國 立 交 通 大 學

電信工程研究所

碩 士 論 文

PCI上雙核心指令溝通管道的建立

Instruction Socket for Two CPUs on the PCI Bus

研 究 生：游雅嵐

指導教授：張文鐘 教授

中 華 民 國 九 十 九 年 七 月

PCI 上雙核心指令溝通管道的建立

Instruction Socket for Two CPUs on the PCI Bus

研 究 生：游雅嵐　　Student：Ya-Lan Yu

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學

電信工程研究所

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

July 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年七月

PCI上雙核心指令溝通管道的建立

學生：游雅嵐　　指導教授：張文鐘 博士

國立交通大學電信工程研究所碩士班

摘　　要

我們發展一套分散式的訊號處理系統，讓一個應用程式分散在兩個核心上同時進行，於是不同核心間就需要做訊息與指令的溝通，所以我們就建造了一個溝通管道讓應用程式可以使用它來做不同核心間的溝通。

我們讓影像處理與影像壓縮由一顆 CPU 負責執行，另一個 CPU 則負責做檔案系統與周邊裝置管理，因此做影像壓縮的 CPU 就需要利用負責檔案管理的 CPU 做影像讀寫，於是我們利用兩顆 CPU 間的共享記憶體(shared memory)做為訊息傳遞媒介，並搭配阻斷式(block)及中斷機制建立雙核心在 PCI 匯流排上的 CPU 間指令與資料溝通。

我們的溝通管道分為兩個層面，應用層面為一個雙向的 client-server 架構，核心層面為中斷處理機制，並利用 signaling 與 semaphore 作為兩層面之間的非同步啟動機制，核心層面的架構讓核心間的溝通由硬體中斷最後達到實質的訊息傳輸，訊息傳輸完才啟動應用層等待中的 server，開始針對接收到的訊息做服務。為了達到這些目標，我們便在作業系統中加入驅動程式來處理硬體中斷後各項資料搬移的動作，並建立一套功能函式讓應用程式使用，以便在兩顆 CPU 間進行指令溝通。

Instruction Socket for Two CPUs on the PCI Bus

Student：Ya-Lan Yu Advisor：Dr. Wen-Thong Chang

Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

ABSTRACT

We develop a distributed signal processing system where an application can distribute functional operations on two different CPUs to complete a service, so we have to build communication protocols to let the CPUs exchange their messages to each other.

One CPU is responsible for capturing the video and processing it, and the other CPU controls the file system and peripheral devices. In order to write to or read from the file system for the first CPU, shared memory, interrupt and blocking mechanism is used to establish the communication between the two CPUs on the PCI bus.

Our communication protocol has two aspects. The first one is application layer. We can see the communication channel as a client-server structure. The second one is the kernel or OS layer. It uses interrupt mechanism to notify the arrival of messages and use signaling and semaphore for the synchronization and notify the application to start receiving messages. So the OS layer completes the communication between two CPUs from hardware interrupt to real message transferring. After that, application can wake up the waiting server to receive the messages transferred by OS layer and processing the messages. For implementation, we design a linux driver to do the OS layer message transferring and we also build a user mode library as an interface of the application and the driver to do the communication.

誌謝

碩士兩年忙碌的生活終於要畫下終點，最感謝我的指導教授張文鐘教授，在我遇到困難或是學習遇到瓶頸，總是幫我尋找資源，也非常有耐心的協助我整合繁複的學問、指導、糾正我的思考與表達方式，雖然有時嚴厲但是我知道這是一種磨練，讓我學習到寶貴的知識與研究經驗！同時也感謝趙禧綠教授、高榮鴻教授及尤信程教授於口試時的指導，讓我能從另一種角度切入我的研究主題，更清楚分析與探討論文的每個環節。

另外要感謝的就是 NXP 半導體公司的陳爾泰學長還有 Chuck Peplinski，總是耐住性子教我板子的使用，讓我在茫茫懂懂之中跟著你們學習解決好多好多問題，讓我對嵌入式環境軟硬體都有更深入的認識，再來就是感謝工研院的工程師們，讓我有機會跟你們一起切磋，雖然每次報告都很緊張，可是你們總是鼓勵我們、也很願意跟我們一起討論，覺得跟你們一起學習是一件很開心的事！

然後要謝謝我的爸爸媽媽，讓我在經濟上、生活上沒有負擔，也在我最煎熬的時候陪伴我度過難關，更給予我自由的發展空間讓我可以在課業、愛情、社團都有良好的成績，謝謝你們在一旁的叮嚀跟鼓勵，未來我也會繼續充實自己的人生！

實驗室的大家庭當然不能忘記，有家豪學長的愛心電風扇還有用心的關懷讓我既能散熱又感覺溫暖，有信妤學妹的愛心電燈伴我度過七月挑燈夜戰的論文寫作生活，還有那個從慈青一直伴我到研究所的養魚的 twin，總是在我遇到困難的時候伸出援手，而且也是能分享心情的好朋友，再來就是漂亮的怡如，雖然說我們交集不多，不過真的還好有你陪著我一起闖蕩嵌入式環境，打敗 DSP!!!!還有細心聰明的喔拉拉跟耀駿，我再也不會忘記 FD 跟 FR 的差異，然後要感謝舒萍跟小不點，跟你們一起打球的生活真的很開心，也要感謝已經離開的于董，提供我許多驅動程式的資訊，讓我的研究生涯無往不利。最後謝謝 strike、我的男朋友以及其他的朋友們，謝謝你們曾經在我找工作、學業上跟生活上給予我的幫助，祝福你們也都有美好的未來！

最後，再次謝謝所有陪伴在我身旁的人，謝謝你們！

誌於 2010.夏 新竹ｏ交大

雅嵐

目錄

表目錄

圖目錄

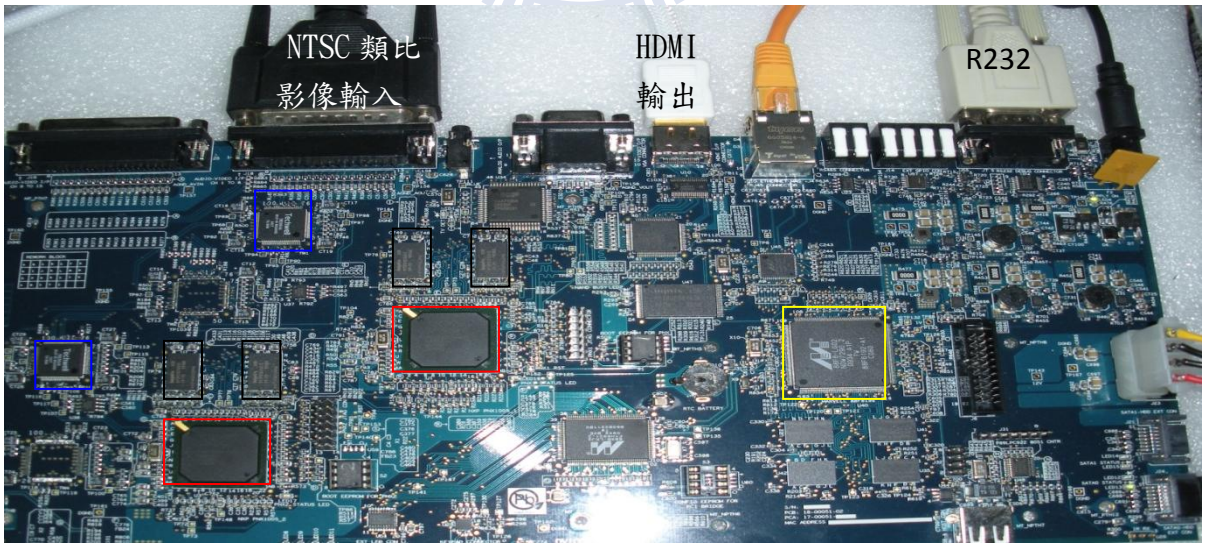
第一章 緒論

研究背景與動機

多核心是目前硬體環境的發展主軸，通常會有一個中控核心，使用具有龐大控制功能的作業系統來管理檔案系統與周邊裝置，周邊核心如果想要使用檔案系統，就必須與中控核心做溝通，由中控核心提供周邊核心檔案系統的操作。這樣的環境可以應用於手機或相機，將 DSP 端核心壓縮完的檔案交由另一個負責管理檔案系統的核心做檔案存取的動作。由於周邊核心如 DSP 核心，它所使用的作業系統通常是很簡單的即時作業系統，所以並不適合用來管理複雜的檔案系統，而且周邊核心主力是在提供強大的運算功能或影像處理功能，為了讓多個周邊核心可以使用到檔案系統，我們還是需要中控核心來負責檔案系統的管理，然後提供檔案系統服務給周邊的核心。

在這樣多核 CPU 進行平行處理(parallel processing)的嵌入式環境中，為了讓中控系統核心可以與周邊核心進行訊息交換以共同完成一個應用，兩個核心之間彼此如何溝通就是很重要的課題。

於是我們使用 PNX1005 Based 8 Channel DVR Demonstrator 嵌入式平台來作視訊監控的應用，環境中有兩顆 DSP 晶片(PNX1005)完全負責影像擷取、影像處理、影像輸出，而另一顆 ARM CPU 晶片則負責做 DSP 端影像的檔案系統管理、與外界做影像串流還有對其他 ARM 端硬體的控制。如下圖。



（圖 1.1）PNX1005 Based 8 Channel DVR Demonstrator 嵌入式平台概觀

說明:這個 DVR board 是以 NXP founded by Philips Semiconductors 的 PNX1005 Based 16 Channel DVR Demonstrator 為基礎，另外設計出的一塊 8 Channel DVR Demonstrator，主要是由 Marvell ARM(88F6192)和兩組負責做影像處理的 PNX1005 裝置所組成，目前平台上有兩個 PNX1005 裝置，每個 PNX1005 裝置

內部有一個 Media processor(Trimedia 3282 CPU)、還有兩個負責處理影像輸入的 SVIP 硬體及一個處理影像輸出的 QVCP 硬體，外部則有負責類比轉數位與多工的一個 TechWell 2864 晶片及 RAM。圖中紅色框起來的部分即為 PNX1005 晶片，藍框為 TechWell 2864 晶片，黑框為 PNX1005 的 DRAM，黃框則為 ARM。

每個 TechWell 2864 會接收 4 個通道的影像訊號傳入其中一個 SVIP，所以一個 PNX1005 最多傳入 4 個通道的影像。4 個影像輸入會先由 TechWell 2864 將他們轉成 digital 訊號後，再多工成一條通道，之後傳到 PNX1005 的 SVIP 硬體去解多工後才繼續進行影像處理。影像經過處理後可以直接由 PNX1005 裝置使用 HDMI 呈現到螢幕上，或是利用 H.264 等壓縮技術將影像壓縮後存到系統硬碟裡，有需要的話再從硬碟裡將資料讀出解壓縮播放。

研究方法與目標

利用這個視訊監控開發嵌入式平台，我們使用作業系統為 linux 的 ARM CPU 來做 linux 檔案系統管理、作業系統為 pSOS 的 DSP CPU 來負責做影像擷取與影像壓縮，於是一個視訊監控應用便可以分散到兩個負責不同任務的 CPU 去運作，但是處理不同任務的 CPU 為了完成同一個應用，兩顆 CPU 間勢必需要做訊息溝通與資料交換。

為了讓 DSP 端的影像可以傳到 ARM 端記憶體做進一步處理或將影像存到檔案系統中，DSP 端需要傳送指令及影像資料(影像位址與大小)給 ARM，於是我們便在兩顆 CPU 運作的應用層建立雙向 client-server 架構， DSP 端一開始是 client，它只能請求 ARM 端有提供的指令服務讓 ARM 端來幫助它完成檔案系統操作，而 ARM 就是 server，必須在接收到 DSP 端傳送訊息過來後，提供 DSP 端指令操作的服務，當 DSP 端傳送請求給 ARM 後，那個傳送請求的程序就好像是 DSP 端的 server 執行緒，等待著 ARM 端回覆訊息，之後 ARM 會直接指定要啟動 DSP 端哪一個在等待的 server 來讀取回覆訊息。

為了瞭解兩端指令傳輸的溝通，我們於是需要了解應用層下方是如何使用共享記憶體來傳送訊息，而且因為每個應用中可能使用到不同的 service，例如處理指令的 service、傳送數字訊息的 service、要等待對方創建好資源後自己才能去使用那個資源所需要的事件觸發 service，所以我們還要了解如何在只有一個共享記憶體與中斷線的情況下，找到正確的應用服務 service 來處理收到的指令，並了解應用層要如何將操作共享記憶體的動作設計於驅動程式與功能函式，讓應用層可以使用功能化介面來做底層的訊息交換動作。

最後我們希望在 ARM 端應用程式可以提供記憶體複製的功能給 DSP 端的應用程式，所以藉由 DSP 端傳送給 ARM 端的指令中，所包含的影像位址與大小，我們在 ARM 端應用程式又會再增加一個 FaceRecog 模組藉由取得的影像位址與大小對 DSP 端的實體記憶體作記憶體複製動作，將 DSP 端人臉偵測應用抓取到的人臉，可以直接複製到 ARM 端虛擬記憶體空間，讓 ARM 端應用程式可以在應用程序環境中對抓的人臉做後續的人臉辨識與其它應用。

論文組織

簡介	第一章	緒論	內容說明
背景知識	第二章	Linux 驅動程式設計環境	1. physical memory 與 virtual memory 2. Linux 環境中各種位址 Physical address、Bus address、virtual address 之間的轉換 3. memory mapping 4. 控制硬體的方式—I/O port、MMIO 5. PCI 裝置 6. PNX1005 的 memory apertures 7. 如何讀取 memory apertures 資訊 8. 程序並行控制 9. 中斷機制
研究主題	第三章	控制 PNX1005 的 Linux 驅動程式	1. Linux 驅動程式功能與操作介面的設計架構 2. 驅動程式的啟動 3. 利用 linux 驅動程式設計架構來實作 TMMan 驅動程式功能 4. 設計 user mode library 作為驅動程式與應用程式的界面
	第四章	ARM 與 DSP 端底層溝通機制與架構	詳細說明分散式處理應用的 CPU 如何藉由底層的共享記憶體、中斷機制傳送訊息，再搭配 client-server 的架構完成溝通
	第五章	指令溝通— C Runtime Service	說明 DSP 如何利用第四章的溝通架構來傳遞指令，並以 ARM 為 server，由負責檔案管理的 ARM 提供的 CRT 應用服務完成檔案輸出入
	第六章	實驗— DSP 端人臉擷取指令設計	在 ARM 端以 CRT 服務為基礎，設計一個可以接收 DSP 端偵測出的人臉位

			址，將人臉從 DSP 端 RAM 複製到 ARM 端記憶體的指令設計
	第七章	結論	
附錄 Appendix	A	PNX1005 暫存器	
	B	DSP 端各種指令對應的 ID	
	C	於 linux 環境開發主機中編譯 tmman 驅動程式、ARM 端應用程式的環境設定	
	D	TMMan 驅動程式辨別多個相同類型的 PNX1005 的機制	
	E	user library 初始化所需的廣域結構	
	F	函式庫應用實例:tmload 中下載 DSP 端可執行檔的運作	
	G	copy_from_user/copy_to_user	
	H	使用 termination_handler 重新請求 ARM 端服務(讀檔)	

第二章 Linux 驅動程式設計環境

DVR 平台上的 ARM CPU 與 DSP 端 PNX1005 裝置是透過 PCI 匯流排來進行資料的傳遞，其中 ARM 為一個中控系統(host)，負責處理與 DSP 端還有與開發主機之間的溝通，對 ARM 來說，PNX1005 就是一個 PCI 裝置，我們便在 ARM 端設計一個可以控制 PNX1005 的驅動程式。

因為 ARM 端的 OS 是 linux，所以我們就以 linux 驅動程式的設計架構與環境來開發這個驅動程式，由於驅動程式必須以軟體來控制硬體，期間會牽涉很多 physical memory 與 virtual memory 的議題，所以 2-1 節會介紹 linux 對 physical memory 與 virtual memory 的管理與對應，讓讀者知道兩者間的映射關係，做為第三章設計 linux 驅動程式中設計 mmap handler 的重要背景知識，2-2 節則會介紹負責將 physical memory 映射到 virtual memory 的映射種類與負責的函式，而 2-3 節另會探討 physical memory 與 virtual memory 位址轉換的關係，於第四章提到的 shared memory 劃分提供位址轉換觀念，了解記憶體映射與位址轉換觀念後，2-4 節 MMIO 控制硬體的機制其實就會透過虛擬記憶體使用到實體記憶體，然後由實體記憶體的設定對裝置上暫存器作控制。

接下來 2-4、2-5 提供第四章兩端 CPU 使用共享記憶體進行溝通時，除了驅動程式對記憶體基本的操作外，還需要使用到的操作機制。2-5 節會介紹設計 linux 驅動程式的中斷機制，以硬體中斷來通知對方接收訊息，而且由於溝通時使用的共享記憶體，程式設計者在軟體上會使用許多結構來操作這塊硬體，為了避免兩個 process 同時去修改共享記憶體中的資料，所以 2-6 節會介紹一般 linux 環境下如何在程序中保護 critical section，還有中斷機制在 linux 環境中存在的意義與使用方式。

最後，由於 ARM 與 PNX1005 是以 PCI 匯流排連接，所以對 ARM 來說 PNX1005 晶片就是一個 PCI 裝置，為了控制 PCI 裝置，2-7 節會介紹一般 linux 下 PCI 裝置會提供哪些重要的記憶體位址資訊與硬體資訊給驅動程式，而驅動程式又是用什麼樣的方式來獲取 PCI 裝置所給的資訊，讓驅動程式可以從這些資訊去操作它。2-8、2-9 節則會介紹 PNX1005 這個 PCI 裝置究竟提供了什麼樣的資訊給 ARM 的驅動程式，ARM 的驅動程式又使用了 2-7 節哪些函式來讀取這些資訊。

2-1 physical memory 與 virtual memory

一般程式設計者都只在 virtual memory 中進行設計，但由於驅動程式不只是寫軟體，還需要控制硬體，因此在必要的時候驅動程式必須負責作 phsical memory 映射 virtual

memory 的動作，這在一般程式設計是很少見的，所以本節會介紹基本的 phsical memory 與 virtual memory 存在的意義，還有 linux 對記憶體的管理與兩種記憶體之間的映射關係。

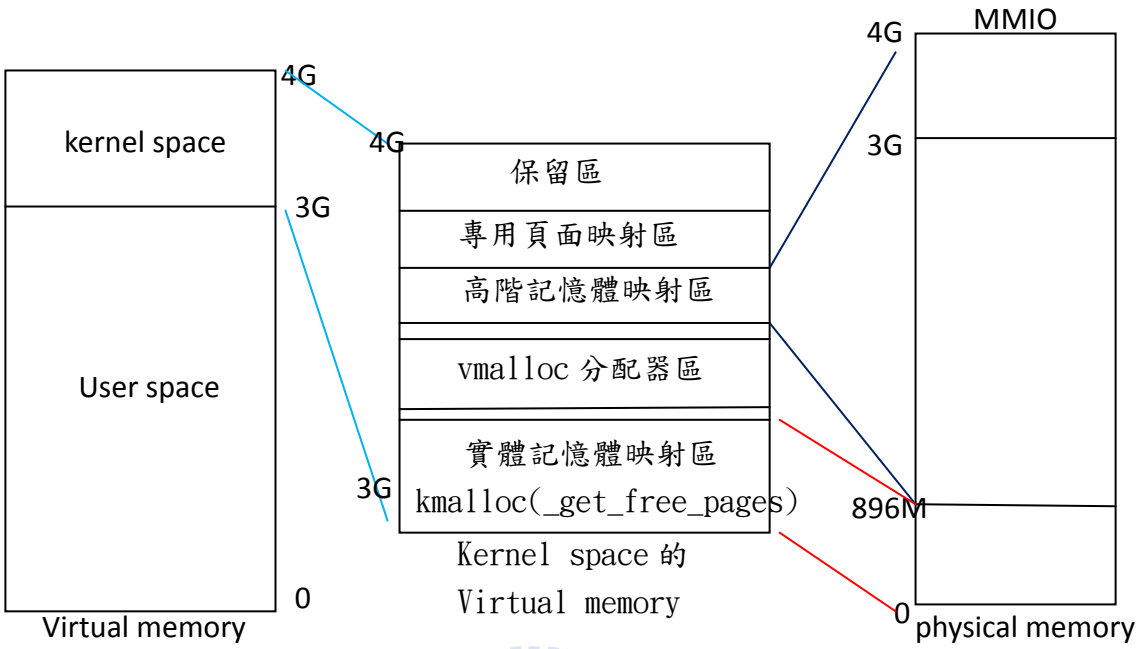
由於 CPU 運算所需資料或程式碼都必須先載入到實體記憶體後才能由 CPU 取用，但是為了讓使用者感覺上有很大的記憶體空間可以用來設計程式，而且讓 physical memory 可以被有效利用，於是使用者便在 virtual memory 環境設計程式，真正運作的時候系統才會把需要運作的部分載入到 physical memory 上執行。雖然在運作的時候系統會自動載入需要運行的部分到 physical memory 上，但是在設計驅動程式時，因為控制硬體的方式可能會希望藉由實體記憶體的操作就可以反應到裝置上(MMIO 的方式)，或是說希望在 user space 就可以藉由特定的 physical 與 virtual memory 的映射關係(mmap)將檔案或實體記憶體空間映射到 virtual memory 後，直接在使用者空間做檔案複製的動作，所以我們還必須知道 linux **如何管理 physical memory 與 virtual memory、它們之間的 memory mapping**，以及環境中**各種記憶體位址的轉換關係**，才能在設計驅動程式時做好 virtual memory 與 physical memory 的對應。

在使用 MMU 的 CPU 架構下，系統設計者可以利用的虛擬記憶體約 4GB。因為一般 user 或驅動程式開發者在開發程式的時候，都是在虛擬記憶體裡運作的，為了讓 user 不能隨便更改重要的硬體設定與系統運作機制，以預防嚴重的系統錯誤或是惡意傷害系統的行為，又把**虛擬記憶體空間分為 user space 與 kernel space**。在 kernel space 中運作的程式，可以讓 kernel 來掌管重要的系統運作行為，所以驅動程式開發者便在 kernel space 設計驅動程式；如果使用者想要使用硬體或利用系統中的重要資源時，則必須使用系統呼叫請求 kernel 的許可與協助，因此如果 user 想要驅動程式做控制硬體的動作，就需要一個與驅動程式連接的系統呼叫來請求 kernel 中的驅動程式協助，因此我們還必須要注意資料如何在 user space 與 kernel space 做複製或交換(3-1-3 的 ioctl handler)。

通常，每個 user process 可以使用的虛擬記憶體空間為 0~3GB，我們把這段空間稱為 user space；由於以前只能在實體記憶體內運作的 kernel 程式現在也可以利用虛擬記憶體來運作了，所以我們把虛擬記憶體的 3G~4G 分配給 kernel 程式，這段空間就像是每個 process 都會擁有的相同區段，我們稱為 kernel space，因此，就算實體記憶體有 4GB，CPU 也看得到 4GB 的物理記憶體空間，但 user process 永遠看得到的使用空間就是 3GB。

另外，虛擬記憶體空間裡 1G 的 kernel space 又可分成實體記憶體映射區、vmalloc 分配器區、高階記憶體映射區、專用頁面映射區及保留區(圖 2.1)，一般情況下，實體記憶體映射區最大長度為 896MB，此區以 1:1 linear mapping 的方式依序將系統的實體記憶體映射到虛擬記憶體的 kernel space 去，而未超過實體記憶體映射區所對應的實體記憶

體也另外稱為常規記憶體；當系統實體記憶體大於 896M 時，超過此界限以上的實體記憶體稱為高階記憶體，會被映射到虛擬記憶體 kernel space 的高階記憶體映射區。



(圖 2.1)虛擬記憶體與實體記憶體的映射關係以及 kernel space 的管理狀況

(表 2.1)詳細說明了 linux kernel 2.6.22.18 如何劃分 kernel space 的每個區塊，從 3G 開始到最大 3G+896M(high_memory)的範圍為實體記憶體映射區，通常 high_memory 以上的空間就是高階記憶體映射區，只是還可以藉由 VMALLOC_START 與 VMALLOC_END-1 位址空間定義出 vmalloc 分配器區，另外還有其它的專用頁面或保留區供 OS 或未來開發使用。通常驅動程式運作時作 kmalloc 動作分配出的虛擬記憶體空間以及變數，會位於實體記憶體映射區(PAGE_OFFSET~high_memory-1)，當 kernel 使用 vmalloc 配置連續的虛擬記憶體空間(但實體記憶體不一定連續)或是利用 ioremap 將實體記憶體空間映射到虛擬記憶體的 kernel space 時，這塊 kernel space 虛擬空間區段會位於 vmalloc 分配器區內(VMALLOC_START~VMALLOC_END-1)；如果像我們使用的平台，為了在 ARM 端創建一塊與 DSP 端溝通的 shared memory 時，使用到 dma_alloc_coherent 函式在實體記憶體中分配連續且 nocache 的記憶體時，則會同時將這塊實體記憶體映射到 kernel space 的 DMA memory mapping region；如果使用者呼叫 mmap 這個系統呼叫，則會將實體記憶體映射到虛擬記憶體的 user space 去。因此，未來我們設計的驅動程式，大部分會運作在 kernel space 的實體記憶體映射區、vmalloc 分配器區與高階記憶體映射區。

(表 2.1)kernel 2.6.22.18 在 virtual memory 的分配狀況:

Start	End	Use
ffff8000	ffffffff	**copy_user_page / clear_user_page use.** For SA11xx and Xscale, this is used to setup a minicache mapping.
ffff1000	ffff7fff	**Reserved.** Platforms must not use this address range
ffff0000	ffff0fff	**CPU vector page.** The CPU vectors are mapped here if the CPU supports vector relocation (control register V bit.)
ffc00000	fffeffff	**DMA memory mapping region.** Memory returned by the dma_alloc_xxx functions will be dynamically mapped here.
ff000000	ffbfffff	Reserved for future expansion of DMA mapping region.
VMALLOC_END	feffffff	Free for platform use, recommended. VMALLOC_END must be aligned to a 2MB boundary.
VMALLOC_START	VMALLOC_END-1	**vmalloc() / ioremap() space.** Memory returned by vmalloc/ioremap will be dynamically placed in this region. VMALLOC_START may be based upon the value of the high_memory variable.
隔離帶		
PAGE_OFFSET (3G)	high_memory-1 (3G+ 896M)	**Kernel direct-mapped RAM region.** This maps the platforms RAM, and typically maps all platform RAM in a 1:1 relationship.
TASK_SIZE	PAGE_OFFSET-1	**Kernel module space** Kernel modules inserted via insmod are placed here using dynamic mappings.
00001000	TASK_SIZE-1	**User space mappings**

高階記憶體映射區&專用頁面區&保留區

隔離帶

Vmalloc 分配器區

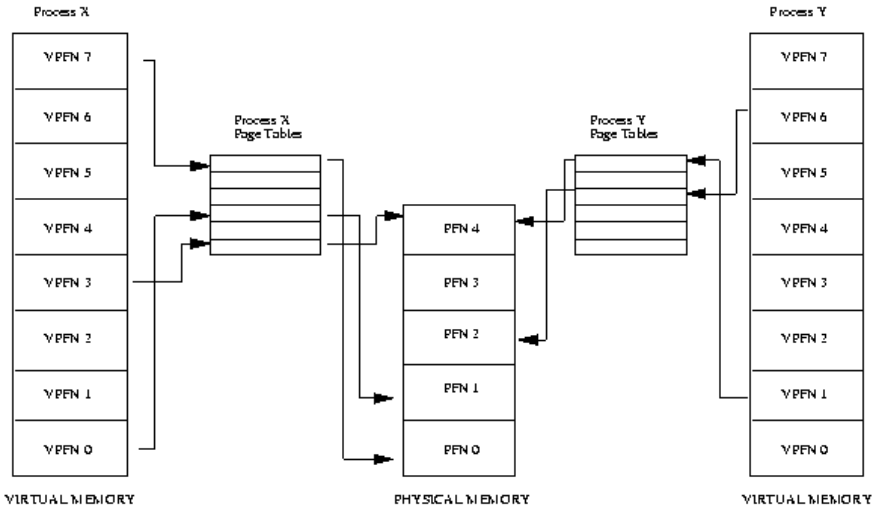
實體記憶體映射區

		Per-thread mappings are placed here viathe **mmap()**	
00000000	00000fff	CPU vector page / null pointer trap CPUs which do not support vector remapping place their vector page here. NULL pointer dereferences by both the kernel and user space are also caught via this mapping.	user space

（資料來源:linux kernel 2.6.22.18 的 document）

從虛擬記憶體對應到實體記憶體

為了讓實體記憶體的使用更有效率，避免因 contiguous memory 配置記憶體的方式造成的記憶體浪費，linux 因此基於 paging 系統來做實體記憶體管理。當程式開始運作的時候，CPU 必須從實體記憶體中讀取 instruction 並 decode 它，在解這個 instruction 的時候可能還需要到實體記憶體中去 fetch 其他 instruction 或是將資料從實體記憶體 load 到 CPU 去，然而這些資料或是 instruction 的位址一開始可能都在 virtual system 裡面，所以會以 virtual address 表示資料或 instruction 所在位址。因此，CPU 必須將 virtual address 轉換成 physical address 才能從實體記憶體中讀寫資料或指令出來。為了建立 physical memory 與 virtual memory 之間的關係，讓 CPU 可以透過產生的 virtual memory 來間接找到對應的 physical memory，我們將 virtual memory 與 physical memory 切成大小相同的 page，當該 virtual memory 的 page 有需要使用到時才將之載入到系統記憶體中對應的 physical page，而 virtual memory 與 physical memory 之間的對應則由 page table 來表示。基本的 memory mapping 模型如(圖 2.2)。



（圖 2.2）memory mapping

（資料來源: The Linux Kernel，http://tldp.org/LDP/tlk/tlk-toc.html）

當 CPU 產生一個 virtual address，它會把這個 virtual address 切成兩部分，前幾個 bit 用來當作 virtual page frame number(VPFN)，剩下的 bit 則表示資料在這個 page number 所處的空間裡的 offset。以 Process Y 來看，假設 virtual address 是 0x2194，而 page size 為 8KB(0x2000)(一般的 x86 系統為 4K)，則表示該 virtual address 位於 VPFN1 的 page 中。接著用 VPFN1 查詢 page table 找到對應的 physical memory 的 PFN4，最後以 PFN4 的基底位址加上 offset 0x194 就是 physical address 了。另外，linux 其實已提供了一些函式用來做 virtual address 與 physical address 之間的轉換，詳情請看 2-2。

由於每個 user process 在虛擬記憶體中使用到的 **user space** 都是完全獨立的，因此 user process 各自有不同的 page table 對應到 physical memory，即使任一個 user process 發生錯誤，也不會影響到系統整體的運作，而且彼此共同參考到的函式庫或是檔案都可以透過虛擬記憶體對應到同一塊實體記憶體，只有當程序需要對這塊共同的實體記憶體內容做修改時，才會藉由 Copy-On-Write 的機制，使用另一塊實體記憶體空間來紀錄改變的資料內容，在(圖 2.2)中，我們可以看到 physical memory 的 PFN4 同時被兩個 processe 所共用。而 **kernel space** 是由 kernel 負責映射，它並不會跟著 user process 而改變，是固定的，因此 kernel space 會有自己對應的 page table。

從實體記憶體對應到虛擬記憶體

一般來說，程式執行時都會直接在虛擬記憶體作資料的處理，然後系統會自動幫我們把虛擬記憶體對應到實體記憶體去，但有時我們需要把外部裝置(例如 PNX1005)的記憶體空間與 I/O 空間或是檔案系統裡的檔案映射到 ARM 端的 user space，讓 ARM 端應用程式可以在 user space 操作這些空間或檔案，此時應用程式就需要利用 mmap 系統呼叫來將實體記憶體映射到虛擬記憶體中，而兩種不同的記憶體間，kernel 則藉由 page table 做為兩者之間的對應關係。

Kernel 要能建立 page table 前，除了要知道欲映射的實體記憶體，還需要先為實體記憶體分配一塊虛擬記憶體空間，所以 linux kernel 必須管理 virtual memory，當開發者要求將實體記憶體映射到虛擬記憶體時，系統才能從自己管理的 virtual memory 中找到空位，將實體記憶體映射進來，而 linux 在管理時，則會透過資料結構來描述與記錄 virtual memory 的使用狀況。

在 linux 系統裡**每個 process** 會以 task_struct(include/linux/sched.h) 這個資料結構存在，而這個資料結構中用來描述該 process 所使用的 virtual memory 則以 mm_struct 這個結構來表示，然後因為每個程式在 virtual memory 可以被分成很多區段，在需要的時候才載入到實體記憶體，因此每個 virtual memory 區段我們稱為 virtual memory area，

在系統裡以 vm_area_struct 表示，mm_struct 裡也會有一個指標指向第一個 vm_area_struct 結構，而這些 virtual memory areas 則會繼續以 link list 的樹狀結構連接於系統中，方便系統在發生 page fault 的時候，能快速找到要求 demand paging 的 vma，然後將該 vma 使用到的 physical page 載入到實體記憶體中。以下為 vm_area_struct (include/linux/ mm.h)的結構內容(表 2.2):

(表 2.2)vm_area_struct 結構內容

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        struct vm_area_struct *vm_next;
        pgprot_t vm_page_prot;
        unsigned long vm_flags;
        rb_node_t vm_rb;
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;     };
```

每個 vm_area_struct 裡頭會以 vm_start 與 vm_end 來表示該 VMA 所佔的 virtual memory 區段，並以 vm_page_prot 來儲存該虛擬記憶體區段的使用權限，另外也用 vm_operations_struct 結構來存放一組可能需要對這個區段所做的操作，像其中儲存的 nopage 函式可以在系統發生 page fault 時(在 page table 中找不到該 virtual address 對應到的 physical memory)，可以將對應到的 physical page 下載到實體記憶體中，並將這個 physical page 的位址填到 page table 裡，表示現在已有哪些 virtual page 內容存在於實體記憶體上。

因此，每當 kernel 要為實體記憶體分配一塊虛擬記憶體空間，就會先查找該使用者程序中還未使用的虛擬記憶體空間分配給它，然後創建一個 vm_area_struct 來描述這塊空間，有了這個 VMA 結構，我們就可以利用其中的 vm_start 與 vm_end 虛擬記憶體位址，再使用 kernel 提供的 remap_pfn_range 函式，為 VMA 結構描述的虛擬記憶體區塊與實體記憶體區塊建立頁表，這個觀念將會運用於 3-1-3 節的 mmap handler 實作。

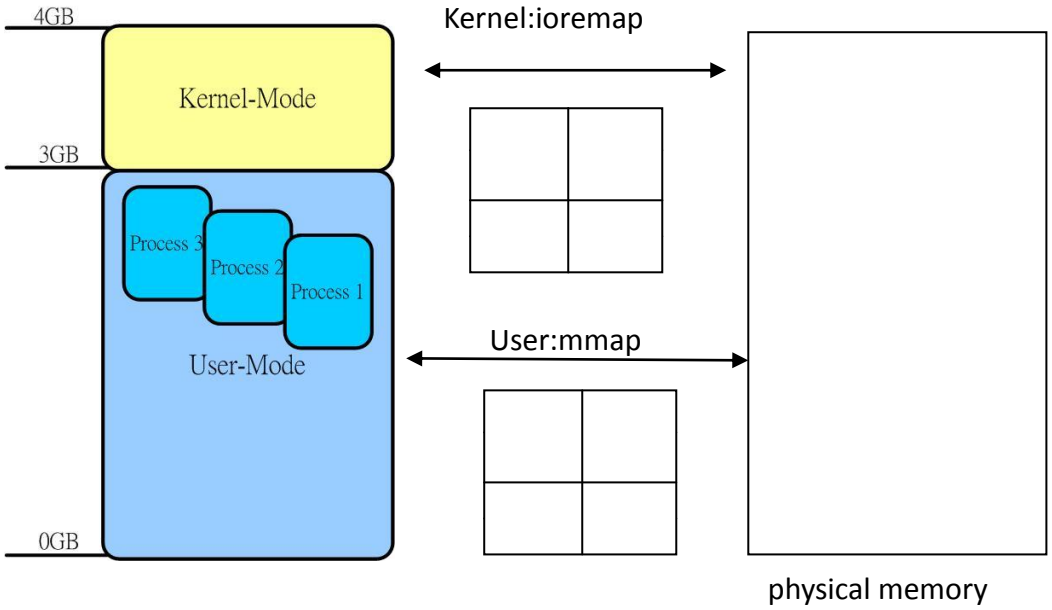
page table 基本上會儲存在實體記憶體裡，但是每次 virtual address 要轉換成

physical address 都需要經過兩次讀取 memory 的動作，一次是透過 VPFN 找到 page table 裡代表它的 entry 並得到 PFN，然後再一次藉由這個 PFN 來讀取其對應的 physical memory 資料。為了加快速度，CPU 裡使用了 MMU，其運作其實就是為這些存在在實體記憶體裡的 page table 建立一個 cache，它暫存了少量虛擬位址與實體位址的轉換關係，我們稱為 TLB(translation look-aside buffer)。每次 CPU 要開始對 virtual address 做轉換時，MMU 會先去查詢 TLB 中的對應表，如果有查到就直接存取 physical address，沒有的話則透過 TTW(translation table walk)從記憶體中找到對應的 page table entry，然後採用 LRU(least recently used)更新 TLB 或是直接填入 TLB 中空白的位置。MMU 除了具有虛擬記憶體位址與實體記憶體位址轉換的功能，也可以在 TLB 多加一個欄位用來記錄該虛擬記憶體現在是哪一個 process 使用的，讓 TLB 可以同時存取多個 process 的虛擬位址對應而不用每次換另一個 process 來查找時，就需要刷新 TLB，另外也提供了記憶體存取的保護權限，讓每個 user process 可以配置獨立的記憶體空間並保證使用者空間不能存取核心空間的位址。

2-2 記憶體映射(memory mapping)動作的種類

2-1 已談過 physical memory 與 virtual memory 存在著映射關係，而且由於 PNX1005 這個 PCI 裝置會把它的暫存器與程式運作時使用的 RAM 這兩個記憶體空間，在開機時映射到 ARM 端系統實體記憶體的 MMIO space，讓 ARM 可以藉由 MMIO 的方式讀寫自己的實體記憶體來控制 PNX1005，我們於是把**PNX1005 開放出來給 ARM 端控制的記憶體區塊稱為 memory apertures**(詳情請看 2-8)。因為不管是上層的應用程式還是位於 kernel 層的 TMMan 驅動程式都是在 virtual memory 下設計的，但 PNX1005 開出來的 memory apertures 目前則映射於 physical memory 中，所以我們必須了解如何將實體記憶體映射到虛擬記憶體中，讓應用程式以及驅動程式可以透過 virtual address 將該記憶體的讀寫動作轉成對 PNX1005 暫存器或記憶體的操作。本節將針對 TMMan 驅動程式使用到的記憶體映射動作做介紹。

通常，在 kernel 中會以 ioremap 的函式來建立 page table，將 physical memory 空間映射到虛擬記憶體空間裡的 kernel space 去，而 user 則必須使用 mmap 這個系統呼叫，請求 kernel 的協助，建立虛擬記憶體裡的 user space 與 physical memory 之間的 page table，而要使用 mmap 機制，除了呼叫 mmap 系統呼叫外，我們還必須在 kernel 中的 driver 實作 mmap handler(3-1-3 節)才能完成虛擬記憶體空間的 user space 與 physical memory 的對應。



（圖 2.3）mmap 與 ioremap 在 physcial memory 與 virtual memory 中的角色

2-2-1 虛擬記憶體的 kernel space 與 physical memory 間的映射--ioremap

為了藉由實體記憶體的讀寫來啟動 PNX1005 或是對 PNX1005 發出中斷等，我們需要設定 PNX1005 的暫存器，這些暫存器在 PNX1005 中被規畫到 MMIO apertures 中並在開機時映射到 ARM 端實體記憶體的 MMIO 區域，但是因為 TMMan 驅動程式位於 kernel 中，因此我們必須把 PNX1005 的 MMIO apertures 映射到虛擬記憶體的 kernel space 讓驅動程式來控制。我們使用到的函式為 kernel 提供的 ioremap_nocache，會傳入欲映射到 kernel space 的實體記憶體起始位址與欲映射的大小，將裝置所指定的實體記憶體起始位址以上的某個區段映射到 kernel space 的 vmalloc 分配器區(表 2.1)，並回傳給驅動程式一個 kernel space 的 virtual address。

```
void *ioremap_nocache(unsigned long offset,unsigned long size)
```

註:原始的 ioremap 函式即具有映射的功能，加上 nocache 是為了讓這段被映射到 physical memory 不會因為 cache 的影響而導致 cache 與記憶體不一致的現象。

2-2-2 虛擬記憶體的 user space 與 physical memory 間的映射—mmap

一般情況下，虛擬記憶體的 user space 是不可能也不應該直接存取裝置的，但是 mmap 這個系統呼叫(system call)讓使用者在 user space 就可存取裝置的實體位址。一般檔案系統中，這個 mmap 機制已經被實作於檔案系統裡，並不需要驅動程式另外去實作它，當使用者把檔案系統中的檔案利用 open 先載入到實體記憶體後，便可使用 mmap 這個系統呼叫搭配檔案系統裡 mmap handler 的實作將檔案從實體記憶體映射到虛擬記憶體的 user space 中。

對於 TMMan 驅動程式來說，因為在開機時已經把 PNX1005 程式運作使用到的 RAM 與暫存器空間映射到 ARM 端實體記憶體 MMIO 空間，如果我們想要在 user space 的應用程式直接將 ARM 端檔案系統裡的檔案複製到 PNX1005 的 RAM 去，我們必須把檔案與 PNX1005 程式運作使用到的 RAM apertures 都映射到虛擬記憶體的 user space 中(有關檔案的映射已在上一段說明過)，而 PNX1005 的 RAM 映射則必須透過 TMMan 驅動程式裡的 mmap handler 實作，將虛擬記憶體中 user space 的一段 VMA 與 PNX1005 的記憶體建立映射關係，如此一來，當使用者修改 user space 這段空間的內容時，才會轉成對該裝置記憶體空間的設定，不再需要將資料從虛擬記憶體的 user space 複製到 kernel space 後，再由 kernel 的驅動程式來做檔案的讀寫，可以簡化資料的傳遞過程。

綜合使用檔案系統與 TMMan driver 的 mmap，PNX1005 程式運作的記憶體空間，這個在開機時也已經被映射到 ARM 端實體記憶體 MMIO 區段的 RAM aperture，我們會使用 TMMan 驅動程式的 mmap 將這段已映射到 MMIO 區段的 DSP 端程式運作空間，映射到 ARM 端目前程式運作的虛擬記憶體 user space 去，然後 ARM 端可以從檔案系統中將 DSP 端可執行檔載入到實體記憶體後，使用檔案系統的 mmap 機制將這個檔案內容，同樣映射到 user space 去，如此一來，我們就可以在 user space 透過簡單的 memcpy 函式將 ARM 端記憶體中的檔案複製到 DSP 端記憶體，完成 DSP 端可執行檔的下載。

2-3 Kernel 提供用來轉換空間位址的函式

第四章我們在探討共享記憶體區塊的劃分時，會藉由 kernel 提供的 dma_alloc_coherent 函式，在 ARM 端 RAM 上配置出一塊空間來使用，由於共享記憶體的區塊其實是從軟體上利用不同的結構來劃分的，所以我們會先從這個函式得到 kernel space 的虛擬記憶體位址，依照不同結構大小一一求得每個結構控制的共享記憶體的起始位址，然後再利用空間轉換關係從虛擬記憶體位址求得實體記憶體位址，因此本節就介紹一下 linux 環境中，虛擬記憶體與實體記憶體間的轉換關係。

在支援 MMU 的 CPU 架構下，系統可利用 virtual memory 與 physical memory 的轉換，讓使用者利用看似很大的虛體記憶體空間來運作程式，只把真正需要運作的部分放到 physical memory，讓 physical memory 可以被有效利用，因此在程式設計時，會遇到的把記憶體位址種類分為 virtual、physical，另外對於外部裝置而言，還使用了 bus address 來表示外部裝置眼中的記憶體位址。(表 2.3)為三種位址所代表的意思。

(表 2.3)三種位址表示

位址	意義
Virtual	CPU 對 physical address 轉換過後所得到的位址，也是一般使用者(user)所看到的位址。
Physical	未經 CPU 轉換過後的位址，也是 CPU 告訴位址匯流排的值。
Bus	bus address 是每個外部裝置看記憶體看到的位址，而不是 CPU 看記憶體時看到的位址，而且每個外部裝置看到記憶體的位址可能都不同，但為了減少硬體設計的複雜度，所以都會設計讓外部裝置看到的記憶體位址是一樣的。另外在一般 PC、還有使用 PCI 匯流排、ISA 匯流排的情況中，也會設計讓 CPU 看記憶體使用到的 physical address 與裝置看記憶體用到的 bus address 的值是一樣的。

1.僅適用於虛擬記憶體中 kernel space 的實體記憶體映射區與系統實體記憶體的常規記憶體間**位址的轉換**，不適用於高階記憶體映射區:

```
#include <asm/io.h>
    phys_addr = virt_to_phys(virt_addr);
    virt_addr = phys_to_virt(phys_addr);
     bus_addr = virt_to_bus(virt_addr);
    virt_addr = bus_to_virt(bus_addr);
```

原理:

- vir←→phy

  因為 kernel space 的實體記憶體映射區是以 1:1 linear mapping 的方式由 physical memory 映射到 kernel space 去的，所以我們可以想成虛擬記憶體的位址 3G 其實就代表了實體記憶體的位址 0，如果使用 vir_to_phy，則只要將虛擬記憶體位址減掉 3G 即等於實體記憶體位址；phy_to_vir 就是將實體記憶體位址加上 3G 即為虛擬記憶體位址。

- vir←→bus

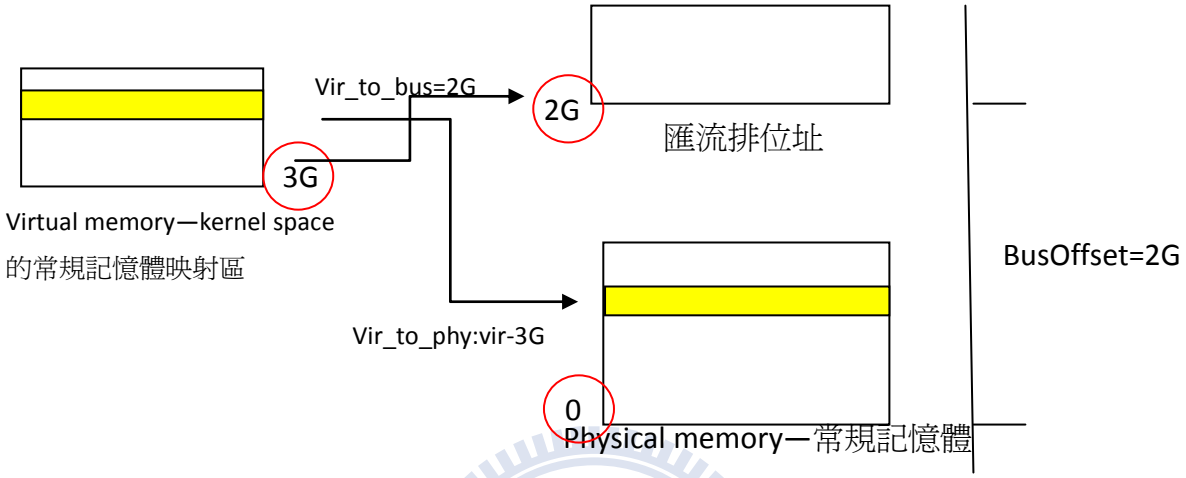
  有些硬體(尤其是以 DMA 為基礎的硬體)使用的位址是匯流排位址而非實體記憶體位址，所以我們會需要知道匯流排位址。雖然對 PC 上 ISA 和 PCI 而言，匯流排位址即為實體位址，但並非每個平台皆如此，有的匯流排經過電路的橋接以後會把 I/O 位址映射到不同的實體位址，例如:以使用 32-bit bus(板子上總共有 4G 的位址可以使用)的 PowerPC Reference Platform 來說，它的位址分配如下，而且因為虛擬記憶體 kernel space 的常規記憶體映射區與小於 896M 的實體記憶體有 1:1 的 linear mapping 關係，因此在虛擬記憶體 kernel space 的起始位址 3G 將會對應到實體記憶體的位址 0，我

們也可以再利用 vir_to_bus(3G)得到匯流排位址 2G，表示 CPU 想要讀寫的實體記憶體位址 0 的資料，對於其他裝置來說，匯流排位址 2G 也會對應到 CPU 看到的實體位址 0 的資料。

0-2 GB "real memory"

2 GB-3 GB "system IO" (in/out and similar accesses on x86)

3 GB-4 GB "IO memory" (共享記憶體 over the IO bus)



(圖 2.4)各種位址起始位址的對應

情況 1:已知 physical address 要求得 bus address

- 利用 vir_to_bus(3G)得到 BusOffset:2G
- 因為 physical address 與 bus address 存在位移的關係(差 2G)，因此 bus address=physical address+BusOffset

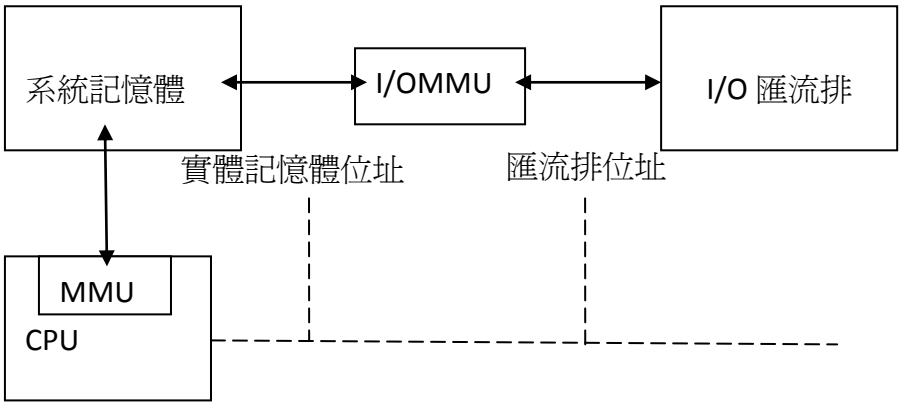
情況 2:已知 bus address，要求 physical address

- 利用 vir_to_bus(3G)得到 BusOffset:2G
- physical address=bus address-BusOffset

=>因為 DVR 平台的 bus address 與 physical address 相同，所以 vir_to_bus(3G)=0

2. vir_to_bus 僅適用於簡單的虛擬位址與匯流排位址的轉換，只要實體記憶體位址與匯流排位址之間存在著單純的 linear mapping 的話均可使用。我們可以利用 vir_to_bus 來做上述這種基本的起始位址關係轉換求出 BusOffset，然後利用已知的 physical address/bus address 藉由 BusOffset 來算出 bus address/physical address，但是這種關係在有 IOMMU 的系統裡是不適用的。

3. IOMMU 與 CPU 的 MMU 工作類似，它可以將任何實體記憶體頁面映射成連續的匯流排位址，所以實體記憶體位址與匯流排位址間就沒有 linear mapping 的關係，vir←→bus 不一定能正常運作，因此 vir_to_bus 後來不管是否使用 IOMMU，都只用來做簡單的虛擬位址與匯流排位址的轉換。下圖為 IOMMU 與 CPU 的 MMU 示意圖。



（圖 2.5）IOMMU 與 CPU 的 MMU 示意圖

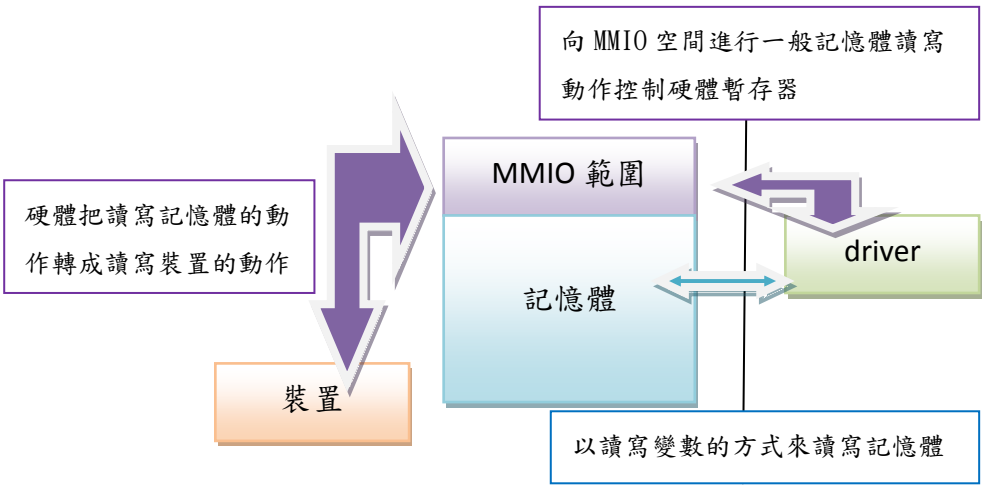
在配置共享記憶體時，我們則選擇使用 dma_alloc_coherent 函式，由系統配置一段 noncache 緩衝區並自動分配這空間的匯流排位址與 kernel space 虛擬記憶體位址讓驅動程式知道，以利驅動程式運用，替驅動程式設計者省去 virtual address to bus address 轉址的過程。

2-4 控制硬體的方式

在設計驅動程式時，除了了解記憶體議題，還需要決定要以何種方式來控制硬體，通常，I/O mapped I/O(I/O port)與 Memory mapped I/O(MMIO)是 CPU 或驅動程式控制硬體的兩種方式。

I/O port 指的是系統為這個設備準備好的一塊 I/O space，而 I/O space 裡面的位址就稱為 I/O port。很久以前的裝置多半各自擁有 I/O port，這是一個讀寫硬體所需的控制硬體介面，只要硬體擁有 I/O port，就可以由驅動程式直接透過 I/O port 進行控制、修改暫存器。但是以 I/O port 的方式來控制硬體只能以 1 byte、2 bytes 或 4 bytes 為單位執行讀寫，大量資料不適合經由 I/O port 傳送，而且控制硬體需要用到的 I/O space 會佔用電路板的位址空間，因此 I/O port 的控制方式已漸漸被 MMIO 取代。於是在資源有限的嵌入式系統裡，便使用了 MMIO 的機制來控制 PNX1005，可以節省 I/O port 占用的電路板空間。

MMIO 是以讀寫系統記憶體(RAM)的動作代替 I/O port 來操控外部裝置，以 PCI 裝置為例，開機時 BIOS 或 OS 會把 PCI 裝置的暫存器或是外部 CPU 控制的記憶體位址空間，映射到系統記憶體去，驅動程式可以透過平常讀寫記憶體的做法，來讀寫 PCI 裝置的暫存器與 PCI 裝置控管的記憶體。在 linux 環境下，會把實體記憶體 3G~4G 的位址空間規劃為 MMIO space，假如實體記憶體不到 4G，當軟體要求讀寫的記憶體範圍超過目前硬體所有的實體記憶體範圍時，只要硬體允許，還是會被硬體當成 MMIO 的讀寫動作，轉換成對各種周邊裝置的讀寫動作。



(圖 2.6)MMIO 運作機制

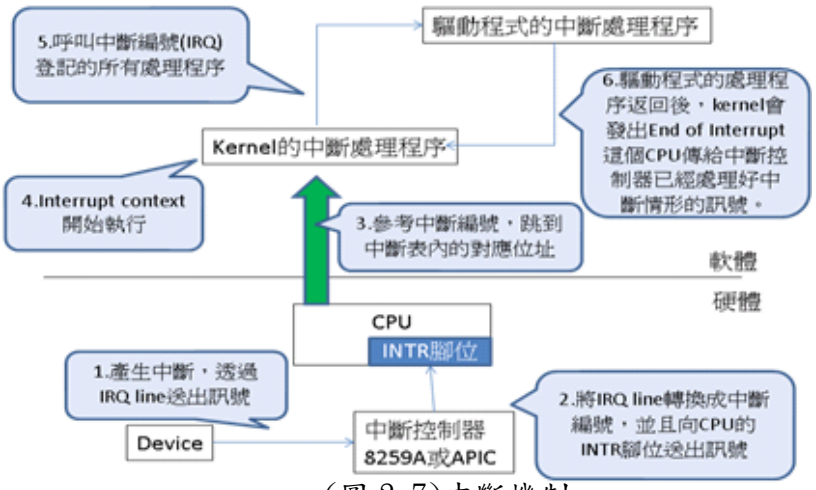
於是，TMMan 驅動程式要控制 PNX1005 的話，同樣也會使用 MMIO 的機制，系統會先將 PNX1005 的 I/O 空間(MMIO aperture)映射到 ARM 端實體記憶體的 3G~4G 這個 MMIO 範圍，當 TMMan 驅動程式要設定 PNX1005 的暫存器時，就可以透過修改 ARM 端實體記憶體，直接設定到 PNX1005 的暫存器。

2-5 interrupt 與 interrupt service routine

第四章在探討兩端 CPU 的溝通機制時，會使用到中斷機制的概念做為通知對方訊息抵達的工具。基本上，不管是哪一種作業系統，只要是藉由中斷來達到通知的目的，都需要有一個中斷處理程序(interrupt service routine，ISR)執行收到中斷後的處理。因為每次中斷就會讓 CPU 暫時停止目前程式的運作，跳到 ISR 去處理，所以我們不希望中斷處理的時間太久，於是使用了多層的觀念來減短硬體 interrupt 的處理時間，將 ISR 分成接收到硬體中斷的頂半部，還有可以藉由 OS 來作排程的底半部，頂半部的 ISR 處理是不能被中斷的，是硬體中斷與軟體處理的中介，而後底半部的處理都可以經由 OS 做適當的排程來完成中斷處理。如果每次收到中斷後只有很少量的後續處理動作，其實只要在頂半部馬上處理即可，但是像 DSP 送過來的訊息還需要做事件與應用服務的分辨，耗費的時間可長可短，所以我們為 ISR 的處理分層，每一層中 ISR 就是一個 callback function，在第四章中會提到有硬體抽象層(屬於頂半部)、虛擬中斷層、同步層與通道層/訊息層，在訊息層作了一個 first in-first out 佇列(queue)，當 DSP 間接傳送許多命令過來時可以依照順序暫時存放在 queue 中等待處理，至此才完成一整個 ISR 的運作。

而 pSos 端收到中斷後，同樣也要有 ISR 來處理 ARM 端回傳回來的 ACK 訊息，為了與 ARM 端有對稱的溝通管道，同樣會將 ISR 的設計分成硬體抽象層、虛擬中斷層、通道層/訊息層、同步層，利用 callback function 一層層傳遞到上層。在軟體設計層面，DSP 端利用 ISR callback function 的傳遞架構與 ARM 端使用的語法跟指標使用方式都是一樣的，

只有在最初註冊 interrupt 的方式不同，會跟著不同的 OS 而有不同的註冊方式。我們以 ARM 端 linux 環境為例，介紹 linux 環境的中斷機制。



(圖 2.7)中斷機制

首先，我們先了解一下 linux 環境裡中斷的概念(如圖 2.7)。裝置與中斷控制器之間有稱為 IRQ(interrupt request) line 的中斷線路，要知道是哪個裝置要求的中斷，只要看是哪一條 IRQ line 送來的訊號就行了。中斷控制器在收到中斷之後會把 IRQ line 轉換成對應的 IRQ 編號，並且向 CPU 的 INTR 腳位送出訊號，通知發生中斷。而裝置的中斷訊號要分配給哪個 IRQ line 是由 BIOS 決定或是設計者一開始就寫死了，與 OS 無關。當驅動程式要註冊 ISR 給 PCI 裝置時，驅動程式會從 pci_dev 結構中的 irq 變數讀出 kernel 分配給它的 IRQ 編號，之後驅動程式就會先讀取出 IRQ 編號，並註冊 interrupt handler(=ISR)給這個 IRQ 編號，因此當 CPU 收到 interrupt 訊號，就會在 kernel 中找到與這個 IRQ 編號對應的 interrupt handler 進行中斷處理。由於 PCI 裝置通常會採用中斷共享模式，也就是與其他硬體共享一條中斷線，因此在註冊 ISR 時會同時給予辨別用的身分參數(可以是驅動程式某個私有位址)，來識別具體產生中斷的裝置。

所以要完成中斷溝通前，首先就要設計好 ISR，接著透過不同的 OS 註冊中斷程序的方法將 ISR 與中斷線/IRQ 編號之間的關係連接起來。在 Linux 環境中，當使用者利用 insmod 指令，執行驅動程式的程式進入點--init_module 時，就可以利用 request_irq 這個函式註冊 interrupt handler 給 IRQ 編號:

```
int request_irq( unsigned int irq,void (*handler)(int, void *),unsigned long
                 irqflags, const char *devname,void* dev_id );
```

說明:request_irq 第一個參數為 PCI 裝置的 IRQ 編號，而 handler 則為指向 interrupt handler 的 function pointer；另外在註冊的時候一個 IRQ 編號是可以讓多個 interrupt handler 共用的，linux kernel 在 CPU 收到中斷時會依序執行同一個 IRQ 編號的 interrupt handler，期間只會去執行由這個驅動程式負責的裝置

產生的 interrupt 所對應的 interrupt handler，這個判斷就必須藉由最後一個參數來決定，因此如果這個 IRQ 編號是共享的，dev_id 一定不可以 NULL；flag 則用來設定這個 IRQ 編號是共享(SA_SHIRQ)的、當執行 interrupt 處理時是停止其他中斷(SA_INTERRUPT)的

2-6 程序並行控制:

在 DVR 平台上，為了讓 DSP 端與 ARM 端溝通(第四章)，在軟體上，DSP 端與 ARM 端都會各自在自己的 OS 使用到許多資料結構來使用共享記憶體。然而 ARM 端與 DSP 端本身都具備兩種角色--傳送端與接收端，以 OS 為 linux 的 ARM 為例，當它為接收端，這些資料結構就會被 interrupt context 使用，但是有可能應用程式處理完資料後也想要將訊息送給 DSP 端，這時 ARM 端又同時為傳送端角色，此時，同樣的資料結構就有可能同時被 user context 使用，為了避免資料結構同時被修改造成結構裡資料的不正確，所以我們必須探討程序並行的控制。

在 linux 環境中常用的方法有兩種: semaphore(旗標)與 spinlock(自旋鎖)。Linux 提供旗標(semaphore 或 mutex)做為一般行程執行環境(user context)同步與鎖定的機制，當某一個 task 要進入已受到 semaphore 保護的 critical section 時，就會讓這個 task 進入睡眠狀態，等到 semaphore 被解放，才能再喚醒這個 task 來對 critical section 進行修改。但是 semaphore 不能用在 interrupt 執行環境(interrupt context)下，因為 interrupt 通常具有很高的優先權，如果讓 interrupt 進入睡眠狀態就再也叫不起來了。所以 interrupt context 下只能使用 spinlock 讓 CPU busy-loop，每次 loop 就是在檢查是否可以使用 critical section。

在運行 semaphore 機制時，通常會使用 down()函式來取得 semaphore，如果 task 不能正常取得 semaphore，就會 sleep 到它可以取得 semaphore 為止；如果要釋放 semaphore 我們則會使用 up()函式，但是如果 task 在無法取得 semaphore 的情況下，希望在必要的時候可以因為使用者下達 Ctrl+C 命令將該 task 結束掉，該 task 在嘗試取得 semaphore 的時候就要使用 down_interrupible()，表示說這個 task 在必要的時候可以接收中斷，結束掉這個 task，所以我們在設計驅動程式的時候也有使用到這個觀念。另外，當 semaphore 初始值為 0 時，則可以用來做同步使用，這個特性在第四章溝通管道中會用來做為阻斷機制。

第二段有提到 interrupt context 下是不能使用 semaphore 的，我們必須使用不會 sleep 的 spinlock 機制以防系統進入 deadlock，而 user context 則可以使用 semaphore 與 spinlock 兩種方式來運作。不過使用 spinlock 必須要注意，busy-loop 可能會 100%占用 CPU 資源，所以一定要想辦法縮短 critical section 的處理時間，然而，使用 spinlock

並不能完全避免 deadlock，在擁有 spinlock 時，這個 task 絕不可以被 sleep，我們從單 CPU 與多 CPU 的環境去說明使用 spinlock 的狀況與其他該注意的地方。

在單 CPU 環境中(例如 ARM 端同時有從 user context 與 interrupt context 要讀取或修改 shared memory 的情況)，如果 critical section 同時出現在 user context 與 interrupt context，兩邊都必須使用 spinlock(只在 user context 出現的 critical section 才可以用 semaphore 來鎖定)。然而，當 user context 取得 spinlock 時沒有禁止本地CPU處理中斷,在user context鎖定其間interrupt context又因為也想使用critical section 導致 interrupt context 也嘗試要鎖定，由於 interrupt 的優先權大於 user context，所以 CPU 一定會處理 interrupt，但是因為使用 spinlock 鎖定的 user context 已經 sleep，表示這個 spinlock 已經解不開了，因此 interrupt context 也只能陷入無限迴圈之中，導致系統當機。所以在一個同時會有多個 user context 與 interrupt context 的環境中，我們必須讓 user context 取得鎖定的同時也要禁止本地 CPU 去處理中斷，於是使用 spin_lock_irqsave(spinlock_t *lock,unsigned long flags)函式達到這個目的，解除所定時則使用 spin_lock_restore(spinlock_t *lock,unsigned long flags)。這也是我們在探討 DSP 與 ARM 端溝通時設計的驅動程式需要注意的地方。

然而上述的問題在多 CPU 做 parallel processing 的環境裡則可能不會發生,假如 user context 在某一顆 CPU(A)上執行,user context 鎖定 critical section 後是另一顆 CPU(B)接收中斷，那麼雖然當時 CPU B 因為無法成功鎖定 critical section，所以暫時進入 busy-loop，但是因為 user context 不會被 sleep，所以 CPU B 一定可以等到 CPU A 上跑的 user context 使用完 critical section 後將 spinlock 釋放,再進入 critical section，因此就不會有 deadlock 問題，於是就可以簡單使用 spin_lock()來鎖定，spin_unlock 則解除鎖定。

不過，不管是單 CPU 還是多 CPU，要盡量避免套疊 spinlock 去鎖定同一個 critical section，例如 user context A 鎖了 critical section 1 與 critical section 2，然後 critical section 1 需要 critical section 2 去執行釋放的動作，但是 critical section 2 要可能必須先由 user context B 來釋放，但是 user context B 因為無法鎖定已被 A 占走的 critical section 2，導致 B 進入 busy-loop，至此進入了 deadlock。

實際應用到 DVR 平台的 DSP 端與 ARM 端溝通上，由於在軟體上用來控制共享記憶體的程序在 ARM 端就不只一個(user/interrupt context)，我們因此也需要使用 spinlock 機制，每次想要修改或讀取共享記憶體的資料結構時，就要先鎖定，處理完後再釋放。

2-7 PCI 裝置

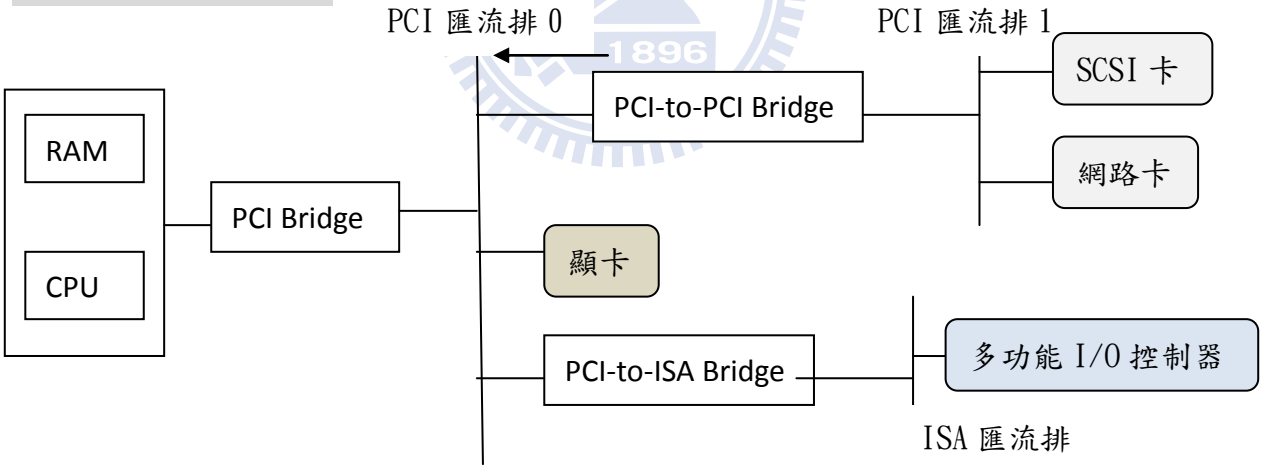
由於 PNX1005 裝置是透過 PCI 匯流排與 ARM 端相連接，所以 PNX1005 對 ARM 來說就是一個外部的 PCI 裝置，因此我們需要了解 PCI 是什麼，然後上面的 PCI 裝置提供哪些資訊讓 host CPU 來控制。

PCI 意即 peripheral component interconnect(周邊元件互連)，是目前最被廣泛使用的一種匯流排標準，利用一組完整的匯流排界面規範，描述了如何將電腦系統中的週邊設備以一種結構化和可控制化的方式連接在一起，並且詳細定義了電腦系統中的各個不同硬體元件之間應該如何正確地進行溝通。我們把接在 PCI 匯流排上的硬體設備稱為 PCI 裝置。

2-7-1 plug & play

PCI 匯流排標準與以往的 ISA 匯流排標準最大的不同在於其具有 plug & play(隨插即用)的功能，當我們插上 PCI 裝置後，分配這個裝置所需資源的工作就交給 BIOS 與 OS 來進行，使用者不必自己去特別設定裝置所需資源(IRQ 編號、I/O port、記憶體映射位址)就能使用這個裝置，不像之前的 ISA 匯流排必須由使用者設定裝置所需資源。

2-7-2 PCI 裝置的連接



(圖 2.8)典型 PCI 匯流排電腦邏輯示意圖

系統的各個 PCI 裝置可以藉由 PCI 匯流排和 PCI-PCI 橋連接在一起，其中 CPU 和 RAM 需要通過 HOST/PCI 橋連接到 PCI 匯流排 0（即主 PCI 匯流排），而具有 PCI 介面的顯卡則可以直接連接到主 PCI 匯流排上。PCI-PCI 橋是一個特殊的 PCI 設備，它負責將 PCI 匯流排 0 和 PCI 匯流排 1（即從 PCI 主線）連接在一起，通常 PCI 匯流排 1 稱為 PCI-PCI 橋的下游（downstream），而 PCI 匯流排 0 則稱為 PCI-PCI 橋的上游（upstream），圖中的下游連接了 SCSI 卡和乙太網卡。為了相容舊的 ISA 匯流排標準，PCI 匯流排還可以通過 PCI-ISA

橋來連接 ISA 匯流排，從而支援以前的 ISA 設備，其中 ISA 匯流排上可以連接一個多功能 I/O 控制器，用於控制鍵盤、滑鼠和軟盤機。

由於 DVR 平台上的 ARM 端使用了 PCIe 的匯流排標準，而 PNX1005 及 DSP 端其他周邊裝置則使用了 PCI 匯流排標準來做連接，因此 ARM 與 PNX1005 之間會有一個 Host/PCI Bridge 做為不同標準間的銜接，這個 Bridge 接出去的 PCI 匯流排上則同時連接了兩個 PNX1005 裝置。

2-7-3 PCI 位址空間

PCI 裝置有三種位址空間，分別為 PCI I/O 空間、PCI 記憶體位址空間、PCI 配置空間(PCI configuration space)。CPU 可以存取所有的位址空間，但我們註冊的裝置驅動程式只能使用 PCI I/O 空間、PCI 記憶體位址空間來控制 PCI 裝置，而 PCI 配置空間則由 linux kernel 提供、與 PCI 初始化相關的函式來設定或使用。實際上，驅動程式控制的 PCI I/O 空間、PCI 記憶體位址空間的資訊會存放在 PCI 配置空間中，驅動程式為了得到 PCI 配置空間的資訊，它於是利用 kernel 提供的函式從 PCI 配置空間讀出 I/O 空間與記憶體位址空間，驅動程式再透過這些空間資訊對這些空間進行讀寫以控制 PCI 裝置。

2-7-4 PCI 配置空間(PCI configuration space)

為了實現 plug & play，每個 PCI 裝置必須能在系統(BIOS 或 kernel 存在的環境)中被偵測出來，並且由系統根據目前硬體資源使用狀況，去分配硬體資源與位址空間給這個 PCI 裝置，所以我們使用 PCI 配置空間來儲存這些由系統分配給它的硬體資源，另外也存放了這個 PCI 裝置的身分代碼。針對不同種類的 PCI 裝置基本上分成三種配置空間，一般的 I/O 介面卡、PCI Bridge 及 CardBus Bridge，而 PNX1005 屬於一般的 I/O 介面卡，所以我們就針對 I/O 介面卡類型的 PCI 配置空間來討論。

PCI 配置空間，它是一個以 32 bit 為單位記錄資訊的位址空間，每個 PCI 裝置都佔有一塊固定大小的 64 dwords(256 byte)區塊，每個 PCI 裝置的 PCI 配置空間的前面 16 dwords(64 byte)欄位都是一樣的，有關 PCI 裝置的硬體資訊都存在這塊 64 byte 的配置空間中，我們稱這塊空間為 PCI 配置空間的 header(如圖 2.9)，下面列出幾個重要的欄位意義(表 2.4)。

Byte Offset	Byte 3	Byte 2	Byte 1	Byte 0
0h	Device ID		Vendor ID	
4h	Status Register		Command Register	
8h	Class Code (020000h)			Revision ID
Ch	BIST (00h)	Header Type (00h)	Latency Timer	Cache Line Size
10h	Base Address 0[a]			
4h	Base Address 1			
18h	Base Address 2			
1Ch	Base Address 3 (unused)			
20h	Base Address 4 (unused)			
2h4	Base Address 5 (unused)			
28h	Cardbus CIS Pointer (not used)			
2Ch	Subsystem ID		Subsystem Vendor ID	
30h	Expansion ROM Base Address			
34h	Reserved			Cap_Ptr
38h	Reserved			
3Ch	Max_Latency (00h)	Min_Grant (FFh)	Interrupt Pin (01h)	Interrupt Line

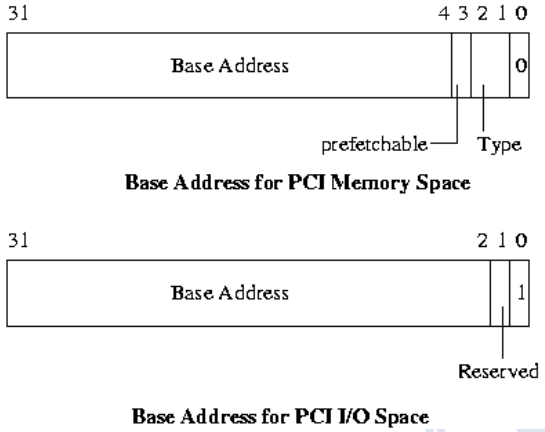
(圖 2.9)PCI 配置空間的 header

(表 2.4)PCI 配置空間 header 的欄位意義

欄位	意義
Vendor ID(廠商標識)	一個唯一的數字標識，用來描述一個 PCI 設備的出處。Digital 的 PCI 廠商標識是 0x1011; Intel 的是 0x8086;而 PNX1005 則為飛利浦的編號 0x1131。
裝置 ID(設備標識)	一個唯一的數字標識用來描述一個設備。例如目前 DVR 平台上的 PNX1005 均為 0x540B。
Base Address 暫存器 (基位址暫存器)	這些暫存器用來決定和分配一個裝置可用的存儲空間及空間類型，可以存放 PCI I/O 空間和 PCI 記憶體空間或 PCI 外部設備的記憶體空間的基底位址。以 PNX1005 來說，存放了暫存器所在的 MMIO、程式運作使用到的 RAM、用來控制外部設備的 XIO 這三個 memory apertures 的基底位址。
狀態暫存器(status)	共有 16 個 bit，用來紀錄 PCI 匯流排相關事件資訊的暫存器。例如: bit3 代表 PCI 裝置的中斷狀態。
命令暫存器(Command)	共有 16 個 bit,可以用來決定是否啟用 PCI 設備的 I/O port 或 MMIO，還有是否可以啟用中斷等。
中斷腳位(interrupt pin)	紀錄 PCI 裝置被分配到哪個 CPU 的中斷腳位。
中斷線(interrupt line)	由 BIOS 或 kernel 分配給這個 PCI 裝置的中斷線。當中斷發生時，藉由這個中斷線轉換成 kernel 中的中斷編號，然後跳到驅動程式裡頭註冊的 interrupt handler 來處理這個事件。

2-7-5 PCI 配置空間的基底位址暫存器

PCI 配置空間的基底位址暫存器(Base address 暫存器,BAR)是驅動程式在讀寫 PCI 裝置時不可或缺的重要暫存器,因為 BAR 存放了可以使用這個 PCI 裝置哪些空間的位址資訊,驅動程式必須得到這些位址資訊後才能控制 PCI 裝置。可是每個 PCI 裝置內部 CPU 看到的這些空間位址不一定和 host CPU 由外面看這些空間的位址一樣,所以我們要特別注意,這個 BAR 儲存的是由外面看 PCI 裝置開出的這塊空間、在外部的系統裡所代表的位址,因為是外面的 CPU 要來控制它,所以也要是這個 CPU 可以看得懂的位址才行。 因為控制 PCI 裝置或讀寫 PCI I/O 及 PCI 記憶體空間的方式有 I/O port 與 MMIO 兩種,BAR 以兩種不同的儲存格式(圖 2.10)來代表該 PCI 空間是以哪種控制方式進行存取的。



Bit0:分辨此 PCI 空間屬於 I/O port space 還是 MMIO space

Bit1 以後根據不同的控制方式有不同的意義。

(圖 2.10)BAR 的儲存格式
(圖片來源:參考文獻[9])

以 PNX1005 裝置為例,我們可以在 linux 環境下利用 **hexdump -C /proc/bus/01/00.0 得到第一顆 PNX1005 的 PCI 配置空間**(表 2.5):PNX1005 PCI 配置空間的 BAR(黑色粗體與紅色代碼所表示的區段)均顯示裝置的位址空間有會映射到 MMIO space,以 0xc0000008 來說,其 least significant 的 4 個 bits 為 1000,所以第 0 個 bit 為 0,對照上圖我們會發現這個 BAR 儲存的 PCI 空間基底位址屬於 MMIO space address,而 0xd8000000 亦同,故以這兩個 BAR 儲存的位址所表示的 PCI 空間均會被映射到 MMIO 空間,表示這段空間會因為外部系統記憶體的讀寫而受到影響。另外,我們必須將 BAR 儲存的位址的 least significant 的 4 個 bits 遮罩成 0,才能代表這段 PCI 空間真正所在的基底位址,因此 PNX1005 的 BAR 中真正得到的基底位址分別為 0xc0000000 與 0xd8000000。而這兩個基底位址在 PNX1005 代表的即是 DSP 端程式運作會使用到的 RAM memory aperture 與暫存器所在的 MMIO memory aperture 的基底位址,如果我們對系統記憶體 0xc0000000 以上的某段空間進行填寫,就會更改到 PNX1005 裡面的記憶體空間,然而在 0xd8000000 以上的某段空間進行填寫,就會設定到 PNX1005 的暫存器。

(表 2.5) PNX1005 的 PCI 配置空間(little endian)

```
00000000  31 11 0b 54 46 03 90 02  00 00 80 04 08 80 00 00  |1..TF...........|
00000010  08 00 00 c0 00 00 00 d8  00 00 00 d0 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 31 11 25 00  |............1.%.|
00000030  00 00 00 00 40 00 00 00  00 00 00 00 09 01 09 18  |....@...........|
00000040  01 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

2-7-6 PCI 配置空間的設定

既然 PCI 配置空間提供的資訊對於後續驅動程式在控制裝置時非常重要，那這些資訊是如何被設定的還有什麼時候被設定的就顯得格外重要。

在 Intel-base 的系統架構下，開機時 BIOS 做完 PCI 與硬體資源分配後(如 MMIO 相關的映射工作、中斷線的分配)，便會將硬體資源資訊設定到 PCI 裝置的 PCI 配置空間中，kernel 也會利用 PCI 初始化相關函式來進行 PCI 的初始化，概念上這些 PCI 初始化函式組成就像是一個在 kernel 中的 PCI 裝置驅動程式，它並不是上一章提到的、用來控制 PCI 裝置真正運作的驅動程式，這個 PCI 裝置驅動程式只是用來協助真正用來操控裝置的驅動程式而已。而真正用來控制裝置運作的驅動程式裡頭必須利用 kernel 提供的函式才可以從這個配置空間讀出這個 PCI 裝置的硬體資訊，真正的驅動程式再利用這個資訊對這個空間進行讀寫來控制這個 PCI 裝置。

但是在非 Intel-base 的系統架構下(例如 ARM 的嵌入式系統)，可能沒有 BIOS 來做資源配置的動作，所以 linux 裡的 kernel 就要多負責一些 BIOS 該做的事。在 linux 作業系統環境中，當 kernel 被啟動的時候會有一個 PCI 初始化的機制代替 BIOS 做板子上 PCI 裝置的掃描與硬體資源配置，這時候就由 kernel 裡具有 BIOS 功能的函式將系統規畫好的 MMIO 空間及其他位址空間設定到 PCI 配置空間去。像 DVR 平台這樣的非 Intel-base 的系統架構，開機時等到 kernel 被 bootloader 載入到 RAM 以後，kernel 會利用 kernel 設計好的 PCI 初始化相關函式來進行 PCI 初始化，kernel 中做 PCI 裝置初始化的 PCI 裝置驅動程式可以偵測出板子上的 PCI 裝置，並利用 kernel 裡具有 BIOS 功能的函式來配置硬體資源。

每個 PCI 裝置的 Vendor ID 與裝置 ID 在開機時、載入 kernel 前就可能藉由 EEPROM 等韌體先寫入 PCI 配置空間的前四個 byte，當 kernel 啟動、要開始進行 PCI 掃描時，kernel 裡的 PCI 裝置驅動程式會從 Bus 0 開始掃描，然後讀取 PCI 裝置的 PCI 配置空間裡的 vendor ID 和裝置 ID，如果讀回的值是 0xFFFF，則說明該 PCI 裝置是不存在的；如果發

現 PCI slot 有被插上 PCI 裝置,便會在系統中建立一個用來描述這個 PCI 裝置的 pci_dev 資料結構(表 2.6),並使用 pci_dev 裡的 global_list 成員將系統中的 pci_dev 都連結起來,完成一個 PCI 設備鏈結表。這裡使用到的 pci_dev 為 Linux 核心中用來描述 PCI 裝置的資料結構,所有種類的 PCI 裝置都可以用 pci_dev 架構體來描述,讓驅動程式不需要接觸到硬體上的 PCI 配置空間就可以讀到 PCI 裝置的資源。這個結構詳細描述了一個 PCI 設備幾乎所有的硬體資訊,包括廠商 ID、設備 ID、IRQ 編號、BAR、各種在 PCI 配置空間所存取的資訊等等。因此透過 kernel 提供的特定函式,我們就可以從 pci_dev 結構裡提取我們需要的 PCI 裝置的硬體資訊。

(表 2.6)kernel 2.6.22.18 的 pci_dev 結構

```
struct pci_dev {
    struct list_head global_list;     /* node in list of all PCI devices */
    struct list_head bus_list;   /* node in per-bus list */
    struct pci_bus *bus;         /* bus this device is on */
    struct pci_bus *subordinate; /* bus this device bridges to */

    void     *sysdata; /* hook for sys-specific extension */
    struct proc_dir_entry *procent;   /* device entry in /proc/bus/pci */

    unsigned int   devfn;        /* encoded device & function index */
    unsigned short vendor;
    unsigned short device;
    unsigned short subsystem_vendor;
    unsigned short subsystem_device;
    unsigned int   class;        /* 3 bytes: (base,sub,prog-if) */
    u8       hdr_type; /* PCI header type (`multi' flag masked out) */
    u8       rom_base_reg; /* which config 暫存器 controls the ROM */
    u8       pin;          /* which interrupt pin this device uses */
    struct pci_driver *driver;   /* which driver has allocated this device */
    u64      dma_mask; /* Mask of the bits of bus address this
                          device implements.  Normally this is
                          0xffffffff.  You only need to change
                          this if your device has broken DMA
                          or supports 64-bit transfers.  */

    pci_power_t     current_state;  /* Current operating state. In ACPI-speak,
                          this is D0-D3, D0 being fully functional,
                          and D3 being off. */

    pci_channel_state_t error_state; /* current connectivity state */
    struct    device    dev;      /* Generic device interface */

    /* device is compatible with these IDs */
    unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
    unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];

    int      cfg_size; /* Size of configuration space */

    /*
     * Instead of touching interrupt line and base address 暫存器
     * directly, use the values stored here. They might be different!
     */
```

```
    unsigned int  irq;
    struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory regions + expansion ROMs
*/

    /* These fields are used by common fixups */
    unsigned int  transparent:1;/* Transparent PCI bridge */
    unsigned int  multifunction:1;/* Part of multi-function device */
    /* keep track of device state */
    unsigned int  is_busmaster:1; /* device is busmaster */
    unsigned int  no_msi:1; /* device may not use msi */
    unsigned int  no_d1d2:1;   /* only allow d0 or d3 */
    unsigned int  block_ucfg_access:1;    /* userspace config space access is blocked */
    unsigned int  broken_parity_status:1; /* Device generates false positive parity */
    unsigned int  msi_enabled:1;
    unsigned int  msix_enabled:1;
    unsigned int  is_managed:1;
    atomic_t  enable_cnt;    /* pci_enable_device has been called */

    u32      saved_config_space[16]; /* config space saved at suspend time */
    struct hlist_head saved_cap_space;
    struct bin_attribute *rom_attr; /* attribute descriptor for sysfs ROM entry */
    int rom_attr_enabled;        /* has display of the rom attribute been enabled? */
    struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE]; /* sysfs file for resources */
#ifdef CONFIG_PCI_MSI
    struct list_head msi_list;
#endif
};
```

掃描完成後，kernel 又會利用 kernel 裡具 BIOS 功能的 PCI fixup 這類函式，為之前由 PCI 裝置驅動程式建立起來的 PCI 設備鏈結表中的每個 PCI 裝置，進行硬體資源配置(I/O 空間、記憶體空間及 IRQ 編號)、設定 I/O 空間或記憶體空間的使用，並將配置好的資料設定到 PCI 配置空間與 pci_dev 結構中，完成 kernel 的 PCI 初始化程序。

kernel 的 PCI 初始化程序結束後，我們便可依照第三章提到的驅動程式註冊程序，將驅動程式運作實體註冊到驅動程式中，之後應用程式與硬體間的溝通，便靠這個驅動程式來操作。

2-7-7 PCI 配置空間的存取

完成 kernel 的 PCI 初始化動作後，由於 PCI 配置空間並不能直接被驅動程式透過指標直接取用，我們必須利用 kernel 提供的幾種函式來讀出 PCI 配置空間的資訊。這些函式裡面都需要放入 Linux 核心用來描述 PCI 裝置的資料結構—pci_dev(在 linux kernel 的原始碼 include/linux/pci.h 中)，因為它詳細描述了一個 PCI 設備幾乎所有的硬體資訊，包括廠商 ID、設備 ID、IRQ 編號、BAR、各種在 PCI 配置空間所存取的資訊等等。以下列出在 TMMan 驅動程式中會用到的函式(表 2.7):

(表 2.7)讀取 PCI 配置空間的 kernel 函式

函式名稱	函式功能
Kernel 2.6 以前的版本: struct pci_dev* pci_find_device(int Vendor ID,int device ID,struct pci_dev* from) kernel 2.6 以上的版本: struct pci_dev* pci_get_device(int Vendor ID,int device ID,struct pci_dev* from)	比對 kernel 中的 pci 設備清單,從 from 指到的這個 pci_dev 結構開始找,找出與要求的 Vendor ID、device ID 相同的代表該裝置的 pci_dev 結構。第一次搜尋時 from 設成 0。
unsigned long pci resource_start(struct* pci_dev* dev,int bar)	讀取 bar 這個參數指定的基底位址暫存器開始位址。bar:0~5,分別對應到配置空間的 10h~24h。
unsigned long pci resource_end(struct* pci_dev* dev,int bar)	讀取 bar 這個參數指定的基底位址暫存器結束位址。bar:0~5,分別對應到配置空間的 10h~24h。
unsigned long pci resource_length(struct* pci_dev* dev,int bar)	讀取 bar 這個參數指定的基底位址暫存器大小。bar:0~5,分別對應到配置空間的 10h~24h。
pci_read_config_dword	讀 PCI 配置空間的資料
pci_write_config_dword	修改或寫入 PCI 配置空間資料

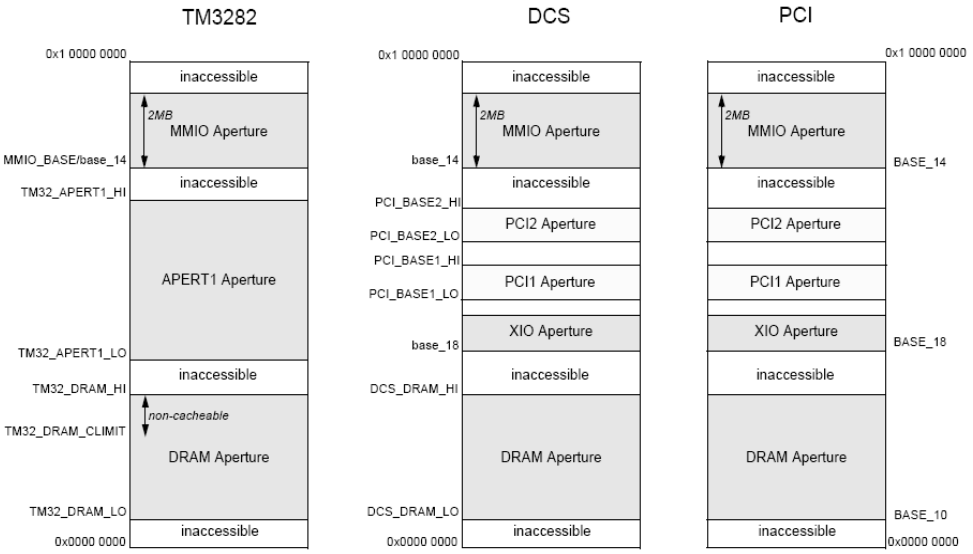
2-8 PNX1005 的 Memory apertures

Memory apertures 為 PNX1005 開出來給 ARM 端,讓 ARM 可以控制 PNX1005 裝置的 DSP 端暫存器與程式運作的記憶體空間,這些空間會由 BIOS 或 kernel 在 PCI 初始化階段映射到 ARM 端實體記憶體上,並設定硬體資源到 PCI 配置空間的 BAR 去。

在 host mode 的模式下(PNX1005 為 target),PNX1005 裝置遵照 PCI memory addressing 規範對 32-bit 位址空間作了規畫,將 PCI bus 的 32-bit address space 規畫出 RAM、MMIO apertures 來和 ARM 溝通,RAM aperture 主要就是 DSP 端執行成式運作的記憶體空間,MMIO apertures 則包含所有 PNX1005 的暫存器。然而如果 PNX1005 是在 standalone mode,它自己就為 host,所以 PNX1005 裡的 CPU 會利用 XIO apertures 與 DSP 端其他比較慢速的裝置(像是 disk、flash)溝通,並自動篩掉外部其他 PCI 裝置對這個 aperture 的操作;但是因為 DVR 平台是在 host mode 下運作,我們並沒有把硬碟接在 DSP 端,而是由 ARM 來管理硬碟,因此用不到 XIO aperture。

有了 PNX1005 memory apertures 後，為了讓 ARM 可以控制 PNX1005，PNX1005 裝置的 MMIO aperture 區段則會被映射到 ARM 的實體記憶體 MMIO 空間中，之後 ARM 便可以利用讀寫自己實體記憶體空間中的 MMIO 空間來設定這些暫存器。另外，ARM 端也可以將 RAM aperture 映射到 ARM 端實體記憶體中，再利用 2-3-2 提到的 mmap 機制將這塊實體記憶體映射到應用程式的 user space 中，利用 memcpy 的方式將 DSP 端可執行檔複製到 DSP 端這塊 aperture 開出來的區段。

了解每個 apertures 的功能後，我們在討論一下三個不同觀點下看到的 apertures—DSP 端內部的 CPU、internal bus(DCS—device control and status)、PCI view（如圖 2.11）。



（圖 2.11）PNX1005 的 memory apertures

（圖片來源: PNX100X Series Data Book）

DCS view

DCS 是 PNX1005 內部的 CPU 與外部 CPU 看這個 PCI 裝置的 aperture 之間的連結，它會判斷 requests 是從外部 CPU(ARM)還是內部 CPU(TM3282)產生的，並且判斷 CPU 傳送的 request 坐落在哪個 aperture 上然後將 request 的送到正確的硬體模組或暫存器，例如，如果 ARM/PCI bus 或是 TM3282 發送的 request 被發送到 MMIO aperture，則 DCS 會把這個 request 送到 PNX1005 裡對應的 MMIO 暫存器去；但是如果 DCS 發現這個從 ARM/PCI bus 或是 TM3282 送過來的 request 位於 DCS_DRAM_HI 與 DCS_DRAM_LO 的區段，則這個 request 會直接透過 MTL bus 傳送到與 DRAM 連接的 MMI module 去(MMI 是一個負責在 DSP 端 DRAM 與 TM3282 接傳送 binary program 硬體模組)，而不經過 DCS bus。如果 request 落在不正確的 apertures 上，DCS 也會直接將這個 request 給。因為 ARM 要控制這個裝置是從 DSP 的外部看進去的，所以我們把重心放在以 PCI 觀點來看這些 apertures。

PNX1005 內部 CPU view

對於 PNX1005 來說，它同樣會看到 DRAM aperture 與 MMIO aperture，只是這些 apertures 的位置會由 host 端設定 TM32_DRAM_HI、TM32_DRAM_LO、TM32_DRAM_CLIMIT、MMIO_BASE 等暫存器來限制 DSP 端 CPU 的使用範圍。因為 PNX1005 晶片裡面也有 CPU，於是這個內部的 CPU 就可以透過 RAM aperture 來讀取 DSP 端 RAM 裡儲存的 instruction 或是其它 PCI master 傳送過來的資料，也利用 MMIO aperture 設定 PNX1005 的暫存器來控制這個 PNX1005，而 Aperture1 則是 PNX1005 用來控制外部其他 DSP 端裝置(PCI slave)的管道。

PCI view

當 ARM 端要使用 DSP 端 RAM 與 MMIO apertures 前，必須先從 PCI 配置空間的 BAR 取出 apertures 映射到實體記憶體中的基底位址，而目前 DVR 平台上所定義的 RAM apertures 基底位址為 Base_10=0xC0000000，MMIO apertures 的基底位址為 Base_14=0xD8000000。由(表 2.5)的說明我們可以發現兩者的 BAR 儲存格式均屬 MMIO address，所以 ARM 想要控制 PNX1005 的暫存器時，只要在自己系統記憶體的 MMIO 區塊內，將 MMIO aperture 映射到的實體記憶體基底位址，利用 ioremap_uncached 轉成虛擬記憶體的 kernel address，就可以讓 kernel 裡的驅動程式使用這個 kernel address 加上適當 offset，對這個記憶體位址做寫入，以設定該位址對應到的特定暫存器來控制 PCI 裝置的運作；而 RAM aperture 則可利用先前提過的 mmap 系統呼叫將占有的 physical memory 空間映射到 virtual memory 的 user space 去，ARM 端的應用程式就可以在 virtual memory 空間進行檔案的複製，讓 ARM 端的應用程式可以將 DSP 端可執行檔直接下載到 DSP 端的 RAM 去。

2-9 TMMan 驅動程式對 PNX1005 這個 PCI 裝置的描述與使用

PNX1005 對 ARM 來說無非就是一個外部的 PCI 裝置，當 kernel 的 PCI 初始化運作結束後，我們可以在/proc/bus/pci 下的 devices 這個檔案，看到板子上所有 PCI 裝置的設備清單，ARM 端的應用程式想要控制 PNX1005，便需要先註冊可以控制 PNX1005 的驅動程式(TMMan 驅動程式)，然後 TMMan 驅動程式藉由上述 kernel 提供、可以用來存取 PCI 配置空間的函式，來取出 PNX1005 的 apertures 基底位址，透過這些位址做適當的操作便可在 ARM 與 PNX1005 傳遞資料。

另外，如果在 linux 環境中想要得到 memory apertures 的大小，TMMan 驅動程式中是用 pci_find_device (或 pci_get_device in kernel 2.6 以上)來查找 kernel 中的 PCI 設備清單，得到描述該 PCI 裝置的 pci_dev 結構，之後可以利用 pci_resource_start (struct pci_dev*,int bar)從 pci_dev 裡，指定要讀取的是第 bar 個 BAR，然後取得 BAR 所存放的基底位址，再使用 pci_resource_end(struct pci_dev*,int bar)從 pci_dev 結

構中取出第 bar 個基底位址代表的 PCI 空間極限位址，然後利用 pci_resource_end-pci_resource_start 將兩個函式得出的值相減，即可得到 apertures 的大小，同時，驅動程式也可以將這些從配置空間取得的資訊填入 DSP 端 MMIO aperture 裡限制範圍用的暫存器中。

2-10 第二章結論

在 linux 環境裡設計應用程式或驅動程式時，virtual memory 提供很大的記憶體空間讓開發者可以不用顧慮記憶體上限的問題來設計與開發，但是程式或指令一定要載入到 physical memory 才可以被 CPU 所用，所以 virtual memory 與 physical memory 之間的對應關係就顯得格外重要，而 virtual memory 又因為分成了 user space 與 kernel space，因此我們需要透過 ioremap 與 mmap 機制建立 page table 將 physical memory 映射到虛擬記憶體的 user space 與 kernel space 去。有了映射機制的好處是，ARM 端使用者與驅動程式的程式開發者可以不用在意實體記憶體的分布狀況，在操作虛擬記憶體的同時，linux 會自動將虛擬記憶體的操作轉換成對實體記憶體的操作，讓 ARM 端可直接對 PNX1005 的暫存器進行設定或是將資料複製到 DSP 端的 RAM 上，節省資料傳送的時間。

有了 linux 提供的記憶體映射機制後，ARM 端要控制 PNX1005 前當然就必須知道它的暫存器、RAM 位址在哪邊，由於 PNX1005 是一個 PCI 裝置，藉由 PCI 裝置的配置空間驅動程式可以從中得到必要的位址資訊與硬體資訊，進而搭配記憶體的映射機制完成 PNX1005 的資料傳輸。然而因為 PNX1005 與 ARM 端在平台上是兩顆獨立運作的 CPU，所以我們還使用了共享記憶體與中斷機制完成兩端的訊息溝通，而且在使用 shared memory 時，還需要注意 critical section 的議題。

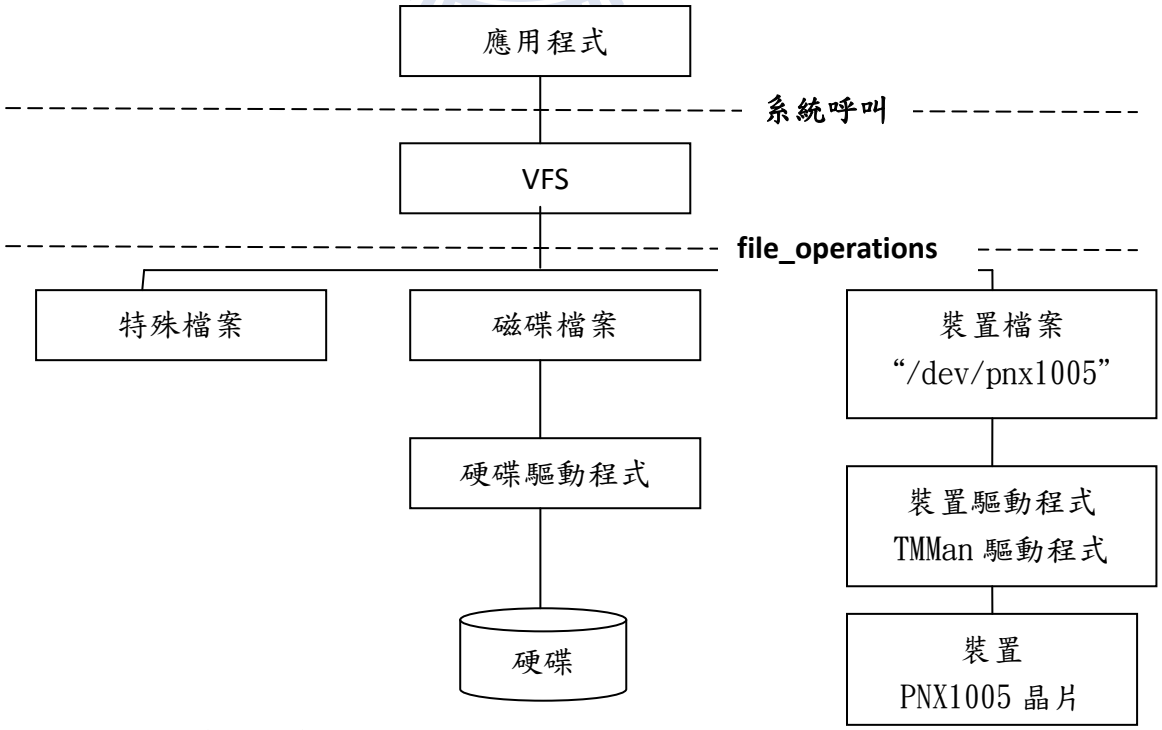
第三章 控制 PNX1005 的 Linux 驅動程式與功能函式庫

為了控制平台上的 PNX1005 裝置，我們在 ARM 端設計了一個驅動程式來控制它，而這個驅動程式我們稱為 TMMan 驅動程式。因為 ARM 的 OS 為 linux，所以 TMMan 驅動程式會利用第二章提到的知識，以 linux 驅動程式架構來實作的驅動程式。3-1 節會先介紹 linux 基本的驅動程式設計環境與操作介面，然後說明設計 TMMan 驅動程式實際上需要設計的是哪些實體操作函式(3-1-3 TMMan 驅動程式的 driver handlers)，而這些實體函數與系統呼叫之間的關係又是什麼。設計完 TMMan 驅動程式的實體操作函式後，藉由 3-2 先說明一般 linux 環境下，啟動驅動程式的方法以及應用程式要使用這個驅動程式時，用來建立應用程式與驅動程式之間的裝置檔所扮演的角色。3-3 節於是總括 3-1、3-2 節所述，統整 TMMan 驅動程式需要設計、介紹 TMMan 驅動程式內部又多加的設計層次(HAL)，並在啟動完 TMMan 驅動程式後，針對 TMMan 驅動程式內部進行的初始化動作做了說明。

最後 3-4 節再以驅動程式提供的基礎硬體控制介面(ioctl 系統呼叫)，設計方便應用程式開發者使用、具功能性的功能函式庫，作為驅動程式與應用程式的介面，讓應用程式開發者可以直接使用這組功能函式，完成一個必須由多個對硬體的基礎控制才能達到的功能。

3-1 Linux 驅動程式設計環境與操作介面的設計架構

3-1-1 linux 驅動程式程式運作環境



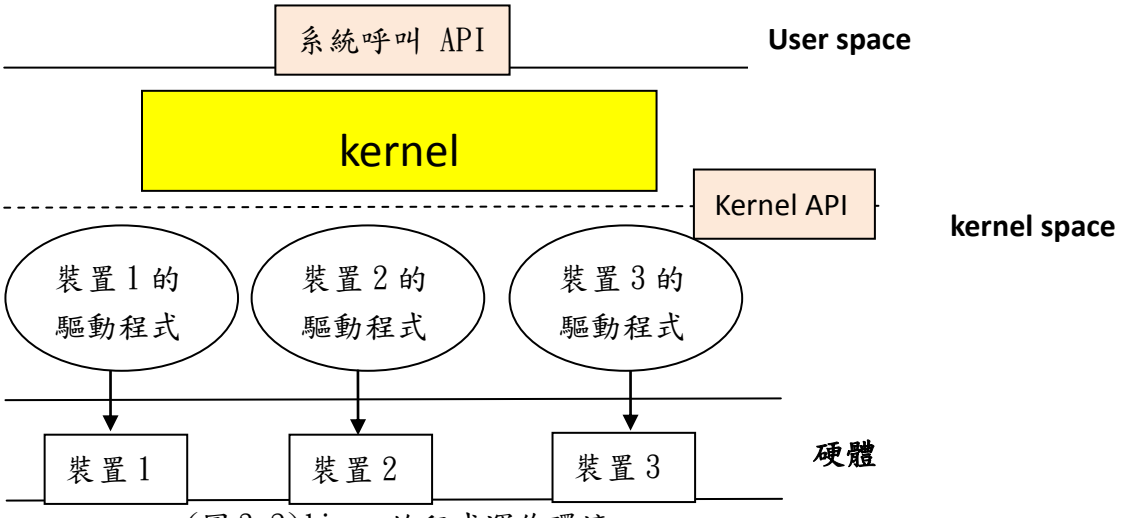
(圖 3.1) 驅動程式與應用程式、裝置之間的關係圖

在 linux 環境中，從(圖 3.1)我們可以看到，應用程式與 VFS(virtual file system)之間的界面是系統呼叫，而 VFS 與磁碟檔案系統以及普通裝置之間的界面是 file_operations 結構體成員函式，file_operations 結構體中每個成員變數均為一個 function pointer，分別指到對檔案進行打開(open)、關閉(close)、讀寫(read/write)、控制(ioctl)、映射(mmap)等一系列成員函式，而這個 file_operations 結構也會在檔案被開啟時儲存到代表該檔案的 file 結構裡去，因此不管是哪一種類型的檔案，基本上都會有與之對應的 file_operations 結構。

在 linux 環境中，會把**裝置看成是一個檔案，也就是所謂的裝置檔**，對於區塊裝置(EX:硬碟)而言，ext2、ext3、fat、jffs2 等檔案系統中就會建置好針對 VFS 的 file_operations 成員函式，裝置驅動程式(EX:硬碟驅動程式)將不用另外再設計 file_operations 儲存的驅動程式實作，而使用者仍可以要求將硬碟裡某個檔案映射到虛擬記憶體的 user space 中。

由於字元裝置(EX:PCI 裝置，非硬碟)的上層沒有磁碟檔案系統，所以字元裝置的驅動程式必須額外設計 file_operations 裡的成員函式，而這些成員函式也就負責了驅動程式對裝置所需的運作，因此我們也把這些成員函式稱為 driver handler。由於 PNX1005 對 ARM 來說就是一個 PCI 裝置，我們並不是用它來做硬碟提供的檔案系統功能，所以設計 TMMan 驅動程式時我們就需要對 file_operations 結構指到的成員函式做設計。

在 linux 環境中設計驅動程式，我們必須了解 linux 中的程式運作環境，它會將程式運作環境分成 user space、kernel space 與硬體(如圖 3.2)，user space 指的就是應用程式所在的虛擬記憶體空間，kernel space 則是 kernel 掌管的實體記憶體位址空間，驅動程式則是 kernel space 的一部分，是用來與硬體做溝通的軟件，當應用程式需要使用驅動程式來控制硬體時，就必須把資料從 user space 複製到 kernel space 後再給驅動程式使用，因此我們必須善用這些程式運作環境來完成驅動程式的開發。



(圖 3.2)linux 的程式運作環境

為了讓資料在不同的程式運作環境裡轉換,我們在不同的 space 之間會定義一個介面,用來銜接兩個不同的程式運作空間，其中 user space 與 kernel space 轉換的介面我們稱為系統呼叫 API，這些 API 就是我們一般常見的 C library 或是 Linux command library；而 kernel space 裡頭 linux kernel 與驅動程式之間的溝通界面則可稱為 kernel API，為 Linux kernel 原始碼中，提供驅動程式實作函式的函式定義(function prototype)。當應用程式使用某個系統呼叫後，kernel 就會將這個系統呼叫與 kernel API 裡某個函式做一對一的對應，所以驅動程式開發者必須實作好 kernel API 裡所有的函式實體，由這些函式實體來完成驅動程式要做的事情。另外要特別注意的是,雖然每個系統呼叫與 kernel API 的函式之間存在一對一的對應，但是它們之間的參數傳遞--應用程式呼叫系統呼叫時傳入的參數型態與資訊,必須經由 OS 轉成特定的結構型態才能被 kernel API 利用,之後 kernel API 再將資料傳給驅動程式使用。

3-1-2 file_operations 結構

因為 kernel API 的所有 function 實體其實就是在實作驅動程式要做的事，所以我們也把 kernel API 的 function 實體也稱為 driver handler，而這些 driver handler 會由 kernel 定義的 file_operations 結構來管理(表 3.1，/linux/fs.h)，在註冊這個驅動程式給 kernel 時，會同時將這個 file_operations 結構告知 kernel。

(表 3.1)linux kernel 2.6.22.18 所定義的 file_operations 結構

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
    long, unsigned long, unsigned long);
```

```
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
    size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
    size_t, unsigned int);
};
```

file_operations 的結構裡每一個成員，都是一個函數指標，指向驅動程式開發者為裝置驅動程式所寫的函數。因此驅動程式的開發者必須根據需要，「依照成員宣告順序」將對應的驅動程式函數實體位址註冊給這些函數指標(如表 3.2)。表中的<trimedia_function>為驅動程式開發者自己設計的函數實體名稱。

(表 3.2)TMMan 驅動程式開發時，註冊到 file_operations 結構中的 driver handler

```
struct file_operations trimedia_fops = {
    .llseek = trimedia_lseek,
    .read   = trimedia_read,
    .write  = trimedia_write,
    .poll   = NULL,                 // trimedia_select
    .ioctl  = trimedia_ioctl,
    .mmap   = trimedia_mmap,        // trimedia_mmap
    .open   = trimedia_open,
    .release = trimedia_release
};
```

而 file_operations 定義的內容其實就是上面提到的 kernel API，裡面的函數指標所指到的每個函數都會對應到 user space 使用的系統呼叫 (如表 3.3)。因為在註冊驅動程式給 kernel 時，就會同時告知 kernel，這個驅動程式是靠哪個 file_operations 結構指到的函數來時作，所以當我們呼叫某個系統呼叫時，就必須指定裝置檔，因為裝置檔中會存有 file_operaionts 結構位址，可以用來決定要使用的驅動程式，而後 kernel 就會知道要利用該 file_operations 裡、與系統呼叫對應的函數來處理事件，這個函數就是該系統呼叫 的 driver handler。

(表 3.3)TMMan 驅動程式的 driver handler 與系統呼叫對應關係

系統呼叫			驅動程式 handler
ioctl	→	kernel	→ trimedia_ioctl
mmap			trimedia_mmap
open			trimedia_open
close			trimedia_release

3-1-3 TMMan 驅動程式的 driver handlers

在 file_operations 結構裡有最重要的兩個 driver handler 是一定要被註冊的，分別是 **open handler 與 close handler** 來做開啟/關閉裝置檔的動作，open handler 會依照

檔名去開啟裝置檔，來連接應用程式與驅動程式，並依照需要，在裝置檔的 file 結構中儲存每個 user process 私有的驅動程式資訊；而 close handler 則負責釋放 open handler 裡占用的記憶體資源並關閉檔案。有了 open 與 close handler 後，驅動程式開發者可以依照裝置功能，去設計 read 與 write handler、mmap handler、ioctl handler 等等的函式實體，並將這些函式位址儲存到 file_operations 結構(表 3.1)中。而這些 handler 使用到的 file 結構，必須來自應用程式使用 open 系統呼叫後，它所回傳、用來代表這個 file 結構的 file descriptor，因為 file 結構指定了要使用的 file_operations 結構，所以當我們在使用其他系統呼叫時傳入 file descriptor，kernel 才能使用正確的驅動程式與 driver handlers 來控制裝置。

如果系統中有設計 read、write handler，使用者呼叫 read、write 系統呼叫時就可以透過 read/write handler 做裝置檔的讀寫，但因為 PNX1005 並不是一個用來做檔案讀寫的裝置，所以我們並沒有實作 read 及 write 的部分；而 **ioctl/ioctl handler** 是一個多功能的驅動程式控制介面，ioctl( )的用法、功能與裝置提供的服務密切關聯，它主要是為了讓應用程式可以透過這個介面要求驅動程式做出應用程式要求的硬體相關操作。舉例來說，在 PNX1005 為 target、ARM 為 host 的環境中，ARM 端的應用程式可以利用不同的 ioctl 系統呼叫取得 DSP 端 memory address 資訊、控制 DSP 的運作、使用共享記憶體來處理 DSP 與 ARM 端的溝通、處理 DSP 與 ARM 端的同步機制、設定 DSP 暫存器等。

另外，一般情況下，虛擬記憶體的 user space 是不可能也不應該直接存取裝置的，但是 **mmap** 這個系統呼叫可以讓使用者在 user space 就可以存取裝置的實體位址。它將虛擬記憶體 user space 的一段空間與裝置的記憶體關聯，當使用者存取 user space 的這段位址範圍時，實際上會轉成對該裝置記憶體空間的存取，可以加速資料的傳遞。因此在 TMMan 驅動程式的設計中，我們也必須實作 file_operations 結構中指到的 mmap handler 函式，依照應用程式所需，分別把 DSP 端的 RAM、暫存器空間映射到 ARM 端虛擬記憶體空間去。以下我們介紹(表 3.3)提到的 driver handlers 與系統呼叫的關係，以及 TMMan 驅動程式中對每個 driver handlers 的實作方式。

Open handler 與 close handler—開啟/關閉裝置檔

open 與 close handler 所需的參數都有來自 kernel 給予的 inode 與 file 結構的指標，inode 結構是在使用者利用 insomd 註冊驅動程式之後，再利用 mknod 這個指令建立裝置檔時由 kernel 建立的，所以每個裝置檔只會有一個對應的 inode 結構，主要存放這個驅動程式控制的裝置的 major 與 minor number 資訊；file 結構內容則如(表 3.4)所列的 file 結構成員，我們輸入裝置檔名稱，由 OS 到檔案系統裡去找尋這個檔案，並取得代表

這個檔案的 file 結構，而這個 file 結構主要存放了 file_operations 這個大結構的位址，file 結構裡的私有資料指標則是驅動程式可以自由使用的指標，能夠用來指到每次在不同的 process 裡、裝置檔被 open 時，由 kernel 配置、用來儲存驅動程式私有資訊的一塊記憶體空間，如果沒有特殊需求，這個指標也是可以不被設定。

(表 3.4)開啟裝置檔

系統呼叫:

```
int open(const char *pathname,int flags);
```

依照 pathname 找到該檔案，然後於 kernel 中取得其 file 結構與 inode 結構

file 結構成員	意義
Struct file_operations *f_op	一個指向 file_operations 結構的指標
Unsigned int f_flags	use space 呼叫 open 這個系統呼叫需要傳入的第二參數，表示可以使用這個裝置檔的使用者權限會由 OS 設定到這個 file 結構中
Void *private_data	驅動程式的私有指標

driver handler:

```
int (*open)(struct node *node,struct file *file);
```

TMMan 驅動程式裡的 open handler 執行時，還會在 kernel 中創建一個資料結構，儲存有關驅動程式在做 mmap 映射時所需的內部資訊—記錄開啟檔案的 process、以代號表示即將對 PNX1005 apertures 或共享記憶體做記憶體映射、以數字 0 或 1 表示要對哪個 PNX1005 操作記憶體映射，最後把 file 結構內的私有指標指到這塊資料結構佔有的空間。

```
int trimedia_open(struct inode *inode, struct file *filep)
{
    Int32       Status;
    ClientObject*Client;

    Client = (ClientObject*)kmalloc(sizeof(ClientObject), GFP_KERNEL) );

    // initialize whatever is in filp->private_data
    Client->Process = GetCurrentProcess();
    Client->MapState = constTMMan_MEMORY_BLOCK_SDRAM;
    Client->DSPNumber = 0;
    filep->private_data = (UInt32*)Client;
    return 0;

    trimedia_open_exit1 :
    return Status;
}
```

（圖 3.3）trmedia_open 函式

當應用程式使用 close 系統呼叫時，透過 int close(int fd)，進入 kernel 後會由 release handler 來負責將 open 時佔用的記憶體資源釋放。

```
int trimedia_release(struct inode *inode,struct file *filep)
{
    kfree(filep->private_data);
    return 0;
}
```

(圖 3.4) trmedia_release 函式

ioctl handler—控制裝置的多功能操作

ioctl handler 其實就像是一個控制總管，我們可以請它依照應用程式不同的需求對裝置做出不同的設定或請求，**在控制 PNX1005 的驅動程式設計中是最為重要的**。為了讓它擁有多種控制選擇，ioctl handler 裡使用一個 switch 結構，每個 case 會對應到一個命令碼，這個命令碼是驅動程式開發者在一開始設計驅動程式時就要定義好的 unique 代碼，當應用程式呼叫 ioctl 系統呼叫時，必須在參數中指定要用到的命令碼，來告知 ioctl handler 需要處理的事件是哪一個，當然我們也要為每個命令碼都實作出相對應的動作。

另外，設計 ioctl handler 時(如表 3.5)，應用程式想要傳給驅動程式的資料，我們可以用一個資料結構包裝起來，然後透過 ioctl 系統呼叫把資料結構的位址傳給 ioctl handler，ioctl handler 便可使用這個資料結構內容或是將資訊填入這個資料結構的成員變數中，再回傳給應用程式。但是因為 linux 環境裡應用程式是在 user space 進行開發，應用程式宣告的資料結構所存的位址是 user space 位址而不是 kernel space 的位址空間，所以當我們要利用 kernel space 的驅動程式來做事時，必須先將資料從 user space 複製到 kernel space。

(表 3.5)TMMan 驅動程式的 ioctl handler

```
trimedia_ioctl(
    struct inode *inode,
    struct file *filep,
    unsigned int cmd,
    unsigned long arg)
{
    IOREQUEST   KernelIoRequest;
    PIOREQUEST  UserIoRequest;
    UInt8*      IOParameters;
    UInt32      SystemLength;//用來做資料讀寫的資料結構大小

    UserIoRequest = (PIOREQUEST)arg;//使用者空間位址
    copy_from_user( (Pointer)&KernelIoRequest, (Pointer)UserIoRequest,
                                                sizeof(IOREQUEST) );
    IOParameters = (UInt8*)kmalloc ( SystemLength, GFP_KERNEL) ;
    copy_from_user( (Pointer)IOParameters,
    (Pointer)KernelIoRequest.InputBuffer, KernelIoRequest.InputLength )

    switch ( cmd )
        {
```

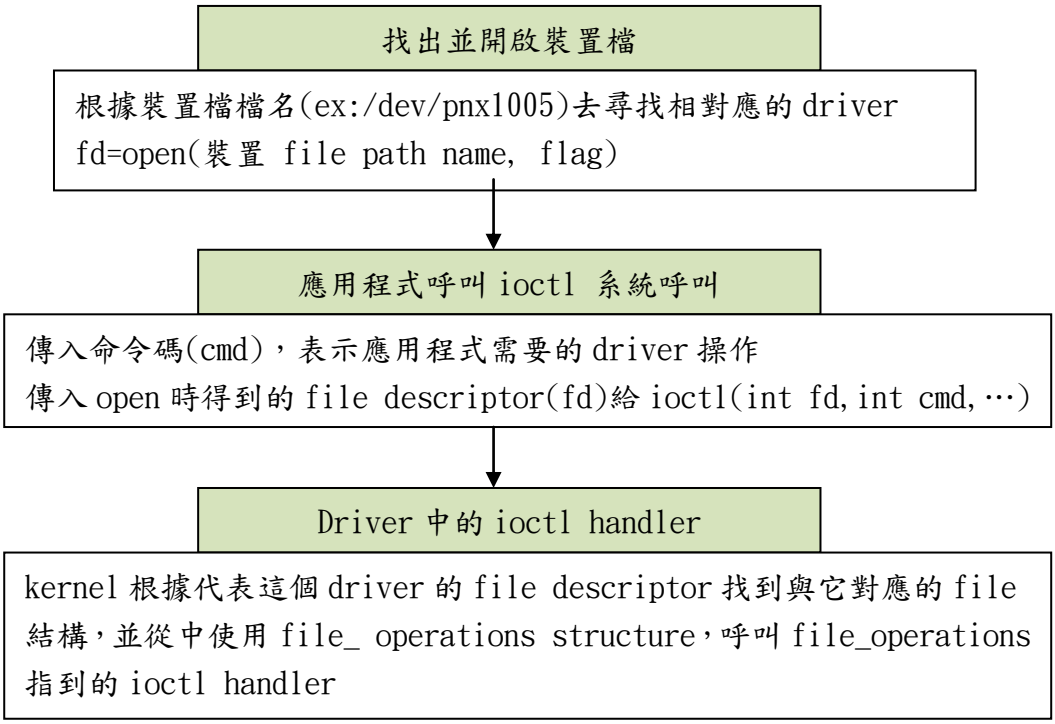
```
        case constIOCTLtmmanDSPOpen :….
        break;
        case constIOCTLtmmanDSPGetNum :….
        break;
        case constIOCTLtmmanDSPInfo :…
        break;
        //……..//
    }

    copy_to_user( (Pointer)KernelIoRequest.OutputBuffer,
    (Pointer)IOParameters,KernelIoRequest.OutputLength );
}
```

我們可以使用 kernel 提供的 copy_from_usr 將 user space 的資料內容複製到 kernel space 去，或是利用 copy_to_usr 將驅動程式處理完的資料內容複製回應用程式所在的 user space。這兩個函式是在 kernel 裡的程式來呼叫的，而使用這兩個函式與 memcpy 主要的不同在於，這兩個函式的內部，都會藉由 kernel 函式 access_ok(type,使用者空間虛擬記憶體位址,要複製的大小)來檢查這個使用者空間的虛擬位址加上欲複製的大小後，是否仍在合法的使用者空間內(有關 copy_to_user/copy_from_user 內部設計請看 Appendix G)。因為在 kernel 中的運作，如果位址不在使用者合法空間或是超出使用者空間大小，亦或是這些虛擬記憶體空間還沒有映射到的實體記憶體空間，kernel 直接做複製動作就會對 kernel 造成影響，沒有問題的話才可以作資料複製的動作。

了解 ioctl handler 設計架構後，我們以 TMMan 驅動程式來認識 ioctl 在系統裡是如何被呼叫的(如圖 3.5)，並說明 ioctl 系統呼叫與 ioctl handler 的函式呼叫(表 3.6)。

當應用程式要求某個驅動程式提供服務時，為了明確指出應用程式需要的驅動程式是哪一個，我們需要利用「裝置檔」將應用程式與驅動程式的關係連接起來。為了找到代表這個驅動程式的裝置檔，我們會先利用 open(裝置檔名, flag)根據裝置檔檔名(ex: /dev/pnx1005)找到描述這個檔案的 file descriptor，有了 file descriptor 後，就代表找到對應的 file 結構，從 file 結構中就可以取得 file_operations 結構儲存的驅動程式 handler 位址(file descriptor 是 OS 分配好用來代表驅動程式的編號，flag 則是使用者對這個 file 的權限)。

因此當我們需要使用驅動程式來控制硬體時，應用程式會呼叫 ioctl 系統呼叫，此時我們必須將開檔得到的驅動程式裝置檔 file descriptor 做為 ioctl 其中一個參數，告知 kernel 這個 ioctl 系統呼叫是要使用到 file descriptor 對應到的 file 結構內容來運作，我們還必須傳入命令碼(cmd)以決定想要執行哪一個驅動程式提供的控制服務。下達 ioctl(int fd,int cmd,…) 這個系統呼叫以後，kernel 利用 file 結構儲存的 file_operations structure，呼叫 file_operations 指到的 ioctl handler，之後 ioctl handler 再根據 cmd 去實作驅動程式實際上對裝置的操作。



(圖 3.5)ioctl 使用流程

(表 3.6) ioctl 系統呼叫與 ioctl handler 的函式呼叫

ioctl 系統呼叫

參數	用途
裝置檔的 file descriptor	判斷控制裝置的驅動程式是哪一個
int command	依照不同的 command 對裝置做不同的控制
可變引數 (可傳入某個結構位址)	應用程式傳給驅動程式所需的資訊或是可以用來讓驅動程式回傳資料。 例如:我們可以傳入一個資料結構位址，這個位址會接著傳給 ioctl handler，在 ioctl handler 裡頭會先使用 copy_from_user 將 user space 位址指到的空間複製到 kernel space 的空間，ioctl handler 再繼續接著處理。驅動程式可以對這個結構內容做修改，然後利用 copy_to_user 將結構內容再複製回應用程式裡。 下面為資料結構範例。 typedef struct tagtmifDSPOpen { OUT TMStatus Status; IN UInt32 DSPNumber; OUT UInt32 DSPHandle;

	}tmifDSPOpen; IN:由應用程式傳入驅動程式的資料 OUT:驅動程式負責填寫的資料

Kernel 將裝置檔內資訊拆成 inode 結構與 file 結構分別傳入 ioctl handler

ioctl handler

參數	用途
inode 結構 struct inode *inode	存有裝置檔 major number/minor number 等資訊。
file 結構 struct file *filep	存有註冊時的 file_operations 結構位址
指令 unsigned int cmd,	由 ioctl 系統呼叫傳下來的 command
可變引數位址 unsigned long arg	由 ioctl 系統呼叫傳下來的可變參數位址

在 TMMan 驅動程式中，實作了大量用來控制 PNX1005 的 ioctl 操作，每個操作都由 command 做為區分，下面整理了第四章提到的溝通機制中，會使用到的 command 及其代表的控制功能。

(表 3.7) TMMan 驅動程式中 ioctl handler 提供的控制功能

command	功能
constIOCTLtmmanNegotiateVersion	判斷驅動程式的版本是否正確，並回傳驅動程式版本內容
constIOCTLtmmanDSPOpen	回傳 kernel 中，用來記錄 PNX1005 的硬體資源與 shared memory 功能區塊位址的資料結構位址，並將結構中的參閱次數加一
constIOCTLtmmanDSPClose	將 constIOCTLtmmanDSPOpen 運作裡提到的資料結構位址減一
constIOCTLtmmanDSPGetNum	取得平台上 OS 偵測到的 PNX1005 個數
constIOCTLtmmanDSPInfo	由 kernel 中，從 constIOCTLtmmanDSPOpen 運作裡提到的資料結構位址裡讀取 MMIO、RAM

	apertures 的基底位址與大小，還有 PNX1005 的裝置 ID、廠商 ID
constIOCTLtmmanDSPLoad constIOCTLtmmanDSPStart constIOCTLtmmanDSPStop constIOCTLtmmanDSPReset	藉由設定 PNX1005 的暫存器，來告知 PNX1005 的 CPU 從 RAM 的哪裡開始運作程式、設定 PNX1005 的 TM32_Start 暫存器以啟動 PNX1005 並將 shared memory 區塊位址記錄到 shared memory 最前面的 220 bytes、停止 PNX1005 的運作、重置 PNX1005 硬體
constIOCTLtmmanDSPGetInternalInfo	得到有關 shared memory 的記憶體大小與動態配置區塊的起始位址
constIOCTLtmmanMessageCreate	為應用層創建訊息溝通管道
constIOCTLtmmanMessageDestroy	釋放訊息溝通管道占用的記憶體空間
constIOCTLtmmanMessageSend	傳送訊息
constIOCTLtmmanMessageReceive	接收訊息
constIOCTLtmmanEventCreate	創建同步管道
constIOCTLtmmanEventDestroy	釋放同步管道占用的記憶體空間
constIOCTLtmmanEventSignal	對另一端做同步，signal 另一端的 semaphore
constIOCTLtmmanSyncObjCreate	創建同步時需要用到的 semaphore 結構並初始為 0
constIOCTLtmmanSyncObjWait	在本地端對 semaphore 進行 Block
constIOCTLtmmanSyncObjSignal	在本地端對 semaphore 做 signal 的動作
constIOCTLtmmanSyncObjDelete	釋放 semaphore 結構空間

mmap handler—將實體記憶體映射到虛擬記憶體的 user space

在 2-3-2 節我們已經有提到過 mmap 的功能，mmap 系統呼叫是由應用程式呼叫的，主要就是從應用程式的使用者虛擬記憶體空間找出一塊還未使用的空間，將實體記憶體某段空間映射到這塊虛擬記憶體去，如此一來，在我們的平台上做 ARM 端應用程式開發時，才可以讓應用程式透過修改這塊被映射到 user space 的記憶體空間，將資料直接寫入 DSP 端的 RAM 或是從 DSP 端的 RAM 讀出資料。

不管是一般檔案裡的檔案或是裝置檔，如果想要使用 mmap 這個系統呼叫，則一定要實作 mmap handler，這個 handler 的位址會儲存在 file_operations 結構中，因此為了讓 kernel 知道現在想要用哪個驅動程式的 mmap handler 做的映射動作，應用程式也需要像

ioctl handler 一樣，使用 open 這個系統呼叫來開啟裝置檔，將 file descriptor 傳入 mmap 這個系統呼叫中。

我們藉由(表 3.7)了解一下 mmap 系統呼叫與 mmap handler 之間的對應關係。使用者呼叫 mmap 系統呼叫時從 user space 來看只知道要用的是哪個裝置檔，還有記憶體映射到 user space 空間的使用權限以及欲映射的記憶體大小，但是並不知道要映射的實體記憶體位址，所以有關實體記憶體的部分必須在 mmap handler 中做好控制。

(表 3.7)mmap 系統呼叫:回傳 user space 的虛擬記憶體位址

參數	意義
caddr_t addr (caddr_t=void*)	指定檔案應被映射到 user space 的起始位址，一般指定為 NULL 讓 kernel 自己去決定要映射到哪個 user space。
size_t len	映射到 user space 的空間大小，以 Byte 為單位。
Int prot	指定使用權限。
Int flags	可決定這塊被映射的實體記憶體區段是否可被其他程式共用或需要使用 copy-on-write 的機制來使用這個區段等。
Int fd	由 open 這個系統呼叫得到的裝置檔 file descriptor，也藉由檔案的 file 結構告知 mmap 要使用哪一個檔案下的 file_operations 儲存的 mmap handler 來做映射的動作。
off_t offset	從被映射檔案的起始點 offset 開始映射，一般為 0。

當使用者呼叫 mmap()這個系統呼叫後，kernel 會進行以下的處理:

1. 在這個使用者程序的虛擬空間找一塊 VMA，因此在 kernel 中就以 vm_start 與 vm_end 來代表這段 size 為 len 的 VMA。
2. 如果裝置驅動程式或者檔案系統的 file_operations 結構裡有儲存 mmap handler 則呼叫它，將欲映射的 physical memory 區塊映射到這塊 VMA。。

mmap handler:回傳 0 的話則代表成功

參數	意義
struct file*	藉由系統呼叫傳入的 file descriptor 找到與之對應的 file 結構。
struct vm_area_struct*	mmap 系統呼叫被呼叫後，kernel 會將找到的 VMA 空間的 vm_start 與 vm_end 設定到 vm_area_struct 去，並將系統呼叫給的參數 flags、prot 分別設定到 vm_area_struct 裡的 vm_flags、vm_page_prot。

在 TMMan 驅動程式中，mmap 的 mmap handler 實作是透過 kernel 另外提供的 remap_pfn_range(表 3.8)映射函式來完成整個實體記憶體與虛擬記憶體之間的映射動作，由於而記憶體基本上都只能以 page 為單位進行映射，如果想要映射的大小沒有剛好是 PAGE_SIZE 的倍數，則必須對該位址進行 page 的對齊，強行以 PAGE_SIZE 的倍數大小來映射，所以它除了利用 mmap 系統呼叫傳給 mmap handler 的參數資訊，還需要先獲知欲映射的實體記憶體位址，並利用「記憶體起始位址>> PAGE_SHIFT」公式求出頁碼(page number)。

但是因為 PNX1005 memory apertures 不只一個，而且在實體記憶體中占據的空間也不連續，所以在裝置檔的 file 結構中，我們使用了 MapState 變數來儲存要映射哪種 apertures 的資訊,於是 file 結構中便有可能出現 RAM aperture、MMIO aperture 與 shared memory 的三種 state 代號，針對不同的代號，mmap handler 中也會利用 switch 做 state 代號的判別，來取得各個位址空間的起始位址與空間大小，並從傳入 mmap handler 的 vm_area_struct 結構取得系統分配的 VMA 起始與結束位址，然後再用 remap_pfn_range 新建實體記憶體與 user space 之間的頁表，完成映射的動作。

mmap 系統呼叫會在應用程式需要的時候被使用，為了程式運作的方便，通常會在應用程式最開始就做映射的動作，而且因為在一個應用環境中只需要映射一次就夠了，所以這三種映射會依照順序來映射，當 SDRAM aperture 映射完後，就把 file 結構裡的 MapState 改成 MMIO aperture 的 state 代號，下回應用程式會繼續使用 mmap 系統呼叫要求映射，便知道這次是要對 MMIO aperture 做映射，以此類推(表 3.9)。

由於我們希望這個被映射的區段不能被 cache，這樣 CPU 才能確定讀寫到的資料都有在真正的實體記憶體中，因此我們在 remap_pfn_range 以前須要先利用 pgprot_noncached (vm_page_prot)將這段映射區段設為 nocached，之後才使用 remap_pfn_range 將 DRAM aperture 映射到 user space 去。

(表 3.8)remap_pfn_range:成功則回傳 0

參數	意義
struct vma_area_struct *vma	Kernel 找到的 VMA 區塊在系統中的描述結構，把實體記憶體區塊映射到 VMA 描述的範圍裡。
Unsigned long addr	記憶體映射開始處的虛擬位址，通常為 vm_start。
Unsigned long pfn	欲映射的實體記憶體位址所處的 page number。實際上就是實體記憶體位址向右 shift PAGE_SHIFT 位元，而 2^PAGE_SHIFT 其實就等於

	PAGE_SIZE。
Unsigned long size	欲映射的記憶體大小，即 vm_end-vm_start。
pgprot_t prot	新 page 要求的保護權限。

(表 3.9)TMMan 驅動程式的 mmap handler

```
int trimedia_mmap(struct file *filp,struct vm_area_struct *vma)
{
     ClientObject* Client;
     UInt32        PageOffset = vma->vm_pgoff << PAGE_SHIFT; // will probably be 0 in this case
     UInt32             VirtualSize;
     UInt32             PhysicalSize;
     UInt32             HalHandle;
     UInt32             BlockPhysicalAddress;
     UInt32             BlockKernelAddress;
     UInt32             BlockSize;
     UInt32             PhysicalAddress;
     UInt32             MapState;
     UInt32             DSPNumber;
     HalObject*         Hal;
     TMManDeviceObject*      TMManDevice;

     VirtualSize = (UInt32)(vma->vm_end - vma->vm_start);

     Client = (ClientObject*)filp->private_data;

     MapState  = Client->MapState;
     DSPNumber = Client->DSPNumber;

     TMManDevice = (TMManDeviceObject *)(TMManGlobal->DeviceList[Client->DSPNumber]);

HalHandle = ((TMManDeviceObject *)(TMManGlobal->DeviceList[Client->DSPNumber]))->HalHandle;
     Hal = (HalObject*)HalHandle;

     // look at our state machine to determine what to map
     switch ( Client->MapState ) {
     case constTMMan_MEMORY_BLOCK_SDRAM :
            halGetSDRAMInfo (
              HalHandle,
              (Pointer*)&BlockPhysicalAddress, // this is TMs  view of PhysicalAddress
              (Pointer*)&BlockKernelAddress,
              (UInt32*)&BlockSize);
             MapState = constTMMan_MEMORY_BLOCK_MMIO;
         break;

     case constTMMan_MEMORY_BLOCK_MMIO :
          halGetMMIOInfo(HalHandle,(Pointer*)&BlockPhysicalAddress,
                                   (Pointer*)&BlockKernelAddress,
                                   (UInt32*)&BlockSize);
          MapState = constTMMan_MEMORY_BLOCK_SHARED;
     break;
```

取得 RAM apertures 起始位址與大小

取得 MMIO apertures 起始位址與大小

```
    case constTMMan_MEMORY_BLOCK_SHARED :
        /*
         * TMManDevice->MemoryBlockAddress.LowPart is a Bus address, to do mmap,
         * need to have cpu i.e pa / physical address, we can subtract with BusOffset
         */
        BlockPhysicalAddress =  TMManDevice->MemoryBlockAddress.LowPart -
        TMManDevice->TMManShmAddr.BusOffset;
        BlockSize = TMManDevice->MemoryBlockSize;
        MapState = constTMMan_MEMORY_BLOCK_SDRAM;
        Client->DSPNumber++;
        break;

    PhysicalAddress = BlockPhysicalAddress + PageOffset;
    PhysicalSize   = BlockSize - PageOffset;

    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

    remap_pfn_range( vma, vma->vm_start, PhysicalAddress >> PAGE_SHIFT, VirtualSize,
                                                        vma->vm_page_prot)

    trimedia_mmap_exit:
    Client->MapState = MapState; // update MapState
    Client->DSPNumber = DSPNumber;//update DSP Nubmer
    return Status;
}
```

取得欲映射的 shared memory 起始位址與大小

3-2 一般驅動程式的啟動

設計好驅動程式控制裝置的運作實體後，要讓 kernel 認得這個驅動程式，最重要的是將這個驅動程式註冊到 kernel 中，並且能利用驅動程式對裝置做基本的初始，像 TMMan 驅動程式就會利用 pci_find_device 函式去掃描版子上所有的 PNX1005，並將 PNX1005 PCI 配置空間的資訊儲存到驅動程式內特定的結構之中，供之後應用程式呼叫系統呼叫時使用(3-3 節)。

驅動程式跟一般的應用程式相似，要開始運行程式時都需要有一個程式進入點，但是應用程式的進入點為 main 函式，驅動程式的程式進入點則為 init_module()，當我們在 kernel 下達 insomd 指令後，便會進入 init_module()來載入並註冊驅動程式到 kernel。在 TMMan 驅動程式的初始階段，還會做硬體資源分配(包括共享記憶體的配置與劃分)、事先存取硬體資訊(PNX1005 memory apertures 的起始位址、IRQ 編號等)，以及設定管理硬體資源的資料結構等工作(第 3-3 節)。

在 linux 環境中，進入 init _module ()後第一項重要的工作就是呼叫 register_chrdev(unsigned int major, const char * name, struct file_operations *fops)，用來註冊字元型驅動程式給 kernel。註冊時應用程式必須提供的三個參數如下:

Major: 每個驅動程式都有一個 unique major number 以便管理裝置檔與對應的驅動程式，可以由 kernel 自動分配或是直接指定。

Name:驅動程式名稱

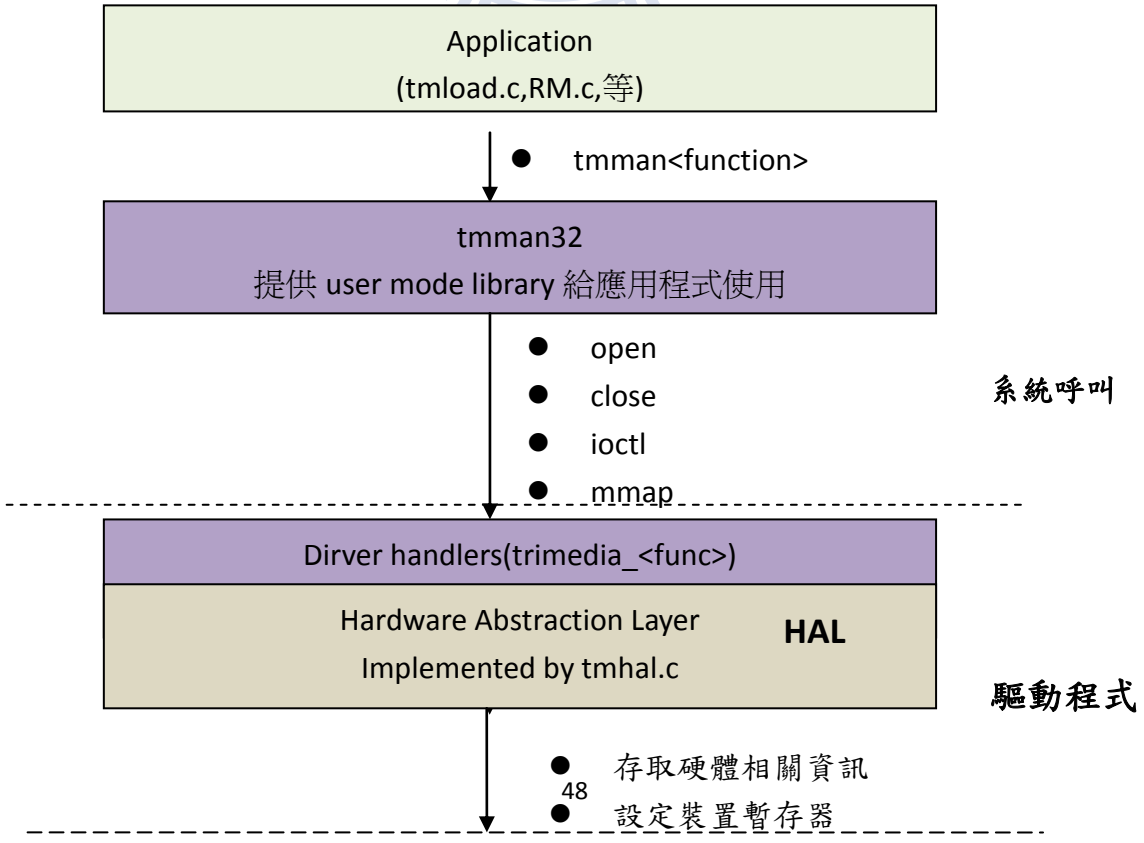
Fops:事先設計好的 file_operations 結構

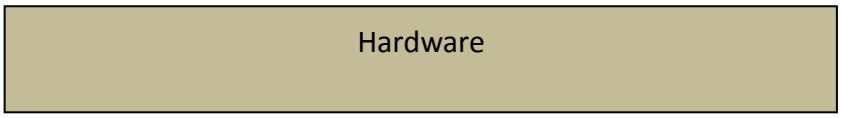
由於每個硬體都有其對應的驅動程式來控制，系統中會有多個驅動程式，所以每個驅動程式都必須要有一個獨一無二的 major number，而 major number 可以是 OS 自動分配的，也可以是自己制定的。在「註冊」--呼叫 register_chrdev 時，驅動程式在載入的同時會將 major number 以及該驅動程式對應的 file_operations 結構註冊到 kernel，所以即使系統中有很多不同的驅動程式與 file_operations 結構，但在註冊時就已經將他們一對一的關係建好了，而且也以不同的 major number 來代表不同的驅動程式/裝置，而 insmod 完後，在/proc/devices 這個檔案裡，也會列出使用 register_chrdev 註冊時，該驅動程式分配到的 major number 與驅動程式名稱。

當應用程式要求某個驅動程式提供服務時，為了明確指出應用程式需要的驅動程式是哪一個，所以我們需要利用「裝置檔」將應用程式與驅動程式的關係連接起來。於是在 insmod 註冊完驅動程式後，還要使用 mknod 指令來建立裝置檔以提供之後程式運作使用，而 mknod 建立裝置檔時所需的 major number 則可利用/proc/devices 列出的資料取得，建好的裝置檔則會置於/dev 這個目錄下。因為在執行 mknod 時必須提供「裝置檔名」以及之前註冊驅動程式時取得的 major number，此時裝置檔就會和對應的驅動程式做好了連接。

之後當應用程式想要使用某個驅動程式，就要先利用裝置檔名去找到該裝置檔並開啟它，得到這個裝置檔的 file descriptor 後，才能告知 ioctl、read、write 等系統呼叫應用程式需要的驅動程式是哪一個，由它來處理硬體操作。

3-3 TMMan 驅動程式





(圖 3.6)TMMan 驅動程式設計架構

TMMan 驅動程式有兩部分，一部分為 kernel mode 驅動程式，主要由 driver handlers 來負責驅動程式的運作，並由 init_module()函式來初始化驅動程式控制硬體時所需資料，這些是基於 linux 驅動程式架構組成，提供控制與設定硬體暫存器的介面、註冊程序、硬體資源配置、硬體資源存取。有了 kernel mode 驅動程式之後，另一部分則為 user mode library，我們又依照不同的使用者功能需求，將多個不同命令碼(不同控制操作)的 ioctl handler，組裝成應用程式可以使用的功能，使用者只要知道如何使用功能函式庫(user mode library)裡面提供的人性化呼叫介面，就可以使用 PNX1005 這個裝置(第 3-4 節)。

Driver handlers

kernel mode 驅動程式很重要的是依照每個裝置可以提供的服務實作出必要的驅動程式 handlers，並將它們的位址一一指定到 file_operations 結構中，TMMan 驅動程式主要提供了 trimedia_open、trimedia_release、trimedia_mmap、trimedia_ioctl 這幾個驅動程式 handlers。

Hardware Arbstraction Layer

HAL 是驅動程式中，用來負責控制裝置動作或提供控制裝置所需資訊的一組介面，會由許多函式組成，以 TMMan 驅動程式來說，就包括讀取資料用的 halGetSDRAMInfo、halGetMMIOInfo 函式以及使用裝置暫存器時會用到的 halStartDSP、halCloseDSP、halReset、halGenerateInterrupt 函式，讓驅動程式可以透過它存取硬體相關資訊，然後再藉由驅動程式內部變數回傳這些資料給應用程式，或讓驅動程式對裝置進行設定與操作。

原本隨著裝置的不同，驅動程式裡有關裝置的操作就要跟著修改，為了減少繁複的更動，驅動程式可以設計 Hardware Abstraction Layer 所需函式介面，針對不同的裝置，對該裝置提供的暫存器做設定，或是在不同的 OS 環境中，進行 ISR 的註冊動作。這些 HAL 的函式具有可攜性，如果不同的裝置提供的功能類似，就不需要因為裝置的不同而需要對驅動程式做大幅運做修改，到了不同的 OS 環境，只須在 HAL 內部做函式的修改即可，所以驅動程式可以使用同樣的 HAL 函式去控制裝置。

在 TMMan 驅動程式中，HAL 最重要的就是負責所有有關 PNX1005 硬體暫存器的設定、註冊接收到硬體中斷後馬上要執行的 HAL ISR 給 kernel，然而為了設定 PNX1005 的暫存器，

HAL 事先還必須取得 PNX1005 的 MMIO aperture 實體記憶體位址，然後使用 ioremap_noncache 函式(2-2-1)，將 MMIO aperture 映射到虛擬記憶體的 kernel space 中，並儲存映射到的 kernel address，以利之後 HAL 在 kernel 中藉由 kernel address 對 PNX1005 暫存器進行設定。牽涉到設定硬體暫存器的運作包括啟動 DSP(halStartDSP)、關閉 DSP(halCloseDSP)、重置 DSP(halReset)、對 DSP 產生中斷(halGenerateInterrupt)，註冊 HAL ISR 的動作則是在設計完 HAL ISR(halHardwareInterruptHandler)後，用 kernel 提供的 request_irq(2-6 節)來註冊。

```
request_irq((int)Hal->InterruptVector,halHardwareInterruptHandler,
            SA_INTERRUPT | SA_SHIRQ,” pnx1005”,(void *)Hal);
```

在啟動 PNX1005 前，它還會將 shared memory 各區塊的起始位址寫到 shared memory 最前面的 220 bytes，讓 DSP 端可以藉由這塊資訊得到 ARM 端規劃好的區塊空間。

另外，為了讓 HAL 層儲存控制裝置所需資訊，在開機載入 TMMan 驅動程式時，每次只要掃描到板子上的 PNX1005，就會於 HAL 先儲存這個 PNX1005 的 memory aperture 實體記憶體位址/kernel space 虛擬記憶體位址與空間大小、PNX1005 的 IRQ 編號、ARM 端中斷腳位等 PCI 配置空間(pci_dev 結構)資訊，如此一來，在 ioctl handler 中便可利用 HAL 函式(halGetSDRAMInfo、halGetMMIOInfo)，讀取使用 PNX1005 硬體資源時所需的硬體與記憶體資訊。

TMMan 驅動程式中 HAL 與 driver handler 的關係

從上一段我們了解了 TMMan 驅動程式中 HAL 的功能，而這些功能則會被驅動程式中的 driver handler 來調用。本來驅動程式的大觀念就是用來控制硬體的，只是現在內部多分了一層 HAL，把真正接觸到與硬體相關的暫存器或硬體資源等資訊由 HAL 來做，driver handler 就只是利用 HAL 提供的資訊來完成功能並回傳給應用層的一個軟體，或是一個用來判斷要做哪種控制功能的邏輯操作，而真正完成此控制功能、對硬體操作的則是 HAL。

以第四章在討論訊息傳送的動作為例，TMMan 驅動程式會以 command 為 constIOCTLtmmanMessageSend 的 ioctl handler 來做處理，它會將訊息寫入 shared memory 中，然後對另一端發出中斷，但是寫入 shared memory 的動作因為還不需要設定 PNX1005 的暫存器，所以 ioctl handler 就利用 memcpy 作記憶體複製，而後因為要對 PNX1005 發出中斷，此時就會用到 HAL，先取出儲存於 HAL 的 PNX1005 MMIO aperture 虛擬記憶體起始位址，然後加上特定的位移找到 PNX1005 的 IPENDING 這個暫存器，對其進行設定以完成中斷的發出。

為了讓 TMMan 驅動程式可以使用 HAL 做各種 PNX1005 的控制動作，在 TMMan 驅動程式

進行初始化動作時，必須把控制 PNX1005 所需資訊儲存於 HAL 結構中(表 3.10)。之後，ARM 端應用程式不管是要取得硬體資訊、與 DSP 端溝通還是啟動 PNX1005，都會透過 ioctl 這個系統呼叫進入 kernel mode 驅動程式中的 trimedia_ioctl 中，最後都會利用到 HAL 函式來回傳硬體資訊還有設定暫存器等控制動作；而應用程式使用 mmap 系統呼叫後，在 kernel 中執行 trimedia_mmap 時，也會先使用到 HAL 函式來讀取 PNX1005 memory apertures 的實體記憶體起始位址與大小，然後才進行映射的動作。

(表 3.10)HAL 儲存控制 PNX1005 時所需資料

```
typedef struct tagHalObject
{
    GenericObject       Object;
    Bool                IsBridge;

    UInt32        TMDeviceVendorID;
    UInt32        TMSubsystemID;
    UInt32        TMClassRevisionID;

    UInt32        BridgeDeviceVendorID;
    UInt32        BridgeSubsystemID;
    UInt32        BridgeClassRevisionID;

    UInt32      SelfInterrupt;  // from Target -> Host
    UInt32      PeerInterrupt;  // Host -> Target

    HalInterruptHandler Handler;
    Pointer             Context;

    HalControl*         Control;//兩端 CPU 進行溝通時會使用

    /*  true -  target processor is running in different indianess */
    /*  false -  target processor is running in different indianess */
    Bool                Swapping;

    UInt32              PeerMajorVersion;
    UInt32              PeerMinorVersion;

    /* BEGIN Platform Specific */

    UInt32              BusNumber;
    UInt8               SlotNumber;

    PHYSICAL_ADDRESS    MMIOAddrPhysicalBridge;
    PHYSICAL_ADDRESS    MMIOAddrPhysical;
    UInt32              MMIOLength;
    UInt8*              MMIOAddrKernel;

    PHYSICAL_ADDRESS    SDRAMAddrPhysicalBridge;
    PHYSICAL_ADDRESS    SDRAMAddrPhysical;
    UInt32              SDRAMLength;
    UInt8*              SDRAMAddrKernel;

    PHYSICAL_ADDRESS    XIOAddrPhysical;
```

設定中斷暫存器時會使用到

ISR 函式位址與 ISR 所需參數位址

控制 MMIO aperture 暫存器時所需的 MMIO 兩種起始位址與空間大小

使用 SDRAM aperture 所需的兩種起始位址與空間大小

```
    UInt32              XIOLength;
    UInt8*              XIOAddrKernel;
    UInt32              InterruptVector; //中斷向量編號

    UInt32              TriMediaAccessAddress;

    Bool                FirstTimeReset;

    UInt32              DSPNumber;

    UInt32              SDRAMMapCount;
    UInt32              Offset;

    UInt32              TM32Apert1Lo;
    UInt32              TM32Apert1Hi;
    UInt32              TM32DramOffset;
    UInt32              TM32DramLen;
    UInt32              TM32DramCLimitOffset;
    UInt32              TM32StartOffset;
    UInt32              TM32StartAddress;

//儲存初始時的 PCI 裝置的配置空間的 header 上每個暫存器的初始設定，供重置 DSP 時使用
    UInt32              PCIRegisters[constTMMANPCIRegisters];
//

    Pointer             TMManDevice;//指向(表 3.12)的結構


    //儲存每次利用 pci_get_device 偵測到環境中的 PCI 裝置後，就會回傳的 pci_dev
      結構位址
    struct pci_dev*     PciDevice;

}   HalObject;
```

設定 DSP 端啟動後從 DSP 端 RAM 的哪個位址開始運作，以及設定 PNX1005 可以使用的記憶體範圍

Init_module—TMMan 驅動程式的初始化

我們為驅動程式準備一個程式進入點—init_module()，當使用者於 console 界面呼叫 insmod 指令要求註冊驅動程式時，便開始自進入點做實質的註冊動作--register_chrdev，把驅動程式跟 file_operations 結構做好連接。

```
int init_module( void)
{   //註冊名為" pnx1005" 的驅動程式給 kernel 時，因為 DEVICENUMBER=0，所以 kernel
    會自動分配 major number，並將已經存有各個 trimedia driver handler 位址的
    trimedia_fops 這個 file_operations 資料結構與這個驅動程式連接好//
    result = register_chrdev(DEVICENUMBER, MODULENAME, &trimedia_fops);


    /*對環境中的 PCI 裝置做偵測，並使用 2-7-7(表 3.6)的函式，將偵測到的 PCI 裝置
    它的 PCI 配置空間資訊讀取出來，儲存於 HAL 的結構中*/
}
```

然而 PNX1005 是一個 PCI 裝置，我們還需要讓 ARM 與 PNX1005 可以做訊息的交換，所以 init_module()除了做註冊的動作外，還會去偵測環境中有哪些 PCI 裝置，然後讀取出 PCI 裝置提供以用來控制它的硬體資訊(表 3.10)，另外 ARM 端也會在自己的 RAM 裡面配置出一塊共享記憶體，供之後做訊息交換、同步使用。下面會介紹 TMMan 驅動程式中用來儲存硬體資訊以及初始時所需設定的資料結構，而後會說明偵測 PCI 裝置的方式，至於共享記憶體的劃分，在此只提出有哪些資料結構儲存了劃分後的共享記憶體區塊起始位址，但共享記憶體的劃分過程請看第四章。

為了儲存驅動程式運作時所需資訊而使用到的結構

在 init_module 做驅動程式的初始時，kernel 裡的驅動程式會準備一個 global structure(表 3.11)來存取驅動程式開發者事先設定好、用在 ARM 與 PNX1005 作訊息交換時所需的固定硬體資源個數與大小的限制，像是共享記憶體可容納的 channel 個數、每個 channel 可容納的 mailbox 個數、NameSpace 區塊個數、Vintr 區塊個數、Event 區塊個數、驅動程式的 major number，而這個 global structure 裡面還有一個陣列，其中的每個 element 內容(表 3.12)則用來存放在 init_module()時，kernel 利用 PCI 設備清單查找出 PNX1005 後，linux 看到的每一個 PNX1005 PCI 配置空間(pci_dev 結構)的資料結構位址、驅動程式配置出的共享記憶體實體記憶體位址與 kernel space 的虛擬記憶體位址及儲存控制 PNX1005 所需硬體資訊的 HAL 結構位址等。

(表 3.11) GlobalObject 結構

用來儲存每個 PNX1005 的 TMManDeviceObject 結構

```
typedef struct  tagGlobalObject
{
    UInt32          MaximumDevices;
    UInt32          DeviceCount;
    Pointer         DeviceList[constTMMANMaximumDeviceCount];

    UInt32      CurrentDevice;
    UInt32          MaximumClients;
    ..............
    /*driver 開發者事先設定好、用在 ARM 與 PNX1005 作訊息交換時所需的 constant 硬
        體資源個數與每個區塊大小*/
} GlobalObject;
```

(表 3.12)TMManDeviceObject 結構

```
typedef struct tagTMManDeviceObject
{
    UInt32  DSPNumber; //這個 TMManDeviceObject 是給哪一個 PNX1005 使用的

    TMManSharedStruct* SharedData; //shared memroy 的 kernel space 虛擬記憶體起
                                                                始位址
    UInt32              SharedDataSize;
```

```
UInt32  HalHandle;                  ──→ 指向(表10)結構以儲存硬體資源與 kernel 位址

UInt32  ChannelManagerHandle;
UInt32  VIntrManagerHandle;
UInt32  EventManagerHandle;
UInt32  MessageManagerHandle;       ──→ 用來管理 shared memory 的總管理元件的 kernel 虛擬記憶體位址
UInt32  MemoryManagerHandle;
UInt32  NameSpaceManagerHandle;

Pointer HalSharedData;
Pointer MemorySharedData;
Pointer NameSpaceSharedData;        ──→ shared memory 中各區塊的 kernel 虛擬記憶體起始位址
UInt32  ChannelSharedData;
UInt32  VIntrSharedData;
UInt32  EventSharedData;

PHYSICAL_ADDRESS    HalSharedAddress;
PHYSICAL_ADDRESS    EventSharedAddress;
PHYSICAL_ADDRESS    ChannelSharedAddress;
PHYSICAL_ADDRESS    VIntrSharedAddress;
PHYSICAL_ADDRESS    DebugSharedAddress;
PHYSICAL_ADDRESS    MemorySharedAddress;    ──→ 存放各種 shared memory 上控制結構的實體記憶體位址
PHYSICAL_ADDRESS    MemoryBlockAddress;
PHYSICAL_ADDRESS    NameSpaceSharedAddress;

PHYSICAL_ADDRESS TMManSharedAddress; //shared memory 的實體記憶體起始位址
}TMManDeviceObject;
```

由於這個 global structure 在整個驅動程式裡面是一個全域變數，等驅動程式的註冊程序結束後，凡是從應用程式呼叫的系統呼叫到了 kernel，file_operations 結構裡儲存的驅動程式 handler 都可以直接調用 global structure 裡的內容。

設定驅動程式運作所需結構內容

為了填入global structure裡該PNX1005的陣列內容,有兩件事要做,在 init_module()裡，首先必須對板子上的 PNX1005 做查找的動作，從找到的裝置裡讀出 PCI 配置空間裡的硬體資源資訊；其次是針對這個 PNX1005 做共享記憶體的配置及管理，然後把共享記憶體的位址資訊及對共享記憶體劃分完後的區塊實體記憶體位址(共享記憶體的劃分請看第四章)儲存到陣列中。

在 init_module 裡是利用 pci_find_device(在 kernel 2.6.22.0 以上的系統則用 pci_get_device)這個 kernel 提供的函式做查找的動作，依據驅動程式開發者事先給定的 PCI list(表 3.13)去掃描 kernel 裡的 PCI 設備清單(cat /proc/bus/pci/devices)，而所謂的設備清單，其實就是 kernel 啟動時，由 kernel 裡負責做 PCI 初始化的 PCI 裝置驅動程式,利用 pci_dev(2-7-6 的(表 2.6))建立起的 PCI 拓樸。這個 kernel 提供的查找函式，

於是會去比對設備清單中哪一個 pci_dev 與 list 裡面的 vendor ID 與裝置 ID 謀合，如果有比對到，則回傳 pci_dev 結構給驅動程式(式 1)。之後驅動程式便可以從這個比對到的 pci_dev 結構中，讀取控制 PNX1005 所需的 PCI 配置空間提供得硬體資源資訊，如 RAM aperture、MMIO aperture 起始位址、中斷向量編號等，將 pci_dev 結構與這些硬體資源資訊都存到(表 3.10)的 HAL 結構去，之後驅動程式利用 HAL 函式對裝置進行控制或讀取硬體資訊時，就會用到這個結構裡儲存的內容。

(表 3.13)DVR 平台使用到的 PCI list

```
TriMediaPCIID   TriMediaPCIIDList[] = {
    { constTMMANDECBridgeVendorID, constTMMANDECBridgeDeviceID, TRUE,TRUE },
    { constTMMANIntelBridgeVendorID, constTMMANIntelBridgeDeviceID,TRUE, TRUE},
    { constTMMANTM1005VendorID, constTMMANTM1005DeviceID1, TRUE, FALSE },
    { constTMMANTM1005VendorID, constTMMANTM1005DeviceID2, TRUE, FALSE },
    { constTMMANTM1005VendorID, constTMMANTM1005DeviceID3, TRUE, FALSE },
    { constTMMANTM1005VendorID, constTMMANTM1005DeviceID4, TRUE, FALSE }
};                  1131                        540b

Kernel 中的設備清單:(列出我們使用的 PNX1005 的 PCI 配置空間資訊)
0100  1131540b    9   c0000008   d8000000  d0000000   0   0
```

(式 1)將 PCI list 一一傳入 pci_get_device 比對

```
PciDevice = pci_get_device
(TriMediaPCIIDList[TypeIdx].Vendor,TriMediaPCIIDList[TypeIdx].Device,PciDevice)
```

查找完 PCI 裝置並讀取完 PCI 配置空間的資訊後，初始化程序還會利用全域結構中「在 ARM 與 PNX1005 作訊息交換時所需的固定硬體資源個數」再為該 PNX1005 配置足夠的共享記憶體空間，並依照第四章所探討的功能來劃分共享記憶體的區塊，然後再把這些區塊的實體記憶體位址都儲存到該 PNX1005 的陣列結構中，之後應用程式需要使用共享記憶體進行溝通時，驅動程式就可以從這個結構中找到共享記憶體中各區塊位址再加以使用。

驅動程式使用初始時填入的資料內容

完成整個 init_module()後，已經將之後要使用到的硬體資源(含共享記憶體)都配置、讀取好，並儲存各項環境資源於驅動程式裡的一個廣域資料結構中，只要驅動程式沒有被卸除就會一直存在。如果要對 PNX1005 進行操作，應用程式便可利用 ioctl 系統呼叫，進入 kernel 的 ioctl handler，ioctl handler 便會到這個廣域資料結構中取得 DSP 端各個 apertures 位址等硬體資訊，然後應用程式再利用這些硬體資訊，使用 mmap 或其他不同命令碼的 ioctl 系統呼叫，完成記憶體映射，ioctl handler 也可以從廣域資料結構的 HAL 結構中取出 MMIO apertures 的 kernel address，對裝置進行暫存器設定；如果是要進行 ARM 與 DSP 端的溝通，也可以使用 ioctl 系統呼叫，透過 ioctl handler 在 kernel 中，使

用初始時儲存於(表 3.12)中的各區塊 ManagerObject 位址，藉由 ManagerObject 裡儲存的資料來建立溝通管道、使用溝通管道(請看第四章)。

3-4 user mode library

有了 TMMan 驅動程式提供基本的控制操作後，應用程式在開發時就可以利用基本的系統呼叫來使用 TMMan 驅動程式，但是利用系統呼叫使用到 kernel 中的驅動程式，只負責對硬體做操作或讀取資訊的動作，如果在運作時發生錯誤，我們希望可以將發生錯誤的原因，給予不同的錯誤代碼回傳給應用程式以告知程式開發者，而且為了讓應用程式設計上更為精簡,所以我們在應用程式與驅動程式之間設計了功能化界面,它會從使用者的角度，來設計需要的功能，而這些功能可能就需要多個 TMMan 驅動程式的基本控制操作來完成。user mode library 提供的功能化函式庫，主要有下列幾個函式最常用到(表 3.14)。

(表 3.14)常用的 user mode library

tmman<function>	驅動程式內部實作
將 DSP 端可執行檔下載到 DSP 的 RAM 時所用	
Bool tmmanMapDeviceMemory (PTMMAN_DEVICE_INFO DeviceMap) *由多個 ioctl 系統呼叫組成，每個功能的 case 如下: constIOCTLtmmanDSPOpen, constIOCTLtmmanDSPInfo, constIOCTLtmmanDSPGetInternalInfo ,最後再利用 mmap 系統呼叫	利用 ioctl handler，從 kernel 中 globalstructure 裡的陣列讀出 PNX1005 的 RAM apertures、MMIO apertures、shared memory 的實體記憶體基底位址及 apertures 的大小，然後將得到的大小傳給 mmap 系統呼叫，把 PNX1005 的 memory apertures 從 physical memory 映射到 user space，並回傳映射的 user space 虛擬記憶體起始位址，儲存到 global structure 裡的 DeviceMap 結構中。
Bool tmmanUnmapDeviceMemory (PTMMAN_DEVICE_INFO DeviceMap)	利用 unmmap 這個系統呼叫，利用 tmmanMapDeviceMemory 時儲存到 DeviceMap 結構中的各種映射到虛擬記憶體位址與大小，將實體記憶體空間映射到虛擬記憶體之間的映射關係取消。
Bool **tmmanInitialize**(void)	用來初始化 user mode library 的執行環境:對之後運用 user library 時所需的廣域結構內容(Appendex E)作初始設定，它會去開啟 PNX1005 裝置檔並將回傳的 file descriptor 儲存於廣域

	結構中，作為其它 user library 中使用 ioctl 實作時辨別環境中的驅動程式之用。另外還會確認 driver 與 user library 的版本是否相容，最後使用 tmmanMapDeviceMemory 函式作記憶體映射的動作，讓應用程式之後可以在 user space 使用 PNX1005 的記憶體與 MMIO aperture 空間。
TMStatus tmmanDSPGetInfo (UInt32 DSPHandle, tmmanDSPInfo *DSPInfo)	利用 command 為 constIOCTLtmmanDSPInfo 的 ioctl 系統呼叫到 kernel，從 global structure 裡的陣列結構讀出 PNX1005 的 RAM apertures、MMIO apertures 的實體記憶體基底位址及 apertures 的大小。另外還回傳 tmmanInitialize 時作完 tmmanMapDeviceMemory 後得到的 RAM apertures、MMIO apertures 被映射到 user space 後的虛擬記憶體位址。
TMStatus tmmanDSPOpen (UInt32 DSPNumber, UInt32 *DSPHandlePointer)	利用 command 為 constIOCTLtmmanDSPOpen 的 ioctl 系統呼叫進入 kernel，從 global structure 裡儲存的陣列，根據 DSPNumber 取得該 PNX1005 對應的陣列內容，每參照一次就將該陣列內容中記錄參照次數的變數加一，最後會回傳該陣列的 kernel address 給 user space。
TMStatus tmmanDSPClose (UInt32 DSPNumber, UInt32 *DSPHandlePointer)	利用 command 為 constIOCTLtmmanDSPClose 的 ioctl 系統呼叫到 kernel 裡，將 DSPNumber 指到的陣列結構內容中的記錄參照次數減一。
TMStatus tmmanDSPLoad (UInt32 DSPHandle, UInt32 LoadAddress, UInt8* ImagePath)	內部利用 ioctl 這個系統呼叫，先將 PNX1005 啟動時開始運作的位址儲存 HAL 裡的結構中。之後做 tmmanDSPStart 時才會從 HAL 結構中再把這個位址設定到 PNX1005 的 TM32_Start_ADR 這個暫存器，讓 PNX1005 知道當它被啟動時要從哪個位址開始運作。
TMStatus tmmanDSPStart (UInt32 DSPHandle)	利用 command 為 constIOCTLtmmanDSPStart 的 ioctl 系統呼叫到 kernel 裡，去設定啟動 PNX1005 需要設定的暫存器，以啟動 PNX1005 並

	開始做 DSP 端可執行檔的程式內容。
TMStatus tmmanDSPStop (UInt32 DSPHandle)	利用 command 為 constIOCTLtmmanDSPStop 的 ioctl 系統呼叫 到 kernel 裡去設定 PNX1005 的暫存器，停止 PNX1005 的運作。
傳送訊息的函式庫	
TMStatus tmmanMessageCreate(UInt32 DSPHandle, UInt8* Name, UInt32 SynchronizationHandle, UInt32 SynchronizationFlags, UInt32 *MessageHandlePointer)	利用 command 為 constIOCTLtmmanMessageCreate 的 ioctl 系統呼叫進入 kernel，創建 queue buffer 以接收訊息，並設置 SynchronizationHandle (semaphore 結構或其他 signal function 位址)，用來作等待接收的動作，並在接收到訊息後去 signal。 *MessageHandlePointer 會指向 kernel 中，用來描述 Message 管道的結構位址。
tmmanMessageReceive (UInt32 MessageHandle, void *DataPointer)	藉由 kernel 中描述 Message 管道的結構位址內儲存的 queue buffer 位址，從 queue buffer 中複製 packet 內容給應用程式的 DataPointer 指標接收。 *MessageHandle 就是 tmmanMessageCreate 時得到的 MessageHandlePointer
tmmanMessageSend (UInt32 MessageHandle, void *DataPointer)	將 DataPointer 指向的 packet 空間複製到共享記憶體的 channel 區塊中，藉此傳送給另一端的 CPU。
tmmanMessageDestroy (UInt32 MessageHandle)	釋放 queue buffer 等在 tmmanMessageCreate 時所占用的記憶體空間。
為兩端做同步的函式庫	
tmmanEventCreate(UInt32 DSPHandle, UInt8* Name, UInt32 SynchronizationHandle, UInt32 SynchronizationFlags, UInt32 *EventHandlePointer)	設定用來做同步的 OS 同步物件(例如 semaphore 結構)，其中的同步物件在 linux 中會由 tmmanSyncObjCreate 先創建好並初始化，而 pSos 則使用 Appsem_create+Appsem_p。 *EventHandlePointer 會指向同步物件位址
tmmanEventSignal	去 signal 另一端 CPU 的某個同步物件，讓一端

(UInt32 EventHandle)	CPU 可以 unblock 繼續進行下面的程式。
tmmanEventDestroy (UInt32 EventHandle)	釋放掉所有 tmmanEventCreate 時所創物件占有的記憶體空間。
依照本地端 OS 進行同步物件操作的函式庫	
tmmanSyncObjCreate (UInt32 *handle)	由 linux 環境中的創建用來同步的 OS 物件(*handle 為 semaphore 結構位址)。
tmmanSyncObjWait (UInt32 handle)	在 linux 環境中使用 down_interruptible 函式來 block 程序的進行
tmmanSyncObjSignal (UInt32 handle)	在 linux 環境中使用 up 函式來 unblock
tmmanSyncObjDelete (UInt32 handle)	釋放 tmmanSyncObjCreate 時所創的同步物件占有的記憶體空間

每次使用 user mode library 前，都要先使用 tmmanInitialize 函式先初始化 user library 裡的全域結構，裡頭重要的資料包括控制 PNX1005 裝置檔的 file descriptor、memory apertures 映射到 user space 之後的虛擬記憶體位址。

而 user mode library 裡提供的函式主要就由多個不同命令碼的 ioctl 系統呼叫組成，讓每個 user mode library 裡的函式都可以是 ARM 與 DSP 溝通的其中一個工具，由於 tmmanInitialize 時就已經知道裝置檔的 file descriptor，所以實作功能函式時使用系統呼叫就不怕用到別的驅動程式的 driver handler 去了。大部份的功能函式功能已於(表 3.14)說明，以下介紹幾個特別的功能函式它的實作與應用。

3-4-1 tmmanMapDeviceMemory—將 DSP 端 RAM aperture、MMIO aperture 與 shared memory 空間映射到 user space

(表 3.15) tmmanMapDeviceMemory 內部呼叫

```
BooltmmanMapDeviceMemory (PTMMAN_DEVICE_INFO DeviceMap)
{
 tmifDSPOpen                TMIFOpen;
 tmifDSPInfo                TMIFDSPInfo;
 tmifDSPInternalInfo        TMIFDSPInternalInfo;
 tmifGenericFunction        TMIFGeneric;
 UInt32                     DSPHandle;
 Bool                       Result;
 UInt32                     BytesReturned;

 //tmmanDSPOpen()
 //map the global number to a driver specific number
 TMIFOpen.DSPNumber= DeviceMap->DSPNumber;

 IOCTL_FILE( DeviceMap->DriverMap->DriverHandle,
```

```
        constIOCTLtmmanDSPOpen,
        &TMIFOpen, sizeof(tmifDSPOpen),
        &TMIFOpen, sizeof(tmifDSPOpen),
        &BytesReturned, &Result );

    DSPHandle = TMIFOpen.DSPHandle; //DSPOpen 後得到在 kernel 中儲存 PNX1005 資訊的陣列內容位址
    TMIFDSPInfo.DSPHandle = DSPHandle;//即將透過 ioctl 從 kernel 中陣列內容取出資料

    // tmmanDSPGetInfo()
    IOCTL_FILE( DeviceMap->DriverMap->DriverHandle,
        constIOCTLtmmanDSPInfo,
        &TMIFDSPInfo, sizeof( tmifDSPInfo),
        &TMIFDSPInfo, sizeof( tmifDSPInfo),
        &BytesReturned, &Result );

    // tmmandDSPGetInternalInfo()
    TMIFDSPInternalInfo.DSPHandle = DSPHandle;//即將透過 ioctl 從 kernel 中陣列內容取出資料

    IOCTL_FILE( DeviceMap->DriverMap->DriverHandle,
        constIOCTLtmmanDSPGetInternalInfo,
        &TMIFDSPInternalInfo, sizeof(tmifDSPInternalInfo),
        &TMIFDSPInternalInfo, sizeof(tmifDSPInternalInfo),
        &BytesReturned, &Result );
    ////////////////////////////////////////////////////////////////////////
    // BEGIN KERNEL->USER MEMORY MAPPING
    // ********** DO NOT CHANGE THE SEQUENCE OF THE FOLLOWING CALLS **********
    // kernel mode routine trimedia_mmap() mirrors this sequence,
    // trimedia_mmap() needs to be updated in case of any changes here
    ////////////////////////////////////////////////////////////////////////

    // SDRAM MAPPING

    DeviceMap->SDRAMSize = TMIFDSPInfo.Info.SDRAM.Size;

    DeviceMap->SDRAMUserAddress=mmap(0,DeviceMap->SDRAMSize,PROT_READ|PROT_WRITE, MAP_SHARED,
                                     DeviceMap->DriverMap->DriverHandle,0)
    // MMIO MAPPING

    DeviceMap->MMIOSize = TMIFDSPInfo.Info.MMIO.Size;

    DeviceMap->MMIOUserAddress=mmap(0, DeviceMap->MMIOSize, PROT_READ|PROT_WRITE, MAP_SHARED,
                                     DeviceMap->DriverMap->DriverHandle,0)

    // SHARED MEMORY MAPPING

    DeviceMap->ShmemSize = TMIFDSPInternalInfo.Info.Memory.Size;

    DeviceMap->ShmemUserAddress=mmap(0,DeviceMap->ShmemSize,PROT_READ|PROT_WRITE, MAP_SHARED,
                                     DeviceMap->DriverMap->DriverHandle,0)

    ////////////////////////////////////////////////////////////////////////
    // END KERNEL->USER MEMORY MAPPING
    ////////////////////////////////////////////////////////////////////////
    // tmmanDSPClose()
    TMIFGeneric.Handle = DSPHandle;

    IOCTL_FILE( DeviceMap->DriverMap->DriverHandle,
        constIOCTLtmmanDSPClose,
        &TMIFGeneric, sizeof( tmifGenericFunction),
```

```
        &TMIFGeneric, sizeof( tmifGenericFunction),
        &BytesReturned, &Result );
    return True;
}
```

先指定要做 constIOCTLtmmanDSPOpen 這個 command 的 ioctl 系統呼叫，透過 DSPNmber 得到儲存該 PNX1005 的 RAM 與 MMIO memory aperture 基底位址資訊及共享記憶體位址的資料結構位址；之後做 constIOCTLtmmanDSPInfo 的 ioctl 動作，利用前面呼叫 constIOCTLtmmanDSPOpen 的 ioctl 得到的資料結構位址，從中取出 PNX1005 的 RAM aperture 、MMIO aperture 的大小；接著利用同樣的方法，指定做 constIOCTLtmmanDSPGetInternalInfo 這個 command 的 ioctl 動作，得到在 init_module() 時配置好的共享記憶體大小；最後再從 RAM 開始做 mmap 這個系統呼叫，傳入映射空間大小，驅動程式的 mmap handler(3-1-3 節)再根據 file 結構中的 MapState 判斷要哪段空間作映射，便使用該段空間的起始位址加上 mmap 系統呼叫傳入的空間大小，將記憶體映射到 user space 去並回傳每個區間的 user space 位址給應用程式。

tmmanMapDeviceMemory 的應用—tmload 指令中，用來下載 DSP 端可執行檔

這個動作在將 DSP 端可執行檔複製到 DSP 端 RAM 時特別重要。使用 tmmanMapDeviceMemory 將 RAM 映射到 user space 的虛擬記憶體空間後，我們就可以把虛擬記憶體空間中的 RAM aperture 起始位址當成是 DSP 端可執行檔要下載到的目標位址，而後要求將 ARM 端檔案系統中儲存的 DSP 端可執行檔也映射到虛擬記憶體空間中當作來源。此時雖然同樣呼叫 mmap 系統呼叫，但藉由傳入 mmap 系統呼叫的 file descriptor 的不同，可以分辨出要使用的是哪個 mmap handler，於是，在映射檔案系統的檔案到 user space 時，會先去開啟要映射的檔案得到 file descriptor，然後指定要使用檔案系統的 mmap 機制來對檔案作映射，所以就不會使用到 TMMan 驅動程式的 mmap handler(式 2)。最後，應用程式便可以透過簡單的 memcpy 函式將 DSP 端可執行檔複製到 DSP 端指定的 RAM 位址空間去。

(式 2)映射檔案系統中的 DSP 端可執行檔到 user space

```
fd = open(TargetExecutableName, O_RDWR);
sourceBuf = mmap(0, stat_buf.st_size, PROT_READ,MAP_PRIVATE, fd, 0);
```

3-4-2 啟動 DSP 端運作的函式庫

等 ARM 端將 DSP 端可執行檔複製到 PNX1005 的 RAM aperture 起始位址後(3-4-1)，ARM 就可以利用 tmmanDSPLoad 功能函式透過 command 為 constIOCTLtmmanDSPLoad 的 ioctl 系統呼叫，由 ioctl handler 到 HAL 中先儲存這個 PNX1005 要開始執行的 RAM aperture 起

始位址，也就是說 PNX1005 會從 DSP 端可執行檔開始執行。接著使用 tmmanDSPStart 功能函式透過 command 為 constIOCTLtmmanDSPStart 的 ioctl 系統呼叫進入 kernel，利用已經映射到 kernel space 的 MMIO apertures，對記憶體進行寫入來直接設定 PNX1005 的暫存器（表 3.16），以啟動 PNX1005 開始運作。

PNX1005 被啟動前，ARM 端會先將 PNX1005 的 TM_CONTROL 這個暫存器的 TM32_IDLE_REQ 這個 bit 設為 1 以通知 PNX1005 的 CPU 進入 powerdown 的程序，而後 ARM 端會去檢查 TM_STATUS 暫存器的 TM32_IDLE_ACK 這個 bit 是否為 1，是的話表示 DSP 已經知道要做 powerdown 的動作並停止 PNX1005 的 CPU 運作，於是 ARM 端再接著設定 LOCAL_SW_RESETS 暫存器來重置 PNX1005 的 CPU。藉由上述確認 DSP 端沒有在運轉後，ARM 端會接著將 IClear 暫存器每個 bit 設為 1 並把 IMASK 暫存器全設為 0，用來清空之前發生的中斷狀態，然後設定 TM32_DRAM_LO 為 RAM apertures 起始位址、TM32_DRAM_HI 為 RAM apertures 終止位址以及將 DSP 端可執行檔被下載到的位址設定到 TM32_START_ADR 暫存器中。最後，當 ARM 端設定 TM32_START 這個暫存器後 PNX1005 的 CPU 就開始啟動，而且會從 TM32_START_ADR 暫存器記錄的位址開始運算。

(表 3.16)啟動 DSP 端需要設定的暫存器

```
// Ensure the TM is stopped, leave PLL & Clock setting as is
      status = halStopDSP(HalHandle);
// clear the IMask & IClear register
      *(UInt32*)(Hal->MMIOAddrKernel + ICLEAR) = (UInt32)(~0x0);
      *(UInt32*)(Hal->MMIOAddrKernel + IMASK) = (UInt32)(0x0);

      *(UInt32*)(Hal->MMIOAddrKernel + TM32_DRAM_LO) =
      halAccess32(HalHandle,Hal->SDRAMAddrPhysical.LowPart + Hal->TM32DramOffset);
      *(UInt32*)(Hal->MMIOAddrKernel + TM32_DRAM_HI) =
      halAccess32(HalHandle,Hal->SDRAMAddrPhysical.LowPart + Hal->TM32DramOffset +
      Hal->TM32DramLen);
      *(UInt32*)(Hal->MMIOAddrKernel + TM32_DRAM_CLIMIT) =
      halAccess32(HalHandle,Hal->SDRAMAddrPhysical.LowPart + Hal->TM32DramOffset +
                                                          Hal->TM32DramCLimitOffset);
      *(UInt32*)(Hal->MMIOAddrKernel + TM32_START_ADR) =
      halAccess32(HalHandle,Hal->SDRAMAddrPhysical.LowPart + Hal->TM32DramOffset +
                                                                  Hal->TM32StartOffset);
       // Take the TriMedia out of reset
       temp = halAccess32(HalHandle,*(UInt32*)(Hal->MMIOAddrKernel + TM32_START)) &
       ~constTMManTM32_START_ST;
       temp |= constTMManTM32_START_ST;

      *(UInt32*)(Hal->MMIOAddrKernel + TM32_START) = halAccess32(HalHandle,temp);
```

之後 DSP 端便會獨立執行 DSP 端可執行檔的運作，然而此時 ARM 端應用程式並沒有結束，它還有一個負責處理 DSP 端 C I/O 請求的處理程式在運作(第五章 CRT service)，會使用訊息函式庫(3-4-4 節)進行底層的訊息交換，也可能需要做兩端 CPU 的程式同步，因此就會使用同步函式庫(3-4-5 節)。

3-4-3 依照本地端 OS 進行同步物件操作的函式庫

這組函式庫主要由 tmmanSyncObjCreate、tmmanSyncObjSignal、tmmanSyncWait、tmmanSyncObjDelete 組成，均由 ioctl 系統呼叫完成驅動程式的運作，由 ioctl 系統呼叫進入 ioctl handler 後，會由 osal.c 來實作對應的 syncobjCreate、syncobjSignal、syncobjWait 及 syncobjDelete 函式。

(表 3.17) 本地端 OS 進行同步物件操作的驅動程式模組

```
case constIOCTLtmmanSyncObjCreate:
{
   tmifGenericFunction*   TMIF =(tmifGenericFunction*)IOParameters;

   if ( syncobjCreate(0, 0, &TMIF->Handle, 0) ) {
                TMIF->Status = statusSuccess;
    } else {
                TMIF->Status = statusObjectAllocFail;
            }
 }
     break;
```

在 linux 中會先在 kernel 創建一個 semaphore 結構，並將之初始為 0

```
struct semaphore *sem = Null;
  sem = (struct semaphore *)kmalloc(sizeof(*sem),GFP_KERNEL);
     if (!sem) {
      DPF(0,("osal.c: syncobjCreate: kmalloc failed.\n"));
      return False;
     }

  init_MUTEX_LOCKED(sem);//counter 被設成 0
  *SynchronizationHandlePointer = (UInt32)sem;
```

```
case constIOCTLtmmanSyncObjWait:
{
    tmifGenericFunction*   TMIF =(tmifGenericFunction*)IOParameters;

    if ( syncobjBlock(TMIF->Handle) ) {
          TMIF->Status = statusSuccess;
    } else {
                TMIF->Status = statusUnknownErrorCode;
            }
}
          break;
```

在 linux 中會使用 down_interruptible 函式將 semaphore 結構減一

```
down_interruptible((struct semaphore*)SynchronizationHandle);
```

```
case constIOCTLtmmanSyncObjSignal:
{
    tmifGenericFunction *TMIF =(tmifGenericFunction*)IOParameters;

    if (syncobjSignal(TMIF->Handle) ) {
           TMIF->Status = statusSuccess;
    } else {
                TMIF->Status = statusUnknownErrorCode;
            }
}
          break;
```

在 linux 中會使用 up 函式將 semaphore 結構加一

```
up((struct semaphore *)SynchronizationHandle);
```

```
case constIOCTLtmmanSyncObjDelete:
{
    tmifGenericFunction *TMIF =(tmifGenericFunction*)IOParameters;
```

```
    if ( syncobjDestroy(TMIF->Handle) ) {
                TMIF->Status = statusSuccess;
    } else {
                TMIF->Status = statusUnknownErrorCode;
}
          break;
```

在 linux 中會使用釋放 syncobjCreate 時創建的 semaphore 結構空間
kfree((struct semaphore *)SynchronizationHandle);

3-4-4 傳遞訊息函式庫

這組函式庫主要由 tmmanMessageCreate、tmmanMessageSend、tmmanMessageReceive、tmmanMessageDestroy 組成，均由 ioctl 系統呼叫完成驅動程式的運作，由 ioctl 系統呼叫進入 ioctl handler 後，會由 message.c 來實作對應的 messageCreate、messageSend、messageReceive、messageDestroy 完成傳遞訊息相關的運作(表 3.18)。

(表 3.18)ioctl handler 中有關傳遞訊息的 command 與運作

```
case constIOCTLtmmanMessageCreate :
{
    tmifMessageCreate*  TMIF =(tmifMessageCreate*)IOParameters;
    TMManDeviceObject* TMManDevice = (TMManDeviceObject*)TMIF->DSPHandle;
    TMIF->Status = messageCreate (
                TMManDevice->MessageManagerHandle,
                filep, //GetCurrentProcess(),
                TMIF->Name,
                TMIF->SynchObject,
                TMIF->SynchFlags,
                &TMIF->MessageHandle);
           ReturnInformation = sizeof ( tmifMessageCreate );
 }
        break;

case constIOCTLtmmanMessageDestroy :
{
    tmifGenericFunction*  TMIF =(tmifGenericFunction*)IOParameters;
    TMIF->Status = messageDestroy (TMIF->Handle);
    ReturnInformation = sizeof ( tmifGenericFunction );

 }
        break;

case constIOCTLtmmanMessageSend :
{
     tmifMessageSR* TMIF =(tmifMessageSR*)IOParameters;
     TMIF->Status = messageSend (TMIF->MessageHandle,TMIF->Packet );
     ReturnInformation = sizeof ( tmifMessageSR );
 }
        break;

case constIOCTLtmmanMessageReceive :
{
    tmifMessageSR*  TMIF =(tmifMessageSR*)IOParameters;
     TMIF->Status = messageReceive (TMIF->MessageHandle,TMIF->Packet );
     ReturnInformation = sizeof ( tmifMessageSR );
        break; }
```

在啟動 PNX1005 前，應用程式需要使用 tmmanMessageCreate 創建一個資料接收時要使用的 queue buffer、註冊阻斷機制所需而且事先由 tmmanSyncObjCreate 創建的 semaphore 結構於溝通管道中，等 DSP 端啟動後，再使用 tmmanMessageReceive 從 queue buffer 接收 DSP 端的請求或使用 tmmanMessageSend 將資料傳送複製到 shared memory 上。(詳細溝通機制請看第四章)

3-4-5 同步函式庫

同步函式庫由 tmmanEventCreate、tmmanEventDestroy、tmmanEventSignal 組成，與傳遞訊息函式庫類似的是，它們也是由 ioctl 系統呼叫設計而成，在 kernel 中也另外存在一個 event.c 來實作 eventCreate、eventDestroy、eventSignal(表 3.19)。

(表 3.19) ioctl handler 中有關同步的 command 與運作

```
case constIOCTLtmmanEventCreate :
{
    tmifEventCreate*    TMIF =(tmifEventCreate*)IOParameters;
    TMManDeviceObject* TMManDevice = (TMManDeviceObject*)TMIF->DSPHandle;
    TMIF->Status = eventCreate (
                    TMManDevice->EventManagerHandle,
                    filep, //GetCurrentProcess(),
                    TMIF->Name,
                    TMIF->SynchObject,
                    TMIF->SynchFlags,
                    &TMIF->EventHandle);
    ReturnInformation = sizeof ( tmifEventCreate );
}
        break;

case constIOCTLtmmanEventDestroy :
{
    tmifGenericFunction*    TMIF =(tmifGenericFunction*)IOParameters;

    TMIF->Status = eventDestroy (TMIF->Handle );
    ReturnInformation = sizeof ( tmifGenericFunction );

 }
        break;

case constIOCTLtmmanEventSignal :
{
    tmifGenericFunction*    TMIF =(tmifGenericFunction*)IOParameters;

    TMIF->Status = eventSignal (TMIF->Handle );
    ReturnInformation = sizeof ( tmifGenericFunction );
 }
        break;
```

當需要做兩端 CPU 的同步時，基本上就是透過兩端 OS 自己的阻斷機制與喚醒機制來運作，只是因為這是兩個獨立的 CPU，所以在 DSP 端啟動前還需要額外建立同步管道，用來告訴對方要喚醒哪個控制程序的變數(例如 semaphore 結構)。於是我們需要先使用

tmmanSyncObjCreate 於 kernel 中創建用在兩端同步用的 semaphore 結構並初始化，然後再使用同步函式 tmmanEventCreate，將這個 semaphore 結構註冊於同步管道中，如果這個 semaphore 在之後有使用 tmmanSyncObjWait，就會依照該 OS 環境下做 block 動作的機制對這個 semaphore 進行阻斷(以 Linux 來說就是使用 down_interruptible 函式)，然後等著另一端 CPU 送出同步訊息(有關同步的詳細機制請看第四章)。

如果今天本地端要同步另一端的 CPU，則會利用 tmmanEventSignal 經由 ioctl handler 中的 eventSignal，透過指定的同步管道到 shared memory 的 Event 區塊做旗標設定，以表示是要對對方註冊於同步管道中的 semaphore 結構做 signal 的動作。

3-5 第三章結論

ARM 與 DSP 端的 PNX1005 裝置必須使用 TMMan 驅動程式來做資源的分配與溝通，因為 TMMan 驅動程式是在 linux 環境下開發的，所以必須實作基本的 kernel mode 驅動程式：實作出驅動程式 handlers 並存到 file_operations 中，還有實作驅動程式的程式進入點 init_module 來註冊驅動程式、配置 shared memodry 等動作，完成基本的 kernel mode 驅動程式設計後，我們還設計 user mode library 讓應用程式來使用。

對於熟悉驅動程式開發的人來說，在做應用程式開發時，不一定要使用 user mode library，它可以直接使用 ioctl 等系統呼叫調用 file_operations 結構指到的驅動程式 handler 去控制裝置即可，可以節省還要再去熟悉 user library 使用介面的時間；對於不了解驅動程式運作架構也不了解 kernel/OS 與驅動程式關係的人，藉由人性化命名的呼叫介面，可以很容易知道怎麼使用這些工具來獲得硬體相關資訊、要求硬體提供需要的服務，節省很多需要重新了解整個 OS 控制驅動程式的背景知識所花的時間，而且 user mode library 把應用程式原本繁複的系統呼叫使用功能化，原本可能散落在應用程式裡的系統呼叫還有一些控制操作，我們可以把他們統整出一個一個的功能函式，讓應用程式撰寫更為精簡，可以參考附錄 F 的 tmload.c 實例。

除了 ARM 端有這樣的 user mode library，DSP 端也有同功能的 user mode library，只是內部實作方式會因應 OS 的不同而有差異(DSP 端為 pSos)，所以下兩章在探討兩端 CPU 的溝通機制時，ARM 端應用程式主要會用到依照本地端 OS 進行同步物件操作的函式庫、傳遞訊息函式庫與同步函式庫，而 DSP 端也會使用對應於 ARM 端函式庫的 user mode library 搭配著完成一項 service。第四章我們將會先說明訊息函式庫下的 OS 層，實質的訊息溝通機制，並在最後帶入訊息函式庫，說明應用層是如何透過訊息函式庫介面，與 OS 層的傳輸做好連結；第五章則以 ARM 端 user mode library 起始，往上層為應用程式設計一套 CRT 服務機制來處理 DSP 端傳送過來的指令，並探討 DSP 端如何包裝指令、使用 DSP 端 user mode library 將訊息傳遞給 ARM。

第四章 ARM 與 DSP 端底層溝通機制與架構

兩個各自獨立執行應用程式的核心要進行溝通時，如果沒有共享媒介，應用程式與應用程式之間是不能直接進行訊息交換的，所以在兩個核心之間必須有一塊共享的記憶體空間(共享記憶體)做為傳遞訊息的媒介，來儲存兩個核心之間要傳送的訊息。但是一個應用中可能會有多個 service 都藉由共享記憶體傳送訊息，為了讓接收端可以區分處理訊息的 service，所以在軟體上我們使用分層架構去控制共享記憶體的不同區塊，經由每層的判斷最後喚醒特定的 service 執行緒，由它對這個訊息做處理。

4-1 節會先介紹兩個核心之間會利用共享記憶體進行溝通的模式，主要有兩類，一類為訊息交換模式，一類則為事件觸發模式。為了讓另一端知道要處理的訊息是以何種溝通模式進行，或為了區分利用相同溝通模式進行溝通的多個服務，4-2 節會接著探討 ARM 端如何做共享記憶體的劃分，以達到分辨事件與區分服務的目的。為了從軟體上來使用共享記憶體，又因為共享記憶體上被劃分了多個控制區塊，因此 4-3 節會介紹每個區塊對應到的階層，並利用分層概念來說明分層的溝通機制，最後於 4-4 節，會在 ARM 端 Linux 環境下，介紹軟體上實作分層溝通機制的方法，並搭配 user mode library 提供的傳遞訊息函式庫、同步函式庫來完成分層溝通管道的建置與溝通。

4-1 使用共享記憶體進行的溝通模式

為了要讓兩端核心可以溝通，我們就在 ARM 端開一塊記憶體空間當做**共享記憶體**，這塊共享記憶體就用來儲存 DSP 與 ARM 端溝通的訊息。兩端核心都有機會扮演傳送或接收的角色，當傳送端把訊息寫到共享記憶體後，必須通知對方訊息已經抵達，請他做接收的動作，所以我們還使用了硬體**中斷機制**來達到溝通的目的。

使用共享記憶體可以完成的溝通模式有兩種,一種是訊息交換,一種是事件觸發同步，兩種模式運作上都使用到應用層與 OS 層的概念，OS 層主要就負責底層訊息收發的動作，應用層則透過阻斷介面與 OS 層訊息交換運作做分隔。訊息交換模式指的是說，接收端應用層以 Message 層，作為應用層與 OS 層底層處理訊息傳遞動作的分隔，接收端應用層會透過 Message 層的阻斷介面,暫時將應用程序 block 住,避免 packet 還沒插入 queue buffer 就對它作讀取，直到傳送端應用層將存有資料的 packet 傳入 OS 層並經由 OS 層寫入共享記憶體以後，傳送端 OS 層對接收端發出中斷，接收端 OS 層收到中斷，OS 層會從共享記憶體中取得 packet，然後將 packet 塞入傳送 packet 的終點--queue buffer 後，再由 kernel 去 signal Message 層的阻斷介面，喚醒被 block 的應用程式，讓它可以到 Message 層的

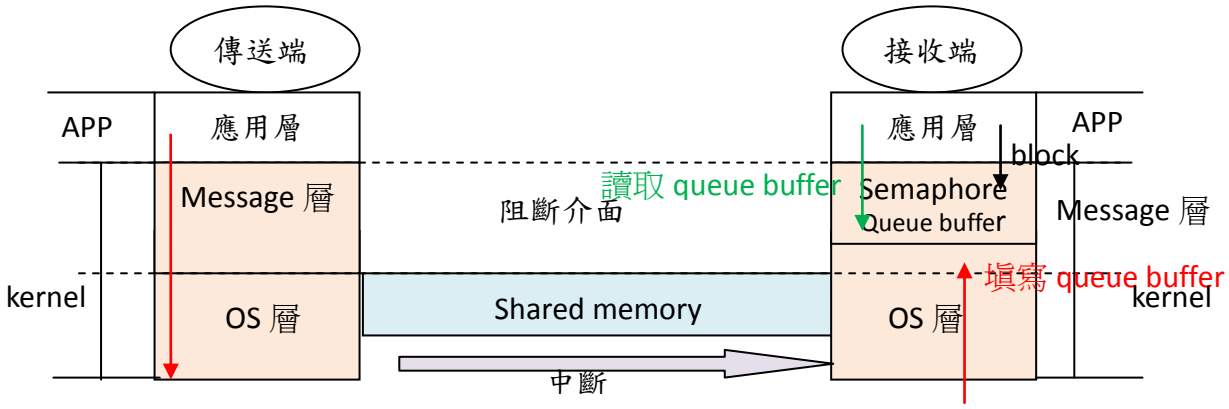
queue buffer 讀 packet，這樣的過程稱為訊息交換模式，主要用於指令傳輸。

而事件同步指的是說，EventWait 端的應用層，以 Event 層作為應用層與 OS 層底層處理訊息傳遞動作的分隔，主要用來控制兩端應用程序的執行順序。和訊息傳遞模式最大的不同在於，它不是在傳送 packet，所以不需要 queue buffer，雖然都有阻斷介面來完成接收端等待訊息的動作，但是被喚醒的執行程序之後要做的事情則是不同的，訊息傳遞模式會接著到 queue buffer 作讀取 packet 的動作，事件觸發模式則是在應用程序被喚醒之後，繼續執行應用程序後續的動作，不一定是針對 queue buffer 做事情。

4-1-1 訊息交換模式

一般網路環境下，應用程式在 client-server 的架構下，可以透過單純的 RevFrom() 與 SendTo()函式，在完全不了解 OS 底層溝通的情況下，只要正確使用函式，就能順利將訊息從遠端接收過來，或是將訊息傳送到遠方；仿照這樣的觀念，兩個核心之間的溝通由底層的共享記憶體與 interrupt 機制來做訊息的交換，然後在各自的系統環境中則分成應用層與 OS 層，應用層則利用本地端 OS 提供的阻斷機制來與 OS 層做銜接，而且應用層可以使用第三章提到的傳遞訊息功能函式，提供應用層傳或收的函式介面。

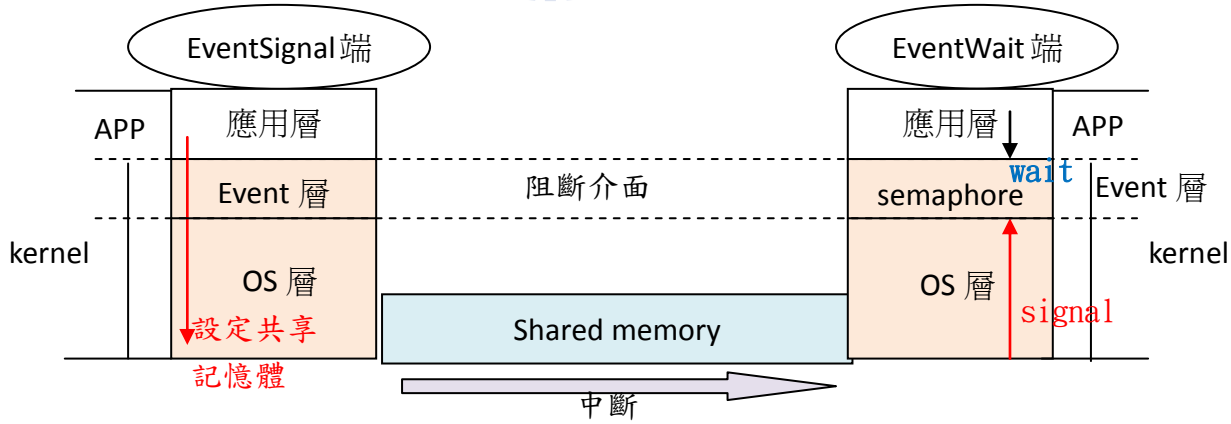
因此，共享記憶體的底層溝通屬於兩端 OS 層的運作，而兩端應用層是看不到底層共享記憶體的訊息交換的，但是為了讓 OS 層讀寫到的訊息可以被應用程式使用，因此在 OS 層便設置了一個 first-in-first-out 且為 circular 型態的 queue buffer，而且應用層使用 tmmanSyncobjCreate 函式在 kernel 設置了程序控制變數(semaphore)，並在 tmmanMessageCreate 時將這個變數位址存入 Message 層，而這個 Message 層就做為 OS 層與應用層的介面。一開始為了避免接收端對沒有訊息的 queue buffer 作讀取，於是應用程式先利用 tmmanSyncobjWait 函式對 semaphore 作 block，讓應用程序讀取 queue buffer 的動作無法執行，傳送端應用層則利用 tmmanMessageSend 函式將 packet 傳入 OS 層，然後由 OS 層負責寫入共享記憶體，傳送端接著對接收端發出中斷，接收端的 OS 層接收到中斷，於是由共享記憶體讀出訊息後，便將 OS 層讀到的 packet 填寫到 Message 層裡的 queue buffer，並在填入資料於 queue buffer 後直接請 kernel 對 Message 層的 semaphore 做 signal，以喚醒應用程式繼續以 tmmanMessageReceive 函式做讀取 queue buffer 的動作。所以 OS 層驅動程式的運作，就是在實作訊息傳遞的函式庫的內容，讓應用層可以使用訊息傳遞函式庫從 Message 層接收資料。



（圖 4.1）訊息交換模式運作機制

4-1-2 事件觸發的兩端同步行為

兩端溝通除了用來做訊息的傳遞外，兩個各自運作應用程式的 CPU，執行某一行程式前可能必須等待另一個 CPU 完成某件事之後，才能繼續執行下面的程式，此時便需要事件觸發的動作來完成兩顆 CPU 之間程序上的同步，以保證資料執行的正確。應用層在事件觸發的處理上，並不需透過 Message 層與 Channel 層做接收訊息的動作。EventWait 端應用層會先使用 tmmanSyncobjCreate 函式在自己的 kernel 設置一個 semaphore，然後應用程序會接著使用 tmmanSyncobjWait 函式請求 OS 把這個 semaphore block 住，於是 EventWait 端應用程序就不能繼續執行。直到 Eventsignal 端應用程序使用 tmmanEventSignal 函式，進入 OS 層對共享記憶體寫入簡單的 signal 信號或 flags，並對 Eventwati 端送出中斷，EventWait 端收到中斷後 OS 層就知道有信號或 flags 傳送過來，然後才請 kernel 去 signal 之前被 block 的 semaphore，EventWait 端的應用程序才可以繼續執行。



（圖 4.2）事件觸發同步模式運作機制

4-2 共享記憶體的使用與劃分

目前介紹的溝通模式就有兩種，我們把一個應用使用到的 service，依照溝通模式，將使用訊息交換功能的 service 歸類為 channel 事件，使用事件觸發功能的 service 歸類

為 Event 事件。然而應用層的應用程式可能會同時用到訊息交換或是事件觸發 service，而且也可能同時使用多個傳遞訊息 service 或一個以上的事件同步 service 來完成一個應用，在這樣的 multitasking 環境中，卻只有一塊共享記憶體與一條中斷線！

為了區分目前是應用程式裡哪個 service 要進行溝通，所以我們由 ARM 端開始，為每個 service 提供一個代號，稱為 service 代號，又由於 service 可能分屬於 channel 事件或 Event 事件，所以 ARM 端又為不同的事件分配一個代號，channel 事件就稱為 channel 事件代號，Event 事件就稱為 Event 事件代號，每個事件代號會對應到一組 flags，藉由設定某一組 flags 來區分要做的是 channel 事件還是 Event 事件。

為了傳遞 service 代號與 flags，ARM 端於是對共享記憶體劃分一塊空間當作 Channel 區塊，用來儲存欲傳送的 packet 與傳遞訊息 service 代號，在 Channel 區塊共可存放 64 個 mailbox，每個 mailbox 就是一組 packet 與 service 代號，然後還存有讀取編號與寫入編號，在運作時用來指示可讀寫的 mailbox。另外，共享記憶體還劃分了 Vintr 區塊，裡頭再分成 4 個小區塊，每個小區塊就儲存一個事件的一組 flags，然後利用 Channel 事件代號表示一個小區塊、Event 事件代號則對應到另一個小區塊，如此傳送端就可以利用事件代號設定對應的 flags，接收端則檢查每個小區塊的 flags，看誰被設定了，就由該小區塊代號對應的事件來做後續的處理，並清除 flags 設定。

事件觸發 service 運作上還使用了共享記憶體的 Event 區塊，區塊裡分成 16 個小區塊，每個小區塊也儲存了一組 flags，由事件觸發 service 代號來對應其中一個小區塊，與 Vintr 區塊類似，EventSignal 端藉由事件觸發 service 代號設定對應小區塊中的 flags，EventWait 端就去檢查所有的小區塊看哪一個 flags 有被設定，就知道是要去 signal Event service 代號對應的 semaphore，之後再清除 flags 的設定。由於溝通是雙向的，所以 Channel 區塊、Vintr 區塊與 Event 區塊都各有兩組，給兩端的 service 使用。

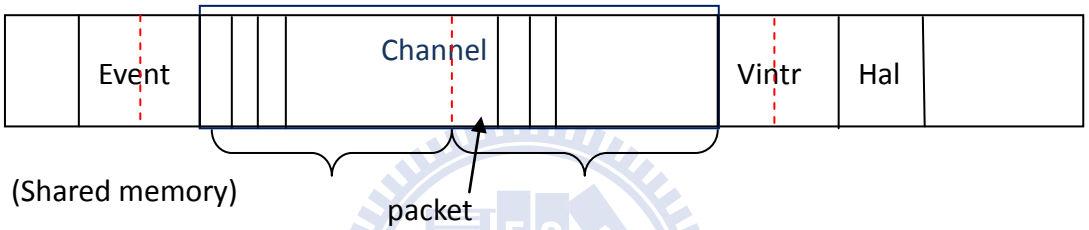
為了讓 ARM 端分配的 service 代號與事件代號讓 DSP 端知道，這樣傳收時兩端才有一致的判斷與設定標準，所以共享記憶體裡還有一塊 NameSpace 區塊，用來儲存兩端都需知道的 service 代號或事件代號，另外也為了把 ARM 端劃分好的區塊起始位址告訴 DSP 端，這樣 DSP 端才知道各個區塊在哪裡，所以 ARM 也在共享記憶體上劃分一塊 TMManSharedStruct 區塊，儲存每個區塊的起始位址。

最後共享記憶體還有一個 HAL 層區塊，儲存了一組 flags，讓 ARM 端接收中斷後想要清除中斷暫存器的設定前，確認 DSP 端目前沒有正在對 ARM 端發出中斷。有關共享記憶體的區塊分割請看 4-2-1 節。

4-2-1 共享記憶體的分區

4-2-1-1 Vintr、Channel、Event 區塊

由前幾段，我們知道共享記憶體劃分出至少三個區塊來儲存Vintr、Channel、Event的代號與flags，以及記錄暫存器狀態的Hal區塊，然而因為ARM與DSP端都可以作為傳送端或接收端，所以我們需要確定訊息傳遞的方向，身為傳送端當然就要使用「傳送給對方」方向的區塊，接收端則從同樣的區塊來接收訊息，所以我們的環境中會為每個CPU提供兩組方向，ARM端為傳送端時便將訊息填入「傳送訊息給DSP」方向上的區塊中，對DSP來說這個區塊就是「接收ARM端訊息」方向的區塊，於是便能從中讀取ARM傳遞過來的訊息；同理，ARM端為接收端時，它就等著DSP端將訊息寫到「傳送訊息給ARM端」方向的區塊，然後ARM再從同一塊區塊中讀取訊息。因此在這三個大區塊中，還需要在其中分成兩份，每一份就負責一個方向的訊息傳輸。

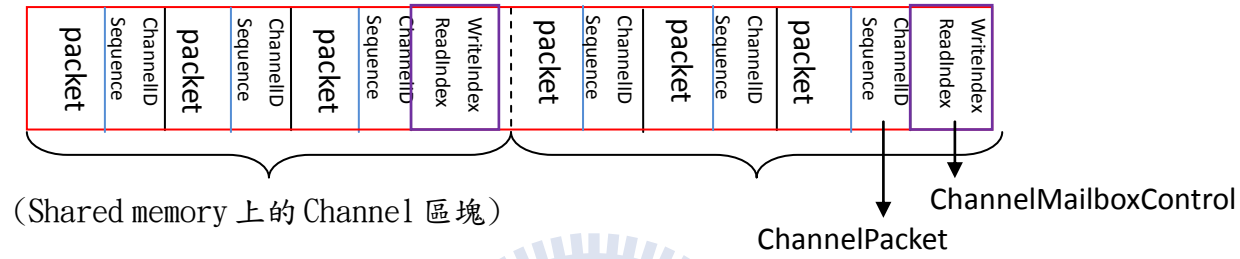


(圖 4.3)共享記憶體區塊劃分—Vintr、Channel、Event 區塊

目前我們在每個方向的**Vintr區塊**切成四小塊，channel事件與Event事件需要設定的flags則分別存於其中的兩小塊，當傳送端想做傳送資料的動作時，便會到「傳送訊息給對方」方向的channel事件所在小區塊去設定flags，如果是要做同步的話就到Event事件的小區塊去設定flags，接收端於是就到這個方向的Vintr區塊檢查這兩個有被使用的小區塊，看看flags是否被設定，有被設定的話就表示該事件發生了，於是便可以利用該事件的ISR往上繼續做該事件的處理。相同的，每個方向的**Event區塊**會被切成16小塊，每個小塊裡面儲存了一組flags，每一小塊都由一個Event service使用，於是wait的那端在自己的OS裡面被block住以後，便等著另一端來signal，另一端於是在「傳送訊息給對方」方向上去設定該Event service使用的flags並發出中斷，EventWait端收到中斷後便會檢查Event區塊中有被使用的小區塊，利用其中的flags判斷是由哪一個Event service引起的，然後對該Event service對應的、被block住的thread做unblock/signal。

而**Channel區塊**裡每個方向的子channel都會有一組讀取編號與寫入編號，還有64個mailbox這樣的傳送單位，每個mailbox則包含兩個部分，一個是真正傳送資料(指令資料結構位址或是數字訊息)的packet，另一部分則儲存了service代號與傳送的順序編號。每

次傳送端會到「傳送訊息給對方」方向上的Channel區塊，依照該方向上的寫入編號，把欲傳送的資料，寫入該編號表示的mailbox裡的packet中，並同時寫入service 代號以表示是哪個service傳送這個packet，然後把寫入編號加一；接收端藉由Vintr區塊的flags設定知道要做的是Channel事件後，就會到這個方向上的Channel區塊檢查讀取編號與寫入編號，只要兩者不同就表示有新資料需要接收，於是便依照讀取編號表示的mailbox區塊從中複製packet到自己的系統裡，然後依據service代號將這個packet，存到Message層屬於這個service的queue buffer去，處理完後便把讀取編號加一，如果讀取編號與寫入編號不相等，表示還有新資料在Channel區塊中，所以再繼續重複資料接收的動作直到讀取編號與寫入編號相同為止。

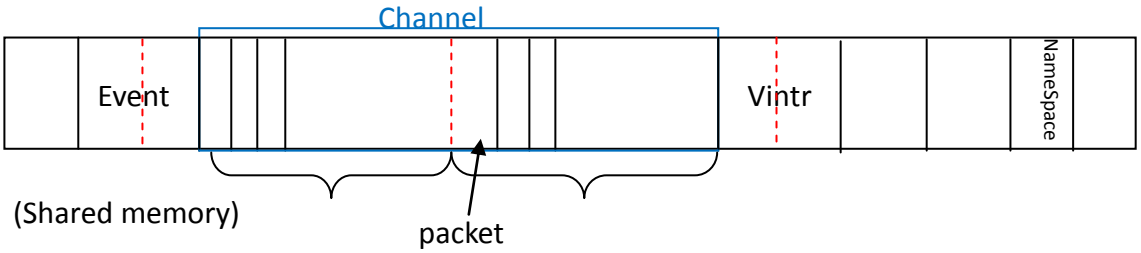


(圖 4.4)Channel 區塊的細部劃分

4-2-1-2 儲存代號的區塊

前面我們有提到過，service代號可以用來分辨收到訊息後要處理的是哪種service，其實也暗示了不管是傳送端還是接收端，傳送該訊息的service與接收該訊息後做處理的service必須是相互對應的，例如，DSP端要將影像寫到ARM端檔案系統，必須使用到ARM端的「C Runtime Service」服務，那麼DSP端本身就要有這個service的代號，在請求這個服務處理訊息時，也順便把這個代號傳過去ARM端，ARM收到之後才知道這個訊息是要由C Runtime Service來處理。

所以就**由ARM開始**，先在ARM端為每個service分配好service代號，而這個代號要告訴DSP端的話，就要先把它寫到共享記憶體上的NameSpace區塊，DSP於是可以從這塊空間讀取service代號，做為傳送訊息時的附屬資訊，在接收ARM端傳送過來的訊息時，也會藉由這個service代號，知道這個訊息需要呼叫哪個service的ISR來處理。另外，在判斷「事件種類」或「同步哪一個程序」時，我們需要知道該事件或同步要設定的flags在哪裡，所以我們也可以為不同的事件種類與不同的同步程序分配一個事件代號或同步代號，然後可以用這些代號指示控制的flags位於Vintr區塊的哪個小區塊、Event區塊的哪個小區塊，於是傳送端便會利用事件代號設定其對應到的小區塊，接收端便會去檢查哪個小區塊的flags有被設定，之後便可根據被設定的那個小區塊對應到的事件代號，呼叫該事件的ISR。



（圖 4.5）共享記憶體區塊劃分--NameSpace 區塊

代號分配的方式可以有很多種，只要在環境中它可以為每種事件、service、同步提供辨識即可，在我們設計的環境中，service代號用來區分使用Channel層來傳送資料的各種service，而事件代號對傳送端來說，就指示了該事件可以設定的flags Vintr區塊，對接收端來說就是可以藉由flags知道是哪個事件發生了，然後藉由這個Vintr小區塊對應到的事件代號去呼叫處理這個事件的ISR。同步代號則與事件代號用途相似，只是最後ISR的處理是去signal該同步代號所表示的semaphore。下面提供我們使用的代號分配機制。

NameSpace 代號分配機制

在進行溝通以前，為了避免之後傳送不同 service 訊息時造成混亂，我們需要為每個 service 分配一個代號，並利用 flags 做特定的事件判別。為了分配代號給它們，我們於是依照「使用的區塊型態」、使用這個區塊 service 或事件(傳收訊息或同步)的「名字」，來分配代號「ID」給不同的 service 或事件，而這些 **triple**〈ID, 名字, 區塊型態〉就是 service/事件的特徵或身分，而同種型態的 triple 是不能擁有相同的名字的。

於是，在兩端的 OS 環境中，我們就建立了一個 NameSpace 表單，用來存放已經創建的身分(triple)，每次有新的身分需要被創建時，就會先到表單中比對想創建的身分型態是否有在表單中出現過，針對型態有重複的部分，我們再去比對身分的名字，如果新的身分名字又與這些表單中同型態的身分所擁有的名字相同的話，就表示這個身分已經在系統中出現過，就會回傳錯誤訊息，反之，則為這個新的身分配置一個 ID。

在 Channel 層，我們利用 NameSpace 機制分配出來的 ID 就稱為 ChannelID，由於 4-3-2-2 節在創建溝通管道時，Message 層與 Channel 層是同時被建立的，而且目前每個服務在 Message 層使用到的 Channel 只有一個，所以會發現 Message 層的 service ID 與 ChannelID 是一樣的，因此之後，我們便以 ChannelID 或 serviceID 表示某種服務的代號。而分配出來的這個 ChannelID，會跟著 packet 一起被寫入 mailbox 傳送給對方，對方收到 ChannelID 後可以決定這個 packet 要被塞入哪一個 service 的 queue buffer 中。

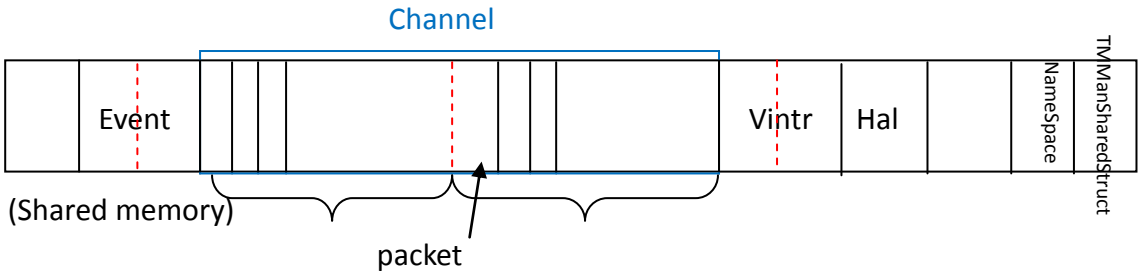
另外，Vintr 層與 Event 層因為共享記憶體上已經為 Vintr 區塊與 Event 區塊劃分好幾組 flags 小區塊，因此 Vintr 層的 VintrID 與 Event 層的 EventID 則用來指示該事件或

Event service 可以對哪個小區塊的 flags 進行設定，同時，VintrID 也可以作為呼叫 Channel 的 ISR 還是 Event ISR 的判斷，EventID 則可做為 signal 哪一個 semaphore 的依據。

由於 ARM 與 DSP 端必須為完成一個整體應用來合作，因此傳送端與接收端在進行某種 service 或事件判斷時所使用到的區塊部分是一致的，因此，ARM 與 DSP 端都需要知道每個 service 的各種 ID 資訊。因此，每次 ARM 端先使用 NameSpace 機制分配得到的 triple 資訊就會填入共享記憶體指定到的區塊(NameSpace)中，而這個區塊我們分成 64 個 NameSpaceControl 結構組成，每個 NameSpaceControl 有 24Bytes 儲存了該 service 或事件的 triple。DSP 端某個 service 或事件要使用共享記憶體裡某種型態區塊其中的小區塊前，就藉由”Name”與”Type”直接從 NameSpacke 區塊讀取對應的 triple 資訊出來，得到的 ID 就可以用來做 service 的判斷與 flags 所在的小區塊。

4-2-1-3 TMManShared 區塊

基本上shared memroy的劃分均由ARM端起始，所以在劃分共享記憶體區塊的過程中，ARM端就會將這些區塊的起始位址儲存起來，要使用這些區塊前再拿來使用，但是DSP要怎麼知道這些區塊位址在哪裡呢?所以我們在共享記憶體最前方就設置了TMManShared區域，將所有共享記憶體上被劃分出來的區塊起始位址記錄在TMManShared區域中，DSP做為傳送端要使用共享記憶體時，就可以得知Vintr區塊、Channel區塊、Event區塊在哪，然後再搭配事件代號來使用Vintr區塊或Event區塊裡的某個小區塊，或把訊息複製到Channel區塊裡。



（圖 4.6）共享記憶體劃分—TMManSharedStruct

4-2-2 共享記憶體的劃分實作

共享記憶體的配置與劃分是使用者在 console 下 insmod 指令後，在 kernel 中由 init_module 函式對 TMMan 驅動程式做初始時做的。

配置 shared memory 時，會先利用 dma_alloc_coherent(式 1)函式配置一塊固定大小且 uncache 的 ARM 端記憶體空間來當 shared memory，而總共配置的共享記憶體大小則等

於--使用者從 command line 來更改 Vintr 層、Channel 層、Event 層總共需要的大小，例如:insomd tmman_PNX1005.ko **mb_size**=0x200000 表示配置區塊有 2MB 或是使用預設的 64KB，再加上環境中額外針對區塊對齊預留的 64KB 及 1KB 的 TMManSharedStruct、HAL、NameSpace 空間。

(式 1)

```
void *dma_alloc_coherent(struct device *dev,size_t size,dma_addr_t
*dma_handle,int flag)
```

說明:這個函式會把指向這塊記憶體開頭的 physical address 存在 dma_addr_t，同時回傳其相對應的 kernel address 給驅動程式，因為 ARM 和 x86 架構的 CPU 不同的地方在於他不保證能 cache coherency，所以我們利用這個函式本身就會配置出 noncached 的記憶體這個特性來創建 shared memory。

有了共享記憶體空間後，接著就要依照欲配置的共享記憶體區塊所需大小依序去分配。一開始 dma_alloc_coherent 函式回傳的共享記憶體虛擬記憶體起始位址與實體記憶體位址，我們就分別儲存於(表 3.12)TMManDeviceObject 結構的 SharedData 變數與 TMManSharedAddress 變數。之後要配置 TMManSharedStruct 區塊時，就將共享記憶體的虛擬記憶體起始位址加上 TMManSharedStruct 區塊大小(220 bytes)，得到下一個區塊的虛擬記憶體起始位址，如此，從前一個區塊起始位址到下一個區塊起是位址就是前一個區塊可使用的共享記憶體空間。

為了取得區塊在共享記憶體的實體記憶體位址，我們利用 2-2 的位址轉換關係，先利用 `virt_to_bus((void *)PAGE_OFFSET)` 求得 kernel 的虛擬記憶體起始 3G 對應到的 bus 起始位址 0，表示 bus address 等於 physical address，於是，我們就把區塊虛擬記憶體起始位址減掉共享記憶體的起始位址找出區塊起始位置對於共享記憶體起始位置的 Offset，然後將共享記憶體的 Bus address 起始位址加上 Offset，即為此區塊的實體記憶體起始位址。然後我們就可以把區塊的虛擬記憶體位址與實體記憶體位址儲存於(表 3.12)裡對應的變數去。

之後要配置下一個區塊，就從分配上一個區塊時得知的下一個區塊的虛擬記憶體起始位址開始，再加上要分配的區塊大小，又可以得到下一個區塊的虛擬記憶體起始位址，然後又可以從虛擬記憶體位址換算出實體記憶體位址，我們以下表依照順序，列出 TMMan 驅動程式劃分共享記憶體的區塊與其大小。

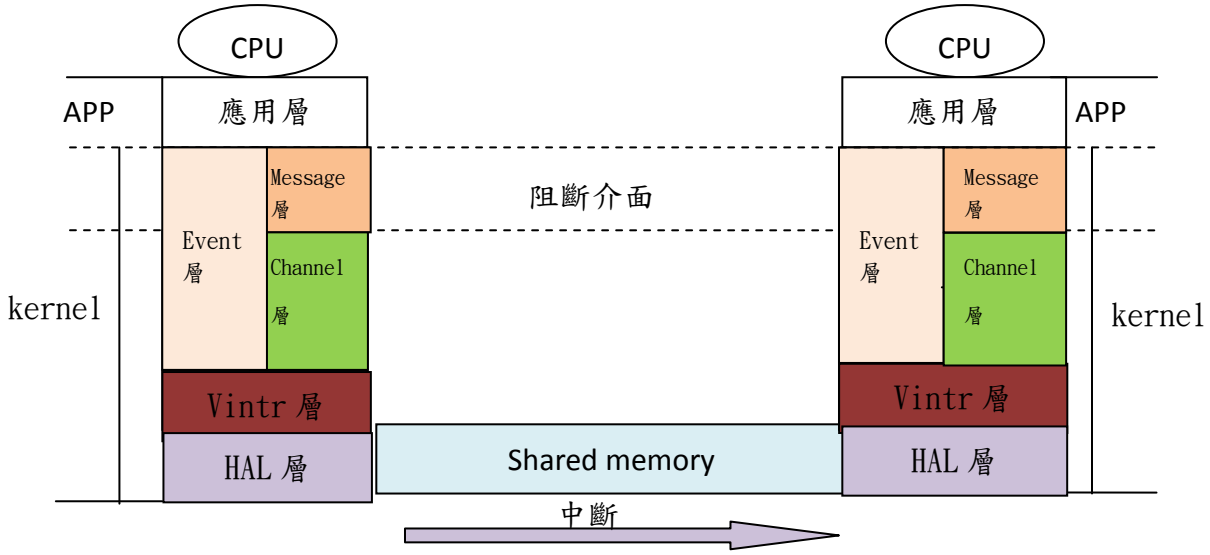
(表 4.1) 共享記憶體各區塊大小

區塊名	大小
TMManSharedStruct	220 bytes
HAL	48 bytes

NameSpace	(表 3.11)儲存的 NameSpace 區塊個數乘上一個 NameSpace 小區塊的大小(24bytes)
Vintr	(表 3.11)儲存的 Vintr 區塊個數乘上一個 Vintr 小區塊的大小(24bytes)，還要在乘上 2，因為是雙向的
Channel	子 Channel 大小乘上 2。 一個子 channel 大小等於(表 3.11) 儲存的 Mailbox 個數乘上一個 mailbox 大小再加上 ChannelMailboxControl 結構的大小(12 bytes)。而一個 mailbox 大小又是一個 packet 結構大小(20 bytes)加上一個 channelPacket 結構大小(8 bytes)。
Event	(表 3.11)儲存的 Event 區塊個數乘上一個 Event 小區塊的大小(8 bytes)，還要在乘上 2，因為是雙向的

4-3 溝通時的軟體架構

藉由4-2節共享記憶體的區塊劃分，我們在軟體上就設計了HAL、Vintr、Channel、Event層分別負責控制HAL區塊、Vintr區塊、Channel區塊與Event區塊，透過這幾層對共享記憶體的控制來完成4-1節溝通模式裡OS層底層的實質溝通動作，所以兩種溝通模式的分層結果則如(圖4.7)，其中OS層與應用層的介面，在訊息傳遞模式下是靠Message層的queue buffer做為packet的終點，而且以semaphore來作為接收端應用層與OS層底層運作的阻斷介面，在事件觸發同步模式下則是靠Event層裡的semaphore作為應用層與OS層底層運作的阻斷介面，於是不管哪種溝通模式，等待端的應用層應用程式一開始都會被阻斷介面block住，等到OS層底層溝通動作做完，才會去signal阻斷介面，啟動應用程式繼續運作。



(圖 4.7)OS 層內部的分層

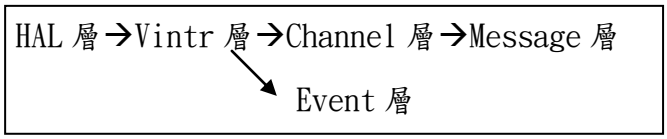
4-3-1 分層設計

為了控制共享記憶體上各種區塊，在軟體上則設計了**Vintr層**、**Channel層**、**Event層**，分別控制共享記憶體上的三個區塊。在Vintr層，主要就是到Vintr區塊根據事件代號去設定其對應的flags，來判斷要做同步的動作還是做傳送資料的動作；而傳送端的Channel層會去檢查channel區塊是否還有目前未被使用或是已經使用完的mailbox，有的話就將應用層傳給OS層的packet複製到共享記憶體上，並在傳送packet時順便挾帶傳送packet的訊息交換service代號；接收端的Channel層則會判斷是否有新的mailbox傳送過來，有的話就從共享記憶體複製packet到kernel操作環境，接收packet時還一併讀取service代號來決定packet要塞入哪個service在Message層的queue buffer；Event層就是到共享記憶體的Event區塊根據Event service代號，去設定對應的小區塊裡的flags，以分辨要對哪一個service的semaphore阻斷介面做signal的動作。

Message層只有傳送訊息模式才會使用，其中會為每個訊息傳遞的service創建一個queue buffer，利用這個queue buffer我們可以用來調節接收端處理packet與傳送端傳送packet的不同速度。每個queue buffer是一塊可以儲存64個packets的空間，空間的使用控制靠的是Message層用來控制queue buffer所儲存的寫入指標與讀取指標，每次要對queue buffer進行讀寫，就以一個packet大小為單位，讀完就把讀取指標加一，寫完就把寫入指標加一，下次要進行讀寫就知道要從哪個queue buffer空間裡的packet進行讀寫。不過寫入者在填入packet前必須先判斷queue buffer是否還有目前沒被使用或已使用完的packet可供填入，而讀取者要對queue buffer讀取packet時，則要判斷是否有新的packet填入目前讀取指標指到的packet空間。基本上這個queue是一個FIFO且可重複使用的buffer，只要讀寫指標讀或寫到底了，就把指標歸0。而且因為每個service只會有一個queue buffer，所以service代號就對應到一個queue buffer;為了避免queue buffer中還沒有塞入新訊息，應用程式就對他做讀取的動作，所以我們在訊息層還為每個service儲存阻斷介面(4-4-2-1)，應用程式先被block在做讀取動作之前，直到有訊息塞入queue之後，kernel才會signal這個阻斷介面以啟動應用程式的運作。

HAL層其實就是直接控制共享記憶體的HAL區塊，檢查區塊中的flags以避免ARM端清除中斷暫存器時，DSP端正在產生中斷。

了解各層功能後，對扮演接收角色的一端來說，執行順序為如下，扮演傳送角色的一端則以反方向進行。

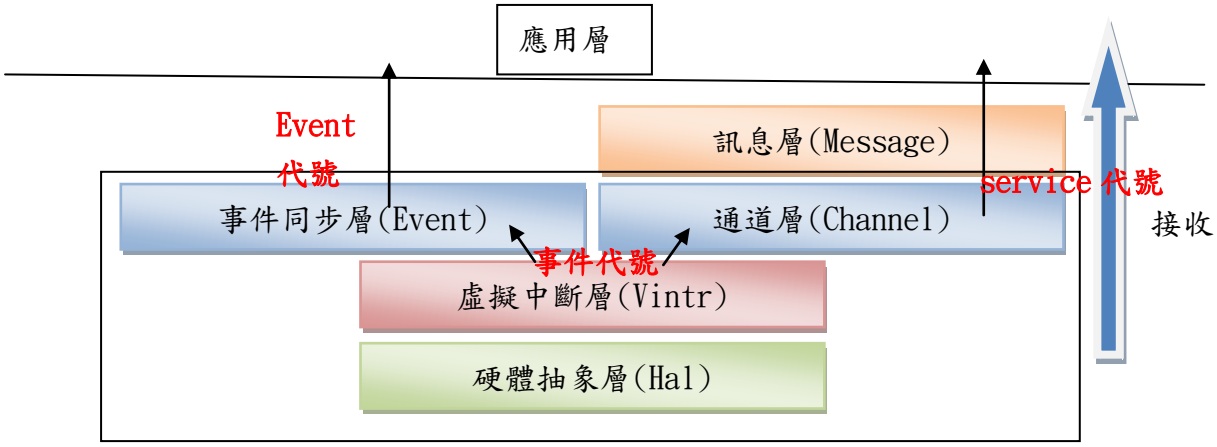


當應用程式某個 service 要使用共享記憶體進行溝通前，兩端要先分配好各層會用到的 service 代號或事件代號，如果是訊息交換模式的 service 就把 service 代號儲存於 Message 層與 Channel 層，如果是事件觸發模式的 service 就把 service 代號儲存於 Event 層；然後 Channel 事件與 Event 事件代號則會儲存於 Vintr 層。在各層也儲存了身為接收角色在處理中斷時，要往上傳送一層所需的 ISR，在 HAL 層會儲存進入 Vintr 層的 ISR，在 Vintr 層則以事件代號分別對應到兩個不同的 ISR，以進入不同的事件處理程序，在 Channel 層則有一個 ISR 依照訊息交換 service 代號，把底層訊息交換收到的 packet 插入 service 代號對應的 queue buffer 中，而 Event 層中 service 代號已經對應到一個 semaphore 阻斷介面，所以直接請 kernel 對這個 semaphore 做 signal 即可。

該 service 如果扮演傳送角色的一端，從應用層往下，如果是事件觸發同步模式的 service，就會進到 Event 層，到共享記憶體 Event 區塊中，依照 service 代號找到對應的 Event 小區塊，設定其中的 flags，接著進入 Vintr 層，將 Event 事件代號對應到的 Vintr 小區塊的 flags 做設定；如果是訊息傳遞模式的 service，就會將 packet 直接傳入 Channel 層，依照共享記憶體的 Channel 區塊中的寫入編號指到的 mailbox 填入 packet 與 service 代號，接著進入 Vintr 層，將 Channel 事件代號對應到的 Vintr 小區塊的 flags 做設定。最後都進入 HAL 層去設定中斷相關暫存器，對對方產生中斷。

每當扮演接收角色的一端(圖 4.8)，接收到中斷後，便會從 HAL 層開始，檢查共享記憶體的 HAL 區塊中的 flags，讓 ARM 端接收中斷後想要清除中斷暫存器的設定前，確認 DSP 端目前沒有正在對 ARM 端發出中斷；然後由儲存於 HAL 層的 ISR 進入 Vintr 層，檢查共享記憶體 Vintr 區塊中的每個小區塊，只要遇到 flags 有被設定的就代表該小區塊對應到的事件發生了，於是使用該小區塊對應到的事件代號來決定要使用的 ISR，使用完後要清掉小區塊中的 flags 設定，如果是 Channel 事件代號對應到的小區塊有被設定，就使用進入 Channel 層的 ISR，如果 Event 事件代號對應到的小區塊有被設定，就使用進入 Event 層的 ISR；假如進入 Channel 層，Channel 層便會到共享記憶體的 Channel 區塊，檢查讀取編號是否與寫入編號相同，不同的話表示有新的 packet 傳進來，於是依照讀取編號指到的 mailbox 讀取 packet 與 service 代號，然後把讀取編號加一，表示下次從更新後的讀取編號表示的 mailbox 作讀取，然後依照 service 代號使用進入 Message 層的 ISR 將 packet 塞入 service 代號代表的 service，它所使用的 queue buffer 中，填入時使用控制 queue buffer 的讀取編號與寫入編號，檢查寫入編號加一的值是否讀取編號，如果是，就表示 queue buffer 中所有 packet 空間都在等待接收，已經沒有空間可以寫入，就會結束這次傳送但是不喚醒應用程序，不然的話就會依照寫入編號指到的 queue buffer 空間將 packet 填進去，然後 kernel 就會直接 signal Message 層的阻斷介面，啟動應用程式繼續執行接

收的動作；如果是進入 Event 層，就會到共享記憶體的 Event 區塊，去檢查每個小區塊中的 flags 是否有被設定，只要發現有被設定的，就用該小區塊對應的 Event service 代號，由 kernel 去 signal 它所代表的 semaphore 結構，啟動應用程序的運作。



(圖 4.8)訊息傳遞分層架構

4-3-2 分層溝通機制

了解 ARM 端對共享記憶體的劃分與溝通管道的分層設計後，本節先假設已建好溝通管道後，傳送端與接收端在做訊息傳收或兩端同步時，會使用哪些 user mode library 來啟用 OS 層的溝通機制，然後 OS 層是如何由分層架構來使用共享記憶體的各個區塊，完成事件與服務的區分、訊息傳遞或同步。了解各分層存在的意義與功能後，4-4 節再說明實作上，進行溝通以前如何建置溝通管道，還有使用的 Message 層阻斷機制，並說明各分層會使用哪些結構來記錄各種事件或服務的代號以追蹤到應該呼叫的 ISR，然後應用程式如何使用 user mode library 來創建溝通管道與傳送訊息或進行同步。

不管 ARM 還是 DSP 為訊息傳送端或事件觸發同步時的 Eventsignal 端，基本上 OS 層溝通時所做的判斷與運作是相似的，因此使用共享記憶體的方式也大同小異，唯有 HAL 在做有關暫存器設定或呼叫 ISR 的動作有些不同，這就是 OS 的議題了，所以我們在此均以 ARM 的角度出發，探討訊息傳遞時 ARM 為傳送端或接收端、事件同步時為 Eventsignal 端或 EventWait 端其 OS 層內部使用分層進行溝通的狀況，並以(圖 4.9)、(圖 4.10)表示傳遞訊息、事件同步時各層與共享記憶體的互動。

4-3-2-1 訊息傳遞的分層溝通機制

傳送端

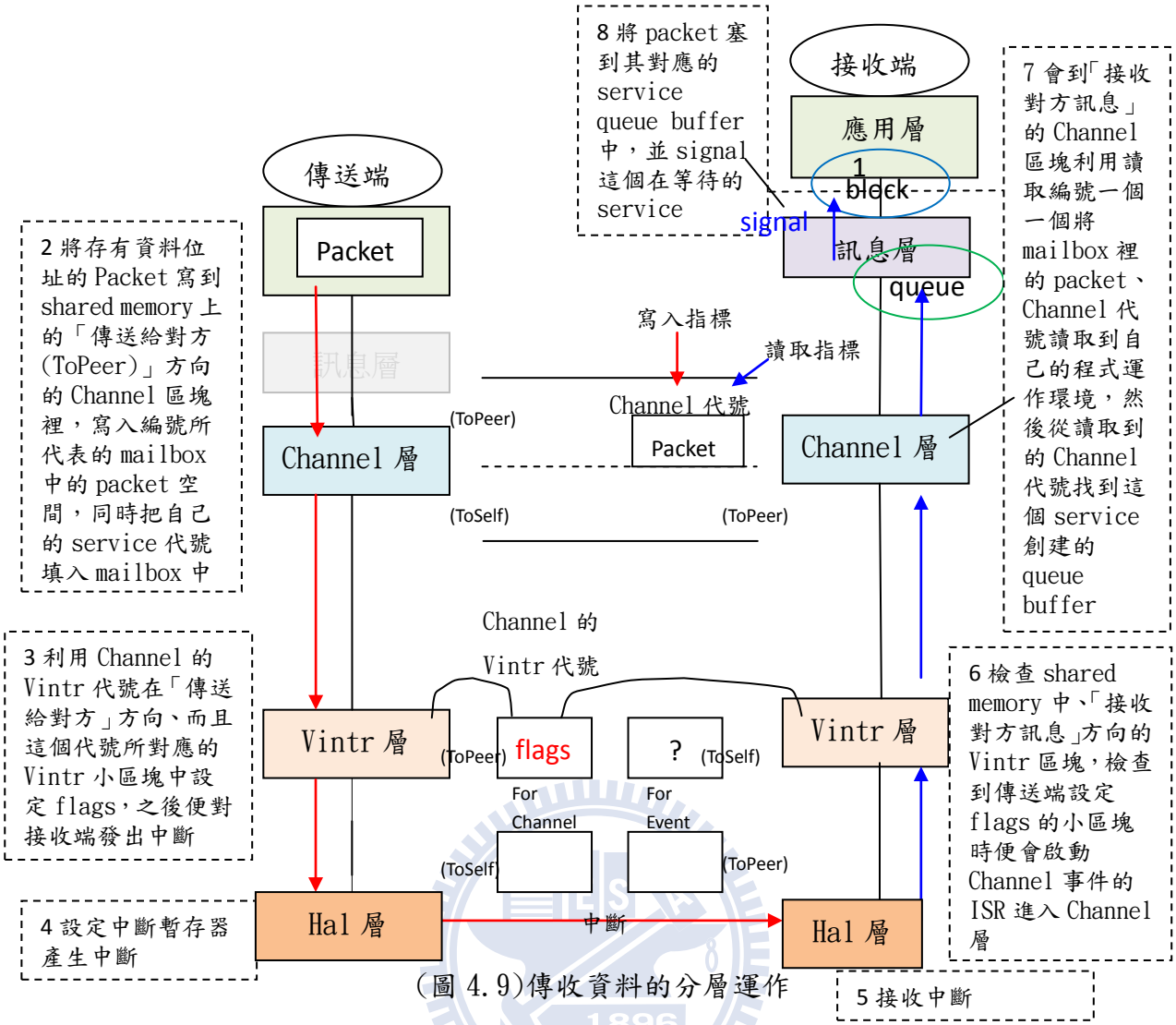
身為傳送訊息的主動端，每當有 packet 需要傳送，應用程式裡需要傳送資料的應用服務(例如第五章的 CRT 服務)就會使用 tmmanMessageSend 函式，透過驅動程式中的 message 模組(3-4-4)的 messageSend 函式開始一連傳的訊息傳遞動作。首先，驅動程式會

直接將 packet 傳入 Channel 層，在 Channel 層，驅動程式於是會將 packet 複製到共享記憶體上的「傳送給對方」方向的 Channel 區塊裡、寫入編號所代表的 mailbox 中的 packet 空間，同時把自己的 service 代號填入這個 mailbox 中，如果有多筆資料要寫入，寫入編號就不斷加一指到下一個 mailbox，然後把資料位址與 service 代號依照順序填進去。接著,Channel 層就會激起中斷機制,首先透過 vintrGenerateInterrupt 函式進入 Vintr 層，在 Vintr 層就利用代表 Channel 的 Vintr 代號在「傳送給對方」方向、而且這個代號所對應的 Vintr 小區塊中設定 flags，而後 Vintr 層會使用 halGenerateIneterrupt 函式進入 HAL 層，HAL 層於是會利用第三章驅動程式初始時在 HAL 結構儲存的 MMIO aperture 的 kernel address，加上特定的 shift(0x100820)對應到 IPENDING 暫存器，加以設定，對接收端發出中斷。

接收端

接收端先使用tmmanSyncCreate函式於kernel中創建一個semaphore結構，並將semaphore結構位址在tmmanMessageCreate時存入Message層中，以Message service代號來連結這個service與使用的semaphore結構，所以每次service要接收訊息時，就會先利用tmmanSyncWait函式去block這個semaphore，使得應用層這個傳遞訊息service無法作讀取queue的動作。直到接收端收到中斷後，由HAL層先clear掉中斷相關暫存器，然後使用第一個註冊於OS的ISR，進入Vintr層，此時接收端會去檢查共享記憶體中、「接收對方訊息」方向的Vintr區塊，檢查到傳送端設定flags的小區塊時便會啟動Channel事件的ISR進入Channel層，進到Channel層後便會到「接收對方訊息」的Channel區塊利用讀取編號一個一個將mailbox裡的packet、Channel代號讀取到自己的程式運作環境，然後從讀取到的Channel代號找到這個service創建的queue buffer，將packet塞到其對應的service queue buffer中，並signal這個semaphore以喚醒在等待的service，繼續往下執行tmmanMessageReceive來做讀取queue buffer的動作。

從Vintr層引起的Channel事件處理完後，他會繼續檢查下一個Vintr小區塊，看看是否其中的flags也有被設定，有的話表示傳送端也有送來同步事件的要求，於是接收端便需要處理同步(請看下一節的EventWait端)。



(圖 4.9)傳收資料的分層運作

4-3-2-2 事件同步的分層溝通機制

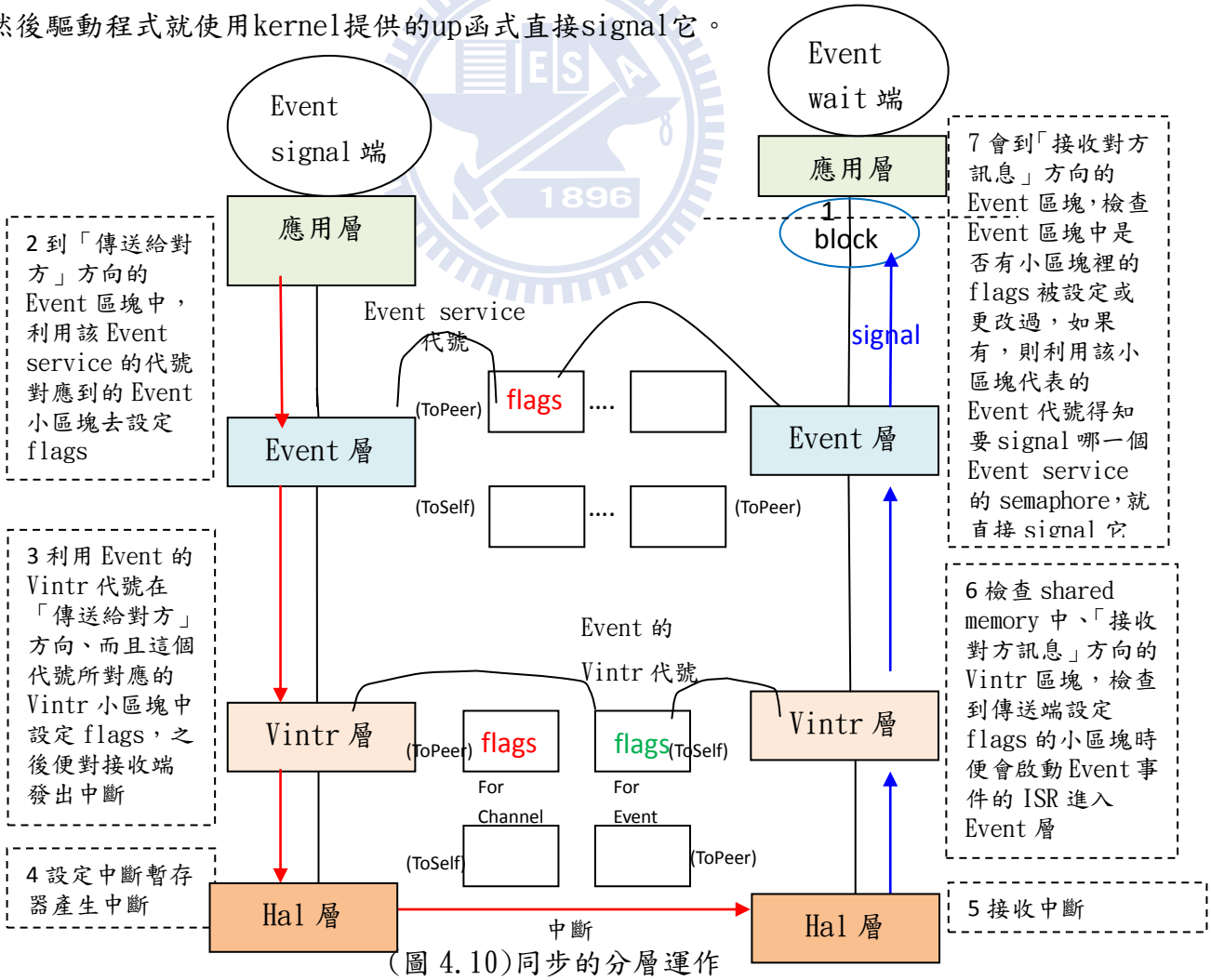
EventSignal端

身為要去喚醒別人的主動端，如果想要去喚醒別人的其中一個在等待的事件同步service，例如「等待DSP 端程式初始化完畢」，那麼本地端的應用程式會使用tmmanEvent函式，透過驅動程式中的Event模組的eventSignal函式，到「傳送給對方」方向的Event區塊中，利用該Event service的代號對應到的Event小區塊去設定flags，而後便由Event層激起中斷機制，如同傳遞訊息的分層溝通機制，首先透過vintrGenerateInterrupt函式進入Vintr層，在Vintr層就利用代表Event的Vintr代號在「傳送給對方」方向、而且這個代號所對應的Vintr小區塊中設定flags，而後Vintr層會使用halGenerateIneterrupt函式進入HAL層，HAL層於是會利用第三章驅動程式初始時在HAL結構儲存的MMIO aperture的kernel address，加上特定的shift(0x100820)對應到IPENDING暫存器，加以設定，對接收端發出中斷。

EventWait端

每個事件同步service都會先使用tmmanSyncCreate函式，於kernel中創建一個semaphore結構，並將semaphore位址儲存到Event層中，以Event service代號來連結這個service與使用的semaphore結構，所以該事件同步service每次利用tmmanSyncWait函式去block這個semaphore後，就要等著對方使用上一段的方式傳送signal消息過來，然後以Event service代號去signal這個也註冊於Event層的semaphore。

當EventWait端收到中斷後，便與訊息傳遞時相同，由HAL層先clear掉中斷相關暫存器，然後使用第一個註冊於OS的ISR，進入Vintr層，此時EventWait端會去檢查共享記憶體中、「接收對方訊息」方向的Vintr區塊，從裡頭的第一個小區塊開始檢查，如果Channel事件使用的是第一個區塊，而且發現這個區塊的flags有被設定，就會先處理傳遞訊息的動作，不然，就會再檢查第二個小區塊，如果這個小區塊就是EventSignal端設定flags的小區塊時，便會啟動同步事件的ISR進入Event層，進到Event層後便會到「接收對方訊息」的Event區塊，檢查Event區塊中是否有小區塊裡的flags被設定或更改過，如果有，則利用該小區塊代表的Event service代號得知要signal哪一個Event service的semaphore，然後驅動程式就使用kernel提供的up函式直接signal它。



(圖 4.10)同步的分層運作

4-4 分層溝通機制的實作

上面我們已經說明了分層溝通機制，為了實作每層功能，我們必須在每層儲存操作共享記憶體區塊的區塊起始位址與 service 或事件代號，這樣才能透過軟體來控制硬體上的共享記憶體空間，因此我們為每層設計以 ManagerObject 與 Object 為名的資料結構，儲存溝通所需資源，然後利用這些資料結構儲存的資訊對共享記憶體進行控制來達到溝通的目的。

我們為每層提供一個 ManagerObject 資料結構來儲存每層所對應的共享記憶體區塊的起始位址，還有各個區塊更細部的劃分位址資訊(例如 Channel 區塊)，讓每層在控制對應的共享記憶體區塊時可以直接取用到想要控制的位置，而且 ManagerObject 也建立一個表單來管理該層多個 Object 資料結構，該層的每個 Object 資料結構都儲存了應用程式 service 在溝通時，在階層裡的辨識代號，如果在 Vintr 層，就有 Channel 使用的 vintrObject 結構與 Event 使用的 vintrObject 結構,分別儲存 Channel 事件代號與 Event 事件代號,Message 層裡為每個 service 創建的 Object 結構則儲存了每個 service 的代號，以此類推，而每層的 Object 結構中也儲存了服務該 Service 或啟動事件所需私有資訊，包括 ISR、ISR 參數位址、queue buffer、阻斷介面等，像在 Message 層中，每個 service 的 MessageObject 結構就儲存了服務該 service 所需的阻斷介面、queue buffer；Vintr 層則為 Channel 事件與 Event 事件提供不同的 ISR,在接收到中斷時使用。因此透過代號，我們可以在各層 Object 結構中找到該服務或事件運作所需資源。

有了 Object 結構與 ManagerObject 結構後，我們就可以透過 Object 結構裡的 ISR 為接收端建立訊息接收運作路徑，而 ManagerObject 結構中也會儲存下面一個階層的 Object 結構，提供傳送端傳遞訊息的運作路徑。

在 TMMan 驅動程式裡，我們設計了 message 模組讓驅動程式使用 MessageObject、Object 結構來操控 Message 層的運作，設計 channel、Event 模組讓驅動程式使用 ChannelManagerObject/ChannelObject 資料結構、EventManagerObject/EventObjt 資料結構來操控共享記憶體的 Channel 區塊、Event 區塊，以此類推，這些模組都是使用 Object 與 ManagerObject 結構儲存的資訊來藉由操作共享記憶體來完成 OS 層訊息的交換，於是像 message 模組中的 messageSend 函式就是用來實作訊息傳遞模式下,傳送端的運作機制，由上面我們知道傳送時 Message 層沒有做什麼設定，所以 packet 就直接交由 channel 模組的 channelSend 函式，去運作 channel 層的傳送功能(將 packet 複製到共享記憶體的 channel 區塊)。

每個模組提供的功能都會以 ioctl 某個 command 代號表示，所以我們可以再以這些模組為基本功能，藉由第三章驅動程式的設計，搭配其他偵錯機制、設計更上層的傳遞訊息

函式庫、事件同步函式庫給應用程式使用，所以最後我們可以用 tmmanMessageCreate 函式為訊息傳遞模式的 service 創建溝通管道，然後使用 tmmanMessageSend 將 packet 傳輸到另一端，另一端 OS 層傳輸完，將 packet 插入 Message 層的 queue buffer 後，應用層才被啟動,然後使用 tmmanMessageReceive 從 Message 層的 queue buffer 中讀取 packet。

另外我們對於訊息傳遞模式的 service 或事件觸發同步的 service，則使用 tmmanSyncobjCreate 於 kernel 創建一個 semaphore 結構來作為 service 的阻斷介面，之後扮演等待角色的一端就可以使用 tmmanSyncobjWait 將 semaphore block 住，使得應用程式被阻斷，直到 OS 層溝通結束，kernel 將這個 semaphore signal 後應用程式才會繼續執行下面的 packet 接收或是執行剩下的應用程式區段。

本節會先介紹每層內部使用到的組成元件，然後說明如何使用 user mode library 來建立 ARM 為接收端時所使用的阻斷機制、溝通管道時需要在各層建立的資源與完成事件同步、訊息傳遞的動作。

4-4-1 組成各層的元件

上述的階層觀念在軟體實作時,我們在 kernel 中,利用一個統管結構(ManagerObject)與多個小 Object 結構來組成 OS 層中的一個階層，而且為了要實作各層運作順序，我們會從傳送端/Eventsignal 端與接收端/EventWait 端的角度，分別探討使用哪些機制來建立層與層之間的銜接。

Object 結構

每個 Object 結構就是以一個資料結構，藉由儲存 service 代號或事件代號(例如:ChannelID、VintrID、EventID),以及完成這個 service 或事件需處理的函式位址(ISR)來代表一個 service 或事件。這些代號於是可以用來決定要設定哪一個共享記憶體區塊中的 flags、作為 service 的區分、呼叫代號表示的 Object 中儲存的 ISR，另外根據不同 service 需要使用到的資源，例如同步需要使用的 semaphore 結構位址與訊息傳遞模式下用來控制 queue buffer 的資料結構位址等，也會分別存到該層 Object 中。

因此每次要運作一個 service 前，應用層會利用 user mode library 提供的 tmmanEventCreate 函式為同步 service 到 Event 層創建 EventObject，或使用 tmmanMessageCreate 函式到 Message 層及 Channel 層分別創建 MessageObject 與 ChannelObject，同時儲存該種服務的 ServiceID 與該種服務所需資源於 Object 結構中，至於不管什麼 Service 都會用到的 VintrObject(for channel/event)則可以在一開始註冊溝通的驅動程式時就創建好(有關溝通管道創建請看 4-4-2-3)，溝通開始時，傳送端便可

從應用層開始，利用每層 Object 裡的資訊來做正確的 service 或事件處理。

ManagerObject 結構

由於每層的 Object 結構不只一個，所以我們使用一個稱為 **ManagerObject 的資料結構**來管理這些 Object，而這些 Object 結構因為代表了一個事件或 service，因此我們就利用 service 或事件的 ID 當作 ManagerObject 管理它們時所用的編號。另外，在每個階層中處理事情的時候，我們希望可以直接知道該層控制的共享記憶體區塊虛擬記憶體起始位址，或是這些區塊裡面更細部的一些資料存放位址(例如區塊中分出來的兩個方向的區塊位址)，然後使用它，於是我們就將這些位址資訊也儲存在 ManagerObject 裡，在 Vintr 層與 Event 層中藉由這些位址資訊與它們的 ID，我們還可以知道要使用到 Vintr/Event 區塊中哪個小區塊儲存的 flags。

因為 ManagerObject 結構中必須儲存取該層控制共享記憶體的區塊虛擬記憶體起始位址，而且也是 TMMan 驅動程式在運作兩端 CPU 的溝通時要一直存在的一個管理結構，所以各層 ManagerObject 結構會在驅動程式初始化時就創建好，並把初始化在對共享記憶體進行劃分時(4-2-2)得到的區塊虛擬記憶體起始位址儲存到 ManagerObject 裡，當然 ManagerObject 中就可以從這個起始位址開始，對該層控制的區塊記憶體記錄更多細部的區域起始位址，例如以 ChannelManagerObject 來說，他會以 Channel 區塊的虛擬記憶體起始位址開始當做一個子 channel 的起始位址，然後加上一個子 channel 所需空間後得到的位址，則為下一個子 channel 的起始位址，當然利用各個子 channel 的起始位址，又可以在必要的時候得到如 mailbox 的位置等等。

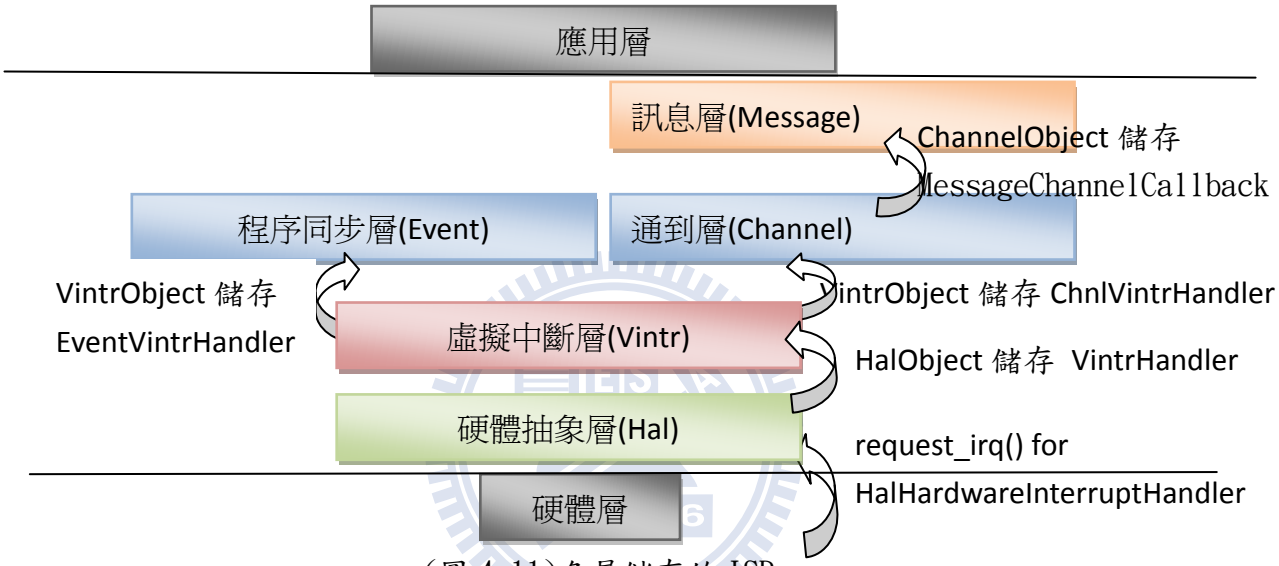
接收端/EventWait 端往上的各層銜接--ISR

接收端或 EventWait 端在剛接收到中斷時，硬體中斷並沒有方向性，也就是說光只有中斷並不知道到底要做的是哪一種處理，既然 OS 層的溝通已經分層，而且已從 4-3-2 節得知接收端或 EventWait 端的分層處理順序，於是我們以與硬體中斷直接相連的 HAL 層 ISR(halHardwareInterruptHandler)做為導火線，只要導火線一點燃，就會啟動後續軟體設計上的 ISR，於是我們為每一層設計一個 ISR，做為每層每層之間能夠銜接的關鍵，同時也用來確定訊息處理路線。

於是，接收端或 EventWait 端收到中斷後，首先會經由 3-3 節使用 request_irq 函式註冊的 halHardwareInterruptHandler 啟動一連串 4-3-2 節提到的接收端/EventWait 端的處理，首先這個 ISR 做完 HAL 層該做的處理後，就會呼叫註冊於 HAL 層的 VintrHandler，進入 Vintr 層作事件判斷，接著 Vintr 層會依照判斷出的事件，呼叫註冊於 Vintr 層的

chnlHandler 或 eventHandler，進入 Channel 層作訊息複製或 Event 層對 Event service 代號對應的 semaphore 做 signal。於 Event 層與 Channel 層處理完後，Channel 層還會繼續呼叫 messageChannelCallback 函式將收到的 packet 塞入 service ID 對應到的 queue buffer 中，並對應用程式發出中斷。

然而要能使用這些 ISR，這些 ISR 在創建溝通管道時(4-4-2-2)，就將上層提供的 ISR 位址與 ISR 所需參數位址儲存於下一層的 object 結構中(圖 4.11)。由於為各層設計出的 ISR 主要就是帶領下層的人進入本層，然後在本層對對應的共享記憶體區塊作控制，而該區塊各種位址資訊都是儲存於該層的 ManagerObject 結構中，因此為各層設計出的 ISR 所需參數均為該層的 ManagerObject 結構位址。



(圖 4.11)各層儲存的 ISR

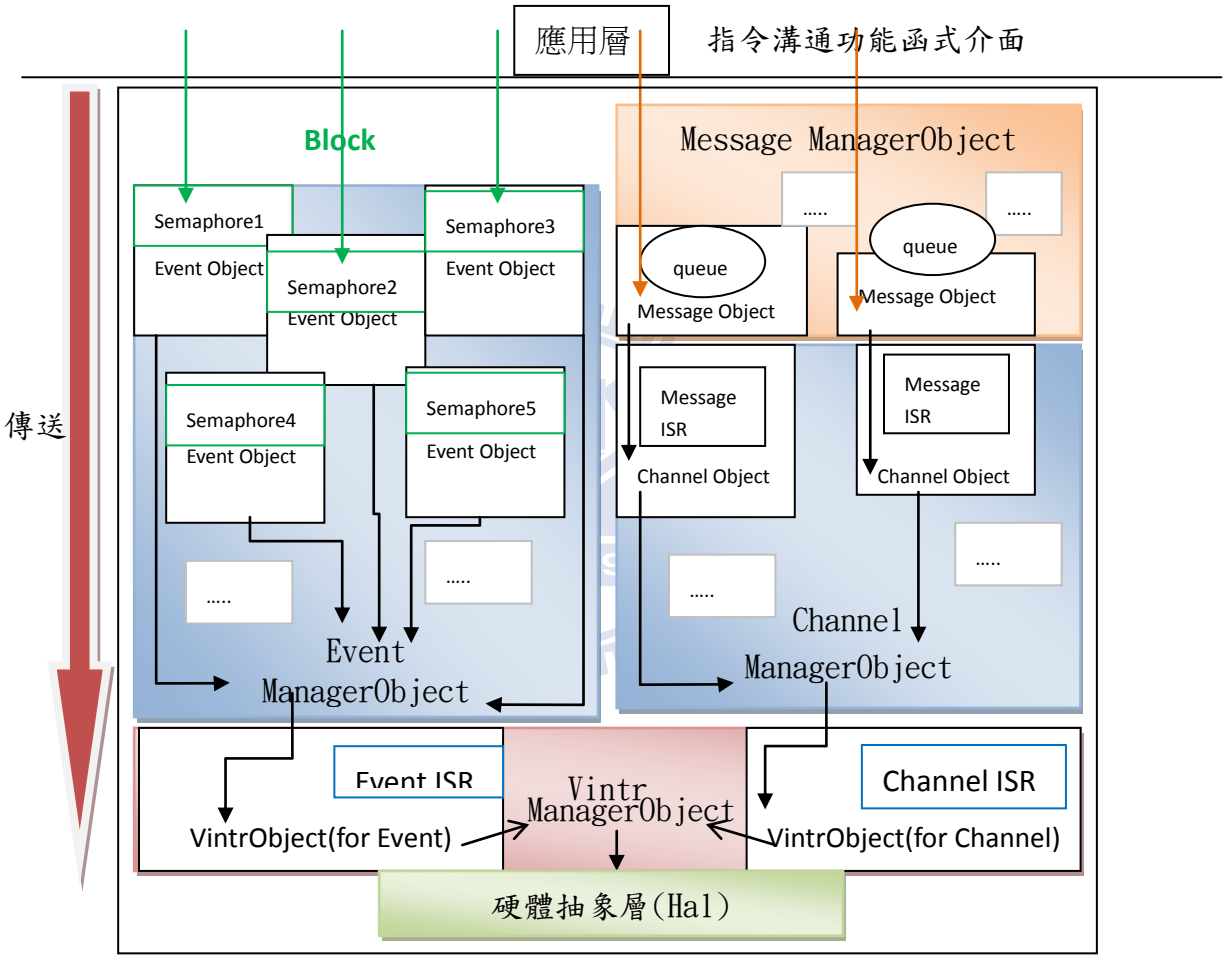
傳送端/Eventsignal 端往下的各層銜接

傳送端或 Eventsignal 端是由應用層向下，經過各層來使用共享記憶體，為了讓訊息處理路徑可以依照 4-3-2 節所談，往下層的銜接動作，我們會藉由 ManagerObject，讓它儲存該種 service 運作時會使用到的下一層 Object 結構位址，這樣上層的人處理完，就可以利用下層的 Object 結構位址進入下一層，而且 Object 結構也要儲存同層的 ManagerObject 位址，這樣進入下層後，就可以透過這個 Object 找到 ManagerObject，然後利用 ManagerObject 中儲存的共享記憶體位址資訊，來控制共享記憶體。

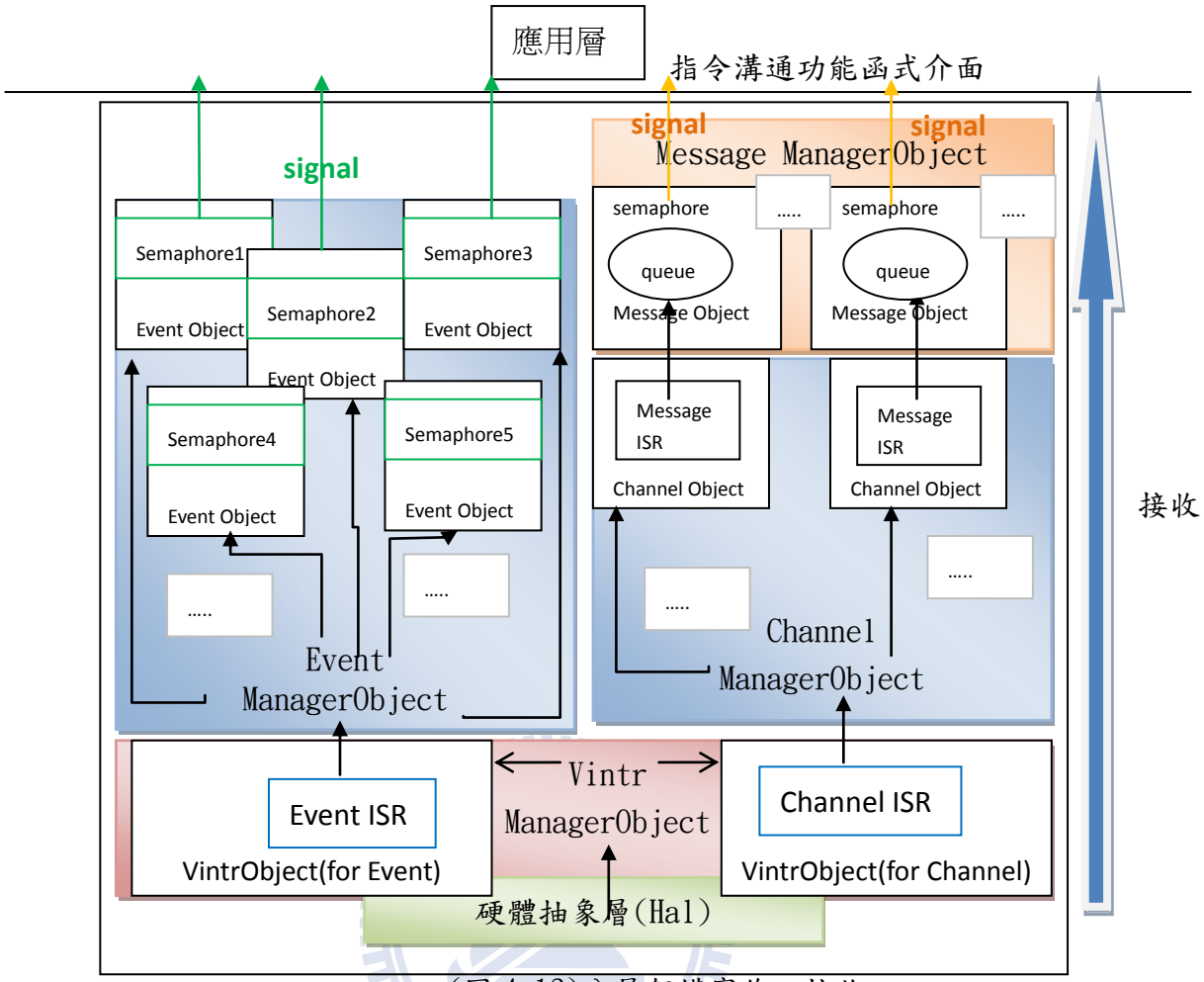
(圖 4.12)為傳送方向的分層實作，(圖 4.13)為接收方向的分層實作。每張圖左半部為事件同步的運作，右半部則為傳遞訊息的運作，每層均由 ManagerObject 與 Object 組成，箭頭表示連接或呼叫順序的關係，我們分為傳跟收來展示各層之間的銜接。

傳:由 Object 進入各層,然後從 Object 中找到 ManagerObject 位址,再搭配 ManagerObject 中的資訊運作。

收:由 ManagerObject 進入各層，然後利用代號找出對應的 Object，然後利用 ManagerObject 提供資訊處理該層的運作，最後使用 Object 儲存的 ISR 進入上層



（圖 4.12）分層架構實作—傳送



(圖 4.13)分層架構實作—接收

說明:指令溝通功能函式介面指的是第三章提到的 user mode library，它所提供的傳遞訊息函式庫與同步函式庫，這些函式庫的使用將於 4-4-2 節詳述。

4-4-2 應用層使用 user mode library 完成 ARM 端訊息接收與事件同步

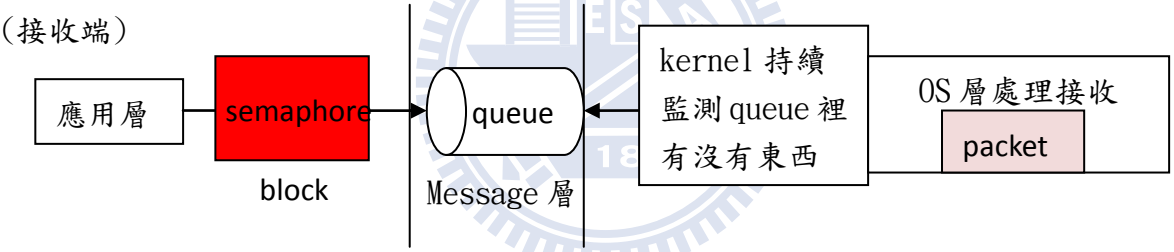
為了讓應用層可以使用底層的溝通機制完成訊息交換，我們把每層操控記憶體的動作設計成一個模組，每個模組裡都包含有創建溝通所需資源的函式、傳收訊息需要的函式以及進入到上一層繼續對共享記憶體做判斷的 ISR 函式，然後我們會把這些模組函式(3-4 節)設計到 ioctl driver handler(3-1-3 節)中，讓應用層可以使用系統呼叫來使用到驅動程式裡的模組函式。我們更在第 3-4 節提到可以在系統呼叫外包裝一層具功能性的函式，讓應用層的人可以直接使用這些功能性函式，間接使用驅動程式中的模組函式來做訊息的交換。

4-4-2-1 訊息傳遞時應用層與 OS 層的阻斷介面

由於區塊上所作的訊息交換是在kernel中運轉的，應用層是看不到的，所以應用層與OS層之間需要一個queue buffer來儲存接收到的訊息，這個queue buffer就位於Message層。然而接收端為了避免應用程式在訊息還未抵達時就對queue 讀取資料，所以應用層會在讀取queue的動作前先向OS請求把自己block住，這樣的觀念會應用在第五章ARM端CRT服務接收OS層的packet時。

要實現block機制，我們會使用第三章的tmmanSyncobjCreate函式，先向OS請求，在kernel中建立一個semaphore結構，這個結構一開始會被初始為0，並在創建溝通管道時(4-4-2-2節)把這個結構位址儲存到Message層中該service使用的MessageObject結構裡，每次應用層想要讀取queue buffer中的packet時，就要先取得semaphore才行。

於是，應用層在使用tmmanMessageReceive函式從queue buffer讀取packet前，應用程式必須使用tmmanSyncobjWait函式，進入kernel後利用kernel提供的down_interruptible()函式試著去獲取semaphore，但是由於semaphore一開始就被初始為0，表示沒有資源可以使用了，不能繼續執行應用層的程式，於是應用程式就會被block住，而這個程式就會暫時處於wait的狀態。

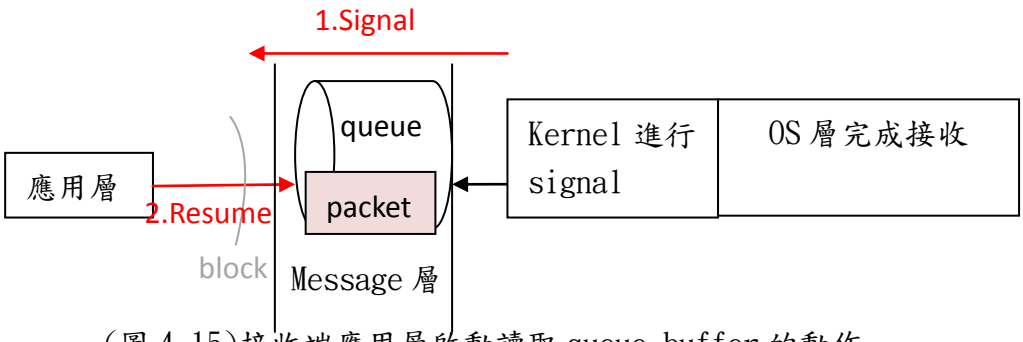


(圖 4.14)接收端應用層等待讀取 queue buffer

而後OS層每次做完訊息交換，並將收到的訊息，透過該service控管queue buffer空間的寫入指標，其加一後的值如果不等於目前的讀取指標，表示queue buffer還沒有被塞滿，所以就可以將packet複製到寫入指標指到queue buffer的packet空間，之後OS層的驅動程式就會直接透過kernel提供的up()函式去對這個service的MessageObject結構儲存的semaphore結構加一，做signal的動作，此時之前被block住的程式部分就可以繼續執行，於是應用層才可以繼續執行之前被block住的應用程式，使用tmmanMessageReceive函式，進入OS層裡的Message層，透過控制queue buffer空間的讀取指標，如果目前讀取指標不等於寫入指標，表示有新的packet存於讀取指標指到的queue buffer空間，所以就從讀取指標指到queue buffer中的packet空間，先複製到驅動程式ioctl運作的程式空間中，然後再從ioctl運作空間複製到user mode library程式空間，最後user mode library才會再將packet複製到應用程式運作空間去。(圖4.16)以第五章CRT服務為例，寫出應用層的

等待與接收實作。

(接收端)



(圖 4.15)接收端應用層啟動讀取 queue buffer 的動作

```
static void tm1if_serve(int id)
{
      int i;
      while (!gServingStopped)
      {
            tmmanPacket packet;

            (void) tmmanSyncObjWait(gEvMsgSync);//acquire sem

            for (i = 0; !gServingStopped && (i < gNumDsps); i++)
            {
                  tmmanMessageReceive(gNodeData[i].hostcall_channel, &packet);
                        ⋮
}
```

等待 → 被 signal 後 → 接收

(圖 4.16)應用層的 CRT 服務所作的訊息等待與接收

4-4-2-2 應用層使用 user mode library 做溝通管道的建置

溝通管道這個抽象的概念，在實作上，其實就是由 4-4-1 每層提供的結構組成，每層結構就像是溝通管道的骨架，而這些結構裡儲存的內容，就是兩端進行溝通時，用來操控溝通管道的重要資訊。

因為用來做事件判斷的Vintr層是不管處理Event還是Channel事件都會使用到的，所以Vintr層在一開始初始化驅動程式時就可以先把該層需要使用到的結構準備好(4-4-1)，而且會利用4-2-1-2節提到的NameSpace代號分配機制，去分配Event與Channel事件代號給Vintr層，使用兩個VintrObject分別儲存Event與Channel的代號、兩者的ISR位址與ISR參數位址，還有兩者都須儲存的VintrManagerObject位址。如此，之後收發訊息就可以藉由代號到其代表的Vintr區塊去設定flags或是從flags的設定得知對應的事件是哪一個。

由於Vintr層與HAL層都是在做驅動程式的初始動作時就會創建好，所以應用程式做訊息傳遞或同步時做創建溝通管道的動作時，就不需要再創建Vintr層所需資源，於是在溝通前，還需要建置的溝通管道就只有Message層與Channel層(傳遞訊息)/Event層(事件同步)的資源，user moder libray便提供應用程式tmmanMessageCreate與tmmanEventCreate函式，讓應用程式可請求OS為傳遞訊息與事件同步建立所需的溝通管道。

訊息傳遞的溝通管道—Message層與Channel層的內容建置

如果應用程式要**傳收訊息**，應用程式就會使用tmmanMessageCreate函式，傳入4-4-2-1節創建好、用來當作阻斷介面的semaphore結構位址、此項service的名字等參數，內部呼叫ioctl系統呼叫後，就會在ioctl handler，使用在kernel中，負責建立訊息傳遞溝通管道的messageCreate函式，這個函式會先創建出一個代表溝通管道的MessageObject結構，重要資訊就是**service代號、控管queue buffer空間的資料結構位址(此結構包含讀取指標、寫入指標、queue buffer空間起始位址)、channelObject位址與阻斷介面(由應用程式傳入)**，之後應用程式要使用這個溝通管道，無非就是利用這個MessageObject的資訊，提供下層的人去運用，而Message層以下的資訊基本上應用程式就不會再碰觸到，驅動程式會自動運作。為了填寫好MessageObject結構內容，messageCreate函式於是會利用NameSpace機制為ARM端分配一個service代號給傳遞訊息的service，並把這個service代號填寫到共享記憶體的NameSpace區塊中，讓DSP端也可以得到這個service代號，然後messageCreate函式內部還會呼叫queueCreate函式，為這個service在Message層建立一個接收資料時要使用的64個packet結構大小的queue buffer，另外也會呼叫channelCreate函式，傳入Message層的ISR位址與ISR參數位址、channel的名字，在Channel層創建一個channelObject結構，儲存使用NameSpace機制申請到的Channel ID(目前與service代號的值是相同的)還有儲存Message層的ISR位址與ISR參數位址、ChannelManagerObject位址。

事件同步的溝通管道—Event層的內容建置

如果應用程式要做同步的動作，他會先利用tmmanSyncobjCreate函式先到kernel創建一個初始值為0的semaphore做為同步介面，並回傳這個semaphore的位址，然後使用tmmanEventCreate函式，傳入這個Event service的名字、還有之前創建好的semaphore位址，其內部呼叫ioctl系統呼叫後，就會在ioctl handler，使用kernel中負責建立同步事件溝通管道的eventCreate函式，這個函式會先創建出一個代表溝通管道的EventObject結構，重要資訊就是**service代號、同步阻斷介面(由應用程式傳入)**，之後應用程式要使用這個溝通管道，無非就是利用這個EventObject的資訊，提供下層的人去運用，而Event層以下的資訊基本上應用程式就不會再碰觸到，驅動程式會自己處理。為了填寫EventObject

內容，eventCreate函式於是會使用NameSpace機制為這個同步service分配一個service代號，然後將應用程式傳入的同步阻斷介面儲存於這個EventObject結構中。

4-4-2-3 應用層使用 user mode library 進行訊息交換或同步

使用 4-4-2-2 的 tmmanMessageCreate 或 tmmanEventCreate 創建好溝通管道後，應用程式接下來就要運作訊息交換與同步的動作了，我們在 4-3-2 的已經探討過 OS 層的溝通機制，下面我們會使用 user mode library 來要求 kernel 中的驅動程式來做訊息交換與同步的動作，並說明 user mode library 利用 ioctl 系統呼叫進入 kernel 中的 ioctl handler 後，還使用了 message 模組與 event 模組的哪些函式將 4-3-2 的動作進行包裝，完成溝通實作。

訊息交換

當應用程式的某個 service 想將存有資料位址或數字的 packet 傳給對方，可以使用 **tmmanMessageSend 函式**，傳入 packet 位址與利用 tmmanMessageCreate 函式得到的這個 service 的 MessageObject 位址，而後 tmmanMessageSend 會利用 ioctl 系統呼叫把這兩個資訊傳入 kernel 的驅動程式中，當作驅動程式 message 模組的 messageSend 函式的參數，而這個 messageSend 並沒有在 Message 層多做什麼就直接再把 MessageObject 儲存的 ChannelObject 位址與這個 packet 的位址傳入 channelSend 函式中，使用 channelSend 這個函式來完成 4-3-2-1 提到的傳送端在通道層以下的所有動作，如複製 packet 到共享記憶體、啟動中斷。

另外，如果應用程式想要從 Message 層的 queue buffer 讀取 OS 層接收到的 packet，每接收一個 packet 就使用 **tmmanMessageReceive 函式一次**，每次呼叫只要傳入要接收的 service 它的 MessageObject 結構位址及指向接收到的 packet 的指標，透過 ioctl 系統呼叫進入 kernel 後，就會利用驅動程式 message 模組的 messageReceive 函式從 MessageObject 結構中儲存的控管 queue buffer 空間的資料結構位址，依據結構裡頭儲存的讀取指標，先確認有新的 packet 填入讀取指標指到的 queue buffer 空間，沒問題的話才把目前讀取指標指到的 queue buffer 空間儲存的 packet 結構內容，從 Message 層中 queue buffer 空間複製到驅動程式運作 ioctl_handler 的程式空間，之後才從 ioctl_handler 運作空間複製到 tmmanMessageReceive 程式運作的 user space 空間，最後 tmmamMessageReceive 再把 packet 複製到應用程式傳下來的指標所指到的空間，如此，應用程式才得到 queue buffer 中的 packet。

兩端事件同步

當應用程式想要對對方某個 Event service 創建好的 Event 溝通管道裡的 semaphore 做 unblock 的動作時，就會使用 **tmmanEventSignal 函式**，傳入這個 Event service 的 EventObject 結構位址，透過 ioctl 系統函式進入 kernel，然後由驅動程式 event 模組的 eventSiganl 函式，完成 4-3-2-2 提到的 Eventsignal 端運作。

如果應用程式想要將自己的程式 block 住以等待對方來 signal 它，則可以使用 **tmmanSyncobjWait 函式**，傳入這個 Event service 的 semaphore 結構位址，利用 ioctl 系統函式進入kernel 後，使用 syncobjBlock 函式裡面的 down_interruptible 做直接的 block 動作。

4-5 第四章結論

利用上層雙向 client-server 的傳遞架構(ARM 或 DSP 都可當傳送端或接收端)，搭配底層共享記憶體與 interrupt 機制的運作，我們提供上層與下層訊息傳遞介面，利用應用層與 kernel 之間的 interprocess 運作來完成兩者間訊息的交換。為了在 multitasking 的環境中讓接收端在收到訊息後知道要做怎樣的服務，所以每個 service/事件都會有一個代號/flags 做為區別，為了記錄這些代號/flags，我們因此會在共享記憶體切割出幾個區塊，並使用階層式的架構來控制共享記憶體上的各個區塊以及設計 ISR 函式，讓 ARM 與 DSP 端可以處理兩端的程序同步或是收發 packet 的任務。

第四章最後我們介紹 user mode library 的使用，來達到訊息傳遞與事件同步的功能，然而除了 ARM 端有這樣的 user mode library，DSP 端也有同功能的 user mode library，第五章為了完成 CRT 服務，兩端都需要使用傳遞訊息函式庫的 tmmanMessageCreate，來建立 DSP 端與 ARM 端指令交換的溝通管道，然後 DSP 端會使用 tmmanMessageSend 來傳送指令而 ARM 端則以阻斷介面搭配 tmmanMessageReceive 函式來接收指令，最後對接收到的指令加以處理。

第五章 指令溝通--C Runtime Service

由於 DSP 端並沒有檔案系統，檔案系統在 ARM 端，如果 DSP 端想將影像處理完的資料送到 ARM 端檔案系統儲存，DSP 端必須想辦法使用 ARM 端的寫檔系統呼叫，將資料寫入 ARM 端硬碟裡，兩端應用層便使用 user mode library 設計一套 DSP 端傳送指令給 ARM 的程式，讓 DSP 端應用程式所下的系統呼叫，可以透過系統呼叫下的指令傳輸與 ARM 端的處理指令服務完成一個 DSP 檔案系統，只是這個檔案系統類似網路硬碟，DSP 端系統呼叫必須透過 RPC 指令來使用位於 ARM 端的檔案系統，這整個運作，我們稱為 C Runtime Service。

5-1 節我們會介紹 CRT 的服務內容並分別說明 ARM 與 DSP 端完成 CRT 服務，在應用層設計了哪些函式庫，還有這些函式庫提供了哪些功能，5-2 節便利用 5-1 節提到的函式庫，說明 DSP 傳送指令、ARM 接收指令並處理指令到最後 ARM 回傳資料給 DSP 端的過程，5-3 節便以 DSP 端使用 fwrite、tmMain_EXIT 的情況作為例子，搭配 5-2 節的指令傳輸機制，由 ARM 來完成寫檔與結束 CRT 服務的動作。

5-1 CRT 簡介

5-1-1 CRT 服務內容

由於 DSP 端並沒有檔案系統，如果 DSP 端應用程式需要從 ARM 端的檔案系統中讀取影像檔案來做解壓縮，或是將 DSP 端影像處理、壓縮完的資料送到 ARM 端檔案系統儲存，因為檔案系統在 ARM 端，所以 DSP 端必須想辦法使用 ARM 端儲存檔案時所用的檔案系統 I/O 驅動程式，將資料存到 ARM 端硬碟裡，C Runtime Service (CRT)就是提供這樣的服務給 DSP 端，讓 DSP 端應用程式下達的指令(C I/O call)可以被傳送到 ARM 端，由 ARM 來協助完成 DSP 端 TCS C library 的系統呼叫，主要有標準輸出入(monitor、 keyboard)、FILE I/O、讀取 DSP 端的應用程式所需參數等功能。另外，有些關於檔案系統會用到的指令如 mkdir、rmdir、link、rename、sync、有關 socket 的運作等可能只有在特定 OS 環境下才會使用的指令，DSP 端也可以將這些指令傳送給 ARM，由 ARM 直接調用 Linux 本身環境裡的函式來運作。不過本章我們會著重討論標準輸出入(monitor、 keyboard)、跟檔案系統有關的操作指令使用(表 1 中間兩排)。

(表 5.1)DSP 端可使用的 C I/O call

accept()	fclose()	__argc	putenv()
access()	fcntl()	__argv	system()
bind()	fgetc()	gets()	time()
closesocket()	fgets()	getc()	
connect()	fopen()	printf()	_psos_exit()
getenv()	fprintf()	putc()	exit()
gethostbyaddr()	fputc()	puts()	tmMain_EXIT()
gethostbyaddr_r()	fputs()	read()	
gethostbyname()	fread()	scanf()	
gethostbyname_r()	fscanf()	close()	
gethostname()	fseek()	open()	
getprotobyname()	fstat()	write()	
getsockname()	fsync()	lseek()	
getsockopt()	fwrite()	isatty()	
recv()	stat()	sync()	
recvfrom()	mkdir()	tmpnam()	
send()	opendir()	unlink()	
sendto()	rmdir()	link()	
setsockopt()	rewinddir()	closedir()	
socket()	rename()	readdir()	
inet_addr()	mktemp()		
ioctlsocket()			
select()			
listen()			

5-1-2 CRT 服務架構

CRT 的服務架構是一個建構於兩端 OS 層之上，雙向但不對稱式的 client-server 架構。提供 CRT 服務的程式主要在 ARM 端執行，在啟動 PNX1005 前，他會創建多個接收與處理指令的 server 執行緒，共同使用一個 queue buffer 來接收 packet，透過這個 queue buffer，DSP 端可以同時傳入多個系統呼叫指令給 ARM 端，ARM 端可以使用 queue buffer 當作緩衝區，然後再由 OS 選擇 ARM 端 server 執行緒從第一個插入的 packet 開始作讀取與處理，但是因為 PNX1005 還未啟動，所以每個 server 執行緒會使用公用的 semaphore 將每個執行緒阻斷在作讀取 queue buffer 的動作前。當 ARM 端應用程式透過驅動程式將 DSP 端可執行檔下載到 DSP 端 RAM 並啟動 PNX1005 後，ARM 端等待的 server 執行緒在接收到 DSP 端傳送過來、攜帶有指令位址的 packet 後，ARM 端 OS 就會選擇一個等待中的 server 執行緒，

啟動它來處理指令。

而在 pSOS 下運作的 DSP 端，在傳送系統呼叫請求時就是 client，他會將包裝完的指令內容的位址，透過 user mode library 將存有指令位址的 packet 傳送給 ARM 端。為了讓 DSP 端的系統呼叫可以是一個 aynchoronous I/O call，這樣 DSP 端在等待 ARM 端處理指令的同時，DSP 端還可以另外處理其他的系統呼叫，因此 DSP 端在送出 packet 後，目前系統呼叫的 thread 就會被 suspend 起來，而這個被 suspend 的系統呼叫執行緒就變成等待 ARM 端回覆的 server，之後收到 ARM 端訊息時，這個被 suspend 的 server 執行緒就會被喚醒，負責從得到的回覆中做回傳資料給 DSP 端應用程式的動作。

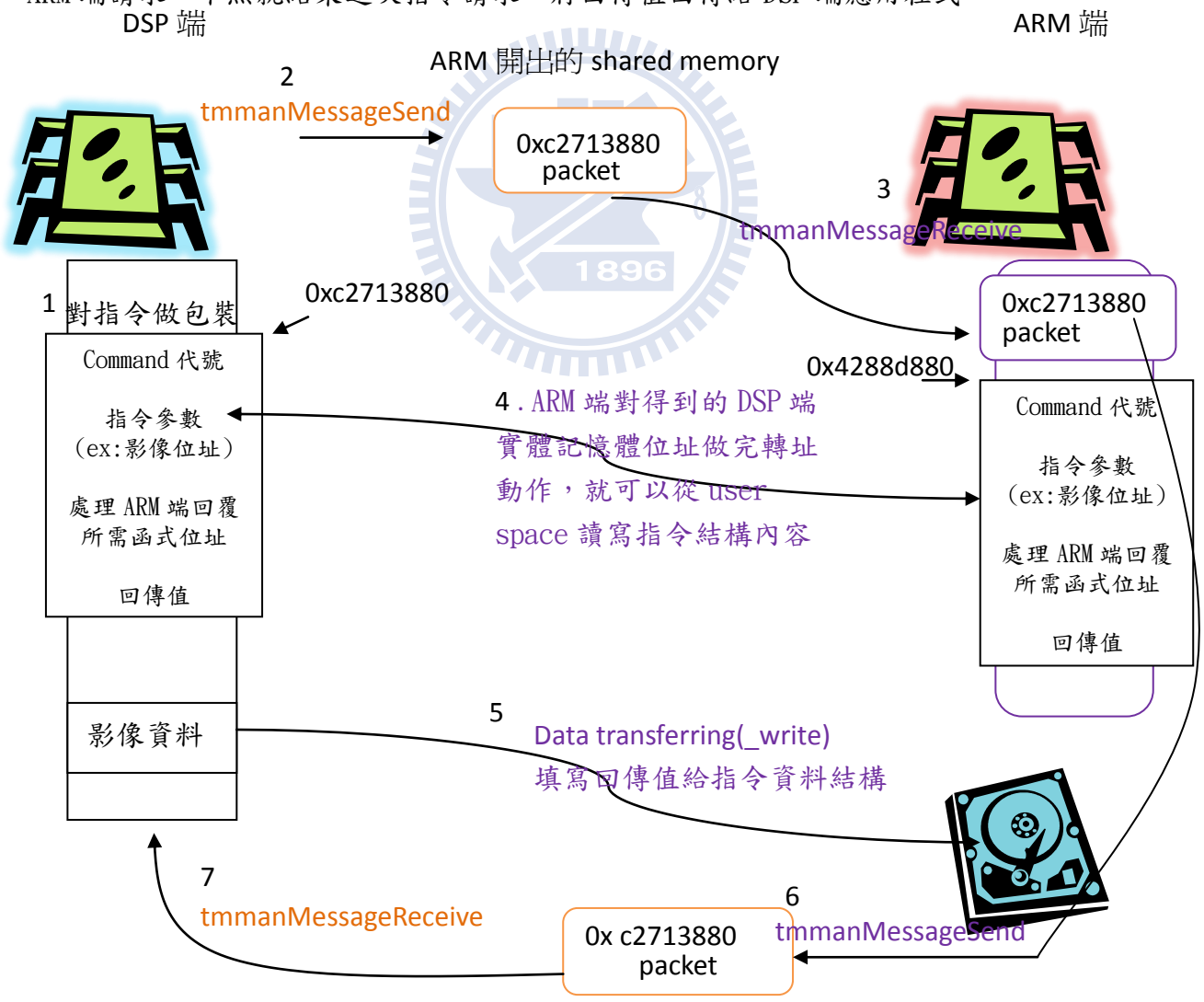
但是因為 DSP 端被喚醒的系統呼叫執行緒，必須在 DSP 端傳送指令前就儲存在指令內容中，這樣 ARM 端回覆訊息給 DSP 端時才知道要喚醒哪一個被 suspend 的執行緒，所以 DSP 端並不像 ARM 端，可以讓 DSP 端不斷傳送訊息過來，然後由 OS 自動選擇一個 server 執行緒從 queue buffer 中讀取 packet 做指令處理，ARM 端必須很直接地指定要喚醒 DSP 端哪一個特定的 server 執行緒來做後續的回傳動作，因此 DSP 端是一個比較被動的 server，因此兩端的 server 特性是不對稱的。

因為要傳送指令資訊，所以兩端要運作 CRT 服務所使用的溝通模式是第四章所提到的訊息傳遞模式，而且因為 DSP 端一開始傳給 ARM 端的是指令”位址”並不直接是指令的內容，這個位址是 DSP 端實體記憶體的位址，為了讓 ARM 端應用程式可以在應用層的 CRT 服務模組中控制 DSP 端實體記憶體空間，所以在 ARM 端啟動 CRT 服務前，會先使用 tmmanMapDeviceMemory 函式，在目前執行的程序中向系統請求一塊 user space 空間，然後把 DSP 端 RAM aperutres 空間映射到這塊 user space 空間來，之後 ARM 端便在應用層的 CRT 服務 thread 中，直接可以使用對虛擬記憶體位址的操作對對應到的 DSP 端實體記憶體空間進行操作。

之後 CRT server thread 被啟動，ARM 端 CRT server thread 於是先以 Message 層的阻斷介面 block 住，DSP 端應用層則將指令轉成用 command 代號來代表，並將 command 代號、指令參數儲存於特定的資料結構中，這個資料結構就變成指令的化身，用來儲存傳遞給 ARM 端的指令訊息。將指令轉成特定的資料結構後(下圖 1)，DSP 應用層於是使用 tmmanMessageSend 透過 OS 層溝通機制將存有指令資料結構位址的 packet 傳送給 ARM 端(下圖 2)，然後 DSP 端就先將目前處理指令的 thread 給 suspend 起來；ARM 端在收到中斷、完成 OS 層溝通機制後，kernel 就會 signal Message 層的阻斷介面，啟動應用層中再等待的 CRT server，server 於是使用 tmmanMessageReceive 從 Message 層接收 packet(下圖 3)，然後從 packet 取出這個指令資料結構的位址，此時要特別注意，ARM 端應用程式是在虛擬記憶體上操作，DSP 端應用程式則是直接使用 DSP 端實體記憶體，所以 ARM 端要對 DSP

端的資料做處理時就需要注意記憶體位址間的轉換(5-2-3)，於是 server 必須先將得到的 DSP 端指令資料結構位址轉換成 ARM 端 user space 的虛擬記憶體位址，ARM 端於是可以在 server 的 user space 透過這個虛體記憶體位址直接讀寫 DSP 端的指令資料結構內容(下圖 4)，然後 server 會依照 command 代號選用 ARM 端系統中提供的指令來運作，假如是 DSP 端下的是寫檔指令，ARM 端此時會利用資料結構內容解讀出 DSP 端想做的指令，然後利用 ARM 端自己的環境中的寫檔指令將 DSP 端的資料寫入檔案系統中，並將指令運作完後的回傳值填入 DSP 端的指令資料結構中(下圖 5)。

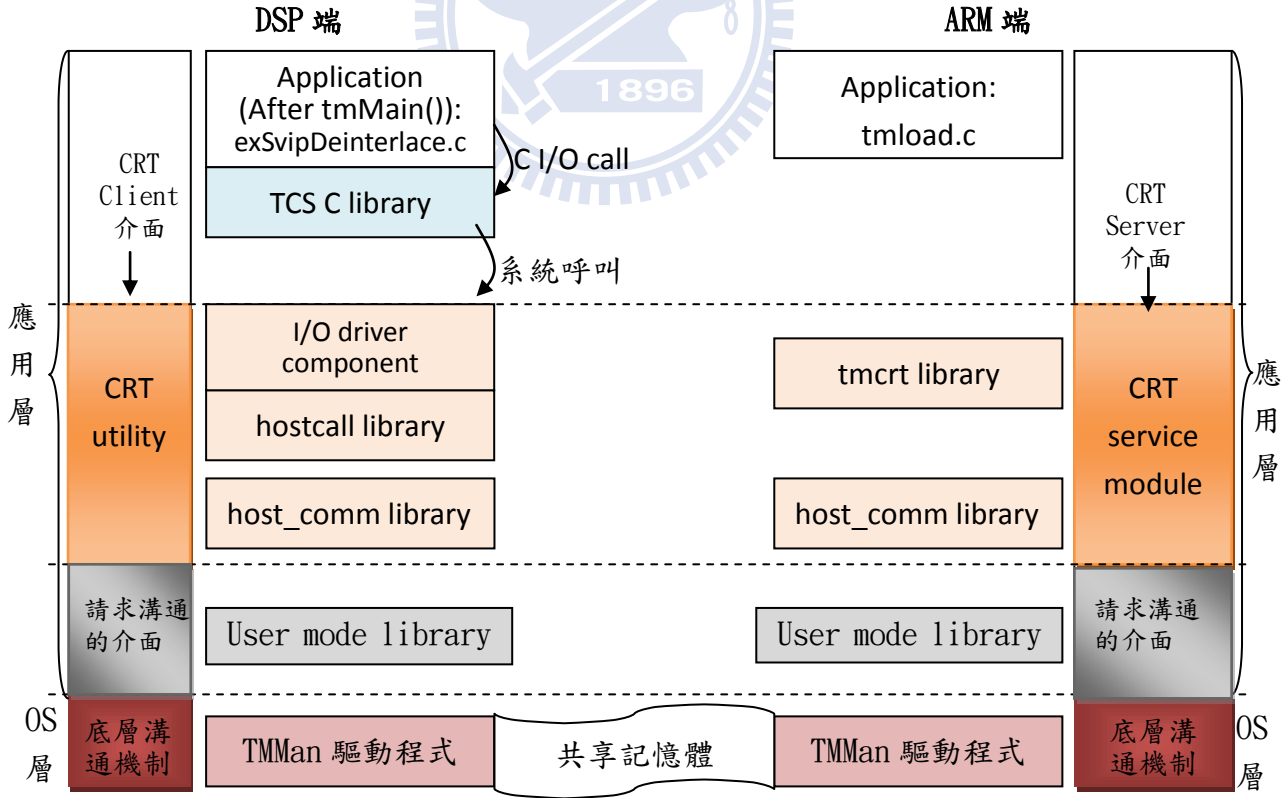
ARM 端處理完後再次利用 tmmanMessageSend 直接把之前讀取到的 DSP 端指令資料結構位址傳回給 DSP 端(下圖 6)，DSP 端 OS 層收到中斷後，OS 層就會從收到的 packet 中使用傳送前儲存於指令資料結構中用來處理 ARM 端回覆時所需函式位址，來喚醒 DSP 端被 suspend 的 thread，被喚醒的 thread 才會繼續使用結束此次指令請求的函式從 ARM 端更新過後的指令結構內容中，利用回傳值或是處理進度狀態來判斷是否需要再進行下一次 ARM 端請求，不然就結束這次指令請求，將回傳值回傳給 DSP 端應用程式。



（圖 5.1）指令傳輸過程

在此，我們可能會想，為什麼(上圖 4)的步驟中，如果已經判斷為是做寫檔的動作，代表 ARM 端端已經可以從指令內容讀取到寫檔指令的參數--影像位址了，卻不直接使用這個位址就在記憶體上直接操作影像呢？因為既然是要寫檔，最後的目的地就是硬碟，可是如果不直接利用 Linux 提供的寫檔系統呼叫，ARM 端應用程式必須先利用影像位址將整個影像完整的複製到 user space，然後才再對複製到 user space 的影像作寫檔的動作，這樣就多了一個步驟，會比較沒有效率，所以如果直接把影像位址給 Linux 的寫檔系統呼叫，影像資料就可以直接從 DSP 端實體記憶體搬移到 ARM 端硬碟中！

由於 DSP 端在將指令送出前還需要對指令內容做包裝，ARM 端將指令接收後也需要從獲得的指令內容作對應的處理，所以我們在 ARM 端應用層設置了 **CRT service module**，提供 ARM 端一套用來作為指令處理的應用，應用程式可以使用它來對 DSP 端傳送過來的指令位址進行處理並使用 user mode library 進行指令內容交換，而 DSP 端的 CRT 請求則設置了 **CRT utility**，提供一套機制為 DSP 端做指令內容包裝並利用 user mode library 進行指令內容交換，等於是搭配 ARM 端 CRT service module，為應用程式與 TCS C library 提供指令系統服務。(圖 5.2)為整個環境下 CRT 服務在兩端的組成架構以及使用 user mode library 完成兩端指令傳輸的概念， 5-1-2-1 與 5-1-2-2 節將會詳細說明此圖每個單元的用途。



（圖 5.2）CRT 整體服務架構

5-1-2-1 ARM 端的 CRT service module

ARM 端 CRT service module 由 tmcrt library、host_comm library 組成。tmcrt library 主要是提供應用程式與 CRT service module 的 CRT server 介面，而 host_comm library 就是要來實作這個介面的功能，另外 tmcrt library 也提供了 CRT service module 與檔案系統有關操作函數介面以及「啟動」結束 CRT 服務的動作介面。

tmcrt library

tmcrt library 有兩部分，一部分為 ARM 端服務 DSP 請求時，ARM 端 CRT service module 與操作檔案系統的一些函數介面(表 2)，其實 ARM 端收到請求後，依照傳過來的指令內容就可以判斷是要去執行哪一個 ARM 端 kernel 提供的檔案系統操作函式來完成指令動作，不過使用者可以自己設計額外的函數介面，對欲操作的檔案型態做檢查(檢查是否為標準輸出入)，或是列印使用者想知道的操作訊息，然後再呼叫 linux kernel 對檔案系統的操作函式，像是檔案輸出入的 FILE I/O 指令(_open、_read、_write 等)。而(表 2)這些額外設計的函數，後來會由 tmcrt library 提供給應用程式使用的 cruntimeInit 函式，將這些操作位址儲存到 host_comm library 去，這樣 host_comm library 利用訊息傳遞函式庫接收到指令位址，解析指令內容後，便可使用 C I/O call 對應到的操作函數對檔案系統進行操作。

tmcrt library 除了對檔案系統操作提供了函數介面外，另外還提供了一個用來結束 CRT 服務的函數介面，因為 CRT 服務開始執行後就不能隨意結束，正常情況下 DSP 端要結束 DSP 端的應用程式時，才會請求 ARM 端做 CRT 服務的結束動作，因此，ARM 端就需要事先準備好用來做這個動作的函式，同樣的，這個函式也會儲存到 host_comm library 中，讓 host_comm library 來使用。

(表 5.2)ARM 端 tmcrt library 提供的 FILE I/O 函數介面及啟動結束 CRT 服務的函數介面

```
(RPCServ_OpenFunc) OpenFunc,
(RPCServ_OpenDllFunc) OpenDLLFunc,
(RPCServ_CloseFunc) CloseFunc,
(RPCServ_ReadFunc) ReadFunc,
(RPCServ_WriteFunc) WriteFunc,
(RPCServ_SeekFunc) LseekFunc,
(RPCServ_IsattyFunc) IsattyFunc,
(RPCServ_FstatFunc) FstatFunc,
(RPCServ_FcntlFunc) FcntlFunc,
(RPCServ_StatFunc) StatFunc,

(RPCServ_ExitFunc) ExitCodeFunc,//非 FILE I/O，用來停止 ARM 端 CRT 服務
```

這些 tmcrt library 用來支援 DSP 端的 FILE I/O 函數是以適用於 Linux/UNIX 系統的 POSIX –style 的 I/O 標準為基礎去設計的(open、read、wrtie、close 等)，它們會以整

數型態的 file descriptor 來描述一個檔案，而且是對 unformatted 資料進行操作，而不是使用 C FILE STREAM I/O(fopen、fwrite、fclose 等)，所以我之後把非 C FILE STREAM I/O 的系統呼叫都稱為 POSIX C I/O call。雖然 C FILE STREAM I/O 也是基於 POSIX 實作的，可是他要先將 STREAM 層的資料先 data-copy 到一個 local buffer 再繼續做 POSIX 層的 I/O 動作，這樣就多花了一些時間與資源了。

tmcrt library 的另一部分則是應用程式與 CRT service module 的介面(上圖的 CRT Server 介面)，由 cruntimeInit、cruntimeCreate、cruntimeDestroy 及 cruntimeExit 函式組成，讓應用程式可以傳入使用者對 CRT 服務要求的環境設定，然後直接使用這些函式來完成有關運行 CRT 服務的特定功能，然而這些函式內部則都會由 host_comm library 來實作功能，因此下面我們會將 CRT Server 介面的函數一一提出來，看看 tmcrt library 會提供哪些資訊給 host_comm library，然後 host_comm library 實作出的介面功能。

host_comm library

host_comm library 主要由 TM1IF 與 RPCServ 模組組成，由 **TM1IF 模組提供 tmcrt library 功能函式介面**，完成 CRT thread 的建置與終止與 CRT 指令溝通管道的建立，因為 TM1IF 模組建置的 thread 它會使用 tmmanMessageSend 藉由溝通管道將指令傳送到 ARM 端，所以這個模組才有 Trimedia interface 之稱。

其中 TM1IF_start_serving 函式會去創建 CRT 服務的 thread、TM1IF_init 函式去創建 Message 層與應用層的阻斷介面函有承接 tmcrt library 傳入的操作函數介面，利用 RPCServ_init 儲存進 RPCServ 模組中，然後 TM1IF_add_node_info 函式去創建溝通管道、TM1IF_term 函式去終止 CRT 服務的 thread 的動作、TM1IF_remove_node_info 函式去釋放溝通管道，為了在所創 CRT 服務的 thread 中接收 DSP 端傳過來的指令請求，所以 TM1IF 模組開出的 CRT 服務 thread 會使用 user mode library 來接收 DSP 端指令，然後請求 RPCServ 模組作指令處理，處理完後 CRT 服務 thread 又會再使用 user mode library 對 DSP 端送出回覆的動作。

而 **RPCServ 模組**則負責指令處理、儲存指令處理運作上需要使用到的環境資訊及釋放溝通管道， RPCServ 模組會提供處理指令的功能給 TM1IF 模組開出的 CRT 服務 thread，然而因為處理指令時會需要 tmcrt library 提供的操作函式介面以及處理指令時所需環境資訊，所以 RPCServ 模組會提供 RPCServ_init 函式給 TM1IF 模組，然後 TM1IF 模組再提供 TM1IF_init 函式給 tmcrt library，讓 tmcrt library 裡的操作函式介面以及 TM1IF 模組中用來釋放溝通管道的 termination 函式位址，可以透過兩層 init 函式將函式位址儲存到 RPCServ 模組的運作環境中。透過 host_comm library 裡這兩個模組，我們於是可以

往上層再實作出 tmcrt library 提供給應用程式使用的 cruntimeInit、cruntimeCreate、cruntimeDestroy 及 cruntimeExit 函式。

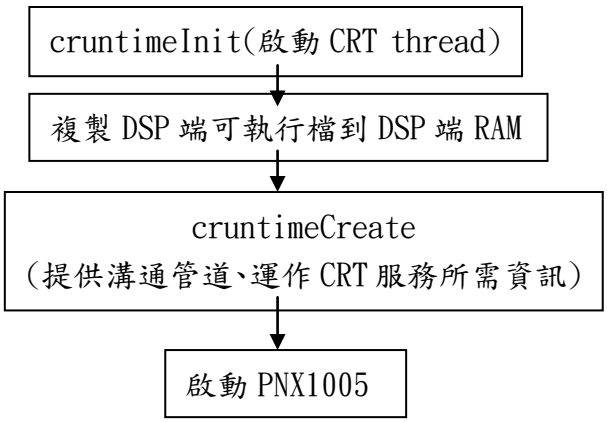
1. cruntimeInit：

cruntimeInit 函式會使用 TM1IF_init 完成將 tmcrt library 裡額外設計的 FILE I/O 函數介面及啟動結束 CRT 服務的函數介面給 RPCServ 模組、創建應用層與 OS 層的阻斷介面，然後用 TM1IF_start_serving 函式啟動 CRT 服務。

TM1IF_init 函式有兩件事要做，一個是將 tmcrt library 裡設計好的 FILE I/O 函數介面及啟動結束 CRT 服務的函數介面(表 5.2)，直接利用 RPCServ_init 將提供的介面函式存入RPCServ模組中。另一個則是使用第三章的 tmmanSyncobjCreate 函式，在 ARM 端 kernel 裡創建一個 semaphore 結構，並將這個 semaphore 位址回傳給 TM1IF 模組，儲存於 TM1IF 模組宣告的全域變數，之後這個 semaphore 位址還會因為應用程式使用 cruntimeCreate 創建的 CRT 訊息溝通管道，而儲存到 OS 層的 Message 層去(請看 cruntimeCreate 的說明)作為應用層與 OS 層的阻斷介面。

而 TM1IF 模組的 TM1IF_start_serving 函式則用來創建一個以上提供 CRT 服務的 thread (5-2-1 ARM 端)，當 CRT 服務的 thread 被創建後即被啟動，應用層便會一直存在這幾個 thread 來提供 CRT 的服務，除非 DSP 端要求停止 CRT 服務或是 ARM 端應用程式被強迫停止。

要注意的是，因為 cruntimeInit 函式會去啟動 CRT 服務的 thread，所以在 ARM 端應用程式啟動 PNX1005 的運作前，應用程式就必須執行 cruntimeInit 函式，如此一來，DSP 端開始運作後就可以利用 CRT 提供的服務了。而且在 DSP 端可執行檔複製到 DSP 端 RAM 以後，ARM 端應用程式要求將 PNX1005 啟動前，ARM 端還會繼續使用下面的 cruntimeCreate 儲存 CRT 運作所需環境資訊(圖 5.3)。



(圖 5.3)ARM 端初始概況

2. cruntimeCreate：

(i) **cruntimeCreate 函式用來儲存 CRT 運作所需環境資訊到負責處理指令的 RPCServ 模組中**。運作所需環境資訊包含 DSP 端 RAM 映射到 ARM 端 user space 的起始位址、physical 起始位址與 RAM 的空間大小以及 user space 起始位址減掉 physical 起始位址後得到的位址差，還有應用程式從 console 讀入的參數並經由應用程式解譯後得到的參數個數(Argc 值)與參數內容起始位址(Argv 位址)、設定標準輸出入(stdin 0/stdout 1/stderr 2)，另外也配置了 CRT 指令溝通管道並儲存其 MessageObject 位址給 RPCServ 模組，並為了在必要時能夠釋放 TM1IF 模組在 cruntimeInit 時創建的 CRT 指令溝通管道，RPCServ 模組還需要儲存 TM1IF 模組提供用來釋放溝通管道的 termination 位址。

上面所提到的環境資訊，**資訊來源有三種，一種是應用程式給的，一種是 TM1IF 模組創的，一種則是 cruntimeCreate 函式直接使用 user mode library 透過 TMMan 驅動程式讀取來的**，於是 cruntimeCreate 函式分成三步來做，第一，直接使用 user mode library 的 tmmanDSPGetInfo，來讀取 DSP 端 RAM 映射到 ARM 端 user space 的起始位址、physical 起始位址與 RAM 的空間大小，然後將 user space 起始位址減掉 physical 起始位址後得到位址差，之後接收到 DSP 端指令位址時可以作為位址轉換用(5-2-3 節）；第二，藉由 cruntimeCreate 函式，應用程式可以傳入 CRT 服務運作時所需環境的設定((表 5.3)上半部),包括應用程式從 console 讀入的參數並經由應用程式解譯後得到的參數個數(Argc 值)與參數內容起始位址(Argv 位址)、設定標準輸出入(stdin 0/stdout 1/stderr 2)，這些環境資訊會由 cruntimeCreate 函式透過 TM1IF_add_node_info 函式進一步使用 RPCServ_add_node_info 函式存入 RPCServ 模組；第三，TM1IF_add_node_info 函式裡也會使用 tmmanMessageCreate 創建 CRT 訊息傳遞的溝通管道，而且 TM1IF 模組也設計了一個用來釋放溝通管道的 termination 函式，所以 TM1IF_add_node_info 函式也會同時透過 RPCServ_add_node_info 函式把 MessageObject 位址與 termination 位址存入 RPCServ 模組中((表 5.3)下半部)。

(表 5.3)cruntimeCreate 函式提供給 host_comm 函式庫運作 CRT 服務所需的環境資訊

```
typedef struct {
    UInt32    node_number;
    UInt32    argc;
    String   *argv;
    UInt32    Stdin;
    UInt32    Stdout;
    UInt32    Stderr;
    UInt32    address_shift;
    UInt32    sdram_base;
    UInt32    sdram_limit;

    RPCServ_Node_Terminator  terminate;
```

處理指令所需記憶體空間資訊與標準輸出入資訊

```
    Pointer    data;//指到 NodeData 結構
    stackTraceData_t  stackInfo;
} RPCServ_NodeInfo;

typedef struct
{
    UInt32  dsp_number;
    UInt32  dsp_handle;
    UInt32  hostcall_channel;/溝通管道位址(MessageObject 位址)
    Bool    valid;
} NodeData;
```

(ii) 在 tmcrt library 中儲存用來啟動應用程式進行終止 CRT 服務動作的 mutex 位址。CRT 服務的 thread 在應用程式呼叫 cruntimeInit 函式被啟用後，此時環境中除了主程式之外，還有開出的 CRT 服務 thread，thread 本身有 semaphore 阻斷介面做為接收動作的觸發， 此時主程式仍會繼續執行，主程式的最後會做終止 CRT 服務的動作，為了讓 DSP 端應用程式要結束前對 ARM 端請求結束 CRT 服務或是 ARM 端應用程式使用 crtl-c 強迫終止應用程式時，才能停止 CRT 服務的 thread，所以在主程式中我們使用 mutex 結構來做為終止 CRT 服務的動作的阻斷介面。

為了讓應用程式只有在上述兩種情形可以做終止 CRT 服務 thread 的動作，應用程式會先在 user space 宣告一個全域的 mutex 變數，而這個 mutex 在應用程式啟動 PNX1005 後就會被鎖定，之後我們讓應用程式不斷嘗試再去鎖定這個 mutex 的動作，但是因為應用程式一直鎖不到 mutex，所以應用程式就會一直被擋在做終止 CRT 服務 thread 的動作前，只有由上述兩種終止情況去把這個 mutex 解除鎖定時，應用程式才會成功鎖定到 mutex，執行下面的終止動作。

如果啟動終止的事件是由 DSP 端請求產生，那麼這個 mutex 位址必須先由應用程式透過 cruntimeCreate 函式，將這個 mutex 位址儲存到 tmcrt library 裡另一個全域變數去，這樣 RPCServ 模組處理完 DSP 端傳送過來要求終止 CRT 的指令，在使用 tmcrt library 提供給 host_comm library 的啟動結束 CRT 服務的函式介面--ExitCodeFunc 函式時，這個函式就可以去解除應用程式之前對這個 mutex 的鎖定。詳細終止動作請看 5-2-5 節。

3.cruntimeDestroy—釋放訊息溝通管道：

cruntimeDestroy 用於經由 ctrl-c 強迫終止 ARM 端應用程式時做釋放訊息溝通管道的動作，這個函式同樣會在應用程式不斷嘗試再去鎖定這個 mutex 的動作之後才被執行。

為了讓 ctrl-c 事件發生時，應用程式知道要執行什麼處理動作，所以應用程式會事先設計一個處理 ctrl-c 事件的 handler，然後利用 signal(SIGINT, ctrlc_signal_handler)將這個 handler 註冊給這個 ctrl-c 中斷事件。之後當使用者下達 ctrl-c 指令，這個 handler 就會與上一段提到的 ExitCodeFunc 函式類似，先將被應用程

式鎖定的 mutex 解除鎖定後，應用程式就會呼叫 cruntimeDestroy 函式，這個函式會利用儲存於 host_comm library 的 termination 函式，由它呼叫 tmmanMessageDestroy 函式來釋放訊息溝通管道。

4.cruntimeExit—結束 CRT 服務的 thread：

由 2.(ii)我們已經知道結束 CRT 服務 thread 的動作(cruntimeExit)何時會被執行，而應用程式啟用 cruntimeExit 後所做的事情，就是由 host_comm library 的 TM1IF 模組的 TM1IF_term 函式，使用 pthread_join 函式與 tmmanSyncobjSignal 函式去 signal 在等待接收的 CRT thread，等待目前還在執行的 CRT thread 執行完這回合後，才對 thread 做結束與資源釋放的動作。

因為 cruntimeInit 函式藉由 host_comm library 的 TM1IF_start_serving 函式創建的 CRT thread 在運作時，會在一個無窮迴圈裡，讓 thread 不停透過阻斷介面等待底層完成傳輸，然後啟動 thread 來利用 tmmanMessageReceive 函式從 Message 層的 queue buffer 做接收 packet 的動作。所以如果我們想要等 CRT thread 把目前 while loop 裡這一回合的動作做完再終止 CRT 服務，除了在 host_comm library 中宣告一個全域變數 gServingStopped，讓 thread 每次進入 while loop 前都要藉由這個變數判斷是否繼續進行 while loop 的下一回合，這樣應用程式使用 cruntimeExit 函式時，就可以透過 host_comm library 將這個變數設定為 True，讓 thread 無法進行下一回合，另外，cruntimeExit 函式還必須 signal OS 層與應用層的阻斷介面，確保每個 CRT 服務 thread 這回合運作都可以正常結束，不會一直被 block 住，這樣每個 CRT 服務的 thread 才可以藉由 pthread_join 函式成功結束並釋放掉 thread 資源。詳細終止動作請看 5-2-5 節。

5-1-2-2 DSP 端的 CRT utility

CRT utility 其實就是用來實作 DSP 端應用程式呼叫系統 C 函式(C I/O call)後(表 5.1)，每個函式的內部實際運作，於是應用程式使用的 C I/O call 與 CRT utility 的函式實作介面我們稱為 CRT client 介面，對於 C I/O call 中執行與檔案系統操作有關的各種指令與啟動結束 CRT 服務的指令介面，是由 I/O driver componenets 裡許多 I/O 函數與啟動結束 CRT 服務的函數組成。而 C I/O call 的實作則又靠 hostcall library 及 host_comm library 來完成。

由(5-1-2-1 節)ARM 端的 tmcrt library，我們知道 ARM 端會使用系統中的 POSIX C I/O call 來做 FILE I/O 的操作，所以 DSP 端不管是在操作 FILE I/O 時是使用 POSIX C I/O call 或是 C STREAM I/O call，在使用 CRT utility 前會由 **TCS C library** 將同類功能的 C STREAM

I/O call 轉換成 POSIX I/O call，像是 read(), fread(), getc(), fgetc(), gets(), fgets(), scanf(), fscanf()到了 ARM 端，就全部會統一由 read 來運作，於是像 fwrite 在傳送大於 8K byte 的資料量時，會經由 TCS C library 轉換成每次以 8K byte 為單位，請求 CRT utility 向 ARM 送出請求一次，所以可能會向 ARM 發出多次請求，但是如果是 write 的話，則只需要向 ARM 提出一次請求就可以做完寫入動作。因為 TCS C library 有做這樣的轉換，所以 DSP 端操作 FILE I/O 指令數量與 ARM 端提供服務的對應操作函式數量就是多對一的關係。

I/O driver components—為 CRT client 介面，提供各種操作檔案系統的函數，用來銜接 TCS C library，這些函數會對指令進行特定包裝，並啟動 hostcall library 開始一連串處理動作，而這些函數必須等到 hostcall library 執行結束，才表示 CRT 的動作處理完也取得回傳值，然後再把這個回傳值回傳給 TCS C library ：

因為 DSP 端的 pSos，本身並沒有提供「請求 ARM 端完成 C I/O call」的函式實體，所以我們需要設計一組操作函數(表 5.4)來實作每個 C I/O call 的功能，這些實作函數稱為 I/O driver components，這個 components 裡頭由三類函數組成，用來實作 FILE I/O(含標準輸出入)的函式、啟動結束 CRT 服務的函式以及針對 ARM 端 Linux 環境可以使用的其他特殊檔案系統函式，而這些函數的 FILE I/O(含標準輸出入)部分與結束 CRT 服務的函式其實就與(表 5.2)ARM 端 tmcrt library 提供的 FILE I/O 函數介面互相呼應，其他特殊函式則會在將指令傳到 ARM 端後，由 ARM 端 kernel 直接處理，所以 ARM 端就沒有特別設計函式來實作它們。

(表 5.4) DSP 端 I/O driver components (粗體部分屬於 FILE I/O 與啟動終止 CRT 服務的操作函式)

```
static Int32 OpenFunc  (String  path,Int32  oflag, Int32  mode )
static Int32 OpenDllFunc( String path )
static Int32 CloseFunc (  Int32 file )
static Int32 ReadFunc  (Int32   file,Pointer buf,Int32   nbyte )
static Int32 WriteFunc (Int32   file,Pointer buf,Int32   nbyte )
static Int32 SeekFunc  (Int32   file,Int32  offset,Int32   whence)
static Int32 IsattyFunc( Int32 file )
static Int32 FstatFunc (Int32   file,struct stat  *buf)
static Int32 StatFunc ( String path, struct stat  *buf )
static Int32 FcntlFunc (Int32 file, Int32 cmd, Int32 flags )

void _report_exit_status ( Int status )

static Int32 SyncFunc  ( )
static Int32 FSyncFunc (Int32 file )
```

```
static Int32 UnlinkFunc( String path )
static Int32 MoveFunc  ( String src,  String dest )
static Int32 LinkFunc  ( String src,  String dest )
static Int32 MkdirFunc ( String path, Int32 mode )
static Int32 RmdirFunc ( String path )
static Int32 AccessFunc( String path, Int32 amode )
static DIR*  OpendirFunc  ( ConstString path )
static Int32 ClosedirFunc (DIR*   dir  )
static void RewinddirFunc( DIR*   dir  )
static struct dirent*  ReaddirFunc  ( DIR*   dir  )
                ⋮
```

由 5-1-2-2 一開始介紹 TCS C library 時，就有提到 DSP 端應用程式可以使用的 FILE I/O 指令與 ARM 端真正做 FILE I/O 的操作函式有著多對一的關係，為了讓 DSP 端多個 I/O 指令只對應到 ARM 端的一個 I/O 指令，這些指令便需要由 TCS C library 將操作 FILE I/O 的 C I/O call 轉成 POSIX I/O call，而且不管是什麼指令，都需要經過特殊包裝才能送給 ARM 端。為了辨識不同的指令功能，於是，類似功能的 DSP 端指令，就以一個 I/O 指令的代號表示(表 5.5)，不同功能的 DSP 端指令也會有自己的代號，我們把這些代號稱為 C Runtime Call ID。

(表 5.5)DSP 端 FILE I/O 的 C I/O Call 、結束 CRT 服務的 Exit 函式與 C Runtime Call 代號的對應

FILE I/O 的 C I/O Call Functions	C Runtime Call ID
close()、fclose()	HostCall_CLOSE
fgetc() 、fgets()、fscanf() getc()、gets()、read()、scanf()	HostCall_READ
fopen()、open()	HostCall_OPEN
fprintf()、fputc()、fputs() printf()、putc()、puts()、write()	HostCall_WRITE
isatty()	HostCall_ISATTY
stat()	HostCall_STAT
fcntl()	HostCall_FCNTL
fseek()、lseek()	HostCall_LSEEK
fstat()	HostCall_FSTAT
_psos_exit()、exit()、tmMain_EXIT()	HostCall_EXIT

(其它指令 ID 請看 Appendix B)

在 pSos 中 DSP 端可執行檔開始執行時，會將 I/O driver components 中各個操作函數位址註冊到特定的結構--UID_Driver_t 中(表 5.6)，這個結構的成員都是函數指標，儲存 I/O driver components 中各個操作函數的位址。於是，指令經由 TCS C library 的轉換後使用的系統呼叫，藉由(表 5.7)系統呼叫介面進入處理系統呼叫的模組後，有關檔案系統呼叫部分，模組中用來處理該檔案系統呼叫的函式就會到 UID_Driver_t 結構中，取用事先有被註冊的檔案系統操作函數，然後由檔案系統操作函數對系統呼叫指令進行傳送給 ARM 前所需的指令包裝，然後再把傳送包裝過的指令內容傳送給 ARM。

(表 5.6) UID_Driver_t 結構

```
struct UID_Driver_t {
     UID_Driver          next;
     IOD_RecogFunc       recog;
     IOD_InitFunc        init;
     IOD_TermFunc        term;
     IOD_OpenFunc        open;
     IOD_CloseFunc       close;
     IOD_ReadFunc        read;
     IOD_WriteFunc       write;
     IOD_SeekFunc        seek;
     IOD_IsattyFunc      isatty;
     IOD_FstatFunc       fstat;
     IOD_FcntlFunc       fcntl;
     IOD_OpenDllFunc     open_dll;
     IOD_StatFunc        stat;
     IOD_SyncFunc        sync;
     IOD_FSyncFunc       fsync;
     IOD_UnlinkFunc      unlink;
     IOD_LinkFunc        link;
     IOD_MkdirFunc       mkdir;
     IOD_RmdirFunc       rmdir;
     IOD_AccessFunc      access;

     IOD_OpendirFunc     opendir;
     IOD_ClosedirFunc    closedir;
     IOD_RewinddirFunc   rewinddir;
     IOD_ReaddirFunc     readdir;

     IOD_MoveFunc        move;
};
```

(表 5.7) 有關檔案系統的系統呼叫介

```
int     _access(const char *path, int amode);
int     _close(int fildes);
void    _exit(int status);
int     _fsync(int fildes);  /* non-POSIX */
int     _isatty(int fildes);
int     _link(const char *existing, const char *newfile);
off_t   _lseek(int fildes, off_t offset, int whence);
int     _mkdir(char *path, int mode);
ssize_t     _read(int fildes, void *buf, size_t nbyte);
int     _rmdir(char *path);
int     _sync(void);
int     _unlink(const char *path);
ssize_t     _write(int fildes, const void *buf, size_t nbyte);
int           _closedir(DIR *);
DIR         *_opendir(const char *);
struct    dirent *_readdir(DIR *);
void          _rewinddir(DIR *);
int    _open(const char *path, int oflag, ...);
int    _fcntl(int fd, int cmd, ...);
int    _fstat(int fd, struct stat *sbuf);
int    _stat(const char *path, struct stat *sbuf);
```

在 I/O driver components 中，應用程式使用的每個指令都會有需要傳入的參數，所以 I/O driver components 中(表 5.4)與這個指令對應的函式裡就宣告一個(表 5.8)的結構，儲存這個 C Runtime Call ID，並使用 HostCall_command 結構裡的 parameters 結構，來儲存指令傳入的參數，即(表 5.8)虛線以下資訊。

(表 5.8) HostCall_command 結構(虛線以下由 I/O driver components 填寫，以上由 hostcall library 設定)

```
struct HostCall_command {
    void    *requester;                                  → 儲存目前傳送指令的 thread 資訊
    Int32    status;                                     → 儲存處理進度狀態
    Int32    notification_status;    /* HostCall_status */
    Int32    returned_errno;
    HostCall_Termination_Handler notification_handler;   → 儲存 notification_handler 位址
    HostCall_Termination_Handler termination_handler;    → 儲存 termination_handler 位址
    Int32    code;                   → 儲存指令代號
union {
    struct {
        long        retval;
        long        fildes;
    } close_args;
    struct {
        long        retval;
        char *      path;
        long        oflag;
        long        mode;
    } open_args;
    struct {
        long        retval;
        long        fildes;
        void *      buf;
        long        nbyte;
    } read_args;
    struct {
        long        retval;
        long        fildes;
        void *      buf;
        long        nbyte;
    } write_args;
    struct {
        long        retval;
        long        fildes;
        TCS_Stat    *buf;
    } fstat_args;
    struct {
        long        retval;
        long        fildes;
    } isatty_args;
    struct {
        long        retval;
        long        fildes;
        long        offset;
        long        whence;
    } lseek_args;
    struct {
        long        retval;
        char *      path;
        TCS_Stat    *buf;
    } stat_args;
    struct {
        long        retval;
        char *      path;
    } open_dll_args;
```

以資料結構儲存各個指令所需參數
(目前僅列出 FILE I/O 相關、結束 CRT 服務的函式所需參數結構)

```
struct {                                    }
        long        retval;                 }
        long        fildes;                 }
        long        cmd;                    }
        long        flags;
    } fcntl_args;
    struct {
        long        status;
    } exit_args;
          ⋮
  } parameters;
 }//end for Union;
```

所以 I/O driver componenets 主要就是針對 components 裡各種操作函數，對指令做特定的參數與代號的設定。之後就藉由 hostcall library 提供的 _HostCall_send(HostCall_command *command)函式介面，啟動 hostcall library 來負責 DSP 端後續的 CRT 指令包裝與傳送運作。I/O driver components 裡被呼叫的操作函數都要等到 hostcall library 以下的運作整個執行結束，才會從 ARM 端更改過的 HostCall_command 結構，讀取其參數結構中的 retval 值並將這個值回傳給 TCS C library 或應用程式,如果在進入 hostcall library 以後的運作上出現任何錯誤,hostcall library 或 host_comm library 也會將 HostCall_command 結構的 status 設為 HostCall_ERROR，操作函數就會根據這個 status 判斷有錯誤發生，然後回傳-1 給 TCS C library 或應用程式。

hostcall library—為一個 asyncronous I/O call 介面，負責填入不同操作函數都需要的資訊於指令包裝結構中,並使用 host_comm library 進行 result buffer 的配置與傳送，傳送完指令就接著 suspend thread,讓其他 DSP 端應用程式中的 task 可以繼續處理指令；等 ARM 端回覆後，會使用 notification_handler 喚醒 thread，再繼續使用 host_comm library 提供的 termination_handler 做後續處理或結束這次指令請求

對於後續傳送指令或是針對未來接收到 ARM 端回覆時的處理，不管是哪種操作函數，所做的動作都是類似的，包括儲存處理 ARM 端回覆時所需運作的 callback function 位址(hostcall library 提供 notification_handler、host_comm library 提供 termination_handler)、thread 身分、設定處理進度狀態於指令包裝結構中，還有根據指令決定是否配置 result buffer、傳送包裝完的指令位址給 ARM，這些事便交由 hostcall library 來負責。所以 hostcall library 提供了_HostCall_send(HostCall_command *command)這個函式介面讓每個 I/O driver components 中的操作函數使用，以啟動這些事件的處理。

不過 host_comm library 分擔了一部分 hostcall library 要負責的事情，所以它提供了_HostCall_host_send(HostCall_command *command)這個函式介面給 hostcall library 使用，讓 hostcall library 將設定好 callback function 位址、thread 身分、處理進度狀態的 HostCall_command 結構傳入 host_comm library，由 host_comm library 繼續針對這個指令做記憶體配置與傳送(請看 **host_comm library**)。

在 host_comm library 送出指令位址後，hostcall library 就會 suspend 這個處理指令的 thread(由 5-2-2 節詳述 DSP 端指令包裝與指令位址傳送)，pSos 會繼續執行其他 DSP 端指令處理，原本的 thread 一直要等到收到 ARM 端回傳該指令位址後，才會針對收到的指令內容做後續的處理或回傳的動作，所以 hostcall library 提供的是一種 asyncronous I/O call 介面。為了在收到 ARM 端回覆後可以喚醒原本被 hostcall library suspend 的 thread，hostcall library 於是又提供了 notification_handler(5-2-4 節會詳述喚醒的過程)，並將這個函式位址與 thread 身分於傳送指令位址前，將它們儲存於指令包裝結構中。在交由 host_comm library 處理前，它還會再把處理狀態(status 變數)設為 HostCall_BUSY(*註 1)，表示正在處理這個指令。

之後 5-2-4 節會說明 ARM 端回傳 packet 並對 DSP 端發出中斷後，DSP 端如何喚醒之前被 hostcall library suspend 的 thread，被喚醒後的這個 thread 便從之前被 hostcall library suspend 的地方繼續往下執行，便會執行到之前儲存於 HostCall_command 結構中，由 host_comm library 提供的 termination_handler，來做這個指令處理的結束或是繼續重新呼叫 hostcall library 來做下一次 CRT 請求(結束指令處理的部分請看 5-2-4)。(hostcall library 的運作請看 5-2-2 節)。

host_comm library—協助 hostcall library 配置指令所需資源，並使用 user mode library 做指令位址的傳收，另提供函數來「啟動」喚醒 thread 的動作、判斷指令處理是否執行結束，提供 I/O driver components 創建訊息溝通管道的函式

host_comm library 由 HostIF 與 RPCClient 模組組成，HostIF 模組會利用 user modelibrary 作指令的傳送與接收 ARM 端的回覆、CRT 溝通管道的建置與釋放、喚醒被 suspend 的 thread，而 RPCClient 模組則負責指令的處理，包括傳送前對指令進行的包裝、接收 ARM 端回覆後檢查指令是否處理完畢的動作。

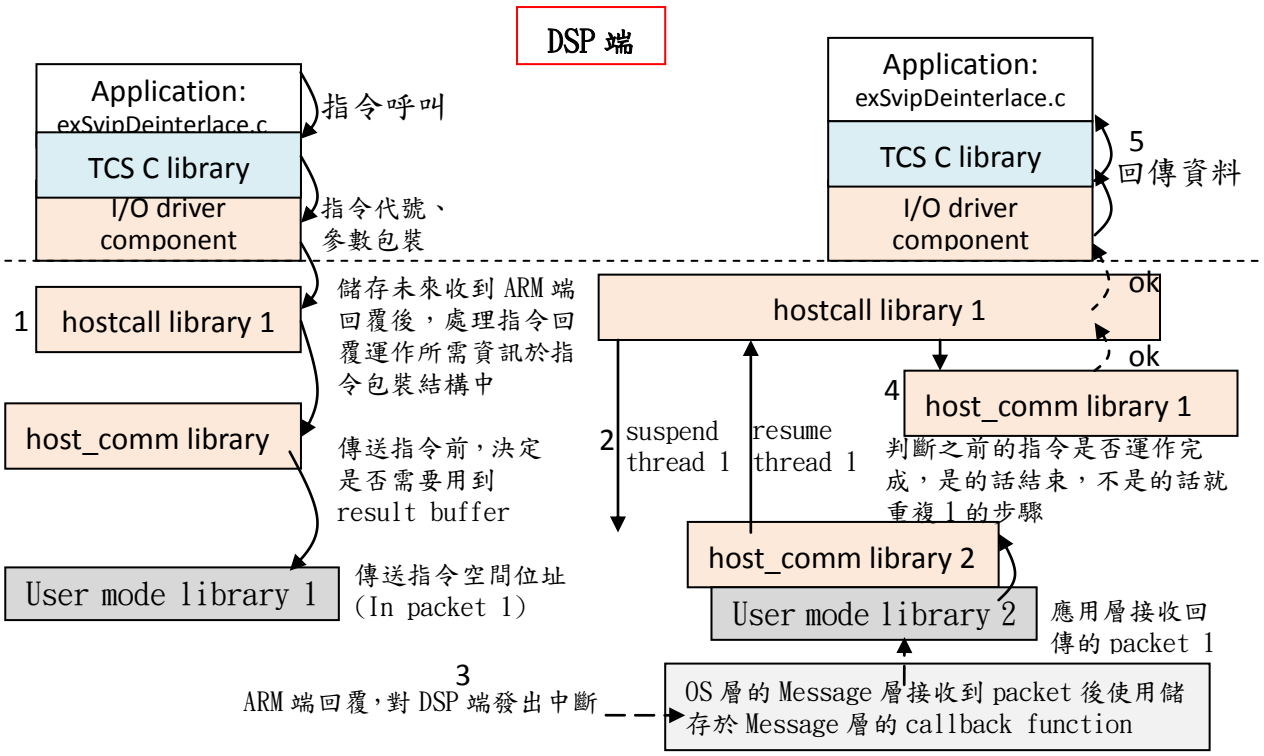
(*註 1) 服務狀態設定有三種，HostCall_BUSY、HostCall_Done、HostCall_ERROR。HostCall_BUSY 表示 DSP 端正要求 ARM 端提供指令服務，HostCall_Done 表示 ARM 端以處理完指令並順利回傳資料回 DSP 端，HostCall_ERROR 則可能出現在指令沒有成功送出、指令沒有包裝成功。

上一段我們提過 host_comm library 會提供_HostCall_host_send 函式給 hostcall library 使用，內部會在由 RPCSend 模組利用 hostcall library 傳入的 HostCall_command 結構中由 I/O driver components 填入的 C Runtime Call ID，決定該指令是否需要 result buffer,RPCSend 模組於是在實體記憶體另外配置一塊 HostCall_command 與 result buffer 所用的空間，將設定好的 HostCall_command 內容複製進去，我們稱這塊空間為 control block，而 control block 的位址則為後來所謂的指令位址(5-2-2 節)，這個指令位址儲存到 packet 結構後,就會使用 HostIF 模組提供的_HostIF_send 函式,將指令位址存入 packet 後再利用 user moder library 傳給 ARM 端。

為了結束並檢查是否已藉由 ARM 的處理完成這個指令請求(是否有未讀取完的資料)，所以 host_comm library 的 RPCClient 模組提供 termination_handler 函式，並於傳送指令位址給 ARM 端前就在 HostCall_command 結構裡儲存 termination_handler 位址(5-2-4 節會詳細說明其運作)，如此一來，等到 ARM 端處理完指令並回覆指令位址給 DSP 端、HostIF 模組提供的 hostif_TMManNotify 函式會使用 user mode library 接收指令位址,並啟動 hostcall library 已儲存的 notification_handler 位址來喚醒之前被 suspend 的 thread，然後就可以利用 termination_handler 對原本還未回傳值給 TCS C library 或應用程式的 thread 進行後續資料讀取的請求，或者如果確定沒有需要再次請求 ARM 端 CRT 服務，就可以成功結束這個指令處理(*註 2)。至此，hostcall library 處理完這次指令請求，而 I/O dirver components 裡的操作函數便可從更新過後的指令包裝內容，回傳資料給 TCS C library 或應用程式(圖 5.4)，完成了此次指令處理。有關兩端之間詳細的運作過程，請看 5-2 節 CRT 運作機制。

另外，因為 HostIF 模組會使用 user mode library 來做指令位址傳收的動作，所以 PNX1005 一啟動，在還未進入 tmMain 應用程式主體之前，也會透過_HostIF_init 來創建 DSP 端溝通管道，其中便透過 tmmanMessageCreate 函式，傳入與 ARM 端創建溝通管道時相同的 service 名字及 OS 層收到訊息後，用來接收訊息與 signal thread 的 hostif_TMManNotify 函式(請看 5-2-1 DSP 端)，來創建 MessageObject 及其內容，而後這個描述溝通管道的 MessageObject 位址便會儲存於 host_comm library 的全域變數中，在使用這個溝通管道進行傳收 packet 時會使用到。

(*註 2)基本上除了做讀檔相關動作的指令，在收到 ARM 端回覆後還有可能需要藉由 termination handler，再次要求 hostcall library 重新啟動 CRT 傳送前準備過程與指令傳送，其餘指令即使呼叫了 termination handler 也只是把 control block 的 HostCall_command 結構內容複製到原本 I/O driver components 宣告的 HostCall_command 中，然後把 control block 裡 HostCall_command 結構內容釋放掉)



(圖 5.4)DSP 端運作概況

5-2 CRT 運作機制

由 5-1-2 節我們已經知道 ARM 端 CRT service module 與 DSP 端 CRT utility 提供的函式介面與各個 library 負責的功能，應用程式就是利用使用 CRT Service Module 的 tmctr library 創建出執行接收 DSP 端指令與處理指令的 CRT thread，我們先看一下創建出來的 CRT thread 在運作上的特性與 CRT thread 在運作上與應用程式主程式之間的關係。

CRT thread 運作特性

應用程式在創建執行接收 DSP 端指令與處理指令的 CRT thread 後，主程式還是可以繼續執行，不過環境中就多了幾個 CRT thread，不停地在等待 DSP 端傳指令過來。我們讓多個 thread 都只從共同的一塊 queue buffer 來接收 DSP 端傳過來的指令，OS 會自動分配好時間，等到指令傳送過來就選擇其中一個 thread 從 queue buffer 中讀取 packet，然後再根據接收到的 packet 裡儲存的指令位址做指令處理，並不是讓每個 CRT thread 都有一個 queue buffer，因為如果每個 CRT thread 就有一個 queue buffer，這樣 DSP 端在傳送系統呼叫的時候，也要同時指定現在這個指令最後要傳給 ARM 端 CRT thread 的哪個 queue buffer，這就增加了程式開發者設計指令溝通的負擔；如果只有一個 queue buffer，對 ARM 端的 OS 層來說，DSP 端傳送 packet 時不需要區分是要塞到哪個 queue buffer，而且因為每個 CRT thread 執行內容都是一樣的，所以我們還善用了 ARM 端 Linux OS scheduling 功能，只要其中一個 thread 有被喚醒就對 queue buffer 做接收即可，而且如果同時有多

個 packet 傳送到 queue buffer，OS 也可以同時讓其他的 thread 來處理 queue buffer 中的 packet，增加處理指令的效率。

然而，因為多個 CRT thread 都只使用到一個 queue buffer，所以控制它們何時可以去讀取 queue buffer 的動作所需的阻斷介面(semaphore)，就必須是大家都看得到的，這樣執行相同動作的 CRT thread，就會試著都去取得這個共同的 semaphore，只要 semaphore 的 counter 不為 0，就可以到 queue buffer 讀取 packet。

CRT thread 在運作上與應用程式主程式之間的關係

因為在 DSP 端沒有結束之前，是不能停掉 CRT 服務的，不然 DSP 端就無法進行檔案系統的操作，所以每次只要 CRT thread 讀取到 packet 並處理完指令，就會以 while loop 再繼續下一回合的等待、試著再去取得 semaphore。但是，主程式在 CRT thread 運作時仍一路往下，程式到最後一定會做到結束 CRT 服務的動作，為了避免它在 DSP 端還沒結束前就自己把 CRT 服務終止掉，所以我們必須讓 DSP 端被啟動後，主程式就在執行結束 CRT 服務動作前先被 block 起來，詳細方法請看(5-2-5 節)。

接下來我們大致跑一下應用程式使用 CRT service module 的程序，詳細內容請看(5-2-1~5-2-5 節)。

在 ARM 端應用程式將 PNX1005 啟動前，ARM 端應用程式必須先創建指令溝通時需要的溝通管道並創建負責接收指令並處理的 CRT 服務的 thread，然後儲存接收到指令後處理指令時所需的環境資訊及最後執行指令動作的檔案系統介面與啟動結束 CRT 服務的介面(圖 5.3)。於是，應用程式使用 tmcrt library 提供的 cruntimeInit 函式用來啟動 CRT server，透過 host_comm library 中的 TM1IF 模組創建 CRT 溝通管道 Message 層的阻斷介面、傳入檔案系統操作函式介面與啟動結束 CRT 服務的函數介面給 RPCServ 模組，然後在環境中創建出提供 CRT 服務 thread 的 thread。

接著，應用程式再使用 cruntimeCreate 函式儲存 server 處理指令時的環境資訊，藉由 TM1IF 模組創建出 CRT 溝通管道，並將創建出來的溝通管道、應用程式讀入的 command line 資訊與標準輸出輸入還有 RPCServ 模組處理指令時作位址轉換時所需的 DSP 端實體記憶體位址與 ARM 端 user space 虛擬記憶體位址的位址差、用來釋放溝通管道的函數位址儲存到 RPCServ 模組中。然後應用程式才利用 user mode library 請求 TMMan 驅動程式啟動 PNX1005。

由於創建完 CRT 服務的 thread 以後，主程式仍繼續執行，為了避免主程式直接執行到結束 CRT 服務 thread 的 cuntrimeExit 函式，所以應用程式會使用 mutex 作為應用程式運作上的阻斷介面，暫時將主程式阻斷在結束 CRT 服務之前。而創建 CRT 服務的 thread

後，CRT server 就啟動，透過 TM1IF 模組創建的阻斷介面將 thread 先 block 在接收指令之前，等待 DSP 端藉由 OS 層完成指令傳輸後，才會由 kernel 去啟動 thread 繼續作接收與處理的動作。

DSP 端在傳送應用程式所下的檔案系統呼叫指令前，必須把裝有指令參數、代表指令的代號與接收到 ARM 端回覆時負責做後續處理的函式位址的 HostCall_command 結構，先填寫好，ARM 端主要就是利用 HostCall_command 結構內容決定要調用系統裡哪個指令來完成操作。而 DSP 端的 I/O driver components 就提供許多包裝檔案系統呼叫的操作函數，並在 pSos 啟動時就將這些操作函數位址儲存到 UID_Driver_t 結構(表 5.6)中，UID_Driver 結構是應用程式檔案系統呼叫與使用 CRT 服務來完成檔案系統呼叫實作函式的介面，所以應用程式使用檔案系統呼叫後，專門處理系統呼叫的模組裡，專門處理該檔案系統呼叫的函式會從 UID_Driver_t 結構中取出該檔案系統呼叫對應到的實質 CRT 服務檔案系統操作函數，也就是 I/O driver components 提供的操作函數，因此就開始進行指令的包裝動作。

I/O driver components 的操作函數首先會將指令代號、指令參數填入 HostCall_command 結構中，然後使用_HostCall_send 函式啟動 asynchronous I/O call 運作機制來處理應用程式請求的檔案系統呼叫(請看圖 5.4)，asynchronous I/O call 的意思就是不管此次系統呼叫是否執行完，該系統呼叫的 thread 都會先被 suspend，讓 DSP 端可以繼續處理其他的系統呼叫處理程序。而 asynchronous I/O call 實作由 hostcall library 負責，它會繼續為操作函數傳下來的 HostCall_command 結構填入 ARM 端回覆 DSP 端時，DSP 端運作所需的函式位址，包括用來喚醒被 suspend 的系統呼叫程序的函式，還有儲存這個系統呼叫程序的身分資訊，以及喚醒後的系統呼叫程序用來檢查此次 I/O call 是否處理完畢的函式。

包裝完後的 HostCall_command 結構便繼續傳給 host_comm library 的 RPCClient 模組，檢查這個系統呼叫是否需要 result buffer 來儲存從 ARM 端讀入的檔案內容，然後在環境中另外配置一塊 HostCall_command 結構內容與 result buffer 空間大小的 control block，最後將這個 control block 位址傳入 HostIF 模組，由它將 control block 位址存入 packet 後，利用 tmmanMessageSend 將 packet 送出。

將 packet 送出後，hostcall library 就會把此次系統呼叫程序 suspend 起來。

DSP 端送出指令請求後，ARM 端 OS 層接收完 packet 將之塞入 queue 以後，CRT 服務的 thread 就會被啟動，thread 便會使用 tmmanMessageReceive 接收 packet，並從 packet 中取出 control block 位址，然後傳入 RPCServ 模組進行指令處理。RPCServ 模組收到 control block 位址後就會先利用先前儲存好的位址差，將 control block 在 DSP 端實體記憶體位址轉成 ARM 端 user space 虛擬記憶體位址，然後 ARM 端就可以在 user space 讀到 control

block 裡的指令內容，根據指令代號呼叫對應的檔案系統操作函式，這些檔案系統操作函式可能來自 kernel，也可能是由 tmcrt library 提供的檔案系統函式介面或啟動結束 CRT 服務的函式介面，傳入檔案系統操作函式的參數也是從 control block 內容讀取出來。

操作完指令後可能會有回傳值，所以 RPCServ 模組會把回傳值填入 control block 的參數欄位內。接著 thread 又會使用 tmmanMessageSend 將原本接收到的 packet 直接回傳給 DSP 端。

DSP 端 OS 層收到 packet 後，Message 層會直接從 packet 裡儲存的 control block 位址找到在 DSP 端 control block 內容，然後呼叫用來喚醒 thread 的函式，此時之前被 suspend 的系統呼叫程序就從被 hostcall library suspend 的地方開始運作，於是又呼叫儲存於 HostCall_command 結構用來檢查此次 I/O call 是否處理完畢的函式，在這個函式中依照 ARM 端填入的回傳值，決定是否還需要繼續下一次讀檔的動作，如果要就再次啟動請求 hostcall library 重新包裝想要讀取的檔案大小以及讀取後要放置的位置，再對 ARM 端繼續發出指令請求，如果 I/O call 已經處理完畢，就結束 asynchronous I/O call，I/O components 函數便可從 HostCall_command 結構中把得到的回傳值回傳給應用程式。

5-2-1 ARM 端與 DSP 端的 CRT 運作初始化

ARM 端—應用程式使用 cruntimeInit 與 cruntimeCreate 函式

ARM 端應用程式必須在啟動 PNX1005 前就先利用 cruntimeInit 函式，交由 host_comm library 使用 tmmanSyncobjCreate 函式，在 kernel 中創建一個 semaphore 結構並將其 counter 初始為 0，提供 CRT 服務 thread 裡做 tmmanMessageReceive 前所使用的 OS 層與應用層阻斷介面(CRT 服務 thread 實體請看 5-2-3 節)，然後 host_comm library 會再繼續使用 pthread_create 函式創建幾個 thread 來作 CRT 服務的指令接收處理程序(圖 5.5)。

另外，cruntimeInit 函式會將 tmcrt library 中提供的 FILE I/O 函數介面及啟動結束 CRT 服務的函數介面(表 5.2)透過 host_comm library 的 RPCServ_init 函式，將這些函式位址一一儲存給 host_comm library 裡 RPCServ 模組的函式指標(表 5.9)，這樣 ARM 端的 RPCServ 模組之後就可以利用這些函式來完成 DSP 端的指令請求。

```
for (i = 0; i < NROF_SERVERS; i++)
{
    pthread_t thread;
    status = pthread_create(&thread, NULL, (void *) &tmlif_serve,//5-2-3 會說明
                                    (Pointer) gNumCreatedServers);

    /*儲存每個被成功開啟的 thread 的 ID，作為結束 thread 時所需資訊*/
    if (status == 0)  gMsgServers[gNumCreatedServers++] = thread;
    else  continue; }
```

(圖 5.5)創建 CRT 服務的 thread:定義 NROF_SERVERS 變數值，決定要創建的 thread 個數

(表 5.9) RPCServ_init 函式(左半部為 host_comm library 裡的函式指標，右半為(表 5.2)函數位址)

```
Bool RPCServ_init (
    RPCServ_OpenFunc     d_open,      // I: open func
    RPCServ_OpenDllFunc  d_open_dll,  // I: open dll func
    RPCServ_CloseFunc    d_close,     // I: close func
    RPCServ_ReadFunc     d_read,      // I: read func
    RPCServ_WriteFunc    d_write,     // I: write func
    RPCServ_SeekFunc     d_seek,      // I: seek func
    RPCServ_IsattyFunc   d_isatty,    // I: isatty func
    RPCServ_FstatFunc    d_fstat,     // I: fstat func
    RPCServ_FcntlFunc    d_fcntl,     // I: fctrl func
    RPCServ_StatFunc     d_stat,      // I: stat func
    RPCServ_ExitFunc     d_exit,      // I: exit func
    Endian               d_endian )   // I: node's endianness
```

接著，為了讓 ARM 端可以進行指令溝通，應用程式需要使用 cruntimeCreate 函式，其中藉由 TM1IF_add_node_info 函式來使用 tmmanMessageCreate 函式(圖 5.6)，建立 ARM 端負責做 CRT 指令溝通的訊息溝通管道，由於 CRT thread 必須透過之前創建的 OS 層與應用層的阻斷介面決定應用層的 CRT thread 是否可以做接收的動作，所以創建溝通管道的同時會把這個 semaphore 位址(阻斷介面)儲存到 Message 層中。

(圖 5.6)ARM 端創建 CRT 指令溝通管道

而且 TM1IF 模組開啟的 CRT 服務 thread 在運行時，運行 thread 實體的 RPCServ 模組函式 RPCServ_serve 需要使用到的環境資訊也需要事先準備好，所以 RPCServ 模組中準備了(表 5.10)的 RPCServ_NodeInfo 結構，儲存(5-1-2-1)介紹 cruntimeCreate 函式時所提到的各種環境資訊於 host_comm library 的 RPCServ 模組中。cruntimeCreate 函式並利用 TM1IF_add_node_info 函式，其中使用 tmmanMessageCreate 函式創建 CRT 指令溝通管道，並將其 MessageObject 結構位址儲存於 tmcrt library 的變數中。

等到 cruntimeCreate 執行完，ARM 端應用程式才會要求啟動 PNX1005(複製 DSP 端可執行檔的動作之前已做完)，啟動後的 PNX1005 才開始 DSP 端的 CRT 初始化。

(表 5.10)host_comm library 儲存運作所需環境資訊的結構

```
typedef struct {
          UInt32    node_number;
          UInt32    argc;
          String   *argv;
          UInt32    Stdin;
          UInt32    Stdout;
          UInt32    Stderr;
          UInt32    address_shift;
          UInt32    sdram_base;
          UInt32    sdram_limit;

          RPCServ_Node_Terminator  terminate;
          Pointer                  data;
          stackTraceData_t     stackInfo;
} RPCServ_NodeInfo;
```

```
typedef struct
{
    UInt32  dsp_number;
    UInt32  dsp_handle;
    UInt32  hostcall_channel;//溝通管道位址(MessageObject 位址)
    Bool    valid;
} NodeData;
```

DSP 端

DSP 端所有要執行的程式(包括 pSos)，都會在編譯時包進 DSP 端可執行檔中，所以當 ARM 端把 DSP 端可執行檔下載到 DSP 端 RAM 並啟動 PNX1005 後，DSP 端在執行應用程式進入點以後的程式前(tmMain 以下的主體應用程式)，pSos 就會使用 active 函式(圖 5.7)，對CRT運作環境進行初始，並不需要在DSP端應用程式主體(tmMain)裡特別做啟動的動作，初始結束後，CRT utility 在 DSP 端環境中就已經存在，恭候應用程式主體搭配 TCS C library 來使用。

在 active 函式中，會使用 HostIF 模組提供的_HostIF_init 函式來完成 CRT 運作時所需的 OS 層資源。它會運行與 ARM 端 init_module 同類功能的程序進行 OS 層運輸時所需 Vintr 層、HAL 層資源以及各層 ManagerObject 結構的創建，還有從共享記憶體的 NameSpace 區塊讀取 ARM 端已經配置過有關 CRT 溝通管道的 CRT 服務代號與 type，這樣 DSP 端 host_comm library 的 HostIF 模組就可以使用與 ARM 端 user mode library 同樣功能的訊息傳遞函式介面，藉由先儲存、分配好的資源來運作指令的溝通。然後，在_HostIF_init 函式中，還會再使用 tmmanMessageCreate 函式創建訊息溝通管道(圖 5.8)。

active 函式而後還會將 5-1-2-2 節介紹的 I/O driver components 的操作函數位址儲存到 UID_Driver_t 結構中的各個函式指標去，這個 UID_Driver_t 就像是 Linux 環境下的 file_operations 結構，應用程式所下指令經由 TCS C library 轉換後，可以透過系統指

令呼叫，查找註冊於 UID_Driver_t 結構中的操作函數指標，每個系統指令呼叫都會有一個操作函數指標與之對應，然後使用函式指標指到的函式進行後續 CRT client 端的運作。

```
 static tmLibdevErr_t activate(pcomponent_t comp)
{
    _HostCall_init();//創建 CRT 指令溝通管道
    _StdDrivers_init();//用來將 I/O driver components 儲存於 pSos
    initialized = True;
    return TMLIBDEV_OK;
}

static Bool _StdDrivers_init( void )
{
    return
      IOD_install_fsdriver(
        RecogFunc, InitFunc, TermFunc,
        OpenFunc, OpenDllFunc, CloseFunc,
        ReadFunc, WriteFunc, SeekFunc, IsattyFunc, FstatFunc,
        FcntlFunc, StatFunc, SyncFunc, FSyncFunc, UnlinkFunc,
        LinkFunc, MkdirFunc, RmdirFunc, AccessFunc,
        OpendirFunc, ClosedirFunc,
        RewinddirFunc, ReaddirFunc,
        MoveFunc
      ) != NULL;
}
```

(圖 5.7)active 函式

```
tmmanMessageCreate (
        DSPHandle,
        "CRunTime", /* message ID - should be the same on the host */
        (UInt32)hostif_TMManNotify,
        constTMManModuleTargetCallback,
        &hostcall_channel )
static TMStatus hostif_TMManNotify ( void )
{
    TMStatus status;
    tmmanPacket Packet;
    Pointer command;
    while ((status = tmmanMessageReceive(hostcall_channel, &Packet)) ==
            statusSuccess)
    {
        command = (Pointer)Packet.Argument[0];//指令位址
        _RPCClient_notify(command);
    }
    status = statusSuccess;
    return status;
}

void _RPCClient_notify ( HostCall_command *command )
{
   command->notification_handler( command );
}

static void notify( HostCall_command  *command )
{    AppModel_resume ( command->requester ); }
```

指向一個啟動喚醒 DSP 端 thread 動作的函式

(圖 5.8)DSP 端創建 CRT 指令溝通管道

在創建 DSP 端指令溝通管道時，同樣需要給予 DSP 端 OS 層與應用層一個阻斷介面，但是因為 DSP 端應用程式只有在需要的時候才會使用系統指令，DSP 端傳完指令位址後，為了以 asynchronous I/O 的方式來建構 DSP 端的系統指令處理運作，我們不希望處理該指令的 thread 在等待 ARM 端回覆的時候還需要由系統不停檢查 semaphore 釋放與否，所以 DSP 端會透過 hostcall library，使用 AppModel_suspend_self 函式直接把處理這個指令的程序 suspend 起來，我們因此不需要額外使用 semaphore 做為阻斷介面。

而後，DSP 端等待 ARM 端回覆、對 DSP 端發出中斷，DSP 端收到中斷後 OS 層的運作將 packet 塞入 Message 層的 queue buffer，需要的就是一個可以讓 DSP 端 OS，做喚醒這個被 suspend 的 thread 的動作，所以 DSP 端在為 CRT 創建溝通管道時，在 Message 層儲存的則是一個 callback function，這個 function 必須先從 queue buffer 中讀取 ARM 端回傳的、存有指令位址的 packet，然後對這個 thread 啟動喚醒動作，才可以由傳送指令前、hostcall library 儲存於 HostCall_command 結構中的函式指標指到的 notification_handler、thread 身分，利用 AppModel_resume 對這個 thread 做喚醒動作。

5-2-2 DSP 端傳送指令

當 DSP 端應用程式使用了某個 FILE I/O 的 C I/O call，這個指令會藉由 TCS C library 轉成 POSIX I/O 方式的指令，然後呼叫 I/O driver components 中對應的 FILE I/O 操作函數，將代表這個指令的 C Runtime Call ID 以及這個指令用到的參數先填入 HostCall_command 資料結構中(表 5.6)，(圖 5.9)以 fwrite/write 函式對應到的 FILE I/O 操作函數為例，認識一下操作函數的概觀。

```
static Int32 WriteFunc ( Int32   file, Pointer   buf, Int32      nbyte )
{
    HostCall_command command;

    command.code                          = HostCall_WRITE;
    command.parameters.write_args.fildes = file;
    command.parameters.write_args.buf     = buf;
    command.parameters.write_args.nbyte   = nbyte;

    _HostCall_send( &command );//啟動 hostcall library 運作

    if (command.status == HostCall_ERROR) {
        errno= EAGAIN;
        return -1;
    } else {

        return( command.parameters.write_args.retval );
    }
}
```

（圖 5.9）fwrite/write 函式對應到的 I/O 函數

接著他會透過 hostcall library 提供的 CRT client 介面函式--_HostCall_send，將 HostCall_command 結構位址傳給 hostcall library，開始了一連串 CRT 處理，而 hostcall library 也是處理指令的 asynchronous call 介面，操作函數必須等到 hostcall library 執行結束，才表示 CRT 的動作處理完也取得回傳值，然後再把這個回傳值回傳給 TCS C library。使用_HostCall_send 函式進入 hostcall library 後在傳送指令時進行的 CRT 處理分為兩階段：

第一階段—進階包裝與傳送

不管是何種操作函數傳給 hostcall library 的 HostCall_command 資料結構，hostcall library 會再填入未來處理 ARM 端回覆時所需的資訊，有儲存處理 ARM 端回覆時所需運作的 callback function 位址(hostcall library 提供 notification_handler、host_comm library 提供 termination_handler)、thread 身分、處理進度狀態(5-1-2-2 的 hostcall library 與 host_comm library)，(表 5.11)以寫檔為例列出 I/O driver components 函式使用 WriteFunc 函數與 hostcall library 填寫完 HostCall_command 結構的結構內容。

然後 hostcall library 會將這個 HostCall_command 結構位址，藉由 host_comm library 提供的_HostCall_host_send 函式開始進階指令包裝，於是由負責處理指令的 RPCClient 模組中的 RPCSend 模組根據結構中的 C Runtime Call ID 決定是否配置 result buffer(寫檔動作就不需要 result buffer)，然後另外在記憶體配置 result buffer 加上 HostCall_command 結構大小的 control block 後(寫檔動作就只有含 HostCall_command 內容的 control block)，將 hostcall library 傳給 RPCClient 模組的 HostCall_command 結構內容複製到 control block 中，然後 HostIF 模組提供的_HostIF_send 函式會再將 control block 的位址當作指令位址，儲存於 packet 結構的 Argument[0]變數中，藉由 user mode library 的 tmmanMessageSend 函式將 packet 藉由初始時 HostIF 模組創建出的 CRT 指令溝通管道，由第四章 OS 層的溝通機制(在此不再詳述)傳送給 ARM 端(圖 5.10)。

(表 5.11) DSP 端應用程式呼叫 fwrite 或 write 函式後，經 I/O driver components 與 hostcall library 填入的 command 資訊

```
struct HostCall_command :
command.code            = HostCall_WRITE;                    } I/O driver components 的
command.parameters.write_args.fildes    = file;              } WriteFunc 所填寫
command.parameters.write_args.buf       = buf;               }
command.parameters.write_args.nbyte     = nbyte;             }

command->requester              = AppModel_current_thread;   } hostcall
command->status                 = HostCall_BUSY;             } library
command->notification_status    = HostCall_BUSY;             }
command->termination_handler = (HostCall_Termination_Handler)termination_handler
command->notification_handler = notify;
command->returned_errno       = 0;
command->sending_node         = _node_number
```

```
void _HostCall_host_send( HostCall_command *command )

{
    HostCall_command *shadow_command = Null;

    /*--------------------RPCSend 模組-------------------------*/
    Switch (command->code) {//判斷各種 C Runtime Call ID
        case  case HostCall_ARGV_ARGC_INFO:
        case HostCall_ISATTY:
        case HostCall_LSEEK:
        case HostCall_CLOSE:
        case HostCall_FCNTL:
        case HostCall_EXIT:
        case HostCall_REWINDDIR:
        case HostCall_CLOSEDIR:
        case HostCall_FSYNC:
        case HostCall_SYNC:
        case HostCall_SOCK_STATUS:
        case HostCall_SOCK_DATA:

        standard_RESULT_BUFF_DEF:
             /*
            儲存 termination_handler 位址於 HostCall_command 結構中，另外在記憶
            體配置只有 HostCall_command 結構大小的 control block (shadow_command
            儲存其位址)，將 hostcall library 傳給 host_comm library 的
            HostCall_command 結構內容複製到 control block 中
            */
            break;
        case HostCall_READ:
            /*
            儲存 termination_handler 位址於 HostCall_command 結構中，另外在記憶
            體配置 result buffer 加上 HostCall_command 結構大小的 control
            block(shadow_command 儲存其位址)後，將 hostcall library 傳給 host_comm
            library 的 HostCall_command 結構內容複製到 control block 中
            */
            Break;
        case HostCall_WRITE: goto standard_RESULT_BUFF_DEF;
        default:
                    shadow_command  = Null;
                    command->status = HostCall_ERROR;
                    break;
    }//end of command->code
    /*-----------------------------------------------------------*/
    _HostIF_send (shadow_command);//傳送指令位址
}

Bool _HostIF_send( Pointer command )
    packet.Argument[0] = (UInt32)command;//將指令位址儲存於 packet 中
    do {
        status = tmmanMessageSend(hostcall_channel, &packet);
    } while (status == statusChannelMailboxFullError);//如果沒傳成功要重送
    return True;
}
```

(圖 5.10)host_comm library 從 hostcall library 接收到 HostCall_command 結構內容後的處理

第二階段—suspend 處理指令的 thread

host_comm library 的 HostIF 模組將存有指令位址的 packet 被送出後，hostcall library 會檢查 HostCall_command 結構中目前的處理進度狀態，如果是 HostCall_BUSY，就會使用 AppModel_suspend_self 函式將這個等待 ARM 端處理指令的 thread suspend 起來，表示 DSP 端已送出訊息，正等待 ARM 的處理與回覆，讓其他 DSP 端可以繼續為其它指令做處理(圖 5.11)。

```
void _HostCall_send( HostCall_command  *command )
{
    command->requester            = AppModel_current_thread;
    command->status               = HostCall_BUSY;
    command->notification_status  = HostCall_BUSY;
    command->termination_handler  = NULL;
    command->notification_handler = notify;                    第一階段
    command->returned_errno       = 0;
    command->sending_node         = _node_number;

    _HostCall_host_send( command );//請見程式 5
   switch (command->status) {
    case HostCall_BUSY:
        AppModel_suspend_self();  ──────────────────────►  第二階段

        if (command->termination_handler) {
            command->termination_handler(command);//5-2-4 節會用到
        }
        command->status = command->notification_status;
        break;
    case HostCall_DONE:
        if (command->termination_handler) {
            command->termination_handler(command);
        }
        break;
    case HostCall_ERROR:    break;
   }
    if (command->returned_errno) {errno = command->returned_errno; }
}
```

(圖 5.11)hostcall library 的兩階段 CRT 指令傳送處理

5-2-3 ARM 端指令處理

ARM 端收到中斷後，會由 ARM 端 OS 層完成底層的訊息存取，最後會將 DSP 端傳出的 packet 塞入 ARM 端在 5-2-1 節由 cruntimCreate 使用 tmmanMessageCreate 函式創建的 CRT 溝通管道的 Message 層 queue buffer 中，並對儲存於 Message 層的 semaphore 做 signal 的動作，而這個 queue buffer 則是由應用程式使用 cruntimeInit 函式，於其中使用 TM1IF 模組的 TM1IF_start_serving 函式創建的 CRT 服務 thread 來讀取。

CRT 服務 thread 的函式實體設計

CRT 服務 thread 的函式實體 tmlif_serve 為提供 CRT 服務的核心(圖 5.12)，負責指令接收、處理及回覆的程序，是設計於 host_comm library HostIF 模組的一個函式。因為每個 thread 就像一個 CRT server，只要 DSP 端沒有結束或是使用 ctrl-c 強迫結束 ARM 端應用程式，我們就希望它不停運轉，所以設計時基本上會使用一個無窮迴圈(除非有需要終止服務，終止情況的探討於 5-2-5 說明)，每經一輪，就使用 tmmanSyncobjWait(&semaphore)試著對應用程式使用 cruntimeInit 函式時由 TM1IF 模組創建的 semaphore 的 counter 做減一的動作，如果 semaphore 是 0 就表示沒有資源可以減了，這個 CRT thread 就會暫時被 block 住，如此一來， tmlif_serve 就暫時無法執行 tmmanMessageReceive 的動作。

直到 OS 層接收到 DSP 端傳來的 packet 並塞入 Message 層的 queue buffer 中，才由 Message 層去 signalthread、對 semaphore 的 counter 加一，此時 CRT thread 才可以繼續往下執行，使用 tmmanMessageReceive 函式讀取 DSP 的 packet，得到 packet 後的 tmlif_serve 便會用這個 packet 的指令位址來做下一段介紹的指令處理，指令處理完 tmlif_serve 會再使用 tmmanMessageSend 函式將指令位址存到 packet 裡送回 DSP 端，做完傳送後，才會進入下一個迴圈。不過基於 5-1-2-1 節提到的多個 thread 同時讀取同一個溝通管道裡 queue buffer 的 packet 問題，這個阻斷介面 semaphore 位址必須儲存於一個全域變數，這樣每個 thread 才可以藉由同一個 semaphore 來控制它們對同一個 queue buffer 的讀取。

```
static UInt32    gEvMsgSync;//全域變數


static void tmlif_serve(int id)
{
    int i;
    TMStatus tmStatus;

    while (!gServingStopped)//迴圈
    {
        tmmanPacket packet;

        tmmanSyncObjWait(gEvMsgSync); //嘗試取得 semaphore 以繼續往下執行

        for (i = 0; !gServingStopped && (i < gNumDsps); i++)
        {
            if (gNodeData[i].valid) {
         tmStatus =tmmanMessageReceive(gNodeData[i].hostcall_channel,&packet);


        /*成功接收，就對收到的 packet 進行資訊解析與指令處理*/
          if (tmStatus == statusSuccess)
          {
```

```
                Pointer raw_command;
                HostCall_command *command;

                raw_command = (Pointer) packet.Argument[0];//取得指令位址
                command->notification_status = HostCall_BUSY;

        /*執行負責藉由指令位址來獲取指令資訊並加以處理的函式，處理完進行回傳*/
                if (RPCServ_serve(raw_command, i) && gNodeData[i].valid)
                    tmmanMessageSend(gNodeData[i].hostcall_channel, &packet);
            }
        }
    }
    // wake up next remaining (if any) server so it can terminate too
    (void) tmmanSyncObjSignal(gEvMsgSync);
}
```

（圖 5.12）CRT 服務的 thread 函式實體—tmlif_serve

指令處理：讀取指令位址→處理指令的函式—RPCServ_serve

ARM 端 CRT service thread 會使用 tmmanMessageReceive 得到 DSP 端傳送過來的 packet，而這個 packet 結構的 Argument[0]則存有 DSP 端儲存的指令位址。於是，取得 packet 後，tmlif_serve 會從 packet 取得 control block 在 DSP 端的實體記憶體位址。

因為這個 control block 位址是 DSP 端 RAM 的實體記憶體位址，ARM 端無法直接藉由這個位址來讀取仍在 DSP 端記憶體裡的 control block 內容，但是在 ARM 端應用程式一開始執行時，3-4 節提到為了讓應用程式可以使用 user mode library 來操作 PNX1005，所以應用程式一開始就會執行 tmmanInitialze，其中，它就會使用 tmmanMapDeviceMemory 函式將 DSP 的 RAM aperture 空間映射到 ARM 端 user space 來，因此 DSP 端 RAM aperture 與 ARM 端 user space 已經可以互相 mapping。ARM 端應用程式於是可以在 user space 在特定虛擬記憶體空間對 DSP 的 RAM 進行操作。

而 5-1-2-1 節 ARM 端應用程式使用 cruntimeCreate 時存入 RPCServ 模組的許多環境資訊中，已經包含了 DSP 端 RAM 實體記憶體起始位址映射到 ARM 端虛擬記憶體 user space 起始位址減掉 DSP 端 RAM 實體記憶體起始位址後得到的差。所以當 tmlif_serve 取得 control block 位址後，tmlif_serve 會將這個位址傳給負責做指令處理的 RPCServ_serve 函式(圖 5.13)，在這個函式裡，首先就會將 control 位址加上 host_comm library RPCServ 模組裡儲存的位址差，得到 ARM 端虛擬記憶體中用來控制 control block 的虛擬記憶體位址，於是，CRT 服務的 thread 就可以在 user space 讀寫 control block 的內容，。
有了 control block 內容，就根據 control block 裡 HostCall_command 結構的 C Runtime Call ID，決定要呼叫哪一個 tmcrt library 在 ARM 端初始化時儲存於 RPCServ 模組的檔案系統的 I/O 函數介面、啟動結束 CRT 服務的函數介面(表 5.2)或直接使用 linux kernel 提供的函式來完成真正的指令動作，而傳入這些函數介面的參數則來自 control block 中

的 HostCall_command 結構、該指令使用到的參數結構，如果參數結構有儲存 DSP 端的實體記憶體來源位址或目的地位址，都會先加上位址差後才傳入函數介面(例如讀寫檔案時)。如果執行的指令會有回傳值，像寫檔就要回傳寫了多少 bytes、開檔就要回傳 file descriptor 等，RPCServ_serve 函式就會把回傳值填寫到 control block 的 HostCall_command 結構中、該指令使用的參數結構的 retval 變數，如果像讀檔這種有 result buffer 的 control block，則直接使用 read 函數透過 PCI 匯流排將資料複製從 ARM 端記憶體複製到 result buffer 中。RPCServ_serve 處理完指令後，就會將 HostCall_command 結構的 notification_status 狀態設為 HostCall_DONE，表示 ARM 端已成功處理完。

```
BoolRPCServ_serve (
    Pointer pRawCmd,  // I: raw (target) pointer to command; must be
                      //    address-translated and endian-converted
    int nodeNumber    // I: node (TM/board) that sent command     )
{
   RPCServ_NodeInfo  *ninfo;
   HostCall_command  *command;
   HostCall_command  *wcommand;
   ninfo   = RPCServ_raw_to_info (pRawCmd, nodeNumber);
/*將 DSP 端實體記憶體位址轉到 ARM 端虛擬記憶體的 user space*/
   command = (Pointer) COMMAND_BUFFER (pRawCmd);
   wcommand = command;   /* non-MMU: wcommand and command are same thing */
    switch (convert_tcs_long (command->code))
    {
        /* ------------------------------------------- */
         case HostCall_OPEN:
         wcommand->parameters.open_args.retval =        使用 FILE I/O 操作函數介面
            convert_host_long (
             parm_open (IN_BUFFER (command->parameters.open_args.path),
                   convert_tcs_o    (command->parameters.open_args.oflag),
                   convert_tcs_long (command->parameters.open_args.mode)));
        break;
        /* ------------------------------------------- */
        case HostCall_WRITE:
         wcommand->parameters.write_args.retval =       使用 FILE I/O 操作函數介面
            convert_host_long (
             parm_write (
                convert_tcs_long (command->parameters.write_args.fildes),
                IN_BUFFER (command->parameters.write_args.buf), 傳入轉換成 ARM 端虛擬記憶體的位址
                convert_tcs_long (command->parameters.write_args.nbyte)));
         wcommand->returned_errno       = convert_host_errno (last_io_errno ());
         wcommand->notification_status = convert_host_long (HostCall_DONE);
              break;
        /* ------------------------------------------- */
        case HostCall_UNLINK:
        wcommand->parameters.unlink_args.retval =       直接使用 kernel 提供的函數
                 convert_host_long (
                  unlink ( IN_BUFFER (command->parameters.unlink_args.path)));
              wcommand->returned_errno  = convert_host_errno (last_io_errno ());
              wcommand->notification_status = convert_host_long (HostCall_DONE);
          break;
        /* ------------------------------------------- */
```

```
                                                    釋放完溝通管道後
                                                    使用啟動結束 CRT 服務的函數介面
        case HostCall_EXIT:
             host_exit (ninfo, convert_tcs_long (command->parameters.exit_args.status));

           wcommand->returned_errno       = 0;
           wcommand->notification_status  = convert_host_long (HostCall_DONE);
          break;
        /* ------------------------------------------- */
                        ⋮


        default:
         wcommand->returned_errno      = convert_host_errno (ENOSYS);
         wcommand->notification_status = convert_host_long (HostCall_ERROR);
         break;
    }//end of switch
    PCI_STABILISE (&command->code, &saved);
    return True;
}
```

(圖 5.13)負責做指令處理的 RPCServ_serve

回覆 DSP 端:tmmanMessageSend

處理完 DSP 請求的指令後，CRT 服務 thread 又把存有 control block 位址(為 DSP RAM 位址)的 packet，利用 tmmanMessageSend 函式藉由 OS 層的指令溝通機制送給 DSP 端，最後 packet 就會被塞入 DSP 端 Message 層的 queue buffer 中，DSP 端才會再由 5-2-1 節註冊於 Message 層的 hostif_TMManNotify 函式啟動後續的 DSP 端處理(5-2-4 節)。

5-2-4 DSP 端接收 ARM 的回傳

接續5-2-2，DSP端處理ARM端的回傳主要有兩個動作要做，一個是喚醒在DSP端傳送packet之後被suspend的處理指令的thread，另一個是由termination_handler來檢查是否需要對ARM端做進一步的指令請求，不然就會結束I/O driver components中hostcall library的運行，然後I/O driver components才可以對TCS C library或應用程式做回傳。

喚醒之前由 hostcall library suspend 的處理指令的 thread

5-2-1節曾說明，應用程式使用cruntimeCreate創建CRT溝通管道時，會註冊ARM端回覆DSP端時，用來啟動signal動作的函式hostif_TMManNotify的位址，此處的signal動作是要去喚醒之前被hostcall library suspend的處理指令的thread。

當DSP端Message層的queue buffer被塞入新的packet，OS就會呼叫註冊於Message層的hostif_TMManNotify函式，這個函式會先利用tmmanMessageReceive從queue buffer中取出packet，再從這個packet中取出control block位址，因為這個位址本來就是DSP端的實體記憶體位址，所以不需要對位址做轉換。

利用control block內容儲存的之前被block的thread的身分與notification_handler位址，就可以透過真正用來喚醒thread的notification_handler，使用AppModel_resume函式喚醒具有這個thread身分的thread，被喚醒的thread就會從之前被suspend的地方(圖5.11的AppModel_suspend_self)繼續執行該thread原本HostCall_command結構裡儲存的termination_handler。

使用 termination_handler 檢查指令處理是否執行完畢

被喚醒的處理指令的thread，會由原本運行的hostcall library程序使用當時的HostCall_command結構裡儲存的termination_handler去檢查是否有需要再次請求ARM端CRT的服務(由於termination handler再對ARM發出重新請求只有在讀檔時才會用到，所以有關讀檔時termination handler運作請看附錄H)，是的話就先重新調整control block裡被ARM端修改完的HostCall_command結構內容，然後把HostCall_command結構內容複製回原來hostcall library運作程序使用的HostCall_command空間，再釋放掉這次傳遞指令使用的control block空間，而後直接傳入更改完的HostCall_command結構內容給_HostCall_Send函式，於同一個指令處理程序，重新啟動5-2-2節談到的hostcall library運作，但是此時處理指令的程序並沒有回傳給TCS C library或應用程式，它是一直在hostcall library這個asynchronous 介面下不斷做處理，除非到喚醒處理指令的thread執行termination_handler時，確定沒有要做對ARM端的請求了，才會直接把control block裡被ARM端修改完的HostCall_command結構內容複製回原來hostcall library運作程序中使用的HostCall_command空間，再把control block釋放掉，然後hostcall library會將notification狀態(HostCall_DONE)設定到command->status(處理進度狀態)，表示DSP端已經完成這回指令，就可結束hostcall library的運行。

hostcall library運行結束，就代表這個I/O driver components的函數介面處理完了CRT運作，於是就這些函數繼續從HostCall_command結構內容讀取參數結構中的retval值，並將之回傳給TCS C library或應用程式，當然如果本身函數就不需要做回傳的動作(例如提出終止CRT服務請求的_report_exit_status函式)，termination_handler執行完等於結束了這個指令處理動作。

5-2-5 ARM 端 CRT 服務的終止

因為應用程式使用 cruntimeInit 函式開啟的 CRT thread，thread 實體函式基本上使用了 while loop 不停的為 DSP 端提供 CRT 服務，所以唯有 DSP 端應用程式結束前請求 ARM 端結束 CRT 服務或是 ARM 端應用程式因為 ctrl-c 被強迫終止，才會由 ARM 端應用程式呼

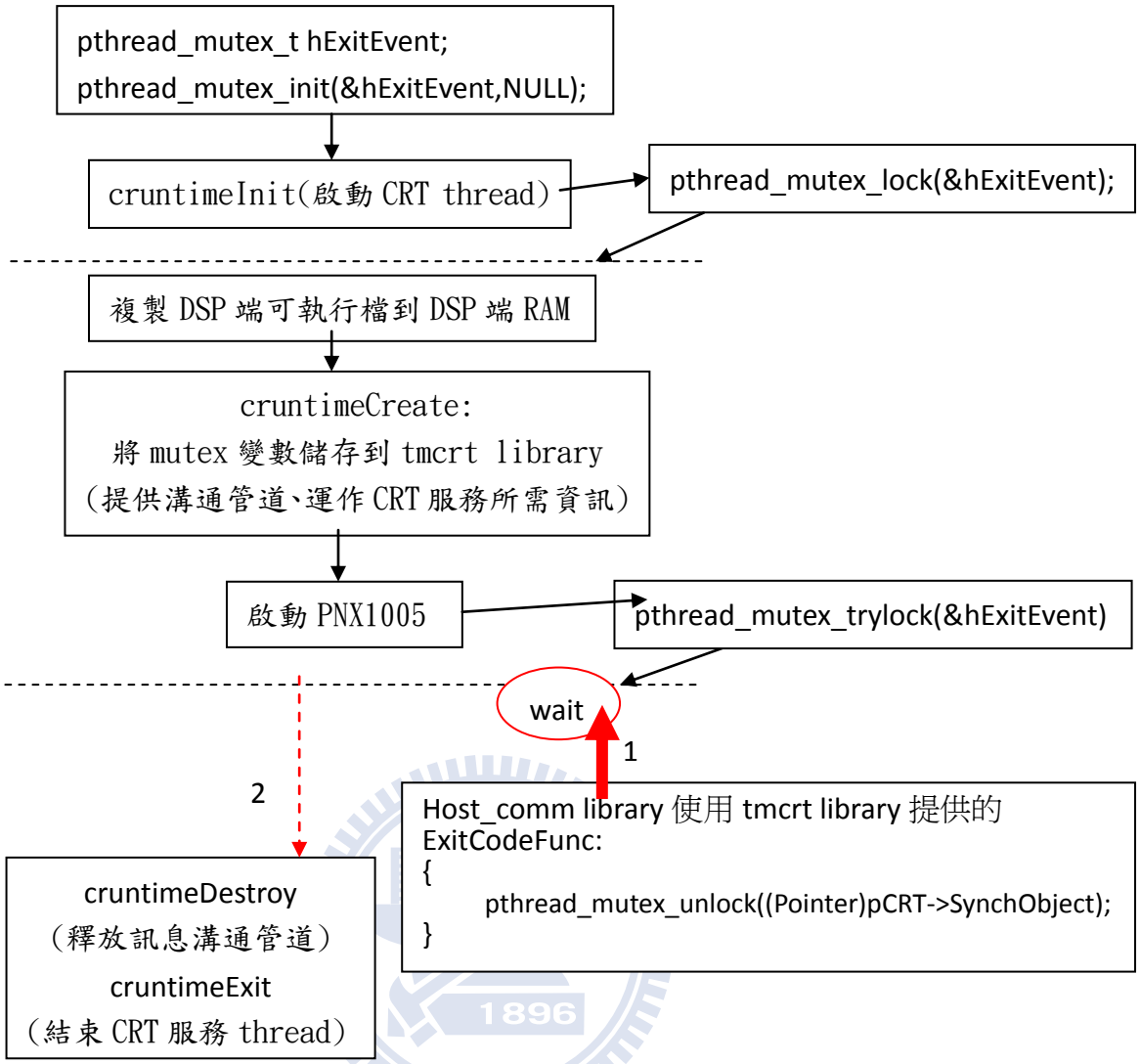
叫 cruntimeExit 函式，對在 host_comm library 的 TM1IF 模組裡運作的 CRT 服務 thread 做終止與資源釋放。知道如何啟動 cruntimeExit 後，接著就要再探討 cruntimeExit 本身如何終止 CRT 服務。

如何啟動 cruntimeExit

為了讓 ARM 端應用程式可以進行 CRT 服務的終止，應用程式會先在 user space 宣告一個全域的 mutex 變數並利用 pthread_mutex_init(&mutex,NULL)對 mutex 變數進行初始化。在應用程式使用 cruntimeInit 創建 CRT 服務的 thread 後，就使用 pthread_mutex_lock 函式將這個 mutex 鎖定，並把應用程式結束 CRT 服務的動作(cruntimeExit)寫在 pthread_mutex_trylock 這個 mutex 後。於是當 ARM 端應用程式啟動 PNX1005 後，去使用 pthread_mutex_trylock 函式試著再去鎖定這個 mutex，但因為這個 mutex 還沒有被解除鎖定，所以應用程式就等著鎖定的解除，無法繼續執行下面的 cruntimeExit 函式，所以只要這個 mutex 沒有被解除鎖定，就不會執行結束 CRT 服務的動作(圖 5.14)。

而鎖定的解除，必須搭配上述的兩種終止情況。如果是 DSP 端應用程式呼叫 tmMain_EXIT()/_psos_exit()後，就會使用 CRT 機制請求 ARM 端做終止的動作，而終止的動作就是由 5-2-3 節 ARM 端指令處理時，使用 tmcrt library 提供給 RPCServ 模組的 ExitCodeFunc 函式來做，函式中使用 pthread_mutex_unlock 對這個 mutex 作解除鎖定的動作。為了讓 ExitCodeFunc 可以控制這個 mutex，所以在應用程式呼叫 cruntimeCreate 函式時，就要將這個決定 CRT 服務生死的 mutex 位址傳入 tmcrt library，tmcrt library 再使用一個全域變數將 mutex 位址儲存起來，這樣 ARM 端負責做指令處理的 RPCServ_serv 要使用 tmcrt library 提供的 ExitCodeFunc 函式時，就可以控制到這個 semaphore 了。

如果是使用 ctrl-c 中斷來強迫 ARM 端應用程式終止，應用程式必須先使用 signal 函式接受一個指定的信號，並指定處理的方法，於是應用程式就使用 signal(SIGINT, ctrlc_signal_handler)在系統中註冊一個處理 ctrl-c 中斷的 ctrlc_signal_handler 函式，當 ctrl-c 中斷事件發生，這個 ctrlc_signal_handler 就會直接使用 pthread_mutex_unlock 去解除應用程式的 mutex 鎖定，至此，在 pthread_mutex_trylock 之後的 cruntimeDestroy(5-1-2-1 節 host_comm library 的第 3 點)與 cruntimeExit 動作就會被執行。



(圖 5.14)應用程式(tmload)啟動終止 CRT 服務的方式

cruntimeExit 如何終止 CRT 服務 thread

有鑑於 5-1-2-1 host_comm library 第 4 點提到的，要讓 CRT 服務 thread 先正常結束該回合的處理後，才對 CRT 服務 thread 做結束與 thread 資源的釋放。

所以 cruntimeExit 內部會透過 host_comm library 的 TM1IF_term 函式，使用 pthread_join 來實現「等待 thread 做完才結束與釋放 thread 資源」，然後為了讓無限迴圈運作的 CRT 服務 thread 可以做完這一回合處理後就終止，所以 TM1IF 模組中另外提供了一個全域變數 gServingStopped 來控制是否進入無限迴圈，一旦設成 True，CRT 服務 thread 就不再進入 while 的下一回合。

另外為了避免被 thread 本身用來控制多個 thread 同時讀取同一個 queue buffer 時，所使用的 semaphore 阻斷介面給 block 住，所以 cruntimExit 內部再藉由 TM1IF 模組的 TM1IF_term 函式使用 tmmanSyncobjSignal 對這個 semaphore 做 unblock，確保目前環境中的 thread 都不會被 block 住，這樣 thread 就不會被困在某個回合裡，host_comm library

就可以使用 pthread_join 成功的終止 CRT 服務的每個 thread 了(圖 5.15)。

```
void TM1IF_term()
{
    int i;
    if (gServingStopped) {return;}
    gServingStopped = True;
    tmmanSyncObjSignal(gEvMsgSync);

    // wait for the server threads to finish
    for (I = 0; I < gNumCreatedServers; i++)
    {   //利用創建時儲存的 thread ID，一一做結束動作
        pthread_join(gMsgServers[i], NULL);
    }
}
```

(圖 5.15)host_comm library 實作 CRT 服務 thread 的終止動作

5-3 第五章結論

藉由硬體上的共享記憶體與中斷機制、軟體上的溝通機制與 user mode library，我們可以設計 DSP 端 CRT utility，使用 user mode library 將 command 送到 ARM 端後，再由 ARM 端的 CRT service module 來做提供 CRT 服務給 DSP 端所下的指令，完成指令實作。DSP 端應用程式於是就可以擁有像在撰寫一般的程式時使用的 standard C library、請求 ARM 端結束 CRT 服務等指令，來做 FILE I/O、結束 ARM 端 CRT 服務的動作。

由於目前 CRT 指令溝通機制還沒有提供能夠動態將 DSP 端影像，複製到 ARM 端應用程式的功能，所以第六章的實驗，我們就在 ARM 端的 CRT service module 做了一些修改，讓外圍應用程式可以得到影像位址，並透過影像位址將 DSP 端影像複製到 ARM 端應用程式後，可以在 ARM 端直接對影像做處理。

第六章 實驗—DSP 端人臉擷取指令設計

這個實驗目標是將 DSP 端人臉偵測完的人臉，傳送給 ARM 端應用程式，讓 ARM 端應用程式可以做即時的人臉處理(如人臉辨識)。基本上有兩種方式，一個是在 DSP 端直接使用 fwrite 將人臉寫到 FIFO 檔案後，ARM 才從檔案系統的 FIFO 中讀出人臉來繼續處理，一種則是想辦法在 ARM 端得到人臉影像位址，然後使用 memcpy 透過 PCI 匯流排將影像直接複製到 ARM 端應用程式中。兩種方式都是要先取得人臉位址才能運作，但是第一種方式不僅需要使用到檔案系統空間，在 ARM 端也多了一個從檔案系統做讀檔的動作，因此，我們採用第二個想法。第二種想法的設計我們依賴第五章的 CRT 指令溝通機制來完成，設計架構請看 6-2。6-1 則先說明如何在 DSP 端人臉偵測應用程式中，增加挖取人臉的動作。

6-1 DSP 端 人臉擷取

人臉偵測演算法主要是運作在 RGB 的錄像上，最後會以平面二維(x,y)記錄人臉每個 pixel 的位置，而每個 pixel 位置都會以一個資料結構儲存起來，這個資料結構裡就儲存了 pixel 的位置(x,y)、此次框人臉的方框長寬、下一個記錄位置的資料結構位址指標，然後以 linked list 將每個資料結構鏈接起來，整個鍊接就是框出人臉的方框位置。

於是每次偵測到人臉，我就利用框人臉的方框長寬創造一個與方框同大小的 buffer，用來存偵測到的人臉，然後把每個資料結構裡記錄的 pixel 位置範圍內錄像的 RGB 值取出，一一排列到 buffer 中(圖 6.1)。

```
typedef struct RECTList
{       int x;
        int y;
        int width;
        int height;
        int label;
        struct RECTList *Next;
}RECTList;
```

RECTList *acurRect;→用來記錄方框 pixel 位置

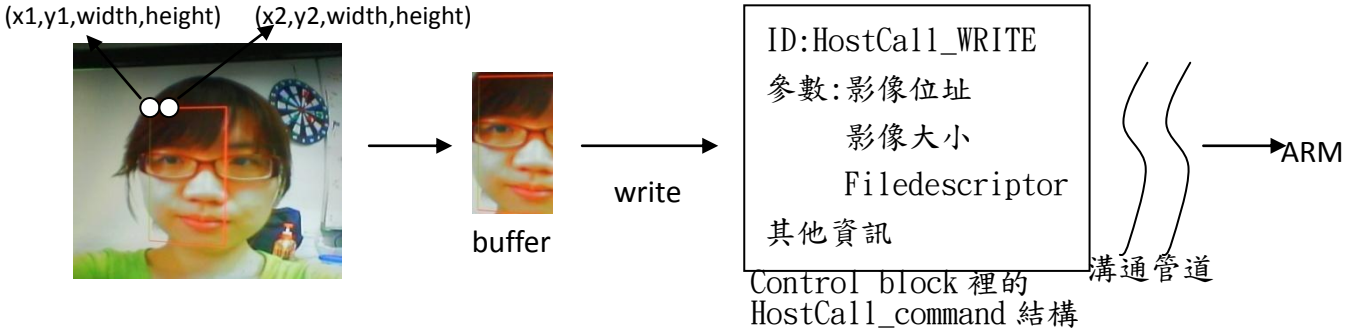
```
/*人臉偵測演算法*/→填入偵測到人臉的方框 pixel 位置 acurRet，一一串接起來

for(acurRect=ahRect;acurRect;acurRect=acurRect->Next)
{
    int x,y,xx,yy;

    filenumber++;
    sprintf(filename,"%d.bmp",filenumber);
    FD=open(filename,O_WRONLY|O_CREAT, 0700);

    /*配置存放人臉的 buffer*/
    facedetect=
    (unsigned char*)calloc(acurRect->height*acurRect->width*3,sizeof(unsigned char));
    printf("**********facedetect**********\n");
```

```
    y_adj=acurRect->y;x_adj=acurRect->x;
    for(y=acurRect->y;y<acurRect->y+acurRect->height;y++){
        for(x=acurRect->x;x<acurRect->x+acurRect->width;x++){
            xx=x-x_adj;yy=y-y_adj;

            /*挖臉*/
            facedetect[yy*acurRect->width*3+xx*3]=timg[y*imageStride*3+x*3];
            facedetect[yy*acurRect->width*3+xx*3+1]=timg[y*imageStride*3+x*3+1];
            facedetect[yy*acurRect->width*3+xx*3+2]=timg[y*imageStride*3+x*3+2];

            /*畫框*/
            if(y==acurRect->y || y==acurRect->y+acurRect->height-1 ||
                        x==acurRect->x || x==acurRect->x+acurRect->width-1)
            {

                timg[y*imageStride*3+x*3]=0;
                timg[y*imageStride*3+x*3+1]=0;
                timg[y*imageStride*3+x*3+2]=255;
            }
        }
    }
    write(FD,facedetect,acurRect->height*acurRect->width*3);//寫檔
    printf("*********free facedetect*********\n");
    free(facedetect);
    close(FD);
}
    deleteRECTNode(&ahRect);
```

（圖 6.1）DSP 端挖臉、畫框應用程式

為了把 buffer 的所在 DSP 端位址及大小傳送給 ARM，ARM 才可以利用這些資訊做記憶體複製的動作，又 DSP 端在使用 CRT utility 做 buffer 的寫檔時，I/O driver components 的 writeFunc 函式就必須將 buffer 在 DSP 端的實體記憶體起始位址做為來源位址，儲存於 HostCall_command 結構中的 write_args 參數結構裡的 buf 指標變數中(表 5.8)，如此一來，ARM 端負責指令處理的 RPCServ_serve 函式(5-2-3 指令處理(圖 5.13))執行時，就可以讀取到存有挖出的人臉 buffer 在 DSP 端的實體記憶體位址，再將它轉成 ARM 端 user space 的虛擬記憶體位址。而後 6-2 節，就是在討論如何將 buffer 在 ARM 端 user space 虛擬記憶體位址傳給 ARM 端應用程式，讓應用程式來做 memcpy 與人臉處理的動作。

不過為了讓寫檔的動作可以一次性的把框到人臉的 buffer 內容送給 ARM，所以在 DSP 端應用程式我就直接使用 POSIX C I/O call 的 write 函式，指定 buffer 位址與影像大小做寫檔的動作。



6-2 ARM 端應用程式動態複製 DSP 端傳送的人臉

6-2-1 設計架構

背景

由第五章 CRT 指令溝通機制,當 DSP 端應用程式使用 write 將 buffer 內容做寫檔時,因為會傳入 buffer 在 DSP 端實體記憶體位址與影像大小、欲寫入檔案的 filedescriptor,所以 I/O driver components 的 writeFunc 就會把這三樣資訊填入 HostCall_command 結構的 write_args 中,然後啟動 hostcall library 進行進階包裝,並由 host_comm library 將含有 HostCall_command 結構內容的 control block 位址存入 packet 的 Argument[0]變數中,利用 tmmanMessageSend 將這個 packet 送出。

因為 ARM 端已經有負責做 CRT 服務的 thread 在等待做接收的動作,等到 packet 一抵達訊息層,訊息層會 signal 儲存於訊息層的 OS 層與應用層阻斷介面,系統自動會讓一個提供 CRT 服務的 thread,利用 tmmanMessageReceive 接收 DSP 端傳送過來的 packet,於是 CRT 服務 thread 再繼續從 packet 中讀取 control block 位址,並將 control block 位址傳入 RPCServ_serve 函式做指令處理。而指令處理時,就會先將 control block 位址轉成 ARM 端控制 control block 使用的虛擬記憶體位址,然後就可以從 control block 的 HostCall_command 結構中讀取到 write_args 結構裡的人臉 buffer 在 DSP 端的實體記憶體位址,而 RPCServ_serve 也會把這個位址轉成 ARM 端控制 buffer 使用的虛擬記憶體位址,再把 file descriptor、影像大小一起傳進 ARM 端 tmcrt library 提供給 host_comm library 的檔案系統 I/O 介面函數去執行真正的 FILE I/O。

目標:動態複製 DSP 端傳送的人臉

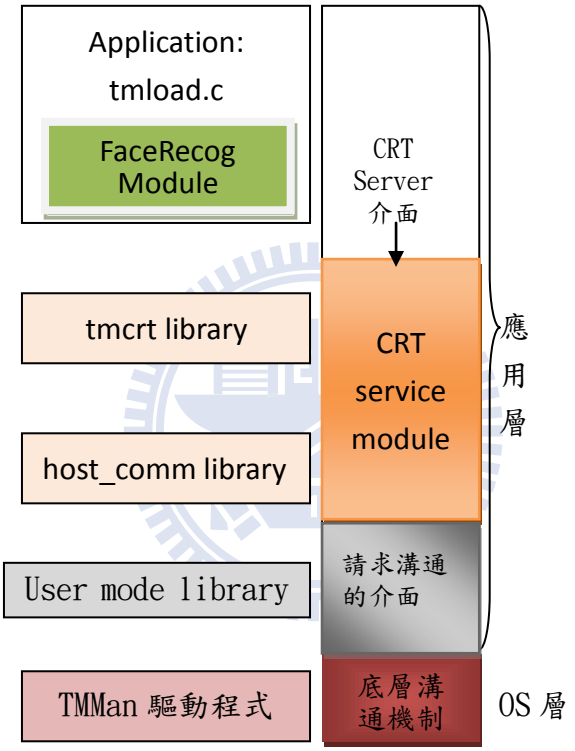
為了讓 DSP 端挖出的人臉可以直接在 ARM 端應用程式被處理,基本想法就是透過上一段背景,從 ARM 端獲得的人臉 buffer 虛擬記憶體位址開始,想辦法讓 ARM 端應用程式可以取得這個虛擬記憶體位址,然後再使用 memcpy,以 ARM 端應用程式另開的 buffer 位址為目的地位址,ARM 端獲得的人臉 buffer 虛擬記憶體位址為起始位址,透過 PCI 匯流排將 DSP 端人臉 buffer 中的影像複製到 ARM 端應用程式新開出的 buffer 中,這樣應用程式就可以利用 ARM 端應用程式新開出的 buffer 內容做運算,像是做 face recognition 的服務。

方法

因為目前環境中不是只有一個提供 CRT 服務的 thread 在運行,所以同一時間 ARM 端可能從不同的 thread 接收到一個以上的 DSP 端人臉 buffer 位址,於是在應用程式中,又另外設計了一個 Face Recognition Module(圖 6.2),這個 Module 包含創建與 CRT 服務的

thread 同樣數目的 Face Recognition thread 的函式(FaceRecog)、做 Face Recognition 內容的函式(FaceRecog_serve)以及用來終止 Face Recognition thread 的函式(FaceRecogExit)。

每當創建 Face Recognition thread 的函式被執行，環境中就會多出與 CRT 服務 thread 相同數目的「作 face recognition 內容」的 thread(簡稱 FR thread)來讀取 CRT 服務的 thread 得到的 ARM 端控制人臉 buffer 的虛擬記憶體位址與影像大小，然後依照得到的影像大小在應用程式的 Face Recognition 內容函式開一塊 buffer，提供影像位址的來源與目的地位址、影像大小給 memcpy 函式，就可以將影像從 DSP 端 RAM 複製到 ARM 端應用程式的 buffer 來，如此一來，不管是需要對人臉做何種處理都可以進行了。



(圖 6.2)ARM 端 Face Recognition Module 與 CRT module 示意圖

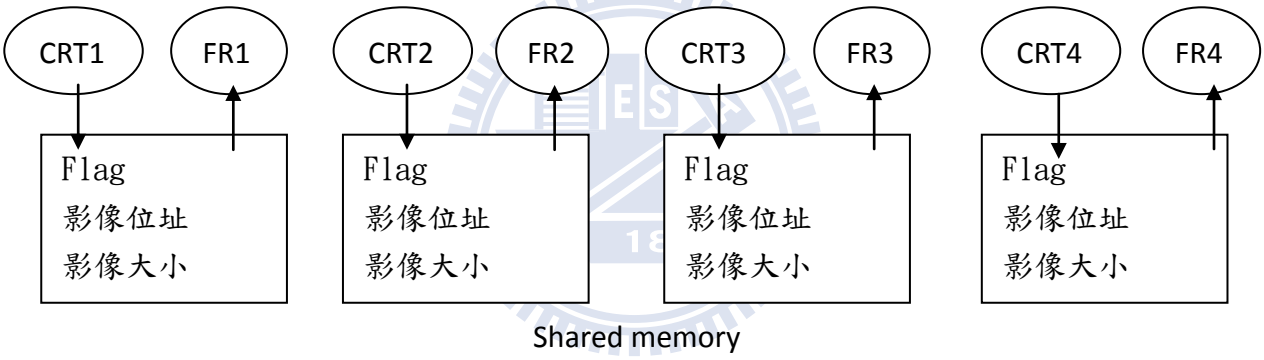
已經擁有與 CRT 服務的 thread 相同數目的 Face Recognition thread 後，我希望每個 Face Recognition thread 都會與 CRT thread 做一對一的對應，這樣的話每次 CRT thread 讀到的影像位址與大小就可以由對應的 Face Recognition thread 繼續在應用程式中做記憶體複製的動作。

但因為 CRT thread 原本是直接透過 RPCServ_serv 就處理完指令操作，並不會回傳影像位址給 CRT thread，我卻希望是由 CRT thread 將人臉 buffer 虛擬記憶體位址與大小告訴我設計的 Face Recognition thread，所以我會讓 RPCServ_serve 函式先回傳人臉 buffer 虛擬記憶體位址與大小給 CRT thread。

然後為了讓 CRT thread 可以從 RPCServ_serve 得到的人臉 buffer 虛擬記憶體位址與

大小告訴 Face Recognition thread，我就提供相同的編號給一個 CRT thread 與一個 Face Recognition thread，然後以 CRT thread 與 FR thread 的編號作為製作共享記憶體身分代號的元素，並在這組 CRT thread 與 FR thread 之間配置一塊以兩個 thread 編號命名的共享記憶體，於是同編號的 CRT thread 與 Face Recognition thread 就會使用到同一塊共享記憶體，而且 CRT thread 會負責寫入影像經轉換後的虛擬記憶體位址與大小到共享記憶體，FR thread 則負責從共享記憶體讀取影像位址與大小。

有了共享記憶體後，開出的 Face Recognition thread 就需要判斷共享記憶體是否有 CRT thread 新寫入的影像位址與大小，這個判斷我是在共享記憶體中提供一個 flag 變數。CRT thread 在寫入新的影像位址與大小前會先判斷 flag 是否為 True，不是的話才會寫入新的資料，然後把 flag 設為 True，表示有新填入資料；而 Face Recognition thread 則利用 while loop 不停檢查 flag 是否為 True，只有 True 的時候才代表有新的資料寫入，然後才做讀取，讀取完後再把 flag 設回 false。下圖假設環境中有 4 個 CRT 與 Face Recognition thread 的情況:



(圖 6.3)CRT thread、FR thread 與共享記憶體的創建

由於 DSP 端應用程式使用 printf 時，也會以 HostCall_WRITE 為 ID 傳送到 ARM 端 CRT Module，所以 CRT thread 也會接收到 printf 字串長度及字串起始位址，然後以 write 的動作來處理，這樣可能會跟我們人臉 buffer 資料混淆，所以我只希望 DSP 端應用程式有做人臉 buffer 的寫檔動作時，才讓 ARM 端的 CRT thread 將接收到的人臉 buffer 位址寫入共享記憶體，然後 FR thread 才會從共享記憶體中讀到影像大小與位址來做人臉複製的動作。有鑒於挖出來的人臉都至少有 100x100x3 byte 的大小、一般 printf 指令要求的寫出 byte 數不會超過 10000，所以我以 20000 byte 為界，當 CRT thread 收到的 DSP 端指令請求，其傳過來的資料 byte 數大於 20000byte 時才會判斷為 DSP 端是做人臉 buffer 的寫檔動作，然後將影像位址、影像大小寫入共享記憶體，並設定共享記憶體中的 flag，Face Recognition thread 這也才會去做讀取與後續記憶體複製的動作，將人臉複製到 ARM 端應用程式中。

6-2-2 實作內容

共享記憶體內容

為了能夠讓 CRT thread 獲得 RPCServ_serve 函式回傳的影像位址與大小，我另外在 RPCServ 模組中定義了一個資料結構 ret_t（圖 6.4），並在 RPCServe_serve 的 function prototype 中多加了 ret_t 結構指標，然後 CRT thread 一開始就宣告一個 ret_t 結構，將這個結構的位址傳入 RPCServ_serve 函式，這樣 RPCServ_serve 函式就可以將人臉 buffer 虛擬記憶體位址與人臉大小寫入 ret_t 結構，CRT thread 就可以從 ret_t 結構讀取內容，再繼續以 ret_t 結構為單位，寫入與相同編號的 Face Recognition thread 使用的共享記憶體中(圖 6.8)，因此，我為每個 thread 開出的共享記憶體大小就是這個資料結構的大小。

```
typedef struct {
    unsigned int ret_address;//影像位址
    unsigned int ret_byte;//影像大小
    Bool state;//讀寫判斷的 flag
}ret_t;
```

(圖 6.4)ret_t 結構

```
Bool RPCServ_serve (
    Pointer pRawCmd,
    int nodeNumber,
    ret_t *ret_value   //另外定義用來儲存 ARM 端虛擬記憶體影像位址與大小的結構
){
/* ----------------------------------------*/
      case HostCall_WRITE:
         wcommand->parameters.write_args.retval =
            convert_host_long (
               parm_write (
                  convert_tcs_long (command->parameters.write_args.fildes),
                  IN_BUFFER (command->parameters.write_args.buf),
                  convert_tcs_long (command->parameters.write_args.nbyte)));

    //.......yalan........//
    ret_value->ret_address=IN_BUFFER (command->parameters.write_args.buf);
    ret_value->ret_byte=convert_tcs_long(command->parameters.write_args.nbyte);

         PCI_STABILISE (&command->code, &saved);
         wcommand->returned_errno      = convert_host_errno (last_io_errno ());
         wcommand->notification_status = convert_host_long (HostCall_DONE);
      break;
/*-----------------------------------------------*/
}
```

（圖 6.5）RPCServ_serve 函式

共享記憶體創建與使用

而共享記憶體的創建，我則利用 shmget 和 shmat 這兩個 api，由 shmget 透過一個指定給它的身分代號(key)以及所需的記憶體大小(sizeof(ret_t))，在 kernel space 內開出一塊空間，並傳回一個共享記憶體 id。從此之後，凡是再呼叫 shmget 並使用同一個 key 的人，就不會再開出新的空間，而是對應到之前開出的空間。

但是，光只是在 kernel space 內開出記憶體空間，process 是無法使用它的，所以還要用 shmat 這個函式，把剛剛開出的記憶體空間，回傳一個 user space 虛擬記憶體位址給 process，如此一來才可以在應用層對該記憶體空間進行操作，而 shmat 所需參數有：透過 shmget 傳回的共享記憶體 id,指定要把該記憶體空間掛到 user space 的哪個位址(設為零可由系統幫你指定)，還有一個設定這塊空間使用權限的 flag。要注意的是，如果自行指定要掛到哪個位址，flag 必須有 SHM_RND 的屬性，有這個屬性系統才會幫忙把我們指定的位址做 page 的對齊。

在開啟 Face Recognition 服務前，使用者必須先依據 CRT 服務開出的 thread 個數去設定應用程序中我另外定義出的 NROF_FaceRecog 變數，這樣啟動 FR 服務的函式執行時，才會去開出與 CRT thread 個數相同的 FR thread(FaceRecog_thread)於應用層中。然後應用程式會先使用 cruntimeInit 函式啟用 CRT 服務，接著才使用啟動 FR thread 的函式來創建 FR thread。

因為每個 FR thread 的編號要與 CRT 的相同，這樣我們才可以利用相同的編號去創建共享記憶體。所以在使用 pthread_create 創建 CRT thread 時(圖 6.6)，就以一個初始為 0 的全域變數 gNumCreatedServers 當作創建出的 CRTthread 實體函式所需參數，然後每創建一次就加一，因此每個被創建出的 CRT thread 實體函式裡頭就有從 0~NROF_SERVERS-1 的代號。相同的，在創建 FR thread 實體函式時(圖 6.7)，我也利用 pthread_create 傳入初始為 0 的 gNumberFaceRecog 變數，當成 FR thread 實體函式(FaceRecog_thread)實體函式需要傳入的參數，如此一來，環境中就有 NROF_FaceRecog(或 NROF_SERVERS)組互相對應的 CRT thread 與 FR thread。

```
#define NROF_SERVERS  4
TM1IF_Served_Status TM1IF_start_serving(void)
{    int i, status;
    gNumCreatedServers = 0;
    memset(gMsgServers, sizeof(gMsgServers), 0);
    for (i = 0; i < NROF_SERVERS; i++)
    {   pthread_t thread;

        status = pthread_create(&thread, NULL, (void *) &tmlif_serve,
            (Pointer) gNumCreatedServers);
                                          tmlif_serve(id),id 為 tmlif_serve 參數
        if (status == 0)                  此處 id= gNumCreatedServers
            gMsgServers[gNumCreatedServers++] = thread;
        else
            continue; }
.....}
```

(圖 6.6)TM1IF 模組中啟動 CRT 服務 thread 的函式

```
#define NROF_FaceRecog 4 //same as NROF_SERVERS in TM1IF_pc.c
void FaceRecog(void){
   int i, status;

    gNumberFaceRecog = 0;
    memset(gFaceRecog, sizeof(gFaceRecog), 0);
    tmmanSyncObjCreate(&gEvFaceSync);

    for (i = 0; i < NROF_FaceRecog; i++)
    {
        pthread_t FaceRecog_thread;

        status = pthread_create(&FaceRecog_thread, NULL, (void *)
                                &FaceRecog_serve,(Pointer) gNumberFaceRecog);

        if (status == 0){
            gFaceRecog[gNumberFaceRecog++] = FaceRecog_thread;
            printf("Number %d FaceRecog_serve created!\n",gNumberFaceRecog-1);
    }
        else
            continue;
    }
}
```

(圖 6.7)應用程式中 FR 模組啟動 FR thread 的函式

所以每個 CRT thread 就以開出 thread 的順序編號(1~NROF_SERVERS)當成是創建共享記憶體所需的 key(如果是 0 系統會自動分配，所以 key 值一定不能為 0，如果 thread 編號是 0，那一定要讓大家的編號都加上一些值)，然後使用 shmget 創建一塊 ret_t 大小、可讀寫的共享記憶體:`shmid = shmget(key,sizeof(ret_t), IPC_CREAT | 0666)`，接著再利用回傳的共享記憶體 id 做 `shmret=(ret_t*)shmat(shmid, NULL, 0)`，這個 shmret 就是在 CRT thread 的 user space 中看到的共享記憶體虛擬記憶體起始位址，利用這個位址，CRT thread 可以將 RPCServ_serve 回傳得到的 ret_t 結構內容，先判斷讀到的影像大小是否大於 20000 bytes，是的話才在共享記憶體寫入影像的虛擬記憶體位址與大小，並將

state 設為 True(圖 6.8)。

```
static void tmlif_serve(int id)
{
    int i;
    TMStatus tmStatus;

    FILE *fptr_test;
    ret_t ret_struct;//傳入 RPCServ_serve 的結構指標

    ret_t *shmret;//指到共享記憶體的起始位址
    int shmid;//共享記憶體 id
     //…..……yalan…..……//
    key_t key= id+1;//創建共享記憶體需要使用的身分代號

    if ((shmid = shmget(key,sizeof(ret_t), IPC_CREAT | 0666)) < 0) {
        perror( "shmget" );
        exit(1);    }
       /** Now we attach the segment to our data space.**/
    if ((shmret=(ret_t*)shmat(shmid, NULL, 0)) == (char *) -1) {
        perror( "shmat" );exit(1);
    }
    Shmret->state=False;//確保 FR thread 還不能讀，因為 CRT thread 還沒收到資料
   //……//

    while (!gServingStopped)
    {   tmmanPacket packet;
        (void) tmmanSyncObjWait(gEvMsgSync);
        for (i = 0; !gServingStopped && (i < gNumDsps); i++)
        {
            if (gNodeData[i].valid) {
          tmStatus = tmmanMessageReceive(gNodeData[i].hostcall_channel,&packet);
          if (tmStatus == statusSuccess)
          {
                Pointer raw_command;
                HostCall_command *command;

                raw_command = (Pointer) packet.Argument[0];
                command->notification_status = HostCall_BUSY;

                ret_struct.ret_address=0;
                ret_struct.ret_byte=0;
            if ( RPCServ_serve(raw_command, i, &ret_struct) && gNodeData[i].valid){
             //……yalan……//
           /** Now put some things into the memory for the other process to read.**/
                if(ret_struct.ret_byte>=20000 && shmret->state!=True){
                shmret->ret_address=ret_struct.ret_address;
                shmret->ret_byte=ret_struct.ret_byte;
                shmret->state=True;
                tmmanMessageSend(gNodeData[i].hostcall_channel, &packet);
                }
            }
        }
     }//end of for
    }//end of while

    (void) tmmanSyncObjSignal(gEvMsgSync);
}
```

透過共享記憶體起始位址 shmret，依序將影像大小、位址以及 state 寫入共享記憶體

(圖 6.8)CRT thread 函式實體

同樣的，每個開出一個 FR thread 時(圖 6.9)，便使用與 CRT thread 同樣的編號當成使用共享記憶體的 key，因為 FR thread 使用共享記憶體的 key 與 CRT thread 創建共享記憶體時給的 key 是一樣的，所以當 FR thread 要使用共享記憶體前、利用 shmget 做取得可讀寫的共享記憶體動作時，就只是對 CRT thread 開出的共享記憶體做讀取的動作，不需要再另外創建共享記憶體，因此使用

`shmid = shmget(key,sizeof(ret_t),0666)`得到 shmid 後，同樣使用 shmat 就可以得到 FR thread 程序執行中看到的、具有同樣編號的 CRT thread 已開出的共享記憶體虛擬記憶體起始位址，然後就可以從共享記憶體中**讀取**影像大小、位址、state。

```
static void FaceRecog_serve(int index){
    key_t key= index+1;//can't use 0
    int shmid;
    ret_t *shmret;
    ret_t ret_struct;
    unsigned char *buffer;

    /* Locate the segment.*/
    if ((shmid = shmget(key,sizeof(ret_t),0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    if ((shmret=(ret_t*)shmat(shmid, NULL, 0)) == (char *) -1)
    {perror("shmat");exit(1);}
    /*  人臉記憶體複製或繼續對人臉做處理的動作 (下一段程式 8) */
}
```

(圖 6.9) 做人臉複製與處理的 Face Recognition thread 實體函式

而 FR thread 實體中，我使用了一個 gFaceRecogStopped 變數來決定是否還要繼續做人臉複製的動作，如果要做，則必須判斷該 thread 使用的共享記憶體中，ret_t 結構裡的 state 變數是否為 True，是的話才可以利用 shmat 得到的共享記憶體起始位址，讀取影像位址與影像大小，不然的話就會一直在 loop 裡空轉。只要有做讀取影像位址與大小的動作，就會開出一個影像大小為 size 的 buffer，然後利用得到的影像位址與大小作 memcpy 的動作，將偵測到的人臉複製到 buffer 中，複製完後就可以繼續對 buffer 中的人臉做辨識或是寫檔等運作。

```
while(!gFaceRecogStopped){
    while(shmret->state!=True){
        if(gFaceRecogStopped) return;
    };

    ret_struct.ret_address=shmret->ret_address;
    ret_struct.ret_byte=shmret->ret_byte;
    shmret->state=False;

    buffer=(unsigned char*)calloc(1,ret_struct.ret_byte);
    memcpy(buffer,ret_struct.ret_address,ret_struct.ret_byte);

    /* 對 buffer 中的人臉做其他處理(寫檔或是人臉辨識)*/
    free(buffer);
}
```

(圖 6.10)終止 FR thread 時強迫 return

Face Recognition thread 的終止

另外，FR Module 中，還有一個用來終止所有 FR thread 的函式。因為自從應用程式啟用 CRT 與 FRthread 以後，會使用 5-2-5 節使用的 mutex 機制，將最上層應用程式 block 在進入終止動作前，所以才能一直替 DSP 端做 CRT 與 FR 的服務。但是當 DSP 端呼叫 tmMain_Exit() /_psos_exit()，終止 CRT 服務的請求會透過 CRT 機制傳遞給 ARM 端，ARM 接收到請求後就會去 unlock 這個 mutex，所以便會開始做終止 CRT thread 的動作，既然 CRT 服務終止，那麼與其對應的 FR thread 也該終止，於是我就讓應用程式在終止完 CRT thread 後，另外增加 face recognition 的終止運作:

```
void FaceRecogExit(void){
    int i;
    if(gFaceRecogStopped==True) return;

    gFaceRecogStopped=True;

    for (i = 0; i < gNumberFaceRecog; i++)
    {
        pthread_join(gFaceRecog[i], NULL);
    }

   gNumberFaceRecog = 0;
}
```

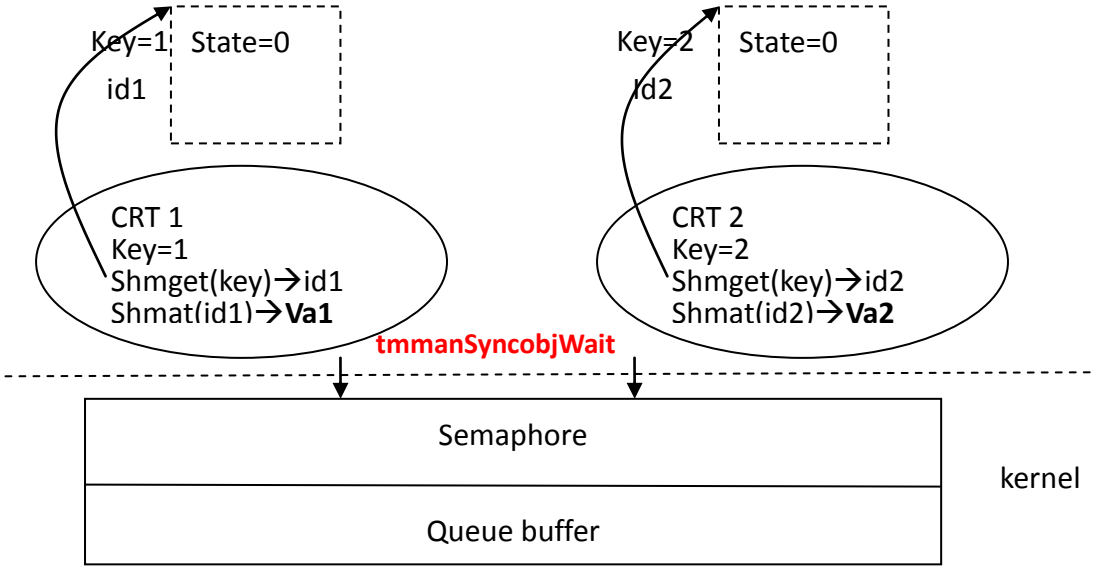
(圖 6.11)終止 FR thread 的函式

為了讓還在運作中的 thread 可以先處理完一輪後才終止，避免中間運作被強制中斷，所以我使用 pthread_join 函式來做終止 thread 的動作，並且在呼叫 pthread_join 前先把用來決定是否還要繼續做人臉複製動作的 gFaceRecogStopped 變數設為 True，表示下一輪就停止從共享記憶體讀資料與停止做人臉複製的動作，但是，如果該 thread 本來對應的共享記憶體中的 state 在 DSP 端請求結束服務前，就已經處於 False 的話，因為 DSP 已經結束了，ARM 端 CRT thread 也結束了，不可能會有任何的程式去把 state 設為 True，

就會一直在無限迴圈中等著 state 變為 true。這樣的話即使終止 FR thread 的函式將 gFaceRecogStopped 變數設為 True，想讓下一輪的處理動作終止，但因為 thread 已經在 loop 中不停的等待，根本無法進入下一輪的判斷，所以 pthread_join 無法終止該 thread。為了解決這個問題，我就在 FR thread 判斷共享記憶體 state 的 loop 裡頭多加一個對 gFaceRecogStopped 變數的判斷，如果發現 gFaceRecogStopped 變數已經變為 True，就強迫 return 掉這個 thread 的運作(圖 6.10 紫色粗體部分)。

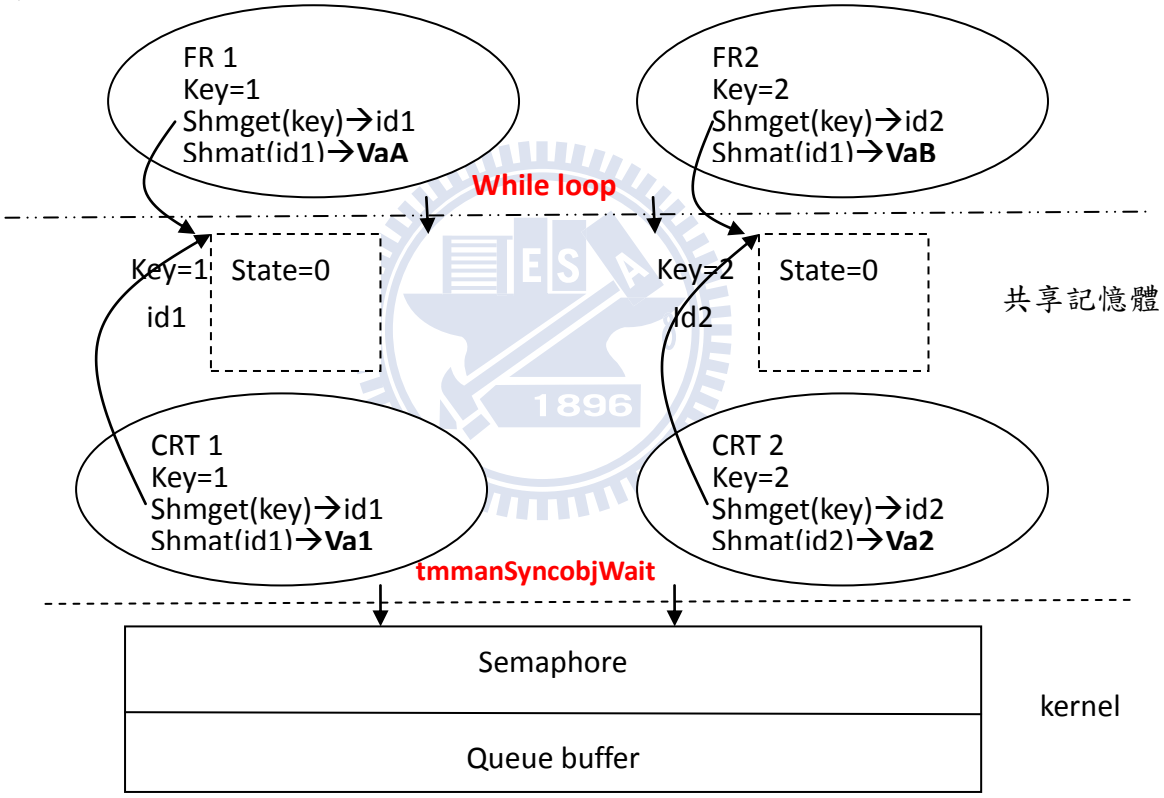
6-2-3 ARM 端應用程式加上人臉複製動作後的執行程序

在應用程式裡，首先初始化一個未來會用來阻斷結束 CRT 與 FR thread 的運作的 mutex，，接著在應用程式使用 cruntimeInit 函式創建阻斷介面並啟動 CRT 服務的 thread，CRT thread 被創建時會傳入該 thread 的編號給 thread 實體函式，並開始 thread 實體函式的執行，thread 實體函式首先會利用 thread 編號(不為 0)當成創建共享記憶體所需的 key，然後使用 shmget 函式，創建一塊 ret_t 結構大小、可讀寫的共享記憶體空間，創建完後此函式會回傳共享記憶體的 id，接下來，為了要讓 thread 可以在虛擬記憶體空間使用這塊共享記憶體，所以 thread 實體函式還要使用 shmat 函式，在目前運作的程序中由系統分配一塊虛擬記憶體空間給擁有這個共享記憶體 id 的共享記憶體空間，然後回傳一個共享記憶體在虛擬記憶體空間的起始位址，這樣 thread 實體函式就可以在 user space 使用這個共享記憶體，得到共享記憶體虛擬記憶體後，一開始就會把共享記憶體裡的 state 設成 False。然後就 thread 實體函式就會使用 tmmanSyncobjWait 函式試著去取得 semaphore，但是因為 semaphore 初始為 0，所以 thread 程序就被 block 住。



(圖 6.12)CRT thread 被創建時在環境中的運作與共享記憶體使用狀況

接著 ARM 端應用程式就會呼叫 FaceRecong 函式，利用 pthread_create 來創建與 CRT 服務 thread 相同數量的 Face Recognition thread，並傳入 thread 編號給 FR thread 的實體運作函式--FaceRecog_serve，thread 實體函式首先會利用 thread 編號(不為 0)當成使用共享記憶體所需的 key，然後使用 shmget 函式去找到 CRT thread 之前用相同的 key 創建出來的共享記憶體空間，此函式就會回傳與 CRT thread 創建時相同的共享記憶體 id 給 FR thread，接下來，FR thread 實體函式便會使用 shmat 函式，在目前運作的程序中由系統分配一塊虛擬記憶體空間給擁有這個共享記憶體 id 的共享記憶體空間，然後回傳一個共享記憶體在虛擬記憶體空間的起始位址，然後 FR thread 會以 loop 的形式反覆確認共享記憶體裡的 state 狀態，除非 state 有被更改為 true，不然就會一直執行 loop，直到 state 被設成 True，才會繼續從共享記憶體中讀取影像位址與大小，然後繼續完成複製影像的動作。



(圖 6.13)FR thread 被創建時在環境中的運作與共享記憶體使用狀況

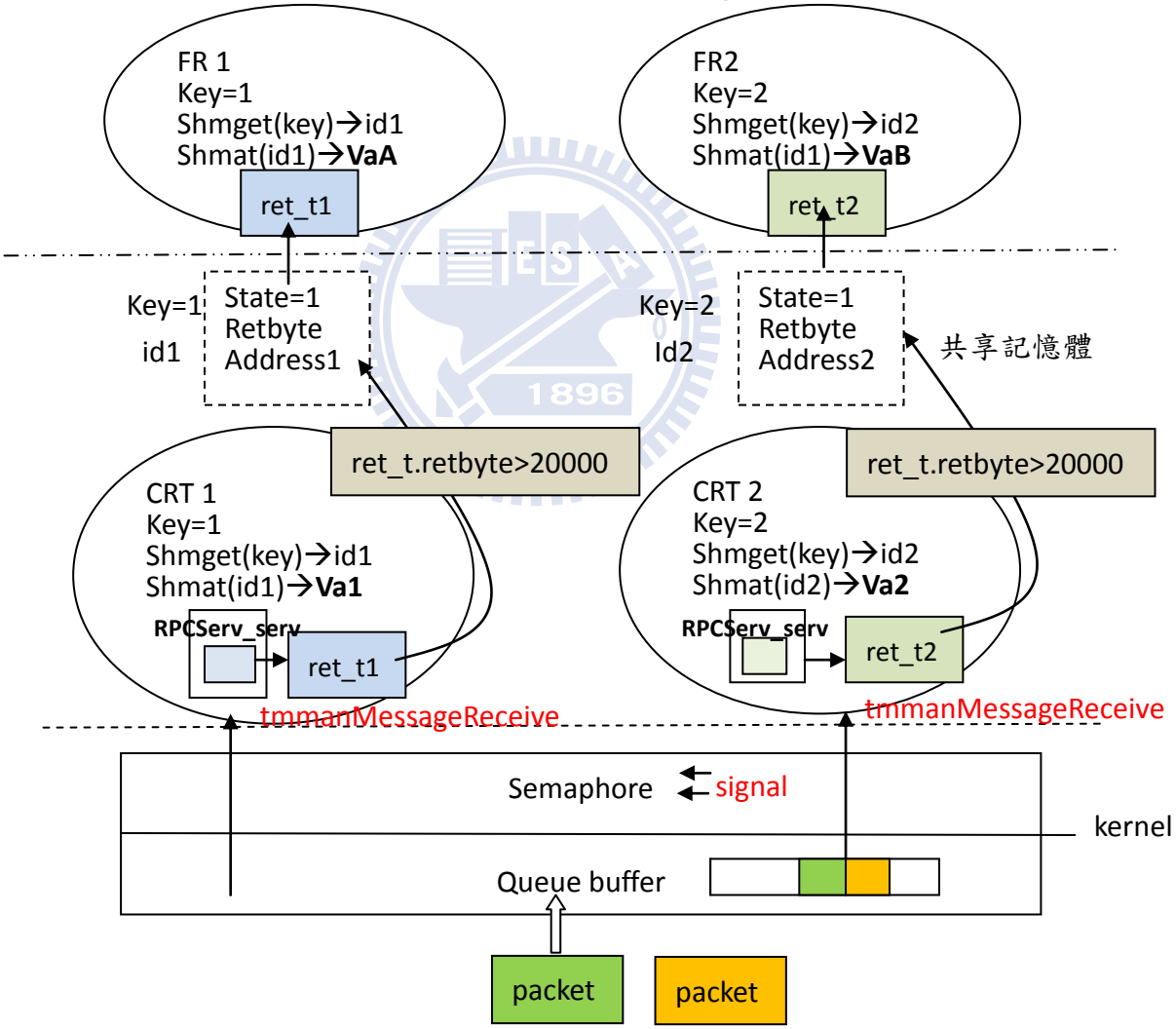
接著應用程式會先利用 pthread_mutex_lock 鎖定 mutex，再使用 cruntimeCreate 創建 CRT 溝通管道、儲存處理指令時所需環境資訊給 RPCServ 模組，然後再使用 pthread_mutex_trylock 試著再去所定 mutex，但是因為 mutex 之前已經被鎖定了，所以應用程式的主程式就被 block 住，無法繼續作結束 CRT 服務與 FR 的動作。

而後，DSP 端每次挖出人臉就存放於另外配置出的 buffer，然後 open 一個檔案，接著使用 write 檔案系統呼叫，透過 DSP 端 CRT utility 將人臉 buffer 在 DSP 端的實體位

址、人臉大小存入 HostCall_command 結構，最後他把結構內容複製到另一塊 controlblock 空間，利用 tmmanMessageSend 將存有 control block 位址的 packet 傳送給 ARM 端。

ARM 端 OS 層完成底層溝通，將 packet 塞入 queue buffer 後，就會請 kernel 去釋放 Message 層的 semaphore，以啟動 CRT thread 繼續執行從 queue buffer 讀取 packet 的動作，然後，將 contrl block 位址交給 thread 實體中的 RPCServ_serve 函式，RPCServ_serve 函式將 control block 位址轉換成 ARM 端目前程序中 user space 的虛擬記憶體後，就根據 control block 內容進行 write 指令操作，接著會回傳 control block 中轉換成虛擬記憶體位址的資料位址與資料大小給 CRT thread。

CRT thread 函式於是會先判斷資料大小是否大於 20000 bytes 來判斷這筆資料是否為人臉，如果是的話就把位址與大小都寫入共享記憶體，並把共享記憶體的 state 設成 True。設完共享記憶體後 CRT thread 就會繼續使用 tmmanMessageSend 回覆 DSP 端的請求。

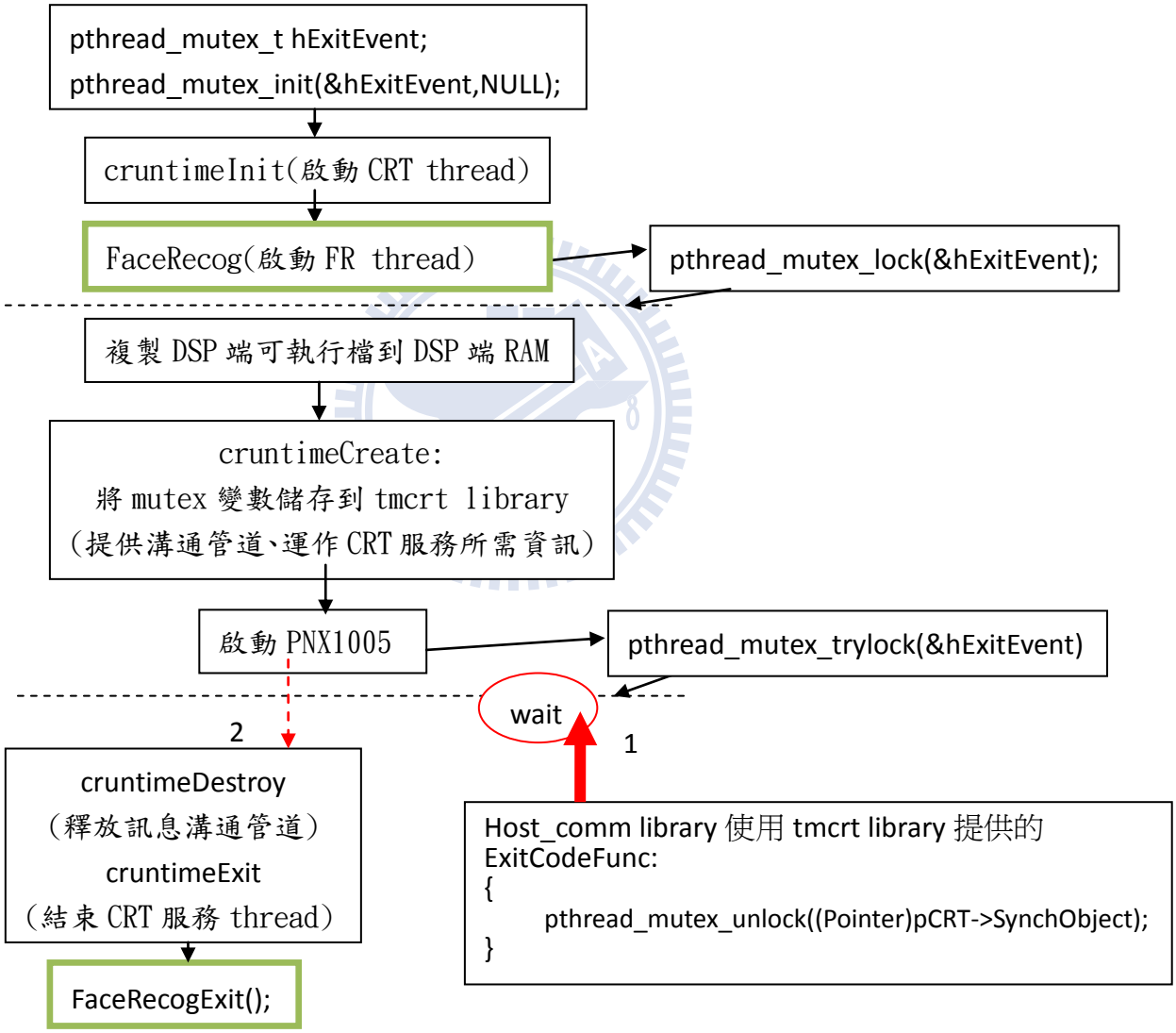


(圖 6.14)啟動 DSP 端後，CRT thread 收到 packet 與 FR thread 的訊息交換狀況

因為 FR thread 的實體函式 FaceRecog_serve 會一直利用 loop 去檢查共享記憶體的 state，而 CRT thread 已經把 state 設成 True，表示有把資料寫入共享記憶體，所以 FR

thread 就從共享記憶體中讀取影像位址與大小，然後把共享記憶體的 state 設為 False。接著利用影像位址、大小，我們在 FR thread 的實體函式中就配置另一塊 buffer，以這個 buffer 為目的地，影像位址指到的影像空間為來源，使用 memcpy 將影像透過 PCI 匯流排將影像從 DSP 端複製到 FR thread 實體函式配置出的 buffer 中。之後實體函式就可以對這塊 buffer 進行任何處理。每次處理完 buffer 內容後，FR thread 就會進入下一個回合，繼續藉由 loop 判斷是否可以從共享記憶體讀取資料，不然就一直等待。

之後當 DSP 端請求 ARM 端結束 CRT 服務時，就會把之前被鎖定的 mutex 給解除鎖定，主程式被喚醒後就會繼續執行 cruntimeExit 與 FaceRecogExit 函式來結束 CRT 服務與 FR 處理。



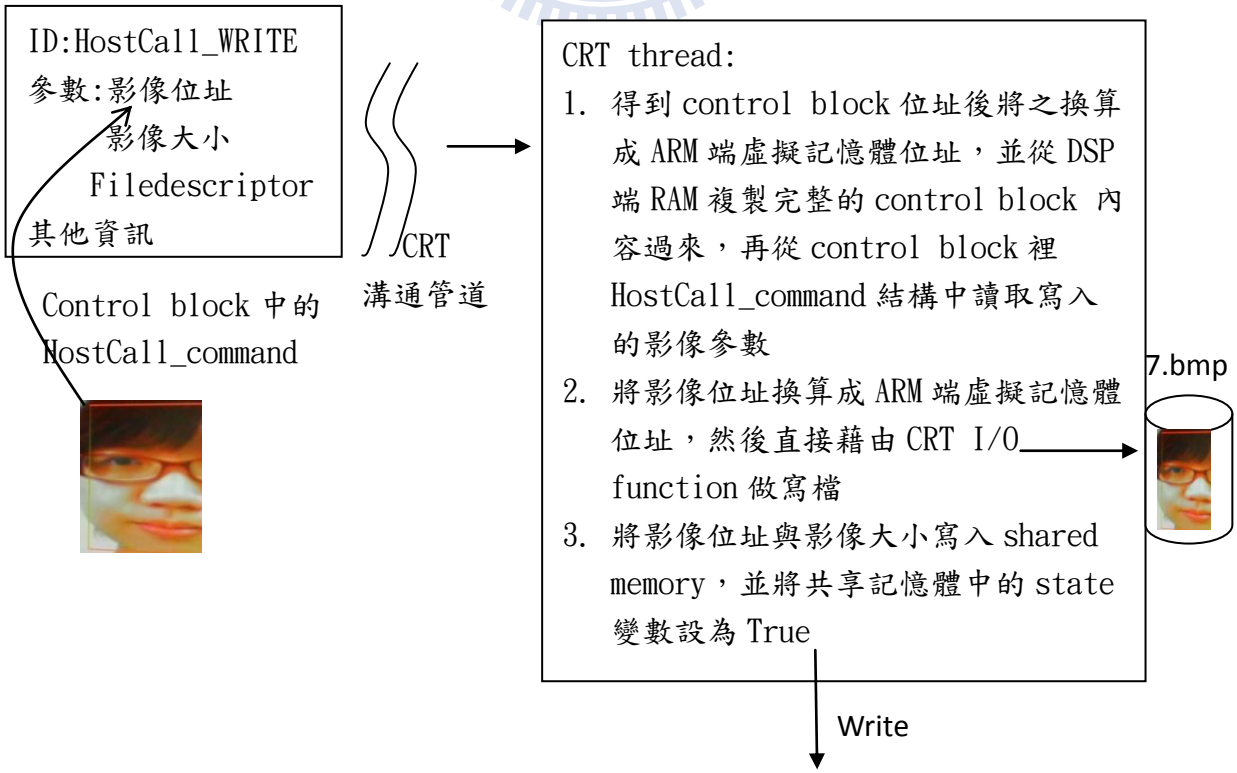
(圖 6.15)FR thread 的結束

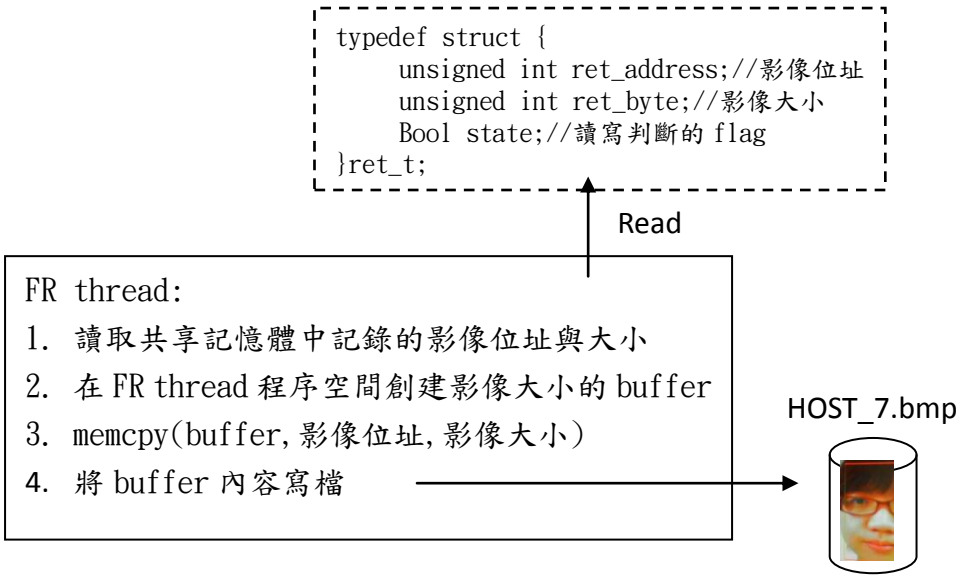
6-3 實驗結果

實驗環境設定:

ARM OS	Linux/ kernel 2.6.22.18
DSP OS	pSos
CRT thread/ FR thread 個數	4
gFaceRecogStopped	初始為 False

為了證明我開出的 FR thread 做 memcpy 複製到的影像的確與 DSP 端挖出的人臉是一樣的二進位內容,所以我在 DSP 端每次都為新偵測到的人臉開一個檔案,然後使用 write,直接透過 CRT 服務路徑作寫檔的動作;另外,在 FR thread 中,因為 ARM 端 CRT 在處理 write 的同時,會知道寫入的影像大小已經大於 20000 byte,所以 CRT thread 會把轉換過後的影像位址與影像大小填入自己的共享記憶體,然後由 FR thread 從共享記憶體中讀取資料,開出影像大小的 buffer 然後做 memcpy 的動作,而且每次複製一個 buffer,就開一次檔案,將 buffer 裡的內容寫到檔案中。最後,我藉由確認 DSP 端直接寫入的檔案與 FR thread 寫入的檔案是否相同來判斷,FR thread 是否有正確抓到 DSP 端人臉的影像位址,也證實 CRT 指令溝通機制的確可以取得影像位址,而且可以透過 memcpy 將資料從 DSP 端 RAM 透過 PCI 匯流排將影像傳到 ARM 端的 RAM。





（圖 6.16）驗證的運作流程

實驗數據:

DSP 端做 write(FD, facedetect, acurRect->height*acurRect->width*3)後，CRT 端從訊息層收到 DSP 端傳出 control block 位址以後的處理過程。上面 write 參數：FD 為 7.bmp 的 file descriptor，facedetect 為人臉所在實體記憶體位址，acurRect->height*acurRect->width*3 為人臉影像大小。

tmman:queue:queueCreate:buffer[c7e81800](KernelAddress)// Queue buffer 起始位址

ninfo->address_shift=8017a000 //ARM端虛擬記憶體位址減掉DSP端實體記憶體得到的位址差

從 queue buffer 起始位址加上 57 個 packet 的 offset，即為目前 writeindex(57)指到的 queue buffer 空間(c7e81800+57*24= c7e81d58)，而插入的 packet 內容為 c2713880

tmman:messageChannelCallback:queueInsert:DT[c2713880],DT(From channel KernelAddress)[c6db3ba4]

tmman:ARM:queueInsert:writeIndex=57,from Item[c6db3ba4](channel) to ItemQueue[c7e81d58]:DT[c2713880]

1. 從 ReadIndex 指到的 queue buffer 空間讀 packet，發現 ReadIndex 指到的空間與之前 WriteIndex 指到的空間是一樣的，packet 內容也是一樣的，確認 packet 讀取無誤
2. 將此 packet 從 queue buffer 中複製到 message 模組的運作空間(c75f2c48)

pnx1005:trimedia_ioctl() i=c7663e60 f=c7d2e4a0 MessageReceive

tmman:queue:queueDelete:ReadIndex=57,itemqueue[c7e81d58](KernelAddress),Messagebuffer[c75f2c48]

tmman:ARM:messageReceive(After queueDelete):DT(message buffer kernel address)[c75f2c48]:DT[c2713880]

將 packet 從 message 模組運作空間轉到 tmmanMessageReceive 函式運作空間([59d98d94])後，tmmanMessageReceive 函式會把 packet 再複製到應用程式傳下來的指標所指到的空間

DT[c2713880],DTaddress[59d98d94](usermode buffer),DTaddress[59d98dd4](application)

所以應用程式從 tmmanMessageReceive 得到 packet 後，就取出 packet 內容，因為和上面 packet 的內容一樣，所以証實 packet 被塞入 queue，然後應用程式再由 queue 讀出 packet 的過程是無誤的

thread:packetAddress[59d98dd4],packet.Argument[0](raw_command)=c2713880

在RPCServ_serve函式中換算位址，位址差8017a000+packet中儲存的control block位址c2713880(取八位)

RPCServ:raw_command=c2713880

ninfo->address_shift=8017a000,tm_address=c2713880

virtual command address=4288d880

RPCServ_serve函式從control block中取得write指令傳入的影像位址與大小，同樣作轉址的動作，然後為DSP端先作寫檔(7.bmp)的動作，最後將轉址後的位址與大小填入ret_t，然後將ret_t內容回傳給CRT thread

command->parameters.write_args.buf=c1f28f6c

ninfo->address_shift=8017a000,tm_address=c1f28f6c

IN_BUFFER (command->parameters.write_args.buf)=420a2f6c

ret_value->ret_address=420a2f6c

ret_value->ret_byte=30000

CRT thread收到ret_t後，檢查資料大小是否大於20000 byte，是的話就將資料且入共享記憶體中，並把共享記憶體state設為1，此處寫入共享記憶體的thread的編號為2

thread take face!!

tm1if id=2,**shmret->ret_address=420a2f6c,shmret->ret_byte=30000,shmret->state=1**

與CRT thread編號相同的FR thread(編號為2)因為發現共享記憶體的state變為1，於是從共享記憶體讀出CRT thread寫入的影像位址與大小，然後FR thread就可以透過得到的位址，從DSP端複製影像到ARM端FR thread中，並開出HOST_7.bmp檔對它作寫入的動作

FaceRecog index=2,**shmret->ret_address=420a2f6c,shmret->ret_byte=30000**,shmret->state=1

tmload:filenumber=7

```
00000000h: 33 58 68 36 5B 6B 34 59 69 34 58 6A 35 59 6B 34
00000010h: 58 6A 33 57 69 33 57 69 35 59 6B 33 57 69 31 55
00000020h: 67 30 56 67 2F 57 68 2F 57 68 2F 57 68 30 57 6B
00000030h: 33 58 6C 34 59 6D 33 58 6C 33 58 6C 36 59 6E 36
00000040h: 59 6E 37 5A 6F 37 5A 6F 38 5B 70 3B 5E 74 3D 60
00000050h: 76 3D 61 77 3E 65 79 3C 65 79 3C 65 79 41 6A 7E
00000060h: 43 6C 80 43 6C 80 47 70 84 47 70 84 47 70 84 4A
00000070h: 73 87 4C 75 89 4F 75 8B 50 74 8A 52 76 8C 55 78
00000080h: 8D 55 79 8B 58 7D 8D 5A 7F 8F 5D 80 8F 5F 80 8F
00000090h: 64 83 92 66 85 94 65 84 93 65 84 93 68 86 93 6A
000000a0h: 88 95 6B 8A 95 6C 8B 96 6D 8C 95 6E 8E 95 71 8F
000000b0h: 95 73 91 97 76 93 96 77 95 96 7A 95 97 7C 96 98
000000c0h: 80 98 98 80 99 97 83 9A 97 86 9D 9A 8A 9F 9C 8A
000000d0h: 9F 9C 8D A0 9D 8D A0 9D 8D A0 9D 8F A0 9E 92 A1
000000e0h: 9F 91 A0 9E 92 A1 9F 91 A0 9E 92 A1 9F 91 A2 A0
000000f0h: 91 A2 A0 8C A1 9E 8A A1 9E 84 9F 9D 81 A0 9D 7E
```

7.bmp & HOST_7.bmp 的 binary 內容均相同

第七章 結論

為了讓兩個各自負責自己功能的CPU可以同時運作來完成一個應用，我們需要設計一個溝通管道讓兩個CPU在必要的時候做資料傳輸與訊息溝通，於是我們會建立一個雙向的client-server的架構，表示ARM與DSP可具備傳送端做指令請求或接收端等待接收的特性，而硬體上則使用了共享記憶體與中斷機制作為訊息傳遞的媒介，軟體上則為傳送端與接收端分為應用層與OS層，在接收端的應用層與OS層則使用 semaphore 來做阻斷機制以完成等待的動作，而OS層又分為硬體抽象層、虛擬中斷層、通到層/訊息層、同步層，每層都會使用到共享記憶體的對應區塊做設定，來完成不同事件與應用功能的區分(第四章)。

而這樣的溝通架構我們則設計於ARM端的驅動程式中，並提供功能函式庫作為驅動程式與應用層的介面，讓應用層可以直接透過功能函式完成溝通管道的建立、訊息的傳遞與接收。於是在兩端的應用層就設計了一個CRT服務，讓DSP端可以透過RPC的模式，將指令內容的位址傳送到ARM端，ARM接收到這個位址後便可以透過PCI 匯流排將指令內容從DSP端複製過來ARM端，藉由這些內容ARM再來決定要做怎樣的I/O處理(第五章)。

但因為目前ARM端CRT只提供63個特定的I/O服務給DSP(Appendix B)，並不包含從DSP端直接把影像複製到ARM端user space的動作，為了在ARM端user space可以直接得到影像並進行即時處理，所以我利用與CRT相同的溝通管道，DSP端同樣利用write指令將影像大小與位址資訊傳送到ARM端，而ARM端CRT實作最後的I/O服務前我們另外將影像位址與大小擷取出來，然後另外在上層應用程式設計一個FR 服務的thread與interprocess溝通所用到的共享記憶體，CRT thread會先將得到的影像位址與大小寫到共享記憶體，而後再由FR thread來接收並做記憶體複製的動作(第六章)。

藉由了解DSP端操作ARM端檔案系統的機制，我們可以修改或新增ARM端檔案系統操作介面，對DSP端不斷傳入的影像做好檔案儲存管理 。

參考文獻

[1] PNX100X Series Data Book，Volume 1 of 1，1.2版，NXP恩智浦半導體公司， July/2009

[2] TriMedia Manager and Embedded Linux for NDK6.2，NXP恩智浦半導體公司，Nov/2009

[2] C Runtime:tmcrt for NDK4.11，NXP 恩智浦半導體公司，May/2004

[3] Trimedia Manager User Manual for TCS5.2，NXP 恩智浦半導體公司, Oct/2008

[4] System Utilities for TCS5.2，Volume 7，Oct/2008

[5] Software Architecture，Book3-PartA，Philips Semiconductors，Oct/1999

[6] User Guide 16 Channel Video Demonstrator，1.02版，Nov/2009

[7] The Linux Kernel， http://tldp.org/LDP/tlk/tlk-toc.html

[8] Linux 裝置 Deriver入門，Jollen網路學院，www.jollen.org

[9] PCI Local Bus Specification，Revision 2.2，PCI Special Interest Group， 1998

[10]Abraham Silberschatz et al.,Operating System Principles,seven edition,John Wiley & Sons.INC,2004

[11]Wayne Wolf,Computers as components,Principles of Embedded Computing System Design,Morgan Kaufmann Publishers, San Francisco,2001

[12]David A.Patterson/John L.Hennessy,Computer Organization and Design,the Hardware/Software Interface, third edition, Morgan Kaufmann Publishers,San Francisco,2005

[13] Bovet, Daniel P./ Cesati, Marco, Understanding The Linux Kernel, Oreilly & Associates Inc,Nov/2005

[14] 平田豐，Linux 裝置驅動程式 Programming 驅動程式設計，鄧瑋敦譯，博碩文化出版，Jan/2009

[15] 宋寶華，Linux 裝置驅動程式之開發，松崗出版社，Nov/2008

[16] 王進德，嵌入式 Linux 程式設計，全華圖書公司，修訂二版，Aug/2008

[17] 位元文化,C 語言入門進階,松崗出版社,May/2003

Appendix A—PNX1005 的暫存器

暫存器	功能
Reset Module	
RST_CTL	做 PNX1005 內部的重置或是 PNX1005 外部整個系統的重置。
LOCAL_SW_RESETS	負責做 PNX1005 內部某個的硬體重置(TM3282、QVCP、SVIP 等)
TM System Module	
TM_CONTROL	用來控制 PNX1005 與 TM3282 的運作。例如:是否啓動 TM3282、要求開機或關機。
TM_STATUS	檢查目前 PNX1005 是否處在 idle(reset)裝態、TM3282 是 big endian 還是 little endian。
Global MMIO 暫存器	
PCI_INTA_暫存器	是一個中斷腳位。PNX1005 或是 ARM host 可以藉由設定它來產生 interrupt。
其它與 interrupt 有關的暫存器	
IPENDING	中斷懸置暫存器，用來判斷是由哪一個中斷來源發出的中斷。
ICLEAR	清除中斷。
IMASK	中斷遮罩暫存器，是否接受該中斷來源要求的服務而發中斷出去。
TM 其他暫存器	
TM32_START	正式啓動 TM3282。
TM32_DRAM_HI	DRAM aperture 的最大值。
TM32_DRAM_LO	DRAM aperture 的最小值。
TM32_DRAM_CLIMIT	DRAM aperture 中從 TM32_DRAM_LO 開始可以被 TM3282 cache 的最大範圍。
TM32_START_ADR	從何處開始執行 DSP 程式。

(資料來源:[1] PNX100X Series Data Book)

In tm32mmio.h

```
#define TM32_RESET_CTL          (0x060000) // Reset module
#define TM32_RESET_ALL          (0x05)     // Resets the TM and all board
peripherals
#define TM32_SYSRESET_ASSERT    (0x01)
#define TM32_SYSRESET_DEASSERT  (0x02)

#define TM_CONTROL             (0x063700)
#define TM_CONTROL_TM_APERT_MODIFIABLE_MASK (0x00000008)

#define TM_STATUS              (0x063704)
#define LOCAL_SW_RESETS        (0x060010)

#define TM32_START               (0x100030)
#define TM32_DRAM_LO             (0x100034)
#define TM32_DRAM_HI             (0x100038)
#define TM32_DRAM_CLIMIT         (0x10003c)
#define TM32_APERT1_LO           (0x100040)
#define TM32_APERT1_HI           (0x100044)
#define TM32_START_ADR           (0x100048)
#define TM32_PC                  (0x10004C)
#define TM32_MODID               (0x100FFC)

#define IPENDING              (0x100820)
#define ICLEAR                (0x100824)
#define IMASK                 (0x100828)

#define DC_LOCK_CTL           (0x100010)
#define TM3218_2B80_MOD_ID      (0x00002B95)

#define PCI_MMIO_BASE            (0x00040000)
#define PCI_BASE1_LO             ((PCI_MMIO_BASE) | (0x0018))
#define PCI_BASE1_HI             ((PCI_MMIO_BASE) | (0x001C))

#define GLOBAL_REGS_BASE         (0x00063000)
#define PCI_INTA_暫存器          ((GLOBAL_REGS_BASE) | (0x0050))
#define PCI_INTA                 (0x00000002)
#define PCI_INTA_OE              (0x00000001)

#define GPIO_CFG_VALUE           (0x000AAAAA)
#define GPIO_CFG_0_15_REG_OFFSET (0x104000)
#define GPIO_VAL_0_15_REG_OFFSET (0x104010)
```

Appendix B—DSP 端各種指令對應的 ID

DSP 端指令	C Runtime Call ID
__argc	HostCall_ARGV_ARGC_INFO
__argv	HostCall_GET_ARGUMENT_STRING
close()、fclose()	HostCall_CLOSE
fgetc() 、fgets()、fscanf() getc()、gets()、read()、scanf()	HostCall_READ
fopen()、open()	HostCall_OPEN
fprintf()、fputc()、fputs() printf()、putc()、puts()、write()	HostCall_WRITE
isatty()	HostCall_ISATTY
stat()	HostCall_STAT
fcntl()	HostCall_FCNTL
fseek()、lseek()	HostCall_LSEEK
fstat()	HostCall_FSTAT
_psos_exit()、exit()、tmMain_EXIT()	HostCall_EXIT
accept()	HostCall_ACCEPT
access()	HostCall_ACCESS
bind()	HostCall_BIND
Closedir()	HostCall_CLOSEDIR
Closesocket()	HostCall_CLOSESOCKET
fsync()	HostCall_FSYNC
getenv()	HostCall_GETENV
gethostbyaddr()、gethostbyaddr_r()	HostCall_GETHOSTBYADDR_R
gethostbyname()、gethostbyname_r()	HostCall_GETHOSTBYNAME_R
gethostname()	HostCall_GETHOSTNAME
getprotobyname()	HostCall_GETPROTOBYNAME
getsockname()	HostCall_GETSOCKNAME
getsockopt()	HostCall_GETSOCKOPT
inet_addr()	HostCall_INETADDR

ioctlsocket()	HostCall_IOCTLSOCKET
link()	HostCall_LINK
listen()	HostCall_LISTEN
mkdir()	HostCall_MKDIR
mktemp()	HostCall_MKTEMP
opendir()	HostCall_OPENDIR
putenv()	HostCall_PUTENV
readdir()	HostCall_READDIR
recv()	HostCall_RECV
recvfrom()	HostCall_RECVFROM
rename()	HostCall_MOVE
rewinddir()	HostCall_REWINDDIR
rmdir()	HostCall_RMDIR
select()	HostCall_SEND
sendto()	HostCall_SENDTO
setsockopt()	HostCall_SETSOCKOPT
socket()	HostCall_SOCKET
sync()	HostCall_SYNC
system()	HostCall_SYSTEM
time()	HostCall_TIME
tmpnam()	HostCall_TMPNAM
unlink()	HostCall_UNLINK

/*-------------------------定義各個 C Runtime Call ID 的值 ------------------------------*/

```
typedef enum {
/*0*/     HostCall_OPEN,
/*1*/     HostCall_FSTAT,
/*2*/     HostCall_ISATTY,
/*3*/     HostCall_READ,
/*4*/     HostCall_LSEEK,
/*5*/     HostCall_WRITE,
/*6*/     HostCall_CLOSE,
/*7*/     HostCall_UNLINK,
/*8*/     HostCall_MKTEMP,
/*9*/     HostCall_GETENV,
/*10*/    HostCall_LINK,
/*11*/    HostCall_TIME,

/* ---------- obsolete messages for tmsim/tmdbg connection ---------- */
```

```
/*12*/    HostCall_SOCK_SEND,
/*13*/    HostCall_SOCK_RECV,
/*14*/    HostCall_SOCK_STATUS,
/*15*/    HostCall_SOCK_DATA,
/* ---------- end of obsolete tmsim/tmdbg messages ------------------------- */

/*16*/    HostCall_ARGV_ARGC_INFO,
/*17*/    HostCall_GET_ARGUMENT_STRING,
/*18*/    HostCall_EXIT,
/*19*/    HostCall_TMPNAM,
/*20*/    HostCall_FCNTL,
/*21*/    HostCall_SYSTEM,
/*22*/    HostCall_OPENDLL,
/*23*/    HostCall_STAT,
/*24*/    HostCall_MALLOCBUFF,
/*25*/    HostCall_FREEBUFF,
/*26*/    HostCall_READBUFF,
/*27*/    HostCall_WRITEBUFF,
/*28*/    HostCall_MKDIR,
/*29*/    HostCall_RMDIR,
/*30*/    HostCall_PUTENV,
/*31*/    HostCall_FSYNC,
/*32*/    HostCall_SYNC,
/*33*/    HostCall_ACCESS,
/*34*/    HostCall_OPENDIR,
/*35*/    HostCall_READDIR,
/*36*/    HostCall_REWINDDIR,
/*37*/    HostCall_CLOSEDIR,

/* ---------- messages for socket calls ---------------------------------- */
/*38*/    HostCall_SOCKET,
/*39*/    HostCall_ACCEPT,
/*40*/    HostCall_BIND,
/*41*/    HostCall_CLOSESOCKET,
/*42*/    HostCall_CONNECT,
/*43*/    HostCall_INETADDR,
/*44*/    HostCall_LISTEN,
/*45*/    HostCall_RECV,
/*46*/    HostCall_SEND,
/*47*/    HostCall_GETHOSTBYADDR_R,
/*48*/    HostCall_GETHOSTBYNAME_R,
/*49*/    HostCall_GETHOSTNAME,
/*50*/    HostCall_GETPROTOBYNAME,
/*51*/    HostCall_GETSOCKNAME,
/*52*/    HostCall_GETSOCKOPT,
/*53*/    HostCall_IOCTLSOCKET,
/*54*/    HostCall_RECVFROM,
/*55*/    HostCall_SELECT,
/*56*/    HostCall_SENDTO,
/*57*/    HostCall_SETSOCKOPT,
/* ---------- end of messages for socket calls ----------------------------- */

            /* New Calls */
/*58*/    HostCall_MOVE,
/*59*/    HostCall_STACKTRACE,
/*60*/    HostCall_STAT64,                /* 64 bits IO functions */
/*61*/    HostCall_FSTAT64,               /* 64 bits IO functions */
/*62*/    HostCall_LSEEK64                 /* 64 bits IO functions */
} HostCall_FunctionCode;
```

Appendix C—linux 開發主機中編譯 tmman 驅動程式、ARM 端應用程式的環境設定

1. 編譯 tmman 驅動程式的環境設定：
   --安裝 TCS5.4
   --使用 tmComm_patchv0.99 修補 TCS5.4/tmComm
   (/TCS5.4)
   --從 NXP FTP 取得 LINUX kernel 2.6.22.18
   (/usr/src/linux-2.6.22.18-NXP,而且要注意其中的 Makefile 裡的 cross compiler 設定)
   --進入/TCS5.4/tmComm/src
   --修改 uservars.sh 的路徑(export 的部份)
   --source uservars.sh 設定環境路徑
   --./buildscript
   --到 uservars.sh 裡面設定 $BUILDTOP 路徑的 private-tmComm 目錄下去取得驅動程式 s 以及 tmload 等常用的 tools

2. 編譯 ARM 端 TCS5.4/tmComm/src/examples 中的範例應用程式的環境設定：
   --host:改/TCS5.4/tmComm/src/examples/inc/mklinux.inc 路徑
   --target:改/TCS5.4/tmComm/src/examples/XXX/target/Makefile.Linux 的
   TCS.ENDIAN(el)、CPUTYPE(pnx1005)...
   --make -f Makefile.Linux

Appendix D--TMMan 驅動程式辨別多個相同類型的 PNX1005 的機制

要求同類型的裝置提供不同的服務: minor number

一個驅動程式可以控制多個相同類型的裝置，傳統上驅動程式內部會利用 minor number 來辨別要控制的是哪個裝置，如果希望每個裝置提供不同的功能，那就必須額外再設計幾個 file_operations 結構出來，針對不同的裝置，我們可以用一個 switch 結構依照 minor number 去分配不同的 file_operations 結構給不同的裝置，讓驅動程式可以要求不同的裝置做不同的處理。如(表 1)。

如果使用這種方式，就必須為每個裝置都建立一個裝置檔，並給予它們同樣的 major number 與不同的 minor number 來做裝置檔的身分別。當應用程式呼叫 open 這個系統呼叫要打開某個裝置檔，首先會利用註冊的時候指定給 kernel 的 default 的 file_operations 結構來處理這個 open 呼叫，這個 open 是為了從系統眾多的裝置 驅動程式 s 裡挑出代表該裝置的裝置檔所對應到的驅動程式，也就是挑出對應的 default file_operations 結構。kernel 同時會在 kernel space 載入該裝置檔的 file 結構而且這個 file 結構裡目前儲存的是一開始註冊時告知 kernel 的 default file_operaionts 結構。之後因為這個驅動程式可能可以控制多個同類型的裝置，所以驅動程式內部要依照代表不同的裝置所用的 minor number 去修改 file 結構裡儲存的 file_operations 結構，來切換驅動程式 handlers。於是原本的 open handler 裡會依照不同的 minor number 指定不同的 file_operations 結構給它們，最後呼叫新的 file_operations 結構裡的 open handler 來做該裝置真正的操作。

(表 1)

```
int devone_open(struct node * inode,struct file *file){
    Switch(iminor(inode)){
        case 0:
            file->f_op=&zero_fops;break;
        case 1:
            file->f_op-&one_fops;break;
        default:
        break;
    }
    if(file->f_op &&   file->f_op->open){
        return file->f_op->open(inode,file);
    }
return 0;}
```

註冊 drive 時所指定的 file_operations 結構

根據裝置檔的 minor number 來分別 assign 不同的 file_operations

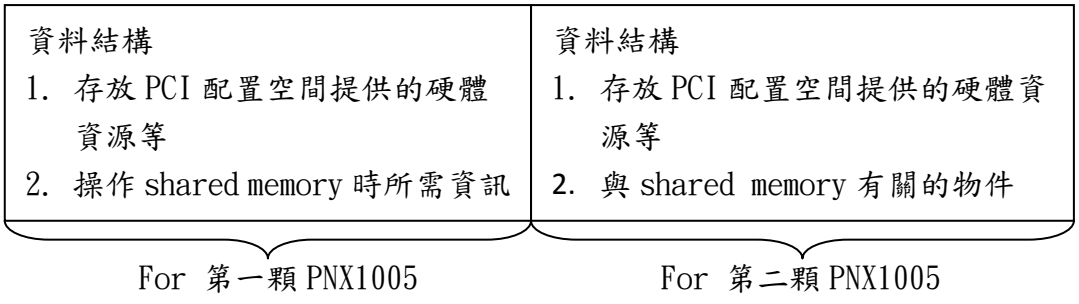
呼叫 minor number 所指定的 file_operations 結構裡的 open handler

在應用程式呼叫 open 這個系統呼叫後，驅動程式在內部就做完了 file_operaions 的切換，也等於把該裝置的裝置檔所對應的 file_operaions 結構轉到由 minor number 去對應的 file_operaions 結構了，之後當我們使用其他系統呼叫時，因為會指定要對哪個裝置檔做操作，所以就會直接使用到切換過後的 file_operations 結構裡的驅動程式 handlers 來控制這個裝置。

要求同類型的裝置提供類似的服務:資料結構

除了使用 minor number 來讓一個驅動程式可以分辨並控制不同的裝置，如果系統裡同類型的裝置所要提供的服務是相似或相同的，我們大可不必為不同的裝置去建置不一樣的裝置檔、一一分配相同的 file_operations 結構給它們，我們可以利用一些資料結構來區分這些同類型的裝置。

在 TMMan 驅動程式中，因為對每個 PNX1005 要求的服務是一樣的，所以每個 PNX1005 所使用的 file_operations 結構是一樣的,於是我們只利用 mknod 建立一個名為”pnx1005”的裝置檔，利用 major number 將裝置檔與驅動程式的關係建立起來。由於我們並不需要用 minor number 去分別 assign 不同的 file_operations 結構給不同的 PNX1005，但是因為每個 PNX1005 的硬體資源以及與 ARM 溝通的共享記憶體所在空間及管理共享記憶體的管理架構所使用的 kernel space 都不同，所以它在 kernel space 虛擬記憶體空間裡使用了一個陣列(如圖 1)，陣列裡的每個 element 都儲存了一個資料結構，而這個資料結構則儲存了每個 PNX1005 要與 ARM 溝通所需的共享記憶體位置、管理共享記憶體的功能區塊的位址資訊以及 PNX1005 裝置在 PCI 配置空間裡被配置到的硬體資源(表 2)，因此我們也可以把這個資料結構當成是描述 PNX1005 的結構，可以用來代表一個 PNX1005。



(圖 1)結構陣列

(表 2)代表 PNX1005 的資料結構內容

結構成員	用途
UInt32 DSPNumber	這個結構是給哪一個 PNX1005 使用的
TMManSharedStruct* SharedData;	在共享記憶體最開頭提供了讓 DSP 可以從中獲取 ARM 開出的共享記憶體資訊的資料結構，其

	kernel layer 的起始位址，第四章會討論到。
PHYSICAL_ADDRESS TMManSharedAddress;	在共享記憶體最開頭提供了讓 DSP 可以從中獲取 ARM 開出的共享記憶體資訊的資料結構，其實體記憶體起始位址。
UInt32 **HalHandle**; UInt32 ChannelManagerHandle; UInt32 VIntrManagerHandle; UInt32 EventManagerHandle; UInt32 MessageManagerHandle; UInt32 MemoryManagerHandle; UInt32 NameSpaceManagerHandle;	各種運作共享記憶體時會使用到的結構的虛擬記憶體位址。HalHandle 儲存了 PCI 配置空間提供的資訊，其他部分則與共享記憶體的使用有關。
PHYSICAL_ADDRESS HalSharedAddress; PHYSICAL_ADDRESS EventSharedAddress; PHYSICAL_ADDRESS ChannelSharedAddress; PHYSICAL_ADDRESS VIntrSharedAddress; PHYSICAL_ADDRESS DebugSharedAddress; PHYSICAL_ADDRESS MemorySharedAddress; PHYSICAL_ADDRESS MemoryBlockAddress; PHYSICAL_ADDRESS NameSpaceSharedAddress;	存放各種共享記憶體上各種功能區塊的 physical 位址，第四章會討論。
其他	

Appendix E—user library 初始化所需的廣域結構

1. 全域結構

```
typedef struct  _TMMAN_GLOBAL {
    // total number of DSPs (of all types) in the system
    UInt32                  DSPCount;

    // flags to ensure that tmmanInitialize() is called only once
    Bool                    TMManInialized;

    UInt32                  MapSDRAM;
    UInt32                  DisableDialogBox;

    //mutex to perform MP safe operations in this device
    LOCK                    MessageCriticalSection;

    // handle lists for debugging and detecting leaks
    HANDLE_LIST             DSPList;
    HANDLE_LIST             MessageList;
    HANDLE_LIST             EventList;
    HANDLE_LIST             MemoryList;
    HANDLE_LIST             SGBufferList;

    /// one entry per driver in the system
    TMMAN_DRIVER_INFO       DriverMap[TMMAN_MAXIMUM_DRIVERS];

    // one entry for every device in the system
    TMMAN_DEVICE_INFO       DeviceMap[constTMMANMaximumDeviceCount];

}   TMMAN_GLOBAL, *PTMMAN_GLOBAL;
```

2. 全域結構中的 TMMAN_DRIVER_INFO 結構

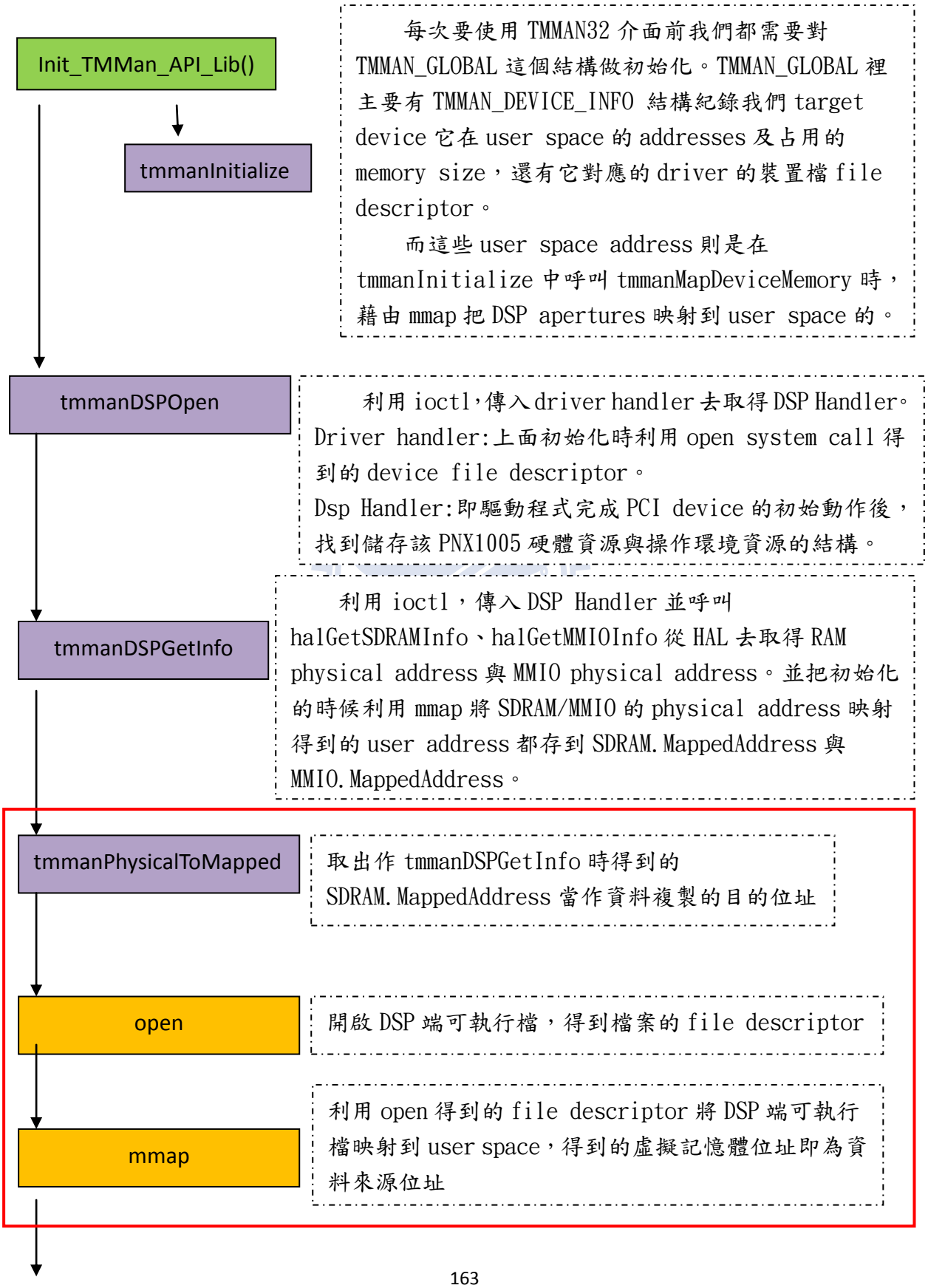
```
typedef struct _TMMAN_DRIVER_INFO
{
    // number of DSPs supported by this driver
    UInt32  DSPCount;

    // OS handle to the driver ( or device )
    HANDLE  DriverHandle;
} TMMAN_DRIVER_INFO, *PTMMAN_DRIVER_INFO;
```

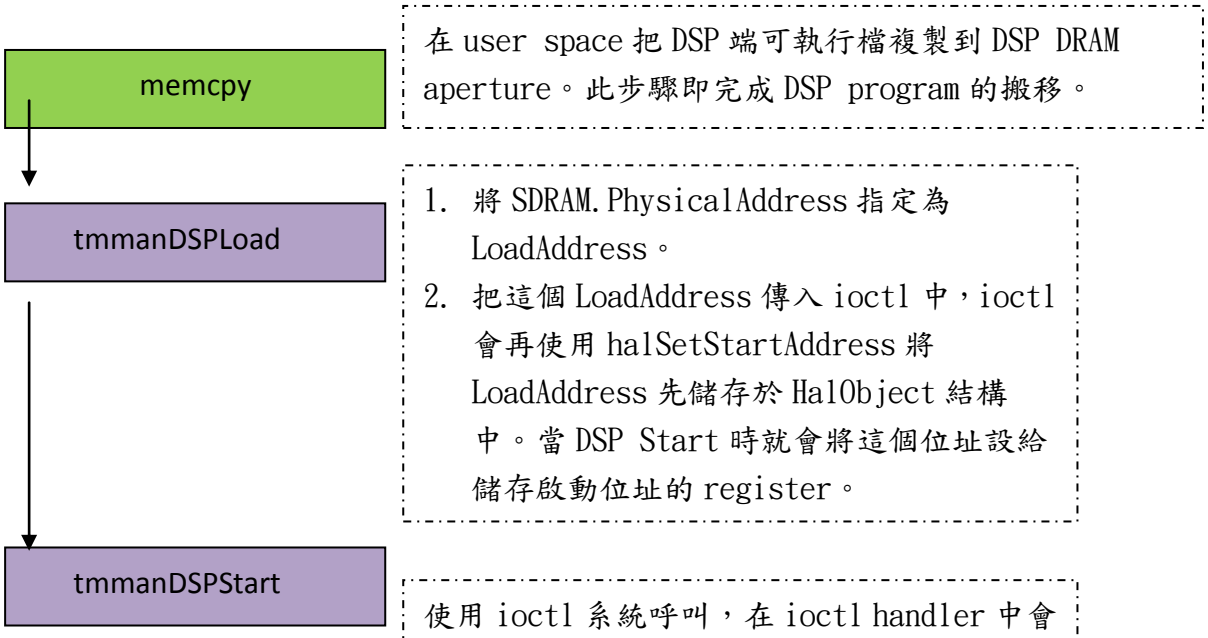
3.全域結構中的 TMMAN_DEVICE_INFO 結構

```
typedef struct _TMMAN_DEVICE_INFO
    {
    //SDRAM User Mode Mapping
    UInt8*                  SDRAMUserAddress;
    UInt32                  SDRAMSize;

    //MMIO User Mode Mapping
    UInt8*                  MMIOUserAddress;
    UInt32                  MMIOSize;

    //XIO User Mode Mapping
    UInt8*                  XIOUserAddress;
    UInt32                  XIOSize;
    //SHMEM User Mode Mapping
```

```
    UInt8*                  ShmemUserAddress;
    UInt32                  ShmemSize;

    // DSP number within this dirver
    UInt32                  DSPNumber;

    // pointer driver that supported this device
    PTMMAN_DRIVER_INFO      DriverMap;
} TMMAN_DEVICE_INFO, *PTMMAN_DEVICE_INFO;
```

Appendix F—函式庫應用實例:tmload 中下載 DSP 端可執行檔的運作




memcpy
在 user space 把 DSP 端可執行檔複製到 DSP DRAM aperture。此步驟即完成 DSP program 的搬移。
tmmanDSPLoad
1. 將 SDRAM.PhysicalAddress 指定為 LoadAddress。
2. 把這個 LoadAddress 傳入 ioctl 中，ioctl 會再使用 halSetStartAddress 將 LoadAddress 先儲存於 HalObject 結構中。當 DSP Start 時就會將這個位址設給儲存啟動位址的 register。
tmmanDSPStart
使用 ioctl 系統呼叫，在 ioctl handler 中會呼叫 halStartDSP，從 HAL 取出 MMIOKernelAddress，然後加上適當的 offset 指到不同的 register 做好設定，就能啟動 DSP CPU。

Appendix G—copy_from_user/copy_to_user

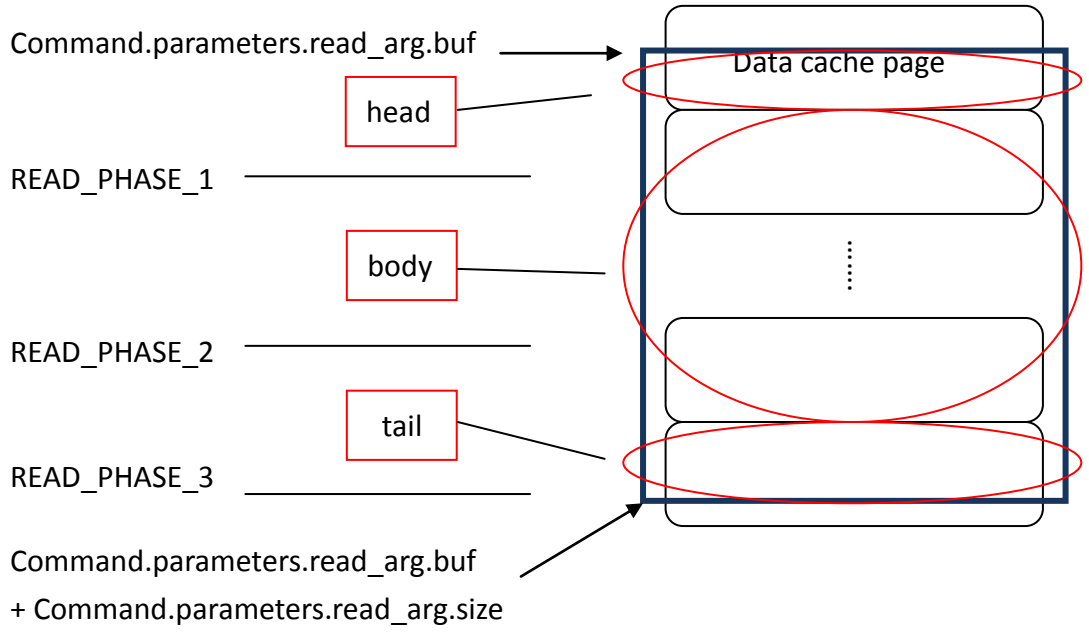
```
unsigned long
copy_from_user(void *to, const void __user *from, unsigned long n)
{
    might_sleep();
    if (access_ok(VERIFY_READ, from, n)) //type 表示對使用者空間作讀取
    n = __copy_from_user(to, from, n); //用組合語言寫複製的動作
    else
    memset(to, 0, n);
    return n;
}

unsigned long
copy_to_user(void __user *to, const void *from, unsigned long n)
{
    if (access_ok(VERIFY_WRITE, to, n)) //type 表示對使用者空間作寫入
    n = __copy_to_user(to, from, n); //用組合語言寫複製的動作
    return n;
}
```

Appendix H—使用 termination_handler 重新請求 ARM 端服務(讀檔)

fread/read/scanf 都是以 read()形式在運作，但由於 DSP 端在系統中只是一個 slave 的角色，我們並沒有把 ARM 端的記憶體空間映射到 DSP 端的 DRAM 上，而且從 ARM 端讀過來的資料每次的大小不盡相同，也不一定可以與 DSP 端的 Data cache page 對齊(每個 Data cache page 大小約為 64B)，而且 DSP 端有規定想以 DMA 的方式來傳送資料的話，整體資料大小必須在 320B 以上，所以並不是每次所要讀的檔案大小都可以做 DMA(direct host write)，有時候還是只能用 data-copy 的方式進行，比較耗費 CPU 資源。

在實作 TCS C library 裡的 read 請求時，除了小於 DMA threshold 大小的資料只能透過 ARM 端做 data-copy 讀檔，其他大小的資料會被切成三塊(head、body、tail，如圖 1)，假設資料大小都已大於 DMA threshold，head 為該資料起始位址到第一個遇到的 data cache page 邊界之間的區塊，body 則為從 data cache page 邊界開始到最後一個資料所在 data cache page 的邊界之間的區塊，而 tail 就是剩下來的部分資料。DSP 端想要讀取的資料的 Head 和 tail 我們會為他們另外一塊記憶體中預留一部分空間讓 ARM 端可以以 data－copy 的方式將資料先複製到這個記憶體空間後，DSP 端再將這段空間讀到的資料複製到應用程式開出的空間去；而 body 部分的資料 DSP 端只需要向 ARM 端請求一次，ARM 端則可以直接透過 DMA 方式將資料存到應用程式開出的記憶體空間，減少 data-copy 帶來的 delay，因此可以增加 throughput。

(圖 1)讀檔時對資料的分段



於是，DSP 端應用程式如果使用了，read()、fread()、getc()、fgetc()、gets()、fgets()、scanf()、fscanf()等其它讀檔類的函式，都會先經由 TCS C library 轉成 POSIX I/O call，然後利用 I/O driver components 將指令包裝成以 HostCall_READ 為 command 代號的 HostCall_command 結構，並會設定好讀回的檔案所存放的位址、還有所讀檔案的 file descriptor 等於 HostCall_command 結構 read_args 參數結構欄位中。5-4 節一開始有提到，為了增加讀檔效率，我們需要為欲讀取的檔案資料做大小的分配，分為可利用 DMA 或只能利用 data-copy 做資料讀取的區塊，因此，在 DSP 端我們基本上會把讀檔的動作分成三個 phase 來進行(類似 state machine，讓讀檔動作以更小的單位來進行)，在這三個 phase 中主要就是去重新設定 HostCall_command 結構的欲讀取檔案的大小與儲存的位址，然後把修改過的 HostCall_command 結構位址重新啟動 hostcall library，將新的請求傳到 ARM 端繼續下一部分的讀檔動作，至於如何更新 HostCall_command 結構則於下段說明。

所有的讀檔相關調整都是由 DSP 負責的，ARM 端只負責針對收到的 control block 裡的 HostCall_command 結構內容做讀取，確認收到的請求為 HostCall_READ 這個 command 代號，然後就參數結構中的資訊作讀檔動作；當 DSP 端送出 control block 位址後，DSP 端就會去調整 phase 以記錄目前讀檔進度，這個 phase 會設定於 HostCall_command 結構的 code 變數中，為了避免 DSP 端在內部做 phase 調整時會更改到送給 ARM 端的 control block 內容，這樣 ARM 端收到 control block 時會無法判斷要執行哪種指令，所以 DSP 端是在 hostcall library 那時候傳下來的 HostCall_command 結構內容做更新，而不是 control block 裡的 HostCall_command 結構內容，而調整的動作則是在 termination_handler 裡執行。以下針對欲讀取的檔案超過 DMA threshold 大小的情況，做了簡單的說明：假設 I/O driver components 裡的 ReadFunc 已填寫完 HostCall_command 結構的 command 代號與初步的 read_args 參數結構內容，然後啟動 hostcall library，

1. 由 hostcall library 設定 HostCall_command 結構的內容(command 代號、服務狀態、termination_handler 位址、notification_handler 位址、thread 身分(表 1)，然後 host_comm library 會根據 command 代號、欲讀取的檔案大小決定讀回的檔案是否要在 control block 中需要使用額外的 result buffer 來儲存，而且是要預留多大的 result buffer(只有使用 DMA 的方式傳輸時才不需要 result buffer)。然後另外以 HostCall_command 結構大小加上 result buffer 大小創建一個 control block，將目前 HostCall_command 結構內容複製進去，如果有 result buffer 就在 control block 中預留這塊空間。

(表 1)每次要求讀檔時 DSP 填入的 command buffer 內容

```
command.code                              = HostCall_READ;
command.parameters.read_args.fildes  = file;
command.parameters.read_args.buf      = buf;
command.parameters.read_args.nbyte    = nbyte;
command->requester              = AppModel_current_thread;
command->status                 = HostCall_BUSY;
command->notification_status    = HostCall_BUSY;
command->termination_handler    = (HostCall_Termination_Handler)termination_handler;
command->notification_handler = notify;
command->returned_errno         = 0;
command->sending_node           = _node_number;
```

2. 檢查非 control block 裡的 HostCall_command 結構裡 code 變數儲存的 command 代號，設定為原本的 HostCall_READ，表示這是第一次傳送，所以我們就把 code 變數更新為 READ_phase_1，然而如果已經發現 code 變數已經為 READ_phase_1，則把 phase 更新為 READ_phase_2，以此類推，最多會更新至 READ_phase_3；更新完 code 變數後，就會檢查非 control block 裡 HostCall_command 結構的參數結構，如果欲接收的檔案大小 read_args.nbyte 為 0，表示已經沒有要接收的檔案了，就直接把非 control block 裡的 HostCall_command 結構的服務狀態(status)設為 HOST_DONE，停止傳送 control block 位址的動作，並把 termination_handler 設為 Null，將目前 HostCall_command 結構中的回傳值大小 retval 回傳給應用程式；不然的話，就將 control block 位址存入 packet 中，然後使用 tmmanMessageSend，透過底層溝通機制傳送到 ARM 端，繼續完成下面的動作。
3. 當 ARM 端做完 DSP 提出的讀檔請求後，不管檔案有無讀取完，當 DSP 端收到 ARM 端的回覆，會由 termination_handler 判斷這個讀檔動作是否使用到 result buffer，有的話就需要從 result buffer 將資料複製回應用程式開出的 buffer 裡，如果沒有的話就代表資料利用 DMA 直接傳送到應用程式開出的 buffer 裡了，然後更新非 control block 的 HostCall_command 結構中，讀檔參數結構的「讀檔資料放置的位址」。
4. termination_handler 接著會從 control block 裡 command buffer 的 retval 變數中得到這次已經讀取了多少大小的檔案，然後與 control block 裡 HostCall_command 結構參數結構儲存的--所要讀取的檔案大小--做相減，得到還需要讀取多少大小檔案，如果還有需要讀取的話，就更新非 control block 的 HostCall_command 結構中參數結構的 read_args.nbyte 為剩下的而且欲讀取的檔案大小，並將非 control block 的 HostCall_command 結構的 retval 變數加上 ARM 端做修改後，寫入 control block 裡 HostCall_command 結構的 retval 變數，以更新非 control block 裡 command buffer 的回傳值資訊。
5. termination_handler 會將前一個 control block 的 command buffer 區域釋放掉，而

更新完內容後，非 control block 的 HostCall_command 結構中的 code 變數如果不為 READ_phase_3，在傳送訊息給 ARM 以前，又會從第一點開始重新運作；如果此時發現，非 control block 的 HostCall_command 結構的 code 變數已為 READ_phase_3，就表示所有資料已經傳送完畢，於是 TCS C library 就會將目前 HostCall_command 結構中的 retval 值回傳給應用程式。

由上可知，termination_handler 負責了檔案讀取時 command buffer 的資訊更新(欲讀取的檔案大小、存檔位址)，如果處理進度已經到 READ_PHASE_3 或所需讀取檔案大小為 0，就終止對 ARM 端發出請求的程序。(表 2)為讀檔時 DMA 與 data-copy 可能產生的組合。(表 2)

欲讀取的總檔案大小	使用的 hostcall 次數與讀檔處理機制
小於 DMA threshold 或小於一個 data cache page	一次 hostcall with data-copy
大於 DMA threshlod 但是剩下來的資料卻不到一個 data cache page	共兩次 hostcall: 兩個 data-copy
大於 DMA threshold 且剩餘資料剛好是 data cache page 大小的倍數	共兩次 hostcall: 一個 data-copy 傳 head;一個 DMA 傳 body
大於 DMA threshold 且剩餘資料不會剛好是 data cache page 大小的倍數	共三次 hostcall: 一個 data-copy 傳 head;一個 DMA 傳 body;一個 data-copy 傳 tail