AN12709

NXP Touch Development Guide

Rev. 1 — 07 December 2021 Application Note

Contents

1 NXP Touch

NXP® Touch software is designed to speed development of your touch applications and is ideal for use with Kinetis®MCUs. Available in source code, this software download features touch detection algorithms and be ideally suited for RTOS-based applications. NXP Touch software employs a modular architecture with various touch centric controls, modules, and electrode data objects, enabling integrated and customizable features.

NXP Touch targets to help customers do a smooth migration and easy designing in touch applications.

It is a total solution for touch control applications, and NXP Touch offers touch software and hardware touch sensing IP TSI.

SW library in combination with the Kinetis KE1x platform brings lots of improvements listed below:

- Great EMC performance, noise immunity, pass the IEC61000-4-6 standard test both 3 V/10 V test.
- Support both Self-cap and mutual-cap mode, up to 6x6 matrix touch pads.
- Great performance in liquid tolerance, water, oil, cold steam, and so on.
- High configurable sensitivity, supports up to 10 mm thick glass/plastic overlay.

2 Touch sensing HW support

NXP offers the latest TSI v5 HW peripheral available on Kinetis KE1x devices based on charge-transfer physical principle. This method provides appropriate performance regarding the sensitivity and immunity against the environmental changes and EMC.

Touch sensing demo SW example is targeted for FRDM-KE15z with inserted FRDM-TOUCH module, which is not included together with the FRDM board and must be ordered separately.

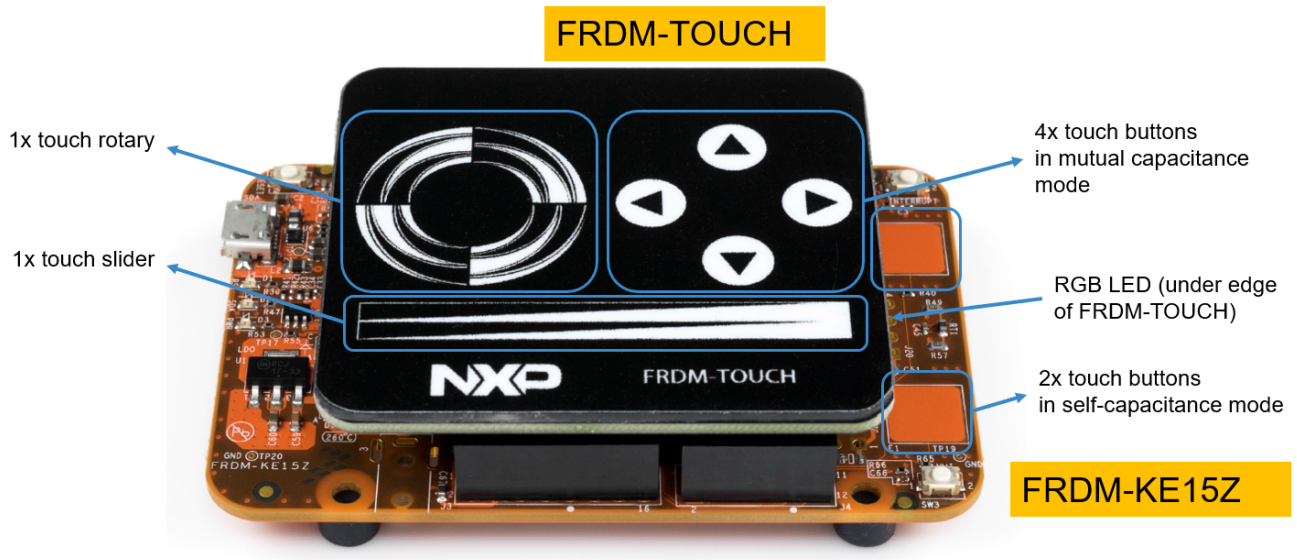


Figure 1. FRDM-KE15Z and FRDM-TOUCH

3 NT SW Library

NXP offers a touch software library, free of charge. It provides all the software required to detect touches and to implement more advanced controllers like sliders or keypads.

TSI background algorithms are available for touch keypad and analog decoders, sensitivity auto calibration, low power, proximity, water tolerance. SW distributed in source code form in “object C language code structure”

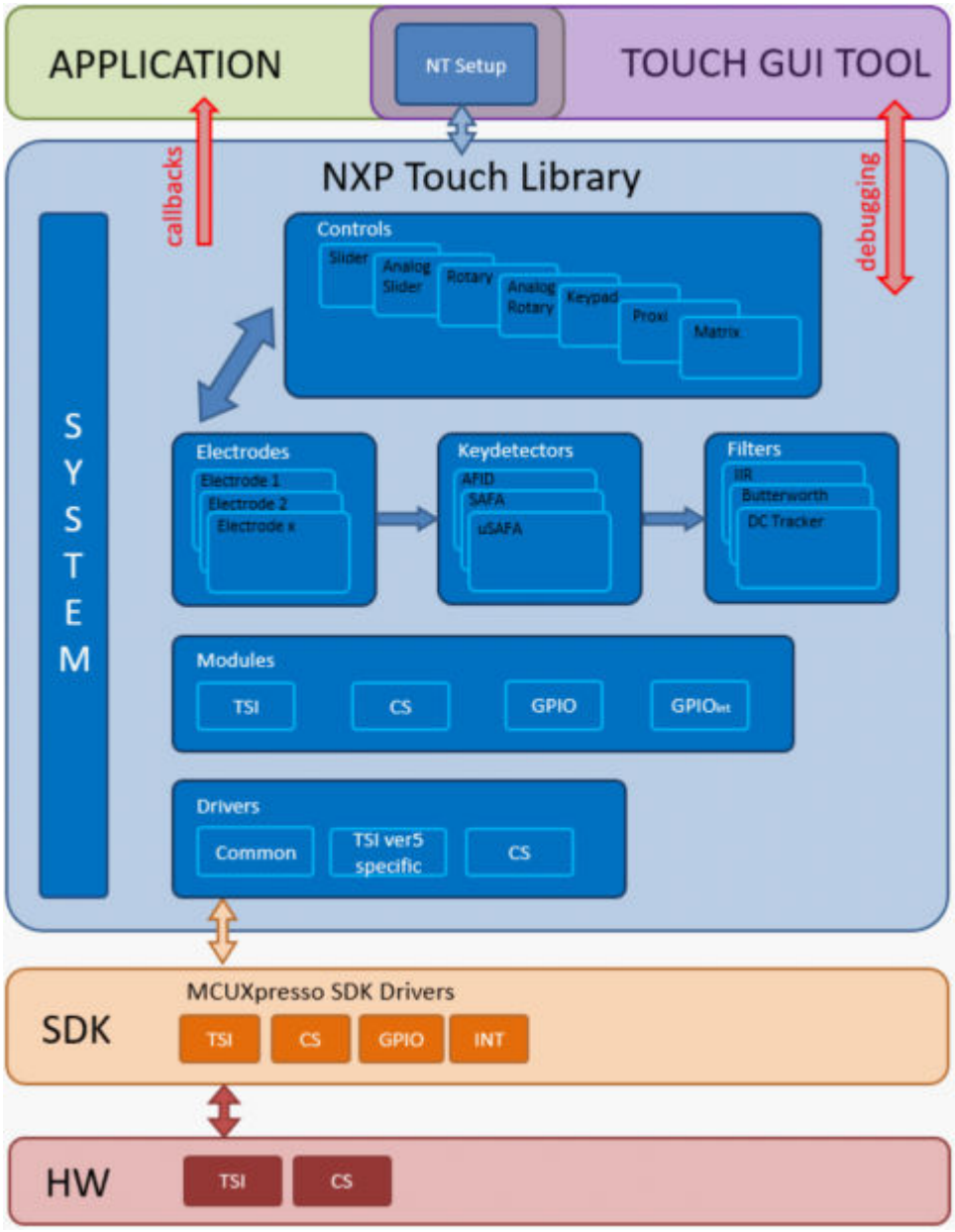


Figure 2. NXP Touch SW library block diagram

4 FreeMASTER Run-Time Debugging Tool

FreeMASTER software is debugging and visualization tool by NXP available for free download. Communication with target processor is supported via debug port (JTAG/SWD for Kinetis) or serial port (UART) interface. Direct JTAG/SWD connection is faster and does not require additional UART connection with board. Serial port driver code for target MCU is available for most MCU targets. Any variable in the project can be easily monitored and modified (reads memory based on the Target-Side Addressing TSA tables). HTML / JavaScript based GUIs can be created for further functionality.

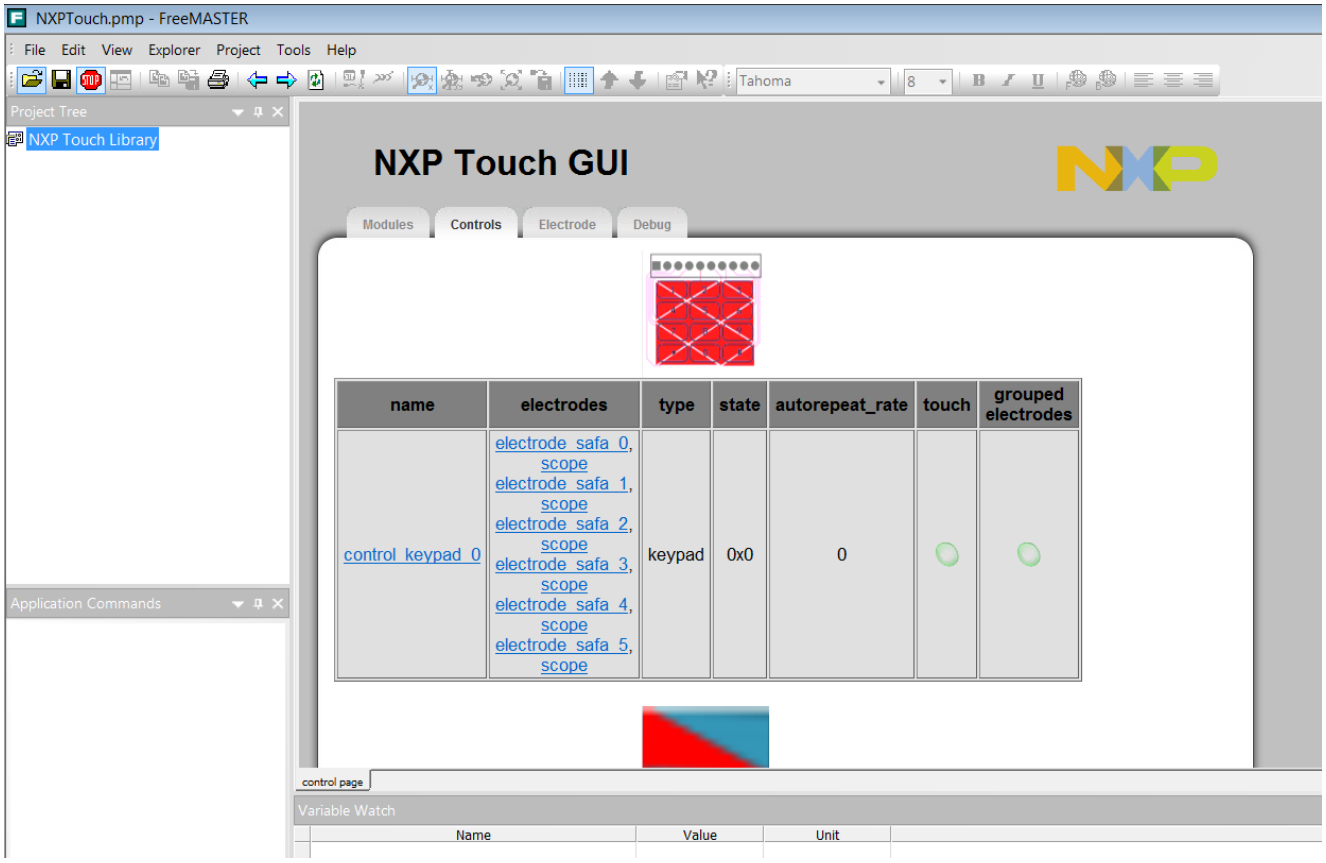


Figure 3. NXP Touch GUI in FreeMASTER

5 Supported compilers

The NXP touch library is distributed as an optional middleware component selectable during the Kinetis SDK download. Proper compiler must be selected before SDK package generation.

Supported compiler IDEs are listed below:

- IAR EWArm
- Keil uVision
- MCUXpresso available for free from NXP.

5.1 Download and Install MCUXpresso

MCUXpresso can be downloaded from the NXP webpage:

https://www.nxp.com/search?keyword=mcuxpresso

Follow the steps below to install the IDE.

- Create login if not already registered.
- Download installer.
- Install SW.
- Launch MCUXpresso.

6 Software download

Follow the steps below to build and download the SDK for FRDM-KE15z:

- Go to the webpage

https://mcuxpresso.nxp.com/en/welcome

- Login to NXP
- Select Development Board
- From Boards/Kinetis menu, select FRDM-KE15Z
- Build SDK

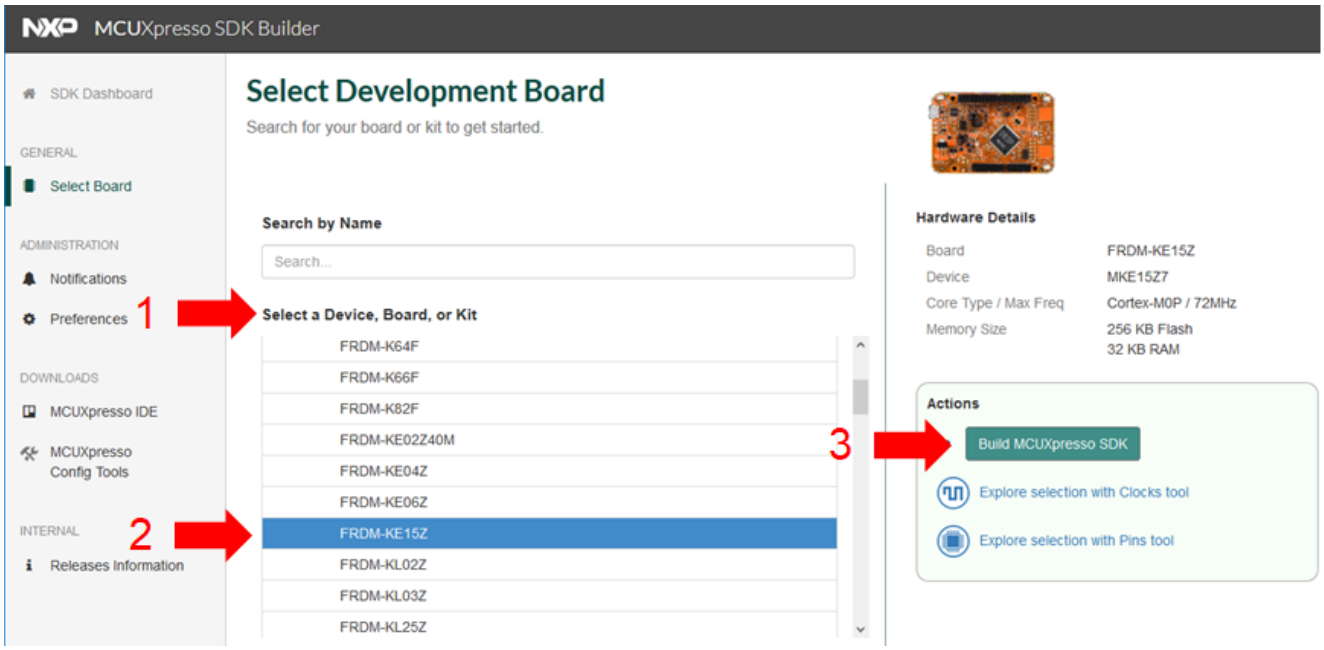


Figure 4. Building SDK package

6.1 Add touch support to SDK

Follow the steps below to add “Touch” optional middleware (software component) and select the proper Toolchain / IDE. You can build the package for single toolchain/IDE or select all toolchains supported.

- Add software component (optional middleware)
- Select Touch
- Save changes
- Note SDK Version
- Use version in prefix for archive name
- Request Build

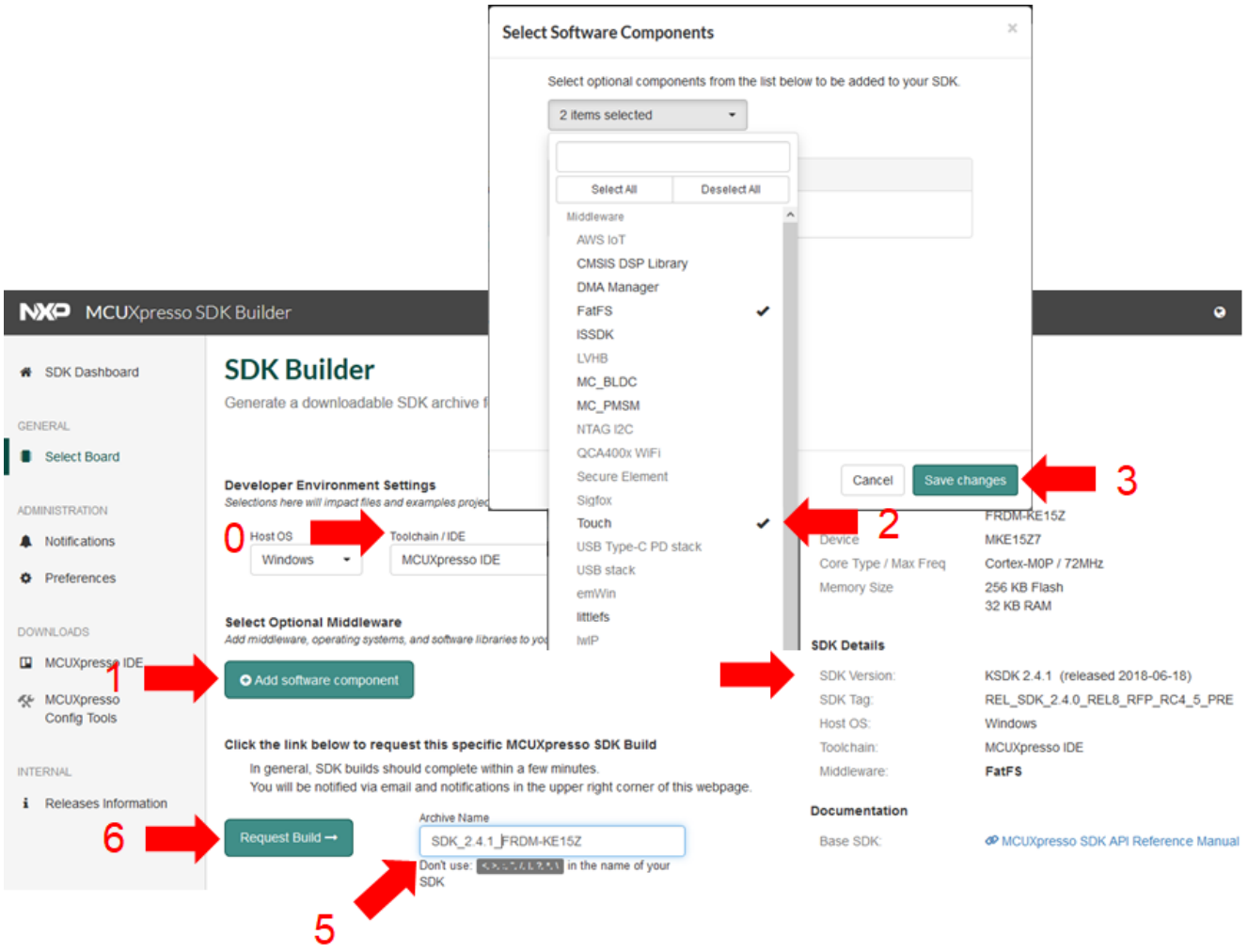


Figure 5. Add Touch Support to SDK

6.2 Downloading SDK and Documentation

- Download SDK Archive
- Download SDK Documentation
- Agree to EULAs

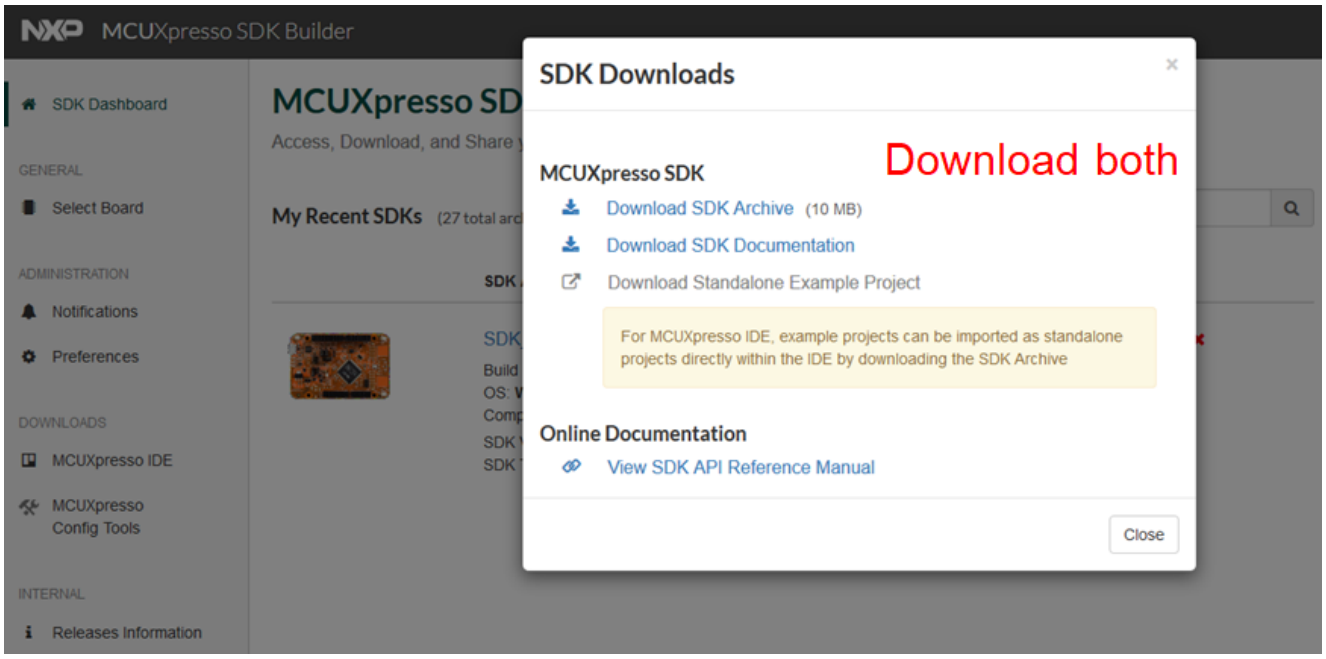


Figure 6. SDK Downloads

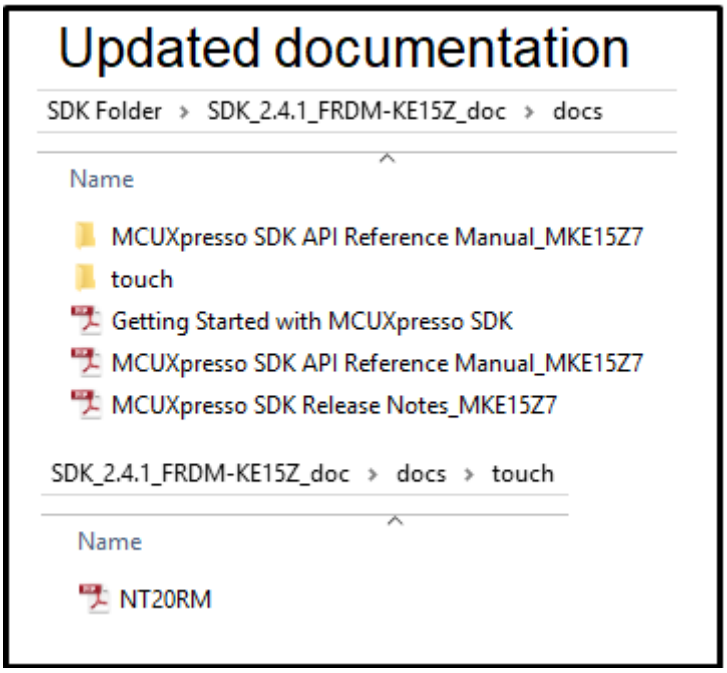


Figure 7. SDK Documentation download

6.3 FreeMASTER Download and Installation

The latest version of the FreeMASTER is available for free download from NXP webpage below:

https://www.nxp.com/support/developer-resources/software-development-tools/freemaster-run-time-debugging-tool:FREEMASTER

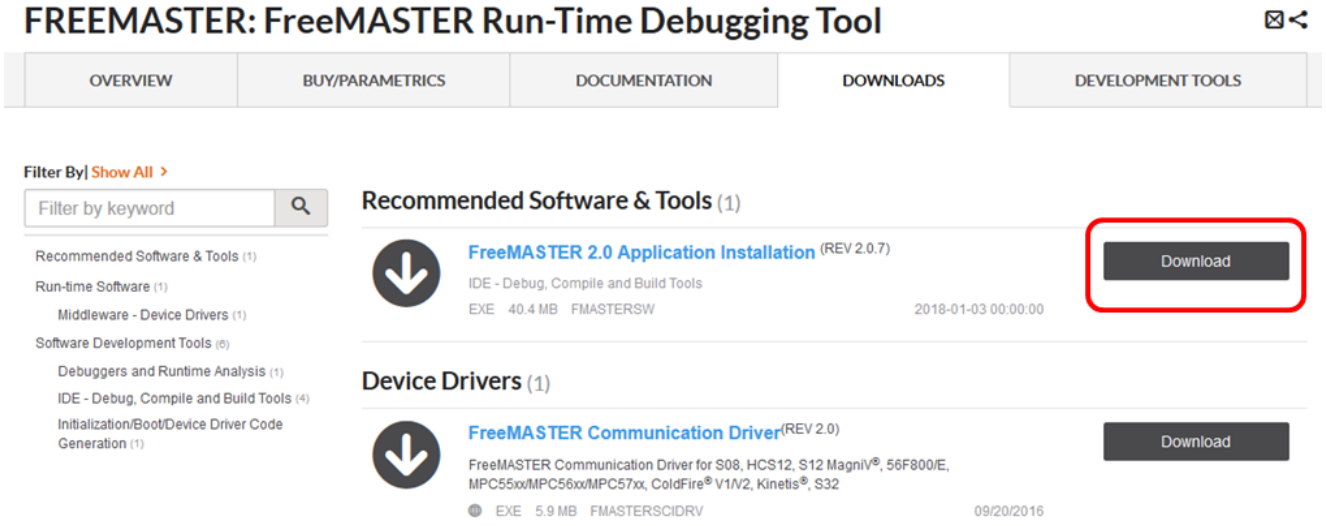


Figure 8. FreeMASTER Application Download

7 Beginning with FRDM board and Touch Demo

7.1 FRDM board setup

- Ensure that J15 is across positions 2 and 3 for 3.3 V operation
- Connect the FRDM-TOUCH board to FRDM-KE15Z. Power on the Freedom board by connecting the USB cable to your PC.
- The “touch_sensing” demo must be flashed to board.

7.2 Touch sensing demo example

- Download the latest FRDM-KE15Z SDK package from the MCUXpresso website as described in the chapter above.
- Launch MCUXpresso
- Drag the SDK package (.zip) to the Installed SDKs view in MCUXpresso to install the SDK

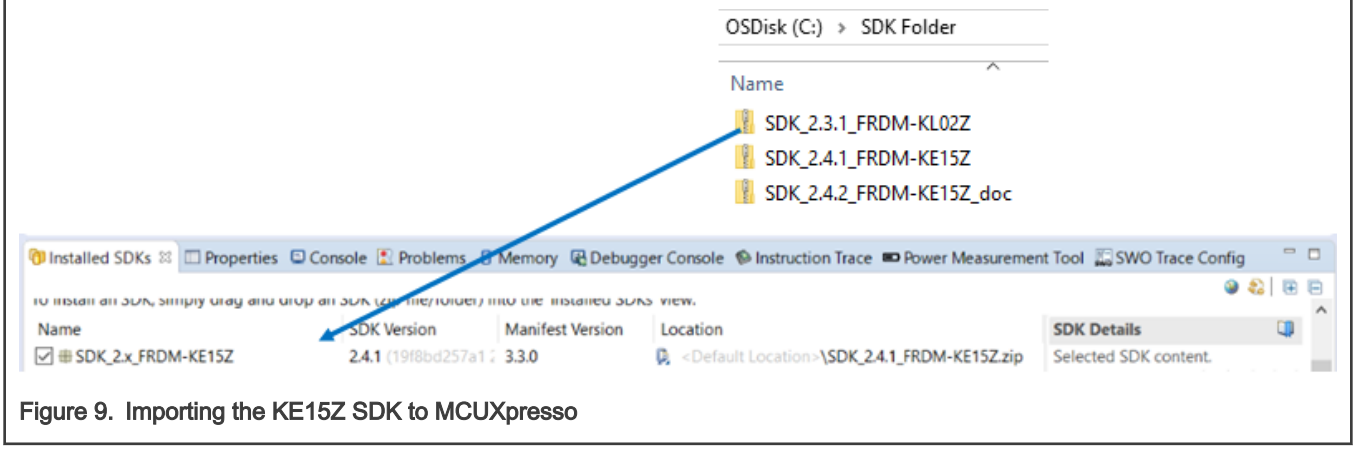


Figure 9. Importing the KE15Z SDK to MCUXpresso

7.2.1 Importing the touch_sensing demo

Follow the steps below to import the sw example project to the MCUXpresso IDE.

- Import SDK example (touch_sensing)
- Select frdmke15z, click Next
- Expand demo_apps
- Check touch_sensing, click Next

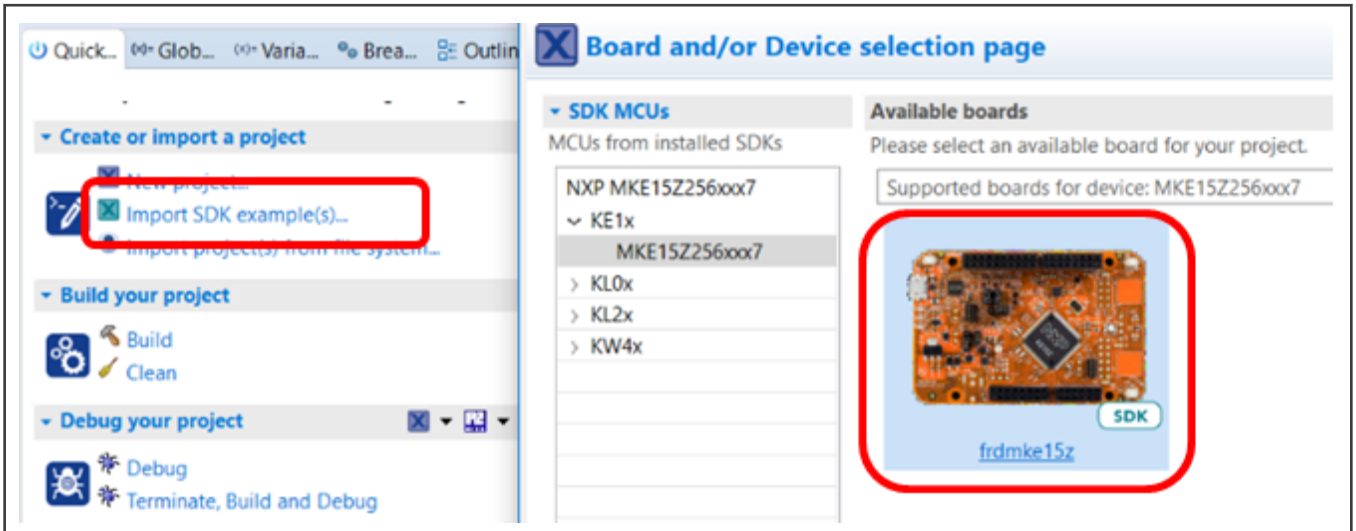


Figure 10. Importing the SDK examples

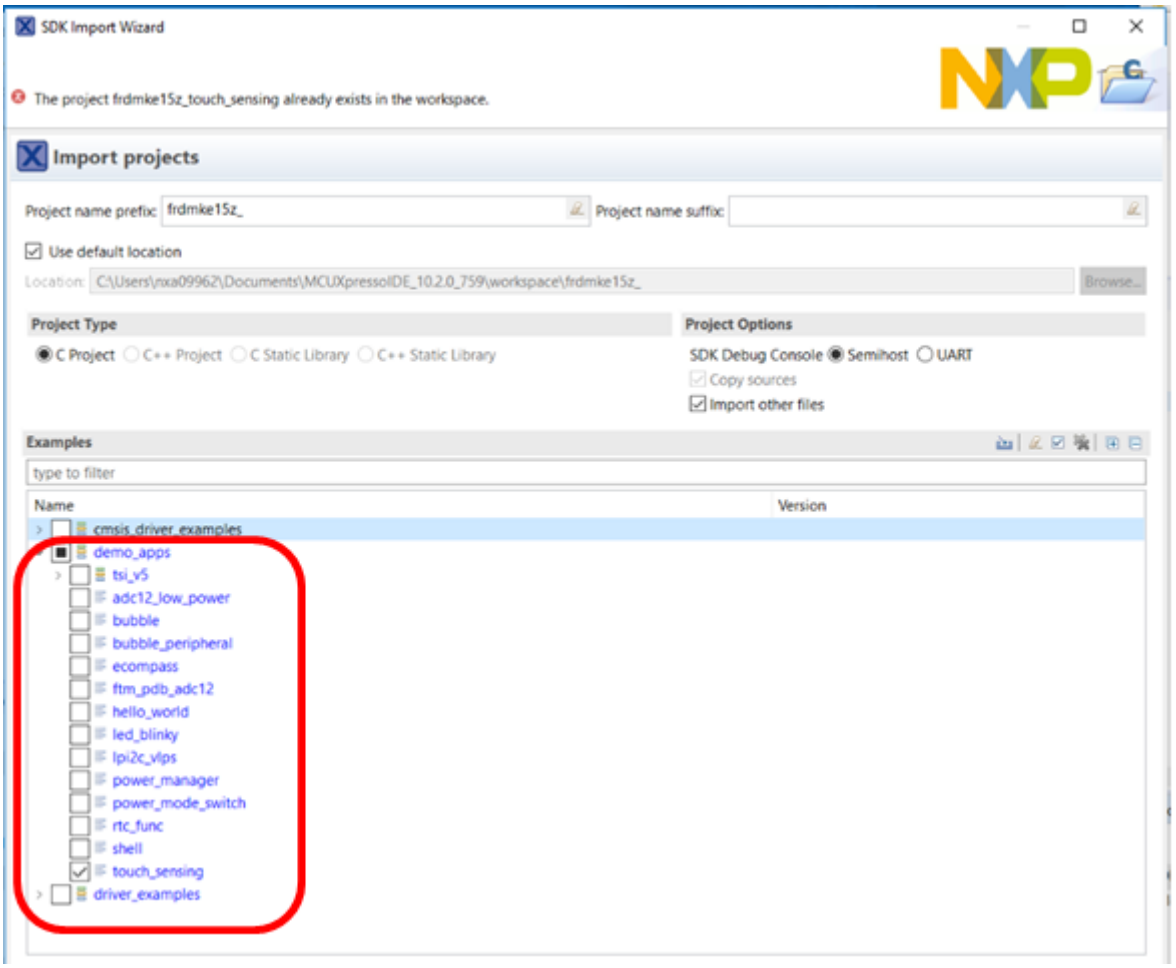


Figure 11. Importing the touch sensing demo

7.2.2 Running and debugging the touch sensing example

- In the MCUXpresso IDE Project window, expand frdmke15z_touch_sensing
- Double-click main.c
- Build and debug the touch sensing project (to load the app into flash)
- Click the Resume icon to run the demo

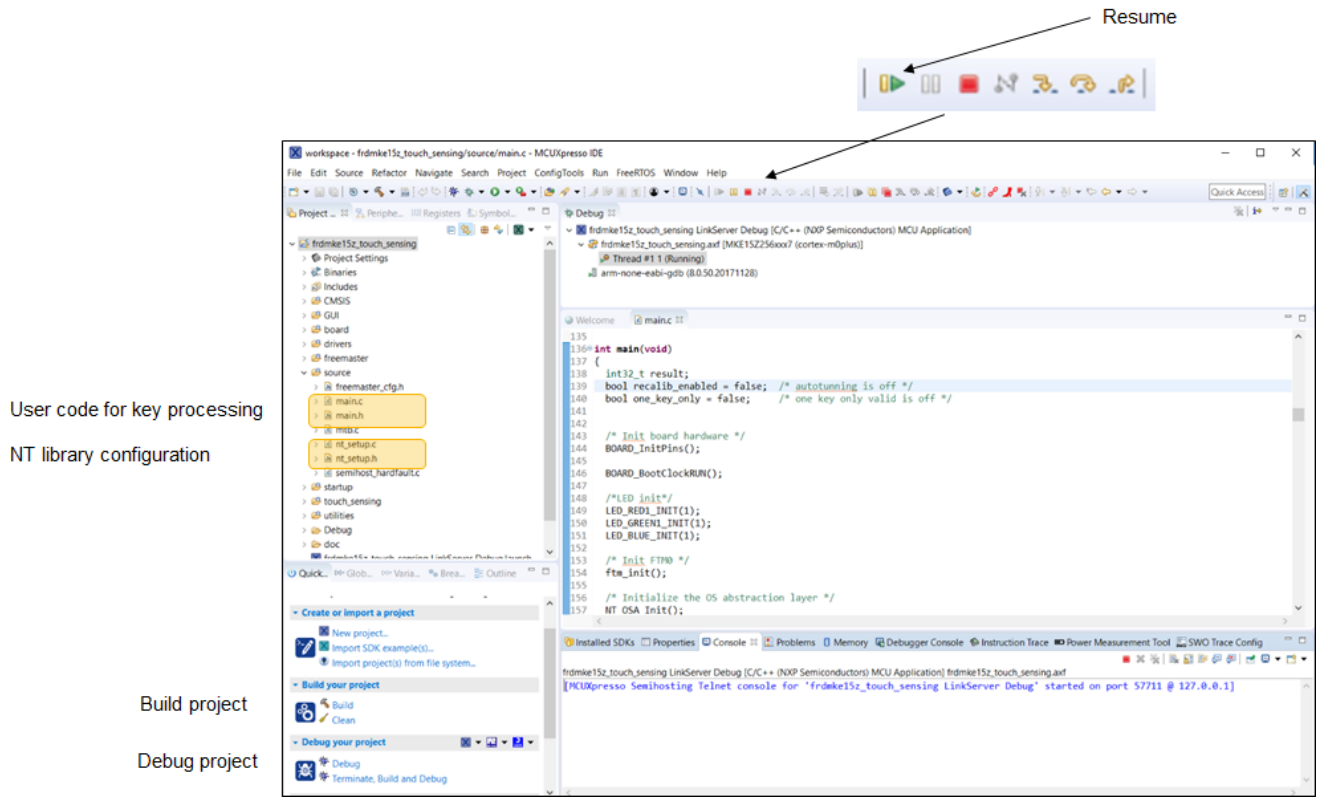


Figure 12. Debugging the demo

7.2.3 Explore the touch_sensing demo

- Press E1 on FRDM-KE15Z – RGB LED turns Yellow
- Press E2 on FRDM-KE15Z – RGB LED turns Cyan
- Press Up on FRDM-TOUCH – RGB LED turns Yellow
- Press Right on FRDM-TOUCH – RGB LED turns Green
- Press Down on FRDM-TOUCH – RGB LED turns Blue
- Press Left on FRDM-TOUCH – RGB LED turns White
- Scroll slider on FRDM-TOUCH from right to left – RGB intensity is reduced
- Scroll around rotary on FRDM-TOUCH – RGB hues change

7.3 Standalone FreeMASTER GUI

- With FRDM-TOUCH installed on FRDM-KE15Z and the touch_sensing demo loaded in flash, connect USB cable between board and PC

-All the touch electrodes should respond by changing color or intensity on the RGB LED.

- Launch FreeMASTER
- Drag the NXPTouchKE15Z.pmp file onto the FreeMASTER window

-Project Tree changes to NXP Touch Library

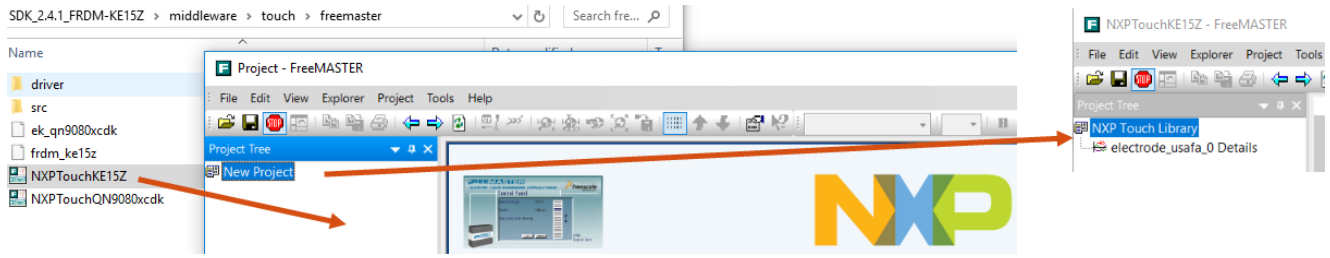


Figure 13. FreeMASTER Example

7.3.1 FreeMASTER GUI Connection

Freemaster supports several communication protocols like serial COMx or direct connection over the debug interface. Follow the steps to connect via onboard Mbed USB to serial port.

- Select Connection Wizard from Tools menu and click Next
- Use direct connection to on-board USB port, Next
- Ensure Mbed Serial Port (COMxx) selected, Next
- COMxx UART port will be detected, select Yes, Finish

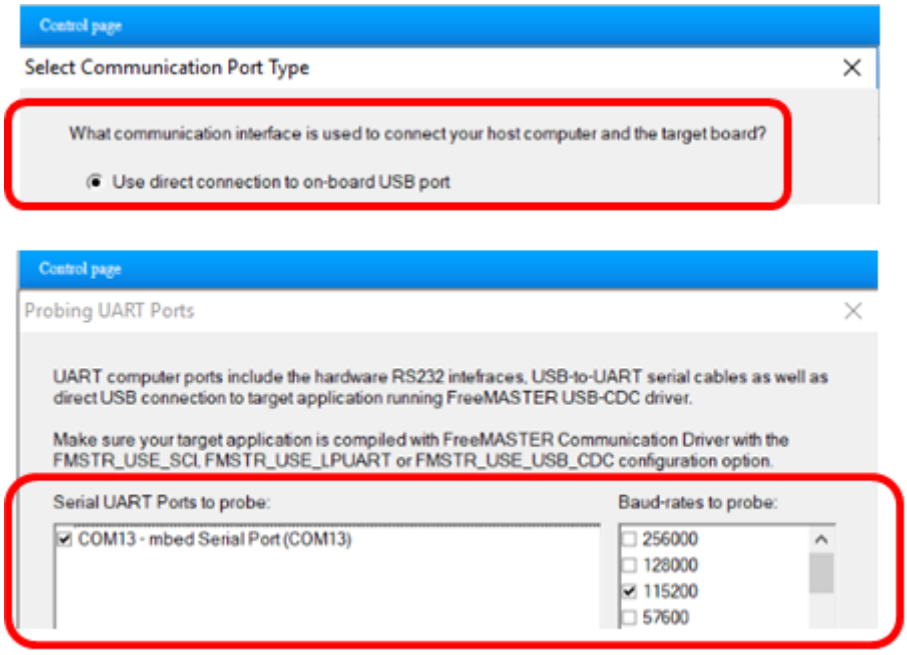


Figure 14. Set FreeMASTER communication

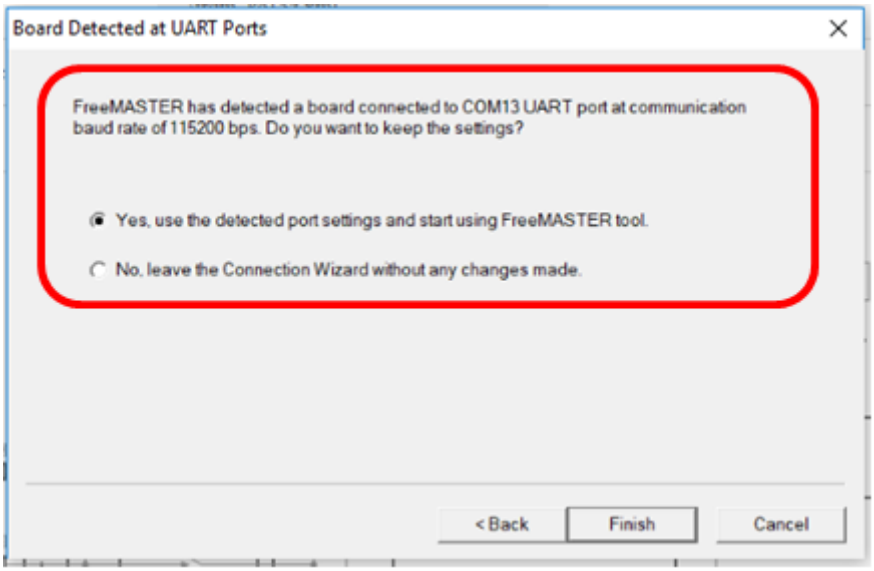


Figure 15. Select UART

7.3.2 TOUCH SW LAYERS Tab

To see the information about the touch sw components select the “TOUCH SW LAYERS” tab in the Control page window and then click “READ CONFIGURATION FROM BOARD” and wait for upload to complete.

You can scroll down to examine:

- NT CONTROLS

Keypad_1, Aslider_2, Arotary_3 and electrodes used for each control

- NT ELECTRODES

Information for all 12 electrodes used

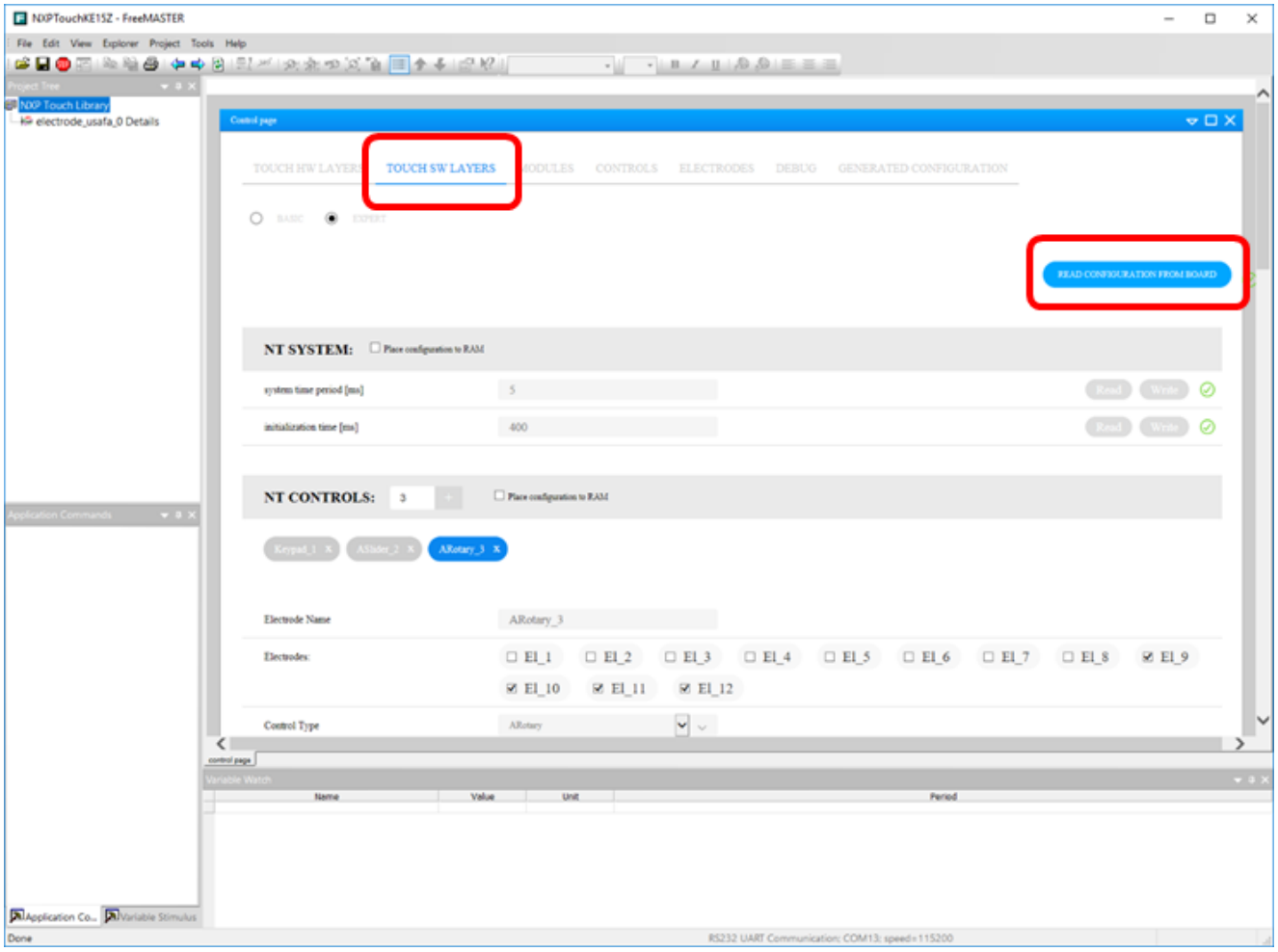


Figure 16. FreeMASTER-Touch SW Layers

7.3.3 MODULES Tab

If the MODULES tab is selected in Control page window, all 12 electrodes defined in sample project are shown. Signal values change during scans. When the Up arrow on FRDM-TOUCH is pressed a valid touch is indicated for electrode 2. Please note that electrodes are indexed starting from zero, no matter on the electrode names defined in SW setup.

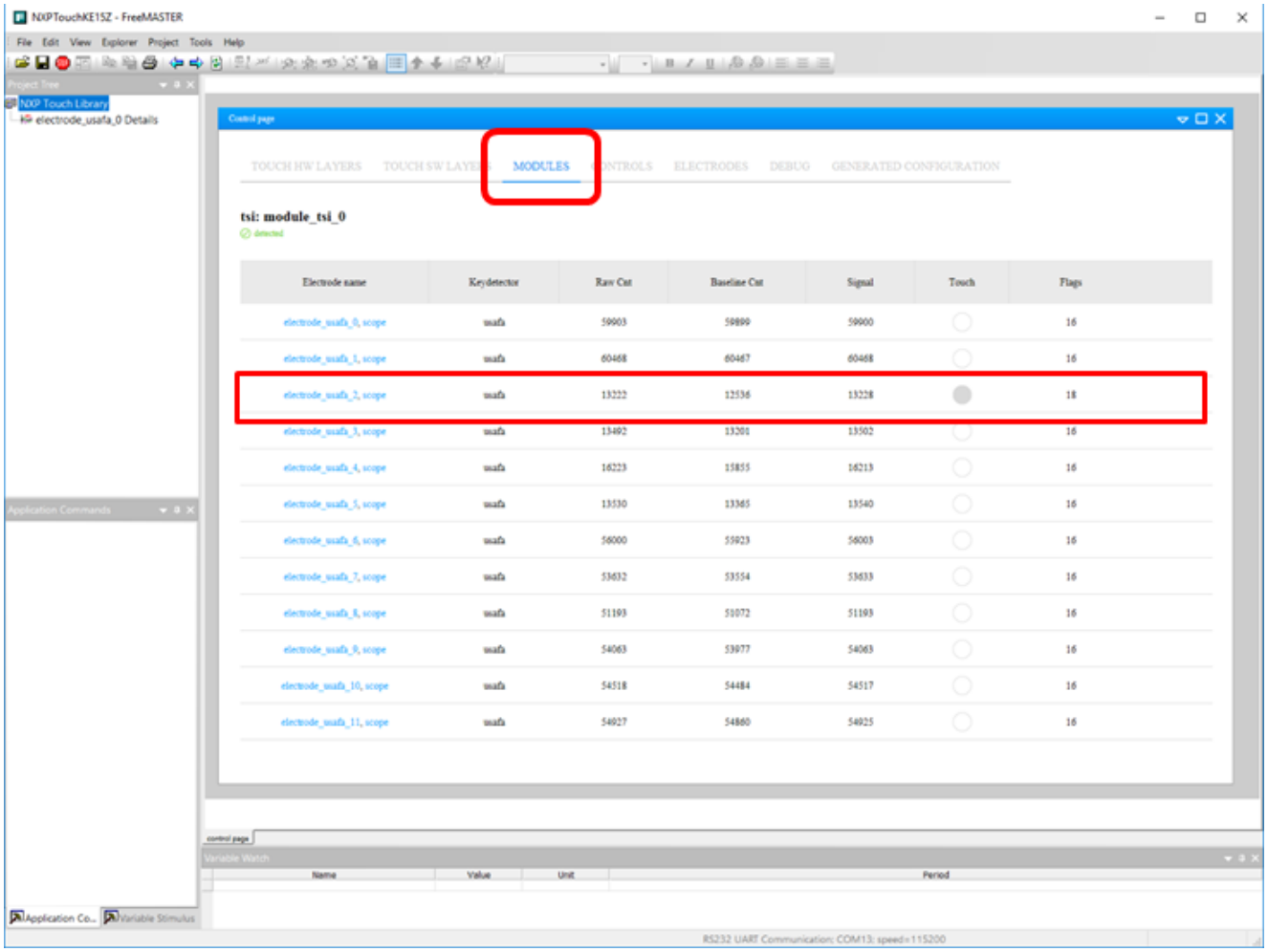


Figure 17. FreeMASTER-MODULES Tab, electrodes view

```
const struct nt_electrode * const nt_tsi_module_electrodes[] = {
  &El_1,  &El_2,  &El_3,  &El_4,  &El_5,  &El_6,  &El_7,  &El_8,  &El_9,  &El_10,  &El_11,  &El_12, NULL
};
```

Figure 18. Module electrode assignment in “nt_setup.c”

7.3.4 CONTROLS Tab

If the CONTROLS tab in Control page window is selected, the 3 touch controls (keypad, rotary, and slider) in sample project are shown.

When any of the keypad electrodes on FRDM-TOUCH is pressed a valid touch is indicated for the keypad control.

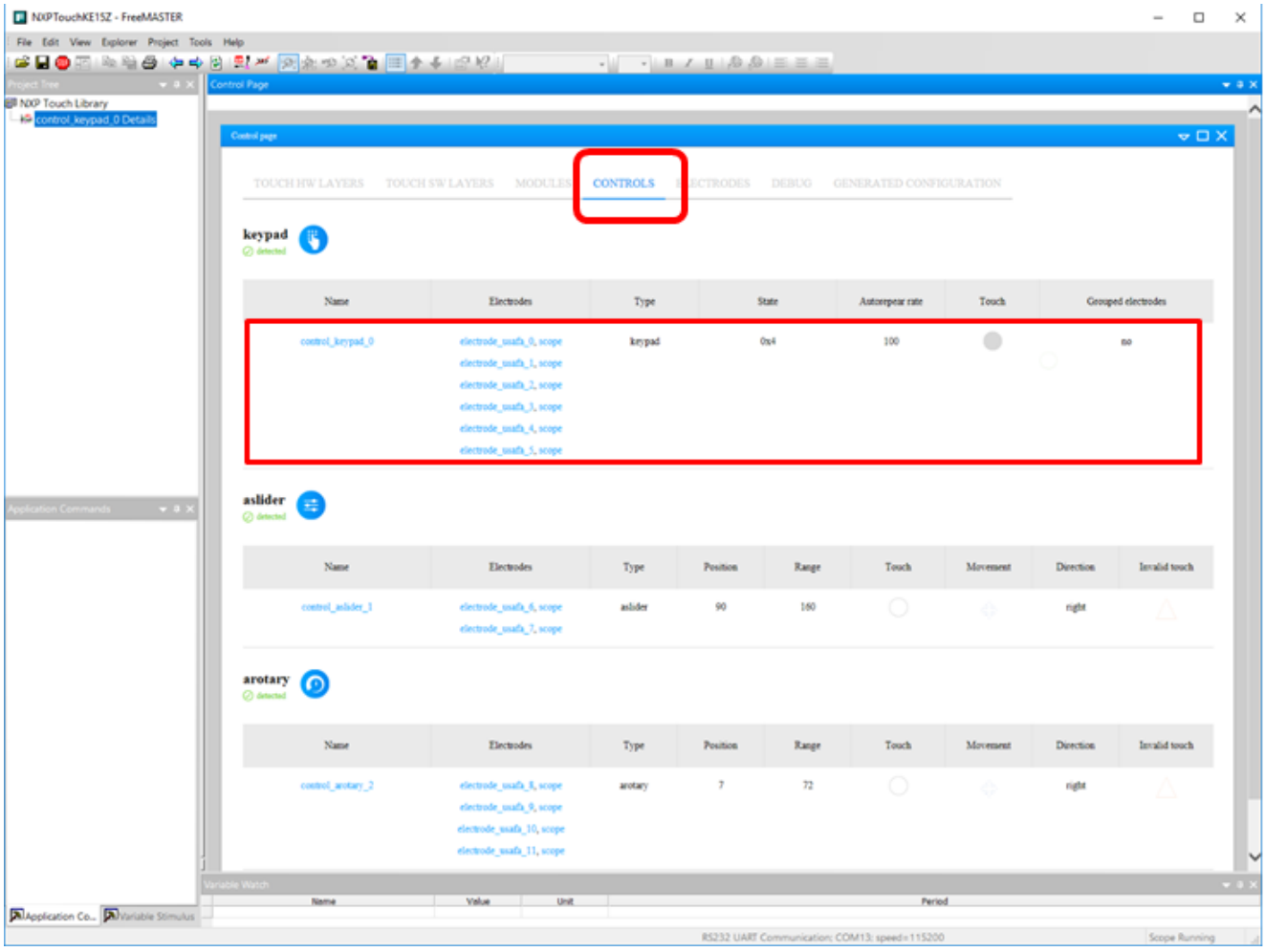


Figure 19. CONTROLS Tab - keypad

7.3.5 ELECTRODES Tab

Detailed information about the electrode Touch/Release events can be monitored in ELECTRODES Tab. Follow the steps below, please:

- Select the ELECTRODES tab in Control page window
- Use the SELECTED ELECTRODE pulldown to select electrode_usafa_1
- Press and hold the E2 electrode on FRDM-KE15Z and a valid touch is shown in State 0 (or 2)
- Release the E2 electrode and a valid release is shown in State 1 (or 4)

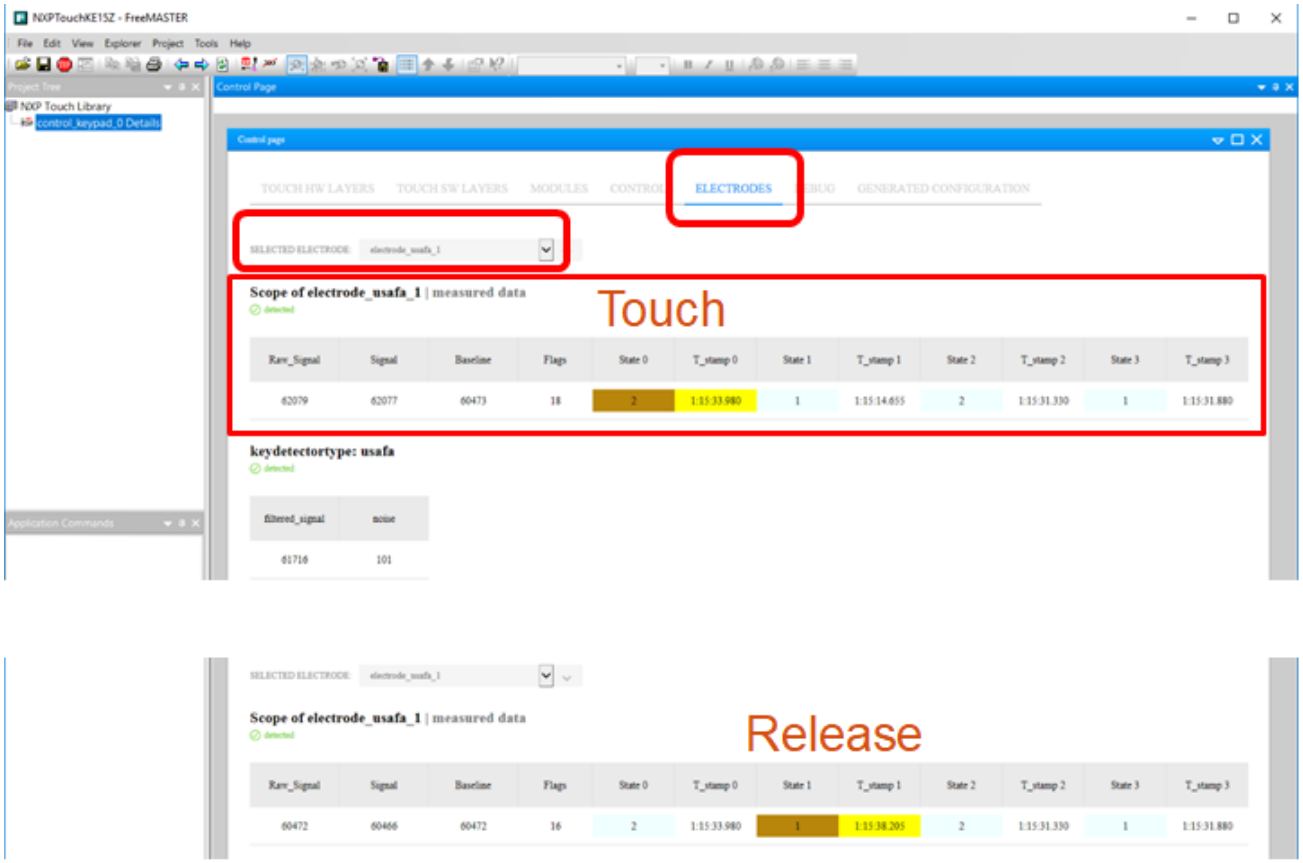


Figure 20. ELECTRODED Tab – Touch and Release events

7.3.6 FreeMASTER Oscilloscope View

After clicking to “scope”, the most of the important touch sensing signals and key detector values can be monitored. Just single electrode can be selected and monitored. Follow the steps below to watch the signals for the selected electrode, several options are possible:

- From the MODULES tab, click the scope link for an electrode

(An error appears if you mistakenly click the electrode_usafa_x, just click Yes)

- From the CONTROLS tab, click the Name of the control to see all electrodes in that control
- From the CONTROLS tab, click the scope link for a single electrode
- From the ELECTRODES tab, select an electrode from the scroll window, then click oscilloscope at the bottom of the Control page window

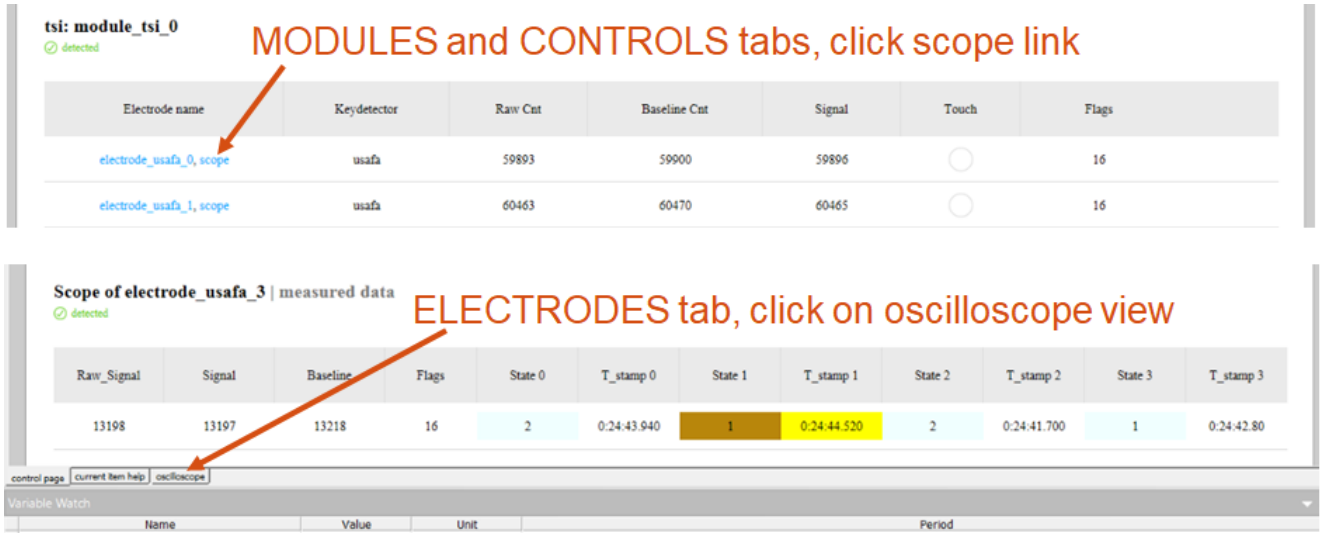


Figure 21. Entering Oscilloscope view

7.3.7 Single Electrode Scope View

All-important key detector signals are visible in the scope window so that the sensitivity and touch thresholds can be easily tuned for the individual electrodes. When Electrode 2 (Up arrow) is pressed on FRDM-TOUCH, the signal step and touch detection are visible.

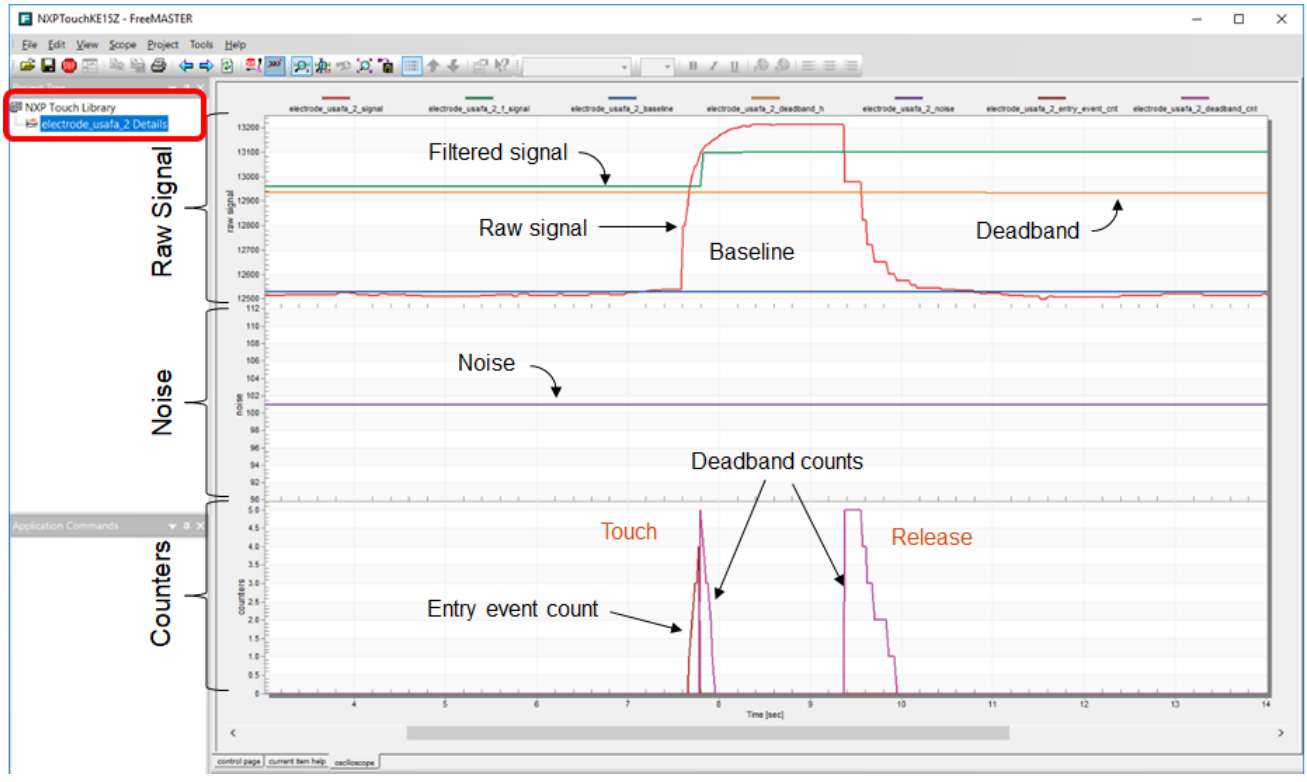


Figure 22. Single Electrode Scope View – key detector signals

7.3.8 Slider Control Scope View

Values like slider detected position, keypad button masks (flags) can be monitored as well.

For instance, analog slider consists from two electrodes. If we slide by finger from left to right, we will see the signal changes on electrode 0 and electrode 1. The position and direction is calculated from both electrode signals.

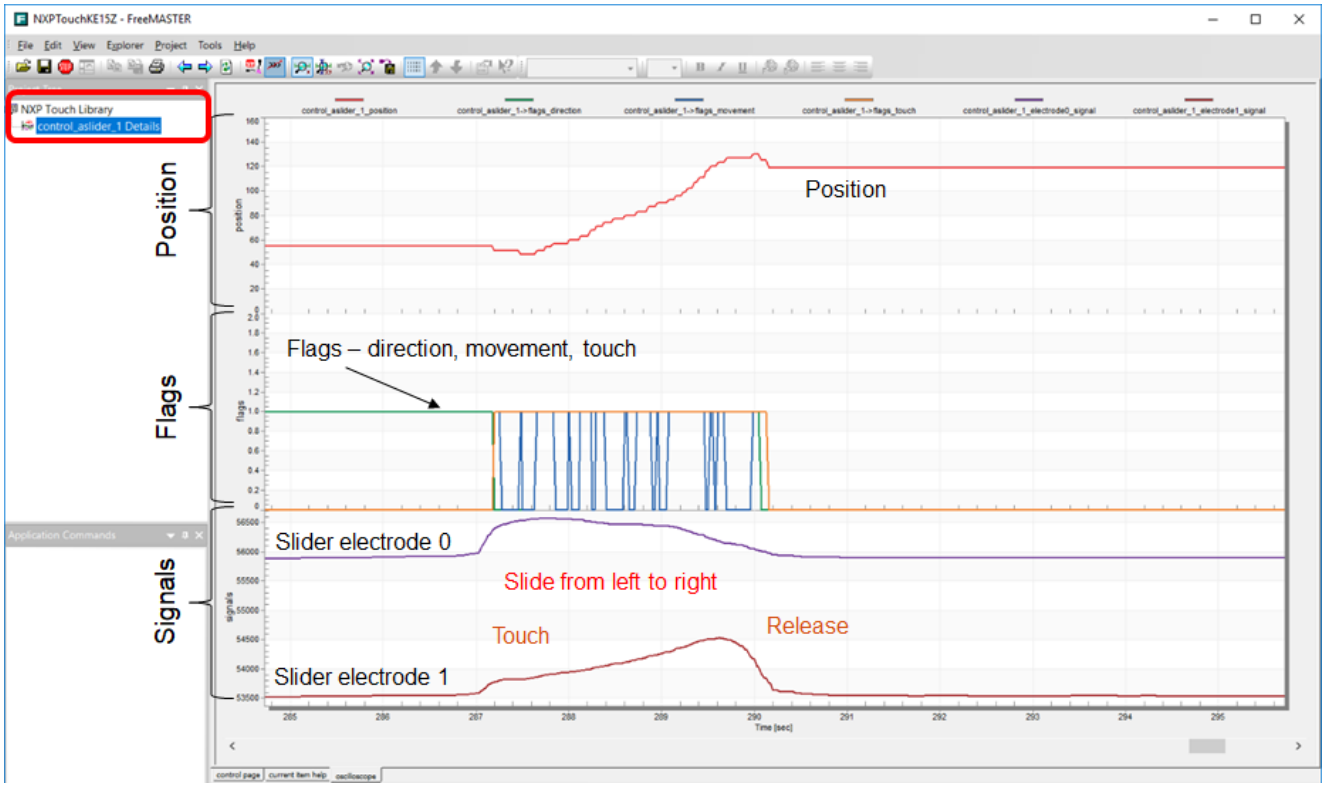


Figure 23. Slider signals

7.4 Touch Sensing demo SW configuration

7.4.1 FRDM-TOUCH

RGB LED on FRDM-KE15Z responds to touches on:

- 4 Touch buttons
- Touch Rotary
- Touch slider

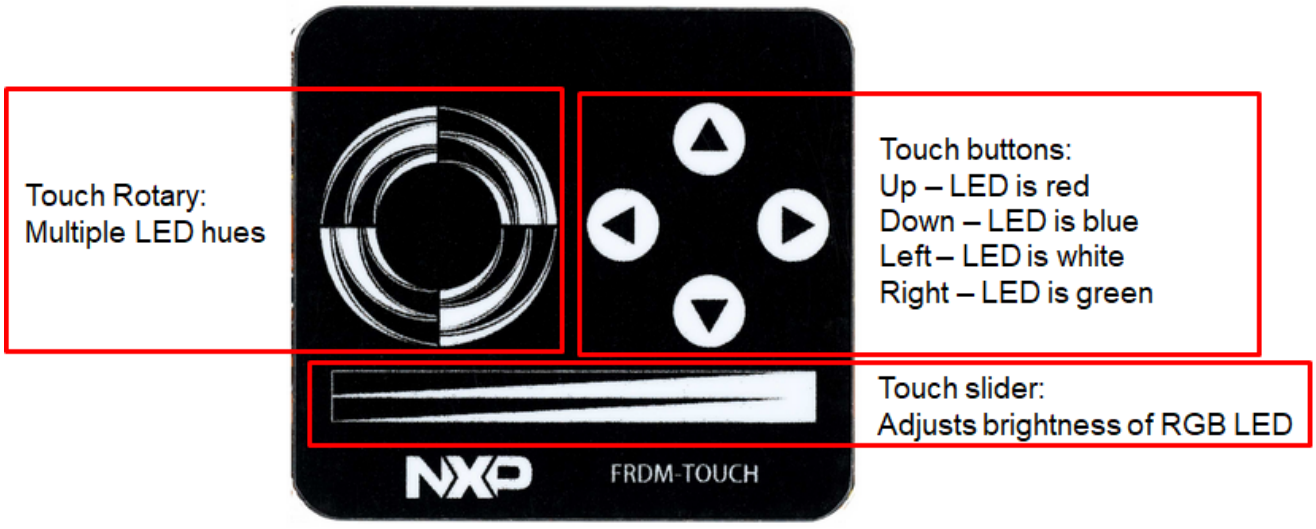


Figure 24. FRDM-TOUCH controls

7.4.2 TSI channel assignment

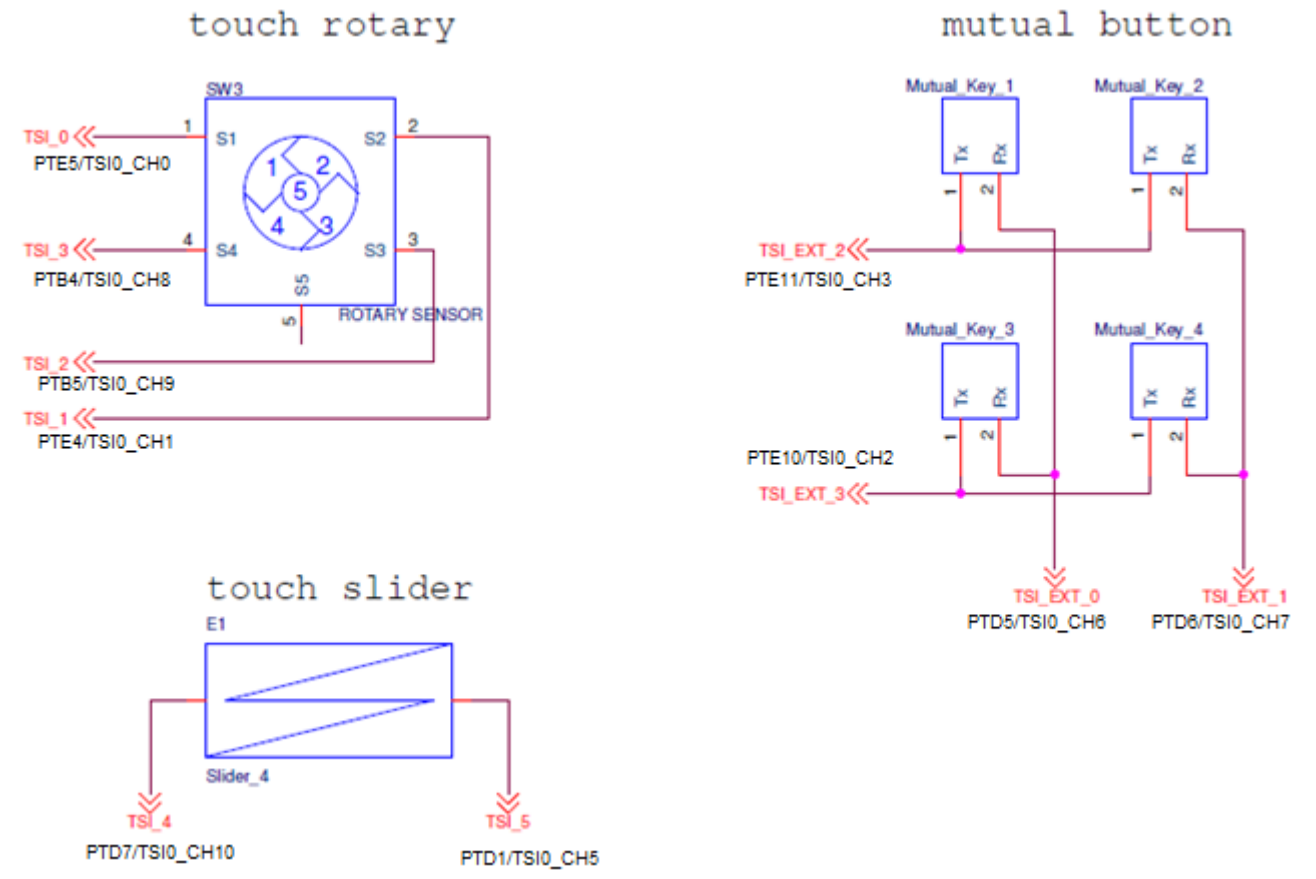


Figure 25. Schematic of TSI inputs

7.4.2.1 Assigning the TSI channels in SW

The touch demo SW is based on Kinetis SDK. The advantage of this is that the SDK contains support for all MCU peripherals, like UART, Timers, SPI drivers, and so on.

However, only a few files from SDK are utilized by the touch library. Mainly the low-level drivers for TSI and timer are needed. But these files may be replaced by the user-defined drivers. It means that the library source files can be easily integrated to another SW project independently on Kinetis SDK.

In the Kinetis SDK, the virtual electrode assignment to the physical TSI channel is done in “board.h” file.

```
/* Push buttons - mutual electrodes */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_1  NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_1,FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_1)//TF_TSI_MUTUAL_CAP_TX_CHANNEL_3
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_2  NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_2,FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_2)//TF_TSI_MUTUAL_CAP_TX_CHANNEL_3
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_3  NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_3,FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_3)//TF_TSI_MUTUAL_CAP_TX_CHANNEL_2
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_4  NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_4,FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_4)//TF_TSI_MUTUAL_CAP_TX_CHANNEL_2

#define FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_1  TF_TSI_MUTUAL_CAP_TX_CHANNEL_3      /* PTE11 */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_1  TF_TSI_MUTUAL_CAP_RX_CHANNEL_6      /* PTD5  */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_2  TF_TSI_MUTUAL_CAP_TX_CHANNEL_3      /* PTE11 */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_2  TF_TSI_MUTUAL_CAP_RX_CHANNEL_7      /* PTD6  */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_3  TF_TSI_MUTUAL_CAP_TX_CHANNEL_2      /* PTE10 */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_3  TF_TSI_MUTUAL_CAP_RX_CHANNEL_6      /* PTD5  */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_4  TF_TSI_MUTUAL_CAP_TX_CHANNEL_2      /* PTE10 */
#define FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_4  TF_TSI_MUTUAL_CAP_RX_CHANNEL_7      /* PTD6  */

/* Slider - self electrodes */
#define FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_1  TF_TSI_SELF_CAP_CHANNEL_10     /* PTD1 */
#define FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_2  TF_TSI_SELF_CAP_CHANNEL_5      /* PTD7 */
#define FRDM_TOUCH_BOARD_TSI_1  TF_TSI_SELF_CAP_CHANNEL_22
#define FRDM_TOUCH_BOARD_TSI_2  TF_TSI_SELF_CAP_CHANNEL_23

/* Rotary - self electrodes */
#define FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_1  TF_TSI_SELF_CAP_CHANNEL_0      /* PTE5 */
#define FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_2  TF_TSI_SELF_CAP_CHANNEL_1      /* PTE4 */
#define FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_3  TF_TSI_SELF_CAP_CHANNEL_9      /* PTB5 */
#define FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_4  TF_TSI_SELF_CAP_CHANNEL_8      /* PTB4 */
```

Figure 26. TSI channel assignment in SW

7.4.3 Application code in main() Function

```
if ((result = nt_init(&System_0, nt_memory_pool, sizeof(nt_memory_pool))) != NT_S
{
  switch(result)
  {
  case NT_FAILURE:
    nt_printf("\nCannot initialize NXP Touch due to a non-specific error.\n");
    break;
  case NT_OUT_OF_MEMORY:
    nt_printf("\nCannot initialize NXP Touch due to a lack of free memory.\n");
    break;
  }
  while(1); /* add code to handle this error */
}

nt_printf("\nNXP Touch is successfully initialized.\n");

nt_printf("Unused memory: %d bytes, you can make the memory pool smaller without

/* Enable electrodes and controls */
nt_enable();

/* Keypad electrodes*/
nt_control_keypad_set_autorepeat_rate(&Keypad_1, 100, 1000);
nt_control_keypad_register_callback(&Keypad_1, &keypad_callback);

/* Slider electrodes */
nt_control_aslider_register_callback(&ASlider_2, &aslider_callback);

/* Rotary electrodes */
nt_control_arotary_register_callback(&ARotary_3, &arotary_callback);

/* System TSI overflow warning callback */
nt_system_register_callback(&system_callback);

/* Auto TSI register recalibration function, not used as default */
if (recalib_enabled)
{
    recalib_status = (tsi_status_t) nt_module_recalibrate(&nt_tsi_module);
}

if (one_key_only)
nt_control_keypad_only_one_key_valid(&Keypad_1, true);

pit_init();

while(1)
{
  nt_task();

  FMSTR_Poll();
}
}
```


Figure 27. Application init

7.4.4 SW library synchronization and processing functions

There are two main api functions: “nt_trigger()” and “nt_task()” that must be periodically called by the SW application in order to trigger the touch sensing measurement and process the results.

7.4.4.1 nt_trigger function

This function should be called by the application periodically in a timer interrupt, or in a task to trigger

new data measurement. Depending on the module implementation, this function may take the data

immediately, or may only start the hardware sampling with interrupt enabled. If the TSI module measurement is triggered, whole sequence of input channels will be scanned consequently. It means that the new measurement sequence should not be triggered until the previous sequence is completed. The function returns:

- NT_SUCCESS when the trigger was performed without any errors or warnings.
- NT_FAILURE when a problem is detected, such as module not ready, overrun (data loss) error,

and so on. Regardless of the error, the trigger is always initiated.

This is an example of the NT library triggering:

```
//For example, there is a callback routine from any periodical source (for example 5 ms)
static void Timer_5msCallBack(void)
{
 if(nt_trigger() != NT_SUCCESS)
  {
    // Trigger error
  }
}
```

Figure 28. nt_trigger() function call

7.4.4.2 nt_task function

This function should be called by the application as often as possible, in order to process the data acquired during the data trigger. This function should be called at least once per trigger time. Internally, this function passes the NT_SYSTEM_MODULE_PROCESS and NT_SYSTEM_CONTROL_PROCESS command calls to each object configured in Modules and Controls. The function returns:

- NT_SUCCESS when data acquired during the last trigger are now processed
- NT_FAILURE when no new data are ready

This is an example of running a task of the NT library:

```
// Main never-ending loop of the application
while(1)
{
  if(nt_task() == NT_SUCCESS)
  {
    // New data has been processed
  }
}
```

Figure 29. nt_task() function call

7.4.5 Event callback functions

Every user enabled control must have its “callback” function defined, which is used for servicing the events generated by control like “Touch” or “Release” events. See the example below for keypad callback function.

```
static void keypad_callback(const struct nt_control *control,
                            enum nt_control_keypad_event event,
                            uint32_t index)
{
  switch(event)
  {
  case NT_KEYPAD_RELEASE:

    switch (index) {
        case 0:
            break;
        case 1:
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        default:
            break;
    }
    break;
  case NT_KEYPAD_TOUCH:

    switch (index) {
        case 0:
            break;
        case 1:
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        default:
            break;
    }
    break;
```

Release event

Touch event

Figure 30. Keypad callback function

7.4.6 SW Application setup in “nt_setup.c” file

Most of the configuration, like TSI register HW sensitivity, key detector settings, assignments of electrodes and global timebase is provided in “nt_setup.c”.

To define the modules, electrodes, controls, and system, create the initialized instances of the structure types, as described in the following section.

The code below shows an example configuration of four electrodes on the FRDM-KE15z board.

There are several key detectors (touch-evaluation algorithms) available in the NXP Touch library. The

electrode structure types must always match the module and algorithm types.

In the SW demo example key detector “uSAFA” is used. All keys can share single key detector settings or different key detector setting structures can be assigned to different electrodes for better flexibility.

```
struct nt_keydetector_usafa nt_keydetector_usafa_El_1 = {
  .signal_filter.coef1 = 2,
  .base_avrg.n2_order = 9,
  .non_activity_avrg.n2_order = 15,
  .entry_event_cnt = 1,
  .deadband_cnt = 1,
  .signal_to_noise_ratio = 5,
  .min_noise_limit = 200,
  .dc_track_enabled = 0,
  .dc_track_cnt = 100,
};
const struct nt_keydetector_usafa nt_keydetector_usafa_El_3 = {
  .signal_filter.coef1 = 2,
  .base_avrg.n2_order = 12,
  .non_activity_avrg.n2_order = 15,
  .entry_event_cnt = 4,
  .deadband_cnt = 5,
  .signal_to_noise_ratio = 4,
  .min_noise_limit = 60,
  .dc_track_enabled = 1,
  .dc_track_cnt = 100,
};
```

placed in RAM

placed in Flash

Figure 31. Key detector SW definitions

The electrode structure types must match the hardware module used for the data-measurement algorithm in the application. In this case, it is the “nt_electrode” type. Define the electrode parameters and the “nt_key detector” interface.

```
/* Electrodes */
const struct nt_electrode El_1 = {
  .shielding_electrode = NULL,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_1,
};
const struct nt_electrode El_2 = {
  .shielding_electrode = NULL,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_2,
};
```

Figure 32. Electrode SW definitions

```
/* Modules */
const struct nt_electrode * const nt_tsi_module_electrodes[] = {
  &El_1,  &El_2,  &El_3,  &El_4,  &El_5,  &El_6,  &El_7,  &El_8,  &El_9,  &El_10,
};

const struct nt_module nt_tsi_module = {
  .interface = &nt_module_tsi_interface,
  .wtrmark_hi = 65535,
  .wtrmark_lo = 0,
  .config = (void*)&tsi_hw_config,
  .instance = 0,
  .electrodes = &nt_tsi_module_electrodes[0],
  .safety_interface = &nt_safety_interface,
  .safety_params.gpio = (void*)&my_safety_params,
  .recalib_config = (void*)&recalib_configuration,
};
```

Figure 33. TSI module definition

Once the modules and electrodes are set up, you can define the Controls. In this case, the control_0 is the Analog Slider control.

```
const struct nt_electrode * const Keypad_1_controls[] = {
  &El_1,  &El_2,  &El_3,  &El_4,  &El_5,  &El_6, NULL
};
const struct nt_electrode * const ASlider_2_controls[] = {
  &El_7,  &El_8, NULL
};
const struct nt_electrode * const ARotary_3_controls[] = {
  &El_9,  &El_10,  &El_11,  &El_12, NULL
};
const struct nt_control_arotary nt_control_arotary_ARotary_3 = {
  .range = 72,
};
const struct nt_control_aslider nt_control_aslider_ASlider_2 = {
  .range = 160,
  .insensitivity = 2,
};
const struct nt_control_keypad nt_control_keypad_Keypad_1 = {
  .groups = NULL,
  .groups_size = 0,
  .multi_touch = (uint32_t []){0x0C,0x18,0x30,0x24,0x3C,0},
  .multi_touch_size = 5,
};
const struct nt_control Keypad_1 = {
  .electrodes = &Keypad_1_controls[0],
  .control_params.keypad = &nt_control_keypad_Keypad_1,
  .interface = &nt_control_keypad_interface,
};
const struct nt_control ASlider_2 = {
  .electrodes = &ASlider_2_controls[0],
  .control_params.aslider = &nt_control_aslider_ASlider_2,
  .interface = &nt_control_aslider_interface,
};
const struct nt_control ARotary_3 = {
  .electrodes = &ARotary_3_controls[0],
  .control_params.arotary = &nt_control_arotary_ARotary_3,
  .interface = &nt_control_arotary_interface,
};
```

Keypad electrodes

Slider electrodes

Rotary electrodes

Rotary range

Slider range and granularity

Figure 34. Example of Keypad control SW definitions

Now we are ready to connect all the pieces together in the “system” structure.

```
/* System */
const struct nt_control * const System_0_controls[] = {
  &Keypad_1,  &ASlider_2,  &ARotary_3, NULL
};
const struct nt_module * const System_0_modules[] = {
  &nt_tsi_module, NULL
};
const struct nt_system System_0 = {
  .time_period = TIME_PERIOD,
  .init_time = 400,
  .safety_period_multiple = 0,
  .safety_crc_hw = true,
  .controls = &System_0_controls[0],
  .modules = &System_0_modules[0],
};
```

Figure 35. nt_system sw configuration

The Kinetis E family of MCUs contains the most advanced TSI v5 peripheral. The module must be configured for a proper operation. However, the NT library helps during the application development, and it is not necessary to deal with the TSI module differences. The TSI hardware setup is displayed below. The “tsi_hw_config” contains register settings used for both Self and Mutual capacitive modes.

If one of these 2 modes is unused by application, the redundant register settings are ignored.

```
const  tsi_config_t tsi_hw_config = {
  .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,
  .configSelfCap.commonConfig.ssc_mode = kTSI_ssc_prbs_method,
  .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,
  .configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_3,
  .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
  .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,
  .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_4,
  .configSelfCap.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,
  .configSelfCap.commonConfig.prbsOutsel = kTSI_SscPrbsOutsel_2,
  .configSelfCap.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,
  .configSelfCap.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,
  .configSelfCap.enableSensitivity = true,
  .configSelfCap.enableShield = false,
  .configSelfCap.xdn = kTSI_SensitivityXdnOption_3,
  .configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,
  .configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,
  .configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0,
  .configMutual.commonConfig.mainClock = kTSI_MainClockSlection_0,
  .configMutual.commonConfig.ssc_mode = kTSI_ssc_prbs_method,
  .configMutual.commonConfig.mode = kTSI_SensingModeSlection_Mutual,
  .configMutual.commonConfig.dvolt = kTSI_DvoltOption_3,
  .configMutual.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
  .configMutual.commonConfig.order = kTSI_SincFilterOrder_2,
  .configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4,
  .configMutual.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,
  .configMutual.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,
  .configMutual.commonConfig.prbsOutsel = kTSI_SscPrbsOutsel_2,
  .configMutual.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,
  .configMutual.preCurrent = kTSI_MutualPreCurrent_4uA,
  .configMutual.preResistor = kTSI_MutualPreResistor_4k,
  .configMutual.senseResistor = kTSI_MutualSenseResistor_10k,
  .configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA,
  .configMutual.txDriveMode = kTSI_MutualTxDriveModeOption_0,
  .configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32,
  .configMutual.pmosRightCurrent = kTSI_MutualPmosCurrentMirrorRight_1,
  .configMutual.enableNmosMirror = true,
  .configMutual.nmosCurrent = kTSI_MutualNmosCurrentMirror_1,
  .thresl = 0,
  .thresh = 65535,
};
```

Figure 36. TSI HW settings in SW

7.5 NXP Touch library memory requirements

Memory requirements depend on the “size of the application”, very basically said on the number of electrode inputs and on the number of controls like keypad keys, sliders, and so on used in the application.

The Touch library was newly written and targeted for the 32-bit Kinetis Arm cortex-M MCUs. In order to use the benefits of the 32-bit architecture, we stored most of the structures, pointers, and constants into flash in 32-bit format. Almost nothing is placed to RAM.

Because of this approach the data in flash is aligned properly and it has a proper size for Arm calculations without any further bit-manipulations required by CPU and without losing of the precision. This allows us to save CPU time during the calculations.

This is also preventing against the issues connected with porting to the different compilers.

The smallest Kinetis-L device had 16 kB Flash, while most of the devices supporting the TSI peripheral have 32 kB or more Flash on the chip. It is possible to fit the touch sensing part of the SW into 32 kB Flash Kinetis devices with reserves for the rest of the application.

See the typical memory requirements in table below for the SW projects passed on the KSDK (including) and FreeMASTER disabled, please.

The most of the Flash (ROM) constants are defined in structures contained in “nt_setup.c“ config. file. By the size and complexity of this file, we can estimate the total memory requirements.

7.5.1 Memory size optimization

The total flash memory required depends on the complexity of the touch application, that is number of electrodes and controls enabled. With the rising complexity, the size of Flash and RAM required rises proportionally. The key detector C-structures can be temporarily placed to RAM, by removing the “const” keyword before the structure definition, see figure 29. The most of the runtime variables used by electrode data, key detectors and by filter calculations are created on the “nt_memory_pool[]“, which is defined as a static RAM array, which size should be adequately selected depending on the application needs. See the nt_memory_pool definition as 4000 byte array. Please note that depending on the compiler, the proper alignment is required for Arm Cortex-M cores. The proper array size may be estimated using the “nt_mem_get_free_size”, which returns the remaining memory after the proper initialization.

```
#if defined(__ICCARM__)
  uint8_t nt_memory_pool[4000]; /* IAR EWARM compiler */
#else
  uint8_t nt_memory_pool[4000] __attribute__((aligned (4)));  /* Keil, GCC compi
#endif
```

Figure 37. nt_memory_pool initialization

```
if ((result = nt_init(&System_0, nt_memory_pool, sizeof(nt_memory_pool))) != NT_SUCCESS)
{
  switch(result)
  {
  case NT_FAILURE:
    nt_printf("\nCannot initialize NXP Touch due to a non-specific error.\n");
    break;
  case NT_OUT_OF_MEMORY:
    nt_printf("\nCannot initialize NXP Touch due to a lack of free memory.\n");
    break;
  }
  while(1); /* add code to handle this error */
}

nt_printf("\nNXP Touch is successfully initialized.\n");
nt_printf("Unused memory: %d bytes \n", (int)nt_mem_get_free_size());
```

Figure 38. Nt_memory_pool size

7.5.2 Removing the FreeMASTER

By default FreeMASTER is enabled in the touch demo application, which is needed to run the FreeMASTER GUI during the development stage. FreeMASTER itself consumes some resources, so it is recommended to remove it from the final SW project as soon as the touch sensing tuning is completed. The FreeMASTER support can be globally disabled in the touch application by the definition “NT_FREEMASTER_SUPPORT 0”, see the picture below.

```
#ifndef NT_FREEMASTER_SUPPORT
  #define NT_FREEMASTER_SUPPORT 0
#endif
```

Figure 39. Disable FreeMASTER support in the touch library

Then the rest of the FreeMASTER definitions and references like the TSA table, “init_freemaster_uart()” and “FMSTR_Init()” functions must be removed from the SW project, see the figure below, please.

```
/*
 * This list describes all TSA tables that should be exported to the
 * FreeMASTER application.
 */
//#ifndef FMSTR_PE_USED
//  FMSTR_TSA_TABLE_LIST_BEGIN()
//      FMSTR_TSA_TABLE(nt_frmstr_tsa_table)
//  FMSTR_TSA_TABLE_LIST_END()
//#endif



 /* FreeMASTER communicates over the default UART instance */
 //init_freemaster_uart();

 /* FreeMASTER initialization */
 //(void)FMSTR_Init();

  while(1)
  {
    nt_task();

    //FMSTR_Poll();
  }
```

Figure 40. Remove FreeMASTER references

Table 1. NXP Touch typical memory requirements (FreeMASTER removed)

App. size	2 electrodes	28 electrodes
FLASH [kB]	14	28
SRAM [kB]	2.2	7.2

8 Key detector uSAFA

Three key detectors are supported by NXP Touch Library:

- AFID (patented, easy CPU calculations)
- SAFA – Self-Adaptive Filter Algorithm (patented)
- uSAFA – “unidirectional” SAFA (recommended)

SAFA is the most advanced algorithm used.

SAFA means that the Signal Adaptive Filter Algorithm is a filtering SW algorithm patented by NXP, where “u” means unidirectional. It is based on moving average filter with different weight. Noise level (dead-band) limitation is used together with noise level tracking and automatic threshold adaptation. Typical touch signal level is tracked and used as “predicted” touch signal value.

8.1 Key detector uSAFA signals

Following signals are the most important:

- **Baseline** is the basic reference signal, which moves very slowly with the time and environmental changes. All of the other signals are referenced to the baseline.
- **Signal level**, is an elementary filtered raw touch sensing signal (SW low-pass filter used). If you touch, this signal reflects the change.
- Noise floor (**min noise limit**), we expect that the ambient system noise will not cross this value in the normal environment (no added EMC noise). It means that normally the system noise is much below this value. We have set it to **100** in SW experimentally, while the real value may be much lower.
- **Deadband**, is the noise level, which must be crossed by signal to detect the Touch or Release condition. Deadband is defined as (min_noise_limit x SNR) in this case. For instance, when SNR = 6. So that **Deadband = 100 * 6 = 600 counts**.
- **Predicted signal**, is the typical signal level, when the button is touched. We adapt this value, upon a touch or release.
- For the **TOUCH** event, the signal must rise above the 25% of the Predicted signal and it crosses the Deadband level.
- For the **RELEASE** event, the signal must drop below the 80% of the Predicted signal.
- Event counters (“entry_event_cnt” and “deadband_cnt”) are used for debouncing the Touch and Release Events

```
/* SAFA keydetector settings */
const struct nt_keydetector_usafa  keydec_usafa =
{
/* Electrodes */
    .signal_filter = {2},
    .base_avrg  = {.n2_order  = 10},
    .non_activity_avrg  =  {.n2_order  = NT_FILTER_MOVING_AVERAGE_MAX_ORDER},
    .entry_event_cnt  = 4,
    .deadband_cnt  = 4,
    .signal_to_noise_ratio = 6,
    .min_noise_limit = 100,
    .dc_track_enabled = 1,
    .dc_track_cnt = 100,
};
```

Figure 41. uSAFA deadband threshold setting

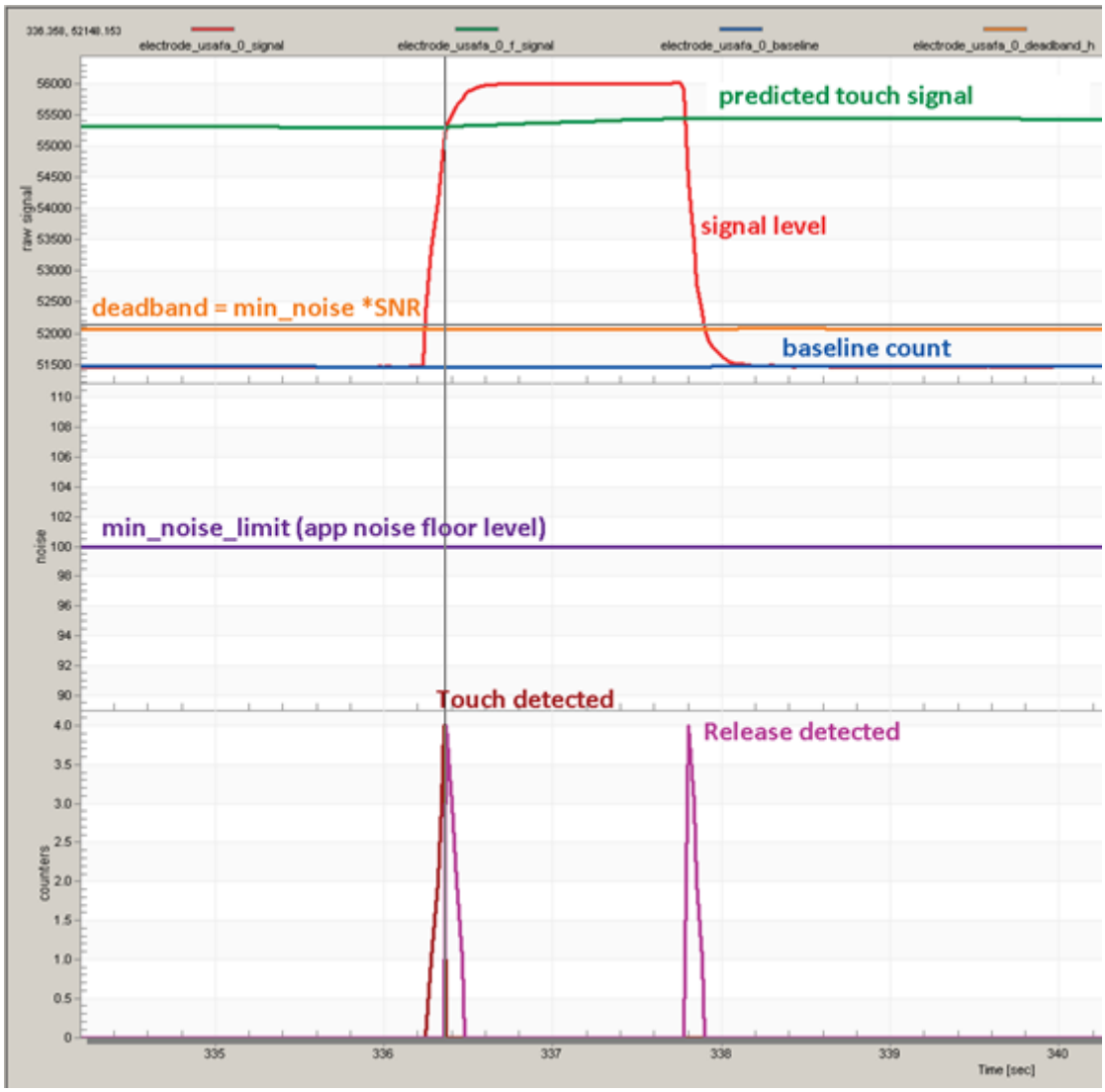


Figure 42. Key detector uSAFA signals

8.2 Key detector uSAFA filter parameters

All filters are based on moving average SW filtering, where the order is taken as power of two (2^n).

For instance, order = 10 means, 210 = 1024 samples will be averaged.

It means if we increase to “11”, 2048 samples will be averaged, and the filtering will be 2-times slower.

Oppositely, if we decrease to “9” or “8”, it will be 2 or 4-times faster. So that by the changing the order of the filters, we can control the response and adaptation speed. All filters and debouncing counters are depending on the “time_period” parameter which is equal with the TSI scan period.

See the figure below with comments describing the parameters:

```
/* SAFA keydetector settings */
const struct nt_keydetector_usafa keydec_usafa =
{
.signal_filter = {2},                  // Coefficient of the input IIR signal filter, used to suppress high-frequency noise.
    .base_avrg = {.n2_order = 10}, // Settings of the moving average filter for the baseline in the release state of an electrode.
    .non_activity_avrg =  {.n2_order = 15}, // Settings of the moving average filter for the signals in the inactivity state of an electrode. (for example baseline in a touch state).
    .entry_event_cnt = 4, // Sample count for the touch event. This means that this count of samples must meet the touch condition to trigger a real touch event.
    .deadband_cnt = 4,    // Sample count for the deadband filter. This field specifies the number of samples that cannot proceed to the next event.
    .signal_to_noise_ratio = 6, // Signal-to-noise ratio – it is used for counting the minimum size of the signal that is ignored
    .min_noise_limit = 100,      // Minimal system noise floor level
    .dc_track_enabled = 1,       // When the DC-tracker is enabled, then the Keydetector resets, when there is a significant signal drop below the baseline detected
    .dc_track_cnt = 100,         // Number of samples requested (length of the negative signal drop) after that the DC tracker resets the keydetector.
};

const struct nt_system system_0 = {
    .controls = &controls[0],
    .modules = &modules[0],
    .time_period = 50, // Time base [ms] used for sampling the electrodes (HW timer trigger firing the sequence)  and to measure DC tracker time-out.
    .init_time = 400,  // Application warm-up delay period [ms], while the signals can settle but aren't evaluated
};

LPIT_SetTimerPeriod(LPIT0, kLPIT_Chnl_0, 50*CLOCK_GetFreq(kCLOCK_ScgSircAsyncDiv2Clk)/1000); // Configure HW timer period in "main.c" properly
```

Figure 43. Key detector uSAFA parameters

8.3 DC tracker feature

DC-tracker helps to adapt to the special situation, when the signal suddenly drops much below the baseline. This can happen in the real-word application.

For instance, when there is some object present on the sensor electrode during a power-up, and it is removed after a while, it must recover and adapt thresholds to the new situation to be able to detect the “regular” touches.

The DC tracker resets the key detector when the signal drop is higher than (2 * min_noise_limit).

The DC tracker reaction time is configurable and given by multiple: **dc_track_cnt * time_period**.

In the example below, the key detector is rests after the timeout of 5000 ms.

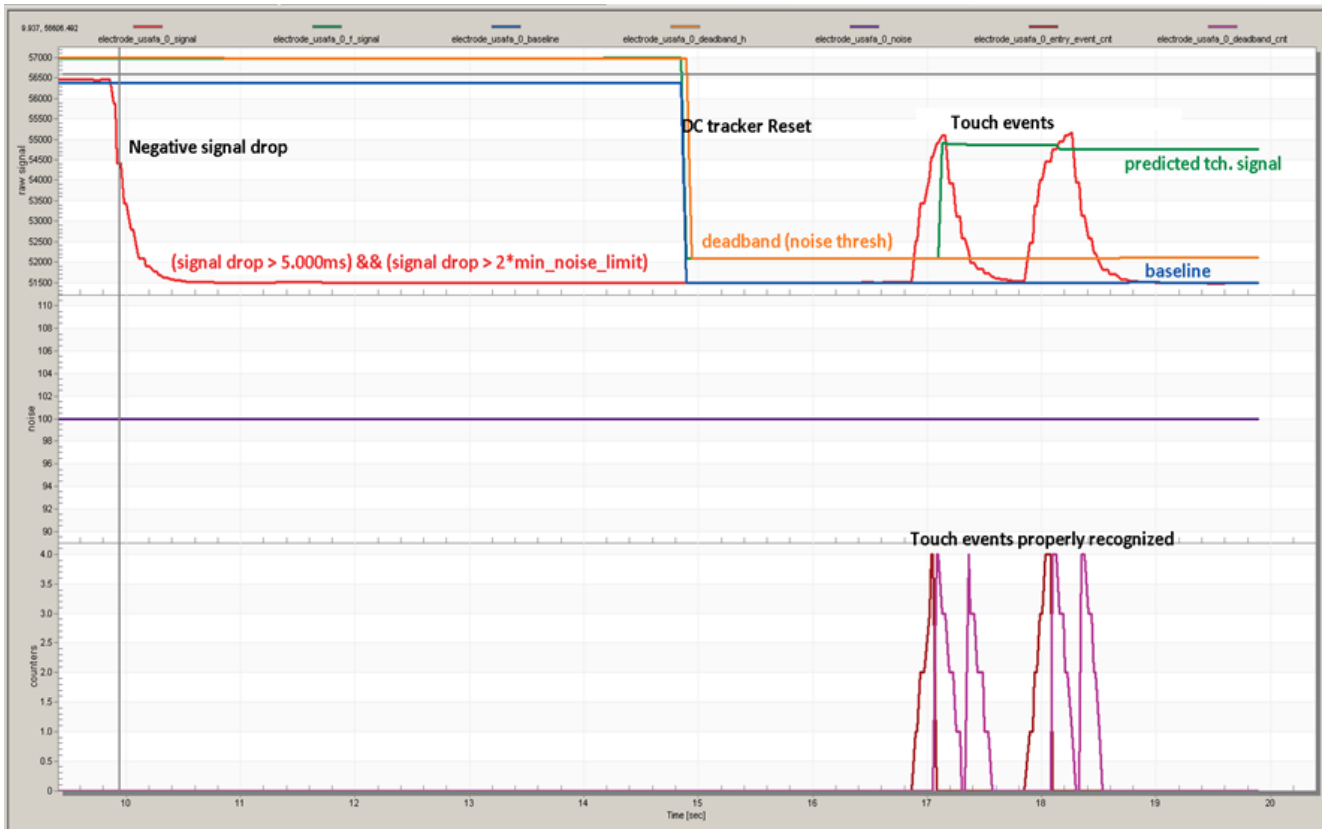


Figure 44. DC tracker reaction

8.4 Key detector uSAFA tuning

By changing of the key detector parameters we can tune the touch sensitive threshold, change the noise adaptation speed and robustness against the external influences.

If we change the SNR from 6 to 15, the Deadband threshold will change from 600 to 1500.

- Delta signal values < min_noise_limit (100) are ignored, taken as noise floor.
- Delta signal values > min_noise_limit (100), but < Deadband taken as increased system noise and used for the automatic deadband threshold adaptation
- Delta values > Deadband are handled as a 1st Touch event condition.
- 2nd Touch event condition is to cross the 25% of the predicted touch signal
- Both conditions must be passed to trigger the Touch event counter.
- If the number of Touch events >= “entry_event_cnt” (used to debounce the glitches), then the TOUCH is evaluated by SW.

```
const struct nt_keydetector_usafa keydec_usafa =
{
.signal_filter = {2},
    .base_avrg = {.n2_order = 10},
    .non_activity_avrg =  {.n2_order = 15},
    .entry_event_cnt = 4,
    .deadband_cnt = 4,
    .signal_to_noise_ratio = 15, // 6
    .min_noise_limit = 100,
    .dc_track_enabled = 1,
    .dc_track_cnt = 100,
};
```

Figure 45. uSAFA key detector settings

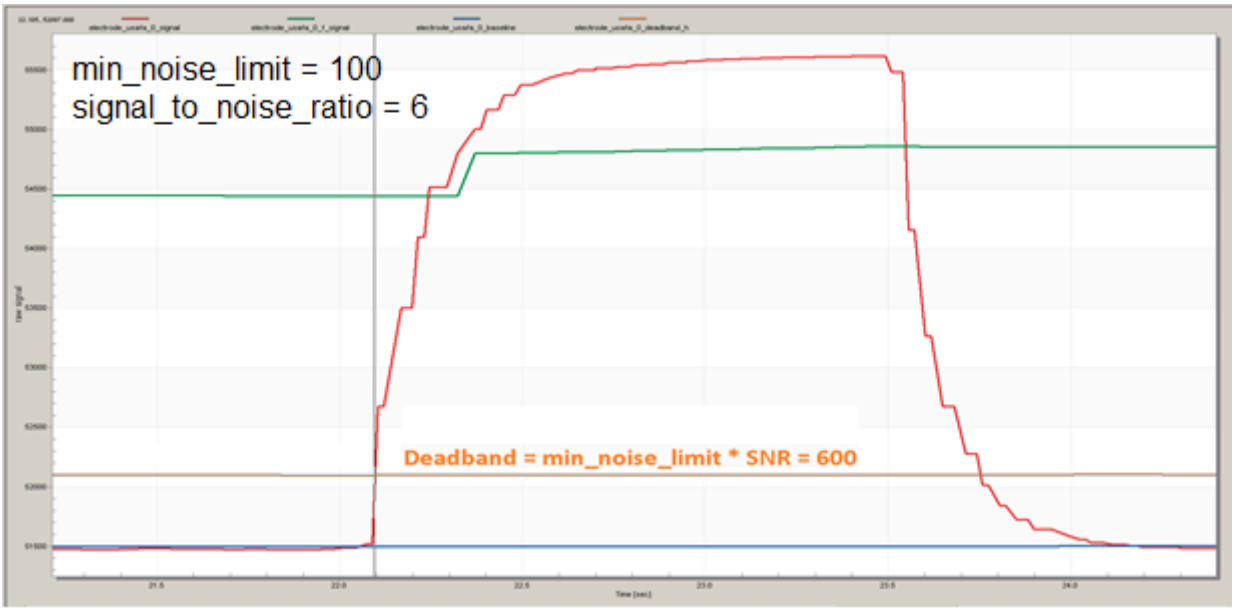


Figure 46. uSAFA tuning SNR = 6

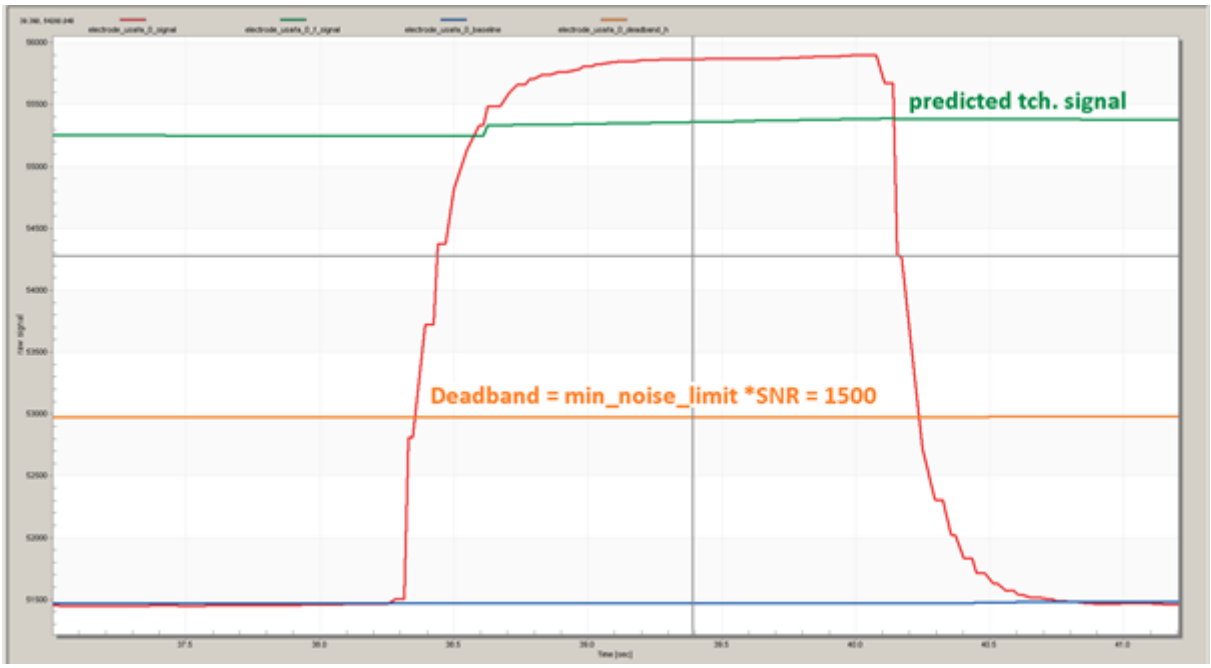


Figure 47. uSAFA tuning SNR = 15

8.4.1 Noise level adaptation

Noise detection and noise level adaptation is used to overcome the harsh environment or to pass the EMC immunity tests on the EMC bench.

When the noise signal crosses the “min_noise_level” (100), then the noise level is accumulated and increased, resulting to the Deadband update given be: “noise_level * SNR”

For instance, if the noise_level rises to 130, then the Deadband is updated from (100*15) to (130*15) so that the threshold rises to 130% to adapt to the increased noise conditions.

When the noise level decreases, it will return automatically down to “min_noise_level” (100) after a while. The speed of the noise level adaptation and recovering can be controlled by the order of the SW filter “base_avrg”.

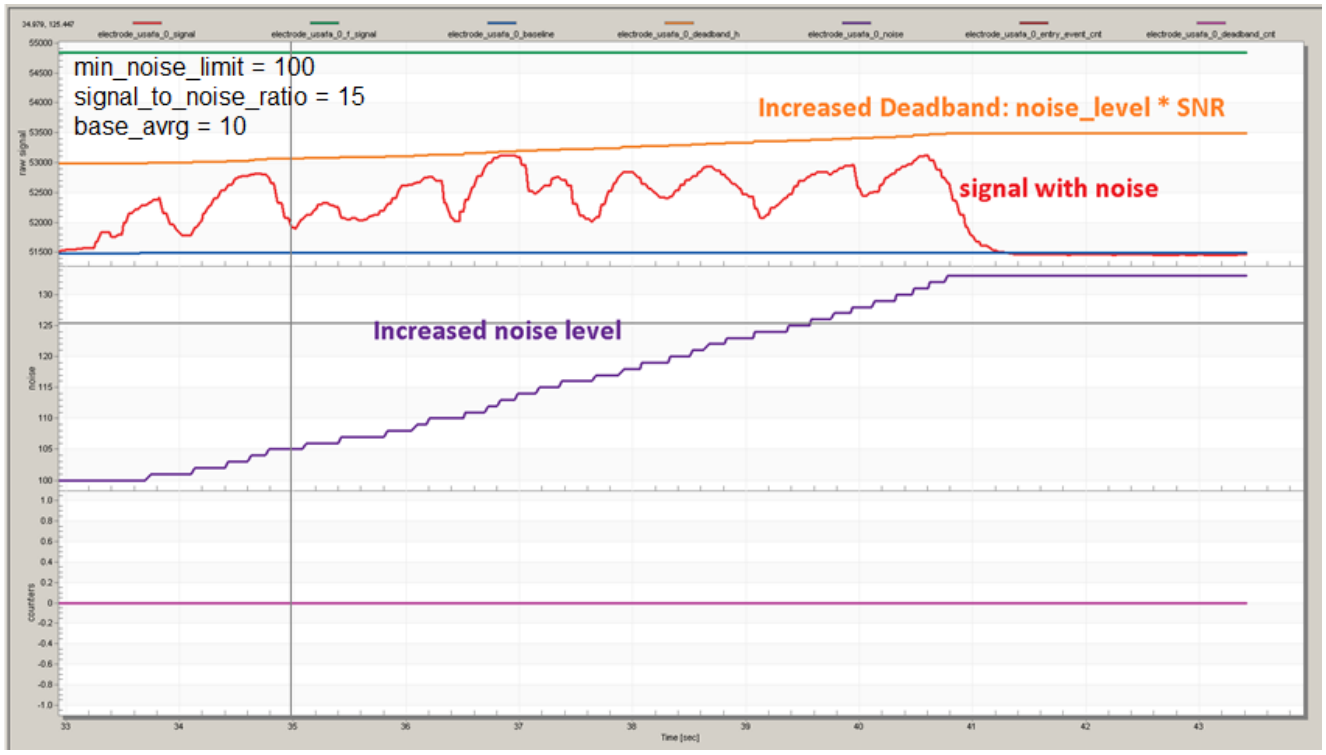


Figure 48. Noise level adaptation

If we change the “base_avrg” filter order from 10 to 8, the noise level adaptation will be 4x faster than before, so that it can update the deadband threshold much faster and react to the increased noise. Note that the noise level recovers automatically, when the noise signal is gone. The noise level recovery speed is hardcoded to be 16x faster than the noise accumulation, which is suitable for most cases, but it can be accelerated in the SW if desired.

```
const struct nt_keydetector_usafa keydec_usafa =
{
.signal_filter = {2},
  .base_avrg = {.n2_order = 8}, //10
   .non_activity_avrg =  {.n2_order = 15},
   .entry_event_cnt = 4,
   .deadband_cnt = 4,
   .signal_to_noise_ratio = 15, // 6
   .min_noise_limit = 100,
   .dc_track_enabled = 1,
   .dc_track_cnt = 100,
};
```

Figure 49. uSAFA base filter tuning for noise adaptation

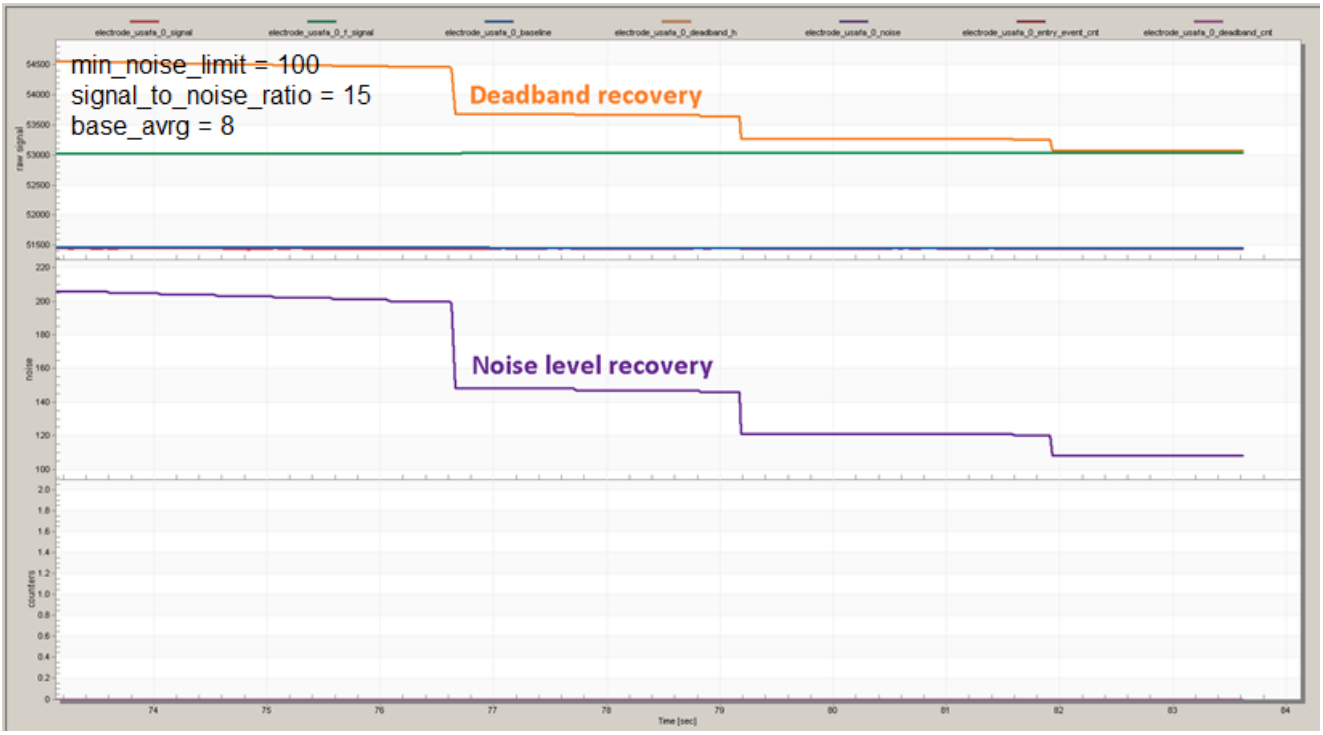


Figure 50. Noise level recovery

9 TSI module HW introduction

9.1 TSI v5 main features

- Support both of Self-cap sensor and Mutual-cap sensor
- Enhanced noise immunity to support system EMC standard Test
- Enhanced sensitivity to support wide range of overlay thickness
- Capability to wake MCU from stop2 and low-power modes
- Fully support NXP touch sensing software (NT) library
- Support DMA data transfer

9.2 TSI methods

Touch sensing interface (TSI) provides touch sensing detection on capacitive touch sensors. The external capacitive touch sensor is typically formed on PCB and the sensor electrodes are connected to TSI input channels through the I/O pins in the device.

Two different touch sensing methods are supported, the self-capacitive mode and the mutual-capacitive mode.

KE15z MCU supports up to 25 inputs in Self-Capacitive mode and up to 6x6 inputs can be implemented in the Mutual mode. Both mentioned methods can be combined on single PCB, while only the lower 12 TSI channels TSI[0:11] can be used for Mutual mode. Please note that TSI[0:5] are TSI TX pins and TSI[6:11] are TSI RX pins in Mutual mode.

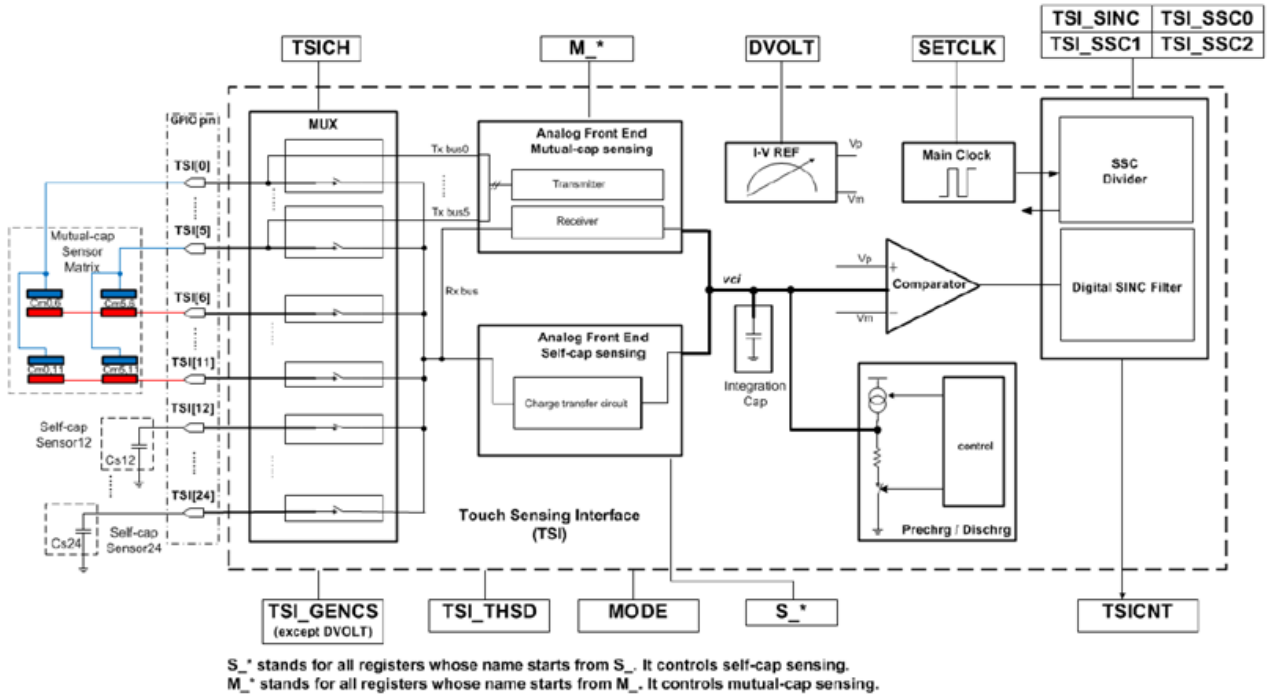


Figure 51. Kinetis KE1x TSI v5 block diagram

9.3 Self vs. Mutual capacitance

E-field distribution between self-cap sensor and mutual-cap sensor is different.

For self-cap, capacitance exists between electrode to system ground. Touch changes field through human body and creates extra capacitance.

For mutual-cap, sense capacitance exists between 2 electrodes. Touch changes field through human body and reduces the mutual capacitance. TSI IP is to convert the capacitance changing from the sensor to digital code for application.

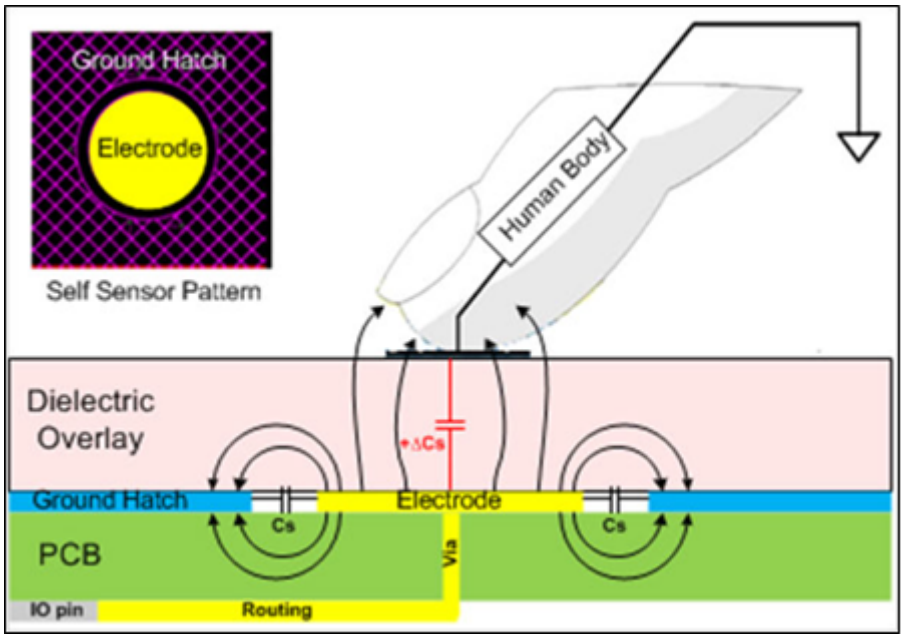


Figure 52. Self-capacitance principle

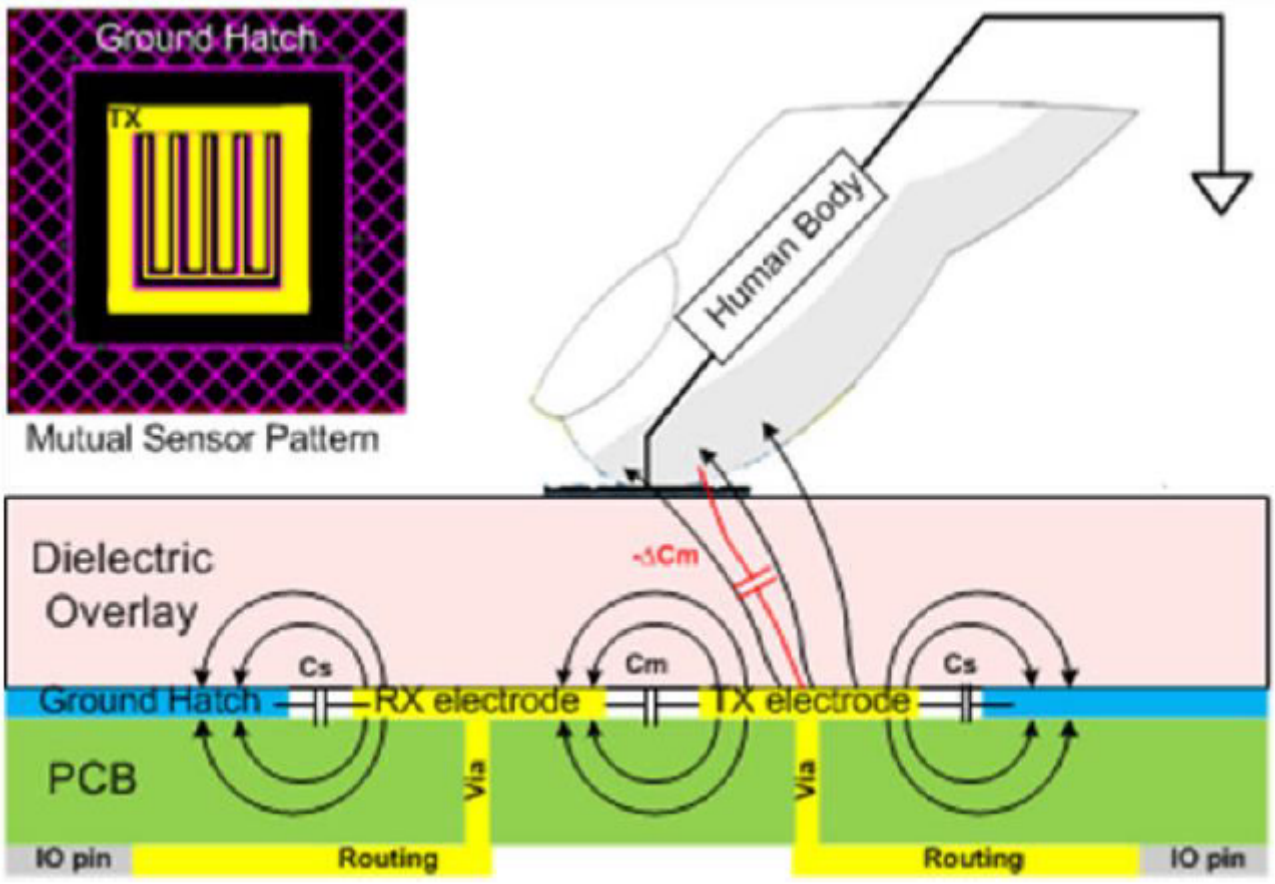


Figure 53. Mutual-capacitance principle

9.4 Self-cap. HW architecture

Charge transfer method (which has intrinsic noise immunity) is used to detect touch event. One sampling clock including non-overlapping ph1 (sample phase) and ph2 (transfer phase) is controlled to charge electrode capacitor and transfer the charge to internal integration capacitor through Charge Transfer Circuit (CTC). Stepped saw tooth generates at node of Vci. Vci is detected by comparator, when it surpasses positive reference Vp, Ci will be discharged to negative reference Vm. Then next scanning cycle continues. When touch happens, input capacitance will increase and then the number of saw tooth ramp up steps is reduced. The difference of the number is detected by digital filter. Digital filter suppresses the noise of number and outputs counts which can be used by software to detect touch.

NXP Touch SW library does the touch signal inversion in the SW.

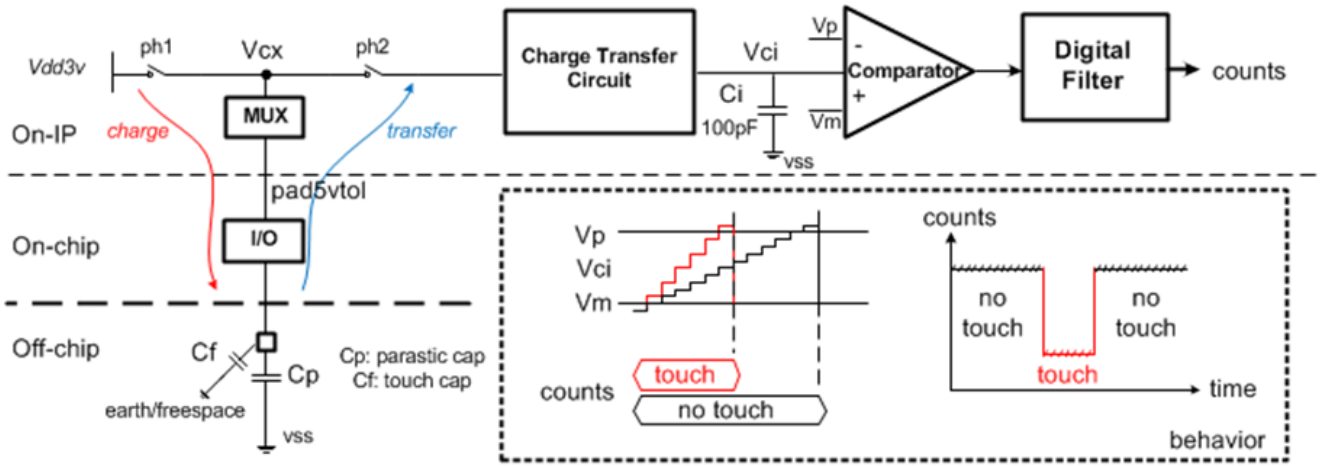


Figure 54. Self-cap sensing HW architecture

9.5 Mutual-cap. HW architecture

Mutual-cap sensing includes transmitter and receiver. Under clocking, transmitter outputs pulses which decouple through mutual cap then reach receiver site. Receiver amplifies the signal with noise cancellation method which is similar as charge transfer circuit in self-cap sensing.

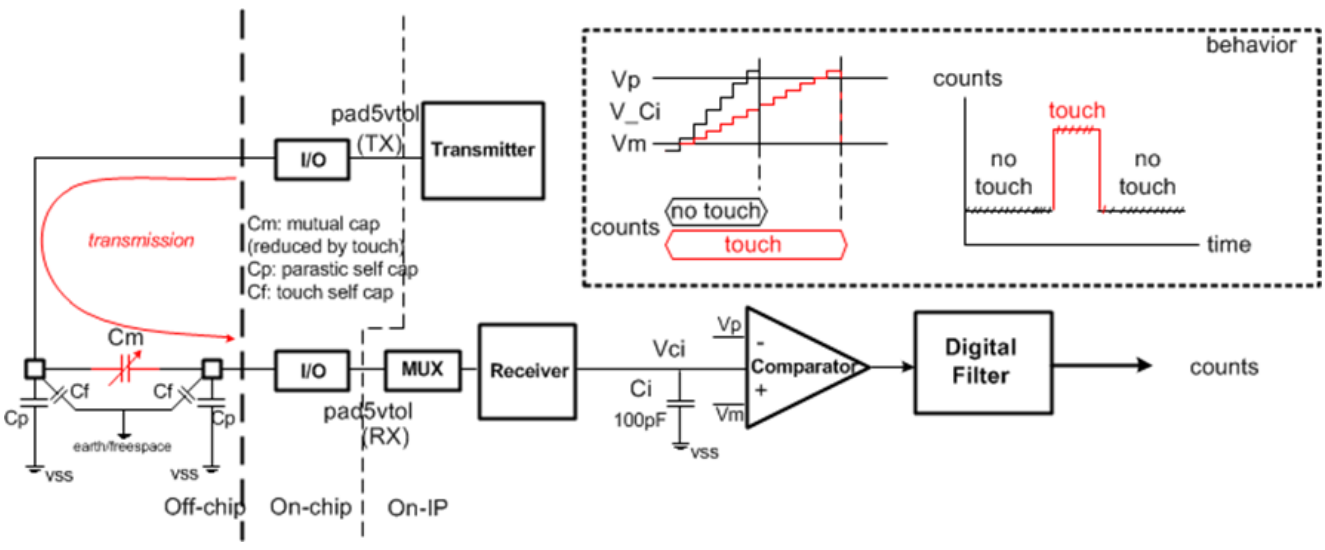


Figure 55. Mutual-cap sensing HW architecture

9.6 Understanding the TSI Measurement

9.6.1 Self-capacitance mode in details

Inside the TSI IP module, the TSI scan is operated by non-overlapping clock ph1/ph2 and trans-conductance amplifier. There are two phases controlled by the ph1 and ph2 respectively for the TSI scan module:

- Sample phase: The switch ph1 controls the sample phase, when ph1 turns on, the external touch electrode Cx is charged by vdd3v.
- Charge phase: The switch ph2 controls the charge phase, when ph1 turns off then ph2 turns on, the charge on the capacitor Cx flows to the internal integrated capacitor Ci, which generates the average current Icx.

Via the trans-conductance amplifier which consists of two current mirrors, the Icx are also be amplified according to the input and charge current mirror setting. The final average current to charge the integrated Ci equals to Xch*Xin*Icx. As the integrated Ci is charged by the average current, the voltage Vci ramps on Ci, when the Vci becomes larger than the pre-setting Vp, the comparator will stop this TSI scan round and the digital filter will record the sample result as TSICNT.

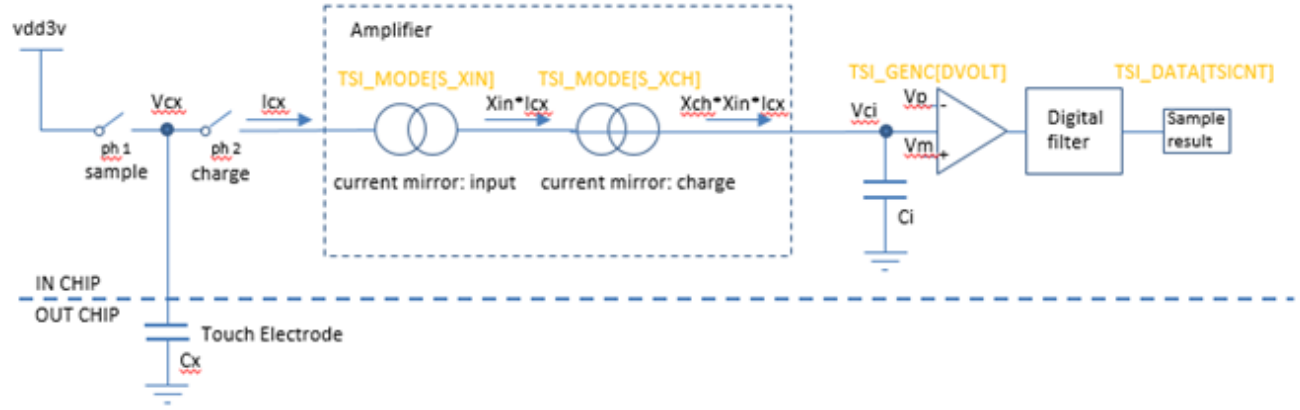


Figure 56. Self-Capacitive Block

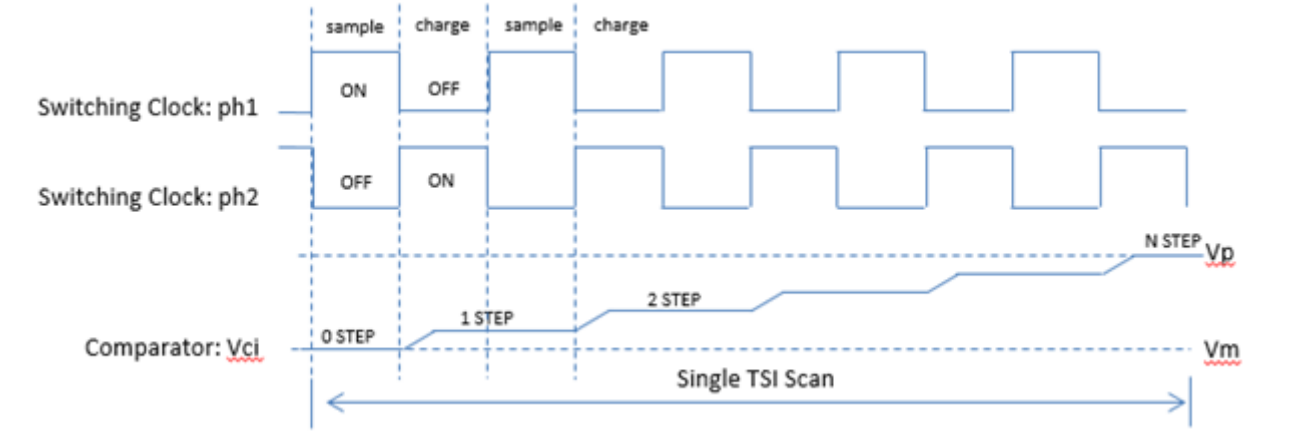


Figure 57. Self-Capacitive Mode Timing

9.6.2 Mutual-capacitance mode in details

The mutual capacitive mode measures the capacitance between two electrodes connected to two TSI channels. One of the TSI channels is used as a transmit (TX) channel and the other as a receive (RX) channel.

There are two phases controlled by the switching clock for the TSI mutual capacitive mode:

- Charge phase: The switch ph1 controls the charge phase, when ph1 turns on, the transmit channel outputs pulses which is coupled through the mutual capacitance Cm. Receiver converts the received voltage pulse (Vpre+ΔV) to the current Icharge through the resistor Rs.
- Discharge phase: The switch ph2 controls the discharge phase, when ph1 turns off then ph2 turns on, the transmit channel changes the voltage from Vdd5v to 0V. Receiver converts the received voltage change (Vpre-ΔV) to the current Idischarge through Rs.

As the integrated Ci is charged/discharged by the mirrored/amplified current from the receiver, the voltage Vci ramps on Ci, when Vci becomes larger than the pre-setting Vp, the comparator will stop this TSI scan round and the digital filter will record the sample result as TSICNT.

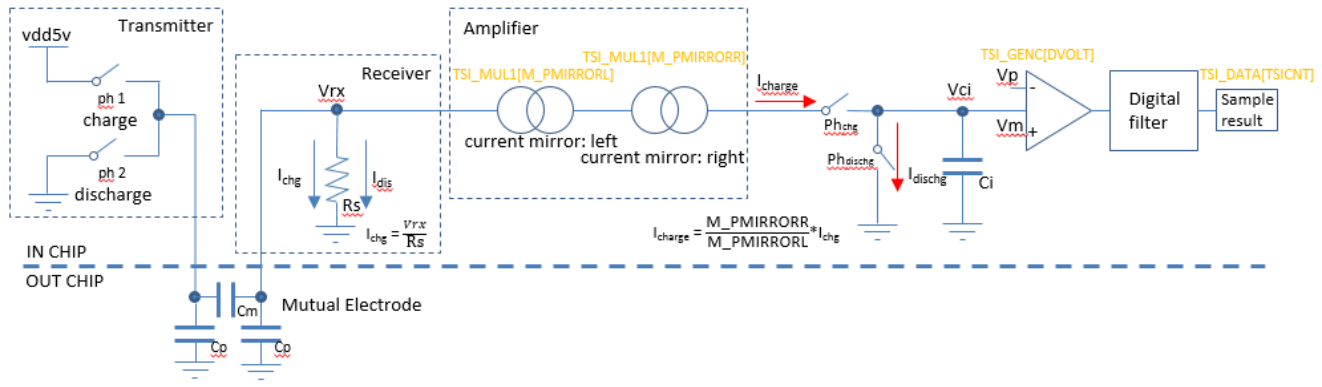


Figure 58. Mutual Capacitive Block

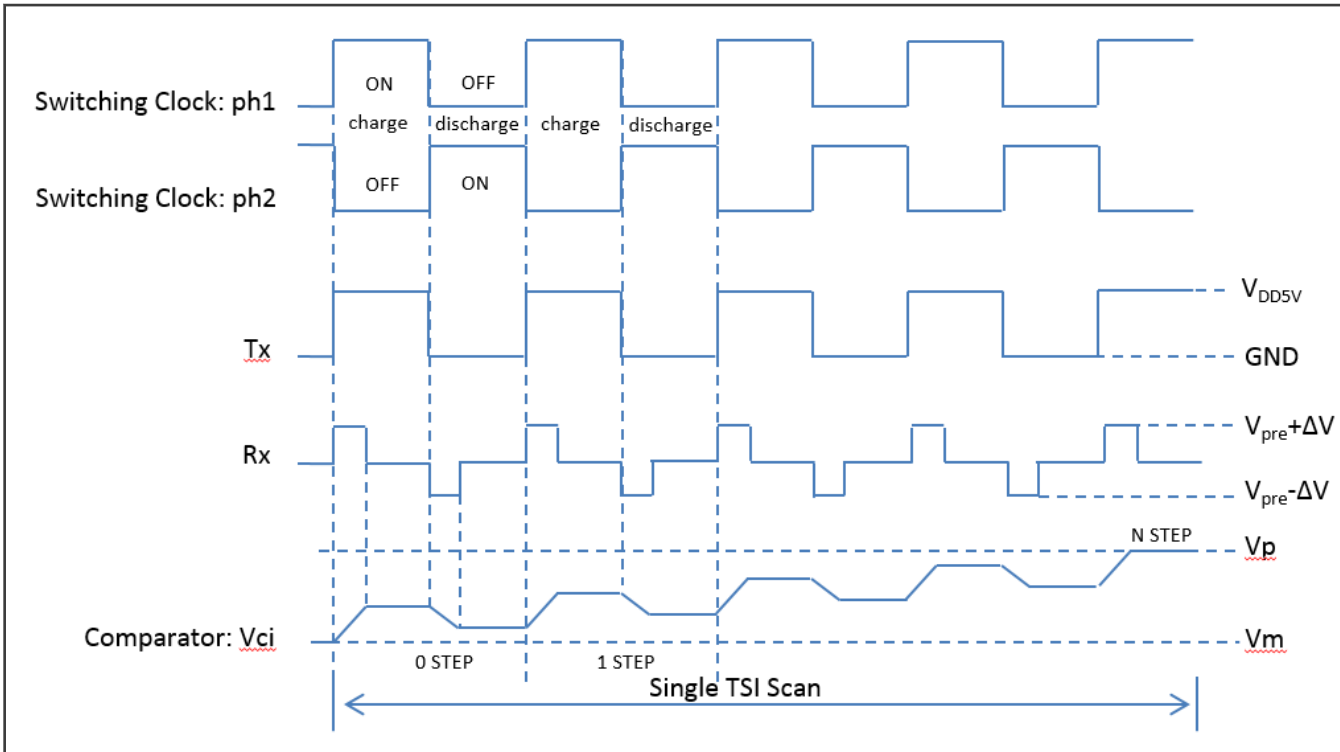


Figure 59. Mutual Capacitive Mode Timing

NOTE

Due HW limitation on the early MKE15z256 silicon, if TSI is operated in the Mutual mode, then TSI channels 0 to 5 are reserved for TSI functionality and cannot be used for other purposes like GPIO. This must be considered during designing the board. This HW limitation has been fixed on the later KE16x devices.

9.7 TSI IP HW register Tuning – Self Cap mode

The HW register settings is available in SW structure: “hw_config” located in file: “nt_setup.c”.

The structure includes set of tunable parameters for both self-cap. and mutual-cap. modes.

In the self-cap. mode the basic mode is with the sensitivity boost disabled. With the sensitivity boost enabled, the tuning becomes trickier, because the more parameters come into play.

9.7.1 Sensitivity in Self Cap with Boost Disabled

In this mode the tuning is easier, because of a few parameters, which are important for sensitivity and length of accumulation. See the picture below for more details about the important parameters highlighted in green color.

NOTE

NSTEP is the result of TSI single scan, Decimation is the factor responsible for multiple scan result accumulation.

```
/* Self-cap and mutual-cap mode config */
const tsi_config_t hw_config =
{
    /* Self capacitance measurement config */
    configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,       // Set main clock
    configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,    // Choose SelfCap sensing mode
    configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_0,                 // DVOLT (Vp-Vm)
    configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,              // Cutoff divider
    configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,             // SINC filter order
    configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_4,      // SINC decimation value
    configSelfCap.enableSensitivity = false,                               // SENS_BOOST = OFF
    configSelfCap.xdn = kTSI_SensitivityXdnOption_3,                       // Sens S_XDN
    configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,                   // Sens S_CTRIM
    configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,         // Sens S_XIN = 1/8
    configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0,     // Sens S_XCH
```

$$NSTEP = \frac{Ci \times (Vp - Vm)}{vdd3v \times Cs \times S\_XIN \times S\_XCH}$$

SDK File: “**fsl_tsi_v5.h**”

```
typedef enum _tsi_current_multiple_input
{
  kTSI_CurrentMultipleInputValue_0 = 0U,  /*!<  S_XIN = 1/8 */
  kTSI_CurrentMultipleInputValue_1 = 1U,  /*!<  S_XIN = 1/4 */
} tsi_current_multiple_input_t;
```

Figure 60. TSI register tuning in Self-Cap mode, Sensitivity Boost = OFF

9.7.2 Sensitivity in Self Cap with Boost Enabled

Enabling Sensitivity Boost feature can increase sensitivity by removing part of parasitic capacitance “virtually”. So touch can work well under the thick overlay with sensitivity boost enabled. The TSI self-capacitive mode implements the sensitivity boost by canceling the external intrinsic capacitance, and the value of the capacitance to be canceled ranges from 2.5pF to 20pF, configurable in register TSI_MODE[S_CTRIM].

For example, given the intrinsic capacitance of the touch electrode is 20pF(it can be calculated by NSTEP formula), setting the S_CTRIM value as 5.0pF can make the effective intrinsic capacitance become 15pF. As the intrinsic sensitivity of the touch key is given by ΔCs/Cs, The less intrinsic capacitance would result in more sensitive touch response.With this sensitivity boost enabled, sensitivity can be improved to ΔCs/(Cs-S_CTRIM*(S_XDN/S_XCH)).

The figure below shows the block diagram of TSI self-capacitive mode with sensitivity boost enabled. The sensitivity boost module generates the average current Ictrim by the similar sample/charge on a configured internal capacitor Ctrim. The final average current to charge the Ci will be the orignal Icx substract the Ictrim. As a result, the capacitance of the external touch electrode seems substracted by the Ctrim. By the way, the actual Ctrim substracted equals to (Xdn/Xch)*Ctrim.

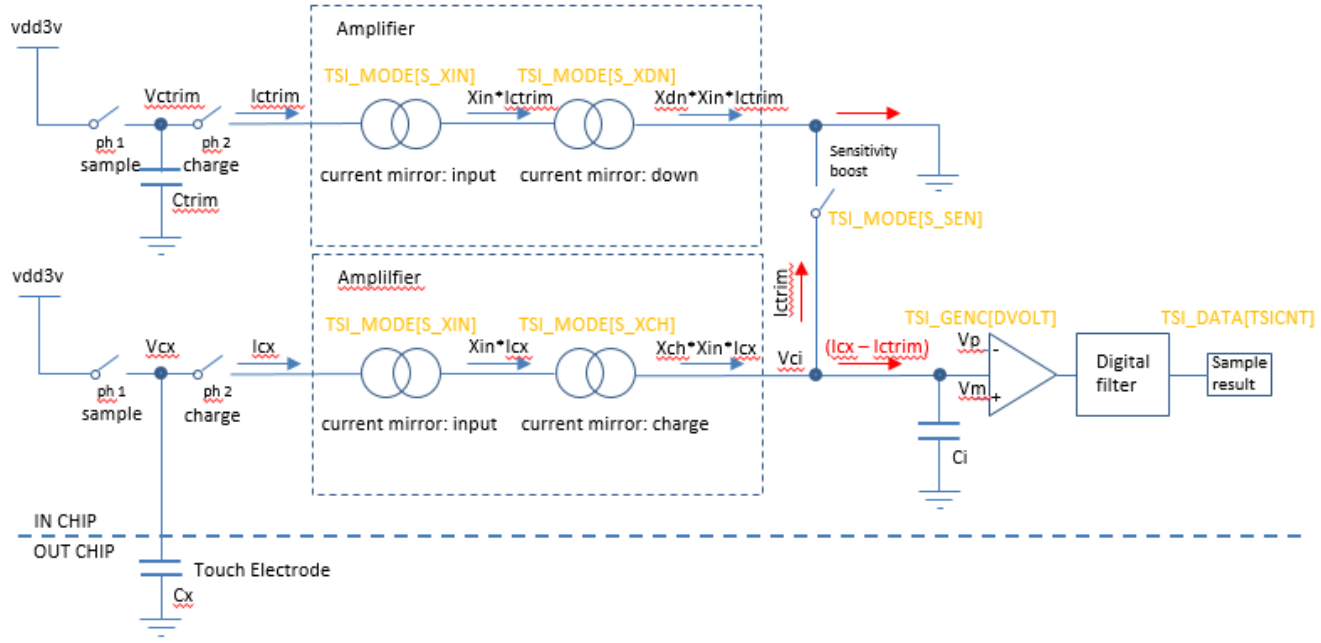


Figure 61. TSI block in Self-Cap mode, Sensitivity Boost = ON

```
/* Self-cap and mutual-cap mode config */
const tsi_config_t hw_config =
{
   /* Self capacitance measurement config */
   .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,      // Set main clock
   .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,    // Choose SelfCap sensing mode
   .configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_0,                // DVOLT (Vp-Vm)
   .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,             // Cutoff divider
   .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,            // SINC filter order
   .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_4,     // SINC decimation value
   .configSelfCap.enableSensitivity = true,                              // SENS_BOOST = ON
   .configSelfCap.xdn = kTSI_SensitivityXdnOption_3,                      // S_XDN
   .configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,                  // S_CTRIM
   .configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,       // S_XIN
   .configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0, // S_XCH
```

$$NSTEP = \frac{Ci \times (Vp - Vm)}{vdd3v \times (Cs - S\_CTRIM \times (S\_XDN \div S\_XCH)) \times S\_XIN \times S\_XCH}$$

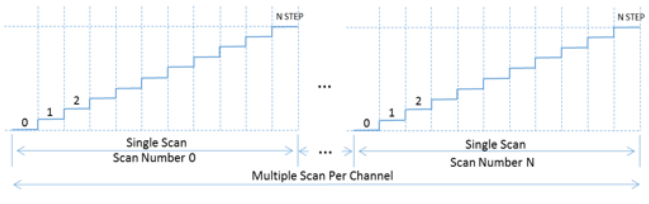
S_CTRIM: configurable, the capacitance to be removed.
S_XDN/S_XCH: configurable, the capacitance multiplier.
The actual capacitance to be removed is : $S\_CTRIM \times (S\_XDN \div S\_XCH)$

Figure 62. Sensitivity tuning in Self cap mode with Boost On

9.7.3 TSI Scan time and result accumulation

The scan time determines the size and time of the conversion result.

TSI supports multiple scan per channel, which means TSI performs multiple scans in order to get better SNR and resolution. The final scan result will be accumulated in TSI_DATA[TSICNT] counter as the NSTEP multiplied by number of scans, and the scan time will multiple of single TSI scan time. Please note that with higher Decimation, the number of scans is increased, which results to the physically longer TSI counter accumulation and increased resolution. Please note that if the Order is higher than 1, then the scan number physically executed by TSI is smaller than the scan number calculated by HW, which may be beneficial to get the higher resolution.



$$ScanNumber_{calculated} = \frac{Decimation^{Order}}{Cutoff} \qquad ScanNumber_{executed} = \frac{Decimation \times Order}{Cutoff}$$

Figure 63. TSI scan time and accumulation

The parameters of **Decimation**, **Order** and **Cutoff** affects the final accumulated scan result and total scan time as well. Setting the Order as 2 is recommended as it can save scan time to achieve the same digital scan result.

$$\text{ScanResult} = NSTEP \times ScanNumber_{calculated} = \frac{NSTEP \times Decimation^{Order}}{Cutoff}$$

$$\text{ScanTime} = Tnstep \times ScanNumber_{executed} = \frac{Tnstep \times Decimation \times Order}{Cutoff}$$

Figure 64. Scan Result vs. Scan Time formulas

```
/* Self-cap and mutual-cap mode config */
const tsi_config_t hw_config =
{
   /* Self capacitance measurement config */
   .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,    // Choose SelfCap sensing mode
   .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,             // Cutoff divider  = 1
   .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,            // ORDER = 2
   .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_8,   // DECIMATION = 8
```

ScanNumberCalc (NSTEP multiple) = (8^2) / 1 = 64

ScanNumberExec (TSI hw really performed scans) = (8*2)/1 = 16

Figure 65. Decimation and Cutoff settings

9.8 Clock Generation and Spread Spectrum Clocking

TSI clock can be derived from the selectable, asynchronous internal clock reference “Main Clock”, which can be furtherly divided to get the final TSI scan clock frequency.

Basic and Advanced (SSC) clock modes are available as the clock options.

SSC (Spread Spectrum Modulated Clock) may be beneficial for higher EMC immunity and reduce the EMI.

- Basic: When SSC_MODE=10b, then the switching clock is divided from main clock directly, as the basic clock generation.
- Advanced (SSC): When SSC_MODE=00b/01b, then the switching clock is generated from SSC module, as the advanced clock generation.

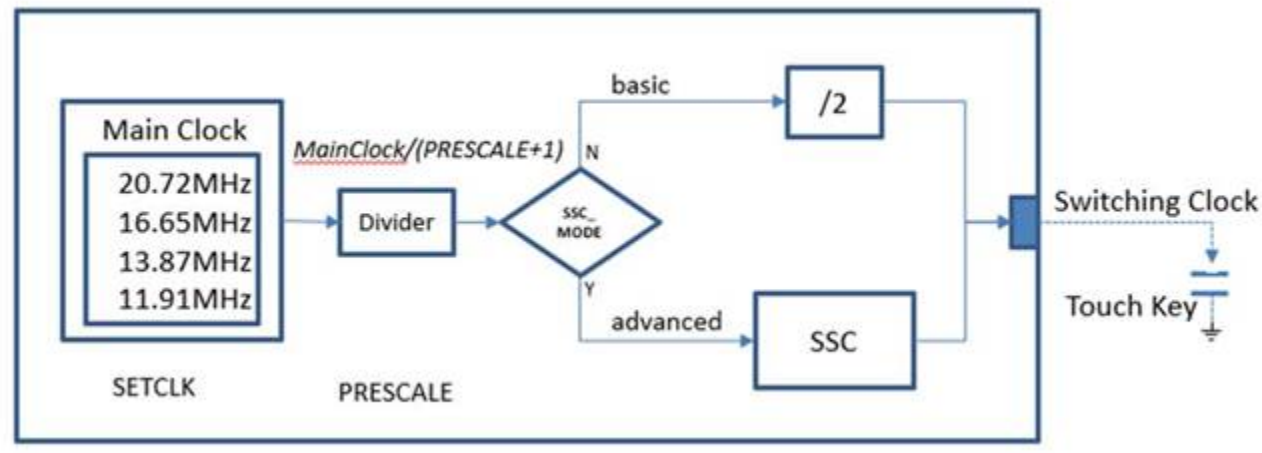


Figure 66. TSI v5 clock generation block diagram

- If SSC is disabled:

The TSI Switching Clock = TSI_MainClock / (*SSC_PRESCALE_NUM+1)* / 2

- If SSC is enabled:

The TSI Switching Clock = TSI_MainClock / (*SSC_PRESCALE_NUM+1)* / ((BASE_NOCHARGE_NUM+1)+ (PRBS_OUTSEL+1)/2+(CHARGE_NUM+ 1))

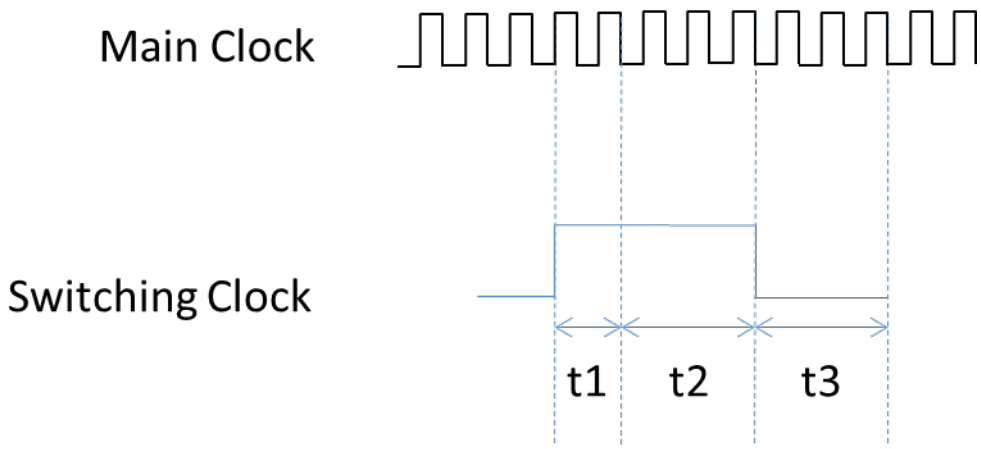


Figure 67. SSC clock principle

Variable	Register	Descriptions
t1	**TSI_SSC0[BASE_NOCHARGE_NUM]**	**SSCHighWidth**
t2	**TSI_SSC0[PRBS_OUTSEL]**	**SSCHighRandomWidth**
t3	**TSI_SSC0[CHARGE_NUM]**	**SSCLowWidth**

Figure 68. PRBS random clock generation

```
/* Self-cap and mutual-cap mode config */
const tsi_config_t hw_config =
{
    /* Self capacitance measurement config */
    .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,        // Set main clock
    .configSelfCap.commonConfig.ssc_mode = kTSI_ssc_prbs_method,             // Select the SSC modulated clock output
    .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,    // Choose SelfCap sensing mode
    .configSelfCap.commonConfig.chargeNum = kTSI_SscChargeNumValue_7,        // SSC T3 output bit0's period setting
    .configSelfCap.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_5, // SSC T1 output bit1's period setting
    .configSelfCap.commonConfig.prbsOutsel = kTSI_SscPrbsOutsel_2,           // SSC T2 output bit1's period setting
    .configSelfCap.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,           // SSC clock prescaler
```

Figure 69. PRBS register Clock settings in SSC mode

9.9 TSI IP HW register Tuning – Mutual Capacitance mode

Since the principle of TSI functionality is different in the Mutual mode, comparing to the Self-capacitance mode, the modified HW block is used with the Transmitting and Receiving circuits and the set of parameters, which must be configured by user is different.

9.9.1 Sensitivity tuning for Mutual mode

The default register configuration is the experimentally proven and should fit for most of the applications. The parameters denoted in "**bold**" are the most important for basic tuning:

.configMutual.commonConfig.mainClock = kTSI_MainClockSlection_0, // Set main clock

.configMutual.commonConfig.mode = kTSI_SensingModeSlection_Mutual, // sensing mode = Mutual OK

.configMutual.commonConfig.dvolt = kTSI_DvoltOption_0, // Default: 0 (best) internal comparator threshold voltage

.configMutual.commonConfig.cutoff = kTSI_SincCutoffDiv_0, // Divides the accumulated result, 0 recommended

.configMutual.commonConfig.order = kTSI_SincFilterOrder_2, // Length and multiply of the accumulated result

***.configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4**, // Multiple of real TSI scans (longer acc.)*

.configMutual.commonConfig.chargeNum = kTSI_SscChargeNumValue_4, // SSC clock settings

.configMutual.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2, //SSC clock settings

.configMutual.preCurrent = kTSI_MutualPreCurrent_4uA, // Default: 4uA, controlling the RX signal bias voltage.

.configMutual.preResistor = kTSI_MutualPreResistor_4k, // Default: 4k, controlling the RX bias voltage; URX > 0

***.configMutual.senseResistor = kTSI_MutualSenseResistor_10k**, // Rs resistor, used for translation of the received U to I*

*.**configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA**, // Sens boost factor minimized (No benefits for SNR)*

.configMutual.TXDriveMode = kTSI_MutualTXDriveModeOption_0, // Default 0: (5V/-5V), 1: (0/5V) TX signal waveform gener

*.**configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32**, // Change this for sensitivity tuning,.*

.configMutual.pmosRightCurrent = kTSI_MutualPmosCurrentMirrorRight_1, // Default: 1

*.configMutual.enableNmosMirror = true, // Default: **true**, Must be enabled*

*.configMutual.nmosCurrent = kTSI_MutualNmosCurrentMirror_1, // Default: 1, **the same as** "MutualPmosCurrentMirrorRight"*

Besides the clock settings, this is directly influencing the speed of the measurement (switching clock) and accumulated result. There are not many parameters which can be tuned differently from the default values.

- ***kTSI_DvoltOption_0*** is recommended
- ***kTSI_SincCutoffDiv_0*** is recommended
- ***kTSI_SincFilterOrder_2*** is recommended, we can try to decrease to "1", while increasing the decimation
- ***SincDecimationValue_4*** can be increased to get higher number of scans, longer accumulation time and higher resolution.

There is no option to control the strength of the Transmitter signal.

We can only control the shape of the generated TX signal:

TX signal options:

- ***kTSI_MutualTXDriveModeOption_0*** = 0U, /*!< TX drive mode is -5v~+5v, used in mutual-cap mode */
- ***kTSI_MutualTXDriveModeOption_1*** = 1U, /*!< TX drive mode is 0v~+5v, used in mutual-cap mode */

The following two parameters are responsible for setting the proper RX signal bias (offset) voltage which should be: **Vpre** > 0 in all cases for proper functionality, see the figure below.

- ***kTSI_MutualPreCurrent_4uA*** (default)
- ***kTSI_MutualPreResistor_4k*** (default)

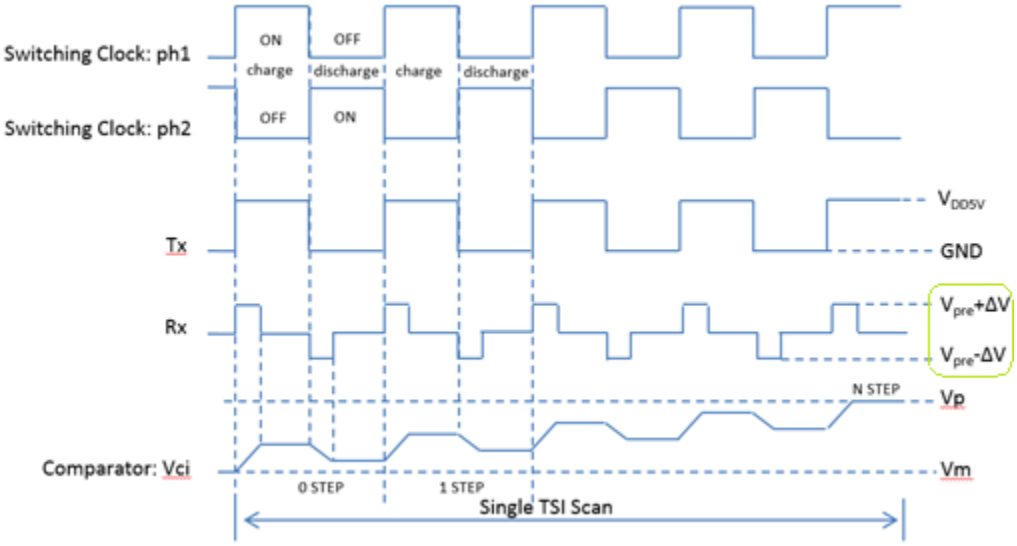


Figure 70. RX offset settings

We can measure the RX signal level and properly tune Vpre. Or we can try to switch the Transmitter to be transmitting the 0-5V levels instead -5V/5V

- *kTSI_MutualTXDriveModeOption_0* - Default 0: (5V/-5V), we can try to change to 0-5V
- *kTSI_MutualSenseResistor_10k* (default value). This resistor “Rs” is used for translating the received VRX voltage to current (Ichg / Idis) for current amplifier input.

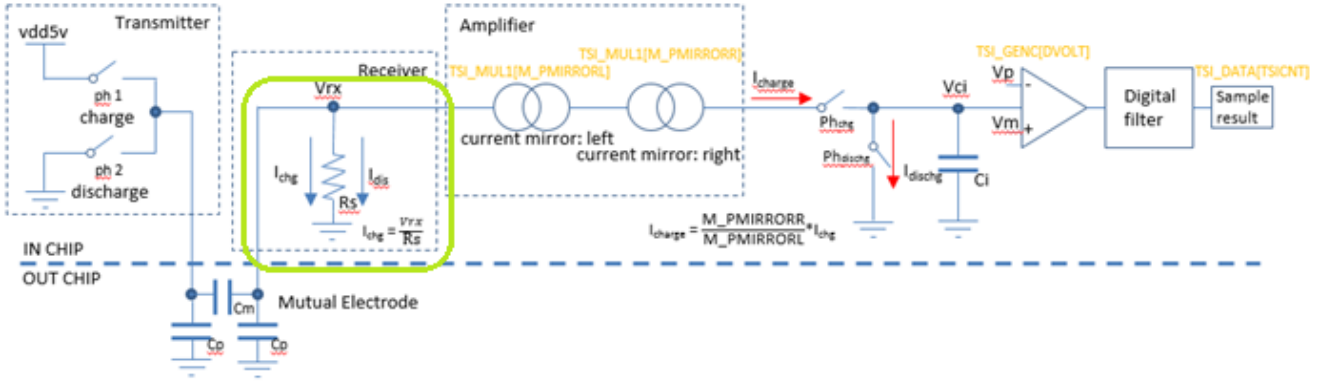


Figure 71. Mutual Sense resistor settings

Since the size of the Rs resistor converts the received voltage signal to Ichg, then the value of Rs is responsible for sensitivity as well. So, it makes sense to play with this value, even that 10kOhm is recommended default value. **Higher Rs value sensitivity**.

- *kTSI_MutualSenseBoostCurrent_0uA* is the default value, responsible for “**Sensitivity Boost**” in Mutual mode, which is different from Self cap mode.

With default settings (0uA) the boost feature is very weak. Increasing the value increases the sensitivity, but also the sensitivity to noise, so that SNR may not be improved. We can try to increase the boost current setting per small steps to get the best results.

- *kTSI_MutualPmosCurrentMirrorLeft_32* this is very important parameter, controlling the gain of the internal current amplifier. **Higher number results to higher amplification.**

Because the amplification factor is given by:

kTSI_MutualPmosCurrentMirrorLeft / kTSI_MutualPmosCurrentMirrorRight.

- *kTSI_MutualPmosCurrentMirrorRight_1* – this is recommended to have default value = 1
- *kTSI_MutualNmosCurrentMirror_1*

Both settings above should stay equal, cut increasing the values may lead to the faster response.

- *configMutual.enableNmosMirror* = **true**

This must be "**true**" in all cases

9.9.2 Mutual sensitivity tuning remarks

- We should play only with the parameters in bold at the beginning.

.configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32, // Change this for sensitivity tuning,.

.configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4, // Multiple of real TSI scans (longer accumulation)

configMutual.commonConfig.order = kTSI_SincFilterOrder_2, // Length and multiply of the accumulated result, try "1" as well.

.configMutual.senseResistor = kTSI_MutualSenseResistor_10k, // Rs resistor, used for translation of the received voltage to current

- Then we can try to increase the sensitivity boost current per small steps to achieve the best results:

.configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA

- Then we can try to adapt the RX bias voltage by:

.configMutual.preCurrent = kTSI_MutualPreCurrent_4uA, // Default: 4uA, controlling the Rx signal bias voltage.

.configMutual.preResistor = kTSI_MutualPreResistor_4k, // Default: 4k, controlling the Rx bias voltage; Urx must be > 0

- Keep M_PMIRRORR and M_NMIRROR the same

kTSI_MutualPmosCurrentMirrorLeft = kTSI_MutualNmosCurrentMirror_1

kTSI_MutualPmosCurrentMirrorRight = kTSI_MutualPmosCurrentMirrorRight_1

- Keep in mind that the clock settings affect the result of the measurement (length of the accumulation) and the accumulated value in the counter.

10 Shielding principles

Shielding methods are used to eliminate or the environmental influences like temperature drifts, humidity on PCB or water droplets on the touch control panel.

- Issue: Critical for humid environments is that new touch interfaces are capable of detecting differences between water drops and water layer capacitance or finger capacitance.
- Workaround: Keypad designs with a "shield" electrode(s) that detects or compensates the overall system noise or overall keypad capacitance.

10.1 SW shield function

The NT library provides the SW shielding function. This function is intended to detect false touches caused by water drops and to eliminate low-frequency noise modulated on the capacitance signal. When shielding function is enabled, the shield capacitive value is subtracted from the related electrode capacitive raw signal.

The library shown good performance under water droplets and thin water films. It just needs the proper calibration to detect touches accurately under these conditions.

Shield electrode itself can be used additionally as a GUARD sensor, when an "invalid touch" is detected on it.

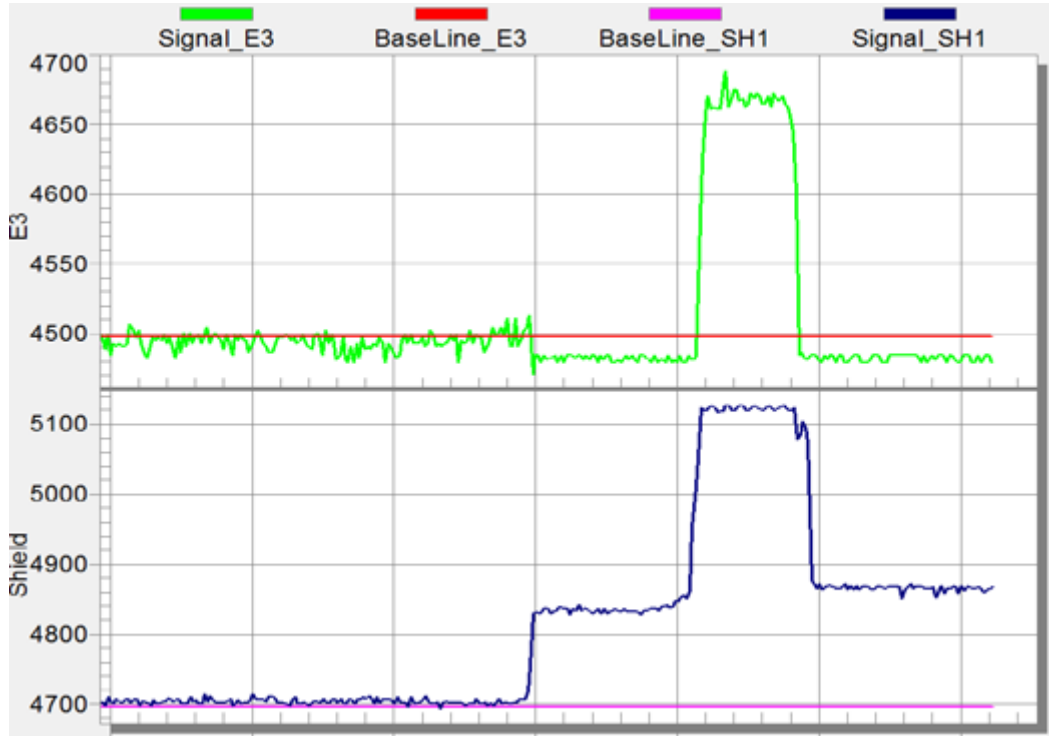


Figure 72. SW shielding compensation principle

Example figure shows an electrode instant signal and its shield. As seen from the shield signal (blue), at time = 10 seconds, a thin water film is placed on the board. But the electrode signal (green) stays around its baseline (red). At time = 12.5 seconds, a finger touch is made. The electrode delta seems like a regular touch signal due to the subtraction from the shield.

10.1.1 SW Shield Setup

The shield electrode is mostly the special electrode or PCB pattern, which is not touched under the normal conditions and it is used to detect the unwanted events.

The shield electrode can be assigned to the regular electrode. All regular touch electrodes may share single shielding electrode or different shield electrodes can be assigned for different touch sensing electrodes. In special cases, the regular electrodes may be used as shields for the other electrodes, for instance to compensate the unwanted touch signal crosstalk's between adjacent buttons to avoid unwanted touch detection.

The SW shielding setup is given by the parameters in "nt_electrode" structure definition and may be assigned to every electrode. If the ".shielding_electrode" is undefined or NULL, the shielding feature is unused and the rest of the parameters are ignored.

```
const struct nt_electrode electrode_22 =
{
    .pin_input = EVB_BOARD_TOUCHPAD_0_ELECTRODE,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_keypad,
    .shielding_electrode = &electrode_27sh,
    .shield_threshold = 5,
    .shield_gain = 30,
    .shield_sens = 800,
};

/* Shield electrode */
const struct nt_electrode electrode_27sh =
{
    .pin_input = EVB_BOARD_SHIELD_ELECTRODE,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_keypad,
};
```

Figure 73. SW shield electrode assignment and config

- "shielding_electrode" is the electrode used for shielding, shielding electrode has its own configuration. If the common signal change is detected simultaneously on the “SH” electrode and the “normal” electrode, then the SW compensation may activate.
- "shield_threshold" is the minimal common signal level, where the shielding is activated
- "shield_gain" is the multiplication factor used for shielding electrode signal (to make the shield electrode signal proportional to the “normal” touch electrode)
- "shield_sens" is the maximal shield electrode offset signal, used for common signal offset compensation. It means that all signal values < "shield_sens“ (800) will be compensated (sw subtracted) and if the value > "shield_sens“ (800), it won’t be compensated and it can be evaluated as a valid touch, under “worse” environmental conditions.

Shielding electrode is not expected to be touched under the normal conditions. We can setup a key detector and “touch” threshold for it as well so it may act as a GUARD sensor as well (water split issue detection, etc). And then block the particular key or complete keypad.

10.2 SW Shield Advantages and Disadvantages

The shielding strength must be configured and tuned for every electrode separately in SW setup. However, this can provide better flexibility, in case of special needs or complex PCB layout.

The shielding electrodes occupies the standard TSI channels and they behave as they are scanned as regular touch electrodes.

10.3 HW Shielding feature (driven shield signal)

Besides the SW shielding available in the library, KE15z device provides another approach for shielding. This is different technique than previously described one, because the parasitic capacitance compensation is done on physical level.

The KE15z device support **one HW shielding signal output** available at **TSI ch12 (PTC5).**

Can be enabled by single TSI module register bit in **hw_config** by:

.configSelfCap.enableShield = true,

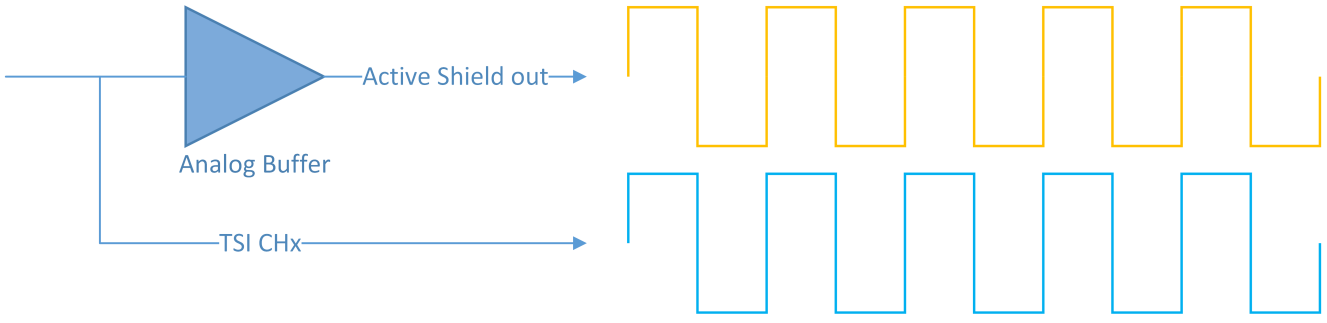


Figure 74. Active Shield output generation

The driven-shield signal is a buffered “copy” of the sensor-charging signal. (sample amplitude, frequency and phase)

The buffer provides enough current to drive the high parasitic capacitance of the hatch fill on the PCB acting as a Shield electrode.

The Effect of Liquid Droplets and Liquid Stream on Cap sensor section, because the shield electrode is driven with a voltage which is the same as the sensor-switching signal, the capacitance added by a liquid droplet when on the touch surface will be nullified.

For the best water-tolerance performance, it is required that the driven shield signal has the same “shape” (voltage and phase) as the sensor-switching signal.

The PCB with active shield feature must be carefully designed and the discrete external components must be properly selected and tuned for good functionality.

Active Shield can reduce the intrinsic capacitance of the PCB, which increases the overall sensitivity of the standard touch electrodes, which can be beneficial for instance for Proximity sensing.

11 New features supported in NXP Touch software library

The hardware and software enhancements that support the latest KE1x devices are as follows:

- KE17 supports two on-chip TSI peripherals and more channels.
- Flexible hardware register configuration for individual TSI channels can be selected.
- More Active Shield outputs can be selected and enabled.

11.1 Adding support for second TSI peripheral module

The KE17Z device supports a second on-chip TSI peripheral. It adds more available TSI channels and allows the TSI scans to be performed in parallel, which reduces the overall response time.

Adding the software configuration for a second TSI peripheral is very easy. The same “tsi_hw_config” structure can be shared by both TSI peripherals or a secondary “tsi_hw_config” structure can be created when different “general” register settings are needed for the second TSI peripheral. In both cases, the “general” register settings may be overridden by a specific “tsi_hw_config” assigned to an individual electrode or a group of electrodes. NXP Touch library selects and reloads the proper TSI register settings before the electrode is scanned.

11.1.1 Second TSI module setup in the “nt_setup.c” file

Both TSI peripheral modules are identical in terms of functionality and register settings, but their TSI inputs are routed to different pins, depending on the device package. See the device reference manual and datasheet for proper TSI channel assignments. Some channels may be unavailable on the smaller package derivatives.

Every on-chip TSI module has its own configuration register group. Both modules can share common hardware register settings (*“tsi_hw_config”* defined in the *“nt_setup.c”* file), but they have different groups of electrodes defined. See Figure 75.

```
const struct nt_electrode *const nt_tsi_module_electrodes_1[] = {&El_1, &El_3,
                                                                 &El_8, &El_9,
const struct nt_electrode *const nt_tsi_module_electrodes_2[] = {&El_2, NULL};

const struct nt_module nt_tsi_module_1                        = {
    .interface          = &nt_module_tsi_interface,
    .wtrmark_hi         = 65535,
    .wtrmark_lo         = 0,
    .config             = (void *)&tsi_hw_config,
    .instance           = 0,
    .electrodes         = &nt_tsi_module_electrodes_1[0],
};
const struct nt_module nt_tsi_module_2                        = {
    .interface          = &nt_module_tsi_interface,
    .wtrmark_hi         = 65535,
    .wtrmark_lo         = 0,
    .config             = (void *)&tsi_hw_config,
    .instance           = 1,
    .electrodes         = &nt_tsi_module_electrodes_2[0],
};
```

Figure 75. Adding support for a second peripheral to the configuration file

11.2 Flexible TSI hardware configuration assignment

It can be beneficial to use the different hardware configuration for different TSI modules to match the application needs. In that case, the two different TSI hardware configuration structures must be created and assigned to different TSI peripherals like *"tsi_hw_config1"* and *"tsi_hw_config2"*.

The NXP Touch library then reads the proper register configuration assigned to the module while performing the individual scans. The TSI hardware configuration structures contain the common register values used globally for all TSI channels (electrodes) assigned to the given TSI module.

This approach works only on devices physically supporting more TSI modules on a chip.

11.2.1 Assigning a specific hardware configuration to an electrode

When a better flexibility is required, a special hardware configuration valid for an individual electrode can be assigned. This situation may occur when a specific electrode or an electrode group must be tuned separately, because a different physical behavior or a different sensitivity is required.

```
const struct nt_electrode El_1 = {
    .shielding_electrode      = NULL,
    .multiplier               = 0,
    .divider                  = 0,
    .shield_threshold         = 5,
    .shield_gain              = 30,
    .shield_attn              = 1,
    .shield_sens              = 800,
    .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
    .keydetector_interface    = &nt_keydetector_usafa_interface,
    .pin_input                = FRDM_TOUCH_BOARD_TSI_1,
    .tsi_hw_config = (void *)&tsi_hw_config_specific,
};
```

Figure 76. Specific hardware configuration assigned to a given electrode

When the *".tsi_hw_config"* parameter in the *"nt_electrode"* structure is left undefined, the software library uses the default hardware configuration assigned globally for the TSI module.

The library just checks this parameter during the runtime and if it is different from *"NULL"*, then an alternative hardware register configuration for a specific electrode is reloaded before a specific electrode is being scanned. This approach can be also used for KE1x devices with a single TSI module on a chip. Different electrodes (channels) assigned to a single TSI module can be scanned with different register settings.

11.3 Added Active Shield options

Starting from the KE1x devices, more Active Shield (AS) outputs are available for the self-capacitive mode. Every TSI peripheral supports up to 3 buffered AS outputs, which adds more flexibility for PCB design and more options for AS usage. See Shielding principles for details.

- The flexibility means that the AS can be enabled only for a single electrode or a group of electrodes and the rest of the electrodes can be scanned without the AS enabled.
- Another option is to use more AS outputs for different group of electrodes.

The individual AS outputs are activated together with the regular TSI channel being scanned by the TSI module. The shield activation/deactivation can be enabled/disabled by the S_W_SHIELD [2:0] individual bits in the TSI_MODE register.

See the below figures for more details about the AS settings in the NXP Touch configuration:

```
const tsi_config_t tsi_hw_config = {
    .configSelfCap.commonConfig.mainClock     = kTSI_MainClockSlection_0,
    .configSelfCap.commonConfig.ssc_mode      = kTSI_ssc_prbs_method,
    .configSelfCap.commonConfig.mode          = kTSI_SensingModeSlection_Self,
    .configSelfCap.commonConfig.dvolt         = kTSI_DvoltOption_0,
    .configSelfCap.commonConfig.cutoff        = kTSI_SincCutoffDiv_0,
    .configSelfCap.commonConfig.order         = kTSI_SincFilterOrder_2,
    .configSelfCap.commonConfig.decimation    = kTSI_SincDecimationValue_6,
    .configSelfCap.commonConfig.chargeNum     = kTSI_SscChargeNumValue_4,
    .configSelfCap.commonConfig.prbsOutsel    = kTSI_SscPrbsOutsel_2,
    .configSelfCap.commonConfig.noChargeNum   = kTSI_SscNoChargeNumValue_2,
    .configSelfCap.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,
    .configSelfCap.enableSensitivity          = false,
    .configSelfCap.enableShield               = kTSI_shieldAllOff,
```

Figure 77. AS settings in NXP Touch software configuration

The *"configSelfCap.enableShield"* parameter contains the AS settings used for the current hardware configuration. Different hardware configuration structures can be assigned to different groups of electrodes.

For KE17z7 and KE16z7, the following possible parameters and AS combinations are available:

```
typedef enum _tsi_shield
{
    kTSI_shieldAllOff  = 0U, /*!< No pin used */
    kTSI_shield0On     = 1U, /*!< Shield 0   pin used */
    kTSI_shield1On     = 2U, /*!< Shield 1   pin used */
    kTSI_shield1and0On = 3U, /*!< Shield 0,1 pins used */
    kTSI_shield2On     = 4U, /*!< Shield 2   pin used */
    kTSI_shield2and0On = 5U, /*!< Shield 2,0 pins used */
    kTSI_shield2and1On = 6U, /*!< Shield 2,1 pins used */
    kTSI_shieldAllOn   = 7U, /*!< Shield 2,1,0 pins used */
} tsi_shield_t;
```

Figure 78. AS options

12 Conclusion

This document describes basic usage and development using the NXP Touch library demonstrated on FRDM-KE1x boards and FreeMASTER-based NXP GUI Tool. The TSI hardware capacitive touch sensing principles and touch sensitivity tuning is described in detail. The last section describes the shielding methods available in the NXP Touch library and TSI hardware.

13 References

1. NXP Touch library reference manual

https://www.nxp.com/docs/en/reference-manual/NT20RM.pdf

1. KE15 Touch Sensing Interface (Document: KE15ZTSIUG)

http://www.nxp.com/doc/KE15ZTSIUG

1. FRDM-TOUCH Quick Start Guide

https://www.nxp.com/docs/en/user-guide/FRDMTOUCHQSG.pdf

1. Kinetis KE1xZ Sub-Family Reference Manual

https://www.nxp.com/webapp/Download?colCode=KE1XZP100M72SF0RM

1. AN3863, Designing Touch Sensing Electrodes – Application Note

https://www.nxp.com/docs/en/application-note/AN3863.pdf

14 Revision history

Table 2. Revision history

Revision number	Date	Substantive changes
1	07 December 2021	Added New features supported in NXP Touch software library
0	01/2020	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07 December 2021

Document identifier: AN12709

arm