

# 浅谈WLS库

## WLS能干啥

wls(Wireless shared lib)是Intel为了基站MAC层和PHY层IPC开发的c语言库，其原理主要是基于共享内存，跟其他共享内存的使用方法一样，你需要将数据复制到共享内存区域，然后接收进程就可以收到

## 精通WLS的安装与卸载

### 安装

按照wls库源码中的readme文件，安装很简单的，只需执行以下几个步骤：

```
1. cd wls_mod
2. wls_mod$ make clean
3. wls_mod$ ./build.sh
4. wls_mod$ sudo make install
```

编译完成后，会生成一个libwls.so文件和一个wls.ko文件，so库可以复制到/usr/lib64下使用也可以在启动程序脚本中指定 LD\_LIBRARY\_PATH 来设定wls库的链接路径

ko文件需要加载进入内核执行：（或者嫌麻烦直接执行wls源码里的脚本wls\_start.sh）

```
sudo modprobe wls wlsMaxClient = 4(可选项，默认为1)
```

或者在wls源码的根目录执行：

```
insmod ./wls.ko wlsMaxClient = 4
```

如果安装成功，会在/dev文件夹下生成一个设备wls[0]，wls将自身的IO挂载为一个字符设备，并针对该设备写了一个驱动，注册进入内核，这也是为什么编译生成ko文件，需要安装进入内核的原因

如果想卸载该模块请执行(两种方法都可以)：

```
sudo modprobe -r wls
rmmod wls
```

### 运行

wls库运行有两个模式：MASTER模式和SLAVE模式，MASTER模式可以管理SLAVE模式下的共享内存，SLAVE模式只能使用MASTER模式管理的共享内存，无权管理

启动很简单，如果你只是想运行wls源码中自带的test例子，执行wls源码根目录下wls\_test.sh脚本就可以了，用于用于进程间通信，所以最少启动两个进程：

```
wls_mod#./wls_test.sh -a 1 -w /dev/wls0 -m &
wls_mod#./wls_test.sh -a 1 -w /dev/wls0
```

上述命令-a参数用于设置cpu亲和性，-w用于指定用于指定打开哪个设备，-m指定以MASTER模式运行，如果不指定则以SLAVE模式运行，如果运行成功，就会看见两个进程会互相收发消息（测试代码里写的是1ms产生10条消息，大小不固定）

如果运行4个小区：

```
wls_mod#./wls_test.sh -a 1 -w /dev/wls0 -m &
wls_mod#./wls_test.sh -a 1 -w /dev/wls0

for four cells simulation
wls_mod#./wls_test.sh -a 1 -w /dev/wls0 -m &
wls_mod#./wls_test.sh -a 1 -w /dev/wls0

wls_mod#./wls_test.sh -a 2 -w /dev/wls1 -m &
wls_mod#./wls_test.sh -a 2 -w /dev/wls1

wls_mod#./wls_test.sh -a 3 -w /dev/wls2 -m &
wls_mod#./wls_test.sh -a 3 -w /dev/wls2

wls_mod#./wls_test.sh -a 4 -w /dev/wls3 -m &
wls_mod#./wls_test.sh -a 4 -w /dev/wls3
```

如果不满足只玩弄一下wls的测试例子，需要在自己的程序中使用wls，那么你需要将wls库的头文件包含在自己的工程中，然后在makefile中只用-L选项指定libwls.so的链接路径，然后编写对应的启动脚本，其实具体的启动脚本在wls源码路径中的 `wls_test.sh` 已经很好的说明了，只需将脚本第一行 `wlsTestBinary="wls_test"` 改为你自己的可执行文件就可以了
















**注意：**

1. 编译wls库时请确认系统内已安装libhugetlbfs库
2. 启动wls程序时，请确认hugepage是否还有剩余，使用命令`cat /proc/meminfo`
3. 如果在调试wls程序时，SLAVE模式连接不上MASTER模式的程序，请重新挂载wls模块
4. 不要想着能不能一个MASTER模式与多个SLAVE模式连接，

# 做一个合格的调包侠

---

观察wls源码目录结构:

	build.sh
	pool.c
	pool.h
	syslib.c
	syslib.h
	testapp.c
	ttypes.h
	wls_debug.h
	wls_drv.c
	wls_drv.h
	wls_lib.c
	wls_lib.h
	wls_start.sh
	wls_test.sh
	wls.h

其中，**pool.h**是wls自己提供的共享内存管理头文件，原理是将共享内存实现为一个环形队列，如果觉得实现的不好，也可以不用

如果只是使用wls库的API，只需要wls\_lib.h就可以了，主要API如下：

```

//用于打开wls设备也就是/dev/wls[i],mode指定是MASTER还是SLAVE模式打开
void* WLS_Open(const char *ifacename, unsigned int mode);

//用于关闭wls设备
int WLS_Close(void* h);

//该函数为MASTER模式和SLAVE模式，MASTER模式将在分块的共享内存中，分配一块，然后加入
//到SLAVE模式内存块中，SLAVE模式则是在自己的共享内存块中取出一块
void* WLS_Alloc(void* h, unsigned int size);

//MASTER模式负责归还内存块，SLAVE模式不能使用，内存由MASTER模式帮助释放
int WLS_Free(void* h, void* pMsg);

//向对端发送消息，该函数的第二个参数为一个物理地址，需要使用WLS_VA2PA进行转换，flags表示是
//否进行分段，msgtype为消息类型，你可以用他过滤某些消息类型
int WLS_Put(void* h, unsigned long long pMsg, unsigned int MsgSize, unsigned short
    MsgTypeID, unsigned short Flags);

//收取对端的消息，通常该API需要与WLS_Wait结合使用，注意该函数的返回值也是一个物理地址，需要
//转换为虚拟地址
unsigned long long WLS_Get(void* h, unsigned int *MsgSize, unsigned short *MsgType
    ID, unsigned short *Flags);

//阻塞线程直到有数据到来
int WLS_Wait(void* h);

unsigned long long WLS_VA2PA(void* h, void* pMsg);

void* WLS_PA2VA(void* h, unsigned long long pMsg);

//MASTER模式用于将内存块加入到对端SLAVE模式的共享内存块中，SLAVE模式不会用到，pMSG是物理地址
int WLS_EnqueueBlock(void* h, unsigned long long pMsg);

//SLAVE模式用于从共享内存块中取出一块，然后返回该快的物理地址
unsigned long long WLS_DequeueBlock(void* h);

//查看有多少个可用的共享内存块
int WLS_NumBlocks(void* h);

```

综上所述，MASTER模式需要这样使用WLS库:

```

wls_open(path,MASTER) //打开设备, 将该设备映射到
wls_alloc(...) //开辟共享内存, 此时等待SLAVE的加入
memory_init(...)//管理之前分配的共享内存, 这时你可以使用自己的方式管理

while(true):
    wls_wait(...) //等待数据的到来
    wls_get(...) //获得该消息
    deal msg...
    wls_close(...)
    something should be sent
    wls_alloc(...) //分配内存
    wls_put(...) //发送数据

```

SLAVE模式这样使用:

```

wls_open(path,SLAVE) //...
wls_alloc(...)//SLAVE模式下该函数只是初始化了hugepage库, 没做其他的就返回了

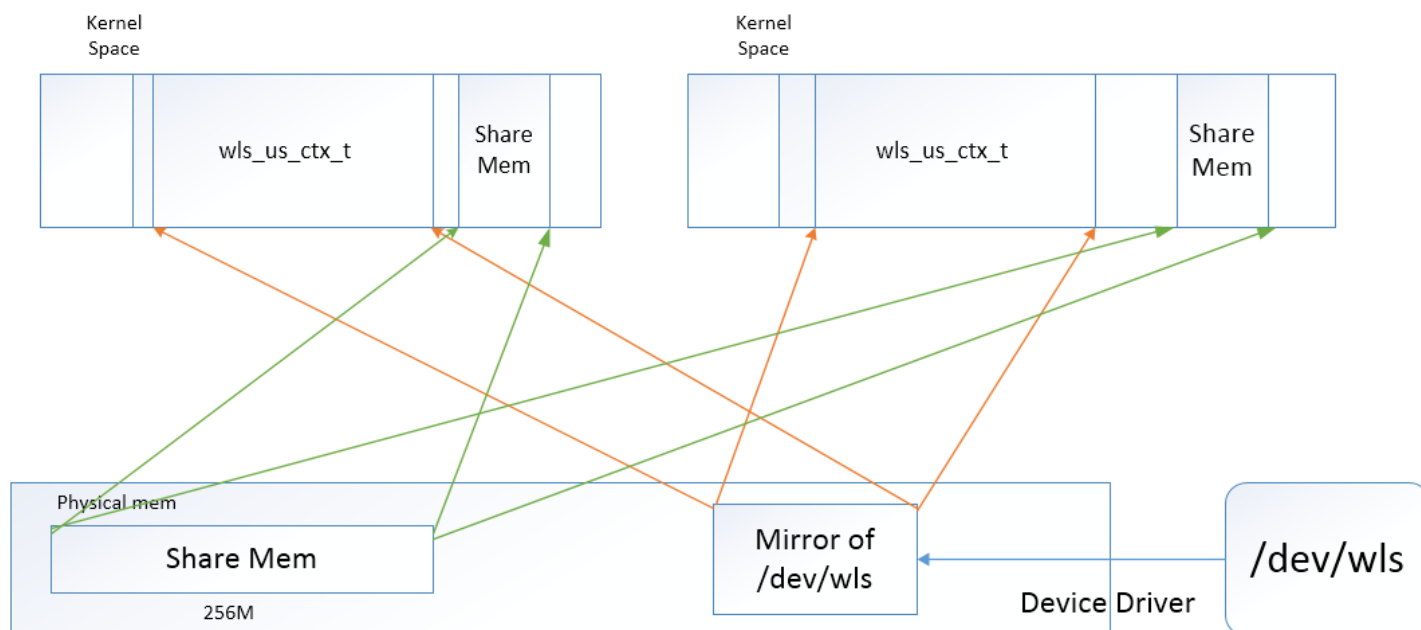
while(true):
    wls_wait(...) //等待数据的到来
    wls_get(...) //获得该消息
    deal msg...
    wls_close(...)
    something should be sent
    wls_alloc(...) //分配内存
    wls_put(...) //发送数据

```

**注意:**

1. WLS库有强烈的主从观念, SLAVE无法自己释放内存, 原因是因为SLAVE模式下无法管理共享内存, 它不知道共享内存是以何种方式组织的, 只能去内存块中取一块, MASTER模式在收到SLAVE模式发送过来的数据以后, 会替SLAVE模式释放内存
2. 由于SLAVE模式无法释放内存, 那么MASTER模式发送给SLAVE模式的数据也必须有MASTER模式释放, 针对LTE环境, MASTER模式会在发送数据10ms之后释放, 这期间必须足够SLAVE模式处理该数据, 否则会造成数据丢失

下面结合具体例子分析, 详情见测试例子代码,该过程在内存中的布局可以用一幅图表示:



## 从内核的角度看WLS

1. MATER是如何得到SLAVE的 `dst_pa` 的?
2. 为什么在调用 `wls_put` 函数后, 命名是放put队列中放入, 对端却在get队列中获得数据?
3. `wls_wait` 函数是如何实现阻塞线程的?
4. 如何实现物理地址和虚拟地址之间的转换?

上述所有问题都可以在 `wls_drv.c` 这个源文件中找到答案, 上文说过wls构造出一个字符设备, 然后针对该设备编写了设备驱动, 生成ko文件, 注册进入内核, 在源文件中对应的代码如下:

```
MODULE_DESCRIPTION("Wireless System Communication interface");
MODULE_AUTHOR("Intel Corporation");
MODULE_LICENSE("GPL v2");
MODULE_VERSION("WLS_DRV_VERSION_FORMAT");

module_init(wls_init);
module_exit(wls_exit);
```

其中init可以理解为用户态的main函数, 该内核模块可以有参数

```
/*
 * Module Parameters
 */
int wlsMaxClients = 1;

module_param(wlsMaxClients, int, S_IRUSR);
```

现在知道挂载wls模块时的wlsMaxClients参数是做什么的了把

此外，由于该驱动管理一个字符设备，所以有一个很关键的结构体：

```
static struct file_operations wls_fops = {
    .owner = THIS_MODULE,
    .open = wls_open,
    .release = wls_release,
    .unlocked_ioctl = wls_ioctl,
    .compat_ioctl = wls_ioctl,
    .mmap = wls_mmap,
};
```

上述结构体将对字符设备对应的操作与对应的函数指针赋值，然年在内核对该字符设备执行对应的操作时，就会进入到该内核模块自己写的函数中处理，该结构体初始化的时机在 `wls_init` 函数中：

```
cdev_init(&wls_dev_loc->cdev, &wls_fops);
```

这样在用户调用ioctl函数时，就会执行wls\_ioctl函数，现在回答第一个问题，用户执行ioctl(fd,open,&param)，该内核模块会为用户在内核空间创建上下文，包括get队列，put队列等等，在创建上下文的函数中，有这样一段代码：

```

if(pDrv_ctx->p_wls_us_ctx[0] && pDrv_ctx->p_wls_us_ctx[1])
{
    //link ctx
    pDrv_ctx->p_wls_us_ctx[0]->dst_kernel_va = (uint64_t)pDrv_ctx->p_wls_us_ctx[1];
    pDrv_ctx->p_wls_us_ctx[0]->dst_pa = (uint64_t) pDrv_ctx->p_wls_us_p
a_ctx[1];

    pDrv_ctx->p_wls_us_ctx[1]->dst_kernel_va = (uint64_t)pDrv_ctx->p_wls_us_ctx[0];
    pDrv_ctx->p_wls_us_ctx[1]->dst_pa = (uint64_t) pDrv_ctx->p_wls_us_p
a_ctx[0];

    pDrv_ctx->p_wls_us_ctx[0]->dst_kernel_va = (uint64_t)pDrv_ctx->p_wls_us_ctx[1];
    pDrv_ctx->p_wls_us_ctx[0]->dst_pa = (uint64_t) pDrv_ctx->p_wls_us_p
a_ctx[1];
    pDrv_ctx->p_wls_us_ctx[1]->dst_kernel_va = (uint64_t)pDrv_ctx->p_wls_us_ctx[0];
    pDrv_ctx->p_wls_us_ctx[1]->dst_pa = (uint64_t) pDrv_ctx->p_wls_us_p
a_ctx[0];

    WLS_DEBUG("link: 0 <-> 1: 0: 0x%016lx 1: 0x%016lx\n", (long unsigned int)p
Drv_ctx->p_wls_us_ctx[0]->dst_kernel_va,
                                                    (long unsigned int)p
Drv_ctx->p_wls_us_ctx[1]->dst_kernel_va);
}

```

这段代码解释了一个设备最多只能有两个进程打开，且内核分别将各自的进程 `dst_pa` 设置为对端的物理地址

**问题2**可以在 `wls_drv.c` 的 `wls_put` 和 `wls_wait` 中找到答案，因为在内核中put函数将msg从发送端的put队列中取出，然后放在了对端的get队列中，然后唤醒阻塞的进程，阻塞的进程被唤醒后，就可以从自己的get队列中取到

**问题3**的答案间如下代码:



```

static int wls_wait(wls_sema_priv_t *priv, unsigned long arg)
{
    char __user *buf = (char __user *)arg;

    if (!likely(atomic_read(&priv->is_irq))) {
        if (unlikely(wait_event_interruptible(priv->queue, atomic_read(&priv->is_irq)))) {
            return -ERESTARTSYS;
        }
    }

    atomic_dec(&priv->is_irq);

    if (priv->drv_block_put != priv->drv_block_get) {
        unsigned int get = priv->drv_block_get + 1;

        if (get >= FIFO_LEN)
            get = 0;

        if (copy_to_user(buf, &priv->drv_block[get], sizeof(wls_wait_req_t))) {
            return -EFAULT;
        }

        priv->drv_block_get = get;

#ifdef DEBUG
        printk(KERN_INFO "[wls]:GET: put=%d get=%d T=%lu is_irq=%d\n",
            priv->drv_block_put, priv->drv_block_get,
            priv->drv_block[get].start_time, atomic_read(&priv->is_irq));
#endif /* DEBUG */

    } else {
#ifdef DEBUG
        printk(KERN_ERR "[wls]: wrong computation of queueing\n");
#endif /* DEBUG */
    }

    return 0;
}

```

`wait_event_interruptible` 是一个内核态函数，它的第二个参数是一个条件，作用是如果条件不满足就将该进程置为TASK\_INTERRUPTIBLE状态，此时CPU不会再调度该进程，直到有人唤醒该进程

第四个问题跟库有关，具体代码如下：

```

static int wls_VirtToPhys(void* virtAddr, uint64_t* physAddr)
{
    int          mapFd;
    uint64_t     page;
    unsigned int  pageSize;
    unsigned long virtualPageNumber;

    mapFd = open ( "/proc/self/pagemap" , O_RDONLY );
    if (mapFd < 0 )
    {
        PLIB_ERR("Could't open pagemap file\n");
        return -1;
    }

    /*get standard page size*/
    pageSize = getpagesize();

    virtualPageNumber = (unsigned long) virtAddr / pageSize ;

    lseek(mapFd , virtualPageNumber * sizeof(uint64_t) , SEEK_SET );

    if(read(mapFd , &page , sizeof(uint64_t)) < 0 )
    {
        close(mapFd);
        PLIB_ERR("Could't read pagemap file\n");
        return -1;
    }

    *physAddr = (( page & 0x007fffffffffffffffULL ) * pageSize );

    close(mapFd);

    return 0;
}

```

/proc/self/pagemap是linux从2.6版本开始提供给用户态的接口，该文件由内核来维护，用户态只可读，文件的结构是一个个8字节的数字，内容为虚拟页号对应的物理页号，该段代码算出虚拟页号以后，去该文件查找对应的物理页号，然后乘页的大小，就可以得到物理地址

在之后写的 `WLS_VA2PA` 函数中可以由虚拟地址算出对应的hugepage号，然后再加上页内偏移量，就可以得到物理地址，反之亦然