國 立 交 通 大 學

電信工程學系

碩 士 論 文

播放器內解多工器與解碼器搜尋與比對

Searching and Comparing of Demultiplexer and Decoder in Player

研 究 生：郭明山

指導教授：張文鐘 博士

中 華 民 國 九十七 年 八 月

播放器內解多工器與解碼器搜尋與比對
Searching and Comparing of Demultiplexer and Decoder in Player

研 究 生: 郭明山　　Student: Ming-Shan Kuo
指導教授 : 張文鐘　博士　　Advisor: Dr. Wen-Thong Chang

國 立 交 通 大 學
電信工程學系
碩 士 論 文

A Thesis
Submitted to Department 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 2008

Hsinchu, Taiwan, Republic of China

中 華 民 國 九十七 年 八 月

播放器內解多工器與解碼器搜尋與比對

研究生：郭明山　　指導教授：張文鐘　博士

國立交通大學

電信工程學系碩士班

摘　　要

一個媒體播放器的播放程序基本架構，是透過合適的解多工器(demultiplexer)將影音資料分離出音訊位元流(audio bitstream)及視訊位元流(video bitstream)，然後再使用合適的解碼器(Decoder)分別對 audio bitstream 及 video bitstream 來解碼，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螢幕及喇叭。

而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解多工器來解析影音檔案的包裝格式，以及尋找並使用合適的解碼器來針對影像和聲音的編碼格式進行解碼。如果不能夠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播放影音檔案。

因此在本論文中我們希望藉由研究 VLC 這個 open source，來了解多媒體播放器的基本播放架構以分析 VLC 是如何尋找並使用合適的解多工器以及解碼器。

Searching and Comparing of Demultiplexer and Decoder in Player

Student: Ming-Shan Kuo Advisor: Dr. Wen-Thong Chang

Department of Communication Engineering
National Chiao Tung University

Abstract

The basic structure of the playing procedure of the media player is that audio bitstream and video bitstream were separated from the media file through appropriate demultiplexer first. Then by using appropriate decoder, the player decodes audio bitstream and video bitstream separately. After decoding, last the player sends audio output to the speaker and video output to the monitor.

During playing the media file with the media player, the main two problems are that how to search for an appropriate demultiplexer to parse content of the media file, and how to search for an appropriate decoder to decode audio and video bistream. If the media player can not accurately search for an appropriate demultiplexer and decoder, the media player can not paly the media file.

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 demultiplexer and decoder through studying open source “VLC media player”.

誌謝

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 范國清教授、黃仲陵教授以及廖維國教授於口試的指導，有了您們的指導才使得此篇論文能更趨於完備。

另外要感謝的是實驗室裡的同學們，包含blue、文賢、俊權、政達、振偉，這兩年一起修課，在實驗室唸書寫報告趕論文的日子裡，感謝他們為我帶來了美好的回憶。另外，還要感謝的是夸克、志偉、盛如、honda、秉謙、建民、小瘋，實驗室有你們這些學弟們，讓實驗室的氣氛更加活潑。

最後，我要感謝我的父母親、兩個妹妹和其他好友們，以及我的女友的支持與鼓勵，總讓我能心無旁騖地從事研究工作，並在我遇到挫折與低潮時讓我能夠重新振作；也感謝住了兩年的室友維庭跟冠勳，懷念與你們一起吃宵夜的時光，再一次的謝謝大家。

誌於 2008.08 風城 交大

Ming-shan

目錄

中文摘要 i
英文摘要 ii
誌謝 iii
目錄 iv
表目錄 vi
圖目錄 vii
第一章 緒論 1
1.1 研究背景與動機 1
1.2 論文章節提要 2
第二章 VLC多媒體播放器介紹 3
2.1 媒體影音檔案介紹 3
2.2 VLC介紹 6
2.3 VLC特色及支援格式 8
2.4 VLC原始碼目錄結構 9
2.4.1 LibVLC library： 9
2.4.2 模組(module)介紹 9
2.5 VLC 主程式架構 11
第三章 VLC播放流程架構介紹 14
3.1 VLC運作機制 15
3.2 檢查目前播放狀態 17
3.3 播放影音程序 20
3.3.1 分解MRL(Media Resource Locator) 22
3.3.2 尋找合適的存取模組 22
3.3.3 尋找合適的解多工器模組 24
3.3.4 進行解多工程序 25
3.3.5 尋找合適的解碼器模組 26
3.4 進行聲音及影像解碼程序 28
第四章 尋找合適模組機制 32
4.1 何謂模組(module) 32
4.2 module_Need()函數運作分析 34
4.2.1 傳入變數說明 35
4.2.2 第一階段--找出candidate modules 36
4.2.2.1 part 1 與part 2 說明 37
4.2.2.2 part 3 與part 4 說明 37
4.2.3 第二階段—尋找出合適的模組 40
4.3 尋找合適解多工器模組程序 42
4.3.1 尋找合適的MPEG-2 PS解多工器模組 43

4.3.1 尋找合適的MPEG-2 TS解多工器模組 44
4.4 尋找合適的解碼器模組程序 45
4.4.1 取得四字元碼 46
4.5 解碼器模組整理 52
第五章 實驗結果 64
5.1 ps module解多工函數說明 64
5.2 實驗數據 66
第六章 結論 71
參考文獻 72
附錄 73

表目錄

表 2.1 stream_id assignments.....48
表 2.2 stream type assignment.....49
表 5.1：pack header.....65
表 5.2：data buffer的變化.....67
表 5.3：block 資料狀態.....68
表 5.4：解析完block之後的資料變化.....69
表 5.5：video 鏈結串列.....70

圖目錄

圖 2.1：VideoLAN 串流應用架構 7
圖 2.2：VLC main()函數 11
圖 2.3：GUI interface on windows 13
圖 2.4：skins2 interface on windows 13
圖 3.1：播放流程基本架構 14
圖 3.2：產生VLC GUI後thread開啟的程序 15
圖 3.3：播放影音時thread開啟的程序 15
圖 3.4：RunThread()函數流程圖 17
圖 3.5：選擇開啟檔案資料來源 19
圖 3.6：改變p_playlist->status.i_status 20
圖 3.7：input thread Run()函數 20
圖 3.8：InputSourceInit()函數程式流程 21
圖 3.9：呼叫MRLSplit()函數分解MRL 22
圖 3.10：呼叫module_Need()尋找合適的access_demux module 23
圖 3.11：呼叫module_Need()尋找合適的access2 module 23
圖 3.12：讀取影音資料 24
圖 3.13：file module內所指定的讀取函數 24
圖 3.14：呼叫module_Need()尋找合適的demux2 module 25
圖 3.15：MainLoop()函數部分程式碼 25
圖 3.16：ps module內所指定的解多工函數 26
圖 3.17：尋找合適的decoder module程序 27
圖 3.18：mpeg-2 program stream 28
圖 3.19：解碼流程 29
圖 3.20：呼叫module_Need()尋找合適的decoder module 29
圖 3.21：libmpeg2 module內所指定的影像解碼函數 30
圖 3.22：DecoderDecode()內部分程式碼 30
圖 3.23：mpeg_audio module內所指定的聲音解碼函數 30
圖 3.24：DecoderDecode()內部分程式碼 31
圖 4.1：libmpeg2 decoder module定義 33
圖 4.2：module_Need()函數定義 34
圖 4.3：第一階段運作流程 36
圖 4.4：將所有module排序 37
圖 4.5：part 4 內的尋找程序 37
圖 4.6：p_module結構變數內的部分成員變數 38
圖 4.7：儲存符合功能需求的modules 39
圖 4.8：part 4 的搜尋程序 40
圖 4.9：第二階段程序之程式碼 41

圖 4.10：MPEG-2 PS 43
圖 4.11：ps module起始函數部分程式碼 43
圖 4.12：MPEG-2 TS 44
圖 4.13：ts module起始函數部分程式碼 45
圖 4.14：header of MPEG-2 PS 47
圖 4.15：ps_track_fill()函數內部分程式碼 50
圖 4.16：es_format_Init()函數內部分程式碼 50
圖 4.17：libmpeg2 module起始函數部分程式碼 51
圖 5.1：ps module解多工函數處理程序 64

第一章 緒論

1.1 研究背景與動機

由於多媒體影音技術不斷的進步及逐漸地蓬勃發展，各式各樣的影音檔案包裝格式和編碼格式也日益增多，然而針對各種影音的檔案格式及編碼格式的不同，可能就需要使用特定的媒體播放器來播放影音檔案，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 RealPlayer 播放器才可以播放；若想要觀看檔案包裝格式為「.mov」的影音檔案，那麼可能就需要使用 QuickTime 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)媒體播放器等等，這些播放器的共同特色都是強調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檔案格式的影片，而不需要在另外安裝使用其他特定的播放器。

而一般媒體播放器的播放程序基本架構是透過合適的解多工器(demultiplexer)將影音資料分離出音訊位元流(audio bitstream)及視訊位元流(video bitstream)，然後再使用合適的解碼器(Decoder)分別對 audio bitstream 及 video bitstream 來解碼，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螢幕及喇叭。

所以我們可以了解到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解多工器來解析影音檔案的包裝格式，以及尋找並使用合適的解碼器來針對影像和聲音的編碼格式進行解碼。因為針對不同的

影音包裝格式本來就要由不同的解多工器來解析影音資料；同樣地，針對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。即使一個播放器包含了許多的解碼器和解多工器，如果不能夠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播放影音檔案。因此在本論文中我們希望藉由研究 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。

1.2 論文章節提要

在了解多媒體播放器所要面臨到的問題之後，接下來的第二章我們會介紹媒體影音檔案以及 VLC 多媒體播放器的相關知識。第三章為描述 VLC 的基本播放程式流程架構。第四章主要討論的是 VLC 是如何解決所面臨到的播放器問題。第五章為實驗結果。第六章為本論文結論。

第二章 VLC 多媒體播放器介紹

所以在本章我們會簡單的說明關於媒體影音檔案的相關知識，然後介紹 VLC 這個多媒體播放器的由來、特色以及支援什麼樣的格式，最後會概述整個 VLC 播放器的原始碼目錄結構。

2.1 媒體影音檔案介紹

一個媒體影音檔案的文件格式包含了兩層涵意，一種是影音包裝格式(container type)，即是定義如何將各種媒體組織在一起，可能是單獨的音訊和視訊，如 MP3、M4V，或者是結合了音訊與視訊，像是 AVI、MPEG、QuickTime…等；另一個就是檔案內容本身的編碼格式(codec type)，也就是文件中的各種媒體內容是通過什麼樣的方式來完成編碼的，比如常見的 MPEG1、MPEG2、MPEG4、AC3 等等。

通常一個影音檔案都是由多個 bitstream 所組成的，每一個 bitstream 就是代表著一種資料類型，例如 audio 或是 video，這些視訊位元流(audio bitstream)或是音訊流(video bitstream)都是各自先經過不同編碼格式的影像編碼器或是聲音編碼器所形成，最後再利用多工器將視訊流和音訊流包裝起來，形成成某種影音檔案格式。

所以一般在多媒體播放器的運作上，必須要能夠先知道所存取影音檔案的影音包裝格式，才能依據不同的影音包裝格式特性進行解多工程序去分離出音訊流和視訊流，進而得到每一個 stream 的資料結構也就是編碼格式，播放器就會因應音訊流和視訊流不同的編碼格式對分別進行解碼動作，最後才能夠正確的解碼出影像和聲音並輸出到我們的螢幕和喇吧。因此我們可以了解到解多工器以及解碼器對於多媒體播放器的整個架構是佔了很重要的部份。

而目前常見的媒體檔案格式包括 MPEG-1、MPEG-2 外，另外還有 Real

Video、QuickTime、WMV、MPEG4 及 H.264…等格式，下面將我們介紹一些媒體檔案格式及相關編碼方式：

✧ MPG影音格式：

副檔名為 .MPG 的檔案，是由 MPEG (Moving Picture Experts Group) 組織所制定的影片檔，也是目前較為常見的影片格式；其影片內容以MPEG組織開發的影像壓縮技術製作而成。而根據使用的用途不同，MPEG 組織也發展出許多不同的壓縮技術，其中較常用的壓縮技術MPEG-1、MPEG-2、MPEG-4。

- ✓ MPEG-1：開始發展的時間為比較早，由於為了因應當時的硬體效能，所以壓縮過程較為簡單，大部分的播放器都可輕鬆應付其編解碼作業；但相對的因為壓縮的比例也比較低，因此同樣長度的檔案，所佔的硬碟容量會比較大。MPEG-1當時主要是用於VCD影片的儲存，所以在VCD的MPEGAV資料夾內的.DAT 檔，就是其壓縮方式可以視為MPG影片。
- ✓ MPEG-2：比MPEG-1較為先進的影像壓縮技術，具有十分優異的壓縮效能，相對的硬體需求也比較高。由於 MPEG-2 影片具有優異的壓縮效能，因此被高畫質的DVD所採用；一般DVD內的影片檔(.VOB 格式)，由於記錄了其他影片外的播放資訊，因此和 MPG 檔並不完全相容，軟體不見得都能支援。
- ✓ MPEG-4：這是專為網路多媒體應用所開發的壓縮技術，不過與其他兩種規格不同，MPEG 組織在開發MPEG-4之時，只發展了壓縮技術的內容，對於後續的檔案規格並未有進一步的規範，導致MPEG-4空有技術、卻無法實際應用。直到微軟自行發展新一代影片規格, MPEG-4才算真正問世。

✧ MOV 影音格式：

MOV 是由Apple公司所推出的影片檔案格式，主要是應用於網路播放， 具備一般串流影音即時播放的特性。由於MOV影片在檔案格式上有其優越之處，因此當初 MPEG組織在研發MPEG-4 技術時，也以 MOV 檔作為基本架構,不過現行實際使用的MPEG-4 技術和MOV並不相容。目前許多電影網站的預告片都是QuickTime影片，加上手機與Mac平台普遍都支援此格式，在

網路上的能見度頗高。

✧ RM/RMVB 影音格式：

RM 與RMVB都是由RealNetworks公司所開發的影音檔案格式，主要用於網路多媒體的應用。RM的起源較早，也是最早針對網路播放設計的影音格式，其特色就是可以『邊下載邊播放』，賦予網路多媒體全新的應用體驗。為了讓RM影片播放更加順暢，RealNetworks 後續不斷更改影片編碼技術，RMVB就是其最新的研發成果。RMVB 最大的特色是可變動的位元率設定，編碼程式會依據畫面內容，自行決定影片壓縮的比例，可以更有效縮減檔案體積但又不會影響畫質，目前是網路分享影片的主要格式。

✧ AVI 影音格式：

AVI（Audio Video Interleave）是一種音訊視訊交差記錄的數位視訊檔格式，是由微軟所制定的多媒體檔案規格。與其他影片格式不同，AVI 檔只能算是媒體影音資料的載具，對於影像、聲音編碼技術並未進一步規範，因此每個 AVI 都可以自由採用不同的編碼技術。正因為如此，也造成許多人播放影片時的困擾，同樣都是 AVI 檔，有些可以看、有些卻不能看。目前最常見到的 AVI 影片多半是採 MPEG-4 編碼，這類編碼技術由於沒有明確的標準，雖然衍生出各種編碼格式，但彼此間卻又不完全相容，造成使用者不小的困擾。

- ✓ MS MPEG-4：是微軟於 1998 年根據 MPEG-4 技術研發而成的編碼引擎，初期遷就當時的硬體配備，壓縮的品質並不理想，而後又推出 V2、V3 等版本，品質雖有改善但還不夠完美。直到 2004 年隨著 Media Player 10.0 問世，微軟順勢推出最新的MS MPEG4 V4，在品質與穩定性上都有十分出色的表現。
- ✓ DivX：MPEG-4編碼技術雖然先進，但實際的應用卻一直不理想，直到2000年研發出新一代的 DivX Codec 編碼器，MPEG-4才算是真正被推廣開來。DivX挾著MPEG-4高效率的壓縮效能，可以將接近 DVD 畫質的

影片，壓縮到一張VCD片上，間接帶動了網路影片分享的熱潮。

- ✓ Xvid：DivX 將MPEG-4編碼技術發揮的淋漓盡致，獲得極大的迴響，也讓DivX逐步邁向商業化的付費軟體。此舉違背了當初 DivX 開發的理念，因此在DivX興起的隔年，就有一部份程式設計師同樣以 MPEG-4 編碼為基礎，開發了另一套為Xvid編碼技術。

✧ WMV 影音格式：

由微軟公司主導所發展出來的多媒體視訊格式，也是微軟所有視訊編碼方式的通稱，該檔案的特點是檔案小，有利於網路上的即時傳送播放，是一種網路上常見的視訊格式。

✧ ASF 影音格式：

Advanced Streaming Formate 的簡稱，是微軟針對串流影音應用所提出的影音新規格。ASF 檔案可以是任何的編碼方式為基礎的影音編碼，而且仍然是 ASF 格式。然因為 ASF 檔案主要是為了提供在網路上直接觀看，所以品質較 VCD 來的差。

✧ 3GP 影音格式:

3GP 是手機專屬的影片檔，由手機大廠 Nokia 與 Apple 公司開發而成，目前是許多手機預設使用的影片格式。3GP 的編碼方式也是源於 MPEG-4技術。

2.2 VLC 介紹

VideoLAN計畫(VideoLAN Project)是一個開發多媒體影音串流播放程式的計畫，這個計畫最早是開始於 1999 年，是由巴黎中央理工學院學生所開發的，原本針對影音串流有兩個應用程式—

- VideoLAN Client (VLC):這是一個跨平台的客戶端應用程式，是用來接收

MPEG 串流影音資料的播放程式。

- VideoLAN Server (VLS):這是一個伺服端應用程式，可以將 MPEG 格式的影音資料串流到網路上。

然而之後VLS大部分的功能都被整合進VLC，使得VLS變得越來越不被重視，並且開始增加其他功能，例如可以存取電腦檔案以及光碟機等等，也支援了更多的影音包裝格式以及解碼格式，而最後便將VLC改稱為VLC多媒體播放器，VLS也逐漸不再發展及更新。並且VLC在遵從GPL(GNU Lesser General Public License)的協議條件之下於2001 年2 月 1 日發佈，現在計畫成員橫跨二十多個國家，下面圖 2.1 則是VideoLAN計畫的串流應用架構圖。

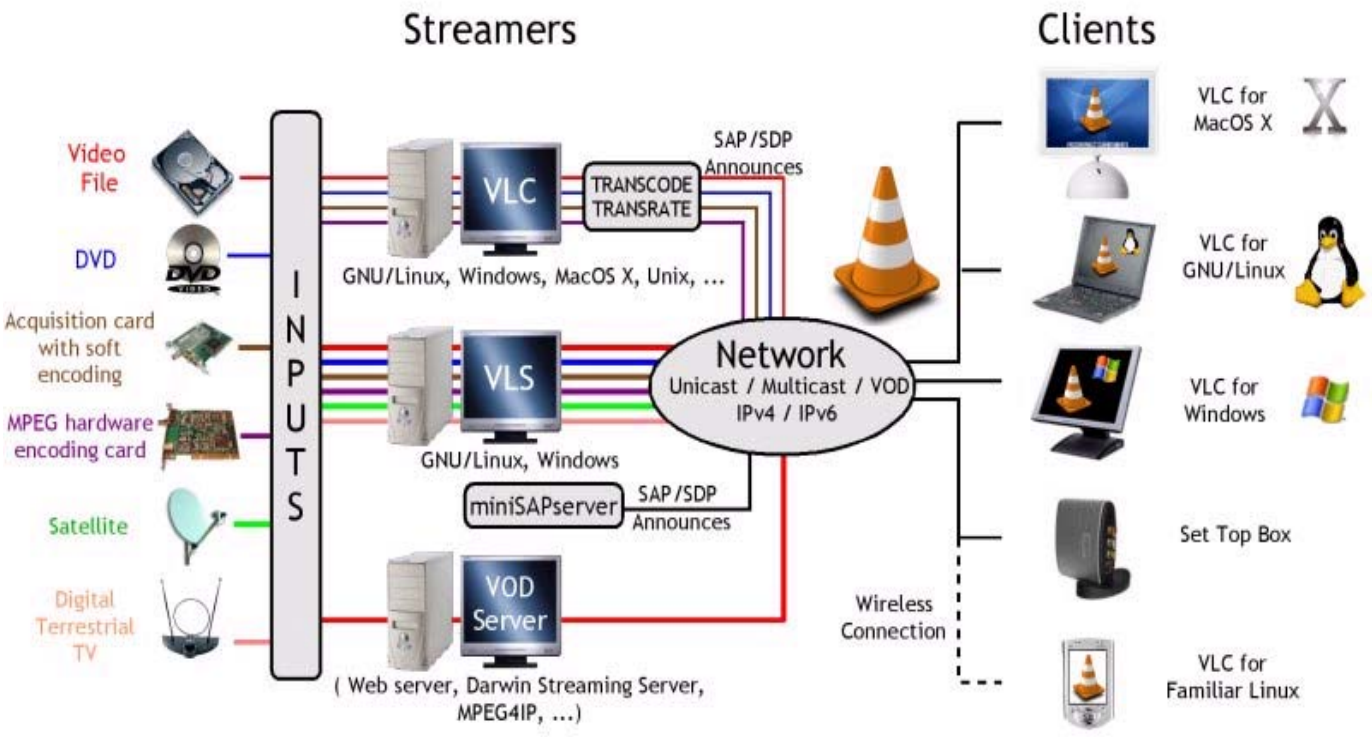

圖 2.1：VideoLAN 串流應用架構

2.3 VLC 特色及支援格式

現今的VLC多媒體播放器就是同時具備有客戶端與伺服端的能力，並且逐漸支持許多audio與video解碼器及檔案格式，例如MPEG-1， MPEG-2， MPEG-4，DivX，mp3，ogg等等，並支持DVD影碟，VCD影碟及各類串流協定。它亦能作為unicast 或 multicast的串流伺服器在IPv4 或 IPv6的高速網路連線下使用。對於不完整、未下載完成或損壞的影片都能順利撥放。而且VLC多媒體播放器也具有跨平臺的特性，它有Linux、Microsoft Windows、Mac OS X、BeOS、BSD、Pocket PC及Solaris的版本。而VLC依照作業系統的不同，支援的的是亦不相同，下面我們只針對Microsoft Windows作業系統，簡單的列出VLC所支援的格式。

✧ 輸入媒體影音資料：

UDP/RTP unicast、UDP/RTP multicast、HTTP/FTP、MMS、RTSP、File、DVDs、VCD、SVCD、Audio CD(without DTS)、MPEG encoder、video acquisition。

✧ 影音包裝格式：

MPEG(ES,PS,TS,PVA,MP3)、ID3 TAG、AVI、ASF/WMV/WMA、Raw DV、MP4/MOV/3GP、OGG/OGM/Annodex、Matroska(MKV)、WAV(incuding DTS)、Raw Audio：DTS, AAC, AC3/A52、FLAC、FLV(Flash)。

✧ 聲音解碼器：

MPEG Layer 1/2、MP3、AC3-A/52、DTS、LPCM、AAC、Vorbis、WMA 1/2、Alaw/ulaw、WMA 3、ADPCM、DV Audio、FLAC、QDM2/QDMC (QuickTime)、MACE、Speex。

✧ 影像解碼器：

MPEG-1/MPEG-2、DIVX(1/2/3)、MPEG-4、DivX 5、Xvid、3ivX D4、H.264、DV、Sorenson 1/3 (QuickTime)、Cinepak、Theora(alpha 3)、H.263/H.263i、MJPEG(A/B)、WMV 1/2、WMV 3/WMV-9/VC-1、Indeo Video v3。

2.4 VLC 原始碼目錄結構

VideoLAN 所開放的 VLC 原始碼是非常龐大，以下我們將介紹在這個原始碼內的目錄結構以及相關功用：

2.4.1 LibVLC library：

LibVLC library 是 VLC 的核心部分，提供了一個與 VLC program 溝通的介面和許多功能，例如串流的存取、audio 與 video 的輸出的操作、thread system 的管理。所有 LibVLC 的程式碼都是放在 VLC 原始碼資料夾內的 src/資料夾內。而在 src/資料夾內大致上又分成好幾個資料夾，底下將做個簡單的介紹：

- ✧ **interface/**：包含與使用者互動的程式碼，例如 key presses 和 device ejection。
- ✧ **playlist/**：管理 playlist 的互動，例如 stop，paly，next 或 random playback。
- ✧ **input/**：開啟一個 input module，讀取封包，解析並傳送重組的基本位元流(elementary streams)給 decoders。
- ✧ **video_output/**：初始化 video display，從 decoders 取得所有的 pictures 和 subpicture，而且可以選擇性的將它們轉換到其他 format(例如 YUV 轉 RGB)，並顯示出來。
- ✧ **audio_output/**：初始化 audio mixer，也就是找到正確的撥放頻率，然後重新取樣從 decoders 中所接收到的 audio frames。
- ✧ **misc/**：各式各樣的功用使用在 libvlc 其他部分，例如 thread system，message queue，CPU detection，object lookupc，platform-specific code。

2.4.2 模組(module)介紹

VLC 將許多功能模組化，可以因應不同功能的需求，VLC 會去尋找合適的模組來處理。例如針對不同來源的影音資料(如從電腦影音檔案或網路)，VLC 就會選擇合適的存取模組來讀取資料；針對不同的影音檔案格式，就可能需要不同的解多工器來解析影音資料，則 VLC 就會選擇合適的解多工器模組來進行解

多工程序；針對不同的影音編碼格式，就可能需要不同的解碼器來解碼，則 vlc 就會選擇合適的解碼模組來進行解碼。vlc 所有的模組均放在 vlc 原始碼資料夾內的 modules/資料夾內，而在這資料夾內，大致上可以分成好幾種不同功能類型的模組，底下我們將介紹幾種模組功能：

- **access/**：協定 vlc 可以存取不同來源的資料；例如從電腦檔案，由網路(http，tcp，udp 等等)或 vcd、dvd 的來源，來讀取資料。
- **codec/**：包含各式各樣的 codec modules。例如 ffmpeg，libmpeg2 等等。
- **video output/**：此資料夾內存放的是影像輸出模組。vlc 針對系統的不同，使用合適的影像輸出 module，顯示影像在螢幕上。
- **audio output/**：此資料夾內存放的是影像輸出模組。vlc 針對系統的不同，選擇適合聲音輸出的 module。
- **visualization/**：當 vlc 在撥放影音時，增加一個視覺化效果的視窗。
- **control/**：提供一些介面可以來控制 vlc 播放器，例如 telnet，http 等等。
- **gui/**：針對不同的作業系統平台(例如Linux、Microsoft Windows、Mac OS X、等等)，使用合適的interface。
- **demux/**：提供各種檔案格式的 demuxer。在影音串流中，video 和 audio 是以”container”的形式表示；一個” container”可以包含幾種不同的 codec 格式，demuxer 會從 container 中取出 stream，並將 stream 送到 decoder。
- **audio_filter/**：可以讓使用者將輸出聲音轉換成其他格式或一些特殊音效，例如立體/反立體聲，單聲道，古典音效，搖滾音效，電子音樂音效等等。
- **video_filter/**：可以讓使用者將輸出的影像作一些特殊的處理，例如用不同的交錯方式(如線性，混合，平均數等等)來交錯 video stream；也可以對影像作複製，失真，放大，旋轉影像等特殊處理。

2.5 VLC 主程式架構

在進入到第三章所要描述的播放程序之前，我們先簡單描述 VLC 主程式以及所呼叫的函數。一開始的進入點是在程式碼 vlc.c 內的 main()函數，而 main()函數內主要是呼叫了 VLC_Create()，VLC_Init()，VLC_AddIntf()，VLC_CleanUp()，VLC_Destroy()這 5 個函數(如下圖 2.2 所示)。以下我們只簡單的描述一下這五個函數的作用。

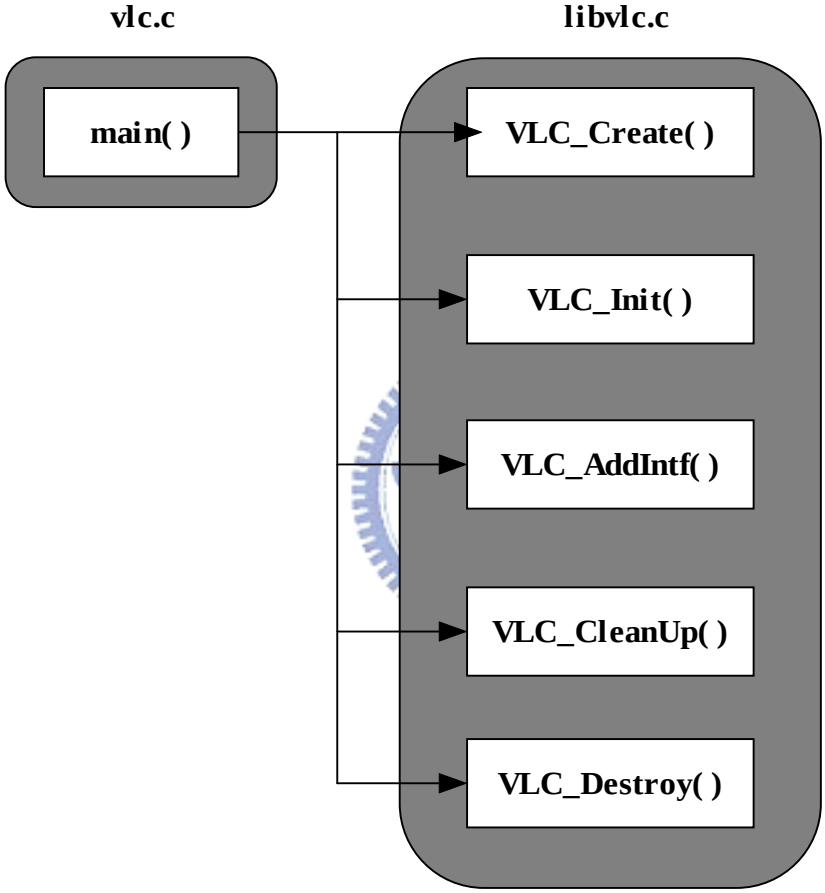

圖 2.2：VLC main()函數

- VLC_Create()函數：

在這個函數主要是先初始化 thread system，並初始化 vlc_t structure 和 libvlc structure 這兩個結構變數；若發生錯誤就回傳負值給 VLC_Create()函數。

➢ VLC_Init()函數：

主要是初始化之前 VLC_Create()函數內所 allocated vlc_t 結構變數，並實做下面所列的動作；且若發生錯誤一樣會回傳負值給 VLC_Init() 函數。

- CPU detection
- gettext initialization
- message queue, module bank and playlist initialization
- configuration and commandline parsing
- interface opening

➢ VLC_AddIntf()函數：

在 VLC_AddIntf()函數裡，主要呼叫 AddIntfItnternal()函數來選擇適當的 interface module 和開啟 interface，在這裡主要是選擇適當的 GUI interface module，並且 vlc 的 interface 又可分為 GUI interface 和 control interface，底下我們簡單的說明到底有哪些 interface。

GUI Interface：有四種主要的 graphical interface

1. wxWidget interface：在 windows 和 linux 作業系統中，預設是使用這種 interface。
2. skins2 interface：在 windows 和 linux 作業系統上提供另一種 interface 的選擇。
3. MacOSX interface：此 interface 只適合 MacOSX 作業系統。
4. BeOS interface：此 interface 只適合 BeOS 作業系統上。

Control Interface：主要是提供非 GUI interface 的模式，來操作 vlc。例如 hotkeys interface 是提供快速鍵的功能，可以利用鍵盤來控制 vlc，其他的 control interface 還有 web interface、telnet interface 等等，提供不同的方式來操作 vlc。

圖 2.3：GUI interface on windows

圖 2.4：skins2 interface on windows

- VLC_CleanUp()函數：

  清除所有 intf，playlist，vout，aout，threads 等等。

- VLC_Destroy()函數：

  消除 libvlc structure 結構變數等等。

上述所提的函數只是很簡單的說明 VLC 主程式所呼叫的函數及相關功用，當然每個函數所延伸的程式流程是非常大的。在 2.4 節我們也有提到過，整個 VLC 原始碼是非常龐大的，我們無法一一的詳盡描述每個函數作用以及了解每個程序作用，因此在第三章開始只針對我們要了解的程序，以及如何解決第一章所討論的問題來做個討論。

第三章 VLC 播放流程架構介紹

一個多媒體播放器的播放流程架構就如同圖 3.1 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，並利用合適的解多工器將存取的資料分離出 audio bitstream 及 video bitstream，然後再使用合適的解碼器分別對 audio bitstream 及 video bitstream 來解碼，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，而 VLC 的播放流程也是這樣的一個基本架構。

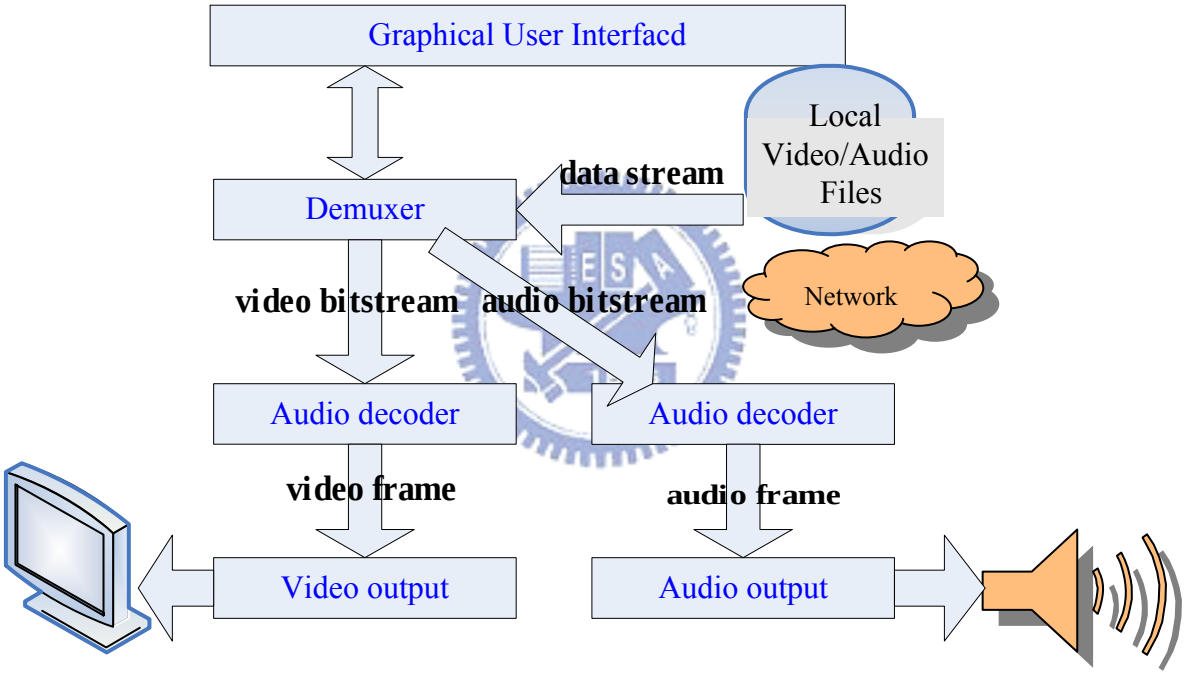

圖 3.1：播放流程基本架構

但是從這整個播放流程當中，我們想知道的是 VLC 是如何去解決在第一章所提到的問題，也就是如何找到並使用合適的解多工器以及解碼器，所以本章主要是描述 VLC 的播放流程架構，並從這流程中了解到 VLC 是如何去解決所面臨的問題。

3.1 VLC 運作機制

VLC 本身是一個 multi-thread 的架構，也就是依靠許多支 thread 的運作整個 VLC，如圖 3.2 所示，在產生 VLC GUI 介面之後，就會有 4 支 thread 已經在運作執行了，每支 thread 都有它所要處理的程序，而我們主要討論的播放流程就是從其中一支叫做 playlist thread 所開始的。

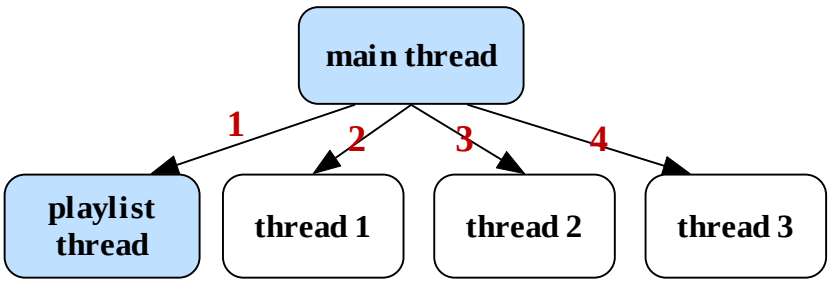

圖 3.2：產生 VLC GUI 後 thread 開啟的程序

如下圖 3.3，這是表示開始播放影音資料時，VLC 內 thread 開啟的程序，而圖中所標明的數字是代表開啟 thread 的先後順序，在說明整個播放程序之前，我們先簡單的說明這些 thread 在整個 VLC 運作上的作用。

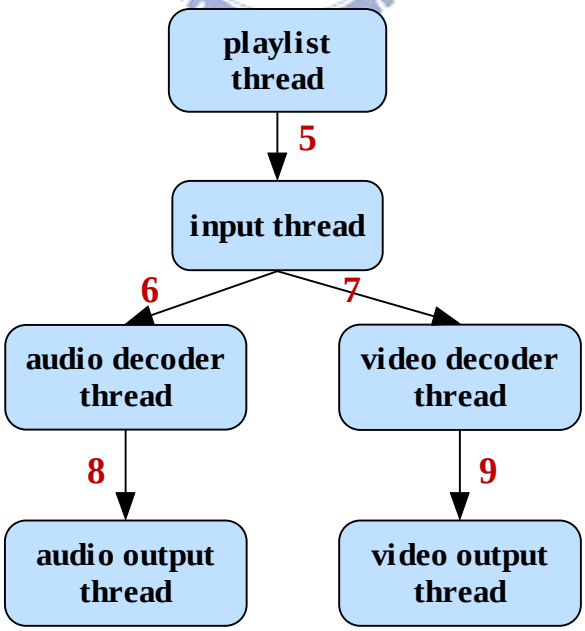

圖 3.3：播放影音時 thread 開啟的程序

✧ **main thread**：這是 VLC 的主程式所在，主要是初始化一些結構變數及其他初始化的設定，並且開啟 playlist thread 與其他 3 支 thread 來開始運作 VLC。

✧ **playlist thread**：主要是檢查目前播放的狀態，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喇吧。

✧ **video output thread**：負責將解碼後的影像輸出到螢幕上。

而 VLC 啟動 playlist thread 的程序，其實主要是由 VLC_Init()函數所呼叫的 playlist_Create()函數來啟動的，所以接下來我們主要就是以這支 playlist thread 作為開端來進行討論。

3.2 檢查目前播放狀態

playlist thread 主要作用是檢查目前播放的狀態，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；而這支 thread 的運作流程就是圖 3.4 虛框所示的 RunThread()函數。

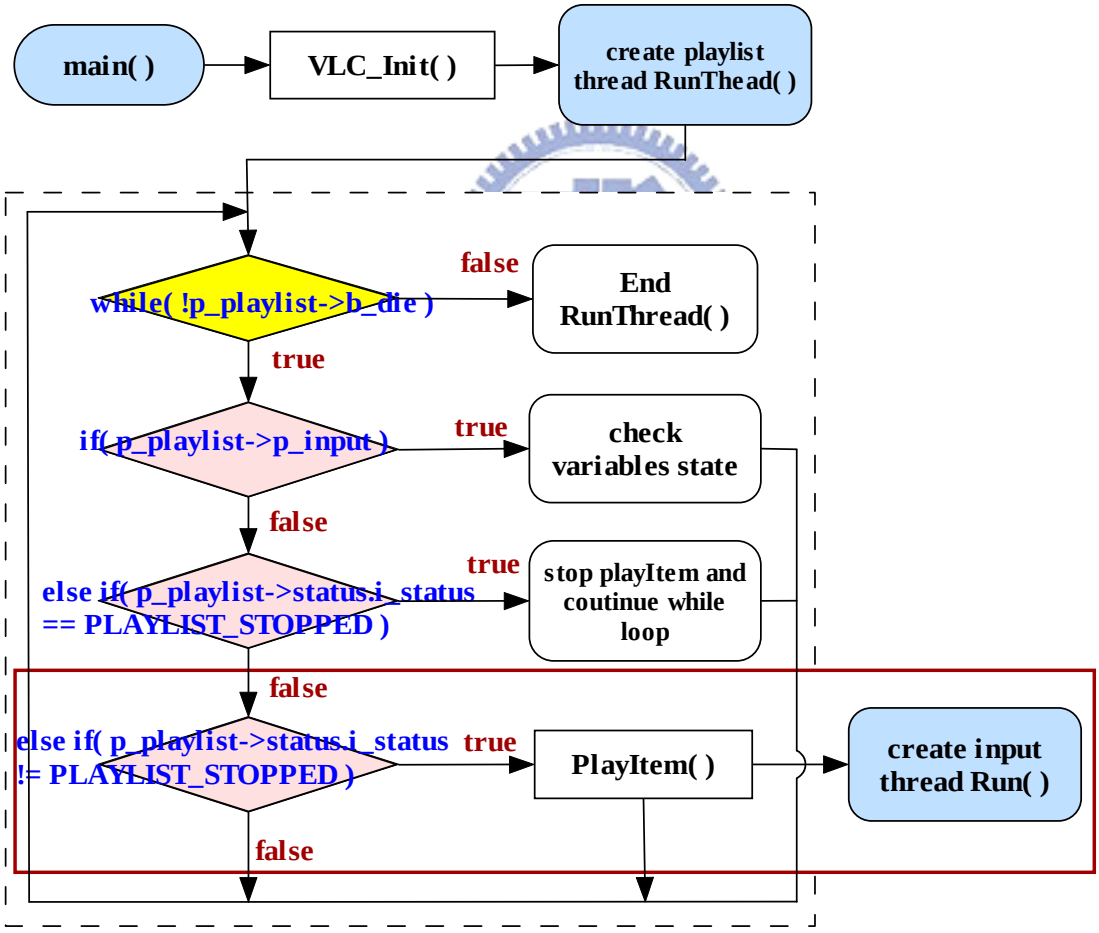

圖 3.4：RunThread()函數流程圖

如上圖 3.4 所示，在進入到 while 的判斷條件 p_playlist->b_die，這個變數是表示是否要結束整個 RunThread()函數，而一開始 p_playlist->b_die 的狀態是設定

為 false，表 VLC 示要執行 RunThread()函數的程序，也就是檢查目前播放的狀態；而當我們要結束掉整個 VLC 程式，也就是按下 VLC GUI 介面右上方的關閉圖示「X」時，p_playlist->b_die 的狀態就會被設定為 true，表示不再執行此 while 迴圈並結束掉 RunThread()函數的程序。

在進到這個 while 迴圈之後，可以看到有三圈的 if-else 判斷條件，主要是判斷 p_playlist->p_input 和 p_playlist->status.i_status 這兩個變數，來決定要執行什麼樣的程序。一開始產生出 VLC GUI 介面時，並沒有影音資料需要播放，因此 p_playlist->p_input 的狀態是被設定為 false，p_playlist->status.i_status 的狀態是被設定為「PLAYLIST_STOPPED」，所以第二圈的條件會先成立，並且會不斷的執行第二圈程序。而在第二圈所要執行的程序，主要是停止目前的播放影音程序，直到 p_playlist->p_input 或 p_playlist->status.i_status 這兩個變數狀態被改變了，才會進到其他圈的程序，所以整個 VLC 開始運作時，會一直不斷的進行第二圈程序。

而當在我們選擇開啟的資料來源(例如從檔案、網路、DVD/VCD 等等)並按下 OK 鈕之後，p_playlist->status.i_status 的狀態就會被改變成「PLAYLIST_RUNNING」(如下圖 3.6)，使得上圖 3.4 紅框內條件成立，也就是 VLC 會進行到第三圈的程序，並且會呼叫 PlayItem()函數來開啟 input thread 來進行整個播放影音的程序，並由 input thread 開啟後面的其他 thread。而開啟完 input thread 後，會回傳一個結構變數指標給 p_playlist->p_input，也就是 p_playlist->p_input 狀態被設為 true，表示已經成功的開啟 input thread 並開始進行播放影音程序。因此回到 RunThread()函數內的 while 迴圈裡，就變成不斷的進行到第一圈的程序，而第一圈的程序主要是檢查某些變數狀態設定是否被改變。因此 VLC 正在播放影音資料時，在 RunThread()函數內的 while 迴圈內就是會一直進到第一圈程序的狀態。

而當 VLC 在播放影音資料時，在前面 3.1 節有提到過，會有其他的 thread 已經被開啟並執行播放影音的程序，所以假設在播放影音資料我們按下停止鍵，p_playlist->p_input 變數會被改變成 false，p_playlist->status.i_status 變數會被改變成「PLAYLIST_STOPPED」，代表要停止目前的播放影音程序。因此原本一直進到第一圈的程序的狀態，會變成進到第二圈的程序，並且停止目前的播放影音程序，所以有關跟播放影音程序時所開啟的 thread 也都會結束掉，也就是前面圖 3.3 所開啟的五支 thread 都會結束。因此整個狀態 thread 架構又是回復到前面圖 3.2 所表示的一開始的五支 thread 在執行。而此時在 RunThread()函數內的 while 迴圈內會變成一直進到第二圈程序的狀態，直到有影音資料需要播放，然後再進到第三圈程序開啟 input thread 並且 p_playlist->p_input 狀態被設為 true，之後又會不斷的一直進到第一圈程序來檢查某些變數狀態設定是否被改變。

所以整個 VLC 播放影音程序的一個控制核心就是在 RunThread()函數內的 while 迴圈，在迴圈內主要是判斷 p_playlist->p_input 和 p_playlist->status.i_status 這兩個變數，來決定要執行哪一圈的程序；一開始產生出 VLC GUI 介面時，會一直進到第二圈的程序，表示目前的播放狀態是停止的；等到按下播放鍵要播放影音資料之後，會先進到第三圈的程序，開啟 input thread 並且 p_playlist->p_input 狀態會被設為 true；最後會進到第一圈的程序檢查一些變數的狀態；直到按下停止鍵時，才又會進入到第二圈的程序。

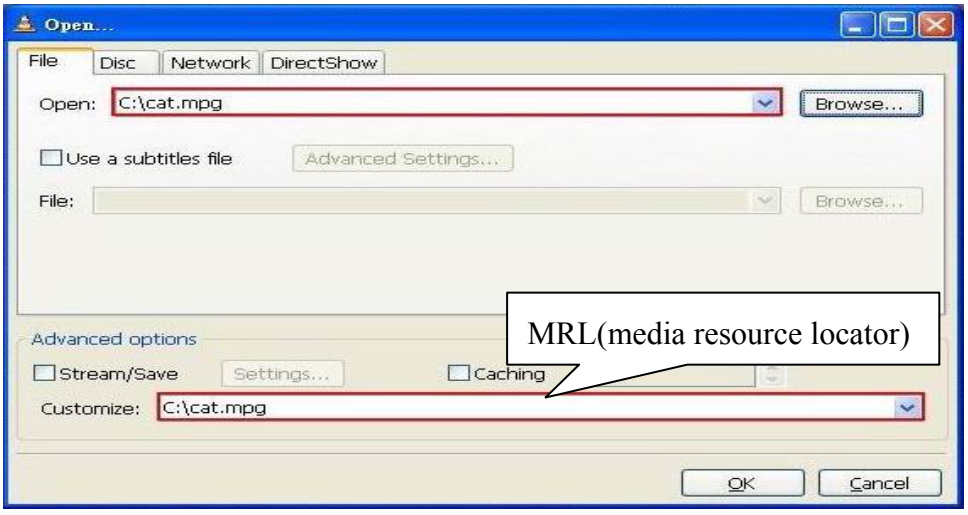

圖 3.5：選擇開啟檔案資料來源

圖 3.6：改變 p_playlist->status.i_status

3.3 播放影音程序

在 PlayItem()函數中，主要是開啟 input thread 來開始整個影音播放程序，而這支 input thread 的主要運作流程就是圖 3.7 虛框所示的 Run()函數。我們可以看到 Run()函數裡主要呼叫了 vlc_thread_ready()函數、Init()函數、MainLoop()函數這 3 個函數，而接下來我們只針對 Init()函數及 MainLoop()函數內部分相關的函數流程來討論。

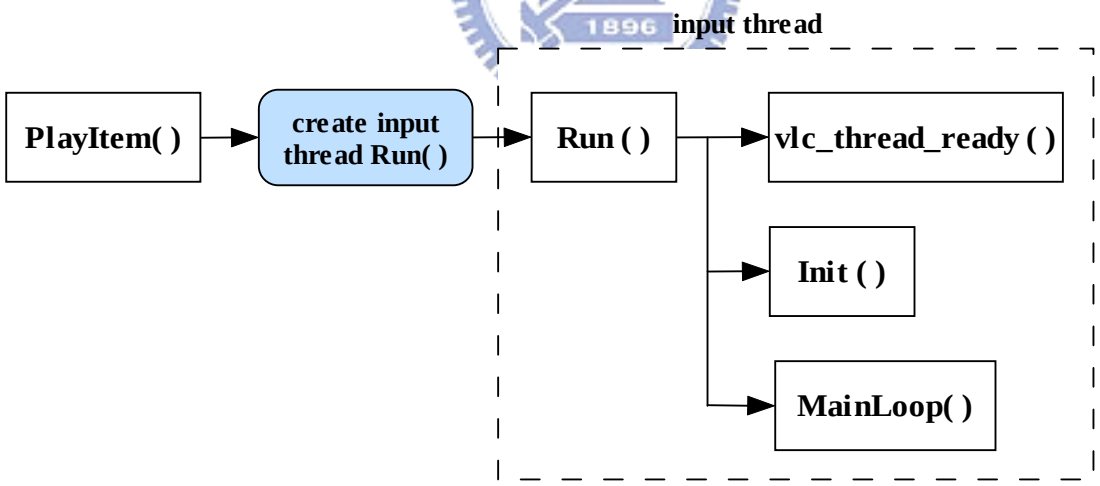

圖 3.7：input thread Run()函數

在本章的一開始我們已經簡單的描述過 VLC 播放流程架構:在我們選擇開啟的資料來源之後，首先 VLC 所要進行的程序是從資料來源去讀取影音資料，並利用解多工器將影音資料分離出 audio bitstream 及 video bitstream。

但是在進行這個程序之前，VLC 必須要針對存取的資料來源找到合適的存取模組，以及針對影音包裝格式找到合適解多工器模組，之後就會利用此存取模組來讀取資料，並且利用此解多工器模組來解析影音資料。而要尋找合適的存取模組以及解多工器模組的部分，就是在 Init()函數內所呼叫的 InputSourceInit()函數來完成的。

下面圖 3.8 就是一個 InputSourceInit()函數的基本程式流程，這個函數的主要的作用是在於針對不同的資料來源，例如從電腦檔案、網路、DVD/VCD 等等，要先找到合適的存取模組來存取影音資料；並且針對不同的影音包裝格式，例如 MPEG-2 program stream、MPEG-2 transport stream 等等，要找到合適解多工器模組來解析影音資料，所以接下來我們就要來說明 VLC 如何找到並使用合適的存取模組以及解多工器模組。

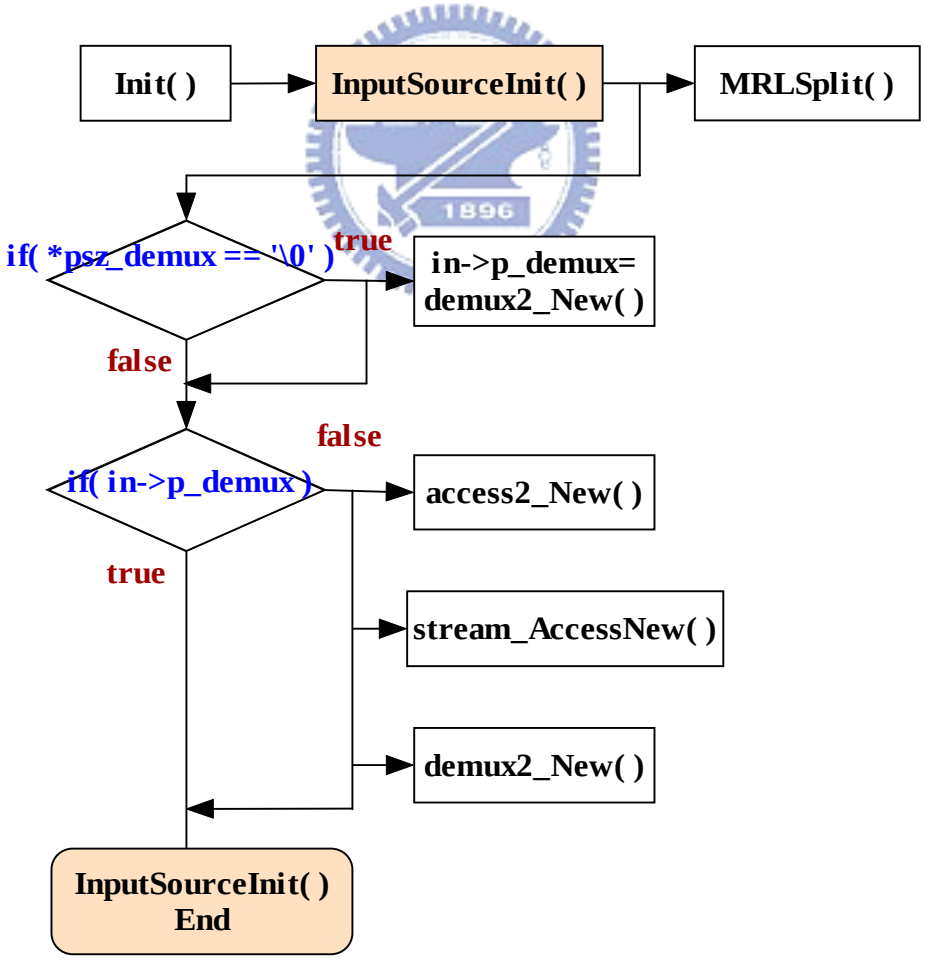

圖 3.8：InputSourceInit()函數程式流程

3.3.1 分解 MRL(Media Resource Locator)

VLC 定義了所謂的 Media Resource Locator(簡稱 MRL)，也就是我們所選擇的影音資料來源 (如前面圖 3.5 所示)，例如 C:\test.mpg 就是一個 MRL，而 MRL 的架構為 access/demux://path。所以在 InputSourceInit()函數內會呼叫 MRLSplit()函數先分解 MRL 的 access、demux、path 這 3 個部分，並分別儲存在 psz_access、psz_demux、psz_path 這 3 個字串指標(如圖 3.9)，而分解 MRL 的主要目的，就是要利用所分解出來的字串來尋找合適的存取模組。

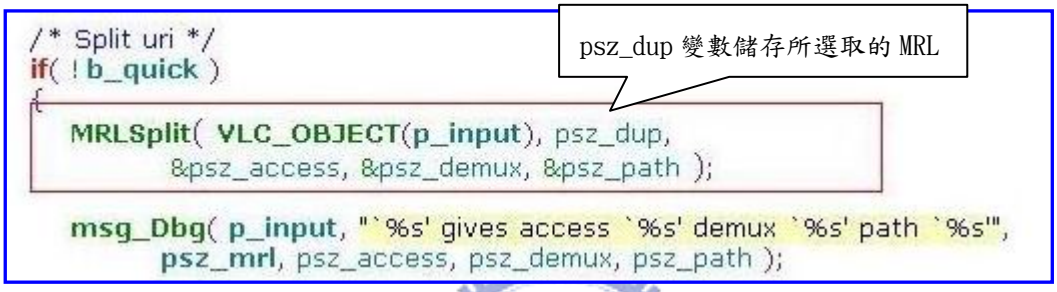

圖 3.9：呼叫 MRLSplit()函數分解 MRL

例如：若 MRL 為 C:\test.mpg，則會被拆解成

➔psz_access=””，psz_demux=””，psz_path=C:\test.mpg

若 MRL 為 rtsp://140.113.13.84/test.mpg，則會被拆解成

➔psz_access=rtsp，psz_demux=””，psz_path=140.113.1.84/test.mpg

若 MRL 為 http://140.113.13.84:1234，則會被拆解成

➔psz_access=http，psz_demux=””，psz_path=140.113.1.84:1234

若 MRL 為 udp://@224.0.0.1:1235，則會被拆解成

➔psz_access=udp，psz_demux=””，psz_path=@224.1.2.3:1235

3.3.2 尋找合適的存取模組

在分解完 MRL 之後，就會先判斷 psz_demux 是否為空字串，一般情況下，我們所選擇的 MRL 是沒有 demux 這個部分，因此 psz_demux 大部分都是為空字

串，所以程式流程會進到 demux2_New()函數內，並呼叫 module_Need()函數來尋找是否有合適的 access_demux module(下圖 3.10)來存取我們所選擇的資料來源及解多工影音資料，其中 psz_module 參數就是前面所分解出來的 psz_access。例如當我們選擇 rtsp 串流連線方式時，就會找到 live555 module 來存取串流資料。假使成功在 demux2()函數內找到合適的 access_demux module，那麼就會回傳一個指標位址給 in->p_demux 結構指標並結束 InputsourceInit()函數。

```
p_demux->p_module =
  module_Need( p_demux, "access_demux", psz_module,
               !strcmp( psz_module, p_demux->psz_access ) ?
               VLC_TRUE : VLC_FALSE );
}
```

圖 3.10：呼叫 module_Need()尋找合適的 access_demux module

如果找不到合適的 access_demux module，那麼就會回傳 NULL 給 in->p_demux 結構指標並進到 access2_New()函數內，然後呼叫 module_Need()函數來尋找是否有合適的 access2 module 來存取我們所選擇的資料來源(如下圖 3.11)，其中 psz_module 參數一樣是前面所分解出來的 psz_access。例如當我們選擇開啟電腦的影音資料時，就會在這裡找到 file module 來讀取檔電腦案資料，而假使在這裡還是找不到合適的 module 來存取資料，那就表示我們所選擇的資料來源是有問題，VLC 也就無法從這個來源存取資料，當然後面的程序也就無法完成了。

```
p_access->p_module =
  module_Need( p_access, "access2", p_access->psz_access,
               b_quick ? VLC_TRUE : VLC_FALSE );
}
```

圖 3.11：呼叫 module_Need()尋找合適的 access2 module

而在找到合適的 access2 module 來存取影音資料後，接下來所呼叫 stream_AccessNew()函數，主要是初始化一些結構變數以及開始先讀取一部分影音資料暫存到 buffer 內，而讀取的方式是利用讀取函數指標來讀取資料，而這個

讀取函數指標就是指向前面所找到的合適的 access2 module 內的讀取函數。因為在找到合適的 access2 module 的同時，就會指定此 access2 module 內的讀取函數給圖 3.13 紅框內的函數指標。

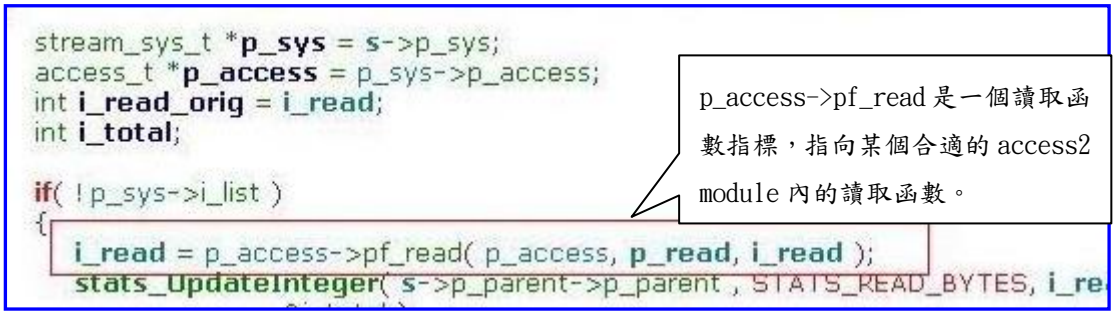

圖 3.12：讀取影音資料

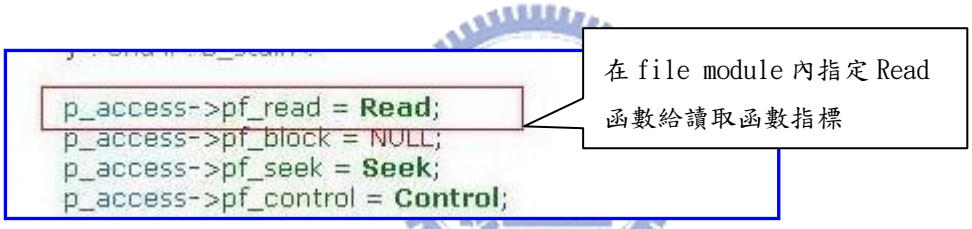

圖 3.13：file module 內所指定的讀取函數

所以當尋找合適的解多工器的時候，就會讀取 buffer 內的資料先確認影音包裝格式是什麼，例如 MPEG-2 program stream、MPEG-2 transport stream 等等。

3.3.3 尋找合適的解多工器模組

那麼 VLC 是如何確認來源資料影音包裝格式呢?在經過前面所描述的程序之後，接下來會在進到 demux2_New()函數內，並呼叫 module_Need()函數來尋找是否有合適的 demux2 module 可以解多工所讀取的影音資料(如下圖 3.14)，其中 psz_module 參數是前面所分解出來的 psz_demux。

```
    p_demux->p_module =
        module_Need( p_demux, "demux2", psz_module,
                     !strcmp( psz_module, p_demux->psz_demux ) ?
                     VLC_TRUE : VLC_FALSE );
}
```

圖 3.14：呼叫 module_Need()尋找合適的 demux2 module

舉例來說，假如影音資料的包裝格式為 MPEG-2 program stream，那麼在這裡就會找到 ps module 來解多工所存取的影音資料；如果影音包裝格式為 MPEG-2 transport stream，那麼就會找到 ts module 來解多工所存取的影音資料。所以 VLC 是利用 module_Need()函數來找到合適的解多工器模組，而如何利用此函數在這些解多工器模組當中找到合適的模組，在第四章我們會舉例來說明。

3.3.4 進行解多工程序

在經過前面 3.3 節所描述的 InputSourceInit()函數內運作流程及重要性之後，接下來我們要描述的是解多工的程序。首先我們直接來看 Run()函數內所呼叫的 MainLoop()函數。在 MainLoop()函數裡主要是利用了 while 迴圈不斷的呼叫解多工函數(如下圖 3.15)，利用此解多工函數將所讀取的影音資料分離出 audio bitstream 及 video bitstream，而此解多工函數其實就是前面所找到合適的 access_demux module 或者是 demux2 module 內所指定的解多工函數。

```
/*****************************************************************************
 * Main loop: Fill buffers from access, and demux
 *****************************************************************************/
static void MainLoop( input_thread_t *p_input )
{
    int64_t i_intf_update = 0;
    while( !p_input->b_die && !p_input->b_error && !p_input->input.b_eof )
    {
        vlc_bool_t b_force_update = VLC_FALSE;
        int i_ret;
        int i_type;
        vlc_value_t val;

        /* Do the read */
        if( p_input->i_state != PAUSE_S )
        {
            if( p_input->i_stop <= 0 || p_input->i_time < p_input->i_stop )
            {
                i_ret=p_input->input.p_demux->pf_demux(p_input->input.p_demux);
            }
```

p_input->input.p_demux->pf_demux 是一個解多工函數指標，指向某個合適的 demux2 module 或 access_demux 內的解多工函數。

圖 3.15：MainLoop()函數部分程式碼

舉例來說，假如影音資料的包裝格式為 MPEG-2 program stream，那麼在前面確認來源資料影音包裝格式的時候，就會找到 ps module 來解多工所存取的影音資料，而找到 ps module 的同時，就會指定此 module 內的解多工函數給的解多工函數指標(如圖 3.16)，所以在 while 迴圈裡所呼叫的解多工函數，就等同於呼叫了此 ps module 內的解多工函數來進行解多工動作。

圖 3.16：ps module 內所指定的解多工函數

3.3.5 尋找合適的解碼器模組

在前面討論完解多工的程序之後，現在我們要討論的是如何進行到解碼的程序，而在進入到解碼程序之前，要先尋找到合適的解碼模組。下面圖 3.17 是在說明開啟不同的影音資料來源，尋找合適的解碼器模組的時機會不太一樣。例如虛框的部分是表示如果是開啟 rtsp 串流或開啟 dvd 光碟時，是在 Init()函數內去尋找合適的 audio 和 video decoder module 和開啟 audio 和 video decoder thread；若是開啟電腦檔案或 http 等連線方式時，會在 demuxer function 內去尋找合適的 audio 和 video decoder module 和開啟 audio 和 video decoder thread。

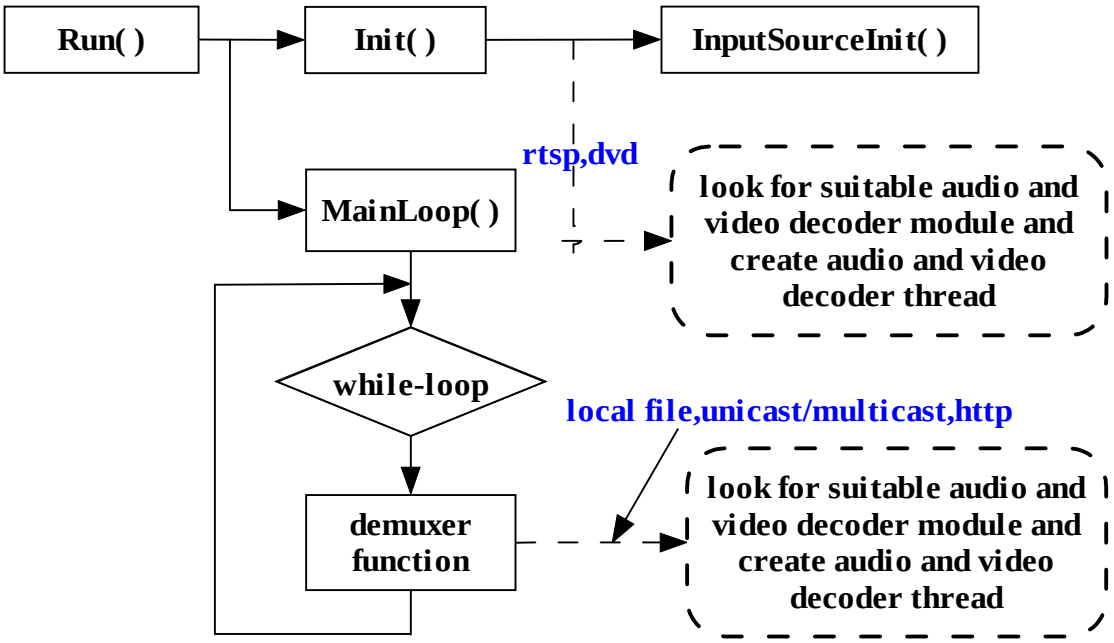

圖 3.17：尋找合適的 decoder module 程序

圖 3.17 中的 demuxer function 就是前面圖 3.16 紅框所表示的解多工函數。在前面有提過，解多工函數主要的作用就是將讀取的影音資料分解成 video bistream 和 audio bistream，然後再分別由 video decoder 及 audio decoder 來進行解碼，但是 VLC 是如何知道 audio 及 video 的編碼格式，以及尋找並使用合適的解碼器來進行解碼呢?

在進行解多工的程序當中，一般都會去解析來源資料的影音包裝格式，最後分解出 video bistream 和 audio bistream。例如 mpeg-2 program stream 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)而形成的(如下圖 3.18 所示)，每一個 PES 所代表的就是一個 audio PES 或者 video PES 等其他類型 PES。所以解多工函數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 就是前面所說的 video bistream 或 audio bistream，最後這些 bistream 就會由解碼器來解碼。

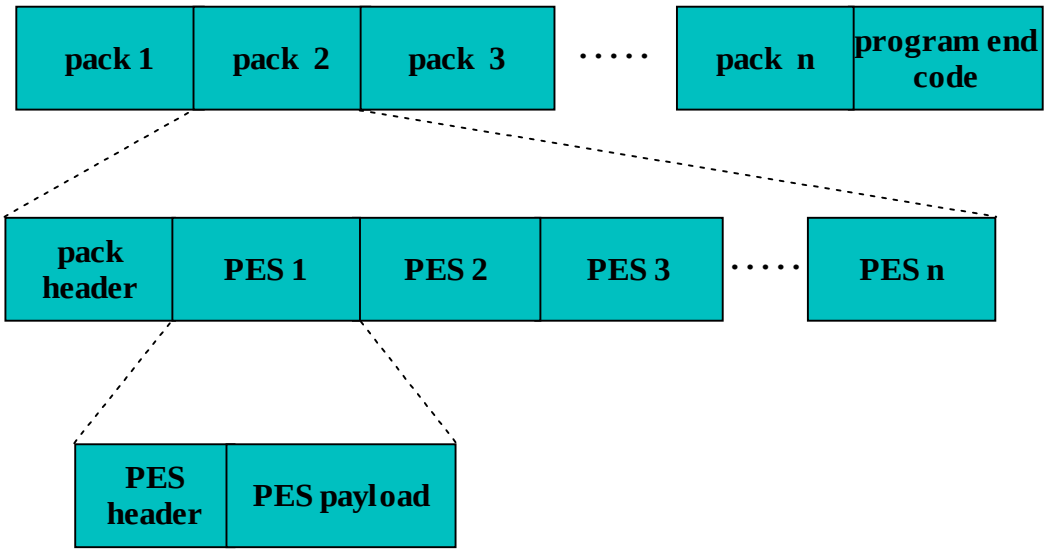

圖 3.18：mpeg-2 program stream

所以 VLC 就是在解析來源資料的影音包裝格式的時候，會找出 audio 與 video 的編碼格式，然後將 audio 與 video 的編碼格式分別以四字元碼型式來表示，並且會呼叫 module_Need()函數來尋找合適的 audio decoder module 和 video decoder module，而如何找到合適的 decoder module 就是跟四字元碼有關。所以有關 audio 與 video 四字元碼型式為何以及如何找到合適的 decoder module 這部分在第四章則會再詳細的說明。

3.4 進行聲音及影像解碼程序

下面圖 3.19 就是前面圖 3.17 虛框內所說明的程式流程。在取得 audio 與 video codec 的四字元碼之後，一樣是先呼叫 module_Need()函數來尋找合適的 audio decoder module 或 video decoder module(如下圖 3.20)，然後開啟 DecoderThread()這支 decoder thread，在這支 thread 當中，利用 while 迴圈不斷的呼叫 block_FifoGet()函數來取得 audio bistream 或 video bitstream，最後由 DecoderDecoder()函數呼叫影像解碼函數或聲音解碼函數來對 audio bistream 或 video bitstream 進行解碼動作。

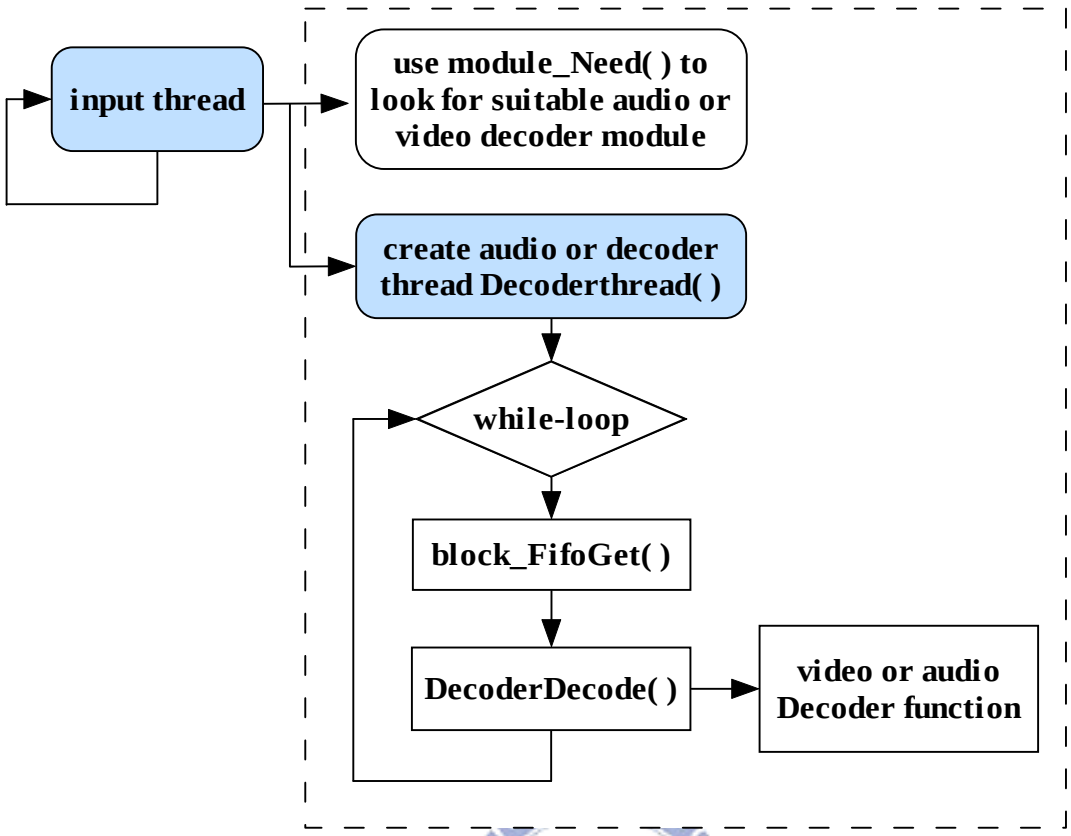

圖 3.19：解碼流程

```
/* Find a suitable decoder/packetizer module */
if( i_object_type == VLC_OBJECT_DECODER )
    p_dec->p_module = module_Need( p_dec, "decoder", "$codec", 0 );
else
```

圖 3.20：呼叫 module_Need()尋找合適的 decoder module

其中在 DecoderThread()函數內所呼叫的解碼函數，就是先前所找到合適的 decoder module 內所指定的解碼函數。舉例來說，假如影音資料的 video 編碼格式為 MPEG2，那麼在前面確認影音資料內的 video 編碼格式的時候，會找到 libmpeg2 module 這個 decoder module 來對 video bistream 進行解碼動作；而在找到合適的 libmpeg2 module 的同時，就會指定此 module 內的影像解碼函數給影像解碼函數指標(如下圖 3.21)，因此在 Decoderthread()函數內所呼叫的影像解碼函數，就等同於呼叫了此 libmpeg2 module 內的影像解碼函數來進行解碼動作(如下圖 3.22)。

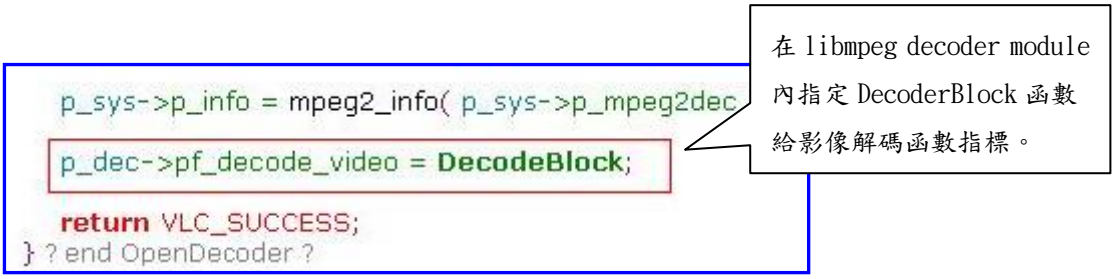

圖 3.21：libmpeg2 module 內所指定的影像解碼函數

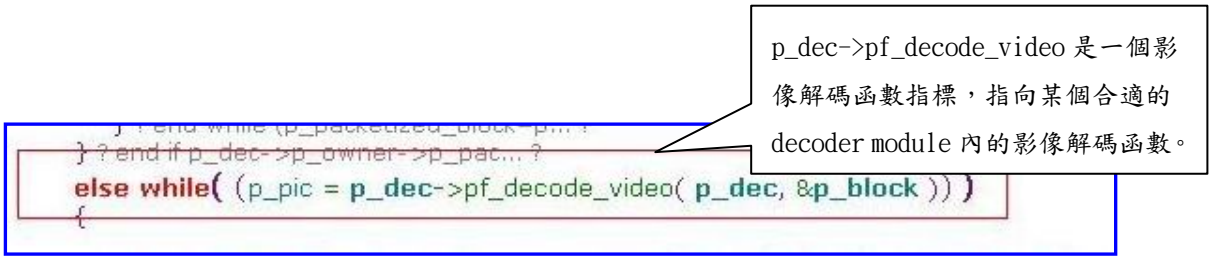

圖 3.22：DecoderDecode()內部分程式碼

同樣地，假如影音資料的 audio 編碼格式為 MP3，那麼確認影音資料內的 audio 編碼格式的時候，會找到 mpeg_audio module 這個 decoder module 來對 audio bistream 進行解碼動作；而在找到合適的 mpeg_audio module 的同時，也會指定此 module 內的聲音解碼函數給聲音解碼函數指標(如下圖 3.23)，因此在 DecoderDecode()函數內所呼叫的聲音解碼函數，就等同於呼叫了此 libmpeg2 module 內的聲音解碼函數來進行解碼動作。

圖 3.23：mpeg_audio module 內所指定的聲音解碼函數

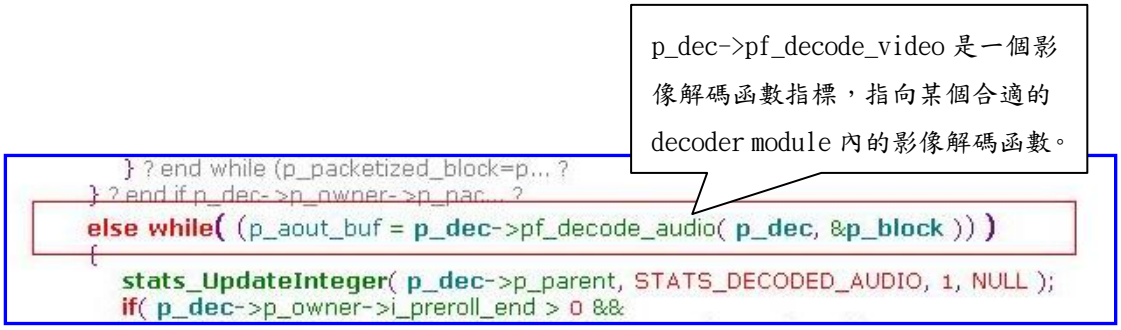

圖 3.24：DecoderDecode()內部分程式碼

第四章 尋找合適模組機制

在經過第三章介紹完整個 VLC 播放流程之後，我們會發現到 VLC 在執行某個程序之前，會利用 module_Need()函數去尋找適合此程序的模組，然後在利用此模組所指定的函數來執行這個程序。例如在進行解多工程序之前，就必須要先找到合適的解多工器模組，然後使用此模組的解多工函數來進行解多工的程序；在進行解碼程序之前，就必須要先找到合適的解碼模組，然後使用此模組的解碼函數來進行解碼的程序。問題是 VLC 如何找到合適的模組？在 VLC 這個 open source 裡面總共有 2 百多個模組，每個模組的功能性質有些相同有些不同，而要如何從這 2 百多個模組當中，找到符合 VLC 所需要合適的功能模組？所以在本章我們要去分析 module_Need()函數的運作機制，以及說明如何利用此函數，找到合適的解多工器模組以及解碼器模組。

4.1 何謂模組(module)

我們在第二章有稍微提到過，VLC 將許多功能模組化，可以因應不同功能的需求，VLC 會去尋找合適的模組來處理，並且 VLC 所有的模組均放在 VLC 原始碼資料夾內的 modules/資料夾內，而在這資料夾內，大致上可以分成好幾種不同功能類型的模組。但是在 VLC 當中是如何定義一個模組呢？我們實際以一個解碼模組為例子來說明。

如下圖 4.1 所示，這是在 libmpeg2.c 這支程式內的部分程式碼，它將 libmpeg2.c 此程式定義成一個 libmpeg2 decoder module，在 VLC 所有的模組當中，都定義了類似圖 4.1 的這樣程式碼。

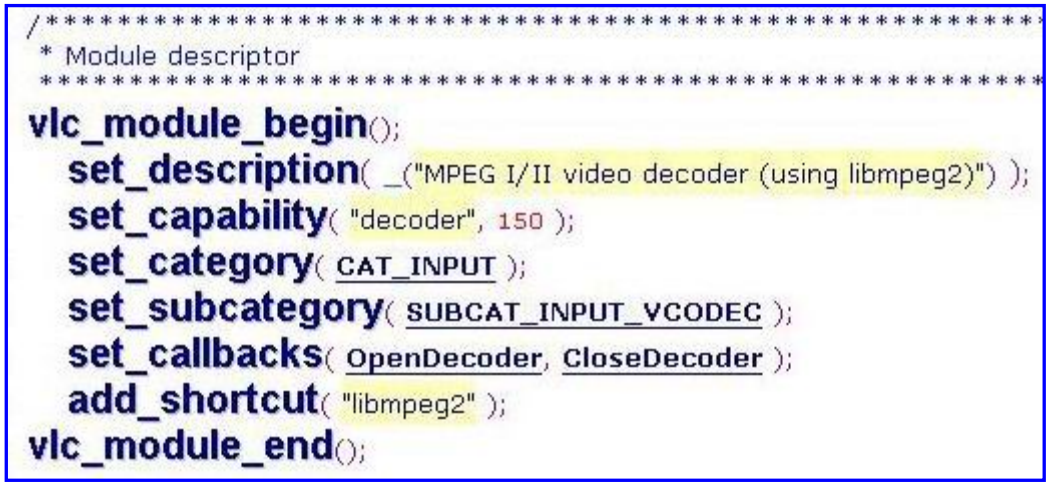

圖 4.1：libmpeg2 decoder module 定義

所以上述所提及的，「VLC 會依據所需要的功能，尋找合適的 module 來處理」，我們可以當成 VLC 其實是呼叫合適的程式(program)來實作這些功能。換句話說，「module」就是代表著某一支程式(program)。就以上面的為例子，當需要一個針對 mpeg2 編碼格式的解碼器時，就會進到 libmpeg2.c 這支程式(program)，並利用此程式內所定義的解碼函數來解碼，因此我們就把這支程式當做是一個 module。而通常定義一個模組必須包含 vlc_module_begin()和 vlc_module_end()這兩個巨集(macro)，巨集之間所代表的是對這個模組的設定，例如 set_capability("decoder", 150)表示設定此模組的功能為解碼器模組，能力值(score)為 150。

所以對於 VLC 模組的功能分類，就是利用 set_capability()來設定模組的功能，而設定模組功能的方式，就要在 set_capability()內設定特定的字串，藉此來設定模組的功能為何，也就是說對於不同的功能模組，所設定的字串也就會不同，因此以下我們整理出 VLC 中每個模組在 set_capability()內所設定的特定字串："access2"、"access_demux"、"access_filter"、"audio filter"、"audio filter2"、"audio mixer"、"audio output"、"chroma"、"crop padd"、"decoder"、"demux2"、"dialogs provider"、"encoder"、"gui-helper"、"id3"、"interface"、"memcpy"、"meta reader"、"mpeg-system"、"network"、"opengl provider"、"packetizer"、"playlist export"、

"services_discovery"、"sout access"、"sout mux"、"sout stream"、"sub filter"、"text"、"renderer"、"tls"、"video blending"、"video filter"、"video filter2"、"video output"、"xml"。

比方說如果要將某個模組分類成一個解碼器模組，那麼在 set_capability()內設定的字串就必須為"decoder"；如果要將某個模組分類成一個解多工器模組，那麼在 set_capability()內設定的字串就必須為"demux2"。因此藉由這個設定可以依照功能的不同來分類這些模組。所以將這些模組依功能的不同來分類的目的，就是為了當 VLC 需要某種功能的模組時，主要就是利用這些所設定字串，先從 2 百多個模組當中搜尋出符合此功能的模組們，之後再從這些符合功能的模組當中找出一個合適的模組。所以在後面我們就會說明如何藉由這樣子的設定，使得 VLC 呼叫 module_Need()時可以尋找出合適的模組。

4.2 module_Need()函數運作分析

之前有提到過，VLC 會因應不同功能的需求，會呼叫 module_Need()函數去尋找合適的 module 來處理，所以我們要先簡單說明 module_Need()函數的參數意義。如下圖 4.2 所示，module_Need()函數參數列裡有四個變數，依照所要尋找的模組不同，傳入的變數值當然也會不同，如果傳入的變數值不符合 VLC 所規定的，那麼就會發生找不到模組的錯誤情況發生，因此接下來我們先稍微說明一下這些傳入的變數值有什麼樣的規定。

```
/*****************************************************************************
 * module_Need: return the best module function, given a capability list.
 *****************************************************************************
 * This function returns the module that best fits the asked capabilities.
 *****************************************************************************/
module_t * __module_Need( vlc_object_t *p_this, const char *psz_capability,
                          const char *psz_name, vlc_bool_t b_strict )
{
```

圖 4.2：module_Need()函數定義

4.2.1 傳入變數說明

首先我們先來說明 p_this、psz_name 和 b_strict 這三個參數。p_this 是一個結構指標變數，主要是用來建立一個模組的鏈結串列；psz_name 是一個字串變數，而 psz_name 的字串型態有 NULL、0、"$name"和"name"這四種字串型態(這邊所指的 name 指的是字串統稱，並不是傳入 name 字串)，至於什麼時候要傳入什麼樣的型態，就要看尋找什麼樣的合適的模組，例如在尋找合適的解碼器模組時，所傳入的 psz_name 變數為$codec；b_strict 則為一個 0 或 1 的整數變數，至於什麼時候要傳入什麼樣的整數型態，一樣要看需要什麼樣的合適的模組才能決定。

而最重要的則是 psz_capability 字串變數。psz_capability 字串型態有很多種，對於需要不同的功能模組，那麼就要傳入特定的字串。以下我們整理出在整個 VLC source code 內，呼叫 module_Need()函數時所傳入 psz_capability 的字串(依字母順序排列)：

"access2"、"access_demux"、"access_filter"、"audio filter"、"audio filter2"、"audio mixer"、"audio output"、"chroma"、"crop padd"、"decoder"、"demux2"、"dialogs provider"、"encoder"、"gui-helper"、"id3"、"interface"、"memcpy"、"meta reader"、"mpeg-system"、"network"、"opengl provider"、"packetizer"、"playlist export"、"services_discovery"、"sout access"、"sout mux"、"sout stream"、"sub filter"、"text renderer"、"tls"、"video blending"、"video filter"、"video filter2"、"video output"、"xml"。

舉例來說，假如現在要尋找一個合適的解多工器模組，則傳入 psz_capability 的字串必須要為"demux2"；若要尋找一個合適的解碼器模組，則傳入 psz_capability 的字串必須要為"decoder"。所以當 VLC 要尋找一個合適的功能模組時，一定要傳入特定的 psz_capability 字串，否則就會發生有合適的功能模組但卻無法找到的錯誤情況發生。而整個 module_Need()函數運作過程大致上可以分為兩個階段：第一階段的主要工作，就是從 VLC 內所有的 modules 中，先找出符合功能需求的 modules，例如現在需要一個合適的解多工器模組或解碼器模

組，那麼就先將所有的解多工器模組篩選出來或是解碼器模組篩選出來，而這些 module 被稱作為 candidate module；第二階段就是從這些 candidate modules 當中，選擇第一個合適的 modules。所以接下來的 4.2.2 以及 4.2.3 小節，就是要分別說明 module_Need()函數運作的這兩個階段。

4.2.2 第一階段--找出 candidate modules

第一階段的主要目的就是從 VLC 內所有的 module 中，先找出符合功能需求的 candidate module，如下圖 4.3 所示，整個第一階段的運作流程，大致上我們它分成四個部分，其中第四部份是最為重要的部分，因為在這邊才是真正進行尋找符合功能需求的 candidate module 的部分。而經過這四個部分的運作處理，就可以先找出 candidate module 並進行到第二階段的部分。

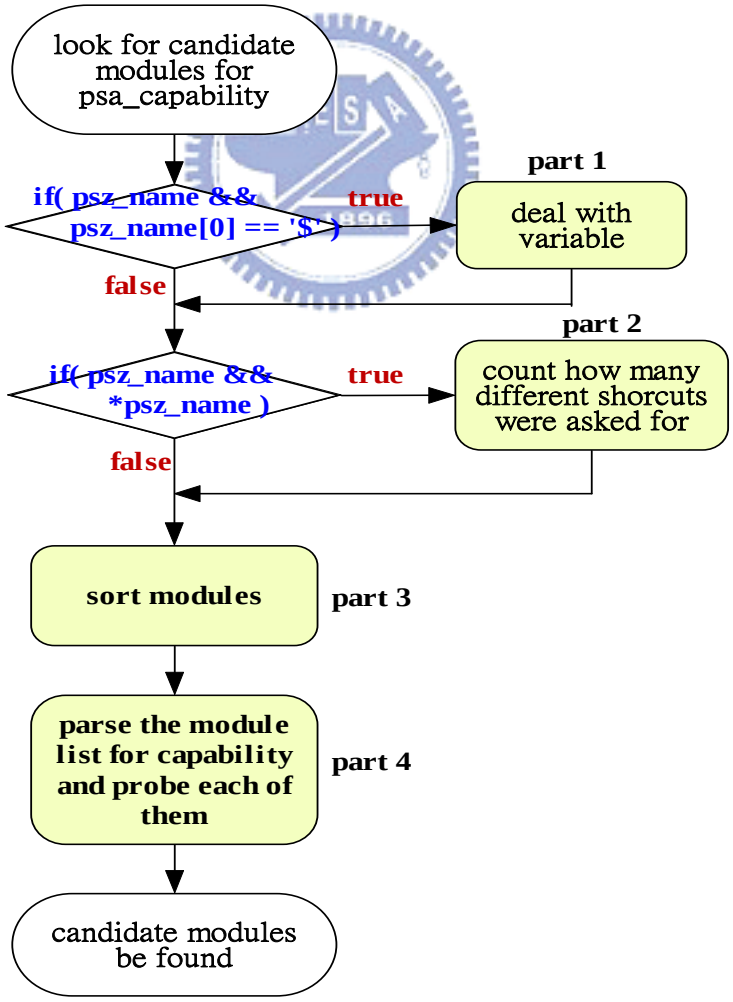

圖 4.3：第一階段運作流程

4.2.2.1 part 1 與 part 2 說明

在進入 part 1 和 part 2 前，會先確認 psz_name 此字串變數的所儲存的字串為何；若符合 part 1 之前的判斷式，在 part1 則會對 psz_name 變數做處理；若符合 part 2 之前的判斷式，在 part 2 則會去計算有多少個 shortcuts，計算 shortcut 的目的是為了在後面 part 4 部分內，利用 shortcuts 數目來決定有多少 i_shortcut_bonus，細節會在後面的 part 4 說明，而 shortcuts 的數目則是以變數 i_shortcuts 儲存之；然而如果 psz_name 此字串變數的所儲存的字串為空字串，那麼當然不會經過 part 1 和 part 2 這兩個部分的處理。

4.2.2.2 part 3 與 part 4 說明

```
/* Sort the modules and test them */
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 4.4：將所有 module 排序

如上圖 4.4 所示，在 part 3 是先將所有 module 排序後，到了 part 4 就會開始從這所有的 module 中，尋找出哪些是符合功能需求的，所以接下來的 part 4 就開始進行尋找的程序。

```
for( i_which_module = 0; i_which_module < p_all->i_count; i_which_module++ )
{
    int i_shortcut_bonus = 0;

    p_module = (module_t *)p_all->p_values[i_which_module].p_object;

    /* Test that this module can do what we need */
    if( strcmp( p_module->psz_capability, psz_capability ) )
    {
        /* Don't recurse through the sub-modules because vlc_list_find()
         * will list them anyway. */
        continue;
```

圖 4.5：part 4 內的尋找程序

如上圖 4.5 所示，在整個尋找的程序中，是在 for 迴圈內來確認每一個 module 是不是符合功能需求的 module，其中 p_all->i_count 所代表的是全部 module 的數量。進到 for 迴圈裡，就會將 part 3 所排序的 modules 一個個的比對檢查，也就是上圖 4.5 中的 p_module 結構變數，這個結構變數所代表的就是一個 module 的屬性，這個屬性的設定就是在 4.1 節裡我們所提到過的。如下圖 4.6 所示，這是 p_module 結構變數內的部分成員變數，以 psz_capability 與 i_score 這兩個成員變數來說，它們的設定就是 set_capability()此巨集定義的，例如前面所提的 libmpeg2 module 來說，set_capability("decoder",150) 就會設定成員變數 psz_capability="decoder"，i_score=150，表示設定這個 libmpeg2 module 的能力是 "decoder"，能力值是 150。而 VLC 就是去比對檢查每個 module 的設定值，來尋找出符合功能的模組。

```
char   *psz_capability;                    /* Capability */
int     i_score;                  /* Score for each capability */
uint32_t i_cpu;                    /* Required CPU capabilities */

vlc_bool_t b_unloadable;                  /* Can we be dlclosed? */
vlc_bool_t b_reentrant;                    /* Are we reentrant? */
vlc_bool_t b_submodule;                   /* Is this a submodule? */

/* Callbacks */
int  ( * pf_activate )   ( vlc_object_t * );
void ( * pf_deactivate ) ( vlc_object_t * );
```

圖 4.6：p_module 結構變數內的部分成員變數

由圖 4.5 可以看到，一開始必須要比對檢查的條件就是 p_module 的 capability 要符合 VLC 所需要的 psz_capability(也就是呼叫 module_Need()函數時所傳入的參數)，這表示 module 的能力要能夠符合 VLC 需要具備某種功能的模組，如果此 module 的能力不符合 VLC 的需要，就會認定此 module 不適合並重新比對檢查下一個 module。

接下來就是判別 i_shortcuts 的值來計算 i_shortcut_bonus 的值，而 i_shortcuts 的值則是在前面 part 2 時所得到的。如果 i_shortcuts 小於等於 0，那麼接下來會

判斷 i_score 是否為 0；若 i_shortcuts 大於 0，才會去計算 i_shortcut_bonus 的值是多少。每個 module 都會針對某個功能而設定的能力值(i_score)，而這個能力值的大小會影響到在第二階段的篩選過程的先後順序，module 的能力值越大，在第二階段時就會先比對檢查此 module。因此在第一階段比對檢查某個 module 時，會將所計算出的 i_shortcut_bonus 加上此 module 原先的 i_score，成為此 module 的新的 i_score。

所以在通過初步的比對檢查之後，就會先將此 modules 先儲存起來，成為 candidate module，並且將此 p_module 的 score 加上 i_shortcut_bonus，變成此 module 新的 score，最後要儲存的 module 也會按照新的 score 值由高到低來重新排列，排列完之後，又會回到 for 迴圈的開始處，繼續比對檢查下一個 module，直到所有的 module 比對檢查完畢。

```
/* Store this new module */
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 4.7：儲存符合功能需求的 modules

而我們把整個 part4 的程序以下圖 4.8 來表示：在符合兩個必要條件後，會判別 i_shortcuts 的值，若 i_shortcuts 大於 0，則會去計算 i_shortcut_bonus；若小於等於 0，就必須再判別 module 的 score 是否等於 0。通過了前面的篩選之後，將合適的 module 儲存起來並更新 modules 的 score 值，並且再按照 score 值的高低來將這些 module 以 link list 的方式來排列。等到第二階段要尋找合適的 module 時，就是從這些 candidate modules 來一一測試。

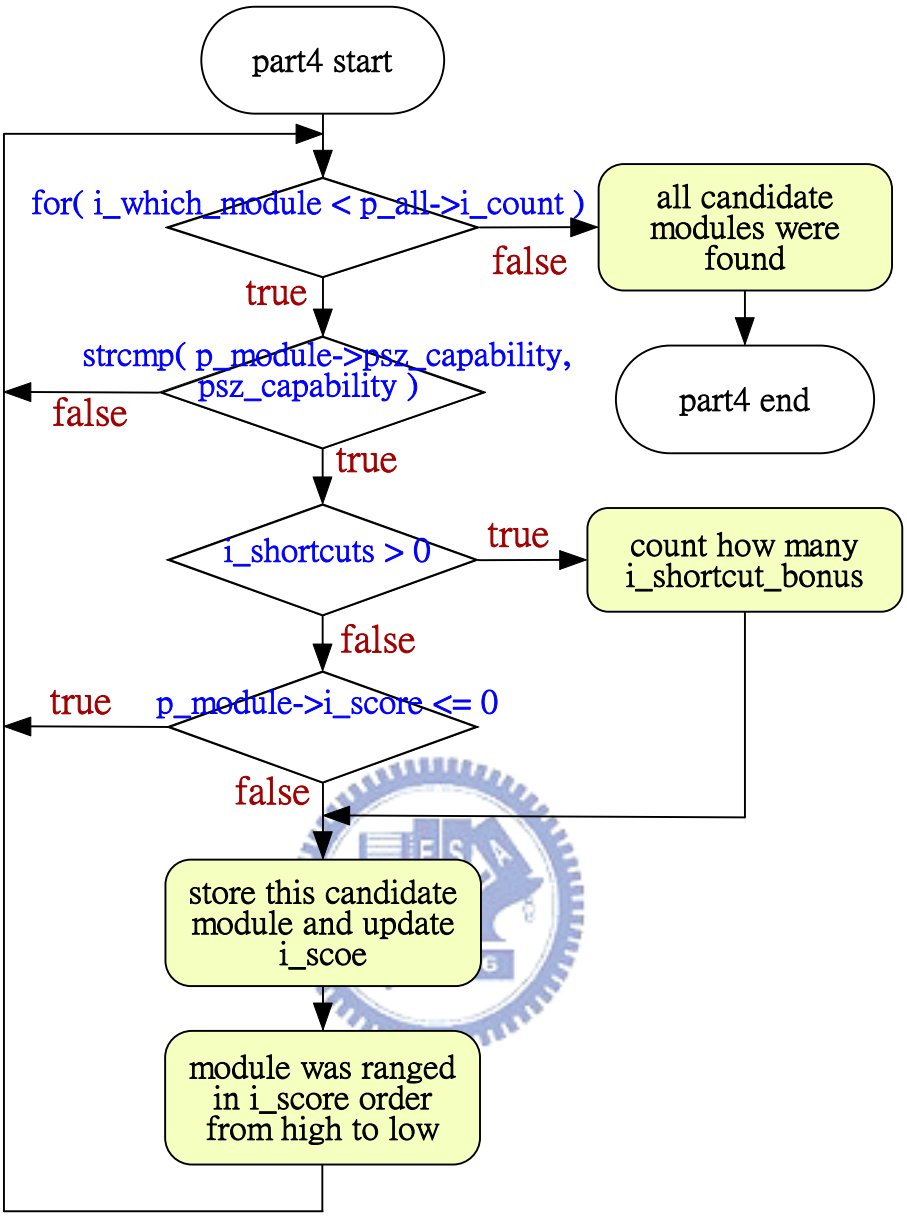

圖 4.8：part 4 的搜尋程序

4.2.3 第二階段—尋找出合適的模組

在經過前面第一階段的篩選之後，就會得到符合功能的 candidate modules，而這些 modules 是以 i_score 高低來排列。接下來的第二階段就是要從這些 candidate modules 中，尋找出一個合適的 module，決定 module 是否合適的方式，是從 i_score 最高的 module 開始測試，"一直到第一個 module 測試成功"，那麼就會選擇使用這個 module，表示此 module 就是合適的 module，而剩下的 candidate modules 就不再去測試，並結束整個 module_Need()程序。

圖 4.9：第二階段程序之程式碼

上圖 4.9 是第二階段實際在選擇合適 module 的過程，主要是在此 while 迴圈內來比對測試第一階段所搜尋出來的 module 中那一個才是否合適的模組，而前面有提到過在第一階段會從 2 百多個模組當中，先搜尋出符合功能需求的模組，並將這些符合功能需求的模組以 link list 的方式串聯起來，因此圖 4.9 中的 p_first 指標就是指向這個 link list 模組開頭的第一個模組，所以在進到 while 迴圈之前，p_tmp 指標就是先指向這個 link list 模組的第一個模組，然後呼叫此 module 內的起始函數來比對測試是否此模組是否為合適的模組，若符合則會跳出此迴圈，不再比對測試其它的 module；假如此模組不是合適的模組，那麼 p_tmp 就會重新指向下一個模組 p_tmp->p_next，並且一樣呼叫此 module 內的起始函數來比對測試是否此模組是否為合適的模組。

現在我們要探討 p_tmp->p_module->pf_activate 這個函數指標，它是指向 module 的"起始函數"，前面圖 4.1 在說明模組的定義時，可以看到 set_callbacks(OpenDecoder, CloseDecoder)此行設定，這就是設定的此 module 的起始函數為 OpenDecoder()函數，而終止函數為 CloseDecoder()函數，這些函數的定義一樣是在 libmpeg2.c 內。所以假如目前是在測試此 libmpeg2 module，呼叫 p_tmp->p_module->pf_activate(p_this)，就等同於是呼叫 OpenDecoder()此函數。

因此在第二階段中如何真正的決定此 module 是否為合適的 module，就是要進到此 module 的起始函數裡，而每個 module 的起始函數內都有各自的比對條件或是初始化條件，若符合此函數內的條件，則最後會回傳 VLC_SUCCESS 值，若不符合，則回傳表示錯誤訊息的值(一般是回傳 VLC_EGENERIC 值)。

所以我們簡單的總結一下整個 module_Need()函數的運作流程：第一階段是從所有的 module 當中，先找出符合功能需求的 candidate modules；第二階段則是進到 module 的起始函數來測試此 module 是否為合適的 moudle。而對於每一種功能 module 的起始函數內的比對條件或初始化條件都不一樣，因此接下來的的 4.3 和 4.4 小節，我們會說明部分 demux2 module 和 decoder module 內的起始函數，來了解 VLC 如何尋找出合適的解多工器模組與解碼器模組。

4.3 尋找合適解多工器模組程序

我們在第三章時有提到，呼叫 module_Need()來尋找合適的解多工器模組時，並且傳入的 psz_capability 字串必須為"demux2"，也就是 VLC 需要具有"demux2"能力的模組。而經過第一階段的比對檢查之後，會從所有的 2 百多個模組當中，篩選出具有"demux2"能力的模組，然後在第二階段開始就進到模組內的起始函數，測試此模組是否為合適的解多工器模組。因此接下來我們實際以 MPEG-2 PS(program stream)/TS(transport stream 這兩種影音檔案格式為例子，說明 VLC 如何找到合適的解多工器模組來解析這兩種影音檔案格式。

4.3.1 尋找合適的 MPEG-2 PS 解多工器模組

如下圖 4.10 所示，一個 MPEG-2 PS 是由許多 pack 組合而成的，每個 pack 都是由一個 pack header 加上多個 PES packet 所形成的，而每個 pack header 都是由一列固定的起始碼開始，起始碼是一串特殊的編碼值，其可以區分出資料流的每層架構；而 pack header 是由 pack_start_code: 0x000001BA 開始的。

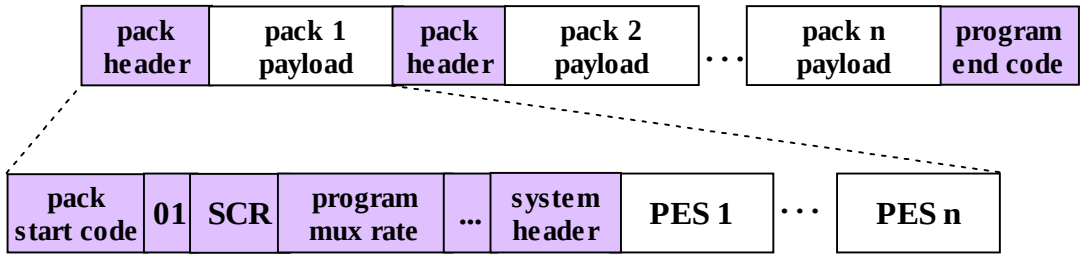

圖 4.10：MPEG-2 PS

而在 VLC 解多工器模組當中，有一個 ps module 是用來解析 MPEG-2 PS 影音包裝格式。在 ps module 的起始函數內一開始會先確認存放影音資料的 buffer 的長度是否小於 4 個 bytes，然後再比對這個 buffer 前面 4 個 bytes 的值是否小於 0x000001b9(如下圖 4.11)，只要 buffer 內的長度沒有小於 4 個 bytes 並且 buffer 前面 4 個 bytes 符合的比對條件，那麼 VLC 就會確認讀取資料的影音包裝格式為 MPEG-2 PS，並且選擇 ps module 為合適的解多工器模組。

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

if( p_peek[0] != 0 || p_peek[1] != 0 ||
    p_peek[2] != 1 || p_peek[3] < 0xb9 )
{
    if( !p_demux->b_force ) return VLC_EGENERIC;

    msg_Warn( p_demux, "this does not look like an MPEG PS stream, "
              "continuing anyway" );
}
```

若讀取的影音資料前面四個位元組符合比對條件，那麼就認定此影音包裝格式為 MPEG-2 PS。

圖 4.11：ps module 起始函數部分程式碼

4.3.1 尋找合適的 MPEG-2 TS 解多工器模組

如下圖 4.12 所示，一個 MPEG-2 TS 是由許多 TS packet 組合而成的，而 TS packet 是由長度固定為 184 bytes 的 PES 再加上 4 bytes 的 TS header 所組成，共 188 bytes，依據 ISO13818-1 中所定義，TS header 是由 8 bits 的 sync_byte (0x47) 所開始的。

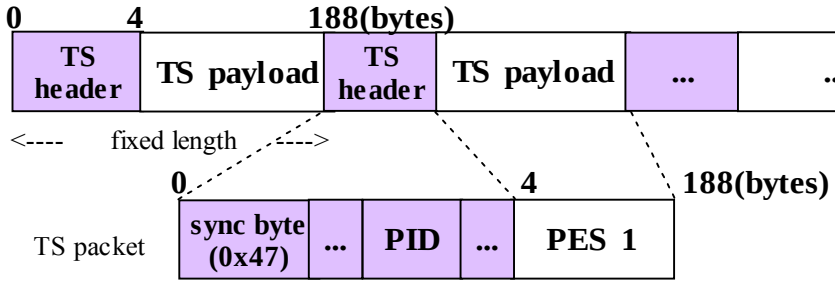

圖 4.12：MPEG-2 TS

而同樣地，在 VLC 解多工器模組當中，有一個 ts module 是用來解析 MPEG-2 TS 影音包裝格式。在 ts module 起始函數內一開始會先確認 buffer 內的長度不能小於 TS_PACKET_SIZE_MAX(定義為 204)個 bytes，然後在 buffer 內的前 204 個 bytes 中必須找到一個 TS sync_byte 0x47，並記錄這個 sync_byte 在 buffer 內的位置為 i_sync；並且再確認 buffer 長度不能小於 i_peek=TS_PACKET_SIZE_MAX* 3 + i_sync + 1，最後就是要在 buffer 內位置 i_sync 之後，每隔 188 bytes 連續比對到 3 個 TS sync_byte 0x47(下圖 4.13)，只要符合上面所說的條件，那麼 vlc 就會判定輸入檔案的影音包裝格式為 transport stream，並且選擇 ts module 為合適的解多工器模組。

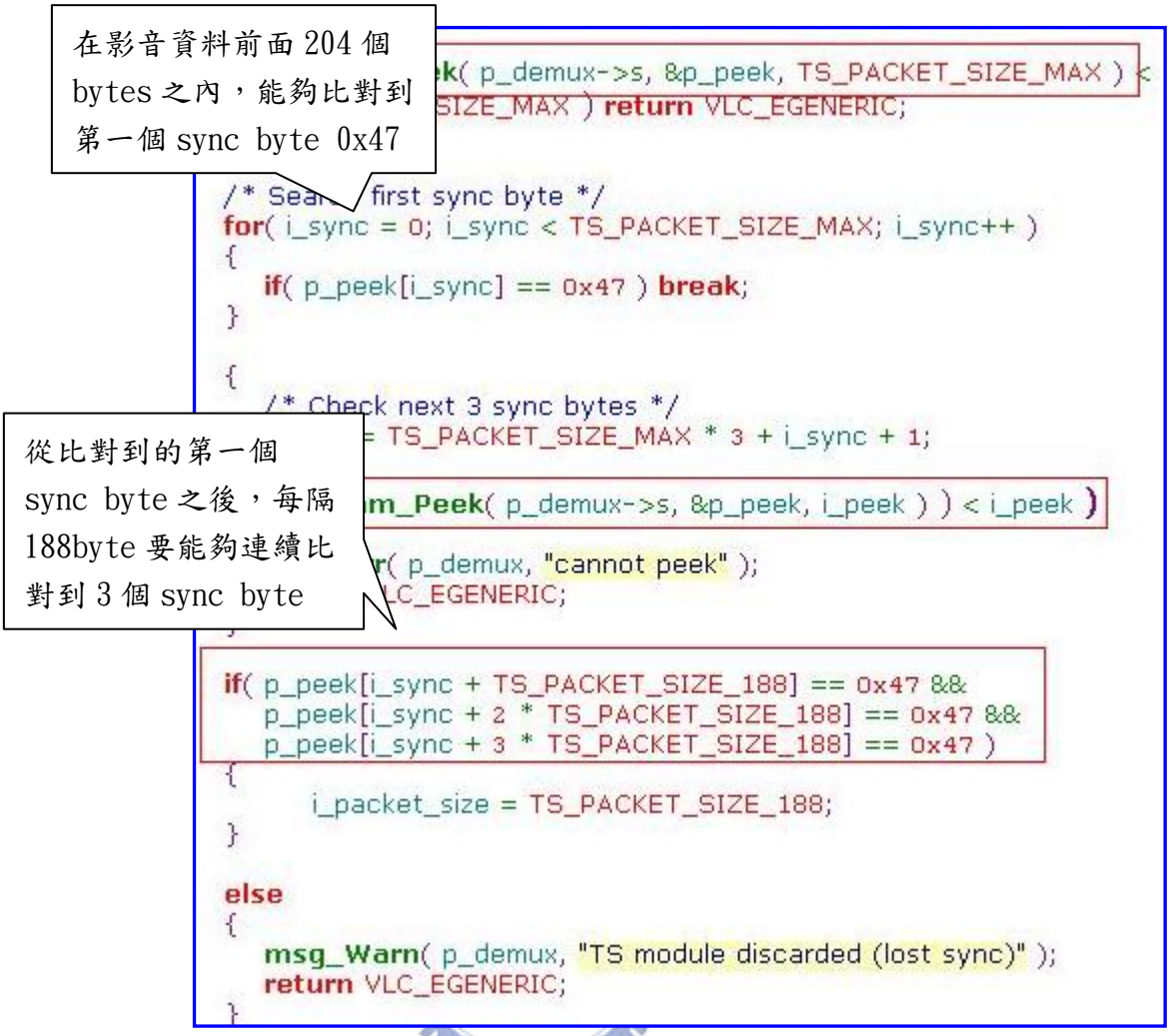

圖 4.13：ts module 起始函數部分程式碼

由上面所舉例的兩種影音格式來看，可以了解到 VLC 尋找一個合適的解多工器模組的方法，就是依據每種解多工器模組的起始函數內的比對條件，來確定此影音包裝格式為何；若符合某個解多工器模組的起始函數內的比對條件，那麼也就代表選擇了此模組為合適的解多工器模組，因此，VLC 才能使用此模組來解多工所讀取的影音資料。

4.4 尋找合適的解碼器模組程序

在第三章說明整個影音播放程序時有提到，VLC 在使用解多工器模組解析來源資料的影音包裝格式的時候，會找出 audio 與 video 的編碼格式，然後將 audio

與 video 的編碼格式分別以四字元碼型式來表示，並且呼叫 module_Need()函數來尋找合適的 audio decoder module 和 video decoder module，並且傳入的 psz_capability 字串必須為"decoder"，也就是 VLC 需要具有"decoder"能力的模組。而經過第一階段的比對檢查之後，會從所有的 2 百多個模組當中，搜尋出具有"decoder"能力的模組，然後在第二階段開始就進到模組內的起始函數，測試此模組是否為合適的解碼器模組，而如何找到合適的 decoder module 就是跟四字元碼有關。在前面有提到針對 MPEG-2 PS 檔案包裝格式，VLC 會找到 ps module 來進行解多工的程序，所以首先在 4.4.1 小節我們會以 ps module 為例子，說明 VLC 如何取得 audio 與 video 的編碼格式的四字元碼型態，然後 4.4.2 小節則是說明如何利用這個四字元碼來找到合適的解碼器模組。

4.4.1 取得四字元碼

在 ps module 進行 MPEG-2 PS 的解多工程序時，並不是一開始就能夠取得四字元碼，也就說 ps module 並不是在一開始進行 MPEG-2 PS 解多工程序時，就能夠馬上確認出 audio 和 video 的編碼格式。因為一個 MPEG-2 PS 是由許多 pack 組合而成的，每個 pack 都是由一個 pack header 加上多個 PES packet 所形成的，所以我們把 MPEG-2 PS 的結構大致上分為三個階層：第一層為 pack header，第二層為 system header，第三層為 PES，每一層都是由特殊的起始碼所開始的，而視訊位元流以及音訊位元流則是存在 PES packet 中。

因此當 ps module 在開始進行 MPEG-2 PS 解多工程序時，會先解多工出 pack header 並解析 pack header；之後再解多工出 system header 並解析 system header，而接下來會將 pack 所包含的 PES packet 一個一個的解多工出來並解析每一個 PES packet，每個 PES packet 的 PES payload 為視訊位元流或音訊位元流等其他資料類型。如果發現解多工出來的 PES packet 資料類型為 PSM(program stream map)，那麼在解析此 PES packet 時才能夠確認出 audio 和 video 的編碼格式並取得相對應的四字元碼。所以 ps module 在開始進行解多工程序時，並不是馬上先找出 PSM PES packet 來確認 audio 和 video 的編碼格式，而是要經過一層一層的解多工之

後，直到解多工出 PSM PES packet，然後再經過解析 PSM PES packet 來確認出 audio 和 video 的編碼格式並取得相對應的四字元碼。而接下來將會說明如何從 PSM PES packet 當中確認出 audio 和 video 的編碼格式並取得相對應的四字元碼。

如下圖 4.14 所示，一個完整的 pack header 會包有其它訊息欄位以及 system header，而 system header 是由 system_header_start_code:0x000001BB 所開始的，並且在 System header 中的 stream_id 欄位會定義此一 PS 的 video stream 和 audio stream 的類型，從表 2.1 中可以知道，當 stream_id 的值在 0xE0~0xEF 間，表示此 PS 中的 PES payload 包含了 video stream；如果位於 0xC0~0xDF 間的話表示此 PS 中的 PES payload 包含了 audio stream。

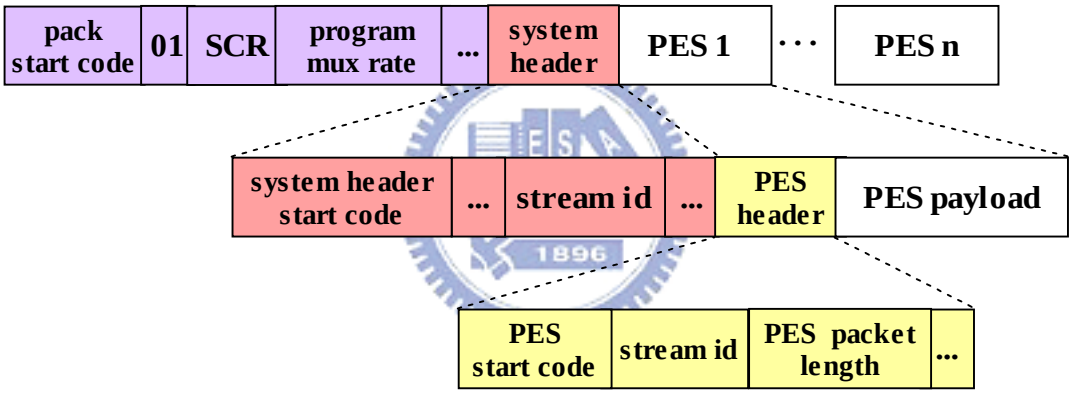

圖 4.14：header of MPEG-2 PS

Stream_id	Stream coding
1011 1100	Program_stream_map
1011 1101	Private_stream_1
1011 1110	Paddinf_stream
1011 1111	Private_stream_2
110x xxxx (C0~DF)	ISO/IEC 13818-3 or ISO/IEC 11172-3 audio stream
1110 xxxx (E0~EF)	ITU-T Rec.H.262 | ISO/IEC 13818-2 or ISO/IEC 11172-3 video stream
1111 0000	ECM_stream
1111 0001	EMM_stream

1111 0010	ITU-T Rec.H.222.0 | ISO/IEC 13818-1 Annex A or ISO/IEC 13818-6 DSMCC stream
1111 0011	ISO/IEC_13522_stream
1111 0100	ITU-T Rec.H.222 | type A
1111 0101	ITU-T Rec.H.222 | type B
1111 0110	ITU-T Rec.H.222 | type C
1111 0111	ITU-T Rec.H.222 | type D
1111 1000	ITU-T Rec.H.222 | type E
1111 1001	Ancillary_stream
1111 (1010-1110)	reserved data stream
1111 1111	program_stream_directory

表 2.1 stream_id assignments

至於 PES header 中的 stream_id 欄位表示其 payload data 的為 video stream 或是 audio stream，舉例來說：當 PES header 中的 stream_id= 0xE0 時，表示此 PES 所攜帶的 payload data 為 video stream；而如果 PES header 的 stream_id=0xBC，表示此 PES 為 PSM(program stream map)，它所挾帶 stream_type 和 elementary_stream_id 這兩個欄位的訊息可以表示這一個 PS 中的 PES payload 的編碼格式。假設 program_stream_map 的 elementary_stream_id =0xE0，stream_type 為 0x02，根據表 2.2 可以知道 stream_type 為 0x02 為保留型別，定義為 mpeg-2 video codec，因此只要是 PES header 中的 stream_id 為 0xE0 的話，代表此 video stream 編碼格式為 mpeg-2 video。

value	Description
0x00	ITU-T | ISO/IEC Reserved
0x01	ISO/IEC 11172 video
0x02	ITU-T Rec.H.262 | ISO/IEC 13818-2 or ISO/IEC 11172-2 constrained parameter video stream
0x03	ISO/IEC 11172 audio
0x04	ISO/IEC 13818-3 audio
0x05	ITU-T Rec.H.222.0 | ISO/IEC 13818-1private_sections
0x06	ITU-T Rec.H.222.0 | ISO/IEC 13818-1PES packets containing private data
0x07	ISO/IEC 13522 MHEG
0x08	ITU-T Rec.H.222.0 | ISO/IEC 13818-1 Annex A DSM CC

0x09	ITU-T Rec.H.222.1
0x0A	ISO/IEC 13818-6 type A
0x0B	ISO/IEC 13818-6 type B
0x0C	ISO/IEC 13818-6 type C
0x0D	ISO/IEC 13818-6 type D
0x0E	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 auxiliary
0x0F	ISO/IEC 13818-7 Audio with ADTS transport syntax
0x10	ISO/IEC 14496-2 Visual
0x11	ISO/IEC 14496-3 Audio with the LATM transport syntax as defined in ISO/IEC 14496-3 / AMD 1
0x12	ISO/IEC 14496-1 SL-packetized stream or FlexMux stream carried in PES packet
0x13	ISO/IEC 14496-1 SL-packetized stream or FlexMux stream carried in ISO/IEC14496_sections
0x14	ISO/IEC 13818-6 Synchronized Download Protocol
0x15-0x7F	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 Reserved
0x80-0xFF	User Private

表 2.2 stream type assignment

所以當 ps module 在解析 MPEG-2 PS 時，就是藉由 PSM 內的 stream_type 以及 elementary_stream_id 這兩個欄位，判斷 audio stream 和 video stream 的編碼格式為何，並且設定相對應的 codec type，也就是以四字元碼來表示什麼樣的 codec type。而設定四字元碼的部分則是在 ps_track_fill()函數來完成(如下圖 4.15)，而在設定完 audio 和 videocodec type 的四字元型態之後，就會利用所設定的 audio codec type 和 video codec type 尋找合適的解碼器模組。

```
es_format_Init( &tk->fmt, UNKNOWN_ES, 0 );

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') )
}
```

圖 4.15：ps_track_fill()函數內部分程式碼

以前面所舉的例子來說，假設 PSM 的 elementary_stream_id=0xE0，stream_type 為 0x02，代表此 video stream 編碼格式為 mpeg-2 video，因此在上圖 4.15 所以就可以看到設定相對應的四字元碼"mpgv"，資料型態為 VIDEO_ES，而這些設定值都儲存在 fmt 這個結構變數內(下圖 4.16)。

```
static inline void es_format_Init( es_format_t *fmt,
                                   int i_cat, vlc_fourcc_t i_codec )
{

    fmt->i_cat              = i_cat;
    fmt->i_codec            = i_codec;
    fmt->i_id               = -1;
    fmt->i_group            = 0;
```

圖 4.16：es_format_Init()函數內部分程式碼

經由解多工器在解析影音資料的過程中，可以得知 video 和 audio 的編碼格式，並且設定相對的四字元碼，因此接下來我們就要說明如何利用所設定的四字元碼來尋找合適的解碼器模組。

4.4.2 利用四字元碼尋找合適解碼器模組

想要選擇合適的 decoder module，會藉由呼叫 module_Need()函數來尋找合適的 decoder module。在 module_Need()函數內有兩階段的篩選動作，在經過第一階段的篩選之後，decoder modules 共有 26 個；第二階段則是會去呼叫每個 decoder module 的起始函數。基本上，每個 decoder modules 的起始函數內，都有不同的比對條件以初始條件，只要其中一個條件不符合，那就會回傳一個表示錯誤訊息的值，並測試下一個 module。但是這些 decoder modules 的起始函數內都有一個共同的比對條件，就是必須要比對先前所設定的四字元碼。每個 decoder module 所針對 codec type 都不同，例如下圖 4.16 所顯示 libmpeg2 module，此 module 只針對四字元碼為"mpgv"、"mpg1"、"PIM1"、"VCR2"、"mp2v"、"mpg2"、"hdv2"這幾種編碼格式來解碼，而 p_dec->fmt_in.i_codec 所儲存的就是先前針對影音編碼格式所設定的四字元碼，比對的過程中如果設定的四字元碼沒有此 module 所支援的編碼格式，那麼就表示此 module 並不是合適的 decoder module，則回傳 VLC_EGENERIC 值並測試下一個 decoder module。

```
/*****************************************************************
 * OpenDecoder: probe the decoder and return score
 *****************************************************************
static int OpenDecoder( vlc_object_t *p_this )
{
    decoder_t *p_dec = (decoder_t*)p_this;
    decoder_sys_t *p_sys;
    uint32_t i_accel = 0;

    if( p_dec->fmt_in.i_codec != VLC_FOURCC('m','p','g','v') &&
        p_dec->fmt_in.i_codec != VLC_FOURCC('m','p','g','1') &&
        /* Pinnacle hardware-mpeg1 */
        p_dec->fmt_in.i_codec != VLC_FOURCC('P','I','M','1') &&
        /* ATI Video */
        p_dec->fmt_in.i_codec != VLC_FOURCC('V','C','R','2') &&
        p_dec->fmt_in.i_codec != VLC_FOURCC('m','p','2','v') &&
        p_dec->fmt_in.i_codec != VLC_FOURCC('m','p','g','2') &&
        p_dec->fmt_in.i_codec != VLC_FOURCC('h','d','v','2') )
    {
        return VLC_EGENERIC;
    }
```

圖 4.17：libmpeg2 module 起始函數部分程式碼

這個四字元碼所代表的就是某種 audio 或是 video 的編碼格式，在每個 decoder module 內的起始函數內所要比對的四字元碼，是表示此 decoder module 可以針對

這幾種編碼格式來解碼。所以解多工器在解析影音資料的過程中所設定 video 和 audio 的編碼格式相對的四字元碼,如果符合某個 decoder module 內的起始函數內所要比對的四字元碼,表示此 decoder module 可以對這種編碼格式進行解碼動作。

所以可以了解到 VLC 尋找一個合適的解碼器模組的方法,就是在解多工器在解析影音資料的過程中確認出 audio 和 video 的編碼格式，並且設定相對應的四字元碼；之後第二階段的篩選時，進到 decoder module 的起始函數內，測試是否符合起始函數內的比對條件以初始條件，若符合某個解碼器模組的起始函數內的比對條件和初始條件，那麼也就代表選擇了此模組為合適的解碼器模組，因此 VLC 才能使用此解碼器模組來進行解碼的程序。

4.5 解碼器模組整理

以下我們將整理出這 26 個 decoder modules，以及各自起始函數內的比對條件以及初始化條件。

1. **cmml decoder module**(在 cmml.c 內)

   decoder type：

   起始函數：static int OpenDecoder(vlc_object_t *p_this)

   codec type：**"cmml"**

   比對條件：比對 p_dec->fmt_in.i_codec 必須符合 codec type

   初始化條件：必須有足夠的記憶體空間來做動態記憶體配置

2. **dmo decoder module**(在 dmo.c 內)

   decoder type：audio/video

   起始函數：static int DecoderOpen(vlc_object_t *p_this)

   codec type：/* WVC1 */：**"WVC1"**、**"wvc1"**
   /* WMV3 */：**"WMV3"**、**"wmv3"**
   /* WMV2 */：**"WMV2"**、**"wmv2"**
   /* WMV1 */：**"WMV1"**、**"wmv1"**

/* WMA 3 */：**"WMA3"**、**"wma3"**、**"wmap"**
/* WMA 2 */：**"WMA2"**、**"wma2"**
/* WMA Speech */："wmas"

比對條件：比對 p_dec->fmt_in.i_codec 必須符合 codec type

3. **ffmpeg decoder module**(在 ffmpeg.c 內)

decoder type：audio/video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：video codecs

/* MPEG-1 Video */：**"mp1v"**
/* MPEG-2 Video */：**"mp2v"**、**"mpgv"**
/* MPEG-4 Video */：**"DTVX"**、**"divx"**、**"MP4S"**、**"mp4s"**、**"M4S2"**、**"m4s2"**、**"xvid"**、**"XVID"**、**"XviD"**、**"XVIX"**、**"xvix"**、**"DX50"**、**"dx50"**、**"mp4v"**、**"MP4V"**、**4000**、**"m4cc"**、**"M4CC"**、**"FMP4"**
/* 3ivx delta 4 */：**"3IV2"**、**"3iv2"**
/* MSMPEG4 v1 */：**"DIV1"**、**"div1"**、**"MPG4"**、**"mpg4"**
/* MSMPEG4 v2 */：**"DIV2"**、**"div2"**、**"MP42"**、**"mp42"**
/* MSMPEG4 v3* /：**"MPG3"**、**"mpg3"**、**"div3"**、**"MP43"**、**"mp43"**
/* DivX 3.20 */：**"DIV3"**、**"DIV4"**、**"div4"**、**"DIV5"**、**"div5"**、**"DIV6"**、**"div6"**
/* AngelPotion stuff */：**"AP41"**
/* 3ivx doctered divx files */：**"3IVD"**、**"3ivd"**
/* unknow */：**"3VID"**、**"3vid"**
/* Sorenson v1 */：**"SVQ1"**
/* Sorenson v3 */：**"SVQ3"**
/* h264 */：**"h264"**、**"H264"**、**"x264"**、**"X264"**
/* avc1: special case h264 */：**"avc1"**、**"VSSH"**、**"vssh"**
/* H263 */：**"H263"**、**"h263"**、**"U263"**、**"M236"**
/* H263i */：**"I263"**、**"i263"**
/* Flash (H263) variant */：**"FLV1"**
#if LIBAVCODEC_BUILD > 4716：**"H261"**
#if LIBAVCODEC_BUILD > 4680：**"FLIC"**
/* MJPEG */：**"MJPG"**、**"mipg"**、**"mjpa"**、**"jpeg"**、**"JPEG"**、**"JFIF"**、**"JPGL"**
/* for mov file */：**"mjpb"**

#if LIBAVCODEC_BUILD > 4680：**"SP5X"**
/* DV */：**"dvs1"、"dvsd"、"DVSD"、"dvhd"、"dvc "、"dvcp"、**
"dvp "、"dvpp"、"CDVC"
/* Windows Media Video */：**"WMV1"、"WMV2"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(10<<8)+1：
"WMV3"、"WCV1"
#if LIBAVCODEC_BUILD >= 4683：
/* Microsoft Video 1 */：**"MSVC"、"msvc"、"CRAM"、"cram"、**
"WHAM"、"wham"
/* Microsoft RLE */：**"mrle"、0x1,0x0,0x0,0x0**
/* Indeo Video Codecs (Quality of this decoder on ppc is not good) */
"IV31"、"iv31"、"IV32"、"iv32"
#if LIBAVCODEC_BUILD >= 4721：**"tscc"**
/* Huff YUV */：**"HFYU"**
/* Creative YUV */：**"CYUV"**
/* On2 VP3 Video Codecs */：**"VP3 "、"VP30"、"VP31"、"vp31"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(14<<8)+0)：
"VP5 "、"VP50"、"VP62"、"vp62"、"VP6F"
#if LIBAVCODEC_VERSION_INT >= ((51<<16)+(27<<8)+0)：
"VP60"、"VP61"
#if LIBAVCODEC_BUILD >= 4685：
/* Xiph.org theora */：**"the0"**
#if (!defined(WORDS_BIGENDIAN))：
/* Asus Video (Another thing that doesn't work on PPC) */：
"ASV1"、"ASV2"
/* FFMPEG Video 1 (lossless codec) */：**"FFV1"**
/* ATI VCR1 */：**"VCR1 "**
/* Cirrus Logic AccuPak */：**"CLJR"**
/* Real Video */：**"RV10"、"RV13"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(15<<8)+1)：**"RV20"**
#if LIBAVCODEC_BUILD >= 4684：
/* Apple Video */：**"rpza"、"smc "**
/* Cinepak */：**"cvid"**
/* Id Quake II CIN */：**"IDCI"**
/* 4X Technologies */：**"4xmv"**
#if LIBAVCODEC_BUILD >= 4694：
/* Duck TrueMotion */：**"DUCK"**
/* Interplay MVE */：**"imve"**

/* Id RoQ */：**"RoQv"**
/* Sony Playstation MDEC */：**"MDEC"**
#if LIBAVCODEC_BUILD >= 4699：
/* Sierra VMD */：**"vmdv"**
#if LIBAVCODEC_BUILD >= 4719：
/* FFMPEG's SNOW wavelet codec */：**"SNOW"**
#if LIBAVCODEC_BUILD >= 4752：**"cle "、"qdrw"、"QPEG"、"Q1.0"、"Q1.1"、"ULTI"、"VIXL"、"LOCO"、"WNV1"、"AASC"**
#if LIBAVCODEC_BUILD >= 4753：**"IV20"、"RT21"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(13<<8)+0)：**"VMnc"**
Image codecs
#if LIBAVCODEC_BUILD >= 4731：**"png "、"ppm "、"pgm "、"pgmy"、"pam "**
#if LIBAVCODEC_BUILD >= ((51<<16)+(0<<8)+0)：**"bmp "**
Audio Codecs
/* Windows Media Audio 1 */：**"WMA1"、"wma1"**
/* Windows Media Audio 2 */：**"WMA2"、"wma2"**
/* DV Audio */：**"dvau"**
/* MACE-3 Audio */：**"MAC3"**
/* MACE-6 Audio */：**"MAC6"**
/* RealAudio 1.0 */：**"14_1"**
/* RealAudio 2.0 */：**"28_8"**
/* MPEG Audio layer 1/2/3 */：**"mpga"、"mp3 "**
/* A52 Audio (aka AC3) */：**"a52 "、"a52b "**
#if LIBAVCODEC_BUILD >= 4719：
/* DTS Audio */：**"dts "**
/* AAC audio */：**"mp4a"**
/* 4X Technologies */：**"4xma"**
/* Interplay DPCM */：**"idpc"**
/* Id RoQ */：**"RoQa"**
#if LIBAVCODEC_BUILD >= 4685：
/* Sony Playstation XA ADPCM */：**"xa "**
/* ADX ADPCM */：**"adx "**
#if LIBAVCODEC_BUILD >= 4699：
/* Sierra VMD */：**"vmda"**

#if LIBAVCODEC_BUILD >= 4706
/* G.726 ADPCM */：**"g726"**
#if LIBAVCODEC_BUILD >= 4683：
/* AMR */：**"samr"**、**"sawb"**
#if LIBAVCODEC_BUILD >= 4703：
/* FLAC */：**"flac"**
#if LIBAVCODEC_BUILD >= 4745：
/* ALAC */：**"alac"**
#if LIBAVCODEC_BUILD >= ((50<<16)+(0<<8)+1)：
/* QDM2 */：**"QDM2"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(0<<8)+0)：
/* COOK */：**"cook"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(4<<8)+0)：
/* TTA: The Lossless True Audio */：**"TTA1"**
#if LIBAVCODEC_BUILD >= ((51<<16)+(8<<8)+0)：
/* Shorten */：**"shn "**
#if LIBAVCODEC_BUILD >= ((51<<16)+(16<<8)+0)：
/* WavPack */：**"WVPK"**
/* PCM */：**"s8 "**、**"u8 "**、**"s161"**、**"s16b"**、**"u161"**、**"u16b"**、
"s241"、**"s24b"**、**"u241"**、**"u24b"**、**"s321"**、**"s32b"**、
"u321"、**"u32b"**、**"alaw"**、**ulaw**

比對條件：

1.比對 p_dec->fmt_in.i_codec 必須符合 codec type。

2.i_codec_id 不能等於 CODEC_ID_AAC。

初始化條件：

1.p_codec=avcodec_find_decoder(i_codec_id);p_codec 不等於 0。

2.p_context = avcodec_alloc_context()；p_context 不等 0。

3.i_cat 等於 VIDEO_ES 且 i_result = E_(InitVideoDec)(p_dec, p_context, p_codec,i_codec_id, psz_namecodec); i_result 等於 0；或 i_cat 等於 AUDIO_ES 且 i_result = E_(InitAudioDec)(p_dec, p_context, p_codec, i_codec_id, psz_namecodec); i_result 等於 0。

4. **a52 decoder module**(在 a52.c 內)

   decoder type：audio

   起始函數：static int OpenDecoder(vlc_object_t *p_this)

   codec type：**"a52 "**、**"a52b"**

   比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type。

   初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

5. **adpcm decoder module**(在 adpcm.c 內)

   decoder type：audio

   起始函數：static int OpenDecoder(vlc_object_t *p_this)

   codec type：/* IMA ADPCM */：**"ima4"**
   /* MS ADPCM */：**"ms0x000x02"**
   /* IMA ADPCM */：**"ms0x000x11"**
   /* Duck DK4 ADPCM */：**"ms0x000x61"**
   /* Duck DK3 ADPCM */：**"ms0x000x62"**
   /* EA ADPCM */：**"XAJ0"**

   比對條件：1.比對 p_dec->fmt_in.i_codec 是否符合 codec type
   2. p_dec->fmt_in.audio.i_channels 的值為整數 1 到 5
   3. p_dec->fmt_in.audio.i_rate 必須大於 0。

   初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

6. **araw decoder module**(在 araw.c 內)

   decoder type：audio

   起始函數：static int DecoderOpen(vlc_object_t *p_this)

   codec type：/* from wav/avi/asf file */：**"araw"**、**"pcm "**、**"aflt"**
   /* _signed_ big endian samples (mov)*/：**"twos"**
   /* _signed_ little endian samples (mov)*/：**"sowt"**
   "alaw"、**"ulaw"**、**"mlaw"**、**"fl64"**、**"fl32"**、**"s32l"**、**"s32b"**、**"s24l"**、**"s24b"**、**"s16l"**、**"s16b"**、**"s8 "**、**"u8 "**

   比對條件：1. 比對 p_dec->fmt_in.i_codec 是否符合 codec type
   2. p_dec->fmt_in.audio.i_channels 的值為整數 1 到 8
   3. p_dec->fmt_in.audio.i_rate 必須大於 0。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

7. **cinepak decoder module**(在 cinepak.c 內)

   decoder type：video

   起始函數：static int OpenDecoder(vlc_object_t *p_this)

   codec type：**"cvid"**、**"CVID"**
   比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

   初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

8. **cvdsub decoder module**(在 cvdsub.c 內)

   decoder type：subtitle

   起始函數：static int DecoderOpen(vlc_object_t *p_this)

   codec type：**"cvd "**
   比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

9. **dts decoder module**(在 dts.c 內)

   decoder type：audio

   起始函數：static int OpenDecoder(vlc_object_t *p_this)

   codec type：**"dts "**、**"dtsb"**
   比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

   初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

10. **dvbsub decoder module**(在 dvbsub.c 內)

    decoder type：subtitle

    起始函數：static int Open(vlc_object_t *p_this)

    codec type：**"dvbs"**
    比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

11. **faad decoder module**(在 faad.c 內)

decoder type：audio

起始函數：static int Open(vlc_object_t *p_this)

codec type："**mp4a**"
比對條件：1. 比對 p_dec->fmt_in.i_codec 是否符合 codec type

2. p_sys->hfaad = faacDecOpen()不等於 NULL。

3. p_dec->fmt_in.i_extra 小於 0，或者 p_dec->fmt_in.i_extr 大於 0

但 faacDecInit2(p_sys->hfaad, p_dec->fmt_in.p_extra,

p_dec->fmt_in.i_extra,&i_rate, &i_channels)要大於 0。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

12. fake decoder module(在 fake.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type："**fake**"
比對條件：1. 比對 p_dec->fmt_in.i_codec 是否符合 codec type

2. (val.psz_string == NULL || !*val.psz_string)條件不成立

3. p_image 不等於 0。

13. flac decoder module(在 flac.c 內)

decoder type：audio

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type："**flac**"
比對條件：

1. 比對 p_dec->fmt_in.i_codec 是否符合 codec type

2. p_sys->p_flac = FLAC__stream_decoder_new()；p_sys->p_flac

不等於 0。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

14. libmpeg2 decoder module(在 libmpeg2.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"mpgv"**、**"mpg1"**
/* Pinnacle hardware-mpeg1 */：**"PIM1"**
/* ATI Video */：**"VCR2"**、**"mp2v"**、**"mpg2"**、**"hdv2"**
比對條件：

1.比對 p_dec->fmt_in.i_codec 是否符合 codec type

2.p_sys->p_mpeg2dec = mpeg2_init()；p_sys->p_mpeg2dec 不等於 0。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

15. lpcm decoder module(在 lpcm.c 內)

decoder type：audio

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"lpcm"**、**"lpcb"**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

16. mpeg_audio decoder module(在 mpeg_audio.c 內)

decoder type：audio

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"mpga"**
比對條件：

1.比對 p_dec->fmt_in.i_codec 是否符合 codec type

2.(p_dec->i_object_type == VLC_OBJECT_DECODER &&
!config_FindModule(p_this, "mpgatofixed32"))條件不成立。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

17. png decoder module(在 png.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"png "**、**"MPNG"**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

18. rawvideo decoder module(在 rawvideo.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：/* Planar YUV */：**"I444 "**、**"I422 "**、**"I420 "**、
"YV12 "、**"IYUV "**、**"I411 "**、
"I410 "、**"YUV9"**
/* Packed YUV */：**"YUV2 "**、**"UYVY "**
/* RGB */：**"RV32"**、**"RV24 "**、**"RV16 "**、**"RV15 "**
"yv12 "

比對條件：

1.比對 p_dec->fmt_in.i_codec 是否符合 codec type

2.(p_dec->fmt_in.video.i_width <= 0 ||
p_dec->fmt_in.video.i_height<= 0)條件不成立。

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

19. sdl_image decoder module(在 sdl_image.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"tga "**、**"bmp "**、**"pnm "**、**"xpm "**、**"xcf "**、**"pcx "**、**"gif "**、**"jpeg"**、
"tiff"、**"lbm "**、**"png "**

比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

20. speex decoder module(在 speex.c 內)

decoder type：audio

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"spx "**

比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

21. subsdec decoder module(在 subsdec.c 內)

decoder type：subtitle

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"subt"、"ssa "**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

22. svcdsub decoder module(在 svcdsub.c 內)

decoder type：subtitle

起始函數：static int DecoderOpen(vlc_object_t *p_this)

codec type：**"ogt "**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

23. telx decoder module(在 telx.c 內)

decoder type：subtitle

起始函數：static int Open(vlc_object_t *p_this)

codec type：**"telx"**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

24. theora decoder module(在 theora.c 內)

decoder type：video

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type：**"theo"**
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

25. vorbis decoder module(在 vorbis.c 內)

decoder type：audio

起始函數：static int OpenDecoder(vlc_object_t *p_this)

codec type："**vorb**"
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

26. spudec decoder module(在 spudec.c 內)

decoder type：subtitle

起始函數：static int DecoderOpen(vlc_object_t *p_this)

codec type："**spu** "、"**spub**"
比對條件：比對 p_dec->fmt_in.i_codec 是否符合 codec type

初始化條件：必須有足夠的記憶體空間來做動態記憶體配置。

第五章 實驗結果

針對 MPEG-2 PS 影音格式的檔案，VLC 是使用 ps module 來進行解多工的程序。所以在本章我們將實際的開啟一個 MPEG-2 PS 影音格式的電腦檔案 test.mpg，說明 ps module 內解多工函數的基本運作程序，並呈現進行解多工程序時各部運作的實驗數據。

5.1 ps module 解多工函數說明

一個 MPEG-2 PS 影音格式的電腦檔案包含了視訊流以及音訊流，整個 MPEG-2 PS 的結構可以視為三個階層：第一層為 pack header，第二層為 system header，第三層為 PES，每一層都是由特殊的起始碼所開始的。而這些視訊位元流以及音訊位元流都是存在 PES packet 中，因此解多工函數的作用就是要分離出視訊位元流以及音訊位元流，然後由解碼函數來對這些視訊位元流以及音訊位元流進行解碼。

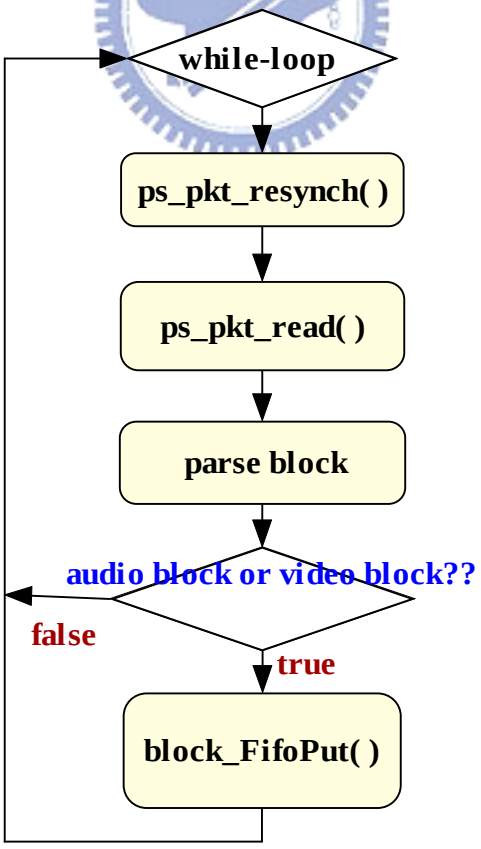

圖 5.1：ps module 解多工函數處理程序

上圖 5.1 是 ps module 內解多工函數的運作流程。ps_pkt_resynch()函數功用主要是從儲存影音資料的 data buffer 先尋找出 MPEG-2 PS 中所定義的 start code，而 start code 範圍是從 0x000001b9 到 0x000001ff，每個 start coder 是代表不同的意義。例如 pack_start_code：0x000001ba 是 pack header 的起始碼；而 system_header_start_code：0x000001bb 則是 system header 的起始碼。因此尋找 start code 的方式是從 data buffer 一開始的連續四個位元組來判斷，第一到第三的位元組一定要為 0x000001，而第四個位元組則必須為大於 0xb9。找到 start code 之後，接下來的 ps_pkt_read()函數就會依照所找到 start code 先計算出要讀取多少個位元組，之後再從 data buffer 讀取資料。

如下表 5.1 所示，這是 MPEG-2 PS 所定義的 pack header，我們可以看到 pack header 內前面的 field 佔了 14 個位元組，以及包含 stuffing byte 和 system header 這兩個部分，其中 pack_stuffing_length 這個 field 是定義接下來的 stuffing bytes 的長度。

總共 14 個 bytes

Syntax	No. of bits	Mnemonic
pack_header() {		
pack_start_code	**32**	**bslbf**
'01'	**2**	**bslbf**
system_clock_reference_base [32..30]	**3**	**bslbf**
marker_bit	**1**	**bslbf**
system_clock_reference_base [29..15]	**15**	**bslbf**
marker_bit	**1**	**bslbf**
system_clock_reference_base [14..0]	**15**	**bslbf**
marker_bit	**1**	**bslbf**
system_clock_reference_extension	**9**	**uimsbf**
marker_bit	**1**	**bslbf**
program_mux_rate	**22**	**uimsbf**
marker_bit	**1**	**bslbf**
marker_bit	**1**	**bslbf**
reserved	**5**	**bslbf**
pack_stuffing_length	**3**	**uimsbf**
for (i = 0; i < pack_stuffing_length; i++) {		
stuffing_byte	**8**	**bslbf**
}		
if (nextbits() == system_header_start_code) {		
system_header ()		
}		
}		

表 5.1：pack header

所以假如尋找到的 start code 為 0x000001ba(pack_start_code)，那麼就會先檢查 pack_stuffing_length 這個 field 來確認 stuffing bytes 的長度，然後再加上前面的

14 個位元組，這兩個長度相加所得的大小就是要讀取 data buffer 資料的大小。所以如果尋找到其他 start code 之後，決定要從 data buffer 中讀取多少個位元組也是利用同樣的方式來計算讀取的大小。

接下來就是從 data buffer 讀取影音資料，並將讀取的資料放在一個 block 裡面。block 是代表一個結構變數，在 block 裡面會有個 buffer 是專門存放所讀取的影音資料，所以每次從 data buffer 讀取影音資料之後，就會產生一個新的 block，並將讀取的資料存放在 block 內的 buffer，而每一個 block 所存放資料的資料型態，就是代表著一個 pack header 或 system header 或者是 PES，所以針對不同資料型態的 block，會有特定的解析函數來解析這些 block。在解析完 block 之後，又回到尋找 start code 程序的地方來開始進行前面說明的程序。

在本章一開始有提到過，視訊位元流以及音訊位元流都是存在 PES packet 中，所以在解析 block 的過程中，如果 block 的資料型態為 audio PES 或者是 video PES，那麼就會呼叫 block_FifoPut()函數，將解析完後的 audio block 或 video block 分別串聯在一個 audio 鏈結串列或 video 鏈結串列的後面，所以 audio 鏈結串列和 video 鏈結串列內 block 的串聯順序，就是解析 audio block 以及 video block 的順序。因此假設在進行影像的解碼程序時，就會先從 video 鏈結串列依序的取出一個 video block，然後再由解碼函數對這個 video block 進行解碼程序。

5.2 實驗數據

下表 5.2 的實驗數據是呈現存放影音資料 data buffer 的變化，以及所讀取的影音資料大小和資料型態為何。其中 tk->p_buffer 指標是指向存放影音資料的 data buffer 起始位址，i_off 表示為從 data buffer 讀取影音資料後的偏移量，一開始 i_off 是設定為 0，peek 指標是指向 tk->p_buffer[i_off]的位址，packet size 表示從 data buffer 讀取影音資料的大小。

進行第一次解多工程序時，由 peek 指標指向 tk->p_buffer[i_off]的位址，然後比對 peek[0-3]這四個位元組正好為 pack start code 0x000001ba，並且計算出此 pack

header 的長度為 14 個 bytes，最後再從 data buffer 中讀取 14bytes；等到進行第二次進行解多工程序時，一樣是由 peek 指向 tk->p_buffer[i_off]的位址，而此時 i_off 的值已經變成 14，然後比對 peek[0-3]這四個位元組正好為 system header 0x000001bb，並計算出此 system header 的長度為 18 個 bytes，最後再從 buffer 中讀取 18bytes。

NO.	i_off	Address of data		peek[0-3]	packet size (byte)
		&tk->p_buffer[i_off]	peek		
1	0	04E90020	04E90020	0x000001ba	14
2	14	04E9002E	04E9002E	0x000001bb	18
3	32	04E90040	04E90040	0x000001bc	24
4	56	04E90058	04E90058	0x000001e0	1288
5	1344	04E90560	04E90560	0x000001e0	37
6	1381	04E90585	04E90585	0x000001c0	327
7	1708	04E906CC	04E906CC	0x000001c0	328
8	2036	04E90814	04E90814	0x000001e0	43
9	2079	04E9083F	04E9083F	0x000001c0	327
10	2406	04E90986	04E90986	0x000001e0	43
11	2449	04E909B1	04E909B1	0x000001c0	328
12	2777	04E90AF9	04E90AF9	0x000001e0	37
13	2814	04E90B1E	04E90B1E	0x000001c0	327
…	…	…	…	…	…
34	18004	04E94674	04E94674	0x000001ba	14
35	18018	04E94682	04E94682	0x000001c0	328
36	18346	04E947CA	04E947CA	0x000001e0	1265
37	19611	04E94CBB	04E94CBB	0x000001c0	327
38	19938	04E94E02	04E94E02	0x000001e0	43
…	…	…	…	…	…

表 5.2：data buffer 的變化

下表 5.2 的實驗數據是呈現每次從 data buffer 所讀取的影音資料存放的位置。p_pkt 指標是指向一個 block 的起始位址，而 p_pkt->i_buffer 表示為存放在 block 內的資料 buffer 大小，p_pkt->p_buffer 表示讀取影音資料後所存放的起始位

址。所以我們可以很清楚的看到每一個 block 內所存放的資料大小和資料位址，以及屬於何種資料型態。

NO.	p_pkt	p_pkt-> i_buffer (byte)	p_pkt-> p_buffer	p_pkt-> p_buffer[0-3]	Data type
1	011066B8	14	01106730	0x000001ba	pack header
2	01156F28	18	01156FA0	0x000001bb	system header
3	01156F28	24	01156FA0	0x000001bc	PSM
4	0115C598	1288	0115C610	0x000001e0	Video PES
5	010EFE98	37	010EFF10	0x000001e0	Video PES
6	01159A38	327	01159AB0	0x000001c0	Audio PES
7	01158B18	328	01158B90	0x000001c0	Audio PES
8	01152128	43	011521A0	0x000001e0	Video PES
9	01158D08	327	01158D80	0x000001c0	Audio PES
10	010F0088	43	010F0100	0x000001e0	Video PES
11	01151F38	328	01151FB0	0x000001c0	Audio PES
12	011743A8	37	01174420	0x000001e0	Video PES
13	0117B708	327	0117B780	0x000001c0	Audio PES
…	…	…	…	…	…
34	011066B8	14	01106730	0x000001ba	pack header
35	0117AD80	328	0117AE00	0x000001c0	Audio PES
36	0124E6B0	1265	0124E730	0x000001e0	Video PES
37	01182E70	327	01182EF0	0x000001c0	Audio PES
38	010F0088	43	010F0100	0x000001e0	Video PES
…	…	…	…	…	…

表 5.3：block 資料狀態

下表 5.3 是呈現解析之後的 audio block 以及 video block，與上表 5.3 比較可以看到資料的大小以及存放資料的起始位址都已經改變了。因為音訊位元流及視訊位元流是存放在 PES payload，所以經過解析之後，block 內存放資料的 buffer 就只剩下 PES payload 的部份，而已經不包含 PES header，。最後會將這些 block 分別串聯成一個 audio 鏈結串列以及 video 鏈結串列。

NO.	p_pkt	p_pkt-> i_buffer	p_pkt->p_buffer	Data type

		(byte)		
1	0115C598	1269	0115C623	Video PES payload
2	010EFE98	28	010EFF19	Video PES payload
3	01159A38	313	01159ABE	Audio PES payload
4	01158B18	314	01158B9E	Audio PES payload
5	01152128	29	011521AE	Video PES payload
6	01158D08	313	01158D8E	Audio PES payload
7	010F0088	29	010F010E	Video PES payload
8	01151F38	314	01151FBE	Audio PES payload
9	011743A8	28	01174429	Video PES payload
10	0117B708	313	0117B78E	Audio PES payload
11	0117C418	314	0117C49E	Audio PES payload
12	0117C608	1130	0117C68E	Video PES payload
13	0117E038	313	0117E0BE	Audio PES payload
14	0117E228	1251	0117E2AE	Video PES payload
15	0117E7C8	314	0117E84E	Audio PES payload
…	…	…	…	…

表 5.4：解析完 block 之後的資料變化

下面表 5.4 是呈現 video 鏈結串列的情況，其中 p_pkt-> p_next 指標是指向下一個 block 的位址，而由 p_pkt->p_buffer 前面 4 個位元組皆為 0x000001b3(sequence_header_code)或 0x00000100(picture_start_code)，由 ISO/IEC 13818-2 文件可以知道圖片的訊息是包含在這些 start code 的後面，因此每一個 block 的 buffer 所存放的就是一張圖片的訊息。所以假設在進行影像的解碼程序時，就會先從這個 video 鏈結串列取出第一個 video block 並由解碼函數對這個 video block 進行解碼程序，解碼完之後，再從 video 鏈結串列取出第二個 block 來進行解碼，所以會一直不斷的依照順序取出 block 來進行解碼。

NO.	p_pkt	p_pkt-> p_next	p_pkt-> i_buffer (byte)	p_pkt-> p_buffer	p_pkt-> p_buffer[0-3]
1	0115C598	010EFE98	1269	0115C623	0x000001b3

2	010EFE98	01152128	28	010EFF19	0x00000100
3	01152128	010F0088	29	011521AE	0x00000100
4	010F0088	011743A8	29	010F010E	0x00000100
5	011743A8	0117C608	28	01174429	0x00000100
6	0117C608	0117E228	1130	0117C68E	0x00000100
7	0117E228	0117E9B8	1251	0117E2AE	0x00000100
…	…		…	…	…
17	01250E30	01251988	2721	01250EB9	0x000001b3
18	01251988	012520E8	1704	01251A0E	0x00000100
19	012520E8	01252588	1002	0125216E	0x00000100
20	01252588	01251988	3128	01252609	0x00000100
…	…		…	…	…

表 5.5：video 鏈結串列

第六章 結論

從本論文的討論中了解一個媒體播放器整個運作流程基本上包含解多工、解碼、輸出這三大部分，而這三個部分也是播放器的運作核心，並且包含著許多領域的知識。例如在進行解多工程序時，如何將影音資料解多工為 audio bistream 以及 video bistream，以及將這些 audio bistream 以及 video bistream 解碼成圖片以及聲音，這都需要各種影音的壓縮解碼技術知識。

現今的媒體播放器大都強調具有強大的解碼功能以及能夠解析各種影音檔案格式，也因為這個原因造成播放器所面臨的問題，就是如何在眾多的解多工器以及解碼器中找到合適的解多工器以及解碼器，所以每個播放器對於解多工器與解碼器的選擇都應該有一套機制存在，避免錯誤的產生。

所以從本論文所研究的 VLC 媒體播放器中發現 VLC 將許多功能模組化，並且設計了一套尋找模組的機制可以因應不同的功能找出合適的模組，也因此能夠針對不同的的影音檔案格式以及編碼格式，可以尋找出合適的解多工器模組以及解碼器模組來進行解多工以及解碼的程序。

參考文獻

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

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

[3] http://www.videolan.org/developers/vlc/doc/developer/html/manual.html

[4] http://wiki.videolan.org/VideoLAN

[5] ISO/IEC 13818-1 Generic coding of moving picture and associated audio information: system

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

[7] http://en.wikipedia.org/wiki/Vlc

[8] 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。

附錄

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.0.8.6c

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