國立交通大學

電機學院　電機與控制學程

碩士論文

多媒體播放器之微控器系統設計

Multimedia Player System Design of The Microcontroller

研 究 生：林文彬

指導教授：林錫寬　教授

中華民國一百年三月

多媒體播放器之微控器系統設計
Multimedia Player System Design of The Microcontroller

研 究 生：林文彬　　Student：Wen-Pin Lin

指導教授：林錫寬　　Advisor：Sire-Kuan Lin

國 立 交 通 大 學

電機學院　電機與控制學程

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 of Science

in

Electrical and Control Engineering

March 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年三月

多媒體播放器之微控器系統設計

學生：林文彬　　　　　　　　　　　　　　　　　　　　　　指導教授：林錫寬

國立交通大學　電機學院　電機與控制學程碩士班

摘　　要

近幾年關於多媒體的應用已經廣泛的出現在日常生活中，例如衛星導航、行動電話或電子書等，都能夠看到多媒體應用的蹤跡。

本論文主要目的是透過多媒體的應用，並使用市面上主流的ARM微控器，來熟悉嵌入式系統的軟硬體設計。在硬體方面以脈衝波寬調變(PWM)模式來取代DAC功能輸出；軟體部分則建立SD記憶卡的底層程序並整合FAT檔案儲存系統，使得在開發嵌入式系統時可以快速的解決檔案儲存的應用。並透過播放WAVE音樂來熟悉脈衝編碼調變(PCM)的編解碼流程，與播放MP3音樂來了解音樂壓縮編碼與解碼處理的程序。

Multimedia Player System Design of The Microcontroller

student：Win-Pin Lin　　　　　　　　Advisors：Dr. Sire-Kuan Lin

Degree Program of Electrical and Computer Engineering

National Chiao Tung University

ABSTRACT

In recent years, the applications of multimedia have widely appeared in our daily lives, for instance, GPS navigation systems, cellar phones and electronic books. You can easily spot multimedia applications everywhere.

The purpose of this study was to gain an understanding of the design of embedded hardware-software systems based on commonly-used ARM micro -controllers and through the application of multimedia. In terms of hardware, pulse-width modulation (PWM) was employed to replace digital-to-analog converter (DAC) output. For software, an underlying SD card process was developed and integrated with the FAT file system, which enables rapid file storage when developing an embedded system. The encoding and decoding process of pulse-code modulation (PCM) is used in playing WAVE music files. In addition, by playing MP3 music files we can understand the encoding and decoding process of music compression.

誌　　　謝

首先我要感謝指導教授 林錫寬教授在學期間不吝的指導與教誨，讓我得以順利的完成本論文。再來要感謝我的家人在我求學的日子里，給予我最大的支持與鼓勵，讓我在專班的學習路途上無後顧之憂。

本篇論文謹獻給指導教授及我的家人與在求學路上不吝幫助我與鼓勵我的長輩和朋友們，與您們一起分享這份喜悅。

目　　　錄

中文提要		i
英文提要		ii
誌謝		iii
目錄		iv
圖目錄		vii
表目錄		xii
一、	簡介	1
1.1	研究背景與動機	1
1.2	章節安排	2
二、	STM32F103x 微控器介紹	3
2.1	系統時脈管理	6
2.2	通用輸入/輸出埠	11
2.3	外部中斷與事件	15
2.4	通用計數器	17
2.4.1	向上計數模式	18
2.4.2	脈衝寬度調變模式	20
2.4.3	PWM 低通濾波器	25
2.5	序列週邊界面	29
2.5.1	SPI 模式的初始設定	33
2.5.2	SPI 模式的資料傳輸	36
2.5.3	SPI 模式下的 DMA 傳輸	38
2.6	SDIO 模組	41
2.6.1	SDIO 功能說明	42
三、	SD 記憶卡介紹	51
3.1	SD 記憶卡概述	52
3.1.1	SD 記憶卡的記憶體配置	53
3.1.2	SD 記憶卡架構	55

3.1.3 SD 記憶卡的匯流排結構 57
3.1.4 SD 記憶卡的暫存器 60
3.1.5 CRC 檢查碼 63
3.2 SPI 傳輸模式 65
3.2.1 SPI 模式選擇 66
3.2.2 指令格式 68
3.2.3 SPI 模式的回應 71
3.2.4 卡識別模式 77
3.2.5 資料傳輸 80
3.3 SD Bus 傳輸模式 89
3.3.1 資料線模式設定 91
3.3.2 SD Bus 模式的回應 93
3.3.3 SD Bus 模式的卡識別模式 97
3.3.4 SD Bus 模式的資料傳輸 101
四、 FAT 檔案系統 107
4.1 FAT 系統概述 108
4.2 FAT 的保留磁區 111
4.2.1 開機磁區 112
4.2.2 磁碟分區表 114
4.2.3 啟動參數區 115
4.3 文件分配表區域 120
4.4 目錄區域 122
4.5 資料區域 125
五、 WAVE 音效檔 129
5.1 WAVE 檔案資料結構 130
5.2 WAVE 檔案播放 135
5.2.1 WAVE 檔頭解碼 136
5.2.2 緩衝區設定 138
5.2.3 音訊解碼流程 140

5.2.4	播放流程	143
六、	MP3 音效格式	145
6.1	MP3 文件的標籤格式	146
6.2	MP3 音框格式	150
6.3	MP3 解碼流程	153
6.4	MP3 解碼程序	154
七、	結果與展望	159
參考文獻		161
附錄一	STM32F103x 記憶體映射	163
附錄二	SD 記憶卡命令描述	164
附錄三	FAT 磁區內容與結構	167
附錄四	MP3 音框檔頭格式	169
附錄五	MP3 播放器電路與 PCB 佈線圖	170
自傳		180

圖 目 錄

圖 2-1 多媒體系統功能方塊圖 5
圖 2-2 HSE/LSE 時脈來源 7
圖 2-3 內部高速時脈(HSI)與相鎖迴路(PLL)電路 8
圖 2-4 內/外部低速時脈(LSE/LSI)電路 8
圖 2-5 主頻率來源電路 9
圖 2-6 RCC 初始設定程序 10
圖 2-7 編譯軟體的 RCC 設定 10
圖 2-8 編譯軟體的 GPIO 設定 13
圖 2-9 SD 記憶卡電源控制與插入偵測電路 14
圖 2-10 SD 記憶卡電源控制與插入偵測程式 14
圖 2-11 外部中斷輸入電路 15
圖 2-12 EXTI 設定副程式 16
圖 2-13 編譯軟體的 EXIT 設定 16
圖 2-14 TIM 中斷處理架構 17
圖 2-15 向上計數模式時序 18
圖 2-16 編譯軟體的 TIM4 設定 19
圖 2-17 PWM duty cycle 波形 20
圖 2-18 PWM 模式 1 與模式 2 波形 21
圖 2-19 CCR 為 4 與 255 的 PWM 輸出波形 22
圖 2-20 TIM3 計數器設定 23
圖 2-21 模擬 PWM 輸出波形 24
圖 2-22 放大器與低通濾波器電路 25
圖 2-23 二階低通濾波器分析 26
圖 2-24 一階高通濾波器分析 27
圖 2-25 PWM 濾波器的分析 27
圖 2-26 輸出 8KHz PWM 訊號經過濾波器的波形 28
圖 2-27 輸出 44.1KHz PWM 訊號經過濾波器的波形 28

圖 2-28 單一 SPI Master 匯流排對複數 Slave 29
圖 2-29 SPI 模組方塊圖 30
圖 2-30 SPI 模式資料傳輸的時序圖 34
圖 2-31 SPI 模式設定副程式 35
圖 2-32 SPI 模式的資料發送副程式 36
圖 2-33 SPI 模式的資料接收副程式 37
圖 2-34 SPI 傳輸的 DMA 設定副程式 39
圖 2-35 SPI 使用迴圈與 DMA 的傳輸程式 40
圖 2-36 SDIO 模組方塊圖 42
圖 2-37 SDIO 轉接器模組方塊圖 42
圖 2-38 SDIO 命令通道狀態機方塊圖 43
圖 2-39 SDIO 命令發送副程式 44
圖 2-40 SDIO 資料通道狀態機方塊圖 45
圖 2-41 DPSM 設定副程式 46
圖 2-42 SDIO 模式設定副程式 47
圖 3-1 SD 記憶卡標誌與尺寸 51
圖 3-2 SD 記憶體配置方式 54
圖 3-3 SD 記憶卡架構 55
圖 3-4 SD Bus 架構 58
圖 3-5 SPI Bus 架構 59
圖 3-6 OCR 結構分析 61
圖 3-7 CMD0 的 CRC7 計算過程 64
圖 3-8 SPI 時序 65
圖 3-9 SPI Power On 時序 66
圖 3-10 SD 記憶卡上電時序 67
圖 3-11 SPI 模式傳送命令副程式 69
圖 3-12 SPI 模式 CMD0 波形 70
圖 3-13 SPI 回應 1b 的時序 72
圖 3-14 SPI 回應 2 的時序 72

圖 3-15 SPI 回應 3 格式 73
圖 3-16 SPI 回應 7 格式 74
圖 3-17 SPI 開始 0xFC 與結束 0xFD 傳送回應的時序 75
圖 3-18 SPI 資料回應格式 75
圖 3-19 SPI 資料回應的時序 76
圖 3-20 SPI 資料錯誤回應 76
圖 3-21 SPI 模式的卡判斷流程 77
圖 3-22 SPI 模式的卡判斷副程式 79
圖 3-23 開啟或關閉 CRC 功能副程式 80
圖 3-24 設定區塊大小副程式 81
圖 3-25 讀取單區塊(CMD17)的操作時序 81
圖 3-26 SPI 模式讀取單區塊副程式 82
圖 3-27 讀取多區塊(CMD18)的操作時序 83
圖 3-28 SPI 模式讀取多區塊副程式 83
圖 3-29 SPI 模式接收資料區塊副程式 84
圖 3-30 寫入單區塊(CMD24)的操作時序 85
圖 3-31 SPI 模式寫入單區塊副程式 86
圖 3-32 寫入多個區塊(CMD25)的操作時序 86
圖 3-33 SPI 模式寫入多區塊副程式 87
圖 3-34 SPI 模式傳送資料區塊副程式 88
圖 3-35 SD Bus 命令格式 89
圖 3-36 SD Bus 回應格式 89
圖 3-37 SD Bus 資料傳輸格式 90
圖 3-38 SD Bus 模式寬資料傳輸啟動副程式 92
圖 3-39 SD Bus 模式命令與回應時序 93
圖 3-40 SD Bus 模式的卡判斷流程 97
圖 3-41 SD 記憶卡的初始化程序 98
圖 3-42 SD 記憶卡的種類與版本判斷程序 99
圖 3-43 讀取 SD 記憶卡暫存器程序 100

圖 3-44	資料傳輸模式狀態圖	101
圖 3-45	SD bus 讀取數據塊操作時序	102
圖 3-46	SD bus 讀取資料磁區副程式	103
圖 3-47	SD bus 寫入資料磁區操作時序	104
圖 3-48	SD bus 寫入資料磁區副程式	106
圖 4-1	讀取 FAT 檔案資料範例	109
圖 4-2	FAT16 系統資料結構	110
圖 4-3	FAT 開機磁區的資料型態	111
圖 4-4	開機磁區資料	113
圖 4-5	FAT 載入 BPB 資訊副程式	116
圖 4-6	導入 FAT 系統副程式	119
圖 4-7	FAT16 的檔案叢集鏈	121
圖 4-8	尋找下一個叢集號碼副程式	121
圖 4-9	根目錄區域磁區資料	122
圖 4-10	讀取文件目錄資訊副程式	124
圖 4-11	資料區域的儲存結構	125
圖 4-12	轉換叢集次區位址副程式	126
圖 4-13	FAT 系統文件資料讀取副程式	128
圖 5-1	WAVE 音訊資料配置結構	133
圖 5-2	RIFF WAVE 檔頭格式	134
圖 5-3	WAVE 播放流程圖	135
圖 5-4	WAVE 檔頭解碼副程式	136
圖 5-5	取樣頻率設定副程式	137
圖 5-6	2 組緩衝區設定副程式	139
圖 5-7	有號數轉換為無號數的副程式	140
圖 5-8	音訊解碼副程式	142
圖 5-9	WAVE 播放功能流程	144
圖 5-10	WAVE 播放功能副程式	144
圖 6-1	ID3V1 資料結構定義	147

圖 6-2 MP3 ID3V1 資料內容 147
圖 6-3 ID3V2.3 結構定義 148
圖 6-4 ID3V2.3 容量計算程式 148
圖 6-5 ID3V2.3 Frame 結構定義 149
圖 6-6 MP3 ID3V2.3 資料內容 149
圖 6-7 MP3 音框結構 150
圖 6-8 MP3 音框頭結構(32 位元) 150
圖 6-9 MP3 音框資料 151
圖 6-10 MP3 旁資訊結構 152
圖 6-11 MP3 主資料結構 152
圖 6-12 MP3 解碼程序 153
圖 6-13 MP3 解碼流程 154
圖 6-14 MP3 解碼副程式 157

表　目　錄

表 2-1　通用輸入/輸出埠的模式設定　11
表 2-2　硬體電路 GPIO 功能應用　12
表 2-3　SPI 模式接腳映射　30
表 2-4　SPI 狀態暫存器　31
表 2-5　SPI 控制暫存器　31
表 2-6　SPI 資料暫存器　32
表 2-7　SPI 控制暫存器　32
表 2-8　SDIO 模組接腳對應　41
表 2-9　SDIO 時鐘控制暫存器　48
表 2-10　SDIO 資料控制暫存器　49
表 2-11　SDIO 命令暫存器　49
表 2-12　SDIO 參數暫存器　50
表 2-13　SDIO 狀態暫存器　50
表 3-1　SD 記憶卡接腳功能描述　57
表 3-2　SD 記憶卡內部暫存器　60
表 3-3　SD 記憶卡命令格式　68
表 3-4　SPI 回應 1 格式　71
表 3-5　SPI 回應 2 格式　73
表 3-6　SD Bus 模式 R1 回應格式　94
表 3-7　SD Bus 模式 R2 回應格式　94
表 3-8　SD Bus 模式 R3 回應格式　95
表 3-9　SD Bus 模式 R6 回應格式　95
表 3-10　SD Bus 模式 R7 回應格式　96
表 4-1　FAT12/16/32 系統比較　107
表 4-2　開機磁區結構　112
表 4-3　FAT 磁碟分區結構　114
表 4-4　FAT16 文件分配表的結構　120

表 5-1	WAVE 文件結構	130
表 5-2	WAVE Format chunk 格式	131
表 5-3	WAVE 編碼方式	131
表 5-4	WAVE fact chunk 結構	132
表 5-5	WAVE data chunk 結構	132
表 5-6	WAVE 結構範例	133
表 6-1	MP3 檔案結構	146
表 6-2	MP3 播放各區塊的執行時間	158

第一章　簡介

1.1 研究背景與動機

近年來多媒體系統已經廣泛的融入生活之中，不論是通訊產品如行動電話或是一般消費性產品如電子書等，都可以看到多媒體系統的應用，使得我們可以容易的獲得影音方面的資訊。而現今應用在多媒體系統上，最主流的微控器為進階精簡指令集微控器（Advanced RISC Machine）簡稱為ARM微控器。其為32位元的微控器，並且廣泛地使用在許多嵌入式系統設計中，主要的設計目標就是提供低耗電高效能的應用特性。

現今多媒體系統應用上，使用最為廣泛的影音格式為MPEG，而其中MP3是大家最常使用到的一種音樂壓縮格式。其為一種數位音訊編碼和破壞性壓縮格式，它被設計用來降低音訊的資料量，而沒有明顯的影響大多數使用者的聽覺感受。

另外由於影音數位化的普及，對於儲存媒體容量的需求也越來越高，一般內建的儲存記憶空間並無法儲存太多的多媒體檔，因此使用擴充的儲存媒體是必需的。目前市面上以SD memory card為主流的擴充儲存裝置，其可格式化為一般作業系統使用的檔案格式，例如FAT檔系統。

因此當我們想要學習嵌入式系統的多媒體應用時，選擇ARM微控器來當主架構與利用SD memory card來做為儲存裝置，並且使用軟體解碼方式來播放MP3音樂檔，如此一來讓我們可以完整的學習到嵌入式系統在多媒體上的應用架構。

1.2 章節安排

本論文的章節安排如下：

第一章說明研究背景與動機。

第二章介紹 STM32F103x 微控器。

第三章介紹 SD 記憶卡。

第四章介紹 FAT 檔案格式。

第五章敘述 Wave PCM 音效格式的播放流程。

第六章簡述 MP3 音效格式的解碼流程。

第七章為結果與未來展望。

第二章　STM32F103x 微控器介紹

本文的多媒體系統使用的是 ARM 微控器來做為核心，我們選用的是意法半導體公司 ST Microelectronics (簡稱 ST)所生產的 STM32F103x 系列 32 位元 ARM 微控器。

STM32F 微控器使用了先進的 32 位元 ARM Cortex™-M3 內核，核心頻率可高達 25MHz，並配備最大可達 512 Kbytes 的快閃記憶體(Flash Memory)與 64 Kbytes SRAM。還具有豐富的內置功能電路，其最高工作頻率可操作在 72MHz，內置電路功能包括有多組的計數器、USB 介面、外部儲存器介面等。

ARM Cortex™-M3 針對中斷回應的問題，在內核上設置了向量中斷控制器(NVIC)，使得基於 Cortex-M3 內核的不同廠牌微控器都具有統一的中斷控制器，使我們進行中斷程式設計與程式移植帶來了很大的便利。

STM32F 微控器還加入了類似於 8 位元處理器的低功耗模式，使整個晶片更具有高性能、低功耗與低電壓特性的優勢 [1]。

本文接下來的章節將針對 STM32F 微控器在多媒體應用所用到的功能進行說明，其他應用的詳細資訊請參考意法半導體公司所公佈的 RM0008 Referance Manual [1] 使用手冊。另外本章節所圖列之範常式式，是從意法半導體公司(ST)所提供的公用程式庫中針對本系統進行修改，公用程式庫可於 ST 公司網頁 http://www.st.com/internet/mcu/product 的 Design Support 中取得。

圖 2-1 為本文所應用的多媒體播放機功能配置，其中 USB 與 USART 功能未導入，而 SD 記憶卡工作在 SPI 模式時，可以使用 LCD 顯示功能，如果工作在 SD Bus 模式時，因為 SDIO 接腳與 LCD 模組使用相對的腳位，所以無法同時使用。另外還有按鍵功能，提供播放動作的選擇與 8 位元的 LED 輸出提供狀態顯示。詳細的硬體電路請參見附錄五，底下將簡述多媒體播放機的功能。

時脈控制(RCC)功能：本系統使用外部晶體振器提供 8MHz 主頻率給微控器，再經過內部相鎖迴路倍頻至 72MHz，供給內置的功能電路使用，詳述內容請參考第 2.1 章節。

通用輸入/輸出埠(GPIO)：本系統使用 GPIOA、GPIOB 與 GPIOC 這三個埠，其需配合各功能電路(如 EXTI、TIMER、SPI、SDIO 等)來設定使用模式。詳述內容請參考第 2.2 章節。

向量中斷(EXTI)：使用 GPIOC 的 PC5~PC8 來做為多媒體播放機的輸入按鍵，詳述內容請參考第 2.3 章節。

通用計數器(TIMER)：本系統使用 TIM3 與 TIM4 這兩個計數器，其中 TIM3 工作在 PWM 模式，用來輸出音樂訊號；而 TIM4 使用在向上計數模式，用來產生音樂訊號的取樣頻率，詳述內容請參考第 2.4 章節。

序列週邊介面(SPI)：用來與 SD 記憶卡溝通的通訊介面，本系統使用 SPI1 模組，並且工作在 Master 模式下，詳述內容請參考第 2.5 章節。

SDIO 模組：與 SD 記憶卡溝通的另一種通訊介面，具有高傳輸率的特性，詳述內容請參考第 2.6 章節。

Audio 電路：分為音訊放大輸出與麥克風輸入兩個部分，詳細電路請參考附錄五圖 5 之應用電路。

系統電源：可由 USB 接頭輸入 5V 電壓，經過 LDO 轉出 3.3V 供給系統使用。或是由 JTAG 轉板直接供給 3.3V。

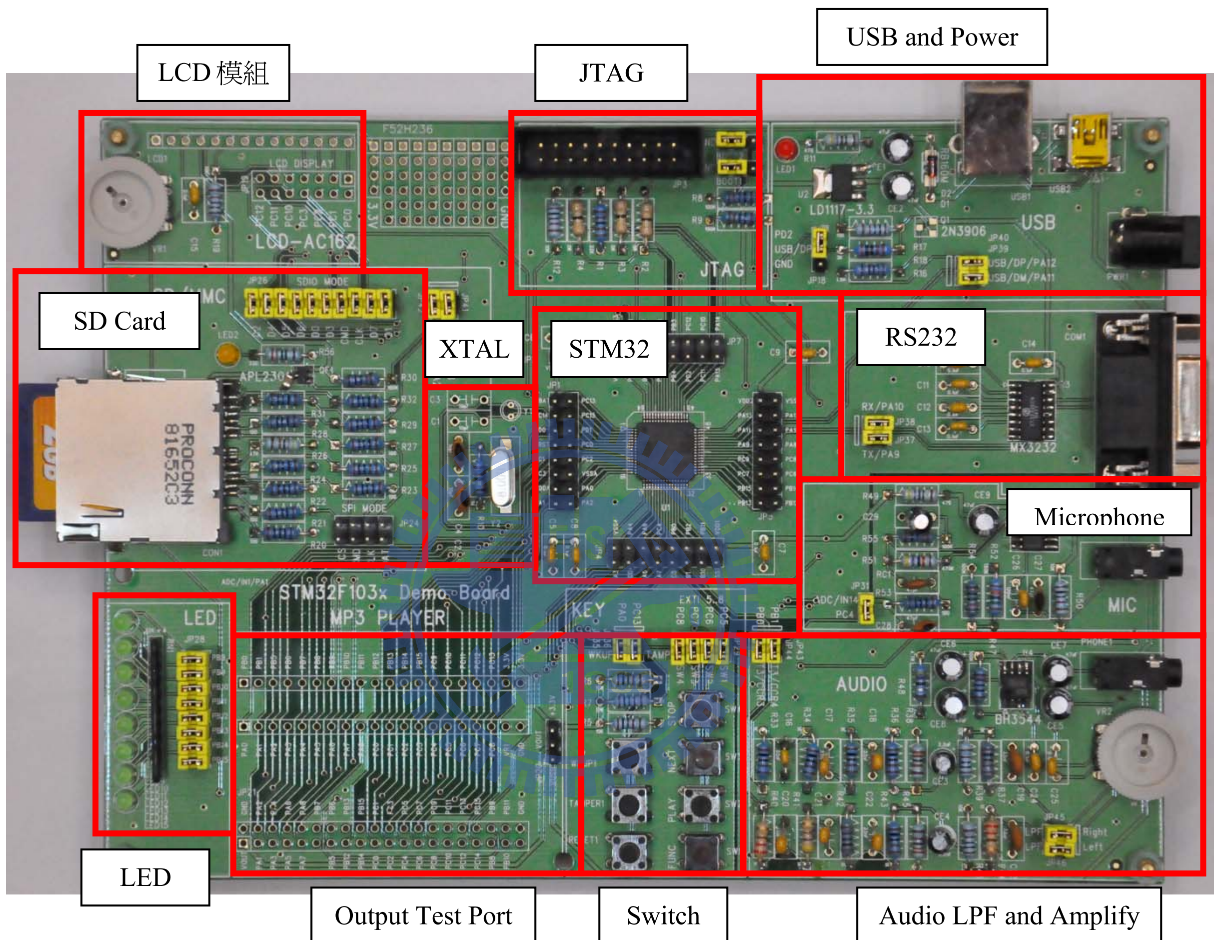


圖 2-1 多媒體系統功能方塊圖

2.1 系統時脈管理

重置與時脈控制(RCC)是用來實現 STM32 微控器的時脈管理，其管理外部、內部和外設的時脈，設置、打開和關閉這些時脈。對於 ARM 微控器來說，CPU 和匯流排以及外設的時脈設置是非常重要的，因為時脈設定錯誤就無法產生正確的時序，組合電路時脈設定錯誤則會造成 I/O 控制混亂。

STM32 微控器有三種不同的時脈來源可被用來驅動系統時脈，外部高速時脈(HSE) ，內部高速時脈(HSI)，相鎖迴路時脈(PLL)。另外還有兩個次級時脈來源：40KHz 低速內部 RC 時脈，32.768KHz 低速外部晶體振盪時脈。使用者可透過多重分頻器設置 AHB、高速 APB2 和低速 APB1 裝置區域的頻率。其中 AHB 和 APB2 裝置區域的最大工作頻率是 72MHz，而 APB1 裝置區域的最高頻率是 36MHz。另外 SDIO 介面的工作頻率固定為 CLK/2，USB介面的工作頻率為48MHz。系統主頻率通過AHB除8倍頻後，提供給 Cortex™系統計數器(SysTick)使用 [1]。

詳細的重置與時脈控制(RCC)資訊請參考意法半導體公司的使用手冊 RM0008 Referance Manual ch.6 [1]。

外部高速時脈(HSE)

外部高速時脈(HSE)可以由兩種時脈型態供應，一個為使用外部晶體振盪器(Crystal)或陶瓷共振器(Ceramic resonator)，另一個為使用外部時脈產生器(External clock)。如果是由外部晶體振盪器來產生時脈，晶體振盪器的頻率範圍為 4 至 16MHz。而在使用外部時脈產生器的情況下，頻率範圍最高可達 25MHz。

圖 2-2 為外部高速時脈的使用方式，圖(a)為使用外部時脈來源時，由 OSC_IN 輸入。圖(b)為使用外部振盪器的連接方式。

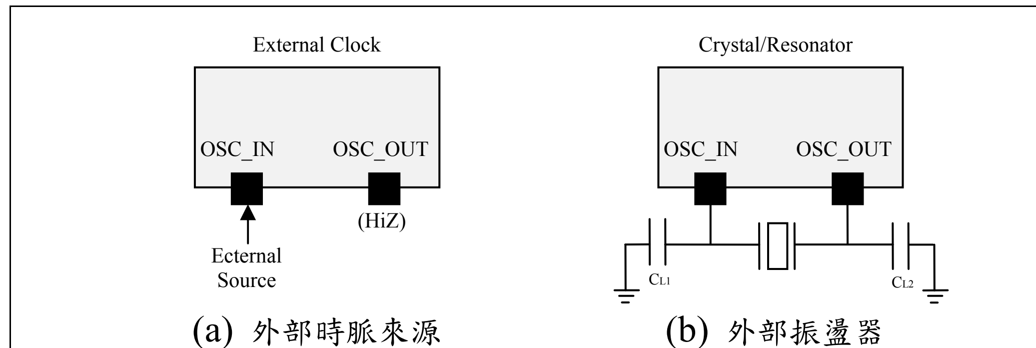


圖 2-2 HSE/LSE 時脈來源 [1]

內部高速時脈(HSI)

內部高速時脈(HSI)是由微控器內部的 RC 振盪器來產生的，它的工作頻率為 8MHz，可直接供應給系統頻率分配器使用，或除 2 後供應給 PLL 當來源。其主要目的為減少外部元件並提供低價位的時脈來源。

相鎖迴路(PLL)

相鎖迴路(PLL)主要是用來倍頻內/外部高速時脈，並提供給系統當作主頻率來源，也會直接提供給高速設備電路如 USB 使用。相鎖迴路的倍頻倍數為 2 到 16 倍，但是輸出頻率最高為 72MHz。

圖 2-3 為相鎖迴路與內部高速時脈的電路，系統頻率(SYSCLK)可選擇由 HSE、HSI 或 PLLCLK 當來源。而 PLLCLK 頻率來源可選擇由 HSI 或 HSE 經過 PLLMUL 倍頻後輸入。

外部低速時脈(LSE)

外部低速時脈(LSE)是由 32.768KHz 的外部晶體或陶瓷共振器所產生，其主要功能為，提供系統 Real-Time Clock 電路所需的低功耗且準確的時脈來源。

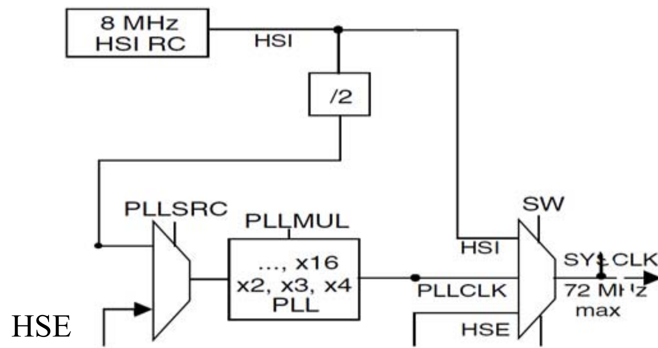


圖 2-3 內部高速時脈(HSI)與相鎖迴路(PLL)電路 [1]

內部低速時脈(LSI)

內部低速時脈(LSI)是由微控器內部的 RC 振盪器來產生的，其頻率大約落在 30KHz 到 60KHz 之間，可以用來取代外部低速時脈的功能。另外也可以在微控器進入待機模式下保持動作，為 Watchdog 電路與自動喚醒電路提供時脈來源。

圖 2-4 為低速時脈的電路，主要做為 RTC 電路的頻率來源，可以選擇由 LSE、LSI 或 HSE 除 128 來做為來源。另外，LSI 又提供給看門狗電路(Watchdog)使用。

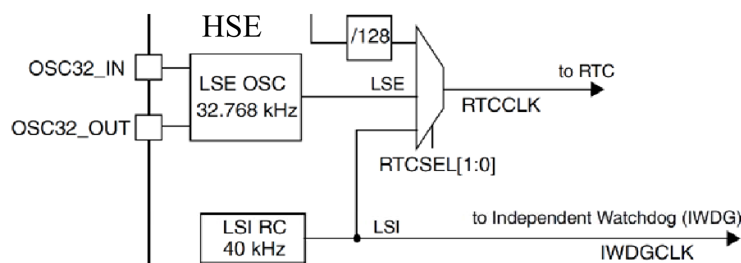


圖 2-4 內/外部低速時脈(LSE/LSI)電路 [1]

STM32 微控器使用重置與時脈控制暫存器(RCC)來設定系統主要的頻率來源，也可以設定與啟動內置功能電路的頻率。其中 RCC_CR 暫存器用來控制系統主頻率來源，RCC_CFGR 暫存器用來設定 PLL、AHB、APB1、

APB2 等區域功能的頻率，而 RCC_AHBENR/APB1ENR/APB2ENR 等暫存器則是用來控制各個功能的頻率啟動。

當系統程式開始執行時，第一步驟需要先設定好 STM32 微控器使用的主頻率來源，再根據頻率來源的頻率來配置內部各裝置的時脈。在本文的應用中，我們使用外部高速時脈(HSE)功能連接 8MHz 振盪器，外部低速時脈(LSE) 連接 32.768KHz 振盪器，設定 PLL 頻率為 72MHz(9 倍頻)，來當作主系統時脈(SysCLK)，設定 AHB 分頻器後的 HCLK 為 72MHz，高速 APB2 區域的時脈為 72MHz，低速 APB1 區域的時脈為 36MHz。

圖 2-5 為本系統的應用電路，PD0 與 PD1 連接 8MHz 外部高速時脈；PC14 與 PC15 連接 32.768KHz 外部低速時脈(LSE)。

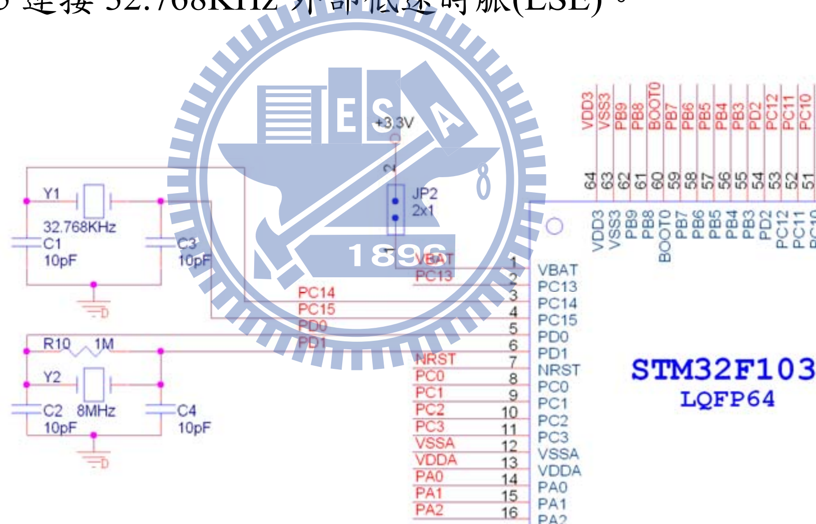


圖 2-5 主頻率來源電路

我們可以使用兩種方式來設定 STM32 微控器的頻率來源，圖 2-6 為設定 STM32 微控器頻率來源與頻率配置 RCC_Initial ( ) 的副程式，本系統使用 8MHz HSE 提供給 PLL，並且倍頻 9 倍至 72MHz，當設置好參數後，將 HSE_ON 與 PLL_ON 位元設為 1，以啟動 HSE 與 PLL 電路，最後等待 HSE 與 PLL 發出準備完成的狀態旗標，以表示頻率正常。

程式名稱：RCC_Initial ( )	輸　入：HSE，PLL 頻率來源
功能敘述：設定系統頻率	輸　出：無

```
viod RCC_Initial (void)
{   RCC->CFGR  |= (7<<18);              // PLLMUL[3:0] : PLL * 9 = 72MHZ
    RCC->CFGR  |= (0<<17);              // HSE clock not divided
    RCC->CFGR  |= (1<<16);              // PLLSRC = 1 : HSE INPUT
    RCC->CFGR  |= (0<<11);              // PPRE2[2:0] APB2CLK=HCLK
    RCC->CFGR  |= (4<<8);               // PPRE1[2:0] APB1CLK=HCLK/2
    RCC->CFGR  |= (0<<4);               // HPRE[2:0] AHBCLK=SysCLK
    RCC->CFGR  |= (2<<0);               // SW[1:0] : SYSTEM CLK = PLL CLK
    RCC->CR    |= (1<<16);              // 開啟 HSE
    while ((RCC->CR & (1<<17))==0);     // 等待 HSE ready
    RCC->CR    |= (1<<24);              // 開啟 PLL
    while ((RCC->CR & (1<<25))==0);     // 等待 PLL ready
}
```

圖 2-6 RCC 初始設定程式

圖 2-7 為使用編譯軟體所提供的 STM32_Init.c 程式庫，其內建了 RCC 暫存器的設定選單，利用下拉選單的方式來設定系統頻率來源與 PLL 倍頻倍數，以及各區域的工作頻率。

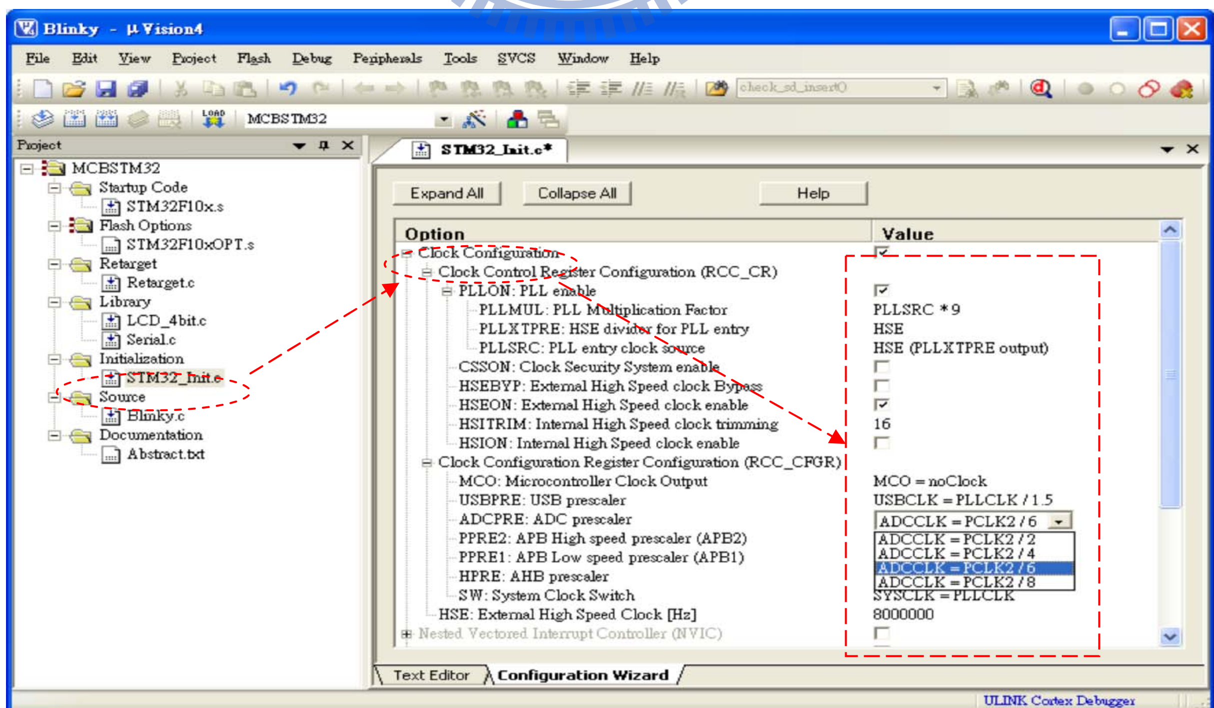


圖 2-7 編譯軟體的 RCC 設定

2.2 通用輸入/輸出埠(General Purpose I/Os - GPIOs)

通用輸入/輸出埠(簡稱GPIO)是STM32F微控器的基本輸入/輸出功能，每一組GPIO都有16根接腳，並且每根接腳都可以獨立的由軟體設置成不同的工作模式，詳細的通用輸入/輸出埠(GPIO)說明請參考意法半導體公司的使用手冊RM0008 Referance Manual ch.7 [1]。

GPIO的模式選擇是由GPIO_CRH與GPIO_CRL暫存器來設置的，其設定方式如表2-1所示，CFNx[1:0]用來設定輸入/輸出模式，MODEx[1:0]用來設定接腳的最高輸出頻率。

表 2-1 通用輸入/輸出埠的模式設定 [1]

<table>
<tr><th colspan="2">Configuration mode</th><th>CFN1</th><th>CFN0</th><th>MODE1</th><th>MODE0</th><th>PxODR</th></tr>
<tr><td rowspan="2">General purpose Output</td><td>Push-pull</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">01
10
11</td><td>0 or 1</td></tr>
<tr><td>Open-drain</td><td>1</td><td>0 or 1</td></tr>
<tr><td rowspan="2">Alternate function output</td><td>Push-pull</td><td rowspan="2">1</td><td>0</td><td>X</td></tr>
<tr><td>Open-drain</td><td>1</td><td>X</td></tr>
<tr><td rowspan="4">Input</td><td>Analog input</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">00</td><td>X</td></tr>
<tr><td>Input floating</td><td>1</td><td>X</td></tr>
<tr><td>Input pull-down</td><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>Input pull-up</td><td>1</td></tr>
<tr><th colspan="2">MODE[1:0]</th><th colspan="5">Meaning</th></tr>
<tr><td colspan="2">00</td><td colspan="5">Reserved</td></tr>
<tr><td colspan="2">01</td><td colspan="5">Max. output speed 10MHz</td></tr>
<tr><td colspan="2">10</td><td colspan="5">Max. output speed 2MHz</td></tr>
<tr><td colspan="2">11</td><td colspan="5">Max. output speed 50MHz</td></tr>
</table>

本系統硬體電路的GPIO功能使用如表2-2所列，PA[2]為SD卡的電源控制訊號，PA[8]為偵測SD卡的插入，PA[4:7]為SPI模組接腳，PB[1:2]為TIM3的PWM輸出，PB[8:15]為LED燈號輸出，PC[4]為麥克風的ADC輸入，PC[5:8]為播放功能的中斷輸入；而沒有用到的接腳，將其設為接腳設定為

浮接輸入模式，避免意外輸出造成電路發生錯誤。

表 2-2 硬體電路 GPIO 功能應用

GPIO	Mode	Discription
PA[2] (參考附錄五，圖 4)	Push-pull output	SD card 電源控制 0 : power on 1 : Power off
PA[8] (參考附錄五，圖 4)	Floating input	SD card 插入偵測 1 : No SD card 0 : SD card insert
PA[4:7] (參考附錄五，圖 4)	Alternate function output	SD card SPI 模式接腳 PA4 : CS PA5 : SCLK PA6 : DAT0 PA7 : CMD
PB[1:2] (參考附錄五，圖 5)	Alternate function output	TIM3 PWM 輸出接腳
PB[8:15] (參考附錄五，圖 4)	Push-pull output	8 位元 LED 輸出
PC[4] (參考附錄五，圖 5)	Analog input	麥克風類比訊號輸入 PC4 : ADC1 channel 14
PC[5:8] (參考附錄五，圖 4)	Pull up/down input	播放功能按鍵 PC5 : Function PC6 : Play PC7 : Next PC8 : Stop
PC[8:12] PD[2] (參考附錄五，圖 4)	Alternate function output	SD card SDIO 模式接腳 PC8 : DAT0 PC9 : DAT1 PC10 : DAT2 PC11 : DAT3 PC12 : CLK PD2 : CMD

我們可以使用編譯軟體所提供的STM32_Init.c程式庫，依照表2-2所列之功能，設置GPIO的工作模式。圖2-8為使用下拉選單的方式來設置GPIO的工作模式。

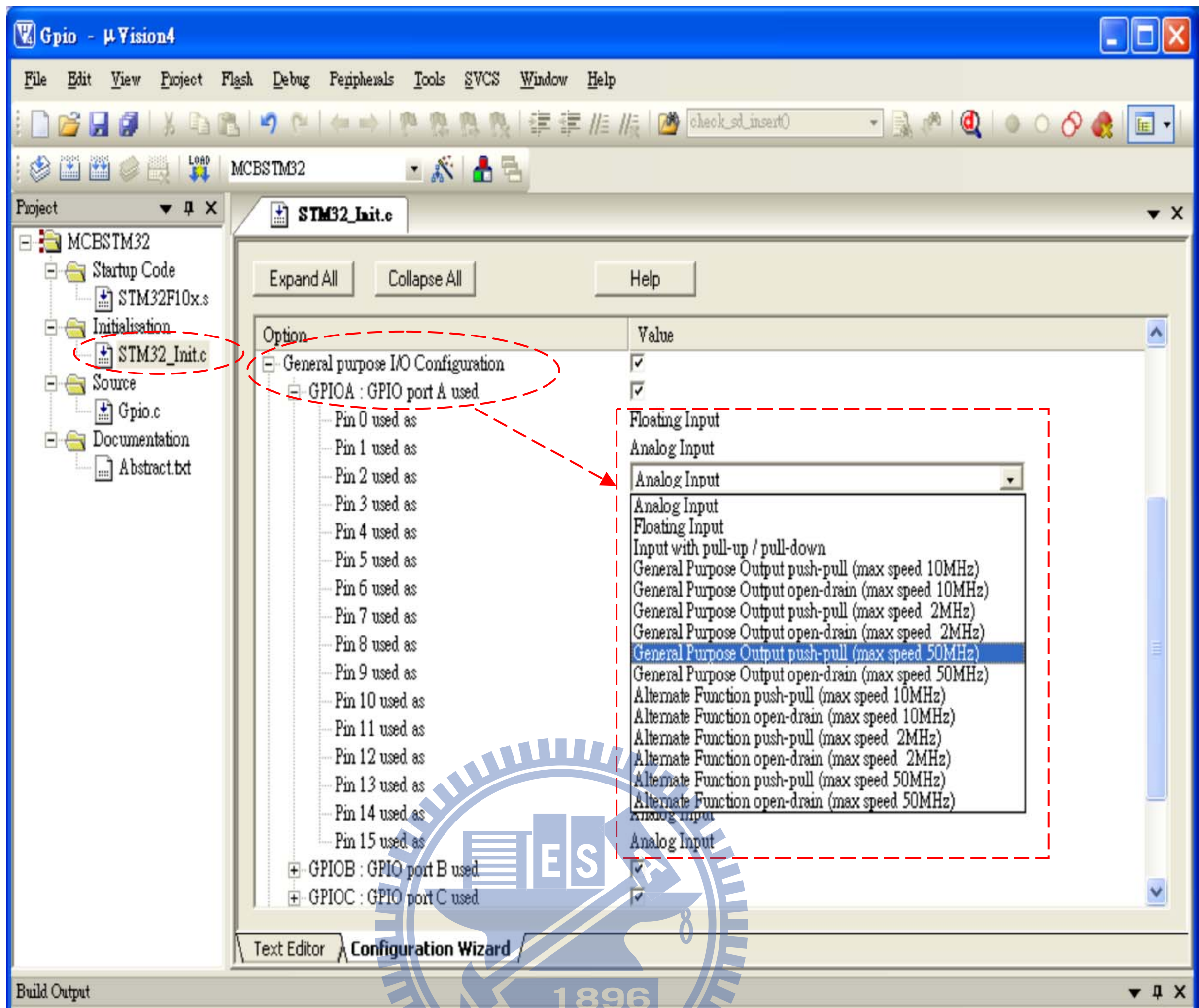
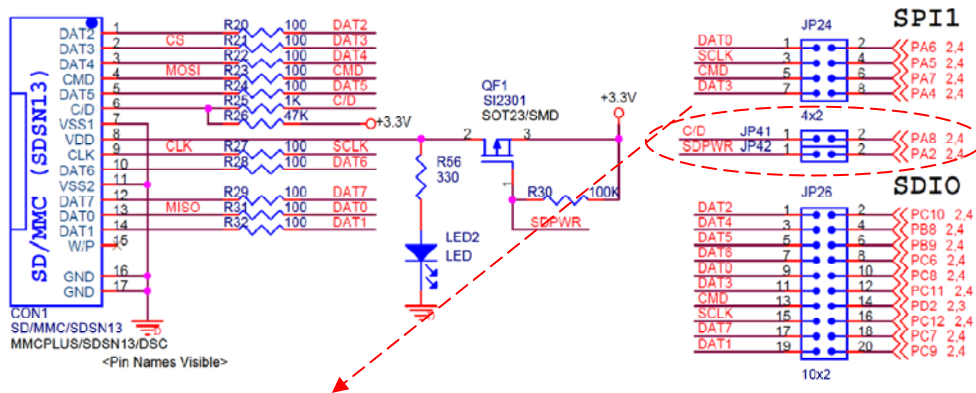


圖 2-8 編譯軟體的 GPIO 設定

舉例本系統的SD記憶卡電源控制與卡片偵測應用電路，來說明GPIO的基本輸入與輸出設定與使用。圖2-9的電路中，透過PA[2]控制P-MOSFET電源開關，當輸出為低電位時，P-MOSFET導通並供給電源給SD記憶卡。若輸出為高電位時，則關閉電源。

PA[8]用來偵測SD記憶卡是否插入或拔除，在記憶卡插槽中，有一根C/D腳位元是用來偵測卡片狀態。當卡片插入插槽時，該C/D腳位會被拉到低電位；相反的，當卡片拔除或沒有插好的狀況下，該腳位會被拉到高電位。



Function	Pin	I/O	High State	Low State
SD Power Control	PA2	Output	Power Off	Power On
SC Card Detect	PA8	Input	No SD card	SD Card Inserted

圖 2-9 SD 記憶卡電源控制與插入偵測電路

圖2-10為本系統的電源控制與卡片偵測範常式式，使用GPIOA_IDR第8位元的狀態，來偵測SD卡。使用GPIOA_ODR第2位元，來決定電源的供給。

程式名稱：**check_sd_insert ( ), SD_power_on ( ) SD_power_off ( )**	輸　入：**PA8 偵測卡片**
功能敘述：**偵測卡片插入，打開 SD card 電源，關閉 SD card 電源**	輸　出：**PA2 控制電源開關**

```
#define SD_PWRON()    GPIOA->ODR &= ~(1<<2)      // PA2:Low = Power On
#define SD_PWROFF()   GPIOA->ODR |= (1<<2)       // PA2:High = Power Off
#define SD_INS()      GPIOA->IDR &= (1<<8)       // PA8:Low = SD insert
u8 check_sd_insert (void)
{    if (SD_INS())                               // 偵測卡片是否插入
         Stat = STA_NOSDCARD;                    // 將狀態設為no SD card
     else         Stat = STA_SDCARD;             // 將狀態設為SD card insert
     return Stat;          }
u8 SD_power_on (void)
{    SD_power_off ();                            // 關閉SD card電源
     if (check_sd_insert () == STA_NOSDCARD)     // 偵測卡片是否插入
         return 0;                               // 無卡片，返回 0
     Delay (140);                                // 延遲140m sec
     SD_PWRON ();                                // 打開SD card電源
     return 1;             }
void SD_power_off (void)
{    SD_PWROFF ( );        }                     //關閉SD card電源
```

圖 2-10 SD 記憶卡電源控制與插入偵測程式

2.3 外部中斷與事件(External Interrupts and Events - EXTI)

STM32F 微控器的向量中斷控制器(NVIC)提供了 60 個可遮罩的中斷通道，和 16 個可程定優先等級的低延遲異常和中斷處理。外部中斷/事件控制器(簡稱 EXTI)是由 19 個產生事件/中斷要求的邊沿檢測器所組成的，每個輸入可以獨立地配置其輸入類型(邊沿或準位)和對應的觸發事件(上升沿或下降沿或者雙邊沿都觸發) [1], [2]。

詳細的向量中斷控制器(NVIC)資訊請參考意法半導體公司的使用手冊 RM0008 Referance Manual ch.8 [1]，以及 ARM 公司的 Cortex-M3 Technical Referance Manual [2]。

圖2-11為本系統的外部中斷輸入電路，使用了PC[5:8]來做為播放動作的按鍵輸入，PC[5]為主功能切換按鍵，PC[6]為播放功能按鍵，PC[7]為選擇歌曲按鍵，PC[8]為停止播放按鍵。

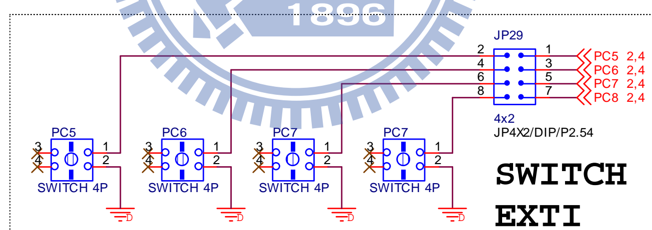


圖 2-11 外部中斷輸入電路

圖 2-12 為中斷輸入設定的 EXTI_Init ( )副程式，設定 AFIO_EXTICRx 暫存器，選擇 GPIOC 埠的 PC[5:8]為中斷輸入來源，EXTI_IMR 暫存器用來開啟中斷功能，EXTI_FTSR 暫存器則是設定中斷觸發功能為向下邊緣觸發。由於 EXTI 功能的中斷觸發 5 至中斷觸發 9，使用同一個中斷向量 EXTI9_5，因此在執行中斷向量副程式時，需要另外判斷是那一個接腳觸發中斷。

程式名稱：EXTI_Init ( )	輸　入：中斷訊號 PC8~5
功能敘述：設定 EXTI 中斷 (PC5 .. PC8)	輸　出：EXTI9~5_IRQ 中斷向量

```
void EXTI_Init(void)
{  RCC->APB2ENR    |= (1<<0);                        // AFIO enable
   AFIO->EXTICR[2] = 0x0002;                         // Set external interrupt source as PC8
   AFIO->EXTICR[1] = 0x2220;                         // Set external interrupt source as PC7-PC5
   EXTI->IMR       = 0x000001E0;                     // EXTI mask enable TR8-TR5
   EXTI->FTSR    = 0x000001E0;                       // Set Falling edge
   NVIC->Enable[0] |= (1 << (EXTI9_5_IRQChannel & 0x1F));
}
```

圖 2-12 EXTI 設定副程式

圖2-13為使用編譯軟體所提供的程式庫，在Configuration Wizard中選擇EXTI暫存器設定的選單，針對系統應用的中斷輸入接腳，直接選擇所須要的模式與觸發條件。

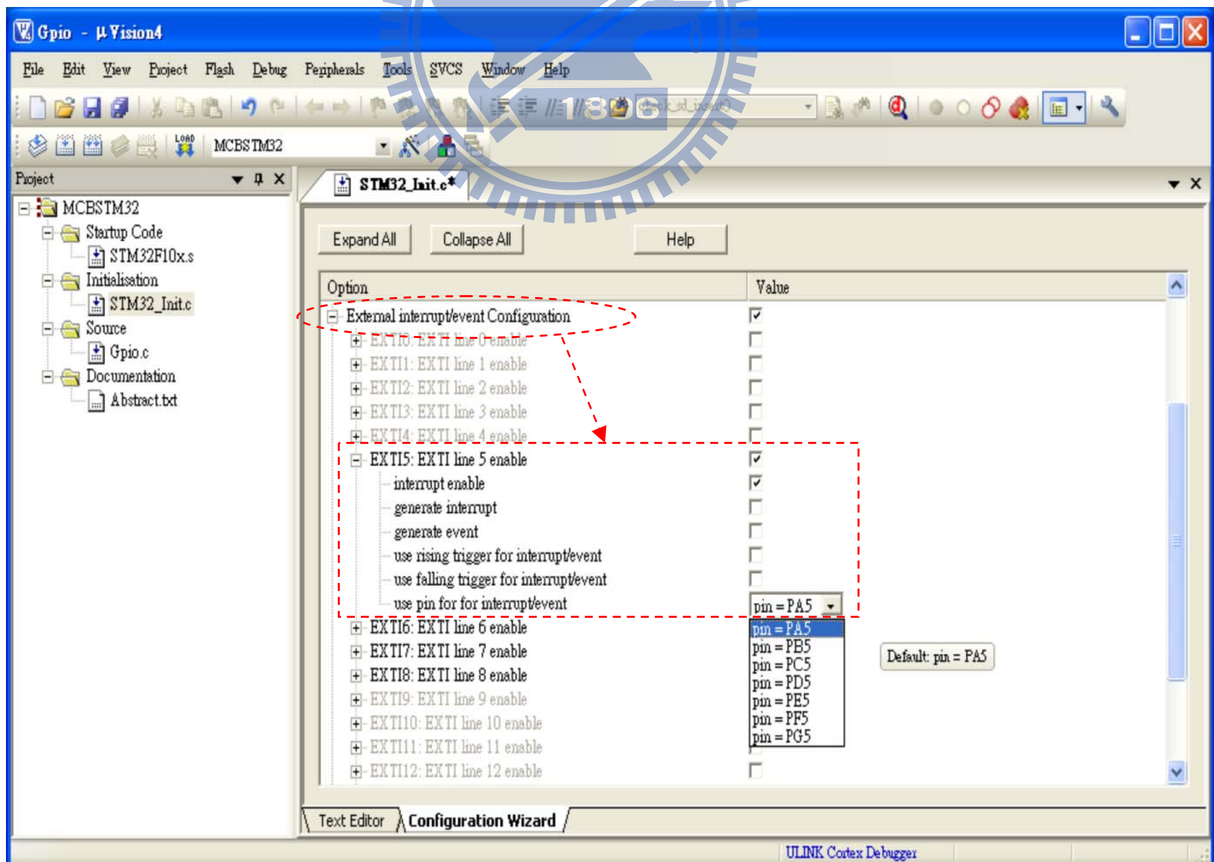


圖 2-13 編譯軟體的 EXIT 設定

2.4 通用計數器(General Purpose Timer - TIMx)

STM32F微控器的通用計數器(簡稱TIM)，是由預分頻器與自動裝載計數器所構成，可以設定為向上計數、向下計數或者雙向計數等三種模式。並且應用在多種場合，如產生定時中斷、測量輸入信號的脈衝寬度和產生PWM輸出波形等。

本文所用到的TIM工作模式為向上計數模式與脈衝寬度調變(PWM)模式，圖2-14為TIM模組的工作架構，我們將TIM4設定為向上計數模式，主要應用在播放音樂時，用來產生與取樣頻率(Sample frequency)一致的定時中斷，該中斷副程式會在每一次中斷發生時，輸出一筆音樂資料 [1]。

另外將TIM3設定為脈衝寬度調變(PWM)模式，由於一般低價位的微控器大都沒有內置數位轉類比(DAC)功能，因此我們將數位音樂資料透過PWM方式來產生相對應的波形輸出，再經由低通濾波器(LPF)來將PWM波形轉為類比訊號。

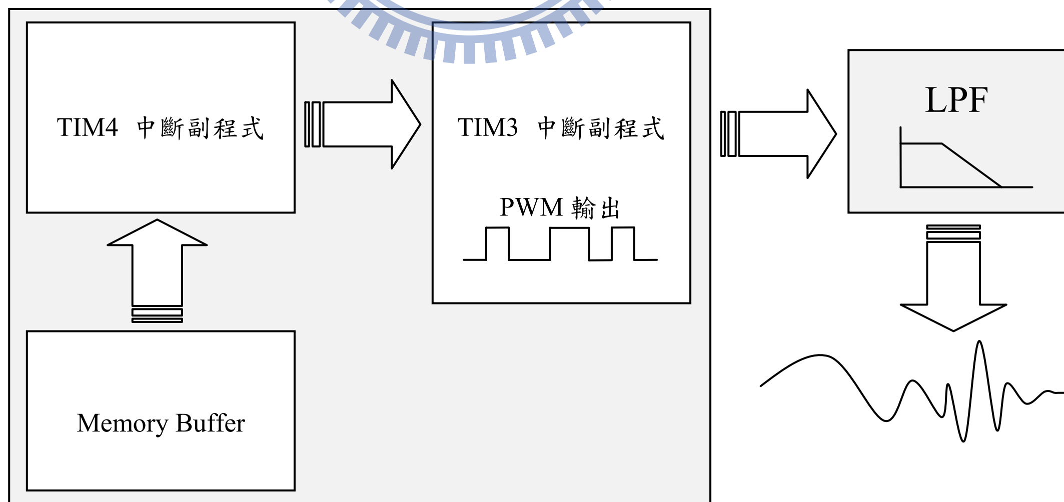


圖 2-14 TIM 中斷處理架構

詳細的通用計數器(TIM)資訊請參考意法半導體公司的使用手冊RM0008 Referance Manual ch.13 [1]。

2-4-1 向上計數模式(Up counting mode)

在向上計數模式中，計數器從0計數到自動載入值(TIM_ARR暫存器)後，會產生一個溢出事件，並且重新從0開始計數，圖2-15為計數器的向上計數模式時序圖，其中計數器頻率(CK_CNT)為系統頻率(CK_INT)除4，TIM_ARR暫存器設定為36，當計數器數到設定值(= 36)時，會發出溢位與更新事件訊號，如果有開起中斷，則同時會發出中斷旗標。

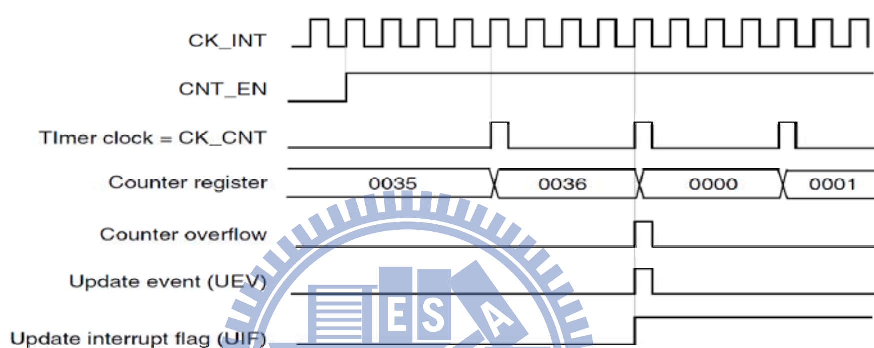


圖 2-15 向上計數模式時序 [1]

TIM計數器的主要工作頻率(f_{INT})是從APB1區域分頻而來的，計數器會先透過TIM_PSC暫存器的預分頻因數，來改變計數器的計數頻率(f_{CNT})。當TIM計數器計數到自動裝載暫存器(TIM_ARR)內的數值時，會發出TIM中斷請求。因此我們可以算出中斷訊號發出的頻率(f_{TIM})。

$$f_{INT} = f_{APB1} \tag{1}$$

$$f_{CNT} = f_{INT} \div (TIM_PSC[15:0] + 1) \tag{2}$$

$$f_{TIM} = f_{CNT} \div (TIM_ARR[15:0] + 1) \tag{3}$$

其中f_{APB1}為功能裝置橋接器的頻率，本系統為72MHz。f_{INT}為TIM4計數器的主頻率，由f_{APB1}提供。f_{CNT}為TIM4計數器經過預分頻器後的頻率。f_{TIM}為TIM4計數器最終輸出頻率。

圖 2-16 為編譯軟體所提供的 STM32_Init.c 程式庫，在 Configuration Wizard 中選擇 TIM4 計數器暫存器設定的選單，針對 TIM4 計數器的設定，直接使用下拉選單的方式選擇所須要的中斷頻率與操作模式。

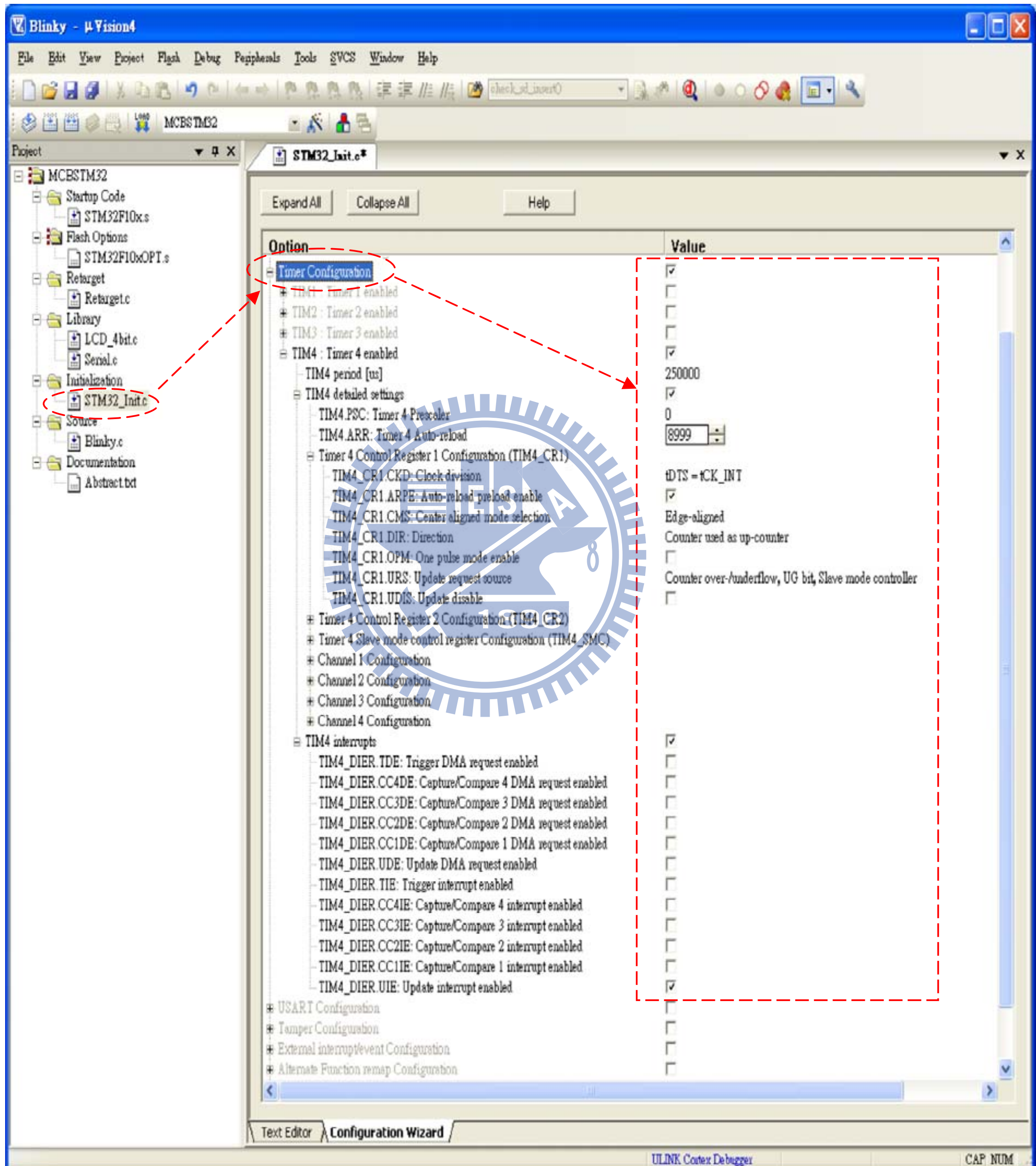


圖 2-16 編譯軟體的 TIM4 設定

2-4-2 脈衝寬度調變模式(PWM mode)

脈衝寬度調變(PWM)是一種將數位信號藉由高解析度計數器的應用，圖 2-17 為方波週期內工作週期的波形與計算方式，(4)為工作週期的算式，工作週期可以用來對應具體的類比信號。PWM 信號仍然屬於數位處理的範圍，因為在給定的任何時刻，PWM 輸出只會有 0 或 1 的狀態。透過 0 和 1 所對應的直流電壓準位，(5)為計算工作週期和電壓的關係式，可將數位的方波換算成類比訊號。理論上只要頻寬足夠，任何類比訊號都可以脈衝寬度調變(PWM)的方式表示出來 [1], [3]。

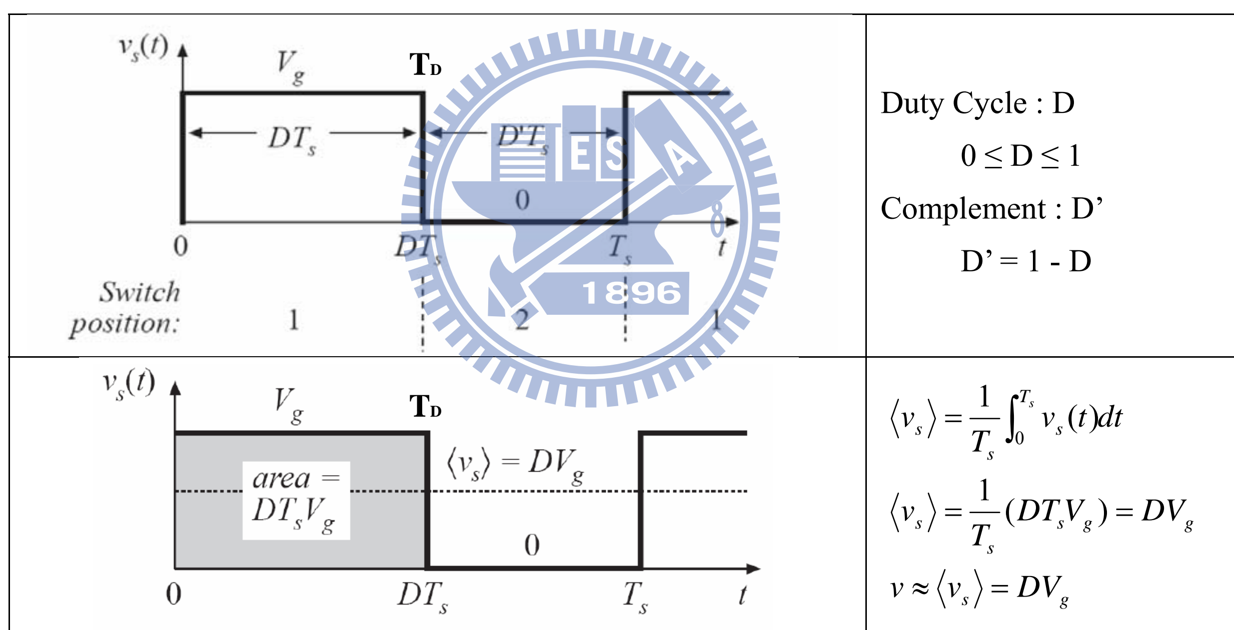


圖 2-17 PWM duty cycle 波形 [3]

$$D = (1 - T_D)/T_S \tag{4}$$

$$V_s = D \times V_g \tag{5}$$

其中D為PWM的工作週期。T_D為週期內導通時間。T_S為整個週期時間。V_s為平均電壓。V_g為輸入電壓。

圖2-18為TIM計數器的脈衝寬度調變(PWM)模式的波形，輸出一個由(6)計算工作頻率的TIM_ARR暫存器設定值，由(8)計算週期的TIM_CCR暫存器設定值的PWM信號。另外，TIM計數器的PWM有兩種工作模式，在PWM模式1下，當計數器 ≥CCR時，PWM輸出為HIGH；在PWM模式2下，當計數器 ≥CCR時，PWM輸出為LOW。

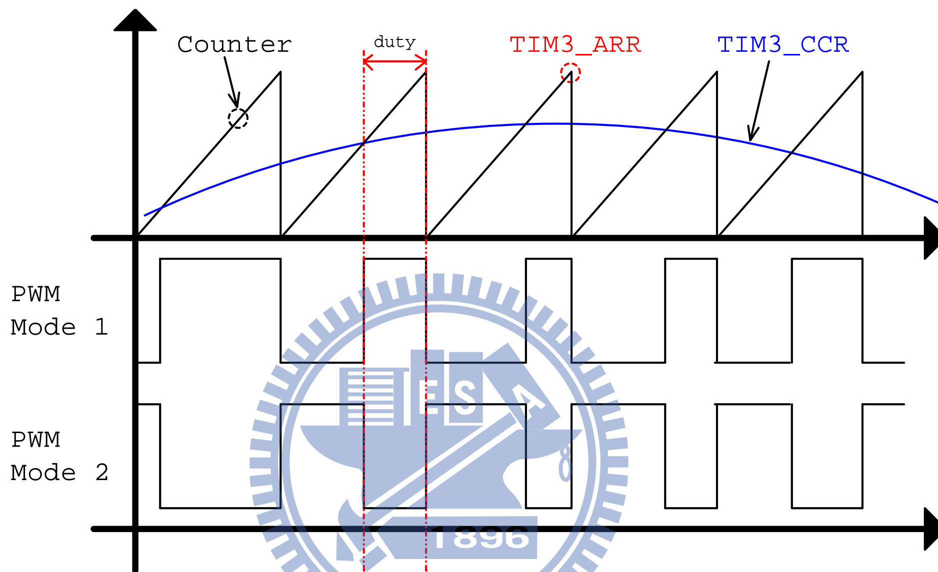


圖 2-18 PWM 模式 1 與模式 2 波形

脈衝寬度調變(PWM)模式的工作頻率，可以由TIM_PSC暫存器與TIM_ARR暫存器來決定，其中TIM_ARR暫存器又可以決定脈衝寬度調變的解析度，再透過比較TIM_CCR暫存器的設定值，得到所要的工作週期。

$$f_{PWM} = f_{INT} \div [(TIM_PSC + 1) \times (TIM_ARR + 1)] \tag{6}$$

$$N_{PWM} = TIM_ARR + 1 \tag{7}$$

$$D = TIM_CCR \div (TIM_ARR + 1) \tag{8}$$

其中f_{PWM}為PWM模組的工作頻率。N_{PWM}為PWM模組的解析度。

圖2-19為解析度設為256階，工作週期為CCR=4與CCR=255的輸出波形，當計數器值與CCR減1相等時，會發出Capture/Compare中斷訊號，並且PWM輸出為低電位，直到計數器值清除為0時，PWM才會輸出為高電位。

當 TIM_PSC=0，TIM_ARR=255(0xFF)時，可得到 PWM 工作頻率(f_{PWM})與 PWM 週期解析度(N_{PWM})為：

$$f_{PWM} = 72MHz \div [(0+1) \times (255+1)] = 281.25KHz$$

$$N_{PWM} = 255 + 1 = 256$$

當 CCRx 設為 4 時，工作週期 D = 4 / (255 + 1) = 1.56%

當 CCRx 設為 255 時，工作週期 D = D = 255 / (255 + 1) = 99.6%

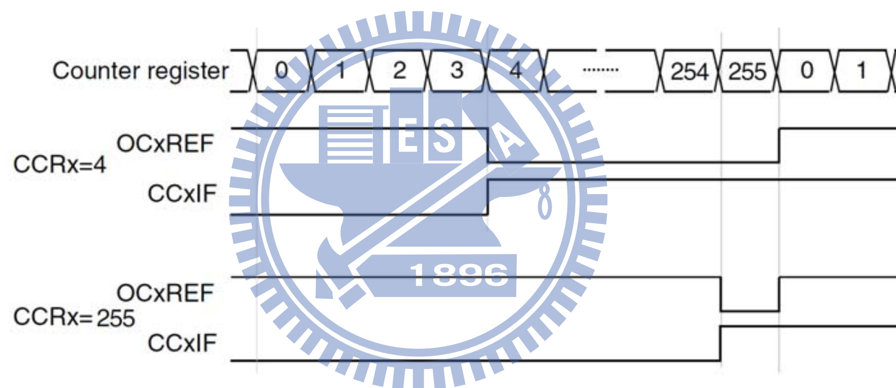


圖 2-19 CCR 為 4 與 255 的 PWM 輸出波形 [1]

圖2-20為TIM3_Init()副程式，設定TIM3計數器來輸出8位元解析度，且工作頻率大於44.1KHz的PWM波形，設置需要的PWM工作頻率與解析度，並開啟TIM3中斷向量來更新CCR值。一般常用的DAC輸出有8、12和16位元，在此三種解析度的狀況下，PWM的工作頻率為：

TIM3_ARR	= $2^{解析度位元數}$	= {256，4096，65536}
$F_{PWM(8bit)}$	= 72MHz / 256	= 281.25KHz
$F_{PWM(12bit)}$	= 72MHz / 4096	= 17578.125Hz

$$F_{PWM(16bit)} = 72MHz / 65536 = 1098.6Hz$$

當PWM的工作頻率大於最大取樣頻率時，PWM輸出才能夠真實反應訊號，因此只有在解析度為8位元的條件下，PWM的工作頻率才會大於44.1KHz，所以TIM3_ARR暫存器的設定值為$2^8 - 1 = 255$。

程式名稱：**TIM3_Init ( )**	輸　入：**TIM3_PSC，TIM3_ARR**
功能敘述：設定 **TIM3 PWM** 模式	輸　出：**TIM3_IRQ** 中斷向量

```
void TIM3_Init (void)
{
    RCC->APB1ENR    |= (1<<1);            // TIM3 enable
    TIM3->PSC       = 0x00;               // CK_CNT = 72MHz / (0 + 1) = 72MHz
    TIM3->ARR       = 0xFF;               // CK_TIM = CK_CNT / 255+ 1) =281.25KHz
    TIM3->CCMR1     = 0x0000;             // OC1M/2M = FROZEN
    TIM3->CCMR2     |= (6<<4);            // OOUPUT COMPARE 3 MODE = PWM MODE 1
    TIM3->CCMR2     |= (7<<12);           // OOUPUT COMPARE 4 MODE = PWM MODE 2
    TIM3->CCER      |= (1<<12);           // CC4E enable
    TIM3->CCER      |= (1<<8);            // CC3E enable
    TIM3->CR1       |= (1<<2);            // UPDATE REQUEST SOURCE ENABLE
    TIM3->DIER      |= (1<<0);            // UPDATE INTERRUPT ENABLE
    NVIC->ISER[0]   |= (1 << (TIM3_IRQChannel & 0x1F));
}
```

圖 2-20 TIM3 計數器設定

圖 2-21 為將一小段音訊資料放在微控器的記憶體中，透過模擬軟體將資料由 TIM4 產生 8KHz 的取樣頻率來輸出到 TIM3 的 PWM 模組，設定 CCR3 工作在模式 1，CCR4 工作在模式 2，我們可以在 PC[0:1]得到每筆資料相對應的 PWM 輸出波形。

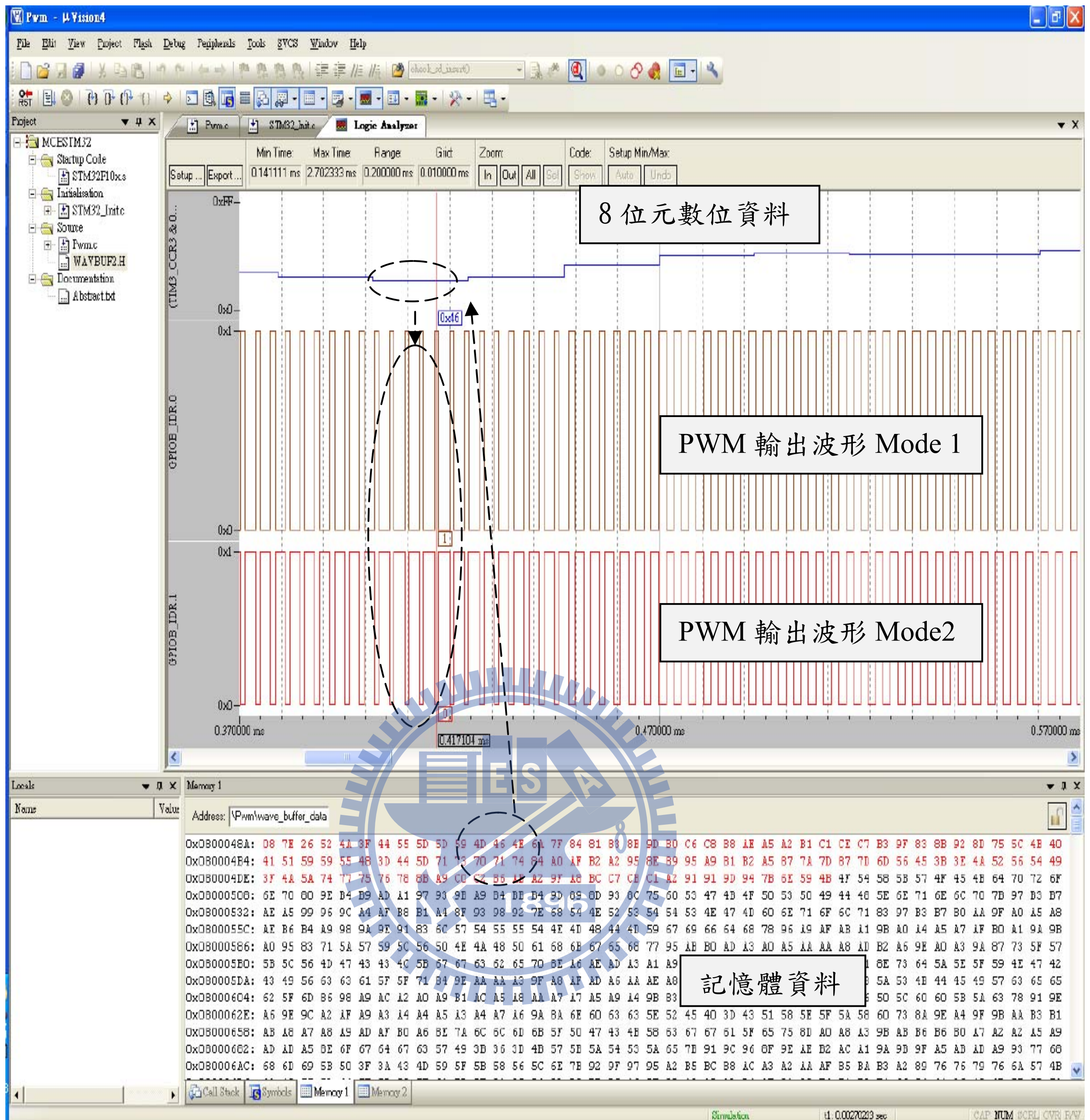


圖 2-21 模擬 PWM 輸出波形

2-4-3 PWM 低通濾波器

本系統使用 PWM 輸出模式來取代 DAC 功能，用以降低硬體成本與減少 DAC 控制流程，其工作原理為將數位音樂資料轉換為脈衝波寬調變的數位訊號，然後經由低通濾波器來將高頻的 PWM 工作頻率消除，保留原始的低頻音樂訊號。聲音頻率的範圍在 20Hz 到 20KHz 左右，而 PWM 的工作頻率一般為聲音頻率的 10 倍以上(至少 200KHz)，本系統的 PWM 工作頻率為 281.25KHz。

使用 PWM 模式來輸出音樂訊號屬於 D 類放大器，一般將輸出訊號通過低通濾波器後就可直接接到喇叭播放，由於本系統使用的 STM32 微控器 PWM 輸出為訊號等級(最大電流 25mA 輸出)，無法直接驅動播放裝置，因此需要使用 OP 放大器來將訊號的功率放大，圖 2-22 為本系統的放大器與濾波器電路。

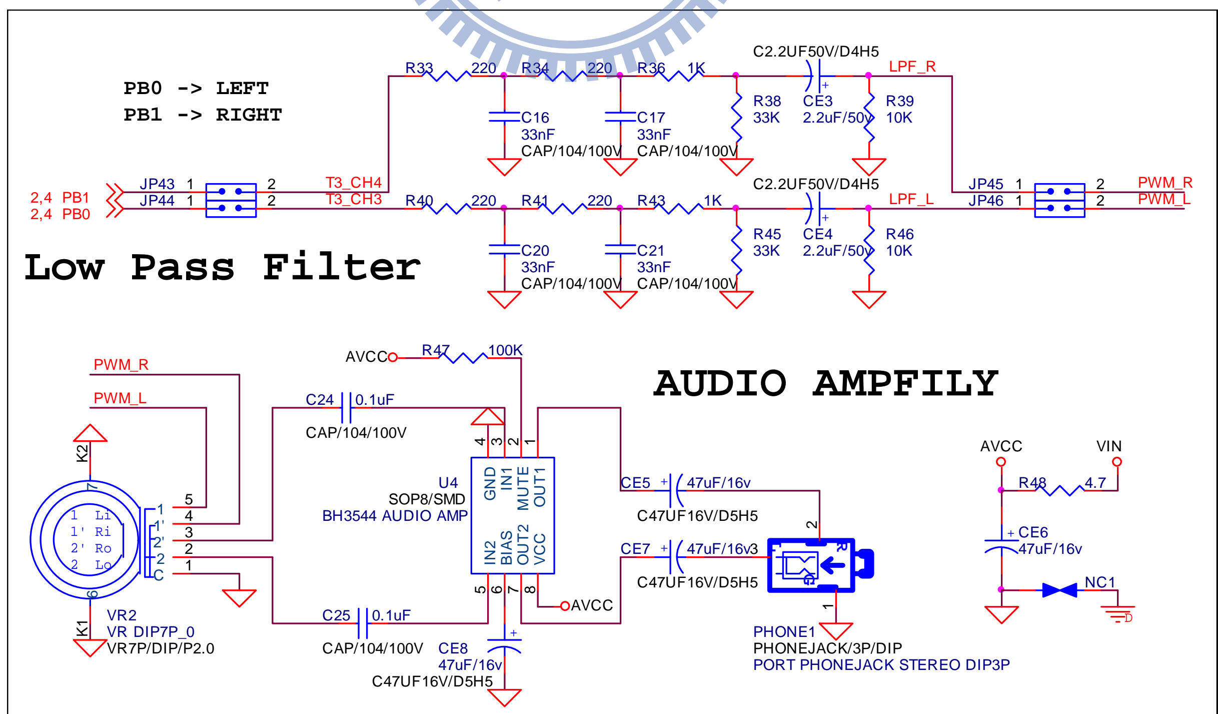
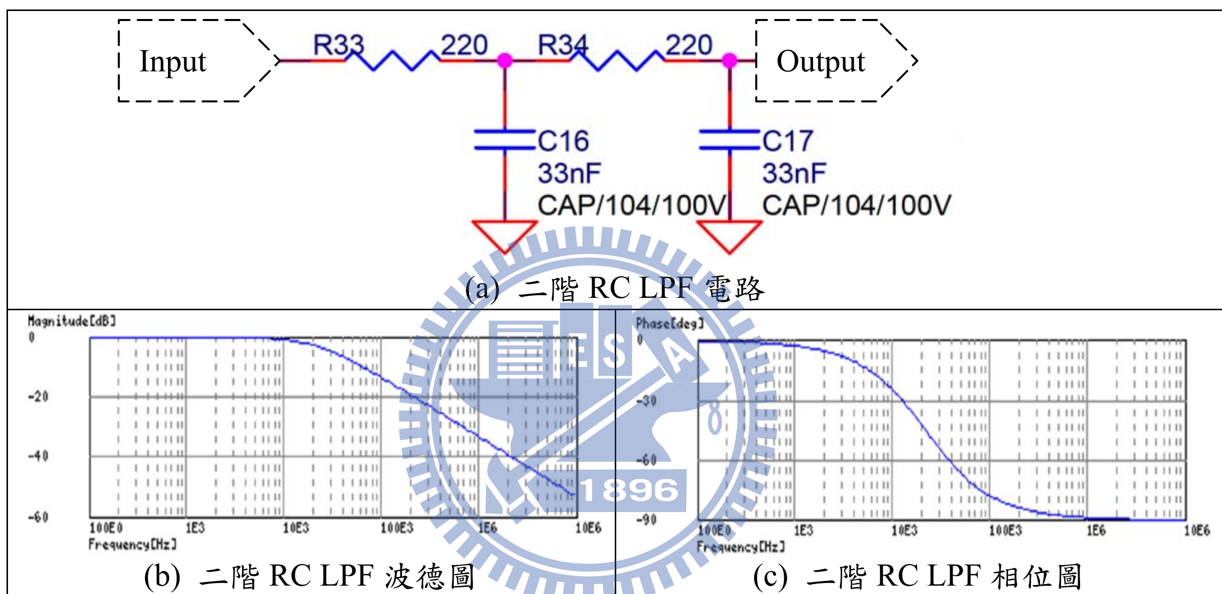


圖 2-22 放大器與低通濾波器電路

我們將濾波器電路拆分為兩個部分來進行分析，圖 2-23 (a)為用來將高頻的 PWM 頻率濾除的二階低通濾波器電路，使用 R 為 220Ω與 C 為 33nF 時，其轉移函數為(9)，-3dB 截止頻率為 21.92KHz，(b)為二階 RC LPF 的波德圖，(c)為其相位圖。

$$G_{LPF}(s) = \frac{18972595982.4}{s^2+413223.14s+18972595982.4} \tag{9}$$



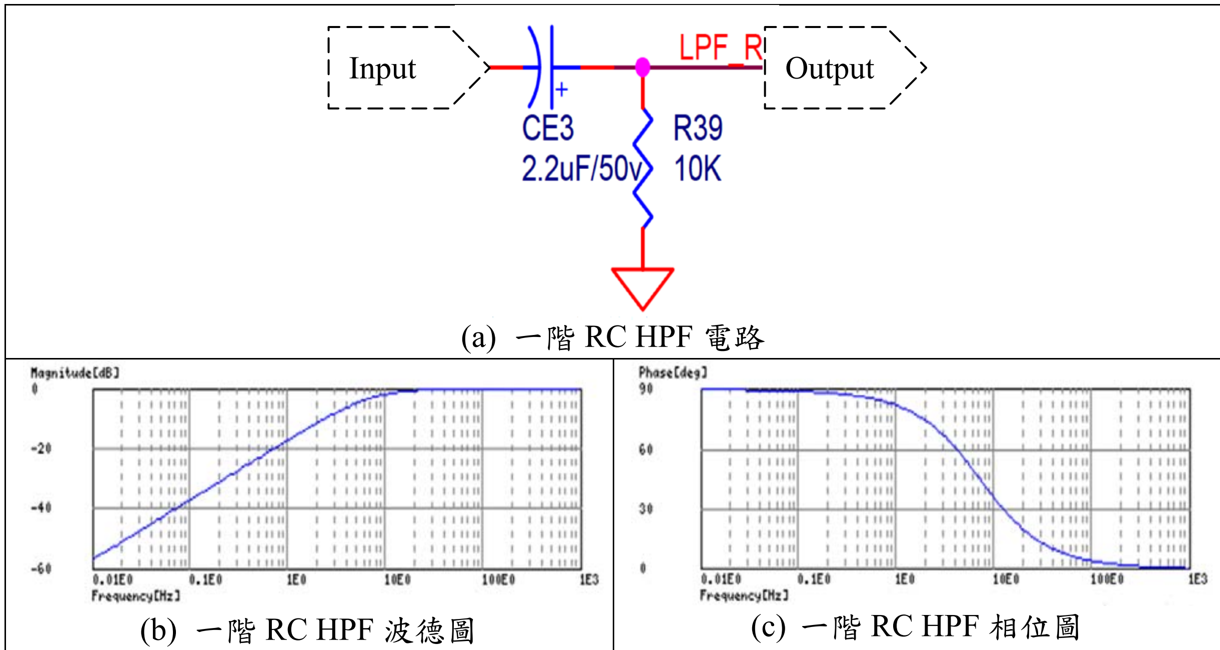
(a) 二階 RC LPF 電路

(b) 二階 RC LPF 波德圖　(c) 二階 RC LPF 相位圖

圖 2-23 二階低通濾波器分析

圖 2-24 (a)為接在低通濾波器後用來把低頻的直流部份濾除的一階高通濾波器(HPF)電路，當使用 R 為 10KΩ與 C 為 2.2uF 時，其轉移函數為(10)，-3dB 截止頻率為 7.23Hz，(b)為一階 RC HPF 的波德圖，(c)為其相位圖。

$$G_{HPF}(s) = \frac{s}{s+45.45} \tag{10}$$



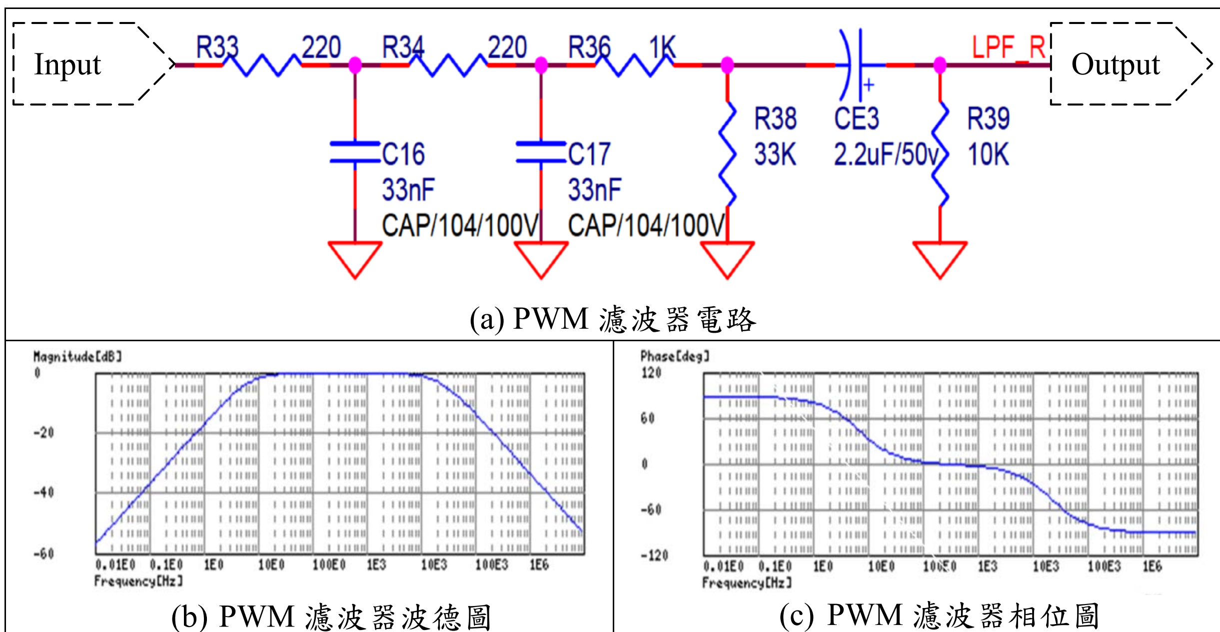
(a) 一階 RC HPF 電路

(b) 一階 RC HPF 波德圖

(c) 一階 RC HPF 相位圖

圖 2-24 一階高通濾波器分析

整個 PWM 波形的濾波電路為圖 2-25(a)所示，其結合了低通與高通濾波器，並且其通過頻帶為 7.23Hz 到 21.92KHz，相當於一個帶通濾波器，(b)為整個濾波器電路的波德圖，(c)為其相位圖。



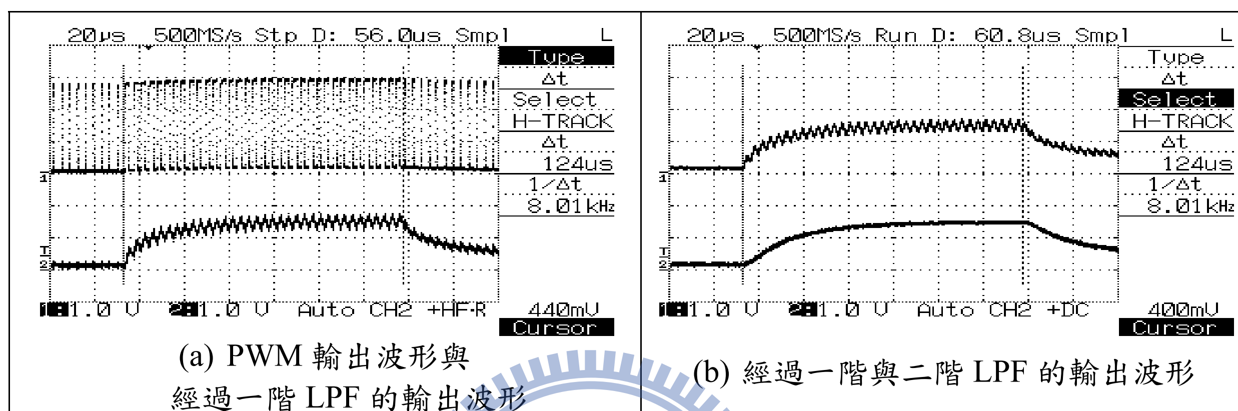
(a) PWM 濾波器電路

(b) PWM 濾波器波德圖

(c) PWM 濾波器相位圖

圖 2-25 PWM 濾波器的分析

我們使用 PWM 訊號來量測濾波器電路的功能，圖 2-26 為以 8KHz 播放頻率輸出 PWM 訊號，經過濾波器電路的波形，(a)為 PWM 輸出訊號與經過一階 LPF 的波形，可以在一階 LPF 的輸出波形上面看到 PWM 的高頻成份，(b)為經過一階與二階 LPF 的波形，在二階 LPF 的輸出波形上面高頻成份已經明顯被消除。

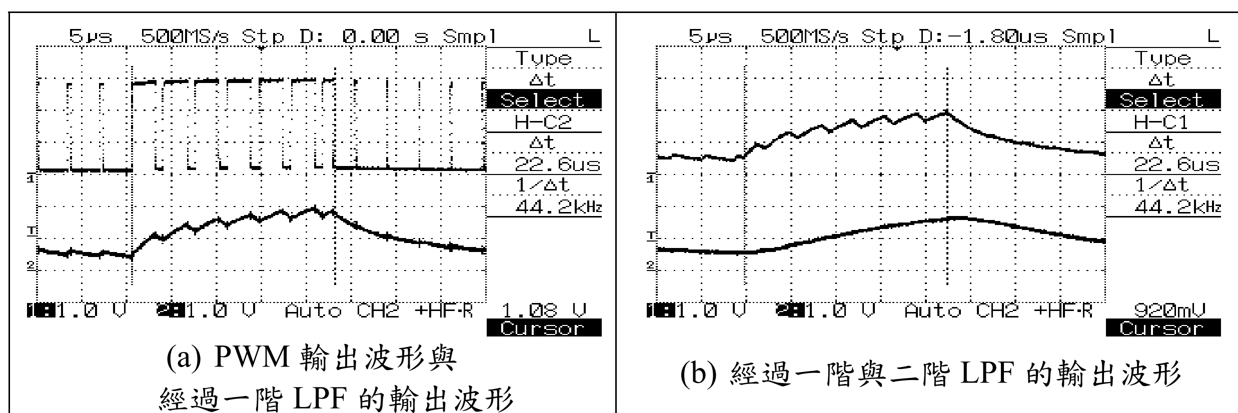


(a) PWM 輸出波形與經過一階 LPF 的輸出波形

(b) 經過一階與二階 LPF 的輸出波形

圖 2-26 輸出 8KHz PWM 訊號經過濾波器的波形

圖 2-27 為以 44.1KHz 播放頻率輸出 PWM 訊號，經過濾波器電路的波形，(a)為 PWM 輸出訊號與經過一階 LPF 的波形，(b)為經過一階與二階 LPF 的波形。



(a) PWM 輸出波形與經過一階 LPF 的輸出波形

(b) 經過一階與二階 LPF 的輸出波形

圖 2-27 輸出 44.1KHz PWM 訊號經過濾波器的波形

2.5 序列週邊介面(Serial Peripheral Interface)

STM32F 微控器內建了兩組序列週邊介面(簡稱 SPI)，允許微控器通過四線制串列匯流排介面與外部設備做資料溝通，STM32 微控器的 SPI 介面可以被設置成主模式(Master)或僕模式(Slave) [1]。在本文的多媒體應用中，我們將 SPI 模式操作在主模式下，用來跟 SD 記憶卡做資料溝通。

圖 2-28 為 SPI 模式的四條導線分別為串列時脈、主出僕入、主入僕出和僕裝置選擇(NSS)訊號連接多個僕裝置。在主模式下，串列時脈(SCK)由主裝置提供，命令由 MOSI 訊號線發出，資料由 MISO 訊號線接收，NSS 為僕裝置元件選擇線，每一個僕裝置需要獨立的 NSS 訊號。

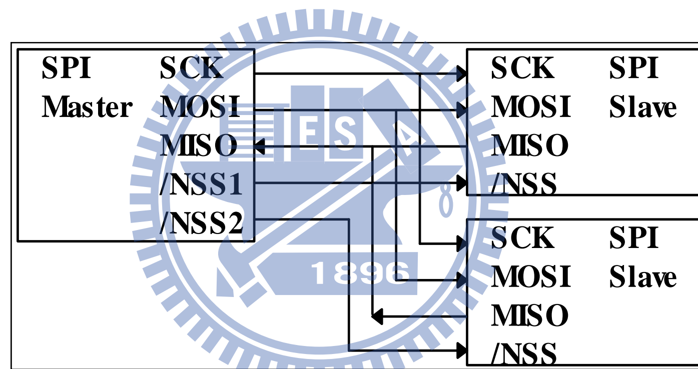


圖 2-28 單一 SPI Master 匯流排對複數 Slave [1]

詳細的序列週邊介面(SPI)資訊請參考意法半導體公司的使用手冊 RM0008 Referance Manual ch.23 [1]。

表 2-3 為本系統的 SPI 模組所對應的 I/O 接腳與功能說明。

SCK：串列時脈，由主裝置輸出頻率，僕裝置接收頻率。

MISO：主入/僕出接腳，在主模式下接收僕裝置發出的資料，在僕模式下發送資料給主裝置。

MOSI：主出/僕入接腳，在主模式下主裝置發送命令與資料，在僕模式下接收主裝置送出的資料。

NSS：僕裝置選擇，這是一個用來選擇僕裝置的接腳，它讓主裝置可以單獨地與特定僕裝置通訊，避免資料線發生衝突，NSS 僕裝置選擇接腳可以使用 GPIO 來取代，用以擴充僕裝置的數量。

表 2-3 SPI 模式接腳映射 [1]

PIN	Function	SPI1	SPI2
NSS	僕裝置選擇，用來選擇主/僕裝置	PA4	PB12
SCK	串列時鐘，作為主裝置的輸出	PA5	PB13
MISO	主裝置輸入/僕裝置輸出接腳	PA6	PB14
MOSI	主裝置輸出/僕裝置輸入接腳	PA7	PB15

圖 2-29 為 STM32F 微控器的 SPI 模組方塊圖，其透過 4 個 SPI 暫存器來控制，SPI_CR1 暫存器用來控制 SPI 模組的傳輸頻率、MSB/LSB 傳輸方式、16/8 位元傳輸方式等；SPI_CR2 暫存器用來設定 SPI 模組的中斷控制；SPI_SR 暫存器用來判斷傳輸狀態；SPI_DR 暫存器用來儲存傳送或接收的資料。

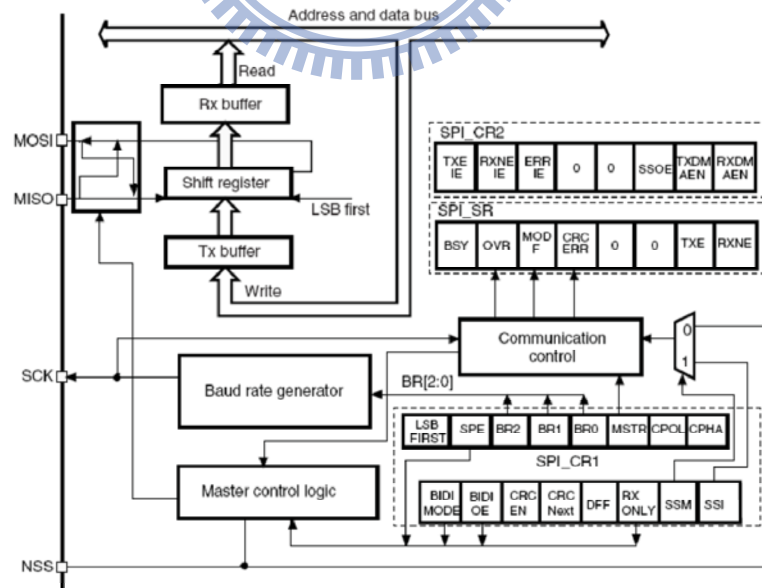


圖 2-29 SPI 模組方塊圖 [1]

SPI 模組功能的暫存器說明請參考意法半導體公司 RM0008 Referance Manual ch22 [1]使用手冊的 SPI 模組部分。

表 2-4 至 2-7 節錄了意法半導體公司 RM0008 Referance Manual ch22 [1] 使用手冊的 SPI 模組暫存器說明，包含狀態暫存器、控制暫存器 1、控制暫存器 2 與資料暫存器。

表 2-4 SPI 狀態暫存器 [1]

31:8	7	6	5	4	3	2	1	0
Res	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
Res	R	R	R	R	R	R	R	R

BSY：忙碌標誌 0：SPI 不忙 1：SPI 正忙於通信，或者發送緩衝非空	UDR：下溢標誌位元 0：未發生下溢 1：發生下溢。
OVR：溢出標誌 0：沒有出現溢出錯誤 1：出現溢出錯誤	CHSIDE：聲道 0：需要傳輸或者接收左聲道 1：需要傳輸或者接收右聲道
MODF：模式錯誤 0：沒有出現模式錯誤 1：出現模式錯誤	TXE：發送緩衝為空 0：發送緩衝非空 1：發送緩衝為空
CRCERR：CRC 錯誤標誌 0：收到的 CRC 值和 SPI_RXCRCR 暫存器中的值匹配 1：收到的 CRC 值和 SPI_RXCRCR 暫存器中的值不匹配	RXNE：接收緩衝非空 0：接收緩衝為空 1：接收緩衝非空

表 2-5 SPI 控制暫存器 2 [1]

31:8	7	6	5	4	3	2	1	0
Res	TXEIE	RXNEIE	ERRIE	Res		SSOE	TXDMAEN	RXDMAEN
Res	RW	RW	RW	Res	Res	RW	RW	RW

TXEIE：發送緩衝區空中斷使能 0：禁止TXE中斷 1：允許 TXE 中斷，當 TXE 標誌置位元時產生插斷要求	SSOE：SS輸出使能 0：禁止在主模式下SS輸出，該設備可以工作在多主設備模式 1：設備開啟時，開啟主模式下 SS 輸出，該設備不能工作在多主設備模式
RXNEIE：接收緩衝區非空中斷使能 0：禁止RXNE中斷 1：允許 RXNE 中斷，當 RXNE 標誌置位元時產生插斷要求	TXDMAEN：發送緩衝區DMA使能，當該位元被設置時，TXE標誌一旦被置位元就發出DMA請求 0：禁止發送緩衝區DMA 1：啟動發送緩衝區 DMA
ERRIR：錯誤中斷使能，當錯誤(CRCERR、OVR、MODF)產生時，該位控制是否產生中斷 0：禁止錯誤中斷 1：允許錯誤中斷	RXDMAEN：接收緩衝區DMA使能，當該位元被設置時，RXNE標誌一旦被置位元就發出DMA請求 0：禁止接收緩衝區DMA 1：啟動接收緩衝區 DMA

表 2-6 SPI 資料暫存器 [1]

31:16	15:0
	DR{15:0}
	RW

DR[15:0]：資料暫存器

待發送或者已經收到的資料，料暫存器對應兩個緩衝區：一個用於寫（發送緩衝）；另外一個用於讀（接收緩衝）。寫操作將資料寫到發送緩衝區；讀操作將返回接收緩衝區裡的資料。

表 2-7 SPI 控制暫存器 1 [1]

31:16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	BIDI MODE	BIDI OE	CRCEN	CEC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR[2:0]			MSTR	CPOL	CP HA
R	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIDIMODE: 雙向資料模式致能 0：選擇“雙線雙向”模式 1：選擇“單線雙向”模式	SSI: 內部從設備選擇 該位只在 SSM 位為’1’時有意義。它決定了 NSS 上的電位，在 NSS 引腳上的 I/O 操作無效
BIDIOE: 雙向模式下的輸出致能，和 BIDIMODE 一起決定在“單線雙向”模式下資料的輸出方向 0：輸出禁止(只收模式) 1：輸出致能(只發模式)	BR[2:0]：串列傳輸速率控制 000： fPCLK/2 001： fPCLK/4 010： fPCLK/8 011： fPCLK/16 100： fPCLK/32 101： fPCLK/64 110： fPCLK/128 111： fPCLK/256
CRCEN: 硬體 CRC 校驗致能 0：禁止 CRC 計算 1：啟動 CRC 計算	SPE: SPI 致能 0：禁止 SPI 設備 1：開啟 SPI 設備。
CRCNEXT: 下一個發送 CRC 0：下一個發送的值來自發送緩衝區。 1：下一個發送的值來自發送 CRC 暫存器。	LSBFIRST：框架格式 0：先發送 MSB 1：先發送 LSB
DFF: 數據框架格式 0：使用 8 位元資料框架格式進行發送/接收 1：使用 16 位元資料框架格式進行發送/接收	MSTR: 主設備選擇 0：配置為從設備 1：配置為主設備
RXONLY：只接收，和 BIDIMODE 位一起決定在“雙線雙向”模式下的傳輸方向 0：全雙工(發送和接收) 1：禁止輸出(只接收模式)	CPOL: 時鐘極性 0： 空閒狀態時，SCK 保持低電位 1： 空閒狀態時，SCK 保持高電位
SSM：軟體從設備管理，當 SSM 被置位時，NSS 引腳上的電位由 SSI 位的值決定 0：禁止軟體從設備管理 1：啟用軟體從設備管理	CPHA: 時鐘相位 0： 資料採樣從第一個時鐘邊沿開始 1： 資料採樣從第二個時鐘邊沿開始

2.5.1 SPI 模式的初始設定

在本文的多媒體應用上，STM32F 微控器作為系統的核心，將 SPI 模組設定為 Master 用來控制 SD 記憶卡的資料傳遞，所以我們將針對在主模式工作的設定進行說明，其設定步驟如下：

1. 設定 SPI_CR1 暫存器的串列傳輸速率控制位元 BR[2:0]，由(11)計算的初始傳輸速率。在 SD 記憶卡的規格書裡，規範在初始化記憶卡與進行卡判斷的流程中，SCK 串列時脈的頻率不可超過 400KHz，在執行完記憶卡判斷的流程後，才能調升頻率到 25MHz。

$$f_{BR} = f_{PCLK}/2^{(N_{BR}[2:0]+1)} \tag{11}$$

其中 f_{BR} 為SPI模組的傳輸速率。f_{PCLK} 為SPI模組的輸入頻率。N_{BR} 為SPI模組輸速率控制位元。

2. 設定 SPI_CR1 暫存器的 CPOL 和 CPHA 位元，圖 2-30 中為 STM32 微控器 SPI 模組的資料傳輸和串列時序的相位關係與極性，總共能夠組合成四種時序關係。CPOL(頻率極性)位元控制在沒有資料傳輸時時鐘的空閒狀態電位，如果 CPOL 清為 0，SCK 輸出接腳在空閒狀態保持低電位；如果 CPOL 設置為 1，則 SCK 輸出接腳在空閒狀態保持高電位。另外，如果 CPHA(頻率相位)位元被設置為 1，則資料位元的採樣在 SCK 時鐘的第二個邊沿進行；CPHA 位元被清為 0，則在 SCK 時鐘的第一個邊沿進行資料位元採樣。

3. 設置 SPI_CR1 暫存器的 DFF 位元來定義 8 或 16 位元資料格式，SD 記憶卡的 SPI 傳輸模式使用 8 位元格式，因此將 DFF 位元設置為 0。

4. 設置 SPI_CR1 暫存器的 LSBFIRST 位元定義資料格式，SD 記憶卡的 SPI 傳輸模式以 MSB 優先傳輸，因此將 LSBFIRST 位元設置為 0。

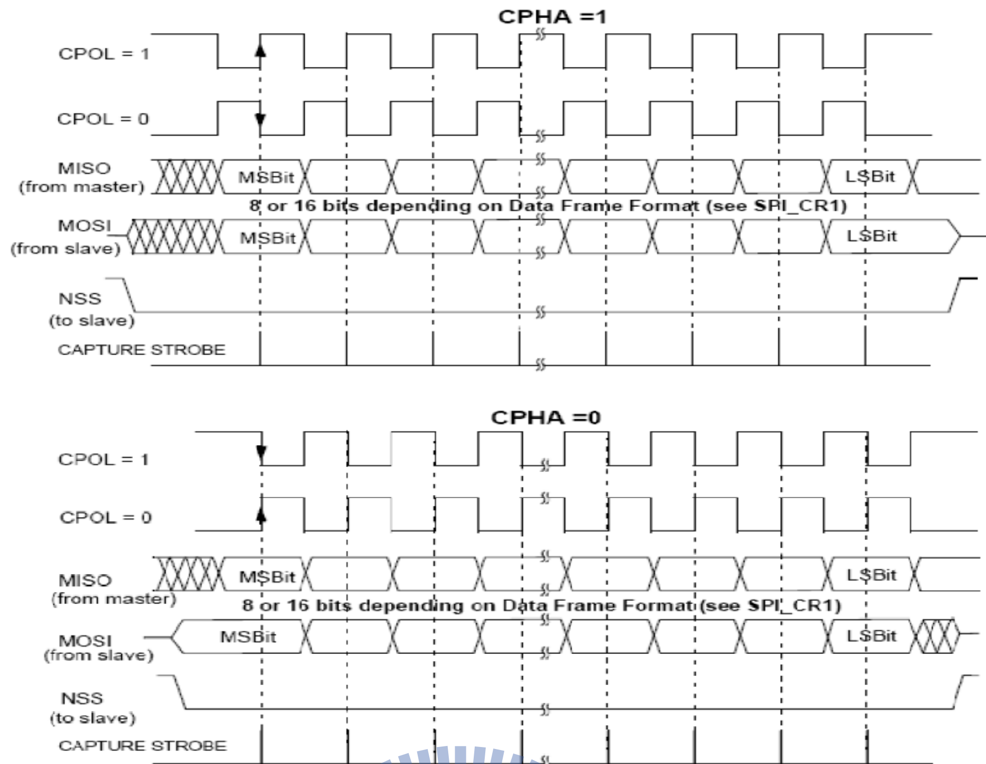


圖 2-30 SPI 模式資料傳輸的時序圖 [1]

5. 設置 SPI_CR1 暫存器 MSTR 位元，我們將 SPI 模組設置為 Master 主裝置，因此將 MSTR 位元設置為 1。

6. 設置 SPI_CR1 暫存器 SPE(Enable)位元，用以開啟 SPI 模組電路功能。

7. 設置 NSS 接腳的 GPIO 模式，將其對應的腳位設定為 Push-pull Output，用來做為選擇僕裝置的輸出接腳。

圖 2-31 為 SPI 模式的初始設定副程式 SPI_Configuration( )，首先必須將 SPI 模式下所對應的 I/O 接腳埠致能，本系統使用 SPI1 模組，因此對應到 GPIOA 埠的 PA[4:7]。接著便依照 SD 記憶卡的規格與主模式的設定設置 SPI_CR1 暫存器，最後將 SPI_CR1 的 SPE 位元設置為 1，用來啟動 SPI 模組電路功能。

程式名稱：SPI_Configuration ()

功能敘述：SPI 模組初始設定

輸　　入：無

輸　　出：無

```
void SPI_Configuration(void)
{
    RCC->APB2ENR |= (1<<2);              // 啟動 GPIOA
    GPIOA->CRL &= 0x000FF0FF;
    GPIOA->CRL |= 0xB8B00300;            // 設定 PA4, 5, 6, 7 模式

    RCC->APB2ENR |= (1<<12);             // 啟動 SPI1 頻率
    SPI1->CR1 = 0;                       // 清除 CR1
    SPI1->CR1 |= 0<<15;                  // 雙向資料模式
    SPI1->CR1 |= 0<<13;                  // 關閉 CRC 功能
    SPI1->CR1 |= 0<<11;                  // 資料格式為 8bit
    SPI1->CR1 |= 0<<10;                  // 全雙工模式
    SPI1->CR1 |= 1<<9;                   // 軟體控制僕裝置選擇
    SPI1->CR1 |= 1<<8;                   // 內部從設備選擇
    SPI1->CR1 |= 0<<7;                   // 高位元先傳輸
    SPI1->CR1 |= 6<<3;                   // 傳輸率 fpclk / 128
    SPI1->CR1 |= 1<<2;                   // 設定為主裝置
    SPI1->CR1 |= 0<<1;                   // 空閒時頻率輸出為 1
    SPI1->CR1 |= 0<<0;                   // First clock at first data
    SPI1->CR1 |= 1<<6;                   // 啟動 SPI 模組
}
```

圖 2-31 SPI 模式設定副程式

2.5.2 SPI 模式的資料傳輸

SPI 模式的資料傳輸是以 1 個位元組為最小單位，我們以圖 2-32 發送 1 位元組資料的 SPI_WriteByte( ) 副程式來說明動作流程，在資料的發送流程裡，當資料寫進資料暫存器後，SPI 模組就會開始執行發送流程。在發送第一個位元資料前，資料暫存器的資料會先傳入移位暫存器內，而後經由 SPI_CR1 暫存器中的 LSBFIRST 位元來決定是 MSB 或 LSB 先發送到 MOSI 腳位上。當資料暫存器的資料傳送到移位暫存器後，狀態暫存器的 TXE 位元會被設置為 1，表示緩衝器是空的。因此在將資料寫入資料暫存器之前，需要先確認 TXE 位元標誌是否為 1，這樣才不會造成資料衝突。

程式名稱：**SPI_WriteByte ()**	輸　　入：**unsigned char data**
功能敘述：使用 SPI 模組發送 **1** 位元組資料	輸　　出：**SPI_DR** 資料

```
u8 SPI_WriteByte (u8 data)
{
    while ((SPI1->SR & 1<<1) == 0);         //輸出暫存器不為空 SPI TXE = 0,
    SPI1->DR = data;                        // load date to SPI_DR
    while ((SPI1->SR & 1<<0) == 0);         // 輸入暫存器為空 SPI RXNE = 0,
    return SPI1->DR;                        // receive SPI_DR data
}
```

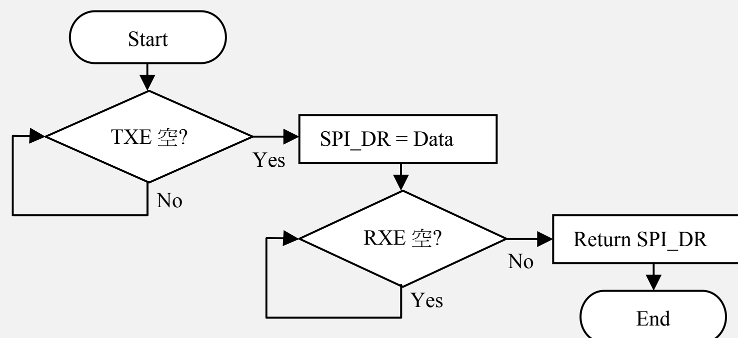


圖 2-32 SPI 模式的資料發送副程式

SPI 模式的資料接收也是以 1 個位元組為基本單位，SPI 模組在資料接收完成後，RX 移位暫存器中接收到的資料字組會被傳送到資料暫存器內，並且在最後一個時鐘邊沿，設置狀態暫存器的 RXNE 標誌位元為 1，表示資料接收完畢。當我們讀取資料暫存器時，SPI 模組會返回接收到的位元組資料，並清除 RXNE 位元。

圖 2-33 為 SPI 模組接收 1 位元組資料的 SPI_ReadByte( )副程式，將代表空資料的 0xFF 送出後，等待狀態暫存器的 RXNE 標誌被設定，如果 RXNE 為 1 時，則返回資料暫存器內的資料。

程式名稱：SPI_ReadByte () 　　**輸　入：無**

功能敘述：使用 SPI 模組讀取 1 位元組資料 　　**輸　出：SPI_DR 資料**

```
u8 SPI_ReadByte (void)
{
    while ((SPI1->SR & 1<<1) == 0);      //輸出暫存器不為空 SPI TXE = 0,
    SPI1->DR = DUMMY;                    // DUMMY = 0xFF
    while ((SPI1->SR & 1<<0) == 0);      // 輸入暫存器為空 SPI RXNE = 0,
    return SPI1->DR;
}
```

Start
TXE 空?
Yes
No
SPI_DR = 0xFF
RXE 空?
No
Yes
Return SPI_DR
End

圖 2-33 SPI 模式的資料接收副程式

2.5.3 SPI 模式下的 DMA 傳輸

STM32F 微控器的 SPI 模組可以使用 DMA 模式來加速傳輸速率，內建 DMA1 與 DMA2 兩組控制器，DMA1 控制器有 7 組通道，DMA2 控制器有 5 組通道，每一個通道對應一個或多個裝置模組。分配給 SPI1 RX 模組使用的是 DMA1 的 Channel 2，給 SPI1 RX 使用的是 DMA1 的 Channel 3。每一個通道都可以在設備暫存器和記憶體位址之間執行 DMA 傳輸 [1]。

詳細的 DMA 說明資訊請參考意法半導體公司的使用手冊 RM0008 Referance Manual ch.9 [1]。

圖 2-34 為使用 DMA 通道來傳遞 SPI1 模組資料的 stm_dma_transfer( ) 設定副程式 [18]，STM32 微控器將 SPI1 的 RX 規劃在 DMA1 的通道 2，TX 規劃在通道 3。

DMA1 的通道 2 用來進行 SPI1 的 RX 接收動作，在 DMA1_CPAR2 暫存器中填入 SPI 設備的 SPI1_DR 暫存器的位址，此為 DMA 傳輸時的資料來源位址；在 DMA1_CMAR2 暫存器中設置記憶體緩衝區的位址，傳輸的資料將寫入這個記憶體位址；需要在 DMA_CCR2 暫存器中設置資料傳輸的方向的 DIR 位元為 0，表示資料來源從記憶體讀取，還需設定外設和記憶體的資料寬度；在 DMA_CNDTR2 暫存器中可以設置所需要的資料量，DMA 控制器在接收每一筆資料後，CNDTR 數值會遞減，當遞減為零時，DMA 接收動作會停止。

DMA1 的通道 3 用來進行 SPI1 的 TX 傳輸動作，在 DMA1_CPAR3 暫存器中設置 SPI 設備的 SPI1_DR 暫存器的位址，此為 DMA 傳輸時的資料目的地；在 DMA_CMAR3 暫存器中設置記憶體位址，傳輸的資料來源將由這個位址提供；在 DMA_CCR3 暫存器中設置資料傳輸的方向、設備和記憶

體的增量模式、外設和記憶體的資料寬度；在 DMA_CNDTR3 暫存器中設置要傳輸資料量，在每筆資料傳輸後，這個數值會遞減。

程式名稱：stm32_dma_transfer ( )	**輸　入：u8 *buff　緩衝區位置**
功能敘述：使用 DMA 傳輸模式來傳送/接收	**u32 btr　　要讀取的位元組數量**
SPI 資 料(rx only)	**輸　出：u8 * buff　資料寫入緩衝區位置**

```
void stm32_dma_transfer(const u8 *buff, u32 btr)
{
    u16 rw_workbyte[] = { 0xffff };
    RCC->AHBENR         |= (1<<0);                              // 啟動 DMA1 頻率
    DMA1->IFCR |= DMA1_Channel2_IT_Mask;                        // 設定 DMA1 ch2 = SPI RX
    DMA1->IFCR |= DMA1_Channel3_IT_Mask;                        // 設定 DMA1 ch3 = SPI TX
// DMA1 channel2 configuration SPI1 RX
    DMA1->CPAR2     = (u32)(&(SPI1->DR));                       // 裝置來源 SPI1_DR
    DMA1->CMAR2     = (u32)buff;                                // 記憶體來源 buff
    DMA1->CCR2      = 0x3080;
    DMA1->CNDTR2    = btr;
// DMA1 channel3 configuration SPI1 TX
    DMA1->CPAR3     = (u32)(&(SPI1->DR));                       // 裝置來源 SPI1_DR
    DMA1->CMAR3     = (u32)rw_workbyte;                         // 記憶體來源 workbyte
    DMA1->CCR3      = 0x3010;
    DMA1->CNDTR3    = btr;
    DMA1->CCR2 |= CCR_ENABLE_Set;                               // 啟動 DMA1 ch2 RX
    DMA1->CCR3 |= CCR_ENABLE_Set;                               // 啟動 DMA1 ch3 TX
    SPI1->CR2 |= (SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx);       // 啟動 SPI RX TX 請求
//  等待 DMA1_Channel 3 Transfer Complete, DMA1_Channel 2 Receive Complete
    while (DMA_GetFlagStatus(DMA_FLAG_SPI_SD_TC_RX) == 0)      { ; }
    DMA1->CCR2 &= CCR_ENABLE_Reset;                             // 關閉 DMA1 ch2 RX
    DMA1->CCR3 &= CCR_ENABLE_Reset;                             // 關閉 DMS1 ch3 TX
    SPI1->CR2 &= (u16)~(SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx); // 關閉 SPI RX TX 請求
}
```

圖 2-34 SPI 傳輸的 DMA 設定副程式

圖 2-35 為使用迴圈或 DMA 通道的 SPI 資料接收程式，如使用迴圈來接收資料會花費較多的時間在軟體處理上面，並造成資料接收時間過長影響系統效率。而使用 DMA 來傳輸資料，則會縮短資料接收時間。

程式名稱：SD_ReceiveData ( )

功能敘述：使用 DMA 傳輸模式來接收設定數量的 SPI 資料

輸　入：u8 *data　資料緩衝區位置

u16 len　要讀取的資料數量

u8 release　CS 是否放開

輸　出：u8 * buff　資料寫入緩衝區位置

```
u8 SD_ReceiveData(u8 *data, u16 len, u8 release)
{
    u16 retry = 0;
    u8 r1;
    SD_Enable();
    do {
        r1 = SPI_RWByte(DUMMY);
        if (retry++ > 5000)      return r1;
    } while(r1 != 0xFE);
    while (len--)   {
        r1 = SPI_RWByte(DUMMY);
        *data = r1;
        data++;
    }
    SPI_RWByte(DUMMY);
    SPI_RWByte(DUMMY);
    if (release == 1)
    {
        SD_Disable();
        SPI_RWByte(DUMMY);
    }
    return 0;
}
```

LOOP Mode

(a) 迴圈模式

```
u8 SD_ReceiveData(u8 *data, u16 len, u8 release)
{
    u16 retry = 0;
    u8 r1;
    SD_Enable();
    do {
        r1 = SPI_RWByte(DUMMY);
        if (retry++ > 5000)      return r1;
    } while(r1 != 0xFE);

    stm32_dma_transfer (TRUE, data, (UINT)len);

    SPI_RWByte(DUMMY);
    SPI_RWByte(DUMMY);
    if (release == 1)
    {
        SD_Disable();
        SPI_RWByte(DUMMY);
    }
    return 0;
}
```

DMA Mode

(b) DMA 模式

圖 2-35 SPI 使用迴圈與 DMA 的傳輸程式

2.6 SDIO 模組

STM32F 微控器內置了與多媒體介面卡規格相容的 SDIO 模組，並且支援三種不同的資料匯流排模式：1 位元(默認)、4 位元和 8 位元，SDIO 模組在重置(Reset)後預設的資料匯流排為 1 位元模式，匯流排寬度可在與 SD 記憶卡溝通過程中自由更改 [1]，詳細的 SDIO 模組說明請參考意法半導體公司的使用手冊 RM0008 Referance Manual ch.19 [1]。

STM32 微控器的 SDIO 模組共有 10 根訊號線，表 2-8 分別說明其功能與對應的 GPIO 埠。

SDIO_CMD 為雙向傳輸的訊號線，用於傳輸命令與接收回應。

SDIO_SCK 用於提供 SD 記憶卡的時鐘頻率，SDIO 模組的 SCK 時鐘頻率可以在 0MHz 至 25MHz 之間變化。

SDIO_D[7:0]用於資料傳輸使用，SDIO 模組重置後，設定使用 SDIO_D[0]於資料傳輸。在記憶卡初始化完成後，系統可以根據記憶卡的規格來改變資料匯流排的寬度，如 4 位元為 SDIO_D[3:0]或 8 位元為 SDIO_D[7:0]。

表 2-8 SDIO 模組接腳對應 [1]

Pin	Direction	Description	GPIO
SDIO_CK	Output	MultiMediaCard/SD/SDIO card clock. This pin is the clock from host to card	**PC12**
SDIO_CMD	Bidirectional	MultiMediaCard/SD/SDIO card command. This pin is the bidirectional command/response signal.	**PD2**
SDIO_D0	Bidirectional	MultiMediaCard/SD/SDIO card data. These pins are the bidirectional databus	**PC8**
SDIO_D1			**PC9**
SDIO_D2			**PC10**
SDIO_D3			**PC11**
SDIO_D4			**PB8**
SDIO_D5			**PB9**
SDIO_D6			**PC6**
SDIO_D7			**PC7**

2.6.1 SDIO 功能說明

STM32F 微控器的 SDIO 功能包含兩個部份，圖 2-36 為 SDIO 模組方塊圖，內含 SDIO 轉接器與 AHB 匯流排界兩個模組，其中 SDIO Adapter 轉接器模組用來實現有關於 MMC/SD/SD-I/O 介面卡的所有功能，如 SCK 頻率的產生、命令和資料的傳送等。AHB 匯流排介面用來操作 SDIO Adapter 轉接器中的暫存器，並且可以產生中斷與 DMA 請求。

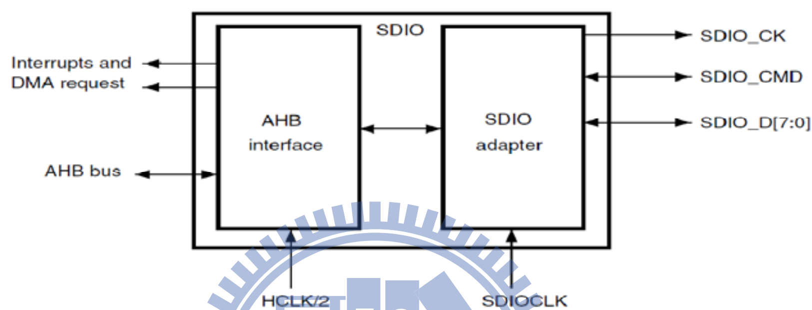


圖 2-36 SDIO 模組方塊圖 [1]

圖 2-37 為 SDIO Adapter 轉接器模組的方塊圖，其包含了轉接器暫存器(Adapter registers)、控制單元(Control unit)、命令通道(Command path)、資料通道(Data path)與數據 FIFO 等 5 個部分，我們將針對命令通道與資料通道進行說明。

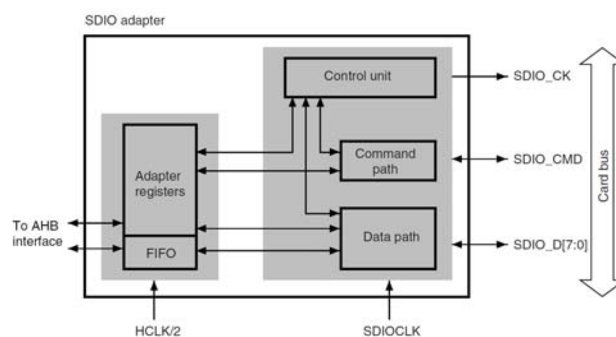


圖 2-37 SDIO 轉接器模組方塊圖 [1]

命令通道(Command Path)

命令通道是執行向多媒體卡發送命令並接收從卡發出回應的單元，命令的運作是通過命令通道狀態機(CPSM)來完成的。圖 2-38 為 SDIO 命令通道狀態機的方塊圖，系統重置後 CPSM 處於空閒狀態(Idle)，當命令參數寫入命令暫存器(SDIO_CMD)並設置了 CPSMEN 位元，就會開始發送命令；命令發送完成時，命令通道狀態機(CPSM)設置狀態暫存器(SDIO_STA)的狀態標誌，並在不需要回應時進入空閒狀態；當收到回應後，接收到的 CRC 碼將會與內部產生的 CRC 碼比較，然後設置相應的狀態標誌；當進入等候狀態時，命令計數器開始運行；當 CPSM 進入接收狀態之前如果產生了超時，則會設置超時標誌並進入空閒狀態。

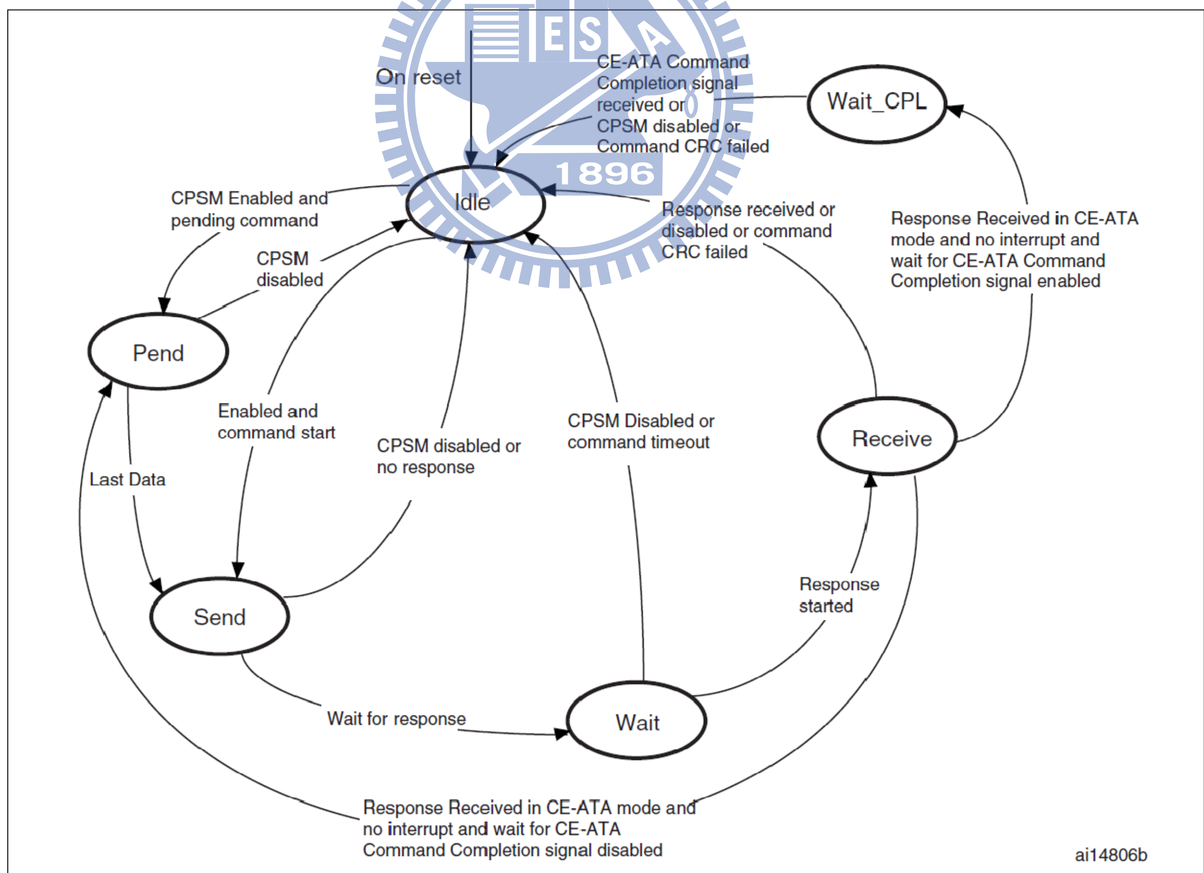


圖 2-38 SDIO 命令通道狀態機方塊圖 [1]

圖 2-39 為用來發送命令的 SDIO_Send_Command( )副程式，將命令參數寫入 SDIO_ARG 暫存器，和將命令索引寫入臨時暫存器中，依照命令索引的回應狀態，在臨時暫存器中設置 WAITRESP 回應位元，和設置 CPSM 啟動位元，最後將設置好的臨時暫存器值寫入 SDIO_CMD 暫存器，就完成命令發送流程。

程式名稱：**SDIO_SendCommand ( )**	輸　入：***SDIO_CmdInitStruct**
功能敘述：發送 **SDIO** 命令	輸　出：無

```
void SDIO_SendCommand (SDIO_CmdInitTypeDef *SDIO_CmdInitStruct)
{
    u32 tmpreg = 0;
    SDIO->ARG = SDIO_CmdInitStruct->SDIO_Argument;        // 設定 Argument 參數
    tmpreg = SDIO->CMD;                                    // 讀取 SDIO_CMD 值到 tmpreg
    tmpreg &= CMD_CLEAR_MASK;                              // 清除 SDIO_CMD 設定位元
    tmpreg |= (u32)SDIO_CmdInitStruct->SDIO_CmdIndex       // 設定 CMD 命令參數
              | SDIO_CmdInitStruct->SDIO_Response         // 設定 WAITRESP 位元
              | SDIO_CmdInitStruct->SDIO_Wait             // 設定 WAITINT and WAITPEND 位元
              | SDIO_CmdInitStruct->SDIO_CPSM;            // 設定 CPSMEN 位元
    SDIO->CMD = tmpreg;                                    // 寫入到 SDIO_CMD 暫存器
}
```

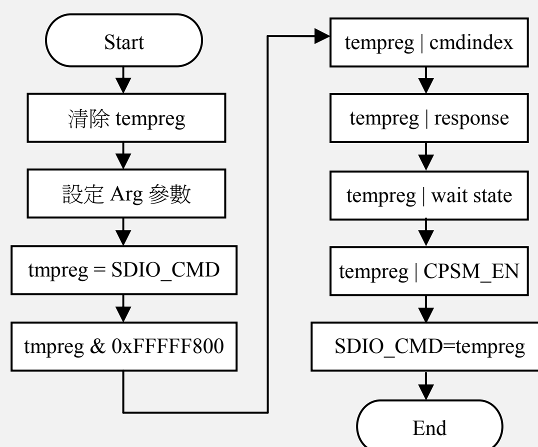


圖 2-39 SDIO 命令發送副程式

資料通道(Data Path)

資料通道控制主裝置與多媒體卡之間的傳輸資料，STM32 微控器的 SDIO 模組，是以 1 位元資料匯流排為初始設定，一個時鐘週期只在 SDIO_D0 上傳輸 1 位元資料，在 SDIO_CLKCR 暫存器中，可以設置 WIDBUS[1:0] 位元來更改資料匯流排的寬度。圖 2-40 為 SDIO 模組的 DPSM 資料通道狀態機的方塊圖，系統重置後 DPSM 處於空閒狀態(Idle)，將資料傳輸模式設定寫入 SDIO_DCTRL 暫存器，並且設置 SDIO_DCTRL 暫存器的致能 DTEN 位元後，資料通道狀態機將根據傳輸的方向(發送或接收)進入 Wait_S 或 Wait_R 狀態。

發送時 DPSM 進入 Wait_S 狀態，如果發送 FIFO 中有資料，則 DPSM 進入發送狀態(Send)，同時資料通道開始向多媒體卡發送資料。

接收時 DPSM 進入 Wait_R 狀態並等待開始位元，當收到開始位元後，DPSM 進入接收狀態(Reveive)，同時資料通道開始從多媒體卡接收資料。

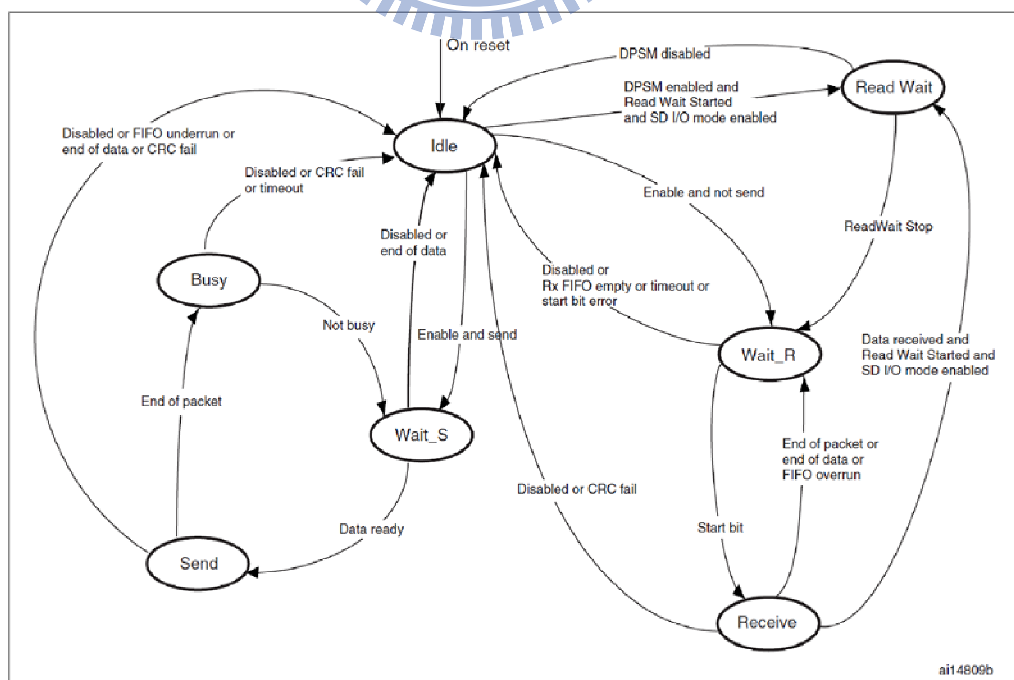


圖 2-40 SDIO 資料通道狀態機方塊圖 [1]

圖 2-41 為資料通道狀態機(DPSM)的設定副程式，先將 Time Out 設定值寫入 SDIO_DTIMER 暫存器，資料長度寫入 SDIO_DLEN 暫存器，再將區塊大小、傳輸方向、傳輸模式寫入 SDIO_DCTRL 暫存器，最後設定 DPSM 啟動位元，即完成資料通道狀態機設定。

程式名稱：SDIO_DataConfig ( )	**輸　入：* SDIO_DataInitStruct**
功能敘述：設定資料通道狀態機	**輸　出：無**

```
void SDIO_DataConfig(SDIO_DataInitTypeDef* SDIO_DataInitStruct)
{
   u32 tmpreg = 0;
   SDIO->DTIMER = SDIO_DataInitStruct->SDIO_DataTimeOut;    // 設定 TimeOut value
   SDIO->DLEN = SDIO_DataInitStruct->SDIO_DataLength;       // 設定 DataLength value
   tmpreg = SDIO->DCTRL;                                    // 讀取 SDIO_DCTRL 值到 tmpreg
   tmpreg &= DCTRL_CLEAR_MASK;                              // 清除 SDIO_DCTRL 設定位元
   tmpreg |= (u32)SDIO_DataInitStruct->SDIO_DataBlockSize   // 設定 DBCKSIZE 位元
             | SDIO_DataInitStruct->SDIO_TransferDir        // 設定 DTDIR 位元
             | SDIO_DataInitStruct->SDIO_TransferMode       // 設定 DTMODE 位元 t
             | SDIO_DataInitStruct->SDIO_DPSM;              // 設定 DEN 位元
   SDIO->DCTRL = tmpreg;                                    // 寫入到 SDIO_DCTRL 暫存器
}
```

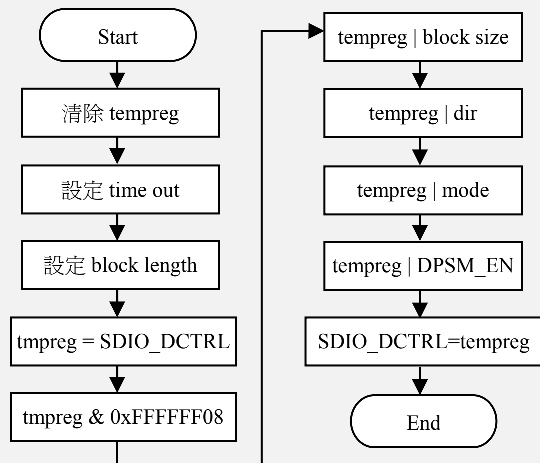


圖 2-41 DPSM 設定副程式

STM32 微控器的 SDIO 模組設定

在使用 SDIO 介面之前，需要對 SDIO 模組進行初始設定與啟動，圖 2-42 為啟動 SDIO 模組的的 SDIO_Init( )設定流程，首先清除時鐘控制暫存器，接著設定時鐘頻率(f = SDICLK/[CLKDIV+2])，再設定是否進入省電模式，最後將設定好的參數填入 SDIO_CLKCR 暫存器內就完成設定。

程式名稱：SDIO_Init ( )	輸　入：SDIO_InitTypeDef* SDIO_InitStruct
功能敘述：STM32 微控器的 SDIO 模組設定	輸　出：無

```
void SDIO_Init(SDIO_InitTypeDef* SDIO_InitStruct)
{
   u32 tmpreg = 0;
   /* Clear CLKDIV, PWRSAV, BYPASS, WIDBUS, NEGEDGE, HWFC_EN bits */
   tmpreg &= CLKCR_CLEAR_MASK;                        // 清除設定資料
   /* Set CLKDIV bits according to SDIO_ClockDiv value *
   /* Set PWRSAV bit according to SDIO_ClockPowerSave value */
   /* Set BYPASS bit according to SDIO_ClockBypass value */
   /* Set WIDBUS bits according to SDIO_BusWide value */
   /* Set NEGEDGE bits according to SDIO_ClockEdge value */
   /* Set HWFC_EN bits according to SDIO_HardwareFlowControl value */

   tmpreg   |= (SDIO_InitStruct->SDIO_ClockDiv
             | SDIO_InitStruct->SDIO_ClockPowerSave      // 設定省電模式
             | SDIO_InitStruct->SDIO_ClockBypass         // 設定頻率來源
             | SDIO_InitStruct->SDIO_BusWide             // 設定資料匯流排
             | SDIO_InitStruct->SDIO_ClockEdge           // 設定頻率與資料對齊方式
             | DIO_InitStruct->SDIO_HardwareFlowControl); // 設定硬體流程控制

   /* Write to SDIO CLKCR */
   SDIO->CLKCR = tmpreg;                              // 寫入控制暫存器
}
```

圖 2-42 SDIO 模式設定副程式

SDIO 暫存器介紹

表 2-9 至 2-13 節錄了意法半導體公司 RM0008 Referance Manual [1] STM32 微控器使用手冊的 SDIO 模組暫存器說明資料，包含 SDIO 模組主要的工作暫存器有，SDIO_CLKCR 暫存器控制 SDIO 模組的工作頻率與資料線模式；SDIO_CMD 暫存器包含命令索引與類型，用來發送命令與控制 CPSM；SDIO_ARG 暫存器做為命令的一部分，用來發送 32 位元的命令參數；SDIO_DCTRL 暫存器用來控制資料狀態機 DPSM；SDIO_STA 暫存器用來顯示 SDIO 模組的工作狀態

表 2-9 SDIO 時鐘控制暫存器 [1]

31:15	14	13	12	11	10	9	7:0
Res	HWFC_EN	NEGEDGE	WIDBUS	BYPAS	PWRSAV	CLKEN	CLKDIV
Res	R/W	R/W	R/W	R/W	R/W	R/W	R/W

HWFC_EN：硬體流控制致能 0：關閉硬體流控制 1：致能硬體流控制	NEGEDGE：SDIO_CK 相位選擇位元 0：在主時鐘 SDIOCLK 的上升沿產生 SDIO_CK 1：在主時鐘 SDIOCLK 的下降沿產生 SDIO_CK
WIDBUS：寬匯流排模式致能位元 00：預設匯流排模式，使用 SDIO_D0 01：4 位元匯流排模式，使用 SDIO_D[3:0] 10：8 位元匯流排模式，使用 SDIO_D[7:0]	BYPASS：旁路時鐘分頻器 0：關閉旁路：驅動 SDIO_CK 輸出信號之前，依據 CLKDIV 數值對 SDIOCLK 分頻 1：致能旁路：SDIOCLK 直接驅動 SDIO_CK 輸出信號
PWRSAV：省電配置位元，為了省電，當匯流排為空閒時，設置 PWRSAV 位元可以關閉 SDIO_CK 時鐘輸出 0：始終輸出 SDIO_CK 1：僅在有匯流排活動時才輸出 SDIO_CK	CLKEN：時鐘致能位元 0：SDIO_CK 關閉 1：SDIO_CK 致能
CLKDIV：時鐘分頻係數 這個域定義了輸入時鐘(SDIOCLK)與輸出時鐘(SDIO_CK)間的分頻係數： SDIO_CK 頻率 ＝SDIOCLK/[CLKDIV＋2]	

表 2-10 SDIO 資料控制暫存器 [1]

31:15	14	13	12	11	10	9	8	7:6	5:0
Res	SDIOEN	PWMOD	RWSTOP	RWSTART	DBLOCKSIZE	DMAEN	DTMODE	DTDIR	DTEN
R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SDIOEN：SD I/O 致能功能，如果設置了該位元，則 DPSM 執行 SD I/O 卡特定的操作	DMAEN：DMA 致能位元 0：關閉 DMA 1：致能 DMA
RWMOD：讀等待模式 0：停止 SDIO_CK 控制讀等待 1：使用 SDIO_D2 控制讀等待	DTMODE：資料傳輸模式 0：塊資料傳輸 1：流資料傳輸
RWSTOP：讀等待停止 0：如果設置了 RWSTART，執行讀等待 1：如果設置了 RWSTART，停止讀等待	DTDIR：資料傳輸方向 0：控制器至記憶卡 1：記憶卡至控制器
RWSTART：讀等待開始，設置該位元開始讀等待操作。	DTEN：資料傳輸致能位元，設置為 1，則開始資料傳輸
DBLOCKSIZE：資料塊長度，當選擇了塊資料傳輸模式，該域定義資料塊長度如下	
0000：(十進位 0)塊長度 ＝20＝1 位元組	1000：(十進位 8)塊長度 ＝28＝256 位元組
0001：(十進位 1)塊長度 ＝21＝2 位元組	1001：(十進位 9)塊長度 ＝29＝512 位元組
0010：(十進位 2)塊長度 ＝22＝4 位元組	1010：(十進位 10)塊長度 ＝210＝1024 位元組
0011：(十進位 3)塊長度 ＝23＝8 位元組	1011：(十進位 11)塊長度 ＝211＝2048 位元組
0100：(十進位 4)塊長度 ＝24＝16 位元組	1100：(十進位 12)塊長度 ＝212＝4096 位元組
0101：(十進位 5)塊長度 ＝25＝32 位元組	1101：(十進位 13)塊長度 ＝213＝8192 位元組
0110：(十進位 6)塊長度 ＝26＝64 位元組	1110：(十進位 14)塊長度 ＝214＝16384 位元組
0111：(十進位 7)塊長度 ＝27＝128 位元組	1111：(十進位 15)保留

表 2-11 SDIO 命令暫存器 [1]

31:15	14	13	12	11	10	9	8	7:6	5:0
Res	ATACMD	nIEN	ENCMD	SDIOSuspend	CPSMEN	WAITPEND	WAITINT	WAITRESP	CMDINDEX
R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

ATACMD：CE-ATA 命令，如果設置該位，CPSM 轉至 CMD61	WAITPEND：CPSM 等待資料傳輸結束(CmdPend 內部信號)，如果設置該位元，則 CPSM 在開始發送一個命令之前等待資料傳輸結束
nIEN：不致能中斷，如果未設置該位元，則致能 CE-ATA 設備的中斷	WAITINT：CPSM 等待插斷要求，如果設置該位元，則 CPSM 關閉命令逾時控制並等待插斷要求
ENCMDcompl：致能 CMD 完成，如果設置該位元，則致能命令完成信號	WAITRESP：等待響應位元，這 2 位元指示 CPSM 是否需要等待回應，如果需要等待回應，則指示回應類型 00：無回應，等待 CMDSENT 標誌 01：短回應，等待 CMDREND 或 CCRCFAIL 標誌 10：無回應，等待 CMDSENT 標誌 11：長回應，等待 CMDREND 或 CCRCFAIL 標誌
SDIOSuspend：SD I/O 暫停命令，如果設置該位元，則將要發送的命令是一個暫停命令(只能用于 SDIO 卡)	
CPSMEN：命令通道狀態機(CPSM)致能位元，如果設置該位元，則致能 CPSM	CMDINDEX：命令索引是作為命令的一部分發送到卡中

1896

表 2-12 SDIO 參數暫存器 [1]

31:0	
CMDARG	
R/W	
CMDARG[31:0]	CMDARG：命令參數是發送到卡中的命令的一部分，如果一個命令包含一個參數，必須在寫命令到命令暫存器之前載入這個暫存器

表 2-13 SDIO 狀態暫存器 [1]

31:24	23	22	21	20	19	18	17	16	15	14	13
Res	CEATACEN	SDIOIT	RXDAVL	TXDAVL	RXFIFOE	TXFIFOE	RXFIFOF	TXFIFOF	RXFIFOHF	TXFIFOHE	RXACT
Res	R	R	R	R	R	R	R	R	R	R	R

12	11	10	9	8	7	6	5	4	3	2	1	0
TXACT	CMDACT	DBCKEND	STBITERR	DATAEND	CMDSENT	CMDREND	RXOVERR	TXUNDERR	DTMEOUT	CTIMEOUT	DCRCFAIL	CCRCFAIL
R	R	R	R	R	R	R	R	R	R	R	R	R

CEATAEND：在 CMD61 接收到 CE-ATA 命令完成信號	CMDACT：正在傳輸命令
SDIOIT：收到 SDIO 中斷。	DBCKEND：已發送/接收資料塊(CRC 檢測成功)
RXDVAL：在接收 FIFO 中的資料可用	STBITERR：在寬匯流排模式，沒有在所有資料信號上檢測到起始位元
TXDVAL：在發送 FIFO 中的資料可用	DATAEND：資料結束(資料計數器，SDIO_DCOUNT = 0)
RXFIFOE：接收 FIFO 空	CMDSENT：命令已發送(不需要回應)
TXFIFOE：發送 FIFO 空	CMDREND：已接收到回應(CRC 檢測成功)
RXFIFOF：接收 FIFO 滿	RXOVERR：接收 FIFO 上溢錯誤
TXFIFOF：發送 FIFO 滿	TXUNDERR：發送 FIFO 下溢錯誤
RXFIFOHF：接收 FIFO 半滿：FIFO 中至少還有 8 個字	DTIMEOUT：數據超時
TXFIFOHE：發送 FIFO 半空：FIFO 中至少還可以寫入 8 個字	CTIMEOUT：命令回應超時 ,命令逾時時間是一個固定的值，為 64 個 SDIO_CK 時鐘週期
RXACT：正在接收資料	DCRCFAIL：已發送/接收資料塊(CRC 檢測失敗)
TXACT：正在發送資料	CCRCFAIL：已收到命令響應(CRC 檢測失敗)

第三章　SD 記憶卡

SD 記憶卡全名為 Secure Digital Memory Card 是一款基於快閃記憶器(NAND Flash)的儲存裝置，由日本松下(Toshiba)、東芝(Panasonic)以及美國新帝公司(SanDisk)於 1999 年共同研究開發，其結合了新帝公司的快閃記憶卡控制技術與 MLC（Multilevel Cell）技術和日本東芝的 NAND 技術，SD 記憶卡的資料傳輸規範與硬體結構規格是從 Multi Media Card (MMC)所延續下來的。

本章 SD 記憶卡規格與說明，主要以 SD Group 所公佈的 Physical Layer Specification 規格書 [4] 與 SanDisk SD Card Product Manual 產品說明書 [5] 的內容為參考，部份範例程式從 STMicroelectronics 的 Design Support (http://www.st.com/internet/mcu/product)以及 Helix Community (https://helixcommunity.org/downloads)提供的 Helix Player 11 Gold 原始碼，針對本系統所使用的微控器與應用架構進行修改。

圖 3-1(a)為 SD Association 所公佈的 SD 記憶卡標誌，(b)為 SD 記憶卡的外觀尺寸圖，如需更多的 SD 記憶卡的類型標誌與尺寸規格可以參考 SD Association (http://www.stcard.org)。

(a) 記憶卡標誌

(b) 記憶卡尺寸

圖 3-1 SD 記憶卡標誌與尺寸

3.1 SD 記憶卡概述

SD 記憶卡最早在 2000 年公佈的版本為 V1.0，其卡容量限制在 2GB 以內。在 2006 年 3 月發表 V2.0 規格並且支援高容量，因此高容量 SD 記憶卡又稱為 SDHC。SDHC 與 SD 的主要差異在於，V1.0 版本使用 FAT16 檔案系統，最多只能管理 65536 個檔案，再考慮每個檔案最小儲存 32KB 的資料量，所以 SD 卡容量上限只能到達 2GB。SDHC 改用了 FAT32 格式來解決 V1.0 支援容量有限的問題；依規格定義，容量最大可達到 32GB。目前 SD Group 已經定義容量可支援到 2TB 的 SDXC 規格。

SD 記憶卡的傳輸速率，是按照 CD-ROM 光碟機的 150 KB/s 傳輸速率來定義為 1 倍速，一般的 SD 記憶卡能夠達到 6 倍的傳輸速率（900KB/s），而 SD 記憶卡最高能傳輸 166 倍（25MB/s）的速率，詳細的 SD 記憶卡功能說明，可參考 SA Association (http://www.sdcard.org)與維基百科(http://en.wikipedia.org/wiki/SDcard)的 Secure Digital Memory Card 部份。

SD 記憶卡的產品功能與特性為 [4]：

- SD 規格支援到 2GB 容量，SDHC 規格支援容量可超過 2GB 以上。
- 支援 SD Bus 與 SPI Bus 傳輸協定。
- 廣泛的工作電壓範圍，從 2.0V 至 3.6V。
- 擁有可變的輸入工作頻率，頻率範圍從 0 至 25MHz。
- 提供軟/硬體防寫與保護功能。
- 支援偵測卡片插入/拔除功能。

3.1.1 SD 記憶卡的記憶體配置

SD 記憶卡儲存的基本單位為一個位元組，而所有關於資料傳輸的操作是以 Block 為單位，較常使用的 Block 容量為 512 的位元組，圖 3-2 為 SD 記憶卡針對記憶體的讀取與寫入操作，主要分為 Block、Sector 與 WP Group 三種容量結構 [5]。

Block

Block 是 SD 記憶卡操作讀取與寫入命令的相關單位，Block 是由一連串的位元組(Byte)所構成的，Block 的大小可以是固定的或是由程式來設定，其定義在 SD 記憶卡的 CSD 暫存器中，讀取 READ_BL_LEN 與 WRITE_BL_LEN 由(12)與(13)計算出 Block 的大小，透過 SET_BLOCK_LEN 命令可以來改變 Block 的容量大小。

$$N_{RBS} = 2^{READ_BL_LEN} \tag{12}$$

$$N_{WBS} = 2^{WRITE_BL_LEN} \tag{13}$$

其中 N_{RBS} 為讀取時的區塊容量；N_{WBS} 為寫入時的區塊容量。

Sector

Sector 是抹除命令的相關單位，由數個 Block 所構成的，SD 記憶卡的 Sector 大小是固定的不可變更，其定義在 SD 記憶卡的 CSD 暫存器中的 SECTOR_SIZE 位元，SECTOR_SIZE 為 7 位元資料格式，數值為 0 表示為 1 個 Block，數值為 127 表示為 128 個 Block。

$$N_{Sector} = (N_{Sector_Reg} + 1) \times N_{Block_size} \tag{14}$$

其中 N_{Sector} 為 Sector 的容量；N_{Sector_Reg} 為 CSD 暫存器中的 SECTOR_SIZE 位元；N_{Block_size} 為 Block 的容量。

WP Group

WP Group 是保護寫入命令的最小單位，是由數個 Sector 所構成的，在 CSD 暫存器的 WP_GRP_SIZE 中定義其大小，並且使用 WP_GRP_ENABLE 位元來設定是否開啟保護寫入的功能。

$$N_{WPG} = (N_{WP_GRP_size} + 1) \times N_{Sector} \quad (15)$$

其中 $N_{WP_GRP_size}$ 為 CSD 暫存器中的 WP_GRP_SIZE 位元；N_{WPG} 為 WP Group 的容量。

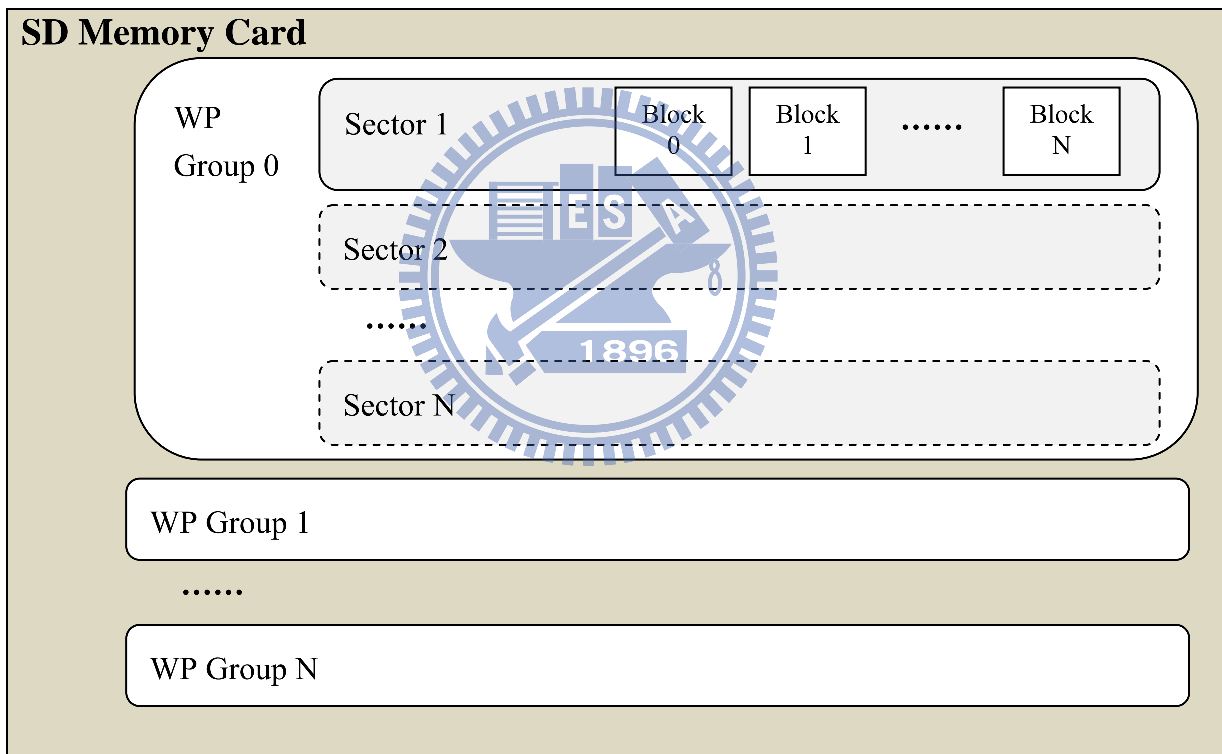


圖 3-2 SD 記憶體配置方式 [5]

3.1.2 SD 記憶卡架構

SD 記憶卡的內部硬體電路架構，如圖 3-3 所示，可分為介面驅動器、卡介面控制器、內部暫存器、電源起動偵測、記憶體控制介面與記憶體區域等部份 [4]。

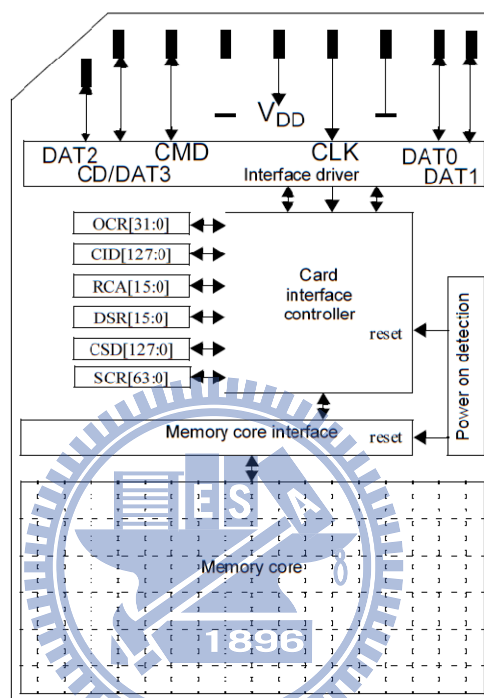


圖 3-3 SD 記憶卡架構 [4]

介面驅動器(Interface driver)

用來與主裝置溝通的介面驅動器，主要的溝通介面有相容於 MMC 記憶卡的 SPI 模式與高速的 SD bus 模式兩種。

卡介面控制器(Card interface controller)

記憶卡的主要控制核心，用來將命令與資料進行編/解碼的動作。

內部暫存器(Registers)

SD 記憶卡共有 6 個內部暫存器，其為 OCR、CID、RCA、DSR、CSD 與 SCR，主要是存放記憶卡的各種參數與設定。

電源起動偵測(Power on detection)

電源起動偵測電路在偵測到電源起動的狀態後，會將記憶卡重置(Reset)並且進入初始狀態。

記憶體控制介面(Memory coard interace)

用來控制記憶體讀取資料和資料寫入記憶體的動作，在進行寫入記憶體的動作時，會將記憶體位址重新編/解碼，避免 NAND Flash 記憶體同一個位址寫入太多次，造成記憶體損壞。

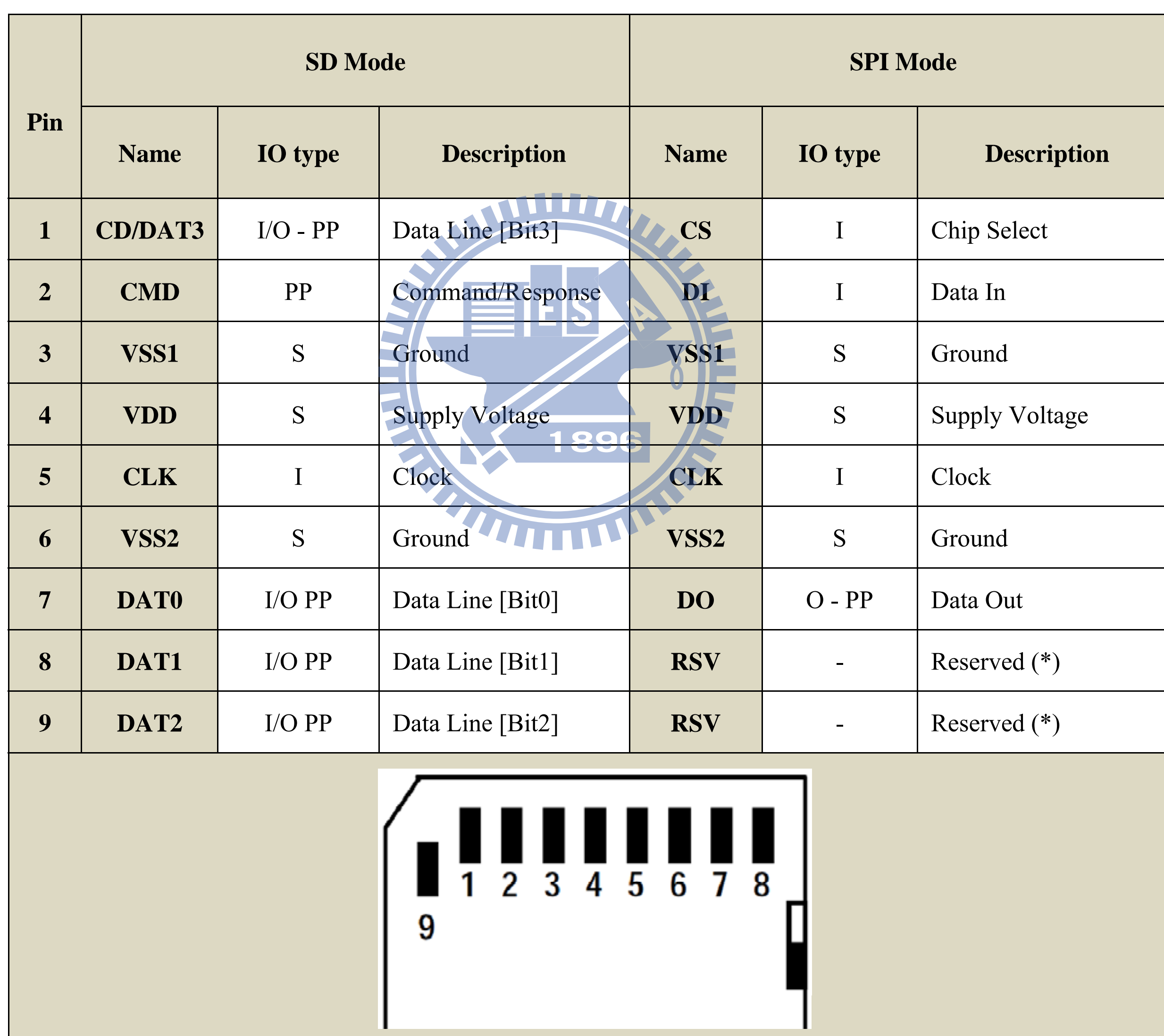
記憶體區域(Memory core area)

SD 記憶卡用來儲存資料的區域是使用 NAND Flash 記憶體，，SD 規格可支援容量為 32MB、64MB、128MB、256MB、512MB、1024MB 與 2048MB。而 SDHC 規格可支援容量超過 2GB 以上。

3.1.3 SD 記憶卡的匯流排結構

SD 記憶卡支援 2 種傳輸模式：SPI 模式(相容於 MMC 介面)與 SD Bus 模式。表 3-1 列出了 SD 記憶卡的連接介面接腳共有 9 根接腳，在 SPI 模式下使用了 CS、DI、DO、CLK 等 4 根接腳；在 SD Bus 模式下使用了 CMD、CLK、DAT0、DAT1、DAT2 與 DAT3 接腳來傳遞訊號。

表 3-1 SD 記憶卡接腳功能描述 [6]



Pin	SD Mode			SPI Mode		
	Name	**IO type**	**Description**	**Name**	**IO type**	**Description**
1	**CD/DAT3**	I/O - PP	Data Line [Bit3]	**CS**	I	Chip Select
2	**CMD**	PP	Command/Response	**DI**	I	Data In
3	**VSS1**	S	Ground	**VSS1**	S	Ground
4	**VDD**	S	Supply Voltage	**VDD**	S	Supply Voltage
5	**CLK**	I	Clock	**CLK**	I	Clock
6	**VSS2**	S	Ground	**VSS2**	S	Ground
7	**DAT0**	I/O PP	Data Line [Bit0]	**DO**	O - PP	Data Out
8	**DAT1**	I/O PP	Data Line [Bit1]	**RSV**	-	Reserved (*)
9	**DAT2**	I/O PP	Data Line [Bit2]	**RSV**	-	Reserved (*)

SD Bus 結構

SD Bus 匯流排有 6 根訊號線(CMD、CLK、DAT0-3)與 3 根電源線(參考表 3-1)，圖 3-4 顯示主裝置在並聯使用 SD 記憶卡的狀況下，所有記憶卡的訊號是個別連接，使主裝置可以同時操作所有記憶卡。主裝置在初始化記憶卡的期間，命令被個別地送到每一個記憶卡，並且分發邏輯位址(RCA)給每一個記憶卡 [6]。

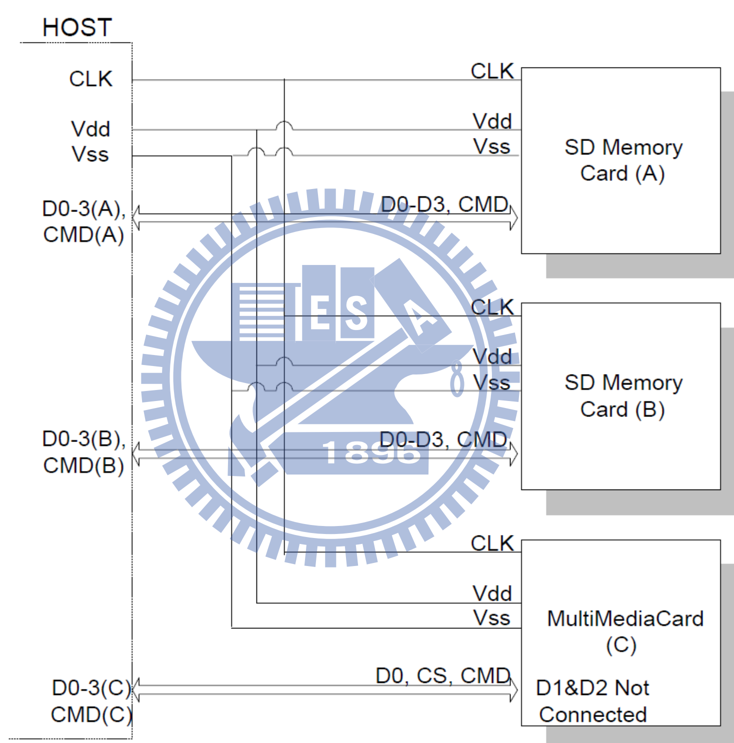


圖 3-4 SD Bus 架構 [6]

SPI Bus 結構

SPI Bus 匯流排有 4 根訊號線(CMD、CLK、CS、DAT)與 3 根電源線，圖 3-5 顯示主裝置操作在 SPI 模式的狀況下，所有記憶卡的 CLK、CMD 與 DAT 訊號線是接在一起的，每一個記憶卡有獨立的 CS 選擇接腳，主裝置可透過 CS 接腳來選擇要使用的記憶卡。其執行動作為發送每一個命令時，將 CS 接腳設為低電位(Active Low)，則卡片被選中，並在整個 SPI Bus 傳輸過程當中必須持續為低電位 [6]。

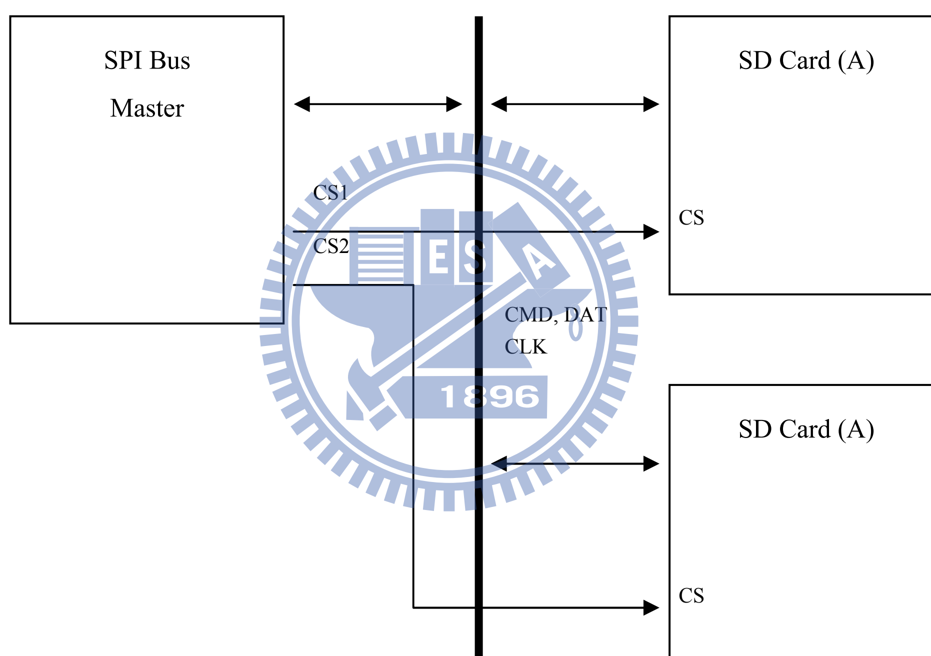


圖 3-5 SPI Bus 架構 [6]

3.1.4 SD 記憶卡的暫存器

在使用 SD 記憶卡時，必須知道卡片的詳細資訊才能夠正確的進行操作，而卡片的各種詳細資訊就存放在內部暫存器中，表 3-2 列出了 SD 記憶卡內部的六個暫存器，其為 OCR、CID、CSD、RCA、DSR 與 SCR，要存取這些暫存器的資料需要透過相對應的命令才能讀取，詳細的暫存器內容說明請參考 SD Memory Card Specification [4]。

表 3-2 SD 記憶卡內部暫存器 [4]

Name	Width	Description
CID	128	Card identification number; card individual number for identification
RCA	16	Relative card address; local system address of a card, dynamically suggested by the card and approved by the host during initialization
DSR	16	Driver Stage Register; to configure the card's output drivers
CSD	128	Card Specific Data; information about the card operation conditions
SCR	64	SD Configuration Register; information about the SD Memory Card's Special Features capabilities
OCR	32	Operation condition register

OCR 暫存器 (Operation Conditions Register)

OCR 暫存器有 32 位元，是儲存 SD 記憶卡的操作電壓範圍，由於記憶卡的匯流排並不支援所有的操作電壓，所以 OCR 暫存器會提供這張記憶卡的操作電壓範圍。每一個位元對應一個電壓檔位，如果該位元為 1，則表示支援該電壓範圍；反之則不支援該電壓範圍。

另外，在第 31 位元還包含了 1 位元的狀態資訊，如果記憶卡的電源起動程序執行完成，則狀態資訊位元會設置為 1，當記憶卡在 BUSY 狀態下超過時間，則清除為 0。

當電源起動程序執行完成後，第 30 位元(CCS)會依據記憶卡容量來設定，如為高容量記憶卡，則 CCS 會設置為 1。如電源起動程序無法執行完成，則 CCS 不會被設置。

圖 3-6 為本系統使用的創見(Transcend) 2GB SD 記憶卡，其讀取的 OCR 數值為 0x80FF800，可以得知其操作電壓範圍為 2.7V~3.6V，且記憶卡的狀態為電源起動程序執行完成，不支援高容量模式。讀取 OCR 暫存器的專用命令為 ACMD41 (SEND_APP_OP_COND)。

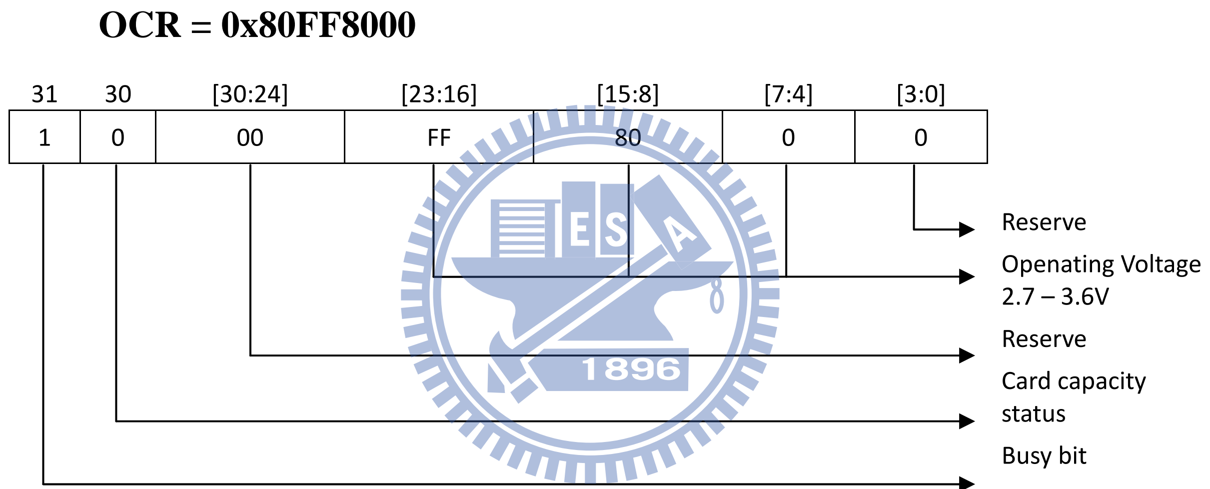


圖 3-6 OCR 結構分析 [4]

CID 暫存器 (Card IDentification)

CID 暫存器的資料格式為 128 位元，使用 CMD10(SEND_CID)命令來讀取，其記錄了該 SD 記憶卡的製造廠商、產品名稱、版本與序號等識別資訊，每一張記憶卡都有一個唯一的識別號碼(PSN)，本系統使用的創見(Transcend) 2GB SD 記憶卡 CID 值為：

CID = { 0x1E，0x41，0x42，0x53，0x44，0x43，0x20，0x20
0x10，0x4B，0x60，0xD0，0x50，0x00，0xA9，0x47 }

CSD 暫存器 (Card-Specific Data)

CSD 暫存器提供如何操作記憶卡的資訊，其為 128 位元資料格式，其定義了資料容量、錯誤校正形式、最大存取時間以及是否可以使用 DSR 暫存器等資訊。可使用 CMD9 (SEND_CSD)命令來讀取 CSD 暫存器的資料。本系統使用的創見(Transcend) 2GB SD 記憶卡 CSD 值為：

CSD = { 0x00，0x2F，0x00，0x32，0x5F，0x5A，0x83，0xB6
0xED，0xB7，0xFF，0xBF，0x96，0x80，0x00，0xF7 }

RCA 暫存器 (Relative Card Address)

RCA 暫存器是一 16 位元的卡片位址暫存器，在卡片識別流程期間登錄，主裝置可用此位址來指定記憶卡，RCA 的初始設定值為 0x0000。CMD3 (Send_Relative_Addr)命令可以用來登錄新的 RCA 位址，可用 CMD7 (Select/ Deselect_Card)命令來選擇特定 RCA 位址的記憶卡。

DSR 暫存器 (Driver Stage Register)

DSR 暫存器儲存記憶卡的命令與資料匯流排驅動電流與斜率，主裝置可以依照 DSR 設定隨意地根據資料線寬度、傳輸速率、卡片數量來改善匯流排驅動能力，出廠設定值為 0x0404。可用 CMD4(SET_DSR) 命令來設定 DSR 暫存器。

SCR 暫存器

64 位元的 SCR(SD CARD Configuration Register)暫存器是用來擴展 CSD 暫存器，其提供該記憶卡特殊功能的設定資訊，此暫存器需在製造後出廠前就被設定好，使用 ACMD51 (SD_APP_SEND_SCR)命令來讀取 SCR 暫存器資料。

3.1.5 CRC 檢查碼

循環冗餘檢查碼(Cyclic Redundancy Check 簡稱 CRC)是用來檢測資料在傳送過程中是否發生錯誤，CRC 演算法是將被保護的資料，用設定好的除數來產生一個餘數，此餘數即為該保護資料的 CRC 檢查碼，傳送端將計算出的餘數和資料一起傳輸，而接收端用此餘數來檢查資料在傳輸過程中是否正常 [6]。

CRC16

CRC16 為 16 位元檢查碼，使用在資料傳遞的過程中，其產生方式如下：

Generator polynomial：$G(x) = x^{16} + x^{12} + x^5 + 1$ (16)

$M(x) = (\text{first bit}) * x^n + (\text{second bit}) * x^{n-1} + \cdots + (\text{last bit}) * x^0$ (17)

$CRC[15\ldots0] = \text{Remainder } [(M(x) * x^{16})/G(x)]$ (18)

CRC7

CRC7 為 7 位元檢查碼，使用在傳送命令和回應(回應 3 例外)上面，以及讀取 CIS 與 CSD 暫存器資料時使用，其產生的計算方式如下：

Generator polynomial：$G(x) = x^7 + x^3 + 1$ (19)

$M(x) = (\text{first bit}) * x^n + (\text{second bit}) * x^{n-1} + \cdots + (\text{last bit}) * x^0$ (20)

$CRC[6\ldots0] = \text{Remainder } [(M(x) * x^7)/G(x)]$ (21)

“frst bit”是位元流的最左邊的位元，多項式“n”是要保護資料的位元數減 1。例如命令為 48 位元，減去 7 位元的 CRC 與 1 位元的停止位元，則需要保護的位元數為 40 位元(n = 39)。

舉例 CRC7 來說明，CRC 檢查碼的餘數計算可以通過一個 8 位元的移位暫存器來運算，將欲保護的資料移入暫存器中，並保持移位暫存器的 MSB 為 1，把移位暫存器的值與多項式$G(x) = x^7 + x^3 + 1$做互斥或(XOR)運算，直到餘數小於除數，此餘數即為所需的 CRC 檢查碼。例如 SD 記憶卡的 CMD0 命令，CRC7 值為 0x4A，圖 3-7 為其運算過程。

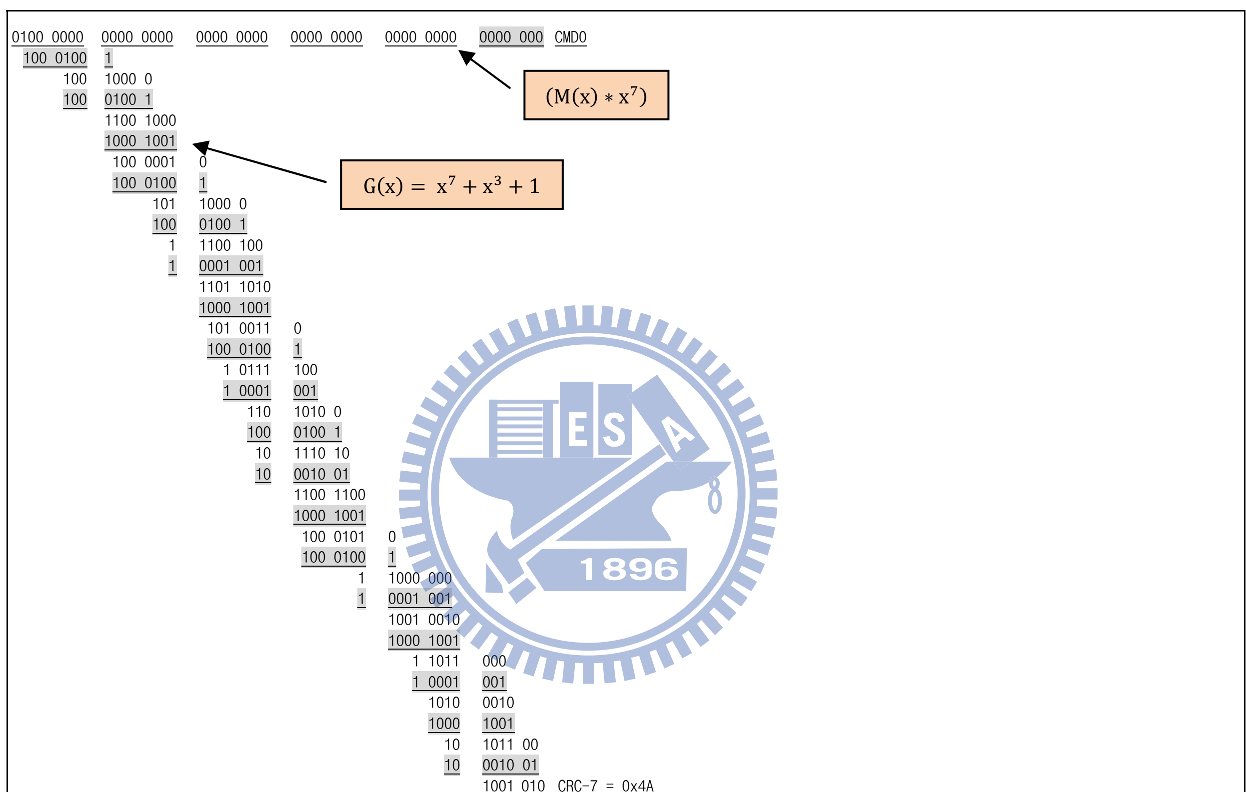


圖 3-7 CMD0 的 CRC7 計算過程

由於 CRC 檢查碼的產生與驗證需要佔用系統許多資源與運算時間，因此當本系統工作在 SPI 模式時，會將 SD 記憶卡的 CRC 驗證功能，透過 CMD59(CRC_ON_OFF)命令來關閉，以降低系統軟體與微控器的負擔。如果系統工作在 SDIO 模式時，STM32 的 SDIO 模組已經內建 CRC 產生與驗證的硬體電路，因此不會增加軟體與微控器的負擔。

3.2 SPI 傳輸模式

SD 記憶卡的傳輸是由命令和資料流所構成的，SPI 格式是位元流的傳輸模式，由一個起始位元開始與一個停止位元來表示傳輸結束，命令或資料由 8 位元為一基本傳輸單位，並對齊 CS 訊號。

SD 記憶卡的 SPI 匯流排介面，相容於 MMC 記憶卡規格，使用 4 根訊號線，其訊號線定義為 CLK：主裝置時脈訊號、DataIn：主裝置資料輸出訊號、DataOut：主裝置資料輸入訊號、CS：主裝置卡片選擇訊號。

SPI 傳輸模式是由命令(Command)、回應(Response)與資料區塊(Data-block tokens)所構成。所有主裝置與記憶卡之間的溝通都是由主裝置來控制，主裝置每次操作都需將 CS 信號設為低電位 [6]。

圖 3-8 為 SPI 發送命令與接受回應的時序圖，在此過程中 CS 訊號線需要保持在低電位，以表示記憶卡被選擇。當主裝置開始輸出工作頻率時，CMD 訊號線就可以開始送出命令，此時 DAT 訊號線保持在高電位，直到記憶卡發出資料訊息。

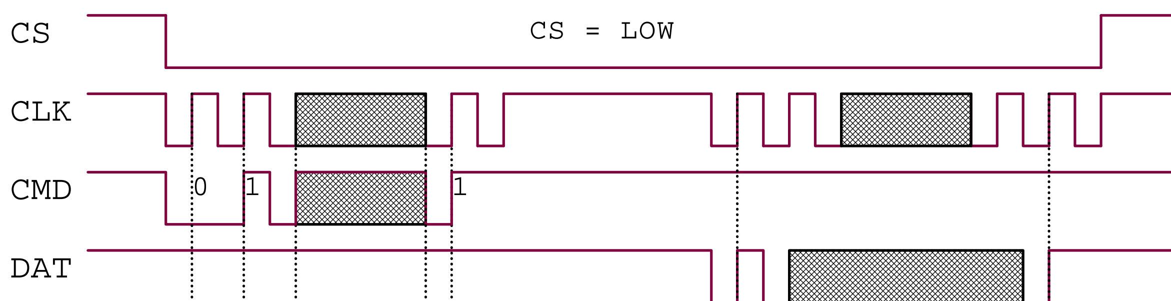


圖 3-8 SPI 時序 [6]

3.2.1 SPI 模式選擇

SD 記憶卡的預設傳輸模式為 SD Bus 模式，因此須執行特定流程來將操作模式切換為 SPI 模式。在電源上電後的第一個命令必須是重置命令 CMD0 (GO_IDLE_STATE)，並且記憶卡選擇訊號(CS)必須為低電位，如果記憶卡進入 SD Bus 模式，則不會回應命令，並保持在 SD Bus 模式下。如果收到命令並且成功切換為 SPI 模式，則記憶卡會發出正確的命令回應，只有重新進行電源啟動流程，才能返回到 SD Bus 模式 [6]。

圖 3-9 為 SD 記憶卡切換到 SPI 工作模式的時序圖，依照 SD 規格書 [6] 的要求，主裝置需要先發送至少 74 個 SPI 頻率週期給記憶卡，然後將 CS 訊號線保持在低電位，並且發送 CMD0 命令，如果 SD 記憶卡發出空閒狀態(In_Idel_State)的回應，則表示成功進入 SPI 模式。

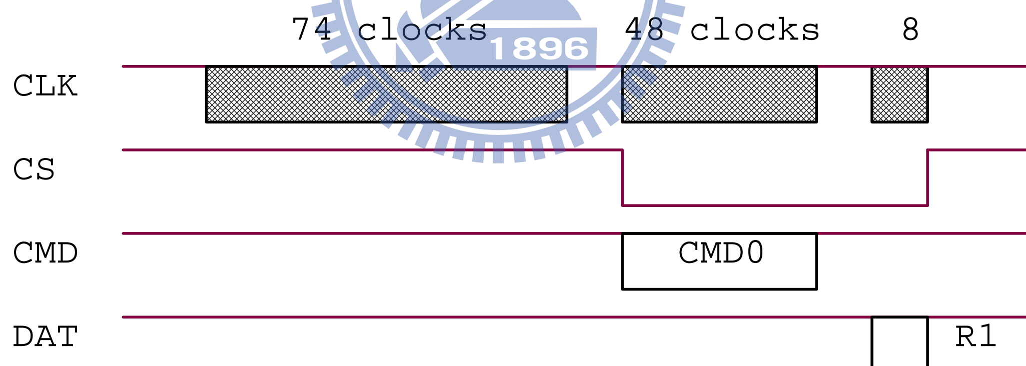


圖 3-9 SPI Power On 時序 [6]

SD 記憶卡的電源啟動流程依照 SD 規格書 [6] 所定義之時序，如圖 3-10 所示，可分為以下 3 個步驟：

Power up Time

開始供應電壓，達到 $V_{DD(min)}$超過 2.0V 為止。

Supply ramp up time

上電過程中電壓達到 2V 後，主裝置啟動 CLK 與 CMD 訊號線來執行初始化程序，這個程序是一串連續的 1，且時間最長為 1ms 或 74 個 SPI CLK 週期，以便 SD 記憶卡完成內部初始化程序。

Initializattion process

SD 記憶卡在上電後(包括熱插拔)會進入空閒狀態(idle state)，在這種狀態下，SD 記憶卡將會忽略所有在匯流排上執行的傳輸訊號，直到收到 ACMD41 命令。ACMD41 命令是一種特殊的同步命令用於取得卡片的工作電壓範圍，並可判斷上電程序是否完成。第 31 位元為忙碌標誌，用來表明該記憶卡是否仍在執行上電程序，主裝置必須等待該位元設為 1。每張憶卡的上電程序時間不得超過 1 秒。

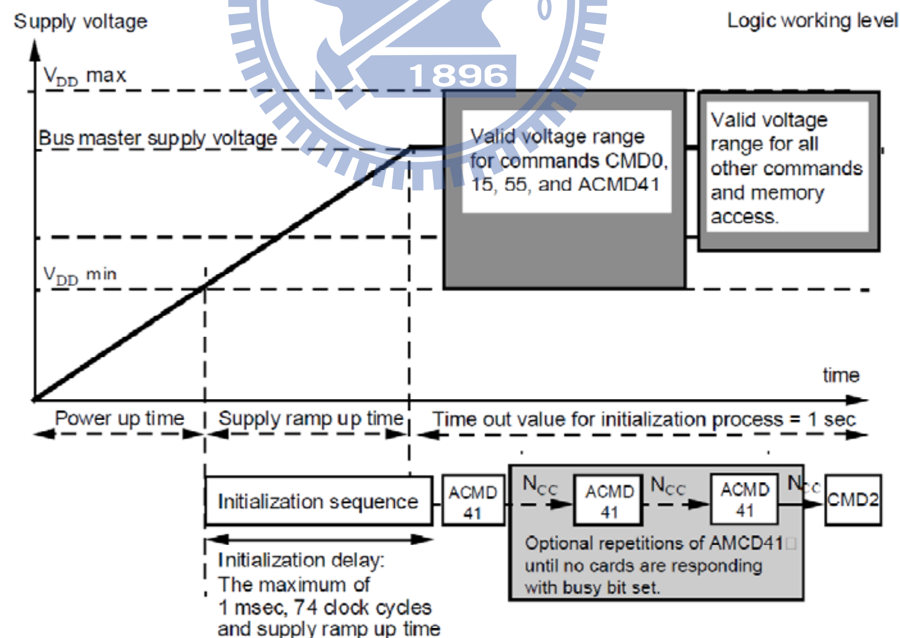


圖 3-10 SD 記憶卡上電時序 [6]

3.2.2 命令格式

SD 記憶卡的命令是採用 48 位元的位元流資料格式，並以 MSB 為優先傳送，命令的內容如表 3-3 可分成 6 個部分 [6]：

開始位元(Start bit)，第 47 個位元，永遠為 0。

傳輸位元(Transmission bit)，第 46 個位元，永遠為 1。

命令索引(Command index)，6 位元的命令索引，是以二進制來編碼。舉例來說，命令 CMD0 其編碼為'000000'、CMD39 其為'100111。

命令參數(Argument)，32 位元的參數資料。

CRC 校驗碼，使用 7 位元的 CRC 檢查碼。

停止位元(Stop bit) 最後一個位元，永遠為 1。

表 3-3 SD 記憶卡命令格式 [6]

Bit position	**47**	**46**	**[45:40]**	**[39:8]**	**[7:1]**	**0**
Width (bits)	1	1	6	32	7	1
Value	0	1	X	X		1
Description	Start bit	Transmission bit	Command index	Argument	CRC7	End bit

SD 記憶卡的命令被分為 12 種不同的類型，每一類命令支援一組功能。CSD 暫存器的 CCC[11:0]參數用來記錄可用的命令類型，然而 SD Bus 模式與 SPI 模式的可用命令與支援的類型有所不同，其中類型 0、2、4、5 與 8 是每張卡都必須支援的命令組，而 SPI 模式不支援類型 1、3 與 9 [4]，詳細的命令類型分類請參考附錄二。

SPI 模式的命令發送程式

SPI 模式發送命令的副程式流程如圖 3-11，首先將 6 位元的命令索引 (command index) 加入開始位元(start bit)與傳送位元(transmission bit)，利用 cmd or 0x40 的方式合成為 8 位元，接著將 32 位元參數(argument)分成 4 個位元組由高位元組先傳送再送低位元組，最後傳送 7 位元的 CRC7 檢查碼。當命令傳遞完成後繼續發送 SPI 週期，用以等待記憶卡的回應，並設定等待回應的重試次數，最後將回應結果回傳就完成一個完整的命令傳遞程序 [1], [6]。

程式名稱：SD_SendCMD ( )	輸　入：命令索引 cmd，命令參數 arg，CRC7
功能敘述：傳送命令，並等待回應(R1)	輸　出：回應 R1

```
u8 SD_SendCMD (u8 cmd, u32 arg, u8 crc)
{  u8 r1;     u16 retry=0;
   SPI_RWByte (cmd | 0x40);                    // 命令索引 | 0x40，用來產生傳送位元
   SPI_RWByte (arg >> 24);
   SPI_RWByte (arg >> 16);
   SPI_RWByte (arg >> 8);
   SPI_RWByte (arg);
   SPI_RWByte (crc);                           // CRC7
   do {                                        // 等待回應
       r1 = SPI_RWByte (DUMMY);                // 讀取回應
   } while((r1 == 0xff) && (retry++ <=5000));  // 無回應或超過時間
   return r1;                                  // 返回回應值
}
```

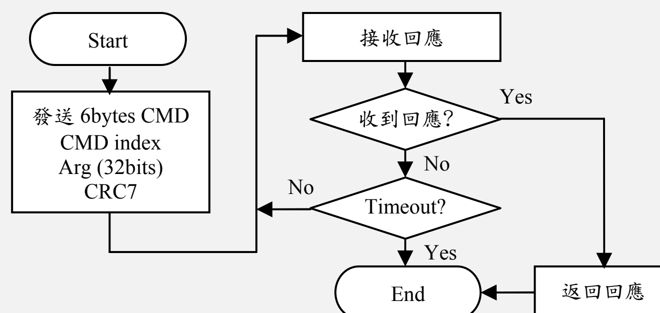


圖 3-11 SPI 模式傳送命令副程式

圖 3-12 為 SPI 發送 CMD0 (GO_IDLE_STATE)的波形，第一個位元組為命令索引 CMD0 (000000b)，第二到第五位元組為命令參數 0x00000000，最後一個位元組為 CRC7 檢查碼再加一個停止位元，CRC7 為 0x4A，記憶卡返回 CMD0 回應為 0x01，表示記憶卡在空閒狀態。

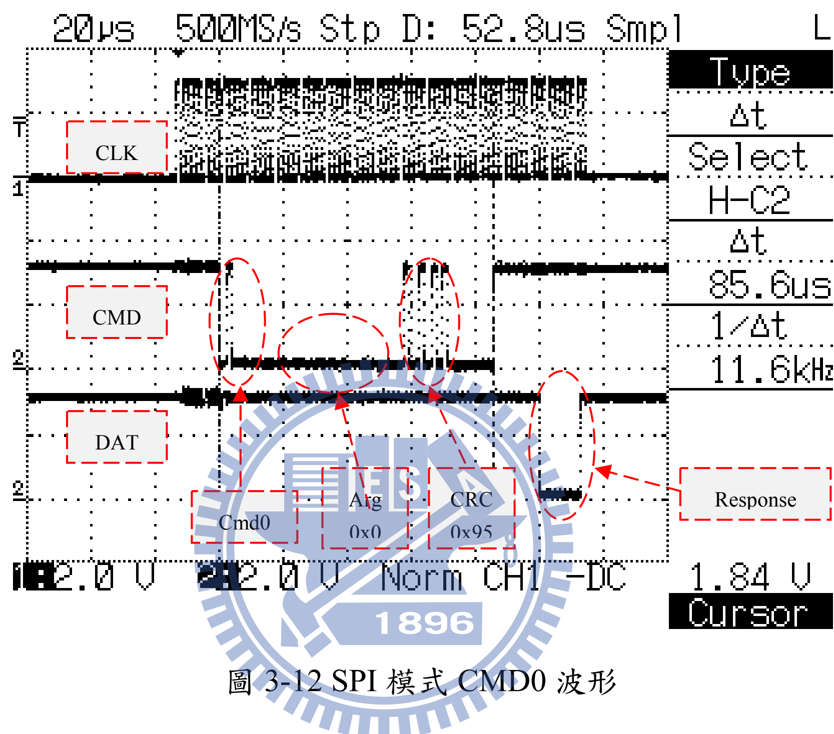


圖 3-12 SPI 模式 CMD0 波形

3.2.3. SPI 模式的回應

SD 記憶卡在接收到命令後會發出回應，且針對不同的命令類型會發出不同型式的回應，命令依照回應可分為四個類型，不需要回應的廣播命令(bc)、需要回應的廣播命令(bcr)、位址命令無資料傳遞(ac)與位址命令有資料傳遞(adct) [4]。

回應 1 (R1)

SD 記憶卡在接收到每一個命令，都會發出回應 1(R1)，除了 CMD13 (Send_Status)命令外。回應 1 的資料長度只有 8 位元，而且高位元永遠為 0，如果其他位元為 1，則表示有錯誤訊息，表 3-4 列出了回應 1 的每個位元所代表的意義。

表 3-4 SPI 回應 1 格式 [6]

Bit	Field	Description
0	**In idle state**	The card is in idle state and running the initializing process
1	**Erase reset**	An erase sequence was cleared before executing because an out of erase sequence command was received.
2	**Illegal command**	An illegal command code was detected
3	**Communication CRC error**	The CRC check of the last command failed
4	**Erase sequence error**	An error in the sequence of erase commands occurred
5	**Address error**	A misaligned address, which did not match the block length, was used in the command
6	**Parameter error**	The command's argument (e.g. address, block length) was out of the allowed range for this card
7	**Reserved**	Always 0

回應 1b (R1b)

回應 1b 的高位元組資料型態與回應 1 相同，再加上一個忙碌(Busy)訊號來表示記憶卡忙碌中。忙碌訊號的格式為所有位元都為 0，且沒有數量限制，當回應訊號恢復為高電位時，則可以接收下一個命令，圖 3-13 為記憶卡發出回應 1b 的時序，其中忙碌(Busy)訊號跟隨在回應 1 之後。

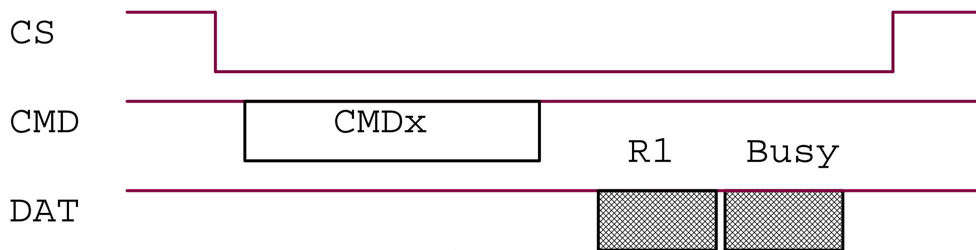


圖 3-13 SPI 回應 1b 的時序 [6]

回應 2 (R2)

回應 2 的資料格式有 2 個位元組，第一個位元組與回應 1 相同，而第二個位元組為記憶卡的狀態資料，圖 3-14 為回應 2 的時序圖，表 3-5 為回應 2 的第二個位元組所代表的意義。

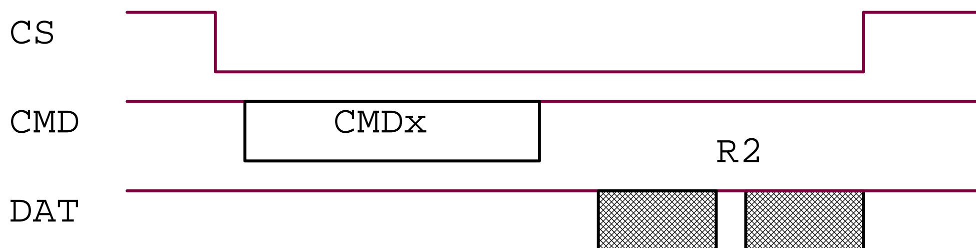


圖 3-14 SPI 回應 2 的時序 [6]

表 3-5 SPI 回應 2 格式 [6]

Bit	Field	Description
15 : 8	R1	Same as response 1
7	Out of range /CSD overwrite	
6	Erase param	An invalid selection, sectors or groups, for erase
5	Write protect violation	The command tried to write a write protected block
4	Card ECC failed	Card internal ECC was applied but failed to correct the data
3	CC error	Internal card controller error
2	Error	A general or an unknown error occurred during the operation
1	WP erase skip, lock/unlock cmd failed	This status bit has two functions overloaded. It is set when the host attempts to erase a write protected sector or makes a sequence or password error during card lock/unlock operation
0	Card is lock	Set when the card is locked by the user. Reset when it is unlocked

回應 3 (R3)

回應 3 為記憶卡收到 CMD58 (Read_OCR)命令時所發出的特別回應訊號，其資料長度為 5 個位元組，圖 3-15 為回應 3 的資料格式，第一個位元組為回應 1 格式，剩餘的 4 個位元組為 OCR 暫存器的資料。

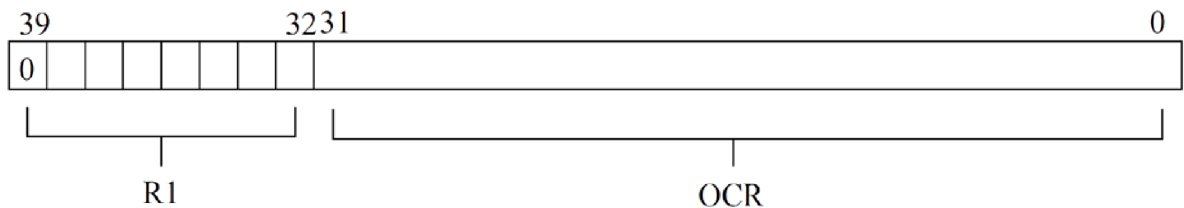


圖 3-15 SPI 回應 3 格式 [4]

回應 4 與 5 (R4 & R5)

這兩個回應為保留給 SD I/O 模式使用，SD 記憶卡不使用。

回應 7 (R7)

回應 7 是記憶卡收到 CMD8(SEND_IF_COND)命令時所發出的，其資料長度為 5 個位元組，最高位元組格式與回應 1 相同，其他四個位元組包含記憶卡的工作電壓資訊和 echo-back 檢查形態，圖 3-16 為回應 7 的資料結構。

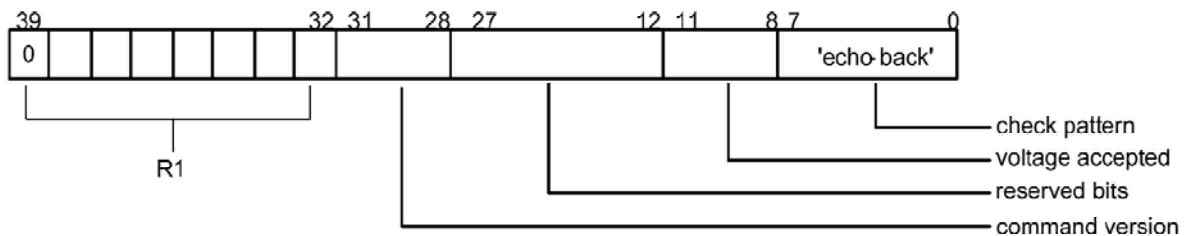


圖 3-16 SPI 回應 7 格式 [4]

資料溝通的回應

主裝置與記憶卡在進行讀寫過程中都需要有資料溝通回應來仲介，用以得知目前的操作狀態。在讀取一個或多個區塊資料與寫入單一個區塊資料時，其資料結構為第一個位元組為開始資料傳遞回應 **0xFE**，2 至 513 位元組為資料，最後 2 個位元組為 16 位元的 CRC 檢查碼。而在寫入多個區塊資料時，主裝置發出開始資料傳遞的回應值為 **0xFC**，結束資料傳遞的回應值為 **0xFD**。

圖 3-17 為主裝置發送 CMD25 命令來寫入多個區塊資料時的時序圖，主裝置在記憶卡發出回應 1 後，發出資料傳遞回應 0xFC 並送出區塊資料，記憶卡每收到一個區塊資料後，會發出接收狀態的資料回應給主裝置，如果接收成功，則主裝置繼續送出下一個區塊資料，最後在資料結束後發出資料傳遞回應 0xFD 來結束資料傳遞。

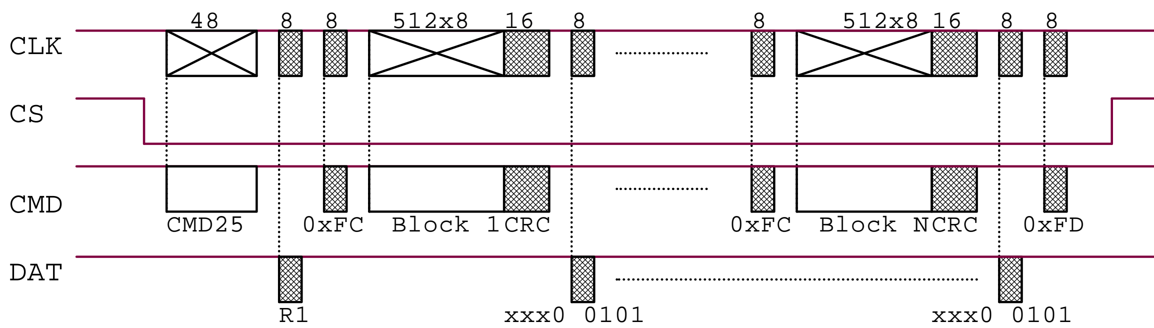


圖 3-17 SPI 開始 0xFC 與結束 0 xFD 傳送回應的時序 [6]

資料回應

主裝置每寫入一個區塊的資料後，記憶卡都會發出資料回應來確認資料是否正確寫入，其資料格式為 5 位元。圖 3-18 為資料回應的格式，位元 7 至 5 不使用，位元 4 固定為“0”，位元 3 至 1 為資料狀態位元，其中“010”表示資料接收成功，“101”表示有 CRC 錯誤資料拒絕接收，“110”表示發生寫入錯誤資料拒絕接收，位元 0 固定為“1”。

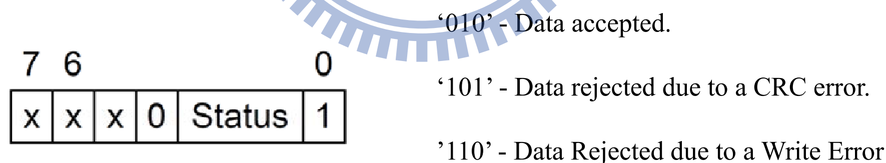


圖 3-18 SPI 資料回應格式 [4]

當執行多個區塊寫入時，如資料回應有任何的錯誤訊息，則需要立即使用 CMD12 來停止寫入動作。如果錯誤訊息為 110b，主裝置還可以透過 CMD13 來得到寫入錯誤的原因，圖 3-19 為寫入單個區塊的操作時序，主裝置發送 CMD12 命令，記憶卡發出回應 1，主裝置再發送“0xFE”表示開始傳遞資料，接著送出 512 位元組資料和 CRC16，最後記憶卡返回資料回應來表示資料是否接收成功。

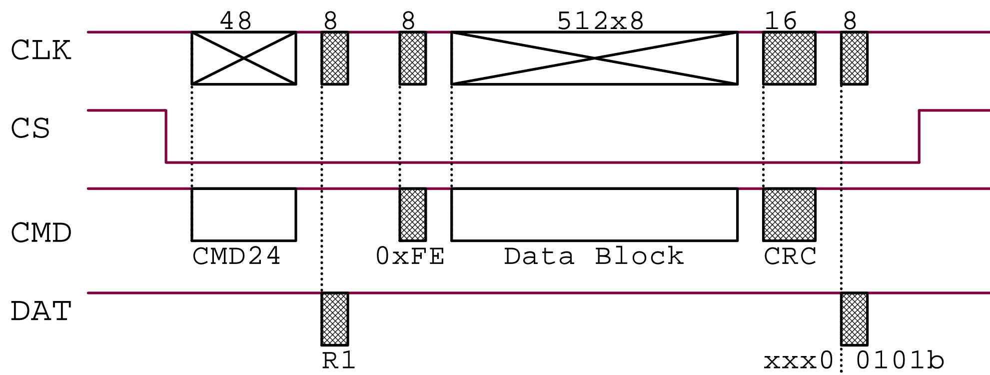


圖 3-19 SPI 資料回應的時序 [6]

資料溝通錯誤的回應

如果讀取動作發生錯誤造成記憶卡無法提供資料，則記憶卡會回應資料溝通錯誤，其資料格式如圖 3-20，長度為 8 位元，並且只使用 4 個位元來表示錯誤狀態，錯誤狀態與回應 2 一樣。

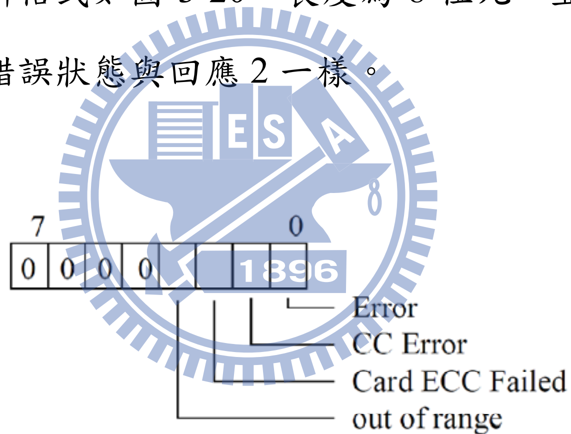


圖 3-20 SPI 資料錯誤回應 [6]

3.2.4 卡識別模式

主裝置在電源重置與尋找新的記憶卡時會進入卡識別模式流程，其主要目的為識別記憶卡的種類，如 MMC、SD V1.0、SD V2.0 落 SDHC 等，並且讀取記憶卡的暫存器資料。圖 3-21 為 SD 規格書 [4] 所公佈的 SPI 模式卡識別流程，其使用 CMD8 命令來判別是否為 V2.0 規格，CMD58 命令來判別是否為 SDHC 規格。

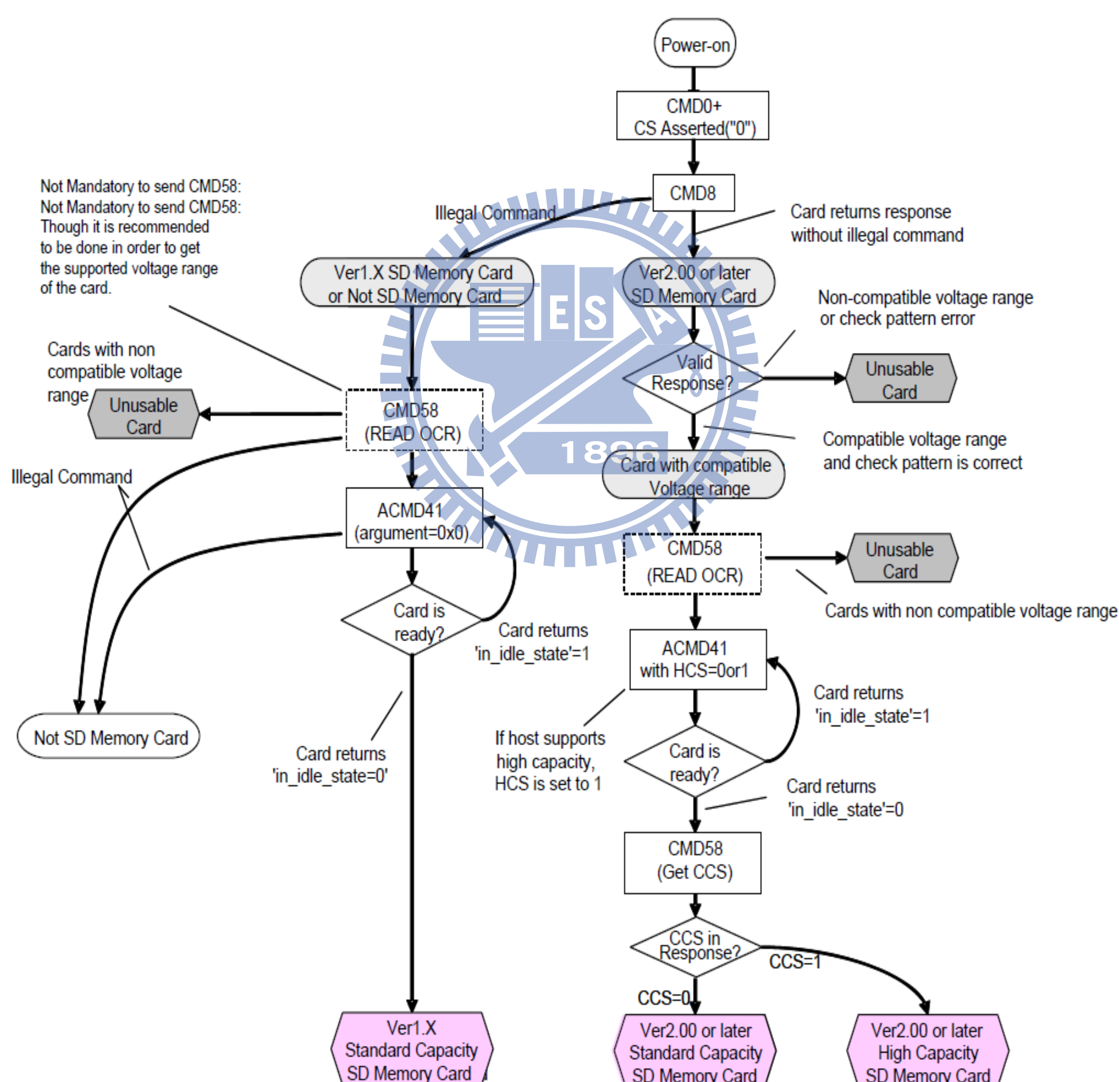


圖 3-21 SPI 模式的卡判斷流程 [4]

SPI 模式的卡識別流程說明與範例程式如圖 3-22。

在電源開啟後，發送 CMD0 (GO_IDLE_STATE)命令並將 CS 接腳輸出低電位，使記憶卡切換到 SPI 模式。

發送 CMD8 (SEND_IF_COND)命令，來判斷記憶卡是否為 V2.0 以上的版本。如果記憶卡發出回應 7，則表示此記憶卡為 V2.0 以上的版本。

再發送 ACMD41 (SEND_OP_COND)命令來讀取 OCR 暫存器值，並且鑑別記憶卡是否為支援高容量模式。

如果記憶卡對 CMD8 命令的 R1 回應為非法命令狀態(0x05)，則此記憶卡為 V1.0 版本或是 MMC 卡。

再發送 ACMD41 (SEND_OP_COND)命令來讀取 OCR 暫存器值，如果記憶卡對 ACMD41 命令的發出回應，則表示此記憶卡為 V1.0 版本。

如果記憶卡對 ACMD41 命令無任何回應，則表示此為 MMC 記憶卡。

程式名稱：SD_initialize ( ) 　　**輸　入：無**

功能敘述：執行電源啟動程式、切換為 SPI 模式、判斷記憶卡型式 　　**輸　出：SD 卡狀態(Stat)**

```
u8 disk_initialize (void)
{  u8 r1, buff[7];      u16 retry=0;
   if (SD_INS())        return SD_EXIT;                 // 檢查 SD 卡是否有插入?
   SPI_Configuration ();                                // 初始化 SPI 模組
   SD_PWROFF ();                                        // 先關閉電源
   Delay (5000000);                                     // 延遲 140m sec
   SD_PWRON ();                                         // 供給電源
   SPI_Delay (500);                                     // 輸出大於 74 個 SPI clock cycle
   Stat |= SD_NOINIT;                                   // 設定 SD 狀態為未初始化
   do {  r1 = SD_SendCMD (CMD0, 0, 0x95, 1);            // 發送 CMD0 (go idle state)
         if (retry++ > 1000)       return r1;           // 超時設定
   } while (r1 != 0x01);                                // 是否有 R1 回應，且 R1 = 0x01
```

```
    r1 = SD_SendCMD (CMD8, 0x1aa, 0x87, 0)                    // 發送 CMD8，V2.0 的命令
    if (r1 == 0x05)                                           // 不接受 CMD8，為 V1.0 或 MMC
    {  SD_Disable ();                                         // 卡選擇 CS 拉 HIGH
       SD_Type = SD_TYPE_V1;                                  // 設定 SD 型號為 V1.0
       SPI_RWByte (DUMMY);
       retry = 0;
       do {  r1 = SD_SendCMD (CMD55, 0, 0, 1);
             r1 = SD_SendCMD (ACMD41, 0, 0, 1);               // 發送 ACMD41，SEND_OP_COND
       }  while ((r1 != 0x00)&&(retry++ < 1000));             // 檢查 R1 回應，R1 = 0x00
       if (retry >= 1000)                                     // 如果超過時間無回應，
       {  retry = 0;
          do {  r1 = SD_SendCMD (CMD1, 0, 0, 1);              // 發送 CMD1
          }  while ((r1 != 0x00)&&(retry++ < 1000));
          SD_Type = SD_TYPE_MMC;                              // 設定 SD 型號為 MMC
    } }
    else if (r1 == 0x01)                                      // 接受 CMD8，表示為 V2.0 以上
    {  for (retry=0; retry<=4; retry++)
          buff[retry] = SPI_RWByte (DUMMY);                   // 接收 CMD8 的 R7 回應資料
       SD_Disable ();                                         // 卡選擇 CS 拉 HIGH
       do {  r1 = SD_SendCMD (CMD55, 0, 0, 1);
             r1 = SD_SendCMD (ACMD41, 0x40000000, 0, 1);      // 發送 ACMD41，SEND_OP_COND
       }  while ((r1 != 0x00)&&(retry++ < 1000));
       r1 = SD_SendCMD (CMD58, 0, 0, 0);                      // 發送 CMD58，READ_OCR
       for (retry=0; retry<=4; retry++)
          buff[retry] = SPI_RWByte (DUMMY);
       SD_Disable ();                                         // 卡選擇 CS 拉 HIGH
       If (buff[0] &0x10)      SD_Type = SD_TYPE_V2HC;        // 設定 SD 型號為 SDHC
       else                    SD_Type = SD_TYPE_V2;          // 設定 SD 型號為 V2.0
    }
    SPI_SetSpeed (SPI_HIGH);
    r1 = SD_SendCMD (CMD59, 0, 0x95, 1);                      // 發送 CMD59，CRC_ON_OFF
    r1 = SD_SendCMD (CMD16, 512, 0x95, 1);                    // 發送 CMD16，SET_BLOCKLEN
    Stat = SD_INITOK;                                         // 設定 SD 狀態為初始化完成
    return Stat;
}
```

圖 3-22 SPI 模式的卡判斷副程式

3.2.5 資料傳輸

SD 記憶卡的命令與資料傳輸都會受到 CRC 的保護，在 SPI 模式下記憶卡提供了無保護模式(不進行 CRC 檢查)，使系統可以移除硬體或軟體產生和檢查 CRC 的功能，用以降低程式與電路的複雜度。在無保護模式下，命令、回應和數據仍需要 CRC 位元，然而，他們被定義為“don’t card”會被主裝置與僕裝置所忽略，主裝置可以使用 CMD59 (CRC_ON_OFF)命令來打開與關閉 CRC 功能 [6]。

圖 3-23 為開啟或關閉 CRC 功能的副程式，使用 CMD59 命令來啟動 CRC 時，命令參數要設為 1，關閉則設為 0。

程式名稱：SD_EnableCRC ( ) **輸　入：u8 enable　：　1 = 開啟，0 = 關閉**
功能敘述：開啟或關閉 CRC 功能 **輸　出：resp 回應狀態**

```
u8 SD_EnableCRC (u8 enable)
{  u8 resp;
   u32 arg;
   if (enable = 1)          arg = 0x00000001;          // 啟動 CRC，命令參數設 1
   else                     arg = 0x00;                // 關閉 CRC，命令參數設 0
   resp = SD_SendCMD (CMD59, arg, 0);                  // 發送命令
   return resp;
}
```

圖 3-23 開啟或關閉 CRC 功能副程式

SD 記憶卡的資料傳輸以區塊(Block)作為其最小單位，CSD 暫存器的 READ/WRITE_BL_LEN 位元定義區塊的大小，主裝置可以使用 CMD16 (SET_BLOCKLEN)命令來設定區塊的大小。SD 記憶卡一般常定義一個區塊為 512 位元組。較早期的 MMC 卡，其區塊大小可以小於 512 位元組，圖 3-24 為設定區塊大小的 SD_SetBlockLen( )副程式。

程式名稱：SD_SetBlockLen ( )	輸　入：u32 length
功能敘述：設定區塊大小	輸　出：resp 回應狀態

```
u8 SD_sSetBlockLen (u32 length)
{
    u8 resp;
    resp = SD_SendCMD (CMD16, length, 0);
    return resp;
}
```

圖 3-24 設定區塊大小副程式

SPI 模式的資料讀取

SD 記憶卡共有三個讀取命令，CMD17 命令為讀取單區塊資料，CMD18 命令為讀取多區塊資料，CMD12 命令為用來停止資料傳遞。其中 CMD18 多區塊讀取命令需要搭配 CMD12 停止傳遞命令來結束資料的讀取。

單區塊讀取 (CMD17)

SD 記憶卡 CMD17 命令的操作時序如圖 3-25 所示，主裝置發送命令給記憶卡，記憶卡發出回應 0x00 表示成功接收命令，然後等待記憶卡返回資料回應 0xFE 表示資料準備完成後，便可以開始接收資料，當記憶卡傳遞資料完成後會自動進入空閒狀態，並等待下一個命令。

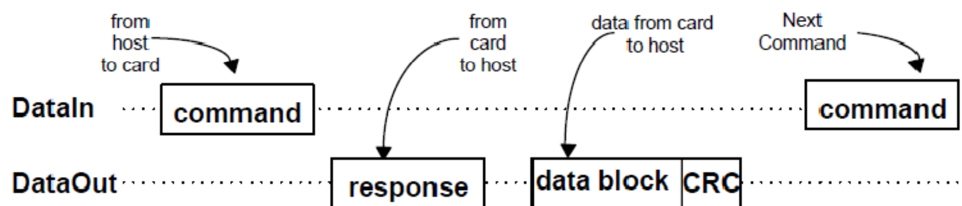


圖 3-25 讀取單區塊(CMD17)的操作時序 [4]

SD 記憶卡在操作讀寫命令時，如果記憶卡的容量型式為 2GB(含)以下的 V2.0 版本，其 32 位元參數為記憶卡的位元組位址。因此我們給予的磁區位址需要乘上磁區容量(512 bytes)，才能定位到正確的資料位址，假設要讀取磁區 2 的資料，參數資料則為 2 × 512；如果記憶卡為高容量型式，則 32 位元參數即為磁區位址，不須要進行轉換。圖 3-26 為使用 CMD17 命令讀取一個磁區資料的副程式，先判別是否為高容量的卡片。如不是，則需將磁區位址乘以 512 進行轉換。然候發送 CMD17 命令並接收資料。

程式名稱：SD_ReadBlock ( )	**輸　入：u32 sector，u8 *buffer**
功能敘述：讀取單區塊資料 (512 bytes)	**輸　出：resp 回應狀態**

```
u8 SD_ReadBlock (u32 sector, u8 *buffer)
{
   if (SD)Type != SDHC)
      sector == sector << 9;                       // 轉換磁區位址
   if (SD_SendCMD (CMD17, sector, 0, 1) != 0x00)   //送出 CMD17 並等待回應成功
      return SD_FAIL;
   SD_ReceiveData (buffer, 512, 1);                //接收 512bytes，並存入 buffer
   return SD_OK;
}
```

圖 3-26 SPI 模式讀取單區塊副程式

多區塊讀取 (CMD18)

SD 記憶卡的 CMD18 命令操作時序如圖 3-27，主裝置發送命令給記憶卡，記憶卡發出回應 1 (0x00)表示成功接收命令，等待記憶卡返回資料準備完成的資料回應 0xFE 後，開始接收資料與 CRC。當主裝置接收完記憶卡傳遞的區塊資料後，我們需要重新等待記憶卡返回區塊資料準備完成的資料回應，才能繼續接收區塊資料，最後由主裝置發送 CMD12 命令來使記憶卡停止資料輸出。

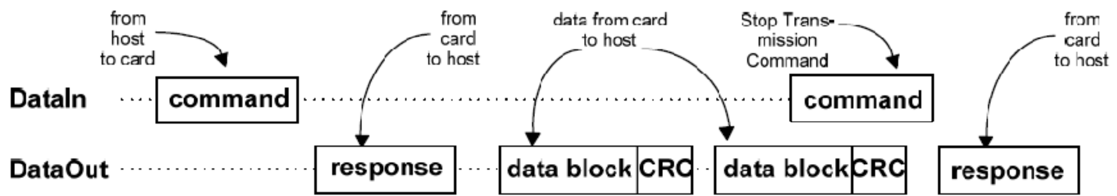


圖 3-27 讀取多區塊(CMD18)的操作時序 [4]

圖 3-28 為使用 CMD18 命令讀取多個磁區資料的副程式，先判別是否為高容量的卡片。如不是，則需將磁區位址乘以 512 進行轉換。然候發送 CMD18 命令並開始接收資料，當資料接收完成後發送 CMD12 命令來使記卡中止資料傳遞。

程式名稱：SD_ReadMultiBlock ( )	輸　入：u32 sector，u8 *buffer，u8 count
功能敘述：讀取多區塊資料	輸　出：resp 回應狀態

```
u8 SD_ReadMultiBlock (u32 sector, u8 *buffer, u8 count)
{
    if (SD)Type != SDHC)
        sector == sector << 9;                        // 轉換磁區位址
    if (SD_SendCMD (CMD18, sector, 0, 1) != 0x00)     //送出 CMD18 並等待回應成功
        return SD_FAIL;
    do {
        if (SD_ReceiveData (buffer, 512, 0) != 0x00)  //接收 512bytes，並存入 buffer
            break;
        buffer += 512;                                //buffer address + 512
    } while (--count);                                //block counter – 1
    SD_SendCMD (CMD12, 0, 0, 1);                      //送出 CMD12
    SPI_RWByte (DUMMY);
    return SD_OK;
}
```

圖 3-28 SPI 模式讀取多區塊副程式

透過 SPI Bus 接收資料

接收區塊資料的 SPI 程式如圖 3-29，當 SD 記憶卡收到讀取命令，並發出正確回應後，主裝置就可以進入接收 512 筆資料 8 位元的程式。等待記憶卡發出資料準備完成的資料回應 0xFE，使用 SPI_RWByte ( )副程式來讀取 1 筆 8 位元的資料，並將資料存入緩衝區內，利用迴圈方式來完成 512 筆的資料接收，最後接收 CRC16 校驗碼。如果關閉記憶卡的 CRC 檢查功能，則亦需發送 2 個 SPI 週期，以完成資料區塊的接收時序。

程式名稱：SD_ReceiveData ( )	輸　入：u8 *buffer，u16 length
功能敘述：接收 SPI 區塊資料 (512 bytes)	輸　出：resp 回應狀態

```
u8 SD_ReceiveData (u8 *data, u16 len, u8 release)
{  u16 retry = 0;
   u8 r1;
   SD_Enable ();                          // 記憶卡選擇 CS = LOW
   do {  r1 = SPI_RWByte (DUMMY);         // 接收資料回應
         if (retry++ > 5000)              // 超時設定 5000 times
            return r1;
   }  while (r1 != 0xFE);                 // 資料準備完成回應 0xFE
   while (len--)                          // 資料長度由 len 到 0
   {  r1 = SPI_RWByte (DUMMY);            // 接收 SPI 資料
      *data = r1;                         // 存入記憶體暫存器
      data++;                             // 記憶體位址加 1
   }
   SPI_RWByte (DUMMY);                    // 接收 CRC16
   SPI_RWByte (DUMMY);
   if (release == 1)                      // 是否釋放 CS
   {  SD_Disable ();                      // 記憶卡不選擇 CS = HIGH
      SPI_RWByte (DUMMY);
   }
   return SD_OK;                          // 返回回應
}
```

圖 3-29 SPI 模式接收資料區塊副程式

SPI 模式的資料寫入

SD 記憶卡的資料寫入也是以一個區塊為基本單位，SPI 模式的寫入命令共有兩個，CMD24 命令可寫入單個區塊資料，CMD25 命令可覆寫多個區塊資料。

單區塊寫入 (CMD24)

SD 記憶卡的 CMD24 命令操作時序如圖 3-30，主裝置發送命令給記憶卡，記憶卡發出回應 0x00 表示成功接收命令，然後主裝置發送資料回應 0xFC 表示開始傳遞資料後，便可以開始傳遞 512 筆資料，當主裝置傳遞單個區塊資料完成後，SD 記憶卡會發出 Data Erroe Token 的回應"00101b"表示資料接受成功，如回應"01011b"表示資料有 CRC 錯誤，如回應"01101b"表示資料寫入錯誤。

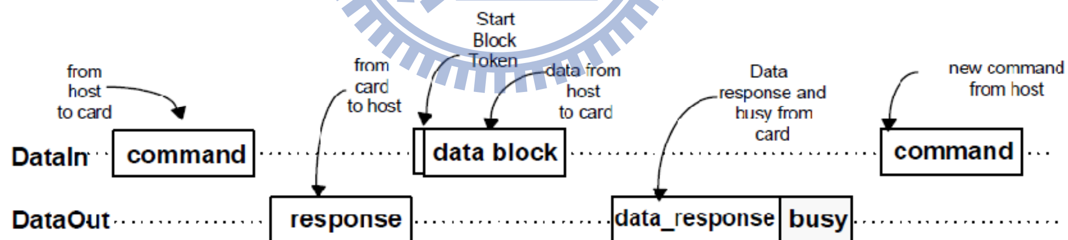


圖 3-30 寫入單區塊(CMD24)的操作時序 [4]

圖 3-31 為寫入單個磁區資料的副程式，先判別是否為高容量的卡片以進行磁區位址的轉換，然候發送 CMD24 命令並開始輸出資料。

程式名稱：SD_WriteBlock ( )	輸　入：u32 sector，u8 *data
功能敘述：寫入區塊資料 (512 bytes)	輸　出：resp 回應狀態

```
u8 SD_WriteBlock (u32 sector, const u8 *data)
{  if (SD)Type != SDHC)
       sector == sector << 9;                        // 轉換磁區位址
   if (SD_SendCMD (CMD24, sector, 0x00, 1) != 0x00)  // 送出 CMD18 並等待回應成功
       return SD_FAIL;
   SD_Enable ();                                     // 記憶卡選擇 CS = LOW
   SD_WriteData (data, 0xFC);                        // 寫入 512byte
   SD_Disable ();                                    // 記憶卡不選擇 CS = HIGH
   SPI_RWByte (DUMMY);
   return SD_OK;
}
```

圖 3-31 SPI 模式寫入單區塊副程式

多個區塊寫入 (CMD25)

圖 3-32 為 SD 記憶卡的 CMD25 命令操作時序，主裝置發送命令給記憶卡，如記憶卡發出回應 0x00 表示成功接收命令，便可以開始進行區塊資料的發送，主裝置先發送資料回應 0xFC 表示開始傳遞資料，便可以開始傳遞 512 筆資料，傳遞完成後，記憶卡會發出回應“00101b”表示資料接受成功。如果記憶卡成功接收資料後，即可繼續發送下一個區塊的資料，當所有區塊資料發送完畢後，主裝置發送停止傳送的資料想應 0xFD 給記憶卡表示資料已發送完畢，然後記憶卡會進入 IDLE 狀態，並等待下一個命令。

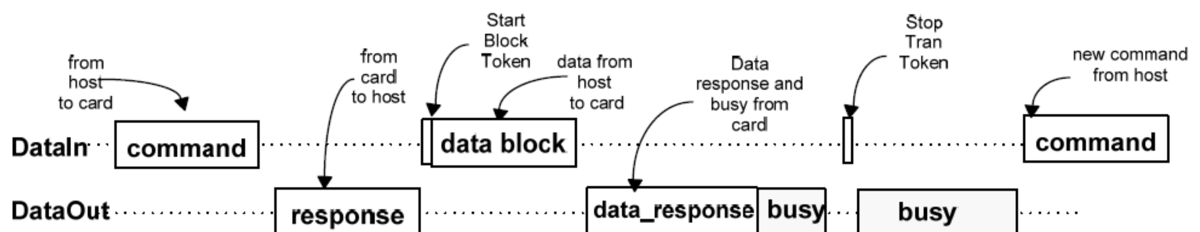


圖 3-32 寫入多個區塊(CMD25)的操作時序 [4]

圖 3-33 為寫入多個磁區資料的副程式，先判別是否需進行磁區位址的轉換，再發送 CMD25 命令並開始輸出第一個磁區資料，然後等待記憶卡返回資料回應，如果成功寫入則繼續輸出下一個磁區資料，最後發出停止傳送資料的回應來結束整個流程。

程式名稱：SD_WriteMultiBlock ( )	輸　入：u32 sector，u8 *buff，u16 num
功能敘述：寫入多個區塊資料	輸　出：r1 回應狀態

```
u8 SD_WriteMultiBlock(u8 *buff, u32 sector,    u8 num)
{  if (SD)Type != SDHC)
       sector == sector << 9;                         // 轉換磁區位址
   SD_Enable ();                                      //記憶卡選擇 CS = LOW
   if (SD_SendCMD (CMD25, sector, 0x00, 1) == 0)      // 寫入多個區塊
   {  do {
          if (!SD_WriteData (data, 0xFC))             // 開始發送區塊資料
              break;
          data += 512;                                // 記憶體位址加 512
      }  while (--count);                             // 區塊數減 1
      if (!SD_WriteData (0, 0xFD))                    // 停止傳送資料
          count = 1;
   }
   SD_Disable ();                                     //記憶卡不選擇 CS = HIGH
   SPI_RWByte (DUMMY);                                // Idle (Release DO)
   return count ? RES_ERROR : RES_OK;
}
```

圖 3-33 SPI 模式寫入多區塊副程式

透過 SPI Bus 寫入資料

圖 3-34 為 SPI 模式的寫入資料流程，當主裝置確認 SD 記憶卡收到命令後，先發送開始傳遞資料(Start block)的資料回應 0xFC，接著使用 SPI 模式的 SPI_RWByte ()副程式來寫入 1 個位元組的資料，再利用迴圈方式來完成 512 筆資料的寫入，與發送 2 位元組的 CRC16 檢查碼，然後等待記憶卡發出資料寫入是否錯誤的回應，當回應為“00101b”表示資料接受成功後，就完成單個區塊資料的寫入動作。

程式名稱：SD_WriteData ( )	**輸　　入：u8 *buffer，u8 Dtoken**
功能敘述：寫入 SPI 區塊資料 (512 bytes)	**輸　　出：r1 回應狀態**

```
u8 SD_WriteData (u8 *buff, u8 Dtoken)
{
    u8 r1, w = 0;
    if (wait_ready () != 0xFF)
        return SD_WFAIL;
    SPI_RWByte (Dtoken);                        // 傳送 data token
    if (token != 0xFD)                          // data token = 0xFD
    {
        for ( w=0; w <512; w ++)                // 傳送 512 次
            SPI_RWByte (*buff++);               // 傳送 1byte 資料
        SPI_RWByte (DUMMY);                     // 傳送 CRC16 (Dummy)
        SPI_RWByte (DUMMY);
        r1 = SPI_RWByte (DUMMY);                // Reveive data response
        if ((resp & 0x1F) != 0x05)              // If not accepted, return with error
            return SD_WFAIL;
    }
    return SD_WOK;
}
```

圖 3-34 SPI 模式傳送資料區塊副程式

3.3 SD Bus 傳輸模式

SD 記憶卡在上電後，設定的工作模式為 SD Bus 傳輸模式，與 SPI Bus 相同，其亦由命令、回應與數據資料所構成 [6]。

命令 (Command)

圖 3-35 為 SD Bus 傳輸模式的命令格式，其與 SPI 模式的用法相同，請參考 3.2.2。

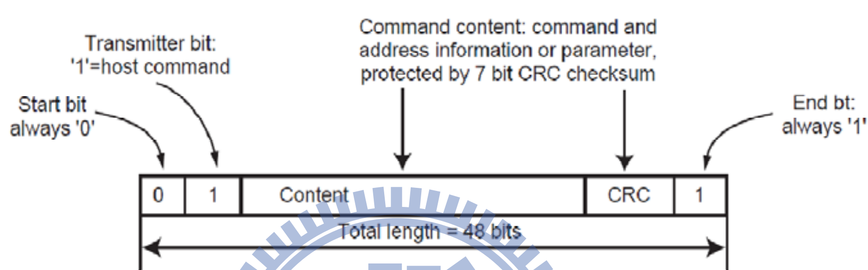


圖 3-35 SD Bus 命令格式 [6]

回應 (Response)

SD Bus 傳輸模式的回應是在 CMD 訊號線上傳輸，圖 3-36 標示出 R1、R2、R3 與 R6 回應的資料格式，其中 R2 回應為 136 位元，其他為 48 位元。

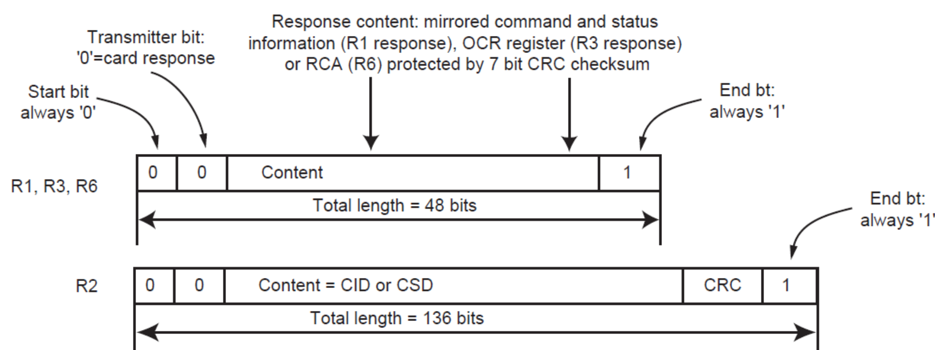


圖 3-36 SD Bus 回應格式 [6]

數據傳輸 (Data)

SD Bus 模式的數據傳遞可以設定為 1 或 4 根資料線來進行傳遞，並且以高位元(MSB)先傳送，圖 3-37 為 1 位元與 4 位元的資料傳輸格式，其中在 4 位元資料線模式，DAT3 會傳送 MSB，而 CRC16 會針對每一根資料線進行保護。

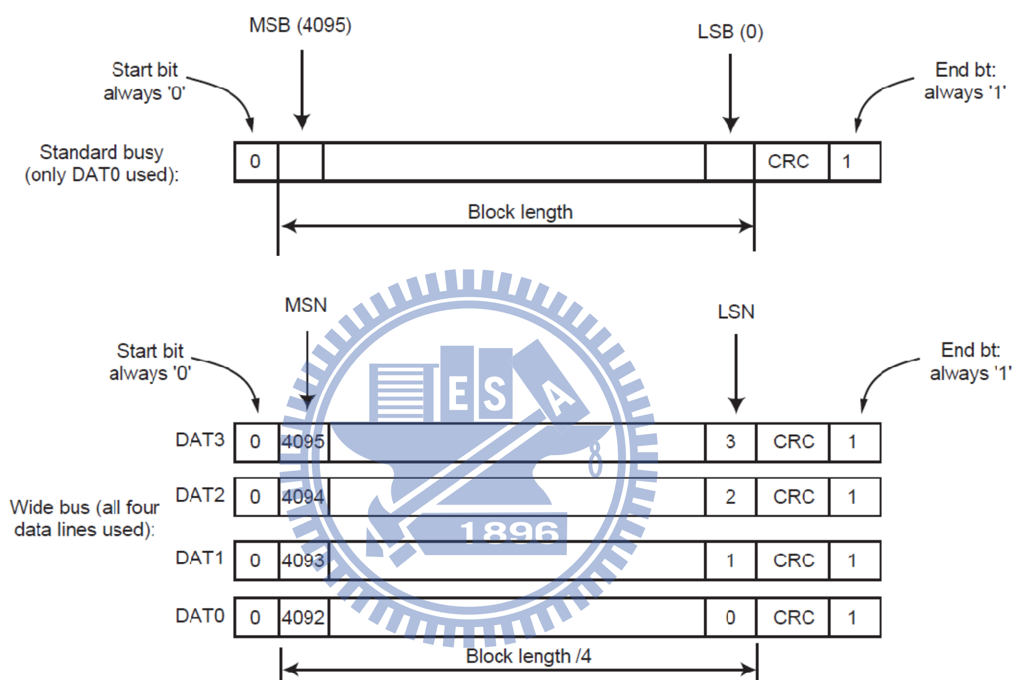


圖 3-37 SD Bus 資料傳輸格式 [6]

3.3.1 資料線模式設定

SD Bus 模式的資料匯流排，允許 1 位元和 4 位元兩種資料線寬度，記憶卡在上電後，預設的資料線寬度為 1 位元(DAT0)，或者主裝置下達命令 CMD0 (GO_IDLE_STATE)後，記憶卡會進入空閒狀態並回到 1 位元模式。主裝置在完成記憶卡的初始化流程後，首先判別是否支援 4 位元模式，如有則下達 CMD7(SELECT/ DIESELECT_CARD)來選中 SD 記憶卡，再用命令 ACMD6(SET_BUS_WIDTH)來切換為寬資料線模式 [6]。

圖 3-38 為設定 4 位元寬資料線模式的程式流程 SD_EnWideBus( )，首先透過 FindSCR (RCA, scr)副程式來選取 RCA 位址的記憶卡，並讀取 SCR 暫存器參數，判斷是否有支援 4 位元寬資料線，如果有支援則送出 ACMD6 命令來設定為 4 位元資料線模式。

程式名稱：SDEnWideBus ( ) 　　輸　入：**FunctionalState NewState**

功能敘述：更改 SD Bus 模式的資料匯流排寬度，1 位元或 4 位元元模式 　　輸　出：錯誤狀態訊

```
static SD_Error SDEnWideBus(FunctionalState NewState)
{
   SD_Error errorstatus = SD_OK;
   u32 scr[2] = {0, 0};
   if (SDIO_GetResponse(SDIO_RESP1) & SD_CARD_LOCKED)
   {
      errorstatus = SD_LOCK_UNLOCK_FAILED;
      return (errorstatus);
   }
   errorstatus = FindSCR (RCA, scr);                    // 讀取 SCR 暫存器
   if (errorstatus != SD_OK)                            // 檢查命令回應
      return(errorstatus);
   if (NewState == ENABLE)                              // 啟動 4 位元匯流排
   {  if ((scr[1] & SD_WIDE_BUS_SUPPORT) != SD_ALLZERO)
```

```
        {  SDIO_CMD (CMD55, (RCA<<16), RESP_S, Enable);              // 發送 CMD55 APP_CMD
           errorstatus = CmdResp1Error(SDIO_APP_CMD);
           if (errorstatus != SD_OK)                                 // 檢查命令回應
               return(errorstatus);
           SDIO_CMD (ACMD6, 0x02, RESP_S, Enable);                   // 發送 ACMD6
           errorstatus = CmdResp1Error(SDIO_APP_SD_SET_BUSWIDTH);
           if (errorstatus != SD_OK)                                 // 檢查命令回應
               return(errorstatus);
        }
        else
        {  errorstatus = SD_REQUEST_NOT_APPLICABLE;
           return (errorstatus);      }
    }
    else                                                             // 啟動 1 位元匯流排
    {  if ((scr[1] & SD_SINGLE_BUS_SUPPORT) != SD_ALLZERO)
        {  SDIO_CMD (CMD55, (RCA<<16), RESP_S, Enable);              // 發送 CMD55 APP_CMD
           errorstatus = CmdResp1Error(SDIO_APP_CMD);
           if (errorstatus != SD_OK)                                 // 檢查命令回應
               return(errorstatus);
           SDIO_CMD (ACMD6, 0x00, RESP_S, Enable);                   // 發送 ACMD6
           errorstatus = CmdResp1Error(SDIO_APP_SD_SET_BUSWIDTH);
           if (errorstatus != SD_OK)                                 // 檢查命令回應
                   return(errorstatus);
        }
        else
        {  errorstatus = SD_REQUEST_NOT_APPLICABLE;
           return (errorstatus);      }
    }
}
```

圖 3-38 SD Bus 模式寬資料傳輸啟動副程式

3.3.2 SD Bus 模式的回應

SD Bus 模式的所有回應都是在 CMD 訊號線上，且都是從高位元的 MSB 開始傳送，回應的資料長度取決於不同類型的回應，SD Bus 模式的回應共有五種類型(R1/R1b、R2、R3、R6 與 R7)，所有類型的回應(除了 R3 以外)都受到 CRC 的保護 [4], [6]。

回應訊號是由開始位元(Start Bit)先傳送，而開始位元永遠為“0”，接著跟隨一個傳輸位元(Transmission Bit)，而傳輸位元也為“0”，然後則是內容資料，最後是 7 位元的 CRC 碼，與結束位元(End Bit)，結束為“1”，圖 3-39 為 SD Bus 模式發送命令後卡片發出回應的時序。

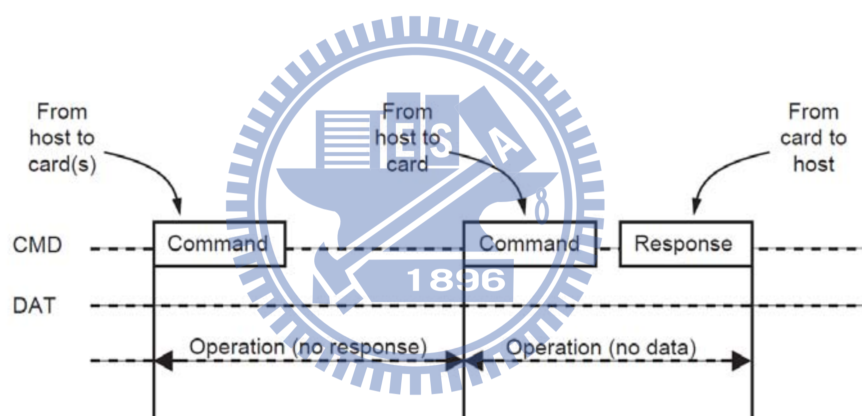


圖 3-39 SD Bus 模式命令與回應時序 [6]

回應 1

主裝置命令發出後，SD 記憶卡會回應 R1，該回應的資料長度為 48 位元，表 3-6 列出了 R1 回應的格式與時序，第 45 至 40 位元表明要回應的命令，第 39 至 8 位元為記憶卡的狀態。但是如果與數據傳輸卡有關時，在傳輸每一個數據塊後，可能會出現一個忙碌的信號，此時主裝置應檢查忙碌的狀態。

表 3-6 SD Bus 模式 R1 回應格式 [4]

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	7
Value	0	0	x	x	x	1
Description	Start bit	Transmission bit	Command index	Card status	CRC7	End bit

CMD

<---- Host command ----> | <-N_{CR} cycles-> | <-------- Response --------->

S	T	content	CRC	E	Z	Z	P	* * *	P	S	T	content	CRC	E	Z	Z	Z

回應 1b

回應 1b 的資料格式與回應 1 相同，但是在數據線上會有一個忙碌的信號在傳送。

回應 2 (回應 CID 與 CSD)

主裝置使用 CMD9 命令讀取 CSD 暫存器與 CMD10 命令讀取 CID 暫存器時，SD 記憶卡發出的特別回應，回應 2 的資料格式如表 3-7，其資料長度為 136 位元，其中[127...1]位元為 SD 記憶卡的 CID 或 CSD 暫存器資料。

表 3-7 SD Bus 模式 R2 回應格式 [4]

Bit position	135	134	[133:128]	[127:1]	0
Width (bits)	1	1	6	127	1
Value	0	0	‘111111’	X	1
Description	Start bit	Transmission bit	Reserved	CID or CSD reguister include internal CEC7	End bit

CMD

<------Host command ----> | <-N_{ID} cycles -> | <--- CID or OCR --->

S	T	content	CRC	E	Z	Z	P	* * *	P	S	T	content	Z	Z	Z

回應 3 (回應 OCR)

回應 3 為 ACMD41 命令的特有回應，主裝置用來讀取 OCR 暫存器資料，其資料長度為 48 位元，表 3-8 為回應 3 的資料格式與時序。

表 3-8 SD Bus 模式 R3 回應格式 [4]

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	7
Value	0	0	‘111111’	x	‘1111111’	1
Description	Start bit	Transmission bit	Command index	OCR register	CRC7	End bit
CMD	<---- Host command ----> | <-N_{CR} cycles-> | <-------- Response ---------> S | T | content | CRC | E | Z | Z | P | * * * | P | S | T | content | CRC | E | Z | Z | Z						

回應 6 (回應 RCA)

記憶卡收到 CMD3 命令所發出的回應，主裝置使用 CMD3 命令讀取 RCA 位址，表 3-9 為回應 6 的資料格式與時序，位元[45:40]為要回應的命令，其固定為“000011b”，位元[39:8]的最高 16 位元則是用來表示該記憶卡的 RCA 位址。

表 3-9 SD Bus 模式 R6 回應格式 [4]

Bit position	47	46	[45:40]	[39:8]		[7:1]	0
Width (bits)	1	1	6	16	16	7	7
Value	0	0	X	X	X	x	1
Description	Start bit	Transmission bit	Command index (000011)	New published RCA[31:16] of the card	[15:0] card status bits	CRC7	End bit
CMD	<---- Host command ----> | <-N_{CR} cycles-> | <-------- Response ---------> S | T | content | CRC | E | Z | Z | P | * * * | P | S | T | content | CRC | E | Z | Z | Z							

回應 7 (回應 Card Interface Condirion)

收到 CMD8 命令(讀取工作電壓資訊)所發出的回應，表 3-10 為其資料格式共有 48 位元，其中位元[19:16]顯示該卡的電壓操作範圍。

表 3-10 SD Bus 模式 R7 回應格式 [4]

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	7
Value	0	0	001000	00000h	X	X	x	1
Description	Start bit	Transmission bit	Command index	Reserved bits	Voltage accepted	Echo-back of check pattern	CRC	End bit
CMD	<---- Host command ----> | S | T | content | CRC | E | Z | Z | P | * * * | P | S | T | content | CRC | E | Z | Z | Z (<-N_{CR} cycles-> between command and response; <-------- Response --------->)								

其中時序圖內容符號所代表的意義為：

S　開始位元(Start bit = 0)

T　傳輸位元(Transmission bit = 0)

P　一個頻率週期(One cycle pull-up = 1)

E　結束位元(End bit = 1)

Z　高阻抗狀態(訊號線準位　= High)

D　資料位元(Data bits)

X　忽略位元(Don’t card data bits = 0)

*　重復(Repeater)

CRC 7 位元 CRC 碼

3.3.3 SD Bus 模式的卡識別模式

圖 3-40 為記憶卡操作在 SD Bus 模式的卡識別流程，其與 SPI 操作模式的卡識別流程相似，請參考 3.2.4 內容。

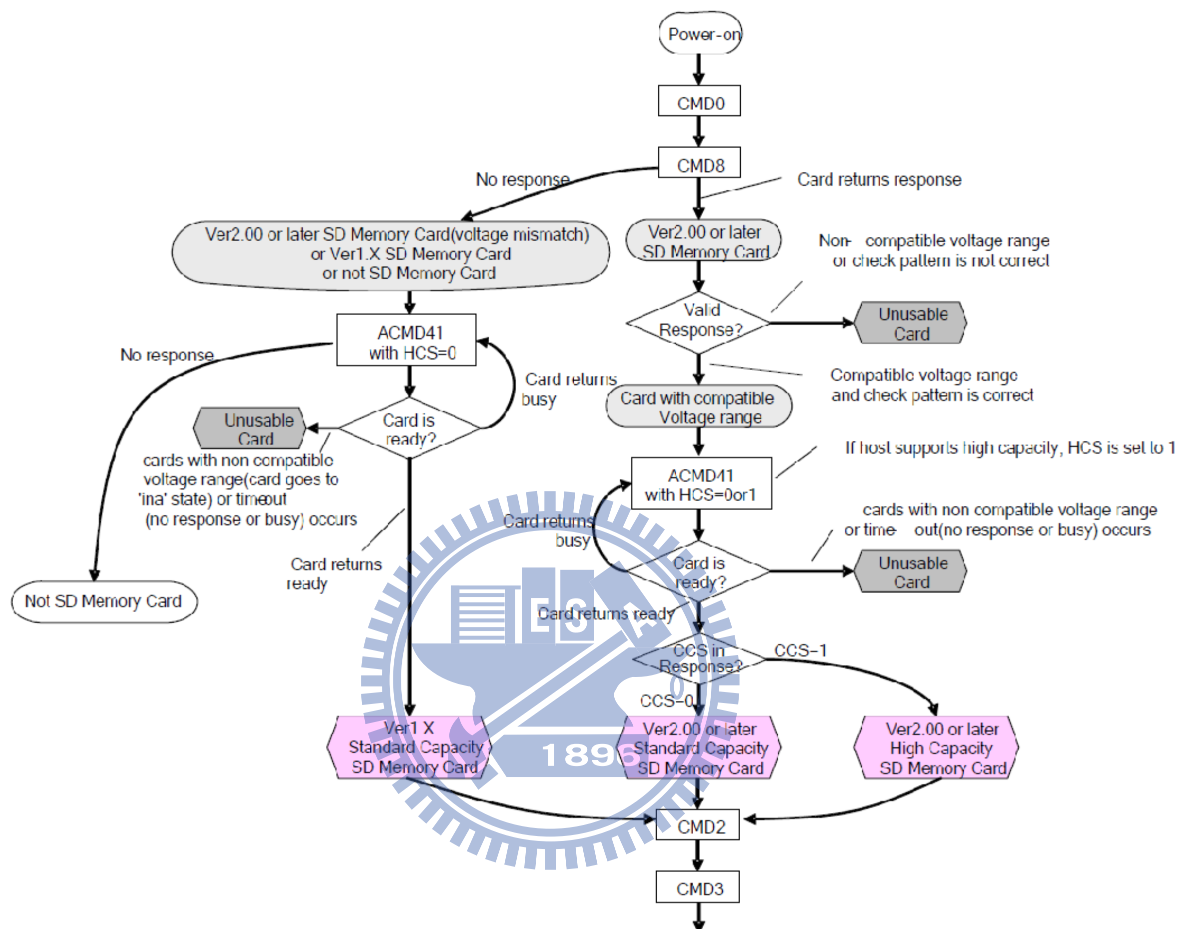


圖 3-40 SD Bus 模式的卡判斷流程 [4]

圖 3-41 為 SD 記憶卡初始化與卡片識別流程 SD_Init ( )，首先設定 SDIO 模組所對應的 GPIO，並設置微控器的 SDIO 模組，再使用圖 3-42 所列的 SD_PowerON ( )副程式來判別記憶卡的種類與版本，接著用圖 3-43 所列的 SD_InitializeCards ( )副程式來讀取 SD 記憶卡的 RCA 與 CID 暫存器值。

由於 SD 記憶卡在初始化的過程中，資料線寬度為 1 位元，且工作頻率不可超過 400KHz，因此當完成記憶卡的初始化與識別流程後，可以自行切換資料線寬度與提高記憶卡的工作頻率到所需要的頻率範圍。

程式名稱：**SD_Init ( )** 　　　輸　入：無

功能敘述：**SD 記憶卡的初始化程式** 　　　輸　出：錯誤狀態訊息

```
SD_Error SD_Init(void)
{   SD_Error errorstatus = SD_OK;
    GPIO_Configuration ();                                   // 設定 SDIO 使用的 GPIO
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_SDIO, ENABLE);  // 開啟 SDIO AHB Clock
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA2, ENABLE); // 開啟 DMA2 Clock
    SDIO_DeInit ();                                          // 初始化 SDIO 模組
    SD_PWRON ();                                             // 供應 SD 卡電源
    errorstatus = SD_PowerON ();                             // 執行 SD 卡判斷流程
    if (errorstatus != SD_OK)          return(errorstatus);
    errorstatus = SD_InitializeCards ();                     // 讀取 SD 卡暫存器資料
    if (errorstatus != SD_OK)          return(errorstatus);
    SDIO_Init_Setup_Value (&SDIO_InitStructure, TRANSFER_CLK_DIV, BusWide_1b, Enable);
    SDIO_Init (&SDIO_InitStructure);                         // 重設 SDIO 模組
    return (errorstatus);
}
```

圖 3-41 SD 記憶卡的初始化程式

程式名稱：**SD_PowerON ( )** 　　　輸　入：無

功能敘述：**SD 記憶卡的種類與版本判斷程式** 　　　輸　出：錯誤狀態訊息

```
SD_Error SD_PowerON(void)
{
    SD_Error errorstatus = SD_OK;
    u32 response = 0, count = 0;
    bool validvoltage = FALSE;
    u32 SDType = SD_STD_CAPACITY;
    SDIO_Init_Setup_Value (&SDIO_InitStructure, INIT_CLK_DIV, BusWide_1b, Disable);
    SDIO_Init(&SDIO_InitStructure);                          // 初始化 SDIO 模組
    SDIO_SetPowerState(SDIO_PowerState_ON);                  // 開啟 SDIO 模組電源
    SDIO_ClockCmd(ENABLE);                                   // 啟動 SDIO 模組頻率
    SDIO_CMD (CMD0, 0x00, RESP_No, Enable);                  // 發送 CMD0
    errorstatus = CmdResp1Error(SDIO_GO_IDLE_STATUS);
    if (errorstatus != SD_OK)                                // 檢查命令回應
        return(errorstatus);
```

```
    SDIO_CMD (CMD8, SD_CHECK_PATTERN, RESP1, Enable);// 發送 CMD8
    errorstatus = CmdResp7Error();                    // 接收 R7 回應
    if (errorstatus == SD_OK)                         // 接收 CMD8 命令
        SDType = SD_STD_CAPACITY                      // 型號為標準 SD
    else                                              // 不接收 CMD8 命令
    {   SDIO_CMD (CMD55, 0x00, RESP_S, Enable);       // 發送 CMD55
        errorstatus = CmdResp1Error(SDIO_APP_CMD);        }
    SDIO_CMD (CMD55, (RCA<<16), RESP_S, Enable);          // 發送 CMD55
    errorstatus = CmdResp1Error(SDIO_APP_CMD);
    if (errorstatus == SD_OK)
    {   while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL))
        {   SDIO_CMD (CMD55, 0x00, RESP_S, Enable);   // 發送 CMD55
            errorstatus = CmdResp1Error(SDIO_APP_CMD);
            if (errorstatus != SD_OK)                 // 檢查命令回應
                return(errorstatus);
            SDIO_CMD (ACMD41, SD_VOLTAGE_WINDOW_SD | 0x40000000, RESP_S, Enable);
            errorstatus = CmdResp3Error();
            if (errorstatus != SD_OK)                 // 檢查命令回應
                return(errorstatus);
            response = SDIO_GetResponse(SDIO_RESP1);
            validvoltage = (bool) (((response >> 31) == 1) ? 1 : 0);
            count++;
        }
        if (count >= SD_MAX_VOLT_TRIAL)
        {       errorstatus = SD_INVALID_VOLTRANGE;
            return (errorstatus);       }
        if (response & 0x40000000)
            SDType = SD_HIGH_CAPACITY;                        // 型號為 SDHC
        if (SDType == SD_HIGH_CAPACITY)
            CardType = SDIO_HIGH_CAPACITY_SD_CARD;            // 型號為 SDHC
        else
            CardType = SDIO_SECURE_DIGITAL_CARD;              // 型號為 MMC Card
    }
    return(errorstatus);
}
```

圖 3-42 SD 記憶卡的種類與版本判斷程式

程式名稱：SD_InitializeCards ( )	輸　入：無
功能敘述：讀取 SD 記憶卡暫存器程式	輸　出：錯誤狀態訊息

```
SD_Error SD_InitializeCards(void)
{  u16 rca = 0x01;       SD_Error errorstatus = SD_OK;
   if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
   {  errorstatus = SD_REQUEST_NOT_APPLICABLE;
      return(errorstatus);      }
   if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
   {  SDIO_CMD (CMD2, 0x00,RESP_L, Enable);                   // 發送 CMD2
      errorstatus = CmdResp2Error();
      if (errorstatus != SD_OK)        return(errorstatus);    // 檢查命令回應
      CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);              // 讀取 CID 暫存器
      CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
      CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
      CID_Tab[6] = SDIO_GetResponse(SDIO_RESP4);        }
   if ((SDIO_SECURE_DIGITAL_CARD == CardType) ||
         (SDIO_SECURE_DIGITAL_IO_CARD == CardType) ||
         (SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType) ||
         (SDIO_HIGH_CAPACITY_SD_CARD == CardType))
   {  SDIO_CMD (CMD3, 0x00,RESP_S, Enable);                   // 發送 CMD3
      errorstatus = CmdResp6Error(SDIO_SET_REL_ADDR, &rca);
      if (errorstatus != SD_OK)        return(errorstatus);    // 檢查命令回應
      if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
      {  RCA = rca;
         SDIO_CMD (CMD9, (rca<<16),RESP_L, Enable);           // 發送 CMD9
         errorstatus = CmdResp2Error();
         if (errorstatus != SD_OK)        return(errorstatus); // 檢查命令回應
         CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);           // 讀取 CSD 暫存器
         CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
         CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
         CSD_Tab[6] = SDIO_GetResponse(SDIO_RESP4);        }
      SD_EnableWideBusOperation(SDIO_BusWide_4b);
      errorstatus = SD_OK; /* All cards get intialized */
      return(errorstatus);
}  }
```

圖 3-43 讀取 SD 記憶卡暫存器程式

3.3.4 SD Bus 模式的資料傳輸

由於 SD 記憶卡在特定時間內只能有一個卡可以為傳輸狀態，因此命令 CMD7 被使用來選擇特定的 SD 記憶卡，使它處於傳輸狀態下。如果上次選擇的 SD 記憶卡還處於傳輸狀態，則其與主裝置的連線會被釋放，且會回到待機狀態。當主裝置發出 RCA 位址為“0x0000”的 CMD7 命令時，所有的 SD 記憶卡都會返回待機狀態 [6]，圖 3-44 為 SD 記憶卡的資料傳輸模式流程圖。

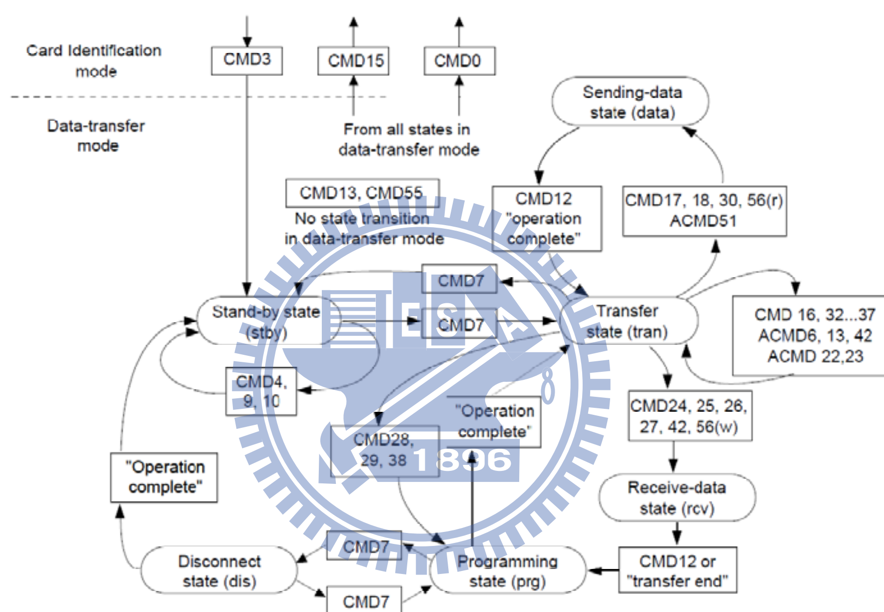


圖 3-44 資料傳輸模式狀態圖 [6]

單區塊資料讀取 (CMD17 與 CMD18)

SD Bus 傳輸模式的讀取資料命令有 CMD17 與 CMD18 兩個，CMD17 READ_SINGLE_BLOCK)為啟動單次讀取數據塊的運作，在傳輸結束後記憶卡回到等待狀態。CMD18 (READ_MULTIPLE_BLOCK)則是啟動多個數據塊的讀取運作。CMD17 與 CMD18 的操作時序如圖 3-45，主裝置在 CMD 訊號線發送命令與接收回應，其中任何時候發送 CMD12 停止傳輸命令即可中止多個數據塊讀取的運作。

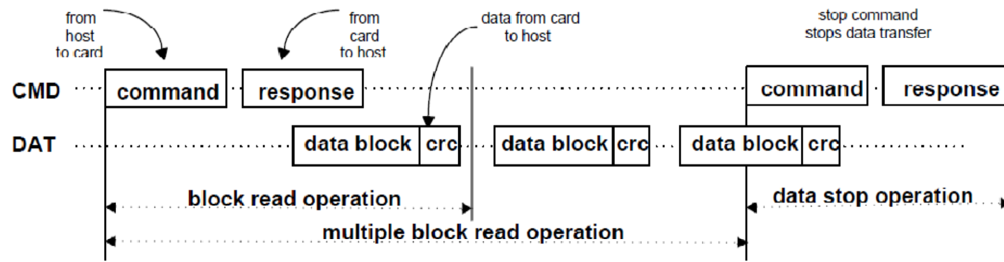


圖 3-45 SD bus 讀取數據塊操作時序 [4]

圖 3-46 為 SD Bus 的單一區塊資料讀取的範例程序，首先清除資料通道狀態機(DPSM)，再使用 CMD16(SET_BLOCKLEN)命令來設定區塊容量，接著設定 SDIO 模組的資料通道狀態機(DPSM)，最後下達 CMD17 (READ SINGLE BLOCK)命令，並接收一個區塊大小的資料。當資料接收完成以後，檢查 SDIO_STA 暫存器的各個旗標，來判斷接收資料的正確性。

程式名稱：**SD_ReadBlock ( )**	輸　入：**u32 addr，u32 *readbuff，u16 BlockSize**
功能敘述：**SD Bus 模式讀取單個區塊資料**	輸　出：錯誤狀態訊息

```
SD_Error SD_ReadBlock(u32 addr, u32 *readbuff, u16 BlockSize)
{   SD_Error errorstatus = SD_OK;
    u32 count = 0, *tempbuff = readbuff;
    u8 power = 0;
    if (NULL == readbuff)
    {   errorstatus = SD_INVALID_PARAMETER;
        return(errorstatus);      }
    SDIO_DPSM_Setup_Value(&SDIO_DataInitStructure, 0x00, 1, ToCard, Disable);  // 清除 DPSM 設定
    SDIO_DataConfig(&SDIO_DataInitStructure);
    SDIO_DMACmd(DISABLE);
    if (SDIO_GetResponse(SDIO_RESP1) & SD_CARD_LOCKED)
    { errorstatus = SD_LOCK_UNLOCK_FAILED;
        return(errorstatus);      }
    if ((BlockSize > 0) && (BlockSize <= 2048) && ((BlockSize & (BlockSize - 1)) == 0))
    {   power = convert_from_bytes_to_power_of_two(BlockSize);
        SDIO_CMD (CMD16, BlockSize, RESP_S, Enable);                // 發送 CMD16
        errorstatus = CmdResp1Error(SDIO_SET_BLOCKLEN);
```

```
    if (errorstatus != SD_OK)        {       return(errorstatus);       }       // 檢查命令回應
  }  else
  {  errorstatus = SD_INVALID_PARAMETER;
     return(errorstatus);       }
  SDIO_DPSM_Setup_Value(&SDIO_DataInitStructure, BlockSize, (power<<4), ToSDIO, Enable);
  SDIO_DataConfig(&SDIO_DataInitStructure);                          // 設定 DPSM
  SDIO_CMD (CMD17, addr, RESP_S, Enable);                            // 發送 CMD17
  errorstatus = CmdResp1Error(SDIO_READ_SINGLE_BLOCK);
  if (errorstatus != SD_OK)        {       return(errorstatus);       }          // 檢查命令回應
  if (DeviceMode == SD_POLLING_MODE)
  {  while (!(SDIO->STA &(SDIO_FLAG_RXOVERR | SDIO_FLAG_DCRCFAIL |
           SDIO_FLAG_DTIMEOUT | SDIO_FLAG_DBCKEND | SDIO_FLAG_STBITERR)))
     {  if (SDIO_GetFlagStatus(SDIO_FLAG_RXFIFOHF) != RESET)
        {  for (count = 0; count < 8; count++)
              *(tempbuff + count) = SDIO_ReadData();
           tempbuff += 8;
     }  }
     if (SDIO_GetFlagStatus(SDIO_FLAG_DTIMEOUT) != RESET)
        return (errorstatus);
     while (SDIO_GetFlagStatus(SDIO_FLAG_RXDAVL) != RESET)
     {  *tempbuff = SDIO_ReadData();
        tempbuff++;       }
     SDIO_ClearFlag(SDIO_STATIC_FLAGS);                          // Clear all the static flags
  }
  else if (DeviceMode == SD_DMA_MODE)
  {  SDIO_ITConfig(SDIO_IT_DCRCFAIL | SDIO_IT_DTIMEOUT | SDIO_IT_DATAEND |
                       SDIO_IT_RXOVERR | SDIO_IT_STBITERR, ENABLE);
     SDIO_DMACmd(ENABLE);
     DMA_RxConfiguration(readbuff, BlockSize);
     while (DMA_GetFlagStatus(DMA2_FLAG_TC4) == RESET)       {}
  }
  return(errorstatus);
}
```

圖 3-46 SD bus 讀取資料磁區副程式

區塊資料寫入 (CMD24 與 CMD25)

SDIO 在寫入資料模式下是由寫資料塊命令(CMD24-27)來執行，圖 3-47 為寫入資料磁區的操作時序，主裝置把一個或多個資料區塊傳送到記憶卡中，同時在每個資料區塊的尾端傳送 16 位元的 CRC 碼，資料區塊的大小一樣由 CMD16 來設定。如果在寫入資料塊時發生 CRC 校驗錯誤，此時記憶卡會通過 SDIO_DAT0 訊號線返回錯誤狀態，而傳送的資料被丟棄不被寫入，所有後續(在多區塊寫入模式)傳送的資料塊將被忽略。

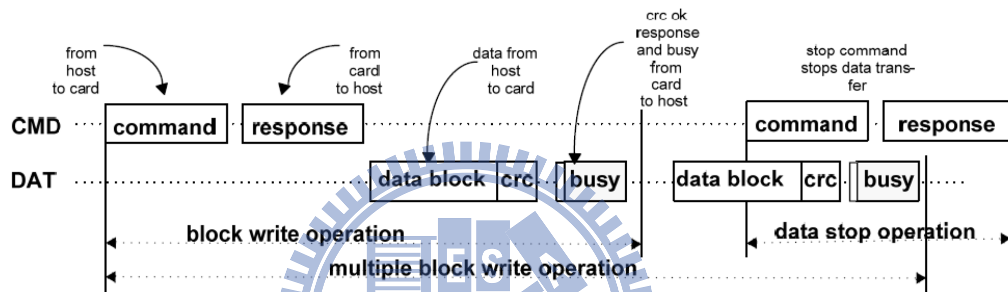


圖 3-47 SD bus 寫入資料磁區操作時序 [4]

圖 3-48 為 SD Bus 的單一區塊資料寫入的範常式，首先清除資料通道狀態機(DPSM)，再使用 CMD16(SET_BLOCKLEN)命令來設定區塊大小，然後用 CMD13(SEND_STATUS)確認記憶卡狀態，接著為 CMD24 (WRITE SINGLE BLOCK)命令，當記憶卡正確回應後，設定資料通道狀態(DPSM)，然後就可傳遞一個區塊大小的資料給記憶卡。當資料傳遞完成以後，檢查 SDIO_STA 暫存器的各個旗標，來判斷傳遞資料的正確性。

程式名稱：SD_WriteBlock ( )	輸　　入：u32 addr，u32 *writebuff，u16 BlockSize
功能敘述：SD Bus 模式寫入單個區塊資料	輸　　出：錯誤狀態訊息

```
SD_Error SD_WriteBlock(u32 addr, u32 *writebuff, u16 BlockSize)
{  SD_Error errorstatus = SD_OK;
   u8   power = 0, cardstate = 0;
```

```
u32 timeout = 0, bytestransferred = 0;
cardstatus = 0, count = 0, restwords = 0; *tempbuff = writebuff;
if (writebuff == NULL)
{   errorstatus = SD_INVALID_PARAMETER;        return(errorstatus);      }
TransferError = SD_OK;          TransferEnd = 0;         TotalNumberOfBytes = 0;
SDIO_DPSM_Setup_Value(&SDIO_DataInitStructure, 0, 1, ToCard, Disable);
SDIO_DataConfig(&SDIO_DataInitStructure);
SDIO_DMACmd(DISABLE);
if (SDIO_GetResponse(SDIO_RESP1) & SD_CARD_LOCKED)
{   errorstatus = SD_LOCK_UNLOCK_FAILED;     return(errorstatus);      }
if (CardType == SDIO_HIGH_CAPACITY_SD_CARD)
{   BlockSize = 512;        addr /= 512;       }
if ((BlockSize > 0) && (BlockSize <= 2048) && ((BlockSize & (BlockSize - 1)) == 0))
{   power = convert_from_bytes_to_power_of_two(BlockSize);
    SDIO_CMD (CMD16, BlockSize, RESP_S, Enable);                  // 發送 CMD16
    errorstatus = CmdResp1Error(SDIO_SET_BLOCKLEN);
    if (errorstatus != SD_OK)        {      return(errorstatus);      }      // 檢查命令回應
}   else
{   errorstatus = SD_INVALID_PARAMETER;     return(errorstatus);      }
timeout = SD_DATATIMEOUT;
do {   timeout--;
    SDIO_CMD (CMD?, (RCA<<16), RESP_S, Enable);                  // 發送 CMD?
    errorstatus = CmdResp1Error(SDIO_SEND_STATUS);
    if (errorstatus != SD_OK)        {      return(errorstatus);      }      // 檢查命令回應
    cardstatus = SDIO_GetResponse(SDIO_RESP1);
}   while (((cardstatus & 0x00000100) == 0) && (timeout > 0));
if (timeout == 0)        {      return(SD_ERROR);       }
SDIO_CMD (CMD24, addr, RESP_S, Enable);                            // 發送 CMD24
errorstatus = CmdResp1Error(SDIO_WRITE_SINGLE_BLOCK);
if (errorstatus != SD_OK)        {      return(errorstatus);      }         // 檢查命令回應
TotalNumberOfBytes = BlockSize;        StopCondition = 0;         SrcBuffer = writebuff;
SDIO_DPSM_Setup_Value(&SDIO_DataInitStructure, BlockSize, (power<<4), ToCard, Enable);
SDIO_DataConfig(&SDIO_DataInitStructure);
if (DeviceMode == SD_POLLING_MODE)
{   while (!(SDIO->STA & (SDIO_FLAG_DBCKEND | SDIO_FLAG_TXUNDERR |
            SDIO_FLAG_DCRCFAIL | SDIO_FLAG_DTIMEOUT | SDIO_FLAG_STBITERR)))
    {   if (SDIO_GetFlagStatus(SDIO_FLAG_TXFIFOHE) != RESET)
        {   if ((TotalNumberOfBytes - bytestransferred) < 32)
```

```
            {   restwords = ((TotalNumberOfBytes - bytestransferred) % 4 == 0) ?
                                ((TotalNumberOfBytes - bytestransferred) / 4) :
                                (( TotalNumberOfBytes -    bytestransferred) / 4 + 1);
                for (count = 0; count < restwords; count++, tempbuff++, bytestransferred += 4)
                    SDIO_WriteData(*tempbuff);
            }   else
            {   for (count = 0; count < 8; count++)             SDIO_WriteData(*(tempbuff + count));
                tempbuff += 8;           bytestransferred += 32;
        }   }   }
        if (SDIO_GetFlagStatus(SDIO_FLAG_DTIMEOUT) != RESET)
        {   errorstatus = SD_DATA_TIMEOUT;          return(errorstatus);        }
    }   else if (DeviceMode == SD_INTERRUPT_MODE)
    {   SDIO_ITConfig(SDIO_IT_DCRCFAIL | SDIO_IT_DTIMEOUT | SDIO_IT_DATAEND |
                SDIO_FLAG_TXFIFOHE | SDIO_IT_TXUNDERR | SDIO_IT_STBITERR, ENABLE);
        while ((TransferEnd == 0) && (TransferError == SD_OK))        {}
        if (TransferError != SD_OK)         {   return(TransferError);      }
    }   else if (DeviceMode == SD_DMA_MODE)
    {   SDIO_ITConfig(SDIO_IT_DCRCFAIL | SDIO_IT_DTIMEOUT | SDIO_IT_DATAEND |
                SDIO_IT_TXUNDERR | SDIO_IT_STBITERR, ENABLE);
        DMA_TxConfiguration(writebuff, BlockSize);
        SDIO_DMACmd(ENABLE);
        while (DMA_GetFlagStatus(DMA2_FLAG_TC4) == RESET)        {}
        while ((TransferEnd == 0) && (TransferError == SD_OK))            {}
        if (TransferError != SD_OK)         {   return(TransferError);      }
    }
    SDIO_ClearFlag(SDIO_STATIC_FLAGS);                  /* Clear all the static flags */
    errorstatus = IsCardProgramming(&cardstate);    /* Wait till the card is in programming state */
    while ((errorstatus == SD_OK) && ((cardstate == SD_CARD_PROGRAMMING) ||
            (cardstate == SD_CARD_RECEIVING)))
    {   errorstatus = IsCardProgramming(&cardstate);      }
    return(errorstatus);
}
```

圖 3-48 SD bus 寫入資料磁區副程式

第四章　FAT 檔案系統

文件分配表檔案系統(簡稱 FAT 系統)是由微軟公司所發展的，主要是用來提供給作業系統使用。早期的 FAT 系統使用 12 位元的定址方式，其最大容量為 32MB，主要應用在軟碟機的檔案系統上。接著在 1984 年，微軟發表了 16 位元定址方式的 FAT 系統，又稱之為 FAT16，其容量可支援到 2GB。最新的 FAT 系統使用 32 位元定址方式，稱之為 FAT32 [7]，表 4-1 為不同位元數的 FAT 系統比較。

16 位元的文件分配表檔案系統是一種廣泛使用的儲存系統，主要應用在中等規模的儲存設備（最大 2GB），如 NAND Flash，多媒體卡（MMC）和 SD 記憶卡等，用以存儲數據與多媒體資料。因此，本章將針對 FAT 文件格式進行說明，以便存取存儲設備的內容。

表 4-1 FAT12/16/32 系統比較 [7]

	FAT12	**FAT16**	**FAT32**
Uses	Floppies and small hard disk volumes	Small to large hard disk volumes	Medium to very large hard disk volumes
Size of each FAT enter	12 bits	16 bits	32 bits (28bits)
Maximum cluster count	4077	65517	268435437
Volume size (maximum)	32 Mbytes	2 GBytes	about 2^41 bytes

本章節將主要介紹 FAT16 系統，其中所使用的 FAT 檔案系統程式，是從 Helix Community 公司(https://helixcommunity.org)所提供的 Helix Player 11 Gold 原始碼進行修改的。

4.1 FAT 系統概述

FAT 文件系統的最小資料儲存空間為叢集(Cluster)，其容量不可超過 32 Kbytes，叢集又由磁區所構成，其具有的磁區數量為 2 的倍數，每個磁區的容量可以為 512、1024，2048 或 4096 bytes，而較常使用的磁區容量為 512 byte，相當於一個叢集最多可以有 64 個磁區 [8]。

在使用 FAT 檔案系統時，當檔案資料小於一個 Cluster 的容量，依然會佔用一個 Cluster，未使用的空間無法給其它檔案使用。如果檔案資料超過一個 Cluster 的容量以上，則會儲存於數個 Cluster 內，而且不用按照 Cluster 順序儲存，其連結順序會紀錄在文件分配表上。

FAT 檔案系統的檔案與目錄資料存放於目錄區(DIR)，其分為短檔名與長檔名兩種格式，本文將針對短檔名進行說明。在短檔名的檔案與目錄資料結構中，每個檔案或目錄會擁有 32 位元組的資料，其記錄該檔案的名稱、屬性、建立日期和時間、資料容量、起始 Cluster 位址等資訊。當我們要讀取檔案目錄資料時，要先從 DIR 區域找到該檔案的檔案與目錄資料，然後得知該檔案的起始 Cluster 位址與資料容量。再由起始 Cluster 位址找到資料所在的磁區位址，即可讀取檔案資料。

由上述可知道要讀取檔案資料會使用到目錄(DIR)、文件分配表(FAT)和資料(DATA)區域，因此系統在執行 FAT 檔案系統時，首要步驟為計算出 DIR、FAT 與 DATA 區域的實體磁區位址(Physical Sector Address)，如此才能正確的進行檔案存取 。

儲存裝置有實體與邏輯兩種磁區位址，實體磁區(PSN)從儲存裝置的第零個磁區開始，邏輯磁區(LSN)則是從磁碟分區的起始磁區開始。

我們以圖 4-1 來解說讀取檔案的流程，假設要讀 AS000D6.WAV 檔案的資料，首先使用 Boot sector 所記錄的參數(RevdSecCnt = 131、NumFAT = 2、FATSz16 = 238、RootEntCnt = 32，見表 4-4)來計算 FAT、DIR 與 DATA 等區域的起始磁區位址(FAT start sector = 132，DIR start sector = 608，First Data Sector = 640)，然後由 DIR 區域讀取 32 bytes 的檔案目錄資訊，並判斷是否為所要的檔案，圖(a)為 DIR 區域的內容，找到該檔案所在的位址後，讀取記錄檔案開始叢集的參數資料(位差 0x1A，FstClus = 0x0002)，利用式 29 可將叢集位址(0x0002)轉換為磁區位址，圖(b)為檔案資料的開始磁區，亦 Cluster 為 2 的資料磁區。當 Cluster 資料讀取完畢後，透過圖(d)的 FAT 表格資料來得到下一個 Cluster 的位址，Cluster 2 位址的數值為 3，表示接下來的資料是存放在 Cluster 3，圖(c)為 Cluster 為 3 的磁區資料。

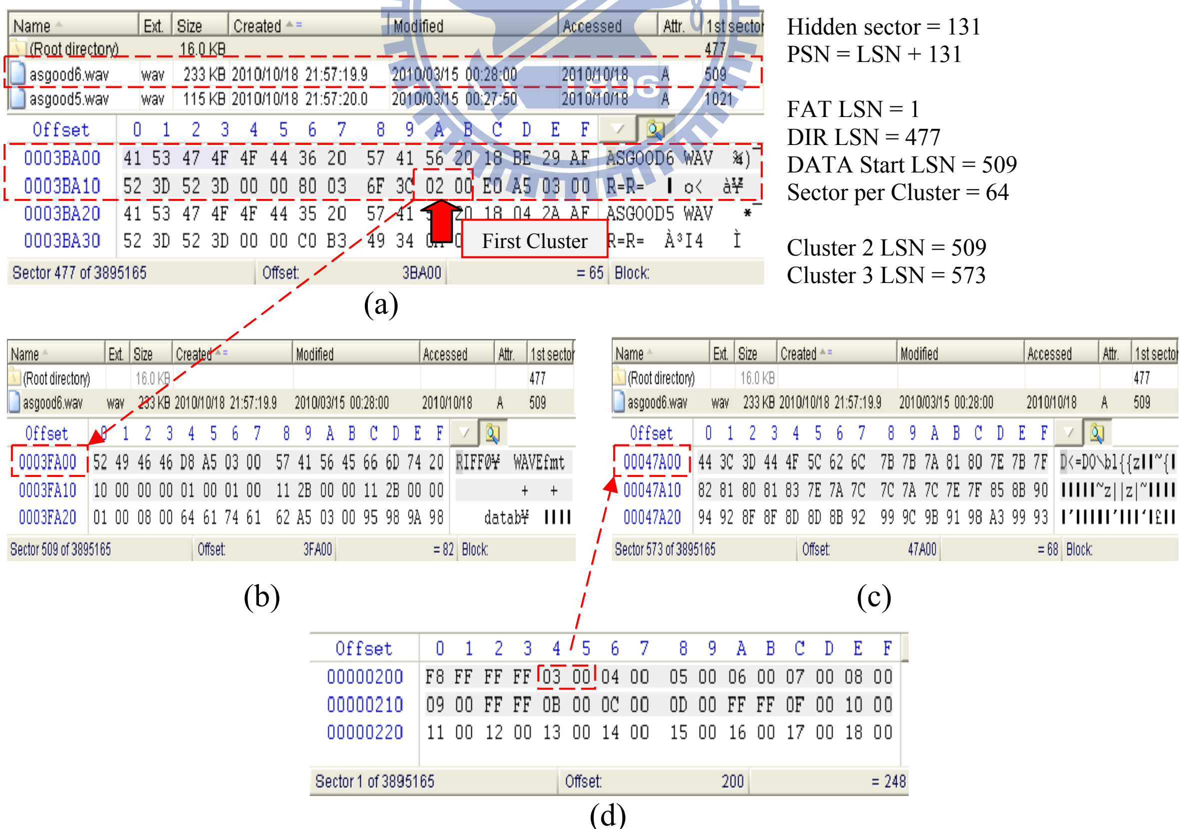


圖 4-1 讀取 FAT 檔案資料範例:(a)DIR 磁區資料;(b)Cluste 2 磁區資料;(c)Cluste 3 磁區資料;(d)FAT 磁區資料

FAT 檔案系統的磁區結構包括四個不同的部份，保留區域(Reserved Region)、文件分配表區域(FAT Region)、根目錄區域(Root Directory Region)與資料區域(Data Region)。圖 4-2 為本系統 2GB SD 記憶卡的各個區域的起始磁區位址與磁區容量配置。

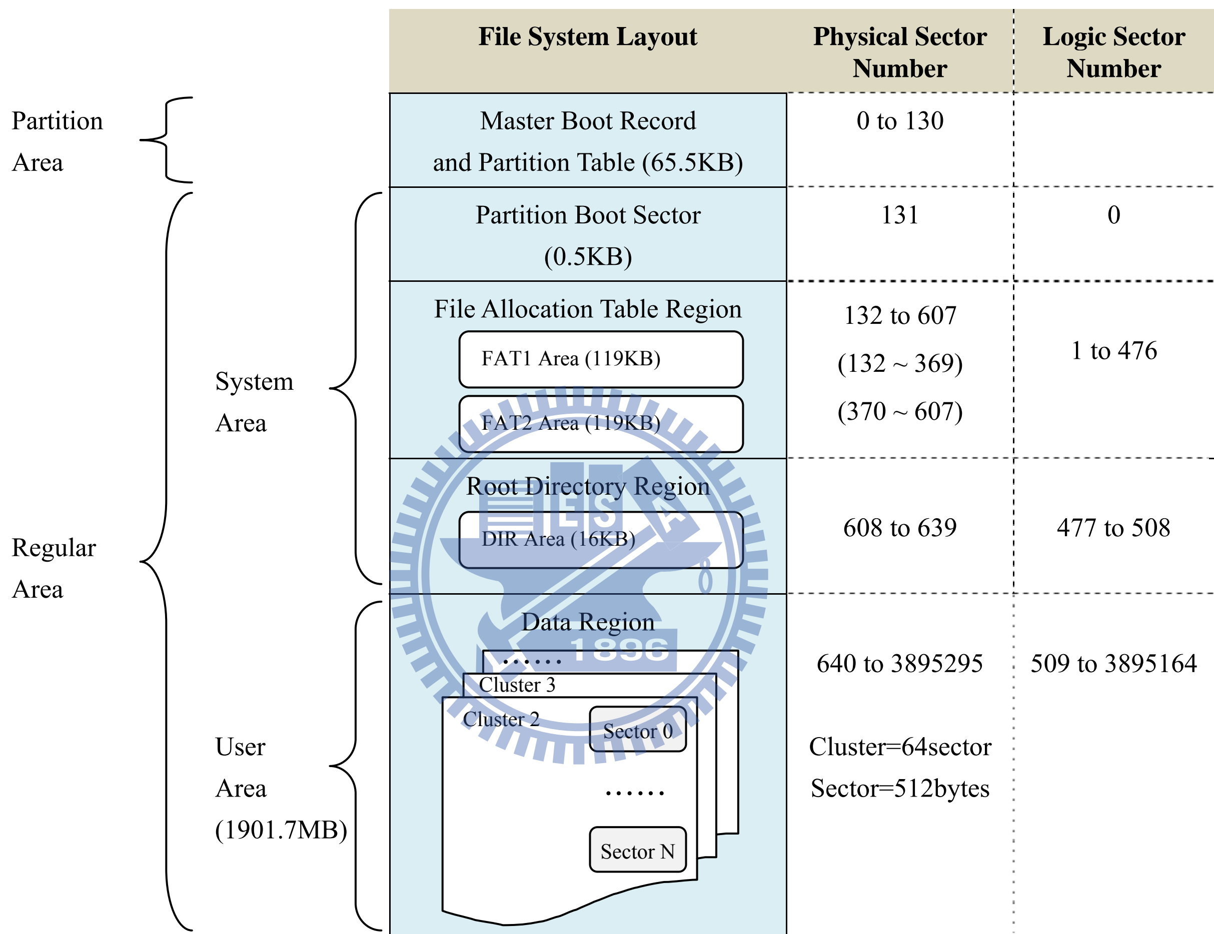


圖 4-2 FAT16 系統資料結構 [9]

4.2 FAT 的保留區域

FAT 保留區域包括了 MBR 開機磁區(Master Boot Record)與 BPB 啟動參數區(BIOS Parameter Block)，其中 MBR 一定位於儲存裝置的第零磁區，並且包含磁碟分區表(Disk Partition Table)的資訊。而 BPB 位於磁碟分區的第一個磁區，每個磁碟分區都會有其專屬的 BPB 資料 [7]。

圖 4-3 為 FAT 開機磁區的資料型態，圖(a)為 MBR 無磁碟分區表，BPB 位於 MBR 內，(b)為 MBR 有磁碟分區表資訊，BPB 則位於磁碟分區表記錄的開始磁區內。

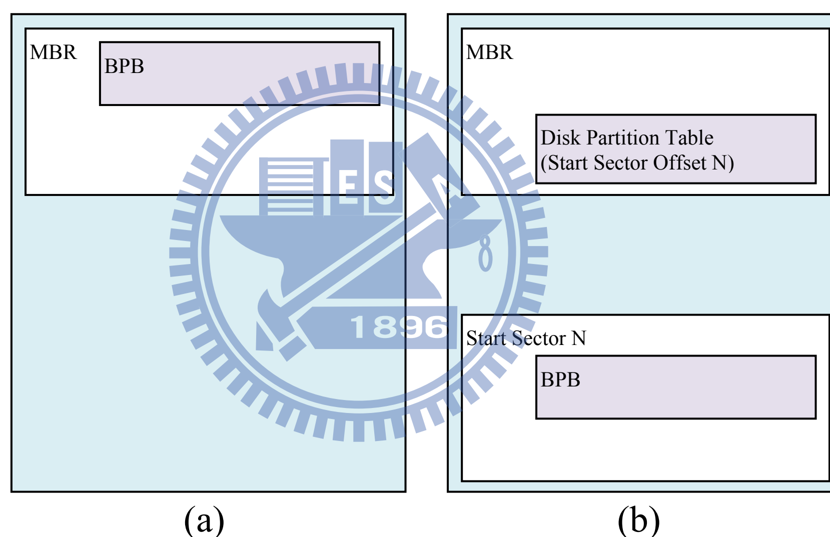


圖 4-3 FAT 開機磁區的資料型態:(a)無磁碟分區表;(b)有磁碟分區表

底下將針對開機磁區（MBR)、磁碟分區表(DPT)與系統引導記錄區（BPB)進行說明。

4.2.1 開機磁區

開機磁區(MBR)位於儲存裝置的第零磁區，MBR 的主要目的是提供開機程序給操作系統使用，MBR 的資料容量為 512 bytes，其中包括磁碟分區表資訊。表 4-2 為 MBR 的結構，前 446 bytes 為啟動電腦的可執行程式碼，接著為 4 組磁碟分區表，其共有 64 bytes 的資料，最後為 2 bytes 的分區結束標誌"0x55 0xAA" [7]。

表 4-2 開機磁區結構 [7]

Offset	Description	Size
000h	Executable Code (Boots Computer)	446 Bytes
1BEh	1st Partition Entry	16 Bytes
1CEh	2st Partition Entry	16 Bytes
1DEh	3st Partition Entry	16 Bytes
1EEh	4st Partition Entry	16 Bytes
1FEh	Executable Marker (55h AAh)	2 Bytes

1896

儲存裝置的開機磁區(MBR)可以沒有磁碟分區表，當沒有磁碟分區表時，BPB 位於 MBR 所在的磁區內；當有磁碟分區表時，MBR 所在的磁區資料不會包括 BPB 資訊，而 BPB 資訊則會放置於每個磁碟分區的起始磁區內。

圖 4-4 為本系統使用不同 SD 記憶卡的 MBR 資訊，圖(a)為無磁碟分區的 MBR 資訊，BPB 資訊位於 MBR 的起始位址。(b)擁有一個磁碟分區，因此 MBR 內不會有 BPB 資訊，必須要透過磁碟分區的起始磁區位址，圖中的起始磁區位址為 0x83 (131)，才能得到 BPB 資訊。

(a)

0	EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	20	04	00
16	02	00	02	00	00	F8	F2	00	3F	00	FF	00	00	00	00	00
32	00	28	1E	00	00	00	29	1A	56	20	8C	4E	4F	20	4E	41
48	4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9
64	8E	D1	BC	F0	7B	8F	D9	B8	00	20	8E	C0	FC	BD	00	7C
80	38	4E	24	7D	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	72	1C	83	EB	3A
96	66	A1	1C	7C	26	66	3B	07	26	8A	57	FC	75	06	80	CA
112	02	88	56	02	80	C3	10	73	EB	33	C9	8A	46	10	98	F7
128	66	16	03	46	1C	13	56	1E	03	46	0E	13	D1	8B	76	11
144	60	89	46	FC	89	56	FE	B8	20	00	F7	E6	8B	5E	0B	03
160	C3	48	F7	F3	01	46	FC	11	4E	FE	61	BF	00	00	E8	E6
176	00	72	39	26	38	2D	74	17	60	B1	0B	BE	A1	7D	F3	A6
192	61	74	32	4E	74	09	83	C7	20	3B	FB	72	E6	EB	DC	A0
208	FB	7D	B4	7D	8B	F0	AC	98	40	74	0C	48	74	13	B4	0E
224	BB	07	00	CD	10	EB	EF	A0	FD	7D	EB	E6	A0	FC	7D	EB
240	E1	CD	16	CD	19	26	8B	55	1A	52	B0	01	BB	00	00	E8
256	3B	00	72	E8	5B	8A	56	24	BE	0B	7C	8B	FC	C7	46	F0
272	3D	7D	C7	46	F4	29	7D	8C	D9	89	4E	F2	89	4E	F6	C6
288	06	96	7D	CB	EA	03	00	00	20	0F	B6	C8	66	8B	46	F8
304	66	03	46	1C	66	8B	D0	66	C1	EA	10	EB	5E	0F	B6	C8
320	4A	4A	8A	46	0D	32	E4	F7	E2	03	46	FC	13	56	FE	EB
336	4A	52	50	06	53	6A	01	6A	10	91	8B	46	18	96	92	33
352	D2	F7	F6	91	F7	F6	42	87	CA	F7	76	1A	8A	F2	8A	E8
368	C0	CC	02	0A	CC	B8	01	02	80	7E	02	0E	75	04	B4	42
384	8B	F4	8A	56	24	CD	13	61	61	72	0B	40	75	01	42	03
400	5E	0B	49	75	06	F8	C3	41	BB	00	00	60	66	6A	00	EB
416	B0	4E	54	4C	44	52	20	20	20	20	20	20	0D	0A	52	65
432	6D	6F	76	65	20	64	69	73	6B	73	20	6F	72	20	6F	74
448	68	65	72	20	6D	65	64	69	61	2E	FF	0D	0A	44	69	73
464	6B	20	65	72	72	6F	72	FF	0D	0A	50	72	65	73	73	20
480	61	6E	79	20	6B	65	79	20	74	6F	20	72	65	73	74	61
496	72	74	0D	0A	00	00	00	00	00	00	00	AC	CB	D8	55	AA

(b)

0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
32	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
64	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
96	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
112	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
128	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
144	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
176	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
192	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
208	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
224	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
256	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
272	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
288	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
304	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
336	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
352	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
368	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
384	00	00	00	00	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00	00	00	00
400	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
416	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
432	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02
448	06	00	06	06	C6	C6	83	00	00	00	7D	6F	3B	00	00	00
464	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
480	00	00	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00	00	00	00	00	00
496	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA

圖 4-4 開機磁區資料:(a)無磁碟分區的 Boot sector;(b)有磁碟分區的 MBR 資訊

4.2.2 磁碟分區表

磁碟分區表(Disk Partition Table)位於 MBR 的 0x1BE 到 0x1FD 共有 64 bytes，其為記錄每個磁碟分區的參數，一個儲存裝置可分割為四個磁碟區，每一磁碟區對應 16 bytes 的磁碟分區表參數，而記憶卡一般只會有一個磁碟區。表 4-3 為 FAT 系統的磁碟分區資料結構，其中記錄了該分區的起始磁區位址與容量大小等資訊，而 BPB 資訊即存放在該分區的起始磁區內，在非磁碟裝置的儲存系統中沒有磁頭(Head)與柱面(Cylinder)的資訊，如 MMC 與 SD 記憶卡等 [7]。

表 4-3 FAT 磁碟分區結構 [7]

Offset	Description	Size
00h	Current State of the Partition	1 Byte
01h	Beginning of the Partition - Head	1 Byte
02h	Beginning of the Partition – Cylinder/Sector	2 Bytes
04h	Type of Partition	1 Byte
05h	End of Partition - Head	1 Byte
06h	End of Partition – Cylinder/Sector	2 Bytes
08h	# of sectors between MBR and Partition	4 Bytes
0Ch	# of Sectors in the Partition	4 Bytes

4.2.3 啟動參數區

每一磁碟分區都會有其專屬的啟動參數區(BPB)，其記錄了該磁碟分區的規格參數，包含了 FAT 檔案系統的型式、磁區容量、叢集容量、FAT 表的數目與容量、根目錄的數量、總磁區數等資訊，詳細的 BPB 資料結構請參考附錄三 [8]。

本系統使用的 FAT 檔案系統其尋找 BPB 磁區的程式流程如圖 4-5，先將 Sector 0 的 MBR 資料導入緩衝區內，並判斷是否為合法的 MBR 資料區，再檢查位址 0x36 是否有 FAT 檔案系統的類型標籤，如 FAT12、FAT16 或 FAT32，用來判斷 BPB 資訊是否存在，若無 FAT 檔案系統類型標籤，則檢查第一磁碟分區的 0x1BE 位址(見表 4-2)是否有磁碟分區的存在，如無磁碟分區資料的存在，則表示此記憶卡尚未格式化，如有磁碟分區資料的存在，則檢查位址 0x1C6 (0x1BE + 0x08，見表 4-3)的磁區位址位差量，並設定該位差位址為 BPB 磁區位址並將其導入。

程式名稱：Load_BPB ( ) 　　**輸　入：無**
功能敘述：載入 FAT 系統的 BPB 磁區 　　**輸　出：BPB**

```
u8 Load_BPB (void)
{  FATFS *fs = FatFs;
   u32 set = 0;
   if (disk_read (fs->buf, sect, 1) == RES_OK)              // Load master boot record
   {
      if (LD_WORD(&(fs->buf[510])) == 0xAA55)              // Is it valid?
      {
         if (!memcmp(&(fs->buf[0x36]),"FAT12", 5))
            return FS_FAT12;
         if (!memcmp(&(fs->buf[0x36]), "FAT16", 5))
            return FS_FAT16;
         if (!memcmp(&(fs->buf[0x52]), "FAT32", 5) && (fs->buf[0x28] == 0))
            return FS_FAT32;
      }
   }
   if (fs->buf[0x1C2])                                     // Is the partition existing?
```

```
    {
        sect = LD_DWORD (&(fs->buf[0x1C6]));                    // Partition offset in LBA
        if (disk_read (fs->buf, sect, 1) == RES_OK)              // Load master boot record
        {
            if (LD_WORD(&(fs->buf[510])) == 0xAA55)               // Is it valid?
            {
                if (!memcmp(&(fs->buf[0x36]),"FAT12", 5))
                    return FS_FAT12;
                if (!memcmp(&(fs->buf[0x36]), "FAT16", 5))
                    return FS_FAT16;
                if (!memcmp(&(fs->buf[0x52]), "FAT32", 5) && (fs->buf[0x28] == 0))
                    return FS_FAT32;
            }
        }
    }
    return Not_FS;
}
```

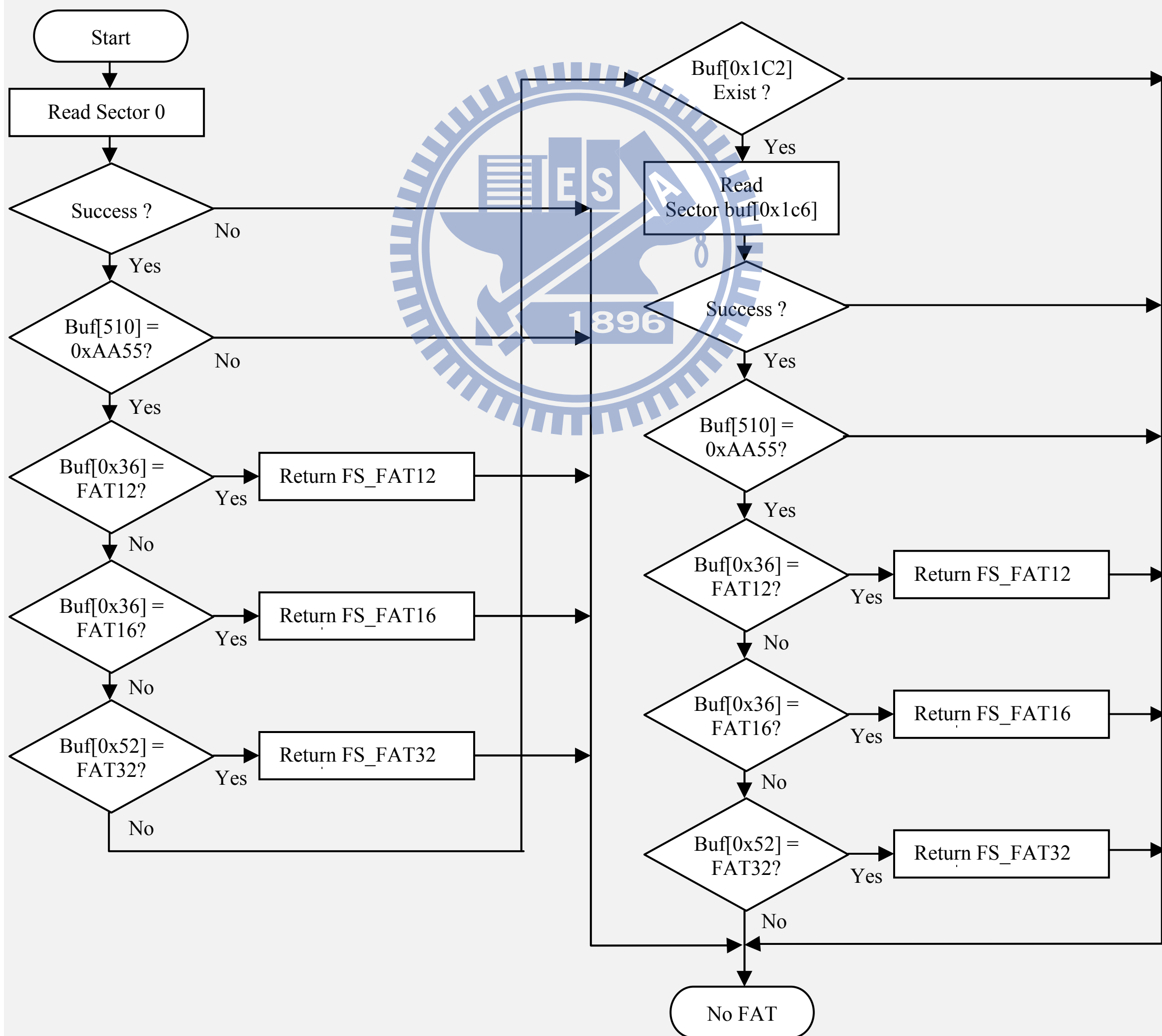


圖 4-5 FAT 載入 BPB 資訊副程式

系統找到 BPB 資訊所在的磁區後，須將其解碼以得到 FAT 檔案系統的型式，如 FAT16 或 FAT32，與磁區結構參數，如 FAT size、BytePerSector、SectorPerCluster 等資訊(見附錄三)。並運算出 FAT 檔案系統的 FAT、DIR 與 DATA 區域的起始磁區位址。其中 FAT Base Sector 為文件分配表的起始磁區位址，用來作 FAT 查表的參考位址；DIR Base Sector 為文件目錄的起始磁區位址，即根目錄磁區位址，為文件目錄的存放區域；DATA Base Sector 做為資料讀取與寫入的起始參考位址，亦為叢集 2 的起始磁區位址，請注意根據規範叢集 0 與叢集 1 是虛擬的，不佔用記憶體空間。[8]。

啟動磁區(MBR)與 BPB 磁區位址計算(參考附錄三)

$$N_{MBR} = 0 \quad (22)$$

BPB 磁區位址計算(參考附錄三)

1. 無磁碟分割區，則 BPB 起始磁區為：

$$N_{BPB} = 0 \quad (23)$$

2. 有磁碟分割區，則 BPB 起始磁區為：

$$N_{BPB} = N_{part} \quad (24)$$

其中 N_{MBR} 為 MBR 起始磁區，N_{BPB} 為 BPB 起始磁區，N_{part} 為磁碟分區的起始磁區。

文件分配表(FAT)起始磁區位址計算(參考附錄三)

$$N_{FAT1} = N_{MBR} + N_{res} \quad (25)$$

$$N_{FAT_Tsize} = N_{FAT_size} \times N_{num} \quad (26)$$

其中 N_{FAT1} 為 FAT1 的起始磁區，N_{res} 為保留磁區數，N_{FAT_size} 為每個 FAT 的磁區數，N_{FAT_Tsize} 為 FAT 的總磁區數量，N_{num} 為 FAT 的數目。

根目錄(Root directory)起始磁區位址計算(參考附錄三)

$$N_{DIR} = N_{FAT1} + N_{FAT_Tsize} \quad (27)$$

$$N_{DIR_size} = (N_{RootEntCnt} \times 32)/N_{BytePerSec} \quad (28)$$

其中 N_{DIR} 為 DIR 的起始磁區；N_{DIR_size} 為 DIR 的總磁區數量；$N_{RootEntCnt}$ 為 DIR 最大使用數量；$N_{BytePerSect}$ 為一個磁區的位元組數量。

資料(Data)起始磁區位址計算(參考附錄三)

$$N_{DATA} = N_{MBR} + N_{FAT_Tsize} + N_{DIR_size} \quad (29)$$

其中 N_{DATA} 為資料區的起始磁區。

本系統導入與解碼 BPB 資訊，以及運算磁區結構參數的程式碼可參考圖 4-6 的 fs_mountdrive ()副程式與流程，其主要功能為執行初始化記憶卡程序(disk_initialize)，並導入 BPB 資訊(Load_BPB)並判別 FAT 型式，再計算 FAT、DIR 與 DATA 區域的起始位址參數。

程式名稱：fs_mounrdrive ( ) **輸　入：無**

功能敘述：判別 SD 卡，確認 FAT 系統存在，載入 FATFS 結構 **輸　出：計算 FATFS 位址參數**

```
u8 fs_mountdrive (void)
{   u8 fs_rsp;
    u32 sector, end, max;
    FATFS *fs = FatFs;
    if (disk_initialize() & SD_NOINIT)                // 檢查記憶卡是否有初始化
        return FAT_NOT_READY;
    if (!(fs_rsp = Load_BPB())                        // 尋找 BPB 磁區位址
        return FAT_NO_FILESYSTEM;
    fs->fs_type = fs_rsp;                             // 設定 FAT 型式
    fs->sects_fat    = (fs_rsp==FAT_32) ? LD_DWORD(&(fs->buf[0x24])) : LD_WORD(&(fs->buf[0x16]));
    fs->sects_clust  = fs->buf[0x0D];                 // 設定每個叢集的磁區數
    fs->n_fats       = fs->buf[0x10];                 // 設定 FAT 的數目
    fs->fatbase      = sect + LD_WORD(&(fs->buf[0x0E]));   // 計算 FAT 開始磁區位址
    end = fs->sects_fat * fs->n_fats + fs->fatbase;
    fs->n_rootdir    = LD_WORD(&(fs->buf[0x11]));     // 設定 DIR 的最大數量
```

```
    if (fs_rsp == FAT_32)
    {   fs->dirbase     = LD_DWORD(&(fs->buf[0x2C]));           // FAT32: Directory start cluster
        fs->database    = fatend;                               // FAT32: Data start sector (physical)
    }   else
    {   fs->dirbase     = fatend;                               // Directory start sector (physical)
        fs->database    = fs->n_rootdir / 16 + fatend;          // Data start sector (physical)
    }
    max = LD_DWORD(&(fs->buf[0x20]));                           // Calculate maximum cluster number
    if (!max)           max = LD_WORD(&(fs->buf[0x13]));
    fs->max_clust       = (max - fs->database + sect) / fs->sects_clust + 2;
    return FAT_OK;
}
```

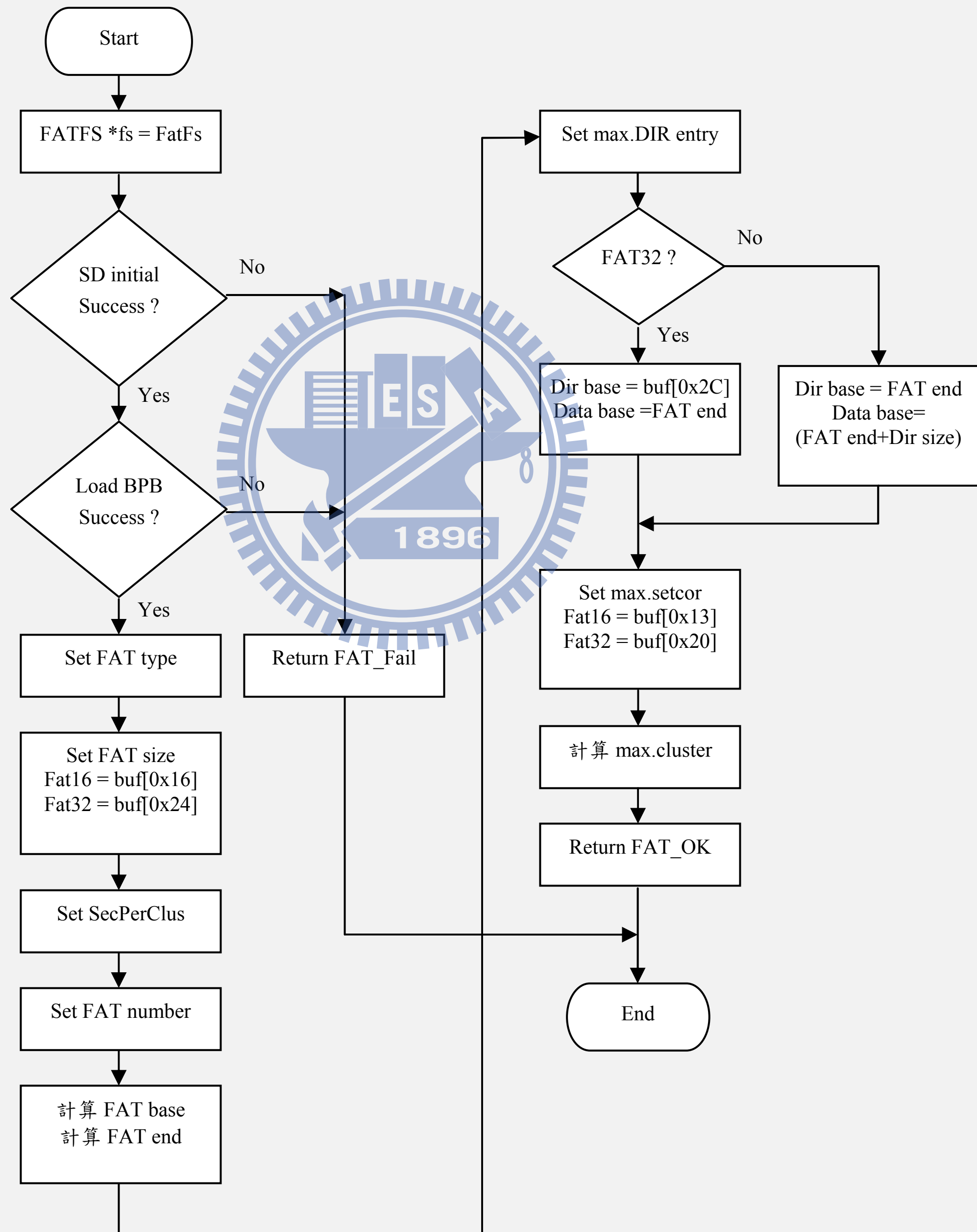


圖 4-6 導入 FAT 檔案系統副程式

4.3 文件分配表區域

文件分配表區域(簡稱 FAT)是儲存檔案分割資訊的對映表，標示檔案的資料是儲存在那個叢集位址裡面。FAT 會有一個以上的備份區域，這是由於系統安全性的考慮，當 FAT1 發生錯誤時，可以由 FAT2 得到正確的叢集位址，而不會發生無法讀取資料的錯誤。

FAT 是一種文件索引與定位所使用的鏈式儲存結構，資料儲存於磁碟裝置的基本容量是叢集。每個文件根據它的資料容量可能會使用一個或者多個叢集空間，而且不需要依照順序儲存，其儲存資料的順序資訊會存放在 FAT 表中。FAT 主要是用來記錄檔案是否有使用到另一個叢集空間，FAT 表的偏移位址為即目前的叢集指標，例如 Offset address = 0x02 即第二個叢集，而偏移位址的內容為下一個叢集位址或是結束符號。表 4-4 為 FAT16 的文件分配表結構與內容描述。

表 4-4 FAT16 文件分配表的結構 [8]

Table	FAT entry	Description	
0	FFF8	Table 0 must be FFF8h	
1	FFFF	Table 1 must be FFFFh	
2	0003	0000h	空閒叢集
3	0007	0001h	保留叢集
….	….	0002h – FFFEh	被佔用的叢集，指向下一個叢集
N	FFFF		
N+1	0000	FFF0h – FFF6h	保留值
….	0000	FFF7h	壞的叢集
M	FFF7	FFF8h - FFFFh	文件的最後一個叢集
….	….		

圖 4-7 為檔案文件儲存在 FAT16 中的叢集鏈結構，在 DIR 目錄中可以讀取到檔案名稱與開始的叢集位址，例如 X.TXT 檔案的開始叢集位址為 0x0002，由 FAT 表可以得到其叢集鏈從 0x0002 開始，接著為 0x0003，最後在 0x0004 結束，共使用 3 個叢集空間。

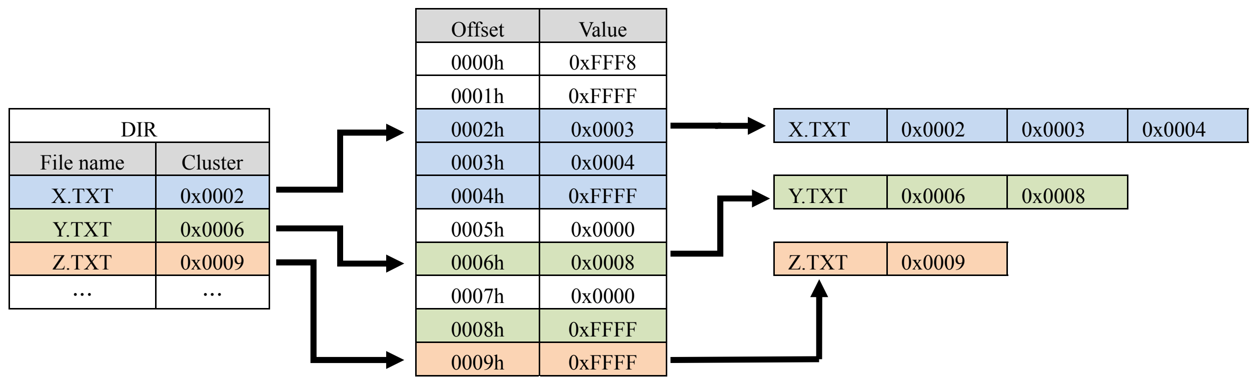


圖 4-7 FAT16 的檔案叢集鏈 [10]

FAT 的儲存結構會造成文件的資料有可能不會存放在連續的區域內，其往往會零散的儲存在磁碟，圖 4-8 為尋找下一個叢集號碼的 get_cluster () 副程式，其流程為先檢查叢集號碼是否在範圍內，再找該叢集號碼所在的 FAT 磁區位址(30)與偏移位址(31)，才能得到下一個叢集號碼的資訊。

$$N_{sec} = N_{FAT_Base} + (N_{clus}/N_x) \quad (30)$$

$$N_{offset} = N_{clus} \bmod 0x200 \quad (31)$$

其中 N_{sec} 為叢集指標所在的磁區位址，N_{FAT_Base} 為 FAT 啟始位址，N_x 為一個磁區所擁有的叢集數目，FAT16 為 256 個，FAT32 為 128 個。

程式名稱：get_cluster ( )	**輸　入：目前的叢集號碼**
功能敘述：讀取下一個叢集號碼	**輸　出：下一個叢集號碼**

```
u32 get_cluster (u32 cluster)
{   u16 wc, bc;
    u32 fatsector;
    FATFS *fs = FatFs;
    if ((cluster >= 2) && (cluster < fs->max_clust))             // 叢集號碼是否有效
    {   fatsector = fs->fatbase;                                  // FAT 表起始位址
        if (!next_block(fatsector + cluster / 256))     break;    // 尋找輸入叢集號碼的 FAT 位址
        return LD_WORD(&(fs->win[((u16)cluster * 2) % 512]));     // 返回下一個叢集號碼
    }
    return 1;
}
```

圖 4-8 尋找下一個叢集號碼副程式

4.4 目錄區域

目錄區域(簡稱 DIR)緊接在 FAT 區域之後，是儲存檔案與目錄名稱資訊的區域，每一個檔案或目錄都有 32 bytes 資訊，裡面記錄了檔案或目錄的名稱、副檔名、屬性、建立的日期和時間、檔案目錄資料起始叢集位址，檔案目錄的容量等資訊。系統在執行檔案處理時會根據 DIR 記錄的起始叢集位址，結合 FAT 表就可以知道檔案在儲存裝置中的實體位址和使用的空間，檔案與目錄區域的 32 Bytes 資料結構請參考附錄三 [8]。

圖 4-9 為本系統使用的 ASG00D6.WAV 檔案資料，圖(a)為檔案在目錄區的資訊，可以得道檔案起始叢集位址為 0x0002，檔案容量為 233.5KB。經由(30)可得到圖(c)的起始磁區位址與資料，再透過圖(b)FAT 表得到下一個叢集位址為 0x0003 後，一樣經由(30)可得到圖(d)的資料。

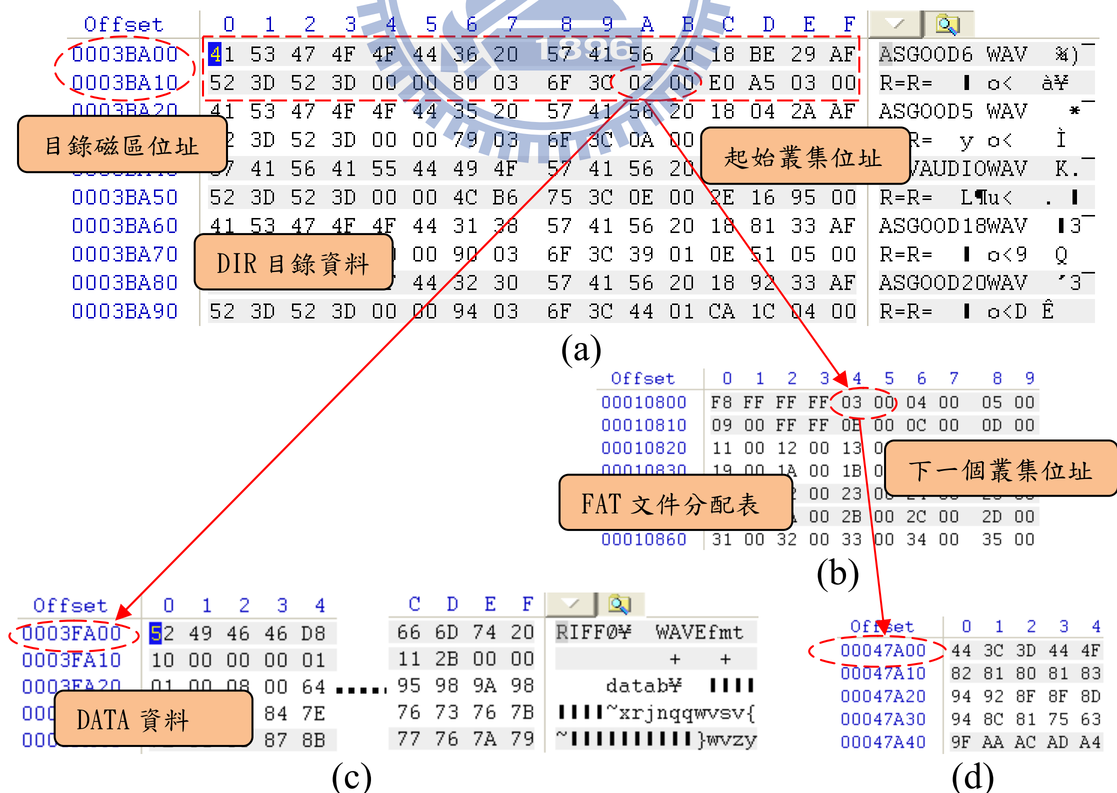


圖 4-9 根目錄區域磁區資料

FAT 檔案系統的檔案與目錄名稱分為短檔名與長檔名兩種格式，針對短檔名的檔案與目錄資料結構，其有一些限制與特殊應用，所使用的字符也有特別的限制與用法。底下將進行說明 [8]。

(1) 如果 DIR_NAME[0]為 0xE5，表示此目錄為空。

(2) 如果 DIR_NAME[0]為 0x00，表示此目錄為空(如同 0xE5)，而且此位址之後不再有目錄資料。

(3) 如果 DIR_NAME[0]為 0x05，因為 0xE5 是日文的合法符號，所以當需要使用 0xE5 時，會用 0x05 來代替。

(4) DIR_NAME[0]不允許使用 0x20 空白符號當第一個字節。

(5) DIR_NAME 由兩個部分組成，8 個位元組的主檔名和 3 個位元組的副檔名，如果字數不夠時，由 0x20 空白符號來填滿。

(6) 不允許出現的字符有下列數個，0x22，0x2A，0x2B，0x2C，0x2E，0x2F，0x3A，0x3B，0x3C，0x3D，0x3E，0x3F，0x5B，0x5C，0x5D，0x7C。

(7) 檔案與擴展名稱之間的點符號“‧”並不存在於 DIR_NAME[ ]之間。

(8) FAT 檔案系統其 DIR_NAME[ ]需全部為大寫字母。

(9) DIR_NAME 不允許使用一些特殊符號，如"+-*/:;<=>?[\]¦。

當知道判斷目錄磁區的名稱規則後，可以對檔案目錄的資料的進行讀取，圖 4-10 為讀取檔案的名稱、檔案容量、檔案建立日期、檔案建立時間、檔案屬性等資訊的 f_read_dir_info( )副程式。

程式名稱：f_read_dir_info ( )	輸　入：文件指標 DIR *scan
功能敘述：讀取文件資訊	輸　出：文件資訊結構 FILINFO *finfo

```
u8 f_read_dir_info (DIR *scan, FILINFO *finfo)
{
    u8 *dir, c;
    FATFS *fs = FatFs;

    while (scan->sect)
    {
        dir = &(fs->win[(scan->index & 15) * 32]);
        c = *dir;
        if (c == 0)
        {
            scan->index = 0;                          // 讀取文件資訊
            scan->sect = fs->dirbase;
            break;
        }
        if ((c != 0xE5) && (c != '.') && !(*(dir+11) & Attr_Volume))
        {
            finfo->fattrib = *(dir+11);               // 屬性
            finfo->fsize = LD_DWORD(dir+28);          // 容量 e
            finfo->fdate = LD_WORD(dir+24);           // 日期
            finfo->ftime = LD_WORD(dir+22);           // 時間
        }
        if (!next_dir_entry(scan))
            scan->sect = 0;                           // 下一個目錄磁區
        if (finfo->fname[0])
            break;                                    // 目錄磁區無資料
    }
    return FAT_OK;
}
```

圖 4-10 讀取文件目錄資訊副程式

4.5 資料區域

資料區域(簡稱 DATA)是儲存檔案資料的區域，它佔據儲存裝置絕大部份的儲存空間，圖 4-11 為 FAT 系統資料區域的儲存結構，叢集位址指標由 2 開始，叢集是由數個磁區所組成的儲存區塊，其所擁有的磁區數量與儲存裝置的總容量大小有直接關係，而且磁區個數需為 2 的倍數，其由 BPB 資訊中的 SectorPerCluster 參數來決定，另外每個叢集的最大容量不可以超過 32 Kbytes，因此當檔案容量小於叢集容量(32)時，該檔案依然會佔據一個叢集空間，多於的記憶空間無法被其他文件所使用 [8]。

$$N_{clus_size} = (N_{sec_per_clus} \times N_{sec_size}) \leq 32KB \qquad (32)$$

其中 $N_{sec_per_clus}$ 為叢集擁有的磁區數量，N_{sec_size} 為磁區容量。

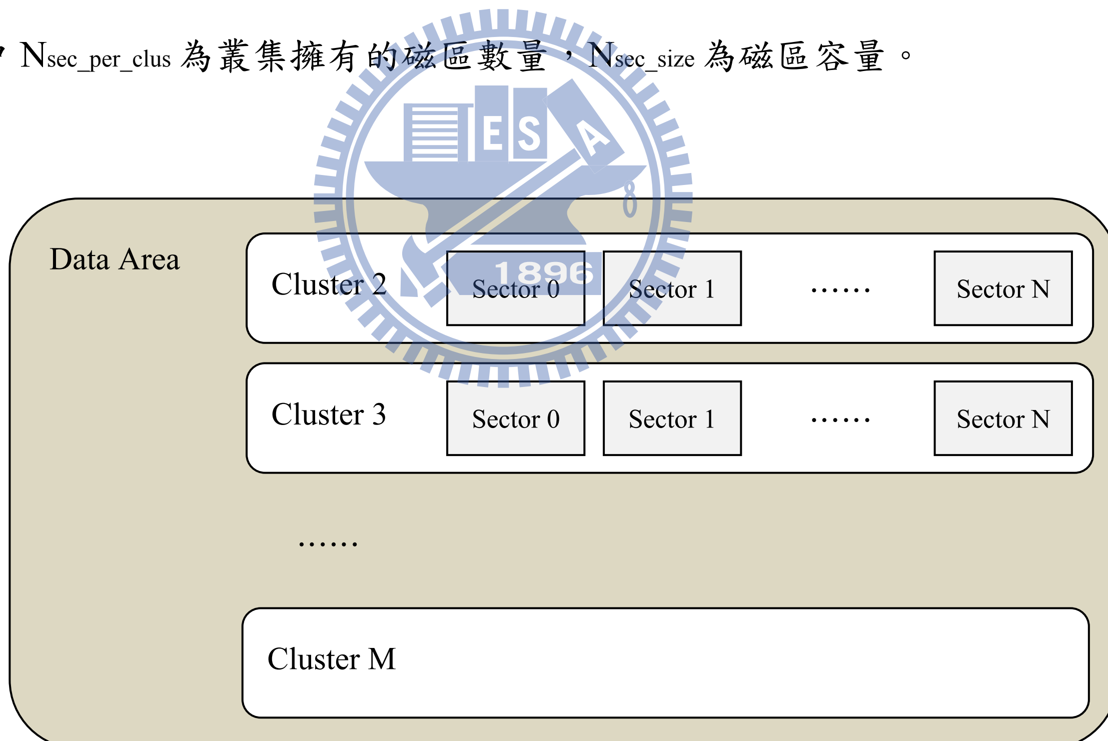


圖 4-11 資料區域的儲存結構

檔案資料是依照叢集位址儲存於磁碟裝置，因此要讀取資料時，需將叢集號碼 N 對應的磁區位址計算出來，才能夠找到資料的真實位址。由於叢集號碼 0 與 1 為保留區不使用，也不佔用叢集空間，因此叢集號碼 2 為第一個鏈結指標，所以叢集號碼減 2，乘上每個叢集磁區數再加上資料區的起始磁區位址，即可獲得叢集的磁區位址(33)，圖 4-12 為其轉換範例程式。

$$N_{clus2sec} = (N_{clus} - 2) \times N_{SecPerClus} + N_{DATA} \quad (33)$$

其中 $N_{clus2sec}$ 為叢集指標對應磁區位址，N_{clus} 為叢集指標位址，$N_{SecPerClus}$ 為叢集的磁區數量。

程式名稱：cluster2sector ( ) **輸　　入：叢集號碼 cluster**

功能敘述：將叢集號碼轉換為磁區位址 **輸　　出：磁區位址 sector**

```
u32 cluster2sector (u32 cluster)
{
    FATFS *fs = FatFs;
    u32 sector;
    cluster = cluster - 2;                                  // 叢集號碼 - 2
    if (cluster >= fs->max_clust)                           // 檢查叢集號碼是否超過最大值
        return 0;
    sector = cluster * fs->sects_clust + fs->database;      // 計算磁區位址
    return sector;                                          // 返回磁區位址
}
```

圖 4-12 轉換叢集次區位址副程式

FAT 文件資料讀取

FAT 檔案系統的資料區域結構是建立在叢集與磁區上面，圖 4-13 為依照緩衝區的容量來讀取 FAT 文件資料的 f_read ( )副程式，其流程為先導入檔案容量(fsize)與已讀取容量指標(fptr)來計算剩餘容量(ln)，如剩餘容量為零，則設定檔案結束(EOF)旗標。再利用迴圈來計數資料讀取的指標，其分

為完整磁區讀取與低於一個磁區讀取流程。在完整磁區讀取流程內，需要判斷是否達到叢集的磁區邊界，如達到邊界，則需切換到下一個叢集的第一個磁區；在小於一個磁區讀取流程中，通常為檔案資料的尾端，直接進行磁區讀取即可。

程式名稱：f_read_dir_info ( )	**輸　　入：FIL *fp，u8 *buff，u16 btw**
功能敘述：讀取文件資訊	**輸　　出：返回 FAT 狀態**

```
u8 f_read (FIL *fp, void *buff, u16 btr, u16 *br)
{
    u32       cluster, secter, ln, a, b;
    u16       rcnt;
    u8        cc, *rbuff = buff;
    FATFS     *fs = FatFs;

    *br = 0;
    a = fp->fsize;                          // 檔案容量
    b = fp->fptr;                           // 已讀取容量指標
    ln = a - b;                             // 未讀取容量
    if (ln == 0)                            // 剩餘容量為 0
        fp->eof = 1;                        // 設定 EOF 旗標= 1
    else
        fp->eof = 0;                        // 清除 EOF 旗標= 0
    if (btr > ln)                           // 緩衝容量指標>未讀取容量
        btr = (u16) ln;                     // 緩衝容量指標=未讀取容量
    for ( ; btr; rbuff += rcnt, fp->fptr += rcnt, *br += rcnt, btr -= rcnt)
    {   if ((fp->fptr % 512) == 0)          // 檢查是否為完整磁區
        {   if (--(fp->secter_cluster))     // 檢查叢集的磁區邊界計數器
                secter = fp->curr_secter + 1;   // 磁區位址+1
            else                            // 開啟下一個叢集區塊
            {   cluster = (fp->fptr == 0) ? fp->org_cluster : get_cluster(fp->curr_cluster);
                if ((cluster < 2) || (cluster >= fs->max_cluster))
                {   fp->flag |= FAT_ERR;
                    return FAT_R_ERR;
                }
                fp->curr_cluster = cluster;         // 設定目前的叢集指標
                secter = cluster2secter(cluster);   // 設定目前叢集的磁區位址
                fp->secter_cluster = fs->secter_cluster;   // 重設磁區邊界計數器
            }
            fp->curr_secter = secter;       // 設定目前的磁區位址
            cc = btr / 512;                 // 剩餘容量(磁區)= 未讀取容量/ 512
```

```
            if (cc)                                           // 剩餘容量(磁區) >= 1
            {  if (cc > fp->secter_cluster)                   // 剩餘容量(磁區) > 檢查叢集的磁區邊界計數器
                    cc = fp->secter_cluster;
               if (disk_read(rbuff, secter, cc) != SD_OK)     // 讀取緩衝容量磁區資料
               {  fp->flag |= FAT_ERR;
                  return FAT_R_ERR;
               }
               fp->secter_cluster -= cc - 1;                  // 剩餘容量(磁區) - 1
               fp->curr_secter += cc - 1;                     // 目前的磁區位址 +1
               rcnt = cc * 512;                               // 已讀取容量(完整磁區)
            }
            if (disk_read(fp->buffer, secter, 1) != SD_OK)    // 讀取一個磁區資料
            {  fp->flag |= FAT_ERR;
               return FAT_R_ERR;
            }
        }
        rcnt = 512 - ((u16)fp->fptr % 512);                   // 讀取資料筆數檢查
        if (rcnt > btr)
            rcnt = btr;                                       // 已讀取容量(不完整磁區)
        memcpy(rbuff, &fp->buffer[fp->fptr % 512], rcnt);
    }
    return FAT_OK;
}
```

圖 4-13 FAT 系統文件資料讀取副程式

FAT 文件資料寫入

本系統主要應用在多媒體的播放，所以並未使用到 FAT 檔案系統的資料寫入程序，因此只說明寫入磁區的方式，將資料寫入叢集磁區時首先要找到一個未使用的叢集位址，並在文件分配表(FAT)與目錄(DIR)磁區建立相關資訊，接著寫入文件資料，如該文件資料未超過一個叢集容量，則完成寫入程序；若文件資料超過一個叢集容量，當該叢集位址所有磁區寫滿後，要找到另一個未使用的叢集位址，並在文件分配表上建立鏈結，重複此動作，直到剩餘資料寫完。

第五章　WAVE 音效檔

WAVE 文件格式是一種由微軟和 IBM 聯合開發的用於音訊數位存儲的音訊文件標準，它採用 RIFF (Resource Interchange File Format) 文件格式的結構 [11]。WAVE 文件支援多種壓縮演算法，也支援多種資料位元數、取樣頻率和多聲道，而採用 44.1kHz 取樣頻率與 16 位元取樣數的音質與 CD 相差無幾。

WAVE 文件格式的優點為使用簡單的編解碼程序及廣泛的支援，而主要的缺點是需要較大的存儲空間。WAVE 文件主要使用三個參數來表示聲音訊號：取樣位元數(Sample per bits)、取樣頻率(Sample rate)和聲道數(Channel number)。聲道數可分為單聲道(Mono)和身歷聲(Stereo)，取樣頻率一般常用的有 8000Hz、11025Hz、22050Hz 和 44100Hz 四種，取樣位元數有 8 bits、16 bits 兩種。(34) 為 WAVE 文件所需要的儲存容量計算方式。

$$N_{size} = f_{sr} \times N_{bit} \times N_{ch} \times T_s / 8 \tag{34}$$

其中 N_{size} 為儲存容量，N_{bit} 為取樣位元數，N_{ch} 為聲道數，f_{sr} 為取樣頻率。

假若 WAVE 文件的聲道數為雙聲道，取樣頻率為 44.1KHz，取樣位元數為 16 位元，取樣時間為 60 秒，則音效資料容量約為 10Mbytes。

N_{size} = 44100Hz × 16bits × 2ch × 60sec / 8bits = 10584000 bytes

5.1 WAVE 檔案資料結構

WAVE 文件是以 RIFF 格式為標準的多媒體音訊文件格式之一，所有的 WAVE 文件都有一個記錄音訊資料的檔頭(以下簡稱 Header)，並且文件是由許多資料塊(以下簡稱 Chunk)所組成的。WAVE 文件起始的 Header 裡面包含了此音效文件的編碼格式(如 PCM 或 IEEE Float)與播放參數(如資料區容量、聲道數、取樣頻率、取樣位元數)等參數。在 Header 後接著的是資料區，其儲存音訊資料，其中雙聲道的存放次序為左聲道、右聲道。在進行播放音效之前需將這些參數解碼後，並載入播放器中才能夠播放正確音效。底下將對 WAVE 文件的各區域進行說明 [11]。

RIFF chunk

RIFF 是多媒體文件的標準格式之一，檔案開頭以“RIFF”作為識別標示，而 RIFF 支援多種多媒體資料格式，其中就包括 WAV 音樂格式，並以“WAVE”作為識別標示。表 5-1 為 RIFF-WAVE 文件的資料結構，開頭的 Chunk 識別碼為 “RIFF”，用來標示屬於 RIFF 文件格式，然後為 Chunk size 容量標示欄位，該容量數值為 WAVE chunks 大小加上 4 位元組。最後為 WAVE 資料區，包含以“WAVE” 為標示的 Chunk ID，與存放音樂資料的 WAVE chunk (包括 Format 和 Sampled data) [12]。

表 5-1 WAVE 文件結構 [12]

WAVE File Format Structure		
Field	**Length**	**Contents**
Chunk ID	4	Chund ID : “RIFF”
Chunk Size	4	Chunk size : 4 + n bytes
WAVE ID	4	WAVE ID : “WAVE”
WAVE chunks	n	Wave chunks containing format information and sampled data.

Format chunk

為 WAVE chunk 的子資料塊(簡稱為 sub-chunk)，表 5-2 為 Format chunk 的資料結構，其以"fmt "作為開始標示，然後為容量欄位元，該數值為 16，18 或 40，用來表示整個 chunk 的資料容量，接著為編碼方式、聲道數、取樣頻率及取樣位元數等資訊。

表 5-2 WAVE Format chunk 格式 [12]

WAVE Format Chunk		
Field	**Length**	**Description**
SubChunk1ID	4	Contains the letters "fmt "(0x666d7420 big-endian form).
SubChunk1Size	4	This is the size of the rest of the Subchunk Chunk size : 16, 18 or 40, (16 for PCM.)
AudioFormat	2	Format code, PCM = 1 (i.e. Linear quantization)
NumChannels	2	Mono = 1, Stereo = 2, etc.
SampleRate	4	8000, 44100, etc.
ByteRate	4	SampleRate * NumChannels * BitsPerSample / 8
BlockAlign	2	NumChannels * BitsPerSample / 8
BitsPerSample	2	8 bits = 8, 16 bits = 16, etc.
ExtensionSize	2	Size of the extension (0 or 22)
ValidBitsPerSample	2	Number of valid bits
ChannelMask	4	Speaker position mask
SubFormat	16	GUID, including the data format code

WAVE format chunk 的 Audio Format 參數如表 5-3 所列，是用來表示音訊的編碼格式。

表 5-3 WAVE 編碼方式 [12]

Format code	Preprocessor Symbol	Data
0x0001	WAVE_FORMAT__PCM	PCM
0x0003	WAVE_FORMAT__IEEE_FLOAT	IEEE float
0x0006	WAVE_FORMAT__ALAW	8-bit ITU-T G.711 A-law
0x0007	WAVE_FORMAT__MULAW	8-bit ITU-T G.711 u-law
0xFFFE	WAVE_FORMAT__EXTENSIBLE	Determined by SubFormat

Fact chunk

所有的非 PCM 編碼格式都必須要有此 Fact chunk，其以“fact”作為標示，表 5-4 為其資料結構，是由某些特殊軟體在轉換音訊時產生的，一般並不常在 WAVE 文件中見到。

表 5-4 WAVE fact chunk 結構 [12]

Fact Chunk		
Field	**Length**	**Contents**
Chunk ID	4	Chund ID : “fact”
Chunk Size	4	Chunk size : minimum 4
SampleLength	4	Number of samples (per channel)

Data chunk

表 5-5 為 Data chunk 的資料結構，以“data”作為開始標示，然後為音訊資料的容量欄位，接著就是編碼過的音訊資料，如果資料容量為奇數，則最後會多一個 pad byte。

表 5-5 WAVE data chunk 結構 [12]

Data Chunk		
Field	**Length**	**Contents**
Chunk ID	4	Chund ID : “data”
Chunk Size	4	Chunk size : n
Sample data	n	Samples
Pad byte	0 or 1	Padding byte if n is odd

Data Chunk 是實際存放 wav 音訊資料的地方，根據 Format Chunk 中的聲道數以及取樣位元數，wave 資料的位置配置方式如圖 5-1，可以分成四種配置形式，圖(a)為 8 位元單聲道，(b)為 8 位元雙聲道，(c)為 16 位元單聲道，(d)為 16 位元雙聲道。

單聲道	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6
8 位元	左聲道	左聲道	左聲道	左聲道	左聲道	左聲道

(a) 8 位元單聲道

雙聲道	Sample 1		Sample 2		Sample 3	
8 位元	左聲道	右聲道	左聲道	右聲道	左聲道	右聲道

(b) 8 位元雙聲道

單聲道	Sample 1		Sample 2		Sample 3	
16 位元	左聲道		左聲道		左聲道	
	低位元	高位元	低位元	高位元	低位元	高位元

(c) 16 位元單聲道

雙道	Sample 1				Sample 2	
16 位元	左聲道		右聲道		左聲道	
	低位元	高位元	低位元	高位元	低位元	高位元

(d) 16 位元雙聲道

圖 5-1 WAVE 音訊資料配置結構

當 WAVE 文件的音訊參數為 PCM 編碼格式，N_{ch} 個聲道數，取樣頻率為 N_{SF}，取樣位元為 N_{B}，取樣時間為 N_{S}，則其 Header 的參數計算方式如表 5-6 所列。

表 5-6 WAVE 結構範例 [12]

Field	Length	Contents
Chunk ID	4	Chund ID : “RIFF”
Chunk size	4	Chunk size : 4 + n
Wave ID	4	Wave chunk ID : “WAVE”
Sub chunk ID	4	Chunk ID : “fmt “
Sub chunk size	4	Chunk size : 16
AudioFormat	2	WAVE_FORMAT_PCM = 0x0001
NumChannels	2	Nch
SampleRate	4	NSF
Byterate	4	NSF * Nch * NB / 8
BlockAlign	2	Nch * NB / 8
BitsPerSample	2	NB
Data chunk	4	Chunk ID : “data”
Data chunk size	4	Chunk size : Nch * Ns * NB / 8
Sampled data	Nch * Ns * NB / 8	Nch * Ns
pad	0 or 1	

圖 5-2 為 Wave header 內容範例，我們可以看到 Header 開頭前 4 bytes Chunk ID 一定是”RIFF”，而且 Format 內容為”WAVE”，接著可得到聲道數為雙聲道，取樣頻率為 22050Hz，取樣位元數為 8 bits，音效資料容量為 9770498 bytes 等資訊。

chunk descriptor = ‘RIFF’				chunk size = 9770534								fmt subchunk = ‘fmt ‘			
52	**49**	**46**	**46**	**26**	**16**	**95**	**00**	**57**	**41**	**56**	**45**	**66**	**6D**	**74**	**20**
R	**I**	**F**	**F**					**W**	**A**	**V**	**E**	**f**	**m**	**t**	
Subchunk1 size = 16				Audio Format = 1 (PCM)		NumChannels = 2		Sample Rate = 22050				Byte rate = 44100			
10	**00**	**00**	**00**	**01**	**00**	**02**	**00**	**22**	**56**	**00**	**00**	**44**	**AC**	**00**	**00**
Block Align = 2		BitPerSample = 8		Data subchunk = ‘data’				Subchunk2Size = 9770498				Sample 1 (L/R)		Sample 2 (L/R)	
02	**00**	**08**	**00**	**64**	**61**	**74**	**61**	**02**	**16**	**95**	**00**	**95**	**90**	**9B**	**96**
				d	**a**	**t**	**a**								
Sample 3 (L/R)		Sample 4 (L/R)		Sample 5 (L/R)		Sample 6 (L/R)		Sample 7 (L/R)		Sample 8 (L/R)		Sample 9 (L/R)		Sample 10 (L/R)	
78	**6D**	**7D**	**74**	**7E**	**75**	**83**	**82**	**84**	**84**	**82**	**8C**	**83**	**90**	**7F**	**85**

圖 5-2 RIFF WAVE 檔頭格式

5.2 WAVE 檔案播放

播放 PCM 編碼格式的 Wave 音效流程並不會很複雜，圖 5-3 為本系統的 WAVE 播放流程，主要分為檔頭解碼、播放頻率設定、填補緩衝區資料、資料位元組讀取與 PWM 中斷輸出等程序，其中 TIM4 工作在 WAVE 的取樣頻率，用來讀取 WAVE 音訊資料並輸出給 PWM 模組產生聲音 [13]。

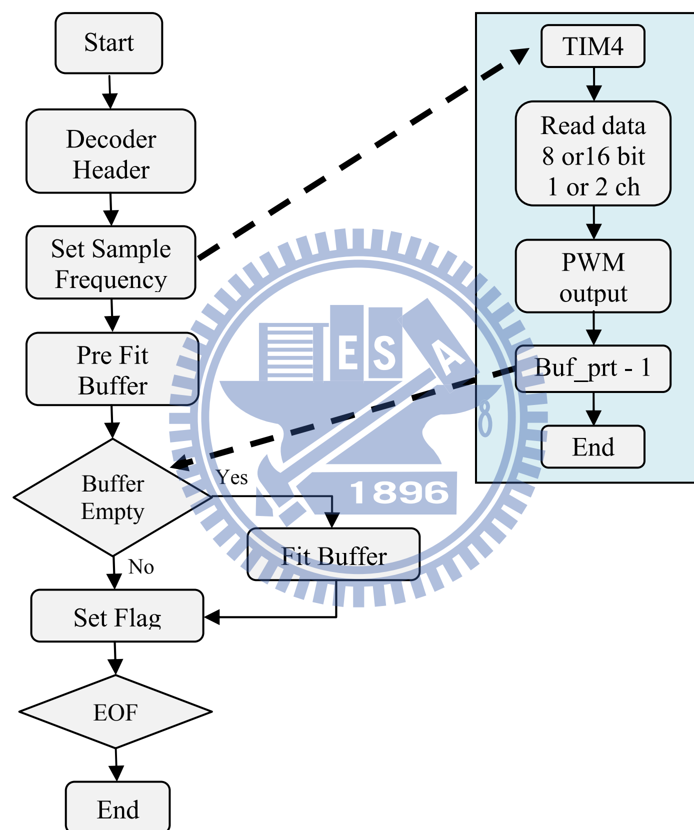


圖 5-3 WAVE 播放流程圖 [13]

5.2.1 WAVE 檔頭解碼

WAVE 檔頭解碼流程如圖 5-4 之範例程式，首先讀取 WAVE 文件的第一個磁區，並將前 36 位元組資料載入 WHEADER 結構中，再判斷 Format chunk 的容量欄位是否為 18，如為 18 則檔頭容量多 2 個位元組，接著設定資料開始位置指標，取樣位元數與設定解碼完成旗標。

程式名稱：**decord_wav_header ( )**	輸　入：**WAVE Header buffer data**
功能敘述：**解碼 WAVE 文件檔頭資訊**	輸　出：**typedef struct WHEADER**

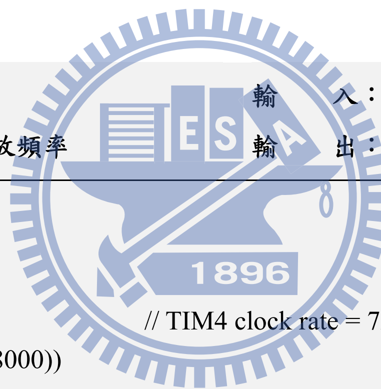
```
void decord_wav_header (void)
{
    int i;
    WHEADER *wh = (WHEADER *) winfo->buf;            //位址指向 buffer

    winfo->fptr         = 0;
    winfo->fsize        = wh->ChunkSize;             //讀取 Chunk size
    winfo->channel      = wh->NumChannel;            //讀取  number of channel
    winfo->align        = wh->BlockAlign;            //讀取  Block align
    winfo->samplerate   = wh->SampleRate;            //讀取  Sample rate refquency
    winfo->sample       = wh->BitPerSample;          //讀取  Bit per sample

    if (wh-> SubChunk1Size == 16)                    //尋找正確的 header 空間
        i = 0;
    else if (wh-> SubChunk1Size == 18)
        i = 2;
    winfo->dsize = LD_DWORD(winfo->buf[40+i])
    winfo->start =36 + 8 + i;

    if (winfo->sample == BIT8)                       //設定 flag
        p_sample = BIT8;
    else if (winfo->sample == BIT16)
        p_sample = BIT16;
    winfo->flag = WAV_INITOK;                        //設定 flag
}
```

圖 5-4 WAVE 檔頭解碼副程式

在檔頭解碼完成之後，依照檔頭的取樣頻率數值來設定 TIM4 計時器的中斷頻率，首先設定 TIM4_PSC 為 0，此時計時器主頻率(f_{CNT})為 72MHz 系統頻率(f_{sys})，再將 f_{CNT} 的時脈頻率除以取樣頻率後再減一，就可以得到 TIM4 計時器的 ARR 暫存器設定參數。圖 5-5 的 set_wav_freq ( )副程式為計時器的中斷頻率設定流程，將取樣頻率使用(35)可得到 TIM4 頻率參數，填入 TIM4_ARR 暫存器即完成設定。

$$N_{ARR} = (f_{TIM4}/f_{sr}) - 1 \tag{35}$$

其中 N_{ARR} 為 ARR 暫存器設定參數，f_{TIM4} 為計時器的工作頻率，f_{sr} 為取樣頻率。



程式名稱：**set_wav_freq ( )**	輸　　入：**u32 freq**
功能敘述：**設定 WAVE 文件播放頻率**	輸　　出：**TIM4_ARR**

```
void set_wav_freq (u32 freq)
{
   u32 f_set;
   TIM4->PSC = 0x0000;                    // TIM4 clock rate = 72MHz / (0 + 1) = 72MHz
   if ((freq<=44100) && (freq>=8000))
      f_set = (72000000 / freq) - 1;
   else
      f_set = 0x2327;                     // 8KHz
   TIM4->ARR = (u16) f_set;
}
```

圖 5-5 取樣頻率設定副程式

5.2.2 緩衝區設定

在播放音樂的狀態下，此時播放器會經由 TIM4 中斷副程式，依照取樣頻率的設定來讀取緩衝區內的音效資訊，當緩衝區內資料被全部讀取完畢後，程式必須要將下一區塊的音效資訊存放進緩衝區內，等待播放器讀取。

在使用一個緩衝區的情況下，當播放完緩衝區內的資料後，必須先填滿緩衝區後才能繼續播放音樂，這時候如果緩衝區太大或是讀取資料速度太慢的情況下，則填滿緩衝區的時間會被拉長，如果拉長的時間超過取樣頻率的週期，會造成播放音樂不流暢的現象。考慮到常見 WAVE 文件的採樣頻率最高為 44.1KHz，則填滿緩衝區的時間就必須小於 22usec，才不會發生播放不流暢的現象。

但是填滿緩衝區小於 22u sec 的規格太過嚴苛，一般的微控器大多無法達到。因此，通常會使用 2 組以上的緩衝區來避免播放不流暢的現象，當緩衝區 1 在播放時，緩衝區 2 則進行資料讀取，播放順序為緩衝區 1、緩衝區 2、緩衝區 1 等循環。假設每個緩衝區的容量為 512 bytes，播放雙聲道、16 位元、44.1KHz 的 WAVE 文件時，每一個緩衝區可播放的音訊筆數可由(36)得知為 128 筆，且播放時間由(37)得知為 2.95m sec，因此填滿緩衝區的時間只要小於 2.95m sec 即不會發生播放不流暢的現象。

$$N_s = N_{buff} / \left(N_{ch} \times (N_{bit} \div 8)\right) \tag{36}$$

$$T_{PB} = N_s / f_{sr} \tag{37}$$

其中 N_s 為取樣資料數，N_{buff} 為緩衝區的容量(Bytes)，N_{ch} 為聲道數，N_{bit} 為取樣位元數，T_{PB} 為播放緩衝區所需的時間(sec)，f_{sr} 為取樣頻率。

圖 5-6 為本系統使用 2 組緩衝區來儲存音樂資料的選擇範例程式。

程式名稱：**wav_2buffer_select ( )**	輸　入：目前使用中的緩衝區，**winfo->ctrl**
功能敘述：**2** 組緩衝區的選擇設定	輸　出：要填入資料的空閒緩衝區，**buf_sel**

```
int wav_2buffer_select (void)
{  int buf_sel;
   switch (winfo->ctrl)
   {  case    0:                        // 目前為緩衝區 0
         if (winfo->ptr[0] == 0)
            buf_sel = 0;                // 緩衝區 0 為空
         else if  (winfo->ptr[1] == 0)
            buf_sel = 1;                // 緩衝區 1 為空
         break;
      case    1:                        // 目前為緩衝區 1
         if (winfo->ptr[1] == 0)
            buf_sel = 1;                // 緩衝區 1 為空
         else if  (winfo->ptr[0] == 0)
            buf_sel = 0;                // 緩衝區 0 為空
         break;
   }
   return buf_sel;
}
```

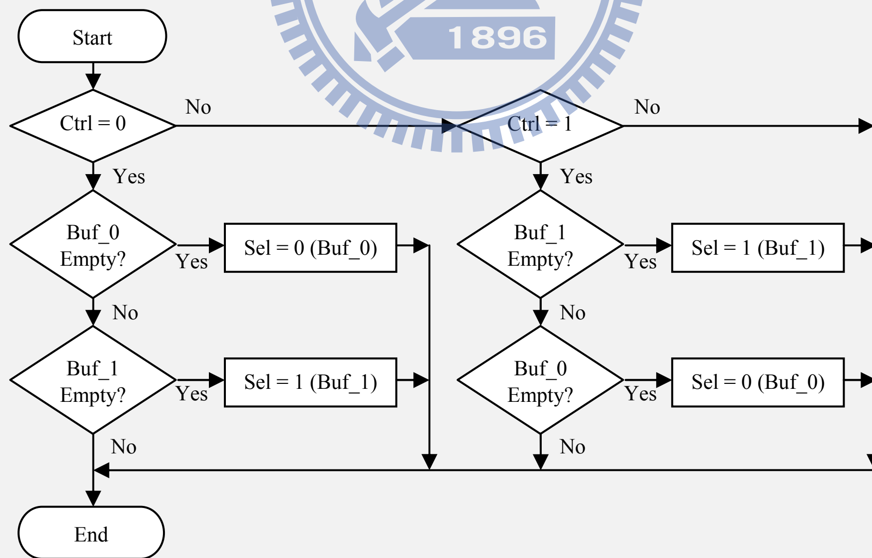


圖 5-6 2 組緩衝區設定副程式

5.2.3 音訊解碼流程

WAVE 文件通常使用的音訊編碼方式是脈衝碼調變(PCM)。脈衝碼調變是一種單純的數位編碼格式，音訊在固定週期內進行取樣並數位化為頻帶值。因此，一個取樣值代表了固定週期內的音訊信號，將特定時間內的音訊經由適當數量的取樣，便能構成完整的音訊文件。因此，對於不同取樣位元、取樣週期與取樣通道的音訊資料，必須從緩衝區中將資料正確的分配到各自的通道上，才能得到原始的音樂。

WAVE 文件的取樣位元數有 8 位元與 16 位元兩種資料型式，其中 8 位元資料取樣格式為無號數(unsigned)的資料型式，資料 0x00 至 0xFF 所代表的數值為 0 至 255。而 16 位元資料取樣格式為有號數(signed)的資料型式，其以最高位元(bit 15)來表示數值是正數或負數，稱為符號位元，當最高位元為 0 時，表示為正數；而最高位元為 1 時，表示為負數。因此資料 0x0000 至 0x7FFF 代表的數值為 0 至 32767；而資料 0x8000 至 0xFFFF 代表的數值為-32768 至-1。因此我們在進行資料處理時，要將有號數轉換為無號數，令其數值轉換為 0 至 65535，因此正數須加 0x8000 偏移量，使其變為 32768 至 65535；而負數需減 0x8000 偏移量，使其變為 32767 至 0。圖 5-7 為將 16 位元有號數的數值轉換為無號數的副程式。

程式名稱：trans2signed ( )	**輸　入：temp_reg**
功能敘述：將 16 位元有號數轉為無號數	**輸　出：temp_reg**

```
u16 trans2signed (u16 temp_reg)
{  if (temp_reg < 0x8000)                // 檢查是否為正數 (0~0x8000 = 0 ~ 32767)
      temp_reg = temp_reg + 0x8000;      // 如為正數，則加 0x8000 (0x8000~ 0xFFFF = 32768 ~ 65535)
   else                                  // 如為負數 (0x8000~0xFFFF = -32768 ~ -1)
      temp_reg = temp_reg – 0x8000;      // 減 0x8000 (0~0x7FFF = 0 ~ 32767)
   return tesmp_reg
}
```

圖 5-7 有號數轉換為無號數的副程式

圖 5-8 為本系統用來從 WAVE 檔案中，依照所需的位元數與聲道數來讀取音訊資料的解碼程式。

程式名稱：wave_decorder ( )	**輸　　入：資料緩衝區，winfo->buff**
功能敘述：解碼 WAVE 音訊資料，單/雙聲道，8/16 位元	**輸　　出：左/右聲道**

```
u8 wave_decorder (u16 *lv, u16 *rv)
{
    u32 ptr;
    u16 tmp_l, tmp_r;
    if (winfo->fptr == 0)          winfo->bptr = 0;        // 資料位置為 0？播放指標清為 0
    if (winfo->fptr >= winfo->dsize)                        // 資料位置大於文件容量 ？
    {  winfo->flag = WAV_EOD;                               // 設定旗標為播放結束
       mute_pwm();                                          // 靜音，PWM 輸出設為 0
       return winfo->flag;
    }
    if (winfo->fptr < winfo->start)                         // 資料位置在檔頭位置中 ？
    {  winfo_pointer_inc (0x01);                            // 播放與資料指標加 1 位元組
       mute_pwm();                                          // 靜音，PWM 輸出設為 0
       return winfo->flag;
    }
    if ( (ptr = get_wav_offset(winfo->ctrl)) == 0)          // 讀取與檢查緩衝區指標，0 為無資料
    {  mute_pwm();                                          // 靜音，PWM 輸出設為 0
       return 0;
    }
    switch (winfo->sample)                                  // 檢查取樣位元數
    {
    case  BIT8:                                             // 取樣位元數 = 8bits
       switch (winfo->channel)                              // 檢查聲道數
       {
       case  MONO:                                          // 單聲道
          tmp_l = winfo->buf[winfo->ctrl][winfo->bptr];     // 讀取 1 位元組單聲道
          tmp_r = tmp_l;                                    // 右聲道 = 左聲道
          break;
       case  STEREO:                                        // 雙聲道
          tmp_l = winfo->buf[winfo->ctrl][winfo->bptr];     // 讀取 1 位元組左聲道
          tmp_r = winfo->buf[winfo->ctrl][winfo->bptr+1];   // 讀取 1 位元組右聲道
          winfo_pointer_inc (0x01);                         // 播放與資料指標加 1 位元組
          break;
       }
       winfo_pointer_inc (0x01);                            // 播放與資料指標加 1 位元組
       break;
```

```
    case  BIT16:                                                    // 取樣位元數 = 16bits
        switch (winfo->channel)                                     // 檢查聲道數
        {
        case  MONO:                                                 // 單聲道
            tmp_l = (u16) winfo->buf[winfo->ctrl][winfo->bptr];                // 讀取左聲道低位元組
            tmp_l = tmp_l + (u16) (winfo->buf[winfo->ctrl][winfo->bptr+1] << 8);   // 讀取左聲道高位元組
            tmp_r = tmp_l;                                          // 右聲道 = 左聲道
            winfo_pointer_inc (0x02);                               // 播放與資料指標加 2 位元組
            break;
        case  STEREO:                                               // 雙聲道
            tmp_l = (u16) winfo->buf[winfo->ctrl][winfo->bptr];                // 讀取左聲道低位元組
            tmp_l = tmp_l + (u16) (winfo->buf[winfo->ctrl][winfo->bptr+1] << 8);   // 讀取左聲道高位元組
            tmp_r = (u16) winfo->buf[winfo->ctrl][winfo->bptr+2];              // 讀取右聲道低位元組
            tmp_r = tmp_r + (u16) (winfo->buf[winfo->ctrl][winfo->bptr+3] << 8);   // 讀取右聲道高位元組
            winfo_pointer_inc (0x04);                     // 播放與資料指標加 4 位元組
            break;
        }
        break;
    }
    if (winfo->bptr == ptr)                               // 播放指標 = 緩衝區指標 ?(緩衝區全部讀取)
    {
        set_wav_offset(winfo->ctrl, 0);                   // 清除緩衝區指標
        winfo->ctrl ++;                                   // 選擇下一個緩衝區
        winfo->bptr = 0;                                  // 清除播放指標
        if (winfo->ctrl >= WAVBUF_NUM)                    // 檢查是否為最後一個緩衝區 ?
            winfo->ctrl = 0;                              // 選擇第一個緩衝區
    }
    *lv = tmp_l;                                          // 更新左聲道播放資料
    *rv = tmp_r;                                          // 更新右聲道播放資料
    return winfo->flag;
}
```

圖 5-8 音訊解碼副程式

5.2.4 播放流程

圖 5-9 為本系統的 WAVE 播放流程圖，其可分成 6 個步驟，當系統讀取到 WAVE 文件後，可以透過 PLAY、STOP、NEXT 按鍵來改變播放狀態旗標的狀態，圖 5-10 為 WAVE 播放控制流程的副程式。

PlayInit 狀態，在按下 PLAY 按鍵後，旗標狀態會設為 PlayInit，此時會先進行 WAVE 檔頭的解碼和 TIM4 中斷程式的頻率設定，完成後將旗標狀態設為 Play。

Play 狀態，在此狀態下會先將所有緩衝區填滿，再進行 WAVE 解碼流程，並檢查文件是否播放完畢，按下 STOP 與 NEXT 按鍵，會停止播放音樂並進入其他狀態。

Next 狀態，在播放過程中按下 NEXT 按鍵，可以直接尋找下一個 WAVE 音效檔案，並且直接進行播放。如果搜尋到目錄尾端時，會回到根目錄下的第一個檔案位置重新搜尋。

Stop 狀態，按下 STOP 按鍵後，會直接中斷播放流程，並將播放位置指標清除，當再次按下 PLAY 鍵時，會從頭開始播放。

Pause 狀態，在播放過程中按下 PLAY 鍵可暫停播放音樂，直到再次按下 PLAY 鍵才會中斷位置繼續播放。

EOF 狀態，當 WAVE 文件播放結束後，播放旗標會設為 EOF 狀態，這時會關閉 TIM4 中斷功能，並使 PWM 模組輸出為 0，達到靜音效果，並尋找下一個 WAVE 文件。

AutoPlay 狀態，當自動播放旗標設定後，每播放完一個 WAVE 文件，系統會自動尋找下一個 WAVE 文件，並自動執行播放動作。

圖 5-9 WAVE 播放功能流程

程式名稱：**wav_play_flow ( )**	輸　入：目前播放狀態，**p_state**
功能敘述：**WAVE 文件播放選擇流程**	輸　出：下一個播放狀態，**p_state**

```
void wav_play_flow (void)
{  switch (p_state)
   {  case S_PLAYINIT:     //Wave 初始化狀態
         wav_init (fil);
         p_state = S_PAUSE;
         if (p_auto == S_AUTO)
            p_state = S_PLAY;
      break;
      case S_PLAY:         //播放狀態
         play_wav(fil);
         p_auto = S_AUTO;
      break;
      case S_EOF:          //檔案播放結束狀態
         p_state = S_NEXT;
      break;
      case S_NEXT:         //找下一首狀態
         turn_off_pwm();
         found_song(dir, fil, filinfo);
         p_state = S_NEXTGO;
      break;
      case S_NEXTGO:
         if (p_auto == S_AUTO)
            p_state = S_PLAYINIT;
         else
            p_state = S_STOP;
         SysDelay(1000);
         while (TDelay !=0);
      break;
      case S_STOP:         //停止播放狀態
         set_top_of_wave ();
         p_auto = S_NOAUTO;
         p_state = S_PAUSE;
      break;
      case S_PAUSE:        //暫時停止播放狀態
         turn_off_pwm();
      break;
   }
}
```

圖 5-10 WAVE 播放功能副程式

第五章　MP3 音效格式

Moving Picture Experts Group Audio Layer III 又被稱為 MP3，是當今廣為流行的一種數位音訊編碼和破壞性壓縮格式，它被設計用來大幅度地降低音訊資料量，而對於大多數使用者的聽覺感受來說，重放的音質與最初的無壓縮音訊相比沒有明顯的下降。

MP3 是一種資料壓縮格式，它捨棄了音訊資料在脈衝編碼調變（Pulse Code Modulation）中對人類聽覺不重要的資料，從而達到了壓縮成較小檔案的目的。當談及數位媒體的所需頻寬時，會以位元速率(N_{BPS})描述，位元速率指的是每一秒位元資料的流量，單位是 bps (bit per second)。而將位元速率乘上取樣時間就可以得到所須要的儲存空間(N_{size}) [14]。

以 CD 數位音訊的音質為例，其取樣位元數為 16 位元，取樣頻率為 44.1KHz，雙聲道音訊，則每秒位元速率由(38)可為得 1411.2kbps，每分鐘所須的儲存空間由(39)可得為 10584000 bytes。當選擇 128Kbps 位元速率來進行 MP3 壓縮時，可算出其壓縮倍數為 11.025 (1411.2 / 128)，並且能達到相當不錯的音質，而每分鐘的歌曲經過壓縮後，約只需要 0.9MBytes 儲存空間即可。

$$N_{BPS} = f_s \times N_{bit} \times N_{ch} \tag{38}$$

$$N_{size} = N_{BPS} \times T_s / 8 \tag{39}$$

其中 N_{BPS} 為位元速率，f_s 為取樣頻率，N_{bit} 為取樣位元，N_{ch} 為聲道數，N_{size} 為資料容量，T_s 為取樣時間。

6.1 MP3 文件的標籤格式

MPEG Layer I、Layer II 與 Layer III(MP3)音頻格式並沒有內置保存信息內容的方式，而為了解決這一問題 MP3 文件引入了“Studio 3”的標籤結構，用來記錄音樂文件的訊息，表 6-1 為 MP3 的檔案結構。其中有兩組標籤資訊，分別為儲存在檔案尾端的 TAG1 與檔案開頭的 TAG2。

標籤(TAG)可以將該歌曲的相關資訊如專輯名稱、曲名、演唱者、出品年代、歌詞等詳細資訊附加在 MP3 檔案中，目前較為常用的標籤格式為 ID3，分為 V1 和 V2 兩個版本，其中 V2 儲存於檔案的開頭，而 V1 儲存於末尾，底下將簡介 ID3V1 與 ID3V2 標籤的結構，詳細的 ID3 標籤應用方式請參考 ID3 官方網頁 http://www.id3.org 的 ID3 tag version 2.3 文件。

表 6-1 MP3 檔案結構

MP3 File Structure				
TAG_2	Frame 1	………	Frame N	TAG_1

ID3V1 標籤

ID3V1 的資料結構可以很容易被程式解碼，其資料長度為固定的 128 位元組，並且位於 MP3 檔案的最尾端，由於其結構並沒有保留供將來使用的空間，因此存放的訊息種類並不多，圖 6-1 為 ID3V1 標籤的資料結構定義，其包含了作者，作曲，專輯內容等訊息，每一個段落最長為 30 個字元，且無法擴充與缺乏彈性。

程式名稱：typedef struct TagID3V1

功能敘述：MP3 ID3V1 結構定義

```
typedef struct TagID3V1
{
      char  Header[3];          // "TAG"
      char  Title[30];          // 標題
      char  Artist[30];         // 作者
      char  Album[30];          // 專輯
      char  Year[4];            // 出品年代
      char  Comment[28];        // 備註欄
      char  reserve;            // 保留, 0
      char  track;              // 音軌
      char  Genre;              // 類型
}     ID3V1;
```

圖 6-1 ID3V1 資料結構定義

圖 6-2 為本系統使用的 MP3 音樂文件所記錄的 ID3V1 標籤內容，我們可以清楚的看出其標題、作者與出品年代等內容訊息。

```
004A5D30  00 00 00 00 00 00 00 00  54 41 47 4B 6E 6F 77 69          TAGKnowi
004A5D40  6E 67 20 4D 65 2C 20 4B  6E 6F 77 69 6E 67 20 59  ng Me, Knowing Y
004A5D50  6F 75 00 00 00 00 00 00  00 41 42 42 41 00 00 00  ou         ABBA
004A5D60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
004A5D70  00 00 00 00 00 00 00 47  6F 6C 64 00 00 00 00 00         Gold
004A5D80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
004A5D90  00 00 00 00 00 31 39 37  37 20 30 30 30 30 31 35       1977 000015
004A5DA0  37 36 20 30 30 30 30 30  36 41 41 20 30 30 30 30  76 000006AA 0000
004A5DB0  43 37 33 32 20 00 02 0D                           C732
```

圖 6-2 MP3 ID3V1 資料內容

ID3V2 標籤

ID3V2 的資料格式定義與 ID3V1 截然不同，其具有較大的靈活度與自由度可供使用者添加訊息，目前最常使用的 ID3V2 格式是第三版，所以又被稱為 ID3V2.3，由於 ID3V1 已經佔據在檔案的最尾端，因此 ID3V2.3 被存放於檔案的起端。ID3V2.3 的結構包含了一個標籤頭(Header)與數個標籤

框(Frame)，Header 共有 10 個位元組用來紀錄整個 ID3V2.3 標籤的容量大小，其結構如圖 6-3。

程式名稱：typedef struct TagID3V2.3_H

功能敘述：MP3 ID3V2.3 結構定義

```
typedef struct TagID3V2.3_H
{
      char  Header[3];                  // "ID3"
      char  Ver;                        // 版本號 ID3V2.3 … "3"
      char  Revision;                   // 副版本號，"0"
      char  Flag;                       // 旗標
      char  Size[4];                    // 標籤內容大小
} H_ID3V2.3;
```

圖 6-3 ID3V2.3 結構定義

在此須要注意標籤容量的計算方式與一般常見的方式不太一樣，標籤容量有 4 bytes，但是每一個 byte 只使用 7 個 bit，最高位元不使用，因此在計算容量時要進行移位的動作，其計算程式如圖 6-4。

程式名稱：Count_ID3V2_Size( )

功能敘述：計算 MP3 ID3V2.3 容量

```
u32 Count_ID3V2_Size (u8 *Size)
{
   u32 ID3V2_size;
   ID3V2_size    = (int) (Size[0] & 0x7E) << 21
                 + (int) (Size[1] & 0x7E) << 14
                 + (int) (Size[2] & 0x7E) << 7
                 + (int) (Size[3] & 0x7E);
   return ID3V2_size;
}
```

圖 6-4 ID3V2.3 容量計算程式

ID3V2 標籤內容是由數個標籤框構成，每個標籤框就代表一種訊息內容，而且長度可以自由的擴展，標籤框由 10 位元組的框頭(Header)與非固定長度的內容所組成，圖 6-5 為 ID3V2.3 標籤框頭的資料結構。圖 6-6 為本系統使用的 MP3 音樂文件所記錄的 ID3V2.3 標籤內容，我們可以看出每個標籤框內容訊息。

程式名稱：typedef struct TagID3V2.3_H

功能敘述：MP3 ID3V2.3 結構定義

```
typedef struct TagID3V2.3_F
{     char   FrameID[4];                         //標籤框標識
      char   Size[4];                            //標籤內容大小
      char   Flags[2];
}     F_ID3V2.3;
```

圖 6-5 ID3V2.3 Frame 結構定義

```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  49 44 33 03 00 00 00 00  11 1C 54 49 54 32 00 00  ID3        TIT2
00000010  00 18 00 00 00 4B 6E 6F  77 69 6E 67 20 4D 65 2C       Knowing Me,
00000020  20 4B 6E 6F 77 69 6E 67  20 59 6F 75 54 41 4C 42   Knowing YouTALB
00000030  00 00 00 05 00 00 00 47  6F 6C 64 54 52 43 4B 00         GoldTRCK
00000040  00 00 05 00 00 00 32 2F  31 39 54 50 4F 53 00 00        2/19TPOS
00000050  00 04 00 00 00 31 2F 31  54 59 45 52 00 00 00 05        1/1TYER
00000060  00 00 00 31 39 37 37 54  43 4F 4E 00 00 00 05 00     1977TCON
00000070  00 00 28 31 33 29 54 45  4E 43 00 00 00 11 00 00    (13)TENC
00000080  00 69 54 75 6E 65 73 20  76 36 2E 30 2E 32 2E 32   iTunes v6.0.2.2
00000090  33 43 4F 4D 4D 00 00 00  5F 00 00 00 65 6E 67 00  3COMM    _   eng
000000A0  20 30 30 30 30 31 35 37  36 20 30 30 30 30 30 36   00001576 000006
000000B0  41 41 20 30 30 30 30 43  37 33 32 20 30 30 30 30  AA 0000C732 0000
000000C0  36 35 37 33 20 30 30 30  32 32 33 43 31 20 30 30  6573 000223C1 00
000000D0  30 32 36 36 42 31 20 30  30 30 30 38 32 37 38 20  0266B1 00008278
000000E0  30 30 30 30 38 34 34 30  20 30 30 30 33 31 42 31  00008440 00031B1
000000F0  41 20 30 30 30 31 34 32  41 37 43 4F 4D 4D 00 00  A 000142A7COMM
00000100  00 79 00 00 00 65 6E 67  00 20 30 30 30 30 30 30   y   eng  000000
00000110  30 30 20 30 30 30 30 30  32 31 30 20 30 30 30 30  00 00000210 0000
00000120  30 38 31 43 20 30 30 30  30 30 30 30 30 30 30 41  081C 000000000A
00000130  33 44 42 44 34 20 30 30  30 30 30 30 30 30 20 30  3DBD4 00000000 0
00000140  30 30 30 30 30 30 30 20  30 30 30 30 30 30 30 30  0000000 00000000
00000150  20 30 30 30 30 30 30 30  30 20 30 30 30 30 30 30   00000000 000000
00000160  30 30 20 30 30 30 30 30  30 30 30 20 30 30 30 30  00 00000000 0000
00000170  30 30 30 30 20 30 30 30  30 30 30 30 30 50 52 49  0000 00000000PRI
00000180  56 00 00 00 0E 00 00 50  65 61 6B 56 61 6C 75 65  V        PeakValue
00000190  00 FF 7F 00 00 50 52 49  56 00 00 00 11 00 00 41   ÿ▮  PRIV       A
000001A0  76 65 72 61 67 65 4C 65  76 65 6C 00 FC 19 00 00  verageLevel ü
000001B0  54 43 4F 4D 00 00 00 0E  00 00 00 53 74 69 67 20  TCOM        Stig
000001C0  41 6E 64 65 72 73 6F 6E  54 50 45 31 00 00 00 05  AndersonTPE1
000001D0  00 00 00 41 42 42 41 54  43 50 00 00 00 00 03 00     ABBATCP
000001E0  00 00 31 00 43 4F 4D 4D  00 00 00 40 00 00 00 65    1 COMM   @   e
000001F0  6E 67 69 54 75 6E 65 73  5F 43 44 44 42 5F 49 44  ngiTunes_CDDB_ID
00000200  73 00 31 39 2B 36 42 37  45 34 37 46 39 32 36 37  s 19+6B7E47F9267
00000210  38 41 35 44 33 36 46 34  34 36 44 45 42 33 43 38  8A5D36F446DEB3C8
00000220  37 44 34 37 37 2B 33 36  37 39 34 32 35 00 00 00  7D477+3679425
00000230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

圖 6-6 MP3 ID3V2.3 資料內容

6.2 MP3 音框(Frame)格式

MP3 檔案的音訊資料是由音框(簡稱 Frame)所組成的，每個 Frame 的大小會依照壓縮位元率而不同，圖 6-7 為 MP3 音框結構，每個 Frame 都是由 Header、CRC、Side Information 和 Main data 等四個部分所組成的，詳細的 MP3 規格請參考 ISO-ICE IS11172-3MPEG Information Technology [15]。

<table>
<tr><td rowspan="4">MP3
Audio Frame</td><td>Header (4 bytes)</td></tr>
<tr><td>CRC (0 or 2 bytes)</td></tr>
<tr><td>Side Information (17 or 32 bytes)</td></tr>
<tr><td>Main Data</td></tr>
</table>

圖 6-7 MP3 音框結構 [15]

框頭 (Header)

音框開始的 32 位元(4bytes)為框頭資訊，其數據格式如圖 6-8 所示，由 12 位元的同步字元(Syncword)開始。

<table>
<tr><th colspan="6">Header Information</th></tr>
<tr><td colspan="6">Sync word (8bits)</td></tr>
<tr><td colspan="2">Sync word (4 bits)</td><td>ID (1bit)</td><td colspan="2">Layer (2bits)</td><td>CRC (1bit)</td></tr>
<tr><td colspan="2">Bitrate index (4bits)</td><td colspan="2">Sample freq (2bits)</td><td>Padding bit (1bit)</td><td>Private (1bit)</td></tr>
<tr><td>Channel mode (2bits)</td><td>Mode extension (2bits)</td><td>Copyright (1bit)</td><td>Original (1bit)</td><td colspan="2">Emphasis (2bits)</td></tr>
</table>

圖 6-8 MP3 音框頭結構(32 位元) [15]

計算音框資料長度的算式為(40)，其由位元率(N_{BR})和取樣頻率(F_S)來決定。

$$N_S = N_{ver} \times N_{BR} / F_S + N_{pad} \tag{40}$$

其中 N_S 為音框容量，N_{BR} 為位元率，F_S 為取樣頻率，N_{pad} 為 Padding Bit，如果 MPEG 版本為 Layer I 則 N_{ver} 為 144，如為 Layer II 則為 72。

圖 6-9 為示範音樂檔案之音框資訊，其框頭資料為 0xFFFBA040，則其音框資訊內容為 Syncword = 0xFFF、ID = MPEG1、Layer = Layer 3、Bit rate = 160 kbpsSample rate = 44.1KHz、Padding bit = 0，計算出的音框資料長度為 FrameSize = (144 × 160000) / 44100 + 0 = 522 bytes。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
033B58A0	00	00	00	00	00	00	FF	FB	A0	40	00	00	03	46	8E	3F
033B58B0	81	01	3B	70	71	D1	27	F0	20	27	6E	4E	52	35	08	03
033B58C0	85	4D	C9	81	46	61	40	70	A9	B9	00	21	84	22	27	CA
033B58D0	22	27	FC	63	FC	6F	FF	02	E5	E3	7F	FC	63	19	9F	F7
033B58E0	B7	DB	E6	19	FF	FF	E9	E8	63	79	FF	FE	79	86	7F	FF
033B58F0	F5	3F	DB	FF	6B	FB	FF	31	FE	67	43	0C	33	57	A1	9F
033B5900	FD	5F	3C	F5	1B	93	3C	6E	EC	C7	9E	7B	9E	79	E3	73
033B5910	07	01	E1	62	80	F0	68	D4	08	00	60	38	1A	0E	1A	37
033B5920	71	BA	0D	04	40	78	34	20	64	7C	9A	18	34	30	CB	CC
033B5930	1A	10	30	00	86	10	28	62	7D	A2	27	FF	C6	F8	DF	F1
033B5940	8C	63	18	C7	F9	F5	18	DF	F8	0C	30	C6	CC	CC	6F	F3
033B5950	DE	7B	7F	BF	43	10	C6	FF	B7	CF	3D	FD	FA	1E	7B	FE
033B5960	F3	1B	ED	CF	3C	FA	99	CF	3C	F7	FD	0C	31	BF	9E	79
033B5970	86	18	66	79	E7	99	3C	F3	F4	E7	BF	F9	86	7A	18	63
033B5980	7A	B6	79	EA	E6	31	8A	79	C6	2B	98	60	8E	0F	C6	EE
033B5990	10	09	02	20	02	03	82	E3	46	A9	EF	A0	D0	1E	09	0C
033B59A0	00	19	D0	F0	F0	0D	22	6B	68	F7	64	DF	D9	7F	32	37
033B59B0	AD	1E	7A	20	E8	FC	AF	DF	DF	B6	BF	53	7D	E6	7F	EF
033B59C0	AF	54	A7	D6	9D	1F	74	DE	9B	3F	FF	E6	FA	F7	6F	FD
033B59D0	ED	DA	9A	FD	26	D7	B7	6A	3D	1B	BB	A3	E6	73	4A	10
033B59E0	B1	A4	46	A9	C3	34	30	A8	AE	71	99	1A	17	0B	C1	02
033B59F0	16	86	A2	24	64	22	07	AE	79	51	68	6A	2C	04	04	63
033B5A00	12	84	C3	84	43	D1	3C	90	F3	09	C2	E0	7A	3C	3D	45
033B5A10	72	52	33	54	A1	C5	E7	02	A4	6B	A5	DF	E5	DC	EF	2E
033B5A20	6B	D6	BE	5D	4E	9F	33	FB	F6	EC	9F	DF	ED	4E	DF	6B
033B5A30	EF	FF	AF	45	B3	53	FE	F7	F4	FE	FF	BF	FF	D1	7F	65
033B5A40	F5	EF	ED	EE	FD	6F	AF	FF	D3	BD	0C	30	F7	75	28	69
033B5A50	87	90	92	8A	E7	98	42	26	98	31	1F	9A	2A	38	F0	9C
033B5A60	1F	15	86	82	A8	5D	10	1A	61	0B	93	29	74	71	50	7A
033B5A70	4A	79	56	52	EE	5D	E9	00	05	7B	76	76	EF	31	24	53
033B5A80	90	02	7E	29	51	7A	47	2F	88	B9	85	96	04	33	25	65
033B5A90	B5	B4	39	4C	BF	B6	9A	52	69	FF	37	98	E8	B3	7D	AE
033B5AA0	6F	45	D9	D3	4A	9B	FC	D4	7A	19	DB	EE	DD	B6	44	5A
033B5AB0	FF	FB	A2	40	1C	00	04	23	8E	41	80	A1	53	72	8A	F1

圖 6-9 MP3 音框資料

旁資訊 (Side Information)

旁資訊主要的用途為記錄音訊解碼時所需要的狀態資訊，並依照聲道數的不同而有所差異，如為單雙聲道則有 17 bytes 的旁資訊，若為雙聲道則會增加為 34 bytes。將這些旁資訊解碼後，MP3 解碼器就可知道該音訊資料是使用那一個霍夫曼表來編碼的 [16]。

單聲道模式：136bits

Main data begin (9 bits)	Private bit (5 bits)	Scfsi[ch][scfsi_band] (4 bits)	Gr0 side information (59 bits)	Gr1 side information (59 bits)

雙聲道模式：256bits

Main data begin (9 bits)	Private bit (3 bits)	Scfsi[ch][scfsi_band] (4*2 = 8 bits)	Gr0 side information (59*2 =118 bits)	Gr1 side information (59*2 = 118 bits)

Gr side information

Part2_3 length (12 bits)	Big value (9 bits)	Global gain (8 bits)	Scalefac compress (4 bits)	Window switch flag (1 bit)

Window switch flag = True	Block type (2 bits)	Mixed block flag (1 bit)	Table select[region] (10 bits)	Subblock gain[window] (9 bits)
Window switch flag = False	Table select[region] (15 bits)		Region0 count (4 bits)	Region1 count (3 bits)

Preflag (1 bit)	Scalefac scale (1 bit)	Count table select (1 bit)

圖 6-10 MP3 旁資訊結構 [16]

主資料 (Main Data)

主資料是由 Granule 0 與 Granule 1 所組成的，而每一個 Granule 資料又分成左聲道與右聲道，每一個聲道內的資訊通過 MP3 解碼後，可以得到 1152 筆音訊資料。

Main Data			
Granule 0		Granule 1	
Left channel	Right channel	Left channel	Right channel

圖 6-11 MP3 主資料結構 [16]

6.3 MP3 解碼流程

MP3 解碼的第一個步驟為解碼串流位元資料，在讀取串流資料後首先找到 32 位元的 Frame Header 並將之解碼，接著再解碼 Side Informaton 得到霍夫曼表號和比例因數等資訊，再將 Main Date 根據霍夫曼表來進行解碼，可以得到 32 組子頻帶訊號，其中每個子頻帶可細分為 18 個次頻帶資料，此時共可得出 576 個頻域數據。

第二個步驟為反量化，經由反量化運算將經過霍夫曼解碼後的 576 個頻域數據乘上量化因數還原為原始的頻譜資料，再透過重新排序用以恢復霍夫曼編碼時打亂的次序，然後進行身歷聲處理和消除重疊等程序。

最後透過反離散餘弦轉換進行頻域轉時域的資料處理，由於 MP3 在進行壓縮音訊資料時，會先將原始聲音資料分成固定的區塊，然後做離散餘弦轉換，將每個區塊的值轉換為 32 組子頻帶訊號。因此必需經由反離散餘弦轉換(IMDCT)將每一組子頻帶的 18 個次頻帶資料作反余弦變換還原成原始的料時域信號，最後經由多相合成濾波器將得到的 32 組子頻帶訊號合成為 18 塊、每塊 32 筆 PCM 的聲音信號 [17]。

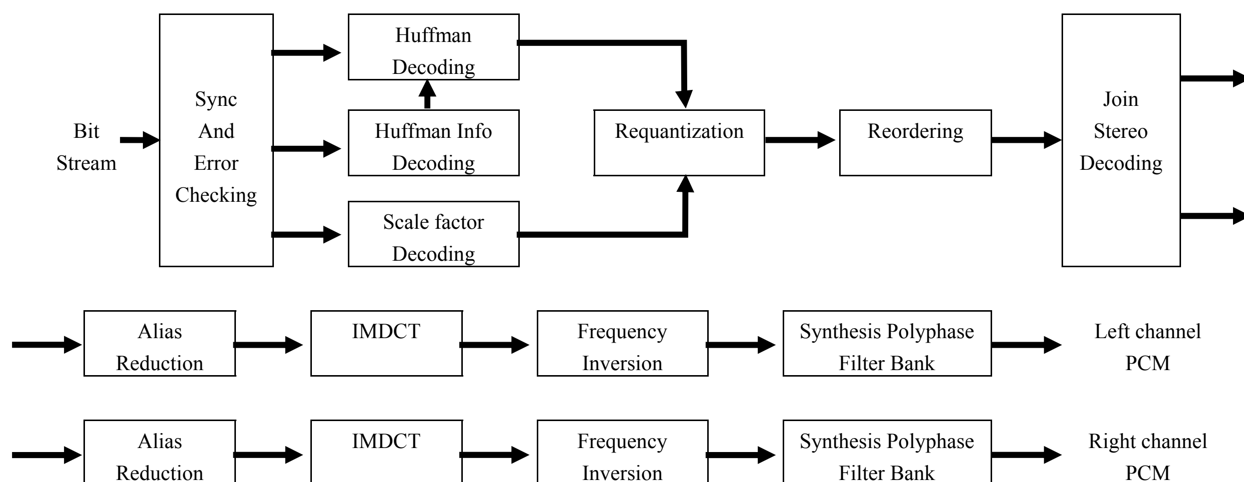


圖 6-12 MP3 解碼程序 [17]

6.4 MP3 解碼程序

本文主要是藉由 MP3 解碼器的使用，學習 ARM 微控器在播放多媒體應用上的整體流程，因此並不對 MP3 解碼器的工程原理進行深入探討。本系統所使用的 MP3 解碼程式是 Helix Community 公司提供的 Helic Player 11 Gold 原始碼(https://helixcommunity.org/downloads)。

圖 6-13 是 Helic Player 11 Gold 原始碼的 MP3 解碼器流程圖，圖 6-14 為其解碼範例程序。

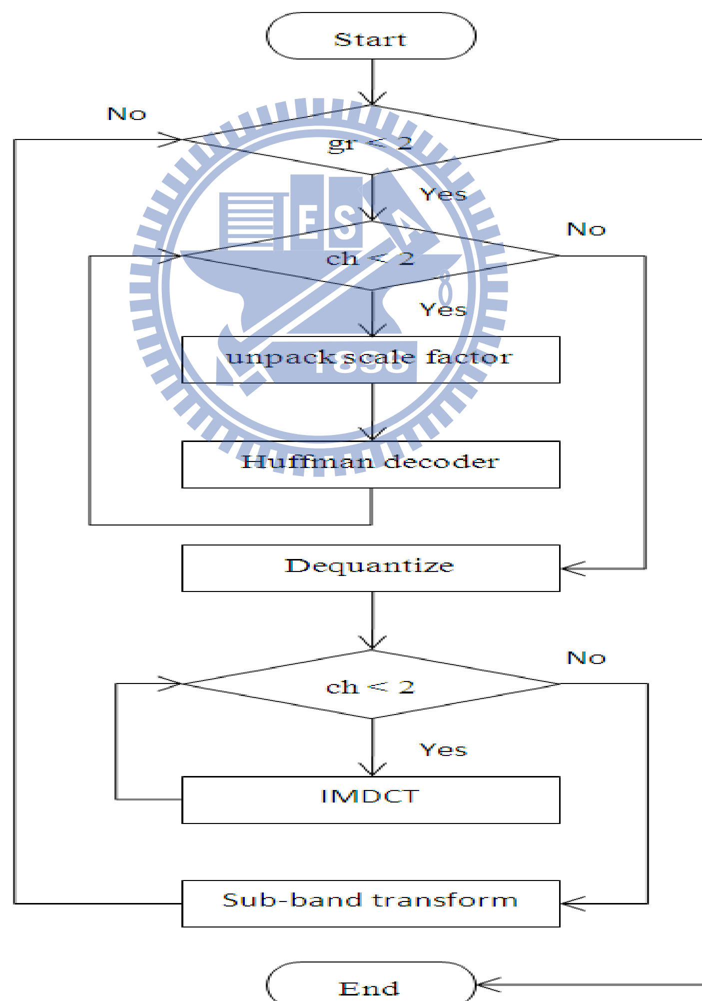


圖 6-13MP3 解碼流程

程式名稱：**MP3Decode ( )**	輸　入：**MP3Decoder,**
功能敘述：**MP3 解碼程序**	輸　出：***outbuf**

```
int MP3Decode(HMP3Decoder hMP3Decoder,     unsigned char **inbuf, int *bytesLeft,
                          short *outbuf, int useSize)
{
    int offset, bitOffset, mainBits, gr, ch, fhBytes, siBytes, freeFrameBytes;
    int prevBitOffset, sfBlockBits, huffBlockBits;
    unsigned char *mainPtr;
    MP3DecInfo *mp3DecInfo = (MP3DecInfo *)hMP3Decoder;

    if (!mp3DecInfo)
        return ERR_MP3_NULL_POINTER;

    // unpack frame header
    fhBytes = UnpackFrameHeader(mp3DecInfo, *inbuf);
    if (fhBytes < 0)
        // don't clear outbuf since we don't know size (failed to parse header)
        return ERR_MP3_INVALID_FRAMEHEADER;
    *inbuf += fhBytes;

    // unpack side info
    siBytes = UnpackSideInfo(mp3DecInfo, *inbuf);
    if (siBytes < 0) {
        MP3ClearBadFrame(mp3DecInfo, outbuf);
        return ERR_MP3_INVALID_SIDEINFO;
    }
    *inbuf += siBytes;
    *bytesLeft -= (fhBytes + siBytes);

    /* if free mode, need to calculate bitrate and nSlots manually, based on frame size */
    if (mp3DecInfo->bitrate == 0 || mp3DecInfo->freeBitrateFlag)
    {
        if (!mp3DecInfo->freeBitrateFlag)
        {
            /* first time through, need to scan for next sync word and figure out frame size */
            mp3DecInfo->freeBitrateFlag = 1;
            mp3DecInfo->freeBitrateSlots = MP3FindFreeSync(*inbuf, *inbuf - fhBytes - siBytes, *bytesLeft);
            if (mp3DecInfo->freeBitrateSlots < 0)
            {
                MP3ClearBadFrame(mp3DecInfo, outbuf);
                return ERR_MP3_FREE_BITRATE_SYNC;
            }
            freeFrameBytes = mp3DecInfo->freeBitrateSlots + fhBytes + siBytes;
            mp3DecInfo->bitrate = (freeFrameBytes * mp3DecInfo->samprate * 8) /
                                        (mp3DecInfo->nGrans * mp3DecInfo->nGranSamps);
        }
        // add pad byte, if required
        mp3DecInfo->nSlots = mp3DecInfo->freeBitrateSlots + CheckPadBit(mp3DecInfo);
```

```
}
/* useSize != 0 means we're getting reformatted (RTP) packets (see RFC 3119)
* -- calling function assembles "self-contained" MP3 frames by shifting any main_data
*     from the bit reservoir (in previous frames) to AFTER the sync word and side info
* -- calling function should set mainDataBegin to 0, and tell us exactly how large this
*     frame is (in bytesLeft)              */
if (useSize)
{
    mp3DecInfo->nSlots = *bytesLeft;
    if (mp3DecInfo->mainDataBegin != 0 || mp3DecInfo->nSlots <= 0)
    {
        // error - non self-contained frame, or missing frame (size <= 0), could do loss concealment here
        MP3ClearBadFrame(mp3DecInfo, outbuf);
        return ERR_MP3_INVALID_FRAMEHEADER;
    }
    / can operate in-place on reformatted frames
    mp3DecInfo->mainDataBytes = mp3DecInfo->nSlots;
    mainPtr = *inbuf;
    *inbuf += mp3DecInfo->nSlots;
    *bytesLeft -= (mp3DecInfo->nSlots);
}
else
{
    // out of data - assume last or truncated frame
    if (mp3DecInfo->nSlots > *bytesLeft)
    {
        MP3ClearBadFrame(mp3DecInfo, outbuf);
        return ERR_MP3_INDATA_UNDERFLOW;
    }
    // fill main data buffer with enough new data for this frame
    if (mp3DecInfo->mainDataBytes >= mp3DecInfo->mainDataBegin)
    {
        // adequate "old" main data available (i.e. bit reservoir)
        memmove(mp3DecInfo->mainBuf, mp3DecInfo->mainBuf + mp3DecInfo->mainDataBytes
                    - mp3DecInfo->mainDataBegin, mp3DecInfo->mainDataBegin);
        memcpy(mp3DecInfo->mainBuf + mp3DecInfo->mainDataBegin, *inbuf, mp3DecInfo->nSlots);
        mp3DecInfo->mainDataBytes = mp3DecInfo->mainDataBegin + mp3DecInfo->nSlots;
        *inbuf += mp3DecInfo->nSlots;
        *bytesLeft -= (mp3DecInfo->nSlots);
        mainPtr = mp3DecInfo->mainBuf;
    }
    else
    {
        // not enough data in bit reservoir from previous frames (perhaps starting in middle of file)
        memcpy(mp3DecInfo->mainBuf + mp3DecInfo->mainDataBytes, *inbuf, mp3DecInfo->nSlots);
        mp3DecInfo->mainDataBytes += mp3DecInfo->nSlots;
        *inbuf += mp3DecInfo->nSlots;
        *bytesLeft -= (mp3DecInfo->nSlots);
        MP3ClearBadFrame(mp3DecInfo, outbuf);
```

```
            return ERR_MP3_MAINDATA_UNDERFLOW;
        }
    }
    bitOffset = 0;
    mainBits = mp3DecInfo->mainDataBytes * 8;
    for (gr = 0; gr < mp3DecInfo->nGrans; gr++)
    {for (ch = 0; ch < mp3DecInfo->nChans; ch++)
        {   /* unpack scale factors and compute size of scale factor block */
            prevBitOffset = bitOffset;
            offset = UnpackScaleFactors(mp3DecInfo, mainPtr, &bitOffset, mainBits, gr, ch);
            sfBlockBits = 8*offset - prevBitOffset + bitOffset;
            huffBlockBits = mp3DecInfo->part23Length[gr][ch] - sfBlockBits;
            mainPtr += offset;
            mainBits -= sfBlockBits;
            if (offset < 0 || mainBits < huffBlockBits)
            {   MP3ClearBadFrame(mp3DecInfo, outbuf);
                return ERR_MP3_INVALID_SCALEFACT;
            }
            /* decode Huffman code words */
            prevBitOffset = bitOffset;
            offset = DecodeHuffman(mp3DecInfo, mainPtr, &bitOffset, huffBlockBits, gr, ch);
            if (offset < 0)
            {   MP3ClearBadFrame(mp3DecInfo, outbuf);
                return ERR_MP3_INVALID_HUFFCODES;
            }
            mainPtr += offset;
            mainBits -= (8*offset - prevBitOffset + bitOffset);
        }
        /* dequantize coefficients, decode stereo, reorder short blocks */
        if (Dequantize(mp3DecInfo, gr) < 0)
        {   MP3ClearBadFrame(mp3DecInfo, outbuf);
            return ERR_MP3_INVALID_DEQUANTIZE;
        }
        /* alias reduction, inverse MDCT, overlap-add, frequency inversion */
        for (ch = 0; ch < mp3DecInfo->nChans; ch++)
        {   if (IMDCT(mp3DecInfo, gr, ch) < 0)
            {   MP3ClearBadFrame(mp3DecInfo, outbuf);
                return ERR_MP3_INVALID_IMDCT;
            }
        }
        /* subband transform - if stereo, interleaves pcm LRLRLR */
        if (Subband    (mp3DecInfo, outbuf + gr*mp3DecInfo->nGranSamps*mp3DecInfo->nChans) < 0)
        {   MP3ClearBadFrame(mp3DecInfo, outbuf);
            return ERR_MP3_INVALID_SUBBAND;
        }
    }
}
```

圖 6-14MP3 解碼副程式

在執行 MP3 解碼時，我們可以針對每一段解碼流程進行分析，查看其所需的執行時間，以便分析每一個解碼流程所佔用的處理時間是否有可以進行優化的空間。表 6-2 為實際量測解碼一個音框所需要的時間與每一個流程所佔的比例。

表 6-2 MP3 播放各區塊的執行時間

功能	單次執行時間	執行次數	佔用比率
Unpack Frame Header ()	5.04us	1	0.02%
Unpack Side Information ()	76.4us	1	0.34%
Unpack Scale Factor ()	30us	4	0.54%
Huffman decoder ()	314us	4	5.66%
Dequantize ()	880us	2	7.93%
IMDCT ()	3.14ms	2	28.29%
Sub Band ()	5.88ms	2	52.97%
Other process	942.56us	1	4.25%
Total MP3 Decoder Process ()	**22.2ms**	**1**	**100%**

第七章　結果與展望

在剛開始研究嵌入式系統應用時，首先要熟悉開發環境與ARM微控器各個模組電路的功能與使用方式，在這部份我們使用PWM模式來取代DAC的應用，可以減少硬體與軟體的負擔。接著便開始研究SD memory card的規格與操作模式，一般使用者可以得到SD記憶卡協會所提供的簡易版本SD memory card規格書，在其中可以初步的了解到SD記憶卡的命令格式與傳輸協定，在本文的應用中，我們將SPI模式與SD Bus模式的底層程序建立起來，可減少針對記憶卡操作的程式開發時間。

當控制 SD 記憶卡的讀寫程序完成後，就需要在記憶卡上建立檔案管理系統，而 FAT 檔案系統是較為主流的檔案管理系統，可透過電腦來存取 SD 記憶卡，因此我們將 FAT 檔案系統修改為使用 SD 記憶卡為儲存裝置，可以容易的應用在攜帶式產品上。

在檔案管理系統整合完成後，再針對音樂檔案來進行處理，我們選擇 WAVE 與 MP3 兩種來進行播放，其中 WAVE 音樂是以 PCM 編碼，我們只要將 PCM 值轉換為相對應的 PWM 輸出即可。而 MP3 解碼是較複雜的整合，由於我們並不是著重於 MP3 解碼器的研究，所以我們利用現有的開放原始程式來整合到系統上。

在進行播放測試時，發現只有使用一個緩衝區來儲存資料時，聲音會有停頓的狀況，這是由於讀取 512 bytes 大約需要 1msec，所以會造成停頓的狀況，將緩衝區修改為兩個以上就可解決此一狀況。接著在播放較高採樣頻率的音樂時(高於 225500Hz)，也有發生停頓的狀況，這是因為播放時間小於讀取 SD 記憶卡的時間所造成的，使用 SD Bus 模式可以解決問題。

而在 MP3 解碼器整合完成後進行播放測試時，我們發現聲音停頓的狀況非常的嚴重，經過量測後發現問題出在 MP3 解碼器解碼一個音框的時間過於長，大約在 50msec 上下，這比每個音框只播放 26msec 的時間還要多出一倍的時間，因此造成音樂播放停頓的狀況。經過對整個程式流程的重新檢討與分析，發現由於產生 PWM 輸出的中斷副程式的頻率過高，會佔用大量的中斷處理時間，因此會影響到 MP3 解碼器的運算效率，在修改 PWM 輸出的程式流程後，即可順暢的播放 MP3 音樂。

透過研究多媒體系統的應用，來瞭解嵌入式系統的開發流程，因此從基礎的 ARM 微控器功能電路開始，一步步將所需的功能結合起來，也讓我們對於 ARM 微控器的能力與應用更為熟悉，由於最近幾年 USB 的普及以及從 Internet 下載音樂越來越方便，因此未來還可以整合 USB 系統與 Internet 系統，透過 USB 或是 Internet 來聽音樂更能夠符合多媒體系統的需求。

本系統的完整程式與硬體電路圖可於國立交通大學電子產業控制實驗室網頁中取得，其網址為 http://www.cn.nctu.edu.tw/faculty/sklin/。

參 考 文 獻

[1] "*Low-, medium- and high-density STM32F101xx, STM32F102xx and STM32F103xx advanced ARM-based 32-bit MCUs, RM0008 Referance manual Rev. 6*," STMicroelectronics, pp. 77-148, pp. 273-329, pp. 412-467, pp. 546-609, Sep. 2008.

[2] "*Cortex-M3 Technical Reference Manual*," ARM Lid., pp. 150-193, June 2008.

[3] Ned Mohan, Tore M. Undeland, and William P. Robbins, *Power Electronics: Converters, Applications and Design*, 2rd Edition, John Wiley & Sons, pp. 161-199, 1995.

[4] "*SD Specifications Part1 Physical Layer Simplified Specification Version 2.00*," SD Group (Panasonic, SanDisk, Toshiba) and SD Card Association, Sep. 2006.

[5] "*SanDisk Secure Digital Card Prodcut Manual Version 2.2*," SanDisk Corporation, Sep. 2004.

[6] "*SD Memory Card Specification Part 1 Physical Layer Specification Version 1.0*," SD Group (MEI, SanDisk, Toshiba), Mar. 2000.

[7] Allan Evans, "*Fat16 Interface for MSP430, Application Note,*," Dept. of Electrical and Computer Engineering, Michigan State University, 2004.

[8] "*Microsoft Extensible Firmware Initiative FAT32 File System Specification, FAT: General Overview of On-Disk Format Version 1.03*," Microsoft Corporation, Dec. 2000.

[9] "*SD Memory Card Specifications Part 2 File System specification Version 1.0*," SD Group (MEI, SanDisk, Toshiba), Feb. 2000.

[10] Mitesh Moonat, Jagadeesh Rayala and Aseem Vasudev, "*Running FAT16 File System and DOS Command on SHARC Processors (EE-329)*," Analog Devices Inc., Sep. 2007.

[11] "*Multimedia Programming Interface and Data Specifications 1.0*," IBM Corporation and Microsoft Corporation, pp. 56-65, Aug 1991.

[12] Heidi Breslauer, "*Microsoft Multimedia Standards Update – New Multimedia Data types and Data Techniques, Revision 3.0*," Microsoft Corporation, pp. 19-74, Apr. 1994.

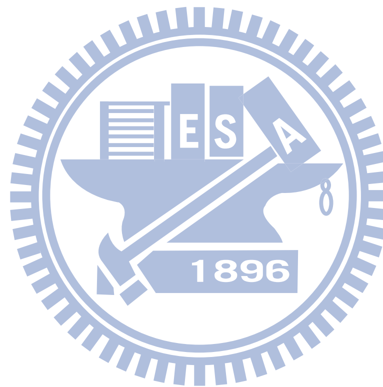
[13] “*Vocoder demonstration using a Speex audio codec on STM32F101xx and STM32F103xx microcontrollers,Rev. 2 AN2812 Application note*,” STMicro-electronics, Oct. 2008.

[14] 吳炳飛, *Audio Coding 技術手術, MP3 篇*, 第二版, 台北, 全華書局, 2007.

[15] ISO/IEC JTC1/SC29/WG11 MPEG, IS11172-3 “*Information Technology – Coding of Moving pictures and Associated Audio for Digital Storage Media at up to About 1.5MBit/s, Part 3 Audio*,” Nov. 1991.

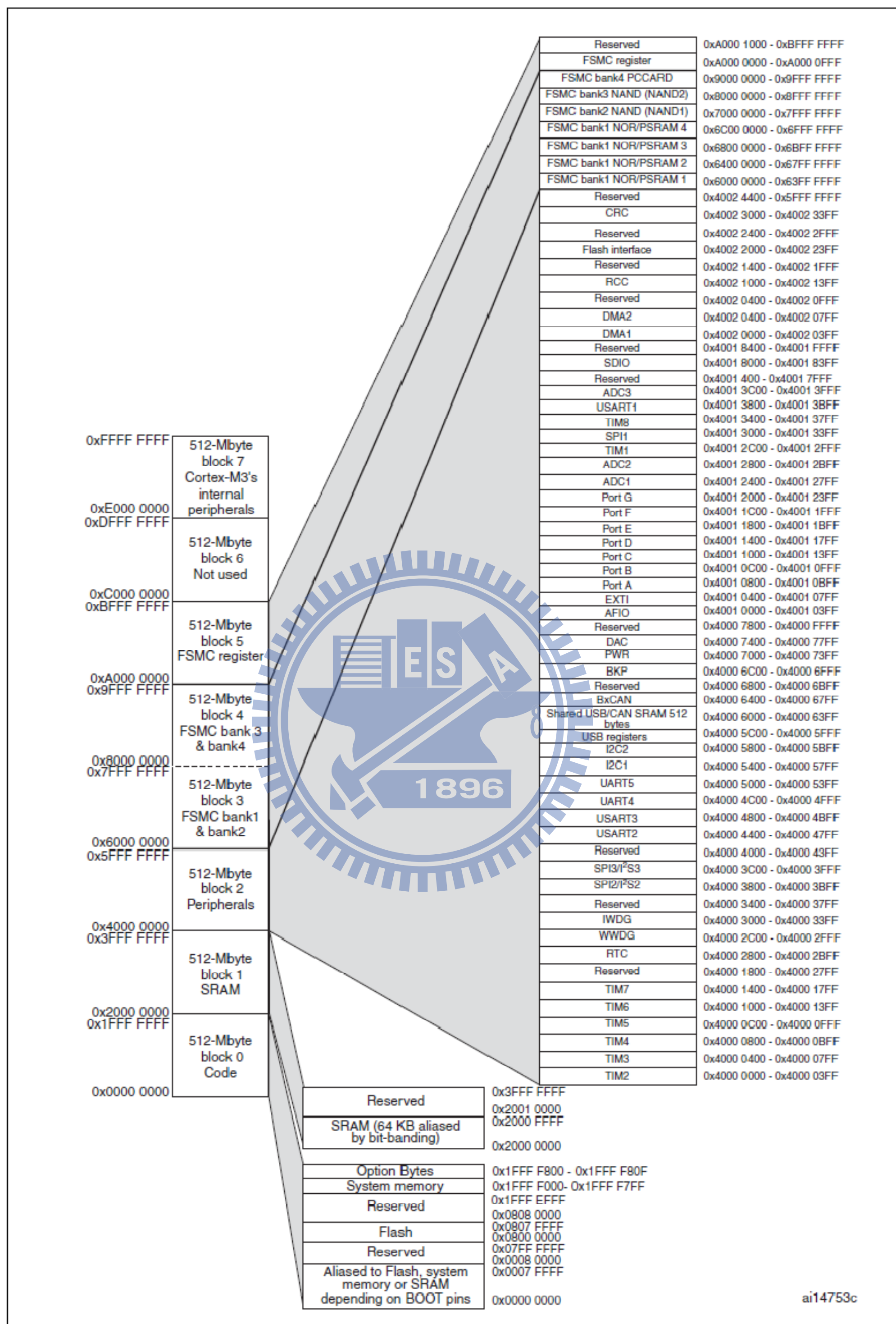
[16] 鄔文傑, “*MP3 與 MPEG-4 AAC 解碼器在 SCREAM DSP-16 上的實作與加速研究*,” 碩士論文, 資訊工程學系, 國立成功大學, pp. 12-35, Jul. 2007.

[17] Jianwei Wang, “*Hardware/software Codesign of MP3 Decoder with 36/32-point (I)DCT Accelerators*,” MSc.Thesis, Department of Electrical Engineering, Technische University Delft, pp. 15-32, July 2005.



附　錄　一

STM32F103x 記憶體映射



附　錄　二

SD 記憶卡命令描述

Basic commands (class 0)

CMD	type	argument	resp	Abbreviation	Command description
CMD0	bc	[31:0] stuff bits		GO_IDLE_STATE	Resets all cards to idle state
CMD1		reserved			
CMD2	bcr	[31:0] stuff bits	R2	ALL_SEND_CID	Asks any card to send the CID numbers on the CMD line (any card that is connected to the host will respond)
CMD3	Bcr	[31:0] stuff bits	R6	SEND_RELATIVE_ADDR	Ask the card to publish a new relative address (RCA)
CMD4	bc	[31:16] DSR [15:0] stuff bits		SET_DSR	Programs the DSR of all cards
CMD5		reserved			
CMD7	ac	[31:16] RCA [15:0] stuff bits	R1b	SELECT /DESELECT_ CARD	Command toggles a card between the stand-by and transfer states or between the programming and disconnect states. In both cases, the card is selected by its own relative address and gets deselected by any other address; address 0 deselects all. In the case that the RCA equals 0, then the host may do one of the following: - Use other RCA number to perform card de-selection. - Re-send CMD3 to change its RCA number to other than 0 and then use CMD7 with RCA=0 for card deselection.
CMD8	Bcr	[31:12]reserved bits [11:8]supply voltage(VHS) [7:0]check pattern	R7	SEND_IF_COND	Sends SD Memory Card interface condition, which includes host supply voltage information and asks the card whether card supports voltage. Reserved bits shall be set to '0'.
CMD9	ac	[31:16] RCA [15:0] stuff bits	R2	SEND_CSD	Addressed card sends its card-specific data (CSD) on the CMD line.
CMD10	ac	[31:16] RCA [15:0] stuff bits	R2	SEND_CID	Addressed card sends its card identification (CID) on CMD the line.
CMD11		reserved			
CMD12	ac	31:0] stuff bits	R1b	STOP_TRANSMISSION	Forces the card to stop transmission
CMD13	ac	[31:16] RCA [15:0] stuff bits	R1	SEND_STATUS	Addressed card sends its status register.
CMD14		reserved			
CMD15	ac	31:16] RCA [15:0] reserved bits		GO_INACTIVE_ STATE	Sends an addressed card into the *Inactive State*. This command is used when the host explicitly wants to deactivate a card. Reserved bits shall be set to '0'.

Block-oriented read commands (class 2)

CMD	type	argument	resp	Abbreviation	Command description
CMD16	ac	[31:0] block length	R1	SET_BLOCKLEN	In the case of a Standard Capacity SD Memory Card, this command sets the block length (in bytes) for all following block commands (read, write, lock). Default block length is fixed to 512 Bytes. Set length is valid for memory access commands only if partial block read operation are allowed in CSD. In the case of a High Capacity SD Memory Card, block length set by CMD16 command does not affect the memory read and write commands. Always 512 Bytes fixed block length is used. This command is effective for LOCK_UNLOCK

					command. In both cases, if block length is set larger than 512Bytes, the card sets the BLOCK_LEN_ERROR bit.
CMD17	adtc	[31:0] data address2	R1	READ_SINGLE_BLOCK	In the case of a Standard Capacity SD Memory Card, this command, this command reads a block of the size selected by the SET_BLOCKLEN command. 1 In the case of a High Capacity Card, block length is fixed 512 Bytes regardless of the SET_BLOCKLEN command.
CMD18	adtc	[31:0] data address	R1	READ_MULTIPLE_BLOCK	Continuously transfers data blocks from card to host until interrupted by a STOP_TRANSMISSION command. Block length is specified the same as READ_SINGLE_BLOCK command.

Block-oriented write commands (class 4)

CMD	type	argument	resp	Abbreviation	Command description
CMD16	ac	31:0] block length	R1	SET_BLOCKLEN	
CMD24	adtc	[31:0] data address	R1	WRITE_BLOCK	In the case of a Standard Capacity SD Memory Card, this command writes a block of the size selected by the SET_BLOCKLEN command. 1 In the case of a High Capacity Card, block length is fixed 512 Bytes regardless of the SET_BLOCKLEN command.
CMD25	adtc	[31:0] data address	R1	WRITE_MULTIPLE_BLOCK	Continuously writes blocks of data until a STOP_TRANSMISSION follows. Block length is specified the same as WRITE_BLOCK command.
CMD27	adtc	31:0] stuff bits	R1	PROGRAM_CSD	Programming of the programmable bits of the CSD.

Block-oriented write protection commands (class 6)

CMD	type	argument	resp	Abbreviation	Command description
CMD28	ac	[31:0] data address	R1	SET_WRITE_PROT	If the card has write protection features, this command sets the write protection bit of the addressed group. The properties of write protection are coded in the card specific data (WP_GRP_SIZE). A High Capacity SD Memory Card does not support this command.
CMD29	ac	[31:0] data address	R1	CLR_WRITE_PROT	If the card provides write protection features, this command clears the write protection bit of the addressed group. A High Capacity SD Memory Card does not support this Command.
CMD30	ac	[31:0] write protect data address	R1b	SEND_WRITE_PROT	If the card provides write protection features, this command asks the card to send the status of the write protection bits.1 A High Capacity SD Memory Card does not support this command.

Lock card (class 7)

CMD	type	argument	resp	Abbreviation	Command description
CMD16	ac	[31:0] block length	R1	SET_BLOCKLEN	
CMD42	adtc	[31:0] Reserved bits (Set all 0)	R1	LOCK_UNLOCK	Used to set/reset the password or lock/unlock the card. The size of the data block is set by the SET_BLOCK_LEN command. Reserved bits in the argument and in Lock Card Data Structure shall be set to 0.

Application specific commands (class 8)

CMD	type	argument	resp	Abbreviation	Command description
CMD55	ac	31:16] RCA [15:0] stuff bits	R1	APP_CMD	Indicates to the card that the next command is an application specific command rather than a standard command
CMD56	adtc	[31:1] stuff bits. [0]: RD/WR	R1	GEN_CMD	Used either to transfer a data block to the card or to get a data block from the card for general purpose/application specific commands. In the case of a Standard Capacity SD Memory Cards, the size of the data block shall be set by the SET_BLOCK_LEN command. In the case of a High Capacity SD Memory Cards, the size of the data block is fixed to 512 byte. The host sets RD/WR=1 for reading data from the card and sets to 0 for writing data to the card.

I/O mode commands (class 9)

CMD	type	argument	resp	Abbreviation	Command description
CMD52 .. CMD54	ac	reserved for I/O mode (refer to the "SDIO Card Specification")			

Switch function commands (class 10)

CMD	type	argument	resp	Abbreviation	Command description
CMD6	adtc	[31] Mode 0:Check function 1:Switch function [30:24] reserved (All ’0’) [23:20] reserved for function group 6 (0h or Fh) [19:16] reserved for function group 5 (0h or Fh) [15:12] reserved for function group 4 (0h or Fh) [11:8] reserved for function group 3 (0h or Fh) [7:4] function group 2 for command system [3:0] function group 1 for access mode	R1	SWITCH_FUNC	Checks switchable function (mode 0) and switch card function (mode 1). See Chapter 4.3.10.

Application specific commands used/reserved by SD memory card

CMD	type	argument	resp	Abbreviation	Command description
ACMD6	ac	31:2] stuff bits [1:0]bus width	R1	SET_BUS_WIDTH	Defines the data bus width (’00’=1bit or ’10’=4 bits bus) to be used for data transfer. The allowed data bus widths are given in SCR register.
ACMD13	adtc	[31:0] stuff bits	R1	SD_STATUS	Send the SD Status.
ACMD22	adtc	[31:0] stuff bits	R1		Send the number of the written (without errors) write blocks. Responds with 32bit+CRC data block. If WRITE_BL_PARTIAL='0', the unit of ACMD22 is always 512 byte. If WRITE_BL_PARTIAL='1', the unit of ACMD22 is a block length which was used when the write command was executed.
ACMD23	ac	[31:23] stuff bits [22:0]Number of blocks	R1	SEND_NUM_WR_ BLOCKS	Set the number of write blocks to be preerased before writing (to be used for faster Multiple Block WR command). “1”=default (one wr block) 2.
ACMD41	bcr	31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] VDD Voltage Window(OCR[23:0])	R3	SD_SEND_OP_COND	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to ‘0’. CCS bit is assigned to OCR[30].
ACMD42	ac	[31:1] stuff bits [0]set_cd	R1	SET_CLR_CARD_ DETECT	Connect[1]/Disconnect[0] the 50 KOhm pull-up resistor on CD/DAT3 (pin 1) of the card.
ACMD51	adtc	31:0] stuff bits	R1	SEND_SCR	Reads the SD Configuration Register (SCR).

附　錄　三

FAT 磁區內容與結構

FAT boot sector and BPB structure

Name	Offset (byte)	Size (bytes)	Description
BS_jmpBoot	0	3	Jump instruction to boot code. This field has two allowed forms: **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90** and **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??** **0x??** indicates that any 8-bit value is allowed in that byte. What this forms is a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code. This code typically occupies the rest of sector 0 of the volume following the BPB and possibly other sectors. Either of these forms is acceptable. **JmpBoot[0] = 0xEB** is the more frequently used format.
BS_OEMName	3	8	"MSWIN4.1" There are many misconceptions about this field. It is only a name string. Microsoft operating systems don't pay any attention to this field. Some FAT drivers do.
BPB_BytsPerSec	11	2	Count of bytes per sector. This value may take on only the following values: 512, 1024, 2048 or 4096. If maximum compatibility with old implementations is desired, only the value 512 should be used. There is a lot of FAT code in the world that is basically "hard wired" to 512 bytes per sector and doesn't bother to check this field to make sure it is 512. Microsoft operating systems will properly support 1024, 2048, and 4096
BPB_SecPerClus	13	1	Number of sectors per allocation unit. This value must be a power of 2 that is greater than 0. The legal values are 1, 2, 4, 8, 16, 32, 64, and 128. Note however, that a value should never be used that results in a "bytes per cluster" value (BPB_BytsPerSec * BPB_SecPerClus) greater than 32K (32 * 1024).
BPB_RsvdSecCnt	14	2	Number of reserved sectors in the Reserved region of the volume starting at the first sector of the volume. This field must not be 0. For FAT12 and FAT16 volumes, this value should never be anything other than 1. For FAT32 volumes, this value is typically 32.
BPB_NumFATs	16	1	The count of FAT data structures on the volume. This field should always contain the value 2 for any FAT volume of any type. Although any value greater than or equal to 1 is perfectly valid, many software programs and a few operating systems' FAT file system drivers may not function properly if the value is something other than 2. All Microsoft file system drivers will support a value other than 2, but it is still highly recommended that no value other than 2 be used in this field.
BPB_RootEntCnt	17	2	For FAT12 and FAT16 volumes, this field contains the count of 32-byte directory entries in the root directory. For FAT32 volumes, this field must be set to 0. For FAT12 and FAT16 volumes, this value should always specify a count that when multiplied by 32 results in an even multiple of BPB_BytsPerSec. For maximum compatibility, FAT16 volumes should use the value 512.
BPB_TotSec16	19	2	This field is the old 16-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then BPB_TotSec32 must be non-zero. For FAT32 volumes, this field must be 0. For FAT12 and FAT16 volumes, this field contains the sector count, and BPB_TotSec32 is 0 if the total sector count "fits" (is less than 0x10000).
BPB_Media	21	1	0xF8 is the standard value for "fixed" (non-removable) media. For removable media, 0xF0 is frequently used. The legal values for this field are 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, and 0xFF.
BPB_FATSz16	22	2	This field is the FAT12/FAT16 16-bit count of sectors occupied by ONE FAT. On FAT32 volumes this field must be 0, and BPB_FATSz32 contains the FAT size count.
BPB_SecPerTrk	24	2	Sectors per track for interrupt 0x13. This field is only relevant for media that have a geometry (volume is broken down into tracks by multiple heads and cylinders) and are visible on interrupt 0x13. This field contains the "sectors per track" geometry value.
BPB_NumHeads	26	2	Number of heads for interrupt 0x13. This field is relevant as discussed earlier for BPB_SecPerTrk. This field contains the one based "count of heads". For example, on a 1.44 MB 3.5-inch floppy drive this value is 2.

BPB_HiddSec	28	4	Count of hidden sectors preceding the partition that contains this FAT volume. This field is generally only relevant for media visible on interrupt 0x13. This field should always be zero on media that are not partitioned. Exactly what value is appropriate is operating system specific.
BPB_TotSec32	32	4	This field is the new 32-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then BPB_TotSec16 must be non-zero. For FAT32 volumes, this field must be non-zero. For FAT12/FAT16 volumes, this field contains the sector count if BPB_TotSec16 is 0 (count is greater than or equal to 0x10000).
BS_DrvNum	36	1	Int 0x13 drive number (e.g. 0x80). This field supports MS-DOS bootstrap and is set to the INT 0x13 drive number of the media (0x00 for floppy disks, 0x80 for hard disks).
BS_Reserved1	37	1	Reserved (used by Windows NT). Code that formats FAT volumes should always set this byte to 0.
BS_BootSig	38	1	Extended boot signature (0x29). This is a signature byte that indicates that the following three fields in the boot sector are present.
BS_VolID	39	4	Volume serial number. This field, together with BS_VolLab, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID is usually generated by simply combining the current date and time into a 32-bit value.
BS_VolLab	43	11	Volume label. This field matches the 11-byte volume label recorded in the root directory. FAT file system drivers should make sure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string "`NO NAME`".
S_FilSysType	54	8	One of the strings "`FAT12   `", "`FAT16   `", or "`FAT     `". NOTE: Many people think that the string in this field has something to do with the determination of what type of FAT—FAT12, FAT16, or FAT32—that the volume has. This is not true. You will note from its name that this field is not actually part of the BPB. This string is informational only and is not used by Microsoft file system drivers to determine FAT type because it is frequently not set correctly or is not present. See the FAT Type Determination section of this document. This string should be set based on the FAT type though, because some non-Microsoft FAT file system drivers do look at it.

FAT 32 bytes directory entry structure

Name	Offset (byte)	Size (bytes)	Description
DIR_Name	0	11	Short name.
DIR_Attr	11	1	File attributes: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID The upper two bits of the attribute byte are reserved and should always be set to 0 when a file is created and never modified or looked at after that.
DIR_NTRes	12	1	Reserved for use by Windows NT. Set value to 0 when a file is created and never modify or look at it after that.
DIR_CrtTimeTenth	13	1	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second. The granularity of the seconds part of DIR_CrtTime is 2 seconds so this field is a count of tenths of a second and its valid value range is 0-199 inclusive.
DIR_CrtTime	14	2	Time file was created.
DIR_CrtDate	14	2	Date file was created.
DIR_LstAccDate	18	2	Last access date. Note that there is no last access time, only a date. This is the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.
DIR_FstClusHI	20	2	High word of this entry's first cluster number (always 0 for a FAT12 or FAT16 volume).
DIR_WrtTime	22	2	Time of last write. Note that file creation is considered a write.
DIR_WrtDate	24	2	Date of last write. Note that file creation is considered a write.
DIR_FstClusLO	26	2	Low word of this entry's first cluster number.
DIR_FileSize	28	4	32-bit DWORD holding this file's size in bytes.

附　錄　四

MP3 音框檔頭格式

<table>
<tr><th rowspan="2">Bytes</th><th colspan="8">Bits</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="8">A</td></tr>
<tr><td>1</td><td colspan="3">A</td><td colspan="2">B</td><td colspan="2">C</td><td>D</td></tr>
<tr><td>2</td><td colspan="4">E</td><td colspan="2">F</td><td>G</td><td>H</td></tr>
<tr><td>3</td><td colspan="2">I</td><td colspan="2">J</td><td>K</td><td>L</td><td colspan="2">M</td></tr>
</table>

<table>
<tr><th>Sign</th><th>Length</th><th>Description</th></tr>
<tr><td>A</td><td>11</td><td>Frame sync (all bits set)</td></tr>
<tr><td>B</td><td>2</td><td>MPEG Audio version
00 - MPEG Version 2.5　01 – reserved　10 - MPEG Version 2　11 - MPEG Version 1</td></tr>
<tr><td>C</td><td>2</td><td>Layer description
00 - reserved　01 - Layer III　10 - Layer II　11 - Layer I</td></tr>
<tr><td>D</td><td>1</td><td>Protection bit
0 - Protected by CRC (16bit crc follows header)　1 - Not protected</td></tr>
<tr><td>E</td><td>4</td><td>Bitrate index
<table>
<tr><th>Bits</th><th>V1, L1</th><th>V1, L2</th><th>V1, L3</th><th>V2, L1</th><th>V2, L2</th><th>V2, L3</th></tr>
<tr><td>0000</td><td>free</td><td>free</td><td>free</td><td>free</td><td>free</td><td>free</td></tr>
<tr><td>0001</td><td>32</td><td>32</td><td>32</td><td>32</td><td>32</td><td>8 (8)</td></tr>
<tr><td>0010</td><td>64</td><td>48</td><td>40</td><td>64</td><td>48</td><td>16 (16)</td></tr>
<tr><td>0011</td><td>96</td><td>56</td><td>48</td><td>96</td><td>56</td><td>24 (24)</td></tr>
<tr><td>0100</td><td>128</td><td>64</td><td>56</td><td>128</td><td>64</td><td>32 (32)</td></tr>
<tr><td>0101</td><td>160</td><td>80</td><td>64</td><td>160</td><td>80</td><td>64 (40)</td></tr>
<tr><td>0110</td><td>192</td><td>96</td><td>80</td><td>192</td><td>96</td><td>80 (48)</td></tr>
<tr><td>0111</td><td>224</td><td>112</td><td>96</td><td>224</td><td>112</td><td>56 (56)</td></tr>
<tr><td>1000</td><td>256</td><td>128</td><td>112</td><td>256</td><td>128</td><td>64 (64)</td></tr>
<tr><td>1001</td><td>288</td><td>160</td><td>128</td><td>288</td><td>160</td><td>128 (80)</td></tr>
<tr><td>1010</td><td>320</td><td>192</td><td>160</td><td>320</td><td>192</td><td>160 (96)</td></tr>
<tr><td>1011</td><td>352</td><td>224</td><td>192</td><td>352</td><td>224</td><td>112 (112)</td></tr>
<tr><td>1100</td><td>384</td><td>256</td><td>224</td><td>384</td><td>256</td><td>128 (128)</td></tr>
<tr><td>1101</td><td>416</td><td>320</td><td>256</td><td>416</td><td>320</td><td>256 (144)</td></tr>
<tr><td>1110</td><td>448</td><td>384</td><td>320</td><td>448</td><td>384</td><td>320 (160)</td></tr>
<tr><td>1111</td><td>bad</td><td>bad</td><td>bad</td><td>bad</td><td>bad</td><td>bad</td></tr>
</table>
NOTES: All values are in kbps
V1 - MPEG Version 1　V2 - MPEG Version 2 and Version 2.5
L1 - Layer I　L2 - Layer II　L3 - Layer III
"free" means variable bitrate.
"bad" means that this is not an allowed value</td></tr>
<tr><td>F</td><td>2</td><td>Sampling rate frequency index (values are in Hz)
<table>
<tr><th>Bits</th><th>MPEG1</th><th>MPEG2</th><th>MPEG2.5</th></tr>
<tr><td>00</td><td>44100</td><td>22050</td><td>11025</td></tr>
<tr><td>01</td><td>38000</td><td>24000</td><td>12000</td></tr>
<tr><td>10</td><td>32000</td><td>16000</td><td>8000</td></tr>
<tr><td>11</td><td>Reserved</td><td>Reserved</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>G</td><td>1</td><td>Padding bit
0 - frame is not padded　1 - frame is padded with one extra bit</td></tr>
<tr><td>H</td><td>1</td><td>Private bit (unknown purpose)</td></tr>
<tr><td>I</td><td>2</td><td>Channel Mode
00 - Stereo　01 - Joint stereo (Stereo)
10 - Dual channel (Stereo)　11 - Single channel (Mono)</td></tr>
<tr><td>J</td><td>2</td><td>Mode extension (Only if Joint stereo)
<table>
<tr><th>Value</th><th>Intensity Stereo</th><th>MS Stereo</th></tr>
<tr><td>00</td><td>Off</td><td>Off</td></tr>
<tr><td>01</td><td>On</td><td>Off</td></tr>
<tr><td>10</td><td>Off</td><td>On</td></tr>
<tr><td>11</td><td>On</td><td>On</td></tr>
</table></td></tr>
<tr><td>K</td><td>1</td><td>Copyright
0 - Audio is not copyrighted　1 - Audio is copyrighted</td></tr>
<tr><td>L</td><td>1</td><td>Original
0 - Copy of original media　1 - Original media</td></tr>
<tr><td>M</td><td>2</td><td>Emphasis
00 - none　01 - 50/15 ms
10 – reserved　11 - CCIT J.17</td></tr>
</table>

附　錄　五

MP3 播放器電路與 PCB 佈線圖

圖 1

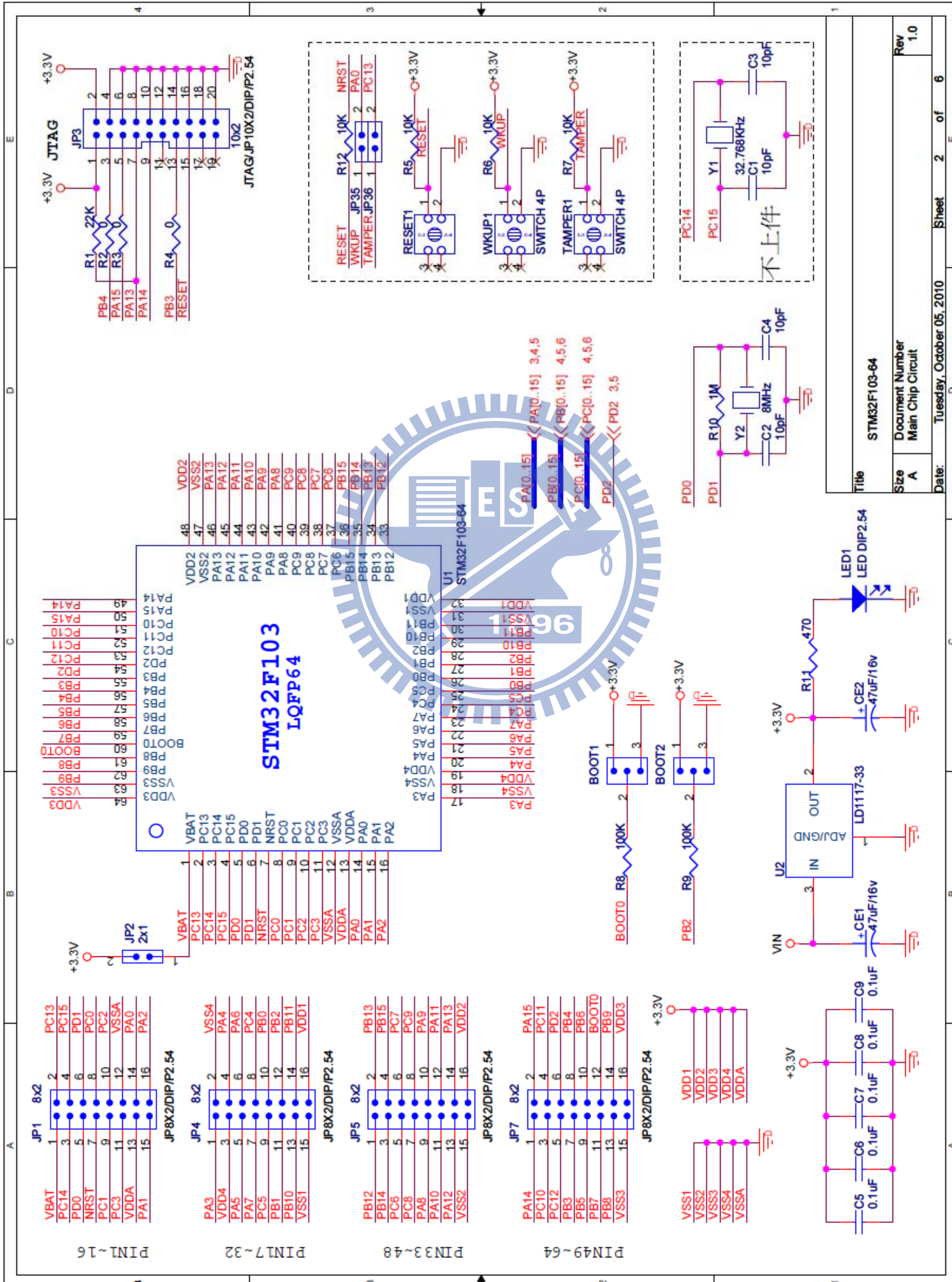


圖 2

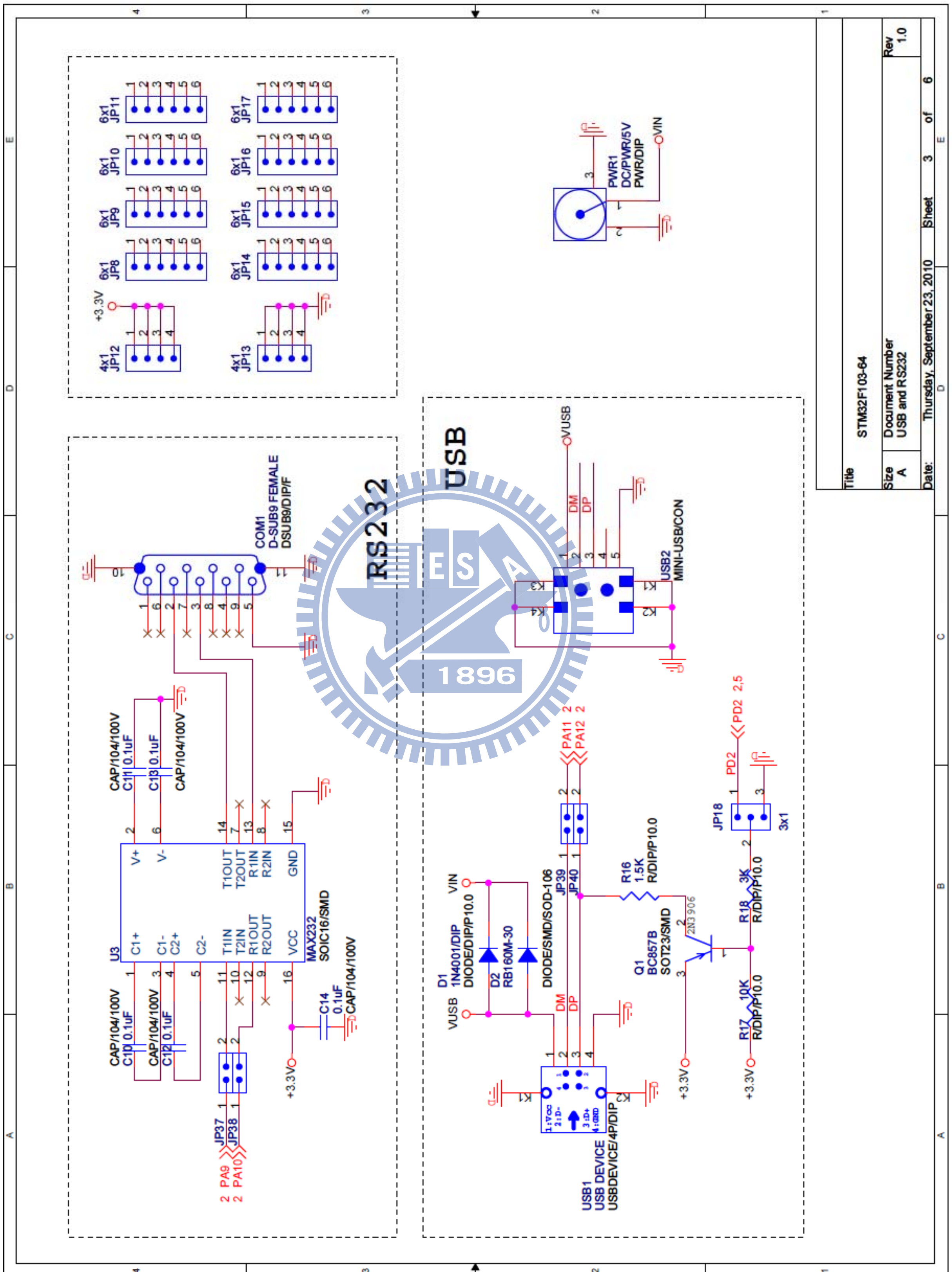


圖 3

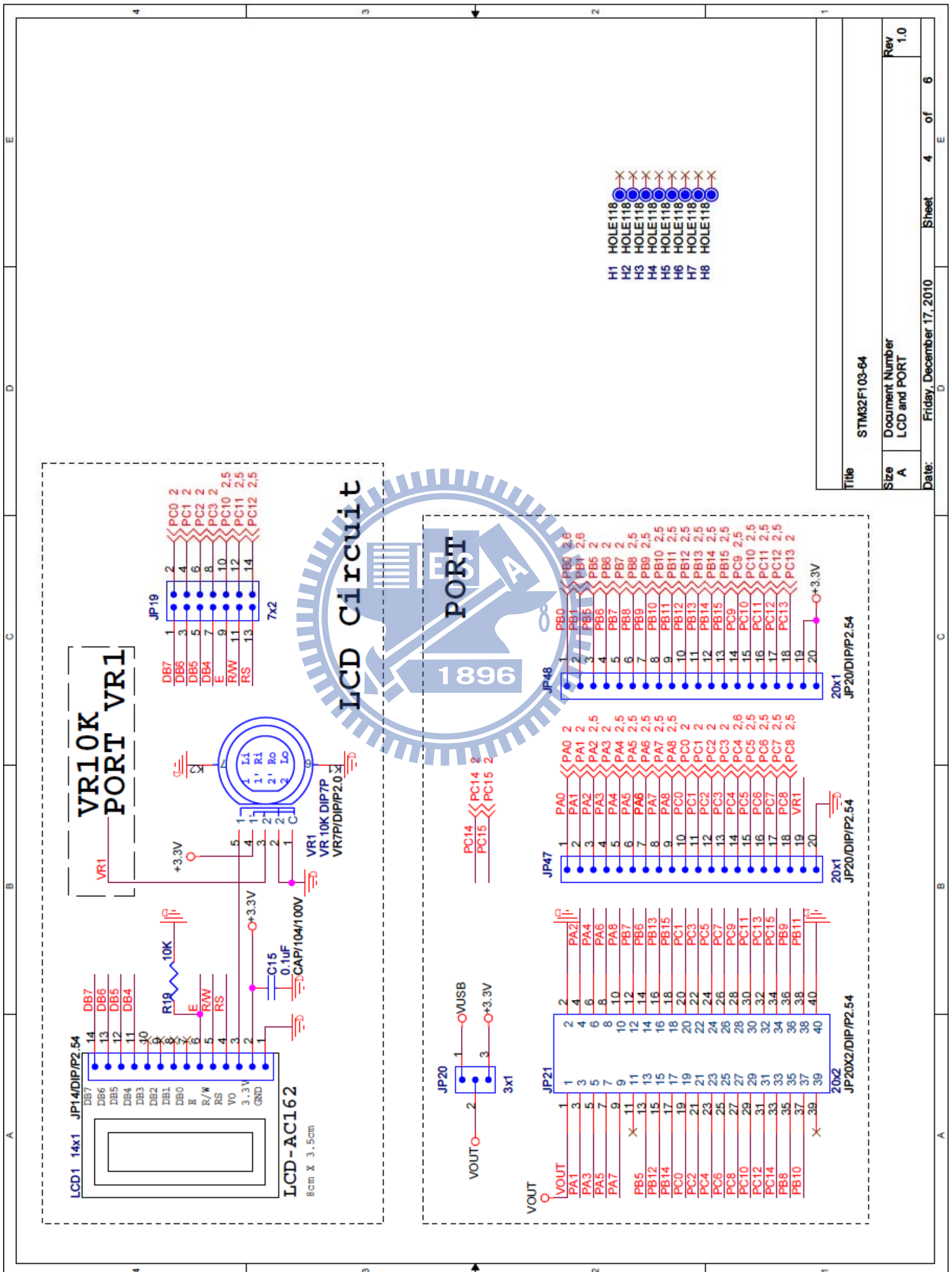


圖 4

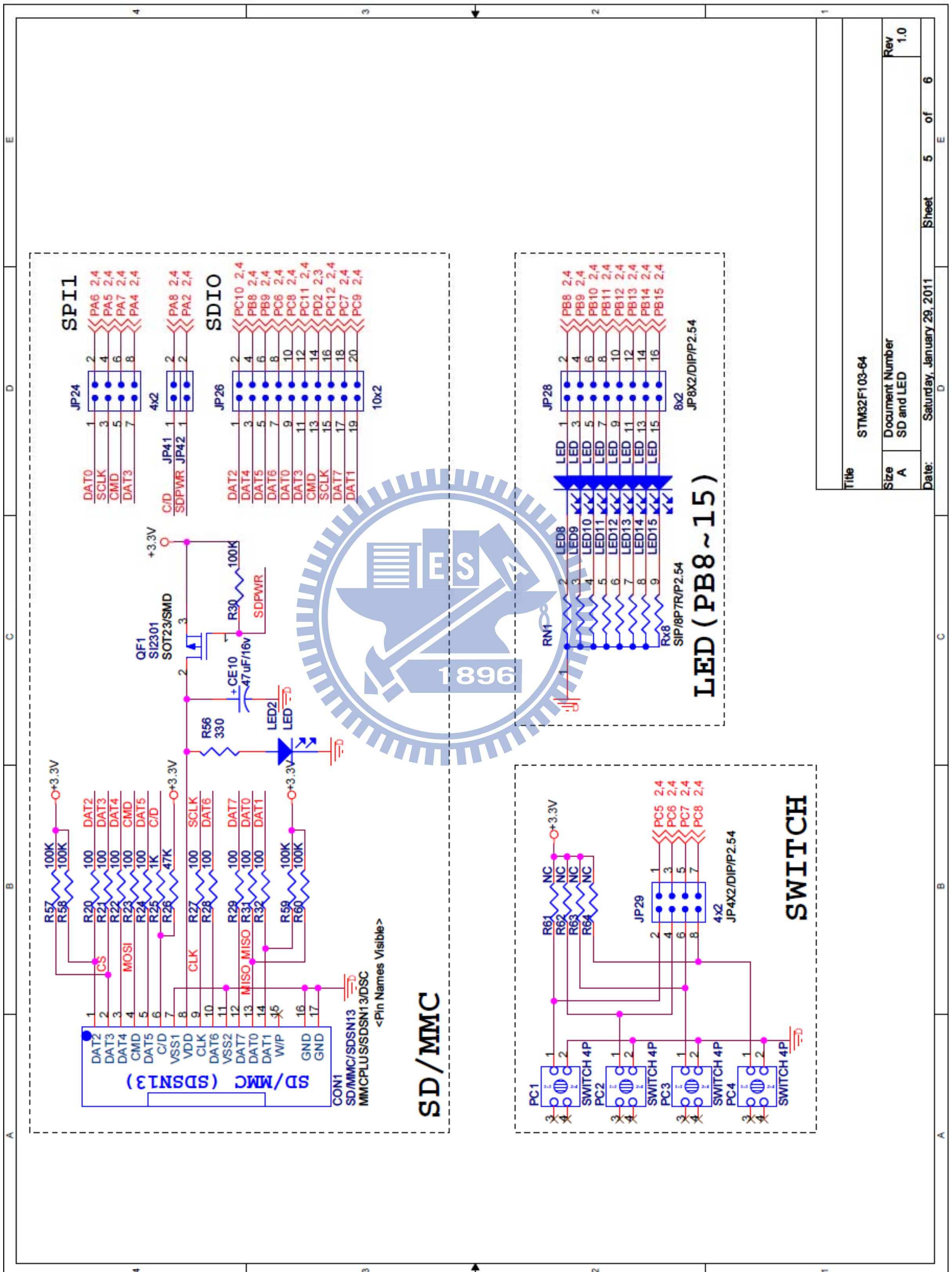
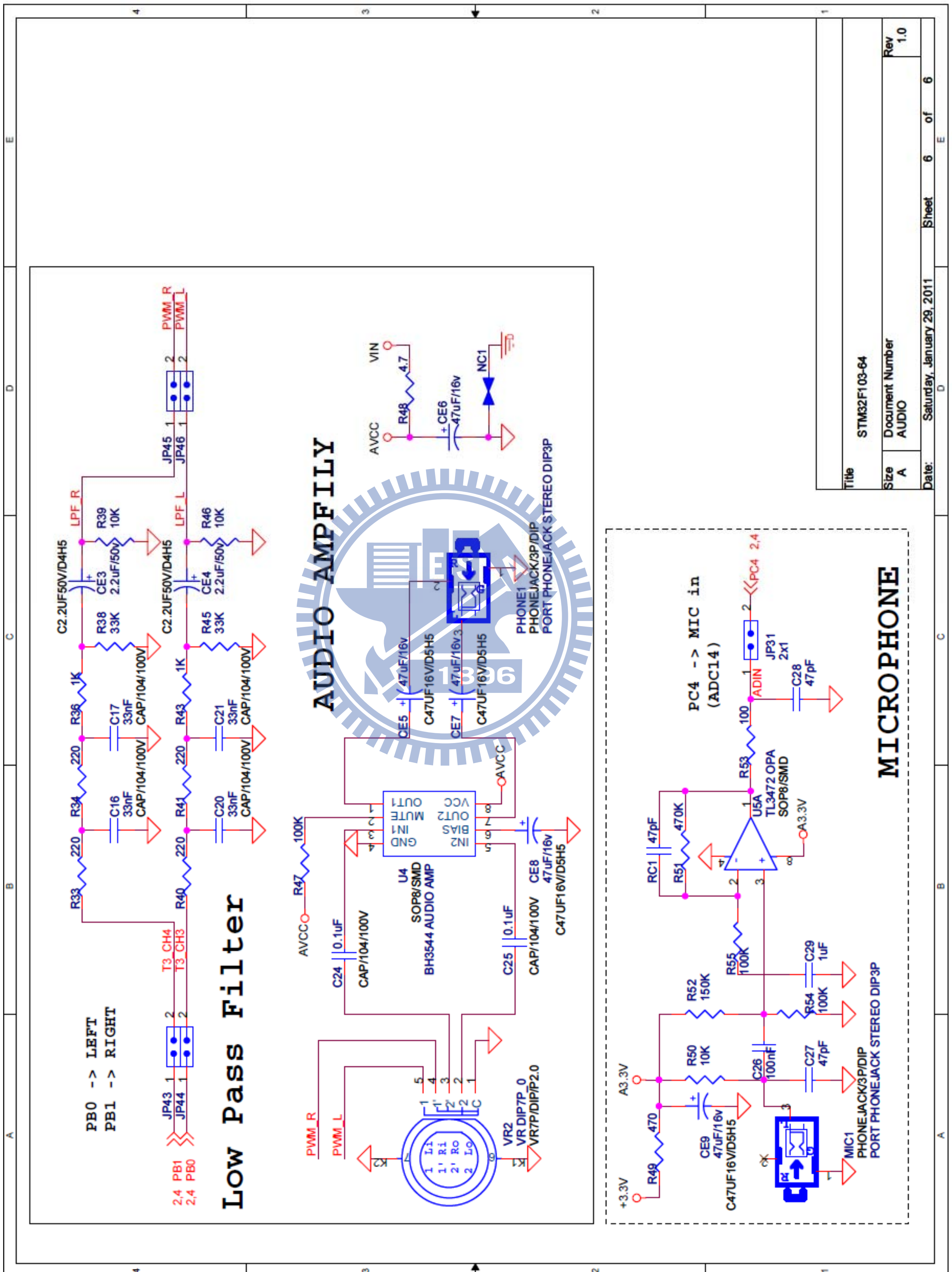
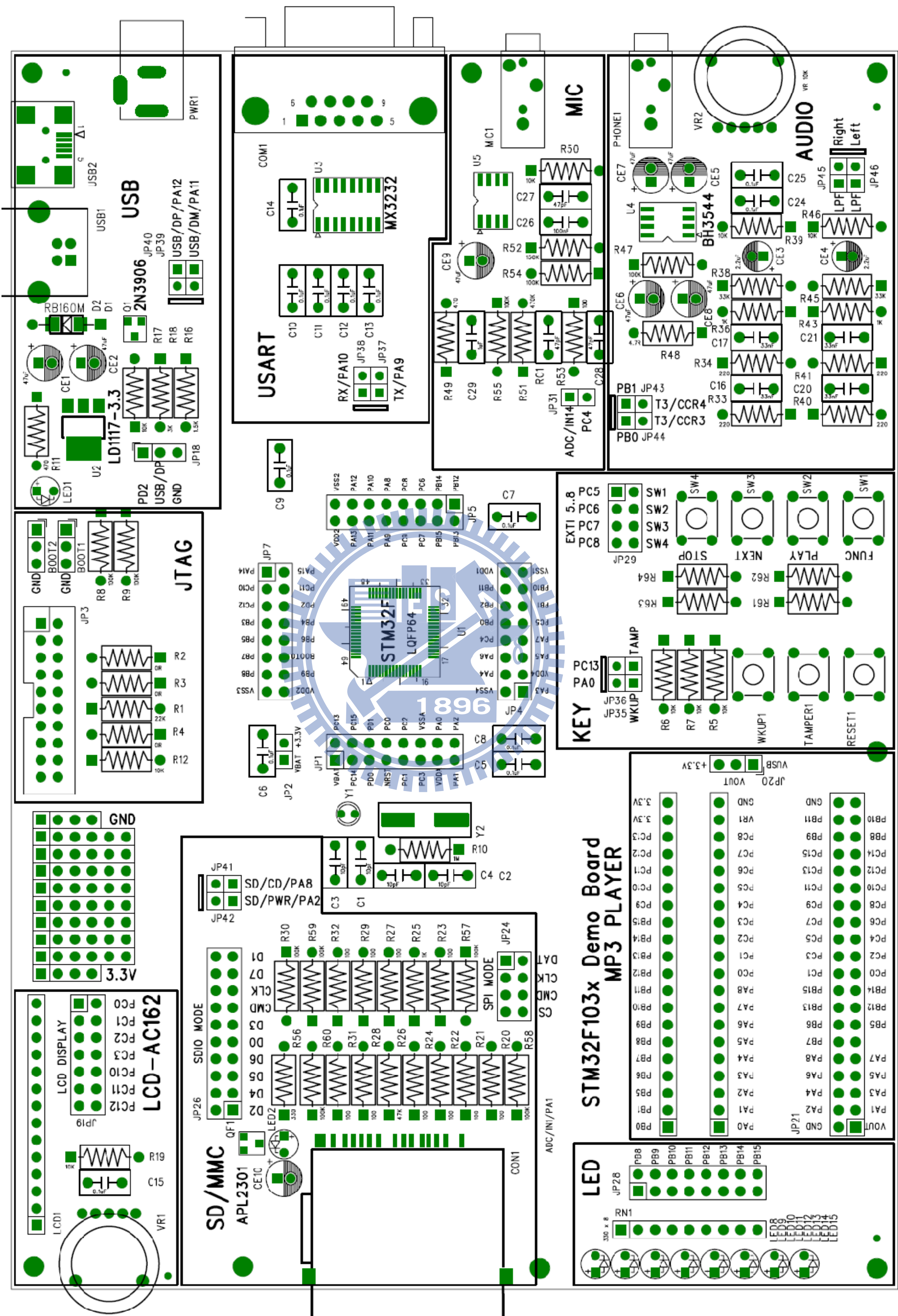


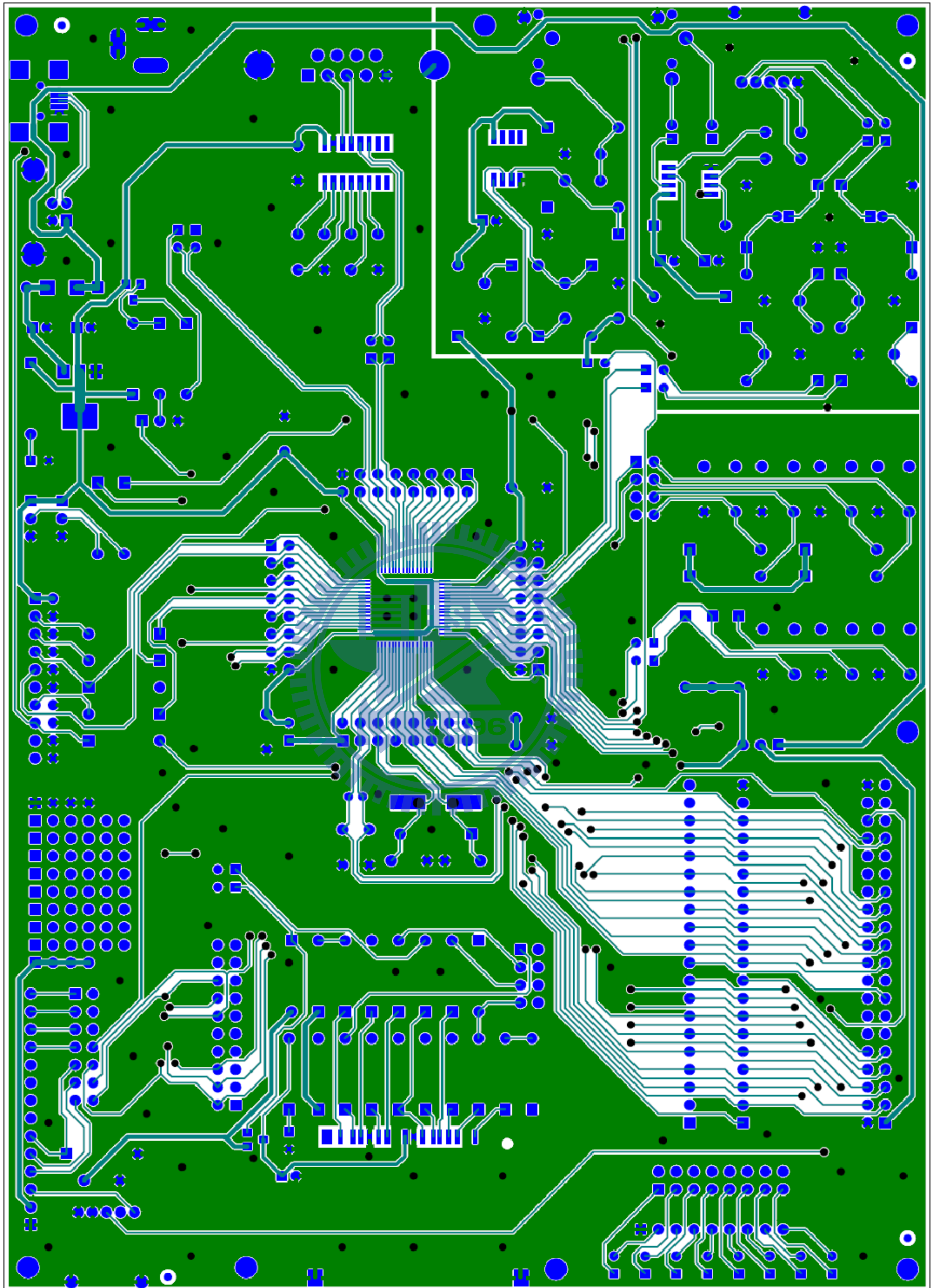
圖 5



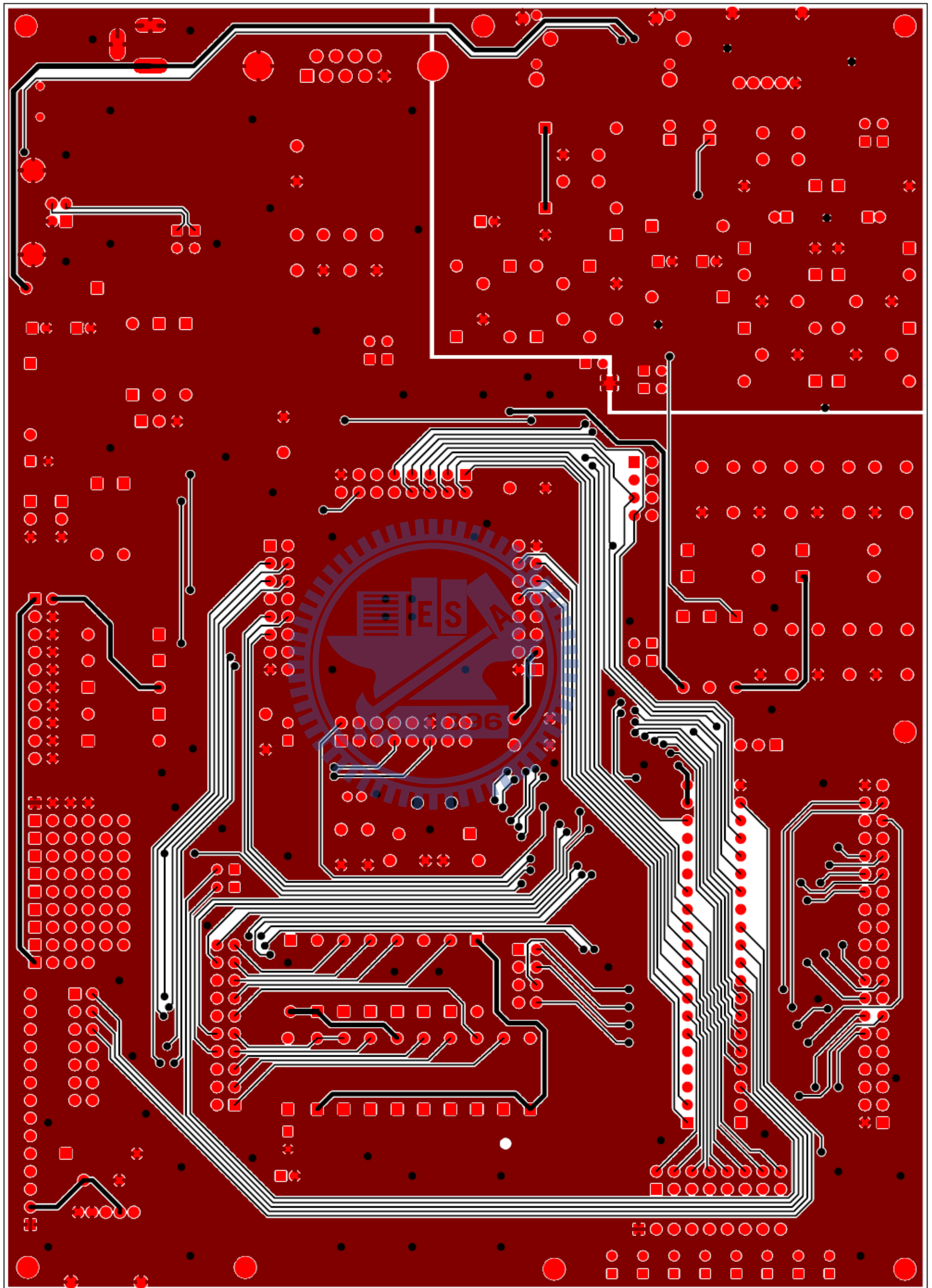
PCB 文字面



PCB 正面佈線圖



PCB 背面佈線圖



零件表

STM32F103 主 IC 電路零件			
Item	Quantity	Referance	Part Name
1	2	BOOT1,BOOT2	JMUPER 3X1
2	2	CE1,CE2	47uF/16v
3	4	C1,C2,C3,C4	10pF
4	5	C5,C6,C7,C8,C9	0.1uF
5	4	JP1,JP4,JP5,JP7	JMUPER 8X2
6	3	JP2,JP35,JP36	JMUPER 2X1
7	1	JP3	JMUPER 10X2
8	1	LED1	LED DIP2.54
9	3	WKUP1,TAMPER1,RESET1	SWITCH 4PIN
10	1	R1	22K
11	3	R2,R3,R4	0
12	4	R5,R6,R7,R12	10K
13	2	R8,R9	100K
14	1	R10	1M
15	1	R11	470
16	1	U1	STM32F103-64
17	1	U2	LDO1117-3.3V
18	1	Y1	XTAL 32.768KHz
19	1	Y2	XTAL 8MHz
20	8	JP8,JP9,JP10,JP11,JP14,JP15,JP16,JP17	JMUPER 8X1
21	2	JP12,JP13	JMUPER 4X1

RS232 電路零件			
Item	Quantity	Referance	Part Name
1	1	COM1	D-SUB9 FEMALE
2	1	U3	RS232 IC MAX232
3	5	C10,C11,C12,C13,C14	0.1uF
4	4	JP37,JP38	JMUPER 2X1

USB 電路零件			
Item	Quantity	Referance	Part Name
1	1	USB1	USB 1.1 PORT (DIP)
2	1	USB2	MINI USB POER (SMD)
3	1	Q1	BC857B
4	1	D1	1H4001/DIP
5	1	D2	RB160M-30V
6	1	PWR1	DC PWR JACK 5V
7	2	JP39,JP40	JMUPER 2X1
8	1	JP18	JMUPER 3X1
9	1	R16	1.5K
10	1	R17	10K
11	1	R18	3K

LCD-AC162 電路零件			
Item	Quantity	Referance	Part Name
1	1	VR1	VR 10K DIP7P
2	1	LCD1	LCD-AC162
3	1	R19	10K
4	1	C15	0.1uF

TEST I/O PORT 電路零件			
Item	Quantity	Referance	Part Name
1	1	JP19	JMUPER 7X2
2	1	JP20	JMUPER 3X1
3	1	JP21	JMUPER 20X2
4	2	JP47,JP48	JMUPER 20X1

SWITCH PORT 電路零件			
Item	Quantity	Referance	Part Name
1	4	PC1,PC2,PC3,PC4	SWITCH 4P
2	1	JP29	JMUPER 4X2

LED PORT 電路零件			
Item	Quantity	Referance	Part Name
1	8	LED8,LED9,LED10,LED11,LED12,LED13,LED14,LED15	LED DIP2.54
2	1	RN1	RN9PIN 470X8
3	1	JP28	JMUPER 8X2

SD CARD 電路零件			
Item	Quantity	Referance	Part Name
1	1	CON1	SD/MMC CON 13PIN
2	1	JP26	MUPER 10X2
3	2	JP24,JP29	MUPER 4X2
4	2	JP41,JP42	MUPER 2X1
5	1	QF1	MOSFET P-CH SI2301
6	1	LED2	LED DIP2.54
7	10	R20,R21,R22,R23,R24,R27,R28,R29,R31,R32	100
8	1	R25	1K
9	1	R26	47K
10	5	R30,R57,R58,R59,R60	100K
11	1	R56	330
12	4	R61,R62,R63,R64	NC

AUDIO AMPFILY AND LOW PASS FILTER 電路零件			
Item	Quantity	Referance	Part Name
1	4	JP43,JP44,JP45,JP46	JUMPER 2X1
2	4	R33,R34,R40,R41	220
3	2	R36,R43	1K
4	2	R39,R46	10K
5	2	R38,R45	33K
6	1	R47	100K
7	1	R48	4.7
8	1	VR2	VR 10K DIP 7P
9	4	C16,C17,C20,C21	33nF
10	2	CE3,CE4	2.2uF/16V DIP
11	2	C24,C25	0.1uF
12	4	CE5,CE6,CE7,CE8	47uF/16V DIP
13	1	PHONE1	PHONEJACK STEREO DIP3P
14	1	U4	BH3544 SOP8

MICROPHONE 電路零件			
Item	Quantity	Referance	Part Name
1	1	U5	TL3472 OPA
2	1	JP31	JUMPER 2X1
3	1	R50	10K
4	2	R54,R55	100K
5	1	R49	470
6	1	R51	470K
7	1	R52	150K
8	1	R53	100
9	1	C26	0.1uF
10	3	RC1,C27,C28	47pF
11	1	C29	1uF
12	1	CE9	47uF/16V DIP
13	1	MIC1	PHONEJACK STEREO DIP3P

自　　　傳

我生於臺北縣板橋市，家族成員有父母親，一個哥哥與兩個姐姐與我共六人，大學畢業後在台北工作約四年後，輾轉來到新竹科學園區工作，目前與妻子和小孩居住於竹北市，我的個性隨和與人相處融洽，平日閒暇之餘的休閒活動有閱讀、玩 Wii、騎腳踏車等等，由於大學畢業後工作至今已將近十年，雖然工作方面小有成就，但有時覺得自己需要再次成長與學習，以便在未來提供自己更寬廣的視野與道路。