S32G How to Solve GPIO Conflict After STR Resume

by John Li (nxa08200)

本文说明在STR Resume后，如何解决M核与A核的GPIO状态冲突的问题，并简单剖析了Linux GPIO驱动。

History	说明	作者
V1	● 创建本文	● John.Li

目录

1 背景说明与参考资料 ... 2
1.1 背景说明 ... 2
1.2 参考资料 ... 2
2 S32G linux GPIO驱动说明 ... 3
2.1 SIUL2 GPO功能说明 ... 3
2.2 Linux GPIO驱动DTS ... 4
2.3 Linux GPIO驱动初始化 ... 6
3 Rework办法 ... 7
3.1 SIUL2 GPO驱动PM函数 ... 7
3.2 Rework方法 ... 9
4 测试办法 ... 10

1 背景说明与参考资料

1.1 背景说明

在使用 M 核 Standby+ A 核 STR 的情况下，参考文档：《S32G_M7_STBYFULLBOOT_A53STR_V*.pdf》，JohnLi，https://community.nxp.com/t5/NXP-Designs-Knowledge-Base/S32G-M7-STBYFULLBOOT-A53STR/ta-p/1652687，关于 GPIO 整个流程是：

- Linux 先进入 STR 过程，在调用 GPIO 驱动时，由于 STR 时，SIUL2 模块是不供电的，所以所有的寄存器值都会丢失，也就是不会保留在寄存器里，所以 GPIO 的 PM suspend 函数会先标志成 GPIO 的状态寄存器已经保留在 DDR 中，假设目前为低，则保存为低。
- 当 Linux 执行 STR 结束，通知 M 核进入 Standby，然后 Resume，假设 M 核开始 full boot，并操作这个 GPIO 状态为高。
- 同时，M 核启动 A 核 Linux，并通知其 Resume，则 GPIO 驱动的 PM Resume 函数会将保存在 DDR 中的寄存器值回写进 SIUL2 的寄存器，从而将 M 核操作过的寄存器拉低，导致 M 核那边 GPIO 状态不正确。比如如果这个 GPIO 是 SPI 的 CS 管脚，则会导致 SPI 的通讯失败。

所以可以得到 M/A 核的 GPIO 驱动是会有冲突的，客户在正常情况下会分割 M/A 核系统间的 GPIO 硬件，一般建议的粒度是以 Group 为单位，也就是 M 核与 A 核使用不同的 GPIO group(一个 group 中常规有 16 个 GPIO 或更少)，同一个 group 的 GPIO 不能用做 M/A 核的同用。

但是 GPIO PM suspend/resume 函数会操作所有的 GPIO group 寄存器，所以需要做修改。

1.2 参考资料

本文基于 S32G2 RDB2 板及 Linux BSP37，G3 及 BSP37 以下版本相似。

分类	文档编号	名称	如何获得
文档	S32G2.pdf S32G3.pdf	S32G 芯片手册	从www.nxp.com/s32g下载
Bsp37 文档包	包括 S32G2/3_LinuxBSP_37.0_User_Manual.pdf	BSP 手册	从www.nxp.com个人帐号下载

2 S32G linux GPIO 驱动说明

2.1 SIUL2 GPO 功能说明

参考芯片手册 S32G2RM.pdf:

有两个 SIUL2 Channel 0/1，其中 GPIO 有两种访问方式：

NOTE

- For the array of 8-bit registers GPDO*n* and GPDI*n*:
  - — An 8-bit access to an unimplemented address (a "hole") within the array region will generate a transfer error.
  - — However, if you do a 16-bit or a 32-bit access and if any register instance is implemented within the accessed range, a transfer error will not be generated even if the range includes a hole.
- For the array of 16-bit registers PGPDO*n* and PGPDI*n*:
  - — A 16-bit access to an unimplemented address (a "hole") within the array region will generate a transfer error.
  - — However, a 32-bit access does not generate a transfer error for a hole irrespective of whether or not the other 16-bit range includes a register instance.

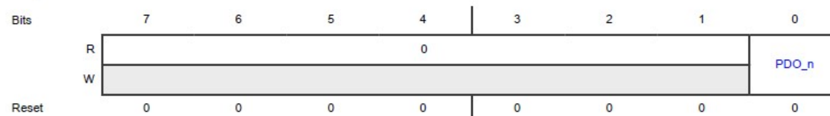
1. 8bit GPDO/I 寄存器方法，每个寄存器的最后一个 bit 标示一个 GPIO 状态，寄存器步进为 1。

1300	SIUL2 GPIO Pad Data Output (GPDO3)	8	RW	00
1301	SIUL2 GPIO Pad Data Output (GPDO2)	8	RW	00
1302	SIUL2 GPIO Pad Data Output (GPDO1)	8	RW	00
1303	SIUL2 GPIO Pad Data Output (GPDO0)	8	RW	00

For n = 0 to 101:

Register	Offset
GPDOn	1300h + (n + 3 - 2 × (n mod 4))

Diagram



Fields

Field	Description
7-1 —	Reserved
0 PDO_n	Pad Data Out Stores the data to be driven out on the external GPIO pad controlled by this register when the pad is configured as an output. PDO_*n* represents PDO[*n*], where *n* is the instance of the register. 0 - Pad Data Out Low. Logic low value 1 - Pad Data Out High. Logic high value

2. 16 bit PGPDO/I 寄存器方法，每个寄存器的 16 个 bits 标示一组 16 个 GPIO 的状态，寄存器步进为 2。

Offset (hex)	Register	Width (In bits)	Access	Reset value (hex)
1700 - 170A	SIUL2 Parallel GPIO Pad Data Out (PGPDO0 - PGPDO5) [1]	16	RW	0000
170E	SIUL2 Parallel GPIO Pad Data Out (PGPDO6)	16	RW	0000
1740 - 174A	SIUL2 Parallel GPIO Pad Data In (PGPDI0 - PGPDI5) [1]	16	RO	0000
174E	SIUL2 Parallel GPIO Pad Data In (PGPDI6)	16	RO	0000
1780 - 1794	SIUL2 Masked Parallel GPIO Pad Data Out (MPGPDO0 - MPGPDO5)	32	WORZ	0000_0000
1798	SIUL2 Masked Parallel GPIO Pad Data Out (MPGPDO6)	32	WORZ	0000_0000

Table continues on the next page...

Table continued from the previous page...

Register	Offset
PGPDO1	1700h
PGPDO0	1702h
PGPDO3	1704h
PGPDO2	1706h
PGPDO5	1708h
PGPDO4	170Ah

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PPDO 15	PPDO 14	PPDO 13	PPDO 12	PPDO 11	PPDO 10	PPDO 9	PPDO 8	PPDO 7	PPDO 6	PPDO 5	PPDO 4	PPDO 3	PPDO 2	PPDO 1	PPDO 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Description
15 PPDO15	Parallel Pad Data Out 15 0 - Logic low 1 - Logic high
14 PPDO14	Parallel Pad Data Out 14 0 - Logic low 1 - Logic high

2.2 Linux GPIO 驱动 DTS

Arch\arm64\boot\dts\freescale\s32g.dtsi

soc {

```
                siul2@4009c000 {
                           compatible = "simple-mfd";
…
/* Output pads */
                                      <OPADS_SIUL2_0 0x0 0x0 0x4009d700 0x0 0x10>, //此为 siul2_0/1 的 group 寄存器，PGPDO 和 PGPDI 地址
                                      <OPADS_SIUL2_1 0x0 0x0 0x44011700 0x0 0x18>,

                                      /* Input pads */
                                      <IPADS_SIUL2_0 0x0 0x0 0x4009d740 0x0 0x10>,
                                      <IPADS_SIUL2_1 0x0 0x0 0x44011740 0x0 0x18>,

                                      /* EIRQs */
                                      <EIRQS_SIUL2_1 0x0 0x0 0x44010010 0x0 0xb4>;
…
gpio: siul2-gpio@4009d700 {
                                     compatible = "nxp,s32g-siul2-gpio", "nxp,s32cc-siul2-gpio";

                                     reg = <OPADS_SIUL2_0        0    0x0 0x10>,//此为 siul2_0/1 的 group 寄存器，PGPDO 和 PGPDI 尺寸
                                          <OPADS_SIUL2_1        0    0x0 0x18>,
                                          <IPADS_SIUL2_0        0    0x0 0x10>,
                                          <IPADS_SIUL2_1        0    0x0 0x18>,
                                          <EIRQS_SIUL2_1        0    0x0 0xb4>,
                                          <IMCRS_SIUL2_1_EIRQS  0    0x0 0x80>;
                                     reg-names = "opads0", "opads1", "ipads0",
                                                    "ipads1", "eirqs", "eirq-imcrs";
                                     #gpio-cells = <2>;
                                     gpio-controller;
                                                /* GPIO 0-101 */ // siul0 有 102 个 GPIO
                                     gpio-ranges = <&pinctrl 0   0   102>,
                                                /* GPIO 112-190 *///  siul1 有 190-111-(21)=58 个 GPIO
                                                     <&pinctrl 112 112 79>;
```

gpio-reserved-ranges = <102 10>, // reserved ranges 表示不可访问的 gpio，注意一下，这儿在 linux 中，只是无法提供 linux 的访问接口，但是 PM suspend/resume 函数是直接操作 group 寄存器的，所以这儿注掉某个，某些 GPIO 并不能阻止 PM suspend/resume 函数访问寄存器。

```
                    <123 21>;
…
        };
```

从上面分析，可以看出来 Linux 驱动实际上是操作 PGPDO/I 而不是 GPDO/I，是以 group 为组操作的，这个比较容易用 regmap 驱动来实现。

2.3 Linux GPIO 驱动初始化

Drivers/gpio/gpio-siul2-s32cc.c

```
siul2_gpio_probe
|-> siul2_gpio_pads_init
|    |-> gpio_dev->siul2[i].opadmap = init_padregmap(pdev, gpio_dev, i,
                                                false);
regmap_conf.reg_stride = 2; //PGPDO/I 寄存器的步进为 2
regmap_conf.rd_table = platdata->pad_access[selector];//寄存器可读表参考 pad_access
if (input) {
            regmap_conf.writeable_reg = not_writable;
            regmap_conf.cache_type = REGCACHE_NONE;
    } else {
            regmap_conf.wr_table = platdata->pad_access[selector]; //寄存器可写表参考 pad_access
    }
return common_regmap_init(pdev, &regmap_conf, dts_tag);
//以下调用准备 GPIO 的数目，名字等
|->siul2_get_gpio_pinspec //of_parse_phandle_with_fixed_args(np, "gpio-ranges", 3,
                                        range_index, pinspec);
|->err = siul2_gpio_populate_names(&pdev->dev, gpio_dev);
//以下调用注册 GPIO 驱动的操作函数
|->gc->set = siul2_gpio_set;
    gc->get = siul2_gpio_get;
    gc->direction_output = siul2_gpio_dir_out;
    gc->direction_input = siul2_gpio_dir_in;
    gc->get_direction = siul2_gpio_get_dir;
```

以 siul2_gpio_set 为例：

```
siul2_gpio_set
|->siul2_gpio_set_val
|   |->siul2_get_pad_offset
siul2_offset_to_regmap
regmap_update_bits
```

所以实际操作 GPIO 的方法是通过 regmap 操作 PGPDO group 寄存器的某一个 bit 来操作某个 GPIO，不操作 GPDO 寄存器。

3 Rework 办法

3.1 SIUL2 GPO 驱动 PM 函数

Drivers/gpio/gpio-siul2-s32cc.c

```
static int __maybe_unused siul2_suspend(struct device *dev)
{
	struct siul2_gpio_dev *gpio_dev = dev_get_drvdata(dev);
	int i;
	for (i = 0; i < ARRAY_SIZE(gpio_dev->siul2); ++i) {
		regcache_cache_only(gpio_dev->siul2[i].opadmap, true);// * When a register map is marked as cache only writes to the register
 * map API will only update the register cache, they will not cause
 * any hardware changes.  This is useful for allowing portions of
 * drivers to act as though the device were functioning as normal when
 * it is disabled for power saving reasons.不再操作寄存器
		regcache_mark_dirty(gpio_dev->siul2[i].opadmap);// * Inform regcache that the device has been powered down or reset, so that
 * on resume, regcache_sync() knows to write out all non-default values
 * stored in the cache.
 *
 * If this function is not called, regcache_sync() will assume that
 * the hardware state still matches the cache state, modulo any writes that
 * happened when cache_only was true.进入 STR 时，表示硬件寄存器内的值已经不可信，在 resume 时需要从 cache 回写为寄存器(由于 regmap 是支持 cache 的，所以硬件寄存器的值在 DDR 中会有一个备份，这种方法实现 STR 时 siul2 寄存器保存在自刷新 DDR 中的功能)
	}
…
```

```
        return 0;
}

static int __maybe_unused siul2_resume(struct device *dev)
{
        struct siul2_gpio_dev *gpio_dev = dev_get_drvdata(dev);
        int ret = 0;
        int i;

        for (i = 0; i < ARRAY_SIZE(gpio_dev->siul2); ++i) {
                regcache_cache_only(gpio_dev->siul2[i].opadmap, false);
                ret = regcache_sync(gpio_dev->siul2[i].opadmap);
                if (ret)
                        dev_err(dev, "Failed to restore opadmap%d: %d\n", i,
                                ret);
        }
…
        return ret;
}
regcache_sync
|->regcache_default_sync
```

|-> regmap_volatile, regmap_writeable, regcache_read, regcache_reg_needs_sync 失败就会直接返回，如果成功则调用_regmap_write 将 cache 中的值写入硬件寄存器

以 regmap_writeable 为例：

```
regmap_writeable
|-> regmap_check_range_table
|    |-> regmap_reg_in_ranges(reg, table->yes_ranges,
                                  table->n_yes_ranges);
```

其中 yes_ranges 的定义为：

```
static const struct regmap_access_table s32g_siul21_pad_access_table = {
        .yes_ranges       = s32g_siul21_pad_yes_ranges,
        .n_yes_ranges     = ARRAY_SIZE(s32g_siul21_pad_yes_ranges),
};
static const struct regmap_range s32cc_siul20_pad_yes_ranges[] = {
```

```
	regmap_reg_range(SIUL2_PGPDO(0), SIUL2_PGPDO(0)),此为 siul2_0 的 7 组 PGPDO
	regmap_reg_range(SIUL2_PGPDO(1), SIUL2_PGPDO(1)),
	regmap_reg_range(SIUL2_PGPDO(2), SIUL2_PGPDO(2)),
	regmap_reg_range(SIUL2_PGPDO(3), SIUL2_PGPDO(3)),
	regmap_reg_range(SIUL2_PGPDO(4), SIUL2_PGPDO(4)),
	regmap_reg_range(SIUL2_PGPDO(5), SIUL2_PGPDO(5)),
	regmap_reg_range(SIUL2_PGPDO(6), SIUL2_PGPDO(6)),
};

static const struct regmap_access_table s32cc_siul20_pad_access_table = {
	.yes_ranges	= s32cc_siul20_pad_yes_ranges,
	.n_yes_ranges	= ARRAY_SIZE(s32cc_siul20_pad_yes_ranges),
};

static const struct regmap_range s32g_siul21_pad_yes_ranges[] = {
	regmap_reg_range(SIUL2_PGPDO(7), SIUL2_PGPDO(7)), 此为 siul2_1 的 4 组 PGPDO，注意没有 group 8
	regmap_reg_range(SIUL2_PGPDO(9), SIUL2_PGPDO(9)),
	regmap_reg_range(SIUL2_PGPDO(10), SIUL2_PGPDO(10)),
	regmap_reg_range(SIUL2_PGPDO(11), SIUL2_PGPDO(11)),
```

3.2 Rework 方法

经过以上分析，我们可以得之：

- Linux GPIO 驱动是以 16 个 GPIO 为一组的 PGPDO 为操作的基本单位的，所以没有办法精细到单个 GPIO，或者比较困难。
- Pad_access table 控制了一组 16 个 GPIO 在 Linux 中是否可以访问。

所以比较简单的 rework 办法是：将 M 核使用的 GPIO 所在组在 Linux 中从 pad_access table 中注掉。

以 PH_09 为例，参考芯片手册附件： S32Gx_IOMUX.xlsx：

PH_09	SIUL2_1	0000_0000	0x44010424	GPIO[121]
PH_09		0000_0001		SPI4_PCS2
PH_09		0000_0010		PFE_MAC0_RXD_O[3]
PH_09		0000_0011		PFE_MAC2_RXD_O[3]
PH_09	SIUL2_1	0000_0010	0x44010F70	PFE_MAC0_RXD_I[3]
PH_09	SIUL2_1	0000_0011	0x44011010	PFE_MAC2_RXD_I[3]

IOMUX Table 中有 PA~PG(对应 siul2_0 0~6 共 7 个 GPIO group)，PH,PJ~PL(对应 siul2_1 7,9~11，共 4 个 GPIO group)。

所以 rework 如下：

```
static const struct regmap_range s32g_siul21_pad_yes_ranges[] = {
	//regmap_reg_range(SIUL2_PGPDO(7), SIUL2_PGPDO(7)), //将 PH_09 所在 group, siul2_1 的第 7 组 PGPDO 寄存器从 pad_access table 中注掉，这样 GPIO 驱动就无法访问这个寄存器了，这样的 suspend/resume 后就不会再回写这个寄存器了。
…
};
```

4 测试办法

Auto Linux BSP 36.0 s32g274ardb2 ttyLF0

```
s32g274ardb2 login: root
root@s32g274ardb2:~# cd /sys/class/ gpio/
root@s32g274ardb2:/sys/class/gpio# ls
export  gpiochip297  gpiochip321  unexport // gpiochip321 为 S32G GPIO
root@s32g274ardb2:/sys/class/gpio# cat /sys/kernel/debug/gpio
…
gpiochip0: GPIOs 321-511, parent: platform/4009d700.siul2-gpio, 4009d700.siul2-gpio:
…
 gpio-442 (PH_09               )
…
root@s32g274ardb2:/sys/class/gpio# echo 442 > export
root@s32g274ardb2:/sys/class/gpio# ls
PH_09  export  gpiochip297  gpiochip321  unexport
cd PH_09
root@s32g274ardb2:/sys/class/gpio/PH_09# ls
active_low  device  direction  power  subsystem  uevent  value
root@s32g274ardb2:/sys/class/gpio/PH_09# cat direction
in
root@s32g274ardb2:/sys/class/gpio/PH_09# echo out > direction
root@s32g274ardb2:/sys/class/gpio/PH_09# cat direction
out
```

```
root@s32g274ardb2:/sys/class/gpio/PH_09# cat value
0
root@s32g274ardb2:/sys/class/gpio/PH_09# echo 1 > value
root@s32g274ardb2:/sys/class/gpio/PH_09# cat value
0  //所以 PH_09 无法操作写
```

同理可以测试 PH group 的所有 GPIO 均无法操作写。

同理测试 PJ_00 是可以的：

```
root@s32g274ardb2:/sys/class/gpio# echo 465 > export
root@s32g274ardb2:/sys/class/gpio# ls
PH_08  PH_09  PJ_00  PL_14  export  gpiochip297  gpiochip321  unexport
root@s32g274ardb2:/sys/class/gpio# cd PJ_00
root@s32g274ardb2:/sys/class/gpio/PJ_00# echo out > direction
root@s32g274ardb2:/sys/class/gpio/PJ_00# cat value
0
root@s32g274ardb2:/sys/class/gpio/PJ_00# echo 1 > value
root@s32g274ardb2:/sys/class/gpio/PJ_00# cat value
1
root@s32g274ardb2:/sys/class/gpio/PJ_00# cat /sys/kernel/debug/gpio
…
gpiochip0: GPIOs 321-511, parent: platform/4009d700.siul2-gpio, 4009d700.siul2-gpio:
…
 gpio-441 (PH_08               |sysfs               ) out lo
 gpio-442 (PH_09               |sysfs               ) out lo
…
 gpio-465 (PJ_00               |sysfs               ) out hi
 …
```

测试中需要注意一点，由于regmap的cache功能存在的原因，所以如果使用M核或是lauterbach修改了 GPIO 相关状态寄存器的值，则在 Linux 端通过/sys 文件系统去读的时候，只会读 cache 里的值，不会读到真实寄存器中的值，所以只有写的时候会真实写入到寄存器中，所以建议使用先写，后读的方式。