这篇blog主要分为三部分：

1，中断初始化

2，vxbus架构中断的挂接

3，非vxbus架构的中断的挂接

1，中断的初始化

函数调用：

usrInit->sysStart->palInit->palDescInit()

/\*为了方便查看，这里对函数 进行了删减

\*palDescInit()函数对默认的中断函数进行了初始化

\*/

void palDescInit (void)

{

int idx;

#ifndef \_WRS\_MIPS\_NONCOMPLIANT\_CONFIG\_REG

UINT32 config;

#endif /\* \_WRS\_MIPS\_NONCOMPLIANT\_CONFIG\_REG \*/

bzero((char \*)&palDesc, sizeof(palDesc));

palDesc.coreNum = 0;

palDesc.version = "1.1";

/\* Hopefully code further down will be able to refine these \*/

palDesc.iCache.present = FALSE;

palDesc.dCache.present = FALSE;

palDesc.l2Cache.present = FALSE;

palDesc.l3Cache.present = FALSE;

if (IS\_KSEGM((UINT32) &palDescInit))

{

/\* executing a mapped kernel \*/

palDesc.tVec.vectorAddr = (VIRT\_ADDR \*)T\_VEC;

palDesc.tVec.excHandler = (VIRT\_ADDR \*)mmuMipsTlbVec;

palDesc.tVec.excSize = mmuMipsTlbVecSize;

palDesc.xVec.vectorAddr = (VIRT\_ADDR \*)X\_VEC;

palDesc.xVec.excHandler = (VIRT\_ADDR \*)mmuMipsXtlbVec;

palDesc.xVec.excSize = (UINT)mmuMipsXtlbVecSize;

palDesc.cVec.vectorAddr = (VIRT\_ADDR \*)C\_VEC;

palDesc.cVec.excHandler = (VIRT\_ADDR\*)excCacheVec;

palDesc.cVec.excSize = (UINT)excCacheVecSize;

palDesc.eVec.vectorAddr = (VIRT\_ADDR \*)E\_VEC;

palDesc.eVec.excHandler = (VIRT\_ADDR \*)excNormVmVec;

palDesc.eVec.excSize = (UINT)excNormVmVecSize;

}

/\* Reasonable defaults \*/

palDesc.tlb.supportedPageSizes = MMU\_PAGE\_MASK\_8K;

palDesc.tlb.mmuCacheable=MMU\_R4K\_STATE\_INVALID\_STATE >> MMU\_R4K\_CACHE\_START；

palDesc.tlb.mmuUncacheable = CFG\_CCA\_UNCACHED;

palDesc.tlb.mmuCacheCopyback = CFG\_CCA\_CACHEABLE;

palDesc.tlb.mmuCacheWritethrough = CFG\_CCA\_ALIAS\_CACHEABLE;

palDesc.tlb.mmuCacheCoherency = CFG\_CCA\_COHERENT;

palDesc.iCache.modes = CACHE\_MODES\_SUPPORTED;

palDesc.dCache.modes = CACHE\_MODES\_SUPPORTED;

}

//重要的结构体说一下

#define T\_VEC K0BASE /\* tlbmiss vector \*/

#define X\_VEC (K0BASE+0x80) /\* xtlbmiss vector \*/

#define C\_VEC (K1BASE+0x100) /\* cache exception vector \*/

#define E\_VEC (K0BASE+0x180) /\* exception vector \*/

#define R\_VEC (K1BASE+0x1fc00000) /\* reset vector \*/

#define BEV\_VEC (K1BASE+0x1fc00380) /\* boot exception vector \*/

typedef struct

{

char \*version; /\* PAL version string \*/

int coreNum; /\* processor number \*/

MIPS\_EXC\_VECTOR tVec; /\*tlbmiss中断\*/

MIPS\_EXC\_VECTOR xVec; /\*xtlbmiss中断\*/

MIPS\_EXC\_VECTOR cVec; /\*cache异常中断\*/

MIPS\_EXC\_VECTOR eVec; /\*异常中断，包括中断(根据exccode码判别)\*/

MIPS\_MMU\_INFO tlb; /\*\*/

MIPS\_CACHE\_INFO iCache; /\*指令cache相关\*/

MIPS\_CACHE\_INFO dCache; /\*数据cache相关\*/

MIPS\_CACHE\_INFO l2Cache; /\*二级cache相关\*/

MIPS\_CACHE\_INFO l3Cache; /\*三级cache相关\*/

int mmuType; /\* CR bits 9:7 \*/

BOOL hasCP2;

BOOL hasWatch;

BOOL hasFPU;

} MIPS\_PAL\_DESC;

默认中断函数的安装：

函数调用：

usrInit->excVecInit

/\*在这里会安装默认的中断处理函数\*/

STATUS excVecInit (void)

{

ULONG srValue; /\* status register placeholder \*/

/\* Load tlb and Xtlb vectors \*/

vecInit (&palDesc.tVec);

vecInit (&palDesc.xVec);

/\* Load cache exception vector \*/

vecInit (&palDesc.cVec);

/\* Load normal/interrupt vector \*/

vecInit (&palDesc.eVec);

srValue = intSRGet();

srValue &= ~SR\_BEV;

intSRSet(srValue);

return (OK);

}

LOCAL void vecInit

(

MIPS\_EXC\_VECTOR \* vectorDesc

)

{

if (vectorDesc->excSize)

{

/\*把函数拷贝到设置的地址处\*/

bcopy ((const char \*) vectorDesc->excHandler,

(char \*) vectorDesc->vectorAddr, vectorDesc->excSize);

/\*刷cache操作\*/

CACHE\_TEXT\_UPDATE (vectorDesc->vectorAddr, vectorDesc->excSize);

}

}

到这里系统默认的中断处理函数就初始化完成。

这里找一个具体的中断控制器的初始化进行说明。(这是一个抽象出来的中断控制器，至于什么是抽象，也就是没有一个具体的物理硬件设备对应这个控制器，这只是一个软件层面的简化，这样对于中断的处理就更明显，直接。)

函数调用关系：

usrInit->sysHwInit->vxbMipsLsnIntCtlrRegister

当这个控制其注册之后：

vxbMipsLsnIntCtlrRegister->vxbDevRegister->vxbNewDriver->vxbDevInitRun

/\*vxbDevInitRun函数会调用设备对应的初始化函数对设备进行初始化\*/

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;

}

}

这里的pDrv->pDrvBusFuncs->devInstanceInit函数对应控制器的lsnIntCtlrInstInit函数，在lsnIntCtlrInstInit函数中调用了vxbMipsLsnIntCtlrInit对中断进行初始化。

LOCAL VOID vxbMipsLsnIntCtlrInit

(

VXB\_DEVICE\_ID pInst

)

{

HCF\_DEVICE \*pHcf = NULL;

BOOL icw4Needed = FALSE;

pVXB\_LSN\_INT\_DRV\_CTRL pDrvCtrl = pInst->pDrvCtrl;

/\*这里注意intCtlrHwConfGet，这是真正具体的初始化函数\*/

intCtlrHwConfGet(pInst, pHcf, &(pDrvCtrl->isrHandle));

Loongson2H\_interrupt\_init();

return;

}

intCtlrHwConfGet这个函数比较长，删减以下，可以看出来这个函数所作的工作就是从hwconf.c中得到设备对应的中断信息，并对每一个对应的引脚初始化为一个具体的结构体，结构体中包含的成员可以看一看。

STATUS intCtlrHwConfGet

(

VXB\_DEVICE\_ID pInst,

HCF\_DEVICE \* pHcf,

struct intCtlrHwConf \* pEntries

)

{

struct intrCtlrInputs \* pCtlrInputs; /\*中断控制引脚\*/

struct intrCtlrXBar \* pXbar; /\*xbar路由\*/

struct intrCtlrCpu \* pCpuTable; /\*对应那个cpu\*/

struct intrCtlrPriority \* pPrioTable; /\*优先级\*/

struct intrCtlrTrigger \* pTrigTable; /\*引发方式,上升沿,高低电平,下降沿\*/

int tableSize; /\*控制引脚数组的大小\*/

int i;

STATUS stat;

void \* pValue;

void \*\* ppValue;

/\* save intCtlr ID \*/

pEntries->ctlrDev = pInst;

ppValue = &pValue;

/\*获取中断控制引脚组成的数组\*/

stat = devResourceGet(pHcf, "input", HCF\_RES\_ADDR, ppValue);

pCtlrInputs = (struct intrCtlrInputs \*)pValue;

if ( stat == OK )

{

/\*获取中断控制引脚数组的大小\*/

stat = devResourceGet(pHcf, "inputTableSize", HCF\_RES\_INT, ppValue);

tableSize = (UINT32)pValue;

/\*遍历表中的每个元素，并为每一个对应的引脚创建相应的结构体\*/

for ( i = 0 ; i < tableSize ; i++ )

{

/\*为引脚创建一个结构体\*/

stat = intCtlrTableCreate(pEntries, pCtlrInputs->inputPin);

/\*为创建的结构体中的元素进行赋值操作，包括引脚号(硬件上连接的引脚号)，驱动名字，drvunit以及drvIndex \*/

intCtlrTableUserSet(pEntries, pCtlrInputs->inputPin, pCtlrInputs->drvName,pCtlrInputs->drvUnit, pCtlrInputs->drvIndex);

pCtlrInputs++;

}

}

/\*获取优先级对应的数组\*/

stat = devResourceGet(pHcf, "priority", HCF\_RES\_ADDR, ppValue);

pPrioTable = (struct intrCtlrPriority \*)pValue;

if ( stat == OK )

{

/\*获取优先级表的大小\*/

stat = devResourceGet(pHcf, "priorityTableSize", HCF\_RES\_INT, ppValue);

tableSize = (UINT32)pValue;

/\*遍历表中所有的元素，把值填入到对应的引脚结构体中\*/

for ( i = 0 ; i < tableSize ; i++ )

{

/\*再次a创建引脚结构体，防止没有创建 \*/

stat = intCtlrTableCreate(pEntries, pPrioTable->inputPin);

/\*为结构体填入优先级 \*/

intCtlrTablePrioSet(pEntries, pPrioTable->inputPin,pPrioTable->priority);

pPrioTable++;

}

}

/\*下面操作基本都是类似的，获取对应的中断信息，并进行遍历填入到引脚结构体中\*/

/\*获取引脚出发的方式\*/

stat = devResourceGet(pHcf, "trigger", HCF\_RES\_ADDR, ppValue);

pTrigTable = (struct intrCtlrTrigger \*)pValue;

if ( stat == OK )

{

stat = devResourceGet(pHcf, "triggerTableSize", HCF\_RES\_INT, ppValue);

tableSize = (UINT32)pValue;

for ( i = 0 ; i < tableSize ; i++ )

{

/\* ensure pin record exists \*/

stat = intCtlrTableCreate(pEntries, pTrigTable->inputPin);

/\* fill in trigger \*/

intCtlrTableTrigSet(pEntries, pTrigTable->inputPin, pTrigTable->trigger);

pTrigTable++;

}

}

/\*获取crossbar路由表\*/

stat = devResourceGet(pHcf, "crossBar", HCF\_RES\_ADDR, ppValue);

pXbar = (struct intrCtlrXBar \*)pValue;

if ( stat == OK )

{

stat = devResourceGet(pHcf, "crossBarTableSize", HCF\_RES\_INT, ppValue);

tableSize = (UINT32)pValue;

for ( i = 0 ; i < tableSize ; i++ )

{

/\* ensure pin record exists \*/

stat = intCtlrTableCreate(pEntries, pXbar->inputPin);

intCtlrTableOutputSet(pEntries, pXbar->inputPin, pXbar->outputPin);

pXbar++;

}

}

stat = devResourceGet(pHcf, "cpuRoute", HCF\_RES\_ADDR, ppValue);

pCpuTable = (struct intrCtlrCpu \*)pValue;

#ifdef \_WRS\_CONFIG\_SMP

if ( stat == OK )

{

stat = devResourceGet(pHcf, "cpuRouteTableSize", HCF\_RES\_INT, ppValue);

tableSize = (UINT32)pValue;

/\* iterate through table \*/

for ( i = 0 ; i < tableSize ; i++ )

{

/\* ensure pin record exists \*/

stat = intCtlrTableCreate(pEntries, pCpuTable->inputPin);

/\* set destination CPU for this input pin \*/

if (stat == OK)

intCtlrTableCpuSet (pEntries, pCpuTable->inputPin,

pCpuTable->cpuNum);

pCpuTable++;

}

}

#endif /\* \_WRS\_CONFIG\_SMP \*/

return(OK);

}

/\*创建中断表\*/

STATUS intCtlrTableCreate

(

struct intCtlrHwConf \* pInputs,

int inputPin

)

{

int tblIndx = VXB\_INTCTLRLIB\_TBL\_IDX(pInputs,inputPin);

int startPin = inputPin - (inputPin % VXB\_INTCTLRLIB\_LOWLVL\_SIZE);

int i;

if ( pInputs->pTop == NULL )

{

/\*分配一个结构体vxbIntCtlrInpTop\*/

pInputs->pTop = (struct vxbIntCtlrInpTop \*)

hwMemAlloc(sizeof(struct vxbIntCtlrInpTop));

}

if ( pInputs->pTop->tbl[tblIndx] == NULL )

{

/\*分配结构体vxbIntCtlrInput\*/

pInputs->pTop->tbl[tblIndx] = (struct vxbIntCtlrInput \*)

hwMemAlloc(sizeof(struct vxbIntCtlrInput) +

VXB\_INTCTLRLIB\_MIN\_CACHE\_LINE\_SIZE);

/\*进行对齐操作\*/

pInputs->pTop->tbl[tblIndx] = (struct vxbIntCtlrInput \*)

ROUND\_UP(pInputs->pTop->tbl[tblIndx],

VXB\_INTCTLRLIB\_MIN\_CACHE\_LINE\_SIZE);

/\*赋值为0\*/

memset(pInputs->pTop->tbl[tblIndx],0,sizeof(struct vxbIntCtlrInput));

/\*为每一个引脚结构体中的中断函数以及中断函数参数赋一个系统默认的值 \*/

for ( i = 0 ; i < VXB\_INTCTLRLIB\_NUM\_PINS(pInputs) ; i++ )

{

pInputs->pTop->tbl[tblIndx]->pins[i].isr = intCtlrStrayISR;

pInputs->pTop->tbl[tblIndx]->pins[i].pArg = pInputs;

pInputs->pTop->tbl[tblIndx]->pins[i].pinFlags = startPin + i;

}

}

return(OK);

}

对应引脚重要的结构体：

struct intCtlrHwConf

{

VXB\_DEVICE\_ID ctlrDev; /\* VXB\_DEVICE\_ID of int ctlr \*/

struct vxbIntCtlrInpTop \* pTop; /\* top-level table of entries \*/

struct vxbLock vxbIntCtlrLibLock;

};

struct vxbIntCtlrInpTop

{

struct vxbIntCtlrInpTop \* pNext; /\* not implemented \*/

struct vxbIntCtlrInput \* tbl[VXB\_INTCTLRLIB\_TOPLVL\_SIZE];

UINT32 reserved;

};

#define VXB\_INTCTLRLIB\_TOPLVL\_SIZE 496

struct vxbIntCtlrInput

{

struct vxbIntCtlrPin pins[VXB\_INTCTLRLIB\_LOWLVL\_SIZE];

};

/\*最后的运算就是 2^x ,这里的定义也就是8\*/

#define VXB\_INTCTLRLIB\_LOWLVL\_SIZE\_POW 3

#define VXB\_INTCTLRLIB\_LOWLVL\_SIZE ( 1<< VXB\_INTCTLRLIB\_LOWLVL\_SIZE\_POW)

struct vxbIntCtlrPin

{

struct intCtlrMoreDrvName \* pMoreNames;

char \* drvName;

int drvUnit;

int intIndex;

void (\*isr)(void \* pArg, int inputPin);

void \* pArg;

int pinFlags;

UINT32 pinPriority;

UINT32 pinOutput;

UINT32 pinCpu;

int reserved [2];

};

这里就相当于创建了一个两级表，第一级是496的大小，第二级分别对应于第一级的每一个元素，大小为8，到这里中断的创建以及初始化就完成了。

2，中断的挂接

vxWorks中的vxbus架构初始化分为三个阶段，第一第二阶段一般都是对设备进行初始化，到第三阶段一般是中断的挂接和使能。这里举例说明的这个有点特殊，这是在第二阶段进行的中断挂接和使能，也从另一面说名代码没有特定的规定，都是为了方便

函数调用关系：

usrRoot->sysClkInit->sysClkConnect->sysHwInit2->vxbDevInit->vxbDevInitInternal->vxbDevIterate->vxbDevInit2Helper->

LOCAL STATUS vxbDevInit2Helper

(

struct vxbDev \* pInst,

void \* pArg

)

{

if ( pInst->pDriver->pDrvBusFuncs->devInstanceInit2 == NULL )

return(OK);

/\*这里调用第二阶段的初始化函数。\*/

(\*pInst->pDriver->pDrvBusFuncs->devInstanceInit2)(pInst);

pInst->flags |= VXB\_INST\_INIT2\_DONE;

return(OK);

}

这里调用的第二阶段初始化函数对应的函数是：

LOCAL VOID lsnIntCtlrInstInit2

(

VXB\_DEVICE\_ID pInst

)

{

vxbIntConnect(pInst, 0, vxbMipsLsnIntCtrlIsr, (void\*)pInst);

vxbIntEnable(pInst, 0, vxbMipsLsnIntCtrlIsr, (void\*)pInst);

return;

}

/\*可以看出这里所作的工作就只有挂接中断和使能\*/

这里中断挂接函数的函数调用关系：

vxbIntConnect->vxbDevIterate->vxbIntCtlrMatch->vxbIntCtlrConnect

直接看函数：熟悉流程即可：

STATUS vxbIntConnect

(

struct vxbDev \* pDev, /\* Device Information \*/

int index, /\* index of interrupt vector \*/

VOIDFUNCPTR pIsr, /\* ISR \*/

void \* pArg /\* parameter \*/

)

{

VXB\_ACCESS\_INTERRUPT\_INFO accessIntrInfo;

struct vxbintCtlrMgmt info;

VXB\_ASSERT(pDev!=NULL, ERROR)

info.found = FALSE;

info.pIntCtlr = NULL;

info.inputPin = 0;

info.pDev = pDev;

info.index = index;

info.pIsr = pIsr;

info.pArg = pArg;

/\*这里一次把所有实例化的设备和要注册的info进行匹配\*/

vxbDevIterate(vxbIntCtlrMatch, &info, VXB\_ITERATE\_INSTANCES);

if ( info.found == TRUE )

return(OK);

return (vxbDevControl (pDev, (pVXB\_DEVCTL\_HDR)&accessIntrInfo));

}

LOCAL STATUS vxbIntCtlrMatch

(

struct vxbDev \* pInst,

void \* pArg

)

{

struct vxbintCtlrMgmt \* pInfo = (struct vxbintCtlrMgmt \*)pArg;

FUNCPTR func;

STATUS stat;

struct plbIntrEntry \* pIntrEntry;

int inputPin;

struct pIntCtlrTable \* pICTab;

struct plbIntCtlrTable \* pICTab2;

int x;

int y;

struct plbIntCtlrEntry \* pEnt;

/\*在这里会调用匹配到的设备的中断挂接函数，换句话说对不同的中断控制器上的中断挂接会调用对应的中断控制器的额挂接函数\*/

func = vxbDevMethodGet(pInst, DEVMETHOD\_CALL(vxbIntCtlrConnect));

stat = (\*func)(pInst, pInfo->pDev, pInfo->index, pInfo->pIsr, pInfo->pArg, &inputPin);

if ( stat != OK )

return(OK);

pInfo->found = TRUE;

return(OK);

}

这里对应的中断挂接函数为：

LOCAL STATUS vxbMipsLsnIntCtlrDevIsrConnect

(

VXB\_DEVICE\_ID pIntCtlr,

VXB\_DEVICE\_ID pDev,

int indx,

void (\*pIsr)(void \* pArg),

void \* pArg,

int \* pInputPin

)

{

pVXB\_LSN\_INT\_DRV\_CTRL pDrvCtrl = pIntCtlr->pDrvCtrl;

int inputPin = ERROR;

struct intCtlrHwConf \* pIsrHandle = NULL;

inputPin = intCtlrPinFind (pDev, indx, pIntCtlr, &pDrvCtrl->isrHandle);

pIsrHandle = &(pDrvCtrl->isrHandle);

/\*在这里调用intCtlrISRAdd函数，这个函数会真正的挂接用户的中断函数\*/

if (intCtlrISRAdd(pIsrHandle, inputPin, pIsr, pArg) == ERROR)

return ERROR;

\*pInputPin = inputPin;

return OK;

}

STATUS intCtlrISRAdd

(

struct intCtlrHwConf \* pEntries,

int inputPin,

void (\*isr),

void \* pArg

)

{

struct intCtlrISRChainEntry \* pChain;

struct intCtlrISRChainEntry \* pChain2;

void \* installedIsr;

/\*这里要确保中断表已经创建 \*/

intCtlrTableCreate(pEntries, inputPin);

/\*这里创建两个结构体，并初始化为用户中断，用于挂接中断时使用\*/

pChain = (struct intCtlrISRChainEntry \*)hwMemAlloc(sizeof(\*pChain));

pChain2 = (struct intCtlrISRChainEntry \*)hwMemAlloc(sizeof(\*pChain));

pChain->isr = isr;

pChain->pArg = pArg;

/\*获取目前表中现在挂接的中断函数，如果是函数intCtlrStrayISR说明已经初始化为默认的中断函数，

只需要替换为用户的中断函数即可，如果为函数intCtlrChainISR，说明已经至少挂接了一个用户中断函数(多个中断函数可以挂接在一个中断引脚上)，

如果为空，说明没有进行过初始化，这时就使用pChain2保存要挂接的中断函数，并赋值为pChain，然后填写到对应引脚上\*/

installedIsr = intCtlrTableIsrGet(pEntries, inputPin);

if ( installedIsr == (void \*)intCtlrStrayISR )

{

intCtlrTableFill(pEntries, inputPin, pChain->isr, pChain->pArg);

}

else if ( installedIsr == (void \*)intCtlrChainISR )

{

pChain->pNext = (struct intCtlrISRChainEntry \*)intCtlrTableArgGet(pEntries, inputPin);

intCtlrTableFill(pEntries, inputPin, intCtlrChainISR, pChain);

}

else

{

/\* create chain, add original entry and the new one \*/

pChain2->flags = intCtlrTableFlagsGet(pEntries, inputPin);

pChain2->pArg = intCtlrTableArgGet(pEntries, inputPin);

pChain2->isr = installedIsr;

pChain->pNext = pChain2;

pChain2->pNext = NULL;

intCtlrTableFill(pEntries, inputPin, intCtlrChainISR, pChain);

}

return(OK);

}

struct intCtlrISRChainEntry

{

struct intCtlrISRChainEntry \* pNext;

void (\*isr)(void \* pArg, int inputPin);

void \* pArg;

UINT32 flags;

};

呼～～，到这里中断的额挂接就完成了。当然了这只是vxWorks系统中属于vxbus架构的函数的挂接，但是这就包含了绝大部分的中断呢。

3，非vxbus中断的挂接

下面说一下不是vxbus结构的中断函数的挂接。

不是vxbus架构的中断函数使用intConnect函数进行挂接，废话少说，看函数：

STATUS intConnect

(

VOIDFUNCPTR \* vector, /\* interrupt vector to attach to \*/

VOIDFUNCPTR routine, /\* routine to be called \*/

int parameter /\* parameter to be passed to routine \*/

)

{

/\*这里首先看到，进行了三次判断，分别代表的三种中断的挂接，是首先第一种\_func\_isrConnect，

      这是isrobjectn中断的挂接，目前对于isrobject还没有深入分析过，不过这必须打开#INCLUDE\_ISR\_OBJECT宏定义才可以，

      这里不运行；intArchConnect这个函数就是结构中断的额挂接，一个中断号挂接一个中断，当然了在这里执行不到；

   \_func\_vxbIntConnect这个函数才是这次的重点，这个函数在vxbLegacyIntInit中被赋值，

    \_func\_vxbIntConnect = vxbLegacyIntConnect，看一下这个初始化函数\*/

if (\_func\_isrConnect != NULL)

if (\_func\_isrConnect (vector, routine, parameter) == OK)

return (OK);

if (\_func\_vxbIntConnect != NULL)

if (\_func\_vxbIntConnect(vector, routine, parameter) == OK)

return (OK);

return (intArchConnect (vector, (VOIDFUNCPTR) routine,

(int) parameter));

}

void vxbLegacyIntInit()

{

VXB\_DEVICE\_ID devID;

struct plbIntrEntry \* pIntrInfo;

/\*申请一个legacy设备(legacy在这里代表不是vxbus架构设备也可以说为遗留设备)\*/

devID = vxbDevStructAlloc(WAIT\_FOREVER);

/\*初始化设备，主要是名字和总线号，用于之后匹配挂接于那个中断控制器\*/

devID->pName = "legacy";

devID->pParentBus = pPlbDev->u.pSubordinateBus;

devID->busID = VXB\_BUSID\_PLB;

devID->pRegBase[0] = (void \*)1;

pIntrInfo = (struct plbIntrEntry \*)hwMemAlloc(sizeof(\*pIntrInfo));

devID->pIntrInfo = pIntrInfo;

/\*声明设备已经注册到系统中\*/

if ( vxbDeviceAnnounce(devID) != OK )

{

vxbDevStructFree(devID);

return;

}

vxbLegacyIntDevice = devID;

\_func\_vxbIntConnect = vxbLegacyIntConnect;

\_func\_vxbIntDisconnect = vxbLegacyIntDisconnect;

\_func\_vxbIntEnable = vxbLegacyIntEnable;

\_func\_vxbIntDisable = vxbLegacyIntDisable;

}

LOCAL STATUS vxbLegacyIntConnect

(

VOIDFUNCPTR \* vector, /\* interrupt vector to attach to \*/

VOIDFUNCPTR routine, /\* routine to be called \*/

int parameter /\* parameter to be passed to routine \*/

)

{

/\*在这里调用vxbIntConnect函数把中断注册进vxbus架构的中断表中\*/

if ((vxbIntConnect(vxbLegacyIntDevice,(int)vector,routine,

(void \*)parameter)) != OK)

return ERROR;

return(vxbIntEnable(vxbLegacyIntDevice,(int)vector,routine,

(void \*)parameter));

}

之后就和上面vxbus中断的挂接一样了：

vxbIntConnect->vxbDevIterate->vxbIntCtlrMatch->vxbDevMethodGet->vxbIntCtlrConnect->vxbMipsLsnIntCtlrDevIsrConnect

OK...到这里非vxbus结构的中断挂接也完成了。

这里要说明一下：

func = vxbDevMethodGet(pInst, DEVMETHOD\_CALL(vxbIntCtlrConnect));这个函数会调用所有注册进系统并实例化的设备对应的vxbIntCtlrConnect函数来进行挂接，但是这里还有一个函数，确保了中断函数会注册进对应的中断控制器中inputPin = intCtlrPinFind (pDev, indx, pIntCtlr, &pDrvCtrl->isrHandle)，这个intCtlrPinFind函数只有在对应中断控制器中才可以找到对应的引脚结构体。这也就保证了会注册到对应的引脚结构体中。

ok～～，分析就先到这里，如果有什么不对地方，请留言。。。

---------------------

作者：傲世阿龍

来源：CSDN

原文：https://blog.csdn.net/u010383937/article/details/72897690?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

->https://blog.csdn.net/u010383937/article/details/72897690