StarFive 赛昉科技

# 昉·惊鸿7110 GPIO开发和移植手册

昉·星光 2

版本：1.0

日期：2022/11/10

Doc ID：JH7110-DGCH-002

# 法律声明

阅读本文件前的重要法律告知。

## 版权注释



本文档中的说明均基于“视为正确”提供，可能包含部分错误。内容可能因产品开发而定期更新或修订。上海赛昉科技有限公司 （以下简称“赛昉科技”）保留对本协议中的任何内容进行更改的权利，恕不另行通知。

赛昉科技明确否认任何形式的担保、解释和条件，无论是明示的还是默示的，包括但不限于适销性、特定用途适用性和非侵权的担保或条件。

赛昉科技无需承担因应用或使用任何产品或电路而产生的任何责任，并明确表示无需承担任何及所有连带责任，包括但不限于间接、偶然、特殊、惩戒性或由此造成的损害。

本文件中的所有材料受版权保护，为赛昉科技所有。不得以任何方式修改、编辑或断章取义本文件中的说明，本文件或其任何部分仅限用于内部使用或教育培训。

## 联系我们：

地址： 浦东新区盛夏路61弄张润大厦2号楼502， 上海市，201203，中国

网站： http://www.starfivetech.com

邮箱：

- sales@starfivetech.com（销售）
- support@starfivetech.com（支持）

# 前言

关于本指南和技术支持信息

## 关于本手册

本手册主要为SDK开发和移植提供赛昉科技新一代SoC平台——昉·惊鸿7110的GPIO编程基础和调试操作。

## 受众

本手册主要服务于与GPIO驱动程序相关的开发人员。如果您正在开发其他模块，请与您的销售或支持顾问联系，获取昉·惊鸿7110的完整文档。

## 修订历史

**表 0-1 修订历史**

| Version | 发布说明 | 修订 |
|---|---|---|
| 1.0 | 2022/11/10 | 首次发布。 |

## 注释和注意事项

本指南中可能会出现以下注释和注意事项：

- **提示：**
  建议如何在某个主题或步骤中应用信息。
- **注：**
  解释某个特例或阐释一个重要的点。
- **重要：**
  指出与某个主题或步骤有关的重要信息。
- **警告：**
  表明某个操作或步骤可能会导致数据丢失、安全问题或性能问题。
- **警告：**
  表明某个操作或步骤可能导致物理伤害或硬件损坏。

# 目录

# 表格清单

# 插图清单

# 1. 简介

GPIO主要基于pinctrl（pin控制）框架。

Pinctrl框架是Linux系统框架，旨在统一不同SoC平台上的pin管理。该框架包括SoC提供商，也包括赛昉科技，他们在创建pin管理子系统方面做了大量的工作。

## 1.1. 功能介绍

赛昉科技昉·惊鸿7110 SoC平台提供pin控制器，支持开发者配置一个或一组pin功能。Linux内核中的pinctrl驱动程序可以帮助开发人员完成以下任务。

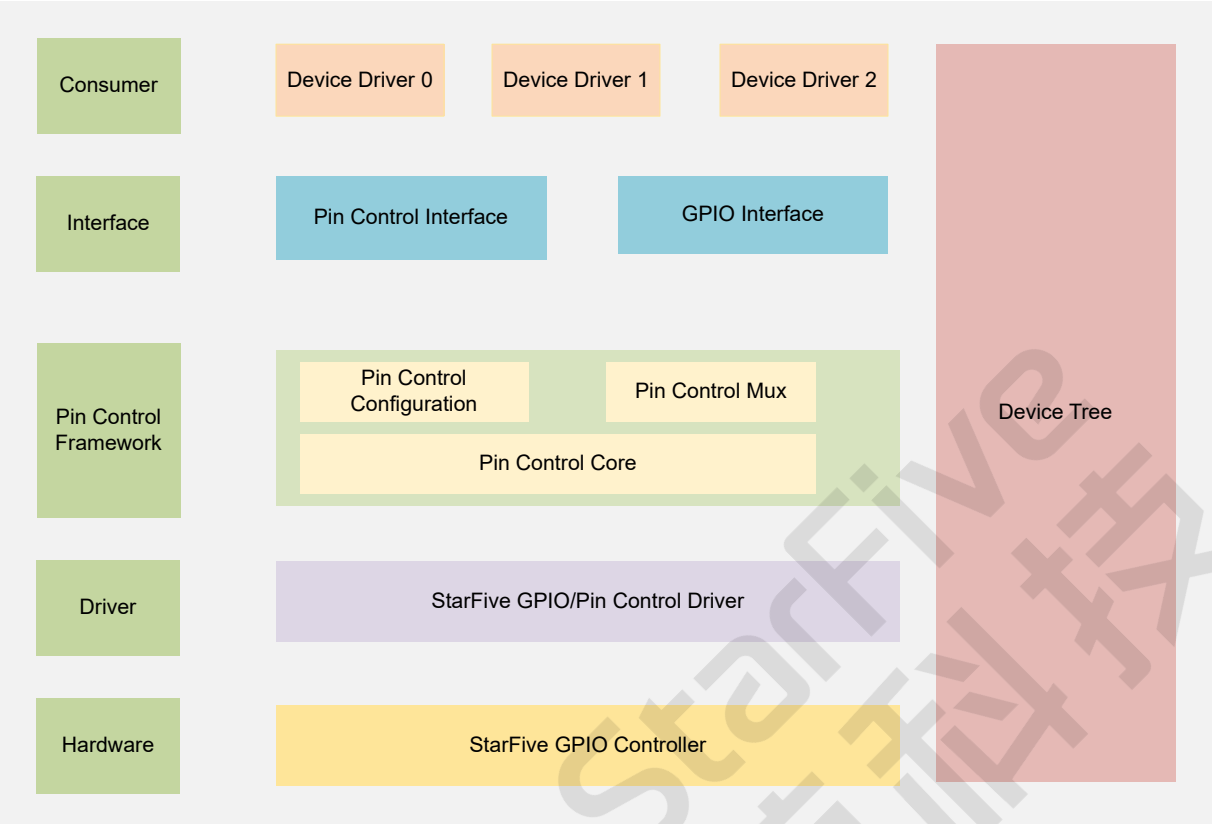
- 定位并命名pin控制器可以管理的所有pin。
- 提供pin多路复用。
- 提供pin配置选项，例如，驱动能力、通电、断电和数据属性等。
- 与GPIO子系统交互。
- 实现pin中断。

## 1.2. 框图

下图显示了昉·惊鸿7100 GPIO pin控制（pinctrl）驱动程序模块的框图。驱动程序模块有以下四个部分：

- Pin控制接口
- Pin控制通用框架
- 赛昉科技昉·惊鸿7110 pin控制驱动程序
- 板级配置

**图 1-1 框图**




上图包含以下层:

- **Consumer（用户）**： 设备驱动程序所使用的pin控制接口和GPIO接口， 如SDIO、PCIE等。
- **Interface（接口）**： 用户的pin控制和GPIO接口， 查看接口介绍 (第 19页)获取更多信息。
- **Pinctrl framework（Pinctrl框架）**： 原始的Linux系统pin控制框架。该框架使开发人员可以配置一个或一组pin功能。
- **Driver（驱动）**： 赛昉科技昉·惊鸿7110 SoC平台的GPIO和pin控制驱动程序。
- **Hardware（硬件）**： 赛昉科技昉·惊鸿7110 SoC平台的GPIO控制器。

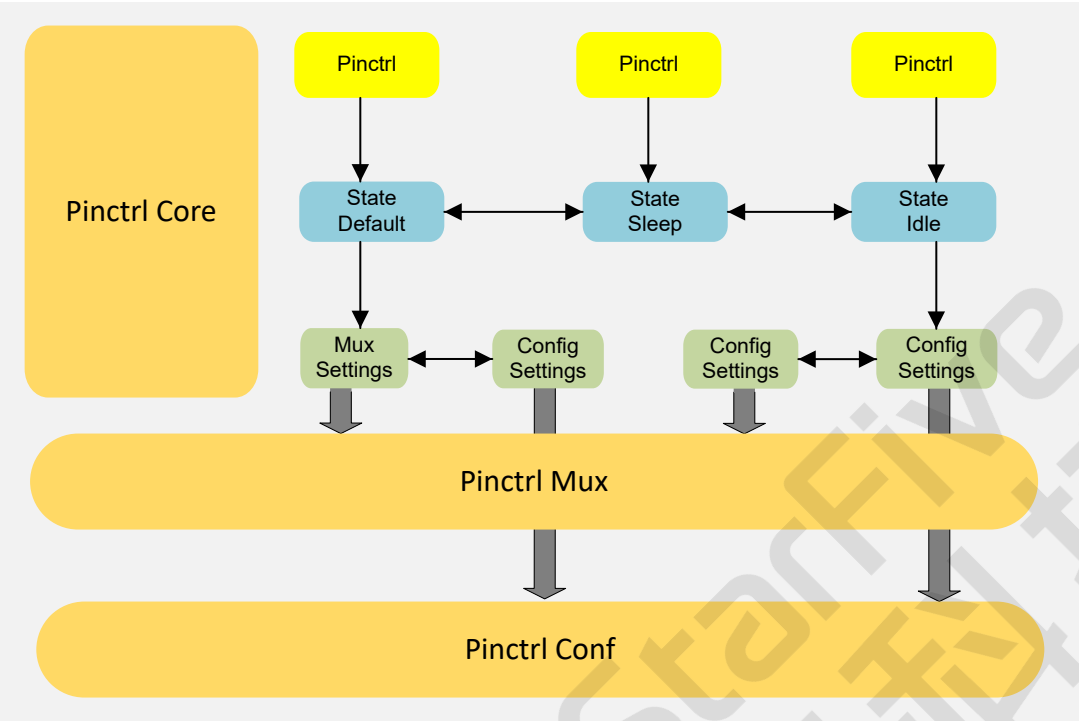
## 1.3. Pin控制框架

昉·惊鸿7110 SoC平台的Pin控制（Pinctrl）框架主要由以下三个组件组成。

- **Pinctrl Core**: pin控制框架的核心层。State Default、State Sleep或State Idle指的是pin控制的电源管理状态。
- **Pinctrol Mux**: pin多路复用功能。
- **Pinctrl Conf**: pin配置设置。

下图显示了这三者的关系。

**图 1-2 Pin控制框架**




对于特定的系统工作模式，pin的配置可能会有所不同。例如，正常模式下，可采用默认配置，待机模式下，则应采用省电的配置模式。因此，您可以使用上述的pin控制框架，按照设备的工作模式来管理pin配置。

## 1.4. 源代码结构

以下代码块为GPIO的源代码结构：

```
linux
├── drivers
│   ├── pinctrl
│   │   ├── core.c
│   │   ├── core.h
│   │   ├── devicetree.c
│   │   ├── devicetree.h
│   │   ├── pinconf.c
│   │   ├── pinconf-generic.c
│   │   ├── pinconf.h
│   │   ├── pinmux.c
│   │   ├── pinmux.h
│   │   ├── starfive
│   │   │   ├── Kconfig
│   │   │   ├── Makefile
│   │   │   ├── pinctrl-starfive.c
│   │   │   ├── pinctrl-starfive.h
```

```
|   |   |   └── pinctrl-starfive-jh7110.c

├── include
|   ├── dt-bindings
|   |   ├── pinctrl
|   |   |   ├── starfive,jh7110-pinfunc.h
|   ├── linux
|   |   ├── pinctrl
|   |   |   ├── consumer.h
|   |   |   ├── devinfo.h
|   |   |   ├── machine.h
|   |   |   ├── pinconf-generic.h
|   |   |   ├── pinconf.h
|   |   |   ├── pinctrl.h
|   |   |   ├── pinctrl-state.h
|   |   |   └── pinmux.h
```

## 1.5. 设备树概述

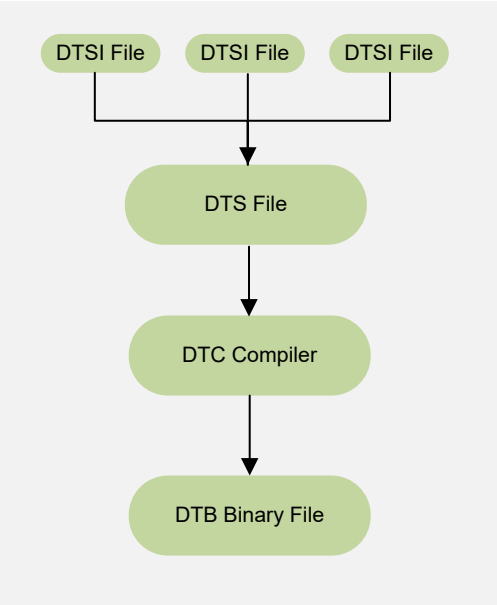
自Linux 3.x以来，系统就引入了设备树作为数据结构和语言来描述硬件配置。设备树是硬件设置的系统可读描述，这样操作系统不必硬编码机器的详细信息。

一个设备树主要有以下呈现形式。

- 设备树编译器（DTC）：用于将设备树编译为系统可读的二进制文件的工具。
- 设备树源码（DTS）：人类可读的设备树描述文件。您可以在此文件中找到目标参数并修改硬件配置。
- 设备树源码信息（DTSI）：可包括在设备树描述中的人类可读的头文件。您可以在此文件中找到目标参数并修改硬件配置。
- 设备树块（DTB）：系统可读设备树二进制blob文件，在系统中烧录以供执行。

下图显示了上述形式的关系（工作流）。

**图 1-3 设备树工作流**




# 1.6. 设备树代码

## 总体结构

昉·惊鸿7110的设备树代码如下：

```
linux
├── arch
│   ├── riscv
│   │   ├── boot
│   │   │   ├── dts
│   │   │   │   └── starfive
│   │   │   │       ├── codecs
│   │   │   │       │   ├── sf_pdm.dtsi
│   │   │   │       │   ├── sf_pwmdac.dtsi
│   │   │   │       │   ├── sf_spdif.dtsi
│   │   │   │       │   ├── sf_tdm.dtsi
│   │   │   │       │   └── sf_wm8960.dtsi
│   │   │   │       ├── evb-overlay
│   │   │   │       │   ├── jh7110-evb-overlay-can.dts
│   │   │   │       │   ├── jh7110-evb-overlay-rgb2hdmi.dts
│   │   │   │       │   ├── jh7110-evb-overlay-sdio.dts
│   │   │   │       │   ├── jh7110-evb-overlay-spi.dts
│   │   │   │       │   ├── jh7110-evb-overlay-uart4-emmc.dts
│   │   │   │       │   ├── jh7110-evb-overlay-uart5-pwm.dts
│   │   │   │       │   └── Makefile
│   │   │   │       ├── jh7110-clk.dtsi
│   │   │   │       ├── jh7110-common.dtsi
│   │   │   │       ├── jh7110.dtsi
```

```
|   |   |   |       ├── jh7110-evb-can-pdm-pwmdac.dts
|   |   |   |       ├── jh7110-evb.dts
|   |   |   |       ├── jh7110-evb.dtsi
|   |   |   |       ├── jh7110-evb-dvp-rgb2hdmi.dts
|   |   |   |       ├── jh7110-evb-pcie-i2s-sd.dts
|   |   |   |       ├── jh7110-evb-pinctrl.dtsi
|   |   |   |       ├── jh7110-evb-spi-uart2.dts
|   |   |   |       ├── jh7110-evb-uart1-rgb2hdmi.dts
|   |   |   |       ├── jh7110-evb-uart4-emmc-spdif.dts
|   |   |   |       ├── jh7110-evb-uart5-pwm-i2c-tdm.dts
|   |   |   |       ├── jh7110-fpga.dts
|   |   |   |       ├── jh7110-visionfive-v2.dts
|   |   |   |       ├── Makefile
|   |   |   |       └── vf2-overlay
|   |   |   |           ├── Makefile
|   |   |   |           └── vf2-overlay-uart3-i2c.dts
```

## SoC平台

昉·惊鸿7110 SoC平台的设备树源代码在以下路径:

```
freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110.dtsi
```

## 昉·星光 2

昉·星光 2 单板计算机（SBC）的设备树源代码在以下路径:

```
freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110-visionfive-v2.dts
-- freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110-common.dtsi
-- freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110.dtsi
```

# 2. 配置

## 2.1. 内核菜单配置

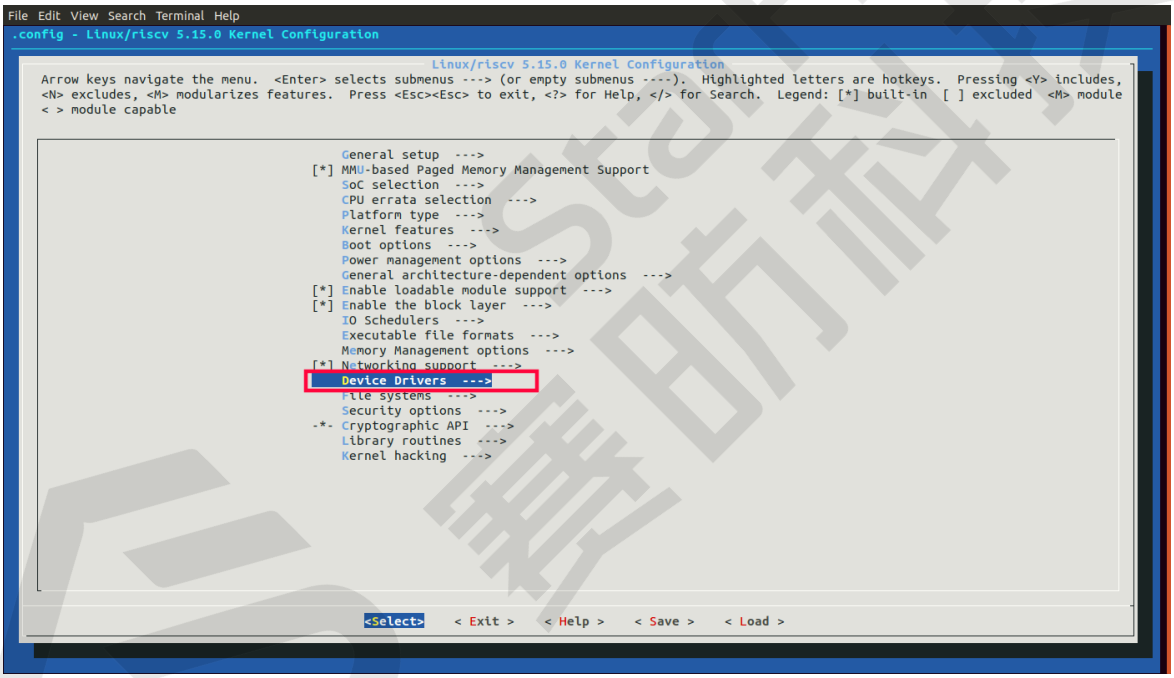
执行以下步骤，创建GPIO的内核配置：

1. 在`freelight-u-sdk`的根目录下，输入以下命令以进入内核菜单配置GUI。

```
make linux-menuconfig
```

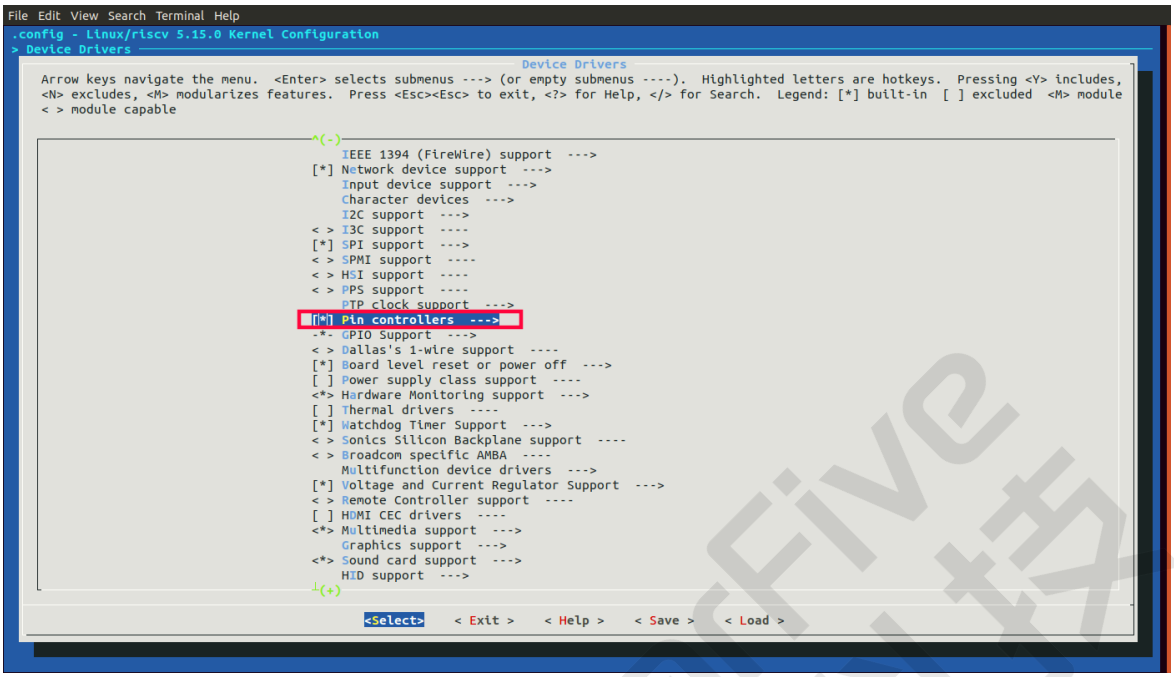
2. 进入**Device Drivers**菜单。

**图 2-1 Device Drivers**



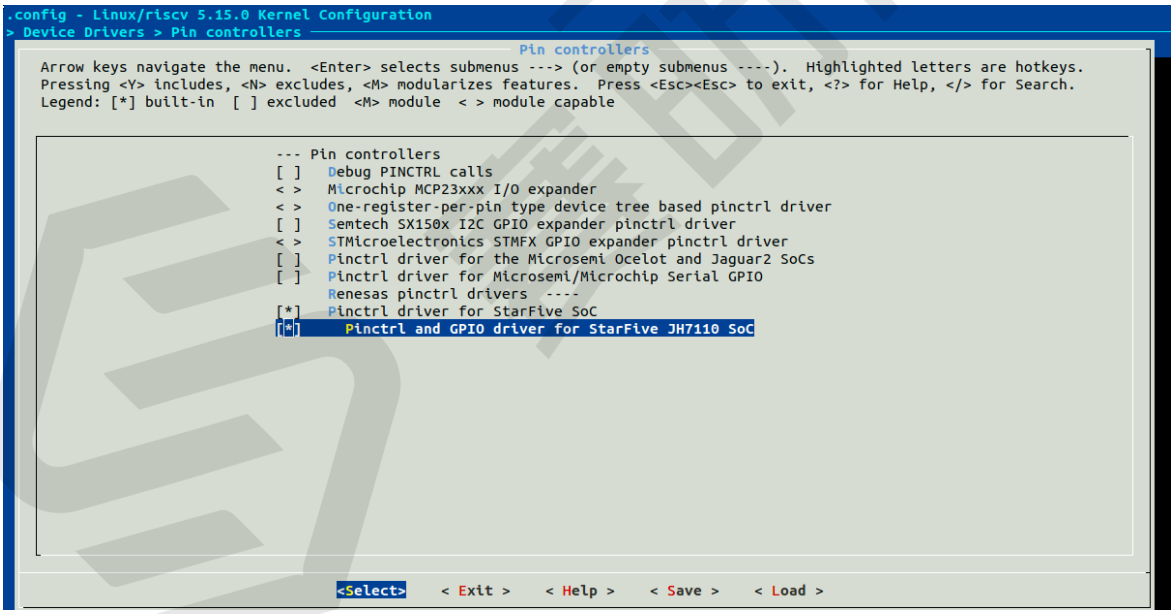

3. 进入**Pin controllers**菜单。

图 2-2 Pin Controllers




4. 选择**Pinctrl driver for StarFive SoC** 选项以启动pinctrl驱动程序，然后选择**Pinctrl and GPIO driver for StarFive JH7110 SoC** 选项以分别启动GPIO驱动程序。

图 2-3 Driver Menu

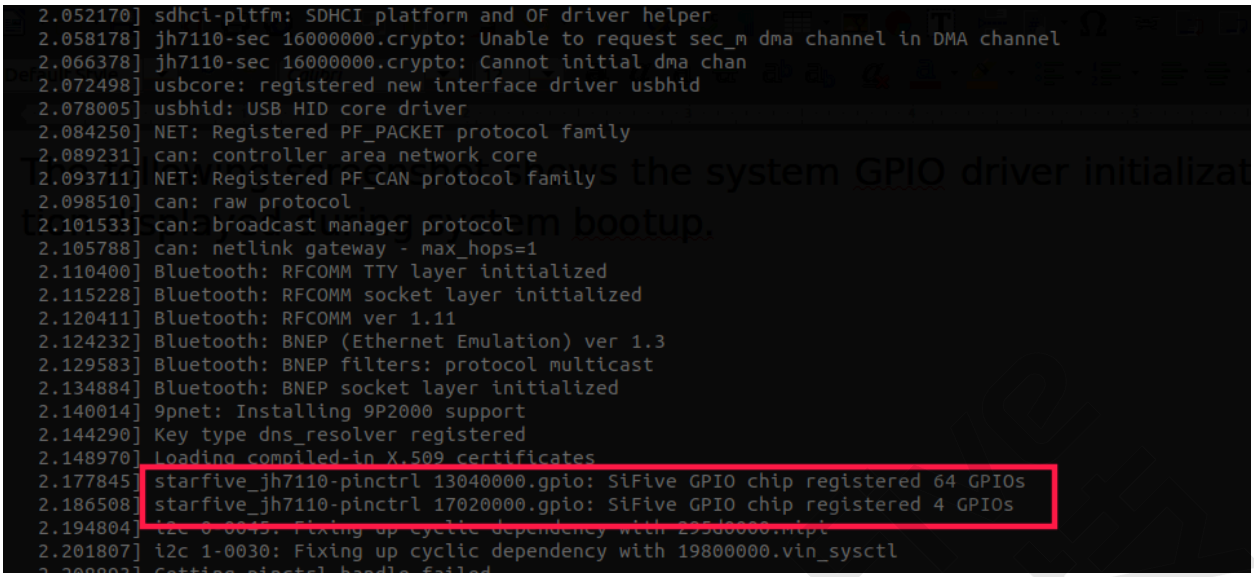



5. 保存更改，并退出内核配置对话框。

## 2.2. 驱动程序初始化

在系统启动期间，当您看到以下红框中的信息，则表示GPIO驱动程序已完成初始化。

**图 2-4 GPIO驱动程序初始化**




## 2.3. 驱动程序验证

系统启动后，您可以运行以下命令验证GPIO驱动程序是否正常工作。

```
# cd /sys/class/gpio/
# ls
export gpiochip0 gpiochip64 unexport
# echo 44 > export
# ls
export gpio44 gpiochip0 gpiochip64 unexport
# cd gpio44/
# ls
active_low direction subsystem value
device edge uevent
# cat direction
in
# cat value
1
```

## 2.4. 设备树配置

昉·惊鸿7110 SoC平台的一般配置文件在以下路径：

```
linux/arch/riscv/boot/dts/starfive/jh7110.dtsi
```

**重要：**
请不要对上述文件进行任何修改！

以下代码块显示了配置文件的内容。

```
        gpio: gpio@13040000 {
                compatible = "starfive,jh7110-sys-pinctrl";
                reg = <0x0 0x13040000 0x0 0x10000>;
                reg-names = "control";
                clocks = <&clkgen JH7110_SYS_IOMUX_PCLK>;
                resets = <&rstgen RSTN_U0_SYS_IOMUX_PRESETN>;
                interrupts = <86>;
                interrupt-controller;
                #gpio-cells = <2>;
                ngpios = <64>;
                status = "okay";
        };

        gpioa: gpio@17020000 {
                compatible = "starfive,jh7110-aon-pinctrl";
                reg = <0x0 0x17020000 0x0 0x10000>;
                reg-names = "control";
                resets = <&rstgen RSTN_U0_AON_IOMUX_PRESETN>;
                interrupts = <85>;
                interrupt-controller;
                #gpio-cells = <2>;
                ngpios = <4>;
                status = "okay";
        };
```

以下列表提供了对上述代码块中的参数说明。

- **compatible**：兼容性信息，用于连接驱动程序和目标设备。
- **reg**：寄存器基本地址“`0x13040000`”和范围“`0x10000`”。
- **reg-names**：GPIO模块使用到的寄存器名称。
- **clocks**：GPIO模块使用到的时钟。
- **clock-names**：上述时钟的名称。
- **resets**：GPIO模块使用到的复位信号。
- **interrupts**：硬件中断ID。
- **status**：GPIO模块的工作状态。要启用模块，请将此位设置为“`okay`”；要禁用该模块，请将此位设置为“`disabled`”。

请确保您没有对**gpio-cells**和**ngpios**的位进行更改。

## 2.5. 板级配置

`board.dts`文件用于存储板级配置文件。

对于昉·星光 2单板计算机，board.dts文件位于以下路径：

```
linux/arch/riscv/boot/dts/starfive/jh7110-visionfive-v2.dts
```

以URAT0为例，其board.dts文件位于以下路径：

```
linux/arch/riscv/boot/dts/starfive/jh7110-visionfive-v2.dts
```

在文件中，您可以找到关于UART pin控制配置的以下配置信息：

```
&gpio {
        uart0_pins: uart0-pins {
                uart0-pins-tx {
                        sf,pins = <PAD_GPIO5>;
                        sf,pin-ioconfig = <IO(GPIO_IE(1) | GPIO_DS(3))>;
                        sf,pin-gpio-dout = <GPO_UART0_SOUT>;
                        sf,pin-gpio-doen = <OEN_LOW>;
                };

                uart0-pins-rx {
                        sf,pins = <PAD_GPIO6>;
                        sf,pinmux = <PAD_GPIO6_FUNC_SEL 0>;
                        sf,pin-ioconfig = <IO(GPIO_IE(1) | GPIO_PU(1))>;
                        sf,pin-gpio-doen = <OEN_HIGH>;
                        sf,pin-gpio-din =  <GPI_UART0_SIN>;
                };
        };
};
```

您也可以找到关于pin控制的配置信息。

```
&uart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins>;
        status = "okay";
};
```

# 3. 接口介绍

## 3.1. Pin控制接口

昉·惊鸿7110的GPIO模块有以下pin控制接口。

### 3.1.1. pinctrl_get

该接口有以下参数。

- **简介**:

```
struct pinctrl *pinctrl_get(struct device *dev)
```

- **描述**: 该接口用于从设备中加载pin操作句柄。所有操作都是基于这个pin控制句柄。
- **参数**:
  - **dev**: Pin操作适用的设备。
- **返回值**:
  - **成功**: Pinctrl句柄。
  - **失败**: 错误代码。

### 3.1.2. pinctrl_put

该接口有以下参数。

- **简介**:

```
void pinctrl_put(struct pinctrl *p)
```

- **描述**: 该接口用于减少对先前声称的pin控制手柄的使用。必须成对使用使用`pinctrl_get()`函数。
- **参数**:
  - **p**: 释放的pinctrl句柄。
- **返回值**: 无

**注:**
只有pinctrl_get (第 19页)接口调用后，该接口才能被调用。否则，调用无效。

### 3.1.3. devm_pinctrl_get

该接口有以下参数。

- **简介**:

```
struct pinctrl *devm_pinctrl_get(struct device *dev)
```

- **描述**: 该接口相当于`pinctrl_get()`（pinctrl_get (第 19页)）接口，但有资源管理功能。
- **参数**:
  - **dev**: Pin操作适用的设备。
- **返回值**:
  - **成功**: Pin控制句柄。
  - **失败**: 错误代码。

### 3.1.4. devm_pinctrl_put

该接口有以下参数。

- **简介**:

```
void devm_pinctrl_put(struct pinctrl *p)
```

- **描述**: 该接口相当于`pinctrl_put()`函数，但具有资源管理功能。您可以重新分配通过`struct pinctrl`获取的`devm_pinctrl_get()`。

  **注：**
  通常情况下，您不需要该函数。因为它可能会释放所有的资源。

- **参数**:
  - **p**: 要释放的pin控制句柄。
- **返回值**: 无

**注：**
只有先调用devm_pinctrl_get (第 20页)接口后，才能调用该接口。否则，调用无效。

### 3.1.5. pinctrl_lookup_state

该接口有以下参数。

- **简介**:

```
struct pinctrl_state *pinctrl_lookup_state(struct pinctrl
 *p, const char *name)
```

- **描述**: 该接口用于从pin控制句柄中检索状态句柄。
- **参数**:
  - **p**: 要释放的pin控制句柄。
  - **name**: 您希望检索到的状态名称。
- **返回值**:
  - **成功**: 从pin控制句柄的状态句柄。
  - **失败**: 错误代码。

### 3.1.6. pinctrl_select_state

该接口有以下参数。

- **简介**:

```
int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state
 *state)
```

- **描述**: 该接口用于选择、激活或编程pin控制状态到“HW”。
- **参数**:
  - **p**: 请求配置的设备的pin控制句柄。
  - **state**: 用于选择、激活或编程的状态句柄。
- **返回值**:
  - **成功**: 0。
  - **失败**: 错误代码。

## 3.2. GPIO接口

昉·惊鸿7110的GPIO模块有以下GPIO接口。

### 3.2.1. gpio_request

该接口有以下参数。

- **简介**:

```
int gpio_request(unsigned gpio, const char * label)
```

- **描述**: 该函数用于请求访问GPIO接口。
- **参数**:
  - **gpio**: GPIO索引号。
  - **label**: GPIO名称。如果您不清楚，请留为空。
- **返回值**:
  - **成功**: 0。
  - **失败**: 错误代码。

### 3.2.2. gpio_free

该接口有以下参数。

- **简介**:

```
void gpio_free(unsigned gpio)
```

- **描述**: 该函数用于释放GPIO接口。
- **参数**:
  - **gpio**: GPIO索引号。
- **返回值**: 无。

### 3.2.3. gpio_direction_input

该接口有以下参数。

- **简介**:

```
int gpio_direction_input(unsigned gpio)
```

- **描述**: 该函数用于将GPIO接口设置为输入。
- **参数**:
  - **gpio**: GPIO索引号。
- **返回值**:
  - **成功**: 0。
  - **失败**: 错误代码。

### 3.2.4. gpio_direction_output

该接口有以下参数。

- **简介**：

```
int gpio_direction_output(unsigned gpio, int value)
```

- **描述**：该函数用于将GPIO接口设置为输出。
- **参数**：
  - **gpio**：GPIO索引号。
  - **value**：预期电平，0为低，1为高。
- **返回值**：
  - **成功**：0。
  - **失败**：错误代码。

### 3.2.5. gpio_get_value

该接口有以下参数。

- **简介**：

```
int gpio_get_value(unsigned gpio)
```

- **描述**：该接口用于从目标GPIO接口获取电平。电平用于定义GPIO是输入还是输出。
- **参数**：
  - **gpio**：GPIO索引号。
- **返回值**：
  - **成功**：目标GPIO接口电平：0为低，1为高。
  - **失败**：-1。

### 3.2.6. gpio_set_value

该接口有以下参数。

- **简介**：

```
void gpio_set_value(unsigned gpio, int value)
```

- **描述**：该函数用于从目标GPIO接口设置电平。该电平可用于将GPIO从输入改为输出。

**注：**
GPIO必须设置为输出，否则，此功能可能无效。

- **参数**：
  - **gpio**：GPIO索引号。
  - **value**：预期电平：0为低，1为高。
- **返回值**：无。

## 3.2.7. of_get_named_gpio

该接口有以下参数。

- **简介**：

```
int of_get_named_gpio(const struct device_node *np, const char
 *propname, int index)
```

- **描述**：该函数用于获取一个GPIO接口，您可以通过它使用GPIO应用程序接口（API）。
- **参数**：
  - **np**：获取GPIO的设备节点。
  - **propname**：包含GPIO说明符的属性的名称。
  - **index**：GPIO的索引。
- **返回值**：
  - **成功**：您可以通过它使用GPIO API的GPIO索引号。
  - **失败**：错误代码。

## 3.2.8. of_get_named_gpio_flags

该接口有以下参数。

- **简介**：

```
int of_get_named_gpio_flags(const struct device_node *np, const char
 *list_name, int index, enum of_gpio_flags *flags)
```

- **描述**：该函数用于从DTS文件中获取GPIO号，并分析GPIO的属性。
- **参数**：
  - **np**：获取GPIO的设备节点。
  - **propname**：包含GPIO说明符（specifiers）的属性的名称。

- **index**：GPIO的索引
- **flags**：枚举of_gpio_flags变量，包括IO配置、上拉和下拉设置，驱动能力设置等。

• **返回值**：

- **成功**：GPIO索引号。
- **失败**：错误代码。

# 4. 示例用例

## 4.1. 使用Pin驱动DTS

这些驱动主要用于配置pin的常用功能，例如：

- 通用的GPIO仅用于输入、输出和中断。
- 功能性GPIO主要用于复用pin，例如，pin转URAT、I2C以及特殊功能。
- 部分pin既可以作为通用GPIO，又可以作为功能性GPIO。

### 4.1.1. 通用GPIO

以下代码块为一个通用GPIO设备树配置的示例：

```
&sdio0 {
        clock-frequency = <102400000>;
        max-frequency = <200000000>;
        card-detect-delay = <300>;
        bus-width = <4>;
        broken-cd;
        cap-sd-highspeed;
        post-power-on-delay-ms = <200>;
        cd-gpios = <&gpio 23 0>;
        status = "okay";
};
```

通用GPIO的输入、输出和中断均在DTS配置文件中配置。

在`cd-gpios = <&gpio 23 0>`行中，请确保您理解以下值的使用：

- **gpio**：GPIO控制器。
- **23**：GPIO索引号。
- **0**：激活电平状态。0：低电平激活；1：高电平激活。

### 4.1.2. 功能性GPIO

以下代码块为一个功能性GPIO设备树配置的示例：

```
        pcie0_perst_default: pcie0_perst_default {
                perst-pins {
                        sf,pins = <PAD_GPIO26>;
                        sf,pinmux = <PAD_GPIO26_FUNC_SEL 0>;
```

```
                        sf,pin-ioconfig = <IO(GPIO_IE(1))>;
                        sf,pin-gpio-dout = <GPO_HIGH>;
                        sf,pin-gpio-doen = <OEN_LOW>;
                };
        };

        pcie0_perst_active: pcie0_perst_active {
                perst-pins {
                        sf,pins = <PAD_GPIO26>;
                        sf,pinmux = <PAD_GPIO26_FUNC_SEL 0>;
                        sf,pin-ioconfig = <IO(GPIO_IE(1))>;
                        sf,pin-gpio-dout = <GPO_LOW>;
                        sf,pin-gpio-doen = <OEN_LOW>;
                };
        };

        pcie0_power_active: pcie0_power_active {
                power-pins {
                        sf,pins = <PAD_GPIO32>;
                        sf,pinmux = <PAD_GPIO32_FUNC_SEL 0>;
                        sf,pin-ioconfig = <IO(GPIO_IE(1))>;
                        sf,pin-gpio-dout = <GPO_HIGH>;
                        sf,pin-gpio-doen = <OEN_LOW>;
                };
        };
```

```
&pcie0 {
        pinctrl-names = "perst-default", "perst-active", "power-active";
        pinctrl-0 = <&pcie0_perst_default>;
        pinctrl-1 = <&pcie0_perst_active>;
        pinctrl-2 = <&pcie0_power_active>;
        status = "okay";
};
```

在上例中，请确保您理解以下值的使用：

- pinctrl-0中的“`perst-default`”参数值表示正常工作模式的pin配置。
- pinctrl-1中的“`perst-active`”参数值表示激活模式的pin配置。

# 4.2. 使用API驱动DTS

## 4.2.1. 获取Pin控制资源

通常，使用`devm_pinctrl_get`接口，您可以获取一个设备的所有pin的状态。

以下代码为一个示例：

```
        struct device *dev = &pcie->pdev->dev;

        pcie->pinctrl = devm_pinctrl_get(dev);
        if (IS_ERR_OR_NULL(pcie->pinctrl)) {
                dev_err(dev, "Getting pinctrl handle failed\n");
                return -EINVAL;
        }
```

### 4.2.2. 获取Pin控制状态

通过使用`pinctrl_lookup_state`接口，您可以获取一个设备的所有pin的状态。

以下代码为一个示例：

```
        pcie->perst_state_def
                = pinctrl_lookup_state(pcie->pinctrl, "perst-default");
        if (IS_ERR_OR_NULL(pcie->perst_state_def)) {
                dev_err(dev, "Failed to get the perst-default pinctrl
han-dle\n");
                return -EINVAL;
        }

        pcie->perst_state_active
                = pinctrl_lookup_state(pcie->pinctrl, "perst-active");
        if (IS_ERR_OR_NULL(pcie->perst_state_active)) {
                dev_err(dev, "Failed to get the perst-active pinctrl
handle\n");
                return -EINVAL;
        }

        pcie->power_state_active
                = pinctrl_lookup_state(pcie->pinctrl, "power-active");
        if (IS_ERR_OR_NULL(pcie->power_state_active)) {
                dev_err(dev, "Failed to get the power-default pinctrl
han-dle\n");
                return -EINVAL;
        }
```

### 4.2.3. 设置Pin控制状态

通过使用`pinctrl_select_state`接口，您可以配置一个设备的所有pin的状态。

以下代码为一个示例：

```
        if (pcie->power_state_active) {
                ret = pinctrl_select_state(pcie->pinctrl,
pcie->power_state_active);
                if (ret)
```

```
                        dev_err(dev, "Cannot set power pin to high\n");
        }

        if (pcie->perst_state_active) {
                ret = pinctrl_select_state(pcie->pinctrl,
pcie->perst_state_active);
                if (ret)
                        dev_err(dev, "Cannot set reset pin to low\n");
        }
```

## 4.3. 实现中断功能

您可以使用gpiod_to_irq接口来加载虚拟中断信号，然后调用中断函数。

以下代码为一个示例：

```
        if (!(host->caps & MMC_CAP_NEEDS_POLL))
                irq = gpiod_to_irq(ctx->cd_gpio);

        if (irq >= 0) {
                if (!ctx->cd_gpio_isr)
                        ctx->cd_gpio_isr = mmc_gpio_cd_irqt;
                ret = devm_request_threaded_irq(host->parent, irq,
                        NULL, ctx->cd_gpio_isr,
                        IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING |
IRQF_ONESHOT,
                        ctx->cd_label, host);
                if (ret < 0)
                        irq = ret;
        }
```

# 5. 常见问题集

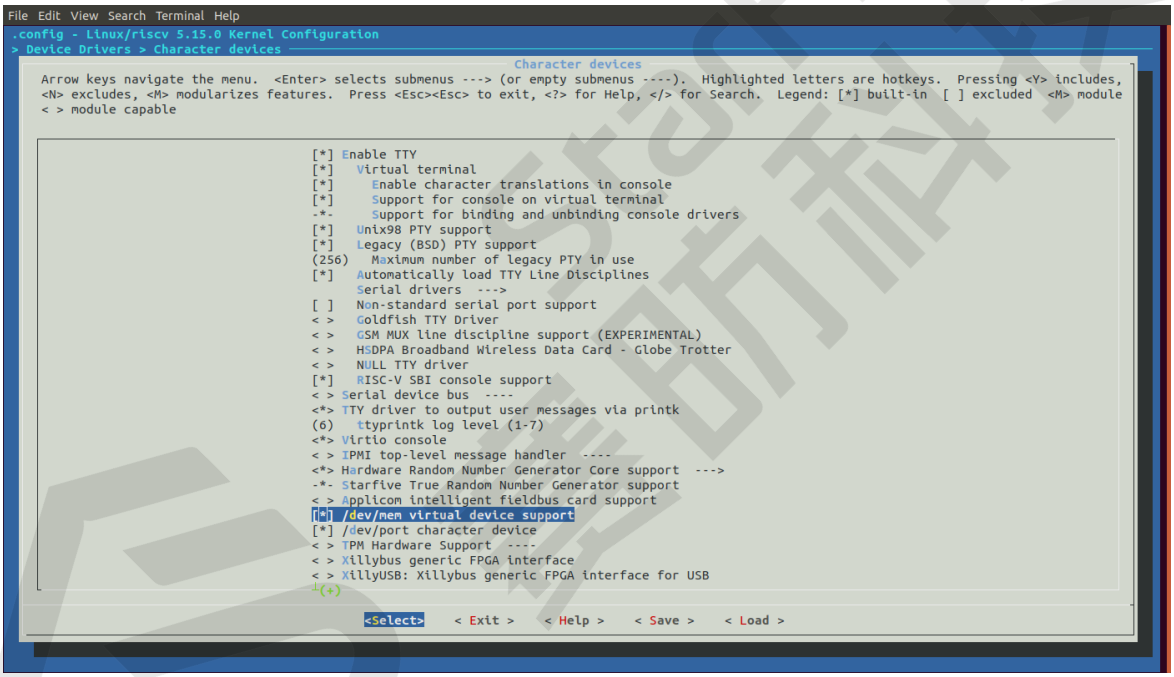
## 5.1. 通用调试方法

### 5.1.1. 通过devmem读写寄存器

按照以下步骤进行调试。

1. 进入**/dev/mem virtual device support**菜单。

**图 5-1 通过devmem读写寄存器**




2. 执行以下命令，运行读写操作。

   - 执行以下命令，从物理地址"0x1304006c"中读取寄存器值：

     ```
     devmem 0x1304006c
     ```

   - 执行以下命令，将寄存器值写入物理地址"0x1304006c"。例如，将低位8位写入0x5D：

     ```
     devmem 0x1304006c 32 0x005b5c5d
     ```

     然后，您可以再次执行上述读取命令，以验证写操作是否成功。下面的代码块显示了这些命令并返回。

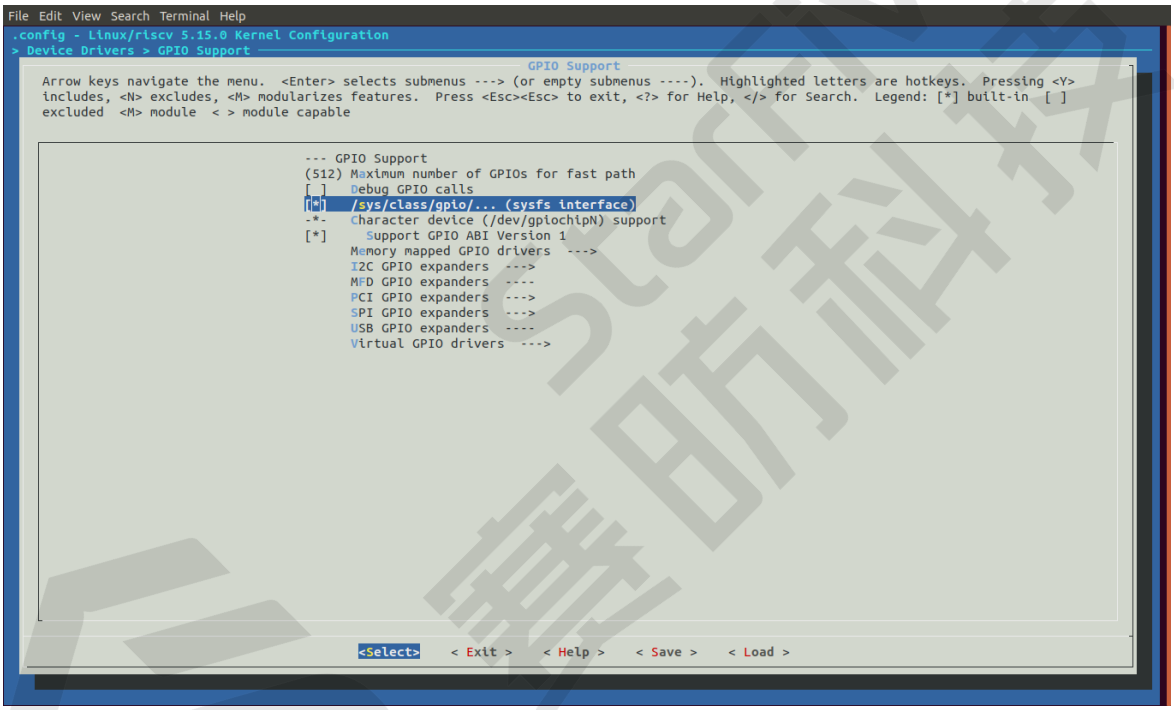
```
#    devmem 0x1304006c
0x00SBSC00
#    devmem 0x1304006c 32 0x005b5c5d
#    devmem 0x1304006c
0X005B5C5D
```

## 5.1.2. 使用sysfs来调试

按照以下步骤进行调试。

1. 进入**/sys/class/gpio/...(sysfs interface)**菜单。

**图 5-2 使用sysfs来调试**




2. 执行以下命令进行调试。

```
# cd /sys/class/gpio/
# ls
export gpiochip0 gpiochip64 unexport
# echo 44 > export
# ls
export gpio44 gpiochip0 gpiochip64 unexport
# cd gpio44/
# ls
active_low direction subsystem value
device edge uevent
# cat direction
in
# cat value
```

```
1
```