全 志 科 技
Allwinner Technology

A20 Script 和 GPIO 开发说明

V1.0

2013-03-15

Revision History

Version	Date	Changes compared to previous issue
v1.0	2013-03-15	初建版本

目录

1. 概述

1.1. 编写目的

介绍 A20 平台上 script 和 GPIO 的接口及使用方法。

1.2. 适用范围

适用于 A20 芯片对应平台.

1.3. 相关人员

Linux 内核和驱动开发人员。

2. 模块介绍

2.1. 模块功能介绍

Script 接口提供了解析 sys_config.fex 脚本的功能.
GPIO 接口提供了 GPIO 操作功能.

2.2. 相关术语介绍

2.2.1. Script 脚本

指的是打包到 img 中的 sys_config.fex 文件. 包含系统各模块配置参数.

2.2.2. Script 接口

指对 sys_config.fex 进行解析的函数.

2.2.3. GPIO

General Purpose Input Output, 即通用输入/输出, 也称总线扩展器.

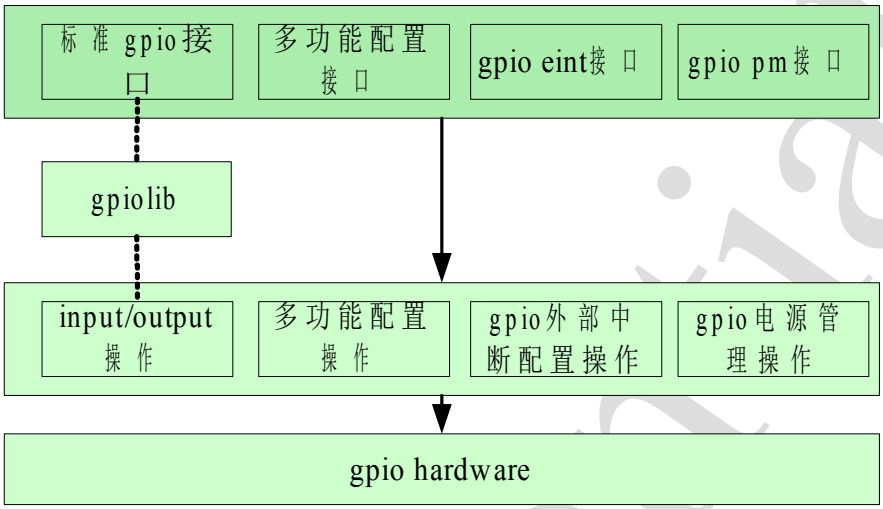
2.3. 模块配置介绍

GPIO 为内核必备的模块, 直接编译到 kernel 中, 无须 sys_config.fex 或 menuconfig 进行配置.

2.4. 源码结构介绍

Script 接口在 \linux-3.3\arch\arm\mach-sun7i\sys_config.c 中实现. 在 \linux-3.3\arch\arm\mach-sun7i\include\mach\sys_config.h 中声明;
gpio 模块源码在 \lichee\linux-3.3\arch\arm\mach-sun7i\gpio 目录下. 在 \linux-3.3\arch\arm\mach-sun7i\include\mach\gpio.h 中声明;
gpio_multi_func.c: 对 pin 脚的功能, pull, driverlevel 进行配置的接口.
gpio_init.c: 模块初始化. 注册 gpio chip.
gpio_eint.c: gpio 中断操作接口.
gpio_base.c: 提供了标准 gpio 需要的平台函数.

3. 模块体系结构描述



(1) 标准 GPIO 接口，处理输入输出, GPIO 申请释放等. 需要 linux 内核的 gpiolib 支持.
(2) 多功能配置接口，处理 GPIO 的功能配置, pull, driverlevel 等.
(3) gpio eint 接口，处理 gpio 外部中断的配置，触发模式等.
(4) gpio pm 接口，管理 gpio 驱动的待机,唤醒操作.

4. 模块数据结构描述

4.1. aw_gpio_chip

对标准 gpio_chip 的扩展（gpio_common.h）：

```
struct aw_gpio_chip {
    struct gpio_chip      chip;
    struct gpio_cfg_t     *cfg;
    struct gpio_eint_cfg_t   *cfg_eint;
    struct gpio_pm_t      *pm;
    void __iomem          *vbase;
    void __iomem          *vbase_einit;    /* gpio einit config reg base */
    u32                   irq_num;
    spinlock_t            lock;
};
```

(1) chip: 标准 gpio_chip 成员，被 gpiolib 管理，实现了对标准 gpio api 的支持.
(2) cfg: 平台定义的 gpio_cfg_t 成员， 实现了对功能配置的支持
(3) cfg_eint: 平台定义的 gpio_eint_cfg_t 成员， 实现了对 gpio 外部中断的支持
(4) pm: 平台定义的 gpio_pm_t 成员， 实现了对电源管理的支持
(5) vbase: 当前 chip 对应的 gpio 功能配置寄存器的起始虚拟地址. 如 PA 对应 0xf1c20800.
(6) vbase_eint: 当前 chip 对应的 gpio external int 配置寄存器的起始虚拟地址. 如 PA 对应 0xf1c20a00.
(7) irq_num: 当前 chip 对应的外部中断号. 如 PA 对应 PA_EINT(43).
(8) lock: 对各 api 的加锁处理.

4.2. gpio_cfg_t

功能配置函数.

```
typedef u32(*pset_cfg)(struct aw_gpio_chip *pchip, u32 offset, u32 val);
typedef u32(*pget_cfg)(struct aw_gpio_chip *pchip, u32 offset);
typedef u32(*pset_pull)(struct aw_gpio_chip *pchip, u32 offset, u32 val);
typedef u32(*pget_pull)(struct aw_gpio_chip *pchip, u32 offset);
typedef u32(*pset_drvlevel)(struct aw_gpio_chip *pchip, u32 offset, u32 val);
typedef u32(*pget_drvlevel)(struct aw_gpio_chip *pchip, u32 offset);

struct gpio_cfg_t {
    pset_cfg      set_cfg;
    pget_cfg      get_cfg;
    pset_pull     set_pull;
```

```
    pget_pull     get_pull;
    pset_drvlevel     set_drvlevel;
    pget_drvlevel     get_drvlevel;
};
```

4.3. gpio_eint_cfg_t

gpio 中断处理函数.

```
typedef u32(*peint_set_trig)(struct aw_gpio_chip *pchip, u32 offset, enum gpio_eint_trigtype trig_val);
typedef u32(*peint_get_trig)(struct aw_gpio_chip *pchip, u32 offset, enum gpio_eint_trigtype *pval);
typedef u32(*peint_get_enable)(struct aw_gpio_chip *pchip, u32 offset, u32 *penable);
typedef u32(*peint_set_enable)(struct aw_gpio_chip *pchip, u32 offset, u32 enable);
typedef u32(*peint_get_irqpd_sta)(struct aw_gpio_chip *pchip, u32 offset);
typedef u32(*peint_clr_irqpd_sta)(struct aw_gpio_chip *pchip, u32 offset);
typedef u32(*peint_set_debounce)(struct aw_gpio_chip *pchip, struct gpio_eint_debounce val);  /* for chip, not just port */
typedef u32(*peint_get_debounce)(struct aw_gpio_chip *pchip, struct gpio_eint_debounce *pval);  /* for chip, not just port */

struct gpio_eint_cfg_t {
    peint_set_trig          eint_set_trig;
    peint_get_trig          eint_get_trig;
    peint_set_enable        eint_set_enable;
    peint_get_enable        eint_get_enable;
    peint_get_irqpd_sta     eint_get_irqpd_sta;
    peint_clr_irqpd_sta     eint_clr_irqpd_sta;
    peint_set_debounce      eint_set_debounce;
    peint_get_debounce      eint_get_debounce;
};
```

4.4. gpio_pm_t

电源管理接口，目前暂不支持.

```
typedef u32(*psave)(struct aw_gpio_chip *pchip);
typedef u32(*presume)(struct aw_gpio_chip *pchip);

struct gpio_pm_t {
```

```
    psave           save;
    presume         resume;
};
```

5. GPIO 接口描述

5.1. Linux 标准 GPIO 接口

Linux 标准 GPIO 在 \linux-3.3\drivers\gpio\gpiolib.c 中实现，在 \linux-3.3\include\linux\gpio.h 中声明.

5.1.1. gpiolib_sysfs_init

注册 gpio_class 类.

5.1.2. gpio_export

int gpio_export(unsigned gpio, bool direction_may_change)
功能: 通过 sysfs 导出一个 gpio.
参数: gpio: gpio 编号
direction_may_change: 描述用户空间是否会改变 gpio 的输入输出状态.
返回值: 0 表示成功，否则表示失败.

5.1.3. gpio_export_link

int gpio_export_link(struct device *dev, const char *name, unsigned gpio)
功能: 为导出的 gpio 端口创建 link.
参数: dev: 创建 link 的设备
name: link 的名称.
gpio: gpio 编号
返回值: 0 表示成功，否则表示失败.

5.1.4. gpio_sysfs_set_active_low

int gpio_sysfs_set_active_low(unsigned gpio, int value)
功能: 设置导出 gpio 的 active_low 属性.
参数: value: 非 0 表示使用 active_low.
gpio: gpio 编号
返回值: 0 表示成功，否则表示失败.

5.1.5. gpio_unexport

void gpio_unexport(unsigned gpio)
功能: 取消 gpio 的导出.

参数: gpio: gpio 编号
返回值: 无.

5.1.6. gpiochip_add

int gpiochip_add(struct gpio_chip *chip)
功能: 注册 gpio_chip.
参数: chip: gpio_chip 结构.
返回值: 0 表示成功, 否则表示失败.
该函数一般被平台调用, 用于支持标准 gpio 接口.

5.1.7. gpiochip_remove

int gpiochip_remove(struct gpio_chip *chip)
功能: 注销 gpio_chip.
参数: chip: gpio_chip 结构.
返回值: 0 表示成功, 否则表示失败.

5.1.8. gpiochip_find

struct gpio_chip *gpiochip_find(void *data,
int (*match)(struct gpio_chip *chip, void *data))
功能: 由 gpio 号查找对应 gpio_chip 结构.
参数: data: match 函数的第二个参数.
match: 平台提供的匹配函数.
返回值: 成功返回找到的 gpio_chip 句柄, NULL 表示失败.

5.1.9. gpio_request

int gpio_request(unsigned gpio, const char *label)
功能: 申请 gpio. 获取 gpio 的访问权.
参数: gpio: gpio 编号.
label: gpio 名称, 可以为 NULL.
返回值: 0 表示成功, 否则表示失败.

5.1.10. gpio_free

void gpio_free(unsigned gpio)
功能: 释放 gpio.
参数: gpio: gpio 编号.
返回值: 无.

5.1.11. gpio_request_one

int gpio_request_one(unsigned gpio, unsigned long flags, const char *label)

功能：申请 gpio，并设置 input/output 状态.

参数： gpio： gpio 编号.

flags： 输入输出状态. 如 GPIOF_IN 表示输入, GPIOF_OUT_INIT_HIGH 表示输出高电平. 在\linux-3.3\include\linux\gpio.h 中定义.

```
/* make these flag values available regardless of GPIO kconfig options */
#define GPIOF_DIR_OUT  (0 << 0)
#define GPIOF_DIR_IN    (1 << 0)

#define GPIOF_INIT_LOW (0 << 1)
#define GPIOF_INIT_HIGH(1 << 1)

#define GPIOF_IN       (GPIOF_DIR_IN)
#define GPIOF_OUT_INIT_LOW   (GPIOF_DIR_OUT | GPIOF_INIT_LOW)
#define GPIOF_OUT_INIT_HIGH   (GPIOF_DIR_OUT | GPIOF_INIT_HIGH)
```

label： gpio 名称，可以为 NULL.

返回值: 0 表示成功，否则表示失败.

5.1.12. gpio_request_array

int gpio_request_array(const struct gpio *array, size_t num)

功能：申请一组 gpio，并设置 input/output 状态. 即对一组 gpio 的每一项执行 gpio_request_one 操作.

参数： array： gpio 数组.

num： array 的项数.

```
/**
 * struct gpio - a structure describing a GPIO with configuration
 * @gpio:   the GPIO number
 * @flags:  GPIO configuration as specified by GPIOF_*
 * @label:  a literal description string of this GPIO
 */
struct gpio {
    unsigned    gpio;
    unsigned long   flags;
    const char  *label;
};
```

返回值: 0 表示成功，否则表示失败.

5.1.13.gpio_free_array

void gpio_free_array(const struct gpio *array, size_t num)
功能: 释放一组 gpio. 即对一组 gpio 的每一项执行 gpio_free 操作.
参数: array: gpio 数组.
num: array 的项数.
返回值: 无.

5.1.14.gpiochip_is_requested

const char *gpiochip_is_requested(struct gpio_chip *chip, unsigned offset)
功能: 测试 gpio 是否已被申请.
参数: chip: gpio 所在 chip.
offset: chip 中的偏移.
返回值: NULL 表示当前 gpio 未被申请, 否则表示已申请.

5.1.15.gpio_direction_input

int gpio_direction_input(unsigned gpio)
功能: 将 gpio 设置为 input.
参数: gpio: gpio 编号.
返回值: 0 表示成功, 否则表示失败.

5.1.16.gpio_direction_output

int gpio_direction_output(unsigned gpio, int value)
功能: 将 gpio 设置为 output, 并设置电平值.
参数: gpio: gpio 编号.
value: gpio 电平值, 非 0 表示高, 0 表示低.
返回值: 0 表示成功, 否则表示失败.

5.1.17.gpio_set_debounce

int gpio_set_debounce(unsigned gpio, unsigned debounce)
功能: 设置 gpio 的 debounce time(硬件特性). 一般不用.
参数: gpio: gpio 编号.
debounce: debounce 值.
返回值: 0 表示成功, 否则表示失败.

5.1.18.__gpio_get_value

int __gpio_get_value(unsigned gpio)
功能: 获取 gpio 电平值. (gpio 已为 input/output 状态)
参数: gpio: gpio 编号.
返回值: gpio 电平, 1 表示高, 0 表示低.

5.1.19.__gpio_set_value

void __gpio_set_value(unsigned gpio, int value)
功能: 设置 gpio 电平值. (gpio 已为 output 状态)
参数: gpio: gpio 编号.
value: gpio 电平值, 非 0 表示高, 0 表示低.
返回值: 无.

5.1.20.__gpio_cansleep

int __gpio_cansleep(unsigned gpio)
功能: 获取 gpio 对应 gpio_chip 的 can_sleep 标记. (描述 gpio 在配置时是否可睡眠)
参数: gpio: gpio 编号.
返回值: 对应 gpio_chip 的 can_sleep 成员. 一般为非 0. 以防止操作 gpio 时 sleep.

5.1.21.__gpio_to_irq

int __gpio_to_irq(unsigned gpio)
功能: 获取 gpio 对应中断号.
参数: gpio: gpio 编号.
返回值: 获取 gpio 对应的 irq 号, 若无则返回-ENXIO.

5.1.22.gpio_get_value_cansleep

int gpio_get_value_cansleep(unsigned gpio)
功能: 功能与__gpio_get_value 相同, 但函数首先会_cond_resched(), 根据需要进行调度.
参数: gpio: gpio 编号.
返回值: 对应 gpio 的电平值.

5.1.23.gpio_set_value_cansleep

void gpio_set_value_cansleep(unsigned gpio, int value)
功能: 功能与__gpio_set_value 相同, 但函数首先会_cond_resched(), 根据需要进

行调度.

参数: gpio: gpio 编号.
value: gpio 电平值, 非 0 表示高, 0 表示低.

返回值: 无.

5.2. 多功能配置接口

5.2.1. sw_gpio_setcfg

u32 sw_gpio_setcfg(u32 gpio, u32 val)
功能: 配置 gpio 的功能.
参数: gpio: 全局 gpio 号
val: 配置值
返回值: 成功返回 0, 失败返回错误行号.

5.2.2. sw_gpio_getcfg

u32 sw_gpio_getcfg(u32 gpio)
功能: 获取 gpio 的配置值.
参数: gpio: 全局 gpio 号
返回值: 成功返回配置值, 失败返回 GPIO_CFG_INVALID.

5.2.3. sw_gpio_setpull

u32 sw_gpio_setpull(u32 gpio, u32 val)
功能: 配置 gpio 的 pull.
参数: gpio: 全局 gpio 号
val: pull 值
返回值: 成功返回 0, 失败返回错误行号.

5.2.4. sw_gpio_getpull

u32 sw_gpio_getpull(u32 gpio)
功能: 获取 gpio 的 pull 值.
参数: gpio: 全局 gpio 号
返回值: 成功返回 pull 值, 失败返回 GPIO_PULL_INVALID.

5.2.5. sw_gpio_setdrvlevel

u32 sw_gpio_setdrvlevel(u32 gpio, u32 val)
功能: 配置 gpio 的 driver level.

参数: gpio: 全局 gpio 号
val: driver level 值
返回值: 成功返回 0, 失败返回错误行号.

5.2.6. sw_gpio_getdrvlevel

u32 sw_gpio_getdrvlevel(u32 gpio)
功能: 获取 gpio 的 driver level 值.
参数: gpio: 全局 gpio 号
返回值: 成功返回 driver level 值, 失败返回 GPIO_DRVLVL_INVALID.

5.2.7. sw_gpio_setall_range

u32 sw_gpio_setall_range(struct gpio_config *pcfg, u32 cfg_num)
功能: 配置一组 gpio 的 mul sel, pull, driver level, data.
参数: pcfg: 配置参数数组, 输入.
cfg_num: pcfg 数组项数
返回值: 成功返回 0, 失败返回错误行号.

```
struct gpio_config {
    u32gpio;        /* gpio global index, must be unique */
    u32     mul_sel;    /* multi sel val: 0 - input, 1 - output... */
    u32     pull;       /* pull val: 0 - pull up/down disable, 1 - pull up... */
    u32     drv_level;  /* driver level val: 0 - level 0, 1 - level 1... */
    u32data;        /* data val: 0 - low, 1 - high, only vaild when mul_sel is
input/output */
};
```

注: 对于 pcfg 的 data 成员, 只有在 pcfg->mul_sel 为 1, 即 output 时, 函数内部才会处理: 将该 pin 的 data 位设为高或低.

5.2.8. sw_gpio_getall_range

u32 sw_gpio_getall_range(struct gpio_config *pcfg, u32 cfg_num)
功能: 获取一组 gpio 的 mul sel, pull, driver level, data.
参数: pcfg: 配置参数数组, 输出.
cfg_num: pcfg 数组项数
返回值: 成功返回 0, 失败返回错误行号.

注: 对于 pcfg 的 data 成员, 只有在 pcfg->mul_sel 为 0/1, 即 input/output 时, 函数内部才会处理: 读取该 pin 的 data 值, 并赋给 pcfg->data.

5.2.9. sw_gpio_dump_config

void sw_gpio_dump_config(struct gpio_config *pcfg, u32 cfg_num)
功能: 打印一组 gpio 的 mul sel, pull, driver level, data.
参数: pcfg: 配置参数数组, 输出.
cfg_num: pcfg 数组项数
返回值: 无.

5.2.10. sw_gpio_suspend3

u32 sw_gpio_suspend(void)
功能: gpio 驱动 suspend 的处理. 暂未实现.
返回值: 成功返回 0, 失败返回错误行号.

5.2.11. sw_gpio_resume

u32 sw_gpio_resume(void)
功能: gpio 驱动 resume 的处理. 暂未实现.
返回值: 成功返回 0, 失败返回错误行号.

5.3. GPIO 中断接口

5.3.1. 功能说明

sw_gpio_eint_set_trigtype: 设置单个 gpio 的触发模式.
sw_gpio_eint_get_trigtype: 获取单个 gpio 的触发模式.
sw_gpio_eint_set_enable: 设置单个 gpio 中断的 enable 状态. 1 表示 enable, 0 表示 disable.
sw_gpio_eint_get_enable: 获取单个 gpio 中断的 enable 状态.
sw_gpio_eint_clr_irqpd_sta: 清单个 gpio 中断的 irq pending. 若 pending 位未置则不处理.
sw_gpio_eint_get_irqpd_st: 获取单个 gpio 中断的 pending 状态. 1 表示产生中断, 0 表示没有.
sw_gpio_eint_set_debounce: 设置单个 gpio 所属 chip 的 debounce, 一般不用.
sw_gpio_eint_get_debounce: 获取单个 gpio 所属 chip 的 debounce 信息, 一般不用.

sw_gpio_eint_setall_range: 设置一组 gpio 的 pull, driver level, trig type 等信息, 设置完后, 可以通过 request_irq 申请该中断.
sw_gpio_eint_getall_range: 获取一组 gpio 的 pull, driver level, trig type 等信息.
sw_gpio_eint_dumpall_range: 打印一组 gpio 的 pull, driver level, trig type 等信息.

调试用.

sw_gpio_irq_request: 申请 gpio 中断. 内部调用 sw_gpio_eint_setall_range 将 gpio 配成中断功能. 因此用户不必重复调用 sw_gpio_eint_setall_range.

sw_gpio_irq_free: 释放 gpio 中断.

实际使用中, 一般只用到 sw_gpio_irq_request 和 sw_gpio_irq_free.

5.3.2. 函数说明

5.3.2.1. sw_gpio_irq_request

u32 sw_gpio_irq_request(u32 gpio, enum gpio_eint_trigtype trig_type,
peint_handle handle, void *para)

功能: 申请 gpio 中断.

参数: gpio: gpio 编号. 如 PA0 对应 GPIOA(0).

trig_type: 触发类型.

```
enum gpio_eint_trigtype {
    TRIG_EDGE_POSITIVE = 0,
    TRIG_EDGE_NEGATIVE,
    TRIG_LEVL_HIGH,
    TRIG_LEVL_LOW,
    TRIG_EDGE_DOUBLE,    /* positive/negative */
    TRIG_INALID
};
```

Handle: 中断回调函数. 当 gpio 中断触发时回调.

Para: handle 的参数. 必须为全局, 或者在堆中, 不能为栈中的局部变量.

返回值: 成功返回句柄, 失败返回 0.

注: sw_gpio_irq_request 内部会进行如下处理:

(1) 申请 gpio 的访问权. 调用 gpio_request.
(2) 检测 gpio 是否可配置成中断
(3) 配置 gpio 的功能(mul sel), pull, driver level, trig type, 并打开 gpio 中断的 enable 位. 这些通过调用 sw_gpio_eint_setall_range 来完成.
(4) 分配 gpio 中断句柄(gpio_irq_handle 结构)作为返回值
(5) 向 linux 内核申请中断. 调用 request_irq, 并传入 IRQF_SHARED 标记.

5.3.2.2. sw_gpio_irq_free

u32 sw_gpio_irq_free(u32 handle)

功能: 释放 gpio 中断. 与 sw_gpio_irq_request 对应.

参数: Handle: sw_gpio_irq_request 返回的句柄.

返回值：成功返回 0，失败返回错误行号.

注：sw_gpio_irq_free 内部会进行如下处理：

(1) 关闭 gpio 中断的 enable 位. 调用 sw_gpio_eint_set_enable.
(2) 清 gpio 中断的 pending 位. 调用 sw_gpio_eint_clr_irqpd_sta.
(3) 向 linux 内核释放 gpio 中断. 调用 free_irq.
(4) 释放 gpio 中断句柄. 即 sw_gpio_irq_request 返回的句柄.
(5) 释放 gpio 的访问权. 调用 gpio_free.

5.3.2.3. sw_gpio_eint_setall_range

u32 sw_gpio_eint_setall_range(struct gpio_config_eint_all *pcfg, u32 cfg_num)

功能：设置一组 gpio 的 pull, driver level, trig type 等信息. 设置完后，可以通过 request_irq 申请该中断

参数： pcfg: gpio 配置结构体.

```
struct gpio_config_eint_all {
    u32gpio;    /* the global gpio index */
    u32pull;        /* gpio pull val */
    u32     drvlvl;  /* gpio driver level */
    u32enabled;    /* in set function: used to enable/disable the eint, 1: enable, 0: disable
                    * in get function: return the eint enabled status, 1: enabled, 0: disabled
                    */
    u32irq_pd;  /* in set function: 1 means to clr irq pend status, 0 no use
                    * in get function: return the actual irq pend stauts, eg, 1 means irq occur.
                    */
    enum gpio_eint_trigtype trig_type; /* trig type of the gpio */
};
```

cfg_num: pcfg 的元素个数.

返回值：成功返回 0，失败返回出错行号.

5.3.2.4. sw_gpio_eint_getall_range

u32 sw_gpio_eint_getall_range(struct gpio_config_eint_all *pcfg, u32 cfg_num)

功能：获取 gpio 中断配置参数.

参数： pcfg: gpio 配置数组.
cfg_num: pcfg 的元素个数.

返回值：成功返回 0，失败返回出错行号.

5.3.2.5. sw_gpio_eint_dumpall_range

void sw_gpio_eint_dumpall_range(struct gpio_config_eint_all *pcfg, u32

cfg_num)
功能: 打印 gpio 中断配置参数.
参数: pcfg: gpio 配置数组.
cfg_num: pcfg 的元素个数.
返回值: 无.

5.3.2.6. sw_gpio_eint_set_trigtype

u32 sw_gpio_eint_set_trigtype(u32 gpio, enum gpio_eint_trigtype trig_type)
功能: 设置单个 gpio 的中断触发类型.
参数: gpio: gpio 编号.
trig_type: 触发类型.
返回值: 成功返回 0, 失败返回出错行号.

5.3.2.7. sw_gpio_eint_get_trigtype

u32 sw_gpio_eint_get_trigtype(u32 gpio, enum gpio_eint_trigtype *pval)
功能: 获取单个 gpio 的中断触发类型.
参数: gpio: gpio 编号.
pval: 保存获取的触发类型.
返回值: 成功返回 0, 失败返回出错行号.

5.3.2.8. sw_gpio_eint_set_debounce

u32 sw_gpio_eint_set_debounce(u32 gpio, struct gpio_eint_debounce dbc)
功能: 设置单个 gpio 所属 chip 的 debounce, 一般不用.
参数: gpio: gpio 编号.
dbc: 设置给硬件的 debounce 值.

```
struct gpio_eint_debounce {
    u32    clk_sel;  /* pio interrupt clock select, 0-LOSC, 1-HOSC */
    u32    clk_pre_scl; /* debounce clk pre-scale n, the select,
                        * clock source is pre-scale by 2^n.
                        */
};
```

返回值: 成功返回 0, 失败返回出错行号.

5.3.2.9. sw_gpio_eint_get_debounce

u32 sw_gpio_eint_get_debounce(u32 gpio, struct gpio_eint_debounce *pdbc)
功能: 获取单个 gpio 所属 chip 的 debounce, 一般不用.
参数: gpio: gpio 编号.
pdbc: 获取到的 debounce 值.
返回值: 成功返回 0, 失败返回出错行号.

5.3.2.10. sw_gpio_eint_clr_irqpd_sta

u32 sw_gpio_eint_clr_irqpd_sta(u32 gpio)
功能: 清单个 gpio 中断的 irq pending. 若 pending 位未置则不处理.
参数: gpio: gpio 编号.
返回值: 成功返回 0, 失败返回出错行号.

5.3.2.11. sw_gpio_eint_get_irqpd_sta

u32 sw_gpio_eint_get_irqpd_sta(u32 gpio)
功能: 获取单个 gpio 中断的 pending 状态. 1 表示中断产生, 0 表示没有.
参数: gpio: gpio 编号.
返回值: 1 表示 gpio 中断已产生, 0 表示未产生或获取失败.

5.3.2.12. sw_gpio_eint_get_enable

u32 sw_gpio_eint_get_enable(u32 gpio, u32 *penable)
功能: 获取单个 gpio 中断的 enable 位.
参数: gpio: gpio 编号.
penable: 获取到的 enable 状态.
返回值: 成功返回 0, 此时 penable 保存获取的 enable 状态, 1 表示 enable, 0 表示 disable; 失败返回出错行号.

5.3.2.13. u32 sw_gpio_eint_set_enable

u32 sw_gpio_eint_set_enable(u32 gpio, u32 enable)
功能: 设置单个 gpio 中断的 enable 位. 1 表示 enable, 0 表示 disable.
参数: gpio: gpio 编号.
enable: enable 状态, 0 表示 disable, 1 表示 enable.
返回值: 设置成功返回 0,; 失败返回出错行号.

6. Script 接口使用描述

6.1. script_get_item

6.1.1. 说明

script_item_value_type_e script_get_item(char *main_key, char *sub_key, script_item_u *item)

功能: 获取配置脚本中某一项子键值.

参数: main_key: 主键名.

Sub_key: 子键名.

Item: 保存获取到的子键值, 可能为 int, string 或 gpio, 因此用 script_item_u 联合体来描述:

```
/*
 * define data structure script item
 * @val: integer value for integer type item
 * @str: string pointer for sting type item
 * @gpio: gpio config for gpio type item
 */
typedef union {
    int                 val;
    char                *str;
    struct gpio_config  gpio;
} script_item_u;
```

返回值: 成功返回子键类型, 失败返回 SCIRPT_ITEM_VALUE_TYPE_INVALID.

子键类型用 script_item_value_type_e 描述:

```
/*
 * define types of script item
 * @SCIRPT_ITEM_VALUE_TYPE_INVALID:  invalid item type
 * @SCIRPT_ITEM_VALUE_TYPE_INT: integer item type
 * @SCIRPT_ITEM_VALUE_TYPE_STR: strint item type
 * @SCIRPT_ITEM_VALUE_TYPE_PIO: gpio item type
 */
typedef enum {
    SCIRPT_ITEM_VALUE_TYPE_INVALID = 0,
    SCIRPT_ITEM_VALUE_TYPE_INT,
    SCIRPT_ITEM_VALUE_TYPE_STR,
    SCIRPT_ITEM_VALUE_TYPE_PIO,
} script_item_value_type_e;
```

注: 对于 str 类型, script_item_u 的 str 的内存不需要调用者分配，由 script 模块内部申请和释放.

6.1.2. 示例

假设 sys_config.fex 有如下内容:

```
[card0_boot_para]
card_line                = 4
machine                  = "evb_v12"
sdc_d1                   = port:PF0<2><1><default><default>
sdc_d0                   = port:PF1<2><1><default><default>
sdc_clk                  = port:PF2<2><1><default><default>
sdc_cmd                  = port:PF3<2><1><default><default>
sdc_d3                   = port:PF4<2><1><default><default>
sdc_d2                   = port:PF5<2><1><default><default>
```

依次获取 card_line, machine , sdc_clk 的值代码如下:

```
script_item_u     val;
script_item_value_type_e    type;

/* 获取 card_line 值 */
type = script_get_item("card0_boot_para", "card_line", &val);
if(SCIRPT_ITEM_VALUE_TYPE_INT != type)
    printk("type err!");
printk("value is %d\n", val.val);

/* 获取 machine 值 */
type = script_get_item("card0_boot_para", "machine", &val);
if(SCIRPT_ITEM_VALUE_TYPE_STR != type)
    printk("type err!");
printk("value is %s\n", val.str);

/* 获取 sdc_clk 值 */
type = script_get_item("card0_boot_para", "sdc_clk", &val);
if(SCIRPT_ITEM_VALUE_TYPE_PIO != type)
    printk("type err!");
printk("value is: gpio %d, mul_sel %d, pull %d, drv_level %d, data %d\n",
    val.gpio.gpio,    val.gpio.mul_sel,    val.gpio.pull,    val.gpio.drv_level,
val.gpio.data, );
```

6.2. script_get_pio_list

6.2.1. 说明

int script_get_pio_list(char *main_key, script_item_u **list)
功能: 获取配置脚本中某一主键的所有 gpio 信息.
参数: main_key: 主键名.
list: 保存获取到的 gpio 数组指针.
返回值: 获取到的有效 gpio 个数.

6.2.2. 示例

假设 sys_config.fex 有如下内容:

```
[card0_boot_para]
card_line             = 4
machine               = "evb_v12"
sdc_d1                = port:PF0<2><1><default><default>
sdc_d0                = port:PF1<2><1><default><default>
sdc_clk               = port:PF2<2><1><default><default>
sdc_cmd         = port:PF3<2><1><default><default>
sdc_d3                = port:PF4<2><1><default><default>
sdc_d2                = port:PF5<2><1><default><default>
```

获取 gpio 数组代码如下:

```
script_item_u     *list = NULL;
int    cnt = 0;

/*  获取 gpio list */
cnt = script_get_pio_list("card0_boot_para", &list);
if(0 == cnt)
    printk("get card0_boot_para gpio list failed!\n");
else
    printk("cnt is %d!\n", cnt); /*  应该为 6 */
```

6.3. script_dump_mainkey

6.3.1. 说明

int script_dump_mainkey(char *main_key)
功能: 打印主键所有子键信息.
参数: main_key: 主键名.

返回值: 成功返回 0, 失败返回负数.

6.3.2. 示例

假设 sys_config.fex 中 card0_boot_para 配置信息如下:

```
[card0_boot_para]
card_ctrl          = 0
card_high_speed    = 1
card_line          = 4
sdc_d1             = port:PF0<2><1><default><default>
sdc_d0             = port:PF1<2><1><default><default>
sdc_clk            = port:PF2<2><1><default><default>
sdc_cmd          = port:PF3<2><1><default><default>
sdc_d3             = port:PF4<2><1><default><default>
sdc_d2             = port:PF5<2><1><default><default>
```

则 script_dump_mainkey("card0_boot_para")会打印:

```
++++++++++++++++++++++++++dump_mainkey++++++++++++++++++++++
+++++
      name:         card0_boot_para
      sub_key:     name              type          value
                   sdc_d1            gpio          (gpio: 119, mul: 2, pull 1, drv -1,
data -1)
                   sdc_d0            gpio          (gpio: 120, mul: 2, pull 1, drv -1,
data -1)
                   sdc_clk           gpio          (gpio: 121, mul: 2, pull 1, drv -1,
data -1)
                   sdc_cmd           gpio          (gpio: 122, mul: 2, pull 1, drv -1,
data -1)
                   sdc_d3            gpio          (gpio: 123, mul: 2, pull 1, drv -1,
data -1)
                   sdc_d2            gpio          (gpio: 124, mul: 2, pull 1, drv -1,
data -1)
                   card_ctrl         int           0
                   card_high_speed int             1
                   card_line         int           4
-------------------------dump_mainkey-------------------------
```

7. GPIO 接口使用描述

7.1. 如何确定用哪一套接口

(1) 若只处理输入或输出的情形, 用标准 GPIO 接口.

虽然多功能配置接口 sw_gpio_setcfg, sw_gpio_setall_range 等也可以配置输入输出, 但为兼顾代码的开放性, 建议用标准 gpio 接口 (gpio_direction_input, gpio_direction_output 等).

比如将 PA2 配置成 input:

推荐用:

```
gpio_direction_input(GPIOA(2));
```

不推荐用:

```
sw_gpio_setcfg(GPIOA(2), GPIO_CFG_INPUT);
```

再比如将 PB2 配置成 output, 并设置 data 为 1, 则

推荐用:

```
gpio_direction_output(GPIOB(2), 1);
```

不推荐用:

```
struct gpio_config cfg = {GPIOB(2), GPIO_CFG_OUTPUT, GPIO_PULL_DEFAULT,
                          GPIO_DRVLVL_DEFAULT, 1};
sw_gpio_setall_range(&cfg , 1);
```

(2) 若需要处理 pull, driver level, 则用多功能配置接口.

比如将 PC(3)设置为 input, 且要求 pull up, driver level 为 2, 则只能用多功能配置接口.

法一(单独配置):

```
gpio_direction_input(GPIOC(3));
sw_gpio_setpull(GPIOC(3), 1);
sw_gpio_setdrvlevel(GPIOC(3), 2);
```

法二(统一配置):

```
struct gpio_config cfg = {GPIOC(3), 0, 1, 2, 0};
sw_gpio_setall_range(&cfg , 1);
```

由于是 input, sw_gpio_setall_range 会将 cfg 的 data 忽略.

推荐用法二.

(3) gpio 中断相关配置用 gpio 中断接口.

7.2. 关于 GPIO 的申请和释放

(1) 申请 gpio 时，仅仅将 gpio 标记为已占用，不关心被谁占用，也不会配置硬件.
Gpio 已被占用时，再次申请会失败. 只有被释放后，解除了占用标记，才能被再次申请.

(2) 申请 gpio 的函数有 gpio_request, gpio_request_one, gpio_request_array. 后二者直接或间接调用了 gpio_request.
因此，以下写法会造成重复申请，导致错误:

```
/* 申请 gpio */
ret = gpio_request(gpio, NULL);
if(0 != ret)
        printk("gpio_request failed\n");
/* 错误，造成了重复申请 */
ret2 = gpio_request_one(gpio, flags, NULL);
if(ret2)
    printk("gpio_request_one failed\n");
...
/* 使用完，释放 gpio */
if(0 == ret)
    gpio_free(gpio);
```

需改为:

```
/* 申请 gpio，并配置 */
ret2 = gpio_request_one(gpio, flags, NULL);
if(ret2)
    printk("gpio_request_one failed\n");
...
/* 使用完，释放 gpio */
if(0 == ret2)
    gpio_free(gpio);
```

(3) 对于模块专用的 gpio，要求在模块初始化时申请 gpio，以防止别的模块再去申请并操作这些 gpio；模块卸载时释放 gpio;
例如, Sys_config.fex 中 PH14, PH15 配给 twi0 用.

```
[twi0_para]
twi_used            = 1
twi_scl             = port:PH14<2><default><default><default>
twi_sda             = port:PH15<2><default><default><default>
```

则在 twi 模块初始化中，检测到 twi0 被使用后，申请 gpio:

```
int  gpio_cnt, i = 0;
script_item_u   val, *list = NULL;
script_item_value_type_e   type;
```

```
/* 检查是否用到了 twi0 */
type = script_get_item("twi0_para", "twi_used", &val);
if(SCIRPT_ITEM_VALUE_TYPE_INT != type) {
    printk("type err!");
    return;
}
/* 如果用到了 twi0，则申请 gpio */
if(1 == val.val) {
    gpio_cnt = script_get_pio_list("twi0_para", &list);
    for(i = 0; i < gpio_cnt; i++)
        if(0 != gpio_request(list[i].gpio.gpio, NULL))
            printk("request gpio failed!");
}
```

在 twi 模块卸载函数中，释放 gpio.

```
int  gpio_cnt;
script_item_u   *list = NULL;

...
/* 释放模块初始化函数申请的 gpio */
gpio_cnt = script_get_pio_list("twi0_para", &list);
for(i = 0; i < gpio_cnt; i++)
    gpio_free(list[i].gpio.gpio);
}
```

(4) 对于标准 gpio 接口，一般情况下，使用前要先申请，使用完后要释放;
例如:

```
int gpio_index = GPIOE(5);
int request_sta = -1;

/* 申请 gpio */
request_sta = gpio_request(gpio_index, "pe_5");
if(0 != request_sta)
    printk("request gpio failed\n");
gpio_direction_input(gpio_index);
...
/* 释放 gpio */
if(0 == request_sta)
    gpio_free(gpio_index);
```

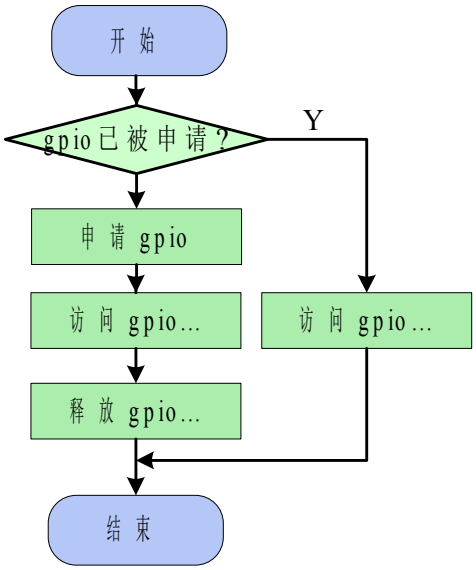
对于某些 api 如 gpio_request_one, gpio_request_array,内部包含了申请操作，因此不用再调 gpio_request. 比如:

```
int gpio_index = GPIOC(1);
int request_sta = 0;

/* 申请 gpio，并配置成 output, data 设为高 */
request_sta = gpio_request_one(gpio_index, GPIOF_OUT_INIT_HIGH,
"pc_1");
if(0 != request_sta)
    printk("request gpio failed\n");
...
/* 释放 gpio */
if(0 == request_sta)
    gpio_free(gpio_index);
```

gpio_request_array:

```
int request_sta = 0;
struct gpio gpio_arry[] = {
    {GPIOA(0), GPIOF_OUT_INIT_HIGH, "pa0"},
    {GPIOB(3), GPIOF_IN, "pb3"},
    {GPIOC(5), GPIOF_OUT_INIT_LOW, "pc5"},
    {GPIOH(2), GPIOF_IN, "ph2"},
};

/* 申请并配置一组 gpio */
request_sta = gpio_request_array(gpio_arry, ARRAY_SIZE(gpio_arry));
if(0 != request_sta)
    printk("request gpio failed\n");
...
/* 释放一组 gpio */
if(0 == request_sta)
    gpio_free_array(gpio_arry, ARRAY_SIZE(gpio_arry));
```

(5) 对于多功能配置接口和中断接口，内部会检测 gpio 是否已被申请，若是，则直接访问 gpio，不进行申请释放；否则申请 gpio，访问完后释放 gpio。 API 内部流程如下:



使用多功能配置接口和中断接口时，若不关心 gpio 的访问冲突，则不必进行申请和释放. 比如:

```
u32upio_index = GPIOE(5);
struct gpio_config_eint_all cfg_eint = {GPIOB(3), GPIO_PULL_DEFAULT, 1 ,
true, 0,
                                        TRIG_EDGE_NEGATIVE};
/* 直接访问，不需申请和释放  */
if(0 != sw_gpio_setcfg(upio_index, 3))
    printk("set gpio function failed\n");
if(0 != sw_gpio_setpull(upio_index, 1))
    printk("set gpio pull failed\n");
if(0 != sw_gpio_setdrvlevel(upio_index, 1))
    printk("set gpio driver level failed\n");
if(0 != sw_gpio_eint_setall_range(&cfg_eint, 1))
    printk("set gpio int failed\n");
...
```

上述代码没有冲突检测功能，比如 PE5 已被其他模块申请，则这里 sw_gpio_setcfg 等操作仍然会生效.

若需要防止 gpio 访问冲突，则还是要在访问前进行申请，访问完后进行释放.

```
u32upio_index = GPIOE(5);
int  req_status = 0;
struct gpio_config_eint_all cfg_eint = {GPIOB(3), GPIO_PULL_DEFAULT, 1 ,
true, 0,
                                        TRIG_EDGE_NEGATIVE};
/* 申请 gpio，防止访问冲突  */
req_status = gpio_request(upio_index, NULL);
```

```
if(0 != req_status) {
    printk("request gpio failed\n");
    return;
}
if(0 != sw_gpio_setcfg(upio_index, 3))
    printk("set gpio function failed\n");
...
/* 访问完释放 gpio */
if(0 == req_status)
    gpio_free(upio_index);
```

上述代码中，若 PE5 已被别的模块申请，则 gpio_request 就会失败并返回，从而防止访问冲突.

7.3. 常用场景

7.3.1. 使用标准 GPIO 接口

7.3.1.1. 将 PF3 设为 input，并获取其 data 值

```
u32upio_index = GPIOF(3);
int  data = -1;

/* 申请 gpio */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return -EINVAL;
}
/* 配置成 input */
if(0 != gpio_direction_input(upio_index))
    printk("set to input failed\n");
/* 获取 data 值 */
data = __gpio_get_value(upio_index);
/* 释放 gpio */
gpio_free(upio_index);
return data;
```

7.3.1.2. 将 PF3 设为 output，并将 data 设为高

```
u32upio_index = GPIOF(3);
int  status = -EINVAL;
```

```
/* 申请 gpio */
status = gpio_request_one(upio_index, GPIOF_OUT_INIT_HIGH, NULL);
if(0 != status)
    printk("gpio_request_one failed, status 0x%x\n", status);
else
    gpio_free(upio_index); /* 释放 gpio */
return;
```

7.3.1.3. PF3 已被设为 output，现将其 data 设为低

```
/* 将 PF3 的 data 设低 */
__gpio_set_value(GPIOF(3), 0);
```

7.3.1.4. 设置一组 gpio 的 input/output 状态

```
int  status;
struct gpio gpio_arry[] = {
    {GPIOA(0), GPIOF_OUT_INIT_HIGH, "pa0"},
    {GPIOB(3), GPIOF_IN, "pb3"},
    {GPIOC(5), GPIOF_OUT_INIT_LOW, "pc5"},
    {GPIOH(2), GPIOF_IN, "ph2"},
};

/* 设置一组 gpio 的 input/output 状态 */
status = gpio_request_array(gpio_arry, ARRAY_SIZE(gpio_arry));
if(0 != status)
    printk("gpio_request_array failed, status 0x%x\n", status);
else
    gpio_free_array(gpio_arry, ARRAY_SIZE(gpio_arry)); /* 释放 gpio */
```

7.3.2. 使用多功能配置接口

7.3.2.1. 将 PF3 配置成 2 号功能(SDC0_CMD)

```
u32upio_index = GPIOF(3);

/* 申请 gpio，防止访问冲突 */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return;
}
/* 功能配置 */
```

```
if(0 != sw_gpio_setcfg(upio_index, 2))
    printk("set gpio function failed\n");
/* 释放 gpio */
gpio_free(upio_index);
```

7.3.2.2. 将 PF3 配置成 2 号功能(SDC0_CMD), pull 设为 1

```
u32upio_index = GPIOF(3);

/* 申请 gpio，防止访问冲突 */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return;
}
/* 功能配置 */
if(0 != sw_gpio_setcfg(upio_index, 2))
    printk("set gpio function failed\n");
/* 设置 pull 值 */
if(0 != sw_gpio_setpull(upio_index, 1))
    printk("set pull failed\n");
/* 释放 gpio */
gpio_free(upio_index);
```

7.3.2.3. PF3 配置

```
u32upio_index = GPIOF(3);

/* 申请 gpio，防止访问冲突 */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return;
}
/* 功能配置 */
if(0 != sw_gpio_setcfg(upio_index, 2))
    printk("set gpio function failed\n");
/* 设置 pull 值 */
if(0 != sw_gpio_setpull(upio_index, 1))
    printk("set pull failed\n");
/* 设置 driver level 值 */
if(0 != sw_gpio_setdrvlevel(upio_index, 2))
    printk("set drv level failed\n");
/* 释放 gpio */
```

```
gpio_free(upio_index);
```

7.3.2.4. 获取 PF3 的 mul sel 值

```
u32upio_index = GPIOF(3);
u32mul_sel;

/* 申请 gpio，防止访问冲突 */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return;
}
/* 获取 mul sel */
mul_sel = sw_gpio_getcfg(upio_index));
/* 释放 gpio */
gpio_free(upio_index);
```

7.3.2.5. 获取 PF3 的 mul sel, pull, driver level, data 值

```
u32upio_index = GPIOF(3);
struct gpio_config gpio_cfg = {GPIOE(10)};

/* 申请 gpio，防止访问冲突 */
if(0 != gpio_request(upio_index, NULL)) {
    printk("request gpio failed\n");
    return;
}
/* 获取 mul sel, pull, driver level, data 值 */
if(0 != sw_gpio_getall_range(&gpio_cfg, 1))
    printk("sw_gpio_getall_range failed\n");
else
    printk("get PF3 mulsel %d, pull %d, driverlevel %d, data %d\n",
        gpio_cfg.mul_sel, gpio_cfg.pull, gpio_cfg.drv_level, gpio_cfg.data);
/* 释放 gpio */
gpio_free(upio_index);
```

7.3.2.6. 设置一组 gpio 的 mul sel, pull, driver level, data 值

```
    int  i;
    struct gpio_config gpio_cfg[] = {
        /* use GPIO_PULL_DEFAULT/GPIO_DRVLVL_DEFAULT if you donot
care */
```

```
        {GPIOE(10),  3,  GPIO_PULL_DEFAULT,  GPIO_DRVLVL_DEFAULT,
0},
        {GPIOA(13), 2, 1, 2, -1},
        {GPIOD(2),   1, 2, 1, 1},
        {GPIOG(8),   0, 1, 1, 0},
    };

    /* 申请 gpio，防止访问冲突   */
    for(i = 0; i < ARRAY_SIZE(gpio_cfg), i++)
        if(0 != gpio_request(gpio_cfg[i].gpio, NULL))
            goto end;
    /* 设置 mul sel, pull, driver level, data 值 */
    if(0 != sw_gpio_setall_range(gpio_cfg, ARRAY_SIZE(gpio_cfg)))
        printk("sw_gpio_setall_range failed\n");
end:
    /* 释放 gpio */
    while(i--)
        gpio_free(gpio_cfg[i].gpio);
```

7.3.3. 使用 gpio 中断接口

7.3.3.1. 申请 PA0 中断，并设为上升沿触发

```
u32upio_index = GPIOA(0);
u32handle = 0;

/* 申请 PA0 中断，并设为上升沿触发 */
handle = sw_gpio_irq_request(utemp, TRIG_EDGE_POSITIVE,
        (peint_handle)gpio_irq_handle_demo, (void *)&upio_index);
if(0 == handle) {
    printk("sw_gpio_irq_request failed\n");
    return;
}
...
/* 释放 PA0 中断 */
sw_gpio_irq_free(handle);
```

gpio_irq_handle_demo 函数如下:

```
/**
 * gpio_irq_handle_demo - gpio irq handle demo.
 * @para:   paras set by sw_gpio_irq_request
 *
```

```
 * Returns 0 if sucess, otherwise failed.
 */
u32 gpio_irq_handle_demo(void *para)
{
    u32upio_index = *(u32 *)para;
    printk("%s: upio_index 0x%08x\n", __func__, upio_index);
    /* 返回 0 表示成功，其余表示错误 */
    return 0;
}
```

注: sw_gpio_irq_request 内部会调用 gpio_request，因此不必在 sw_gpio_irq_request 之前申请 gpio.

7.3.3.2. 释放 PA0 中断

见上例.

7.3.3.3. 将 PA0 配置成下降沿触发

```
u32upio_index = GPIOA(0);
int  req_status;

/* 申请 gpio */
req_status = gpio_request(upio_index);
if(0 != req_status)
    printk("request gpio failed\n");
/* 设置 PA0 为下降沿触发 */
if(0 != sw_gpio_eint_set_trigtype(upio_index, TRIG_EDGE_NEGATIVE))
    printk("set trig type failed\n");
/* 释放 gpio */
if(0 == req_status)
    gpio_free(upio_index);
```

7.3.3.4. 打开 PA0 中断的 enable 位

```
u32upio_index = GPIOA(0);
int  req_status;

/* 申请 gpio */
req_status = gpio_request(upio_index);
if(0 != req_status)
    printk("request gpio failed\n");
/* 设置 PA0 的 enable 位为 1 */
```

```
if(0 != sw_gpio_eint_set_enable(upio_index, 1))
    printk("set enable status failed\n");
/* 释放 gpio */
if(0 == req_status)
    gpio_free(upio_index);
```

7.3.4. Script 接口和 GPIO 接口合用

假设 sys_config.fex 中 card0_boot_para 配置如下:

```
[card0_boot_para]
card_ctrl          = 0
card_high_speed    = 1
card_line          = 4
sdc_d1             = port:PF0<2><1><default><default>
sdc_d0             = port:PF1<2><1><default><default>
sdc_clk            = port:PF2<2><1><default><default>
sdc_cmd          = port:PF3<2><1><default><default>
sdc_d3             = port:PF4<2><1><default><default>
sdc_d2             = port:PF5<2><1><default><default>
```

7.3.4.1. card0_boot_para 主键下的所有 gpio 信息

```
    int  cnt, i;
    script_item_u *list = NULL;

    /* 获取 gpio list */
    cnt = script_get_pio_list("card0_boot_para", &list);
    if(0 == cnt) {
        printk("get card0_boot_para gpio list failed\n");
        return;
    }
    /* 申请 gpio */
    for(i = 0; i < cnt; i++)
        if(0 != gpio_request(list[i].gpio.gpio, NULL))
            goto end;
    /* 配置 gpio list */
    if(0 != sw_gpio_setall_range(&list[0].gpio, cnt))
        printk("sw_gpio_setall_range failed\n");
end:
    /* 释放 gpio */
    while(i--)
        gpio_free(list[i].gpio.gpio);
```

7.3.4.2. sdc_cmd 子键的 gpio 信息

```
    int  req_status;
    script_item_u     item;
    script_item_value_type_e     type;

    /* 获取 gpio list */
    type = script_get_item("card0_boot_para", "sdc_cmd", &item);
    if(SCIRPT_ITEM_VALUE_TYPE_PIO != type) {
        printk("script_get_item return type err\n");
        return;
    }
    /* 申请 gpio */
    req_status = gpio_request(item.gpio.gpio, NULL);
    if(0 != req_status)
        printk("request gpio failed\n");
    /* 配置 gpio */
    if(0 != sw_gpio_setall_range(&item.gpio, 1))
        printk("sw_gpio_setall_range failed\n");
end:
    /* 释放 gpio */
    if(0 == req_status)
        gpio_free(item.gpio.gpio);
```

7.3.4.3. AXP GPIO 的配置

目前 gpio 驱动支持 axp pin，但只能通过标准 gpio 接口．换言之，对于 axp pin 不允许调用多功能配置接口(比如 sw_gpio_setall_range)和 gpio 中断接口．
因此不能通过上面方法来配置 axp pin.. 正确的做法为:
(1) 从 sys_config.fex 中解析出 axp pin 的配置信息
(2) 通过标准 gpio 接口进行配置.

比如配置 sys_config.fex 中 lcd0_para 主键下 lcd_power 子键对应的 axp gpio 信息:

```
[lcd0_para]
lcd_power                       = port:power1<1><0><default><1>
```

```
    int  pio_index;
    int  req_status;
    script_item_u     item;
    script_item_value_type_e     type;
    /* 获取 gpio list */
```

```
    type = script_get_item("lcd0_para", "lcd_power", &item);
    if(SCIRPT_ITEM_VALUE_TYPE_PIO != type) {
        printk("script_get_item return type err\n");
        return;
    }
    /* 申请 gpio */
    pio_index = item.gpio.gpio;
    req_status = gpio_request(pio_index, NULL);
    if(0 != req_status)
        printk("request gpio failed\n");
    /* 配置 gpio */
    if(0 == item.gpio.mul_sel) { /* 输入 */
        if(0 != gpio_direction_input(pio_index))
            printk("gpio_direction_input failed\n");
    } else if(1 == item.gpio.mul_sel) {   /* 输出 */
        if(0 != gpio_direction_output(pio_index, item.gpio.data))
            printk("gpio_direction_output failed\n");
    } else
        printk("invalid sys_config, axp pin can only be input/output\n");
end:
    /* 释放 gpio */
    if(0 == req_status)
        gpio_free(pio_index);
```

7.4. 如何得到 GPIO 编号

GPIO 模块的导出 api 都需要 GPIO 编号作为参数. 每个 GPIO 对应唯一的全局编号, 由标准 GPIO 模块统一管理.

7.4.1. 已知 GPIO 名称

arch/arm/mach-sun7i/include/mach/gpio.h 定义了如下宏:

```
#define GPIOA(n)        (PA_NR_BASE + (n))
#define GPIOB(n)        (PB_NR_BASE + (n))
#define GPIOC(n)        (PC_NR_BASE + (n))
#define GPIOD(n)        (PD_NR_BASE + (n))
#define GPIOE(n)        (PE_NR_BASE + (n))
#define GPIOF(n)        (PF_NR_BASE + (n))
#define GPIOG(n)        (PG_NR_BASE + (n))
#define GPIOH(n)        (PH_NR_BASE + (n))
#define GPIOL(n)        (PL_NR_BASE + (n))
```

```
#define GPIOM(n)        (PM_NR_BASE + (n))
#define GPIO_AXP(n)        (AXP_NR_BASE + (n))
```

比如 PA0, PG(10), PH(28)的编号分别为 GPIOA(0), GPIOG(10), GPIOH(28).
Axp 的 0 号, 1 号 pin 编号分别为 GPIO_AXP(0), GPIO_AXP(1).

7.4.2. sys_config 中的 GPIO 编号

通过 script 接口 script_get_item 和 script_get_pio_list 获取到的 script_item_u 中包含了 GPIO 编号值.

```
/*
 * define data structure script item
 * @val: integer value for integer type item
 * @str: string pointer for sting type item
 * @gpio: gpio config for gpio type item
 */
typedef union {
        int                     val;
        char                    *str;
        struct gpio_config      gpio;
} script_item_u;

/* gpio config info */
struct gpio_config {
    u32 gpio;           /* gpio global index, must be unique */
    u32     mul_sel;    /* multi sel val: 0 - input, 1 - output... */
    u32     pull;       /* pull val: 0 - pull up/down disable, 1 - pull up... */
    u32     drv_level;  /* driver level val: 0 - level 0, 1 - level 1... */
    u32 data;           /* data val: 0 - low, 1 - high, only vaild when mul_sel is
input/output */
};
```

8. Android 系统支持

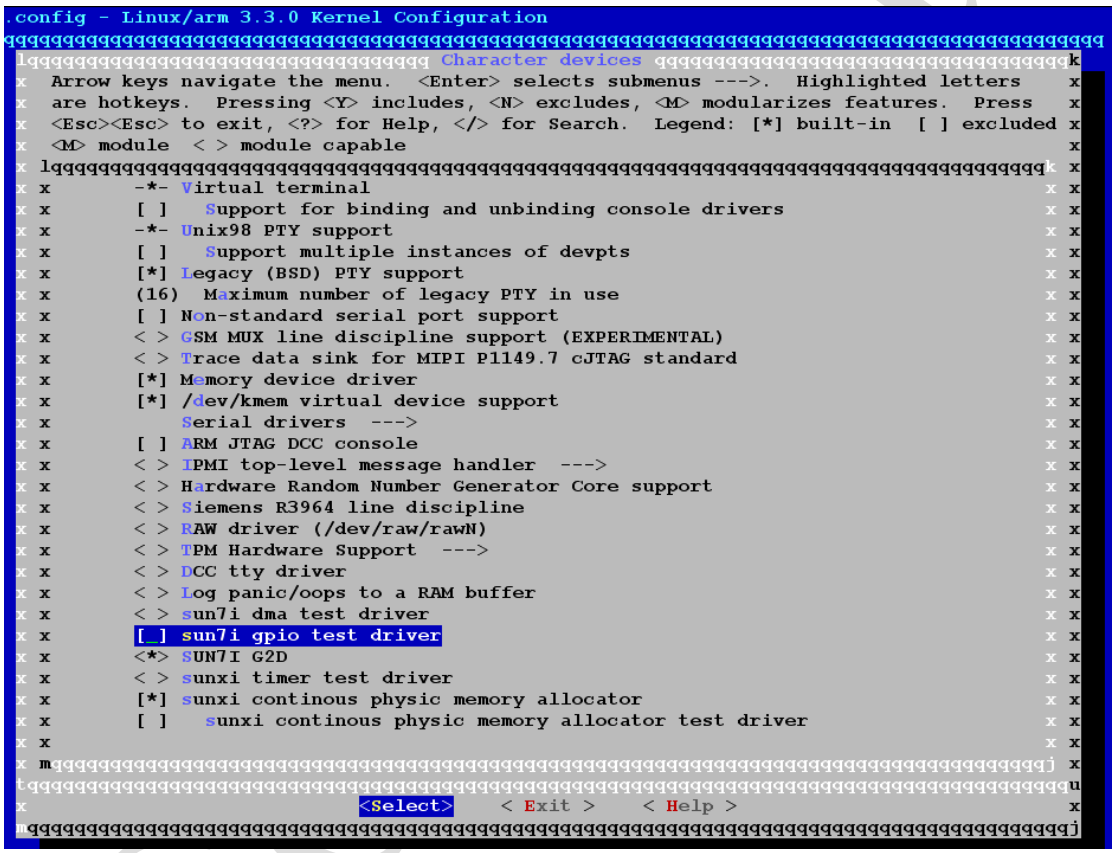
GPIO 属于 Linux 内核 buildin 模块，不直接与 android 相关.

9. 模块调试

9.1. menuconfig 配置

gpio 和 script 是 buildin 的模块，不用加载，调试方法是，在 menuconfig 中选择 gpio_test 驱动，然后编译 linux 镜像启动看打印．若打印 success 表明用例执行成功；打印 fail/err 表明失败．

menuconfig 的配置:
device drivers -> character devices -> sun7i gpio test driver:



9.2. 测试用例介绍

9.2.1. TEST_REQUEST_FREE

测试目的: 测试标准 gpio 的申请和释放接口.
系统配置: linux 系统启动.

操作步骤:

操作	期望结果
申请正常的 gpio	成功
申请无效的 gpio	失败，并有错误打印
释放已被申请的 gpio	成功

实际结果：与期望相符.

9.2.2. TEST_RE_REQUEST_FREE

测试目的：检查驱动能否正常处理重复申请和重复释放情形.
系统配置: linux 系统启动.
操作步骤:

操作	期望结果
释放一个未被申请的 gpio	失败，并有错误打印
申请正常的 gpio - PA	成功
释放 PA	成功
再次释放 PA	失败，并有错误打印
申请正常的 gpio - PB	成功
再次申请 PB	失败，并有错误打印
释放 PB	成功

实际结果：与期望相符.

9.2.3. TEST_GPIOLIB_API

测试目的：测试标准 gpio 接口.
系统配置: linux 系统启动.
操作步骤:

操作	期望结果
申请 gpio - PC1，并设为 output，电平为 high.	成功
获取 PC1 电平	返回 high(1).
申请 gpio - PC1，并设为 output，电平为 low.	成功
获取 PC1 电平	返回 low(0).
申请一组 gpio，并初始化其电平. gpio_request_array	成功
调用 sw_gpio_getall_range，获取这一组 gpio 的状态	打印信息与 gpio_request_array 设置的相符
释放这一组 gpio. gpio_free_array	成功
测试 gpiochip_find 接口: (1) 申请 PB5	成功找到 PB5 对应 gpio_chip 指针; gpiochip_is_requested 返回 true，即检

(2) 调用 gpiochip_find，找到 PB5 对应 gpio_chip 指针. (3) 调用 gpiochip_is_requested，查看 PB5 是否已被申请. (4) 释放 PB5.	测到 PB5 已被申请.
测试 gpio_direction_output, __gpio_get_value, __gpio_set_value, gpio_set_value_cansleep 接口. 请查看测试代码.	成功，与预期相符.

实际结果: 与期望相符.

9.2.4. TEST_CONFIG_API

测试目的: 测试多功能配置接口.
系统配置: linux 系统启动.
操作步骤/期望结果: 请参考测试代码.
实际结果: 与期望相符.

9.2.5. TEST_GPIO_EINT_API

测试目的: 测试 gpio 中断接口.
系统配置: linux 系统启动.
操作步骤/期望结果: 请参考测试代码.
实际结果: 与期望相符.

9.2.6. TEST_GPIO_SCRIPT_API

测试目的: 测试 gpio 脚本解析接口.
系统配置: linux 系统启动.
操作步骤/期望结果: 请参考测试代码.
实际结果: 与期望相符.

10. 总结

本文介绍了 GPIO 和 Script 框架和使用说明，给相关人员提供参考.

11. Declaration

This(A20 Script 和 GPIO 开发说明) is the original work and copyrighted property of Allwinner Technology ("Allwinner"). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner.

The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.