密级：

受控状态：

发 放 号：

Intewell TTOS BSP开发手册

（V1.0）

拟　　制：_____

审　　核：_____

批　　准：_____

科东（广州）软件科技有限公司

年　　月　　日

修订历史记录

日期	版本	说明	修订人	文档更改单号
2020-06-05	V0.1	编写该文档	张敏	
2020-11-30	V1.0	根据需求修改对应点	张敏	

目 录

1 设备管理框架

1.1设备管理体系架构

TTOS 为了规范并简化设备管理，提出了设备统一管理框架(Device Manager,以下简称DM)。DM 对上层应用提供统一的设备访问接口(open/read/write/close/ioctl)，应用使用设备如同操作文件。

设备驱动仅需向 DM 注册上述对应的接口即可。设备驱动可以静态配置在utDR_DeviceDrivers 数组中，也可以调用 dm_install_drv 动态添加，区别在于前者的初始化函数在 DM 初始化时自动被调用，后者需要自己调用驱动的初始化函数。DM 整体框架如下图 1 所示。

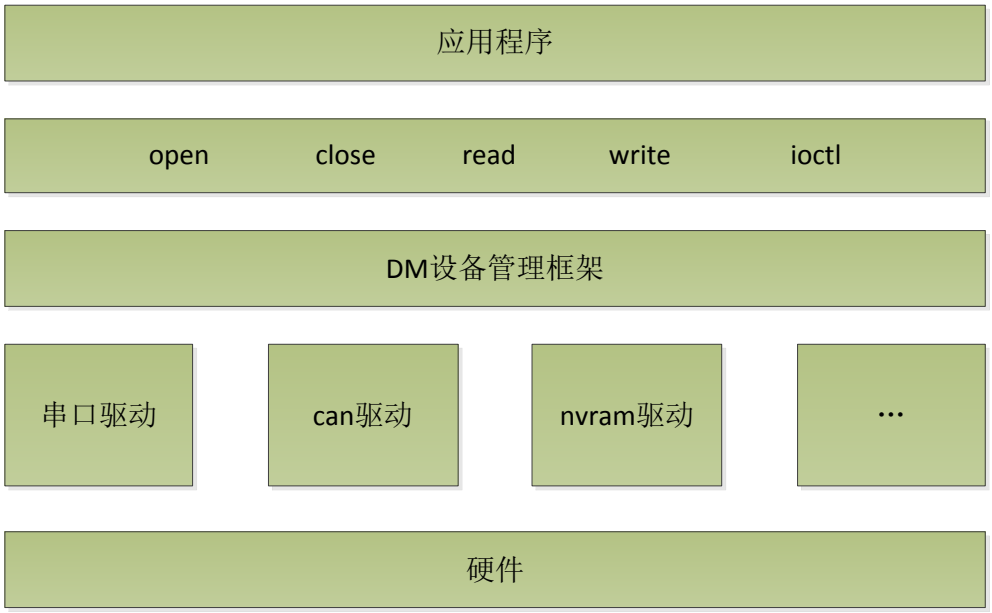

图 1 设备管理体系架构

1.2DM 初始化流程

DM 在虚拟槽中是作为组件存在的，故 DM 的初始化位于组件初始化中。从虚拟槽初始化（lmian）到 DM 初始化，再到用户 main 函数执行流程如图 2 所示。

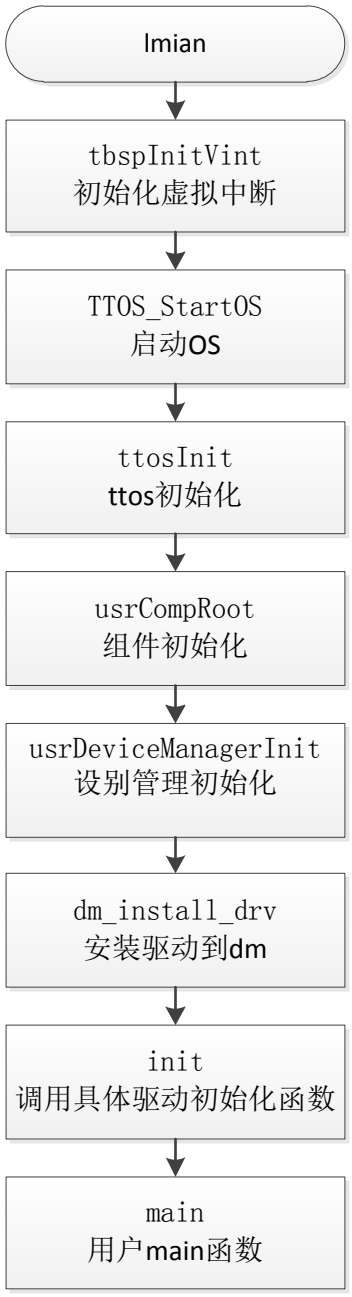

图 2 DM 初始化流程

dm_install_drv 步骤中依次安装 utDR_DeviceDrivers 数组中的所有驱动（静态方式注册到 DM 的驱动），同时调用驱动的 xxx_init 函数初始化。通常在 xxx_init 函数中动态添加驱动对应的硬件设备。

utDR_DeviceDrivers 数组中的元素是 T_DM_DAT 结构体，其规定了设备驱动必须提供的接口，具体定义如下：

```
typedef struct {
   DriverInit init;          /* initialization procedure */
   DriverOpen open;          /* open request procedure */
   DriverClose close;        /* close request procedure */
   DriverRead read;          /* read request procedure */
   DriverWrite write;        /* write request procedure */
```

```
    DriverIoctl control;        /* special functions procedure */
} T_DM_DAT;
```

注：

```
typedef T_WORD (*DriverInit)(T_WORD major, T_WORD minor, T_VOID *arg);
typedef T_WORD (*DriverOpen)(T_WORD value, const char* name, T_WORD flags, T_WORD mode);
typedef T_WORD (*DriverClose)(T_WORD value);
typedef T_WORD (*DriverRead)(T_WORD value, void* buffer, T_UWORD maxBytes);
typedef T_WORD (*DriverWrite)(T_WORD value, const void* buffer, T_UWORD maxBytes);
typedef T_WORD (*DriverIoctl)(T_WORD value, T_WORD function, T_WORD arg);
```

1.3DM 接口介绍

- T_WORD dm_add_dev(Dev_Hdr *pDevHdr, T_UBYTE *name, T_UWORD drvNum, Dev_Type type)

参数	描述
pDevHdr	设备管理头信息
name	设备名
drvNum	设备驱动索引号
type	设备类型
返回	--
ENONE	成功
其他错误码	失败

注：

设备管理头信息定义如下：

```
typedef struct
{
    T_TTOS_ChainNode node;              /* 设备链表节点 */
    T_UWORD driver_num;                 /*设备驱动索引号*/
    Dev_Type type;                      /* 设备类型 */
    T_UBYTE name[DM_NAME_LENGTH + 1];   /* 设备名 */
    T_WORD value;                       /* 设备驱动内部值 */
} Dev_Hdr;
```

设备类型定义如下：

```
typedef enum {
    DM_DEV_FATFS = 0,
    DM_DEV_YAFFSFS,
    DM_DEV_NET,
    DM_DEV_82574,
```

```
    DM_DEV_8139,
    DM_DEV_i350,
    DM_DEV_CHARACTER,
    DM_DEV_BLOCK_HD,
    DM_DEV_BLOCK_RD,
    DM_DEV_BLOCK_NF,
    DM_DEV_BLOCK_MTD,
    DM_DEV_NFS,
    DM_DEV_VNFS,
    DM_DEV_LAST,
} Dev_Type;
```

- T_WORD dm_delete_dev(Dev_Hdr *pDevHdr)

参数	描述
pDevHdr	设备管理头信息
返回	--
ENONE	成功
其他错误码	失败

- T_WORD dm_install_drv (DriverCreate pCreate, DriverDelete pDelete,DriverOpen pOpen, DriverClose pClose, DriverRead pRead, DriverWrite pWrite, DriverIoctl pIoctl, T_UWORD *drvNum)

参数	描述
pCreate	创建设备的函数
pDelete	删除设备的函数
pOpen	打开设备的函数
pClose	关闭设备的函数
pRead	设备接收数据的函数
pWrite	设备发送数据的函数
pIoctl	设备其他操作的函数
drvNum	设备管理头信息
返回	--
ENONE	成功
其他错误码	失败

- T_WORD dm_uninstall_drv(T_UWORD drvNum, BOOL forceClose)

参数	描述
drvNum	设备驱动索引号
forceClose	强制关闭设备驱动标记
返回	--

ENONE	成功
其他错误码	失败

2 新增驱动及验证

2.1 实现 DM 要求的基本接口

- INT32 xxx_init(T_WORD major, T_WORD minor, T_VOID *arg)

参数	描述
major	设备的 major 号
minor	设备的 minor 号
arg	其他参数
返回	--
ENONE	成功
其他错误码	失败

初始化函数中需要根据设备的数目创建设备管理头信息(Dev_Hdr)和设备控制块(T_DR_XxxControl，该结构由驱动定义，不同设备定义不一样)，再调用 dm_add_dev 将设备添加到 DM 中。

- INT32 xxx_open(void *arg, UINT8 *fileName, INT32 flags, INT32 mode)

参数	描述
arg	设备管理头信息
fileName	设备名
flags	打开设备的方式
mode	操作模式
返回	--
ENONE	成功
其他错误码	失败

- INT32 xxx_close(T_VOID *arg)

参数	描述
arg	设备控制块指针
返回	--
ENONE	成功
其他错误码	失败

- INT32 xxx_read(T_VOID *arg, T_BYTE *buffer, T_UWORD maxbytes)

参数	描述

arg	设备控制块指针
buffer	存放接收数据的地址
maxbytes	接收字节数
返回	--
接收的字节数	成功
其他错误码	失败

- INT32 xxx_write(T_VOID *arg, T_BYTE *buffer, T_UWORD nbytes)

参数	描述
arg	设备控制块指针
buffer	存放接发送据的地址
nbytes	发送字节数
返回	--
发送的字节数	成功
其他错误码	失败

- INT32 nvram_control(T_VOID *args, T_WORD function, T_WORD arg)

参数	描述
args	设备控制块指针
function	ioctl 操作功能
arg	ioctl 操作参数
返回	--
ENONE	成功
其他错误码	失败

2.2组织 T_DM_DAT 结构体

把 2.1 中实现的接口组成 T_DM_DAT 结构体：

```
#define DR_XXX_DRIVER_ENTRY \
{ (T_DM_DeviceDriverInitEntry) xxx_init,\
(T_DM_DeviceDriverEntry) xxx_open,\
(T_DM_DeviceDriverEntry) xxx_close, \
(T_DM_DeviceDriverEntry) xxx_read, \
(T_DM_DeviceDriverEntry) xxx_wirte,\
(T_DM_DeviceDriverEntry) xxx_ioctl }
```

2.3添加到 utDR_DeviceDrivers 数组中

将 2.2 中的 DR_NVRAM_DRIVER_ENTRY 置于 utDR_DeviceDrivers 数组中：

```
T_DM_DAT utDR_DeviceDrivers[] = {
```

```
        #if CONFIG_DEVICE_XXX
            DR_XXX_DRIVER_ENTRY,
        #endif
    }
```

DM 初始化的时候自动安装 utDR_DeviceDrivers 中的驱动，并调用驱动的初始化函数(xxx_init)。

2.4验证驱动

以串口为例子：

```
int fd = -1;
int ret = -1;
char buf[10] = {0};
T_DR_UartConfigTable uartCfg;
fd = open("/COM1", MODBUS_IO_BLOCK);
if(fd < 0)
{
    printf("open failed");
    return -1;
}
/* 获取串口配置 */
ret = ioctl(fd, MODBUS_IO_GET_ATTR, &uartCfg);
if (ret < 0)
{
    printf("ioctl failed: MODBUS_IO_GET_ATTR\n");
    return -1;
}
printf("baudrate: %d\n", uartCfg.baudrate);
uartCfg.baudrate = 57600;
ret = ioctl(fd, MODBUS_IO_SET_ATTR, &uartCfg);
if (ret < 0)
{
    printf("ioctl failed: MODBUS_IO_SET_ATTR\n");
    return -1;
}
/* 获取串口配置 */
ret = ioctl(fd, MODBUS_IO_GET_ATTR, &uartCfg);
if (ret < 0)
{
    printf("ioctl failed: MODBUS_IO_GET_ATTR\n");
```

```
        return -1;
    }
    printf("baudrate: %d\n", uartCfg.baudrate);
    /* 接收数据 */
    ret = read(fd, buf, 10);
    if(ret < 0)
    {
        printf("read failed");
        return -1;
    }
    printf("read buf: %s\n", buf);
    /* 发送数据 */
    write(fd, buf, 10);
    sleep(1);
    ret = close(fd);
    if(ret < 0)
    {
        printf("close failed");
        return -1;
    }
```

2.5 中断配置

外部中断只能投递给某一个虚拟槽(Virtual Machine，以下简称 VM)，故外部中断需要和 VM 中的虚拟中断进行映射，配置流程如图 3 所示。

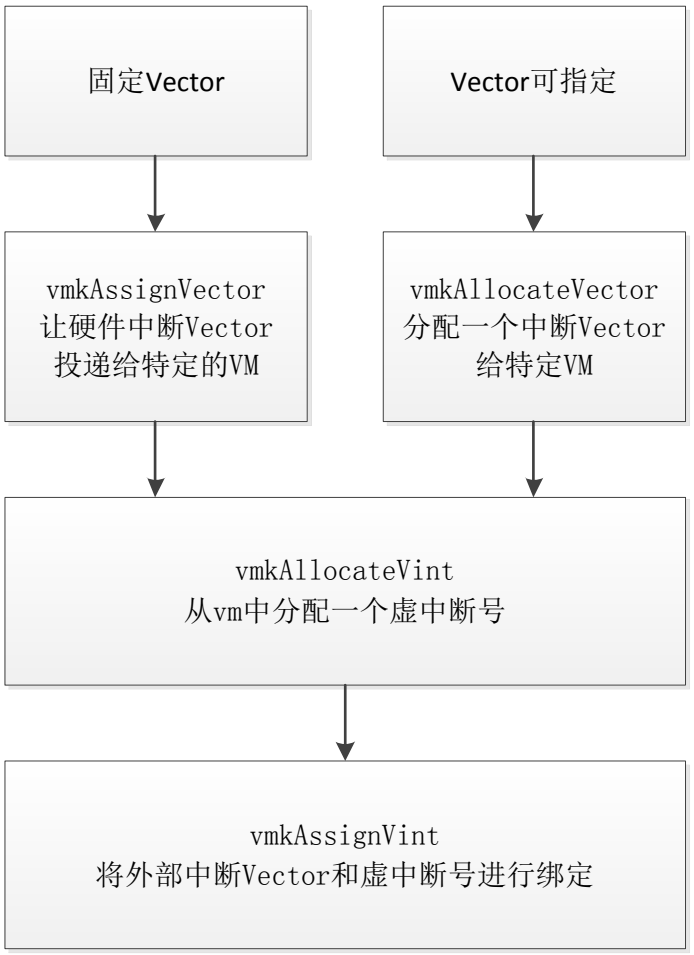

图 3 中断配置流程

对于 X86 平台，如果打开 VT-d 功能，最后还要调用 vtdPciDevsIrqRemapping 更新 IRTE 表项。

- status_t vtdPciDevsIrqRemapping(uint8_t bus, uint8_t dev, uint8_t func, uint32_t index, int32_t vector, uint32_t dest)

参数	描述
bus	总线号
dev	设备号
func	功能号
index	IRTE 表项索引
vector	外部中断 Vector
dest	中断投递的 CPU
返回	--
vector	成功
INVALID_VECTOR	失败

- T_UWORD vmkAllocIrte(void)

分配一个空闲的 IRTE 表项索引，返回值即为空闲的 IRTE 表项索引

注：

以下的接口都以 vmk 开头，指示仅内核态可以访问这些接口，故通常在 setNicMsixInfo.c 文件中完成中断的映射，具体映射流程参考 setSJA1000CanMsixInfo 函数的实现。

- T_UWORD vmkAssignVector(const T_UWORD vmID, const T_UWORD vector)

参数	描述
vmID	外部中断投递的 VM ID
vector	外部中断 Vector
返回	--
vector	成功
INVALID_VECTOR	失败

- T_UWORD vmkAllocateVector(const T_UWORD vmID, T_UBYTE vmPriority)

参数	描述
vmID	外部中断投递的 VM ID
vmPriority	VM 的优先级
返回	--
vector	成功，外部中断 Vector
INVALID_VECTOR	失败

- T_WORD vmkAllocateVint(T_VMK_ConfigExintMap *vmExintMap, const T_UWORD vector)

参数	描述
vmExintMap	VM 外部中断映射配置表
vector	外部中断 Vector
返回	--
Vint	成功，VM 虚拟中断号 Vint
-1	失败

- T_WORD vmkAssignVint(T_VMK_ConfigExintMap *vmExintMap, const T_UWORD vector, const T_UWORD vInt)

参数	描述
vmExintMap	VM 外部中断映射配置表
vector	外部中断 Vector
vInt	VM 虚拟中断号 Vint
返回	--
vInt	成功，VM 虚拟中断号 Vint
-1	失败

2.6安装和使能中断

- T_TTOS_ReturnCode TTOS_InstallIntHandler(T_UBYTE intNum, T_TTOS_ISR_HANDLER handler, T_TTOS_ISR_HANDLER *oldHandler)

参数	描述
intNum	VM 虚拟中断号 Vint
handler	中断处理函数
oldHandler	未使用，传入 NULL 即可
返回	--
TTOS_OK	成功
TTOS_FAIL	失败

注意：

typedef T_BOOL (*T_TTOS_USER_EXCEPTION_HANDLER) (T_UWORD vector,T_ExceptionContextWithType *context);

typedef T_VBSP_EXINT_HANDLER T_TTOS_ISR_HANDLER;

- T_TTOS_ReturnCode TTOS_EnablePIC(T_UBYTE intNum);

参数	描述
intNum	VM 虚拟中断号 Vint
返回	--
TTOS_OK	成功
其他返回值	失败

3 PCI/PCIe

3.1访问 PCI/PCIe 的接口

- pciReadConfig[Byte/Word/Dword](T_UBYTE bus,T_UBYTE device_fn,T_UBYTE where, [T_UBYTE/T_UHWORD/T_UWORD] *value)

参数	描述
bus	总线号
device_fn	设备功能号
where	寄存器号
value	存放读取到的数据
返回	--
PCIBIOS_SUCCESSFUL	成功

- pciWriteConfig[Byte/Word/Dword](T_UBYTE bus,T_UBYTE device_fn,T_UBYTE where,

[T_UBYTE/T_UHWORD/T_UWORD] value)

参数	描述
bus	总线号
device_fn	设备功能号
where	寄存器号
value	待写入的数据
返回	--
PCIBIOS_SUCCESSFUL	成功

3.2PCIe 配置空间映射

I/O 端口只能访问 PCI/PCIe 配置空间的前 256 字节(0-255 Byte)，对于比较复杂的 PCI/PCIe 设备，其特殊功能需要用到 256-4K 的配置空间，只能通过内存映射的方式进行访问。

- T_WORD pcieConfigSpaceMapping(T_UBYTE bus,T_UBYTE device_fn)

参数	描述
bus	总线号
device_fn	设备功能号
返回	--
PCIBIOS_SUCCESSFUL	成功
-ENOMEM	失败

pcie[Read/Write]Config[Byte/Word/Dword]接口格式和功能与 pci[Read/Write]Config[Byte/Word/Dword]一样，区别在于前者需要提前将 PCIe 的配置空间映射到内存，后者不需要映射，而是通过端口访问配置空间的前 256 字节数据。

4 网卡驱动

4.1网卡驱动初始化流程

网卡目前有 82574 和 i350 的驱动，网卡驱动没有挂载到 DM 上，而是与 LWIP 网络协议栈关联。LWIP 网络协议栈是 TTOS 的组件，初始化流程如图 4 所示。

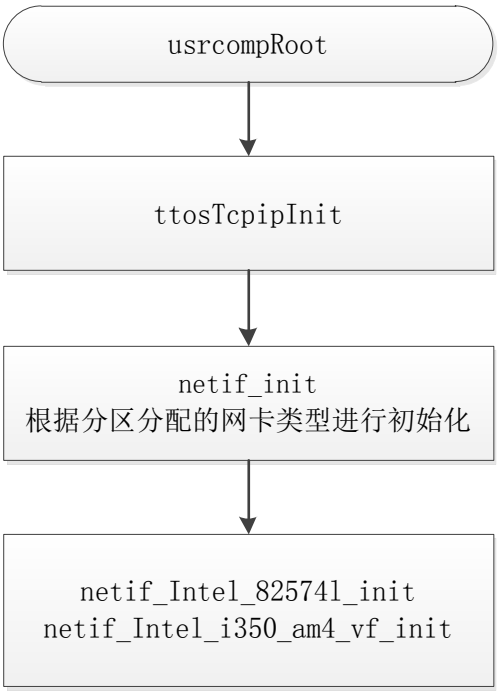

图 4 网卡驱动初始化流程

netif_Intel_82574l_init和netif_Intel_i350_am4_vf_init分别调用各自的初始化函数。

4.2新增网卡驱动

4.2.1 实现网卡驱动接口

- static err_t low_level_output(struct netif *netif, struct pbuf *p)

网卡发送数据的函数

参数	描述
netif	网络接口
p	需要发送的数据包
返回	--
ERR_OK	成功
其他错误	失败

- static int netif_isr_task_init(void *entry, void *arg)

创建处理网卡中断的任务

参数	描述
entry	中断处理函数
arg	中断处理函数的参数
返回	--
0	成功
-1	失败

- static T_TTOS_TaskReturnAction netif_isr_task_routine(T_VOID *arg)

网卡中断处理函数

参数	描述
arg	中断处理函数的参数
返回	--
TTOS_TASK_STOP	该任务执行完成后停止自己
TTOS_TASK_RESET	该任务执行完成后重启自己

注：

```
/* 任务执行完成后返回操作类型 */
typedef enum
{
    /* 表示该任务执行完成后停止自己 */
    TTOS_TASK_STOP = 0,
    /* 表示该任务执行完成后重启自己 */
    TTOS_TASK_RESET = 1
}T_TTOS_TaskReturnAction;
```

```
T_INLINE T_TTOS_ReturnCode TTOS_CreateTaskEx(
T_UBYTE *taskName,
T_UBYTE taskPriority,
T_UBYTE autoStarted,
T_UBYTE preempted,
T_TTOS_TaskRoutine entry,
T_VOID * arg,
T_UWORD stackSize,
TASK_ID *taskID
)
```

参数	描述
taskName	任务名称
taskPriority	任务优先级
autoStarted	是否自启动
preempted	是否可被抢占
entry	任务运行的函数
arg	运行函数的参数
stackSize	任务栈大小
taskID	任务 ID
返回	--
TTOS_OK	成功
其他错误码	失败

注：
typedef T_VOID (*T_TTOS_TaskRoutine)(T_VOID *arg);

- static int low_level_init(struct netif *netif)

初始化网卡硬件

参数	描述
netif	网络接口
返回	--
0	成功
-1	失败

4.2.2 新增网卡驱动与 LWIP 关联

netif_init 函数中添加新增网卡初始化分支，并调用 netif_X_init 初始化分配给 VM 的新增网卡。具体细节参考 82574 和 i350 网卡初始化分支。

- void* netif_X_init(u8_t minor, char *name, char *ip, char *mask, char * gateway)

参数	描述
minor	网络接口索引号
name	网卡名称
ip	IP
mask	掩码
gateway	网关
返回	--

5 驱动框架未来改进计划

5.1优化 2.3.3.2 配置中断重映射

- 该使 VT-d 功能下的中断重映射机制对驱动开发者透明，开发者只需要关注具体设备和虚拟中断即可。

5.2优化 DM 管理设备

- dm 管理所有设备。
- 设备驱动实现 DriverCreate 和 DriverDelete，DriverCreate 用于根据设备的数目创建设备管理头信息和设备控制块，DriverDelete 用于释放 DriverCreate 申请

的内存。

6 vector 与 vint 的分配与映射

TTOS 中，设备 vector 与 vint 的分配有两种方式：静态配置与动态分配。

6.1 动态分配

TTOS 提供了如下两个接口，来支持动态分配 vector 与 vint 及建立二者之间的映射关系:

a) TTOS_ AllocVector

接口原型：

T_WORD TTOS_ AllocVector (T_UWORD *vector)

头文件:

ttos.h

接口说明：

该接口分配一个外部中断号,若(*vector)的值不为 TTOS_INVALID_VECTOR,则尝试分配(*vector)指定的外部中断，否则，动态分配 vector 号。

参数说明：

vector: 外部中断号的地址

返回值说明:

TTOS_UNSATISFIED：无空闲的外部中断号；

TTOS_INVALID_ADDRESS：非法的参数；

TTOS_OK：分配成功成功。

b) TTOS_MapExintToVint

接口原型：

T_WORD TTOS_MapExintToVint (T_UWORD vector, T_UWORD *vint)

头文件:

ttos.h

接口说明:

该接口动态分配虚拟中断并建立虚拟中断与硬件中断之间的映射。

参数说明:

vector: 硬件中断号

vint: 虚中断号指针,

返回值说明:

TTOS_INVALID_ADDRESS：无效的参数地址。

TTOS_INVALID_SIZE: vector 大小不合法。

TTOS_OK：映射成功。

接口示例

```
#include <ttos.h>

/* 中断接口使用示例 */
int interrupt_sample(void)
{
    /*指定分配 65 号 vector*/
    //T_UWORD vector = 65;
    /* 动态分配 vector*/
    T_UWORD vector = TTOS_INVALID_VECTOR;
    T_UWORD vint;
    T_TTOS_ReturnCode ret;

    /* 分配外部中断号 */
    ret = TTOS_AllocVector(&vector);
    if (TTOS_OK != ret)
    {
        return (-1);
    }

    /* 分配 vint 并映射外部中断 */
    ret = TTOS_MapExintToVint(vector, &vint);
    if (TTOS_OK != ret)
    {
        return (-1);
    }

    /* 安装中断处理函数 */
    TTOS_InstallIntHandler(vint, isr_handler, NULL);

    /* 使能中断 */
    TTOS_EnablePIC(vint);

    return (0);
```

```
}
```

6.2 静态配置

1. 在配置项目中，静态设置将 4 号 vector 分配给 vm1

2. 在配置项目中，将 4 号 vector 与 4 号虚拟外部中断建立映射

示例：

```
int interrupt_sample(void)
{
    /* 安装中断处理函数 */
    TTOS_InstallIntHandler(4, isr_handler, NULL);
    /* 使能中断 */
    TTOS_EnablePIC(4);
    return 0;
}
```

7 FAQ

◆ 设备的配置空间如何访问，是否要使用 PCI 驱动，是否需要自己映射？

设备的配置空间可以通过端口和内存(自行映射)的方式访问；PCI/PCIe 配置空间物理地址和虚拟地址的映射目前需要自己调用 pcieConfigSpaceMapping 接口进行映射。具体细节详见第 4 章。

◆ 设备的 vector 等如何指定？

不同平台外设 vector 组织方式可能不一样，以下解释基于 X86 平台。

对于普通中断线的方式的设备 vector 是固定的，虚拟槽分配一个虚拟中断号与之对应；对于 MSI 中断的 vector 则通过 vmkAllocateVector 分配一个 vector，虚拟槽再分配一个虚拟中断号与之对应。具体细节详见第 3 章。

◆ 基于虚拟化以后，如何访问硬件？

目前虚拟化对设备的访问无影响，只对设备中断投递过程有影响。

- 哪些设备需要向 DM 注册，哪些不需要？

  目前只有网卡和 hpet 未向 DM 注册，未来计划所有设备都需要向 DM 注册。
- 网卡适配后，如何挂接到协议栈

  详见 5.2.2 节。
- 驱动故障的影响范围？

- 驱动如何映射 IO 空间？

  通过设备的总线号、设备号和功能号，调用 pcieConfigSpaceMapping 映射即可。详见 4.2 节。
- 驱动能否使用 OS API，哪些 API 不能在驱动中使用？

  驱动能用所有的 OS API，API 接口详见《TTOS 软件编程手册》和《TTOS 软件参考手册》。驱动中不能调用的 API 主要是驱动的中断处理函数中不能调用具有阻塞性的接口。
- 设备驱动入口如何挂接到 OS？

  详见 2.2 和 2.3 节。
- 添加新设备后，IDE、HACL 需要做哪些工作

  对于 IDE(Intewell Developer)需要支持获取硬件配置(新设备的)，用于在配置项目中生成设备的硬件信息，如中断号、设备名称等。对于 HACL 需要增加的对 IO APIC 的配置，IOMMU 的配置。
- DMA 32 和 64 位地址问题
- lfs 初始化 local APIC 问题和共享问题
- lfs 中确认中断已经可以投递到实时：dmesg | grep irq