RX Family

QE Touch module Firmware Integration Technology

Introduction

This application note describes the TOUCH Module.

Target Device

- RX113 Group
- RX130 Group
- RX230 Group
- RX231 Group
- RX23W Group
- RX671 Group
- RX140 Group

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Related Documents

Firmware Integration Technology User's Manual (R01AN1833)
Board Support Package Firmware Integration Technology Module (R01AN1685)
Adding Firmware Integration Technology to Projects (R01AN1723)
RX100 Series VDE Certified IEC60730 Self-Test Code (R01AN2061ED)
RX v2 Core VDE Certified IEC60730 Self-Test Code for RX v2 MCU (R01AN3364EG)

Contents

1. Overview

The TOUCH Module is middleware that uses the CTSU module to provide capacitive touch detection. The TOUCH module assumes access from the user application is possible.

1.1 Functions

The TOUCH module supports the following functions.

1.1.1 QE for Capacitive Touch Usage

Similar to the CTSU module, this module provides various capacitive touch detections based on configuration settings generated by QE for Capacitive Touch (referred to as QE)

As a part of the configuration settings, the touch interface configuration displays configuration information for the CTSU link information and buttons, sliders, and wheels. A multiple touch interface configuration is necessary when both self and mutual capacitance buttons are used in the same product or when using the active shield function.

This module also supports the QE monitor function. The monitor determines whether to use debugger or serial communications, determines the type of the information from QE and sends only the necessary information.

1.1.2 Measurements and Data Processing

The module determines whether the button has been touched based on the change in capacitance and detects the position of the slider or wheel. This requires continued periodic measurements of capacitance. When developing your application, make sure to periodically call R_TOUCH_ScanStart() and R_TOUCH_DataGet(). For more details, refer to the sample application.

1.1.3 Button Touch Determination

(a) Creating reference value and threshold

A touch button is not a mechanical button in which the ON/OF state is switched by hardware. The ON/OFF state is determined via software.

First, a reference value is created based on measurement results in the non-touch state. The initial reference value is the first measured value. The threshold is then determined with an arbitrary offset. If a measured value exceeds the threshold, the button is determined to be in the ON state, if it does not exceed the threshold, it is in the OFF state.

Processing for self-capacitance and mutual capacitance are basically the same. However, because the amount of capacitance decreases when a mutual capacitance button is touched, the user needs to set the threshold based on decreasing measured values to determine the ON/OFF state.

You can set the threshold for each button separately in the configuration settings (threshold in touch_button_cfg). The following functions are also included to deal with issues such as chattering suppression and changes in the external environment which affect actual touch recognition.

(b) Positive Noise Filter/Negative Noise Filter

As a chattering countermeasure, you can confirm the ON/OFF state after a set number of consecutive ON or OFF determinations.

In the configuration settings (on_freq and off_freq in touch_cfg_t) set the number of consecutive ON or OFF states. You can do this for all buttons in the touch interface configuration. Be aware that, although this is an effective solution to improving chattering, the greater the number of consecutive states, the slower the response to actual touch.

(c) Hysteresis

This is another chattering countermeasure. Offset the constant to the threshold after the state goes to ON, and prevent chattering by using hysteresis as the OFF-to-ON and ON-to-OFF threshold.

You can set the hysteresis value for each button in the configuration settings (hysteresis in touch_button_cfg_t). The larger the hysteresis, the more effective the countermeasure is in suppressing chattering. However, keep in mind that this will make it more difficult to return the state from ON-to-OFF of OFF-to-ON.

(d) Chattering suppression type (Build option)

This build option is a function to supplement the above functions (Positive Noise Filter/Negative Noise Filter and Hysteresis) for performing touch judgment.

This build option changes the processing method for Counter of exceed threshold to TypeA or TypeB.

TypeA of chattering suppression : Counter of exceed threshold is hold within hysteresis range.

TypeB of chattering suppression : Counter of exceed threshold is reset within hysteresis range.

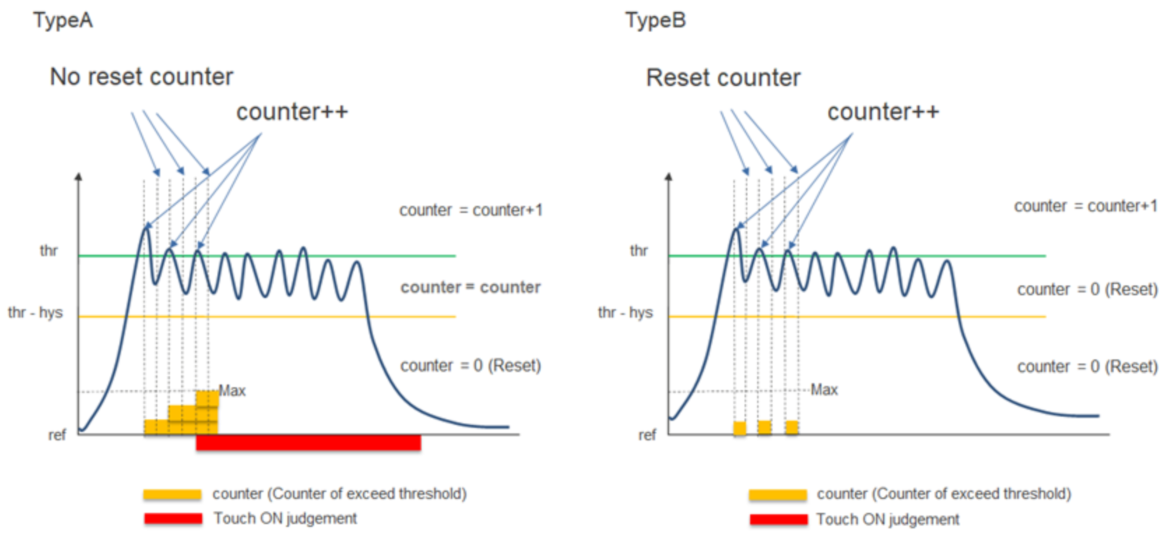


Figure 1 Example of chattering suppression operation

(e) Drift Correction Process

As a countermeasure for changes in the external environment, the drift correction process refreshes the reference value.

After averaging the measured value in the OFF state over a set period, if the button is in the touch OFF state after a set period, the reference value is refreshed. The drift correction is only executed in the OFF state and is cleared when touch ON is determined.

Set the period in the configuration settings (drift_freq in touch_cfg_t). You can do this for all buttons in the touch interface configuration. This allows you to adjust the ability to determine the touch state despite changes in the external environment.

Figure 2 shows an example of the drift correction process.

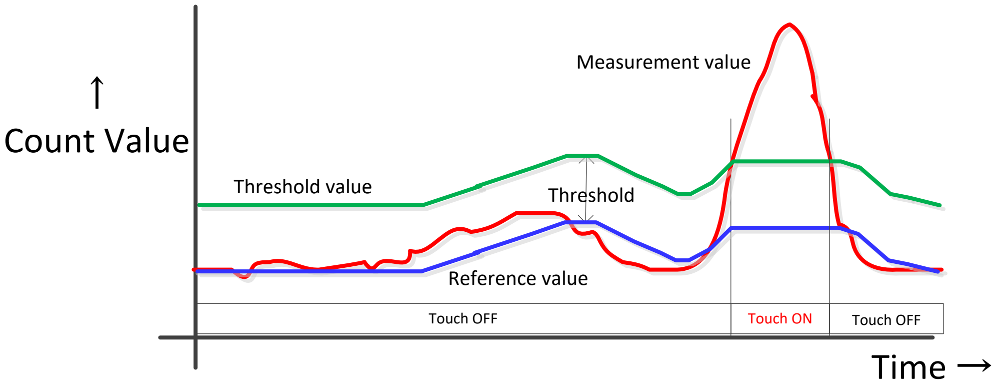


Figure 2 Button Touch Determination

(f) Press and hold cancel

Strong noise or other sudden environment changes can disable the drift correction process, preventing return from the ON state. The press and hold cancel function implements the drift correction process and returns the button from the ON state by forcibly turning the state to OFF after a certain number of consecutive ON state periods.

Set the number of consecutive ON periods required for the press and hold cancel function to return the button to the OFF state in the configuration settings (cancel_freq in touch_cfg_t). You can do this for all buttons in the touch interface configuration.

1.1.4 Touch Position Detection of Slider/Wheel

Configure a slider with multiple terminals to be measured (TS) physically arranged in a straight line. Configure a wheel with multiple terminals physically arranged in a circle.

The touch position is calculated from the measured values of the TS in the configuration. The calculation method for sliders and wheels is fundamentally the same.

1. Detect the maximum value (TS_MAX) among the terminals in the configuration.
2. Calculate the difference (d1, d2) between TS_MAX and the terminals on either side. (If the TS_MAX terminal is at one end of the slider, use the values of the two terminals to the right or left, accordingly.)
3. If the total of d1 and d2 exceeds the threshold, position calculation is initiated. If the total amount does not exceed the threshold, the position calculation process is ended.
4. With TS_MAX as the middle position, the ratio of d1 to d2 is used to calculate the position. The slider has a range of 1 to 100, and the while has a range of 1 to 360.

1.1.5 Tuning the Touch Determination Threshold

When QE tuning, a measurement is performed with a finger touching the button and the tuned parameters are output in the configuration file. The setting value of the threshold is 60% of the touch sensitivity between touch and non-touch state, and the setting value of the hysteresis coefficient is 5% of the threshold.

This module provides the functions for dynamic adjusting of these threshold and hysteresis coefficient.

They are two functions as below.

1. Adjusting the threshold and hysteresis coefficient to an arbitrary ratio.

Use RM_TOUCH_ThresholdAdjust ()

[Example of use]

Wanting to change the threshold to 70% of the touch sensitivity against EMC noise.

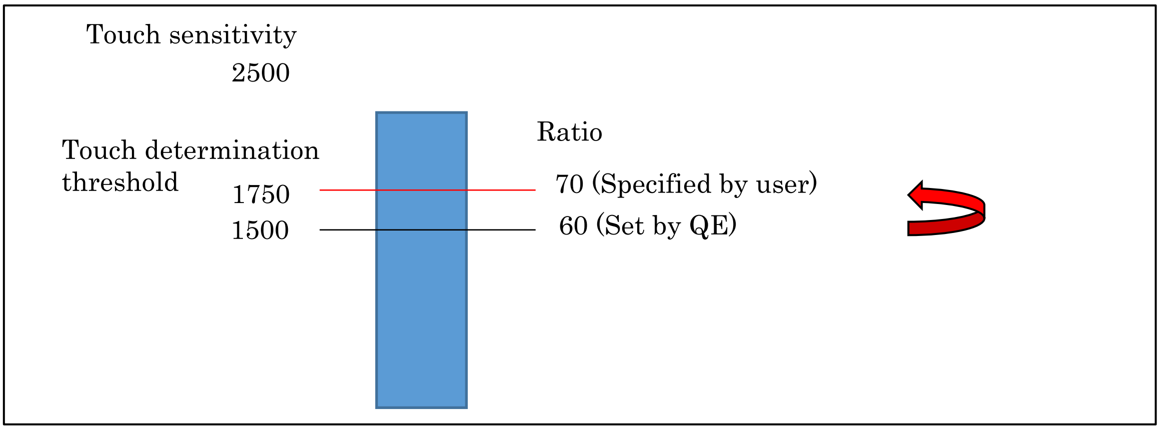


Figure 3 Example of changing the threshold ratio

2. Adjusting the threshold and hysteresis coefficient according to the current touch sensitivity

Use RM_TOUCH_SensitivityRatioGet (), RM_TOUCH_ThresholdAdjust (), and RM_TOUCH_DriftControl().

[Example of use]

When changing the kind of the overlay panel, the touch sensitivity differs from the one QE tuned. Wanting to use the software as it is without re-tuning. If you use a thicker overlay than that at QE tuning, the touch sensitivity decreases, and a touch may not be determined because of the same touch determination threshold. This function adjusts the touch determination threshold based on the ratio of the touch sensitivity after changing the overlay to the touch sensitivity at the QE tuning.

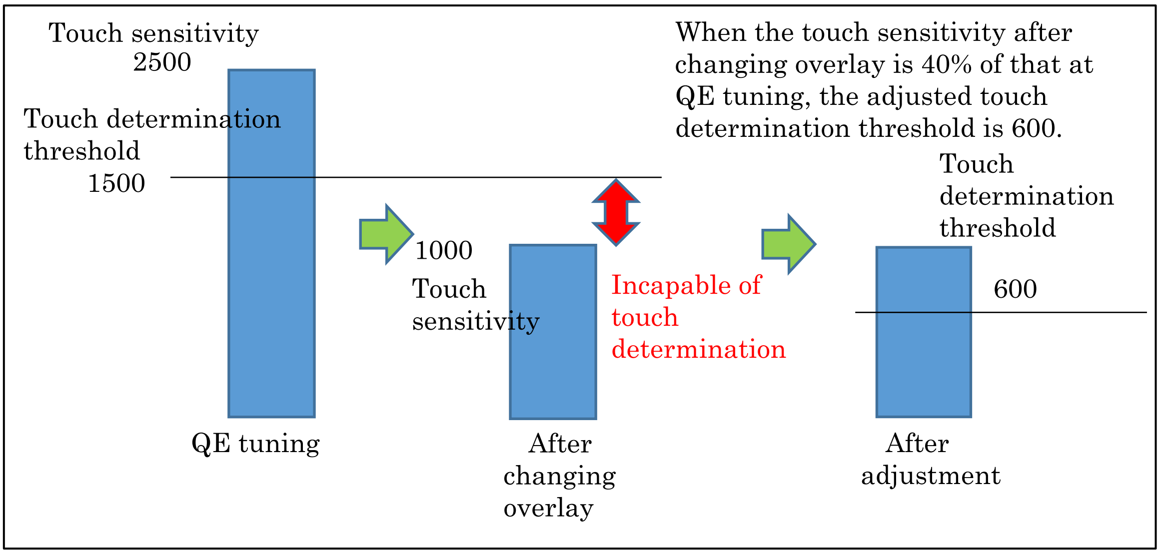


Figure 4 Example of threshold adjustment in the change of touch change amount

This is an example of the application for adjustment using data flash without re-tuning or software rewriting. Enable UART communication to PC and ‘tuning mode’. In tuning mode, the MCU transmits the ratio of the touch sensitivity in the touch state to the PC in real time. A user sends a command to decide the ratio while monitoring on the PC. The MCU stores the received ratio in the data flash. Make sure that the ratio stored in the data flash is read at the software activation, and the touch determination threshold is adjusted based on this stored value.

1.2 API Overview

The TOUCH module includes the following functions.

Function	Description
RM_TOUCH_Open()	Initializes the specified touch interface configuration.
RM_TOUCH_ScanStart()	Starts measurement of specified touch interface configuration.
RM_TOUCH_DataGet()	Gets measured values of specified touch interface configuration.
RM_TOUCH_CallbackSet()	Sets callback function of specified touch interface configuration.
RM_TOUCH_Close()	Closes specified touch interface configuration.
RM_TOUCH_ScanStop()	Stops measuring the specified touch interface configuration.
RM_TOUCH_GetSensitivityRatio()	Calculates the ratio of the current touch sensitivity to that at QE settings.
RM_TOUCH_AdjustThresholdRatio()	Changes the ratio of touch determination threshold and the hysteresis value to the touch sensitivity and adjusts the touch determination threshold and the hysteresis value based on the ratio of the current touch sensitivity.
RM_TOUCH_DriftControl()	Changes drift correction settings.
RM_TOUCH_MonitorAddressGet()	Gets the address of the variable used for the QE monitor.

2. API Information

Operations of this FIT module have been confirmed under the following conditions.

2.1 Hardware Requirements

The MCU used in the development must support one of the following functions:

- CTSU
- CTSU2L
- CTSU2SL

2.2 Software Requirements

This driver depends on the following FIT modules:

- Board support package (r_bsp) v6.10 or newer
- QE CTSU FIT Module (r_ctsu_qe) v2.20
- SCI module (r_sci_rx) v3.90 or newer

The driver also assumes the use of the following tool:

- QE for Capacitive Touch V3.1.0 (capacitive touch sensor development support tool) or newer

2.3 Supported Toolchains

FIT module operations have been confirmed on the following toolchains:

- Renesas CC-RX Toolchain v.3.04.00
- IAR RX Toolchain v4.20.3
- GCC RX Toolchain v8.3.0.202202

2.4 Restrictions

The module code is non-reentrant and protects simultaneous calls for multiple functions.

2.5 Header File

All interface definitions to be called and used in the API are defined in ”rm_touch_qe.h”.

Select “rm_touch_qe_config.h” as the configuration option in each build.

2.6 Integer Type

This driver uses ANSI C99. The types are defined in stdint.h.

2.7 Compilation Settings

The following table provides the names and setting values for the configuration option settings used the TOUCH module.

rm_touch_config.h Configuration Options	
TOUCH_CFG_PARAM_CHECKING_ENABLE *Default value: "BSP_CFG_PARAM_CHECKING_ENABLE"	Selects whether to include the parameter check process in the code. Selecting "0" allows the user to omit the parameter check process from the code to shorten the code size. "1": Omit parameter check process from code. "2": Include parameter check process in code. "BSP_CFG_PARAM_CHECKING_ENABLE": Selection depends on BSP setting.
TOUCH_CFG_MONITOR_ENABLE This option is not available for rm_touch_config.h. The option is defined in the qe_touch_define.h output by the QE; the default value is "1".	Select 1 to enable data generation for the QE monitor.
TOUCH_CFG_UART_MONITOR_SUPPORT *Default value: "0"	This option is used when TOUCH_CFG_MONITOR_ENABLE is enabled. Set to "1" to enable QE and serial communications. Note: When using the UART module, generate this option with the Smart Configurator.
TOUCH_CFG_UART_MONITOR_SUPPORT	Set the use of UART monitor. 0: Disable, 1: Enable
TOUCH_CFG_UART_TUNING_SUPPORT	Set the use of UART tuning. 0: Disable, 1: Enable
TOUCH_CFG_UART_NUMBER	Set the UART channel number.
TOUCH_CFG_UART_BAUDRATE	Set the UART Baudrate.
TOUCH_CFG_UART_PRIORITY	Set the UART interrupt priority.
TOUCH_CFG_DETECTION_OPTION *Default value: "0"	Set the chattering suppression type. Set "0", it is set to TypeA. The counter of the number of times the threshold is exceeded is held within the hysteresis range. Set "1", it is set to TypeB. Resets the counter of the number of times the threshold is exceeded within the hysteresis range.
The following configurations depend on the touch interface configuration and cannot be set using Smart Configurator. These configurations are set when using QE. In this case, QE_TOUCH_CONFIGURATION is defined in the project. Although rm_touch_config.h is invalid, qe_touch_define.h is defined instead.	
CTSU_CFG_NUM_BUTTONS	Sets the total number of buttons.
CTSU_CFG_NUM_SLIDERS	Sets the total number of slides.
CTSU_CFG_NUM_WHEELS	Sets the total number of wheels.
TOUCH_CFG_PAD_ENABLE	Select whether to use the TouchPad. 0: Disable, 1: Enable

2.8 Code Size

ROM (code and constants) and RAM (global data) size are determined according to the configuration options as described in "section 2.7 Compilation Setting" during a build. The values shown are reference values when the compile option is the default for C compiler listed in "section 2.3 Supported Toolchains". The default of compile options is as follows: the optimization level is 2, the optimization type is size priority, and the data-endian is a little endian. The code size varies according to the C compile version and the compile options.

ROM and RAM Usage the configuration options with Self-capacitance 1 button	
TOUCH_CFG_PARAM_CHECKING_ENABLE 0 TOUCH_CFG_MONITOR_ENABLE 0 TOUCH_CFG_UART_MONITOR_SUPPORT 0	ROM: 1637byte RAM: 103byte

ROM and RAM Usage Size of each mode, amount of increase by adding						
	Self-capacitance button 1	+Self-capacitance button	+Wheel	+Slider	Mutual-capacitance button 1	+Mutual-capacitance button
ROM	1637byte	+0 byte	+509 byte	+560 byte	1702 byte	+0 byte
RAM	103byte	+25 byte	+33 byte	+35 byte	105 byte	+27 byte

2.9 Arguments

The followings are the structures and enums used as arguments of the API functions. Many of the parameters used in the API functions are defined by the enums, which provides a way to check types and reduce errors.

These structures and enums are defined in rm_touch_qe.h, rm_touch_qe_api.h along with the prototype declaration.

The following is the control structure for the touch interface configuration. This does not need to be set in the application. Using QE allows the variables corresponding to the touch interface configuration to be output by qe_touch_config.c. Make sure to set qe_touch_config.c in the module's first API argument.

```
typedef struct st_touch_instance_ctrl
{
    uint32_t              open;              ///< Whether or not driver is open.
    touch_button_info_t   binfo;             ///< Information of button.
    touch_slider_info_t   sinfo;             ///< Information of slider.
    touch_wheel_info_t    winfo;             ///< Information of wheel.
    touch_cfg_t const   * p_touch_cfg;       ///< Pointer to initial configurations.
    ctsu_instance_t const * p_ctsu_instance; ///< Pointer to CTSU instance.
} touch_instance_ctrl_t;
```

The following is the configuration setting structure for the touch interface configuration.

Using QE for Capacitive Touch allows the variables and initialization values corresponding to the touch interface configuration to be output by qe_touch_config.c. Make sure to set qe_touch_config.c in the second argument of RM_TOUCH_Open().

```
typedef struct st_touch_cfg
{
    touch_button_cfg_t const * p_buttons;    ///< Pointer to array of button configuration.
    touch_slider_cfg_t const * p_sliders;    ///< Pointer to array of slider configuration.
```

```
    touch_wheel_cfg_t const  * p_wheels;     ///< Pointer to array of wheel configuration.
    uint8_t                  num_buttons;    ///< Number of buttons.
    uint8_t                  num_sliders;    ///< Number of sliders.
    uint8_t                  num_wheels;     ///< Number of wheels.
    uint8_t                  on_freq;        ///< The cumulative number of determinations of ON.
    uint8_t                  off_freq;       ///< The cumulative number of determinations of OFF.
    uint16_t                 drift_freq;     ///< Base value drift frequency. [0 : no use]
    uint16_t                 cancel_freq;    ///< Maximum continuous ON. [0 : no use]
    uint8_t                  number;         ///< Configuration number for QE monitor.
    ctsu_instance_t const * p_ctsu_instance; ///< Pointer to CTSU instance.
    void const             * p_context;      ///< User defined context passed into callback function.
    void const             * p_extend;       ///< Pointer to extended configuration by instance of interface.
} touch_cfg_t;
```

The followings are the enums used for the above listed structures.

```
/** Configuration of each button */
typedef struct st_touch_button_cfg
{
    uint8_t  elem_index;            ///< Element number used by this button.
    uint16_t threshold;             ///< Touch/non-touch judgment threshold
    uint16_t hysteresis;            ///< Threshold hysteresis for chattering prevention.
} touch_button_cfg_t;

/** Configuration of each slider */
typedef struct st_touch_slider_cfg
{
    uint8_t const * p_elem_index;   ///< Element number array used by this slider.
    uint8_t         num_elements;   ///< Number of elements used by this slider.
    uint16_t        threshold;      ///< Position calculation start threshold value.
} touch_slider_cfg_t;

/** Configuration of each wheel */
typedef struct st_touch_wheel_cfg_t
{
    uint8_t const * p_elem_index;   ///< Element number array used by this wheel.
    uint8_t         num_elements;   ///< Number of elements used by this wheel.
    uint16_t        threshold;      ///< Position calculation start threshold value.
} touch_wheel_cfg_t;

/** Callback function parameter data */
typedef struct st_ctsu_callback_args touch_callback_args_t;/** CTSU Events for callback function */
```

2.10 Return Values

The following provides return values for the API functions. The enum is defined in fsp_common_api.h.

```
  /** Common error codes */
  typedef enum e_fsp_err
  {
      FSP_SUCCESS = 0,

      FSP_ERR_ASSERTION             = 1,                      ///< A critical assertion has failed
      FSP_ERR_INVALID_POINTER       = 2,                      ///< Pointer points to invalid memory location
      FSP_ERR_INVALID_ARGUMENT      = 3,                      ///< Invalid input parameter
      FSP_ERR_NOT_OPEN              = 7,                      ///< Requested channel is not configured or API not open
      FSP_ERR_ALREADY_OPEN          = 14,                     ///< Requested channel is already open in a different
configuration
      FSP_ERR_INVALID_HW_CONDITION  = 27,                     ///< Detected hardware is in invalid condition


      /* Start of CTSU Driver specific */
      FSP_ERR_CTSU_SCANNING              = 6000,       ///< Scanning.
      FSP_ERR_CTSU_NOT_GET_DATA          = 6001,       ///< Not processed previous scan data.
      FSP_ERR_CTSU_INCOMPLETE_TUNING     = 6002,       ///< Incomplete initial offset tuning.
      FSP_ERR_CTSU_DIAG_NOT_YET          = 6003,       ///< Diagnosis of data collected no yet.
      FSP_ERR_CTSU_DIAG_LDO_OVER_VOLTAGE = 6004,       ///< Diagnosis of LDO over voltage failed.
      FSP_ERR_CTSU_DIAG_CCO_HIGH         = 6005,       ///< Diagnosis of CCO into 19.2uA failed.
      FSP_ERR_CTSU_DIAG_CCO_LOW          = 6006,       ///< Diagnosis of CCO into 2.4uA failed.
      FSP_ERR_CTSU_DIAG_SSCG             = 6007,       ///< Diagnosis of SSCG frequency failed.
      FSP_ERR_CTSU_DIAG_DAC              = 6008,       ///< Diagnosis of non-touch count value failed.
      FSP_ERR_CTSU_DIAG_OUTPUT_VOLTAGE   = 6009,       ///< Diagnosis of LDO output voltage failed.
      FSP_ERR_CTSU_DIAG_OVER_VOLTAGE     = 6010,       ///< Diagnosis of over voltage detection circuit failed.
      FSP_ERR_CTSU_DIAG_OVER_CURRENT     = 6011,       ///< Diagnosis of over current detection circuit failed.
      FSP_ERR_CTSU_DIAG_LOAD_RESISTANCE  = 6012,       ///< Diagnosis of LDO internal resistance value failed.
      FSP_ERR_CTSU_DIAG_CURRENT_SOURCE   = 6013,       ///< Diagnosis of Current source value failed.
      FSP_ERR_CTSU_DIAG_SENSCLK_GAIN     = 6014,       ///< Diagnosis of SENSCLK frequency gain failed.
      FSP_ERR_CTSU_DIAG_SUCLK_GAIN       = 6015,       ///< Diagnosis of SUCLK frequency gain failed.
      FSP_ERR_CTSU_DIAG_CLOCK_RECOVERY   = 6016,       ///< Diagnosis of SUCLK clock recovery function failed.
      FSP_ERR_CTSU_DIAG_CFC_GAIN         = 6017,       ///< Diagnosis of CFC oscillator gain failed.


} fsp_err_t;
```

2.11 Adding the FIT Module to Your Project

2.11.1 Adding source tree and project include paths

This module must be added to each project in which it is used. Renesas recommends using “Smart Configurator” described in (1) or (3). However, “Smart Configurator” only supports some RX devices. Please use the methods of (2) or (4) for unsupported RX devices.

(1) Adding the FIT module to your project using “Smart Configurator” in e2 studio
By using the “Smart Configurator” in e2 studio, the FIT module is automatically added to your project. Refer to “Renesas e2 studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using “FIT Configurator” in e2 studio
By using the “FIT Configurator” in e2 studio, the FIT module is automatically added to your project. Refer to “Adding Firmware Integration Technology Modules to Projects (R01AN1723)” for details.

(3) Adding the FIT module to your project using “Smart Configurator” on CS+
By using the “Smart Configurator Standalone version” in CS+, the FIT module is automatically added to your project. Refer to “Renesas e2 studio Smart Configurator User Guide (R20AN0451)” for details.

(4) Adding the FIT module to your project in CS+
In CS+, please manually add the FIT module to your project. Refer to “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

2.11.2 Setting driver options when not using Smart Configurator

The Touch-specific options are found and edited in r_config¥r_touch_qe_config.h.

2.12 API compatibility mode

The API function has been completely revised in Rev.2.00.
An API compatibility mode is provided to use the Rev.1.11 API.
Please output the code of API compatibility mode when outputting the code in QE. Check the code output option "Use API compatibility mode" in QE and click the "File output" button to output the code for API compatibility mode.
Please refer to QE Touch Module Firmware Integration Technology Rev.1.11 (R01AN4470JU0111) for the API of Rev.v1.11.
API compatibility mode does not support the error detection of QE_ERR_OT_WINDOW_SIZE that was detected by R_TOUCH_Open ().

3. API Functions

3.1 RM_TOUCH_Open

This function initializes the module and must be executed before using any of the other API functions. Please execute this function for each touch interface.

Format

```
fsp_err_t RM_TOUCH_Open (touch_ctrl_t * const p_ctrl,
                         touch_cfg_t const * const p_cfg)
```

Parameters

p_ctrl Pointer to the control structure (normally generated by QE)

p_cfg Pointer to the config structure (normally generated by QE)

Return Values

FSP_SUCCESS	*/* Successfully completed */*
FSP_ERR_ASSERTION	*/* Argument pointer not specified */*
FSP_ERR_ALREADY_OPEN	*/* Open() is called without calling Close() */*
FSP_ERR_INVALID_ARGUMENT	*/* Configuration parameters are invalid */*

Properties

Prototype is declared in r_touch_qei.h.

Description

This function enables control structure initialization, calls R_CTSU_Open(), and initializes the CTSU2L module according to the argument p_cfg.

By setting TOUCH_CFG_MONITOR_ENABLE, the monitor buffer is initialized. By setting TOUCH_CFG_UART_MONITOR_SUPPORT, the UART monitor and UART module are initialized.

Example

```
fsp_err_t err;

/* Initialize pins (function created by Smart Configurator) */
R_CTSU_PinSetInit();

/* Initialize the API. */
err = RM_TOUCH_Open(&g_touch_ctrl, &g_touch_cfg);

/* Check for errors. */
if (err != FSP_SUCCESS)
{
    . . .
}
```

Special Notes:

The port must be initialized before calling this function. We recommend using the R_CTSU_PinSetInit()

function generated by SmartConfigurator as the port initialization function.

This function calls the CTSU module's R_CTSU_Open().

3.2 RM_TOUCH_ScanStart

This function starts measurement of the specified touch interface configuration.

Format

```
fsp_err_t RM_TOUCH_ScanStart (touch_ctrl_t * const p_ctrl)
```

Parameters

p_ctrl Pointer to the control structure (normally generated by QE)

Return Values

FSP_SUCCESS	*/* Successfully completed */*
FSP_ERR_ASSERTION	*/* Argument pointer not specified */*
FSP_ERR_NOT_OPEN	*/* Called without calling Open() */*
FSP_ERR_CTSU_ERR_SCANNING	*/* Now scanning */*
FSP_ERR_CTSU_NOT_GET_DATA	*/* Did not obtain previous results */*

Properties

Prototype is declared in r_touch_api.h.

Description

This function calls R_CTSU_ScanStart() and starts the measurement.

Example

```
fsp_err_t err;

/* Initiate a sensor scan by software trigger */
err = RM_TOUCH_ScanStart(&g_touch_ctrl);

/* Check for errors. */
if (err != FSP_SUCCESS)
{
    . . .
}
```

Special Notes:

This function calls the CTSU module's R_CTSU_ScanStart(). Reference the R_CTSU_ScanStart() document for more details.

3.3 RM_TOUCH_DataGet

This function reads the specified touch interface configuration.

Format

```
fsp_err_t RM_TOUCH_DataGet (touch_ctrl_t * const p_ctrl,
                            uint64_t         * p_button_status,
                            uint16_t         * p_slider_position,
                            uint16_t         * p_wheel_position)
```

Parameters

p_ctrl	Pointer to the control structure (normally generated by QE)
p_button_status	Pointer to the buffer that stores button state.
p_slider_position	Pointer to the buffer that stores slider position.
p_wheel_position	Pointer to the buffer that stores wheel position.

Return Values

FSP_SUCCESS	*/* Successfully completed */*
FSP_ERR_ASSERTION	*/* Argument pointer not specified */*
FSP_ERR_NOT_OPEN	*/* Called without calling Open() */*
FSP_ERR_CTSU_SCANNING	*/* Now scanning */*
FSP_ERR_CTSU_NOT_GET_DATA	*/* Did not obtain previous results */*
FSP_ERR_CTSU_INCOMPLETE_TUNING	*/* Tuning initial offset */*
FSP_ERR_INVALID_HW_CONDITION	*/ * Values scanned by CTSU show abnormal values */*

Properties

Prototype is declared in r_touch_qe.h.

Description

This function calls R_CTSU_DataGet() and reads all measured values from the previous measurement to determine the touch/non-touch state or position. By setting TOUCH_CFG_MONITOR_ENABLE, data is stored in the monitor buffer. By setting TOUCH_CFG_UART_MONITOR_SUPPORT, the data in the monitor buffer is sent to the UART module.

Example:

```
  fsp_err_t err;
  uint64_t button_status;
  uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
  uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];

  /* Get all sensor values */
  err = RM_TOUCH_DataGet(&g_touch_ctrl, &button_status, slider_position,
wheel_position);
```

Special Notes:

This function calls the CTSU module's R_CTSU_DataGet(). Reference the R_CTSU_DataGet() document for more details.

3.4 RM_TOUCH_CallbackSet

This function sets the function specified for the measurement completion callback function.

Format

```
fsp_err_t RM_TOUCH_CallbackSet (touch_ctrl_t * const p_api_ctrl,
                                void (* p_callback)(touch_callback_args_t *),
                                void const * const p_context,
                                touch_callback_args_t * const p_callback_memory)
```

Parameters

p_api_ctrl Pointer to the control structure (normally generated by QE for Capacitive Touch)

p_callback Pointer to callback function

p_context Pointer to send to callback function

p_callback_memory Set to NULL

Return Values

FSP_SUCCESS	*/* Successfully completed */*
FSP_ERR_ASSERTION	*/* Argument pointer not specified */*
FSP_ERR_NOT_OPEN	*/* Called without calling Open() */*

Properties

Prototype is declared in rm_touch_qe.h.

Description

This function calls R_CTSU_CallbackSet() and sets the callback function.

Example:

```
fsp_err_t err;

/* Set callback function */
err = RM_TOUCH_CallbackSet(&g_ctsu_ctrl, ctsu_callback, NULL, NULL);
```

Special Notes:

This function calls the CTSU module's R_CTSU_CallbackSet(). Reference the R_CTSU_CallbackSet() document for more details.

3.5 RM_TOUCH_Close

This function closes the specified touch interface configuration.

Format

```
fsp_err_t RM_TOUCH_Close (touch_ctrl_t * const p_ctrl)
```

Parameters

p_ctrl Pointer to the control structure (normally generated by QE)

Return Values

FSP_SUCCESS	*/* Successfully completed */*
FSP_ERR_ASSERTION	*/* Argument pointer not specified */*
FSP_ERR_NOT_OPEN	*/* Called without calling Open() */*

Properties

Prototype is declared in r_touch_qe.h.

Description

This function closes the specified touch interface configuration.

Example:

```
fsp_err_t err;

/* Shut down peripheral and close driver */
err = RM_TOUCH_Close(&g_touch_ctrl);
```

Special Notes:

This function calls the CTSU module's R_CTSU_Close(). Reference the R_CTSU_Close() document for more details

3.6 RM_TOUCH_ScanStop

This function stops measuring the specified touch interface configuration.

Format

```
fsp_err_t RM_TOUCH_ScanStop (touch_ctrl_t * const p_ctrl)
```

Parameters

p_ctrl Pointer to the control structure (normally, generated by QE for Capacitive Touch)

Return Values

FSP_SUCCESS */* Successfully completed */*

FSP_ERR_ASSERTION */* Argument pointer not specified */*

FSP_ERR_NOT_OPEN */* Called without calling Open() */*

Properties

Prototype is declared in rm_touch_qe.h.

Description

This function stops measuring the specified touch interface configuration.

Example:

```
fsp_err_t err;

/* Stop CTSU module */
err = RM_TOUCH_ScanStop(&g_touch_ctrl);
```

Special Notes:

None

3.7 RM_TOUCH_SensitivityRatioGet

This function returns the ratio of the current touch sensitivity to that at the QE tuning.

Format

```
fsp_err_t RM_TOUCH_SensitivityRatioGet (touch_ctrl_t * const p_ctrl,
                     touch_sensitivity_info_t * p_touch_sensitivity_info);
```

Parameters

p_ctrl

Pointer to the control structure (normally, generated by QE for Capacitive Touch)

p_modifier

Pointer to the variable storing table information of touch sensitivity ratio calculation

Return Values

FSP_SUCCESS	*/* Successfully got the ratio of touch sensitivity */*
FSP_ERR_INVALID_POINTER	*/* Pointing to the invalid memory location */*
FSP_ERR_CTSU_NOT_GET_DATA	*/* Did not obtain previous results */*
FSP_ERR_CTSU_INCOMPLETE_TUNING	*/* Tuning initial offset */*

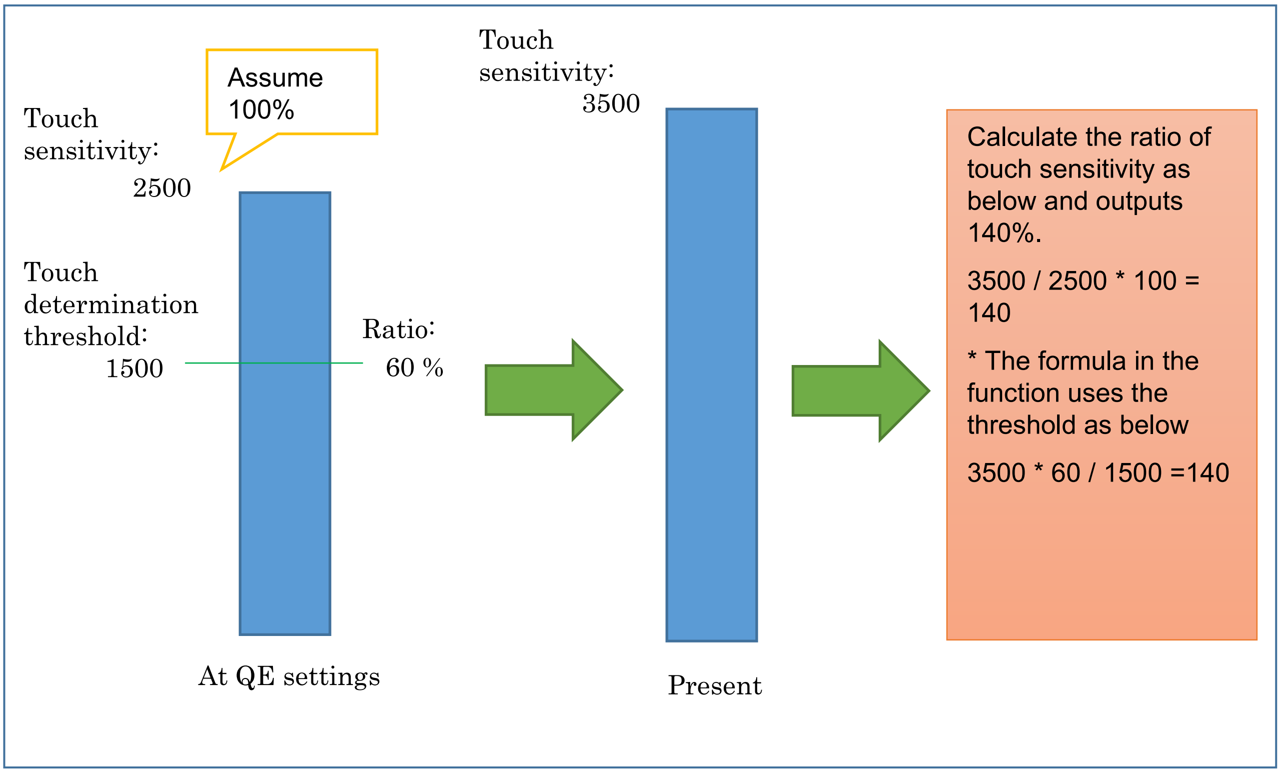
Properties

Prototyped in file “rm_touch_qe.h”

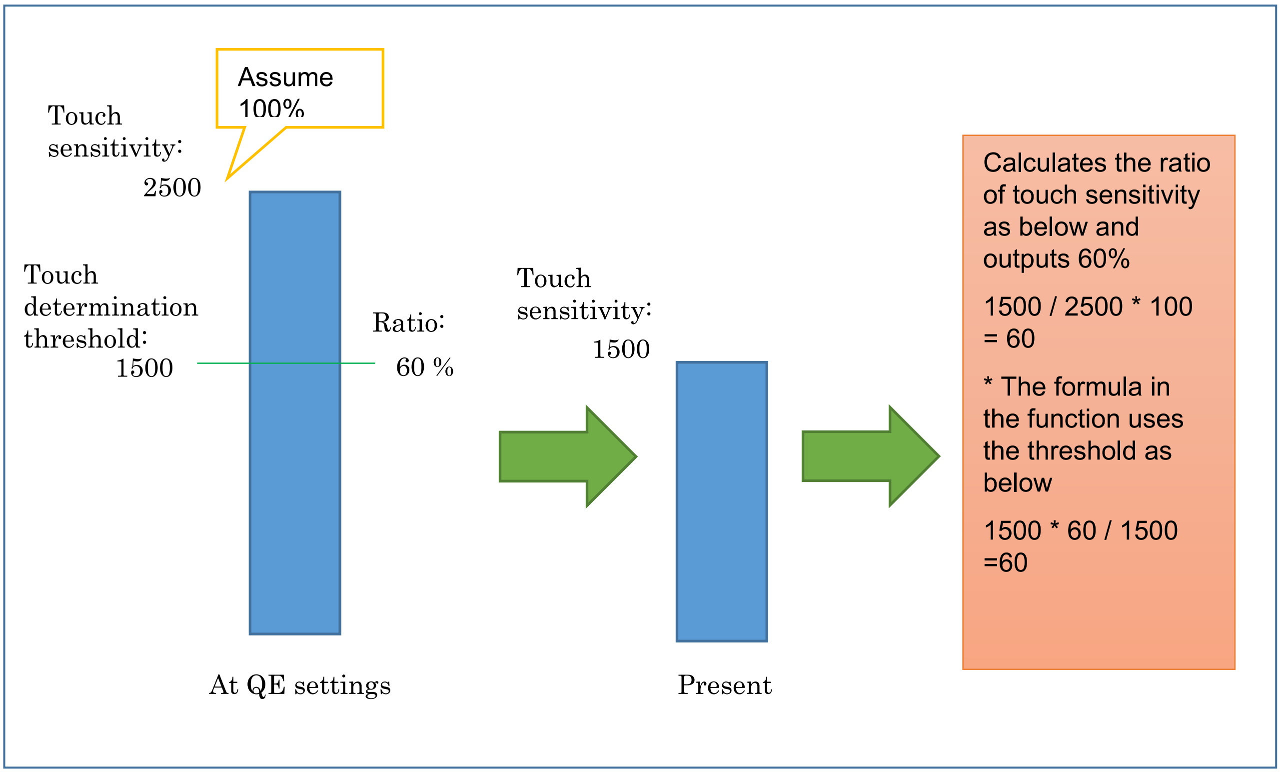
Description

This function outputs the ratio of the current touch sensitivity assuming that the touch sensitivity at the QE setting is 100%.

The following figure shows the case where an overlay panel is thinner and the touch sensitivity increases.



Following figure shows the case where an overlay panel is thicker and the touch sensitivity decreases.



Example:

```
  qe_err_t err;
  touch_sensitivity_info_t touch_sensitivity_table[QE_NUM_METHODS];
  uint16_t touch_sensitivity_first[CONFIG01_NUM_BUTTONS ] = { 100 };

  touch_sensitivity_table[QE_METHOD_CONFIG01].p_touch_sensitivity_ratio =
touch_sensitivity_first;
  touch_sensitivity_table[QE_METHOD_CONFIG01].old_threshold_ratio = 60;
  touch_sensitivity_table[QE_METHOD_CONFIG01].new_threshold_ratio = 70;
  touch_sensitivity_table[QE_METHOD_CONFIG01].new_hysteresis_ratio = 5;

  err = RM_TOUCH_SensitivityRatioGet(g_qe_touch_instance_config01.p_ctrl,
&touch_sensitivity_table[QE_METHOD_CONFIG01]);
```

3.8 RM_TOUCH_ThresholdAdjust

This function changes the ratio of touch determination threshold and hysteresis value to the touch sensitivity and changes the touch determination threshold corresponding to the current touch sensitivity.

Format

```
fsp_err_t RM_TOUCH_ThresholdAdjust  (touch_ctrl_t * const p_ctrl,
                                    touch_sensitivity_info_t * p_touch_sensitivity_info);
```

Parameters

p_ctrl

Pointer to the control structure (normally, generated by QE for Capacitive Touch)

p_modifier

Pointer to the variable storing table information of touch sensitivity ratio calculation

Return Values

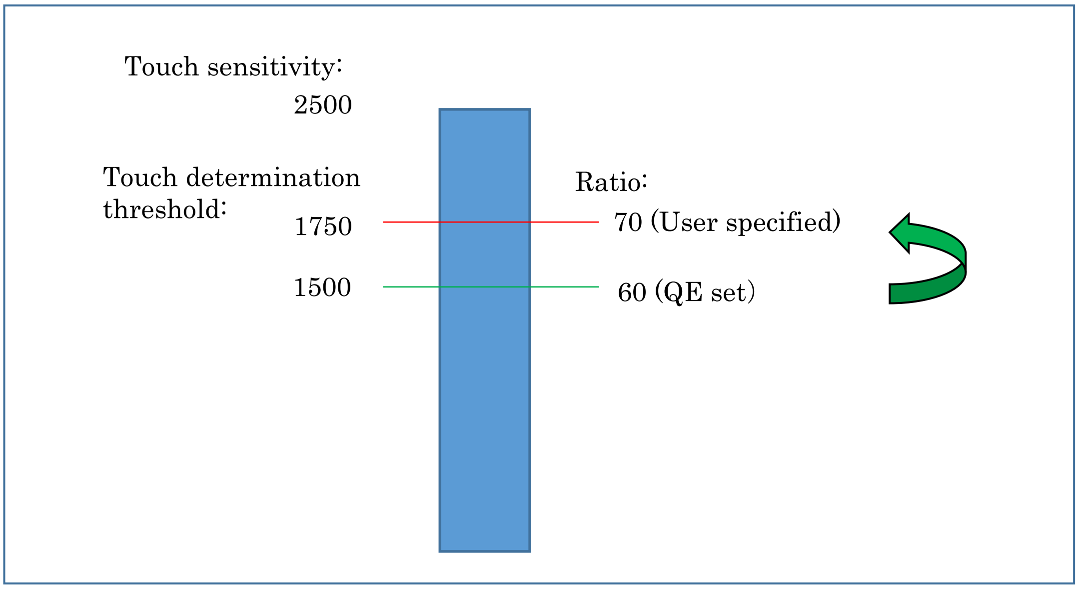
FSP_SUCCESS	*/* Successfully changed touch determination threshold. */*
FSP_ERR_INVALID_POINTER	*/* Pointing to the invalid memory location */*
FSP_ERR_INVALID_ARGUMENT	*/* Configuration parameters are invalid */*

Properties

Prototyped in file “rm_touch_qe.h”

Description

When changing the touch determination threshold ratio from 60% QE set to 70% user specified, the touch determination thresholds are as below.



If you want to make this setting, set the member of the second argument as follows. It is also necessary to set the ratio of the amount of touch change and the hysteresis value.

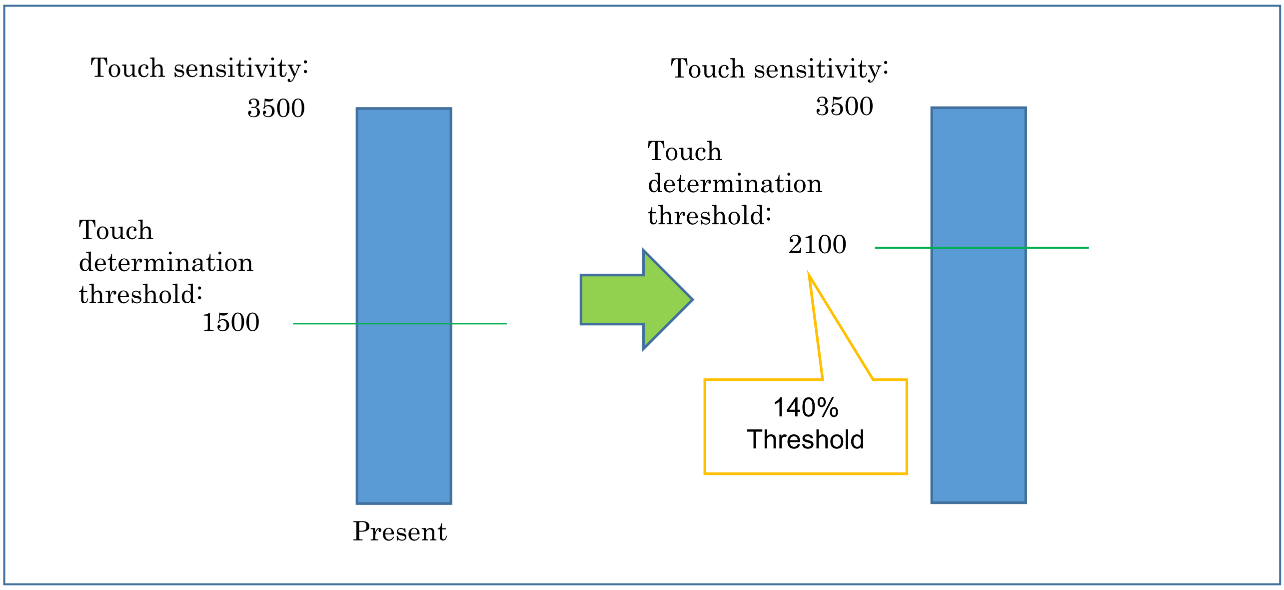
* p_touch_sensitivity_ratio = 100

old_threshold_ratio = 60

new_threshold_ratio = 70
new_hysteresis_ratio = 5

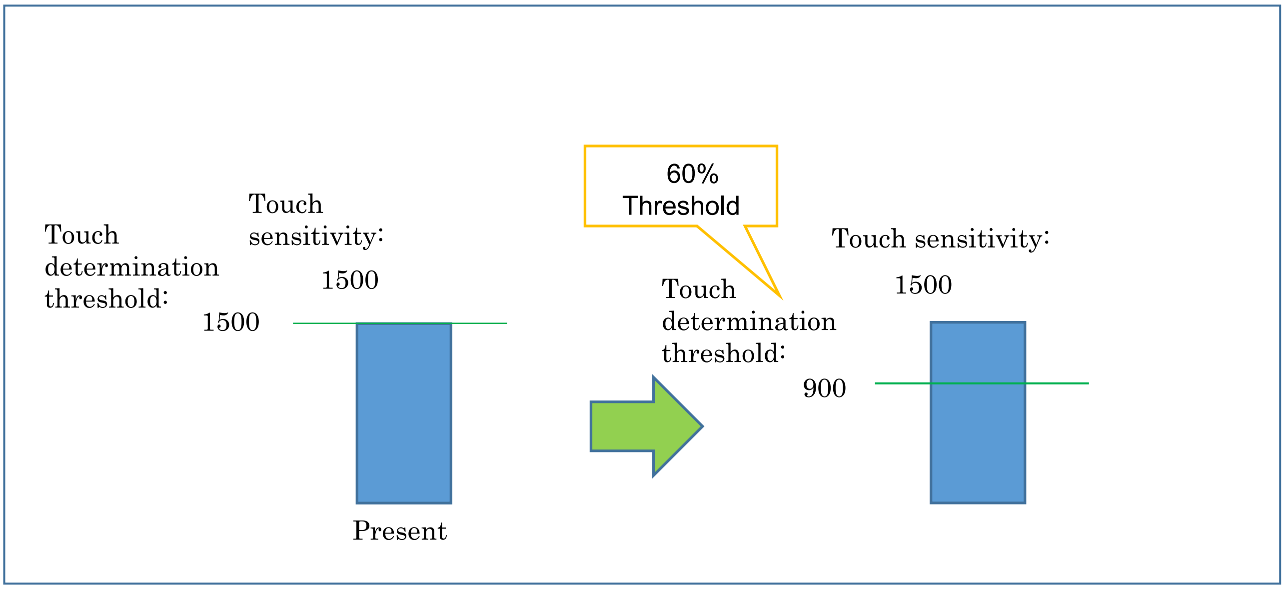
Sets the new touch determination threshold and the hysteresis value by using the touch sensitivity ratio obtained with RM_TOUCH_SensitivityRatioGet () as arguments.

Example of calculation 1: The touch sensitivity ratio is 140%, and the threshold set by QE is 1500.
140 * 1500 / 100 = 2100



*p_touch_sensitivity_ratio = 140
old_threshold_ratio = 60
new_threshold_ratio = 60
new_hysteresis_ratio = 5

Example of calculation 2: The touch sensitivity ratio is 60%, and the threshold set by QE is 1500.
60 * 1500 / 100 = 900



*p_touch_sensitivity_ratio = 60
old_threshold_ratio = 60
new_threshold_ratio = 60
new_hysteresis_ratio = 5

Example:

```
  qe_err_t err;
  touch_sensitivity_info_t touch_sensitivity_table[QE_NUM_METHODS];
  uint16_t touch_sensitivity_first[CONFIG01_NUM_BUTTONS ] = { 100 };

  touch_sensitivity_table[QE_METHOD_CONFIG01].p_touch_sensitivity_ratio =
touch_sensitivity_first;
  touch_sensitivity_table[QE_METHOD_CONFIG01].old_threshold_ratio = 60;
  touch_sensitivity_table[QE_METHOD_CONFIG01].new_threshold_ratio = 70;
  touch_sensitivity_table[QE_METHOD_CONFIG01].new_hysteresis_ratio = 5;

  err = RM_TOUCH_SensitivityRatioGet(g_qe_touch_instance_config01.p_ctrl,
&touch_sensitivity_table[QE_METHOD_CONFIG01]);

  err = RM_TOUCH_ThresholdAdjust(g_qe_touch_instance_config01.p_ctrl,
&touch_sensitivity_table[QE_METHOD_CONFIG01]);
```

Special Notes:

If you want to change the touch change amount without changing the ratio of the touch change amount and the threshold value during QE tuning, set the element of the second argument of RM_TOUCH_ThresholdAdjust () as follows.

old_threshold_ratio = 60

new_threshold_ratio = 60

new_hysteresis_ratio = 5

3.9 RM_TOUCH_DriftControl

This function changes the settings of drift correction.

Format

```
fsp_err_t RM_TOUCH_DriftControl(touch_ctrl_t * const p_ctrl,
                                uint16_t input_drift_freq);
```

Parameters

p_ctrl

Pointer to the control structure (normally, generated by QE for Capacitive Touch

input_drift_freq

Enables / disables interval of drift correction

Return Values

FSP_SUCCESS	*/* Successfully changed drift correction */*
FSP_ERR_ASSERTION	*/* Missing required argument pointer */*

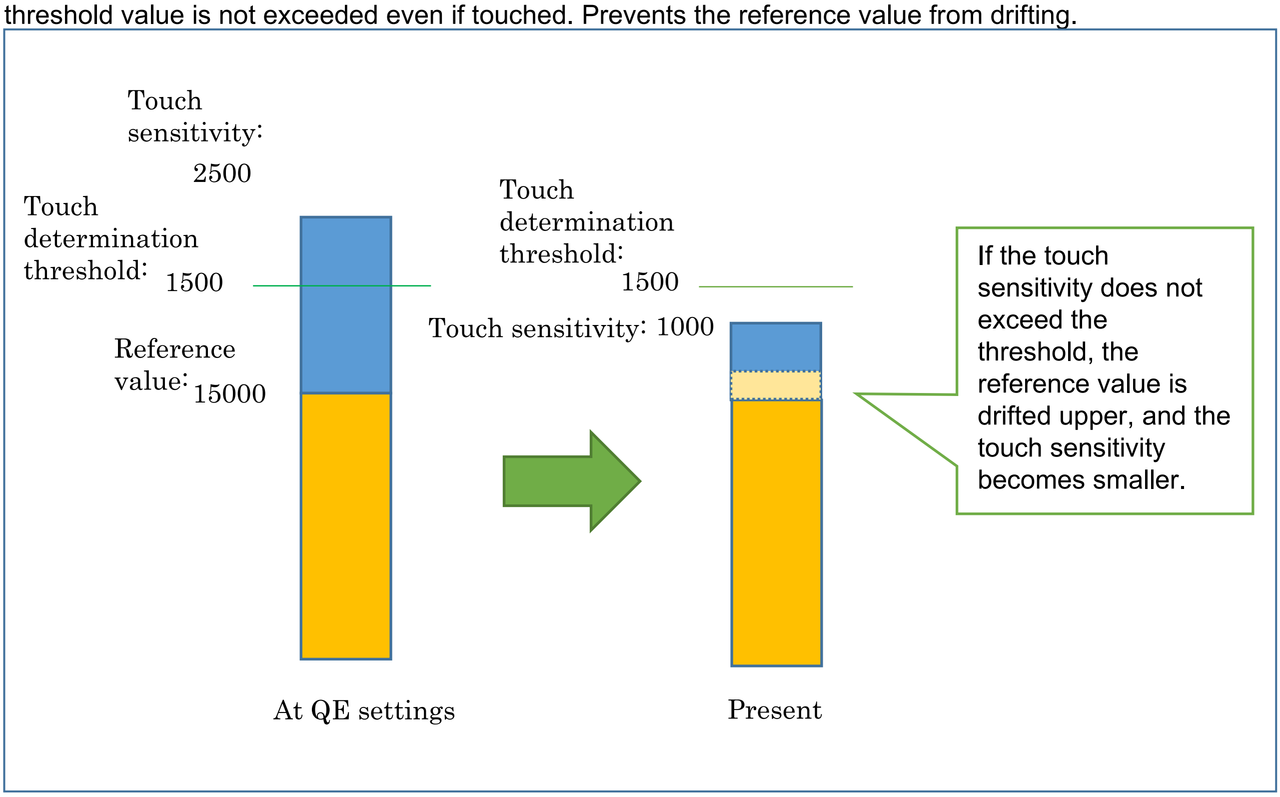
Properties

Prototyped in file rm_touch_qe.h.

Description

Set the drift correction to the number of times set in input_drift_freq. Set to 0 to stop the drift correction function.

As an example of using this API, when calculating the ratio of the touch change amount using RM_TOUCH_SensitivityRatioGet (), the touch change amount decreases due to the thick overlay, and the threshold value is not exceeded even if touched. Prevents the reference value from drifting.



Example:

```
qe_err_t  err;

err = RM_TOUCH_DriftControl(g_qe_touch_instance_config01.p_ctrl, 0);
```

3.10 RM_TOUCH_MonitorAddressGet

This function gets the address of the variable used for the QE monitor.

Format

```
fsp_err_t RM_TOUCH_MonitorAddressGet (touch_ctrl_t * const p_ctrl,
                                      uint32_t * p_monitor_buf,
                                      uint32_t * p_monitor_id,
                                      uint32_t * p_monitor_size)
```

Parameters

p_ctrl

Pointer to the control structure (normally, generated by QE for Capacitive Touch

p_monitor_buf

Pointer to a variable that stores the start address of the monitor

p_monitor_id

Pointer to a variable that stores the address of the monitor ID variable

p_monitor_size

Pointer to a variable that stores the start address of the monitor size

Return Values

FSP_SUCCESS	*/* Successfully QE monitor variable address was got */*
FSP_ERR_ASSERTION	*/* Null pointer passed as a parameter */*
FSP_ERR_NOT_OPEN	*/* Module is not open */*
FSP_ERR_NOT_ENABLED	*/* Requested operation is not enabled */*

Properties

Prototyped in file rm_touch_qe.h.

Description

Use this feature for the QE monitor when you have both automatic judgement and software judgement touch interface configurations. Get the start address of the monitor buffer with the second argument, the address of the monitor ID variable with the third argument, and the start address of the monitor size with the fourth argument.

Example:

```
qe_err_t  err;
uint32_t monitor_buf_address;
uint32_t monitor_id_address;
uint32_t monitor_size_address;

err =  RM_TOUCH_MonitorAddressGet(g_qe_touch_instance_config01.p_ctrl,
                                  &monitor_buf_address,
                                  &monitor_id_address,
                                  &monitor_size_address);
```

Special Notes:

Normally, QE is not used except for the sample application output due to the QE monitor function.

Revision History

<table>
<tr><th rowspan="2">Rev.</th><th rowspan="2">Date</th><th colspan="2">Description</th></tr>
<tr><th>Page</th><th>Summary</th></tr>
<tr><td>1.00</td><td>Oct.04.18</td><td>—</td><td>First edition issued</td></tr>
<tr><td>1.10</td><td>Jul.09.19</td><td>1,39
4-6
15, 18, 22, 30, 31
24, 26

*
14, 16-20

1,20
15,22</td><td>Added RX23W support
Added definitions for “correction” and “offset tuning”.
Updated API return values
Added TOUCH_CMD_GET_FAILED_SENSOR and TOUCH_CMD_GET_LAST_SCAN_METHOD Control() commands
Moved offset tuning processing into R_TOUCH_Open().
Added #pragma section macros and configuration option to driver for Safety Module support (includes GCC/IAR support).
Added IEC 6730 Compliance section.
Added error code QE_ERR_UNSUPPORTED_CLK_CFG.</td></tr>
<tr><td>1.11</td><td>Jan.09.20</td><td>30-31
29,32

5,6,22,35
—

—</td><td>Updated example code.
Added TOUCH_CMD_CLEAR_TOUCH_STATES for low power applications.
Added API function R_TOUCH_GetBtnBaselines().
Fixed bug (CTSU) where a custom callback function was called twice after a scan completes.
Fixed compile error (CTSU) for RX231 when PLL had multiplier of 13.5.</td></tr>
<tr><td>2.00</td><td>Jul.30.21</td><td>-</td><td>Full-fledged revision</td></tr>
<tr><td rowspan="2">2.01</td><td rowspan="2">Dec.17.21</td><td>7</td><td>Fixed 1.2 API Overviews</td></tr>
<tr><td>10</td><td>Fixed 2.8 Code Size</td></tr>
<tr><td rowspan="2">2.10</td><td rowspan="2">Apr.20.22</td><td>7</td><td>Fixed 1.2 API Overviews</td></tr>
<tr><td>29,30</td><td>Added 3.10 RM_TOUCH_MonitorAddressGet</td></tr>
<tr><td>2.20</td><td>Dec.28.22</td><td>4</td><td>Added to 1.13 Button touch detection</td></tr>
<tr><td></td><td></td><td>9</td><td>Updated 2.2 Software requirements</td></tr>
<tr><td></td><td></td><td>9</td><td>Updated 2.3 Supported toolchains</td></tr>
<tr><td></td><td></td><td>11</td><td>Fixed 2.8 Code size</td></tr>
<tr><td></td><td></td><td>13</td><td>Updated 2.10 return value</td></tr>
<tr><td></td><td></td><td>18</td><td>Updated 3.3 R_TOUCH_DataGet</td></tr>
</table>

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.