XPI-3566-ZERO

Linux System User Manual

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

Audience
This document (this guide) is primarily intended for the following agents:
The technical Support Engineer
The software development engineer
Revision record

Time	Version	Author	Purpose
23/10/9	V1.0	LiuZC	First edition

Address: Room 1002-1004, Block A, Building 8, International Innovation Valley, Xili Street, Nanshan
District Website： www.geniatech.com
Customer Service Tel： +86-0755-86028588
Technical support email： support @geniatech.com
Sales service mailbox： sales@geniatech.com

CATALOG

1. Build compilation environment

This chapter introduces build the compil environment of Linux SDK.

Notice：

(1) It is recommended to develop in the x86 _ 64 Ubuntu 18.04 system environment. If other system versions are used, compilation environment need to be adjusted accordingly.

(2) Compile with a normal user, do not with root privileges.

1.1 GET SDK

First, prepare an empty folder to store the SDK. It is recommended to place it under the home directory, this article takes ~/proj as an example.

Do not store or decompress the SDK in shared folders and no-English directories of the virtual machine to avoid unnecessary errors.

The SDK package can be obtained from the download interface of the company's official website.

```
# Decompress
mkdir  ~/proj
cd  ~/proj

// The SDK name may be different from the one you downloaded. Decompress it accord to you got
tar -xf xpi-566-debain10-***.tar
```

1.2 Install dependent packages

The installation command of the software package on which the SDK environment build depends 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 Cross Compilation Tool Chain

Since the Rockchip Buildroot SDK is only compiled under Linux at present, we only provide the cross-compilation tool chain under Linux. The preset directory of the compilation tool chain used by UBoot and Kernel is under prebuilt/gcc, the buildroot uses the tool chain compiled from the open-source software.

U-Boot and Kernel compilation tool chain:

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

Correspond version

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

Buildroot compilation tool chain
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
Correspond version:

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

1.4 Introduction to SDK Project Directory

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

```
.
├── app
├── buildroot                                          #Buildroot Root file system compilation directory
├── build.sh -> device/rockchip/common/build.sh              # Compile the script
├── debian                                                   # Debian Root file system compilation directory
├── device                                                   # Compile the relevant configuration films
├── docs                                                     # word
├── 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 tool chain
├── rkbin
├── rkflash.sh -> device/rockchip/common/rkflash.sh          # Burn script
├── rkbin                                                   #Store Rockchip-related Binary and tools.
├──rockdev                                                  # Store compiled output firmware
├── tools                                                    # Tool catalog
└── u-boot                                                   # U-Boot
```

app： Store the upper application app, mainly some applications such as qcamera/qfm/qplayer/settings.
buildroot： Root filesystem based on buildroot (2018.02-rc3).
debian：The root file system developed based on Debian 10 supports some chips.
device/rockchip：Store the board-level configuration and Parameter files of each chip, as well as some scripts and preparation files for compiling and packaging firmware.
docs：Store chip module development guidance documents, platform support list, chip platform related documents, Linux development guide, etc.
IMAGE： A directory that holds that compile times, XML, patch and firmware for each build.
external： Storage of third-party related warehouses, including audio, video, network, recovery, etc.
kernel： Store the code developed by the kernel 4.4 or 4.19.
prebuilts： Store the cross-compilation tool chain.
rkbin： Store Rockchip-related Binary and tools.

rockdev： And store that compiled output firmware.
tools： Store common tools in Linux and Windows operating system environment.
u-boot：Store the uboot code developed based on the v2017.09 version.

2. Code compilation

2.1 Select a configuration file before compiling

```
#SDK Root Directory Select Configuration File
./build.sh lunch Select 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 Compile the SDK

There are two ways to compile SDK, one is to compile each part separately, and the other is to compile automatically. The first compilation can use the whole compilation, and the second compilation can compile the modified part separately according to the modification.

2.2.1 Compile the U-Boot

Enter the SDK project. Run the following command to compile

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

2.2.2 Compile the Kernel

Enter the root directory of the project directory, execute the following command to automatically compile and pack the kernel.

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

2.2.3 Compile the Debian

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

2.2.4 Fully automatic compilation

After completing the compilation of each part of Kernel/U-Boot/Recovery/Rootfs, enter the root directory of the project directory and execute the following command to automatically complete all compilations:

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

2.3 Firmware pack

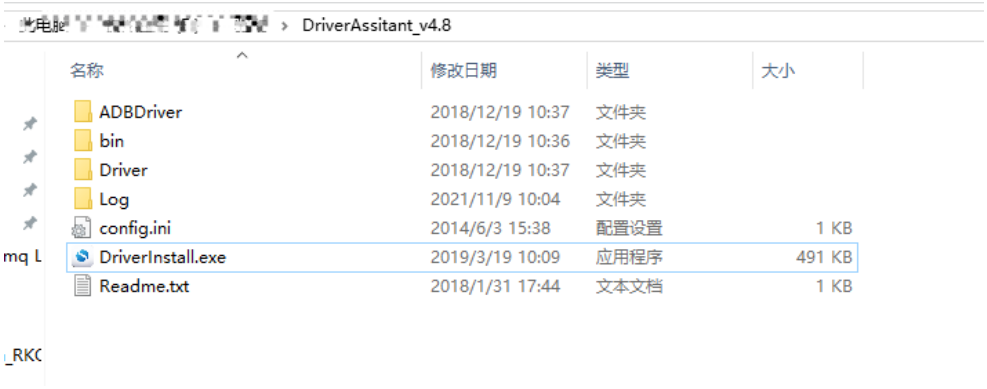
```
# Firmware pack
./build.sh firmware

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

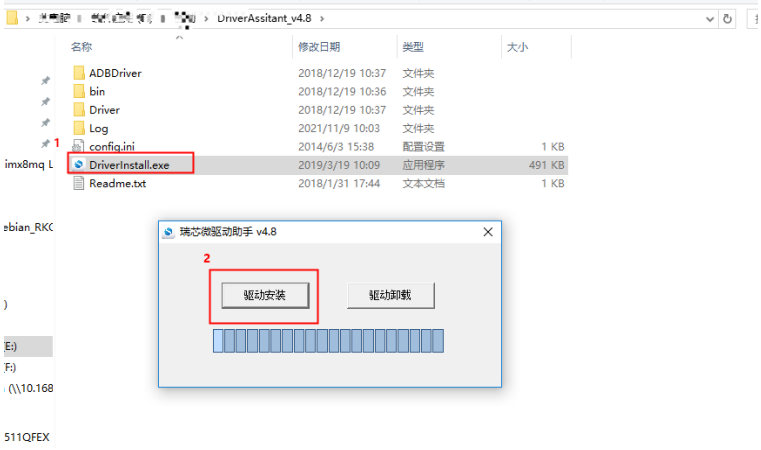
3. Firmware programming

3.1 Install the programming tool driver:

DriverAssitant_v4.8.zip See annex.



Double-click "DriverInstall.exe" application, and click "Install Driver".



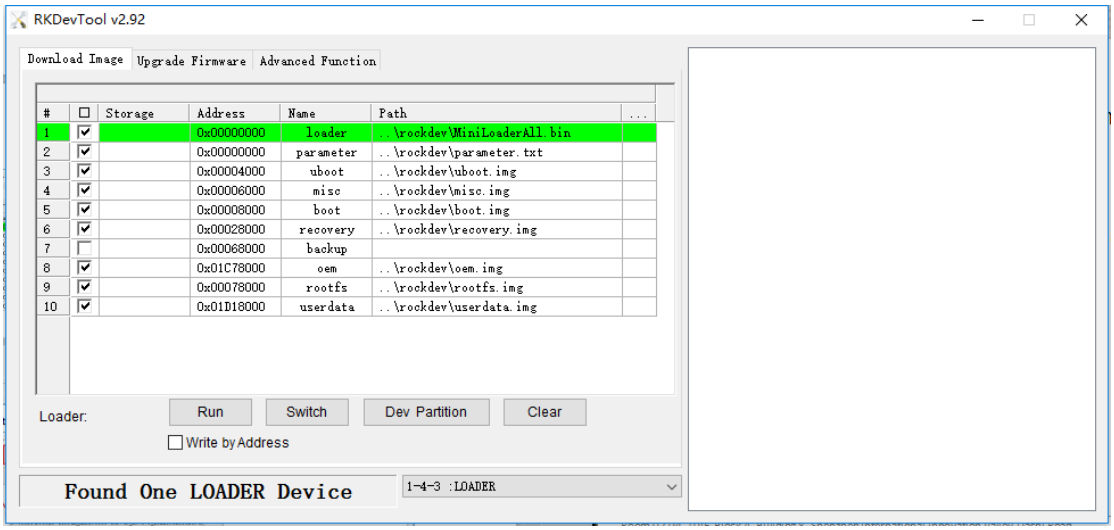
3.2 Windos flash instructions

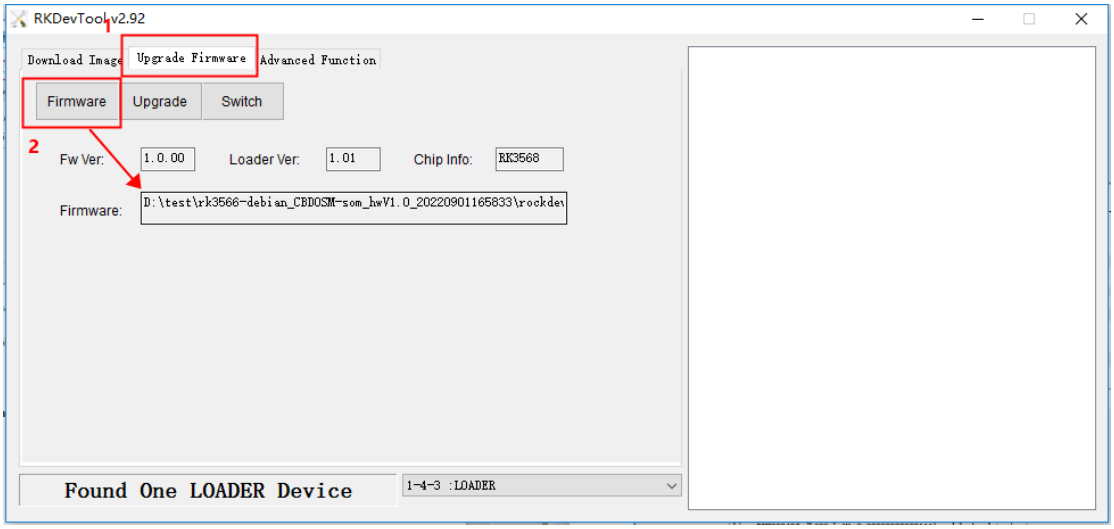
SDK provides Windows programming tool (tool version requires V2.55 or above), and the tool is located in the root directory of the project:

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

As shown in the figure below, after compiling and generating the corresponding firmware, the device programming needs to enter the MASKROM or BootROM programming mode. After connecting the USB download cable, press and hold the key "Upgrade" and plug in the power supply to enter the MASKROM mode. After loading the corresponding

path of the firmware generated by compiling, Click "Execute" to programming.





4. Software development

4.1 Introduction the Software Development Catalog

kernel device tree directory：

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

u-boot device tree directory：

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

defconfig directory:

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

4.2 Introduction to DTS

DTS overview

Earlier versions of the Linux Kernel configured board-related information directly in the board-level configuration file, such as IOMUX, GPIO pulled high/low by default, and client device information under each I2C/SPI bus. In order to get rid of this' hard code ', Linux introduces the concept of Device Tree(Device Tree) to describe different hardware architectures.

Device Tree data is readable, follows the DTS specification, and is usually described in.dtsi and.dts source files. A binary file that is compiled as a.dtb during kernel compilation. During the boot phase, the dtb is loaded into an address space in RAM by a bootloader, such as U-Boot, and the address is passed to the Kernel space as a parameter. The kernel parses the entire dtb file and refines each device information to initialize. The purpose of this article is to introduce how to add a board dts configuration and some common dts syntax descriptions. More detailed dts syntax descriptions are beyond the scope of this article. If you are interested, please refer to: devicetree-specifications and devicetree-bindings

The Linux Kernel currently supports the use of DTS on multiple platforms, and the DTS file for the RK platform is stored in：

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

If the design of the hardware is the structure of the core board and the backplane, or the product has multiple product forms, the common hardware description can be put in the dtsi file, but the dts file describes different hardware modules, and the common hardware description is included through the include "xxx.dtsi".

4.3 Kernel

4.3.1 Kernel customization

First, you need to obtain the SDK. The preparation and acquisition methods of the development environment are shown in Chapter 1 and Chapter 2. After that start adding kernel options:

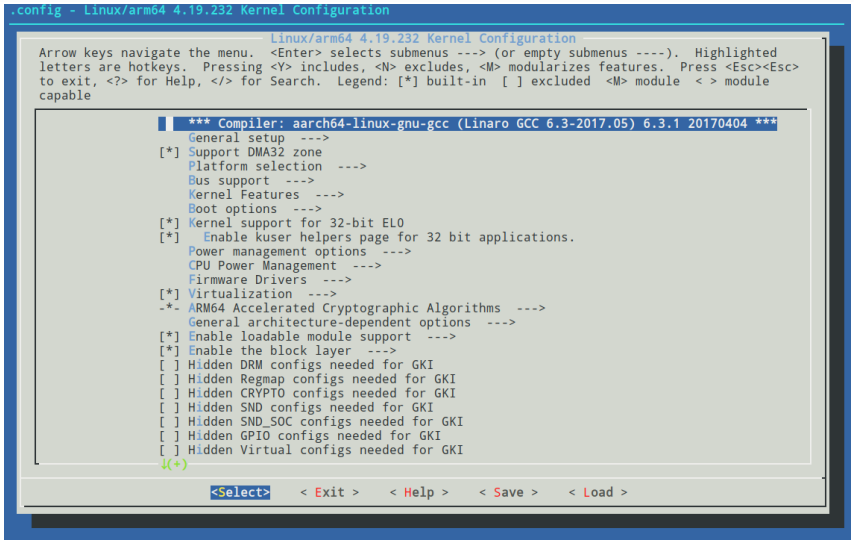
- First go in to the SDK/kernel folder, write the configuration file to .config:

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

- Enter the configuration menu

```
make ARCH=arm64 menuconfig
```

After that, it will enter a graphical interface for configuration.



- Introduction to use

1. An asterisk * in the option box indicates that it is enabled and compiled into the kernel, a blank indicates that it is not enabled, M indicates that it is enabled and compiled into a module.

2. The option box is divided into [] and <>, [] can only select compile (press Y) or remove (press N). <> In addition to selecting compile or remove, you can also select compile as a module (press M);
3. The option is followed by —>, indicating that there is a subdirectory under the change directory;
4. Double-click ESC to exit. Press ? Press the key to display the help information, and press the / Press the key to display the help information, and press the;
5. The four direction keys of up, down, left and right are to move the cursor, Enter is to select;

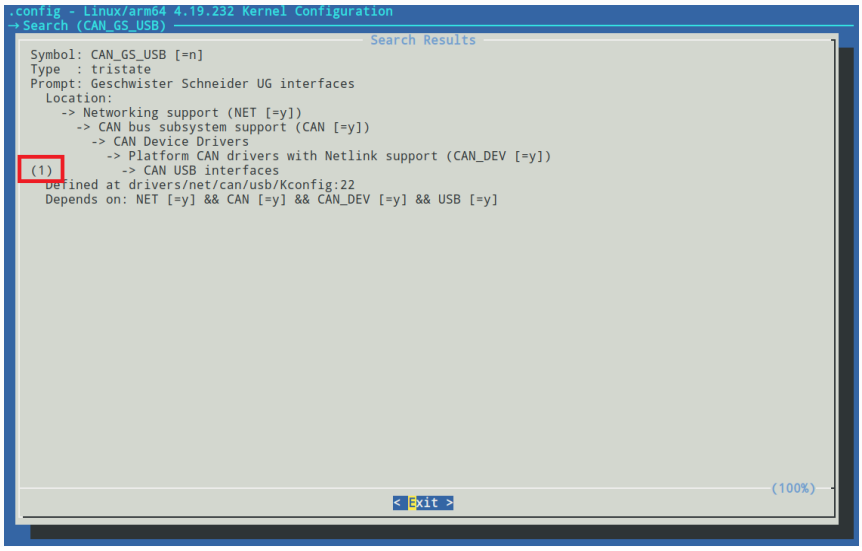
- Precautions

Compile as module (M) requires subsequent installation for normal use, so it is recommended to add a small amount of configuration by compiling into the kernel (Y) to save the installation steps.

- Save

After opening the required configuration, press the direction keys left and right to move the cursor to Save and press Enter for 3 times to save, and then continue to double-click Esc to exit. After exiting, save the modified content to the configuration file:

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



4.3.2 Kernel compilation

Go back to the SDK root directory to compile:

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

Generated file location: SDK/kernel/boot.img

4.4 GPIO

4.4.1 Introduction

GPIO, Full name is General-Purpose Input/Output, is a pin that can be dynamically configured and controlled during software operation. The initial state of all the GPIOs after power-on is an input mode, which can be set as a pull-up pin or a pull-down pin through software, and can also be set as an interrupt pin. The driving strength is programmable, and the core of the method is to fill the method and parameters of the GPIO bank, and to call the gpiochip_add to register in the kernel.

The XPI-3566-ZERO has 14pin GPIO ports: the usage of each GPIO is shown in the following table:

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	Reuse for other functions
GPIO1	GPIO1_A7_d/SPI0_CLK_M0/ UART4_CTSn_M0	Reuse for other functions
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	Reuse for other functions
GPIO4	GPIO4_C1_d/UART1_CTSn_M1 GPIO4_B6_d/ UART1_RTSn_M1 GPIO4_C4_d/ GPIO4_C4 GPIO4_A7_d/SPI3_CS1_M0	Reuse for other functions

The following formula is commonly used to calculate the pin:

```
GPIO pin calculation formula: pin = bank * 32 + number
GPIO Group Number Calculation Formula: number = group * 8 + X
```

The GPIO4_A7 pin calculation method is demonstrated below:
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;

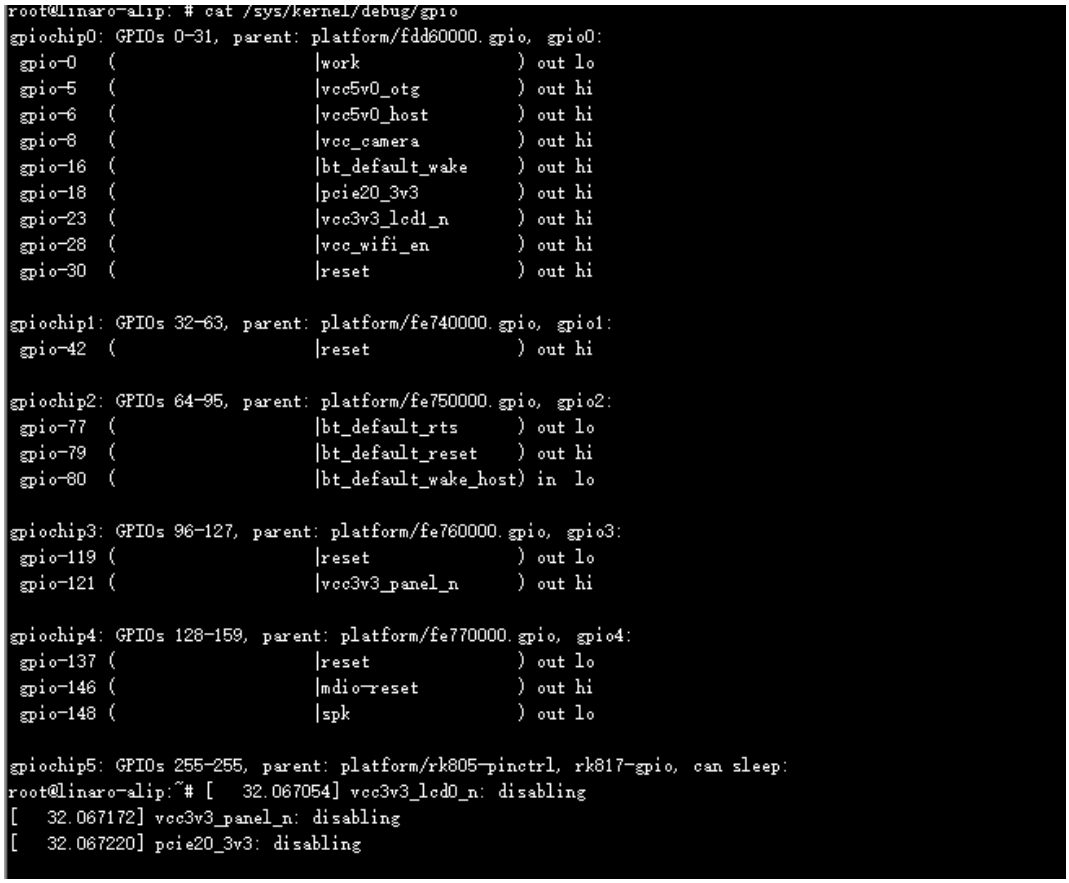
he attribute description of the device tree corresponding to GPIO4_A7 is:<&gpio4 7 IRQ_TYPE_EDGE_RISING> , which can be seen from the macro definition of kernel/include/dt-bindings/pinctrl/rockchip.h ,

GPIO4_A7 can also be described as<&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
......
```

4.4.2 User Space Control GPIO

GPIO4_A7 may be occupied by other functions, and we can use `cat /sys/kernel/debug/gpio` command to see which GPIOs are already occupied. When the pin is not reused by other peripherals, we can export the pin to use it.



The steps to configure the GPIO as a normal I/O port are as follows:
Step 1. Use the echo command on the console to export the GPIO number to be operated: `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 input: `echo out > /sys/class/gpio/gpioN/direction`
3、GPIO orientation can be viewed using the cat command:
`cat /sys/class/gpio/gpioN/direction`

Step 3. Use the cat or echo command on the console to view the GPIO input value or set the GPIO output value:
1. View the input values: `cat /sys/class/gpio/gpioN/value`
2. Output is low: `echo 0 > /sys/class/gpio/gpioN/value`
3. Output high: `echo 1 > /sys/class/gpio/gpioN/value`

4.4.3 Drive control GPIO

This paper takes two GPIO ports, GPIO4_A0 and GPIO_A7, as an example to write a simple GPIO port driver. The path of SDK is: kernel/drivers/gpio/gpio-control.c. The following takes this driver as an example to introduce the operation of GPIO.

A. GPIO as normal input and output

First, add the resource description of the driver in the DTS file:

```
<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>;
    };
```

Then configure the deconfig file to compile to the gpio-control.c file:

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

CONFIG_GPIO_CONTROL=y
```

Here a pin is defined as a general output input port: gtc-gpios GPIO4_A7 GPIO_ACTIVE_HIGH Indicates active high. If you want active low, change to: GPIO_ACTIVE_LOW. This property will be read by the driver.

Then, the resource added by DTS is parsed in the probe function, and the code is as follows:

```
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, the original
gpio _ IRC _ probe function content can be deleted.
```

of_get_named_gpio_flags read the GPIO configuration number and flag of the gtc-gpios from the device tree, the gpio_is_valid judges whether the GPIO number is valid, and the gpio_request applies to occupy the GPIO. If the initialization process fails, the gpio_free needs to be called to release the previously requested and successful GPIO. You can set whether to output high or low level by calling the gpio_direction_output in the drive. The default output here is the effective level GPIO_ACTIVE_HIGH obtained from DTS, that is, high level. If the drive works normally, you can use a multimeter to measure that the corresponding pin should be high level.

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 definition:

```
#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 is used as an interrupt pin

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

```
<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 indicates that the interrupt is triggered by a rising edge, and the interrupt function can be triggered when a rising edge signal is received on this pin. It can also be configured as follows:

```
IRQ_TYPE_NONE         // Default, no interrupt trigger type defined
IRQ_TYPE_EDGE_RISING  // Rising edge trigger
IRQ_TYPE_EDGE_FALLING //Falling edge trigger
IRQ_TYPE_EDGE_BOTH    // Both rising and falling edge trigger
IRQ_TYPE_LEVEL_HIGH   // High level trigger
IRQ_TYPE_LEVEL_LOW    // Low level trigger
```

Then, parse the resource added by DTS in the probe function, and then make an interrupt registration application. The code is as follows:

```
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) // Interrupt function
{
    printk("Enter gtc gpio irq test program!\n");
    return IRQ_HANDLED;
}
```

The gpio_to_irq is called to convert the PIN value of the GPIO into the corresponding IRQ value, the gpio_request is called to apply for occupying the IO port, the request_irq is called to apply for interruption, and the free_irq is called to release if the application fails. In this function, gtc_irq is the hardware interrupt number to be applied, gtc_gpio_irq is the interrupt function, flag is the attribute of interrupt handling, and gtc-gpio is the device driver name.

4.5 I2C

4.5.1 Introduction

Four PINs on the XPI-3566-ZERO development board are I2C controllers, and the usage of each I2C is shown in the following table:

Port	Pin name	Device
I2C1	GPIO0_B4_u / I2C1_SDA GPIO0_B3_u / I2C1_SCL	Reuse for other functions
I2C3	GPIO1_A0_u/I2C3_SDA_M0 GPIO1_A1_u/ I2C3_SCL_M0	Reuse for other functions

This chapter describes how to configure I2C on this board.

Configuring I2C can be divided into two major steps:

Define and register I2C Equipment

Define and register I2C driver

Take the configuration of at24c02 as an example. The driver is located at kernel/drivers/misc/eeprom/at24.c .

4.5.2 Define and register I2C device

When registering an I2C device, the structure i2c_client is required to describe the I2C device. However, in standard Linux, the user only needs to provide the corresponding I2C device information, and Linux will construct the i2c_client structure according to the provided information.
The user-supplied I2C device information is written to the DTS file in the form of a node, as shown below：

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

After configuring the dts node, you need to configure the defconfig file to compile the rtc driver:

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

CONFIG_EEPROM_AT24=y
```

4.5.3 Define and register I2C driver

A. Define the I2C driver

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

```
<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);
```

Define variable 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 as shown below:

```
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,
};
```

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

B. Register the I2C driver

Register the I2C driver using the i2c_register_driver function.

```
module_i2c_driver(at24_driver);
#module_i2c_driver Is a macro definition that can be expanded to:
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);
```

When i2c_register_driver is called to register the I2C driver, the I2C device will be traversed. If the driver supports the traversed device, the probe function of the driver will be called.

4.5.4 Sending and receiving data via I2C

After the I2C driver is registered, I2C communication can be carried out. 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;
 }
```

Read information from 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;
 }
```

4.5.5 Test the I2C device

You can use i2ctool to test whether the I2C device is registered successfully. The steps are as follows:

a. Enter the command to update the software library: 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 whether there is the address of the registered device under the I2C used: 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: -- -- -- -- -- -- -- --
```

You can see that a device with an address of 0x50 is registered under i2c0.

4.6 UART

4.6.1 Introduction

There are 6 pins on the XPI-3566-ZERO development board that are UART controllers, and the usage of each UART is shown in the following table:

Port	Pin name	Device
Uart4	SPI0_MOSI_M0/UART4_TX_M0 SPI0_MISO_M0/UART4_RX_M0	Reuse for other functions
Uart7	GPIO4_A3_d / UART7_RX_M2 GPIO4_A2_d / UART7_TX_M2	Debugging window serial port
Uart9	GPIO4_C5_d/ UART9_TX_M1-PWM12_M1 GPIO4_C6_d/ UART9_RX_M1-PWM13_M1	Reuse for other functions

Note: Other UART pin configurations can be found in th<sdk> /kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl.dtsi files.

This chapter describes how to configure the UART on this board.

Take the configuration of uart4 and board debug port (debug serial port) as an example.

4.6.2 Configure the UART4 interface

a. Determine the pin used by uart4, if uart4 uses the pin: GPIO1_A4、GPIO1_A6、GPIO1_A7、GPIO1_A5

b. Find the top pin number in the rk3568-pinctrl.dtsi films:

```
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. Configure the dts file

```
<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 Configure the debugging serial port

a. Find the debug used by the board and the pins used.
b. Add in dts file rk3568-linux.dtsi:

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

fiq-debugger {
        compatible = "rockchip,fiq-debugger";
        rockchip,serial-id = <7>;    // The string number used
        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>;  // Pins used for serial port
        status = "okay";
    };

Note: uart7m2_xfer is defined in the rk3568-pinctrl.dtsi file.
```

c. Turn off the uart7 function (turn off the uart7 when debugging the serial port):

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

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

Note: This debugging serial port has been configured by default.

4.7 SPI

4.7.1. Introduction

Four PINs on the XPI-3566-ZERO board are SPI3, and the usage of each SPI is shown in the following table:

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	Reuse for other functions

Note: Other SPI pin configurations can be found in th<sdk>/loong/devices/rk3566-xpi-debian10/kernel/arch/arm64/boot/dts/rockchip/rk3568-pinctrl.dtsi films.

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

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

4.7.2 Define and register SPI device

In standard Linux, users only need to provide the corresponding SPI device information, and Linux will construct the spi_client structure according to the provided information.
The SPI device information provided by the user is written to the DTS file in the form of a node, as shown below:

```
<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";
    };
};
```

After the DTS node is configured, you need to configure the defconfig file to compile the spidev.c driver

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

CONFIG_SPI_SPIDEV=y
```

4.7.3 Define and register SPI driver

A. Define the SPI driver

Before defining the SPI driver, the user first defines the variable of_device_id.
The of_device_id is used to call the device information defined in the DTS file in the driver, and its definition is 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 shown below:

```
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. Register the SPI driver

Use the spi_register_driver function to register the SPI driver.
spi_register_driver(&spidev_spi_driver);
When the spi_register_driver is called to register the SPI driver, the SPI device will be traversed. If the driver supports the traversed device, the probe function of the driver will be called.

4.7.4 Test the SPI device:

Check whether the spi node is registered successfully: ls s/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 to test the program. First check whether the device node opened in the test program is the same as the registered device node. If not, modify it.

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

Copy to card and compile: gcc spidev_test.c -o spidev_test

Short the spi_miso and spi_mosi

The input command: ./spidev_test-v directly compares the occurrence of the input and output for consistency.

```
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  |
```