PHY 状态简介

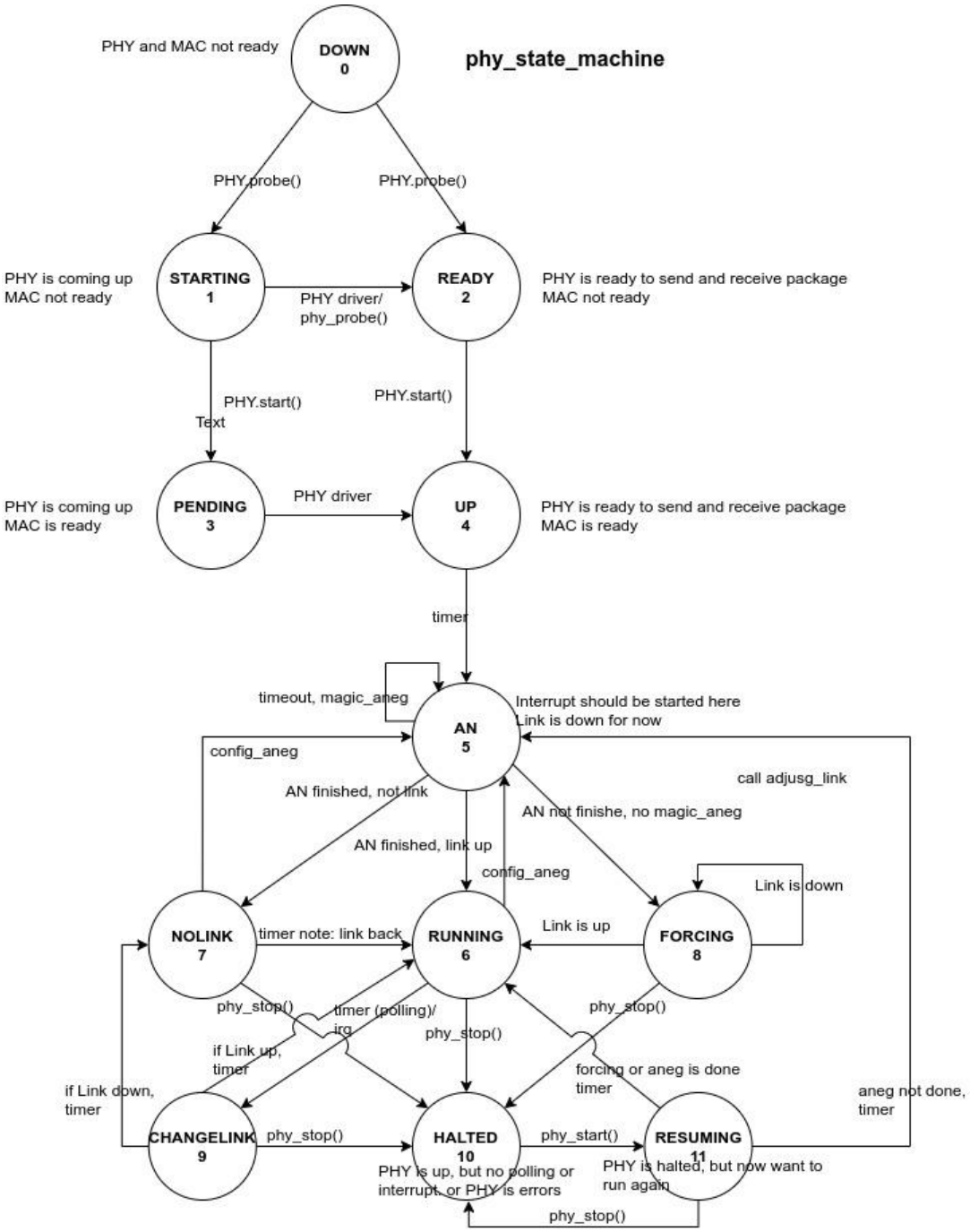
作者：stepforwards
原文链接：https://ld246.com/article/1638787517973
来源网站：链滴

PHY的12种状态

```
enum phy_state {
    PHY_DOWN = 0, //关闭网卡
    PHY_STARTING, //PHY设备准备好了，PHY driver尚为准备好
    PHY_READY,      //PHY设备注册成功
    PHY_PENDING,  //PHY芯片挂起
    PHY_UP,           //开启网卡
    PHY_AN,           //网卡自协商
    PHY_RUNNING, //网卡已经插入网线并建立物理连接，该状态可切换到PHY_CHANGELINK
    PHY_NOLINK,    //断网，拔掉网线
    PHY_FORCING,//自动协商失败，强制处理（读phy状态寄存器，设置速率，设置工作模式）
    PHY_CHANGELINK, //LINK检查，当物理连接存在时切换到PHY_RUNING，物理连接不存在时切
到PHY_NOLINK
    PHY_HALTED,  //网卡关闭时，PHY挂起
    PHY_RESUMING //网卡开启时，PHY恢复
};
```

PHY状态机

原文链接：PHY 状态简介



PHY指PHY芯片，负责数据传送与接收所需要的电与光信号、线路状态、时钟基准、数据编码和电路等并向数据链路层设备提供标准接口。

MAC指MAC芯片，属于数据链路层，提供寻址机构、数据帧的构建、数据差错检查、传送控制、向络层提供标准的数据接口等功能。

PHY_DOWN: phy、phy driver、mac都没准备好

1. 如果phy driver被集成在内核中，PHY.probe后，phydev状态为PHY_READY。
2. 如果phy driver被未集成在内核中，PHY.probe后，phydev状态为PHY_STARTING。

PHY_READY：phy、phy driver已经就绪，mac未准备好

当MAC层加载时，在PHY.start后，phydev状态切换为PHY_UP。

PHY_STARTING：phy准备就绪，phy driver、mac未准备好

1. 当MAC加载时，PHY.start后，phydev状态为PHY_PENDING。
2. 当phy driver加载时，phydev状态为PHY_READY。

PHY_PENDING：phy、mac准备就绪,phy driver未准备好
当phy dirver加载后，phdev状态为PHY_UP

上图中0-->1-->2-->4、0-->2-->4代表phy、phy dirver、mac顺序加载。
0-->1-->3-->4代表phy、mac、phy driver顺序加载。

PHY_UP：phy、phy driver、mac准备就绪
当前状态将启动自动协商，若启动成功则进入PHY_AN，若启动失败则进入PHY_FORCING。

PHY_AN：网卡自协商模式，检测自协商是否完成。
先判断物理链路的状态，如果未LINK则进入PHY_NOLINK，如果LINK则判断自协商是否完成，
自协商完成进入PHY_RUNNING，若自协商超时则重新开启自协商。

PHY_FORCING：强制协商
读link和自协商状态寄存器，如果状态正常则进入PHY_RUNNING模式。

PHY_NOLINK：物理链路未连接
判断物理链路状态，如果LINK，再判断是否支持自协商，若支持待自协商完成后进入PHY_RUNNIN模式，
若不支持，直接进入PHY_RUNNING模式。若自协商处于挂起状态，则进入PHY_AN模式。

PHY_RUNNING：正常运行中
获取当前link状态，当link状态发生改变时，进入PHY_CHANGELINK模式。

PHY_CHANGELINK：检查物理链路
物理链路link时，切换到PHY_RUNNING，非LINK时切换到PHY_NOLINK。

PHY_HALTED：网卡关闭phy_stop
挂起phy
PHY_RESUMING: 网卡启用phy_start
恢复phy

phy_state_machine是PHY的状态机函数

```
/**
 * phy_state_machine - Handle the state machine
 * @work: work_struct that describes the work to be done
 */
void phy_state_machine(struct work_struct *work)
{
    struct delayed_work *dwork = to_delayed_work(work);
    struct phy_device *phydev =
            container_of(dwork, struct phy_device, state_queue);
    bool needs_aneg = false, do_suspend = false;
    enum phy_state old_state;
```

```
int err = 0;
int old_link;

mutex_lock(&phydev->lock);

old_state = phydev->state;

if (phydev->drv->link_change_notify)
    phydev->drv->link_change_notify(phydev);

switch (phydev->state) {
case PHY_DOWN:
case PHY_STARTING:
case PHY_READY:
case PHY_PENDING:
    break;
case PHY_UP:
    needs_aneg = true;

    phydev->link_timeout = PHY_AN_TIMEOUT;

    break;
case PHY_AN:
    err = phy_read_status(phydev);
    if (err < 0)
        break;

    /* If the link is down, give up on negotiation for now */
    if (!phydev->link) {
        phydev->state = PHY_NOLINK;
        netif_carrier_off(phydev->attached_dev);
        phydev->adjust_link(phydev->attached_dev);
        break;
    }

    /* Check if negotiation is done.  Break if there's an error */
    err = phy_aneg_done(phydev);
    if (err < 0)
        break;

    /* If AN is done, we're running */
    if (err > 0) {
        phydev->state = PHY_RUNNING;
        netif_carrier_on(phydev->attached_dev);
        phydev->adjust_link(phydev->attached_dev);

    } else if (0 == phydev->link_timeout--)
        needs_aneg = true;
    break;
case PHY_NOLINK:
    if (phy_interrupt_is_valid(phydev))
        break;

    err = phy_read_status(phydev);
```

```
    if (err)
        break;

    if (phydev->link) {
        if (AUTONEG_ENABLE == phydev->autoneg) {
            err = phy_aneg_done(phydev);
            if (err < 0)
                break;

            if (!err) {
                phydev->state = PHY_AN;
                phydev->link_timeout = PHY_AN_TIMEOUT;
                break;
            }
        }
        phydev->state = PHY_RUNNING;
        netif_carrier_on(phydev->attached_dev);
        phydev->adjust_link(phydev->attached_dev);
    }
    break;
case PHY_FORCING:
    err = genphy_update_link(phydev);
    if (err)
        break;

    if (phydev->link) {
        phydev->state = PHY_RUNNING;
        netif_carrier_on(phydev->attached_dev);
    } else {
        if (0 == phydev->link_timeout--)
            needs_aneg = true;
    }

    phydev->adjust_link(phydev->attached_dev);
    break;
case PHY_RUNNING:
    /* Only register a CHANGE if we are polling or ignoring
     * interrupts and link changed since latest checking.
     */
    if (!phy_interrupt_is_valid(phydev)) {
        old_link = phydev->link;
        err = phy_read_status(phydev);
        if (err)
            break;

        if (old_link != phydev->link)
            phydev->state = PHY_CHANGELINK;
    }
    /*
     * Failsafe: check that nobody set phydev->link=0 between two
     * poll cycles, otherwise we won't leave RUNNING state as long
     * as link remains down.
     */
    if (!phydev->link && phydev->state == PHY_RUNNING) {
```

```
            phydev->state = PHY_CHANGELINK;
            dev_err(&phydev->dev, "no link in PHY_RUNNING\n");
        }
        break;
    case PHY_CHANGELINK:
        err = phy_read_status(phydev);
        if (err)
            break;

        if (phydev->link) {
            phydev->state = PHY_RUNNING;
            netif_carrier_on(phydev->attached_dev);
        } else {
            phydev->state = PHY_NOLINK;
            netif_carrier_off(phydev->attached_dev);
        }

        phydev->adjust_link(phydev->attached_dev);

        if (phy_interrupt_is_valid(phydev))
            err = phy_config_interrupt(phydev,
                        PHY_INTERRUPT_ENABLED);
        break;
    case PHY_HALTED:
        if (phydev->link) {
            phydev->link = 0;
            netif_carrier_off(phydev->attached_dev);
            phydev->adjust_link(phydev->attached_dev);
            do_suspend = true;
        }
        break;
    case PHY_RESUMING:
        if (AUTONEG_ENABLE == phydev->autoneg) {
            err = phy_aneg_done(phydev);
            if (err < 0)
                break;

            /* err > 0 if AN is done.
             * Otherwise, it's 0, and we're  still waiting for AN
             */
            if (err > 0) {
                err = phy_read_status(phydev);
                if (err)
                    break;

                if (phydev->link) {
                    phydev->state = PHY_RUNNING;
                    netif_carrier_on(phydev->attached_dev);
                } else    {
                    phydev->state = PHY_NOLINK;
                }
                phydev->adjust_link(phydev->attached_dev);
            } else {
                phydev->state = PHY_AN;
```

```
                phydev->link_timeout = PHY_AN_TIMEOUT;
            }
        } else {
            err = phy_read_status(phydev);
            if (err)
                break;

            if (phydev->link) {
                phydev->state = PHY_RUNNING;
                netif_carrier_on(phydev->attached_dev);
            } else    {
                phydev->state = PHY_NOLINK;
            }
            phydev->adjust_link(phydev->attached_dev);
        }
        break;
    }

    mutex_unlock(&phydev->lock);

    if (needs_aneg)
        err = phy_start_aneg(phydev);
    else if (do_suspend)
        phy_suspend(phydev);

    if (err < 0)
        phy_error(phydev);

    dev_dbg(&phydev->dev, "PHY state change %s -> %s\n",
        phy_state_to_str(old_state), phy_state_to_str(phydev->state));

    queue_delayed_work(system_power_efficient_wq, &phydev->state_queue,
                PHY_STATE_TIME * HZ);
}
```

问:若操作系统没有加载网卡驱动,网卡虽然在系统设备树上,但网卡接口创建不了,那网卡实际能不能接到数据?

答:这里面有很多细节, 我根据Intel网卡的Spec大概写了写, 想尽量写的通俗一些,所以没有刻意用Spec的术语,另外本文虽然讲的是MAC/PHY,但光口卡的(SERDES)也是类似的.

1. PCI设备做reset以后进入D0uninitialized(非初始化的D0状态, 参考PCI电源管理规范),此时网卡MAC和DMA都不工作,PHY是工作在一个特殊的低电源状态的;

2. 操作系统创建设备树时,初始化这个设备,PCI命令寄存器的 Memory Access Enable or the I/O ccess Enable bit会被enable, 这就是D0active.此时PHY/MAC就使能了;

3. PHY被使能应该就可以接收物理链路上的数据了,否则不能收到FLP/NLP, PHY就不能建立物理连.但这类包一般是流量间歇发送的;

4. 驱动程序一般要通过寄存器来控制PHY, 比如自动协商speed/duplex, 查询物理链路的状态Link p/down;

5. MAC被使能后, 如果没有驱动设置控制寄存器的一个位(CTRL.SLU)的话, MAC和PHY是不能通的, 就是说MAC不知道PHY的link已经ready, 所以收不到任何数据的.这位设置以后, PHY完成自协商, 卡才会有个Link change的中断,知道物理连接已经Link UP了;

6. 即使Link已经UP, MAC还需要enable接收器的一个位(RCTL.RXEN),包才可以被接收进来,如果

卡被reset,这位是0,意味着所有的包都会被直接drop掉,不会存入网卡的 FIFO.老网卡在驱动退出前利这位关掉接收.Intel的最新千兆网卡发送接收队列的动态配置就是依靠这个位的,重新配置的过程一定关掉流量;

7. 无论驱动加载与否, 发生reset后,网卡EEPOM里的mac地址会写入网卡的MAC地址过滤寄存器,动可以去修改这个寄存器,现代网卡通常支持很多MAC地址,也就是说,MAC地址是可以被软件设置的.如,Intel的千兆网卡就支持16个单播 MAC地址,但只有1个是存在EEPROM里的,其它是软件声称和设的;

8. 但如果驱动没有加载,网卡已经在设备树上,操作系统完成了步骤1-2的初始化,此时网卡的PHY应是工作的,但因为没有人设置控制位(CTRL.SLU)来让MAC和PHY建立联系,所以MAC是不收包的.这个制位在reset时会再设置成0;

9. PHY可以被软件设置加电和断电, 断电状态除了接收管理命令以外,不会接收数据.另外,PHY还能作在Smart Power Down模式下,link down就进入省电状态;

10. 有些多口网卡,多个网口共享一个PHY, 所以BIOS里设置disbale了某个网口, 也未必会把PHY的源关掉,反过来,也要小心地关掉PHY的电源;

11. 要详细了解PHY,最终还是要熟悉IEEE以太网的相关协议.