# 通过WLAN按键关闭再打开无线后不能登录单板故障查证记录

【作者】　　刘振华。

【时间】　　2013.1.11

【概述】　故障复现过程：无线的配置为4个SSID同时开启，通过WLAN按键关闭无线，再通过WLAN按键打开无线，此时通过web页面不能登录单板，可以ping通单板，WAN连接正常。故障必现。

## 故障描述

1. H168NWV1.0，H168NW V1.0.0T1，基于csp平台，613001849390，【H168NW V1.0.0T1】WLAN硬件关闭再开启WLAN功能，STA无法连接单板无线，LAN侧PC无法访问单板的WEB页面。
2. **问题单描述**

测试描述：

【测试环境】

武汉研究所3楼固网实验室1，

测试版本：H168NW V1.0.0T1

版本：#2 SMP PREEMPT Wed Jan 9 23:45:51 CST 2013

上行方式：ADSL

浏览器：IE8.0，Chrome　OS:WIN7，WINXP

电源：Output：12V--2A

【测试步骤】

1.升级版本，恢复默认出厂，然后配置无线无g+n模式，开启SSID1~4

2.测试测试WLAN硬件开关，先硬件关闭WLAN，然后再硬件开启WLAN

【现象描述】

1.LAN侧PC访问web页面，发现PC无法访问单板的web管理页面，

2.LAN侧PC上抓包，发现PC发出了HTTP GET，但是单板无回应

3.LAN侧PC此时可以ping通单板192.168.1.1，可以访问WAN侧业务

4.STA连接单板的SSID，但是无法正常连接到单板

5.此时查看串口，发现打印一直停在WLAN接口注册的位置

6.手动在串口中点击回车，WLAN接口才可以正常注册，此时STA才能正常访问单板WEB页面，STA此时也可以成功连接单板的无线

7.配置和LOG如附件所示

H168NW_T1_0111_1338.zip

【故障确认人】刘振华，李瑞，陈振

刘振华：

故障出现后，通过用另外一块单板复现故障，故障为必现。

故障出现时，所有的内核态的打印可以正常输出，但是用户态的打印不能输出。利用魔键发现是接口栈进程在进行日志输出时占用了tty_write_lock的锁，访问页面时web进程也需要输出日志，这样获取不到锁而导致。故障出现时内核是正常的。输入回车先前的打印正常输出后，单板恢复正常。

## 查证过程

**日期（2012014）**

通过多次进行故障的复现，发现当无线启动向接口栈注册打印日志的过程中正好有大量的内核打印出现，故障就会出现，如果内核的打印稍微快一些或者慢一些故障出现的概率就下降。相关日志如下：红色为内核的打印，其他为用户态的打印

1193:01:00 [wlan_config][Info] [wlan_adapter_br(2825)brcm_wlan_wps_i] nvram_set lan_ifname br0

nvram_set:name is wl0_ifname

wps_led_update:val=0,wps_prevstatus=-1

1193:01:02 [ifsGID000000007>1193:01:02 [KDEVCOMM][Notice] [CSPCommon.h(139)SetIfDevcieType] setifdevicetype parameter WLANBIT!

cmd = 17, paras 0x7ebff630

cmd = 22, par_mgr][Info] [ifas 0x7ebff630

cmd = 25, paras 0x7ebff630

<LOs_if.c(1282)hanGID000000007>1193:01:02 [KDEVCOMM][Notice] [CSPCommon.h(139)SetIfDevcieType] setifdevicetype parameter WLANBIT!

cdleSetIFStat] Smd = 17, paras 0x7ebff630

cmdet IF[IGD.LD1.W = 22, paras 0x7ebff630

cmd = 25, paras 0x7ebff630

GID000000007>1193:01:02 [KDEVCOMM][Notice] [CSPCommon.h(139)SetIfDevcieType] setifdevicetype parameter WLANBIT!

<LOGID000000007>1193:01:02 [KDEVCOMM][Notice] [CSPCommon.h(139)SetIfDevcieType] setifdevicetype parameter WLANBIT!

怀疑有两点：

1. 故障可能与接口栈模块有关
2. 故障可能与内核态的打印较多有关

**日期（20130115）**

1. 打开接口栈的调试打印，进行故障的复现，多次测试后发现，故障没有出现
2. 关闭内核态的打印，进行故障复现，多次测试后发现故障没有出现

这样可以推断该故障应该与接口栈没有直接的关系，应该是与日志的打印有较大的关系

用H168NV2.0的版本进行测试，同样存在这样的故障

通过加打印跟踪故障，故障出现时的逻辑如下：

do_syslog-> OssDoSysLog-> LogStdioProcOutput-> CspLogOut-> vfs_write-> tty_write-> do_tty_write ->n_tty_write,

do_tty_write的大体实现如下：

```
static inline ssize_t do_tty_write(
{

    ret = tty_write_lock(tty, file->f_flags & O_NDELAY);

    /* Do the write .. */
    for (;;) {
        size_t size = count;
        if (size > chunk)
            size = chunk;
        ret = -EFAULT;
        if (copy_from_user(tty->write_buf, buf, size))
            break;
        ret = write(tty, file, tty->write_buf, size);
        if (ret <= 0)
            break;
        written += ret;
        buf += ret;
        count -= ret;
        if (!count)
            break;
        ret = -ERESTARTSYS;
        if (signal_pending(current))
            break;
        cond_resched();
    }
    if (written) {
        struct inode *inode = file->f_path.dentry->d_inode;
        inode->i_mtime = current_fs_time(inode->i_sb);
        ret = written;
    }
out:
    tty_write_unlock(tty);
    return ret;
}
```

在日志输出时，首先会获取tty_write的锁，获取后通过调用挂的write函数n_tty_write函数将日志进行输出，输出后再释放该锁。n_tty_write函数的实现如下：

```
static ssize_t n_tty_write(struct tty_struct *tty, struct file *file,
                           const unsigned char *buf, size_t nr)
{
    /* Job control check -- must be done at start (POSIX.1 7.1.1.4). */
    if (L_TOSTOP(tty) && file->f_op->write != redirected_tty_write) {
        retval = tty_check_change(tty);
        if (retval)
            return retval;
    }

    /* Write out any echoed characters that are still pending */
    process_echoes(tty);

    add_wait_queue(&tty->write_wait, &wait);
    while (1) {
        set_current_state(TASK_INTERRUPTIBLE);
        if (signal_pending(current)) {
            retval = -ERESTARTSYS;
            break;
        }
        if (tty_hung_up_p(file) || (tty->link && !tty->link->count)) {
            retval = -EIO;
            break;
        }
        if (O_OPOST(tty) && !(test_bit(TTY_HW_COOK_OUT, &tty->flags))) {
            while (nr > 0) {
                ssize_t num = process_output_block(tty, b, nr);
                if (num < 0) {
                    if (num == -EAGAIN)
                        break;
                    retval = num;
                    goto break_out;
                }
                b += num;
                nr -= num;
                if (nr == 0)
                    break;
                c = *b;
                if (process_output(c, tty) < 0)
                  {
                    break;
                  }
                b++; nr--;
            }
            if (tty->ops->flush_chars)
                tty->ops->flush_chars(tty);
        }
    if (!nr)
            break;
        if (file->f_flags & O_NONBLOCK) {
```

```
                retval = -EAGAIN;
                break;
            }
            schedule();
        }
break_out:
        __set_current_state(TASK_RUNNING);
        remove_wait_queue(&tty->write_wait, &wait);
        if (b - buf != nr && tty->fasync)
            set_bit(TTY_DO_WRITE_WAKEUP, &tty->flags);
        return (b - buf) ? b - buf : retval;
}
```

该函数会将调用进程设置为可中断的睡眠状态，然后调用process_output函数将打印的字符保存到tty的缓存区中，然后调用tty挂的flush_chars函数将字符一个个打印出去。此时如果将tty缓存区中的字符都打印出去了，就会走正常的流程退出，但是如果不满足条件将会调用schedule函数将进程调度走出现占用锁的情况。而process_output函数的实现如下：

```
static int process_output(unsigned char c, struct tty_struct *tty)
{
        int     space, retval;
        mutex_lock(&tty->output_lock);
        space = tty_write_room(tty);
        retval = do_output_char(c, tty, space);
        mutex_unlock(&tty->output_lock);
        if (retval < 0)
            return -1;
        else
            return 0;
}
```

这个函数中会首先判断tty缓存区中是否有空隙的room，如果有的话就放字符，如果没有就返回-1，故障出现时返回-1.如上所述在这个函数退出时会调用flush_chars函数（最后调用bcm63xx_tx_chars）将缓存区中的字符打出去，但是故障出现时没有任何打印。

bcm63xx_tx_chars函数的实现如下：

```
static void bcm63xx_tx_chars(struct uart_port *port)
{
    struct circ_buf *xmit = &port->info->xmit;
    (UART_REG(port))->intMask &= ~TXFIFOTHOLD;
    if (port->x_char) {
        while (!((UART_REG(port))->intStatus & TXFIFOEMT));
        /* Send character */
        (UART_REG(port))->Data = port->x_char;
        port->icount.tx++;
        port->x_char = 0;
        return;
    }
    if (uart_circ_empty(xmit) || uart_tx_stopped(port)) {
        bcm63xx_stop_tx(port);
        return;
    }
    while ((UART_REG(port))->txf_levl < port->fifosize) {
        (UART_REG(port))->Data = xmit->buf[xmit->tail];
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
        if (uart_circ_empty(xmit))
            break;
```

```
    }
    if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
        uart_write_wakeup(port);
    if (uart_circ_empty(xmit))
        {
        printk("end\n");
        bcm63xx_stop_tx(port);
        }
    else
        (UART_REG(port))->intMask |= TXFIFOTHOLD;
}
```

通过上述代码来看貌似是红色部分的条件不满足，所以字符没输出，而缓存区中的字符也超过了WAKEUP_CHARS)没有唤醒，进程不能进行继续的打印而导致锁资源竞争，故障出现。

由于在串口输入的正常分支中加打印会导致串口挂死，所以故障分析到这里无法再继续跟踪下去。另一个想法是跟踪一下为什么键盘输入一个字符故障就会恢复，但是按魔键故障不恢复

**日期（20130117）**

通过加打印及走查串口输入的流程发现：魔键的输入和正常的输入不一致，魔键的操作会进入到其他流程代码里，暂不分析。继续跟踪正常的串口输入，如下：

bcm63xx_int-> bcm63xx_rx_chars-> tty_flip_buffer_push-> flush_to_ldisc

-> n_tty_receive_buf-> flush_chars

Flush_chars（bcm63xx_tx_chars）的时候又将先前未打印完的字符输出后，单板才恢复了正常。

从以上的分析看故障出现是由于在正常打印用户态日志时被内核的打印抢占，但是内核抢占结束后由于某种原因导致串口输出条件不满足不能输出同时不能唤醒进程，但是在键盘输入时需要显示到串口上时又将先前缓存区中的打印输出唤醒了进程，所以故障恢复，单板正常。

从现在的情况看故障的原因可能是由于内核printk的抢占导致正常日志的输出有了问题，

下一步的想法是准备看如何能通过一种手段将故障出现时串口输入的相关信息获取，或者进一步梳理代码，看是否正的与printk的打印有关系，正常情况下是不允许过多的printk输出的。

**日期（xxxx-xx-xx，最后一天）**

格式相对灵活，必要信息有：昨天安排实验的故障分析（若有安排实验必写，附上抓到的log），代码走查结论（若有则必写），头脑风暴讨论结果（包括相关人，讨论信息，疑难故障需要讨论），找到最终问题，实验证明，描述清楚。

## 故障总结

可以按照技术积累的要求来写，以下元素须包含：

故障现象

故障分析和查证过程，导出故障原因。

故障根因：引入人、引入时间、故障引入原因

故障影响范围，临时规避手段

经验教训