XPI-3566-ZERO

Linux 系统软件开发指南

前言

概述

文档作为 XPI-3566-ZERO Debian Linux 系统软件开发指南，旨在帮助软件开发工程师、技术支持工程师更快上手 XPI-3566-ZERO 的开发及调试。

读者对象
本文档（本指南）主要适用于以下工程师：
技术支持工程师
软件开发工程师

修订记录

日期	版本	作者	修改说明
23/10/9	V1.0	LiuZC	初始版本

版权所有© 2023 深圳金亚太科技有限公司（Geniatech）
超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。 深圳金亚太科技有限公司
Shenzhen Geniatech Inc., Ltd.
地址： 南山区西丽街道国际创新谷 8 栋 A 座 1002-1004 室
网址： www.geniatech.com
客户服务电话： +86-0755-86028588
技术支持邮箱： support @geniatech.com
销售服务邮箱： sales@geniatech.com

目录

1. 编译环境搭建 1

1.1 获取 SDK 1
1.2 安装依赖包 1
1.3 交叉编译工具链介绍 2
1.4 SDK 工程目录介绍 3

2. 代码编译 4

2.1 编译前选择配置文件 4
2.2 编译 SDK 4
2.2.1 U-Boot 编译 4
2.2.2 Kernel 编译 4
2.2.3 Debian 编译 4
2.2.4 全自动编译 5
2.3 固件打包 5

3. 固件烧写 6

3.1 安装烧写工具驱动： 6
3.2 WINDOS 刷机说明 6

4. 软件开发 8

4.1 软件开发目录介绍 8
4.2 DTS 介绍 8
4.3 KERNEL 9
4.3.1 内核定制 9
4.3.2 内核编译 10
4.4 GPIO 11
4.4.1 简介 11
4.4.2 用户空间控制 GPIO 12
4.4.3 驱动控制 GPIO 13
4.5 I2C 17
4.5.1 简介 17
4.5.2 定义和注册 I2C 设备 18
4.5.3 定义和注册 I2C 驱动 18
4.5.4 通过 I2C 收发数据 20
4.5.5 测试 I2C 设备 21
4.6 UART 22
4.6.1 简介 22
4.6.2 配置 UART4 接口 23
4.6.3 配置调试串口 24
4.7 SPI 25

4.7.1. 简介 *25*
4.7.2 定义和注册 SPI 设备 *25*
4.7.3 定义和注册 SPI 驱动 *26*
4.7.4 测试 SPI 设备: *27*

1. 编译环境搭建

本章介绍 Linux SDK 的编译环境搭建

注意：

（1）推荐在 X86_64 Ubuntu 18.04 系统环境下进行开发，若使用其它系统版本，可能需要对编译环境做相应调整。

（2）使用普通用户进行编译，不要使用 root 用户权限进行编译。

1.1 获取 SDK

首先准备一个空文件夹用于存放 SDK，建议在 home 目录下，本文以`~/proj`为例

不要在虚拟机共享文件夹以及非英文目录存放、解压 SDK，避免产生不必要的错误

SDK 包可以从公司官网的下载界面获取。

```
#解压
mkdir ~/proj
cd ~/proj

//此 sdk 名称 与你下载的可能不同，根据获取的 sdk 解压
tar -xf xpi-566-debain10-***.tar
```

1.2 安装依赖包

编译 SDK 环境搭建所依赖的软件包安装命令如下：

```
sudo apt-get install repo git ssh make gcc libssl-dev liblz4-tool \
expect g++ patchelf chrpath gawk texinfo chrpath diffstat binfmt-support \
qemu-user-static live-build bison flex fakeroot cmake gcc-multilib
g++-multilib \
unzip device-tree-compiler python-pip ncurses-dev pyelftools \
```

1.3 交叉编译工具链介绍

鉴于 Rockchip Buildroot SDK 目前只在 Linux 下编译，我们也仅提供了 Linux 下的交叉编译工具链。其中 UBoot 及 Kernel 使用的编译工具链预置目录在 prebuilt/gcc 下，buildroot 使用该开源软件中编译出来的工具链。

U-Boot 及 Kernel 编译工具链：

```
prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linuxgnu/bin/aarch64-linux-gnu
```

对应版本

```
gcc version 6.3.1 20170404 (Linaro GCC 6.3-2017.05)
```

Buildroot 编译工具链
64 位系统：
buildroot/output/rockchip_rk356x/host/bin/aarch64-buildroot-linux-gnu
32 位系统：
buildroot/output/rockchip_rk356x/host/usr/arm-linux-gcc
对应版本：

```
gcc version 9.3.0 (Buildroot 2018.02-rc3-02723-gd3fbc6ae13)
```

1.4 SDK 工程目录介绍

一个通用 Linux SDK 工程目录包含有 buildroot、debian、app、kernel、u-boot、device、docs、external 等

```
.
├── app
├── buildroot                                                    #Buildroot 根文件系统编译目录
├── build.sh -> device/rockchip/common/build.sh                  # 编译脚本
├── debian                                                       # Debian 根文件系统编译目录
├── device                                                       # 编译相关配置文件
├── docs                                                         # 文档
├── envsetup.sh -> buildroot/build/envsetup.sh
├── external
├── kernel                                                       # Kernel
├── Makefile -> buildroot/build/Makefile
├── mkfirmware.sh -> device/rockchip/common/mkfirmware.sh        # 链接脚本
├── prebuilts                                                    # 交叉编译工具链
├── rkbin
├── rkflash.sh -> device/rockchip/common/rkflash.sh              # 烧写脚本
├── rkbin                                                        #存放 Rockchip 相关的 Binary 和工具。
├──rockdev                                                       #存放编译输出固件
├── tools                                                        # 工具目录
└── u-boot                                                       # U-Boot
```

app： 存放上层应用 app，主要是 qcamera/qfm/qplayer/settings 等一些应用程序。
buildroot： 基于 buildroot (2018.02-rc3) 开发的根文件系统。
debian： 基于 debian 10 开发的根文件系统，支持部分芯片。
device/rockchip： 存放各芯片板级配置和 Parameter 文件，以及一些编译与打包固件的脚本和预备文件。
docs：存放芯片模块开发指导文档、平台支持列表、芯片平台相关文档、Linux 开发指南等。
IMAGE： 存放每次生成编译时间、XML、补丁和固件目录。
external： 存放第三方相关仓库,包括音频、视频、网络、recovery 等。
kernel： 存放 kernel 4.4 或 4.19 开发的代码。
prebuilts： 存放交叉编译工具链。
rkbin： 存放 Rockchip 相关的 Binary 和工具。
rockdev： 存放编译输出固件。
tools： 存放 Linux 和 Windows 操作系统环境下常用工具。
u-boot：存放基于 v2017.09 版本进行开发的 uboot 代码。

2. 代码编译

2.1 编译前选择配置文件

```
#SDK 根目录选择配置文件
./build.sh lunch 选择 3
```

```
Pick a defconfig:

1. rockchip_defconfig
2. rockchip_rk3566_evb2_lp4x_v10_32bit_defconfig
3. rockchip_rk3566_evb2_lp4x_v10_defconfig
4. rockchip_rk3568_evb1_ddr4_v10_32bit_defconfig
5. rockchip_rk3568_evb1_ddr4_v10_defconfig
6. rockchip_rk3568_uvc_evb1_ddr4_v10_defconfig
```

2.2 编译 SDK

SDK 编译有 2 种方式，一种是单独编译各个部分，一种是全自动编译。第一次编译可使用全部编译，再次编译时可根据修改单独编译修改的部分。

2.2.1 U-Boot 编译

进入 SDK 工程。运行如下命令进行编译

```
#./build.sh uboot
```

2.2.2 Kernel 编译

进入工程目录根目录执行以下命令自动完成 kernel 的编译及打包。

```
#./build.sh kernel
```

2.2.3 Debian 编译

```
./build.sh debian
```

2.2.4 全自动编译

完成上述 Kernel/U-Boot/Recovery/Rootfs 各个部分的编译后，进入工程目录根目录执行以下命令自动完成所有的编译：

```
export RK_ROOTFS_SYSTEM=debian
<SDK>$./build.sh all
```

2.3 固件打包

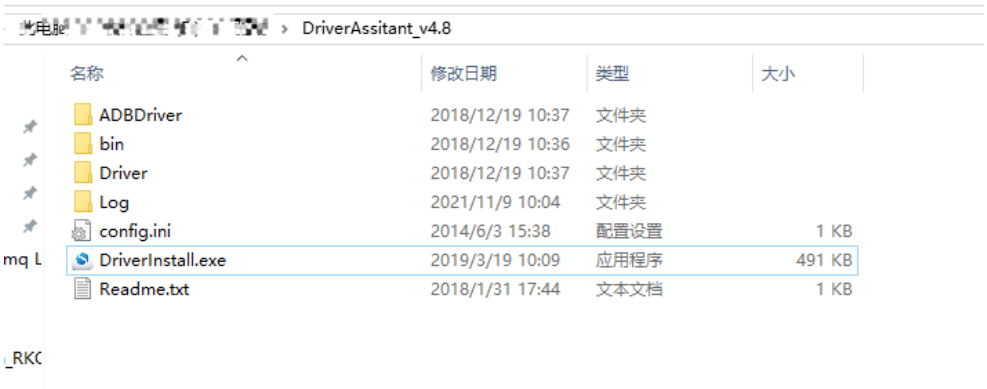
```
# 固件打包
./build.sh firmware

update.img 打包
./build.sh updateimg
```

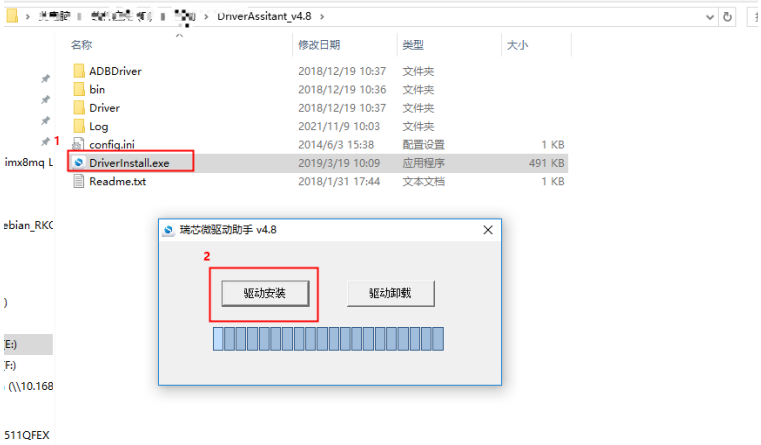
3. 固件烧写

3.1 安装烧写工具驱动:

DriverAssitant_v4.8.zip 见附件。



双击“DriverInstall.exe”应用程序，点击“驱动安装”

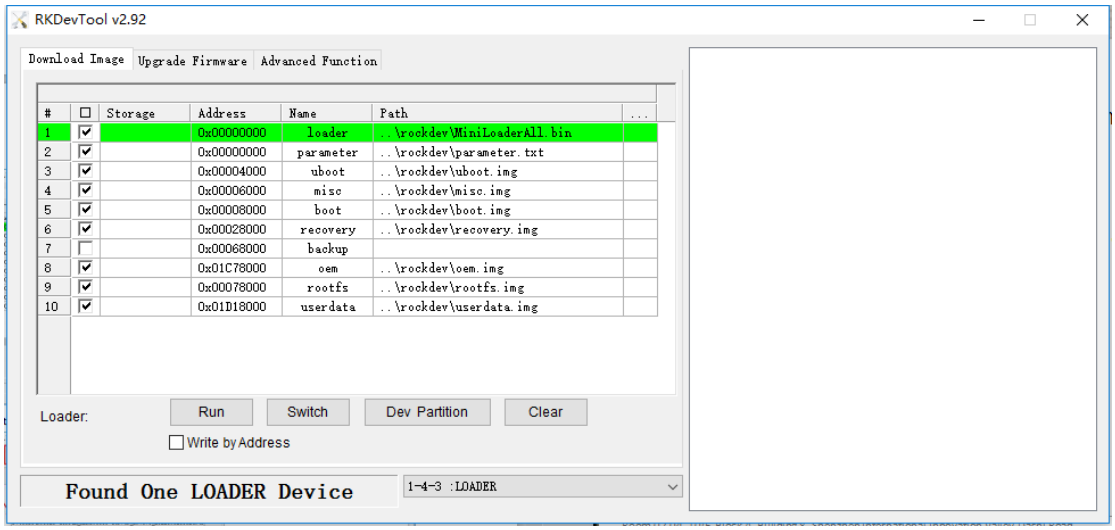


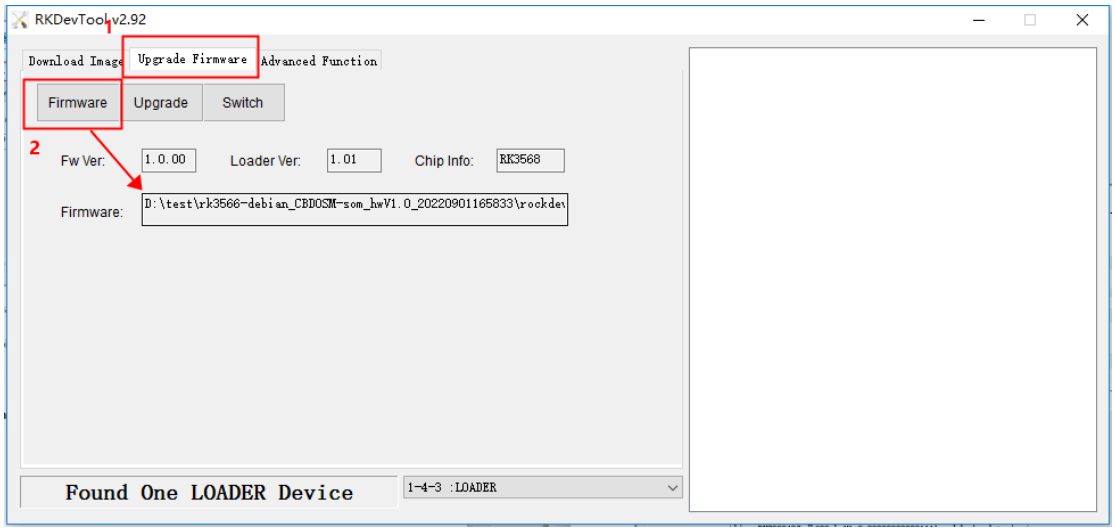
3.2 Windos 刷机说明

SDK 提供 Windows 烧写工具(工具版本需要 V2.55 或以上)，工具位于工程根目录：

```
<SDK>/Tools/windows/RKDevTool/
```

如下图，编译生成相应的固件后，设备烧写需要进入 MASKROM 或 BootROM 烧写模式，
连接好 USB 下载线后，按住按键“Upgrade”不放并插上电源，就能进入 MASKROM 模式，加载编译生成固件的相应路径后，点击“执行”进行烧写。


RKDevTool v2.92
Download Image
Upgrade Firmware
Advanced Function
Storage Address Name Path
1 0x00000000 loader ..\rockdev\MiniLoaderAll.bin
2 0x00000000 parameter ..\rockdev\parameter.txt
3 0x00004000 uboot ..\rockdev\uboot.img
4 0x00006000 misc ..\rockdev\misc.img
5 0x00008000 boot ..\rockdev\boot.img
6 0x00028000 recovery ..\rockdev\recovery.img
7 0x00068000 backup
8 0x01C78000 oem ..\rockdev\oem.img
9 0x00078000 rootfs ..\rockdev\rootfs.img
10 0x01D18000 userdata ..\rockdev\userdata.img
Loader:
Run
Switch
Dev Partition
Clear
Write by Address
Found One LOADER Device
1-4-3 :LOADER


RKDevTool v2.92
1
Download Image
Upgrade Firmware
Advanced Function
Firmware
Upgrade
Switch
2
Fw Ver:
1.0.00
Loader Ver:
1.01
Chip Info:
RK3568
Firmware:
D:\test\rk3566-debian_CBDOSM-som_hwV1.0_20220901165833\rockdev\
Found One LOADER Device
1-4-3 :LOADER

4. 软件开发

4.1 软件开发目录介绍

kerne 设备树目录：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip
```

u-boot 设备树目录：

```
<SDK>/u-boot/arch/arm/dts
```

defconfig 目录

```
<SDK>/kernel/arch/arm64/configs/rockchip_linux_defconfig
```

4.2 DTS 介绍

DTS 概述

早期版本的 Linux Kernel 是直接在板级配置文件配置板子相关的信息，如 IOMUX，默认拉高/低的 GPIO，每个 I2C/SPI 总线下的 client 设备信息。为了摒弃这种'hard code' 的方式，Linux 引入设备树 （Device Tree）的概念来描述不同的硬件结构。

Device Tree 数据可读性较高，遵循 DTS 规范，通常被描述在.dtsi 和.dts 源文件。在内核编译的过程中， 被编译为.dtb 的二进制文件。在开机启动阶段，dtb 会被 bootloader（如 U-Boot）加载到 RAM 的某个地 址空间，并且将该地址作为参数传递给 Kernel space。内核解析整个 dtb 文件，提炼每个设备信息以初始 化。 本文旨在介绍如何新增一个的板子 dts 配置以及一些常见的 dts 语法说明，关于更详细 dts 的 语法介绍不在本文范围内，如有兴趣，请参考：devicetree-specifications 和 devicetree-bindings

Linux Kernel 目前支持多平台使用 dts，RK 平台的 dts 文件存放于：

```
ARM: arch/arm/boot/dts/
ARM64: arch/arm64/boot/dts/rockchip
```

如果硬件设计上是核心板和底板的结构，或者产品有多个产品形态，可以把公用的硬件描述放 在 dtsi 文件，而 dts 文件则描述不同的硬件模块，并且通过 include "xxx.dtsi"将公用的硬件描述 包含进来。

4.3 Kernel

4.3.1 内核定制

首先需要获取 SDK ，开发环境的准备和获取方法如第一章、第二章所示。
之后开始新增内核选项：

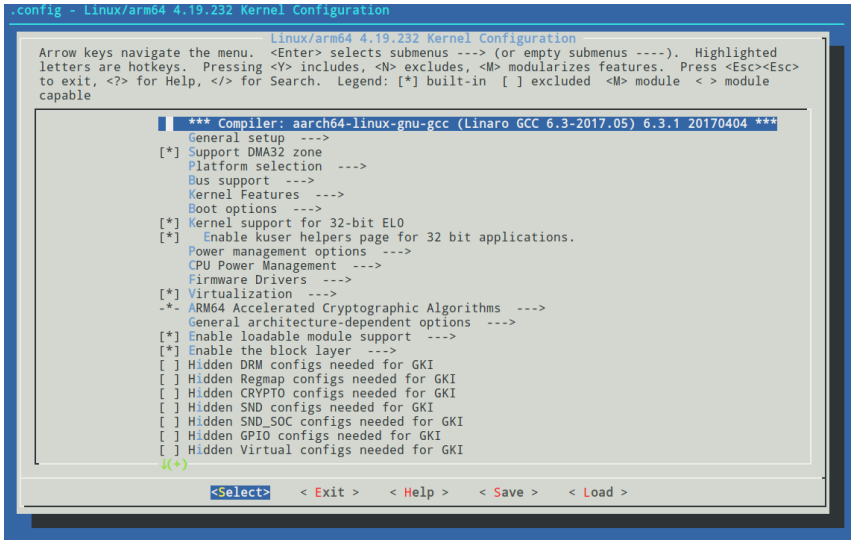
- 首先进入 SDK/kernel 文件夹，将配置文件写入 .config：

```
# Linux
make ARCH=arm64 rockchip_linux_defconfig
```

- 进入配置菜单

```
make ARCH=arm64 menuconfig
```

之后会进入一个图形化界面进行配置



- 使用介绍

1. 选项框内为星号 `*` 表示开启并编译进内核，空白表示不开启，`M` 表示开启并编译为模块
2. 选项框分为 `[]` 和 `<>`，`[]` 只能选择编译(按 Y)或者去除(按 N)，`<>` 除了选择编译或者去除外还可以选择编译为模块(按 M)；
3. 选项后面有 `–>`，说明在改目录下还有子目录；
4. 双击 `ESC` 表示退出，按下 `?` 按键可以显示帮助信息，按下 `/` 按键可以输入搜索内容来全局搜索信息；
5. 上下左右四个方向键是移动光标，`Enter` 是选中；

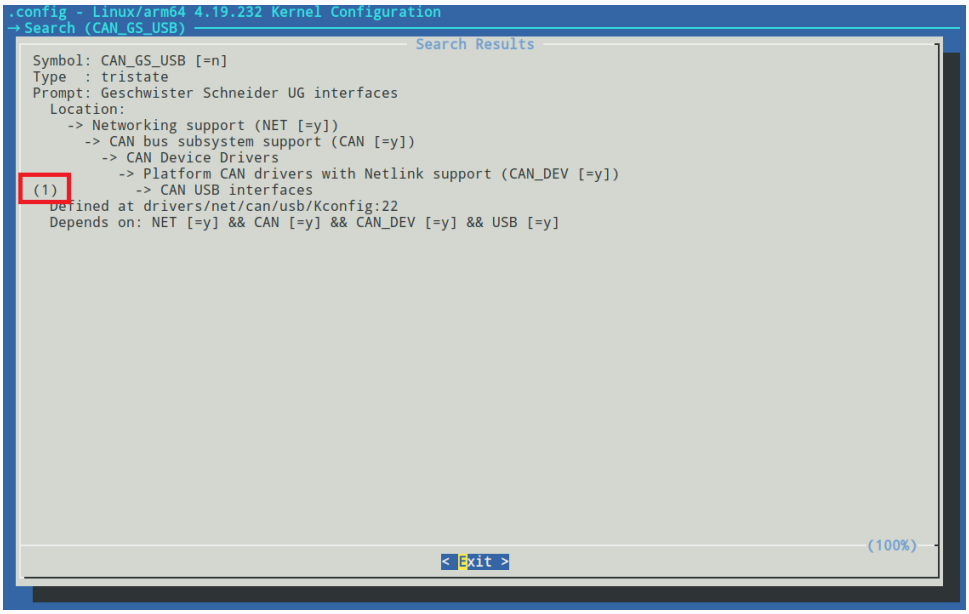
- 注意事项

选择编译为模块 (M) 需要后续进行安装才能正常使用，所以添加少量配置建议使用编译进内核方式 (Y)，省去安装步骤

- 保存

打开需要的配置后，按方向键左右将光标移动到 Save 并按下 3 次回车进行保存之后，持续双击 Esc 退出，退出后将修改内容保存到配置文件：

```
make ARCH=arm64 savedefconfig
mv defconfig arch/arm64/configs/rockchip_linux_defconfig
```



4.3.2 内核编译

回到 SDK 根目录进行编译：

```
# 编译内核
./build.sh kernel
```

生成的文件位置：`SDK/kernel/boot.img`

4.4 GPIO

4.4.1 简介

GPIO，全称 General-Purpose Input/Output（通用输入输出），是一种软件运行期间能够动态配置和控制的通用引脚。 所有的 GPIO 在上电后的初始状态都是输入模式，可以通过软件设为上拉或下拉，也可以设置为中断脚，驱动强度都是可编程的,其核心是填充 GPIO bank 的方法和参数，并调用 gpiochip_add 注册到内核中。

XPI-3566-ZERO 有 14pin 是 GPIO 口：各个 GPIO 的使用情况如下表：

Port	Pin name	Device
GPIO0	GPIO0_A0_d/ REFCLK_OUT GPIO0_C6_d/SPI0_CS0_M0/ UART4_RTSn_M0 GPIO0_C4_d/ SPI0_CS1_M0	复用为其他功能
GPIO1	GPIO1_A7_d/SPI0_CLK_M0/ UART4_CTSn_M0	复用为其他功能
GPIO3	GPIO3_D6_d/UART1_TX_M1 GPIO3_D7_d/UART1_RX_M1 GPIO3_C7_d/I2S1_SCLK_TX_M1 GPIO3_D0_d/ I2S1_LRCK_TX_M1 GPIO3_D2_d/ I2S1_SDI0_M1 GPIO3_D1_d/ I2S1_SDO0_M1	复用为其他功能
GPIO4	GPIO4_C1_d/UART1_CTSn_M1 GPIO4_B6_d/ UART1_RTSn_M1 GPIO4_C4_d/ GPIO4_C4 GPIO4_A7_d/SPI3_CS1_M0	复用为其他功能

常用以下公式计算引脚：

```
GPIO pin 脚计算公式：pin = bank * 32 + number
GPIO 小组编号计算公式：number = group * 8 + X
```

下面演示 GPIO4_A7 pin 脚计算方法：

bank = 4;　　// GPIO4_A7 => 4, bank ∈ [0,4]

group = 0;　　// GPIO4_A7 => 0, group ∈ {(A=0), (B=1), (C=2), (D=3)}

X = 7;　　// GPIO4_A7 => 7, X ∈ [0,7]

number = group * 8 + X = 0 * 8 + 7 = 7

pin = bank*32 + number= 4 * 32 + 7 = 135;

GPIO4_A7 对应的设备树属性描述为:<&gpio4 7 IRQ_TYPE_EDGE_RISING>,由 `kernel/include/dt-bindings/pinctrl/rockchip.h` 的宏定义可知，也可以将 GPIO4_A7 描述为<&gpio4 RK_PA7 IRQ_TYPE_EDGE_RISING>。

```
#define RK_PA0              0
#define RK_PA1              1
#define RK_PA2              2
#define RK_PA3              3
#define RK_PA4              4
#define RK_PA5              5
#define RK_PA6              6
#define RK_PA7              7
#define RK_PB0              8
#define RK_PB1              9
#define RK_PB2              10
#define RK_PB3              11
......
```

cat /sys/kernel/debug/gpio

4.4.2 用户空间控制 GPIO

GPIO4_A7 可能被其他功能占用，我们可以使用
命令查看哪些 GPIO 已经被占用。当引脚没有被其它外设复用时，我们可以通过 export 导出该引脚去使用

```
root@linaro-alip: # cat /sys/kernel/debug/gpio
gpiochip0: GPIOs 0-31, parent: platform/fdd60000.gpio, gpio0:
 gpio-0   (                    |work               ) out lo
 gpio-5   (                    |vcc5v0_otg         ) out hi
 gpio-6   (                    |vcc5v0_host        ) out hi
 gpio-8   (                    |vcc_camera         ) out hi
 gpio-16  (                    |bt_default_wake    ) out hi
 gpio-18  (                    |pcie20_3v3         ) out hi
 gpio-23  (                    |vcc3v3_lcd1_n      ) out hi
 gpio-28  (                    |vcc_wifi_en        ) out hi
 gpio-30  (                    |reset              ) out hi

gpiochip1: GPIOs 32-63, parent: platform/fe740000.gpio, gpio1:
 gpio-42  (                    |reset              ) out hi

gpiochip2: GPIOs 64-95, parent: platform/fe750000.gpio, gpio2:
 gpio-77  (                    |bt_default_rts     ) out lo
 gpio-79  (                    |bt_default_reset   ) out hi
 gpio-80  (                    |bt_default_wake_host) in  lo

gpiochip3: GPIOs 96-127, parent: platform/fe760000.gpio, gpio3:
 gpio-119 (                    |reset              ) out lo
 gpio-121 (                    |vcc3v3_panel_n     ) out hi

gpiochip4: GPIOs 128-159, parent: platform/fe770000.gpio, gpio4:
 gpio-137 (                    |reset              ) out lo
 gpio-146 (                    |mdio-reset         ) out hi
 gpio-148 (                    |spk                ) out lo

gpiochip5: GPIOs 255-255, parent: platform/rk805-pinctrl, rk817-gpio, can sleep:
root@linaro-alip:~# [   32.067054] vcc3v3_lcd0_n: disabling
[   32.067172] vcc3v3_panel_n: disabling
[   32.067220] pcie20_3v3: disabling
```

配置 GPIO 为一般的输入输出口的步骤如下：
步骤一、在控制台使用 echo 命令将要操作的 GPIO 编号 export：
echo N > /sys/class/gpio/export
步骤二、在控制台使用 echo 命令设置 GPIO 方向：
1、对于输入
echo in > /sys/class/gpio/gpioN/direction
2、对于输出
echo out > /sys/class/gpio/gpioN/direction
3、可使用 cat 命令查看 GPIO 方向
cat /sys/class/gpio/gpioN/direction
步骤三、在控制台使用 cat 或 echo 命令查看 GPIO 输入值或设置 GPIO 输出值：
1. 查看输入值
cat /sys/class/gpio/gpioN/value
2. 输出低
echo 0 > /sys/class/gpio/gpioN/value
3. 输出高
echo 1 > /sys/class/gpio/gpioN/value

4.4.3 驱动控制 GPIO

本文以 GPIO0_A0 和 GPIO4_A7 这两个 GPIO 口为例写了一份简单操作 GPIO 口的驱动，在 SDK 的路径为：kernel/drivers/gpio/gpio-control.c，以下就以该驱动为例介绍 GPIO 的操作。

A. GPIO 当做普通输入输出

首先在 DTS 文件中增加驱动的资源描述：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3568-evb.dtsi

gpio_group{
        compatible = "gpio-group";
        pinctrl-names = "gpio-lte";
        status = "okay";
        gtc-gpios = <&gpio4 7 GPIO_ACTIVE_HIGH>;
    };
```

然后配置 deconfig 文件，使其能编译到 gpio-control.c 文件：

```
SDK>/kernel/arch/arm64/configs/rockchip_linux_defconfig

CONFIG_GPIO_CONTROL=y
```

这里定义了一个脚作为一般的输出输入口： gtc-gpios GPIO4_A7 GPIO_ACTIVE_HIGH 表示高电平有效，如果想要低电平有效，可以改为：GPIO_ACTIVE_LOW，这个属性将被驱动所读取。

然后在 probe 函数中对 DTS 所添加的资源进行解析，代码如下：

```
static int gpio_irc_probe(struct platform_device *pdev)
{
    int ret;
    int gpio, gpio_enable_value;
    enum of_gpio_flags flag;
    struct device_node *gtc_gpio_node = pdev->dev.of_node;

    printk("gtc GPIO Test Program Probe\n");

    gpio = of_get_named_gpio_flags(gtc_gpio_node, "gtc-gpios", 0, &flag);
    if (!gpio_is_valid(gpio)) {
        printk("gtc-gpios: %d is invalid\n", gpio);
        return -ENODEV;
    }

    if (gpio_request(gpio, "gtc-gpios")) {
        printk("gpio %d request failed!\n", gpio);
        gpio_free(gpio);
        return -ENODEV;
    }
        gpio_enable_value = (flag == OF_GPIO_ACTIVE_LOW) ? 0:1;
    gpio_direction_output(gpio, gpio_enable_value);
    printk("gtc gpio putout\n");
    ...
}
注：若原先 gpio_irc_probe 函数有程序，可删除原 gpio_irc_probe 函数内容。
```

of_get_named_gpio_flags 从设备树中读取 gtc-gpios 的 GPIO 配置编号和标志，gpio_is_valid 判断该 GPIO 编号是否有效，gpio_request 则申请占用该 GPIO。如果初始化过程出错，需要调用 gpio_free 来释放之前申请过且成功的 GPIO 。在驱动中调用 gpio_direction_output 就可以设置输出高还是低电平，这里默认输出从 DTS 获取得到的有效电平 GPIO_ACTIVE_HIGH，即为高电平，如果驱动正常工作，可以用万用表测得对应的引脚应该为高电平。

实际中如果要读出 GPIO，需要先设置成输入模式，然后再读取值：

```
int val;
gpio_direction_input(your_gpio);
val = gpio_get_value(your_gpio);
```

下面是常用的 GPIO API 定义：

```
#include <linux/gpio.h>
#include <linux/of_gpio.h>

enum of_gpio_flags {
     OF_GPIO_ACTIVE_LOW = 0x1,
};
int of_get_named_gpio_flags(struct device_node *np, const char *propname,
int index, enum of_gpio_flags *flags);
int gpio_is_valid(int gpio);
int gpio_request(unsigned gpio, const char *label);
void gpio_free(unsigned gpio);
int gpio_direction_input(int gpio);
int gpio_direction_output(int gpio, int v);
```

B. GPIO 当做中断脚使用

GPIO 口的中断使用与 GPIO 的输入输出类似，首先在 DTS 文件中增加驱动的资源描述：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3568-evb.dtsi

gpio_group{
        compatible = "gpio-group";
        pinctrl-names = "gpio-lte";
        status = "okay";
       gtc-irq-gpio = <&gpio4 7 IRQ_TYPE_EDGE_RISING>;  /* GPIO4_A7 */
    };
```

IRQ_TYPE_EDGE_RISING 表示中断由上升沿触发，当该引脚接收到上升沿信号时可以触发中断函数。 这里还可以配置成如下：

```
IRQ_TYPE_NONE          //默认值，无定义中断触发类型
IRQ_TYPE_EDGE_RISING   //上升沿触发
IRQ_TYPE_EDGE_FALLING  //下降沿触发
IRQ_TYPE_EDGE_BOTH     //上升沿和下降沿都触发
IRQ_TYPE_LEVEL_HIGH    //高电平触发
IRQ_TYPE_LEVEL_LOW     //低电平触发
```

然后在 probe 函数中对 DTS 所添加的资源进行解析，再做中断的注册申请，代码如下：

```
static int gpio_irc_probe(struct platform_device *pdev)
{
    int ret;
    int gpio, gtc_irq_gpio, gtc_irq;
    enum of_gpio_flags flag;
    struct device_node *gtc_gpio_node = pdev->dev.of_node;
    ...
    gtc_irq_gpio = gpio;
    gtc_irq = gpio_to_irq(gtc_irq_gpio);
    if (gtc_irq) {
        if (gpio_request(gpio, "gtc-irq-gpio")) {
            printk("gpio %d request failed!\n", gpio);
            gpio_free(gpio);
            return IRQ_NONE;
        }
        ret = request_irq(gtc_irq, gtc_gpio_irq, flag, "gtc-gpio", NULL);
        if (ret != 0)
            free_irq(gtc_irq, NULL);
    }
        return 0;
}
static irqreturn_t gtc_gpio_irq(int irq, void *dev_id) //中断函数
{
    printk("Enter gtc gpio irq test program!\n");
    return IRQ_HANDLED;
}
```

调用 gpio_to_irq 把 GPIO 的 PIN 值转换为相应的 IRQ 值，调用 gpio_request 申请占用该 IO 口，调用 request_irq 申请中断，如果失败要调用 free_irq 释放，该函数中 gtc_irq 是要申请的硬件中断号，gtc_gpio_irq 是中断函数，flag 是中断处理的属性，gtc-gpio 是设备驱动程序名称。

4.5 I2C

4.5.1 简介

XPI-3566-ZERO 开发板上有 4 个 PIN 是 I2C 控制器，各个 I2C 的使用情况如下表：

Port	Pin name	Device
I2C1	GPIO0_B4_u / I2C1_SDA GPIO0_B3_u / I2C1_SCL	复用为其他功能
I2C3	GPIO1_A0_u/I2C3_SDA_M0 GPIO1_A1_u/ I2C3_SCL_M0	复用为其他功能

本章主要描述如何在该开发板上配置 I2C。

配置 I2C 可分为两大步骤：

定义和注册 I2C 设备

定义和注册 I2C 驱动

下面以配置 at24c02 为例,驱动所在位置：kernel/drivers/misc/eeprom/at24.c。

4.5.2 定义和注册 I2C 设备

在注册 I2C 设备时，需要结构体 i2c_client 来描述 I2C 设备。然而在标准 Linux 中，用户只需要提供相应的 I2C 设备信息，Linux 就会根据所提供的信息构造 i2c_client 结构体。

用户所提供的 I2C 设备信息以节点的形式写到 DTS 文件中，如下所示：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3568-evb.dtsi
&i2c0 {
        status = "okay";
        eeprom@50 {
        compatible = "atmel,24c02";
        reg = <0x50>;
        status="okay";
    };
};
```

配置好 dts 节点后，需要配置 defconfig 文件，使其编译 rtc 驱动程序：

```
<SDK> /kernel/arch/arm64/configs/ rockchip_linux_defconfig

CONFIG_EEPROM_AT24=y
```

4.5.3 定义和注册 I2C 驱动

A. 定义 I2C 驱动

在定义 I2C 驱动之前，用户首先要定义变量 `of_device_id` 和 `i2c_device_id`。
`of_device_id` 用于在驱动中调用 DTS 文件中定义的设备信息，其定义如下所示：

```
<sdk>/kernel/drivers/misc/eeprom/at24.c

static const struct of_device_id at24_of_match[] = {
    { .compatible = "atmel,24c02" },
    {}
};
MODULE_DEVICE_TABLE(of, at24_of_match);
```

定义变量 i2c_device_id：

```
static const struct i2c_device_id at24_ids[] = {
    { "24c02",      (kernel_ulong_t)&at24_data_24c02 },
    { }
};
MODULE_DEVICE_TABLE(i2c, at24_ids);
```

i2c_driver 如下所示：

```
static struct i2c_driver at24_driver = {
        .driver = {
                .name = "at24",
                .pm = &at24_pm_ops,
                .of_match_table = at24_of_match,
                .acpi_match_table = ACPI_PTR(at24_acpi_ids),
        },
        .probe_new = at24_probe,
        .remove = at24_remove,
        .id_table = at24_ids,
};
```

注：变量 id_table 指示该驱动所支持的设备。

B. 注册 I2C 驱动

使用 i2c_register_driver 函数注册 I2C 驱动。

```
module_i2c_driver(at24_driver);
#module_i2c_driver 为宏定义，可以展开为：
static int __init at24_init(void)
{
        at24_io_limit = rounddown_pow_of_two(at24_io_limit);
        return i2c_add_driver(&at24_driver);
}
module_init(at24_init);
static void __exit at24_exit(void)
{
        i2c_del_driver(&at24_driver);
}
module_exit(at24_exit);
```

在调用 i2c_register_driver 注册 I2C 驱动时，会遍历 I2C 设备，如果该驱动支持所遍历到的设备，则会调用该驱动的 probe 函数。

4.5.4 通过 I2C 收发数据

在注册好 I2C 驱动后，即可进行 I2C 通讯。
向从机发送信息：

```
int i2c_master_send(const struct i2c_client *client, const char *buf, int count)
 {
     int ret;
     struct i2c_adapter *adap = client->adapter;
     struct i2c_msg msg;
     msg.addr = client->addr;
     msg.flags = client->flags & I2C_M_TEN;
     msg.len = count;
     msg.buf = (char *)buf;
     ret = i2c_transfer(adap, &msg, 1);
     /*
      * If everything went ok (i.e. 1 msg transmitted), return #bytes
      * transmitted, else error code.
      */
    return (ret == 1) ? count : ret;
 }
```

向从机读取信息：

```
int i2c_master_recv(const struct i2c_client *client, char *buf, int count)
 {
     struct i2c_adapter *adap = client->adapter;
     struct i2c_msg msg;
     int ret;
     msg.addr = client->addr;
     msg.flags = client->flags & I2C_M_TEN;
     msg.flags |= I2C_M_RD;
     msg.len = count;
     msg.buf = buf;
     ret = i2c_transfer(adap, &msg, 1);
     /*
      * If everything went ok (i.e. 1 msg received), return #bytes received,
      * else error code.
      */
     return (ret == 1) ? count : ret;
 }
```

4.5.5 测试 I2C 设备

可以通过 i2ctool 测试 i2c 设备是否注册成功，步骤如下：

a. 输入命令更新软件库：sudo apt-get update

```
root@linaro-alip:~#
root@linaro-alip:~# sudo apt-get update
```

b. 下载 i2ctool 工具: sudo apt-get install i2c-tools

```
root@linaro-alip:~# sudo apt-get install i2c-tools
```

c. 查看使用的 i2c 下是否有注册设备的地址: i2cdetect -y 0

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- UU -- -- --
20: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

可看到 i2c0 下注册有一个地址为 0x50 的设备。

4.6 UART

4.6.1 简介

XPI-3566-ZERO 开发板上有 6 个 pin 是 uart 控制器，各个 uart 的使用情况如下表：

Port	Pin name	Device
Uart4	SPI0_MOSI_M0/UART4_TX_M0 SPI0_MISO_M0/UART4_RX_M0	复用为其他功能
Uart7	GPIO4_A3_d / UART7_RX_M2 GPIO4_A2_d / UART7_TX_M2	调试窗串口
Uart9	GPIO4_C5_d/ UART9_TX_M1-PWM12_M1 GPIO4_C6_d/ UART9_RX_M1-PWM13_M1	复用为其他功能

注：其它 uart 引脚配置可以在
<sdk> /kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl.dtsi 文件中查看。

本章主要描述如何在该开发板上配置 uart。

下面以配置 uart4 和板卡 debug 口(调试串口)为例。

4.6.2 配置 UART4 接口

a. 确定 uart4 使用的引脚，若 uart4 使用引脚：
GPIO1_A4、GPIO1_A6、GPIO1_A7、GPIO1_A5
b. 在 rk3568-pinctrl.dtsi 文件中查找上边引脚标号：

```
uart4m0_xfer: uart4m0-xfer {
                        rockchip,pins =
                                /* uart4_rxm0 */
                                <1 RK_PA4 2 &pcfg_pull_up>,
                                /* uart4_txm0 */
                                <1 RK_PA6 2 &pcfg_pull_up>;
                };
uart4m0_ctsn: uart4m0-ctsn {
                        rockchip,pins =
                                /* uart4m0_ctsn */
                                <1 RK_PA7 2 &pcfg_pull_none>;
                };
uart4m0_rtsn: uart4m0-rtsn {
                        rockchip,pins =
                                /* uart4m0_rtsn */
                                <1 RK_PA5 2 &pcfg_pull_none>;
                };
```

c. 配置 dts 文件

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3566-evb2-lp4x-v10.dtsi

&uart4 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&uart4m0_xfer &uart4m0_ctsn>;
};
```

4.6.3 配置调试串口

a. 找到板卡使用的 debug 及使用的引脚。
b. 在 dts 文件 rk3568-linux.dtsi 中添加：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3568-linux.dtsi

fiq-debugger {
        compatible = "rockchip,fiq-debugger";
        rockchip,serial-id = <7>;     //使用的串口号
        rockchip,wake-irq = <0>;
        /* If enable uart uses irq instead of fiq */
        rockchip,irq-mode-enable = <1>;
        rockchip,baudrate = <1500000>;  /* Only 115200 and 1500000 */
        interrupts = <GIC_SPI 252 IRQ_TYPE_LEVEL_LOW>;
        pinctrl-names = "default";
        pinctrl-0 = <&uart7m2_xfer>;  //串口使用的引脚
        status = "okay";
    };

注：uart7m2_xfer 在 rk3568-pinctrl.dtsi 文件定义的。
```

c. 关闭 uart7 功能(做调试串口时，要关闭 uart7)：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3566-evb2-lp4x-v10.dtsi

&uart7{
    status = "disabled";
};
```

注：此调试串口默认已配置完成。

4.7 SPI

4.7.1. 简介

XPI-3566-ZERO 开发板上有 4 个 PIN 是 SPI3，各个 spi 的使用情况如下表：

Port	Pin name	Device
spi3	GPIO4_B2_d/ SPI3_MOSI_M0 GPIO4_B0_d/ SPI3_MISO_M0 GPIO4_B3_d/ SPI3_CLK_M0 GPIO4_A6_d/ SPI3_CS0_M0	复用为其他功能

注：其它 SPI 引脚配置可以在 <sdk>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl.dtsi 文件中查看。

本章主要描述如何在该开发板上配置 SPI。

下面以配置 SPI 测试程序 为例。

4.7.2 定义和注册 SPI 设备

在标准 Linux 中，用户只需要提供相应的 SPI 设备信息，Linux 就会根据所提供的信息构造 spi_client 结构体。
用户所提供的 spi 设备信息以节点的形式写到 DTS 文件中，如下所示：

```
<SDK>/kernel/arch/arm64/boot/dts/rockchip/rk3566-evb2-lp4x-v10.dtsi

spi3{
    status = "okay";
//  max-freq = <48000000>; /* spi internal clk, don't modify */
    pinctrl-names = "default";
    pinctrl-0 = <&spi3m0_pins>;
    cs-gpios = <&gpio4 6 0>;
    spi_dev@0 {
        compatible = "rockchip,spidev";
        reg = <0>;
        spi-max-frequency = <12000000>;
        spi-lsb-first;
        status = "okay";
    };
```

配置好 dts 节点后，需要配置 defconfig 文件，使其编译 spidev.c 驱动程序：

```
<SDK>/kernel/arch/arm64/configs/rockchip_linux_defconfig

CONFIG_SPI_SPIDEV=y
```

4.7.3 定义和注册 SPI 驱动

A. 定义 SPI 驱动

在定义 spi 驱动之前，用户首先要定义变量 of_device_id 。
of_device_id 用于在驱动中调用 DTS 文件中定义的设备信息，其定义如下所示：

```
<SDK/kernel/drivers/spi/spidev.c>

static const struct of_device_id spidev_dt_ids[] = {
    { .compatible = "rohm,dh2228fv" },
    { .compatible = "lineartechnology,ltc2488" },
    { .compatible = "ge,achc" },
    { .compatible = "semtech,sx1301" },
    { .compatible = "rockchip,spidev" },
    {},
};
MODULE_DEVICE_TABLE(of, spidev_dt_ids);
```

spi_driver 如下所示：

```
static struct spi_driver spidev_spi_driver = {
    .driver = {
        .name =     "spidev",
        .of_match_table = of_match_ptr(spidev_dt_ids),
        .acpi_match_table = ACPI_PTR(spidev_acpi_ids),
    },
    .probe =    spidev_probe,
    .remove =   spidev_remove,
};
```

B. 注册 SPI 驱动

使用 spi_register_driver 函数注册 SPI 驱动。

spi_register_driver(&spidev_spi_driver);

在调用 spi_register_driver 注册 SPI 驱动时，会遍历 SPI 设备，如果该驱动支持所遍历到的设备，则会调用该驱动的 probe 函数。

4.7.4 测试 SPI 设备:

查看 spi 节点是否注册成功: ls /dev

```
root@imx8mqevk:~# ls /dev
alpum           hugepages     memory_bandwidth   ptyp1   spidev1.0   tty
autofs          hwrng         mmcblk0            ptyp2   stderr      tty
block           i2c-0         mmcblk0boot0       ptyp3   stdin       tty
btrfs-control   i2c-1         mmcblk0boot1       ptyp4   stdout      tty
bus             i2c-2         mmcblk0p1          ptyp5   tee0        tty
cec0            initctl       mmcblk0p2          ptyp6   teepriv0    tty
char            input         mmcblk0p3          ptyp7   tty         tty
```

使用 kernel/tools/spi/spidev_test.c 测试程序。先查看此测试程序内打开的设备节点与注册的设备节点是否相同，不同则修改。

```
static const char *device = "/dev/spidev1.0";
static uint32_t mode;
```

复制到板卡并编译：gcc spidev_test.c -o spidev_test

短接 spi_miso 和 spi_mosi

输入命令：./spidev_test -v 直接比较出现的输入和输出否一致。

```
root@smarc-rzg2l:~# ./spidev_test -v
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
TX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF F0 0D  |
RX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF F0 0D  |
```