**SHRD100 GPIO与HAL移植设计说明**

|  |  |
| --- | --- |
| 拟 制 |  |
| 审 核 |  |
| 会签 |  |
| 批 准 |  |

**修订记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **修订版本** | **日期** | **作者** | **修改描述** | **备注** |
| V1.0 | 2023.07.07 | 谭绍军 | 初始版本 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1 引言 1](#_Toc139724509)

[1.1 目的 1](#_Toc139724510)

[1.2 范围 1](#_Toc139724511)

[1.3 缩略语定义 1](#_Toc139724512)

[1.4 参考资料 1](#_Toc139724513)

[2 移植方案概述 1](#_Toc139724514)

[3 移植方案设计说明 1](#_Toc139724515)

[3.1 GPIO移植说明 1](#_Toc139724516)

[3.1.1 现有GPIO移植方案介绍 1](#_Toc139724517)

[3.1.2 后续GPIO优化 1](#_Toc139724518)

[3.2 HAL移植说明 1](#_Toc139724519)

[3.2.1 HAL现状说明 1](#_Toc139724520)

[3.2.2 HAL移植说明 1](#_Toc139724521)

[3.2.3 移植问题 1](#_Toc139724522)

[3.3 中断移植说明 1](#_Toc139724523)

[3.3.1 底层驱动支持 1](#_Toc139724524)

[3.3.2 应用层接收中断并处理 1](#_Toc139724525)

[4 附件 1](#_Toc139724526)

# 引言

## 目的

本文为“SHRD100 GPIO与HAL移植设计说明”，主要用于定义软件功能，供项目组开发人员和软件维护人员阅读。

## 范围

本文档只限于塞防科技项目组研发、测试、产品以及项目相关人员作为内部信息对齐使用，未经公司批准以及书面授权不允许任何人以任何形式对本文档复制、传播、改动。

## 缩略语定义

|  |  |  |
| --- | --- | --- |
| **缩略语** | **全称** | **描述** |
|  |  |  |

## 参考资料

|  |  |
| --- | --- |
| **名称** | **版本** |
| SHRD100嵌入式软件设计说明 | V1.2 |
|  |  |
|  |  |
|  |  |

# 移植方案概述

由于项目时间紧，为了便于快速接入及调试，本移植方案中尽量保留原RTOS接口，使用简洁方式实现，未考虑性能需求。

# 移植方案设计说明

## GPIO移植说明

### 现有GPIO移植方案介绍

在原RTOS代码里，将GPIO操作分成了output和input两个模块，其中OutputArg[]里存放了所有output型GPIO口，InputArg[]里存放了所有input型GPIO口，内部调用XGpioPs\_SetDirectionPin，XGpioPs\_WritePin，XGpioPs\_ReadPin等自带接口对GPIO进行操作，对应用层提供OutputInit、GPIO\_OutputCtrl、Output\_GetValue、InputInit、Input\_GetValue等接口。

我在将其移植到Linux时，采用了基于sysfs访问方式来操作GPIO，内部调用gpio\_export、gpio\_direction、gpio\_write、gpio\_read接口通过对"/sys/class/gpio/export"、"/sys/class/gpio/gpio%d/direction"、"/sys/class/gpio/gpio%d/value"文件的读写来实现对GPIO口的控制。

fd = open("/sys/class/gpio/gpio%d/value", O\_WRONLY);

write(fd, value == 0 ? “0” : “1”, 1);

close(fd);

在初始化后通过导出export会生成/sys/class/gpio/gpioxxx很多个目录，后续每次GPIO读写操作都会open、close一次/sys/class/gpio/gpioxxx/value文件，如果频繁对GPIO操作会影响效率。

### 后续GPIO优化

从4.8版本开始，Linux内核已经加入了libgpiod的支持，原有基于sfsfs的访问方式将被逐渐放弃。可以直接对/dev/gpiochip\*进行访问，内核空间提供的主要函数有：gpiod\_get、gpiod\_put、gpiod\_direction\_input、gpiod\_direction\_output、gpiod\_get\_value、gpiod\_set\_value、gpiod\_to\_irq等。 用户空间可以通过链接libgpiod库来编程，比如使用gpiod\_chip\_open、gpiod\_chip\_get\_line、gpiod\_line\_request\_input、gpiod\_line\_get\_value、gpiod\_line\_request\_output、gpiod\_line\_set\_value函数。 此外gpiod还提供了命令行配置：gpiodetect、gpioinfo、gpioset、gpioget、gpiomon，这样在应用程序外还可以通过命令行脚本程序来实现功能测试与验证。

## HAL移植说明

### HAL现状说明

在RTOS的hal.c里主要提供寄存器操作接口： axi\_write、axi\_read、axi\_write\_data、axi\_read\_data，

其中axi\_write、axi\_read 是直接对传过来的物理地址进行读写操作，接口未对地址范围做校验和限制，

常用地址定义在enum PL\_REG\_ADDR中，

/\* Definitions for interface M08\_AXI \*/

#define XPAR\_M08\_AXI\_BASEADDR 0x81000000

#define XPAR\_M08\_AXI\_HIGHADDR 0x81000FFF

/\* Definitions for interface M09\_AXI \*/

#define XPAR\_M09\_AXI\_BASEADDR 0x81010000

#define XPAR\_M09\_AXI\_HIGHADDR 0x81010FFF

enum PL\_REG\_ADDR

{

eCFG\_PWR\_GATE\_L = XPAR\_M08\_AXI\_BASEADDR + 0x00,

eCFG\_PWR\_GATE\_H = XPAR\_M08\_AXI\_BASEADDR + 0x04,

eCFG\_RF\_MOD = XPAR\_M08\_AXI\_BASEADDR + 0x08,

eCFG\_PINC\_CH0 = XPAR\_M08\_AXI\_BASEADDR + 0x0c,

eCFG\_PINC\_CH1 = XPAR\_M08\_AXI\_BASEADDR + 0x10,

eIDX\_FREQOFFSET = XPAR\_M08\_AXI\_BASEADDR + 0x20,

eRF\_MOD = XPAR\_M08\_AXI\_BASEADDR + 0x24,

eWR\_DDR\_ENABLE = XPAR\_M09\_AXI\_BASEADDR + 0x14,

};

使用例子：axi\_write(eCFG\_PWR\_GATE\_L, 0x00200000);

axi\_write\_data、axi\_read\_data则是先进行偏移量计算后再读写，

/\* Canonical definitions for peripheral AXI\_BRAM\_CTRL\_PARA \*/

#define XPAR\_BRAM\_0\_BASEADDR 0x80000000U

#define XPAR\_BRAM\_0\_HIGHADDR 0x8000FFFFU

return \*(volatile unsigned int \*)(XPAR\_BRAM\_0\_BASEADDR + (addr << 2))

typedef enum

{

PL\_AD\_START = 0x00, // AD使能寄存器 0：打击状态；1：侦测状态

PL\_LO\_CODE = 0x01, // ad9361本振号

PL\_REG\_AD0 = 0x02, // AD配置寄存器 对应上一版本0x0010地址寄存器(userreg\_0x0010)

PL\_REG\_AD1 = 0x03, // AD配置寄存器 对应上一版本0x0011地址寄存器(userreg\_0x0011)

PL\_REG\_AD2 = 0x04, // AD配置寄存器 对应上一版本0x0012地址寄存器(userreg\_0x0012)

PL\_REG\_AD3 = 0x05, // AD配置寄存器 对应上一版本0x0020地址寄存器(userreg\_0x0020)bit[0]：o\_ad\_txnrx、硬件管脚(AA5)；bit[1]：o\_ad\_enable、硬件管脚(AE3)； bit[2]：o\_ad\_enagc、硬件管脚(Y1)；bit[3]：o\_ad\_rst，硬件管脚(AA6)；

PL\_DATA\_SELA = 0x06, // A通道输出频段选择 0：5.2GHz；1：5.8GHz；2：5.2~5.8

PL\_DATA\_SELB = 0x07, // B通道输出频段选择 0：2.4GHz；1：5.8GHz；2：2.4~3.5

PL\_SL\_SWITCH = 0x08, // 门限检测结果输出、参数测量结果输出、外协算法结果输出功能切换 0：门限检测输出(算法模块一)；      1：参数测量结果输出(算法模块二)；        2：外协算法前处理结果输出(算法模块三)；

PL\_DLY\_EN\_VTC = 0x09, // IDELAY控制

PL\_DLY\_LOAD = 0x0A, // IDELAY控制

PL\_DLY\_D = 0x0B, // IDELAY控制

PL\_FAN\_PWM\_RATE = 0x0C, // FAN转速控制 风扇转速分0~10档,0档转速为0;10档转速最快;

PL\_PL\_START = 0x0D, // PL上电正常，可以进行初始化操作 0：PL端上电异常；      1：PL端上电成功

PL\_IRDRATE = 0x10, // 2-4G/4-6G 打击速度间隔设置 0：5.12us     1：10.24us  2：15.36us 3:20.48

PL\_MOTOR\_AD4 = 0x11, // AD配置寄存器 对应上一版本0x0090地址寄存器(userreg\_0x0090)

PL\_MOTOR\_AD5 = 0x12, // AD配置寄存器 对应上一版本0x0091地址寄存器(userreg\_0x0091)

PL\_MOTOR\_AD6 = 0x13, // AD配置寄存器 对应上一版本0x009f地址寄存器(userreg\_0x009f)

PL\_I\_FPGA\_TEMP0 = 0x14, //

PL\_I\_FPGA\_TEMP1 = 0x15, //

PL\_I\_FPGA\_TEMP2 = 0x16, //

PL\_I\_FPGA\_TEMP3\_DIR = 0x17, // AD9361输出通道选择寄存器 0：全向通道 1：定向通道

PL\_I\_FPGA\_CURR0 = 0x18, //

PL\_I\_FPGA\_CURR1 = 0x19, //

PL\_I\_FPGA\_CURR2 = 0x1A, //

PL\_I\_FPGA\_BAT = 0x1B, //

PL\_AEAG\_VP0 = 0x38, // 版本号0 编号+年：如：00022022

PL\_AEAG\_VP1 = 0x3c, // 版本号1 月+日+小时+分钟  如：10151800

} PL\_REG;

使用例子：axi\_write\_data(PL\_DLY\_LOAD, 0x1FFF);

在AD9361中对接口进行了重新封装，ad9361\_spi\_write\_byte -> write\_ad936x -> write\_spi\_bram -> axi\_write\_data

通过宏定义地址访问，如：write\_spi\_bram(AD936X\_CTR\_ADDR, dat)

#define AD9363\_SPI\_RE 0x02 //bit0 @0x10,read enable\*

#define AD9363\_SPI\_WE 0x02 //bit1 @0x10,write enable\*

#define AD9363\_SPI\_RW\_ADDR 0x03 //bit0-bit9,read and write addr\*

#define AD9363\_SPI\_WDATA 0x04 //bit0-bit7,write data regaddr\*

#define AD9363\_SPI\_RDATA 0x11 //[s\_ad\_spi\_result,s\_ad\_spi\_rdata];

#define AD9363\_SPI\_WRITE\_ENABLE 0x02

#define AD9363\_SPI\_READ\_ENABLE 0x01

#define AD9363\_SPI\_RW\_DISABLE 0x00

#define AD936X\_CTR\_ADDR 0x05 //txnrx -bit0;enable -bit1; enagc -bit2; reset -bit3; \*

#define AD936X\_TXNRX\_CTR\_BIT 0x0

#define AD936X\_ENABLE\_CTR\_BIT 0x01

#define AD936X\_ENAGC\_CTR\_BIT 0x02

#define AD936X\_RST\_CTR\_BIT 0x03

还有基于配置文件的调用，地址范围0x0-0x3ff，如：  
 u8 Ad9361Config[][3] = {

{0x81,0x31,0x01}, //SPIWrite 131,01 // Ext LNA, Int LNA, & Mixer Gain Word

{0x81,0x32,0x09}, //SPIWrite 132,09 // TIA & LPF Word

{0x81,0x33,0x00}, //SPIWrite 133,00 // DC Cal bit & Dig Gain Word

{0x81,0x37,0x1E}, //SPIWrite 137,1E // Write Words

但还有一些其他方式调用，地址不在枚举里，也没有宏定义的，代码可读性比较差，

如write\_fpga\_regs(0x30, 0x01); write\_fpga\_regs(0x31, 0x09);

### HAL移植说明

在Linux系统中无法直接对物理地址进行操作，需要先用mmap方法映射到虚拟内存，对虚拟地址进行读写操作后再msync同步

针对上述使用情况，我在初始化流程中对三段地址进行了映射：

0x80000000: 0x10000

0x81000000: 0x1000

0x81010000: 0x1000

调用axi\_write、axi\_read接口会先将地址进行匹配，看属于上面三段中的哪一段，有没有超出范围，只有在范围内的才进行读写操作。

调用axi\_write\_data、axi\_read\_data接口则只会在第一段0x80000000: 0x10000范围内进行寄存器读写操作。

### 移植问题

上述接口在应用层、HAL层、CLI命令中都有直接调用，HAL层中可以直接调用接口，应用层和CLI命令需要封装接口再通过EAP\_DRVCODE\_SET\_PL\_REG/EAP\_DRVCODE\_GET\_PL\_REG方式来进行寄存器操作，适配新代码框架。

## 中断移植说明

### 底层驱动支持

Tracer使用了和雷达Linux相同的中断方案，即内核驱动模块+设备树的方式，下面以pl\_event模块举例。

在sdk/project-spec/meta-user/recipes-modules/pl-event/下增加了pl\_event内核驱动模块，

# lsmod

Module Size Used by

pl\_event 16384 0

单独编译pl-event模块的方法：

petalinux-build -c pl-event

将编译生成的sdk/build/tmp/sysroots-components/zynqmp\_generic/pl-event/lib/modules/5.15.36-xilinx-v2022.2/extra/pl-event.ko拷贝到板子上，

rmmod pl\_event 再insmod pl-event.ko即可重新加载该驱动模块。

还需要修改设备树文件sdk/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi，添加对应中断号。

/{

adc\_irq\_event {

compatible = "skyfend,pl-event";

interrupt-names = "interrupt";

interrupt-parent = <&gic>;

interrupts = <0 89 1>; // #define XPAR\_FABRIC\_AXI\_DMA\_ADC\_S2MM\_INTROUT\_INTR 121U

status = "okay";

};

rdmap\_irq\_event {

compatible = "skyfend,pl-event";

interrupt-names = "interrupt";

interrupt-parent = <&gic>;

interrupts = <0 90 1>; // #define XPAR\_FABRIC\_AXI\_DMA\_OTHERS\_S2MM\_INTROUT\_INTR 122U

status = "okay";

};

burst\_irq\_event {

compatible = "skyfend,pl-event";

interrupt-names = "interrupt";

interrupt-parent = <&gic>;

interrupts = <0 109 1>; // #define READ\_DDR\_BURST\_ID 141

status = "okay";

};

};

修改完成后petalinux-build重新编译，会在sdk/image/linux下生成新的设备树文件system.dtb，

Ssh连接到板子替换覆盖/run/media/mmcblk0p1/system.dtb，重启后生效。

登录串口或ssh可以查看到141号中断在CPU0中调用次数

# cat /proc/interrupts

67: 6198 0 0 0 GICv2 141 Edge pl-event

### 应用层接收中断并处理

在应用层中open("/dev/PL-burst\_irq\_event", O\_RDONLY)，

通过ret = select(fd+1, &fdset, NULL, NULL, &time);来捕获中断信号，就可以做相应处理了。

# 附件

无。