RZ/A2M Group

RZ/A2M CPG Driver

Introduction

This application note describes the operation of the software CPG 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:

- Drivers
  - STDIO

Referenced Documents

Document Type	Document Name	Document No.
User's Manual	RZ/A2M Hardware Manual	R01UH0746EJ
Application Note	OS Abstraction Middleware	R11AN0309EG

List of Abbreviations and Acronyms

Abbreviation	Full Form
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced RISC Machines
CPG	Clock Pulse Generator
CPU	Central Processing Unit
HLD	High Layer Driver
IDE	Integrated Development Environment
LLD	Low Layer Driver
OS	Operating System
PLL	Phase-Locked Loop
STDIO	Standard Input/Output

Table 1-1 List of Abbreviations and Acronyms

Contents

1. Outline of Software Driver

The CPG (Clock Pulse Generator) driver controls the CPU clock, image processing clock, internal bus clock, and both peripheral clocks.

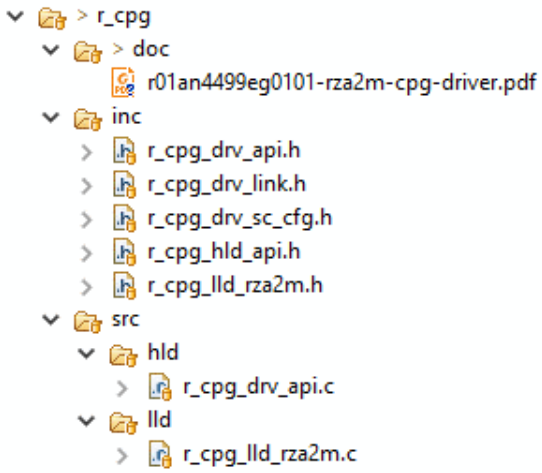
2. Description of the Software Driver

The key features of the driver include:

- Configures the main clock including the PLL and pre-PLL driver
- Sets each configurable sub-clock
- Sets inputs for all input-configurable clocks
- Configuration of external clock pins

2.1 Structure

The CPG 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_cpg_drv_api.h	Application	The only API header file to include in application code
High Layer Driver (HLD) Source		
r_cpg_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_cpg_drv_api.c	Private (HLD only)	High Layer Driver (HLD) source code enabling the driver API functions
r_cpg_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 Level API		
r_cpg_lld_xxxx.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_cpg_lld_cfg_xxxx.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_cpg_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 configuration file "r_cpg_lld_xxxx.h"
r_cpg_device_cfg.h	Should be included in "r_cpg_drv_api.h"	Header file intended as an abstraction between low and high layer. This header will include the device specific configuration file "r_cpg_lld_cfg_xxxx.h"
Low Layer Driver (LLD) Source		
r_cpg_lld_xxxx.c	Private (LLD only)	(Where "xxxx" is a device and board specific identification). Provides the definitions for the Low Layer Driver interface.
Smart Configurator		
r_cpg_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 table below:

Return Type	Function	Description	Arguments	Return
int_t	**cpg_hld_open(***st_stream_ptr_t* p_stream**)**	Driver initialisation interface is mapped to open function called directly using the st_r_driver_t CPG driver handle g_cpg_driver: i.e. **g_cpg_driver.open()**	[in] **p_stream** driver handle	>0: the handle to the driver **DRV_ERROR** Open failed
void	**cpg_hld_close(** *st_stream_ptr_t* p_stream**)**	Driver close interface is mapped to close function. Called directly using the st_r_driver_t CPG driver structure g_cpg_driver: i.e. **g_cpg_driver.close()**	[in] **p_stream** driver handle	**None**
int_t	**cpg_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 CPG driver structure g_cpg_driver: i.e. **g_cpg_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 succeeded **DRV_ERROR** Operation failed
int_t	**cpg_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 CPG driver structure g_cpg_driver: i.e. **g_cpg_driver.get_version()**	[in] **p_stream** Handle to the (pre-opened) channel. [out] **p_ver_info** Pointer to a version information structure.	**DRV_SUCCESS** Operation succeeded

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

3. Accessing the Driver

3.1 STDIO

The Driver 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 "cpg", O_RDWR);
close	DRV_SUCCESS successful operation, or driver specific error	close(gs_stdio_handle);
read	DRV_ERROR (read is not implemented in this STB driver)	read(gs_stdio_handle, buffer, buffer_length)
write	DRV_ERROR (write is not implemented in this STB driver)	write(gs_stdio_handle, buffer, 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 "cpg", &drv_info);

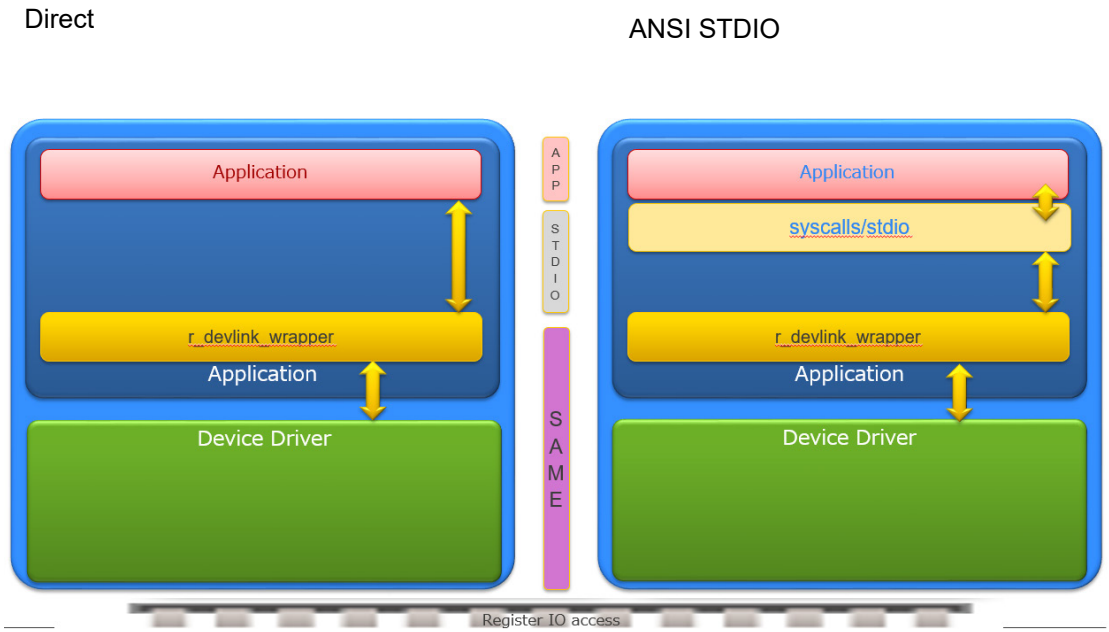
3.2 Direct

The following table shows the available direct functions.

Operation	Return	Function details
open	gs_direct_handle unique handle to driver	direct_open("cpg", 0);
close	DRV_SUCCESS successful operation, or driver specific error	direct_close(gs_direct_handle);
read	DRV_ERROR (read is not implemented in this CPG driver)	direct_read(gs_direct_handle, buff, data_length);
write	DRV_ERROR (write not implemented in this CPG driver)	direct_write(gs_direct_handle, buff, data_length);
control	DRV_SUCCESS control was processed, 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("cpg", &drv_info);

3.3 Comparison

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



4. Example of Use

This section gives simple examples for opening the driver, setting crystal frequency, setting the main clock, setting a sub clock, setting a sub clock source, setting an external clock, closing the driver, and finally getting the driver version.

4.1 Open

```
int_t gs_cpg_handle;
char_t *drv_name = "\\\\.\\cpg";

/* 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_cpg_handle = open(drv_name, O_RDWR);
```

4.2 Control – Set Crystal Frequency

```
int_t result;
float64_t xtal_frequency = 20000;

result = control(gs_cpg_handle, CTL_CPG_SET_XTAL_KHZ,
             (void *) &xtal_frequency);
```

4.3 Control – Set Main Clock

```
st_r_drv_cpg_set_main_t main_clk;

main_clk.main_clk_frequency_khz = 1056000;
main_clk.clk_src = CPG_CLOCK_SOURCE_PLL;
result = control(gs_cpg_handle, CTL_CPG_SET_MAIN_CLK, (void *) &main_clk);
```

4.4 Control – Set Sub Clock

```
st_r_drv_cpg_set_sub_t sub_clk;

sub_clk.clk_sub_src = CPG_SUB_CLOCK_ICLK;
sub_clk.sub_clk_frequency_khz = 100000;
result = control(gs_cpg_handle, CTL_CPG_SET_SUB_CLK, (void *) &sub_clk);
```

4.5 Control – Set Clock Source

```
st_r_drv_cpg_set_src_t clk_source;

clk_source.clk_sub_selection = CPG_SUB_CLOCK_HYPERBUS;
clk_source.clk_src_option = CPG_SUB_CLOCK_P1CLK_IN;
result = control(gs_cpg_handle, CTL_CPG_SET_CLK_SRC, (void *) &clk_source);
```

4.6 Control – Set External Clock

```
st_r_drv_cpg_ext_clk_t ext_clock;

ext_clock.clk_ext = CPG_CKIO_INVALID_UNSTBLE_NORM_ON_STDBY_DEEP_HIZ;
result = control(gs_cpg_handle, CTL_CPG_SET_EXT_CLK, (void *) &ext_clock);
```

4.7 Control – Get Clock Frequency

```
st_r_drv_cpg_get_clk_t cpg_get_clock_t;

cpg_get_clock_t.freq_src = CPG_FREQ_EXTAL;
```

```
result = control(gs_cpg_handle, CTL_CPG_GET_CLK, &cpg_get_clock_t);
```

4.8 Write

The stdio write() function is not supported by the CPG device driver.

4.9 Read

The stdio read() function is not supported by the CPG device driver.

4.10 Close

```
close(gs_cpg_handle);
```

4.11 Get Version

```
st_ver_info_t info;
result = get_version(gs_cpg_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

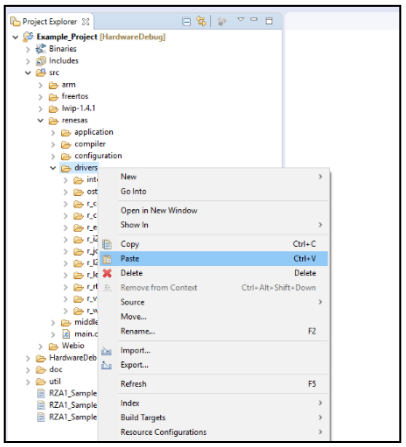
This section describes how to import the driver into your project.
Generally, there are two steps in any IDE:

1) Copy the driver to the location in the source tree that you require for your project.
2) Add the include path of the driver to the compiler.

6.1 e² studio

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

1) In Windows Explorer, right-click on the r_cpg folder, and click **Copy**.
2) In e² 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\drivers\r_cpg\inc'



Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Sep.19.18	All	Created document.
1.01	July.24.19	All	Removed LLD details, added get clock control function