XPI-3566 Linux System Software Development Guide

Preface

Overview

This document serves as a software development guide for the XPI3566 Debian Linux system, and is intended to help software development engineers and technical support engineers get started with the development and debugging of the XPI-3566 more quickly.

Target Audience

This document (this guide) is mainly for the following engineers: Technical Support Engineers Software Development Engineer

Revision Record

Date	Version	Author	Modification instructions
23/2/7	V1.0	WangHx	Initial Version

Shenzhen Geniatech Inc., Ltd.
Address: Room 02-04, 10/F, Block A, Building 8, Shenzhen International Innovation Valley, Dashi Road, Nanshan District, Shenzhen, Guangdong, China Website： www.geniatech.com
Customer Service Tel： +86-0755-86028588
Technical support email： support @geniatech.com
Sales and service email： sales@geniatech.com

Index

1. Compiler Environment Setup

This chapter describes the compilation environment for the Linux SDK

Caution：

（1） It is recommended to develop on X86_64 Ubuntu 18.04. If you use other versions, you may need to adjust the compilation environment accordingly.

（2） Use the normal user to compile, not the root user privilege

1.1 Get the SDK

First prepare an empty folder for the SDK, we recommend the home directory, this article starts with ~/proj as an example

Do not store or extract the SDK in shared virtual machine folders and non-English directories to avoid unnecessary errors.

SDK packages can be obtained from the download screen on the company's website.

```
#Uncompress
mkdir ~/proj
cd ~/proj

//This sdk name may be different from the one you downloaded, unpack it according to the sdk you obtained
tar -xf xpi-566-debain10-***.tar
```

1.2 Installing dependency packages

The command to install the packages on which the compiled SDK environment is built is as follows： :

```
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 Introduction to the cross-compilation tool chain

Since the Rockchip Buildroot SDK currently only compiles under Linux, we only provide the cross-compilation toolchain for Linux. UBoot and Kernel use the prebuilt toolchain under prebuilt/gcc, buildroot uses the toolchain compiled from the open source software.

U-Boot and Kernel compilation toolchain：

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

Corresponding versions

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

Buildroot compilation toolchain

64-bit Systems：

buildroot/output/rockchip_rk356x/host/bin/aarch64-buildroot-linux-gnu

32-bit Systems：

buildroot/output/rockchip_rk356x/host/usr/arm-linux-gcc

Corresponding versions：

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

1.4 Introduction to the SDK project catalogue

A generic Linux SDK project directory contains buildroot, debian, app, kernel, u-boot, device, docs, external, etc.

```
.
├── app
├── buildroot                                                  #Buildroot The root filesystem
├── build.sh -> device/rockchip/common/build.sh            # Compile script
├── debian                                                     # Debian root filesystem build directory
├── device                                                 # Compile the relevant configuration files
├── docs                                                       # Docs
├── envsetup.sh -> buildroot/build/envsetup.sh
├── external
├── kernel                                                     # Kernel
├── Makefile -> buildroot/build/Makefile
├── mkfirmware.sh -> device/rockchip/common/mkfirmware.sh  # Link script
├── prebuilts                                                  # Cross-compilation toolchain
├── rkbin
├── rkflash.sh -> device/rockchip/common/rkflash.sh        # Burn script
├── rkbin                                                      # Store Rockchip related Binary and tools.
├──rockdev                                                     # store the compiled output firmware
├── tools                                                      # Tools directory
└── u-boot                                                     # U-Boot
```

app: Stores the upper-level application app, mainly qcamera/qfm/qplayer/settings and some other applications.

buildroot: Root filesystem based on buildroot (2018.02-rc3)

debian: Root filesystem based on debian 10, supports some chips

device/rockchip: Stores the board-level configuration and Parameter files for each chip, some scripts and preparation files for compiling and packaging firmware

docs: Stores chip module development guidance documents, platform support list, chip platform related documents, Linux Development guides etc

IMAGE: Stores the compile time, XML, patch and firmware directories for each generation

external: Store third-party related repositories, including audio, video, network, recovery, etc

kernel: Holds code developed for kernel 4.4 or 4.19

prebuilts: holds the cross-compilation toolchain

rkbin: Holds Rockchip related Binary and tools

rockdev: holds the compiled output firmware

tools: store common tools for Linux and Windows operating systems. u-boot: holds the uboot code based on the v2017.09 release

u-boot: Store uboot code based on the v2017.09 version

2. Code compilation

2.1 Selecting a configuration file before compiling

```
#SDK root directory selection profile
source lunch.sh Select item 7 in the options and choose the default configuration.
# Select rk3566-xpi
```

```
support project information
--> 1) for debian    rk3568-base
--> 2) for debian    rk3568-k3
--> 3) for debian    rk3568-ubuntu18.04
--> 4) for debian    rk3566-base
--> 5) for debian    rk3566-dk630
--> 6) for debian    rk3568-docker-openwrt
--> 7) for debian    rk3566-xpi
--> 8) for debian    rk3568-smarc
--> 9) for debian    rk3566-som
--> a) for ubuntu    dsrk3566-ubuntu18.04
--> b) for ubuntu    rk3568-vns-ubuntu18.04
--> c) for pi        rk3566-xpi-pi
no project select
```

2.2 Compile SDK

There are 2 ways to compile the SDK, one is to compile each part individually and one is to compile it fully automatically. The first time you compile, you can use the entire compilation, and when you compile again, you can compile the modified parts separately according to the changes.

2.2.1 U-Boot Compilation

Go to the SDK project. Run the following command to compile

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

2.2.2 Kernel Compilation

Go to the root of the project directory and execute the following command to compile and package the kernel automatically.

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

2.2.3 Debian Compilation

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

2.2.4 Fully automated compilation

After completing the compilation of each part of Kernel/UBoot/Recovery/Rootfs above, go to the root of the project directory and execute the following command to complete all the compilation automatically.

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

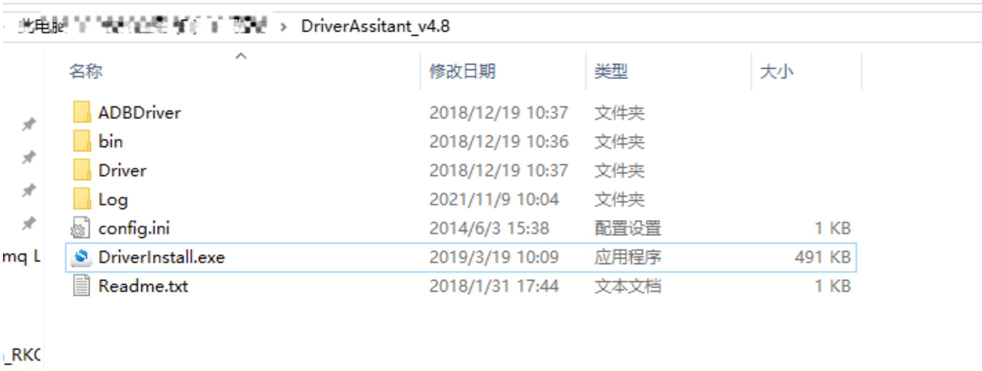
2.3 Firmware packaging

```
# Firmware packaging
. /build.sh firmware

update.img Package
. /build.sh updateimg

#generate release version firmware, path loong/out,longer
. /build.sh pack

Note: The compiled bulk package is available in rockdev/
Whole package in loong/out
```

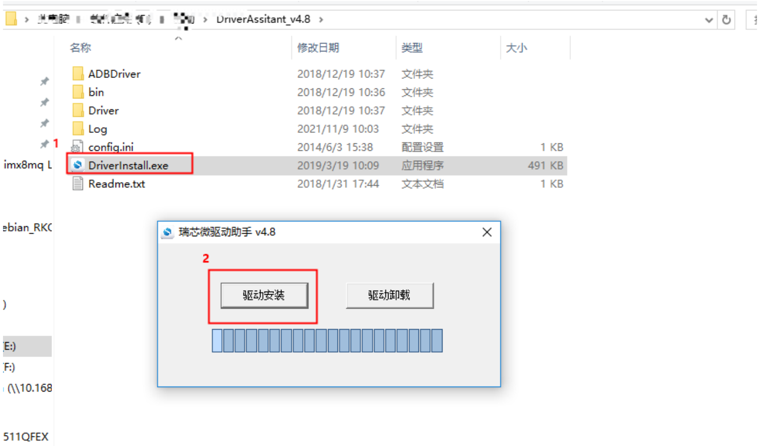
3. Firmware Upgrade

3.1 Install the Rockchip DriverAssitant driver.

DriverAssitant_v4.8.zip See attachment.



Double-click on the "DriverInstall.exe" application and click on "Driver Installation"



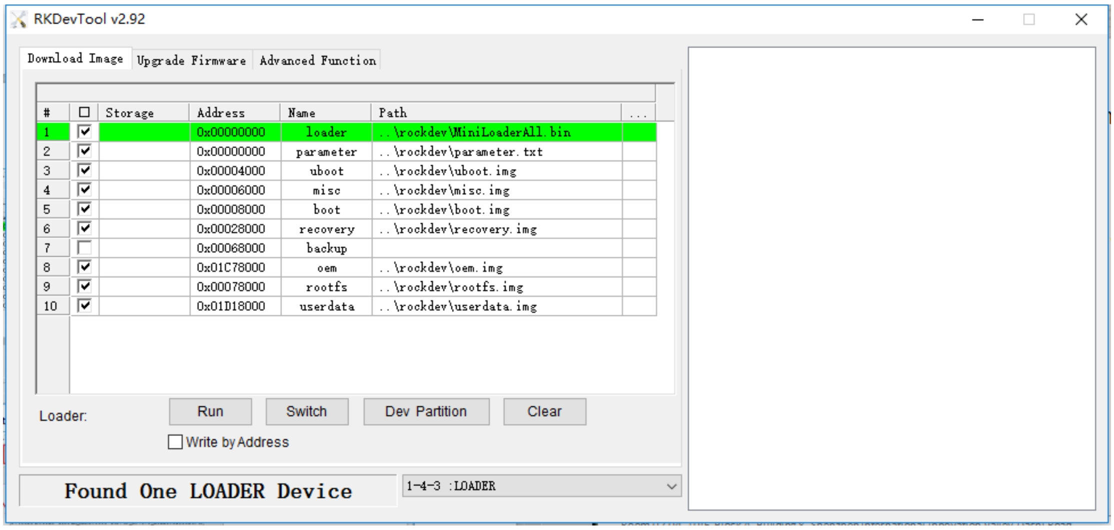
3.2 Windows Brush Instructions

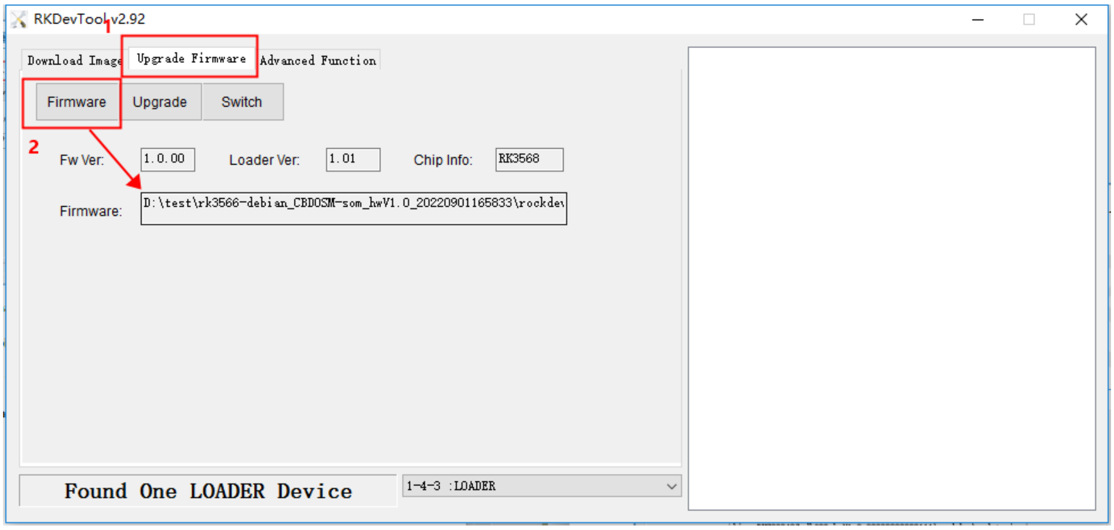
The SDK provides a Windows burn-in tool (tool version V2.55 or above is required), which is located in the project root directory at

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

As shown below, after compiling and generating the corresponding firmware, the device needs to be burned into MASKROM or BootROM mode.
Once the USB download cable is connected, press and hold the "Upgrade" button and plug in the power supply to access
In MASKROM mode, load the corresponding path of the compiled firmware, and

then click "Execute" to burn it.





4. Software Development

4.1 Software Development directory Introduction

Kernel device tree directory：

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/
```

u-boot device tree directory：

```
<SDK>/loong/devices/rk3566-xpi-debian10/u-boot/arch/arm/dts
```

defconfig directory

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/configs/r
ockchip_linux_defconfig
```

4.2 DTS Introduction

DTS Overview

Earlier versions of the Linux Kernel configured board-related information directly in board-level configuration files, such as IOMUX, default pull-high/low GPIOs and client device information under each I2C/SPI bus. In order to move away from this 'hard code' approach, Linux introduced the concept of a Device Tree to describe the different hardware structures.

Device Tree data is highly readable, follows the DTS specification and is usually described in the .dtsi and .dts source files. During kernel compilation, it is compiled into a .dtb binary file. During the boot phase, the dtb is loaded by the bootloader (e.g. U-Boot) into an address space in RAM and passed as an argument to the Kernel space. the kernel parses the entire dtb file and extracts information about each device to initialise it. The purpose of this article is to describe how to add a new board dts configuration and some common dts syntax descriptions, it is beyond the scope of this article to describe the syntax of more detailed dts. If interested, please refer to devicetree-specifications and devicetree-bindings.

The Linux Kernel currently supports multi-platform use of dts. The dts files for the RK platform are stored at：

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

If the hardware design is a core and backplane structure, or if the product has multiple product forms, the common hardware description can be placed in the dtsi file, while the dts file describes the different hardware modules and the common hardware description is included via include "xxx.dtsi".

4.3 Kernel

4.3.1 Kernel customization

First you need to obtain the SDK. The development environment is prepared and obtained as shown in chapters 1 and 2.

After that the new kernel options begin.

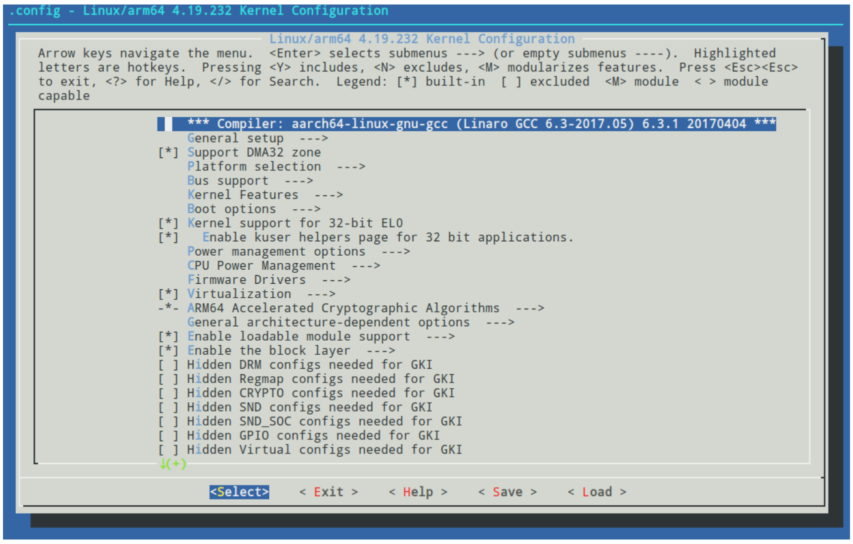
- First go to the SDK/kernel folder and write the configuration file to .config :

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

- Access to the configuration menu

```
make ARCH=arm64 menuconfig
```

This will then take you to a graphical interface for configuration



- Introduction to use

1. An asterisk * in the options box means on and compiled into the kernel, blank means not on, M means on and compiled as a module

2. The option boxes are divided into [] and <>, [] can only be selected for compilation (press Y) or removal (press N), <> in addition to compilation or removal

The option to compile as a module (press M) is also available.

3. An option followed by -> indicates that there are subdirectories under the change of directory.

4. Double click `ESC` to exit, press the `?` button to display help information, press the `/` button to enter a search to Global search information.

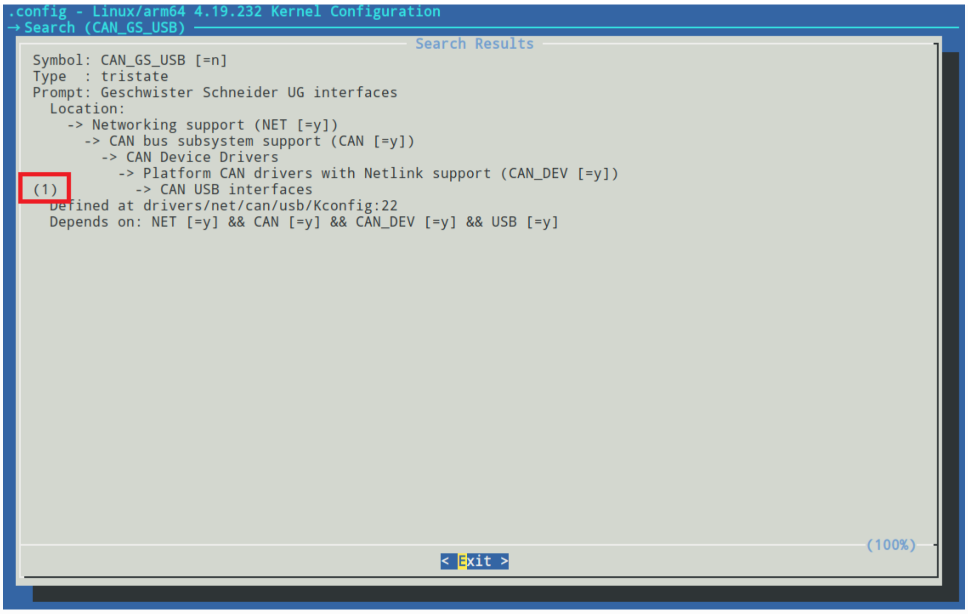
5. Up, down, left and right arrow keys to move the cursor and `Enter` to select.

- Cautions

The option to compile as a module `(M)` requires subsequent installation to work properly, so adding a small amount of configuration is recommended to use compile-in.

- Save

After opening the desired configuration, press the arrow keys left and right to move the cursor to Save and press Enter 3 times to save, then double-click Esc continuously to exit and save the changes to the configuration file.



4.3.2 Kernel Compilation

Go back to the SDK root directory to compile：

```
# Compile the kernel
./build.sh kernel
```

Location of the generated file：`SDK/kernel/boot.img`

4.4 GPIO

4.4.1 Introduction

GPIO, or General-Purpose Input/Output, is a general-purpose pin that can be dynamically configured and controlled during software operation. The initial state of all GPIOs after power-up is input mode, which can be set to pull-up or pull-down by software, or to interrupt pins, and the drive strength is programmable, with the core being t h e method and parameters to populate the GPIO bank and call gpiochip_add to register it to the kernel.

The XPI-3566 has 5 groups of GPIO banks: GPIO0~GPIO4, each group is distinguished by the numbers A0~A7, B0~B7, C0~C7, D0~D7, and the following formula is commonly used to calculate the pins： ：

```
GPIO pin pin calculation formula:
pin = bank * 32+ number
GPIO group number calculation formula:
number = group * 8 + X
```

The following demonstrates the calculation of the GPIO4_D5 pin：
bank = 4; //GPIO4_D5 => 4, bank ∈ [0,4]
group = 3; //GPIO4_D5 => 3, group ∈ {(A=0), (B=1), (C=2), (D=3)}
X = 5; //GPIO4_D5 => 5, X ∈ [0,7]
number = group * 8 + X = 3 * 8 + 5 = 29
pin = bank*32 + number= 4 * 32 + 29 = 157;

GPIO4_D5 corresponds to the device tree attribute described as:<&gpio4 29 IRQ_TYPE_EDGE_RISING>, by the Macro definitions of `kernel/include/dt-bindings/pinctrl/rockchip.h`， GPIO4_D5 can be described as <&gpio4 RK_PD5 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
......
```

4.4.2 User space control GPIO

GPIO4_D5 may be occupied by other functions and we can use command Cat sys/kernel/debug/gpio to see which GPIOs are already occupied. When pins are not multiplexed by other peripherals, we can use the Export to use the pin.

```
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
```

To configure the GPIO as a general input/output port, proceed as follows.
Step 1: Use the echo command on the console to export the

GPIO number to be operated on: `echo N > /sys/class/gpio/export`

Step 2: Use the echo command on the console to set the GPIO direction.
1. For input

```
echo in > /sys/class/gpio/gpioN/direction
```

2. For output

```
echo out > /sys/class/gpio/gpioN/direction
```

3. You can use the cat command to view the GPIO direction

```
cat /sys/class/gpio/gpioN/direction
```

Step 3: Use the cat or echo command on the console to view the GPIO input values or set the GPIO output values.
1. View input values

```
cat /sys/class/gpio/gpioN/value
```

2. Low output

```
echo 0 > /sys/class/gpio/gpioN/value
```

3. High output

```
echo 1 > /sys/class/gpio/gpioN/value
```

4.4.3 Driver control GPIO

In this article, we have written a simple driver to operate the GPIO ports GPIO0_A0 and GPIO2_B4 as an example, the path in the SDK is: kernel/drivers/gpio/gpio-control.c, the following is an example of GPIO operation.

A. GPIO as a normal input and output

First add the resource description of the driver to the DTS file：

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/rockchip/rk3568-evb.dtsi
    gpio_group{
        compatible = "gpio-group";
        pinctrl-names = "gpio-lte";
        status = "okay";
        gtc-gpios = <&gpio2 12  GPIO_ACTIVE_HIGH>;
   };
```

Then configure the deconfig file so that it compiles to the gpio-control.c file：

```
SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/configs
/rockchip_linux_defconfig

CONFIG_GPIO_CONTROL=y
```

Here a pin is defined as a general output input: gtc-gpios GPIO2_B4
GPIO_ACTIVE_HIGH means active high, if you want active low, you can change it to: GPIO_ACTIVE_LOW, this property will be read by the driver.

The resources added by the DTS are then parsed in the probe function, with the following code.

```
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");
    ...
}
```

Note: If the original gpio_irc_probe function has a program, you can delete the content of the original gpio_irc_probe function.

gpio_is_valid determines whether the GPIO number is valid or not, and gpio_request requests the GPIO. gpio_free is called to release the previously requested GPIO if the initialization process is wrong. If the initialization process is wrong, you need to call gpio_free to release the previously requested and successful GPIOs. The default output is GPIO_ACTIVE_HIGH, which is the valid level obtained from DTS, and the corresponding pin should be high if the driver works normally.

In practice, if you want to read out the GPIO, you need to set it to input mode first, and then read the value.

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

The following are common GPIO API definitions：

```
#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 Used as an interrupt pin

The interrupt usage of the GPIO port is similar to the input and output of the GPIO, first adding the resource description of the driver in the DTS file：

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts
/rockchip/rk3568-evb.dtsi

gpio_group{
        compatible = "gpio-group";
        pinctrl-names = "gpio-lte";
        status = "okay";
       gtc-irq-gpio = <&gpio4 29 IRQ_TYPE_EDGE_RISING>;   /* GPIO4_D5
*/
    };
```

IRQ_TYPE_EDGE_RISING indicates that the interrupt is triggered by a rising edge and that the interrupt function can be triggered when a rising edge signal is received on this pin. Here it can also be configured as follows.

```
IRQ_TYPE_NONE            // default, no interrupt trigger type defined
IRQ_TYPE_EDGE_RISING     //Rising edge triggering
IRQ_TYPE_EDGE_FALLING     //Falling edge triggered
IRQ_TYPE_EDGE_BOTH       //triggered on both rising and falling edges
IRQ_TYPE_LEVEL_HIGH      //triggered high
IRQ_TYPE_LEVEL_LOW       // triggered low
```

The resources added by the DTS are then resolved in the probe function, and then the interrupt registration application is made, with the following code：

```
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;
}
```

Call gpio_to_irq to convert the PIN value of the GPIO to the corresponding IRQ value, call gpio_request to request to occupy the IO port, call request_irq to request an interrupt, and call free_irq to release it if it fails. gtc_irq in this function is the hardware interrupt number to request, gtc_gpio_irq is the interrupt gtc_irq is the hardware interrupt number to apply, gtc_gpio_irq is the interrupt function, flag is the interrupt handling attribute, and gtc-gpio is the device driver name.

4.5 I2C

4.5.1 Introduction

There are 6 on-chip I2C controllers on the XPI-3566 development board, and the usage of each I2C is listed in the following table.

Port	Pin name	Device
I2C0	GPIO0_B1/I2C0_SCL GPIO0_B2/I2C0_SDA	RK809
I2C1	GPIO0_B3/I2C1_SCL GPIO0_B4/I2C1_SDA	pcf8563
I2C2_M0	GPIO0_B5/I2C2_SCL_M0 GPIO0_B6/I2C2_SDA_M0	Multiplexing to other functions
I2C2_M1	GPIO4_B5/I2C2_SCL_M1 GPIO4_B4/I2C2_SDA_M1	Camera/dsi
I2C3_M0	GPIO1_A1/I2C3_SCL_M0 GPIO1_A0/I2C3_SDA_M0	Multiplexing to other functions
I2C3_M1	GPIO3_B5/I2C3_SCL_M1 GPIO3_B6/I2C3_SDA_M1	Multiplexing to other functions
I2C4_M0	GPIO4_B3/I2C4_SCL_M0 GPIO4_B2/I2C4_SDA_M0	Multiplexing to other functions
I2C4_M1	GPIO2_B2/I2C4_SCL_M1 GPIO2_B1/I2C4_SDA_M1	Multiplexing to other functions
I2C5_M0	GPIO3_B3/I2C5_SCL_M0 GPIO3_B4/I2C5_SDA_M0	Multiplexing to other functions
I2C5_M0	GPIO4_C7/I2C5_SCL_M1 GPIO4_D0/I2C5_SDA_M1	Multiplexing to other functions

This chapter describes how to configure I2C on this development board.

Configuring I2C can be done in two major steps.

Defining and registering I2C devices

Defining and registering I2C drivers

The following is an example of configuring rtc-pcf8563, driver location: kernel/drivers/rtc/rtc-pcf8563.c.

4.5.2 Defining and registering I2C devices

When registering an I2C device, the structure i2c_client is needed to describe the I2C Simply provide the appropriate I2C device information and Linux will construct the i2c_client structure

The I2C device information provided by the user is written to the DTS file in the form of a node, as follows.

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts
/rockchip/rk3566-evb2-lp4x-v10.dtsi
&i2c1 {
        status = "okay";
        pcf8563_rtc: pcf8563@51 {
                compatible = "nxp,pcf8563";
                dev_name = "rtc_pcf8563";
                reg = <0x51>;
                init_date = "2015/01/01";
                status = "okay";
        };
};
```

After configuring the dts node, you need to configure the defconfig file so that it compiles the rtc driver:

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/configs/
rockchip_linux_defconfig

CONFIG_RTC_DRV_PCF8563=y
```

4.5.3 Defining and registering I2C drivers

A. Defining the I2C driver

Before defining the I2C driver, the user first defines the variables of_device_id and i2c_device_id. of_device_id is used to call the device information defined in the DTS file in the driver and is defined as follows ：

```
<sdk>/kernel/drivers/rtc/rtc-pcf8563.c

static const struct of_device_id pcf8563_of_match[] = {
    { .compatible = "nxp,pcf8563" },
    {}
};
MODULE_DEVICE_TABLE(of, pcf8563_of_match);
```

Define the variable i2c_device_id.:

```
static const struct i2c_device_id pcf8563_id[] = {
    { "pcf8563", 0 },
    { "rtc8564", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, pcf8563_id);
```

The i2c_driver is shown below:

```
static struct i2c_driver pcf8563_driver = {
    .driver     = {
        .name   = "rtc-pcf8563",
        .of_match_table = of_match_ptr(pcf8563_of_match),
    },
    .probe      = pcf8563_probe,
    .id_table   = pcf8563_id,
};
```

Note: The variable id_table indicates the devices supported by this driver.

B. Registering I2C drivers

Use the i2c_register_driver function to register the I2C driver.

```
module_i2c_driver(pcf8563_driver);
#module_i2c_driver module_i2c_driver is a macro definition that can be expanded to:
static int __init pcf8563_driver_init(void)
{
    return i2c_register_driver(&pcf8563_driver);
}
module_init(pcf8563_driver_init);
static void __exit pcf8563_driver_exit(void)
{
    i2c_del_driver (&pcf8563_driver);
}
module_exit(pcf8563_driver_exit);
```

When calling i2c_register_driver to register an I2C driver, it will traverse the I2C devices and call the probe function of the driver if it supports the traversed devices.

4.5.4 Sending and **receiving** data via **I2C**

After registering the I2C driver, you can perform I2C communication.
Send a message to the slave：

```
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;
 }
```

Reading information to the slave：

```
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;
 }
 EXPORT_SYMBOL(i2c_master_recv);
```

4.5.5 Testing I2C Devices

The i2ctool can be used to test whether the i2c device is successfully registered, as follows：

a. Enter the command to update the repository: sudo apt-get update

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

b. Download the i2ctool tool: sudo apt-get install i2c-tools

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

c. Check if the address of the registered device is available under the used i2c: i2cdetect - y 1

```
root@linaro-alip:~# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- UU -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
root@linaro-alip:~#
root@linaro-alip:~#
```

You can see that there is a device registered under i2c1 with address 0x51.

4.6 UART

4.6.1 Introduction

There are 10 on-chip uart controllers on the RK3566-XPI development board, and the usage of each uart is listed in the following table：

Port	Pin name	Device
uart0	GPIO0_C0/uart0_rx GPIO0_C1/uart0_tx GPIO0_C7/uart0_ctsn GPIO0_C4/uart0_rtsn	Multiplexing to other functions
Uart1_m0	GPIO2_B3/uart2_rx GPIO2_B4/uart2_tx GPIO2_B6/uart2_ctsn GPIO2_B5/uart2_rtsn	Multiplexing to other functions
Uart1_m1	GPIO3_D7/uart1_rx GPIO3_D6/uart1_tx GPIO4_C1/uart1_ctsn GPIO4_B6/uart1_rtsn	Multiplexing to other functions
Uart2_m0	GPIO0_D0/uart2_rx GPIO0_D1/uart2_tx	debug
Uart2_m1	GPIO1_D6/uart2_rx GPIO1_D5/uart2_tx	Multiplexing to other functions
Uart3_m0	GPIO1_A0/uart3_rx GPIO1_A1/uart3_tx GPIO1_A3/uart3_ctsn GPIO1_A2/uart3_rtsn	Multiplexing to other functions
......		Multiplexing to other functions
Uart5_m1	GPIO3_C3/uart5_rx GPIO3_C2/uart5_tx	Multiplexing to other functions
......		Multiplexing to other functions

Note: Other uart pin configurations can be found in the /loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl. dtsi file to view.

This chapter describes how to configure uart on this development board.

The following is an example of configuring uart1 and the debug port (debug serial port) of the board.

4.6.2 Configuring the UART1 Interface

a. Determine the pins used by uart1, if uart1 uses the pins: GPIO2_B3, GPIO2_B4, GPIO2_B6， GPIO2_B5

b. In the rk3568-pinctrl.dtsi file, look for the upper pin marker：

```
uart1m0_xfer: uart1m0-xfer {
            rockchip,pins =
                /* uart1_rxm0 */
                <2 RK_PB3 2 &pcfg_pull_up>,
                /* uart1_txm0 */
                <2 RK_PB4 2 &pcfg_pull_up>;
        };

 uart1m0_ctsn: uart1m0-ctsn {
            rockchip,pins =
                /* uart1m0_ctsn */
                <2 RK_PB6 2 &pcfg_pull_none>;
        };

 uart1m0_rtsn: uart1m0-rtsn {
            rockchip,pins =
                /* uart1m0_rtsn */
              <2 RK_PB5 2 &pcfg_pull_none>;
        };
```

c. Configure the dts file

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts
/rockchip/rk3566-evb2-lp4x-v10.dtsi

&uart1 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&uart1m0_xfer &uart1m0_ctsn>;
};
```

4.6.3 Configure debug serial port

a. Find the debug used by the board and the pins used.
b. In the dts file rk3568-linux.dtsi, add the following code：

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dt
s/rockchip/rk3568-linux.dtsi

fiq-debugger {
        compatible = "rockchip,fiq-debugger";
        rockchip,serial-id = <2>;      // serial port
        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 = <&uart2m0_xfer>;   // pin used for serial port
        status = "okay";
    };

Note: uart2m0_xfer defined in the rk3568-pinctrl.dtsi file.
```

c. Disable uart2 function (to disable uart2 when doing debugging of serial ports)：

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dt
s/rockchip/rk3566-evb2-lp4x-v10.dtsi

&uart2{
    status = "disabled";
};
```

Note: This debug serial port is configured by default.

4.7 SPI

4.7.1. Introduction

There are four on-chip SPIs on the XPI-3566 development board, and the usage of each SPI is shown in the following table：

Port	Pin name	Device
spi0_m0	GPIO0_B5/spi0_clkm0 GPIO0_C5/spi0_misom0 GPIO0_B6/spi0_mosim0 GPIO0_C6/spi0_cs0m0 GPIO0_C4/spi0_cs1m0	Multiplexing to other functions
spi0_m1	GPIO2_D3/spi0_clkm1 GPIO2_D0/spi0_misom1 GPIO2_D1/spi0_mosim1 GPIO2_D2/spi0_cs0m1	Multiplexing to other functions
……	……	Multiplexing to other functions
Spi1_m1	GPIO3_C3/spi1_clkm1 GPIO3_C2/spi1_misom1 GPIO3_C1/spi1_mosim1 GPIO3_A1/spi1_cs0m1	Multiplexing to other functions
……	……	Multiplexing to other functions
spi3_m1	GPIO4_C2/spi3_clkm1 GPIO4_C5/spi3_misom1 GPIO4_C3/spi3_mosim1 GPIO4_C6/spi3_cs0m1 GPIO4_D1/spi3_cs1m1	Multiplexing to other functions

Note: Other SPI pin configurations can be found in the /loong/devices /rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl. dtsi file to view.

This chapter describes how to configure the SPI on this development board.

The following is an example of configuring the SPI test program.

4.7.2 Defining and registering **SPI** devices

In standard Linux, the user simply provides the appropriate SPI device information, and Linux constructs the `spi_client` structure based on the information provided. The spi device information provided by the user is written to the DTS file in the form of nodes as follows.

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dt
s/rockchip/rk3566-evb2-lp4x-v10.dtsi

&spi0{
    status = "okay";
    max-freq = <48000000>; /* spi internal clk, don't modify */
    pinctrl-names = "default";
    pinctrl-0 = <&spi0m1_cs0 &spi0m1_pins>;

    spi_dev@0 {
                compatible = "rockchip,spidev";
                reg = <0>;
                spi-max-frequency = <12000000>;
                spi-lsb-first;
        };
```

After configuring the dts node, you need to configure the defconfig file so that it compiles the spidev.c driver.

```
<SDK>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/configs
/rockchip_linux_defconfig

CONFIG_SPI_SPIDEV=y
```

Room 02-04, 10/F, Block A, Building 8, Shenzhen International Innovation Valley, Dashi Road,
Nanshan District, Shenzhen, Guangdong, China
Emai: support@geniatech.com Tel: (+ 86) 755 86028588

4.7.3 Defining and registering **SPI** drivers

A. Defining the SPI Driver

Before defining the spi driver, the user must first define the variable `of_device_id`. `of_device_id` is used to call the device information defined in the DTS file in the driver, which is defined as follows.

```
<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` as follows:

```
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. Registering the SPI Driver

Use the spi_register_driver function to register the SPI driver.
spi_register_driver(&spidev_spi_driver);
When calling spi_register_driver to register the SPI driver, it will traverse the SPI devices and call the probe function of the driver if it supports the traversed devices.

4.7.4 Testing SPI devices:

Check if the spi node is registered successfully: 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
```

Use the **kernel/tools/spi/spidev_test.c** test program. First check that the device node opened in this test program is the same as the registered device node, and modify it if it is different.

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

Copy to the board and compile: gcc spidev_test.c -o spidev_test

Shorting spi_miso and spi_mosi

Enter the command: ./spidev_test -v to directly compare whether the input and output appear to be the same.

```
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  |
```