## 因應系統主控制器

# PCI驅動程式撰寫大不同

Victor

PCI是一種可以擴充周邊元件的匯流排，在個人電腦上非常普遍，為一個共同的標準介面，可以讓很多的周邊元件附加在個人電腦上，雖然現在的嵌入式系統已經把很多的周邊系統做成系統單晶片(SoC)的中央處理器，但有時候在成本的考量下，PCI可以選擇所要擴充的周邊，避免SoC的中央處理器(CPU)成本過高，十分具有優勢。另外，把過多的周邊元件做成SoC，會造成CPU用電過高，更進而造成過高的溫度，必須使用其他的散熱元件，如鋁片或風扇等，反而造成產品體積變大並增加成本。

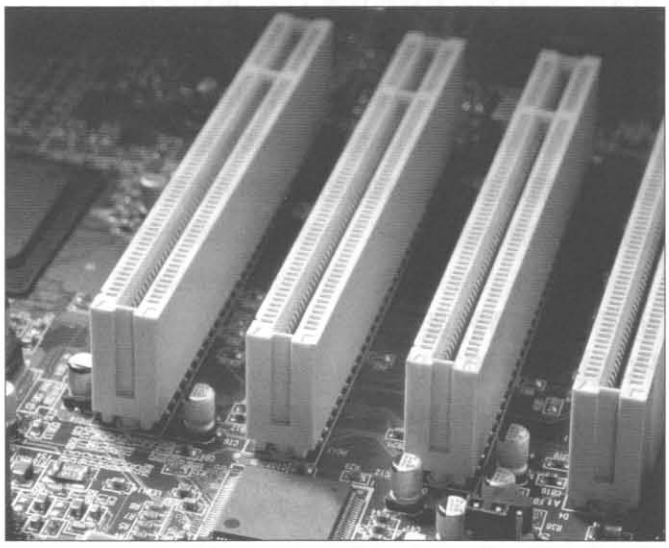

圖1 不同於x86的PCI主控端晶片，ARM架構的控制器種類繁多。

PCI的另一個好處是隨插即用(PNP)，對資源的分配及取得也具有相當大的彈性，如此一來，軟體撰寫者也可受惠，除軟體較容易移植至不同的CPU平台外，也不須遷就不同的CPU而撰寫不同的程式。

## ARM控制器製造商多 PCI軟體須分別撰寫

以硬體來說，PCI不同處在於它本身是一個高速傳輸標準；至於在軟體方面，因為x86只有一種CPU，其PCI主端控制晶片都一樣，並不會因為不同的生產廠商有所不同，因此在PCI主端的驅動程式都由作業系統廠商撰寫，但是採用安謀國際(ARM)所生產的PCI主端控制器都不同，所以軟體人員必須各別撰寫，以應付不同的中央處理器，故必須了解如何撰寫PCI主端驅動程式(圖1)。

此外，PCI採取主從式架構，主端(Host)由主機板廠商負責，以x86平台而言，通常會留下幾個標準的擴充槽，讓使用者購買其他從端(Client)的擴充卡增加周邊元件，在以ARM為基礎的嵌入系統中，PCI主端會是在CPU內的SoC，對於從端的元件，原則上不會做成插卡式的擴充槽，通常是直接將PCI匯流排的訊號線從PCI主端連接至PCI從端，如此一來就可以節省空間及成本。

| 31 | 16 15 | | 0 | |
|---|---|---|---|---|
| Device ID | | Vendor ID | | 00h |
| Status | | Command | | 04h |
| Class Code | | | Revision ID | 08h |
| BIST | Header Type | Lat. Timer | Cache Line S. | 0Ch |
| Base Address Registers | | | | 10h<br>14h<br>18h<br>1Ch<br>20h<br>24h |
| Cardbus CIS Pointer | | | | 28h |
| Subsystem ID | | Subsystem Vendor ID | | 2Ch |
| Expansion ROM Base Address | | | | 30h |
| Reserved | | | Cap. Pointer | 34h |
| Reserved | | | | 38h |
| Max Lat. | Min Gnt. | Interrupt Pin | Interrupt Line | 3Ch |

圖2 結構表

## PCI逐一分配周邊元件資源

在開機程序中，開機時PCI主端會先初始化自我控制器，之後再將可分配及使用的資源和系統登錄，掃描是否有任何PCI從端的周邊元件連線上系統，一旦確認連線後，PCI主端則依照要求分配資源給其他周邊元件。分配結束之後，PCI主端須檢查是否有相對應的從端登錄驅動程式，以呼叫其所登錄的Call Back Function，Call Back Function則由PCI從端的驅動程式登錄，在Call Back Function中通常會完成元件初始化，PCI主端將會逐一針對連結上的PCI從端執行重覆的動作，直到完成所有周邊元件的資源分配。

## 擔任主/從端橋梁 結構表擔任資源交換依據

在PCI主端和PCI從端之間存在結構表(Configuration Table)，該表具有二百五十六字元，並作為主端和從端之間溝通資源分配及資訊交換的依據(圖2)，此表存放在PCI從端上，其中Vendor ID & Device ID屬於唯讀，並交由PCI主端儲存。PCI從端也是用這兩個ID向主端註冊，PCI主端也會依照此ID進行比對，當前面所提要呼叫從端時，才知道呼叫的驅動程式。PCI主端要求從端所執行的動作稱為指令，這些指令在PCI的標準受到定義。狀態是目前從端現狀的表示，也在PCI標準中有所定義。

另外，非常重要的項目還包括Base Address Registers，它主要是用來描述PCI從端和主端的資源，而且它是一個可以雙向溝通的資料，對32位元的設備而言，總共最多有六個資源可以取得；若是64位元的設備，則最多只有三個資源可以取得，其實作法很簡單，就是將兩個32位元合成一個64位元。

在此僅針對32位元的設備進行說明，因為目前64位元的嵌入式並不多見，在說明之前必須了解，當CPU和周邊進行溝通可以從有兩個路徑著手，分別為輸入/輸出(I/O)存取及記憶體存取。由於硬體有所差異，所以必須透過欄位使用最低的兩個位元，以表示需要的資源。以二進位來看，00代表需要記憶，01代表需要I/O，所要求的大小從位元2開始，若不需要則以零作為表示，不然則填上需要的數值大小，當PCI主端掃描之後，會填上從端周邊所分配到的起始位置，才完成資源的分配。

## 毋須改變硬體架構 PCI主端僅改寫HAL

在了解PCI匯流排的基本定義之後，接著必須了解實作部分。在PCI主端方面，因為前面所提的部分標準和硬體無關，或者可以說，硬體工作內容一樣，因為這部分已由GNU's Not Unix(GNU)撰寫完成，不須進行任何更動。這部分的原始碼在核心原始碼(Kernel Source)的drivers/pci目錄之下，使用者所要寫的是硬體抽象(HAL)層部分，必須在arch/arm/mach-xxxx的目錄之下，這個目錄是使用者移植(Porting)SoC時建立的目錄，其中的xxxx是使用者自行定義的名稱)，撰寫一個PCI主端的一個C程式，而在這程式有幾個部分須要撰寫，首先須要宣告一個資料結構如下：

```
struct hw_pci ixdp425_pci __initdata = {
        .nr_controllers = 1,
        .preinit        = ixdp425_pci_preinit,
        .swizzle        = pci_std_swizzle,
        .setup          = ixp4xx_setup,
        .scan           = ixp4xx_scan_bus,
        .map_irq        = ixdp425_map_irq,
};
```

在此先說明一下這個資料結構的定義，其資料結構原型為struct hw_pci，其定義在arch/arm/mach/pci.h，這是核心版本2.6.31之後，ixdp425_pci則是使用者自行為資料稱呼的名字，__initdata定義為這個資料在核心初始化之後，會給予Free收回使用，至於各個資料成員的說明如下。

.nr_controllers目的在指示PCI主端控制晶片的數量，正常來說，只有一個PCI主控端晶片。至於.preinit則是在初始化該PCI主端晶片組的Call Back Function，這也和每一個SoC有所不同，所必須進行的初始化內容也依SoC不盡相同，當註冊一個PCI主端時，第一個被呼叫的副程式，若有需要中斷的設定，必須要在此完成和系統註冊。

.setup則是一個Call Back Function Pointer，這個副程式的原型為int ixp4xx_setup(int nr, struct pci_sys_data *sys)，系統在呼叫PCI主控端晶片的時候，將會傳送一個序號nr及一個指標struct pci_sys_data *sys給PCI主控端晶片，這個指標須要填入這PCI主控端晶片可以使用的資源，以下是一個範例：

```
struct resource *res;
res = kzalloc(sizeof(*res) * 2, GFP_KERNEL);
res[0].name = "PCI I/O Space";
res[0].start = 0x00000000;
res[0].end = 0x0000ffff;
res[0].flags = IORESOURCE_IO;
res[1].name = "PCI Memory Space";
res[1].start = 0x4b000000;
res[1].end = 0x4bffffff;
res[1].flags = IORESOURCE_MEM
request_resource(&ioport_resource, &res[0]);
request_resource(&iomem_resource, &res[1]);
sys->resource[0] = &res[0];
sys->resource[1] = &res[1];
sys->resource[2] = NULL;
```

從以上的註冊系統將會保留所要的資源，而在之前所提的標準的PCI驅動器會協助分配資源給PCI從端，所以之後如何分配及管理這些資源，就不用費心了。.scan是系統要開始掃描在PCI匯流排有插上那些PCI從端時會呼叫的副程式，通常都會直接再輾轉呼叫系統的掃描程式，以下是其範例程式：

```
return pci_scan_bus(sys->busnr, &ixp4xx_ops,
sys);
```

在以上的sys變數為系統傳過來的，ixp4xx_ops則是使用者所要準備的一些有關PCI匯流排設定的一些HAL Call Back Function，其內容為如下：

```
struct pci_ops ixp4xx_ops = {
 .read =  ixp4xx_pci_read_config,
 .write = ixp4xx_pci_write_config,
};
```

上面的read/write為在讀寫前面所談的結構表的方式，這和每一個晶片不同，其控制的位置也不同，所以根據每一顆系統單晶片來寫，而這兩個的函數原型如下：

```
int ixp4xx_pci_read_config(struct pci_bus
*bus, unsigned int devfn, int where, int size,
u32 *value)
int ixp4xx_pci_write_config(struct pci_bus
*bus,  unsigned int devfn, int where, int size,
u32 value)
```

在系統呼叫進來的匯流排資料結構中成員bus->number是這個系統中第幾個匯流排，在控制晶片時可能會用到，devfn則是slot number和function number的結合，可以使用巨集指令PCI_SLOT()或PCI_FUNC()分別取得其資訊，size則是表示1、2或4位元組的存取，where則是須要特別注意，它是在存取1、2位元組會用到，在二百五十六位元組結構表的相對位置，大部分的PCI主控端晶片會有boundary存取的問題，這邊須要特別注意做轉換，value則是較簡單，是讀取或寫入所放的值。

.map_irq則是當PCI Host在掃描時在要指定IRQ時會呼叫這個副程式，而這個副程式則會回傳要指定的中斷號碼。

當以上的資料準備好之後，你須要撰寫一個初始化的進入點，讓系統開機會呼叫這個初始化副程式，以下為其範例：

```
int __init ixdp425_pci_init(void)
{
    return pci_common_init(&ixdp425_pci);
}
subsys_initcall(ixdp425_pci_init);
```

由以上的程式碼可以看到，該程式會直接呼叫系統的pci_common_init()而且給它前面所提的資料結構，當呼叫該系統程式時，系統會馬上呼叫所給的資料結構中的Preinit Call Back Function，之後會呼叫Setup Call Back Function，再接著就是呼叫Scan Call Back Function，如此而完成一個PCI主端的載入動作(圖3)。

最後記得使用巨集指令subsys_initcall()，將所完成的副程式註冊為一個開機會被呼叫的副程式。

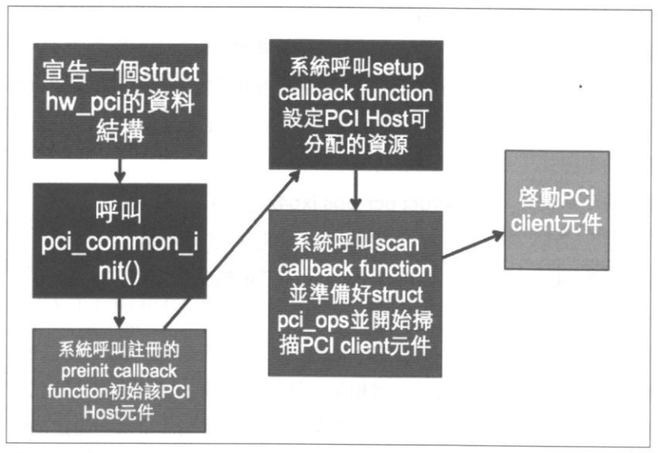


圖3 PCI載入動作流程圖

## 透過暫存器產生I/O訊號 化解ARM僅具備記憶體存取限制

原則上，上述應該就完成PCI主端的移植，因為在移植時只要完成這部分會和硬體有關的硬體抽象層即可，但是還是有一些地方必須注意，首先，在x86的平台上是有所謂的I/O存取，CPU本身就有此支援，CPU對周邊元件的存取方式，本來就有兩種，一種是記憶體存取，一種是I/O存取，而在PCI匯流排也有這兩種的存取方式，所以這兩種不同的存取方法，被視為兩種不同的資源，但是在ARM中央處理器中，沒有I/O存取，僅有記憶體存取，因此撰寫驅動程式會針對I/O存取使用inb()、oub()、inw()、outw()、inl()、outl()等副程式，若是記憶體存取則會使





用readb()、writeb()、readw()、writew()、readl()、writel()等副程式，因為ARM中央處理器並沒有I/O存取的功能，所以在HAL層會改寫inx()、outx()等副程式內容，改成使用記憶體存取(圖4)。

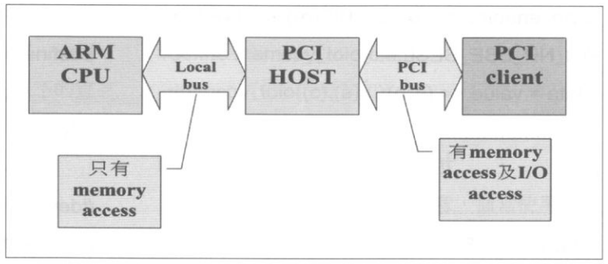


圖4 CPU存取示意圖

另外，I/O存取驅動程式所使用的資源，並不會將其轉換為虛擬位址，可能產生非法存取記憶體位置，因為核心須要使用虛擬位址驅動所有的程式碼，這一個部分可透過修改PCI從端的驅動程式碼化解疑慮，但缺點是原始碼只能使用在ARM平台，其他如x86的平台將無法採用相同的方法，另外也必須面臨修改驅動程式，因此並不是受歡迎的選擇。另一個解決方法是修改PCI主端的資源分配，就是之前有提到在初始化時須要將可分配的資源向系統註冊，註冊前先將其轉換為虛擬位址，再向系統註冊，此方法的優點為修改幅度小，且可避免如前一個方法的疑慮。

值得注意的是，PCI主端的晶片有兩種，一種是其本身對PCI匯流排兩種不同存取方式的自動轉換，如果PCI從端是使用I/O存取方式的周邊，它會自動將ARM中央處理器的記憶體存取方式，自動轉換為I/O存取方式，所以將可達成前面對於第二種I/O存取的修改方式，而對於ARM中央處理器的記憶體存取方式，則直接轉換至PCI匯流排即可，對於PCI主控端晶片可省略很多工夫。但是有另外一種PCI匯流排晶片卻無法依此方法，這種晶片比較麻煩，必須將其中的一些暫存器在PCI匯流排上產生I/O存取的訊號，以下則是其寫法說明。

首先，必須在arch/arm/mach-xxxx的目錄之下，建立一個檔案include/mach/io.h，注意其路徑及檔名不可更改，因為在整個核心原始碼中的Makefile及編譯時要尋找C語言的include file路徑都已寫好，更改將會造成編譯錯誤，在這個檔案之中將可以改寫ARM中央處理器已內定改好的inx()及outx()的副程式，若在這個檔案之中宣告如下：

```
#define __io(v)          __typesafe_io(v)
```

表示使用ARM中央處理器中已改好的inx()及outx()副程式，這是使用PCI主端會自動轉換I/O存取至PCI匯流排的晶片，若不是所需要的修改如下：

```
#define  outb(p, v)    __ixp4xx_outb(p, v)
#define  outw(p, v)    __ixp4xx_outw(p, v)
#define  outl(p, v)    __ixp4xx_outl(p, v)
#define  outsb(p, v, l)  __ixp4xx_outsb(p, v, l)
#define  outsw(p, v, l)  __ixp4xx_outsw(p, v, l)
#define  outsl(p, v, l)  __ixp4xx_outsl(p, v, l)
#define  inb(p)          __ixp4xx_inb(p)
#define  inw(p)          __ixp4xx_inw(p)
#define  inl(p)          __ixp4xx_inl(p)
#define insb(p, v, l)    __ixp4xx_insb(p, v, l)
#define  insw(p, v, l)   __ixp4xx_insw(p, v, l)
#define  insl(p, v, l)   __ixp4xx_insl(p, v, l)
```

設計者必須針對每一個副程式重新定義至另一個改寫的內容，而這些改寫的內容，則是針對所使用的PCI主端晶片進行控制，請參考以下的範例：

```
static inline void
__ixp4xx_outb(u8 value, u32 addr)
{
  u32 n, byte_enables, data;
  n = addr % 4;
```

```
  byte_enables = (0xf & ~BIT(n)) << IXP4XX_
PCI_NP_CBE_BESL;
  data = value << (8*n);
  ..............................
}
..............................
static inline u8
__ixp4xx_inb(u32 addr)
{
  u32 n, byte_enables, data;
  n = addr % 4;
  byte_enables = (0xf & ~BIT(n)) << IXP4XX_
PCI_NP_CBE_BESL;
  ..............................
  return data >> (8*n);
}
..............................
```

另外幾個和I/O有關的副程式也必須重新改寫如下：

```
#define   ioread8(p)     __ixp4xx_ioread8(p)
#define   ioread16(p)    __ixp4xx_ioread16(p)
#define   ioread32(p)    __ixp4xx_ioread32(p)
#define   ioread8_rep(p, v, c)
          __ixp4xx_ioread8_rep(p, v, c)
#define   ioread16_rep(p, v, c)
          __ixp4xx_ioread16_rep(p, v, c)
#define   ioread32_rep(p, v, c)
          __ixp4xx_ioread32_rep(p, v, c)
#define   iowrite8(v,p)
          __ixp4xx_iowrite8(v,p)
#define   iowrite16(v,p)
          __ixp4xx_iowrite16(v,p)
#define  iowrite32(v,p)
          __ixp4xx_iowrite32(v,p)
#define   iowrite8_rep(p, v, c)
          __ixp4xx_iowrite8_rep(p, v, c)
#define   iowrite16_rep(p, v, c)
          __ixp4xx_iowrite16_rep(p, v, c)
#define   iowrite32_rep(p, v, c)
          __ixp4xx_iowrite32_rep(p, v, c)
#define   ioport_map(port, nr)
          ((void __iomem*)(port + PIO_OFFSET))
#define   ioport_unmap(addr)
#define PIO_OFFSET      0x10000UL
#define PIO_MASK        0x0ffffUL
#define __is_io_address(p)
(((unsigned long)p >= PIO_OFFSET) && \
((unsigned long)p<=(PIO_MASK + PIO_OFFSET)))
static inline unsigned int
__ixp4xx_ioread8(const void __iomem *addr)
{
 unsigned long port = (unsigned long __force)addr;
 if (__is_io_address(port))
  return (unsigned int)__ixp4xx_inb(port &
PIO_MASK);
 else
 return (unsigned int)__ixp4xx_readb(addr);
}
..............................
static inline void
__ixp4xx_iowrite8(u8 value, void __iomem
*addr)
{
 unsigned long port = (unsigned long
__force)addr;
 if (__is_io_address(port))
  __ixp4xx_outb(value, port & PIO_MASK);
 else
 __ixp4xx_writeb(value, addr);
}
..............................
```

除此之外，有些PCI主端的晶片對於ARM中央處理器的記憶體存取方式，也不會直接自動轉換至PCI匯流排，和I/O存取一樣必須控制PCI主端晶片，進而產生一個PCI匯流排上的記憶體存取，其改寫內容十分類似I/O改寫，若不須要改寫，使用ARM中央處理器內定的程式碼則必須有以下的宣告：

```
#define __mem_pci(a)            (a)
```

若要改寫以下為其範例：

```
#define   writeb(v, p)    __ixp4xx_writeb(v, p)
#define   writew(v, p)    __ixp4xx_writew(v, p)
#define   writel(v, p)    __ixp4xx_writel(v, p)
#define   writesb(p, v, l) __ixp4xx_writesb(p, v, l)
#define   writesw(p, v, l) __ixp4xx_writesw(p, v, l)
#define   writesl(p, v, l)  __ixp4xx_writesl(p, v, l)
#define  readb(p)        __ixp4xx_readb(p)
#define  readw(p)        __ixp4xx_readw(p)
#define  readl(p)        __ixp4xx_readl(p)
#define  readsb(p, v, l)  __ixp4xx_readsb(p, v, l)
#define  readsw(p, v, l) __ixp4xx_readsw(p, v, l)
#define  readsl(p, v, l) __ixp4xx_readsl(p, v, l)
static inline void
__ixp4xx_writeb(u8 value, volatile void
__iomem *p)
{
  ..............................
  if (addr >= VMALLOC_START) {
    __raw_writeb(value, addr);
      return;
   }
  ..............................
}
..............................
/*
 * We can use the built-in functions b/c they
end up calling writeb/readb
 */
#define memset_io(c,v,l)
_memset_io((c),(v),(l))
#define memcpy_fromio(a,c,l)
_memcpy_fromio((a),(c),(l))
#define memcpy_toio(c,a,l)
_memcpy_toio((c),(a),(l))
```

## PCI轉記憶體為虛擬位址

最後仍一個問題須要特別注意，通常使用記憶體存取之前，可以將一個實體記憶體轉換為虛擬位址，若PCI主端是前面的方式，須要使用PCI主端來產生一個存取動作，那麼也可改寫ioremap() & iounmap()的副程式，其範例程式如下：

```
static inline void __iomem *
__ixp4xx_ioremap(unsigned long addr, size_t
size, unsigned int mtype)
{
 if((addr < PCIBIOS_MIN_MEM) || (addr >
0x4fffffff))
  return __arm_ioremap(addr, size, mtype);
  return (void __iomem *)addr;
}
static inline void
__ixp4xx_iounmap(void __iomem *addr)
{
 if ((__force u32)addr >= VMALLOC_START)
   __iounmap(addr);
}
#define __arch_ioremap(a, s, f)
__ixp4xx_ioremap(a, s, f)
#define __arch_iounmap(a)
__ixp4xx_iounmap(a)
```

到達此步驟，應該已經可以完成PCI主端程式的設定，而且可以讓PCI從端的程式具有相容性，且可使用於多種中央處理器之下，建立這種相容性的目的，主要是因為PCI從端的驅動程式可能由晶片廠商提供，但是只有在x86的平台測試過，為了可以正確地使用這些原始碼，了解以上述所提及的概念為當務之急。

**(本文作者為艾鍗學院教學中心特聘講師)**