基于335X的UBOOT网口驱动分析 - lh03061238 - 博客园

lh03061238 关注 - 6 粉丝 - 23 +加关注

基于335X的UBOOT网口驱动分析

一、软硬件平台资料

1、 开发板：创龙AM3359核心板，网口采用RMII形式

2、 UBOOT版本：U-Boot-2016.05，采用FDT和DM。

参考链接:

https://blog.csdn.net/hahachenchen789/article/details/53339181

二、网口相关代码位置

1、 网口的PINMUX设置

RMII接口的相关PINMUX在MLO中进行设置，具体的设置代码为
|-board_init_f

|-board_early_init_f

|-set_mux_conf_regs

|-enable_board_pin_mux

configure_module_pin_mux(rmii1_pin_mux);

2、DTS文件中的CPSW的配置

```
&cpsw_emac0 {
phy_id = <&davinci_mdio>, <0x12>;  //phy_id【1】为初始的phy_addr,为SW的PORT2口的ADDR。
phy-mode = "rmii";      //RMII 模式
};

&mac {                                  //未使用此处配置
slaves = <1>;
pinctrl-names = "default", "sleep";
pinctrl-0 = <&cpsw_default>;
pinctrl-1 = <&cpsw_sleep>;
status = "okay";
};
```

```
&phy_sel {
rmii-clock-ext;  //RMII模式的时钟为外部时钟
};

&davinci_mdio {                         //未使用此处配置
pinctrl-names = "default", "sleep";
pinctrl-0 = <&davinci_mdio_default>;
pinctrl-1 = <&davinci_mdio_sleep>;
status = "okay";
reset-gpios = <&gpio2 5 GPIO_ACTIVE_LOW>;
reset-delay-us = <2>; /* PHY datasheet states 1uS min */
};
```

3、 网口的初始化设置

```
|-board_init_r

 |-init_sequence_r

  |-initr_net

   |- eth_initialize  （eth-uclass.c）
```

三、有关网口的DM&FDT分析

1、 驱动实现方式

此版本的UBOOT中使用了FDT文件进行外设的相关配置，驱动模型使用了DM方式，有关FDT以及DM相关的知识请参考如下文章

https://blog.csdn.net/ooonebook/article/details/53206623

https://blog.csdn.net/ooonebook/article/details/53234020

2、 UBOOT中DM初始化

DM的初始化

.创建根设备root的udevice，存放在gd->dm_root中。

.根设备其实是一个虚拟设备，主要是为uboot的其他设备提供一个挂载点。

.初始化uclass链表gd->uclass_root

DM中udevice和uclass的解析

.udevice的创建和uclass的创建

.udevice和uclass的绑定

.uclass_driver和uclass的绑定

.driver和udevice的绑定

.部分driver函数的调用

（1）DM初始化调用过程

dm初始化的接口在dm_init_and_scan中。可以发现在uboot relocate之前的initf_dm和之后的initr_dm都调用了这个函数。

```
static int initf_dm(void)

{

#if defined(CONFIG_DM) && defined(CONFIG_SYS_MALLOC_F_LEN)

   int ret;

   ret = dm_init_and_scan(true); // 调用dm_init_and_scan对DM进行初始化和设备的解析

   if (ret)

      return ret;

#endif

   return 0;

}

#ifdef CONFIG_DM

static int initr_dm(void)

{

   int ret;

   /* Save the pre-reloc driver model and start a new one */

   gd->dm_root_f = gd->dm_root; // 存储relocate之前的根设备

   gd->dm_root = NULL;

   ret = dm_init_and_scan(false); // 调用dm_init_and_scan对DM进行初始化和设备的解析

   if (ret)

      return ret;

   return 0;

}

#endif
```

主要区别在于参数。

首先说明一下dts节点中的“u-boot,dm-pre-reloc”属性，当设置了这个属性时，则表示这个设备在relocate之前就需要使用。

当dm_init_and_scan的参数为true时，只会对带有“u-boot,dm-pre-reloc”属性的节点进行解析。而当参数为false的时候，则会对所有节点都进行解析。

由于“u-boot,dm-pre-reloc”的情况比较少，所以这里只学习参数为false的情况。也就是initr_dm里面的dm_init_and_scan(false);。

dm_init_and_scan（driver/core/root.c）说明

```
int dm_init_and_scan(bool pre_reloc_only)

{

   int ret;

   ret = dm_init();   // DM的初始化

   if (ret) {

      debug("dm_init() failed: %d\n", ret);

      return ret;

   }

   ret = dm_scan_platdata(pre_reloc_only); //  从平台设备中解析udevice和
uclass

   if (ret) {

      debug("dm_scan_platdata() failed: %d\n", ret);

      return ret;

   }

   if (CONFIG_IS_ENABLED(OF_CONTROL)) {

      ret = dm_scan_fdt(gd->fdt_blob, pre_reloc_only); // 从dtb中解析udevice和
uclass

      if (ret) {

         debug("dm_scan_fdt() failed: %d\n", ret);

         return ret;

      }

   }
```

```
    ret = dm_scan_other(pre_reloc_only);

    if (ret)

        return ret;

    return 0;

}
```

DM的初始化—dm_init（driver/core/root.c）

```
#define DM_ROOT_NON_CONST       (((gd_t *)gd)->dm_root) // 宏定义根设备指针gd->dm_root

#define DM_UCLASS_ROOT_NON_CONST    (((gd_t *)gd)->uclass_root) // 宏定义gd->uclass_root，uclass的链表

int dm_init(void)

{

    int ret;

    if (gd->dm_root) {

    // 根设备已经存在，说明DM已经初始化过了

        dm_warn("Virtual root driver already exists!\n");

        return -EINVAL;

    }

    INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);

        // 初始化uclass链表

    ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);

        // DM_ROOT_NON_CONST是指根设备udevice，root_info是表示根设备的设备信息

        // device_bind_by_name会查找和设备信息匹配的driver，然后创建对应的udevice和uclass并进行绑定，最后放在DM_ROOT_NON_CONST中。

        // device_bind_by_name后续我们会进行说明，这里我们暂时只需要了解root根设备的udevice以及对应的uclass都已经创建完成。

    if (ret)

        return ret;
```

```
#if CONFIG_IS_ENABLED(OF_CONTROL)
   DM_ROOT_NON_CONST->of_offset = 0;
#endif
   ret = device_probe(DM_ROOT_NON_CONST);
      // 对根设备执行probe操作，
      // device_probe后续再进行说明
   if (ret)
      return ret;
   return 0;
}
```

这里就完成的DM的初始化了

1）创建根设备root的udevice，存放在gd->dm_root中。

2）初始化uclass链表gd->uclass_root

（2）从平台设备中解析udevice和uclass—dm_scan_platdata（不涉及）

（3）从dtb中解析udevice和uclass——dm_scan_fdt

对应代码如下driver/core/root.c

```
int dm_scan_fdt(const void *blob, bool pre_reloc_only)
// 此时传进来的参数blob=gd->fdt_blob， pre_reloc_only=0
{
   return dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);
// 直接调用dm_scan_fdt_node
}
int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
            bool pre_reloc_only)
// 此时传进来的参数
// parent=gd->dm_root，表示以root设备作为父设备开始解析
// blob=gd->fdt_blob，指定了对应的dtb
```

```
// offset=0，从偏移0的节点开始扫描
// pre_reloc_only=0，不只是解析relotion之前的设备
{
    int ret = 0, err;
        /*  以下步骤相当于是遍历每一个dts节点并且调用lists_bind_fdt对其进行解析 */
    for (offset = fdt_first_subnode(blob, offset);
        // 获得blob设备树的offset偏移下的节点的第一个子节点
         offset > 0;
         offset = fdt_next_subnode(blob, offset)) {
              // 循环查找下一个子节点
        if (!fdtdec_get_is_enabled(blob, offset)) {
                    // 判断节点状态是否是disable，如果是的话直接忽略
            dm_dbg("   - ignoring disabled device\n");
            continue;
        }
        err = lists_bind_fdt(parent, blob, offset, NULL);
              // 解析绑定这个节点，dm_scan_fdt的核心，下面具体分析
        if (err && !ret) {
            ret = err;
            debug("%s: ret=%d\n", fdt_get_name(blob, offset, NULL),
                  ret);
        }
    }
    return ret;
}
```

lists_bind_fdt是从dtb中解析udevice和uclass的核心。

其具体实现如下： driver/core/lists.c

```
int lists_bind_fdt(struct udevice *parent, const void *blob, int offset,
        struct udevice **devp)
// parent指定了父设备，通过blob和offset可以获得对应的设备的dts节点，对应udevice结构通过devp返回
{
    struct driver *driver = ll_entry_start(struct driver, driver);
// 获取driver table地址
    const int n_ents = ll_entry_count(struct driver, driver);
// 获取driver table长度
    const struct udevice_id *id;
    struct driver *entry;
    struct udevice *dev;
    bool found = false;
    const char *name;
    int result = 0;
    int ret = 0;
    dm_dbg("bind node %s\n", fdt_get_name(blob, offset, NULL));
// 打印当前解析的节点的名称
    if (devp)
        *devp = NULL;
    for (entry = driver; entry != driver + n_ents; entry++) {
// 遍历driver table中的所有driver，具体参考三、4一节
        ret = driver_check_compatible(blob, offset, entry->of_match,
                      &id);
// 判断driver中的compatibile字段和dts节点是否匹配
        name = fdt_get_name(blob, offset, NULL);
// 获取节点名称
        if (ret == -ENOENT) {
```

```
            continue;
        } else if (ret == -ENODEV) {
            dm_dbg("Device '%s' has no compatible string\n", name);
            break;
        } else if (ret) {
            dm_warn("Device tree error at offset %d\n", offset);
            result = ret;
            break;
        }
        dm_dbg("   - found match at '%s'\n", entry->name);
        ret = device_bind(parent, entry, name, NULL, offset, &dev);
// 找到对应的driver，调用device_bind进行绑定，会在这个函数中创建对应udevice和uclass并切进行绑定，后面继续说明
        if (ret) {
            dm_warn("Error binding driver '%s': %d\n", entry->name,
                ret);
            return ret;
        } else {
            dev->driver_data = id->data;
            found = true;
            if (devp)
                *devp = dev;
// 将udevice设置到devp指向的地方中，进行返回
        }
        break;
    }
    if (!found && !result && ret != -ENODEV) {
        dm_dbg("No match for node '%s'\n",
```

```
            fdt_get_name(blob, offset, NULL));
    }
    return result;
}
```

在device_bind中实现了udevice和uclass的创建和绑定以及一些初始化操作，这里专门学习一下device_bind。

device_bind的实现如下(去除部分代码)

driver/core/device.c

```
int device_bind(struct udevice *parent, const struct driver *drv,
        const char *name, void *platdata, int of_offset,
        struct udevice **devp)
// parent:父设备
// drv：设备对应的driver
// name：设备名称
// platdata：设备的平台数据指针
// of_offset：在dtb中的偏移，即代表了其dts节点
// devp：所创建的udevice的指针，用于返回
{
    struct udevice *dev;
    struct uclass *uc;
    int size, ret = 0;
    ret = uclass_get(drv->id, &uc);
        // 获取driver id对应的uclass，如果uclass原先并不存在，那么会在这里创建uclass并其uclass_driver进行绑定
    dev = calloc(1, sizeof(struct udevice));
        // 分配一个udevice
    dev->platdata = platdata; // 设置udevice的平台数据指针
    dev->name = name; // 设置udevice的name
```

```
    dev->of_offset = of_offset; // 设置udevice的dts节点偏移
    dev->parent = parent; // 设置udevice的父设备
    dev->driver = drv;    // 设置udevice的对应的driver，相当于driver和udevice
的绑定
    dev->uclass = uc;    // 设置udevice的所属uclass
    dev->seq = -1;
    dev->req_seq = -1;
    if (CONFIG_IS_ENABLED(OF_CONTROL) &&
CONFIG_IS_ENABLED(DM_SEQ_ALIAS)) {
        /*
         * Some devices, such as a SPI bus, I2C bus and serial ports
         * are numbered using aliases.
         *
         * This is just a 'requested' sequence, and will be
         * resolved (and ->seq updated) when the device is probed.
         */
        if (uc->uc_drv->flags & DM_UC_FLAG_SEQ_ALIAS) {
            if (uc->uc_drv->name && of_offset != -1) {
                fdtdec_get_alias_seq(gd->fdt_blob,
                        uc->uc_drv->name, of_offset,
                        &dev->req_seq);
            }
                // 设置udevice的alias请求序号
        }
    }
    if (!dev->platdata && drv->platdata_auto_alloc_size) {
        dev->flags |= DM_FLAG_ALLOC_PDATA;
        dev->platdata = calloc(1, drv->platdata_auto_alloc_size);
```

```
            // 为udevice分配平台数据的空间，由driver中的
platdata_auto_alloc_size决定

    }

    size = uc->uc_drv->per_device_platdata_auto_alloc_size;

    if (size) {

        dev->flags |= DM_FLAG_ALLOC_UCLASS_PDATA;

        dev->uclass_platdata = calloc(1, size);

            // 为udevice分配给其所属uclass使用的平台数据的空间，由所属
uclass的driver中的per_device_platdata_auto_alloc_size决定

    }

    /* put dev into parent's successor list */

    if (parent)

        list_add_tail(&dev->sibling_node, &parent->child_head);

        // 添加到父设备的子设备链表中

    ret = uclass_bind_device(dev);

        // uclass和udevice进行绑定，主要是实现了将udevice链接到uclass的设
备链表中

    /* if we fail to bind we remove device from successors and free it */

    if (drv->bind) {

        ret = drv->bind(dev);

        // 执行udevice对应driver的bind函数

    }

    if (parent && parent->driver->child_post_bind) {

        ret = parent->driver->child_post_bind(dev);

        // 执行父设备的driver的child_post_bind函数

    }

    if (uc->uc_drv->post_bind) {

        ret = uc->uc_drv->post_bind(dev);

        if (ret)
```

```
            goto fail_uclass_post_bind;

        // 执行所属uclass的post_bind函数

    }

    if (devp)

        *devp = dev;

        // 将udevice进行返回

    dev->flags |= DM_FLAG_BOUND;

        // 设置已经绑定的标志

        // 后续可以通过dev->flags & DM_FLAG_ACTIVATED或者device_active
宏来判断设备是否已经被激活

    return 0;
```

在init_sequence_r中的initr_dm中，完成了FDT的解析，解析了所有的外设node,并针对各个节点进行了 udevice和uclass的创建，以及各个组成部分的绑定关系。

注意，这里只是绑定，即调用了driver的bind函数，但是设备还没有真正激活，也就是还没有执行设备的probe函数。

将在网口初始化阶段进行相关driver的bind操作。

四、网口的初始化过程分析

1、 eth_initialize函数

网口初始化，其中最主要的工作是调用uclass_first_device(UCLASS_ETH, &dev)函数，从uclass的设备链表中获取第一个udevice，并且进行probe。最终，是通过调用device_probe(dev)进行网口设备的激活和驱动的注册。下面分析device_probe(dev)的实现的部分过程。

```
int device_probe(struct udevice *dev)

{

    const struct driver *drv;

    int size = 0;

    int ret;

    int seq;

    if (dev->flags & DM_FLAG_ACTIVATED)

        return 0;
```

```
// 表示这个设备已经被激活了
    drv = dev->driver;
    assert(drv);
// 获取这个设备对应的driver
    /* Allocate private data if requested and not reentered */
    if (drv->priv_auto_alloc_size && !dev->priv) {
        dev->priv = alloc_priv(drv->priv_auto_alloc_size, drv->flags);
// 为设备分配私有数据
    }
    /* Allocate private data if requested and not reentered */
    size = dev->uclass->uc_drv->per_device_auto_alloc_size;
    if (size && !dev->uclass_priv) {
        dev->uclass_priv = calloc(1, size);
// 为设备所属uclass分配私有数据
    }
// 这里过滤父设备的probe
    seq = uclass_resolve_seq(dev);
    if (seq < 0) {
        ret = seq;
        goto fail;
    }
    dev->seq = seq;
    dev->flags |= DM_FLAG_ACTIVATED;
// 设置udevice的激活标志
    ret = uclass_pre_probe_device(dev);
// uclass在probe device之前的一些函数的调用
    if (drv->ofdata_to_platdata && dev->of_offset >= 0) {
```

```
        ret = drv->ofdata_to_platdata(dev);
// 调用driver中的ofdata_to_platdata将dts信息转化为设备的平台数据
    }
    if (drv->probe) {
        ret = drv->probe(dev);
// 调用driver的probe函数，到这里设备才真正激活了
    }
    ret = uclass_post_probe_device(dev);
    return ret;
}
```

主要工作归纳如下：

.分配设备的私有数据

.对父设备进行probe

.执行probe device之前uclass需要调用的一些函数

.调用driver的ofdata_to_platdata，将dts信息转化为设备的平台数据（重要）

.调用driver的probe函数（重要）

.执行probe device之后uclass需要调用的一些函数

在CPSW.c中有相关定义：

```
U_BOOT_DRIVER(eth_cpsw) = {
        .name      = "eth_cpsw",
        .id     = UCLASS_ETH,
        .of_match = cpsw_eth_ids,
        .ofdata_to_platdata = cpsw_eth_ofdata_to_platdata,
        .probe      = cpsw_eth_probe,
        .ops = &cpsw_eth_ops,
        .priv_auto_alloc_size = sizeof(struct cpsw_priv),
        .platdata_auto_alloc_size = sizeof(struct eth_pdata),
```

```
        .flags = DM_FLAG_ALLOC_PRIV_DMA,
};
```

2、有关DTS配置信息转化的函数（drv->ofdata_to_platdata）

```
static int cpsw_eth_ofdata_to_platdata(struct udevice *dev)
{
        struct eth_pdata *pdata = dev_get_platdata(dev);
        struct cpsw_priv *priv = dev_get_priv(dev);
        const char *phy_mode;
        const char *phy_sel_compat = NULL;
        const void *fdt = gd->fdt_blob;
        int node = dev->of_offset;
        int subnode;
        int slave_index = 0;
        int active_slave;
        int ret;
        pdata->iobase = dev_get_addr(dev);
        priv->data.version = CPSW_CTRL_VERSION_2;
        priv->data.bd_ram_ofs = CPSW_BD_OFFSET;
        priv->data.ale_reg_ofs = CPSW_ALE_OFFSET;
        priv->data.cpdma_reg_ofs = CPSW_CPDMA_OFFSET;
        priv->data.mdio_div = CPSW_MDIO_DIV;
        priv->data.host_port_reg_ofs = CPSW_HOST_PORT_OFFSET,
        pdata->phy_interface = -1;
        priv->data.cpsw_base = pdata->iobase;
        priv->data.channels = fdtdec_get_int(fdt, node, "cpdma_channels", -1);
        if (priv->data.channels <= 0) {
                printf("error: cpdma_channels not found in dt\n");
```

```
        return -ENOENT;
    }
    priv->data.slaves = fdtdec_get_int(fdt, node, "slaves", -1);
    if (priv->data.slaves <= 0) {
        printf("error: slaves not found in dt\n");
        return -ENOENT;
    }
    priv->data.slave_data = malloc(sizeof(struct cpsw_slave_data) *
                priv->data.slaves);
    priv->data.ale_entries = fdtdec_get_int(fdt, node, "ale_entries", -1);
    if (priv->data.ale_entries <= 0) {
        printf("error: ale_entries not found in dt\n");
        return -ENOENT;
    }
    priv->data.bd_ram_ofs = fdtdec_get_int(fdt, node, "bd_ram_size", -1);
    if (priv->data.bd_ram_ofs <= 0) {
        printf("error: bd_ram_size not found in dt\n");
        return -ENOENT;
    }
    priv->data.mac_control = fdtdec_get_int(fdt, node, "mac_control", -1);
    if (priv->data.mac_control <= 0) {
        printf("error: ale_entries not found in dt\n");
        return -ENOENT;
    }
    active_slave = fdtdec_get_int(fdt, node, "active_slave", 0);
    priv->data.active_slave = active_slave;
    fdt_for_each_subnode(fdt, subnode, node) {
```

```
int len;
const char *name;
name = fdt_get_name(fdt, subnode, &len);
if (!strncmp(name, "mdio", 4)) {
        u32 mdio_base;
        mdio_base = cpsw_get_addr_by_node(fdt, subnode);
        if (mdio_base == FDT_ADDR_T_NONE) {
                error("Not able to get MDIO address space\n");
                return -ENOENT;
        }
        priv->data.mdio_base = mdio_base;
}
if (!strncmp(name, "slave", 5)) {
        u32 phy_id[2];
        if (slave_index >= priv->data.slaves)
                continue;
        phy_mode = fdt_getprop(fdt, subnode, "phy-mode", NULL);
        if (phy_mode)
                priv->data.slave_data[slave_index].phy_if =
                        phy_get_interface_by_name(phy_mode);
        priv->data.slave_data[slave_index].phy_of_handle =
                fdtdec_lookup_phandle(fdt, subnode, "phy-handle");
        if (priv->data.slave_data[slave_index].phy_of_handle >= 0) {
                priv->data.slave_data[slave_index].phy_addr =
                        fdtdec_get_int(gd->fdt_blob,
```

```
priv->data.slave_data[slave_index].phy_of_handle,
                                        "reg", -1);
```

```
                } else {
                        fdtdec_get_int_array(fdt, subnode, "phy_id", phy_id, 2);
                        priv->data.slave_data[slave_index].phy_addr = phy_id[1];
                }
                slave_index++;
        }
        if (!strncmp(name, "cpsw-phy-sel", 12)) {
                priv->data.gmii_sel = cpsw_get_addr_by_node(fdt,
                                                        subnode);
                if (priv->data.gmii_sel == FDT_ADDR_T_NONE) {
                        error("Not able to get gmii_sel reg address\n");
                        return -ENOENT;
                }
                if (fdt_get_property(fdt, subnode, "rmii-clock-ext",
                                NULL))
                        {
                            priv->data.rmii_clock_external = true;
                            printf("data.rmii_clock_external is true\n");
                        }
                phy_sel_compat = fdt_getprop(fdt, subnode, "compatible",
                                        NULL);
                if (!phy_sel_compat) {
                        printf("Not able to get gmii_sel compatible\n");
                        return -ENOENT;
                }
        }
}
```

```
        priv->data.slave_data[0].slave_reg_ofs = CPSW_SLAVE0_OFFSET;

        priv->data.slave_data[0].sliver_reg_ofs = CPSW_SLIVER0_OFFSET;

        if (priv->data.slaves == 2) {

                priv->data.slave_data[1].slave_reg_ofs = CPSW_SLAVE1_OFFSET;

                priv->data.slave_data[1].sliver_reg_ofs =
CPSW_SLIVER1_OFFSET;

        }

        ret = ti_cm_get_macid(dev, active_slave, pdata->enetaddr);

        if (ret < 0) {

                error("cpsw read efuse mac failed\n");

                return ret;

        }

        pdata->phy_interface = priv->data.slave_data[active_slave].phy_if;

        if (pdata->phy_interface == -1) {

                debug("%s: Invalid PHY interface '%s'\n", __func__, phy_mode);

                return -EINVAL;

        }

        /* Select phy interface in control module */

        cpsw_phy_sel(priv, phy_sel_compat, pdata->phy_interface);

        return 0;

}
```

可以看到，在cpsw_eth_ofdata_to_platdata函数中将各种与CPSW有关的平台数据宏定义以及DTS中的配置信息（包含个子节点）转化为了平台数据存储在了priv->data的相关部分中。主要涉及priv->data的相关设置，此部分重要的信息是MAC的接口形式，比如RMII的设置，RMII时钟的使能，phy_addr的设置。

3、有关驱动注册的函数（drv->probe(dev)）

```
static int cpsw_eth_probe(struct udevice *dev)

{

        struct cpsw_priv *priv = dev_get_priv(dev);
```

```
    printf("cpsw_eth_probe now\n");

     priv->dev = dev;

     return _cpsw_register(priv);

}
```

TI对于网卡设备的通用管理是CPSW方式，通过cpsw_priv结构体来进行相关的管理，cpsw_priv结构体中包含有CPSW平台数据、cpsw_slave的信息、priv->bus（MII接口管理）、phy_device设备的配置及管理等。

cpsw_register(priv)函数主要进行以下工作

（1）、首先是声明几个结构体变量，其中包括cpsw的主：cpsw_priv和从：cpsw_slave，然后是设置cpsw的基础寄存器的地址cpsw_base，然后调用calloc函数为这些结构体分配空间。

```
struct cpsw_slave   *slave;

struct cpsw_platform_data *data = &priv->data;

void                *regs = (void *)data->cpsw_base;

priv->slaves = malloc(sizeof(struct cpsw_slave) * data->slaves);
```

（2）、分配好后对priv结构体中的成员进行初始化，host_port=0表示主机端口号是0，然后成员的寄存器的偏移地址进行初始化。

```
Priv->host_port             = data->host_port_num;

     priv->regs              = regs;

     priv->host_port_regs    = regs + data->host_port_reg_ofs;

     priv->dma_regs          = regs + data->cpdma_reg_ofs;

     priv->ale_regs          = regs + data->ale_reg_ofs;

     priv->descs             = (void *)regs + data->bd_ram_ofs;
```

（3）、对每个salve进行初始化，这里采用for循环的意义在于可能有多个网卡，am335支持双网卡。

```
for_each_slave(slave, priv) {

          cpsw_slave_setup(slave, idx, priv);

          idx = idx + 1;

               }
```

（4）、对MDIO接口的操作集进行初始化配置

cpsw_mdio_init(priv->dev->name, data->mdio_base, data->mdio_div);

.进行了mii_dev设备的创建

.进行了mdio_regs寄存器的配置（set enable and clock divider）

.进行了cpsw_mdio_read/ cpsw_mdio_write的定义（用此函数对PHY进行读写）

.mii_dev设备注册（加到mii_devs链表，并指定为current_mii）

(4)指定priv->bus为上一步创建的设备

```
  priv->bus = miiphy_get_dev_by_name(priv->dev->name);
```

(5) phydev初始化/配置（重点）

cpsw_phy_init函数定义：

```
static int cpsw_phy_init(struct eth_device *dev, struct cpsw_slave *slave)
{
        struct cpsw_priv *priv = (struct cpsw_priv *)dev->priv;
        struct phy_device *phydev;
        u32 supported = PHY_GBIT_FEATURES;
   printf("cpsw_phy_init \n");
   printf("phy_addr:%d \n",slave->data->phy_addr);
        phydev = phy_connect(priv->bus,
                        slave->data->phy_addr,
                        dev,
                        slave->data->phy_if);
        if (!phydev)
                return -1;
        phydev->supported &= supported;
        phydev->advertising = phydev->supported;
        priv->phydev = phydev;
        phy_config(phydev);
        return 1;
```

```
}
```

该函数调用phy_connect函数连接网卡，返回的值如果合理就调用phy_config函数对该网卡进行配置，主要是配置网卡的速率和半双工，自动协商等，此部分需要再进一步调试熟悉。

首先分析phy_connect函数：

```
struct phy_device *phy_connect(struct mii_dev *bus, int addr,
        struct eth_device *dev, phy_interface_t interface)
#endif
{
    struct phy_device *phydev;
    phydev = phy_find_by_mask(bus, 1 << addr, interface);
    if (phydev)
        phy_connect_dev(phydev, dev);
    else
        printf("Could not get PHY for %s: addr %d\n", bus->name, addr);
    return phydev;
}
```

该函数首先调用phy_find_by_mask函数查询网卡设备，如果存在则调用phy_connect_dev函数连接，否则就打印出错信息

```
struct phy_device *phy_find_by_mask(struct mii_dev *bus, unsigned phy_mask,
        phy_interface_t interface)
{
    /* Reset the bus */
    if (bus->reset) {
        bus->reset(bus);
        /* Wait 15ms to make sure the PHY has come out of hard reset */
        udelay(15000);
    }
    return get_phy_device_by_mask(bus, phy_mask, interface);
```

```
}
```

该函数主要是调用get_phy_device_by_mask函数进行设备的查找，get_phy_device_by_mask函数的实现至关重要，包含了对于网卡的主要mdio通信。

```
static struct phy_device *get_phy_device_by_mask(struct mii_dev *bus,
        unsigned phy_mask, phy_interface_t interface)
{
    int i;
    struct phy_device *phydev;
    phydev = search_for_existing_phy(bus, phy_mask, interface);
    if (phydev)
        return phydev;
    /* Try Standard (ie Clause 22) access */
    /* Otherwise we have to try Clause 45 */
    for (i = 0; i < 5; i++) {
        phydev = create_phy_by_mask(bus, phy_mask,
                i ? i : MDIO_DEVAD_NONE, interface);
        if (IS_ERR(phydev))
            return NULL;
        if (phydev)
            return phydev;
    }
    printf("Phy %d not found\n", ffs(phy_mask) - 1);
    return phy_device_create(bus, ffs(phy_mask) - 1, 0xffffffff, interface);
}
```

该函数首先调用search_for_existing_phy函数查找当前存在的设备，如果存在则将该设备返回，不存在则调用create_phy_by_mask函数进行创建。重点看下create_phy_by_mask函数

```
static struct phy_device *create_phy_by_mask(struct mii_dev *bus,
```

```
                unsigned phy_mask, int devad, phy_interface_t interface)
{
        u32 phy_id = 0xffffffff;
        while (phy_mask) {
                int addr = ffs(phy_mask) - 1;
                int r = get_phy_id(bus, addr, devad, &phy_id);
                /* If the PHY ID is mostly f's, we didn't find anything */
                if (r == 0 && (phy_id & 0x1fffffff) != 0x1fffffff)
                        return phy_device_create(bus, addr, phy_id, interface);
                phy_mask &= ~(1 << addr);
        }
        return NULL;
}
```

该函数调用get_phy_id函数让处理器通过mdio总线查看网卡寄存器存储的ID，如果ID都是f，说明没有ID，就返回空，否则返回phy_device_create函数进行创建一个网卡设备。

get_phy_id函数实现：

```
int __weak get_phy_id(struct mii_dev *bus, int addr, int devad, u32 *phy_id)
{
        int phy_reg;
        /* Grab the bits from PHYIR1, and put them
         * in the upper half */
        phy_reg = bus->read(bus, addr, devad, MII_PHYSID1);
        if (phy_reg < 0)
                return -EIO;
        *phy_id = (phy_reg & 0xffff) << 16;
        /* Grab the bits from PHYIR2, and put them in the lower half */
        phy_reg = bus->read(bus, addr, devad, MII_PHYSID2);
```

```
        if (phy_reg < 0)
                return -EIO;
        *phy_id |= (phy_reg & 0xffff);
        return 0;
}
```

该函数就调用了bus->read总线读函数，来读取网卡寄存器的值，这里是读取寄存器存储的网卡ID，bus->read函数定义为cpsw_mdio_read

之后的phy_device_create函数为新创建一个phy_device及相关参数，以及相对应的phy_driver。

```
static struct phy_device *phy_device_create(struct mii_dev *bus, int addr,
                                            u32 phy_id,
                                            phy_interface_t interface)
{
        struct phy_device *dev;
        /* We allocate the device, and initialize the
         * default values */
        dev = malloc(sizeof(*dev));
        if (!dev) {
                printf("Failed to allocate PHY device for %s:%d\n",
                        bus->name, addr);
                return NULL;
        }
        memset(dev, 0, sizeof(*dev));
        dev->duplex = -1;
        dev->link = 0;
        dev->interface = interface;
        dev->autoneg = AUTONEG_ENABLE;
        dev->addr = addr;
```

```
        dev->phy_id = phy_id;

        dev->bus = bus;

        dev->drv = get_phy_driver(dev, interface);

        phy_probe(dev);

        bus->phymap[addr] = dev;

        return dev;

}
```

其中get_phy_driver会根据phy_id进行phy_driver的查找，若没有找到，则分配一个"Generic PHY"。

综上：cpsw_eth_probe的最终结果，是初始化了cpsw_priv各个部分，包括各个参数及mii_dev及phy_dev.

其中phy_dev非常重要，从后面的逻辑看出，phy_dev存在的情况下会根据LINK状态下的mac_control值对slave->sliver->mac_control寄存器进行配置。这决定了RMII接口的正确配置。

所以，必须有一个phy_dev？或者有一个mac_control值对slave->sliver->mac_control寄存器进行配置?

1. 驱动的初始化及调用

eth_init->eth_get_ops(current)->start(current)来进行网口通信的底层配置

```
static int cpsw_eth_start(struct udevice *dev)

{

struct eth_pdata *pdata = dev_get_platdata(dev);

struct cpsw_priv *priv = dev_get_priv(dev);

     printf("cpsw_eth_start_now\n");

return _cpsw_init(priv, pdata->enetaddr);

}

static int _cpsw_init(struct cpsw_priv *priv, u8 *enetaddr)

{

        struct cpsw_slave   *slave;

        int i, ret;

        printf("_cpsw_init now\n");
```

```
        /* soft reset the controller and initialize priv */
        setbit_and_wait_for_clear32(&priv->regs->soft_reset);
        /* initialize and reset the address lookup engine */
        cpsw_ale_enable(priv, 1);
        cpsw_ale_clear(priv, 1);
        cpsw_ale_vlan_aware(priv, 0); /* vlan unaware mode */
        /* setup host port priority mapping */
        __raw_writel(0x76543210, &priv->host_port_regs->cpdma_tx_pri_map);
        __raw_writel(0, &priv->host_port_regs->cpdma_rx_chan_map);
        /* disable priority elevation and enable statistics on all ports */
        __raw_writel(0, &priv->regs->ptype);
        /* enable statistics collection only on the host port */
        __raw_writel(BIT(priv->host_port), &priv->regs->stat_port_en);
        __raw_writel(0x7, &priv->regs->stat_port_en);
        cpsw_ale_port_state(priv, priv->host_port,
ALE_PORT_STATE_FORWARD);
        cpsw_ale_add_ucast(priv, enetaddr, priv->host_port, ALE_SECURE);
        cpsw_ale_add_mcast(priv, net_bcast_ethaddr, 1 << priv->host_port);
        for_active_slave(slave, priv)
        cpsw_slave_init(slave, priv);
        cpsw_update_link(priv);
        /* init descriptor pool */
        for (i = 0; i < NUM_DESCS; i++) {
                desc_write(&priv->descs[i], hw_next,
                        (i == (NUM_DESCS - 1)) ? 0 : &priv->descs[i+1]);
        }
        priv->desc_free = &priv->descs[0];
        /* initialize channels */
```

```
        if (priv->data.version == CPSW_CTRL_VERSION_2) {
                memset(&priv->rx_chan, 0, sizeof(struct cpdma_chan));
                priv->rx_chan.hdp       = priv->dma_regs +
CPDMA_RXHDP_VER2;
                priv->rx_chan.cp        = priv->dma_regs + CPDMA_RXCP_VER2;
                priv->rx_chan.rxfree    = priv->dma_regs + CPDMA_RXFREE;
                memset(&priv->tx_chan, 0, sizeof(struct cpdma_chan));
                priv->tx_chan.hdp       = priv->dma_regs +
CPDMA_TXHDP_VER2;
                priv->tx_chan.cp        = priv->dma_regs + CPDMA_TXCP_VER2;
        } else {
                memset(&priv->rx_chan, 0, sizeof(struct cpdma_chan));
                priv->rx_chan.hdp       = priv->dma_regs +
CPDMA_RXHDP_VER1;
                priv->rx_chan.cp        = priv->dma_regs + CPDMA_RXCP_VER1;
                priv->rx_chan.rxfree    = priv->dma_regs + CPDMA_RXFREE;
                memset(&priv->tx_chan, 0, sizeof(struct cpdma_chan));
                priv->tx_chan.hdp       = priv->dma_regs +
CPDMA_TXHDP_VER1;
                priv->tx_chan.cp        = priv->dma_regs + CPDMA_TXCP_VER1;
        }
        /* clear dma state */
        setbit_and_wait_for_clear32(priv->dma_regs + CPDMA_SOFTRESET);
        if (priv->data.version == CPSW_CTRL_VERSION_2) {
                for (i = 0; i < priv->data.channels; i++) {
                        __raw_writel(0, priv->dma_regs + CPDMA_RXHDP_VER2 +
4
                                        * i);
                        __raw_writel(0, priv->dma_regs + CPDMA_RXFREE + 4
                                        * i);
```

```
                __raw_writel(0, priv->dma_regs + CPDMA_RXCP_VER2 + 4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_TXHDP_VER2 +
4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_TXCP_VER2 + 4
                        * i);
        }
    } else {
        for (i = 0; i < priv->data.channels; i++) {
                __raw_writel(0, priv->dma_regs + CPDMA_RXHDP_VER1 +
4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_RXFREE + 4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_RXCP_VER1 + 4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_TXHDP_VER1 +
4
                        * i);
                __raw_writel(0, priv->dma_regs + CPDMA_TXCP_VER1 + 4
                        * i);
        }
    }
    __raw_writel(1, priv->dma_regs + CPDMA_TXCONTROL);
    __raw_writel(1, priv->dma_regs + CPDMA_RXCONTROL);
    /* submit rx descs */
    for (i = 0; i < PKTBUFSRX; i++) {
        ret = cpdma_submit(priv, &priv->rx_chan, net_rx_packets[i],
```

```
                                PKTSIZE);

                if (ret < 0) {

                        printf("error %d submitting rx desc\n", ret);

                        break;

                }

        }

        return 0;

}
```

其中重点关注下cpsw_update_link(priv)-> cpsw_slave_update_link(slave, priv, &link),这个函数会根据根据现有的priv->phydev设备发起phy_startup（在LINK的状态下解析phydev->speed、phydev->duplex等状态），之后根据phy_startupde 的结果更新mac_control，最终通过此函数_raw_writel(mac_control, &slave->sliver->mac_control)将mac_control写入到相关cpsw_priv 的slave->sliver->mac_control寄存器。只有在link状态下正确配置了slave->sliver->mac_control寄存器,才能与phy正常进行通信。

以上配置好后，就可以在后续使用PING命令进行测试，PING命令使用之前，还需要配置好IP地址，可使用环境变量进行配置，如setenv ipaddr 192.168.1.30。