# RZ/A2M Group

## RZ/A2M SCIFA Driver

R11AN0307EG0100
Rev.1.0
Sept 25, 2018

## Introduction

This application note describes the operation of the software SCIFA Driver for the RZ/A2 device on the RZ/A2M CPU Board.

It provides a comprehensive overview of the Driver. For further details please refer to the software driver itself.

The user is assumed to have knowledge of e$^2$ studio and to be equipped with an RZ/A2M CPU Board.

## Target Device

RZ/A2M Group

## Driver Dependencies

This driver depends on:

- Middleware:
    - Renesas OS Abstraction (FreeRTOS, RTX or OSless version).
- Drivers
    - STDIO
    - INTC Driver
    - CPG Driver
    - GPIO Driver

## Referenced Documents

| Document Type | Document Name | Document No. |
|---|---|---|
| User's Manual | RZ/A2M Hardware Manual | R01UH0746EJ |

## List of Abbreviations and Acronyms

| Abbreviation | Full Form |
|---|---|
| ANSI | American National Standards Institute |
| API | Application Programming Interface |
| ARM | Advanced RISC Machine |
| CPG | Clock Pulse Generator |
| CPU | Central Processing Unit |
| FIFO | First In First Out |
| GPIO | General Purpose Input/Output |
| HLD | High Layer Driver |
| IDE | Integrated Development Environment |
| INTC | INTerrupt Controller |
| LLD | Low Layer Driver |
| MCU | Microcontroller Unit |
| MODEM | MOdulate DEModulate |
| OS | Operating System |
| RISC | Reduced Instruction Set Computer |
| RTX | Short for CMSIS-RTOS Keil RTX real-time operating system |
| RX | Receive |
| RXI | Receive FIFO data full Interrupt |
| SCIFA | Serial Communications Interface with FIFO |
| STDIO | Standard Input/Output |
| TX | Transmit |
| TXI | Transmit data empty Interrupt |

**Table 1-1** List of Abbreviations and Acronyms

## Contents

# 1. Outline of SCIFA Driver

The MCU provides the 'Serial Communications Interface with FIFO (SCIFA)' peripheral. The peripheral has five channels that support both asynchronous and clock synchronous serial communication. The SCIFA makes use of a 16-stage FIFO buffers for both transmission and reception to allow for efficient high speed continuous communication.

For further information regarding the hardware specifics of the SCIFA peripheral please refer to the appropriate hardware manual.

# 2. Description of the Software Driver

The key features of the driver include selectable:

- Channels
- Baud rates
- Data bits
- Stop bits
- Parity
- Transmission mode: asynchronous or clock synchronous
- Data order
- Loopback mode

The extended features include:

- Configurable transmit FIFO trigger
- Configurable receive FIFO trigger
- MODEM mode
- Configure external clocks
- Noise cancellation

## 2.1 Structure

The SCIFA driver is split into two parts: the High Layer Driver (HLD) and the Low Layer Driver (LLD). The HLD includes platform independent features of the driver, implemented via the STDIO Standard functions. The LLD includes all the hardware specific functions.

```
r_scifa
    doc
        r11an0307eg0100-rza2m-scifa-driver.pdf
    inc
        r_rza2_scifa_lld_cfg.h
        r_rza2_scifa_lld.h
        r_scifa_device_cfg.h
        r_scifa_drv_api.h
        r_scifa_drv_link.h
        r_scifa_drv_sc_cfg.h
        r_scifa_hld_prv.h
        r_scifa_lld_rza2m_api.h
    src
        hld
            r_scifa_drv_api.c
            r_scifa_hld_prv.c
        lld
            r_rza2_scifa_lld_baud.c
            r_rza2_scifa_lld.c
```

## 2.2 Description of each file

Each file's description can be seen in the following table.

| Filename | Usage | Description |
|---|---|---|
| Application-Facing Driver API | | |
| r_scifa_drv_api.h | Application | The only API header file to include in application code. |
| High Layer Driver (HLD) Source | | |
| r_scifa_hld_prv.h | Private (HLD only) | Private header file intended ONLY for use in High Layer Driver (HLD) source. NOT for application or Low Layer Driver (LLD) use. |
| r_scifa_drv_api.c | Private (HLD only) | High Layer Driver (HLD) source code enabling the driver API functions. |
| r_scifa_hld_prv.c | Private (HLD only) | High Layer Driver (HLD) private source code enabling the functionality of the driver, abstracted from the low-level access. |
| High Layer to Low Layer API | | |
| r_xxxx_scifa_lld.h | Private (HLD/LLD only) | Low Layer Driver (LLD) header file (where "xxxx" is a device and board-specific identification). Intended ONLY to provide access for High Layer Driver (HLD) to required Low Layer Driver functions (LLD). Not for use in application, not to define any device specific enumerations or structures. |
| r_scifa_lld_xxxx_api.h | Private (HLD/LLD only) | Low Layer Driver (LLD) header file (where "xxxx" is a device and board-specific identification). Intended for definitions of device specific settings (in the form of enumerations and structures). |
| r_xxxx_scifa_lld_cfg.h | Private (HLD/LLD only) | Low Layer Driver (LLD) header file (where "xxxx" is a device and board-specific identification). Intended for definitions of device specific settings (in the form of enumerations and structures). No LLD functions to be defined in this file. |
| Abstraction Link between High and Low Layer Drivers (HLD/LLD Link) | | |
| r_scifa_drv_link.h | Private (HLD/LLD only) | Header file intended as an abstraction between low and high layer. This header will include the device specific config file "r_xxxx_scifa_lld.h". |
| r_scifa_device_cfg.h | Should be included in "r_scifa_drv_api.h" | Header file intended as an abstraction between low and high layer. This header will include the device specific config file "r_xxxx_scifa_lld_cfg.h". |
| Low Layer Driver (LLD) Source | | |
| r_xxxx_scifa_lld.c | Private (LLD only) | (Where "xxxx" is a device and board specific identification). Provides the definitions for the Low Layer Driver interface. |
| r_xxxx_scifa_lld_baud.c | Private (LLD only) | Low Layer Driver function definitions for setting baud rate. |
| Smart Configurator | | |
| r_scifa_drv_sc_cfg.h | Private (HLD/LLD only) | This file is intended to be used by Smart Configurator to pass setup information to the driver. This is not for application use. |

## 2.3 High Layer Driver

The High Layer Driver can be either used through STDIO or through direct access. It is recommended not to mix both access methods.

The driver layer functions can be seen in the below table:

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| **int_t** | **scifa_hld_open(** st_stream_ptr_t p_stream**)** | Driver initialisation interface is mapped to open function called directly using the st_r_driver_t SCIFA driver handle g_scifa_driver: i.e. **g_scifa_driver.open()** | [in] **p_stream** driver handle. | **DRV_SUCCESS** Open Success<br>**DRV_ERROR** Open Error |
| **void** | **scifa_hld_close(** st_stream_ptr_t p_stream**)** | Driver close interface is mapped to close function called directly using the st_r_driver_t SCIFA driver structure g_scifa_driver: i.e. **g_scifa_driver.close()** | [in] **p_stream** driver handle. | **None** |
| **int_t** | **scifa_hld_read(** st_stream_ptr_t p_stream, uint8_t *p_buffer, uint32_t count**)** | Driver close interface is mapped to read function called directly using the st_r_driver_t SCIFA driver structure g_scifa_driver: i.e. **g_scifa_driver.read()** | [in] **p_stream** driver handle.<br>[out] **p_buffer** buffer for returned data.<br>[in] **count** size of buffer. | **Amount of Data Read**<br>**DRV_ERROR** Write Error |
| **int_t** | **scifa_hld_write(** st_stream_ptr_t p_stream, uint8_t *p_buffer, uint32_t count**)** | Driver write interface is mapped to write function called directly using the st_r_driver_t SCIFA driver structure g_scifa_driver: i.e. **g_scifa_driver.write()** | [in] **p_stream** driver handle.<br>[in] **p_buffer** data to send.<br>[in] **count** size of data to send. | **DRV_SUCCESS** Write Success<br>**DRV_ERROR** Write Error |
| **int_t** | **scifa_hld_control(** st_stream_ptr_t p_stream**,** uint32_t ctl_code**,** void* p_ctl_struct**)** | Driver control interface function.<br>Maps to ANSI library low level control function.<br>Called directly using the st_r_driver_t SCIFA driver structure g_scifa_driver: i.e. **g_scifa_driver.control()** | [in] **p_stream** driver handle.<br>[in] **ctl_code** The type of control function to use.<br>[in/out] **p_ctl_struct** required parameter is dependent upon the control function. | **DRV_SUCCESS** Operation Success<br>**DRV_ERROR** Operation Error |

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| **int_t** | **scifa_get_version(** st_stream_ptr_t p_stream**,** st_ver_info_ptr_t p_ver_info**)** | Driver get_version interface function<br>Maps to extended non-ANSI library low level get_version function.<br>Called directly using the st_r_driver_t SCIFA driver structure g_scifa_driver: i.e.<br>**g_scifa_driver.get_version()** | [in] **p_stream** handle to the (pre-opened) channel.<br>[out] **p_ver_info** handle to the (pre-opened) channel. | **DRV_SUCCESS** Operation Success |

These High layer functions can be accessed either executed directly or through STDIO.

## 2.4 Low Level Driver

The Low Layer Driver provides the functions to configure the hardware.

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| int_t | **R_SCIFA_Init(**<br>int_t channel<br>**)** | Initialises the channel | **channel** [in] channel to configure | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_Close(**<br>int_t channel<br>**)** | Close the SCIFA driver for channel | **channel** [in] channel to close | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_Start(**<br>int_t channel,<br>int_t read_write<br>**)** | Starts channel operation | **channel** [in] channel to start<br>**read_write** [in] permission level of channel | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_Stop(**<br>int_t channel<br>**)** | Stops channel operation | **channel** [in] channel to stop | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_PutByte(**<br>int_t channel,<br>uint8_t data<br>**)** | Sends 1 byte of data | **channel** [in] channel to send from<br>**data** [in] byte to send | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_GetByte(**<br>int_t channel,<br>uint8_t *p_data<br>**)** | Receives 1 byte of data | **channel** [in] channel to receive on<br>**p_data** [out] pointer to store received byte | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_GetTXIState (**<br>int_t channel<br>**)** | Determines the current status of TXI interrupts for channel | **channel** [in] channel to check interrupt status for | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_ClearTXIFlags(**<br>int_t channel<br>**)** | Clears flags for TXI interrupt of a given channel | **channel** [in] clears flags for TXI of a given channel | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_StartTXI(**<br>int_t channel<br>**)** | Starts the TXI interrupt for a given channel | **channel** [in] channel to start TXI for | **DRV_SUCCESS** on success **DRV_ERROR** on failure |
| int_t | **R_SCIFA_StopTXI(**<br>int_t channel<br>**)** | Stops the TXI interrupt for a given channel | **channel** [in] channel to stop TXI for | **DRV_SUCCESS** on success **DRV_ERROR** on failure |

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| int_t | **R_SCIFA_IsTxFifoFull (**<br><br>int_t channel,<br>int_t *p_fifo_full<br>**)** | Check for space in the TX FIFO | **channel** [in] channel to stop TXI for<br><br>**p_fifo_full** [out] 0 if there's space in the FIFO, 1 if it's full | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| int_t | **R_SCIFA_GetRXIState (**<br><br>int_t channel<br>**)** | Determines the current status of RXI interrupts for channel | **channel** [in] channel to check RXI status for | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| int_t | **R_SCIFA_ClearRXIFlags(**<br><br>int_t channel<br>**)** | Clears flags for RXI interrupt of a given channel | **channel** [in] clears flags for RXI of a given channel | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| int_t | **R_SCIFA_CheckRXIError(**<br><br>int_t channel,<br>st_scifa_rx_error_t *p_rx_err<br>**)** | Populates the RX error structure with any error information and handles errors appropriately | **channel** [in] clears flags for RXI of a given channel<br><br>**p_rx_err** [out] structure of error information for RX of SCIFA | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| int_t | **R_SCIFA_IsRxFifoEmpty(**<br><br>int_t channel,<br>int_t *p_fifo_empty<br>**)** | Check for data in the RX FIFO | **channel** [in] channel to stop RXI for<br><br>**p_fifo_empty** [out] 0 if there's data in the FIFO, 1 if it's empty | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| int_t | **R_SICFA_SetTransmissionMode(**<br><br>int_t channel,<br>e_scifa_mode_t desired_mode<br>**)** | Function to change the transmission mode of a given SCIFA channel | **channel** [in] channel to set the transmission mode for<br><br>**desired_mode** [in] desired transmission mode | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| **int_t** | **R_SCIFA_SetBaud(**<br>int_t channel,<br>uint32_t desired_baud,<br>uint32_t clock_freq,<br>e_scifa_mode_t mode,<br>uint32_t<br>* p_achieved_baud_rat<br>e<br>**)** | Function to determine whether the desired baud rate is possible, and if so, set the baud rate of that SCIFA channel | **channel** [in] channel to set the baud rate for<br>**desired_baud** [in] the baud rate desired.<br>**clock_freq** [in] the current clock frequency<br>**mode** [in] the current transmission mode<br>**p_achieved_baud_rate** [out] value of achieved baud rate | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| **int_t** | **R_SCIFA_SetDataBits(**<br>int_t channel,<br>e_scifa_data_bits_t desired_data_bits<br>**)** | Function to change the data bits of a given SCIFA channel | **channel** [in] channel to set the number of data bits for<br>**desired_data_bits** [in] desired number of data bits | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| **int_t** | **R_SCIFA_SetStopBits(**<br>int_t channel,<br>e_scifa_stop_bits_t desired_stop_bits<br>**)** | Function to change the stop bit(s) of a given SCIFA channel | **channel** [in] channel to set the number of stop bit(s) for<br>**desired_stop_bits** [in] desired number of stop bit(s) | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| **int_t** | **R_SCIFA_SetParity(**<br>int_t channel,<br>e_scifa_parity_t desired_parity<br>**)** | Function to change the parity setting for a given SCIFA channel | **channel** [in] channel to set the parity for<br>**desired_parity** [in] the parity desired | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| **int_t** | **R_SCIFA_SetDataOrder(**<br>int_t channel,<br>e_scifa_data_order_t desired_data_order<br>**)** | Function to change the data bit order of a given SCIFA channel | **channel** [in] channel to set the data order for<br>**desired_data_order** [in] desired data order | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |

| Return Type | Function | Description | Arguments | Return |
|---|---|---|---|---|
| **int_t** | **R_SCIFA_SetLoopBack(**<br><br>int_t channel,<br>e_scifa_loopback_t loopback<br>**)** | Function to configure loopback mode for a given channel | **channel** [in]<br>channel to set the loopback for<br><br>**loopback** [in]<br>loopback setting | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |
| **int_t** | **R_SCIFA_SetExtendedCfg(**<br><br>int_t channel,<br>const st_scifa_extended_t *p_ext_cfg<br>**)** | Function to configure the extended SCIFA peripheral settings | **channel** [in]<br>channel to set the extended settings for<br><br>**p_ext_cfg** [in]<br>extended config settings | **DRV_SUCCESS** on success<br>**DRV_ERROR** on failure |

# 3. Accessing the High Layer Driver

## 3.1 STDIO

The HLD's API can be accessed through the ANSI 'C' Library <stdio.h>. The following table details the operation of each function:

| Operation | Return | Function Details |
|---|---|---|
| open | gs_stdio_handle, unique handle to driver | open (DEVICE_IDENTIFIER "scifa0", O_RDWR); |
| close | DRV_SUCCESS successful operation, or driver specific error | close (gs_stdio_handle); |
| read | Number of characters read, -1 on error | read (gs_stdio_handle, buff, data_length); |
| write | Number of characters written, -1 on error | write (gs_stdio_handle, buff, data_length); |
| control | DRV_SUCCESS control was process, or driver specific error | control (gs_stdio_handle, CTRL, &struct); |
| get_version | DRV_SUCCESS drv_info was updated, or DRV_ERROR drv_info was not updated | get_version (DEVICE_IDENTIFIER "scifa0", &drv_info); |

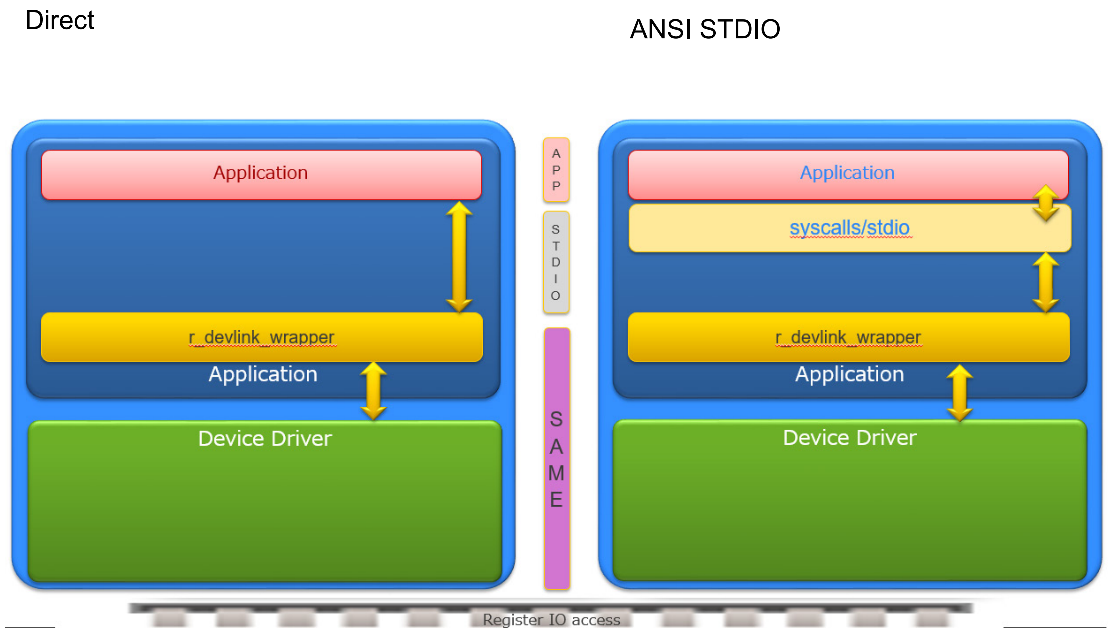
### 3.2 Direct

The following table shows the available direct functions.

| Operation | Return | Function details |
|---|---|---|
| open | gs_direct_handle unique handle to driver | direct_open ("scifa0", 0); |
| close | DRV_SUCCESS successful operation, or driver specific error | direct_close (gs_direct_handle); |
| read | Number of characters read, -1 on error | direct_read (gs_direct_handle, buff, data_length); |
| write | Number of characters written, -1 on error | direct_write (gs_direct_handle, buff, data_length); |
| control | DRV_SUCCESS control was process, or driver specific error | direct_control (gs_direct_handle, CTRL, &struct); |
| get_version | DRV_SUCCESS drv_info was updated, or DRV_ERROR drv_info was not updated | direct_get_version ("scifa0", &drv_info); |

### 3.3 Comparison

The below diagram illustrates the difference between the Direct and ANSI STDIO methods.



# 4. Example of Use

This section describes a simple example of opening the driver, configuring the driver, transmitting and receiving data and closing a driver.

## 4.1 Open

```
int_t gs_scifa_handle;
uint8_t ch0_drv_name[] = "scifa0";

gs_scifa_handle = open(ch0_drv_name, O_RDWR);
```

## 4.2 Control – Set Configuration Settings

```
st_r_drv_scifa_config_t set_cfg;

set_cfg.baud = 115200u;
set_cfg.data_bits = SCIFA_DATA_BITS_EIGHT;
set_cfg.mode = SCIFA_TRANSMISSION_ASYNC;
set_cfg.parity = SCIFA_PARITY_NONE;
set_cfg.stop_bits = SCIFA_STOP_BITS_ONE;
set_cfg.extended_cfg.clk_enable = SCIFA_CLK_ENABLE_MODE_0;
set_cfg.extended_cfg.noise_cancel = SCIFA_NOISE_CANCEL_DISABLE;
set_cfg.extended_cfg.modem = SCIFA_MODEM_DISABLE;
set_cfg.extended_cfg.fifo_tx_trg_bytes = 8u;
set_cfg.extended_cfg.fifo_rx_trg_bytes = 8u;

result = control(gs_scifa_handle, CTL_SCIFA_SET_CONFIGURATION, &set_cfg);
```

## 4.3 Control – Get Configuration Settings

```
st_r_drv_scifa_config_t get_cfg;

result = control(gs_scifa_handle, CTL_SCIFA_GET_CONFIGURATION, &get_cfg);
```

## 4.4 Write

```
uint8_t data[] = "Data to send\r\n";

result = write(gs_scifa_handle, data, sizeof(data));
```

## 4.5 Read

```
uint8_t data[100];

result = read(gs_scifa_handle, data, sizeof(data));
```

## 4.6 Close

```
close(gs_scifa_handle);
```

## 4.7 Get Version

```
st_ver_info_t info;
result = get_version(gs_scifa_handle, &info);
```

# 5. OS Support

This driver supports any OS through using the OS Abstraction module. For more details about the abstraction module please refer to the OS abstraction module application note.

# 6. How to Import the Driver

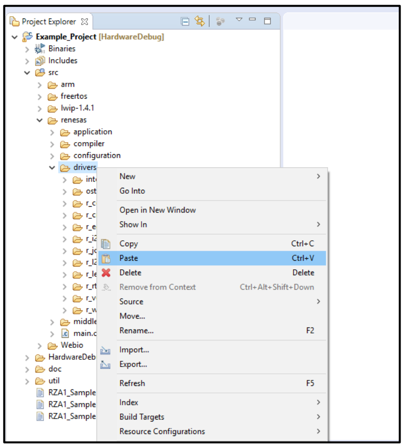
This section describes how to import the driver into your project. Generally, there are two steps in any IDE:

1) Copy the r_scifa driver to the location in the source tree that you require for your project.
2) Add the link to where you copied your driver to the compiler.

## 6.1 e$^2$ studio

To import the driver into your project please follow the instructions below.

1) In Windows Explorer, right-click on the r_scifa folder, and click **Copy**.
2) In e$^2$ studio Project Explorer view, select the folder where you wish the driver project to be located; right-click and click **Paste**.
3) Right-click on the parent project folder (in this case 'Example_Project') and click **Properties ...**
4) In 'C/C++ Build → Settings → Cross ARM Compiler → Includes', add the include folder of the newly added driver, e.g. '${ProjDirPath}\src\renesas\sc_drivers\r_scifa\inc'



## Website and Support

Renesas Electronics website
https://www.renesas.com/

Inquiries
https://www.renesas.com/contact/

## Revision History

| Rev. | Date | Description | |
|---|---|---|---|
| | | Page | Summary |
| 1.00 | Sept 25, 2018 | All | Created document. |
| | | | |

## 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.

   ¾ 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.

   ¾ 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.

   ¾ 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.

   ¾ 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.

   ¾ 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.

RENESAS

**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