全志科技
Allwinner Technology

V40 项目

UART 接口使用说明书 V1.0

文档履历

版本号	日期	制/修订人	内容描述
V1.0	2016-06-08		正式版本

目　录

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

适用于 V40 sdk 配套的 Linux 3.10 内核。

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 2.1 所示，可以分为三个层次：

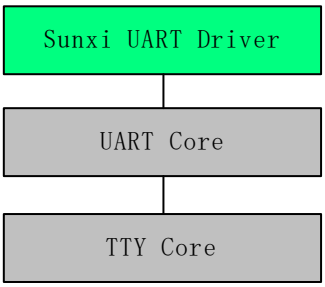

图 2.1 Linux UART 体系结构图

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

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. 模块配置介绍

2.3.1. Device Tree 配置说明

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，如下：

```
uart0: uart@01c28000 {
        compatible = "allwinner,sun8i-uart";
        reg = <0x0 0x01c28000 0x0 0x400>;
        interrupts = <GIC_SPI 1 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clk_uart0>;
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins_a>;
```

```
        pinctrl-1 = <&uart0_pins_b>;
        port-number = <0>;
        io-number = <2>;
        status = "disabled";
};
```

其中：

1. compatible：表征具体的设备，用于驱动和设备的绑定；
2. reg：设备使用的地址；
3. interrupts：设备使用的中断；
4. clocks：设备使用的时钟；
5. pinctrl-0：设备使用的 PIN 脚；
6. port-number：UART 控制器对应的 ttyS 端口编号，取值不能和其它 UART 控制器重复；
7. io-number：UART 控制器的线数，取值范围为 2、4 和 8；
8. status：是否使用该节点。

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：

```
        serial0 = &uart0;
        serial1 = &uart1;
        serial2 = &uart2;
        serial3 = &uart3;
        serial4 = &uart4;
        serial5 = &uart5;
        serial6 = &uart6;
        serial7 = &uart7;
```

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 UART 控制器的数字编号，从而区别每一个 UART 控制器。

2.3.2. menuconfig 配置说明

在命令行中进入内核根目录，执行 make ARCH=arm menuconfig 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：

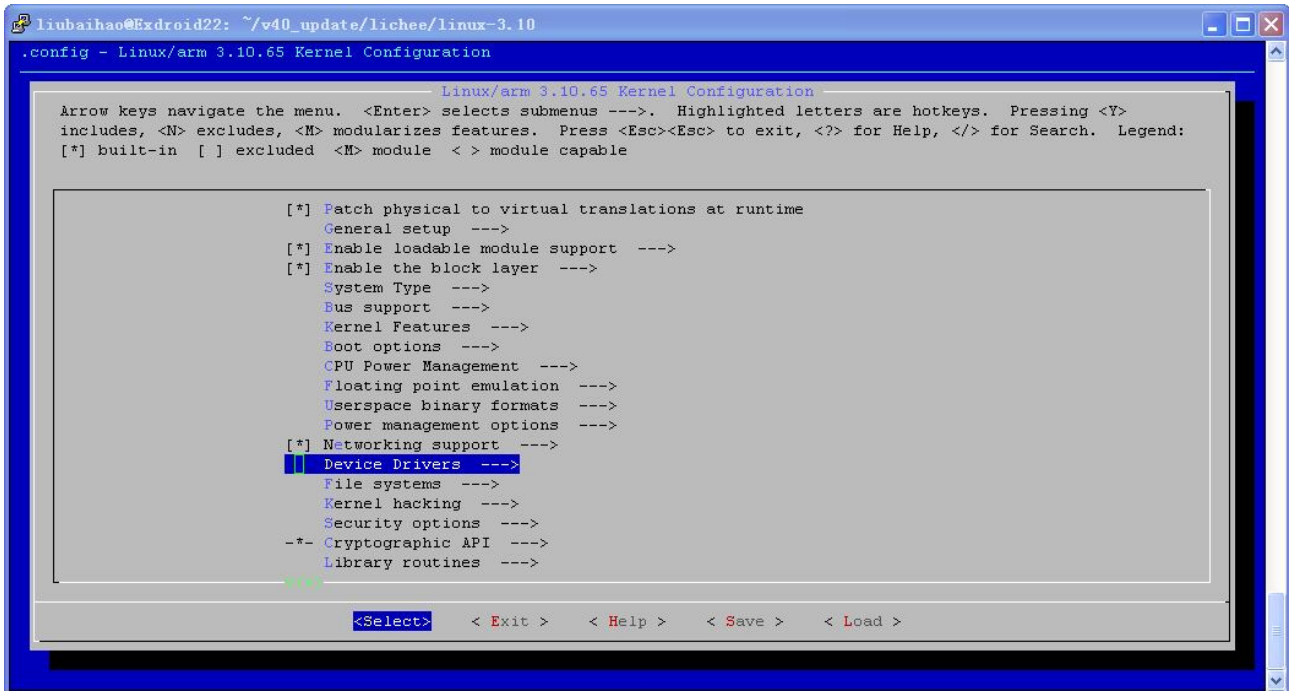

图 2.3 Device Drivers 配置

然后，选择 Character devices 选项，进入下一级配置，如下图所示：

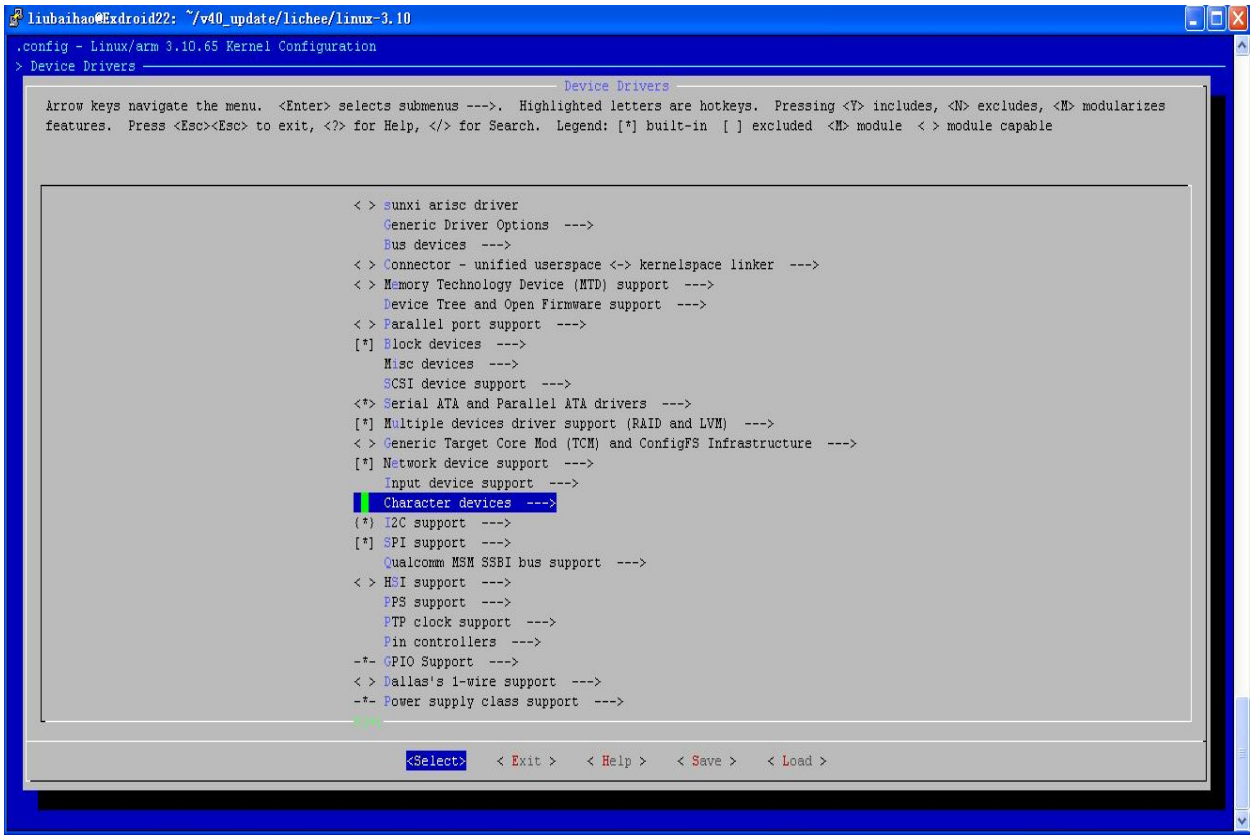

图 2.4 Character devices 配置选项

接着选择 Serial drivers 选项，进入下一级配置，如下图所示：

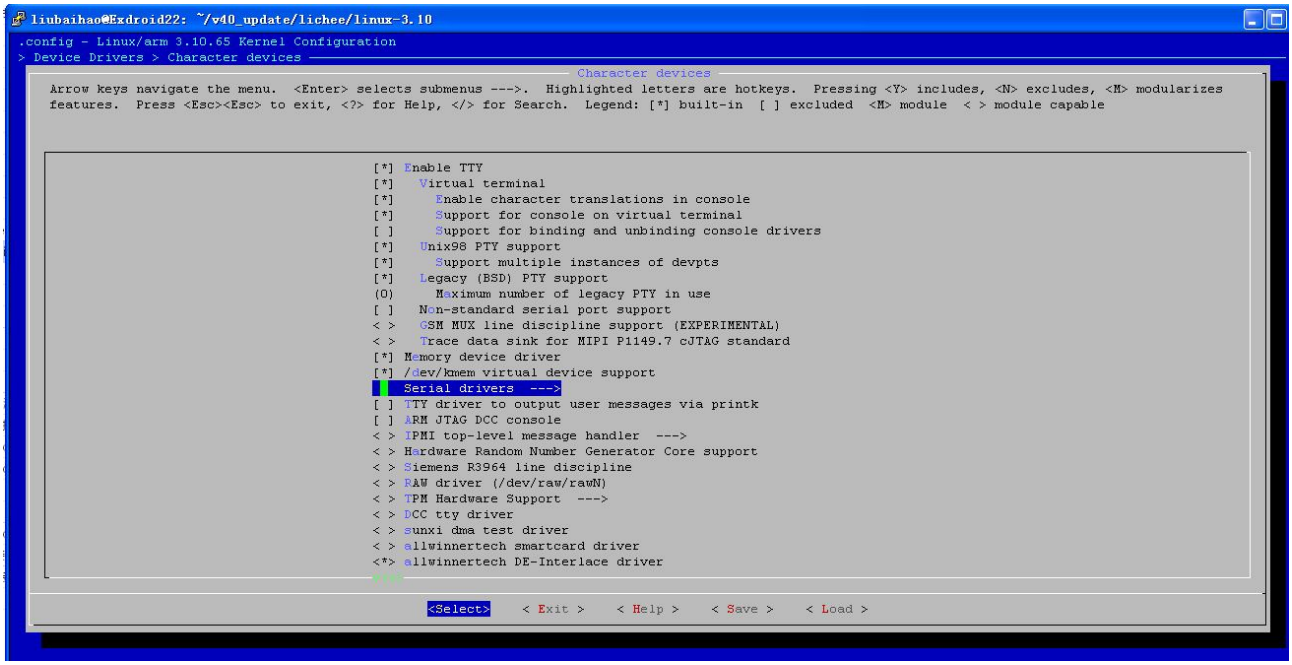

图 2.5 Serial drivers 配置选项

选择 SUNXI UART Controller 选项，因为 UART0 要用作调试串口，所以下面的“Console on SUNXI UART port”默认是选中状态，且不要将 UART 设置成模块的编译方式。如下图：

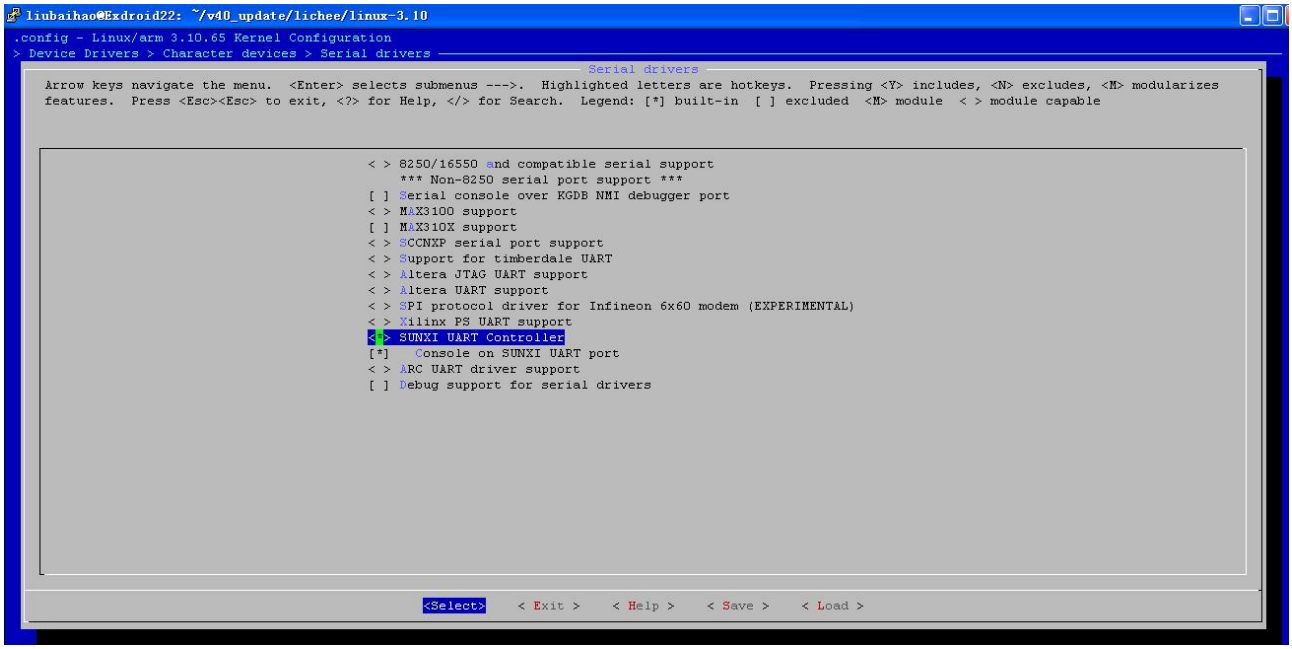

图 2.6 SUNXI UART Controller 配置选项

2.4. 源码结构介绍

UART 驱动的源代码位于内核在 drivers/tty/serial 目录下：

drivers/tty/serial/

├── sunxi-uart.c // Sunxi 平台的 UART 控制器驱动代码

├── sunxi-uart.h // 为 Sunxi 平台的 UART 控制器驱动定义了一些宏、数据结构

3. 模块体系结构设计

UART 驱动模块内部的功能划分如下图所示：

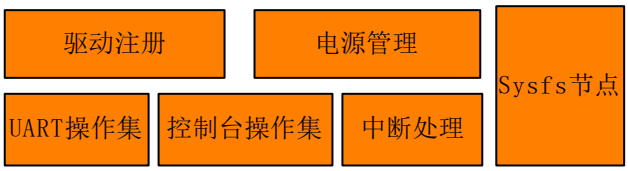

图 3.1 UART 总线驱动的内部结构

其中：

1. **驱动注册**，向 UART Core 注册 UART 驱动，驱动注册后会跟 Device Tree 中描述的 UART 控制器进行匹配，匹配成功后会调用 probe 函数，初始化 UART 控制器；
2. **电源管理**，负责 UART 控制器的休眠、唤醒功能；
3. **UART 操作集**，需要提供一套符合 UART Core 标准的接口集，Uart Core 通过这些接口控制 UART 控制器完成数据的收发；
4. **控制台操作集**，UART0 作为控制台，需要按照 console 的注册标准实现一组读写接口；
5. **中断处理**，UART 控制器在数据传输结束、发生错误时会触发中断，所有 UART 控制器的中断共享此中断处理函数；
6. **Sysfs 节点**，提供一些在线的调试方法。

4. 模块流程设计

UART 操作集 uart_ops 已经非常细化，每个接口实现一个基本功能，定义如下：

```
struct uart_ops {
    unsigned int    (*tx_empty)(struct uart_port *);
    void        (*set_mctrl)(struct uart_port *, unsigned int mctrl);
    unsigned int    (*get_mctrl)(struct uart_port *);
    void        (*stop_tx)(struct uart_port *);
    void        (*start_tx)(struct uart_port *);
    void        (*send_xchar)(struct uart_port *, char ch);
    void        (*stop_rx)(struct uart_port *);
    void        (*enable_ms)(struct uart_port *);
    void        (*break_ctl)(struct uart_port *, int ctl);
    int     (*startup)(struct uart_port *);
    void        (*shutdown)(struct uart_port *);
    void        (*flush_buffer)(struct uart_port *);
    void        (*set_termios)(struct uart_port *, struct ktermios *new,
                       struct ktermios *old);
    void        (*set_ldisc)(struct uart_port *, int new);
    void        (*pm)(struct uart_port *, unsigned int state,
                  unsigned int oldstate);
    int     (*set_wake)(struct uart_port *, unsigned int state);
    void        (*wake_peer)(struct uart_port *);

    const char *(*type)(struct uart_port *);

    void        (*release_port)(struct uart_port *);
    int     (*request_port)(struct uart_port *);
    void        (*config_port)(struct uart_port *, int);
    int     (*verify_port)(struct uart_port *, struct serial_struct *);
    int     (*ioctl)(struct uart_port *, unsigned int, unsigned long);
#ifdef CONFIG_CONSOLE_POLL
    void    (*poll_put_char)(struct uart_port *, unsigned char);
    int     (*poll_get_char)(struct uart_port *);
#endif
};
```

uart_ops 中定义的接口功能在代码实现上没有复杂的流程，大多是控制 UART 寄存器。其中.startup 中首先会注册中断，.set_termios 设置串口属性，当打开一个 UART 设备时都会调用这两个接口来配置 UART 参数，也直接决定了后面的通信是否成功。因此下面仅重点分析 UART 驱动中断处理函数和.set_termios 的处理流程。

4.1. 中断处理

一般情况下，UART 控制器在数据传输准备好、结束后、或者发生错误时会产生中断，就会触发 UART 的中断处理函数 sw_uart_irq()。其处理流程如下图：

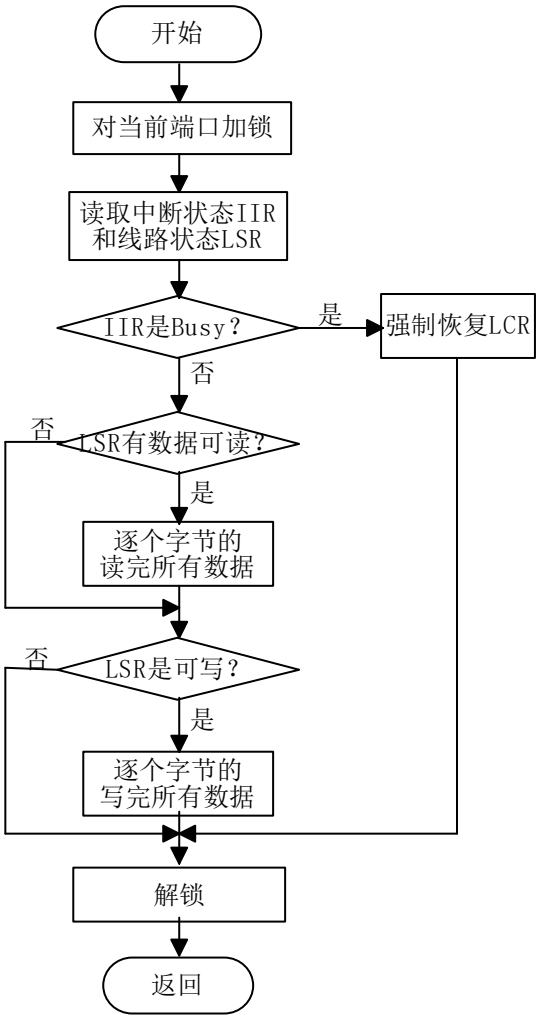

图 4.1 UART 中断处理程序的处理流程

4.2. 设置串口属性

一般情况下，打开一个 UART 设备都需要调整其通信参数，如波特率、停止位、奇偶校验等。UART 驱动中的属性设置接口见函数 sw_uart_set_termios()，其处理流程如下：

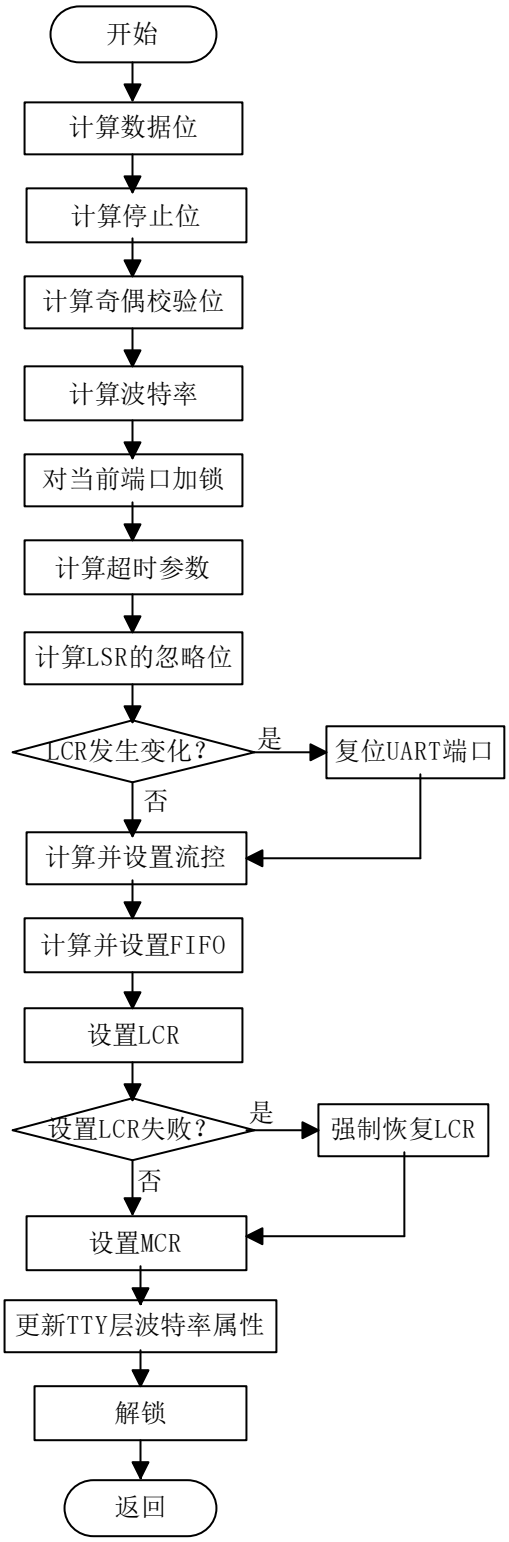

图 4.2 UART 属性设置接口的处理流程

5. 数据结构设计

5.1.跨平台的硬件资源处理

Kernel 会为 Device Tree 中的每一个 UART 节点展开 platform_device 结构，UART 驱动代码中不再需要注册 platform_device。当 UART 驱动和 Device Tree 中描述的 UART 节点匹配成功后，会调用驱动函数的 probe 函数执行，在 probe 函数中再进一步获取 Device Tree 中配置的 UART 节点的硬件资源：

```
static int sw_uart_probe(struct platform_device *pdev)
{
    ...
    pdev->id = of_alias_get_id(np, "serial");
    if (pdev->id < 0) {
        SERIAL_MSG("failed to get alias id\n");
        return -EINVAL;
    }
    ...
    sw_uport->mclk = of_clk_get(np, 0);
    if (IS_ERR(sw_uport->mclk)) {
        SERIAL_MSG("uart%d error to get clk\n", pdev->id);
        return -EINVAL;
    }
    port->uartclk = clk_get_rate(sw_uport->mclk);

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (res == NULL) {
        SERIAL_MSG("uart%d error to get MEM resource\n", pdev->id);
        return -EINVAL;
    }
    port->mapbase = res->start;

    port->irq = irq_of_parse_and_map(np, 0);
    if (port->irq == 0) {
        SERIAL_MSG("uart%d error to get irq\n", pdev->id);
        return -EINVAL;
    }

    ret = of_property_read_u32(np, "port-number", &port->line);
    if (ret) {
        SERIAL_MSG("uart%d error to get reg property\n", pdev->id);
        return -EINVAL;
    }

    ret = of_property_read_u32(np, "io-number", &pdata->io_num);
```

```
    if (ret) {
        SERIAL_MSG("uart%d error to get reg property\n", pdev->id);
        return -EINVAL;
    }
    ...
}
```

这一段代码依次从 Device Tree 中获取 UART 控制器编号、时钟、设备地址、中断号、port-number 和 io-number 属性。

5.2.UART 控制器数据结构

虽然驱动程序可以在 probe 函数中动态分配设备数据结构和初始化设备信息，但是由于 UART 驱动需要支持控制台，而控制台需要通过遍历所有 UART 设备找到支持控制台的 UART 端口，所以定义了两个全局的结构体数组 sw_uart_port[SUNXI_UART_NUM]和 sw_uart_pdata[SUNXI_UART_NUM]，用来保存 UART 控制器的设备信息。其中宏 SUNXI_UART_NUM 用来定义 Sunxi 硬件平台的 UART 控制器数目，如下所示：

```
#ifdef CONFIG_ARCH_SUN8IW1P1
#define SUNXI_UART_NUM          6
#endif
#ifdef CONFIG_ARCH_SUN8IW3P1
#define SUNXI_UART_NUM          5
#endif
#ifdef CONFIG_ARCH_SUN9IW1P1
#define SUNXI_UART_NUM          6
#endif

/* In 50/39 FPGA, two UART is available, but they share one IRQ.
   So we define the number of UART port as 1. */
#ifndef CONFIG_EVB_PLATFORM
#undef SUNXI_UART_NUM
#define SUNXI_UART_NUM          1
#endif
```

5.2.1. sw_uart_port

此结构的定义在 sunxi-uart.h，其中主要是 UART 控制器资源的管理信息。定义如下：

```
struct sw_uart_port {
    struct uart_port port;
    char   name[16];
    struct clk *mclk;
    unsigned char id;
    unsigned char ier;
    unsigned char lcr;
    unsigned char mcr;
    unsigned char fcr;
```

```
    unsigned char dll;
    unsigned char dlh;
    unsigned char msr_saved_flags;
    unsigned int lsr_break_flag;
    struct sw_uart_pdata *pdata;

    /* for debug */
#define MAX_DUMP_SIZE   1024
    unsigned int dump_len;
    char* dump_buff;
    struct proc_dir_entry *proc_root;
    struct proc_dir_entry *proc_info;

    struct pinctrl       *pctrl;
    struct pinctrl_state *pctrl_state;
};
```

5.2.2. sw_uart_pdata

此结构的定义也在 sunxi-uart.h，仅供内部使用。该结构为了方便在 probe 函数中传递当前 UART 端口的几个信息：是否启用、线数、对应的 ttyS 端口号、regulator 信息等。代码如下：

```
struct sw_uart_pdata {
    unsigned int used;
    unsigned int io_num;
    unsigned int port_no;
    char     regulator_id[16];
    struct regulator *regulator;
};
```

6. 接口描述

UART 总线驱动对应用来说没有直接调用的接口，如图 3.1 所示，UART 驱动提供的接口都是给 UART Core 来使用的。

6.1. 内部接口

UART驱动的内部接口除了标准的probe、remove、resume、suspend接口外，这里列出中断处理函数、动态调试接口和 sysfs 调试接口。

6.1.1. sw_uart_irq()

【函数原型】：static irqreturn_t sw_uart_irq(int irq, void *dev_id)
【功能描述】：处理 UART 控制器产生的中断信号
【参数说明】：irq，中断号
dev_id，自定义的回调参数，在此接口中该参数类型是 struct uart_port *
【返 回 值】：IRQ_HANDLED，中断已经出来；IRQ_NONE，不是此设备的中断。

6.1.2. 动态调试接口

UART 驱动使用动态调试接口打印调试信息，通过 CONFIG_DYNAMIC_DEBUG 宏和 CONFIG_SERIAL_DEBUG 宏进行控制，而且 CONFIG_DYNAMIC_DEBUG 宏的优化级高于 CONFIG_SERIAL_DEBUG 宏。

1. CONFIG_DYNAMIC_DEBUG 宏

当打开 CONFIG_DYNAMIC_DEBUG 宏时，可以在有需要时通过文件节点控制驱动打印调试信息，默认情况下不会打印。此文件节点为<debugfs>/dynamic_debug/control，需要挂载 debugfs。

在内核根目录下执行 make ARCH=arm menuconfig，打开 CONFIG_DYNAMIC_DEBUG 宏的方法：

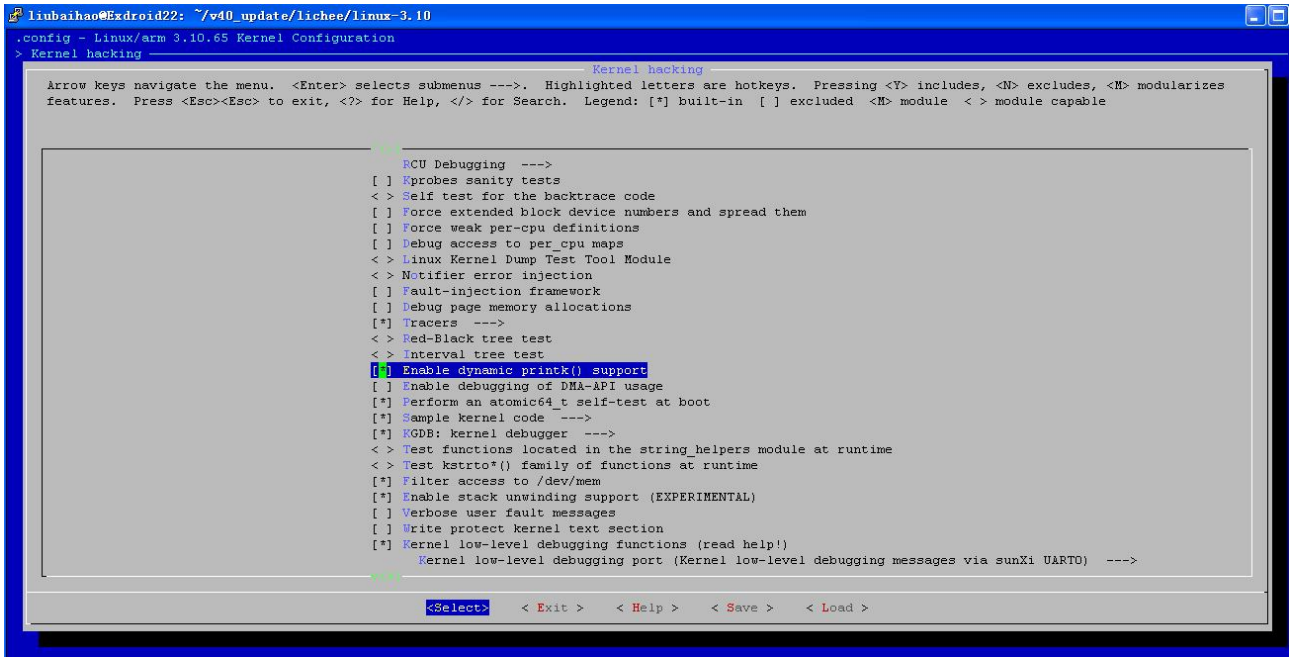

图 6.1 Kernel hacking -> Enable dynamic printk() support

挂载 debugfs 的方法：

mount -t debugfs none /sys/kernel/debug/

查看所有动态调试信息的方法：

cat /sys/kernel/debug/dynamic_debug/control

放开模块所有打印的方法：

echo "module sunxi_uart +p" > /sys/kernel/debug/dynamic_debug/control

放开文件所有打印的方法：

echo "file sunxi-uart.c +p" > /sys/kernel/debug/dynamic_debug/control

放开文件指定行打印的方法：

echo "file sunxi-uart.c line 615 +p" > /sys/kernel/debug/dynamic_debug/control

放开函数所有打印的方法：

echo "func sw_uart_set_termios +p" > /sys/kernel/debug/dynamic_debug/control

要关闭相应的打印只需要把“+p”改成“-p”即可，更多信息可以参考 linux 内核中的文档：Documentation/dynamic-debug-howto.txt。

2. CONFIG_SERIAL_DEBUG 宏

当关闭 CONFIG_DYNAMIC_DEBUG 宏，打开 CONFIG_SERIAL_DEBUG 宏时，不再通过文件节点控制驱动打印调试信息，而是从内核启动阶段开始就可以打印驱动调试信息，打印级别为 KERN_DEBUG。

打开 CONFIG_SERIAL_DEBUG 宏的方法：

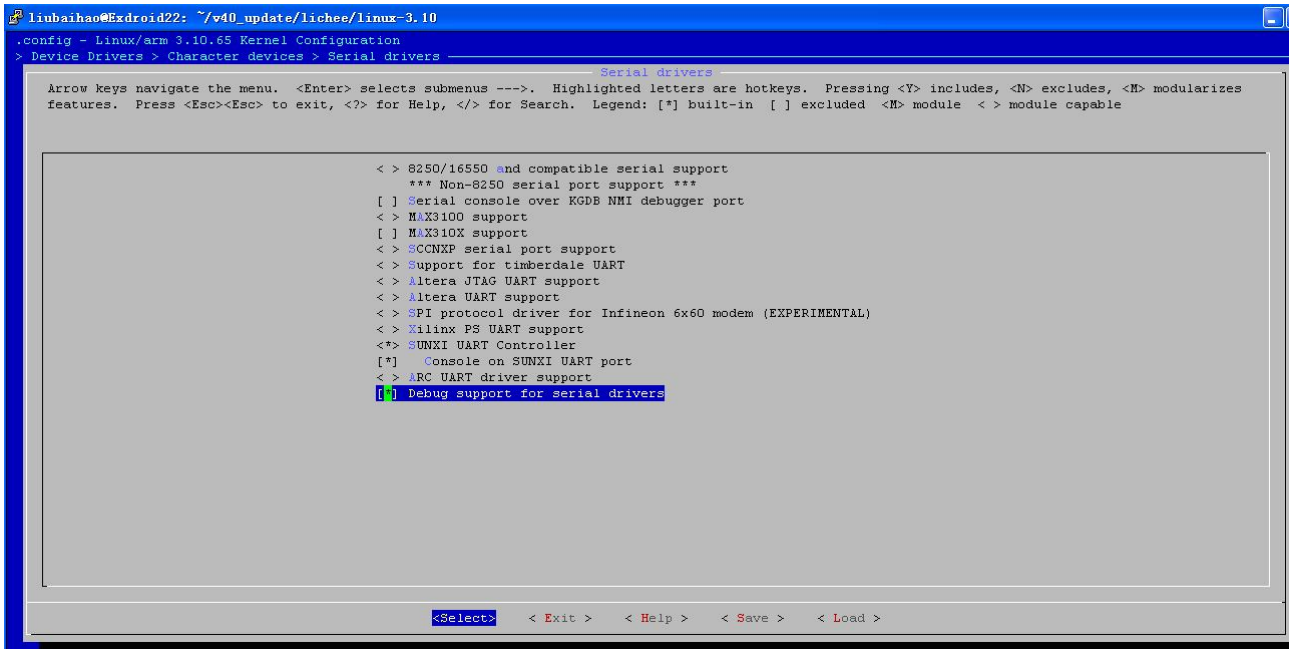

图 6.2 Device Drivers -> Character devices -> Serial drivers -> Debug support for serial drivers

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接口：

1. /sys/devices/soc.2/1c28000.uart/dev_info

此节点文件可以打印出当前 UART 端口的一些硬件资源信息。操作效果图：

```
/ # cat /sys/devices/soc.2/1c28000.uart/dev_info
id      = 0
name    = uart0
irq     = 32
io_num = 2
port->mapbase = 0x0000000001c28000
port->membase = 0xffffff8000012000
port->iobase  = 0x00000000
pdata->regulator     = 0x           (null)
```

2. /sys/devices/soc.2/1c28000.uart/ctrl_info

此节点文件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。操作效果图：

```
/ # cat /sys/devices/soc.2/1c28000.uart/ctrl_info
 ier  : 0x05
 lcr  : 0x13
 mcr  : 0x03
 fcr  : 0xc1
 dll  : 0x0d
 dlh  : 0x00
 last baud : 115384 (dl = 13)

TxRx Statistics:
 tx     : 1237
 rx     : 70
 parity : 0
 frame  : 0
 overrun: 0
```

3. /sys/devices/soc.2/1c28000.uart/status

此节点文件可以打印出当前 UART 端口的一些运行状态信息，包括控制器的各寄存器值。操作效果图：

```
/ # cat /sys/devices/soc.2/1c28000.uart/status
uartclk = 24000000
The Uart controller register[Base: 0xffffff8000012000]:
[RTX] 0x00 = 0x00000041, [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 = 0x00000000
```

4. /sys/devices/soc.2/1c28000.uart/loopback

此节点文件是为了调试时方便设置 UART 控制器内部的环回模式，正常使用过程中环回模式没有实际意义。

该文件缺省值是 0，打开环回模式的方法是写入 1：

echo 1 > /sys/devices/soc.2/1c28000.uart/loopback

关闭的方法是写入 0：

echo 0 > /sys/devices/soc.2/1c28000.uart/loopback

也可以用 cat 命令查询当前的 loopback 状态：

```
/ # cat /sys/devices/soc.2/1c28000.uart/loopback
MCR: 0x00000003, Loopback: 0
```

Declaration