國立交通大學

電機學院通訊與網路科技產業研發碩士班

碩 士 論 文

串流模組間的協商與傳輸機制研究

Research of negotiation and transport mechanism for streaming modulize

研 究 生：蘇琮壹

指導教授：張文鐘　教授

中 華 民 國 九 十 九 年 二 月

串流模組間的協商與傳輸機制研究
Research of negotiation and transport mechanism for streaming modulize

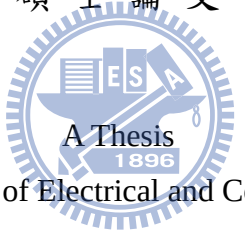
研 究 生：蘇琮壹　　Student：Tsung-Yi Su

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學

電機學院通訊與網路科技產業研發碩士班

碩 士 論 文



A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Industrial Technology R & D Master Program on

Communication Engineering

February 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年二月

串流模組間的協商與傳輸機制研究

研究生：蘇琮壹　　指導教授：張文鐘 博士

國立交通大學電機學院產業研發碩士班

摘要

在現在這個多媒體串流應用充斥的時代，對於串流應用程式的開發以及管理是一個值得去探討的問題。而我們就將針對這個問題提出一個模組化的觀念來解決，當我們將應用程式模組化後帶給我們的便利是方便管理以及可以分工開發增加開發的效率。在方便管理部份，因為每個功能函式都是一個模組所以當要更新以及修改時，只要增加新模組或修改欲修改的模組即可達到目的。而在分工開發部分，則是可以將一個大程式分成好幾個小功能模組，然後分配給多個人去完成各別模組，以增加開發效率。因此未來模組化的觀念對於應用程式將愈來愈重要。

所以本論文將藉由研究微軟 DirecShow 的模組架構，來了解一個模組內部到底如何運作，例如:模組與模組間如何協商連接以及連接之後如何互相傳遞資訊，藉由了解這些機制可以讓我們更清楚模組的運作方式，或許未來可以發展出一套更好的模組化架構。

Research of negotiation and transport mechanism for streaming modulize

Student：Tsung-Yi Su Advisor：Dr. Wen-Thong Chang

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

ABSTRACT

In the present era of multimedia streaming applications flooded, for streaming applications development and management is a worth exploring issue. We will aim at this issue to provide a modular concept with solving, after we modular application, it will give us the convenience to facilitate the management and to increase the efficiency of the development. In the part of management, each function is a function module so when you want to updates and modifications, only to add new modules or modify the module you want to modify can achieve the purpose. In the part of development, we can be a big program is divided into several small functional modules, then allocated to more than one person to complete individual modules for increasing the efficiency of the development. Therefore, in future the concept of modular applications that will become more important.

We will study Microsoft DirecShow by the module architecture to learn a module how to operate, such as how to negotiate between modules and module, and how to transmit data each other etc. By understanding these mechanisms will allow us to know more clearly the operation of the module, maybe we can develop a better architecture for modulize.

致謝

碩士生涯這兩年，最想要感謝的是我的指導教授張文鐘博士，老師對我在論文上的指導以及謹慎的態度，讓我從中學習到的不只是學術上的知識，還學習到對於做一件事情的態度。當然一定要感謝口試委員黃仲陵教授、余孝先副所長、范國清教授對論文的缺點提供意見與指導，使得論文能夠更加完善。

再者，要感謝實驗室的志偉、盛如、honda、夸克、阿民、秉謙在一起修課的日子上的互相扶持與幫助，也要感謝學弟妹明穎、耀葦、怡如、雅嵐在我趕論文與口試時的幫忙以及鼓勵，有了你們在實驗室的日子是充滿我懷念的時光。

最後我還要感謝我的父母親，他們讓我能專心讀書，不用煩惱經濟的問題，也要感謝我的女朋友在我最低潮的時候陪我度過以及加油打氣，還要感謝交大土地公的保佑，讓我可以在交大兩年都很平順，我一定會記得你們的支持的。

誌於 2010.春 新竹。交大

琮壹

目錄

圖目錄

表目錄

第一章　緒論

1.1　研究動機

在這個多媒體應用產品充斥的時代，對於一個開發多媒體應用程式的人而言，如何才能有效且快速的開發程式一直是一個重要的議題，而且科技的快速進步伴隨著的就是產品的不斷更新，於是對於應用程式的開發者而言，程式改版的速度也是非常快速，其所衍生的問題就是如何能在改版時不動用到原始的架構就能快速做到更新的目的。然而這些問題其實可以透過把一個程式模組化來解決如圖 1-1 模組化示意圖

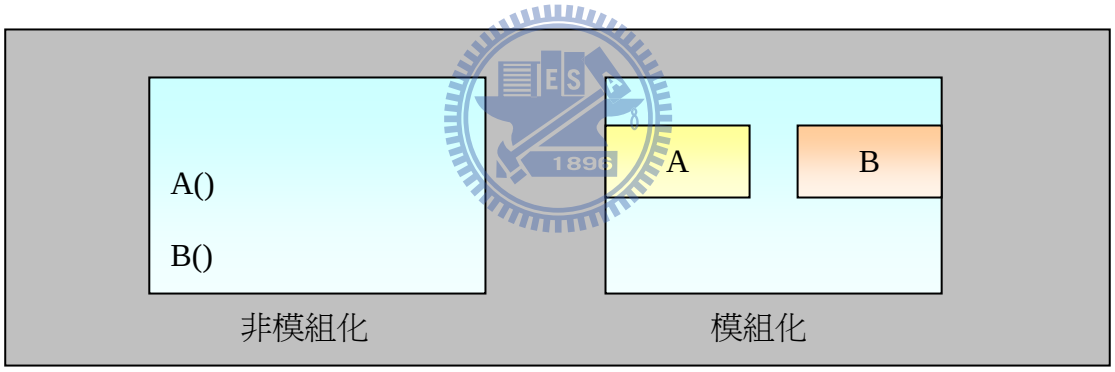


圖 1-1 模組化示意圖

一但程式模組化後可以依每個不同功能分配給不同開發者來作開發，已達到分工合作的目的進而縮短開發的時間，再者模組化後對於之後版本的更新也會變的方便許多，因為我們只需再加入新的模組就能達成更新的目的，而無須動用到原來的基本架構，但是當我們多了一道模組化步驟後，則將延伸一些問題是必須去探討的，例如模組間如何協商連接，連接後的傳輸機制等，所以我們想透過瞭解這些機制來弄清楚整個模組對我們的貢獻與便利。

1.2 研究目標

透過以上的動機後，因此有了探討程式模組化的想法，所以我們選擇微軟的 DirectShow 系統來做研究跟實驗，因為 DirectShow 本身就是一個模組化的架構，而且它對於影像處理或多媒體應用的支援是很足夠的，而這兩項性質正是我們所需要的條件，所以我們會深入研究 DirectShow 的模組與模組間的協商與傳輸機制以及這個架構下模組如何被使用等，進而透過這個架構來封裝我們的人臉偵測功能函式達到模組化的目標。而且我們還會把人臉偵測函式封裝成一般常用的模組化 DLL 的型態，比較兩種模組化的差異，讓我們更了解程式模組化的意義。

1.3 論文架構

一開始將在第二章介紹實驗會用到的微軟 DirectShow 系統架構，包括 Filter 的使用介紹、Filter 的操作原理、Filter 之間如何溝通以及狀態改變機制等。接著第三章介紹另一個實驗會用到的作業系統 Windows CE，我們將在 Win CE 系統下用一般的 DLL 形式封裝模組化，因此需對 Win CE 系統架構有些許的了解，所以會解說系統架構、系統內部四大模組的功能、以及如何使用 Platform Builder 建構出屬於自己的 Win CE 作業系統。再來第四章就會講解人臉偵測系統的演算法從如何訓練人臉圖庫到如何決定判斷人臉的臨界值等。第五章就是實作部份，介紹我們如何撰寫出一個人臉偵測功能的 Filter，以及這個 Filter 如何被應用程式開發者使用，接著介紹第二個 DLL 實驗如何封裝，以及如何應用，並且做兩種模組化的分析與比較，最後第六章做個整體結論。

第二章　模組間內部機制介紹

本章節將從 DirectShow 的大架構開始往裡層探討，首先介紹架構中的模組元件 Filter，接著解說當我們使用這種模組化架構來實現我們的功能函式時，相較於非模組化程式其所衍伸的三大問題，模組間如何協商連接、模組連接後如何傳輸資訊、以及傳輸資訊後執行狀態如何轉換，並探討 DirectShow 是使用何種機制解決這三大問題，最後再去討論 Filter 開發完成後如何讓應用程式呼叫使用的機制。

2.1　DirectShow 系統架構

DirectShow 的系統架構如圖 2-1 所示，首先系統分成兩個層級，應用層以及硬體層。以下將分別對應用層及硬體層做解說:

■　應用層

DirectShow 的核心系統位於應用層，DirectShow 利用 Filter Graph Manager 的模組來管理整個串流(Stream)的處理過程，每一個處理串流(Stream)的功能模組稱之為 Filter，各個功能的 filter 在 Filter Graph 中按照 DirectShow 制定的規則串接成為一條串流渠道，當串流(Stream)沿著這條渠道流過各個 Filter 後，即完成 Filter 功能之數據處理。

依照 Filter 所執行的功能不同，大致可分為三大類:

- **Source Filter:** 主要負責數據來源的提供，舉凡檔案文件、網路傳輸數據、視訊等，皆可做為數據傳送至下個 Filter 做處理。
- **Transform Filter:** 主要負責數據來源的處理，例如數據解碼/編碼(Decoder/coder)、影像處理、音訊處理等。
- **Render Filter:** 主要負責數據最後的顯示或存儲，如傳送至顯示卡顯示、傳送至音效卡播放、傳送至檔案儲存等。

■ 硬體層

硬體層顧名思義即為底層硬體統稱，例如:音效卡、顯示卡、MPEG2 硬體解碼器、VFW(Video for Windows)視訊設備、WDM(Windows Driver Model)設備等。此層主要任務為提供數據給 Source Filter、接收處理完之數據顯示與播放、利用硬體廠商所提供之 API 加速幫助 Transform Filter 達到數據處理之加快等功能。

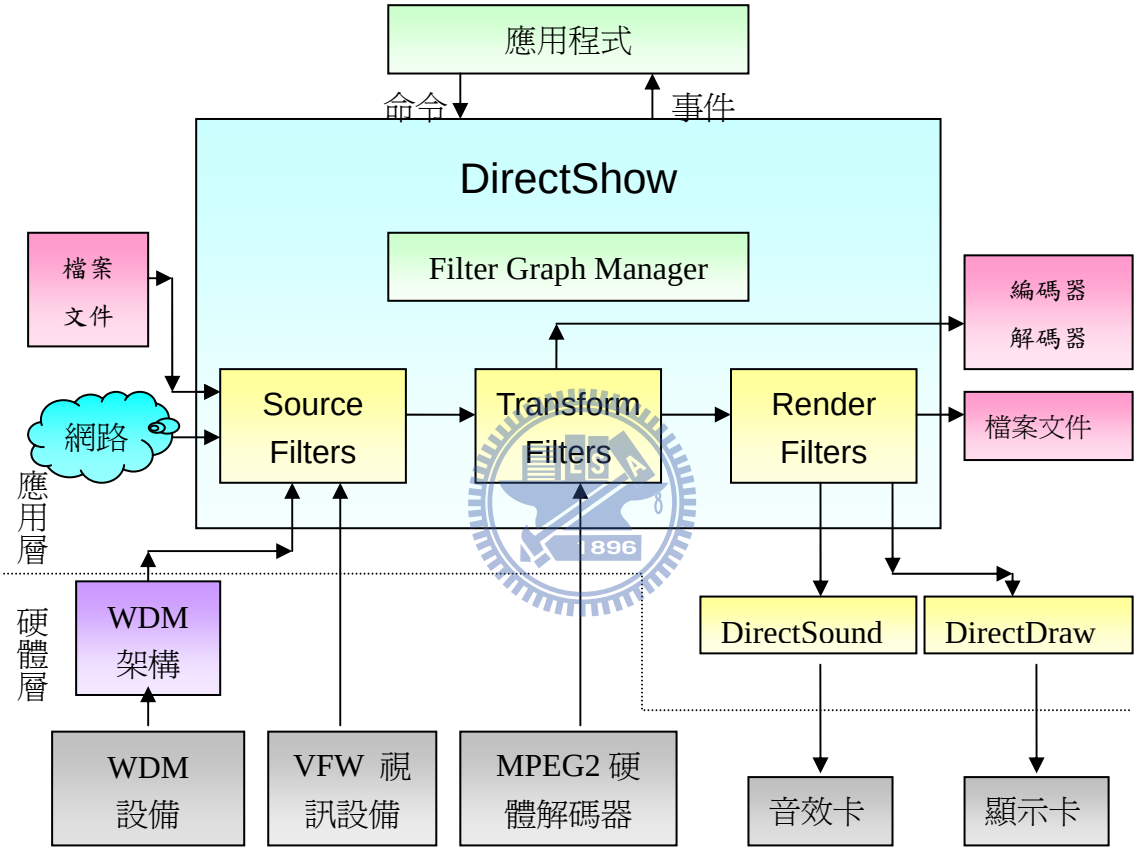


圖 2-1 DirectShow 系統架構

2.2 模組間工作原理

Filter 是 DirectShow 中最小單元，也就是每個模組我們就稱作 Filter，所以 DirectShow 其實就是由每個不同功能的 Filter 模組所組成。而在本節我們將細部討論 Filter 模組間工作原理，首先介紹何謂 Filter，接著分別介紹 DirectShow 運用何種機制解決前面所敘述的模組化三大問題。

2.2.1 模組介紹

Filter 通常由一個或多個接腳(Pin)組成，每個 Filter 透過接腳(Pin)有規則的連接。在 2.1 節有提過要構成 DirectShow 系統有三大 Filter: Source Filter、Transform Filter、Render Filter，此三種 Filter 依操作功能不同來區分或者也可以用輸入接腳(Input Pin)及輸出接腳(Output Pin)的個數來區分，例如 Source Filter 有一支輸出接腳(OutPut Pin)無輸入接腳(Input Pin)，Transform Filter 至少各有一支輸出接腳(OutPut Pin)及輸入接腳(Input Pin)，而 Render Filter 有一支輸入接腳(Input Pin)無輸出接腳(OutPut Pin)，簡圖如圖 2-2 所示。

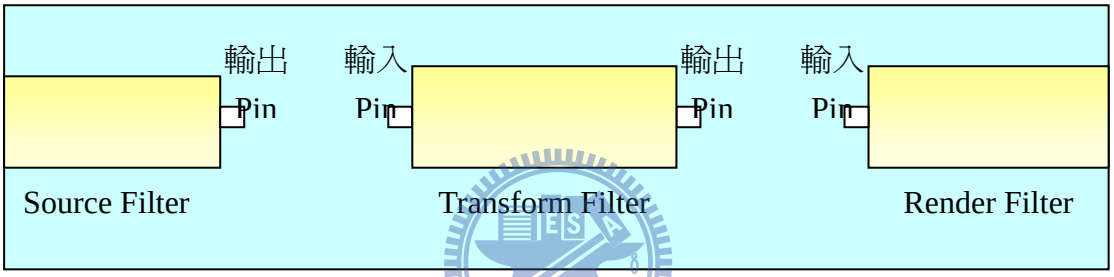


圖 2-2 各式 Filter 圖示

2.2.2 模組間協商機制

本節開始探討第一個問題 Filter 之間如何做連接，在 DirectShow 中它提供了一套協商機制來解決這個連接的問題，所謂 Filter 的連接其實就是接腳(Pin)與接腳(Pin)之間的相互認可，欲連接的兩支接腳去協商一個雙方都認可的傳輸格式，若無法取得共同認可的格式將導致連接失敗，接下來我們將說明這個協商機制如何運作。

我們可以透過 Filter Graph Manager 對 Filter 下達 Connect 指令來做連接的動作。而 DirectShow 規定兩個試圖想要做連接的 Filter 必須在同一個 Filter Graph Manager 裡面，所以連接的前置作業就是要把連接的 Filter 加到同一個 Filter Graph Manager 裡面，加入的動作可以透過函式 AddFilter 來幫我們實現。而下圖 2-3 就是 Filter 連接的流程圖，我們將透過這張圖來講解整

個 Filter 連接的細部過程。

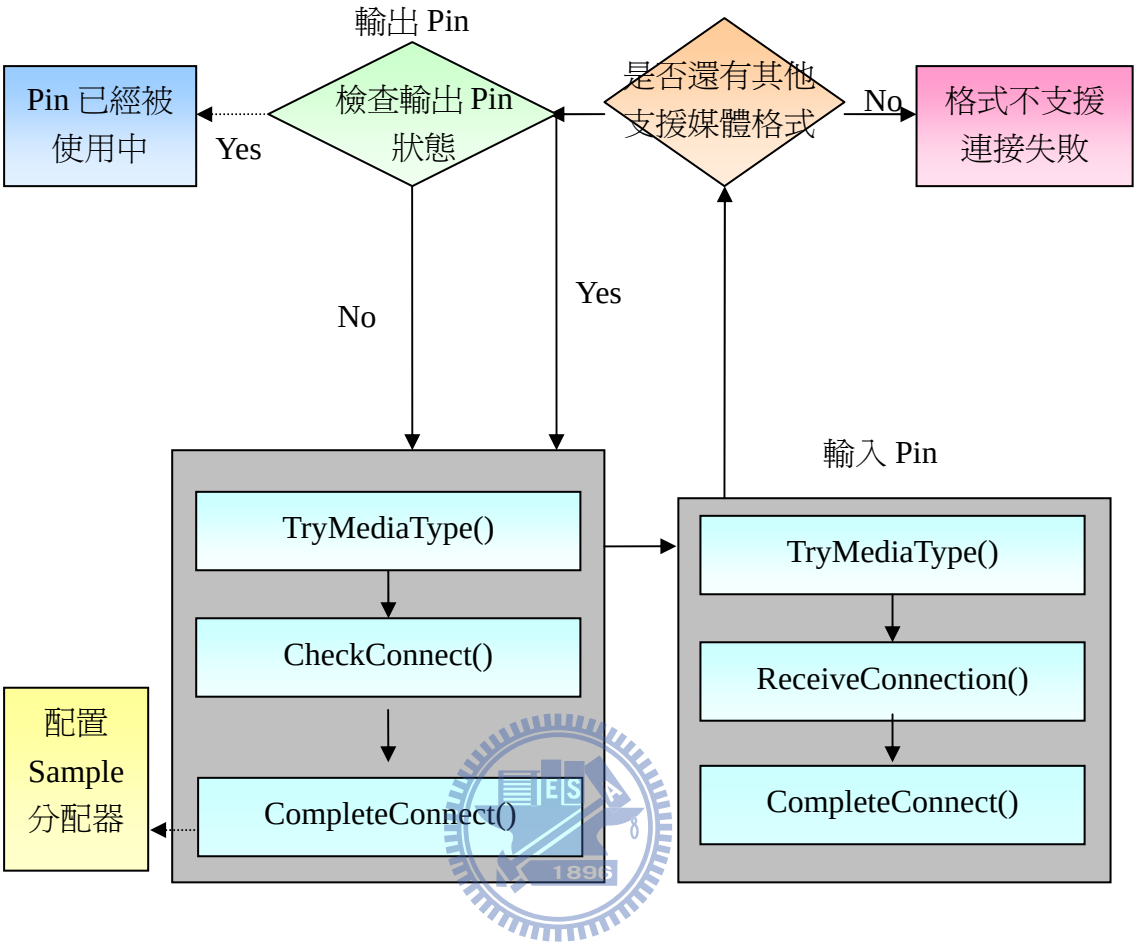


圖 2-3　Filter 連接流程

Filter 連接協商機制我們可以分成四個步驟並配合程式碼的方式來輔助說明以利更加清楚整個機制運作過程:

◆ **Step 1: 檢查 Pin 使用狀態**

應用程式透過 Filter Graph Mnanger 下達 Connect 指令之後，便啟動 Connect 函式開始執行，首先會檢查這個輸出 Pin 是否已經被連接，若已經被連接則導致失敗，若沒有被別的 Filter 連接則表示可以使用，接著檢查 Filter 是否是停止的狀態，因為要連接 Pin，Filter 必須是在停止的狀態才有辦法做連接的動作，否則一樣會導致連接失敗。其 Connect 函式 Virtual Code 如圖 2-4 所示。

```
Connect(…)
{
    if(m_Connect)     //檢查該 Pin 是否連接
    {
        printf(“This Pin is already connection”);
        return VFW_E_ALREADY_CONNECT;
    }
    if(!IsStopped)      //檢查 Filter 是否是停止的狀態
    {
        printf(“This filter is not stopped”);
        return VFW_E_NOT_STOPPED;
    }
    //開始輪詢支援的媒體類型
    AgreeMediaType();
}
```

圖 2-4　Connect 函式

◆ **Step 2: 檢查支援媒體格式**

作完狀態的基本檢查後，確認此 Pin 是可以被使用，接著 Connect 函式會呼叫 AgreeMediaType 函式開始執行，而此函式便會啟動 TryMediaTypes 函式執行，所做的動作就是透過輸出 Pin 所提供的媒體格式(Media Type)，來跟輸入 Pin 上所支援的所有媒體格式做一個比對的動作。若都沒有兩者相同的媒體類型，則此連接是不成立的，若找到一個媒體類型是雙方都支援的則兩個 Filter 就會以這個媒體格式進行連接，至此模組間的協商連接便成功。

◆ **Step 3: 協商成功**

媒體類型以及 Pin 的連接都協商成功後，接著輸入 Pin 會調用 ReceiveConnection 函式，進行一系列的內部檢查，其檢查次序內容如同輸出 Pin 檢查輸入 Pin 的動作一樣，如果檢查成功輸入 Pin 便會設置這個協商成功的媒體格式為目前所支援的媒體格式，最後輸出 Pin 及輸入 Pin 都會調用 CompleteConnect 函式來代表雙方連接協商成功，CompleteConnect 函式的

Virtual code 如圖 2-5 所示。

```
CompleteConnect(…)
{
    return DecideAllocator(…);
}
```

圖 2-5　CompleteConnect 函式

而當連接成功後代表雙方支援的媒體格式已經確定，也就是說雙方之間所要傳輸的數據格式已經確定，因此必須再協商一個 Allocator 來管理建立這個固定的數據格式之記憶體，以下我們先對 Allocator 做一個介紹

- **Sample**

  在介紹 Allocator 之前先來介紹一個名詞 Sample，它是 DirectShow 中的傳輸單元，不管是 Video、Audio 或 File 只要是在 Filter 間傳輸的我們都通稱為 Sample，而 Sample 所代表的不只是實質的數據而已，它還包含數據的一些資料(大小、媒體格式等)，因此一個 Sample 除了有一塊真正存放數據的記憶體外，它還有存放數據的資訊，而這些都由 IMediaSample 這個類別所管理，它提供一些函式來讓我們控制 Sample，例如 GetPoint 取得存放數據的記憶體位址，GetSize 取得數據大小等。

- **Allocator**

  Allocator 就是負責管理建立這些 Sample，它的功能有點類似 C++中 new memory 的指令一樣，不同的是他每次所 new 出來的是一個 Sample 的類別，而 Allocator 是由 IMemAllocator 類別所控制，它一樣提供一些函式，例如 GetBuffer 建立 Sample 類別，SetProperties 設定 Allocato 特性等。

◆ **Step 4: 協商配置 Allocator**

了解 Allocator 功能後，就可以介紹 Allocator 是如何協商配置出來，這個

協商動作是由輸出 Pin 上 CompleteConnect 函式中所衍伸的 DecideAllocator 來實現，DecideAllocator 函式 Virtual code 如圖 2-6 所示。

```
DecideAllocator(…)
{
    hr = GetAllocator(…)     //詢問輸入 Pin 是否提供 Allocator
    if(hr)                   //輸入 Pin 可提供
    {
        DecideBufferSize(…);     // 設定 Allocator 的特性
    }
    else     //輸入 Pin 不提供 Allocator，因此必須由輸出 Pin 創建
    {
        InitAlloctor(…)          //創建 Allocator
        DecideBufferSize(…)   //設定 Allocator 的特性
    }
}
```

圖 2-6 DecideAlloctor 函式

而 Allocator 並沒有規定一定要由哪一方來提供，所以這是要經過協商的，因此在 DecideAllocator 函式中首先會先去詢問輸入 Pin 是否提供 Allocator，如果提供便呼叫 DecideBufferSize 函式來設定 Allocator 的特性，如果不提供則必須由輸出 Pin 來建立 Allocator 並設定特性，當 Allocator 配置完成整個模組間的連接才算真正的完成。

2.2.3 模組間數據傳輸機制

前面已經提到 Filter 之間如何成功的連接，接著本節即將繼續探討第二個問題 Filter 之間如何傳輸數據。在 DirectShow 中有一條專門負責傳輸的線程(Thread)稱為傳輸執行緒，因此我們以下會介紹這條執行緒如何建立以及建立後如何傳輸，而圖 2-7 是一個傳輸機制的整體流程圖，我們將藉由這個圖來說明 DirectShow 透過怎樣的機制來解決數據傳輸的問題。

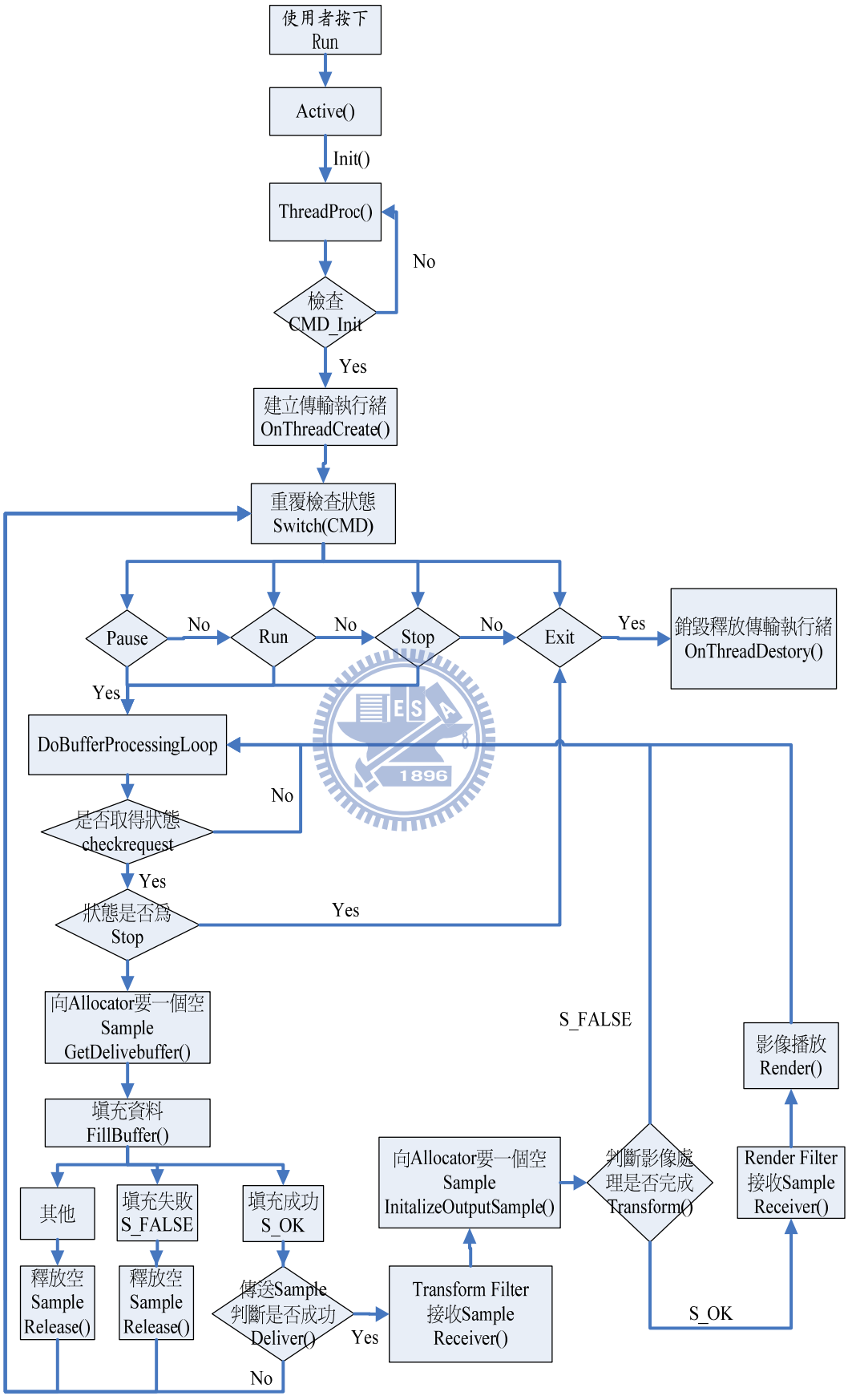

使用者按下 Run
Active()
Init()
ThreadProc()
No
檢查 CMD_Init
Yes
建立傳輸執行緒 OnThreadCreate()
重覆檢查狀態 Switch(CMD)
Pause
No
Run
No
Stop
No
Exit
Yes
銷毀釋放傳輸執行緒 OnThreadDestory()
Yes
DoBufferProcessingLoop
No
是否取得狀態 checkrequest
Yes
狀態是否為 Stop
Yes
向Allocator要一個空 Sample GetDelivebuffer()
S_FALSE
影像播放 Render()
填充資料 FillBuffer()
向Allocator要一個空 Sample InitalizeOutputSample()
判斷影像處理是否完成 Transform()
Render Filter 接收Sample Receiver()
其他
填充失敗 S_FALSE
填充成功 S_OK
釋放空 Sample Release()
釋放空 Sample Release()
傳送Sample 判斷是否成功 Deliver()
Yes
Transform Filter 接收Sample Receiver()
S_OK
No

圖 2-7　傳輸機制流程圖

1. 當使用者按下 Run 後，便將 Run 命令傳送給 Filter Graph Manager。

2. Filter Graph Manager 便啟動 Source Filter 的 Active()函式開始執行，Active 函式程式碼如圖 2-8 所示。

```
HRESULT CSourceStream::Active(void) {

    CAutoLock lock(m_pFilter->pStateLock());

    HRESULT hr;

    if (m_pFilter->IsActive()) {
    return S_FALSE; // succeeded, but did not allocate resources (they already exist...)
    }

    if (!IsConnected()) {
        return NOERROR;
    }

    // Tell thread to initialize.
    hr = Init();
    if (FAILED(hr))
    return hr;

    return Pause();
}
```

圖 2-8　Active 函式程式碼

我們可以看到 Activec 函式會先把目前狀態鎖住，接著檢查 Source Filter 是否為 Active 狀態，以及檢查 Source Filter 是否連接，必須是在 Active 狀態以及 Filter 是連接的狀態才可以繼續下面的動作。

3. 通過以上檢查後，Active 函式就會透過 Init()函式發出 CMD_INIT 的命令，Filter Graph Manager 收到 CMD_INIT 命令後就會啟動 ThreadProc()函式開始執行，ThreadProc 函式程式碼如圖 2-9 所示。

```
DWORD CSourceStream::ThreadProc(void) {

    HRESULT hr;  // the return code from calls
    Command com;

    do {
    com = GetRequest();
    if (com != CMD_INIT) {
        Reply((DWORD) E_UNEXPECTED);
    }
    } while (com != CMD_INIT);

    hr = OnThreadCreate(); // perform set up tasks
    if (FAILED(hr)) {
        OnThreadDestroy();
    Reply(hr);  // send failed return code from OnThreadCreate
        return 1;
    }

    Command cmd;
    do {
    cmd = GetRequest();
```

```
switch (cmd) {
case CMD_EXIT:
    Reply(NOERROR);
    break;

case CMD_RUN:
    DbgLog((LOG_ERROR, 1, TEXT("CMD_RUN received before a CMD_PAUSE???")));
    // !!! fall through???

case CMD_PAUSE:
    Reply(NOERROR);
    DoBufferProcessingLoop();
    break;

case CMD_STOP:
    Reply(NOERROR);
    break;

default:
    DbgLog((LOG_ERROR, 1, TEXT("Unknown command %d received!"), cmd));
    Reply((DWORD) E_NOTIMPL);
    break;
}
} while (cmd != CMD_EXIT);

hr = OnThreadDestroy(); // tidy up.
if (FAILED(hr)) {
    return 1;
}
return 0;
```

圖 2-9　ThreadProc 函式程式碼

4. ThreadProc()函式先確定目前確實是 CMD_INIT 命令後，才會透過 OnThreadCreate()函式建立一條傳輸執行緒，當傳輸執行緒建立起來後，接著便進入一個重覆檢查狀態的 while 迴圈，迴圈中將檢查四種狀態，分別對應的動作如下

   - Pause :啟動 DoBufferProcessingLoop 函式開始執行
   - Run :觸發 Pause 狀態(因為 DirectShow 規定，要轉換到 Run 或 Stop 都必須先經過 Pause 的狀態)
   - Stop :觸發 Pause 狀態
   - Exit :啟動 OnThreadDestory()函式執行銷毀釋放傳輸執行緒的動作

5. 因此除了 Exit 狀態外，其他狀態其實都會啟動 DoBufferProcessingLoop 函式，其函式程式碼如圖 2-10 所示，而這函式一開始會先透過 OnthreadStartPlay 函式啟動傳輸執行緒準備做傳輸的動作，接著會檢查是否有收到狀態的命令，有接收到才繼續檢查目前狀態是否為 Stop 狀態，若是 Stop 就發出 Exit 指令銷毀釋放執行緒，若不是 Stop 就透過 GetDeliverBuffer()函式向 Allocator 要求一個空的 Sample，並傳給 Source Filter 的 Fillbuffer()函式做一個填充資料的動作，接著 FillBuffer()函式將回傳填充的狀況，當狀況為填充失敗或其他時將把要來的空 Sample 釋放

掉，並回到 ThreadProc 函式中的重覆檢查狀態的樣子，當狀況為填充成功時就會透過 Deliver()函式來做傳送的動作，如果傳送成功 Transform Filter 的 Receiver()函式就會接收到 Sample，反之則一樣回到 ThreadProc 函式中的重覆檢查狀態的樣子。

```
HRESULT CSourceStream::DoBufferProcessingLoop(void) {

    Command com;
    OnThreadStartPlay();
    do {
    while (!CheckRequest(&com)) {

        IMediaSample *pSample;

        HRESULT hr = GetDeliveryBuffer(&pSample,NULL,NULL,0);
        hr = FillBuffer(pSample);

        if (hr == S_OK) {
        hr = Deliver(pSample);
                pSample->Release();
                if(hr != S_OK)
                {
                  return S_OK;
                }

        } else if (hr == S_FALSE) {
        pSample->Release();
        DeliverEndOfStream();
        return S_OK;
        } else {
                // derived class encountered an error
                pSample->Release();
                DeliverEndOfStream();
                m_pFilter->NotifyEvent(EC_ERRORABORT, hr, 0);
                return hr;}
    }
}
```

圖 2-10　DoBufferProcessingLoop 函式程式碼

6. 當 Transform Filter 透過 Receiver()函式接收到 Sample 後就會先呼叫 InitalizeOutputSample()函式向 Allocator 要求一個空的 Sample 以便作為傳送到下一級的準備，接著便把接收到的 Sample 以及要求的空的 Sample 傳給本身的 Transform()函式做影像處理等動作。
7. Transform()函式做完處理後會把處理完的資料直接填充到準備好的那個空的 Sample 中，填充成功便回傳 S_OK，反之則回傳 S_FALSE 並且回到 DoBufferProcessingLoop()函式。
8. 若收到 S_OK 代表填充成功便會呼叫 Render Filter 的 Receiver()函式來做一個接收的動作，當 Render Filter 接收到 Sample 就呼叫 Render()函式做播放的動作，撥放完就釋放 Sample，接著就又回到 DoBufferProcessingLoop()函式做下一回合的傳輸。

2.2.4 模組間狀態轉換機制

當討論完連接與傳輸的問題之後，接下來本節將繼續探討第三個問題就是 Filter 的狀態如何做轉換才不會有傳輸執行緒(Thread)鎖死的問題產生。所以我們以下將解說狀態轉換機制。

狀態轉換機制

上一節我們已經講解傳輸執行緒如何被建立起來，接下來我們再來探討當 Filter 狀態轉換的時候，它是透過怎樣的機制，才比較不會使傳輸執行緒發生錯亂死結的情形產生。首先我們來看 Filter 三種狀態停止、暫停、開啟對應傳輸執行緒的動作為何

- **停止(Stopped):**讓傳輸執行緒停止，並清空執行緒內的所有殘留 Sample。
- **暫停(Pause):**讓傳輸執行緒暫時鎖住(Lock)，讓 Sample 無法傳送。
- **開啟(Running):**打開鎖住的傳輸執行緒(UnLock)，讓 Sample 繼續傳送。

其中暫停是一個轉換狀態的起始點，不管要開啟(Running)或停止(Stopped)都會先做暫停的動作，目的是要讓執行緒先暫時鎖住不要再有 Sample 傳送，不然轉換狀態的時候如果執行緒內還有 Sample 在傳送的話可能會發生 Sample 丟失的情形。而且 DirectShow 發展了一套由後往前的回朔機制來解決傳輸執行緒容易錯亂鎖死的問題。回朔機制就是當狀態要轉換時都先由最後一級的 Render Filter 開始轉換起，確保最後一級 Filter 已經轉換完成才叫上一級 Filter 作轉換，這樣做的目的可以避免前面都已經轉換成停止了後面卻還沒轉換還在要數據導致要不到數據而當掉的情形產生。

接著我們透過打開應用程式後要轉換到開啟(Running)的例子來了解各 Filter 之間的狀態如何變化，首先要轉換到 Run 狀態必須先通過 Pause 狀態，因此 Source Filter 將第一個 Sample 推給下一級的 Transform Filter 此時 Source Filter 是暫停狀態，而 Transform Filter 接到第一個 Sample 後對它做完處理，再送給下一級的 Render Filter 此時 Transform Filter 也是暫停狀態，

而 Render Filter 收到第一個 Sample 把它顯示出來，此時 Render Filter 也是暫停狀態，而當使用者下達 Play 的指令後，即表示狀態要轉換成開啟(Running)，需先將 Render Filter 轉換到開啟狀態，把鎖住的傳輸執行緒打開(Unlock)接著再通知上一級轉換成開啟，以此類推一級一級往上通報，等到全部 Filter 都轉換成開啟時就完成整個動作。

這裡我們再看一個 Real time Source 的應用程式打開後要轉換到開啟狀態的情形，相較於上一個例子有些許的不同，因為 Real time Source 的 Sample 是即時的過去就沒了，所以如果還是保留第一個 Sample 在畫面上，這樣難免有些奇怪，所以若是 Real time Source 的 Filter chain 它的 Pause 狀態是不要求先處理第一個 Sample 的，只需將傳輸執行緒鎖住(Lock)即可，當要轉換成開啟(Running)狀態一樣需先從 Render Filter 開始做轉換這部份就跟一般 Source 一樣，而與一般 Source 的差別只在於此時打開(Unlock)傳輸執行緒後要從第一個 Sample 開始處理。因此 Real time Source 的開啟可能會比一般 Source 來的慢一點點，因為 Real time Source 沒有預先處理第一個 Sample 的關係。

最後再舉個暫停轉換到停止的例子來說明，讓我們能更清楚整個機制的情形。

- **暫停->停止**

  一開始大家都是暫停的狀態，當接收到停止命令時，首先一樣先由 Render Filter 開始進行停止狀態轉換且釋放所有 Sample 記憶體，然後通知前一級的 Transform Filter 開始轉換，這時 Transform Filter 會調用 Inactive 函式進行釋放所有 Sample 記憶體且終止 Receive 函式中所有 Sample，最後通知前一級的 Source Filter 開始轉換，此時 Source Filter 一樣會清除 Sample 記憶體並且發出 Exit 命令來銷毀釋放傳輸執行緒，這時所有 Filter 便無法接收任何 Sample，至此就完成了暫停->停止的狀態轉換。

2.3 模組註冊資訊介紹

以上瞭解整個內部機制後，再來我們將來介紹模組開發完成後，如何被外部使用者使用問題，講解 Filter 如何被外部使用著使用前，先來說明一下 Filter 的註冊資訊，Filter 是基於 COM 規範下的一個元件，所以它使用到 COM 規範的一個重要的特性，就是註冊功能，而這個功能的目的就是讓系統可以去註冊表中查找我們所註冊的 Filter 並自動連接 Library 而無需使用者手動連接(這個查找連結的機制我們將在下一節作詳盡的探討)，而所謂註冊到底是寫了什麼資料到註冊表以及我們要如何去填寫這些資料? 這是本節所要去探討的問題。而在 DirectShow 中它有提供四種制式的結構幫助我們做填寫註冊資訊的動作，四種結構如表 2-1 所示

Structure 名稱	敘述
AMOVIESETUP_MEDIATYPE	Filter 支援的多媒體類型描述
AMOVIESETUP_PIN	Filter 輸出入 Pin 的描述
AMOVIESETUP_FILTER	Filter 資料描述
CFactoryTemplate	Filter 類別工廠描述

表 2-1　註冊資料結構

GUID 的產生

在開始介紹這些結構之前我們先來說明一個名詞 GUID 以及如何產生 GUID，GUID 是一個 128 位元獨一無二的號碼，而 DirectShow 就是透過這個號碼來查找在註冊表中所對應唯一的 Filter 模組，而這個號碼會在我們設定 Filter 註冊資訊的時候會使用到，因此我們必須先介紹這個名詞以及如何產生這個號碼。因為它獨一無二的特性所以無法由使用者自行亂給，不然可能會發生重覆的問題，因而導致一個號碼對應兩個模組而產生錯亂，因此系統就有提供一個 GUID 的產生器稱為 GuidGen.exe 給有需要用到 GUID 的開發者去使用，如圖 2-11 所示，它提供了四種形式的產生，可以依個人喜歡選擇一種形式後，按下 New GUID，它就會幫我們產生一組新的 GUID。

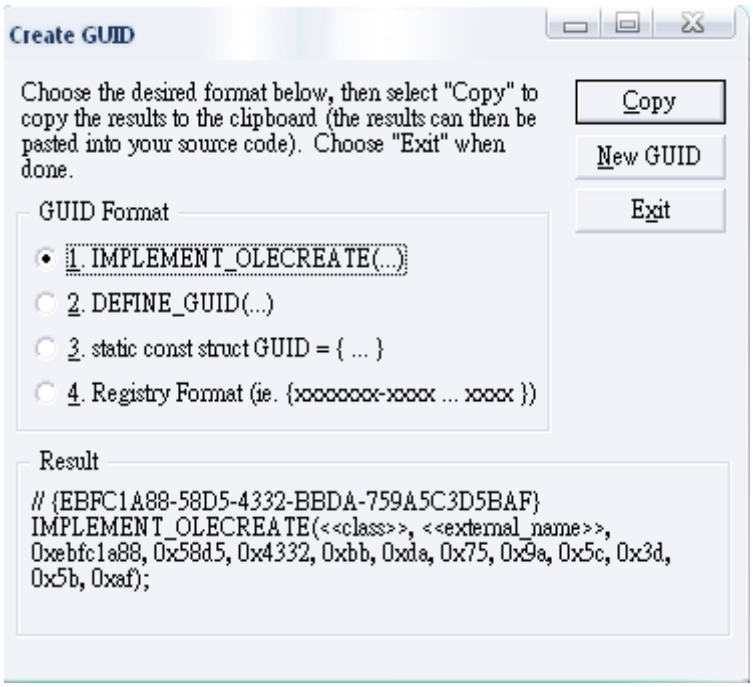


圖 2-11　GuidGen.exe

AMOVIESETUP_MEDIATYPE

以下我們將分別對四種註冊資訊結構作深入的探討，首先討論 AMOVIESETUP_MEDIATYPE ，這個結構包含 Major Type 以及 Minor Type 兩個成員，這兩種成員都由一個叫 AM_MEDIA_TYPE 的結構去描述，AM_MEDIA_TYPE 其中又有一個成員叫 formattype 的結構去描述這個多媒體格式的細節，其關係圖如圖 2-12 所示

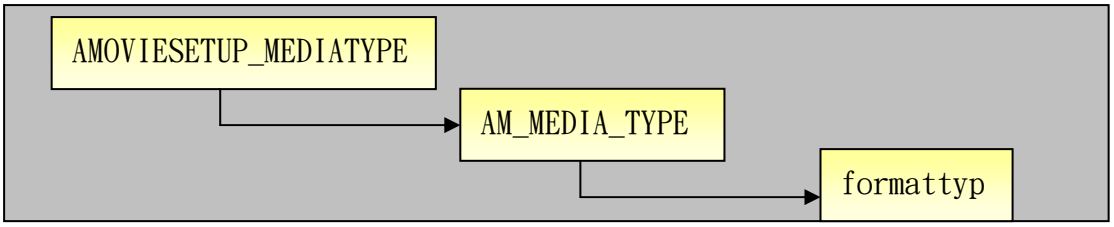


圖 2-12 AMOVIESETUP_MEDIATYPE 關係圖

首先詳細介紹 AM_MEDIA_TYPE 結構，它包含九個成員如表 2-2 所示

成員	型態
majortype	GUID

subtype	GUID
bFixedSizeSamples	BOOL
bTemporalCompression	BOOL
lSampleSize	ULONG
formattype	GUID
*pUnk	IUnKnow
cbFormat	ULONG
pbFormat	BYTE*

表 2-2　AM_MEDIA_TYPE 結構

Majortype ：媒體主類型，它的作用是定性的描述媒體類型，如視訊(MEDIATYPE_Video)、音訊(MEDIATYPE_Audio)、或串流(MEDIATYPE_Stream)等。

Subtype ：輔助說明類型，它的作用是輔助說明主類型的格式如YUV2(MEDIASUBTYPE_YUV2)、RGB24(MEDIASUBTYPE_RGB24)等。

bFixedSizeSamples :如果這個值設定 TRUE 代表每個 Sample 是固定大小，所以一般音訊都是設定 TURE，視訊沒有壓縮的設定 TRUE 有壓縮的設定 FALSE。

bTemporalCompression :如果這個值設定 TRUE 代表 Sample 是有壓縮的，反之則沒有壓縮。

lSampleSize :代表每個 Sample(Filter 的傳輸單元)的大小(單位為 Byte)。

Formattype :詳細描述媒體規格的結構，例如 MPEG1VIDEOINFO 這個結構就是要詳細描述 MPEG1 這個格式的，所以它包含了幾個成員

dwStartTimeCode: 25-Bit GOP time code

cbSequenceHeader: bSequenceHeader 的大小，單位為 Byte

bSequenceHeader: Header 陣列的起始位址，陣列大小由 cbSequenceHeader 提供。

***pUnk** :目前不使用的成員，所以都設定 NULL。

cbFormat :formattype 的大小，單位為 Byte。

pbFormat :指向 Formattype 的指標，大小由 cbFormat 提供。

了解以上 AM_MEDIA_TYPE 結構後，可以知道它是一個可變長度的結構，隨著 Formattype 結構的大小不同可能導致不同大小的 AM_MEDIA_TYPE。所以如果我們要新創一個屬於自己的媒體格式時必須重新定義這個結構的所有成員，其中較重要的有三個成員 majortype、subtype、formattype，這三個成員都是 GUID 形式都是必須在註冊表中去新增它們，使系統能看懂新創的格式。

AMOVIESETUP_PIN

這個結構主要是要去設定我們 Filter 的 Pin 資訊，它包含九個成員如表 2-3 所示

成員	型態
strName	LPWSTR
bRendered	BOOL
bOutput	BOOL
bZero	BOOL
bMany	BOOL
clsConnectsToFilter	const CLSID*
strConnectsToPin	LPWSTR
nMediaTypes	UINT
lpMediaType	AMOVIESETUP_MEDIATYPE

表 2-3　AMOVIESETUP_PIN 結構

strName: Pin 的名字，例如 Input 或 Output 等。

bRendered: 是否為 Render，設定為 TRUE 就是 Render，設定為 FALSE 就非 Render。

bOutput: 是否為 Output Pin，設定為 TRUE 就是 Output Pin，設定為 FALSE 就

是 Input Pin。

bZero: 是否為零實例，設定 TRUE 是零實例，設定 FALSE 是非零實例，例如一個影片 Filter 就可以在音訊的輸出 Pin 上設定零實例，也就代表這個 Pin 可以接受沒有連接任何裝置或輸出物而不會使整個 Filter Chain 發生問題。

bMany: 是否為多實例，設定 TRUE 是多實例，設定 FALSE 是非多實例，例如一個 Mixer 可能有多種 Input Pin 實例。

clsConnectsToFilter: 所要連接 Filter 的 CLSID。

strConnectsToPin: 該 Pin 要連接的 Pin 之類別。

nMediaTypes: 這個 Pin 所支援媒體類型的個數。

lpMediaType: 設定一個AMOVIESETUP_MEDIATYPE結構，提供媒體類型資料。

AMOVIESETUP_FILTER

這個結構主要是要讓我們設定 Filter 的資訊，它包含五個成員如表 2-4 所示

成員	型態
clsID	const CLSID *
strName	const WCHAR *
dwMerit	DWORD
nPins	UINT
lpPin	const

表 2-4 AMOVIESETUP_FILTER 結構

clsID : Filter 的 CLSID 描述，這個值可以透過 GUIDGen.exe 來獲得。

strName : Filter 的名字，顯示在 GraphEdit 上的名字。

dwMerit : Filter 被選取的優先值，這個值越大代表越優先被選取做連結，例如系統中可能有多個解碼器都可以解 MPEG，這時就要比較這個值來決定要用那

一個 Filter 來做解碼器，值越大就越先被採用，一般沒有特別要求的話是設定 MERIT_DO_NOT_USE,如果有特別要求要優先被選用可以使用 MERIT_DO_NOT_USE + n (n 為 1.2.3…)來提高自己 Filter 的優先權。

nPins : 這個 Filter 的 Pin 個數。

lpPin : 設定一個 AMOVIESETUP_PIN 的結構，提供每個 Pin 的資訊。

CFactoryTemplate

這個結構是一個類別工廠的結構，主要是當系統要建立一個 Filter 類別的時候，就可以透過使用者所給的 Filter CLSID 在 Filter 資料庫裡查找出所屬的 Filter 類別工廠結構，並藉由這個結構來建立 Filter 類別以供外部使用者使用，這個結構包含五個成員如表 2-5 所示

成員	型態
m_Name	const WCHAR *
m_ClsID	const CLSID *
m_lpfnNew	LPFNNewCOMObject
m_lpfnInit	LPFNInitRoutine
m_pAMovieSetup_Filter	const AMOVIESETUP_FILTER *

表 2-5　CFactoryTemplate 結構

m_Name : Filter的名字。

m_ClsID : Filter的CLSID。

m_lpfnNew : 函式指標指向Filter類別的CreateInstancec函式。

m_lpfnInit : 函式指標指向一個callback函式，能被入口函式呼叫，做一些判斷的功能，如DLL是否有Load完成等。

m_pAMovieSetup_Filter : 設定一個AMOVIESETUP_FILTER的結構，提供Filter所需的資訊。

2.4 模組使用機制

本節我們將探討當 Filter 開發完成後，外部使用者要如何使用這個 Filter 的問題，而因為 Filter 是基於 COM 規範下的一個產物，所以如何使用 COM 元件的過程就等於如何使用 Filter 的過程，因此本節我們採用 COM 的觀點來講述 Filter 的使用機制。

2.4.1 COM 簡介

COM(Component Object Model)是一種微軟所提供規範，它用來幫助程式設計者封裝功能函式成為模組元件，而 Filter 模組就是基於這種規範下的產物。而使用 COM 規範來封裝模組所帶來的好處是，以往如果程式設計者要呼叫別人已經開發好的函式或 API 時，需要知道別人的 Library 的位置以及要取得.h 檔、lib 檔等去做 Link Object 的動作，而如果 Library 位置改變了，就又要重新 Link 且重新 Compiler 非常麻煩，但如果封裝成 COM 的形式時，使用者只要對 COM 模組做註冊的動作並且取得模組的 GUID，即可透過 COM 規範的步驟去建立模組並呼叫模組內的函式，不需要 h 檔或 lib 檔也不需要知道 Library 的位置更不會受 Library 位置改變的影響，對開發者帶來不少便利，而這個自動連結 Lib 檔的機制我們將在以下做詳細介紹。

2.4.2 模組使用流程

一個模組要被外部使用者使用時，其所面臨的將有三大問題，

問題一:外部使用者要如何呼叫出我們的模組來使用?

問題二:模組呼叫出來以後外部使用者要如何調用模組內的介面功能函式?

問題三:使用者使用這些介面函式時，其管理機制是如何?

而這三個問題的解決方式其實就是我們本節所要講的模組使用流程的三個步驟，第一呼叫模組元件、第二模組元件介面 (Interface)調用、第三管理模組元件介面，因此我們下面就開始分別介紹這三種步驟的細部流程。

2.4.2.1 呼叫模組元件

透過解析這個步驟的原理，將幫助我們了解外部使用者如何使用 COM 模組的問題，而且也可以了解 COM 模組如何自動連結 Library，而無須使用者手動連結的機制，以下我們先從呼叫 COM 模組所要用到的一個指令開始介紹，接著才介紹使用者下達這個指令時，這個指令內部是如何幫我們找出我們的模組並建立起來。

當要呼叫 COM 模組時我們是調用 COM 函式庫內的 API CoCreateInstance() 來幫我們完成呼叫的動作，其 API 函式原型如圖 2-13，以下我們將分別解釋這個 API 所帶參數的意義，了解這些參數後可以幫助我們知道如何去呼叫自己的 Filter 模組。

```
HRESULT CoCreateInstance (
      REFCLSID        rclsid,
      LPUNKNOWN    pUnkOuter,
      DWORD           dwClsContext,
      REFIID           riid,
      LPVOID*          ppv );
```

圖 2-13 CoCreateInstance()函式原型

- **rclsid**: COM 模組 CLSID， CLSID 即是 GUID(Globally unique identifier) 的意思，所謂 GUID 是一個 128 位數字的全球唯一標示符，因此每個模組都有自己一組獨一無二的 CLSID 註冊在操作系統中，外部使用者知道此 ID 即可呼叫 COM 模組。
- **pUnkOuter**: 負責 COM 模組的外部函式，可以利用此參數對 COM 模組添加外部函式，NULL 表示不使用外部函式。
- **dwClsContext**: 表示 COM 模組所使用服務器種類，例如在 DirectShow 中使用的是 in-process DLL，代表的意思就是在本機器操作並以 DLL 的方式進行連接，所代表的參數值是 CLSCTX_INPROC_SERVER。

- **riid:** 所要求介面函式的 IID，例如可以設定 IID_ITest 來取得 ITest 的介面指標。
- **ppv:** 介面指標的位址(address)，通過此參數接收由 riid 所要求的介面函式指標。

了解以上各個參數的意義之後，我們藉由撰寫一個外部使用者透過 CoCreateInstance 呼叫一個 COM 模組的範例來說明這個 API 如何使用，其程式碼如圖 2-14 所示

```
HRESULT        hr;
IExample* pIEX;
     hr = CoCreateInstance (    CLSID_EXAMPLE,
                                NULL,
                                CLSCTX_INPROC_SERVER,
                                IID_IExample,
                                (void**) &pIEX );
     if ( SUCCEEDED ( hr ) )
          {
          // 接續動作}
     else
          {
          // 創建失敗}
```

圖 2-14　CoCreateInstance 範例程式碼

以這個範例可以看到，首先呼叫了一個叫 CLSID_EXAMPLE 的 COM 模組，不使用外部函式，使用 In-Process DLL 的服務器，並且向 COM 模組要求一個 IID_IExample 的介面，取得之介面函式指標傳送給 pIEX 接收。

而 CoCreateInstance 會有 HRESULT 的回傳值，回傳值代表之意義如表 2-6 所示。當 COM 模組使用完畢，可以透過 Release()來釋放 COM 模組，一但釋放 COM 模組將從記憶體刪除，因此也就無法在使用其介面，適當的釋放將能更有效的運用記憶體，避免記憶體被不必要的 COM 模組所佔住空間。

返回值	代表意義
S_OK	COM 元件呼叫成功
REGDB_E_CLASSNOTREG	此 COM 元件沒有註冊或找不到此註冊值而發生錯誤
CLASS_E_NOAGGREGATION	COM 元件不能被創建，要求外部函式不支持
E_NOINTERFACE	介面錯誤，COM 元件無法取得介面
E_POINTER	位址錯誤，介面位址為 NULL

表 2-6　HRESULT 回傳值代表意義

◆ **CoCreateInstance 內部流程解說**

以上知道 CoCreateInstance 這個 API 如何使用之後，接著我們要來深入探討當我們使用這個指令來呼叫 Filter 的時候，這個指令的內部是透過怎樣的機制來幫我們把 Filter 給建立起來，以及如何幫我們連接 Library。

在講解流程之前先介紹一個重要的介面，稱為 IClassFactory，這個介面對於 CoCreateInstance 來說極為重要，因為 Filter 模組在還沒有呼叫之前是不知道 Filter 類別(Class)的名稱，而不知道類別的名稱將無法取得類別裡面所實現的功能函式。因此 COM 規範針對這個問題就規定了每個 Filter 模組都必須實現一個與之相對應的類別工廠(Class Factory)，而這個類別工廠(Class Factory)的任務就是負責管理這個 Filter 模組的資訊，因此在開發 Filter 模組的過程必須把我們所開發的 Filter 類別工廠新增到 Filter 資料庫中以方便管理，而 CoCreateInstance 就是透過 IClassFactory 介面來下達指令給 Filter 資料庫以查找我們所需的 Filter 類別工廠。

了解這個介面之後，我們將透過下面圖 2-15 的流程圖分成六個步驟來說明 Filter 模組呼叫的整體過程。

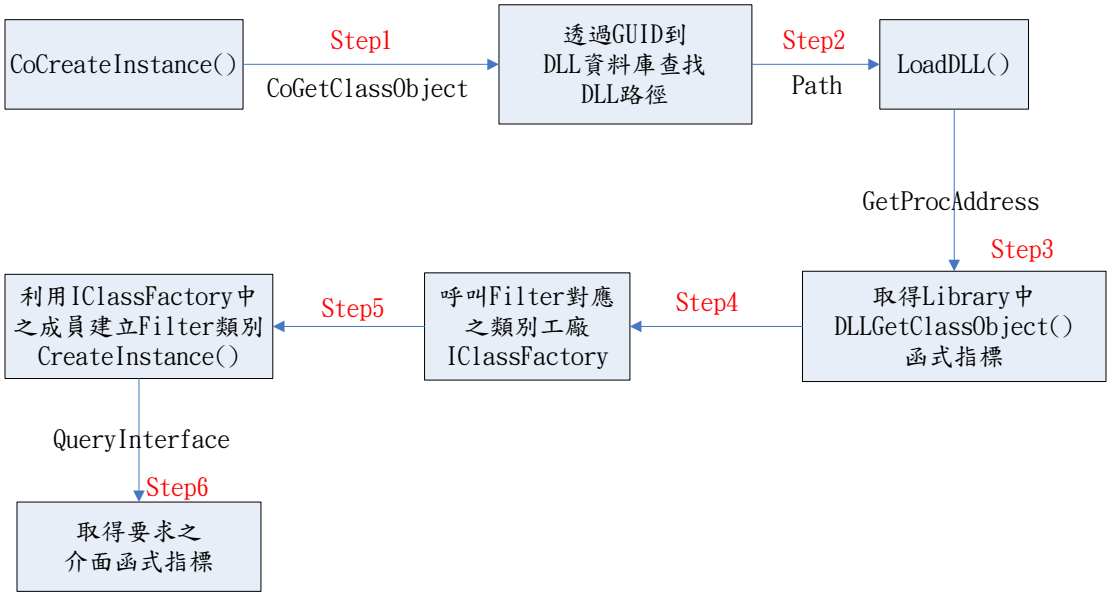


圖 2-15 模組呼叫流程

- **Step 1:** 每一個 Filter 模組都有一個專屬的 GUID，所以首先 CoCreateInstance 呼叫 CoGetClassObject 啟動執行，此函式 Virtual code 如圖 2-16 所示，此函式便會透過使用者所給之 GUID 向 DLL 資料庫查找屬於這個 GUID 的 DLL 路徑。

```
CoGetClassObject(…)
{
    RegReadPath(…);      //查詢註冊表此 GUID 的 DLL 文件路徑
    LoadLibrary(…);      //載入 DLL
    //取得 DLL 中 DllGetClassObject 函式指標
    GetPrcoAddress(…DllGetClassObject);
}
```

圖 2-16 CoGetClassObject 函式原型

- **Step 2:** 找到 DLL 路徑之後便把 DLL 路徑輸入 LoadLlibrary 函式，而此函式便會將其 Filter 模組的 DLL 載入。
- **Step 3:** DLL 載入之後，接著透過 GetProcAddress 函式來取得 DLL 中的 DllGetClassObject 的功能函式指標。

- **Step 4:**透過 Step3 所取得之 DllGetClassObject 指標就可以幫我去 Filter 資料庫查找並透過 QueryInterFace 函式呼叫出其 Filter 的類別工廠介面 IClassFactory，DllGetClassObject 函式 Virtual code 如圖 2-17 所示。

```
DllGetClassObject(…)
{
    //透過 Filter GUID 查找其類別工廠
    CFactory* pFactory = new CFactory;
    //查找 IClassFactory 介面指標
    pFactory->QueryInterFace(IID_IClassFactory,(void**)&pClassFactory);
    pFactory->Release();
}
```

圖 2-17　DllGetClassObject 函式原型

- **Step 5:**呼叫出類別工廠後，接著利用類別工廠裡面的成員函式 CreateInstance 便可以把要求的 Filter 類別呼叫出來。
- **Step 6:**Filter 類別呼叫完成後就可以透過 QueryInterFace 就可以取得所要求功能介面函式指標提供給使用者使用。

2.4.2.2　模組元件介面 (Interface)呼叫

所謂介面的意思，其實指的就是一般所說的 API 函式庫，所以呼叫模組元件介面(Interface)就是意味著呼叫模組元件裡面的 API 函式庫。每個 COM 元件都有一個基本的介面稱為 IUnKnown，IUnKnown 的功能是外部使用者與 COM 元件介面的溝通橋樑。當外部使用者想要呼叫 COM 元件裡面的某個介面時，系統便是透過 IUnKnown 這個介面去取得，同時 IUnKnown 也有管理介面整個生命週期的功能。而這個介面裡面有三種常用的 API 函式如以下所示

- AddRef()

通知 COM 元件增加介面的引用次數，如果進行了一次介面指標的拷貝，就必須調用一次這個 API 函式，計錄介面的引用次數對於管理介面是很有幫助的，可以知道介面的生命週期，例如介面被呼叫了幾次，介面是否還在使用中等等。如圖 2-18 是一個 AddRef 的函式例子

```
ULONG IUnknown::AddRef ()
{
        return ++m_interface;
}
```

圖 2-18　AddRef 函式

在這個例子中知道，當使用著呼叫一個介面叫 interface 時要同時呼叫 AddRef 這個函式來做計數的動作，以達到管理介面的工作。

- Release()

通知 COM 元件減少介面的引用次數，跟 AddRef()是同生同滅的，調用了 AddRef()幾次就要 Release()幾次，而 COM 有一種自銷毀的技術，即是引用計數器為 0 時 COM 元件就會自動調用 delete this 指令來達到自動銷毀介面的目的，以維持記憶體的高使用率。如圖 2-19 就是一個 Release 函式實現的例子。

```
ULONG IUnknown::Release ()
{
        --m_interface;
        if(m_interface==0)
        { delete this; }
        return m_intreface;
}
```

圖 2-19　Release 函式

在這個例子中實現，當調用 Release 函式將對介面計數器做減一的動作，並且判斷是否為零，若為零就銷毀介面，若不為零就返回計數器的值。

- QueryInterFace()

向 COM 元件要求一個介面的指標，當 COM 元件設計一個以上的介面時就需要用到此 API。一開始呼叫 COM 元件時用到的 CoCreateInstance()裡面的 ppv 參數只能在一開始呼叫 COM 時返回一個介面的指標，而如果這個 COM 元件還有其他的介面時，就必須透過 QueryInterFace()來查找取得其他介面指標。QueryInterFace()的函式原型如圖 2-20 所示

```
HRESULT IUnknown::QueryInterface ( REFIID iid, void** ppv )
{
        if(rrid==IID_IInterface1)
          *ppv = (IInterface1*)(this);
        else if(rrid==IID_IInterface2)
          *ppv = (IInterface2*)(this);
        Else
        { *ppv=0;
          return E_NOINTERFACE;}
          return S_OK;
}
```

圖 2-20　QueryInterFace()函式原型

- rrid: 所要求介面的 IID
- ppv: 接收所要求介面指標的位址(address)

例子中若 COM 元件還有 Interface1 或 Interface2 等其他介面的話就要透過 QueryInterface 來取的其他介面的指標位址。

2.4.2.3　管理模組元件介面

這部份其實在上一小節已經提過了，要做好介面的管理就是在適當的時間做 AddRef()以及 Release()的調用，其實管理的目的只是意味著介面從創建到銷毀的生命週期過程中確保不會出錯，而 AddRef()的功能即是創建的功能，Release()即是銷毀的功能，所以想管理好 COM 元件介面，亦即是正確適時的調用好 AddRef()和 Release()即可。

第三章　Windows CE 環境介紹

Windows CE(簡稱 Win CE)，是微軟公司專為嵌入式系統所開發的 32 位元即時作業系統，具有極佳可塑性、核心體積小等特點。Win CE 是一種高度模組化的系統，每個模組都有其專屬的功能，而系統設計者可以根據所需要的功能及設備來加入或去除模組元件，打造屬於自己的作業系統，也是因為具有這些特性，所以可以應用於各種智慧型嵌入式設備、家電、工業手持裝置等，如本論文 Win CE 實驗部分由精聯科技所提供的手持裝置就是基於 Win CE 作業系統下的一個產品。

3.1　Windows CE 系統架構

Windows CE系統是一種分層架構，如圖3-1所示，由下而上分別為硬體層(Hardware Layer)、OEM層(OEM Layer)、作業系統層(Operating System Layer)以及應用層(Application Layer)。每一層都由各種功能的模組所組成，每個模組又可細分成不同的元件所構成。這種分層架構的好處是儘量將硬體和軟體、作業系統與應用程式分離開，以利於開發人員可以細部分工，硬體開發人員只需專注硬體層的開發，而軟體開發人員也只需專注軟體層開發，這樣可以達到有效的分工，不會讓開發人員浪費太多時間在自己不熟悉的層級上。也因此Windows CE可以支援多種平台如X86、ARM、MIPS等，硬體廠商寫好硬體層(Hardware Layer)、OEM層(OEM Layer)的開發，即可讓該硬體支援Windows CE系統。

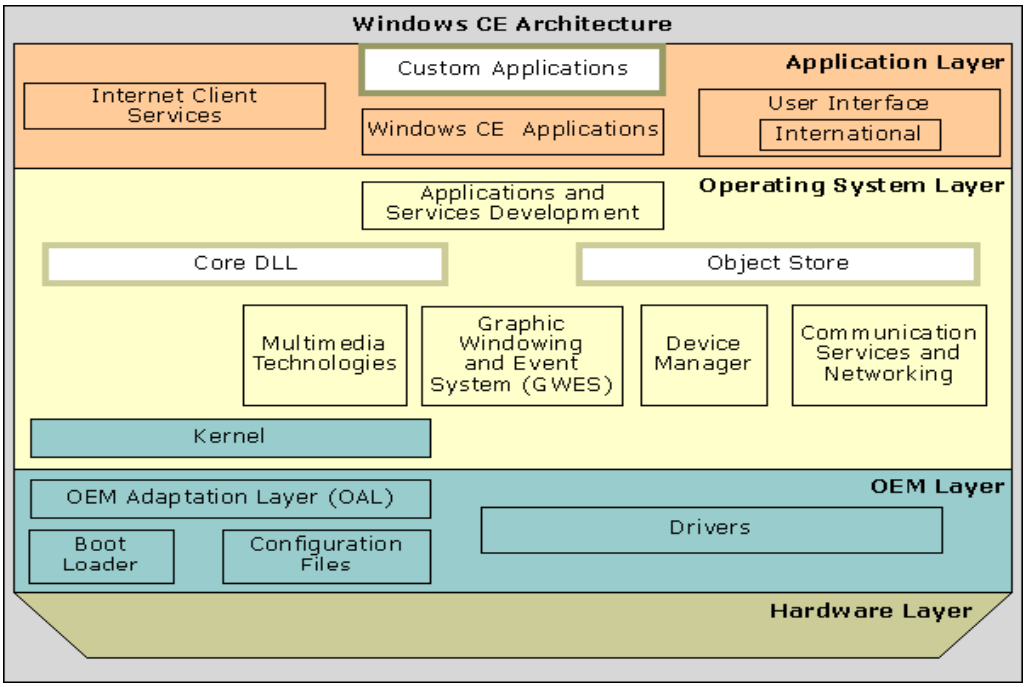


圖 3-1　Windows CE 系統架構

- **硬體層(Hardware Layer)**

  此層負責所有底層的周邊硬體設備，例如螢幕、按鍵、Camera 等硬體設施。

- **OEM 層(OEM Layer)**

  此層負責底層硬體與上層作業系統層(Operating System Layer)之間的溝通橋樑，且也負責設備開機所需的 Boot Loader 和硬體要跟系統核心(Kernel)溝通的 OAL 模組，而以上這些功能都由硬體廠商開發包覆在主機板支援套件(BSP)內。

- **作業系統層(Operating System Layer)**

  此層就包括最重要的系統核心(Kernel)、多媒體技術、通信網路服務等系統應用相關介面。

- **應用層(Application Layer)**

  此層就比較廣泛，也跟一般開發者比較相關，舉凡一些 Windows CE 的應用程式、使用者介面或網際網路 Client 端的服務都是這層所管轄的範圍。

3.1.1 BSP(Board Support Package)

BSP 稱為主機板支援套件，是由硬體廠商所開發提供的軟體套件，所做的功能即是 OEM 層(OEM Layer)的所有工作，因此套件包括啟動程式(Boot Loader)、OEM 配置程式(OAL)、標準開發版(SDB)以及相關硬體驅動程式。對於嵌入式系統來說不像 PC 具有各種工業標準，它往往是依需求功能的不同而有各自不同的硬體環境，這種多變的硬體環境決定了無法完全由作業系統來實現軟體與硬體之間的無關性，因此才需將系統中與硬體有直接關係的一層軟體獨立出來變成 BSP。所以每個開發版都有一個由硬體廠商所提供的 BSP，而此 BSP 內含各元件所做的事如表 3-1 所示

元件	說明
啟動程式(Boot loader)	系統啟動時，啟動程式會下載作業系統(OS)映像檔。
OEM 配置(OAL)	連結 kernel，管理硬體並初始化。
驅動程式	負責驅動主板或週邊的設備。
組態檔	藉由環境變數或修改*.bib 和*.reg 檔案，可將 BSP 重新組態。

表 3-1 BSP 內元件功能

BSP 的主要功能在於配置系統硬體使其工作在正常狀態，並且完成硬體與軟體之間的資料交流，為 OS 及上層應用程式提供一個與硬體無關的軟體平台。

3.1.2 Boot Loader

當系統啟動就會執行 Boot Loader，並且初始化一些設備之後，接著就會將 OS 的映像檔(image)載入到記憶體中，並且跳至記憶體的起始位置，此時的主控權便在 OS 手中。

因為嵌入式系統通常並沒有 BIOS，所以開機程序與 PC 並不一樣。開機時，boot loader 要負責初始化的工作，包括:配置記憶體、初始化所有的週邊、關閉 I/O 中斷等。而硬體開機啟動位址通常是在 0x00000000 的位置，所以為了讓

系統正常啟動，要將 boot loader 設定在 0x00000000。而 Boot loader 工作流程如圖 3-2 所示。

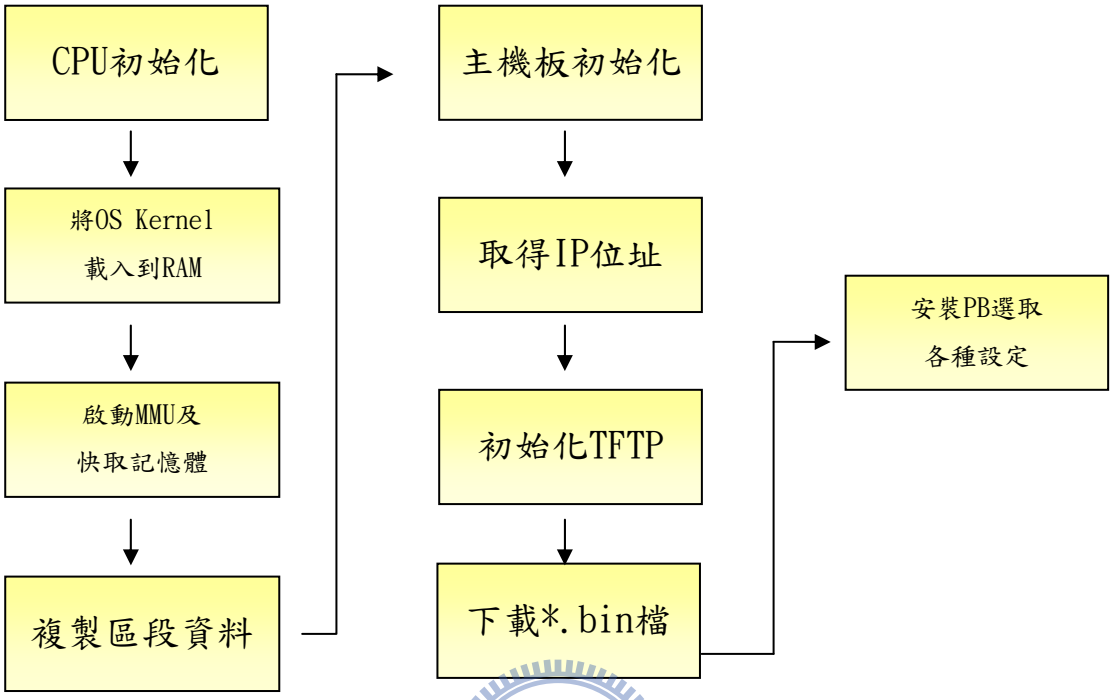


圖 3-2 Boot loader 工作流程

首先 Boot Loader 將對 CPU 做初始化動作，然後將作業系統(OS)的核心(Kernel)配置到記憶體中，接著啟動記憶體管理元件(MMU)以及快取記憶體(Flash)，記憶體管理開啟後即可複製區段的資料(如主機板所要用到的資訊等)到記憶體中，此時主機板做初始化動作周邊硬體(如網路、按鍵等)也楚於準備狀態，這時就可透過使用者輸入或預設的 IP 位址來初始化 TFTP 這套資訊傳輸套件，並藉由這套件來下載*.bin 檔，所謂*.bin 檔是 Platform Builder 根據所選的系統功能，所為我們產生的一種二進制映像檔，這種檔案格式可以使資料數量最小化並可直接下載到目標設備，不需透過輸入指令來操作。當*.bin 檔下載到目標設備後，它會依據設計者在 Platform Builder 所選取的基本功能(如 IE、Media Player、PC Card Storage 等)逐一的安裝在作業系統(OS)核心(Kernel)上，至此一個完整的作業系統即被安裝在目標設備中並可開始正常使用。

在 Windows CE 中 Boot Loader 的載入有三種方式

一、乙太網路載入

Boot Loader 名稱為 Eboot.nb0，透過 Ethernet 的方式下載並啟動。

二、序列埠載入

Boot Loader 名稱為 Sboot.nb0，透過序列埠(RS-232)的方式下載並啟動。

三、USB 載入

USB 裝置可以模擬成網路卡或序列埠以下載並啟動 Boot Loader。

3.1.3 OEM adaptation layer(OAL)

OAL 是介於 Windows CE Kernel 與硬體之間的溝通橋樑，OAL 是包含在硬體開發商所提供的 BSP 中，主要功能是中斷處理、電源管理、計時器及 Clock 等。所以當 Win CE 要從一個硬體平台移植到另一個硬體平台，只需移植 OAL，就可以達到平台的轉移。而其實 OAL 就是由一堆硬體驅動函式所組成的，如圖 3-3 是一個基本 OAL 的架構圖。

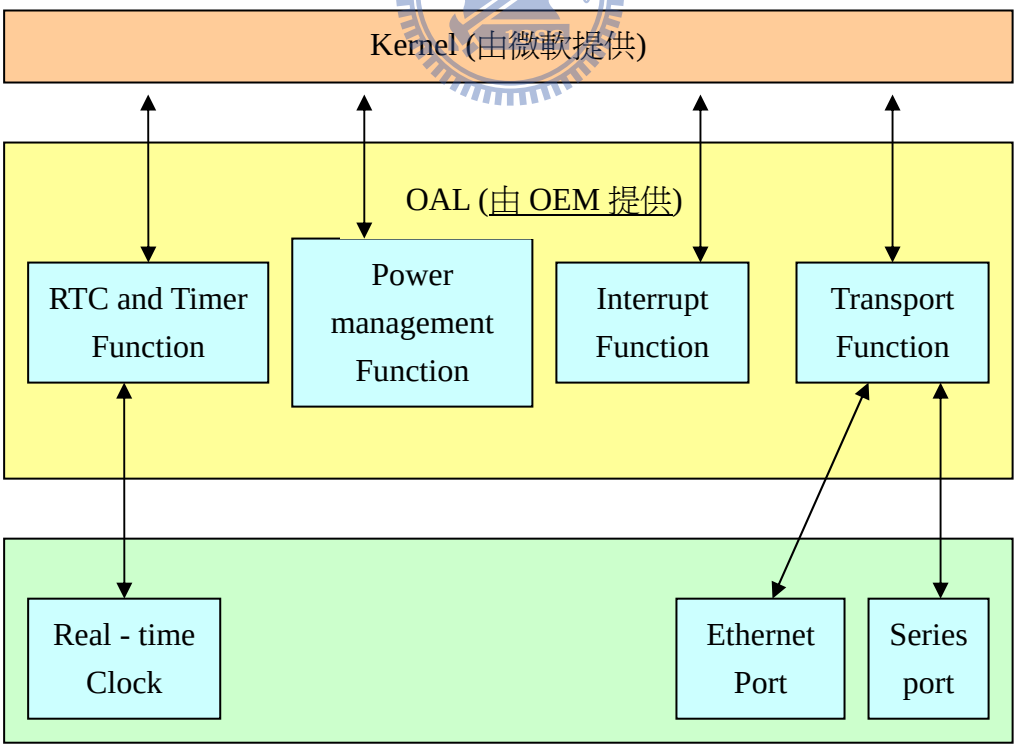


圖 3-3 OAL 架構

3.2 Windows CE 模組

Win CE 是一種高度模組化的作業系統，因此系統是由多個模組所組成，而其實模組的意思就是把相關用到的函式整理成一個函式庫的組件。Win CE 主要的模組包括核心模組(Kernel)、物件儲存模組(Storage)、圖形視窗事件模組(GWES)、通訊模組(Communication)等，而一個最小體積的 Win CE 作業系統是只有核心模組(Kernel)和物件儲存模組(Storage)兩種模組所組成。

■ **核心模組(Kernel)**

核心模組(Kernel)顧名思義就是 Win CE 系統的核心，其功能有提供 Win CE 設備記憶體管理、CPU 調度、系統異常處理以及系統內部通訊等，並為應用程式提供核心服務。而核心模組(Kernel)是透過 CoreDLL 函式庫來代表，CoreDLL 負責系統 API 的管理和安裝應用程式，管理系統應用程式中斷等，且 CoreDLL 是第一段被載入的系統共用程式碼。所有 Win CE 系統都必須包含這個模組，但這也不一定表示這個模組裡面的所有元件都是必要的，有些核心元件是可以讓系統設計者去選擇是否需要的。

■ **物件儲存模組(Storage)**

物件儲存模組(Storage)就是作業系統中的檔案系統(File System)，其作用跟硬碟比較相關，而這個模組包含了檔案系統、系統資料庫、系統註冊表等，讓系統在即使沒有主電源供應的狀態下，也能維持應用程式及檔案文件不會丟失，且也可將使用者資料和應用程式資料存入檔案或註冊表中。它提供三種檔案系統 RAM-Based 檔案系統、ROM-Based 檔案系統、FAT 檔案系統，前兩者是 Win CE 內建的檔案系統，後者是屬於可安裝性的檔案系統用以支援周邊儲存裝置如 SD 卡 ATA 裝置等。且在作業系統建立的過程中，對於這些不同的檔案物件進行管理。Win CE 的檔案路徑跟一般的作業系統有所不同，它並不使用磁碟機名稱的形式，取而代之是採用檔案系統的根目錄來定義，不同的硬碟分割區只是代表在檔案系統根目錄下的一個次目錄而已，而且也沒有所謂的目前目錄，所有的檔案都要透

過完整的路徑才有辦法取得。

■ **圖形視窗事件模組(GWES)**

Win CE 整合微軟 Win32 API、使用者介面和 GDI(Graphics Device Interface)的函式庫建構了 GWES(Graphic，Window，Events，Subsystem)模組。因此 GWES 可說是三大模組所組合而成分別是圖形(Graphic)介面、視窗(Window)管理器、事件(Event)管理器。

- **圖形介面(Graphic):** 根據設計者所需要顯示的圖形透過圖形介面所提供的 API 將其圖形繪至視窗當中。
- **視窗(Window)管理器:** 透過圖形介面可以把實現的功能繪置成小圖形(Icon)或按鍵(Button)而這些功能圖示組合起來就是一個視窗，而視窗管理器就是管理這些功能圖示，例如擺放位置，大小等，讓這些圖示組合成一個視窗。
- **事件(Event)管理器:** 當使用者透過視窗按了某一個功能圖示後，將觸發此功能的事件(Event)訊息，而事件管理器就是在把這些事件訊息送給應用程式或作業系統作處理後，再把處理完的事件訊息送回給視窗，完成一個事件的處理流程。

而 GWES 模組又由 User 及 GDI 兩個層面所組成

- User: 負責的功能是處理使用者訊息、事件或滑鼠鍵盤等輸入訊號。
- GUI: 負責的功能主要是處理螢幕或列印等輸出訊息。

所以 GWES 模組可以說是使用者、應用程式、作業系統間的溝通介面，例如它透過處理鍵盤或滑鼠訊息與使用者作溝通，再把收到的使用者訊息傳送給應用程式或作業系統達成跟應用程式與作業系統的溝通。此模組的中心是視窗，所有的應用程式都透過視窗接收作業系統的訊息，就算是沒有圖形顯示設備的系統也是如此，且它提供控制器、功能表、對話方塊、圖形顯示等設備資源，也提供 GDI 去控制文字和圖像的顯示。

■ **通訊模組(Communication)**

通訊模組是為了提供 Win CE 設備有線及無線通訊能力所產生的，可以使得 Win CE 設備跟其他設備或電腦做連接或通訊時無障礙，通訊模組支援下列基本的通訊協定:

串列埠 I/O、遠端存取服務(RAS)、網路通訊協定(TCP/IP)、區域網路(LAN)、電話技術(TAPI)、無線服務(Wireless)等。

當然除了上述這些基本的支援，使用者也可以選擇其他想要的模組元件如多媒體支援模組、Win CE 外殼模組等，且 Win CE 提供每一個元件都有一套相關的 API 函式可以使用。

3.3 系統映像(image)檔建構流程

上述介紹了許多組成系統的模組，而當這些模組在系統建構工具完成設置後，建構工具將會把所選的特性模組、系統核心及 BSP 封裝成一個系統映像檔(image)，而這個系統映像檔就包括作業系統的 Kernel、檔案系統、儲存的程式文件、系統配置檔、註冊表資料庫等。在這個映像檔的建構過大致分為四個部份，CESYSGEN、BSP、BUILDER、MAKEIMG。

■ **CESYSGEN**

這部份是透過 cebuild.bat 以及 sysgen.bat 來控制。依據設計者的配置來產生系統配置檔，而它們主要是負責產生四種檔案:

1. bib 檔:說明需要涵蓋在映像檔裡的 Win CE 檔案
2. dat 檔:檔案系統的描述
3. db 檔:Win CE 物件儲存資料庫的描述
4. reg 檔:系統註冊表描述

整個過程就是系統的前置處理，作用就是根據設計者配置的描述，將其描述轉換成系統的形式，就類似翻譯員一樣，將系統不懂的語言翻譯成系統了解的語言。

■ BSP

這部份建構工具會編譯連結 BSP 驅動程式以及 OAL 程式碼。且會使用兩種檔案來確認需要那些程式碼或程式庫檔案，分別是系統配置檔(確認需要那些特性模組)以及 MAKEFILE 檔(編譯程式碼需要那些規則)，而這裡的 MAKEFILE 和一般所看到的 MAKEFILE 不同，它是由 DIRS 和 SOURCE 檔所組成，DIRS 檔紀錄程式碼位置，SOURCE 檔紀錄編譯規則。

■ BUILDER

這部份的工作就是單純的複製，將已經編譯好的東西和系統所需的各種檔案複製到特定的目錄下。

■ MAKEIMG

這部份就是把映像檔建構出來。首先將 CESYSGEN 產生的四個系統配置檔組合成一個主要的配置檔，接著重新配置所有可執行檔的資源以適應目前的語言，最後對映像檔內容進行佈局，例如建構一個只有根目錄的檔案系統，連結所有可執行程式等，跟著就產生名為 NK.bin 的系統映像檔。

3.4 Platform Builder 介紹

以上介紹了 Win CE 的系統理論，本節將介紹如何運用 Platform Builder(以下簡稱 PB)把以上的理論實現並建造 Win CE 系統出來。PB 是設計 Win CE 作業系統的一個集成開發環境(IDE)，它包含了設計、建立、編譯、測試等等功能，而設計人員可以透過 PB 進行設置核心或選擇系統特性等工作，因此它可依設計者的需求建構出自己特有的 Win CE 作業系統出來，當編譯 Win CE 系統成功之後，即會產生一個 nk.bin 的映像檔，只要將這個映像檔下載到目標板中，就可以開始運作這個 Win CE 系統。以下開始介紹 PB 如何使用。

首先要使用 PB 建構系統之前，要先安裝硬體廠商所提供的 BSP，目的是為了讓 PB 能建構出支援此硬體設備的系統出來。安裝完 BSP 後即可打開 PB，如圖 3-4 所示。

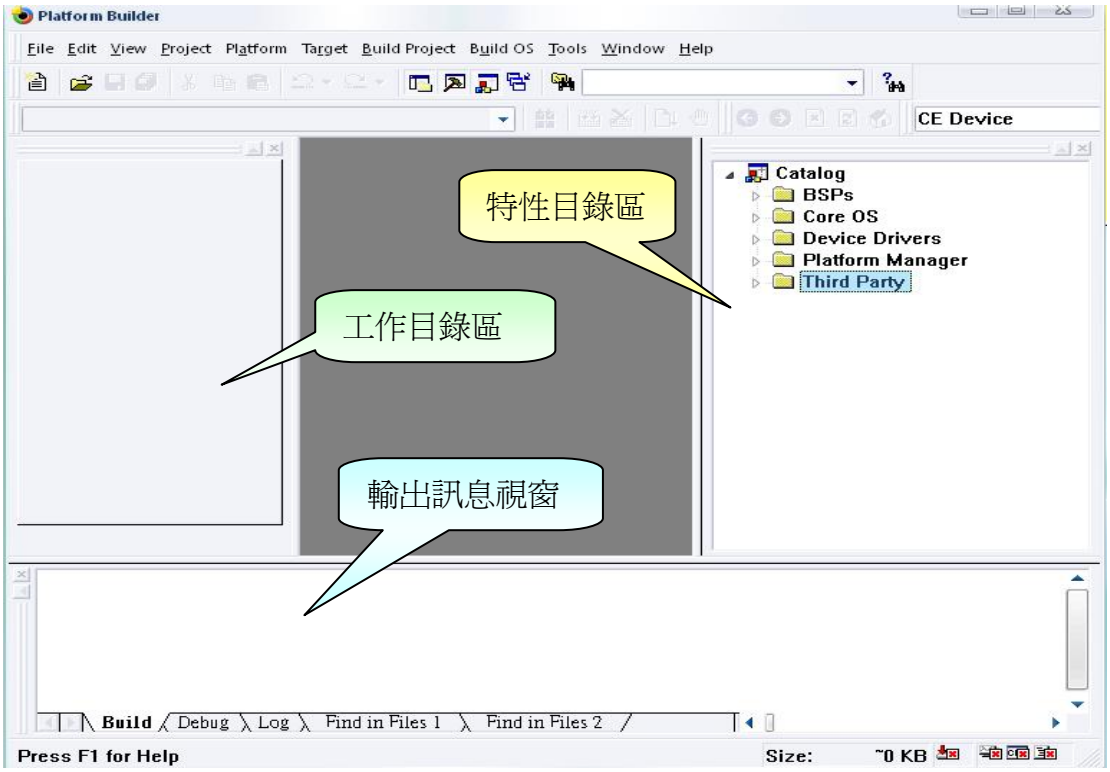


圖 3-4 Platform Builder 開啟畫面

- **特性目錄區:** Win CE 作業系統中可讓設計者去做選擇的特性都在這個目錄區中，且如果 BSP 安裝成功也可以在這個目錄區看到多出 Third Party 的目錄，裡面有所安裝的 BSP 套件供選擇，其他目錄像 BSPs 是 CE 內建的一些常見的 BSP 如 x86 或 ARMV4I 等，而 Core OS 裡面就是核心(Kernel)內可以選擇的功能特性，Device Drivers 就是一些內建的標準驅動程式如 USB、PC Card 等，Platform Manager 內就是 Embedded visual C++調試模組。
- **工作目錄區:** 當設計者建立一個新的 Win CE 作業系統時，而這個新系統的一些資料就會顯示在這個工作區內，而設計者可以將特性目錄區內的特性模組加入到這個目錄區，來為新系統增加需要的特性模組。
- **輸出訊息區:** 當系統都設定好之後按下 Build OS，此時會再這個區域輸出建立的過程是否成功或有錯誤，若有錯誤也會在這個區域顯示出來，讓使用者可以做修改。

了解軟體介面後，就可以開始建構專屬的 Win CE 作業系統。首先先選擇 New Platform，接著會有步驟來引導完成建立系統，如圖 3-5 即是讓我們選擇想要支

援的 CPU

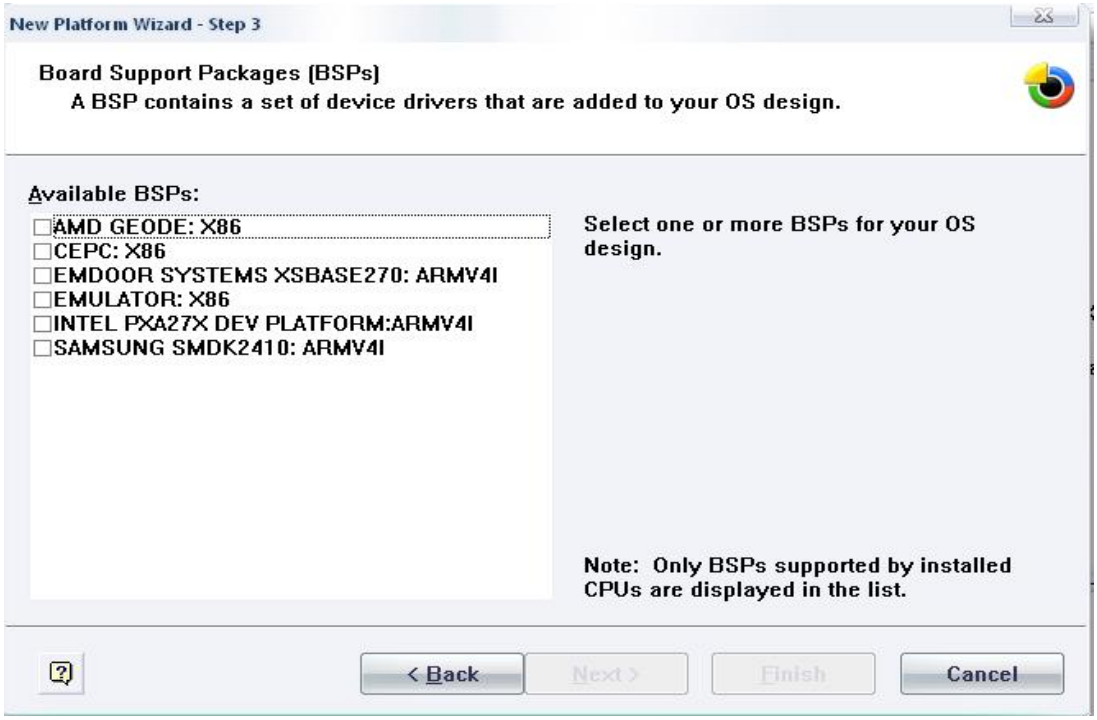


圖 3-5　支援 BSP 選擇

選擇完支援的 BSP 後，接著可以選擇這個系統是做何用途的樣板，依所選的樣板會有不同的內建特性，如圖 3-6 所示。

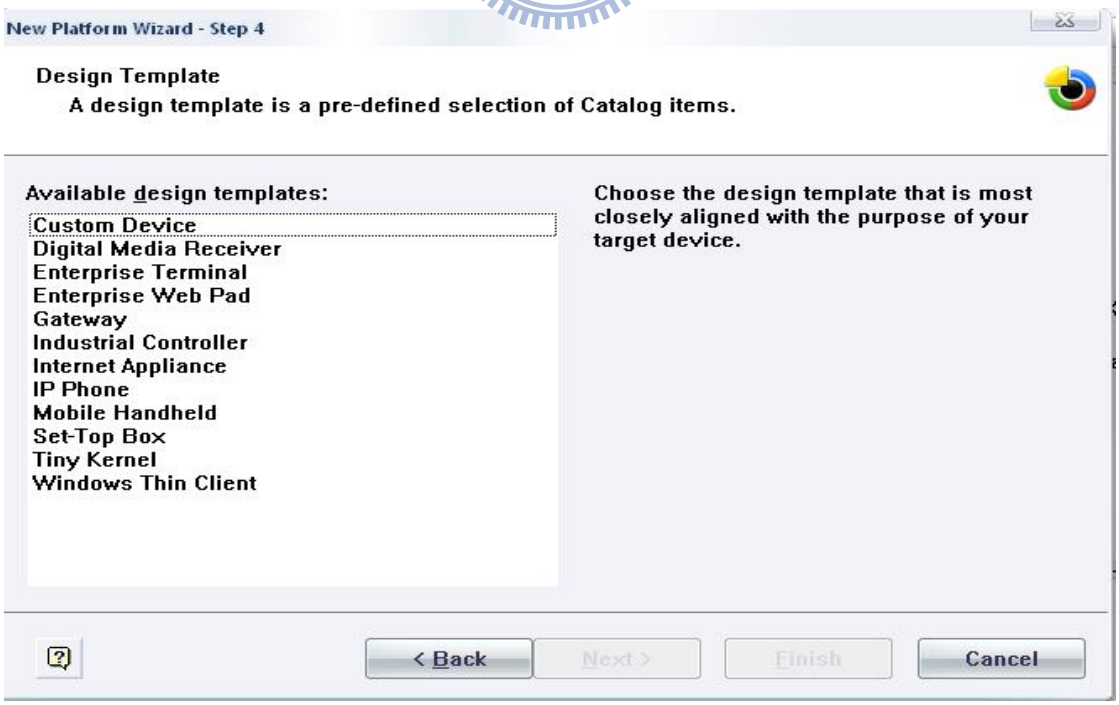


圖 3-6　設計系統樣板

接著還有通訊或網路的特性可供選擇，如圖 3-7 所示

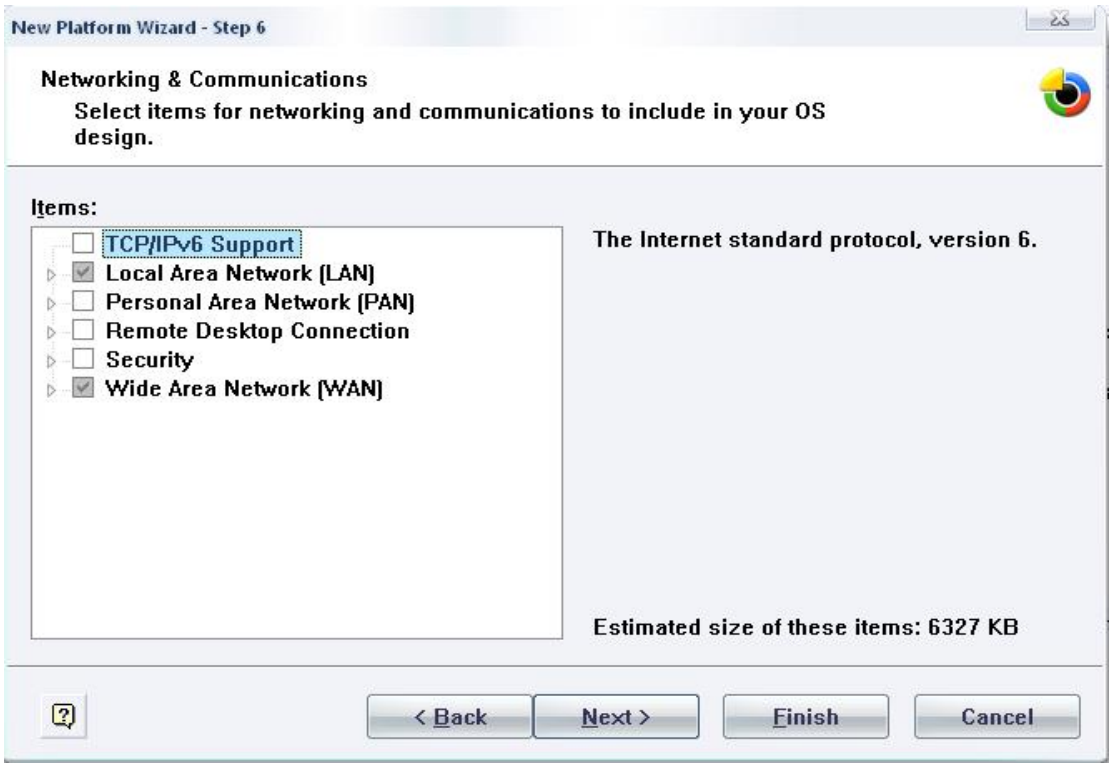


圖 3-7　網路與通訊特性

在做完一連串的特性選擇後，就會在輸出訊息區看到，所建立的系統建立成功的訊息，並且在工作目錄區也可看到選好特性系統的目錄，也可以看到這個系統將佔用多少大小的空間，如圖 3-8 所示

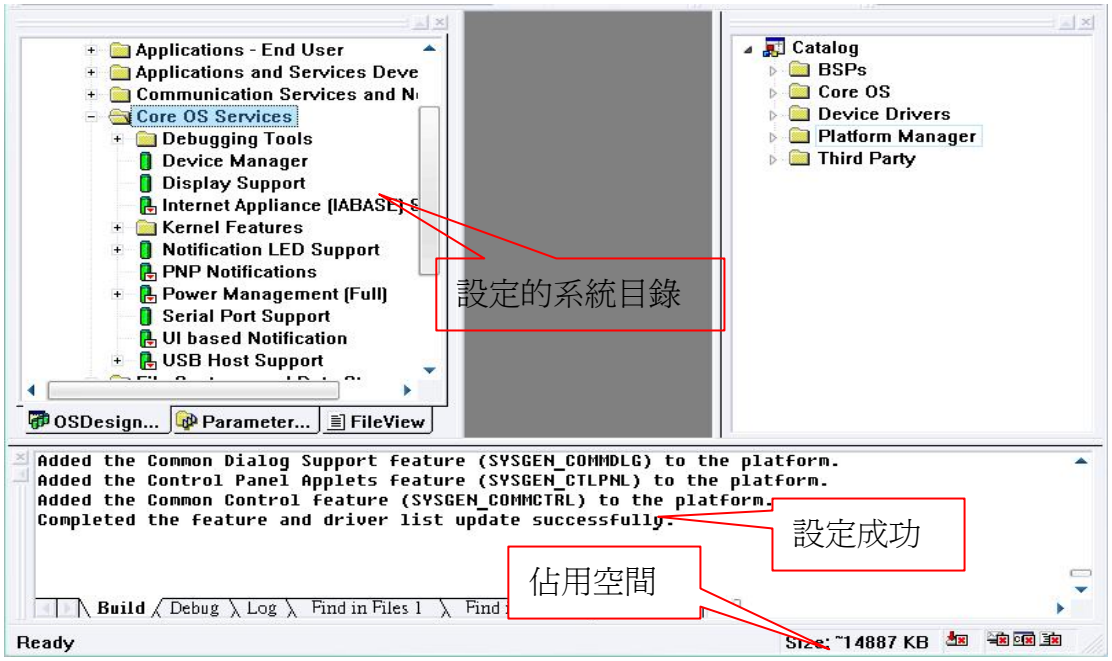


圖 3-8　設定系統完成

到此一個簡易的 Win CE 作業系統就設定好了，接著如果還有要加一些額外的特性功能的話，就可以從特性目錄區選擇要加入的特性後再選擇 Add to OS Design 就可以看到所選的特性功能被加到工作目錄區中，當全部需要的特性都選好後就可以選擇 Build OS，此時 PB 就會為這個系統做編譯並產生重要的 NK.bin 的系統映像檔，再把這個映像檔下載到目標設備上去就可以看到 Win CE 系統在目標設備上 Run 起來的畫面，至此就是一個 Win CE 系統從無到有的簡單流程。

第四章　人臉偵測介紹

人臉偵測的應用越來越多，在現今的數位相機中更是基本功能，其他像監視器也使用甚多，因此本論文實驗部分選用人臉偵測技術來模組化以達到爾後開發的便利性，而此章節將介紹本論文所用人臉偵測的原理。

4.1　人臉訓練圖庫

人臉訓練圖庫的人臉圖像來自 FERET 圖像資料庫，而非人臉圖像則由其他非人臉的部份經過剪裁成與人臉圖像相同大小所產生。為了減少人臉位置對判斷所造成的影響，因此將我們將人臉圖像都裁減成 24*24 像素大小的圖像。圖 4.1 列出經過裁減後的人臉及非人臉圖像。

(a)

(b)

圖 4-1 (a)人臉圖像.　(b)非人臉圖像

4.2　矩形特徵

人臉偵測是將偵測圖像根據矩形特徵(Rectangle Feature)計算出來的值來做是否為人臉的判斷。矩形特徵對於邊緣、線條和簡單的圖形結構相當的靈敏，所以能夠支持有效的訓練機制。圖 4-2 列出在訓練中所使用的五種不同種類特徵。

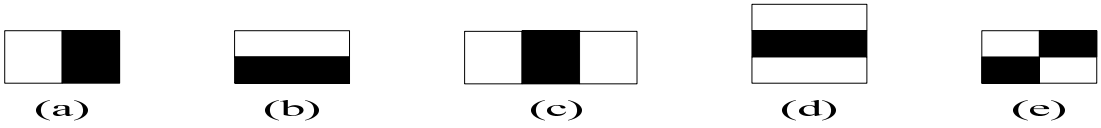


圖 4-2 五種不同種類特徵

每一張訓練圖像都是 24*24 像素點，圖 4-2(a)為長和寬的比例最小為二比一，最大的矩形特徵為 24*24 其中黑白矩形各佔 1/2 為 12*24。圖 4-2(b)則與(a)反之長和寬的比例最小為一比二，而圖 4-2(c)為長和寬的比例最小為三比一，圖 4-2(d)則與(c)反之為一比三，最後圖 4-2(e)長和寬的比例為一比一，但最小的特徵為 2*2。

而所謂矩形特徵值就是黑色矩形區域減去白色矩形區域的差值。假設我們以 r = (x, y, w, h)來代表一個矩形，其 x, y 為矩形座標 w 為寬 h 為高，則固定寬和高時可以改變 x 和 y 值來創造出所有的特徵，通常一個矩形特徵至少要用兩個 r 來代表，黑白矩形各一個，而圖 4-2(e)的矩形特徵則需要用到三個 r 來代表。圖 4-2(a)和圖 4-2(b)的特徵在單一訓練圖像中存在著 12*12*25*12 = 43200 個特徵，圖 4-2(c)和圖 4-2(d)存在著 27600 個特徵，圖 4-2(e)存在著 20736 個特徵，所以總合以上特徵，單一訓練圖像中有著 162336 種可能的特徵存在。

4.3 積分圖

然而為了能快速算出矩形特徵，我們可以使用積分的方式去代表圖像，這種方式就是積分圖。在積分圖中，(x,y)位置的值代表包含(x,y)位置左上角所有像素值的總合，其數學式可以表示如下

$$ii(x,y)=\sum_{x'\le x,y'\le y} i(x',y') \qquad ii(x,y):\text{積分圖}，i(x,y):\text{原始圖像} \qquad (1)$$

在積分圖中，只需要透過四個積分點就可以算出各個矩形的總和，以方便我們計算特徵植。以圖 4-3 為例如果我們要算出 A 矩形的總和，可以透過(4+1)-(2+3)的值即可算出，因為位置 1 的值為矩形 A 像素的總合，位置 2 的值為矩形 A+B 像素總合，位置 3 為矩形 A+C 像素總合，位置 4 為矩形 A+B+C+D 像素總合。

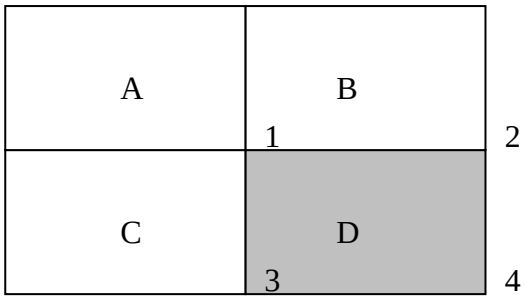


圖 4-3　矩形積分示意圖

而以上例子只是能算出一個矩形的總和，接著我們來看跟計算特徵植有關的例子，就是要如何計算兩個矩形的差值，因為我們的矩形特徵要計算黑矩形減掉白舉行的數值，所以我們必須知道如何計算差值。當要計算兩塊矩形的差值時，需要從六個點來做計算。計算兩塊矩形差值示意圖如圖 4-4 所示。

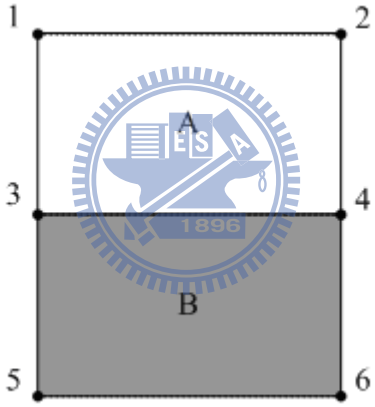


圖 4-4　計算兩塊矩形差值示意圖

矩形 A+B 像素值總合可以透過 (6+1)-(2+5)的值獲得，矩形 B 的像素值總合可以透過(6+3)-(4+5)的值求得。最後計算矩形特徵值時就是將矩形 B(黑色矩形)的總合乘以兩倍再減去矩形 A+B 的總合，這樣就是在同樣大小下兩塊矩形的差值，我們稱此值為矩形特徵值，而此種計算方式相當簡單快速。之後變成三個矩形的特徵時，計算時就要使用到九個積分點，並且黑色矩形的總合要乘以三倍。因此我們可以歸納以上解法列出一個矩形特徵值的數學式如下

$$feature = -\omega_0 \cdot \mathrm{RecSum}(r_0) + \omega_1 \cdot \mathrm{RecSum}(r_1)$$
$$r_0 = (x_0, y_0, w_0, h_0), \quad r_1 = (x_1, y_1, w_1, h_1) \qquad \omega_0 \text{ 為 1，} \omega_1 \text{ 為 2 或 3} \qquad (2)$$

如果為圖 4-2(e)的特徵時需要三個矩形來代表則數學式會變成如下

$$feature_i = -\omega_0 \cdot \mathrm{RecSum}(r_0) + \omega_1 \cdot \mathrm{RecSum}(r_1) + \omega_2 \cdot \mathrm{RecSum}(r_2)$$
$$r_0 = (x_0, y_0, w_0, h_0), \quad r_1 = (x_1, y_1, w_1, h_1), \quad r_2 = (x_2, y_2, w_2, h_2) \tag{3}$$

此時 ω_0 為 1、ω_1 和 ω_2 都為 2

積分圖在計算積分點時只是用了加法做總合，而計算特徵值時只對積分點做少許的加法或減法，計算量並不大而且非常簡單易懂，因此積分圖對於我們算矩形特徵值時是非常有幫助的。

4.4 Adaboost

在 4.2 節我們提過總共有 162336 種矩形特徵，如果我們都一個一個去計算的話，這個計算量是非常龐大的，因此我們在本節將介紹一種演算法能幫我們減少一些不需要計算的特徵值，這樣便能解省計算時間，而這種演算法稱為 Adaboost 演算法。

Adaboost 演算法的原理是可以篩選出少量的特徵組成分類器並且訓練此分類器。Adaboost 演算法在一次迴圈執行完後就會選出一個新的弱分類器，所謂的弱分類器是僅依靠單一個矩形特徵去對圖像做分類，最後由這些選出的弱分類器再去組成一個強分類器。

當我們透過 4.1 節的人臉訓練資料庫去做訓練時，首先可以將訓練圖像分為正負兩類，正類由人臉圖像所組成，負類則由非人臉圖像所組成。一個弱學習演算法能夠選出的單一個矩形特徵去區分正負類以達到最好，而它的方式就是找出錯誤最少的特徵。對每個特徵而言，如何判斷正和負是靠臨界值(Threshold)來決定，弱演算法則決定了一個最佳臨界值去組成分類方程式，這個方程式為

$$h_j(x) = \begin{cases} 1 & if \ \ p_j f_j(x) < p_j \theta_j \\ 0 & otherwise \end{cases} \tag{4}$$

j 為特徵的個數，h_j 是弱分類器，f_j 是矩形特徵值，θ_j 是臨界值，x 是 24*24 的子視窗，而 p_j 為不等式的方向，也就是當 p_j 為負號時，f_j 大於 θ_j 則 h_j 的值為一，相反的當 p_j 為正號時，f_j 大於 θ_j 則 h_j 的值為負一，因此 p_j 就是決定 h_j 為一時應該是大於或小於 θ_j 。

4.4.1 Adaboost 演算法流程

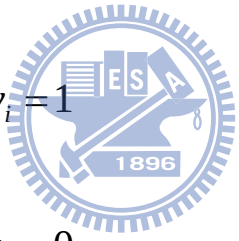
而以下我們就將對整個 Adaboost 演算法做一個介紹

- 給定 example 圖像 $(x_1, y_1), \ldots, (x_n, y_n)$

$$\text{where } y_i = \begin{cases} 1 & for\ \ face\ example \\ 0 & for\ non-face\ examples \end{cases}$$

- 初始化 weights：

$$w_{1,i} = \begin{cases} \dfrac{1}{2l} & for\ y_i = 1 \\ \dfrac{1}{2m} & for\ y_i = 0 \end{cases}$$



l 和 m 是正和負的個數總合。

- For 迴圈 $t = 1, \ldots, T$ ：

1. 正規化 weights

$$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$

2. 對每個特徵 j ，其 $\varepsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$

3. 選擇 h_t 使得 ε_t 為最低

4. 更新 weights：

$$w_{t+1,i} = \begin{cases} w_{t,i}\beta_t & \text{if example } x_i \text{ is classified correctly} \\ w_{t,i} & \text{if example } x_i \text{ is classified incorrectly} \end{cases}$$

where $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$

- 最後強分類器為：

$$h(x) = \begin{cases} 1 & \frac{\sum_{t=1}^{T}\alpha_t h_t(x)}{\sum_{t=1}^{T}\alpha_t} \geq \frac{1}{2} \\ -1 & \text{otherwise} \end{cases} \qquad \text{where } \alpha_t = \log\frac{1}{\beta_t}$$

每一次迴圈執行完就會從所有特徵中找到一個錯誤率最低的特徵，因此透過 T 次的 boosting，可以得到 T 個弱分類器，藉由這 T 個弱分類器組成強分類器用於偵測人臉。而決定是否為人臉的方法是透過若分類器來投票決定，因此每個弱分類器都有一個 α 值，其用意就是選票的意思，當然每個弱分類器的選票數量並不相同，越強的弱分類器其 α 值也就越大代表選票也就越多，當子視窗得到的選票超過 1/2 的總數時，此子視窗就會被判定為人臉，反之則為非人臉。

4.4.2 更新 Weight 的說明

假設訓練圖像的資料庫中有 n 張人臉和 m 張非人臉的圖像，首先分別給其 $1/2n$ 和 $1/2m$ 的 weight。如果不更新 weight 的情況下，只能夠找到錯誤率最低的一個特徵，可是當需要多個特徵時，就必須給每個圖像更新 weight 才能找出第二好的特徵，或找出更多有用的特徵。

雖然是錯誤率最低的特徵，但畢竟不是零所以還是有可能將人臉圖像分類錯

誤的情況發生，假設有 p 張人臉分類錯誤，這 P 張人臉的 weight 就將會被改變，用意是希望在下一次分類此圖像時可以被正確的分類，最後能將所有人臉圖像都正確分類，這就是 weight 更新的意義。

因此 weight 加大的目的在於，如果某一特徵再次分類錯誤此圖像的情形下，它的錯誤率將會很高，也就不是我們所需要的特徵。而當某一特徵能夠正確分類錯誤圖像並且它的錯誤率也是最小時，則此特徵就是我們所需要的。所以這 P 張人臉圖像在第二好的特徵之下，如果都能分類正確，那就只需要結合這兩個特徵便能將所有的人臉圖像正確分類。但如果 P 張裡面還是分類錯誤 x 張的話，我們就必須一直加大錯誤圖像的 weight，找出第三個或更多個特徵來正確分類這些人臉圖像，直到所有的人臉圖像都能分類正確為止，則強分類器就由這些所找到的特徵來組成。

4.5 Cascade 分類器

我們利用以上 adaboost 演算法選出 T 個特徵後，在每個檢測子視窗中都需要利用這 T 個矩形特徵計算其矩形特徵值，感覺有些浪費時間，因此本節將介紹 Cascade 分類器，這分類器可以幫助我們去減少計算時間。關鍵在於 Cascade 使用多個 stage 來減少特徵值的計算，當上一級 stage 認定此檢測子視窗為人臉，它才會丟給下一級 stage 做更嚴格的檢測，因此能夠預先排除大部分的非人臉圖像，並且偵測出幾乎所有的人臉圖像，而可以調整弱分類器的臨界值使得 hit rate 值達到 100%，也就是所有的人臉圖像都會被正確分類。圖 4-5 就是一個 cascade 的架構圖。

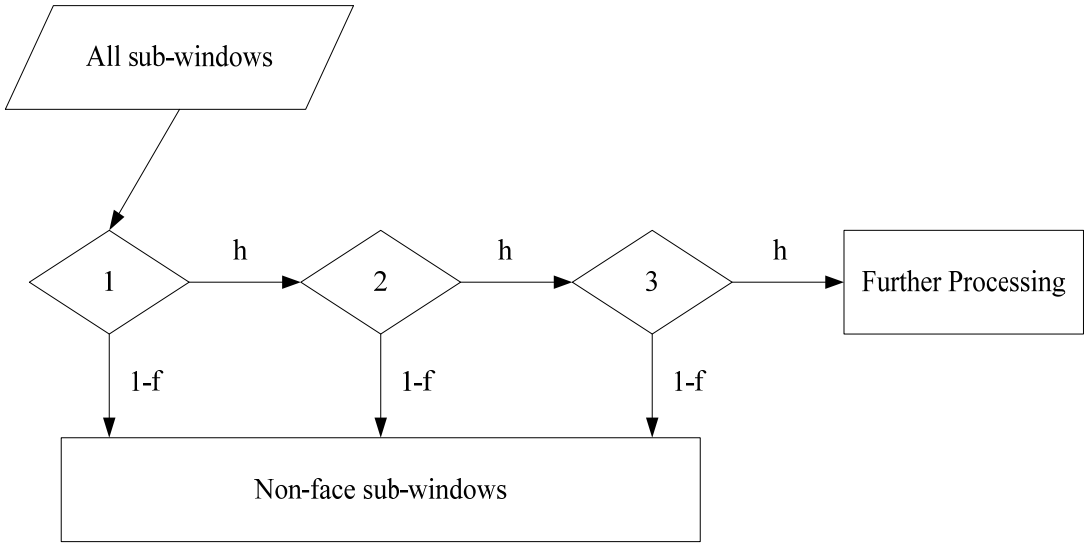


圖 4-5　Cascade 架構圖

首先第一個 stage 的分類器由全部的訓練圖像做 adaboost 演算法，接著我們可以去設定兩個參數

I.　**hit rate**：正確分類的人臉圖像數目除以人臉圖像的總數

II.　**false alarm**：錯誤分類的非人臉圖像數目除以非人臉圖像的總數

而要達到以上兩個參數之限制時所需要的特徵數量，就是組成此 stage 的特徵。一般我們會設定 hit rate 為 100%，假設第一個 stage 由 n 個特徵所組成，則這 n 個特徵能將所有人臉分類正確，不過還是會有非人臉圖像被分類錯誤的情形發生，因此設定 false alarm 想把錯誤分類降的一個定值，因為我們不要求在第一個 stage 就能完成所有分類的工作，只要能夠刪除大部分的非人臉圖像就夠了。之後觸發下一個 stage 分類器再刪除前一個 stage 未能去除的非人臉圖像。建立這個 stage 一樣透過 adaboost 選出特徵，不過訓練圖像有所不同，人臉圖像為相同的數量，非人臉圖像則為前一個 stage 未能正確分類的非人臉圖像，這樣做的目的在於這個 stage 只要專心分類前一 stage 無法正確分類的非人臉圖像即可，那些已經正確分類的非人臉圖像在前一個 stage 就已經刪除了，這就好像 adaboost 的 weight 更新是要專心分類錯誤的圖像的道理一樣，因此這個 stage 一樣會設定 hit rate 和 false alarm 去選出所需之特徵，而此時會將 false alarm

的值設的更低，因此分類的難度也會變高，會使用更多數量的特徵做分類。而如果能夠在這個 stage 就分類完成所有的圖像，那就只需要兩個 stage 即可，但是如果還是有錯誤分類發生時，就又會觸發下一個 stage 分類器的形成，直到達成正確分類所有圖像，所建立完成的 cascade 分類器，就是所需要的分類器。圖 4-6 為 cascade 分類器建立流程圖。

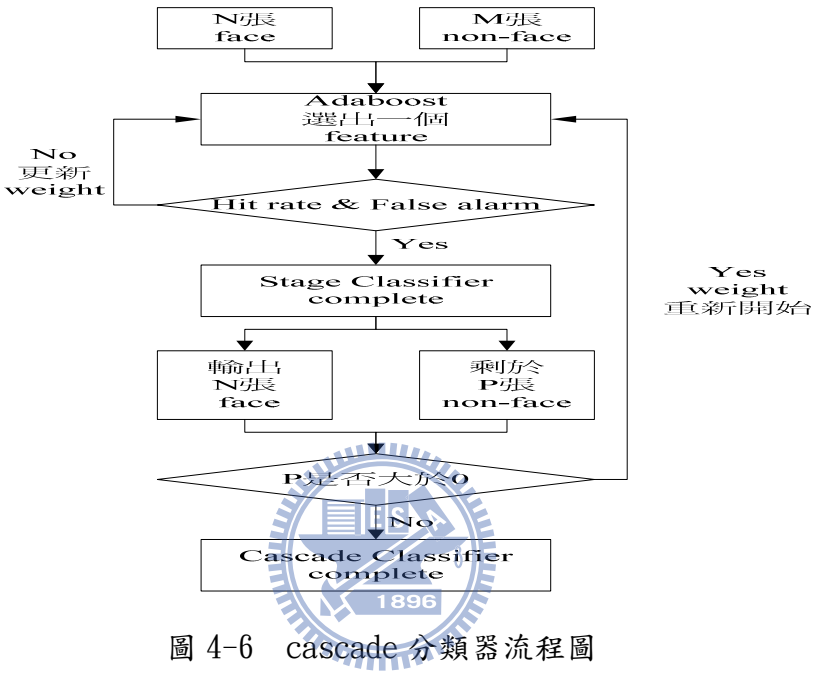


圖 4-6　cascade 分類器流程圖

然而 hit rate 要達到 100%也不是那麼簡單而且還得控制 false alarm 的值，這是需要適當的調整每一個特徵的臨界值去達到將人臉圖像當作非人臉圖像的錯誤值為零，雖然較低的臨界值可以提高偵測率達到所要的正確比例，但將伴隨著較高的將非人臉圖像當作人臉圖像的錯誤發生，因此如何調整平衡兩者之間關係，是不容易的。

一般來說，一張照片中大部分都是非人臉，人臉通常只佔據圖像的一小部分，所以訓練圖像中非人臉圖像的數量往往會遠大於人臉圖像，就是依據照片的特性所設計。cascade 的架構可以在每個 stage 刪除部分的非人臉圖像，而越前面的 stage 所刪除的會越多，當這些非人臉圖像被刪除後，接下來的 stage 就不會再對這些非人臉圖像做檢測，因此 Cascade 的架構可以節省很多的計算時間，因為每經過一個 stage 要計算的特徵就會減少。

4-6 臨界值(Threshold)的決定

每個特徵會在圖像上計算其矩形特徵值，當計算完成後都有一個臨界值去判斷此圖像是否為人臉，因此決定臨界值的好壞變得相當的重要，因為其所影響到的是人臉的判定是否正確，如果臨界值定的不好，將會造成過多的人臉或是幾乎偵測不到人臉。而本節將討論本論文使用的臨界值決策方式，其所採用的是 OpenCV 的方法，以下我們介紹這個方法的流程。

- 假設有 n 張 samples，首先選定某一個特徵並計算出 n 個矩形特徵值，再將這 n 個值所對應的 weight 值做升冪排序，並開始做 n 次迴圈的計算。

假設目前為第 p 次迴圈，則透過以下算式算出錯誤率

- **wl** :前 p 個 weight 值相加
- **wr** :剩下 n-p 個 weight 值相加
- **wyl** :前 p 個 weight 乘上其對應的 y 值相加(人臉為 1、非人臉為-1)
- **wyr** :剩下 n-p 個 weight 乘上其對應的 y 值相加
- **curleft** : wyl/wl
- **curright** : wyr/wr
- **curlerror** :前 p 個 y_k 值減掉 curleft 後平方再乘上對應的 weight 值

$$curlerror = \sum_{k=1}^{p} w_k \cdot (y_k - curleft)^2 \qquad (5)$$

- **currerror** :剩下 n-p 個 y_k 減掉 curright 後平方乘上對應的 weight 值

$$currerror = \sum_{k=p+1}^{n} w_k \cdot (y_k - curright)^2 \qquad (6)$$

- **ferror** : curlerror + currerroe

通過以上算式算出 ferror 之後，比較這個特徵值的錯誤率是否小於目前最小錯誤率 minerror，如果成立的話這個特徵值加上前一個特徵值除以二後，就

是目前找到的臨界值，並且 ferror 重新定義為最小錯誤率。跑完 n 次迴圈之後，最後找最小錯誤率的臨界值就是最佳的臨界值。圖 4-7 為 OpenCV threshold 決策的流程圖。

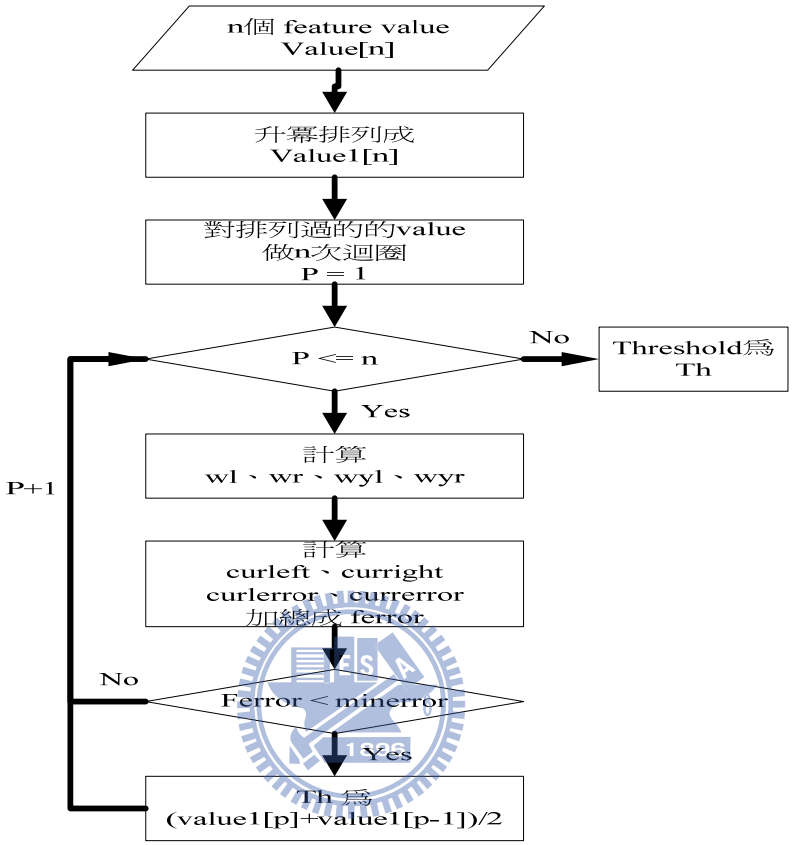


圖 4-7 threshold 決策流程

這種方法是在找出一個能夠使得 curleft 與 curright 越趨近於-1 與+1 的值，也就是能清楚的分開人臉和非人臉圖像的界線，當然不可能每次都能剛好找到是-1 和+1 的那個界線，因為兩邊都或多或少會有少數人臉和非人臉的圖像出現，所以只要找到最靠近-1 和+1 界線的值即可。而判斷 curleft 和 curright 是否趨近於-1 和+1 就是透過 curlerror 和 currerror 來當判斷的標準。

第五章　實驗介紹與結果

本章節將分成兩大實驗，第一是如何利用 DirectShow 架構開發出人臉偵測功能的 Filter，以及當 Filter 開發完成後如何在我們的應用程式中去使用它，第二是如何利用 DLL 方式去封裝我們的人臉偵測系統，以及封裝好之後如何能夠在我們 Win CE 平台的裝置中使用這個功能，藉由這兩個實驗讓我們充分了解不同模組化的形式，他們之間的差異性及特性。以下首先詳細說明如何開發一個 Filter 的流程。

5.1　開發 Filter 流程

開發 Filter 首先要先分析這個 Filter 的功能，以本論文為例是要做一個影像進來之後，對影像做人臉偵測的演算法計算，接著框出人臉，就這個功能分析起來，此 Filter 需要一個輸入 Pin 來接收影像資料，也需要一個輸出 Pin 來把處理完的影像資料送出去，因此可以判斷這個 Filter 是一個 Transform Filter 的取向，所以就可以選定 CTransformFilter 這個 DirectShow 提供的基礎類別來當我們的父類別，當然 DirectShow 還有提供其他功能取向的父類別，例如 CSource、CBaseRenderer 等等。選定一個父類別來繼承對於 Filter 開發者來說可以大大降低開發時間，因為這個父類別會把一些通用的函式都寫好，這樣開發者就省去要開發這些函式的時間，以 CTransformFilter 這個類別為例它就已經幫我們實現好 CTransformInputPin 和 CTransformOutputPin，所以當我們選擇 CTransformFilter 這個類別當我們的父類別時，其實 Pin 類都已經幫我們實現好了，我們就可以使用它自帶的 Input Pin 以及 Output Pin 的功能而不用再自行去開發，因此減少許多開發時間，但是如果開發者對 Pin 有獨特的需求，當然也可以重寫 CTransformInputPin 和 CTransformOutputPin 來實現獨特的功能。

經過以上分析後，我們宣告一個 CFDFilter 繼承 CTransformFilter 的類別，程式碼實現如圖 5-1 所示

```
class CFDFilter : public CTransformFilter
```

圖 5-1　宣告 Filter 類別

當新建出一個類別來代表我們的 Filter 之後就可以開始一個一個實現這個 Filter 的功能函式。基本 Filter 的開發可以分成兩大部分，一部分是 COM 註冊架構的實現另一部分是 Filter 架構的實現。首先 COM 註冊架構的實現就是要把這個 Filter 註冊到系統的註冊表，而這個註冊的用意就是要解決傳統動態連結的缺點，讓外部使用者不需再透過自己手動去連結檔案，只要給外部使用者一個 GUID，就可以透過這個 GUID 去註冊表中查找到這個 Filter 的資料並自動完成連結的動作，以下我們將先介紹 COM 註冊架構的實現。

5.1.1　COM 註冊架構實現(Filter 註冊)

在第二章的 2.3 節我們已經有介紹過註冊資訊有四種結構，也有對每個結構的成員所代表的意義作詳盡的介紹，而在本節我們就是要依據我們 Filter 的需求去設定每個結構的成員變數，實現的程式碼如圖 5-2 所示

```
const AMOVIESETUP_MEDIATYPE sudPinTypes =
{
    &MEDIATYPE_Video,       // Major type
    &MEDIASUBTYPE_RGB24     // Minor type
};

const AMOVIESETUP_PIN sudpPins[] =
{
    { L"Input",             // Pins string name
      FALSE,                // Is it rendered
      FALSE,                // Is it an output
      FALSE,                // Are we allowed none
      FALSE,                // And allowed many
      &CLSID_NULL,          // Connects to filter
      NULL,                 // Connects to pin
      1,                    // Number of types
      &sudPinTypes          // Pin information
    },
    { L"Output",            // Pins string name
      FALSE,                // Is it rendered
      TRUE,                 // Is it an output
      FALSE,                // Are we allowed none
      FALSE,                // And allowed many
      &CLSID_NULL,          // Connects to filter
      NULL,                 // Connects to pin
      1,                    // Number of types
      &sudPinTypes          // Pin information
    }
};
```

支援媒體

Pin 的說明

圖 5-2 註冊程式碼實現

接著我們講解我們所設定的意義，首先從 Filter 說明開始，我們要實現的 Filter 名稱叫做 FaceDetect，而它有一個 CLSID 叫做 CLSID_FaceDetect，沒有特別要求優先權所以 Merit 設定為 MERIT_DO_NOT_USE，且這個 Filter 有兩支 Pin(Pin 的資訊在 sudpPins 這個結構中)，一個是輸入 Pin 名稱為 Input，一個是輸出 Pin 名稱為 Output，而這兩支 Pin 支援的主要媒體格式(媒體格式在 sudPinTypes 這個結構中)是視訊格式，且這個視訊格式要是 RGB24 的形式，所以我們分別對 Major Type 和 Minor Type 做 MEDIATYPE_Video 以及 MEDIASUBTYPE_RGB24 的設定。Filter 細節都訂好之後，透過 CFactoryTemplate 結構，我們在類別工廠創建我們的 Filter，包括名字(FaceDetect)、CLSID(CLSID_FaceDetect)、類別創建函式(CFDFilter::CreateInstanc)等。接著我們透過 DllRegisterServer()以及 DllUnregisterServer()兩個必要函式來幫我們註冊或反註冊以上的結構到註冊表，而這兩個函式的實現只要簡單的調用 AMovieDllRegisterServer2 這個 API 即可，這個 API 是專門給 DirectShow 的 Filter 註冊使用，它會自動幫我們把以上的結構註冊到註冊表。最後再實現入口函式 DllMain，這個入口函式也是只要調用 DllEntryPoint 這個 API 即可，程式碼實現如圖 5-3 所示


```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  dwReason,
                      LPVOID lpReserved)
{
    return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}

STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
} // DllRegisterServer

STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
} // DllUnregisterServer
```

圖 5-3　入口及註冊函式程式碼

到這裡為止，其實我們已經實現了開發 Filter 流程的一半了，也就是 COM 架構的部分我們已經實現完成。當然如果是要開發一般 COM 架構不是基於 DirectShow，可能就沒有這些 DirectShow 提供的方便結構去使用，就變成每個註冊資料都要自己去定義自己去創造出來，所以用 DirectShow 開發多媒體應用程式真的為程式設計者節省很多時間。

5.1.2　Filter 架構實現

Filter 架構實現這部份，其所必須實現的函式會依所選擇的父類別不同而有所不一樣，依我們 Filter 的功能所選擇的父類別是 CTransformFilter，而這個父類別有五個必須實現的函式，如表 5-1 所示

函式名稱	敘述
CheckInputType	檢查輸入 Pin 媒體類型是否支援
CheckTransform	檢查輸出入 Pin 媒體類型是否正確
DecideBufferSize	決定輸出 Pin 要建立多大 Buffer
GetMediaType	取得媒體類型
Transform	輸入資料處理函式

表 5-1　CTransformFilter 必須實現函式

CheckInputType

這個函式實現的目的是當上一級的輸出 Pin 要跟我們 Filter 的輸入 Pin 做連接動作的時候，將先觸發這個函式做檢查的動作，以確定是否可以連接成功。如果是雙方都支援的格式將返回 NOERROR 的值，反之將返回 E_FAIL 的值，而我們實現的程式碼如圖 5-4 所示。

```
HRESULT CFDFilter::CheckInputType(const CMediaType *mtIn)
{
    CheckPointer(mtIn,E_POINTER);

    // check this is a VIDEOINFOHEADER type
    if (*mtIn->FormatType() != FORMAT_VideoInfo)
    {
        return E_INVALIDARG;
    }

    // Can we transform this type
    if (CanPerformFD(mtIn))
    {
        return NOERROR;
    }

    return E_FAIL;
}
```

圖 5-4　CheckInputType 程式碼

程式碼的實現首先透過 CheckPoint 的 API 檢查 Input Pin 的媒體類型是否正常可以工作，確定可以工作之後，先初步檢查這個媒體結構中的成員 FormatType 是否為我們所要求的 Format_VideoInfo 的型態，如果不是 Format_VideoInfo 的型態就跳出函式返回 E_INVALIDARG，如果成功我們自己又寫了一個 CanPerformFD 的函式來細部檢查媒體類型，CanPerformFD 的程式碼實現如圖 5-5 所示。

```
BOOL CFDFilter::CanPerformFD(const CMediaType *pMediaType) const
{
    CheckPointer(pMediaType,FALSE);

    if (IsEqualGUID(*pMediaType->Type(), MEDIATYPE_Video))
    {
        if (IsEqualGUID(*pMediaType->Subtype(), MEDIASUBTYPE_RGB24))
        {
            VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pMediaType->Format();
            return (pvi->bmiHeader.biBitCount == 24);
        }
    }

    return FALSE;

} // CanPerformEZrgb24
```

圖 5-5　CanPerformFD 程式碼

CanPerformFD 這個函式將一層一層檢查所支援的媒體類型，因為我們這個 Filter 是設定支援 RGB24 的視訊資料，所以首先檢查 Type 是否為 Video 的類型，再來下一層檢查輔助 Type 是否為 RGB24，如果是 RGB24 就再更近一步確認這個結構的 Format 成員裡面的 biBitCount 成員是否為 24(biBitCount 成員代表每個 Pixel 的總 bit 數)，如果都正確就返回 TRUE，反之返回 FALSE。

CheckTransform

這個函式實現的目的是在資訊處理之前，會觸發這個函式對輸出入 Pin 再作一次檢查的動作以確保輸出入 Pin 為我們所支援的媒體格式。然而依我們 Filter 的定義輸入 Pin 和輸出 Pin 是同一類型，所以程式碼的實現只要確認輸入 Pin，而輸入 Pin 確認後再判斷兩支 Pin 是否相等即可，程式碼如圖 5-6 所示。

```
HRESULT CFDFilter::CheckTransform(const CMediaType *mtIn, const CMediaType *mtOut)
{
    CheckPointer(mtIn,E_POINTER);
    CheckPointer(mtOut,E_POINTER);

    if (CanPerformFD(mtIn))
    {
        if (*mtIn == *mtOut)
        {
            return NOERROR;
        }
    }

    return E_FAIL;

} // CheckTransform
```

圖 5-6　CheckTransform 程式碼

程式碼實現比較簡單，可以先呼叫之前已經寫好的 CanPerformFD 函式幫我們判斷輸入 Pin 是否符合媒體類型，如果符合只要再判斷輸出入 Pin 兩個是否相同即可達到我們的目的。

DecideBufferSize

當我們 Filter 的輸出 Pin 完成與下一級的輸入 Pin 連接時，必須分配一個共同的記憶體以供傳輸使用，而實現這個函式的目的就是在做分配這個記憶體的動作，在介紹這個函式實現之前要先介紹管理 Sample 分配器的一個結構叫做 ALLOCATOR_PROPERTIES，它包含四個成員如表 5-2 所示

成員	型態
cBuffers	long
cbBuffer	long
cbAlign	long
cbPrefix	long

表 5-2　ALLOCATOR_PROPERTIES

cBuffers ： 在分配器中要建立的 Buffer 個數。

cbBuffer ： 所要建立 Buffer 的大小，單位為 Byte。

cbAlign ： buffer 起始位址要求多少 Byte。

cbPrefix： 每個 Buffer 前面有多少前置碼空間，這個值對於有參數要傳遞給下一級 Filter 時可以使用，塞在 Buffer 前面一起傳下去。

了解這個結構後就可以比較清楚如何去實現 DecideBufferSize 這個函式，實現的程式碼如圖 5-7 所示

```
HRESULT CFDFilter::DecideBufferSize(IMemAllocator *pAlloc,ALLOCATOR_PROPERTIES *pProperties)
{
    // Is the input pin connected

    if (m_pInput->IsConnected() == FALSE) {
        return E_UNEXPECTED;
    }

    CheckPointer(pAlloc,E_POINTER);
    CheckPointer(pProperties,E_POINTER);
    HRESULT hr;

    pProperties->cBuffers = 1;      //Number of buffers created by the allocator
    pProperties->cbBuffer = m_pInput->CurrentMediaType().GetSampleSize();
    ASSERT(pProperties->cbBuffer);

    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProperties,&Actual);
    if (FAILED(hr)) {
        return hr;
    }

    ASSERT( Actual.cBuffers == 1 );

    if (pProperties->cBuffers > Actual.cBuffers ||
        pProperties->cbBuffer > Actual.cbBuffer) {
        return E_FAIL;
    }
    return NOERROR;

} // DecideBufferSize
```

圖 5-7　DecideBufferSize 程式碼

程式碼一開始透過輸入 Pin 的 IsConnect 函式去判斷輸入 Pin 是否為連接狀態，因為 DecideBufferSize 這個函式要做的事是去決定以完成連接輸出 Pin 的 Sample 分配器大小，而要完成輸出 Pin 的連接前提是輸入 Pin 要是連接的狀態下，因此我們才要先確認輸入 Pin 是連接的狀態，確定輸入 Pin 是連接之後就可以透過設定 ALLOCATOR_PROPERTIES 這個結構的成員來達到建立分配器的功能，首先我們透過設定 CBuffers=1 來決定我們 Buffer 的個數，再來是設定大小，那因為我們 Filter 的輸入 Pin 跟輸出 Pin 是共同使用一個媒體格式，所以兩支 Pin 的 Sample 大小是一樣的，因此我們透過輸入 Pin 的 GetSampleSize 來取得 Sample 大小的值設定到 CbBuffer 這個成員即完成分配器屬性的設定，接著透過 SetProperties 函式把我們設定好的結構告訴系統分配器，則系統就會依照我們設定的個數大小幫我們完成配置，這個函式實現就算完成了。

GetMediaType

實現這個函式的目的是要讓下一級 Filter 的輸入 Pin 能夠透過這個函式獲取我們 Filter 輸出 Pin 所支援的媒體格式方便連接時做檢查的動作，所以當我們輸出 Pin 要跟下一級輸入 Pin 連接時就會觸發這個函式去提供下一級輸入 Pin 媒體資訊以供比對，而這是輸出 Pin 連接時會使用到的函式，因此使用這個函式的前提是輸入 Pin 要處於連接的狀態，程式碼如圖 5-8 所示

```
HRESULT CFDFilter::GetMediaType(int iPosition, CMediaType *pMediaType)
{
    // Is the input pin connected

    if (m_pInput->IsConnected() == FALSE) {
        return E_UNEXPECTED;
    }

    // This should never happen

    if (iPosition < 0) {
        return E_INVALIDARG;
    }

    // Do we have more items to offer

    if (iPosition > 0) {
        return VFW_S_NO_MORE_ITEMS;
    }

    CheckPointer(pMediaType,E_POINTER);
    *pMediaType = m_pInput->CurrentMediaType();

    return NOERROR;

} // GetMediaType
```

圖 5-8 GetMediaType 程式碼

函式一開始要先檢查輸入 Pin 是否連接，必須是連接的狀態才能繼續往下做，再來會檢查 iPosition 這個參數是否為 0，這是一個媒體序號必須為 0 才可以繼續工作，一切檢查工作完畢後，即可返回輸出 Pin 的媒體類型以供下一級 Filter 使用，而我們這個 Filter 輸出入 Pin 是使用同一個媒體類型，因此可以簡單的返回輸入 Pin 的媒體類型即可。

Transform

實現這個函式對 Transform Filter 是最重要的，它的目的是對 Source Filter 傳送來的資料作處理，舉凡編解碼、影像處理、音訊處理等，都是透過 Transform Filter 的 Transform 函式所運作。而我們的 Filter 是要對影像作人臉偵測，所以我們所實現的人臉偵測程式碼將會被放入這個函式中，其程式碼的

實現如圖 5-9 所示

```
HRESULT CFDFilter::Transform(IMediaSample *pIn, IMediaSample *pOut)
{
    CheckPointer(pIn,E_POINTER);
    CheckPointer(pOut,E_POINTER);

    // Copy the properties across

    HRESULT hr = Copy(pIn, pOut);
    if (FAILED(hr)) {
        return hr;
    }

    return Transform(pOut);

    return NOERROR;

} // Transform
```

圖 5-9　Transform 程式碼

程式碼的實現我們的作法是先把輸入 Pin 資料搬到輸出 Pin，然後直接在輸出 Pin 上做處理的動作，當然另一種作法也可以先在輸入 Pin 上作完處理再搬到輸出 Pin 上。因此我們延伸了兩個函式 Copy 以及 Transform 來處理搬資料以及影像處理兩件事情，而 Copy 函式是將輸入 Pin 資料複製到輸出 Pin 上，Transform 函式是將輸出 Pin 上的資料做我們所要求的處理後再放回輸出 Pin 的 Sample 分配器上等下一級 Filter 來拿取，如此就實現了 Transform 的功能。接下來要仔細介紹 Copy 和 Transform 內部程式碼到底如何實現，首先是 Copy 函式的介紹，程式碼如圖 5-10 所示

```
HRESULT CFDFilter::Copy(IMediaSample *pSource, IMediaSample *pDest) const
{
    CheckPointer(pSource,E_POINTER);
    CheckPointer(pDest,E_POINTER);

    // Copy the sample data

    BYTE *pSourceBuffer, *pDestBuffer;
    long lSourceSize = pSource->GetActualDataLength();

    pSource->GetPointer(&pSourceBuffer);
    pDest->GetPointer(&pDestBuffer);

    CopyMemory( (PVOID) pDestBuffer,(PVOID) pSourceBuffer,lSourceSize);

    long lDataLength = pSource->GetActualDataLength();
    pDest->SetActualDataLength(lDataLength);

    return NOERROR;

} // Copy
```

圖 5-10　Copy 程式碼

因為這個函式主要是要做資料複製的動作，所以一開始會呼叫 GetActualDataLength 的 API 來獲取輸入 Pin 上資料的總大小，接著在呼叫

GetPointer 的 API 來取得輸入 Pin 以及輸出 Pin 在記憶體中放資料的起始位址，然後把以上得到的參數丟給 CopyMemory 函式，而它就會幫我們把輸入 Pin 的資料複製到輸出 Pin 上去，最後再將複製的總大小設定到輸出 Pin 上，如此就完成 Copy 函式的實現了。再來就要仔細介紹最重要的 Transform 函式，其程式碼實現如圖 5-11 所示

```
HRESULT CFDFilter::Transform(IMediaSample *pMediaSample)
{
    BYTE *pData;                // Pointer to the actual image buffer
    long lDataLen;              // Holds length of any given sample
    int iPixel;
    AM_MEDIA_TYPE* pType = &m_pInput->CurrentMediaType();
    VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pType->pbFormat;
    int cxImage    = pvi->bmiHeader.biWidth;
    int cyImage    = pvi->bmiHeader.biHeight;
    CheckPointer(pMediaSample,E_POINTER);
    pMediaSample->GetPointer(&pData);

    int numPixels  = cxImage * cyImage;
    BYTE* TempBuffer = new BYTE[numPixels * 3];
    int n_temp = cyImage-1;

    for(iPixel=0;iPixel<cyImage;iPixel++)
    {
        memcpy(TempBuffer+n_temp*cxImage*3,pData+iPixel*cxImage*3,cxImage*3);
        n_temp--;
    }
    TempBuffer = FaceDetect(TempBuffer,cxImage,cyImage,30,30,1.1);
    n_temp = cyImage-1;
    for(iPixel=0;iPixel<cyImage;iPixel++)
    {
        memcpy(pData+n_temp*cxImage*3,TempBuffer+iPixel*cxImage*3,cxImage*3);
        n_temp--;
    }
    delete [] TempBuffer;
    return NOERROR;

} // Transform (in place)
```

上下翻轉

圖 5-11　Transform 程式碼

這個函式是要對輸出 Pin 的資料作人臉偵測處理後再放回輸出 Pin，以供下一級 Filter 來拿取處理完的資料。所以首先要取得資料才可以做處理，而取得資料之前必須知道資料的規格才可以按照規格作存取的動作，所以我們透過 CurrentMediaType 獲得媒體格式，再從媒體格式這個結構的 Format 成員即可取得這個 Sample 的長以及寬，且可透過 GetPoint 的 API 來幫我們取得資料存放的起始位址，有長寬也有起始位址如此一來我們就可以取得所有資料，即是一個長乘以寬的陣列。而我們的 Formattype 是 FORMAT_VideoInfo 的形式所以它的資料是 BMP(BitMaP)格式，查看 BMP 規範後知道 BMP 資料是上下顛倒的形式，所以我們先對取得的資料作一個上下翻轉回來的處理讓資料變成我們熟悉的由上往下的形式，接著把這個資料送進我們已經開發好的人臉偵測函式(FaceDetect)中做

人臉偵測，FaceDetect 這個函式有六個參數必須代入分別是

- Image: 要做人臉徵測的圖像資料
- Image_width: 圖像資料的寬
- Image_Height: 圖像資料的高
- Win_width: 人臉檢測視窗寬的初始值
- Win_Height: 人臉檢測視窗高的初始值
- Scale: 每次檢測視窗加大的比率

跑完 FaceDetect 這個函式後，它會返回一個偵測完的資料陣列，最後再把這個資料陣列複製到輸出 Pin 的 Sample 分配器上，如此即完成了整個 Transform 函式的功能實現，而且也完成了我們人臉偵測 Transform Filter 的開發，如圖 5-12 即是我們開發出來的 Filter 圖示顯示在 GraphEdit 中的樣子

圖 5-12 人臉偵測 Filter

5.2 應用程式使用 Filter 的方法

當 Filter 開發完成後，最重要的是如何使用這個 Filter，所以本節將詳細介紹如何使用我們開發出來的人臉偵測 Filter 來撰寫應用程式。首先當我們開發完一個 Filter 時可以先透過 GraphEdit 調試看看這個 Filter 的功能正不正常，而 GraphEdit 是由 DiectShow SDK 所提供的一個調試工具，如圖 5-13 所示就是我們透過 GraphEdit 調試的結果

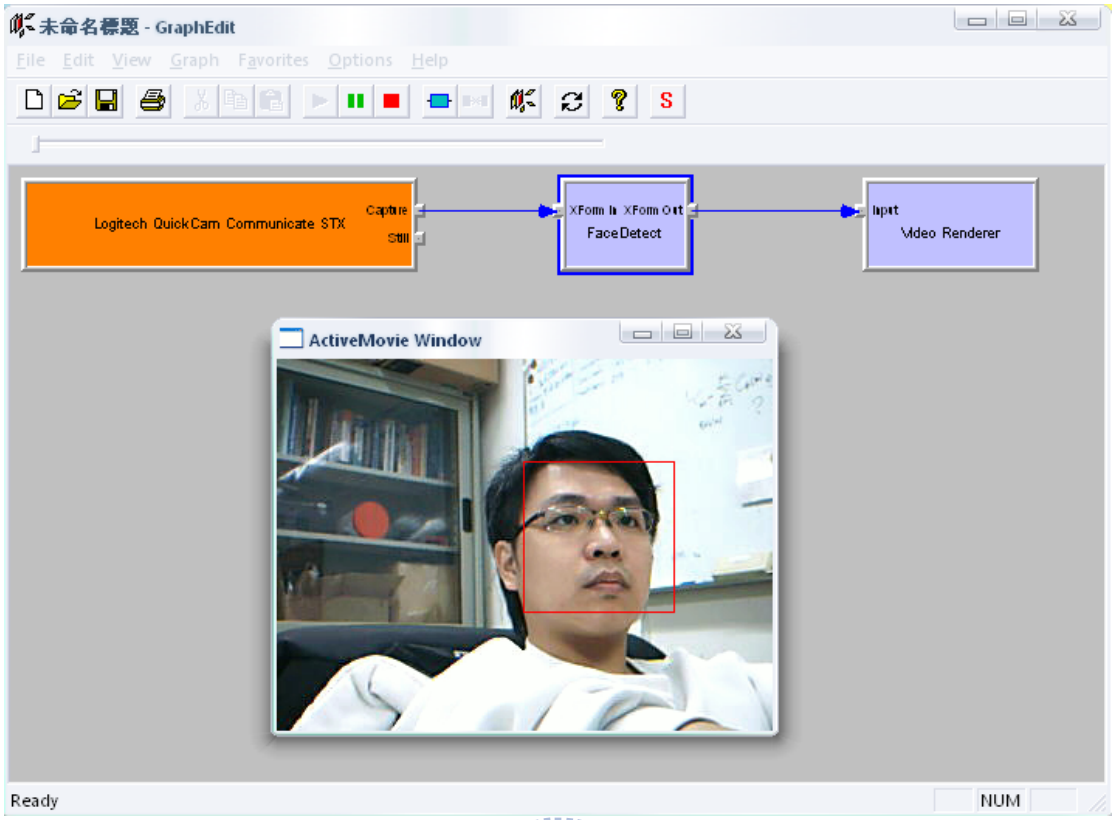


圖 5-13 GraphEdit 調試

在這個調試中，我們可以把 GraphEdit 看成是一個 Filter Graph Manager，而我們要在這個 Graph 上拉出一條 Filter chain 去執行我們所要求的任務。所以一開始拉一個視訊 Camera 當作 Source Filter，來為我們提供影像資訊來源，接著連接我們開發好的人臉偵測的 Transform Filter，為我們對影像資訊做人臉偵測的處理，最後再接一個 Video Render 來為我們顯示影像結果在螢幕上，再來按下 play the graph 就可以開始看這整個功能是否正常，結果如預期是有辦法徵測出人臉的位置，所以透過調試可以知道我們的 Filter 是正常可用的。那就可以把我們的 Filter 拿給應用程式人員使用，而我們會給程式人員一個 Filter 的註冊檔(FDFilter.ax)，以供他註冊使用，並且給他 Filter 的 CLSID(或者他註冊後自行到註冊表查詢)，供他在撰寫程式時呼叫 Filter 使用，前置準備完成就可以來介紹如何開發應用程式。

要使用 Filter 開發當然要先建立一個 Filter Graph Manager 來幫我們管理 Filter，而且也要把我們會使用到的 Filter 都建立出來，以方便我們後續使用

而這些建立工作的程式碼如圖 5-14 所示

```
//Creating Instance of GraphBuilder
CoCreateInstance (CLSID_FilterGraph,NULL,CLSCTX_INPROC_SERVER,IID_IGraphBuilder,(void**)&pGraph);

//Creating Video capture filter
HRESULT hr;
ICreateDevEnum *pSysDevEnum = NULL;
IEnumMoniker *pEnumCat = NULL;

hr = CoCreateInstance(CLSID_SystemDeviceEnum,NULL,CLSCTX_INPROC_SERVER,IID_ICreateDevEnum,(void **)&pSysDevEnum);
hr=pSysDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory,&pEnumCat,0);
if(hr==S_OK)
{
    IMoniker *pMoniker = NULL;
    ULONG cFetched;

    pEnumCat->Next(1,&pMoniker,&cFetched);
    hr=pMoniker->BindToObject(NULL,NULL,IID_IBaseFilter,(void **)&pSource);
    pMoniker->Release();
    pEnumCat->Release();
}
pSysDevEnum->Release();

//creating FaceDetect Filter
hr = CoCreateInstance(CLSID_FaceDetect, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&pFaceDetect);
//Creating Render Filter
hr = CoCreateInstance(CLSID_Rend, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&pVideoRenderer);
```

Filter Graph Manager 建立

視訊裝置建立

FaceDetect Filter 建立

圖 5-14　建立 Filter 程式碼

第一步:建立 Filter Graph Manager

透過 CoCreateInstance 可以幫我們呼叫建立 COM 元件，所以我們使用 CoCreateInstance 來呼叫建立 Filter Graph Manager。

第二步:建立用到的 Filter

首先建立 Source Filter，而我們的 Source Filter 是視訊 Camera 硬體設備，而要建立硬體設備需要透過 DirectShow 提供的 SystemDeviceEnum 這個系統設備枚舉器來幫我們找到這個視訊設備，其用法就是先建立系統設備枚舉器，再來透過 CreateClassEnumerator 函式來告知枚舉器我們要在那一個屬性的資料下做搜尋，而這邊我們需求是視訊裝置，所以我們是在 VideoInputDevice 下做查找，當設定好之後就可以開始使用 Next 做逐一的查找動作，那因為本電腦只有一個 VideoInput 的設備所以只會找到唯一一個裝置就是我們所要的，不然如果有很多裝置就要再做一些判斷裝置名字的動作才能找到想要的那個裝置，而找到之後

透過 BindToObject 函式就會幫我們把硬體裝置建立起來。接著呼叫我們的 FaceDetect Filter 以及 Render Filter 就只要透過 CoCreateInstance 即可完成。到這裡所有要使用的 Filter 都建立完成，接下來就要實現串接的動作，其程式碼如圖 5-15 所示

```
pGraph->AddFilter(pSource, L"MM_Source");
pGraph->AddFilter(pFaceDetect, L"MM_Facedetect");
pGraph->AddFilter(pVideoRenderer, L"MM_VideoRenderer");

pSource->EnumPins(&EnumPins);
EnumPins->AddRef();
EnumPins->Reset();
EnumPins->Next(1, &OutPin, &fetched);
EnumPins->Release();

pFaceDetect->EnumPins(&EnumPins);
EnumPins->AddRef();
EnumPins->Reset();
EnumPins->Next(1, &InPin, &fetched);
InPin->QueryPinInfo(&pinfo);
while(pinfo.dir != PINDIR_INPUT)
{
     InPin->Release();
     EnumPins->Next(1, &InPin, &fetched);
     InPin->QueryPinInfo(&pinfo);
}
EnumPins->Release();

pGraph->Connect(OutPin, InPin);
InPin->Release();
OutPin->Release();

pFaceDetect->EnumPins(&EnumPins);
EnumPins->AddRef();
EnumPins->Reset();
EnumPins->Next(1, &OutPin, &fetched);
OutPin->QueryPinInfo(&pinfo);
while(pinfo.dir != PINDIR_OUTPUT)
{
     OutPin->Release();
     EnumPins->Next(1, &OutPin, &fetched);
     OutPin->QueryPinInfo(&pinfo);
}
EnumPins->Release();

pVideoRenderer->EnumPins(&EnumPins);
EnumPins->Reset();
EnumPins->Next(1, &InPin, &fetched);
EnumPins->Release();

// connect
pGraph->Connect(OutPin, InPin);
InPin->Release();
OutPin->Release();
```

取得 Source Filter 的輸出 Pin

取得 FaceDetect Filter 的輸入 Pin 與 Source Filter 的輸出 Pin 做連接

圖 5-15　Filter 串接程式碼

第三步:添加 Filter 到 Filter Graph Manager

待連接的 Filter 必須處於同一個 Filter Graph Manager 中才能進行連接，所以一開始要先透過 AddFilter 把要連接的 Filter 通通都加到 Filter Graph Manager 裡面。

第四步:Filter 連接

Filter 連接的基本流程就是三個步驟，取得 Pin、判斷 Pin、連接 Pin。以 Source

Filter 與 Transform Filter 連接為例，首先以 EnumPins 函式來建立 Pin 枚舉器，接著再以 Next 來取得 Pin，因為 Source 只有一支 Pin 為輸出 Pin，所以我們就無須再判斷取得的 Pin 是否為輸出 Pin，但 FaceDetect Filter 有兩支 Pin 一支輸出 Pin 一支輸入 Pin，所以當我們用 Next 取得 Pin 時，還要加以判斷是否為我們所需要的輸入 Pin，而透過 PIN_INFO 結構裡面的 PIN_DIRECTION 成員就可以幫助我們判斷是否為輸入 Pin，判斷出輸入 Pin 之後，就可以使用 Connect 函式以 Source Filter 的輸出 Pin 和 FaceDetect Filter 的輸入 Pin 作為參數完成 Filter 之間連接的動作，用同樣的方法也可以把 FaceDetect Filter 跟 Render Filter 連接起來。

接著我們來介紹如何把視訊視窗嵌入到我們的應用程式視窗，以及 Filter Graph Manager 如何管理狀態，程式碼實現如圖 5-16 所示

```
pGraph->QueryInterface(IID_IMediaControl, (void **)&pMediaControl);
pGraph->QueryInterface(IID_IVideoWindow, (void **)&pVideoWindow);

RECT rc;
HWND hwnd;
hwnd = GetDlgItem(IDC_SCREEM)->m_hWnd;
GetDlgItem(IDC_SCREEM)->GetClientRect(&rc);
pVideoWindow->put_Owner((OAHWND)GetDlgItem(IDC_SCREEM)->GetSafeHwnd());
pVideoWindow->put_WindowStyle(WS_CHILD | WS_CLIPCHILDREN | WS_CLIPCHILDREN);
pVideoWindow->put_Left(0);
pVideoWindow->put_Top(0);
pVideoWindow->put_Width(rc.right - rc.left);
pVideoWindow->put_Height(rc.bottom - rc.top);
pVideoWindow->put_Visible(OATRUE);

//Play the file
pMediaControl->Run();
```

圖 5-16　嵌入視訊視窗程式碼

第五步:管理狀態

我們透過 Filter Graph Manager 的 QueryInterface 為我們建立 IMediaControl 這個介面，而 IMediaControl 這個介面提供三個 API 給我們去執行 Graph 的三種狀態，分別是 Run(Play the graph)、Pause(Pause the graph)、Stop(Stop the graph)，因此我們就可以藉由這個介面去管理我們 Graph 的狀態。

第六步:嵌入 Render 視窗

透過 Filter Graph Manager 的 QueryInterface 為我們建立 IVideoWindow 這個介面，這個介面就是提供給我們把 Filter Graph 的結果視窗嵌入到我們的應用程式視窗中，這個介面有提供很多 API 讓我們可以輕易的去設定視窗，以下介紹這些 API 的功能

- put_Owner ： 設定要嵌入的視窗
- put_WindowStyle ： 設置視窗的風格特性
- put_Left ： 設置視窗 X 軸起始座標
- put_Top ： 設置視窗 Y 軸起始座標
- put_Width ： 設置視窗寬度
- put_Height ： 設置視窗高度
- put_Visible ： 顯示或隱藏視窗，OATRUE 代表顯示，OAFALSE 代表隱藏

設置好嵌入的視窗之後，我們的應用程式就算完成，接著只要使用 IMediaControl 這個介面來把狀態轉換成 Run 的形態就會開始在我們設定好的視窗中播放處理好的視訊畫面，而且在這個應用程式中我們還實現了 Pause 以及 Stop 的狀態按鍵，這兩個按鍵也是簡單的用 IMediaControl 介面的 Pause 以及 Stop 的 API 就可以實現。如圖 5-17 就是應用程式開發完成的結果圖示。

圖 5-17　應用程式實驗結果

5.3 DLL 實驗介紹

前面已經介紹完 DirectShow Filter 的開發，接下來本節將介紹以 DLL(Dynamic Link Library)的方式來封裝我們的人臉偵測系統，首先介紹一下我們的開發環境，這是由精聯科技所提供的一個裝置如圖 5-18 所示，此裝置的作業系統是 Windows CE，因此我們的目標就是在 Win CE 的作業系統底下開發一個人臉偵測的 DLL，讓這個裝置的應用程式人員能夠透過 DLL 方式來呼叫我們偵測函式達到人臉偵測的功能。

圖 5-18 實驗裝置

在開發這個 DLL 時，我們所使用的編譯器是 Embedded Visual C++，而要封裝 DLL 其實是非常簡單的，只需要在我們一般開發的程式作一些修改即可

第一步:實現入口函式

實現 DLLMain 這個函式來提供 DLL 一個入口點，程式碼實現如圖 5-19 所示，

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}
```

圖 5-19 DLLMain 程式碼

第二步:添加導出指令

將我們要提供給外部使用者使用的函式前面加上導出指令__declspec(dllexport)就可以完成 DLL 的封裝，而一般我們會在導出指令前面在多加一個指令 extern "C"，其目的是我們不知道函式調用者使用的編譯器是什麼，而使用這個指令可以幫助我們解決使用不同 C 編譯器會產生的不同問題。其程式碼的實現如 5-20 所示。

```
extern "C" __declspec(dllexport)BYTE* FaceDetect
```

圖 5-20　導出函式程式碼

完成以上修改後就算完成 DLL 的封裝，且編譯器會產生.dll 檔以及.lib 檔，這些檔案就是要提供給別人使用的函式庫檔案，而應用程式開發者拿到這兩個檔案以及我們的.h 檔之後，他有兩種連結方式可以去選擇，分別為顯性連結以及隱性連結

- 顯性連結: 在應用程式某個地方需要用到我們 DLL 的函式時，必須透過 LoadLibrary 這個指令先把我們的 DLL 掛載進來，接著再透過 GetProcAddress 指令來取得我們所導出的函式位址，即可使用我們的函式。但每次要用到函式時都要重新 LoadLibrary 再重新 GetProcAddress 有些麻煩，不過好處是比較彈性化，用到時再載入不用時可以把它釋放掉。
- 隱性連結: 在應用程式開發一開始就要設定好連結的檔案及路徑透過編譯器的 Setting->Link 功能來設定，即可使用我們的函式。這種方式使用方便載入方法由編譯器幫我們負責處理，但壞處是比較不彈性，因為它一開始就連結起來不管你是否要用，當 DLL 很多時，應用程式開啟速度將變慢。

而我們封裝好人臉偵測功能後，我們選擇使用隱性連結的方式直接連結到精聯科技開發好的應用程式 CameraDemo 中，其設定如圖 5-21 所示

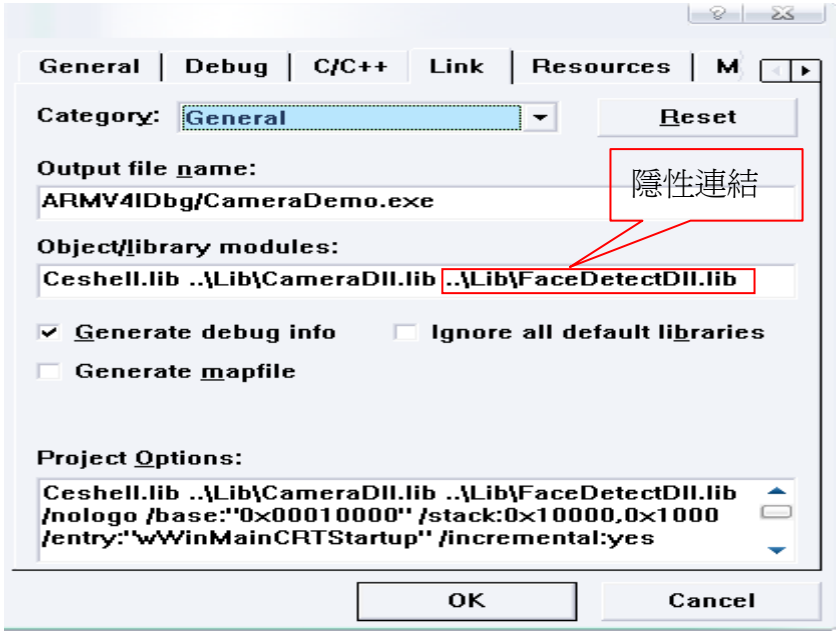


圖 5-21　隱性連結設定

設定好之後在程式中 include 我們的.h 檔後即可直接使用我們的人臉偵測函式，而我們加入到應用程式的程式碼實現如圖 5-22 所示

```
if   (MessageBox(TEXT("Do you want to face detect?"),TEXT("Question"),MB_YESNO)==IDYES)
{
    CShowBMPImage ShowBMP;
    temp= BMP2RAW(temp,_PrvCfg._ImageWidth,_PrvCfg._ImageHeight);
    temp = (unsigned char*)FaceDetect((BYTE*)temp,_PrvCfg._ImageWidth,_PrvCfg._ImageHeight,30,30,1.1);
    ShowBMP.Show(temp,_PrvCfg._ImageWidth,_PrvCfg._ImageHeight);
    ShowBMP.DoModal();
}
else
{
    OnPreviewStart();
}
```

圖 5-22　呼叫 DLL 人臉偵測函式程式碼

當系統 Capture 到一張 BMP 圖像之後就進入到我們所加入的這段程式碼當中，首先我們先去詢問是否要對這張圖像作人臉偵測，如果選否那就繼續做視訊 Preview 的動作，如果選是就會先透過 BMP2RAW 的函式將 BMP 圖像的影像部分資料取出並去掉 Header 的部份，再來就可以把影像資料丟到我們 DLL 所導出的 FaceDetect 函式做偵測的動作，這裡我們可以看到我們是使用隱性連結的方式，所以並不需要 LoadLibrary 的指令就可以直接用，非常方便。接著把做完偵

測的資料當作參數丟給 Show 函式，Show 函式就會幫我們把偵測結果顯示在螢幕上即完成了整個 DLL 實驗的部份，其實驗結果如圖 5-23 所示

圖 5-23 DLL 實驗結果

5.4 分析與比較

以上兩個實驗，我們對人臉偵測系統以 DirectShow Filter 和 DLL 兩種不同的模組化方式進行過封裝，對於這兩種模組化他們都有兩個共同的好處，一個就是原始碼的保護，每個人所開發的原始碼都是自己的財產，但是當自己開發的功能想與人分享又不想把自己辛苦撰寫出來的原始碼讓他人知道時，透過這些封裝的方式就可以輕鬆的達到目的，只需給他人 CLSID 或者 dll、lib 檔，不需要給原始碼就可以分享自己開發的新功能又可以兼顧到自己的智財權是其一好處。另一個好處是程式維護修改的方便性，以這種模組化方式來撰寫程式一來可增加程式的可讀性，不會像一般程式裡一大堆函式，模組化程式或許只會有一個主函式而已，再者當程式需要更新修改功能時，只需更動到要修改的那個模組即可，無須更動到整體架構，要更新也只需加入新的模組。

當兩種模組分別分析時，DirectShow Filter 的形式對於應用程式開發者是

比較方便,因為以Filter而言,應用程式開發者只需使用一個CoCreateInstance的指令即會自動幫我們連結我們的函式庫以方便使用模組內的函式，但是 DLL卻都要手動去連結函式庫，以顯性連結來說要自己寫連結指令，以隱性連結來說要自己手動去設定連結，相較於Filter的自動連結是麻煩了一些。但是以模組開發者而言就是DLL方式比Filter方式來的簡單，以DLL而言只需增加個幾行指令便能封裝完成，但以Filter而言就複雜許多，依功能取向不同需要多實現兩個以上的必要函式，所以DLL在封裝上是非常簡單的，因此兩者各有利弊。

第六章　結論

一般開發應用程式若不是模組化開發，一旦編譯完成後，若要再更新其中一個小函式，就得把整個應用程式再重新編譯只為了更新一個小函式，但是如果改成模組化開發，則這個問題就只需對要修改的模組函式做重新編譯即可，如此一來軟體開發完成後要新增功能或更新就變的方便許多只需加入新模組或修改模組即可達成，不必整個程式再重新編譯過。

再者，一個模組開發完成後要可以給其他外部使用者能快速又簡單的使用，這個模組才算是一個好模組，解決這個問題的一般作法就是把程式封裝成 DLL，然後透過產生的 Lib 檔以及 H 檔給外部使用者，而外部使用者拿到這兩個檔後設定好連結路徑，才可以開始使用這個模組內的功能函式，但是根據本論文實做過後發現如果是要開發多媒體應用程式，選擇 DirectShow Filter 的模組方式對於外部使用者是更方便的，因為它只需一個 CoCreateInstance 的指令就可以完成 DLL 需手動的所有連結動作，節省了開發時間也達到簡單使用的效果。

本研究重點著重在模組化的觀念，這觀念對於學術研究的程式開發者是重要的，因為學術研究常常會開發各種不同演算法的功能函式，如果把這些歷年來開發的函式模組化將有利於我們去管理甚至更新修改它。因此我們藉由研究 DirectShow 的架構來了解模組化的觀念，進而封裝出我們自己的一個人臉偵測 Filter 出來，也藉由與 DLL 的比較來讓我們更了解不同模組化形式差異性，讓之後想模組化的開發者能依自己的取向去選擇適合自己的模組化形式，或者藉由這些模組化的觀念為基礎而發展出另一套新的模組規範出來。

參考文獻

[1]MSDN-Win32&COM http://msdn.microsoft.com/en-us/library/aa139672.aspx

[2]COM http://www.vckbase.com/document/viewdoc/?id=212

[3]COM AddRef http://www.vckbase.com/document/viewdoc/?id=926

[4]陸其明 著，DirectShow 開發指南，北京 - 清華大學出版社 2003

[5]華亨科技股份有限公司，XSB270(EELiod) ADS/Linux/WinCE 實驗開發與實務

[6]黃泰一等 著，Windows CE:嵌入式系統理論與實務，2004

[7]胡宏達，『比較 Real 和 Gentle Adaboost 應用於人臉偵測』，國立交通大學，碩士論文，98 學年度。

[8]Open Source Computer Vision Library (OpenCV)

http://www.intel.com/technology/computing/opencv/index.htm

[9]MSDN-DLL http://msdn.microsoft.com/en-us/library/ms682589(VS.85).aspx

[10]DLL http://forum.slime.com.tw/thread99559.html