RZ/A2M Group

RZ/A2M GPIO Driver

Introduction

This application note describes the operation of the software GPIO 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.

Driver Dependencies

This driver depends on:

- Drivers
  - STDIO

Target Device

RZ/A2M Group

Contents

1. Outline of the GPIO Driver 4

2. Description of the Software Driver 4

2.1 Structure 4

2.2 Description of each file 5

2.3 Driver API 6

3. Accessing the Driver 8

3.1 STDIO 8

3.2 Direct 9

3.3 Comparison 9

4. Example of Use 10

4.1 Open 10

4.2 Control – Set Configuration Settings 10

4.3 Control – Get Configuration Settings 10

4.4 Write the Logic Level to a Pin 10

4.5 Read the Logic Level from a Pin 10

4.6 Close 11

4.7 Get Version 11

5. OS Support 11

6. How to Import the Driver 11

6.1 e^2 studio 11

Revision History 12

Referenced Documents

Document Type	Document Name	Document No.
User's Manual	RZ/A2M Hardware Manual	R01UH0746EJ
Application Note	RZ/A2M Smart Configurator User's Guide: e^2 studio	R20AN0583EJ
Application Note	OS Abstraction Middleware	R11AN0309EG

List of Abbreviations and Acronyms

Abbreviation	Full Form
ANSI	American National Standards Institute
GPIO	General Purpose Input Output
HLD	High Layer / Level Driver
LLD	Low Layer / Level Driver
MCU	Microcontroller Unit
OS	Operating System
PDR	Port Direction Register
PFS	Pin Function Control (Select) Register
PIDR	Port Input Data Register
PMR	Port Mode Register
PODR	Port Output Data Register
RTOS	Real Time Operating System
STDIO	Standard Input / Output

Table 1-1 List of Abbreviations and Acronyms

1. Outline of the GPIO Driver

This MCU provides a total of 22 sets of dedicated port pins and general-purpose input/output port pins P0 to PL, and JP00 and JP01. Each pin is also configurable as an I/O pin of a peripheral module or an input pin for an interrupt. All pins are set to nonuse immediately after a reset (Hi-Z input protection), and pin functions can be switched by register settings.

The general-purpose input/output port pins are multiplexed with the peripheral on-chip module functions. The functions multiplexed on these pins can be selected as desired by setting of the registers.

Each port has the port direction register (PDR) that selects non-use, input, or output, the port output data register (PODR) that holds data for output, the port input register (PIDR) that indicates the pin states, and the port mode register (PMR) that specifies the pin function of each port.

Each port has the control register (PFS) that is used to select the input/output function and interrupt pin from the multiplexed port pins, and to assign the function to the selected pin.

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

The GPIO driver provides a layer of abstraction between these registers and the user application, allowing for simplified porting of the application to a different device.

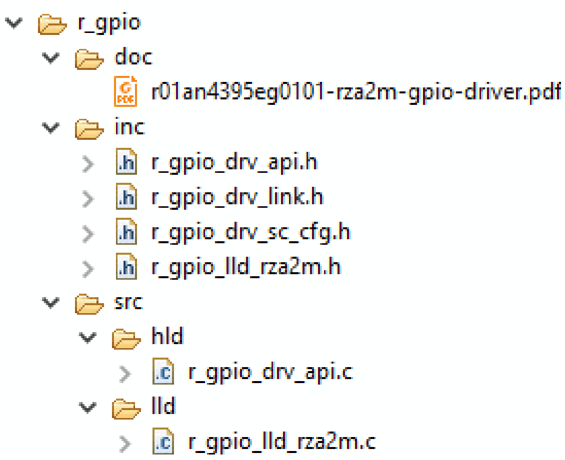
2. Description of the Software Driver

The key features of the driver include:

- Allocating individual pin function to GPIO or peripheral
- Setting GPIO pin to input / output / or Hi-Z (not used)
- Configuring GPIO pin to generate interrupts
- Writing the logic level on an output pin
- Reading the logic level of an input pin
- Setting the peripheral function for a pin allocated to a peripheral

2.1 Structure

The GPIO 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.



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_gpio_drv_api.h	Application	The only API header file to include in application code
High Layer Driver (HLD) Source		
r_gpio_drv_api.c	Private (HLD only)	High Layer Driver (HLD) source code implementing the driver API functions
Abstraction Link between High and Low Layer Drivers (HLD/LLD Link)		
r_gpio_drv_link.h	Private (HLD/LLD only)	Header file intended as an abstraction between low and high layers. This header will include the device specific configuration file "r_gpio_lld_rza2m.h"
Low Layer Driver (LLD) Source		
r_gpio_lld_rza2m.c	Private (LLD only)	Provides the implementation of the Low Layer Driver interface
r_gpio_lld_rza2m.h	Private (LLD only)	Low Layer Driver header file. Provides the High Layer Driver with access to the LLD driver functions
Smart Configurator		
r_gpio_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 Driver API

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

The API functions can be seen in the below table:

Return Type	Function	Description	Arguments	Return
int_t	**gpio_hld_open(** st_stream_ptr_t p_stream**)**	Driver initialisation interface is mapped to open function called directly using the st_r_driver_t GPIO driver handle g_gpio_driver: i.e. **g_gpio_driver.open()**	[in] **p_stream** driver handle.	**DRV_SUCCESS** Open Success **DRV_ERROR** Open Error
void	**gpio_hld_close(** st_stream_ptr_t p_stream**)**	Driver close interface is mapped to close function called directly using the st_r_driver_t GPIO driver structure g_gpio_driver: i.e. **g_gpio_driver.close()**	[in] **p_stream** driver handle.	**None**
int_t	**gpio_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 GPIO driver structure g_gpio_driver: i.e. **g_gpio_driver.read()**	[in] **p_stream** driver handle. [out] **p_buffer** buffer for returned data. [in] **count** size of buffer.	**Amount of Data Read** **DRV_ERROR** Write Error
int_t	**gpio_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 GPIO driver structure g_gpio_driver: i.e. **g_gpio_driver.write()**	[in] **p_stream** driver handle. [in] **p_buffer** data to send. [in] **count** size of data to send.	**DRV_SUCCESS** Write Success **DRV_ERROR** Write Error
int_t	**gpio_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 GPIO driver structure g_gpio_driver: i.e. **g_gpio_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** Operation Success **DRV_ERROR** Operation Error

Return Type	Function	Description	Arguments	Return
int_t	**gpio_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 GPIO driver structure g_gpio_driver: i.e. **g_gpio_driver.get_version()**	[in] **p_stream** handle to the (pre-opened) channel. [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.

3. Accessing the Driver

3.1 STDIO

The 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 "gpio", 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 "gpio", &drv_info);

3.2 Direct

The following table shows the available direct functions.

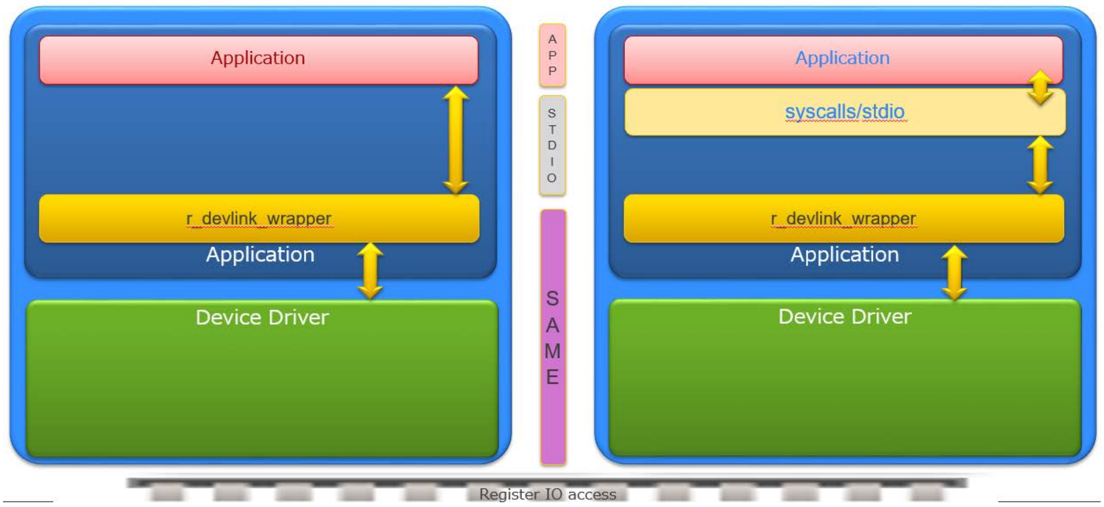
Operation	Return	Function details
open	gs_direct_handle unique handle to driver	direct_open ("gpio", 0);
close	DRV_SUCCESS successful operation, or driver specific error	direct_close (gs_direct_handle);
read	Number of characters read, -1 on error	direct_read (gs_direct_handle, buff, data_length);
write	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 ("gpio", &drv_info);

3.3 Comparison

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

Direct

ANSI STDIO



4. Example of Use

This section illustrates a simple example of opening the driver, configuring a port, writing to a pin, reading from a pin, and then closing the driver.

4.1 Open

```
int_t gs_gpio_handle;
char_t *driver_name = "\\\\.\\gpio";

/* 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_gpio_handle = open(driver_name, O_RDWR);
```

4.2 Control – Set Configuration Settings

```
st_r_drv_gpio_config_t pin_configuration;
int_t result;

pin_configuration.config.pin = GPIO_PORT_0_PIN_2;
pin_configuration.config.configuration.function = GPIO_FUNC_OUT;
pin_configuration.config.configuration.tint = GPIO_TINT_DISABLE;
pin_configuration.config.configuration.current = GPIO_CURRENT_8mA;

result = control(handle, CTL_GPIO_SET_CONFIGURATION,
                                    (void *) &pin_configuration);
```

4.3 Control – Get Configuration Settings

```
st_r_drv_gpio_config_t pin_configuration;
int_t result;

result = control(handle, CTL_GPIO_GET_CONFIGURATION,
                                    (void *) &pin_configuration);
```

4.4 Write the Logic Level to a Pin

```
st_r_drv_gpio_pin_rw_t pin_read_write;
int_t result;

pin_read_write.pin = GPIO_PORT_0_PIN_6;
pin_read_write.level = GPIO_LEVEL_HIGH;
result = control(handle, CTL_GPIO_PIN_WRITE,
                                    (void *) &pin_read_write);
```

4.5 Read the Logic Level from a Pin

```
st_r_drv_gpio_pin_rw_t pin_read_write;
e_r_drv_gpio_level_t logic_level;
int_t result;

pin_read_write.pin = GPIO_PORT_0_PIN_6;
result = control(handle, CTL_GPIO_PIN_READ,
                                    (void *) &pin_read_write);
logic_level = pin_read_write.level;
```

4.6 Close

```
close(gs_gpio_handle);
```

4.7 Get Version

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

5. OS Support

Operating system support for this driver is available using the OS abstraction module. For more details, please refer to the OS abstraction module application note (R11AN0309EG).

6. How to Import the Driver

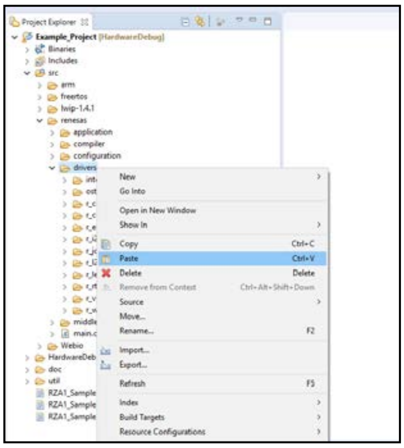
This section describes how to import the driver into your project.
Generally, there are two steps in any IDE:

1) Copy the 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_gpio 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}\generate\drivers\r_gpio\inc'



Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Sep.25.18	All	Created document.
1.01	May.10.19	All	Updated format

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

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

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/.