

RX200, RX100 Series

R01AN3821EU0100

Rev 1.00

CTSU Module Using Firmware Integration Technology

May 9, 2017

Introduction

This Firmware Integration Technology (FIT) Module implements a Capacitive Touch Sensing Unit driver.

Target Device

RX231, RX230, RX130, RX113 with onboard capacitive touch sensing units.

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833EU)
- Board Support Package Module Using Firmware Integration Technology (R01AN1685EU)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723EU)
- Touch Module Using Firmware Integration Technology (R01AN3820EU)
- Touch Button Module Using Firmware Integration Technology (R01AN3822EU)
- Touch Slider Module Using Firmware Integration Technology (R01AN3823EU)
- Workbench6 User Manual (R20UT3842EJ)
- Capacitive Touch for RX Devices (R11AN0139EU)

Contents

1. Overview of the Capacitive Touch Software Environment	3
2. CTSU FIT Module	3
3. FIT Module Specifications.....	4
4. API Information	5
4.1 Hardware Requirements.....	5
4.2 Software Requirements.....	5
4.3 Limitations to this driver include:	5
4.4 Supported Toolchains	5
4.5 Header Files.....	5
4.6 Integer Types.....	5
4.7 Configuration Overview.....	5
4.8 Code Size	6
4.9 Parameters and Header files.....	7
4.10 Return Values	9
4.11 Events.....	10
4.12 Adding the FIT module to your project.....	10
5. API Functions	11
5.1 Summary	11
5.2 R_CTSU_Open.....	12
5.3 R_CTSU_Close	14
5.4 R_CTSU_Scan	16
5.5 R_CTSU_Read	18
5.6 R_CTSU_Control	21
5.7 R_CTSU_Calibrate	24
5.8 R_CTSU_GetVersion.....	26
6. Appendices.....	27
6.1 Sample CTSU Configuration (Self-Capacitance configuration).....	27
6.2 Sample CTSU Configuration (Mutual-Capacitance configuration).....	29

1. Overview of the Capacitive Touch Software Environment

The Capacitive Touch Sensing Unit (CTSUS) measures the electrostatic capacitance of a touch sensor. The entirety of the capacitive touch solution stack is comprised of three different components: The CTSUS layer, the touch layer, and the button/slider layer. The module covered in this documentation provides an abstraction layer for controlling the CTSUS special function registers (SFR), scanning the enabled Touch Sensor (TS) pins, and reading output generated for each sensor. This FIT CTSUS driver allows the user to operate the CTSUS in self-capacitance multi-scan mode or in mutual-capacitance full scan mode. This module comes with python (2.7) scripts; these translate optimized parameters generated by the Renesas Workbench calibration tool to data that can be input to this driver. For more information on the translation process, using python refer to R11AN0139EU.

2. CTSUS FIT Module

This module is intended for use along with the Renesas touch (r_touch) and button (r_touch_button) middleware as shown in the illustration below and is used by being incorporated into a FIT project when there is a need to operate/control the CTSUS.

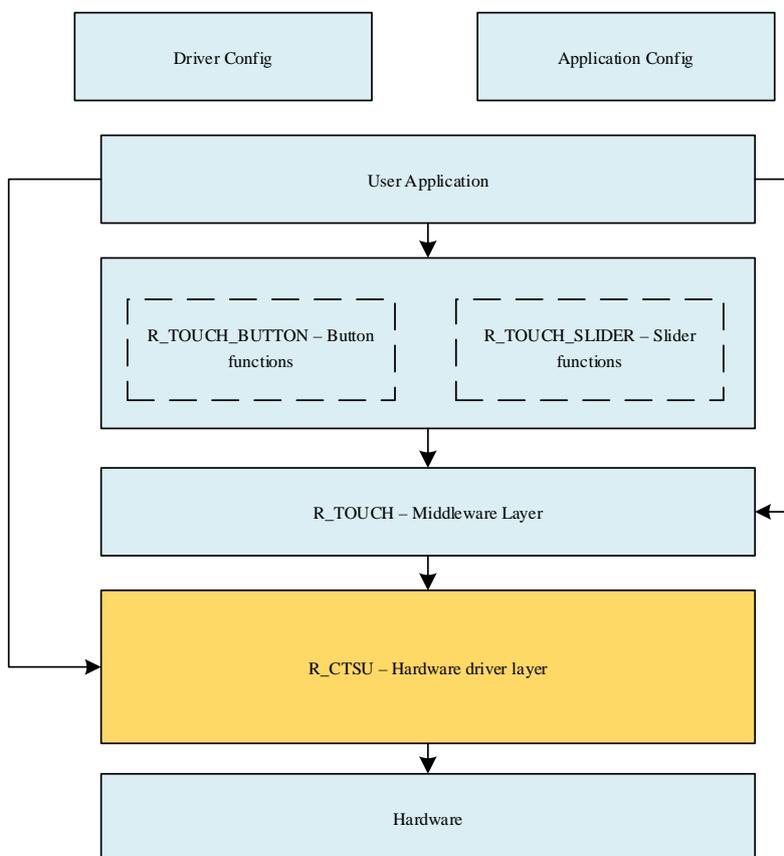


Figure 1: Overview of the capacitive touch solution and the various FIT modules

The way the driver interacts with the user application is described in Figure 1, the highlighted part is the r_ctsu_rx FIT module which is described as the CTSUS Layer. This layer is responsible for interpreting the counts from the hardware layer and transferring them via the data transfer controller. It contains functions that facilitate the opening, closing, reading and control of the hardware peripheral and puts limits on the digital counts and tuning offsets. At this layer is where the CTSUS control block is held, it is a data structure that contains the configuration data that operates the hardware. For API documentation on the touch, button and slider layers refer to the following documents respectively- R01AN3820EU, R01AN3822EU, and R01AN3823EU.

Similar to the configuration of other MCU peripherals such as the analog to digital converter, the first step to configuring the CTSUS after power on reset is to configure the appropriate pins as touch sensing (TS) and TSCAP pins. After proper input/output (I/O) initialization, the user must then open the CTSUS configuration provided by Workbench

via the tuning process. This configuration block holds the SFR settings, sensor data specific to each sensor, the peripheral clock setting, and the settings for each active sensor.

Through the open function, the user passes the configuration they wish to open and a pointer to where an identifier for the configuration is to be stored, the driver will then be configured to that active configuration. This document refers to the configuration ID pointer of the locally stored control block array in RAM as a 'handle.' The handle is always passed as the first argument to any CTSU API function.

After the open command loads the configuration into memory, the user then interacts with the CTSU layer through the touch and button middleware layers, which call the scan and read functions. There is one configuration block per group of sensors in a given mode of operation (self -or- mutual capacitance). It is expected that typically the sensor configuration blocks will be created/configured by the calibration tool, Workbench6. An example of the initialization process is shown in Figure 2.

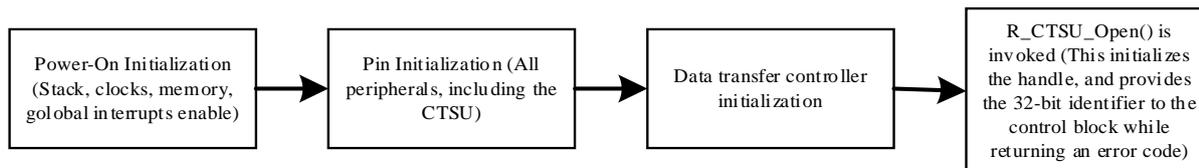


Figure 2: Example initialization process

3. FIT Module Specifications

- Supports self-capacitance and mutual capacitance mode of operation.
- Operation in Self-capacitance multi-scan mode
- Operation in Mutual capacitance full scan mode
- Multiple configurations can be opened at a given time using the R_CTSU_Open function.
- Information provided in a configuration is checked against previous and a new control block is opened only if the information is unique.
- The maximum number of unique configurations open can be specified through CTSU_CFG_MAX_CONTROL_BLOCK_COUNT in r_ctsu_rx_config.h.
- Locking mechanism in control blocks prevents hardware from being used simultaneously.
- CTSU Interrupts are enabled and low-power operation is disabled when first control block is opened.
- CTSU Interrupts are disabled and low-power operation is enabled when all control blocks are closed.
- A callback can be provided to indicate start and completion of a scan cycle.
- CTSU measurement data is output to user-specified location provided through the configuration passed to the R_CTSU_Open function as argument.
- Supports data transfer using the Data Transfer Controller.
- Provides API to calculate MCU-to-MCU CTSU variation compensation factor.
- Error messages can be asserted onto the Renesas Virtual Debug Console by setting BSP_CFG_IO_LIB_ENABLE to 1 in the board support package.

4. API Information

This driver follows the Renesas API naming standards

4.1 Hardware Requirements

This driver requires that your MCU support the following features:

- CTSU peripheral
- (Optional) DTC

4.2 Software Requirements

This driver is dependent upon the following packages:

- r_bsp

4.3 Limitations to this driver include:

- This module does not support CTSU operation in self-capacitance single scan mode.
- Port pins to use as TSCAP and TS Pins must be configured using the general purpose input output (GPIO) and multi-function pin controller (MPC) FIT modules.
- When the data transfer controller (DTC) is used, the DTC vector table (DTCVBR) indexes 60 and 61 should be mutable.

4.4 Supported Toolchains

This driver is tested and works with the following toolchain:

- Renesas RX Toolchain v.2.05

4.5 Header Files

All API calls and their supporting interface definitions are located in r_cts_rx_if.h

4.6 Integer Types

This module uses ANSI C99. These types are defined instdint.h

4.7 Configuration Overview

The configuration options in this module are specified in r_cts_rx_config.h. The option names and setting values are listed in the table below where applicable 0 represents disabled and 1 represents enabled:

Option	Description
CTS_CFG_PARAM_CHECKING_ENABLE	Select whether to perform parameter checking on function call. Default=Enabled.
CTS_CFG_MAX_CONTROL_BLOCK_COUNT	Select maximum control blocks allowed to open. Default=1
CTS_INTERRUPT_PRIORITY_LEVEL	Select Interrupt priority level for the CTSUWR, CTSURD, and CTSUFN interrupts. Default=8;
CTS_CFG_ENABLE_CORRECTION	Select if code to calculate correction factor is included in the build, and apply correction factor to measured data. Default=Disabled (Refer special notes for R_CTSU_Calibrate)
CTS_CFG_CALIBRATION_ENABLE	Select if code for R_CTSU_Calibrate is included in the build. This allow fine calibration of CTSU count readings. Default = Disabled

Option	Description
CTS_CFG_DTC_TRANSFER_ENABLE	Select if DTC is used for transferring data from RAM to SFRs and SFRs to RAM during CTSUWR and CTSURD interrupts. At least 16 bytes of additional memory is allocated to the control block. (Default=Disabled)
CTS_CFG_DTC_CHAIN_TRANSFERS	Select if DTC chain transfers are used to reset transfer control blocks. An additional 32 bytes of memory is allocated to each control block. (Default = Disabled)

4.8 Code Size

The table lists values when the compile options are set to default values (with parameter checking disabled) and the RX130 Group is used. The required memory size varies depending on the C compiler version and compile options. The table does not reflect memory consumed by the configuration as it varies depending upon the sensor count.

Memory Type	Size (bytes)
ROM	2011
RAM	72
Maximum User Stack usage	64
Maximum Interrupt Stack usage	60

4.9 Parameters and Header files

This section describes the structure instances, which are defined by the user while creating a touch configuration. This structure is located in `r_ctsus_rx_if.h` as are the prototype declarations of API functions. An example of the of the overall CTSUS control block configuration is found below:

```

/** Structure defining a configuration to be passed to the Open function. */
typedef struct st_ctsus_cfg
{
    ctsus_const_sfrs_t * p_ctsus_settings;          //User defined SFR
                                                    //settings Common to all
sensors

    ctsus_sensor_setting_t * p_sensor_settings; //Pointer to an array with
                                                    //settings unique to each
sensor.

    uint16_t * p_sensor_data;                      // User defined memory location
                                                    // where CTSUS data is output.

    uint32_t   pclkb_hz;                          // Frequency of the PCLKB when
                                                    // configuration is valid.

    fit_callback_t      p_callback;               // Pointer to callback function for
                                                    // event notifications.

    uint16_t correction_factor_pri_cal;           // Correction factor for primary
sensor
                                                    // count

    uint16_t correction_factor_sec_cal;           // CTSUS correction factor for
secondary
                                                    // sensor count

    uint8_t correction_ctsuso_delta;             // Change in ctsuso for measuring
                                                    // correction factor.

    uint16_t const * const p_correction_ideal_sensor_ico; // CTSUS
correction
                                                    // factor for primary sensor count
                                                    // readings measured from MCU used for
                                                    // calibration.

    uint16_t * const p_correction_buffer; // CTSUS correction factor for
secondary
                                                    // sensor count readings measured
from
                                                    // MCU used for calibration.

    uint8_t const tscal_pin;                     // Pin used for calculation of CTSUS
                                                    // correction factor.
} ctsus_cfg_t;

```

An example of the structure containing the SFRs common to that configuration of the CTSU is represented below.

```
/** Structure defining SFRs common to a CTSU mode. */
typedef struct st_ctsu_const_sfrs {
union { unsigned char BYTE;} CTSUCR0;          // CTSU Control Register 0

union { unsigned char BYTE;} CTSUCR1;          // CTSU Control Register 1

union { unsigned char BYTE;} CTSUSDPRS;        // CTSU Synchronous Noise Reduction
// Setting Register

union { unsigned char BYTE;} CTSUSST;         // CTSU Sensor Stabilization Wait
// Control Register

union { unsigned char BYTE;} CTSUCHAC0;        // CTSU Channel Enable Control
// Register 0

union { unsigned char BYTE;} CTSUCHAC1;        // CTSU Channel Enable Control
// Register 1

union { unsigned char BYTE;} CTSUCHAC2;        // CTSU Channel Enable Control
// Register 2

union { unsigned char BYTE;} CTSUCHAC3;        // CTSU Channel Enable Control
// Register 3

union { unsigned char BYTE;} CTSUCHAC4;        // CTSU Channel Enable Control
// Register 4

union { unsigned char BYTE;} CTSUCHTRC0;       // CTSU Channel Transmit/Receive
// Control Register 0

union { unsigned char BYTE;} CTSUCHTRC1;       // CTSU Channel Transmit/Receive
// Control Register 1

union { unsigned char BYTE;} CTSUCHTRC2;       // CTSU Channel Transmit/Receive
// Control Register 2

union { unsigned char BYTE;} CTSUCHTRC3;       // CTSU Channel Transmit/Receive
// Control Register 3

union { unsigned char BYTE;} CTSUCHTRC4;       // CTSU Channel Transmit/Receive
// Control Register 4

union { unsigned char BYTE;} CTSUDCLKC;        // CTSU High-Pass Noise Reduction
// Control Register

}ctsu_const_sfrs_t;
```

An example of the structure, which holds values to be written to the CTSU SFRs, such as the offsets, is represented below.

```
/** Structure to be holding values to be written to the SFRs in the WR ISR*/
typedef struct st_sensor_setting
{
    volatile uint16_t ctsussc; // CTSU High-Pass Noise Reduction Spectrum
                             // Diffusion Control

    volatile uint16_t ctsuso0; // CTSU Sensor Offset Register 0

    volatile uint16_t ctsusol; // CTSU Sensor Offset Register 1
} ctsu_sensor_setting_t;
```

4.10 Return Values

This section describes the return values of API functions. This enumeration is located in `r_ctsu_rx_if.h` as are the prototype declarations of API functions. The enumerations for the return codes are contained below.

```
/** Different error status returned by the API. */
typedef enum e_ctsu_err
{
    CTSU_SUCCESS = 0, // No Errors.

    CTSU_ERR_INVALID_PARAM = 0x0001, // Received invalid information
                                     // for important parameter.

    CTSU_ERR_INSUFFICIENT_MEMORY = 0x0002, // No more control blocks
                                             // available.

    CTSU_ERR_CTSU_OPERATIONAL = 0x0004, // Attempted operation while CTSU
                                          // is performing scan measurement.

    CTSU_ERR_LOCKED = 0x0004, // Attempted to use a locked
                              // handle.

    CTSU_ERR_INVALID_CMD = 0x0008, // Command requested is invalid.

    CTSU_ERR_CTSU_SC_OVF = 0x0010, // Sensor count overflowed when
                                    // performing CTSU scan.

    CTSU_ERR_CTSU_RC_OVF = 0x0020, // Reference count overflowed when
                                    // performing CTSU scan.

    CTSU_ERR_CTSU_ICOMP = 0x0040, // Abnormal TSCAP voltage.

    CTSU_ERR_CORRECTION_FACTOR = 0x0080, // Failed to calculate correction
                                          // factor.
} ctsu_err_t;
```

4.11 Events

The CTSU layer generates events in response to certain conditions. Upper layers are notified when a callback function is specified. The following events are generated by this layer.

Event Type	Notes
CTSU_EVENT_SFERS_CHANGED	Generated when CTSU SFRs have changed from the previous setting
CTSU_EVENT_STARTING_SCAN	Generated when a CTSU scan is started by invoking R_CTSU_Start_Scan
CTSU_EVENT_SCAN_COMPLETED	Generated when a CTSU scan completes. NOTE: The complexity of operations performed when this event is generated should be kept to a minimum.

An example of the structure that holds the call back arguments, which communicate the handle number, event, and error codes, is represented below.

```
typedef struct st_ctsu_callback_arg
{
    int          handle_num; // Handle identifier which generated this
                        // event.

    uint32_t     event_mask; // Event code mask.

    uint32_t     info;       // Error code (if any).
} ctstu_callback_arg_t;
```

As seen above, the event code is located in the second long word.

4.12 Adding the FIT module to your project

This section outlines the process for integrating the generated FIT style drivers into an already created project, after the board has been tuned using Workbench6. For information regarding instillation for the entirety of the driver stack, please refer to (R11AN0138EU).

1. Ensure that the User Project uses FIT, and is based on r_bsp v3.40 or later.
2. Copy the modules r_ctsu_rx to the FIT project from the Base Project.
 - This includes adding the following locations to the compile include paths:
`${workspace_loc}/${ProjName}/r_ctsu_rx`
 - As well as copying the following files from the Base Project's r_config folder to the User Project:
`r_ctsu_rx_config.h`
3. Ensure that the Parameter Checking preprocessor is enabled for all layers using the file mentioned in item 4. As a note this can be turned off once all layers are verified to be operate correctly.
4. Copy the folder src/ctsu from the Base Project and place them in the User Project.
5. Ensure that the clock settings in r_bsp_config.h match for both the Base and User Project.
6. Initialize all CTSU pins using the r_mpc_rx and r_gpio_rx FIT modules.

When using the FIT module, the BSP FIT module also needs to be added. For details on the BSP FIT module, refer to the "Board Support Package Module Using Firmware Integration Technology" application note (R01AN1685EU).

5. API Functions

5.1 Summary

Table below lists the API functions.

Function	Description
R_CTSU_Open	Initialize memory local to module, i.e. a control block, and save argument parameters. Some argument parameters must remain valid until control block is in use.
R_CTSU_Close	Close a control block that was previously opened.
R_CTSU_Scan	Load parameters from a control block to the CTSU SFRs and scan all enabled TS pins.
R_CTSU_Read	Read CTSU measurement data generated by the latest scan.
R_CTSU_Control	Change contents of a control block.
R_CTSU_Calibrate	Calibrate TS pins enabled in a control block.

5.2 R_CTSU_Open

This function is required to be invoked first before any other API calls with this module are used.

Format

```
r_ctsu_err_t R_CTSU_Open (uint32_t * p_hdl, r_ctsu_cfg_t * p_ctsu_cfg)
```

Parameters

- p_hdl – Pointer to a 32-bit integer. When Open successfully initializes a control block, the index of the control block will be stored at this location. This index must be used when API is to be used with the values passed through the second argument.
- p_ctsu_cfg – Pointer to a CTSU sensor configuration. This configuration contains all information needed to operate the CTSU and a set of TS pins in a given mode of operation.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_INVALID_PARAM – Invalid argument/parameter detected.
- CTSU_ERR_INSUFFICIENT_MEMORY – Could not assign control block. All control blocks are in use.
- CTSU_ERR_LOCKED – Attempted operation on a locked control block.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

This function checks parameters being used, powers up CTSU, and initializes an unused CTSU control block for operation.

1. Check if parameters match previously initialized control blocks. If true, return matching control block index.
2. Get an unused control block from array.
3. Initialize/Copy local information from provided configuration into control block.
4. (optional) Initialize the control block's DTC TCB (chain) for CTSUWR IRQ.
5. (optional) Initialize the control block's DTC TCB (chain) for CTSURD IRQ.
6. (optional) Enable the use of DTC to move data when CTSUWR interrupt occurs using DTC Activation Enable Register bits.
7. (optional) Enable the use DTC to move data when CTSURD interrupt occurs using DTC enable bits.
8. Power ON the CTSU
9. Enable CTSU interrupt operation.
10. Indicate control block has been successfully opened and return the control block number to user.

Re-entrant

Function is re-entrant for different configurations passed.

Example

```
void my_func(void)
{
    uint32_t hdl_idx = UINT32_MAX;
    ctsu_cfg_t * p_ctsu_cfg;
    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;
    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;
    if(CTSUSUCCESS == R_CTSU_Open(&hdl_idx, p_ctsu_cfg))
    {
        // Actions on a successful open
    }
}
```

Special Notes

Parameter locations passed through the configuration must remain valid until control block is closed.

5.3 R_CTSU_Close

This function closes an open control block, freeing the memory occupied by that control block.

Format

```
r_ctsu_err_t R_CTSU_Close (uint32_t hdl)
```

Parameters

- hdl - Handle identifier provided to user by a successful call to R_CTSU_Open function.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_INVALID_PARAM – Invalid argument/parameter detected.
- CTSU_ERR_LOCKED – Attempted operation on a locked control block.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

1. Check if this control block is in use.
2. Clean the control block information using memset.
3. Exit if any other control blocks are open.
4. If no more control blocks are open it then:
 - A. Re-initialize the CTSU special function registers (SFR).
 - B. (optional) Disable use of the DTC to move data when CTSUWR interrupt occurs
 - C. (optional) Disable use of the DTC to move data when CTSURD interrupt occurs
 - D. Disable CTSU interrupt operation.
 - E. Unlock Hardware (if it was left as locked somehow).
 - F. Power OFF the CTSU

Re-entrant

Function is reentrant for different identifiers passed. Locking mechanism prevents two APIs accessing the same block while in use.

Example

```
void my_func(void)
{
    uint32_t hdl_idx = UINT32_MAX;
    ctsu_err_t ctsu_err;
    ctsu_cfg_t * p_ctsu_cfg;
    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;
    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;
    if(CTSU_SUCCESS == R_CTSU_Open(&hdl_idx, p_ctsu_cfg))
    {
        ctsu_err = R_CTSU_Close(hdl_idx);
    }
}
```

Special Notes

None.

5.4 R_CTSU_Scan

Perform a single scan with a parameters stored in a successfully opened control block.

Format

```
r_ctsu_err_t R_CTSU_Scan (uint32_t hdl)
```

Parameters

- hdl - Handle identifier provided to user by a successful call to R_CTSU_Open function.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_INVALID_PARAM – Invalid argument/parameter detected.
- CTSU_ERR_LOCKED – Attempted operation on a locked control block.
- CTSU_ERR_CTSU_OPERATIONAL – CTSU hardware is currently scanning.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

1. Select the control block.
2. Lock the hardware.
3. Lock the control block as the raw_result buffer is about to be updated.
4. If control block is different from previous or the reload flag for the control block is set,
 - A. Load new CTSU SFR settings.
 - B. Set the DTC vectors to point to the chain in the control block.
 - C. Set the global control block identifier (which is used in the ISRs).
5. If DTC TCB chains are not used then initialize TCB0 manually for both interrupts.
6. If User has provided a callback, Notify User.
7. Start CTSU scan operation using CTSUSTRT bit.

Re-entrant

Function is reentrant for different identifiers passed. Locking mechanism prevents two APIs accessing the same block while in use.

Example

```
void my_func(void)
{
    uint32_t hdl_idx = UINT32_MAX;
    ctsu_err_t ctsu_err;
    ctsu_cfg_t * p_ctsu_cfg;
    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;
    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;
    if(CTSU_SUCCESS == R_CTSU_Open(&hdl_idx, p_ctsu_cfg))
    {
        ctsu_err = R_CTSU_Scan(hdl_idx);
        delay();
    }
}
```

Special Notes

Interrupts must be enabled globally before this function is invoked.

5.5 R_CTSU_Read

Copy the results generated by latest CTSU scan operation to user provided destination.

Format

```
r_ctsu_err_t R_CTSU_Read(uint32_t usr_hdl, ctsu_read_t * p_arg)
```

Parameters

- hdl - Handle identifier provided to user by a successful call to R_CTSU_Open function.
- p_arg – Argument providing information about information to be read from latest scan.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_INVALID_PARAM – Invalid argument/parameter detected.
- CTSU_ERR_LOCKED – Attempted operation on a locked control block.
- CTSU_ERR_INVALID_CMD – Read command requested is not recognized.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

1. Select the control block (exit if locked)
2. Calculate memory required to copy
3. Check if the control block is in use
4. Verify the result type requested and copy the requested results into the user provided buffer location.
5. Unlock the control block so it can be used by other CTSU functions.

Below is a list of commands you can send to this function:

Command (result type)	Description
CTSU_READ_SENSOR_COUNT_PRIMARY	Read the primary sensor counters for sensor offsets provided in the r_ctsu_read_result_t::sensor_offset array.
CTSU_READ_SENSOR_COUNT_SECONDARY	Read the primary sensor counters for sensor offsets provided in the r_ctsu_read_result_t::sensor_offset array.
CTSU_READ_SENSOR_OUTPUT	Read the count representing the electrostatic capacitance (Self: = SCPRIMARY Mutual: = SCSECONDARY - SCPRIMARY) for sensor offsets provided in the r_ctsu_read_result_t::sensor_offset array.
CTSU_READ_REFERENCE_COUNT_PRIMARY	Read the primary reference counters value for the sensor offsets provided in the r_ctsu_read_result_t::sensor_offset array.
CTSU_READ_REFERENCE_COUNT_SECONDARY	Read the primary reference counters value for the sensor offsets provided in the r_ctsu_read_result_t::sensor_offset array.
CTSU_RESULT_BUFFER_ALL	Read latest counter data generated by the CTSU for all sensors as generated by CTSU

Re-entrant

Function is reentrant for different identifiers passed. Locking mechanism prevents two APIs accessing the same block while in use.

Example

```

void my_func(void)
{
    uint32_t    hdl_idx = UINT32_MAX;
    ctsu_err_t  ctsu_err;
    ctsu_cfg_t * p_ctsu_cfg;
    ctsu_read_t read_arg;
    uint16_t    sensor_result;
    uint16_t    sensor_offset = 0;

    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;

    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;

    if(CTSUS_SUCCESS == R_CTSUS_Open(&hdl_idx, p_ctsu_cfg))
    {
        // More actions on successful open

        ctsu_err = R_CTSUS_Scan(hdl_idx);
        delay();
        read_arg.p_dest = &sensor_result;
        read_arg.size = sizeof(sensor_result);
        read_arg.sensor_offset = &sensor_offset;
        read_arg.count = 1;
        read_arg.cmd = CTSUS_READ_SENSOR_OUTPUT;
        ctsu_err = R_CTSUS_Read(hdl_idx, &read_arg);
    }
}

```

Special Notes

The second argument is of type `ctsus_read_t` with a structure definition as shown below:

```

typedef struct st_ctsus_read
{
    void* p_dest;           // Pointer to user buffer where a copy of the
    result                 // should be created.
    size_t size;           // Size of the user buffer provided (in bytes).
    r_ctsus_read_cmd_t cmd; // The result type user wants to read.
    uint8_t count;         // Number of valid indexes in
                           // st_ctsus_read::sensor_offset.
    uint16_t * sensor_offset; // Array of sensor offsets (index).
}ctsus_read_t;

```

5.6 R_CTSU_Control

A control block contains parameters that are initialized privately. This function provides access to these parameters so that they may be changed when the control block is not in use.

Format

```
r_ctsu_err_t R_CTSU_Control (uint32_t usr_hdl, r_ctsu_control_arg_t * p_arg)
```

Parameters

- hdl - Handle identifier provided to user by a successful call to R_CTSU_Open function.
- p_arg – Argument specifying the parameter in the control block to change.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_INVALID_PARAM – Invalid argument/parameter detected.
- CTSU_ERR_INVALID_CMD – Read command requested is not recognized.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

1. Select the control block
2. Read or write value depending upon GET/SET command respectively.

The following table illustrates commands that use one argument only.

Command	Description	Return Type
CTSU_CMD_GET_MODE	Read the mode of operation for the specified handle	ctsu_mode_t
CTSU_CMD_GET_PCLK	Read the PCLK frequency at which the handle can be operated	uint32_t
CTSU_CMD_GET_SENSOR_COUNT	Read the number of touch sensors enabled in a handle	uint16_t
CTSU_CMD_GET_ERROR_INFORMATION	Read the error information from handle during last scan	uint32_t
CTSU_CMD_GET_RESULT_OUTPUT_BUFFER	Read the location address set for measurement data output	void*
CTSU_CMD_GET_CALLBACK	Read the callback function from handle	fit_callback_t
CTSU_CMD_GET_CTSUCR0	Read SFR CR0 for handle	uint8_t
CTSU_CMD_GET_CTSUCR1	Read SFR CR1 for handle	uint8_t
CTSU_CMD_GET_CTSUSDPRS	Read SFR SDPRS for handle	uint8_t
CTSU_CMD_GET_CTSUSST	Read SFR SST for handle	uint8_t
CTSU_CMD_GET_CTSUCHAC	Read SFR CHAC for handle	uint64_t
CTSU_CMD_GET_CTSUCHTRC	Read SFR CHTRC for handle	uint64_t
CTSU_CMD_GET_CTSUDCLKC	Read SFR DCLKC for handle	uint8_t
CTSU_CMD_SET_RESULT_OUTPUT_BUFFER	Change the location address used for measurement data output	void*
CTSU_CMD_SET_CALLBACK	Change the callback function for a handle	fit_callback_t
CTSU_CMD_SET_CTSUCR0	Write SFR CR0 to handle	uint8_t
CTSU_CMD_SET_CTSUCR1	Write SFR CR1 to handle	uint8_t
CTSU_CMD_SET_CTSUSDPRS	Write SFR SDPRS to handle	uint8_t
CTSU_CMD_SET_CTSUSST	Write SFR SST to handle	uint8_t
CTSU_CMD_SET_CTSUCHAC	Write SFR CHAC to handle	uint64_t
CTSU_CMD_SET_CTSUCHTRC	Write SFR CHTRC to handle	uint64_t
CTSU_CMD_SET_CTSUDCLKC	Write SFR DCLKC to handle	uint8_t

The following table lists commands that need a structured argument extension (r_ctsu_control_arg_ext_t) to be provided.

Command	Description	Return Type
CTSU_CMD_GET_CTSUSSC	Read SFR SSC for sensor in handle	uint16_t
CTSU_CMD_GET_CTSUSO0	Read SFR SO0 for sensor in handle	uint16_t
CTSU_CMD_GET_CTSUSO1	Read SFR SO1 for sensor in handle	uint16_t
CTSU_CMD_SET_CTSUSSC	Write SFR SSC to sensor in handle	uint16_t
CTSU_CMD_SET_CTSUSO0	Write SFR SO0 to sensor in handle	uint16_t
CTSU_CMD_SET_CTSUSO1	Write SFR SO1 to sensor in handle	uint16_t

Re-entrant

Function is reentrant for different identifiers passed.

Example

```

void my_func(void)
{
    uint32_t    hdl_idx = UINT32_MAX;
    ctsu_err_t  ctsu_err;
    ctsu_cfg_t * p_ctsu_cfg;
    ctsu_control_arg_t arg;
    uint16_t    num_sensors = 0;

    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;

    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;
    if(CTSUS_SUCCESS == R_CTSUS_Open(&hdl_idx, p_ctsu_cfg))
    {
        arg.p_context = &num_sensors;
        arg.cmd = CTSUS_CMD_GET_SENSOR_COUNT;
        ctsu_err = R_CTSUS_Control(hdl_idx, &read_arg);
    }
}

```

Special Notes

This function does not check the locks in the control block. It is recommended to ensure the control block is not being used when this function call is invoked.

The addition argument taken by this function has a structure as shown below:

```

/** Structure passed to the control function */
typedef struct st_ctsu_control_arg
{
    r_ctsu_cmd_t cmd;          // Type of control command to execute
    void* p_extend;          // Extended arguments passed into the control command
}r_ctsu_control_arg_t;

```

Some commands may require more than one argument. In these cases, the following structure instance must be passed through the p_extend:

```

/** Structure passed to the control function as extension to context. Used
when r_ctsu_control_arg_t::cmd is either CTSUS_CMD_SET_CTSUSSC,
CTSUS_CMD_SET_CTSUSO0, or CTSUS_CMD_SET_CTSUSO1 */

typedef struct st_ctsu_control_arg_ext
{
    uint16_t offset;          // Offset of sensor
    uint16_t value;          // Value of special function register.
}r_ctsu_control_arg_ext_t;

```

5.7 R_CTSU_Calibrate

Calibrate the CTSU control block and calculate CTSU Correction factor. It is recommended that this function is called after a successful call to the R_CTSU_Open function.

Format

```
r_ctsu_err_t R_CTSU_Calibrate (uint32_t usr_hdl)
```

Parameters

- hdl - Handle identifier provided to user by a successful call to R_CTSU_Open function.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_CORRECTION_FACTOR – Failed to calculate correction factor.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

Calibration involves changing the CTSUSO SFR bit for each sensor such that the primary sensor count ~ primary reference count measured by the CTSU. Then the correction factor to compensate for MCU-to-MCU variations is calculated.

Re-entrant

Function is reentrant for different identifiers passed.

Example

```
void my_func(void)
{
    uint32_t hdl_idx = UINT32_MAX;
    ctsu_err_t ctsu_err;
    ctsu_cfg_t * p_ctsu_cfg;
    extern ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self;
    p_ctsu_cfg = &g_ctsu_cfg_ctsensor_rx130_self;
    if (CTSUSUCCESS == R_CTSU_Open(&hdl_idx, p_ctsu_cfg))
    {
        ctsu_err = R_CTSU_Calibrate(hdl_idx);
        // More actions after calibration
    }
}
```

Special Notes

- The functionality of this API is disabled by default and must be enabled using options available in `r_ctsu_rx_config.h`.
- This API function internally calls `R_CTSU_Scan`.
- This API function waits for scans to complete and blocks program control until calibration succeeds or fails.
- This function will also calculate the MCU-to-MCU correction factor as described in R01AN3610EJ, to do this `CTSUS_CFG_ENABLE_CORRECTION` must be set to 1.
- When calibrating the CTSUS with an external capacitor (12pF) connected to a TS pin, the user must set the `CTSUS_CFG_ENABLE_CORRECTION` preprocessor value to 2. The user must then ensure that the TS pin number is provided as part of the `ctsu_cfg_t` structure.

5.8 R_CTSU_GetVersion

This function returns the driver version number at runtime

Format

```
uint32_t R_CTSU_GetVersion (void)
```

Parameters

None.

Return Values

- CTSU_SUCCESS – Operation completed successfully.
- CTSU_ERR_CORRECTION_FACTOR – Failed to calculate correction factor.

Properties

Prototyped in r_ctsu_rx_if.h.

Description

Returns the version of this module. The version number is encoded such that the top two bytes are the major version number and the bottom two bytes are the minor version number.

Re-entrant

Yes

Example

```
void my_func(void)
{
    uint32_version;
    version = R_CTSU_GetVersion();
}
```

Special Notes

This function is inlined using the #pragma inline directive.

6. Appendices

6.1 Sample CTSU Configuration (Self-Capacitance configuration)

An example configuration for 20 Sensors in total including a button, wheel and slider; below are the sensor drive settings.

```
static ctsu_sensor_setting_t sensor_drive_setting_self01[] = {
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x181)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1A1)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x156)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x185)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1A1)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1B4)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1B2)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x273)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x277)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x232)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x212)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x204)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1F9)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x208)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1A8)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x205)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x22B)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x230)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x287)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
    { .ctsusssc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x282)), .ctsusol
= ((1) << 13) | ((0x03) << 8) | (0x5F), },
};
```

The structure that holds the SFRs.

```
static uint16_t
self_p_sensor_datas_self01[sizeof(sensor_drive_setting_self01)/sizeof(ctsu_sensor_setting_t)*2] = {0};

static ctsu_const_sfrs_t ctsu_const_sfrs_self01 = {
    .CTSUCR0.BYTE = (0x00),
    .CTSUCR1.BYTE = (((1)<<6)|((0)<<4)|(((0))<<3)|((0)<<2)|((1)<<1)|((1)<<0)),
    .CTSUSDPRS.BYTE = (((0))<<6)|(((2))<<4)|((3)<<0)),
    .CTSUSST.BYTE = ((16)),
    .CTSUCHAC0.BYTE = (((1) << 0) | ((1) << 1) | ((1) << 2) | ((1) << 3) |
((1) << 4) | ((1) << 5) | ((1) << 6) | ((1) << 7))),
    .CTSUCHAC1.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0) << 3) |
((0) << 4) | ((0) << 5) | ((0) << 6) | ((1) << 7))),
    .CTSUCHAC2.BYTE = (((1) << 0) | ((1) << 1) | ((1) << 2) | ((1) << 3) |
((1) << 4) | ((1) << 5) | ((1) << 6) | ((0) << 7))),
    .CTSUCHAC3.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((1) << 3) |
((0) << 4) | ((0) << 5) | ((1) << 6) | ((0) << 7))),
    .CTSUCHAC4.BYTE = (((0) << 0) | ((1) << 1) | ((0) << 2) | ((1) << 3)),
    .CTSUDCLKC.BYTE = (((3)<<4)|((0)<<0)),
};
```

An example of the CTSU sensor configuration.

```
ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_self = {

    .p_ctsu_settings = &ctsu_const_sfrs_self01,

    .p_sensor_settings = (ctsu_sensor_setting_t*)sensor_drive_setting_self01,
    .p_sensor_data = (uint16_t*)self_p_sensor_datas_self01,
    .pclk_hz = 32000000,
    .p_callback = 0,
};
```

6.2 Sample CTSU Configuration (Mutual-Capacitance configuration)

An example configuration for 20 Sensors in total in a mutual capacitance configuration; below are the sensor drive settings.

```
static ctsu_sensor_setting_t sensor_setting_mutual0[] = {
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1BE)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1CC)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1CF)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1CE)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1CE)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1E3)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1F5)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1FB)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x1F9)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x201)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x23E)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x256)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x255)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x260)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x268)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x01) << 8), .ctsuso0 = (((3) << 10) | (0x0B8)), .ctsusol
= (((1) << 13) | ((7) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x21D)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x226)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x22A)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
    { .ctsussc = ((0x00) << 8), .ctsuso0 = (((7) << 10) | (0x237)), .ctsusol
= (((1) << 13) | ((3) << 8) | (0x3F)), },
};
```

The structure that holds the SFRs.

```
static uint16_t
mutual01_p_sensor_datas[sizeof(sensor_setting_mutual0)/sizeof(ctsu_sensor_setting_t)*4] = {0};
static ctsu_const_sfrs_t ctsu_const_sfrs_mutual01 = {
    .CTSUCR0.BYTE = (((0)<<7),
    .CTSUCR1.BYTE = (((3)<<6)|((0)<<4)|(((1)<<3)|((0)<<2)|((1)<<1)|((1)<<0))),
    .CTSUSDPRS.BYTE = (((0)<<6)|((2)<<4)|((3)<<0)),
    .CTSUSST.BYTE = ((16)),
    .CTSUCHAC0.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0) << 3) |
    ((0) << 4) | ((1) << 5) | ((1) << 6) | ((1) << 7))),
    .CTSUCHAC1.BYTE = (((1) << 0) | ((1) << 1) | ((0) << 2) | ((0) << 3) |
    ((0) << 4) | ((0) << 5) | ((0) << 6) | ((0) << 7))),
    .CTSUCHAC2.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0) << 3) |
    ((0) << 4) | ((0) << 5) | ((0) << 6) | ((0) << 7))),
    .CTSUCHAC3.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((1) << 3) |
    ((0) << 4) | ((0) << 5) | ((1) << 6) | ((0) << 7))),
    .CTSUCHAC4.BYTE = (((0) << 0) | ((1) << 1) | ((0) << 2) | ((1) << 3))),
    .CTSUCHTRC0.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0)
    << 3) | ((0) << 4) | ((1) << 5) | ((1) << 6) | ((1) << 7))),
    .CTSUCHTRC1.BYTE = (((1) << 0) | ((1) << 1) | ((0) << 2) | ((0)
    << 3) | ((0) << 4) | ((0) << 5) | ((0) << 6) | ((0) << 7))),
    .CTSUCHTRC2.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0)
    << 3) | ((0) << 4) | ((0) << 5) | ((0) << 6) | ((0) << 7))),
    .CTSUCHTRC3.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0)
    << 3) | ((0) << 4) | ((0) << 5) | ((0) << 6) | ((0) << 7))),
    .CTSUCHTRC4.BYTE = (((0) << 0) | ((0) << 1) | ((0) << 2) | ((0)
    << 3))),
    .CTSUDCLKC.BYTE = (((3)<<4)|((0)<<0)),
};
```

An example of the CTSU sensor configuration.

```
ctsu_cfg_t g_ctsu_cfg_ctsensor_rx130_mutual0_all = {
    .p_ctsu_settings = &ctsu_const_sfrs_mutual01,
    .p_sensor_settings = (ctsu_sensor_setting_t*)sensor_setting_mutual0,
    .p_sensor_data = (uint16_t*)mutual01_p_sensor_datas,
    .pclk_hz = 32000000,
    .p_callback = 0,
};
```

Website and Support

Renesas Electronics Website

<http://www.renesas.com/>

Inquiries

<http://www.renesas.com/contact/>

All trademarks and registered trademarks are the property of their respective owners.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	May 9, 2017	-	Initial Version Released

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. 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 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. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

The state of the product is undefined at the moment 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 moment 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 moment 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 moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

Access to reserved addresses is prohibited.

- The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has 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. Moreover, 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.

5. Differences between Products

Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

- The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the 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.

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 disputes 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, drawing, chart, program, algorithm, 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 otherwise misappropriate 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, copy or otherwise misappropriation of Renesas Electronics products.
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; and 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.
Renesas Electronics products are neither intended nor 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 damages (space and 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 third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the 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 radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond 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. Further, Renesas Electronics products are not subject to radiation resistance design. Please ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of 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 by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate 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. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use 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 not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
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 majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.3.0-1 November 2016)



SALES OFFICES

Renesas Electronics Corporation

<http://www.renesas.com>

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

Renesas Electronics America Inc.

2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

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: +44-1628-585-100, Fax: +44-1628-585-900

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, P. R. China 200333
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 II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.

12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141