AN0828: Capacitive Sensing Library Overview

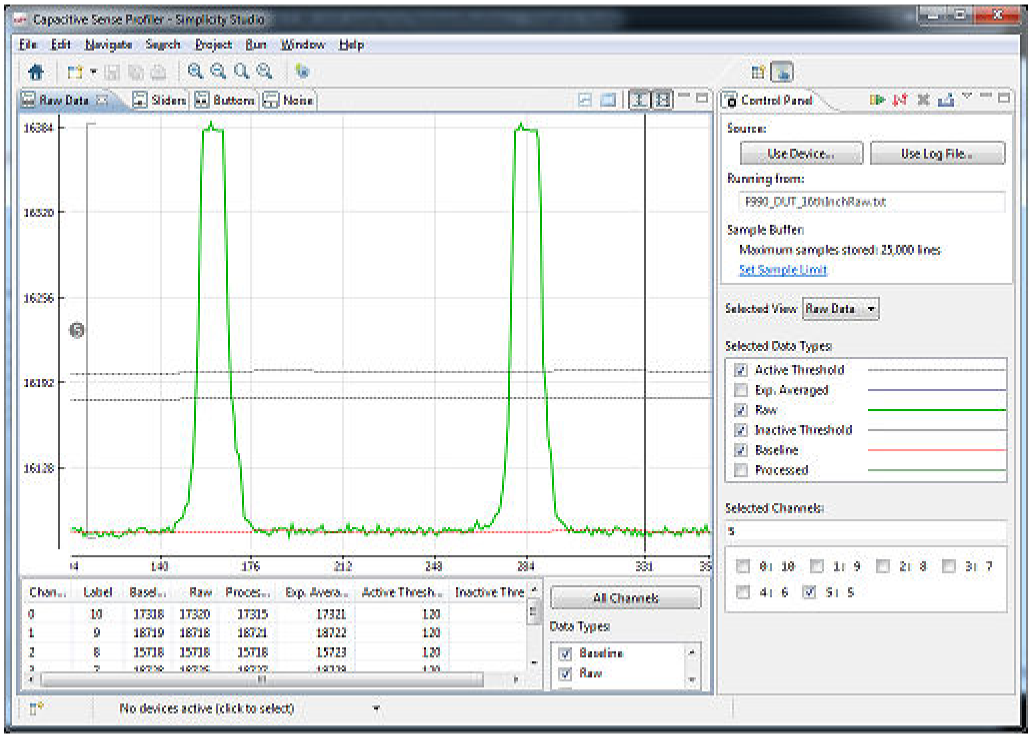
The capacitive sensing library is a production-ready library including all the features and algorithms required to make a capacitive sensing product.

The capacitive sensing library includes the following:

- Scanning routines that buffer raw samples.
- Filters that process and shape raw data.
- Baselining routines that maintain an expected inactive sensor capacitive value.
- Threshold detection routines that qualify touches using application-defined thresholds, choosing between filtered data types dynamically depending on the system's signal to noise ratio.
- Interference characterization routines used to control touch qualification.
- Low power control routines that transition the system between active mode and low power Sleep mode.

KEY POINTS

- Add capacitive sensing to a project in just a few steps
- With minimal setup, the library can scan and process inputs as well as qualify touches
- Easy-to-use APIs enable firmware to detect a touch and respond efficiently



Chan...	Label	Basel...	Raw	Proces...	Exp. Avera...	Active Thresh...	Inactive Thre
0	10	17318	17320	17315	17321	120	
1	9	18719	18718	18721	18722	120	
2	8	15718	15718	15718	15723	120	

Data Types: [✓] Baseline [✓] Raw

Source:

Running from: F990_DUT_16thInchRaw.txt

Sample Buffer:
Maximum samples stored: 25,000 lines
Set Sample Limit

Selected View: Raw Data

Selected Data Types:
[✓] Active Threshold
[ ] Exp. Averaged
[✓] Raw
[✓] Inactive Threshold
[✓] Baseline
[ ] Processed

Selected Channels: 5
[ ] 0: 10 [ ] 1: 9 [ ] 2: 8 [ ] 3: 7
[ ] 4: 6 [✓] 5: 5

No devices active (click to select)

1. Overview

This document and the library refer to the electrode connected to a capacitive sensing input pin on a Silicon Labs MCU as a sensor. A sensor whose capacitance is in an off state, where no conductive object is in proximity to that sensor, is called inactive. The inactive state is qualified by comparing the output of the sensor to an inactive threshold maintained by the library. Firmware qualifies a sensor as active when output crosses an active threshold maintained by the library.

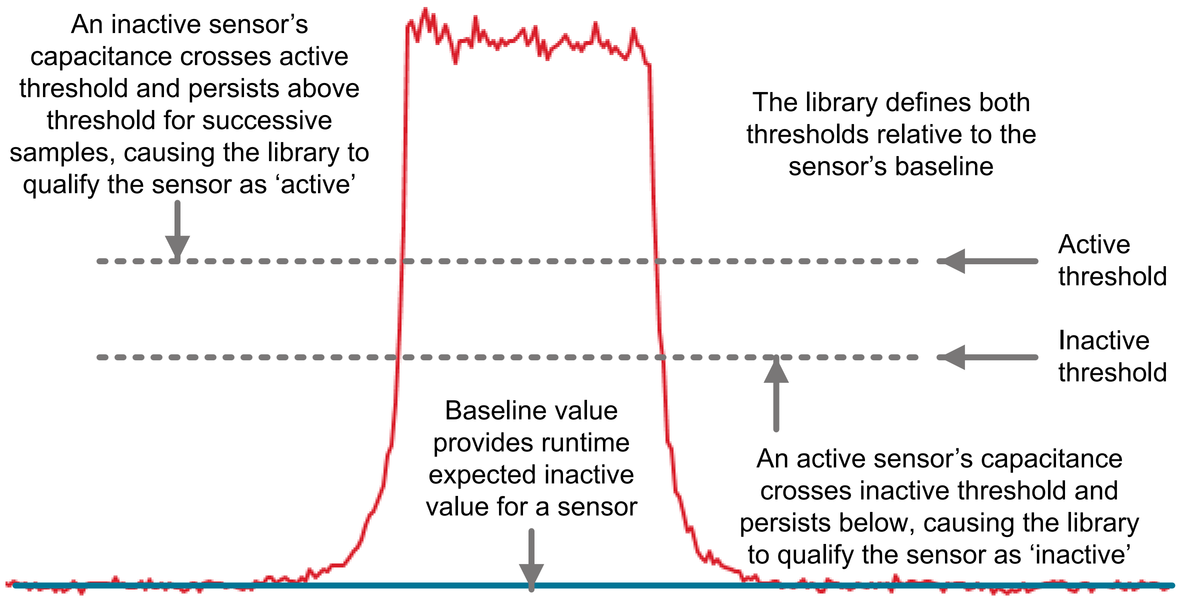

An inactive sensor's capacitance crosses active threshold and persists above threshold for successive samples, causing the library to qualify the sensor as 'active'
The library defines both thresholds relative to the sensor's baseline
Active threshold
Inactive threshold
Baseline value provides runtime expected inactive value for a sensor
An active sensor's capacitance crosses inactive threshold and persists below, causing the library to qualify the sensor as 'inactive'

Figure 1.1. Basic Library Functionality

The library also includes low power features that enable the system to switch between an active mode, where sensors are either active or have been active in the recent past, and sleep mode, which configures the MCU to its lowest power, wake-on-touch state. This document covers the configuration of the library as well as the APIs and data structures the library exposes and maintains for access by an application layer of firmware.

The capacitive sensing library is available on EFM32 devices and 8-bit devices with the capacitive sensing module (EFM8SB1, C8051F99x, and C8051F97x). See *AN0028: Low Energy Sensor Interface — Capacitive Sense* for more information on the EFM32 hardware capacitive sense implementation.

2. Capacitive Sensing Project Structure Overview

The capacitive sensing project structure's design enables users to drop their own routines and functionality into an existing project with minimal editing of source code and other predesigned components. The basic project structure is shown in the figure below.

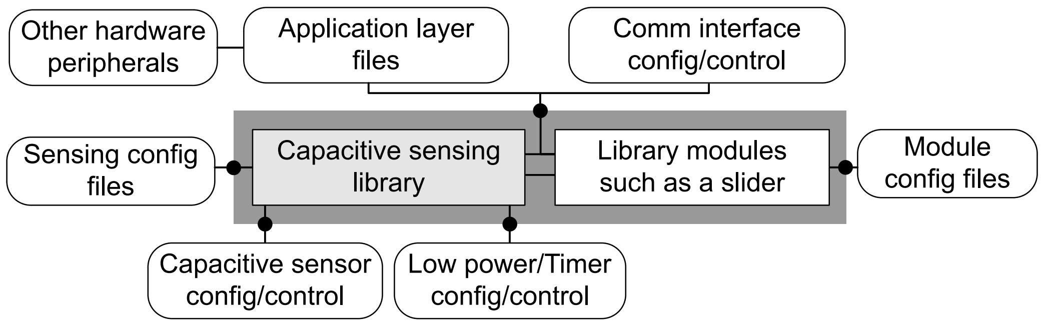

Other hardware peripherals
Application layer files
Comm interface config/control
Sensing config files
Capacitive sensing library
Library modules such as a slider
Module config files
Capacitive sensor config/control
Low power/Timer config/control

Figure 2.1. System Overview

The following sections describe each of these components in greater detail. A review of this structure helps to fully understand how hardware resources are being allocated. However, to concentrate only on capacitive sensing configuration, see *AN0829: Capacitive Sensing Library Configuration Guide*. Application notes can be found on the Silicon Labs website (http://www.silabs.com/8bit-appnotes) and in Simplicity Studio. The firmware examples included with Simplicity Studio use the naming conventions in the above figure and throughout this document.

For EFM32 examples, the configuration of the hardware block is slightly different from the 8-bit implementation, though similar in structure. See the examples included in Simplicity Studio v4 (Gecko SDK v4.4.0 or later) for more information.

3. Application Layer Files

This layer of a capacitive sensing project is the primary place to add application-specific code. The main() routine is located in this section. In the example files, the `profiler_interface.c/h` routines provide an example of how code can interface with the library data structures to output capacitive sensing information. The protocol created by `profiler_interface.c` is compatible with Simplicity Studio's Capacitive Sensing Profiler and enables realtime display of capacitive sensing output and library-maintained variables. Board examples included with Simplicity Studio show how touch detection can control the state of LEDs and other features.

The `main()` routine includes the top-level calls into the capacitive sensing API. Calls initialize capacitive-sensing controlled hardware and capacitive sensing state variables before entering a `while(1)` loop where the library's functionality is exercised through `CSLIB_update()`. A call to the low-power state machine is also present, although this call can be removed if low-power features have not been enabled.

Note that the application layer must include function definitions for a set of callback routines used by the library. The requirements for these functions are discussed in 7. Library Execution and Data Access.

4. Communications Layer

The Simplicity Studio code examples include a file group called the Comm Interface. The code in that interface provides one example of how data can be retrieved from the Capacitive Sensing Library data structures to be output through a serial interface. Also, the protocol used by the examples is specially formatted for the Capacitive Sense Profiler. For a description of that protocol, see AN0829.

5. Sensing Config Files

The Configuration Files layer includes the files listed in the table below; each file has a specific function. Note that Simplicity Studio's hardware configurator provides capacitive sensing firmware library configuration support for capacitive sensing-enabled MCUs. For supported MCUs, we suggest configuring and generating capacitive sensing projects through that tool instead of editing files manually as described in the sections below.

Table 5.1. Config Layer Files

File Name	Function Description
`cslib_config.c`	Contains variable declarations for data used by the capacitive sensing library that may also need to be used by other library modules or by the application layer. No changes should be made to this file.
`cslib_config.h`	Contains performance controls for the library's routines such as sampling frequency, number of sensors, and other controls. For recommendations on configuring this file, see AN0829.
`cslib_hwconfig.c`	Stores arrays of data used for technology-specific hardware configuration of the capacitive sensing peripheral. Data defined here is used by functions in `hardware_routines.c` to configure the sensing peripheral. For a guide on how to configure this file for your application, see AN0829.
`cslib_hwconfig.h`	Defines the technology-specific structure used to collect all configuration information defined in `cslib_hwconfig.c`. The contents of this file should not change for projects using the same sensing technology.
`hardware_routines.c`	Contains function definitions for all code that configures the capacitive sensing technology to scan a sensing input. This file contains callback functions that must be defined in order for the library to function properly. The required callback routines are defined in 7.3 Capacitive Sensing Hardware Callback Functions.
`cslib.h`	This header file includes declarations for all callback functions used by the library that must be defined in an application's source code. It also includes a definition of the library's data structure, which stores all state information for each sensor. These variables can be accessed directly in source code, and some variables and buffers can be accessed through the library API, which is also declared in this header file. That API is discussed in 7.3 Capacitive Sensing Hardware Callback Functions.

5.1 Values in `ProjectConfig.h`

The table below lists the definitions found in `cslib_config.h`, their meanings, and suggested values.

Table 5.2. `cslib_config.h` Definitions

`ProjectConfig.h` Definition	Meaning	Units	Suggested Value
DEF_NUM_SENSORS	Defines the size of the sensor node array as well as other non-volatile and volatile arrays that have a one-to-one correspondence to the number of sensors in the project.	Elements of the Sensor Node Structure declared in `cslib.h`	The number of sensors to scan when in active mode, usually all sensors in a project.
DEF_AVERAGE_TOUCH_DELTA	Determines the expected difference between untouched sensor output and touched sensor output.	Sensor output codes	• For projects using the CS0 peripheral with a finger-sized sensor and 1/8th inch overlay: 300 codes. • For projects using the CS0 peripheral with a 1/16th inch overlay: 500 codes.
DEF_INACTIVE_SENSOR_DELTA	Defines the threshold relative to a sensor's baseline value below which an active sensor will be qualified as inactive.	Sensor output codes	• For projects using CS0 with 1/8 inch overlay, 85 and 70, respectively. • For 1/16 inch, 150 and 130.
DEF_ACTIVE_SENSOR_DELTA	Defines the threshold relative to a sensor's baseline value above which an inactive sensor will be qualified as active.	Sensor output codes	• For projects using CS0 with 1/8 inch overlay, 200. • For projects using CS0 with a 1/16 inch overlay, 300.
DEF_BUTTON_DEBOUNCE	Sets the number of consecutive samples that must rise above the sensor's active delta threshold before a sensor is characterized as active.	# of samples	More debounce counts yields a sensor that is more robust against spurious noise events, at the expense of responsiveness. This cost of responsiveness can be offset by a high sample frequency. For most systems, 2–4 sample debounce is sufficient.
DEF_ACTIVE_MODE_PERIOD	Defines the periodicity at which an MCU running in active mode will execute a sample sequence and scan all sensors. This value has a relationship to DEF_- FREE_RUN_SETTING below. Also, this parameter assumes correct configuration of the timer configured in the Low power config file group.	ms	A higher period means less frequent scanning and lower current draw if FREE_RUN_SETTING = 0. Touch responsiveness decreases as the period gets higher.
DEF_SLEEP_MODE_PERIOD	Defines the periodicity at which an MCU running in sleep mode will execute a sample sequence and scan all sensors. Also, this parameter assumes correct configuration of the timer configured in the low power config file group.	ms	A higher periodicity lowers current draw because the device will stay in its low power state for a higher percentage of time. Increasing the periodicity lowers wake-on-touch responsiveness.
DEF_COUNTS_BEFORE_SLEEP	Value used to determine when an active mode to sleep mode transition occurs. If no sensors are qualified as active, that transition will occur in a time period defined as DEF_COUNTS_BEFORE_SLEEP x DEF_ACTIVE_MODE_PERIOD.	Conversion counts	The number of consecutive active mode scans without a charactrized touch that must be seen before the system switches to sleep mode.

`ProjectConfig.h` Definition	Meaning	Units	Suggested Value
FREE_RUN_SETTING	If set to 1, a system in active mode will scan sensors as fast as system resources allow, regardless of DEF_ACTIVE_- MODE_PERIOD value. If set to 0, an active mode system will scan sensors once per DEF_ACTIVE_MODE_ PERIOD be-fore falling into a low-power state.	0 or 1	For fastest response time, set to 1 to enable free running scans. To con-serve power in active mode, set to 1.
SLEEP_MODE_ENABLE	Set this to 1 to enable sleep mode sensing; set this to 0 to always re-main in active mode.	0 or 1	Setting to 1 enables the low-power li-brary to enter sleep mode to con-serve power.

The library's usage of the timer and low power configuration functions are shown in Figure 5.1 State Machine Operation with FREE_RUN_SETTING = 1 on page 8 and Figure 5.2 State Machine Operation with FREE_RUN_SETTING = 0 on page 9.

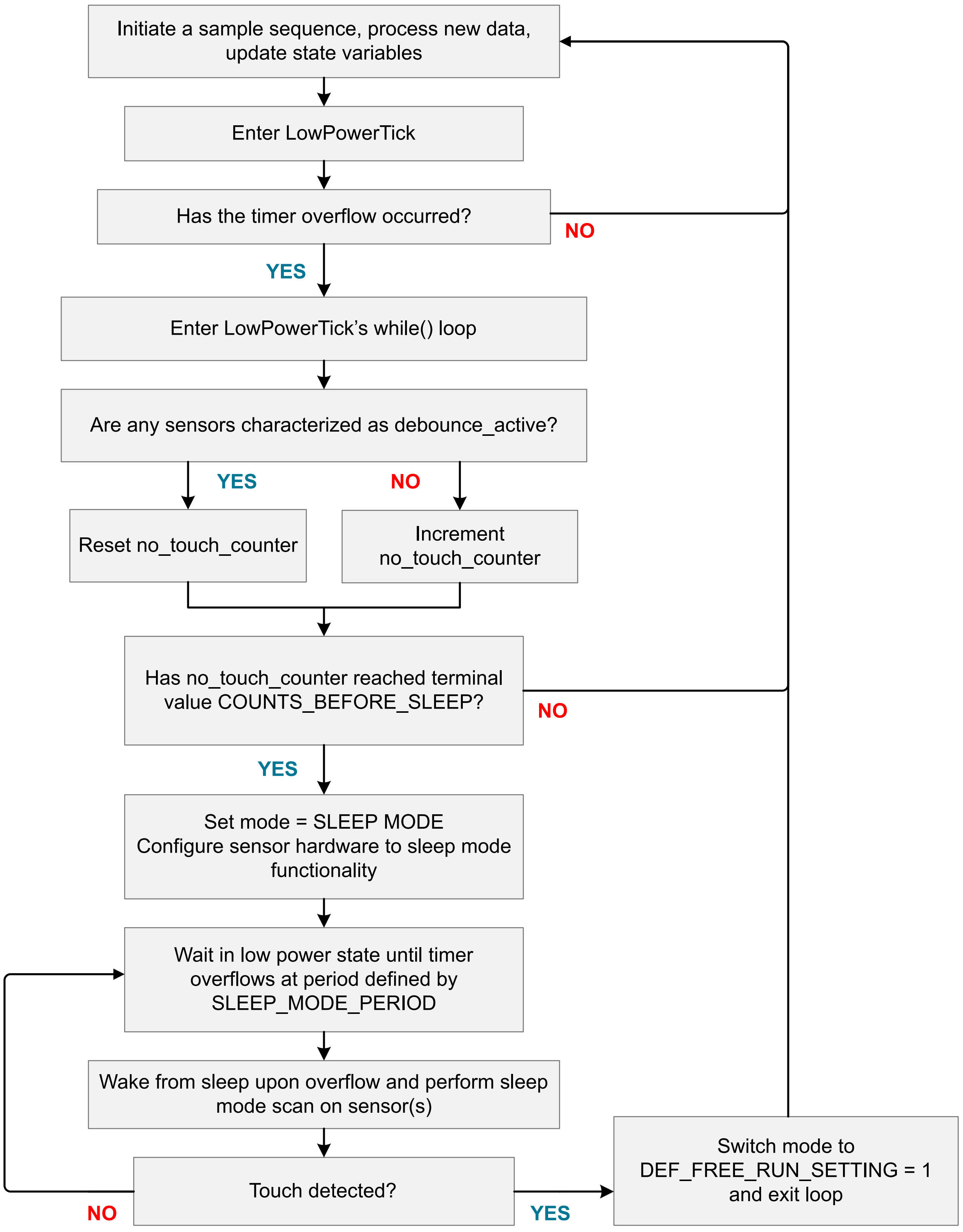

Initiate a sample sequence, process new data, update state variables
Enter LowPowerTick
Has the timer overflow occurred?
NO
YES
Enter LowPowerTick's while() loop
Are any sensors characterized as debounce_active?
YES
NO
Reset no_touch_counter
Increment no_touch_counter
Has no_touch_counter reached terminal value COUNTS_BEFORE_SLEEP?
NO
YES
Set mode = SLEEP MODE Configure sensor hardware to sleep mode functionality
Wait in low power state until timer overflows at period defined by SLEEP_MODE_PERIOD
Wake from sleep upon overflow and perform sleep mode scan on sensor(s)
Touch detected?
NO
YES
Switch mode to DEF_FREE_RUN_SETTING = 1 and exit loop

Figure 5.1. State Machine Operation with FREE_RUN_SETTING = 1

Figure 5.2 State Machine Operation with FREE_RUN_SETTING = 0 on page 9 shows operation of the low-power state machine when using the ACTIVE_MODE_ACTIVE_TIMER mode.

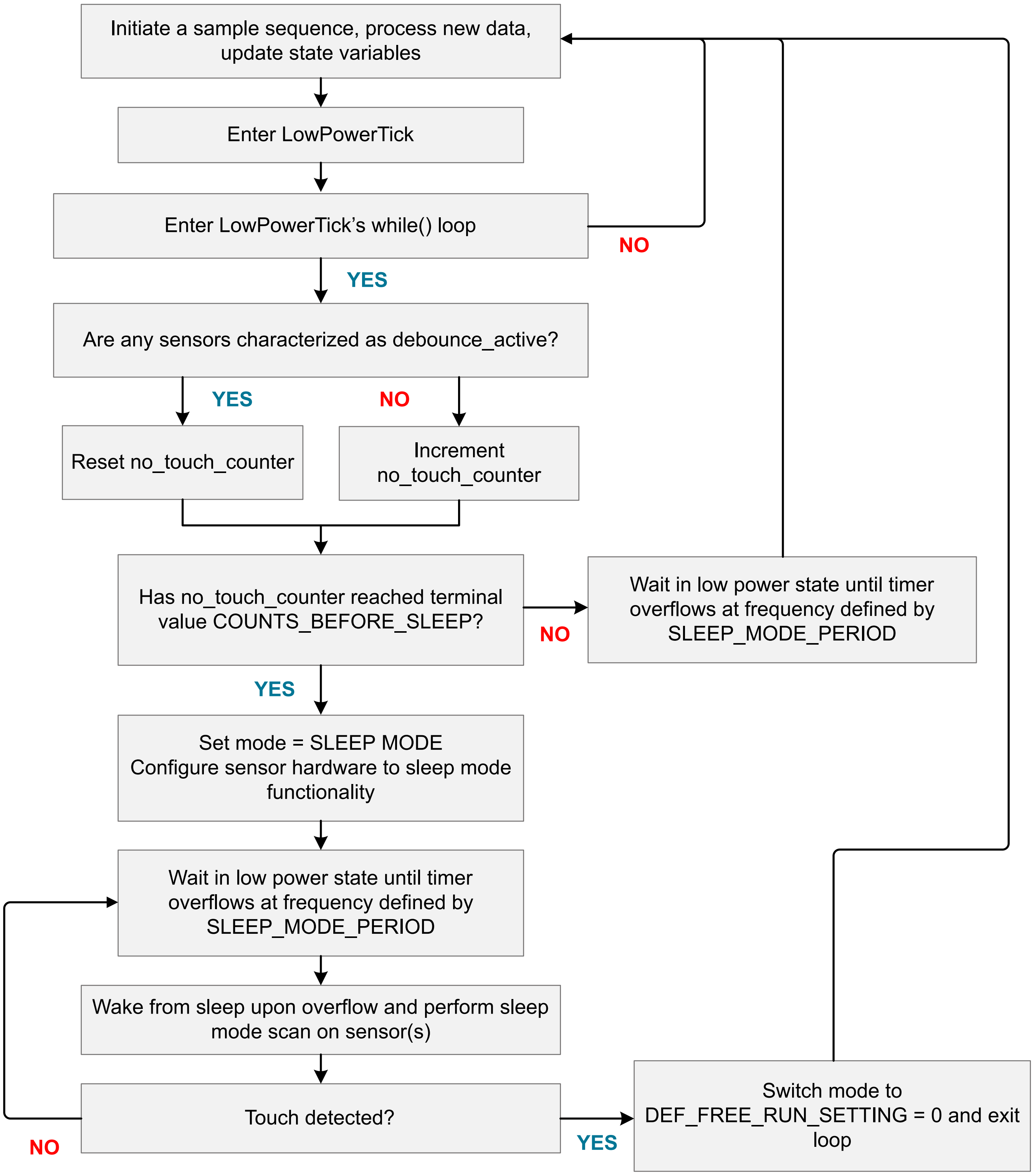

Initiate a sample sequence, process new data, update state variables
Enter LowPowerTick
Enter LowPowerTick's while() loop
NO
YES
Are any sensors characterized as debounce_active?
YES
NO
Reset no_touch_counter
Increment no_touch_counter
Has no_touch_counter reached terminal value COUNTS_BEFORE_SLEEP?
NO
Wait in low power state until timer overflows at frequency defined by SLEEP_MODE_PERIOD
YES
Set mode = SLEEP MODE Configure sensor hardware to sleep mode functionality
Wait in low power state until timer overflows at frequency defined by SLEEP_MODE_PERIOD
Wake from sleep upon overflow and perform sleep mode scan on sensor(s)
Touch detected?
NO
YES
Switch mode to DEF_FREE_RUN_SETTING = 0 and exit loop

Figure 5.2. State Machine Operation with FREE_RUN_SETTING = 0

5.2 Processed Data Array

The data processing filter runs a dejitter algorithm and saves the output into the processed data buffer for that channel. The dejitter algorithm behaves as shown in the figure below. A sensor's new value is not saved to the processed data buffer unless that value falls outside of an envelope surrounding the previous value that has been defined at compile time.

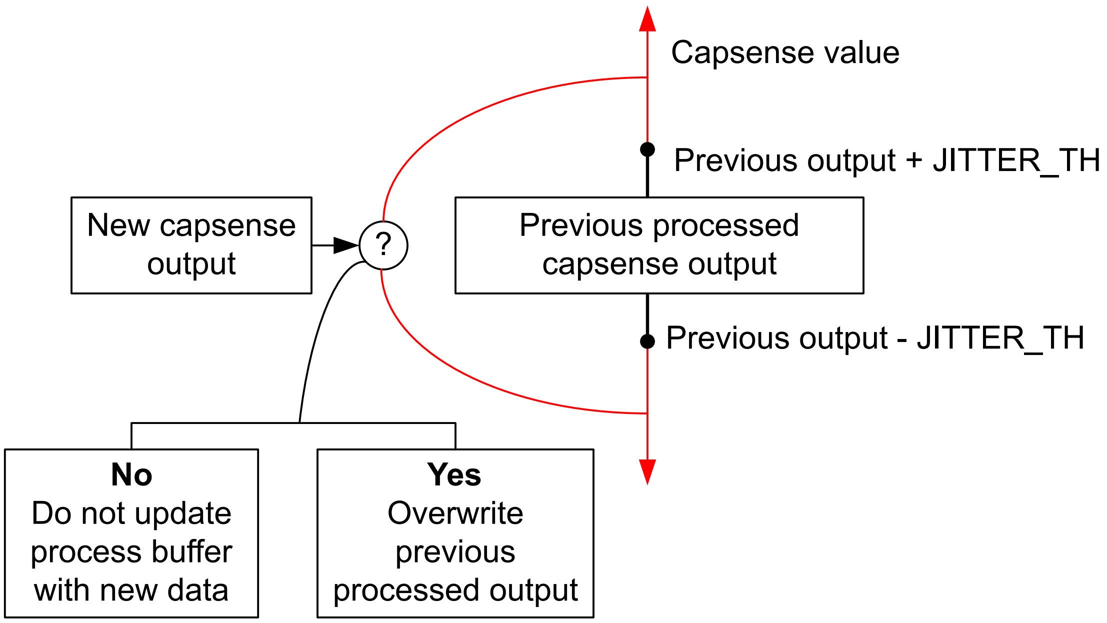
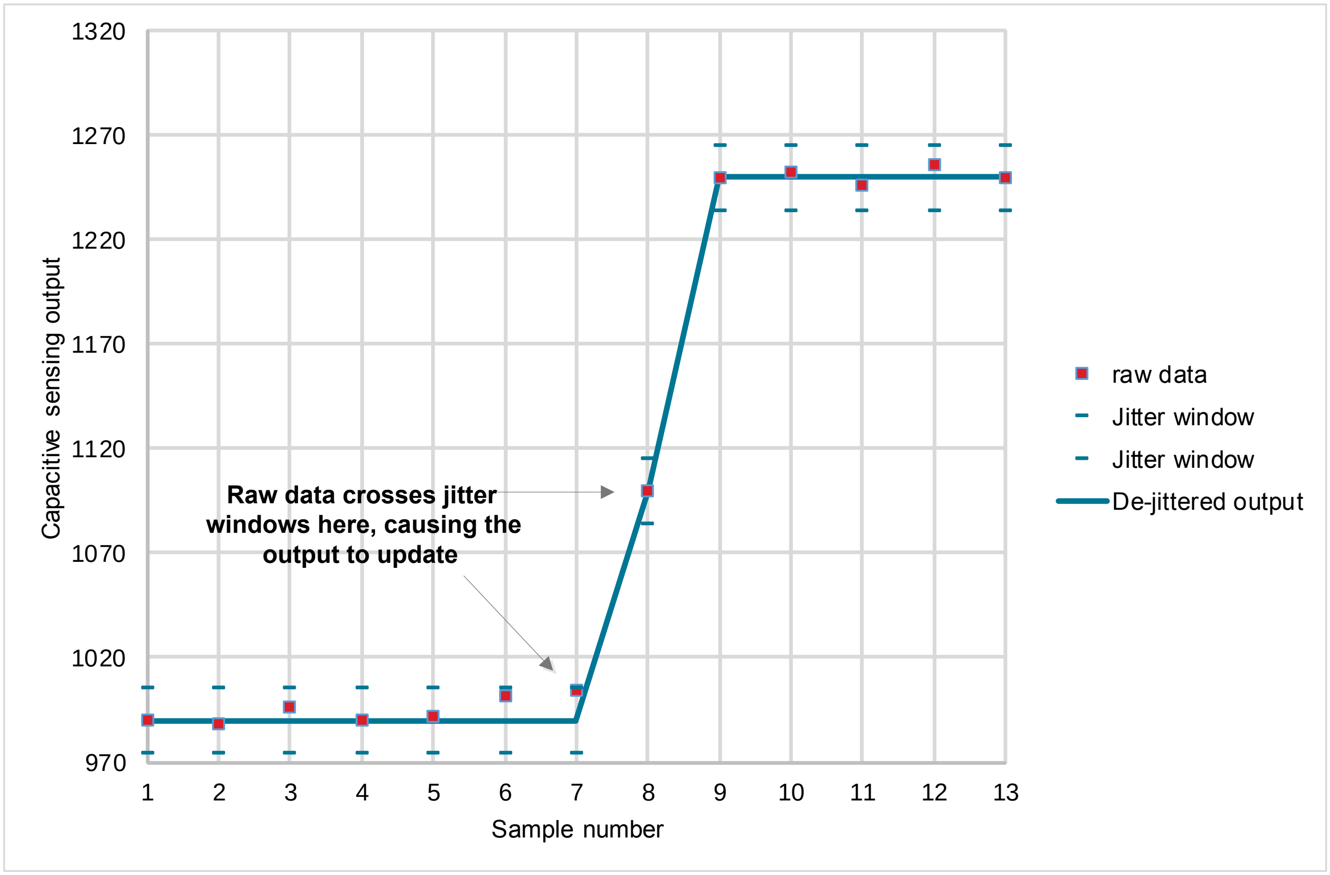

Capsense value
Previous output + JITTER_TH
New capsense output
?
Previous processed capsense output
Previous output - JITTER_TH
No
Do not update process buffer with new data
Yes
Overwrite previous processed output

Figure 5.3. Dejitter Algorithm

This algorithm is useful for routines that benefit from a steady capacitive sensing output, when small variations in raw output are not necessary for processing. In a low-noise environment, this processing algorithm yields a high degree of responsiveness. An example of the effect of the de-jitter processing can be seen in the chart below. Note that the algorithm tracks a 200-code touch but ignores the minor variations in untouched and touched capacitive sensing output.

Figure 5.4. Dejitter Filter Response Example



5.3 Exponential Averaged Values

The exponential averaging filter provides a filter for higher noise environments. The algorithm uses a form of exponential averaging described by the following equation:

$$Y[n] = \left(\frac{1}{\text{exp. av. constant}} \times x[n]\right) + \left(\frac{\text{exp. av. constant} - 1}{\text{exp. av. constant}} \times y[n-1]\right)$$

The exponential averaging constant is 64, which provides a balance between aggressive filtering and acceptable response time for 1–40 sensors when DEF_FREE_RUN_SETTING = 1. This filter will result in a touch detection that has a slower response time than the other filters included in the algorithm, but the exponentially averaged output is more reliable than the other filter options in terms of filtering out false positive events. Note that each sensor's exponentially-averaged value is stored in a 4-byte variable in the sensor node structure. In this variable, the upper 2 bytes show the averaged value, analogous to the 2-byte raw or processed values for that sensor. The lower 2 bytes preserve LSBs from the division inherent in the averaging filter and yield more reliable results from the filter.

6. Touch Qualification

The library characterizes activity on a sensor using hysteresis and debouncing to screen out false-positive events. The algorithm uses inactive and active thresholds defined above baseline as shown in Figure 6.1 Derivation of Touch Delta for a Sensor on page 12 and Figure 6.2 Derivation of Active and Inactive Thresholds for a Sensor on page 13. The debounce state machine used by the algorithm is defined in Figure 6.3 Threshold Detection State Machine on page 14, and a visual representation of the state machine is included in Figure 6.4 Example Threshold Detection State Machine Processing on page 15.

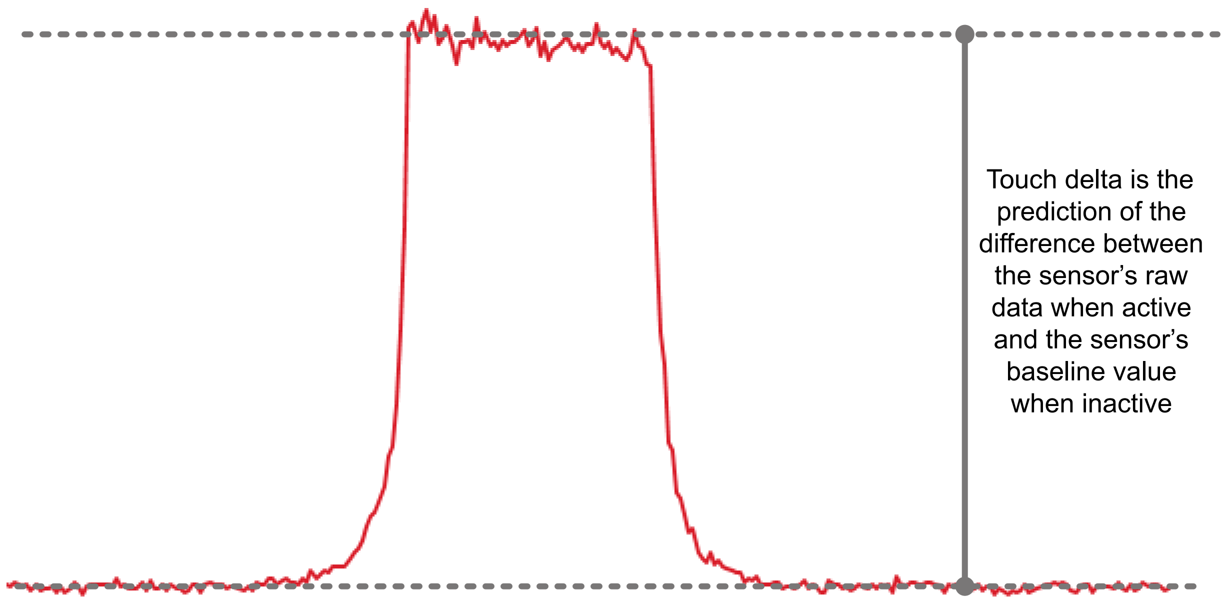


Figure 6.1. Derivation of Touch Delta for a Sensor

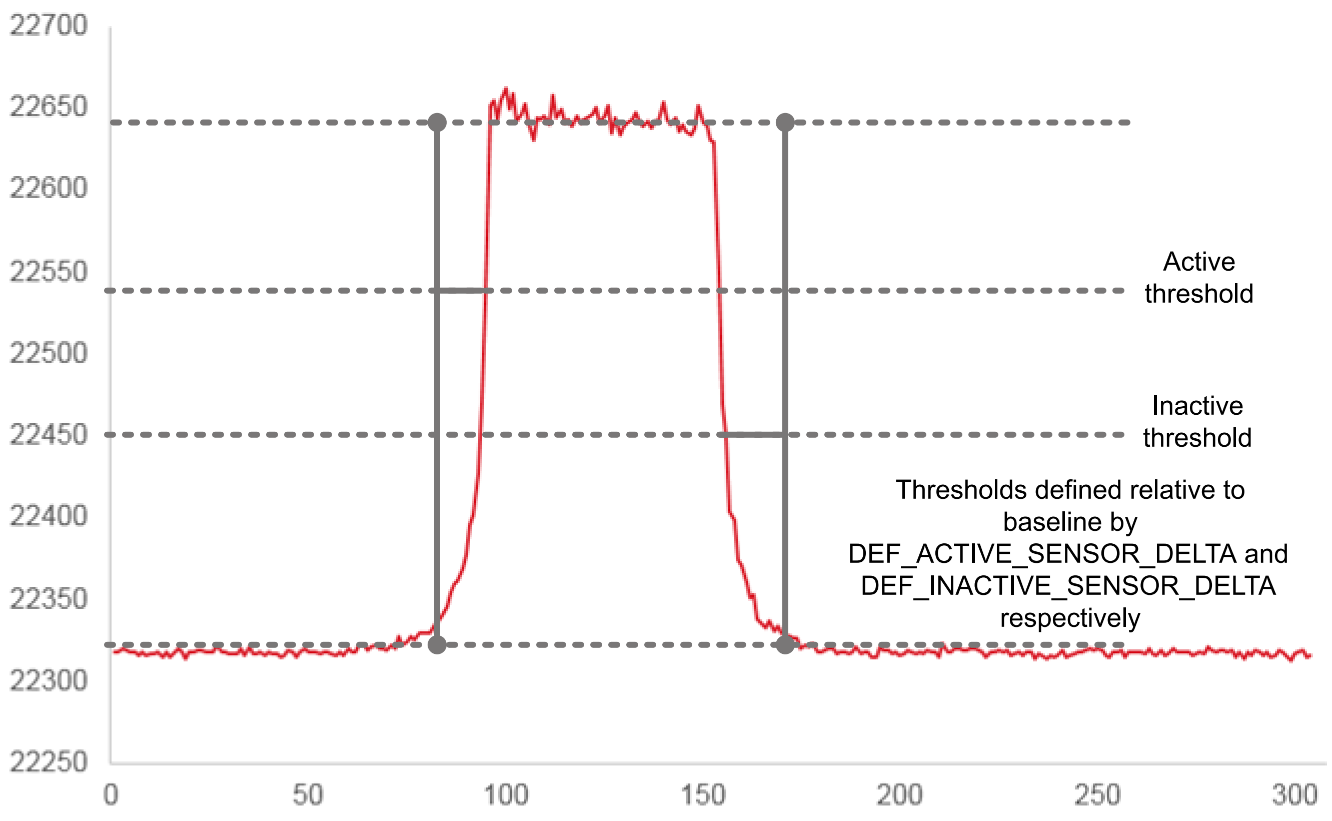


Figure 6.2. Derivation of Active and Inactive Thresholds for a Sensor

Once the algorithm derives the touch delta and thresholds for the sensor, the function analyzes the position of the newest sensor data relative to the thresholds, and updates the sensor's state information saved in debounce_active_count and debounce_inactive_count to derive the new threshold state. The threshold state determination state machine executes as described in the figure below.

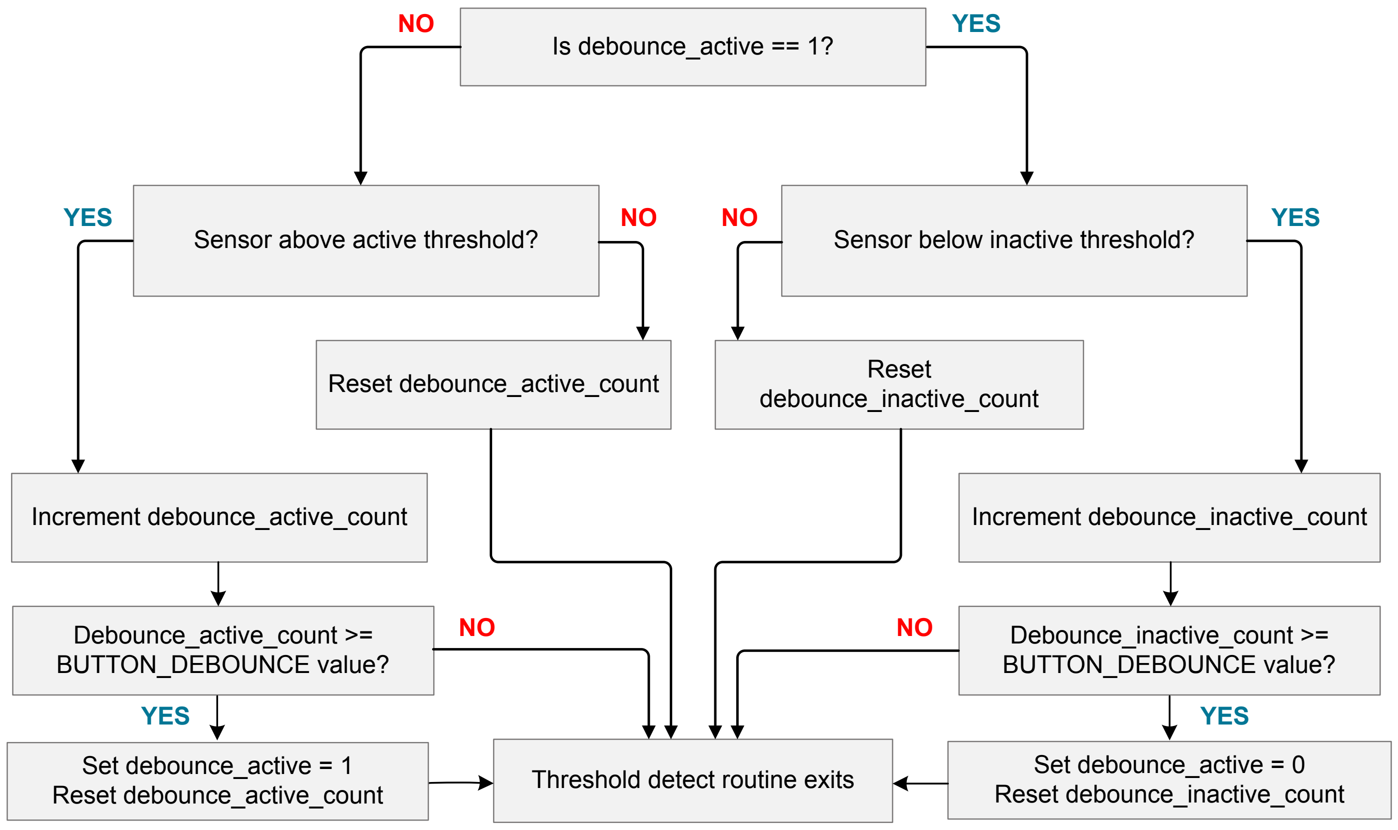

NO
Is debounce_active == 1?
YES
YES
Sensor above active threshold?
NO
NO
Sensor below inactive threshold?
YES
Reset debounce_active_count
Reset debounce_inactive_count
Increment debounce_active_count
Increment debounce_inactive_count
Debounce_active_count >= BUTTON_DEBOUNCE value?
NO
NO
Debounce_inactive_count >= BUTTON_DEBOUNCE value?
YES
YES
Set debounce_active = 1
Reset debounce_active_count
Threshold detect routine exits
Set debounce_active = 0
Reset debounce_inactive_count

Figure 6.3. Threshold Detection State Machine

The figure below shows the process of qualifying a new touch event. Note that the debounce_active_count begins incrementing when the capacitive sensing values cross the active threshold, and the event is not qualified as an active sensor until a set number of consecutive sensor samples above threshold have occurred. In this example, BUTTON_DEBOUNCE is set to a value of 3, and the data used in the threshold detection function consists of processed, de-jittered samples.

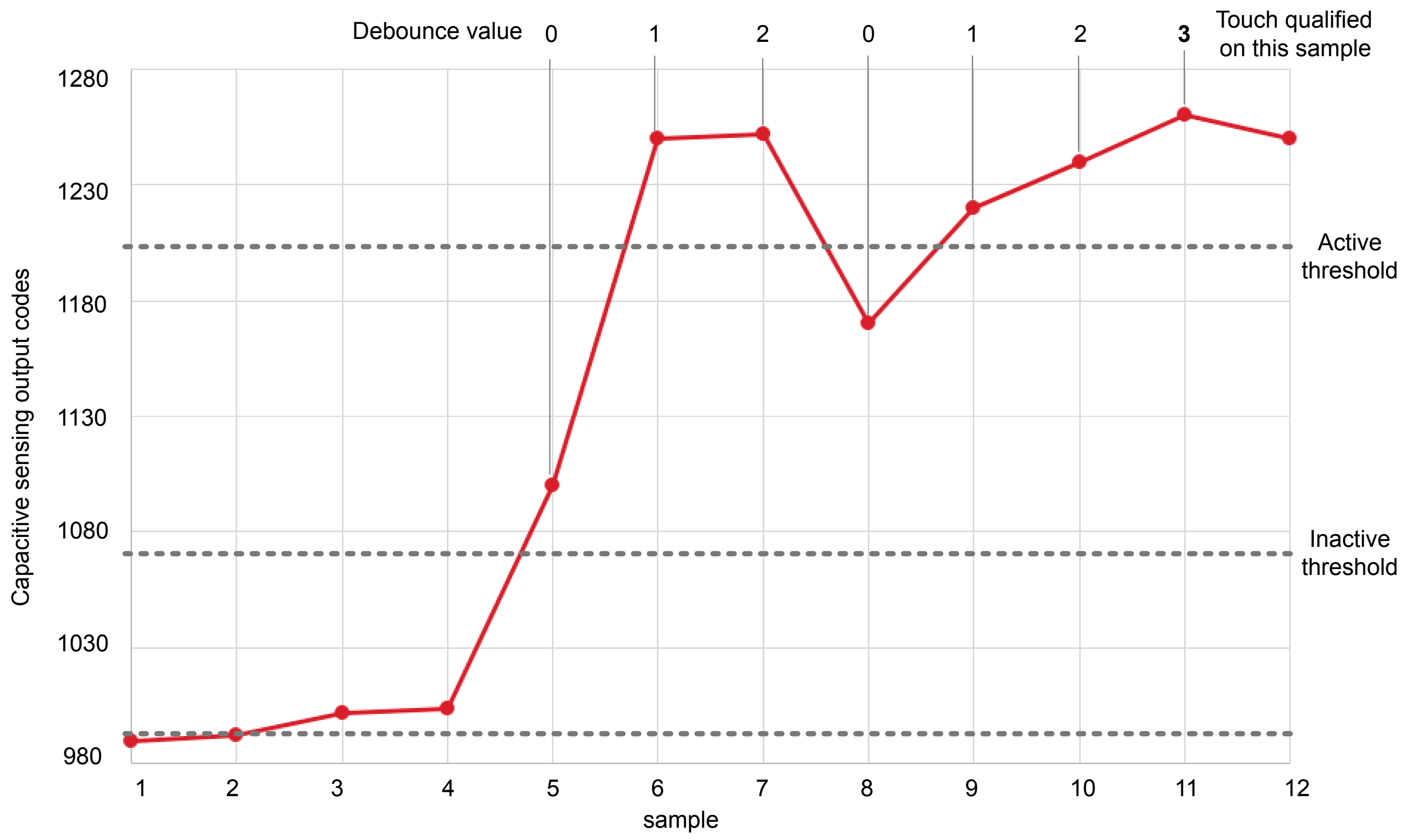


Figure 6.4. Example Threshold Detection State Machine Processing

7. Library Execution and Data Access

The library executes code in the foreground rather than backgrounding algorithms in asynchronous interrupts. Because code executes in the foreground, the application layer must call a few routines to scan sensors, process data, and update the system's state.

The Simplicity Studio code examples contain a main.c routine that shows the suggested coding structure and placement of the top-level library APIs. Initialization of the library occurs after reset by calling `CSLIB_initHardware()` and `CSLIB_initLibrary()`. These two functions initialize all library state variables and technology-specific hardware. For more information on allocated hardware, see the sections below called capacitive sensing hardware callback routines and low power and timer configuration callback routines.

Inside the `main()` routine's `while(1)` loop, the function `CSLIB_update()`updates the library's state. `CSLIB_update()` executes all scanning, processing, and touch qualification. After that function returns, the library's data structures have been updated and new data is ready to for use by the application layer code.

Also in the `while(1)` look is the function `CSLIB_lowPowerUpdate()`, which controls the power state of the MCU depending on settings in `cslib_config.h`. Without this call in the build, the system will not be able to enter its sleep mode.

7.1 Sensor Data Structure

The library saves the state of all defined sensors in a data structure defined in CSLibrary.h as the SensorNode struct. The contents of that structure are defined in the table below.

Table 7.1. SensorNode Data Structure Contents

Variable	Description
`rawBuffer`	Array storing raw samples read from a sensor starting with the current sample and ending with sample (current-DEF_SENSOR_BUFFER_SIZE – 1). This array can be accessed directly or through a set of access APIs described below.
`processBuffer`	Array storing raw samples processed and shaped, starting with the current processed sample and ending with sample (current-DEF_SENSOR_BUFFER_SIZE – 1). This array can be accessed directly or through a set of access APIs described below.
`currentBaseline`	The library's baseline for the sensor, which is the expected off/inactive state of that sensor.
`touchDeltaDiv16`	The expected difference between the fully active value of a sensor and the sensor's baseline value. A sensor can be considered fully active if a conductive object such as a finger is in closest possible proximity to the entire surface area of that sensor's electrode. This value is compressed so that the 4 LSBs are not defined in RAM. To determine the touch delta for the sensor, touchDeltaDiv16 should be right-shifted by 4. This shifted value can be accessed using an API described below.
`activeIndicator`	This byte describes the touch state of the sensor. Bit 6 indicates that the sensor is in a candidate 'single' touch state, where the last capacitive sensing scan shows that the sensor's capacitance has crossed the DEF_SINGLE_ACTIVE threshold. Bit 7 indicates whether the sensor is in a qualified 'debounced' touch state by crossing the defined active threshold and persisted above that threshold for a number of samples defined by DEF_BUTTON_DEBOUNCE. This value can be accessed directly through the structure or through a set of APIs defined below. These values can be accessed using APIs defined below.
`baselineAccumulator`	This value is used by the baseline update algorithm in the library to determine when the current_baseline variable should be adjusted upward or downward to track the expected sensor output when in an inactive state.
`expValue`	This is a filtered version of raw data that has been put through an aggressive low pass filter.

7.2 Data Access APIs

The library includes some functions to streamline access of some commonly examined variables in the sensor node structure. Each available function is described in the table below. Like the declaration for the sensor node structure, these functions are also declared inside `cslib.h`.

Table 7.2. Data Access API Functions

Function Name	Description
`uint8_t CSLIB_isSensorSingleActive(uint8_t index)`	Returns 1 if any defined sensor's single_active state is 1, returns 0 otherwise.
`uint8_t CSLIB_anySensorSingleActive()`	Returns 1 if any defined sensor's single_active state is 1, returns 0 otherwise.
`uint8_t CSLIB_isSensorDebounceActive(uint8_t index)`	Returns 1 if sensor at `index` has a debounce_active state of 1, returns 0 otherwise.
`uint8_t CSLIB_anySensorDebounceActive()`	Returns 1 if any defined sensor's debounce_active state is 1, returns 0 otherwise.
`uint16_t CSLIB_nodeGetRaw(uint8_t index, uint8_t offset)`	Returns the raw sample of sensor `index` at buffer index newest sample – `offset`.
`uint16_t CSLIB_nodeGetProcess(uint8_t index, uint8_t offset)`	Returns the processed sample of sensor `index` at buffer index newest sample – `offset`.
`uint8_t CSLIB_getInfoArrayPointer()`	Returns a pointer to an array of bytes defining the version information of the library and other characteristics of the library. See 7.4 Low Power and Timer Configuration Callback Routines for more information.
`uint16_tCSLIB_getUnpackedTouchDelta()`	The library stores touch delta with the 4 LSBs removed. This API left-shifts the stored touch delta by 4, resulting in a touch delta that is in units of capacitive sensing output codes.

7.3 Capacitive Sensing Hardware Callback Functions

The library relies on a set of callback routines defined outside the pre-compiled library that are responsible for capacitive sensing and low-power control. The declarations for these functions are found in `hardware_routines.h` and `low_power_config.h`.

The following sections list all capacitive sensing-related callback routines and their requirements. The tables also define the level of modularity of each routine. Function means that changes will need to be made to the function body to meet the requirements of each application. Struct means that the function body won't need to be changed, but changes will need to be made to the contents of the technology-specifc struct provided by the template and example files used to configure the capacitive sensor. Technology means that the functions should not need to be changed across applications, and must only be replaced by other template/example files for different sensing technologies.

Table 7.3. Callback Routines in `hardware_routines.c`

Function Name	Requirements	Modular Constraints
`uint16_t executeConversion(void)`	Must start a capacitive sensing scan without changing configurable parameters of the capacitive sensing block such as the level of accumulation, mux settings, etc. Function must return the sensor's output in a 16 bit value.	Technology
`uint16_t scanSensor(uint8_t index)`	Must perform all technology-specific configuration of the capacitive sensing peripheral (accumulation, mux setting, etc.) and call `executeConversion()`. Function takes as a parameter the sensor index and returns the value returned by `executeConversion()`.	Struct
`void configureSensorForActiveMode(void)`	Must make all all necessary peripheral and port configuration for the system's active mode so that a call to `scanSensor()` will execute correctly. In the examples and template, this function calls a port configuration routine, and the body of this function must be changed depending on which MCU pins are used as capacitive sensing inputs.	Function
`void nodeInit(void)`	This function is used during initialization to populate the technology-specific structure.	Struct

7.4 Low Power and Timer Configuration Callback Routines

The capacitive sensing library makes calls into routines to configure for low power mode, and those calls must be defined in the project's source code. Additionally, the library needs a time base for sensing and for low power timing, and those functions must be defined in source as well. Fortunately, most of these files can be lifted from the examples and template projects as-is.

Table 7.4. Callback Routines in `low_power_config.c`

Function Name	Requirements	Modular Constraints
`void configureSensorForSleepMode(void)`	Must configure whatever port pins are needed to do a scan in sleep mode for wake-on-touch functionality.	Function
`void configureTimerForSleepMode(void)`	Must configure the timer used by the system to wake for a scan during sleep mode.	Technology
`void configureTimerForActiveMode(void)`	Must configure the timer used by the system in active mode for periodic scanning.	Technology
`void enterLowPowerState(void)`	Must configure the low power peripheral to enter its optimal low power mode.	Technology
`void checkTimer(void)`	Must return 1 if the configured timer has reached its overflow/terminal value, which is configured by the `configureTimer…` routines.	Technology

7.5 Info Array Contents

The library includes a function called `uint8_t* CSLIB_getInfoArrayPointer()`, which returns a pointer to an array of bytes describing the characteristics of the library. The first two bytes of the array indicate the version of the library. Based on the version number, firmware can know the total size of the array and what the other contents mean. The version number can be decoded as shown in the table below.

Table 7.5. Byte Number and Contents

Byte Number	Byte Contents
0	Major version number
1	Minor version number

The information array structure is defined version-by-version. For version 1.0 of the library, the information array contains no other contents.

8. Revision History

8.1 Revision 0.3

2016-10-14

Added EFM32 to 1. Overview.

Added mention of EFM32 to 2. Capacitive Sensing Project Structure Overview.

8.2 Revision 0.2

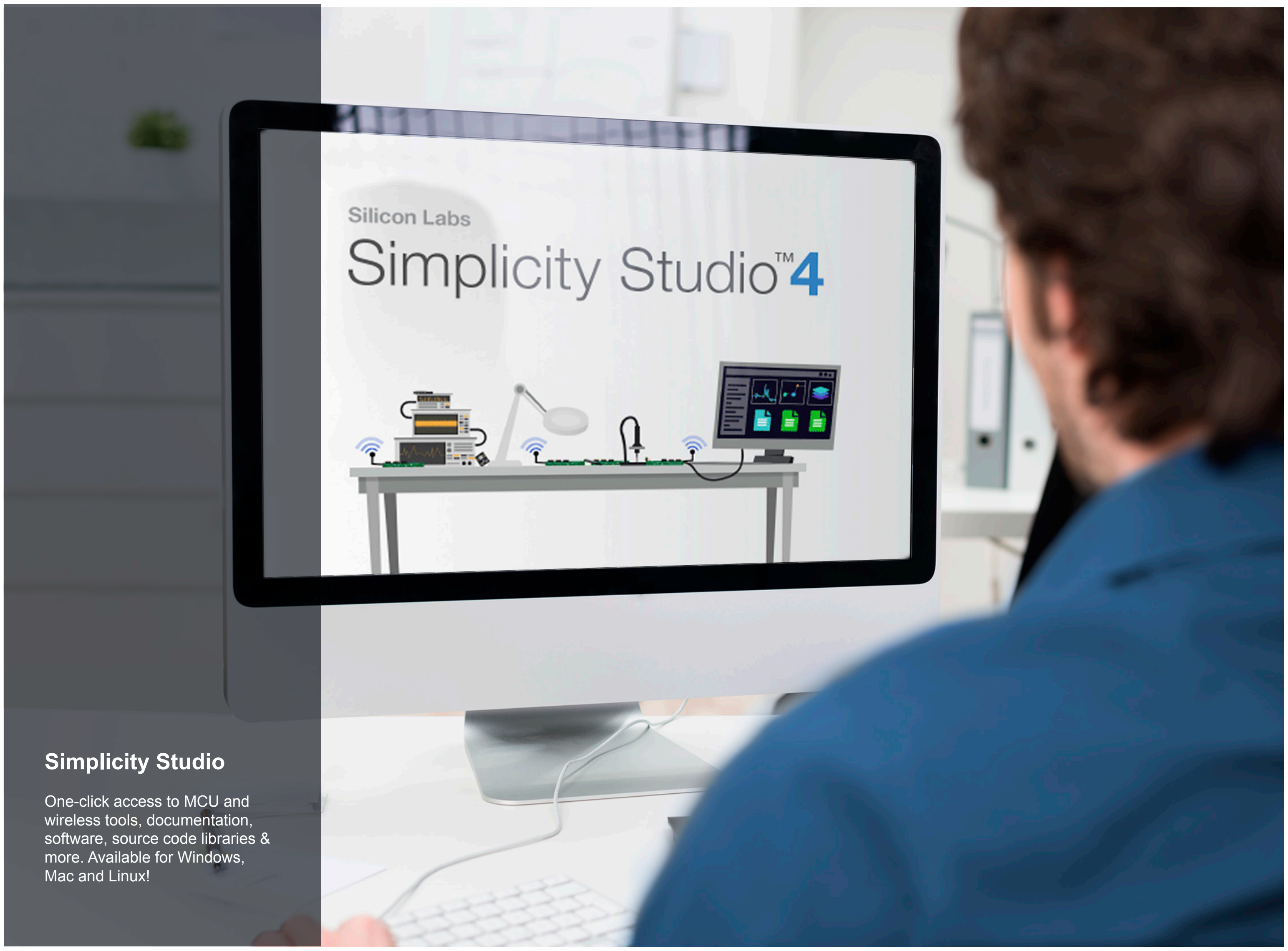
2015-02-13

Updated formatting.

8.3 Revision 0.1

2014-06-13

Initial revision.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio *www.silabs.com/IoT*	**SW/HW** *www.silabs.com/simplicity*	**Quality** *www.silabs.com/quality*	**Support and Community** *community.silabs.com*

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com