配置系统与 GPIO 使用文档

本文档详细介绍了配置系统与 GPIO 管理的目的，概念，以及其中的函数接口。

Version	*Data*	*Author*	*Change Description*
1.00	2011-4-22	wangflord < wangflord@allwinnertech.com >	原始版本
1.01	2011-4-22	wangflord < wangflord@allwinnertech.com >	增加配置系统工作流程
1.02	2011-10-19	Panglong <panlong@allwinnertech.com	函数名称更新并且添加

目录

一. 设计目的

本手册提供了配置系统与 GPIO 管理中的一些概念，以及可能使用到的函数接口，并在其中给出了实际范例，希望可以通过阅读这个手册，来解决实际使用中遇到的，和配置系统或 GPIO 管理相关的问题。

同时，配置系统，作为一个通用的软件平台，还希望通过它，可以适应用户不同的方案。通过给出一个对应的配置，用户的方案就可以自动运行，而不需要修改系统里面的代码，或者重新给出参数。

本文的阅读对象应该是使用到配置系统和 GPIO 管理的任何人员。对于系统来说，有系统编写和维护人员，驱动，模块的编写与维护人员。对于使用者来说，只要需要修改配置脚本，都应该阅读这个文档。本文也适用于对配置系统有兴趣的人员。

二. 相关概念解释

在本手册中出现了不少名词，用户描述配置脚本与 GPIO 管理。在这里对这些概念做一个统一的介绍。

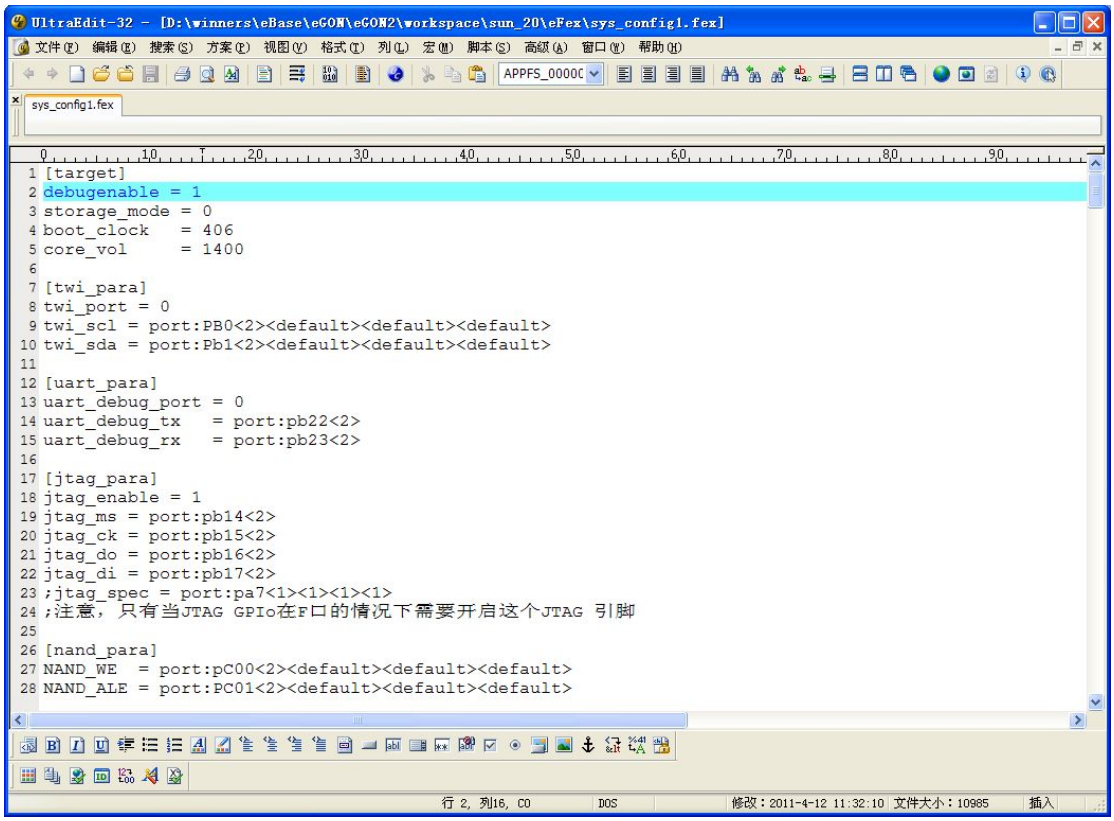
2.1 配置脚本

配置系统，在 PC 上体现为一个脚本文件(以 ini 作为后缀)。文件的内容以如下的形式存在，如图 1。

2.2 主键

在图 1 中，中括号括起来的字符串，例如[target]，[twi_para]，等等，中括号里面的字符串，如 target，twi_para 都被称为主键。主键的长度最大可以达到 32 个字符长度，不计结束符。在一个配置脚本中，主键的名称不允许相同。

2.3 子键



图一　配置脚本示例

在主键下方，是主键的成员，称为子键。如，[target]下方有四个条目，分别是

Debugenable = 1

Storage_mode = 0

Boot_clock = 406

Core_vol = 1400

这四个条目中，等号左边的字符串，都被称为主键 target 的子键。子键的长度最大可以达到 32 个字符，不计结束符。

在同一个主键范围内，子键的名称不允许相同。在脚本范围内，不同主键下的子键名称可以相同。

2.4 子键的值

有三种形式。

在图 1 中，主键 target 的子键的值是整型数据，例如 1,0,406，等等。可以是十六进制，不能是表达式。

主键 twi_para 中，子键 twi_port 的值是 0，twi_scl 的值是以如下的形式出现的：

twi_scl = port:PB0<2><default><default><default>

它表示，主键 twi_para 的子键 twi_scl 的值是一个 GPIO 描述信息，因为等号后方有一个“port:”前缀。具体描述信息参见下方。

子键还有一种值的形式是字符串。字符串的书写形式如下：

String_test = string:1234567890

表示，子键 string_test 的值是一个字符串，因为等号后方有一个 “string：”前缀，后面的就是这些字符串。这个字符串是“1234567890”。字符串最长的个数不超过 128 字节。

2.5 GPIO 描述信息

在硬件中，每个 GPIO 的控制都有如下的信息，

1） 端口分组，比如 PORTA，PORTB，等等，每组都包括了个数不等的 GPIO。不同的 IC 的同一组的 GPIO 可能不同，存在的组的个数也可能不同。

2） 组内序号。组内序号是指一个 GPIO 在一组端口组内的排序。比如 PORTA 的第 22 脚，22 就是其组内序号。每组的序号都是从 0 开始算。通常说的 PB3，就是指端口 PORTB 的第 3

个引脚，按照从 0 算起的规则，实际排序时第 4 个。使用的时候，不用关心从 0 算起的规则，按照电路图上给出的 IC 引脚直接使用即可。

在主键 twi_para 中，可以看到，twi_scl 使用的是端口 PORTB 的第 0 个引脚；twi_sda 使用的是端口 PORTB 的第 1 个引脚。

3) 功能分配。功能分配时 GPIO 的一个非常重要的属性，它标示着当前 GPIO 被什么模块所使用。具体的数值需要查阅 SPCE，或者询问 SD 人员。

在主键 twi_para 中，twi_para 的两个子键都被分配了功能 2，即 PB0 后面紧挨着的尖括号中的数值，这个数值是根据 SPCE 给出的。

4) 内置电阻。内置电阻是 IC 提供给引脚的电路属性，也就是通常所说的上拉电阻，或者下拉电阻。通常使用的是上拉电阻，也就是默认状态。

在 GPIO 描述中，紧挨着功能分配信息后的尖括号，即第 2 个尖括号描述了这个信息。使用 0 的话，标示内部电阻高阻态，如果是 1 则是内部电阻上拉，2 就代表内部电阻下拉。使用 default 的话代表默认状态，即电阻上拉。其它数据无效。

在 twi_para 中，twi_scl 的第一个 default 代表内部电阻状态，使用的是 default，即是 1，内部电阻上拉。

5) 驱动能力。驱动能力表示提供给当前 GPIO 的能力水平，这个数值越大，在 IO 口上的电平变化将约陡峭。通常，使用默认值 default 即可。

GPIO 描述中，紧挨着内置电阻信息的尖括号描述了这个信息，即 GPIO 中的第 3 个尖括号。括号内的值是 0，代表驱动能力等级为 0；括号内的值是 1，代表驱动能力等级为 1；括号内的值是 2，代表驱动能力等级为 2；括号内的值是 3，代表驱动能力等级为 3。其它数据无效。

在 twi_para 中，SCL 和 SDA 都是用了 default 来描述驱动能力，则实际给出的驱动能力是等级 1。

6) GPIO 描述信息的最后一项表示当前 GPIO 的输出电平。当一个 GPIO 作为 IO 口使用的时候，即功能分配时 1 的时候，这个信息可以设置当前 IO 口输出的电平状态。如果是 0，表示电平是低；如果是 1，表示电平是高。

如果功能分配不是 1，在这个电平描述信息不起作用。

总结以上的信息，还是以 twi_para 为例，

twi_scl = port:PB0<2><default><default><default> （1）

twi_scl = port:PB0<2><default><default> （2）

twi_scl = port:PB0<2><default> （3）

twi_scl = port:PB0<2> （4）

在描述一个 GPIO 信息的时候，可以看到以上的四种形式，这四种形式的意思都是一样的，都表示：

端口 B 的第 0 脚，功能分配时 2，内置电阻默认(1，上拉)，驱动能力默认(1，等级 1)，输出电平默认，由于分配的是 TWI 功能，非输出状态，输出电平无效。

描述一个 GPIO，形式是：

Port:端口+组内序号<功能分配><内部电阻状态><驱动能力><输出电平状态>

上面的条目(1)中，给出的是一个完整的 GPIO 描述信息；

条目(2)中，给出的是省略掉输出电平的信息，用于 GPIO 的功能分配不是输出状态的情况，如果 GPIO 设置成输出状态，则 IO 口的电平将保持当前状态不变；

条目(3)中，给出的是省略掉驱动能力，输出电平的情况，则驱动能力按照等级 1 设置，输出电平保持不变；

条目(4)中，给出的是省略掉内置电阻，驱动能力，输出电平的情况，则按照内置电阻上拉，驱动能力等级 1，输出电平保持不变的情况设置 GPIO 信息。

2.6 修改配置脚本

配置脚本作为一个文件存在于 PC 上，可以很方便的被用户所修改，删除或者添加。用户可以使用任意文件编辑工具，打开需要修改的配置脚本，达到自己的目的。

用户可以进行如下的修改

1) 修改主键的名称，比如在图 1 中，修改 [target] 为 [targetX]

2) 修改主键下的子键的名称，比如修改主键 target 的子键 boot_clock 为 boot0_clock

3) 修改主键下的子键的值，

比如修改主键 target 的子键 boot_clock 的值从 406 变成 384

比如修改主键 twi_para 的子键 twi_sda = port:pb0<2><2><2><1>

4) 删除一个主键

删除一个主键必须把主键所有的子键全部删除，比如要删除 target，除了删除了[target]本身外，还必须删除从属于它的子键 debugenable，storage_mode，boot_clock，core_vol 以及它们的值。

5) 删除一个子键

删除一个子键，需要把子键名称和它的值一同删除

6) 增加一个主键

一个主键可以是空，即没有从属于它的子键。

比如，可以列出一个[empty]的主键，它的下方是另一个主键[target]。这样，empty 就是一个为空的主键。

7) 增加一个子键

增加一个子键的时候，必须确定这个子键属于哪一个主键。一个子键不可能单独存在。当一个子键在脚本中出现的时候，它就从属于它上方最近的一个主键。

增加一个子键的时候，必须同时给出主键的值，如果实在不能确定，也可以给出一个为 0 的值。

8) 调整子键的位置

调整一个子键可以在主键内部调整，但是这样调整没有意义；也可以把一个子键放到另一个主键的范围内，这样这个子键将具有不同的意义。具体的含义由脚本定义者来决定。

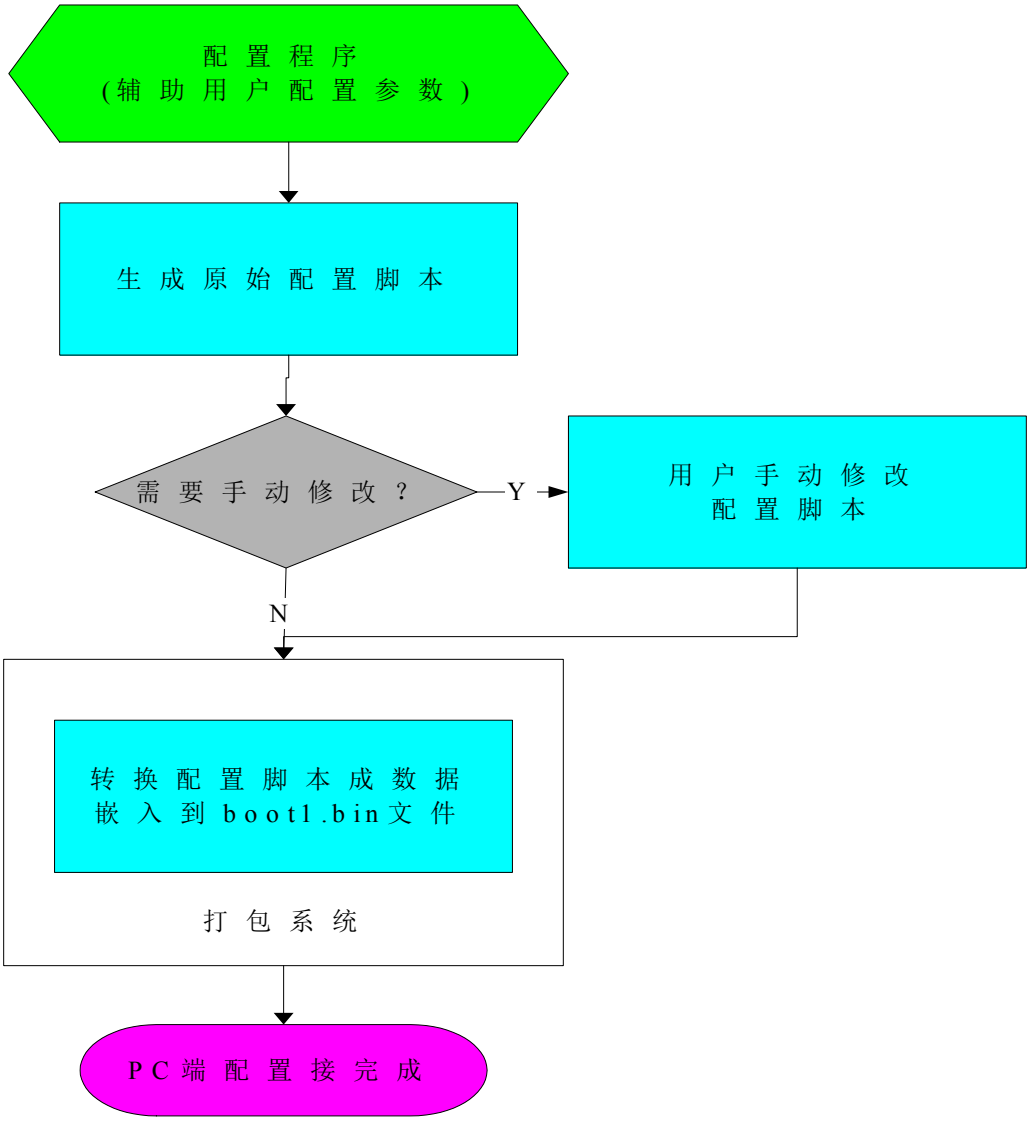
在脚本中，各个主键的排列顺序没有意义，即排在前方或者排在后方都一样。

三．配置系统工作流程与用途

本章将介绍配置脚本如何从 PC 端一步步的到达小机端，以及用户在使用配置脚本的时候，数据如何在系统中传递。

3.1 PC 端配置数据的生成

配置脚本本质上是 PC 端的一个文本文件，通过一个固定的格式形成可以被我们使用的文件，里面保存了大量的配置信息。在图二中，可以看到，PC 端的一个数据文件如何变成了小机端可以用到的文件。

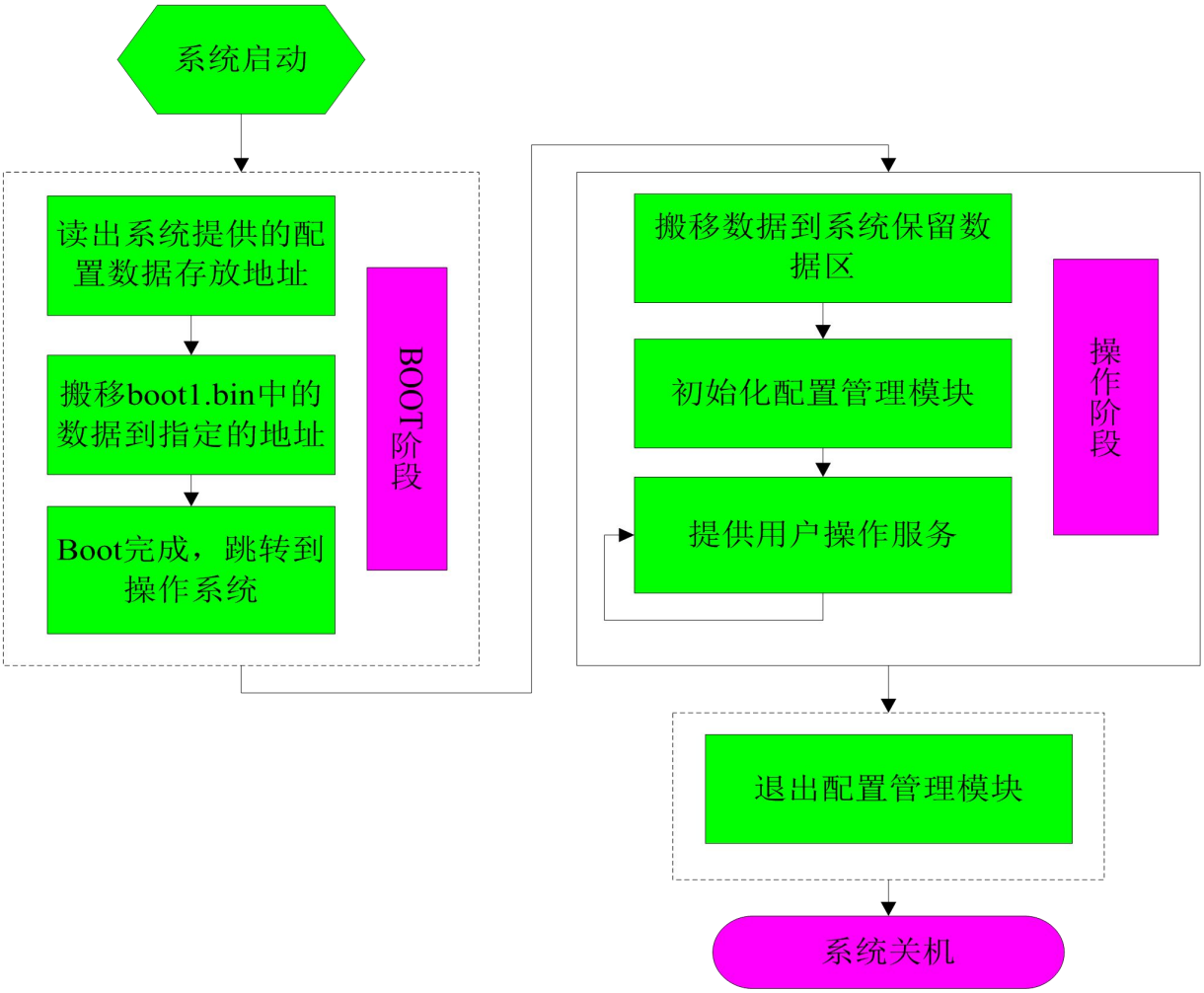


图二 小机端配置文件生成

图二中可以看出，当用户生成一个配置文件之后，不需要做额外的操作，只要按照正常的打包，烧写过程，配置文件的数据就自动被嵌入到 boot 相关的数据中了。

3.2 系统启动的数据传递

在小机端，系统启动之后存在数据传递的过程，这个过程主要是数据从 boot 中读出，然后存放到操作系统指定的位置。然后操作系统可以自己搬移这块数据，或者直接使用这块和配置有关的数据。相关的处理过程可以参见图三。



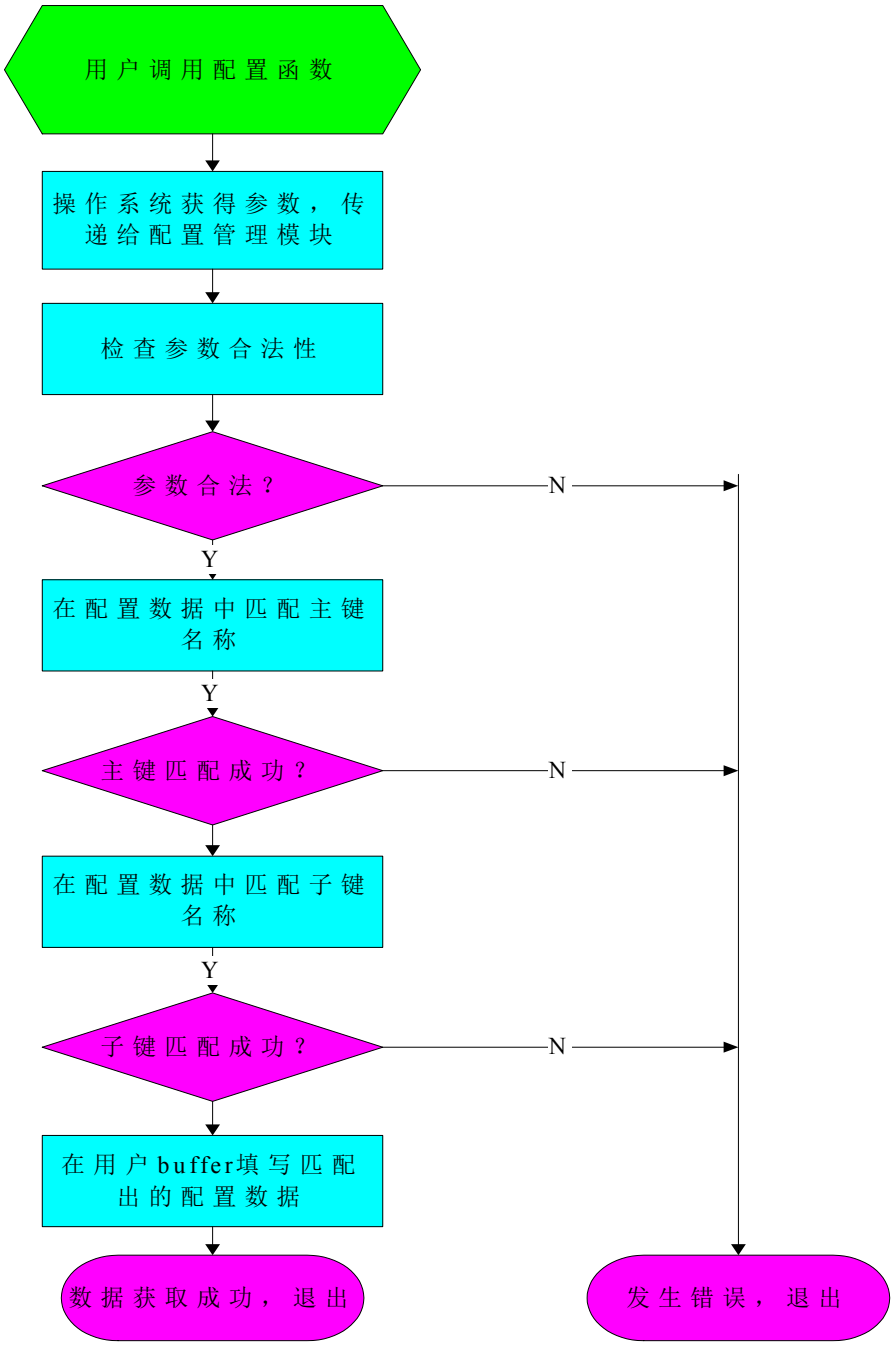
图三　配置系统在系统中的流程

从图三中可以看出，boot 阶段把数据从 boot1.bin 中读出，然后传递给了操作系统。操作系统拿到数据之后，做一次初始化动作，然后就一直等待用户进行操作。当系统关机的时候，操

作系统需要调用一次配置管理的退出函数，然后，整个配置系统的运行就结束。

3.3 用户调用配置系统的数据传递

当用户调用配置系统的时候，里面存在数据传递。图四表示了用户的数据如何传递到系统，以及系统如何做出相应的。



图四　配置系统使用中数据传递流程

通过图四，用户可以看出，当调用配置相关的函数的时候，系统中以及配置管理模块如何

管理用户传入的数据。

3.4 配置系统的用途

3.4.1 用途综述

配置脚本的本意是给系统传递参数。作为一个稳定的系统，本身应该和方案无关，不管不同方案的差别有多大，系统都不应该重新编译才能运行。这里所说的系统，不单单指操作系统，也包括其中的驱动，模块，等等。

不同方案的差别，通常体现在：

1) 使用的硬件模块不同，比如使用了不同的 NAND FLASH，RTC 模块，等等。

2) 相同模块使用的参数不同，其中包括 GPIO 不同，比如卡检测脚不同，SPI 的引脚不同，等等；包括执行的频率不同，如 DRAM 的频率，CPU 的频率，等等。

3) 走线的方式不同。

4) 没有列举出的差别。

通常说来，上面列举的方案差别中，(3)和(4)对于一个系统来说没有任何差别，只有(1)和(2)，才可能导致一个系统需要重新编译。

3.4.2 解决硬件模块不同的问题

在差别(1)中，有的需要代码做自适应，比如可以自动扫描出 NAND 的型号，这样就可以避免 FLASH 驱动重新编译；有的无法扫描出硬件模块型号，就需要通过配置来做管理。

以 RTC 为例，不同的方案中，有可能使用 IC 内部的 RTC 模块，有可能使用外部的 RTC，而外部的 RTC 又有不同的型号。在具体的做法中，就可以使用配置系统来管理 RTC。首先，需要在 RTC 驱动中给出支持的所有 RTC 的型号，并给出相应的代码。然后，在配置中给出配置项目，指明配置和 RTC 的对应关系，比如给出了如下的关系：

配置 0 对应 IC 内部 RTC

配置 1 对应外部 RTC 型号 XXXX

配置 2 对应外部 RTC 型号 YYYY(假设驱动已经给出了内部 RTC，以及外挂的 XXXX，YYYY 的驱动程序)

在配置脚本中，要做的是给出一个主键项，名称为(这个名称可以随意，但是建议可以让人

看到名字能猜出功能)

```
[rtc_select]
```

然后在主键项下面给出子键项目。

```
rtc_pattern = 0
```

顾名思义，看出配置的意思是，第一个子键代码使用了 IC 的内部 RTC，然后 RTC 的型号是 0。或者，子键项目如下：

```
rtc_pattern = 2
```

这个配置表示，使用了外部 RTC，RTC 的型号是 YYYY。

有了这样的配置，用户就可以很方便的使用不同的 RTC 了，只需要在配置脚本中改变 RTC 子键的值，就能够使用驱动中支持的 RTC(所有支持的 RTC 型号以及对应的配置值可以事先知道，比如通过说明书等等)。

事实上，RTC 驱动内部使用了如下的函数来使用这里的配置：

```
{
    int    rtc_pattern;
    int    ret;

    ret = Script_parser_fetch("rtc_select", "rtc_pattern", &rtc_pattern, 1);
    if(ret >= 0)
        return rtc_pattern;
    else
        return ret;
}
```

通过这个函数，RTC 驱动可以获取到用户设置的 RTC 的型号，只要用户的配置和方案的一致，就可以让 RTC 正常工作。这种做法避免了改变 RTC 型号就需要重新生成或者替换 RTC 驱动的弊端，只需要一个 SDK 就适应不同 RTC 模块。

3.4.3 解决参数不同的问题

在不同方案中，一定存在参数不同的问题。这次以卡检测为例，通常使用了一个 GPIO 作为卡的检测引脚，当电平为低的时候，认为卡插入；当电平为高的时候，认为没有卡插入。在不同的方案中，使用的卡检测引脚有可能不同，这样，就需要一个配置项来告诉系统，当前方案使用的卡检测脚是哪一个，而并不需要更换一个适应当前卡检测脚的检测模块。

在配置脚本中，有如下的项目

```
[sddet_para]
detect_pin = port:PI4<0><1><1>
```

主键 sddet_para 表示这是和卡检测相关的参数，主键名称是模块编写者给出，可以通过名字猜测相关信息。

子键 detect_pin 的值表示了检测引脚是 IC 的哪一个，从值可以看出，当前方案的检测引脚是端口 I 的第 4 脚，分配了功能为输入，内置电阻上拉，驱动能力等级 1。

如果用户需要更换卡检测引脚，则只需要更新这里即可，比如换成了端口 A 的第 14 脚，就写成如下的方式：

```
[sddet_para]
detect_pin = port:PA14<0><1><1>
```

通过以上的方式可以看出，使用了配置系统后，用户可以很方便的使用新的方案板，只要在配置中正确的修改数值，就可以让一块刚刚设计出来的放案板正常工作，省掉了大量更换驱动的工作，并且，更换驱动的工作也容易出错。

3.4.4 用户自定义参数举例

前面讲的，都是关于配置脚本中已有的配置项目，用户只需要根据说明正确修改配置项目即可。由于用户也可以根据自己的需要修改配置脚本，这里就给出范例，讲述如何使用用户自行添加的配置项目。

比如，用户希望使用配置来管理 FM 模块(尽管通常不会这么干)，就要做如下的事情。

首先，在配置中给出如下的配置项目：

```
[fm_para]
fm_pattern = 0
```

在 FM 的类型中，应该已有 fm_type 的值和具体 FM 型号对应的关系，如 0 代表 RDA5820，1 代表 TEA5767，2 代表 QN8006，等等。当前的配置是 0，表示使用的 FM 型号是 RDA5820。

在 FM 驱动中，应该有如下的代码来获取用户配置的 FM 型号。这样，驱动就知道当前使用的 FM 芯片是 RDA5820，然后在程序执行的时候就能够找到对应的代码，让 FM 驱动正常工作。

这样的好处是通过同一个驱动，可以管理不同的 FM 型号，省掉了更换驱动的麻烦。

```
{
    int    fm_pattern;
    int    ret;

    ret = Script_parser_fetch("fm_para", "fm_pattern", &fm_pattern, 1);
    if(ret >= 0)
        return fm_pattern;
    else
        return ret;
}
```

四. 使用配置脚本

这一节中，将介绍脚本的使用方法。

在系统中，提供了如下的几个函数，提供给用户在系统中读取配置信息的数据。

4.1 获取配置参数

函数原型： int Script_parser_fetch(char *main_name, char *sub_name, int value[], int count);

参数：main_name 主键名称，即配置脚本中的主键名称，字符串形式

sub_name 子键名称，配置脚本中的子键名称，字符串形式

value 数据指针，用于存放用户获取的数据

count 用户传进的数据空间的最大 word 个数

返回值： 成功返回 0

失败返回-1

这个函数的功能很强大，可以获取配置脚本中任意一项的值。

比如，用户需要获取配置脚本中，主键 target 下的子键 boot_clock 的值，可以写成

```
{
    int    value;
    int    ret;

    ret = Script_parser_fetch("target", "boot_clock", &value, 1);
    if(ret < 0)
        printf("fetch script data fail\n");
    else
        printf("fetch script data ok, value = %d\n", value);

    return ret;
}
```

在这个函数中，获取到的值存放在整型变量 value 中，正常情况下，函数调用的结果是 value = 406

如果要获取一个配置的 GPIO 信息，比如 twi_para 的 twi_scl 可以使用如下的形式

```
{
    user_gpio_set_t    gpio_info[1];
    int    ret;

    ret = Script_parser_fetch("twi_para", "twi_scl", gpio_info, sizeof(user_gpio_set_t)/sizeof(int));
    if(ret < 0)
        printf("fetch script gpio infomation fail\n");
    else
        printf("fetch script gpio infomation ok \n");

    return ret;
}
```

这个函数将把获取到的 GPIO 信息存放到结构体 gpio_info 中。用户可以使用这个结果，来调用 GPIO 管理模块提供的函数。

用户也可以使用脚本函数来获取一个字符串。

比如，存在如下的一个主键和子键项目

[string_test]

string_demo = string:abcdefghijklmn

现在，可以用这个函数来获取出主键 string_test 的子键 string_demo 的值。正常情况下，调用如下的函数之后，string_info 中保存的值将是“abcdefghijklmn”(没有引号)。

```
{
    char    string_info[128];
    int    ret;

    memset(string_info, 0, 128);
    ret = Script_parser_fetch("string_test", "string_demo", string_info, 128/sizeof(int));
    if(ret < 0)
        printf("fetch script string infomation fail\n");
    else
        printf("fetch script string infomation ok \n");

    return ret;
}
```

4.2 获取子键个数

函数原型：int Script_parser_subkey_count(char *main_name);

参数： main_name 主键名称，即配置脚本中的主键名称，字符串形式

返回值： 成功返回 主键下的子键个数

失败返回 -1

这个函数返回的是一个主键下所有的子键的个数，通常用户不会关心它。这个函数更大的用途还在于做检查。

```
{
    int    sub_key_count;

    sub_key_count = Script_parser_subkey_count ("target");
    if(sub_key_count < 0)
        printf("fetch script sub key count fail\n");
    else
        printf("fetch script sub key count ok , sub_key_count = %d\n",
sub_key_count);

    return sub_key_count;
}
```

调用如上的函数，将获取到主键 target 下的所有子键的个数，即得到数值 4。

4.3 获取主键个数

函数原型：int Script_parser_mainkey_count(void);

参数：无

返回值：　成功返回 配置脚本中主键的总的个数

　　失败返回 -1

这个函数将获取所有主键的个数，和 Script_parser_subkey_count 一样，主要用途还是做检查使用。

```
{
    int    main_key_count;

    main_key_count = Script_parser_mainkey_count();
    if(main_key_count < 0)
        printf("fetch script sub key count fail\n");
    else
        printf("fetch script main key count ok , main_key_count = %d\n",
main_key_count);

    return main_key_count;
}
```

调用如上的函数，将获取到配置脚本中主键的个数。

4.4 获取主键下 GPIO 个数

函数原型：int Script_parser_mainkey_get_gpio_count(char *main_name);

参数：main_name　配置脚本中主键的名称，字符串形式

返回值：　成功返回 配置脚本中主键下的，数据 GPIO 类型的子键个数

失败返回 -1

这个函数的调用将得到主键下的子键中，值属于 GPIO 类型的子键个数。

比如，当获取 twi_para 下的子键中的 GPIO 类型时，将获取到数值 2。

```
    {
        int    gpio_key_count;

        gpio_key_count = Script_parser_mainkey_get_gpio_count (“twi_para”);
        if(gpio_key_count < 0)
            printf(“fetch script sub key count fail\n”);
        else
            printf(“fetch script gpio key count ok , gpio_key_count = %d\n”,
gpio_key_count);

        return gpio_key_count;
    }
```

如果把上面函数的参数 twi_para 替换成 target，则得到的将是 0。如果把上面函数的参数 twi_para 替换成 nand_para，则得到的将是 23。

4.5 获取主键下 GPIO 配置

这个函数将获取一个主键下，所有属于 GPIO 的子键的 GPIO 描述值。

```
函数原型：int Script_parser_mainkey_get_gpio_cfg(char *main_name, void *gpio_cfg,
int gpio_count);
参数：main_name  主键名称，即配置脚本中的主键名称，字符串形式
        gpio_cfg     用于存放 GPIO 信息的地址，应该是属于 user_gpio_set_t 的数据
结构
        gpio_coumt  用户传进的结构体的个数
        返回值：      成功返回 0
                      失败返回-1
```

调用这个函数，将把配置脚本中匹配主键名称的，属于 GPIO 类型的子键的个数。

```
{
    user_gpio_set_t    gpio_info[2];
    int    ret;

    ret = Script_parser_mainkey_get_gpio_cfg(“twi_para”,gpio_info, 2);
    if(ret < 0)
        printf(“fetch script gpio infomation fail\n”);
    else
        printf(“fetch script gpio infomation ok \n”);

    return ret;
}
```

调用这个函数，将获取配置脚本里，twi_para 的子键中，属于 GPIO 类型的描述信息。

五．使用 GPIO 管理

GPIO 配置用于用户定义于脚本中的 GPIO 的使用方法。当用户在配置脚本中对于某一个或者多个子键定义了 GPIO 数据类型，现在可以通过系统提供的一整套函数，把这个配置中的 GPIO 转化到实际的 GPIO 控制上。

5.1 申请 GPIO

这个函数用于申请操作 GPIO 的句柄。通过这个函数，可以设置一个 GPIO 的属性或者是一组 GPIO 中的某个或者多个的属性。

使用这个函数，需要配合配置脚本使用，或者是用户自己定义一个 GPIO。

函数原型：__hdle Gpio_request (user_gpio_set_t *gpio_list,
unsigned group_count_max);
参数：gpio_list 数据地址，保存 GPIO 属性，来自于配置脚本或者是用户自定义
group_count_max 用户保存 GPIO 数据的结构体的最大个数
返回值： 成功返回非空指针

user_gpio_set_t 是一个如下的数据结构

```
typedef struct
{
    char    gpio_name[32];          //代表当前 GPIO 的名称
    int port;                       //GPIO 使用的端口号
    int port_num;                   //GPIO 在当前端口的序号
    int mul_sel;                    //GPIO 的功能选择
    int pull;                       //GPIO 的内置电阻状态
    int drv_level;                  //GPIO 的驱动能力
    int data;                       //GPIO 的电平
}
user_gpio_set_t;
```

这个函数一旦被调用之后，所有的配置将立刻生效。

要配置 GPIO 的方法，通常有如下三种。

5.1.1 通过主键查找申请 GPIO

GPIO 信息来源于配置脚本，只根据配置中主键进行查找，然后获取主键下的所有 GPIO，然后进行配置。主键下的 GPIO 可以是 1 个，也可以是多个。如果主键下没有 GPIO，则获取的时候就会失败。这种做法很简单，举例如下。

```
{
    user_gpio_set_t    gpio_info[2];
    int    ret;
    __hdle      gpio_hd;

    ret = Script_parser_mainkey_get_gpio_cfg(“twi_para”, (void *)gpio_info, 2);
    if(ret < 0)
    {
        printf(“fetch script gpio infomation fail\n”);
        return -1;
    }
    gpio_hd = Gpio_request(gpio_info, 2);
    return gpio_hd;
}
```

上述的实例代码中，首先调用一个获取 GPIO 配置的函数 Script_parser_mainkey_get_gpio_cfg，利用传进的参数“twi_para”，查找配置脚本中的名称为 twi_para 的主键下的所有 GPIO 属性的子键。查找出的 GPIO 配置参数存放在结构体 gpio_info 中。所有的属性都按照各自的定义存放，其中，gpio_name 一项存放的是配置脚本中子键的名称。比如，twi_scl 对应的 GPIO 的名称就是 twi_scl，twi_sda 对应的 GPIO 的名称就是 twi_sda。

函数 Script_parser_mainkey_get_gpio_cfg 的第三个参数“2”表示只要获取 2 个 GPIO。用户对这个参数可以填写超过实际 GPIO 的整数。这个参数的意义在于，当用户不确定一个模块

到底使用了几个 GPIO 的时候，填写一个足够大的数值，可以保证所用到的 GPIO 一定被完整取出。如果这个数值比实际的 GPIO 个数要小，那么取出的 GPIO 个数和这个参数相等。这样，可以确保取 GPIO 个数的时候，不会因为取了多余的数值，导致用户的内存溢出。

接下来，调用了函数 Gpio_request，传进的第一个参数，就是从前面获取到的。第 2 个参数，和前面函数使用的保持一致即可。调用成功的话，会返回一个句柄给用户，否则，将只返回一个零值给用户。用户可以使用这个句柄，操作以后的函数。同时，twi_para 的配置也在硬件中生效了，即 TWI 的两个 GPIO 已经被配置成功。

5.1.2 通过主键加子键查找申请 GPIO

第二种方式，适用于只需要配置一个 GPIO 的情况。根据配置脚本中的主键名称以及子键名称匹配，来查找出 GPIO。这个方法只适用于需要获取一个 GPIO 的情况，如果主键下有多个 GPIO，可以调用多次函数来获取主键下的 GPIO 配置，然后会得到多个句柄。比如配置卡的检测引脚。

以下给出示例代码。

```
{
    user_gpio_set_t    gpio_info;
    int    ret;
    __hdle     gpio_hd;

    ret    =    Script_parser_fetch    (“twi_para”,    “twi_scl”,    (int    *)&gpio_info,
sizeof(user_gpio_set_t)/sizeof(int));
    if(ret < 0)
    {
        printf(“fetch script gpio infomation fail\n”);
        return -1;
    }
    gpio_hd = Gpio_request(&gpio_info, 1);
    return gpio_hd;
```

这个函数中，首先调用 Script_parser_fetch 来获取主键名称为 twi_para，子键名称为 twi_scl 的 GPIO 的全部配置信息。然后，获取到的信息存放在 gpio_info 这个结构体当中，然后告诉配置系统，当前要取出的数据的大小是 sizeof(user_gpio_set_t)/sizeof(int)。这样，按照图 1 给出的示例，在 gpio_info 中得到的数据就是

gpio_info.gpio_name = “twi_scl”

gpio_info.port = 2

gpio_info.port_num = 0

gpio_info.mul_sel = 2

gpio_info.pull = 1

gpio_info.drv_level = 1

gpio_info.data = 1

然后，调用函数 Gpio_request，第一个参数是前面获取到 GPIO 配置的结构体 gpio_info，第二个参数是 1，表示只有一个 GPIO 需要配置。这样，就可以拿到一个句柄，用于以后的操作。同样，twi_para 的 twi_scl 引脚也被配置完成。

5.1.3 用户自定义查找申请 GPIO

第三个方式适用于用户自定义的 GPIO 的使用。当用户需要使用一个 GPIO 的时候，这个 GPIO 可以是自己写代码来确定要使用哪个，而不是通过配置脚本。

比如，用户需要使用 GPIOA 的第 14 脚的输入输出功能，做一个功能控制，可以使用如下的方法。

首先定义一个结构体变量，user_gpio_set_t gpio_info，然后对这个变量赋值。默认处于输出状态，输出电平默认是高。gpio 的名称属性可以不写，也可以自己给 gpio 取名。

strcpy(gpio_info.gpio_name, “test_key_name”); //这个步骤不是必须的，属于可选

gpio_info.port = 1; //表示使用 PA 端口

gpio_info.port_num = 14; //表示使用第 14 引脚

gpio_info.mul_sel = 1; //表示使用输出功能，1 代表输出功能，0 代表输入功能

gpio_info.pull = 1; //表示内置电阻上拉，0 代表高阻，1 代表上拉，2 代表下拉

gpio_info.drv_level = 1; //驱动能力等级 1, 0-3 分别代表驱动能力等级 0-3

gpio_info.data = 1; //输出电平默认是高，0 代表输出低电平，1 代表高电平

变量赋值完成以后，直接调用函数即可，如下操作。

```
{
    __hdle     gpio_hd;

    gpio_hd = Gpio_request(&gpio_info, 1);
    return gpio_hd;
}
```

注意，示例代码里没有对变量 gpio_info 赋值，赋值的动作假定已经按照前面的说明完成。调用这个函数之后，将获取到对应于 PA14 操作的一个句柄，同时，PA14 也处于用户设定的，输出功能，内部电阻上拉，驱动能力 1，同时输出高电平的状态。

5.2 释放 GPIO

这个函数可以释放通过申请获取到的 GPIO 的句柄，并不关心是哪种方式获取到的 GPIO。

函数原型：unsigned Gpio_release(__hdle p_handler, unsigned if_release_to_default_status);

参数：p_handler GPIO 控制句柄，来自于 GPIO 的申请函数

if_release_to_default_status 控制 GPIO 释放后的状态。

返回值： 成功返回 0

通过这个函数，用户可以控制 GPIO 被释放之后，处于什么样的状态。

参数 if_release_to_default_status 的值可以是 0，1，2。如果是 0 或者 1，表示释放后的 GPIO 处于输入状态，输入状状态不会导致外部电平的错误。如果是 2，表示释放后的 GPIO 状态不变，即释放的时候不管理当前 GPIO 的硬件寄存器。

函数的使用非常简单，直接调用这个函数，传入一个合法的句柄，以及一个表示释放后状态的参数即可。用户需要自己判断返回值，来检查释放句柄是否成功，以及是否句柄时候传入

的参数是否正确。

5.3 获取句柄下所有 GPIO 配置

这个函数可以通过句柄，获取到句柄对应的 GPIO 的所有状态。

函数原型：Int Gpio_get_all_pin_status(__hdle devpin, user_gpio_set_t *gpio_status, unsigned gpio_count_max , unsigned if_get_from_hardware);

参数：devpin GPIO 控制句柄，来自于 GPIO 申请函数

gpio_status 数据地址，用于保存 GPIO 的配置属性

gpio_count_max 数据的最大个数

if_get_from_hardware 表示获取什么样的配置属性，0 表示获取用户最初配置的属性，1 表示获取当前硬件实际的配置属性

返回值： 成功返回 0

失败返回-1

通过调用这个函数，可以取出句柄所控制的所有 GPIO 的状态属性。参数列表中，gpio_status 将保存获取到的信息，并按照结构体的属性一项一项依次填写。对于非输入输出状态的 GPIO，在 data 项目将只能得到一个“-1”，表示这个项目没有意义。

gpio_count_max 表示这个结构体变量的个数。如果句柄中实际控制的 GPIO 个数超过 gpio_count_max 的值，则取出的 GPIO 个数只是这个变量的大小。如果 gpio_count_max 超过 GPIO 个数，则取出的 GPIO 个数将是实际个数。

if_get_from_hardware 表示获取的 GPIO 配置来源于何处。如果是 1，表示要取出句柄控制的 GPIO 的硬件控制器中的值；如果是 0，表示获取的值来源于用户申请的时候传入的 GPIO 配置参数。下面将以 twi_para 给出示例，表示获取用户传入的配置参数。

假定已经申请了配置脚本中 twi_para 的值，，对应的 GPIO 句柄是 gpio_hd，现在需要获取出 GPIO 配置信息，则给出如下的函数

```
{
    user_gpio_set_t    gpio_info[10];
    int    ret;

    ret = Gpio_get_all_pin_status(gpio_hd, gpio_info, 10, 0);
    if(ret < 0)
    {
        printf("fetch gpio infomation fail\n");
    }
    else
    {
        printf("fetch gpio infomation ok\n");
    }
    return ret;
}
```

在这个函数中，给出一个类型为 user_gpio_set_t，成员个数为 10 的数组 gpio_info。这里故意写了一个 10，表示空间足够大；实际使用中，如果知道 twi_para 的 GPIO 个数只有 2 个，则写 2 即可。如果不确定，可以像这个例子这样，写一个 10。

示例代码中，要获取最初用户配置的参数。通过这个函数，得到的结果就应该是，gpio_info[0]中保存了 twi_scl 的值，gpio_info[1]中保存了 twi_sda 的值，然后 gpio_info[2]-gpio_info[9]中的值全是 0。

gpio_info[0].gpio_name = "twi_scl"	gpio_info[1].gpio_name = "twi_sda"
gpio_info[0].port = 2	gpio_info[1].port = 2
gpio_info[0].port_num = 0	gpio_info[1].port_num = 1
gpio_info[0].mul_sel = 2	gpio_info[1].mul_sel = 2
gpio_info[0].pull = 1	gpio_info[1].pull = 1
gpio_info[0].drv_level = 1	gpio_info[1].drv_level = 1
gpio_info[0].data = -1	gpio_info[1].data = -1

下面，再给出一个示例，表示从硬件获取，然后空间不足的情况，还是以 twi_para 为例。

```
{
    user_gpio_set_t    gpio_info[1];
    int    ret;

    ret = Gpio_get_all_pin_status(gpio_hd, gpio_info, 1, 1);
    if(ret < 0)
    {
        printf("fetch gpio infomation fail\n");
    }
    else
    {
        printf("fetch gpio infomation ok\n");
    }
    return ret;
}
```

现在，结构体数组只有一个成员，明显不足 twi_para 中 2 个 GPIO 的数量，那么调用这个函数，将只能获取到一个 GPIO 的配置属性。同时，获取到的 GPIO 配置来源于当前硬件的实际参数。

由于在申请过后，没有对硬件进行操作，则看到的值不变，实际上，这些值来源于硬件控制器。

gpio_info[0].gpio_name = "twi_scl"

gpio_info[0].port = 2

gpio_info[0].port_num = 0

gpio_info[0].mul_sel = 2

gpio_info[0].pull = 1

gpio_info[0].drv_level = 1

gpio_info[0].data = -1

5.4 获取句柄下单个 GPIO 配置

Gpio_get_one_pin_status 和 Gpio_get_all_pin_status 比较类似。但是，顾名思义，Gpio_get_all_pin_status 获取到的是句柄控制下所有 GPIO 的属性，而 Gpio_get_one_pin_status 只能获取到一个 GPIO 的属性。

函数原型： Int Gpio_get_one_pin_status (__hdle devpin, user_gpio_set_t *gpio_status, const char *gpio_name, unsigned if_get_from_hardware);

参数：devpin GPIO 控制句柄，来自于 GPIO 申请函数

gpio_status 数据地址，用于保存 GPIO 的配置属性，这个地址被认为只有一个成员

gpio_name GPIO 的名称，来源于配置脚本

if_get_from_hardware 表示获取什么样的配置属性，0 表示获取用户最初配置的属性，1 表示获取当前硬件实际的配置属性

返回值： 成功返回 0

失败返回-1

调用这个函数，可以获取到 GPIO 句柄控制下的其中一个 GPIO。gpio_status 用于保存 GPIO 的配置信息，这些信息的来源由第四个参数 if_get_from_hardware 确定，如果是 0，表示获取于用户最初的配置；如果是 1，表示获取于实际的硬件寄存器。

gpio_name 表示 GPIO 的名称，用于匹配一组 GPIO 中的具体某一个。比如 twi_para 中的 twi_scl 脚，则在获取的时候 GPIO 名称就要填写 twi_scl。下面举例说明，要获取 twi_para 下的 twi_sda 引脚的配置，要求获取当前硬件的配置，假定句柄还是 gpio_hd。

```
{
    user_gpio_set_t    gpio_info[1];
    int    ret;

    ret = Gpio_get_one_pin_status(gpio_hd, gpio_info, “twi_sdl”, 1);
    if(ret < 0)
    {
        printf(“fetch gpio infomation fail\n”);
    }
    else
    {
        printf(“fetch gpio infomation ok\n”);
    }
    return ret;
}
```

调用这个函数，能获取出 gpio_hd 句柄控制下的，名称为 twi_sda 的 GPIO 的配置属性，同时，这个配置来源于实际硬件控制器。

5.5 设置句柄下单个 GPIO 配置

函数 Gpio_set_one_pin_status 和 Gpio_get_one_pin_status 是相反的关系。Gpio_get_one_pin_status 会读取 GPIO 的配置，而函数 Gpio_set_one_pin_status 会修改句柄控制下的某一个 GPIO 的配置。

函数原型： Int Gpio_set_one_pin_status (__hdle devpin, user_gpio_set_t *gpio_status, const char *gpio_name, unsigned if_set_to_current_input_status);

参数：devpin GPIO 控制句柄，来自于 GPIO 申请函数

gpio_status 数据地址，用于保存 GPIO 的配置属性，这个地址被认为只有一个成员

gpio_name GPIO 的名称，来源于配置脚本

if_set_to_current_input_status 设置的参数来源，0 表示按照用户最初配置的属性修改硬件控制器，1 表示按照当前用户传入的参数修改硬件控制器

返回值： 成功返回 0

失败返回-1

函数的参数列表中，要修改的 GPIO 的属性被指明于 gpio_name 一项，这和 Gpio_get_one_pin_status 一致。

修改的数据源来源于第四个参数 if_set_to_current_input_status，如果是 0，表示按照用户最初配置的属性进行修改，这时，第二个参数没有意义，可以不写；如果是 1，表示按照用户当前输入的属性进行修改，这时，也就是按照第二个参数的属性进行修改。

使用这个函数修改 GPIO 配置的时候，能修改的，只有 GPIO 的功能分配，内置电阻状态，驱动能力，以及输出电平，而 GPIO 名称，端口与端口编号是在申请的时候确定，以后一直无法修改的。

下面，还是以 twi_para 为例，说明如何使用这个函数。

假定 GPIO 句柄是 gpio_hd，已经控制了 twi_sda 和 twi_scl 两个 GPIO，现在 twi_sda 已经被修改过，现在希望改回最初配置的值，那么代码如下

```
{
    int    ret;

    ret = Gpio_set_one_pin_status (gpio_hd, 0, “twi_sda”, 0);
    if(ret < 0)
    {
        printf(“set gpio infomation fail\n”);
    }
    else
    {
        printf(“set gpio infomation ok\n”);
    }
    return ret;
}
```

调用这个函数，就可以把 twi_para 中的 twi_sda 还原成最初的值。可以看到，参数列表中，句柄不可缺少，第二个参数是 0，是一个空指针，第三个参数是 gpio 的名称，第三个参数是 0，表示使用用户最初传入的参数设置，因此第二个参数无效。如果第四个参数写 1，而第二个参数还是 0 的话，则函数会返回一个错误，因为用户当前传入的数据无效。

```
{
    Gpio_set_one_pin_status (gpio_hd, 0, “twi_sda”, 1);
}
```

错误：需要使用用户传入的参数，但是用户传入了一个空指针

再提供一个示例，表示按照用户当前传入的参数修改一个 GPIO，还是以 twi_para 为例，假定 GPIO 句柄是 gpio_hd，需要修改 twi_scl。

```
{
    int    ret;
    user_gpio_set_t     gpio_info;

    gpio_info.mul_sel = 1;          //功能分配修改成输出状态
    gpio_info.pull      = 1;        //内置电阻修改成上拉
    gpio_info.drv_level=1;          //驱动能力设置等级 1
    gpio_info.data      =1;         //输出电平设置为高电平
    ret = Gpio_set_one_pin_status (gpio_hd, &gpio_info, "twi_scl", 1);
    if(ret < 0)
    {
        printf("set gpio infomation fail\n");
    }
    else
    {
        printf("set gpio infomation ok\n");
    }
    return ret;
}
```

函数调用完成，可以看到，twi_para 句柄下的 twi_scl 的状态已经被改变成为输出状态，即 PB1 脚成为了输出功能，同时电平时高。可以利用万用电表看到电平为高。

5.6 设置句柄下单个 GPIO 输入输出状态

这个函数用于修改一个 GPIO 的输入输出状态。注意的是，只能修改输入输出状态，不能用于修改 GPIO 的其它功能分配。当用于非 IO 状态的功能分配时，函数会报错返回。

函数原型： Int Gpio_set_one_pin_io_status(__hdle devpin, unsigned if_set_to_output_status, const char *gpio_name);

参数：devpin GPIO 控制句柄，来自于 GPIO 申请函数

if_set_to_output_status 是否设置成输出状态 0 设置成输入状态，1 设置成输出状态

gpio_name GPIO 的名称，来源于配置脚本

返回值： 成功返回 0

失败返回-1

这个函数功能很简单，只能分配 IO 功能。gpio_name 来源于配置脚本，对于单个的 GPIO，这个值可以不用填写。假设，脚本中有一个如下的配置

```
[test]
test_key = port:pa14<1><1><1><1>
; 这个 GPIO 表示使用端口 A 的第 14 个引脚作为输出功能，驱动能力 1，内置电阻上拉，默认输出高电平
```

现在首先已经获取到了这个 GPIO 控制的句柄，然后要修改这个 GPIO 成为输入状态，则调用如下的函数。

```
{
    int  ret;

    ret = Gpio_set_one_pin_io_status (gpio_hd, 0, “test_key”);

    return ret;
}
```

如果用户非常确定 test 主键下只有一个 GPIO 属性的子键，或者在获取句柄的时候，使用

的是主键加子键查找 GPIO 的方式，或者，使用的是用户自定义查找 GPIO 的方式，则可以采用隐藏按键名称的方式来调用函数，形式如下：

```
{
int    ret;

       ret = Gpio_set_one_pin_io_status (gpio_hd, 0, 0);

       return ret;
}
```

使用如下的函数，可以修改 GPIO 的输入输出状态。

5.7 设置句柄下单个 GPIO 内部电阻状态

这个函数用于修改句柄控制下的一个 GPIO 的内置电阻状态。可以选择的是内置电阻上拉，下拉，或者是处于高阻状态。选择其它的状态函数会报错返回。

函数原型：Int Gpio_set_one_pin_pull (__hdle devpin, unsigned set_pull_status, const char *gpio_name);

参数：devpin　　GPIO 控制句柄，来自于 GPIO 申请函数

Set_pull_status　设置内部电阻的状态，0 高阻，1 上拉，2 下拉

gpio_name　　GPIO 的名称，来源于配置脚本

返回值：　成功返回 0

失败返回-1

函数功能明确，可以设置内部电阻的状态。内部电阻只有高阻，上拉，下拉状态，配置在函数的第二个参数处。

如果句柄控制的 GPIO 个数大于 1 个，则需要通过 GPIO 的名称来识别是具体的哪一个 GPIO。GPIO 的名称来源于配置文件，或者用户自定义的时候给 GPIO 取的一个字符串标示。

举例说明如何通过这个函数控制 GPIO 的内置电阻属性。还是以 twi_para 为例，假定句柄

是 gpio_hd，现在希望改变 twi_scl 的内置电阻属性为下拉。

```
{
    int    ret;

    ret = Gpio_set_one_pin_pull    (gpio_hd, 2, “twi_scl”);

    return ret;
}
```

根据规定，2 代表下拉，因此这里传的第二个参数是 2。如果希望修改内置电阻为上拉，则传的参数是 1。如果要保持内置电阻为高阻态，需要传递的参数是 0。传递其它参数无效，会返回一个错误。

和改变输入输出功能类似的，对于句柄只控制了一个 GPIO 的情况，GPIO 名称处可以写 0，即不需要传递 GPIO 名称。如果 twi_para 下只有一个 GPIO，其名称是 twi_scl，则也可以使用如下的写法。

```
{
    int    ret;

    ret = Gpio_set_one_pin_pull    (gpio_hd, 2, 0);

    return ret;
}
```

如果 twi_para 下有多于一个 GPIO 的情况，则第三个参数处传递 0 会直接返回一个错误。

5.8 设置句柄下单个 GPIO 驱动能力等级

这个函数使用于修改驱动能力。驱动能力表示 GPIO 在驱动外部设备时候的能力。驱动能力越高，表示在推动电平变化的能力越强，波形上会看到从低到高或者从高到低变化很陡峭，但是会出现过冲；驱动能力等级低，波形上看到电平变化平缓，不会出现过冲，但是变化的时间会比较长。通常，使用驱动能力 1 就能满足要求，某些苛刻的设备可能需要其他的驱动能力等级。

函数原型：Int Gpio_set_one_pin_driver_level(__hdle devpin, unsigned set_driver_level, const char *gpio_name);

参数：devpin GPIO 控制句柄，来自于 GPIO 申请函数

set_driver_level 要设置的驱动能力等级，0-3 分别代表驱动能力等级 0-3

gpio_name GPIO 的名称，来源于配置脚本

返回值： 成功返回 0

失败返回-1

函数的用法和修改内置电阻的用法非常相似，除了函数中第二个参数代表的意义不一致以外，其它的参数，以及函数的用法都完全一样。

这里还是给出两个例子，描述如何使用它来修改一个 GPIO 的驱动能力。

```
{
    int    ret;

    ret = Gpio_set_one_pin_pull    (gpio_hd, 3, “twi_scl”);

    return ret;
}
```

这个例子假定了句柄 gpio_hd 控制了 twi_para 的两个 GPIO，分别是 twi_scl 和 twi_sda。现在，通过这个函数，设置了 GPIO twi_scl 的驱动能力为 3。

又如有如下的一个 GPIO，用户自行定义了属性，然后要修改它的驱动能力。

```
{
    int    ret;
    user_gpio_set_t    gpio_info;
    __hdle      gpio_hd;
    gpio_info.port = 1;                 //使用 PORTA
    gpio_info.port_num = 14;      //使用端口 A 的第 14 引脚
    gpio_info.mul_sel = 1;           //设置为输出功能
    gpio_info.pull = 1;                 //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;        //设置默认驱动能力为 1
    gpio_info.data       = 1;        //设置默认输出高电平
    gpio_hd = Gpio_request (&gpio_info, 1); //只申请一个 GPIO
    if(!gpio_hd)
    {
        return -1;
    }
    ret = Gpio_set_one_pin_pull    (gpio_hd, 2, 0);
    return ret;
}
```

上面的例子中，用户使用 PA14 来做输出功能。然后在后面的函数中，修改了 PA14 的驱动能力从等级 1 变成等级 2。

5.9 读出句柄下单个 GPIO 端口电平

这个函数可以读出一个功能为输入的 GPIO 的电平，返回值如果是 0，表示端口电平是低；如果返回值是 1，表示端口电平是高；返回值如果是-1，表示读出的值无效，当前 GPIO 并没有处于输入状态。

函数原型：Int Gpio_read_one_pin_value (__hdle devpin, const char *gpio_name);

参数：devpin　　GPIO 控制句柄，来自于 GPIO 申请函数

　　gpio_name　　GPIO 的名称，来源于配置脚本或者是用于自定义

返回值：　高电平返回 1

　　低电平返回 0

　　非输入状态返回-1

通常，用户会读入一个 GPIO 的电平，这是一个使用会比较频繁的函数。假如 gpio_hd 控制了 twi_para 的两个 GPIO，现在希望读出 sda 和 scl 的端口电平，则要使用如下的方法。

```
{
    int    ret1, ret2;

    ret1 = Gpio_read_one_pin_value(gpio_hd, “twi_scl”);
    ret2 = Gpio_read_one_pin_value(gpio_hd, “twi_sda”);

    return 0;
}
```

很明显的是，函数的返回值 ret1 和 ret2 都将是-1。因为他们都处于非输入状态，是不能读到其所对应的端口电平的，因此会给以-1 的返回值。

现在，有用户自己定义了一个 GPIO，希望获取电平，用户的代码编写方式如下。

```
{
    int    ret;
    user_gpio_set_t    gpio_info;
    __hdle      gpio_hd;
    gpio_info.port = 1;                 //使用 PORTA
    gpio_info.port_num = 14;      //使用端口 A 的第 14 引脚
    gpio_info.mul_sel = 0;           //设置为输入功能
    gpio_info.pull = 1;                 //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;        //设置默认驱动能力为 1
    gpio_info.data       = 1;        //设置默认输出高电平，由于是输入状态，无效
    gpio_hd = Gpio_request (&gpio_info, 1); //只申请一个 GPIO
    if(!gpio_hd)
    {
        return -1;
    }
    ret = Gpio_read_one_pin_value(gpio_hd, 0);         (1)
    Gpio_set_one_pin_io_status(gpio_hd, 1, 0);
    ret = Gpio_read_one_pin_value(gpio_hd, 0);      (2)

    return 0;
}
```

在上面的例子中，调用了两次 Gpio_read_one_pin_value 函数。由于用于开始的时候把 PA14 设置成输入状态，因此在(1)处，可以读出端口的电平。这个电平的高低由具体的电路来决定。紧接着，用户调用函数把 PA14 的功能修改成输出状态，在(2)处试图再去读出端口的值，就无法读出了。因为这个时候 PA14 已经处于输出状态，无法读出端口电平。

函数的第二个参数和前面的函数用法一致，对于单个 GPIO 的句柄，这个参数可以写 0；对于多个 GPIO 的句柄，这个参数必须和其中的需要控制的 GPIO 的名称匹配。

5.10 设置句柄下单个 GPIO 端口电平

当一个 GPIO 处于输出状态的时候，可以使用这个函数改变当前 GPIO 的端口电平，可以修改成低电平 0，或者是高电平 1。但是如果 GPIO 没有处于输出状态，使用这个函数无效，会返回一个-1。

函数原型：Int Gpio_write_one_pin_value(__hdle devpin, unsigned value_to_gpio, const char *gpio_name);

参数：devpin　　GPIO 控制句柄，来自于 GPIO 申请函数

value_to_gpio　　设定输出电平的值，0 代表低电平，1 代表高电平

gpio_name　　GPIO 的名称，来源于配置脚本或者是用于自定义

返回值：　成功返回 0

失败返回-1

下面，还是以实际示例来说明如何修改一个 GPIO 的输出电平。

首先查看一个会返回失败的示例。假定已经通过 gpio_hd 控制了 twi_para 下的两个 GPIO，twi_sda 和 twi_scl。现在希望控制它们的输出电平。使用方法如下。

```
{
    int    ret1, ret2;

    ret1 = Gpio_write_one_pin_value(gpio_hd, 1, “twi_scl”);
    ret2 = Gpio_write_one_pin_value(gpio_hd, 0, “twi_sda”);

    return 0;
}
```

很明显，这个函数调用无法成功，因为 twi_sda 和 twi_scl 两个 GPIO 没有处于输出状态。可行的办法是修改这两个 GPIO 为输出状态，然后去修改端口的电平。

```
{
    int    ret1, ret2;

    Gpio_set_one_pin_io_status(gpio_hd, 1, “twi_scl”);
    Gpio_set_one_pin_io_status(gpio_hd, 1, “twi_sda”);

    ret1 = Gpio_write_one_pin_value(gpio_hd, 1, “twi_scl”);
    ret2 = Gpio_write_one_pin_value(gpio_hd, 0, “twi_sda”);

    return 0;
}
```

现在通过增加调用函数 Gpio_set_one_pin_io_status，修改 GPIO 为输出状态，就可以设置端口电平了。通过这个函数示例，设置 twi_scl 为高电平，设置 twi_sda 为低电平。

现在，有用户自定义了一个 GPIO 控制(PA14)，由于其中只有一个 GPIO，则在 gpio_name 处的参数可以不用填写。

```
{
    int    ret;
    user_gpio_set_t    gpio_info;
    __hdle      gpio_hd;
    gpio_info.port = 1;                 //使用 PORTA
    gpio_info.port_num = 14;      //使用端口 A 的第 14 引脚
    gpio_info.mul_sel = 1;           //设置为输出功能
    gpio_info.pull = 1;                 //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;        //设置默认驱动能力为 1
    gpio_info.data       = 1;         //设置默认输出高电平
    gpio_hd = Gpio_request (&gpio_info, 1); //只申请一个 GPIO
    if(!gpio_hd)
    {
        return -1;
    }
    ret = Gpio_write_one_pin_value(gpio_hd, 0, 0);          (1)
    Gpio_set_one_pin_io_status(gpio_hd, 0, 0);
    ret = Gpio_write_one_pin_value(gpio_hd, 1, 0);         (2)

    return 0;
}
```

在这个函数中，当用户申请 GPIO 成功之后，PA14 就已经被配置成为输出状态了，默认电平是高。调用函数(1)之后，用户把 PA14 的输出电平设置成低电平。接下来，用户修改 PA14 的功能成为输入状态，这时候，就无法继续设置端口电平。再调用函数(2)，用户试图设置电平重新为高，由于不处于输出状态，设置电平无法成功，会返回一个-1。端口实际的电平由外部电路决定。

5.11 GPIO 管理中的句柄

5.11.1 为什么要使用句柄

这里简单说说为什么要在 GPIO 管理中添加句柄这个项目。

在程序中，句柄通常只是一个地址，它指向了一片内存区域，这片区域保存了一系列有意义的数据。很多地方可以看到句柄的存在，比如文件操作中的文件指针，就是句柄的典型。

GPIO 管理中，使用句柄是为了管理 GPIO 的安全，特别是存在 GPIO 复用的情况。当同一个 GPIO，假如是 PC0，即端口 C 的第 0 脚，既可以被 NAND 使用，也可以被卡使用的时候，那么，到底给谁使用呢？

GPIO 管理中，有冲突检查的机制。如果 PC0 没有人使用过，则 NAND 申请的时候，可以得到它的控制权，想怎么用就怎么用。在 NAND 使用的过程中，如果卡也希望使用 PC0，在申请的时候就会发现无法申请到，那么，它只有等待 NAND 用完释放之后才能拿到 PC0 的控制权。这个道理很容易理解，PC0 是一个资源，可以被资源锁控制。

有了资源锁在 GPIO 管理里面，可以解决以下三种情况。

1) 避免模块间使用 GPIO 的冲突

2) 避免模块功能和 IO 功能的冲突

3) 避免 IO 功能中输入和输出功能的冲突

5.11.2 避免模块间的使用冲突

如果没有句柄，也就不需要申请，当 NAND 想要 PC0 的时候，直接就可以控制了；当卡想要 PC0 的时候，也是可以直接控制。但是，卡并不关心当前有没有人在使用 PC0，这样很容易导致 NAND 出错。当卡在用的时候，NAND 来控制 PC0，同样会导致卡出错，因此，没有申请的步骤，就会导致控制的混乱。

现在有了申请的动作，如果不需要返回句柄，比较可行的办法就是，NAND 在每次操作 GPIO 的过程中，都直接使用配置脚本中的数据来控制 GPIO，这样子看起来省掉了句柄的概念。但是事实上，对于 GPIO 管理模块来说，就不知道到底是谁在控制这些 GPIO 了。比如，NAND 申请了 PC0，不需要返回句柄；现在，卡也希望使用 PC0，但是申请不到，怎么办？正常情况下它应该等待，但是如果耐心不好，就可能直接使用 PC0 的相关参数去释放。

```
{
    user_gpio_set_t    gpio_info[1];

    gpio_info.port = 3;              //使用 PORTC
    gpio_info.port_num = 0;          //使用端口 C 的第 0 引脚
    gpio_info.mul_sel = 2;           //设置为 NAND 功能
    gpio_info.pull = 1;              //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;         //设置默认驱动能力为 1

    Gpio_request (gpio_info, 1);
    return 0;
}
```

比如，NAND 做了上方的申请:

```
{
    user_gpio_set_t    gpio_info[1];

    gpio_info.port = 3;              //使用 PORTC
    gpio_info.port_num = 0;          //使用端口 C 的第 0 引脚
    gpio_info.mul_sel = 3;           //设置为卡功能
    gpio_info.pull = 1;              //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;         //设置默认驱动能力为 1

    Gpio_request (gpio_info, 1);
    return 0;
}
```

申请的过程中，不需要没有返回句柄。则卡在使用的过程中，也希望 PC0 用于卡的功能，

它也做如上方的申请：

由于 PC0 已经被 NAND 申请到了，卡无法申请到，它只有等待 NAND 用完 PC0 之后释放。但是，前面的做法中有一个安全隐患，即，释放 PC0 的时候，GPIO 管理模块并不知道到底是谁在释放 PC0。那么，卡在申请不成的时候，也可以做如下的操作，直接释放掉 NAND 的 PC0。

```
{
    user_gpio_set_t    gpio_info[1];

    gpio_info.port = 3;              //使用 PORTC
    gpio_info.port_num = 0;          //使用端口 C 的第 0 引脚
    gpio_info.mul_sel = 3;           //设置为卡功能
    gpio_info.pull = 1;              //设置默认为内置电阻上拉
    gpio_info.drv_level = 1;         //设置默认驱动能力为 1

    Gpio_release(gpio_info, 0);
    return 0;
}
```

卡如果这样野蛮的释放掉 NAND 使用中的 GPIO，则 NAND 会出错。所以，没有句柄概念在其中管理不同的 GPIO，GPIO 模块的管理工作存在安全隐患。当有了句柄概念，NAND 申请的时候获得控制 PC0 的句柄，释放的时候也必须通过句柄释放。那么卡在希望使用 PC0 的时候，无法野蛮释放掉 NAND 的 PC0 的控制权，因此，GPIO 模块管理的安全性得到极大的提高。

5.11.3 避免模块和 IO 的使用冲突

在 LINUX 操作系统中，通常会直接操作一个 GPIO，读出端口的电平，或者设置端口的电平，看起来使用非常方便，不管读还是写一个端口都是一个函数直接搞定。但是，方便的前提是必须知道当前使用的 GPIO 没有任何人在使用，其实这个要求比较高，通常用户不太会关心这个 GPIO 有没有人在使用。因此，就可能存在如下的问题，当 NAND 在使用 PC0 的时候，用户并不关心，他想知道 PC0 的电平，直接去读出 PC0 的端口的值，这个操作，将导致 PC0 的

功能变成输入功能，而失去了当做 NAND 使用的功能，将导致 NAND 操作出错。同样的，在 PC0 输出一个电平也会导致同样的问题出现。

正确的做法是在操作 PC0 之前，申请一下 PC0 的控制权，如果申请不到，表示当前 GPIO 已经有人使用，用户就不应该继续操作 PC0。如果坚持，很可能导致使用 PC0 的模块出错。当然，在有句柄的管理模式下，当申请不到 PC0 的控制权的情况下，是无法继续控制 PC0 的。

5.11.4 避免 IO 间的使用冲突

IO 冲突和模块使用冲突非常相似，这里就不举例解释。现在假设有两个模块在使用同一个 IO 的输入输出功能，比如 PC0。

模块 A 希望能够长时间读入 PC0 的电平，当电平变低的时候做一件事情，当电平变高的时候做另一件事情。模块 B 则希望改变 PC0 的电平，能够在自己的意图下，有时候输出高电平，有时候输出低电平。

很明显，两个模块同时操作一个 GPIO 会导致任何一个模块的工作都不正常。事实上，只要它们在使用之前做一个申请动作，就可以在一个模块拿到控制权之后，另一个模块无法使用这个 GPIO，避免了 GPIO 使用中功能冲突的问题。

六．FAQ

略

七．结束

欢迎各位提出宝贵意见，把系统中的配置与 GPIO 管理做得更加优秀，使用更加方便。