RL78 Family

TOUCH Module Software Integration System

Introduction

This application note describes the RL78 Family TOUCH Module.

Target Device

RL78/G23 Group

When using this application note with an MCU other than that specified here, adjust the contents to meet the specifications of your target MCU and fully evaluate before using the TOUCH module.

Related Documents

RL78 Family CTSU Module (R11AN0484)

Contents

1. Overview

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

1.1 Functions

The TOUCH module supports the following functions.

1.1.1 QE for Capacitive Touch Usage

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

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

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

1.1.2 Measurements and Data Processing

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

1.1.3 Button Touch Determination

(a) Creating reference value and threshold

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

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

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

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

(b) Positive Noise Filter/Negative Noise Filter

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

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

(c) Hysteresis

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

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

(d) Drift Correction Process

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

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

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

Figure 1 shows an example of the drift correction process.

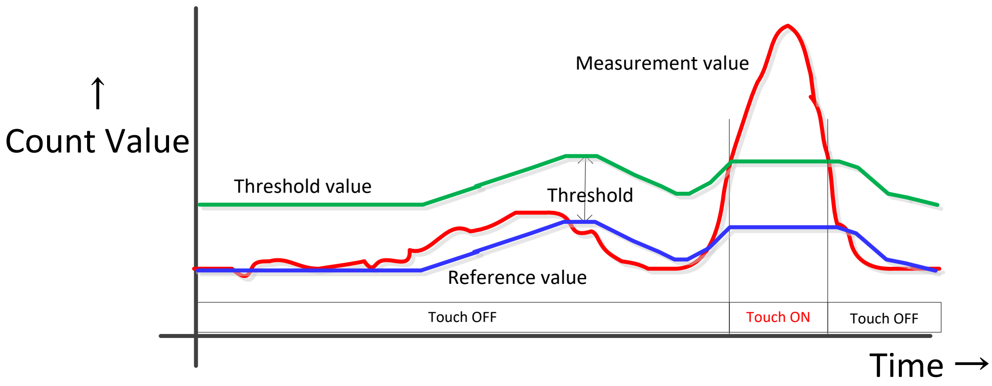

Figure 1Button Touch Determination

(e) Press and hold cancel

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

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

1.1.4 Touch Position Detection of Slider/Wheel

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

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

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

1.2 API Overview

The TOUCH module includes the following functions.

Function	Description
RM_TOUCH_Open()	Initializes the specified touch interface configuration.
RM_TOUCH_StartScan()	Starts measurement of specified touch interface configuration.
RM_TOUCH_DataGet()	Gets measured values of specified touch interface configuration.
RM_TOUCH_CallbackSet()	Sets callback function of specified touch interface configuration.
RM_TOUCH_Close()	Closes specified touch interface configuration.
RM_TOUCH_VersionGet()	Gets module version number.

2. API Information

Operations of this module has been confirmed under the following conditions.

2.1 Hardware Requirements

The MCU used in the development must support the following function:

- CTSU2L

2.2 Software Requirements

This driver depends on the following modules:

- Board support package (r_bsp) v1.00 or newer
- CTSU module (r_ctsu) v1.00 or newer

Also, the driver assumes use of the following tool:

- QE for Capacitive Touch [RA, RL78] V1.3.0 or newer (capacitive touch sensor development support tool

2.3 Supported Toolchains

Module operations have been confirmed on the following toolchains:

- Renesas CC-RL Toolchain v1.09.00
- IAR Embedded Workbench for Renesas RL78 v4.20.1
- LLVM for RL78 10.0.0.202102

2.4 Restrictions

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

2.5 Header File

All interfaces definitions to be called and used in the API are defined in “rm_touch_api.h”.

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

2.6 Integer Type

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

2.7 Compilation Settings

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

rm_touch_config.h Configuration Options	
TOUCH_CFG_PARAM_CHECKING_ENABLE *Default value: "BSP_CFG_PARAM_CHECKING_ENABLE"	Selects whether to include the parameter check process in the code. Selecting "0" allows the user to omit the parameter check process from the code to shorten the code size. "1": Omit parameter check process from code. "2": Include parameter check process in code. "BSP_CFG_PARAM_CHECKING_ENABLE": Selection depends on BSP setting.
TOUCH_CFG_MONITOR_ENABLE This option is not available for rm_touch_config.h. The option is defined in the qe_touch_define.h output by the QE; the default value is "1".	Select 1 to enable data generation for the QE monitor.
TOUCH_CFG_UART_MONITOR_SUPPORT *Default value: "0"	This option is used when TOUCH_CFG_MONITOR_ENABLE is enabled. Set to "1" to enable QE and serial communications. Note: When using the UART module, generate this option with the Smart Configurator.
The following configurations depend on the touch interface configuration and cannot be set using Smart Configurator. These configurations are set when using QE. In this case, QE_TOUCH_CONFIGURATION is defined in the project. Although rm_touch_config.h is invalid, qe_touch_define.h is defined instead.	
CTSU_CFG_NUM_BUTTONS	Sets the total number of buttons.
CTSU_CFG_NUM_SLIDERS	Sets the total number of slides.
CTSU_CFG_NUM_WHEELS	Sets the total number of wheels.

2.8 Code Size

ROM (code and constants) and RAM (global data) size are determined according to the configuration options as described in “section 2.7 Compilation Settings”. Compilation Setting” during a build. The values shown are reference values when the compile option is the default for the CC-RL C compiler listed in “section 2.3 Supported Toolchains”. The code size varies according to the C compile version and compile options.

This is the value when one self-capacity button is set in the default setting of Smart Configurator. It also includes sample applications generated by the CTSU module and QE for Capacitive Touch.

Table 1 Code Size

ROM and RAM Usage	
RAM	379 bytes
ROM	7268 bytes

2.9 Arguments

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

These structures and enums are defined in rm_touch_api.h along with the prototype declaration.

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

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

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

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

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

```
} touch_cfg_t;
```

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

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

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

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

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

2.10 Return Values

The following provides return values for the API functions. The enum is defined in r_ctsu_api.h, along with the API function prototype declaration.

```
/* Return error codes */
typedef enum e_fsp_err
{
    FSP_SUCCESS,
    FSP_ERR_ASSERTION,        ///< A critical assertion has failed
    FSP_ERR_INVALID_ARGUMENT,  ///< Invalid input parameter
    FSP_ERR_NOT_OPEN,         ///< Requested channel is not configured or API not open
    FSP_ERR_ALREADY_OPEN,     ///< Requested channel is already open in a different configuration
    FSP_ERR_NOT_ENABLED,      ///< Requested operation is not enabled
    FSP_ERR_INVALID_STATE,    ///< API or command not valid in the current state
    FSP_ERR_CTSU_SCANNING,        ///< Scanning.
    FSP_ERR_CTSU_NOT_GET_DATA,     ///< Not processed previous scan data.
    FSP_ERR_CTSU_INCOMPLETE_TUNING, ///< Incomplete initial offset tuning.
} fsp_err_t;
```

3. API Functions

3.1 RM_TOUCH_Open

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

Format

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

Parameters

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

Return Values

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

Properties

Prototype is declared in r_touch_api.h.

Description

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

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

Reentrant

This function is non-entrant.

Example

```
fsp_err_t err;

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

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

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

Special Notes:

The port must be initialized before calling this function. We recommend using the R_CTSU_PinSetInit() function generated by SmartConfigurator as the port initialization function.

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

3.2 RM_TOUCH_ScanStart

This function starts measurement of the specified touch interface configuration.

Format

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

Parameters

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

Return Values

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

Properties

Prototype is declared in r_touch_api.h.

Description

This function calls R_CTSU_ScanStart() and starts the measurement.

Reentrant

This function is non-reentrant.

Example

```
fsp_err_t err;

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

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

Special Notes:

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

3.3 RM_TOUCH_DataGet

This function reads the specified touch interface configuration.

Format

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

Parameters

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

Return Values

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

Properties

Prototype is declared in r_touch_api.h.

Description

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

Reentrant

This function is non-reentrant.

Example:

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

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

Special Notes:

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

3.4 RM_TOUCH_CallbackSet

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

Format

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

Parameters

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

p_callback Pointer to callback function

p_context Pointer to send to callback function

p_callback_memory Set to NULL

Return Values

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

Properties

Prototype is declared in rm_touch_api.h.

Description

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

Reentrant

This function is non-reentrant.

Example:

```
fsp_err_t err;

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

Special Notes:

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

3.5 RM_TOUCH_Close

This function closes the specified touch interface configuration.

Format

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

Parameters

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

Return Values

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

Properties

Prototype is declared in r_touch_api.h.

Description

This function closes the specified touch interface configuration.

Reentrant

This function is non-reentrant.

Example:

```
fsp_err_t err;

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

Special Notes:

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

3.6 RM_TOUCH_VersionGet

This function returns the version information of this module.

Format

```
fsp_err_t RM_TOUCH_VersionGet (fsp_version_t * const p_version)
```

Parameters

p_version Pointer that stores the obtained version information.

Return Values

FSP_SUCCESS */* Successfully completed */*

FSP_ERR_ASSERTION */* Argument pointer not specified */*

Properties

Prototype is declared in r_touch_api.h.

Description

This function returns the version information of this module, which is encoded as follows: the upper 2 bytes contain the API version number and the lower two bytes contain the code version number. For example, when the CODE version number is 1.10 and the CODE version number is 1.31, the version information is encoded as 0x010a011f.

Reentrant

This function is reentrant.

Example:

```
fsp_err_t err;
fsp_version_t fsp_version;

/* Get version of installed CTSU API. */
err = RM_TOUCH_GetVersion(&fsp_version);

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

Special Notes:

None

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Apr.13.21	-	First edition issued

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

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

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

Corporate Headquarters

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

Trademarks

Contact information

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