

國立交通大學

電信工程研究所

碩士論文

適合影像處理的串流模組設計

Streaming Module Design for Video Processing

研究生：陳怡如

指導教授：張文鐘 博士

中華民國 九十九 年 八 月

適合影像處理的串流模組設計

Streaming Module Design for Video Processing

研究生：陳怡如

Student : Yi-Ju Chen

指導教授：張文鐘

Advisor : Wen-Tong Chang

國立交通大學

電信工程研究所

碩士論文

A Thesis

Submitted to Institute of Communication Engineering
College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Communication Engineering

August 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年八月

適合影像處理的串流模組設計

研究生：陳怡如

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



本論文是以 TSSA (Trimedia Streaming Software Architecture) 軟體架構所建立的模組結構以及其模組化的串流方式為探討的重點。模組化的概念是將不同性質的影像處理程式區分開來，讓各模組具有單一化的執行功能，並且具備獨立的資料結構以及執行函式。此外模組會以單純的介面函式來與應用程式以及系統連結，提供給外界簡易的操作方式，而模組本身的影像處理內容，則是透過模組內部的函式來呼叫執行，不讓外部的應用程式或者其它模組直接來操作，避免模組本身的資料內容遭到修改，給予模組多一層的保護。TSSA 模組的串流方式是透過傳輸模組來運作。在兩兩影像處理模組之間會連接一個傳輸模組，而傳輸模組上的傳輸條件以及傳送單位，則會根據前後連接的影像處理模組特性來設定。當模組之間要傳送影像資訊時，會透過系統函式來取得傳輸模組上的封包，提供給影像處理模組來讀取或寫入資料。這部分的運作機制會透過對應用程式執行上實際的測試來了解，此外會再藉由於影像處理模組中增添新的函式功能，對模組的運作方式做進一步的了解與應用。

Streaming Module Design for Video Processing

Student: Yi-Ju Chen

Advisor: Dr. Wen-Thong Chang

Institute of Communication Engineering

National Chiao Tung University

ABSTRACT

The primary issue of this thesis is the component architecture and streaming of TSSA (Trimedia Streaming Software Architecture). The concept of modular is separating different types of image processing programs into several parts each with its own data structures and functions. In addition, each component would be linked to the application program and system by some simple interfaces, which provide an easy way for users to operate. Meanwhile, components' image processing procedure would be executed by the internal functions to avoid modifying from application programs or other components. The streaming of TSSA components is operating on the in-out descriptor, which is generalized as a connection component. The in-out descriptor is connected between two image processing components, and the transmitting conditions of in-out descriptor are based on the properties of components it's connected. While the image processing components need to transmit data, the packets of in-out descriptor are provided via system functions. The mechanism of communication between components would be verified through the experiment of some programs and an additional function would be added into the component to further clarify the operations.

誌 謝

時光荏苒，研究所兩年的歲月匆匆流逝，在這段期間裡最要感謝的人是我的指導教授 張文鐘博士。不論在學業或生活上，老師都給予我相當多的指導與幫助，尤其在最後這段整理論文以及準備口試的日子裡，老師更是抽出了寶貴的時間辛苦的待在實驗室，指導我在研究上所需要加強的部分，並為我修改論文上的不足，對於老師的付出，內心真的只有滿滿的感激。同時也要感謝口試委員黃仲陵教授、范國清教授以及余孝先博士，在口試時給予我研究上與論文上的建議，有了您們的指導才使得這篇論文更趨完整。

很幸運能成為 821 實驗室的一員，在兩年實驗室的生涯裡，大家總是不吝惜對我的關懷與照顧，讓我能感受到實驗室大家庭的溫暖。博班學長家豪，96 級學長琮壹，98 級與 99 級的學弟妹舒評、信好、維哲、耀駿、詩倩，助理立杰，以及 97 級的好夥伴們耀葦、明穎、雅嵐，感謝你們對我研究上以及生活上的幫助，有了你們的陪伴，我才能夠順利走過這段研究所時光。

最後我要感謝我心愛的家人，父母親以及哥哥，謝謝你們對我的支持與關心，尤其是媽媽每天都從電話裡關心我在這裡生活的一切，給予我最大的鼓勵，真的很感激你們對我的付出。此外我也由衷的感謝男朋友以及男朋友家人對我的關懷與照顧，還有所有關心我的好朋友們。有你們才有今天的我，再次的謝謝你們。

誌於 2010.夏 新竹。交大

怡如

目 錄

摘要.....	i
ABSTRACT.....	ii
誌謝.....	iii
目錄.....	iv
圖目錄.....	vi
表目錄.....	viii
第一章 緒論.....	1
1.1 研究背景.....	1
1.2 論文架構.....	2
第二章 TSSA 模組架構.....	3
2.1 模組架構.....	3
2.2 模組的資料結構.....	6
2.2.1 Capabilities 資料結構.....	6
2.2.2 Instance 資料結構.....	8
2.3 模組的函式功能.....	12
2.4 TSSA 模組應用.....	17
第三章 傳輸模組 IOD.....	20
3.1 IOD 架構概念.....	20
3.2 IOD 的建立流程.....	21
3.2.1 IOD 結構參數.....	21
3.2.2 Default 層上 IOD 的建立.....	25
3.2.3 IOD 的傳送端與接收端設定.....	31
3.3 模組間資料串流.....	32
第四章 輸入端與輸出端模組.....	35
4.1 系統初始化設定.....	35

4.2 輸入端模組架構.....	39
4.3 SVIP 模組.....	42
4.3.1 SVIP Unit.....	42
4.3.2 SVIP Stream.....	47
4.4 輸出端模組架構.....	52
第五章 實驗驗證.....	55
5.1 傳輸數據驗證.....	55
5.2 模組實驗.....	61
5.2.1 In-Place 模組結構.....	62
5.2.2 模組實驗結果.....	64
第六章 結論.....	67
參考文獻.....	68

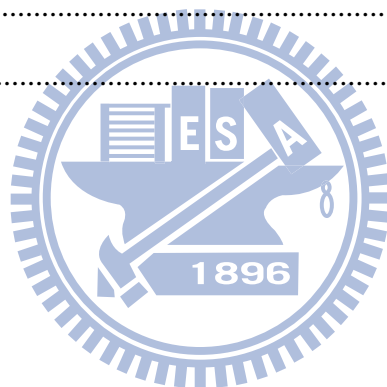


圖 目 錄

圖 2.1	模組的分層結構.....	4
圖 2.2	模組內的分層關係連結.....	5
圖 2.3	Encoder 模組 Capabilities 資料結構.....	7
圖 2.4	Instance 的結構組成關係.....	9
圖 2.5	Encoder 模組 Instance 資料結構.....	9
圖 2.6	設定 Capabilities 的函式流程.....	13
圖 2.7	Open() 函式的執行內容.....	14
圖 2.8	Default 層 Start() 函式執行流程.....	15
圖 2.9	模組 AL 層上 Start 函式的內容.....	16
圖 2.10	應用層程式全域變數的資料結構.....	17
圖 2.11	TSSA 模組與分層架構圖.....	18
圖 3.1	IOD 與影像處理模組之間的關係架構.....	20
圖 3.2	IOD 資料結構在分層上的設定關係.....	21
圖 3.3	複製 Setup 結構內容到 IOD 結構流程.....	25
圖 3.4	Queue 的建構流程.....	27
圖 3.5	Packet 的建構流程.....	29
圖 3.6	tmPacket_ArrayCreate() 函式內容.....	29
圖 3.7	Packet 的結構關係.....	30
圖 3.8	buffer 的設定流程.....	31
圖 3.9	傳送端與接收端模組的 IOD 設定.....	31
圖 3.10	拿取空 packet 的函式流程.....	32
圖 3.11	Release packet 的函式流程.....	34
圖 4.1	tmMain() 函式內容.....	36
圖 4.2	tmbslCore 系統架構.....	36

圖 4.3	BoardActivateList[] 資料陣列.....	37
圖 4.4	PCI Demo 板 A/D 晶片的註冊流程.....	38
圖 4.5	輸入端分層與模組架構.....	39
圖 4.6	tmVdecAna 模組存放硬體資訊的資料結構.....	40
圖 4.7	tmVdecAna 模組取得硬體資訊的流程.....	40
圖 4.8	Unit Capabilities 資料結構.....	43
圖 4.9	SvipUnit_localInit()內容.....	43
圖 4.10	取 unit 數的程式流程.....	44
圖 4.11	Unit Capabilities 的結構關係.....	45
圖 4.12	Unit 的 Instance 結構.....	45
圖 4.13	Unit Open 函式內容.....	46
圖 4.14	SVIP 結構.....	48
圖 4.15	tmVcapSvip 模組 AL 層 Start()函式內容.....	49
圖 4.16	svip_SetVidPacket()的函式內容.....	50
圖 4.17	SVIP 的 Video Stream 處理.....	50
圖 4.18	Buffer 位址與暫存器的設定關係.....	51
圖 4.19	啟動影像擷取的暫存器設定.....	51
圖 4.20	輸出端分層與模組架構.....	52
圖 5.1	模組之間的傳輸架構.....	56
圖 5.2	IOD、packet、buffer 的連結關係.....	56
圖 5.3	queue、packet 與 buffer 的結構關係.....	57
圖 5.4	應用程式 exH264Codec.c 模組架構.....	60
圖 5.5	In-Place 模組結構.....	62
圖 5.6	應用程式 exSvipDeinterlace.c 模組架構.....	62
圖 5.7	ModifyPacket()中加入的程式內容.....	65
圖 5.8	人臉偵測實驗結果.....	66

表 目 錄

表 2.1	tsaDefaultCapabilities 結構.....	6
表 2.2	tsaDefaultInstanceSetup 結構.....	10
表 3.1	tsaInOutDescriptorSetup 結構.....	22
表 3.2	tsaInOutDescriptor 結構.....	24
表 3.3	tmPacket_Setup 結構.....	28
表 4.1	板子與使用的 A/D 晶片	38
表 4.2	tmVdecAna 模組函式.....	42
表 5.1	IOD 的傳輸單位設定.....	58
表 5.2	packet與buffer位址.....	59
表 5.3	packet位址的數據比對.....	61
表 5.4	empty序列上儲存的packet資料結構位址.....	63
表 5.5	In-Place模組的packet傳輸數據.....	64



第一章 緒論

1.1 研究背景

目前，模組化的概念正廣泛的運用於軟體工程中，由於多媒體相關產品日益增加，產品上軟體套件功能的擴充需求也會逐漸上升，因此透過模組化的概念應用，可直接將所要擴充的功能模組新增到軟體套件上，於相同的軟體平台上運作。所以在整個軟體套件的環境中，儲存了多個獨立的功能模組，提供給應用程式來操作使用，倘若在一個應用程式上使用到了多個模組來執行，則每個獨立的模組之間就必須具備相同的介面以便傳輸溝通。對於模組化的概念以及模組的運作與傳輸方式，在本論文中，以 PNX Lite PCI Demonstration 嵌入式平台為環境，探討於該裝置上所執行的程式架構。

其中，PNX Lite PCI Demonstration 是一個簡易的嵌入式環境，以 PCI 介面連接於電腦上，在此環境中具備了一組 DSP 晶片 PNX1005、一組類比數位轉換晶片 SAA7115、一組處理高畫質影像輸入的晶片 TDA19978、一組數位類比轉換晶片 SAA7104、以及一組處理高畫質影像輸出的換晶片 TDA9983。而主要探討的內容為應用於 DSP 晶片上的軟體架構 TSSA (Trimedia Streaming Software Architecture)，以及依據 TSSA 架構所建立的軟體模組在設定與運作上流程，並於實作的部分對模組之間的資料傳輸方式進行測試，同時在其中一個影像處理模組上新增了人臉偵測的功能，對模組的資料結構以及函式設定上做進一步的應用。

在 TSSA 模組化的串流系統上，每個模組都是一個獨立的執行緒，這些執行緒會依照順序串成一個執行的序列，以平行的方式共同存在於系統上，因此在 TSSA 架構上會具備一個控管所有模組的系統核心，稱為 Default Layer。在 Default 系統核心上，首先會為應用程式中所使用到的每個模組建立一個執行緒，當應用程式要開始執行運作時，會依序呼叫模組的啟動函式，而 Default 系統在收到啟動模組的指令後，就會接著去啟動該模組的執行緒。其中在每個執行緒上都會給予一個主要的執行程式，這個主要的執行程式是包裝在模組的架構裡，所以當執行緒啟動後，就會進入到模組的執行程式裡不斷的運算處理資料，直到使用者下達停止的指令為止。因此 TSSA 架構上定義了一套撰寫模組的標準方式，內容包括了一個模組在運作上所需要的資料結構與運作函式，以及提供給應用程式在模組的操控上所需要的 API。透過這些模組的 API，系統上管理執行

緒的核心函式，就會去建造每個模組在執行上所需要的環境。這部分模組的建立與運作機制，會在論文的第二章裡進行更詳盡的說明。

應用程式上若使用到多個模組，系統上就會存在多個執行緒，而執行緒之間是以非同步的方式來做資料交換，執行緒本身只負責資料的運算處理，並不會對資料交換所需要的記憶體空間做管理，因此這部分會由系統核心來為兩兩相連接的執行緒，配置資料交換所需要使用到的記憶體空間。由於資料交換為非同步的方式，所以需要配置多個記憶體區塊，一次使用一個區塊來交換資料。在記憶體區塊上的管理方式，則會利用到兩個 Queue 來分別記錄記憶體的使用狀態，一個記錄可使用的記憶體區域位址，另一個記錄已使用記憶體區域位址。在論文的第三章裡，會針對模組之間資料傳輸的設定與運作方式進行探討，並於第五章以模組之間所傳輸數據來分析並驗證模組的傳輸方式。

1.2 論文架構

在本論文中，會以 DSP 晶片上所執行的程式為主要探討的內容，因此在論文章節的順序上，首先在第二章裡，會先針對以 TSSA 架構建立的影像處理模組進行探討，包括了模組所必須具備的資料結構和函式內容，以及模組在執行上與系統之間的介面函式連結關係；接下來在第三章裡，則會討論連結於影像處理模組之間的傳輸模組結構，並探討其資料傳輸的設定流程以及傳輸方式；第四章裡所要討論的為輸入端與輸出端的模組，由於這兩個模組在運作上會使用到 DSP 上的硬體資源，在結構上相較於一般純粹由軟體來運算的模組會更複雜些，因此獨立於此章節中說明；第五章為實作驗證的部份，藉由模組之間傳輸的數據來分析模組的傳輸機制，並將人臉偵測的功能函式應用於模組中；最後第六章為本論文的結論。

第二章 TSSA 模組架構

本章以 TSSA 架構上的模組以及其運作方式為探討的重點，內容上主要會針對在每個模組中必備的資料結構及執行函式來進行討論，並了解其相關的設定內容及運作流程；接下來則會探討如何在一個應用程式上來使用模組，並且以 TSSA 的架構建立於 DSP 晶片上執行。而在章節的順序上，首先 2.1 節中，會先針對模組的組成架構來進行討論；接下來在 2.2 節中，則會探討模組中的資料結構成員；而 2.3 節中，則是討論模組的相關函式以及其功能；最後在 2.4 節中，會對於應用程式如何來使用模組以及整體的 TSSA 架構做進一步的探討。

2.1 模組架構

模組 (Component) 是一個具備某種特定功能的影像處理單元，例如影像擷取、影像編碼、影像解碼、影像輸出……等，依據影像資料處理的類型來分成多個各具功能的模組。由於在 TSSA 系統上支援多重程序執行 (Multitasking) 的作業方式，因此每個影像處理模組都具備自己的執行緒 (thread，在程式中稱之為 task)，而這些具備執行緒的模組在 TSSA 架構上又稱為 Active Component，系統在執行上會依照各模組所設定的優先順序來進行控制權的轉換。

其中模組在架構上會具備兩大資料結構 Capabilities 以及 Instance，各模組的特性則會藉由在這兩個資料結構上不同的設定值來描述；同時在模組上還必須具備相關的執行函式，來設定或運算資料結構的內容，因此在模組中依據執行的內容來區分並定義了以下八個函式：GetCapabilities()、Open()、GetInstanceSetup()、InstanceSetup()、Start()、InstanceConfig()、Stop()、Close()，函式詳細的執行內容會在 2.3 節中做介紹。而在 TSSA 的架構上，具備一組操作所有模組的核心程式稱為 Default 層函式，各模組會以上述所提及的八個函式做為模組的操作介面，來與 Default 層連結。為了讓程式在操作上能夠更有系統，在模組上定義了兩層介面函式，分別提供給應用程式以及 Default 層函式來連結，其中提供給應用程式操作的介面函式稱為 OL 層 (Operation Layer) 函式，提供給 Default 層操作的介面函式稱為 AL 層 (Abstraction Layer) 函式，層次之間的結構關係如圖 2.1 所示。

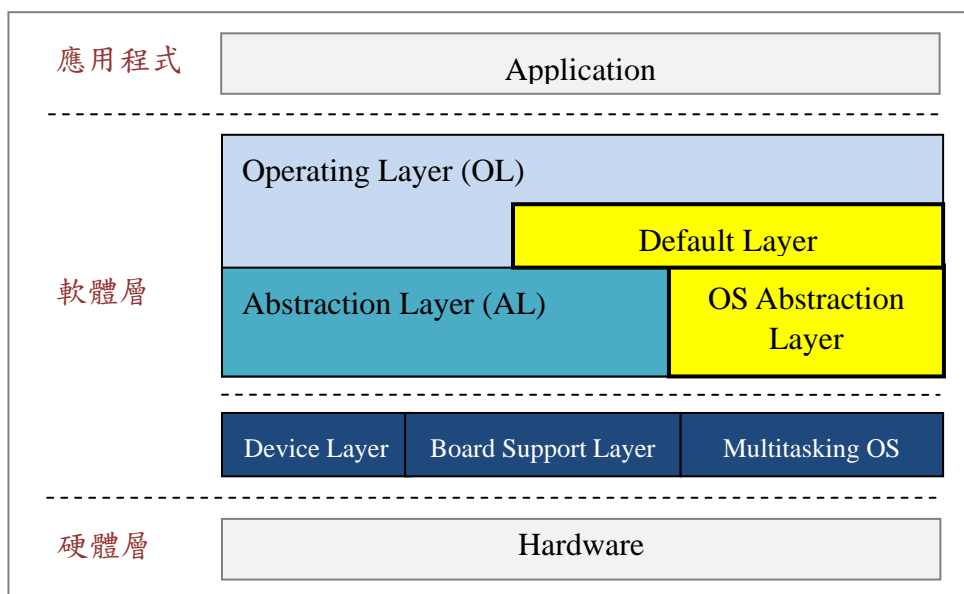


圖 2.1 模組的分層結構

接下來就以模組函式在收到應用程式傳入參數後所執行的程序，來探討模組的運作流程，以及各層面的函式在模組運作上所執行的內容。首先在應用程式上，若要對模組進行資料結構的設定或函式的操作時，就會將參數傳入模組所提供的 OL 層介面函式上，交由各模組來進行處理。而在模組 OL 層上的函式通常不具備資料運算或處理的功能，主要是做為模組在應用程式與 Default 層系統函式之間的溝通介面，模組實際處理影像資料的部分會寫在 AL 層的函式裡。因此當模組 OL 層上的函式收到應用程式傳入的參數時，就會接著將這些傳入的指標或設定值傳入 Default 層中，同時也會把該模組於 AL 層上的影像處理函式指標一併傳入，交由 Default 層函式來做一些預設的執行程序，例如資料結構的初始值設定，並提供了 AL 層的介面函式位址來給 Default 層操作。

Default 層管理了在使用到的所有模組，包括影像處理模組以及傳輸模組，同時也定義了所有模組在運作上皆會使用到的資料結構以及函式，讓模組能夠以這些預設好的資料結構或函式來設定參數，並提供模組在執行上一套固定的流程。因此當 Default 層在收到由模組 OL 層傳入的資料後，Default 層上的相關函式則會針對傳入的指令來判斷所要執行的內容，若所要執行的內容是模組中特定的處理程序，則會利用 OL 層所提供的 AL 層函式位址，連結到模組 AL 層的函式上，來取得相關的資料結構設定以及處理函式。

而模組的 AL 層又可稱為訊號處理層(Signal Processing Layer)，是存放模組特定資料結構的設定值以及影像處理函式所在的層面。模組中的 AL 層是一個純粹的影像處理層面，不需要考量到排程以及作業系統的執行，只需要在該層的函式被呼叫時提供出資料結構的設定值，或者對傳入的資料進行處理運算即可，因此可將 AL 層視為是模組的一個資料庫。圖 2.2 為模組與 Default 層之間的連接關係，由於 Default 層中定義了所有模組共同的運作程序，因此架構上的每個模組只需要編輯在 OL 層與 AL 層的程式內容，即可透過 Default 層函式來傳輸處理。

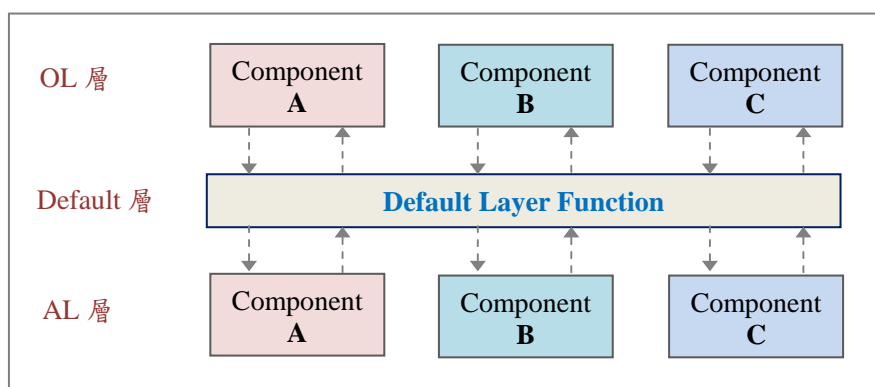


圖 2.2 模組內的分層關係連結

若 Default 層系統函式執行到與 OS 相關的內容時，會透過架構上負責 OS 相關函式運作的 OSAL 層 (OS Abstraction Layer)來處理。其中 OSAL 層是建立於作業系統之上的一層介面函式，當模組所要執行的內容牽涉到作業系統的運作時，就會透過 OSAL 層的函式來與作業系統溝通。而模組中經由 OSAL 層來管理的內容包含了 Task 的運作、Task-ISR 同步機制、傳輸序列(Queue)、計時器、Interrupts 的順序、Semaphores、Mutexes……等。

而模組在資料處理上若有使用到硬體上資源，例如輸入端模組必須從硬體上來讀取影像的資料，則會藉由硬體的介面 DL 層(Device Layer)以及 BSL 層(Board Support Library)來設定或拿取硬體上的資訊。其中 DL 層的內容主要是與 DSP 晶片上的硬體環境相關，並藉由設定暫存器的值來操作硬體。而 BSL 層中的內容是對 DSP 晶片周邊的硬體環境來描述，當模組執行的內容牽涉到周邊硬體的相關設定時，例如輸入端與輸出端模組在訊號的傳輸上，必須取得連接於 DSP 晶片前後的硬體資訊，才能夠順利取得或傳輸訊號，而在這樣的情況下，模組就會透過 BSL 層的函式來連結到周圍硬體環境的資料庫，拿取硬體環境上相關的設定資訊。關於 DL 與 BSL 這部分詳細的內容會在第

四章裡繼續討論。

2.2 模組的資料結構

在本節中將會針對模組的 Capabilities 以及 Instance 兩大資料結構進行討論，說明這些資料結構在模組運作上所扮演的角色，並探討其組成的結構成員特性以及設定的參數內容。在 2.2.1 小節中，會先對 Capabilities 資料結構進行討論；2.2.2 小節則是對 Instance 資料結構的探討。

2.2.1 Capabilities 資料結構

Capabilities 是用來標示模組特性的資料結構，包括了標明模組身分的 ID、模組在傳輸上所支援的資料格式、以及在該模組的輸入與輸出端上可連接的模組數量……等，因此在建立模組的過程中，會先將 Capabilities 的結構參數值設定好，並將資料存放於模組的 AL 層中，而在需要用到 Capabilities 的設定資訊的狀況下，例如在設定模組的連接關係上必須要以模組的代號來做為連接的依據，以及判斷相連接的模組在傳輸上格式是否相容……等，應用程式就會透過函式到 AL 層中來拿取 Capabilities 的設定值。在 Capabilities 的結構上主要的結構成員為 Default 層所定義的 default Capabilities 資料結構，也是在每個模組的 Capabilities 結構上都會具備的部分，default Capabilities 的結構成員則如表 2.1 所示。

表 2.1 tsaDefaultCapabilities 結構

struct tsaDefaultCapabilities		
componentClass	numSupportedInstance	receiverFormatSetup
version	numCurrentInstance	olFuncs
capabilityFlags	numberOfInputs	inputPinCapabilities
textmemoryRequirement	inputFormats	outputPinCapabilities
datamemoryRequirement	numberOfOutputs	
processorRequirement	OutputFormats	

若模組在 default Capabilities 所定義的結構成員之外還有額外的結構參數需要設定，則會將這些成員附加於 Capabilities 結構上，例如圖 2.3 所示的 H.264 Encoder 模組，在 default Capabilities 結構外，另外定義了 H264_Profiles、H264_Levels、H264_Complexity-Modes……等的結構成員，提供模組在運作上所需的參數設定。


```

typedef struct _tmalVencH264_Capabilities_t
{
    // Default Capabilities 資料結構
    ptsaDefaultCapabilities_t pDefCap;
    //由 H.264 Encoder 模組所增加的 Capabilities 結構成員
    tmalVencH264_Profiles_t supportedProfiles;
    tmalVencH264_Levels_t supportedLevels;
    tmalVencH264_ComplexityModes_t supportedComplexityModes;
    tmalVencH264_BitrateControlModes_t supportedBitrateControlModes;
    tmalVencH264_BitStreamOutputModes_t supportedBitStreamOutputModes;
}tmalVencH264_Capabilities_t, *ptmalVencH264_Capabilities_t;

```

圖 2.3 Encoder 模組 Capabilities 資料結構

接下來就針對模組 Capabilities 資料結構上，主要的 default Capabilities 結構成員來進行討論。

- **componentClass**

架構上的所有模組都擁有自己的一組 ID，稱為 CID (Component Identifier)，會在模組建立的同時，到系統中管理各模組 ID 的資料庫中新增。

- **version**

由於在軟體的編譯上會不斷的修改程式內容並更新版本，因此在這部分利用到 majorVersion、minorVersion、buildVersion 等三個參數來標明模組的版本。

- **capabilityFlags**

旗標是用來標明模組的結構特性，例如模組的資料是透過快取記憶體 (cache) 傳遞給記憶體，或者直接對記憶體區塊來讀寫；或者該模組是否支援多種影像格式的輸入與輸出；以及模組在輸出端是否具備與多個模組連接的結構……等。而之後在設定模組的連接關係時，則會依據旗標來判斷所要執行的內容。

- **textmemoryRequirement、datamemoryRequirement**

這兩個結構成員為模組在設定上所需要使用到的記憶體空間大小。textmemoryRequirement 為模組的程式碼資料庫所使用到的記憶體大小，同時也代表在應用層中 include 一個模組所使用到的記憶體；datamemoryRequirement 則為模組在設定 Instance 資料結構時所使用到的記憶體大小。

- **numSupportedInstance**

模組中所支援的 Instance 數量。若模組有控制到硬體層結構，則會將該值設定為硬體的單位數量；而一般純軟體沒有牽涉到硬體結構的模組，該值則會設定為 1。

- **numCurrentInstance**

在程式執行中該模組所設定的 Instance 數量，因此在這部分會將初始值的設定為零，而之後會隨著程式的設定來做增減。

- **inputFormats、OutputFormats**

分別代表輸入以及輸出該模組的資料格式，而以下則列舉了主要幾個設定的資料格式：

- **dataClass**

標明傳輸資料的種類，例如影像訊號、聲音訊號或者是壓縮過後的資料封包。

- **dataType**

依據傳輸資料的種類再分類，例如影像訊號可再依照色彩的編碼方式分成 RGB 或 YUV……等。

- **dataSupType**

在經過 dataType 分類後，可依據影像畫面的取樣方式再進一步細部的區分，例如 YUV 色彩的取樣方式有 4:4:4、4:2:2、4:2:0……等，而取樣出來的 YUV 成分又可依據 Planar、SemiPlanar、Sequence……等方式來擺放。

- **description**

標明出影像畫面是以何種方式呈現，例如 Interlaced、Frame……等。

- **numberOfInputs、numberOfOutputs**

在模組與模組的連接上，有些模組可能會接收來自多個模組的輸出資料，或者將處理完的資料傳給多個模組接收。因此會由 numberOfInputs 來標明該模組在輸入端所支援與其它模組的連接數量；而 numberOfOutput 則代表模組的輸出端所支援與其它模組的連接數量。

2.2.2 Instance 資料結構

Instance 資料結構的內容主要是設定模組運作時必要的參數或函式連結，在一個模組的 Instance 結構中涵蓋了兩個部分，第一部分為 Instance 結構上提供給應用層來變更設定的 InstanceSetup 子結構，而在 InstanceSetup 結構上的組成也可再分為兩個部分，其一是由 Default 層所定義，提供給所有模組使用的 DefaultInstanceSetup 結構，其二則是模組在 Default 層所提供的 Setup 結構成員外，需要由應用層來設定的特定結構參數；而 Instance 結構的另一部分則是不需透過應用層來設定的參數，通常包括了各模組特定的一些變數，以及由 Default 層對模組在運作上所做的相關設定，Instance 的結構關係可參

照圖 2.4。

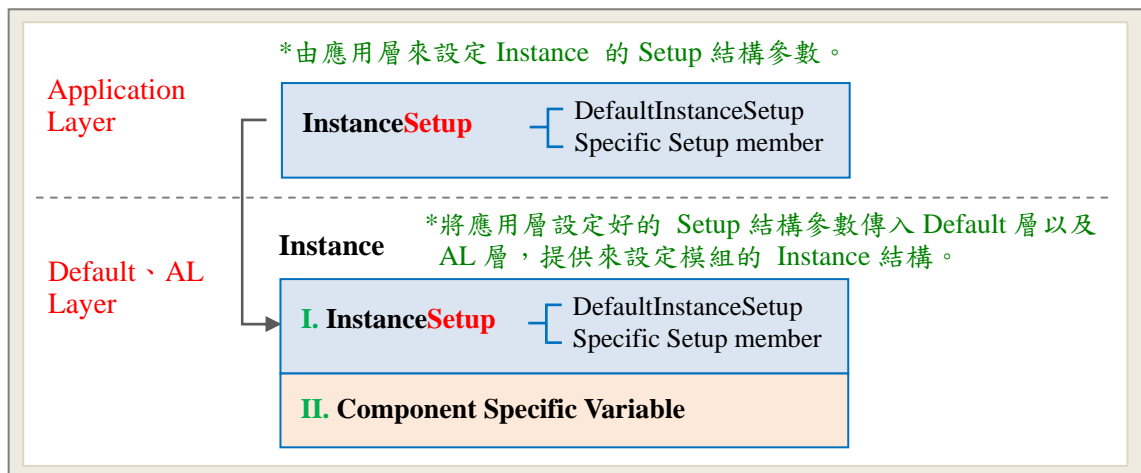


圖 2.4 Instance 的結構組成關係

以 H.264 Encoder 模組的 Instance 資料結構為例，如圖 2.5 所示，在該模組的 Instance 結構上分成了可由應用程式來設定的 Instance Setup 結構，以及由 Default 層與 AL 層來設定的結構參數兩部分。而在模組的 Instance Setup 結構這部分，除了由 Default 層所定義的 Instance Setup 結構外，還包括了模組自行定義的 Setup 結構成員，皆是用來提供給應用程式做參數上的設定。

```
typedef struct _tmalVencH264_InstVars_t
{
    ptmalVencH264_InstanceSetup_t pInstSetup; //模組所定義的 Instance Setup 結構
    //由 Default 層設定，應用於運作上的參數
    {
        tsaClockHandle_t * ClockHandle;
        tsaCompState_t componentState;
        tsaDatainFunc_t datainFunc;
        tsaDataoutFunc_t datainFunc;
        .....
    }
    //由模組自訂的 Instance 成員
    {
        tmVencH264_Input_t inputModule;
        tmVencH264_Output_t outputModule;
        tmVencCoreH264_FrameBuffer_t currentFb;
        tmVencCoreH264_BitStreamBuffer_t currentBs;
        .....
    }
};
```

//Default Instance Setup 結構

```
ptsaDefaultInstanceSetup_t pDefInstSetup;
//模組自訂的 Instance Setup 成員
Int8 slice_alpha_c0_offest_div2;
Int8 slice_beta_offest_div2;
UInt32 iframeInterval;
Bool useConstrainedIntraPredict;
Bool enableExportMV;
Bool frameSkipEnable;
.....
```

圖 2.5 Encoder 模組 Instance 資料結構

接下來就針對在每個模組上都具備的 Default Instance Setup 結構，其中幾個主要的結構成員來做介紹，Default Instance Setup 的資料結構如表 2.2 所示。

表 2.2 tsaDefaultInstanceSetup 結構

struct tsaDefaultInstanceSetup		
qualityLevel	startPinFunc	taskFlags
errorFunc	stopPinFunc	createNoTask
progressReprotFlags	clockHandle	taskStartArgument
progressFunc	*inputDescriptors	powerState
completionFunc	*outputDescriptors	debugFlags
datainFunc	parentId	normalMmsp
dataoutFunc	controlFunc	sharedCacheableMmsp
memalloc	controlDescriptor	sharedUncacheableMmsp
memfreefunc	priority	*inputConnections
getformatFunc	taskName[16]	*outputConnections
installFormatFunc	stackSize	periodOfComponent
tmaInstance	unitNumber	task

– **qualityLevel**

代表模組在執行中所能獲得處理器提供的品質高低，由數字 (1, 2, 3...) 來標示，數字越大代表從處理器上獲得的資源越多，而通常模組在這部分值皆設定為 0。

– **errorFunc**

當模組在串流的過程發生錯誤時所會呼叫的函式。而在 errorFunc 的函式中則會標示目前是在哪個模組中出錯的訊息，提供偵錯上的使用。

– **progressFunc**

當模組執行的程序到某個段落時所會呼叫的函式，此函式會依據模組傳入的旗標，標明出模組在串流資訊上所做的處理。

– **completionFunc**

當模組停止運作時所會呼叫的函式，並由函式標明出停止運作的模組名稱。

– **tmaInstance**

用來存放 Instance 結構在 AL 層的設定。由於 Default 層會在模組的 OL 層與 AL 層之間做連結與設定，而在層與層之間的設定過程上可能會修改到相同的結構參數，因此為了對 AL 層的設定值多一層保護，建立了 tmaInstance 結構，用來存放 Default

層與 AL 層之間傳遞的 Instance 設定參數。

– **datainFunc**

是負責處理模組輸入端資料傳輸的函式，而執行的內容主要是從模組輸入端所連接的連接模組(IOD¹)中拿取寫有資料的 packet，提供給模組來做運算處理，接著在模組處理完畢後將 packet 的資料清空再放回 IOD 中。由於每個模組在輸入端運作的模式都相同，因此在 Default 層中便定義一個 tsaDefaultDatainFunction() 函式，將上述的運作方式寫入此函式中，提供給所有模組使用。

– **dataoutFunc**

是負責處理模組輸出端資料傳輸的函式，而這部分資料傳輸的方式是先從輸出端的 IOD 中拿取空的 packet，接著將模組處理好的資料寫入此 packet 中再放到 IOD 上。與上述的 datainFunc 相同，在 Default 層中也定義了一個處理資料輸出的函式 tsaDefaultDataoutFunction()，提供給所有模組使用。

– **starPinFunc**

在模組啟動後，會先將該模組輸入端與輸出端的 pin 腳皆設定為啟動的狀態，也就是讓輸入端與輸出端可以開始傳輸資料。而在 Default 層中有定義了執行此功能的函式 tsaDefaultStartPin()，提供給所有的模組來使用。

– **stopPinFunc**

當模組的運作結束時，會將模組輸入端與輸出端的 pin 腳設定回停止的狀態，讓模組停止對外的資料傳輸。而同樣的在 Default 層中也提供了執行此功能的函式 tsaDefaultStopPin() 來給所有模組使用。

– **inputDescriptors**

於模組輸入端所銜接的連接模組。當 IOD 建立完成後，會將此建立好的 IOD 結構指標設定到模組的 inputDescriptors 中，讓模組取得輸入端 IOD 的資訊，使得在傳輸資料的時候，模組可以辨認出要至哪個 IOD 中拿取。

– **outputDescriptors**

於模組輸出端所銜接的連接模組。設定方法與上述的 inputDescriptors 相同，此 outputDescriptors 則是讓模組得知要將處理完畢的資料寫到哪個 IOD 中。

¹ IOD 為 TSSA 架構上另一個型態的模組 Connection Component，當兩個影像處理模組之間要傳輸資料時，就會利用此連接模組來協助兩者資料的讀寫與傳輸，由於在連接模組上會描述影像處理模組之間資料輸入與輸出的特性與設定，因此連接模組又可稱為 IOD (In-Out Descriptors)，關於 IOD 詳細的內容會在第三章進行介紹。

- **priority**
模組執行緒的優先順序，是由應用層設定，提供給作業系統來操作模組之間在執行運作上的順序。
- **taskName[16]**
模組執行緒的名稱，是由應用層所設定，當 Default 層在建立模組的執行緒時，可將 taskName 傳入執行緒的建立函式提供命名使用。
- **taskFlag**
標明執行緒 task 的執行狀態，例如已經啟動的 task，在 task 的旗標上就會標明為“tmosalTaskStarted”，而正在等待記數中的 task，在 task 旗標上則會標明為“tmosalTaskCounting”。
- **task**
當模組透過 Default 層到 OSAL 層中建立好執行緒後，便可將執行緒的指標回傳設定到 task 中，讓每個模組的 Instance 結構裡都能具備自己的執行緒資料。

而在了解上述這些資料結構成員後，接下來所要討論的重點為模組如何拿取並設定這些資料結構，以及模組從結構設定到執行運作的流程上所使用到的相關函式。



2.3 模組的函式功能

在 TSSA 架構中為每個模組定義了八個在運作上的相關函式，分別負責處理模組中不同的程序設定，而這八個函式列舉如下：

- tm <Layer> <Component> GetCapabilities()
- tm <Layer> <Component> Open()
- tm <Layer> <Component> GetInstanceSetup()
- tm <Layer> <Component> InstanceSetup()
- tm <Layer> <Component> Start()
- tm <Layer> <Component> Stop()
- tm <Layer> <Component> Close()
- tm <Layer> <Component> InstanceConfig()

在每個模組中的 OL 層以及 AL 層上都會具備這些函式，因此在函式名稱上，<Layer> 代表函式所在的層面，<Component> 則代表模組的名稱。而由於 Default 層是負責模組 OL 層與 AL 層的連結，因此在 Default 層中也具備了這八個函式，並以 <tsaDefault> 做為函式名稱的開頭，例如 tsaDefaultGetCapabilities()、tsaDefaultOpen()……等。接下來就針對這八個函式的執行內容來進行討論。

1. GetCapabilities()

由於 Capabilities 資料結構中儲存了模組的結構特性以及標示模組的代號，因此在模組的執行程序上，首先必須取得 Capabilities 的資料設定。而 Capabilities 結構參數的值設定於模組的 AL 層中，因此應用層會透過 OL 層以及 Default 層的連結來向 AL 層取得 Capabilities 的設定。函式的流程如圖 2.6 所示，首先在應用層上會呼叫模組 OL 層的 GetCapabilities() 函式，並傳入應用層中存放 Capabilities 結構的指標；接下來在 OL 層中，會呼叫 Default 層的 GetCapabilities() 函式，將該模組 AL 層的函式指標位址與存放 Capabilities 的指標一併傳入到 Default 層中，讓 Default 層取得 AL 層運作的函式位址；最後，就由 Default 層連結到該模組 AL 層的 GetCapabilities() 函式，並在該函式中，將存放 Capabilities 資料結構的位址設定到由上層傳入用來取得 Capabilities 設定的變數中，完成 Capabilities 的設定。



圖 2.6 設定 Capabilities 的函式流程

2. Open()

Open() 函式主要的功能是分配記憶體空間給 Instance 資料結構。由於接下來所要設定的部分為模組的 Instance 結構，因此在各層的 Open() 函式裡會先分配記憶體空間給 Instance 以及 Instance 的 Setup 資料結構，提供給模組一個新的設定環境，並且讓模組在資料結構的設定上能夠避免被其它的模組修改到，也是對模組在設定上多一層的保護。

```
tmalComponentOpen()  
{ //依照 Instance 結構的 Size，分配記憶體空間給 Instance 變數  
  ivp = tmDefault_Calloc (sizeof (ComponentInstance_t);  
  //依照 InstanceSetup 結構的 Size，分配記憶體空間給 Instance Setup 變數  
  ivp->pSetup = tmDefault_Calloc (sizeof (ComponentInstanceSetup_t));  
  .....
```

圖 2.7 Open() 函式的執行內容

圖 2.7 為 AL 層 Open() 函式的執行內容，由於各層 Open() 函式執行的內容類似，這部分便以 AL 層為例。在 Open() 函式中，主要是藉由 TSSA 中所定義分配記憶體的函式 tmDefault_Calloc()，依據 Instance 以及 InstanceSetup 資料結構的大小來分配記憶給模組的變數。其中在 Default 層的 Open() 函式裡，在做好記憶體區塊分配後，還會針對 Instance 結構中部分的結構成員做初始值的設定。

3. GetInstanceSetup()

GetInstanceSetup() 函式是去取得模組的 Instance Setup 結構，提供給應用層來設定。而在取得模組 Instance Setup 結構的流程上，與前面所描述 GetCapabilities() 函式在取得 Capabilities 結構的方法類似，都是先藉由 OL 層的函式將待設定的結構指標以及 AL 層的函式位址傳入 Default 層中，再由 Default 層連結到 AL 層的函式上，取得資料結構的位址。而應用層在獲得模組的 Instance Setup 結構時，便可針對模組在運作上的需求來設定結構參數，而除了模組本身所具備特定的參數須要設定外，在每個模組中都會設定到的結構成員包括了 errorFunc、progressFunc、completeFunc 等模組在運作上用來標示以及回報狀態的函式，模組的執行緒在作業系統上的優先順訊 priority，模組輸入端所連接的傳輸模組 inputDescriptors，模組輸出端所連接的傳輸模組 outputDescriptors……等。

4. InstanceSetup()

而當應用層設定完上述的 Instance Setup 結構後，便透過 InstanceSetup() 函式將設定好的 Setup 結構傳入下層，而在 Default 層以及 AL 層中則會將這些 Setup 結構的參數一一設定到 Instance 結構上，完成模組 Instance 資料結構的設定。

5. Start()

當每個模組的資料結構參數都設定完成之後，便可透過 Start() 函式來啟動模組的運做。模組啟動的流程同樣是透過 OL 層介面函式，連結到 Default 系統函式，並於 Default 層中建立以及啟動模組的執行緒，而模組主要的運算處理函式則會在系統啟動執行緒時，一併傳入給 psos 系統，讓系統拿到一個啟動狀態的執行緒以及在此執行緒上所要執行的函式。系統上的函式的流程如圖 2.8 所示，當 Default 層收到啟動模組的指令後，首先 Default 層會呼叫 OSAL 層執行緒的建立函式 tmosalTaskCreate()，判斷該模組是否具備建立好的執行緒，若模組的執行緒尚未建立，則會藉由 psos 的系統函式 t_create() 為模組建立一個執行緒。當模組的執行緒建立完成後，就會透過 psos 的系統函式 t_start() 來啟動執行緒。

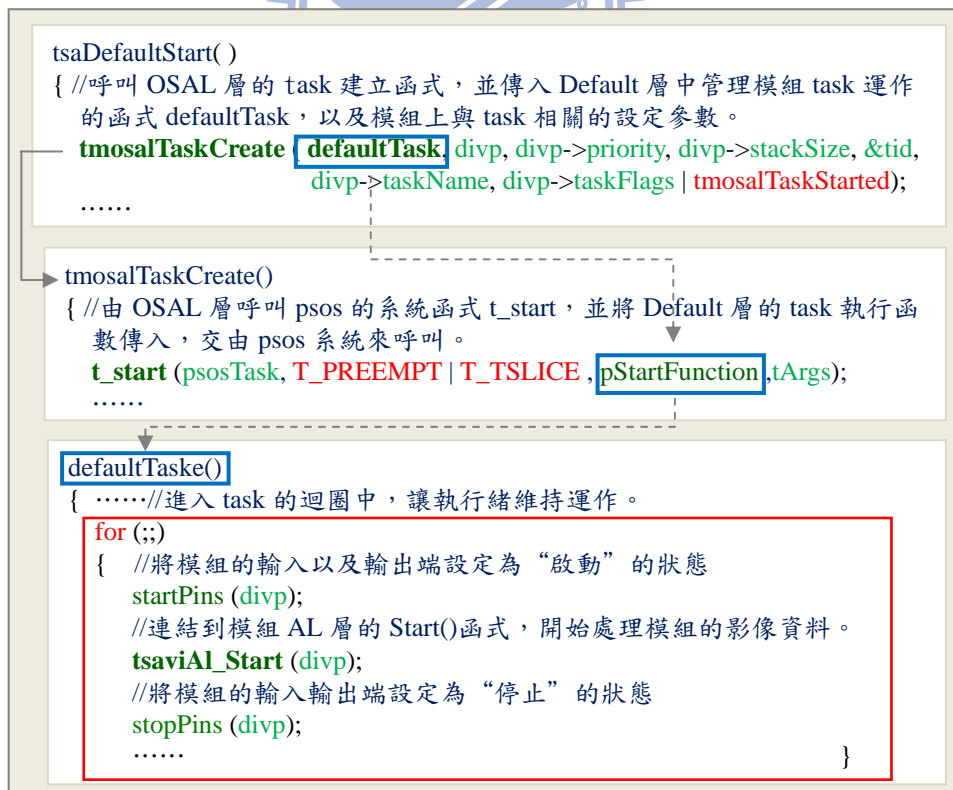


圖 2.8 Default 層 Start() 函式執行流程

而在 OSAL 層呼叫 psos 系統函式 `t_start()` 執行的同時，會傳入一個 Default 層上所定義的 root function 稱為 `defaultTask()`，在這個函式裡將模組運作上所會執行的內容寫在一個迴圈裡，包含了一開始要先將模組的輸入端與輸出端 pin 腳設定為啟動的狀態，接著再呼叫該模組 AL 層上的執行運作函式，進入到模組影像處理的程式部分。而在啟動模組之後，必須讓模組的函式保持在運作的狀態，才能不斷處理輸入的串流資料。因此在每個模組 AL 層 Start 函式的執行程序上，會定義一個無限迴圈，將模組從拿取影像資料、運算處理、到儲存處理好的影像資料的流程，都寫在這個迴圈裡，若模組一直保持在啟動的執行狀態，就會不斷的在這個迴圈裡執行運算，直到收到停止的指令才會停止運算並跳出迴圈，圖 2.9 為 ProcessInPlace 模組 AL 層 Start() 函式的執行內容。當應用程式啟動所有的模組之後，系統上會取得每一個模組的執行緒，以及執行緒上所要運作的函式，而每個執行緒這時候都會保持在啟動的狀態，由系統依照執行緒的優先權來分配運作的先後順序。

```

tmalProcessInPlaceStart(Int instance)
{..... //進入模組的運算迴圈，讓模組能夠不斷的執行運算
while (ivp->componentState == tsaCompStateRunning)
{ //呼叫 datain function 讀取輸入影像資料
  tsaviStreaming_DataIn ();
  //呼叫模組的影像處理函式來運算資料
  modifyPacket (ivp->handle, ivp->packet);
  //呼叫 dataout function 輸出模組處理完的影像資料
  tsaviStreaming_DataOut ();
  ..... }

```

圖 2.9 模組 AL 層上 Start 函式的內容

6. InstanceConfig()

當所有模組皆開始運作後，某些應用程式或許會提供一些變更影像畫面的功能給使用者來選擇，例如暫停影像畫面、取消影像 deinterlaced 的功能、或停止程式的運作……等選項，而在使用者下達指令後，應用層就會透過 OL 層的 InstanceConfig() 函式將變更的部分傳入下層，並於 Default 層以及 AL 層的 InstanceConfig() 函式中去更改 Instance 資料結構的設定。

7. Stop()

暫停模組中所有處理的程序，並變更標明模組狀態的旗標設定，將原本為“Running”或者“Requesting”的狀態更改為“Stop”以及“Stop_Requesting”。而模組會在暫停的狀態上停留，直到模組的 Start() 函式再度被呼叫時，才會恢復模組運作。

8. Close()

結束整個模組的運作，並藉由 TSSA 中所定義釋放記憶體之函式 `tmDefault_Free()`，將前面在 `Open()` 以及 `Start()` 函式中所分配的記憶體空間釋放掉。

一個標準的 TSSA 軟體模組架構，就是藉由 Capabilities 以及 Instance 這兩大資料結構在上述的這些函式中，搭配模組的分層機制來設定以及運作而構成的。

2.4 TSSA 模組應用

當一個影像處理模組具備了上述的資料結構與函式功能後，即可將模組儲存於 TSSA 的模組資料庫中，當應用程式在執行上需要使用到該影像處理功能時，就可以直接到此資料庫中來拿取。而在應用程式拿取模組來使用的流程上，首先必須在執行程式的 `makefile` 檔裡宣告需要的模組名稱以及函式資料庫名稱，接下來在應用層的程式上則要對所使用的模組進行宣告，如圖 2.10 所示。

```
struct exSvipInstance // Global Structure
{
    .....
    /*----- tmVdecAna -----*/ //A/D 晶片的操作模組
    Int hVdecAna;
    tmbslVdecAnaVideoInSources_t VISources;
    .....
    /*----- tmVcapSvip -----*/ //影像擷取模組
    Int hVcapSvipUnit;
    Int hVcapSvipStream[MAX_NUMBER_STREAMS];
    ptmolVcapSvipUnit_Capabilities_t pVcapSvipUnitCap;
    ptmolVcapSvipUnit_InstanceSetup_t pVcapSvipUnitSetup;
    ptmolVcapSvipStream_Capabilities_t pVcapSvipStreamCap[MAX_NUMBER_STREAMS];
    ptmolVcapSvipStream_InstanceSetup_t pVcapSvipStreamSetup[MAX_NUMBER_STREAMS];
    /*----- tmVideoProc -----*/ //影像處理模組
    Int videoProc;
    ptmolProcessInPlaceCapabilities_t videoProcCap;
    ptmolProcessInPlaceInstanceSetup_t videoProcSetup;
    .....
    /*----- tmVrendGfxVo -----*/ //影像輸出模組
    .....
    /*----- tmVencAna -----*/ // D/A 晶片的操作模組
    .....
    /*----- IO Descriptors -----*/ //IOD 相關宣告
    ptsaInOutDescriptor_t vcapSvipOd;
    ptsaInOutDescriptor_t videoProcOd;
    .....
}
```

圖 2.10 應用層程式全域變數的資料結構

在應用程式的一開始，會宣告一個全域的結構變數（Global Structure），定義在此程式中所會使用到的結構變數，而在該資料結構上，則會宣告此應用程式裡所有影像處理模組以及傳輸模組 IOD 的變數。其中對於影像處理模組的宣告內容，包含了用來描述模組特性的資料結構 Capabilities，以及維持模組運作所要設定的 Instance 資料結構，和 Instance 結構中提供給應用程式來設定的 InstanceSetup 資料結構；而傳輸模組 IOD 的宣告內容，則是用來設定 IOD 架構的 Setup 資料結構。而在應用程式一開始所宣告的全域資料結構中，每個結構成員的內容都還是空的，尚未設定任何的參數值，因此應用程式在接下來所要執行的程序，就會利用到在 2.3 節裡所提到的模組函式，來取得各模組中的結構參數預設值，並且對尚未有值的資料結構來做參數設定。而當各影像處理模組都取得設定值，同時傳輸模組也建立完成之後，此應用程式即可開始執行運作。

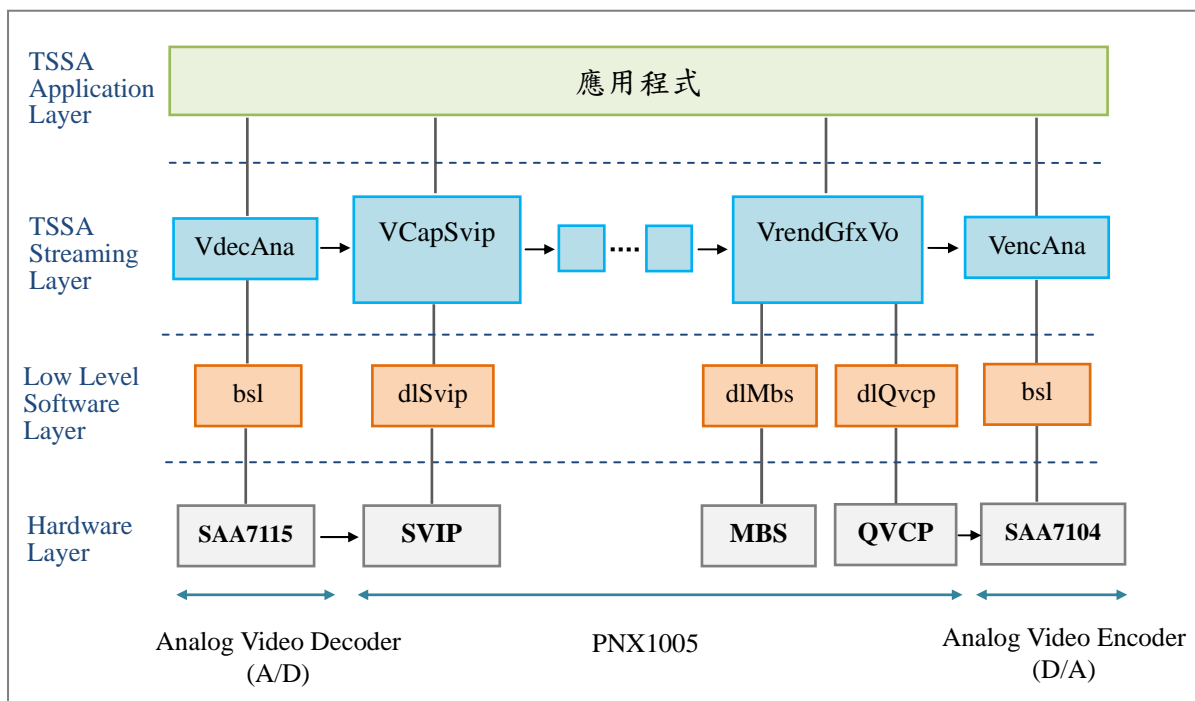


圖 2.11 TSSA 模組與分層架構圖

而整個應用程式會以 TSSA 的架構來建立於 DSP 晶片上執行。由使用者的操作層面至硬體環境之間，TSSA 架構會再以分層化（Layer）的模式，由上而下分成四個層面，如圖 2.11 的模組分層架構所示。在最上層的 TSSA Application Layer 即為所要執行的應用程式，也是使用者所操作編譯的部分。而第二層 TSSA Streaming Layer，為較高階的軟體層，負責影像處理以及資料串流的功能，也是在前面幾個章節所描述的軟體模組主要建構的層面。而圖 2.11 中該層面上的每一個方塊，代表的就是一個影像處理模組，而

此圖是以一個處理即時影像的應用程式為例，使用到的模組包括了操作 A/D 訊號轉換晶片的模組 tmVdecAna (trimedia Video Decode Analog)、將影像資訊從硬體上擷取到軟體層的輸入端模組 tmVcapSvip (trimedia Video Capture Svip)、連接於輸入端與輸出端模組之間的數個影像處理模組、將處理完的影像資訊傳入到硬體層輸出的影像輸出模組 tmVrenderGfxVo (trimedia Video Renderer)、以及操作 D/A 訊號轉換晶片的模組 tmVencAna (trimedia Video Encode Analog)。這些獨立的模組，透過傳輸模組的連接因而能夠互相溝通並且傳遞資料，至於傳輸模組的資料串流方式則會在第三章裡做詳細的探討。

接下來第三層為較低階的軟體層，是提供給上層的軟體模組來操作硬體環境的一層介面，提供的資訊包括了驅動 DSP 晶片 PNX1005 所需的 DL 層函式資料庫、以及支援 DSP 晶片周圍硬體環境的 BSL 層函式資料庫連結；而在分層結構中的最底層，即為上述這些軟體程式運作的硬體環境，在硬體環境上主要包含了三個部分，分別為 DSP 晶片 PNX1005，DSP 晶片輸入端所連結的類比轉換數位(Analog to Digital, A/D)晶片 SAA7115，以及連結於 DSP 晶片輸出端的數位轉換類比(Digital to Analog, D/A)晶片 SAA7104。其中在 PNX1005 上，還具備了多個硬體單元來處理影像訊號，例如負責影像輸入的 SVIP (Shared Video Input Processor) 單元，以及調整影像畫面的 MBS (Memory Scale Based) 單元，與負責影像輸出的 QVCP (Quality Video Composition Processor) 單元，這些硬體環境的運用則會在第四章裡做進一步的探討與說明。

第三章 傳輸模組 IOD

本章所要探討的內容為 TSSA 架構上的傳輸模組 IOD (In-Out Descriptor)，由於 IOD 是用來連接兩個影像處理模組，因此在與模組之間的連接設定以及資料的傳輸流程都是在本章所要討論的重點。而在章節的順序上，首先 3.1 節會先針對 IOD 的架構進行討論；接下來在 3.2 節中，則會討論 IOD 的建立流程，包含了 IOD 的資料結構設定以及 IOD 相關函式的執行內容；最後在 3.3 節中，則會討論在影像處理模組上如何藉由 IOD 來傳輸資料。

3.1 IOD 架構概念

IOD 為 TSSA 架構上為連接模組(Connection Component)，是做為影像處理模組 (Active Component)之間傳輸資料的橋梁。IOD 實際上為一個特定的記憶體區塊，是由應用層在設定模組間的連接關係時所建立的，每兩個相連接的影像處理模組之間都會有一個特定的 IOD，當前一級的模組處理完影像後，便會把資料寫入所連接的 IOD 中，而下一級的模組則會到此 IOD 中拿取資料，在連接上的架構如圖 3.1 所示。

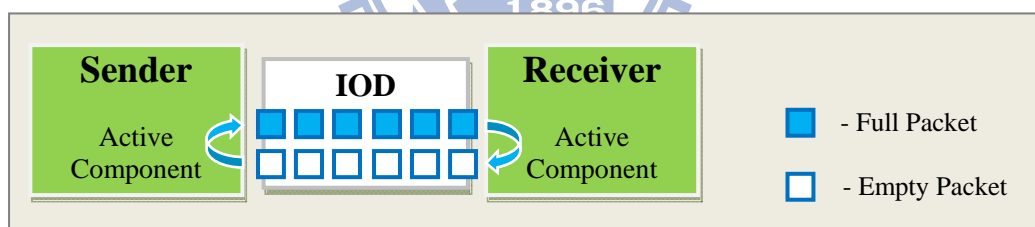


圖 3.1 IOD 與影像處理模組之間的關係架構

模組之間資料的傳輸是以封包(packet)為單位，一個 packet 通常用來傳送一張影像畫面，而一個 packet 可具備一到多個 buffer，用來存放影像中不同的成分，例如同一張影像的亮度(luminance)與彩度(chrominance)的資料就會分別放在不同的 buffer 裡來傳送，packet 的這部分詳細的格式設定會在 3.2.2 小節中進一步討論。當模組之間要傳輸資料時，如圖 3.1 中的傳送端(Sender)會至 IOD 的 packet 序列中拿取空的 packet 來寫入處理完畢的影像資料；而接收端(Receiver)在需要影像資料時，則會到 IOD 的 packet 序列中拿取填有資料的 packet 來處理，當接收端處理完資料後，則會將此 packet 所傳送的影像資訊清空，把空的 packet 放回 IOD 序列中，提供給傳送端模組寫入資料，形成

一個循環式的傳輸方式。

3.2 IOD 的建立流程

由於 IOD 是用來串連兩個影像處理模組，因此當應用層在設定模組之間的連接關係時，就會開始針對兩兩模組之間的 IOD 設定特定的結構參數，並且會直接呼叫 Default 層的函式來建立 IOD。在本節中，將會以程式上執行的流程來探討 IOD 的建立過程。首先在 3.2.1 小節中，會先針對 IOD 的結構參數進行討論，並了解如何設定兩個模組之間專屬的 IOD；接下來在 3.2.2 小節中，則會探討在 Default 層中的函式如何建立 IOD，以及在該函式中對結構參數所做的設定；最後 3.2.3 小節，則是討論如何將建立好的 IOD 設定到 IOD 前後所連接的兩個模組中。

3.2.1 IOD 結構參數

當應用層要建立兩個模組之間的 IOD 時，首先必須將 IOD 前後所連接的模組資料結構設定到 IOD 的結構參數中，讓 IOD 能夠確認前後模組的連接關係；而另一方面，必須讓前後兩個模組也能夠辨認出之間所連接的 IOD，因此在 IOD 建立完成之後，會將此 IOD 的指標設定到模組的 Instance 結構參數中，讓模組取得 IOD 的資料，使得雙方都能夠確認連接的關係。

由於 IOD 實質上是由 Default 層來建立的，因此在 Default 層中定義了建立一個 IOD 所必須具備的資料結構，用來設定每個 IOD 特定的條件。其中，在這個資料結構上，部分的結構參數要在應用層上設定，因此 Default 層便將這些參數成員獨立出來，定義了另一個 IOD 的 Setup 資料結構，提供給應用層來設定。其 IOD 結構與 IOD Setup 結構在應用層與 Default 層上的設定關係如圖 3.2 所示。

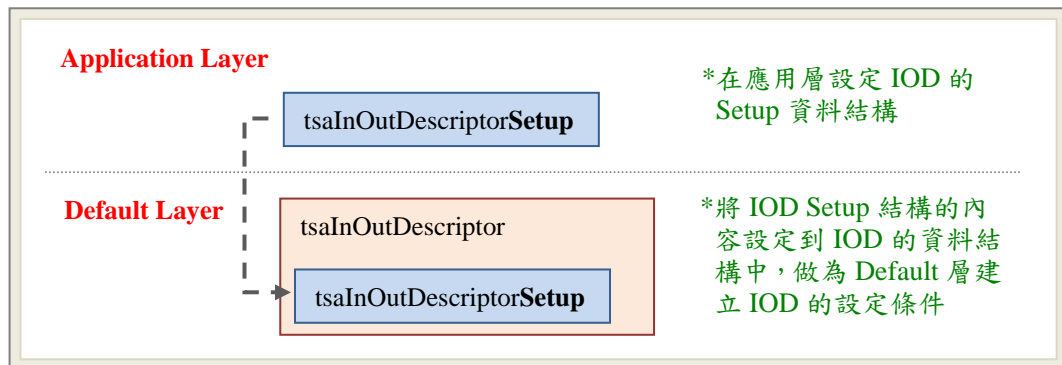


圖 3.2 IOD 資料結構在分層上的設定關係

在應用層開始設定 IOD 的 Setup 結構參數之前，首先必須取得前後兩個模組的代號以及處理的影像格式等資訊，而這部分的資訊主要設定在模組的 Capabilities 結構中，因此應用層會先藉由模組的 GeaCapabilities() 函式，取得模組的 Capabilities 資料結構設定，再開始建立模組之間傳輸的 IOD。IOD 的 Setup 資料結構成員如表 3.1 所示，在應用層上會以模組之間的傳輸條件來設定結構內容，接下來就以 Setup 結構中幾個主要的結構成員來做說明。

表 3.1 tsaInOutDescriptorSetup 結構

struct tsaInOutDescriptorSetup		
format	receiverCap	numofProperties
flags	senderIndex	propertySetups
fullQName	receiverIndex	connectionProxy
emptyQName	bufferMem	numberOfBuffers
queueFlags	bufferProperty	bufSize[N]
queueSize	packetBase	
senderCap	numberOfPackets	

- **format**

標明在這個 IOD 上傳輸的資料格式，包含了目前所傳輸的資料類型為影像、音訊或其他形態的訊號；若為影像資料，則會標明出影像的格式，例如 YUV、RGB、JPEG、MPRG……等；接下來則更進一步標示出影像的取樣格式，例如 YUV 的影像有 4:4:4、4:2:2、4:2:0……等多種的取樣格式，以及影像的取樣內容是以 sequence、planar 或其他的方式排列擺放；此外，還會標示出目前每筆傳輸資料的大小，例如影像畫面的長寬資訊。

- **flags**

旗標是用來標明此 IOD 的一些結構特性，例如標明此 IOD 是否具備 Tee¹ 的結構、IOD 的 buffer 在記憶體空間上的分配是否要與快取記憶體(cache)區塊對齊、或是要再額外配置一個記憶體空間來放置 IOD 的 buffer……等。而 Default 層在設定 IOD 結構參數時，則會依據旗標來判斷 IOD 所要設定的參數內容。

¹ 當有多個模組要接收來自同一個模組的輸出資料時，可透過建立 Tee 來複製資料。Tee 的結構就像樹木一樣，具有主幹和分支(trunk & branch)，在程式的執行上會選定一個建立好的 IOD 當作主幹，其餘做為分支的 IOD 則會複製主幹上的資料來傳輸。

- **fullQName**

當 Default 層在建立 IOD 的過程中，會建立兩個序列，一個為 fullQueue，排列 IOD 中填滿資料的 packet；另一個序列為 emptyQueue，排列 IOD 中空的 packet。因此在序列的建立過程上，會需要一個序列名稱來標示，fullQName 即為 fullQueue 序列的名稱。

- **emptyQName**

如同上述，emptyQName 即為 emptyQueue 序列的名稱。

- **packetBase**

在 IOD 上傳輸的每個 packet 都需要一個 id 編號，做為 packet 傳輸上的辨別，而 packetBase 則是標明此 IOD 中 packet 的起始編號，例如連接輸入端模組 tmVcapSvip IOD 的 packetBase 為 0x100，則此 IOD 的 packet 編號會從 256 開始算起，接下來為 257、258、259……逐一增加。

- **senderCap**

將 IOD 傳送端模組所取得的 Capabilities 資料結構設定到 senderCap 上，讓 IOD 取得傳送端模組的 id 以及影像格式等相關資料。

- **senderIndex**

由於部分的模組在結構上會具備多支輸出 pin 腳，可以將處理好的影像資料往下傳送給多個模組，因此在 IOD 的 senderIndex 上會標明要從傳送端模組的那一支 pin 腳來取得影像。

- **receiverCap**

將 IOD 接收端模組所取得的 Capabilities 資料結構設定到 receiverCap 上，讓 IOD 取得接收端模組的 id 以及影像格式等相關資料。

- **receiverIndex**

用來標明 IOD 的接收端模組要以哪一支 pin 腳來接收資料。大部分模組在結構上都只具備一個輸入的 pin 腳，只有在少部分的模組中(例如 Render)具備多個輸入 pin 腳，可以接收來自不同 IOD 的資料，因此在 IOD 的設定上還是必須標明接收端的 pin 腳資訊，做為模組連接上的一個確認。

- **numberOfPackets**

在 IOD 上傳送的 packet 數量。由於每個模組在影像處理上的速度並不一致，因此在存取 packet 的速度上也會不同，應用層則可以依據模組的特性來調整在 IOD 上傳輸的 packet 數量，讓影像最後能夠流暢的輸出。

- **numberOfBuffers**

在每個 packet 中所具備的 buffer 數量。Buffer 的數量是依據影像的取樣以及排列格式來設定的，例如傳輸 YUV422SemiPlanar 的影像，則代表影像的取樣方式為 4:2:2，並將 Y 和 UV 放置到兩個不同的 plane，因此需要兩個 buffer 來存放這兩個不同的成分；若影像經過壓縮的處理，則在 packet 上只需要設定一個 buffer 來存放資料。

- **bufSize[N]**

定義 packet 中每個 buffer 的大小。

當應用層將以上的 Setup 結構參數設定完畢之後，會把此設定好的 Setup 結構傳入 Default 層中，提供給 Default 層建立 IOD 使用。而 Default 層在拿到 Setup 結構後，會將 Setup 結構中的參數設定到 IOD 的資料結構中，並依照 Setup 結構中設定的內容來建立 IOD。表 3.2 為 IOD 的資料結構成員，其中表格的第三欄即為和 Setup 資料結構一致的部分，Default 層會根據這部分結構成員所設定的參數來設定其它結構成員的值，接下來的 3.2.2 小節就開始探討 IOD 在 Default 層上的建立流程。

表 3.2 tsaInOutDescriptor 結構

struct tsaInOutDescriptor		
senderState	receiverStopped	format
receiverState	cmdFullWakeupSent	flags
fullQueue	cmdEmptyWakeupSent	senderCap
emptyQueue	mmspFlags	receiverCap
*packetArray	sleepOnStop	senderIndex
fullsize	clock	receiverIndex
emptysize	connectionProxy	packetBase
lastFormat	numofFullPackets	numberOfpackets
waitsemaphore	numofEmptyPackets	bufferMem
bTee	maxNumofFullPackets	bufferProperty
bCopyData	maxInUse	queuesize
ioTeeing	maxPacketInUse	fullQueueName[16]
parent	Mutex	emptyQueueName[16]
mmsp	PendingPacket	

3.2.2 Default層上IOD的建立

當應用層設定好 Setup 結構參數後，會藉由 Default 層建立 IOD 的函式 `tsaDefaultInOutDescriptorCreate()` 將 Setup 結構傳入 Default 層中，並開始執行該函式的內容。在 `tsaDefaultInOutDescriptorCreate()` 函式中，首先會將傳入的 Setup 結構參數複製到 IOD 結構中，如圖 3.3 所示。

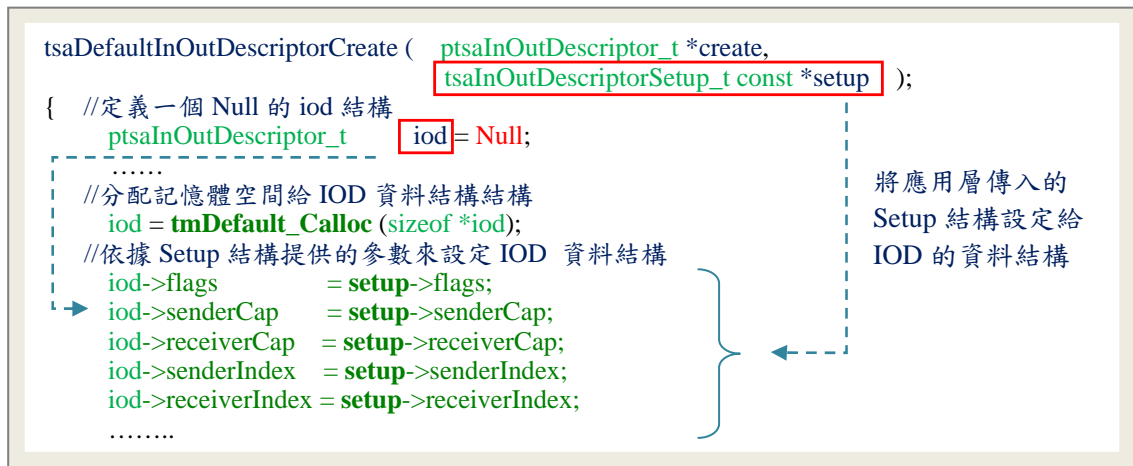


圖 3.3 複製 Setup 結構內容到 IOD 結構流程

當 Default 層取得 Setup 的設定之後，會將這些設定值傳入到下列的函式裡，藉由執行這些函式來設定 IOD 結構中待設定的成員，而接下來則會針對這些函式執行的內容一一探討。

- `setInOutDescriptorFlags()`
- `tsaDefaultCheckCapabilitiesFormat()`
- `tsaDefaultInstallFormat()`
- `tmGeneralMutex_create()`
- `createQueue()`
- `tmPacket_create()`
- Packet 與 Queue 的設定：`tmPacket_SetBufferSize()`、`tmPacket_SetBufferAddr()`、`tmosalQueueSend()`。

1. `setInOutDescriptorFlags()`

在這部分會依據 IOD 的傳送端以及接收端模組在 Capabilities 結構上旗標的設定，來修改 IOD 結構中的旗標。首先，會確認傳送端以及接收端模組是否具備 Tee 的結構，

若兩邊模組皆不具備的話，則會在 IOD 的旗標上加入(or, ||)不支援 Tee 結構的旗標。而接下來會確認接收端模組的旗標中是否有包含“tsaCapFlagCopybackDatain”，此旗標代表模組會直接向記憶體來拿取資料，而不用透過 cache 間接來跟記憶體取得資料。若接收端模組的旗標中包含了這個旗標，則會在 IOD 的旗標中加入“tsaIODescSetupFlagCopybackDatain”的旗標，標明出此接收端模組特性。在檢視完接收端模組後，會接著確認傳送端模組的旗標設定，若傳送端模組的旗標設定中包含了“tsaCapFlagsInvalidateDataout”，表示傳送端的資料是直接寫入記憶體中，不會透過 cache 來間接存取，而在 IOD 的旗標設定上會加入“tsaIODescSetupInvalidateDataout”旗標，標明出此傳送端模組的特性。

在接下來要討論的 `tsaDefaultInstallFormat()` 以及 `tsaDefaultInstallFormat()` 兩個函式裡，主要是針對 IOD 結構中的 `format` 來做設定。當應用層在設定 Setup 結構時，若有在 Setup 的結構成員 `format` 中設定特定的格式，則 Default 層在將 Setup 的 `format` 內容設定到 IOD 結構中之前，會先針對設定的格式和傳送端模組以及接收端模組的格式設定做比對，確認模組在資料傳輸上格式的相容。

2. `tsaDefaultCheckCapabilitiesFormat()`

在本函式裡，會比對 Setup 結構上的 `format` 設定與傳送端、接收端模組的格式設定是否相符合，其中傳送端與接收端模組的格式內容是設定在 `Capabilities` 結構裡，因此這部分會依序將傳送端模組的 `Capabilities` 結構以及 `format` 設定、接收端模組的 `Capabilities` 結構以及 `format` 設定傳入比對格式的函式 `tmFormat_Negotiate()` 中處理。在 `tmFormat_Negotiate()` 函式中，主要判斷的格式設定包含了 `dataClass`、`dataType`、`dataSubtype`，若 `format` 與 `Capabilities` 結構在這幾個參數上的設定不一致，則函式會回傳錯誤的訊息給 Default 層，讓 Default 層停止 IOD 的建立。

3. `tsaDefaultInstallFormat()`

當上述函式 `tsaDefaultCheckCapabilitiesFormat()` 執行完格式的比對後，若 Setup 結構中的 `format` 設定與傳送接收端模組的格式相符合，則在本函式中，會將 Setup 結構的 `format` 設定到 IOD 結構裡，完成 IOD 的 `format` 設定。

4. tmGeneralMutex_create()

建立 IOD 的 Mutex。如同影像處理模組建立 Mutex 的過程，在這部分，做為連接模組的 IOD 也會在 psos 上建立一個自己的 Mutex，提供給系統執行操作。

5. createQueue()

在這個函式裡，會來建立 IOD 中傳送 packet 的序列：full queue、empty queue。而序列是由 Default 層透過 OSAL 層的函式來跟系統要求建立，因此在執行的程序上，如圖 3.4 所示，首先 Default 層會將序列的設定資訊傳入到 OSAL 層的 tmosalQueueNameCreate() 函式裡，而這些傳入的參數包含了序列名稱、序列的大小、序列的旗標以及序列編號，其中序列編號是一個待設值，會透過接下來要介紹的幾個 OSAL 層函式來取得設定。當 OSAL 層取得這些設定後，會再將參數傳入 spOsalQueueNameCreate() 函式裡來執行建立序列的動作，這麼做的原因是為了因應不具備序列名稱的序列，若目前所要建立的是一個未命名的序列，則會透過 OSAL 層的 tmosalQueueCreate() 函式將序列名稱設定為 Null 再傳入 spOsalQueueNameCreate() 函式裡。

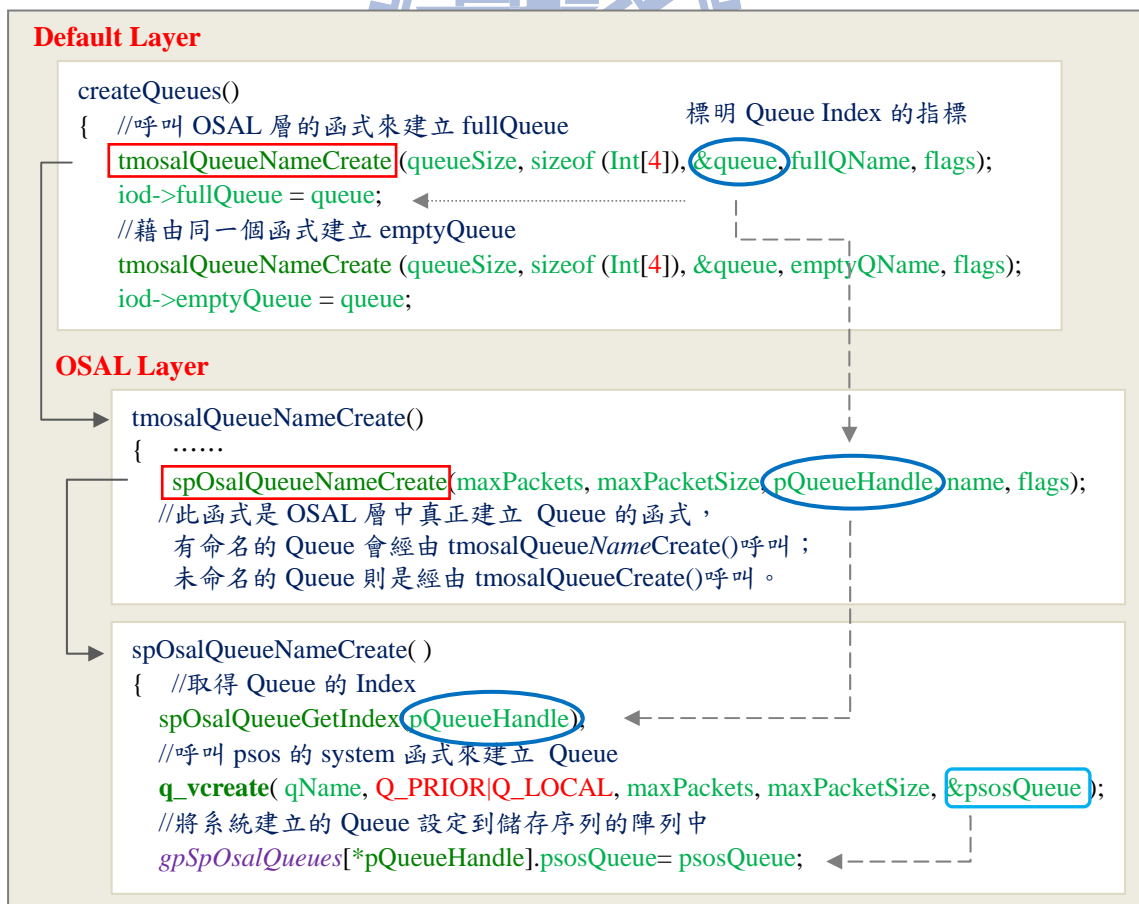


圖 3.4 Queue 的建構流程

在 `spOsalQueueNameCreate()` 函式的執行上，首先會透過一個取得序列編號的函式 `spOsalQueueGetIndex()`，來向 OSAL 層中一個專門存放序列資料的陣列拿取序列的編號。此陣列稱為 `gpSpOsalQueues[]`，當系統建立了一個序列，會將序列的位址搭配一個序列編號儲存到此陣列中，因此每個序列都具有一個獨一無二的序列編號。而 `spOsalQueueGetIndex()` 函式在拿取序列編號時，會由陣列 0 開始檢查，若判斷出該元素的值不等於零，則代表陣列中此元素已經存有序列資料了，則函式會再往下一個元素來檢查，直到取得空的陣列元素為止，而此元素在陣列中的編號即為序列編號。在設定好序列編號後，OSAL 層便呼叫 `psos` 的系統函式 `q_vcreate()` 來建立序列，當系統函式執行完畢後會將所建立的序列位址回傳，而此位址便可以搭配前面搜尋到的序列編號一起儲存到 `gpSpOsalQueues[]` 陣列中，完成序列的建立。

6. `tmPacket_create()`

這部分是來建立 IOD 上傳輸的 `packet`，在執行 `tmPacket_create()` 函式之前，會將 IOD Setup 結構中與 `packet` 相關的參數設定到 `packet` 的 Setup 結構中。`packet` 的 Setup 資料結構成員如表 3.3 所示，包括了 `packet` 資料結構所要放置的記憶體空間、`packet` 上所具備的 buffer 數量、在此 IOD 上傳輸的 `packet` 起始編號、釋放 `packet` 資料內容的函式、`packet` 特定條件設定的參數、以及此 `packet` 所屬的 IOD 指標……等，而在設定好這些 `packet` 的 Setup 結構後，會將此結構傳入 `tmPacket_create()` 函式中，提供給 `packet` 建立時的參數設定。

表 3.3 `tmPacket_Setup` 結構

<code>tmPacket_Setup</code>		
<code>mem</code>	<code>packetBase</code>	<code>numOfBuffers</code>
<code>numOfProperties</code>	<code>propertySetups</code>	<code>releaseFunc</code>
<code>cookie</code>		

由於 `tmPacket_Create()` 函式在執行上一次只會設定一個 `packet`，因此 Default 層會依據應用層在 Setup 結構中設定的 `packet` 數量來決定呼叫 `tmPacket_Create()` 的次數。而傳入 `tmPacket_Create()` 的參數，除了已設定好的 `packet` Setup 結構外，還有一個待設定的 `packet` 陣列，該陣列是用來存放在此 IOD 上傳輸的 `packet` 資料，因此在這部分會將其傳入給 `packet` 的建立函式，讓函式能將建立好的 `packet` 存放到此陣列中。建構 `packet` 的流程如圖 3.5 所示，在 `tmPacket_Create()` 函式中，會再透過一個建立

packet 陣列的函式 `tmPacket_ArrayCreate()` 來設定 packet 的結構，而 `tmPacket_ArrayCreate()` 函式具有一次可以設定多個 packet 結構的功能，但由於在 Default 層上已經將所要建立的 packet 數寫入迴圈中，設定好函式所要執行的次數，因此在這部分會將 `tmPacket_ArrayCreate()` 函式設定為一次建立一個 packet。

```

for (i=0; i!= iod->numberOfPackets; ++i) // numberOfPacket 即為所要建立的 packet 數
{
    // 待設定的 packet 資料陣列
    tmPacket_Create( &iod->packetArray[i], &packetSetup );
    ++packetSetup.packetBase;
    // 每建立完一個 packet，packetBase 加 1
    // 設定好的 packet setup 結構
}

tmPacket_Create( ptmPacket_t *create, tmPacket_Setup_t const *setup)
{
    /* 呼叫建立 packet 陣列的函式，傳入值 "1" 代表建立陣列中的一個元素，
    也就是建立一個 packet 的意思 */
    tmPacket_ArrayCreate( create, 1, setup);
    .....
}

```

圖 3.5 Packet 的建構流程

在 `tmPacket_ArrayCreate()` 函式建立 packet 的過程中，如圖 3.6 所示，首先會配置一個記憶體空間，提供給 packet 存放資料結構的設定，接著會將傳入的 packet setup 參數一一設定到 packet 的資料結構裡。當這部分的結構參數設定結束後，便完成了 packet 初步的建立，但是在 packet 的資料結構中仍有尚未設定的結構成員，因此接下來會針對這部分結構成員的設定做進一步的討論。

```

tmPacket_ArrayCreate(ptmPacket_t create[], Int n, tmPacket_Setup_t const *setup)
{
    // 配置記憶體區塊給 packet 結構設定參數使用
    tmvMem_Calloc (setup->mem, &tmp, n*Packet_SIZE
                  (setup->numOfBuffers + setup->numOfProperties));
    // 將 packet setup 結構的參數設定到 packet 結構裡
    create[0] = tmp;
    create[0]->numOfBuffers = setup->numOfBuffers;
    create[0]->numOfProperties = setup->numOfProperties;
    create[0]->header->id = setup->packetBase;
    create[0]->header->mem = setup->mem;
    create[0]->header->releaseFunc = setup->releaseFunc;
    .....
}

```

圖 3.6 tmPacket_ArrayCreate() 函式內容

7. Packet 與 Queue 的設定

在一個 packet 的資料結構 tmPacket 中，如圖 3.7 所示，包含了描述此 packet 的標頭資訊(header)以及 buffer 資訊兩大資料結構。在前一個函式 tmPacket_Create() 裡，主要是將 packet Setup 的結構參數設定到 packet 資料結構中，由於 packet Setup 裡的結構成員大部分與 header 結構相關，因此 header 部分的結構參數在前一個步驟已大致設定完成，接下來所要設定的為 packet 結構中 buffer 部分。

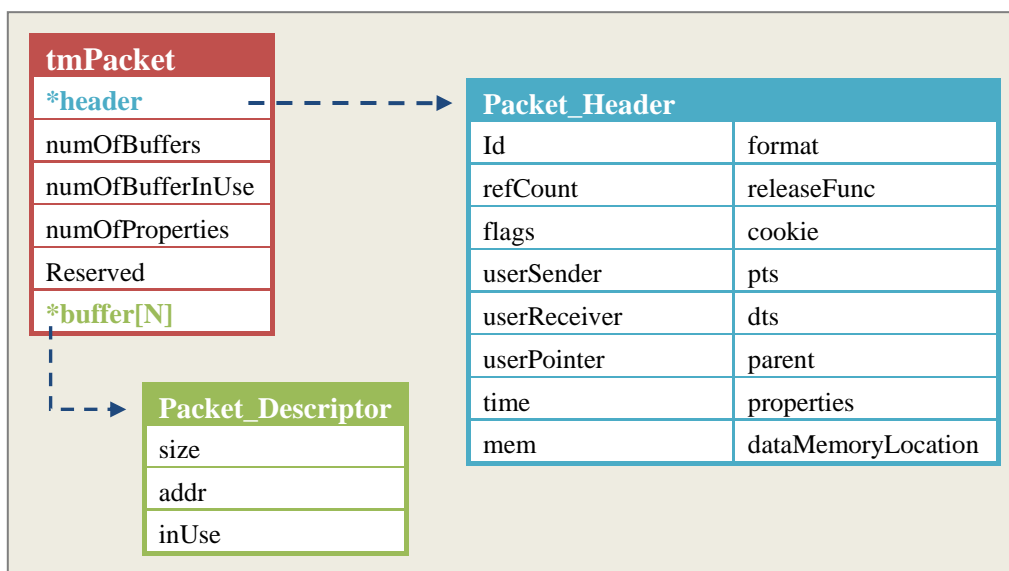


圖 3.7 Packet 的結構關係

在這部分會依照前一步所建立的 packet，依序設定每個 packet 上的 buffer 大小以及位址，函式的設定流程如圖 3.8 所示。首先會依據應用層所設定的 buffer 尺寸來配置一塊記憶體區塊，提供給 buffer 傳輸資料時使用；接著會藉由設定 buffer 尺寸的函式 tmPacket_SetBuffereSize()，將此 buffer 的尺寸設定到 packet 資料結構中存放 buffer 尺寸的結構成員裡 (buffer[n] .size)；而接下來會將取得的記憶體區塊位址，藉由設定 buffer 位址的函式 tmPacket_SetBufferAddr()，設定到 packet 資料結構中存放 buffer 位址的結構成員裡 (buffer[n] .addr)。當 packet 中的每個 buffer 都完成上述的設定後，此 packet 即設定完畢。

設定完成的 packet 即可放置到前面建立好的序列(emptyQueue)中，因此這裡會將 emptyQueue 以及 packet 的指標傳入至 OSAL 層上負責傳送 Queue 的函式 tmosal-QueueSend()，讓 tmosalQueueSend() 函式藉由呼叫 psos 的系統函式 q_vsends() 將 packet 的設定資訊複製到 Queue 上面。當把所有 packet 資料都複製到 emptyQueue

上之後，整個 IOD 的建立就算是告一段落。

```

for (i = 0; i != iod->numberOfPackets; ++i) //依序設定每個 packet
{
    for (j = 0; j != setup->numberOfBuffers; ++j) //設定 packet 裡的每個 buffer
    {
        tmviMem_Alloc (iod->bufferMem, &tmp, setup->bufSize[j]); //配置記憶體區塊給 buffer
        tmPacket_SetBufferSize (iod->packetArray[i], j, setup->bufSize[j]); //設定 buffer 的大小尺寸
        addr=&tmp;
        tmPacket_SetBufferAddr (iod->packetArray[i], j, addr); //設定 buffer 的位址
        .....
    }
    msgBuf = (UInt32) iod->packetArray[i]; //將設定好的 packet 傳入 Queue 序列
    tmosalQueueSend (iod->emptyQueue, msgBuf, sizeof msgBuf, &noWait, 0U);
}

```

-----> emptyQueue 的位址

```

tmosalQueueSend()
{
    //藉由 psos 的系統函式將 packet 複製到序列上
    q_vsend(gpSpOsaiQueues[queueHandle].psosQueue, (void*) pBuffer, packetSize);
    .....
}

```

圖 3.8 buffer 的設定流程

3.2.3 IOD 的傳送端與接收端模組設定

當 IOD 建立完成之後，此時 IOD 的資料結構內已設定好前後所連接的模組資訊，但是在前後這兩個模組中仍尚未取得 IOD 資訊，因此在模組 Instance Setup 的過程中，會將此 IOD 設定到模組的 Instance 結構內。每個模組的 Instance 結構中，都具備了兩個 IOD 類型的資料結構，一個用來設定模組輸入端所連接的 IOD，稱為 inputDescriptors，另一個則用來設定模組輸出端所連接的 IOD，稱為 outputDescriptors。連接關係的設定以圖 3.9 為例，IOD 位於 Sender 模組的輸出端，因此在 Sender 模組中會將此名稱為 A 的 IOD 設定給 outputDescriptors；而 IOD 位於 Receiver 模組的輸入端，所以 Receiver 模組會將此 IOD 設定到 inputDescriptors 中。

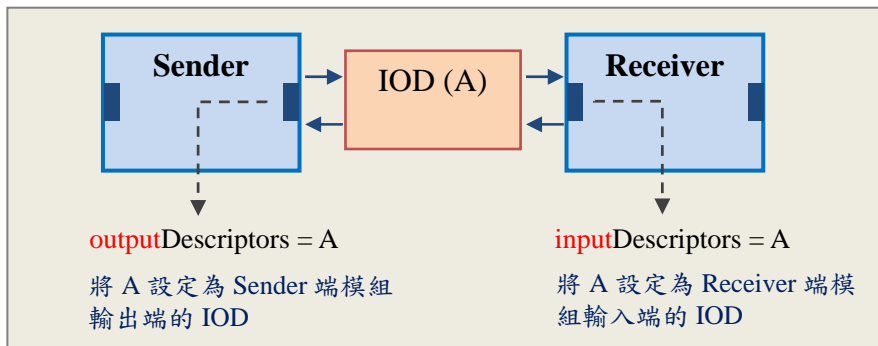


圖 3.9 傳送端與接收端模組的 IOD 設定

當傳送端與接收端模組皆把 IOD 資訊設定到資料結構上之後，便確定了 IOD 與前後模組之間的連接關係，同時，與 IOD 相關的設定也到此告一段落。

3.3 模組間資料串流

當模組啟動後，必須透過IOD上傳輸的packet來讀寫影像資料，因此在這部分所要探討的重點為模組如何從IOD的序列中取得影像的位址，以及如何將處理完畢的影像資料位址儲存到IOD的序列上。

以IOD的傳送端模組為例，要將處理完畢的影像資料輸出到IOD上，會透過Default層中一個處理輸出資料的函式tsaDefaultDataoutFunction()來執行。而首先必須取得此IOD序列上尚未儲存資料的packet，才能將模組的影像資料寫入packet中，因此模組在呼叫tsaDefaultDataoutFunction()時，會傳入一個拿取空packet的旗標“tsaDataoutGet-Empty”，函式就會依據此輸入的旗標來判斷後續所要執行的內容，函式的流程如圖3.10所示。



圖 3.10 拿取空 packet 的函式流程

在拿取空 packet 旗標的 case 中，首先會連結到一個處理 IOD 傳送端 packet 的函式 Connection_SourceReceive()，由於 packet 以及 Queue 都是建立在 OSAL 層中(在 3.2.2 小節 Queue 的建立中有提到)，因此在這個函式裡會再藉由 OSAL 層上接收 Queue 的函式

tmosalQueueReceive()，來取得空的packet。而在OSAL層接收Queue的函式中，則會將此IOD的emptyQueue指標傳給psos的系統函式q_vreceive()，讓系統函式辨認出目前所要取得的是哪一個序列中的packet。

當系統函式取得packet後，便可將此packet的位址回傳給傳送端模組，而傳送端模組則會將處理完畢的影像資料寫入此packet中。接下來，就要將這些寫有資料的packet放入IOD的序列中傳送出去，傳送packet的途徑與取得packet的方法類似，傳送端模組同樣呼叫了Default層的tsaDefaultDataoutFunction()函式，但這部分傳入的旗標為“tsaDataoutPutFull”，是要將存有資料的packet放入序列中，因此在寫入packet的case中，會連結至處理傳送端輸出packet的函式Connection_SourceSend()來執行。而在這個函式裡，則會將傳送端模組寫好的packet傳入OSAL層上傳送Queue的函式tmosalQueueSend()，再由OSAL層來呼叫psos的系統函式q_vsend()，藉由系統函式將packet傳送到fullQueue的序列上。

接下來，位於此IOD接收端的模組會來接收傳送端模組寫入的影像資料。而接收端模組在這部分會藉由Default層中處理輸入資料的函式tsaDefaultDatainFunction()來取得packet，因此接收端模組會傳入拿取full packet的旗標“tsaDatainGetFull”給Default層，讓tsaDefaultDatainFunction()函式執行取得full packet的case。而執行的程序與前一段敘述在傳送端上取得空packet的流程類似，但由於這部分是要拿取攜帶資料的packet，因此必須將IOD上的fullQueue指標傳入到OSAL層接收Queue的函式tmosalQueueReceive()裡，讓接下來所要執行的psos系統函式q_vreceive()能夠從正確的fullQueue序列上取得的packet。

而在接收端模組讀取完packet的資料後，會將packet裡的影像資料清空並放回到IOD的序列中，因此這部分會將“tsaDatainPutEmpty”旗標傳入到Default層的tsaDefaultDatainFunction()函式裡，讓該函式執行把空的packet放回序列的case。執行的程序與傳送端模組將full packet傳入IOD的序列類似，但這部分在將packet傳入OSAL層傳送Queue的函式tmosalQueueSend()之前，會先透過一個清除packet資料的函式tmPacket_Release()來清空packet上所儲存的影像資料，函式流程如圖 3.11所示。在tmPacket_Release()函式中，會執行packet結構上的清除packet的函式releaseFunc，由於在設定packet資料結構時已將tsaConnection_ReleasePacket()函式設定給releaseFunc，因此在這部分會直接連結到

tsaConnection_ReleasePacket()函式。而接下來則會透過tmPacket_Empty()函式來清除 packet上資料，將packet上所使用到的buffer設定為Null；再將這個清空的packet透過OSAL層傳送Queue的函式tmosalQueueSend()，傳入到IOD的emptyQueue序列上。

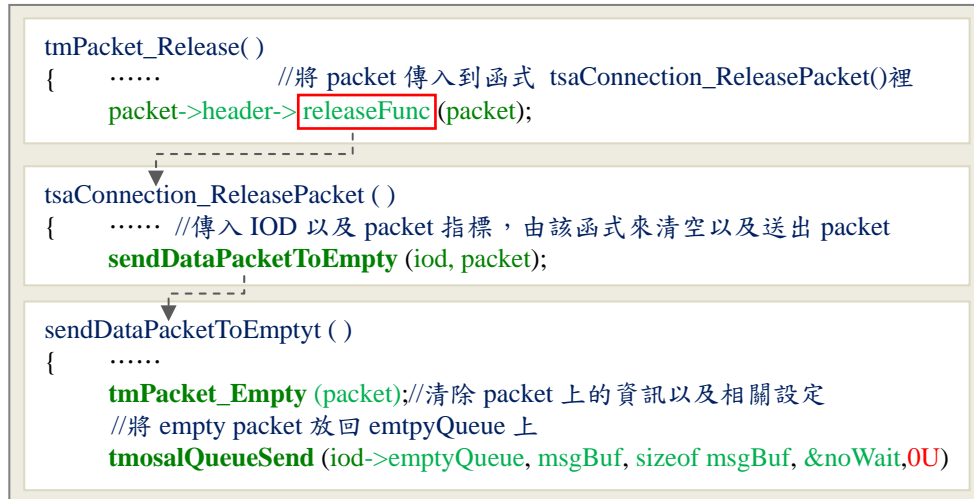


圖 3.11 Release packet 的函式流程

在上述操作 packet 的四個程序中：Get full、Put full、Get empty、Put empty，前三個操作程序 Get full、Put full、Get empty 主要是根據模組傳入的 IOD 資料結構上所記錄的 fullQueue 與 emptyQueue 序列編號，到系統儲存序列的陣列裡尋找該編號所代表的序列，進而在該序列上做 packet 的存取動作。但是在 Put empty 這個部分，則不是根據模組傳入的 IOD 結構所儲存的 emptyQueue 編號來選取序列，而是依據目前所要傳送的 packet 上所記錄的 emptyQueue 編號來選取。由於在 packet 的建立過程中，會將 packet 所屬的 IOD 指標設定到 packet 的結構成員 cookie 中，因此在每個 packet 上都記錄著自己是屬於哪一個 IOD 的 packet。而在 Put empty 的程序上，就會依據 packet 所屬的 IOD 上記錄的序列編號，來選取 packet 所要存放的 emptyQueue。

而在接收端模組將 packet 放回 emptyQueue 序列上之後，當傳送端模組需要空的 packet 來寫入資料時，會再重複前面所描述的步驟，到 emptyQueue 上拿取 packet，取得一個空的記憶體區塊來寫入資料。而在 IOD 上的每個 packet 也都會透過上述的程序，不斷的重複使用，以一個循環的 packet 傳輸方式，來提供給傳送端與輸出端模組。

第四章 輸入端與輸出端模組

本章以 TSSA 架構中的輸入端與輸出端模組為探討的重點，由於在輸入端與輸出端模組上都會使用到 DSP 晶片上的硬體區塊，並牽涉到外部與 DSP 晶片相連接的訊號處理晶片，因此在結構上與一般的軟體模組較為不同，在本章節中會藉由片段的程式內容來了解模組的運作。而在章節的順序上，首先 4.1 節中會先針對系統的初始化設定進行討論，探討如何將硬體環境上的設定資訊提供給模組來使用；接下來 4.2 節裡，會對輸入端模組的架構進行說明，並探討架構上模組拿取硬體資訊的函式內容；而在 4.3 節則會討論輸入端的硬體結構以及軟體模組的設定與運作；最後在 4.4 節則是探討輸出端模組的結構。

4.1 系統初始化設定

在應用程式開始執行之前，必須要先對整個系統環境進行初始化的設定，讓管理板子的 BSL 層能夠取得硬體上的相關資訊，並且讓 DSP 晶片能夠辨認出周圍所連接的晶片裝置。而這部分會以輸入端的環境為例，探討輸入端模組的硬體資源在系統初始化的流程設定，其中輸入端模組在硬體環境上主要分為兩部分，訊號的類比數位轉換晶片單元 (SAA7115) 以及 DSP 晶片上處理輸入訊號的 SVIP 單元。

在應用程式上所執行的主程式 `tmMain()` 是由 TSA (Trimedia Software Architecture) 定義的一個函式，其功能就相當於一般 C 語言中的 `main()` 函式，但附加了許多系統上功能函式的連結。因此當應用程式在執行 `tmMain()` 函式，但尚未開始執行主函式裡的內容之前，會先連結到 `tmMain()` 函式裡，並藉由定義於 `tmMain()` 上的系統功能函式，對整個 DSP 的環境做初始化的動作。而 `tmMain()` 的函式內容如圖 4.1 所示，`tmMain()` 實際上是執行一個叫 `root()` 的函式，而 `root()` 函式裡的 `tmMain_BODY()` 則會開始呼叫相關的函式來初始化 DSP 上的環境。其中 DSP 環境的初始化，主要包含了作業系統 `psos` 以及 DSP 外部所連結的晶片兩部分。首先在作業系統 `psos` 上的初始化內容，是對計時器 (timer) 以及錯誤處理器 (error handler) 做一些初始值設定，兩者都是藉由 `psos` 所提供的系統函式來執行。而在初始化 DSP 外部連結晶片的部分，也是本節所要探討的重點，在執行的程序上，當 `tmMain_BODY()` 函式的連結到 `tmMain_START()` 函式時，在 `tmMain_START()` 函式裡則會進一步連結到 BSL 層管理初始化的函式 `tmbslMgrInit()`，

而此 `tmbslMgrInit()` 函式會判斷 BSL 層的內容設定是否初始化過，若有，會回傳一個確認值，若無則開始初始化的動作。

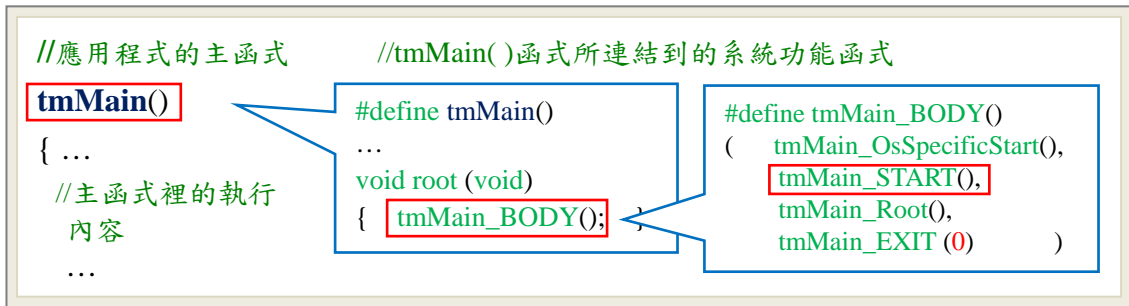


圖 4.1 `tmMain()`函式內容

由於 BSL 是用來連結 DSP 與周圍的硬體裝置，所以 BSL 內部具備了一套管理的核心架構，稱為 `tmbslCore`。`BSL` 核心提供了介面函式來初始化硬體裝置、設定或取得系統的資訊(例如記憶體分配以及記憶體的實體和虛擬位址)、軟硬體的模組資訊、中央處理器的資料快取控制等等功能，整體的 `BSL` 系統架構如圖 4.2 所示。

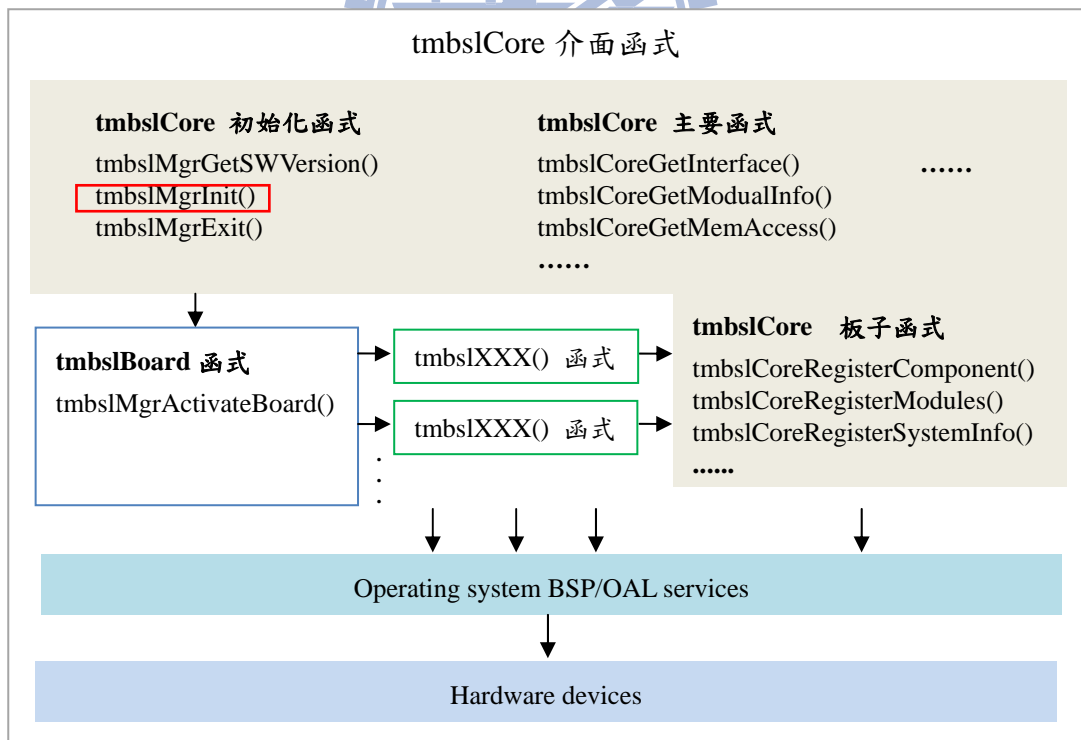



圖 4.2 `tmbslCore` 系統架構

由 4.2 圖所示的系統架構可發現，在整個架構的最上層為 `BSL` 介面函式，而介面函

式大致上可分為兩部分，一部分提供給上層軟體模組操作，在這裡面包含了 tmbslCore 的初始化函式以及主函式，另一部分的函式則是提供給硬體平台註冊使用，稱為 tmbslCore 板子函式。所以當 BSL 的上層呼叫了 tmbslCore 初始化函式，BSL 核心管理便會開始執行一連串啟動硬體環境的動作。首先會藉由管理啟動硬體環境的函式 tmbslMgrActivateBoard()，針對目前所使用的硬體環境啟動其相關的函式，同時這些被啟動的硬體也會將自己的資料註冊給 tmbslCore 的板子函式，如同圖 4.2 中的第二層，最終讓 BSL 的核心管理部分能夠掌握所有硬體環境的資料。

在 tmbslMgrActivateBoard() 函式啟動硬體環境的過程中，首先會檢查一個存放板子目錄的陣列，陣列的資料內容如圖 4.3 所示。在此陣列結構中存放了各種不同型號的板子 id，例如在我們測試環境上用到的 DVR Demo 板以及 PCI Lite Demo 板。其中板子的 id 同時也是用來啟動板子的函式指標，所以當 BSL 管理啟動板子的函式在逐一檢查陣列中的 id 時，就會同時連結到該板子啟動函式，並判斷何者為目前所操作硬體環境，而只有該硬體環境的啟動函式能夠順利運作執行。



```
static compActivateFunc_t *ppBoardActivateList [] =
{
    //其它硬體環境的板子 id
    &tmbslMgrBoardActivateOfBoardId0x00081131,
    &tmbslMgrBoardActivateOfBoardId0x10001131,
    &tmbslMgrBoardActivateOfBoardId0x10011131_XTV,
    &tmbslMgrBoardActivateOfBoardId0x10011131,
    &tmbslMgrBoardActivateOfBoardId0x10021131,    // tmbslPnxDVRDemo
    //本硬體環境的板子 id
    &tmbslMgrBoardActivateOfBoardId0x10041131,    // tmbslPnxLiteDemo
    .....
};
```

圖 4.3 BoardActivateList[] 資料陣列

在板子的啟動函式中會以下列的步驟，依序對板子做啟動的程序設定：

- **啟動(Activation)**

在啟動的這個步驟中，會一併執行接下來的三個步驟偵測、初始化以及註冊，所以在 tmbslBoardActivate() 函式裡，會去呼叫偵測、初始化以及註冊的相關函式。

- **偵測(Detection)**

偵測時主要是在確認板子 id 是否與硬體符合、設定硬體的記憶體位址分配、並回傳啟動硬體資訊的指標。

- **初始化(Initialization)**

初始化過程中會更新 BSL 的系統資訊以及記憶體的资料結構，包括把在偵測時所設定的記憶體位址更新到系統的記憶體區塊上。

- **註冊(Register)**

註冊則是把在前幾個步驟得到的資料結構存放到 BSL 的核心管理層上，包含了系統資訊、記憶體使用區塊、硬體上的模組資源等資訊。

而在板子硬體單元的啟動順序上，首先會對 DSP 晶片 PNX1005 做初始化的動作，接著再啟動 DSP 晶片周邊所連接的硬體單元，其中在處理輸入影像的類比數位轉換(A/D)晶片部分，是藉由呼叫板子上處理影像的初始化函式 `tmbslPnxLiteDemo_VideoInit()` 來啟動(`PnxLiteDemo` 為板子的名稱)，同時在這個函式裡，會去辨認目前有哪些 A/D 的晶片存在，表 1.列舉了在兩種硬體平台上所使用的 A/D 晶片型號，而在確認好晶片型號之後，便將晶片的資料內容註冊到 BSL 的核心管理層，提供操作介面給 DSP。

表 4.1 板子與使用的 A/D 晶片

板子的類型	A/D 型號
PCI Lite Demo Board	SAA7115、TDA19978
DVR Demo Board	Techwell2864

註冊的流程如圖 4.4 所示，在 `VideoInit()` 的函式中會呼叫各晶片的註冊函式，而這些註冊函式的主要目的就是向 `tmbslCore` 註冊自己的資料，註冊的方法是藉由 `tmbslCore` 上的註冊函式 `tmbslCoreRegisterComponent()`，將自己的資料結構位址以及函式指標等資訊存放到 `tmbslCore` 的陣列裡面，此陣列即為圖 4.4 中 `gBslInterfaceInfo [index]`，是專門用來儲存這些 DSP 周邊硬體裝置的資訊。

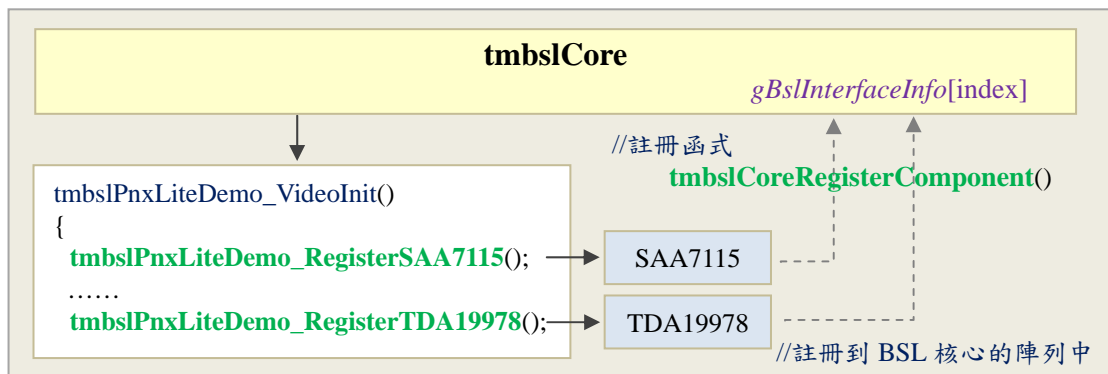


圖 4.4 PCI Demo 板 A/D 晶片的註冊流程

當 tmbslCore 取得這些硬體的資料結構以及函式位址資訊後，整個 BSL 初始化的步驟就算是告一段落，接下來上層的軟體模組若有需要硬體的相關資訊，就可以直接到 tmbslCore 中來拿取資料。

4.2 輸入端模組架構

在 DSP 晶片上負責處理輸入影像的硬體區塊為 SVIP，而在軟體層所建立用來操作此硬體區塊的模組名稱為 tmVcapSvip，該模組具有 TSSA 的基本分層架構，但由於影像的擷取來自硬體，相較於其它無硬體結構的模組，必須多具備一層操控硬體區塊的層面，該層位於軟體架構的最底部稱為 DL 層，是用來控制設定 SVIP 硬體區塊。而影像資料在進入 DSP 晶片之前必須先經過一個類比訊號轉換為數位訊號的過程，由於執行這部分功能的晶片位於 DSP 的外部，所以必須把晶片的資訊提供給 DSP，而這些晶片的設定資訊則是由 BSL 層來管理，並提供給輸入端的模組 tmVdecAna 以及 tmVcapSvip 來使用，整體輸入端的架構如圖 4.5 所示。

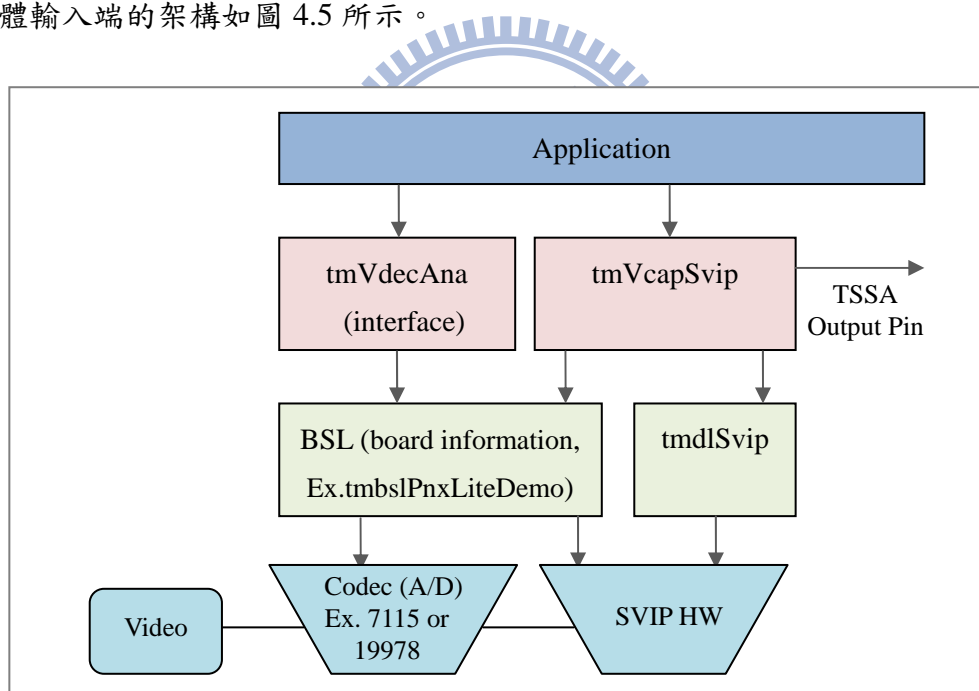


圖 4.5 輸入端分層與模組架構

由圖 4.5 可以看到在執行輸入端的模組時，應用程式會直接操控上層的軟體模組 (tmVdecAna、tmVcapSvip)，操控方式則是藉由呼叫模組所提供的介面函式來執行，而這些被呼叫的介面函式則會往下層傳遞訊息，告知模組下層目前所要處理的程序有哪些，並藉由 DL 層與 BSL 層的函式內容來協助模組與硬體之間的設定。接下來會先針對 tmVdecAna 模組的設定資訊來探討，了解模組與外部晶片的設定關係，而 tmVcapSvip

模組的內容，則會在下一小節中進行討論。

當應用程式在取得 A/D 硬體資訊的過程中，會透過 tmVdecAna 模組來操作，誠如前一節所說，在 BSL 初始化的過程中，會將硬體的資訊存放在 tmbslCore 裡，因此 tmVdecAna 就可直接透過 BSL 層的函式來尋找 A/D 硬體的資源。而在 tmVdecAna 模組中具備一個專門用來存放硬體資訊的資料結構，如圖 4.6 所示，當應用程式上需要輸入端 A/D 的資訊時，就會藉由 tmVdecAna 模組的函式 tmVdecAna_Open()，到 BSL 層拿取 A/D 的硬體設定資訊，並將取得的資訊存放在此資料結構中，同時將此資料結構的位址設定給應用程式上的指標，讓應用程式也取得這些設定資訊。

```
typedef struct _tmVdecAnaUnitInfo_t
{
    tmbslVdecAnaVideoInSources_t    VISources[MAX_PHYSICAL_UNITS];
    .....
    ptmbslVdecAnaConfig_t           pDecBoardCfg[MAX_PHYSICAL_UNITS];
    ptmbslVdecAnaVbiConfig_t        pDecVbiBoardCfg[MAX_PHYSICAL_UNITS];
    ptmbslVdecAnaExt2Config_t        pDecExt2BoardCfg[MAX_PHYSICAL_UNITS];
} tmVdecAnaUnitInfo_t, *ptmVdecAnaUnitInfo_t;
```

圖 4.6 tmVdecAna 模組存放硬體資訊的資料結構

而 tmVdecAna_Open() 函式主要是透過 tmbslCore 中 GetInterface() 的函式來取得 BSL 層中硬體註冊的資訊，由於註冊的硬體資訊不只一種，所以 tmVdecAna 模組就會利用到不同的巨集來讀取資料，如圖 4.7 所示，而這些巨集會傳入不同的介面 id 給 tmbslCore 中的 GetInterface() 函式，GetInterface() 函式就會依據傳入的介面 id，到儲存於 BSL 層中的資料來搜尋比對，並將取得的資訊的位址回傳給 tmVdecAna 模組。

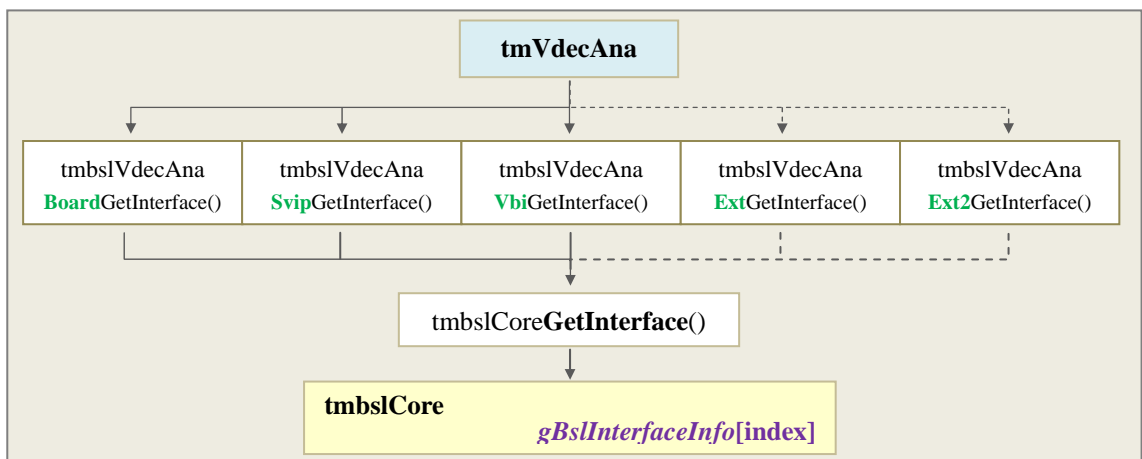


圖 4.7 tmVdecAna 模組取得硬體資訊的流程

tmVdecAna 模組所呼叫的巨集種類大致可分為以下五種：

- **tmbslVdecAnaBoardGetInterface()**

是用來取得目前板子上的 A/D 晶片的相關資訊，所以在定義巨集的時候，會把能讓 BSL 識別的介面 id 傳入給 tmbslCoreGetInterface() 函式，讓 tmbslCore 層知道目前所需要的是 A/D 的資訊，並到儲存硬體資訊的資料結構中尋找吻合的 id，再將資料回傳給上層。

- **tmbslVdecAnaSvipGetInterface()**

是用來取得 DSP 晶片上提供給 A/D 晶片連接的 SVIP 資訊。在 DSP 上的處理輸入影像的部分，在硬體上分成兩個單元，一組是 SVIP，另一組為 FGPI(Fast General Purpose Input port)，FGPI 除了可以處理攝錄的影像資訊，當板子上有兩組 DSP 晶片要傳送資訊時，FGPI 可以用在第二顆 DSP 晶片，處理前一顆 DSP 晶片傳遞過來的資料。由於目前皆是使用 SVIP 來處理輸入訊號，所以這個部分主要是將 SVIP 單元的資訊提供給上層。

- **tmbslVdecAnaVbiGetInterface()**

是用來取得影像中的 VBI (Vertical Blank Interval) 資料的設定資訊，不同的 A/D 晶片都具有自己的 VBI 設定方式，在註冊時會將這部分的資訊儲存到 tmbslCore 中，並在這部分把資訊提供給上層。

- **tmbslVdecAnaExtGetInterface()**

此巨集主要是提供一些擴充的資訊來給軟體層的輸入模組使用，但程式上未必會使用到，在這裡只是一併寫入執行。

- **tmbslVdecAnaExt2GetInterface()**

同上 tmbslVdecAnaExtGetInterface()。

當 tmVdecAna 模組取得 BSL 資訊後，就會將這些設定資訊存放到圖 4.7 所示的結構中，並將這個結構的位址回傳給應用程式的全域資料結構中設定，此時應用程式便取得輸入端 A/D 這塊硬體單元的資料，接下來即可依據這些資料來設定程式上的一些參數。此外 tmVdecAna 模組上還提供了一些介面函式來給應用程式操作，表 4.2 中列舉了幾個常使用到的函式，例如取得輸入影像的狀態、格式標準、串流數……等資訊的功能函式。應用程式便可藉由這些函式取得的資訊內容，來對運作上的一些結構參數做設定，而關於 tmVdecAna 模組這部分的探討也到此告一段落，接下來下一節就針對輸入端架構上的另一部分，tmVcapSvip 模組開始進行介紹。

表 4.2 tmVdecAna 模組函式

tmVdecAna_GetSupportedSources()	取得目前環境中所支援的輸入影像數目。
tmVdecAna_GetVideoStatus()	取得目前輸入影像的狀態，判斷輸入訊號是否有被鎖住。
tmVdecAna_SetVideoStd()	設定影像的格式標準，例如 ntsc。
tmVdecAna_SetSourceType()	設定影像的型態，例如 TV、video、camera 等。
tmVdecAna_SetAcquisitionWindow()	設定輸入影像的視窗座標。

4.3 SVIP 模組

在軟體層中控制 SVIP 的模組為 tmVcapSvip，由於這個模組有使用到硬體區塊的部分，模組架構上具備了 OL、AL、DL 三層結構，架構較為龐大，因此在操控上就分成了 Unit 以及 Stream 兩個部分來執行。首先 Unit 是 tmVcapSvip 模組中用來操作硬體資訊的部分，應用程式是會透過 OL 層直接呼叫 DL 層以及 BSL 的函式，來取得輸入端的硬體資訊；而 Stream 這部分則是用來處理上層軟體模組之間的串流設定，主要的運作是由 AL 層上的函式來執行。以下兩小節就開始針對 Unit 和 Stream 的性質以及運作方式來做介紹。

4.3.1 SVIP Unit

在讓 tmVcapSvip 模組能夠運作之前，必須先取得 tmVcapSvip 模組在硬體環境上的一些資訊，例如關於 SVIP 的資料設定、A/D 晶片上的設定資訊、以及上述兩者之間的連接關係，這些資料主要是透過 tmVcapSvip 模組中 Unit 的函式來存取，因此應用層可以透過以下幾個 OL 層所提供的 Unit 介面函式來執行操作。

- tmolVcapSvip**Unit**_GetCapabilities()
- tmolVcapSvip**Unit**_Open()
- tmolVcapSvip**Unit**_GetInstanceSetup()
- tmolVcapSvip**Unit**_InstanceSetup()
- tmolVcapSvip**Unit**_Close()

Unit 在應用層上函式執行的順序與一般模組相同，皆是先取得描述 Capabilities 的資料結構，接著分配記憶體空間給 Instance 結構，取得並設定 Instance，所以接下來便

依序討論上述的函式內容，並藉由這些函式的運作流程來了解 Unit 的內容設定。

1. **tmolVcapSvipUnit_GetCapabilities()**

在這個函式中主要是取得 Unit 的 Capabilities 設定，而 Unit 的 Capabilities 如圖 4.8 所示，具有三個結構成員，Default 層的 Capabilities、DL 層的 Capabilities、Unit 所支援的 Stream 數等，除了 Default Capabilities 的值是設定在 OL 層中，可以直接連結到位址取得資料，其餘兩個結構成員則必須到 DL 層或 BSL 層才能得到設定的資料，因此在這部分會藉由一個 SVIP Unit 的初始化函式 `SvipUnit_localInit()` 來執行。

```
typedef struct _tmolVcapSvipUnit_Capabilities_t
{
    ptsaDefaultCapabilities_t      pDefCapabilities;    //Default capabilities
    tmdlSvipUnit_Capabilities_t    * psvipCapabilities; //DL capabilities
    UInt32                          numberOfStreams;
} tmolVcapSvipUnit_Capabilities_t, *ptmolVcapSvipUnit_Capabilities_t;
```

圖 4.8 Unit Capabilities 資料結構

在 SVIP Unit 初始化函式 `SvipUnit_localInit()` 中，初始化的方式主要是藉由 DL 層以及 BSL 層的函式來取得 SVIP 上的設定的資訊，包含了 SVIP 上的拿取影像的時脈頻率、MMIO 的位址、輸入影像的 Stream 設定……等，並將這些取得的訊息設定在 Unit 的資料結構中。圖 4.9 為 `SvipUnit_localInit()` 函式的內容，在這個函式中會再藉由連結到三個相關的函式來執行初始化的動作，以下依序為這三個函式的內容介紹。

```
static tmErrorCode_t svipUnit_localInit (tmUnitSelect_t svipUnitNumber)
{
    // 取得可使用的 unit 數目
    tmolVcapSvip_GetNumberOfUnits();
    // 取得 DL 層的 Capabilities
    tmdlSvipUnit_GetCapabilities();
    // 取得 VdecAna 的介面
    tmolSvip_VdecGetNumberOfStreams();
}
```

圖 4.9 `SvipUnit_localInit()` 內容

(1) `tmolVcapSvip_GetNumberOfUnit()`

Unit 的初始化，首先必須取得硬體上可使用的 unit 數，unit 是指在 DSP 上 SVIP 所在的區塊，由於 DSP 上有兩組 SVIP，所以不論是否有使用到兩組 SVIP，取得的 unit 數通常會是二。這部分在程式上的流程可參照圖 4.10，當 OL 層的 `GetNumberOfUnit()` 函式被呼叫時，會連結到 DL 層的 `GetNumberOfUnit()` 函式來取得 unit 數，而在取得 unit 數的同時，DL 層的 `dlSvipLocalInit()` 函式會開始對 unit 的 instance 結構做一些初始值的設定。其中部分 instance 的設定資訊與硬體相關，例如暫存器的位址，由於這方面的資訊在 SVIP 向 `tmbslCore` 註冊時，已將資料存放置 BSL 層上的一個結構中，所以 DL 層就會利用 `tmbslCore` 的介面函式，到 BSL 層中取得 SVIP 硬體模組的資訊。

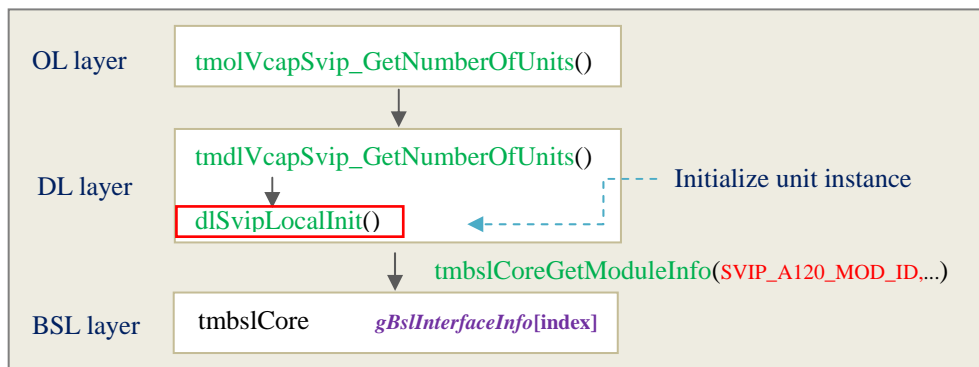


圖 4.10 取 unit 數的程式流程

(2) `tmdlSvipUnit_GetCapabilities()`

Unit 的 Capabilities 資料結構包含了三個結構成員，分別是 Default 的 Capabilities、DL 的 Capabilities、Unit 所支援的 Stream 數。DL 上的 Capabilities 是 `tmVcapSvip` 模組中對 Default Capabilities 沒提供的部分所增加的結構成員，主要內容是關於模組版本以及 SVIP 所支援 Instance 數目的設定，而 OL 以及 DL Capabilities 的資料結構關係則如圖 4.11 所示。其中在執行上述第一階段的 `tmolVcapSvip_GetNumberOfUnit()` 函式時，DL 層的 Capabilities 資料結構已經在初始化函式 `dlSvipLocalInit()` 中，設定好結構成員的參數值，因此在 DL 層的 `tmdlSvipUnit_GetCapabilities()` 函式裡，會將已經設定好的 Capabilities 資料結構位址直接回傳給模組的 OL 層。

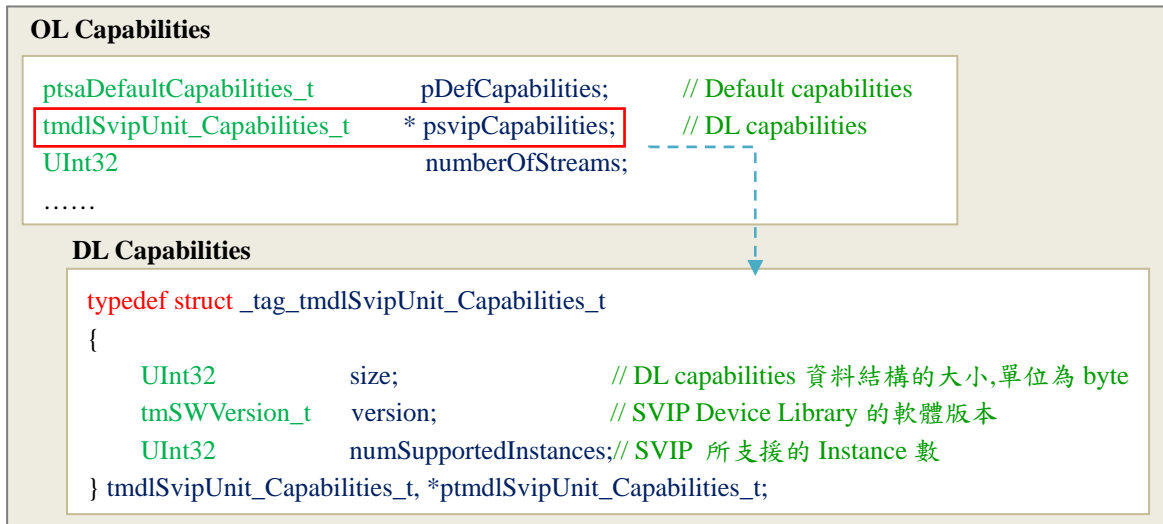


圖 4.11 Unit Capabilities 的結構關係

(3) tmolSvip_VdecGetNumberOfStream()

這部分主要是取得從 A/D 單元傳到入 SVIP 上的影像數量(Streams)，而 A/D 單元的資訊存放在 tmbslCore 層裡，因此 tmVcapSvip 模組會透過 BSL 的介面函式去取得 Stream 數的資訊，並將取得的數字設定給 OL 層 Capabilities 中存放 Stream 數的結構成員。

當執行完上述函式的內容，Unit 的初始化算是告一段落，同時 Unit Capabilities 的資料結構也設定完成，接下來就繼續討論 Unit 另一個資料結構 Instance 的相關設定。由於 Unit 是 tmVcapSvip 模組用來處理硬體相關資訊，所以在 Instance 結構中並不具有 Default 層所提供的 Setup 資料結構，而設定的過程上也不會透過 Default 層的函式來執行，而是直接至 DL 層或 BSL 層來存取資料。因此在 Unit 的 Instance 資料結構中，包含了許多 DL 層以及 BSL 層所定義的結構成員，如圖 4.12 所示，用來存放 DL 層與 BSL 層上資料結構的設定參數。

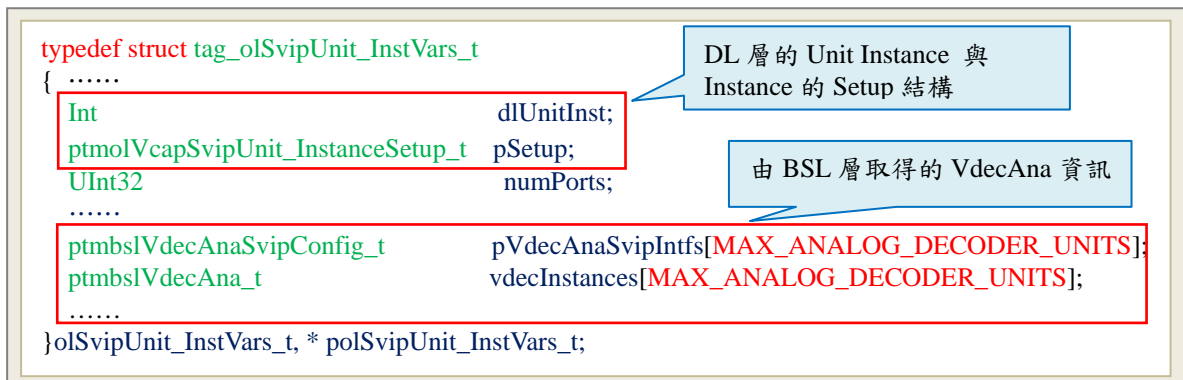


圖 4.12 Unit 的 Instance 結構

而在應用程式上，為了取得硬體 A/D 與 SVIP 這兩部分的設定資訊，以及設定 Unit Instance 結構中尚未有值的結構成員，會透過 Unit 所提供的 `Open()`、`GetInstanceSetup()`、以及 `InstanceSetup()` 三個函式，來存取 Instance 的結構參數，以下即為這三個函式執行內容的說明。

2. `tmolVcapSvipUnit_Open()`

在 Unit 的 `Open()` 函式中，首先會分配記憶體給 Unit 的 Instance 結構，提供 Unit 存放 Instance 設定的空間，接著藉由 DL 層的 `Open()` 函式來取得 DL 層 Instance 的資料設定。其中 DL 層的 Instance 在前面描述過的 `tmolVcapSvip_GetNumberOfUnit()` 函式執行過程中，在呼叫初始化函式 `dlSvipLocalInit()` 來執行時，已經設定好 Instance 裡部分的值，因此這部分 DL 層的 `Open()` 函式，則會直接回傳 DL Instance 資料結構的位址給 OL 層。除了回傳 DL Instance 位址之外，DL 層的 `Open()` 函式還會呼叫一個初始化硬體的函式，`tmdlSvipUnit_InitHw()`，在這個函式裡會設定 SVIP 的暫存器初始值，而暫存器這部分的詳細內容會在下一小節 Stream 裡說明。

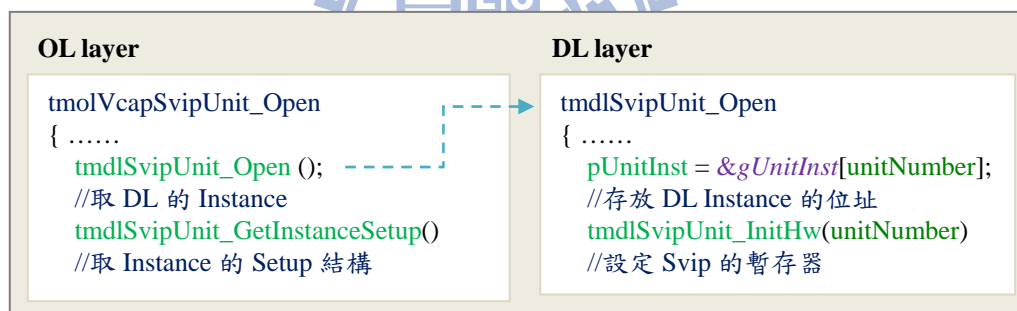


圖 4.13 Unit Open 函式內容

Unit 的 Instance 在取得 DL 層 Instance 的結構位址之後，接下來會藉由 DL 層的 `GetInstanceSetup()` 函式來取得 DL Instance 的 Setup 結構，Instance 的 Setup 結構是 Instance 結構裡提供給應用層修改設定的部分，所以在取得 DL Instance 的 Setup 結構後，會將這個結構的值設定到 Unit Instance 裡，做為 OL 層提供給應用層變更設定的初始值，Unit 的 `Open()` 函式內容如圖 4.13 所示。

3. `tmolVcapSvipUnit_GetInstanceSetup()`

當應用層呼叫此函式時，是為了要取得 Instance Setup 的結構位址以便設定結構裡的參數，由於在 `Open()` 函式中已經取得 Instance Setup 的位址，所以在這個函式裡

只要將結構位址回傳給應用層即可。而應用程式是在取得位址後，會對 Instance Setup 裡的參數重新設定一次，設定的內容大致上與預設值相同，只有在一些判斷的布林系數上會做更改。

4. tmolVcapSvipUnit_InstanceSetup()

當應用程式設定好 Instance Setup 結構後，即可藉由 InstanceSetup() 函式將資料傳給下層來更新設定。在 InstanceSetup() 函式中，會把從應用層傳入的資料複製到 DL 層的 Instance Setup 結構中，除了複製資料外，DL Instance Setup 結構上還有一個 port 的參數需要設定，port 這個參數結構主要是在設定 A/D 晶片與 SVIP 之間的連接埠關係。由於 DSP 上具有兩組 SVIP，所以當 A/D 晶片處理完訊號的轉換後，必須將影像資料傳入其中一個 SVIP，這部分的資訊設定在 A/D 晶片的資料結構裡，因此這裡會藉由函式 tmolSvip_VdecGetPortSetup() 來取得 A/D 與 DSP 的硬體連接格式、數位輸入的格式……等設定。

在 4.2 節 tmVdecAna 模組設定中有提到，tmVdecAna 模組會把到 BSL 層中取得的硬體資訊存放在自己的資料結構裡，A/D 晶片的設定資訊也包含於其中，因此便可將這些資料應用於目前的所要設定的連接埠上。而在 tmolSvip_VdecGetPortSetup() 函式中，則會依據 tmVdecAna 模組所提供的數據來設定 port 的結構參數。當取得連接埠的設定資訊後，整個 Instance 的資料結構就算是設定完成，此時 InstanceSetup() 函式會呼叫 DL 層的 InstanceSetup() 函式，把前面對 Instance 所做的變更設定到 DL 層裡。接下來 DL 層則會依據這些 Instance 的值來設定 SVIP 的暫存器。而 Unit 這部分的設定到此就大致上告一段落，下一小節則會針對模組的另一部分 Stream，以及其運作流程與影像擷取的方式進行討論。

4.3.2 SVIP Stream

Stream 為 tmVcapSvip 模組中用來處理影像串流設定的部分，建立在標準的 TSSA 架構之上，因此 Stream 這部分具備了 OL 層、AL 層以及 DL 層的執行函式，運作的程序與一般模組相同，由上而下的函式呼叫以及資料結構設定。但由於 tmVcapSvip 模組的輸入訊號是來自 DSP 晶片的外部，所以在模組輸入端這部分的設定上會與一般模組不同，這裡會藉由設定硬體層上 SVIP 暫存器的值來操作影像的輸入。接下來就先探討 SVIP 上的結構，再來了解 Stream 這部分在函式上如何運作，並且是如何從 SVIP 取得

影像。

在 DSP 上的兩組 SVIP，每一組都可以接收四筆影像的輸入，但是在輸入端只具有一個輸入埠，接收來自 A/D 晶片轉換後的訊號，所以當有八筆影像資料要傳入 SVIP 時，A/D 晶片必須藉由多工器將影像處理成兩筆資料送入。當 SVIP 收到輸入的資料後，必須經過一些例如解多工、影像分流的程序，才能將影像寫入記憶體中，讓後續的軟體模組可以處理影像。而在 SVIP 要處理輸入的資料時，上層必須透過暫存器的設定來控制硬體上的運作，這些控制 SVIP 的暫存器則如圖 4.14 所示。

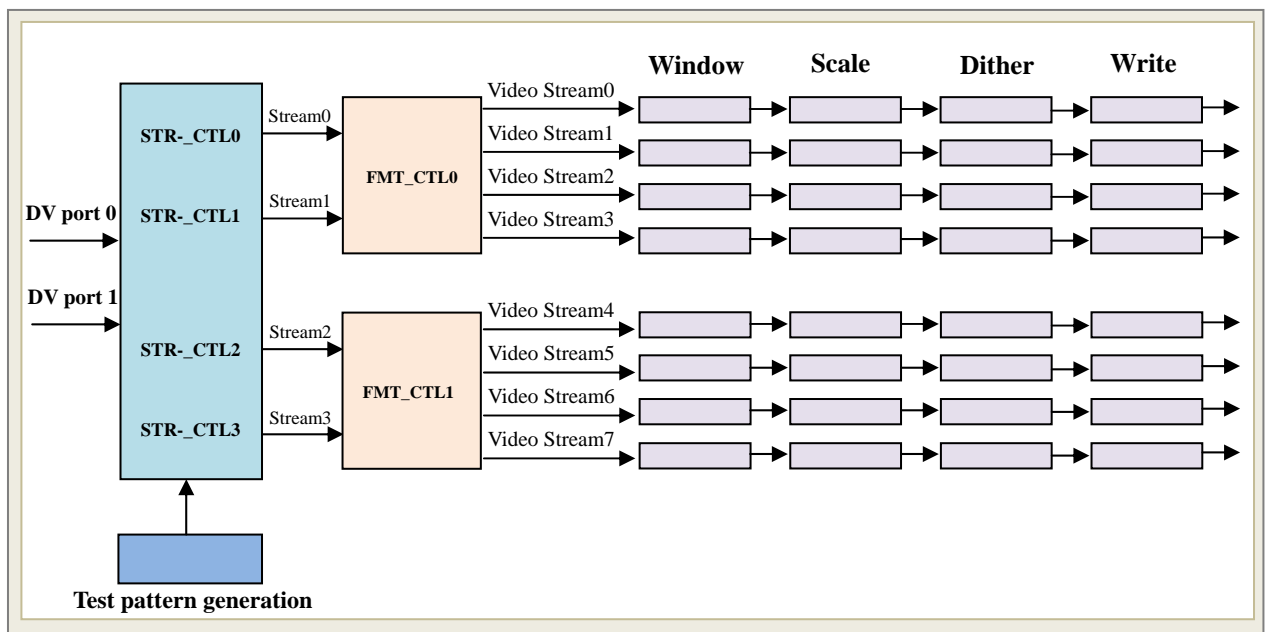


圖 4.14 SVIP 結構

處理的流程大致上可分成六個部分，首先是影像的擷取(Capture)，影像的來源有兩種，包含了由攝影機即時錄下的影像以及測試的影像樣本；當取得的影像的資料後，若為即時的影像輸入則必須進行解多工(Decode)的步驟，將多筆影像位元流從單筆的資料中獨立出來；接下來會對每一筆獨立的影像做視窗上處理(Window)，將影像分成主要與附屬的兩個視窗，但這個步驟的執行與否是由軟體層來設定控制的；而畫面在比例上的縮放(Scale)則是 SVIP 所提供的另一個功能，可以將影像的畫面尺寸縮放一半，給後續的模組進行影像處理，當影像要輸出時，在輸出端再將畫面的尺寸復原；由於 SVIP 支援八位元以及十位元的影像輸入，在這裡可以進行位元上的轉換(Dither)，將十位元轉換為八位元；最後則是寫入(Write)的動作，將擷取到的影像寫入記憶體中儲存。

而這些暫存器的控制是由 DL 層的 Instance 來設定，但由於 tmVcapSvip 模組的架構較為龐大，在 OL、AL、DL 層上的 Instance 結構組成不太一樣，所以在整個上下層的前後設定步驟會較為繁複，因此接下來就直接以 tmVcapSvip 這個模組開始運作後，如何擷取到影像的角度來探討 Instance 的設定問題。

模組的啟動是由模組中的 Start()函式來執行，所以當應用層呼叫了 OL 層的 Start()函式時，會進而連結到 AL 層的 Start()函式，而在 AL 層的 Start()函式中則會執行兩個主要的程序來開啟影像的擷取。首先必須先取得影像輸入後所要放置的記憶體空間位址，也就是去取得 tmVcapSvip 模組與下一級模組之間連結的 IOD 位址，接著會將取得的空間位址傳到 DL 層設定給 SVIP 的暫存器，讓暫存器取得記憶體的位址，如此一來在 SVIP 取得影像後，便可直接將影像寫入記憶體中；而下一個步驟則是藉由 DL 層啟動 Stream 的函式，更改 SVIP 暫存器中部分的設定值，讓 SVIP 開始執行影像擷取的動作。接下來如圖 4.15 所示，為 AL 層 Start()函式中所執行的內容。

```
tmalVcapSvipStream_Start(Int instance)
{
    .....
    svipDataOut(); //取得 tmVcapSvip 模組與下一級模組之間 IOD 的 packet 位址
    .....
    svip_SetVidPacket(); //將 packet 位址設定到 DL 層
    .....
    tmdlSvipStream_PrimaryStart(); //設定 SVIP 的暫存器，開始影像的擷取
    .....
}
```

圖 4.15 tmVcapSvip 模組 AL 層 Start()函式內容

在這部分最主要的兩個函式如上圖所框起的部分，分別為設定 packet 位址到 SVIP 暫存器的 svip_SetVidPacket()函式，以及啟動影像擷取的 tmdlSvipStream_PrimaryStart()函式。首先在 Start()函式的一開始，會藉由 svipDataOut()函式來取得影像從 tmVcapSvip 模組輸出時所要放置的記憶體位址，接下來將取得的位址傳入 svip_SetVidPacket()函式。在 svip_SetVidPacket()函式中，會將傳入的 packet 位址依照影像儲存時資料的排列方式來設定 buffer 的位址，例如影像所設定的 subtype 為 YUVSemiPlanar，則在 buffer 的設定上會分配兩個 buffer 來存放影像，一個存放影像的 Y(Luminance)部分，另一個存放影像的 UV(Chrominance)部分。svip_SetVidPacket()函式的流程內容如圖 4.16 所示，在設定好存放影像的 buffer 位址後，即可把 buffer 的位址傳入 DL 層中，讓 DL 層的函式將 buffer

的位址設定到暫存器的寫入埠(port)。

```
svip_SetVidPacket()
{
    .....
    case vdfYUV422SemiPlanar:
        tmPacket_GetBufferAddr(pPacket,0,((Pointer *)&vidBuf.buffer1));
        //buffer 1 存放影像的 Y 部分
        tmPacket_GetBufferAddr(pPacket,1,((Pointer *)&vidBuf.buffer2));
        //buffer 2 存放影像的 UV 部分
        (pStreamInfo->pSetBufAddr)ivp->dlStreamInst,&vidBuf); //將 buffer 位址傳入 DL 層
}

//在 AL 層的 Instance Setup 中已將 DL 層的 tmdlSvip_SetVidBuf()函式設定給 Instance 的成員 pSetBufAddr。
ivp->primStreamInfo.pSetBufAddr = tmdlSvip_SetVidBuf;
```

圖 4.16 svip_SetVidPacket()的函式內容

在 DL 層中負責將 buffer 位址設定到暫存器上的函式為 tmdlSvip_SetVidBuf()，這個函式在 AL 層的 Instance Setup 過程中，會設定給 Instance 結構裡的成員 pSetBufAddr，所以當 AL 層要將 buffer 位址傳入 DL 層時，只要將 buffer 位址傳給 pSetBufAddr 這個結構成員即可。而 DL 層的 tmdlSvip_SetVidBuf()函式在接收到 AL 層傳入的 buffer 位址後，則會開始將位址一一設定到 SVIP 上的寫入(Write)暫存器。

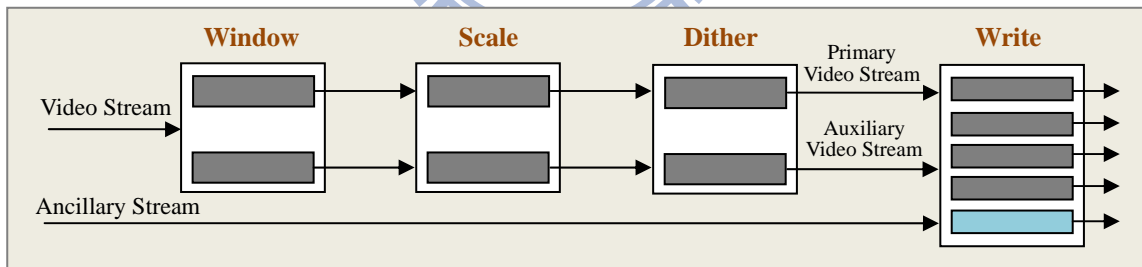


圖 4.17 SVIP 的 Video Stream 處理

由於 SVIP 會將同一筆影像資料分成 Primary Stream、Auxiliary Stream、Ancillary Stream 等三種型態的 Stream 來處理，區分出 Stream 的流程如圖 4.17 所示。首先，SVIP 由外部所接收到的影像資訊在經過 Decode 暫存器解多工後，會分出一到八筆的影像資料，視所連接的攝影機數目而定，而每一筆的影像資料都會具有 Video Stream 以及 Ancillary Stream 兩種型態的 Stream，Video Stream 即為攝影機所攝錄到的影像訊息，而 Ancillary Stream 則是儲存於影像畫面上的一些資訊。主要的 Video Stream 會再經過 Window、Scale、Dither 等三個暫存器對影像的畫面做調整，若應用層上有設定要在同

一個畫面中取得另一個大小的視窗，則 Window 暫存器會從影像上擷取出一個次要的視窗，即為 Auxiliary Stream，並與原始大小的畫面資訊 Primary Stream 一起傳送出去。

由於 Primary Stream 與 Auxiliary Stream 皆為 YUV 所組成的影像畫面，因此在寫入記憶體時需要不同的寫入埠來將 YUV 分配到所要儲存的 buffer 裡面，加上前面所提到的 Ancillary Stream 這筆資料也需要一個獨立的寫入埠來寫入 buffer，所以在 Write 暫存器中具有五個寫入埠來分別處理這些影像資料。而在一般執行的程式上，通常只會選擇 Primary Stream 這筆資料來處理，因此在寫入暫存器的設定上會將由 AL 層傳入的 buffer 位址設定到其中幾個寫入埠上，設定關係如圖 4.18 所示。

```
tmdlSvip_SetVidBuf ()
{
    // 寫入暫存器 Port 0 的位址設定為存放 Y 資訊的 buffer1 位址
    pSvipRegs->psuWrCtl[WRITE_PORT_0][streamIndex].psuWrBaseAddr.u32 = pPhysBuffer->buffer1;
    .....
    // 寫入暫存器 Port 1 的位址設定為存放 UV 資訊的 buffer2 位址
    pSvipRegs->psuWrCtl[WRITE_PORT_1][streamIndex].psuWrBaseAddr.u32 = pPhysBuffer->buffer2;
    .....
}
```

圖 4.18 Buffer 位址與暫存器的設定關係

當 SVIP 的暫存器取得影像要寫入的記憶體位址後，會回傳一個確認值給 AL 層的 Start() 函式，讓 Start() 函式可以往下繼續執行影像擷取的動作，而接下來所呼叫的函式為 DL 層的 tmdlSvipStream_PrimStart()，在這個函式裡會將暫存器中的啟動位元值設定為啟動的狀態，如圖 4.19 所示。這部分有變更到啟動位元的暫存器，包含了寫入(Write)暫存器以及擷取(Capture)暫存器，在寫入暫存器上會設定所用到的寫入埠啟動位元，開啟暫存器寫入的功能；而擷取暫存器的啟動位元一變更為啟動狀態，則會開始將外部輸入的影像擷取進 SVIP 上。

```
tmdlSvipStream_PrimStart()
{
    .....
    svipPsuWrCtl.u32 = pSvipRegs->psuWrCtl[WRITE_PORT_0][streamIndex].psuWrCtl.u32;
    svipPsuWrCtl.bits.valid0 = True; //設定寫入暫存器的 Port 0 啟動的位元
    .....
    svipPsuWrCtl.u32 = pSvipRegs->psuWrCtl[WRITE_PORT_1][streamIndex].psuWrCtl.u32;
    svipPsuWrCtl.bits.valid0 = True; //設定寫入暫存器的 Port 1 啟動的位元
    .....
    pSvipRegs->ctl.u32 |= 1; //將 Capture 暫存器的啟動位元 Enable，開始影像的擷取
    .....
}
```

圖 4.19 啟動影像擷取的暫存器設定

輸入端模組的結構參數設定以及影像擷取的啟動設定，大致上到此告一段落，而接下來為輸出端模組的討論。

4.4 輸出端模組架構

輸出端模組與輸入端模組的架構類似，都是在處理 DSP 晶片與 DSP 晶片外部連接硬體之間的訊號傳輸，所以在輸出端的架構上除了軟體層的模組外，也具備了硬體的區塊來將影像訊號從 DSP 晶片的輸出。而輸出端在架構上主要包含了兩部分，第一部分是影像從 DSP 晶片輸出前末端的軟體模組 tmVrenderGfxVo，以及此軟體模組所操控的硬體區塊 MBS(Memory Based Scale)和 QVCP(Quality Video Composition Processor)；第二部分則是影像從 DSP 晶片輸出後所銜接的訊號類比數位轉換晶片，以及用來控制此硬體區塊的軟體模組 tmVencAna，整體的輸出端架構如圖 4.20 所示。

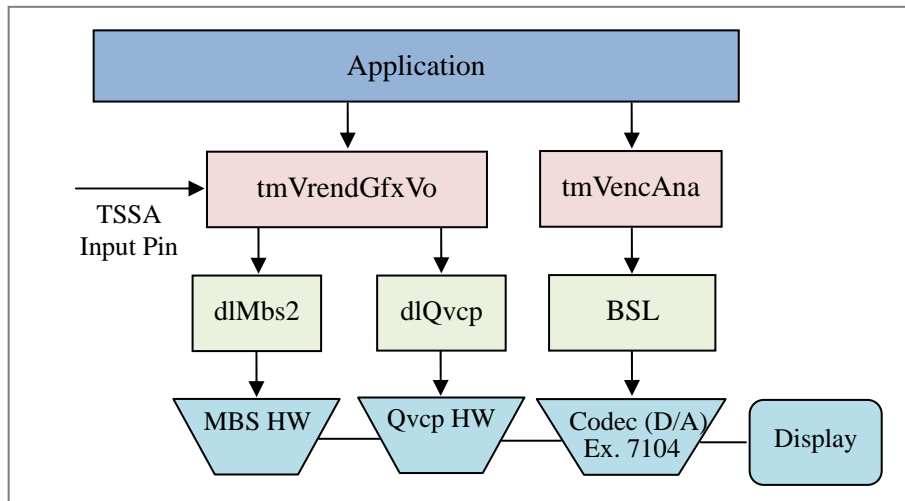


圖 4.20 輸出端分層與模組架構

輸出端在程式上的設定程序與輸入端類似，都必須要先取得與 DSP 晶片相連接的硬體資訊，再來設定軟體層的模組。因此在設定上，應用層會先藉由 tmVencAna 模組的函式來取得 D/A 晶片的資訊，而這些 D/A 晶片的資料則是在程式的一開始在對硬體初始化時就註冊到 BSL 層裡，所以當 tmVencAna 模組中的函式在取得 D/A 晶片資訊時，會直接到 BSL 層存放硬體資訊的陣列中讀取資料，並將取得的資料設定到自己的資料結構裡儲存。而 tmVrenderGfxVo 模組的設定步驟則和一般模組的設定方式類似，透過模組中的 OL 層函式來往下層設定模組的 Capabilities 以及 Instance 資料結構。

輸出端與一般模組不同處在於輸出端操控了兩組硬體區塊 MBS 以及 QVCP，而輸出端的影像資料主要也是交給這兩塊硬體來做處理，因此當輸出端模組啟動後，會至前一個模組寫入資料的記憶體區塊中拿取影像資料，並啟動硬體暫存器的控制位元，將記憶體中的資料直接交給硬體模組來處理。因此接下來就針對 MBS 以及 QVCP 這兩組硬體模組的特色來做討論。

- **MBS(Memory Based Scalar)**

MBS 這塊硬體模組主要的功能是轉換影像的格式或調整畫面的尺寸大小，MBS 模組中的函式會直接至記憶體中讀取影像，經過影像的調整之後，再寫入記憶體中儲存。而 MBS 在影像處理上功能列舉如下：

- 影像畫面的放大或縮小
- 將交織式(interlace)的畫面影像合成為整張的影像(frame)
- 顯示影像的長寬比例轉換
- 在影像取樣 4:2:0，4:2:2，4:4:4 格式上的轉換
- 色彩空間上的轉換
- 像素格式的轉換

由於影像的顯示是由輸出端模組 `VrenderGfxVo` 來控制，因此在顯示條件上若有用到 MBS 所提供的影像處理功能，則會在 `VrenderGfxVo` 模組中呼叫 MBS 的函式來做處理。除了輸出端模組會使用到 MBS 硬體區塊外，在 NDK 的程式資料庫中具有一個的 TSSA 架構的軟體模組 “`tmVtransMbs2`” 用來控制 MBS 這塊硬體，而 `tmVtransMbs2` 模組在程式上的執行順序，通常是位於影像擷取模組與編碼模組之間，對於要編碼的影像做一些畫面上的調整。

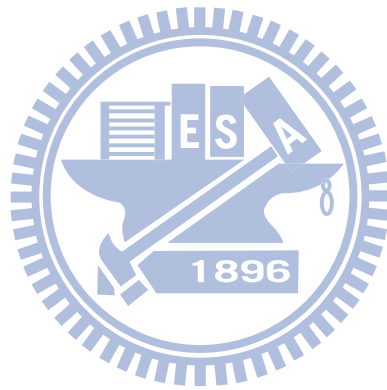
- **QVCP(Quality Video Composition Processor)**

QVCP 是 DSP 晶片上用來處理影像顯示的硬體單元，也是影像訊號在輸出 DSP 晶片前所必經的一個處理單元，是由上層的軟體模組 `tmVrenderGfxVo` 透過模組 DL 層的函式來設定執行內容。QVCP 主要提供了一個高解析度的影像輸出功能，影像的類型除了攝錄到的影片外，還可以輸出磁碟中的圖片影像，因此在 QVCP 中會藉由分層器(layers)來處理不同來源的影像，以及組合器(mixers)來將不同的資料合成

在同一個顯示畫面上。而 QVCP 所提供的影像處理功能列舉如下：

- 色彩的補償
- 反向伽瑪(gamma)值的校正
- 修正影像品質，例如讓色彩的明亮度(luminance)更明顯、色度濃度(chrominance)的改善、膚色的校正、藍色成分的延伸、綠色成分的加強……等
- 水平軸上的比例的修正，讓影像的全景能夠呈現在螢幕上
- 影像顯示於不同螢幕上(例如 SD-TV、HD-TV)的顯示條件控制
- 降低雜訊的成分

當 MBS 與 QVCP 影像處理完畢後，便會將資料直接傳送給 D/A 晶片，將數位訊號轉換為類比訊號，接著將影像輸出於螢幕上顯示。



第五章 實作驗證

本章首先會先針對模組與模組之間的傳輸方式進行探討，延續第三章對傳輸模組 IOD 的描述，藉由實際應用程式(exH264Codec.c)的執行，來測試模組之間在傳輸資料時，如何操作 IOD 上的序列與 packet，並驗證傳輸模組的溝通方式；接下來則會以另一個應用程式(exSvipDeinterlace.c)來測試，在此應用程式中所使用到的影像處理模組架構較為特殊，因此會先對該影像處理模組進行介紹，並探討其資料的傳輸方式，接著會在此影像處理模組上新增一個人臉偵測的功能函式，並載至 NXP 的 PCI Lite Demo 板上測試是否能順利執行。在章節的順序上，5.1 節會以第一個應用程式的測試結果，來對 IOD 的序列以及 packet 的傳輸關係進行討論；5.2 節中則會討論第二個應用程式上模組的傳輸架構，並對加入的程式片段進行說明。

5.1 傳輸數據驗證

模組之間溝通主要是透過傳輸模組 IOD 中的 packet 來傳遞資料，在第三章對傳輸模組的探討中有提到，當模組要讀取或寫入資料時，會將 IOD 結構成員上所存放的序列編號傳入到 OSAL 層上，到存放所有序列資料的陣列中尋找該編號所代表的序列，再藉由 psos 的系統函式來取得該序列上的 packet。其中 IOD 上傳輸的序列可分為兩種：full queue 以及 empty queue，在 full queue 上傳送的是寫有資料的 packet，也就是 packet 上所記錄的 buffer 記憶體區塊中存有影像資料；而 empty queue 上傳送的則為空的 packet，代表該 packet 所記錄的 buffer 記憶體區塊中沒有影像資料。假設目前系統上使用到 A、B、C 三個影像處理模組，如圖 5.1 所示，在兩兩模組之間會有一個 IOD 來做資料傳輸，因此在這個系統上會存在兩個 IOD。其中每個 IOD 上所傳輸的序列都會具備一組序列編號，以數字(1)、(2)、(3)、(4)……來標示序列，此編號是在序列的建立過程中，依照序列建立的前後順序所標示上的，會記錄在 IOD 的結構成員結構成員 full queue 以及 empty queue 中(序列的建立過程在第三章的 2.2 小節中有詳細討論)。因此在模組傳輸資料的過程中，當模組要讀取影像資料時，就會將連結於模組輸入端 IOD 所記錄的序列編號傳入系統中，來取得儲存於序列上 packet 的資訊，藉由 packet 上所記錄的 buffer 位址進而取得影像資訊，例如圖 5.1 中的 Component B，輸入的影像資訊會存放在 IOD 1 的序列上，所以當 Component B 要讀取資料時，就會將 IOD 1 上所記錄的序列編號傳入系統中，由系統函式來取得該序列上所儲存 packet；同樣的當模組要將資料傳遞給下一

級模組時，就會藉由模組輸出端所連結 IOD 上記錄的 packet 資料，來取得影像資料所要寫入的記憶體區塊位址，例如圖 5.1 中的 Component B 在輸出影像的時候，就必須將輸出端 IOD 2 上的序列編號傳入系統中，取得屬於此 IOD 的記憶體區塊來寫入資料。

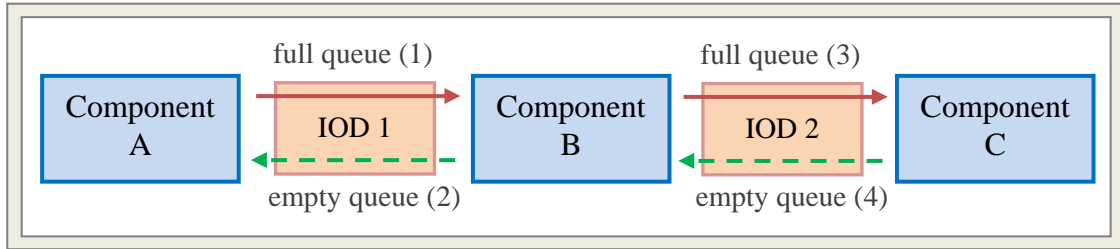


圖 5.1 模組之間的傳輸架構

在 IOD 上的序列主要是記錄了模組之間傳送的 packet 資訊，包括 packet 的資料結構位址、以及標示該 packet 處理程序的旗標等，在序列上會以一個大小為 16 - byte 的陣列 (Int[4]) 做為傳輸單位，來儲存上述的 packet 資訊。而在一個序列上可以傳送的傳輸單位數量，為該序列所屬的 IOD 上存在的 packet 數再加上 32，其中 32 為程式上的預設值，當應用程式中沒有定義 packet 數時，系統就會自動為序列分配 32 個單位來傳送 packet 資料，此外屬於同一個 IOD 的 full queue 以及 empty queue 會具備相同數量的傳輸單位。而在序列傳輸單位中所儲存的 packet 資料結構位址，則是記錄了 packet 的結構參數設定，這些參數是在 IOD 建立 packet 的過程中，依照應用程式所給定的條件，例如 packet 數量、packet id、buffer 數量、buffer 尺寸大小……等資訊來設定的。在設定好 packet 的結構參數之後，系統上會再根據 buffer 尺寸的設定值，到實體記憶體上分配記憶體區塊，並將 buffer 所分配到的記憶體位址設定到 packet 的結構參數中，讓 packet 上能夠記錄影像資料所存放的位址，整個 IOD、packet、buffer 的連結關係如圖 5.2 所示。

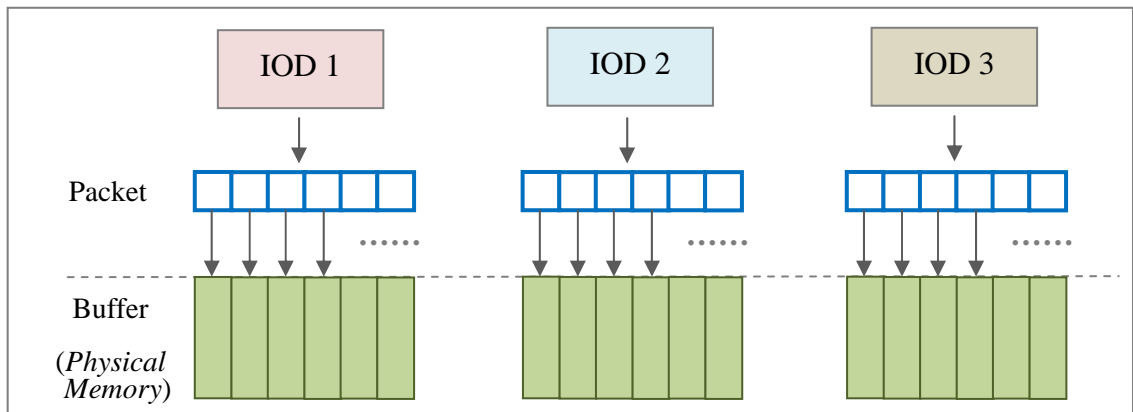


圖 5.2 IOD、packet、buffer 的連結關係

在每個 IOD 中都會有屬於自己的 packet，這些 packet 裡就會記錄著 buffer 的位址，也就是分配給該 IOD 的記憶體區塊，因此在每一個 IOD 中都會具備一塊實體記憶體區塊，提供給前後連結的模組來讀寫資料。而連接於 IOD 輸入端的模組，會固定將處理完畢的影像資料寫入此記憶體區塊中，連接於 IOD 輸出端的模組，則會固定到此記憶體區塊中來讀取資料。這些 packet 與 buffer 的資訊記錄在 IOD 的兩個序列上，序列與 packet、buffer 之間的連結關係如圖 5.3 所示。

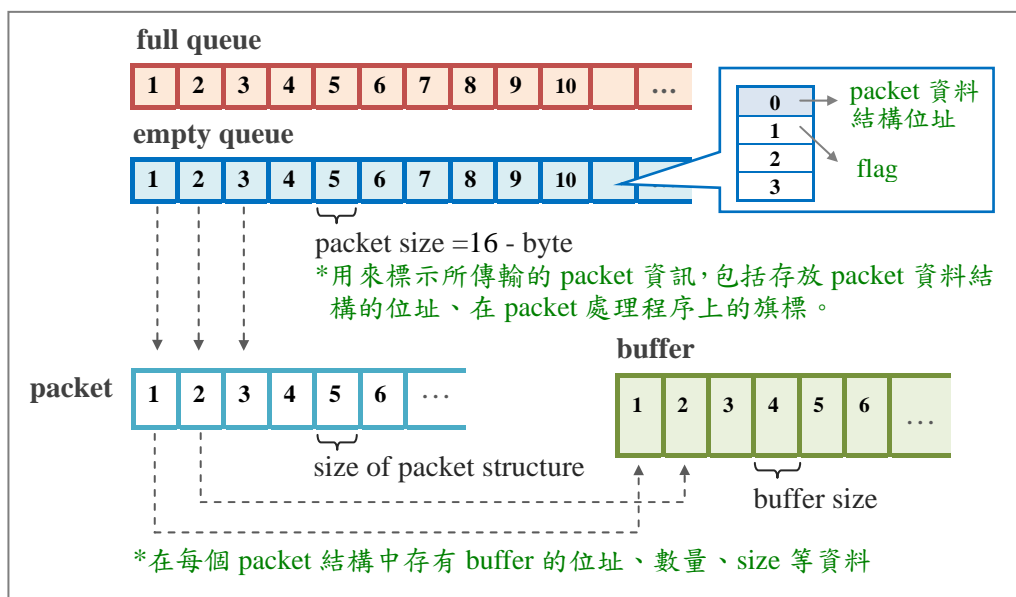


圖 5.3 queue、packet 與 buffer 的結構關係

透過這樣的 queue、packet、buffer 結構關係，在系統上首先會將建立好且尚未儲存資料的 packet 結構位址依序設定到 empty queue 中，如此一來在應用程式開始執行前，所有 packet 都會放置在 empty queue 上就緒，等待模組來拿取並寫入資料。一旦程式開始運作後，模組就會不斷至 empty queue 中拿取空的 packet 來寫入影像資料，當模組把影像資料儲存到 packet 所指定的記憶體區塊後，接著就會將此 packet 的資料結構位址填到 full queue 上，也就是將此 packet 標示為已使用的部分，這樣下一級的模組就可以直接到 full queue 上拿取 packet，並由 packet 取得影像所儲存的位址。當下一級模組讀取完影像資料後，就會將該 packet 所指定記憶體區塊中的資料清空，並將此 packet 的資料結構位址填到 empty queue 上，也就是標示為未使用的 packet，繼續提供給前一級的模組來寫入資料。

因此在這部分便以 NDK6.1 所提供的應用程式“exH264Codec.c”來進行測試，觀

察 queue、packet 以及 buffer 之間的關係。在“exH264Codec.c”這個應用程式中，定義了六個影像處理模組，因此在這些模組之間總共會使用到五個 IOD 來做資料的傳輸，表 5.1 即為在程式上為這五個 IOD 定義的傳輸單位的設定值，包含了 queue 的名稱、代號、packet 數量、在序列上所能傳輸的 packet 數 (Queue size)、以及 buffer 的尺寸大小。

表 5.1 IOD 的傳輸單位設定

	Queue	Queue size	Buffer size
IOD_1	Full Queue – Name [FS2M] , No. 0 Empty Queue – Name [ES2M] , No.1	Packet 數= 12 Queue size = 12 + 32 = 44	345600
IOD_2	Full Queue – Name [M2EF] , No. 2 Empty Queue – Name [M2EE] , No.3	Packet 數= 15 Queue size = 15 + 32 = 47	345600
IOD_3	Full Queue – Name [E2RF] , No. 4 Empty Queue – Name [E2RE] , No.5	Packet 數= 8 Queue size = 8 + 32 = 40	426496
IOD_4	Full Queue – Name [E2DF] , No. 6 Empty Queue – Name [E2DE] , No.7	Packet 數= 25 Queue size = 25+ 32 = 57	1048576
IOD_5	Full Queue – Name [DRF0] , No. 8 Empty Queue – Name [DRE0] , No.9	Packet 數= 19 Queue size = 19+ 32 = 51	773376

由於建立好 packet 在模組開始運作前，皆為未使用的狀態，並存放於 empty queue 中，因此就藉由這個時機印出了每個 IOD 上 empty queue 所儲存的資料，也就是 IOD 中所有 packet 的資料結構位址。表 5.2 中則分別列出了三個 IOD：IOD 1、IOD 2、IOD4 中前五個 packet 與 buffer 的位址資訊。

表 5.2 packet與buffer位址

IOD & Queue No.	Empty Packet順序	Packet資料結構位址	Buffer [1]	Buffer[2]
IOD 1 & Queue.1	1	-645249612	-657570432	-657224704
	2	-657699012	-656878976	-656533248
	3	-645249480	-656187520	-655841792
	4	-657698880	-655496064	-655150336
	5	-645249348	-654804608	654458880
IOD 2 & Queue.3	1	-645248428	-645247360	差距=345728，為 IOD 1 所定義的 buffer size : 345600+128
	2	-657696560	-644555904	
	3	-645248296	-643864448	-643518720
	4	-657696428	-643172992	-642827264
	5	-645248164	-642481408	642135680
IOD 4 & Queue.7	1	-657692176	-618216064	差距=345728，為 IOD 2 所定義的 buffer size : 345600+128
	2	-657692060	-617167360	
	3	-618218760	-616118656	差距=1048704，為 IOD 2 所定義的 buffer size : 1048576+128
	4	-657692024	-615069952	
	5	-618218564	-614021248	

觀察表 5.2中的數據，可發現在buffer位址也就是記憶體位址的分配上，兩兩差距皆為該IOD所定義的buffer尺寸大小再加上128 - byte，在IOD 1以及IOD 2中buffer是用來儲存一張影像的畫面，因此大小即為影像畫面的尺寸345600 (720x480) - byte，而IOD 4由於是接在Encoder模組之後，因此傳輸的buffer數量只有一個，用來傳送壓縮過的資料封包，buffer的尺寸大小則為壓縮過後的影像封包大小。而這些目前存放於empty queue上的packet的資料結構位址，則會在接下來模組開始運作後，拿來與同一個IOD上full queue所記錄的packet資料結構位址比對，測試IOD的傳送端模組在拿取empty queue上的packet來寫入資料後，是否有確實將此packet的位址標示到full queue上，而IOD的輸出端模組在從full queue上讀取packet的資料時，是否也有操作到同一個packet。

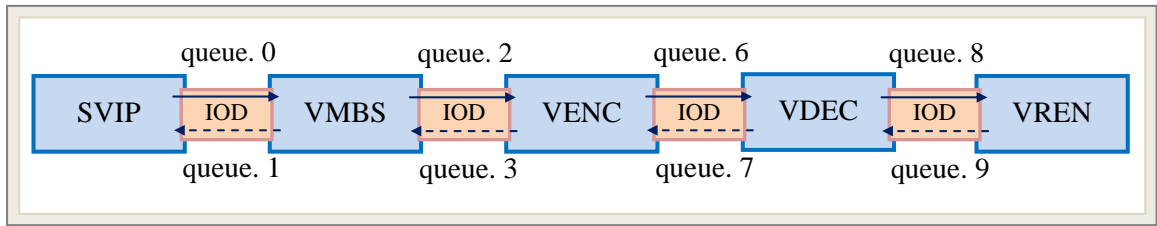


圖 5.4 應用程式 exH264Codec.c 模組架構

在此應用程式“exH264Codec.c”的執行上，由於執行的功能選擇，在程式實際的運作上只有使用到五個模組來做處理，因此模組之間只會使用到四個IOD來傳輸資料，分別為表 5.1中所列舉出來的IOD 1、IOD 2、IOD 4以及IOD 5，整體的模組架構如圖 5.4所示。而接下來則整理了此應用程式在執行上，各IOD的接收端模組從full queue上拿取的packet資料結構位址，在表 5.3裡列出了程式運作時的前十五筆數據，表格上的第一欄為拿取full packet的模組名稱，第二欄為所拿取的full queue代號，第三欄則為full queue上所儲存的packet資料結構位址。而在程式運作上的前十五筆數據中，只有使用到程式架構上的前三個IOD，代表資料尚未傳入到第四個IOD上，因此在這部分就以前三個IOD的資料來比對，在表格的右方列出了在IOD 1、IOD 2以及IOD 4的empty queue中所儲存的packet資料結構位址，即在上一頁表 5.2的第三欄(Packet 資料結構位址)中所列舉的資料。

在這部分對照圖 5.4的模組架構關係與表 5.3的傳輸資料，首先傳輸順序上第一個拿取資料的模組為“VMBS”，該模組會從IOD 1的full queue中來拿取“SVIP”模組處理完畢的packet的資料，其中IOD 1的full queue的編號為“0”，因此接著觀察表格上Full Queue No.等於0的packet資料結構位址，來與同樣屬於IOD 1的empty queue (No.1) 上所記錄的packet資料結構位址比對，可以發現兩者的數據是一致的，代表在IOD 1上的full queue與empty queue所傳輸的是同一個packet，也就處理同一個記憶體區塊所儲存的影像資料。而接下來當“VMBS”模組做完影像處理後，會由IOD 2的empty queue (No.3) 上拿取packet，將資料寫入屬於IOD 2的記憶體區塊中，並在寫入後將packet的位址標示到IOD 2的full queue (No.2) 上，提供給下一級的模組“VENC”來讀取，因此比對模組“VENC”從full queue上拿取的packet資料結構位址，以及empty queue中儲存的packet資料結構位址，可以發現兩者所記錄的數據是一致的，代表傳送同一筆影像資料。同樣的在“VENC”模組處理完影像資料後會將資料存放到IOD 3裡，而下一級的模組“VDEC”就會到IOD 3的full queue上讀取packet，因此觀察IOD 3裡的full queue

(N0.6) 以及empty queue (No.7) 所儲存的packet資料結構位址，可以發現兩者的數據也會一致。藉由以上三組IOD的傳輸資料，可以了解到模組之間的資料傳輸方式，並驗證了packet在full queue以及empty queue上的循環傳輸機制。

表 5.3 packet位址的數據比對

Task name	Full queue No.	Packet資料結構位址
VMBS	0	-645249612
VENCH264	2	-645248428
VMBS	0	-657699012
VMBS	0	-645249480
VMBS	0	-657698880
VENCH264	2	-645248296
VDEC	6	-657692176
VMBS	0	-645249348
VMBS	0	-657698748
VENCH264	2	-645248164
VDEC	6	-657692060
VMBS	0	-645249216
VMBS	0	-657698616
VENCH264	2	-645248032
VDEC	6	-618218760

Empty queue. 1 Packet資料結構位址
-645249612
-657699012
-645249480
-657698880
-645249348

Empty queue. 3 Packet資料結構位址
-645248428/512
-657696560/513
-645248296/514
-657696428/515
-645248164/516

Empty queue. 7 Packet資料結構位址
-657692176
-657692060
-618218760
-657692024
-618218564

5.2 模組實驗

在這部分利用NDK6.1軟體開發包所提供的應用程式“exSvipDeinterlace.c”來進行測試，在該應用程式中使用到了三個影像處理模組，包含了輸入端模組tmVcapSvip、影像處理模組tmProcessInPlace、以及輸出端模組tmVrendGfxVo。其中人臉偵測的功能函式會附加於tmProcessInPlace模組中，但由於該模組是屬於一個In-Place結構的模組，在packet資料的傳送與一般模組有部分的差異，因此在5.2.1小節中，會先針對In-Place模

組的運作方式來進行討論；而5.2.2小節，則會說明如何在模組中附加上人臉偵測的功能函式。

5.2.1 In-Place模組結構

In-Place模組的傳輸架構如圖 5.5所示，不同於一般模組的傳輸機制，當In-Place模組處理完Source端模組傳入的packet資料後，在將packet資料交給Sink端模組的過程中，不會去拿取empty queue上的空packet來寫入資料，而是直接將該packet的資料結構位址寫入到full queue上的傳送單元裡儲存，並直接將此packet傳送給Sink端模組，交由Sink端模組來處理。而當Sink端模組處理完packet的資料後，則會將packet上的資料清空並且傳回給In-Place模組的Source端，提供給Source端模組繼續寫入資料。

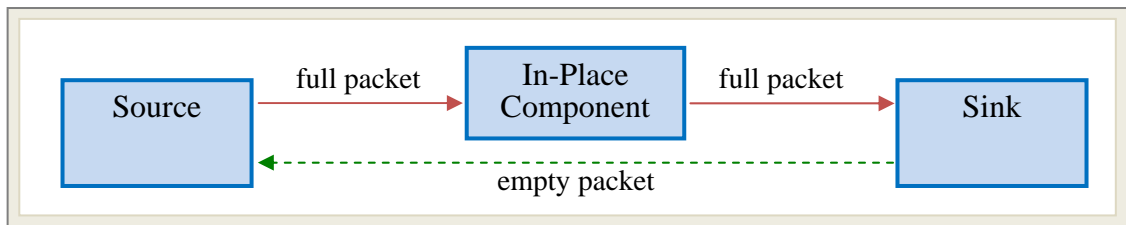


圖 5.5 In-Place 模組結構

因此在這部分同樣會藉由對queue上所儲存的packet資料結構位址轉換，來觀察In-Place模組的運作方式，而除了測試In-Place模組packet拿取以及寫入的序列外，還會對連接於In-Place模組輸出端的下一級模組歸還packet的部分做驗證。圖 5.6為應用程式“exSvipDeinterlace.c”的模組架構，其中“PROCESS”為一個In-Place結構的模組。在程式的執行上，一開始同樣會在兩兩模組之間建立IOD，並且為所有的IOD建立full queue以及empty queue序列，因此在本應用程式裡會存在兩個IOD以及四個序列，而在packet數量的設定上，則會設定讓In-Place模組的輸入端與輸出端數量一致。表 5.4為在模組運作前，兩個IOD上empty queue所儲存的packet資料結構位址。

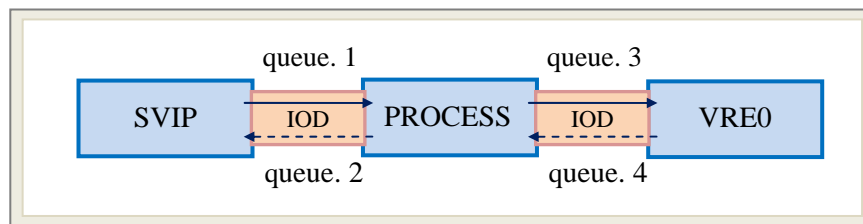


圖 5.6 應用程式 exSvipDeinterlace.c 模組架構

在本應用程式裡，將傳送的packet數量由原本程式上設定的12個變更為6個，因此在下表列出了兩個empty queue上儲存的所有packet資料結構位址。

表 5.4 empty序列上儲存的packet資料結構位址

IOD & Queue No.	Empty Packet順序	Packet資料結構位址	IOD & Queue No.	Empty Packet順序	Packet資料結構位址
IOD 1 & Queue.2	1	-649179748	IOD 2 & Queue.4	1	-649132472
	2	-649149592		2	-649149328
	3	-649132888		3	-646730612
	4	-649073808		4	-649132340
	5	-649149460		5	-649149196
	6	-649132756		6	-646730480

而接下來在表 5.5 中，整理了“exSvipDeinterlace.c”這個應用程式中的模組開始運作後，前二十筆full queue與empty queue上的packet傳輸數據。首先觀察full queue上packet資料結構位址傳遞的部分，可發現模組“PROCESS”從IOD 1上的full queue (No.1) 拿取的packet資料位址，與IOD 1上empty queue (No.2) 所存放的資料結構位址相同，代表兩者處理的是同一個記憶體區塊的資料，這部分如同前小節描述的內容，與一般模組的運作方式相同；而接下來模組“vRE0”到IOD 2上的full queue (No.3) 所拿取的packet資料結構位址，與IOD 1上所傳輸的packet資料結構位址相同，代表模組“vRE0”與模組“PROCESS”所處理的是儲存於同一個記憶體區塊的資料。這說明了模組“PROCESS”在處理完影像資訊後，不會使用IOD 2的empty queue上所存放的packet來寫入資料，也就是不會去操作到IOD 2所分配的記憶體區塊，而是直接將IOD 1的packet資料結構位址寫到IOD 2的full queue中，讓IOD 2上儲存的記憶體區塊與IOD 1相同。接著下一級模組“vRE0”就會根據packet上所記錄的記憶體位址，到同一個記憶體區塊中拿取影像資料。

接著觀察模組之間empty queue上packet資料結構位址的傳遞狀況，可發現當模組“vRE0”從full queue No.3上取得packet的資訊後，會將該packet放回屬於IOD 1的empty queue (No.2)上。這說明了將packet的資料清空以及歸還到empty queue上的動作是由“vRE0”模組處理，“PROCESS”模組在運作上不會去操作到empty queue的部分，只

單純負責full packet的讀寫操作。

表 5.5 In-Place模組的packet傳輸數據

Task name	Full queue No.	Empty queue No.	Packet 資料結構位址
PROCESS	1		-649149592
PROCESS	1		-649132888
vRE0	3		-649149592
PROCESS	1		-649073808
vRE0	3		-649132888
PROCESS	1		-649149460
vRE0	3		-649073808
vRE0		2	-649149592
PROCESS	1		-649132756
vRE0	3		-649149460
vRE0		2	-649132888
PROCESS	1		-649179748
vRE0	3		-649132756
vRE0		2	-649073808
PROCESS	1		-649149592
vRE0	3		-649179748
vRE0		2	-649149460
PROCESS	1		-649132888
vRE0	3		-649149592
vRE0		2	-649132756

Empty queue. 2 Packet資料結構位址
-649179748
-649149592
-649132888
-649073808
-649149460
-649132756

5.2.2 模組實驗結果

當了解In-Place結構的運作方式後，接下來就在tmProcessInPlace模組中加入一個人臉偵測的功能函式來做應用。因此首先必須找到tmProcessInPlace模組主要的運作函式，將人臉偵測的函式加入到此運作函式裡。而這個部分，tmProcessInPlace模組主要是在

Instance資料結構中，定義了一個稱為modifyPacket的結構成員，該成員的型態為函式指標，在設定tmProcessInPlace模組Instance資料結構的過程中，會將模組上影像處理的執行函式ModifyPacket()設定到此結構成員上，因此當tmProcessInPlace模組開始運作後，就可藉由函式指標modifyPacket連結到ModifyPacket()函式來做影像資料的處理。所以這部分就將人臉偵測的函式就掛載在ModifyPacket()函式裡，並在應用程式執行的功能選單上，增加了人臉偵測的選項，當使用者想要執行人臉偵測的功能時，便可在應用程式開始執行前下達人臉偵測的指令，接下來tmProcessInPlace模組會在執行ModifyPacket()函式內容的過程中，進而連結到人臉偵測的功能函式，來進行人臉偵測的影像處理。

```

ModifyPacket (Pointer handle, ptmPacket_t packet)
{ //取得 packet 中所儲存的 buffer 位址
  tmPacket_GetBufferAddr (packet, 0, (Pointer *) &luma); //luma 值
  tmPacket_GetBufferAddr (packet, 1, (Pointer *) &chroma); //chroma 值
  .....
  if (ivp->face) // 人臉偵測的運算部分
  { // 分配記憶體位址來暫存影像處理的資料
    RGBset=(unsigned char*)calloc(imageWidth*imageHeight*3,sizeof(unsigned char*));
    dst1=(unsigned char*)calloc(imageWidth*imageHeight,sizeof(unsigned char*));
    dst2=(unsigned char*)calloc(imageWidth*imageHeight,sizeof(unsigned char*));

    for(ypos = 0; ypos < imageHeight; ypos++) // YUV→RGB 轉換
    {
      for(xpos = 0; xpos < imageWidth; xpos++)
      {
        RGBset[ypos*imageStride*3+xpos*3] =..... //B
        RGBset[ypos*imageStride*3+xpos*3+1] =.....//G
        RGBset[ypos*imageStride*3+xpos*3+2]=.....//R
      }
    }
    // 將轉換成 RGB 的影像資料傳入人臉偵測的函式中
    RGBset=FaceDetect(RGBset,imageWidth,imageHeight,100,100,1.1);

    for(ypos = 0; ypos < imageHeight; ypos++) // RGB→YUV 轉換
    {
      for(xpos = 0; xpos < imageWidth; xpos++)
      {
        dst1[ypos*imageStride+xpos] =.....//Y
        dst2[ypos*imageStride+xpos] =.....//U
        dst2[ypos*imageStride+xpos] =.....//V
      }
    }
    memcpy(luma,dst1,lInUse); // 將運算的結果複製回 buffer 裡儲存
    memcpy(chroma,dst2,cInUse);
  }
}

```

圖 5.7 ModifyPacket()中加入的程式內容

而在ModifyPacket()函式中增加的程式內容如圖 5.6所示，當要執行ModifyPacket()函式時，模組必須傳入包括了模組的Instance資料結構指標以及所要處理的packet資料位址等資訊。當判斷出應用程式中有要執行人臉偵測這部分的功能時，就會進而連結人臉偵測函式FaceDetect()來對影像做處理。但由於在FaceDetect()函式中的影像資料是以RGB的型態來處理，因此在呼叫FaceDetect()函式之前，會先對影像資料的型態做轉換，將YUV轉換為RGB的型態，接著才將資料傳入FaceDetect()函式中；而在FaceDetect()函式處理完畢後，傳回來的值會再經過一次的反轉換，將影像資料轉回YUV的型態，並將處理完的資料存放回buffer中。

其中在呼叫人臉偵測函式FaceDetect()時，必須要傳入六個參數值提供給該函式在運算上的設定，這六個參數分別為影像資料的指標、影像的長寬值、人臉偵測視窗的初始長寬值、以及偵測視窗在每次運算上的加大比例；而這部分則以傳輸的影像長度為720寬度為480，偵測視窗的長寬初始值皆等於100，視窗加大比例等於1.1的參數設定值傳入。圖 5.7即為在tmProcessInPlace模組中加上人臉偵測功能後，於“exSvipDeinterlace.c”應用程式的執行結果。



圖 5.8 人臉偵測實驗結果

第六章 結論

本論文以 DSP 晶片 PNX1005 上執行的程式為探討的重點，研究如何藉由軟體架構 - TSSA 來建立程式上的影像處理模組，以及在一個模組中所必須具備的資料結構內容與運算處理函式，並探討模組如何透過分層之間函式的連結來運作執行。而模組與模組之間的資料傳輸溝通，則是在 TSSA 軟體架構上另一個所要探討重點，這部分會藉由實際測試模組在運作時 queue、packet 以及 buffer 的資料傳輸，來驗證模組之間的傳輸溝通機制。因此在實驗的章節上，整理了模組之間傳送的封包測試數據，並在 In-Place 影像處理模組中附加上人臉偵測功能，測試模組在運作時的執行流程，以及了解如何對各模組來變更或新增其影像處理功能。

在模組化的架構上，每個模組都是藉由相同的運作機制來執程式，因此每個模組只需要提供出自己的設定資料以及函式介面，系統即可透過相同的方式來操作各模組。藉由這樣的模組化結構，各模組在設計上可省略掉許多由系統負責執行的部份，並依照預設的架構來編輯影像處理程式，如此一來便能簡化模組在程式編輯上的複雜度。而在執行模組的人臉偵測過程中，影像處理的運算速度較為緩慢，除了功能函式本身在程式上的設計問題外，這部分還必須考量到硬體環境與軟體程式在執行上的最佳化設定，包含了在操作該硬體環境時可以使用的特定指令、以及程式在編輯時可使用的特定語法、或者在記憶體使用上特定的區塊分配……等設定，以便讓應用程式在運作時能達到較高的執行效率。

透過此 TSSA 軟體架構，不論是建立一個新的影像處理模組，或者是將影像處理的功能函式附加到既有的模組中來執行，都能夠提供給系統在程式上更清晰的管理，並且讓應用程式在模組的使用上，能藉由簡單的介面函式來操作多元的影像處理模組。

參考文獻

- [1] The Architecture of TSSA1, vol. 1, Koninklijke Philips Electronics N.V., July 2009.
- [2] TSSA1-Classic APIs, vol. 3, Koninklijke Philips Electronics N.V., July 2006.
- [3] Video IO Components, Koninklijke Philips Electronics N.V., July 2009.
- [4] Video Codec Components, Koninklijke Philips Electronics N.V., July 2009.
- [5] PNX100X Series Data Book, vol. 1 of 1, Rev. 1.2, Koninklijke Philips Electronics N.V., July 2009.
- [6] pSOSyste System Calls, Integrated Systems ,Inc., January 1999.
- [7] PSOS documentation, vol. 8, Koninklijke Philips Electronics N.V., October 2008.
- [8] Infrastructure Subsystem APIs, vol. 2, Koninklijke Philips Electronics N.V., December 2008.
- [9] Booting and Board Support APIs , vol. 5, Koninklijke Philips Electronics N.V., December 2007.
- [10] Iain E.G. Richardson, H.264 and MPEG-4 Video Compression, John Wiley & Sons, Ltd., England, 2003.
- [11] Ze-Nian Li, Mark S. Drew, Fundamentals of Multimedia, Pearson Prentice Hall, America, 2004.
- [12] YUV <http://en.wikipedia.org/wiki/YUV>.
- [13] P.J.Deitel , H.M.Deitel 著，C 程式設計藝術，五版，陳心璋，陳大任譯，台灣培生教育，台北，民國九十七年。
- [14] 蘇琮壹，『串流模組間的協商與傳輸機制研究』，國立交通大學，碩士論文，民國 99 年。