RZ/A2M Group

RZ/A2M RIIC Driver

R01AN4645EG
Rev.1.0
Jan 03, 2019

Introduction

This application note describes the operation of the software RIIC 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
ACK	ACKnowledge
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
I^2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
INTC	INTerrupt Controller
LLD	Low Layer Driver
MCU	Microcontroller Unit
MHz	MegaHertz
MODEM	MOdulate DEModulate
NACK	Not ACKnowledged
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
RIIC	Renesas Inter-Integrated Circuit
SCL	Serial Clock
STDIO	Standard Input/Output
TX	Transmit
TXI	Transmit data empty Interrupt

Table 1-1 List of Abbreviations and Acronyms

Contents

1. Outline of RIIC Driver

The MCU provides the 'Renesas Inter-Integrated Circuit (RIIC)' peripheral. The peripheral has four channels that support I^2C communication at a clock frequency up to 1MHz.

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

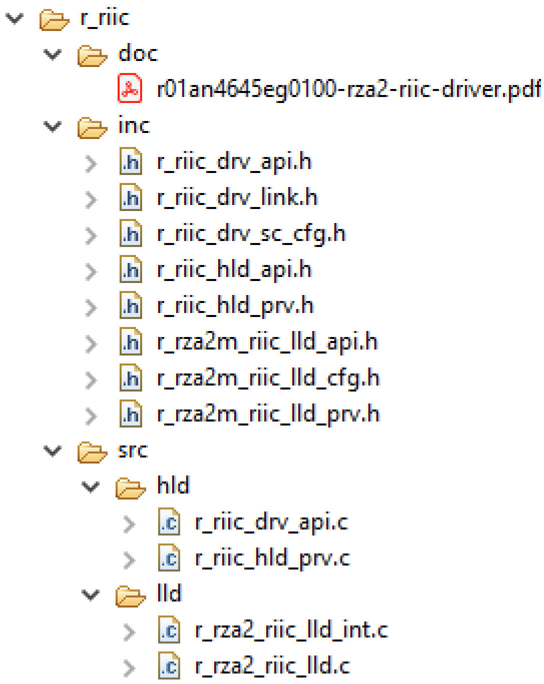
2. Description of the Software Driver

The key features of the driver include selectable:

- Channels
- Clock Frequency
- Clock Duty Cycle
- Noise Filter
- Slave Device Addressing
- Timeout monitoring

2.1 Structure

The RIIC driver is split into two parts: High Layer Driver (HLD) and 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.



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_riic_drv_api.h	Application	The only API header file to include in application code.
High Layer Driver (HLD) Source		
r_riic_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_riic_drv_api.c	Private (HLD only)	High Layer Driver (HLD) source code enabling the driver API functions.
r_riic_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_riic_hld_api.h	Private (HLD/LLD only)	High Layer Driver (HLD) header file intended to interface to the Low Layer Driver (LLD) to provide callback functions for various interrupt events. Not for use in application
r_xxxx_riic_lld_api.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 or directly in High Layer Driver (HLD). For the HLD it should be included indirectly in the file r_riic_drv_link.h only to provide abstraction for the HLD.
Abstraction Link between High and Low Layer Drivers (HLD/LLD Link)		
r_riic_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_riic_lld_api.h".
Low Layer Driver (LLD) Source		
r_xxxx_riic_lld.c	Private (LLD only)	(Where "xxxx" is a device and board specific identification). Provides the source code for the Low Layer Driver interface.
r_xxxx_riic_lld_int.c	Private (LLD only)	(Where "xxxx" is a device and board specific identification). Source code for the device interrupt handling code.
r_xxxx_riic_lld_cfg.h	Private (LLD only)	Provides device specific information for the Low Layer Driver only. NOT for application or High Layer Driver (HLD) use.
r_xxxx_riic_lld_prv.h	Private (LLD only)	Private header file intended ONLY for use in Low Layer Driver (LLD) source. NOT for application or High Layer Driver (HLD) use.
Smart Configurator		
r_riic_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	**RIIC_hld_open(** st_stream_ptr_t p_stream**)**	Driver initialisation interface is mapped to open function called directly using the st_r_driver_t RIIC driver handle g_RIIC_driver: i.e. **g_RIIC_driver.open()**	[in] **p_stream** driver handle.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
void	**RIIC_hld_close(** st_stream_ptr_t p_stream**)**	Driver close interface is mapped to close function called directly using the st_r_driver_t RIIC driver structure g_RIIC_driver: i.e. **g_RIIC_driver.close()**	[in] **p_stream** driver handle.	**None**
int_t	**RIIC_hld_control(** st_stream_ptr_t p_stream**,** uint32_t ctl_code**,** void* p_ctl_struct**)**	Driver control interface function. Maps to ANSI library low level control function. Called directly using the st_r_driver_t RIIC driver structure g_RIIC_driver: i.e. **g_RIIC_driver.control()**	[in] **p_stream** driver handle. [in] **ctl_code** The type of control function to use. [in/out] **p_ctl_struct** required parameter is dependent upon the control function.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
int_t	**RIIC_get_version(** st_stream_ptr_t p_stream**,** st_ver_info_ptr_t p_ver_info **)**	Driver get_version interface function Maps to extended non-ANSI library low level get_version function. Called directly using the st_r_driver_t RIIC driver structure g_RIIC_driver: i.e. **g_RIIC_driver.get_version()**	[in] **p_stream** handle to the (pre-opened) channel. [out] **p_ver_info** handle to the driver (pre-opened) channel.	**DRV_SUCCESS** on success Does not return any other values

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

2.3.1 Available Control Commands

The control functionality of the RIIC driver is defined in the enumeration e_ctrl_code_riic_t and supports the commands listed below. They can be accessed via the STDIO control function in the following way:

result = control(driver_handle, Control Command, Pointer to appropriate structure);

The command is defined in the enumeration e_ctrl_code_riic_t

See section 4 for examples of usage.

Control Command	Description	Arguments	Return
CTL_RIIC_SET_CONFIG	Set Driver configuration according to the values in the st_riic_config_t structure passed as a parameter to the control call.	[in] **st_riic_config_t * riic_config_ptr** Pointer to config structure containing required configuration.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
CTL_RIIC_GET_CONFIG	Place the current Driver settings into the st_riic_config_t structure passed as a parameter to the control call.	[out] **st_riic_config_t * riic_config_ptr** Pointer to the config structure into which the current settings are to be placed.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
CTL_RIIC_READ	Read data from I^2C slave device on selected channel, using the details defined in the st_r_drv_riic_transfer_t parameter provided to the control call.	[in] **st_r_drv_riic_transfer_t * riic_transfer_ptr** Pointer to configuration structure for the read operation.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
CTL_RIIC_WRITE	Write data to I^2C slave device on selected channel, using the details defined in the st_r_drv_riic_transfer_t parameter provided to the control call.	[in] **st_r_drv_riic_transfer_t * riic_transfer_ptr** Pointer to configuration structure for the write operation.	**DRV_SUCCESS** on success **DRV_ERROR** on failure

2.4 Low Layer Driver

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

Return Values	Function	Description	Arguments	Return Values
int_t	**R_RIIC_ScInitChannel (int_t channel, uint32_t pclk_frequency_hz)**	Initializes RIIC using SmartConfigurator parameters.	[in] **int_t channel** RIIC channel to initialise. [in] **uint32_t pclk_frequency_hz**	**DRV_SUCCESS** on success **DRV_ERROR** on failure
int_t	**R_RIIC_SetConfig (int_t channel, st_r_drv_riic_config_t*p_cfg)**	Change configuration of the selected RIIC channel.	[in] **int_t channel** RIIC channel to set configuration. [in] **st_r_drv_riic_config_t *p_cfg** pointer to RIIC configuration parameter.	**DRV_SUCCESS** on success **DRV_ERROR** on failure
int_t	**R_RIIC_GetConfig (int_t channel, st_r_drv_riic_config_t*p_cfg)**	Get configuration of the selected RIIC channel into holding variable.	[in] **int_t channel** RIIC channel to get configuration. [in] **st_r_drv_riic_config_t *p_cfg** pointer to st_riic_config_t structure to hold configuration information	**DRV_SUCCESS** on success **DRV_ERROR** on failure
void	**R_RIIC_CloseChannel (int_t channel)**	Close the selected RIIC channel peripheral into lowest power configuration.	[in] **int_t channel** RIIC channel to close.	void
void	**R_RIIC_TransmitStop (int_t channel)**	Issue a STOP condition on the selected RIIC channel.	[in] **int_t channel** RIIC channel to issue the STOP condition on.	void
void	**R_RIIC_ClearStop(int_t channel)**	Clear STOP condition on the selected RIIC channel.	[in] **int_t channel** RIIC channel to release STOP condition on.	void

void	**R_RIIC_TransmitStart (int_t channel)**	Issue a START condition on the selected RIIC channel.	[in] **int_t channel** RIIC channel to issue the START condition on.	void
void	**R_RIIC_TransmitRestart (int_t channel)**	Issue a RESTART condition on the selected RIIC channel.	[in] **int_t channel** RIIC channel to issue the RESTART condition on.	void
void	**R_RIIC_ClearNack (int_t channel)**	Clear NACK detection flag on the selected RIIC channel.	[in] **int_t channel** RIIC channel to issue the RESTART condition on.	void
uint8_t	**R_RIIC_GetAckStatus (int_t channel)**	Return the ACK status of the selected RIIC channel. (Should be called only within callback functions because the NACK receive interrupt is a level interrupt and is cleared by LLD at the end of the interrupt handler).	[in] **int_t channel** RIIC channel.	0 : ACK received 1 : ACK not received
void	**R_RIIC_TransmitAck (int_t channel)**	Issue an ACK on the selected RIIC channel.	[in] **int_t channel** RIIC channel to issue the ACK on.	void
void	**R_RIIC_TransmitNack (int_t channel)**	Issue an NACK on the selected RIIC channel. Should only be called in the last byte-1 receive data full interrupt handler callback function.	[in] **int_t channel** RIIC channel to issue the NACK on.	void
void	**R_RIIC_AssertLowHold (int_t channel)**	Assert a wait period by holding low during the period between the 9th and 1st clock cycles on the selected RIIC channel. Should only with the last byte - 2 receive data full interrupt handler callback function.	[in] **int_t channel** RIIC channel to assert the wait period on.	void

void	**R_RIIC_ReleaseLowHold (int_t channel)**	Release a wait period holding low during the period between the 9th and 1st clock cycles on the selected RIIC channel.	[in] **int_t channel** RIIC channel to release the wait period on.	void
void	**R_RIIC_WriteByte (int_t channel, uint8_t byte)**	Write a byte on the selected RIIC channel. Should be used after confirming that the transmit data empty interrupt of the previous data transmission has occurred	[in] **int_t channel** RIIC channel to write on. (in) uint8_t byte byte value to write.	void
uint8_t	**R_RIIC_ReadByte (int_t channel)**	Reads a byte on the selected RIIC channel. Should be called after confirming that the receive data full interrupt of the previous data reception has occurred	[in] **int_t channel** RIIC channel to read byte from.	value read from channel
uint8_t	**R_RIIC_IsBusBusy (int_t channel)**	Determines if bus is busy.	[in] **int_t channel** RIIC channel to read byte from.	0 : Bus is free 1 : Bus is busy
uint8_t	**R_RIIC_IsStopAsserted (int_t channel)**	Determine if a STOP condition has been asserted on the selected RIIC channel. (Note that the stop condition will be cleared when a stop interrupt occurs. Appropriate case must be taken when using this function to ensure that it is valid).	[in] **int_t channel** RIIC channel to read byte from.	0 : STOP condition not detected. 1 : STOP condition detected.
void	**R_RIIC_DetectTimeoutStart(int_t channel)**	Start counting SCL-stop timeout counter, and enable SCL timeout interrupt.	[in] **int_t channel** RIIC channel.	void

void	**R_RIIC_DetectTimeoutStop(int_t channel)**	Stops counting SCL-stop timeout counter and disable SCL timeout interrupt. Note that even if this function is called, the counter will not be cleared. The counter is cleared when the level of the SCL line changes (high/low) or when RIIC HW is reset	[in] **int_t channel** RIIC channel.	void
void	**R_RIIC_DetectArbitrationStart(int_t channel)**	Start arbitration failure detection and enable arbitration failed interrupt	[in] **int_t channel** RIIC channel to start detection of arbitration interrupt.	void
void	**R_RIIC_DetectArbitrationStop(int_t channel)**	Stop arbitration failure detection and ensure arbitration failed interrupt is disabled	[in] **int_t channel** RIIC channel to stop detection of arbitration interrupt.	void
Int_t	**R_RIIC_WaitSlaveAddr(int_t channel)**	Start waiting to receive slave address and waiting for STOP condition on slave mode. When a slave address is received, it notifies by calling "r_riic_hld_set_rx_end". Also, when a stop condition is issued after calling this function, the callback function "r_riic_hld_set_stop_asserted" is called.	[in] **int_t channel** RIIC channel to wait to slave address.	**DRV_SUCCESS** on success **DRV_ERROR** on failure

e_riic_slave_addr_num_t	**R_RIIC_GetSlaveAddrNum(int_t channel)**	Get the registration number of the slave address	[in] **int_t channel** RIIC channel to get to registration number of own slave address.	The registration number of the slave address that matches the slave address received from the master device. (RIIC_SLAVE_ADDR_NUM_MAX if no match).
int_t	**R_RIIC_GetVersion (st_ver_info_t *pinfo);**	Return version information for Low Level driver	[out] **st_ver_info_t *pinfo** pointer to version information struct to add version info to	**DRV_SUCCESS** on success Does not return any other values

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 "RIIC0", 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 "RIIC0", &drv_info);

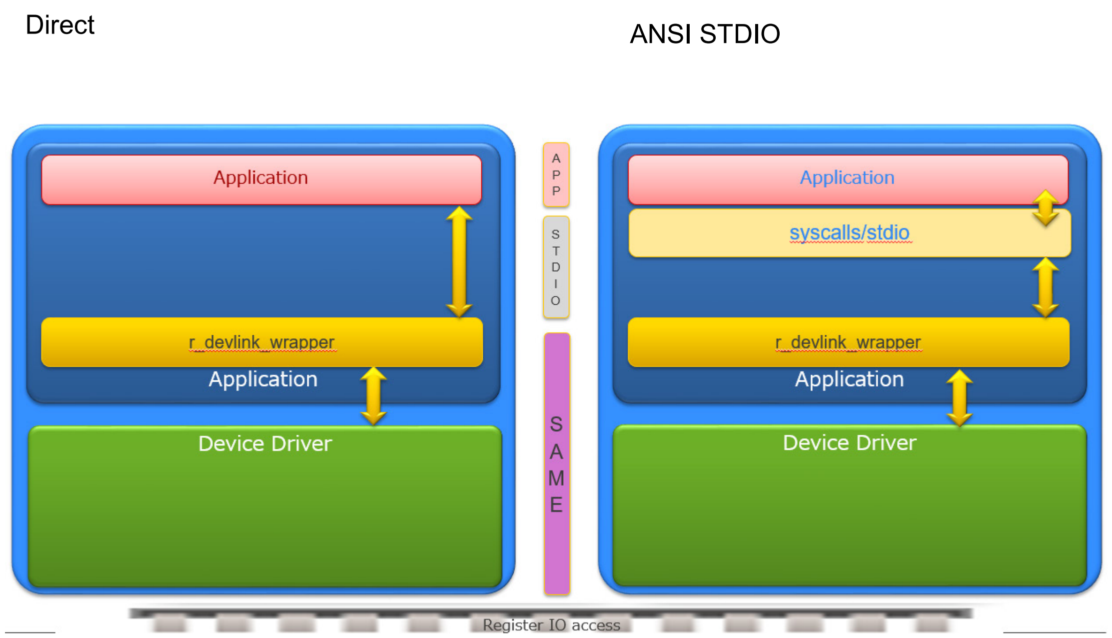
3.2 Direct

The following table shows the available direct functions.

Operation	Return	Function details
open	gs_direct_handle unique handle to driver	direct_open ("riic3", 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 ("riic3", &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_riic_handle;
uint8_t ch3_drv_name[] = "\\\\.\\riic3";                    /* open riic driver on channel 3 */

/* Note that the text “\\\\.\\” in the drive name signifies to the STDIO interface that the handle is to a
peripheral and is not an access to a standard file-based structure */

gs_riic_handle = open(ch3_drv_name, O_RDWR);
```

4.2 Control – Set Configuration Settings

```
st_riic_config_t set_cfg;
set_cfg.riic_mode = RIIC_MODE_MASTER;
set_cfg.slave_address_enable[0] = false;
set_cfg.slave_address_length[0] = RIIC_SUB_ADDR_WIDTH_16_BITS;
set_cfg.frequency = RIIC_FREQUENCY_100KHZ;
set_cfg.duty = RIIC_DUTY_50;
set_cfg.rise_time = 0u;
set_cfg.fall_time = 0u;
set_cfg.noise_filter_stage = RIIC_FILTER_NOT_USED;
set_cfg.timeout = RIIC_TIMEOUT_NOT_USED ;
set_cfg.format = RIIC_FORMAT_I2C;
set_cfg.host_address_enabled = false;
set_cfg.tei_priority = 9u;
set_cfg.ri_priority = 9u;
set_cfg.ti_priority = 9u;
set_cfg.spi_priority = 9u;
set_cfg.sti_priority = 9u;
set_cfg.naki_priority = 9u;
set_cfg.ali_priority = 9u;
set_cfg.tmoi_priority = 9u;

result = control(gs_riic_handle, CTL_RIIC_SET_CFG, &set_cfg);
```

4.3 Control – Get Configuration Settings

```
st_riic_config_t get_cfg;

result = control(gs_riic_handle, CTL_RIIC_GET_CFG, &get_cfg);
```

4.4 Control – Read

```
st_r_drv_riic_transfer_t transfer_parameters;

transfer_parameters.device_address = 0xA0u;                /* device I2C address */
transfer_parameters.sub_address_type = RIIC_SUB_ADDR_WIDTH_16_BITS;
                                                            /* device internal addressing mode */
transfer_parameters.sub_address = 0u;                      /* device internal address to read from */
transfer_parameters.number_of_bytes = 8u;                  /* Number of bytes to read from device */
transfer_parameters.p_data_puffer = &buffer_location;      /* where to store data read from device */

result = control(gs_riic_handle, CTL_RIIC_READ, &transfer_parameters);
```

4.5 Control – Write

```
st_r_drv_riic_transfer_t transfer_parameters;

transfer_parameters.device_address = 0x85u;                /* device I2C address */
transfer_parameters.sub_address_type = RIIC_SUB_ADDR_WIDTH_16_BITS;
                                                            /* device internal addressing mode */
transfer_parameters.sub_address = 0u;                      /* device internal address to write to */
transfer_parameters.number_of_bytes = 8u;                  /* Number of bytes to write to device */
transfer_parameters.p_data_puffer = &buffer_location;      /* Location of data to write to device */

result = control(gs_riic_handle, CTL_RIIC_WRITE, &transfer_parameters);
```

4.6 Close

```
close(gs_riic_handle);
```

4.7 Get Version

```
st_ver_info_t info;
result = get_version(gs_riic_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

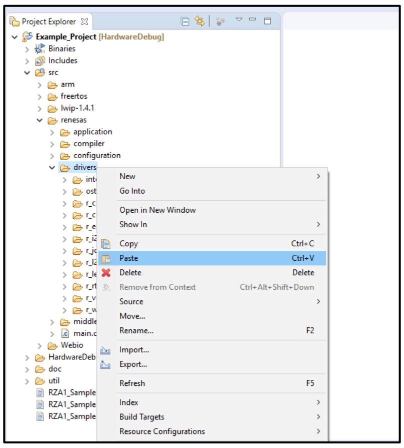
This section describes how to import the driver into your project. Generally, there are two steps in any IDE:

1) Copy the r_riic 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_riic 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_riic\inc'



Website and Support

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

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

Revision History

Rev.	Date	Description	
		Page	Summary
Rev.1.0	Jan 03, 2019	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.

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