國立中山大學 電機工程學系

碩士論文

基於 ARM 微控制器 linux 平台的開發

Porting Linux on ARM-Based Micro-controllers

研究生：蔡汝欽 撰

指導教授：李錫智 博士

中華民國 九十五 年 六 月

序言

研究所碩士班煎熬了多年，心力消磨俱疲，這段期間的失敗都歸咎自己努力多顯不足，蹉跎寶貴時間，因個人心情因素致研究多所耽擱，造成師長家人為我牽掛，敝人實感抱歉。

或許是我個人對事情處理能力欠佳，於事情決擇分叉點總覺茫然不知所措，未能理性來處理遭受的生活挫敗，導致這段學習歷程缺陷不甚完美。延畢心情固然沉重複雜，冷靜細細思索，最大障礙還是在自己的 EQ 休養略顯不足。

首先我要感謝指導教授李錫智老師給了我那麼多指教，反覆地給我機會，並予我生活津貼補助，提供我實作論文的方向與環境，也感謝樂謙電腦科技公司提供實作上支援。

這段期間家人給我的鼓勵使我堅定繼續完成學業，感謝大哥大嫂細心照顧生活上的點滴，因為你們無倦怠地付出，改善我生活起居，飲食獲得充足營養，得以有體力支撐完成學業，身體保持健康。

論文實作上碰到瓶頸，覺得研究上孤獨寂寞時，本校光電所 92 級畢業好友王俊鎧總是於百忙工作閒暇給我電話加油打氣，想起以前在中山一起學習的快樂日子，不時倍感溫暖，打起精神衝勁繼續往前邁進。

最後要特別向大陸深圳的朋友陳媛小姐表達致謝，非常開心有緣能認識她，感謝近快半年來她帶給我支持與鼓勵。

論文摘要

愈來愈多的嵌入式系統選擇 ARM 微控制器當作中央處理器，這類系統若沒有作業系統支持，應用範圍及便利性將受到限制，因此嵌入式作業系統變得迫切需要。市面上存在多種商業化嵌入式作業系統，其中 Linux 具有多項優點，已廣為接受。商業版 Linux 支付權利金比其他種類的嵌入式作業系統低，核心和應用程式源碼大多以 GPL 版權公開原始碼發佈，易於移植跨多種機器平台。

現今硬體技術發展成熟，3C 產值利潤毛利逐步快速下滑，因此整合現有技術符合實用經濟以使得產品增值，並提高自行開發比例以降低成本。開發者根據產品功能需求選用 ARM 微控制器，Linux 自由發表版未盡支持該廠牌 ARM 處理器，此時開發人員得自行移植 Linux 支援該機器平台，嵌入式作業系統無需額外支出權利金；或者尋求有支持的商業版 Linux，如此成本將額外支付權利金。

嵌入式系統以輕巧實用配置週邊設備，例如 UART 介面是低速率且經濟的傳輸方式，主機與目標板溝通可採用 RS-232 埠，Linux 下 RS-232 埠實現序列操控台功能，扮演鍵盤與螢幕的角色；工業控制常搭配 RS-xxx 資料傳輸介面，例如 RS-485 結合應用 modbus 協定構建廉價的監控網。網路傳輸也是嵌入式系統不可或缺的功能，需求較高資料傳輸速率的應用一般都選擇乙太網，socket 系統呼叫方便網路應用程式開發。嵌入式系統一般不搭載磁碟，為了保存永久性資料，flash 區塊磁碟系統是一個普遍的策略，藉由 MTD 子系統層（[28]）驅動 flash。MTD 區分成驅動模組及使用者模組，驅動模組採用 CFI（[40]）偵測 flash 晶片基本資訊（大小、廠牌 ID、單位可抹除區塊大小、分割區），以及抹除、寫入方法；使用者模組提供 flash 轉譯層及檔案系統，MTD BLOCK 模擬 flash 分割區成為磁碟區塊裝置，如此 Linux 能以檔案系統掛載 flash 分割區，在此我選用 JFFS2 檔案系統，JFFS2 是專為 flash 裝置特性設計的檔案系統型態。

本論文對移植 Linux 到 ARM 微控制器步驟做探討，第一章導論說明本實作動機，第二章簡介各項用於開發的工具與軟體，以及簡述移植實作的主要步驟。第三章說明 LH79525 架構平台及模擬板主要週邊，再介紹 ARM 的程式設計者模型。第四章全盤概括地提出 ARM Linux 基本知識，以及移植 ARM Linux 的重要議題。第五章詳述移植 Linux 至夏普 LH79525 機器平台，主要嵌入驅動週邊界面含 UART、網路 MAC、MTD 子系統層，各部份原始碼修改及核心組態設定。第六章我們一步步測試移植的 ARM Linux，並與樂謙電腦科技公司的 LF-314CP 溫度控制器（[46]）整合應用。第七章將整個系統實作提出未來展望、改善空間與結論，由於移植工作包含許多技術細節，我們希望本移植使 ARM Linux 移植有效率及降低成本，為開發者提供嵌入式系統發展程序及實驗平台帶來助益。

關鍵字：Embedded Linux，MTD subsystem，CFI，JFFS2，MCU，UART，MAC，JTAG ，FTL，XIP，ARM EVB，BSP，NOR flash，SDRAM，modbus

Abstract

More and more embedded systems choose ARM-based micro-controllers as CPU. If no embedded OS built with the system, the application scope will be restricted. Therefore, the need of embedded OS is vital. There are many embedded OS's in the market, but the embedded Linux has many advantages and is widely accepted. Commercial embedded Linux takes less refund than other embedded OS's. The kernel and most applications are distributed in GPL open source copyright, and is highly portable to many machine platforms.

Presently, the hardware key-technology is highly skilled. The margin of 3C industrial has gone down rapidly. Therefore, people focus on adapting integrated technology to practicality and innovation to make cost down. Developers choose appropriate ARM micro-controllers according to demanding functionality of their products. The microcontroller is not necessary running with Linux distribution. Two approaches can be used to resolve the embedded OS issue. The first approach is porting Linux to the platform without any refund. The second approach is to pay for commercial Linux.

Embedded system peripheral devices aim at powerful functionalities and economy. For instance, UART interface is cheap and low data transfer rate. The target board communicates with host via RS-232. RS-232 acts as serial console to play dumb terminal under Linux. Industrial applications often make use of RS-xxx for UART physical transmission layer. For instance, RS-485 applies modbus protocol to build cheap monitor systems. Network transmission is a necessary function, and it generally achieves high data transfer rate application through Ethernet. The UNIX-like network socket has served network application very well. Embedded systems are usually diskless systems. In order to keep permanent data, using flash memory as block disk system is a widely adapted strategy and which operates flash memory through MTD subsystems（[28]）. An MTD subsystem contains two different modules, "user" and "driver". In the driver module, CFI（[40]） is applied to probe flash chip, partition it and provide operating function. Flash translation layer and file-system are applied in the user module. MTD BLOCK is used to emulate the flash partitions as block devices which are then mounted into Linux virtual file system（VFS）with JFFS2 type, designed according to the feature of flash devices.

In this thesis, we will describe in detail the procedure of porting Linux to ARM micro-controllers. The motivation of the work is introduced in chapter 1. In chapter 2, we introduce development tools and the main flow of the porting procedure. In chapter 3, we describe the LH79525 platform and the main perepherals on the target board, then introduce the ARM programmer model. In chapter 4, we examine the

required knowledge and the important issues for porting ARM Linux. In chapter 5, we describe the details of porting Linux to run with Sharp LH79525, including modifying the key source codes and adjusting kernel configuration for embedding the UART, ethernet MAC, and MTD subsystem. In chapter 6, we do step-by-step validation and apply an integrated application with the LF-314CP temperature controller（[46]） by law-chain technology for the LH79525 target board running with the ported ARM Linux. In chapter 7, we present some issues for future work and improvement, then make a conclusion for the thesis.

Key-word： Embedded Linux，MTD subsystem，CFI，JFFS2，MCU，UART，MAC，JTAG ，FTL，XIP，ARM EVB，BSP，NOR flash，SDRAM，modbus

目錄

第 1 章 導論 1

1.1 研究動機 1

1.2 研究貢獻 2

1.3 論文架構 4

第 2 章 嵌入式系統與 ARM Linux 開發流程與工具 5

2.1 嵌入式系統應用領域 5

2.1.1 軟硬系統整合：嵌入式作業系統(RTOS) 6
2.1.2 SOC 設計 6
2.1.3 應用程式發展 7
2.1.4 內容服務 7

2.2 ARM Linux 開發流程 7

2.2.1 選用微控制器 8
2.2.2 選用作業系統 8
2.2.3 實作流程 9

2.3 開發環境規劃 10

2.3.1 開發主機平台 10
2.3.2 Jtag-ICE 除錯 11

2.4 開發工具與使用的軟體套件 12

第 3 章 機器架構平台與模擬板主要元件 14

3.1 LH79525 微控制器 14

3.1.1 ARM720T 組成單元 15
3.1.2 外部記憶體控制器（EMC） 16
3.1.3 非同步傳輸控制（UART） 17
3.1.4 Real Time Clock、Watchdog Timer、Timer 18
3.1.5 Reset, Clock, Power Controler（RCPC） 18

3.1.6 Ethernet MAC（EMAC） 19
3.2 Flash Memory 20
3.2.1 NAND v.s. NOR flash 20
3.2.2 NOR flash：MX29LV640BT 22
3.3 SDRAM 23
3.4 ARM 處理器程式設計者模型 23
3.4.1 記憶體格式 23
3.4.2 暫存器群 23
3.4.3 程式旗標暫存器 26
3.4.4 ARM 處理器模式與例外處理 26
第 4 章 移植 Linux 作業系統 28
4.1 Linux kernel 2.6 28
4.1.1 Linux 核心發展歷史 28
4.1.2 Linux 核心組成主要部份 28
4.1.3 Linux 核心的分層次概念 29
4.1.4 Linux 原始碼目錄 31
4.2 移值 ARM Linux 的一般化步驟 32
4.3 Bootloader 與作業系統載入初期初始化 36
4.4 Linux 驅動程式運行原理 37
4.4.1 檔案裝置型驅動程式主要功能 37
4.4.2 網路裝置區動程式 38
4.4.3 驅動程式與使用者層之關係 38
4.4.4 新增驅動程式模組至 Linux 核心 39
4.5 操控台（console）、TTY、序列埠 39
4.6 程式區段規劃與連結草本（linking script） 40
4.7 核心映像與根目錄檔案系統映像配置規劃 41
4.8 根目錄檔案系統掛載 43

4.8.1 掛載 RAM disk 成根目錄檔案系統 43
4.8.2 掛載 MTD 裝置成根目錄檔案系統 44
4.9 Linux 下存取 NOR flash 44
第 5 章 移值 ARM Linux 支援 Sharp LH79525 47
5.1 新增支援機器平台的目錄與組態參數 47
5.1.1 命令列參數 CONFIG_CMDLINE 48
5.1.2 重要的符號 49
5.1.3 定義連結草本（linking script）區段 50
5.2 ARM linux 核心啟動步驟 51
5.2.1 Bootstrapping 51
5.2.2 核心啟動 53
5.2.3 start_kernel 函式 55
5.2.4 setup_arch 函式 55
5.2.5 例外向量表初始化：trap_init 函式 56
5.2.6 ARM Linux 建立虛擬記憶體分頁表 57
5.3 註冊 MCU ID 與 ARM Linux 啟動 ID 檢查 58
5.3.1 機器平台 ID 58
5.3.2 機器平台描述子 59
5.3.3 處理器平台描述子 60
5.3.4 MCU ID 及 ARM CORE ID 檢查 61
5.4 機器平台初始化及週邊驅動程式 61
5.4.1 計時器初始化 61
5.4.2 中斷控制器與中斷資源管理描述子初始化 63
5.4.3 初始化操控台 64
5.4.4 序列埠驅動 65
5.4.5 乙太網 MAC 層驅動 68
5.4.6 MTD 驅動 NOR flash：MX29LV640BT 73
5.5 網路服務設置 76
第 6 章 驗證移植至 Sharp LH79525 的 ARM Linux 78
6.1 基本測試 78

6.2 連接溫度控制器應用 79

第 7 章 結論及未來展望..... 80

參考文獻及附錄 82

附錄 A. ARM cross compiler、uclibc、busybox..... 84

附錄 B. Linux 核心組態參數設置與原始碼關係..... 85

附錄 C. 初始化 Sharp LH79525 的 SDRAM 控制器..... 87

附錄 D. 設定 Nor flash 存取時序..... 90

附錄 E. ARM 虛擬記憶體存取流程 91

附錄 F. Skyeye 模擬器安裝與使用..... 93

附錄 G. 幾種燒錄 NOR flash 的方法..... 96

附錄 H. tty 層與終端機設定..... 97

附錄 I. MTD 裝置下使用 JFFS2 檔案系統..... 98

附錄 J. 模擬板線路圖 101

附錄 K. 模擬板設計要點說明..... 104

附錄 L. 原始碼重要檔案索引列表..... 108

圖目錄

圖 2-1 嵌入式系統用戶需求與技術需求要點.....5
圖 2-2 典型的嵌入式系統應用.....6
圖 2-3 嵌入式系統主要處理器市佔率.....8
圖 2-4 開發環境規劃.....10
圖 2-5 Jtag-ICE 軟硬體運作關係圖.....11
圖 2-6 Linux、uClibc、busybox 生成關係.....13
圖 3-1 LH79525 主要模組區塊.....14
圖 3-2 ARM720T 處理器內部單元.....16
圖 3-3 LH79525 各記憶體 bank 對應位址.....17
圖 3-4 LH79525 EMAC block diagram.....20
圖 3-5 NAND v.s. NOR flash cell layout.....22
圖 3-6 big-endian 記憶體排列次序.....24
圖 3-7 little-endian 記憶體排列次序.....24
圖 3-8 ARM/Thumb 下的可視暫存器.....25
圖 3-9 各種模式下的暫存器.....25
圖 3-10 CPSR 程式旗標暫存器.....26
圖 4-1 作業系統之核心層與使用者層.....29
圖 4-2 核心碼的架構相依與機器相依分層.....30
圖 4-3 核心碼與硬體分層關係.....30
圖 4-4 Linux 核心目錄.....31
圖 4-5 檔案裝置型驅動程式.....37
圖 4-6 使用者程序與驅動程式間關係.....39
圖 4-7 連結草本應用實例.....41
圖 4-8 bootloader、核心及根目錄映像檔空間配置.....42
圖 4-9 掛載根目錄檔案系統步驟.....43
圖 4-10 MTD 子系統層與 VFS 層關係.....46
圖 5-1 核心啟動主要函式執行流程.....54
圖 5-2 計時器初始化函式呼叫流程.....62
圖 5-3 中斷控製器及中斷資源管理描述子初始化.....64
圖 5-4 Linux 序列埠終端運作原理.....66
圖 5-5 序列裝置描述子註冊呼叫函式關係.....67
圖 5-6 網路分層架構.....69
圖 5-7 LH79525 MAC 驅動程式初始化、開啟、結束呼叫函式展開 72
圖 5-8 網路服務 daemon.....77
圖 6-1 RS-485 典型應用.....79

表目錄

表 2-1 幾種常見運行於 ARM 的嵌入式作業系統比較 9
表 3-1 LH79525 記憶體映射分佈 17
表 3-2 NOR v.s. NAND flash 比較 21
表 3-3 例外事件處理及進入模式、優先順序 27
表 3-4 例外處理向量表 27
表 3-5 例外處理結束指令 27
表 4-1 Linux 核心移植分層 31
表 4-2 核心目錄說明 32
表 4-3 MTD 分割區配置 42
表 5-1 重要核心符定義 50
表 5-2 ARM Linux 連結草本重要初始化區段 50
表 5-3 重要 tag 參數 52
表 5-4 tag 參數與核心之關係 52
表 5-5 LH79525 機器描述子 59
表 5-6 呼叫函式與其對應之機器平台描述子的函式指標 61
表 5-7 ARM Linux 中斷資源管理描述子陣列維護函式 64
表 5-8 struct console 結構成員 65

Appendix 圖目錄

圖-APP. 1　SDRAM 狀態機 87
圖-APP. 2　ARM720T 內部 MMU 存取流程 92
圖-APP. 3　vmware 下 Linux 運行 Skyeye 95
圖-APP. 4　vmware 下運行 microsoft windows 95
圖-APP. 5　Sharp LH79525 外接腳位（[13]） 101
圖-APP. 6　8MB NOR flash：MX29LV640BT/TSOP（[29]）接腳圖 102
圖-APP. 7　32MB SDRAM：W982516CH-75（[26]）接腳圖 102
圖-APP. 8　PHY transciever：DM9161A（[43]） 103
圖-APP. 9　RS-232 transceiver：HIN232 104
圖-APP. 10　Banyuan-U 抓取模擬板結果 106

Appendix 表目錄

表-APP. 1　開機記憶體裝置選擇 105
表-APP. 2　原始碼重要檔案索引列表 108

第1章　導論

1.1　研究動機

今日電子產品發展市場趨向實用性，其內部設計常常不需要非常高深技術，造就廣大的消費性電子推陳出新，但探究其核心技術基礎並無太大突破。多數數位式產品設計多以微控制器或微處理器當作系統核心，此類微控制系統一般具特定應用場合，通稱這類系統為「嵌入式系統」，這類產品特性不外乎體積小、省電、操作界面人性化。就硬體而言，特殊運用 IC（ASIC）進入 SOC（system-on-chip）世紀新元，類比與數位混合界面 IC 技術純熟，在應用上，硬體電路與架構幾乎已達到制式化標準，現今 ASIC 提供內部暫存器或韌體程式化方法，目地在於增加硬體運用彈性度與達成多功能性；軟體方面，其彈性程度與創造力更凌駕於硬體之上，軟體核心價值在於演算法與策略機制運用，補足硬體於效能不足部份，擔任系統與使用者之間的媒介。尤其 C 語言廣泛被各軟硬體平台公認為標準語言，縱使有各種 C 語言（以 GNU、Microsoft 兩者為最大宗），UNIX-like 作業系統下發展的應用程式存在高移植性，而且在 GNU 自由軟體執照宣言下，Linux 的自由與發展多元性提供了低成本的開發環境。藉由基本（primitive）功能 IC 元件與軟體工具，發展系統變得多元化，軟硬體整合可以創造更多應用，帶給產業更多產值，提升產業升級，也完美呈現精湛科技。

舉兩個實際嵌入式系統設備應用例子，第一是結合家電產品控制，以區域網路（如紅外線、RS-232、ethernet）連結，主機再以無線上網（如 GPRS）將資料回報至用戶手機，使用者也可透過遠端遙控家電設備。第二個是工廠運用，例如工廠機器需要做監測（例如溫溼度），為了集中化管理，可以在每台機器上掛載監控器，串接所有監測端，一般工業標準 ITU/EIA RS-485 搭配 modbus 協定。廣泛被採用，視廠房大小決定是否中繼增加信號增強器（repeater），或選擇其它區域網路連結方式，如此工廠作業可節省設備看管人力

，對不需現場人員的非危險性設備，更可透過網路遠端做監控，有狀況再請人員到場查看即可。如此簡單的應用，都顯示出嵌入式系統廣大可擴充度，也讓人們更貼近便利的生活，工廠企業人力成本降低，在在都呼應「科技使終來自於人性」。

ARM 微處理器已廣泛被運用於許多嵌入式系統，它兼具低功率優點，採用 RISC 管道執行程序設計，ARM core 嵌入了 Jtag（IEEE std1149.1）介面標準（[9]）以提供除錯功能，透過 Jtag 界面搭配 ICE（in-circuit emulator）整合開發軟體，系統開發者方便程式除錯。根據 linuxdevices.com 對嵌入式系統應用市場 94 年度調查報告，ARM 處理器擁有市場上近五成市佔率，顯現出 ARM 除了能滿足特定應用產品（如手機、遊戲機）需求外，更能勝任中高階運用（多媒體影音編解碼 codec 運算），尤其可攜式裝置將產品的成本效益值（C/P）淋漓盡致地發揮，又兼具產品等級擴充彈性（flexibility）。

鑒於 ARM 日趨普遍，且 ARM 微控制器型號眾多，移植 Linux 至支援特定 ARM 機器平台愈顯重要。本論文移植 ARM 基礎微控制器 Linux 平台，首先我們完成了 Linux 作業系統移植，接著設計一塊模擬板供測試 ARM Linux，從中詳細了解軟硬體整合發展步驟方法。本論文選用了 Sharp LH79525（[13]）微控制器當實驗平台，此 MCU 核心 CPU 為 ARM720T（[7]），整合 UART、LCD、乙太網 MAC 層等常見標準週邊硬體控制介面，屬典型的 SOC 微控制器。

1.2 研究貢獻

本論文詳細提出移植 ARM Linux 步驟，發展過程具備以下應用特點：

- 採用業界內受歡迎的 ARM7 SOC 嵌入式微控制器為系統核心處理器，整合了多項週邊控制。驅動的主要週邊包括 UART、MAC、MTD 子系統。本系統尚未嵌入 LCD 及觸控式介面相關軟硬體，日後可依應用需求擴充搭配嵌入式 GUI（如 MiniGUI、PicoGUI）

發展圖形使用者界面。

- 開發過程中，開發主機由 RS-232 下載資料至目標板。在 Linux 運行下，目標板的 RS-232 埠實現標準輸出與輸入，透過 serial console（早期稱 dumb terminal）操作管理 ARM 目標板上的 Linux 。
- Boot-loader 尚無驅動 Ethernet MAC（EMAC），若完成驅動 EMAC，則開發主機便可藉由乙太網下載資料至目標板，縮短下載程式時間以增加測試效率。
- ARM Linux 作業系統核心支援 NFS，對於跨平台發展應用程式極具效益。發展目標板的應用程式先於開發主機中編輯與編譯，開發主機啟用 NFS 伺服器將存放此應用程式的目錄分享，目標板下的 Linux 透過 NFS 客戶端直接使用主機上的執行檔。如此可以加速跨平台 Linux 的應用程式開發。
- 發展之 ARM Linux 作業系統安裝於 NOR flash 內，選用 NOR 型快閃記憶體，比起 NAND 其優勢在於可具備 XIP（eXecute In Place）能力，節省執行程式佔用空間。本作業系統核心目地碼低於 2MB；所有應用程式採用 uclibc（[4]）動態連結程式庫編譯，以 busybox（[4]）為 Linux 的系統公用程式，因此根目錄檔案系統大小占用不到 2MB 空間，整套系統僅佔用約 4MB 空間。
- ARM Linux 核心的 MTD 子系統支援 char 及 block 兩種 RAW 模式存取 NOR flash，block 模式使得 OS 能將 NOR flash 的分割區模擬成磁碟，再以 JFFS2 檔案系統將它掛載(mount)至 Linux 的虛擬檔案系統（VFS）。
- GNU C、GNU Linux、GDB、Skyeye 等發展環境工具均可免費取得，我們成功地以這些免費工具完成移植 ARM Linux，雖較耗時但成本卻極低於商業版 WindowsCE，而且無權利金問題。

- ARM Linux 的 UART 驅動程式支持工業化標準 ITU/EIA RS-485 半雙工存取控制，透過 IOCTL 方法控制 RS-485 的 TX/RX 方向切換（通常由 GPIO，General Purpose I/O pin），對即時性較低的控制應用相當合適，成本也低廉。
- 目標板外接 Jtag 埠，可直接在線燒錄 NOR flash，不需額外燒錄設備。ARM Linux 的 MTD 子系統層搭配 MTD 公用程式以提供燒錄 NOR flash 的 方法。Jtag 亦使用於 bootstrap 及 ARM Linux 發展階段的除錯，搭配 GDB 或 ARM 公司的 multi-ice 除錯環境。

1.3 論文架構

本篇論文章節安排如下：第二章簡介各項用於開發的工具與軟體，以及簡述移植實作的主要步驟。第三章說明 LH79525 架構平台及模擬板主要週邊，再介紹 ARM 的程式設計者模型。第四章嵌入式系統目前應用與發展概況，全盤概括地提出 ARM Linux 基本知識，以及移植 ARM Linux 的重要議題。第五章詳述移植 Linux 至夏普 LH79525 機器平台，部份移植的原始碼修正需根據我們自行設計模擬板硬體資源配置，主要嵌入驅動週邊界面含 UART、網路 MAC、MTD 子系統層，各部份原始碼修改及核心組態設定。第六章我們一步步測試移植的 ARM Linux，並與樂謙電腦科技公司的 LF-314CP 溫度控制器（[46]）整合應用，完成移植 ARM Linux 實際測試。第七章將整個系統實作提出未來展望、改善空間與結論。因移植建構一套可運作的嵌入式系統，涉及許多技術細節，在，而且不可能一一說明 Linux 原始碼，我們僅就與移植相依的部份提出，未能全面講述，於 Appendix 我們將補強一些額外附加要點說明。

第2章　嵌入式系統與 ARM Linux 開發流程與工具

2.1　嵌入式系統應用領域

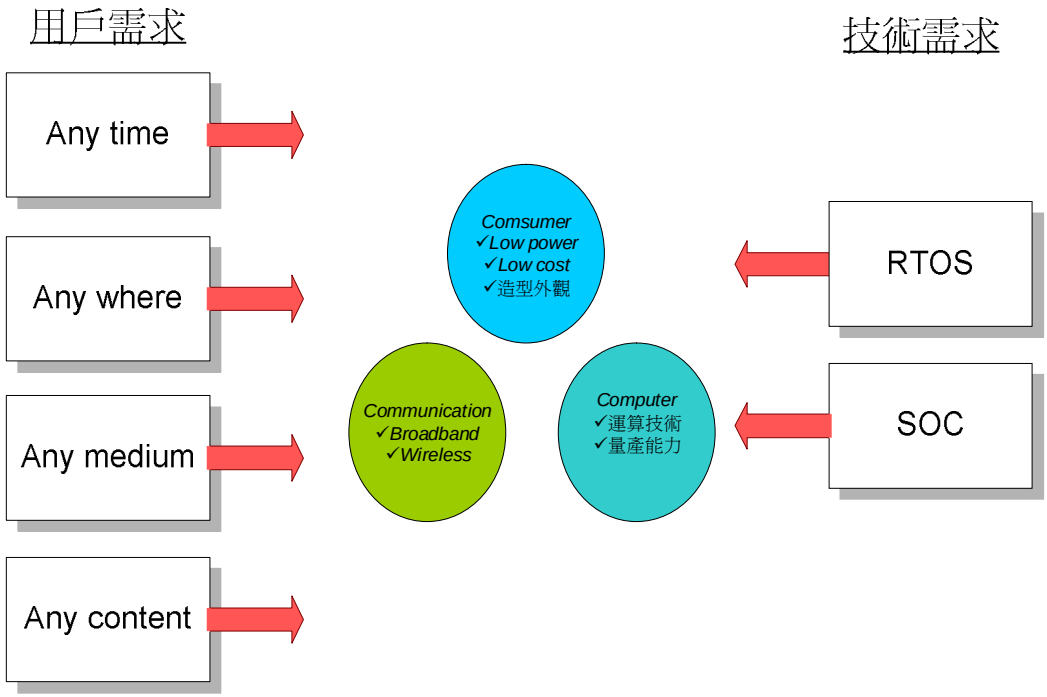

圖 2-1　嵌入式系統用戶需求與技術需求要點

依據英國電機工程師協會的定義，嵌入式系統為控制、監視或輔助設備、機器或甚至工廠運作的裝置。它是一種電腦軟體與硬體的綜合體，並且特別強調「量身定做」的原則，也就是基於某一種特殊用途，我們就會針對這項用途開發出截然不同的一項系統出來，也就是所謂的客制化（Customize）。在這後 PC 時代，產品花樣多采多姿，PC 產業面臨傳統產業化挑戰，在幾年前市場再度導入「嵌入式系統」，試圖將資訊化帶入生活其他領域應用，更結合了 3C（computer、comsumer、communication）技術，圖 2-1 說明 3C 用戶及技術需求，各應用層次有不同著重特點，例如消費性電子講究低功率消耗、價位低廉、造型美觀，這些應用採用的技術並無太多新穎技術，但卻讓資訊產業開拓更大商機。在資訊產業裡，目前學校教育中不是偏向極硬 (hardware design)，就是超軟(software development)，硬體設計人員比較缺乏系統全面整合設計，而軟體

發展人員只要看到硬體問題就傻眼了，因此嵌入式系統開發者本身對軟硬體都需具一定熟稔水準，但並非深入所有細節，而是要具備軟硬體系統整合能力。本章節說明嵌入式系統的軟硬系統整合、SOC 設計、應用程式發展以及內容服務四方面。

典型嵌入式系統產品如手機、PDA、GPS、Set-Top-Box 或是嵌入式伺服器（embedded server）及精簡型終端設備（thin client）、家用 DVD 撥放機…等消費性電均是，乃至近期讓 Apple 熱賣的 i-Pod，其技術都是已有的，成功關鍵在於能掌握大眾使用產品習性，相信未來這塊應用領域仍會主導電子資訊產品。

圖 2-2　典型的嵌入式系統應用

2.1.1　軟硬系統整合：嵌入式作業系統(RTOS)

與 PC 作業系統比較，嵌入式作業系統並未要求全能，但必須能夠依據系統設計規格，有效率的發揮出硬體的運算能力，使得產品達到效率／價格比（C/P 值）的最佳化，大多數的系統會要求全自動完成所設定的工作，例如工廠或是銀行的系統；除了原本在嵌入式領域耕耘已久的 VxWORK、QNX、Nucleus 等等之外，新興的主要競爭產品包括 Palm OS、Windows CE、Linux 等，其中 Embedded Linux 作業系統免費授權的特性，已廣為採用。

2.1.2　SOC 設計

嵌入式產品所需之處理器及晶片組較 PC 要求體積小、散熱佳、省電，因

此多採用高整合度的 SOC (System-on-Chip)為其處理器核心，為了儘速縮小製程技術進步與設計生產力間的差距，並加速 SOC 的實現，SIP(Silicon Intellectual Porperty)的重複使用(Re-Use)成為各方矚目的焦點。

2.1.3 應用程式發展

嵌入式軟體可區分為使用者端的應用軟體及伺服器端的整合軟體，伺服器端的軟體可能以 Linux 或是 Windows 為核心，並搭配各種資料庫系統；使用者端由於各種產品種類繁多，可開發出的軟體也相對增加，例如 Palm 號稱有上萬種應用軟體可以使用。除了原本各種平台專屬的應用軟體之外，現在更有利用 Java 跨平台程式開發的軟體加入這個陣容，軟體的種類變得更多。不同軟硬體平台的應用程式主架構雖不變，但必須做移植與跨平台編譯才有辦法重新使用。

2.1.4 內容服務

由於嵌入式產品必須能隨身攜帶或走入居家生活，故其體積上要求輕薄短小、造型及顏色必須個人化、輸入必須自然化、輸出必須多媒體化才能吸引消費者；另方面嵌入式產品與網路結合，所以與網路服務提供者或電子商務業者極易結合，也就是嵌入式產品連上網路之入口網站及其內容(HTML/XML)可能由廠商負責提供，像是日本 NTT DoCoMo 所發展的 iMode 服務就是一個最好的例子，國內的電信三雄中華、遠傳、臺灣大也都積極朝內容服務加值做提昇。

2.2 ARM Linux 開發流程

移植的 ARM Linux 最終需實際在目標機器平台執行測試，而開發一項嵌入式應用產品必需對處理器與作業系統選用作評估，因為市場上處理器型號眾多，考量多以成本與符合應用挑出最合適的處理器來使用；為了能在機器上驗證

移植的 Linux，以 Skyeye（[30]）對移植的 ARM Linux 做初步模擬，若經費允許再購買以該型號處理器為基礎的開發板套件來驗證，最終再客製化一塊自家產品的模擬板驗證，如此能獲得更精確開發驗證。

2.2.1 選用微控制器

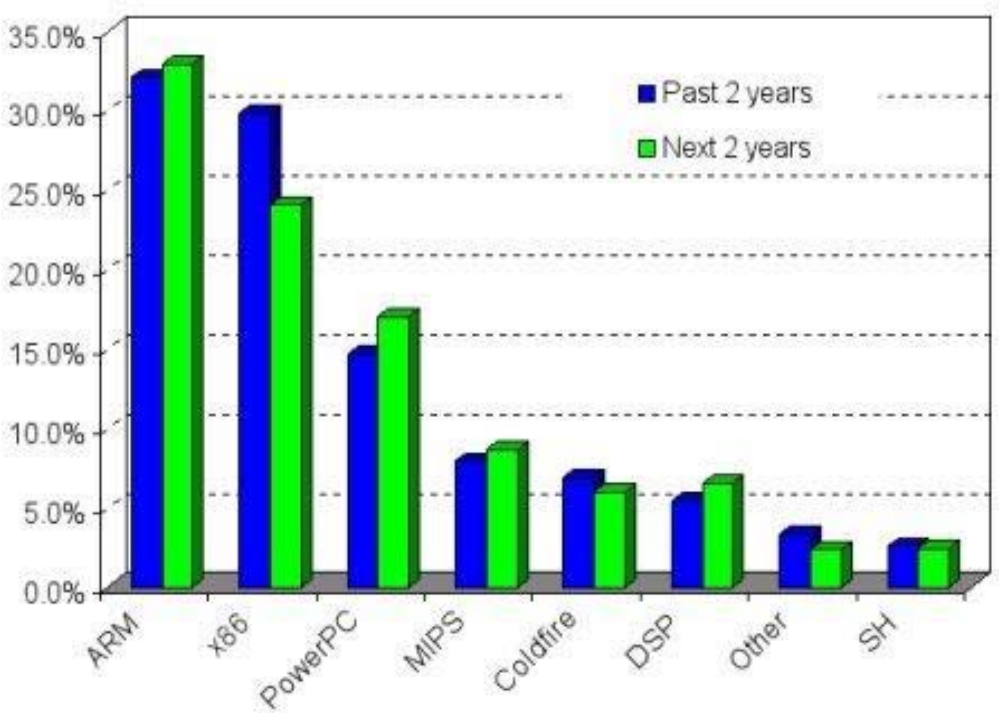

圖 2-3 嵌入式系統主要處理器市佔率

根據（[1]）2005 年度市場調查報告，在嵌入式系統處理器市佔統計表（圖 2-3），ARM 坐擁最高比例，在本實作考量中階應用，我們選用 ARM 7 系列微控制器，型號 Sharp LH79525（[10][13]），此 MCU 發表於 2005 年，核心工作頻率最高可達 76.25MHz，整合多項週邊控制功能，16C550 UART、I2C、Ethernet MAC、中斷控制器、LCD。另一因素則是 SDK-LH79524-3126-10（[5]）開發套件亦可用來實地驗證開發於 LH79525 上的軟體。

2.2.2 選用作業系統

ARM 平台可供選用較受歡迎的作業系統有 Windows CE、uClinux、ARM Linux、uC/OS-II、各種商業化 Linux。我們從使用執照、系統大小、權利金、是否支持 MMU、開放原始碼、等因素列表比較，考量到資源上的取得有限，以及降低開發成本因素，我們選用 ARM Linux（[16]），雖然開發時效性較長，但多項以 GNU GPL 公開發行的軟體計劃在進行，應用程式資源豐厚。商業版 Linux 主要好處是廠商收集多應用程式，包裝成套件，使用者可免去尋找應用程式的時間，一旦使用商業版 Linux 開發產品，產品需支付權利金，其缺點是也未必支援開發者選用的處理器。

表 2-1　幾種常見運行於 ARM 的嵌入式作業系統比較

評估要點 / 項目	License	Open source	popularity	Kernel+ filesystem Size(KB)	maintenance available	With MMU	GUI	Refund
uclinux	Free	Yes	good	<3000(typical)	Yes	No	MiniGUI	No
armlinux	Free	Yes	good	<4000(typical)	Yes	Yes	MiniGUI	No
MontaVista linux	No	Yes	Ordinary	<3000(typical)	Yes	Yes	MiniGUI	Yes
uC/OS-II	No	Yes	Ordinary	<1000(typical)	Yes	Yes	uC/GUI (licensed)	Yes
WindowsCE	No	No	medium	<3000(typical)	Yes	Yes		Yes

2.2.3 實作流程

實作流程分軟體與硬體各自獨立發展驗證，軟體開發由 Skyeye（[30]）做初步移植模擬，再由 SDK-LH79524-3126-10 發展套件驗證，最後再與我們客製化的模擬目標板合併測試，完成 ARM Linux 移植運行。

軟體開發步驟要確定微處理器、作業系統選用評估、移植作業系統、模擬作業系統。硬體開發步驟包含客制化模擬板、開機載入程式（bootstrap）測試、Jtag 通訊測試、NOR flash 燒錄測試、RS-232 埠。利用 Jtag-ICE 硬體與除錯軟體工具反覆測試直到作業系統於模擬板上正常運作。

2.3 開發環境規劃

簡易開發環境規劃如圖 2-4 所示，我們選用 Linux+x86 PC 當發展平台，主機（host）安裝跨平台編譯器與程式庫，程式可透過 RS-232 下載至目標板（target），並扮演目標板的操控台角色，等到網路界面發展完成後，程式下載可透過 Ethernet，進一步改善重覆下載的測試效率。目標板運行 Linux 後，更可透過 NFS 存取主機上編譯完成的跨平台應用程式 ，不需每次重新將應用程試下載至目標板內。

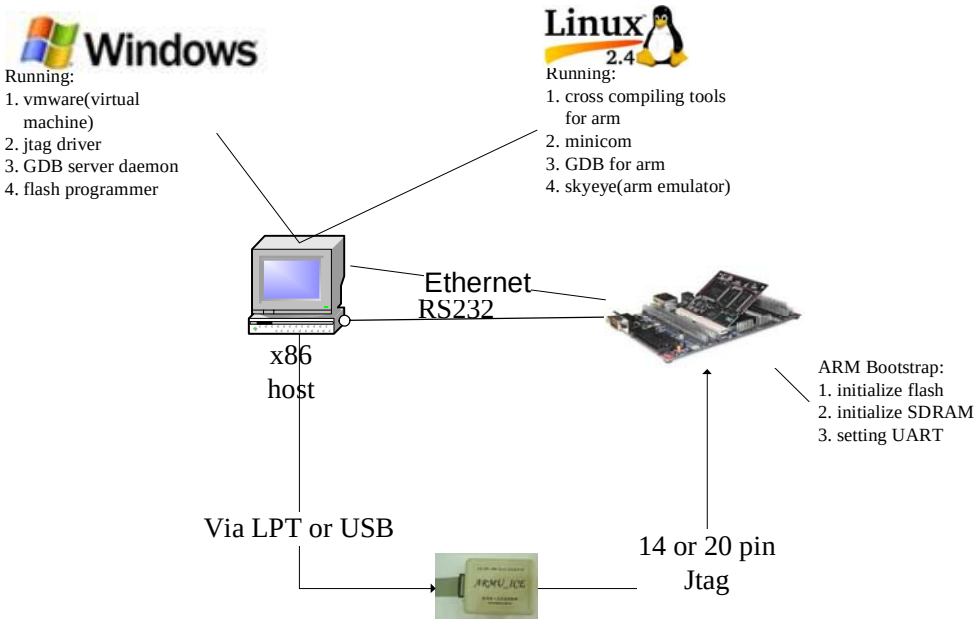

圖 2-4 開發環境規劃

2.3.1 開發主機平台

開發主機安裝了 vmware（[45]），這是一套運行在微軟 windows 下的 x86 虛擬機器，我們將 Linux 作業系統安裝在虛擬機器上。虛擬機器的所有硬體資源與 windows 下的一樣，好處是方便使用者同時使用不同的 PC 平台作業系統，例如本開發採用 Jtag 硬體的驅動程式只於 windows 下有，因此我們既同時需要 windows 和 Linux，但不可能因這樣而動用兩台 PC 主機，最方便的方式就是用虛擬機器讓不同的 x86 作業系統同時並存運作。虛擬機器可以保存目前作業

系統狀態，一旦不小心損毀，只要回存前一次狀態，幾分鐘內就可再正常運作，對於開發中常常需要一再重新編譯與修改某些軟體，可避免因嘗試這些修改失敗而造成系統破壞的風險。

2.3.2 Jtag-ICE 除錯

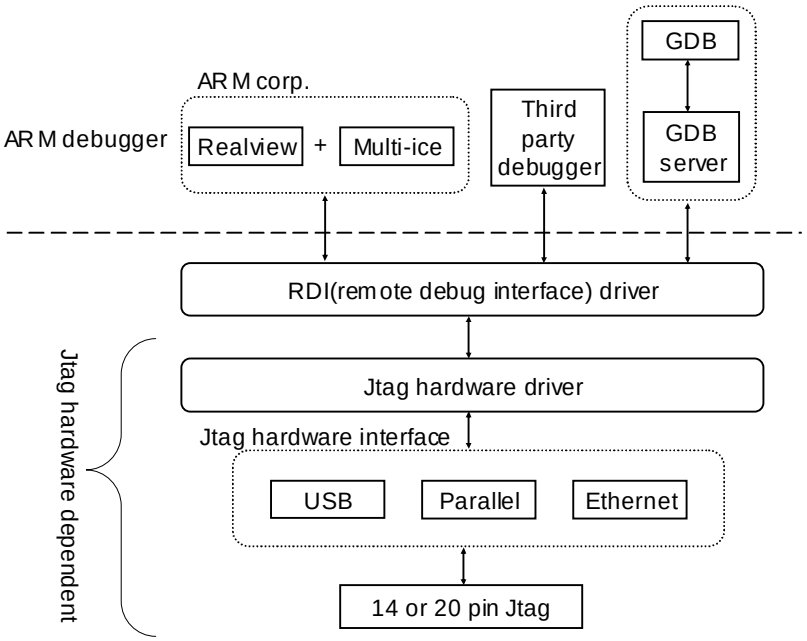

圖 2-5 Jtag-ICE 軟硬體運作關係圖

除錯軟硬體原理架構如圖 2-5 所示，ARM 公司制定了 RDI（remote debug interface）除錯 API 呼叫介面標準，遵循這個標準所開發的 Jtag 除錯硬體，即可搭配 ARM 原廠商業版 Multi-ICE 及 RealView 輔助開發環境的除錯器，由於市售 ICE 除錯軟體價格昂貴，故我們選用較經濟的 GDB+GDB server 當除錯環境，其運作模式為 client-server，GDB 透過序列或網路連接 GDB server，GDB server（或稱 multi-ice-gdb-server，[24]）82 將 GDB 的除錯命令轉成 RDI 格式，除錯器再將 RDI 命令轉成 Jtag 介面控制信號，達成以 TAP 控制 CPU 之單步執行、設置插斷點、讀取暫存器。正因為 Jtag 能直接控制 CPU 運行，所以除了透過它來進行除錯外，亦可透過它在線燒錄週邊的 NOR flash 記憶體。

2.4 開發工具與使用的軟體套件

開發本系統所需的軟硬體工具約十多種，我們選用的軟體簡列如下：

- GNU C compiler（v-3.4.3）：跨平台編譯器，將 C 原始碼編譯成 arm 指令集目的二元碼。
- Binutil-2.6：elf 及 aout 格式二元檔處理工具，如 ld、strip、nm、ar。
- uClibc 0.9.28：一套精簡型之 C 程式庫，非常適合嵌入式應用。
- GDB+GDB server：GDB 透過 RDI 使用 Jtag-ICE 除錯。
- Linux 2.6 source：Linux 原始碼，本論文移植此版本 Linux 至 LH79525 目標平台。
- Skyeye：架構於 GDB 下，支援 arm7 及 arm9 指令集，Skyeye 可以用來模擬 ARM 微控器（MCU），好處是在目標板未完成前，先用它來對移植的作業系統行為做初步驗證。
- busybox：簡易 Linux 系統工具套件，如：init、cp、rm、ifconfig 等基本的系統命令及公用程式。
- minicom：RS-232 終端模擬軟體，作 serial console（dumb terminal）終端機顯示，可接收與傳送資料給目標板。
- flash programmer：Jtag 燒錄 flash 工具，Wiggler（[31]）提供了並列埠轉 Jtag 的介面；Banyan-U（[32]）則是 USB 轉 Jatg 的介面。
- SDK-LH79524-3126-10：logicpd 公司（[5]）以 Sharp LH79524 MCU 的模擬板，用來驗證移植的作業系統。
- Sharp LH79524/5 BSP：BSP（board support package），提供驅動 MCU 週邊裝置呼叫所需的 API，藉由這些模組幫助了解週邊裝置運作細節。

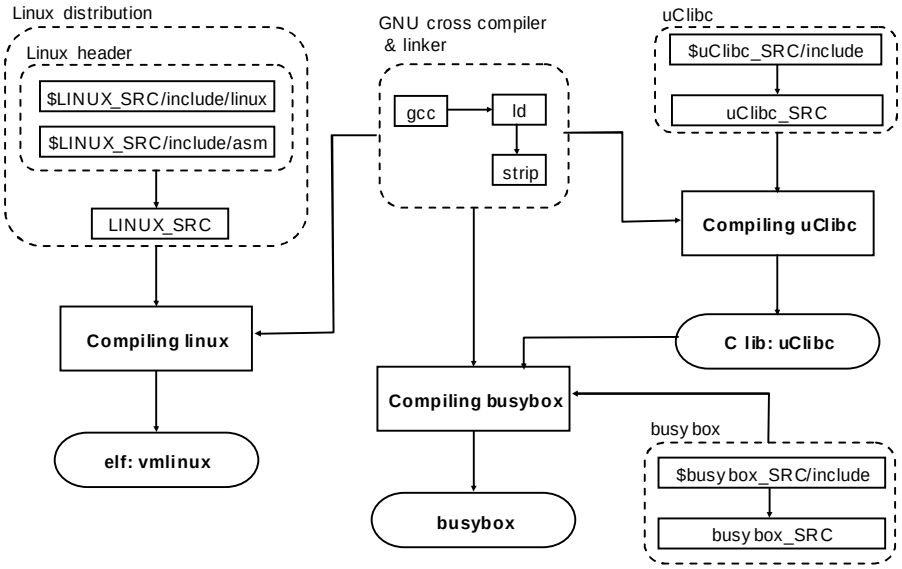

圖 2-6　Linux、uClibc、busybox 生成關係

圖 2-6 為 Linux、uClibc、busybox 生成關係，GNU 編譯器整合命令 gcc，gcc 是編譯器，整合呼叫連結器 ld 及外部連結程式庫路徑，若需去除程式中的非程式資料節區而只保留機器可執行碼，ld 則再調用 strip 以產生 RAW binary 執行檔。Linux 核心目地碼的生成不需其他額外的程式庫，uClibc 程式庫建立需要核心標頭檔目錄 **include/linux** 及 **include/asm**，uClibc 是一套標準 C 程式庫，佔用空間很小，適合嵌入式系統使用，busybox 生成則需 uClibc 程式庫標頭檔和連結程式庫。

第3章　機器架構平台與模擬板主要元件

在正式討論 ARM linux 移植過程前，我們先認識 LH79525 的硬體主架構及各項整合的週邊，這將對我們往下移植 ARM Linux 的硬體平台相依原始碼會有所幫助。

3.1　LH79525 微控制器

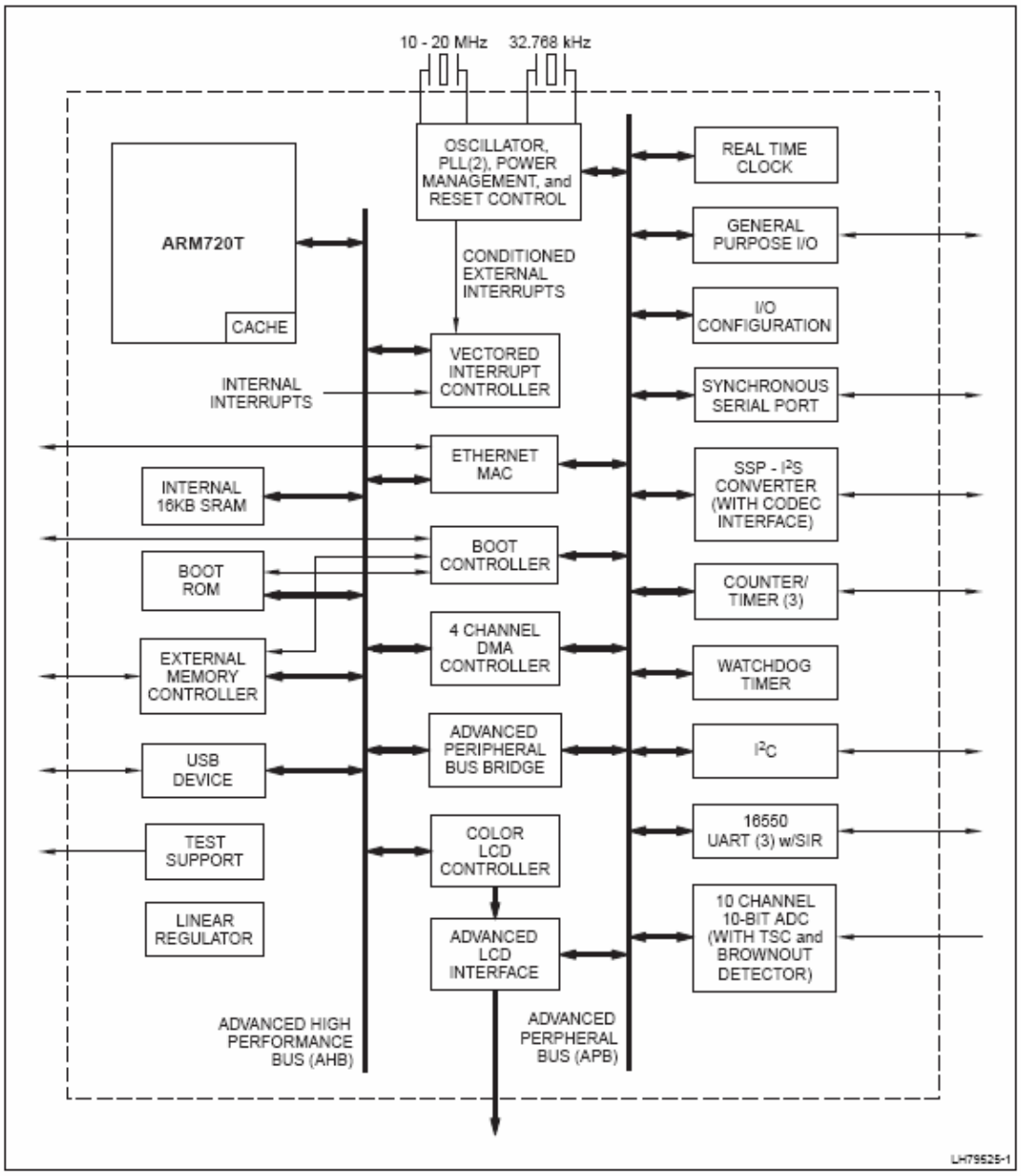

圖 3-1　LH79525 主要模組區塊

圖 3-1 所示為 LH79525 內部主要功能區塊圖，和其它廠牌的 ARM SOC 一樣，夏普公司需支付 ARM 公司核心處理器使用版權。AMBA（AHB 及 APB）（[10]）是 ARM 公司規範的匯流排介面，用以與各週邊控制器資料傳輸介面用，AHB 以連接高速資料傳輸的裝置（如乙太網 MAC、液晶控制器 CLCDC）；APB 則以連接低資料流的裝置（UART、GPIO、counter/timer），經由 APBB 與 AHB 連接。

3.1.1 ARM720T 組成單元

圖 3-2 說明 ARM720T 內部主要組成單元，ARM（Advanced RISC Machine）採用管線執行設計，指令截取單元從記憶體抓取指令，指令解碼單元將 32 位元長指令解碼產生控制信號，指令執行依控制信號控制運算單元、暫存器存取、記憶體存取。ARM 公司擁有 ARM 處理器矽智產權，大部份微控器（MCU）製造商向 ARM 公司購買 CPU 的 IP（intellectual property），再依 MCU 應用需求嵌入週邊控制單元。為了方便各家廠商整合週邊至 ARM CPU 核心，ARM 公司制定 AMBA（Advanced Microprocessor Bus Architecture，[10]），AMBA 匯流排協定規範了 AHB（Advanced High-Performance Bus）及 APB（Advanced Peripheral Bus）兩種匯流排，週邊需求高速資料傳輸則採用 AHB，否則用 APB 即可，APB 與 AHB 之間則透過 APB bridge 連接。

Tap（test access port，[9]）嵌入至 ASIC 內部主要是為了能夠由 Jtag 傳送測試向量（test vectors），Tap 定義了控制序向邏輯，通過 IEEE Jtag 埠標準傳送命令給 Tap 控制單元，達到控制晶片內部運作。ARM CPU 嵌入 Tap，能執行於硬體除錯模式，而一般的硬體除錯設備稱為 ARM Jtag-ICE，由 Jtag 埠將除錯控制命令送交 CPU。Jtag 也可用來下載資料至記憶體內，因此也能燒錄 flash，典型的應用例如燒錄場可規劃閘陣列（FPGA）。以 Jtag 燒錄 flash 需事先將 flash 存取時序做設定，以免速度過慢。（參照附錄 D）

MMU 及 8Kb cache 是 ARM720T 與 ARM7TDMI 唯一差異，內部暫存器 CP15 專用來設定 MMU 及 cache 等重要控制（[7]），藉由 MMU 來保護管理記憶體使系統程式不致於讓非使用者空間資料被破壞，任何非法位址存取硬體會產生例外處理。開啟進入 MMU 之前 ARM720T 處於真實位址模式，實體位址（PA）不經解譯直接送至位址匯流排；進入位址保護模式前，首先要建立分頁表，分頁表基底位址暫存器 TTB 指向分頁表的實際位址，最後切換至 MMU 模式。

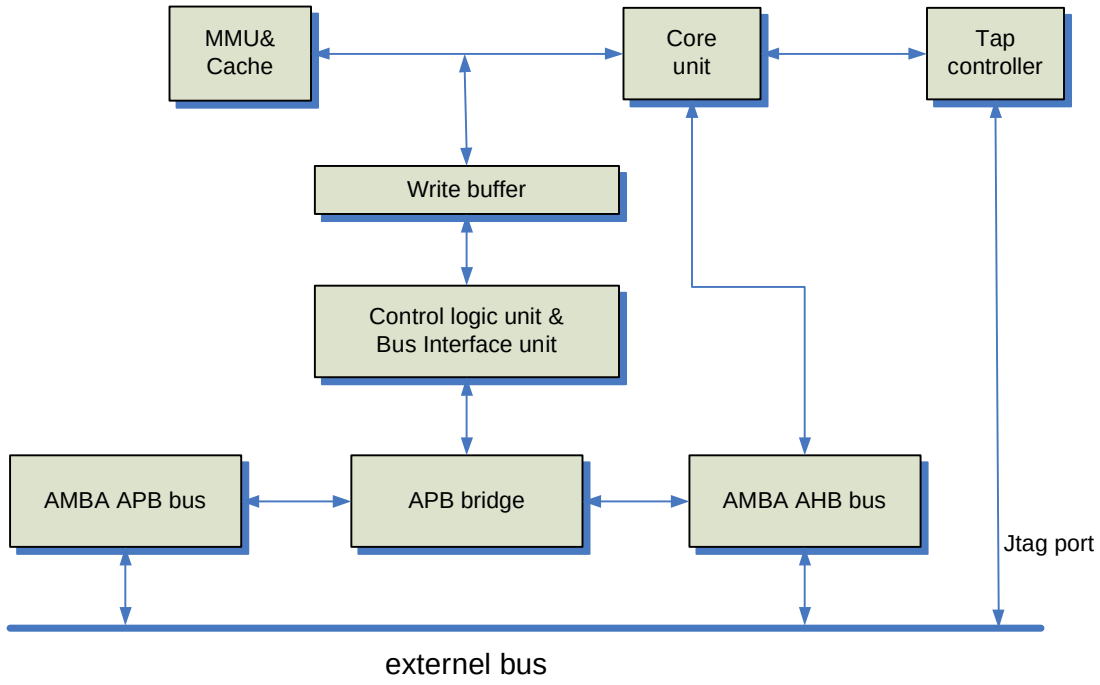

圖 3-2　ARM720T 處理器內部單元

3.1.2　外部記憶體控制器（EMC）

EMC 提供了存取各類記憶體時序控制，支援存取的記憶體有 flash、ROM、SRAM、SDRAM，以存取時序種類區分成同步與非同步記憶體。LH79525 的記憶體字組寬度只有 16 位元，但 ARM 每道指令長度 32 位元，需要耗掉兩次存取週期。圖 3-3 為 LH79525 各記憶體 bank 對應位址，位址 0x60000000 起始後的 16Kb 映射至 ARM720T 內部的 SRAM，根據 REMAP 值，位址 0 映射至

不同實體記憶體 bank，表 3-1 列出 LH79525 所支援的記憶體映射分佈，由映射控制暫存器 REMAP 選擇映射，在系統重置時 REMAP=00， nCS1 映至位址 0，而且程式指標暫存器 PC=0，因此 nCS1 應規劃至非揮發性記憶體，在本論文採用 NOR flash，將 bootstrap 程式以及 ARM Linux 核心映像檔及根目錄檔案系統映像檔燒錄至 NOR flash，如此系統重置時先執行 bootstrap 將開機準備任務完成，再將核心映像檔及根目錄檔案系統映像檔搬移至 SDRAM，映射 nDCS0 至位址 0，開始執行載入作業系統。

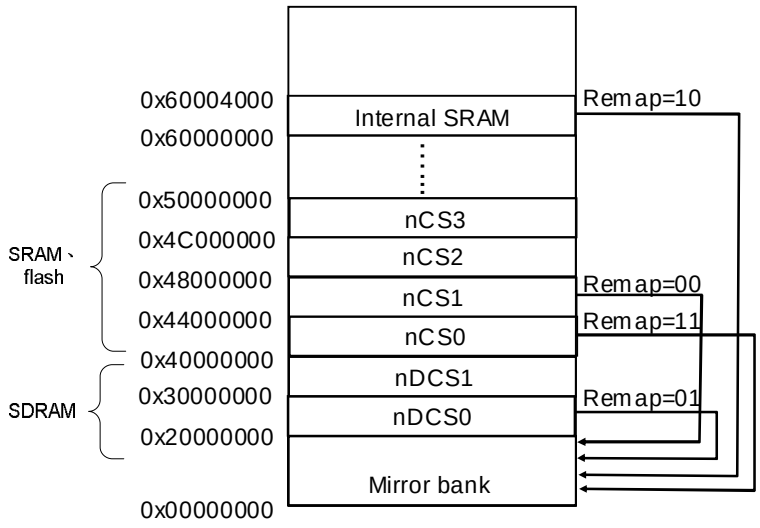

圖 3-3 LH79525 各記憶體 bank 對應位址

表 3-1 LH79525 記憶體映射分佈

ADDRESS	REMAP = 00	REMAP = 01	REMAP = 10	REMAP = 11
0x00000000 - 0x1FFFFFFF	nCS1	nDCS0	Internal SRAM	nCS0
0x20000000 - 0x3FFFFFFF	SDRAM	SDRAM	SDRAM	SDRAM
0x40000000 - 0x5FFFFFFF	Static Memory	Static Memory	Static Memory	Static Memory
0x60000000 - 0x7FFFFFFF	Internal SRAM	Internal SRAM	Internal SRAM	Internal SRAM
0x80000000 - 0x9FFFFFFF	Boot ROM	Boot ROM	Boot ROM	Boot ROM
0xA0000000 - 0xFFFBFFFF	Invalid Access			

3.1.3 非同步傳輸控制（UART）

UART 以 AMBA APB 介面與 CPU 連接，LH79525 提供三組 16C550 標準 UART， 16C550 UART 的主要功能與架構特色如下：

- 字組長度支援 5、6、7、8、9 位元；可選擇附加奇偶位元；停止位元長度 1 或 2 個位元；可程式化包率（baud rate）。
- TX 及 RX FIFO 各 32 組，可分別設定 TX 及 RX FIFO 中斷觸發準位。若寫出或讀入的字元不使用 FIFO，則比較容易發生 overrun。
- UART 可佔用中斷 IRQ，使用中斷以減低 RX 的 overrun 情況，及避免忙碌請求（polling）帶來系統效能衝擊。
- 支援 RX/TX loopback，方便測試 UART 是否正常。
- UART 硬體流量控制信號 RTS/CTS 交握以解決通訊雙方的流量控制，避免傳輸太快的情況而造成 overrun。

3.1.4 Real Time Clock、Watchdog Timer、Timer

RTC（Real Time Clock）應用於長時間計數，頻率單位為 1Hz，以 32 位元上數計數器。初始值設定至載入暫存器，中斷目標計數值記錄在比對暫存器，當啟用 RTC 的 IRQ，計數器的值增加至與比對暫存器相同時發生中斷。

看門狗計時器（Watchdog Timer）主要功能做系統監控，防止因系統當機造成長時間不工作，藉由控制 CPU 的重置信號，當按下重置扭或者因電源異常造成系統重置，看門狗計時器發生作用，若啟用看門狗計時中斷， 此時會先執行完 ISR 再進入系統重置。

系統計時器（Timer）則是每個微控制器必備的週邊，作業系統需要計時器中斷達成時間單位 tick 計數，使排程器得以運作。

3.1.5 Reset, Clock, Power Controler（RCPC）

Reset 來源有手動重置及 power-on 重置，重置信號致能時間太短可能導致

CPU 內部狀態未完全進入預設。CPU 於除錯模式由 TAP 控制單步執行，有兩種情況 TAP 要重置，一是 Jtag 重置信號 nTRST 致能，另一則是當 CPU 重置時，否則 TAP 可能停在未知的狀態。

LH79525 內部有兩組石英震盪電路，分別接受石英震盪輸入頻率 32.768KHz 及 10~20MHz，10~20M 這組用來驅動系統 PLL 及 UART，LH79525 的週邊所需 Clock 都是由系統 PLL 產生，透過 RCPC 暫存器調整 CPU 的工作頻率。32.768KHz 石英震盪器產生 1Hz 頻率供 RTC 使用。

3.1.6 Ethernet MAC（EMAC）

LH79525 整合了相容於 IEEE 802.3 乙太網 MAC 層，主要標準架構有：

- 支援 MII 介面與 PHY 作溝通。
- 自動過濾錯誤的資料訊框（frame），包裝資料訊框及 CRC 檢查。
- 支援 10Mb 及 100Mb 速率，全雙工及半雙工。
- 提供四組可程式化 MAC 位址暫存器。
- Hash 比對單點播送及多點播送位址。
- 傳送與接收 fifo 緩衝區與系統主記憶體透過 DMA 傳輸。
- 支援正常 MII 模式及序列網路介面（SNI）模式。

圖 3-4 為 EMAC 內部模組方塊圖，MII 包含傳送與接收模組分開獨立運作，可以在全雙工或半雙工下運作，除此之外也負責了碰撞偵測、載波與傳輸錯誤偵測。ETHERMDIO 是序向埠，MII 通過此埠讀寫 PHY 控制暫存器。

傳輸資料前，必須在主記憶體中規劃一塊 TX 及 RX 緩衝區。當傳送封包時，先把欲傳送的封包放入 TX 緩衝區，對 EMAC 下傳送命令，EMAC 透過 DMA 將 TX 緩衝區的資料搬至 TX FIFO 再送出；反之，當 EMAC 接收 frame，首先比對 MAC 位址是否穩合，資料是否有錯誤，沒錯誤則放入 RX FIFO，DMA 再將資料搬至主記憶體，啟動中斷觸發讓 CPU 執行 ISR 資料從 RX 緩衝

區讀走。

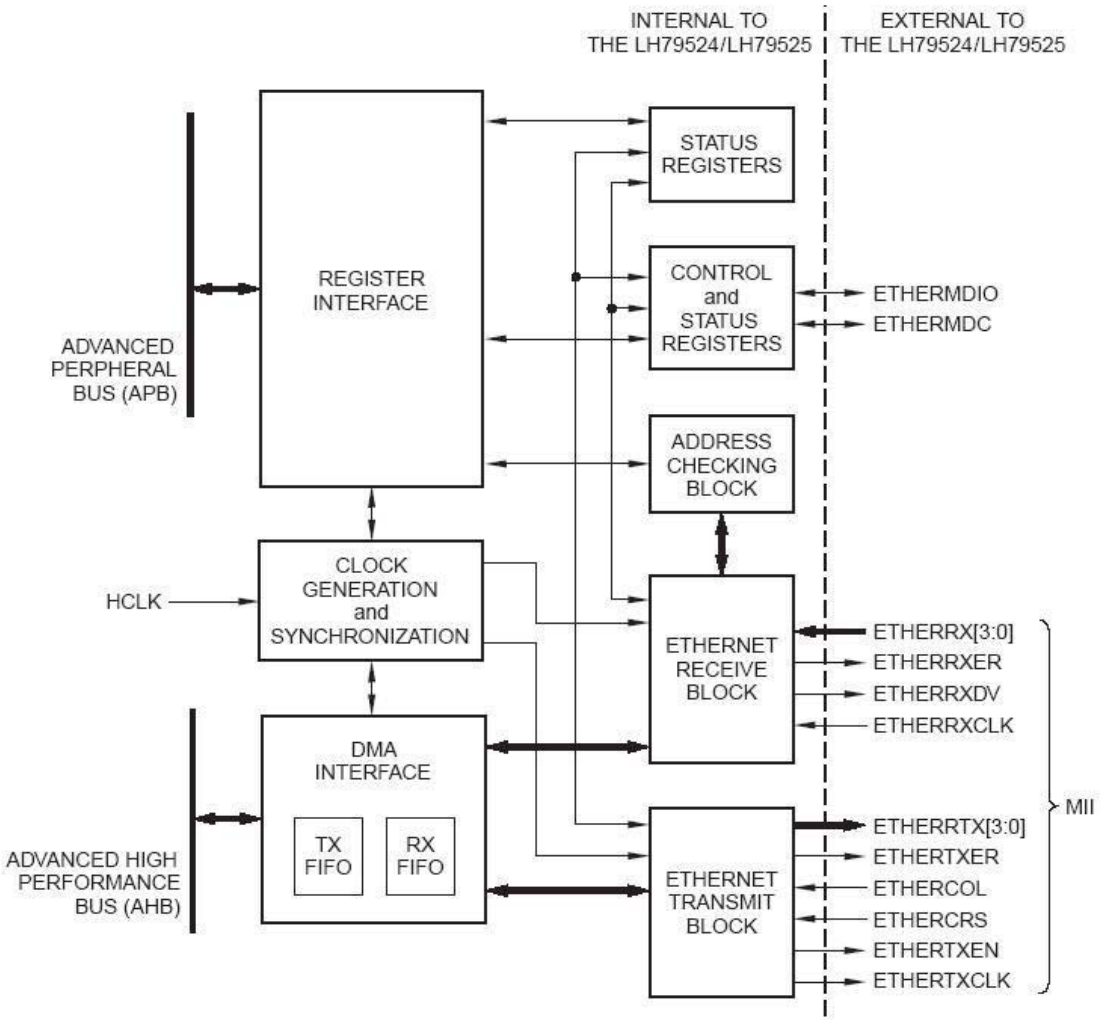

圖 3-4　LH79525 EMAC block diagram

3.2　Flash Memory

快閃記憶體是 EPROM 及 EEPROM 兩者技術結合，快閃記憶體的抹除以區塊為單位，但是 EEPROM 則是要每個位元組單獨抹除，因此快閃記憶體抹除及寫入較 EEPROM 有效率。而且快閃記憶體容量比 EPROM 及 EEPROM 大很多，寫入抹除不用額外使用燒錄設備，在系統中直接由 CPU 執行燒錄程式即可，因此更新資料方便，已成為嵌入式系統主要儲存元件。

3.2.1　NAND v.s. NOR flash

Flash Memory 主要可以分為兩類，分別是 NAND Flash 跟 NOR Flash，兩個主要的差別請見表 3-2。由於 Flash Memory 的特性是不允許對同一塊相同的記憶體位置連續寫入一次以上，除非事先對那塊記憶體位置做清除(erase)的動作。

表 3-2　NOR v.s. NAND flash 比較

項目	NOR v.s. NAND 差異比較說明
讀取速度	NOR 比 NAND 快，但 NAND 採用序列存取方式，節省硬體上成本。
抹除與寫入	NAND 每個單元（cell）電路較小，抹除速度比較快，其寫入與抹除區塊典型為 512 位元組；NOR 抹除區塊一般有 128k、64k、32k 三種。
存取控制腳位	NAND 序列存取控制只有 8 個腳位，其 I/O 控制較複雜；NOR 則是與一般非同步 SRAM 有相同存取介面時序規格。
製造成本	NAND 單元硬體耗用面積大小約為 NOR 一半，相同容量的 flash，NAND 價格上會比較便宜。
可靠度與耐用性	NAND 的位元錯誤率比較高，錯誤偵測與改正（EDC/ECC）做資料修復；NAND 最大插寫次數約百萬次，NOR 則約為十萬次。
市場上主要應用	NAND Flash 還是著重在大容量的儲存設備，而 NOR Flash 則是在小容量且需要快速讀取的產品上較常見。
在地執行（XIP）	NAND 的讀取需要有驅動軟體支援，因此 NAND 不能直接運行地執行，必須先將程式搬至 RAM 中。

圖 3-5 為 NAND 和 NOR flash 的積體電路內部佈局（[47]），很清楚看到 NOR 在每兩個電晶體單元間都加了擴散層金屬接觸點（diffusion contact），佔

用較多的面積，因此其電流導電性比較好，NOR 的讀取速度比 NAND 快，而且資料正確性較高，但相對的 NAND 佔用面積比 NOR 小很多。AMD 及 INTEL 兩大全球快閃記憶體製造商乃採用 NOR 單元設計。

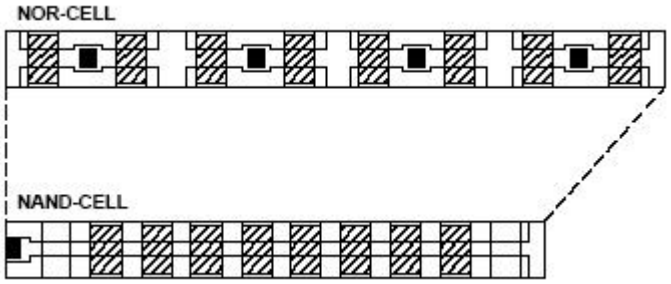

圖 3-5　NAND v.s. NOR flash cell layout

為了讓 Flash Memory 可以在原有作業系統底下運作，例如在原來的檔案系統(FAT16/32、NTFS、EXT2)下運作，常被採用的方法是建立位址轉換表（Address Translation Table）的方法，此方法是藉由把邏輯區塊位置(Logical Block Address)對應到真實的 Flash Memory 的位置，我們可以把 Flash Memory 模擬成是一個連續的記憶體空間，因此就可以在不變動原有的設定下，把 Flash Memory 當做是一個 Hard Disk 裝置使用，此種方法稱之為轉譯層「Flash Translation Layer (FTL)」。

MTD 子系統層（[28]）為 Linux 提供 flash 的操作方法，區分使用者及驅動模組，驅動模組負責偵測、讀寫、抹除方法，使用者模組提供了轉譯層方法，使得 flash 分割區能被模擬成區塊裝置使用，在 4.9 節我們將探討說明 MTD 子系統層如何運行於 ARM Linux。

3.2.2　NOR flash：MX29LV640BT

本論文模擬板使用 MX29LV640BT（[29]），大小 4Mbx16 位元，抹除區塊單位大小為 32kx16，在最後一塊抹除區又進一步分割成 4kx16，支援軟體 CFI（common flash interface）介面操作，方便驅動軟體撰寫（[40]）。

NOR flash 硬體介面和 SRAM 一樣，讀取時序雖然都只需一個存取週期，但在寫入或抹除 NOR flash 需要經過多個存取週期來完成序向命令，參照[29]中的 Table3 為 MX29LV640BT 定義的 flash 操作命令，依廠牌或型號其製造商 ID 及裝製 ID 或許不同，但操作命令主要不同在於 AMD 和 INTEL 兩大規格，兩者均提供了 CFI 相容，JEDEC 制定了 CFI 共通介面（[40]），讓驅動程式發展者不需針對不同 flash 命令改寫不同的程式碼。

3.3 SDRAM

0x20000000~0x3fffffff 為 SDRAM 映射位址（圖 3-3），分別為 nDCS0、nDCS1 兩個 bank。SDRAM 將位址線區分成行、列位址，使用 SDRAM 前必需先設定外部記憶體控制器（EMC），根據使用的 SDRAM 記憶體大小選擇正確的行、列位址腳位映射（參照[13]中的 Table7-2~5）；根據選用的 SDRAM 型號規格書設定更新時間週期，256MB 需要完成 64ms/8k，以及設置存取時序參數。SDRAM 晶片使用前也必需對內部初始化，使其模式進入 idle 狀態，詳細 SDRAM 使用請參照附錄 C。

3.4 ARM 處理器程式設計者模型

3.4.1 記憶體格式

圖 3-6 及圖 3-7 分別是兩種不同記憶體多位元排列格式，ARM720T 可透過 CP15 暫存器修選擇字組格式。兩者並沒有絕對特別優勢，在一些應用上各有採用，一般以 little-endian 較廣為使用。一般 OS 原始碼中為了同時支持這兩種記憶體格式，以巨集定義區分開這兩種不同的記憶體操作。

3.4.2 暫存器群

ARM720T 有 ARM 及 Thumb 兩種不同長度指令集，ARM state 指令集長度 32 位元，Thumb state 指令集長度 16 位元，處理器處在這兩者不同狀態下的可視暫存器如圖 3-8，ARM 與 Thumb 互相切換時不影響處理器目前的模式，除非對空間需求很在意，一般使用不會做切換。

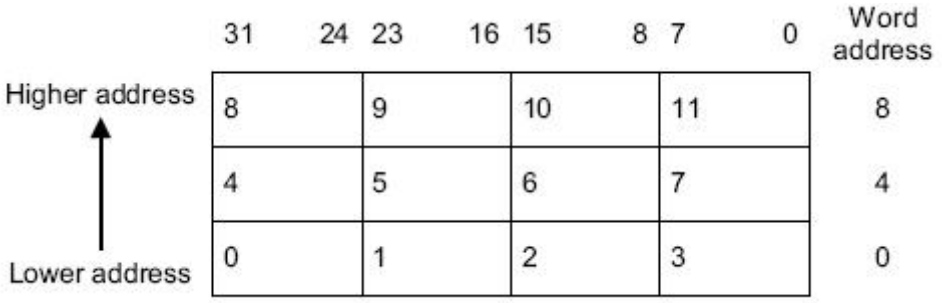

圖 3-6 big-endian 記憶體排列次序

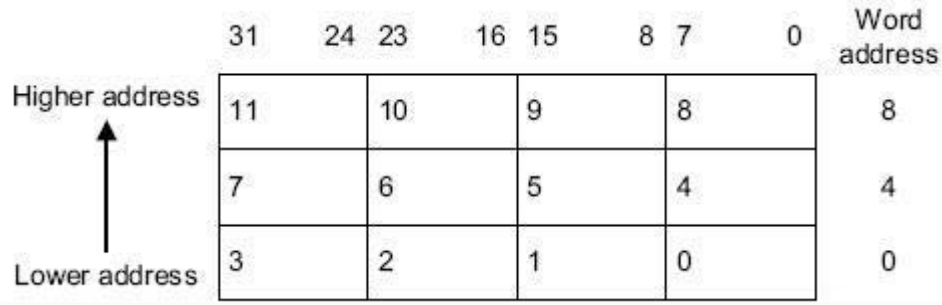

圖 3-7 little-endian 記憶體排列次序

ARM720T 有 17 個暫存器，R0~R12 用作一般用途，R13（SP）用作堆疊暫存器，R15（PC）則是程式指位暫存器，R14（LR）連結暫存器，CPSR 目前程式狀態暫存器，當 CPU 進入例外處理，SPSR 則是用來儲存 CPSR，圖 3-9 可看見 ARM 七種不模式下的可見暫存器有所不一樣，尤其是特殊功用暫存器。

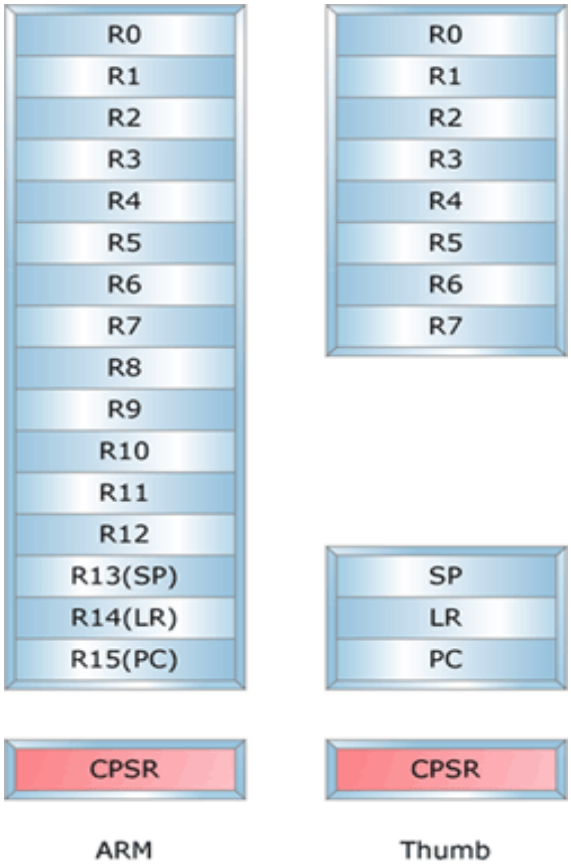

圖 3-8　ARM/Thumb 下的可視暫存器

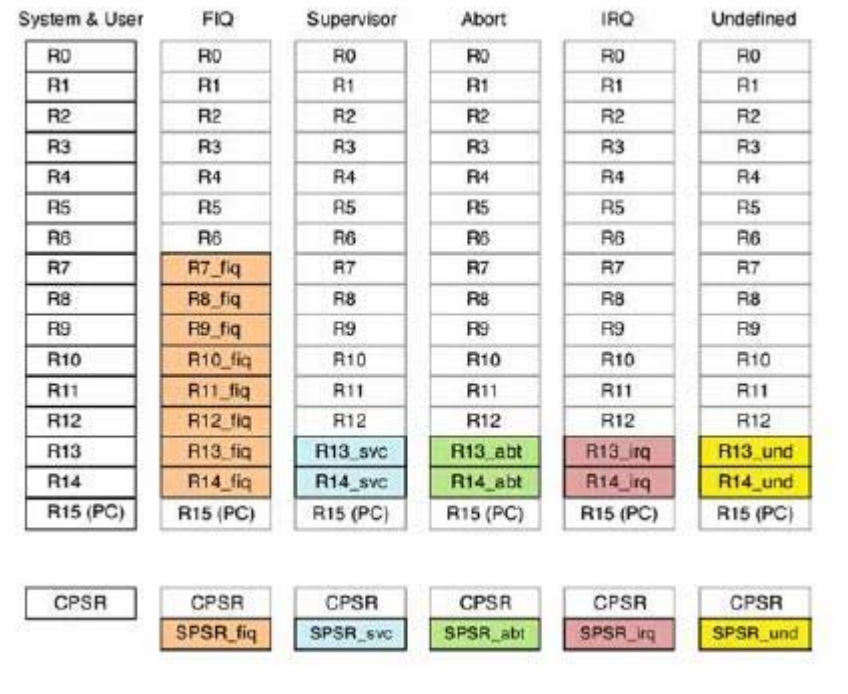

圖 3-9　各種模式下的暫存器

3.4.3 程式旗標暫存器

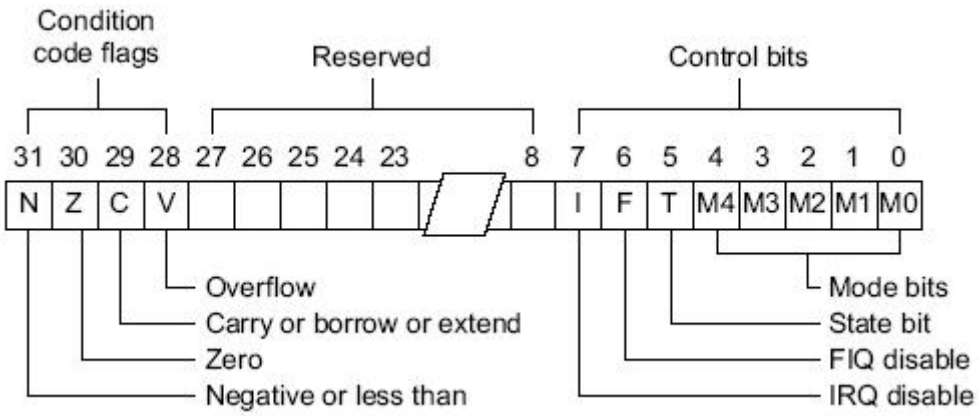

圖 3-10 CPSR 程式旗標暫存器

圖 3-10 為 ARM 處理器執行時期的程式旗標暫存器

- N、Z、C、V 位元：運算結果為負數，MSB=1 則 N=1，運算結果為零則 Z=1，發生進/借位則 C=1，運算結果發生溢位則 V=1。
- I、F：CPU 中斷致能控制旗標。
- T：ARM/Thumb 狀態，ARM state 指令集長度 32 位元，Thumb state 指令集長度 16 位元。
- M4~M0：CPU 工作模式，User、FIQ、IRQ、Supervisor、Abort、Undefined、System。

3.4.4 ARM 處理器模式與例外處理

ARM 處理器定義了七種例外事件，例外處理向量表存放例外處理函式進入位址，每種例外處理相對應於一種特有的處理器模式，當 CPU 遇到例外事件，CPU 跳至向量表對應的進入位址執行例外處理函式（表 3-4），當執行完畢欲返回例外事件發生點後下一道指令繼續執行，例外處理程式返回指令請參照表 3-5 處理器結束指令。表 3-3 簡略列出這七種例外處理事件，以及發生的情況場合、CPU 進入模式。

表 3-3　例外事件處理及進入模式、優先順序

Exception	Priority	Entry mode	Meanning
Reset	1（highest）	Supervisor	System reset
Undefined	6	Undefined	Fetch an undefined instruction
SWI	6	Supervisor	Software interrupt（執行 SWI 指令）
Prefetch abort	5	Abort	Prefetch access memory violation
Data abort	2		Data access memory violation
IRQ	4	IRQ	General interrupt request
FIQ	3	FIQ	Fast interrupt request

表 3-4　例外處理向量表

Label	Exception	Low address	High address
Vector_reset	Reset	0x00000000	0xffff0000
Vector_und	Undefined	0x00000004	0xffff0004
Vector_swi	Software interrupt	0x00000008	0xffff0008
Vector_pabt	Prefetch abort	0x0000000c	0xffff000c
Vector_dabt	Data abort	0x00000010	0xffff0010
Vector_irq	IRQ	0x00000018	0xffff0018
Vector_fiq	FIQ	0x0000001c	0xffff001c

表 3-5　例外處理結束指令

Exception	Return instruction	Previous state	
		ARM R14_x	Thumb R14_x
Undefined	movs PC, R14_und	PC+4	PC+2
SWI	movs PC, R14_svc	PC+4	PC+2
Prefetch abort	subs PC, R14_abt, #4	PC+4	PC+4
Data abort	subs PC, R14_abt, #8	PC+8	PC+8
IRQ	subs PC, R14_irq, #4	PC+4	PC+4
FIQ	subs PC, R14_fiq, #4	PC+4	PC+4

第4章　移植 Linux 作業系統

4.1　Linux kernel 2.6

4.1.1　Linux 核心發展歷史

UNIX 作業系統最早發展於 AT&T 貝爾實驗室，1983 年 UNIX SVR1 發表，直至 1989 年 System V Release 4（SVR4）版本問世，逐漸成為 UNIX-like 作業系統的標準規範。GNU 計劃於 1984 年開始進行，主要目地是開發一套完整的自由軟體作業系統（稱 GNU 系統），由於這些系統程式在 UNIX 下開發，因此廣泛被移植散播，至 1990 年 GNU 系統幾近完備（編譯器 gcc、程式庫 glibc、除錯器 gdb…等工具），獨缺遺漏的部件是內核（kernel）。幸運的是，1991 年 Linus Torvalds 於新聞群組 comp.os.minix 散播他開發的一套 UNIX 相容核心，這套核心名為 linux，初版發表硬體平台為 intel 386，隔年 GNU 系統與 linux 核心成一套完整的自由軟體作業系統，GNU/linux 成為 GNU 作業系統完整名詞，以表達它是 GNU 系統和以 Linux 作為內核的組合。

Linux-2.6 發表版支援常見的處理器架構平台如下：arm、i386、ppc、mips、alpha、m68k…等數十種，每種處理器平台架構又可細分不同機器平台。

4.1.2　Linux 核心組成主要部份

作業系統區分核心層與使用者層（圖 4-1），核心層包含例外處理函式、程序管理、記憶體管理、裝置驅動程式、檔案系統、系統呼叫（API）。使用者透過 Shell 與核心做溝通，使用者無法直接對硬體做直接存取，必須透過驅動程式提供硬體運作方法（讀寫、IOCTL），Linux 將硬體週邊裝置抽象化成裝置設備檔名稱（/dev/xxx），裝置設備檔的結構型別提供裝置的主編號及次編號，系統乃根據這組編號正確調用驅動程式，操作週邊裝置就如同操作檔案，透過檔案處理的系統呼叫，調用正確的運作方法。（[22]）。

Linux 核心原始碼定義各種資源管理標頭檔（*.h），這些抽象資源的描述子（descriptor）詳細紀錄各管理資源狀態及函數操作方法，描述子的維護則是透過呼叫函式操作方法。

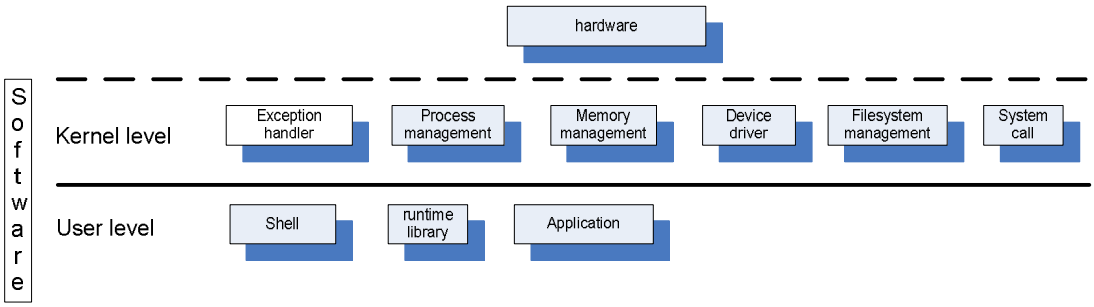

圖 4-1　作業系統之核心層與使用者層

4.1.3 Linux 核心的分層次概念

Linux 核心碼可區分成處理器架構與非架構相依（圖 4-2），同一處理器架構又可區分成硬體（機器相依）與非硬體相依。非架構相依的核心碼如系統呼叫（含 socket）、系統排程機制、虛擬記憶體管理。架構相依規範了例外處理、CPU 的匯流排規格（寬度及仲裁機制）、定址能力、內部暫存器群（register file）、MMU 管理單元、記憶體格式大小位元組排列（big/little endian），

若把核心碼以同心圓來描述（圖 4-3），Linux 原始碼發展的次序區分層為非架構相依、架構相依、機器相依，移植 Linux 至某種平台，必先確定處理器的架構、機器架構相依項目，我們把各發展層次發展項目與困難度分列表比較（表 4-1），第 1 層移植內容與任何處理器非架構相依困難度最高，也是 OS 機制決策演算的精神核心，約等同開發一套新的 OS。第 2 層處理器架構相依困難度次之，在 Linux 核心發表版內已經支持主要常用的處理器架構，第 3 層機器相依則是普遍最常碰到的移植，也是產品開發應用者最常面臨的部份。本論文移植 ARM Linux 支援 Sharp LH79525，由於 Linux Kernel 2.6 發表版已支援 ARM720T，因此我們的移植角色偏向定位在第三層。

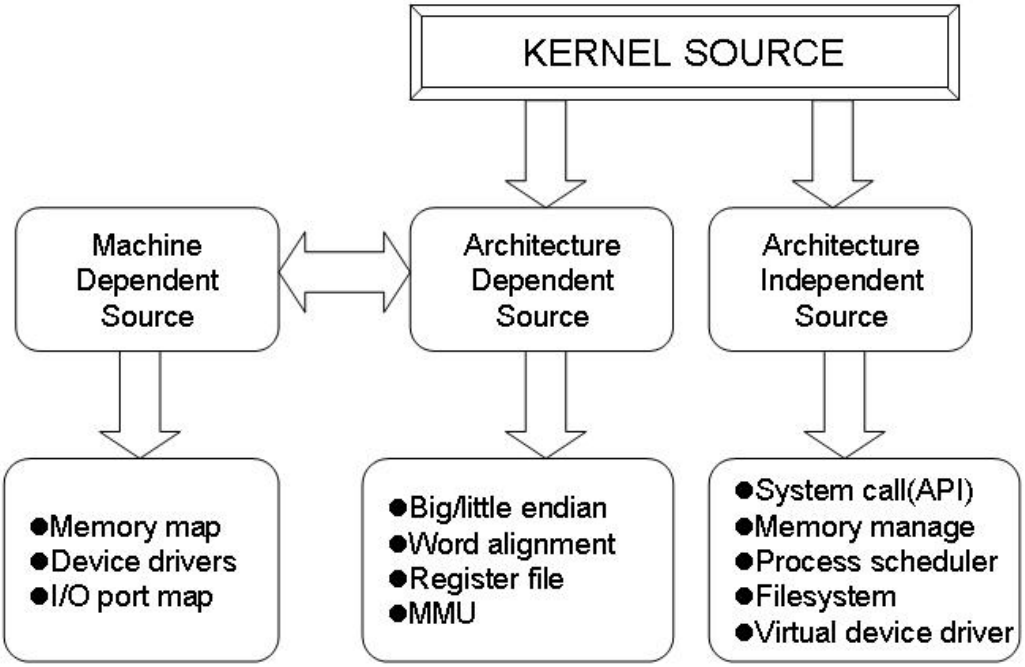

圖 4-2　核心碼的架構相依與機器相依分層

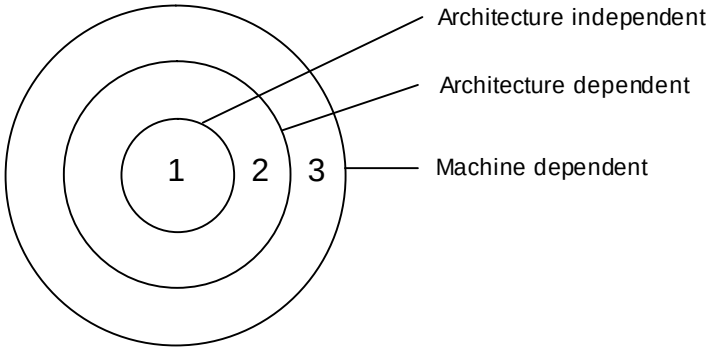

圖 4-3　核心碼與硬體分層關係

表 4-1　Linux 核心移植分層

發展層次	已有層次	發展項目內容	困難度
1	N/A	全新的開發一套OS	最高
2	1	移值支援特定處理器架構	次之
3	1, 2	移值支援特定微控制器	最低

4.1.4　Linux 原始碼目錄

移植前必需了解各主要原始碼目錄，以清楚主要架構相依、機器相依及各項管理資源的檔案所在目錄，區分擺放位置才不會顯得凌亂。每層存放原始碼的子目錄其編譯規則由 Makefile 描述，核心組態參數（CONFIG_XXX）於 Makefile 含入使用。圖 4-4 原始碼目錄中，＜arch＞實際代換成處理器架構名稱，＜mach＞則代換成機器名稱，移植新的平台必須把新增的原始碼檔案放到這些特定地方，並定義新增的組態參數，修改編譯規則檔。

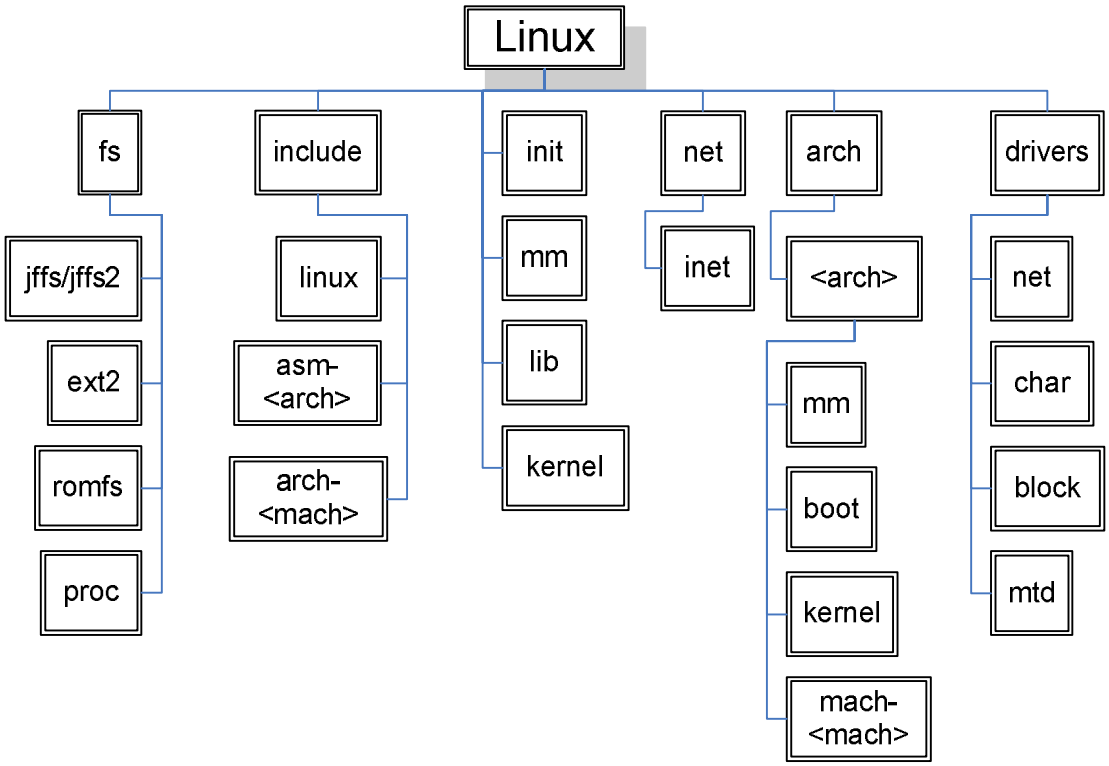

圖 4-4　Linux 核心目錄

表 4-2　核心目錄說明

目錄階層	重要功能簡述
include/asm-<arch>	處理器架構平台相依的標頭檔
include/asm-<arch>/ arch-<mach>	機器平台相依標頭檔定義，移植新mcu時必需新增此目錄
include/linux	相關於核心資源管理的標頭檔
arch/<arch>	處理器架構平台相依原始碼
arch/<arch>/mach-<mach>	機器平台相依的原始碼，移植新mcu所必需新增此目錄
driver	週邊裝置的驅動原始碼，移植新mcu必需為週邊硬體新增適當的驅動程式至此目錄下
net	網路協定、系統呼叫API（含網路socket）
mm	與硬體無關的記憶管理原始碼
kernel	主要核心原始碼，程序、訊息傳遞、timer tick等原始碼
lib	核心原始碼調用到的程式庫
init	C原始碼最上層，main.c裡的*start_kernel*函式為核心碼最關鍵進入點
ipc	interprocess之間訊息傳遞管理
fs	各種檔案系統管理運作方法原始碼

4.2　移值 ARM Linux 的一般化步驟

我們將移植 ARM Linux 的一般化條列式步驟敘述如下：

- 取得 Linux 原始碼，解開原始碼根目錄$（LINUX），複製一份原始碼至目錄$（LINUX_PORT），移值新增或修改的檔案於$（LINUX_PORT），將來用 *diff* 工具產生所移值平台的修補檔。
- 命名機器平台"XXX"，新建機器平台相依目錄 **include/asm/arch-XXX**，以及原始碼目錄 **arch/arm/mach-XXX**。修改 **arch/arm/Makefile**，加入機器平台變數及其核心組態參數（型式為 machin-$（CONFIG_MACH_XXX）+=XXX）。
- 註冊機器平台 ID 於 **arch/arm/tools/mach-types**。
- 調整重要的核心符號之巨集定義，設定開機時核心及根目錄檔案係統

解壓載入目地位址於 **arch/arm/mach-XXX/Makefile.boot**。

- 新增處理器架構描述子定義於 **arch/arm/mm/proc-armXXX.S**，並加入處理器相依的設定原始碼，修改 Kconfig 及 Makefile。
- 新增機器平台描述子定義、計時器初始化、中斷控制器初始化及中斷資源管理描述子等原始碼於 **arch/arm/mach-XXX** 下，新增核心組態選單設定檔 Kconfig 及編譯定義檔 Makefile。
- 新增驅動程式模組至 drivers 目錄下，修改 Kconfig 及 Makefile。
- 新增機器平台相依的硬體資源常數及重新巨集定義，包含控制暫存器、IRQ 定義、RAM 及 Flash 實際起始位址、實體與虛擬位址映射，這些定義撰寫於 **include/asm/arch-XXX** 內的標頭檔。
- 執行核心安裝編譯步驟。

機器平台相依的原始碼及標頭檔目錄內重要的修改內容分列說明如下：

- arch/arm/Makefile<modified>：新增機器平台的核心組態參數 **CONFIG_ARCH_XXX**，並加入以下編譯規則修正，XXX 為機器名稱。

```
ifeq ((CONFIG_ARCH_XXX), y)
   MACHINE = XXX
endif
```

- arch/arm/boot/Makefile<modified>：修正核心解壓縮擺放目地位址，以及解壓縮程式起始位址。

```
ifeq ((CONFIG_ARCH_XXX), y)
   ZTEXTADDR= XXXX8000
endif
```

- arch/arm/mach-XXX/Makefile<new>：把此目錄的原始碼加入編譯規則

檔內。

```
obj-y += <all source module object>
```

- arch/arm/mach-XXX/arch-XXX.c<new>：定義機器平台描述子，系統保留的 IO 位址映射初始化。

```
MACHINE_START(KEV79525, "XXX")
MAINTAINER(“Ju-Chin Tsai”)
BOOT_MEM(XXX, YYY, ZZZ)
BOOT_PARAMS (0x20000100)
MAPIO(XXX_map_io)
INITIRQ(XXX_init_irq)
.timer =XXX_timer
MACHINE_END
static struct map_desc XXX_io_desc[] __initdata =
{    IO BASE, IO START, IO SIZE, DOMAIN IO, 0,1,0,0,
……
};
void __init XXX_map_io(void)
{
   iotable_init(XXX_io_desc);
}
```

- arch/arm/mach-XXX/irq-XXX.c<new>：機器相依的中斷控制器初始化，中斷優先次序設定，初始化核心的中斷資源管理描述子。

```
static void XXX_vic_mask_irq (u32 irq)
{
   /* Adding the code to mask IRQ */
```

```
}


static void XXX_vic_unmask_irq (u32 irq)
{
   /* Adding the code to unmask IRQ */
}


static struct irqchip XXX_vic_chip = {
      .ack = XXX_vic_mask_irq,
      .mask      = XXX_vic_mask_irq,
      .unmask   = XXX_vic_unmask_irq,
};
void __init XXX_init_irq(void)
{
   /* Initial VIC and interrupt resource descriptor */
   for (irq = 0; irq < NR_IRQS; ++irq) {
            set_irq_chip (irq, &XXX_vic_chip);
            set_irq_handler (irq, do_level_IRQ);
            set_irq_flags (irq, IRQF_VALID);
   }
}
```

- arch/arm/mach-XXX/timer-XXX.c<new>：計時器初始化，調整計時器每秒鐘的中斷觸發頻率 HZ 巨集常數可控制排程器活躍度。
- include/asm/arch-XXX/hardware.h<new>：機器平台相依的硬體資源常數定義。

- include/asm/arch-XXX/io.h<new>：機器平台相依的 IO 標頭檔。
- include/asm/arch-XXX/irq.h、irqs.h<new>：機器平台相依的 IRQ 標頭檔。
- include/asm/arch-XXX/memory.h<new>：機器平台相依的記憶體標頭檔。
- include/asm/arch-XXX/system.h<new>：定義 arch_idle 及 arch_reset 函式。
- include/asm/arch-XXX/timex.h<new>：機器平台相依的計時器標頭檔。

4.3 Bootloader 與作業系統載入初期初始化

Bootloader 於任何機器平台扮演第一個執行程式，它的主要任務如下：

- 初始化 CPU 的工作頻率，通常 MCU 內部將外部石英震盪的頻率當基頻，以 PLL 方式產生內部所需的各種頻率，透過控制暫存器來修改調整 CPU 工作頻率。
- 建立和初始化 RAM：探索 RAM 起始位置、大小，正確設定 SDRAM 及 SRAM 控制參數。外部記憶體控制器支援不同種類記憶體介面，初始化記憶體參數用以選擇正確記憶設定，調整存取時序，提高效能。
- 初始化 UART（RS-232）：設定 UART 控制器的 Baud rate、parity、stop bits、flow control、fifo。當作終端操控台，扮演基本輸出輸入角色，由 RS-232 傳輸資料至系統的 RAM。
- 提供基本操作目標板的功能，供使用者存取目標板資源，如查看或修改記憶體內容，從主機下載資料至目標板。
- 檢測機器的系統結構：檢查 MCU ID 及 ARM CORE 型號。

- 傳遞核心參數，並將 Linux 核心映像與根目錄檔案映像解開搬至 RAM，開始執行 Linux 核心碼，至此 Bootloader 任務完成。

Linux 執行初期有一部份任務完成機器相依的設定，即正式進入核心啟始點 *start_kernel* 函式前的所有前置工作，這些初始化處理如下：

- 檢測機器的系統結構：檢查 MCU ID 及 ARM CORE 型號。
- CPU 初始化設定，切換 CPU 工作模式，開啟指令和資料 cache。
- 若 Linux 需支援 MMU，則要設定分頁表及分頁表基底暫存器、MMU 控制暫存器。

4.4 Linux 驅動程式運行原理

4.4.1 檔案裝置型驅動程式主要功能

檔案裝置型驅動程式為區塊或字元裝置提供運作方法，圖 4-5 列舉了主要模組，在作業系統載入階段，驅動程式的初始化函式將驅動程式註冊至核心，中斷服務程式（ISR）註冊至核心裡的中斷資源管理描述子；裝置開啟、關閉與讀寫存取、IOCTL 為裝置的操作函式。核心的裝置描述子陣列（分區塊與字元

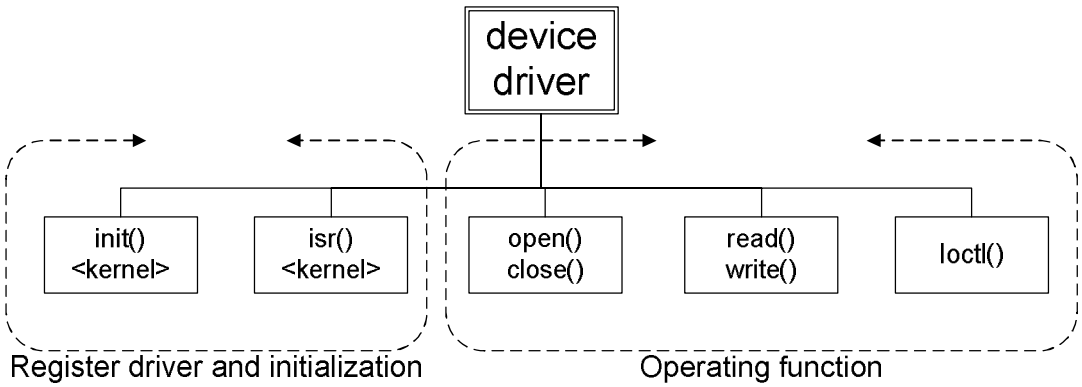

圖 4-5 檔案裝置型驅動程式

）記錄每一個裝置的名稱與一組代碼、操作函式指標進入點；如裹該裝置配有中斷請求，則該裝置的中斷編號（IRQ）及中斷服務程式（ISR）必須在初始化時向核心註冊。（[22]）

區塊裝置每次存取最小單位為 block（一般為 512 bytes 之倍數），磁碟儲存設備一般都是區塊裝置；字元裝置以位元組為存取單位，一般 teletype 型態的裝置屬於此類（序列埠、虛擬操控台）。

在目錄/dev 下提供裝置設備檔名稱，它的結構含有裝置檔型態(b/c)及代碼（major/minor number）。使用者透過檔案系統呼叫（open、read、write、ioctl、close）來操作裝置，其原理便是從裝置檔名(/dev/xxx)找出對應的代碼，由此代碼索引裝置描述子底層驅動程式。

4.4.2 網路裝置區動程式

網路卡 MAC 層驅動程式既非區塊亦非字元裝置，因此網路介面並不依附在檔案系統下，撰寫網路 MAC 層驅動程式和一般驅動程式沒兩樣，要將網路卡的介面註冊至適當的資料結構。Linux 網路子系統分層為與協定相依與非協定相依，而網路驅動程式屬於非協定相依，網路驅動程式處理封包時可以不用管協定如何運作，而協定也用不著顧慮實際傳輸的過程。每一個網路介面存在一個識別名稱 **eth*n***，應用程式以此識別名透過 socket 系統呼叫操作網路介面。

4.4.3 驅動程式與使用者層之關係

圖 4-6 說明了使用者層的應用程序與核心之間的關係，使用者程序調用系統呼叫 API 與驅動程式做溝通。裝置的被動操作（如讀取或接收）由該裝置提供的中斷服務程式先把外部進來的資料先放置緩衝區，使用者再從緩衝區讀走裝置的資料。

當我們直接對磁碟裝置使用 dd 指令寫入或讀出，此時的存取是以 RAW 資料流形式運作（資料未再經解譯）；當磁碟裝置被掛載（mount）成目錄，磁碟裝置必須格式化檔案系統（romfs、ext2、jffs2）。

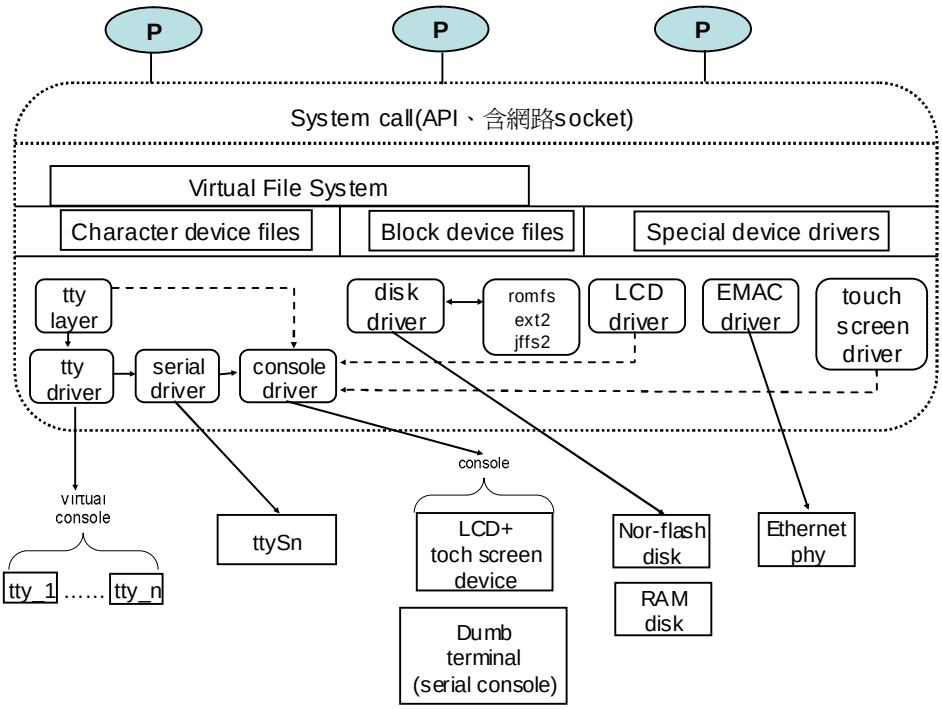

圖 4-6　使用者程序與驅動程式間關係

4.4.4　新增驅動程式模組至 Linux 核心

當移植 Linux 至一個機器平台，常常我們需要新增驅動程式模組至核心，當完成此驅動程式模組，也需要將它放至核心原始碼內適當的目錄位置以利發表。新增核心組態參數 **CONFIG_XXX**，編譯核心第一步驟“make menuconfig”過程內得以選定此參數，自動產生 **include/linux/autoconf.h** 內的核心組態參數巨集定義，以及**“.config”**核心組態檔。include/linux/autoconf.h 供 C 原始碼引用；“.config”內包含核心組態定義，Makefile 把“.config”引入，在 Makefile 裡就可以引用 CONFIG_XXX 組態參數控制編譯環境參數。

4.5　操控台（console）、TTY、序列埠

在/dev 目錄下，console、tty、序列埠（ttySn）為三種類型字元裝置檔，console 和 tty 是 Linux 下虛擬的抽象設備，而序列埠則是實體設備，初始化順序依次為 console、tty、序列埠。

console 設備初始化時連結至一個真正的設備（如 ttyS*n* 或 VGA 顯示卡、或

LCD），開發一套系統之初，以設定核心命令列參數“console=ttyS0” 選定序列操控台最簡易。Linux 核心調用 *printk* 函式列印訊息，printk 把訊息打印至/dev/console。

tele-type 型態的終端設備包括虛擬操控台/dev/tty*n* 及序列操控台/dev/ttyS*n*，以及 telnet 登入接口至/dev/pty，為了提供這些終端型態一致的操作方式，Linux 架構 tty 層提供與上層 VFS 統一接口。

序列埠驅動程式架構在 tty_driver 結構（表-APP. 2，索引 37），我們將在 5.4.4 節說明實作序列埠驅動程式原理。

4.6 程式區段規劃與連結草本（linking script）

程式區段至少區分成“.text”、“.data”、“.bss”，.text”為主要可執行程式碼，“.data”為已初始化變數資料區，“.bss”為尚未初始化變數資料，一般以 C 撰寫應用程式不會特別去安排程式區段，gcc 整合編譯環境內部連結程式區段乃參考連結草本（linking script），若編譯時不指定連結草本，連結器會採用內部預設的草本。規劃程式區段目地在於能將同性質程式碼集中放在一起，方便管理與取用，例如驅動程式的註冊初始化程式函式進入點集中， 如此可很彈性的增加新的驅動程式，以迴圈方式一一執行與載入驅動程式。

圖 4-7 說明了連結草本的運作範例，連結草本 dummy.lds 標記名_sec1_start、_sec2_start、_sec3_start 表示的意義是位址，程式區段開始於 0x10000，編譯輸出的 elf 執行檔排列次序按照“.text”程式區段定義，因此可以很清楚掌握各區段的起始，對系統程式碼分段管理很有效率，假設某區段已不再需要佔用記憶體，即可被 Linux 回收，Linux 中一次性執行程式碼僅在系統啟動時被執行，因此當啟動程序完成後，這些程式碼所佔用的區域即被回收再利用。

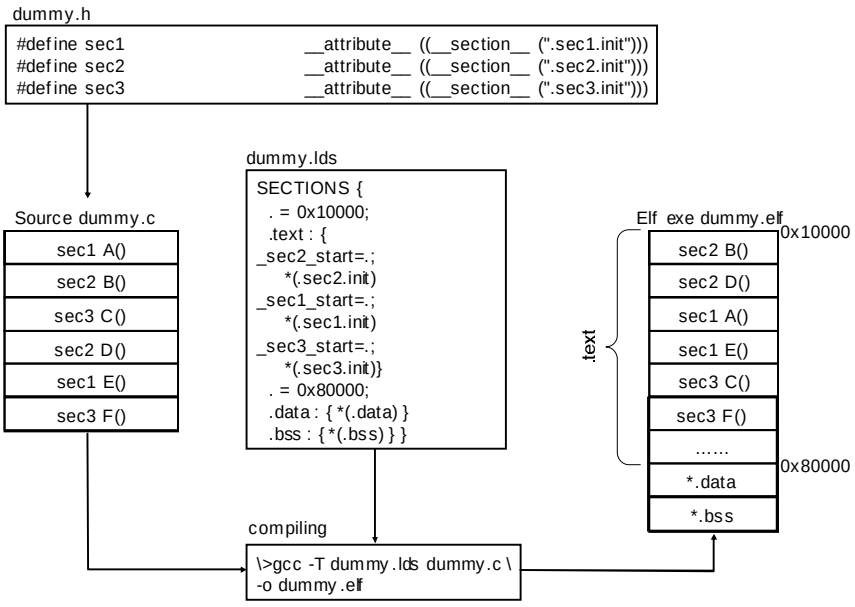

圖 4-7 連結草本應用實例

ARM linux 連結草本（表-APP. 2，索引 27）安排核心程式區段，同質性的程式碼或資料擺在同一區塊。**include/linux/init.h** 定義巨集*__init* 及 *module_init*，以 *module_init* 將驅動式初始化函式進入點放至區段__initcall_start 及__initcall_end；*__init* 巨集宣告函式位置擺放至 **".init.text"** 區段，以*__init* 宣告的函式為一次執行函式。原始碼 **inti/main.c** 裡的 *do_mount* 函式一一執行__initcall_start 及__initcall_end 區段存放的進入點所指向的函式。在執行第一個程序**/sbin/init** 前 *free_initmem* 函式將 Linux 一次執行模組（以__init 宣告的程式區段）佔用的空間全部釋放。

4.7 核心映像與根目錄檔案系統映像配置規劃

本系統選用了 8Mb NOR flash 當永久儲存，根據圖 3-3 所示 Sharp LH79525 記憶體映射，系統重置後 nCS1 映射至位址 0，以 bootloader（bootstrap）程式負責將映像檔放至 SDRAM 內，接著將記憶體 nDCS0 映射至位址 0，跳至核心起始點執行。

圖 4-8 為本模擬板空間配置範例，rootfs（jffs2）、kernel、rootfs(initrd)劃入 MTD 分割區（表 4-3）所示，bootloader 不需劃入 MTD 分割區中。

核心參數“**root=/dev/xxx**”用來設定根目錄檔案系統的裝置，當使用分割區 rootfs 當根檔案系統，分割區 my_root 及 kernel 可允許被更新；當使用分割區 my_root 區塊當根檔案系統，分割區 rootfs 及 kernel 可以被更新。我們把最基本的系統公用程式包含至 rootfs，此分割區不需要更新變動，它的角色功能當救急或更新分割區 my_root 時使用；發展其他應用程式或擴展其他功能都的資料則置放在 my_root 內，存放最終根目錄檔案系統或更新分割區 my_root 時使用。

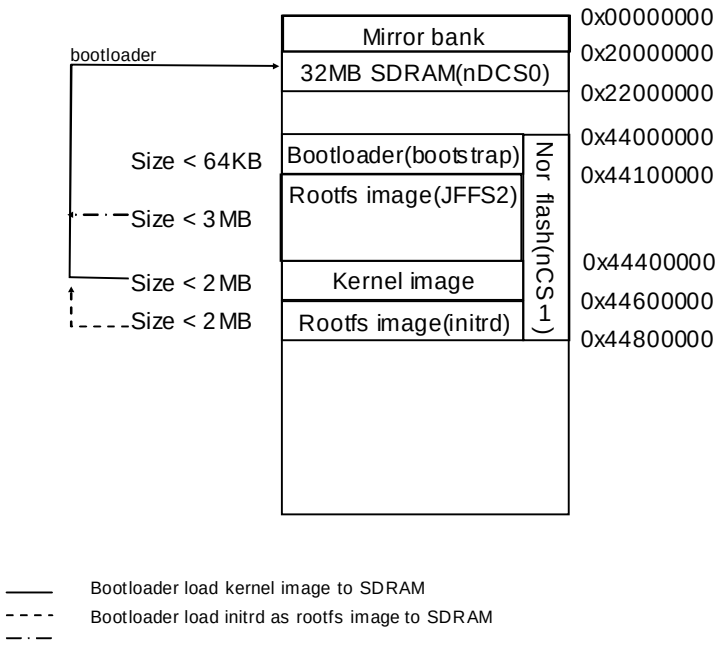

圖 4-8 bootloader、核心及根目錄映像檔空間配置

表 4-3 MTD 分割區配置

MTD 分割區	起始規劃	裝置檔 (/dev/)	裝置主編號	裝置次編號
my_root	0x44100000~ 0x44400000	mtd0	90	0
		mtdblock0	31	0
kernel	0x44400000~ 0x44600000	mtd1	90	2
		mtdblock1	31	1
rootfs	0x44600000~	mtd2	90	4

	0x44800000	mtdblock2	31	2

4.8 根目錄檔案系統掛載

Linux 核心啟動執行完所有初始化模組後，接著需掛載根目錄檔案系統（/），核心參數“**root=/dev/xxx**”設定將裝置檔“**/dev/xxx**”掛載成根目錄檔案系統。我們可以將根目錄檔案系統掛載至 RAM disk 上（可搭配 ext2、romfs 檔案系統），或者透過 MTD 子系統層直接使用 flash 裝置的 MTD 分割區。

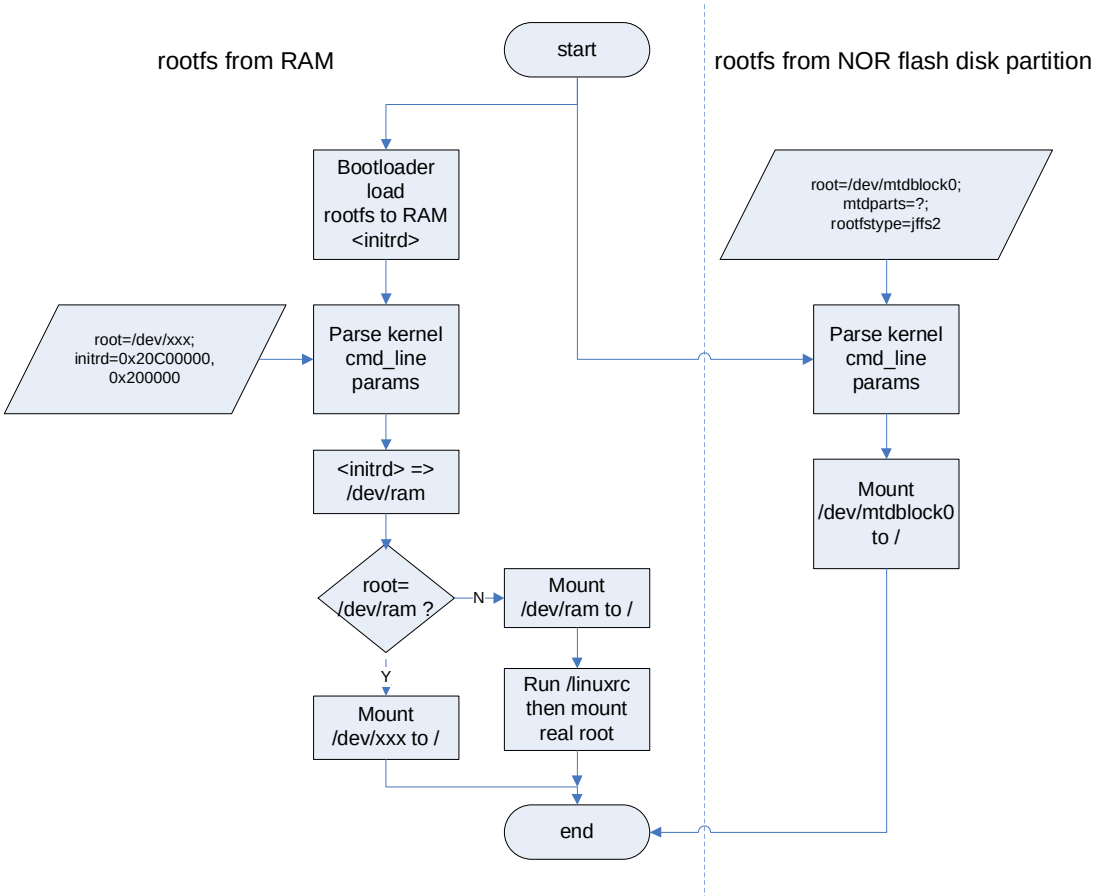

圖 4-9 掛載根目錄檔案系統步驟

4.8.1 掛載 RAM disk 成根目錄檔案系統

RAM disk 意指將 RAM 模擬成磁碟，因此可以掛載於虛擬檔案系統（VFS），Linux 基本支援了 ext2、romfs 檔案系統。。

根目錄檔案系統（rootfs）掛載至 RAM，由 bootloader 將 rootfs 的映像先行放至 RAM 內，此映像稱之為初始化根檔案磁碟（initial root disk，簡稱 initrd）

，參照圖 4-8 的配置傳遞核心數 root 及 initrd 如下：

```
root=/dev/xxx； 實際掛載之 root（/）的 block 裝置
initrd=0x44600000, 0x200000； 告知核心 initrd 所在 RAM 起始位址及大小
```

圖 4-9 左半部份說明將 RAMDISK 掛載成根目錄檔案系統，核心解開 RAM 中的 initrd 並且寫入/dev/ram，將/dev/ram 掛載成 root（/），如果核心數 root=/dev/ram，則完成掛載根檔案系統；否則執行/linurc，完成系統環境參數初始化，卸載/dev/ram，再重新掛載/dev/xxx 成 root。

4.8.2 掛載 MTD 裝置成根目錄檔案系統

MTD 子系統層支援存取 NOR flash 的驅動程式（[28]），參照圖 4-8 的配置傳遞核心參數 root、mtdpart、rootfstype 以指定 flash 磁碟分割資訊如下：

```
root=/dev/mtdblock0；實際掛載 root（/）裝置/dev/mtdblock0
mtdparts=phys_mapped_flash:3m@0x100000(my_root),2m@0x400000(kernel),2m
@0x600000(rootfs)rw；從 0x44100000 起 3mb
rootfstype=jffs2；選用 jffs2 檔案系統
```

my_root 分割區對應至裝置檔/dev/mtdblock0，掛載成最終的檔案系統，因此此檔案系統映像選用專為嵌入式系統設計的 jffs2 檔案系統。

4.9 Linux 下存取 NOR flash

為使 NOR flash 像一搬的磁碟運作在 Linux 的虛擬檔案系統層（VFS），我們選用嵌入 MTD 子系統，VFS 層與 MTD 子系統之間的運作關係如所示，我們說明如下：

- VFS 虛擬檔案系統層（[21]）：檔案以檔案系統結構被儲放在實體儲

存裝置，當我們存取非裝置檔會透過 VFS 層定義的檔案共同屬性及操作方法，也是抽象的檔案結構層，分兩個層次處理存取，第一先調用檔案系統解譯層，第二再呼叫實體磁碟驅動程式把資料讀進來。存取裝置檔時，則依據區塊或字元裝置檔對應的裝編號取得適當的描述子執行操作方法（file operation）。而使用者不用煩腦裝置檔與非裝置檔，因為 VFS 已經提供使用者檔案存取共通介面，調用系統呼叫 API 事實上就是操作 VFS 層，VFS 幫我們處理下層所有工作了。

- MTD RAW 模式存取：MTD 核心支援字元或區塊 MTD 裝置， MTD 字元裝置檔（**/dev/mtd*N***）直接被開檔使用，其上層向主核心層字元裝置檔描述子註冊； MTD 區塊裝置檔（**/dev/mtdblock*N***）其上層向主核心層區塊裝置檔描述子註冊，磁碟為基礎的儲存裝置其上層由檔案系統結構層（eg.：romfs、ext2、jffs2）管理檔案資料結構及如何維護，必須將 MTD 區塊裝置檔掛載至根目錄檔案系統下的目錄才能存取。
- 模擬區塊裝置（FTL、NTFL）：將 flash 裝置模擬成一般 512 位元組倍數的區塊單位存取方式。實作上已經引進了快閃日誌型檔案系統管理概念，提供斷電可靠度演算技術，耗損平衡（wear leveling）機制，但這技術是有版權上的限制，在美國，FTL 授權給 PCMCIA 硬體上，NTFL 由 M-system 公司持有專利，因此只在授權的 DOC（nand, DiskOnChip）裝置上能使用。
- JFFS/JFFS2：快閃日誌型檔案系統（[42]），磁碟分割區遭到系統斷電或當機意外所造成不正常卸載，系統重新啟動不需 fsck 檢測分割區，因此本系統中我們採用 MTD 區塊裝置搭配 JFFS2（參照附錄 I）。
- 實體晶片驅動程式：NOR flash、M-systems 公司的 DiskOnChip、Onboard NAND flash、PCMCIA flash。本系統採用 MX29LV640BT 是標準 CFI 之 NOR flash。

- MTD 記憶體映射配置：系統設計者規劃記憶體配置資訊，將這些資訊以核心參數或者直接將之寫在記憶體晶片映射配置描述子結構（struct mtd_partition，表-APP. 2，索引 44）。

在下一章 5.4.6 節我們將新增支援 MX29LV640BT 的 MTD 記憶體配置映射模組，開發者將了解如何修改原始碼支援 CFI 相容的 NOR flash。

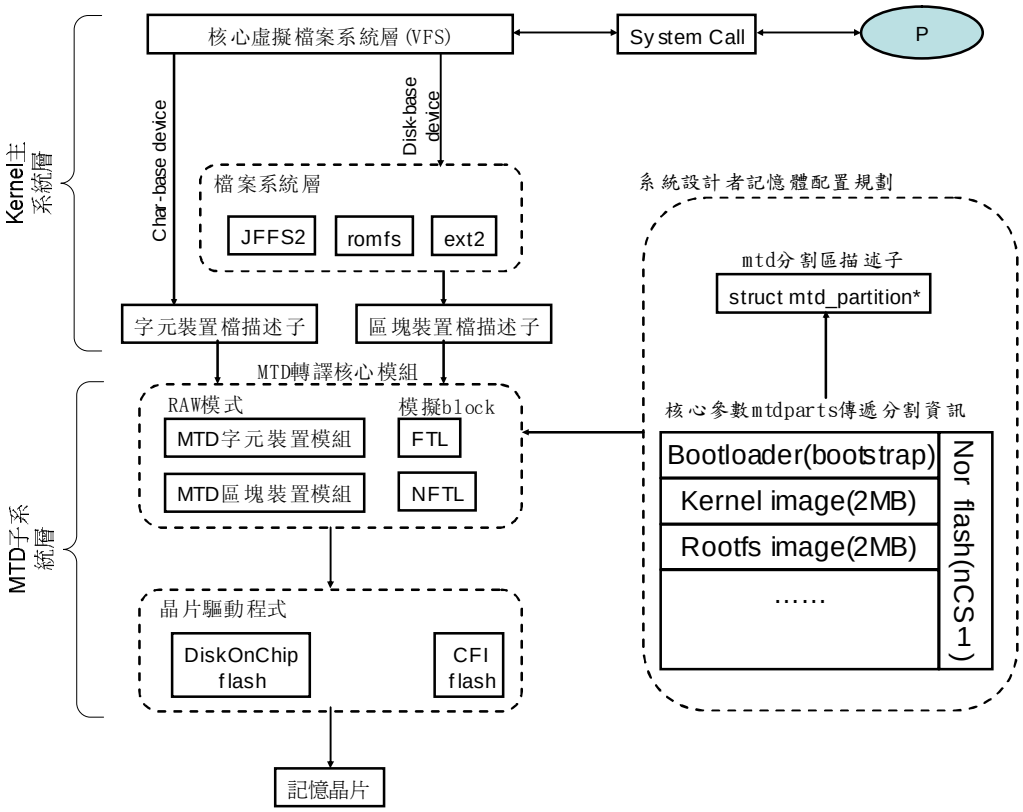

圖 4-10　MTD 子系統層與 VFS 層關係

第5章　移值 ARM Linux 支援 Sharp LH79525

本章將對移植 ARM Linux 支援 Sharp LH79525 做探討，原始碼需要新增驅動程式模組，新增核心組態項目，新增此機器平台相依的程式碼。由於 Linux 原始碼結構相當龐大，很難將所有細節一一剖析說明，並沒有任何資料能把整個系統做很詳細剖述，而且更細節的部份如原始碼的一些資料結構也隨 Linux 改版有些不同，本論文移值 ARM Linux 版本為 2.6 版，我們僅就大原則，即重要修改部份提出，有些細節在 Appendix 補強，最好了解 ARM Linux 啟動運作步驟的方法就是用 Skyeye（[30]）追蹤程式碼流程。在此我們假設開發者對 Linux 已經具有一定程度熟悉，GNU 編譯器及公用程式工具了解，以及知道編譯 Linux 核心的步驟。

5.1　新增支援機器平台的目錄與組態參數

依據 4.1.4 節所描述的原始碼目錄，新增主要與機器平台相依的目錄如下：

- include/arm/arch-lh7952x：Sharp LH79525 標頭檔目錄。
- arch/arm/mach-lh7952x：Sharp LH79525 機器平台相依程式碼。

arch/arm 包含 ARM Linux 架構相依的原始碼，其下子目錄如下：

- kernel：與 ARM 平台關聯的核心碼。
- mm：記憶體管理，多階層式分頁表建立。
- lib：ARM 內部使用到的一些程式庫。
- nwfpe、fastfpe：兩種實現浮點運算的方法。
- boot：存放完成編譯的核心，包含壓縮核心的功能，由 zImage 編譯目標產生。
- tools：自動產生機器 ID 定義的巨集（include/asm/mach-types.h），由 script 將 mach-types 裡定義的機器 ID 資訊以 awk 資料流剖析輸出。

- def-configs：存放每一種機器平台預設的組態設定檔。

組態參數設定步驟“make menuconfig”過程中，新增機器平台組態參數供選擇，定義 **CONFIG_ARCH_LH7952x** 參數，**include/arm/arch-lh7952x** 是機器相依的標頭檔目錄，為了在跨不同平台程式中方便引入，以 **include/arm/arch** 為共通目錄名稱，將它指向欲編譯建立的機器平台，在 arch/arm/Makefile 加入一行：

```
machine-$(CONFIG_ARCH_LH7952x)=lh7952x
```

Documentation\kbuild\kconfig-language.txt 說明了組態參數定義語意規則，每一個存放原始碼的目錄層都有“Makefile”及“Kconfig”，“Makefile”定義了編譯規則，“Kconfig”則是定義了組態參數。組態參數設定步驟產生輸出檔“.config”及“include/linux/autoconf.h”，“.config”檔再由 Linux 原始碼最上層之 Makefile 引入，所有在此層的環境參數可往子目錄層之 Makefile 傳遞，在“.config”輸出中組態參數的值為 bool 型態則以‘y’或‘n’表示組態開啟或關閉。因此當選擇 CONFIG_ARCH_LH7952x 組態，“machine-y=lh7952x”，Makefile 裡即使用代換過的參數來編寫編譯規則。組態參數於 Makefile 中可選定欲編譯的原始碼，下面舉一個範例，運用此技巧於新增驅動程式，以組態參數來開啟或關閉某功能，並關聯至原始碼。

```
obj-（CONFIG_A）+=A.c；  組態參數 CONFIG_A 相依於原始碼 A.c
obj-（CONFIG_B）+=B1.c  B2.c；組態參數  CONFIG_A  相依於原始碼  B1.c、
B2.c
obj-y:
    commands for target obj-y
```

5.1.1 命令列參數 CONFIG_CMDLINE

此為字串型態組態參數，組態參數設定步驟可以設定核心命令列參數，結束後在原始碼根目錄下產生“.config” 及“include/linux/autoconf.h”。本系統最終的永久儲存設備是 NOR flash，其分割區設定如表 4-3 所示，根目錄檔案系統預設使用 my_root 這塊區域，因此我們將預設的核心命令參數設置於 “.config” 內的 CONFIG_CMDLINE：

```
CONFIG_CMDLINE="root=/dev/mtdblock0 rootfstype=jffs2 noinitrd   \
init=/sbin/init console=ttyAM0 rw \
mtdparts=phys_mapped_flash:3m@0x100000(my_root)rw,2m@0x400000(kernel), \
2m@0x600000(rootfs)"
```

“include/linux/autoconf.h”下的 CONFIG_CMDLINE 為如下的巨集形式：

```
#define CONFIG_CMDLINE “……”; 同“.config”內的值
```

核心命令列參數隨 Linux 版本可能會新增與異動，以下列點說明本系統的核心命令列參數：

- root：指定掛載根目錄檔案系統（rootfs）裝置。
- rootfstype：rootfs 的檔案系統型態，ext2、romfs 核心本身已支援，可以不用明確指示。我們於 NOR flash 上採用了 jffs2 檔案系統。
- init：指定 init 的目錄，init 是核心執行的第一個程序，也就是所有程序的父程序，由/etc/inittab 設定程序啟動的時機場合，其主要程序是執行 getty 為操控台取得虛擬操控台行程。
- mtdparts：指定 NOR flash 上 MTD 分割區資訊，
- console：指定欲設為序列操控台的序列埠裝置。

5.1.2 重要的符號

對於一些因系統相異所造成的硬體參數相異，核心原始碼統一以巨集符號

來取代之，除了增加可讀性，開發移植者僅須修改這些符號的值，提高可移植度。表 5-1 列出符合本系統的巨集符號數值定義，虛擬位址 0xC0000000 之前保留了 32Kb，0xC0000000~0xC0004000 保留給 tag 參數列，tag 參數列起始位址設定於機器平台描述子，0xC0004000~0xC0008000 則是初使化第一階虛擬位址分頁表，每個描述子管理 1MB 分頁大小。

表 5-1　重要核心符定義

巨集符號	意義
PHYS_OFFSET	RAM 的實際起始位址（0x20000000）
PAGE_ OFFSET	虛擬位址起始位址（0xC0000000）
PAGE_SIZE	虛擬位址每個分頁大小，預設 4Kb
TEXTADDR	核心入口點（虛擬位址 0xC0008000）

5.1.3 定義連結草本（linking script）區段

由於 Linux 核心原始碼結構複雜，有必要將程式裡同樣性質或屬性的資料在連結在同一區段，方便管理或取用，我們在**錯誤! 找不到參照來源。**節講述連結草本（linking script）運作原理方法。在 ARM Linux 裡，連結草本（表-APP. 2，索引 25）規劃了幾個重要區段，**“.init”**區段放置一次執行的核心啟動初始化資料，此區段的其餘重要子區段列於表 5-2。在 GNU ARM 組合語言，提供“.section 區段名稱”宣告將此行後的程式放入指定的區段名稱，C 語言結區屬性宣告請參照**錯誤! 找不到參照來源。**節。

表 5-2　ARM Linux 連結草本重要初始化區段

開始區段標記	結束區段標記	區段名稱	區段資料內容
__proc_info_begin	__proc_info_end	*(.proc.info)	處理器描述子
__arch_info_begin	__arch_info_end	*(.arch.info)	機器平台描述子

__tagtable_begin	__tagtable_end	*(.taglist)	tag 參數處理原始碼
__setup_start	__setup_end	*(.init.setup)	核心參數處理原始碼
__initcall_start	__initcall_end	*(.initcall1.init) *(.initcall2.init) *(.initcall3.init) *(.initcall4.init) *(.initcall5.init) *(.initcall6.init) *(.initcall7.init)	這區段是收集所有驅動程式入口點，每個入口佔用 4 位元。因此在核心內能用回圈一一執行它。
__con_initcall_start	__con_initcall_end		console 裝置初始化進入點

5.2 ARM linux 核心啟動步驟

5.2.1 Bootstrapping

- 建立和初始化 RAM：探索 RAM 起始位置、大小，正確設定 SDRAM 及 SRAM 控制參數。
- 初始化一個序列埠：用來當作序列操控台。UART 控制器傳輸參數的 Baud rate、parity、stop bits、flow control、fifo 設定。
- 檢測機器的系統結構：檢查 MCU ID 及 ARM CORE 型號。
- 建立內核的 tagged list，與前一穩定版不同的是，bootloader 至少要提供以下的 tag 參數給內核，各 tag 參數請參照（表 5-3、表-APP. 2，索引 12、13）。
- 調用核心映像：將核心映像搬至記憶體，傳遞核心啟動參數，跳至核心起始點。
- 載入 initrd 根目錄檔案系統映像：若使用 RAMDISK 來載入 initrd，則

將 rootfs 映像搬至指定 RAM，傳遞適當的 initrd 參數告知核心起始位址及大小。

- ARM 的 R1 暫存器設定成機器 ID。
- 跳至核心入口點 stext 程式標記，開始核心啟動步驟。

核心的命令參數設定有兩個管道，第一種方式直接設定核心組態參數（CONFIG_CMDLINE），第二種方式透過 tag 參數傳遞，其 scope 為第二種方式將覆蓋第一種方式的設定，因此我們在編譯時期可先將系統常用的核心命令列參數寫死在 CONFIG_CMDLINE，若需調整則透過 bootloader 傳遞 tag 參數來覆蓋設定，增加使用上的彈性度。 ARM Linux 的 tag 參數結構開始於標記 ATAG_CORE，結束於標記 ATAG_NONE，tag 參數與核心關係列於表 5-4，需注意的是當同時設定 ATAG_INITRD2 與“initrd=”， “initrd=”優先被選用。

表 5-3 重要 tag 參數

tag 巨集名稱	結構名稱	結構成員	核心參數意義
ATAG_CORE	tag_core	flags	根檔案 root（/）僅唯讀或可讀寫
		pagesize	記憶體總分頁數目
		rootdev	根檔案對應之裝置主次要代碼
ATAG_MEM	tag_mem32	size	RAM 大小
		start	RAM 實際起始位址
ATAG_INITRD2	tag_initrd	start	根目錄檔案系統映像所存放的實際起始位址
		size	根目錄檔案系統映像大小
ATAG_RAMDISK	tag_ramdisk	flags	載入 ramdisk 提示與否
		size	Ramdisk 映像大小
		start	Ramdisk 映像存放之起始位址
ATAG_NONE			指示 tag 參數結束

表 5-4 tag 參數與核心之關係

tag 巨集名稱	核心命令列參數或組態參數

ATAG_INITRD2	“initrd=”
ATAG_RAMDISK	“prompt_ramdisk=” “load_ramdisk=” “ramdisk_start_setup=”
ATAG_CMDLINE	CONFIG_CMDLINE

5.2.2 核心啟動

Bootloader 執行完畢後，接著跳至核心進入口，核心啟動在此我們分兩個階段探討，分別以程式標記 *stext*（arch/arm/kernel/head.S）及 *start_kernel* 函式（表-APP. 2，索引 17）為分段。第一段從 *stext* 至 *start_kernel* 主要處理部份架構與機器相依的工作，*start_kernel* 則是主要執行非架構相依的程式碼，整個步驟簡列如下：

- ARM 進入 supervisor 模式，CPSR 的 I、F 位元設成 1（CPU 拒絕任何中斷）。
- 檢查 MCU ID 及 ARM CORE 型號。
- 初始化第一階段虛擬位址分頁表（page table），分頁表描述子。（[7]）ARM linux 虛擬位址分頁大小為 1Mb，分頁表大小 4x4Kb，最大可管理 4Gb 的虛擬空間。不過我們的核心碼大小約 2Mb 左右，預留 4Mb 的空間已經足夠。
- invalid cache、flush cache，致能 MMU。
- 執行 *start_kernel* 函數。

圖 5-1 為核心啟動階段主要函式執行流程，由於 Linux 整個原始碼結構大且分散，我們僅挑出重要的部份並標註其所在位置，最好的完整了解方式是實際追蹤程式碼。

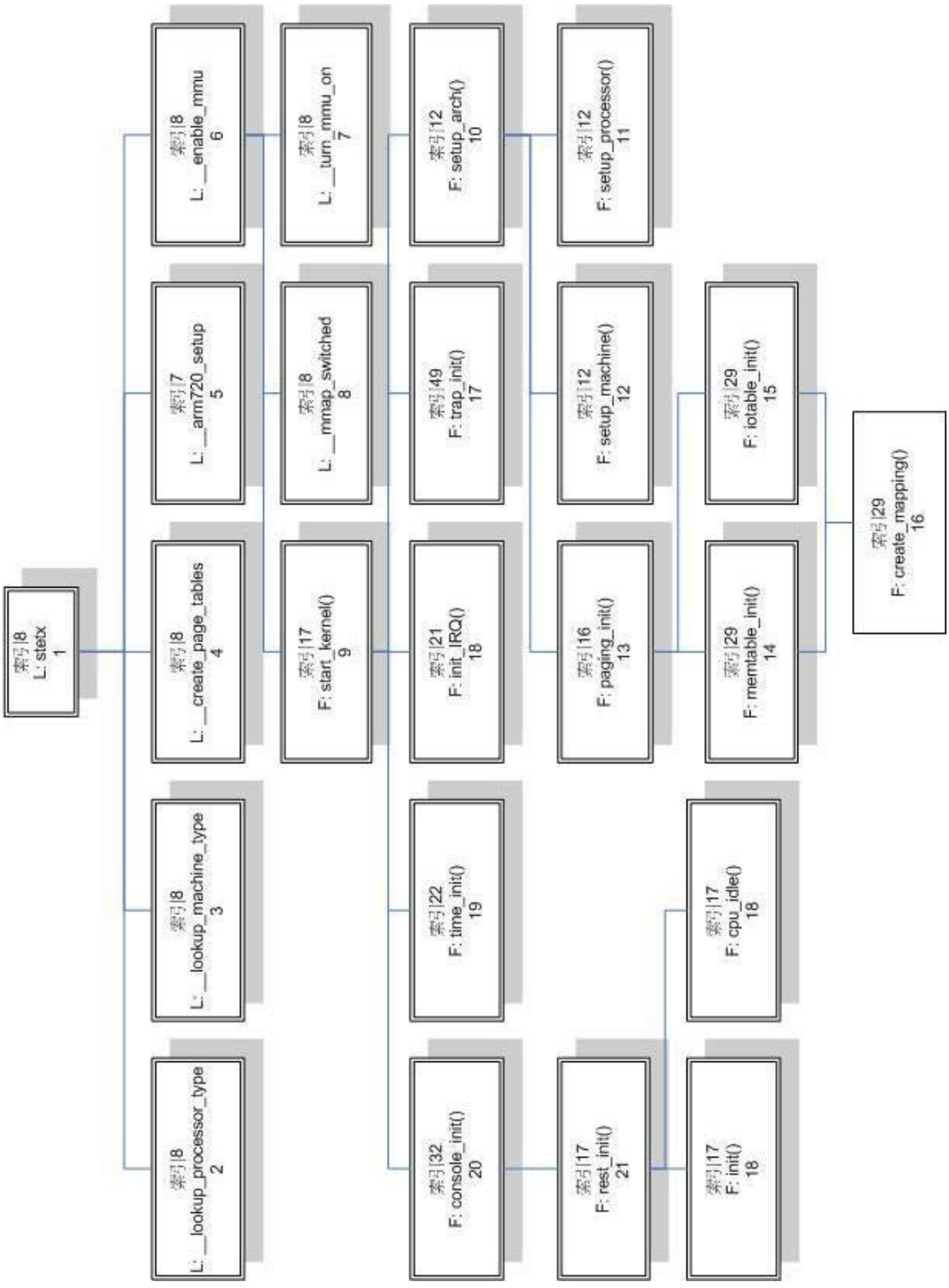

圖 5-1　核心啟動主要函式執行流程

註：所有函數所在位置請依索引參照表-APP. 2

L：lebel

F：function name

Fv：function variable

5.2.3 start_kernel 函式

start_kernel 函式（init/main.c）是主要非架構相依原始碼起始點，雖只有短短數十行，但裡面所呼叫的每一個函式都是重量級。在執行完 *lock_kernel* 及列印 banner 後，*setup_arch* 執行 ARM 平台設定，檢查系統型態，完成 tag 參數處理，核心命令列參數處理，記憶體分頁初始化，接著 *trap_init* 函式初始化例外處理向量表，*timer_init* 函式初始化一組 timer，*init_IRQ* 函式初始化中斷控制器（[13]），*console_init* 函式初始化操控台，以及各種資源管理描述子結構初始化，直到 *kernel_thread* 函式分開兩條執行線，一線執行 *init* 函式，*do_initcalls* 函式執行驅動程式模組，*prepare_namespace* 函式掛載檔案系統，接著執行 init（process id 1），最後執行 Shell，Shell 一直等待使用者下命令；另一線掉入 *cpu_idle* 函式（或稱 task 0），無窮迴圈 busy waiting。最後執行 *lock_kernel* 允許排程器開始運作，此時系統於 task 0、init、Shell…等程序間輪流執行。

5.2.4 setup_arch 函式

檔案 **arch/arm/setup.c** 內定義預設的 tag 參數串列，調用 *parse_tags* 函式處理 tag 參數，ATAG_MEM 相當重要，提供系統 RAM 大小的資訊給記憶體描述子資料結構，因此 ARM Linux 的 tag 參數串列至少需包含 ATAG_MEM，以 ATAG_CORE 參數串列開頭，ATAG_NONE 當結束。我們的系統 SDRAM 大小 32MB，圖 4-8 及表 4-3 規劃了 initrd 根目錄檔案系統起始位址於 0x44600000，大小 2Mb，因此初始化的 tag 參數資料串列新增 ATAG_MEM 及 ATAG_INITRD2，新增修改如下：

```
#define   MEM_SIZE    (32*1024*1024)        /* RAM size: 32Mb */
#define   INITRD_START   0x44600000         /* default initrd start address */
#define   INITRD_SIZE      (2*1024*1024)    /* size of initrd image */
static struct init_tags {
```

```
        struct tag_header hdr1;         /* tag header for ATAG_CORE */
        struct tag_core     core;
        struct tag_header hdr2;         /* tag header for ATAG_MEM */
        struct tag_mem32    mem;
        struct tag_header hdr3;         /* tag header for ATAG_INITRD2*/
        struct tag_initrd2    initrd;
        struct tag_header hdr4;         /* tag header for ATAG_NONE */
} init_tags __initdata = {
        { tag_size(tag_core), ATAG_CORE },
        { 1, PAGE_SIZE, 0xff },
        { tag_size(tag_mem32), ATAG_MEM },
        { MEM_SIZE, PHYS_OFFSET },
        { tag_size(tag_initrd2), ATAG_INITRD2 },
        { INITRD_START, INITRD_SIZE },
        { 0, ATAG_NONE }
};
```

setup_arch 讀入 5.1.1 節設定 CONFIG_CMDLINE 當預設核心命令列參數，若 tag 參數列包含 ATAG_CMDLINE，那麼它將取代預設核心命令列參數。paging_init 函式執行記憶體分頁初始化，建立以 4Kb 大小的分頁表，將系統的可存取記憶空間的虛擬位址屬性填入分頁表。（[21][27]）

5.2.5 例外向量表初始化：trap_init 函式

trap_init 初始化函式建立例外處理向量表（如表 3-4 所示），kernel/entry-common.S 裡的程式標記**__vectors_start** 和**__vectors_end** 之間存放了例外事件

進入點，程式標記**__stubs_start** 和**__stubs_end** 區段放置例外事件處理動作程式碼。

vector_swi 用來實現系統呼叫，Linux 系統呼叫提供使用者存取被保護的系統資源，執行系統呼叫時 OS 會進入 kernel mode（ARM 處理器進入 supervisor 模式）。所有系統呼叫的進入點（entries）在 OS 裡集中列表管理，當我們呼叫一個系統呼叫函式，實際編譯完成後的目的程式碼是轉成系統呼叫的代碼，並且由 **swi** 指令產生例外處理，我們可以根據需求增加系統呼叫。

vector_irq 主要工作是讀取中斷狀態暫存器（pending register），並且處理相對應 IRQ 的 ISR。

表 3-4 所示例外處理向量表的高位址（0xffff0000~0xffff001c）或低位址對照表，經由 CP15 暫存器 c1 的 V 位元（[7]）選擇，向量表安排在高或低位址，必須定義巨集“vectors_hight()”為 1 或 0，記憶體分頁初始化時會用到此巨集。

trap_init 把向量表和例外事件處理的原始碼區段搬移至目地位址：

- 區段（**__vectors_start, __vectors_end**）：若採用高位址向量表則複製一份到起始位址 0xffff0000；若採用低位址向量表則複製一份到起始位址 0x00000000。
- 區段（**__stubs_start**, **__stubs_end**）：若採用高位址向量表則複製一份到起始位址 0xffff0200；若採用低位址向量表則複製一份到起始位址 0x00000200。

5.2.6 ARM Linux 建立虛擬記憶體分頁表

Linux 的虛擬記憶體管理相當複雜（[21][27]），由難而易依序分層非架構相依、架構相依、機器相依三層，ARM Linux 發表版已完成前兩層。移植

ARM Linux 只需將機器相依的記憶體描述子和 memory-mapped IO 映射描述子陣列（map_desc，表-APP. 2，索引 10）註冊至核心內部的虛擬記憶體分頁表，每個 map entry 資訊包含實體起始位址、映射目地虛擬起始位址、區段空間大小、區段型態區分，藉由調用 *iotable_init* 函式（表-APP. 2，索引 29）完成 memory-mapped IO 虛擬映射註冊。

5.3 註冊 MCU ID 與 ARM Linux 啟動 ID 檢查

5.3.1 機器平台 ID

Linux 相當嚴謹於機器平台 ID 比對檢查，其主要目地方便於原始碼樹管理，移植一套 ARM Linux 平台必須加入一個唯一的 ID，因為我們僅供內部自行使用，因此可以不必向 ARM Linux 官方組織提出註冊。

機器平台描述子 **machine_desc**（表-APP. 2，索引 9）結構存放與機器相依的函式指標及資源定義，為了讓此描述子結構定義更具可讀性，重新定義各個欄位的巨集，當中機器平台 ID 有關的欄位如下：

```
#define MACHINE_START(_type,_name)        \
const struct machine_desc __mach_desc_##_type\
 __attribute__((__section__(".arch.info"))) = {    \     @程式區段名稱
     .nr          = MACH_TYPE_##_type,    \     @機器 id
     .name             = _name, @機器名稱簡述
```

“MACH_TYPE_##_type” 必需定義在 **arch/arm/tools/mach-types** 內，例如我們為此目標板新增名稱 LPD79525，新增一個機器 ID 資訊如下：

# machine_is_xxx	CONFIG_xxxx	MACH_TYPE_xxx	number
lpd79525	MACH_LPD79525	LPD79525	xxx

arch/arm/tools 下的 scripts 於建立核心時自動將 arch/arm/tools/mach-types 解析產生 **include/asm/mach-types.h**，組態參數 **CONFIG_ MACH_LPD79525** 與 **CONFIG_ ARCH_LPD7952x** 同時開啟，**MACH_TYPE_ LPD79525** 巨集定義成機器 ID，在此我們只需隨便挑一個未用過的數字即可。

5.3.2 機器平台描述子

移植 ARM Linux 至新機器平台必需為新機器定義機器平台描述子 machine_desc（表-APP. 2，索引 9），LH79525 平台描述子列於表 5-5，並參照表 5-1 的巨集符號 PHYS_OFFSET，此實體位址對應之虛擬位址為巨集符號 PAGE_OFFSET，TEXTADDR 開始放置核心執行碼，在 PHYS_OFFSET 後 16Kb 保留供 bootloader 放置 tag 參數列，tag 參數列實際起始位址安排至 0x20000100。

表 5-5 LH79525 機器描述子

巨集名稱	傳遞參數	LH79525 平台參數值
BOOT_MEM(_pram,_pio,_vio)	.phys_ram=_pram	0x20000000
	.phys_io=_pio	0xfffc0000
	.io_pg_offst=_vio	io_p2v (0xfffc0000)
BOOT_PARAMS(_params)	.param_offset=_params	0x20000100
MAPIO(_func)	.map_io=_func	lpd79525_map_io
INITIRQ(_func)	.init_irq=_func	lh79525_init_irq
INIT_MACHINE(_func)	.init_machine =_func	lpd79525_init

新增 LH79525 機器平台描述子（arch/arm/mach-lh7952x/arch-lpd7952x.c）定義

如下：

```
MACHINE_START (LPD79525, "Sharp MCU")
    BOOT_MEM (0x20000000, 0xfffc0000, io_p2v (0xfffc0000))
    BOOT_PARAMS (0x20000100)
    MAPIO (lpd79525_map_io)
    INITIRQ (lh79525_init_irq)
    .timer =    &lh79525_timer,
    INIT_MACHINE (lpd79525_init)
MACHINE_END
```

MAPIO、INITIRQ、 INIT_MACHINE 這三個巨集把我們根據 LH79525 機器相依的函式起始位址註冊至機器平台描述子內的函式指標。MAPIO 註冊 IO 映射初始化函式，INITIRQ 註冊中斷控制器初始化函式，INIT_MACHINE 則是註冊比較不重要的硬體額外設定初始化放置於此函式。timer 為 sys_timer 結構指標變數，記錄機器平台定義的計時器描述結構。移植開發者只要專注於這些機器平台相依函式與描述子，上層操作機器平台描述子內的函式指標就能正確執行到我們定義的平台函式。

5.3.3 處理器平台描述子

Linux 2.6 版支援的 ARM 處理器包含了 v3、v4、v4t、v5、v6、v7 等架構版本，每個版本內又可細分成數個子系列產品，不同的處理器初始化 cache 及 MMU 方式不同，因此要針對不同處理器的初始化由不同的原始碼來完成它，各式型號 ARM 處理器初始化原始碼放置在 arch/arm/mm。以本移植而言，LH79525 微控制器處理器為 ARM720T，Linux 發表版已支援 ARM720T，

arch/arm/mm//proc-arm720.S 內包含有關於 ARM720T 處理器初始化，也定義了 ARM720T 處理器平台描述子（表-APP. 2，索引 6），因此我們的 ARM Linux 移植不需再新增處理器平台描述子。。

5.3.4 MCU ID 及 ARM CORE ID 檢查

從圖 5-1 中核心啟動主要函式流程可看見 MCU ID 及 ARM CORE ID 檢查分別調用程式標記**__lookup_machine_type** 及**__lookup_processor_type** 完成。函式__lookup_machine_type 利用表 5-2 區段程式標記取出機器平台描述子結構資訊；__lookup_processor_type，利用表 5-2 區段程式標記取出處理器平台描述子結構資訊。分別與 ARM Linux 定義的機器 ID 與處理器 ID 比對，吻合了才算通過硬體支援檢查，往下才得以繼續執行。

5.4 機器平台初始化及週邊驅動程式

5.3.2 節機器平台描述子的指標函式成員記錄了初始化函式，最上層 start_kernel 函式調用執行，比較重要的項目如表 5-6 所示。

表 5-6 呼叫函式與其對應之機器平台描述子的函式指標

呼叫之函式（上層之函式）	機器平台描述子的函式指標指向的函式
timer_init（start_kernel）	lh79525_timer（表-APP. 2，索引 18）
init_IRQ（start_kernel）	lh79525_init_irq（表-APP. 2，索引 26）
mdesc->map_io（paging_init）	lpd79525_map_io（表-APP. 2，索引 10）
*.initcall1.init 函式進入點（do_initcalls）	lpd79525_init（表-APP. 2，索引 10）

5.4.1 計時器初始化

OS 需要計時器來維持掌握系統的時間，以達成排程及多工處理必需的時間間隔計數，計時器每中斷觸發，執行 OS 內的計時器中斷服務程式，更新 OS 的時間計數。每發生一次計時中斷的時間單位稱為 tick，Linux 定義巨集常數 HZ，意義為每秒的 tick 數目，預設值為 100，HZ 參數愈大表示每秒鐘發生計時器中斷次數愈頻繁，中斷服務造成額外負擔也愈大，但相對的作業系統的排程變得緊湊。如果我們設計一個多工執行系統，以固定最大同時執行程序數目，跑模擬可以調整出更合適的 HZ 數值，一般的系統均以預設 HZ=100。Linux 定義一個全域變數（名稱為 jiffies）記錄系統開機後發生的計時中斷總次數，在 PC 平台定義為 32 位元，若以 HZ=100 計算，機器可以連續 497 天不關機而不發生溢位；但在 ARM Linux 的 jiffies 定義成 64 位元，這足足可以允許一台機器售命內都不關機了。

LH7952x 提供三組計時器，使用其中一組來當作業系統計時器，計時器初始化函式由 *start_kernel* 呼叫 *time_init*，調用 system_timer 變數，其型態為 **struct sys_timer**，於 *setup_arch* 中，system_timer 設定指向機器平台的計時器，指標函式成員 init 存放機器的計時器函式，移植 ARM Linux 時開發者需抽換機器相依的計時器函式，圖 5-2 展開計時器初始化函式，各模組所在檔案位置請依索引參照表-APP. 2。

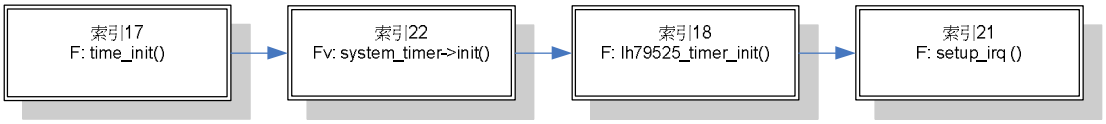

圖 5-2　計時器初始化函式呼叫流程

計時器初始化除了設定計時器的控制暫存器外，還要註冊計時器的中斷服務程式，*setup_irq* 函式用來新增硬體中斷 IRQ 及中斷服務程式 ISR 至 OS 的中斷資源管理描述子裡（**struct irqdesc**，表-APP. 2，索引 24），在下節會詳細討論中斷資源管理描述子陣列結構的成員。

5.4.2 中斷控制器與中斷資源管理描述子初始化

arch/arm/kernel/irq.c 包含了操作與設定 ARM Linux 平台的中斷資源管理描述子的函式，Linux 的中斷資源管理描述子陣列（**struct irqdesc**，表-APP. 2，索引 24）記錄所有 IRQ 的屬性，以 IRQ 為陣列索引，**struct irqdesc** 重要成員說明如下：

- ***handle***：根據 IRQ 中斷觸發型態（edge 或 level）選定 IRQ 處理函式為 arch/arm/kernel/irq.c 中的 do_level_IRQ 或 do_edge_IRQ。
- ***chip***：中斷控制器硬體設定描述子（**struct irqchip**，表-APP. 2，索引 24）記錄機器平台相依的操作方法，如中斷遮罩與反遮罩功能。
- ***action***：型態為 **struct irqaction** 結構（表-APP. 2，索引 25），中斷服務程式描述子，記錄著 ISR 進入點。

由於各機器平台的中斷控制器初始化不一樣，而且中斷遮罩與反遮罩功能的控制暫存器也都是與硬體相依的，因此移植 ARM Linux 時需要抽換中斷初始化函式。

CPU 可控制中斷的致能與否，中斷控制器可遮罩某特定 IRQ，這些行為不論什麼機器平台都存在，為了避免程式碼雜亂不好維護，把它分割成抽象行為及硬體細節兩部份，移植者只需動到硬體平台的細節，而可以不必理會作業系統的中斷行為機制處理。在 UNIX-like OS 系統定義巨集 sti()及 cli()用來致能 CPU 接受中斷觸發與否，移植新架構平台需根據處理器重新定義此兩個巨集的動作細節。

在 5.3.2 節我們將機器平台的中斷初始化函式註冊至機器平台描述子，於 *start_kernel* 下呼叫 *init_IRQ* 初始化中斷，再往下層呼叫機器相依的初始化函式，圖 5-3 追蹤列出中斷初始化呼叫函式流程。

LH79525 的中斷初始化函式 *lh79525_init_irq*（表-APP. 2，索引 26）除了初

始化中斷控制器，更重要是初始化中斷資源管理描述子陣列，這些動作更進一步要調用 **arch/arm/kernel/irq.c** 內定義的幾個函式來完成（表 5-7）。

表 5-7　ARM Linux 中斷資源管理描述子陣列維護函式

函式名稱	維護之 irqdesc 結構成員	功能說明
set_irq_chip	chip	中斷控制器相依的操作方法，設定遮罩暫存器。
set_irq_handler	handle	IRQ 中斷觸發型態
set_irq_flag	valid	允許 OS 視此 IRQ 合法
	prob_ok	IRQ 允許被 probe
	noautoenable	IRQ 不自動啟動
setup_irq	Action	為 IRQ 新增一個 ISR
request_irq	Action	同上，供驅動程式調用

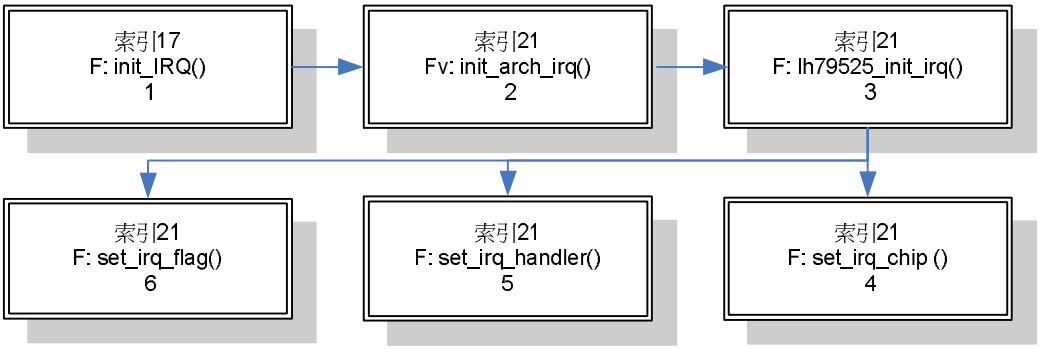

圖 5-3　中斷控製器及中斷資源管理描述子初始化

5.4.3　初始化操控台

序列操控台初始化註冊操控台描述子（**struct console**，表-APP. 2，索引 34），*printk* 得以將核心訊息從緩衝區做輸出至操控台。*start_kernel* 裡呼叫 *console_init* 函式（表-APP. 2，索引 32）執行操控台初始化，調用 *tty_register_ldisc* 將 tty_ldiscs_N_TTY（**struct tty_ldisc**，表-APP. 2，索引 36）

賦予行線規程描述子陣列 ldisc 第 N_TTY 索引，調用 *register_console* 註冊我們的序列操控台。

序列驅動程式於驅動程式載入階段完成載入，本系統第一個序列裝置命名為 ttyAM0，因此建立裝置節點/dev/ttyAM0 關聯至驅動程式編號（MAJ=204，MIN=16，參照[33]），檔案處理系統呼叫即可對序列埠進行存取操作，下節將探討序列埠驅動。

表 5-8　struct console 結構成員

struct console（表-APP. 2，索引 34）	意義
.name	操控台名稱
.write	寫出方法
.device	
.setup	設定操控台所連接的實體硬體
.flags	CON_PRINTBUFFER
.data	指向序列埠驅動描述子

5.4.4　序列埠驅動

圖 5-4 說明序列埠終端運作原理，Linux 系統中序列通訊分三層，TTY 層、行線規程(Line Discipline)層和硬體底層驅動層。TTY 抽象層提供 tele-type 型態裝置共通行為特性的操作方法（file operation），相容於 VFS 層，應用程式存取 tele-type 裝置就像操作一般檔案一樣。

行線規程層也稱做語意層，硬體底層與 TTY 層之間中介角色，處理及傳送使用者空間與核心空間的資料，作不同應用時過濾處理應用所制定的協定或資料格式，行線規程可以自由彈性擴充（[21][22]）。圖 5-5 為序列埠裝置驅動程式往上層註冊的呼叫函式關係，舉序列埠當終端機、SLIP、PPP 三種不同應用範例，序列埠行線規程 N_TTY 模組來處理終端機控制，使用 SLIP 或 PPP 撥接

連線時採用行線規程 N_SLIP 及 N_PPP 來做封包及資料格式轉換；其餘受歡迎的應用如 bluetooth、Infrared Data (IrDa)都是採用行線規程來實現。透過序列埠支援各種傳輸協定方法，客制化行線規程模組是很合適的實現法，使得序列裝置能實現更多不同的應用。近來很熱門的車輛使用 GPS 定位技術實現定位、監控、與導航，透過車輛定位監控系統（[34]），可在嵌入式平台實現車載資訊終端處理，由於 GPRS 上網需要透過 PPP 協定，因此可以把 GPRS 協定訊框處理模組嵌入於行線規程 N_PPP 與底層之間。

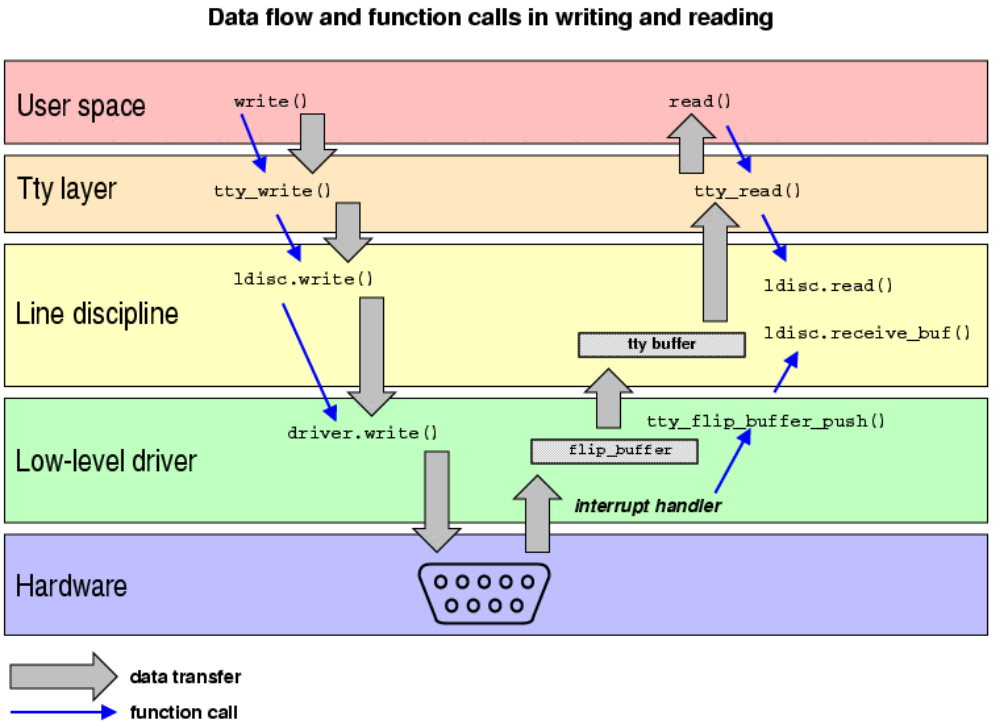

圖 5-4 Linux 序列埠終端運作原理

LH79525 序列埠驅動程式 **drivers/serial/serial_lh7952x.c**，序列埠驅動程式於 Linux 2.6 版序是架構在列埠的行為操作層 **drivers/serial/serial_core.c** 之下。序列埠驅動程式調用 *uart_register_driver* 註冊序列硬體描述子 **struct uart_driver** 結構，操作方法描述子 **struct uart_ops** 結構。每個驅動程式最終要以巨集 *module_init* 將初始化函式入口點宣告於**__initcall_start** 及**__initcall_end**，如此一來驅動程式才能集中且連續依續地被載入。

serial_lh7952x.c（表-APP. 2，索引 30）與 serial_core.c（表-APP. 2，索引 31）主要幾個模組切入點說明如下。

start_kernel 序列操控台初始化呼叫函式如下：

start_kernel() → console_init() → lh7952xuart_console_init()

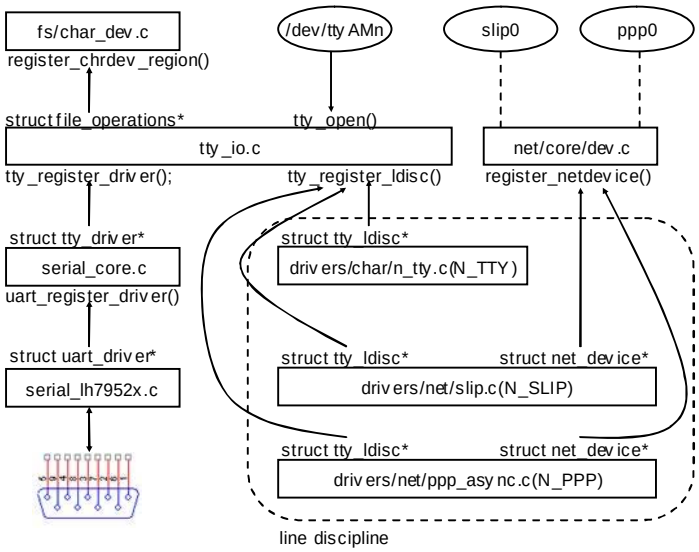

圖 5-5　序列裝置描述子註冊呼叫函式關係

驅動程式入口點的註冊與卸載方式，註冊驅動程式入口點至程式區段（_initcall_start 至_initcall_end）；驅動程式卸載函式進入點收集至區段*.（exitcall.exit）。

```
module_init (lh7952xuart_init);
module_exit (lh7952xuart_exit);
```

序列驅動程式以 tty_driver 向 tty 層註冊後，當使用 *open* 系統呼叫開啟序列埠，其呼叫順序從 tty 層的 *tty_open* 函式往下層逐層關連至底層硬體驅動程式

tty_open() → uart_open() → uart_startup() → lh7952xuart_startup() → request_irq(lh7952xuart_int)

對任何通訊而言，TX 動作屬於主動角色；RX 則是屬於被動（on-demand）。RX 動作採用中斷機制避免忙碌請求（polling），最重要的因素就是太耗系統時間，而且有可能發生接收 overrun，造成字元遺漏。序列硬體提供了 RX 及 RX_TIMEOUT（[13]）中斷條件，如此只有當接收 FIFO 裡有資料或偵測到 RX 腳位發生 start bit 時，就會觸發中斷執行 ISR 把硬體緩衝區 FIFO 裡的資料讀入核心內部。*request_irq* 把 *lh7952xuart_int* 註冊成序列埠的 ISR，外部有資料等待接收時會觸發序列中斷運作步驟如下：

RX 或 RX_timeout 中斷 → lh7952xuart_int() → lh7952xuart_rx_chars() → 將資料推入 tty_flip_buffer 結構內

struct tty_flip_buffer 結構（kernel-space 緩衝區）的資料被 Line discipline 層推入 tty buffer 內，*read* 系統呼叫將資料般至使用者緩衝區。同樣地，當使用者對序列埠使用 *write*，資料被搬至寫出緩衝區，再由底層函式寫出至序列埠。

tty_write() → uart_write() → uart_start() → __uart_start() → lh7952xuart_start_tx() → lh7952xuart_stop_tx()

5.4.5 乙太網 MAC 層驅動

市面上許多廠商的 ARM MCU 幾乎都已配有網路 MAC 控制，網路的驅動程式主要操作是針對 MAC 層，設定 PHY 層的控制暫存器，因此一般 MCU 製造商提供 BSP（board support package）支援各項週邊操作的 API 有助於嵌入週邊裝置驅動程式於移植式的作業系統。

圖 5-6 所示為 ISO 七層網路模型，IEEE 802.3（[35]）規範了 MAC 層的資訊框格式以及 PHY 層的電氣特性。MAC 層又稱資料連結層，資料單元稱為 frame，MAC 內有一組 RX/TX FIFO，流程控制負責處理全半雙工傳輸，MAC 與主記憶體間的匯流排一般能支援 DMA，以 DMA 通道來執行主記憶體與 MAC 的 FIFO 間資料傳輸，符合乙太網路傳輸速率 10/100Mb 效能提升，降低高速傳輸佔用太多 CPU 時間。

PHY 與 MAC 間的標準介面 MII，透過 MII management（MDC、MDIO）序列存取實體層內部的標準及進階延申控制暫存器，進階延申的暫存器可視需求使用，通常完成 MAC 層軟體驅動（含標準 PHY 暫存器設定），不論搭配那款 PHY 晶片均可正常運作。

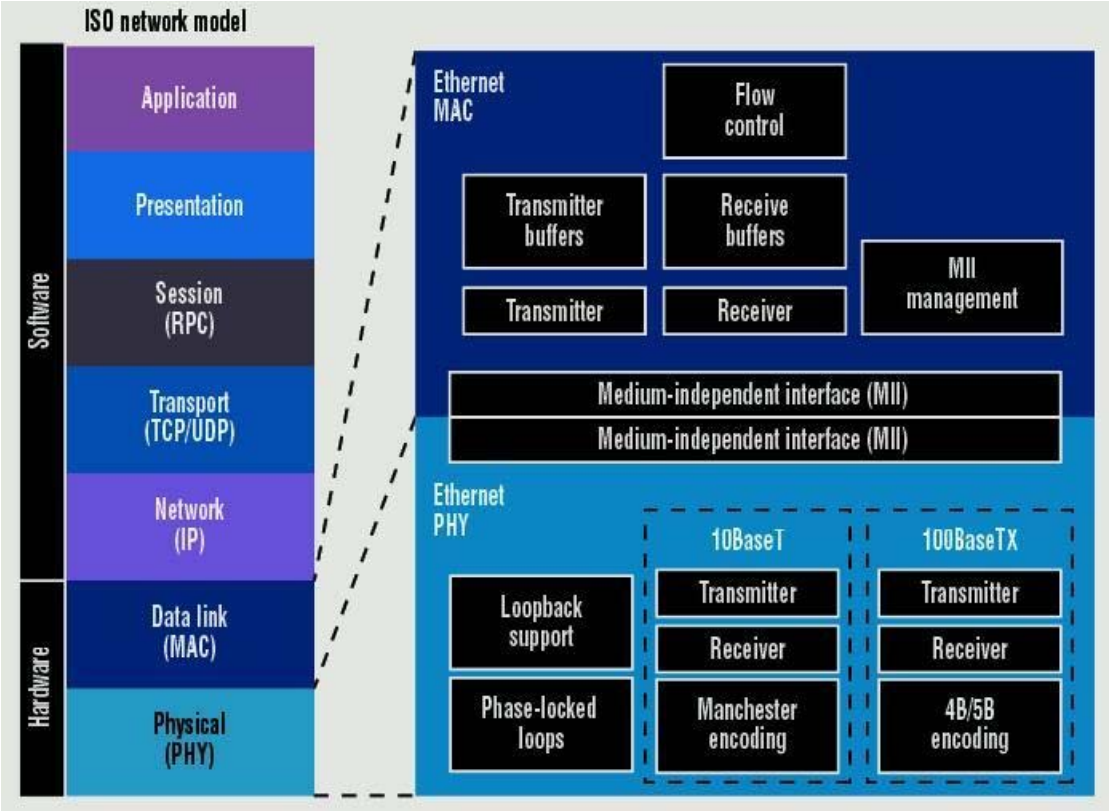

圖 5-6　網路分層架構

LH79525 內建 MAC 控制器，ARM-CORE 與 MAC 橋樑（圖 3-1）以 ARM 公司的 AMBA 標準匯流排，週邊佔用的 IRQ 及記憶體資源已經固定；不同於一般設計於 PC 上的 PCI 2.1 介面卡標準，PCI 2.1 規格制定上包含了硬體交握、探測硬體資源協定，週邊裝置的 IRQ 可由系統動態分配，驅動程式利用這項機制自動探索中斷 IRQ 及佔用記憶資源。

建構嵌入式網路功能的主要課題在驅動 MAC 及 PHY 層，MAC 裝置在 Linux 下並不屬於區塊或字元裝置，核心啟動期間執行 MAC 驅動程式初始化，註冊硬體資源，處理函式。網路驅動程式複雜度高，本系統直接由 LH79525 提

供的 BSP 內取用部份模組，並參照 Linux 源碼發表的驅動程式（源碼目錄下 drivers/net）：loopback.c、plip.c、ibm_emac.c，修改成符合 LH79525 的 MAC 層驅動程式，以下將對此驅動程式簡略說明，請依列表參照原始碼會更詳盡。（表-APP. 2，索引 52）

網路裝置驅動程式的載入之初要對系統提出硬體資源請求（IRQ 及 I/O 位址），網路驅動程式不需要裝置代碼，而是在啟動時為每一個新探測到的介面建立資料結構，並將這個結構安插至網路裝置的全域串列之中。以 **struct resource**（表-APP. 2，索引 41）定義網路裝置硬體資源，**struct platform_device**（表-APP. 2，索引 39）定義裝置資訊，調用 *platform_add_devices* 加入 **platform_device** 描述子陣列所描述的裝置資訊：

```
static struct platform_device eth_lh79524_device = { … };
static struct resource eth_lh79524_resources[] = { … };
static struct platform_device* lh79524_devs[] __initdata = { eth_lh79524_device };
platform_add_devices (lh79524_devs, ARRAY_SIZE (lh79524_devs));
```

MAC 驅動程式初始化函式 *emac_init* 進入點放至程式區段_initcall_start 至 _initcall_end 間，以 _init 宣告擺放至的一次執行區段。**static struct device_driver** 定義驅動程式主結構，*driver_register* 函式把驅動程式資料結構註冊給核心。

```
/* declare main struct of MAC driver */
static struct device_driver emac_driver = {
    .name       = CARDNAME,         /* MAC name*/
    .bus        = &platform_bus_type,
    .probe      = drv_emac_probe,   /* probe device function*/
    .remove     = drv_emac_remove,  /* unload device function*/
};
static int __init emac_init(void)
{
```

```
    return driver_register (&emac_driver);  /* register MAC driver */
}
static void __exit emac_cleanup(void)
{
    driver_unregister (&emac_driver);       /* unregister MAC driver*/
}

/* put function entry of emac_init to {_initcall_start,   _initcall_end} */
module_init(emac_init);
module_exit(emac_cleanup);
```

Linux 中描述一般的裝置驅動程式概念非常抽象，上面我們只提到 MAC 裝置佔用的硬體資源及註冊、卸載模阻的進入點，接下來具體定義 MAC 裝置型態的資料結構，成員包含了 MAC 私有的變數資料、rx/tx 緩衝區、指標變數指向核心上層的串列 strcut device、重量級的結構 **struct net_device** 存放了 MAC 裝置共通特性的變數及操作函式的指標、**struct net_device_stats** 記錄 MAC 資料傳輸量統計值。

```
typedef struct {
    struct device* device;          /* maintain pointer of kernel device list*/
    struct net_device* net_d;       /* function pointer of net device */
    struct net_device_stats stats;  /* hold net device status */
……other MAC private data
} emac_t;
net_d->open = emac_open;            /* function point to open function */
net_d->stop = emac_stop;            /* function point to stop function */
net_d->hard_start_xmit = emac_hard_start_xmit;    /* function point to tx function */
……other field setting of net_d
net_d->irq = IRQ_ETHERNET;      /* Ethernet IRQ */
/* register ISR: emac_interrupt () to IRQ: IRQ_ETHERNET */
request_irq (net_d->irq, &emac_interrupt, 0, net_d->name, net_d);
```

圖 5-7 所示為 LH79525 網路驅動程式初始化、開啟、停止的呼叫函式展開

，初始化調用巨集 *request_mem_region* 保留硬體所佔用的位址區段，*ioremap* 將 MAC 控制記憶體區段空間的實體位址註冊至虛擬分頁位址表裡，*emac_prob* 裡建立 MAC 裝置型態 **struct emac_t** 的實例（instance），net_d 是最重要的成員，以 *register_netdev(net_d)* 向核心註冊 net_d 成為合法的網路裝置，名稱即為我們熟識的 **eth*n***，第一張介面為 eth0，其餘類推，此名稱供網路應用程式識別網路裝置使用。例如當我們下達 **ifconfig** 啟動網路介面卡時，底層最終會調用到 net_d 指向的 open 函式；反之卸載網路介面時則呼叫 net_d 指向的 stop 函式（[22]）。net_d 成員 hard_start_xmit 指向 TX 傳送封包的函式；RX 接收封包的函式（ISR）以 *request_irq* 註冊至核心的中斷資源管理描述子陣列。

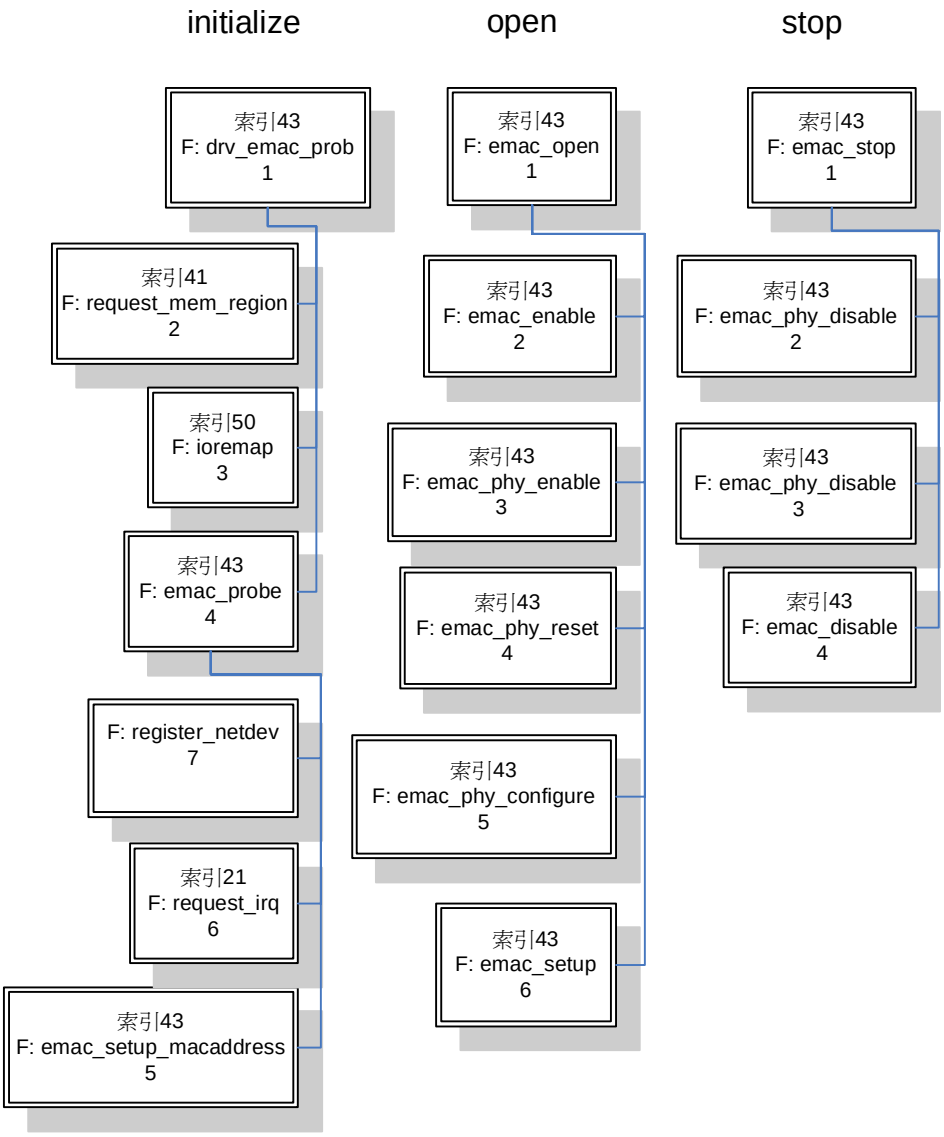

圖 5-7　LH79525 MAC 驅動程式初始化、開啟、結束呼叫函式展開

全世界所有連上 internet 的乙太網存在一組唯一的 MAC 位址，一般 MCU 出廠後，其 MAC 控制器並未內含任何合法的 MAC 位址在裡面，因此若是應用產品欲上市使用網路功能需先向受理組織註冊 MAC 位址，然後再把 MAC 位址寫入每套產品上。在本系統中我們隨意挑一組，於開機後以 ifconfig 設定 MAC 位址。

5.4.6 MTD 驅動 NOR flash：MX29LV640BT

在正式說明如何新增修改 MTD 原始碼前，我們先看一些重要的 MTD 子系統層的重要檔案，了解它們的主要功能對我們理解 MTD 有很大的幫助。

- drivers/mtd/chips/cfi_probe.c

  CFI 偵測 flash 晶片基本資訊（大小、ID、單位可抹除區塊大小）。
- drivers/mtd/chips/gen_probe.c

  CFI-type 常用的基本函式。
- drivers/mtd/chips/ cfi_cmdset_0001.c

  相容 INTEL 的 CFI 命令集處理函式。
- drivers/mtd/chips/ cfi_cmdset_0001.c

  相容 AMD 的 CFI 命令集處理函式。
- drivers/mtd/mtdcore.c

  add_mtd_device 函式註冊 MTD 裝置至 MTD 裝置串列。
- drivers/mtd/mtdpart.c

  MTD 分割區的操作方法，包括新增、讀寫函式。
- drivers/mtd/map/physmap.c

  MTD 記憶體映射配置驅動程式。
- include/linux/mtd/cfi.h

Common Flash Interface structure

- include/linux/mtd/map.h

  Overhauled routines for dealing with different mmap regions of flash。

  struct mtd_info 用來描述一顆晶片的資訊，包含起始位址、容量 大小、每次存取的位元組、存取函式指標。

- include/linux/mtd/ partitions.h

  struct mtd_partition 描述 MTD 分割區資訊 add_mtd_partitions

```
struct mtd_partition {
char *name;             /* identifier string */
u_int32_t size;         /* partition size */
u_int32_t offset;       /* offset within the master MTD space */
u_int32_t mask_flags;       /* master MTD flags to mask out for this partition
*/
struct mtd_info **mtdp;     /* pointer to store the MTD object */
};
```

於本系統實現存取 AMD 相容的 MX29LV640BT，要開啟幾個重要的核心組態參數：

- CONFIG_MTD_PARTITIONS：指示 MTD 是否使用分割區功能。
- CONFIG_MTD_CMDLINE_PARTS：是否允許使用核心命令列所設定的 MTD 分割區參數“mtdparts=”。
- CONFIG_MTD_BLOCK、CONFIG_MTD_CHAR：支援字元及區塊的 RAW 存取模式。
- CONFIG_MTD_CFI_1、CONFIG_MTD_CFI_2：分別引入支援 INTEL 的 CFI 指令集處理函式及支援 AMD 的 CFI 指令集處理函式。

- CONFIG_MTD_PHYSMAP：引入 MTD 記憶體映射配置驅動程式。
- CONFIG_MTD_PHYSMAP_START：NOR flash 的起始位址。
- CONFIG_MTD_PHYSMAP_LEN：NOR flash 的大小。
- CONFIG_MTD_PHYSMAP_BANDWIDTH：每個存取週期字組大小，以位元組數目計算。

移植支援新的 NOR flash， 以 MTD map 驅動程式定義晶片描述子（struct mtd_info），呼叫函式 *simple_map_init* 設定描述子的存取函式指標，調用函式 *do_map_probe* 檢視探索晶片，*parse_mtd_partitions* 函式讀取核心命令列所設定的 MTD 分割區參數“mtdparts=”，若無設定，則採用內部預設的 MTD 分割區描述子，因此我們在 drivers/mtd/map/physmap.c 裡加入預設的分割區定義，新增核心組態 CONFIG_MTD_PHYSMAP_MX29LV640BT，參照表 4-3 的本系統分割區規劃，修正程式碼如下：

```
#define MX29LV640BT_NUM_PARTS 3    /* partitions for our system */
/* MTD partition definition */
static struct mtd_partition MX29LV640BT_partitions[] = {
      {
         .name = " my_root",              /* default rootfs image */
         .offset = 0x100000,
         .size = 0x300000,                /* 3MB                  */
      },
      {
         .name = " kernel",               /* default kernel image */
         .offset = 0x400000,
         .size = 0x200000,                /* 2MB                  */
         .mask_flags = MTD_WRITEABLE,    /* force read-only */
      },
      {
         .name = " rootfs ",              /* default rootfs in JFFS2 image */
         .offset = 0x600000,
```

```
        .size = 0x200000,                   /* 2MB                  */
        .mask_flags = MTD_WRITEABLE,    /* force read-only */
    }
};


static int __init init_physmap(void) {
……
#ifdef CONFIG_MTD_PARTITIONS
   mtd_parts_nb = parse_mtd_partitions(mymtd, part_probes, &mtd_parts, 0);
/* add support default partition for MX29LV640BT */
#ifdef CONFIG_MTD_PHYSMAP_MX29LV640BT
   physmap_set_partitions(MX29LV640BT_partitions,\
   MX29LV640BT_NUM_PARTS)
#endif
……
```

5.5 網路服務設置

Linux 下提供各種 tcp/ip 網路應用伺服軟體（ftp、pop3、smtp、telnet），這些網路服務程式通常被實作成伺服程式（daemon），如圖 5-8 所示，早期實作方式是在 Linux 啟動期間掛載網路卡驅動程式及 tcp/ip 層協定，伺服程式（daemon）於第一個程序 init（所有程序之父程序）之後載入，每個網路伺服程式監看/etc/services 指定的應用協定及埠號設定；因為所有伺服程式都是由程序 init 產生的子程序，只要系統一直處於運行，即使沒有網路服務請求，伺服程式仍然是處於隨時執行狀態，如此浪費了系統時間。為了解決這缺點，netkit 套件裡包含一套超級伺服器（internet Super-Server），稱為 inetd，網路服務只需啟動 inetd，由 inetd 統一監看所有連線服務閘道埠號，再執行設定檔/etc/inetd.conf 裡

指定埠別的實際服務程式接管此連線，例如 telnetd 回應 telnet 服務，ftpd 回應 ftp 服務請求。如此系統的網路服務伺服程式以 on-demand 方式啟動，所有伺服程式由程序 inetd 來產生（spawn），當連線結束，該伺服程式也結束。

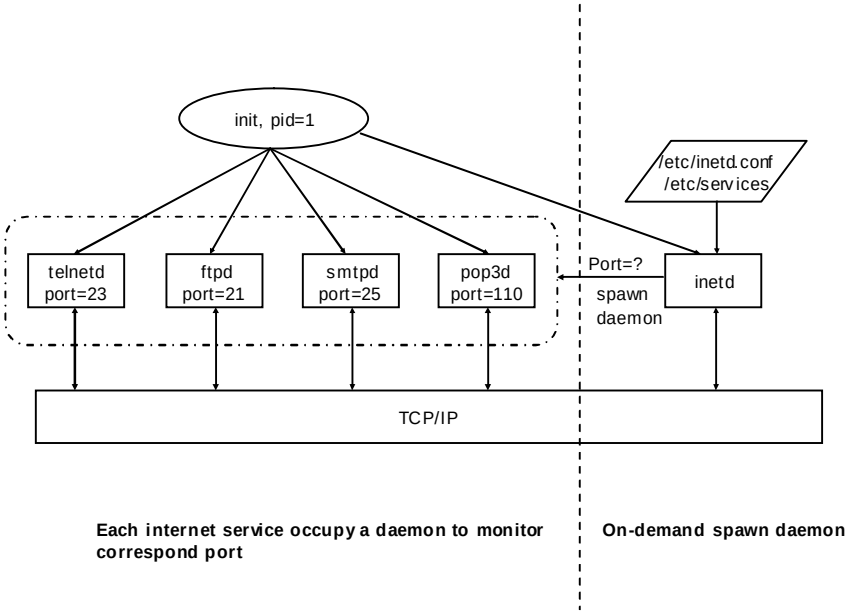

圖 5-8　網路服務 daemon

第6章　驗證移植至 Sharp LH79525 的 ARM Linux

6.1　基本測試

我們的系統啟動後，/dev/mtdblock0 被掛載至根目錄（/），遠端開發主機的/tmp2/nfs 被掛載至目標板上的/nfs。當然實際上我們必須透過 Jtag-ICE 搭配除錯軟體先排除 bug，以下幾點簡單步驟依序簡查系統是否正常運作：

1. 開機期間系統訊息列印，這些訊息是在 console_init 註冊 LH79252 的操控台之後生效，所有核心空間的訊息均以 printk 函式印出。
2. 序列埠驅動程式正常載入後，我們於/etc/inittab 設定第一組序列埠裝置/dev/ttyAM0 當作 getty 的終端，getty 正確啟用後會顯示出登入訊息至/dev/ttyAM0。
3. ifconfig 啟動 eth0 網路介面，並設定它的 MAC 位址、IP 位址，route 指令新增通訊閘道。連上開發主機，本系統採用的實體層 DM9161A（[43]）晶片，我們的目標板設定成自動協議連線速度的模式，在目標板的序列操控台下執行 ping 來檢測乙太網路是否連通。
4. MTD 子系統存取 NOR flash 驗證，以 echo 將一個字串寫入根目錄下，再以 cat 讀出，比對是否正確：

```
\@LH79525>echo “Hello World” >  test_msg
\@LH79525>cat test_msg
```

5. NFS 存取主機驗證，開發的主機啟動 NFS 伺服器，目標板以 NFS 客戶端登入掛載遠端目錄至/nfs，將目標板下的目錄/etc 整個拷貝至 nfs 遠端掛載目錄/nfs，再以 diff 比對執行驗證：

```
\@LH79525>cp –r /etc /nfs
\@LH79525>diff –N –u -r /etc /nfs/etc
```

6.2 連接溫度控制器應用

本系統最後以一個應用實例來做序列及網路整合應用（圖 6-1），我們搭配了樂謙電腦科技生產的溫控器 LF-314CP，此溫控器採用 RS485 傳輸介面及 modbus 協定格式資料（[23]），其連接架構以一台主機連串接多台溫控器，每台溫控器指定一個位址，由主機每隔一段時間抓取溫控器的值或可設定控制器參數，監控軟體則由遠端以網路方式透過 TCP/IP modbus 協定與監控主機傳送與接收監控命令。

我們在目標板上執行一個測試的 server daemon，接收客戶端軟體由 TCP/IP 送來的指令，將此命令再轉譯成 RS-485 modbus 格式發送至 RS-485 匯流排；溫控器回傳結果則以相同方式反向處理，經測試我們的目標板能正確處理運作，至此本系統完成模擬板與 ARM Linux 的移植測試。

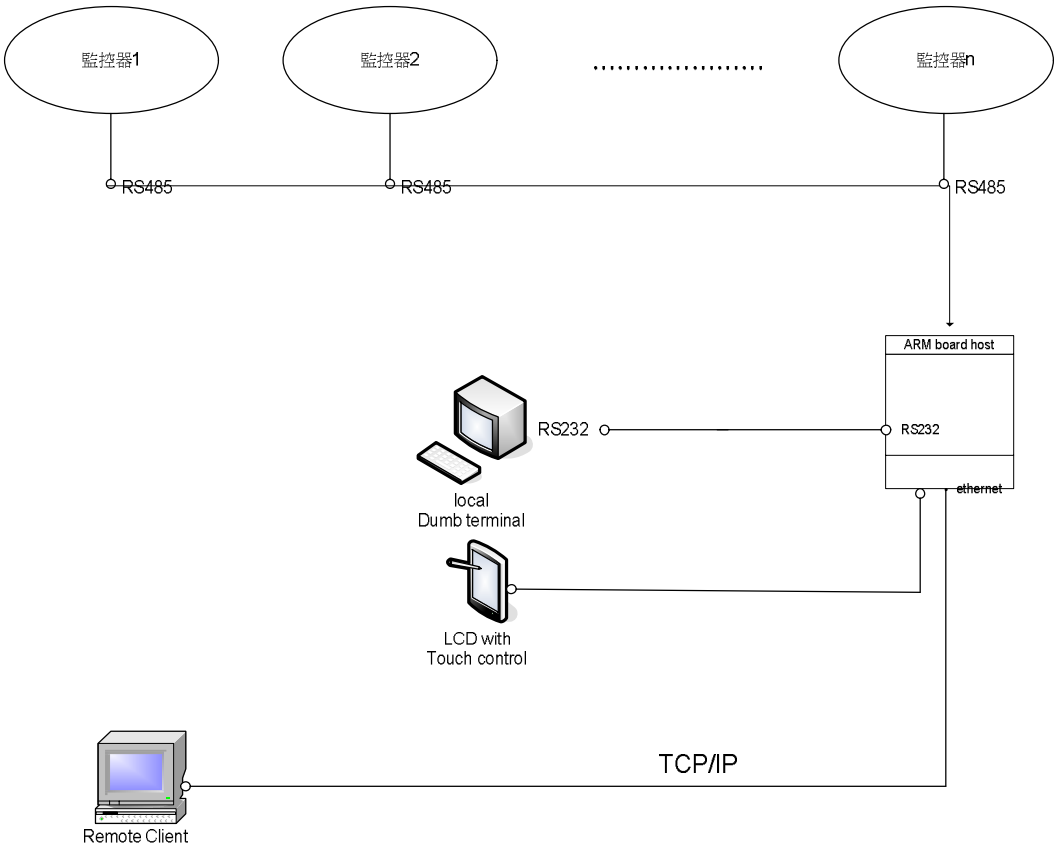

圖 6-1 RS-485 典型應用

第7章　結論及未來展望

本論文實作 ARM Linux 至 Sharp LH79525 微控制器，Linux 是一套相當大的系統，而且每一版本的描述子資料結構都會有差異，整個原始碼目錄檔案及函式相當分散很難結構化它的可讀性，因此對於描述完整系統的原始碼有其困難性，因此自行實地以除錯軟體追蹤是了解各函式所在檔案位置的最好方式。

我們探索 Linux 內軟體的分層，切割出與移植工作有較大相關連的部份，以符合我們開發移植的所需，對於一些與任何架構平台無關的作業系統概念理論不做探討，一般移植會牽涉到與硬體平台相依的部份，而這正是目前各家發展應用產品的公司面臨到的情況。另一方面特殊應用微控器推陳出新，同為 ARM 架構的處理器有數十家廠商製造，產品型號多達上百種，廠商並無法對每顆處理器提供完善的軟體作業系統支援，一般廠家提供的軟體支援只針週邊硬體功能相關撰寫 bsp（board support package），雖然在開發產品上能用 bsp 驅動週邊，但應用上會受限於沒有太多 API 可用，僅能運用於簡易應用。若需牽涉 tcp/ip 網路協定，則 Linux 是最能勝任的作業平台。

我們希望藉由本論文的實作經驗為廣大以 ARM 微控制器設計系統開發者，提供 ARM Linux 移植步驟方法，更清楚的洞悉內部與移植相關的部份。ARM Linux 移植開發者需先了解所選用的 MCU，熟悉 GNU 的編譯及其它好用的工具，為了保有原始碼的原始性，建議另外從新複製一份，將來完成後僅以 patch 包裝發行。

於第二章我們提出開發一套產品的軟硬體評估要項，第三章針對 LH79525 做全面性概述，第四章我們講述了各種平台的嵌入式系統常選用的解決方案與基本系統規劃，第五章針對本系統支援細節做解說，包括核心執行的流程，開發者了解到如何維護原始碼樹的檔案，新增模組至核心內，新增核心組態參數，主要的移植支援項目與主要與硬體相依檔案的目錄，所有移植細節項目很分散，而且要搭配成一套可以應用的系統牽涉到多項技術，因此我們逐一分項探

討。第六章實際以簡單步驟測試我們的目標板，以一個實際 RS-485 應用做為測試，驗證我們的系統的正確性。

目前我們移值的 ARM Linux 是載入 RAM 執行，未來希望能加入 XIP（eXecute in place）技術降低 RAM 的使用量。程式通常會被規類成 text、data、bss 三大區塊，XIP 運作方式是保留 text 於非揮發性記憶體內，data 及 bss 被搬至 RAM，如此不論一支程式被同時多人啟用，都只需要一份 text 空間。另外 JFFS2 檔案系統將儲存的資料做壓縮過，因此無法使用 XIP 執行程式，我們可以將儲存於 JFFS2 的檔案屬性加入可執行程式標記，對於可執行檔即未經壓縮儲存於 flash，僅壓縮非執行檔，如此讓 JFFS2 檔案系統更能結合 XIP 的使用彈性。除此之外，JFFS2 每個 inode 結構包含了 CRC-32 偵錯，但沒有錯誤修正能力，我們希望能加入錯誤改正，讓 JFFS2 檔案系統更趨最佳化。

移植 ARM Linux 主要是要為硬體平台搭建一個便利的開發環境，最終目地還是以應用為導向，因此開發過程在 PC 上所建置的軟體工具特別重要，我們開發 ARM Linux 貴在於主要軟體工具都是自由軟體版權（GPL），無任何版權問題，對現今 3C 消費應用提昇自有開發能力，降低成本支出獲得更高毛利有所幫助。我們希望以本篇論文開發 Linux 過程能提供給嵌入式系統移植方法與過程有所貢獻，藉由本研究開發能有效幫助 ARM Linux 移植開發者，縮短 ARM Linux 移植時程。

參考文獻及附錄

[1] **http://www.linuxdevices.com/**

[2] **http://www.microsoft.com/windows/embedded/default.mspx**

[3] KARIM YAGHMOR, “Building embedded system”, Sebastopol, CA: O’Reilly.

[4] Micro C lib project, **www.uclibc.org;** Busybox project, **http://busybox.net/**

[5] EVB: SDK-LH79524-3126-10, **http://www.logicpd.com/**

[6] ARM corp., **www.arm.com**

[7] ARM720（Rev 3）Technical Reference Manual, ARM Doc. No.: ARM DDI-0192A.

[8] ARM Developer Suite-Assembler Guide, ARM Doc. No.: ARM DUI-0068B.

[9] “Standard Test Access Port and Boundary Scan Architecture”, IEEE std 1149.1-2001.

[10] David Seal, “ARM Architecture Reference Manual”, Boston: Addison Wesley (Known as the “ARM ARM”, ARM Doc No.: DDI-0100)

[11] Sharp corp., **http://www.sharpsma.com**

[12] Sharp corp., “LH79524-525 Data Sheet”, date 2-28-05

[13] Sharp corp., “LH79524-LH79525 Advanced Users Guide”, date 10-25-04

[14] **http://www.ucos-ii.com/**

[15] **www.uclinux.org**

[16] ARM Linux original distribution, **www.arm.linux.org.uk**

[17] Linux author Linus Torvalds web home, **http://www.cs.helsinki.fi/u/torvalds/**

[18] Linux kernel archive, **http://www.kernel.org/**

[19] **http://www.linux.org/**

[20] Andrew S. Tanenbaum and Albert S. Woodhull, “Operating system”, Second Edition, Upper Saddle River, NJ: Prentice-Hall.

[21] Daniel P. Bovet and Marco Cesati, “Understanding the linux kernel”, Sebastopol, CA: O’Reilly.

[22] Alessandro Rubini, “Linux device drivers”, Sebastopol, CA: O’Reilly.

[23] Modbus protocol via RS-485 and TCP/IP, **http://www.modbus.org/**

[24] multi-ice-gdb-server, **http://ecos.sourceware.org/multi-ice.html**

[25] Sharp technology application note, “LH79520 SDRAM Connection Usage”

[26] SDRAM, “winbond W982516 datasheet”

[27] Mel Gorman, “Understanding the linux virtual memory management”, Upper Saddle River, NJ: Prentice-Hall.

[28] MTD（memory technology system）subsystem official site, **http://www.linux-mtd.infradead.org/index.html**

[29] NOR flash：MXIC MX29LV640BT, datasheet, REV. 0.5, JUL. 08, 2004

[30] ARM simulator/emulator, **http://www.skyeye.org**
[31] Wiggler-jtag：parallel-jtag, **http://www.macraigor.com/**
[32] Banyau-U：usb-jtag, agent by **http://www.kaise.com.tw/**
[33] The newest linux device number assignment, **http://www.lanana.org/docs/device-list/**
[34] Q2400 GPRS/GSM modem, **http://www.ravirajtech.com/gsmgprsmodule.html**
[35] M. Beck, “linux kernel internel”, Boston: Addison Wesley.
[36] HOWTO：Linux kernelanalysis HOWTO, mini-HOWTO
[37] newsgroup：**comp.arch.embedded**、**comp.sys.arm**、**comp.os.linux**
[38] IEEE Std 802.3x-1997 and IEEE Std 802.3y-1997, ISBN 1-55937-905-7
[39] Archive of many GNU projects, **http://directory.fsf.org/**
[40] JEDEC standard: JESD 68.01, “Common Flash Interface（CFI）”, **http://www.jedec.org**
[41] Resources and information on Linux Training and Linux Seminars **http://www.ottawalinuxsymposium.org/**
[42] David Woodhouse, “JFFS：The Journalling Flash File System”, Ottawa Linux Symposium 2001
[43] DM9161A, “10/100 Mbps Fast Ethernet Physical Layer Single Chip Transceiver datasheet”, Version：DM9161A-DS-P04, May 17, 2005.
[44] “DM9161 Layout Guide”, Version：DM9161-LG-V01
[45] Vmware Gsx Server, **http://www.vmware.com/**
[46] Law-chain computer technology co. ltd., **www.law-chain.com.tw**
[47] “Flash Memory Technology”, Integrated Circuit Corporation, **smithsonianchips.si.edu/ice/cd/MEMORY97/SEC10.PDF**

附錄 A.　ARM cross compiler、uclibc、busybox

GNU 擁有廣大軟體發展計劃群（[39]），除了自由使用，跨多種軟硬體平台更吸引人，支援眾多的 UNIX-like 作業系統，更可生成多種機器平台的目地碼，要如此多元化發展，有賴於長期發展成熟的基本工具，例如軟體維護包裝工具有 autoconfig 及 automake 工具來輔助，編譯器則有整合編譯工具 gcc，除了標準 GNU C 程試庫 glibc 還有其他豐富的程式庫可選擇。

我們的開發平台在 PC Linux 下，將 C 或 ARM 組合語言編譯成 ARM 目地碼稱之為跨平台編譯，建立跨平台編譯器我們選用以下幾樣套件：

binutils-2.16、gcc-3.4.3（[39]）、uClibc-0.9.28（[4]）

接著說明如何建立 ARM 跨平台 C 編譯器與標準 C 程式庫，首先解開這些套件，我們可將這些安裝步驟寫成批次檔（script）：

```
#! /bin/sh
#編譯與安裝 binutil-2.16，二元檔工具，建立編譯器時需用到
tar zjxvf <archive of binutil-2.16>
cd < binutil-2.16 原始碼目錄>
./configure –prefix=/tools –target=arm-elf-linux; make; make install

#編譯與安裝bootstrap gcc-3.4.3，建立純C編譯器
tar zjxvf <archive of gcc-3.4.3>
cd < gcc-3.4.3原始碼目錄>
./configure –target= arm-elf-linux –prefix=/tools \
  --with-float=soft –disable-__cxa_atexit –enable-target-optspace –with-gnu-ld \
  --disable-nls –enable-threads –enable-multilib –disable-shared –enable-languages=c
make all-gcc; make install-gcc
#編譯與安裝完整gcc-3.4.3，建立C與C++編譯器
cd < gcc-3.4.3原始碼目錄>
./configure –target=arm-elf-linux –prefix=/tools \
  --enable-shared –disable-__cxa_atexit –enable-target-optspace \
  --with-gnu-ld –disable-nls –enable-threads –enable-multilib \
  --with-float=soft –enable-languages=c,c++
make all; make install
```

```
#編譯與安裝uClibc-0.9.28
cd < uClibc-0.9.28原始碼目錄>
make menuconfig;
make clean; make CROSS=arm-elf-linux-; make install

#編譯與安裝busybox
cd <busybox原始碼目錄>
make menuconfig;
export PREFIX=<target root>
make clean; make; make install
```

編譯與安裝 uClibc-0.9.28，很多套件（如 Linux 核心套件、busybox）都採用這種安裝方式，第一步執行 make 的目標 menuconfig 是設定選項內容，完成產生名為.config 的設定檔輸出，請將這檔案備份，日後如需重新編譯，直接將設定檔拷貝至該目錄下，省去重新選定的麻煩。為了節省應用程式佔用空間，生成 uClibc 動態連結程式庫，linker 於 uClibc 與應用程式連結時會先搜尋動態連結程式庫，若找不到再找靜態連結程式庫。

附錄 B. Linux 核心組態參數設置與原始碼關係

原始碼目錄依功能或性質區分，每個目錄的編譯規則由 Makefile 檔描述編譯參數、連結參數、target-dependent 規則，make 工具編譯時以 target-dependent 解析規則加快編譯速度，Makefile 草本語言讓編譯規則描述具彈性，由環境參數定義控制編譯規則。

當加入一項新功能於核心時，核心組態參數命名 **CONFIG_xxx**，Kconfig 檔在目錄下描述組態參數的相依與屬性（表-APP. 2，索引 48），如 arch/arm/Kconfig 定義了 arm 架構的核心組態。**“make menuconfig”** 核心編譯第一步驟設定核心組態，組態參數選項由 *script/kconfi*g 讀入 Kconfig 的內容解析，並由 *script/lxdialog* 以視化選單界面呈現，組態設定完成後於核心原始碼根目錄存成 **.config** 檔及 **include/linux/autoconfig.h**，.config 內的組態參數

CONFIG_xxx 格式如下：

CONFIG_xxx=y；組態為bool型態，選定組態
CONFIG_xxx=<string> or <hex value>；組態參數為字串或十六進制值
CONFIG_yyy is not set；未選定組態

Makefile 中含入**.config**，因此在 Makefile 草本裡得以判斷式來檢查組態參數，設定正確的編譯規則與參數。我們建議保存**.config**，以後從新編譯時不用再全部功能重選一次，只需選定更動或欲新增的組態項目。

include/linux/autoconfig.h 則是將 CONFIG_xxx 以**"#define"**巨集輸出，於程式裡引入 autoconfig.h 便使用 C 語言的條件巨集判斷檢查 CONFIG_xxx 的值，以**"#ifdef…#endif"**將組態相依的程式碼框出。

舉序列埠模組為例，以及序列操控台功能，定義兩個組態參數

CONFIG_SERIAL_LH7952X、CONFIG_SERIAL_LH7952X_CONSOLE

序列埠驅動程式 serial_lh7952x.c（表-APP. 2，索引 30），該目錄下 Makefile 檔裡編譯 target 為變數 **obj-y**，因此加入下列設定，將組態參數與編譯規則關連在一起，組態參數 **CONFIG_SERIAL_LH7952X** 決定是否將模組 serial_lh7952x.o 編譯至核心內

```
obj-$(CONFIG_ SERIAL_LH7952X) += serial_lh7952x.o
ifdef CONFIG_ SERIAL_LH7952X
  選定組態時的參數或編譯規則
else
  未選定組態時的參數或編譯規則
endif
```

附錄 C.　初始化 Sharp LH79525 的 SDRAM 控制器

DRAM 基本記憶單元（cell），僅需一個電容串接一個 MOS，比起傳統的 SRAM，面積使用率更高。SDRAM 存取會破壞記憶單元的位元準位，因此每次存取後都必須對 SDRAM 作預充（Precharge），而且每隔 64ms 必須對 SDRAM 做完一次刷新（Refresh）。

連續存取（burst）就是 SDRAM 每次執行 Read、Write 命令最大可連續存取的記憶體位置數，增加存取效率，連續存取長度（burst length）於 SDRAM 初始化時做設定。

SDRAM 運作的狀態機如圖-APP. 1 所示，當 SDRAM 重置時狀態不確定，初始化第一道指令送出 Nop，等待 100~200us，送出 Precharge_all 命令，再來設定刷新週期，最後設定工作模式（[26]），SDRAM 最後進入 Idle 模式。

LH79525 的 SDRAM 控制器設定請參照（[13][25][26]），根據 SDRAM 晶片規格書，正確選定 SDRAM 腳位介面，設定最佳時序參數及刷新週期、工作模式，即可獲得最好存取效率。

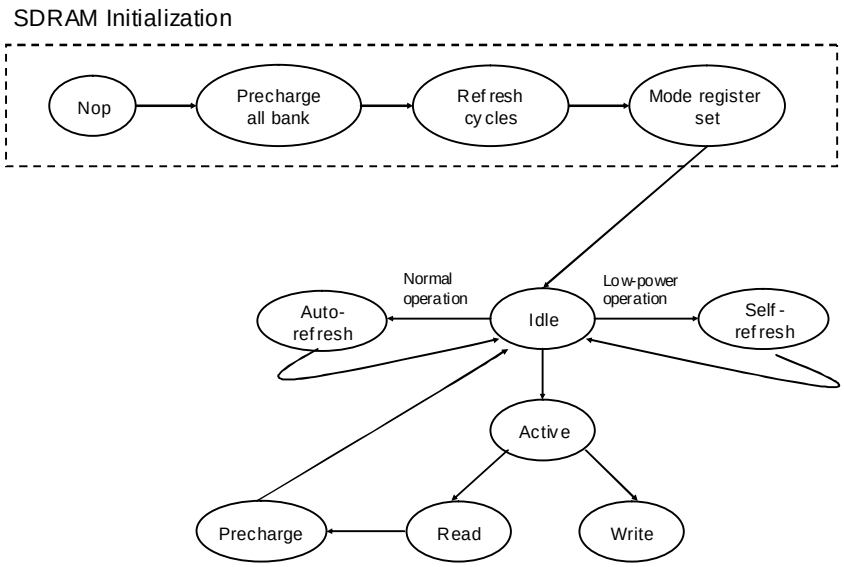

圖-APP. 1　SDRAM 狀態機

以下我們節錄 bootstrap 原始碼中的 SDRAM 初始化程式碼，我們選用的 SDRAM 是 W982516[26]，這顆 SDRAM 大小 256Mb，不同廠家的同等級 SDRAM 一樣可以使用此段初始化程式碼，SDRAM 存取時序不會有太大差異。應用以下程式碼請取得 Sharp LH79525 BSP（board support package）並將它安裝在 Linux 下。

```
#include "lh79524_iocon.h"
#include "lh79524_chip.h"
#include "lh79524_emc.h"
#include "string.h"
#include "sdk79524_board.h"
#include "lh79524_rcpc.h"
#include "lh79524_timer.h"
#include "lh79524_timer_driver.h"

/* fix me, please reference SDRAM sheet and 79525 guide for sdram mode register
setting */
#define HCLK 50803200
#define SDRAM_CHIP_MODE    (0x23<<12)     // CAS3 BURST8
//#define SDRAM_CHIP_MODE  (0x23<<10)     // CAS3 BURST8, \
low-power sdram

void sleep_us(unsigned long us)
{
   volatile TIMER_REGS_T *pregs = TIMER1_BASE;
   unsigned long val, real_val;

   pregs->ctrl = 0;
   pregs->ctrl = TM12_CTRL_CCL | (TIMER_HCLK_DIV_64 << 2);
   pregs->ctrl |= TM12_CTRL_CS;

   real_val = us - us/4;
   do {
      val = pregs->cnt;
   } while (val < real_val);
}
```

```
void sdram_init(void)
{
    volatile EMC_REGS_T *pregs;
    volatile unsigned long hclk, tmp;

    pregs = (EMC_REGS_T *) (EMC_REGS_BASE);
    hclk = (volatile unsigned long) HCLK;

    pregs->sdramc_cfg0 = SDRAMC_16HP_16Mx16_4B_R13_C9;
    pregs->sdramc_rascas0 = \
    EMC_SDRAMC_RASCAS_RAS3 | EMC_SDRAMC_RASCAS_CAS3;
    pregs->sdramc_mrd = 2;
    pregs->sdramc_rrd = 2;
    pregs->sdramc_xsr = 6;
    pregs->sdramc_rfc = 6;
    pregs->sdramc_rc = 6;
    pregs->sdramc_wr = 2;
    pregs->sdramc_dal = 6;
    pregs->sdramc_apr = 6;
    pregs->sdramc_srex = 6;
    pregs->sdramc_ras = 3;
    pregs->sdramc_rp = 1;



    pregs->sdramc_ctrl = ((1<<1)|(1<<0));
    sleep_us (250);
    pregs->sdramc_ctrl = (1<<1)|(1<<0)|(3<<7); /* NOP command */
    sleep_us (250);
    pregs->sdramc_ctrl = (1<<1)|(1<<0)|(2<<7); /* PRECHARGE ALL*/
    sleep_us (250);
    pregs->sdramc_refresh = (EMC_SDRAMC_REFRESH(20, hclk) * 5) / (16*2) + 1;
    sleep_us (250);
    pregs->sdramc_refresh = (EMC_SDRAMC_REFRESH(20, hclk) * 396) / (4*2);
    sleep_us (250);
    pregs->sdramc_ctrl        = (1<<1)|(1<<0)|(1<<7); /* MODE command */
    sleep_us (250);

/* Program the SDRAM internal mode registers on bank nSDCS0-1
```

```
* Burst Length - 4 (A2:A0 = 0b010)
* Burst Type - Sequential (A3 = 0)
* CAS Latency - 2 (A6:A4 = 0x010)
* Operating Mode - Standard (A8:A7 = 0)
* Write Burst Mode - Programmed Burst Length (A9 = 0)
*/
   tmp = *((int *)(EMC_SDRAMC_DCS0_BASE | SDRAM_CHIP_MODE));
   sleep_us (250);
/*    tmp = *((int *)(EMC_SDRAMC_DCS1_BASE | SDRAM_CHIP_MODE));
   sleep_us (250);
   tmp = *((int *)(EMC_SDRAMC_DCS0_BASE | SDRAM_CHIP_MODE));
   sleep_us (250);*/
//   tmp = *((int *)(EMC_SDRAMC_DCS0_BASE));
//   while (pregs->status & EMC_STATUS_BUSY);


   pregs->sdramc_ctrl       = (1<<1)|(1<<0)|(0<<7); /* NORMAL */
   while (pregs->status & EMC_STATUS_BUSY);
   pregs->sdramc_cfg0 |= EMC_SDRAMC_CFG_BUF_EN;
   while (pregs->status & EMC_STATUS_BUSY);
}
```

附錄 D.　設定 Nor flash 存取時序

MCU 重置後，記憶體控制器（EMC）內預設的非同步記憶體存取時序參數為最大值，若不做調整，仍然正確存取 NOR flash，但效能極低。藉由設定 EMC 參數改變非同步記憶體讀寫控制信號 nCS、nWR、nRD 的致能週期時間長短，以獲得最佳存取效能，本系統之 NOR flash 採用 MXIC MX29LV640BT（[29]），存取週期約為 90ns，我們的模擬板 MCU 為 50MHz，因此建議調適之 EMC 非同步記憶體時序參數如下（參照[13]中的 7.1.2 及 7.2 節）。

SRAM timing param	Time duration	HCLK cycles/ns
SWAITWENx	from nCS to nWR	1/20
SWAITOENx	from nCS to nRD	1/20
SWAITRDx	nRD assert time duration	6/120
SWAITWRx	nWR assert time duration	6/120

目標板於 Jtag 模式時，燒錄 NOR flash 前需設定此參數，否則燒錄時間會拉很長；目標板的 bootloader 程式也要嵌入此項設定，NOR flash 存取才能有效率。

附錄 E. ARM 虛擬記憶體存取流程

啟動虛擬記憶體前，分頁表及其基底位址暫存器 TTB（translation table base）要先完成設定。圖-APP. 2 為 ARM720T 的 MMU 虛擬記憶體存取流程，處理器發出存取位址，如果處理器處於 MMU 模式（[7]），則從 TLB（translation lookaside buffer）找尋分頁位址的描述子，找不到則執行 TTW（translation table walk），將新查找到的描述子更新至 TLB 裡。在 MMU 模式下若發生違規存取（如位址 misalignment、R/W 權限），則處理器產生例外處裡。若允許存取 cache，則檢查 cache 內的資料是否有效，無效則執行 cache 存取，最後更新 cache 內容，完成整個 MMU 存取流程。

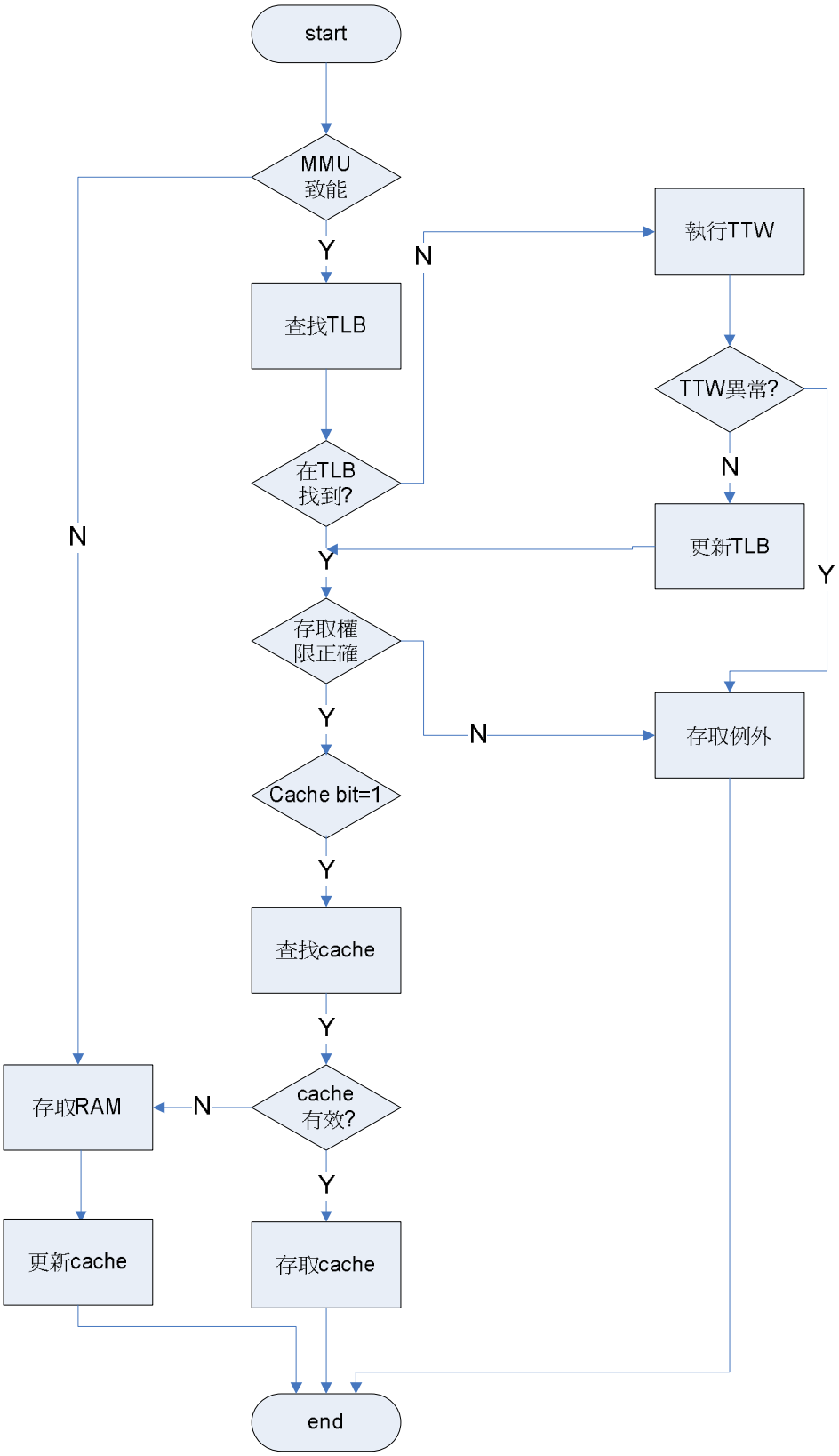

圖-APP. 2　ARM720T 內部 MMU 存取流程

附錄 F.　Skyeye 模擬器安裝與使用

Skyeye 是一套免費的 ARM 指令集模擬器發展計劃，其核心引擎依然是架構在 GDB 之下，模擬了 ARM 指令集，不同的是它加入 MCU 的週邊硬體行為模擬，啟動時載入基本資源配置檔（skyeye.conf）。目前 Skyeye 支援的 MCU 種類不下數十顆，陸續還會新增，對於開發或研究一顆新的 ARM MCU，若能用 Skyey 當硬體模擬平台好處是不用事先買模擬板或自行設計一塊目標板，而且每次修改重新編譯 Linux 核心後可馬上放到模擬器上驗證，節省下載至目標板的時間，較快速的得知大致結果，而且 Skyeye 就像 GDB 一樣，開發者用它可以追蹤龐大雜亂分散的 Linux 原始碼，幫助我們看出函式模組呼叫前後關連，是開發初期不可多得的好工具。若開發者選用的 MCU 型號 Skyeye 未支援，仍然可以將程式放入模擬器執行，只是牽涉到機器相依的控制暫存器會顯示未定義，開發者可花點時間為新增目標硬體行為的程式碼，再重新編譯 Skyeye，就能使用 Skyeye 當作系統程式開發的硬體模擬平台，注意一點是它僅就有定義的硬體行為做模擬，因此通過模擬的程式再放到目標板上還是有可能因為系統時序參數而造成問題。

Skyeye 並不支援 LH79525 硬體週邊，但它支援 LH79520 硬體週邊，因此我們從 LH79520 新增 LH79525 的模擬行為定義，我們僅新增 UART 硬體，讓使用者真正透過序列操控台與 Linux 溝通。若嵌入網路驅動程式就不適合拿 Skyeye 做模擬，各種不同 MCU 的 MAC 層雖然協定一樣，但內部的控制暫存器配置都不一樣，因此模擬 MAC 層行為付出代價太高。Skyeye 支援 realteck 8019AS 網路晶片（MAC+PHY），模擬應用層的網路軟體尚能勝任。

Skyey 於還可以模擬 LCD，近來更可模擬 MTD 層的 NOR flash，因此 Skyey 不失為一套良好學習開發嵌入式系統的免費工具。

安裝 Skyey 前系統要確定以下套件已裝妥，這些套件彼此相依，要按底下次序安裝，版本可以是更新版本：

```
glib-2.6.0、atk-1.9.0、fontconfig-2.2.98、freetype-2.1.9、pango-1.8.1、tiff-v3.6.1、jpegsrc-v6b、libpng-1.2.8、libXft-2.1.6、gtk+-2.6.0
```

將 Skyeye 解開，至 Skyeye 原始碼目錄，在命令列上執行

```
\>export LDFLAGS="/usr/X11R6/lib"
\>./configure –target=arm-elf –prefix=<install_dir>
\>make; make install
```

安裝成功後，skyeye 提供命令及 GUI 兩種使用借面，舉本移值 ARM linux 核心除錯為例，首先編譯好核心，啟動 Skyeye 載入核心。為了驗證序列操控台正常運作，至少要有兩台實體機器，一台執行 PC Linux，Skyeye 運行在此台電腦，角色如同虛擬的 ARM 目標板，在此目標板上運行移值的 ARM Linux 核心；另一台電腦執行終端機模擬軟體，負責當目標板的序列操控台。選用 vmware（[45]）的模擬環境當開發平台，同時模擬多台 x86 指令集虛擬機器，方便同時跨不同工作平台。Vmware 下運行 PC Linux 如圖-APP. 3，圖中為 Skyey 執行畫面；圖-APP. 4 為 windows 2000 下執行超級終端機畫面，其序列埠接收的資料來自於 Skyeye 下的 ARM Linux。

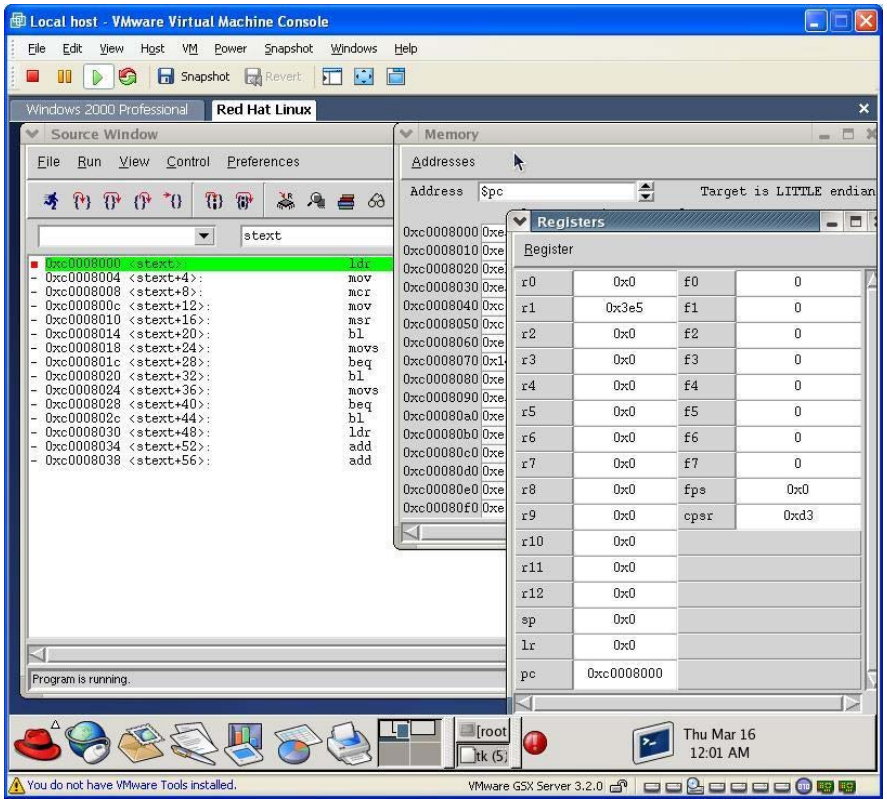

圖-APP. 3　vmware 下 Linux 運行 Skyeye

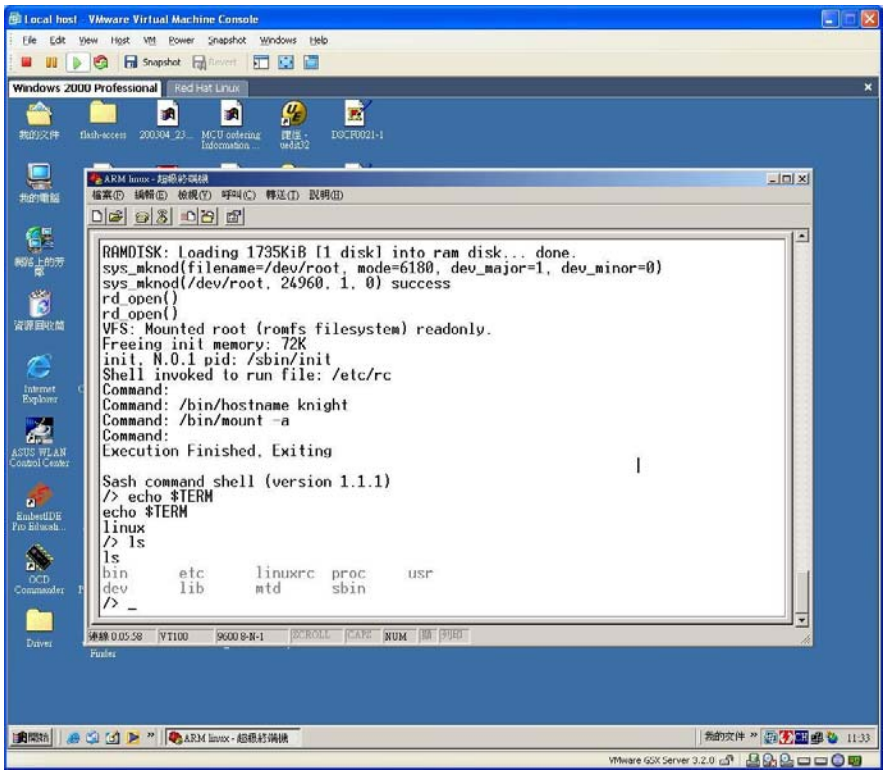

圖-APP. 4　vmware 下運行 microsoft windows

附錄 G.　幾種燒錄 NOR flash 的方法

Flash 抹除軟體運作步驟如下：

辨識flash晶片的ID → 抹除裝置內的區塊 → 寫入裝置

目前市面上 NOR flash 存取演算法來有 **Intel®**及 **AMD®**，這兩類抹除與寫入演算法不一樣，但 flash 的基本特性與操作方法是相同的，因此為了整合不同演算法的 flash 讓軟體撰寫更方便，制定一層名為 CFI（[40]）（common flash interface）的中介層，寫軟體的工程師可以不用去管實體演算法，而透過 CFI 去操作 NOR flash。

嵌入式系統產品的軟體最終會下載至 NOR flash 保存，我們把燒錄 NOR flash 的方法分成兩類，第一類方式是直接用硬體燒錄裝置燒錄，每次系統的 flash 都必需插拔，不符合大量使用效益，而且一旦產品發表，後續韌體維護更新不可能要求使用者都具備燒錄設備。第二類是直接系統在線燒錄（in-system programmed），直接由處理器執行燒錄程序命令，這種方式最經濟。如圖 2-4 所示，我們可以選用 usb 或並列埠界面的 jtag 硬體（[31] [32]），banyuan-U（[32]）是由大陸雁行軟體工作室開發，它將 usb 傳輸來的控制命令轉成 jtag 介面信號（**TCK、TMS、TDI、TDO、nTRST**），使用此 Jtag-ICE 產品需搭配隨附的驅動軟體及燒錄軟體；並列埠的 wiggler-jtag（[31]）很簡單，jtag 介面信號直接定義在並列埠，由一顆 HC74244 完成處理 3.3v-5v 邏輯準位調整，使用此種方式燒錄需取得 macraigor 公司的 flash programmer。

PC 主機執行 flash 燒錄程式，將指令及資料直接從 Jtag 埠送給 MCU；另一種方式將 flash 燒錄程式及要燒錄的資料先從 Jtag 埠載入 RAM 中，再行燒錄。系統開發之初，NOR flash 中未有任何工具程式（OS with MTD 支援或 bootstrap），透過 Jtag 直接燒錄 flash 最方便快速。一旦 OS 及 bootstrap 存在於 flash，將 OS 加入支援 MTD 子系統，就能以 MTD 公用程式對 MTD 裝置操作寫入或抹除。

MTD 記憶體分割區存在 block 及 char 兩種存取模式（**/dev/mtd*N*** 用於字元裝置，**/dev/mtdblock*N*** 用於區塊裝置），區塊裝置模式運作時內部的資料含有檔案系統層解譯，必需先將 mtd 區塊裝置掛載（mount）至根目錄檔案下。

例如表 4-3，我們將根檔案映像檔寫入分割區 my_root，映像檔以 RAW 資料流型式寫入**/dev/mtd0**，再掛載此 mtd 區塊裝置才能讓 ARM Linux 認得分割區檔案系統型態。以下操作將 JFFS2 根目錄檔案系統映像以 cat 寫至/dev/mtd0，再以 mount 命令掛載成磁碟分割區。

```
\>cat root-jffs2.img > /dev/mtd0
\>mount –t jffs2 /dev/mtdblock0 /root
```

附錄 H.　tty 層與終端機設定

Linux 中軟體分層主要將設備裝置相關與非相關區分開，在非設備裝置上又可將共同特性行為取出獨立分層，這樣好處是使得軟體擴充彈性高，軟體層次分明易於抽換模組。tty 抽象層在 UNIX 占有相當成熟的分層結構，它主要是用來管理各種終端設備。序列埠裝置分 tty、line discipline、tty driver 三層次，tty 層由 struct tty_struct（表-APP. 2，索引 xxx），每個執行中的程序其程序表（process table）有一欄 struct tty_struct 用來記錄顯示對應的終端機設定。tty 層結構創建一個實例時，預設的行線規程為 N_TTY 為標準的終端設備所使用，tty 層的 ioctl()方法實現了功能 TIOCSETD 可用來更改行線規程設定。struct tty_struct 成員 strcut termios（表-APP. 2，索引 38）定義終端字元資料流的處理解譯，c_iflag、c_oflag 主要針對了 CR 及 CR/LF 解譯作設定；c_cflag 設定 baud、parity、stop bit、word length、flow control，與序列硬體的控制有關，最終由底層硬體驅動程式完成序列埠硬體參數。還有一些字元在 tty 層被解譯成特殊控制碼，這些會被解譯的控制碼存放在 c_cc 內。關於 termios 設定終端後置字元及特殊控制字元設定些許複雜點，建議以 man 指令查閱，在此我們僅就 RAW 資料流形式設定做說明。

```
struct termios
  {
    tcflag_t c_iflag;          /* input mode flags */
    tcflag_t c_oflag;          /* output mode flags */
    tcflag_t c_cflag;          /* control mode flags */
    tcflag_t c_lflag;          /* local mode flags */
    cc_t c_line;               /* line discipline */
    cc_t c_cc[NCCS];           /* control characters */
    speed_t c_ispeed;          /* input speed */
    speed_t c_ospeed;          /* output speed */
  };
```

當序列裝置以 block 模式打開時（read 會等待），此時模式為標準終端機，所有控制字元會經過解譯，讀入函式以 CR 當結束，函式會一直等待 CR（Enter 鍵）做為結束，因此序列埠傳輸 RAW 資料，要做以下設定。

```
int fd;
struct termios term;
fd = open("/dev/ttyAM0", O_RDWR);        /* open serial port /dev/ttyAM0 */
tcgetattr(fd, &term);                    /* get the original termios setting*/
term.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
term.c_oflag &= ~(OPOST);
term.c_cflag |= (CS8);
term_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
term.c_cc[VMIN] = 1; raw.c_cc[VTIME] = 8; /*  讀取結束條件：至少讀走1byte或時間經過0.8秒  */
tcsetattr(fd, TCSANOW, &pterm);
close(fd);
```

附錄 I.MTD 裝置下使用 JFFS2 檔案系統

NOR Flash 以 block 為單位來抹除或寫入記憶空間，每個區塊有 100,000 次的最大抹除次數限制。為了均衡使用每個區塊，均衡損耗（wear leveling）機制

被引入，其實作原理藉由檔案系統追蹤每個區塊被抹除的次數，優先使用累積抹除次數最小者。

傳統 Linux 發生意外造成不正常重新啟動，若使用 ext2 或 romfs 檔案系統的磁碟，系統可能會提示手動檢查修復其完整性，嵌入式應用通常無法接受此種情況，因此嵌入式系統需要高斷電可靠度。

為了節省空間，檔案系統下可實作壓縮功能，如果可執行檔以壓縮形式儲存，那麼執行此程式前必須先載入 RAM，而無法以 XIP（eXecute in place）（[21]）方式執行。

傳統的 romfs、ext2 用於軟硬式磁碟並無日誌功能，每次系統意外重開機，檔案系統強制以 fsck 修復；而且無法追蹤區塊使用次數狀況，無均衡使用各區塊機制，因此不適合用於 NOR flash 上。在 ext3 引入了日誌（[21]），雖然相容了 ext2，但 ext3 實現功能複雜度大，不適合嵌入式系統運用，在 2000 年 Axis Communications AB 發表了 JFFS（Journaling Flash File-system），redhat 公司改良發表第二版稱 JFFS2（[42]）。

JFFS2 全名是 Journalling Flash File System Version 2，其功能就是在管理 MTD 裝置上所實作的日誌型檔案系統，JFFS2 會在 mount 的時候，掃瞄 MTD 裝置的日誌內容，並在 RAM 中重新建立檔案系統結構本身

MTD 可以看作是 FLASH 晶片的翻譯層（translation layer），基於 MTD 這個硬體抽像層，JFFS2 幾乎可以被 mount 在任何可隨機存取的裝置上。JFFS2 整合了前述快閃記憶體需求特性，列點說明如下：

- 斷電可靠度：意外重啟系統，無需任何檢測檔案系統。
- 耗損平衡(wear leveling)：在 MTD 裝置上實作耗損平衡，確保 flash 上所有的區塊使用率平均, 以平衡每個區塊的損耗程度。
- 資料壓縮(data compression)：除了節省空間外，在使用資料前先將它解壓到 RAM 上.不過， JFFS2 無法使用 XIP(就地執行 eXecute In Place)。

- 實做垃圾收集(garbage collection)：確保應用程式不會增長到填滿整個檔案系統，也就是寫入前會先檢查檔案系統的可用空間。

圖 4-10 中雖然 NFTL、FTL 亦提供快閃日誌型的管理機制，不過限於特定授權產品，因此以原始的 MTD 字元裝置或區塊裝置搭配上層 JFFS2 是很實際的應用。

建立 MTD 公用程式，這些工具可用來對 MTD 裝置操作抹除或寫入，mkfs.jffs2 用來建立 JFFS2 映像檔。MTD 子系統附於 Linux 發表版，可至官方站（[28]）下載最新 MTD 子系統，執行以下步驟建立 host 及 arm 平台的 MTD 公用程式。

```
\>cd $(DRIVER_DIR)/mtd/util
\>export DESTDIR=<assign a directory for install utility>
\>export CROSS=<prefix of cross compiler>
\>make; make install
```

表 4-3 MTD 分割區配置，my_root 分割區採用 JFFS2 格式，首先在主機上用 mkfs.jffs2 建立映像檔，主機啟用 NFS 伺服器，目標板 NFS 客戶端存取主機上的映像檔，再將它寫入 mtd 裝置內。

```
\>mkfs.jffs2 – r <target root> – o my_root.jffs2
\@LH79525>mount – t nfs – o nolock – o mountvers=2 <host ip>:<nfs export dir> /nfs
\@LH79525>flash_erase /dev/mtd0
\@LH79525>cat /nfs/my_root.jffs2 > /dev/mtd0
```

附錄 J.　模擬板線路圖

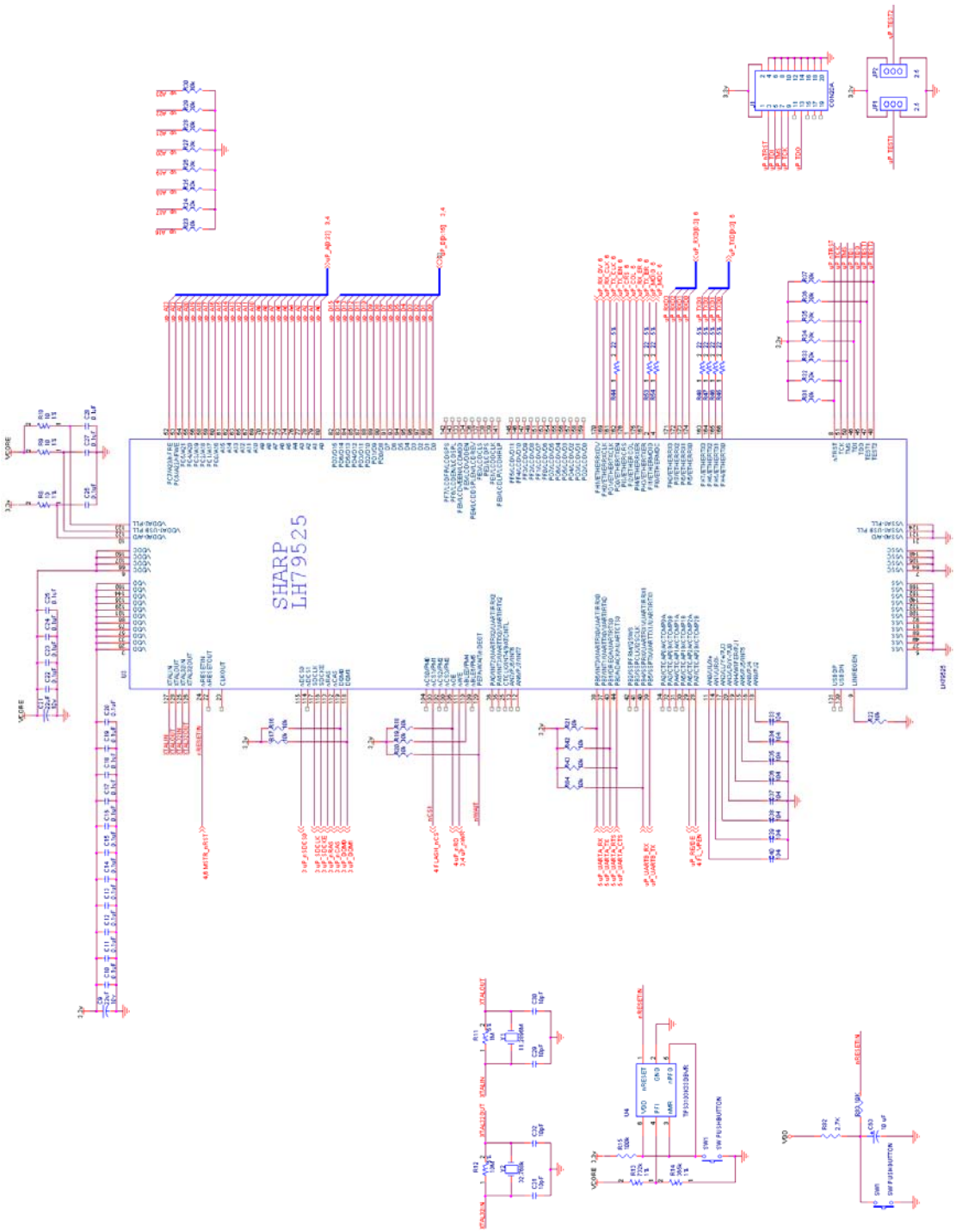

圖-APP. 5　Sharp LH79525 外接腳位（[13]）

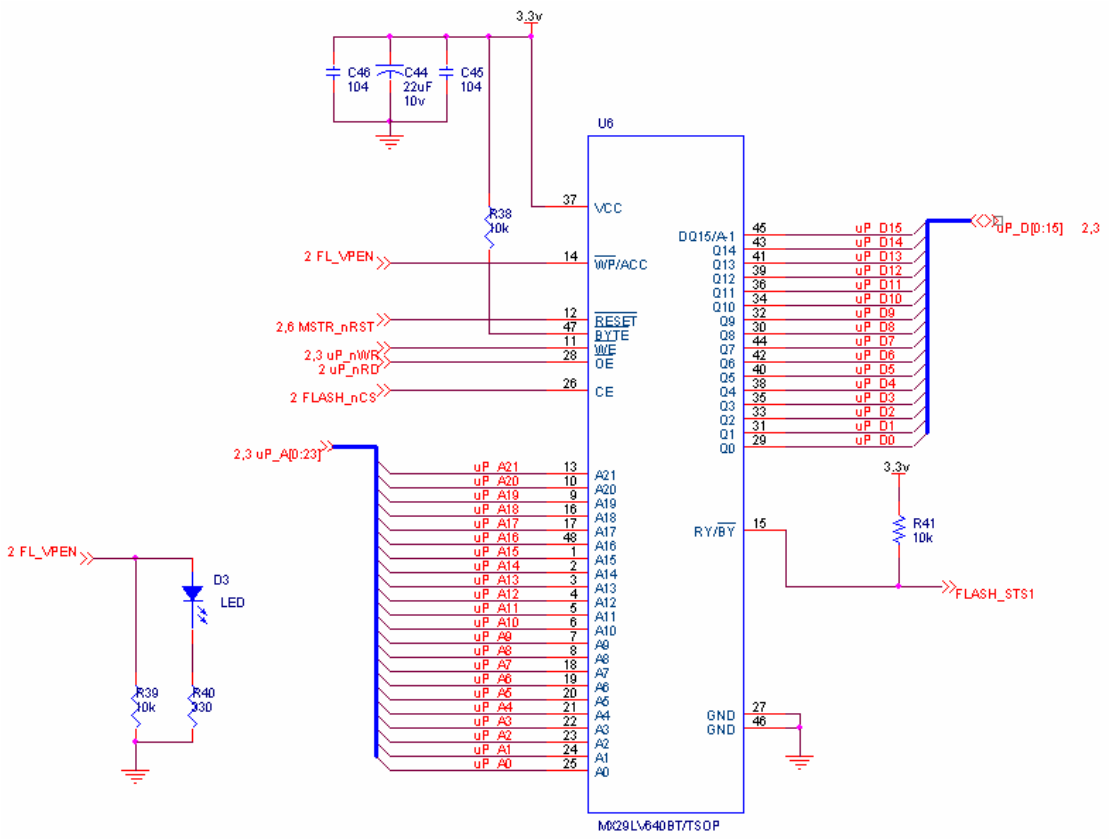

圖-APP. 6　8MB NOR flash：MX29LV640BT/TSOP（[29]）接腳圖

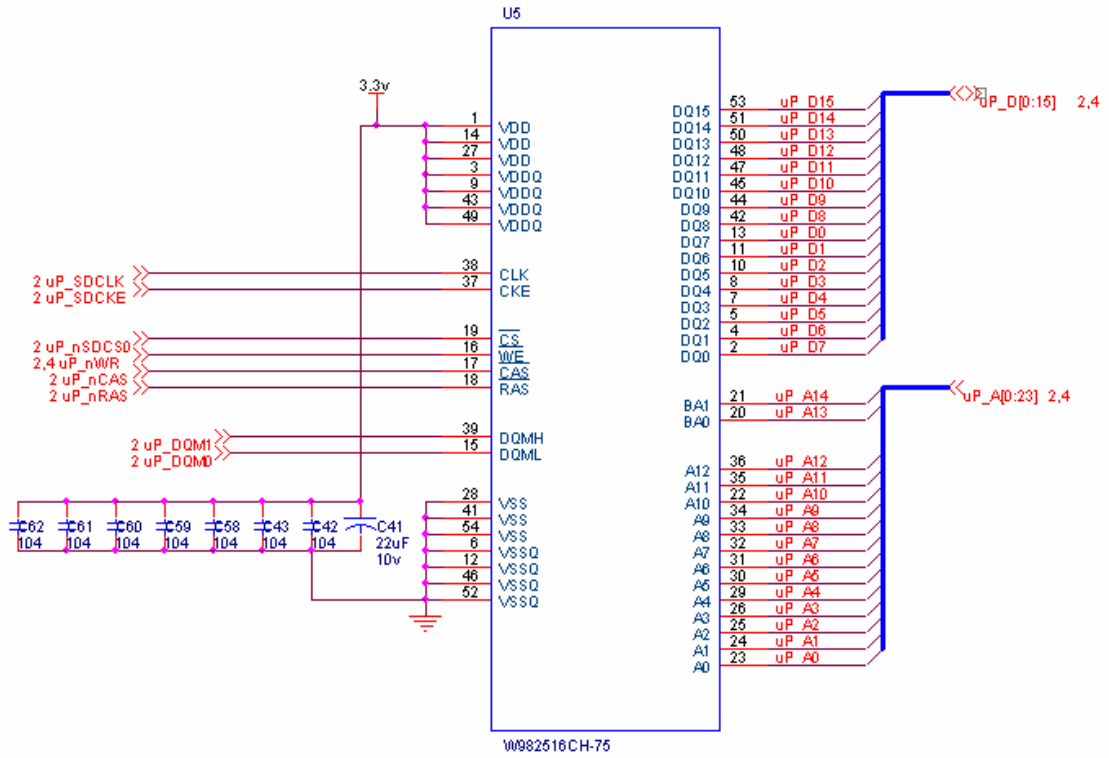

圖-APP. 7　32MB SDRAM：W982516CH-75（[26]）接腳圖

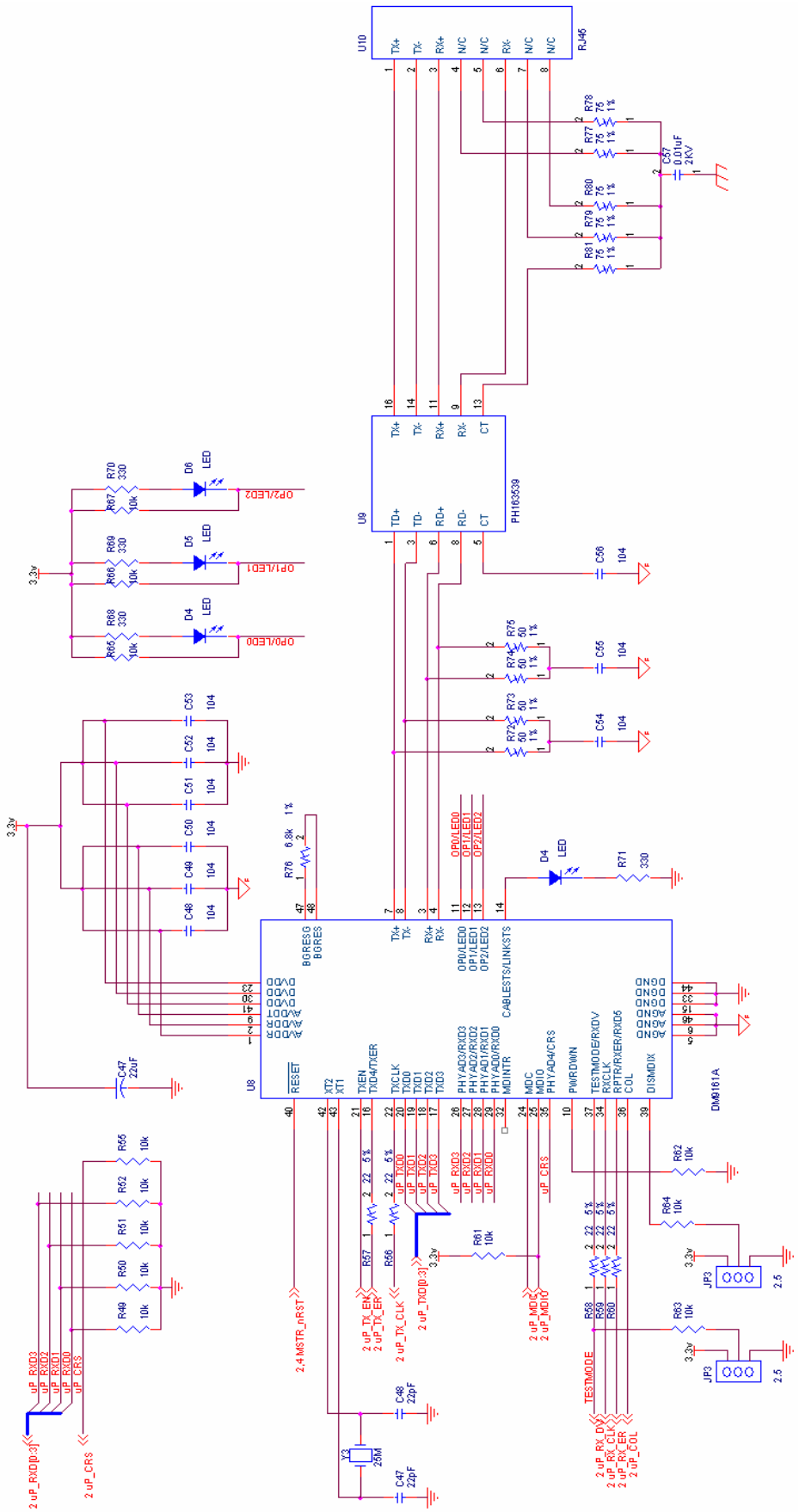

圖-APP. 8　PHY transciever：DM9161A（[43]）

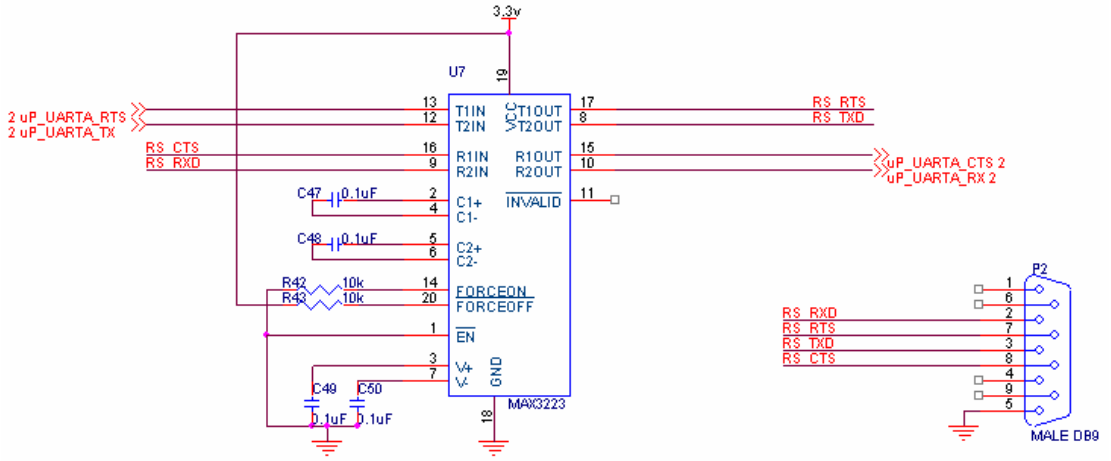

圖-APP. 9　RS-232 transceiver：HIN232

附錄 K.　模擬板設計要點說明

I.　模擬板工作電壓與功率消耗

模擬板需求 1.8v、3.3v、5v 三種不同電壓，1.8v 供給 MCU 核心電壓，3.3v 供給 flash、SDRAM、MCU 的 I/O，5v 供給 RS-232 及 RS-485 介面 IC。

5v 及 3.3v 由電源穩壓 IC（regulator）產生，1.8v 由 Sharp LH79525 內部由 3.3v 再產生。為了防止電源遭受突波干擾，應在每個電源腳位怖局加一顆小的電容，典型值為 0.1uF 或 0.01uF。

模擬板功率消耗以用料各 IC 元件規格額定值做估算加總，本系統模擬板約消耗 800(mA)負載電流，當作選定 regulator IC 的最大負載電流參考，如果 regulator IC 供電瞬間最大電流小於模擬板需求，那麼極有可能發生系統不穩定狀況。

II.　開機記憶體選定

圖 3-3 說明了 Sharp LH79525 內部記憶體映射，當系統一開機 nCS1 被映至起始位址 0，模擬板以 8M NOR flash 儲存 bootstrap、核心、根檔案映像。表-APP. 1 列出可選用的開機裝置（[13]Table 3-1），重置信號關閉時將 PC7~PC4 的值鎖住，PC7~PC4 值用來選擇使用的 flash 規格，最簡易方式即用被動負載

，PC7~PC4 以 10k 負儀電阻連接 pull-high 或 pull-low 設定其值。

表-APP. 1　開機記憶體裝置選擇

PC[7:4]	BOOT CONFIGURATION
0x0	NOR Flash or SRAM; 16-bit data bus; nBLEx is LOW for reads
0x1	NOR Flash or SRAM; 16-bit data bus; nBLEx is HIGH for reads
0x2	NOR Flash or SRAM; 8-bit data bus; nBLEx is LOW for reads
0x3	NOR Flash or SRAM; 8-bit data bus; nBLEx is HIGH for reads
0x4	NAND Flash; 8-bit data bus; 3-byte address
0x5	NAND Flash; 8-bit data bus; 4-byte address
0x6	NAND Flash; 8-bit data bus; 5-byte address
0x7	NAND Flash; 16-bit data bus; 3-byte address
0x8	NOR Flash or SRAM; 32-bit data bus; nBLEx is LOW for reads
0x9	NOR Flash or SRAM; 32-bit data bus; nBLEx is HIGH for reads
0xA	Undefined
0xB	Undefined
0xC	NAND Flash; 16-bit data bus; 4-byte address
0xD	NAND Flash; 16-bit data bus; 5-byte address
0xE	I^2C
0xF	UART0

III. 系統重置與系統時脈

模擬板的重置信號 nRESETIN 連至 MCU 及 NOR flash、網路 PHY 晶片的重置腳位，nRESTIN 致能時重置處理器內部所有狀態，nRESTIN 或 nTRST 致能時 MCU 內部亦需重置 TAP（test access point），意即系統重置時必須要重置 TAP。

Sharp LH79525 以 11.2896MHz 石英振盪為基頻，內部鎖相迴路根據除頻及倍頻參數產生 CPU 的工作頻率，本系統工作時脈（HCLK）調整為 50.802MHz，AHB 工作時脈（FCLK）調整為 50.802MHz，其餘時脈依據[13]Table 1-2 作調整。

IV. Jtag 介面及 UART0 介面測試

Jtag 用來當作除錯界面，以 Banyan-U（[32]）連上模擬板的 Jtag port，嘗試抓取模擬板上 Sharp LH79525 的處理器內核，正確無誤顯示為 ARM720 核心，如所示。

啟動 GDB server，再啟動 GDB，透過 Jtag 下載 UART0 測試程式至模擬板上 Sharp LH79525 內部的 SRAM 內。由 GDB 除錯視窗設置中斷點，讀取暫存器與記憶體內資料，將 UART0 所外接的 RS-232 埠連至虛擬終端機（minicom），如果能正常接收鍵盤資料並顯示於終端機上，則表示 UART0 的 RS-232 介面正常。

V. Nor flash 和 SDRAM 介面

NOR flash 介面與一般 SRAM 一樣，附錄 D 說明時序參數調整，附錄 G 介紹幾種不同燒錄 NOR flash 的方法。

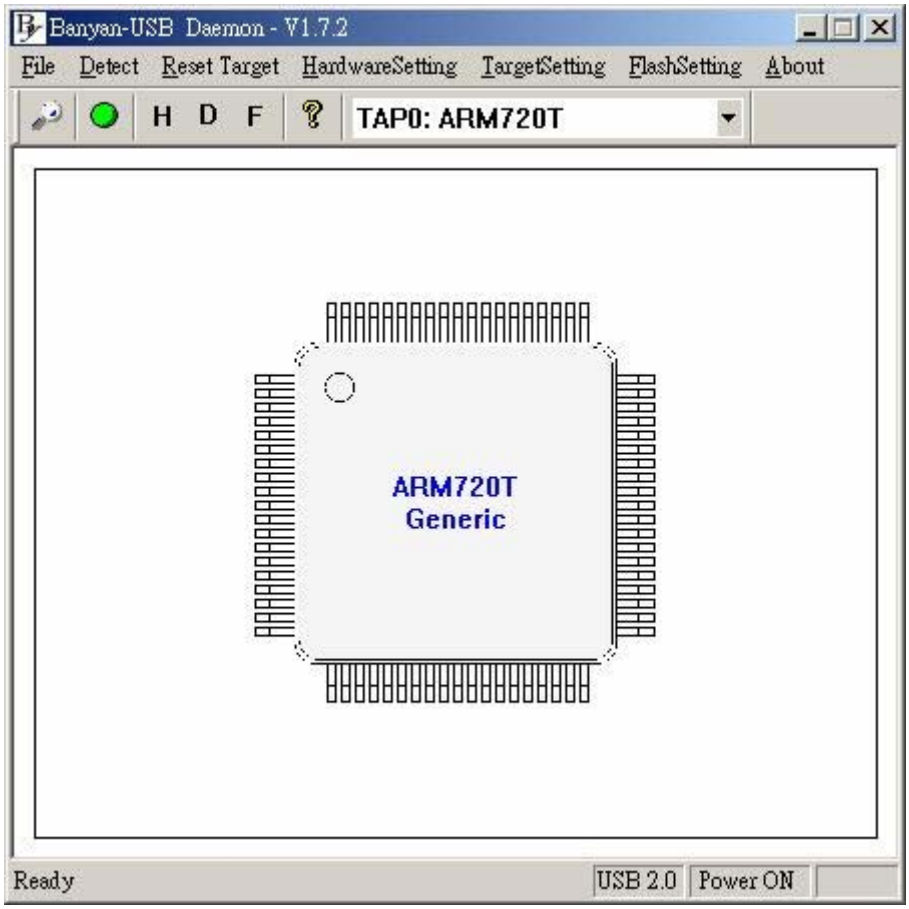

圖-APP. 10　Banyuan-U 抓取模擬板結果

SDRAM 位址分開行、列位址送，Sharp LH79525 的腳位 A14~A0 用來傳送 SDRAM 位址，各種不同容量、word 長度不一的 SDRAM 其行、列位址分怖亦不同，。以 W982516 內部大小 256Mb，其 word 長度 16 位元，意即 W982516 大小為 16M×16，參照[13] 7.4.3 節節錄其腳位映射分怖如下。SDRAM 初始化於 bootstrap 時完成，詳細步驟如附錄 C。

16-BIT WIDE DEVICE 256M SDRAM (16M × 16, RBC)															
External Address, A[14:0]	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory Device Connections	BA1	BA0	12	11	10/AP	9	8	7	6	5	4	3	2	1	0
AHB Address To Row Address	11	10	24	23	22	21	20	19	18	17	16	15	14	13	12
AHB Address To Column Address	11	12	—	—	AP	—	9	8	7	6	5	4	3	2	**

VI. PHY 晶片介面

本模擬板的 PHY 晶片選用 DM9161A（[43]），PHY 晶片支援標準 MII 介面，PHY 內部控制或狀態暫存器統稱為 MII 暫存器，用來設定 PHY 的工作模式及反應當前狀態，MAC 層藉由腳位 MDIO 以序列方式存取 MII 暫存器。[43]第八章列表 MII 暫存器，編號 0~6 為標準，編號 7~15 保留未用，編號 16 後各 PHY 廠家自行進階使用，一般只需用到編號 0~6 暫存器。

一個 MAC 層晶片可串多顆 PHY，因此 MAC 層存取 MII 暫存器必須指定 PHY 位址，PHY 晶片比對 MDIO 送來的存取指令內的 PHY 位址是否符合，符合表示存取該顆 PHY 晶片。圖-APP. 8 所示之 PHYADR[4..0]指定 PHY 位址，於系統重置時被栓入。

OP[2..0]選澤 PHY 啟動時的速率及雙工模式，自動協議模式（Auto-negotiation）最有彈性。實體信號 TX+/TX-及 RX+/RX-為差動信號，單獨用類比示波器抓取 TX 端看到的波形為 sinusoidal，很難由此去判讀資料，PHY 晶片應用時的除錯以 MII 信號及 MII 暫存器搭配。

IEEE 標準 PHY 層傳輸距離為 100 公尺，差動信號經過網路隔離變壓器的

差模耦合的線圈增強訊號，並將直流分量濾除，所以不會因兩端系統接地電壓差異而造成問題。選用網路隔離變壓器（magnetic）要依據 PHY 晶片廠商建議，它與 PHY 晶片功率及電磁效應關係密切，PHY 層信號和網路隔離變壓器佈局時遵守 PHY 廠商的佈局建議（[44]），否則容易出現無法預期的 EMI 干擾。

附錄 L. 原始碼重要檔案索引列表

表-APP. 2 原始碼重要檔案索引列表

索引	檔案或目錄名稱	說明或重要函式
1	$(ROOTDIR)	Linux source home directory
2	$(LINUX_INC)：＝ $(ROOTDIR) /include	標頭檔目錄
3	$(ARCH_SRC)：＝ $(ROOTDIR) /arch/arm	ARM架構平台相依的主要原始碼目錄
4	$(MACH_SRC)：＝ $(ROOTDIR) /arch/arm/mach-lh7952x<new>	LH79525的機器相依主要原始碼目錄
5	$(DRIVER_DIR) ：＝ $(ROOTDIR) /drivers	週邊驅動程式主目錄
6	$(LINUX_INC)/asm-arm/procinfo.h	ARM處理器資訊描述子結構
7	$(ARCH_SRC)/mm/proc-arm720.S	ARM720處理器資訊描述
8	$(ARCH_SRC)/kernel/head.S	硬體平台檢測，建立第一階段虛擬記憶的描述表
9	$(LINUX_INC)/asm-arm/mach/arch.h	機器平台資訊描述子結構
10	$(MACH_SRC)/arch-lpd7952x.c<new>	註冊LH79525機器平台描述子、IO資源描述子
11	$(ARCH_SRC)/arch/arm/tools /mach-types<mod>	機器的id，每一台被移植支援的目標機器要有一個唯一的註冊id
12	$(ARCH_SRC)/kernel/setup.c<mod>	*setup_arch*，解析tag參數
13	$(LINUX_INC)/asm-arm/setup.h	tag參數資料結構
14	$(LINUX_INC)/asm-arm/arch-lh7952x /memory.h<new>	LH79525的記憶體參數及虛擬映射巨集
15	$(LINUX_INC)/asm-arm/arch-lh7952x /vmalloc.h<new>	vmalloc位置配置定義
16	$(ARCH_SRC)/mm/init.c	*paging_init*建立第二階段虛擬記憶的描述表

17	$(ROOTDIR)/init/main.c	*start_kernel*
18	$(MACH_SRC)/time-lh7952x.c<new>	LH79525系統timer初始化
19	$(ARCH_SRC)/kernel/entry-armv.c	例外處理
20	$(ARCH_SRC)/kernel/entry-common.S	系統呼叫前置、後置處理
21	$(ARCH_SRC)/kernel/irq.c	*request_irq*、*setup_irq*設定中斷服務程式（ISR）
22	$(ARCH_SRC)/kernel/time.c	
23	$(MACH_SRC)/time.c<new>	與硬體相依的timer初始化原始碼
24	$(LINUX_INC)/asm-arm/mach/irq.h	中斷資源管理描述子資料結構**struct irqdesc**
25	$(LINUX_INC)/interrupt.h	中斷**struct irqaction**結構
26	$(MACH_SRC)/ irq-lh7952x.c<new>	中斷控制器初始及設定
27	$(ARCH_SRC)/kernel/vmlinux.lds.S	ARM Linux核心連結草本
28	$(LINUX_INC)/asm-arm/mach/ map.h	建構虛擬記憶體分頁表函式原型，map位址區段描述子
29	$(ARCH_SRC)/mm/mm-armv.c	ARM架構相依的記憶體管理原始碼
30	$(DRIVER_DIR)/serial/serial_lh7952x.c <new>	序列埠硬體驅動程式
31	$(DRIVER_DIR)/serial/serial_core.c	序列埠核心層運作方法
32	$(DRIVER_DIR)/char/tty_io.c	tty層核心處理函式
33	$(DRIVER_DIR)/char/n_tty.c	N_TTY line discipline
34	$(LINUX_INC)/linux/console.h	操控台描述子
35	$(LINUX_INC)/linux/tty.h	**struct tty_struct**，tty層
36	$(LINUX_INC)/linux/tty_ldisc.h	行線規程結構**struct tty_ldisc**
37	$(LINUX_INC)/linux/tty_driver.h	**struct tty_driver**
38	/include/termios.h	**struct termios**
39	$(LINUX_INC)/linux/device.h	**struct device** **struct platform_device** **struct device_driver**
40	$(LINUX_INC)/linux/netdevice.h	**struct net_device** **struct net_device_stats**
41	$(LINUX_INC)/linux/ioport.h	**struct resource** 巨集**request_mem_region**
42	$(LINUX_INC)/kernel/resource.c	硬體resource維護方法
43	$(LINUX_INC)/drivers/net/arm/	emac driver (base on

	emac-lh7952x.c<new>	ibm_emac、fec_8xx）
44	$(LINUX_INC)/include/linux/mtd/partitions.h	**struct mtd_partition**
45	$(LINUX_INC)/include/linux/mtd/mtd.h	**struct mtd_info**
46	$(LINUX_INC)/drivers/mtd/maps/physmap.c<mod>	mtd裝置bank、分割區定義
47	$(LINUX_INC)/drivers/mtd/cmdlinepart.c	核心參數mtdparts處理方法
48	$(ROOTDIR)/Documentation/kbuild/kconfig-language.txt	核心組態參數選項描述語法
49	$(ARCH_SRC)/kernel/trap.c	例外向量表初始化trap_init
50	$(ARCH_SRC)/mm/ioremap.c	ioremap，將實體記憶區段映射至虛擬位址