RZ/A2M Group

RZ/A2M OSTM Driver

Introduction

This application note describes the operation of the software OSTM 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
  - INTC (Interrupt Controller)
  - CPG (Clock Pulse Generator)
  - STB (Standby Module)

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: e² studio	R20AN0583EJ
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
INTC	Interrupt Controller
LLD	Low Layer Driver
OS	Operating System
OSTM	Operating System Timer Module
STB	Standby
STDIO	Standard Input/Output

Table 1-1 List of Abbreviations and Acronyms

Contents

1. Outline of Software Driver

The OSTM (Operating System Timer Module) driver controls the 3 timer channels provided by the RZ/A2M MPU.

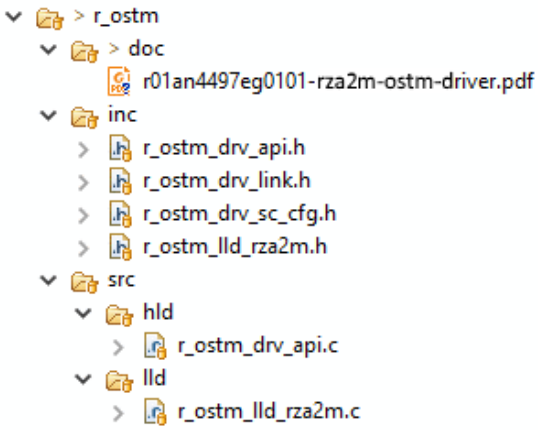
2. Description of the Software Driver

The key features of the driver include:

- Setting a timer into free-running mode for generating interrupts at non-fixed intervals
- Setting a timer into interval timer mode for generating interrupt requests at a fixed interval

2.1 Structure

The OSTM 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_ostm_drv_api.h	Application	The only API header file to include in application code
High Layer Driver (HLD) Source		
r_ostm_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_ostm_drv_api.c	Private (HLD only)	High Layer Driver (HLD) source code enabling the driver API functions
r_ostm_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 _ostm_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_ostm_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_ostm_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_ostm_lld_xxxx.h"
r_ostm_device_cfg.h	Should be included in "r_ostm_drv_api.h"	Header file intended as an abstraction between low and high layer. This header will include the device specific configuration file "r_ostm_lld_cfg_xxxx.h"
Low Layer Driver (LLD) Source		
r_ostm_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_ostm_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	**ostm_hld_open(***st_stream_ptr_t* p_stream**)**	Driver initialisation interface is mapped to open function called directly using the st_r_driver_t OSTM driver handle g_ostm_driver: i.e. **g_ostm_driver.open()**	[in] **p_stream** driver handle	>0: the handle to the driver **DRV_ERROR** Open failed
void	**ostm_hld_close(** *st_stream_ptr_t* p_stream**)**	Driver close interface is mapped to close function. Called directly using the st_r_driver_t OSTM driver structure g_ostm_driver: i.e. **g_ostm_driver.close()**	[in] **p_stream** driver handle	**None**
int_t	**ostm_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 OSTM driver structure g_ostm_driver: i.e. **g_ostm_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	**ostm_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 OSTM driver structure g_ostm_driver: i.e. **g_ostm_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 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 "ostm", O_RDWR);
close	DRV_SUCCESS successful operation, or driver specific error	close(gs_stdio_handle);
read	DRV_ERROR (read is not implemented in this OSTM driver)	read(gs_stdio_handle, buffer, buffer_length)
write	DRV_ERROR (write is not implemented in this OSTM 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 "ostm", &drv_info);

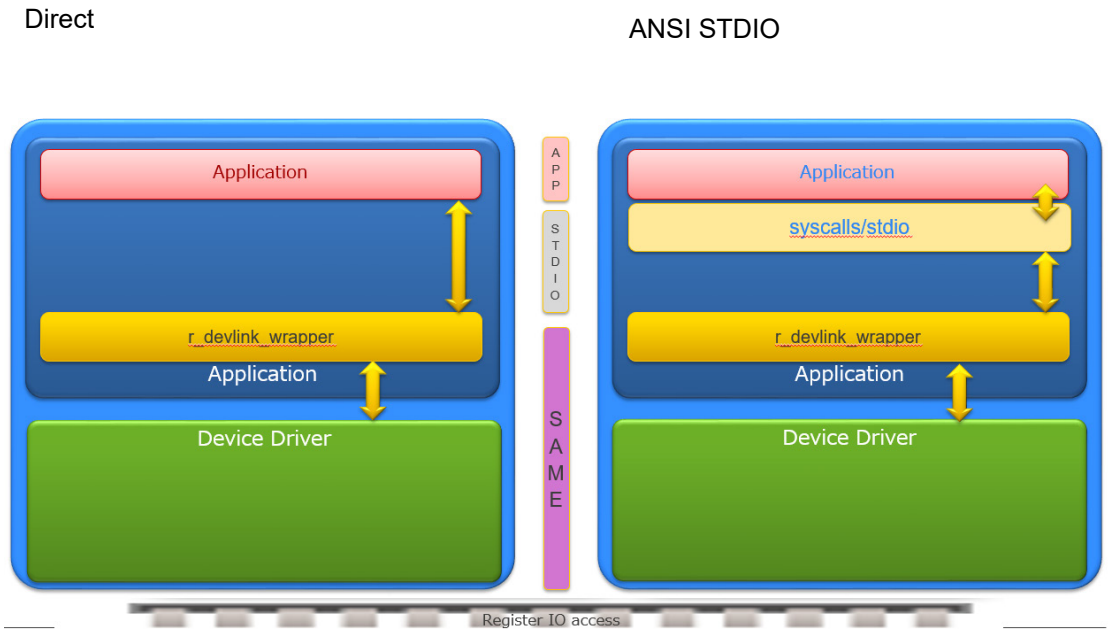
3.2 Direct

The following table shows the available direct functions.

Operation	Return	Function details
open	gs_direct_handle unique handle to driver	direct_open("ostm", 0);
close	DRV_SUCCESS successful operation, or driver specific error	direct_close(gs_direct_handle);
read	DRV_ERROR (read is not implemented in this OSTM driver)	direct_read(gs_direct_handle, buff, data_length);
write	DRV_ERROR (write not implemented in this OSTM 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("ostm", &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, starting a timer, stopping a timer, reconfiguring a timer, reading the counter, closing the driver, and finally getting the driver version.

4.1 Open

```
int_t gs_ostm_handle;
char_t *drv_name = "\\\\\.\\ostm";

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

4.2 Control – Start Timer

```
e_stb_module_t module;
int_t result;

module = MODULE_JCU;

result = control(gs_ostm_handle, CTRL_OSTM_START_TIMER, (void *) &module);
```

4.3 Control – Stop Timer

```
result = control(gs_ostm_handle, CTRL_OSTM_STOP_TIMER, (void *) &module);
```

4.4 Control – Reconfigure

```
result = control(gs_ostm_handle, CTRL_OSTM_RECONFIGURE, (void *) &module);
```

4.5 Control – Read Counter

```
uint32_t counter;
result = control(gs_ostm_handle, CTRL_OSTM_READ_COUNTER, (void *) &counter);
```

4.6 Write

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

4.7 Read

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

4.8 Close

```
close(gs_ostm_handle);
```

4.9 Get Version

```
st_ver_info_t info;
result = get_version(gs_ostm_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

6.1 e² studio

Please refer to the RZ/A2M Smart Configurator User's Guide: e² studio (R20AN0583EJ) for details on how to import drivers into projects in e2 studio using the Smart Configurator tool.

6.2 For Projects created outside e² studio

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.

Other required drivers, e.g. r_cbuffer, must be imported similarly.

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Sept 18, 2018	All	Created document
1.01	July 26, 2019	9	Added CTRL_OSTM_READ_COUNTER
		10	Added SC Import Details. Removed LLD API