AN0829: Capacitive Sensing Library Configuration Guide

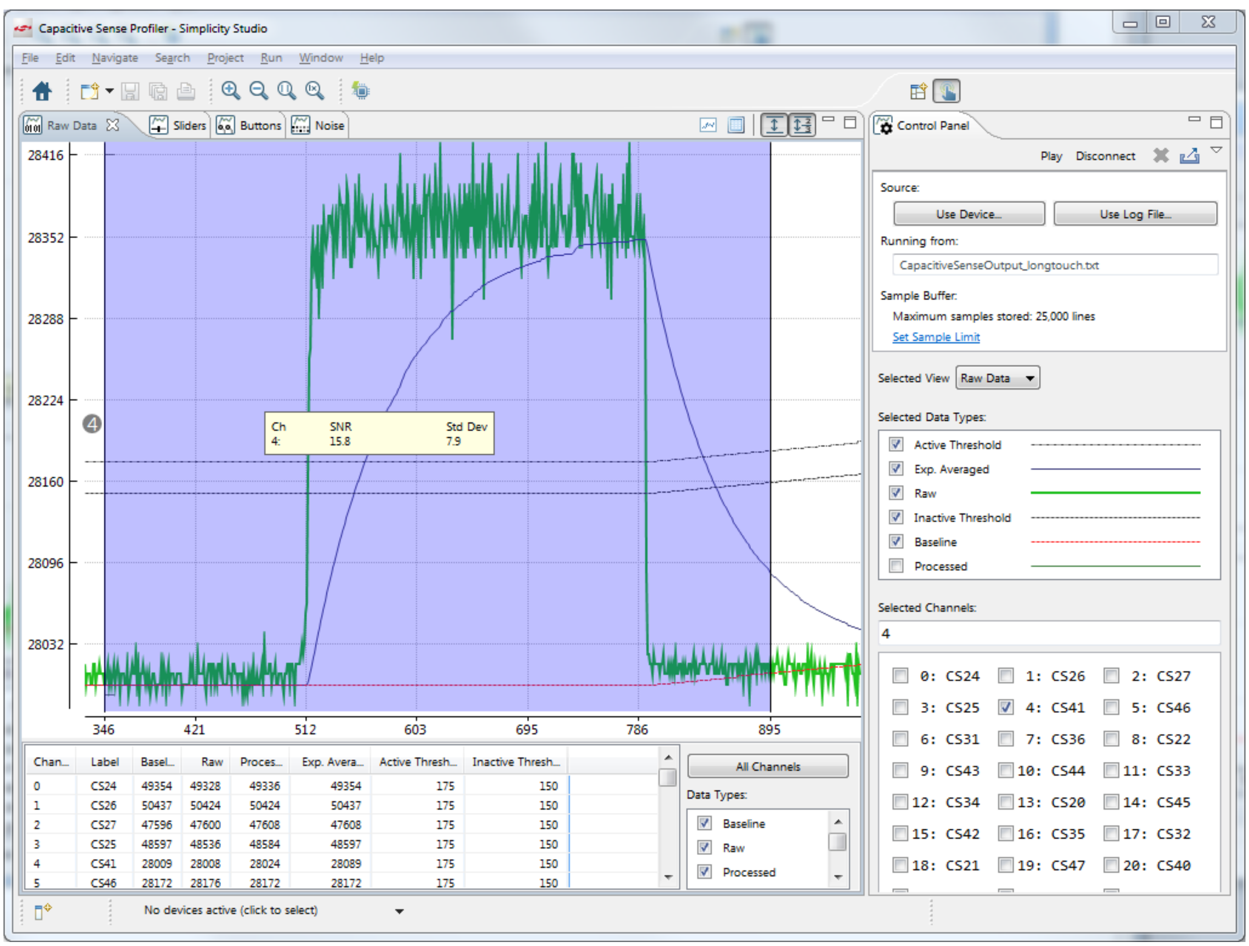
This document describes configuration strategies and recommendations for using the Silicon Labs capacitive sensing library.

Answers to these questions will help tailor the library to suit a particular project.

- What is the pinout for the project?
- How many capacitive sensing buttons will be used?
- Will sliders be used?
- What is the minimum response time required by the application?
- What is the current power budget of the product?

KEY POINTS

- Hardware configuration for capacitive sensing
- Algorithm configuration recommendations for optimal performance



1. Library Overview

For a detailed description of the capacitive sensing library, see AN0828: Capacitive Sensing Library Overview. Application notes can be found on the Silicon Labs website (http://www.silabs.com/8bit-appnotes) or in Simplicity Studio.

The library is divided between technology-specific files that access the MCU's hardware layer and technology-agnostic files that perform scanning, processing, touch qualification, and data storage through calls into the technology-specific files. The firmware operates in one of two active modes and a low power mode. The functionality of the active mode and low power mode can be customized to suit a project's requirements.

2. Getting Started

To get started with a capacitive sensing project, open Simplicity Studio, select the desired capacitive sensing MCU family, and open the demos for that MCU. Next, choose either one of the capacitive sensing demos or the capacitive sensing template file. The EFM8SB1, C8051F99x, and C8051F97x families are currently supported by the capacitive sensing library and have library projects.

Note that Simplicity Studio's hardware configurator can be used to generate and configure the capactive sensing library with the EFM8SB1 MCU.

The sections in this document should be followed to ensure correct project configuration. After completing these steps, a project will have correct configuration for:

- Library performance and data structure.
- Active mode sensing and port settings.
- Slider functionality if needed by the project.
- Low power sensing.

3. Capacitive Sense Profiler Overview

Simplicity Studio includes a Capacitive Sense Profiler utility that enables users to do the following:

- See realtime output of the capacitive sensing peripheral.
- View the algorithmically derived values that have taken raw capacitive sensing data as inputs, such as filtered values, baseline information, sensor activity state, and slider output values.
- Log data to file and review data at a later time.
- Generate SNR and standard deviation information on sensor.
- Selectively view sensors and adjust the visual arrangement on selected sensors.

The Profiler is meant to be used during capacitive sensing application development. Users can switch between the Simplicity IDE and the Profiler, downloading code images, observing behavior, and going back to the IDE to make incremental changes.

3.1 Walkthrough

This walkthrough shows one way that the Profiler can be used in coordination with the Simplicity IDE during development. In this example, we'll start with a project that has a single capacitive sensor with hardware configured as described in the Hardware Configuration section found in this document. For the purposes of this example, start with the example written for a board with no overlay and run the code on a board with a 1/16 inch overlay.

1. Run Simplicity Studio and open the no overlay example.
2. Open Simplicity studio and click on the [**Software Examples**] box as shown in Figure 3.1 Software Examples on page 5.

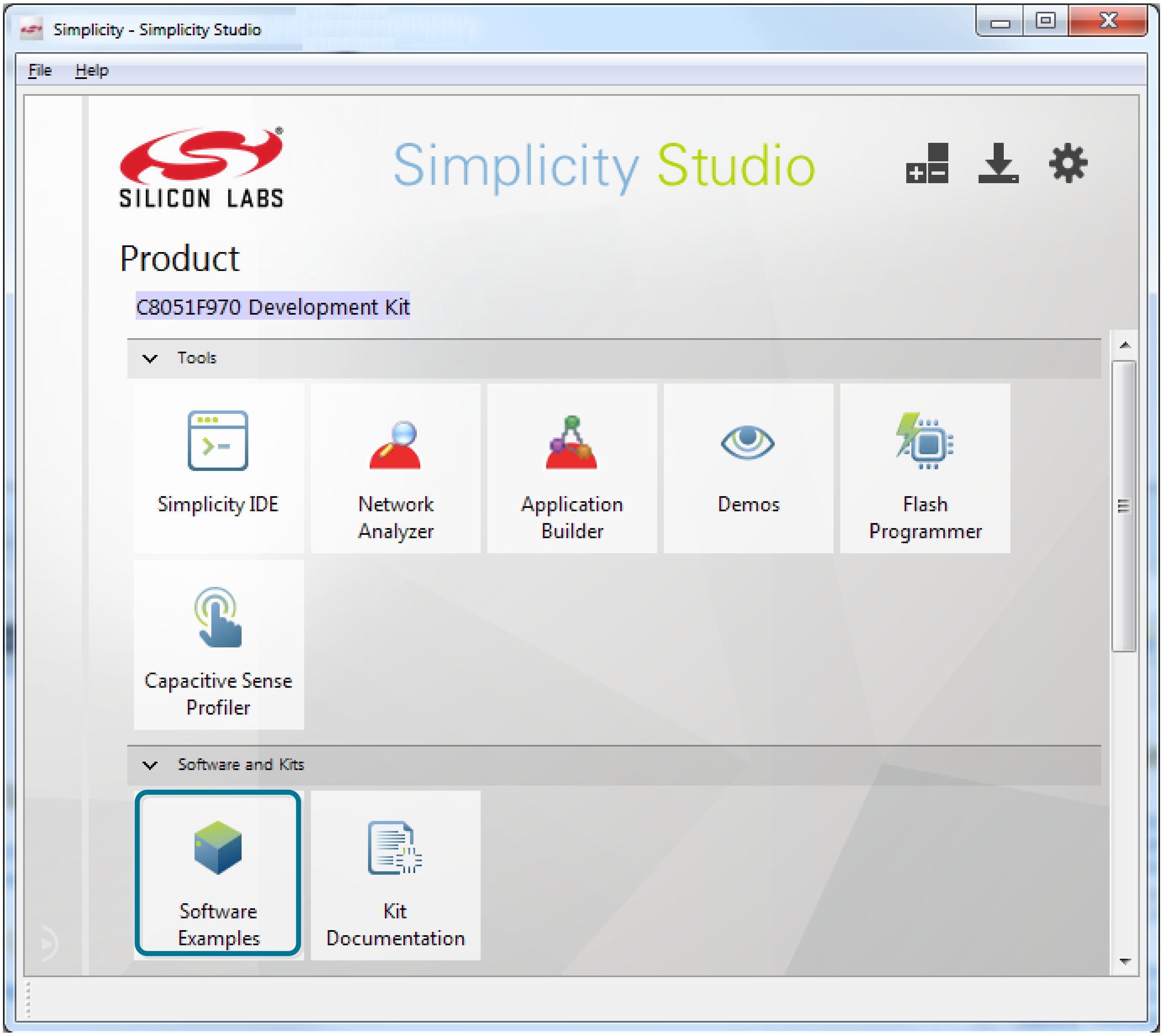


Figure 3.1. Software Examples

After navigating any menus that appear where the device family is chosen, find the [**Capacitive Sense for overlay eval no overlay**] capacitive sense example and open that project as shown in Figure 3.2 Example Selection on page 6.

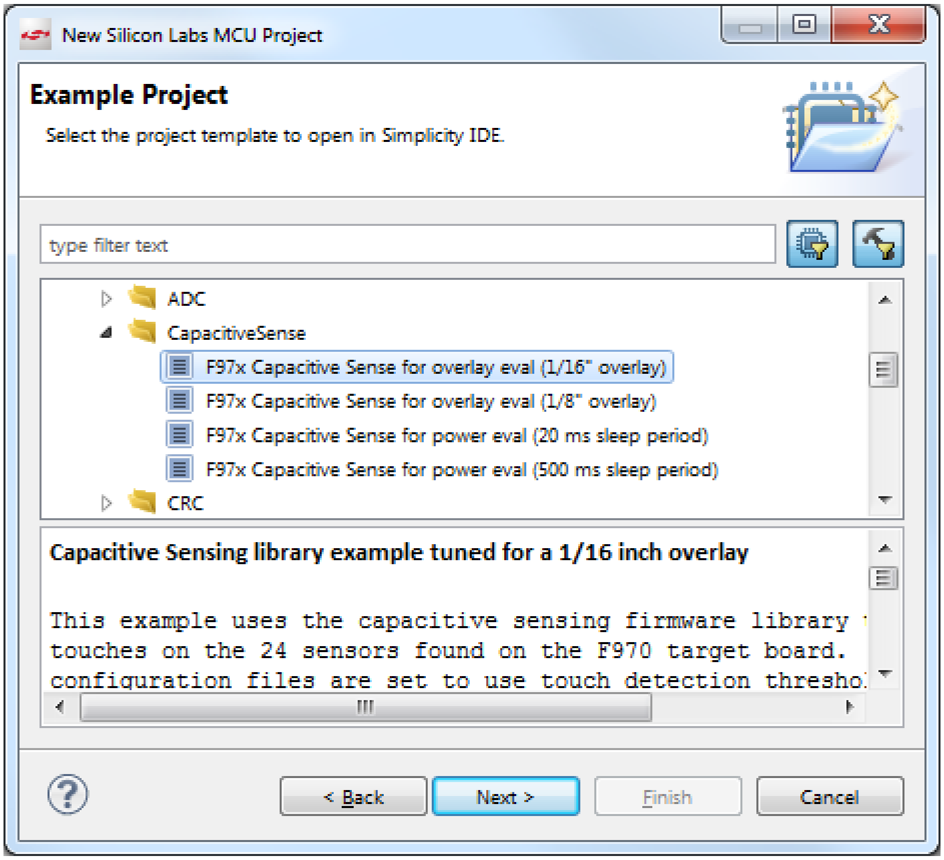


Figure 3.2. Example Selection

Step 1 — Check Thresholds

In the Simplicity IDE, open the file `cslib_config.h`. Open the file as shown in Figure 3.3 Opening `cslib_config.h` on page 7.

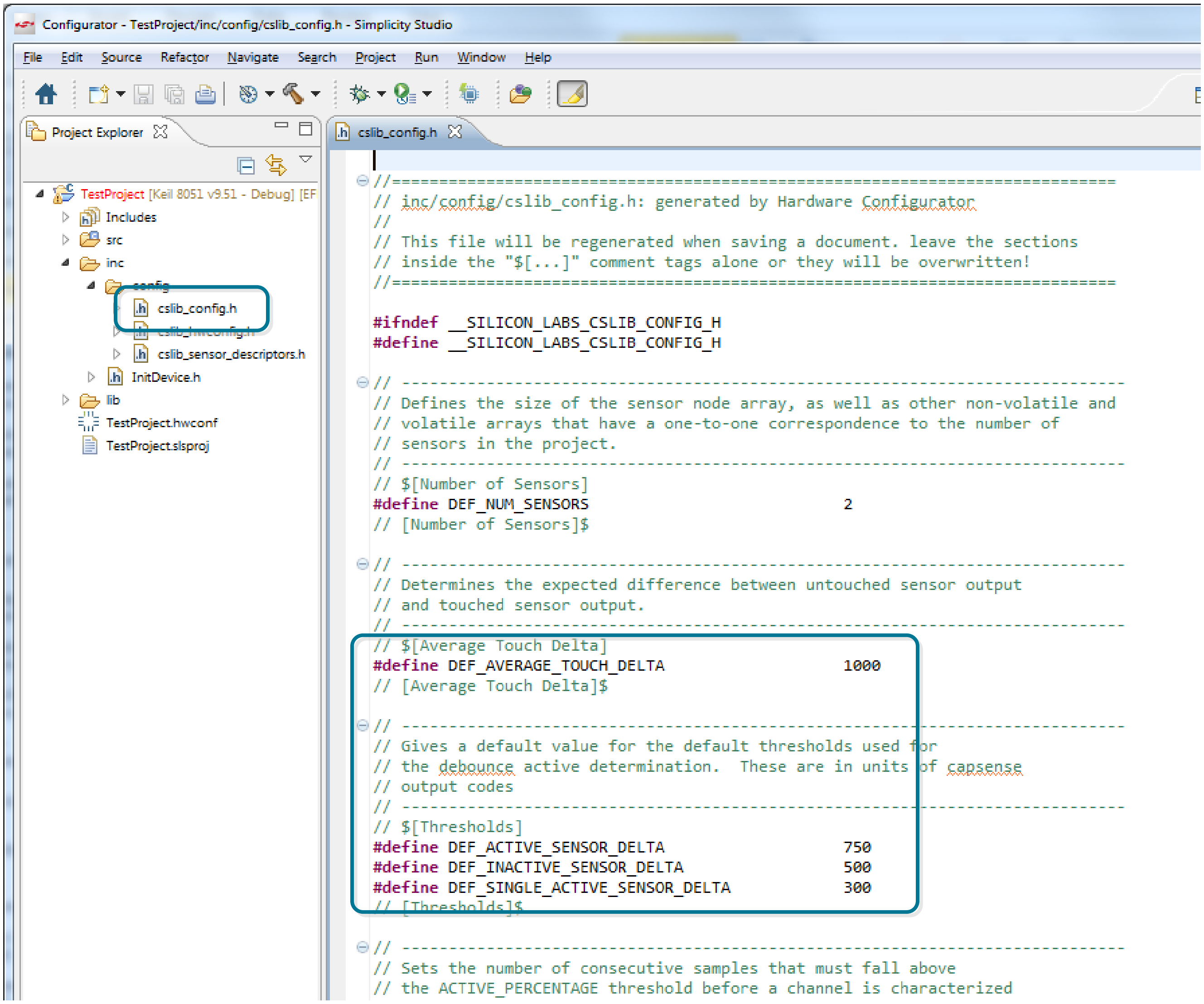


Figure 3.3. Opening `cslib_config.h`

In that file, find the definitions for threshold and configure them as follows:

```
#define DEF_ACTIVE_SENSOR_DELTA 750
#define DEF_INACTIVE_SENSOR_DELTA 500
```

Note that these are thresholds defined for sensors that are not using an overlay. In the next steps we will see how these thresholds behave when an overlay is present.

Step 2 — Build and Download Code

Build and download code to the connected device using the IDE's tools using the button highlighted in Figure 3.4 Downloading Code on page 8.

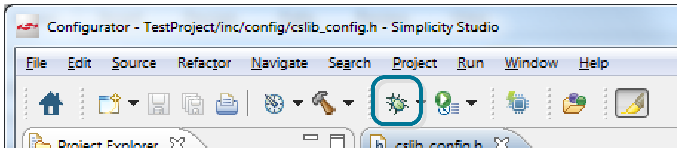


Figure 3.4. Downloading Code

Step 3 — Open Profiler

Open the [**Capacitive Sense Profiler**] by clicking on the button shown in Figure 3.5 Opening the Capacitive Sense Profiler on page 8.

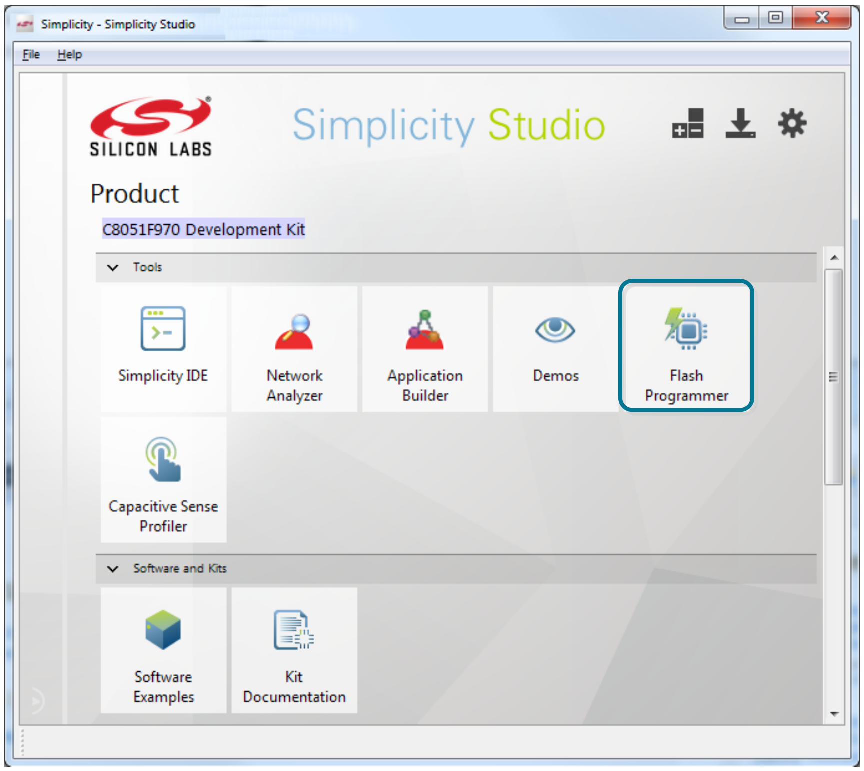


Figure 3.5. Opening the Capacitive Sense Profiler

Make sure that the PC is connected to the USB port on the board to establish a connection.

Step 4 — Connect to the Device

Click [**Use Device**] and follow the menus to choose the communications port that is connected to the device. The [**Use Device**] button is shown in Figure 3.6 Connecting to a Capacitive Sense Device on page 9.

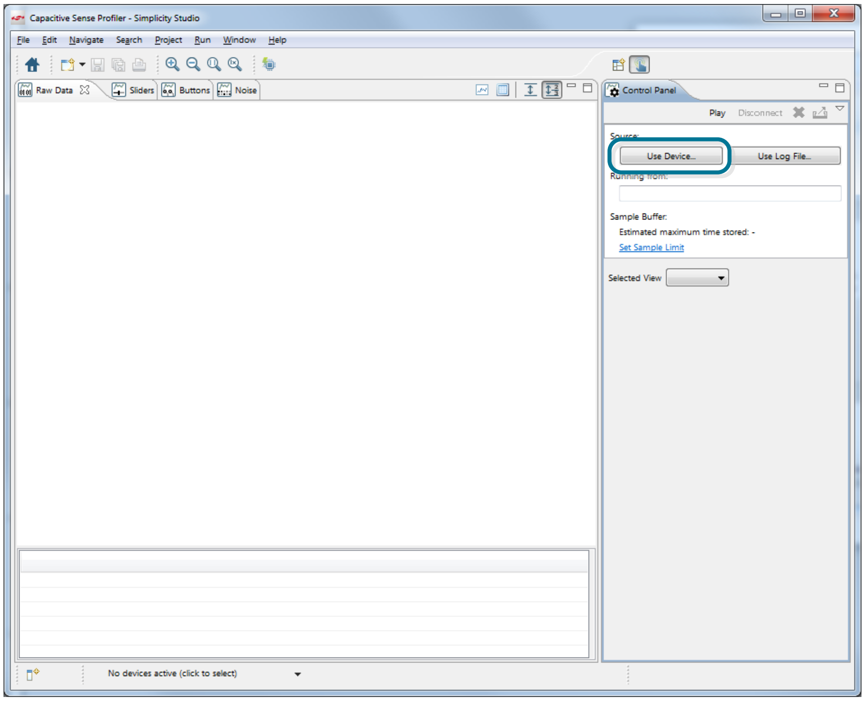


Figure 3.6. Connecting to a Capacitive Sense Device

If Profiler prompts to reset that device, press the reset switch on the board. After resetting, data should be displayed in the graphical view.

Step 5 — Filtering Displayed Data

By default, all channels of data are displayed by the Profiler. In order to get a better understanding of performance, it can be necessary to limit the amount of data displayed. In this case, we should look at a single channel of data, CS41, which is Channel 4. Select that channel by typing [**4**] into the [**Selected Channels**] text box as shown in Figure 3.7 Selecting a Channel on page 10.

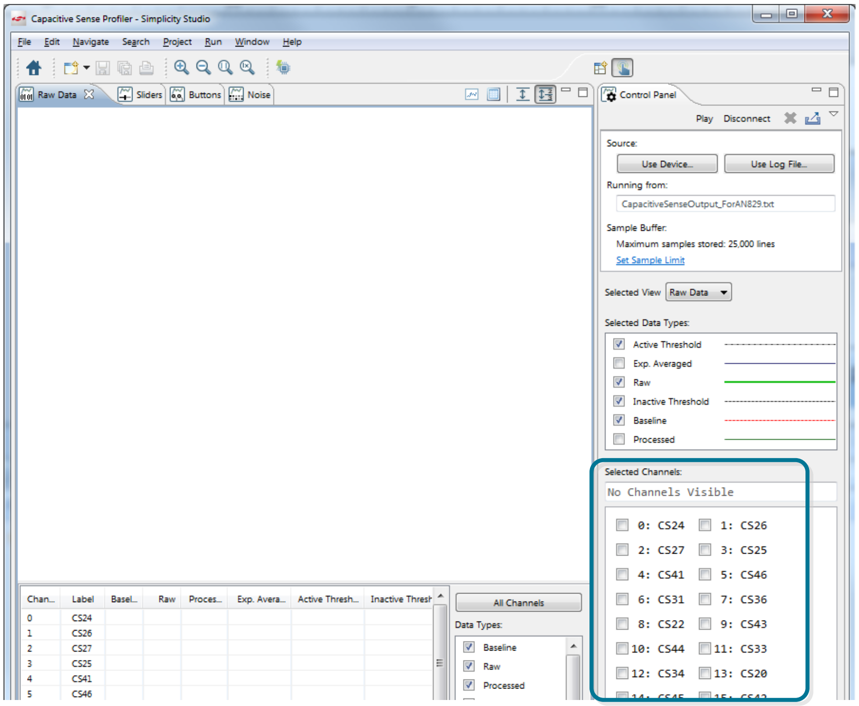


Figure 3.7. Selecting a Channel

By default, threshold levels are not displayed in the graphical view. Add thresholds by checking the [**Active Threshold**] and [**Inactive Threshold**] check boxes in the [**Selected Data Types**] view as shown in Figure 3.8 Configuring Shown Data on page 10.

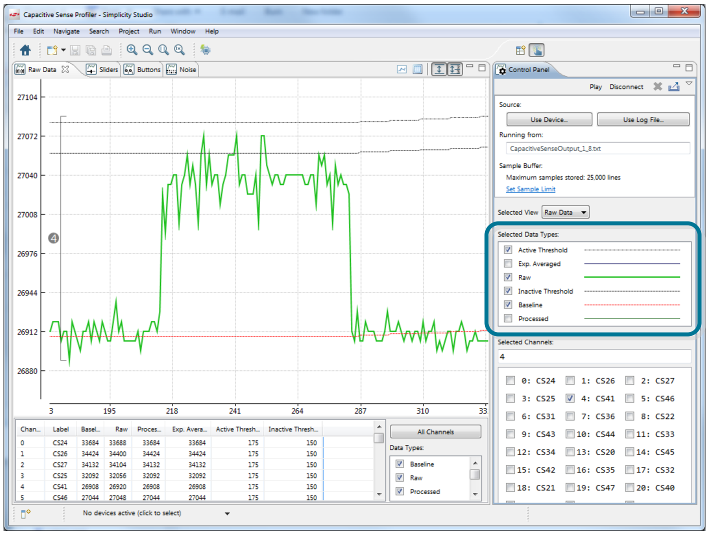


Figure 3.8. Configuring Shown Data

Note that the graphical view automatically resizes when new data is included or excluded from the view. This is because auto adjust is enabled for the graphical view. Auto-adjust can be enabled/disabled with the buttons shown in Figure 3.9 Configuring Auto-Adjust on page 11.

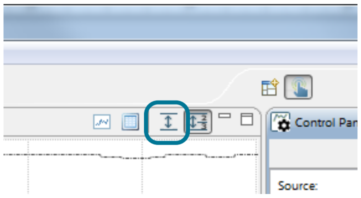


Figure 3.9. Configuring Auto-Adjust

Step 6 — Touch the Sensor

Touch the CS41 sensor (or any selected sensor) and watch as the green "raw" data line rises to show the touch delta in real time. Note that the raw data does not cross the active or inactive thresholds because those thresholds were configured for a system without an overlay. The touch delta and thresholds are shown in Figure 3.10 Touch Delta and Thresholds in Profiler on page 11.

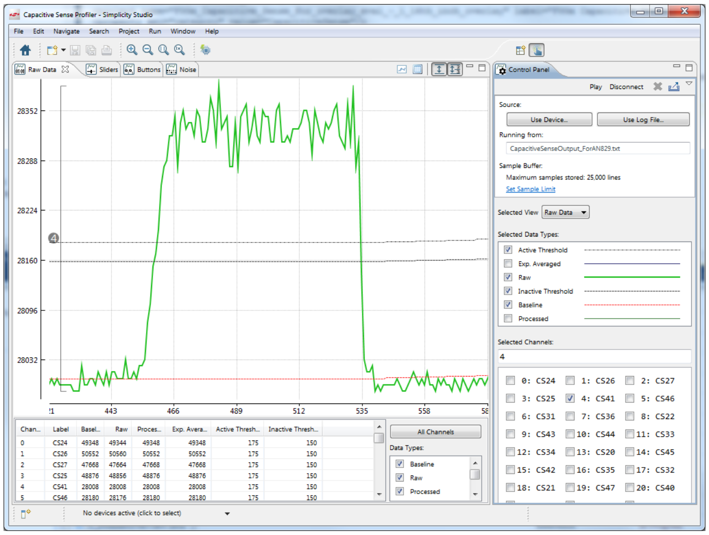


Figure 3.10. Touch Delta and Thresholds in Profiler

Step 7 — Adjusting the Thresholds in the IDE

Go back to the IDE and change the defined values for thresholds as follows, so that the thresholds are closer to the 350 code delta seen during testing, as shown in Figure 3.11 Editing `cslib_config.h` on page 12.

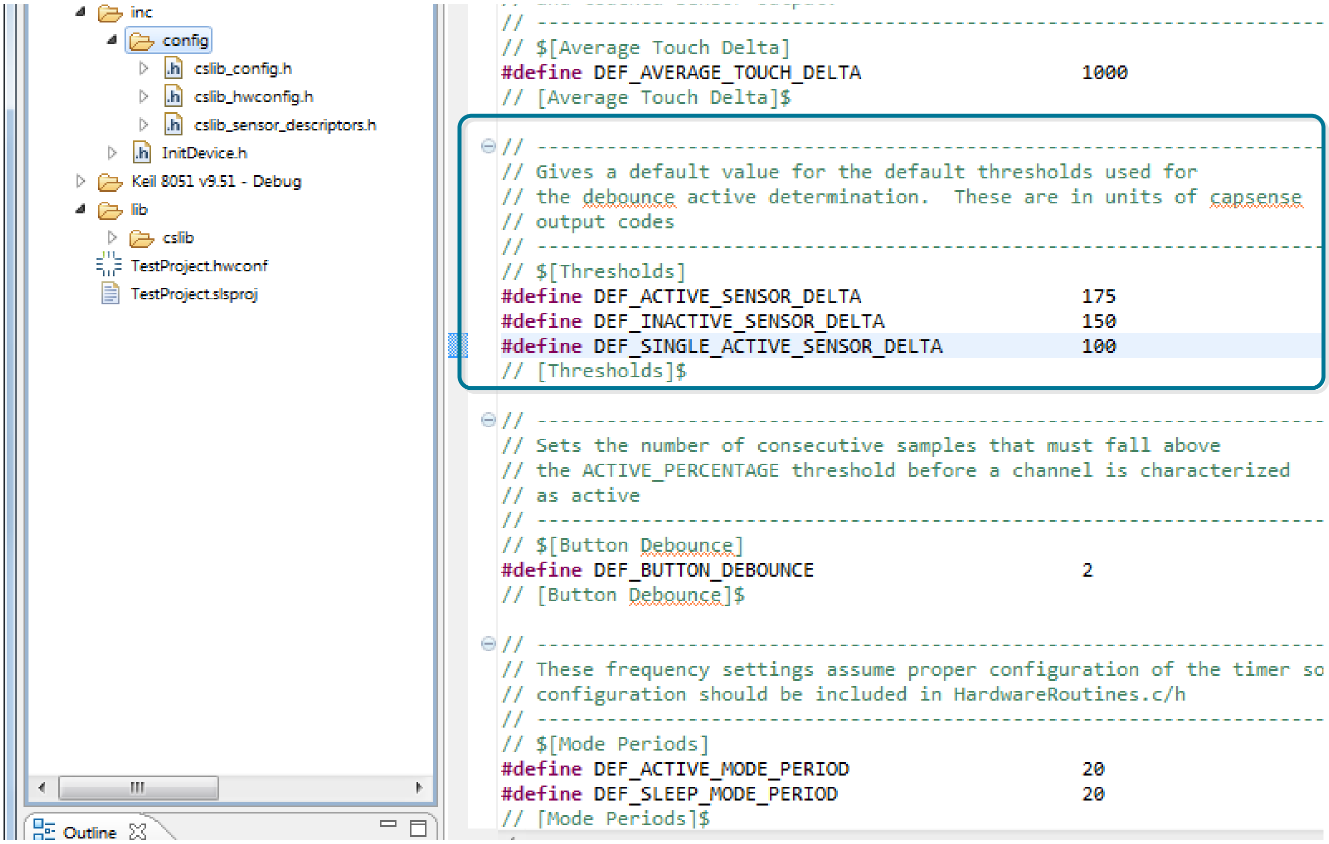


Figure 3.11. Editing `cslib_config.h`

```
#define DEF_ACTIVE_SENSOR_DELTA 175
#define DEF_INACTIVE_SENSOR_DELTA 150
```

Build and download an image as shown in the previous steps with the new configuration.

Step 8 — Going Back to Profiler to Retest

Reconnect with the MCU and touch the sensor under test again. Now the thresholds have been placed in a way that gives the thresholds more margin above baselines, which will yield a more robust sensor. Separating the inactive and active thresholds can help to further improve robustness by adding more hysteresis to the touch qualification. Figure 3.12 Testing the New Thresholds on page 12 shows a touch with the newly configured thresholds.

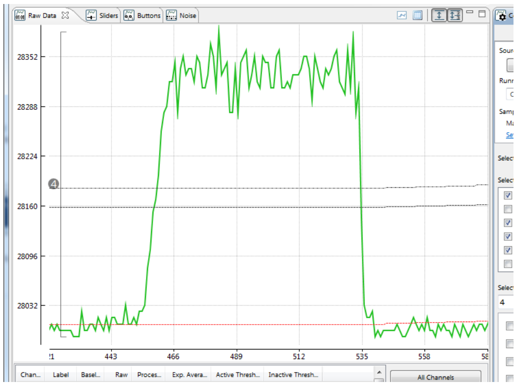


Figure 3.12. Testing the New Thresholds

4. Performance Configuration

The bulk of project configuration occurs in `ProjectConfig.c/h`. These files contain definitions and arrays that must be set to values that are project-specific.

AN0828: Capacitive Sensing Library Overview describes each definition in `ProjectConfig.h` and gives some recommended settings. Many of those definitions will not need to be changed from project to project. Table 4.1 `ProjectConfig.h` Configuration Notes on page 13 focuses on only those definitions that are likely to need to change.

Table 4.1. `ProjectConfig.h` Configuration Notes

Definition	Description and Recommendations
DEF_NUM_SENSORS	Must be set to the number of sensors in the project in order for the library functions to scan the sensors properly.
DEF_AVERAGE_TOUCH_DELTA	This is the expected difference in code output between a sensor in an untouched/inactive state and a sensor in a touched/active state. This value applies to all sensors on a project. The touch delta is a function of the sensor size and the thickness of the overlay. For a finger-sized sensor with a 1/8 inch overlay, a touch delta of ~150 codes is expected. For a 1/16 inch overlay, the delta could be as high as 300–400 codes. Tests can be run with the user monitoring data buffer values in the IDE through the SFR window to verify the touch delta seen on a particular board. Setting the value too high will result in sensors that are difficult to activate. Setting the value too low could result in sensors that are more susceptible to noise than is necessary.
DEF_ACTIVE_SENSOR_DELTA DEF_INACTIVE_SENSOR_DELTA	Set in units of capacitive sensing codes. These two values set the thresholds to detect an active transition on an inactive sensor and an inactive transition on an active sensor. See the section below for more information on how these values should be set.
DEF_SINGLE_ACTIVE_SENSOR_DELTA	This value determines single active sensor threshold, which is used to determine if the sensor is in “candidate active” state.
DEF_BUTTON_DEBOUNCE	Determines how many successive samples determine sensor state. More information can be found in the threshold discussion below.
DEF_ACTIVE_MODE_PERIOD DEF_SLEEP_MODE_PERIOD	These figures set the sampling/sleep-timing period in milliseconds in the system. Lower frequency means faster response time and higher average current consumption.
DEF_COUNTS_BEFORE_SLEEP	Sets the number of consecutive sample sequences in active mode that must pass before a system transitions from active mode to sleep mode. Higher numbers here mean that the system stays in a more responsive active mode for longer, at the expense of higher current draw.
DEF_FREE_RUN_SETTING	Controls whether active mode scanning occurs as fast as possible or only scans at DEF_ACTIVE_MODE_PERIOD. DEF_FREE_RUN_SETTING == 1 means more responsiveness at the cost of higher current draw.
DEF_SLEEP_MODE_ENABLE	Determines whether the system can enter sleep mode. Can be disabled if minimizing current consumption is not a priority.

4.1 Performance Trade-Offs

Most configuration decisions in the capacitive sensing project balance responsiveness and robustness with current consumption. Most broadly, the more a system is actively sampling, the faster the responsiveness and the higher the current draw. Table 4.2 on page 14 shows two different configurations, one with decisions made to optimize responsiveness, and other made to minimize current draw.

Table 4.2.

Definition	Optimal Responsiveness	Minimal Current Draw
DEF_ACTIVE_SENSOR_DELTA DEF_INACTIVE_SENSOR_DELTA	Set to 50% and 30% of the expected touch delta, respectively, to respond to touches faster. Deeper buffers and debouncing will guard against false positive touch events and make it "safe" to have lower active thresholds.	Set to 80% and 50% of the expected touch delta, respectively, to ensure that a strong touch is required before a touch is validated. Higher thresholds compensate for lower debounce settings to help guard against false positive touch events.
DEF_BUTTON_DEBOUNCE	8–10 counts, to require a consistent touch above active threshold in order to validate touch. Because the system is using DEF_FREE_RUN_SETTING == 1, sampling frequency is not bound by DEF_ACTIVE_MODE_PERIOD, and so debounce counts of 8–10 can be reached as quickly as hardware can take samples to be processed.	2–3 counts. Because DEF_FREE_RUN_SETTING == 0 in this configuration, touch validation occurs at a rate of DEF_BUTTON_DEBOUNCE / DEF_ACTIVE_MODE_PERIOD. Keeping debounce values low here ensures acceptable response time while still being mindful of current draw.
DEF_ACTIVE_MODE_PERIOD DEF_INACTIVE_MODE_PERIOD	Active = 10, Sleep = 100; Active only governs the rate at which sleep will be entered because DEF_FREE_RUN_SETTING = 1, sleep at 100 ensures a fast wakeon-touch response time.	Active = 20, sleep = 250. Active mode of 20 ensures acceptable response time in low noise environments, sleep of 250 ensures that average current consumption in sleep mode is low while still yielding acceptable response time.
DEF_COUNTS_BEFORE_SLEEP	20, MCU will enter sleep mode at 100 ms x 20 = 2 seconds.	5, MCU will enter sleep mode at 50 ms x 5 = 250 ms.
DEF_FREE_RUN_SETTING	1 ensures fastest response time when in active mode.	0 governs current draw even in active mode.

4.2 Touch Qualification Control

Firmware qualifies touches using a two-threshold system with debouncing to guard against false positives. The algorithm is described in detail in "AN0828: Capacitive Sensing Library Overview."

Figure 4.1 Button Debounce Functionality on page 15 shows a touch event detected using processed data with dejittering, along with a debounce setting of 3 and an active threshold that is 60% of the touch delta.

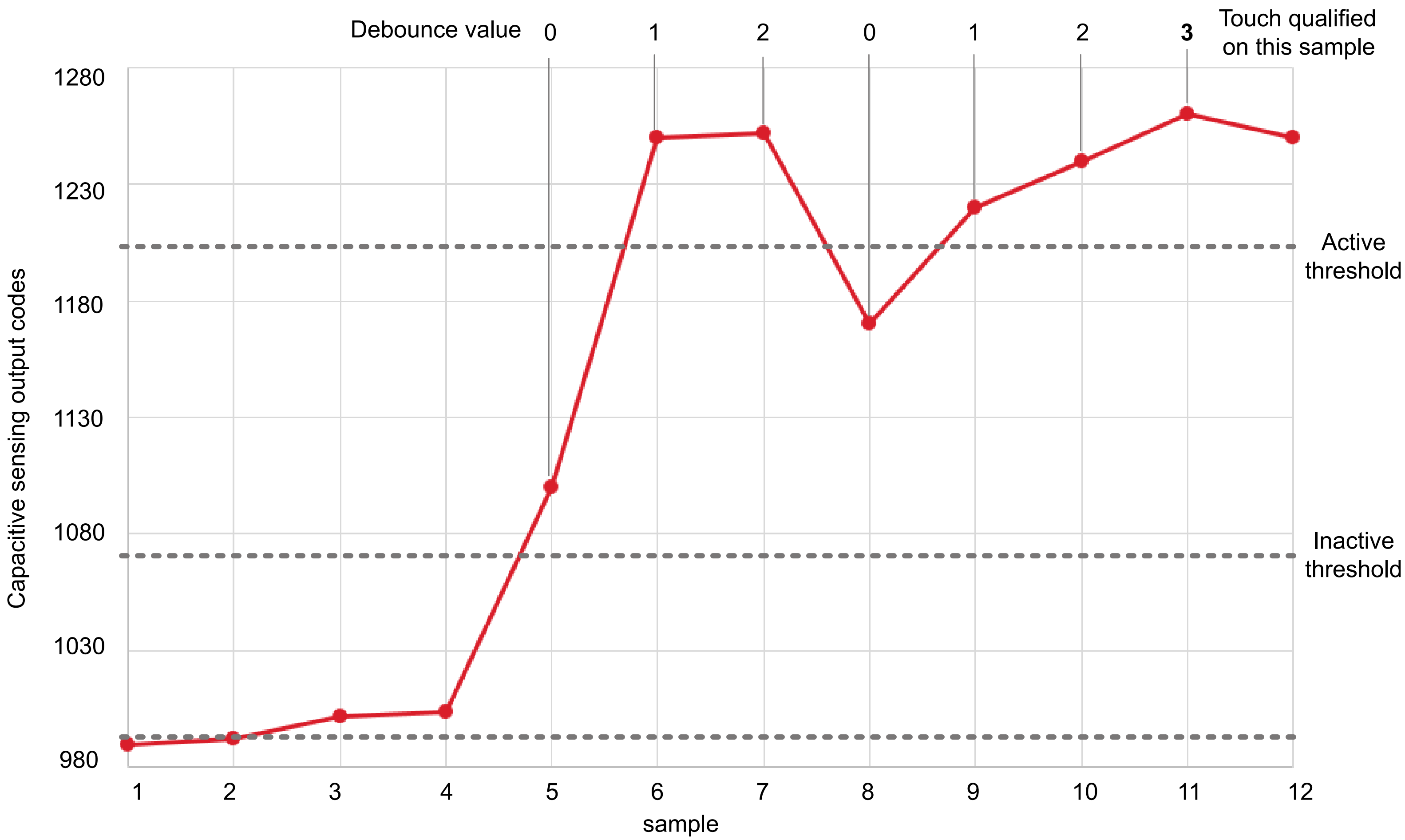


Figure 4.1. Button Debounce Functionality

The higher the DEF_BUTTON_DEBOUNCE setting, the higher the number of consecutive samples above the active threshold that will need to be processed before the button can be processed. Multiple factors govern the response time of a sensor, as shown in Table 4.3 Configuration Effects on Responsiveness on page 15.

Table 4.3. Configuration Effects on Responsiveness

Factor	Response Time Control
DEF_ACTIVE_SENSOR_DELTA	The closer the threshold is to the baseline, the faster the qualified touch.
DEF_BUTTON_DEBOUNCE	Lower value qualifies touch faster.
DEF_ACTIVE_MODE_PERIOD	In systems where FREE_RUN_SETTING == 0, higher sampling frequency means faster response time.
DEF_FREE_RUN_SETTING	Setting to 0 means fast-as-possible sampling, yields faster response time.
DEF_SLEEP_MODE_PERIOD	Higher sleep mode frequency means faster wake-on-touch response time.

5. Hardware Configuration

The capacitive sensing library includes algorithms for sensing and processing samples, but it relies on project source code to configure the capacitive sensing peripheral and port pins for scanning. The following sections cover the configuration of port pins and the peripheral using the source code in the Simplicity Studio examples when referring to file names and functions. The library uses callback functions to configure and access hardware, and some of these functions are library callbacks. For more information on callback functions, see "AN0828: Capacitive Sensing Library Overview."

5.1 Configuring the Library to Use Pins and the Sensing Peripheral

The capacitive sensing library operates in both active and sleep modes. The library allows systems to configure port pins for those two modes, enabling the system to configure pins that need to be sampled in active mode and sleep mode independently.

Configuration of pins in active mode happens in the `hardware_routines.c`'s function, called `configurePortPinsActiveMode()`. Any capacitive sensing port pins used by the library in active mode should be configured here. Configuration of pins in sleep mode happens in the `low_power_config.c`'s function, called `configurePortsSleepMode()`. Pin configuration differs from device to device. See the example below for an illustration of how pins can be configured on the C8051F970.

In addition to low-level hardware configuration, the library needs to be bound to the configured pins. Additionally, the library configures the capacitive sensing peripheral independently for each capacitive sensing input. Active mode and sleep mode port configuration is contained within `cslib_hwconfig.h` .

5.2 Example Hardware Configuration

In this example, we'll configure the library to read two port pins (P2.0 and P3.7) to be capacitive sensing inputs on the C8051F970. The pins will be used in both active mode and sleep mode. To enable these pins as capacitive sensing inputs, open cslib_hwconfig.h and configure the port masks as follows. Note that the only bits set are those which correspond to port pins P2.0 and P3.7.

```
#define ACTIVE_MODE_MASK_P0             0x00
#define ACTIVE_MODE_MASK_P1             0x00
#define ACTIVE_MODE_MASK_P2             0x01
#define ACTIVE_MODE_MASK_P3             0x80
#define ACTIVE_MODE_MASK_P4             0x00
#define ACTIVE_MODE_MASK_P5             0x00

#define SLEEP_MODE_MASK_P0              0x00
#define SLEEP_MODE_MASK_P1              0x00
#define SLEEP_MODE_MASK_P2              0x01
#define SLEEP_MODE_MASK_P3              0x80
#define SLEEP_MODE_MASK_P4              0x00
#define SLEEP_MODE_MASK_P5              0x00
```

After configuring the port pins, we'll configure the mux values, gain values, and accumulation settings for each enabled bit. In this example, we have two enabled capacitive sensing inputs, and so each configurable parameter will need two entries. To configure the capacitive sensing scan engine to scan capacitive sense input mux values 16 and 23, using a gain value of 7 and an auto-accumulation value of 4 for both inputs, the definitions in `cslib_hwconfig.h`look as follows:

```
#define MUX_VALUE_ARRAY 16, 23
#define GAIN_VALUE_ARRAY 0x07, 0x07
#define ACCUMULATION_VALUE_ARRAY 4, 4
```

6. Slider Module

Firmware includes an optional library and associated source files that can be included and used in coordination with the capacitive sensing library to track touches on a contiguous chain of sensors, commonly called a slider. The sections below describe the functionality and configuration of the slider library.

6.1 Functional Overview

The slider algorithm is used to interpolate a conductive object's position among one or more sensors laid out in an array, with one sensor directly against a successive sensor. The algorithm assigns a bin value to each sensor of which the slider is composed. For every slider sensor with a touch delta above that sensor's baseline value, that delta is multiplied by the sensor's corresponding bin value, and that value is accumulated with all other delta x bin values. That resulting summation is then divided by the sum of all touch deltas. The resulting value is a weighted average, centroid output with values bound by the highest and lowest bin values. Figure 6.1 Slider Centroid Calculation and Processing Example on page 17 shows an example centroid calculation.

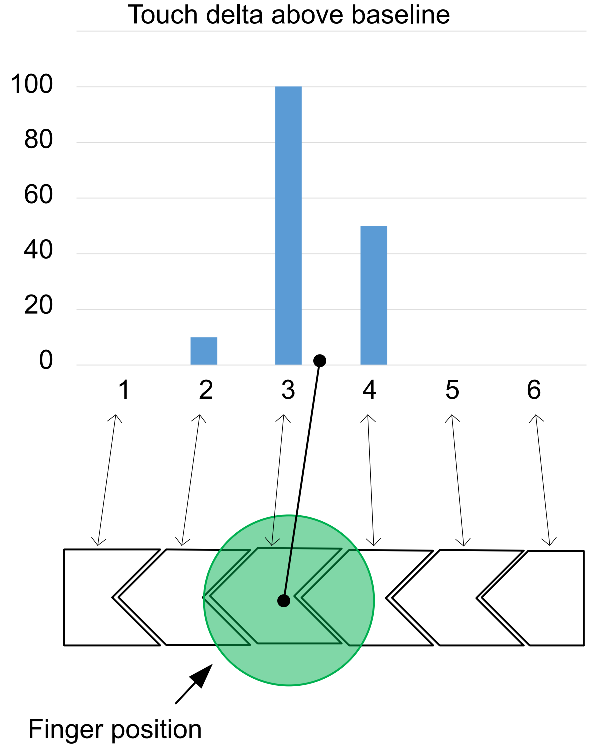


Figure 6.1. Slider Centroid Calculation and Processing Example

6.2 Configuring the Slider Module

The slider module uses data collected in the sensor node struct as inputs that generate a centroid output. The slider library requires that three source files be included in the build. The table below describes those files.

The slider data structure requires that pointers to arrays with slider configuration be assigned at startup, before values can be derived from the slider sensors. The structure is defined in `SliderLibrary.h` and the arrays are assigned in `SliderConfig.c`. Please see the section that follows for more information.

Table 6.1. Slider File Overview

File	Description
`SliderConfig.h`	Includes a definition specifying the number of sliders in a project. This file can also hold optional #define statements used in `SliderConfig.c` as defined by the user.
`SliderConfig.c`	This file contains definitions for the arrays of configuration information that must be assigned to the sliders. The file also includes an initialization routine that calls into the library to assign those pointers.
`SliderLibrary.h`	Includes the definition of the slider data structure and some APIs for accessing slider data.

Configuring `SliderConfig.c`

Each defined slider must have the following values defined for it:

- Indexes of the sensors bound to the slider, defined in an order that matches physical layout.
- The bin values for each sensor, as shown in Figure 6.1 Slider Centroid Calculation and Processing Example on page 17.
- The number of sensors defined for the slider.

The index list and the bin values list should both be defined as arrays of unsigned characters. The number of sensors can be an unsigned character. To assign an array of bin values to a slider, call AssignBinArray() with the slider index and the pointer as parameters. To assign an array of sensors to the slider, call AssignSliderChannelArray() with the slider index and the pointer as parameters. To assign a number of sliders, call AssignSliderLength() with the slider index and the number of sensors in the slider.

Example Slider Configuration

In this example, we'll configure a four-sensor slider to use sensors 0, 1, 2, and 3 with bin values of 10, 20, 30, and 40 respectively.

In `SliderConfig.c`, the sensor array could be defined as:

```
SI_SEGMENT_VARIABLE (slider0mux[SLIDER0_NUMCHANNELS], uint8_t, SI_SEG_XDATA) =
{
3,
2,
1,
0
};
```

The bin array can be defined as:

```
SI_SEGMENT_VARIABLE (slider0binvalues[SLIDER0_NUMCHANNELS], uint8_t, SI_SEG_XDATA) =
{
10,
20,
30,
40
};
```

The InitSliders() routine in `SliderConfig.c` could then be defined as follows:

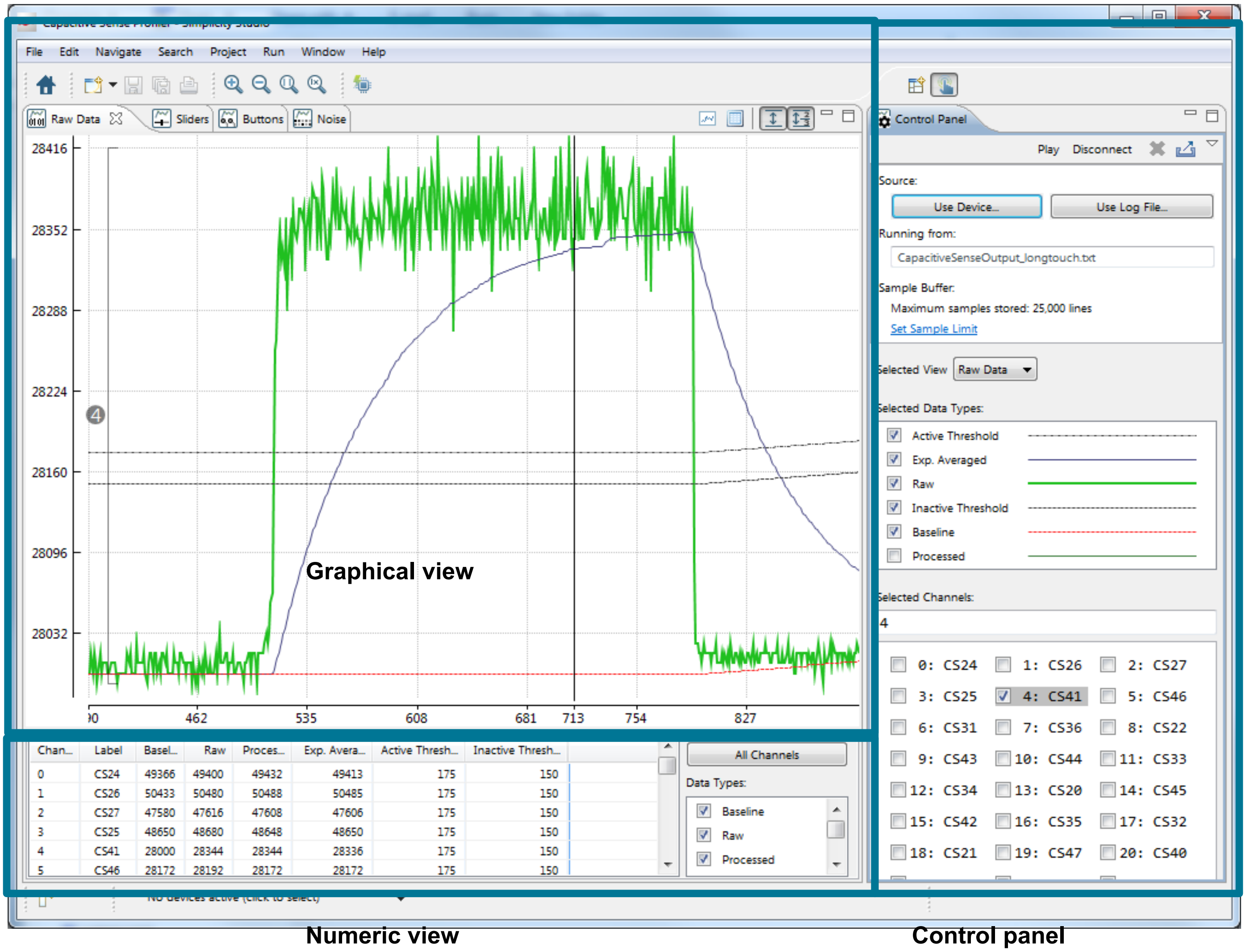
```
void InitSliders(void)
{
    ResetSliderStruct();
    AssignSliderChannelArray(0, slider0mux);
    AssignBinArray(0, slider0binvalues);
```

```
    AssignSliderLength(0, SLIDER0_NUMCHANNELS);
}
```

7. Profiler Functional Overview

The Profiler is broken into three main on-screen components: Graphical View, Numeric View, and the Control Panel.

Figure 7.1. Profiler Components



7.1 Using the Graphical View

The graphical view displays different types of data in real time. The view offers four different perspectives on data streaming from the capacitive sensing device. Data shown in each perspective can be customized in the control panel view.

- Raw data—This perspective provides the most comprehensive look at data, including capacitive sensing data directly from the device and algorithmically derived data processed after sampling. This view can be used to tune thresholds and examine behavior of the device.
- Slider—If the capacitive sensing device includes the slider library, the slider view shows the centroid output of the algorithm for a slider. In this view, the user can also see the sensor data used to derive centroid output in the numeric view.
- Buttons—This view focuses on touch qualification, showing output in a logic analyzer-like display where the digital state of buttons is shown with square waves. This view is useful if the user wants to do top-level testing of button functionality without examining data in the raw data view.
- Noise—This view displays the sensor-specific interference estimation, as well as the global interference characteristic used by the algorithms to determine how to most reliably qualify touch events. This view is useful when performing interference tests on a system.

7.2 Control Panel

Data can be selectively included or removed from the graphical view using the control panel. The control panel is divided up into three sections.

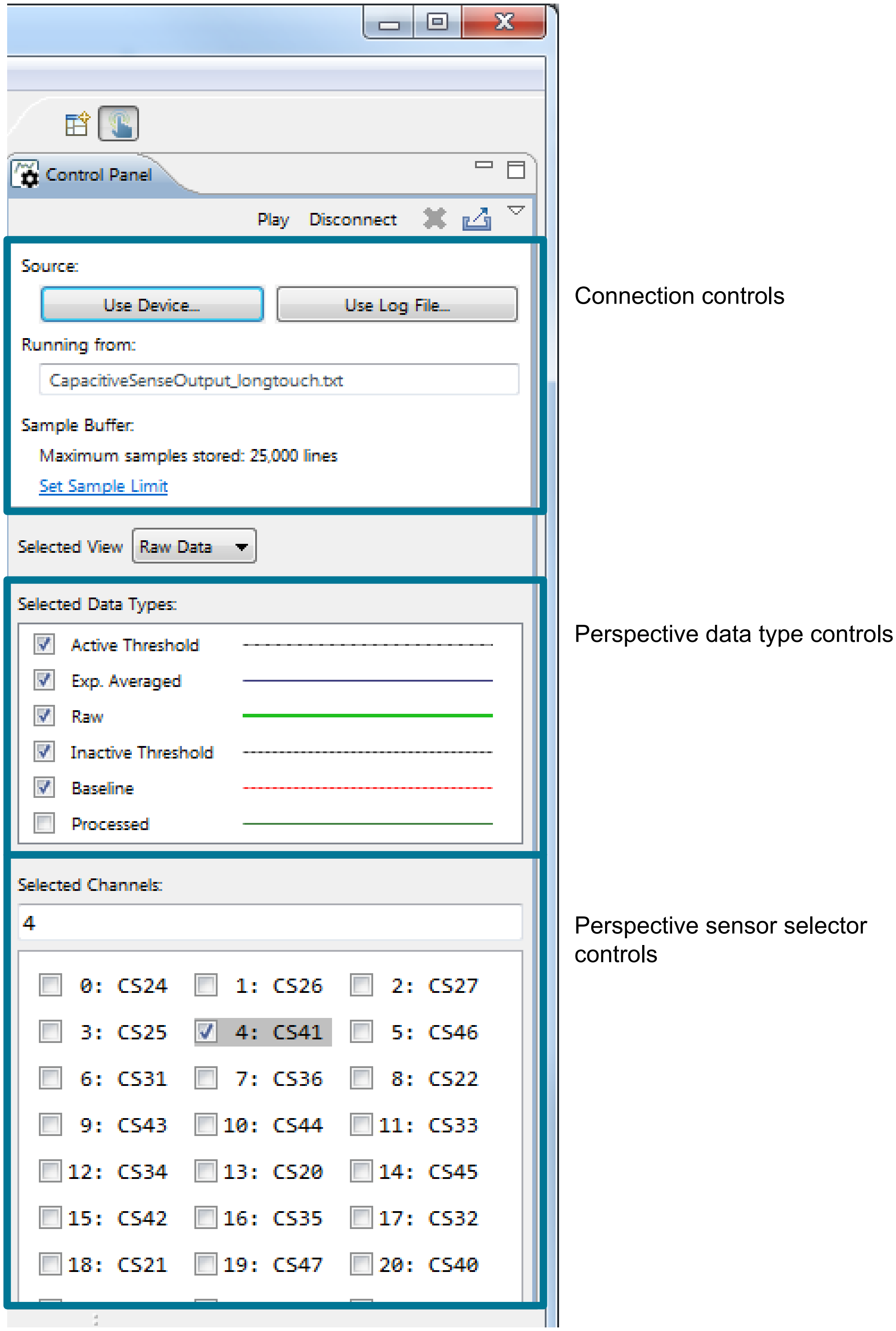


Figure 7.2. Control Panel Components

- Connection Controls—These controls configure the Profiler to either read new data from a device, or play back logged data. The [**Use log file**] button configures the tool to use a previously saved log file, while [**Use device**] connects to a device through a communications port.
- Perspective Data Type Controls—These graphical view-specific settings enable/disable visualization of data types relevant to the selected graphical view perspective. Removing data types from a perspective can help to clear away information that is not immediately relevant to a step in development and clarify the information displayed.

- Perspective Channel Selector Controls—These controls enable/disable data instances which have data types listed in the perspective data type controls. Limiting the number of channels displayed is useful when a user needs to examine behavior on one region of a board for certain tests. In systems with many channels, it can be useful to limit the view to a subset for added clarity.

7.3 Numeric View

The numeric view expresses newest data shown in the graphical view. The graphical view is useful because it provides a historical perspective and a top-level view of how the data is behaving. The numeric view only displays a single cycle of updated data, but allows the user to see at a glance the precise numeric values of data received.

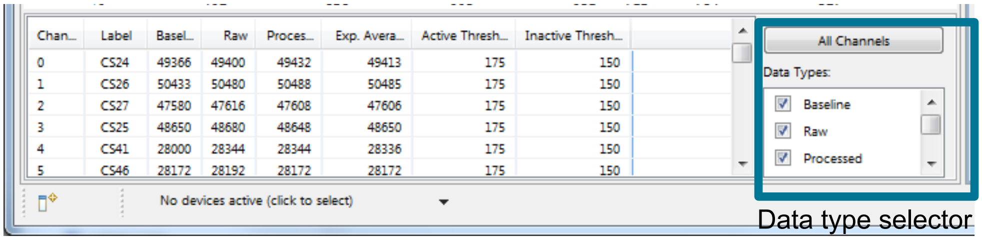


Figure 7.3. Numeric View Control

Display of the data types in the numeric view can be enabled/disabled independently from the data types in the graphical view using the data type selector list.

7.4 Logging Data

Users can save data received from a device to a log file using the data export tool. This tool exports the data stream to a log file without modification or filtering. These logs can be imported and reviewed by clicking [**Use Log File**] and importing a previously saved file.

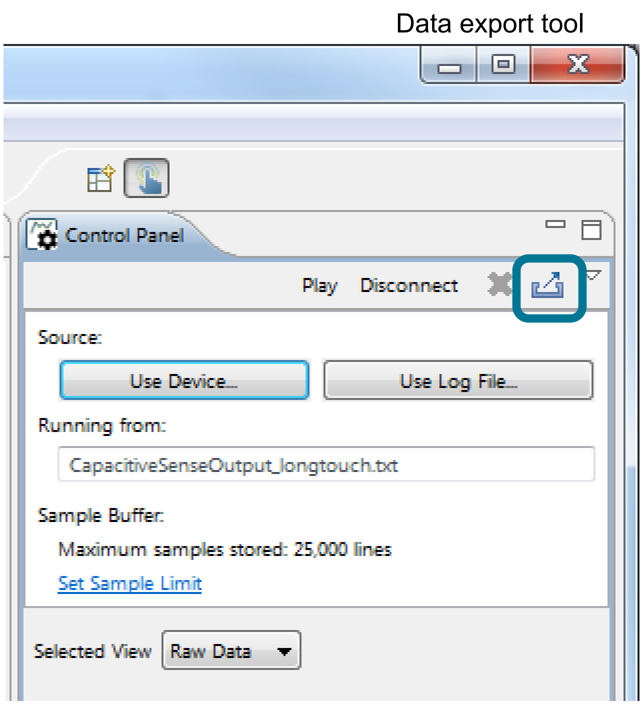


Figure 7.4. Exporting Logged Data

Reviewing log files can be useful during development for a few reasons:

- Comparing different library performance configurations.
- Saving product test results.
- Capturing behavior for team review.

7.5 Data in Real Time vs. Playback

Both logged and real-time data can be viewed as a scrolling stream. For logged data, pressing the [**play**] button will begin playing the stream until either the [**pause**] button is pressed or the Profiler reaches the end of the log file. As data plays, the numeric display updates with the latest received/retrieved data.

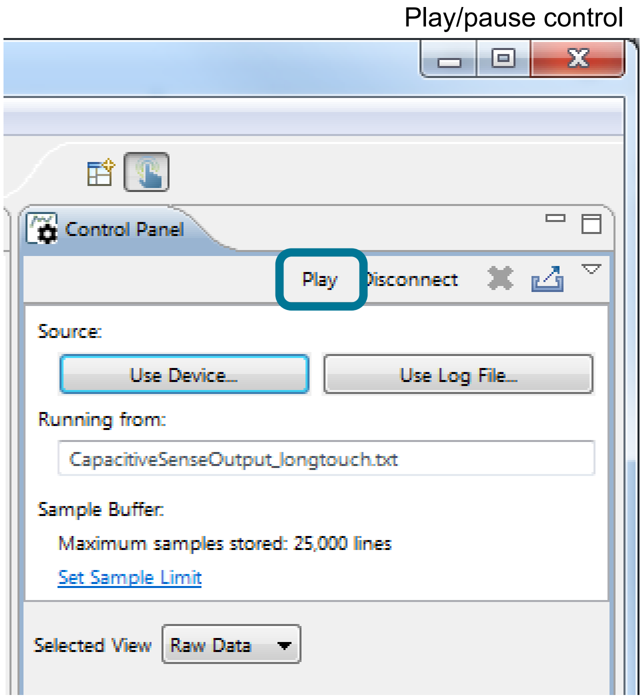


Figure 7.5. Playing or Pausing the Data

When the [**pause**] button is pressed, data can be reviewed in a number of ways. If the user clicks on the graphical view, the view can be scrolled backward and forward in time by clicking the left/right arrow keys on the keyboard, or by clicking and holding the center button on the mouse and then moving the mouse to the left or right.

When playback is paused, the user can click on the graphical view to place a cursor on data at that point on the view's X axis. After the click, the numeric view updates with the data at that point in time on the x-axis.

Playback is useful to see real-time responsiveness and algorithmic performance. The controls enabled when playback is paused offer tools for customers to examine data more carefully.

7.6 Display Controls

The graphical view offers a few display tools to help customers adjust the view to include the amount of data that's needed, ordered in a way that is most useful to them.

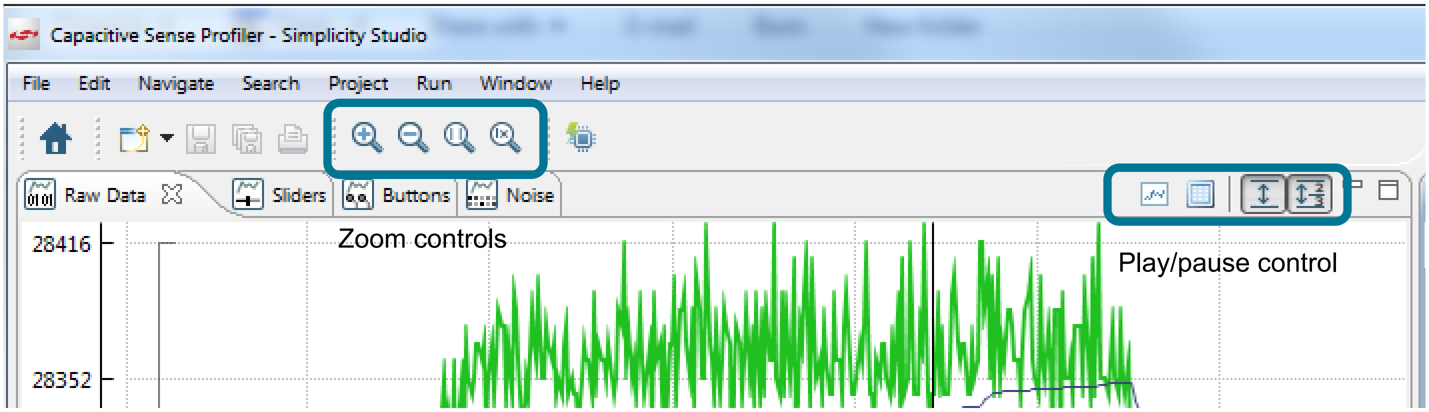


Figure 7.6. Display Controls

The zoom controls enable the user to increase or decrease the x-axis grid spacing. Seeing a wider span of data is useful when examining performance over multiple seconds because it preserves a historical perspective on samples in the recent past. Zooming in is more useful when playback is paused and the user wants to closely examine behavior at an isolated point in the data stream.

The data scaling/organization controls change the way channels are organized in the graphical display.

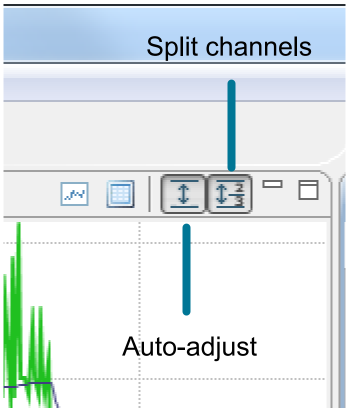


Figure 7.7. Data Scaling and Oranization Controls

Enabling data scaling enables the Profiler to dynamically size the Y axis grid spacing and min/max points so that the view is only wide enough to include all selected data in the view. Disabling this view forces the Y axis to show the full range of data points, starting at the axis origin.

Enabling split channels splits the Y axis into separate y-axes, with one axis for each selected channel. This is useful when the user is attempting to simultaneously examine multiple channels, because the Profiler will ensure that data from one channel do not overlap with data from another channel. However, because the y-axis is split, data is no longer all mapped to a single axis, and information about the relationships between channels is obscured.

It is useful to enable both data scaling and split channels in some applications. When both features are enabled, each y-axis will be scaled dynamically so that data from each channel is always displayed.

7.7 Mining Data in the Graphical View

The raw data perspective in the graphical view includes some features to help users easily examine system behavior.

The algorithm qualifies button presses using the debounce toggle data type. This data type is used in the raw perspective to show when a touch qualification has occurred using horizontally oriented brackets.

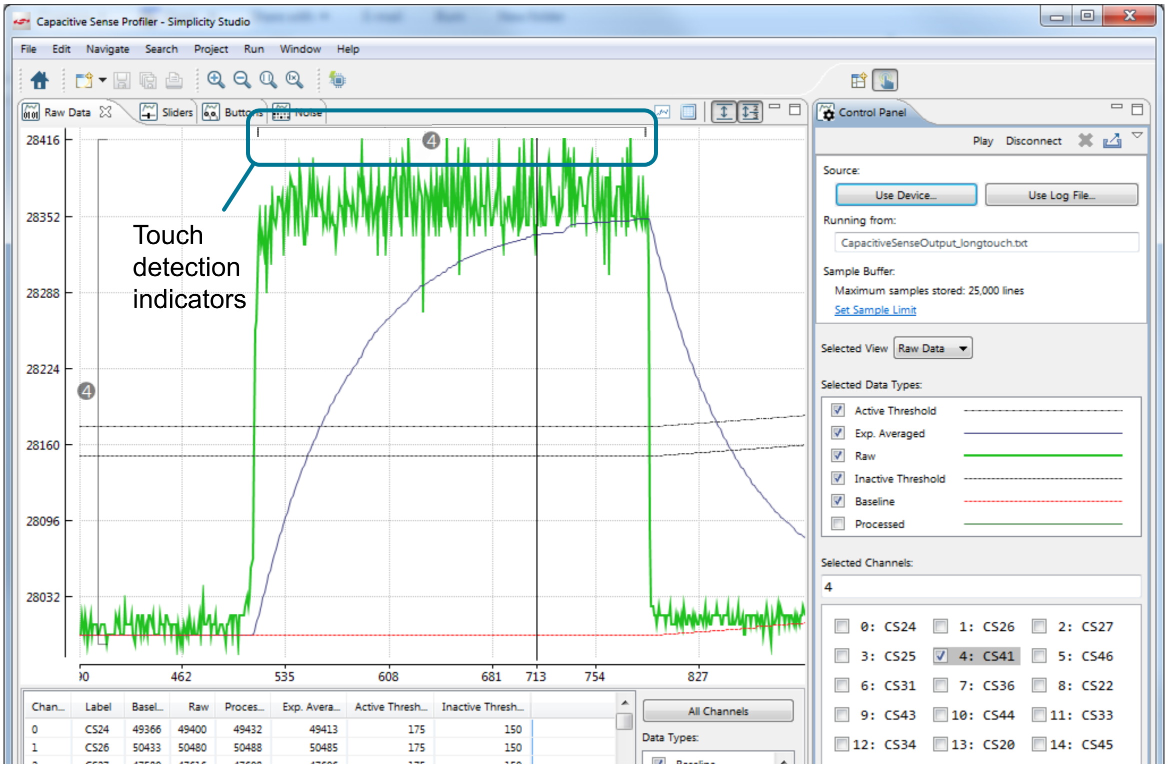


Figure 7.8. Button Detection Indicators

Vertically oriented brackets placed on the left side of each y-axis, with one bracket for each channel, show the latest minimum and maximum value of a sensor. Hovering the mouse over this bracket shows the touch delta numerically.

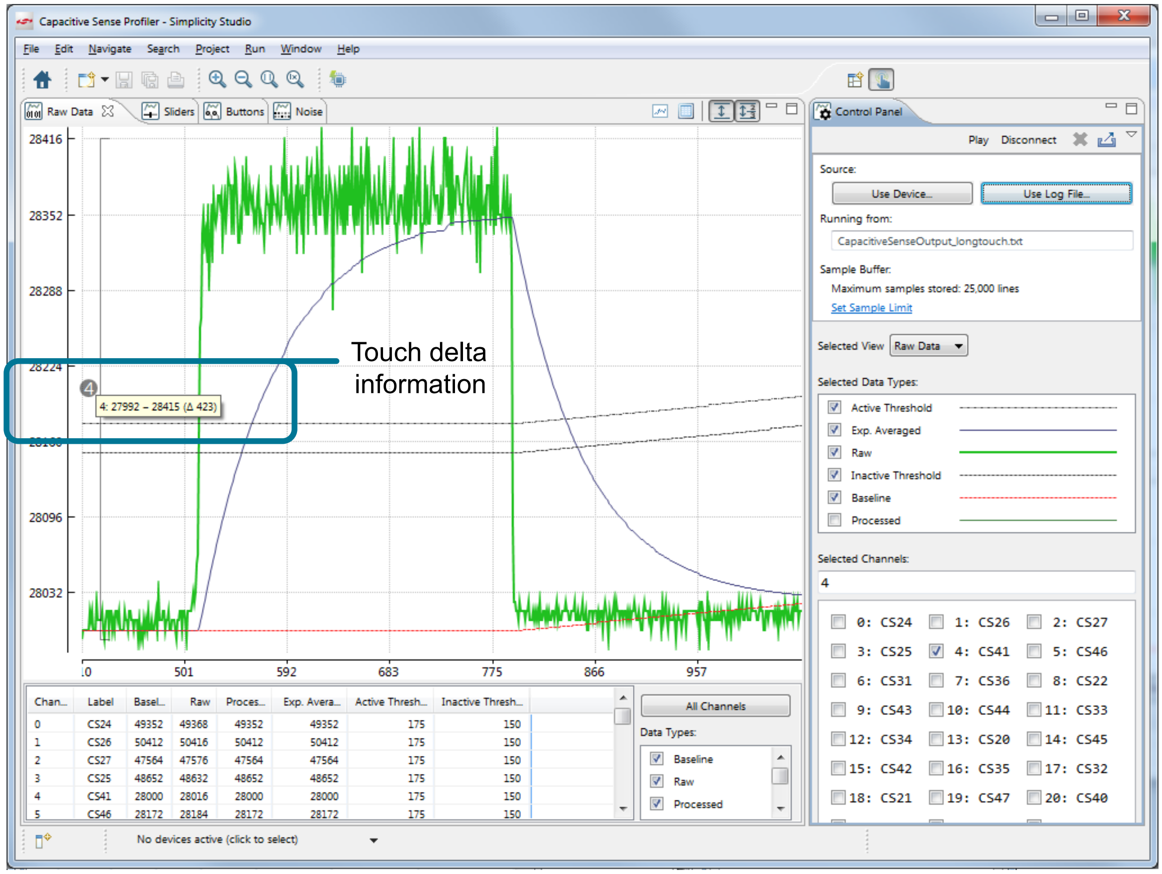


Figure 7.9. Touch Delta Bracket with Hover-Over Information

The Profiler includes a tool to derive SNR and standard deviation on a data selection. To use this feature, playback must be paused. The user should highlight a selection of data and then over the mouse over the selection. The Profiler will derive values if the data selected shows a touch and includes at least 50 samples of untouched and touched data.

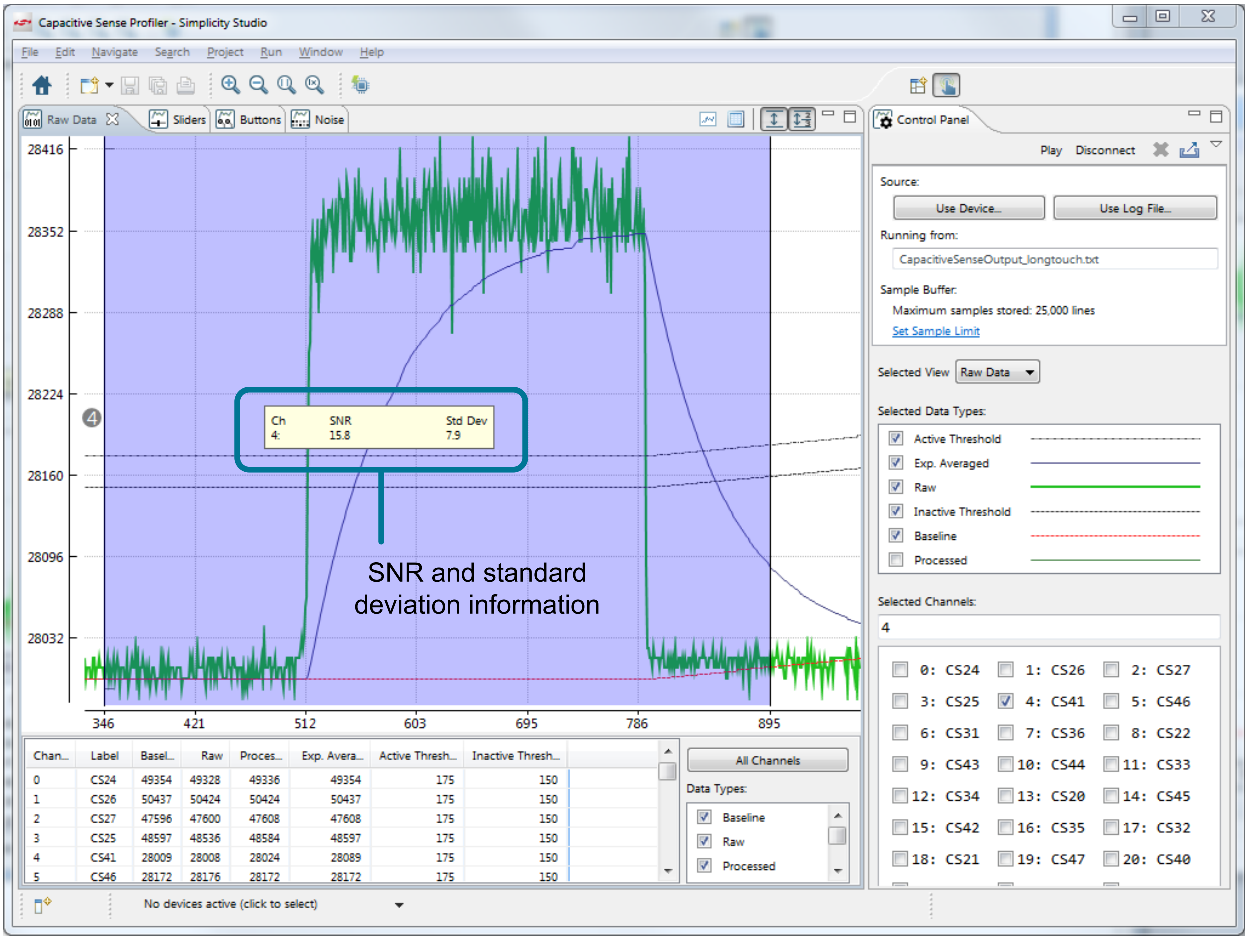


Figure 7.10. SNR and Standard Deviation Derivation

8. Revision History

Revision 0.3

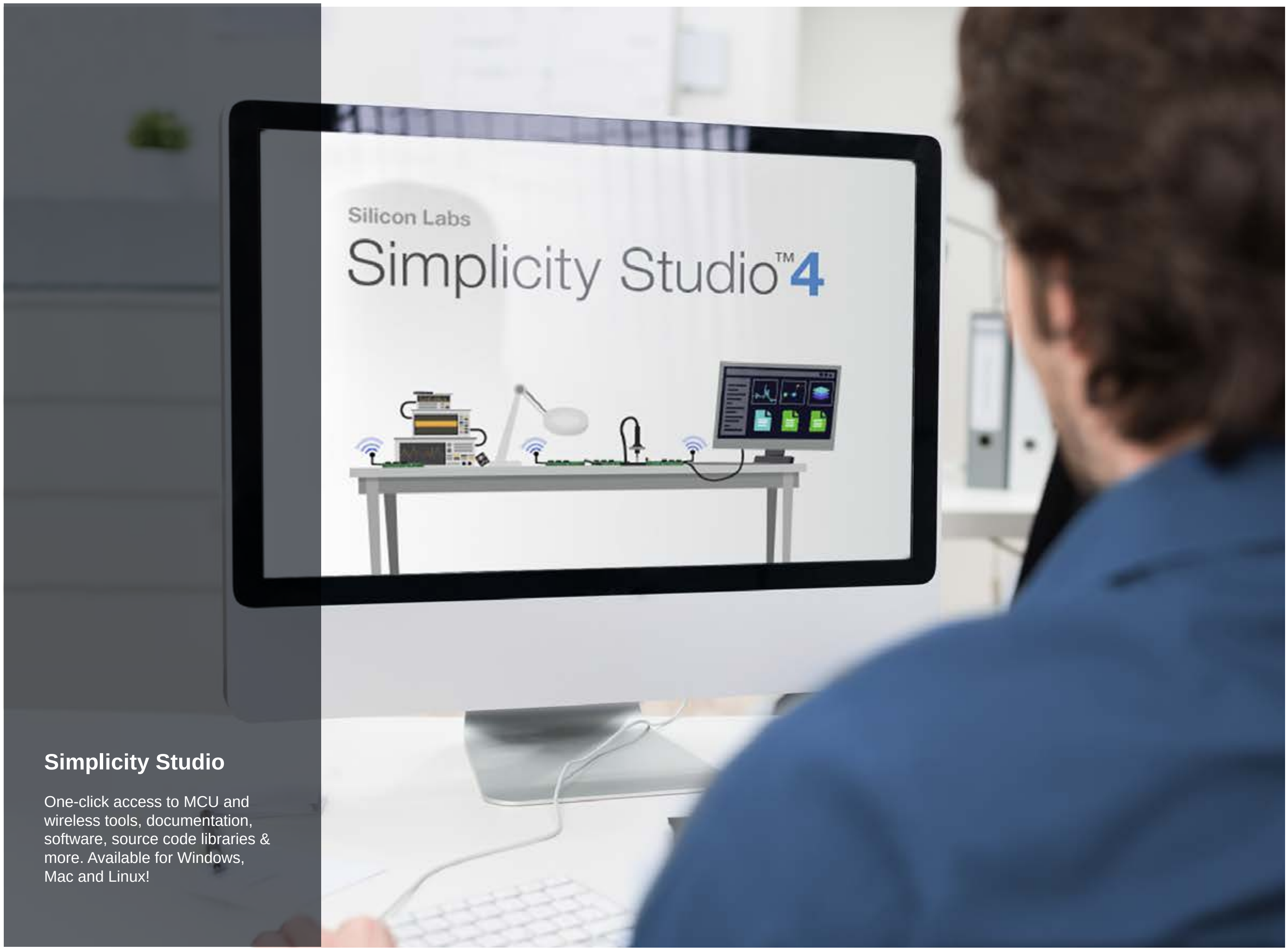
September, 2019

- Replaced code references to compiler_defs.h macros with their equivalent si_toolchain.h macros.

Revision 0.2

2017

- Initial Release



IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com