RX Family

QE Touch Module Using Firmware Integration Technology

R01AN4470EU0111
Rev. 1.11
Jan 9, 2020

Introduction

The QE Touch Module Using Firmware Integration Technology (FIT) is part of the QE for Capacitive Touch tool suite. It provides the capacitive touch API to be used by the main application.

Target Device

The following is a list of devices that are currently supported by this API:

- RX113 Groups
- RX130 Group
- RX23W, RX230, RX231 Groups

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

Related Documents

- Using QE and FIT to Develop Capacitive Touch Applications (R01AN4516EU)
- QE Touch Diagnostic API Users' Guide (R01AN4785EU)
- Firmware Integration Technology User's Manual (R01AN1833EU)
- Board Support Package Firmware Integration Technology Module (R01AN1685EU)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723EU)
- 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 QE Touch FIT module is part of the QE for Capacitive Touch tool suite. Within e^2 studio, the QE for Capacitive Touch Tool is used to define scan configurations and generate input data and constants used by this module. This module provides the API for the user's application. Transparent to the user, this module relies upon the QE CTSU FIT module for the peripheral driver.

1.1 Features

Below is a list of the features supported by the QE Touch FIT module.

- All arguments for Open() are generated by the QE for Capacitive Touch tool
- Sensors can be configured for Self or Mutual mode operation
- Buttons, sliders, and wheels are supported
- Scans may begin by a software trigger or an external trigger
- Up to eight scan configurations can be supported in a single application.

1.2 Self Mode

The driver makes use of the following terminology and definitions when in self-capacitance mode:

Self Mode
In self-capacitance mode, only one CTSU TS sensor is necessary to functionally operate a touch button/key. A series of these sensors physically aligned can be used to create a **slider**. If the series of sensors are aligned such that they create a circular pattern, this is referred to as a **wheel**.

Scan Order
In Self mode, the hardware scans the specified number of sensors in ascending order. For example, if sensors 5, 8, 2, 3, and 6 are specified in your application, the hardware will scan them in the order: 2, 3, 5, 6 and 8.

Element
An element refers to the index of a sensor within the scan order. Using the previous example, sensor number 5 is element 2.

Scan Buffer Contents
At the lowest level, both the CTSUSC sensor count and CTSURC reference count registers are loaded into the driver buffer for each sensor in the scan configuration. Though the RC is not used, both registers are placed into the buffer for two reasons. 1) Both registers must have their contents read for proper scan operation. 2) This allows for identical processing for both interrupt and DTC operation at a later point. Note, however, that API calls such as R_TOUCH_GetData() which access this buffer will load only the sensor values.

Scan Time
The scanning of sensors occurs in the background by the CTSU peripheral and does not utilize any main processor time. It takes approximately 500us to scan a single sensor. If DTC is not used, an additional 2.2us overhead (system clock 54MHz) is added for the main processor to transfer data to/from registers when each sensor is scanned.

Methods/Scan Configurations
These terms are used interchangeably. A single method (scan configuration) refers to the set of sensors to be scanned along with what mode they are to be scanned in (Self or Mutual). One to eight methods per application can be defined within the QE for Capacitive Touch tool suite. Typically there is only one method used per application. However, more advanced applications may require different scan configurations to be enabled as different features are enabled within the product, or when a combination of Self and Mutual sensor configurations are present.

Correction
When the CTSU peripheral is first initialized, it temporarily goes into correction mode where it simulates various touch input values and compares against ideal sensor counts. These counts should fall along a linear line, but realistically will need slight adjustment to do so. The correction process alters internal CTSU register values and generates correction factors for the Touch layer to ensure the most accurate sensor readings possible. This addresses potential subtle differences in the MCU manufacturing process.

Offset Tuning
Offset tuning is performed by the Touch middleware layer during its Open() process. This tuning makes slight adjustments to the QE Tool generated CTSU register values in order to address subtle capacitance differences due to the surrounding environment, such as variations in temperature, humidity, and proximity to other electronic components or devices. Once this process is complete, the system is considered tuned and ready for operation.

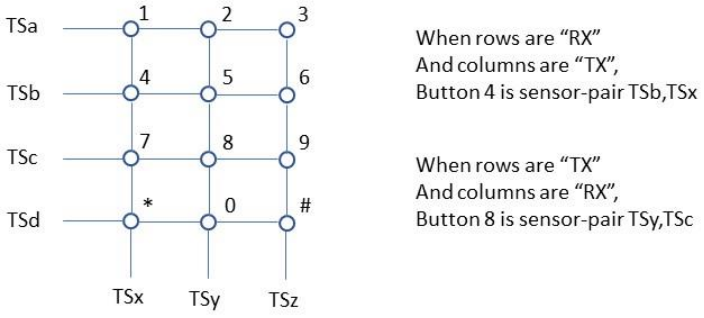
Baseline
After Open() completes, a long term moving average is updated for each button after every scan by the Touch middleware layer. Every xxx_DRIFT_FREQ number of scans (see "qe_<method.h>.h" file), this moving average is saved as the button's baseline value. The baseline value is the sensor count which a threshold offset is added to for determining if a button is touched or not. Its initial value is set to the button's moving average calculated during the offset tuning process.

1.3 Mutual Mode

The driver makes use of the following terminology and definitions when in mutual-capacitance mode:

Mutual Mode
In mutual-capacitance mode, two CTSU TS sensors are necessary to functionally operate a single touch button/key. This mode is typically used when a matrix/keypad with more than four keys is present. This is because Mutual mode requires fewer sensors to operate than the equivalent in Self mode does.

Consider a standard phone keypad. You can think of it as a matrix of four rows and three columns. One TS sensor is shared for each row, and another sensor is shared for each column. Each button/key is identified by a **sensor-pair**. The RX sensor is always listed first in the pair. (All row or all column sensors are designated as RX, and the others are designated TX. The peripheral scans both the RX and TX sensors in a sensor-pair, subtracts the difference, and the result is used to determine whether a button/key is touched or not.)



As you can see, only seven sensors are necessary to scan 12 buttons. In self mode, 12 sensors are required.

At this time, sliders and wheels are not supported in Mutual mode.

Scan Order
In Mutual mode, the hardware scans the sensor-pairs in ascending order, with sensors configured as RX being the **primary**, and the sensors configured as TX the **secondary**. For example, if sensors 10, 11, and 3 are specified as RX sensors, and sensors 2, 7, and 4 are specified as TX sensors, the hardware will scan them in the following sensor-pair order:

3,2 - 3,4 - 3,7
10,2 - 10,4 - 10,7
11,2 - 11,4 - 11,7

Element
In mutual-capacitance mode, an element refers to the index of a sensor-pair within the scan order. Using the previous example, sensor-pair 10,7 is element 5.

Scan Buffer Contents
At the lowest level, both the CTSUSC sensor count and CTSURC reference count registers are loaded into the driver buffer for each sensor in a sensor-pair in the scan configuration. Though the RC is not used, both registers are placed

into the buffer for two reasons. 1) Both registers must have their contents read for proper scan operation. 2) This allows for identical processing for both interrupt and DTC operation at a later point. Note, however, that API calls such as R_TOUCH_GetData() which access this buffer will load only the sensor values.

Scan Time
The scanning of sensors occurs in the background by the CTSU peripheral and does not utilize any main processor time. It takes approximately 1000us (1ms) to scan a single sensor-pair. If DTC is not used, an additional 4.4us overhead (system clock 54MHz) is added for the main processor to transfer data to/from registers when each sensor-pair is scanned.

Methods/Configurations
These terms are used interchangeably. A single method (scan configuration) refers to the set of sensors to be scanned along with what mode they are to be scanned in (self or mutual). One to eight methods per application can be defined within the QE for Capacitive Touch Tool. Typically there is only one method used per application. However, more advanced applications may require different scan configurations to be enabled as different features are enabled within the product, or when a combination of Self and Mutual sensor configurations are present.

Correction
When the CTSU peripheral is first initialized, it temporarily goes into correction mode where it simulates various touch input values and compares against ideal sensor counts. These counts should fall along a linear line, but realistically will need slight adjustment to do so. The correction process alters internal CTSU register values and generates correction factors for the Touch layer to ensure the most accurate sensor readings possible. This addresses potential subtle differences in the MCU manufacturing process.

Offset Tuning
Offset tuning is performed by the Touch middleware layer during its Open() process. This tuning makes slight adjustments to the QE Tool generated CTSU register values in order to address subtle capacitance differences due to the surrounding environment, such as variations in temperature, humidity, and proximity to other electronic components or devices. Once this process is complete, the system is considered tuned and ready for operation.

Baseline
After Open() completes, a long term moving average is updated for each button after every scan by the Touch middleware layer. Every xxx_DRIFT_FREQ number of scans (see “qe_<method.h>.h” file), this moving average is saved as the button’s baseline value. The baseline value is the sensor count which a threshold offset is added to for determining if a button is touched or not. Its initial value is set to the button’s moving average calculated during the offset tuning process.

1.4 Trigger Sources

Scanning of sensors may begin by either a software trigger or an external event initiated by the Event Link Controller (ELC). Typically, a software trigger is used. Common usage is to have a periodic timer initiate scans. The subtle differences in timing between periodic software and external triggers is illustrated in the following sections.

For software triggers, a periodic timer such as the CMT is configured whose interval is large enough to allow for all sensors to be scanned and data to be updated (see *Scan Time* in sections 1.2 and 1.3). When the timer expires, the following sequence of events occurs:

- A flag is set in the timer interrupt routine
- A slight delay occurs until the main application detects that the flag has been set (red bar in Figure 1).
- Using a Touch Middleware driver, the main application makes an API call to load the scanned data, update internal values (such as moving averages), and issue a software trigger to begin another scan. Using the FIT QE Touch driver, this is done using the API function R_TOUCH_UpdateDataAndStartScan().

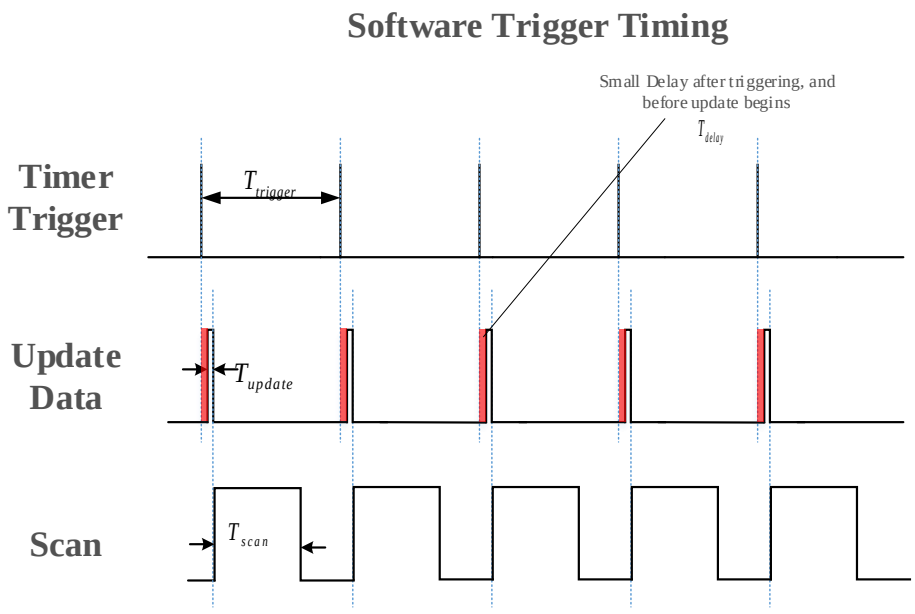


Figure 1: Periodic Software Trigger Timing Diagram (timing not to scale)

Using an external trigger is processed almost identically to using software triggers. Typically, a periodic timer such as the MTU is configured whose interval is large enough to allow for all sensors to be scanned and data to be updated (see *Scan Time* in sections 1.2 and 1.3). This timer is then linked to the ELC which in turn is linked to the CTSU. When the timer expires, the following sequence of events occurs:

- The ELC is triggered which triggers the CTSU to start a scan.
- When a scan completes, a flag is set in the CTSU scan-complete interrupt routine
- A slight delay occurs until the main application detects that the flag has been set (red bar in Figure 2.).
- Using a Touch Middleware driver, the main application makes an API call to load the scanned data and update internal values (such as moving averages). Using the FIT QE Touch driver, this is done using the API function R_TOUCH_UpdateData().

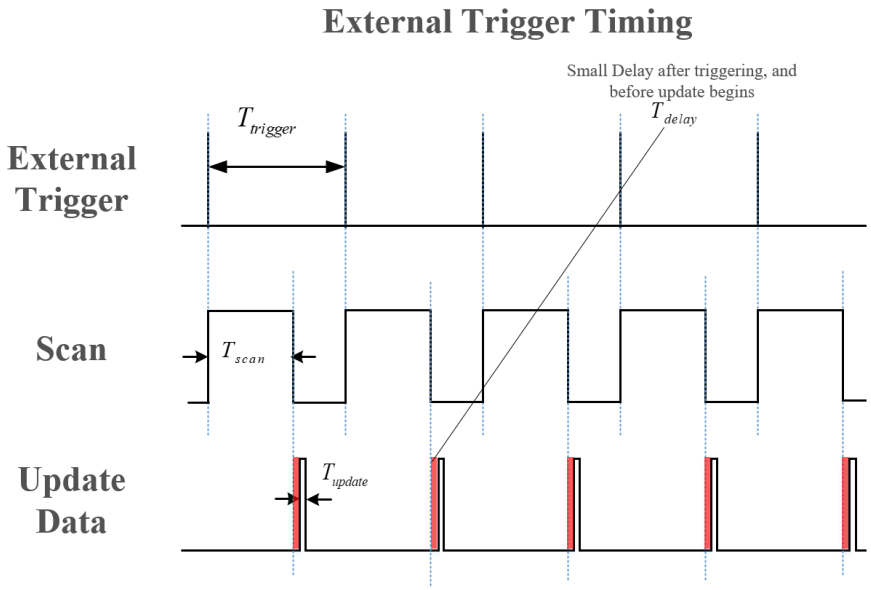


Figure 2: Periodic External Trigger Timing Diagram (timing not to scale)

With either trigger mechanism, the time it takes for the main application to detect a flag set in an interrupt routine can vary by a several microseconds with each scan depending upon the user's polling algorithm. Also, the time it takes to update the scanned data can vary by several microseconds depending upon such things as whether a sensor is touched or not (different paths through an else-if statement). In general, these subtle differences are minimal with each scan, and are extremely small compared to overall time spent performing a scan. For these reasons, the majority of users prefer the software triggering mechanism because of its simpler setup. But for those rare cases where a slight variation in scan interval is not acceptable, the external trigger mechanism can be used. **NOTE: DTC cannot be used when external triggers are used.**

1.5 Scanning Approaches

Scans can be performed on a periodic or non-periodic basis using software or external triggers. The following sections provide high level processing diagrams for some different approaches.

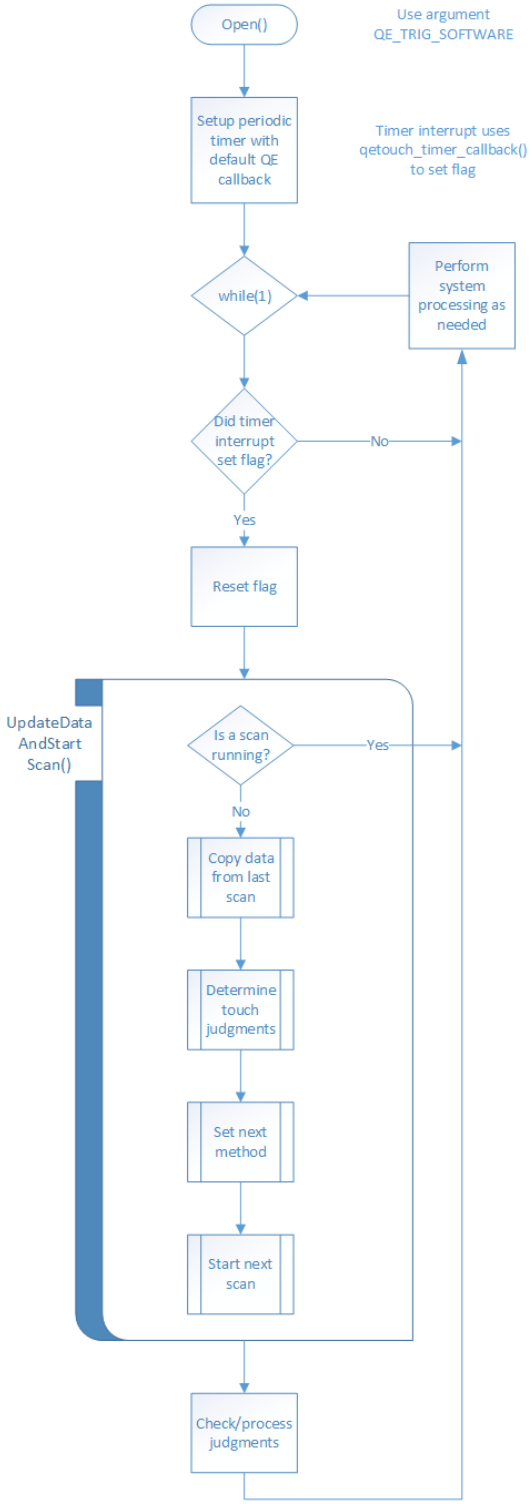
1.5.1 Periodic scanning with software triggers

Advantages:

- Simple to implement.

Disadvantages:

- If scan timer too short, some scans will not be started (UpdateDataAndStartScan() returns QE_ERR_BUSY).
- Scan interval varies due to UpdateDataAndStartScan() processing time (usually ok for HMI).
- Scan starts may be delayed significantly if user processing takes much longer than idle time between scans.



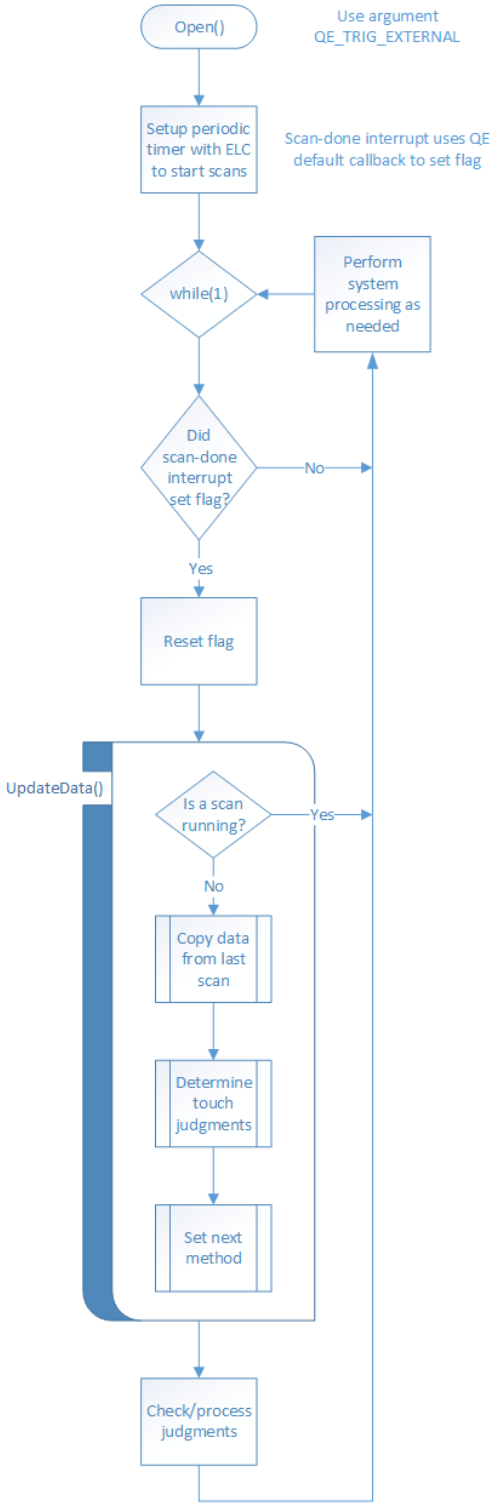
1.5.2 Periodic scanning with external triggers

Advantages:

- Scan interval very precise.

Disadvantages:

- Slightly more complex with setting up ELC.
- If scan timer too short, some scans will not be started (CTSU ignores trigger if already scanning)
- Scan data may be missed if user processing takes longer than idle time between scans.



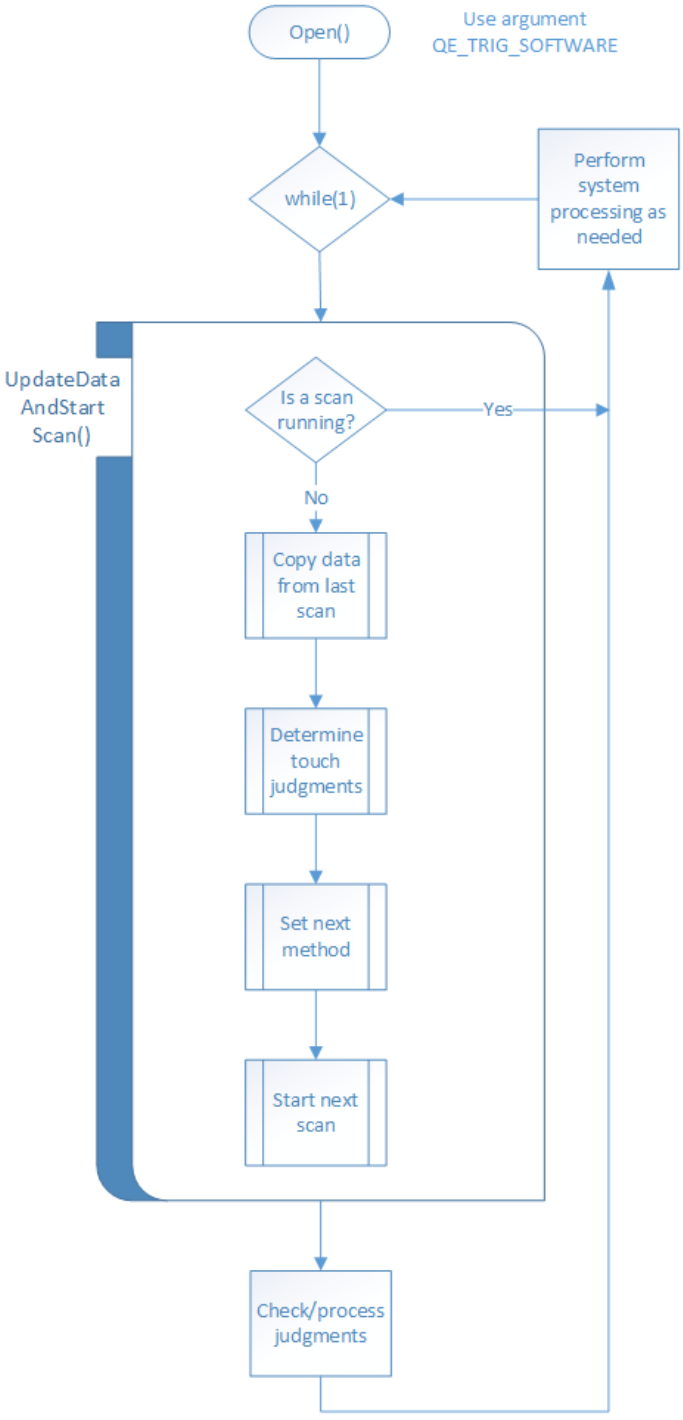
1.5.3 Continuous scanning with software triggers

Advantages:

- Can use when no timer resources available.

Disadvantages:

- Scan intervals vary based upon UpdateDataAndStartScan() processing and user processing time.
- If app is not dedicated to capacitive touch processing, and significant system processing occurs, drift compensation may not be as accurate due to unknown scan interval.



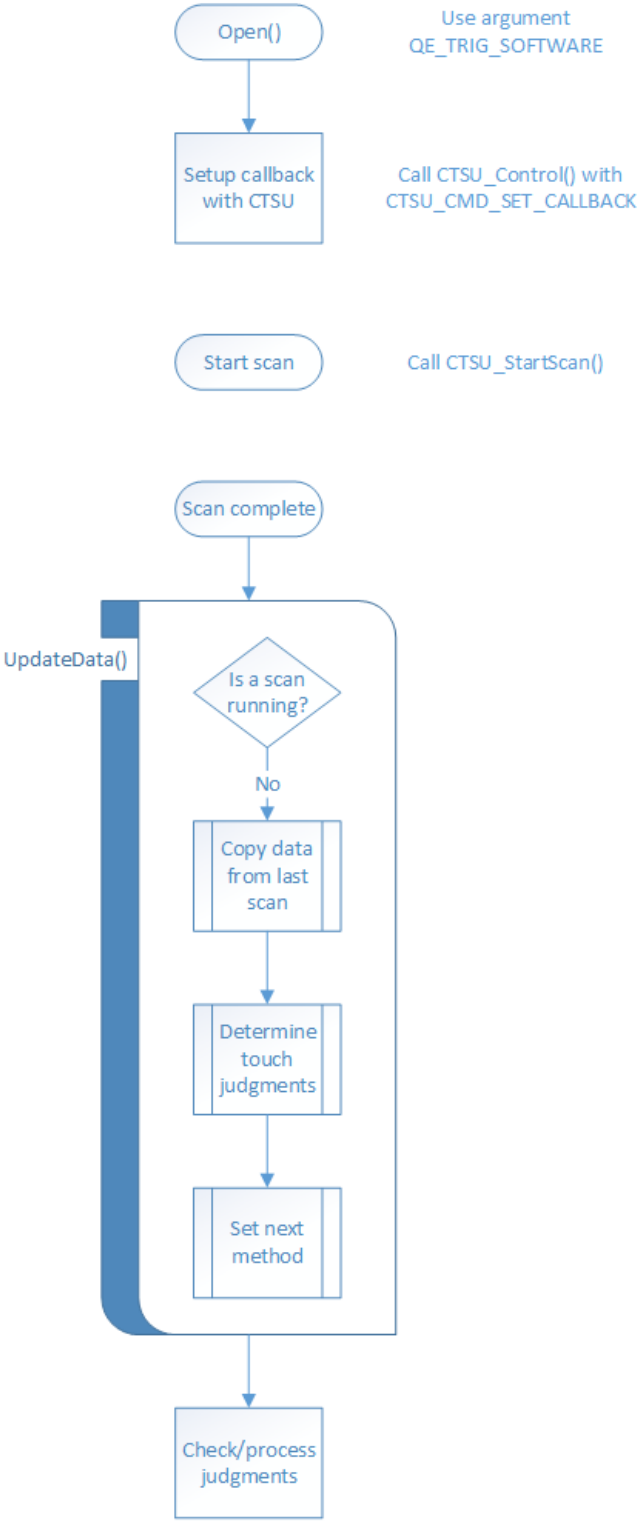
1.5.4 Non-Periodic scanning with software triggers

Advantages:

- Ultimate flexibility.

Disadvantages:

- Moving averages and drift compensation may not be accurate due to unknown scan intervals.



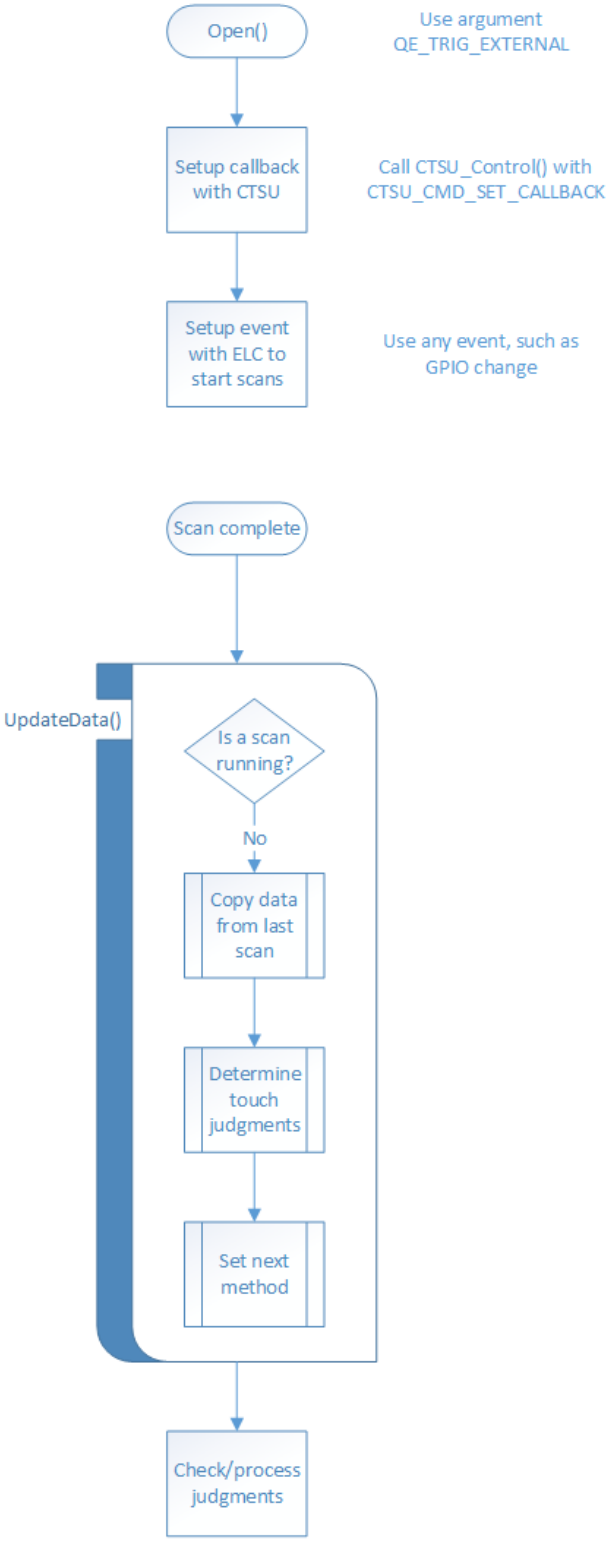
1.5.5 Non-Periodic scanning with external triggers

Advantages:

- Ultimate flexibility.

Disadvantages:

- Moving averages and drift compensation may not be accurate due to unknown scan intervals.



-

2. API Information

This Driver API follows the Renesas API naming standards.

2.1 Hardware Requirements

This driver requires that your MCU supports the following peripheral(s):

- CTSU

2.2 Software Requirements

This driver is dependent upon the following FIT packages:

- Renesas Board Support Package (r_bsp) v4.24 or later (v5.20 for GCC/IAR support).
- QE CTSU FIT module (r_ctsu_qe) v1.10.
- Renesas QE for Capacitive Touch e^2 studio plugin v1.10.

2.3 Limitations

- This code is not re-entrant and protects against multiple concurrent function calls.

2.4 Supported Toolchains

This driver is tested and working with the following toolchains:

- Renesas CC-RX Toolchain v3.01.00
- IAR RX Toolchain v4.11.1
- GCC RX Toolchain v4.8.4.201801

2.5 Header Files

All API calls and their supporting interface definitions are located in “r_touch_qe_if.h”. This file should be included by the user’s application.

Build-time configuration options are selected or defined in the file "r_touch_qe_config.h”.

2.6 Integer Types

This project uses ANSI C99 “Exact width integer types” in order to make the code clearer and more portable. These types are defined in stdint.h.

2.7 Configuration Overview

Configuring this module is done through the supplied r_touch_qe_config.h header file. Each configuration item is represented by a macro definition in this file. Each configurable item is detailed in the table below.

Configuration options in r_touch_qe_config.h		
Equate	**Default Value**	**Description**
TOUCH_CFG_PARAM_CHECKING_ENABLE	1	Setting to 0 omits parameter checking. Setting to 1 includes parameter checking.
TOUCH_CFG_UPDATE_MONITOR	1	Setting to 1 provides data for the monitoring tool to display data. Setting to 0 excludes the code needed for the monitoring tool to display data.
TOUCH_CFG_SAFETY_LINKAGE_ENABLE	0	Setting to 1 provides section information used by safety module. Setting to 0 is used for standard operation (safety module not used).

Table 1: QE Touch general configuration settings

2.8 Code Size

The code size is based on optimization level 2 and optimization type for size for the CC-RX toolchain in Section 2.4. The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options set in the module configuration header file. Note: The "+" in the table denotes QE generated file usage in addition to driver. RAM usage which varies with number of sensors and how they are configured.

ROM and RAM usage example: 2 button sensors and 4 slider sensors	
ROM usage: **TOUCH_PARAM_CHECKING_ENABLE 1 > TOUCH_PARAM_CHECKING_ENABLE 0** **TOUCH_CFG_UPDATE_MONITOR 1 > TOUCH_CFG_UPDATE_MONITOR 0**	
Minimum Size	ROM: 6485 bytes
	RAM: 568 bytes
Maximum Size	ROM: 7434 bytes
	RAM: 685 bytes

2.9 API Data Types

The API data structures are located in the file "r_typedefs_qe.h" and discussed in Section 3.

2.10 Return Values

This shows the different values API functions can return. This return type is defined in "r_typedefs_qe.h".

```
/* Return error codes */
typedef enum e_qe_err
{
    QE_SUCCESS = 0,
    QE_ERR_NULL_PTR,            // missing argument
    QE_ERR_INVALID_ARG,
    QE_ERR_BUSY,
    QE_ERR_ALREADY_OPEN,
    QE_ERR_CHAN_NOT_FOUND,
    QE_ERR_UNSUPPORTED_CLK_CFG, // unsupported clock configuration
    QE_ERR_SENSOR_SATURATION,   // sensor value detected beyond linear portion
                                //  of correction curve
    QE_ERR_TUNING_IN_PROGRESS,  // offset tuning for method not complete
    QE_ERR_ABNORMAL_TSCAP,      // abnormal TSCAP detected during scan
    QE_ERR_SENSOR_OVERFLOW,     // sensor overflow detected during scan
    QE_ERR_OT_MAX_OFFSET,       // CTSU SO0 offset reached max value and
                                //   sensor offset tuning incomplete
    QE_ERR_OT_MIN_OFFSET,       // CTSU SO0 offset reached min value and
                                //   sensor offset tuning incomplete
    QE_ERR_OT_WINDOW_SIZE,      // offset tuning window too small for sensor
                                //   to establish a reference count
    QE_ERR_OT_MAX_ATTEMPTS,     // 1+ sensors still not tuned for method
    QE_ERR_OT_INCOMPLETE,       // 1+ sensors still not tuned for method
    QE_ERR_TRIGGER_TYPE,        // function not available for trigger type
} qe_err_t;
```

2.11 Adding the QE Touch FIT Module to Your Project

For detailed explanation of how to add a FIT Module to your project, see document R01AN1723EU “Adding FIT Modules to Projects”.

2.11.1 Adding source tree and project include paths

In general, a FIT Module may be added in 3 ways:

1. Using an e^2 studio FIT tool, such as File>New>Renesas FIT Module (prior to v5.3.0), Renesas Views->e2 solutions toolkit->FIT Configurator (v5.3.0 or later), or projects created using the Smart Configurator (v5.3.0 or later). This adds the module and project include paths.
2. Using e^2 studio File>Import>General>Archive File from the project context menu.
3. Unzipping the .zip file into the project directory directly from Windows.

When using methods 2 or 3, the include paths must be manually added to the project. This is done in e^2 studio from the project context menu by selecting Properties>C/C++ Build>Settings and selecting Compiler>Source in the ToolSettings tab. The green “+” sign in the box to the right is used to pop a dialog box to add the include paths. In that box, click on the Workspace button and select the directories needed from the project tree structure displayed. The directories needed for this module are:

- ${workspace_loc:/${ProjName}/r_touch_qe
- ${workspace_loc:/${ProjName}/r_touch_qe/src
- ${workspace_loc:/${ProjName}/r_config

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.

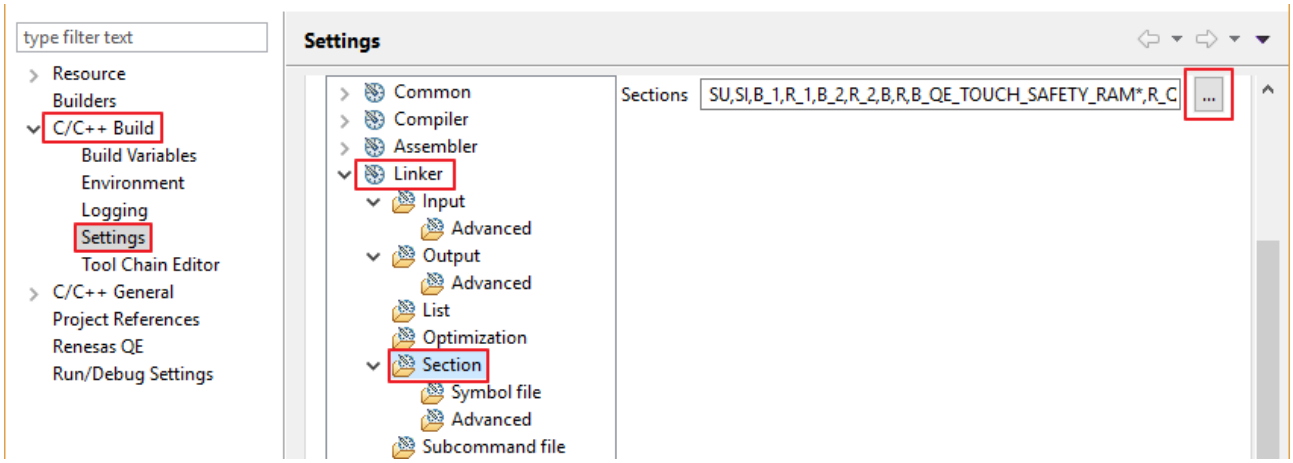
A reference copy (not for editing) containing the default values for this file is stored in \r_touch_qe\ref\r_touch_qe_config_reference.h.

2.12 Settings for Safety Module Usage

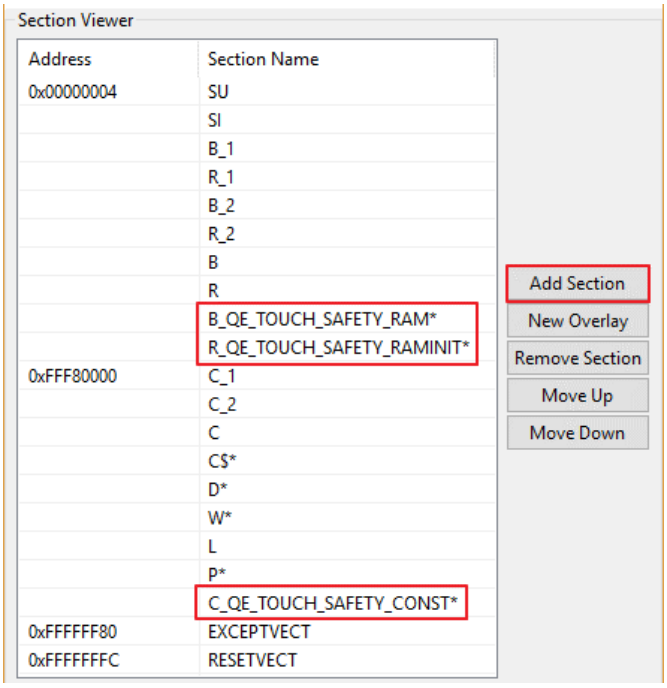
When building with the Safety Module for Class B operation (see application note R01AN4785EU), special sections must be added to the project’s linker script and their corresponding #pragmas generated via a setting in the driver configuration files.

2.12.1 Renesas Toolchain

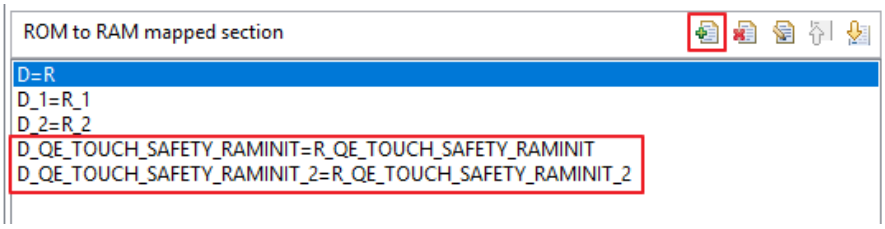
1) In e^2 studio, go to project Properties->C/C++ Build->Settings->Linker->Section and click the “…” button to the right of the Sections text box to open the Section Viewer.



2) Within the Section Viewer, add sections “B_QE_TOUCH_SAFETY_RAM*”, “R_QE_TOUCH_SAFETY_RAMINIT*”, and “C_QE_TOUCH_SAFETY_CONST*”.

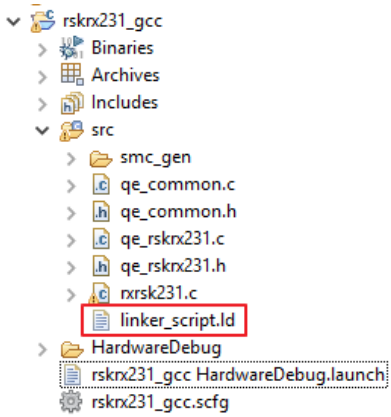


3) Go to project Properties->C/C++ Build->Settings->Linker->Section->Symbol file and add the following assignments:



2.12.2 GCC Toolchain

1) In e^2 studio, open the project linker file “linker_script.ld” in the text editor. This is located in the project “src” directory.



2) Add section “P_QE_TOUCH_DRIVER” to the “.text” ROM area:

```
.text 0xFFF80000: AT(0xFFF80000)
{
    *(.text)
    *(.text.*)
    *(P)

    /* Adding P sections for QE safety code */
    *(P_QE_TOUCH_DRIVER)

    etext = .;
} > ROM
```

3) Add sections “C_QE_TOUCH_SAFETY_CONSTANT”, “C_QE_TOUCH_SAFETY_CONSTANT_1”, and “C_QE_TOUCH_SAFETY_CONSTANT_2” to the ROM “.rodata” area:

```
.rodata :
{
    *(.rodata)
    *(.rodata.*)
    *(C_1)
    *(C_2)
    *(C)

    /* Adding C sections for QE safety code */
    *(C_QE_TOUCH_SAFETY_CONSTANT)
    *(C_QE_TOUCH_SAFETY_CONSTANT_1)
    *(C_QE_TOUCH_SAFETY_CONSTANT_2)

    _erodata = .;
} > ROM
```

4) Add sections “D_QE_TOUCH_SAFETY_RAMINIT” and “D_QE_TOUCH_SAFETY_RAMINIT_2” to the initialized “.data” RAM area:

```
.data : AT(_mdata)
{
    _data = .;
    *(.data)
    *(.data.*)
    *(D)
    *(D_1)
    *(D_2)

    /* Adding D sections for QE safety code */
    *(D_QE_TOUCH_SAFETY_RAMINIT)
    *(D_QE_TOUCH_SAFETY_RAMINIT_2)

    _edata = .;
} > RAM
```

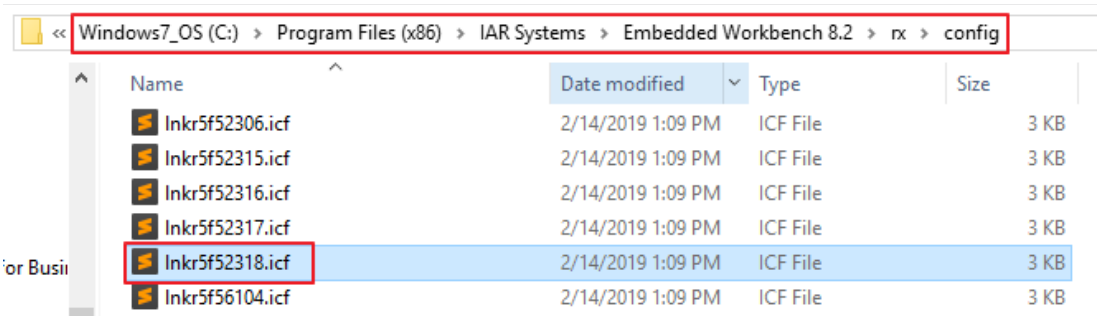
5) Add section “B_QE_TOUCH_SAFETY_RAM_2” to the uninitialized “.bss” RAM area:

```
.bss :
{
    _bss = .;
    *(.bss)
    *(.bss.**)
    *(COMMON)
    *(B)
    *(B_1)
    *(B_2)

    /* Adding B sections for QE safety code */
    *(B_QE_TOUCH_SAFETY_RAM_2)

    _ebss = .;
    _end = .;
} > RAM
```

6) NOTE: Please see BSP Application Note for possible additional sections to add or modify in linker script.

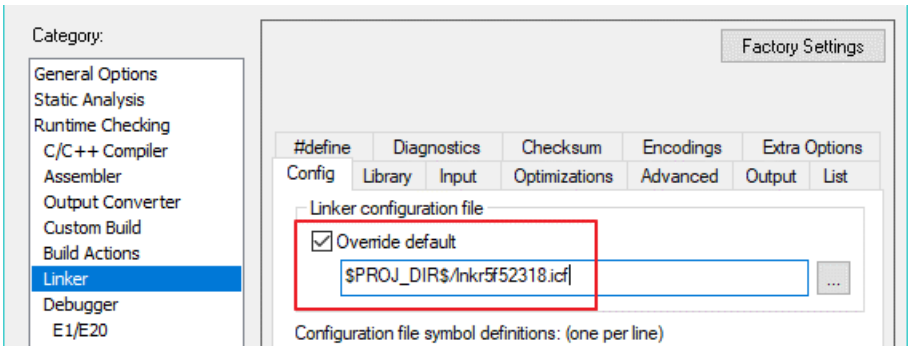
2.12.3 IAR Toolchain

1) In Windows Explorer, copy the template linker file for the target MCU device into the IAR Workbench project top level directory. Below is the default IAR Toolchain installation path with the example linker file outlined for an RSKRX231:



2) For the linker file to be included in the project, right-click on the project name and select Add->Add Files.

3) Once the file is added, right-click on the project name again and select Options. In the dialog box which opens, select the "Linker" category in the left pane, check "Override default" in the right pane, and change the path to the file just added:



4) Open the linker file in the editor and add the sections "D_QE_TOUCH_SAFETY_RAMINIT" and "D_QE_TOUCH_SAFETY_RAMINIT_2" to the "initialize by copy" area:

```
22
23  initialize by copy {    rw,
24                          ro section D,
25                          ro section D_1,
26                          ro section D_2,
27                          ro section D_2,
28                          ro section D_QE_TOUCH_SAFETY_RAMINIT,
29                          ro section D_QE_TOUCH_SAFETY_RAMINIT_2,
30                                                  };
```

5) Add the "C" and "P" sections to the "ROM32" area, and add the "D" and "B" sections to the "RAM32" area as shown below. Notice the differences in "ro" and "rw" section types:

```
55  "ROM32":place in ROM_region32       {  ro,
56                                        ro section C_QE_TOUCH_SAFETY_CONSTANT,
57                                        ro section C_QE_TOUCH_SAFETY_CONSTANT_1,
58                                        ro section C_QE_TOUCH_SAFETY_CONSTANT_2,
59                                        ro section P_QE_TOUCH_DRIVER
60                                                                          };
61  "RAM32":place in RAM_region32       { rw,
62                                        ro section D,
63                                        ro section D_1,
64                                        ro section D_2,
65                                        ro section D_QE_TOUCH_SAFETY_RAMINIT,
66                                        ro section D_QE_TOUCH_SAFETY_RAMINIT_2,
67                                        rw section B_QE_TOUCH_SAFETY_RAM_2,
68                                        block HEAP };
```

2.12.4 Configuration Files

For proper section name generation, the following settings must be made in the driver configuration files:

File “r_ctsu_qe_config.h”:

```
#define CTSU_CFG_SAFETY_LINKAGE_ENABLE    (1)
```

File “r_touch_qe_config.h”:

```
#define TOUCH_CFG_SAFETY_LINKAGE_ENABLE   (1)
```

2.13 IEC 60730 Compliance

For both R.1 (IEC 60335-1) and software class B (IEC 60730-1) compliance, the CTSU and Touch drivers along with the diagnostics code module must be used with the RX 60730 self-test code library(s) for the applicable MCU in the end application. The RX 60730 self-test code libraries can be found on the Renesas website https://www.renesas.com/us/en/ or by contacting Renesas support staff for additional information.

These self-test software libraries and application note numbers can be found on the Renesas website:

- RX100 Series VDE Certified IEC60730 Self-Test Code (R01AN2061ED)
- RX v2 Core VDE Certified IEC60730 Self-Test Code for RX v2 MCU (R01AN3364EG)

Note that the files “r_ctsu_qe_pinset.c” and “r_ctsu_qe_pinset.h” are generated by the Smart Configurator and are not considered part of these driver packages and are not required as part of 60730 compliance.

3. API Functions

3.1 Summary

The following functions are included in this design:

Function	Description
R_TOUCH_Open()	Initializes the Touch and CTSU layer modules.
R_TOUCH_UpdateDataAndStartScan()	Updates buffers with CTSU data from the scan that just completed and then starts another scan.
R_TOUCH_UpdateData()	Updates buffers with CTSU data from the scan that just completed.
R_TOUCH_Control()	Performs special operations defined by the command passed to it.
R_TOUCH_GetRawData()	Gets the sensor values as scanned by the CTSU (no correction or filters applied)
R_TOUCH_GetData()	Gets the sensor values as scanned by the CTSU after correction and filters are applied.
R_TOUCH_GetBtnBaselines()	Gets the sensor levels which a threshold offset is added to and checked for button touch conditions.
R_TOUCH_GetAllBtnStates()	This function loads a variable indicating which sensors are touched.
R_TOUCH_GetSliderPosition()	This function returns the current location of where the slider is being touched (0-100).
R_TOUCH_GetWheelPosition()	This function returns the current location of where the wheel is being touched (1-360).
R_TOUCH_Close()	Shuts down the Touch and CTSU layer modules.
R_CTSU_GetVersion()	Returns software version of driver.

3.2 R_TOUCH_Open

This function initializes the Touch layer and opens the CTSU.

Format

```
qe_err_t R_TOUCH_Open(ctsu_cfg_t   *p_ctsu_cfgs[],
                      touch_cfg_tg *p_touch_cfgs[],
                      uint8_t       num_methods,
                      qe_trig_t     trigger);
```

Parameters

p_ctsu_cfgs

Pointer to array of scan configurations for the CTSU driver (gp_ctsu_cfgs[ ] generated by the QE for Capacitive Touch tool)

p_touch_cfgs

Pointer to array of scan configurations for the TOUCH driver (gp_touch_cfgs[ ] generated by the QE for Capacitive Touch tool)

num_methods

Number of scan configurations in array (QE_NUM_METHODS generated by the QE for Capacitive Touch tool)

trigger

Scan trigger source (QE_TRIG_SOFTWARE or QE_TRIG_EXTERNAL)

Return Values

QE_SUCCESS:	*CTSU initialized successfully.*
QE_ERR_NULL_PTR:	*Missing argument pointer.*
QE_ERR_INVALID_ARG:	*"num_methods" or "trigger" is invalid (QE Tool generated p_ctsu_cfgs[] contents are not inspected)*
QE_ERR_BUSY:	*Cannot run because another CTSU operation is in progress.*
QE_ERR_ALREADY_OPEN:	*Open() called without intermediate call to Close().*
QE_ERR_UNSUPPORTED_CLK_CFG:	*Unsupported clock configuration.*
QE_ERR_ABNORMAL_TSCAP:	*TSCAP error detected during correction.*
QE_ERR_SENSOR_OVERFLOW:	*Sensor overflow error detected during correction.*
QE_ERR_SENSOR_SATURATION:	*Initial sensor value beyond linear portion of correction curve.*
QE_ERR_OT_MAX_OFFSET:	*Cannot tune SO0 offset any higher.*
QE_ERR_OT_MIN_OFFSET:	*Cannot tune SO0 offset any lower.*
QE_ERR_OT_WINDOW_SIZE:	*Tuning window too small. SO0 adjustments keep counts outside of window.*
QE_ERR_OT_MAX_ATTEMPTS:	*Maximum scans performed and all sensors still not in target window.*

Properties

Prototyped in file "r_touch_qe_if.h"

Description

This function initializes the QE TOUCH FIT module which makes calls to the lower level CTSU FIT module. This includes register initialization, enabling interrupts, and initializing the DTC if configured for operation in "r_touch_qe_config.h".

This function also runs an initial correction algorithm that is used for optimizing the QE Capacitive Touch tuning parameters. This is performed to account for small board-to-board variations as well as any minor environmental or physical differences in the Capacitive Touch application system.

Note that this function must be called before any other Touch API function.

Reentrant

No.

Example: Software Triggering

```
void main(void)
{
    qe_err_t    err;
    bool        success;
    uint32_t    cmt_ch;


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

    /* Open Touch driver (opens CTSU driver as well) */
    err = R_TOUCH_Open(gp_ctsu_cfgs, gp_touch_cfgs,
                                QE_NUM_METHODS, QE_TRIG_SOFTWARE);
    if (err != QE_SUCCESS)
    {
        while(1) ;
    }

    /* Setup scan timer for 20ms (50Hz) */
    success = R_CMT_CreatePeriodic(50, qetouch_timer_callback, &cmt_ch);
    if (success == false)
    {
        while(1) ;
    }


    /* Main loop */
    while(1)
    {
        if (g_qetouch_timer_flg == true)
        {
            g_qetouch_timer_flg = false;
            R_TOUCH_UpdateDataAndStartScan();

            /* process data here */
        }
    }

}
```

Example: External Triggering

```
void main(void)
{
    qe_err_t    err;
    elc_err_t           elc_err;
    elc_event_signal_t  ev_signal;
    elc_link_module_t   ev_module;


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

```
    /* Open Touch driver (opens CTSU driver as well) */
    err = R_TOUCH_Open(gp_ctsu_cfgs, gp_touch_cfgs,
                                QE_NUM_METHODS, QE_TRIG_EXTERNAL);
    if (err != QE_SUCCESS)
    {
        while(1) ;
    }


    /* Setup ELC for MTU timer to trigger CTSU */
    elc_err = R_ELC_Open();
    if (elc_err != ELC_SUCCESS)
    {
        while(1) ;
    }

    /* WARNING! Cannot use DTC when using external trigger! */
    ev_signal.event_signal = ELC_MTU2_CMP2A;
    ev_module.link_module = ELC_CTSU;
    elc_err = R_ELC_Set(&ev_signal, &ev_module);
    if (elc_err != ELC_SUCCESS)
    {
        while(1) ;
    }

    elc_err = R_ELC_Control(ELC_CMD_START, FIT_NO_PTR);
    if (elc_err != ELC_SUCCESS)
    {
        while(1) ;
    }


    /* Setup MTU timer for 20ms (50Hz) */
    mtu_timer_chnl_settings_t  my_timer_cfg;
    mtu_err_t  result;

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action =
                    (mtu_actions_t)(MTU_ACTION_INTERRUPT | MTU_ACTION_REPEAT);
    my_timer_cfg.timer_a.actions.do_action = MTU_ACTION_REPEAT;
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_NO_OUTPUT;
    my_timer_cfg.timer_a.freq              = 50; // 50 Hz.
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_2, &my_timer_cfg, FIT_NO_FUNC);
    result |= R_MTU_Control(MTU_CHANNEL_2, MTU_CMD_START, FIT_NO_PTR);
    if (result != MTU_SUCCESS)
    {
        while(1) ;
    }


    /* Main loop */
    while(1)
    {
        if (g_qetouch_timer_flg == true)
        {
```

```
            g_qetouch_timer_flg = false;
            err = R_TOUCH_UpdateData();

            /* process data here */
        }
    }

}
```

Special Notes:
Pins must be initialized prior to calling this function.

3.3 R_TOUCH_UpdateDataAndStartScan

This function is used to process data from the previous scan and start another scan by software trigger.

Format

```
qe_err_t  R_TOUCH_UpdateDataAndStartScan(void);
```

Parameters

None

Return Values

QE_SUCCESS:	*Data from previous scan updated and scan started successfully.*
QE_ERR_TRIGGER_TYPE:	*Function called when using external triggers.*
QE_ERR_BUSY:	*Cannot run because another CTSU operation is in progress. Most likely previous scan still running (lengthen timer interval).*
QE_ERR_ABNORMAL_TSCAP:	*TSCAP error detected during scan.*
QE_ERR_SENSOR_OVERFLOW:	*Sensor overflow error detected during scan.*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function is used when software triggering is specified at Open(). It should be called each time a periodic timer expires (see Section 3.2 Software Triggering example). The first several calls are used to establish a baseline for the sensors. Once that is complete, normal processing of data occurs. Similarly, when a different method is run, its first several calls to this function are used to establish a baseline for its sensors as well.

After processing CTSU scan data, another scan is initiated. If more than one method/scan configuration is present, the next method is automatically scanned. If a TOUCH_CMD_SET_METHOD Control() command was issued, that method continues to be scanned until another Control() command is issued to change it.

This function must be called prior to calling any “R_TOUCH_Get” function to get recent data.

Reentrant

No.

Example

```
qe_err_t err;

/* Process data from previous scan and initiate another by software trigger */
err = R_TOUCH_UpdateDataAndStartScan();
if (err != QE_SUCCESS)
{
    . . .
}
```

Special Notes:

None.

3.4 R_TOUCH_UpdateData

This function is used to process data from the previous scan.

Format

```
qe_err_t  R_TOUCH_UpdateData(void);
```

Parameters

None.

Return Values

QE_SUCCESS:	*Data from previous scan updated and scan started successfully.*
QE_ERR_BUSY:	*Cannot run because another CTSU operation is in progress. Most likely previous scan still running (lengthen timer interval).*
QE_ERR_ABNORMAL_TSCAP:	*TSCAP error detected during scan.*
QE_ERR_SENSOR_OVERFLOW:	*Sensor overflow error detected during scan.*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function is used when external triggering is specified at Open(). It should be called each time a scan completes (see Section 3.2 External Triggering example). The first several calls are used to establish a baseline for the sensors. Once that is complete, normal processing of data occurs. Similarly, when a different method is run, its first several calls to this function are used to establish a baseline for its sensors as well.

After processing CTSU scan data, if more than one method/scan configuration is present, the next method is automatically set to be scanned. If a TOUCH_CMD_SET_METHOD Control() command was issued, that method continues to be scanned until another Control() command is issued to change it.

This function must be called prior to calling any “R_TOUCH_Get” function to get recent data.

Reentrant

No.

Example

```
qe_err_t err;

/* Process data from previous scan */
err = R_TOUCH_UpdateData();
if (err != QE_SUCCESS)
{
    . . .
}
```

Special Notes:

None.

3.5 R_TOUCH_Control

This function is used to perform special operations pertaining to the control of the TOUCH driver.

Format

```
qe_err_t  R_TOUCH_Control(touch_cmd_t cmd, void *p_arg);
```

Parameters

cmd

Command to perform.

p_arg

Pointer to command-specific argument.

Return Values

QE_SUCCESS:	*Command completed successfully.*
QE_ERR_NULL_PTR:	*Command-specific required argument is missing.*
QE_ERR_INVALID_ARG:	*Command-specific argument is invalid.*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function is used to perform special operations with the Touch driver. The “cmd” commands are as follows:

TOUCH_CMD_SET_METHOD. Used to stop cycling through methods if was previously cycling, and sets the current method to be the only method used for scanning going forward. Equates are defined in “qe_common.h” and have the form QE_METHOD_xxx. The argument “p_arg” is a uint8_t pointer to a variable containing the method number to scan.

TOUCH_CMD_CYCLE_ALL_METHODS. Starts cycling through all methods present beginning with the method after the current one. The argument “p_arg” is unused.

TOUCH_CMD_CYCLE_METHOD_LIST. Starts cycling through specified list of methods only. The argument “p_arg” points to a structure of type touch_mlist_t.

```
typedef struct
{
    uint8_t     num_methods;                // number of methods to scan
    uint8_t     methods[QE_MAX_METHODS];    // methods in scan order
    uint8_t     cur_index;                  // (unused by application)
} touch_mlist_t;
```

TOUCH_CMD_GET_LAST_SCAN_METHOD. Gets method number of last completed scan. The argument “p_arg” is a uint8_t pointer to a variable to load the method into.

TOUCH_CMD_GET_FAILED_SENSOR. Used with R_TOUCH_Open() to identify first failed sensor for err return codes QE_ERR_ABNORMAL_TSCAP, QE_ERR_SENSOR_xxx, and QE_ERR_OT_xxx. The argument “p_arg” points to a structure of type touch_sensor_t.

```
typedef struct st_touch_sensor
{
    uint8_t         method;
    uint8_t         element;                // element index
} touch_sensor_t;
```

TOUCH_CMD_CLEAR_TOUCH_STATES. Clears the touch states of buttons, sliders, and wheels as stored internally by the driver. This is primarily used when a button has different functions (e.g. “on” or “off”) depending upon the method running, and want to ensure that states from the last time the method ran are not carried over. This is for applications which do not continuously cycle through all methods. The argument “p_arg” is a uint8_t pointer to a variable containing the method number to clear the touch states for.

Reentrant
No.

Example: TOUCH_CMD_SET_METHOD
```
qe_err_t err;
uint8_t  method
uint64_t btn_states1, btn_states2;

    . . .

    method = QE_METHOD_CONFIG02;
    R_TOUCH_Control(TOUCH_CMD_SET_METHOD, &method);
    R_CTSU_StartScan();


    /* Main loop */
    while(1)
    {
        if (g_qetouch_timer_flg == true)
        {
            g_qetouch_timer_flg = false;
            R_TOUCH_UpdateData();


            /* if currently running method 1 */
            if (method == QE_METHOD_CONFIG01)
            {
                /* if the ON button is touched, go to method 2 */
                R_TOUCH_GetAllBtnStates(QE_METHOD_CONFIG01, &btn_states1);
                if ((btn_states1 & CONFIG01_MASK_ON) != 0)
                {
                    method = QE_METHOD_CONFIG02;
                    R_TOUCH_Control(TOUCH_CMD_SET_METHOD, &method);
                }
            }

            /* else if currently running method 2 */
            else if (method == QE_METHOD_CONFIG02)
            {
                /* if the OFF button is touched, go to method 1 */
                R_TOUCH_GetAllBtnStates(QE_METHOD_CONFIG02, &btn_states2);
                if ((btn_states2 & CONFIG02_MASK_OFF) != 0)
                {
                    method = QE_METHOD_CONFIG01;
                    R_TOUCH_Control(TOUCH_CMD_SET_METHOD, &method);
                }
            }

            R_CTSU_StartScan();
        }
    }
```

Example: TOUCH_CMD_CYCLE_ALL_METHODS
```
    /* Resume cycling through all methods present */
    R_TOUCH_Control(TOUCH_CMD_CYCLE_ALL_METHODS, NULL);
```

Example: TOUCH_CMD_CYCLE_METHOD_LIST

```
    qe_err_t       err;
    touch_mlist_t  list;

    /* (QE_NUM_METHODS == 4)
    R_TOUCH_Open(gp_ctsu_cfgs, gp_touch_cfgs, QE_NUM_METHODS, QE_TRIG_SOFTWARE);


    /* Multiple models of product use same processing code.
     * Build application to run only methods tuned for specific model.
     */
#if (GLASS_VERSION == 1)
    list.methods[0] = QE_METHOD_GLASS_PANEL_A;     // (QE method 0)
    list.methods[1] = QE_METHOD_GLASS_PANEL_B;     // (QE method 1)
#else // PLASTIC version
    list.methods[0] = QE_METHOD_PLASTIC_PANEL_A;   // (QE method 2)
    list.methods[1] = QE_METHOD_PLASTIC_PANEL_B;   // (QE method 3)
#endif
    list.num_methods = 2;
    err = R_TOUCH_Control(TOUCH_CMD_CYCLE_METHOD_LIST, &list);
```

Example: TOUCH_CMD_GET_LAST_SCAN_METHOD

```
    uint8_t  last_method;

    /* Discover which method completed a scan last */
    R_TOUCH_Control(TOUCH_CMD_GET_LAST_SCAN_METHOD, &last_method);
```

Example: TOUCH_CMD_GET_FAILED_SENSOR

```
    qe_err_t       err;
    touch_sensor_t sensor_info;

    err = R_TOUCH_Open(gp_ctsu_cfgs, gp_touch_cfgs,
                                QE_NUM_METHODS, QE_TRIG_SOFTWARE);
    if (err != QE_SUCCESS)
    {
        /* check for hardware related issue */
        if ((err == QE_ERR_ABNORMAL_TSCAP)
         || (err == QE_ERR_SENSOR_OVERFLOW)
         || (err == QE_ERR_OT_MAX_OFFSET)
         || (err == QE_ERR_OT_MIN_OFFSET)
         || (err == QE_ERR_OT_WINDOW_SIZE)
         || (err == QE_ERR_OT_MAX_ATTEMPTS))
        {
            /* identify where first failure detected */
            R_TOUCH_Control(TOUCH_CMD_GET_FAILED_SENSOR, &sensor_info);

            /* failed method: sensor_info.method
             * bad sensor:    sensor_info.element
             */
        }
    }
```

Example: TOUCH_CMD_CLEAR_TOUCH_STATES

```
    uint8_t  last_method;

    if ((g_sys_state == SYS_STATE_LOW_PWR_MODE)
     && (change == CHANGE_TOUCH_DETECTED))
    {
        /* go to full power */
        g_method = QE_METHOD_FULL_PWR;
        R_TOUCH_Control(TOUCH_CMD_SET_METHOD, &g_method);
        R_TOUCH_Control(TOUCH_CMD_CLEAR_TOUCH_STATES, &g_method);
        g_sys_state = SYS_STATE_FULL_PWR_WAKEUP;
        R_CTSU_StartScan();
    }
```

Special Notes:

None.

3.6 R_TOUCH_GetRawData

This function gets the sensor values as scanned by the CTSU without correction or filters applied.

Format

```
qe_err_t  R_TOUCH_GetRawData(uint8_t    method,
                             uint16_t  *p_buf,
                             uint8_t   *p_cnt);
```

Parameters

method

Method from which to get data from.

p_buf

Pointer to buffer to load data into.

p_cnt

Pointer to variable to load number of words loaded into buffer.

Return Values

QE_SUCCESS:	*Sensors values read successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Method is invalid*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

WARNING! This function is for advanced users only.

This function gets data from the most recently successfully completed scan by the CTSU, without correction or filters applied.

Reentrant

No.

Example: Self Mode

```
qe_err_t  err;
uint16_t  sensor_buf[PANEL1_NUM_ELEMENTS];
uint8_t   cnt;

err = R_TOUCH_GetRawData(QE_METHOD_PANEL1, sensor_buf, &cnt);
```

Example: Mutual Mode

```
qe_err_t  err;
uint16_t  sensor_buf[PANEL1_NUM_ELEMENTS*2];
uint8_t   cnt;

err = R_TOUCH_GetRawData(QE_METHOD_PANEL1, sensor_buf, &cnt);
```

Special Notes:

None.

3.7 R_TOUCH_GetData

This function gets the sensor values as scanned by the CTSU with correction and filters applied.

Format

```
qe_err_t  R_TOUCH_GetData(uint8_t    method,
                          uint16_t  *p_buf,
                          uint8_t   *p_cnt);
```

Parameters

method

Method from which to get data from.

p_buf

Pointer to buffer to load data into.

p_cnt

Pointer to variable to load number of words loaded into buffer.

Return Values

QE_SUCCESS:	*Sensors values read successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Method is invalid*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

WARNING! This function is for advanced users only.

This function gets data from the most recently successfully completed scan by the CTSU, with correction and filters applied.

Reentrant

No.

Example: Self Mode

```
qe_err_t  err;
uint16_t  sensor_buf[PANEL1_NUM_ELEMENTS];
uint8_t   cnt;

err = R_TOUCH_GetData(QE_METHOD_PANEL1, sensor_buf, &cnt);
```

Example: Mutual Mode

```
qe_err_t  err;
uint16_t  sensor_buf[PANEL1_NUM_ELEMENTS];
uint8_t   cnt;

err = R_TOUCH_GetData(QE_METHOD_PANEL1, sensor_buf, &cnt);
```

Special Notes:

None.

3.8 R_TOUCH_GetBtnBaselines

This function gets the sensor levels which a threshold offset is added to and checked for button touch conditions.

Format

```
qe_err_t  R_TOUCH_GetBtnBaselines(uint8_t    method,
                                  uint16_t  *p_buf,
                                  uint8_t   *p_cnt);
```

Parameters

method

Method from which to get data from.

p_buf

Pointer to buffer to load data into.

p_cnt

Pointer to variable to load number of buttons which exist (buffer size).

Return Values

QE_SUCCESS:	*Long-term moving averages read successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Method is invalid*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

WARNING! This function is for advanced users only.

This function loads the baseline value for each button in its corresponding element index location in p_buf (non-button elements are not loaded). A baseline value is the sensor count which a threshold offset is added to for determining if a button touch condition exists. The baseline and threshold offset values are unique to each button. The baseline value is updated every xxx_DRIFT_FREQ number of scans (see “qe_<method>.h” file). It is equal to the moving average calculated over that number of scans.

Reentrant

No.

Example: Self or Mutual Mode

```
qe_err_t  err;
uint16_t  baseline_buf[PANEL_NUM_ELEMENTS];
uint8_t   cnt;

err = R_TOUCH_GetBtnBaselines(QE_METHOD_PANEL, baseline_buf, &cnt);
```

Special Notes:

None.

3.9 R_TOUCH_GetAllBtnStates

This function is used to get a mask of all buttons touched.

Format

```
qe_err_t  R_TOUCH_GetAllBtnStates(uint8_t method, uint64_t *p_mask);
```

Parameters

method

Method from which to get data from.

p_mask

Pointer to variable to load button-touched mask into. Button masks are defined in qe_<method>.h.

Return Values

QE_SUCCESS:	*Touched button mask loaded successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Invalid method*
QE_ERR_OT_INCOMPLETE:	*Offset tuning during Open() did not complete*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function loads the variable pointed to by “p_mask” with a bitmask of all the buttons touched. Button masks are defined in the QE generated method .h file.

Reentrant

No.

Example: Method is PANEL2

```
qe_err_t  err;
uint64_t  btn_touched_mask;

err = R_TOUCH_GetAllBtnStates(QE_METHOD_PANEL2, &btn_touched_mask);
```

Special Notes:

None.

3.10 R_TOUCH_GetSliderPosition

This function gets the current location of where a slider is being touched (if at all).

Format

```
qe_err_t  R_TOUCH_GetSliderPosition(uint16_t slider_id, uint16_t *p_position)
```

Parameters

slider_id

ID of slider to inspect. Slider IDs are defined qe_<method>.h.

p_position

Pointer to variable to load the slider position into. The position ranges from 0-100, low to high. A value of 65535 indicates the slider is not touched.

Return Values

QE_SUCCESS:	*Slider position read successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Slider ID is invalid*
QE_ERR_OT_INCOMPLETE:	*Offset tuning during Open() did not complete*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function returns the current location of where the slider is being touched.

Reentrant

No.

Example: Method is PANEL1. Slider is DIMMER.

```
qe_err_t  err;
uint16_t  slider_position;

err = R_TOUCH_GetSliderPosition(PANEL1_ID_DIMMER, &slider_position);
```

Special Notes:

None.

3.11 R_TOUCH_GetWheelPosition

This function gets the current location of where a wheel is being touched (if at all).

Format

```
qe_err_t  R_TOUCH_GetWheelPosition(uint16_t wheel_id, uint16_t *p_position);
```

Parameters

wheel_id

ID of wheel to inspect. Wheel IDs are defined qe_<method>.h.

p_position

Pointer to variable to load the wheel position into. Position ranges from 1 to 360 degrees. 65535 implies wheel is not being touched.

Return Values

QE_SUCCESS:	*Wheel position read successfully*
QE_ERR_NULL_PTR:	*Missing argument pointer*
QE_ERR_INVALID_ARG:	*Wheel ID is invalid*
QE_ERR_OT_INCOMPLETE:	*Offset tuning during Open() did not complete*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function returns the current location of where the wheel is being touched.

Reentrant

No.

Example: Method is PANEL2. Wheel is SERVO.

```
qe_err_t  err;
uint16_t  wheel_position;

err = R_TOUCH_GetWheelPosition(PANEL2_ID_SERVO, &wheel_position);
```

Special Notes:

None.

3.12 R_TOUCH_Close

This function closes the Capacitive Touch drivers at both the Touch and CTSU levels.

Format

```
qe_err_t  R_TOUCH_Close(void);
```

Parameters

None.

Return Values

QE_SUCCESS:	*The CTSU peripheral is successfully closed.*
QE_ERR_BUSY:	*Cannot run because another CTSU operation is in progress.*

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function closes the Touch driver as well as the CTSU driver and the peripheral. It disables interrupts associated with the CTSU and disables the clock to the peripheral.

Reentrant

No.

Example:

```
qe_err_t err;

err = R_TOUCH_Close();
```

Special Notes:

None.

3.13 R_TOUCH_GetVersion

Returns the current version of the QE TOUCH FIT module.

Format

```
uint32_t R_TOUCH_GetVersion(void);
```

Parameters

None.

Return Values

Version of the CTSU FIT module.

Properties

Prototyped in file “r_touch_qe_if.h”

Description

This function will return the version of the currently installed QE TOUCH module. The version number is encoded where the top 2 bytes are the major version number and the bottom 2 bytes are the minor version number. For example, Version 4.25 would be returned as 0x00040019.

Reentrant

Yes.

Example

```
uint32_t cur_version;

/* Get version of installed TOUCH API. */
cur_version = R_TOUCH_GetVersion();

/* Check to make sure version is new enough for this application’s use. */
if (MIN_VERSION > cur_version)
{
    /* This QE TOUCH API version is not new enough and does not have XXX feature
       that is needed by this application. Alert user. */
    ...
}
```

Special Notes:

This function is specified to be an inline function.

4. Demo Projects

Demo projects are complete stand-alone programs. They include function main() that utilizes the module and its dependent modules (e.g. r_bsp). The standard naming convention for the demo project is <module>_demo_<board> where <module> is the peripheral acronym (e.g. s12ad, cmt, sci) and the <board> is the standard RSK (e.g. rskrx113). For example, s12ad FIT module demo project for RSKRX113 will be named as s12ad_demo_rskrx113. Similarly the exported .zip file will be <module>_demo_<board>.zip. For the same example, the zipped export/import file will be named as s12ad_demo_rskrx113.zip

4.1 touch_demo_rskrx231

This is a simple e^2 studio v7.1 (or later) demo for the RSKRX231 starter kit using software triggers. Button and slider responses may not be accurate due to differences from original tuning environment and usage (e.g. expectation of how hard or how long a button is pressed when tuned; environmental factors, especially proximity to grounded surfaces).

Setup and Execution

1. Import, compile, and download project.
2. Add “btn_states” and “sldr_pos” to the Expressions window and set for Real-time Refresh.
3. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.
4. Observe changes in “btn_states” and “sldr_pos” as slider and buttons are touched.

Boards Supported

RSKRX231

4.2 touch_demo_rskrx231_ext_trig

This is a simple e^2 studio v7.1 (or later) demo for the RSKRX231 starter kit using external triggers. Button and slider responses may not be accurate due to differences from original tuning environment and usage (e.g. expectation of how hard or how long a button is pressed when tuned; environmental factors, especially proximity to grounded surfaces).

Setup and Execution

1. Import, compile, and download project.
2. Add “btn_states” and “sldr_pos” to the Expressions window and set for Real-time Refresh.
3. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.
4. Observe changes in “btn_states” and “sldr_pos” as slider and buttons are touched.

Boards Supported

RSKRX231

4.3 touch_demo_rskrx130

This is a simple e^2 studio v7.1 (or later) demo for the RSKRX130 starter kit using software triggers. Button and slider responses may not be accurate due to differences from original tuning environment and usage (e.g. expectation of how hard or how long a button is pressed when tuned; environmental factors, especially proximity to grounded surfaces).

Setup and Execution

1. Import, compile, and download project.
2. Add “btn_states” and “sldr_pos” to the Expressions window and set for Real-time Refresh.
3. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.
4. Observe changes in “btn_states” and “sldr_pos” as slider and buttons are touched.

Boards Supported

RSKRX130

4.4 touch_demo_rsskrx23w

This is a simple e^2 studio v7.5 (or later) demo for the RSSKRX23W starter kit using software triggers. Button and slider responses may not be accurate due to differences from original tuning environment and usage (e.g. expectation of how hard or how long a button is pressed when tuned; environmental factors, especially proximity to grounded surfaces).

Setup and Execution

5. Import, compile, and download project.
6. Add “btn_states” and “sldr_pos” to the Expressions window and set for Real-time Refresh.
7. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.
8. Observe changes in “btn_states” and “sldr_pos” as slider and button are touched.

Boards Supported

RSSKRX23W

4.5 Adding a Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. To add a demo project to a workspace, select File>Import>General>Existing Projects into Workspace, then click “Next”. From the Import Projects dialog, choose the “Select archive file” radio button. “Browse” to the FITDemos subdirectory, select the desired demo zip file, then click “Finish”.

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Oct.04.18	—	First edition issued
1.10	Jul.09.19	1,39	Added RX23W support
		4-6	Added definitions for “correction” and “offset tuning”.
		15, 18, 22, 30, 31	Updated API return values
		24, 26	Added TOUCH_CMD_GET_FAILED_SENSOR and TOUCH_CMD_GET_LAST_SCAN_METHOD Control() commands
		*	Moved offset tuning processing into R_TOUCH_Open().
		14, 16-20	Added #pragma section macros and configuration option to driver for Safety Module support (includes GCC/IAR support).
		1,20	Added IEC 6730 Compliance section.
		15,22	Added error code QE_ERR_UNSUPPORTED_CLK_CFG.
1.11	Jan.09.20	30-31	Updated example code.
		29,32	Added TOUCH_CMD_CLEAR_TOUCH_STATES for low power applications.
		5,6,22,35	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.

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 VIL (Max.) and VIH (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 VIL (Max.) and VIH (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 systemevaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc. Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products. (Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries. (Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035,
U.S.A. Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3 Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH,
U.K Tel: +44-1628-651-700

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543,
Taiwan Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949 Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038,
India Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265
Korea Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 7.2