (BIT2),
                 // Pointer to elements
                 .arrayPtr[0] = &element0,  // point to first element
                 .cboutTAxDirRegister = (uint8_t *)&P3DIR,  // PxDIR
                 .cboutTAxSelRegister = (uint8_t *)&P3SEL,  // PxSEL
                 .cboutTAxBits = BIT4, // P3.4
                 // Timer Information
                 .measGateSource= GATE_WDTA_SMCLK,    //  0->SMCLK, 1-> ACLK
                 .accumulationCycles= WDTA_GATE_512K  //
              };
```

A.4.2 fRO_COMPB_TAx_xx

```
const struct Element element0 = {
             .inputBits = CBIMSEL_2,  // CB2
             .maxResponse = 250,
             .threshold = 160
};

const struct Sensor group =
              {
                 .halDefinition = fRO_COMPB_TA0_SW,
                 .numElements = 1,
                 .baseOffset = 0,
                 .cbpdBits = (BIT2),
                 // Pointer to elements
                 .arrayPtr[0] = &element0,  // point to first element
                 .cboutTAxDirRegister = (uint8_t *)&P3DIR,  // PxDIR
                 .cboutTAxSelRegister = (uint8_t *)&P3SEL,  // PxSEL
                 .cboutTAxBits = BIT4, // P3.4
                 // Timer Information
                 .accumulationCycles= 1000  /
              };
```

A.5 RC_PAIR_TAx

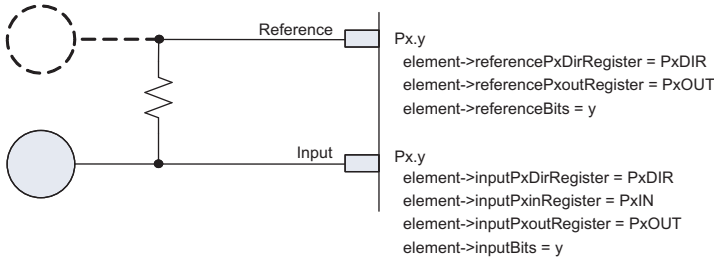


Figure 32. RC_PAIR_TAx Element and Sensor Definitions

```
const struct Element element0 = {
              .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
};

const struct Sensor group =
               {
                  .halDefinition = RC_PAIR_TA0,
                  .numElements = 1,
                  .baseOffset = 0,
                  // Pointer to elements
                  .arrayPtr[0] = &element0,           // point to first element
                  // Timer Information
                  .accumulationCycles= 4             // 4 charge/discharge cycles
               };
```

Appendix B Capacitive Touch Sensor Layer Detailed Description

B.1 Capacitive Touch Sensor Layer

The capacitive touch sensor layer performs several functions and it can be divided up into several layers. At the top (closest to the actual application) is the presentation layer. Depending upon the API that is called, the element information is translated or 'presented' to the application layer in the expected format. Closely related to the presentation layer is the dominant key detection. This function is not associated with an API call, but it is used by the presentation layer to determine the dominant element. The final two layers are the custom or delta layer and the raw capacitance layer. The API can call directly into the custom layer or the raw layer and the refinement and interpretation is left up to the application layer. In the case of the custom API call the function manages the base capacitance tracking as well as provides the change in capacitance or delta between the current measurement and the tracked base capacitance. The RAW API call simply supplies the raw instantaneous capacitance measurement without any adjustments or consideration of the base capacitance.

B.1.1 Status/Baseline Control Register

```
uint16 ctsStatusReg = 0;
```

15	14	13	12	11	10	9	8
Unused							

7	6	5	4	3	2	1	0
TRADOI		**TRIDOI**		**Unused**	**PAST_EVNT**	**DOI**	**EVNT**

Unused	Bits 15-8	Unused	
TRADOI	Bits 7-6	Tracking rate against direction of interest	
		00	Very slow
		01	Slow
		10	Medium
		11	Fast
TRIDOI	Bits 5-4	Tracking rate in direction of interest	
		00	Very slow
		01	Slow
		10	Medium
		11	Fast
Unused	Bit 3	Unused	
PAST_EVNT	Bit 2	Past event. An even occurred on the previous scan	
		0	No event occurred on the previous scan
		1	An event occurred on the previous scan
DOI	Bit 1	Direction of interest	
		0	Increasing capacitance
		1	Decreasing capacitance
EVNT	Bit 0	Event. One of the elements in the group has detected a threshold crossing	
		0	No event occurred
		1	An event occurred

Figure 33. Description of Status/Baseline Control Register (RAM)

B.1.2 Baseline Tracking

The delta calculation is the difference between the current measurement and the previous capacitance measurements or base line capacitance (baseCnt in the context of the code). The base line capacitance is followed or tracked to account for any environmental changes that would impact the mechanism used to make the capacitance measurement. This includes but is not limited to Vcc, temperature, and humidity.

B.1.2.1 Direction of Interest

As previously mentioned 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 would require 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. The delta capacitance function sets the result to 0 to indicate a change in capacitance that is against the direction of interest.

B.1.2.2 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.

B.1.2.3 Updating the Baseline Capacitance

The delta calculation results in a zero value (representing a change against or opposite the direction of interest), a non-zero value less than the threshold, or a value greater than the threshold. As shown in Figure 34, a zero value results in a baseline update per the TRADIO setting.

A delta value 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. Therefore when an event occurs within a sensor it is important to suspend baseline updates in the direction of interest.

Baseline updates in the direction of interest only occur if the delta calculation result is non-zero and less than the threshold. The past event flag, PAST_EVNT, indicates that one of the elements within a sensor has experienced a threshold crossing. If the PAST_EVNT flag is true, then the baseline should not be updated, because the increase may be the result of a touch on a neighboring device.

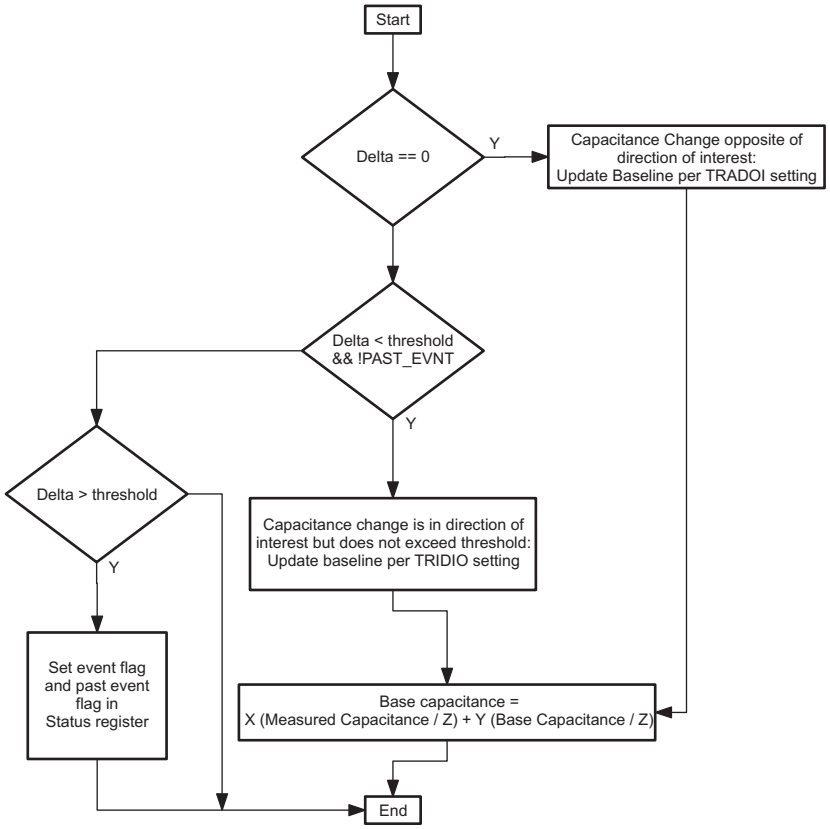


Figure 34. Base Capacitance Update

Table 22. Tracking Settings Against Direction of Interest

Setting	Description	RO (meas < base)	RC, Fast RO (meas > base)
3	Fast	base = meas / 2 + base / 2	base = meas / 2 +base / 2
2	Medium	base = meas / 4 + 3 × base / 4	base = meas / 4 + 3 × base / 4
1	Slow	base = meas / 64 + 63 × base / 64	base = meas / 64 + 63 × base / 64
0	Very Slow (Default)	base = meas / 128 + 127 × (base / 128)	base = meas / 128 + 127 × (base / 128)

Table 23. Tracking Settings in Direction of Interest

Setting	Description	RO (meas < base)	RC, Fast RO (meas > base)
3	Fast	base = 3 × meas / 4 + base / 4	base = 3 × meas / 4 + base / 4
2	Medium	base = meas / 2 + base / 2	base = meas / 2 + base / 2
1	Slow (Default)	Decrement base	Increment base
0	Very Slow	Decrement base only if difference between meas and base is greater than 15	Increment base only if difference between meas and base is greater than 15

B.1.3 Measurement Functions

B.1.3.1 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.

B.1.3.2 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 whose 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 base line capacitance of a given element. A deltaCnt of 0 indicates that the change in capacitance is opposite (against) the direction of interest.

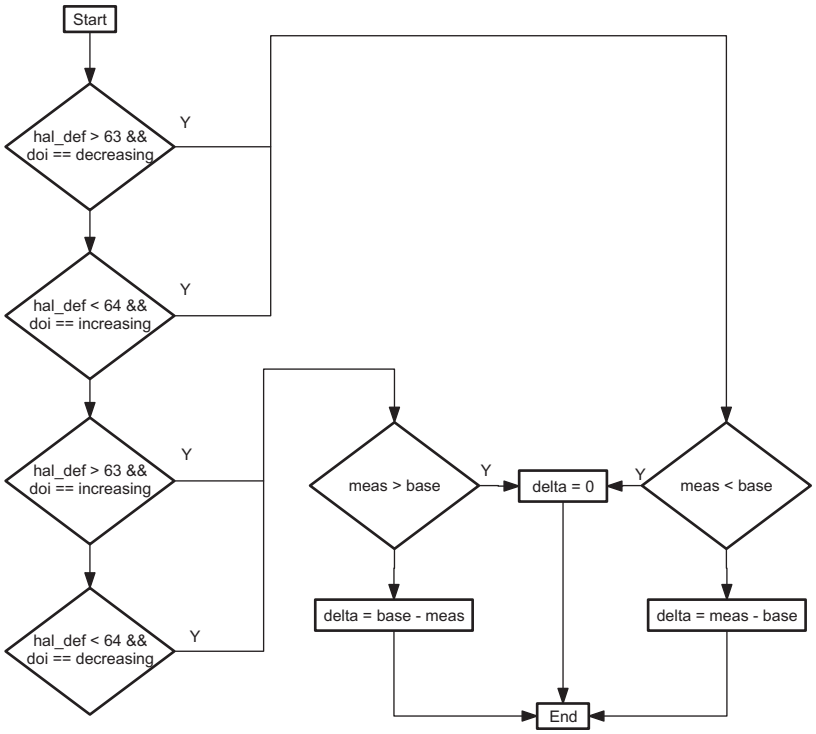


Figure 35. Delta Measurement

B.1.3.3 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 are 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 24. HAL Definitions

Name	Type	Number	Measure Hardware	Measure Time	Gate Time
RC_PAIR_TA0	RC	1	Digital I/O	Timer_A0	NA
RC_SINGLE_TA0	RC	2	Digital I/O	Timer_A0	NA
RC_PAIR_TA1	RC	3	Digital I/O	Timer_A1	NA
RC_PAIR_TD0	RC	4	Digital I/O	Timer_D0	NA
fRO_PINOSC_TA0_SW	fRO	25	Digital I/O	Timer_A0	Software
fRO_COMPB_TA0_TA1	fRO	26	Comp_B	Timer_A1	Timer_A0
fRO_COMPB_TA0_TD0	fRO	27	Comp_B	Timer_D0	Timer_A0
RO_COMPAp_TA0_WDTp	RO	64	Comp_A+	Timer_A0	WDT+
RO_PINOSC_TA0_WDTp	RO	65	Digital I/O	Timer_A0	WDT+
RO_PINOSC_TA0	RO	66	Digital I/O	Timer_A0	ACLK
RO_COMPAp_TA1_WDTp	RO	67	Comp_A+	Timer_A1	WDT+
RO_COMPB_TA0_WDTA	RO	68	Comp_B	Timer_A0	WDTA
RO_COMPAp_TA0_SW	RO	69	Comp_A+	Timer_A0	Software
RO_PINOSC_TA0_SW	RO	70	Digital I/O	Timer_A0	Software

B.1.4 Sensor Abstractions

B.1.4.1 Button(s)

```
uint8_t TI_CAPT_Button(Sensor *);
```

Inputs: Pointer to Sensor, which 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.

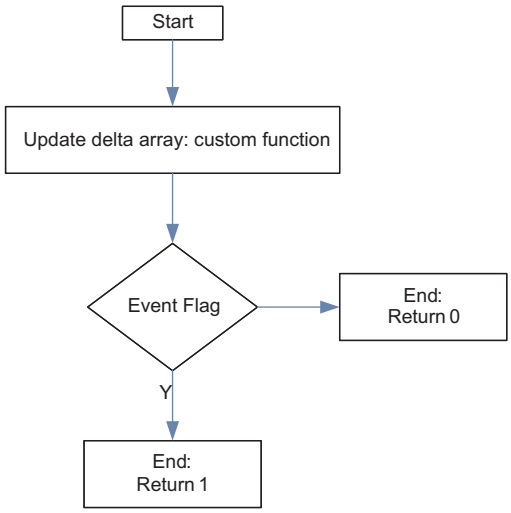


Figure 36. Single Button Algorithm

Element * TI_CAPT_ Buttons(Sensor *);

Inputs: Pointer to Sensor, which 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'.

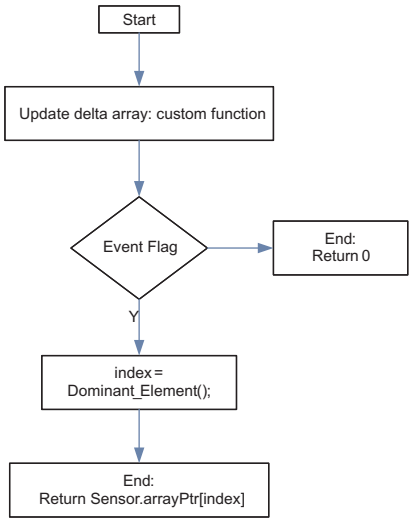


Figure 37. Array of Buttons Algorithm

B.1.4.2 Slider/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 38. 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 B.1.3.1).

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.

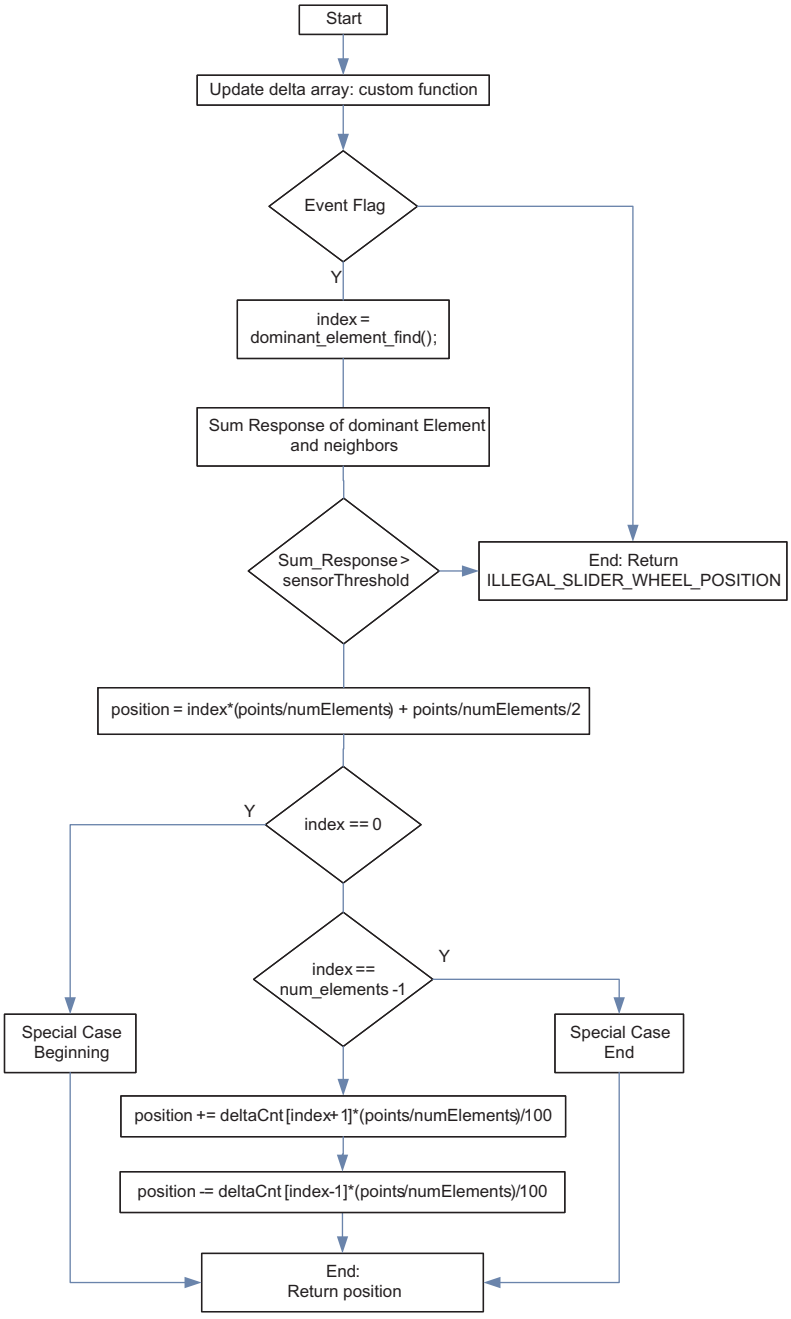


Figure 38. Slider/Wheel Algorithm

B.1.4.2.1 Slider Detection

The slider sensorThreshold is a compared with the response of the dominant element and its neighbors. As shown in Figure 39, The endpoints are a special case that requires a comparison of only the end element (the dominant element) and the one neighbor.

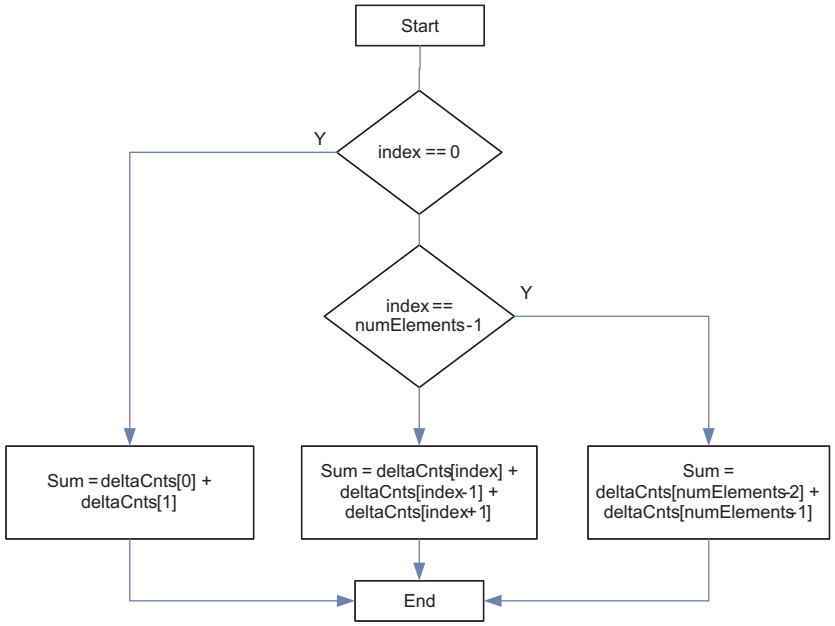


Figure 39. Slider Threshold Detection

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.

B.1.4.2.2 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 neighbor(s). 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 40, the slider has 64 positions and there are four elements in the slider array.

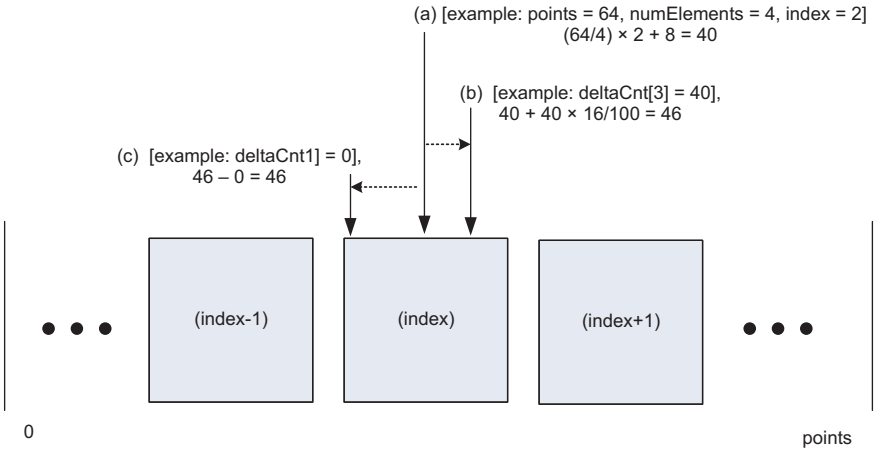


Figure 40. Slider/Wheel Process Middle Algorithm

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.

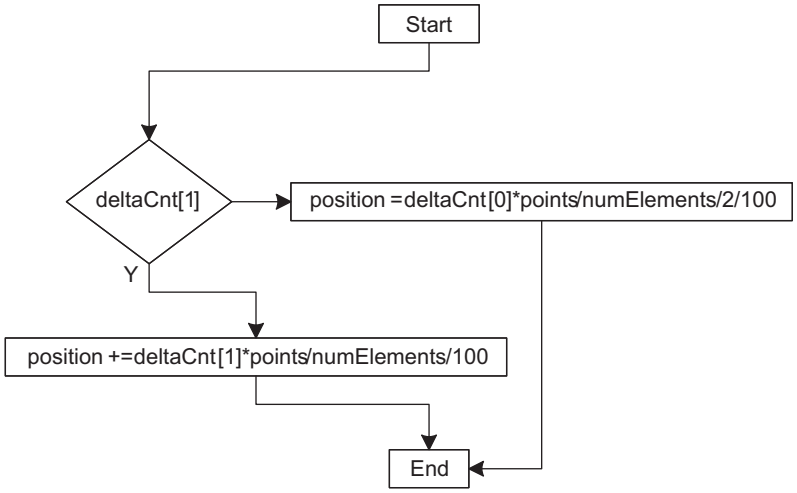


Figure 41. Slider Algorithm: Beginning of Slider

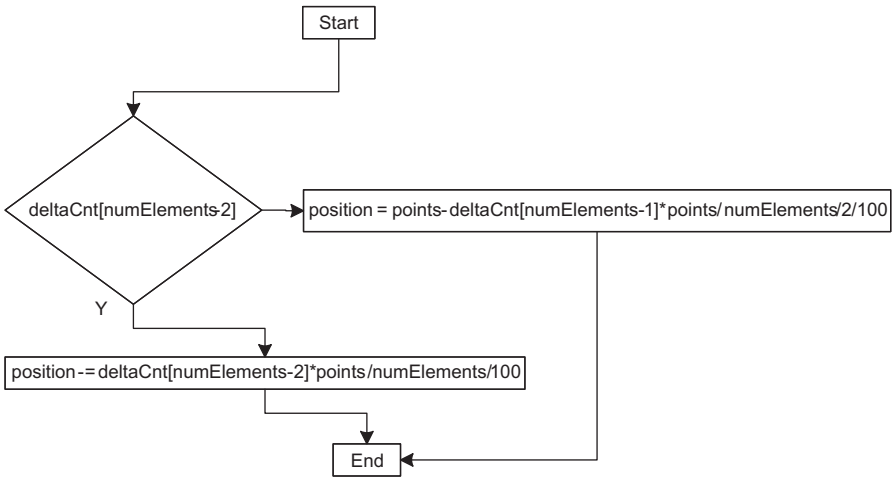


Figure 42. Slider Algorithm: End Of Slider

B.1.4.2.3 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 that requires the 'wrap around' to be accounted for, see Figure 43. 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.

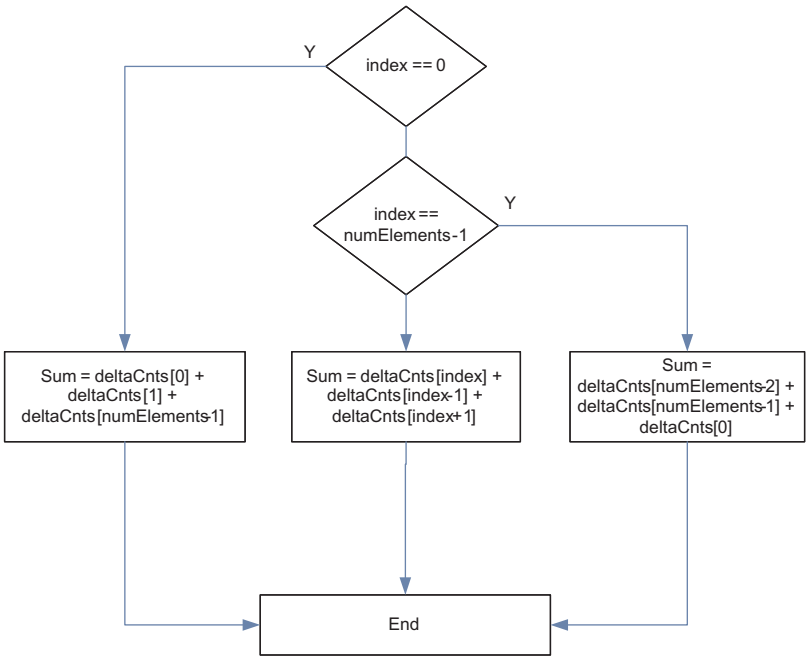


Figure 43. Wheel Threshold Detection

B.1.4.2.4 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 44 and Figure 45 show the algorithm for calculating the position when the dominant element is the beginning and end elements of the array,

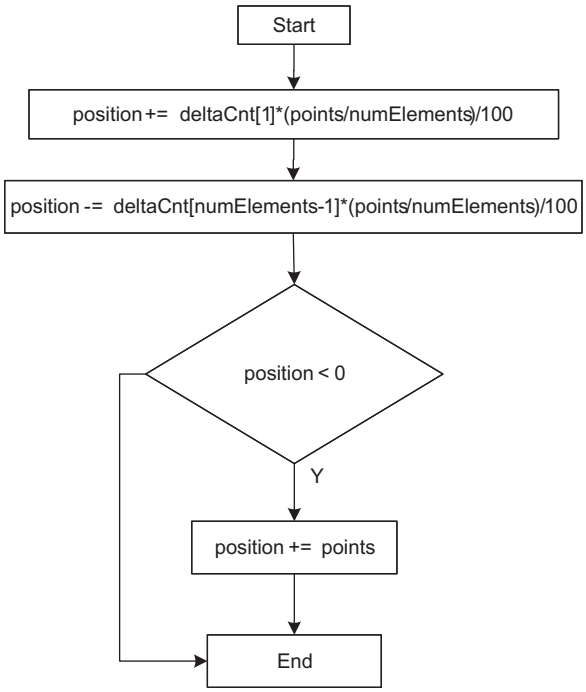


Figure 44. Wheel Algorithm: Beginning

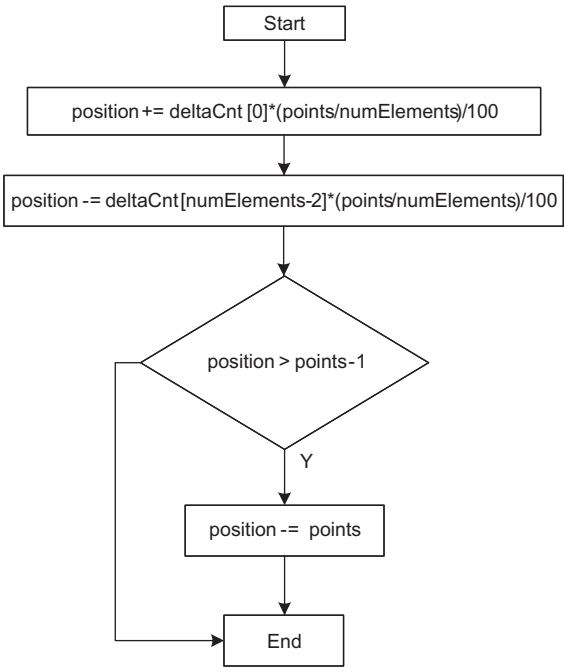


Figure 45. Wheel Algorithm: Ending

B.1.5 Dominant Element Identification

The identification of a threshold crossing actually takes place in the base capacitance update function (see Section B.1.3.1). 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 would indicate that the response is equal or less than the threshold and a value of 100 would indicate a response equal to the maxResponse.

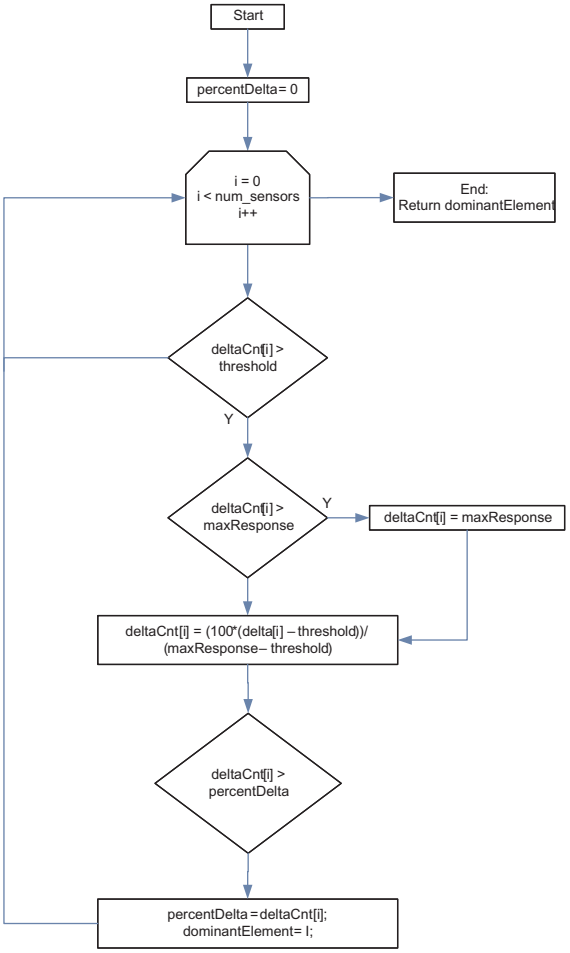


Figure 46. Dominant Element Identification

Appendix C Beta Testing

The number of combinations and permutations of settings and applications make a completely comprehensive test plan unrealistic. At this time, the functionality of all possible use cases cannot be ensured. Table 25 is intended as a proxy to help customers identify if their application has not been tested for compatibility with the library. For example, the RO_PINOSC_TA0_WDTp measurement method was tested with the custom and wheel APIs. This method was also tested with the button API, while another measurement method was used to support the wheel API. The button example is intended to show interoperability with different measurement methods and API calls.

Table 25. Beta Testing

	HAL	Functional	Sensor	Additional HAL (1)	Additional Sensor (2)
1	TI_CTS_RO_COMPAp_TA0_WDTp_HAL	TI_CAPT_Custom	TI_CAPT_Slider		
2	TI_CTS_fRO_COMPAp_TA0_SW_HAL	TI_CAPT_Custom	TI_CAPT_Slider		
3	TI_CTS_fRO_COMPAp_SW_TA0_HAL	TI_CAPT_Custom	TI_CAPT_Slider		
4	TI_CTS_RO_COMPAp_TA1_WDTp_HAL	TI_CAPT_Custom			
5	TI_CTS_fRO_COMPAp_TA1_SW_HAL	TI_CAPT_Custom			
6	TI_CTS_RC_PAIR_TA0_HAL	TI_CAPT_Custom			
7	TI_CTS_RO_PINOSC_TA0_WDTp_HAL	TI_CAPT_Custom	TI_CAPT_Wheel	6	Button
8	TI_CTS_RO_PINOSC_TA0_HAL	TI_CAPT_Custom	TI_CAPT_Wheel	7	Button
9	TI_CTS_fRO_PINOSC_TA0_SW_HAL	TI_CAPT_Custom	TI_CAPT_Wheel	8	Button
10	TI_CTS_RO_COMPB_TA0_WDTA_HAL	TI_CAPT_Custom	TI_CAPT_Slider	10	Second slider
11	TI_CTS_fRO_COMPB_TA0_SW_HAL				
12	TI_CTS_RO_COMPB_TA1_WDTA_HAL	TI_CAPT_Custom	TI_CAPT_Button		
13	TI_CTS_fRO_COMPB_TA1_SW_HAL	TI_CAPT_Custom	TI_CAPT_Button		

(1) The Additional HAL shows that the HAL functions are independent and do not impact the baseline tracking or other shared function calls.

(2) The Additional Sensor shows that the sensor functions are independent and do not corrupt other sensors that also use the baseline tracking and other functions.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com	Wireless	www.ti.com/wireless-apps
RF/IF and ZigBee® Solutions	www.ti.com/lprf		
	TI E2E Community Home Page		e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265