國 立 交 通 大 學

電信工程研究所

碩 士 論 文

影音播放程序模組化與模組的搜尋機制

Modulize the process of playing the media file and search modules

研 究 生：游盛如
指導教授：張文鐘 博士

中 華 民 國 九十八年八月

影音播放程序模組化與模組的搜尋機制

Modulize the process of playing the media file and search modules

研 究 生：游盛如　　Student: Sheng- Ru Yuo
指導教授：張文鐘　博士　　Advisor: Dr. 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 Communication Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
In

Communication Engineering

August 2009

Hsinchu, Taiwan, Republic of China

中 華 民 國 九十八 年 八月

影音播放程序模組化與模組的搜尋機制

研究生：游盛如　　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

摘　要

VLC 是由 multi-thread 的架構所組成，利用每一個 thread 運作和資料分享的方式來建立起一個完整的播放影片系統，例如:解多工器模組和解碼器模組就是在各自在不同的 thread 下工作，但是解碼器模組會不斷讀取由解多工器模組得到的 audio bitstream 及 video bitstream 來進行解碼程序。

而多媒體播放器在播放影音檔案的過程中，主要會面臨到的問題就是如何尋找並使用合適的模組播放影音檔案和影音播放程序的模組化，所以在 VLC 中有許多重要的函數會提供影音檔案的訊息並存入結構資料中，並依據這些結構資料所存的影音檔案訊息來找到合適的模組。

所以在本論文中我們藉由研究 VLC 這整個 open source，來了解多媒體播放器的基本播放架構以及了解 VLC 是如何尋找並使用合適的模組。還會更進一步比較 VLC 和 FFmpeg 之間的不同。

Modulize the process of playing the media file and search modules

Student: **Sheng- Ru** Yuo Advisor: Dr. Wen-Thong Chang

Institute of Communication Engineering
National Chiao Tung University

Abstract

The structure of VLC is combined by multi-thread, through every thread worked and shared with data to build a multimedia player system, take example:demultiplexer and decoder worked in different thread, the decoder continually accessed audio bitstream and video bitstream from the demultiplexer and worked the process of decoding.

During playing the media file with the media player, the main problem is that how to search for appropriate modules to display the media file and modulize process of playing the media file, there are many important process that stored the information in data of structure, according the information about media file in the data of structure to find appropriate modules.

The topics of this thesis mainly let us understand the basic structure of the playing procedure of the media player and analyse how to search for the appropriate modules, besides compared with the difference between VLC and FFmpeg.

誌謝

我能夠順利完成此篇論文，最要感謝的人是我的指導教授 張文鐘 博士。老師在我二年的碩士生涯中，不管在學業或生活上都給予了相當多的幫助與指導，尤其在整理論文以及準備口試的期間中，難為老師如此辛苦與花費許多時間為我指出研究中的盲點和問題的關鍵。同時也感謝黃仲陵教授、范國清教授以及何文楨教授於口試的指導，有了您們的指導才使得此篇論文能更趨於完整。

另外要感謝的是實驗室裡的同學們，包含志偉、弘達、夸克、小瘋、秉謙，這兩年來大家一起修課，一起討論功課，在實驗室大家一起讀書寫報告趕論文的日子裡，感謝他們陪我度過這些日子，讓我有難忘的回憶。另外，還要感謝的是怡如、明穎、雅嵐、耀葦、實驗室有你們這些學弟和學妹們，讓實驗室裡的氣氛更加合諧和歡樂。

最後，我要感謝我的父母親、妹妹、哥哥和其他好友們，以及我的女友一直默默的支持與鼓勵我，總是可以讓我能沒有顧忌地從事研究工作，並且在我遇到挫折與低潮時適時的給我很大的鼓勵，再一次的謝謝大家。

誌於 2009.08 風城 交大

Sheng- Ru

目錄

圖目錄

第一章 序論

1.1 研究背景與動機

由於現代的科技不斷的進步多媒體影音播放的技術也不斷的進步以及逐漸地廣泛的應用在娛樂方面，各式各樣的影音檔案包裝格式和編碼格式也越來越多，然而不同的影音的檔案格式其編碼方式也是不同，所以可能就需要使用特定的媒體播放器來播放特定影音檔案，例如當我們想要觀看檔案包裝格式為「.rm」或者「.rmvb」是的影音檔案，就可能需要使用有支援「.rm」或者「.rmvb」多媒體播放器才可以播放。所以如果針對不同的影音格式都需要不一樣的播放器來播放影音檔案的話，那麼對於使用者們來說是非常不方便的。

因此在最近幾年來，陸陸續續出現了一些支持許多影音格式的媒體播放器，例如在本論文所要研究的 VLC(VideoLAN Clinet) 媒體播放器，還有其他像 KMPlayer(K-Multimedia Player) 媒體播放器、MPlayer(The Movie Player)媒體播放器和微軟本身的(Media- Player)等等，這些播放器的共同特色都是具有強大的解碼功能，以及可以觀看各種檔案格式的影片，讓使用者們只需使用一種播放器就可以觀看各種不同檔案格式的影片，

而一般媒體播放器的播放程序基本架構大多是依據影片編碼的格式來尋找一組合適的解多工器(demultiplexer)將影音資料分離出音訊位元流(audio bitstream)及視訊位元流(video bitstream)，然後再尋找一組合適的解碼器(Decoder)分別對 audio bitstream 及 video bitstream 來進行解碼，最後再藉由 audio output 和 video output 將解碼完後的影像跟聲音分別輸出到使用者的螢幕及喇叭。

所以我們可以了解到多媒體播放器在播放影音檔案的過程中，主要討論的問題和機制，就是一個多媒體播放器是如何尋找並使用合適的解多工器來分解影音檔案的包裝格式，以及尋找合適的解碼器來針對影像和聲音的編碼格式進行解碼的動作，其中也包括如何將整個影音播放程序模組化，並藉由模組和模組之間的機制建立起整個影音播放架

構。

因為不同的影音包裝格式就要由不同的解多工器來分解影音資料，同樣地不同的影像或聲音的編碼格式就是要由不同的解碼器來解碼。如果一個播放器包含了許多的解碼器和解多工器，如果不能夠有好的機制來確實的找到並使用合適的解多工器及解碼器的話，那麼還是可能造成播放器無法順利的播放影音檔案。所以在本論文中我們會藉由研究 VLC 媒體播放器這個 open source 所提供的程式碼，來了解媒體播放器的基本播放架構以及了解 VLC 是如何尋找並使用合適的解多工器以及解碼器。以及 VLC 中 thread 的機制，另外也會介紹 VLC 模組外掛的過程。我們還會另外比較 VLC 和 FFmpeg 模組的定義和模組尋找的機制做一個比較和整理。

1.2 論文章節提要

本論文會針對 VLC 整個播放流程做一個詳細的介紹，其中包括 VLC 在 windows 下影像輸出以及其串流功能都會做一個介紹。第二章是在說明 VLC 基本的工作原理。第三章主要是針對 VLC 中對模組設定的重要的基本函數做一個詳細介紹。第四章再說明解多工器和解碼器模組之間是如何溝通與傳遞資料。第五章則是會針對 VLC 和 FFmpeg 這兩個不同的多媒體播放器的模組機制做一個比較和說明。第六章則是在說明 VLC 中外掛新的模組流程。

第二章 VLC 多媒體播放器機制介紹

在第二章中我們會簡單介紹一個播放器的基本架構，藉由基本架構來了解一個多媒體的檔案是如何在 VLC 播放器中運作。本章還會介紹 VLC 播放器 thread 的機制，也會說明 VLC 是如何利用 thread 的機制來完成多媒體檔案的播放，以及 thread 的重要性和必要性。

2.1 多媒體播放器之架構

一個多媒體播放器的播放流程架構就(如圖 2.1)所示，在一開始執行時一般會有 GUI 介面的產生，然後依照我們所選擇影音資料來源去存取資料。一般來說一個多媒體影音檔案都是由許多的 bitstream 所組成的，然而這些 bitstream 都是事先經過編碼的視訊位元流(video bitstream)和音訊位元流(audio bitstream)，最後在經過多工器把這些視訊位元流和音訊位元流封裝起來。所以當我們要播放多媒體影音檔案時我們就必須利用一個適合的解多工器將封裝的資料分離出 audio bitstream 及 video bitstream，然後再使用適合的解碼器分別對 audio bitstream 及 video bitstream 來進行解碼，最後藉由 audio output 和 video output 將解碼完後的影像跟聲音分別輸出到螢幕及喇叭，因此解多工器和解碼器在多媒體播放器中扮演了不可缺少的重要角色。而在 VLC 播放器中也是如此。

我們將以圖 2.1 的架構為基礎來討論 VLC 播放器整個播放影片的流程和架構，其中包含解多工和解碼器模組的尋找機制以及其開啟條件的比對機制，並會探入討論模組與模組之間的資料和訊息的傳遞機制等等。在之後的章節我們都會有詳細的介紹。

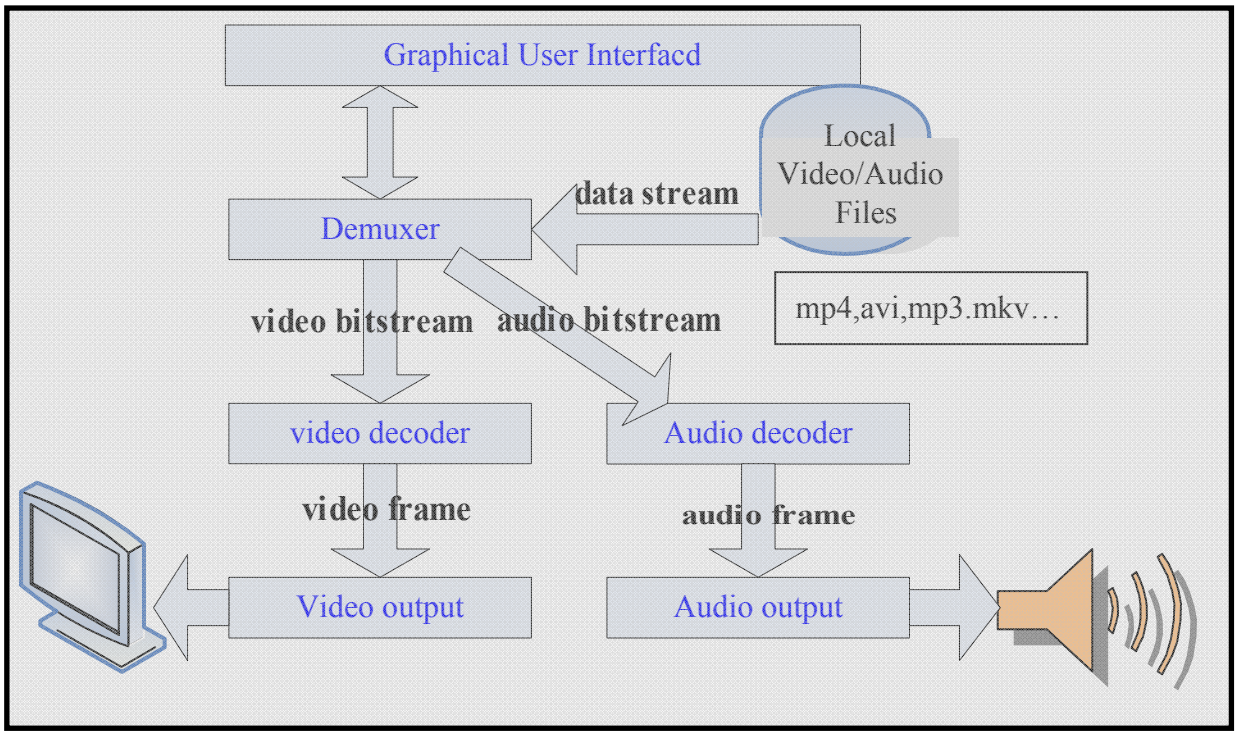


圖 2.1 多媒體播放器架構

2.2 VLC 播放器之 thread 架構

VLC 是由 multi-thread 的架構所組成，利用每一個 thread 運作和資料分享的方式來建立起一個完整的播放影片系統，例如:解多工器模組和解碼器模組就是在各自在不同的 thread 下工作，但是解碼器模組會不斷讀取由解多工器模組得到的 audio bitstream 及 video bitstream 來進行解碼程序。

當我們開啟 VLC 時，就會出現一個 GUI 的介面，此時的 VLC 就會同時啟動 4 個 thread; 每個 thread 都有自己的程序要執行，其中以 playlist thread 最為重要，因為它會不斷的等待使用者下達播放影音檔案的指令，是啟動 VLC 播放程序的重要關鍵點。上述的解多工器模組和解碼器模組的 thread 也要由 playlist thread 正式接收到使用者需要開啟影音檔案的訊息後才會分別的開始啟動。

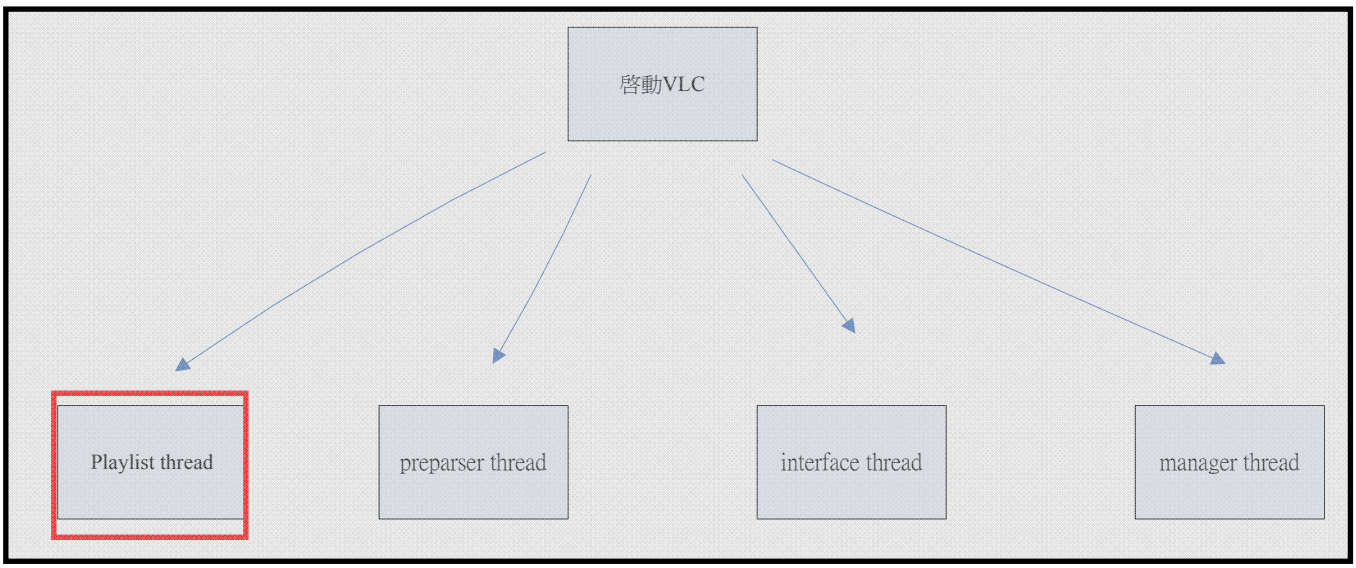


圖 2.2 VLC multi-thread 的架構示意圖

資源擷取(Access)、解多工器(demuxer)、解碼器(decoder)等等模組的尋找和基本的參數設定都會在 Input thread 下的 Run()函數中完成，不過解多工器模組是在 Input thread 下 MainLoop()函數中被呼叫並開始對影音資料進行解多工的動作，這時 MainLoop()函數是不斷的利用 callback function 的方式來呼叫解多工器模組，然後再經過層層的呼叫來尋找解碼器的模組，接著就會各自產生影像和聲音解碼的 thread。此時 Audio Output 和 Video Output thread 分別會在影像和聲音的解碼器模組中被呼叫並開始運作。

MainLoop()函數會利用 while()迴圈持續作影音資料解多工的動作並將資料放入 FIFO Buffer 中，而 Audio Dcoder 和 Video Dcoder thread 也會在 DecoderDecode()函數中利用 while()迴圈不斷讀取 FIFO Buffer 中已經連結串列好的影音資料來進行解碼的動作，並開始開啟各自的輸出模組，上述詳細的過程會在後面章節一一的介紹。

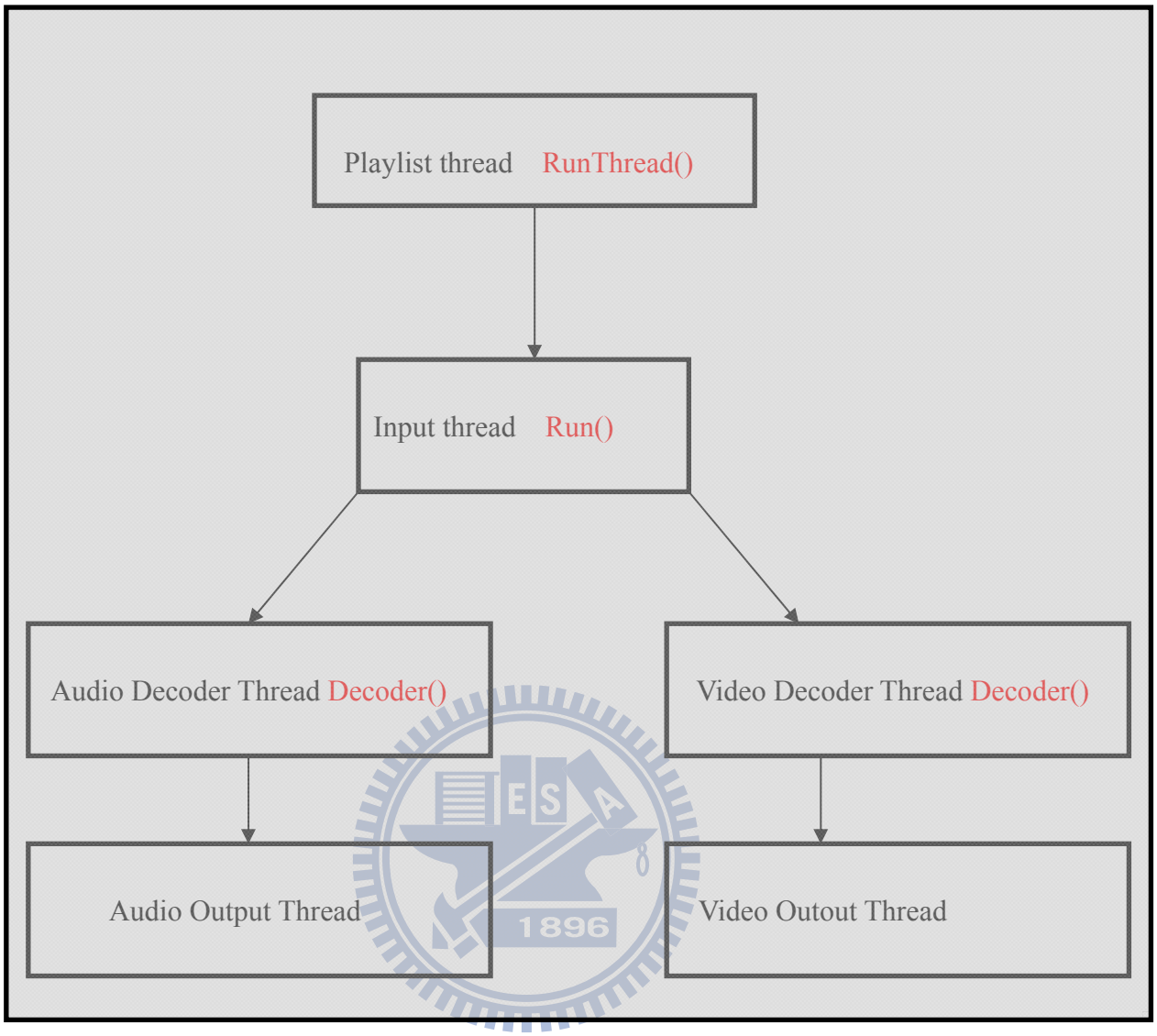


圖 2.3 VLC Thread 的關係示意圖

✧ **playlist thread**：

主要是檢查目前播放的狀態和處理使用者的指令，例如是否要開始播放影音資料或者停止 目前所播放的影音資料。若確定有影音資料要播放，那麼就會開啟 input thread 來開始進行整個播放影音的程序。

✧ **input thread**：

主要是先尋找合適存取模組以及解多工模組，然後將所讀取的影音資料解多工成 audio bitstream 及 video bitstream。而在這裡也會尋找合適的聲音和影像的解碼器模組，並分別開啟 audio decoder thread 和 video decoder thread。

✧ **audio decoder thread**：

負責將解多工後的 audio bitstream，進行聲音的解碼程序；而進行解碼的程序前，會先尋找合適的聲音輸出模組並開啟 audio output thread。

✧ **video decoder thread**：

負責將解多工後的 video bitstream，進行影像的解碼程序；而開始進行解碼的程序之前，會先尋找合適的影像輸出模組並開啟 video output thread。

✧ **audio output thread**：

負責將解碼後的聲音輸出到使用者的喇叭上。

✧ **video output thread**：

負責將解碼後的影像輸出到使用者的螢幕上。

2.3 VLC 播放器之 thread 原理

Thread 是一種可以讓一個 process 同時的執行 2 個或多個的 functions，但是使用 Thread 時需要注意許多地方。拿同時執行 2 個 functions(function A; function B)來說;所謂 Thread 的機制是讓 CPU 同時去平均去執行 2 個 functions，若此時不用 Thread 的方式去執行，而是用執行完 funcion A 後再去執行 function B 的話，從 CPU 的執行效率和執行時間來說，或許 Thread 的方式不會有特別好的 CPU 的執行效率，甚至會有比較差的情形出現。但是我們要注意的是我們目前是以 CPU 的執行效率的角度去看，如果我們是以程式〝執行結果效能〞的角度來看的話，CPU 所執行效率就不等於程式執行結果的效能。

拿一個多媒體影片的解多工 function 和解碼 function 來說，我們可以用先不斷將影片執行解多工程序後，最後在全部的解多工後的資料一次全丟給執行解碼的程序去進行解碼後在進行輸出。但是這種方式並不符合我們影片播放程式〝執行結果〞的效能，因為不能使用者執行播放影片功能後，還要等待影片全部解多工後再等全部解碼後，才能看到影片，這樣不能達到 real time 的〝執行結果效能〞就算 CPU 的執行效率很好也並非是我們所需的。反觀 Thread 的機制，可以讓影片播放程式一邊的對影片解多工又可以一邊的做影片的解碼動作然後輸出。這樣當使用者執行播放影片功能後就可以馬上看到影片，這樣才有 real time 的效果，也才達到我們〝執行結果效能〞。此時的 CPU 的執行效率就不是我們所在意的議題。

在我們設計程式時，要考慮一個 Process 所 Create 起來的 Thread 愈多時，然後又會有很多的 Threads 同時在執行的時候，所耗費的 CPU 效能相對的當然愈高，所以我們在設計程式時應當注意的是，不是總共所有的 Threads 所耗費的 CPU 效能，而是在 Multi-Thread 的程式設計的架構下，我們所期望得到的程式〝執行結果效能〞是不是我們所期待的，所以我們可以得到一個結論就是 CPU 所費的效能不等於程式執行結果的效能。

所以在 VLC 中要使用 Thread 的機制是因為解碼器模組解碼時必需要有由解多工器

模組所分離的 audio bitstream 或 video bitstream 才能進行解碼的動作，所以在考慮〝執行結果效能〞也就是 real time 的效果下，使用 Thread 的機制在多媒體播放器的程式設計上是必要的。

下面的圖 2.4 我們可以很清楚地知道不使用 thread 機制的 R.T 會很長;R.T 很長是在多媒體播放器中是我們最不想見到的〝執行結果效能〞，然而圖 2.5 使用 thread 機制可以大大的減少 R.T 的時間，雖然整體來看解多工和解碼的程序所需的時間變長，但是使用者能感覺到的只有 R.T 的長短，所以在這個時候 CPU 所費的效能在多媒體播放器中就不是我們所在意的了。

圖 2.4 在單 CPU 執行下沒使用 thread 機制的 Response Time

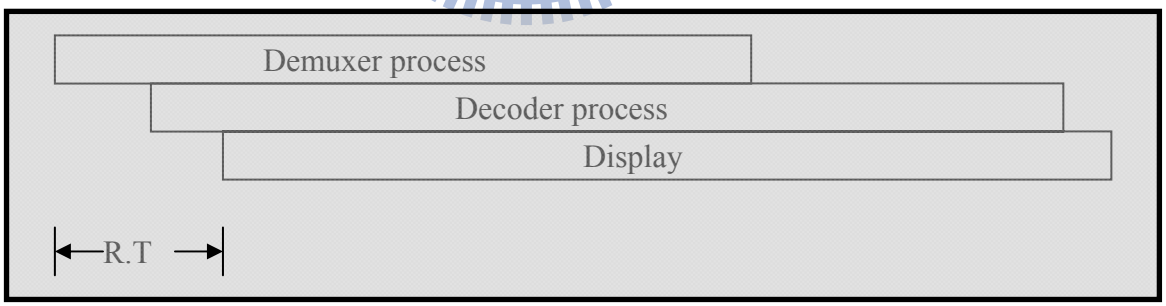


圖 2.5 在單 CPU 執行下使用 thread 機制的 Response Time 圖

第三章 VLC 播放程式流程架構

在本章中我們會針對 VLC 從開啟一個 GUI 介面後到尋找基本模組重要的流程做一個簡單的介紹，在其過程中會呼叫許多在 Input thread 下執行的重要函式，這些函數都是 VLC 中的核心函數，因為這些函數若不能順利執行的話那就代表發生錯誤當然 VLC 也就無法繼續執行下去。所以我們會對這些重要的函數一一做說明。其中我們會去深入討論 module_Need()函數，因為在第二章中我們知道一個多媒體影音檔案必須經過一個解多工器來將其分離成影像流和聲音流，然後再依據其編碼格式尋找到適合的解碼器模組，所以 module_Need()函數就扮演著在 VLC 中找到適合的模組的角色。

3.1 VLC 播放影音程序

接著我們來看 VLC 大致上從主程式(vlc.c)到開始執行解多工程序的基本流程，我們會將基本流程分成三大步驟來作一個討論。下圖 3.1 只是一個大概的流程，因為其中還是有呼叫其他的和函數，然而我們會將重點放在 Init()和 MainLoop()上這兩大函數上，因為這兩大函數是 VLC 播放多媒體資料的核心函數，若這兩大函數在初始化或執行過程中出了問題，整個 VLC 就會無法繼續工作下去。

Init()函數是用來尋找解多工程序的模組資訊和參數的初始化。MainLoop()函數則是當資源存取和解多工器模組都準備就緒後，就會開始呼叫解多工器模組來將影片分離成影像流和聲音流，然後就會開始尋找各自的解碼器模組。下面我們就針對每一個步驟來詳細討論。

Step1:

是我們剛開始啟動 VLC 時就會先做初始化的動作然後接著立即產生四個主要的 thread，來建構起整個 VLC 播放器的架構。其中 playlist thread 下的 Run Thread()函數的功能是不斷等待使用者下撥放多媒體影音資料指令。

Step2:

主要是在使用者下達播放多媒體影音資料指令給 VLC 播放器後(如開檔，做串流等等)，就會開始檢查使用者下達何種指令並作適當的處理。同時也會對 Run Thread 下的許多變數作處理。

Step3:

當使用者決定要開啟多媒體檔案或串流功能時，此時就會啟動一個 Input thread，在 Input thread 下的 Init()函數會開始做各種模組的尋找和變數的初始化，在 Init()函數執行無誤後會在 MainLoop()函數中不斷的執行多媒體影音的解多工程序，並且會做影片的暫停、快轉、結束或連續播放動作等等的處理。下面的章節我們會做詳細介紹。

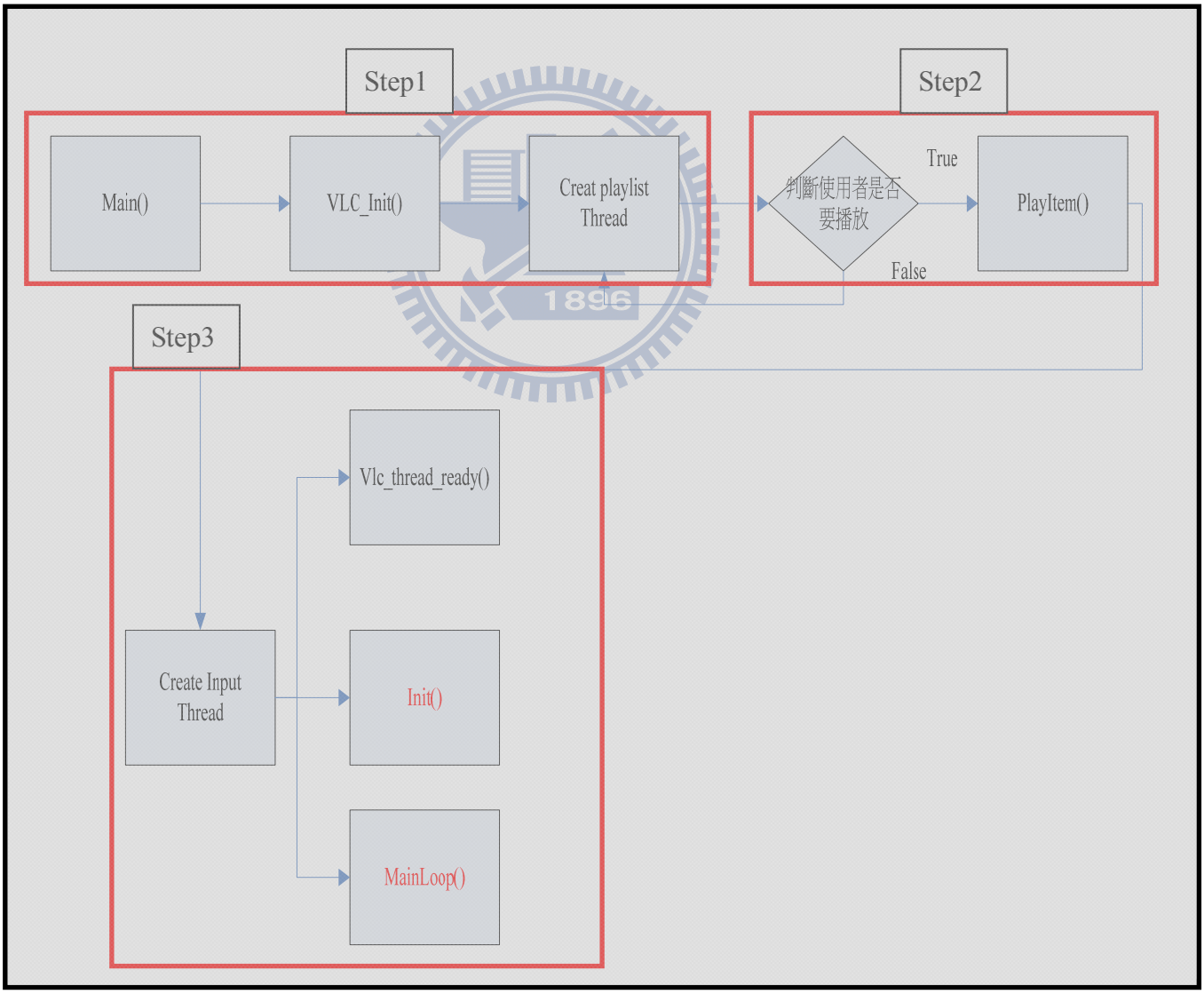


圖 3.1 主程式到尋找基本模組流程圖

3.2 VLC 重要函數說明

當我們開始對 VLC 作啟動讀檔播放資料的動作時 Run()函數就會被啟動，此時 Input thread 也會啟動，Init()函數就會被用來對型態 input_thread_t 的結構成員作設定如圖 3.2 所示; 結構 input_thread_t 就像是進行解多工程序的大資料庫，因為其中成員包含資源存取模組和解多工器模組的資訊和一些模組的控制方面的設定，其中也會呼叫解多工器模組所需的傳送資料系統模組，資料傳送系統模組是每一個解多工器模組在解析完資料後，對資料作處理和傳送時會呼叫到的模組，可以說是 VLC 中解多工器模組的公用模組，其中還會定義有關播放狀態的變數。而 MainLoop()函數則會接收由 Init()函數設定好的 input_thread_t 的結構成員並開始利用其成員的資訊來播放影片。

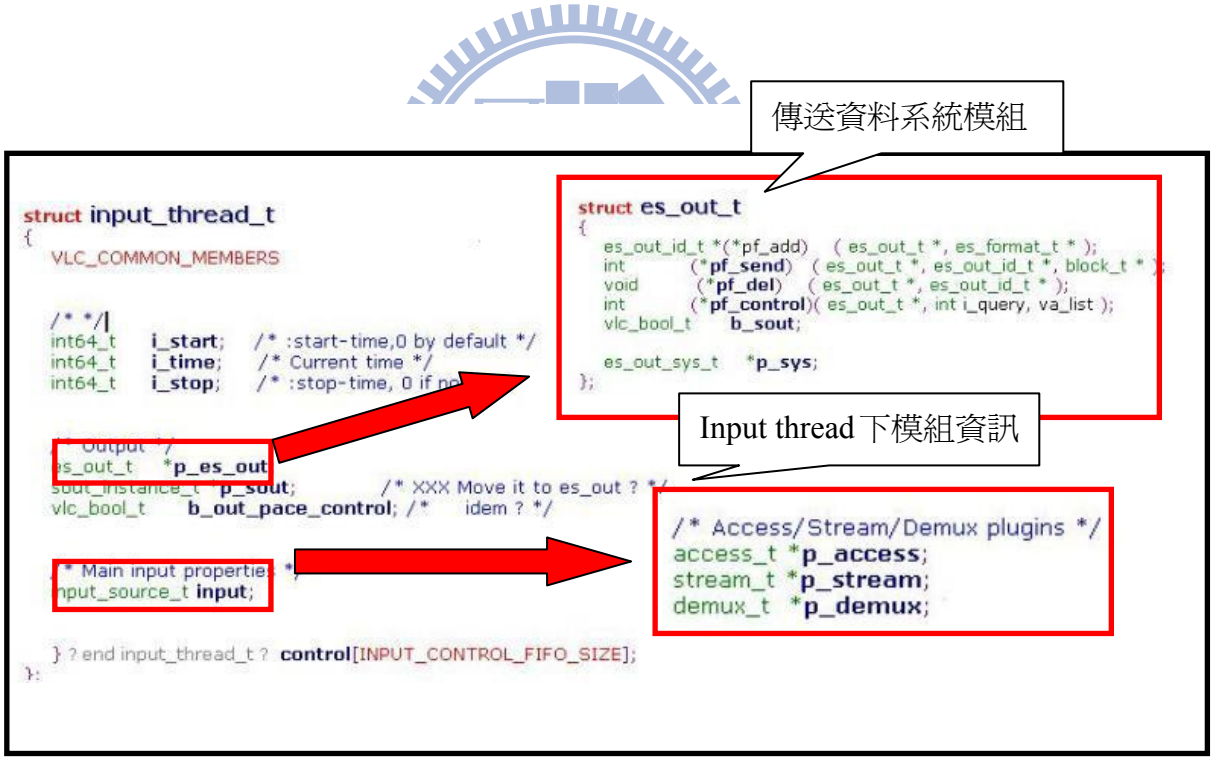


圖 3.2 input_thread_t 結構重要成員

3.2.1 Init()函數

如圖 3.3 所示，在呼叫 Init()函數時會設定一個非常重要的參數，那就是型態為 input_thread_t 的結構指標變數* p_input，因為在 input_thread_t 結構內的成員都是整個 Input thread 下所需要的變數和結構變數;其中也包含了會在 Input thread 下執行解多工程序所需的所有模組資訊，所以 input_thread_t 內的成員必須藉由 Init()函數來設定結構內所有的成員最後再將整個 input_thread_t 結構傳給 MainLoop()函數來使用。所以我們可以清楚看到在 Init()函數中所呼叫的函數和所有的設定幾乎都是在設定結構 input_thread_t 中的成員。所以 Init()函數是用來判斷 VLC 是否能播放我們所選取的多媒體檔案的重要函數，下面會說明 Init()函數內一些重要的函數。

```
static int Init( input_thread_t * p_input, vlc_bool_t b_quick )
 {.....
 /* Create es out */
 p_input->p_es_out = input_EsOutNew( p_input );
 es_out_Control( p_input->p_es_out, ES_OUT_SET_ACTIVE, VLC_FALSE );
 es_out_Control(p_input->p_es_out,ES_OUT_SET_MODE, ES_OUT_MODE_NONE );
.....
      if( InputSourceInit( p_input, &p_input->input,
                           p_input->input.p_item->psz_uri, NULL, b_quick ) )
```

圖 3.3 Init()函數設定示意圖

在圖 3.3 中的 input_EsOutNew()函數就是用來設定結構 input_thread_t 的成員，這函數會用一個型態為 es_out_t 的結構指標變數*out 來設定解多工器模組進行解多工程序時所必需的資料輸出系統模組，資料輸出系統模組就像是公用的模組;每個解多工器模組都會透過資料輸出系統模組中的函數來傳送資料。如圖 3.4 所示，我們可以很清楚地看到 input_EsOutNew()函數就是為了設定 p_input->p_es_out 所呼叫的函數。

如圖 3.5 所示，我們用 input_EsOutNew()函數中的 out->pf_send = EsOutSend 來說明，out->pf_send 是 es_out_t *out 的結構指標變數內的成員之一，他會指向一個 EsOutSend()函數並接收 3 個參數，要注意的是這個時候只是做一個指向的動作，指向的

函數 EsOutSend()但並不會立即執行，之後會我們會發現 MPEG-2 program stream 的 ps.c 解多工器模組中所定義的解多工函數就會呼叫 es_out_Send()函數來傳送 audio bitstream 或 video bitstream 給解碼器模組進行解碼程序。

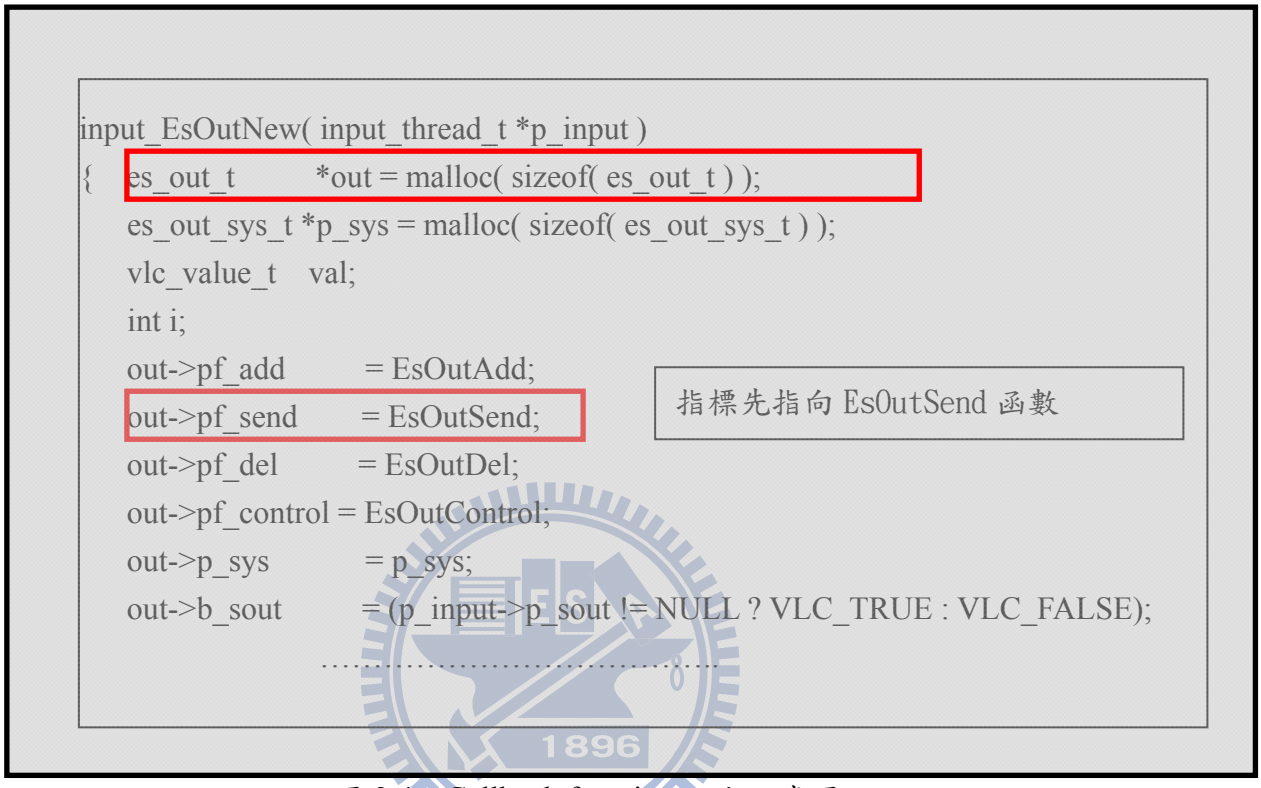


圖 3.4 Callback function 設定示意圖

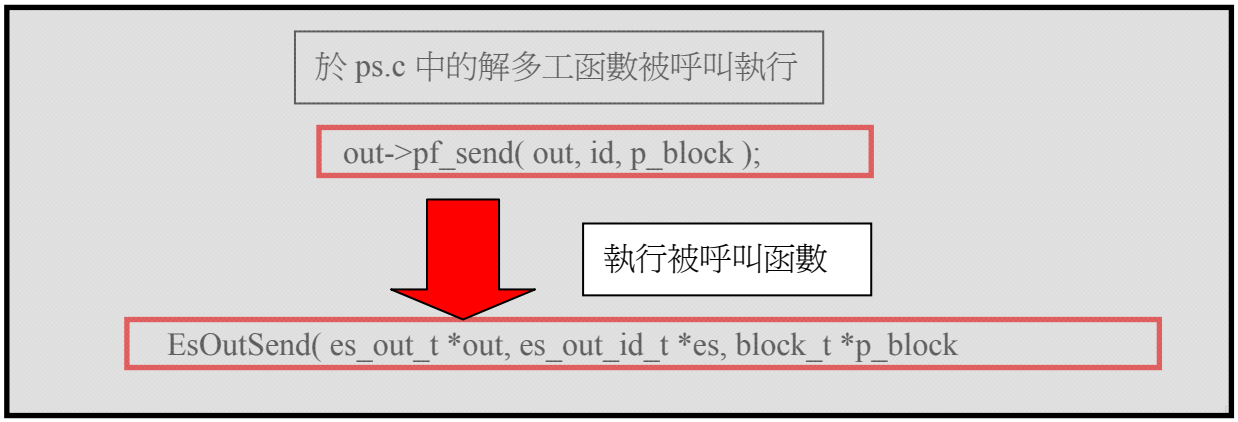


圖 3.5 呼叫 callback function 過程示意圖

```
struct es_out_t
{
es_out_id_t *(*pf_add)      ( es_out_t *, es_format_t * );
int          (*pf_send)     ( es_out_t *, es_out_id_t *, block_t * );
void         (*pf_del)      ( es_out_t *, es_out_id_t * );
int          (*pf_control)( es_out_t *, int i_query, va_list );
vlc_bool_t    b_sout;
es_out_sys_t  *p_sys;
}
```

圖 3.6 資料傳送系統模組成員示意圖

如圖 3.7 所示，我們可以很清楚地看到資料傳送系統模組內定義了許多 callback functions 的回傳以及其傳入參數的型態，其中定義 callback functionu (*pf_add)時多一個符號〝*〞是因為其回傳是型態是 es_out_id_t 的指標變數。

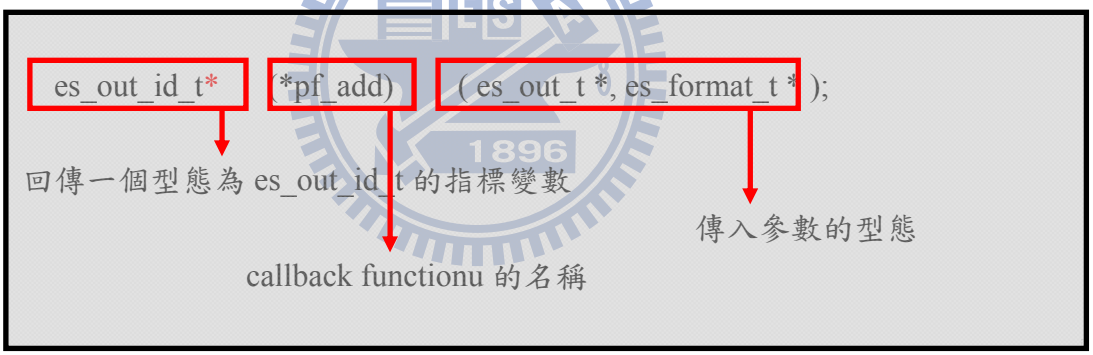


圖 3.7 callback function 結構的示意圖

3.2.2 InputSourceInit()函數

VLC 在開啟一個多媒體檔案時的來源有可能是從網路，也有可能是電腦硬碟內的檔案。然而檔案又可以分成很多不同的格式，常見的多媒體檔案格式包括 MPEG-1、MPEG-2 外，另外還有 Real Video、QuickTime、WMV、MPEG4 及 H.264…等格式，在這麼多的多媒體檔案格式中，VLC 是如何找到適合的資源存取、解多工器、解碼器這修重要的模組呢?下面我們會說明 VLC 是如何尋找適合模組的重要函數 module_Need()。

InputSourceInit()這個函數會根據我們多媒體資料的來源和封包格式來找到進行解多工程序時所需的資源存取和解多工器解碼器的模組。下面會說明 InputSourceInit()函數中一些重要的函式。在此函數會傳入 2 個重要參數，一個當然是 input_thread_t 型態的指標變數*p_input，另一個是型態 input_source_t 的結構指標變數*in;其成員定義了資源存取、解多工器模組的資訊，圖 3.8 所示;而結構 in 也是結構 p_input 中的一個成員。為了找到適合的模組在 InputSourceInit()函數中會呼叫 module_Need()函數並找到適合的模組並儲存其位址，必需注意的是在這個 Input thread 下會找到資源存取、解多工器、解碼器這三個主要模組，然而真正會在 Input thread 下執行動作的只有資源存取、解多工器這兩個模組，解碼器模組則是在 Decoder thread 下執行。下面我們會說明 InputSourceInit()函數中有那些重要的函數。

```
access_t    *p_access;
stream_t    *p_stream;
demux_t     *p_demux;
```

圖 3.8 input_source_t 部分成員

1. MRLSplit():

   主要用來對我們所選擇的影音資料的來源做分析，如圖 3.9 所示，MRLSplit()函數先分解 MRL(Media Resource Locator)的 access、demux、path 這 3 個部分，並分別儲存在 psz_access、psz_demux、psz_path 這些字串指標，如圖 3.10 所示，這些資訊也會存在解多工器模組中並會在訊息視窗列印出來。

```
MRLSplit( VLC_OBJECT(p_input), psz_dup,
                &psz_access, &psz_demux, &psz_path );
```

圖 3.9 MRLSplit()函數參數

圖 3.10 MRLSplit()函數的訊息

2. access2_New():

如圖 3.11 所示，是用來尋找適合資源存取模組，由於 VLC 的檔案來源可能是網路也有可能是從電腦的資料夾中的檔案，而在截取網路資源的時候可能會有不同的 Data rate 或是網路協定，所以須要不同的資源存取模組來開啟不同來源的檔案。其中會利用 module_need()函數就是來尋找適合的模組，若 module_need()函數沒找到適合的資源存取模組的話就代表我們檔案的來源有問題，當然整個 VLC 就無法繼續播放資料。在後面我們會針對 module_need()這個重要的函數來討論。

```
/* Now try a real access */
in->p_access = access2_New( p_input, psz_access, psz_demux, psz_path,
                            b_quick );
```

圖 3.11 access2_New()函數

以開啟電腦中檔案的例子來說，前面的 MRLSplit()函數會先找出欲開啟檔案的路徑用 psz_path 指標變數指來儲存並提供欲開啟檔案的路徑給資源存取模組。然後在 VLC 中開啟電腦中的檔案大致上可分成四種資源存取模組，分別是 Dvdnav.c、Vcd.c、Directory.c、File.c，我們會以常用的三個來討論。VLC 一開始就會依照模組分數依序用 Dvdnav.c 模組來判斷欲開啟檔案是否 DVD 格式，然後再用 Vcd.c 來判斷是否為 VCD 格式，若不為以上兩種格式最後才用 File.c 來開啟在電腦中的影音檔案。如圖 3.12 所示，File.c 模組會對要開啟檔案的路徑作是否有錯誤的檢查，只要路徑沒有錯誤就會用 File.c 模組來當作資源存取模組並開始設定開檔所需的 callback functions。

```
p_access->pf_read = Read;
p_access->pf_block = NULL;
p_access->pf_seek = Seek;
p_access->pf_control = Control;
```

圖 3.12 File.c 的 callback functions

3. stream_AccessNew():

如圖 3.13 所示，是為資源存取模組及解多工器模組之間建起一個溝通的橋樑並將讀取到的多媒體資料處理成一連串的 streams 或 blocks 的資料讀取系統模組，會利用一個型態 stream_t 結構來設定開啟資料時所需的 callback functions，所以在解多工器模組會呼叫資料讀取系統模組來取得資料並進行解多工的程序。

```
in->p_stream = stream_AccessNew( in->p_access, b_quick );
```

圖 3.13　stream_AccessNew()函數

4. demux2_New():

一部影片為了壓縮大小或其影片品質會將影像和聲音資料封裝成很多不同格式的封包(i.e avi,mp4...) ，在每一個不同格式的封包中其影像和聲音資料都有不同的 hearder

來區分，為了要將一個封包內的影像和聲音資料給分離成 video bitstreame 和 audio bitstreame，就必須靠解多工器模組來完成，如圖 3.14 所示，而 demux2_New()函數就是 VLC 中用來找適合的解多工器模組。

.

```
in->p_demux = demux2_New( p_input, psz_access, psz_demux, psz_path,
in->p_stream, p_input->p_es_out, b_quick );
```

圖 3.14 尋找解多工器模組函數

我們就先針對資源存取模組和解多工器模組之間的關係來做一些說明，若在呼叫 access2_New()函數後又找到適合的資源存取模組的情形下，就會建立起資源存取模組與解多工器模組溝通的讀取系統模組，此時若是開啟 DVD 或 VCD 檔案格式其讀取方式就是用 block 的方式來讀取資料，若是開啟 Mpeg-2 或 Avi 類型的檔案格式的話則是利用 stream 的方式讀取資料。如圖 3.15 所示，stream_AccessNew()函數中會呼叫讀取資料系統模組中的函數。

```
block_t *(*pf_block)   ( stream_t *, int i_size );
int          (*pf_read)     ( stream_t *, void *p_read, int i_read );
int          (*pf_peek)     ( stream_t *, uint8_t **pp_peek, int i_peek );
int          (*pf_control)( stream_t *, int i_query, va_list );
void         (*pf_destroy)( stream_t *);
```

圖 3.15 結構 stream_t 成員

資源存取模組和解多工器模組是在 input thread 下執行，當解多工器模組需要資料時就會呼叫資源存取模組來讀取資料。拿一個 Mpeg-2 的影片檔來說，在 MainLoop()函數的迴圈中會不斷呼叫解多工器模組來將影片分解成 audio bitstream 和 video bitstream，此時解多工器模組就會呼叫讀取資料系統模組中讀取資料的函數來呼叫資源存取模組來讀取資料流並進行解多工程序。

3.2.3 module_Need()函數

VLC 中模組的定義: 在上面幾個小節中我們知道在 VLC 中要播放一個多媒體影音資料時，會尋找一些模組來完成特定的程序，例如:負責 MPEG-2 PS 格式解多工的 ps 模組和解碼的 libmpeg2 模組，然而這些模組都會利用巨集的方式表示其模組的特性，也由於每個模組都有各自的特性，所以 module_Need()函數才能根據每個模組所定義的特性來找到最適合的模組。而且每個模組都會將自己封裝成 DLL 格式;讓 VLC 可以用外掛的方式來呼叫，其詳細的 DLL 原理會在第五章節中詳述。

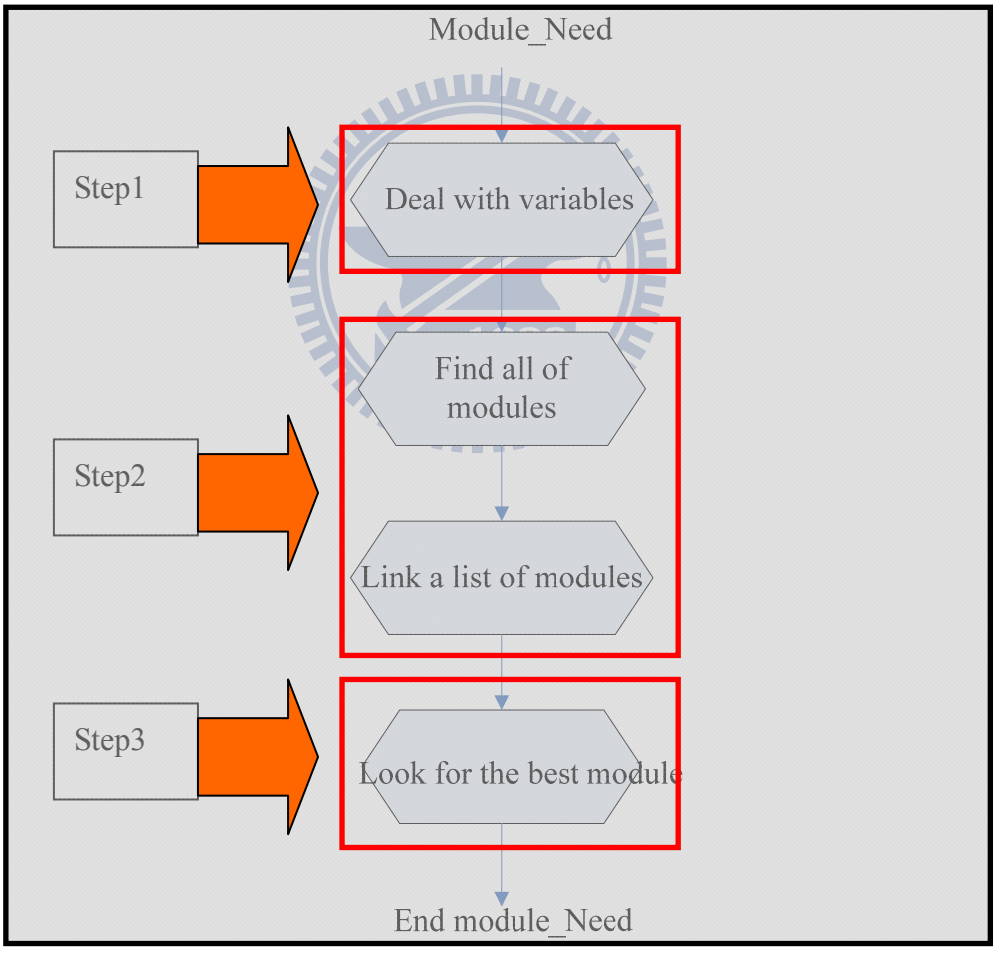


圖 3.16 module_Need() 簡易流程

我們介紹完 3.2.2 章後，我們可以知道 VLC 在需要執行某些處理多媒體程序時，會利用 module_Need()函數來尋找適合的模組來完成某些程序。譬如開啟檔案的程序需要資源存取模組，要將影音資料分離成 audio bitstream 和 video bitstream 的程序需要解多工器模組等等，接下來我們就會去分析 VLC 尋找模組的函式 module_Need()函數。

上圖 3.16 表示整個 module_Need()函數的流程，我們將整個流程分三大步驟來討論，當然整個過程和其運作機制並沒那麼簡單，每個步驟裡面都有很多的條件判斷和變數設定。其中步驟 1 的 Deal with variables 程序只是會因為需要尋找何種功能的模組不同而會不同。下面只會做個簡單的說明，我們將會把重點放在其他三個步驟來加以說明整個 module_Need()函數的重點所在。

Step1:

module_Need()函數是一個尋找模組的重要函數，我們先來看看傳入的參數所代表的意義。參數 p_this 代表尋找模組時所需的資訊,如模組在記憶體中的位置。參數 const char *psz_name 會依所要尋找模組的功能不同而不同，參數 psz_capability 則是尋找的模組功能的重要參數，譬如若是要找一個解多工器模組，所傳入的字串就是〝demux2〞;若是要尋找解碼器模組，則會傳入字串為〝decoder〞，參數 vlc_bool_t b_strict 則是一個旗標變數。

Step2:

在第二步驟中我們可以了解到 module_Need()函數是如何可以找到所有的模組，並又是如何選出符合能力的模組。其中宣告的 module_list_t 結構成員中的*p_module 指標變數就是用來存放符合傳入參數 psz_capability 能力的模組其中包含了模組的所有特性，其成員 i_score 則是紀錄能力值的變數，然後 module_list_t *p_next 則是用來對符合能力的模組做 linked list 而宣告的，如圖 3.17 所示，module_list_t 的結構就像是一個模組的標籤，所以藉著模組的標籤 module_Need()函數能將符合同能力條件的模組(符合 psz_capability 字串)做 linked list 的動作。

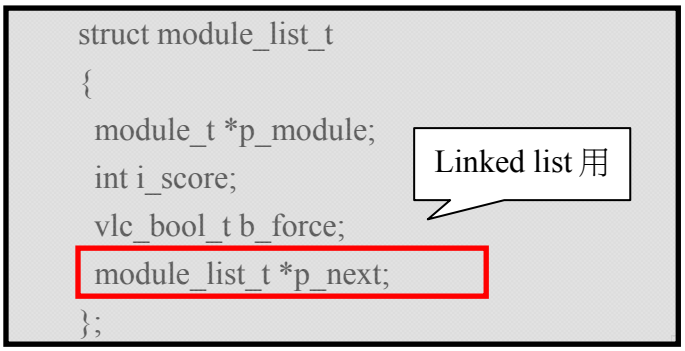


圖 3.17 module_list_t 結構成員

那在 module_Need()函數又是如何知道 VLC 中所有的模組在記憶體中的位址呢?為了可以知道所有模組在記憶體中的位址和配合 VLC 中可以任意彈性地增加和減少模組數量的開放平台特點,所以如圖 3.18 所示,在函數中就會定義一個結構型態為 vlc_list_t 的指標變數 *p_all,然後其中的成員 p_all->p_values[i].p_object 是結構陣列會用來存放由函數 vlc_list_find()所找到所有 VLC 中所有模組在記憶體中的位址,其中的成員 p_all->i_count 就是來儲存模組的總數。下面我們就來討論 vlc_list_find()函數的功用。

```
p_all = vlc_list_find( p_this, VLC_OBJECT_MODULE, FIND_ANYWHERE );
p_list = malloc( p_all->i_count * sizeof( module_list_t ) );
p_first = NULL;
```

圖 3.18 儲存所有模組

在函數 module_Need()中的 vlc_list_find()函數主要是在 VLC 所有的物件中找出是模組的物件並用結構陣列的形式儲存其模組的特性。首先會用雙重指標變數**pp_current 和**pp_end 來指向 VLC 中所的物件於記憶體中的最前端和最末端的位址,如圖 3.19 所示,並利用一個 for()迴圈及判斷條件來算出 VLC 中有多少物件是屬於模組型態,在函數 module_Need()中 i_type 的值 VLC _ OBJECT _MODULE,並用變數 i_count 來計算總共有多少個模組。

```
for( ; pp_current < pp_end ; pp_current++ )
 {
 if( (*pp_current)->b_attached
   && (*pp_current)->i_object_type == i_type )
     {
         i_count+
```

圖 3.19 判斷模組條件

計算完模組總數之後，就會將 i_count 變數傳入 NewList()函數中做記憶體分配的工作，如圖 3.20 所示，然後再由一個 for()迴圈內執行 ListReplace()函數來儲存我們的模組特性。要注意的是我們必須先經 NewList()函數做記憶體分配後， p_list->p_values 才能以結構陣列的方式表達。

```
for( ; pp_current < pp_end ; pp_current++ )
  {     if( (*pp_current)->b_attached
              && (*pp_current)->i_object_type == i_type )
        {
        ListReplace( p_list, *pp_current, i_index );
        if( i_index < i_count ) i_index++;
                    以結構陣列形式儲存
  }
p_list->p_values[i_index].p_object = p_object;
```

圖 3.20 用陣列形式儲存所有模組

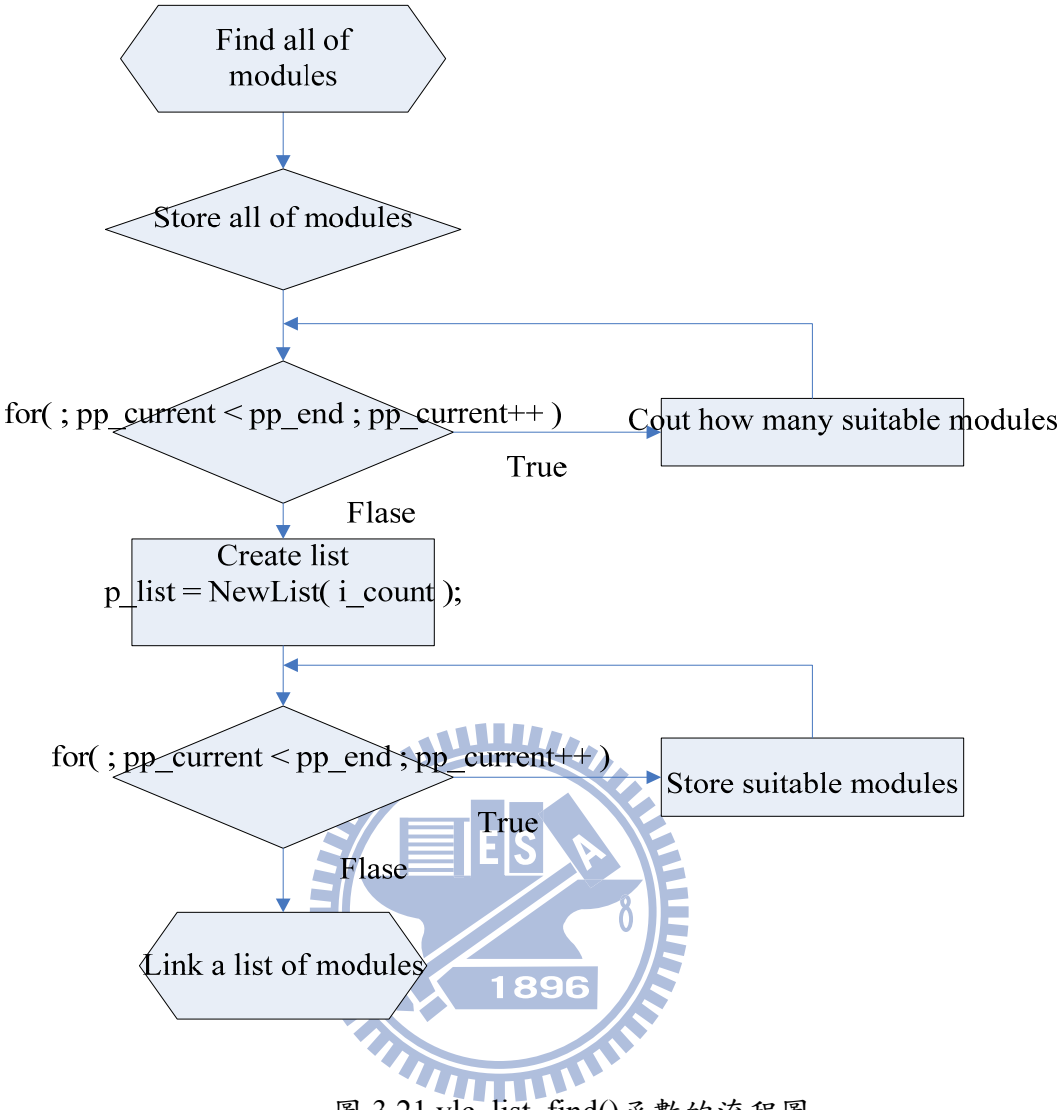


圖 3.21 vlc_list_find()函數的流程圖

由於在 VLC 中有許多的內建物件，其中當然也包括了 200 多個模組，所以 VLC 就會利用 vlc_list_find()函數來進行所有模組的額外的儲存，主要的原因是防止別的程式不小心去修改到記憶體中模組的特性。然後 module_Need()函數會將全部的模組一個個拿來做能力的比對，在下面我們就會說明如何將這些模組來做能力的分類。在這裡 module_Need()函數裡會找出全部的模組並另外儲存。首先每個模組的特性在編譯時會利用的巨集函數來做一個模組基本的設定。

在儲存好所有的模組之後，便會開始利用一個 for()迴圈來尋找我們所需的模組。在迴圈內會先利用指標變數 p_module 指向儲存模組的 p_all->p_values[].p_object，然後做能力字串的比較，若能力字串不一樣就會執行 continue 指令回到迴圈的起點並判斷下一

個模組，藉此來找出我們符合能力的模組。

```
p_module = (module_t *)p_all->p_values[i_which_module].p_object; /* Test that this
module can do what we need
                                                    若字串一樣回傳 0
if( strcmp( p_module->psz_capability, psz_capability ) )          {

        continue;
    }
```

圖 3.22 module_Need()函式中比較能力片斷程式

如圖 3.22 所示，做完模組能力字串的比較後，我們就會將候選模組(candidate modules)另外使用變數 p_list 以結構陣列的方式儲存起來，如圖 3.23 所示。並準備將候選模組依能力值的高低用氣泡排序的方式來排序並做 linked list 的動作。在這裡 module_Need()函數是用儲存一個候選模組後就立即進行排序的方式。

```
p_list[ i_index ].p_module = p_module;
p_list[ i_index ].i_score = p_module->i_score + i_shortcut_bonus;
p_list[ i_index ].b_force = i_shortcut_bonus && b_strict;
```

圖 3.23 儲存要測試的模組

在比對完模組能力後，然後 module_Need()函數就會開始會將符合能力的候選模組連結在一起。如圖 3.24 所示，首先會先利用變數 i_index 判斷是否為第一個候選模組，若 i_index==0 代表是第一個候選模組，然後就用*p_first 指標變數指向第一個候選模組。

```
if( i_index == 0 )
{
    p_list[ 0 ].p_next = NULL;
    p_first = p_list;
}
```

圖 3.24 一次比較程式

如圖 3.25 所示，再次進入迴圈且又為符合能力的模組情形下，就會用*p_newlist 指標變數存放*p_first 指標變數指向的位址，並判斷此時的候選的模組能力值是否大於第一個候選(目前能力值最高)模組，若大於最高能力值的候選模組，就會將其被判斷的候選模組(假設為 module X)的*p_next 指向*p_first 指標所指向的位址，並將*p_first 指標指向其候選模組，如圖 3.26 所示。

```
module_list_t *p_newlist = p_first;
if( p_first->i_score < p_list[ i_index ].i_score )
{
    p_list[ i_index ].p_next = p_first;//將原先的排到下一個
    p_first = &p_list[ i_index ];//指向分數高的
}
```

圖 3.25 被判斷的候選模組大於分數最高的候選模組程式

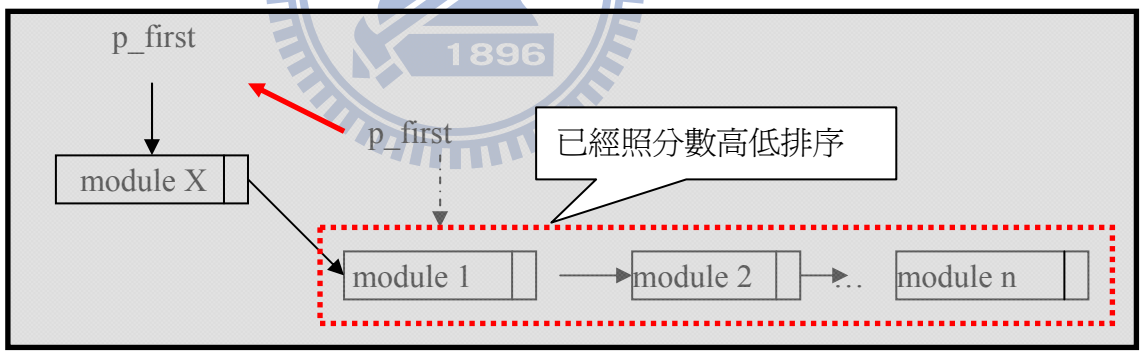


圖 3.26 被判斷的候選模組大於分數最高的候選模組時情形

如圖 3.27 所示，若其被判斷的候選模組(module X)不大於能力值最高的候選模組時，就會利用*p_newlist 指標來指向次高能力值的候選模組，直到在 linked list 中的候選模組找到*p_newlist 指標來指向的下一個候選模組比被判斷的候選模組還低的能力值為止。並將其被判斷候選模組的*p_next 指標指向 p_newlist->p_next 所指向的候選模組，然後 p_newlist->p_next 再指向被判斷的候選模組的位址。最後被判斷的候選模組就會插入連結好的候選模組中，如圖 3.28 所示。

```
while( p_newlist->p_next != NULL &&
p_newlist->p_next->i_score >= p_list[ i_index ].i_score )
{
 p_newlist = p_newlist->p_next;//指向下一個
 }
 p_list[ i_index ].p_next = p_newlist->p_next;
 p_newlist->p_next = &p_list[ i_index ];
```

圖 3.27 被判斷的候選模組不大於分數最高的候選模組程式

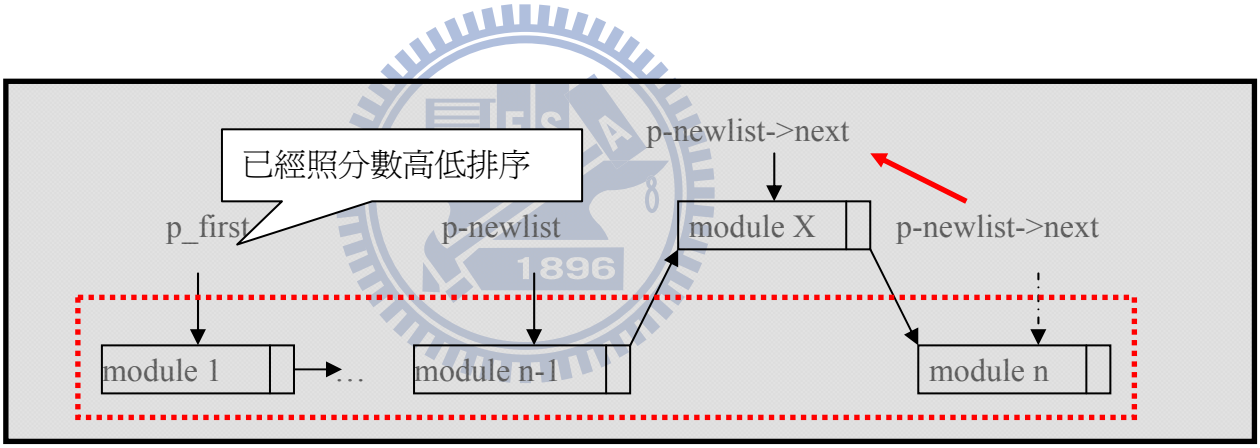


圖 3.28 被判斷的候選模組不大於分數最高的候選模組時情形

我們可以很清楚地看到*p_newlist 指標扮演著一個偵查和分配的角色，負責在上述各種不同的情況下，將被判斷的候選模組分配到已經連結好的候選模組中。程式中的 if(i_shortcuts > 0)來判斷先在這個候選模組有沒有被設定符合捷近的條件，若有符合捷近的條件的話就會讓這個候選模組的能力值大大的增加，最後必定是此候選模組排在第一順位，不過此設定只有在*psz_name=$memcpy 的條件下或選擇特定串流模組下才會執行。如圖 3.29 是 Step2 的流程圖。

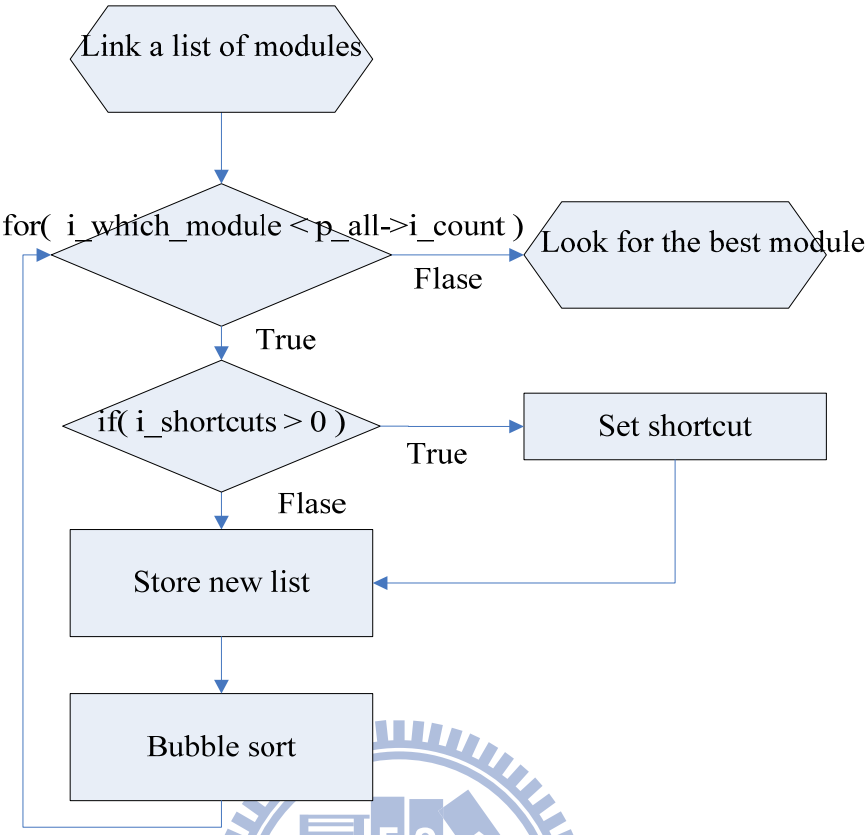


圖 3.29 module_Need()函數 Step2 流程圖

Step3:

在 Step2 結束後，符合能力的候選模組就會依照分數的高低作好 linked list。然後就會利用*p_tmp 指標指向第一個候選模組並鎖定 linked list 內的所有候選模組，因為我們已經儲存所有符合能力條件的模組，所以會將存放所有模組的 p_all 給釋放掉。然後進入一個 while()迴圈〝p_tmp->p_module->pf_activate〞會指向一個候選模組並執行由一個模組在經由編譯器編譯時由巨集所設定好的開啟測試函數的 callback function 進行候選模組開啟條件的測試，如圖 3.30 所示，此時的測試函數會因不同能力的模組而有不同的測試方式，譬如解多工器模組就是利用看資料 header 的方式來進行測試，而解碼器模組則是利用四字元比較的方式進行測試。若開啟條件不符合的情形下就會指向下一個候選模組，然後執行下一個模組的測試函數，並釋放不適合的候選模組，若成功開啟就會設定此模組所需的 callback functions 並跳出尋找模組的迴圈且不在尋找其他的模組。在

成功開啟模組後，會先儲存其候選模組的資訊，並將其他的候選模組都釋放掉。最後會回傳其候選模組的位址，圖 3.31 為 step3 的流程。

```
if( p_tmp->p_module->pf_activate
    && p_tmp->p_module->pf_activate( p_this ) == VLC_SUCCESS )
{
    break;
}
```

圖 3.30 判斷是否為適合模組條件程式

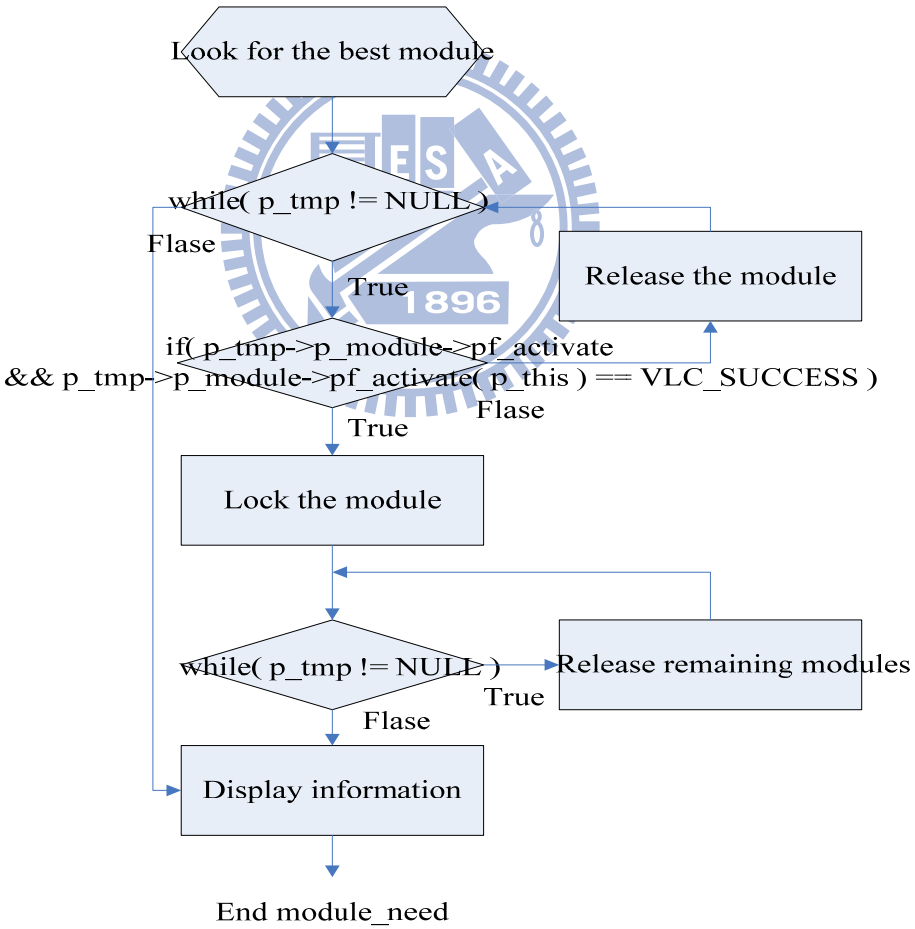


圖 3.31 module_Need()函數 Step3 流程圖

整個 module_Need()函數尋找模組的機制就是由上述的三大步驟所構成，雖然在尋找不同能力的模組時 module_Need()函數會有些許的不同變數處理程序，但是大致上尋找的程序是一樣的，都是會透過 linked list 的方式將適合的模組串在一起。但為什麼一定需要使用 linked list 呢? 因為 linked list 可以動態的連結我們所需的模組，不像宣告陣列的方式必須先將陣列的大小事先設定好，而且在宣告陣列所配置記憶體的大小後就不能更改，若配置太大則會浪費寶貴的記憶體空間，配置太小又會造成記憶體不足的問題，造成程式執行的錯誤。相對之下 linked list 的方式就彈性了許多，除非是在系統上有記憶體不足的原因，無法滿足動態記憶體的分配需要時，linked list 才會塞滿。另外最重要的就是因為我們沒法無法事先得知在比較各種不同的條件後各有多少適合的模組。而且不論在程式設計者額外增加或減少模組都不用特別去修改程式本身，在程式維護的方面來看 linked list 的方式確實比宣告陣列的方式來的方便多了。

3.2.4 MainLoop()函數

在 Init()函數執行成功後，就表示結構 input_thread_t 內成員的設定無誤，此時就會利用 MainLoop()函數來真正執行 Init()函數中所找到的模組。在 MainLoop()函數中主要是用來呼叫解多工器模組執行解多工的程序以及對使用者使用影片的暫停、停止、重複播放等功能作適當的處理。MainLoop()函數可以說是對多媒體影音檔案真正開始做處理程序的地方，我們在討論 MainLoop()函數前，我們會先針對他傳入的參數*p_input 做一個概述，前面我們已經知道型態 input_thread_t 的指標變數*p_input 內存放所有在 input thread 下所需的變數和結構設定，其中也包含了經過 Init()函數呼叫 module_Need()函數所找到適合的資源存取模組以及解多工器模組的所有資訊，所以 MainLoop()函數就會利用這些資訊去進行解多工程序。

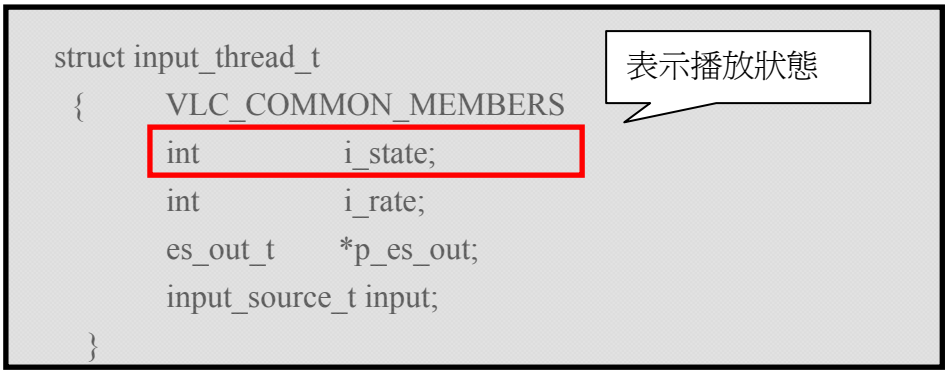


圖 3.32 播放狀態變數

在 MainLoop()函數開始後會先進入一個 while()迴圈就會先檢查三個旗標變數來判斷解多工的必要條件變數，如圖 3.33 所示。進入迴圈後就會用 i_ret 變數來代表解多工器模組的目前狀態， i_ret=0 代表資料已經結束了， i_ret>0 代表還有資料， i_ret<0 則表示解多工器模組有錯誤。在 MainLoop()函數第一次呼叫解多工器模組時，VLC 同時也會藉著解多工器模組內的層層呼叫開始尋找影像和聲音各自的解碼器模組以及產生各自解碼的 thread。MainLoop()函數會不斷的更新和檢查資源存取模組以及解多工器模組的狀態和變數。

旗標變數	旗標意義
p_input->b_die	判斷 input thread 是否有出錯
p_input->b_error	判斷 Init()函數是否初始化成功
p_input->input.b_eof	判斷是否有設定重複播放

圖 3.33 旗標變數

在函數不斷解多工的同時也會用 p_input->i_state 來判斷目前解多工的狀態;若 p_input->i_state 不為 PAUSE_S 時，代表使用者沒有使用暫停功能，此時的 i_ret 會進入條件判斷。藉此來判定是否還有資料要執行解多工程序，若有還有資料要就會開始對解多工器模組和資源存取模組做變數和狀態的更新並開始執行解多工程序。若資料達到了尾端，此時會檢查變數 repeat.i_int。如圖 3.34 所示，變數 repeat.i_int 是儲存 VLC 的偏

好設定中所輸入重複播放的次數。若 repeat.i_int==0 表示不會再次播放已結束的檔案，那麼變數 p_input->input.b_eof 會被設定成 VLC_TRUE 之後就會跳出解多工的迴圈，結束解多工程序。

如圖 3.35 所示，因為 MainLoop()函數在執行解多工程序的同時，也判斷使用者使否有下達指令所以還會執行一個 while()迴圈專門在處理使用者下達的暫停和停止等等指令。VLC 會用變數 control type 來記錄使用者下達下達何種指令;如圖 3.36、3.37 所示，我們整理出 control type 的意義和整個 MainLoop()函數的流程圖。

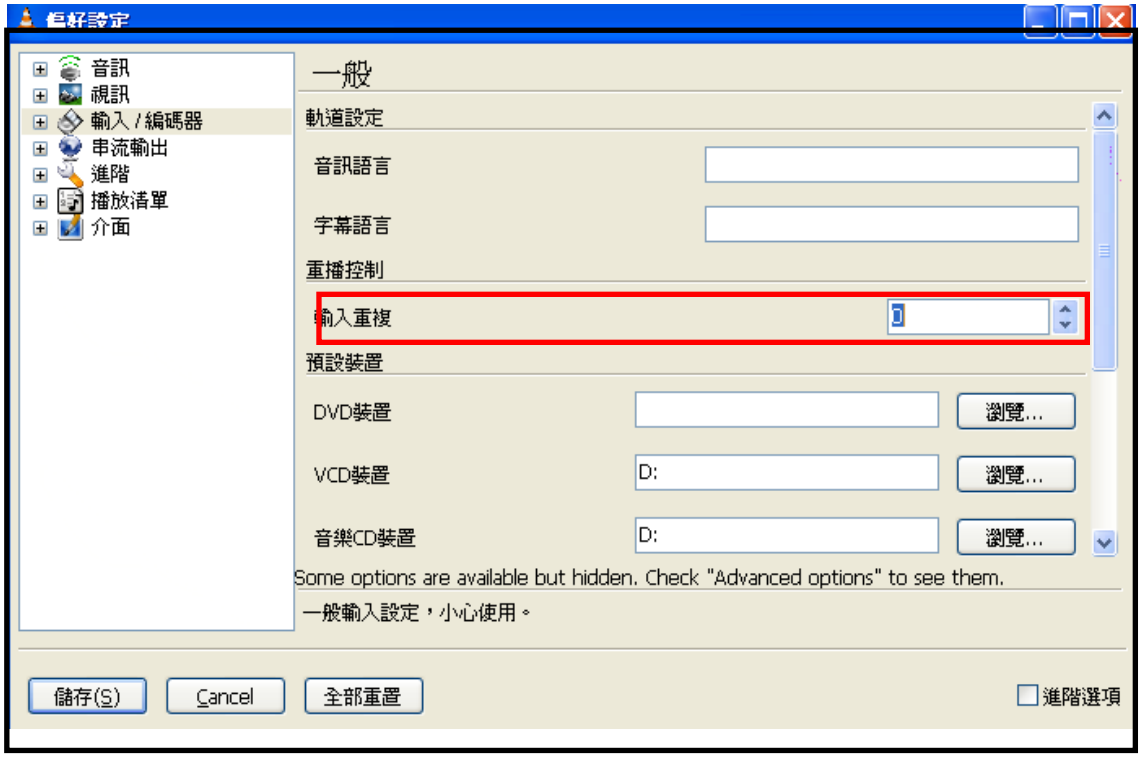


圖 3.34 VLC 中設定重複播放

```
while( !ControlPopNoLock( p_input, &i_type, &val ) )
{
  msg_Dbg( p_input, "control type=%d", i_type );
    if( Control( p_input, i_type, val ) )
    b_force_update = VLC_TRUE;
}
```

圖 3.35 播放控制處理

變數 control type	變數意義
0	結束
1	暫停,再次播放
3	慢速播放
4	快速播放
5	使用影片搜尋

圖 3.36 control type 的意義圖示

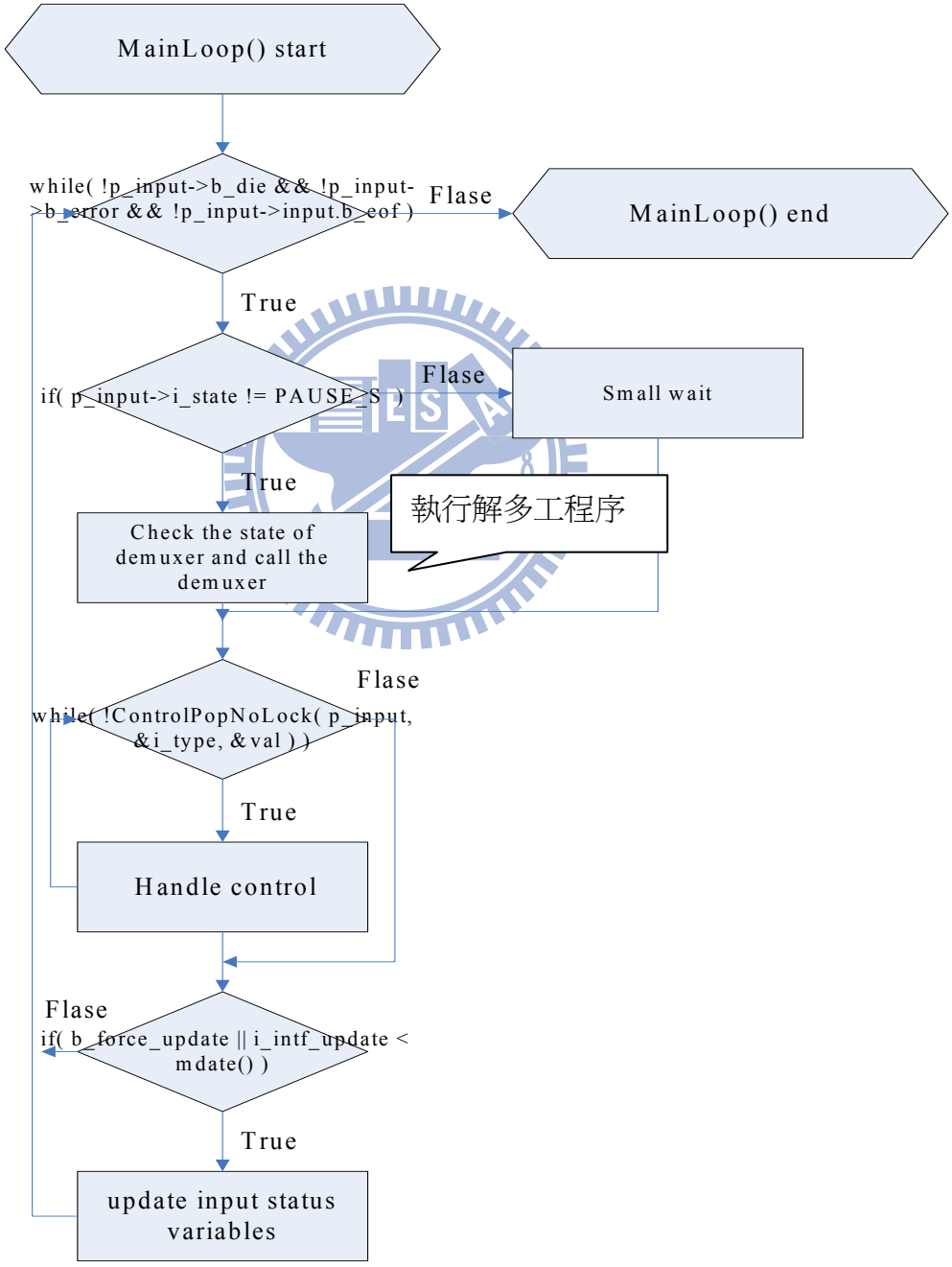


圖 3.37 MainLoop()函數流程圖

第四章 VLC 解多工器與解碼器模組

第三章介紹完 VLC 播放影音的流程和一些重要的函數後，我們會發現 VLC 要執行一些程序時就會呼叫 module_Need()函數來尋找程序所需要的模組，然後利用 module_Need()函數的機制來找到最佳的模組。在本章會進一步的介紹當 module_Need()函數找出適合的解多工器模組後如何藉由從解多工器模組所得到的影音資訊去尋找適合的解碼器模組的過程和尋找最佳解碼器模組的四字元比較機制。

本章也會討論解多工器與解碼器模組之間是如何溝通以及解多工器模組是如何將解多工好的資料放入一個 FIFO buffer 中，而解碼器模組又是如何去讀取 buffer 中的資料? 並會進一步討論在不同的 thread 下的解多工器與解碼器模組之間讀取資料的機制，本章會利用 program stream mpeg-2 的例子做一個討論。

4.1 解多工器模組

在第三章我們有提到 Init()函數會設定 VLC 中的大資料庫結構"input_thread_t"; 結構 input_thread_t 中包含了資源存取和解多工器模組等等資訊，Init()函數會呼叫 module_Need()函數尋找適合的解多工模組，最後 Init()函數會將設定完成的結構 input_thread_t 傳給 MainLoop()函數，所以 MainLoop()函數就會呼叫由 Init()函數中所找到的解多工器模組來進行解多工程序。

Module_Need()函數在尋找適合的解多工器模組時會先比對每個模組中的能力，然後就可以在全部的模組中找出具有解多工能力的模組並會依據模組的能力值高低來做 linked list 的動作。因為每個解多工器模組都會定義一個開啟測試函數;所以 Module_Need()函數會依序呼叫 linked list 中解多工器模組的開啟測試函數;若當 linked list 其中一個模組開啟測試函數成功的比對條件後就會選定此模組為適合的解多工器模組，並回傳成功的訊息給 Module_Need()函數，開啟測試函數就會設定此模組中所定義的解多工函數來進行解多工的程序。在 Module_Need()函數找到適合的解多工器模組

後，圖 4.1 中解多工器模組結構 demux_t 中的解多工模組資訊和解多工函數都會被設定好，然後在 MainLoop()函數中就會呼叫指向適合解多工器模組中所定義的解多工器函數的 callback function 〝 p_demux->pf_demux〞來進行解多工函數，下面我們會用一個 mpeg-2 programstream 的影音檔案來說明 VLC 是如何找到適合的解多工器模組，然後也會介紹如何用解多工器模組所提供的四字元找到適合的解碼器模組。

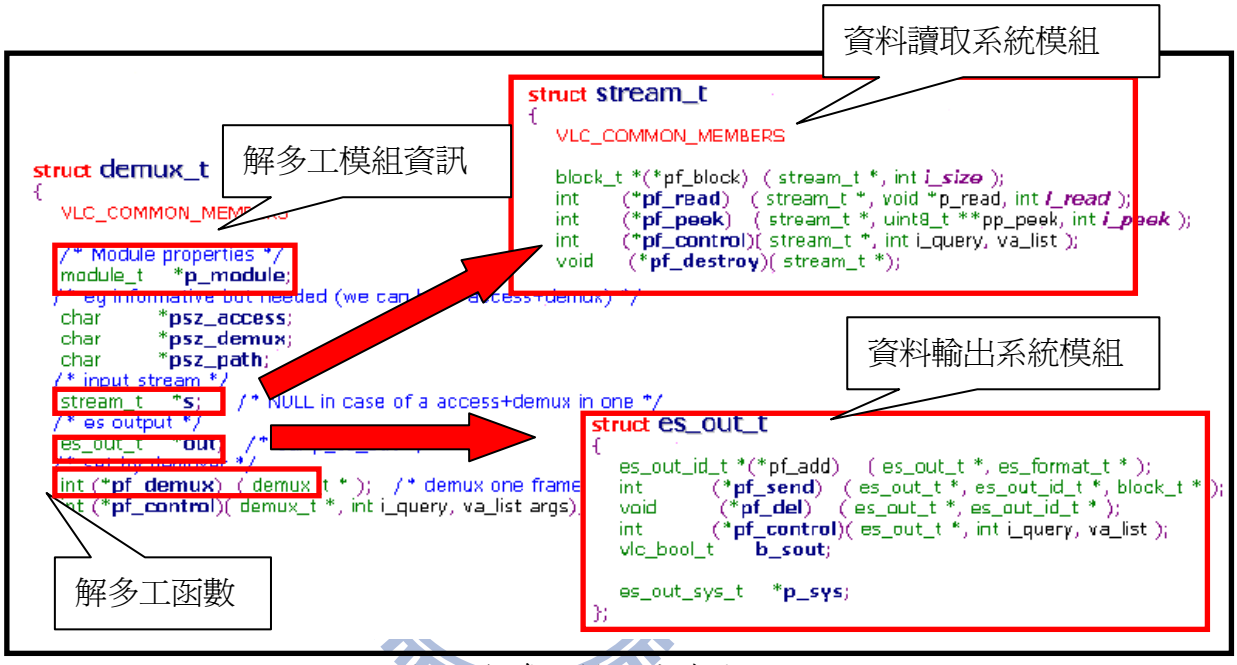


圖 4.1 解多工模組結構圖

4.1.1 PS 解多工器模組

Mpeg-2 programstream 是由許多 pack 所組成，每個 pack 是由一個 pack header 再加上多個 PES(Packetized Elementary Stream)所組成的，許多 PES 組成一個 PS 前必須要加上 Pack header 和 System header，每一個 header 都是由一序列固定的起始碼，其中可區分出資料的每層架構，而 PS 大致上可分成三層。

如 4.2 圖所示每一個 PES 所代表的就是一個 audio PES 或者 video PES 或是其他類型 PES。所以解多工器模組就會一層層的的解析出 audio PES 和 video PES，並再從 audio PES 和 video PES 當中取出 PES payload，而這些 PES paylaod 就是第二章我們所提到的

video bistream 或 audio bistream，最後這些 bistream 就會由解碼器模組來進行解碼動作。

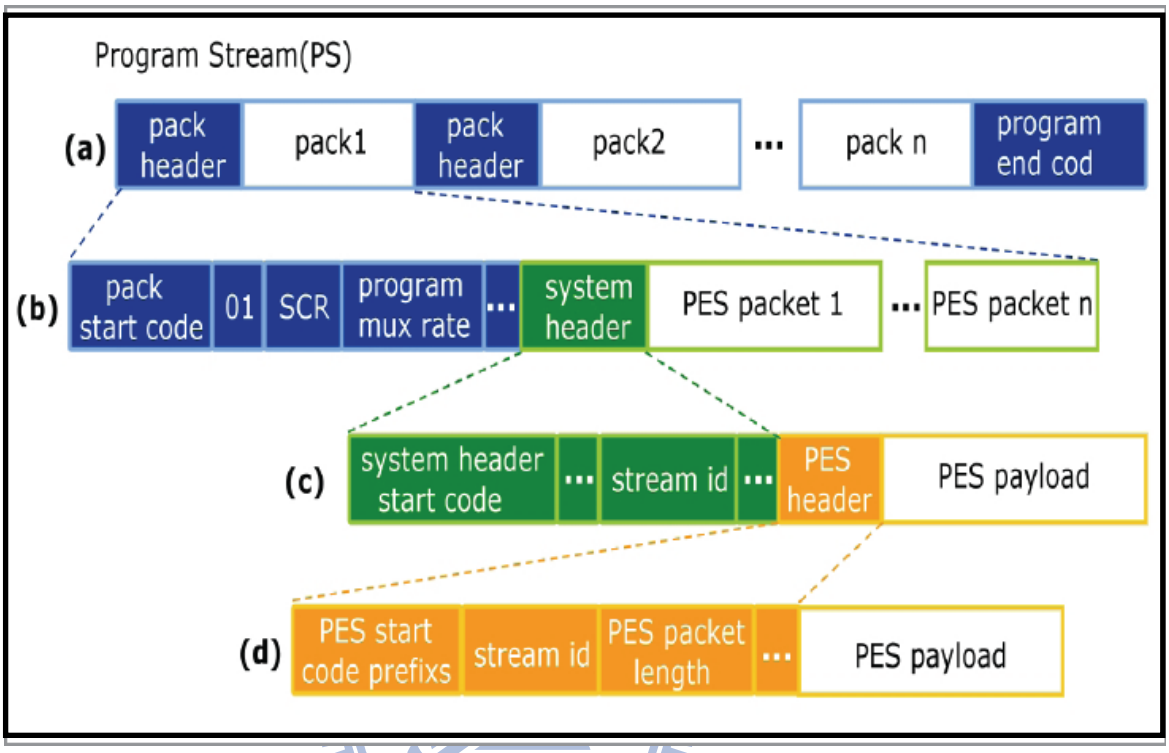


圖 4.2 mpeg-2 programstream 結構圖

在 module_Need()函數要測試 ps.c 解多工器模組是否為適合的模組時，會先檢查讀到的 header 是否為 mpeg-2 programstream pack header 的標準格式，每個 pack 的最外層的 pack header 是由 4 bytes 的 pack_start_code:oxooooo1BA 開始，所以 module_Need()函數中 p_tmp->p_module->pf_activate 會指向 ps.c 模組中的開啟測試函數 Open()來測試 pack header 是否大於 4 bytes，若小於 4 bytes 代表不為 mpeg-2 programstream 的格式，就會回傳錯誤訊息給 module_Need()函數然後 p_tmp->p_module->pf_activate 就會指向下一個模組的開啟測試函數進行條件比對。

```
if( stream_Peek( p_demux->s, &p_peek, 4 ) < 4 )
{
 msg_Err( p_demux, "cannot peek" );
 return VLC_EGENERIC;
}
```

圖 4.3 判斷 pack header 是否大於 4 bytes

ps.c 模組的開啟測試函數中會呼叫解多工模組結構中讀取資料函數來讀取 MPEG-2 header 資訊，並判斷其 header 資訊是否有符合的必需條件如上圖 4.3 所示。因為 pack_start_code 在標準中就是定義為 4 個 bytes 的長度所以在 ps.c 模組的開啟測試函數就會先判斷 header 的長度是否為 4 個 bytes 的基本條件，接著就會判定讀取 pack header 的資訊是不是符合 mpeg-2 programstream 的標準格式，如圖 4.4 所示。

```
if( p_peek[0] != 0 || p_peek[1] != 0 ||
        p_peek[2] != 1 || p_peek[3] < 0xb9 )
  {
   msg_Warn( p_demux, "this does not look like an MPEG PS stream, "
                      "continuing anyway" );
  }
```

圖 4.4 判斷是否符合 mpeg-2 programstream 的標準格式

若符合其開啟的條件後，就會開始設定解多工模組結構 demux_t 中的解多工函數的 callback functions，也就是說 ps.c 模組中所定義的解多工函數 Demux()將被用來進行解多工程序，這些 callback functions 所指向的解多工函數才是真正解多工器模組對影音資料解多工的地方，然後就會回傳 VLC_SUCCESS 訊息告訴 module_Need()函式已經找到最佳的解多工器模組，在結束 Open()函數後整個解多工模組結構就會設定完成，如圖 4.5 所示。之後解多工函數會在 MainLoop()函數中不斷被呼叫。

```
p_demux->pf_demux = Demux;
p_demux->pf_control = Control;
```

圖 4.5 ps.c 模組設定 callback function

當 MainLoop()函數呼叫 p_demux->pf_demux 時會傳入整個解多工器模組結構 demux_t 後進行解多工程序，此時 ps.c 模組的所有資訊都存在解多工器模組結構中，所以呼叫 p_demux->pf_demux 時就等於在呼叫 ps.c 所定義的解多工函數 Demux()進行解多工程序。

在 mpeg-2 programstream 中有一個重要的特性，就是每個 PES 都具有共同的時間基準(time base) ，所以同一個 PS 中所有的 PES(Packetized Elementary stream)都享有共同的時間基準，當 MainLoop()函數呼叫 ps.c 模組中的 Demux()函數進行解多工程序時會先做同步檢查的動作以達到對影像和聲音同步的解碼，之後就會開始解析 PS 中的 header，藉由 header 的資訊可以判斷 video stream 和 audio stream 的編碼格式來取得四字元。

4.1.2 取得四字元

第三章中我們知道 VLC 是利用 module_Need()函數來尋找適合的模組，並將適合的候選模組用 linked list 的方式串聯起來，並依照各候選模組能力值的高低依序測試是否符合開啟條件。尋找不同能力的模組會有不同的開啟條件，在這小節我們會介紹解碼器模組的開啟條件〝四字元〞。

四字元是在解多工器模組判斷好影像和聲音的編碼格式後，就會設定四字元來表示影像和聲音的編碼格式。所以在 module_Need()函數尋找適合解碼器模組時，只要利用四字元的比較就可以很快的判斷出是否為適合的解碼器模組。

在 ps.c 解多工器模組中必須要有 header 中資訊才能得知 PS 中 video stream 和 audio stream 類型和 payload 編碼格式等等資訊，依據這些 header 資訊去判斷四字元。因此 ps.c

解多工器模組在進行解多工器模組時，會一層一層去解析 MPEG-2 PS 的 header;首先會先解析 pack header，然後在解析 System header;而 System header 中的 stream id 代表著 audio stream 和 video stream 的類型，其中 stream id 的值若在 oxCo~oxDF 間其 payload 為 audio stream，若在 oxEo~oxEF 間 payload video stream。

接下來會對 pack 中的 PES pack 進行解析，每個 PES pack 中的 PES payload 為 video stream 或 audio stream 資料。若當 PES pack 資料類型為 PSM(program stream map)時，那麼在解析此 PES pack 就能確認 video payload 和 audio payload 的編碼格式;因為 PSM 中紀錄了許多不同的 stream id 和編碼格式的對應關係;然而不同的 stream id 就會對應不同的編碼格式，假設 PSM 中 stream id=0xE0 所對應的編碼格式為 mpeg-2 video codec 的話，那麼只要是 PES header 中的 stream id 為 0xE0，就可以判定此 PES 中的 payload 是一個編碼為 mpeg-2 video 的 video stream。所以當 ps.c 模組在解析 MPEG-2 PS 時，就是藉由 PSM 中不同 stream id 所對應不同的編碼格式來判斷 video birstream 和 audio birstream 的編碼格式，然後 ps_track_fill()函數就能依據 stream id 和其對應的編碼格式來判斷四字元。

ps_track_fill()函數會依據 video bitstream 和 audio bitstream 的編碼格式和 stream id 來設定相對應的四字元。假設在 stream id=0xE0 和對應的編碼格式為 0x1b 時，就會判定此 payload 為 video stream 且編碼格式為 H.264，並設定四字元〝h264〞、資料型態 VIDEO_ES，如圖 4.6 所示。然後就會用結構 es_format_t 來儲存所判斷的結果，用其結構成員 i_cat 儲存是影像或聲音格式，成員 i_codec 則是用來儲存設定的四字元，如圖 4.7 所示。之後在判斷解碼器模組的開啟條件函數就是用型態結構 es_format_t 中的成員來取得四字元的資訊來進一步的來判斷是否為適合的解碼器模組。

```
if( (i_id&0xf0) == 0xe0 && i_type == 0x1b )
{
    es_format_Init( &tk->fmt, VIDEO_ES, VLC_FOURCC('h','2','6','4') );
}
else if( (i_id&0xf0) == 0xe0 && i_type == 0x10 )
{
    es_format_Init( &tk->fmt, VIDEO_ES, VLC_FOURCC('m','p','4','v') );
}
else if( (i_id&0xf0) == 0xe0 && i_type == 0x02 )
{
    es_format_Init( &tk->fmt, VIDEO_ES, VLC_FOURCC('m','p','g','v') );
}
else if( ( i_id&0xe0 ) == 0xc0 && i_type == 0x0f )
{
    es_format_Init( &tk->fmt, AUDIO_ES, VLC_FOURCC('m','p','4','a') );
}
else if( ( i_id&0xe0 ) == 0xc0 && i_type == 0x03 )
{
    es_format_Init( &tk->fmt, AUDIO_ES, VLC_FOURCC('m','p','g','a') );
}

if( tk->fmt.i_cat == UNKNOWN_ES && ( i_id&0xf0 ) == 0xe0 )
{
    es_format_Init( &tk->fmt, VIDEO_ES, VLC_FOURCC('m','p','g','v') );
}
else if( tk->fmt.i_cat == UNKNOWN_ES && ( i_id&0xe0 ) == 0xc0 )
{
    es_format_Init( &tk->fmt, AUDIO_ES, VLC_FOURCC('m','p','g','a') );
}
```

圖 4.6 ps_track_fill()函數部份程式

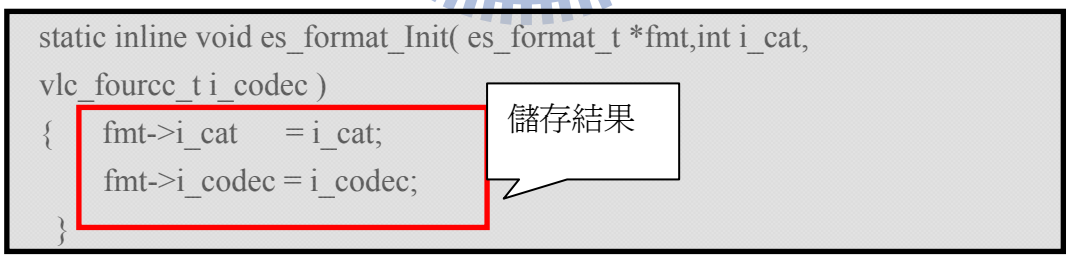


圖 4.7 es_format_Init()函數部份程式

在設定完四字元後所有 MPEG-2 header 中的影像和聲音的 stream id 以及編碼格式和四字元等等資訊都會存在型態 ps_track_t 結構中，如圖 4.8 所示。然後會呼叫圖 4.1 所示的資料輸出模組中的 pf_add 所指向的 EsOutAdd()函數並經過層層的呼叫來找到解碼器模組，在尋找適合的解碼器模組時 module_Need()函數就會由 ps _ track_t 結構中的四字元來對每個解碼器模組中所定義的四字元進行比對來找出適合的影像和聲音解碼

器模組。

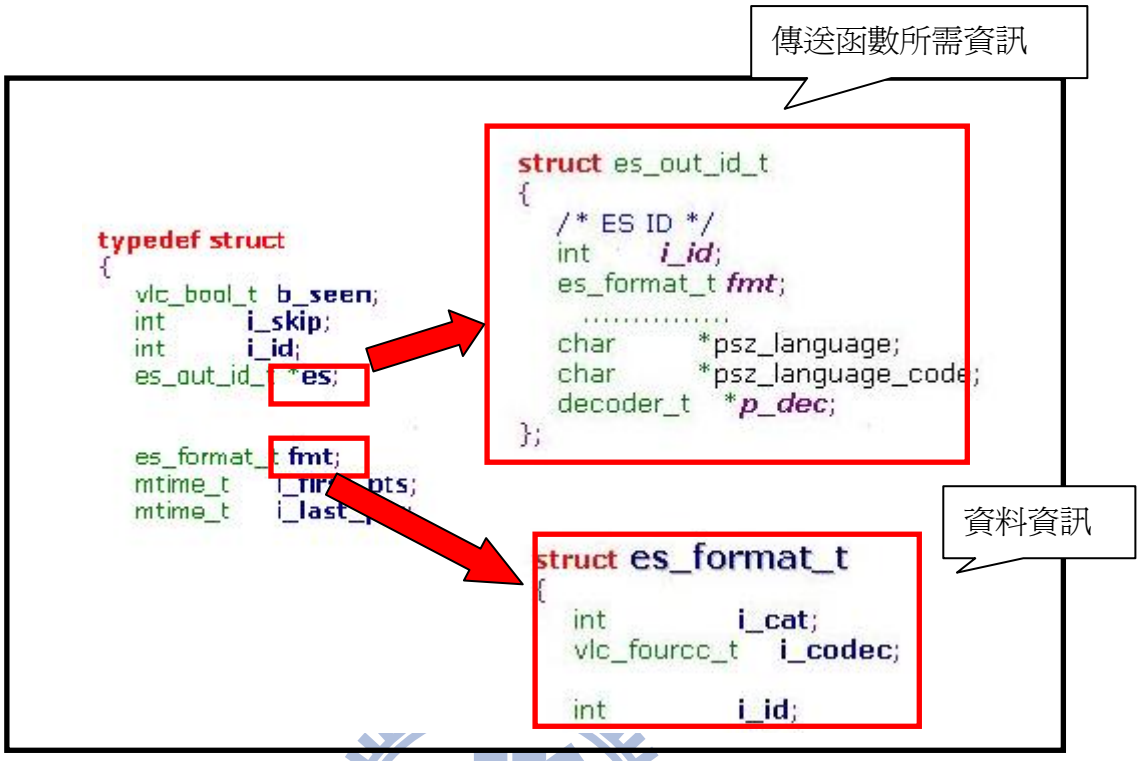


圖 4.8 結構 ps_track_t 示意圖

4.1.3 尋找解碼器模組流程

從 4.1.2 章節我們可以知道 ps.c 解多工模組中設定好四字元後，就會用四字元來尋找適合的解碼器模組，在尋找適合的解碼器模組時會呼叫 module_Need()函數來找到最合適的解碼器模組模組，在 module_Need()函數中會先經過能力的比較篩選出具有解碼能力的模組然後再依能力值的高低做好 linked list 的動作，再來就是會呼叫每個解碼器模組的開啟測試函數，每個解碼器模組的開啟測試中都會定義不同的四字元，所以只要判斷開啟測試函數中所設定的四字元是否符合由解多工器模組所設定的四字元就可以判定是否為適合的模組，譬如 libmpeg2.c 模組的開啟測試函數就會判定從 ps.c 模組中所設定的四字元是否為〝mpgv〞、〝mpg1〞、〝PTM1〞、〝VCR2〞、〝mp2v〞、〝mpg2〞、〝hdv2〞等等格式，如圖 4.8 所示。若有符合的四字元就會將解碼器模組結構中的解碼

callback function 指向 libmpeg2.c 解碼器模組中所定義的 DecodeBlock()解碼函數。若沒有符合的四字元就會判定 libmpeg2.c 解碼器模組不是適合的解碼器模組，並回傳 VLC_EGENERIC 的失敗訊息給 module_Need()函數，並檢查下一個解碼器模組中的開啟測試函數是否有符合的四字元。

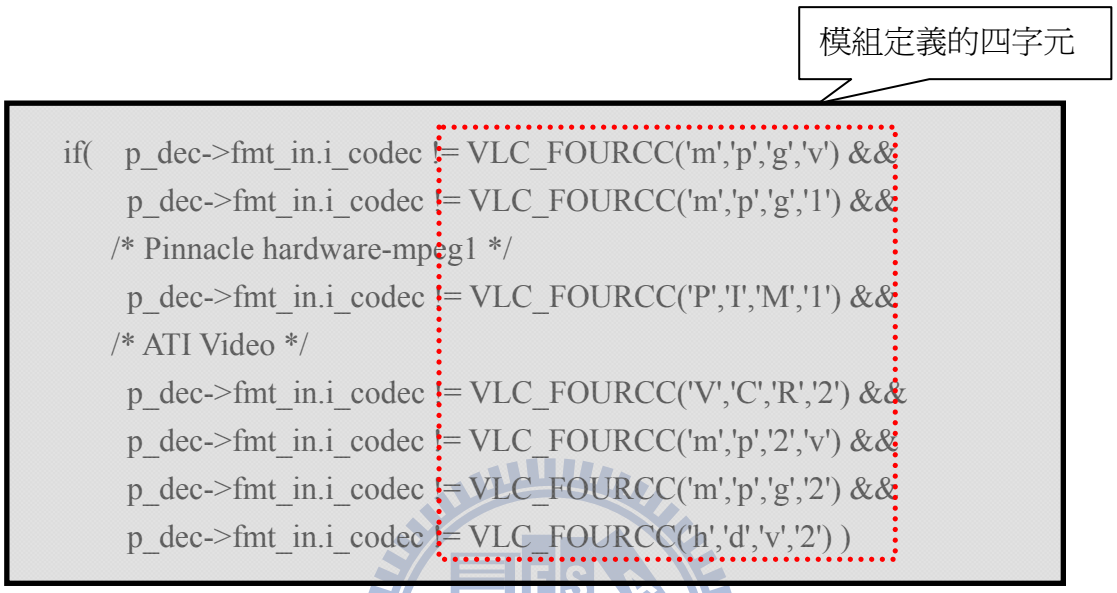


圖 4.9 libmpeg2.c 的四字元比較

在 module_Need()函數找到適合的解碼器模組後，就會將解碼器模組的結構 decoder_t 都設定好，如圖 4.10 所示。譬如以 libmpeg2.c 解碼器模組為適合的解碼器來說結構 decoder_t 中就會儲存 libmpeg2.c 解碼器模組的特性，以及 pf_decode_video 會指向 libmpeg2.c 解碼器模組中所定義的 DecodeBlock()解碼函數，所以在進行 MPEG-2 檔案的影像解碼程序時呼叫 pf_decode_video 就等於呼叫 libmpeg2.c 解碼器所定義的 DecodeBlock()解碼函數進行影像的解碼。

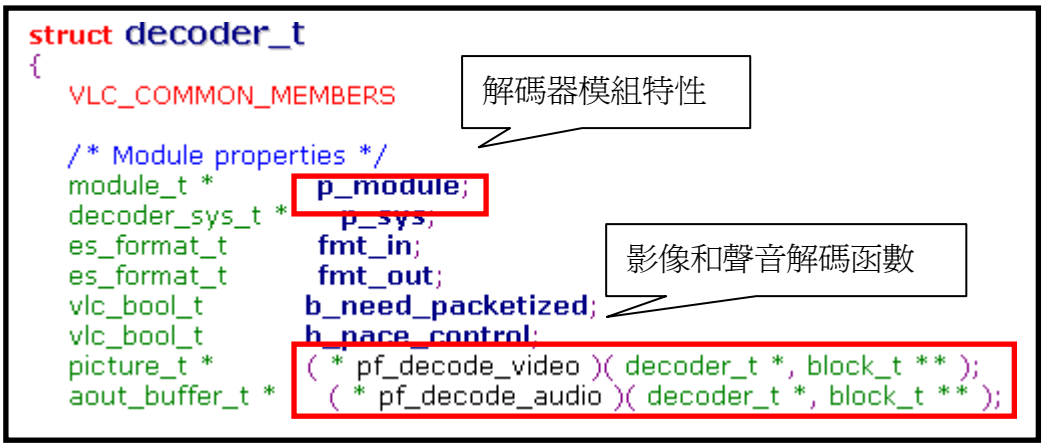


圖 4.10 decoder_t 結構內部份成員

所以我們可以了解到 VLC 要尋找一個適合的解碼器模組的方法就是藉由解多工器模組在解析影音資料時的過程中所判斷的 video stream 和 audio stream 編碼格式和資料型態去設定相對應的四字元，然後在 module_Need()函數中就會呼叫解碼器模組中開啟測試函數是否有定義相對應的四字元，若有符合的四字元那麼 module_Need()函數就會判定此解碼器模組為適合的解碼器模組。

4.2 解碼器模組解碼流程

在 module_Need()函數找到適合的影像和聲音解碼器模組後就會產生各自解碼的 thread 並開始進行解碼程序，如圖 4.11 所示此時解多工器模組就會不斷將影音檔案分離出 video biestream 和 audio bitstream 後會呼叫在 init()函數中所設定好的傳送資料的 es_out_Send()函數，並依據 ps_track_t 結構中的資訊將 video bitstream 和 audio bitstream 放入影像和聲音的 FIFO buffer 中。

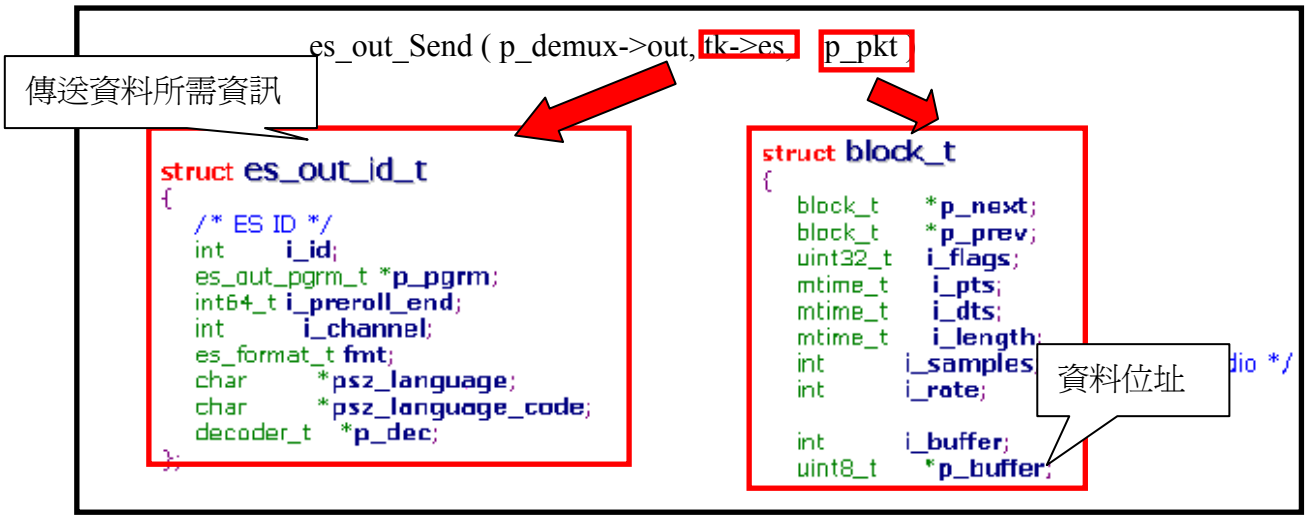


圖 4.11 輸出函數 es_out_Send()參數示意圖

圖 4.12 所示在 decoder thread 下 DecoderThread()函數是進行解碼程序的重要函數，呼叫時會傳入由 module_Need()函數所找到適合解碼器模組的所有資訊，並會用一個的結構 block_t 來讀取由解多工器模組放入 FIFO buffer 的資訊，然後接著就會進入一個解碼的主迴圈，我們可以很清楚地看到進入解碼主迴圈後就會呼叫 block_FifoGet()函數來讀取解多工好的 audio bitstream 或是 video bitstream 進行並進行解碼程序，然後會將讀取好資料位址傳給 DecoderDecode()函數中來開始進行解碼程序，若是資料讀取結束後就會跳出解碼主迴圈並結束解碼程序。

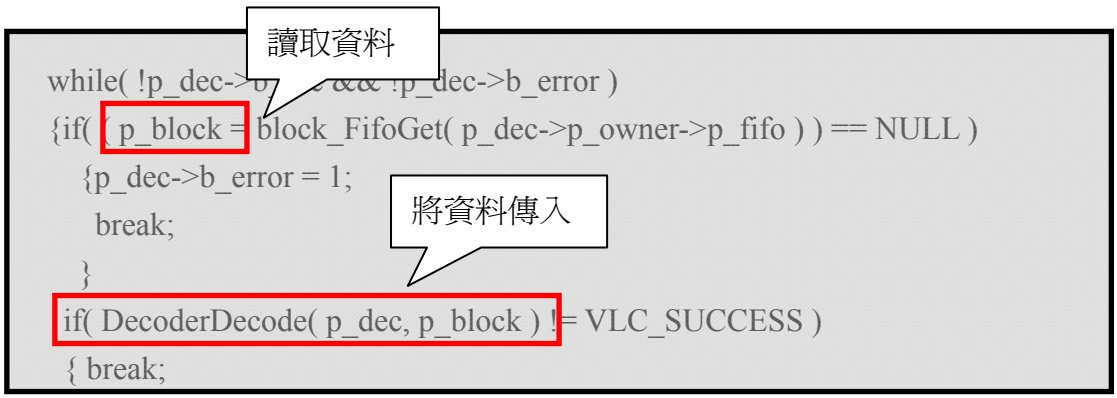


圖 4.12 Decoder thread 中的解碼主迴圈

其中在 DecoderDecode()函數中所呼叫的解碼函數就是先前 module_Need()函數比對四字元後找到適合解碼器模組中所定義的解碼函數，也就是說假設今天是開啟一個 MPEG2 的影音資料，那麼在呼叫 module_Need()函數時就會找到 libmpeg2.c 的影像解碼器模組來對 video stream 進行解碼的動作;也就是在判定 libmpeg2.c 模組為適合的影像解碼器模組時就必須將解碼器模組結構 decode_t 中的解碼 callback functions 指向 libmpeg2.c 解碼器模組中的影像解碼函數，因此在 DecoderDecode()函數中所呼叫的解碼函數就是呼叫在 libmpeg2.c 解碼器模組中的解碼函數來進行影像解碼程序，如圖 4.13 所示。

。

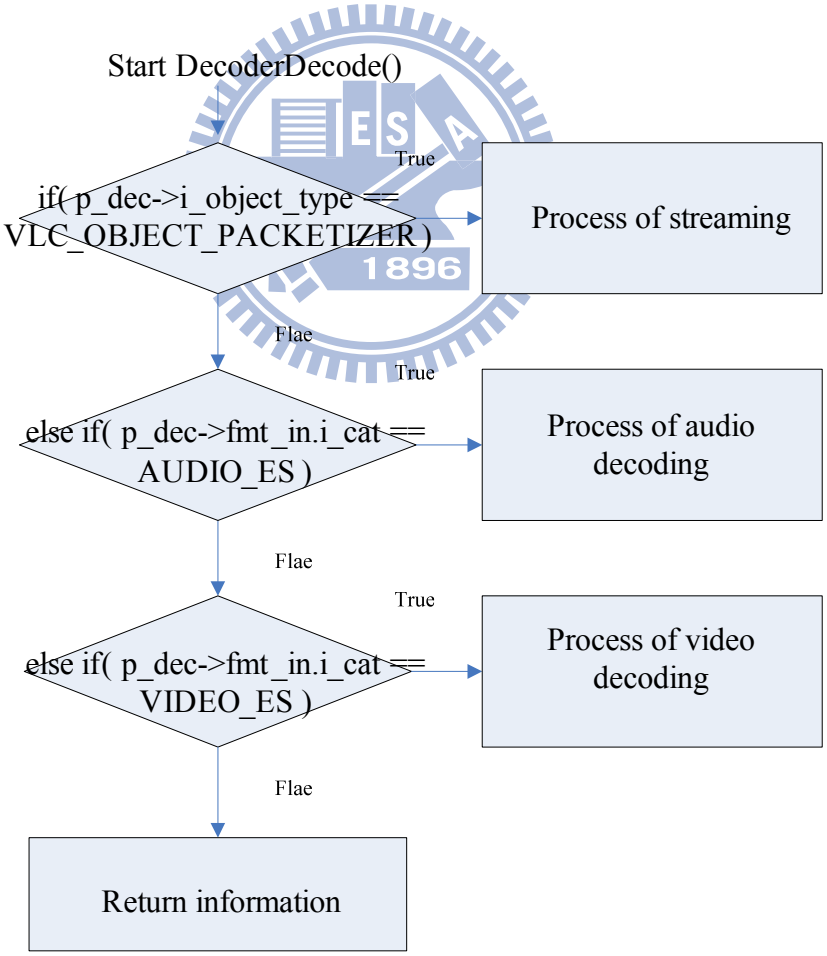


圖 4.13 DecoderDecode()函數的解碼架構

在 process of video decoding 程序中會有一個迴圈用不斷呼叫解碼的 callback function 函數，如圖 4.14 所示。其由 callback function 會指向由 module_Need()函數所找到的影像解碼器模組中所定義的影像解碼的函數進行解碼程序 。

```
else while( (p_pic = p_dec->pf_decode_video( p_dec, &p_block )) )
```

圖 4.14 影像解碼的 callback function

4.3 解多工器與解碼器模組之間的資料傳遞機制

在 4.2 節我們已經知道 VLC 找到適合的解碼器模組後就會開始產生一個 decoder thread 並開始進行解碼程序，此時在 input thread 下的解多工器模組會不斷開始對影音檔案進行解多工程序，被分離出來的 audio stream 和 video stream 會被放入不同的 FIFO buffer 中，然後影像和聲音解碼器就會到各自的 FIFO buffer 中去讀取資料，但是在 VLC 中的解多工器模組和解碼器模組是在不同的 thread 下運作，所以解多工器和解碼器模組之間的資料傳遞機制就變得很重要，在這一節會以開啟 MPEG-2 影音檔案為例子探討解碼器模組是如何存取解多工好的的資料封包來解碼，並討論其中的存取機制。

ps.c 解多工器模組在分析完封包的 header 後，就會找到 libmpeg2.c 解碼器模組並創造一個 decoder thread，ps.c 解多工器模組會呼叫函數 es_out_Send() 將解多工好的 audio stream 和 video stream 不斷傳送到不同的 FIFO buffer 中，而 es_out_Send()函數是在 Init()函數中所設定好傳送資料的函數;es_out_Send()函數則會判定是否有找到適合的解碼器模組，若已有適合的解碼器模組便會呼叫 block_FifoPut()函數來將解多工器模組所分離出的 audio stream 和 video stream 資料進行 linked list。在說明 block_FifoPut()函數之前，我們要先說明在尋找適合的解碼器模組時，如圖 4.15 所示會先呼叫 block_FifoNew()函

數來創造一個解碼器模組和解多工器模組之間傳遞資料的 FIFO buffer。下面會詳細說明 FIFO buffer 中的資料是如何進行 linked list。

```
if( ( p_dec->p_owner->p_fifo = block_FifoNew( p_dec ) ) == NULL )
{
 msg_Err( p_dec, "out of memory" );
 return NULL;
 }
```

圖 4.15 創造 FIFO buffer

在 block_FifoNew()函數中會利用 block_fifo_t 結構內的成員 i_depth 和 i_size 來記錄 linked list 資料的長度，而 block_fifo_t 結構內的成員型態 block_t 的指標變數 p_first 和雙重指標變數 pp_last 標示 linked list 資料的起始和結束點。如圖 4.16 所示在程式中我們可以看到 p_first 會先指向 NULL，然後 pp_last 指向 p_first 指標在記憶體位置，形成下面的圖 4.17 關係。

```
block_fifo_t *__block_FifoNew( vlc_object_t *p_obj )
{
 block_fifo_t *p_fifo;
 p_fifo = malloc( sizeof( block_fifo_t ) );
 vlc_mutex_init( p_obj, &p_fifo->lock );
 vlc_cond_init( p_obj, &p_fifo->wait );
 p_fifo->i_depth = p_fifo->i_size = 0;
 p_fifo->p_first = NULL;
 p_fifo->pp_last = &p_fifo->p_first;
return p_fifo;
 }
```

圖 4.16 資料的起頭和結束點

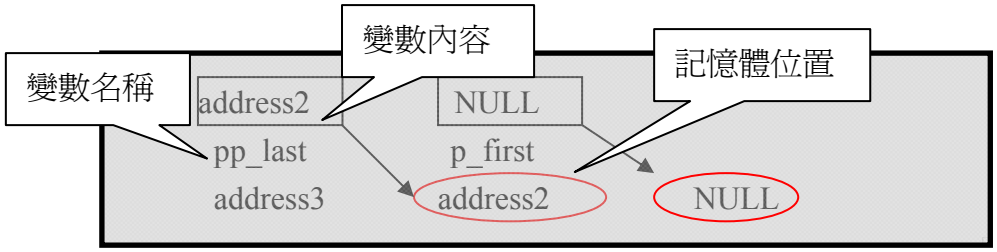


圖 4.17 記憶體位置關係示意圖一

在 block_FifoPut()函數中會利用一個 while()迴圈來放入由解多工器模組解析好的資料。在執行*p_fifo->pp_last = p_block 後由於 pp_last 是存放 p_first 的記憶體位址，所以*p_fifo->pp_last = p_block 會改變 p_first 所指向的位址，變成指向傳入的 p_block 所指向的位置;p_block 是指向由解多工器模組解析好的資料在記憶體中的位置。此時指標關係會變成圖 4.18。

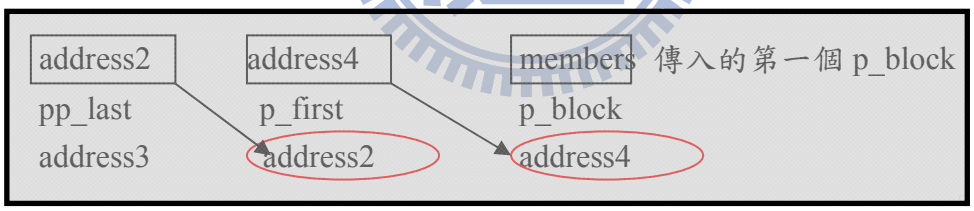


圖 4.18 記憶體位置關係示意圖二

```
do
{    i_size += p_block->i_buffer;
     *p_fifo->pp_last = p_block;
     p_fifo->pp_last = &p_block->p_next;
     p_fifo->i_depth++;
     p_fifo->i_size += p_block->i_buffer;
     p_block = p_block->p_next;//
  } while( p_block );
```

圖 4.19 FIFO buffer 中資料的 linked list 部份程式

如圖 4.19 所示當執行 p_fifo->pp_last = &p_block->p_next 時，p_first 就會固定的指向 linked list 的第一個資料。然後 pp_last 存放著 p_block->p_next 記憶體位址，所以在第二次進入迴圈後就 p_block->p_next 會指向第二個傳入的資料，在第一次迴圈結束後其指標關係如圖 4.20，其 linked list 關係如圖 4.21。

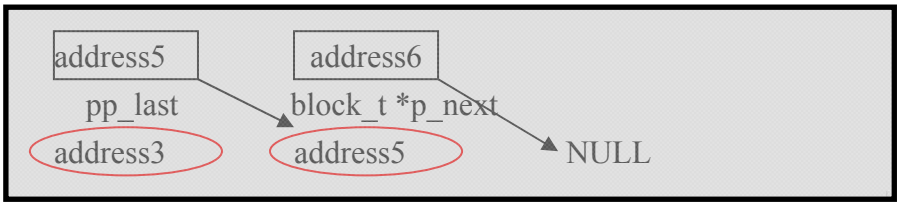


圖 4.20 記憶體位置關係示意圖三

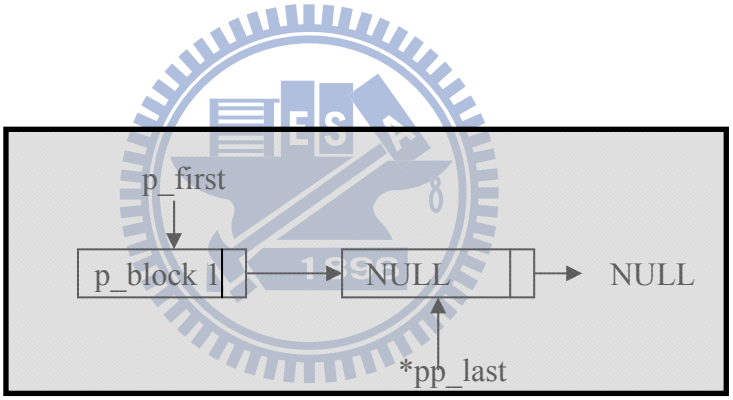


圖 4.21 第一次迴圈結束其 linked list 關係圖

在第二次進入迴圈會傳入第二個資料在記憶體中的位址，然後執行完 *p_fifo->pp_last = p_block 時會將第一輪的 p_block->p_next 指向新的 block_t 結構並開始 linked 第二個 block_t 結構，其指標關係如圖 4.22。

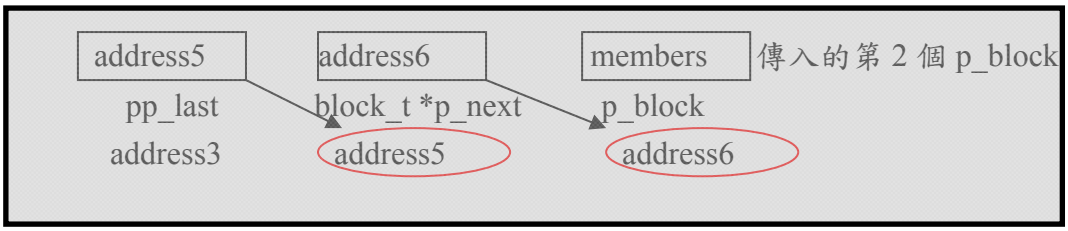


圖 4.22 記憶體位置關係示意圖四

假設最後會傳入 n-1 個 block_t 結構，其 linked list 關係會變成圖 4.23。我們可以很清楚地看到雙重指標 pp_last 扮演了很重要的角色。我們可以發現只用*p_fifo->pp_last = p_block 和 p_fifo->pp_last = &p_block->p_next 這兩行程式就可以巧妙的建立了資料 linked list 的架構，這是一個相當特別的技巧。

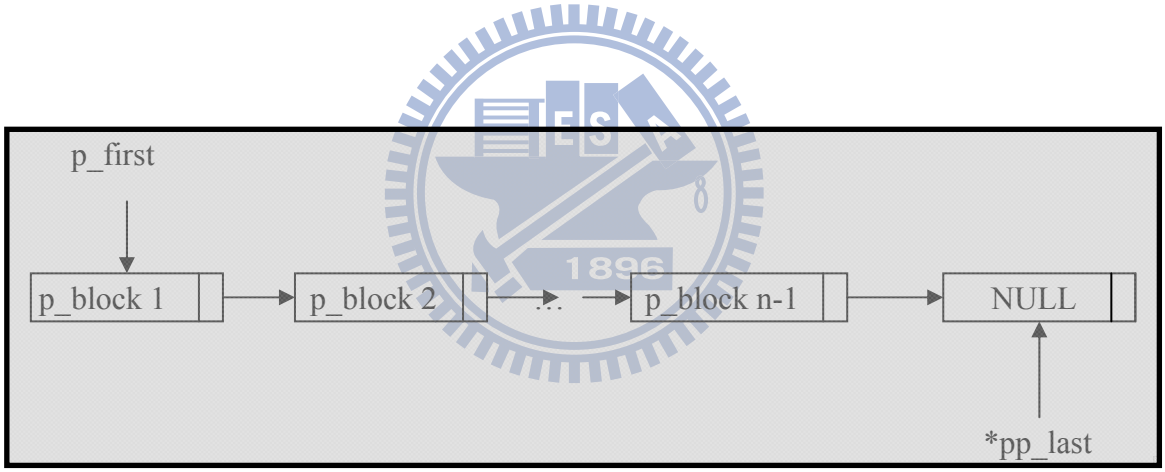


圖 4.23 第 n-1 次迴圈結束其 linked list 關係圖

在解多工器模組不斷用上述的方式將 audio stream 和 video stream 放入 FIFO buffer 的同時解碼器模組也會不斷的讀取 FIFO buffer 中的資料來進行解碼程序。解碼器模組會不斷呼叫 block_FifoGet()函數來讀取 FIFO buffer 內的資料並進行解碼程序。前面我們有提過由於解碼器模組和解多工器模組是在不同的 thread 下運作而且解多工器模組是要將影音資料分離給影像和聲音解碼器模組進行解碼程序，所以可能會有解多工器模組擺放資料的速度趕不上解碼器模組讀取資料的速度的情形發生，在這種情形下解碼器模組會停止讀取 FIFO buffer 的資料並進入等待的狀態，解多工器模組就有緩衝的時間將

資料放入 FIFO buffer 中後，解碼器模組才會再次去讀取 FIFO buffer 中的資料。

VLC 的解多工器模組在分離好的影音資料後就會利用 Init()函數中所設定好的資料傳送模組中的函數來傳送資料，同時會根據資料是影像或聲音的類型放入不同的 FIFO buffer 中，也就是說如圖 4.24 所示假設現在 ps.c 解多工器模組解析出 PES 中的 payload 是 video bitstream，那麼呼叫 block _FifoPut()函數時就會將此 payload 放入影像的 FIFO buffer 中，所以當在進行影像解碼程序時，就會呼叫 block_FifoGet()函數在影像的 FIFO buffer 中取出 video stream 並由 libmpeg2.c 解碼器模組中所定義的解碼函數來進行影像的解碼程序。

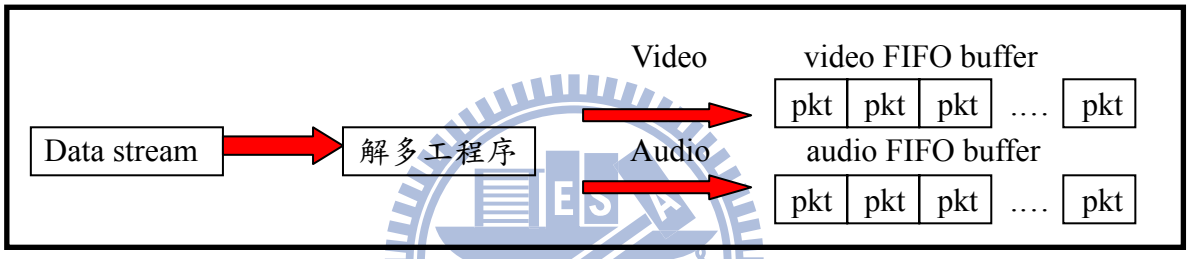


圖 4.24 傳送 bitstream 示意圖

4.4 呼叫輸出模組過程簡介

在解碼器模組開始解碼由解多工器分離的 audio stream 和 video stream 前，會先利用 module_Need()函數來尋找適合影像或聲音輸出模組。接下來我們會針對 libmpeg2.c 影像解碼器模如何找到影像輸出模組的整個過程做一個簡介。

首先在 CreateDecoder()函數中會先對解碼器模組所需的 callback functions 做設定圖 4.36，其中的 p_dec->pf_vout_buffer_new callback function 會指向 vout_new_buffer()函數，此函數是呼叫影像輸出模組的起點，然後是在 libmpeg2.c 的解碼主程式 DecodeBlock()函數中會被呼叫，如圖 4.25 所示。

```
/* Set buffers allocation callbacks for the decoders */
p_dec->pf_aout_buffer_new = aout_new_buffer;
p_dec->pf_aout_buffer_del = aout_del_buffer;
p_dec->pf_vout_buffer_new = vout_new_buffer;
p_dec->pf_vout_buffer_del = vout_del_buffer;
p_dec->pf_picture_link      = vout_link_picture;
p_dec->pf_picture_unlink    = vout_unlink_picture;
p_dec->pf_spu_buffer_new    = spu_new_buffer;
p_dec->pf_spu_buffer_del    = spu_del_buffer;
```

圖 4.25 設定解碼器模組輸出所需的 callback functions

```
/* Get a new picture */
p_pic = p_dec->pf_vout_buffer_new( p_dec );
```

圖 4.26 libmpeg2.c 呼叫輸出模組的程式

下面的圖 4.28 是整個從呼叫 vout_new_buffer()函數到找到影像輸出模組的整個過程，其過程會有很多的設定，vout_Request()函數就是會依據程式目前的特性去尋找適合的影像輸出模組，此時會呼叫 module_Need()函數找到一個適合的影像輸出模組並創一個新的影像輸出 thread，如圖 4.27 所示。

```
if( vlc_thread_create( p_vout, "video output", RunThread,
                       VLC_THREAD_PRIORITY_OUTPUT, VLC_TRUE ) )
{     msg_Err( p_vout, "out of memory" );
module_Unneed( p_vout, p_vout->p_module );
vlc_object_detach( p_vout );
vlc_object_destroy( p_vout );
return NULL;}
```

圖 4.27 創造輸出 thread

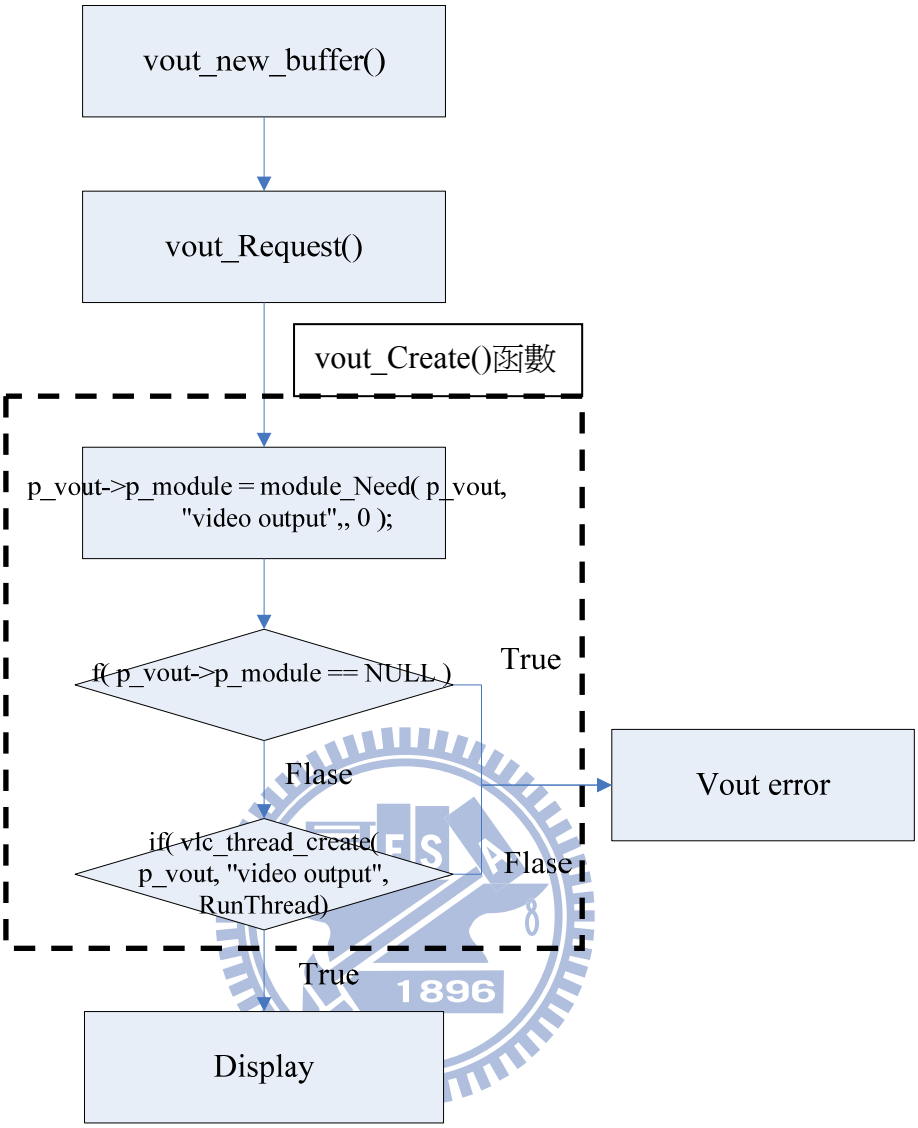


圖 4.28 呼叫輸出模組之流程圖

第五章 VLC 與 FFmpeg 模組的尋找機制與比較

在第三章中我們已經知道 module_Need()函數是用來尋找適合的模組，而且我們也對其尋找模組的機制有了清楚地了解，但是 VLC 是如何表示和定義每一個模組?又是如何讓 module_Need()函數可以依據影音檔案的不同而找到最佳的模組?這些都是值得去討論的議題，那麼其他的影音播放器又是如何去表示和定義其模組?其尋找模組的機制又為何?在 VLC 所包含的模組中有一個非常特別的模組，就是 FFmpeg，因為 FFmpeg 本身就是一套強大的多媒體播放器，而且可以播放大多的影音格式。所以本章會針對 VLC 和 FFmpeg 這兩個不同的播放器對模組的表示和定義以及模組尋找的機製作一個比較。

5.1 VLC 模組架構

VLC 中的模組巨集(macros)概念是在每個模組在編譯的過程中會分配一個記憶體空間用來表示此模組的特性，並對其模組的敘述、名稱、能力、開啟測試 callback functions 等等特性作設定。如此一來就可以在程式編譯完成後就建立好所有模組的特性組織，這樣一來 module_Need()函數就可以依據這些模組特性找到適合的模組，我們就以一個 Libmpeg2.c 的模組來舉例。其中 set_category 和 set_subcategory 是 VLC 用來分類模組用的，而 set_capability 是設定一個模組的功能和能力，在 module_Need()函數中是做為能力條件比對和排序的順序的依據。如果今天要多外掛一個自己寫的解碼器模組那麼就必須遵守 VLC 規定的模組格式來定義模組，所以就必須以巨集的方式來設定一個模組的特性，如此一來才能和 VLC 有一個溝通的標準，而且必須在 VLC 外部的資料文件中做一個註冊的動作和重新將 VLC 整個從頭再一次完整編譯後才可以使用，外掛模組的實驗過程會在第 6 章會詳細解說。

5.1.1 VLC 的模組的定義

在一開始的我們就提到 VLC 是可以支援許多不同檔案格式進行播放的多媒體播放器，所以一個支援多格式的多媒體播放器就必需針對不同的檔案格式選出適合的多媒體處理程序，譬如我們要用 VLC 播放一個 MPEG-2 的影音檔案就必需用 ps.c 解多工器模組來進行解多工程序，然後用 libmpeg2.c 解碼器模組進行影像解碼的程序，然而 VLC 中定義了好幾種不同種類的模組，那麼 VLC 是如何選定 ps.c 和 libmpeg2.c 模組來進行 MPEG-2 影音檔案的解多工和解碼程序呢?我們這裡會詳細說明一個模組的基本架構。

從圖 5.1 我們可以知道 VLC 會用結構來定義一個模組的架構，模組結構中包含了模組的名稱和基本敘述等等資訊，其中最重要的就是模組的能力和能力值，因為在 module_Need()函數中就是依據模組的能力來進行模組的分類，並用能力值來決定測試模組的先後順序，module_Need()函數要測試一個模組是否為適合的模組時都會判斷模組中測試函數是否有符合開啟條件。

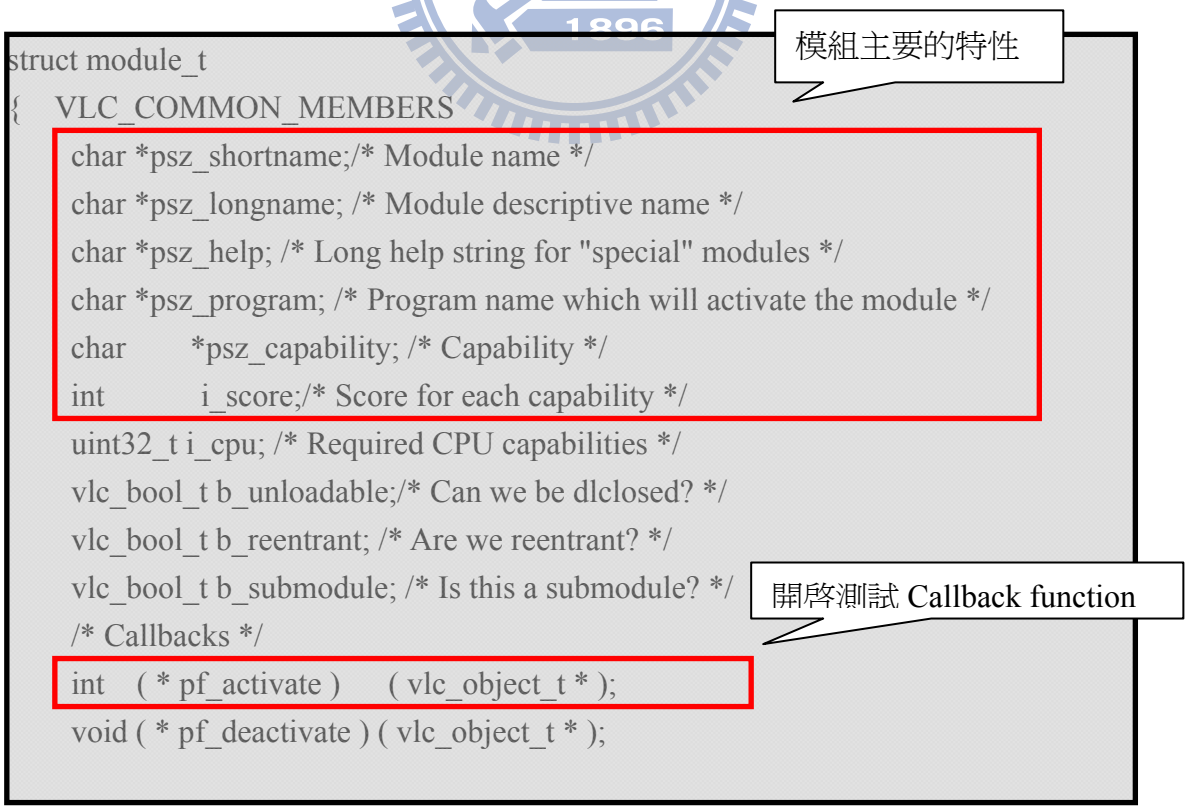


圖 5.1 型態 module_t 結構之成員是意圖

5.1.2 VLC 的模組的設定

我們已經知道 VLC 是如何表示一個模組的特性，但是一個模組結構在還沒被設定前都只一個空白標籤; 標籤上有模組的名稱和能力等等資訊需要被設定，所以現在我們更進一步來討論 VLC 如何用巨集的方式來設定模組的特性讓每個模組都有自己特定的標籤。如圖 5.2 所示我們會用實際的 libmpeg2.c 模組來說明 VLC 如何用巨集設定模組的過程。

```
vlc_module_begin();
    set_description( _("MPEG I/II video decoder (using libmpeg2)") );
    set_capability( "decoder", 150 );
    set_category( CAT_INPUT );
    set_subcategory( SUBCAT_INPUT_VCODEC );
    set_callbacks( OpenDecoder, CloseDecoder );
    add_shortcut( "libmpeg2" );
vlc_module_end();
```

圖 5.2Libmpeg2.c 巨集設定圖

在圖 5.3 中是用來設定 libmpeg2.c 模組的巨集識別字，可以清楚看到會用識別字 vlc_module_begin() 起頭，vlc_module_end() 結尾。模組的特性時會用巨集識別字 vlc_module_begin() 來對模組做初始化的動作，首先會先傳入由圖所示的模組結構 module_t ，然後會先對其模組的特性一一做初始化的動作。

```
    p_module->psz_shortname=NULL;
    p_module->psz_longname=MODULE_STRING;
    p_module->pp_shortcuts[0]= MODULE_STRING;
    p_module->i_cpu=0;
    p_module->psz_program=NULL;
    p_module->psz_capability= "";
    p_module->i_score = 1;
    p_module->pf_activate = NULL;
```

圖 5.3 vlc_module_begin()巨集部分展開程式圖

如圖 5.4 所示，識別字 set_capability("decoder", 150)就是以巨集的方式對 libmpeg2.c 模組的能力和能力值作設定，此時 libmpeg2.c 模組的能力執會被設定成〝decoder〞;能力值會設定成 150 分，set_callbacks(OpenDecoder, CloseDecoder)是來設定其開啟測試函數的 callback functions 指向由 libmpeg2.c 模組所定義的 OpenDecoder()函數。

圖 5.5 代表以個完整的 libmpeg2.c 模組，所以當 module_Need()函數在尋找 MPEG-2 檔案的影像解碼器時，第一步驟就是比較所有模組的能力找出能力被設定成〝decoder〞的模組，此時具有解碼功能的 libmpeg2.c 模組會因為能力被設定成〝decoder〞而被 module_Need()函數找出來並行 linked list 程序，接著 module_Need()函數會開始依序呼叫 linked list 中的解碼器模組的開啟測試函數，當叫到 libmpeg2.c 模組所定義的開啟測試函數 OpenDecoder()時就會比對成功並回傳成功訊息給 module_Need()函數，同時 libmpeg2.c 模組中所定義的影像解碼函數就會在進行 MPEG-2 檔案影像解碼程序時被呼叫。

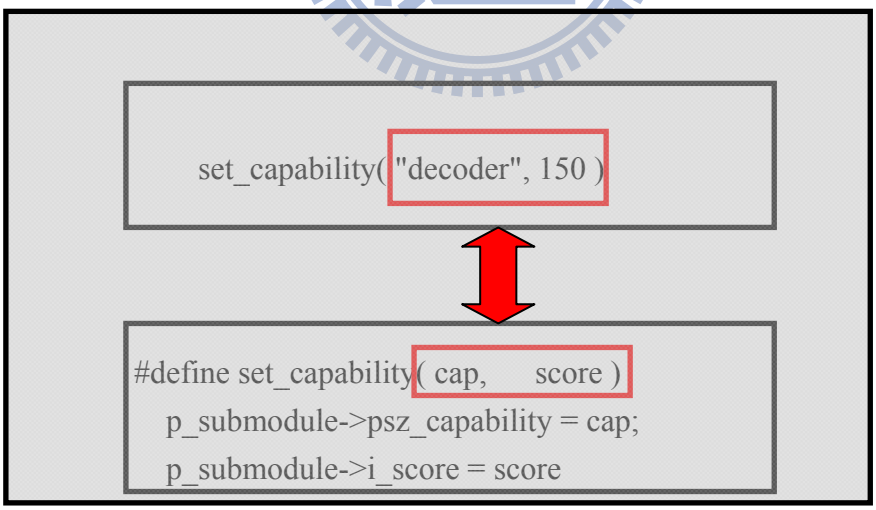


圖 5.4 set_capability()能力設定關係圖

由以上的結果我們可以做一些整理，當 VLC 要找尋找一些適合的處理程序時，就會尋找特定的模組來完成，而所謂的模組是由一些表示處理程序特性的結構和處理程序主要功能程式所組合在一起，將一個特定處理程序的資訊和功能用一個模組的方式來表

示我們就稱為〝模組化〞。VLC 中將許多不同的處理程序都模組化成模組，每個模組的特性都會由 Macros 機制設定，所以 module_Need()函數才能藉由每一個模組的特性來尋找適當的模組。我們用巨集的方式來設定模組主要的原因之一是方便撰寫程式，因為 VLC 中有 200 多個模組，若不用 Macros 的方式設定模組的特性的話，那就必須再每個模組的 .c 檔中重覆的寫上一大堆相同的設定模組特性的程式碼，若使用 Macros 的方式則可以使每個程式的.c 檔版面較為整齊，也大大的提高程式的維護性和可讀性。

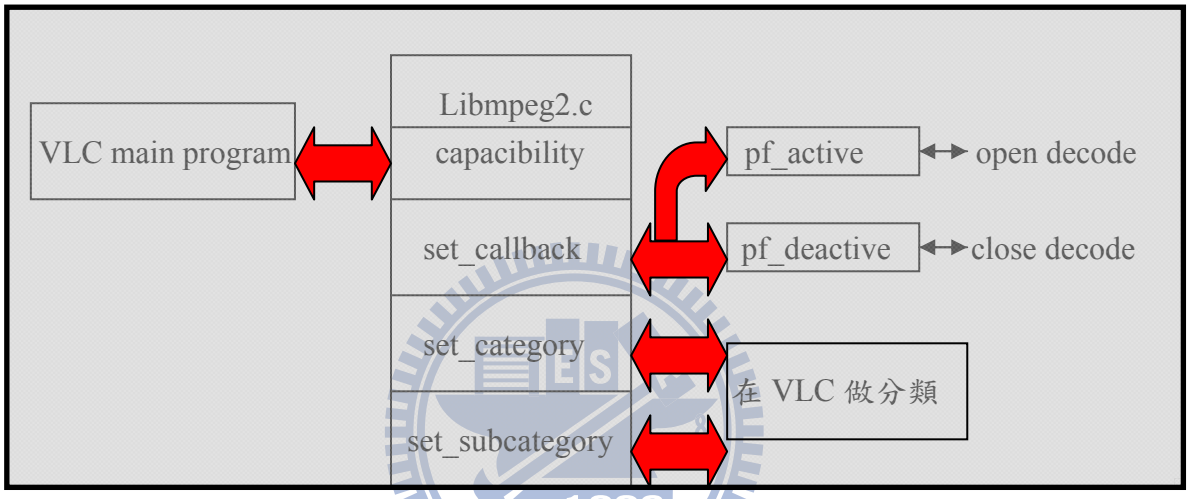


圖 5.5 利用 marco 建立基本的 module 組織示意圖

5.1.3 VLC 模組的 DLL 封裝原理

在 VLC 中所有模組是要將自己輸出給 VLC 的主要函數使用，為了程式執行的效率上的考量，所以就需要將許多模組做 DLL 的封裝，要使用 DLL 函數輸出只要在需要輸出的模組函數前面加上__declspec(dllexport)就可以達到模組的 DLL 封裝，如圖 5.6 所示。

```
__declspec(dllexport) void module_func(void);
```

圖 5.6 函式輸出之 DLL 語法

```
Step1:
Modules_inner.h 中有定義
# define DLL_SYMBOL                    __declspec(dllexport)
Step2:
在每個模組的巨集有定義
#    define EXTERN_SYMBOL                extern "C"
#define vlc_module_begin( )
 DECLARE_SYMBOLS;
 EXTERN_SYMBOL DLL_SYMBOL int CDECL_SYMBOL
 __VLC_SYMBOL(vlc_entry) ( module_t *p_module )
 {
```

圖 5.7 在 VLC 中 DLL 語法

圖 5.7 所示，其中的 extern "C"字串是被定義成 EXTERN_SYMBOL，主要是用來處理在 DLL 的隱式連結中使用不同的 C 編譯器會產生不同的結果的問題，所以會用 extern "C"來修正。

VLC 的程式碼在利用 cygwin 編譯完成後就會產生一個 vlc.exe 檔，然而由於 VLC 有 200 多個模組，所以 vlc.exe 檔不可能將所有的模組的程式碼都包進一個.exe 檔內，因為若這麼做的話，那麼整個 VLC 執行起來必定會變的沒效率，而且在管理和維護程式時會變的困難。所以 VLC 中的 200 多個模組都會產生各自的 DLL 檔來以外掛的方式將自己輸出，所以在 VLC 中主程式是用 DLL 的方式來和所需的模組連結在一起，現在就來討論 DLL 在 VLC 中的特性和存在之意義。

1 有效率的重複使用程式碼

由於解多工器和解碼器模組都是不斷的被主程式呼叫來處理資料，故重複被呼叫的

次數多，可藉 DLL 來有效率的重複呼叫解多工和解碼的程式碼。

2 區分程式碼

可以輕鬆的區分主程式和模組程式並可以降低多媒體播放器程式的複雜度，以及方便修改和維護程式。

3 節省記憶體的使用量

由於 VLC 有 200 多個模組，所以利用 DLL 的方式來連接可以節省寶貴的記憶體空間，增加執行的效率。VLC 中是使用了隱式連結（Implicitly Link），所謂的隱式連結是指主程式和模組之間的溝通是由編譯器來負責，程式設計者不需考慮主程式和模組之間是如何溝通。以下是其優缺點。

隱式連結 DLL 優點：

1.靜態載入方式所使用到的這個 DLL 會在應用程式執行時載入，然後就可以呼叫出所有由 DLL 中匯出的函式就好像是包含在程式中一般。

2. 動作較為簡單，載入的方法由編譯器來負責處理，程式設計者不須額外動腦筋。

隱式連結 DLL 缺點：

1.當這個程式靜態載入方式所使用到的這個 DLL 不存在時，這個程式在開始時就出現無法找到 DLL 的訊息而 導致應用程式無執行。

2. 編譯時需要加入額外的 import library。

3. 若是要載入的 DLL 一多，載入應用程式的執行速度會變慢。

圖 5.8 所示，在 VLC 中每個模組被編譯成功後就會各自產生 DLL 檔，並由編譯器來負責主程式和被 DLL 包裝的模組程式的連接。然後 module_Need()函數中就會尋找那些由 DLL 連接好的模組，在需要用到時就會將其模組載入。如此一來不但可以增加程式的執行效率以及避免浪費寶貴的記憶體，並且對程式的維護和可讀性來說都有很大的幫助，所以 DLL 的機制是非常的重要。

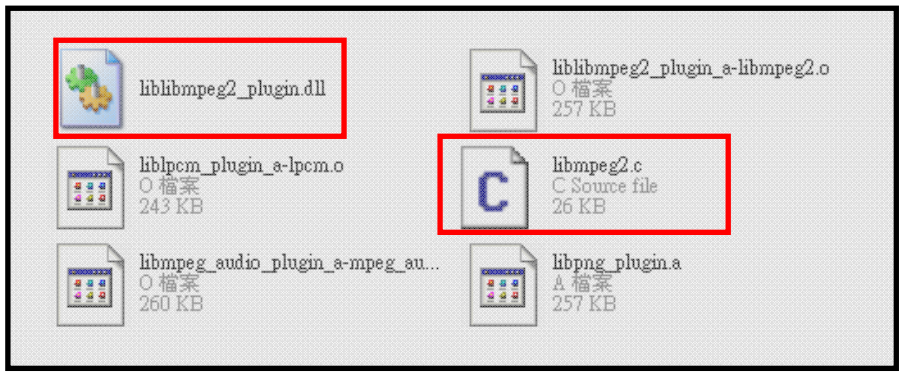


圖5.8 Libmpeg2.c產生之對應DLL圖

5.2 VLC 中的 FFmpeg

1. ffmpeg 介紹

FFmpeg 是一套功能強大的多媒體開發工具軟體，可以用於 video stream 與 audio stream 的分離,也就是 demux 的功能、encode、decode，主要是由以下幾個部份所組成。

1 .1 ffmpeg，是個轉碼工具，也就是將某一個影音格式轉碼成另一個影音格式，例如 ffmpeg –i inputfile.mpg outputfile.mp4 就是將一個.mpg 檔轉換成.mp4 檔。

1.2 ffserver，是個串流伺服器，藉由 HTTP 進行即時串流。

1.3 ffplay，是一個用 FFmpeg 函式庫所開發出來的多媒體播放器。

1.4 libavcodec 是 FFmpeg 所提供的函數庫，幾乎涵蓋了所有影音格式的編碼及解碼功能，程式開發者只要將此函數庫引入 project 即可使用其所提供的編碼/解碼功能。

1.5.libavformat 是 FFmpeg 所提供的函數庫，利用這個函式庫將聲音流與影像流從檔案中抽取出來或是將壓縮過的聲音流與影像流結合成某種格式的檔案。

2. VLC 中 ffmpeg 扮演的角色

在 VLC 中 FFmpeg 扮演一個具有解碼、編碼和解多工等等的多功能的角色，不同於前面所討論的模組，都只具有單一的功能。在 VLC 中 ffmpeg 可以視為是以解碼為主要功能其餘皆為附加功能的多功能模組。如圖 5.9 所示，ffmpeg 模組利用巨集的方式將其他的功能給附加進來，然而其影像和聲音的解碼程式和解多工程式是個別寫在一個 video.c 和 audio.c 以及一個 demux.c 中並在 ffmpeg.c 中以〝#include<ffmpeg.h>〞的方式將 video.c 和 audio.c 中的函數包進 ffmpeg.c 中。所以 VLC 的主程式才能間接的利用了由 ffmpeg 的巨集所產生的 DLL 檔來和 ffmpeg 的 video 解碼程式和 audio 解碼程式來連接。我們可以從下圖來觀察其中的關係。

```
add_submodule();
    set_section( N_("Encoding") , NULL );
    set_description( _("FFmpeg audio/video encoder") );
    set_capability( "encoder", 100 );
    set_callbacks( E_(OpenEncoder), E_(CloseEncoder) );
        ..............
add_submodule();
    set_description( _("FFmpeg demuxer" ) );
    set_capability( "demux2", 2 );
    set_callbacks( E_(OpenDemux), E_(CloseDemux) );
```

圖 5.9 巨集附加模組之語法

上圖可見我們利用巨集識別字 add_submodule()所展開的函數會對 ffmpeg 做附加功能的子模組功能和其能力以及開啟測試 callback functions 等等的設定。

```
/* Video decoder module */
int    E_( InitVideoDec )( decoder_t *, AVCodecContext *, AVCodec *,int, char * );
void E_( EndVideoDec ) ( decoder_t * );
picture_t *E_( DecodeVideo ) ( decoder_t *, block_t ** );
/* Audio decoder module */
int    E_( InitAudioDec )( decoder_t *, AVCodecContext *, AVCodec *,int, char * );
void E_( EndAudioDec ) ( decoder_t * );
aout_buffer_t *E_( DecodeAudio ) ( decoder_t *, block_t ** );
```

圖 5.10 ffmpeg.h 定義 video 和 audio 解碼程式圖

上圖 5.10 中我們可以知道#include<ffmpeg.h>定義了 ffmpeg 中 video decoder 和 audio decoder 的函數，所以在 ffmpeg.c 的主程式中才能直接的使用。所以 ffmpeg.c 本身就已經建立好一套完整的系統，再藉由 DLL 的方式外掛給 VLC 的主程式來進行呼叫的動作

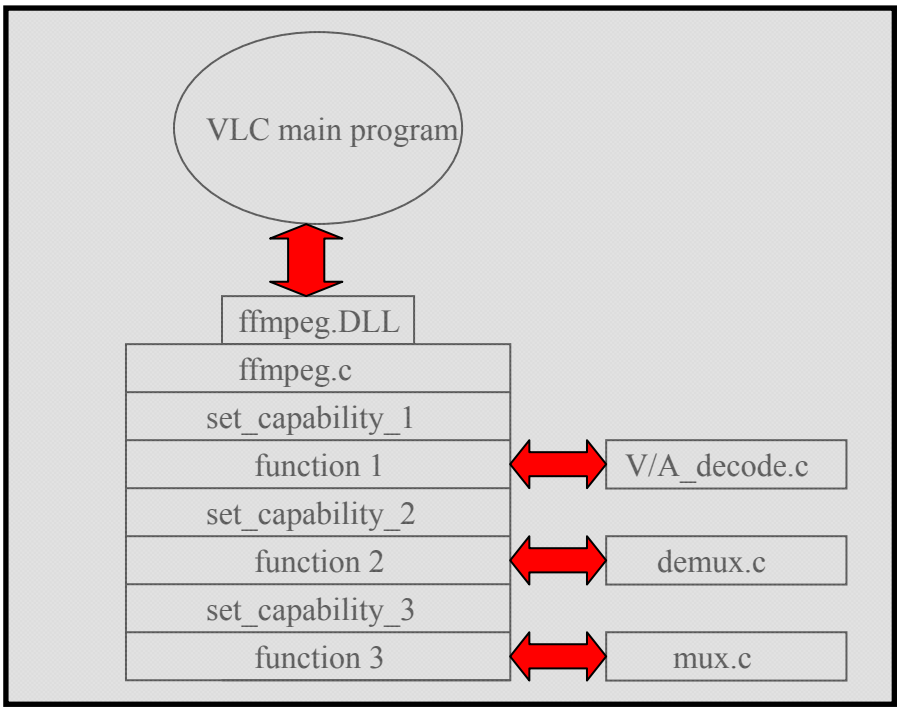


圖 5.11ffmpeg.c 模組是意

從關係圖 5.11 中我們可以知道，若當 VLC 中需要 ffmpeg 來進行解碼程序時，首先會在 module_Need()函數中進入到 ffmpeg.c 中的開啟測試函數 OpenDecoder()來判斷 ffmpeg 是否可以對由解多工模組解多工後的 video stream 或 audio stream 編碼格式進行解碼的程序，若不是適合格式就會回傳失敗訊息給 module_Need()函數，若有適合的格式情形下就會設定其影像和聲音解碼程序所需的 callback functions，其解碼函數都是另外定義在各自的.c 檔中。

我們在後面會對 ffmpeg 中的重要函數來做一些說明。在 ffmpeg 的解多工器模組開啟測試函數中會執行到 av_register_all()函數;av_register_all()函數會建立起 ffmpeg 中解多工器模組的 linked list，下面我們會針對 av_register_all()函數做一個詳細的說明。

5.2.1 FFmpeg 模組的定義與設定

Ffmpeg 模組的定義和 VLC 有些不同，以 FLV 格式的解多工器模組來舉例，如圖 5.12 所示，是用一個型態結構 AVIntputFormat 來定義一個 FLV 的解多工器模組。在 AVIntputFormat 結構成員是由許多的不同型態指標變數和 callback functions 所組成，在針對不同的解多工器模組會有不同的設定。

在圖中的對應關係中我們可以很清楚地看到 AVIntputFormat 結構成員的設定方式。所以我們可以很清楚知道 ffmpeg 就是利用結構的方式定義解多工器模組。下面我們會討論 av_register_all()函數是如何建立解多工器模組的 linked list。

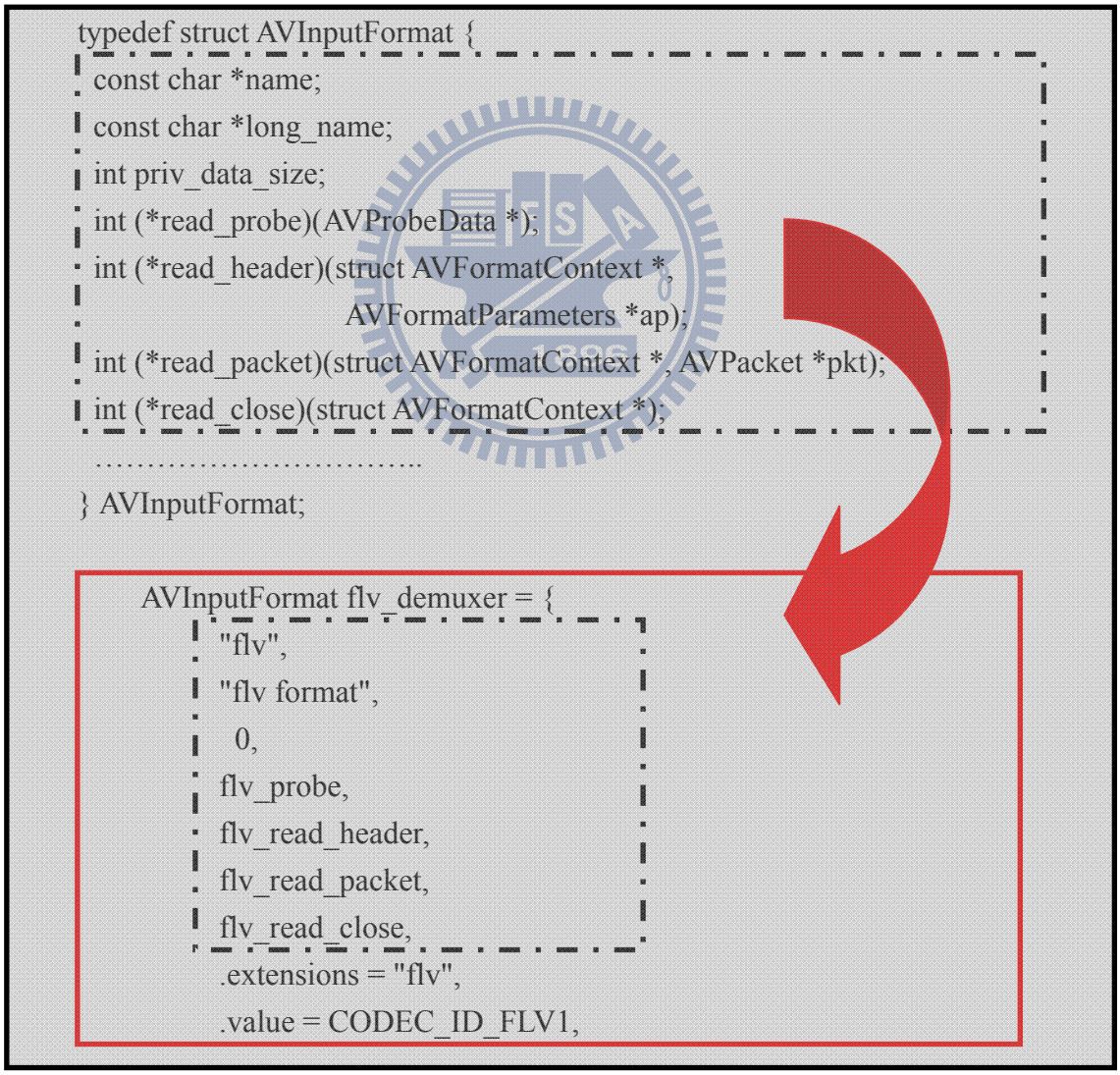


圖 5.12 AVIutputFormat t 結構變數成員設定對應圖

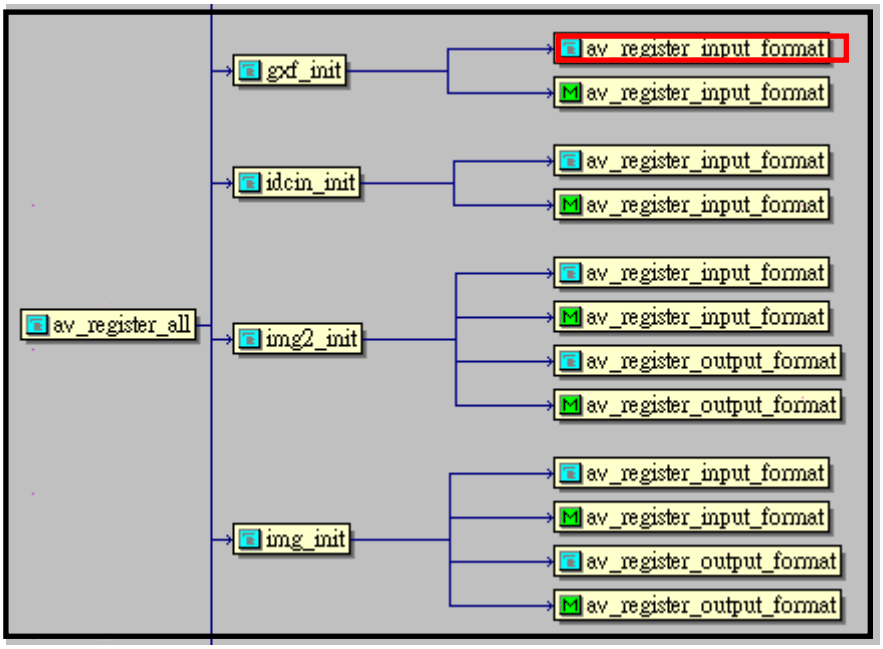


圖 5.13 av_register_all()函數關係圖

如圖 5.13 所示，從 av_register_all()函數關係圖來看，會先呼叫一連串解多工器模組的初始函數，譬如 gxf_int()、img2_init()、flvdec_init()等等，然而這些初始函數都會呼叫 av_ register_input_format()函數並傳入每個解多工器模組各自的 AVIutputFormat 結構成員。所以 av_register_all()函數就是利用 av_register_input_format()函數來建立解多工器模組的 linked list。

```
void av_register_input_format(AVInputFormat *format)
{
 AVInputFormat **p;
 p = &first_iformat;
 while (*p != NULL) p = &(*p)->next;
 *p = format;
 format->next = NULL;
 }
```

圖 5.14 av_register_input_format()函數

如圖 5.15 所示，我們假設已經有 linked list 好的解多工器模組並要加入 flv 解多工器模組的情形來舉例;flvdec_init()函數會呼叫 av_ register_input_format()函數，然後會將圖 5.13 所示的 flv 解多工器模組的記憶體位置傳入 av_input_format()函數中，並定義一個型態 AVInputFormat 的雙重指標變指向全域變數 first_iformat 的位置。全域變數 first_iformat 是指向 linked list 好的解多工器模組第一個模組，接著指標變數會利用一個 while()迴圈來找出 linked list 好的解多工器模組的尾端後就會加入 flv 解多工器模組。

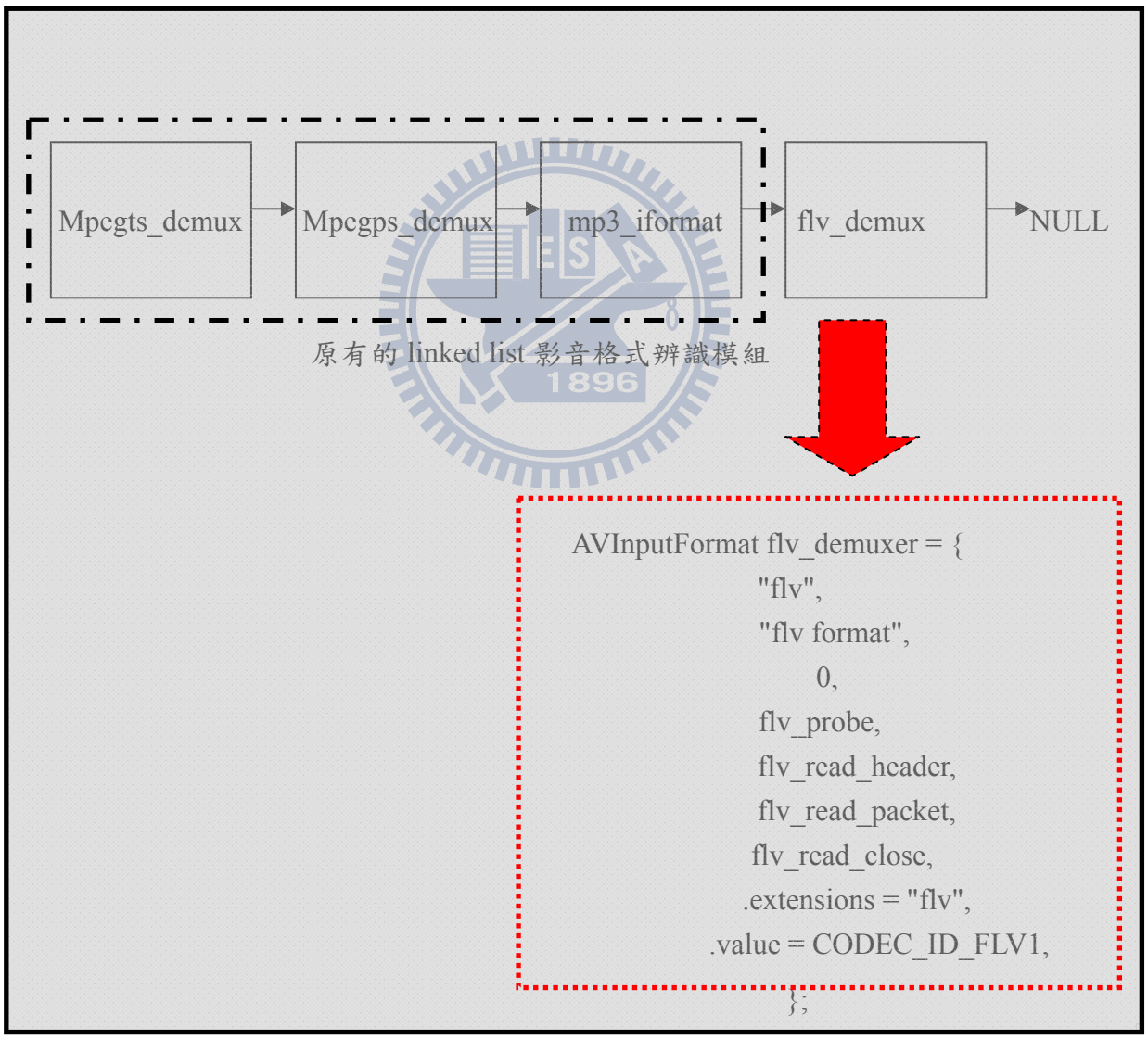


圖 5.15 解多工器模組 linked list 關係圖

我們可以很清楚地看到 ffmpeg 就是利用 av_register_all()函數對所有的解多工器模組做 linked list，並呼叫每個解多工器模組的開啟測試函數作測試。

在 VLC 中是先利用排序分數的方式來決定測試模組的先後順序，ffmpeg 則是利用程式中模組初始函數的執行先後順序來決定模組在 linked list 中的先後順序。但是相同是都有分數的機制，最後都是選擇分數最高的模組，但是 VLC 因為事先有對模組做分數作排序的動作，所以不用像 ffmpeg 必須測試 linked list 中全部的模組才能找到分數最高的模組，所以在尋找模組的技巧來說 VLC 在效能上似乎有比較好的表現。

5.2.2 FFmpeg 尋找模組機制

在 ffmpge 將所有的解多工器模組都做好 linked list 後，就會呼叫 av_probe_input_format()函數來判定我們的影音資料是何種格式的包裝。如圖 5.16 所示，在呼叫 av_probe_input_format() 函數時會先傳入已經讀好的標頭檔資料的型態結構 AVProbeData，接著會利用指標變數來指向做好 linked list 的解多工器模組，並利用一個 for()迴圈來將 AVProbeData 指標變數傳入每個解多工器模組中 read_probe()函數中來進行影音格式的判斷。若判斷成功就會回傳一個模組分數，並判斷此模組分數是否大於目前最高的模組分數，最後會回傳分數最高的解多工器模組的位址。不同於 VLC 的是 ffmpeg 就算是已經找到可以開啟的解多工器模組也不會停止尋找，直到整個 linked list 中的解多工器模組都測試完畢。

```
for(fmt1 = first_iformat; fmt1 != NULL; fmt1 = fmt1->next) {
    if (!is_opened && !(fmt1->flags & AVFMT_NOFILE))
        continue;
    score = 0;
    if (fmt1->read_probe) {
    score = fmt1->read_probe(pd);
            ..............................
```

圖 5.16 av_probe_input_format()函數部份程式

我們會實際用 FLV 的解多工器模組的例子來說明 ffmpge 整個尋找解多工器模組的過程，FLV(Flash Video)是依據其格式而命名的，通常都用於網路上影片串流用，其 FLV 也可能被嵌入在 swf 的檔案中。在傳統的 FLV 檔案中通常都是擺放 H.264 和 ACC 編碼格式的 stream。每個 FLV 檔案中的 tag 都代表了一個單一的 stream 所以一個 tag 只能包含一個 audio 或 video stream。

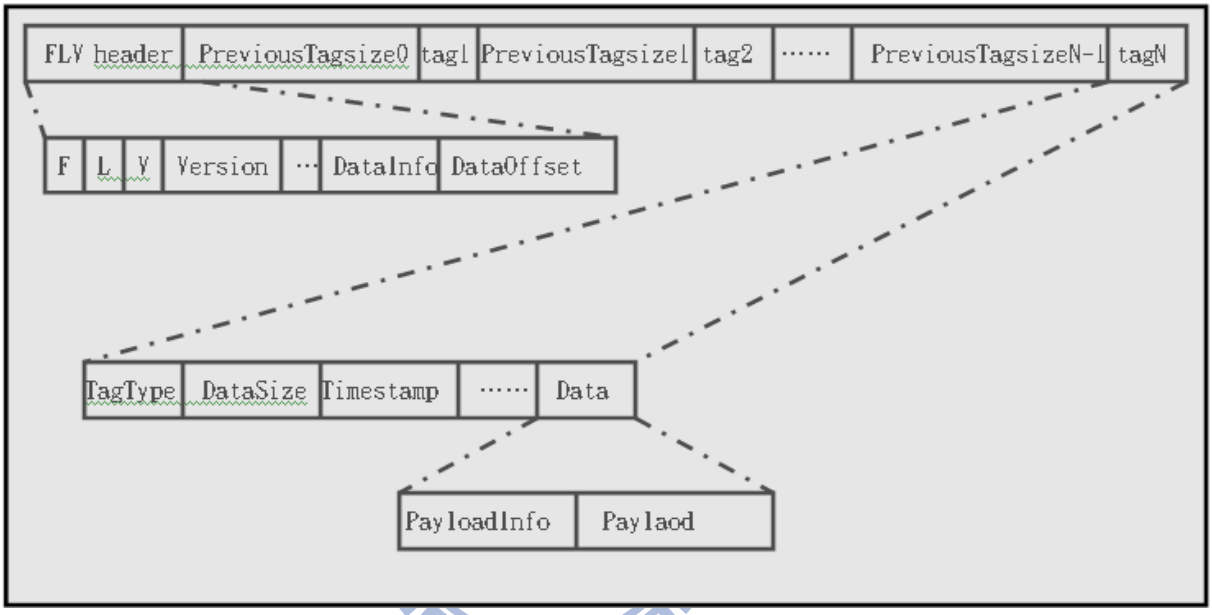


圖 5.17 FLV 檔案格式示意圖

如圖 5.17 所示，在 FLV hearder 中的前 3 個位元組分別是字元 F、 L、 V，這 3 個字元是 ffmpeg 中用來判定是否為 FLV 影音格式的依據。第 4 個位元組則是用來判定 FLV 的版本。DataInfo 則表示了 tags 的訊息。DataOffset 則是代表了 hearder 的大小。

在 FLV hearder 之後的部分我們稱為 FLV file body;FLV file body 是由一連串的 tags 和 PreviousTagsize 所組成，PreviousTagsize 是記錄著上一個 tag 的大小。

每個 tag 的第一個位元組紀錄著此 tag 是何種資料型態;若是 8 代表此 tag 是 audio payload，若是 9 則代表是 video paylaod，若是 18 代表是 script data。DataSize 佔了 3 個位元組;代表了 Data 資料的大小，Timestamp 佔了 3 個位元組;代表了此 tag 被應用的時間通常會和第一個 tag 的 Timestamp 有關係。最後 Data 則是 payload。

每個解多工器模組都會定義不同的 read_probe()函數。如圖 5.18 所示，在 FLV 解多工模組中的 read_probe()函數會判斷前 3 個位元組是否為 F、L、V 3 個字元，若判定為真就會回傳能力值 50，假設 FLV 解多工模組是能力值最高的模組情形下就會選定 FLV 解多工模組模組為適合解多工器模組。我們可以發現 ffmpeg 的 read_probe()函數就相當於 VLC 中模組的開啟測試函數。

```
static int flv_probe(AVProbeData *p)
{
  const uint8_t *d;
  if (p->buf_size < 6)
  return 0;
  d = p->buf;
if (d[0] == 'F' && d[1] == 'L' && d[2] == 'V') {
  return 50;
}return 0;
  }
```

圖 5.18 FLV 的 read_probe()函數

在確定使用 FLV 解多工器模組後，就會開始呼叫 read_header()函數解析 FLV 的 header 藉此判定 FLV 的格式和版本等等資訊，此時我們已經可以知道影音檔案是 FLV 包裝格式，但是我們還是不知道 payload 是 audio 或 video 資料以及編碼的格式，所以就會開始解析我們的 payload 編碼格式和資料型態。FLV 解多工器模組所定義的 read_packet()函數會不斷讀取 FLV 中 tags 的資訊，tag 中記錄了 payload 的型態和大小，最後還會判定 payload 是用何種編碼格式，如圖 5.19 所示，假設今天 payload 是一個 MP3 格式的聲音檔，那麼就會用 st->codec->codec_id= CODEC_ID_MP3 來儲存其聲音編碼格式，之後會依據儲存的聲音編碼格式來找到適合的解碼器模組。由前述我們可以發現 FFmpeg 解多工器中的成員 read_header()和 read_packet()函數相當於 VLC 解多工器模組中 Demux()函數的角色。

```
switch(flags >> 4){
        case 0: if (flags&2) st->codec->codec_id = CODEC_ID_PCM_S16BE;
                    else st->codec->codec_id = CODEC_ID_PCM_S8; break;
        case 1: st->codec->codec_id = CODEC_ID_ADPCM_SWF; break;
        case 2: st->codec->codec_id = CODEC_ID_MP3; break;
        // this is not listed at FLV but at SWF, strange...
        case 3: if (flags&2) st->codec->codec_id = CODEC_ID_PCM_S16LE;
                    else st->codec->codec_id = CODEC_ID_PCM_S8; break;
         default:
         v_log(s, AV_LOG_INFO, "Unsupported audio codec (%x)\n", flags >> 4);
               st->codec->codec_tag= (flags >> 4);
         }
```

圖 5.19 判定編碼格式

在確定 FLV 的 payload 為何種編碼格式後，如圖 5.20 所示，就會呼叫 avcodec_find_decoder()函數並傳入由 FLV 解多工器模組所提供的參數 CodecID id，然後在解碼器模組的 linked list 中尋找適合的解碼器模組。

```
AVCodec *avcodec_find_decoder(enum CodecID id)
{
AVCodec *p;
p = first_avcodec;
while (p) {
      if (p->decode != NULL && p->id == id)
           return    p;
      p = p->next;
}
 return NULL;
}
```

比較 CodecID

圖 5.20 avcodec_find_decoder()函數

5.3 VLC 與 FFmpeg 尋找模組機制與比較

VLC 是由結構來定義一個模組並用巨集的方式來設定一個模組內的成員。FFmpeg也是由結構來定義一個模組但是模組內的成員是由每個模組的.c檔自行去設定。VLC 和 FFmpeg 各自有定義模組的方式，完全看程式設計者的習慣。

在尋找模組方面，VLC 是用 module_Need()函數來搜尋適合的模組，module_Need()函數會先分類出相同能力的模組並依據模組能力值高低由高到低排序，再依序測試是否是適合的模組，而且若有找到適合的模組就不再測試剩餘的模組。然而 FFmpeg 在尋找適合的模組方面則有不同的機制，像是在尋找適合的解多工器和解碼器模組的方式不像 VLC 都是用 module_Need()函數，而是使用不同的函數來尋找不同能力的的模組，譬如 FFmpeg 是用 av_probe_input_format()函數來尋找適合的解多工器模組;av_probe_input_format()函數會呼叫 linked list 上每一個解多工器模組的開啟測試函數 read_probe()，若有模組的 read_probe()函數比對條件成功就會回傳一個能力值並進行能力值的比較，譬如在 linked list 上有三個模組的開啟測試函數可以比對條件成功，那麼就會這三個模組都會回傳一個能力值並進行比對，最後會回傳一個能力值最高的模組。而解碼器模組方面是用 avcodec_find_decoder()函數來尋找;avcodec_find_decoder()函數則是會依據適合的解多工器模組分析資料後所提供的解碼 ID 來比對在 linked list 中解碼器模組的解碼 ID 是否一樣，比對解碼 ID 相符合就會判定此解碼器模組為適合的解碼器模組，而比對解碼器模組中的解碼 ID 機制就像是 VLC 中比對解碼器模組中的四字元的機制類似，此函數跟 VLC 一樣的是一找到適合的解碼器模組也不會再測試剩下的解碼器模組。

VLC 和 FFmpeg 在尋找適合的解多工器模組時都有分數的機制，最後都是選擇能力值最高的模組，但是 VLC 因為事先有對模組做分數作排序的動作，所以不用像 FFmpeg 必須測試 linked list 中全部的模組才能找到分數最高的解多工器模組。但是以尋找模組的過程來說 VLC 似乎比較複雜，因為 Module_Need()函數的功能相當於整合了 FFmpeg 中許多設定模組的函數。VLC 雖然尋找模組過程較 FFmpeg 複雜但

是由於所有能力模組都是用 module_Need()函數來尋找，所以在程式的可讀性和維護性來說 VLC 有較好表現。

第六章 VLC 模組實驗

在 VLC 中並不支援 RM 格式的影片，若要看 RM 格式的影片就需要額外加入可以對 RM 格式的影片作解多工和解碼的模組才能達成，本章會以增加一組 real 解多工解多工器模組和 realvideo 解碼器的模組來達到 VLC 可以播放 RM 格式的影片的例子詳述在 VLC 中如何外掛模組的流程。

6.1 實驗步驟

原理:

因為 VLC 本身就將許多不同功能的程序進行模組化，所以才會有許多不同功能的模組，而模組跟模組之間存在了一套溝通機制;譬如在進行解多工程序時，解多工器模組會呼叫資料讀取系統模組來取得檔案的 Data stream 後再由解多工器模組將 Data stream 解析成 video bitstream 和 audio bitstream 後，就會呼叫資料輸出系統模組來把 bitstream 交給解碼器模組來進行解碼程序。所以今天假設要新加入一個解多工器模組時，我們要注意的是如何和前後級的模組進行溝通，如圖 6.1 所示。並為了能夠在進行模組的尋找時能被找到，所以我們定義新模組的特性時就必須遵守 VLC 定義模組的規則和機制。

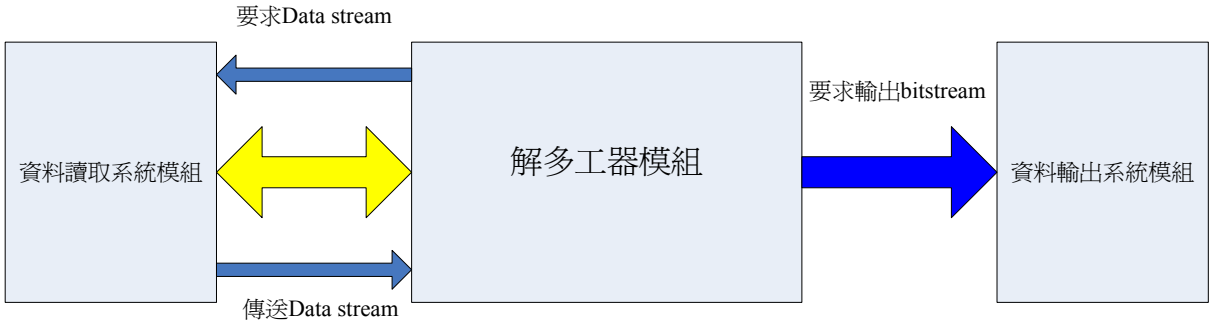


圖 6.1 解多工器模組與前後級模組關係圖

Step1:

把 realvideo.c 複製到 modules\codec 目錄下，然後把 real.c 複製到 modules\demux 目錄下，把原來的 real.c 程式移除。

Step2:

修改主目錄下面的 configure.ac 檔，把 565-567 行改為如下

AC_CHECK_LIB(m,pow,[

VLC_ADD_LDFLAGS([ffmpeg ffmpegaltivec stream_out_transrate i420_rgb faad twolame equalizer param_eq vlc freetype mpc dmo quicktime realaudio **realvideo** galaktos],[-lm])

])

把 2944-2946 行改為如下

if test "${enable_real}" = "yes"; then

VLC_ADD_PLUGINS([realaudio])

VLC_ADD_PLUGINS([realvideo])

fi

把 5092-5101 行改為如下：

AS_IF([test "${enable_loader}" = "yes"],

[VLC_ADD_PLUGINS([dmo quicktime])

VLC_ADD_CPPFLAGS([dmo],[-I../../../@top_srcdir@/loader])

VLC_ADD_LDFLAGS([dmo],[../../../loader/libloader.la])

VLC_ADD_CPPFLAGS([quicktime],[-I../../@top_srcdir@/loader])

VLC_ADD_LDFLAGS([quicktime],[../../loader/libloader.la])

VLC_ADD_CPPFLAGS([realaudio],[-I../../@top_srcdir@/loader -DLOADER])

VLC_ADD_LDFLAGS([realaudio],[../../loader/libloader.la])

VLC_ADD_CPPFLAGS([realvideo],[-I../../@top_srcdir@/loader -DLOADER])

VLC_ADD_LDFLAGS([realvideo],[../../loader/libloader.la])

])

把 modules\codec 目錄下的 Modules.am 檔的 31 行增加如下：

SOURCES_realvideo = realvideo.c

Step3

然後，建立一個 configure-vlc.sh 文件，其內容如下：

```
./bootstrap && \
PKG_CONFIG_PATH=/usr/win32/lib/pkgconfig\
CPPFLAGS="-I/usr/win32/include -I/usr/win32/include/ebml" \
LDFLAGS=-L/usr/win32/lib \
CC="gcc -mno-cygwin" CXX="g++ -mno-cygwin" \
./configure \
    --disable-gtk \
    --enable-nls --enable-sdl --with-sdl-config-path=/usr/win32/bin \
    --enable-ffmpeg --with-ffmpeg-mp3lame --with-ffmpeg-faac \
    --with-ffmpeg-zlib --enable-faad --enable-flac --enable-theora \
    --with-wx-config-path=/usr/win32/bin \
    --with-freetype-config-path=/usr/win32/bin \
    --with-fribidi-config-path=/usr/win32/bin \
    --enable-live555 --with-live555-tree=/usr/win32/live.com \
    --enable-caca --with-caca-config-path=/usr/win32/bin \
    --with-xml2-config-path=/usr/win32/bin \
    --with-dvdnav-config-path=/usr/win32/bin \
    --disable-cddax --disable-vcdx --enable-goom \
    --enable-twolame --enable-dvdread \
    --enable-real \
    --enable-debug
```

Step4

修改好後，在 cygwin 指令視窗下執行指令：

dos2unix configure-vlc.sh + enter
./configure-vlc.sh + enter
make+enter
make package-win32-base + enter

Step5

在 Step4 結束後會將 VLC 的原始碼封裝成一個 VLC 的資料夾，最後把 drv43260.dll

檔複製到所產生的 VLC 的資料夾中 plugins 目錄下，執行 VLC 就可以播放 rm 格式影片了。

以上的步驟我們可以想成是將外掛的 realvideo 和 real 模組做一個註冊的動作，這樣在 VLC 重新編譯後才會將 realvideo 和 real 這兩個新模組看成是 VLC 中的物件，所以在 module_Need()函數尋找適合的解 RM 格式的解碼器模組時才會找的到 realvideo 解碼器模組，下面我們就來針對 realvideo 解碼器模組在 VLC 的語法做一個討論。

首先就是模組特性的設定，在上一章我們有提到在 VLC 中就是利用巨集的方式來對模組特性作設定，所以我們外掛一個模組時也必須對此模組的特性用巨集的方式來設定並要遵守 module_Need()函數對模組的分類和測試機制，譬如定義 realvideo 模組特性時就必須設定能力字串為**"decoder"**，如此一來才可以在 module_Need()函數尋找模組時被找到，如圖 6.2 所示。

```
vlc_module_begin();
    set_description( _("RealVideo library decoder") );
    set_capability( "decoder", 10 );
    set_category( CAT_INPUT );
    set_subcategory( SUBCAT_INPUT_VCODEC );
    set_callbacks( Open, Close );
vlc_module_end();
```

圖 6.2 realvideo.c 巨集設定

再來就是解碼器模組中開啟測試函數的四字元的比較機制，module_Need()函數在尋找適合的解碼器模組時都是利用解多工器模組所判斷的四字元來做尋找適合的解碼器模組的依據，如圖 6.3 所示。所以 realvide 解碼器模組的開啟測試函數也就必需要有一套對 real 解多工器模組中所判斷的四字元來判定 realvideo 解碼器模組是否為適合的解碼器模組的機制，也就是必須符合 VLC 中尋找解碼器模組的機制。

```
 if( GetDWBE( &p_peek[30] ) == 0x10003000 ||
 GetDWBE( &p_peek[30] ) == 0x10003001 )
{fmt.i_codec = VLC_FOURCC( 'R','V','1','3' );
 }
 else if( GetDWBE( &p_peek[30] ) == 0x20001000 ||
   GetDWBE( &p_peek[30] ) == 0x20100001 ||
   GetDWBE( &p_peek[30] ) == 0x20200002 )
{
fmt.i_codec = VLC_FOURCC( 'R','V','2','0' );
}
            .............
```

圖 6.3 在 real.c 中分配 realvideo.c 相對的四字元

在獲得從解多工器模組所得到的四字元資訊後便會開始做四字元的比較，如圖 6.4 所示，若符合四字元的比較後就會開始對 realvideo 解碼器模組做解碼函數的設定。以上的步驟都是根據 VLC 中尋找解碼器模組的規則來做，所以當我們要開發一個外掛模組時就必須跟隨 VLC 的規則來撰寫我們的程式。

```
 switch ( p_dec->fmt_in.i_codec )
 {case VLC_FOURCC('R','V','1','0'):
  case VLC_FOURCC('R','V','2','0'):
  case VLC_FOURCC('R','V','3','0'):
  case VLC_FOURCC('R','V','4','0'):
  p_dec->p_sys = NULL;
  p_dec->pf_decode_video = DecodeVideo;
  return InitVideo(p_dec);
  default:
  return VLC_EGENERIC;
  }
```

圖 6.4 比較四字元和 callback functions 的設定

6.2 實驗結果

從圖 6.5 和 6.6 所示的實驗結果得知，經過 6.1 節的步驟後我們已經成功外掛了 RM real 解多工器模組和 realvideo 解碼器模組，並已經可以讀取和播放 RM 格式的檔案。

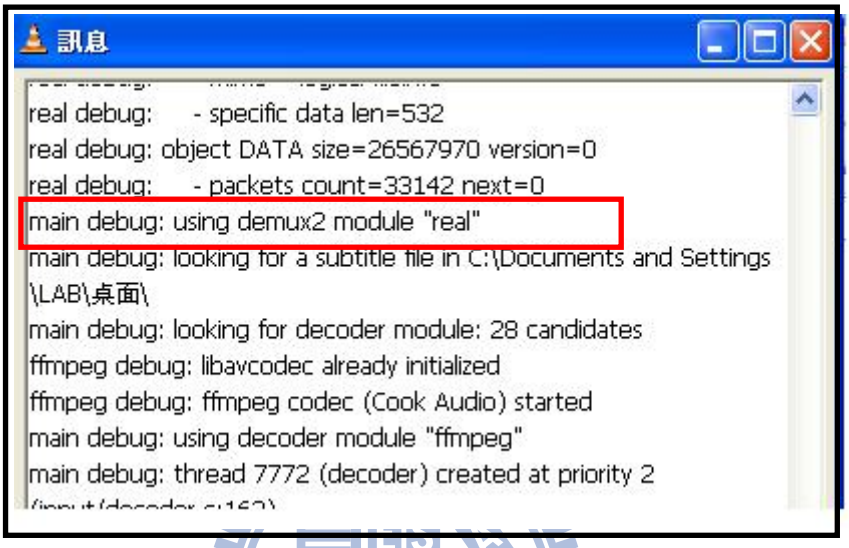


圖 6.5 成功外掛 RM 解多工器模組

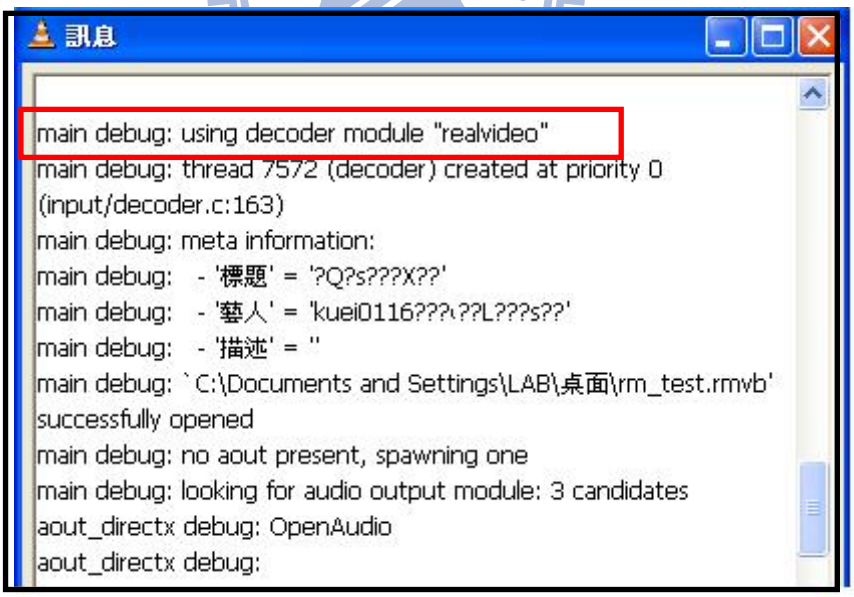


圖 6.6 成功外掛 RM 解碼器模組

第七章 結論

在本論文中我們討論了從使用者開啟 VLC 的執行檔產生一個 GUI 介面到 VLC 開始播放一個多媒體檔案整個的基本流程;以簡單的多媒體播放器架構為基礎,探討多媒體播放器運作的機制。我們以模組化為主軸去討論 VLC 中許多函數的重要性，從尋找資源存取、解多工器、解碼器等等模組的 module_Need()函數去介紹一個支援播放多格式的多媒體播放器是如何由影音檔案資訊去找到適合的模組。

目前許多的播放器都具有支援播放多格式影音檔案以及強大的解碼的功能，所以播放器會面臨到的問題是如何依據不同格式的影音檔案找到適合的解多工和解碼程序，其中也包括如何建立解多工、解碼程序跟讀取、傳送資料這些系統程序的溝通機制。所以播放器會將解多工和解碼程序以及讀取、傳送資料程序進行模組化，以解多工器模組來說;因為其中定義了模組的特性和解多工函數的 callback functions 以及讀取、傳送資料的系統模組，所以解多工器模組會呼叫讀取資料系統模組來取得影音的 data stream 後進行解多工程序，將 data stream 分成 audio bitstream 或 video bitstream，然後再呼叫傳送資料系統模組把 audio bitstream 或 video bitstream 交給解碼器模組進行解碼程序;當然解多工器模組中的模組特性和解多工函數的 callback functions 會因為不同的影音檔案格式有所不同，所以必須藉由一套完整模組搜尋的機制來決定。所以播放器可以藉由透過模組搜尋機制所找到的模組與系統模組之間的溝通建立起完整的影音播放程序。

所以本論文藉由研究 VLC 的 source code 去了解一個多媒體播放器是如何播放一個影音檔案;並如何藉由不同格式的檔案所提供的資訊去找到適合的解多工器、解碼器和輸出等等模組，然後藉由這些程序來了解功能模組化與模組的搜尋機制之間的關係和重要性，並以軟體工程方面來學習程式的技巧。

參考文獻

[1] VLC http://www.videolan.org/vlc/

[2] VLC user guide http://www.tldp.org/REF/VLC-User-Guide/

[3] VLC media player API Documentation
http://www.videolan.org/developers/vlc/doc/developer/html/manual.html

[4] Videolan's Wiki http://wiki.videolan.org/VideoLAN

[5] ISO/IEC 13818-2 Generic coding of moving picture and associated audio information: Video

[6] 陳瑩甄，『串流伺服器資料解析及封包包裝』，國立交通大學，碩士論文，96學年度。

[7] 郭明山，『播放器內解多工器與解碼器搜尋與比對』，國立交通大學，碩士論文，97學年度。

附錄 A

如果想要 build vlc from source code on windows，大致上有下列三個的方法：

1. 使用 cygwin(在 windowns 下 compile，建議用此方法)。
2. 使用 MSYS+MINGW。
3. 使用 Microsoft Visual C++。

在這邊我們只介紹如何用 cygwin 去 building soutce code of vlc。

編譯步驟:

1. 到 http://www.cygwin.com/ 下載 cygwin.exe，執行並安裝所有 packgae.
2. 到 http://www.videolan.org/ 下載 vlc source code，解壓縮至 C:\cygwin\home\kyo 目錄底下。
3. 到 http://download.videolan.org/pub/testing/win32/ ，下載 contrib-20061202-win32-bin-gcc-3.4.5-only.tar.bz2 並解壓縮之後，將資料夾裡的 win32 資料夾複製到 C:\cygwin\usr 目錄底下。
4. 開啟 UltraEdit(或筆記本)，儲存下列文字到 vlc source code 的資料夾裡(檔名存為 configure-vlc.sh)

```
CONTRIB_TREE=/usr/win32
PATH=${CONTRIB_TREE}/bin:$PATH \
./bootstrap && \
CPPFLAGS="-I${CONTRIB_TREE}/include -I${CONTRIB_TREE}/include/ebml" \
LDFLAGS=-L${CONTRIB_TREE}/lib \
PKG_CONFIG_LIBDIR=${CONTRIB_TREE}/lib/pkgconfig \
CC="gcc -mno-cygwin" CXX="g++ -mno-cygwin" \
./configure \
      --host=i686-pc-mingw32 \
      --enable-sdl --with-sdl-config-path=${CONTRIB_TREE}/bin --disable-gtk \
```

--enable-nls \
--enable-ffmpeg --with-ffmpeg-mp3lame --with-ffmpeg-faac \
--with-ffmpeg-zlib --enable-faad --enable-flac --enable-theora \
--with-wx-config-path=${CONTRIB_TREE}/bin \
--with-freetype-config-path=${CONTRIB_TREE}/bin \
--with-fribidi-config-path=${CONTRIB_TREE}/bin \
--enable-live555 --with-live555-tree=${CONTRIB_TREE}/live.com \
--enable-caca --with-caca-config-path=${CONTRIB_TREE}/bin \
--with-xml2-config-path=${CONTRIB_TREE}/bin \
--with-dvdnav-config-path=${CONTRIB_TREE}/bin \
--disable-cddax --disable-vcdx --enable-goom \
--enable-twolame --enable-dvdread \
--disable-gnomevfs \
--enable-dts \
--disable-optimizations \
--enable-debug \

5. 開啟 CYGWIN，輸入下列指令後按 enter 進到 vlc source code 資料夾裡：

   cd vlc.0.8.6i

6. 輸入下列指令後按 enter：

   dos2unix configure-vlc.sh

7. 輸入下列指令後按 enter：

   ./configure-vlc.sh

8. 執行下列指令後按 enter：

   make

9. 成功 compile vlc source code，並產生 vlc.exe 執行檔。

10. *Creating self contained packages：(optional)

    Once the compilation is done, you can either run VLC directly from the source tree or you can build self-contained VLC packages with the following "make" commands:

make package-win32-base

(This will create a subdirectory named vlc-x.x.x with all the binaries "stripped" without any debugging symbols).

make package-win32-zip

(Same as above but will package the directory in a zip file).

make package-win32

(Same as above but will also create an auto-installer package. You will need to have NSIS installed in its default location for this to work).

(註)*步驟 10 選擇性執行

11. cygwin 編譯 VLC 錯誤修正:

由於 cygwin 版本問題，如果依照前 10 步驟來建立 VLC 還會產生錯誤的話，請依照下面 cygwin 介面所提示的錯誤訊息加以修正。

問題 1：

gcc -mno-cygwin -Wsign-compare -Wall -mms-bitfields -pipe -o
libaccess_output_dummy_plugin.dll -g
-shared -u _vlc_entry__0_8_6 -L/usr/win32/lib libaccess_output_dummy_plugin.a -L/usr/local/lib
if gcc -mno-cygwin -DHAVE_CONFIG_H -I. -I. -I../.. -I/usr/win32/include
-I/usr/win32/include/ebm
l -D_OFF_T_ -D_off_t=long -DSYS_MINGW32 -I../../include `top_builddir="../.." ../../vlc-config
--c
flags plugin access_output_file` -Wsign-compare -Wall -mms-bitfields -pipe -MT
libaccess_output_f
ile_plugin_a-file.o -MD -MP -MF ".deps/libaccess_output_file_plugin_a-file.Tpo" -c -o
libaccess_ou
tput_file_plugin_a-file.o `test -f 'file.c' || echo './'`file.c; \

then mv -f ".deps/libaccess_output_file_plugin_a-file.Tpo" ".deps/libaccess_output_file_pl
ugin_a-file.Po"; else rm -f ".deps/libaccess_output_file_plugin_a-file.Tpo"; exit 1; fi
In file included from file.c:30:
/usr/lib/gcc/i686-pc-mingw32/3.4.4/../../../../i686-pc-mingw32/include/sys/stat.h:113: error: pars
e error before "off_t"
/usr/lib/gcc/i686-pc-mingw32/3.4.4/../../../../i686-pc-mingw32/include/sys/stat.h:118: error: pars
e error before '}' token
make[4]: *** [libaccess_output_file_plugin_a-file.o] 錯誤 1
make[4]: Leaving directory `/tmp/vlc-0.8.6i/modules/access_output'
make[3]: *** [all-modules] 錯誤 1
make[3]: Leaving directory `/tmp/vlc-0.8.6i/modules/access_output'
make[2]: *** [all-recursive] 錯誤 1
make[2]: Leaving directory `/tmp/vlc-0.8.6i/modules'
make[1]: *** [all-recursive] 錯誤 1
make[1]: Leaving directory `/tmp/vlc-0.8.6i'
make: *** [all] 錯誤 2

問題 1 解決辦法:
/usr/i686-pc-mingw32/include/sys 目錄下
在 stat.h 檔案起頭增加

```
#ifndef _OFF_T DEFINED
typedef long off_t;
#define _OFF_T DEFINED
#endif
```

問題 2：

directory `/tmp/vlc-0.8.6i/activex'
if g++ -mno-cygwin -DHAVE_CONFIG_H -I. -I. -I.. -I/usr/win32/include
-I/usr/win32/include/ebml -
D_OFF_T_ -D_off_t=long -DSYS_MINGW32 -I../include `top_builddir=".." ../vlc-config
--cxxflags acti
vex` -Wsign-compare -Wall -mms-bitfields -pipe -MT libaxvlc_a-main.o -MD -MP -MF
".deps/libaxvlc
_a-main.Tpo" -c -o libaxvlc_a-main.o `test -f 'main.cpp' || echo './'`main.cpp; \

```
  then mv -f ".deps/libaxvlc_a-main.Tpo" ".deps/libaxvlc_a-main.Po"; else rm -f ".deps/libax
vlc_a-main.Tpo"; exit 1; fi
In file included from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/cwchar:54,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/bits/postypes.h:46,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/iosfwd:50,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/bits/stl_algobase.h:70,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/vector:67,
  from utils.h:28,
  from main.cpp:24:
/usr/lib/gcc/i686-pc-mingw32/3.4.4/../../../../i686-pc-mingw32/include/wchar.h:419: error: `off_t'
does not name a type
make[3]: *** [libaxvlc_a-main.o] 錯誤 1
make[3]: Leaving directory `/tmp/vlc-0.8.6i/activex'
make[2]: *** [all] 錯誤 2
make[2]: Leaving directory `/tmp/vlc-0.8.6i/activex'
make[1]: *** [all-recursive] 錯誤 1
make[1]: Leaving directory `/tmp/vlc-0.8.6i'
make: *** [all] 錯誤 2
```

問題 2 解決辦法::
/usr/i686-pc-mingw32/include/目錄下

在 wchar.h 的文件中增加

```
#ifndef _OFF_T DEFINED
typedef long off_t;
#define _OFF_T DEFINED
#endif
```

問題 3：

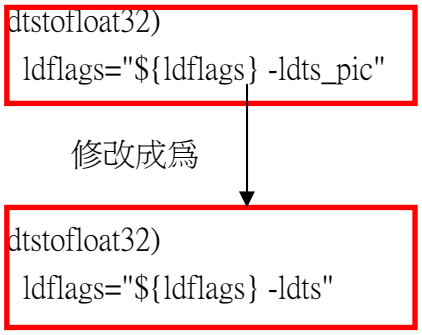
```
gcc -mno-cygwin -Wsign-compare -Wall -mms-bitfields -pipe -o libdtstofloat32_plugin.dll -g
-shared
-u _vlc_entry__0_8_6 -L/usr/win32/lib libdtstofloat32_plugin.a -L/usr/local/lib -ldts_pic
libdtstofloat32_plugin.a(libdtstofloat32_plugin_a-dtstofloat32.o): In function `Open':
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:219: undefined reference to `_dca_in
```

```
it'
libdtstofloat32_plugin.a(libdtstofloat32_plugin_a-dtstofloat32.o): In function `DoWork':
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:303: undefined reference to `_dca_syncinfo'
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:312: undefined reference to `_dca_frame'
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:331: undefined reference to `_dca_blocks_num'
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:335: undefined reference to `_dca_block'
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:341: undefined reference to `_dca_samples'
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:331: undefined reference to `_dca_blocks_num'
libdtstofloat32_plugin.a(libdtstofloat32_plugin_a-dtstofloat32.o): In function `Destroy':
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:376: undefined reference to `_dca_free'
libdtstofloat32_plugin.a(libdtstofloat32_plugin_a-dtstofloat32.o): In function `CloseFilter':
/tmp/vlc-0.8.6i/modules/audio_filter/converter/dtstofloat32.c:430: undefined reference to `_dca_free'
collect2: ld returned 1 exit status
make[6]: *** [libdtstofloat32_plugin.dll] 錯誤 1
make[6]: Leaving directory `/tmp/vlc-0.8.6i/modules/audio_filter/converter'
make[5]: *** [all-modules] 錯誤 1
make[5]: Leaving directory `/tmp/vlc-0.8.6i/modules/audio_filter/converter'
make[4]: *** [all-recursive] 錯誤 1
make[4]: Leaving directory `/tmp/vlc-0.8.6i/modules/audio_filter'
make[3]: *** [all] 錯誤 2
make[3]: Leaving directory `/tmp/vlc-0.8.6i/modules/audio_filter'
make[2]: *** [all-recursive] 錯誤 1
make[2]: Leaving directory `/tmp/vlc-0.8.6i/modules'
make[1]: *** [all-recursive] 錯誤 1
make[1]: Leaving directory `/tmp/vlc-0.8.6i'
make: *** [all] 錯誤 2
```

問題 3 解決方法：

./configure 後會產生 vlc-config 文件

修改 vlc-config 文件中



重新 make

問題 4：

gcc -mno-cygwin -Wsign-compare -Wall -mms-bitfields -pipe -o libflacdec_plugin.dll -g -shared -u _vlc_entry__0_8_6 -L/usr/win32/lib libflacdec_plugin.a -L/usr/local/lib /usr/win32/lib/libFLAC.a -lm /usr/win32/lib/libogg.a
/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x3fb): undefined reference to `__ntohl@4'

/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x488): undefined reference to `__ntohl@4'

/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x4e5): undefined reference to `__ntohl@4'

/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x57f): undefined reference to `__ntohl@4'

/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x5d5): undefined reference to `__ntohl@4'

/usr/win32/lib/libFLAC.a(bitwriter.o):bitwriter.c:(.text+0x66f): more undefined references to `__ntohl@4' follow
collect2: ld returned 1 exit status
make[4]: *** [libflacdec_plugin.dll] 錯誤 1
make[4]: Leaving directory `/tmp/vlc-0.8.6i/modules/codec'
make[3]: *** [all-modules] 錯誤 1
make[3]: Leaving directory `/tmp/vlc-0.8.6i/modules/codec'
make[2]: *** [all-recursive] 錯誤 1
make[2]: Leaving directory `/tmp/vlc-0.8.6i/modules'
make[1]: *** [all-recursive] 錯誤 1
make[1]: Leaving directory `/tmp/vlc-0.8.6i'
make: *** [all] 錯誤 2

問題 4 解決方法

./configure 後會產生 vlc-config 檔

修改 vlc-config 檔中

```
flacdec)
ldflags="${ldflags} -lFLAC"
```

修改成爲:

```
flacdec)
 ldflags="${ldflags} -lFLAC -lws2_32"
```

重新 make

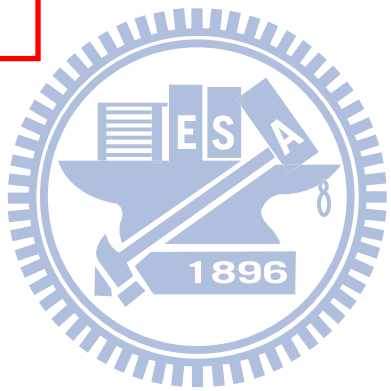
問題 5：

```
Making all in activex
make[2]: Entering directory `/tmp/vlc-0.8.6i/activex'
make all-am
make[3]: Entering directory `/tmp/vlc-0.8.6i/activex'
if g++ -mno-cygwin -DHAVE_CONFIG_H -I. -I. -I..  -I/usr/win32/include
-I/usr/win32/include/ebml -
D_OFF_T_ -D_off_t=long -DSYS_MINGW32 -I../include `top_builddir=".." ../vlc-config
--cxxflags acti
vex` -Wsign-compare -Wall -mms-bitfields -pipe -MT libaxvlc_a-main.o -MD -MP -MF
".deps/libaxvlc
_a-main.Tpo" -c -o libaxvlc_a-main.o `test -f 'main.cpp' || echo './'`main.cpp; \
  then mv -f ".deps/libaxvlc_a-main.Tpo" ".deps/libaxvlc_a-main.Po"; else rm -f ".deps/libax
vlc_a-main.Tpo"; exit 1; fi
In file included from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/cwchar:54,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/bits/postypes.h:46,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/iosfwd:50,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/bits/stl_algobase.h:70,
  from /usr/lib/gcc/i686-pc-mingw32/3.4.4/include/c++/vector:67,
  from utils.h:28,
  from main.cpp:24:
/usr/lib/gcc/i686-pc-mingw32/3.4.4/../../../../i686-pc-mingw32/include/wchar.h:419: error: `off_t'
```

does not name a type
make[3]: *** [libaxvlc_a-main.o] 錯誤 1
make[3]: Leaving directory `/tmp/vlc-0.8.6i/activex'
make[2]: *** [all] 錯誤 2
make[2]: Leaving directory `/tmp/vlc-0.8.6i/activex'
make[1]: *** [all-recursive] 錯誤 1
make[1]: Leaving directory `/tmp/vlc-0.8.6i'
make: *** [all] 錯誤 2

問題 5 解決方法:
/usr/i686-pc-mingw32/include/目錄下
在 wchar.h 的檔案起頭增加

```
#ifndef _OFF_T DEFINED
typedef long off_t;
#define _OFF_T DEFINED
#endif
```



附錄 B

1. VLC 中 demuxer 整理

1.1 a52.c:

處理檔案類型:WMV 檔

設定解碼四字元: VLC_FOURCC('a', '5', '2', ' ')

1.2 aiff.c

處理檔案類型:音頻交換格式(Audio Interchange File Format,AIFF)

設定解碼四字元: VLC_FOURCC('t', 'w', 'o', 's')

1.3 aiff.c

處理檔案類型:ISDN 音頻檔案

設定解碼四字元:

8-bit ISDN u-law	VLC_FOURCC('u','l','a','w')
8-bit linear PCM	VLC_FOURCC('t','w','o','s')
16-bit linear PCM	VLC_FOURCC('t','w','o','s')
24-bit linear PCM	VLC_FOURCC('t','w','o','s')
32-bit linear PCM	VLC_FOURCC('t','w','o','s')
32-bit IEEE floating point	VLC_FOURCC('a', 'u', 0, AU_FLOAT)
64-bit IEEE floating point	VLC_FOURCC('a', 'u', 0, AU_DOUBLE)
4-bit CCITT g.721 ADPCM	VLC_FOURCC('a', 'u', 0, AU_ADPCM_G721)
CCITT g.722 ADPCM	VLC_FOURCC('a', 'u', 0, AU_ADPCM_G722)
CCITT g.723 3-bit ADPCM	VLC_FOURCC('a', 'u', 0, AU_ADPCM_G723_3)
CCITT g.723 5-bit ADPCM	VLC_FOURCC('a', 'u', 0, AU_ADPCM_G723_5)

1.4 dts.c

處理檔案類型:5.1 聲道音頻檔案

設定解碼四字元: VLC_FOURCC('d','t','s',' ')

1.5 aiff.c

處理檔案類型:完整壓縮音頻檔案

設定解碼四字元:VLC_FOURCC('t', 'w', 'o', 's')

1.6 mjpeg.c

處理檔案類型: M-JPEG camera 檔案

設定解碼四字元: VLC_FOURCC('m','j','p','g')

1.7 mod.c

處理檔案類型: MOD 檔案

設定解碼四字元: VLC_FOURCC('t', 'w', 'o', 's')
VLC_FOURCC('a', 'r', 'a', 'w')

1.8 mpc.c

處理檔案類型: 處理檔案類型: MusePack 檔案

設定解碼四字元:VLC_FOURCC('f', 'l', '3', '2'));
VLC_FOURCC('s', '3', '2', 'b'));
VLC_FOURCC('s', '3', '2', 'l'));

1.9 nsv.c

處理檔案類型:NullSoft 檔案

設定解碼四字元: VLC_FOURCC('s', 'u', 'b', 't')
VLC_FOURCC('a', 'r', 'a', 'w')

1.10 nuv.c

處理檔案類型: Nuv 檔案

設定解碼四字元: VLC_FOURCC('m','p','g','a'))

1.11 ogg.c

處理檔案類型: ogg 檔案

設定解碼四字元: VLC_FOURCC('t','h','e','o')
VLC_FOURCC('v','o','r','b')
VLC_FOURCC('c','m','m','l')
VLC_FOURCC('v','o','r','b')
VLC_FOURCC('s','p','x',' ')
VLC_FOURCC('f','l','a','c'))

1.12 ps.c

處理檔案類型:MPEG-PS 檔案

設定解碼四字元: VLC_FOURCC('d','t','s',' ')
VLC_FOURCC('a','5','2',' ')
VLC_FOURCC('s','p','u',' ')
VLC_FOURCC('l','p','c','m')
VLC_FOURCC('o','g','t',' ')
VLC_FOURCC('c','v','d',' ')
VLC_FOURCC('h','2','6','4')
VLC_FOURCC('m','p','4','v')
VLC_FOURCC('m','p','g','v')
VLC_FOURCC('m','p','4','a')
VLC_FOURCC('m','p','g','a')
VLC_FOURCC('m','p','g','v')
VLC_FOURCC('m','p','g','a')

1.13 pav.c

處理檔案類型: PVA 檔案

設定解碼四字元:VLC_FOURCC('m', 'p', 'g', 'v')
VLC_FOURCC('m', 'p', 'g', 'a')

1.14 rawdv.c

處理檔案類型: DV (Digital Video)檔案

設定解碼四字元: VLC_FOURCC('d','v','s','d')

1.15 subtitle.c

處理檔案類型: Text subtitles 檔案

設定解碼四字元: VLC_FOURCC('s','s','a',' ')
VLC_FOURCC('s','u','b','t')

1.16 tta.c

處理檔案類型: TTA 檔案

設定解碼四字元: VLC_FOURCC('T', 'T', 'A', '1')

1.17 ts.c

處理檔案類型:MPEG-TS 檔案

設定解碼四字元:VLC_FOURCC('m', 'p', 'g', 'v'))
VLC_FOURCC('m', 'p', 'g', 'a')
VLC_FOURCC('m', 'p', '4', 'a')
VLC_FOURCC('m', 'p', '4', 'v')
VLC_FOURCC('h', '2', '6', '4')
VLC_FOURCC('a', '5', '2', ' ')
VLC_FOURCC('s', 'p', 'u', ' ')
VLC_FOURCC('l', 'p', 'c', 'm')
VLC_FOURCC('s', 'd', 'd', 's')
VLC_FOURCC('d', 't', 's', ' ')
VLC_FOURCC('a', '5', '2', 'b')
VLC_FOURCC('s', 'p', 'u', 'b')
VLC_FOURCC('s', 'd', 'd', 'b')

1.18 ty.c

處理檔案類型: TY Stream audio/video 檔案

設定解碼四字元: VLC_FOURCC('m', 'p', 'g', 'v')
VLC_FOURCC('a', '5', '2', ' ')
VLC_FOURCC('m', 'p', 'g', 'a')

1.19 vobsub.c

處理檔案類型: Vobsub subtitles parser 檔案

設定解碼四字元: VLC_FOURCC('s','p','u',' ')

1.20 voc.c

處理檔案類型: VOC 檔案

設定解碼四字元: VLC_FOURCC('u','8',' ',' ')
VLC_FOURCC('u','1','6','l')
VLC_FOURCC('s','8',' ',' ')
VLC_FOURCC('s','1','6','l')

1.21 wav.c

處理檔案類型: WAV 檔案

設定解碼四字元: VLC_FOURCC('a','r','a','w')

VLC_FOURCC('p','c','m',' ')

VLC_FOURCC('a','f','l','t')

VLC_FOURCC('a', 'r', 'a', 'w'):

VLC_FOURCC('a', 'f', 'l', 't'):

VLC_FOURCC('u', 'l', 'a', 'w'):

VLC_FOURCC('a', 'l', 'a', 'w'):

VLC_FOURCC('m', 'l', 'a', 'w'):

VLC_FOURCC('p', 'c', 'm', ' '):

VLC_FOURCC('m', 's', 0x00, 0x02):

VLC_FOURCC('m', 's', 0x00, 0x11)

VLC_FOURCC('m', 's', 0x00, 0x61):

VLC_FOURCC('m', 's', 0x00, 0x62):

VLC_FOURCC('m', 'p', 'g', 'a'):

VLC_FOURCC('a', '5', '2', ' '):

1.22 xa.c

處理檔案類型:XA 檔案

設定解碼四字元: VLC_FOURCC('X','A','J',0)