Linux UART
开发指南

版本号: 2.1
发布日期: 2020.12.24

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.6.29	AWA1440	添加初版
1.1	2020.11.10	AWA1440	update for linux-5.4
1.2	2020.11.16	AWA1636	增加设置 uart 波特率章节
2.0	2020.11.16	AWA1440	update setting baud rate for linux-5.4
2.1	2020.12.24	AWA1440	update for supporting s_uart for linux-4.9

目　录

插　图

1 概述

1.1 编写目的

介绍 Linux 内核中 UART 驱动的接口及使用方法，为 UART 设备的使用者提供参考。

1.2 适用范围

表 1-1: 适用产品列表

内核版本	驱动文件
Linux-4.9 及以上	sunxi-uart.c

1.3 相关人员

UART 驱动、及应用层的开发/维护人员。

2 模块介绍

2.1 模块功能介绍

Linux 内核中,UART 驱动的结构图 1 所示, 可以分为三个层次:

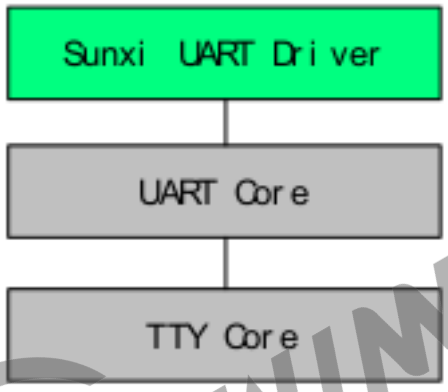

图 2-1：Linux UART 体系结构图

1. Sunxi UART Driver, 负责 SUNXI 平台 UART 控制器的初始化、数据通信等, 也是我们要实现的部分。
2. UART Core, 为 UART 驱动提供了一套 API, 完成设备和驱动的注册等。
3. TTY core, 实现了内核中所有 TTY 设备的注册和管理。

2.2 相关术语介绍

表 2-1：UART 模块相关术语介绍

术语	解释说明
Sunxi	指 Allwinner 的一系列 SoC 硬件平台
UART	Universal Asynchronous Receiver/Transmitter，通用异步收发传输器
Console	控制台，Linux 内核中用于输出调试信息的 TTY 设备
TTY	TeleType/TeleTypewriters 的一个老缩写，原来指的是电传打字机，现在泛指和计算机串行端口连接的终端设备。TTY 设备还包括虚拟控制台，串口以及伪终端设备

2.3 源码结构介绍

```
linux4.9
|-- drivers
|    |-- tty
|    |    |-- serial
|    |          |-- serial_core.c
|    |          |-- sunxi-uart.c
|    |          |-- sunxi-uart.h
```

3 模块配置介绍

3.1 kernel menuconfig 配置

在 longan 顶层目录，执行./build.sh menuconfig(需要先执行./build.sh config) 进入配置主界面，并按以下步骤操作：首先，选择 Device Drivers 选项进入下一级配置，如下图所示：

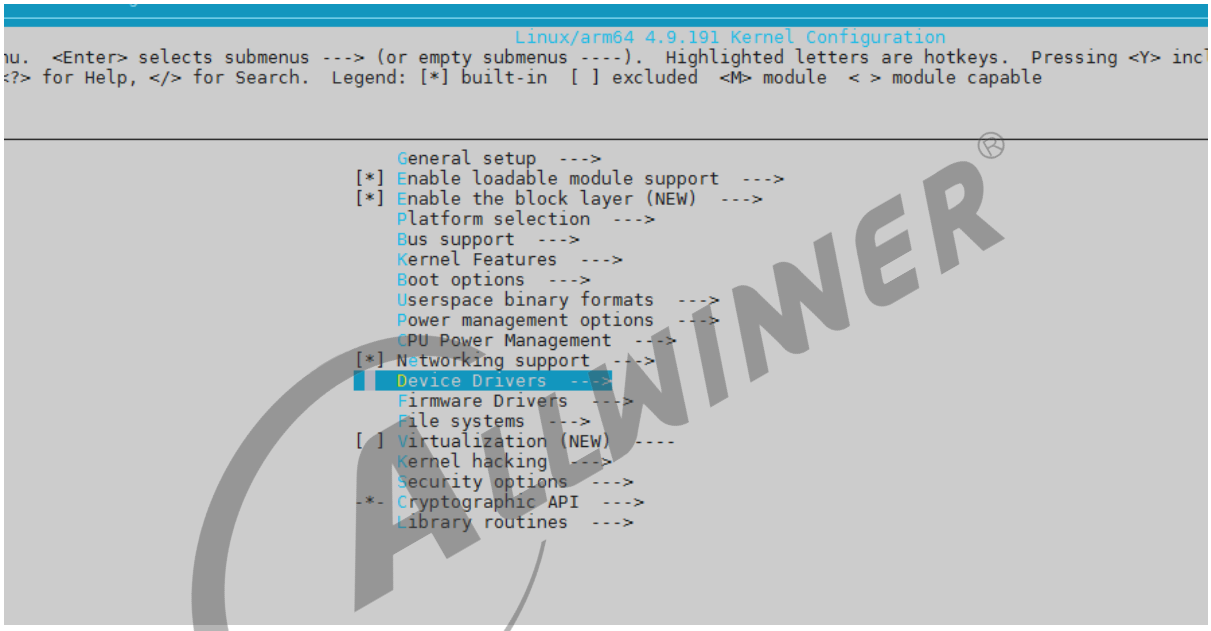

图 3-1: 内核 menuconfig 根菜单

选择 Character devices, 进入下级配置，如下图所示：

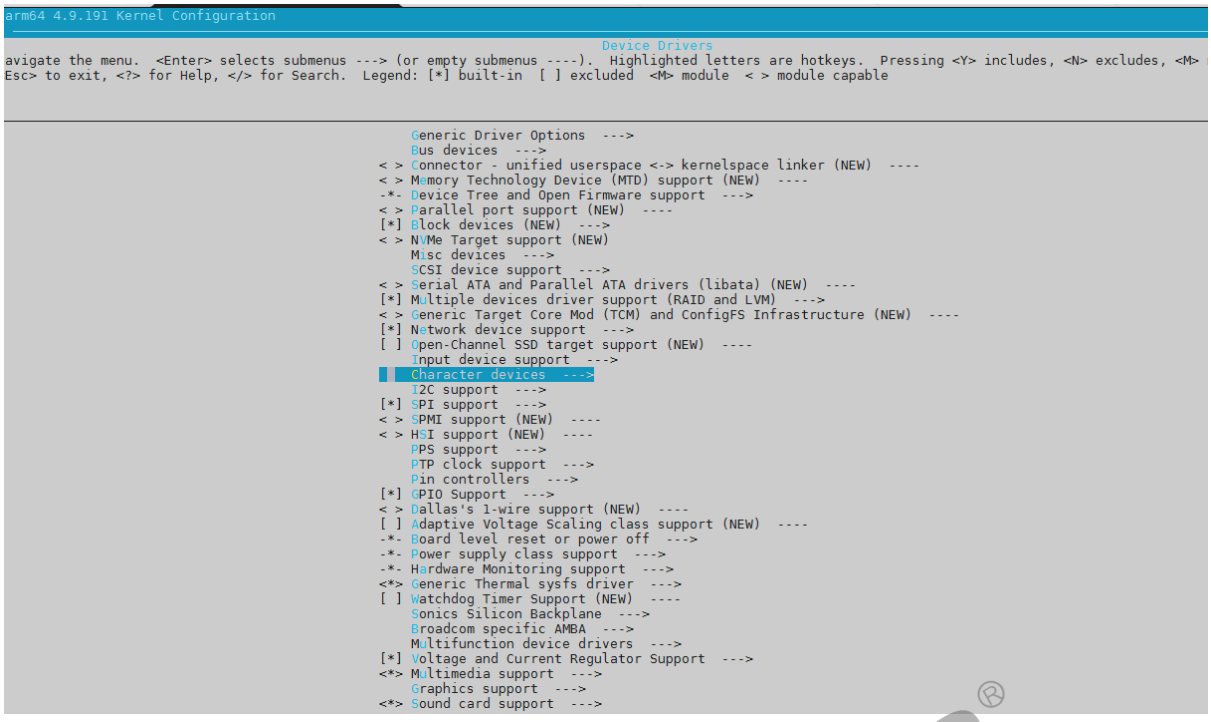

图 3-2: 内核 menuconfig device drivers 菜单

选择 Serial drivers, 进入下级配置，如下图所示：

图 3-3: 内核 menuconfig Character drivers 菜单

选择 SUNXI UART Controller 和 Console on SUNXI UART port 选项，如下图

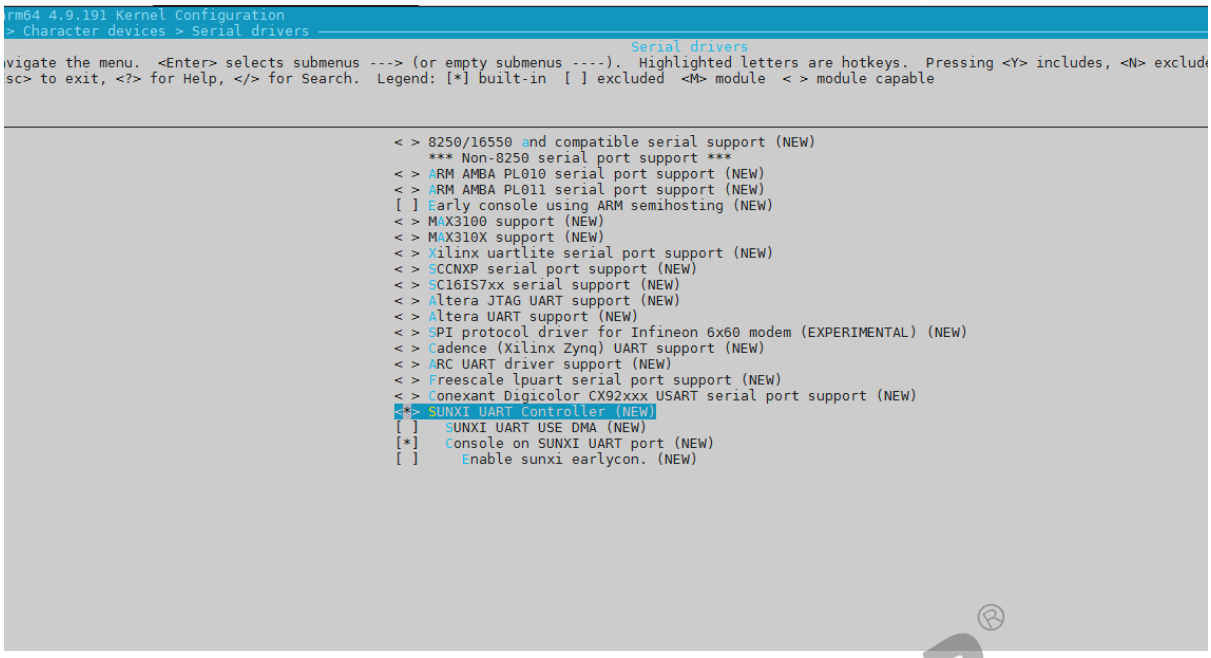

图 3-4: 内核 menuconfig sunxi uart 配置菜单

如果需要 UART 支持 DMA 传输，则可以打开 SUNXI UART USE DMA 选项。

3.2 device tree 源码结构和路径

- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM64 CPU 而言，设备树的路径为内核目录下：arch/arm64/boot/dts/sunxi/sun*.dtsi。
- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM32 CPU 而言，设备树的路径为内核目录下：arch/arm/boot/dts/sun*.dtsi
- 板级设备树 (board.dts) 路径：/device/config/chips/{IC}/configs/{BOARD}/board.dts

device tree 的源码结构关系如下：

```
linux-4.9
board.dts
  |--------sun*.dtsi
                |------sun*-pinctrl.dtsi
                |------sun*-clk.dtsi

linux-5.4
board.dts
  |-------sun*.dtsi
```

3.2.1 device tree 对 uart 控制器的通用配置

linux-4.9 的通用配置如下:

```
/ {
    model = "sun*";
    compatible = "arm,sun*";
    interrupt-parent = <&wakeupgen>;
    #address-cells = <2>;
    #size-cells = <2>;

    aliases {
        serial0 = &uart0;
        serial1 = &uart1;
        serial2 = &uart2;
        serial3 = &uart3;
        serial4 = &uart4;
        serial5 = &uart5;
        ...
    };

    uart0: uart@05000000 {
    compatible = "allwinner,sun50i-uart";    /* 用于驱动和设备绑定*/
    device_type = "uart0";           /* 设备类型*/
    reg = <0x0 0x05000000 0x0 0x400>;        /* 设备使用的寄存器基地址以及范围*/
    interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>;    /* 设备使用的硬件中断号*/
    clocks = <&clk_uart0>;                   /* 设备使用的时钟*/
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&uart0_pins_a>;             /* 设备正常状态下使用的pin脚*/
    pinctrl-1 = <&uart0_pins_b>;             /* 设备休眠状态下使用的pin脚*/
    uart0_port = <0>;                /* uart控制器对应的ttyS唯一端口号，不能与其他uart控制器重复*/
    uart0_type = <2>;                /* uart控制器线数，取值2/4/8*/
    use_dma = <0>;                   /* 是否采用DMA 方式传输，0：不启用，1：只启用TX，2：只启用RX，3：启用TX 与RX*/
    status = "okay";                 /* 是否使能该节点*/
};
```

linux-5.4 的通用配置如下:

```
    uart0: uart@5000000 {
        compatible = "allwinner,sun50i-uart";
        device_type = "uart0";
        reg = <0x0 0x05000000 0x0 0x400>;
        interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>;
        sunxi,uart-fifosize = <64>;
        clocks = <&ccu CLK_BUS_UART0>;  /* 设备使用的时钟 */
        clock-names = "uart0";
        resets = <&ccu RST_BUS_UART0>;  /* 设备reset时钟 */
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&uart0_pins_a>;
        pinctrl-1 = <&uart0_pins_b>;
        uart0_port = <0>;
        uart0_type = <2>;
        dmas = <&dma 14>;       /* 14表示DRQ */
        dma-names = "tx";
        use_dma = <0>;  /* 是否采用DMA 方式传输，0：不启用，1：只启用TX，2：只启用RX，3：启用TX 与RX */
```

```
18      };
```

在 Device Tree 中对每一个 UART 控制器进行配置，一个 UART 控制器对应一个 UART 节点，节点属性的含义见注释。为了在 UART 驱动代码中区分每一个 UART 控制器，需要在 Device Tree 中的 aliases 节点中未每一个 UART 节点指定别名，如上 aliases 节点所示。别名形式为字符串 “serial” 加连续编号的数字，在 UART 驱动程序中可以通过 of_alias_get_id() 函数获取对应的 UART 控制器的数字编号，从而区分每一个 UART 控制器。

3.2.2 board.dts 板级配置

board.dts 用于保存每个板级平台的设备信息 (如 demo 板、demo2.0 板等等)，board.dts 路径如下：

/device/config/chips/{IC}/configs/{BOARD}/board.dts

在 board.dts 中的配置信息如果在 *.dtsi(如 sun50iw9p1.dtsi 等) 存在，则会存在以下覆盖规则：

1. 相同属性和结点，board.dts 的配置信息会覆盖 *.dtsi 中的配置信息
2. 新增加的属性和结点，会添加到编译生成的 dtb 文件中

uart 在 board.dts 的简单配置如下：

```
soc@03000000 {
    ...
    &uart0 {
        uart-supply = <&reg_dcdc1>; /* IO使用的电 */
        status = "okay";
    };
    &uart1 {
        status = "okay";
    };
    ...
}
```

3.2.3 uart dma 模式配置

1. 在内核配置菜单打开 CONFIG_SERIAL_SUNXI_DMA 配置，如下图所示:

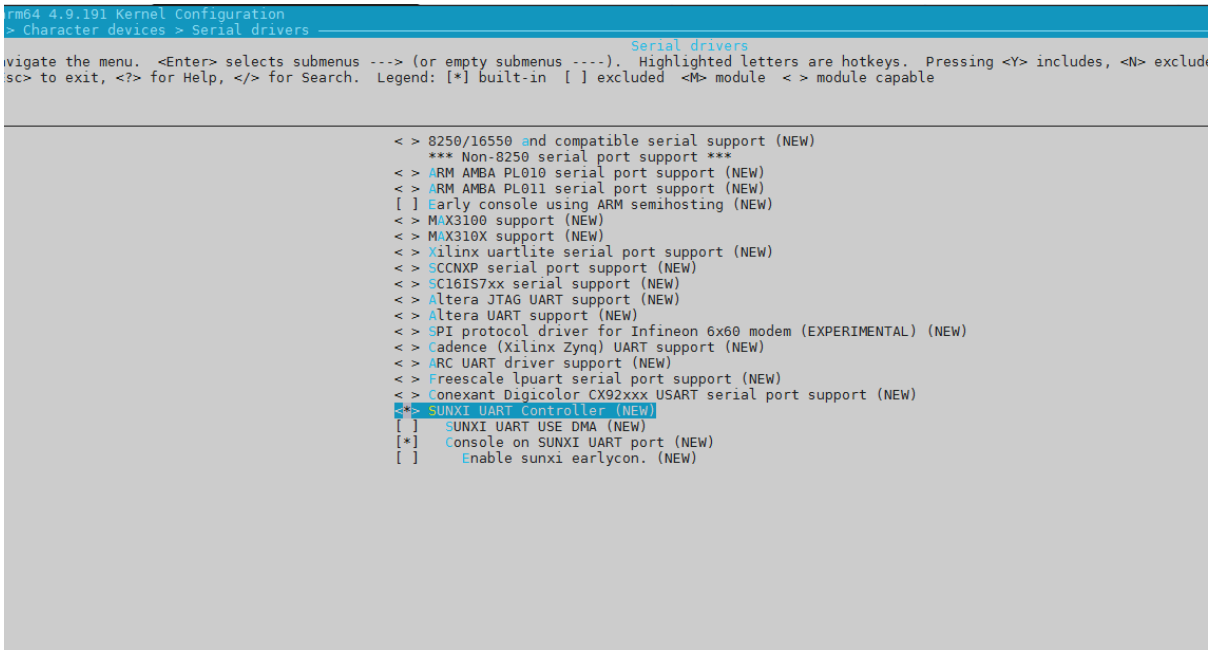

图 3-5: 内核 menuconfig sunxi uart 配置菜单

2. 在对应 dts 配置使用 dma，如下所示:

Linux-4.9 配置如下:

```
uart1: uart@05000400 {
  ...
  use_dma = <3>;  /* 是否采用DMA 方式传输，0: 不启用，1: 只启用TX，2: 只启用RX，3: 启用TX 与RX */
  status = "okay";
};
```

linux-5.4 配置如下:

```
    uart0: uart@5000000 {
        ...
        dmas = <&dma 14>;    /* 14表示DRQ, 查看dma spec */
        dma-names = "tx";
        use_dma = <0>;  /* 是否采用DMA 方式传输，0: 不启用，1: 只启用TX，2: 只启用RX，3: 启用TX 与RX
    */
    };
```

3.2.4 设置其他 uart 为打印 conole

1. 从 dts 确保设置为 console 的 uart 已经使能，如 uart1

```
uart1: uart@05000400 {
  compatible = "allwinner,sun50i-uart";
  device_type = "uart1";
```

```
  reg = <0x0 0x05000400 0x0 0x400>;
  interrupts = <GIC_SPI 1 IRQ_TYPE_LEVEL_HIGH>;
  clocks = <&clk_uart1>;
  pinctrl-names = "default", "sleep";
  pinctrl-0 = <&uart1_pins_a>;
  pinctrl-1 = <&uart1_pins_b>;
  uart1_port = <1>;
  uart1_type = <4>;
  status = "okay"; /* 确保该uart已经使能 */
};
```

2. 修改方案使用的 env*.cfg 文件，如下所示：

```
console=ttyS1,115200

说明:
ttyS0 <===> uart0
ttyS1 <===> uart1
...
```

3.2.5 设置 uart 波特率

在不同的 Sunxi 硬件平台中，UART 控制器的时钟源选择、配置略有不同，总体上的时钟关系如下：

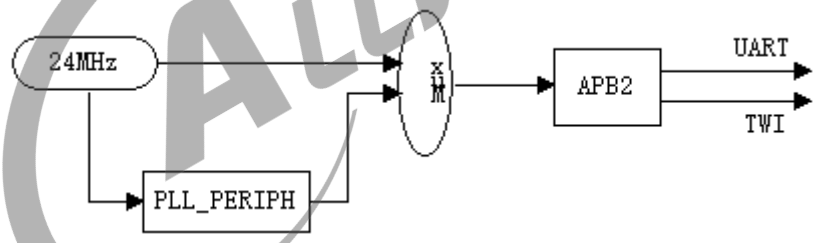

图 3-6: 时钟说明

UART 控制器会对输入的时钟源进行分频，最终输出一个频率满足（或近似）UART 波特率的时钟信号。UART 常用的标准波特率有：

```
#define  B4800  0000014
#define  B9600  0000015
#define  B19200 0000016
#define  B38400 0000017
#define  B57600 0010001
#define  B115200 0010002
#define  B230400 0010003
#define  B460800 0010004
#define  B500000 0010005
#define  B576000 0010006
#define  B921600 0010007
#define  B1000000 0010010
#define  B1152000 0010011
```

```
#define  B1500000 0010012
#define  B2000000 0010013
#define  B2500000 0010014
#define  B3000000 0010015
#define  B3500000 0010016
#define  B4000000 0010017
```

UART 时钟的分频比是 16 的整数倍，分频难免会有误差，所以输出 UART Device 通信的波特率是否足够精准，很大程度取决于输入的时钟源频率。考虑到功耗，UART 驱动中一般默认使用 24M 时钟源，但是根据应用场景我们有时候需要切换别的时钟源，基于两个原因：

1. 24MHz/16=1.5MHz，这个最大频率满足不了 1.5M 以上的波特率应用；
2. 24M 分频后得到波特率误差可能太大，也满足不了某些 UART 外设的冗余要求（一般要求 2% 或 5% 以内，由外设决定）。

UART 时钟源来自 APB2，APB2 的时钟源有两个，分别是 24MHz（HOSC）和 PLL_PERIPH（即驱动中的 PLL_PERIPH_CLK），系统默认配置 APB2 的时钟源是 24M，如果要提高 UART 的时钟就要将 APB2 的时钟源设置为 PLL_PERIPH。同时要注意到 APB2 也是 TWI 的时钟源，所以需要兼顾 TWI 时钟。

各个 uart 波特率对应频点关系如下：

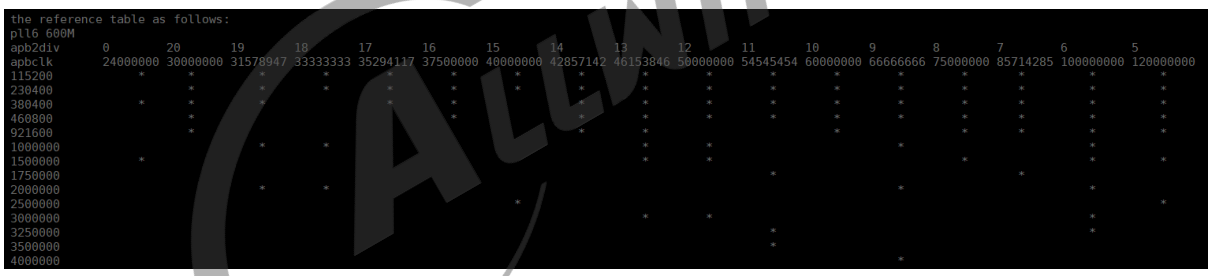

图 3-7: 波特率关系

例如需要配置 uart2 的波特率为 460800，在上述关系表中可以看出，对应的时钟为 30M、37.5M、42.857M、46.153M 和 50M 等，所以需要在设备树里修改 uart2 时钟：

linux-4.9 修改波特率如下：

```
            device_type = "uart2";
            reg = <0x0 0x05000800 0x0 0x400>;
            interrupts = <GIC_SPI 2 IRQ_TYPE_LEVEL_HIGH>;
-                       clocks = <&clk_uart2>;
+                       clocks = <&clk_uart2>, <&clk_apb2>, <&clk_psi>;
+                       clock-frequency = <50000000>;
            pinctrl-names = "default", "sleep";
```

有的情况下，APB2 不一定能准确分出 30M 或者 37.5M 等时钟，所以这里我们选择配置为 50M 时钟。

如果我们修改了 APB2 时钟，可能会对 uart0 有影响，启动串口会出现乱码，那么我们最好也将 uart0 的时钟配置为 50M 时钟。

```
            device_type = "uart0";
            reg = <0x0 0x05000000 0x0 0x400>;
            interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>;
-                       clocks = <&clk_uart0>;
+                       clocks = <&clk_uart0>, <&clk_apb2>, <&clk_psi>;
+                       clock-frequency = <50000000>;
            pinctrl-names = "default", "sleep";
```

linux-5.4 修改波特率如下:

```
            device_type = "uart2";
            reg = <0x0 0x05000800 0x0 0x400>;
            interrupts = <GIC_SPI 2 IRQ_TYPE_LEVEL_HIGH>;
-                       clocks = <&clk_uart2>;
+                       clocks = <&ccu CLK_BUS_UART0>,
+                                <&ccu CLK_APB2>,
+                                <&ccu CLK_PSI_AHB1_AHB2>;
+                       clock-frequency = <50000000>;
            pinctrl-names = "default", "sleep";
```

说明

修改 *APB2* 总线时钟，会影响到使用到 *APB2* 总线时钟的相应模块。

3.2.6 支持 cpus 域的 uart

在不同的 Sunxi 硬件平台中，UART 控制器根据电源域划分了 CPUX 域和 CPUS 域，系统默认对 CPUX 域的 uart 控制器都会默认配置上，但对 CPUS 域的 uart 控制器可能没有支持上，当希望通过 CPUX 使用 CPUS 域的 uart 时，请参考以下步骤进行支持:

1. 通过 datasheet 或者 spec，获取 CPUS 域 uart 控制器的 pin 脚、clk、控制器地址等信息。
2. 确保 cpus 运行的系统没有使用到 CPUS 域的 uart 控制器，只有 CPUX 域在使用
3. 在 r_pio 域配置 pin 的 dtsi 文件 (*.-pinctrl.dtsi)

```
/ {
    soc@03000000{
        r_pio: pinctrl@07022000 {
            ...
            /* 配置CPUS域uart pin信息 */
            s_uart0_pins_a: s_uart0@0 {
                allwinner,pins = "PL2", "PL3";
                allwinner,function = "s_uart0";
                allwinner,muxsel = <2>;
                allwinner,drive = <1>;
                allwinner,pull = <1>;
            };
            ...
```

```
        };
    ...
};
```

4. 在 clk dtsi 配置时钟信息 (*-clk.dtsi)

```
clk_suart0: suart0 {
    #clock-cells = <0>;
    compatible = "allwinner,periph-cpus-clock";
    clock-output-names = "suart0";
};
```

5. 在 dtsi 配置 uart 控制器信息 (*.dtsi)

```
aliases {
    serial0 = &uart0;
    ...
    serial5 = &uart5;   //添加uart5的别名，必须要添加
};

uart0:uart@05000000 {
};
....
/* 添加uart控制器信息 ，必须要添加，具体属性含义，见上文 */
uart5: uart@07080000 {
    compatible = "allwinner,sun8i-uart";
    device_type = "uart5";
    reg = <0x0 0x07080000 0x0 0x400>;
    interrupts = <GIC_SPI 111 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clk_suart0>;
    pinctrl-names = "default", "sleep";
    pinctrl-1 = <&s_uart0_pins_a>;
    uart5_port = <5>;
    uart5_type = <2>;
    status = "okay";
};
```

6. 检查或添加 pin 描述符

有可能 pinctrl 驱动里，没有把 CPUS 域 uart 控制器的 pin 描述出来，因此要的 pinctrl 驱动里检查或添加 pin 的描述信息，查看 pinctrl-*-r.c 文件。

```
static const struct sunxi_desc_pin *_r_pins[] = {
    SUNXI_PIN(SUNXI_PINCTRL_PIN(L, 0),
            ...
    SUNXI_PIN(SUNXI_PINCTRL_PIN(L, 2),
        ...
        SUNXI_FUNCTION(0x2, "s_uart0"), //没有则要添加
        ...
    SUNXI_PIN(SUNXI_PINCTRL_PIN(L, 3),
        ...
```

```
        SUNXI_FUNCTION(0x2, "s_uart0"), //没有则要添加
        ...
};
```

7. 检查 clk 的描述信息

有可能 clk 驱动根据情况，也没有把 CPUS 域的 uart 控制器的时钟信息描述出来，因此要 clk 驱动里检查或添加 clk 的描述信息，查看 clk-*.c* 和 *clk*-.h 文件

```
/* clk-*.h */
#define CPUS_UART_GATE      0x018C

/* clk-*.c */

static const char *suart_parents[] = {"cpurapbs2"};

SUNXI_CLK_PERIPH(suart0,         0,               0,       0,            0,
      0,         0,          0,          0,          0,          0,
    CPUS_UART_GATE,  CPUS_UART_GATE, 0,             0,          16,         0,
    0,             &clk_lock, NULL,             0);

truct periph_init_data sunxi_periphs_cpus_init[] = {
    ...
    {"suart0",          0,                                  suart_parents,
    ARRAY_SIZE(suart_parents),          &sunxi_clk_periph_suart0          },
};
```

8. 检查 uart 驱动支持的 uart 数量是否足够，查看 sunxi-uart.h 文件

通过 SUNXI_UART_NUM 宏确认支持 uart 的数量。

9. 修改完毕后，启动小机，查看 ttySn 设备是否生成，通过该设备测试 uart 是否正常使用即可。

4 接口描述

UART 驱动会注册生成串口设备/dev/ttySx，应用层的使用只需遵循 Linux 系统中的标准串口编程方法即可。

4.1 打开/关闭串口

使用标准的文件打开函数：

```
int open(const char *pathname, int flags);
int close(int fd);
```

需要引用头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

4.2 读/写串口

同样使用标准的文件读写函数：

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

需要引用头文件：

```
#include <unistd.h>
```

4.3 设置串口属性

串口属性包括波特率、数据位、停止位、校验位、流控等，这部分是串口设备特有的接口。串口属性的数据结构 termios 定义如下：（terminos.h）

```
#define NCCS 19
struct termios {
    tcflag_t c_iflag;       /* input mode flags */
    tcflag_t c_oflag;       /* output mode flags */
    tcflag_t c_cflag;       /* control mode flags */
    tcflag_t c_lflag;       /* local mode flags */
    cc_t c_line;            /* line discipline */
    cc_t c_cc[NCCS];        /* control characters */
};
```

其中，c_iflag 的标志常量定义如下：

标志	说明
IGNBRK	忽略输入中的 BREAK 状态。
BRKINT	如果设置了 IGNBRK，将忽略 BREAK。如果没有设置，但是设置了 BRKINT，那么 BREAK 将使得输入和输出队列被刷新，如果终端是一个前台进程组的控制终端，这个进程组中所有进程将收到 SIGINT 信号。如果既未设置 IGNBRK 也未设置 BRKINT，BREAK 将视为与 NUL 字符同义，除非设置了 PARMRK，这种情况下它被视为序列 \377 \0 \0。
IGNPAR	忽略桢错误和奇偶校验错。
PARMRK	如果没有设置 IGNPAR，在有奇偶校验错或桢错误的字符前插入 \377 \0。如果既没有设置 IGNPAR 也没有设置 PARMRK，将有奇偶校验错或桢错误的字符视为 \0。
INPCK	启用输入奇偶检测。
ISTRIP	去掉第八位。
INLCR	将输入中的 NL 翻译为 CR。
IGNCR	忽略输入中的回车。
ICRNL	将输入中的回车翻译为新行 (除非设置了 IGNCR)。
IUCLC	(不属于 POSIX) 将输入中的大写字母映射为小写字母。
IXON	启用输出的 XON/XOFF 流控制。
IXANY	(不属于 POSIX.1；XSI) 允许任何字符来重新开始输出。
IXOFF	启用输入的 XON/XOFF 流控制。
IMAXBEL	(不属于 POSIX) 当输入队列满时响零。Linux 没有实现这一位，总是将它视为已设置。

c_oflag 的标志常量定义如下：

标志	说明
OLCUC	(不属于 POSIX) 将输出中的小写字母映射为大写字母。
ONLCR	(XSI) 将输出中的新行符映射为回车-换行。
OCRNL	将输出中的回车映射为新行符
ONOCR	不在第 0 列输出回车。
ONLRET	不输出回车。
OFILL	发送填充字符作为延时，而不是使用定时来延时。

标志	说明
OFDEL	(不属于 POSIX) 填充字符是 ASCII DEL (0177)。如果不设置，填充字符则是 ASCII NUL。
NLDLY	新行延时掩码。取值为 NL0 和 NL1。
CRDLY	回车延时掩码。取值为 CR0, CR1, CR2, 或 CR3。
TABDLY	水平跳格延时掩码。取值为 TAB0, TAB1, TAB2, TAB3 (或 XTABS)。取值为 TAB3，即 XTABS，将扩展跳格为空格 (每个跳格符填充 8 个空格)。
BSDLY	回退延时掩码。取值为 BS0 或 BS1。(从来没有被实现过)
VTDLY	竖直跳格延时掩码。取值为 VT0 或 VT1。
FFDLY	进表延时掩码。取值为 FF0 或 FF1。

c_cflag 的标志常量定义如下：

标志	说明
CBAUD	(不属于 POSIX) 波特率掩码 (4+1 位)。
CBAUDEX	(不属于 POSIX) 扩展的波特率掩码 (1 位)，包含在 CBAUD 中。(POSIX 规定波特率存储在 termios 结构中，并未精确指定它的位置，而是提供了函数 cfgetispeed() 和 cfsetispeed() 来存取它。一些系统使用 c_cflag 中 CBAUD 选择的位，其他系统使用单独的变量，例如 sg_ispeed 和 sg_ospeed 。)
CSIZE	字符长度掩码。取值为 CS5, CS6, CS7, 或 CS8。
CSTOPB	设置两个停止位，而不是一个。
CREAD	打开接受者。
PARENB	允许输出产生奇偶信息以及输入的奇偶校验。
PARODD	输入和输出是奇校验。
HUPCL	在最后一个进程关闭设备后，降低 modem 控制线 (挂断)。
CLOCAL	忽略 modem 控制线。
LOBLK	(不属于 POSIX) 从非当前 shell 层阻塞输出 (用于 shl)。
CIBAUD	(不属于 POSIX) 输入速度的掩码。CIBAUD 各位的值与 CBAUD 各位相同，左移了 IBSHIFT 位。
CRTSCTS	(不属于 POSIX) 启用 RTS/CTS (硬件) 流控制。

c_lflag 的标志常量定义如下：

标志	说明
ISIG	当接受到字符 INTR, QUIT, SUSP, 或 DSUSP 时，产生相应的信号。
ICANON	启用标准模式 (canonical mode)。允许使用特殊字符 EOF, EOL, EOL2, ERASE, KILL, LNEXT, REPRINT, STATUS, 和 WERASE，以及按行的缓冲。

标志	说明
XCASE	(不属于 POSIX; Linux 下不被支持) 如果同时设置了 ICANON，终端只有大写。输入被转换为小写，除了以 前缀的字符。输出时，大写字符被前缀，小写字符被转换成大写。
ECHO	回显输入字符。
ECHOE	如果同时设置了 ICANON，字符 ERASE 擦除前一个输入字符，WERASE 擦除前一个词。
ECHOK	如果同时设置了 ICANON，字符 KILL 删除当前行。
ECHONL	如果同时设置了 ICANON，回显字符 NL，即使没有设置 ECHO。
ECHOCTL	(不属于 POSIX) 如果同时设置了 ECHO，除了 TAB, NL, START, 和 STOP 之外的 ASCII 控制信号被回显为 ^X, 这里 X 是比控制信号大 0x40 的 ASCII 码。例如，字符 0x08 (BS) 被回显为 ^H。
ECHOPRT	(不属于 POSIX) 如果同时设置了 ICANON 和 IECHO，字符在删除的同时被打印。
ECHOKE	(不属于 POSIX) 如果同时设置了 ICANON，回显 KILL 时将删除一行中的每个字符，如同指定了 ECHOE 和 ECHOPRT 一样。
DEFECHO	(不属于 POSIX) 只在一个进程读的时候回显。
FLUSHO	(不属于 POSIX; Linux 下不被支持) 输出被刷新。这个标志可以通过键入字符 DISCARD 来开关。
NOFLSH	禁止在产生 SIGINT, SIGQUIT 和 SIGSUSP 信号时刷新输入和输出队列。
PENDIN	(不属于 POSIX; Linux 下不被支持) 在读入下一个字符时，输入队列中所有字符被重新输出。(bash 用它来处理 typeahead)
TOSTOP	向试图写控制终端的后台进程组发送 SIGTTOU 信号。
IEXTEN	启用实现自定义的输入处理。这个标志必须与 ICANON 同时使用，才能解释特殊字符 EOL2，LNEXT，REPRINT 和 WERASE，IUCLC 标志才有效。

c_cc 数组定义了特殊的控制字符。符号下标 (初始值) 和意义为：

标志	说明
VINTR	(003, ETX, Ctrl-C, or also 0177, DEL, rubout) 中断字符。发出 SIGINT 信号。当设置 ISIG 时可被识别，不再作为输入传递。
VQUIT	(034, FS, Ctrl-) 退出字符。发出 SIGQUIT 信号。当设置 ISIG 时可被识别，不再作为输入传递。
VERASE	(0177, DEL, rubout, or 010, BS, Ctrl-H, or also #) 删除字符。删除上一个还没有删掉的字符，但不删除上一个 EOF 或行首。当设置 ICANON 时可被识别，不再作为输入传递。
VKILL	(025, NAK, Ctrl-U, or Ctrl-X, or also @) 终止字符。删除自上一个 EOF 或行首以来的输入。当设置 ICANON 时可被识别，不再作为输入传递。

标志	说明
VEOF	(004, EOT, Ctrl-D) 文件尾字符。更精确地说，这个字符使得 tty 缓冲中的内容被送到等待输入的用户程序中，而不必等到 EOL。如果它是一行的第一个字符，那么用户程序的 read() 将返回 0，指示读到了 EOF。当设置 ICANON 时可被识别，不再作为输入传递。
VMIN	非 canonical 模式读的最小字符数。
VEOL	(0, NUL) 附加的行尾字符。当设置 ICANON 时可被识别。
VTIME	非 canonical 模式读时的延时，以十分之一秒为单位。
VEOL2	(not in POSIX; 0, NUL) 另一个行尾字符。当设置 ICANON 时可被识别。
VSTART	(021, DC1, Ctrl-Q) 开始字符。重新开始被 Stop 字符中止的输出。当设置 IXON 时可被识别，不再作为输入传递。
VSTOP	(023, DC3, Ctrl-S) 停止字符。停止输出，直到键入 Start 字符。当设置 IXON 时可被识别，不再作为输入传递。
VSUSP	(032, SUB, Ctrl-Z) 挂起字符。发送 SIGTSTP 信号。当设置 ISIG 时可被识别，不再作为输入传递。
VLNEXT	(not in POSIX; 026, SYN, Ctrl-V) 字面上的下一个。引用下一个输入字符，取消它的任何特殊含义。当设置 IEXTEN 时可被识别，不再作为输入传递。
VWERASE	(not in POSIX; 027, ETB, Ctrl-W) 删除词。当设置 ICANON 和 IEXTEN 时可被识别，不再作为输入传递。
VREPRINT	(not in POSIX; 022, DC2, Ctrl-R) 重新输出未读的字符。当设置 ICANON 和 IEXTEN 时可被识别，不再作为输入传递。

4.3.1 tcgetattr

- 作用：获取串口设备的属性。
- 参数：
  - fd，串口设备的文件描述符。
  - termios_p，用于保存串口属性。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码。

4.3.2 tcsetattr

- 作用：设置串口设备的属性。
- 参数：
  - fd，串口设备的文件描述符。
  - optional_actions，本次设置什么时候生效。

- termios_p，指向要设置的属性结构。

- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码

说明

其中，*optional_actions* 的取值有：
***TCSANOW*：会立即生效。**
***TCSADRAIN*：当前的输出数据完成传输后生效，适用于修改了输出相关的参数。**
***TCSAFLUSH*：当前的输出数据完成传输，如果输入有数据可读但没有读就会被丢弃。**

4.3.3 cfgetispeed

- 作用：返回串口属性中的输入波特率。
- 参数：
  - termios_p，指向保存有串口属性的结构。
- 返回：
  - 成功，返回波特率，取值是一组宏，定义在 terminos.h。
  - 失败，返回-1，errnor 给出具体错误码。

说明

波特率定义如下所示：

```
#define B0 0000000
#define B50 0000001
#define B75 0000002
#define B110 0000003
#define B134 0000004
#define B150 0000005
#define B200 0000006
#define B300 0000007
#define B600 0000010
#define B1200 0000011
#define B1800 0000012
#define B2400 0000013
#define B4800 0000014
#define B9600 0000015
#define B19200 0000016
#define B38400 0000017
#define B57600 0010001
#define B115200 0010002
#define B230400 0010003
#define B460800 0010004
#define B500000 0010005
#define B576000 0010006
#define B921600 0010007
#define B1000000 0010010
#define B1152000 0010011
#define B1500000 0010012
#define B2000000 0010013
#define B2500000 0010014
#define B3000000 0010015
#define B3500000 0010016
#define B4000000 0010017
```

4.3.4 cfgetospeed

- 作用：返回串口属性中的输出波特率。
- 参数：
  - termios_p，指向保存有串口属性的结构。
- 返回：
  - 成功，返回波特率，取值是一组宏，定义在 terminos.h，见 4.3.3
  - 失败，返回-1，errnor 给出具体错误码。

4.3.5 cfsetispeed

- 作用：设置输入波特率到属性结构中。
- 参数：
  - termios_p，指向保存有串口属性的结构。

- speed，波特率，取值同 4.3.3。

- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码。

4.3.6 cfsetospeed

- 作用：设置输出波特率到属性结构中。
- 参数：
  - termios_p，指向保存有串口属性的结构。
  - speed，波特率，取值同 4.3.3。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码

4.3.7 cfsetspeed

- 作用：同时设置输入和输出波特率到属性结构中。
- 参数：
  - termios_p，指向保存有串口属性的结构。
  - speed，波特率，取值同 4.3.3。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码

4.3.8 tcflush

- 作用：清空输出缓冲区、或输入缓冲区的数据，具体取决于参数 queue_selector。
- 参数：
  - fd，串口设备的文件描述符。
  - queue_selector，清空数据的操作。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，errnor 给出具体错误码。

说明

参数 *queue_selector* 的取值有三个：
***TCIFLUSH*：清空输入缓冲区的数据。**
***TCOFLUSH*：清空输出缓冲区的数据。**
***TCIOFLUSH*：同时清空输入/输出缓冲区的数据。**

5 模块使用范例

此 demo 程序是打开一个串口设备，然后侦听这个设备，如果有数据可读就读出来并打印。设备名称、侦听的循环次数都可以由参数指定

```
#include     <stdio.h>      /*标准输入输出定义*/
#include     <stdlib.h>     /*标准函数库定义*/
#include     <unistd.h>     /*Unix 标准函数定义*/
#include     <sys/types.h>
#include     <sys/stat.h>
#include     <fcntl.h>      /*文件控制定义*/
#include     <termios.h>    /*PPSIX 终端控制定义*/
#include     <errno.h>      /*错误号定义*/
#include <string.h>

enum parameter_type {
    PT_PROGRAM_NAME = 0,
    PT_DEV_NAME,
    PT_CYCLE,

    PT_NUM
};

#define DBG(string, args...) \
    do { \
        printf("%s, %s()%u---", __FILE__, __FUNCTION__, __LINE__); \
        printf(string, ##args); \
        printf("\n"); \
    } while (0)

void usage(void)
{
    printf("You should input as: \n");
    printf("\t select_test [/dev/name] [Cycle Cnt]\n");
}

int OpenDev(char *name)
{
    int fd = open(name, O_RDWR );         //| O_NOCTTY | O_NDELAY
    if (-1 == fd)
        DBG("Can't Open(%s)!", name);

    return fd;
}

/**
*@brief  设置串口通信速率
*@param  fd     类型 int  打开串口的文件句柄
*@param  speed  类型 int  串口速度
*@return  void
*/
void set_speed(int fd, int speed){
```

```
    int   i;
    int   status;
    struct termios  Opt = {0};
    int speed_arr[] = { B38400, B19200, B9600, B4800, B2400, B1200, B300,
              B38400, B19200, B9600, B4800, B2400, B1200, B300, };
    int name_arr[] = {38400,  19200,  9600,  4800,  2400,  1200,  300, 38400,
              19200,  9600, 4800, 2400, 1200,  300, };

    tcgetattr(fd, &Opt);

    for ( i= 0;  i < sizeof(speed_arr) / sizeof(int);  i++) {
        if  (speed == name_arr[i])
            break;
    }

    tcflush(fd, TCIOFLUSH);
    cfsetispeed(&Opt, speed_arr[i]);
    cfsetospeed(&Opt, speed_arr[i]);

    Opt.c_lflag  &= ~(ICANON | ECHO | ECHOE | ISIG);  /*Input*/
    Opt.c_oflag  &= ~OPOST;   /*Output*/

    status = tcsetattr(fd, TCSANOW, &Opt);
    if  (status != 0) {
        DBG("tcsetattr fd");
        return;
    }
    tcflush(fd, TCIOFLUSH);
}

/**
*@brief   设置串口数据位，停止位和效验位
*@param  fd     类型  int  打开的串口文件句柄
*@param  databits 类型  int 数据位   取值 为 7 或者8
*@param  stopbits 类型  int 停止位   取值为 1 或者2
*@param  parity  类型  int  效验类型 取值为N,E,O,,S
*/
int set_Parity(int fd,int databits,int stopbits,int parity)
{
    struct termios options;

    if  ( tcgetattr( fd,&options)  !=  0) {
        perror("SetupSerial 1");
        return -1;
    }
    options.c_cflag &= ~CSIZE;

    switch (databits) /*设置数据位数*/
    {
    case 7:
        options.c_cflag |= CS7;
        break;
    case 8:
        options.c_cflag |= CS8;
        break;
    default:
        fprintf(stderr,"Unsupported data size\n");
        return -1;
    }

```

```
    switch (parity)
    {
    case 'n':
    case 'N':
        options.c_cflag &= ~PARENB;   /* Clear parity enable */
        options.c_iflag &= ~INPCK;     /* Enable parity checking */
        break;
    case 'o':
    case 'O':
        options.c_cflag |= (PARODD | PARENB); /* 设置为奇效验*/
        options.c_iflag |= INPCK;             /* Disnable parity checking */
        break;
    case 'e':
    case 'E':
        options.c_cflag |= PARENB;     /* Enable parity */
        options.c_cflag &= ~PARODD;   /* 转换为偶效验*/
        options.c_iflag |= INPCK;       /* Disnable parity checking */
        break;
    case 'S':
    case 's':  /*as no parity*/
        options.c_cflag &= ~PARENB;
        options.c_cflag &= ~CSTOPB;break;
    default:
        fprintf(stderr,"Unsupported parity\n");
        return -1;
    }

    /* 设置停止位*/
    switch (stopbits)
    {
        case 1:
            options.c_cflag &= ~CSTOPB;
            break;
        case 2:
            options.c_cflag |= CSTOPB;
           break;
        default:
             fprintf(stderr,"Unsupported stop bits\n");
             return -1;
    }

    /* Set input parity option */
    if (parity != 'n')
        options.c_iflag |= INPCK;
    tcflush(fd,TCIFLUSH);
    options.c_cc[VTIME] = 150; /* 设置超时15 seconds*/
    options.c_cc[VMIN] = 0; /* Update the options and do it NOW */
    if (tcsetattr(fd,TCSANOW,&options) != 0)
    {
        perror("SetupSerial 3");
        return -1;
    }
    return 0;
}

void str_print(char *buf, int len)
{
    int i;

    for (i=0; i<len; i++) {
```

```
        if (i%10 == 0)
            printf("\n");

        printf("0x%02x ", buf[i]);
    }
    printf("\n");
}

int main(int argc, char **argv)
{
    int i = 0;
    int fd = 0;
    int cnt = 0;
    char buf[256];

    int ret;
    fd_set rd_fdset;
    struct timeval dly_tm;      // delay time in select()

    if (argc != PT_NUM) {
        usage();
        return -1;
    }

    sscanf(argv[PT_CYCLE], "%d", &cnt);
    if (cnt == 0)
        cnt = 0xFFFF;

    fd = OpenDev(argv[PT_DEV_NAME]);
    if (fd < 0)
        return -1;

    set_speed(fd,19200);
    if (set_Parity(fd,8,1,'N') == -1)  {
        printf("Set Parity Error\n");
        exit (0);
    }

    printf("Select(%s), Cnt %d. \n", argv[PT_DEV_NAME], cnt);
    while (i<cnt) {
        FD_ZERO(&rd_fdset);
        FD_SET(fd, &rd_fdset);

        dly_tm.tv_sec  = 5;
        dly_tm.tv_usec = 0;
        memset(buf, 0, 256);

        ret = select(fd+1, &rd_fdset, NULL, NULL, &dly_tm);
//      DBG("select() return %d, fd = %d", ret, fd);
        if (ret == 0)
            continue;

        if (ret < 0) {
            printf("select(%s) return %d. [%d]: %s \n", argv[PT_DEV_NAME], ret, errno,
    strerror(errno));
            continue;
        }

        i++;
        ret = read(fd, buf, 256);
```

```
227         printf("Cnt%d: read(%s) return %d.\n", i, argv[PT_DEV_NAME], ret);
228         str_print(buf, ret);
229     }
230
231     close(fd);
232     return 0;
233 }
```

6 FAQ

6.1 UART 调试打印开关

6.1.1 通过 debugfs 使用命令打开调试开关

注：内核需打开 CONFIG_DYNAMIC_DEBUG 宏定义

```
1.挂载debugfs
mount -t debugfs none /sys/kernel/debug
2.打开uart模块所有打印
echo "module sunxi_uart +p" > /mnt/dynamic_debug/control
3.打开指定文件的所有打印
echo "file sunxi-uart.c +p" > /mnt/dynamic_debug/control
4.打开指定文件指定行的打印
echo "file sunxi-uart.c line 615 +p" > /mnt/dynamic_debug/control
5.打开指定函数名的打印
echo "func sw_uart_set_termios +p" > /mnt/dynamic_debug/control
6.关闭打印
把上面相应命令中的+p 修改为-p 即可
更多信息可参考linux 内核文档：linux-3.10/Documentation/dynamic-debug-howto.txt
```

6.1.2 代码中打开调试开关

```
1.定义CONFIG_SERIAL_DEBUG宏
linux-4.9 内核版本中默认没有定义CONFIG_SERIAL_DEBUG ， 需要自行在
drivers/tty/serial/Kconfig 中添加CONFIG_SERIAL_DEBUG 定义，然后在drivers/tty/serial/Makefile 文
件中添加代码ccflags-$(CONFIG_SERIAL_DEBUG) := -DDEBUG
注：使用该宏，需要内核关闭CONFIG_DYNAMIC_DEBUG宏
```

6.1.3 sysfs 调试接口

UART 驱动通过 sysfs 节点提供了几个在线调试的接口。

```
1./sys/devices/platform/soc/uart0/dev_info

cupid-p2:/ # cat /sys/devices/platform/soc/uart0/dev_info
id     = 0
name   = uart0
irq    = 247
```

```
io_num = 2
port->mapbase = 0x0000000005000000
port->membase = 0xffffff800b005000
port->iobase  = 0x00000000
pdata->regulator    = 0x          (null)
pdata->regulator_id =
```

从该节点可以看到uart端口的一些硬件资源信息

```
2./sys/devices/platform/soc/uart0/ctrl_info
cupid-p2:/ # cat /sys/devices/platform/soc/uart0/ctrl_info
 ier  : 0x05
 lcr  : 0x13
 mcr  : 0x03
 fcr  : 0xb1
 dll  : 0x0d
 dlh  : 0x00
 last baud : 115384 (dl = 13)

TxRx Statistics:
 tx     : 61123
 rx     : 351
 parity : 0
 frame  : 0
 overrun: 0
```

此节点可以打印出软件中保存的一些控制信息，如当前UART 端口的寄存器值、收发数据的统计等

```
3./sys/devices/platform/soc/uart0/status
cupid-p2:/ # cat /sys/devices/platform/soc/uart0/status
uartclk = 24000000
The Uart controller register[Base: 0xffffff800b005000]:
[RTX] 0x00 = 0x0000000d, [IER] 0x04 = 0x00000005, [FCR] 0x08 = 0x000000c1
[LCR] 0x0c = 0x00000013, [MCR] 0x10 = 0x00000003, [LSR] 0x14 = 0x00000060
[MSR] 0x18 = 0x00000000, [SCH] 0x1c = 0x00000000, [USR] 0x7c = 0x00000006
[TFL] 0x80 = 0x00000000, [RFL] 0x84 = 0x00000000, [HALT] 0xa4 = 0x00000002
```

此节点可以打印出当前UART 端口的一些运行状态信息，包括控制器的各寄存器值

著作权声明

商标声明

免责声明