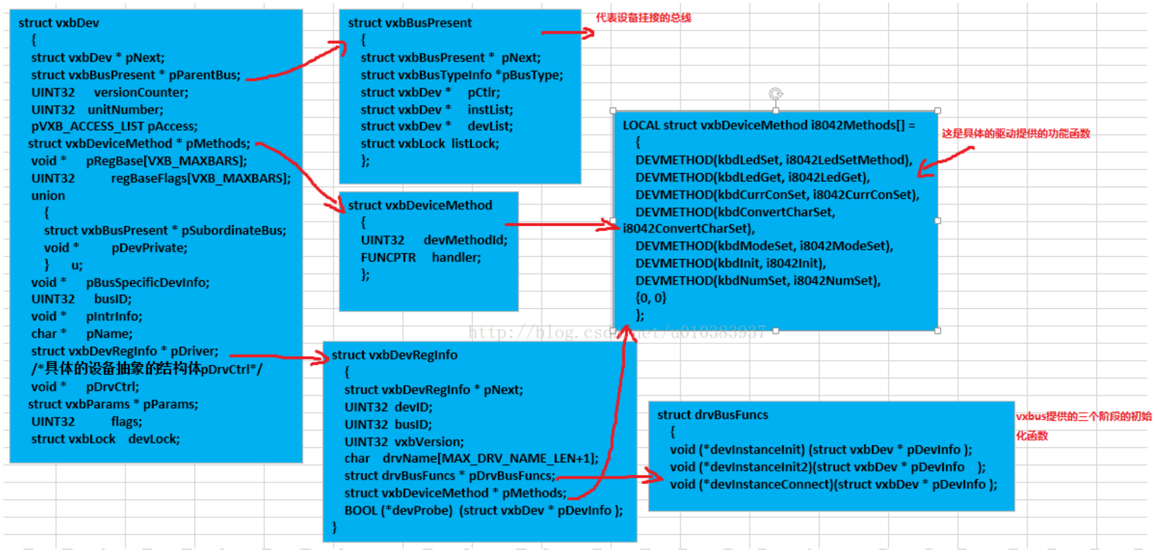
VxWorks 系统中 vxbus 机制总结

本文介绍一下 VxWorks 的设备以及驱动的表示方法，以及总的关系：设备和驱动根本都抽象成一个结构体，设备结构体中包含了设备名字、ID 号、功能函数指针等必备的信息，驱动结构体包含了初始化函数、名字、ID 等信息。总的如图看一下重要的几个结构体的关系：



```
struct vxbDev
   {
   struct vxbDev * pNext;
   struct vxbBusPresent * pParentBus;
   UINT32    versionCounter;
   UINT32   unitNumber;
   pVXB_ACCESS_LIST pAccess;
  struct vxbDeviceMethod * pMethods;
   void *     pRegBase[VXB_MAXBARS];
   UINT32         regBaseFlags[VXB_MAXBARS];
   union
      {
      struct vxbBusPresent * pSubordinateBus;
      void *         pDevPrivate;
      }    u;
   void *     pBusSpecificDevInfo;
   UINT32     busID;
   void *     pIntrInfo;
   char *     pName;
   struct vxbDevRegInfo * pDriver;
   /*具体的设备抽象的结构体pDrvCtrl*/
   void *     pDrvCtrl;
  struct vxbParams * pParams;
   UINT32         flags;
   struct vxbLock    devLock;
```

```
struct vxbBusPresent
   {
   struct vxbBusPresent *  pNext;
   struct vxbBusTypeInfo *pBusType;
   struct vxbDev *     pCtlr;
   struct vxbDev *     instList;
   struct vxbDev *     devList;
   struct vxbLock  listLock;
   };
```

```
struct vxbDeviceMethod
   {
   UINT32     devMethodId;
   FUNCPTR     handler;
   };
```

```
LOCAL struct vxbDeviceMethod i8042Methods[] =
   {
   DEVMETHOD(kbdLedSet, i8042LedSetMethod),
   DEVMETHOD(kbdLedGet, i8042LedGet),
   DEVMETHOD(kbdCurrConSet, i8042CurrConSet),
   DEVMETHOD(kbdConvertCharSet,
i8042ConvertCharSet),
   DEVMETHOD(kbdModeSet, i8042ModeSet),
   DEVMETHOD(kbdInit, i8042Init),
   DEVMETHOD(kbdNumSet, i8042NumSet),
   {0, 0}
   };
```

```
struct vxbDevRegInfo
   {
   struct vxbDevRegInfo * pNext;
   UINT32  devID;
   UINT32  busID;
   UINT32  vxbVersion;
   char    drvName[MAX_DRV_NAME_LEN+1];
   struct drvBusFuncs * pDrvBusFuncs;
   struct vxbDeviceMethod * pMethods;
   BOOL (*devProbe)  (struct vxbDev * pDevInfo );
}
```

```
struct drvBusFuncs
   {
   void (*devInstanceInit) (struct vxbDev * pDevInfo );
   void (*devInstanceInit2)(struct vxbDev * pDevInfo    );
   void (*devInstanceConnect)(struct vxbDev * pDevInfo );
```

vxbus 结构设计了几个链表：

```
/*放置驱动的链表*/
struct vxbDevRegInfo * pDriverListHead = NULL;
/*放置注册的总线的链表*/
struct vxbBusTypeInfo * pBusListHead = NULL;
/*匹配好的设备和驱动称为instlist，没有找到驱动的设备链表，*/
struct vxbBusPresent * pBusHead = NULL;
```

当一个设备注册进来之后，就会从 pDriverListHead 中查找驱动，如果找到了就会放到 pBusHead 链表中的 instList 链表中，没有找到驱动就会放置到 pBusHead->devList 中；而当一个新的总线注册进来后就会放置到 pBusListHead 链表中。

vxbus 的初始化：

vxbus 的初始化流程中的函数调用：

```
usrInit
    sysHwInit
        hardWareInterFaceInit
                hardWareInterFaceBusInit
                        vxbLibInit (vxbInitPhase = 0)
                        plbRegister
                        vxbInit(vxbInitPhase = 1)
                            plbInit1
    usrRoot
        sysClkInit
                sysClkConnect
                        sysHwInit2
                             vxbDevInit
                                vxbDevInitInternal (vxbInitPhase = 2)
                                        vxbDevInit2Helper(对所有设备进行)
```

```
  vxbDevConnect
  vxbDevConnectInternal(vxbInitPhase = 3)
```

```
vxbDevConnectHelper(对所有设备进行)
```

vxbus 的初始化分为按照 vxbInitPhase=0,1,2,3 分为 4 部分进行，接下来分别进行介绍。

vxbInitPhase=0 时：函数 vxbLibInit 所做工作"

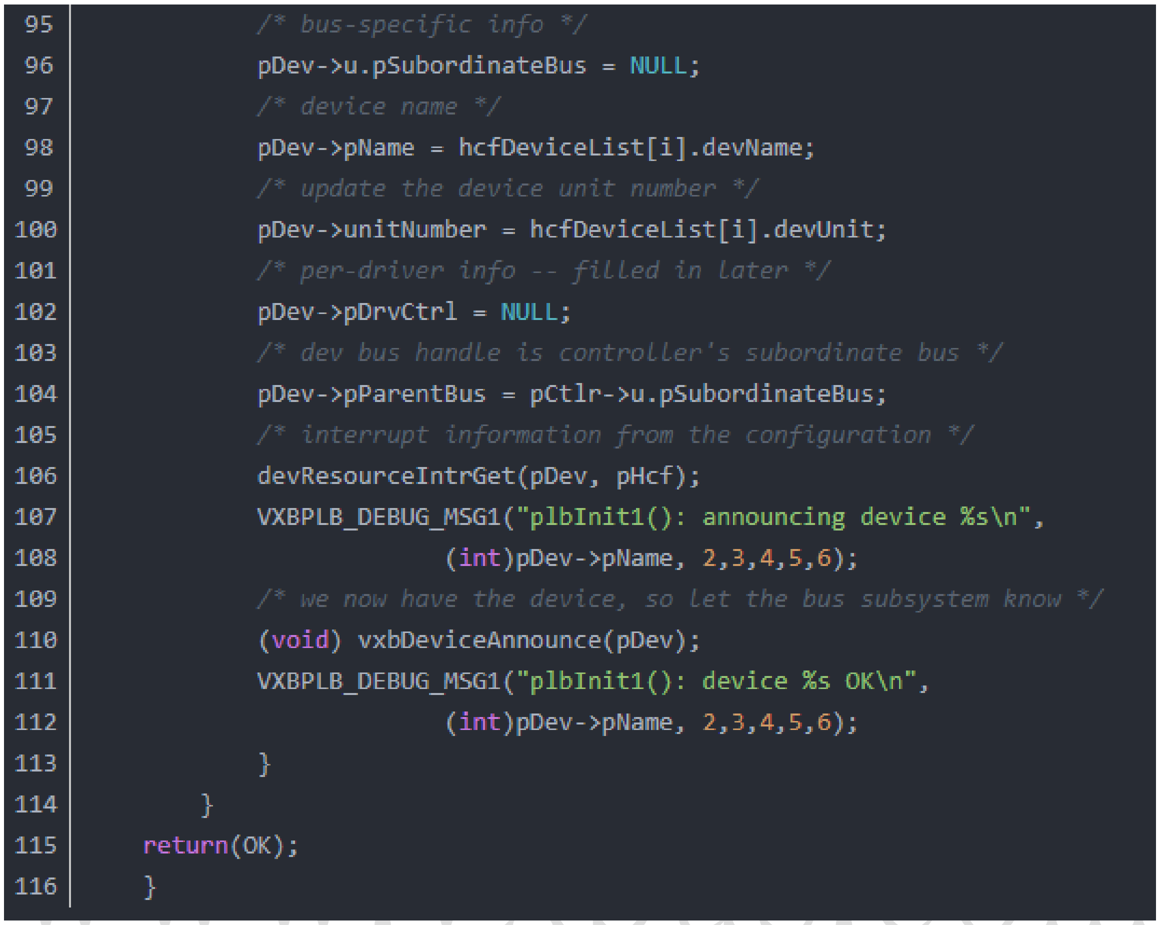
```
STATUS vxbLibInit (void)
    {
    vxbInitPhase = 0;
    return(OK);
    }
```

只是声明了这是 vxbInitPhase=0 阶段，之后就是驱动函数的注册。也就是说 0 阶段只要是驱动函数的注册。

vxbInitPhase=1：

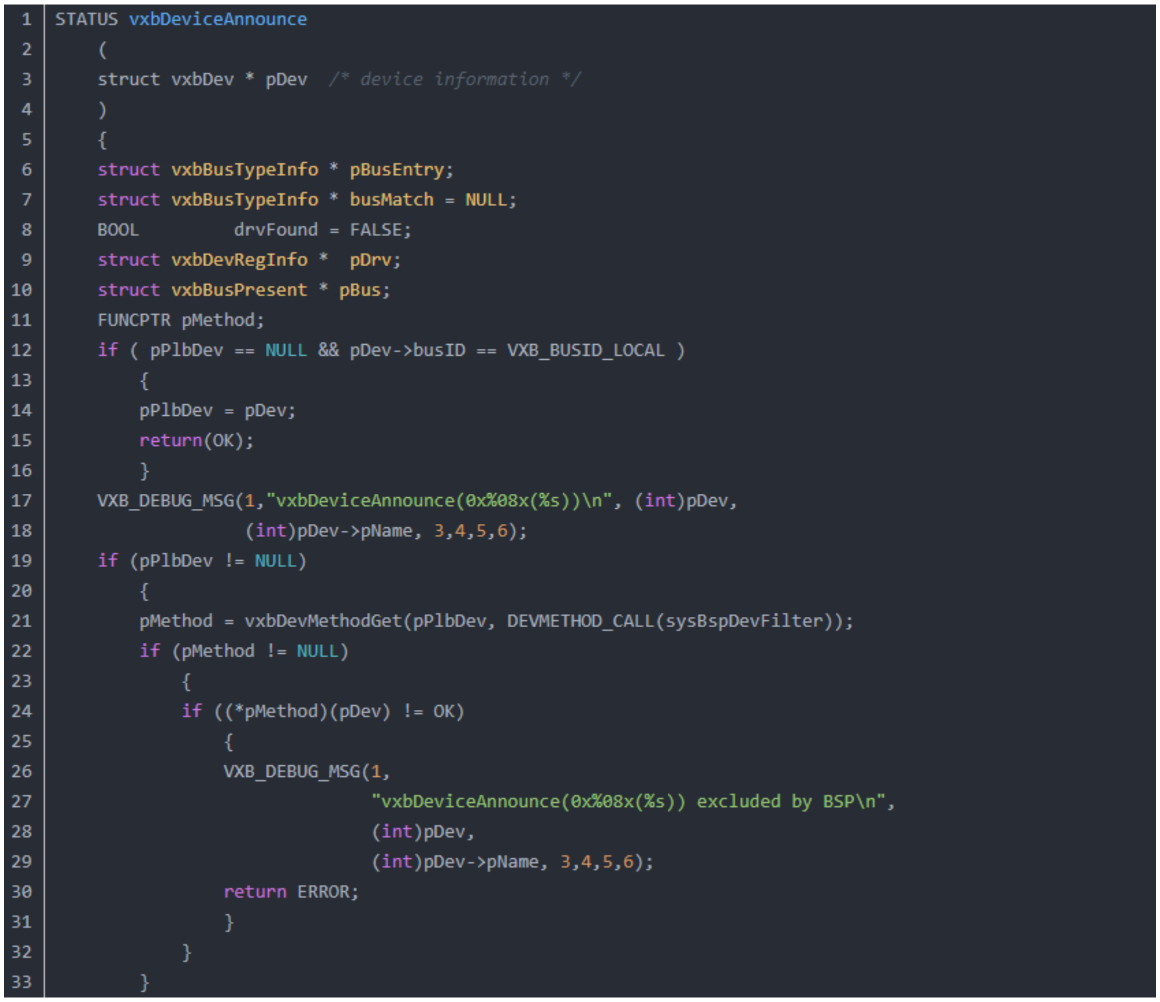
```
STATUS vxbInit (void)
    {
    vxbInitPhase = 1;

    plbInit1(pPlbDev);
    return(OK);
    }
```

```
STATUS plbInit1
    (
    struct vxbDev * pCtlr
    )
    {
    int i, j;
    char regBase[] = "regBase0";
    struct vxbDev * pDev;
    HCF_DEVICE *    pHcf;

    VXBPLB_DEBUG_MSG1("plbInit1() called\n", 1,2,3,4,5,6);

#ifdef  VXB_LEGACY_ACCESS
    /* call the function to initialize the access module of PLB */
    plbAccessInit ();
#endif  /* VXB_LEGACY_ACCESS */

    /*
     * now populate the bus with devices from the table created by the
     * configuration tool
     */
    for ( i = 0 ; i < hcfDeviceNum ; i++ )
        {
        if ( hcfDeviceList[i].busType != VXB_BUSID_PLB )
            {
            VXBPLB_DEBUG_MSG1("plbInit1(): skipping non-PLB device %s\n",
                             (int)hcfDeviceList[i].devName, 2,3,4,5,6);
            }
        else
            {
            VXBPLB_DEBUG_MSG1("plbInit1(): processing device %s\n",
                             (int)hcfDeviceList[i].devName, 2,3,4,5,6);
            pHcf = &hcfDeviceList[i];
            /* allocate device structure */
            pDev = vxbDevStructAlloc(WAIT_FOREVER);
            if ( pDev == NULL )
                {
                return(ERROR);
                }
            /* bus subsystem overhead */
            pDev->busID = VXB_BUSID_PLB;
            pDev->pNext = NULL;
            /* save resources with device */
            pDev->pBusSpecificDevInfo = (void *)&hcfDeviceList[i];
            /* access functions */
#ifdef  VXB_LEGACY_ACCESS
            pDev->pAccess = (pVXB_ACCESS_LIST)hwMemAlloc(sizeof(VXB_ACCESS_LIST));
            if ( pDev->pAccess == NULL )
                {
                /* hwMemFree((char *)pDev); */
                vxbDevStructFree(pDev);
                return(ERROR);
                }
            /* copy the arch specific access function pointers */
            vxbPlbAccessCopy (pDev->pAccess);
#endif  /* VXB_LEGACY_ACCESS */
            /* nothing needs to be done if the return value is not ok */
            /* assign register base address */
           for (j = 0; j < VXB_MAXBARS; j++)
                {
                pDev->pRegBase[j] = 0;
                /*
                 * Decode all BARs labeled "regBase0" to "regBase9." For
                 * backwards compatibility, "regBase0" and "regBase" are
                 * equivalent.
                 */
                regBase[7] = '0'+j; /* Avoid use of sprintf()/strcat()/etc) */
                /*
                 * resourceDesc {
                 * The regBaseN resources specify up to 10
                 * base addresses for device registers (N = [0-9]). }
                 */
                if (devResourceGet (pHcf, regBase, HCF_RES_INT,
                    (void *)&(pDev->pRegBase[j])) != OK && j == 0)
                /*
                 * resourceDesc {
                 * The regBase resource is synonymous with regBase0. }
                 */
                    devResourceGet (pHcf, "regBase", HCF_RES_INT,
                        (void *)&(pDev->pRegBase[j]));
                /*
                 * On x86, local bus devices are assumed to be
                 * accessed using I/O space registers. Everywhere
                 * else, registers are simply memory mapped.
                 */
                if (pDev->pRegBase[j] == NULL)
                    pDev->regBaseFlags[j] = VXB_REG_NONE;
                else
#if (CPU_FAMILY == I80X86)
                    pDev->regBaseFlags[j] = VXB_REG_IO;
#else
                    pDev->regBaseFlags[j] = VXB_REG_MEM;
#endif
                }
```



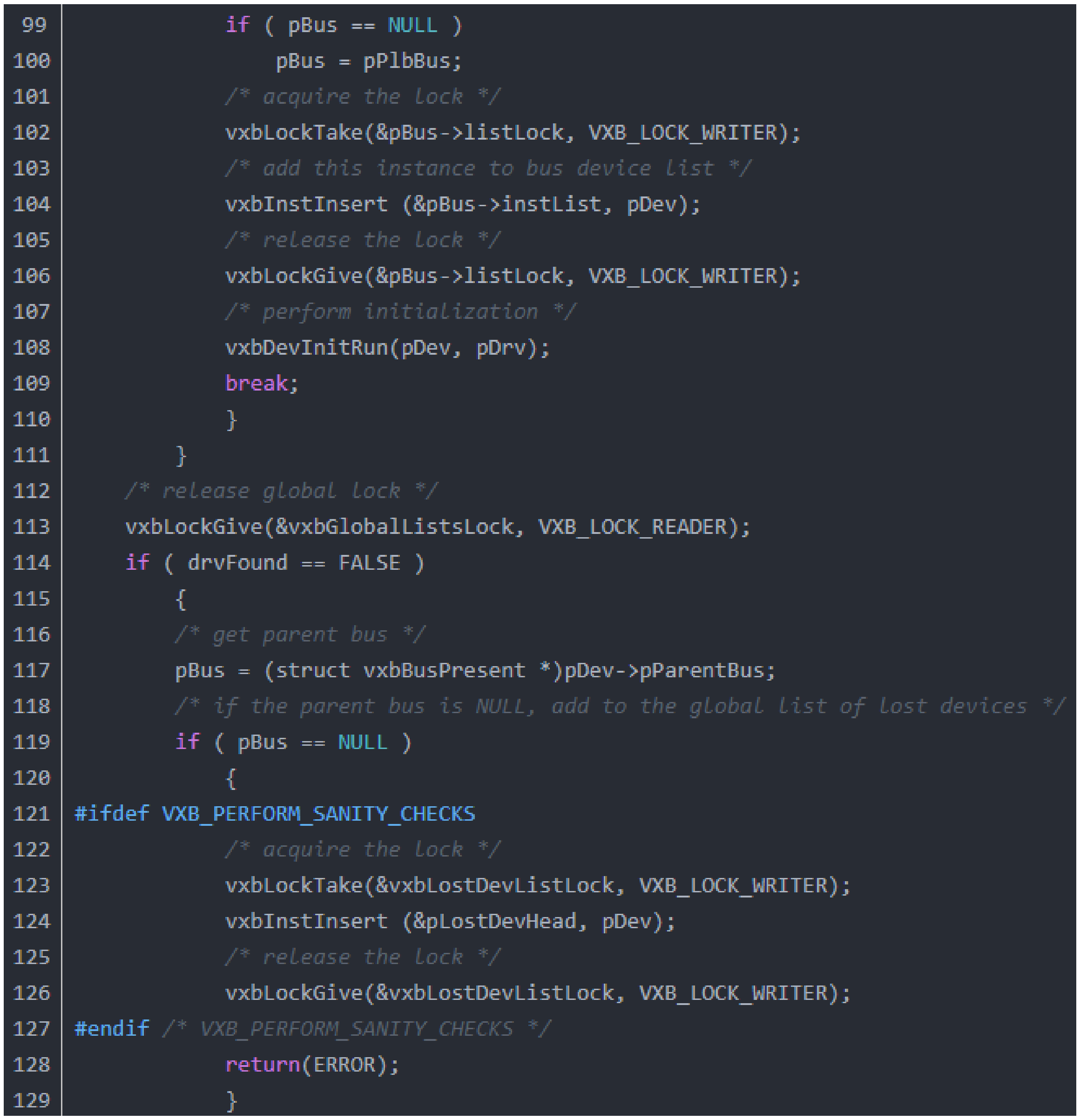
```
            /* bus-specific info */
            pDev->u.pSubordinateBus = NULL;
            /* device name */
            pDev->pName = hcfDeviceList[i].devName;
            /* update the device unit number */
            pDev->unitNumber = hcfDeviceList[i].devUnit;
            /* per-driver info -- filled in later */
            pDev->pDrvCtrl = NULL;
            /* dev bus handle is controller's subordinate bus */
            pDev->pParentBus = pCtlr->u.pSubordinateBus;
            /* interrupt information from the configuration */
            devResourceIntrGet(pDev, pHcf);
            VXBPLB_DEBUG_MSG1("plbInit1(): announcing device %s\n",
                         (int)pDev->pName, 2,3,4,5,6);
            /* we now have the device, so let the bus subsystem know */
            (void) vxbDeviceAnnounce(pDev);
            VXBPLB_DEBUG_MSG1("plbInit1(): device %s OK\n",
                         (int)pDev->pName, 2,3,4,5,6);
            }
        }
    return(OK);
    }
```

这个函数所做的主要工作就是对系统中所定义的设备进行结构体的构建，也就是为每个设备抽象一个结构体，并按照所定义的信息对结构体进行填充。

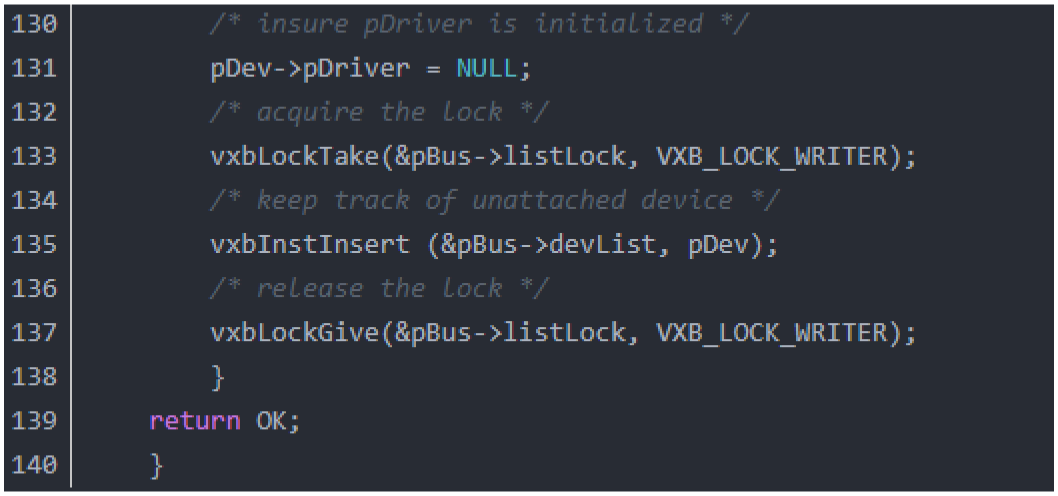


```
STATUS vxbDeviceAnnounce
    (
    struct vxbDev * pDev  /* device information */
    )
    {
    struct vxbBusTypeInfo * pBusEntry;
    struct vxbBusTypeInfo * busMatch = NULL;
    BOOL         drvFound = FALSE;
    struct vxbDevRegInfo *  pDrv;
    struct vxbBusPresent * pBus;
    FUNCPTR pMethod;
    if ( pPlbDev == NULL && pDev->busID == VXB_BUSID_LOCAL )
        {
        pPlbDev = pDev;
        return(OK);
        }
    VXB_DEBUG_MSG(1,"vxbDeviceAnnounce(0x%08x(%s))\n", (int)pDev,
                  (int)pDev->pName, 3,4,5,6);
    if (pPlbDev != NULL)
        {
        pMethod = vxbDevMethodGet(pPlbDev, DEVMETHOD_CALL(sysBspDevFilter));
        if (pMethod != NULL)
            {
            if ((*pMethod)(pDev) != OK)
                {
                VXB_DEBUG_MSG(1,
                              "vxbDeviceAnnounce(0x%08x(%s)) excluded by BSP\n",
                              (int)pDev,
                              (int)pDev->pName, 3,4,5,6);
                return ERROR;
                }
            }
        }
    /* acquire global lock as reader */
    vxbLockTake(&vxbGlobalListsLock, VXB_LOCK_READER);
    for ( pBusEntry = pBusListHead ; pBusEntry != NULL ;
          pBusEntry = pBusEntry->pNext )
        {
        /* check for matching bus type */
        if ( pBusEntry->busID != pDev->busID )
            continue;
        for ( pDrv = pDriverListHead ; pDrv != NULL ;
             pDrv = pDrv->pNext )
            {
            VXB_DEBUG_MSG(1,"vxbDeviceAnnounce(): checking 0x%08x (%s) "
                            "against %s\n",
                        (int)pDev, (int)pDev->pName,
                        (int)&pDrv->drvName[0], 4,5,6);
            if ( pDrv->busID != pDev->busID )
                {
                VXB_DEBUG_MSG(1,"vxbDeviceAnnounce(): "
                                "%s@%p failed type check\n",
                            (int)pDev->pName, (int)pDev,
                            3,4,5,6);
                continue;
                }
            /* check bus-specific match routine */
            drvFound = (*(pBusEntry->vxbDevMatch))(pDrv, pDev);
            if ( ! drvFound )
                {
                VXB_DEBUG_MSG(1,"vxbDeviceAnnounce(): "
                                "%s@%p failed bus match\n",
                            (int)pDev->pName, (int)pDev,
                            3,4,5,6);
                continue;
                }
```

```c
busMatch = pBusEntry;
/* check driver-supplied probe routine */
if ( pDrv->devProbe == NULL )
    {
    VXB_DEBUG_MSG(1,"vxbDeviceAnnounce(): "
                    "no driver probe available\n",
                  1,2,3,4,5,6);
    drvFound = TRUE;
    }
else
    {
    VXB_DEBUG_MSG(1, "vxbDeviceAnnounce(): "
                    "calling driver probe\n",
                  1,2,3,4,5,6);
    drvFound = (*(pDrv->devProbe))(pDev);
    if ( drvFound == FALSE )
        {
        VXB_DEBUG_MSG(1, "vxbDeviceAnnounce(): "
                        "driver probe failed\n",
                      1,2,3,4,5,6);
        continue;
        }
    }
VXB_DEBUG_MSG(1, "vxbDeviceAnnounce(): "
                "found match, driver @ %p\n",
              (int)pDrv, 2,3,4,5,6);
/* attach driver registration info */
pDev->pDriver = pDrv;
/* adjust name */
pDev->pName = &pDrv->drvName[0];
/* get parent bus */
pBus = (struct vxbBusPresent *)pDev->pParentBus;
```



```
            if ( pBus == NULL )
                pBus = pPlbBus;
            /* acquire the lock */
            vxbLockTake(&pBus->listLock, VXB_LOCK_WRITER);
            /* add this instance to bus device list */
            vxbInstInsert (&pBus->instList, pDev);
            /* release the lock */
            vxbLockGive(&pBus->listLock, VXB_LOCK_WRITER);
            /* perform initialization */
            vxbDevInitRun(pDev, pDrv);
            break;
            }
        }
    /* release global lock */
    vxbLockGive(&vxbGlobalListsLock, VXB_LOCK_READER);
    if ( drvFound == FALSE )
        {
        /* get parent bus */
        pBus = (struct vxbBusPresent *)pDev->pParentBus;
        /* if the parent bus is NULL, add to the global list of lost devices */
        if ( pBus == NULL )
            {
#ifdef VXB_PERFORM_SANITY_CHECKS
            /* acquire the lock */
            vxbLockTake(&vxbLostDevListLock, VXB_LOCK_WRITER);
            vxbInstInsert (&pLostDevHead, pDev);
            /* release the lock */
            vxbLockGive(&vxbLostDevListLock, VXB_LOCK_WRITER);
#endif /* VXB_PERFORM_SANITY_CHECKS */
            return(ERROR);
            }
```



```
        /* insure pDriver is initialized */
        pDev->pDriver = NULL;
        /* acquire the lock */
        vxbLockTake(&pBus->listLock, VXB_LOCK_WRITER);
        /* keep track of unattached device */
        vxbInstInsert (&pBus->devList, pDev);
        /* release the lock */
        vxbLockGive(&pBus->listLock, VXB_LOCK_WRITER);
        }
    return OK;
    }
```

每当一个设备的结构体初始化完成时，也就是一个设备构建完成，这时调用 vxbDeviceAnnounce，告诉系统有新设备了。vxbDeviceAnnounce 的功能就是在驱动链表中进行查找，看有没有与当前设备匹配的驱动，分别比对 name,之后比对 ID，如果相同就匹配完成，放到 pBusHead->instList 链表中，如果没有驱动就放到 pBusHead->devlist 中，之后调用函数 vxbDevInitRun：

```c
LOCAL void vxbDevInitRun
    (
    VXB_DEVICE_ID devID,
    struct vxbDevRegInfo * pDrv
    )
    {
    /* first pass */
    if (!(devID->flags & VXB_INST_INIT_DONE))
        {
        if ( pDrv->pDrvBusFuncs->devInstanceInit != NULL )
            (*(pDrv->pDrvBusFuncs->devInstanceInit))(devID);
        devID->flags |= VXB_INST_INIT_DONE;
        }
    /* second pass */
    if (vxbInitPhase >= 2 && !(devID->flags & VXB_INST_INIT2_DONE))
        {
        if ( pDrv->pDrvBusFuncs->devInstanceInit2 != NULL )
            (*(pDrv->pDrvBusFuncs->devInstanceInit2))(devID);
        devID->flags |= VXB_INST_INIT2_DONE;
        }
    /* third pass */
    if (vxbInitPhase >= 3 && !(devID->flags & VXB_INST_CONNECT_DONE))
        {
        if ( pDrv->pDrvBusFuncs->devInstanceConnect != NULL )
            (*(pDrv->pDrvBusFuncs->devInstanceConnect))(devID);
        devID->flags |= VXB_INST_CONNECT_DONE;
        }
    }
```

vxbDevInitRun 的作用就是分别调用设备的初始化函数，对设备进行初始化。

这时第一阶段的初始化就完成了，完成工作：大部分设备和驱动已经进行了匹配，并放入到相应的链表中，并且匹配好的设备进行了第一阶段的初始化。但是此时请注意了，因为部分设备没有匹配到驱动，放入到了 pBusHead->devlist 中，那这个设备也就没有进行初始化操作。所以之后的第二第三阶段的初始化操作，主要是对这些没有匹配成功的设备进行的。

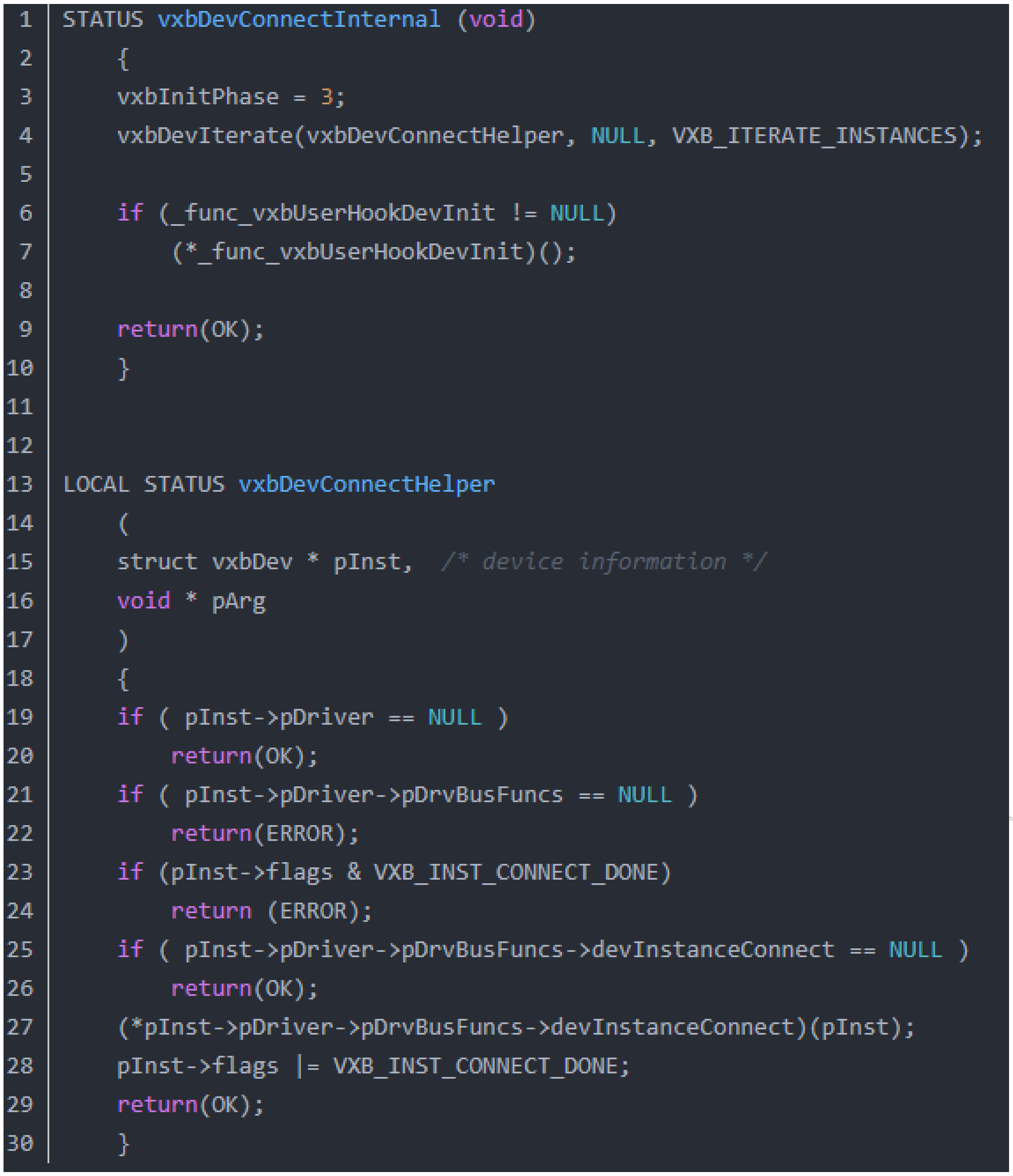
vxbInitPhase=2 时：

```c
STATUS vxbDevInitInternal (void)
    {
    vxbInitPhase = 2;
    pVxbSpinLockTake = spinLockIsrTake;
    pVxbSpinLockGive = spinLockIsrGive;
    vxbLockInit(&vxbGlobalListsLock);
    vxbLockInit(&vxbBusListLock);
#ifdef VXB_PERFORM_SANITY_CHECKS
    vxbLockInit(&vxbLostDevListLock);
#endif /* VXB_PERFORM_SANITY_CHECKS */
    vxbLockInit(&vxbDevStructListLock);

    /* execute init phase 2 for all devices */
    vxbDevIterate(vxbDevInit2Helper, NULL, VXB_ITERATE_INSTANCES);
    return(OK);
    }
```

```c
LOCAL STATUS vxbDevInit2Helper
    (
    struct vxbDev * pInst,
    void * pArg
    )
    {
    if ( pInst->pDriver == NULL )
        return(OK);
    if ( pInst->pDriver->pDrvBusFuncs == NULL )
        return(ERROR);
    if (pInst->flags & VXB_INST_INIT2_DONE)
        return (ERROR);
    if ( pInst->pDriver->pDrvBusFuncs->devInstanceInit2 == NULL )
        return(OK);
    (*pInst->pDriver->pDrvBusFuncs->devInstanceInit2)(pInst);
    pInst->flags |= VXB_INST_INIT2_DONE;
    return(OK);
    }
```

此时是对总线 plb 上所有的设备进行初始化操作，也就是会对那些没有初始化的也进行初始化。

vxbInitPhase=3 时：



```
STATUS vxbDevConnectInternal (void)
    {
    vxbInitPhase = 3;
    vxbDevIterate(vxbDevConnectHelper, NULL, VXB_ITERATE_INSTANCES);

    if (_func_vxbUserHookDevInit != NULL)
        (*_func_vxbUserHookDevInit)();

    return(OK);
    }


LOCAL STATUS vxbDevConnectHelper
    (
    struct vxbDev * pInst,  /* device information */
    void * pArg
    )
    {
    if ( pInst->pDriver == NULL )
        return(OK);
    if ( pInst->pDriver->pDrvBusFuncs == NULL )
        return(ERROR);
    if (pInst->flags & VXB_INST_CONNECT_DONE)
        return (ERROR);
    if ( pInst->pDriver->pDrvBusFuncs->devInstanceConnect == NULL )
        return(OK);
    (*pInst->pDriver->pDrvBusFuncs->devInstanceConnect)(pInst);
    pInst->flags |= VXB_INST_CONNECT_DONE;
    return(OK);
    }
```

第三阶段也是对所有的设备进行的。

这样经过三个阶段的初始化之后，所有的设备都进行了初始化。vxbus 也就初始化完成了。