# 实验一环境搭建与鸿蒙LiteOS-a内核体验

1. 实验目的
2. 实验内容

## 1.1 内核里打印

内核打印函数是PRINT\_RELEASE，它的内部调用关系如下：

PRINT\_RELEASE  
 LOS\_LkPrint  
         g\_osLkHook  
                 OsLkDefaultFunc  
                         OsVprintf  
                                 UartPuts  
                                         UartPutsReg  
                                                 UartPutStr  
                                                         UartPutcReg

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

我们要实现UartPutcReg，用来输出单个字符。

## 1.2 APP控制台

我们编写的应用程序，调用printf时，那些信息从哪里打印出来？从**控制台**。 在串口上运行程序，**控制台**就是串口。  
 远程登录板子后运行程序，**控制台**就是远程登录终端。

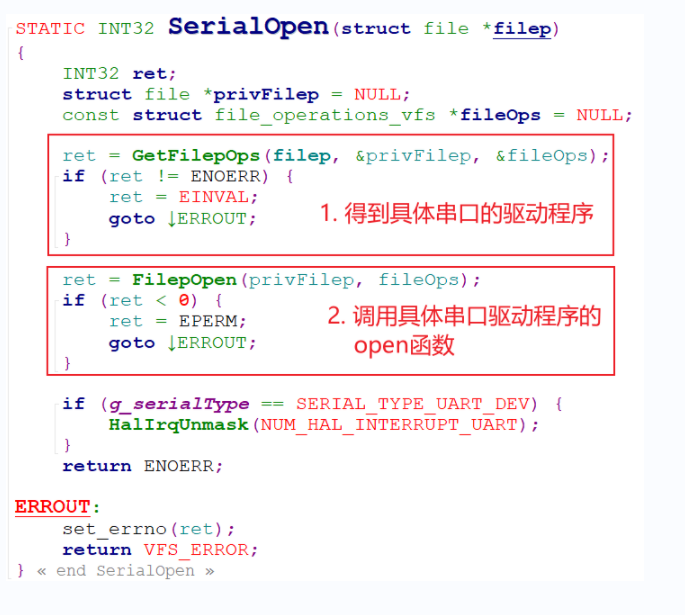
控制台的实现分为4层：

### 1.2.1 /dev/console

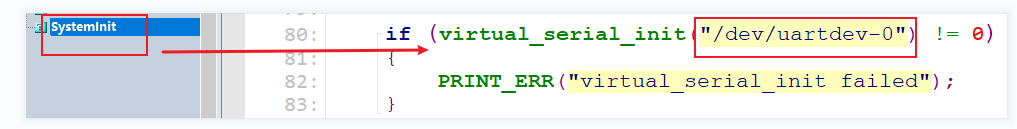
init进程打开的就是/dev/console，它会打开shell。  
 我们在shell里执行各种APP时，这些APP会继承父进程的3个设备：标准输入、标准输出、标准错误，都对应/dev/console。  
 我们编写的APP，一般不需要自己去打开/dev/console，它已经继承得到了。  
 在串口上运行程序，/dev/console就是串口。 远程登录板子后运行程序，/dev/console就是远程登录终端。  
 所以/dev/console表示的是当前终端，它可能对应不同的设备，比如/dev/serial或/dev/telnet。

### 1.2.2 /dev/serial

在Liteos-a中，/dev/serial被称为**virtual serial**，[虚拟串口](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E4%B8%B2%E5%8F%A3&spm=1001.2101.3001.7020)。它只是起一个中转的作用，无论是APP还是内核，使用/dev/serial时，都是再次跳转去执行具体串口设备驱动程序的函数。比如：



那么，/dev/serial这个虚拟串口，怎么跟具体串口挂钩？也就是上图中，GetFileOps函数为何能得到具体串口的驱动程序？  
 方法如下图所示：



virtual\_serial\_init函数会找到/dev/uartdev-0的驱动程序(即它对应的struct inode，里面含有file\_operations\_vfs)。

### 1.2.3 /dev/uartddev-0

#### *1. 总体介绍*

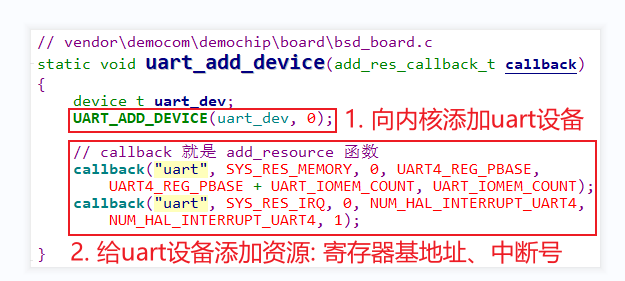
这是真正操作硬件的驱动程序，它分为两部分：device\_t、driver\_t。

* 在device\_t中设置**资源**，比如寄存器物理基地址、中断号等
* 在driver\_t中提供函数，比如device\_probe、device\_attach函数
* 当内核发现有名字系统的device\_t、driver\_t时
* 就会调用driver\_t中的device\_probe、device\_attach函数
* 在里面根据device\_t得到**资源**、注册驱动register\_driver

这种写驱动程序的方法，被称为**分离**：操作函数、资源分离。  
 以后想换一个硬件，只需要修改device\_t就可以，driver\_t保存不变。

#### *2. device\_t*

示例代码：



#### *3. drvier\_t*

先注册一个drvier\_t结构体，它里面带有各类device\_method\_t：



当内核发现有同名的device\_t和driver\_t时，就会调用driver\_t里面提供的device\_probe、device\_attach函数。

#### *4. uartdev\_fops*

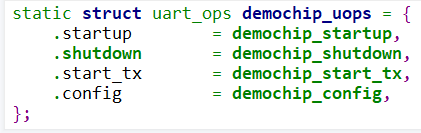
在device\_attach函数里从device\_t里获取硬件资源、注册驱动：



/dev/uartdev-0对应的驱动程序时uartdev\_fops，它通过uart\_ops来操作硬件。

### 1.2.4 uart\_ops

在UART驱动程序里，我们只需要提供硬件操作部分：



uart\_ops里有4个函数：

* config：配置串口，比如波特率等
* startup：启动串口，比如注册中断处理函数、启动串口
* start\_tx：发送字符串
* shutdown：关闭串口

串口就两大功能：发送数据、接收数据。  
 在Liteos-a中，发送数据比较简单：没有使用中断，而是使用查询方式逐个发送，核心是UartPutcReg。  
 接收数据时使用中断，所以需要注册**串口接收中断处理函数**，它要做的事情是：

* 发生中断时，读取硬件获得字符，可能有多个字符
* 处理特殊字符：比如不`\r`换为`\n`
* 通知上层代码：udd->recv(udd, buf, count);

# 2. 串口移植

我们的目标是：让最小系统启动。 那么对于串口，不需要考虑得很全面：

* 不需要初始化串口：u-boot已经初始化串口了
* 不需要动态配置串口：固定使用某个波特率等配置就可以(在u-boot里设置过了)

移植工作只需要实现这几点：

* 串口发送单个字符
* 注册串口接收中断函数：确定中断号、使能中断、在中断函数中读取数据

## 2.1 最终结果

本章节做的修改会制作为补丁文件：

* 03\_openharmony\_uart\_imx6ull.patch

假设目录openharmony中是未修改的代码，从没打过补丁；  
 假设补丁文件放在openharmony的同级目录； 打补丁方法如下：

$ cd openharmony  
$ patch -p1 < ../openharmony\_100ask\_v1.2.patch  
$ patch -p1 < ../01\_openharmony\_add\_demo\_board.patch  
$ patch -p1 < ../02\_openharmony\_memmap\_imx6ull.patch   
$ patch -p1 < ../03\_openharmony\_uart\_imx6ull.patch

* 1
* 2
* 3
* 4
* 5

打上补丁后，可以如此编译：

$ cd kernel/liteos\_a  
$ cp tools/build/config/debug/demochip\_clang.config .config  
$ make clean  
$ make

* 1
* 2
* 3
* 4

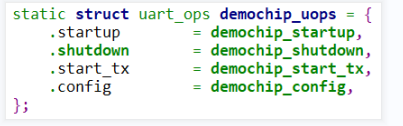
## 2.2 串口发送单个字符

## 2.3 在device\_t中指定资源

需要确定2个资源：寄存器地址、中断号

## 2.4 实现uart\_ops

在UART驱动程序里，uart\_ops结构体封装了UART的硬件操作：



uart\_ops里有4个函数：

* config：配置串口，比如波特率等
* startup：启动串口，比如注册中断处理函数、启动串口
* start\_tx：发送字符串
* shutdown：关闭串口

我们只需要实现startup、start\_tx，其他函数可以设为空：

* startup：确定中断号、request\_irq、使能中断、提供中断处理函数
* start\_tx：发送字符串

## 2.5 GIC

在kernel\liteos\_a\platform\main.c中，调用OsSystemInfo打印系统信息时，代码如下：

PRINT\_RELEASE("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Welcome\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n"  
 "Processor : %s"  
#if (LOSCFG\_KERNEL\_SMP == YES)  
 " \* %d\n"  
 "Run Mode : SMP\n"  
#else  
 "\n"  
 "Run Mode : UP\n"  
#endif  
 "GIC Rev : %s\n"  
 "build time : %s %s\n"  
 "Kernel : %s %d.%d.%d.%d/%s\n"  
 "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n",  
 LOS\_CpuInfo(),  
#if (LOSCFG\_KERNEL\_SMP == YES)  
 LOSCFG\_KERNEL\_SMP\_CORE\_NUM,  
#endif  
 HalIrqVersion(), \_\_DATE\_\_, \_\_TIME\_\_,\  
 KERNEL\_NAME, KERNEL\_MAJOR, KERNEL\_MINOR, KERNEL\_PATCH, KERNEL\_ITRE, buildType);



* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

里面的HalIrqVersion函数用到的GIC的虚拟地址，要正确设置，否则没有打印信息。  
 IMX6ULL的内存映射代码里，设备空间从GIC开始映射，所以GIC的虚拟地址就是PERIPH\_DEVICE\_BASE：

// kernel/liteos\_a/kernel/base/include/los\_vm\_zone.h  
#define GIC\_VIRT\_BASE PERIPH\_DEVICE\_BASE  
  
// vendor/democom/demochip/board/include/asm/platform.h  
#define GIC\_BASE\_ADDR (GIC\_VIRT\_BASE)

* 1
* 2
* 3
* 4
* 5