



# Linux UART 开发指南

**版本号: 1.3**

**发布日期: 2024.3.14**

## 版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.08.04	XAA0249	初始版本
1.1	2022.11.03	AWA1979	更新文档模板
1.2	2024.1.30	XAA0249	补充波特率配置说明
1.3	2024.3.14	XAA0312	更新文档内容

# 目 录

<b>1 前言</b>	<b>1</b>
1.1 文档简介	1
1.2 目标读者	1
1.3 适用范围	1
1.4 相关术语介绍	1
1.4.1 硬件术语	1
<b>2 模块介绍</b>	<b>3</b>
2.1 模块功能介绍	3
2.2 模块流程介绍	3
2.3 模块配置介绍	5
2.3.1 设备树配置说明	5
2.3.1.1 soc 设备树配置	5
2.3.1.2 板级设备树配置	6
2.3.1.3 dma 模式配置	6
2.3.2 menuconfig 配置说明	7
2.4 源码结构介绍	11
2.5 驱动框架介绍	11
<b>3 模块接口说明</b>	<b>13</b>
3.1 打开/关闭串口	13
3.2 读/写串口	13
3.3 设置串口属性	13
3.3.1 tcgetattr	17
3.3.2 tcsetattr	17
3.3.3 cfgetispeed	18
3.3.4 cfgetospeed	18
3.3.5 cfsetispeed	19
3.3.6 cfsetospeed	19
3.3.7 cfsetspeed	19
3.3.8 tcflush	20
<b>4 模块使用范例</b>	<b>21</b>
<b>5 调试方法</b>	<b>26</b>
5.1 调试工具	26
5.2 调试节点	26
5.2.1 /sys/devices/platform/soc*/uart*/dev_info	26
5.2.2 /sys/devices/platform/soc*/uart*/status	27

5.2.3	/sys/devices/platform/soc*/uart*/ctrl_info	27
-------	--	----

## 6 FAQ 28

6.1	如何切换 uart 口为打印 conole	28
6.2	如何切换 uart 波特率	28
6.3	如何测试 UART 回环功能	30
6.4	接收数据时为何分段	30

## 插图

图 2-1	初始化流程	4
图 2-2	内核 menuconfig 根菜单	8
图 2-3	Allwinner BSP 菜单	9
图 2-4	Device Drivers 菜单	10
图 2-5	UART Drivers 配置菜单	11
图 2-6	Linux UART 体系结构图	12
图 6-1	时钟说明	28
图 6-2	波特率关系	29
图 6-3	接收数据分段	31

# 1 前言

## 1.1 文档简介

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 目标读者

UART 驱动、及应用层的开发/维护人员。

## 1.3 适用范围

表 1-1: 适用产品列表

内核版本	驱动文件
Linux-5.10	sunxi-uart.c
Linux-5.15	sunxi-uart.c

## 1.4 相关术语介绍

### 1.4.1 硬件术语

表 1-2: 硬件术语

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

术语	解释说明
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 模块介绍

### 2.1 模块功能介绍

通用异步收发器 (Universal Asynchronous Receiver/Transmitter)，通常称作UART，是一种串行、异步、全双工的通信协议，在嵌入式领域应用非常广泛。在 19 世纪 60 年代，为了解决计算机和电传打字机通信，Bell 发明了 UART 协议，将并行输入信号转换成串行输出信号。因为 UART 简单实用的特性，其已经成为一种使用非常广泛的通讯协议。全志的 uart 控制器支持以下功能：

- 兼容标准 16550 串口
- 支持发送接收 FIFO（不同平台 FIFO 大小有差异，具体情况见 user manual/datasheet）
- 支持 DMA 控制接口
- 支持软件/硬件流控配置
- 支持 2/4/8 线串口接口
- 支持 5-8 数据位和 1/1.5/2 停止位
- 支持奇偶校验，无校验位

### 2.2 模块流程介绍

下图为 UART 模块初始化流程介绍：



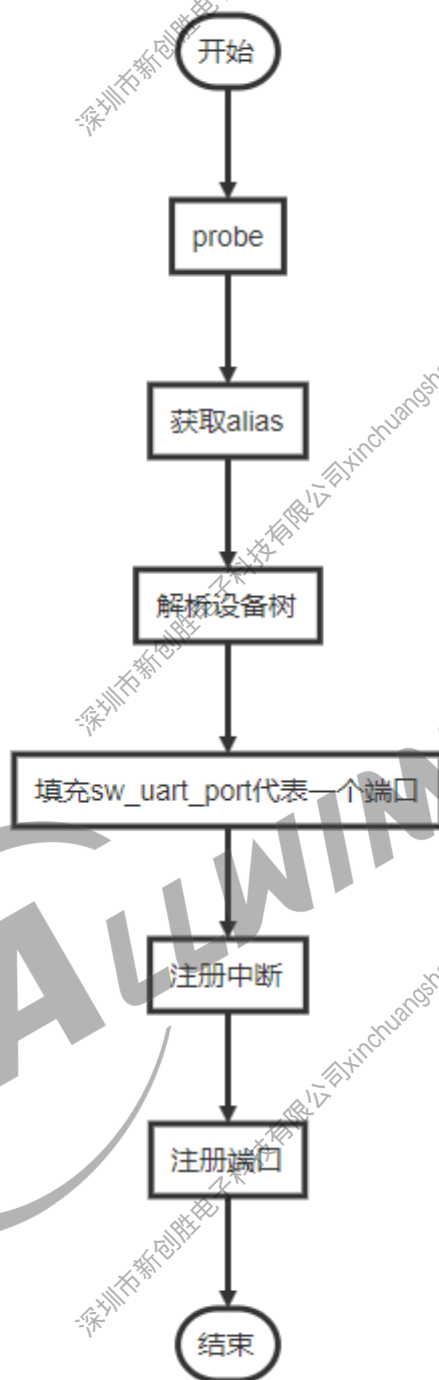


图 2-1: 初始化流程

## 2.3 模块配置介绍

### 2.3.1 设备树配置说明

- 设备树文件的配置是该SoC所有方案的通用配置，soc 设备树的路径为内核目录下：bsp/configs/linux/sun\*.dtsi
- 板级设备树路径：device/config/chips/{IC}/configs/{BOARD}/linux-{kernel-ver}/board.dts。

#### 2.3.1.1 soc 设备树配置

```
uart0: uart@1c28000 {
    compatible = "allwinner,sun8i-uart";
    device_type = "uart0";
    reg = <0x0 0x01c28000 0x0 0x400>;
    interrupts = <GIC_SPI 1 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&ccu CLK_BUS_UART0>; /* 设备使用的时钟 */
    resets = <&ccu RST_BUS_UART0>; /* 设备reset时钟 */
    uart0_port = <0>; /* 端口号 */
    uart0_type = <2>; /* 该串口支持的模式，通常有2/4/8 */
    sunxi,uart-fifosize = <64>; /* 模块FIFO深度 */
    status = "disabled";
};
```

1. compatible表征具体的设备,用于驱动和设备的绑定；（注：客户无需关注）
2. reg设备使用的地址；（注：客户无需关注）
3. interrupts设备使用的中断；（注：客户无需关注）
4. clocks/reset设备使用的时钟；（注：客户无需关注）
5. uart0\_port端口号；（注：客户无需关注）
6. uart0\_type表示当前为几线模式；（注：客户无需关注）
7. sunxi,uart-fifosize表示fifo深度（注：客户无需关注）
8. status是否使能该设备节点；（注：客户无需关注）

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 节点中为每一个UART 节点指定别名。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。如下所示：

```
aliases {
    .....
    serial0 = &uart0;
    serial1 = &uart1;
    serial2 = &uart2;
    serial3 = &uart3;
    serial4 = &uart4;
    serial5 = &uart5;
    serial6 = &uart6;
```

```
serial7 = &uart7;  
.....  
};
```

### 2.3.1.2 板级设备树配置

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, 则遵循以下规则:

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。

board.dts配置示例如下:

```
&pio {  
    uart0_pins_a: uart0_pins@0 {  
        pins = "PB22", "PB23";  
        function = "uart0";  
    };  
  
    uart0_pins_b: uart0_pins@1 {  
        pins = "PB22", "PB23";  
        function = "gpio_in";  
    };  
};  
  
&uart0 {  
    pinctrl-names = "default", "sleep";  
    pinctrl-0 = <&uart0_pins_a>;  
    pinctrl-1 = <&uart0_pins_b>;  
    status = "okay";  
};
```

1. “pinctrl-0” 设备使用状态下的 GPIO 配置, “pinctrl-1” 设备休眠状态下的 GPIO 配置; (注: 客户需关注, 需与真正使用的 GPIO 引脚一致)
2. status = "okay" 当前 uart 的状态; (注: 客户需关注, 当使用 uart 时需要打开)
3. pins 当前 uart 使用的硬件 pin 脚; (注: 若开启硬件流控时, 需要添加 rts 及 cts 引脚, 具体的引脚号需查看芯片手册。)

### 2.3.1.3 dma 模式配置

使用前提:

当设备传输数据量**很大**时，若使用cpu传输，会导致系统负载过大，响应不及时。因此可以使用DMA模式来降低CPU负载。

若传输数据量较小时，请使用默认的cpu传输模式，因为dma模式不一定能够提升传输效率。

dma模式配置流程如下：

1. 在内核配置菜单打开CONFIG\_AW\_SERIAL\_DMA配置。
2. 在对应dts配置开启dma，如下所示：

```
&uart6 {  
    .....  
    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 */  
    dmas = <&dma 14>, <&dma 14>; /* 14表示DRQ, 查看dma spec */  
    dma-names = "tx", "rx";  
    .....  
};
```

### 2.3.2 menuconfig 配置说明

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 Allwinner BSP 选项进入下一级配置，如下图所示。

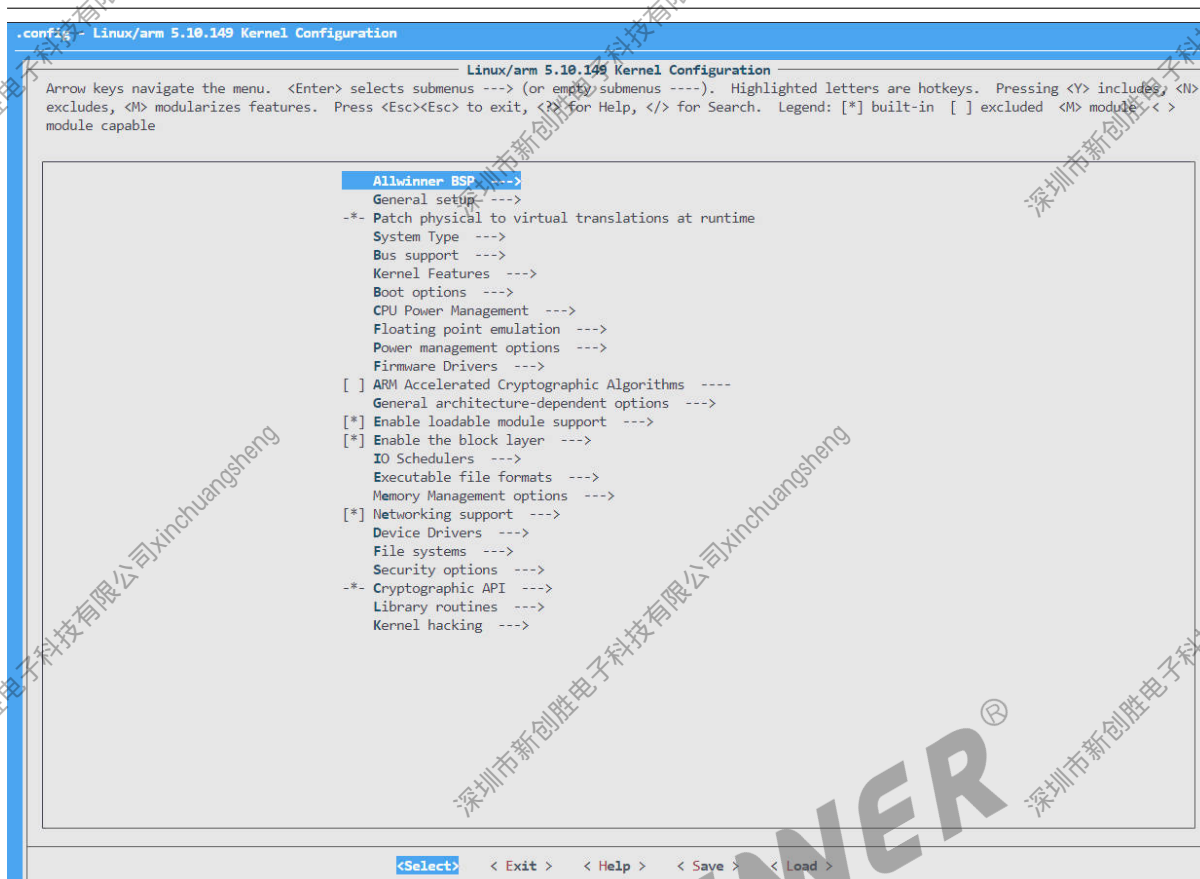


图 2-2: 内核 menuconfig 根菜单

- 选择 Device Drivers 进入下级配置，如下图所示：

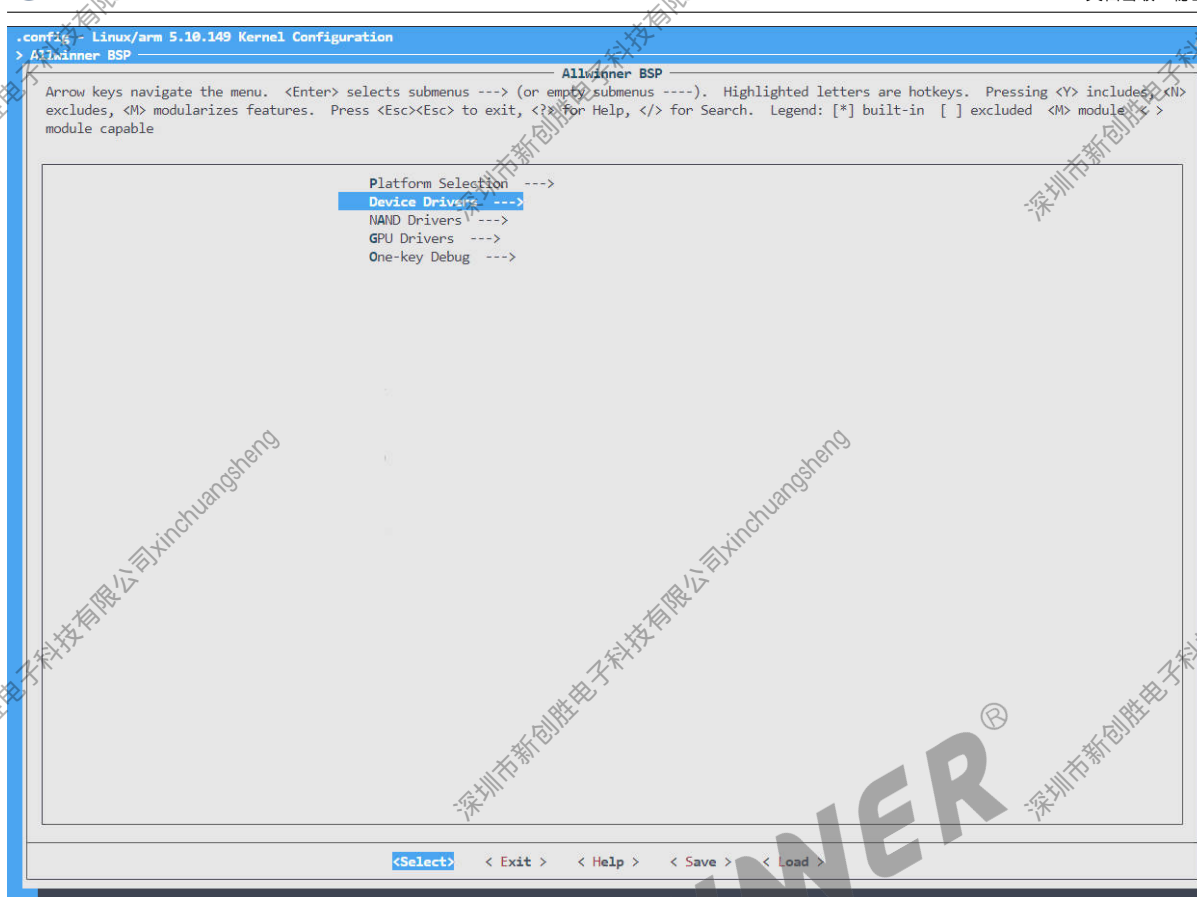


图 2-3: Allwinner BSP 菜单

- 选择 UART Drivers 进入下级配置，如下图所示：

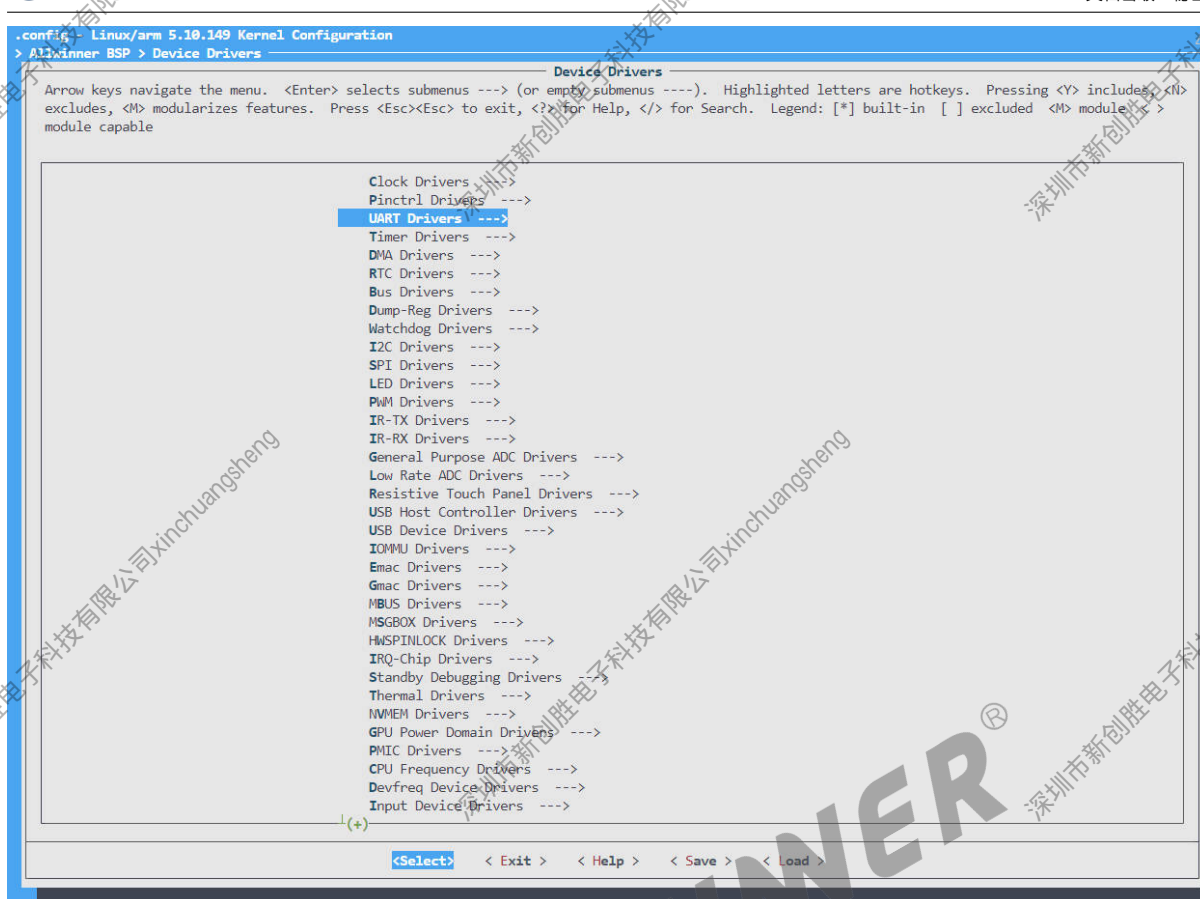


图 2-4: Device Drivers 菜单

- 选择 UART Support for Allwinner SoCs 和 Enable Console on UART 选项，如下图：

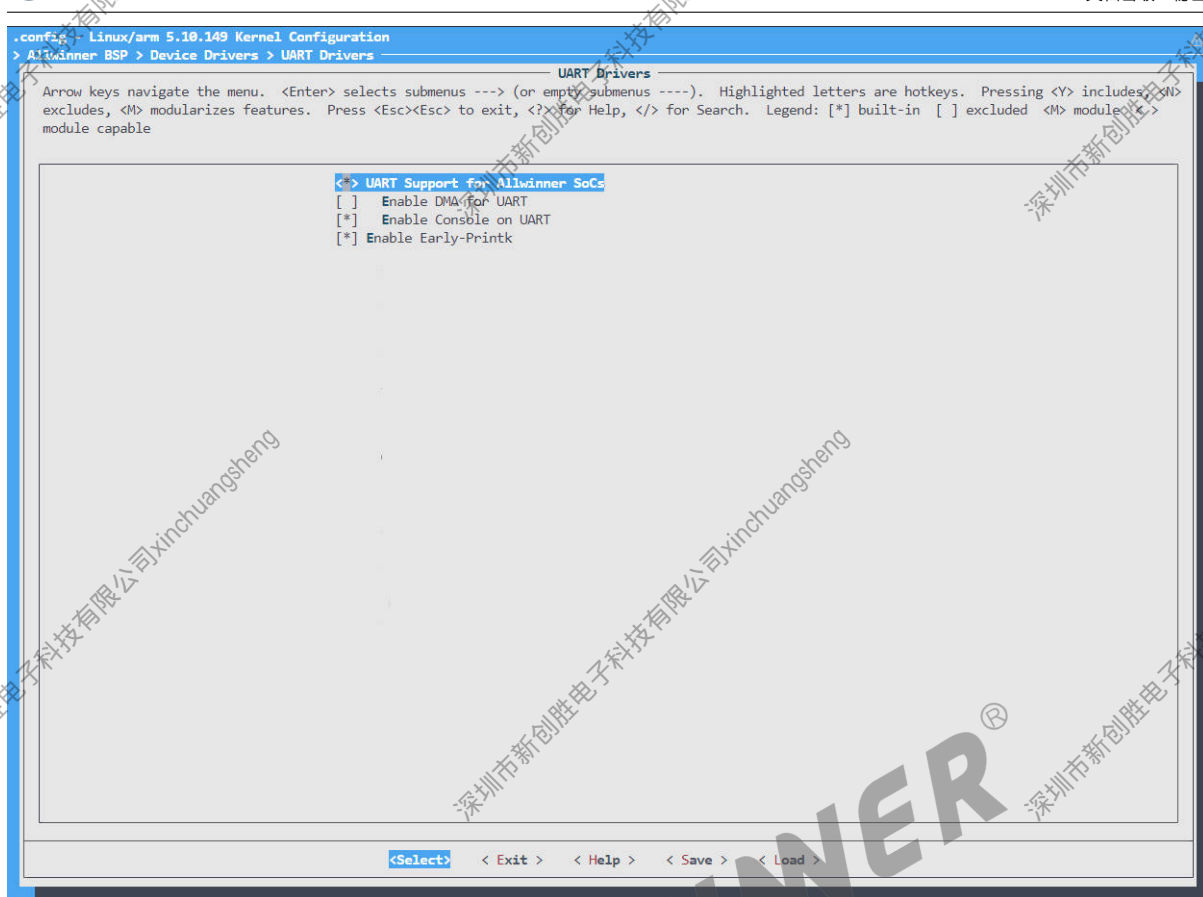


图 2-5: UART Drivers 配置菜单

## 2.4 源码结构介绍

```
-- bsp
|-- drivers
|   |-- uart
|       |-- early_printk.c
|       |-- Kconfig
|       |-- Makefile
|       |-- sunxi-uart.c
|       |-- sunxi-uart.h
```

- early\_printk.c 用于 earlyprintk 功能，一般打开。
- sunxi-uart.c：sunxi 平台驱动核心文件。
- sunxi-uart.h：sunxi 头文件，用来描述硬件信息。

## 2.5 驱动框架介绍

Linux 内核中，UART 驱动的结构图 1 所示，可以分为三个层次：



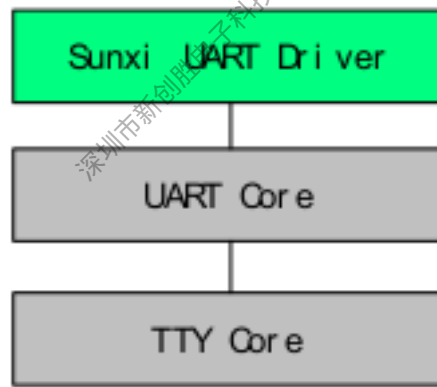


图 2-6: Linux UART 体系结构图

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。

## 3 模块接口说明

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。

### 3.1 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：

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
```

### 3.2 读/写串口

同样使用标准的文件读写函数：

```
1 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：

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

### 3.3 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：（termios.h）。

```
1 #define NCCS 19
2 struct termios {
3     tcflag_t c_iflag; /* input mode flags */
```

```

4  tcflag_t c_oflag;      /* output mode flags */
5  tcflag_t c_cflag;      /* control mode flags */
6  tcflag_t c_lflag;      /* local mode flags */
7  cc_t c_line;           /* line discipline */
8  cc_t c_cc[NCCS];       /* control characters */
9  };

```

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	忽略帧错误和奇偶校验错。
PARMRK	如果没有设置 IGNPAR，在有奇偶校验错或帧错误的字符前插入 \377\0。如果既没有设置 IGNPAR 也没有设置 PARMRK，将有奇偶校验错或帧错误的字符视为 \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) 当输入队列满时响铃。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。
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。

标志	说明
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 sh1)。
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。
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。

标志	说明
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。
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。

标志	说明
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。

### 3.3.1 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, errno 给出具体错误码。

### 3.3.2 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, errno 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。



### 3.3.3 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 `termios.h`。
  - 失败，返回-1，`errno` 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
```

### 3.3.4 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`。
  - 失败，返回-1，`errnor` 给出具体错误码。

### 3.3.5 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。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，`errnor` 给出具体错误码。

### 3.3.6 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。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，`errnor` 给出具体错误码。

### 3.3.7 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。
- 返回：
  - 成功，返回 0。
  - 失败，返回-1，`errnor` 给出具体错误码。



### 3.3.8 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，`errno` 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。

## 4 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。

```
1 #include <stdio.h> /*标准输入输出定义*/
2 #include <stdlib.h> /*标准函数库定义*/
3 #include <unistd.h> /* Unix 标准函数定义*/
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h> /*文件控制定义*/
7 #include <termios.h> /*PPSIX 终端控制定义*/
8 #include <errno.h> /*错误号定义*/
9 #include <string.h>
10
11 enum parameter_type {
12     PT_PROGRAM_NAME = 0,
13     PT_DEV_NAME,
14     PT_CYCLE,
15
16     PT_NUM
17 };
18
19 #define DBG(string, args...) \
20     do { \
21         printf("%s, %s() %u---", __FILE__, __FUNCTION__, __LINE__); \
22         printf(string, ##args); \
23         printf("\n"); \
24     } while (0)
25
26 void usage(void)
27 {
28     printf("You should input as: \n");
29     printf("\t select_test [/dev/name] [Cycle Cnt]\n");
30 }
31
32 int OpenDev(char *name)
33 {
34     int fd = open(name, O_RDWR);    /// O_NOCTTY | O_NDELAY
35     if (-1 == fd)
36         DBG("Can't Open(%s)!", name);
37
38     return fd;
39 }
40
41 /**
42  * @brief 设置串口通信速率
43  * @param fd 类型 int 打开串口的文件句柄
44  * @param speed 类型 int 串口速度
45  * @return void
46  */
47 void set_speed(int fd, int speed){
```

```
48     int i;
49     int status;
50     struct termios Opt = {0};
51     int speed_arr[] = { B38400, B19200, B9600, B4800, B2400, B1200, B300,
52                        B38400, B19200, B9600, B4800, B2400, B1200, B300, };
53     int name_arr[] = {38400, 19200, 9600, 4800, 2400, 1200, 300, 38400,
54                      19200, 9600, 4800, 2400, 1200, 300, };
55
56     tcgetattr(fd, &Opt);
57
58     for ( i = 0; i < sizeof(speed_arr) / sizeof(int); i++) {
59         if (speed == name_arr[i])
60             break;
61     }
62
63     tcflush(fd, TCIOFLUSH);
64     cfsetispeed(&Opt, speed_arr[i]);
65     cfsetospeed(&Opt, speed_arr[i]);
66
67     Opt.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); /*Input*/
68     Opt.c_oflag &= ~OPOST; /*Output*/
69
70     status = tcsetattr(fd, TCSANOW, &Opt);
71     if (status != 0) {
72         DBG("tcsetattr fd");
73         return;
74     }
75     tcflush(fd, TCIOFLUSH);
76 }
77
78 /**
79  * @brief 设置串口数据位，停止位和校验位
80  * @param fd 类型 int 打开的串口文件句柄
81  * @param databits 类型 int 数据位 取值为 7 或者 8
82  * @param stopbits 类型 int 停止位 取值为 1 或者 2
83  * @param parity 类型 int 校验类型 取值为 N,E,O,S
84  */
85 int set_Parity(int fd, int databits, int stopbits, int parity)
86 {
87     struct termios options;
88
89     if ( tcgetattr( fd, &options) != 0) {
90         perror("SetupSerial 1");
91         return -1;
92     }
93     options.c_cflag &= ~CSIZE;
94
95     switch (databits) /*设置数据位数*/
96     {
97     case 7:
98         options.c_cflag |= CS7;
99         break;
100    case 8:
101        options.c_cflag |= CS8;
102        break;
103    default:
104        fprintf(stderr, "Unsupported data size\n");
105        return -1;
106    }
107 }
```

```
108 switch (parity)
109 {
110 case 'n':
111 case 'N':
112     options.c_cflag &= ~PARENB; /* Clear parity enable */
113     options.c_iflag &= ~INPCK; /* Enable parity checking */
114     break;
115 case 'o':
116 case 'O':
117     options.c_cflag |= (PARODD | PARENB); /* 设置为奇效验 */
118     options.c_iflag |= INPCK; /* Disable parity checking */
119     break;
120 case 'e':
121 case 'E':
122     options.c_cflag |= PARENB; /* Enable parity */
123     options.c_cflag &= ~PARODD; /* 转换为偶效验 */
124     options.c_iflag |= INPCK; /* Disable parity checking */
125     break;
126 case 'S':
127 case 's': /* as no parity */
128     options.c_cflag &= ~PARENB;
129     options.c_cflag &= ~CSTOPB; break;
130 default:
131     fprintf (stderr, "Unsupported parity\n");
132     return -1;
133 }
134
135 /* 设置停止位 */
136 switch (stopbits)
137 {
138 case 1:
139     options.c_cflag &= ~CSTOPB;
140     break;
141 case 2:
142     options.c_cflag |= CSTOPB;
143     break;
144 default:
145     fprintf (stderr, "Unsupported stop bits\n");
146     return -1;
147 }
148
149 /* Set input parity option */
150 if (parity != 'n')
151     options.c_iflag |= INPCK;
152 tcflush(fd, TCIFLUSH);
153 options.c_cc[VTIME] = 150; /* 设置超时15 seconds */
154 options.c_cc[VMIN] = 0; /* Update the options and do it NOW */
155 if (tcsetattr (fd, TCSANOW, &options) != 0)
156 {
157     perror("SetupSerial 3");
158     return -1;
159 }
160 return 0;
161 }
162
163 void str_print(char *buf, int len)
164 {
165     int i;
166     for (i=0; i<len; i++) {
```

```
168     if (i%10 == 0)
169         printf("\n");
170
171         printf("0x%02x ", buf[i]);
172     }
173     printf("\n");
174 }
175
176 int main(int argc, char **argv)
177 {
178     int i = 0;
179     int fd = 0;
180     int cnt = 0;
181     char buf[256];
182
183     int ret;
184     fd_set rd_fdset;
185     struct timeval dly_tm;    // delay time in select()
186
187     if (argc != PT_NUM) {
188         usage();
189         return -1;
190     }
191
192     sscanf(argv[PT_CYCLE], "%d", &cnt);
193     if (cnt == 0)
194         cnt = 0xFFFF;
195
196     fd = OpenDev(argv[PT_DEV_NAME]);
197     if (fd < 0)
198         return -1;
199
200     set_speed(fd, 19200);
201     if (set_Parity(fd, 8, 1, 'N') == -1) {
202         printf("Set Parity Error\n");
203         exit (0);
204     }
205
206     printf("Select(%s), Cnt %d. \n", argv[PT_DEV_NAME], cnt);
207     while (i < cnt) {
208         FD_ZERO(&rd_fdset);
209         FD_SET(fd, &rd_fdset);
210
211         dly_tm.tv_sec = 5;
212         dly_tm.tv_usec = 0;
213         memset(buf, 0, 256);
214
215         ret = select(fd+1, &rd_fdset, NULL, NULL, &dly_tm);
216         // DBG("select() return %d, fd = %d", ret, fd);
217         if (ret == 0)
218             continue;
219
220         if (ret < 0) {
221             printf("select(%s) return %d. [%d]: %s \n", argv[PT_DEV_NAME], ret, errno, strerror(errno));
222             continue;
223         }
224
225         i++;
226         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 }
```

## 5 调试方法

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

### 5.1 调试工具

### 5.2 调试节点

#### 5.2.1 /sys/devices/platform/soc/\*uart\*/dev\_info

从该节点可以看到 UART 端口的一些硬件资源信息。以uart0为例：

```
# cat /sys/devices/platform/soc/uart0/dev_info
id = 0
name = uart0
irq = 247
io_num = 2
port->mapbase = 0x0000000005000000
port->membase = 0xfffff800b005000
port->iobase = 0x00000000
pdata->regulator = 0x (null)
pdata->regulator_id
```

- id: uart编号
- name: uart名称
- irq: uart中断号
- io\_num: 表明 uart0 目前被配置为为 2 线模式

## 5.2.2 /sys/devices/platform/soc\*/\*uart\*/status

此节点可以打印出 UART 端口的一些运行状态信息，包括控制器的各寄存器值。以uart0为例：

```
# cat /sys/devices/platform/soc/uart0/status
uartclk = 24000000
The Uart controller register[Base: 0xfffff800b005000]:
[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
```

## 5.2.3 /sys/devices/platform/soc\*/\*uart\*/ctrl\_info

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。

```
# 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
```

重要信息如下：

- tx：当前发出的数据总量
- rx：接收到的数据总量
- frame：frame 错误的次数。
- overrun：overrun 的次数，代表当前 fifo 溢出。



## 6 FAQ

### 6.1 如何切换 uart 口为打印 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
```

### 6.2 如何切换 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：

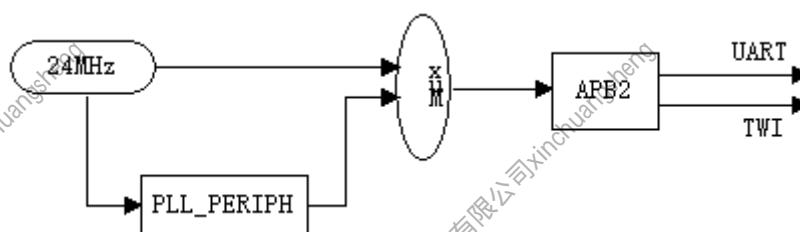


图 6-1: 时钟说明

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.  $24\text{MHz}/16=1.5\text{MHz}$ ，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 APB，在不同平台上可能是 APB1 或 APB2，APB 的时钟源有多个，包括 24MHz（HOSC）和 PLL\_PERIPH（即驱动中的 PLL\_PERIPH\_CLK），系统默认配置 APB 的时钟源是 24M，如果要提高 UART 的时钟就要将 APB 的时钟源设置为 PLL\_PERIPH，一般挑选 600M 的 PLL\_PERIPH 时钟。同时要注意到 APB 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：

the reference table as follows:

pll6 600M	0	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5
apb2div	24000000	30000000	31578947	33333333	35294117	37500000	40000000	42857142	46153846	50000000	54545454	60000000	66666666	75000000	85714285	100000000	120000000
115200	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
230400	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
384000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
460800	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
921600	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1000000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1500000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1750000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
2000000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
2500000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
3000000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
3250000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
3500000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
4000000	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

图 6-2: 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 APB 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 时钟源：

支持 460800 波特率需要修改的 dts 内容如下，注意 uart0 及其他在用时钟也需要同步修改保持一致性，避免乱码：

```
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。

## 6.3 如何测试 UART 回环功能

回环测试分为硬件回环与软件回环：

- 若需要使用硬件回环，需要短接 uart 的 tx 与 rx 引脚
- 若需要使用软件回环，可以在 /sys/devices/platform 路径下找到 uart 的 loopback 节点，使用如下指令即可配置软件回环：

```
echo enable > loopback
```

- 关闭软件回环

```
echo disable > loopback
```

若需要测试 uart 回环模式是否正常工作，可向我司获取 uarttest 文件进行测试。或者参考“模块使用范例”章节。

注意：除了 "enable"，其他所有字符都表示不开启回环

## 6.4 接收数据时为何分段

当波特率小于等于 9600 时，uart 接收数据可能会分段。

比如一次性发送数据“12345678”，但是接收会分多次，第一次接收“1234”，第二次接收“5678”。现象类似下图：



图 6-3: 接收数据分段

根本原因:

- 当波特率小于等于 9600 时:

硬件接收到 1 个字节数据时, 即触发接收中断。此时 cpu 运行速率远大于 uart 传输速率, 因此 cpu 读取完一个字节之后就会立刻上报。此时用户层 read 线程被唤醒, 大概率只能读到一个字节数据。

- 大于 9600 波特率时:

触发接收中断的场景有两种: (1) 当接收数据到达 1/2 FIFO

(2) 未到达 1/2 FIFO, 但在一定的时间未接收到数据

此时 cpu 触发中断的时间会有一定延迟, FIFO 中会有多个数据存留。此时用户层 read 线程被唤醒, 大概率会完整读到整包数据。

修改方法:

```
@@ -1245,14 +1245,9 @@ static void sw_uart_set_termios(struct uart_port *port, struct ktermios *termios
*/
sw_uart_reset(sw_uport);

- if (baud <= 9600)
-     sw_uport->fcr = SUNXI_UART_FCR_RXTRG_1CH
-         | SUNXI_UART_FCR_TXTRG_1_2
-         | SUNXI_UART_FCR_FIFO_EN;
- else
-     sw_uport->fcr = SUNXI_UART_FCR_RXTRG_1_2
-         | SUNXI_UART_FCR_TXTRG_1_2
-         | SUNXI_UART_FCR_FIFO_EN;
+ sw_uport->fcr = SUNXI_UART_FCR_RXTRG_1_2
+     | SUNXI_UART_FCR_TXTRG_1_2
+     | SUNXI_UART_FCR_FIFO_EN;
```

```
serial_out(port, sw_uport->fcr, SUNXI_UART_FCR);
```




## 著作权声明

版权所有 © 2024 珠海全志科技股份有限公司。保留一切权利。

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。