How to improve conducted noise robustness for touch sensing applications on STM32 MCUs

Introduction

Different levels of immunity to conducted RF voltage are required by touch sensing systems, depending on the application. Moreover, touch sensing systems are often designed to meet the requirements of industry standards, especially in the EMC compliance domain.

It is important to understand the environment in which the touch application is used, and to apply suitably adapted techniques to address the effects of unwanted noise disturbances.

This document provides a basic overview of conducted immunity testing and some guidelines to keep the system reliable when it is exposed to conducted noise.

STMicroelectronics provides free STMTouch touch sensing firmware libraries, directly integrated into the corresponding STM32Cube package.

Table 1. **Applicable products**

Type	Product series
Microcontrollers	• STM32F0 series, STM32F3 series • STM32L0 series, STM32L1 series, STM32L4 series, STM32L4+ series, STM32L5 series • STM32U0 series, STM32U5 series • STM32WB series, STM32WBA series • STM32U3 series

1 General information

This document applies to the STM32 Arm®-based microcontrollers.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

arm

2 Conducted noise immunity

2.1 Noise immunity

The noise immunity is an important characteristic in the evaluation of a touch sensing system.

The standard IEC61000-4-6 details the test environment for the conducted noise.

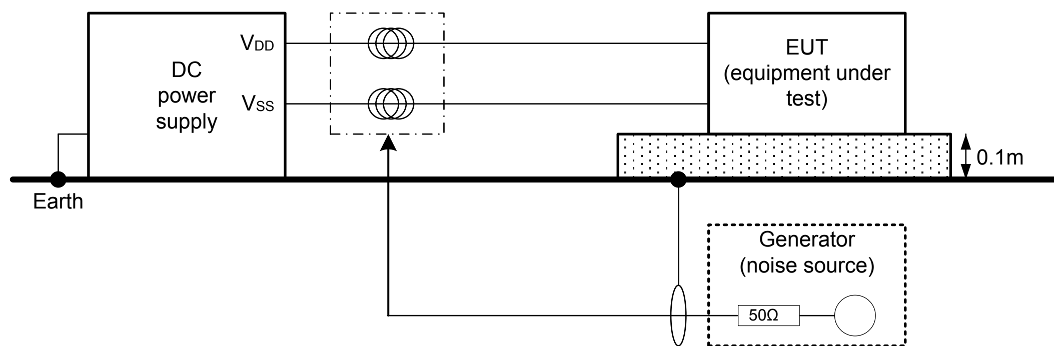
2.2 IEC61000-4-6 standard

The IEC61000-4-6 standard specifies the test procedure to evaluate the conducted noise immunity of an EUT (equipment under test).

2.2.1 Standard IEC61000-4-6 test setup

The figure below shows how the modulated noise signals are injected into the equipment under test.

Figure 1. Standard IEC61000-4-6 test setup



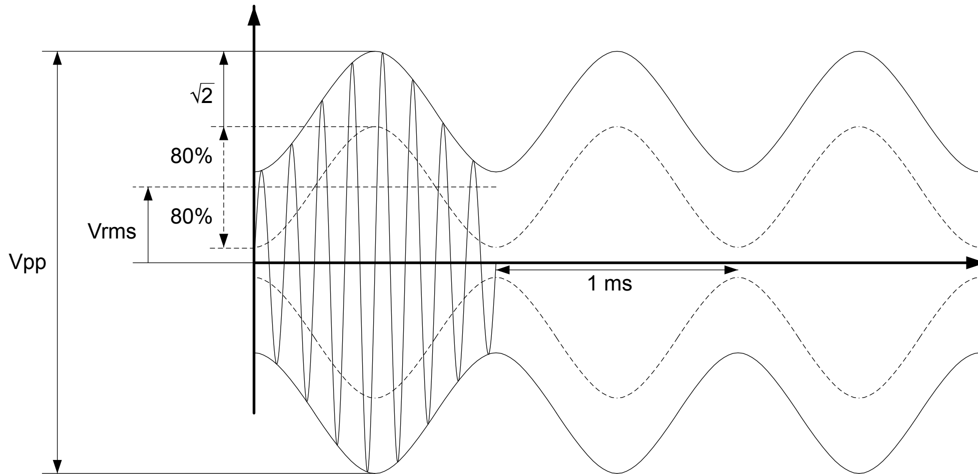
Touch sensing systems are based on capacitor variation measurements. The system must be able to detect capacitive variations as low as few picofarads on the sensor electrodes. Therefore, such systems may be sensitive to conducted noise.

In the test setup shown in Figure 1, the injected signal simulates the noise perturbations to which a system may be exposed. By varying the frequency and the level of the injected signal, the test setup allows the characterization of situations where the touch system becomes unreliable.

2.2.2 Injected signal characteristics

The injected signal is a swept modulated noise source with a sine wave envelope as shown in the figure below.

Figure 2. Injected signal



The noise generator frequency range is swept from 150 kHz to 80 MHz. The frequency is swept incrementally. The step size must not exceed 1 % of the preceding frequency value. The signal is 80 % amplitude modulated with a 1 kHz sine wave.

The modulated noise signal amplitude may be expressed either in V_{rms} or V_{pp}.

Here is the formula to convert values from V_{rms} to V_{pp}:

$$Vpp\ value = Vrms \times \sqrt{2} \times 1.8 \times 1.2$$

2.2.3 Noise immunity evaluation

The EUT noise immunity is evaluated by testing the ability of the EUT to behave according to the definition of a given class when it is submitted to a given noise level. The table below summarizes the different noise levels and classes.

Table 2. Test levels

Standard class \ noise level	Level 1 $1\ V_{rms}$	Level 2 $3\ V_{rms}$	Level 3 $10\ V_{rms}$
Class A: system works normally.	Pass/fail	Pass/fail	Pass/fail
Class B: Some degradation in operation may occur (false touch detection or touch masking), but the product recovers once the stress is removed without any operator intervention.	Pass/fail	Pass/fail	Pass/fail
Class C: same as class B but needs an external action (such as reset or power off/on) to return to normal state.	Pass/fail	Pass/fail	Pass/fail
Class D: system loosing function or degradation of performance that is not recoverable	Pass/fail	Pass/fail	Pass/fail

2.2.4 IEC61000-4-6 standard limitation

The minimum frequency step recommended by the standard to sweep the injected signal from 150 kHz to 80 MHz is 1 % of the preceding frequency value. At 500 kHz this represents a 5 kHz step.

On most touch sensing systems, these steps are too large to be able to isolate the worst case situations. Some applications with narrow critical wave band can pass 3 V_{rms} if the critical frequency falls just in the middle between tested frequencies. The same application does not pass 1 V_{rms} if the user sets the test exactly on the critical frequency.

It is then important to set smaller steps around the critical frequencies. Sometime the standard bench is not able to do the appropriate step (for example 100 Hz). Section 4: Test set up proposal to detect worst case proposes a method to detect the worst case.

3 Surface charge transfer acquisition principle overview

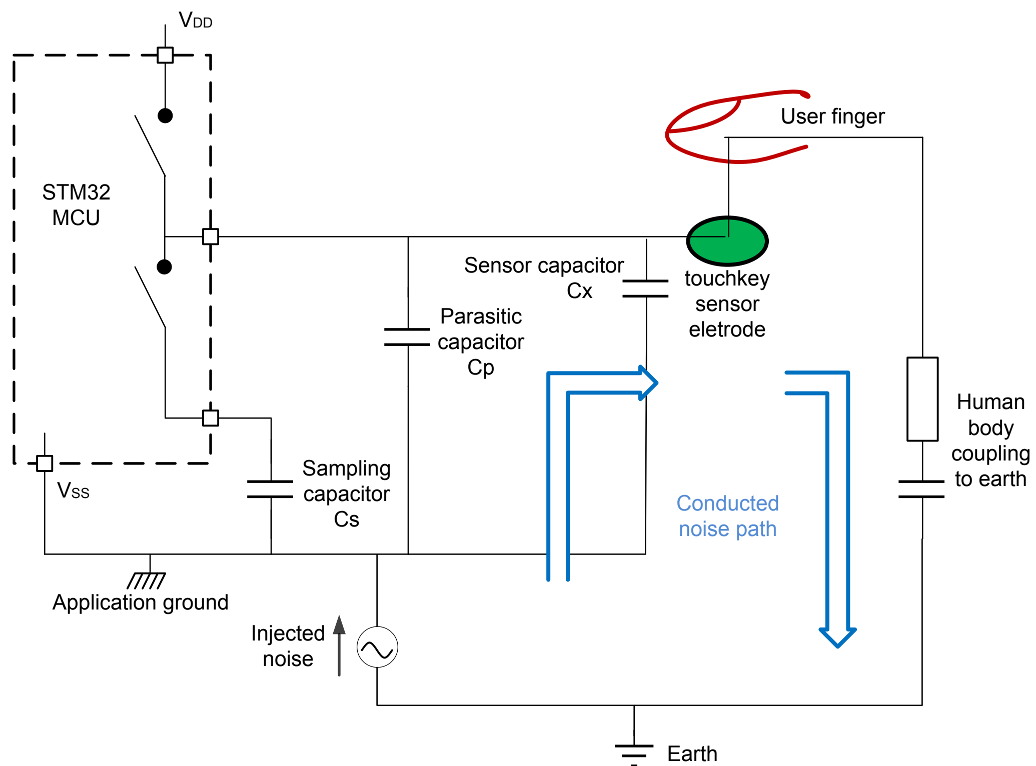
The STM32 microcontrollers use a surface charge transfer acquisition principle, briefly described below.

The surface charge transfer acquisition principle consists in charging a sensor capacitance (C_X) and transferring a part of the accumulated charge into a sampling capacitor (C_S). This sequence is repeated until the voltage across C_S reaches a given threshold (V_{IH} in our case). The number of charge transfers required to reach the threshold is a direct representation of the size of the electrode capacitance.

When the sensor is touched, the sensor capacitance to the earth is increased so the C_S voltage reaches the threshold with less count and the measurement value decreases. When the measurement value falls below a defined threshold, a detection is reported.

The noise injection disturbs the measurement proportionally to its amplitude and depending on its frequency. The worst case is generally found at a noise frequency close to the charge transfer frequency (assuming no techniques are used to spread this frequency).

Figure 3. **Charge transfer equivalent capacitance model**

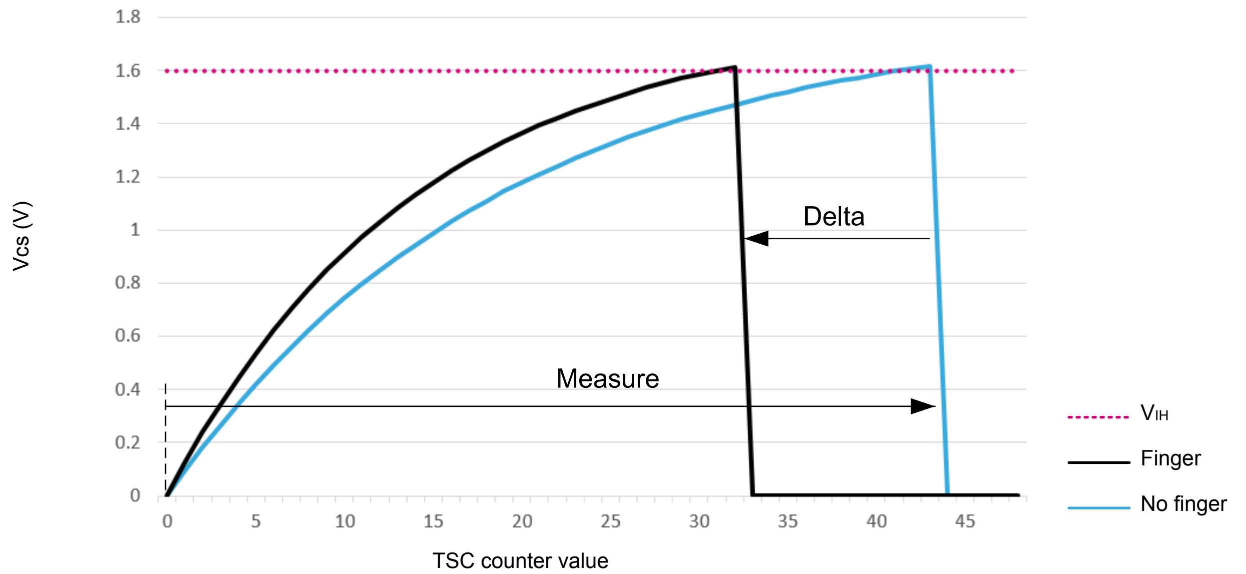


3.1 Sampling capacitor (C_S) voltage level without noise

The figure below describes the C_S voltage behavior with and without finger on the sensor. The finger capacitance added on the sensor speeds up the C_S charge. When V_{cs} reach V_{IH}, the TSC peripheral stops charging the transfer state machine.

After this step, a discharge is done and V_{cs} goes from V_{IH} to 0 V.

Figure 4. Vcs behavior without noise



3.2 Sampling capacitor (Cs) voltage level with noise

The figures below describe C_S voltage behavior with and without finger on sensor during conducted noise injection.

Figure 5. Vcs behavior with high-frequency conducted noise

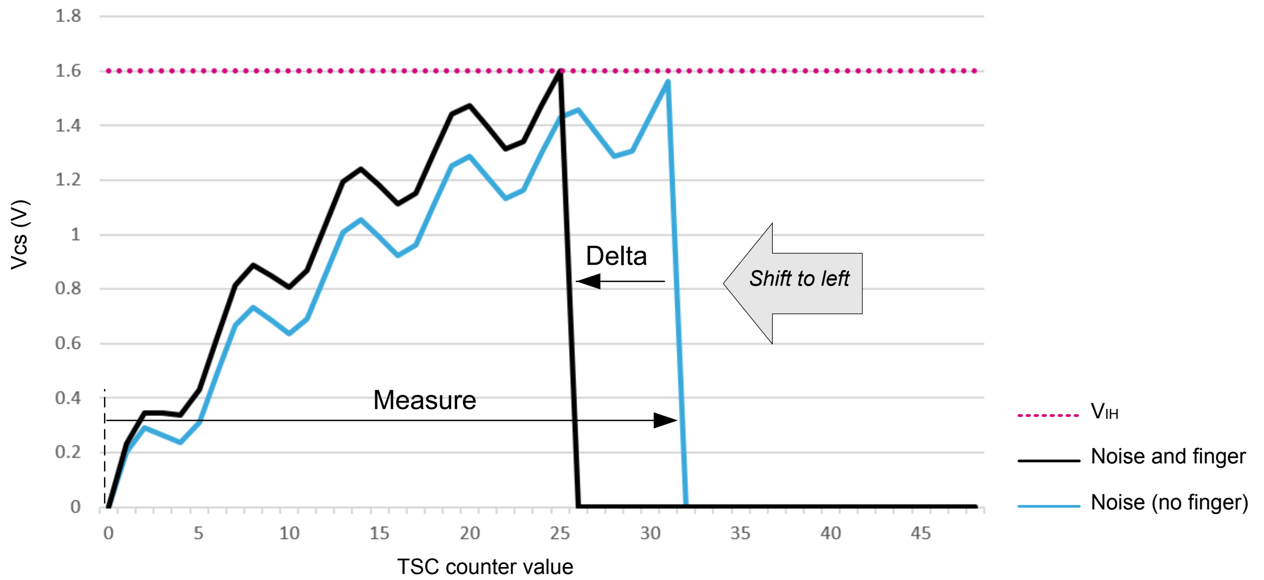
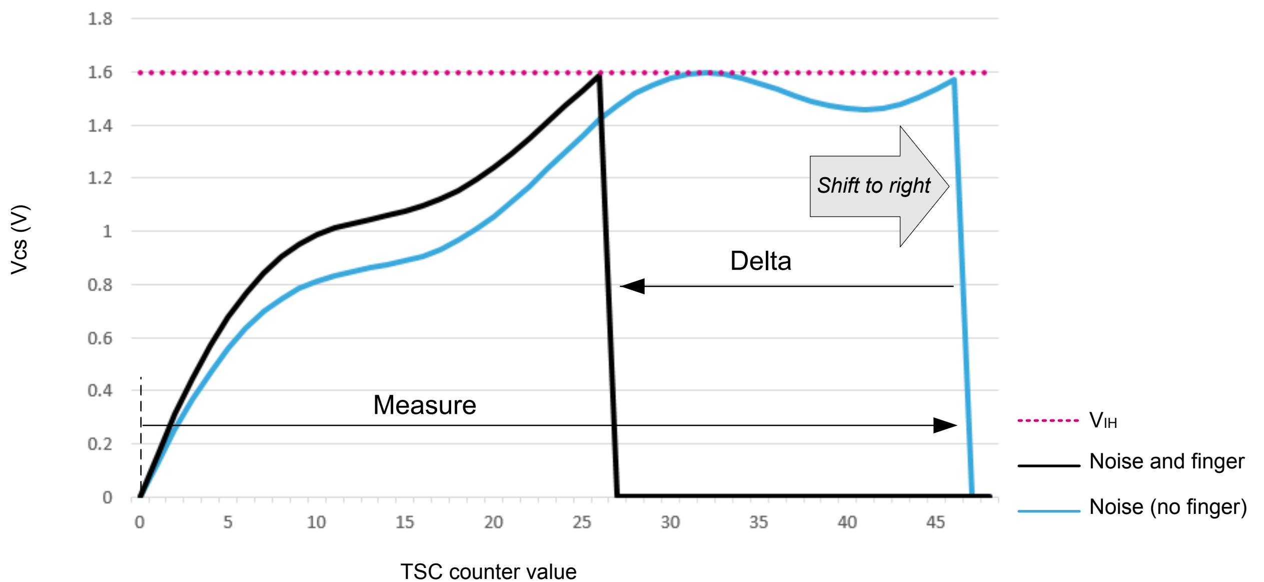


Figure 6. Vcs behavior with low-frequency conducted noise



The measure value increases or decreases with the conducted noise frequency range. Regarding the delta value, the behavior is less predictable. The TSL (touch sensing library) debounce feature helps to filter these effects (see debounce and threshold setting in Section 5.6: Software filter (debounce)).

4 Test set up proposal to detect worst case

Most of the IEC61000-4-6 compliant generators do not offer the ability to generate a step smaller than 1 kHz. In order to determine the most critical noise frequency, it is usually required to use 100 Hz steps or even 10 Hz steps. The setup detailed in this section is an alternative to find out the most critical noise frequencies and to evaluate the robustness of the EUT at these frequencies.

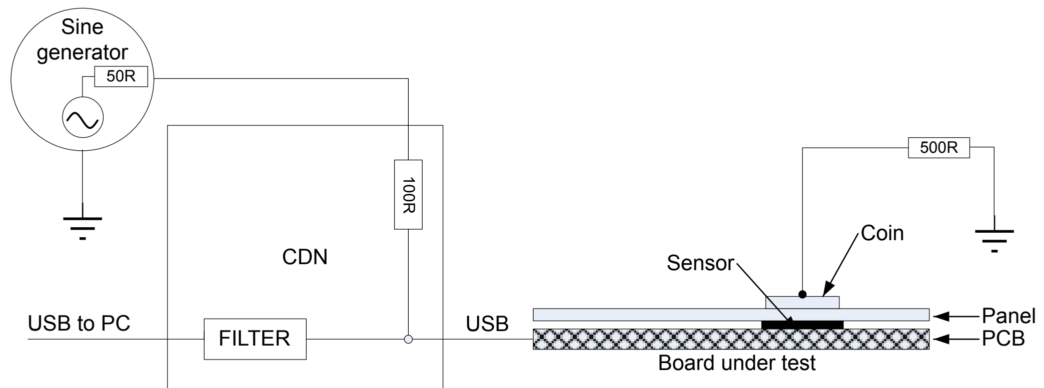
Note: The noise frequencies impacting the equipment are around the charge transfer frequency and up to 40 MHz. Higher noise frequency does not have an impact on the equipment operation.

4.1 Test setup

In order to simulate a human finger, a copper coin (from10 to16 mm diameter) with a 500 Ω serial resistor connected to GND can be used.

The CDN adaptor is the same as the one used in IEC61000-4-6.

Figure 7. **Test condition**



4.2 Generator settings

The following steps are needed to set the generator:

1. Select “sine wave” and “sweep” mode.
2. Sweep menu: time = 300 s, return time = 0, linear, interval = 1 ms
3. Set V_{pp} value on the generator to obtain the corresponding voltage injected on the EUT in case of standard test.
   Example: to test in the same condition as the standard at 3 V_{rms}, the user must adjust the generator voltage to inject 15 V_{pp} measured on the board with an oscilloscope.

   Note: On the AFG3102 Tektronix® generator, the modulation is not available in sweep mode.
4. Set the start and end frequencies such that the sweep range does not exceed 60 kHz for a 300 s duration. This recommendation allows the detection of the worst case level with sufficient accuracy (typically less than 5 % error).

Note: An external amplifier may be needed in case the generator is not able to reach the appropriate injection level.

4.3 Data logging and data processing

Data logging

The log function of the STM Studio tool can be used to collect data from the STM32 target (refer to the *STM-STUDIO-STM32* data brief for more details).

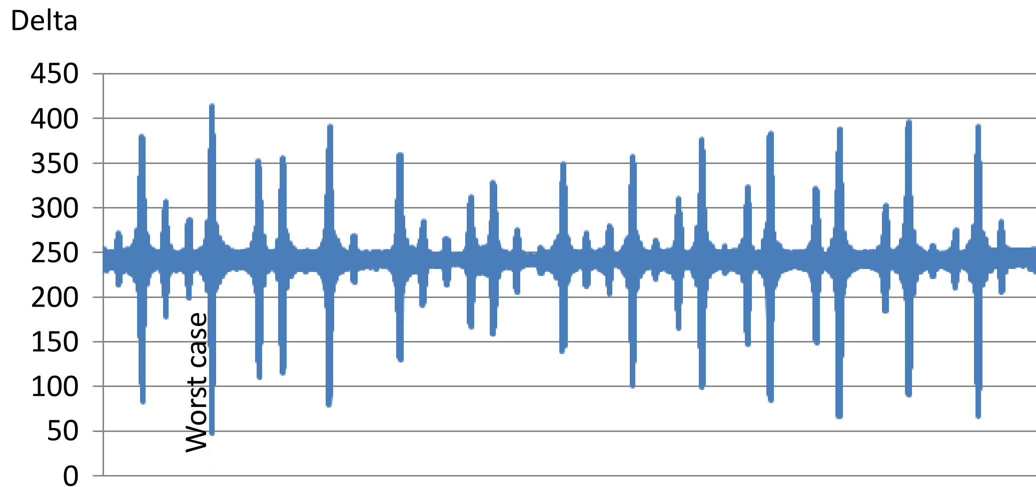
Example: “MyChannels_Data[x].Delta” and “MyTKeys[(x)].p_Data->StateId”

Data processing

In this example, the frequency sweep starts at 500 Hz and ends at 550 kHz. Noise level is set on the generator to 4.6 V_{pp} (no modulation). The detect out threshold is set to 50. The sensor is touched and the detection is valid if the measured delta is upper 50, else the touch detection is lost (error is reported). It means that 4.6 V_{pp} is the limit to avoid detection loss (4.5 V_{pp} pass).

As a comparison, IEC61000-4-6 standard recommendation to use 1 % frequency steps would have lead to explore only 10 frequencies in this range.

Figure 8. Data processing



5 How to improve noise immunity

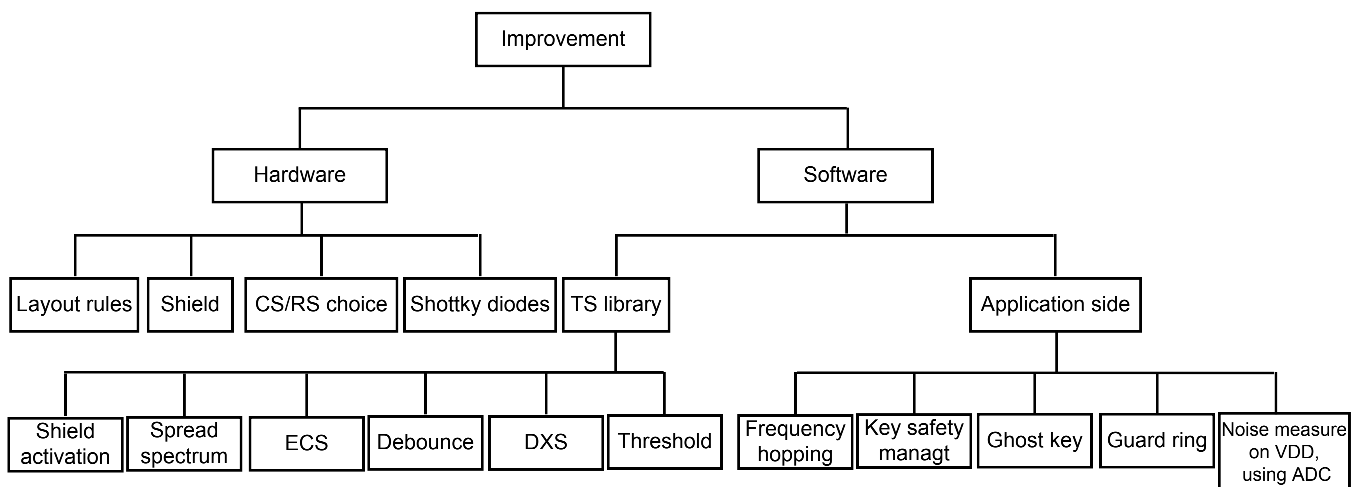
The two following directions can be followed to improve noise immunity:
- Decrease noise level.
- Increase signal (measurement sensitivity).

Note: In order to obtain a real benefit on the SNR, the noise reduction must not degrade the sensitivity and vice versa.

5.1 Proposed improvement techniques

The improvement techniques detailed below are based on hardware and software choices.

Figure 9. Process flow



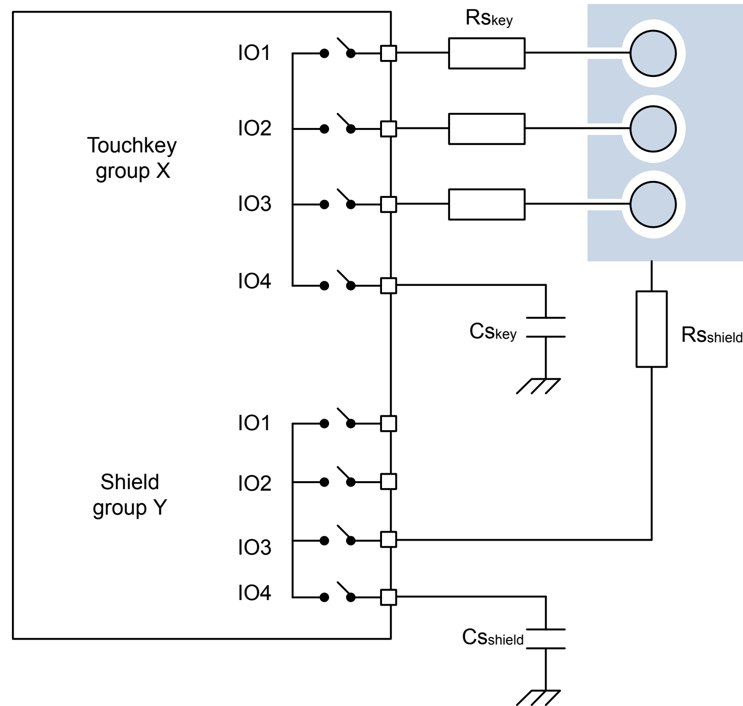
Several improvement techniques are introduced below:
- Active shield
  This feature increases the measurement sensitivity.
- Spread spectrum
  When activated, the spread spectrum creates several acquisition frequencies. This feature is particularly appropriate for reducing overall noise level inside the measurement when the noise is concentrated on certain frequencies (as opposed to a white noise).
- Detection thresholds
  Adjusting these parameters allows the optimization of the reported detection that is a compromise between false detection and detection loss.
- SW filter
  This feature, such as debounce filter, can be used to remove short and unwanted detections.
- Frequency hopping
  Upon detection of excessive noise on a dedicated channel, the firmware is able to change the acquisition frequency in order to move out of the disturbed frequency range. There may be some situations where frequency hopping and spread spectrum cannot be activated both at the same time. This is useful to get a Class B operation with no false touch detections (generally preferable than having some false touch detections).
- Channel blocking
  This feature uses an additional channel to detect noise and cancel touchkey detection if noise reaches a determined threshold. Sensors involved are ghost and guard-ring. Sensors involved are ghost and guard-ring.
- Impedance path to earth
  Decreasing the impedance path to earth is a good way to cancel the conducted noise effect (use of a metallic chassis, system ground and earth connected together). For instance, an application offering a direct connection between the earth and the system ground is not impacted by the conducted noise.

- Schottky diodes on sensor and shield
  Adding a low-capacitance Schottky diode like ST BAS70, increases the conducted-noise voltage level immunity. These Schottky diodes must be used on both sensors and shield.
- Reduce ripple on V_{DD}
- Sensor sensitivity
  Increasing sensor sensitivity decreases conducted-noise effects.
- Sampling capacitor C_S and serial resistor R_S
  Increasing C_S and/or R_S increases the acquisition time but decreases conducted-noise effects.
- Noise measurement on V_{DD}
  On application using the STM32 general purpose ADC, the LDO (or SMPS if any) input ripple can be measured and the application can decide to stop touch sensing acquisition when the conducted-noise voltage level is too high, or another counter measure such as increasing the debounce parameters.

5.2 Active shield

The active shield is an electrode that wraps around the sensor. The goal is to minimize the parasitic capacitance between the sensor and the ground. To drive the shield electrode, a channel of a dedicated group with its own C_S can be used (see the figure below).

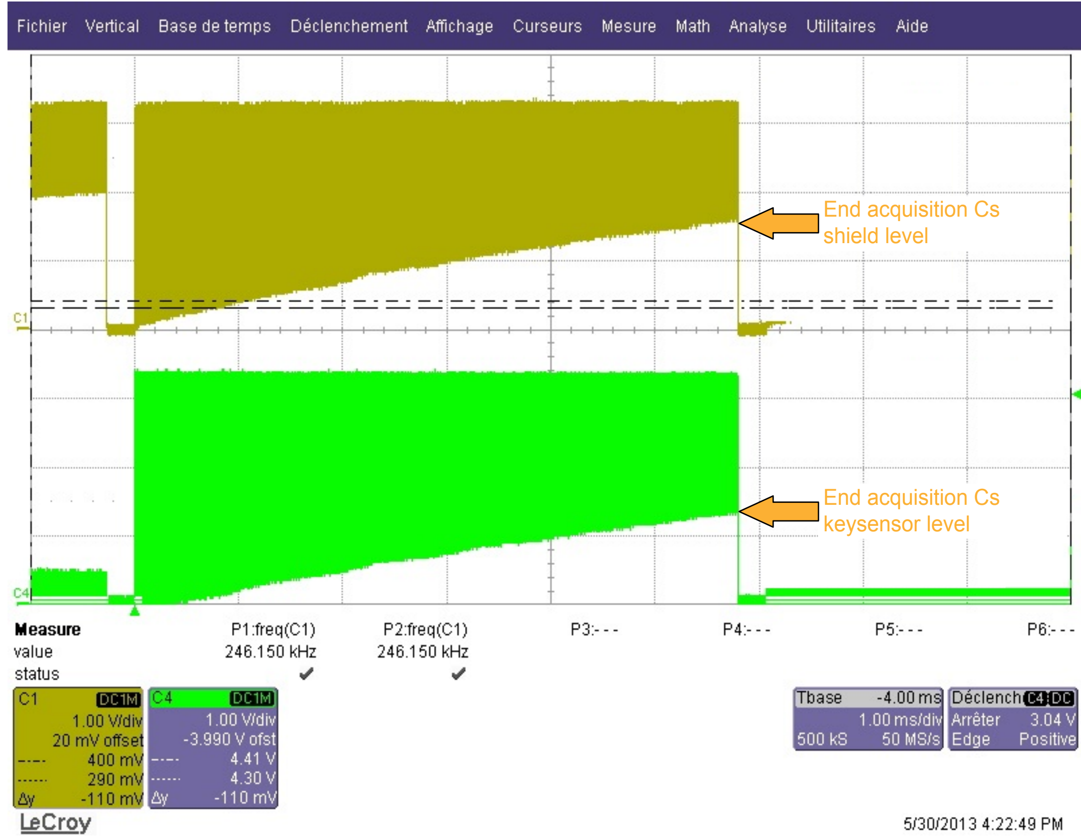
Figure 10. Active shield



It is important to check the shied electrode waveform. $C_{sshield}$ capacitance value and $R_{sshield}$ serial resistor value must be adjusted in order to obtain the same signal shape as sensor electrode waveform (same amplitude, same rising and falling time).

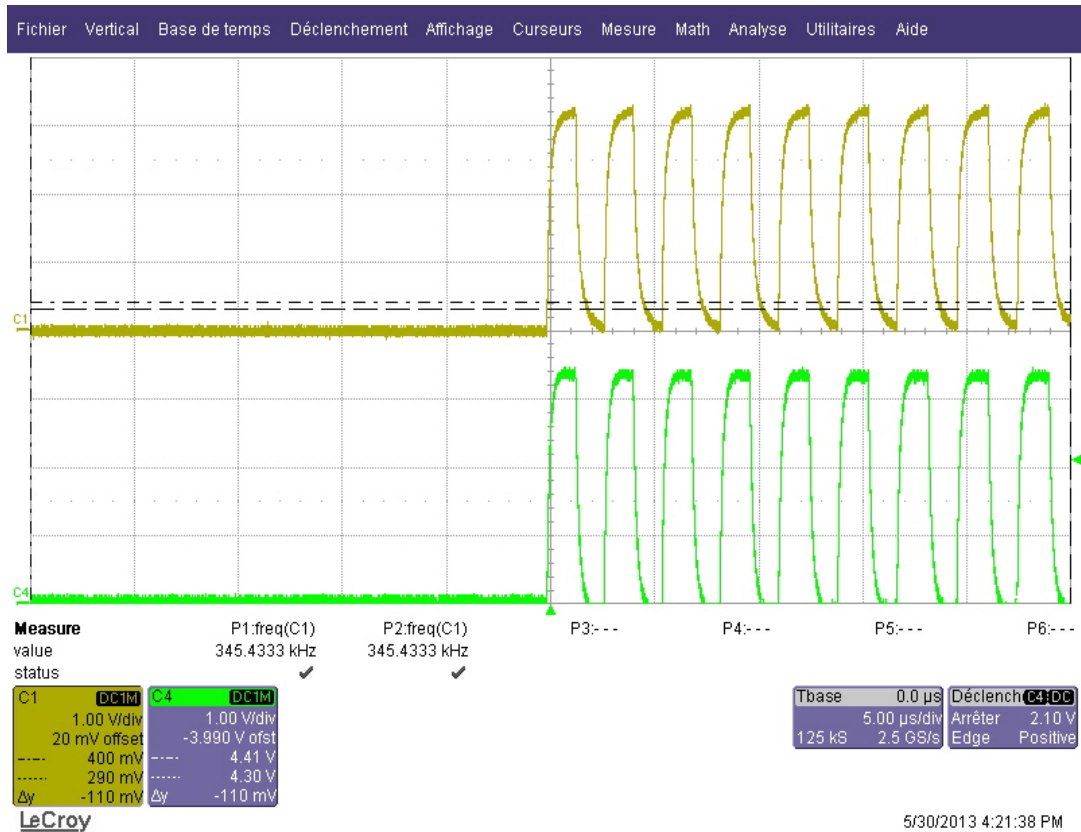
The figure below shows the electrode sensor waveform in green and shield electrode waveform in yellow. The $C_{sshield}$ is adjusted to obtain approximately the same charge level at the end of the acquisition.

Figure 11. Electrode and active shield waveforms



The figure below is a zoom of the previous one at the beginning of acquisition: $R_{sshield}$ and $C_{sshield}$ are adjusted to obtain approximately the same rise and fall time on the shield electrode as sensor electrode waveform.

Figure 12. Waveform detail



When the active shield is properly implemented, the count value is about twice as much as the count value without active shield. Since the noise level is not increased, the SNR is improved by a factor of 2, and noise immunity as well.

The negative impact of this feature is the requirement to dedicate two more I/Os and one more touch sensing group.

Refer to the following documents for more details on the active shield implementation:

- 'Drive shield' section of the application note *Design with surface sensors for touch sensing applications on MCUs* (AN4312)
- 'Shield adjustment' section of the application note *Tuning a touch sensing application on MCUs* (AN4316)
- TSC part of the STM32L4 online web training

To activate the shield functionality, from software point of view, using TSL, `HAL_TSC_Init` must be call with the following parameters:

```
htsc.Instance = TSC;
…
htsc.Init.ShieldIOs = TSC_GROUPx_IOy;
htsc.Init.SamplingIOs = … |TSC_GROUPx_IOz|….;
if (HAL_TSC_Init(&htsc) != HAL_OK)
{
  Error_Handler();
}
```

where `TSC_GROUPx_IOy` is the group of I/Os connected to $R_{sshield}$. (x = 1..8, y = 1..4) and `TSC_GROUPx_IOz` is the group of I/Os connected to $C_{sshield}$. (x = 1..8, z = 1..4 and z ≠ y).

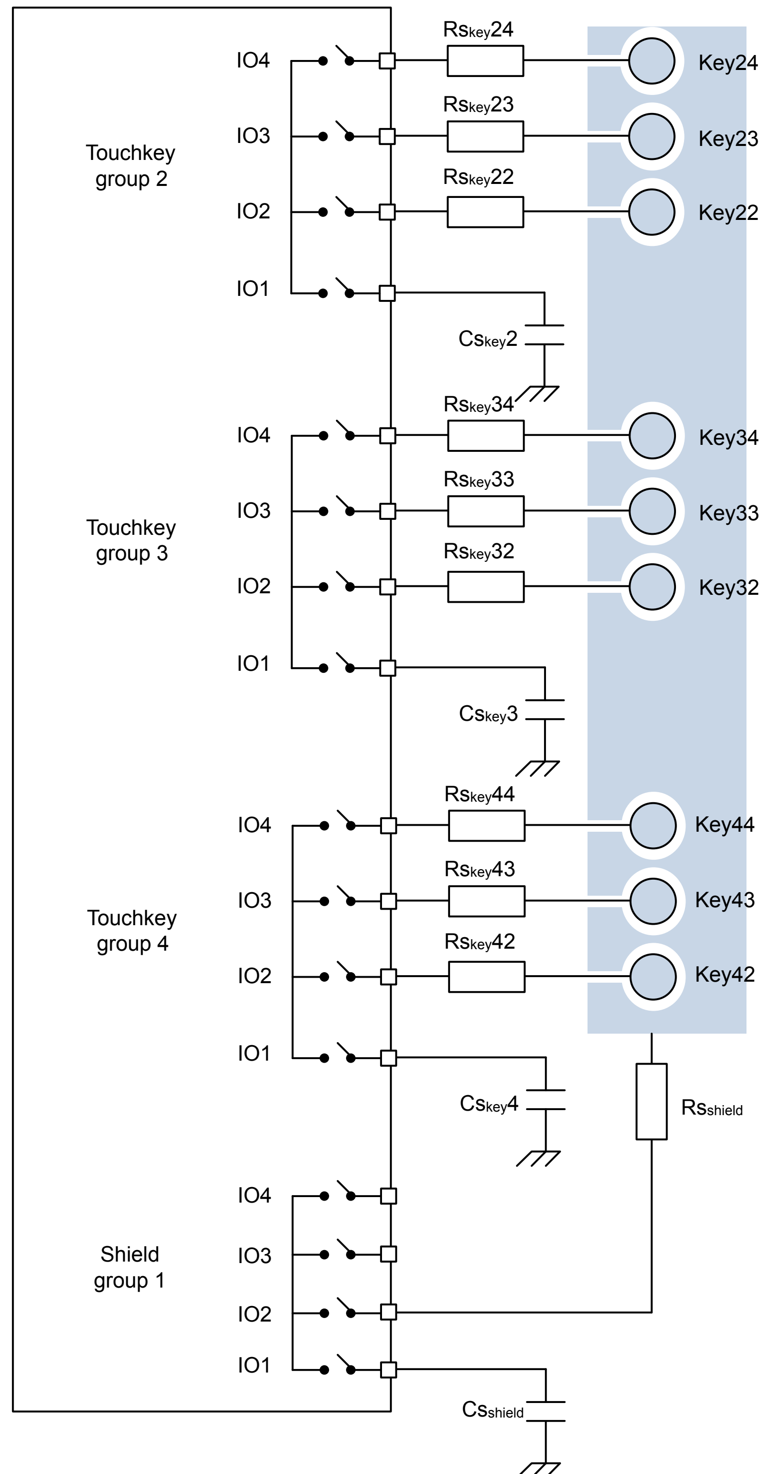
In *tsl_user.h* file from bank setting point of view, `SHIELD_IO_MSK` must be set as follows:

```
/* Shield IOs definition */
#define SHIELD_IO_MSK       (TSC_GROUPx_IOy)

/* Bank(s) definition */
#define BANK_0_NBCHANNELS (2)
#define BANK_0_MSK_CHANNELS   (CHANNEL_0_IO_MSK | CHANNEL_1_IO_MSK | SHIELD_IO_MSK)
#define BANK_0_MSK_GROUPS     (CHANNEL_0_GRP_MSK | CHANNEL_1_GRP_MSK)
```

Example

In this example, the configuration is the one given in the figure below.

Figure 13. **Active shield implementation example**



- Shield on group 1
  - G1_IO1 connect to $C_{sshield}$
  - G1_IO2 connect to $R_{sshield}$ and shield copper
- Sensors on group 2
  - G2_IO1 connect to $C_{skey}2$
  - G2_IO2 connect to $R_{skey}22$ and sensor (Key22)
  - G2_IO3 connect to $R_{skey}23$ and sensor (Key23)
  - G2_IO4 connect to $R_{skey}24$ and sensor (Key24)
- Sensors on group 3
  - G3_IO1 connect to $C_{skey}3$
  - G3_IO2 connect to $R_{skey}32$ and sensor (Key32)
  - G3_IO3 connect to $R_{skey}33$ and sensor (Key33)
  - G3_IO4 connect to $R_{skey}34$ and sensor (Key34)
- Sensors on group 4
  - G4_IO1 connect to $C_{skey}4$
  - G4_IO2 connect to $R_{skey}42$ and sensor (Key42)
  - G4_IO3 connect to $R_{skey}43$ and sensor (Key43)
  - G4_IO4 connect to $R_{skey}44$ and sensor (Key44)

To get the status of Key22 to Key44, the followings banks assignation is used:

- Bank1 to get Key22, Key32 and Key42 stateID
- Bank2 to get Key23, Key33 and Key43 stateID
- Bank3 to get Key24, Key34 and Key44 stateID

With this banks mapping, the waveforms on R_{sxx} and C_{sxx} are the ones given in the figures below.

Figure 14. Waveform on sampling capacitors

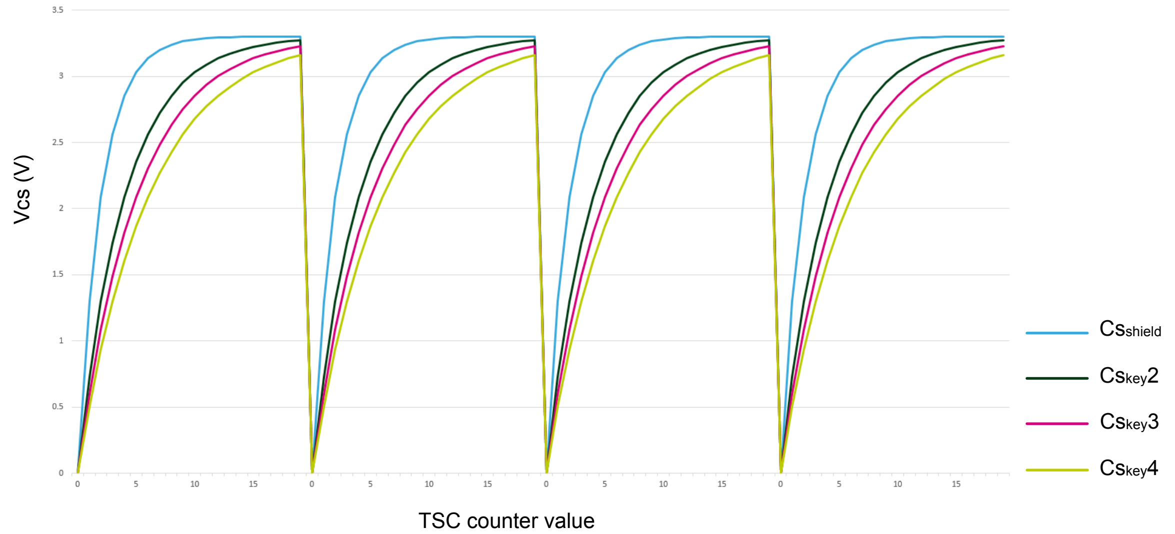
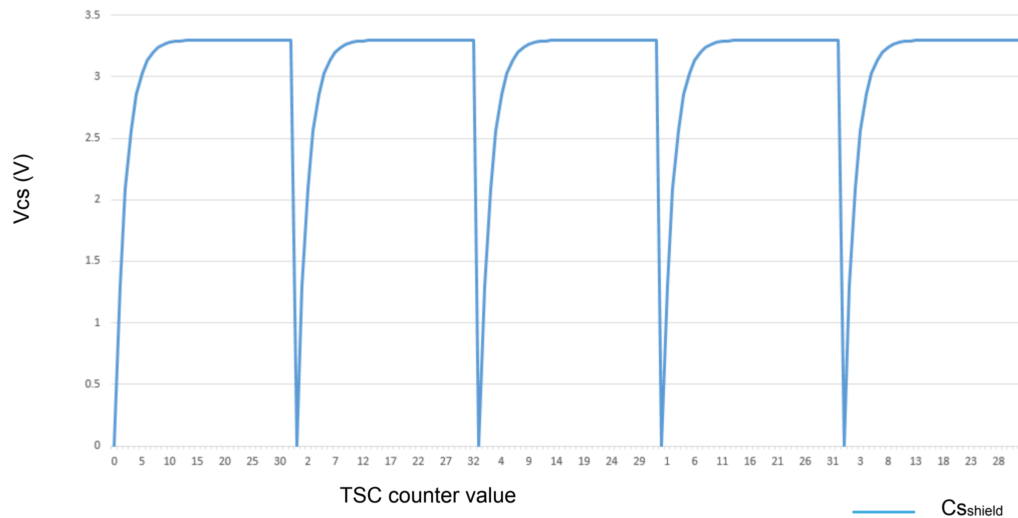


Figure 15. Waveform on $C_{sshield}$



To improve the conducted-noise immunity, the following R_S and C_S values give good performance:

- $C_{sshield}$ = 100 nF (to adjust according to previous explanations in this section)
- $R_{sshield}$ = 1 kΩ (to adjust according to previous explanations in this section)
- C_{skey}x = 47 nF
- R_{skey}xx = 10 kΩ

Figure 16. Waveform on G2_IO2, G3_IO2 and G3_IO2

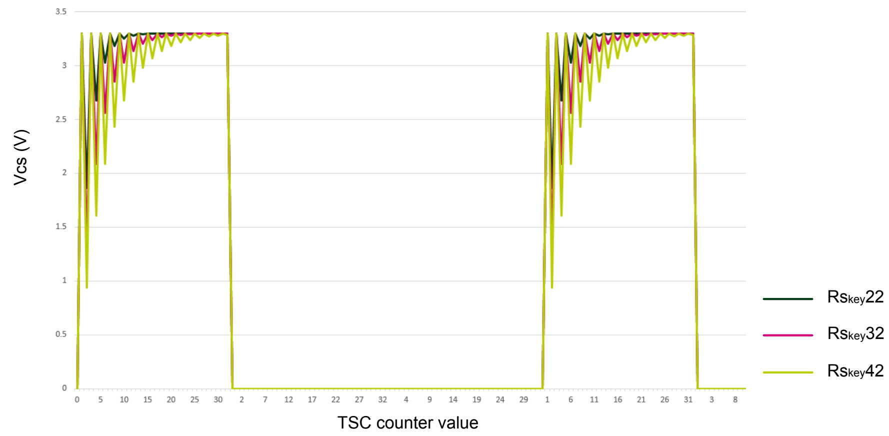


Figure 17. Waveform on G2_IO3, G3_IO3 and G3_IO3

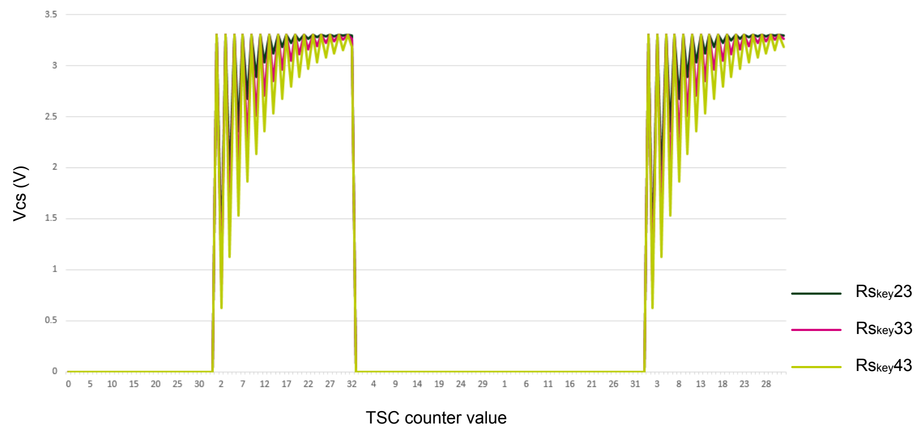
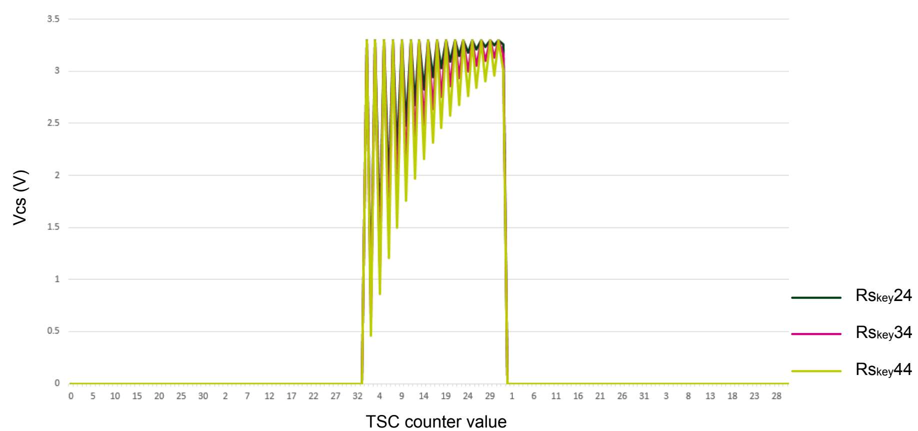


Figure 18. Waveform on G2_IO4, G3_IO4 and G3_IO4



5.3 Spread spectrum

Without spread spectrum, the main noise susceptibility is found at the acquisition frequency and its value is 1/TCD (TCD = transfer cycle duration).

The main frequency (HCLK) in the STM32 MCUs comes from the PLL output. Preferably the highest frequency recommended by specification is used to offer an optimum response time (example: 48 MHz for the STM32F0 series).

This frequency is divided in the TSC cell by the programmable prescaler (PGCLK). This frequency determines the basic timing units for CTPH, CTPL as follows:

Transfer cycle duration = (1/ (PGCLK) x ((CTPH + 1) + (CTPL + 1))) + (dead time = 2 x 1/ (HCLK)

with dead time = 2

By enabling the spread spectrum feature (with the SSE, spread spectrum enable), the noise susceptibility is distributed on multiple frequencies. This is done by adding HCLK timing units (period) to CTPH.

The SSD, spread spectrum deviation) allows the number of distributed frequencies to be set as follows:

from 0 (1 x t_{SSCLK}) to 127 (128 x t_{SSCLK})

Setting SSD to 32 gives usually good results.

The negative impact of this feature is the degradation of the acquisition speed and thus the response time. For example, SSD set to 127 adds an average of 64 x (1/ 48 MHz) = 1.33 µs to each count. For a 2000 counts acquisition duration, 2.6 ms is added due to spread spectrum activation.

Usually end users need a response time in less than 60 ms. Assuming the application uses three banks, the individual acquisition must be reported in less than 20 ms. If moreover a debounce filter is used (set to 2), this time constraint must be divided further by a factor of three. This leads to a maximum target time for one acquisition equal to 6.6 ms.

One acquisition time = count number x transfer cycle duration (see the transfer-cycle duration formula above)

To activate the spread-spectrum functionality, from software point of view, `TSL, HAL_TSC_Init` must be called with the following parameters:

```
htsc.Instance = TSC;
  …
  htsc.Init.SpreadSpectrum = ENABLE;
  htsc.Init.SpreadSpectrumDeviation = 32;
  htsc.Init.SpreadSpectrumPrescaler = TSC_SS_PRESC_DIV1;
  …
  if (HAL_TSC_Init(&htsc) != HAL_OK)
  {
    Error_Handler();
  }
```

where:

- `SpreadSpectrumDeviation` is SSD (from 1 to 127 when SSE = 1); typically set to 32
- `SpreadSpectrumPrescaler` is SSPSC (spread spectrum prescaler); can be divide-by-one or divide-by-two.

With frequency hopping, it is recommended to disable the spread spectrum for the following reasons:

- The spread spectrum decreases the impact when the noise is applied but this effect applies to a larger frequency spectrum.
- When using frequency hopping, only one of the acquisition frequencies must be impacted by the noise. It is important that only one acquisition frequency is impacted by one noise frequency.
- While the noise frequency is between the two acquisition frequencies, it must not impact both of them. It may be the case while the spread spectrum is enabled.

5.4 Channel blocking

The TSL parameter of the detection exclusion system (DXS) can be set with the code below, to avoid multiple detections and improve the system level safety:

```
/*============================================================================*/
/* Detection Exclusion System (DXS)                                           */
/*============================================================================*/

/** @defgroup Common_Parameters_DXS 12 - DXS
  * @{ */

/** Detection Exclusion System (0=No, 1=Yes)
*/
#define TSLPRM_USE_DXS (1)

/** @} Common_Parameters_DXS */
```

DXS can be set at group level also by setting the THIS_DXSLOCK object. Getting ghost or guard-ring sensors state level can be used to suspend on application level remaining sensors acquisition. Getting ghost or guard-ring sensors state level can be used to suspend the next sensors acquisition at application level.

5.5 Threshold adjustment

The two following thresholds can be adjusted:

- DETECT_IN: threshold to set a detection, recommended to be set to 2/3 of delta signal while touched with a normalized finger
- DETECT_OUT: threshold to reset a detection, set to 1/3 of delta signal

Example: if the delta when there is a touch is 150 counts, set TSLPRM_TKEY_DETECT_IN_TH to 100 and TSLPRM_TKEY_DETECT_OUT_TH to 50.

Those values can be adjusted knowing they are a compromise between the two following requirements:

- Avoid false detection on untouched adjacent key sensors.
- Avoid detection loss.

To handle the threshold functionality, from software point of view, two levels must be take into account:

- By default, all sensors threshold values are defined in *tsl_conf.h* file, with the code below.

```
/** TouchKeys Detect state input threshold (range=0..255)
  - Enter Detect state if delta is above
*/
#define TSLPRM_TKEY_DETECT_IN_TH (100)

/** TouchKeys Detect state output threshold (range=0..255)
  - Exit Detect state if delta is below
*/
#define TSLPRM_TKEY_DETECT_OUT_TH (50)
```

- Individual sensor threshold can be set, using TSL in *tsl_user.c*, with `tsl_user_SetThresholds api` call, as follows.

```
/**
  * @brief  Set thresholds for each object (optional).
  * @param  None
  * @retval None
  */
void tsl_user_SetThresholds(void)
{
/* USER CODE BEGIN Tsl_user_SetThresholds */
      /* Example: Adjust on sensor 0, 1 and 2 Detect thresholds */
      MyTKeys_Param[0].DetectInTh -= 10;
      MyTKeys_Param[0].DetectOutTh += 10;
      MyTKeys_Param[1].DetectInTh -= 8;
      MyTKeys_Param[1].DetectOutTh += 22;
      MyTKeys_Param[2].DetectInTh += 10;
      MyTKeys_Param[2].DetectOutTh += 20;
/* USER CODE END Tsl_user_SetThresholds */
  }
```

As all sensors do not have the same sensitivity, this feature can be used to adjust the sensor trigger level.

5.6 Software filter (debounce)

The software filter allows the reduction of false detections or detection losses. It is configured with `TSLPRM_DEBOUNCE_DETECT` and `TSLPRM_DEBOUNCE_RELEASE` parameters.

Setting the `TSLPRM_DEBOUNCE_DETECT` parameter to four or five, means that five consecutive acquisitions with a touch detected are needed to report a touch detection.

There is a trade-off. Increasing this parameter results in a longer response time between the moment when a user touch change occurs and when it is actually reported to the system.

To handle the debounce functionality, from software point of view, two levels must be taken into account:

- The conducted noise may increase or decrease the “measure, delta” values. The TSL library is able to handle these cases using the debounce feature (see `DETECT` and `RELEASE` defined in *tsl_conf.h*):

```
/*============================================================================*/
/* Debounce counters                                                          */
/*============================================================================*/

/** @defgroup Common_Parameters_Debounce 09 - Debounce counters
  * @{ */

/** Proximity state debounce in samples unit (range=0..63)
  - A Low value will result in a higher sensitivity during the Proximity detection
 but with less noise filtering.
  - A High value will result in improving the system noise immunity
 but will increase the system response time.
*/
#define TSLPRM_DEBOUNCE_PROX (2)

/** Detect state debounce in samples unit (range=0..63)
  - A Low value will result in a higher sensitivity during the detection
 but with less noise filtering.
  - A High value will result in improving the system noise immunity
 but will increase the system response time.
*/
#define TSLPRM_DEBOUNCE_DETECT (4)

/** Release state debounce in samples unit (range=0..63)
  - A Low value will result in a higher sensitivity during the end-detection
 but with less noise filtering.
  - A High value will result in a lower sensitivity during the end-detection
 but with more noise filtering.
*/
#define TSLPRM_DEBOUNCE_RELEASE (6)

/** Re-calibration state debounce in samples unit (range=0..63)
  - A Low value will result in a higher sensitivity during the recalibration
 but with less noise filtering.
  - A High value will result in a lower sensitivity during the recalibration
 but with more noise filtering.
*/
#define TSLPRM_DEBOUNCE_CALIB (4)

/** Error state debounce in samples unit (range=0..63)
  - A Low value will result in a higher sensitivity to enter in error state.
  - A High value will result in a lower sensitivity to enter in error state.
*/
#define TSLPRM_DEBOUNCE_ERROR (3)

/** @} Common_Parameters_Debounce */
```

- Individual sensor debounce can be set, using TSL in *tsl_user.c*, with the `tsl_user_SetThresholds api` call.

```
/**
  * @brief  Set thresholds for each object (optional).
  * @param  None
  * @retval None
  */
void tsl_user_SetThresholds(void)
{
/* USER CODE BEGIN Tsl_user_SetThresholds */
      /* Example: Increase the debounce detect values for the TKEY 0 */
      MyTKeys_Param[0]. CounterDebDetect += 2;
      MyTKeys_Param[0]. CounterDebRelease += 2;
/* We can do the same for CounterDebRelease, CounterDebError, CounterDebCalib and
CounterDebProx*/
/* USER CODE END Tsl_user_SetThresholds */
  }
```

- In order to pass the FTB (fast transient burst test) from IEC61000-4-4, a 15 ms extra delay must be added between each acquisition of the same touch key. Debounce value must be taken into account (see the figures below).

Figure 19. **With debounce = 1**

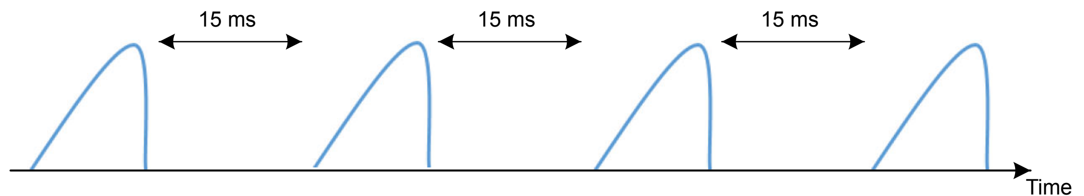
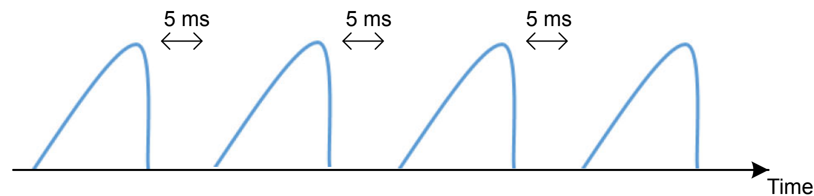


Figure 20. **With debounce = 3**



5.7 Schottky diodes on sensor and shield

Adding a low-capacitance Schottky diode close to the MCU, like an ST BAS70, increases the voltage level immunity versus the conducted noise:

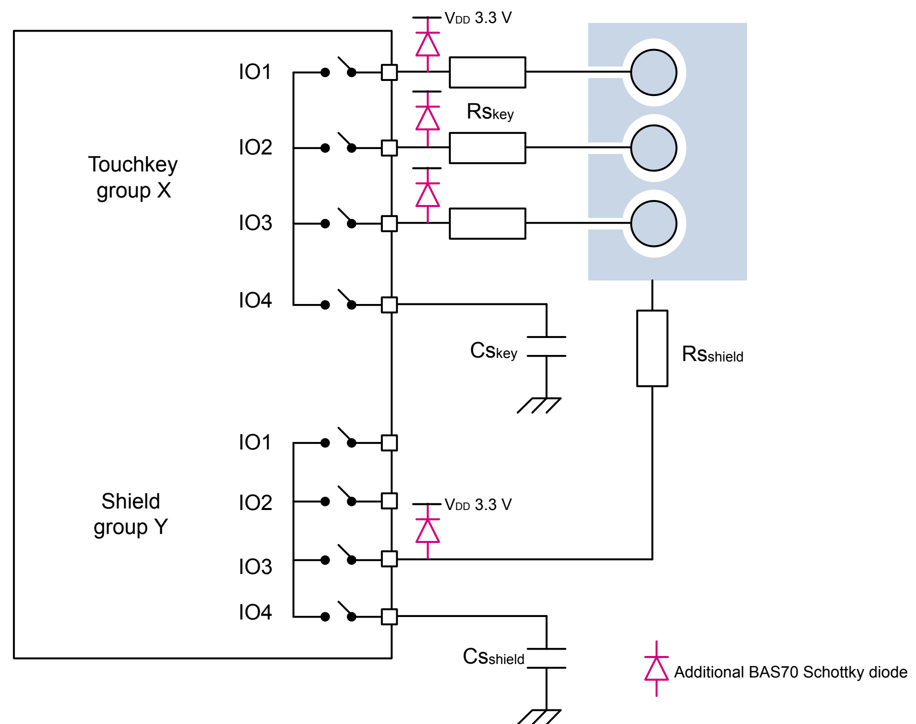
- On shield group Y, one Schottky diode must be added: single-diode packages BAS70ZFILM (SOD-123), BAS70JFILM (SOD-323) and BAS70KFILM (SOD-523) are the best choices.
- On touchkey group X, three Schottky diodes must be added: a three-parallel-diodes package, like BAS70-08SFILM (SOT-323-6L), or three single-diode packages, are the best choice.

Refer to the BAS70 datasheet for more details on the available packages.

Important: *These additional Schottky diodes must be used on all STM32 devices. Any unused GPIO must be set to Reset state or grounded.*

The figure below explains how to connect these extra Schottky diodes.

Figure 21. Schottky BAS70 diodes on sensor and shield

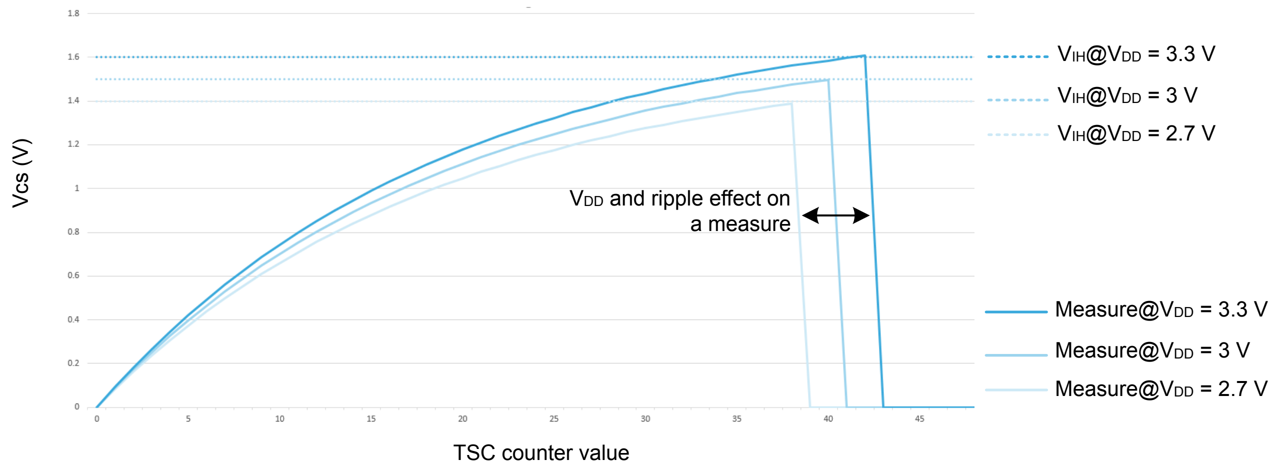


5.8 Reduce ripple on V_{DD}

The absolute value of any measure is impacted by V_{IH} (input high-level voltage). V_{IH} voltage level varies with V_{DD}. The ripple on V_{DD} impacts also the C_x (sensor capacitance) charged voltage.

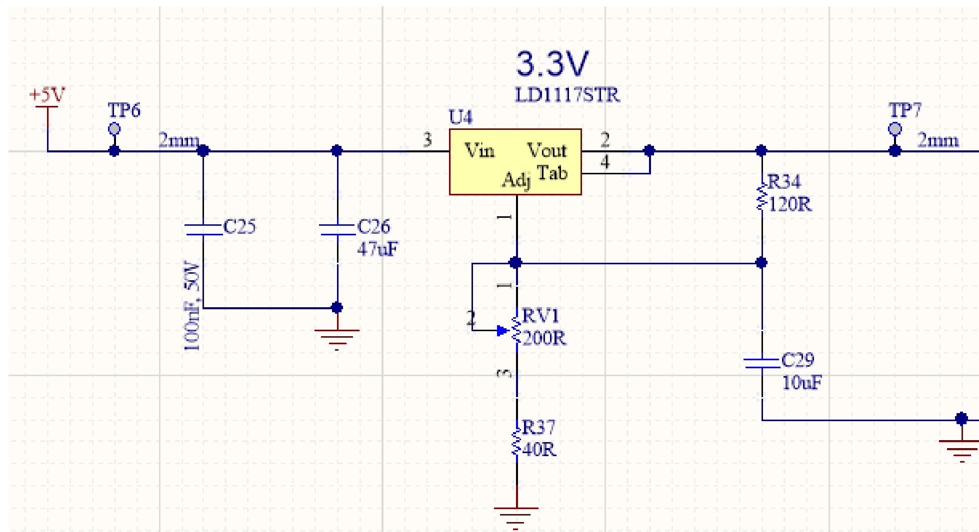
The figure below describes the impact of V_{DD} and ripple effect on a measure.

Figure 22. V_{DD} and ripple effect on a measure



To avoid unstable measurements, V_{DD} ripple and voltage level changes must be removed. A good example, using the LD1117 (an ST LDO), is to add a 10 µF capacitor on Vsense, as shown in the figure below (see the LD1117 datasheet for more details).

Figure 23. LDO application to improve ripple injection



Regarding low-speed V_{DD} voltage level changes in battery-powered systems (≥ 10 s), the ECS TSL feature is in charge to handle this use case. To activate the ECS feature by software, look at the following examples:

- Inside interrupt

```
/**
  * @brief This function handles Touch sensing controller interrupt.
  */
void TSC_IRQHandler(void)
{
  HAL_TSC_IRQHandler(&htsc);
  TSL_acq_BankGetResult(TSCidxGroup, 0, 0);
  TSCidxGroup++;
  if (TSCidxGroup > TSLPRM_TOTAL_BANKS-1)
  {
    TSCidxGroup=0;
    // Start process to handle all results, we do this after all bank acquisition (this
 is a choice)
    TSCAcqDone++;
  }else{
    // We restart bank acquisition only if we have more than 1 bank (this is a choice)
    TSL_acq_BankConfig(TSCidxGroup);
    TSL_acq_BankStartAcq_IT();
  }
}
```

- In main loop

```
int TSCidxGroup=0;
int TSCAcqDone=0;
#define TKEY_DET(NB) (MyTKeys[(NB)].p_Data->StateId == TSL_STATEID_DETECT)
#define TKEY_PRX(NB) (MyTKeys[(NB)].p_Data->StateId == TSL_STATEID_PROX)
#define TKEY_REL(NB) (MyTKeys[(NB)].p_Data->StateId == TSL_STATEID_RELEASE)
#define TKEY_CAL(NB) (MyTKeys[(NB)].p_Data->StateId == TSL_STATEID_CALIB)
extern __IO TSL_tTick_ms_T ECSLastTick;

int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TSC_Init();
  MX_TOUCHSENSING_Init();
  // Wait end of calibration on sensor 0
  while(TKEY_CAL(0))
  {
    tsl_user_Exec();
    HAL_Delay(20);
  }
  // Start First TSC acquisition
  TSCidxGroup=0;
  TSCAcqDone=0;
  TSL_acq_BankConfig(TSCidxGroup);
  TSL_acq_BankStartAcq_IT();
  __WFI();
  /* Infinite loop */
  while (1)
  {
    HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
    // Process TSC acquisition
    if(TSCAcqDone != 0){
      TSCAcqDone = 0;
      // Get TSC/TSL keys status
      TSL_obj_GroupProcess(&MyObjGroup);
      // Process DXS
      TSL_dxs_FirstObj(&MyObjGroup);
      // Process ECS algo
      if (TSL_tim_CheckDelay_ms(TSLPRM_ECS_DELAY, &ECSLastTick) == TSL_STATUS_OK)
      {
        TSL_ecs_Process(&MyObjGroup);
      }
      // activate led if key is press
      if(TKEY_DET(0)){
        HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_SET);}
      else if(TKEY_REL(0)){
        HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_RESET);
      }
      //FTB, or Fast Transient Burst immunity: wait for 15ms
      HAL_Delay(15);
      // After last bank acquisition we restart all bank acquisition
      TSCidxGroup = 0;
      TSCAcqDone = 0;
      TSL_acq_BankConfig(TSCidxGroup);
      TSL_acq_BankStartAcq_IT();
    }else{
      __WFI();
    }
  }
}
```

The ECS parameters or filters in *tsl_user.h*, are adjusted and activated with the code below.

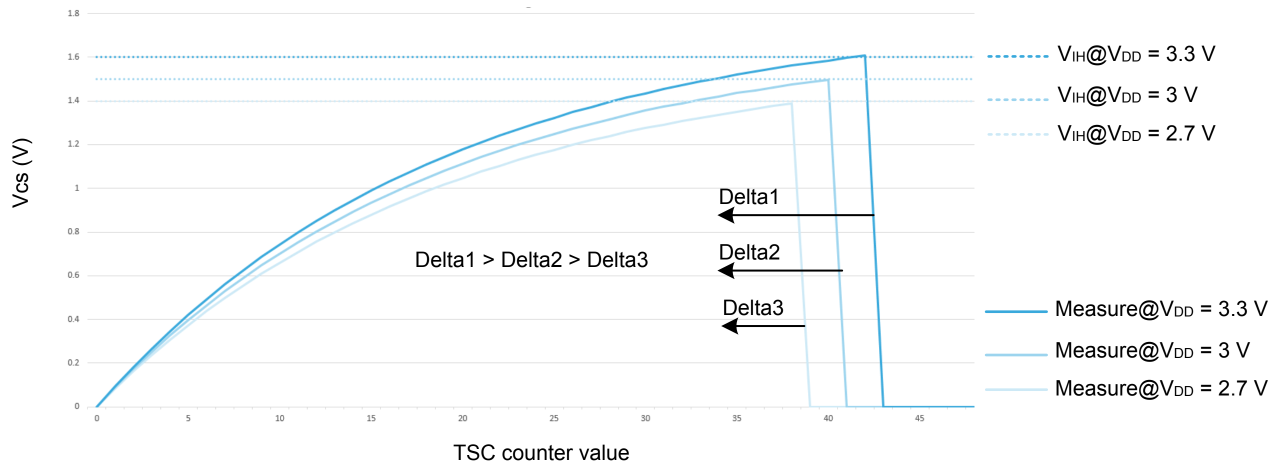
```
/*============================================================================*/
/* Environment Change System (ECS)                                            */
/*============================================================================*/

/** @defgroup Common_Parameters_ECS 10 - ECS
  * @{ */

/** Environment Change System Slow K factor (range=0..255)
  - The higher value is K, the faster is the response time.
*/
#define TSLPRM_ECS_K_SLOW (10)

/** Environment Change System Fast K factor (range=0..255)
  - The higher value is K, the faster is the response time.
*/
#define TSLPRM_ECS_K_FAST (20)

/** Environment Change System delay in msec (range=0..5000)
  - The ECS will be started after this delay and when all sensors are in Release state.
*/
#define TSLPRM_ECS_DELAY (500)

/** @} Common_Parameters_ECS */
```

5.9 Effect of V_{DD} level and V_{IH} range

As explain in the previous section, the absolute value of any measurement is impacted by V_{DD} and V_{IH}. The effect increases when V_{DD} increases, and decreases when V_{DD} decreases.

The figure below describes these effects on measures.

Figure 24. V_{DD} **and** V_{IH} **effect on a measure**



5.10 Frequency hopping

Another way to increase the noise immunity is to use the frequency-hopping algorithm. The charge-transfer frequency at the end of all groups acquisition can be changed by updating CTPH, CTPL, and DIV TSC registers fields. This spreads the charge-transfer frequency. Two, three or more frequency values can be used.

An example of this algorithm is given below. It can be used to update CTPH and CTPL at the end of each complete sensors acquisition. Cautions must be taken as the acquisition waveform must respect the criteria described previously:

- At TSC init stage, in `MX_TSC_Init`:
  - Define for this algorithm

```
// TSC Frequency Hopping
#define FREQ_VALUES     3
typedef struct{
    int div;
    int ctph;
    int ctpl;
}tsc_setup_t;
tsc_setup_t FreqHopp[FREQ_VALUES]={0};

TSC_HandleTypeDef htsc;
TSC_HandleTypeDef htsc_freqhopp;

#define CTPH_TO_INT(a)     ((uint32_t)( (a) >> 28) + 1)    // From 1 to 16
#define INT_TO_CTPH(a)     ((uint32_t)( (a-1) << 28))      // From 1 to 16
#define CTPL_TO_INT(a)     ((uint32_t)( (a) >> 24) + 1 )   // From 1 to 16
#define INT_TO_CTPL(a)     ((uint32_t)( (a-1) << 24))      // From 1 to 16
#define PRESC_TO_INT(a)    ((uint32_t)( (a) >> 12) + 1)    // 1/2/4/8/16/32/64/128
#define INT_TO_PRESC(a)    ((uint32_t)( (a-1) << 12))      // 1/2/4/8/16/32/64/128
```

  - Choose DIV, CTPH, and CTPL to get a stable measure on all sensors. These parameters are linked to the board layout, sampling capacitors and serial resistors. See Section 5.2: Active shield and section 'Charge transfer period tuning' in the application note *Tuning a touch sensing application on MCUs* (AN4316).

```
OptimumTscCfgCTPH=x;    //With:
OptimumTscCfgCTPL=x;    //x = 1 to 16
OptimumTscCfgDIV=y;     //y = 1 to 128 (power of 2)
```

  - Call TSC init

```
...
htsc.Init.CTPulseHighLength = INT_TO_CTPH(OptimumTscCfgCTPH);
htsc.Init.CTPulseLowLength = INT_TO_CTPL(OptimumTscCfgCTPL);
htsc.Init.SpreadSpectrum = DISABLE;
...
htsc.Init.PulseGeneratorPrescaler = INT_TO_PRESC(OptimumTscCfgDIV);
...
if (HAL_TSC_Init(&htsc) != HAL_OK)
...
```

  - Compute the three frequencies with this type of code sequence

```
for(int f=0; f<FREQ_VALUES; f++){
  if(f!=0){
    if(OptimunTscCfgCTPH+1 <=16 && OptimunTscCfgCTPL+1 <=16){
      OptimunTscCfgCTPH++;
      OptimunTscCfgCTPL++;
    }else{
      // We assume CTPH <= CTPL
      OptimunTscCfgDIV  *= 2;
      OptimunTscCfgCTPH  = 1 + (OptimunTscCfgCTPH/2);
      OptimunTscCfgCTPL  = 1 + (OptimunTscCfgCTPL/2);
    }
  }
  FreqHopp[f].div  = OptimunTscCfgDIV;
  FreqHopp[f].ctph = OptimunTscCfgCTPH;
  FreqHopp[f].ctpl = OptimunTscCfgCTPL;
}
```

- During `while(1)` loop

```
while (1)
{
  // Process TSC acquisition
  if(TSCAcqDone != 0){
    TSCAcqDone = 0;
    // Get TSC/TSL keys status
    TSL_obj_GroupProcess(&MyObjGroup);
    // Process DXS
    …
    // Process ECS algo
    …
    // activate led if key is press
    …
    //FTB, or Fast Transient Burst immunity: wait for 15ms
    …
    // We move to next frequency
    MoveToNextFreq();
    // After last bank acquisition we restart all bank acquisition
    TSCidxGroup = 0;
    TSCAcqDone = 0;
    TSL_acq_BankConfig(TSCidxGroup);
    TSL_acq_BankStartAcq_IT();
  }else{
    __WFI();
  }
}
```

- API used to change frequency after each acquisition

```
* Api
void MoveToNextFreq(void){
  static int f=0;
  f++;
  f %= FREQ_VALUES;

  // Change TSC frequency DIV divider
  htsc.Init.PulseGeneratorPrescaler = INT_TO_PRESC(FreqHopp[f].div);
  htsc.Init.CTPulseHighLength = INT_TO_CTPH(FreqHopp[f].ctph);
  htsc.Init.CTPulseLowLength  = INT_TO_CTPL(FreqHopp[f].ctpl);

  // Re Init TSC IP with New parameters...
  HAL_TSC_Init(&htsc);
  return;
}
```

Important: *The following points must be carefully checked:*

- *Measures on all sensors must be equal when all frequencies are used: when moving from F0 to F1, no changes must be observed on any sensor, from measure point of view.*
- *The frequencies must be far enough of each other, not a multiple.*
- *The same number of acquisitions at each frequency must be performed. Instead of three samples, it is recommended to use four samples to validate a touch detection. Two acquisitions at the two frequencies are performed.*

5.11 C_S and R_S

Increasing C_S, sampling capacitor, increases the acquisition time but decreases the conducted-noise effects.

The configuration R_{skey} = 10 kΩ and C_{skey} = 47 nF gives good results on the immunity to conducted-noise.

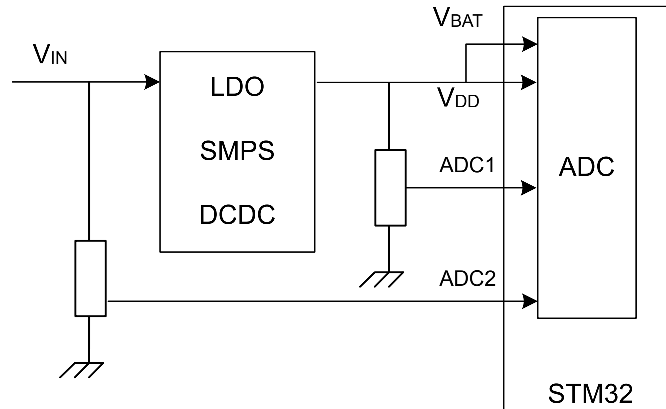
See section 'Hardware timing' of the application note *Tuning a touch sensing application on MCUs* (AN4316) for more details R_{skey} and C_{skey}.

See section 'Hardware timing' of the application note *Guidelines for designing touch sensing applications with projected sensors* (AN4313) for more details $R_{sshield}$ and $C_{sshield}$.

5.12 Noise or ripple measurement on V_{DD}

On application using the STM32 general purpose ADC, the V_{IN} ripple can be measured and the user can decide to stop touch sensing acquisition. The same measurement can be done on V_{DD} side (see the figure below).

Figure 25. V_{IN} ripple measurement on V_{DD}



Regarding V_{DD} decoupling, additional 10 nF and 1 nF capacitors can be used. This decoupling method spreads the decoupling capacitor impedance and increases the decoupling noise rejection range.

Figure 26. Decoupling capacitors on V_{DD}

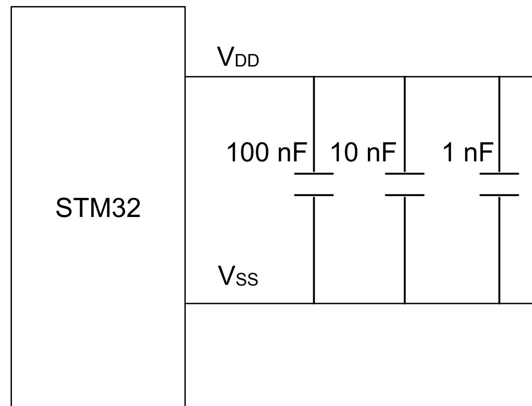
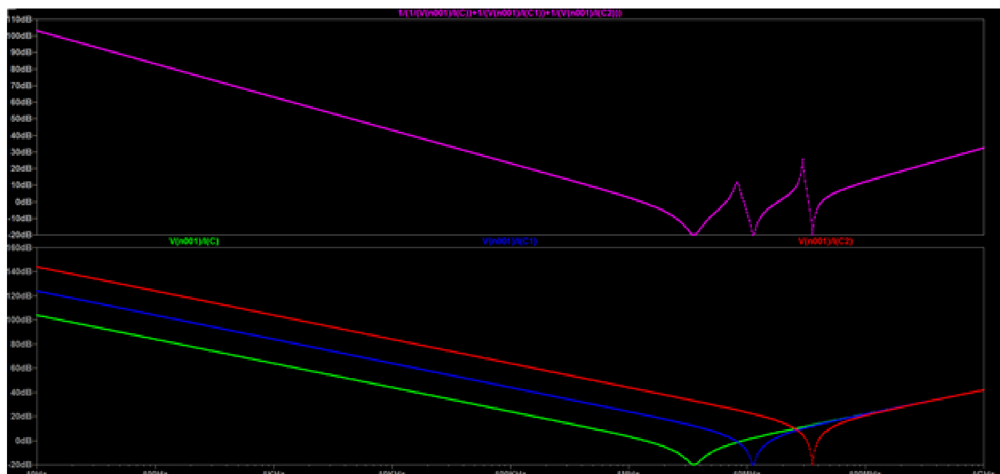


Figure 27. ESR behavior



5.13 Application level

5.13.1 Using DXS (detection exclusion system)

TSL provides a way to avoid multiple detections. This feature is activated by setting USE_DXS as shown below.

```
/*============================================================================*/
/* Detection Exclusion System (DXS)                                           */
/*============================================================================*/

/** @defgroup Common_Parameters_DXS 12 - DXS
  * @{ */

/** Detection Exclusion System (0=No, 1=Yes)
*/
#define TSLPRM_USE_DXS (1)

/** @} Common_Parameters_DXS */
```

5.13.2 Using TSL stateID sensors status

Usually all sensors are in `TSL_STATEID_RELEASE` state. When a sensor is pressed, its state moves to `TSL_STATEID_DETECT`.

Unusual system behavior can be detected using `SensorNbInStateId` API as shown below.

```
/**
* @brief  Count Sensor in stateId
* @param  StateId Sensor state id, can NOT be OR-ed.
* @retval Sensor number in StateId state
*/
int SensorNbInState(TSL_StateId_enum_T StateId){
  int   cntStateId=0,cntTKey=0;
  // Parse TKey sensors
#if TSLPRM_TOTAL_TOUCHKEYS > 0
  for (cntTKey=0;cntTKey<TSLPRM_TOTAL_TOUCHKEYS;cntTKey++){
    if(MyTKeys[cntTKey].p_Data->StateId == StateId){
      cntStateId++;
    }
  }
#endif
  // Parse LinRots sensors
#if TSLPRM_TOTAL_LINROTS > 0
  for (cntTKey=0;cntTKey<TSLPRM_TOTAL_LINROTS;cntTKey++){
    if(MyLinRots[cntTKey].p_Data->StateId == StateId){
      cntStateId++;
    }
  }
#endif
  return cntStateId;
}
```

Inside the main loop, sensors not in RELEASE state are counted and the acquisition is stopped if more than n sensors are not in a stable state. This may occur in case of incoming noise or when hands touch all sensors simultaneously.

The algorithm detailed below, added to DXS, improves the system noise immunity.

```
#define NOISE_COMPUTE_WINDOWS 10      /* Use to monitor noise */
#define NOISE_COMPUTE_THRESHOLD 20       /* Threshold level to decide when noise is too high
*/
#define NOISE_SUSPEND_DELAY 5000         /* Suspend delay when we detect noise (5s) */
#define NOISE_SENSOR_NOT_IN_RELEASE 3   /* Suspend acquisition when sensors are not
in release state */
while (1)
{
if(TSCAcqDone != 0 && WaitEndOfNoiseDuration == 0){
  TSCAcqDone = 0;

  // Get TSC/TSL keys status
  TSL_obj_GroupProcess(&MyObjGroup);

  // Point to DXS first object
  // Process ECS algo
  // Application sensor processing
  // For FTB test, wait for 15ms
  // Handle Frequency Hopping

  // Handle noise on application level
  nbSensorInRelease = SensorNumberInState(TSL_STATEID_RELEASE);

  // Handle next acquisition
  if((((TSLPRM_TOTAL_TOUCHKEYS+TSLPRM_TOTAL_LINROTS)-nbSensorInRelease)
 >= NOISE_SENSOR_NOT_IN_RELEASE)){
    // Noise detect, we suspend sensors acquisition
    WaitEndOfNoiseDuration = NOISE_SUSPEND_DELAY;
  }else{
    // After last bank acquisition we restart all bank acquisition
    TSCidxGroup = 0;
    TSCAcqDone = 0;
    TSL_acq_BankConfig(TSCidxGroup);
    TSL_acq_BankStartAcq_IT();
  }
}else{
  __WFI();
  // We assume only 1ms tick interrupt occurs
  if(WaitEndOfNoiseDuration > 0) {
    WaitEndOfNoiseDuration--;
    if(WaitEndOfNoiseDuration==0){
      nbSensorInRelease = 0;
      WaitEndOfNoiseDuration = 0;
      // We restart all acquisition
      TSCidxGroup = 0;
      TSCAcqDone = 0;
      TSL_acq_BankConfig(TSCidxGroup);
      TSL_acq_BankStartAcq_IT();
    }
  }
}
```

5.13.3 Using noise level measurement on delta

The `GetNoiseLevel` API can be used to get median noise level on all channels and decide to stop system when a value is too high. This API must be combined with the `StateNbInSate` one.

```
/**
* @brief  Get noise level
* @param  None
* @retval mean noise level
*/
int phaseMeasurementNoise=0;
int GetNoiseLevel(void){
    int meanNoiseLevel=0, noiseLevelTKEYS=0, noiseLevelLROT=0;
    uint32_t id = 0, channel=0, div=0;

    //Reset array
```

```
    if(phaseMeasurementNoise==0){
#if TSLPRM_TOTAL_TOUCHKEYS > 0
        for (channel=0;channel<TSLPRM_TOTAL_TOUCHKEYS;channel++){
            for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
                NoiseTKEYS[channel][id] = 0;
            }
        }
#endif
#if TSLPRM_TOTAL_LINROTS > 0
        for (channel=0;channel<TSLPRM_TOTAL_LINROTS;channel++){
            for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
                NoiseLROT[channel][id] = 0;
            }
        }
#endif
    }
    phaseMeasurementNoise++;
    //Start to measure noise, level.....
#if TSLPRM_TOTAL_TOUCHKEYS > 0
    // Parse TKey sensors
    for (channel=0;channel<TSLPRM_TOTAL_TOUCHKEYS;channel++){
        NoiseTKEYS[channel][phaseMeasurementNoise%NOISE_COMPUTE_WINDOWS]
=MyTKeys[channel].p_ChD->Delta;
    }
#endif
#if TSLPRM_TOTAL_LINROTS > 0
    // Parse LinRots sensors
    for (channel=0;channel<TSLPRM_TOTAL_LINROTS;channel++){
        NoiseLROT[channel][phaseMeasurementNoise%NOISE_COMPUTE_WINDOWS]
=MyLinRots[channel].p_ChD->Delta;
    }
#endif

    // Compute noise inside the rolling windows
#if TSLPRM_TOTAL_TOUCHKEYS > 0
    for (channel=0;channel<TSLPRM_TOTAL_TOUCHKEYS;channel++){
        for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
            noiseLevelTKEYS += NoiseTKEYS[channel][id];
        }
    }
    noiseLevelTKEYS = abs(noiseLevelTKEYS);
    noiseLevelTKEYS /= NOISE_COMPUTE_WINDOWS;
    div += TSLPRM_TOTAL_TOUCHKEYS; // Will be used to compute noise mean level
#endif
#if TSLPRM_TOTAL_LINROTS > 0
    for (channel=0;channel<TSLPRM_TOTAL_LINROTS;channel++){
        for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
            noiseLevelLROT += NoiseLROT[channel][id];
        }
    }
    noiseLevelLROT = abs(noiseLevelLROT);
    noiseLevelLROT /= NOISE_COMPUTE_WINDOWS;
    div += TSLPRM_TOTAL_LINROTS; // Will be used to compute noise mean level
#endif

// For debug purpose
//#if TSLPRM_TOTAL_TOUCHKEYS > 0
//    for (channel=0;channel<TSLPRM_TOTAL_TOUCHKEYS;channel++){
//                printf("Sensor TKEY %02d Delta(s) :",channel);
//        for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
//            printf(" %4d",NoiseTKEYS[channel][id]);
//        }
//      printf("\n");
//    }
//#endif
//#if TSLPRM_TOTAL_LINROTS > 0
//    for (channel=0;channel<TSLPRM_TOTAL_LINROTS;channel++){
//                printf("Slider LROT %02d Delta(s) :",channel);
//        for (id=0;id<NOISE_COMPUTE_WINDOWS;id++){
//            printf(" %4d",NoiseLROT[channel][id]);
```

```
//        }
//      printf("\n");
//    }
//#endif

    // mean noise level take into account keys and sliders
    meanNoiseLevel = (noiseLevelTKEYS + noiseLevelLROT)/div;
    printf("==> Noise ..... noiseLevelTKEYS %d noiseLevelLROT %d [meanNoiseLevel %d]\n",
noiseLevelTKEYS,noiseLevelLROT,meanNoiseLevel);

    return meanNoiseLevel;
}
```

Inside the main loop, noise is measured at the end of all groups acquisition. It is important to get the noise value only when all sensors are in `RELEASE` state.

The algorithm detailed below, added to DXS, improves the system noise immunity.

```
#define NOISE_COMPUTE_WINDOWS  10 /* Use to monitor noise */
#define NOISE_COMPUTE_THRESHOLD  20 /* Threshold level to decide when noise is too high */
#define NOISE_SUSPEND_DELAY  5000 /* Suspend delay when we detect noise (5s) */
#define NOISE_SENSOR_NOT_IN_RELEASE  3 /* Suspend acquisition when sensors are not
in release state */
while (1)
{
if(TSCAcqDone != 0 && WaitEndOfNoiseDuration == 0){
  TSCAcqDone = 0;

  // Get TSC/TSL keys status
  TSL_obj_GroupProcess(&MyObjGroup);

  // Point to DXS first object
  // Process ECS algo
  // Application sensor processing
  // For FTB test, wait for 15ms
  // Handle Frequency Hopping

  // Handle noise on application level
  nbSensorInRelease = STMTBOX_SensorNumberInState(TSL_STATEID_RELEASE);
  // We compute noise level only if all sensors are in Release state
  if(nbSensorInRelease==(TSLPRM_TOTAL_TOUCHKEYS+TSLPRM_TOTAL_LINROTS)){
    noiseLevel = STMTBOX_GetNoiseLevel();
    if (noiseLevel >= NOISE_COMPUTE_THRESHOLD)
      noiseIsDetected = 1;
    else
      noiseIsDetected = 0;
  }

  // Handle next acquisition
  if((((TSLPRM_TOTAL_TOUCHKEYS+TSLPRM_TOTAL_LINROTS)-nbSensorInRelease)
>= NOISE_SENSOR_NOT_IN_RELEASE) || (noiseIsDetected)){
    // Noise detect, we suspend sensors acquisition
    WaitEndOfNoiseDuration = NOISE_SUSPEND_DELAY;
  }else{
    // After last bank acquisition we restart all bank acquisition
    TSCidxGroup = 0;
    TSCAcqDone = 0;
    TSL_acq_BankConfig(TSCidxGroup);
    TSL_acq_BankStartAcq_IT();
  }
}else{
  __WFI();
  // We assume only 1ms tick interrupt occurs
  if(WaitEndOfNoiseDuration > 0) {
    WaitEndOfNoiseDuration--;
    if(WaitEndOfNoiseDuration==0){
      nbSensorInRelease = 0;
      WaitEndOfNoiseDuration = 0;
      noiseIsDetected = 0;
      // We restart all acquisition
      TSCidxGroup = 0;
      TSCAcqDone = 0;
      TSL_acq_BankConfig(TSCidxGroup);
      TSL_acq_BankStartAcq_IT();
    }
  }
}
```

5.13.4 Using the internal ADC to detect noise on V_{DD}

Examples of software reading the ADC value are available in STM32CubeMX deliveries. Windowing can be used to trig the interrupts. Under the ADC interrupt routine, `WaitEndOfNoiseDuration` must be set to match with the algorithm detailed in the previous section.

```
// Noise detect, we suspend sensors acquisition
    WaitEndOfNoiseDuration = NOISE_SUSPEND_DELAY;
```

Examples:

- ./STM32Cube_FW_F0_Vx/Projects/STM32F091RC-Nucleo/Examples/ADC/ADC_AnalogWatchdog/readme.txt
- ./STM32Cube_FW_F0_Vx/Projects/STM32F072RB-Nucleo/Examples_LL/ADC/ADC_MultiChannelSingleConversion/readme.txt

6 STM32303E-EVAL board example

6.1 Firmware

The firmware used with the STM32303E-EVAL is the STM32F303_Ex01_2TKeys_EVAL and it belongs to the STM32F3xx STMTouch library. The results provided in this section are obtained with the configuration values presented in the table below.

Table 3. User configuration settings related to immunity improvements

Parameter	Configuration value
HCLK	48 MHz
TSLPRM_TKEY_DETECT_IN_TH	100
TSLPRM_TKEY_DETECT_OUT_TH	50
TSLPRM_DEBOUNCE_DETECT	2
TSLPRM_TSC_CTPH	1
TSLPRM_TSC_CTPL	1
TSLPRM_TSC_PGPSC	5
TSLPRM_TSC_USE_SS	1
TSLPRM_TSC_SSD	127

6.2 Performance

The performances of an STM32303E-EVAL board, with the configuration described above, are the following:

- Acquisition performed in 1200 counts
- Acquisition duration: 4.7 ms (target < 6.6 ms)

6.3 Conducted noise evaluation

The conducted noise evaluation results performed on the STM32303E-EVAL board according to IEC61000-4-6 standard is above 3 V_{rms} class A,

with the following test conditions:

- Frequency range from 150 kHz to 80 MHz
- 1 % frequency steps
- Dwell time 0.5 s

On worst case bandwidth, no false detection or loss is observed up to 4 V_{rms}, from 200 to 400 kHz with 100 Hz steps.

7 Conclusion

The touch sensing controller peripheral of the STM32 devices shows a high noise immunity level in compliance with IEC61000-4-6 standard (above 3 V_{rms} class A). These results can easily be reached by putting in practice the following recommendations:

- Implement an active shield electrode and enable the active shield feature in the firmware.
- Optimize the detection thresholds.
- Use the debounce filter.
- Use frequency hopping.
- Use application filtering based on ripple, noise measurement and guard-ring or ghost detections.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
03-Jul-2013	1	Initial release.
11-Jun-2014	2	Added support for STM32L0 Series. Updated last bullet in Section 4.1. Proposed improvement techniques.
22-Oct-2015	3	Added support for STM32L4 Series.
14-Mar-2018	4	Updated: • Figure 2. Injected signal • Figure 5. Data processing Added Section1. General information.
18-Jan-2019	5	Updated: • Title of the document • Table 1. Applicable products
22-Oct-2019	6	Updated: • Table 1. Applicable products • Section 3 • Section 5.1 , Section 5.2 , Section 5.5 and Section 5.6 • Section 6 STM32303E-EVAL board example Added: • Section 3.1 Sampling capacitor (Cs) voltage level without noise • Section 3.2 Sampling capacitor (Cs) voltage level with noise • Figure 9. Process flow • Section 5.4 to Section 5.13
11-Aug-2021	7	Updated the Applicable products table.
10-Jan-2023	8	Updated the Table 1. Applicable products in the Section Introduction to incorporate the STM32WBA series. Updated the whole document with minor changes.
06-Mar-2024	9	Added STM32U0 series Updated document title
03-Dec-2024	10	Added STM32U3 series

Contents

1 **General information** . . . 2
2 **Conducted noise immunity** . . . 3
2.1 Noise immunity . . . 3
2.2 IEC61000-4-6 standard . . . 3
2.2.1 Standard IEC61000-4-6 test setup . . . 3
2.2.2 Injected signal characteristics . . . 4
2.2.3 Noise immunity evaluation . . . 4
2.2.4 IEC61000-4-6 standard limitation . . . 4
3 **Surface charge transfer acquisition principle overview** . . . 5
3.1 Sampling capacitor (C_S) voltage level without noise . . . 6
3.2 Sampling capacitor (Cs) voltage level with noise . . . 7
4 **Test set up proposal to detect worst case** . . . 8
4.1 Test setup . . . 8
4.2 Generator settings . . . 8
4.3 Data logging and data processing . . . 9
5 **How to improve noise immunity** . . . 10
5.1 Proposed improvement techniques . . . 10
5.2 Active shield . . . 12
5.3 Spread spectrum . . . 20
5.4 Channel blocking . . . 21
5.5 Threshold adjustment . . . 21
5.6 Software filter (debounce) . . . 23
5.7 Schottky diodes on sensor and shield . . . 25
5.8 Reduce ripple on V_{DD} . . . 26
5.9 Effect of V_{DD} level and V_{IH} range . . . 30
5.10 Frequency hopping . . . 30
5.11 C_S and R_S . . . 32
5.12 Noise or ripple measurement on V_{DD} . . . 33
5.13 Application level . . . 34
5.13.1 Using DXS (detection exclusion system) . . . 34
5.13.2 Using TSL stateID sensors status . . . 34
5.13.3 Using noise level measurement on delta . . . 35
5.13.4 Using the internal ADC to detect noise on V_{DD} . . . 39
6 **STM32303E-EVAL board example** . . . 40

6.1 Firmware 40
6.2 Performance 40
6.3 Conducted noise evaluation. 40
7 Conclusion 41
Revision history 42
List of tables 45
List of figures. 46

List of tables

Table 1. Applicable products 1
Table 2. Test levels. 4
Table 3. User configuration settings related to immunity improvements 40
Table 4. Document revision history 42

List of figures

Figure 1. Standard IEC61000-4-6 test setup . . . 3
Figure 2. Injected signal . . . 4
Figure 3. Charge transfer equivalent capacitance model . . . 5
Figure 4. Vcs behavior without noise . . . 6
Figure 5. Vcs behavior with high-frequency conducted noise . . . 7
Figure 6. Vcs behavior with low-frequency conducted noise . . . 7
Figure 7. Test condition . . . 8
Figure 8. Data processing . . . 9
Figure 9. Process flow . . . 10
Figure 10. Active shield . . . 12
Figure 11. Electrode and active shield waveforms . . . 13
Figure 12. Waveform detail . . . 14
Figure 13. Active shield implementation example . . . 16
Figure 14. Waveform on sampling capacitors . . . 18
Figure 15. Waveform on $C_{sshield}$ . . . 18
Figure 16. Waveform on G2_IO2, G3_IO2 and G3_IO2 . . . 19
Figure 17. Waveform on G2_IO3, G3_IO3 and G3_IO3 . . . 19
Figure 18. Waveform on G2_IO4, G3_IO4 and G3_IO4 . . . 19
Figure 19. With debounce = 1 . . . 24
Figure 20. With debounce = 3 . . . 24
Figure 21. Schottky BAS70 diodes on sensor and shield . . . 25
Figure 22. V_{DD} and ripple effect on a measure . . . 26
Figure 23. LDO application to improve ripple injection . . . 26
Figure 24. V_{DD} and V_{IH} effect on a measure . . . 30
Figure 25. V_{IN} ripple measurement on V_{DD} . . . 33
Figure 26. Decoupling capacitors on V_{DD} . . . 33
Figure 27. ESR behavior . . . 33

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2024 STMicroelectronics – All rights reserved