Receive and Send Packets on xCat 研究报告

交换机 刘家宁

版 本 历 史

版本/状态	责任人	起止日期	备注
V1.0/草稿	刘家宁	Nov16	创建文档

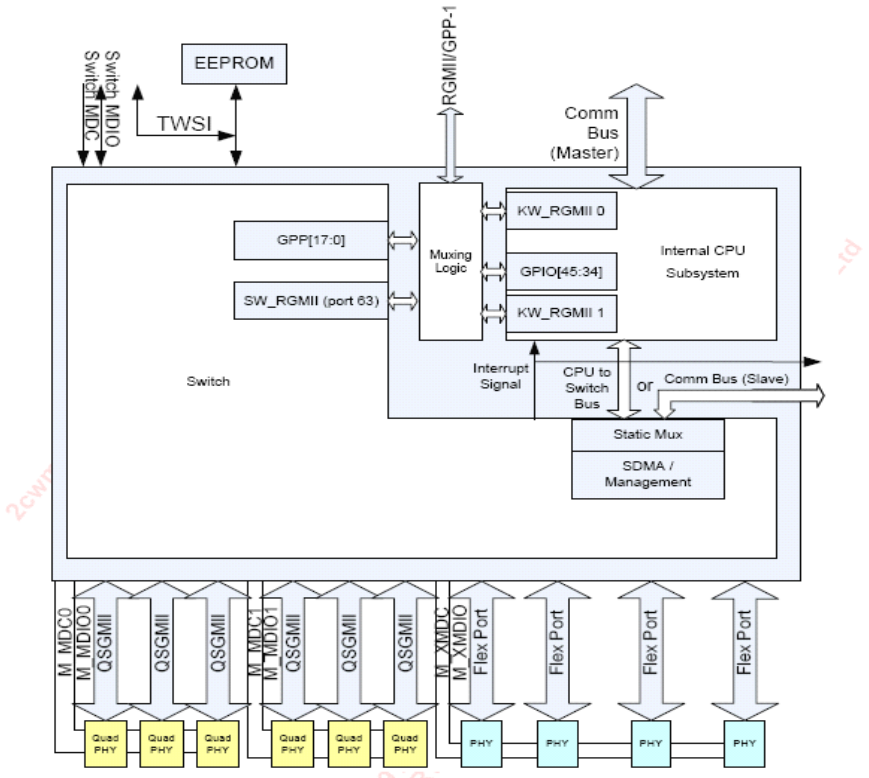
目 录

1. 项目背景

xCat 是 Marvell 公司的一款为多层网络交换设计的，新一代的芯片系列。它有多个子系列组成：AlleyCat、TomCat 等。xCat 芯片为百兆/千兆以太网的交换机提供了良好的包处理能力、较低的功耗、高度集成减少体积的设计。它集成了一个包处理器和一个提供管理功能的 CPU 子系统。CPU 子系统的设计基于 Marvell SheevaTM CPU core，与 ARMv5TE 的体系结构完全兼容。包处理器由以下几个部分组成：

- 24 个百兆或千兆 mac 端口
- 4 个 Flex-Stack ports 端口
- 支持 12 Gbps HyperG Stack
- 片上缓冲区
- policy 引擎、二层交换引擎
- 基于 policy 的 IPv4/6 单播路由
- SCT
- DSA

CPU 子系统与包处理器的 mac 芯片之间没有采用网口和 MII 总线，而是通过 PCI 总线连接，通过 SDMA 在两者之间传输数据。这种设计提高了 CPU 的处理效率，同时使系统更具可拓展性。

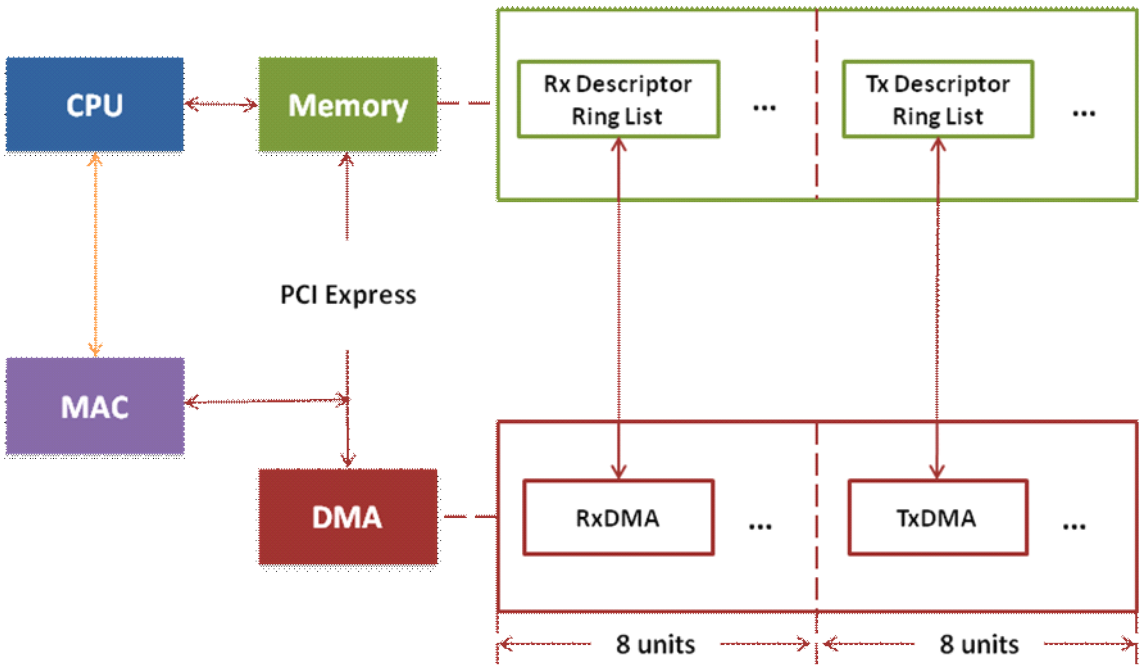


1- 1 TomCat-GE Host Management Interface

2. xCat 的 DMA 功能

2.1 xCat 的 DMA 设计

设备为交换功能集成了 16 个 DMA 控制器，用于 device memory 和 CPU memory 之间传输数据。16 个 DMA 控制器分为两组，分别是 8 个 RxDMAs 和 8 个 TxDMAs，每一个关联到一个 Descriptor Ring List。这里的 DMA 控制器，称为 Serial DMAs (SDMA)，因为每个 DMA 在软件层都是工作在一个存储着描述符的线性的链表上。SDMA 使用 PCI 总线与 CPU memory 进行通信，每次最多传输 128 Bytes，最小传输 8 Bytes。



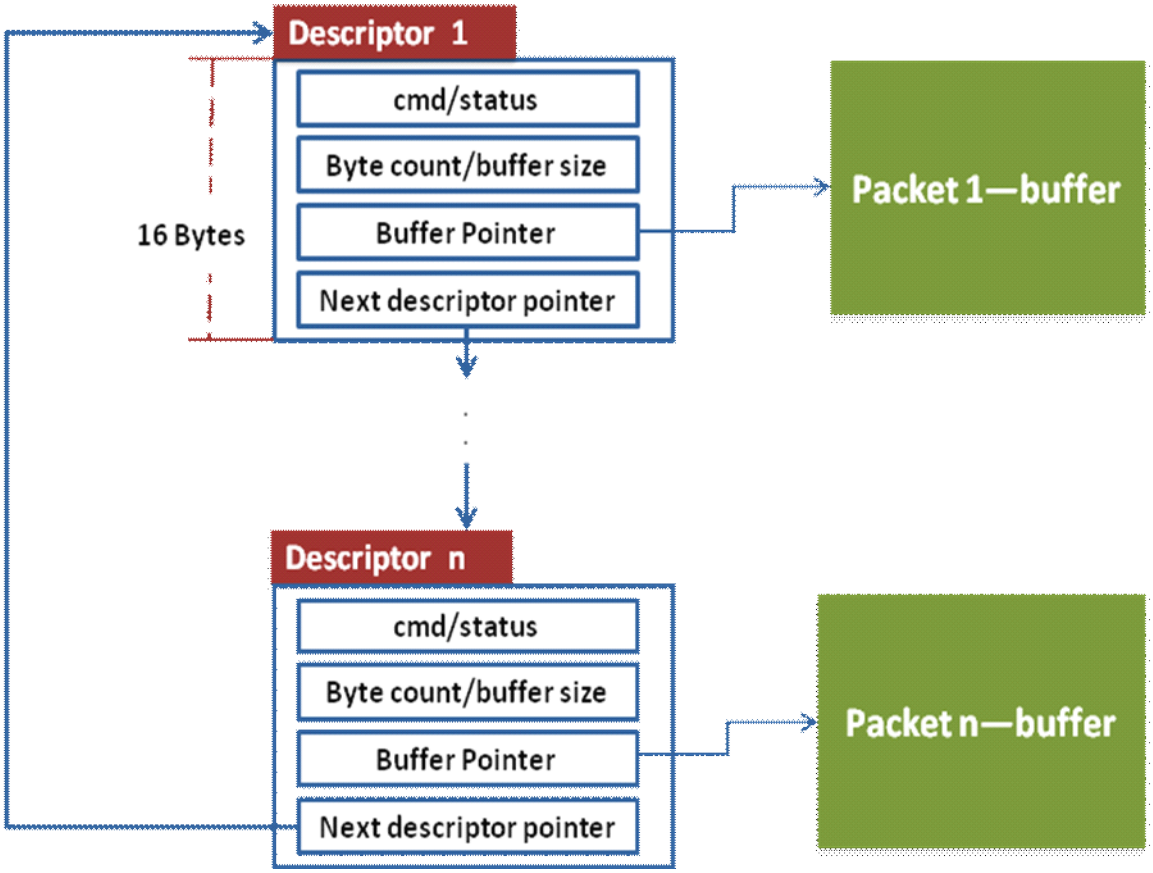
2- 1 DMA 传输的示意图

2.2 Descriptor Ring List 的配置

Descriptor Ring List 的优先级调度算法和它的数量可以在驱动程序中设置，它们依据具体芯片和应用场合的不同，而不同。在笔者的这款芯片中，有 1 个 RxDMA 和 8 个 TxDMA，与此对应，驱动中分别设置了 1 个接收的和 8 个发送的 Descriptor Ring List。下面是系统关于 Descriptor Ring List 数量的宏定义和系统用于管理接收和发送的 Descriptor Ring List 的结构体。

```
#define NUM_OF_RX_QUEUES     (1)
#define NUM_OF_TX_QUEUES     (8)

typedef struct
{
    RX_DESC_LIST     rxDescList[NUM_OF_RX_QUEUES];
    TX_DESC_LIST     txDescList[NUM_OF_TX_QUEUES];
    AU_DESC_CTRL     auDescCtrl;
    AU_DESC_CTRL     fuDescCtrl;

}SDMA_INTERRUPT_CTRL;
```

Descriptor Ring List 的结构如下图所示。描述符环形链表的节点由两部分组成：描述符和缓存。描述符的长度为 16Bytes，要求 16-byte 对齐(i.e.Descriptor_Address[3:0]==0000).。它包含有 4 个字段，分别是命令和状态字段，缓存大小字段，缓存指针字段，下一个描述符字段。命令和状态字段包含有很多关于数据包的状态信息，如描述符的所有权，数据包的传输过程是否发生错误，描述符是包含多个描述符的数据包的第一个描述符，描述符是包含多个描述符的数据包的最后一个描述符，等等。缓存指针字段指向数据包所在的缓存，而缓存大小字段保存了这段内存区域的大小。数据包所在的缓存最大为 16KB，要求是 128-byte 对齐(i.e., Buffer_Address[6:0]==7'b0000000)。



2- 2 Descriptor Ring List 的结构

下面是源码中用于接收的描述符环形链表的定义。

```
typedef struct
{
    STRUCT_SW_RX_DESC    *next2Return;
    STRUCT_SW_RX_DESC    *next2Receive;

    MV_U32               freeDescNum;
    MV_SEM               rxListSem;
    MV_BOOL              forbidQEn;
}RX_DESC_LIST;
```

xCat 驱动的源码中，接收描述符环形链表的节点不是 2-2 图中的描述符。它增加了一个管理接收描述符的结构，这个结构是链表真正的节点，它包含了一个指向描述符的指针。管理接收描述符的结构如下所示。

```
typedef struct _swRxDesc
{
    STRUCT_RX_DESC      *rxDesc;
    struct _swRxDesc    *swNextDesc;
  volatile MV_U32     buffVirtPointer;

    STRUCT_RX_DESC      shadowRxDesc;
}STRUCT_SW_RX_DESC;
```

接收描述符的定义如下。

```
typedef struct _rxDesc
{
    volatile MV_U32         word1;
    volatile MV_U32         word2;

    volatile MV_U32         buffPointer;
    volatile MV_U32         nextDescPointer;

}STRUCT_RX_DESC;
```

下面是源码中用于发送的描述符环形链表的定义。

```
typedef struct
{
    STRUCT_SW_TX_DESC   *next2Free;
    STRUCT_SW_TX_DESC   *next2Feed;

    MV_U32              freeDescNum;
    MV_SEM              txListSem;

    MV_BOOL             freeCoreBuff;
    MV_U32              sizeOfList;
}TX_DESC_LIST;
```

同管理接收描述符的结构相识，管理发送描述符的结构定义如下。

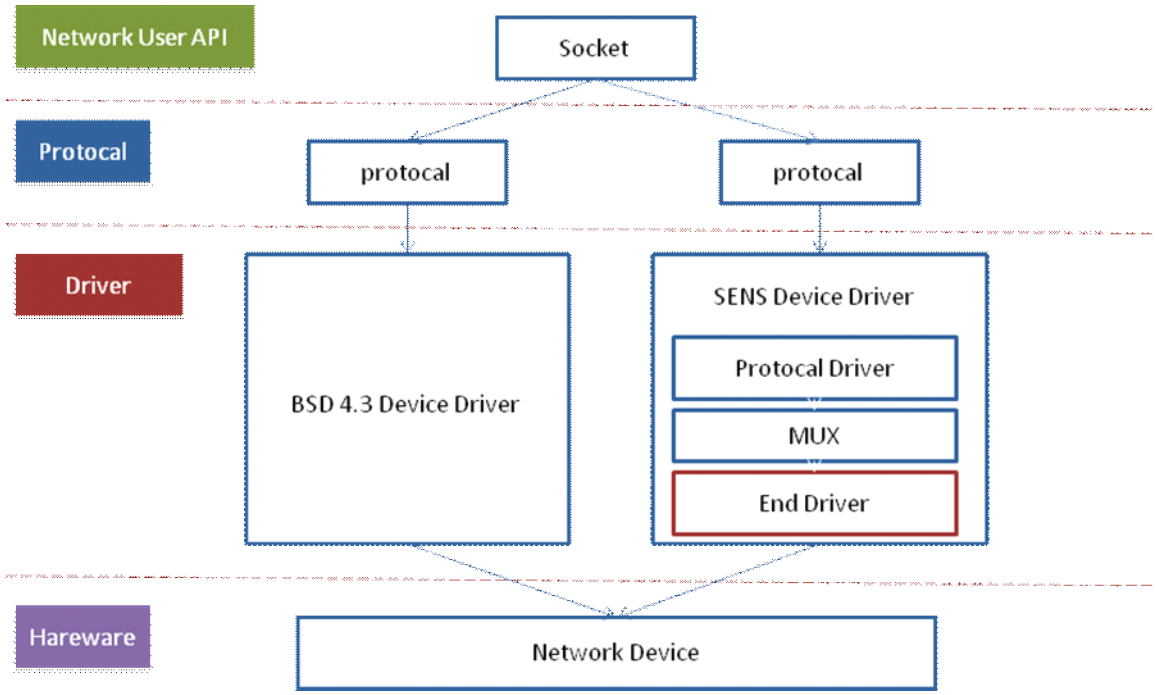
```
typedef struct _swTxDesc
{
    STRUCT_TX_DESC      *txDesc;
    MV_U8               txBufferAligment;
    struct _swTxDesc    *swNextDesc;
    volatile MV_U32     virtBuffPointer;
    MV_PTR              userData;
}STRUCT_SW_TX_DESC;
```

发送描述符的定义如下。

```
typedef struct _txDesc
{
    volatile MV_U32         word1;
    volatile MV_U32         word2;

    volatile MV_U32         buffPointer;
    volatile MV_U32         nextDescPointer;
}STRUCT_TX_DESC;
```

3. VxWorks 网络体系结构

网络设备是一种特殊的字符设备。网络设备并不直接与用户打交道，而是通过多层协议与应用程序交换信息。VxWorks 的网络体系结构可以分为 4 层，如图所示。在发送数据时，协议层通过提供给用户的网络程序编程接口(如：Socket)来获得应用程序的数据。数据根据使用到的协议，在协议栈中进行处理和封装。然后由协议层将数据传送给驱动层，也就是网络设备驱动程序。驱动层有三个子层次，分别是协议驱动层、MUX 层和 END 层。协议驱动层充当驱动层和协议层的接口。MUX 层管理着网络协议接口和底层 END 层程序的通信，使发送、接收过程变得简单、而不再需要通过挂接钩子程序的办法。END 层是驱动层的核心。它操作具体硬件，实现将数据发送出去。



3- 1 VxWorks 网络体系结构

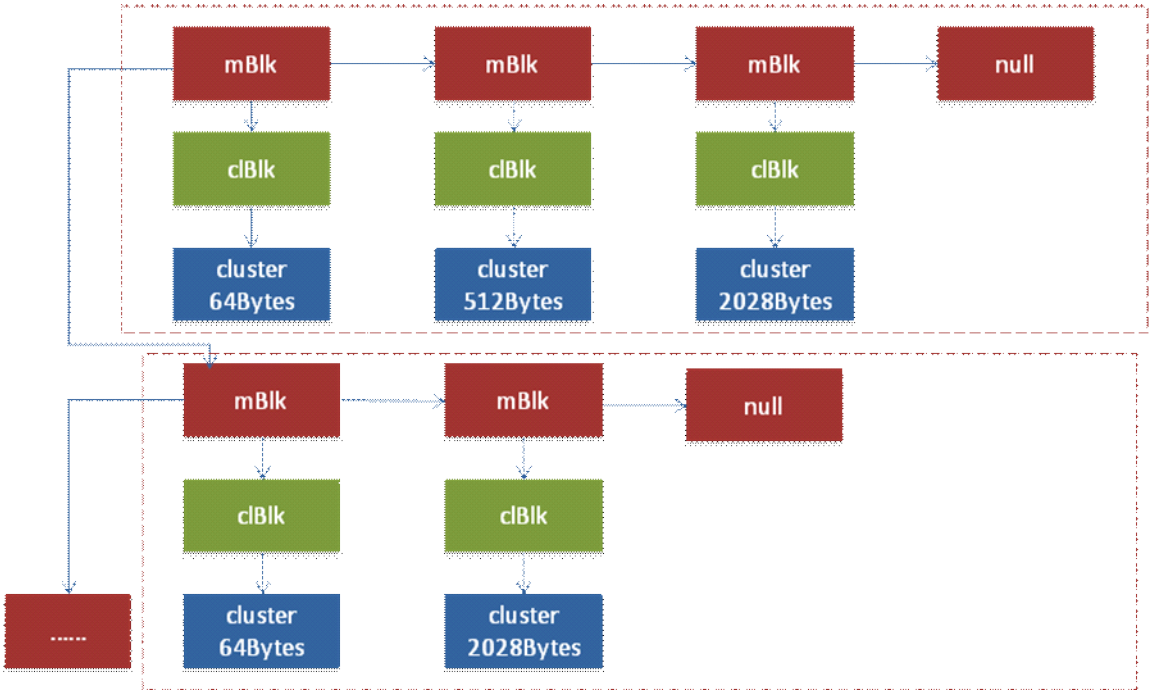
3.1 网络内存池

3.1.1 netBufLib 管理的内存池

netBufLib 是 VxWorks 网络体系结构中用来管理内存池的一个函数库。这个函数库提供用户建立和管理一个供驱动层和协议层使用的内存池的完整的接口。这个内存池采用了三层的设计方式，如图所示。数据被存放在 cluster 中。mBlk 和 clBlk 保存有管理 cluster 用的相关信息和 cluster 的指针。这种设计的好处很多，最明显的一点就是方便了数据包的转移。它使得数据包在不同层之间传送时，只需传递相关的指针，而避免了昂贵的数据拷贝。

mBlk 是用户获取一个数据包内容最主要的结构。它包含唯一的指向 clBlk 的指针。链表式的 mBlks 允许用户一次传送大量的数据。位于链表头的 mBlk 保存使有这个链表的入口，以及下一个数据包的链表的头节点。在图中，一个红色的虚线框表示一个数据包。

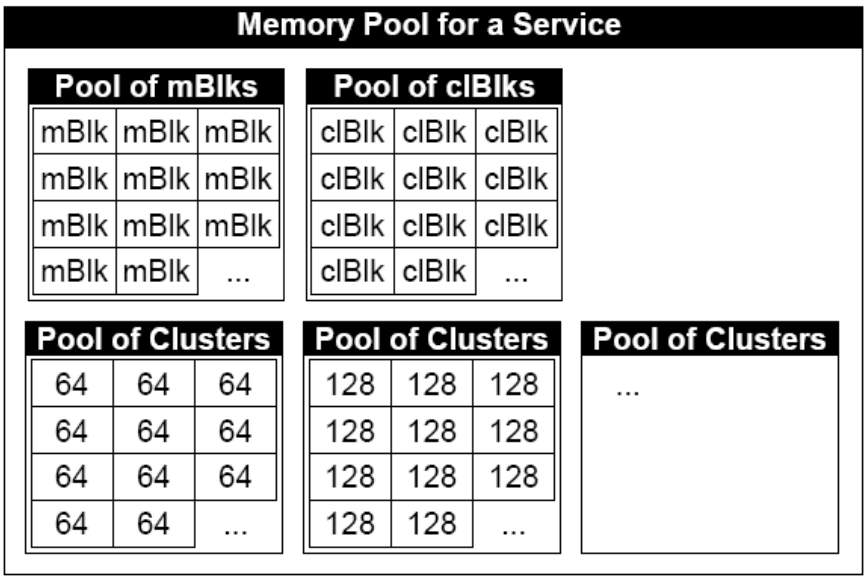
使用 clBlk 增加一个重定向层并不是一种浪费的设计，事实上它在这个内存池中是很关键的。每个 clBlk 保存有被 mBlk 引用的计数。数据包的“拷贝”正是靠 mBlk 传递 clBlk 的指针实现的。当 clBlk 的引用计数为零时，系统将把 cluster 回收进内存池。



3- 2 mBlk、clBlk、cluster 组织方式

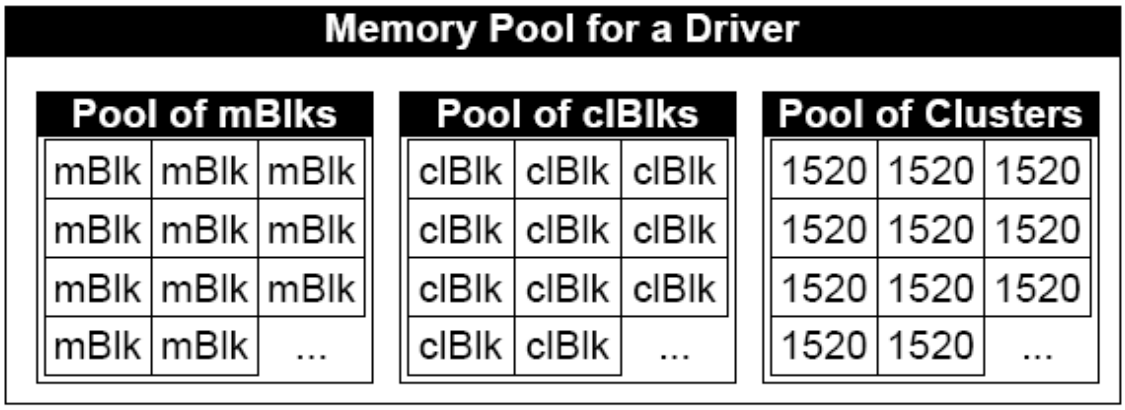
所有的内存池都被组织成 mBlk、clBlk、cluster 的方式。如何配置一个内存池，决定于打算将内存池应用于哪个模块。是协议层中，还是驱动层中。

网络协议层通常需要不同大小的 cluster。它的 cluster 内存池通常是由多个子内存池组成。这些内存池要求是 2 的次方数。



3- 3 网络协议层的 mBlk 内存池

驱动层只需要固定大小的 cluster。这个 cluster 的大小由下层设备的最大传输单元(MTU)决定。在以太网驱动程序中，这个一般大小为 1520 Bytes。

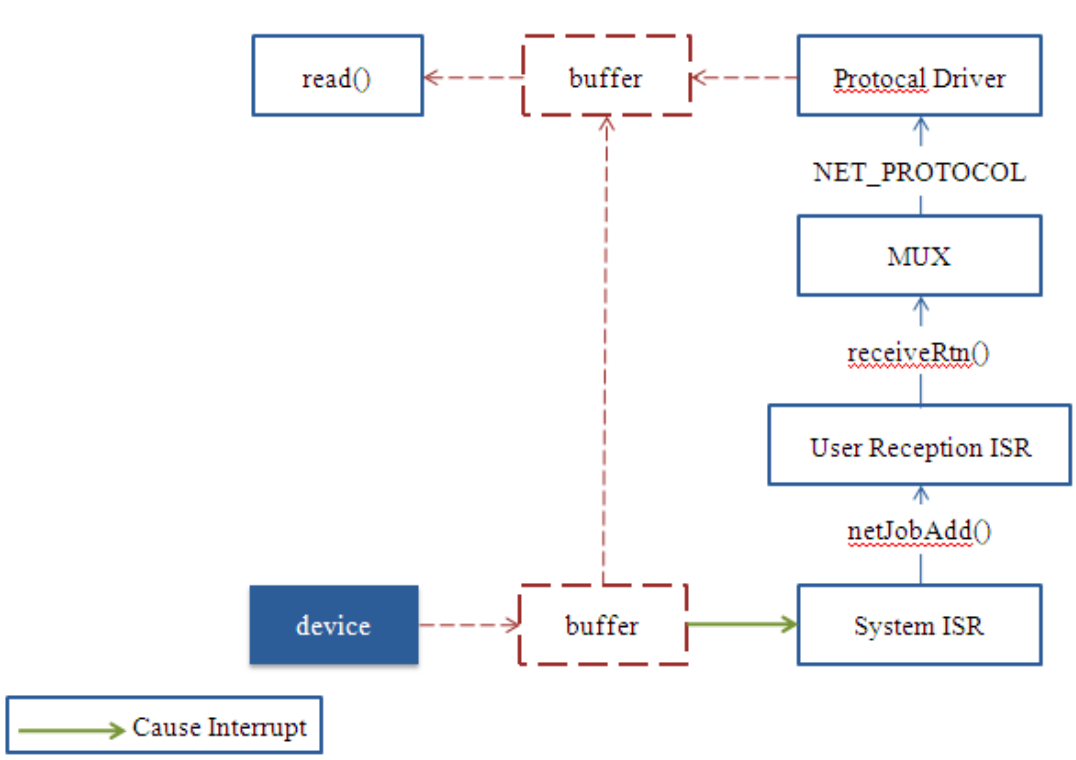


3- 4 驱动层的 mBlk 内存池

3.2 接收数据

在 VxWorks 的网络体系结构中，接收数据的流程如图。因为在从驱动层向上传递数据包的过程中，只是传递数据缓冲区的指针，而不是真正拷贝，所以图中 buffer 字段用红色虚线标注。这里将分析接受过程中的主要处理机制：

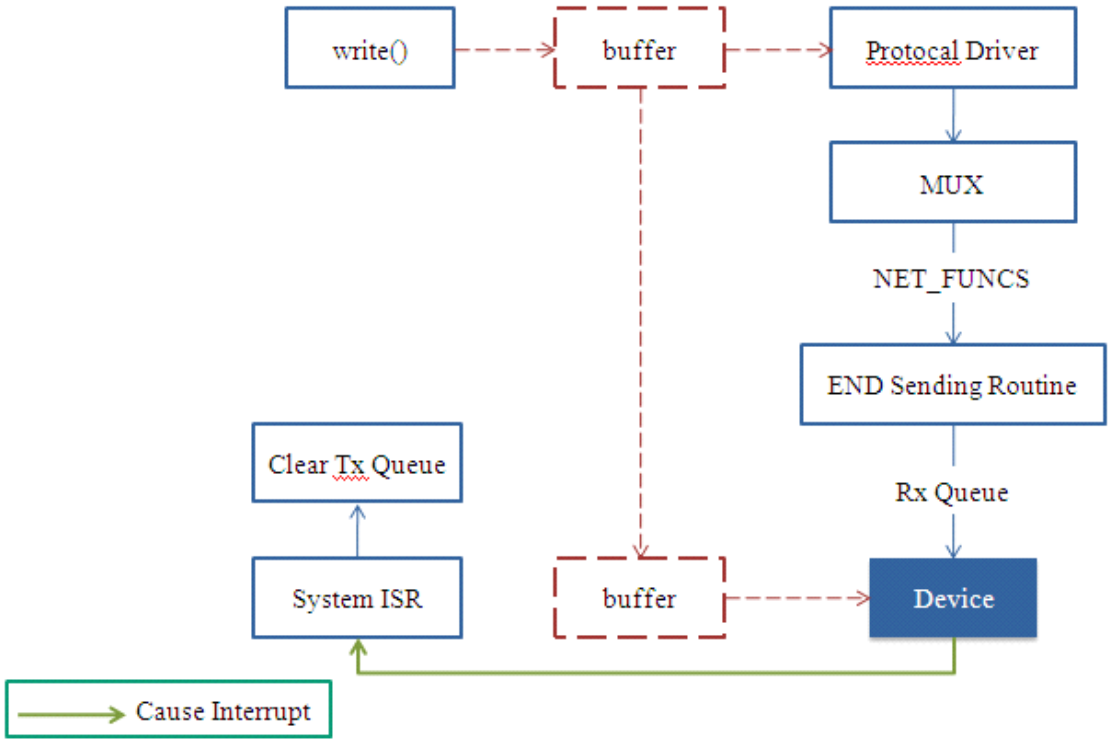
1. 设备接受完数据包后会引起一个“接收完成”中断。
2. 系统的中断服务程序获得引起中断的设备号，并跳转到用户定义的服务程序。
3. 用户服务程序从内存池获取 mBlk，clBlk，cluster，并将数据包填充进 cluster 中，最后调用回调函数 receiveRtn()递给 mux 层。
4. Mux 层完成的工作是调用协议注册的 stackRcvRtn()函数，把数据包传递给应用层。
5. 用户层使用 read()函数，通过 socket 访问得到的数据包。



3- 5 接收数据的流程图

3.3 发送数据

发送数据的过程和接收数据的过程类似。它的处理过程如下图所示。这里不再叙述。



3- 6 发送数据的流程图

4. xCat 收发包的代码分析

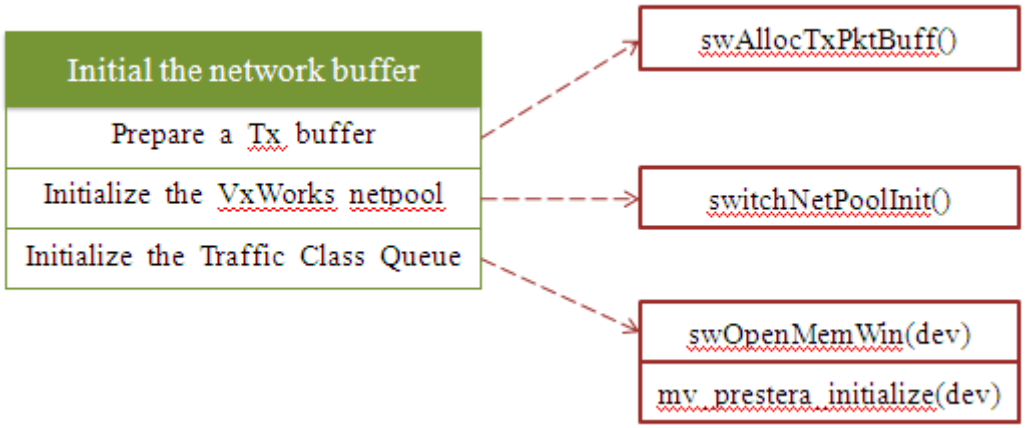
4.1 mvSwitchLoad 函数

mvSwitchLoad 是 xCat 网络驱动程序模块运行的第一个函数。这个函数负责初始化所有 end 层驱动相关的数据结构。并根据芯片的特性，完成各种硬件的初始化工作。它由三个部分组成：

- 分配空间，初始化内存池。
- 初始化驱动程序的控制结构。
- 配置交换机端口的属性。

分配空间，初始化内存池

驱动程序需要的内存空间都是在这里分配。下图这部分做的主要工作。



4- 1 Initial the network buffer

当发送数据包时，需要一块空间作为临时缓冲区，接收来自上层，由 mBlk->clBlk->cluster 形式存放的数据包。并经过添加 DSA tag 等的控制信息，生成可以被传送的二层数据包。这块临时的发送缓冲的地址保存在 G_txPktBuff 这个全局指针中。

```
G_txPktBuff = swAllocTxPktBuff(SWITCH_MTU + EXTEND_DSA_TAG_SIZE);
if (!G_txPktBuff)
{
            mvOsPrintf("Switch driver failed to alloc mem.\n");
            return NULL;
}
```

为了符合 VxWorks 网络驱动的接口，向上传递数据，END 层需要初始化 mBlk->clBlk->cluster 组成的内存池，在 mvSwitchLoad 中调用 switchNetPoolInit()。

```
if (switchNetPoolInit() == ERROR)
  goto errorExit;
```

在 switchNetPoolInit()中，根据设备的需要，将计算和 Traffic Class Queue 的 Descriptor 相对应的 mBlk，clBlk，cluster 个数，如下所示。

```
/* Calculate number of mBlks, clBlks and Clusters for all queues */
switchMclBlkConfig.mBlkNum = 0;
switchMclBlkConfig.clBlkNum = 0;
for(queue = 0; queue < MV_ETH_RX_Q_NUM; queue++)
{
  switchMclBlkConfig.mBlkNum += switchRxQueueDescrNum[queue];
}

switchMclBlkConfig.mBlkNum *= PRESTERA_PORT_NUM;
switchMclBlkConfig.mBlkNum += CL_LOAN_NUM;
switchMclBlkConfig.clBlkNum = switchMclBlkConfig.mBlkNum;
switchMclBlkConfig.mBlkNum = switchMclBlkConfig.mBlkNum*4;
```

mBlk 和 clBlk 是两个起管理作用的结构。mBlk 的一个链表逻辑上代表一个数据包。它指向一个 clBlk 的结构，而 clBlk 存放着保存数据包内容的 cluster。这里将为 mBlk，clBlk，cluster 分配空间。switchMblkData.pMblkBase 指向 mBlk 和 clBlk 内存池的基地址。switchMblkData.pClsBase 指向 cluster 内存池的基地址。

```
/* Get memory for mBlks and clBlk */
switchMclBlkConfig.memSize = ((switchMclBlkConfig.mBlkNum  * (M_BLK_SZ
+ 4))+(switchMclBlkConfig.clBlkNum * CL_BLK_SZ + 4));
switchMblkData.pMblkBase = mvOsMalloc(switchMclBlkConfig.memSize + 4);

/* Get memory for clusters */
#if defined(MV_UNCACHED_RX_BUFFERS)
      switchMblkData.pClsBase = mvOsIoUncachedMalloc(NULL,
                                (switchClDescTbl[0].memSize  +  0x20),
NULL, NULL);
#else
      switchMblkData.pClsBase = mvOsIoCachedMalloc(NULL,
                                (switchClDescTbl[0].memSize  +  0x20),
NULL, NULL);
#endif
```

调用 netPoolInit 函数建立一个 netBufLib 管理的内存池。

```
/* Init the mem pool */
if (netPoolInit (switchMblkData.pNetPool, &switchMclBlkConfig,
             &switchClDescTbl[0], switchClDescTblNumEnt, pNetPoolFunc)
== ERROR)
{
  mvOsPrintf("netPoolInit failed\n");
        return ERROR;
}
```

switchRingCtrlInit 函数完成 switchMblkData.switchRingCtrl 这个结构的初始化。switchRingCtrl 在 descriptor->buffer 和 mBlk->clBlk->cluster 两个内存池的数据传递中，充当桥梁的作用。

```
if (switchRingCtrlInit(switchMclBlkConfig.clBlkNum) != MV_OK)
{
  return MV_FAIL;
}
```

初始化驱动程序的控制结构

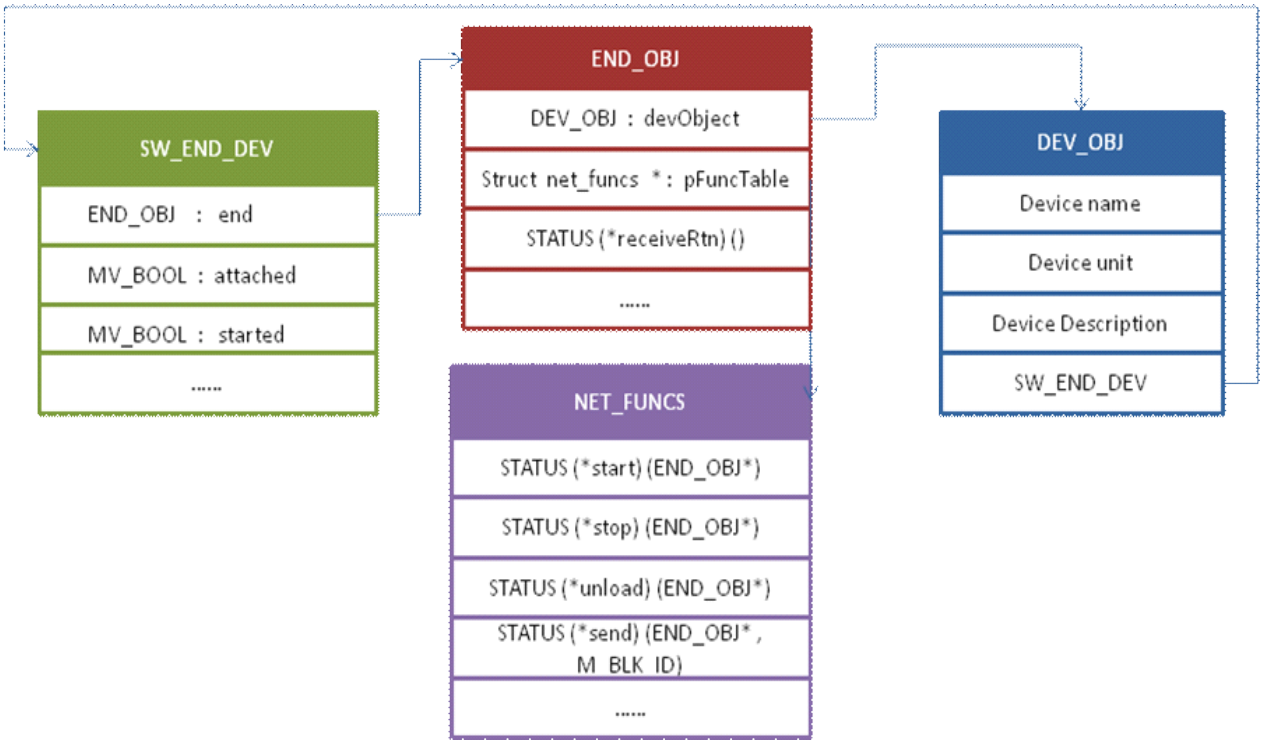
不同网络设备其本身资源、驱动程序等方面各不相同。因此，在编写驱动程序前需要为网络设备定义一个驱动程序控制结构。xCat 设备驱动程序中，这个结构是 SW_END_DEV。驱动程序通过这个结构获得所需要的资源并进行相应的处理。其中这个结构必须包含 END_OBJ 结构。

END_OBJ 结构中定义了所有网络相关的部分，提供了一个独立于设备的数据结构。END_OBJ 中包含了两个重要的数据结构，DEV_OBJ 和 NET_FUNCS。他同时还包含了接受过程的回调函数指针以及与协议相关的数据。这些数据都是为了 MUX 层提供的。

设备控制结构 DEV_OBJ，它包含了设备名、设备描述等信息。用户驱动程序用该结构来控制设备。

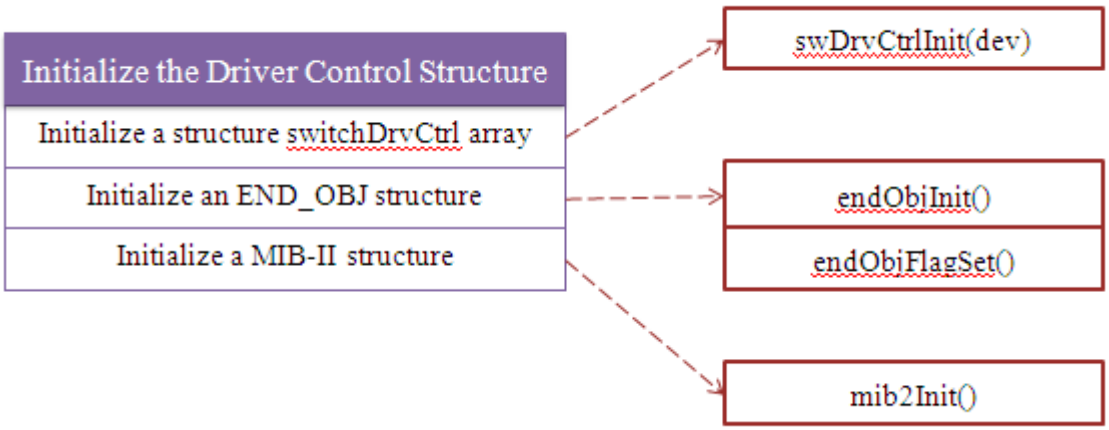
NET_FUNCS 结构为设备驱动程序函数表，这个结构包含了设备驱动程序各函数的入口点。在这个结构中定义的驱动程序函数包括：开始和停止设备的函数，发送数据包的函数，驱动程序的卸载函数，增加、删除、获取多播地址的函数等。注意设备的加载函数的入口存放在 END_TBL_ENTRY 结构中，而不是存放在这里。而中断方式的接收函数是直接注册在中断向量中，在 NET_FUNCS 中也没有入口。

上面所述的几个结构的关系如下图所示：



4- 2 控制结构的关系图

初始化驱动程序控制结构由三步组成。分别是初始： SW_END_OBJ 的数组 switchDrvCtrl，END_OBJ 结构和 MIB_II 结构。它们的初始化函数如图所示：



4- 3 Initialize the Driver Control Structure

每个设备对应着 switchDrvCtrl 数组的一项。swDrvCtrlInit 函数根据设备号初始化 switchDrvCtrl 数组。因为 switchDrvCtrl 表示的是具体的一种设备，因此，这里还对具体设备的非一般化属性进行了初始化。

```
for (dev = 0; dev < mvSwitchGetDevicesNum(); dev++)
{
  switchDrvCtrl[dev] = swDrvCtrlInit(dev);
```

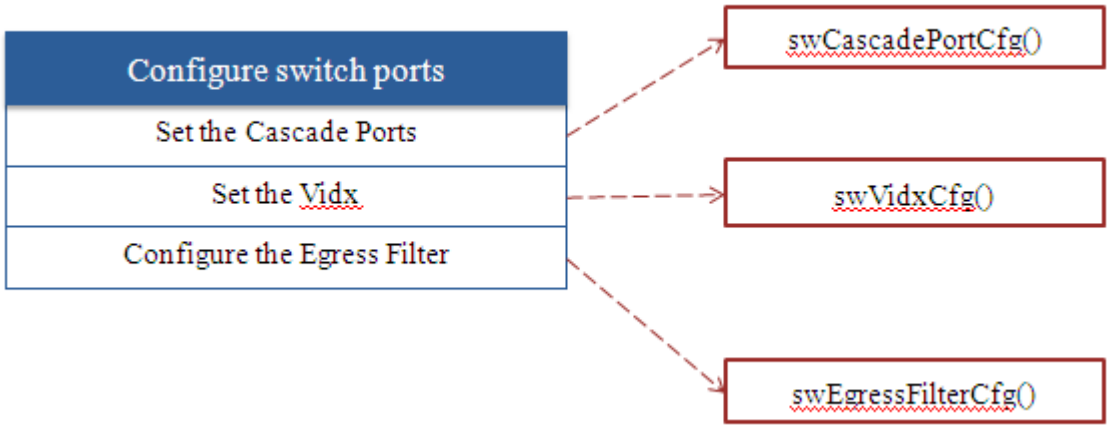
```
  if (!switchDrvCtrl[dev])
        return NULL;

  #ifndef SWITCH_CPU_PORT_TO_RGMII
            swOpenMemWin(dev);
            mv_prestera_initialize(dev);
   #endif

         if(setCPUAddressInMACTAble(dev,switchDrvCtrl[dev]-
>macAddr,VID(1)) != MV_OK)
        {
              mvOsPrintf("Error: (Prestera) Unable to teach CPU MAC
addr\n");
               return NULL;
         }
}
```

swCfgEndObj 函数负责初始化 END_OBJ 结构体。这是一个 VxWorks 网络体系结构约定好的，向 MUX 层传递数据的接口。swCfgEndObj 函数通过调用宏定义函数 END_OBJ_INIT，END_MIB_INIT，END_OBJ_READY 完成 END_OBJ 的初始化工作。

```
if (swCfgEndObj(dev) != OK)
        goto errorExit;
```

配置交换机端口的属性

作为驱动加载函数的最后一个部分，swCascadePortCfg 函数，swVidxCfg 函数，swEgressFilterCfg 函数配置了采用了 Marvell 芯片的交换机所特有的功能。这里不是本文的重点，如果希望深入了解，请读者另行阅读 Marvell 的技术手册。



4- 4 Configure switch ports

4.2 mvSwitchStart 函数

mvSwitchStart 函数执行了以下几个必要操作：设置默认的 mac 地址、注册设备驱动程序的中断服务程序、打开设备中断、打开端口。下面是函数完成中断注册和打开的地方。

```
for (dev = 0; dev < mvSwitchGetDevicesNum(); dev++)
{
```

```
  if (swSysIntConnect(dev))
        mvOsPrintf("%s: int connect for dev=%d failed\n", __func__,
dev);
  if (swSysIntEnable(dev))
  mvOsPrintf("%s: int enable for dev=%d failed\n", __func__, dev);
  swRxIntEnable(dev); /* should be after system int connect and enable
*/
}
```

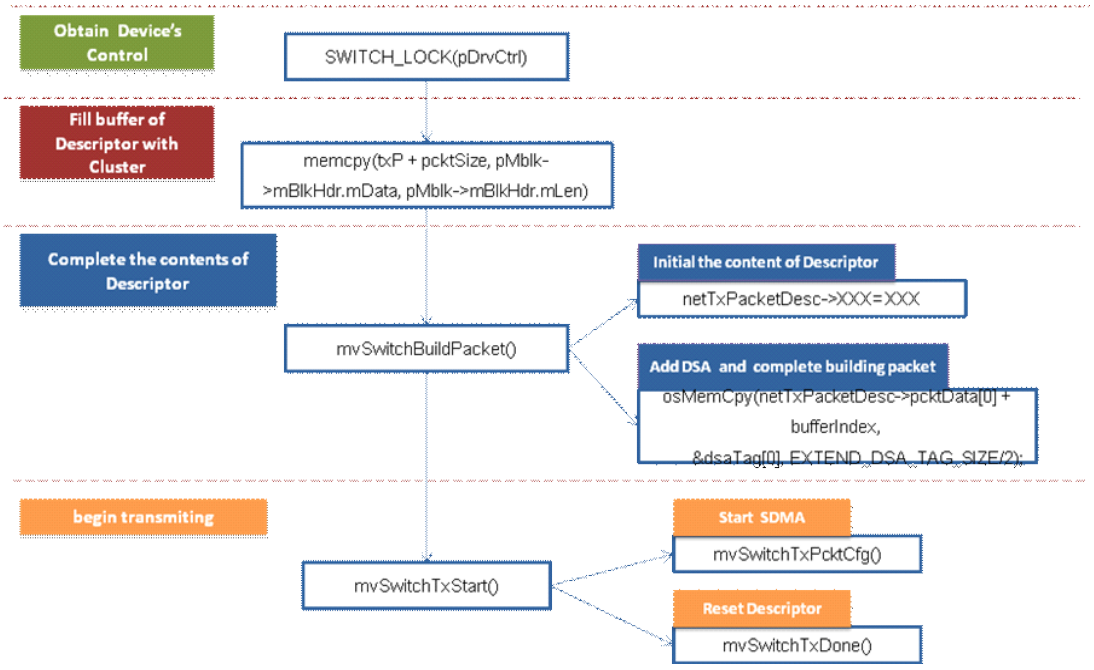
sysPciIntLvl 函数通过调用 sysPciIntConnect 函数将设备驱动程序的中断服务程序、中断号 intLvl 和设备号连接在一起。

```
MV_STATUS swSysIntConnect(MV_U32 swDevNum)
{
    int intLvl = swGetIntLvl(swDevNum);
    return sysPciIntConnect(intLvl,mvSwitchRxReadyIsr,
(int)switchDrvCtrl[swDevNum]);
}
```

swRxIntEnable 函数通过设置 SWITCH_GLOBAL_MASK_REG 寄存器(0x00000034)和 SWITCH_SDMA_RX_MASK_REG 寄存器(0x00002814)，打开了中断向量表中的接收中断。

```
void swRxIntEnable(MV_U32 swDevNum)
{
    hwIfSetReg(swDevNum,SWITCH_GLOBAL_MASK_REG,
SWITCH_RX_SDMA_INT_MASK);
    hwIfSetReg(swDevNum,SWITCH_SDMA_RX_MASK_REG, SWITCH_RX_QUEUE_MASK);
}
```

4.3 mvSwitchSend 函数



4- 5 mvSwitchSend 函数执行图

获得设备的使用权

pDrvCtrl 是 SW_END_DEV 类型的一个指针。它是用户自定义的一个结构，程序中用这个结构来控制设备。这个结构的成员如下所示：

```
typedef struct
{
    END_OBJ     end;        /* The class we inherit from.     */
    MV_U32      swDevNum;   /* Prestera switch dev number     */
    MV_U32      port;       /* unit number                    */
    MV_BOOL     attached;   /* Interface has been attached    */
    MV_BOOL     started;    /* Interface has been started     */
    MV_BOOL     isUp;       /* Link is UP                     */
    MV_U8       errorStat;  /* error status                   */
    MV_U8       macAddr[MV_MAC_ADDR_SIZE];
    MV_U32      rxqCount;
    MV_U32      txqCount;
    MV_U8       align;      /* ip hader need to be align to 0 */
} SW_END_DEV;
```

为了避免访问冲突，系统使用了信号量机制。在进行具体操作时需要先锁定这个结构。

```
SWITCH_LOCK(pDrvCtrl)
```

生成待发送的数据包

从 MUX 层传递下来的数据，存放在系统内存池管理的 cluster 中。一个完整的数据包可能由多个 cluster 组成。这一步的操作将 cluster 中的数据，提取出来，加入 DSA tag 字段，生成待发送的数据包。这里使用了一个全局的临时变量，来组装所有 cluster 的数据。

```
MV_U8 *G_txPktBuff;
```

Cluster 的位置和大小，由 mBlk 和 clBlk 进行管理。这里使用循环，提取出相关的 cluster。

```
while(pMblk && pMblk->mBlkHdr.mLen && pMblk->mBlkHdr.mData)
{
  if (pcktSize + pMblk->mBlkHdr.mLen > SWITCH_MTU)
  {
              mvOsPrintf("MblkPkt too long max = %d\n", pcktSize);
                 return MV_FAIL;
  }
        memcpy(txP    +    pcktSize,    pMblk->mBlkHdr.mData,    pMblk-
>mBlkHdr.mLen);
        pcktSize += pMblk->mBlkHdr.mLen;
        pMblk = pMblk->mBlkHdr.mNext;
}
```

填写描述符的字段

SDMA 通过 Traffic Class Queue 来存储待传输的数据包和记录传输时发生的事情。CPU 需要填写 Traffic Class Queue 中的管理单元 Descriptor 各个字段的内容，来向 MAC 芯片传递相关的信息。

```
tmpDesc.word1 = hwByteSwap(tmpDesc.word1);
tmpDesc.word2 = hwByteSwap(tmpDesc.word2);
TX_DESC_COPY(currSwDesc->txDesc,&tmpDesc);
PRESTERA_DESCR_FLUSH_INV(NULL,currSwDesc->txDesc);
```

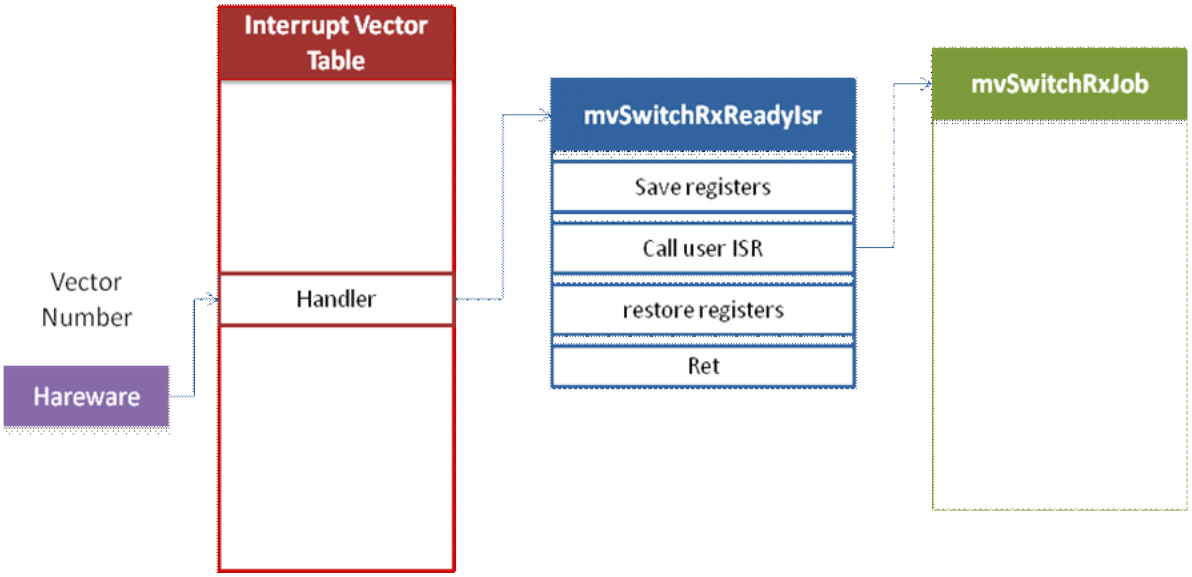
启动传输

CPU 通过设置 Transmit SDMA Queue Command Register (Offset : 0x2868)中对于的传输队列的寄存器的值，通知 SDMA 去传输已经准备好的数据包。

```
MV_U32 tmpData = 0;
U32_SET_BIT(tmpData, txQueue);
CHECK_STATUS(hwIfSetReg(devNum, 0x2868, tmpData));
```

4.4 mvSwitchRxReadyIsr 函数

SDMA 向 CPU 传输完数据后，会触发一个中断。系统捕捉到这个中断，而跳转到响应的中断服务程序中执行。



4- 6 接收中断的处理过程

在中断向量表中注册的接收数据包中断服务程序需要读取两个寄存器的数据位(SWITCH_GLOBAL_CAUSE_REG、SWITCH_SDMA_RX_CAUSE_REG)来判断是否由于事件“SDMA 接收数据包完成”而引起的中断。从而跳转到用户自定义的接收处理函数中来。

```
hwIfGetReg(dev, SWITCH_GLOBAL_CAUSE_REG,  &globalCause);
hwIfGetReg(dev, SWITCH_SDMA_RX_CAUSE_REG, &rxSdmaCause);
if ((globalCause & SWITCH_RX_SDMA_INT_MASK) != 0 &&
(rxSdmaCause & SWITCH_RX_QUEUE_MASK)     != 0)
```

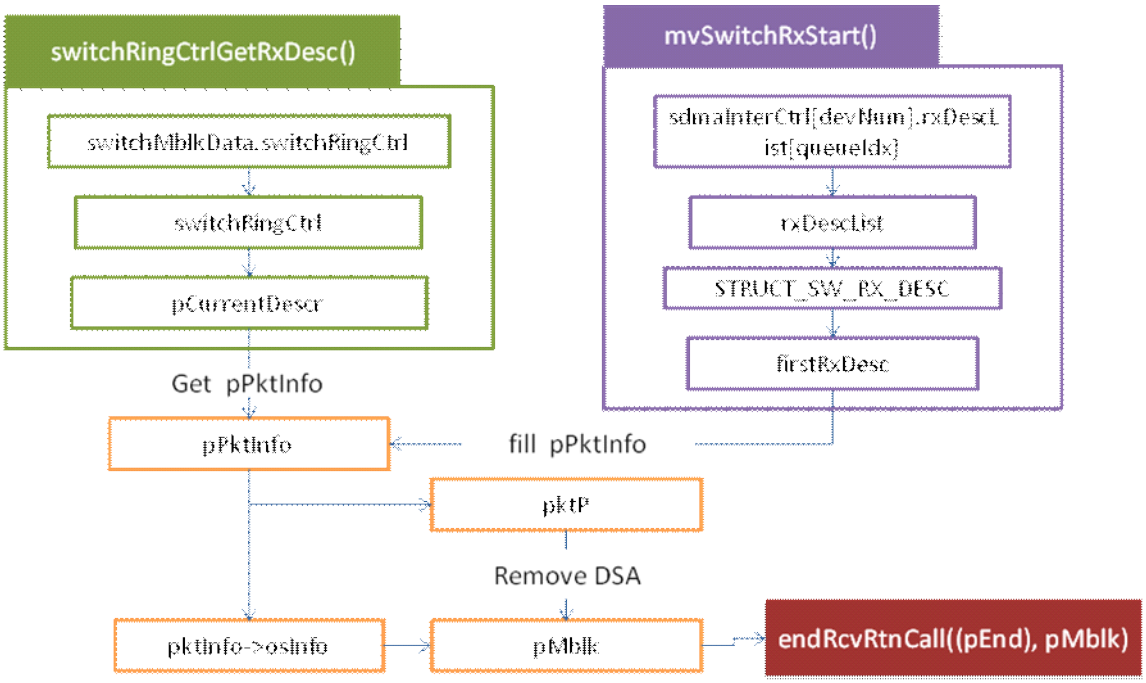
响应“接收”的中断服务程序必须精简，为了及时地让系统返回，它需要调用一个用户自己写的进程，完成大部分的数据包接收工作。

```
rc = netJobAdd((FUNCPTR)mvSwitchRxJob,(int)switchDrvCtrl[dev], 0, 0, 0,
0);
if (rc != OK)
{
  debugStats.netJobAddError++;
```

```
}
```

4.5 mvSwitchRxJob 函数

用户自定义的接收处理函数将完成数据包向 MUX 层传递的任务。为了实现这个功能，接收处理函数需要做两方面的工作。首先是获得传递给上层回调函数的数据结构 mBlk 的内容。然后是获得对于 Traffic Class Queue 中的 Descriptor List。因为这两者分别处于不同的内存池中，所以需要按照相关内存池的设计，获得当前需要的结构体，并维护内存池的正常状态。最后是完成数据的传递和填写相关的描述信息。这个过程总的流程图如下所示：



4- 7 接收数据包的流程图

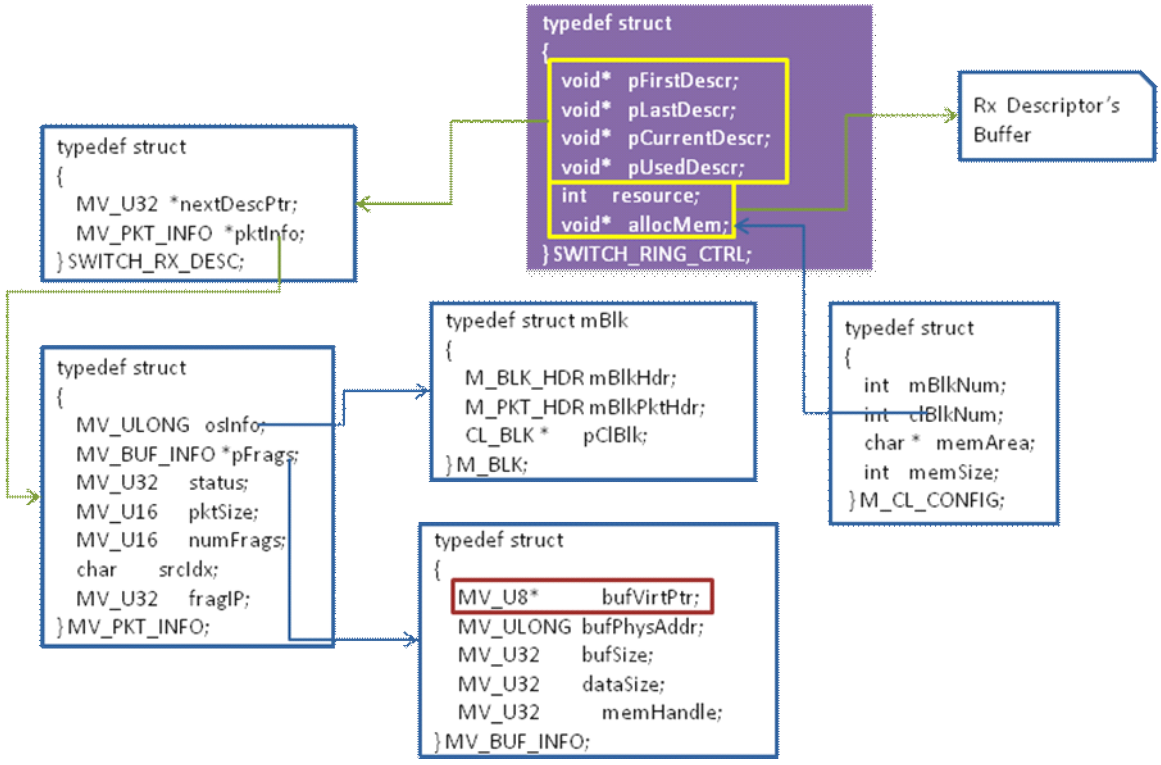
mBlk 的结构保存在 MV_PKT_INFO 结构体中。SWITCH_RING_CTRL 维护着一个 SWITCH_RX_DESC 的线性表，它管理着这个结构的释放和分配。SWITCH_RX_DESC 保存着 MV_PKT_INFO 结构体的指针，程序将从这里获得 mBlk 内存池中的一个 mBlk 结构。它们的关系如图 3.5 所示。代码如下：

```
pktInfo = switchRingCtrlGetRxDesc();
if (!pktInfo)
{
  mvOsPrintf("%s: Rx ring error\n", __func__);
        return MV_FAIL;
}
```

在 switchRingCtrlGetRxDesc()函数中：

```
SWITCH_RING_CTRL* switchRingCtrl = &switchMblkData.switchRingCtrl;

/* Check resources */
if(switchRingCtrl->resource == 0)
{
```

```
   mvOsPrintf("%s: no more resources.\n", __func__);
         return NULL;
}

/* Get the Rx Desc ring 'curr and 'used' indexes */
pRxCurrDesc = switchRingCtrl->pCurrentDescr;
pPktInfo = (MV_PKT_INFO*)pRxCurrDesc->pktInfo;
```



4- 8 SWITCH_RING_CTRL 等结构的关系图

mvSwitchRxStart 是接收过程最核心的函数。它检查当前系统的状态，并读取 rxDescList 中当前数据包的相关内容。最后将读取到的内容赋值给一个返回结果的结构体。

```
status = mvSwitchRxStart(pDrvCtrl->swDevNum, queueIdx, pktInfo);
```

Rx 数据包的 Descriptor 保存在 rxDescList 中。mvSwitchRxStart 函数申请了一个临时变量 DescPtr，从全局变量 sdmaInterCtrl[devNum].rxDescList[queueIdx]中获得当前的描述符链表，并根据当前的 CPU 体系结构完成描述符字段的大小端处理。

```
rxDescList = &(sdmaInterCtrl[devNum].rxDescList[queueIdx]);
descPtr = rxDescList->next2Receive;

descPtr->shadowRxDesc.word1 = hwByteSwap(descPtr->rxDesc->word1);
descPtr->shadowRxDesc.word2 = hwByteSwap(descPtr->rxDesc->word2);
firstRxDesc = descPtr;

/* Get the packet's descriptors.*/
while(RX_DESC_GET_LAST_BIT(&(descPtr->shadowRxDesc)) == 0)
{
       descPtr->swNextDesc;
       descPtr->shadowRxDesc.word1=hwByteSwap(descPtr->rxDesc->word1);
       descPtr->shadowRxDesc.word2=hwByteSwap(descPtr->rxDesc->word2);
       descNum++;
```

```
/* If enable RX process if disabled then the descriptor is cleared */
        if (enableProcessRxMsg == MV_FALSE)
        {
            MV_SYNC;
            RX_DESC_RESET(descPtr->rxDesc);
            MV_SYNC;
        }
        if(RX_DESC_GET_REC_ERR_BIT(&(descPtr->shadowRxDesc)) == 1)
        {
            return MV_OK;
        }
}
```

在有了上面的准备工作，程序将开始拷贝数据包的内容。因为 Desciptor 和它指向的 Buffer 保存在不同的内存池中。所以，拷贝数据包是以传递指针的形式完成的。

```
/* Update packet structure */
pktInfo->pFrags->bufPhysAddr=hwByteSwap(swRxDesc->rxDesc->buffPointer);
pktInfo->pFrags[i].bufVirtPtr = (MV_U8*)swRxDesc->buffVirtPointer;
pktInfo->pFrags[i].bufSize = buffLen[i];
```

程序使用 mBlk，clBlk，Cluster 的内存管理方式，将数据包的内容传给上层。因此将从 rxDescList 获取的数据包和描述信息赋值给 mBlk 的相应字段。

```
/* set the pMblk parameters */
pMblk = (M_BLK_ID)pktInfo->osInfo;

/* join the cluster to pktP */
pMblk->pClBlk->clNode.pClBuf = (void*)pktP;
pMblk->pClBlk->clRefCnt = 1;
/* ETH_DESCR_INV(NULL,pMblk->pClBlk->clNode.pClBuf); */

pMblk->mBlkPktHdr.len      = pktSize;
pMblk->mBlkHdr.mLen        = pktSize;
pMblk->mBlkHdr.mData       = pMblk->pClBlk->clNode.pClBuf;
pMblk->mBlkHdr.mNext       = NULL;
pMblk->mBlkHdr.mNextPkt    = NULL;
pMblk->mBlkHdr.mFlags     |= M_PKTHDR | M_EXT;
pMblk->pClBlk->clFreeArg1  = (MV_U32)pDrvCtrl;
pMblk->pClBlk->clFreeArg2  = (MV_U32)pktInfo;
pMblk->pClBlk->clFreeArg3 |= (pMblk->mBlkHdr.mLen << 16);
```

最后，end 层通过回调函数，完成向 MUX 层传递数据包的任务。

```
END_RCV_RTN_CALL(&switchDrvCtrl[PRESTERA_DEFAULT_DEV]->end, pMblk);
```

5. 附录

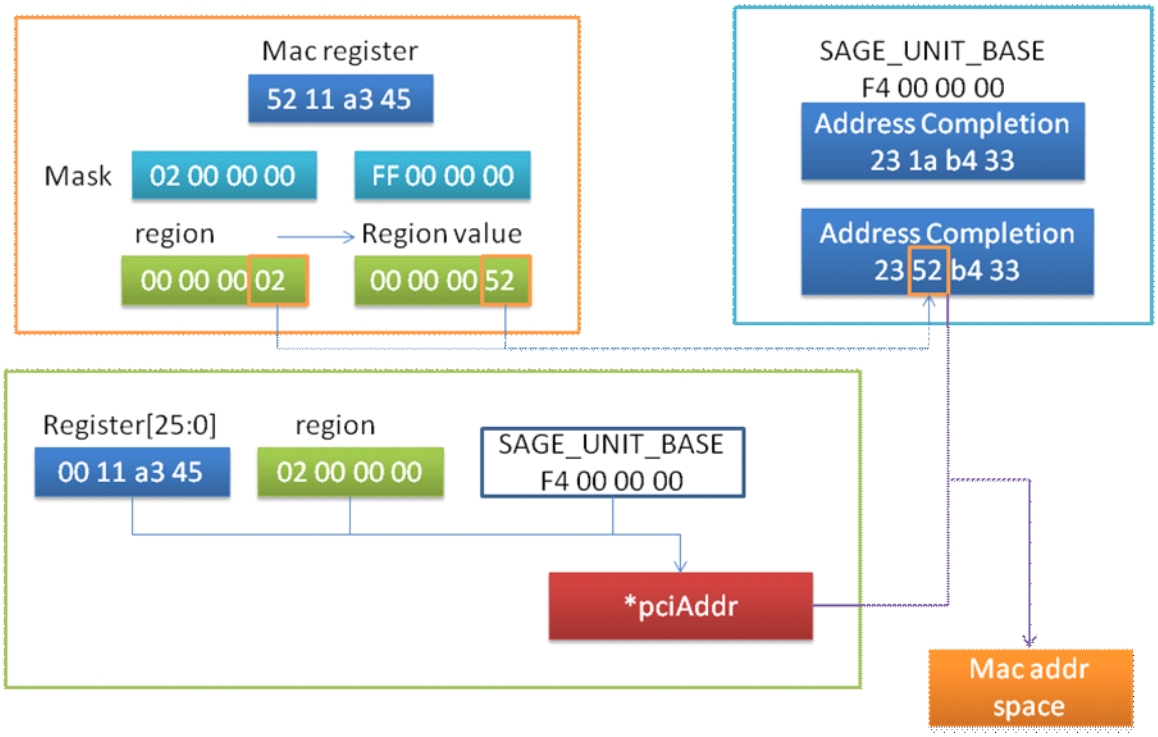
5.1 寄存器操作

xCat 的驱动程序通过读写设备的寄存器实现对设备的控制。操作寄存器常用的函数有：

- mvSwitchWriteReg()
- mvSwitchReadReg()
- mvSwitchReadModWriteReg()
- mvSwitchAddrComp()

在 CPU 子系统中，所有片上系统的外设都被映射到同一个地址空间中。它们的寄存器可以通过与读写 SDRAM 同样的方式访问。但是 CPU 通过不同的总线与各个外设连接，总线的设计方式需要驱动在读写寄存器时，增加一些额外的处理。

出于成本和芯片体积的考虑，xCat 的 CPU 和 MAC 芯片之间用的是 26 位的 PCI 总线。因此 32 位的 CPU 地址需要经过一个变换，才将一个完整的地址传递给 MAC。下面是一个地址的转换图。



5- 1 PCI 总线的地址转换图