D1 Tina Linux 红外开发指南

版本号: 1.0

发布日期: 2021.04.12

版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.04.12	AWA1611	初始版本

目　录

插　图

1 概述

1.1 文档简介

本文主要介绍 Linux 内核红外遥控子系统（Linux Infrared Remote Control System）的基础知识，Linux 内核 RC 子系统框架，红外发送和接收模块驱动配置以及验证流程。

1.2 目标读者

本文档适用于开发、移植红外驱动的开发人员使用。

1.3 适用范围

表 1-1: 适用产品列表

产品名称	内核版本	驱动文件
D1	Linux-5.4	sunxi-ir-dev.c sunxi-ir-tx.c

1.4 相关术语描述

表 1-2: 术语说明表

术语	说明
SUNXI IR 驱动	全志平台红外模块的 IR 驱动
IR	Infrared Remote 红外
RC	Remote Control 远程遥控，文中代表红外

2 红外遥控基础知识

2.1 红外遥控简介

红外遥控协议有有很多，比如 RC-5，RC-6，NEC，SIRC 等，不过协议都比较简单，基本上都是以脉冲宽度或脉冲间隔来编码。

当遥控器上按下按键时，遥控器逻辑单元会产生一个完整的逻辑脉冲波形，这个波形上包含了遥控命令的信息，即红外传输的基带信号。这个波形被送到遥控器的调制单元，经调制单元调制成高频的红外电磁波信号，并由发光二极管发射出去。如下图的左边模块。

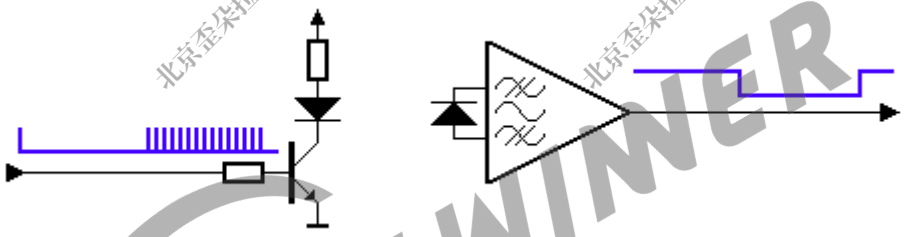

图 2-1: 红外遥控的信号的硬件原理

红外电磁波信号现在一般使用一体化接收头接收，接收头同时完成了信号的解调和放大，其输出信号就是红外的基带脉冲信号。解调后的信号可直接送入信号处理器中由处理器对脉冲波形进行解码，也就是将经编码的脉冲信号翻译成逻辑数字。根据不同的控制协议，解码方式不同。如图 2-2 的红外接收头，一根线用于输出脉冲信号，其他两根是电源线和地线。

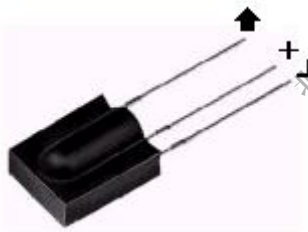

图 2-2: 红外接收头

2.2 红外编码协议

下面以最常见的 NEC 协议为例说明。

2.2.1 NEC 协议特征

- 8 位地址和 8 位命令长度
- 每次传输两遍地址（用户码）和命令（按键值）
- 通过脉冲串之间的时间间隔来实现信号的调制（PPM）
- 38KHz 载波
- 每位的周期为 1.12ms（低电平）或者 2.25ms（高电平）

2.2.2 NEC 协议编码

NEC 协议逻辑 1 与逻辑 0 的表示如图所示，使用脉冲宽度对每一比特位进行编码，逻辑 1 为 2.25ms，脉冲时间 560us；逻辑 0 为 1.12ms，脉冲时间 560us，所以我们根据脉冲时间长短来解码。另外，推荐载波占空比为 1/3 至 1/4。

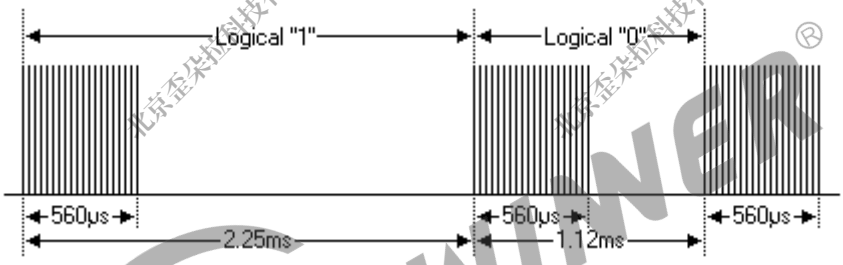

图 2-3: NEC 协议的逻辑 1 与逻辑 0

2.2.3 帧格式

当按下遥控器上的按键时，遥控器会发送一个命令信号，这个信号就是一个帧，它包含了引导码、地址 (设备) 字段和命令字段。首先发送 9ms 的 AGC 自动增益控制脉冲，在早期的红外接收器中用来设置增益。接着是 4.5ms 空闲（低电平），这个是 NEC 协议的引导码。然后是 8bit 的地址码及 8bit 的地址码的反码，接下来是命令及其反码。反码可以用来校验，提高按键的准确性。协议规定，**地址字段和命令字段低位先发送**。一帧总的传输时间是固定的，因为每一位都有反码传送，即 67.42ms。

除了引导码、用户码和数据码以外，协议最后还有 1bit 由 560us 脉冲的停止位，其后是一个较长时间的空闲，可以通过这个超时判断一帧数据接收完毕。

图 2-4: NEC 协议帧格式

如果遥控器上的按键一直按着，这个命令只发生一次，但是每隔 110ms 会发送重复码，直到遥控器按键释放。重复码比较简单，由一个 9ms 的脉冲、2.25ms 低电平和 560us 的脉冲组成，如下图所示。

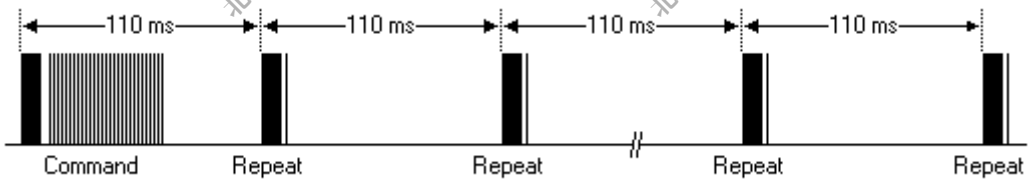

图 2-5: NEC 协议 repeat 帧格式

需要注意的是，一般红外一体接收头为了提高接受灵敏度。输入高电平，其输出的是相反的低电平，实际应用场景下输入给 CPU 处理的波形如下图所示。对于脉冲波形的解码，一般用一个专门的硬件单元完成，也可以在 CPU 中利用如 GPIO 等检测接收器输出的波形，然后使用 CPU 进行软件解码。

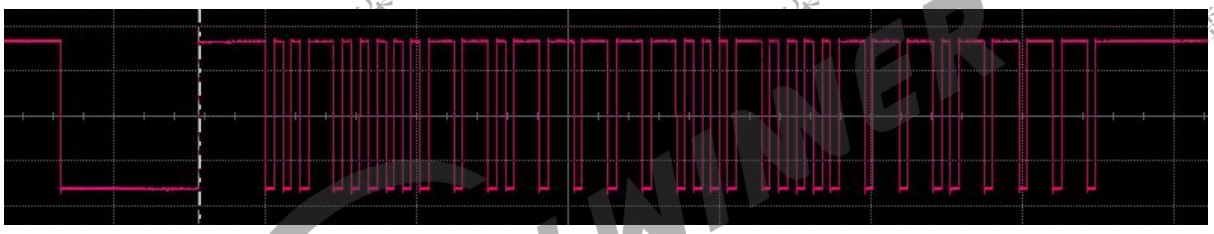

图 2-6: NEC 协议实际波形

3 Linux 内核 RC 子系统

Linux 上 RC 子系统提供了对红外控制的支持，它包含几个部分：RC 核心（RC Core）、协议原始脉冲解码器 (RC Decoders)、按键映射表（RC Keymaps）、红外输入设备驱动（RC Device Driver）和 LIRC（Linux Infrared Remote Control ）接口。

其中，RC Core 负责 RC 设备及事件管理，RC Decoders 实现对 IR 数据的解析，并通过 Input 系统上报按键值；LIRC Interface 实现与 user space 的交互，将接收到 IR RAW 数据提供给应用层，也可以将应用写入的数据发送给 RC Core，并通过 RC Device 发送出去。

RC 子系统的相关代码路径为 tina/lichee/linux-5.4/drivers/media/rc

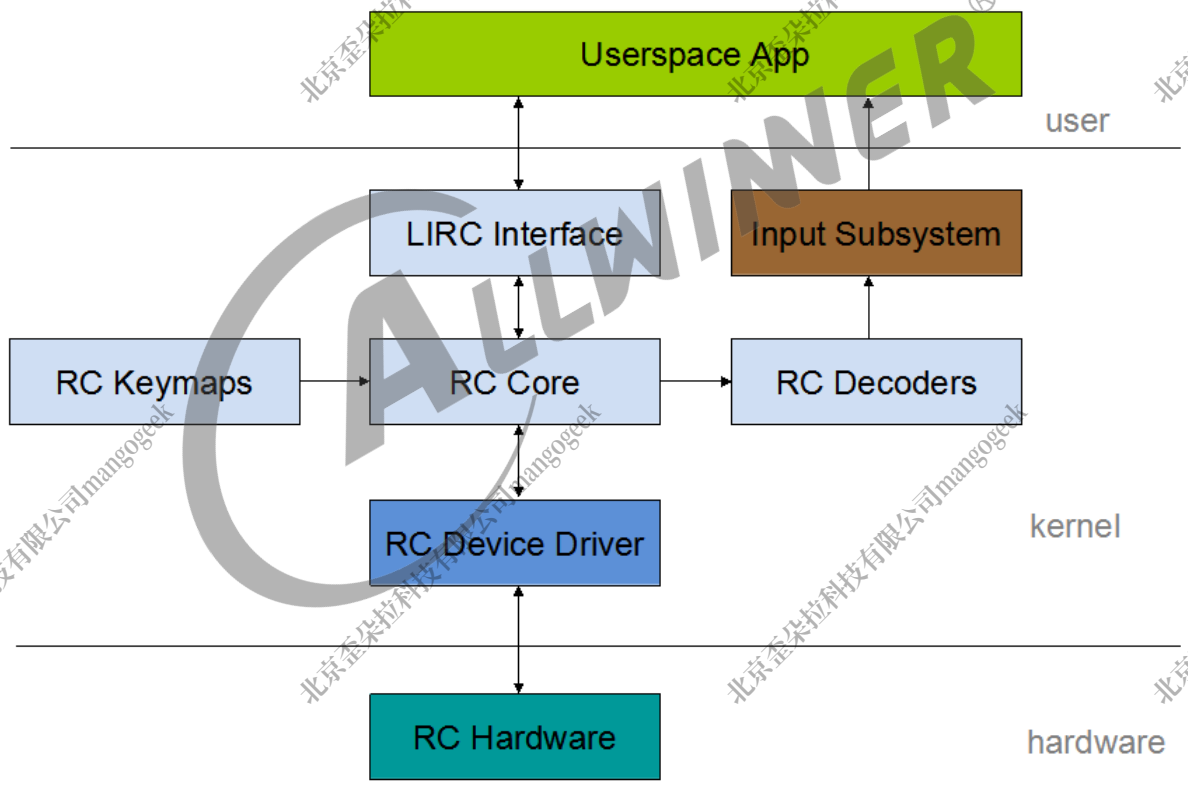

图 3-1: Linux 内核 RC 子系统框架图

3.1 RC Decoders

RC decoders 模块用软件的方法实现对原始脉冲进行解码。解码器用一个 ir_raw_handler 结构体表示。

结构体所在位置：drivers/media/rc/rc-core-priv.h

Linux-5.4 RC 子系统：

```
struct ir_raw_handler {
        struct list_head list;

        u64 protocols; /* which are handled by this handler */
        int (*decode)(struct rc_dev *dev, struct ir_raw_event event);
        int (*encode)(enum rc_proto protocol, u32 scancode,
                      struct ir_raw_event *events, unsigned int max);
        u32 carrier;
        u32 min_timeout;

        /* These two should only be used by the mce kbd decoder */
        int (*raw_register)(struct rc_dev *dev);
        int (*raw_unregister)(struct rc_dev *dev);
};
```

RC Decoder 通过如下两个函数进行注册和卸载（drivers/media/rc/rc-main.c）：

```
int ir_raw_handler_register(struct ir_raw_handler *ir_raw_handler)
void ir_raw_handler_unregister(struct ir_raw_handler *ir_raw_handler)
```

所有注册的 Decoder 放在一个全局链表 ir_raw_handler_list 中，有 IR 事件到来时，RC Core 会调用注册的解码器对 IR 数据解码。注意，此时当任何一个解码器返回一个错误，后面的解码器不会被执行，所以不要将不使用的解码器加载到内核中。LIRC Decoder 也做为一个特殊 Decoder 进行注册，注册时 raw_register 函数被调用创建 lirc dev，应用层边可以通过标准文件接口与 lirc 进行交互，即前面提到的 lirc interface。

Decoder 的主体就是一个 decode 函数，RC Core 会将驱动报告的每个脉冲都传递到 decode 函数，而 decode 函数的实现就是一个状态机，每一次输入导致进入下一状态，直到一次解码完成，然后返回起始状态进行下次解码。

LIRC 中，每个脉冲 (包括脉冲间隔) 用一个 ir_raw_event 结构表示：

```
struct ir_raw_event {
    union {
        u32             duration; /*时间宽度，以ns为单位，一个0ns的脉冲表示重新开始解码 */
        u32             carrier;
    };
    u8                      duty_cycle;

    unsigned                pulse:1; //是脉冲，还是空闲
    unsigned                reset:1;
    unsigned                timeout:1;
    unsigned                carrier_report:1;
};
```

LIRC 中对于脉冲宽度的比较使用 eq_margin()、geq_margin() 函数，它允许宽度值在二分之一单元上下波动。RC Decoder 的完整实现可参考 NEC Decoder 的实现：

```
bool geq_margin(unsigned d1, unsigned d2, unsigned margin)
bool eq_margin(unsigned d1, unsigned d2, unsigned margin)
```

3.2 RC Keymaps

RC Keymaps 都放在/linux-5.4/drivers/media/rc/keymaps 目录下。不同的遥控器有不同的按键映射，按键映射模块的作用就是将扫描码与 Linux input 系统的标准事件对应起来。用户可以自己定义按键映射，比如说可以在 SDK 中添加 rc-aw-nec.c 文件，就是添加的自定义 NEC Code 映射表，其主体就是一个 rc_map_table 结构数组，每个元素是一对按键映射，即扫描值和 input 系统的 key code 对应。

```
static struct rc_map_table aw_nec[] = {
    { 0x04fb1ae5, KEY_POWER},   /* power */
    { 0x04fb23dc, KEY_MUTE},    /* mute */
    { 0x04fb13ec, KEY_1},
    { 0x04fb10ef, KEY_2},
     .......
    { 0x04fb43bc, KEY_VOD},
};
```

通过 rc_map_register 函数注册一个按键映射到 RC 系统中，rc_map_list 保存已注册的 rc_map，新定义 rc_map 时，rc_type 指定按键映射使用的 IR 协议，在 RC Decoder 上报按键时，会与这里定义的 IR 协议类型进行匹配，如果匹配成功则取对应的按键上报。RC 设备驱动中设定的 keymap name 对应此处定义的 name 成员，rc core 根据 name 匹配当前使用的映射表。

```
static struct rc_map_list aw_nec_map = {
    .map = {
        .scan    = aw_nec,
        .size    = ARRAY_SIZE(aw_nec),
        .rc_type  = RC_TYPE_NEC,
        .name    = RC_MAP_AW_NEC,
    }
};
```

3.3 RC 设备驱动

RC（Remote Control）设备驱动负责向 RC Core 报告脉冲 RAW 数据，以及将红外数据调制成协议波形后通过硬件发送。RC 设备用 rc_dev 结构描述（include/media/rc-core.h）：

```
struct rc_dev {
        struct device                   dev;
        bool                            managed_alloc;
        const struct attribute_group    *sysfs_groups[5];
        const char                      *device_name;
```

```
        const char                      *input_phys;
        struct input_id                 input_id;
        const char                      *driver_name;
        const char                      *map_name;
        struct rc_map                   rc_map;
        struct mutex                    lock;
        unsigned int                    minor;
        struct ir_raw_event_ctrl        *raw;
        struct input_dev                *input_dev;
        enum rc_driver_type             driver_type;
        bool                            idle;
        bool                            encode_wakeup;
        u64                             allowed_protocols;
        u64                             enabled_protocols;
        u64                             allowed_wakeup_protocols;
        enum rc_proto                   wakeup_protocol;
        struct rc_scancode_filter       scancode_filter;
        struct rc_scancode_filter       scancode_wakeup_filter;
        u32                             scancode_mask;
        u32                             users;
        void                            *priv;
        spinlock_t                      keylock;
        bool                            keypressed;
        unsigned long                   keyup_jiffies;
        struct timer_list               timer_keyup;
        struct timer_list               timer_repeat;
        u32                             last_keycode;
        enum rc_proto                   last_protocol;
        u32                             last_scancode;
        u8                              last_toggle;
        u32                             timeout;
        u32                             min_timeout;
        u32                             max_timeout;
        u32                             rx_resolution;
        u32                             tx_resolution;
#ifdef CONFIG_LIRC
        struct device                   lirc_dev;
        struct cdev                     lirc_cdev;
        ktime_t                         gap_start;
        u64                             gap_duration;
        bool                            gap;
        spinlock_t                      lirc_fh_lock;
        struct list_head                lirc_fh;
#endif
        bool                            registered;
        int                             (*change_protocol)(struct rc_dev *dev, u64 *
    rc_proto);
        int                             (*open)(struct rc_dev *dev);
        void                            (*close)(struct rc_dev *dev);
        int                             (*s_tx_mask)(struct rc_dev *dev, u32 mask);
        int                             (*s_tx_carrier)(struct rc_dev *dev, u32 carrier);
        int                             (*s_tx_duty_cycle)(struct rc_dev *dev, u32
    duty_cycle);
        int                             (*s_rx_carrier_range)(struct rc_dev *dev, u32 min,
    u32 max);
        int                             (*tx_ir)(struct rc_dev *dev, unsigned *txbuf,
    unsigned n);
        void                            (*s_idle)(struct rc_dev *dev, bool enable);
        int                             (*s_learning_mode)(struct rc_dev *dev, int enable);
```

```
61          int                             (*s_carrier_report) (struct rc_dev *dev, int enable
        );
62          int                             (*s_filter)(struct rc_dev *dev,
63                                                      struct rc_scancode_filter *filter);
64          int                             (*s_wakeup_filter)(struct rc_dev *dev,
65                                                             struct rc_scancode_filter *
        filter);
66          int                             (*s_timeout)(struct rc_dev *dev,
67                                                       unsigned int timeout);
68  };
```

设备注册和卸载：

```
1  int rc_register_device(struct rc_dev *dev)
2  void rc_unregister_device(struct rc_dev *dev)
```

RC 设备注册流程如下：

1. 分配一个 RC 设备，rc_allocate_device();
2. 对 RC 设备进行一些初始化设置，包括 input 设备的参数，RC 设备的驱动类型，使用的 key_map, 支持的 IR decoder 协议；
3. 将分配的 RC 设备结构的地址作为参数调用 rc_register_device() 注册，以后与 lirc 核心的交互都是通过这个 rc 设备结构的地址进行的。

对于可直接从硬件读取到扫描码的设备，RC 设备驱动类型 (driver_type) 定义为 RC_DRIVER_SCANCODE，可用以下函数包括扫描码事件：

```
1  void rc_keydown(struct rc_dev *dev, enum rc_type protocol, u32 scancode, u8 toggle)
```

对于只能获得原始脉冲的设备，需先调用下面函数报告每个脉冲和脉冲间隔：

```
1  int ir_raw_event_store(struct rc_dev *dev, struct ir_raw_event *ev)
2  int ir_raw_event_store_edge(struct rc_dev *dev, bool pulse)
```

第一个函数要求驱动自己填充 ir_raw_event 结构，第二个函数自动生成一个 ir_raw_event 结构，脉冲宽度根据前后两次调用 ir_raw_event_store_edge() 函数的时间间隔计算，这个会在 RC CORE 完成。调用这两个函数后需要再调用 ir_raw_event_handle() 函数启动 RC decoder 开始解码。

ir_raw_event 的 type 参数指定是何脉冲，具体定义为：

```
1  enum raw_event_type
2  {
3      IR_SPACE        = (1 << 0),
4      IR_PULSE        = (1 << 1),
5      IR_START_EVENT  = (1 << 2),
6      IR_STOP_EVENT   = (1 << 3),
7  };
```

其他两个驱动常用的接口：

```
void rc_repeat(struct rc_dev *dev)  /*重复上次按键 */
u32 rc_g_keycode_from_table(struct rc_dev *dev, u32 scancode)  /*从keymap获得扫描码对应的按键*/
```

如果 RC 设备支持 IR 发送，需要在设备驱动中实现如下几个函数，其中 tx_ir 为 IR 发送函数，s_tx_carrier 设置 IR 的载波频率，s_tx_duty_cycle 设置 IR 载波的占空比，如 NEC 为 33（即 1/3），这样在应用层就可以 ioctl 调用到驱动这些函数。

```
int (*s_tx_carrier)(struct rc_dev *dev, u32 carrier);
int (*s_tx_duty_cycle)(struct rc_dev *dev, u32 duty_cycle);
int (*tx_ir)(struct rc_dev *dev, unsigned *txbuf, unsigned n);
```

3.4 RC 对按住按键时重复事件的处理

首先，重复是自动的，它使用了 input 子系统的 REPEAT 功能。当第一次向 RC core 报一个扫描码事件时，RC Core 会向 input 子系统报告相应的按键事件，并启动一个定时器，该定时器在超时后会上报对应按键的松开事件。之后若在 250ms(这个值等于 input 子系统的 rep 延时的默认值) 内该设备又报告了同一个扫描码，这时只是将定时器再推后 250ms，并不报告新的按键事件，也就是说按键的重复由 input 系统处理。

说明

这种设计主要是考虑到遥控器的限制，有的遥控器没有按键按下和松开之分（虽然在按下和松开时都有脉冲，但没有区分字段，而且有时可能会丢失信号）。

4 Sunxi 平台 IR RC 驱动配置

4.1 Kernel menuconfig 配置

配置路径：

```
Device Drivers
└─>Remote Controller support
```

操作图示：

1. 在 root 路径下执行 make kernel_menuconfig，进入 Device Drivers，然后按 PC 键盘的 R 键，可以快速定位到 “Remote Controller support” 选项中，然后进入该选项，选上图 4-1 中的选项。

这里第一个选项是用于启用 keymap 的，第二个启用应用层接口，第三个是用于选择解码方式的，第四个是用于选择设备驱动。

2. 进入 Remote controller decoders 页面，选择所要支持的 IR 协议，这里选择的是 NEC 协议，如图 4-2 所示：

3. 最后选择设备驱动，以 R528 为例，这里使用的是 SUNXI IR RX remote control 以及 SUNXI IR TX remote control，如图 4-3 所示：

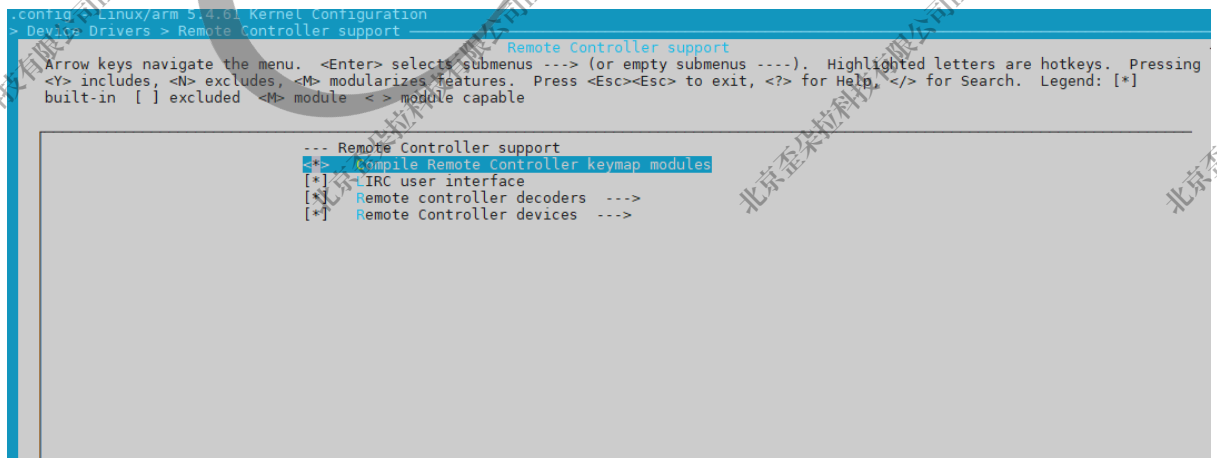

图 4-1: Linux-5.4 RC 配置图

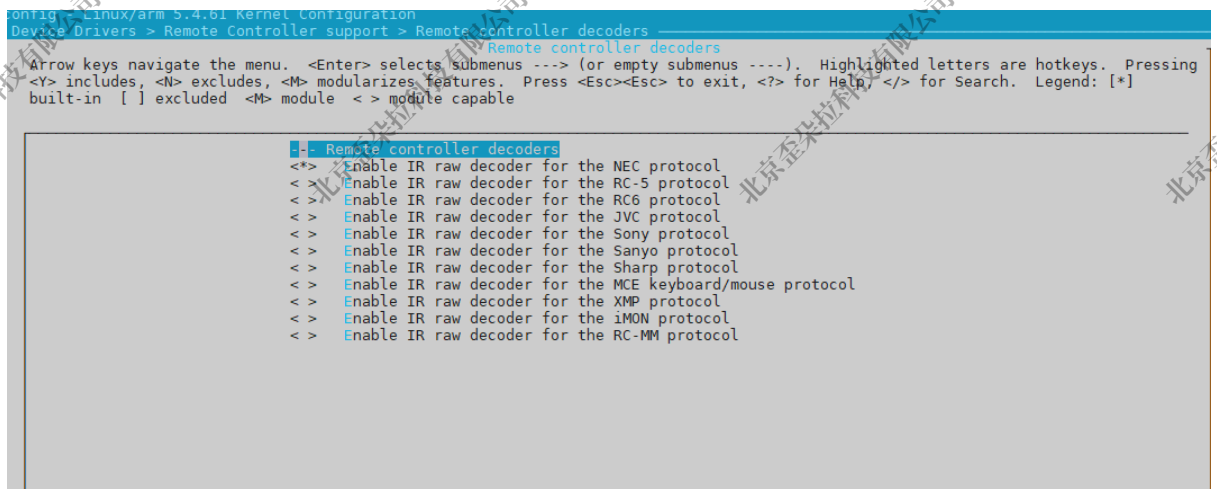

图 4-2: Linux-5.4 decoders 选择界面

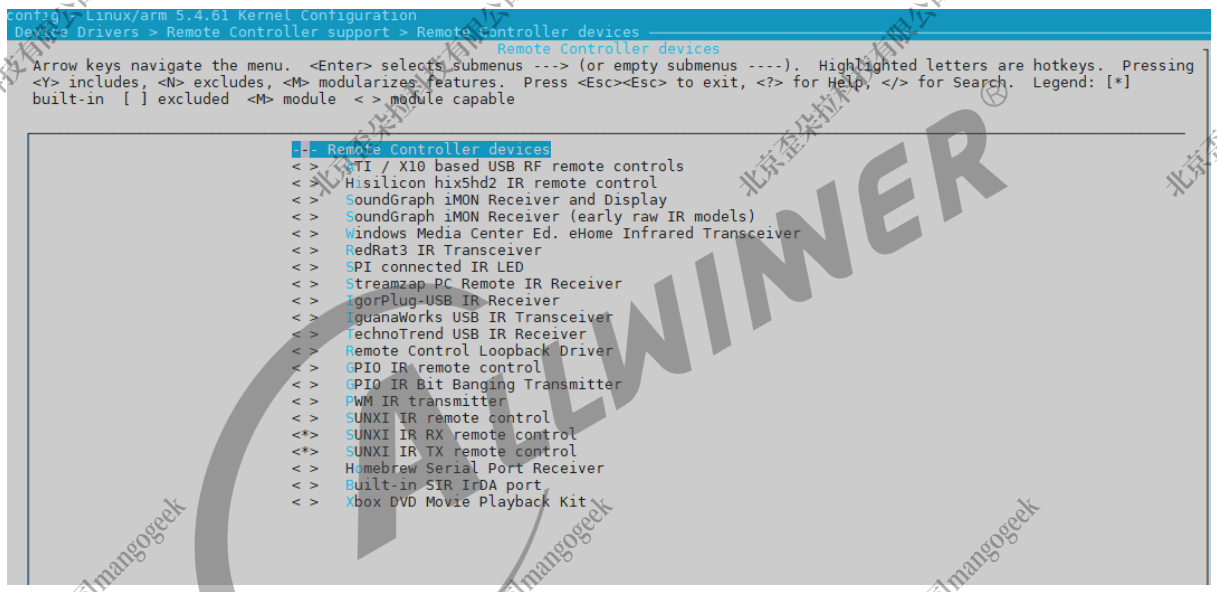

图 4-3: Linux-5.4 SUNXI IR 驱动配置界面

4.2 DTS 配置

D1 方案的 dts 文件如下：

```
tina/lichee/linux-5.4/arch/riscv/boot/dts/sunxi/sun20iw1p1.dtsi
```

IR 配置如下：

```
s_cir0: s_cir@7040000 {
        compatible = "allwinner,s_cir";
        reg = <0x0 0x07040000 0x0 0x400>;
        interrupts = <GIC_SPI 151 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&r_ccu CLK_R_APB0_BUS_IRRX>, <&dcxo24M>, <&r_ccu CLK_R_APB0_IRRX>;
        clock-names = "bus", "pclk", "mclk";
```

```
        resets = <&r_ccu RST_R_APB0_BUS_IRRX>;
        supply = "";
        supply_vol = "";
        status = "disabled";
};

ir1: ir@2003000 {
        compatible = "allwinner,ir_tx";
        reg = <0x0 0x02003000 0x0 0x400>;
        interrupts = <GIC_SPI 19 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&ccu CLK_BUS_IR_TX>, <&dcxo24M>, <&ccu CLK_IR_TX>;
        clock-names = "bus", "pclk", "mclk";
        resets = <&ccu RST_BUS_IR_TX>;
        status = "disabled";
};
```

s_cir0 是红外接收端的节点，ir1 是红外发送端的节点。

说明

方案的 *dts* 配置只配置红外与 *IC* 相关的配置，比如说寄存器地址、时钟、中断等等的配置。但是引脚因为会跟随着板型变化而变化，*PIN* 的配置就不放在该文件了。

boatd.dts 可以通过命令 cconfigs 跳转，路径为：

```
tina/device/config/chips/d1/configs/nezha/linux/board.dts
```

配置如下：

```
&pio {
.................//其他pin配置
        s_cir0_pins_a: s_cir0@0 {
                pins = "PB1";
                function = "ir";
                drive-strength = <10>;
                bias-pull-up;
        };

        ir1_pins_a: ir1@0 {
                pins = "PB0";
                function = "ir";
                drive-strength = <10>;
                bias-pull-up;
        };
.................//其他pin配置
};

&s_cir0 {
        pinctrl-names = "default";
        pinctrl-0 = <&s_cir0_pins_a>;
        status = "disabled";
};

&ir1 {
        pinctrl-names = "default";
        pinctrl-0 = <&ir1_pins_a>;
        status = "disabled";
};
```

4.3 添加 RC keymap

在 tina/lichee/linux-5.4/drivers/media/rc/keymaps 添加 rc_keymap 源文件，定义 IR 扫描值与 linux 标准按键 keycode 的映射，如添加的 rc-aw-nec.c，定义了 NEC 协议扫描值与一个遥控器按键的映射表，并通过 rc_map_register 注册，用户如果想在 driver 下解析 IR，可以自定义相关键值。

另外，在 include/media/rc-map.h 中添加 rc_keymap 的定义，这样在 DTS 中就可以指定这个添加的 key 映射表，如 linux,rc-map-name = “rc-aw-nec”;

```
1 #define RC_MAP_AW_NEC   "rc-aw-nec"
```

5 IR 驱动验证

5.1 文件节点

RC 驱动加载成功后，在 dev 下可以看到 lirc0 设备节点，应用层通过 dev/lirc0 节点同 RC 设备驱动进行数据交互。/sys/class/lirc 目录是 rc 的系统文件目录，可以看到有如下属性：

```
root@TinaLinux:/sys/devices/platform/soc/soc@03000000:gpio-ir/rc/rc0# ls
device     input1     lirc0      power      protocols  subsystem  uevent
```

查看 protocols，可以知道 RC 设备支持的 ir 协议：

```
root@TinaLinux:/sys/devices/platform/soc/soc@03000000:gpio-ir/rc/rc0# cat protocols
rc-5 [nec] rc-6 [lirc]
```

5.2 按键事件

如果 RC 设备驱动使能了 RC Decoder，收到 IR RAW 数据后会转为标准按键上报，通过 getevent 可以查看遥控器按下和释放，并且可以查看到按键值和扫描值，参考如下：

```
root@TinaLinux:/sys/devices/platform/soc/soc@03000000:gpio-ir/rc/rc0# getevent
add device 1: /dev/input/event0
  name:     "sunxi-keyboard"
add device 2: /dev/input/event1
  name:     "sunxi_gpio_ir"
poll 3, returned 1
/dev/input/event1: 0004 0004 04fb13ec /*scan code*/
poll 3, returned 1
/dev/input/event1: 0001 0002 00000001/*key down*/
poll 3, returned 1
/dev/input/event1: 0000 0000 00000000
poll 3, returned 1
/dev/input/event1: 0004 0004 04fb13ec
poll 3, returned 1
/dev/input/event1: 0000 0000 00000000
poll 3, returned 1
/dev/input/event1: 0001 0002 00000000/*key up*/
poll 3, returned 1
/dev/input/event1: 0000 0000 00000000
```

说明

***getevent** 是一款可以获取 **input** 事件并整理打印 **input** 事件信息的工具。*

D1 方案的 SDK 没有默认选上 getevent，在 Tina 根目录下运行`make menuconfig`，选上下面该项：

```
Utilities
    └─>getevent
```

5.3 测试应用

Tina SDK 里面已包括了 RC 框架下的 IR 驱动的测试应用 gpio_ir_test，源码路径为：package/utils/gpio_ir_test，在 menuconfig 中配置上即可使用，menuconfig 路径如下：

```
Utilities
    └─>gpio_ir_test
```

测试程序包括两部分，IR 接收（RX）和 IR 发送（TX）两部分，执行 gpio_ir_test 命令如下：

```
root@TinaLinux:/# gpio_ir_test
usage: gpio_ir_test [options]

gpio ir receive test:
        gpio_ir_test rx

gpio ir send test:
        gpio_ir_test tx <code>

gpio ir send test:
        gpio_ir_test loop
```

5.3.1 测试 IR RX 功能

测试 IR RX 功能，可以直接执行。gpio_ir_test rx，应用层可以通过 poll 的方式向 rc 设备读取 IR RAW 数据，这样也可以在 user space 进行 IR 协议解析。驱动层也进行解析，RC Core 通过 input 上报解析的按键，驱动支持的协议，可以在 menuconfig 进行配置。如果希望在应用进行协议解析，驱动无需解析并上报按键。

需注意的是，读取的 IR RAW 数据，即 IR 波形的脉冲时间（ms）是 32bit 表示的，但是实际有效时间数据为低 24bit，其高 8bit 为脉冲类型，即之前提到的 raw_event_type，用以区分脉冲的高低电平。通过 lirc 设备读取 raw 格式，每帧前面 2 个数值是代码帧之间的间隔，解析红外数据帧需要跳过。

```
1
2  ......
3      fd = open("/dev/lirc0", O_RDWR);
4          if (fd < 0) {
5                  printf("can't open lirc0 recv, check driver!\n");
6                  return 0;
7          } else {
```

```
                printf("lirc0 open succeed.\n");
        }
......
    if (!strcmp(argv[1], "rx")) {
                err = pthread_create(&tid, NULL, (void *)ir_recv_thread, NULL);
                if (err != 0) {
                        printf("create pthread error: %d\n", __LINE__);
                        goto OUT;
                }

                do {
                        usleep(1000);
                } while (!int_exit);

    }
......
void *ir_recv_thread(void *arg)
{
        int size = 0, size_t = 0;
        int i = 0;
        int dura;
        int ret;
        int total = 0;

        poll_fds[0].fd = fd;
        poll_fds[0].events = POLLIN | POLLERR;
        poll_fds[0].revents = 0;

        while (!int_exit) {
                ret = poll(poll_fds, 1, 12);
                if (!ret) {
                        //printf("time out\n");
                        printf("\n--------------------\n");
                        total = 0;
                } else {
                        if (poll_fds[0].revents == POLLIN) {
                                size = read(fd, (char *)(rx_raw_buf),
                                                GPIO_IR_RAW_BUF_SIZE);
                                size_t = size / sizeof(uint32_t);
                                for (i = 0; i < size_t; i++) {
                                        dura = rx_raw_buf[i] & 0xffffff;
                                        printf("%d ", dura);
                                        if ((total++) % 8 == 0)
                                                printf("\n");
                                }
                        }
                }
        }

        return NULL;
}
```

该次示例是使用普通家用遥控器按下按键往电路板上发送红外信号进行的，运行效果：

```
root@TinaLinux:/# gpio_ir_test rx
This is 5.* linux kernel
lirc0 open succeed.
lirc1 open succeed.
```

```
--------------------

--------------------
8966
4483 556 556 556 567 545 1679 556
556 556 556 556 567 545 567 545
567 556 1669 556 1679 545 567 556
1669 556 1679 556 1679 545 1679 556
1679 556 1669 556 567 545 1679 556
1679 556 1690 556 556 556 556 556
556 556 556 556 1679 556 556 556
556 556 556 556 1679 556 1679 556
1669 556 16306
--------------------
```

5.3.2 测试 IR TX 功能

测试发送执行 gpio_ir_test tx code（4 字节），gpio_ir_test 测试 demo 只做了 NEC 格式的编码，如果需要支持其他协议，需添加。其主要功能将发送的 code 编码为 NEC 协议脉冲，然后通过 write 函数写入 dev/lirc1 设备中。

```
1  ......
2          duty_cycle = DEFAULT_DUTY_CYCLE;
3          if (ioctl(fd1, LIRC_SET_SEND_DUTY_CYCLE, &duty_cycle)) {
4                  fprintf(stderr,
5                          "lirc0: could not set carrier duty: %s\n",
6                          strerror(errno));
7                  goto OUT;
8          }
9
10         carrier_freq = DEFAULT_CARRIER_FREQ;
11         if (ioctl(fd1, LIRC_SET_SEND_CARRIER, &carrier_freq)) {
12                 fprintf(stderr,
13                         "lirc0: could not set carrier freq: %s\n",
14                         strerror(errno));
15                 goto OUT;
16         }
17
18         printf("irtest: send key code : 0x%x\n", key_code);
19
20         size = nec_ir_encode(tx_raw_buf, key_code);
21         /*dump the raw data*/
22         for (i = 0; i < size; i++) {
23                 printf("%d ", *(tx_raw_buf + i) & 0x00FFFFFF);
24                 if ((i + 1) % 8 == 0)
25                         printf("\n");
26         }
27         printf("\n");
28         if (argc > 4 && !strcmp(argv[3], "-n")) {
29                 cnt = atoi(argv[4]);
30                 for (i = 0; i < cnt; i++) {
31                         size_t = size * sizeof(uint32_t);
32                         ret = write(fd1, (char *)tx_raw_buf, size_t);
33                         if (ret > 0)
34                                 printf("irtest No.%d: send %d bytes ir raw data\n\n",
```

```
                                        i, ret);
                        usleep(100*1000);
                }
        } else {
                size_t = size * sizeof(uint32_t);
                ret = write(fd1, (char *)tx_raw_buf, size_t);
                if (ret > 0)
                        printf("irtest: send %d bytes ir raw data\n\n", ret);
        }

        }
```

gpio_ir_test 发送示例如下，其发送的字节为 0x04fb13ec，即一个完整的 4bytes 的 NEC code，代表 key code：0x13，在 gpio_ir_test 中会转换为具体的脉冲时间（us），跟接收的 IR RAW 数据处理方式一样，高 8bit 代表脉冲类型，低 24bit 为脉冲时间。

```
root@TinaLinux:/# gpio_ir_test tx 0x04fb13ec
lirc0 open succeed.
[  340.290466] GPIO IR tX: 68 raw samples
send key code : 0x4fb13ec
9000 4500 562 562 562 562 562 1687
562 562 562 562 562 562 562 562
562 562 562 1687 562 1687 562 562
562 1687 562 1687 562 1687 562 1687
562 1687 562 1687 562 1687 562 562
562 562 562 1687 562 562 562 562
562 562 562 562 562 562 562 1687
562 1687 562 562 562 1687 562 1687
562 1687 562 5625
send 272 bytes ir raw data
```

5.3.3 测试 IR LOOP 功能

gpio_ir_test loop 可以进行 IR 的自发自收，其发送的数据为 0x04fb13ec，100ms 发送一次，可以进行对红外收发压测：

```
......
    key_code = 0x04fb13ec;
        err = pthread_create(&tid, NULL, (void *)ir_recv_thread, NULL);
        if (err != 0) {
                printf("create pthread error: %d\n", __LINE__);
                goto OUT;
        }

        duty_cycle = DEFAULT_DUTY_CYCLE;
        if (ioctl(fd1, LIRC_SET_SEND_DUTY_CYCLE, &duty_cycle)) {
                fprintf(stderr,
                        "lirc0: could not set carrier duty: %s\n",
                        strerror(errno));
                goto OUT;
        }
        carrier_freq = DEFAULT_CARRIER_FREQ;
        if (ioctl(fd1, LIRC_SET_SEND_CARRIER, &carrier_freq)) {
                fprintf(stderr,
```

```
                        "lirc0: could not set carrier freq: %s\n",
                        strerror(errno));
                goto OUT;
        }

        /*echo 50ms transmit one frame */

        if (argc > 3 && !strcmp(argv[2], "-n")) {
                cnt = atoi(argv[3]);
                for (i = 0; i < cnt; i++) {
                        size = nec_ir_encode(tx_raw_buf, key_code);
                        size_t = size * sizeof(uint32_t);
                        printf("send key code : 0x%x, No.%d\n", key_code, i);
                        write(fd1, (char *)tx_raw_buf, size_t);
                        usleep(100*1000);
                        if (int_exit)
                                break;
        }
        } else {
                i = 0;
                do {
                        size = nec_ir_encode(tx_raw_buf, key_code);
                        size_t = size * sizeof(uint32_t);
                        printf("send key code : 0x%x, %d\n", key_code, i++);
                        write(fd1, (char *)tx_raw_buf, size_t);
                        usleep(100*1000);
                } while (!int_exit);
```

自发自收的运行效果如下所示：

```
root@TinaLinux:/# gpio_ir_test loop
send key code : 0x4fb13ec, 8 //ir tx
109860
110987 9582 4513 581 608 549 579 580
1681 586 603 578 579 549 579 580
580 549 580 609 1709 548 1678 581
612 547 1738 549 1678 579 1709 579
1743 548 1708 550 1739 548 1709 549
580 581 609 548 1741 548 580 548
614 546 608 549 610 548 579 579
1680 578 1711 577 579 549 1709 580
1712 578 1708 549    //ir rx data
--------------------
send key code : 0x4fb13ec, 9
110345
110927 9612 4514 549 610 579 580 579
1682 546 609 549 581 579 610 549
580 580 548 610 1709 549 1708 579
582 547 1741 549 1709 579 1708 581
1709 548 1710 580 1707 549 1709 579
551 578 610 548 1741 549 579 549
611 548 610 549 610 549 580 579
1680 609 1708 550 578 579 1709 548
1743 577 1678 549
```

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明


、 
、全志科技、  （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。