國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

動態精細分工異質雙核心排程器的設計與分析

Design and Analysis of a Dynamic Fine-Granularity Task Scheduler for Heterogeneous Dual-Core Platforms

研 究 生：李國丞

指導教授：蔡淳仁　教授

中 華 民 國　九 十 五　年 六 月

Design and Analysis of a Dynamic Fine-Granularity Task Scheduler for Heterogeneous Dual-Core Platforms

研 究 生：李國丞　　Student：Kuo-Cheng Lee

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

動態精細分工異質雙核心排程器的設計與分析

摘要

本論文主旨在於設計和分析非對稱式多核心平台上動態精細分工排程器的效能。當系統平台上有著多顆不同類型的處理器，並且以某一處理器爲控制整個系統的依據，即爲非對稱式異質多核心平台。通常在這種平台上，因爲處理器架構不同，所以一個程式的各個程序必須事先決定好要在那一個處理核心執行，而預先編成該核心的執行碼，而不是動態決定每一個程序要在那一個核心執行。爲了增進複雜的嵌入式系統效能，過去我們曾提出一個能讓應用程式中的關鍵程序在執行時才動態決定要在那個處理核心執行的架構[1][22][24]。本論文的主要目標則是將這樣的一個作業系統核心實作出來。另外，爲了能動態地把一個程序分配給不同指令集的處理核心執行，本論文也設計了一個解決方案。

整個系統的實作是以 eCos 爲基本作業系統；在其 kernel 內加入一個 Dispatcher，藉由該 Dispatcher 監控各核心工作的負載，動態即時地分配工作。實驗所用的平台爲 TI 的 OMAP 5912 OSK 發展平台。OMAP 5912 OSK 處理器採用 ARM (RISC 核心) 和 C5510 (DSP 核心) 的雙核心架構.。在本論文設計的系統中，由 ARM 負責執行 Dispatcher，針對每一個註冊爲雙核心的工作，進行動態分配，透過這兩種不同類型的處理器的密切合作, 來達到更高的效能。另外，本論文所提出的新的程式設計模式及作業系統 API，也讓使用者對於應用程式只需做最少的修改，便能充分利用異質雙核心平台的能力，提高應用程式移植的便利性。

Abstract

In this thesis, we have implemented a dynamic fine-granularity task scheduler for heterogeneous dual-core platforms. The target platform contains more than one heterogeneous processor cores and one of them controls the whole system. For a platform which contains heterogeneous processor architecture, application designers typically solve the task partition problem during development time. Since the applications are statically partitioned, subtasks of an application can not be dynamically scheduled across processors at run-time. In order to increase performance of complex multi-tasking embedded systems, we have proposed a dynamic runtime scheduling architecture [1][22][24]. In this thesis, a system that realizes this vision is implemented. Furthermore, we have designed a solution that solves the problem of dynamically executing a function on two processor cores of different instruction set architecture.

The system is based on the eCos embedded operating system. A new dispatcher is added to the eCos kernel to monitor the runtime loading of each processor and dispatch tasks accordingly. The dual-core platform used for system implementation is the TI OMAP 5912 OSK development board with an ARM core (RISC) and a C5510 core (DSP). The ARM core is in charge of running the dispatcher and the main application threads. For each function call that is registered as a dual-core module, the dispatcher will dispaches it either to ARM or DSP on-the-fly.Through the proposed tightly-coupled cooperation approach, the propsed system can achieve higher performance, especially when multiple applications are running on the platform. We also proposed a new programming model and a system API to facilitate porting of a

regular software to take advantage of the heterogeneous dual-core platform.

章節目錄

1. 簡介 11

2. 相關研究 14

2.1. 多核心平台排程演算法 14
2.2. 對稱式多核心平台與非對稱式多核心平台 16
2.3. 同質多核心平台系統下的非對稱式排程 18
2.4. 動態式分工及排程 20
2.5. 共享資源控制 21
2.6. 靜態式分工 23

3. 理論與實作背景 25

3.1. OMAP 5912 Application Processor 25
3.2. OMAP 5912 Starter Kit：OSK 5912 (OMAP 5912 OSK) 30
3.3. eCos 31
3.3.1. eCos Overview 31
3.3.2. Configure Tool 32
3.3.3. Component 32
3.3.4. HAL 34
3.3.5. The Kernel 38
3.3.6. RedBoot 44

4. eCos至OMAP 5912 OSK的移植方法 47

4.1. 搜集硬體資料 48
4.2. eCos configtool使用方法 49
4.3. 系統修改 53
4.3.1. har_arm9.cdl 53
4.3.2. hal_cache.c 53
4.3.3. basetype.h 54
4.3.4. mtl_OSK5912_rom.h 54
4.3.5. Osk5912_misc.c 55
4.3.6. hal_arm_arm9_omap5912.cdl / hal_OSK5912.cdl 56
4.3.7. Omap5912.h 57
4.3.8. hal_omap5912_setup.h / hal_platform_setup.h 57
4.3.9. hal_platform_ints.h 58

4.3.10. omap5912_redboot_comds.c / OSK5912_redboot_comds.c........................58
4.3.11. omap5912_diag.c / hal_diag.h 59
4.3.12. OSK5912_misc.c 59
4.3.13. Plf_io.h..59
4.3.14. plf_stub.h 60
4.3.15. Redboot_ROM-net.ecm..60
4.3.16. ser_omap5912.cdl 61
4.3.17. devs_eth_OSK5912.inl 62
4.3.18. OSK5912_eth_drivers.cdl..64
4.4. Redboot 移植完成..64

5. 異質雙核心動態分工排程器實作 65

5.1. Scheduler API 66
5.2. Service Registrar和Core Service Table 69
5.3. Dispatcher 69
5.3.1. Task Dispatcher..71
5.3.2. Task Terminator 73
5.3.3. Loading Tables..73
5.4. 雙核心溝通方法..74
5.5. DSP API 77

6. 實驗結果 78

6.1. 實驗環境..78
6.2. 動態排程實驗..79
6.3. 相異處理器實驗..81
6.4. 相異bit rate實驗 85
6.5. DSP delay實驗..87

7. 結論與展望 91

8. 參考文獻 93

9. 附錄 95

9.1. 附錄 1: configtool使用流程..95
9.2. 附錄 2: Register 修改 98
9.3. 附錄 3: Interrupt vector 修正 103
9.4. 附錄 4: 應用程式測試 106
9.5. 附錄 5: 需修改的系統檔案 109

9.5.1. Flash 109
9.5.2. Ethernet 109
9.5.3. Serial port 109
9.5.4. HAL 110
9.5.5. ARM 926EJS 111

List of Figures

Figure 1. OMAP 5912 功能區塊圖 26
Figure 2. OSK 5912 正視圖 30
Figure 3. eCos系統區塊圖 34
Figure 4. HAL 啓動流程圖 36
Figure 5. Kernel啓動程序 39
Figure 6. MLQ排程器運做圖 40
Figure 7. Bitmap排程圖運做圖 41
Figure 8. RedBoot ROM monitor 特徵區塊圖 46
Figure 9. Redboot開機畫面 64
Figure 10. 系統架構 66
Figure 11. 一般單核心 idct函數的使用方法 67
Figure 12. HMP scheduler下 idct使用方法 68
Figure 13. Dispatcher架構圖 70
Figure 14. Loading table 74
Figure 15. Parameter table 76
Figure 16. configtool 1 95
Figure 17. configtool 2 96
Figure 18. configtool 3 96
Figure 19. configtool 4 97
Figure 20. 範例程式: hello.c 106
Figure 21. 範例程式: serial.c 107
Figure 22. 範例程式: simple-alarm.c 107
Figure 23. 範例程式: twothreads.c 108

List of Tables

Table 25. 新增Register 98
Table 26. 新增Register 102
Table 27. 修改interrupt vector 103
Table 28. 新增interrupt vector 104

1. 簡介

在街頭上，隨處可見用手機在聊天談事情的人們；或是掛著耳機，利用 mp3 播放器在聆聽音樂的青少年；上班族也幾乎用 PDA 取代了以往紙本記事的習慣。如此廣大流行的手持式裝置，如今越來越擴展它的應用層面，例如手機支援百萬像素的拍照功能，使手機也有了數位相機的能力；3G 的影像電話功能，不只是對話，也能同時看到對方的表情，讓遠距溝通變得更生動；mp3 播放器的文字瀏覽，秀圖系統甚至影片播放功能，令單純聽音樂的 mp3 播放器提高其附加價值，搖身一變成爲微形的數位娛樂中心；另外 PDA 的衛星定位導航功能，打破了我們一向認爲 PDA 只不過是個可以帶著走的超小型桌上電腦的既有想法，發揮了在移動力上的特性。相信未來必定會推出更強大更高品質的應用，使得嵌入式系統的複雜度迅速地提升，相對的嵌入式系統的工作效能也必需提高。

爲了諸如此類眾多新的功能，以多媒體應用來說，嵌入式系統必需完成極大量的多媒體資料處理工作；換句話說，嵌入式系統要在相同甚至更短的時間內，處理更大量的資料，做更多的運算工作。提高嵌入式系統的能力是必要的。就過去電腦系統的發展史來看，提高系統的能力不外乎是提高處理器的能力爲主，而處理器的能力就直接關係到它每秒可以運算的次數，每秒可以執行的計算量，亦即處理器的頻率。然而目前利用此一概念發展的單一核心嵌入式平台已不敷使用。

考慮手持裝置的特性：輕巧以及移動力佳。嵌入式平台便有了體積上的限制，其中便影響到一個重要的耗電量的問題。體積上的考量，手持裝置無法配置大容量大體積的電池，同時顧及其移動的特性，也無法接受一再需要補充電力的要求。提高核心頻率會消耗大量電力，這一點會成爲嵌入式平台的致命傷，再者，高核心頻率相伴而來的是產生許多的熱量，散熱方面也是一個難題。在現今市場上，整體行動裝置效能的提升不是利用提高核心頻率的方法，而是以增加核心數

(處理器數量)來平衡高核心頻率需求及大耗電量和高熱能產生的缺點。這種多個處理器的架構，我們稱之為多核心架構(multiprocessor architecture)。

事實上，異質多核心架構(heterogeneous multiprocessor architecture)在嵌入式系統的發展已被業界廣泛地使用，例如德州儀器公司的 OMAP(Open Multimedia Application Platform)，以及 Freescale 的 MXC。在非對稱式多核心系統晶片(system-on-chip: SoC)架構裡，會有顆一般功能的處理器(general purpose processor: GPP)核心，做為嵌入式系統作業系統的控制核心，配上一顆數位訊號處理器(digital signal processor: DSP)核心。DSP 可以大量即時處理多媒體資料，如 MPEG 1、MPEG 2、MPEG 4 或是音訊資料等等。以德州儀器公司的 OMAP 5912 OSK (OMAP Starter Kit)為例 [1]，其 GPP 採用 ARM 公司 ARM926EJS，DSP 則是德儀自行研發的 TMS320C55X。研發人員可以依照資料處理的性質，將工作分配給 OMAP 架構微處理器中的 ARM 微處理器或者是 DSP 微處理器去處理。非對稱式多核心架構可有效率利的處理嵌入式系統上的工作(task)，發揮系統的最大效能，特別是對於多媒體的應用程式有令人亮眼的表現。

現存的即時作業系統(real-time operating system)對於非對稱式多核心架構大部份是採用靜態式分工(statically partitioned)的方法。所謂靜態式分工方法是系統設計時研發人員就做好工作的分配，屬於控制流程的工作就交由 GPP 執行，屬於多媒體運算處理的工作多交由 DSP 執行。在這種分工架構之下，有兩個不同的 schedulers 為兩顆核心獨立運行已分配好的工作。換句話說，兩顆核心各自處理已分配好份內的工作，完成之後，在下一個工作來臨之前是閒置的狀態，因為在獨立的視野裡，已是最好的效能發揮。這類型的分工方式在傳統行動通訊平台及應用程式環境下是相當有效的法。過去常用的 GPP 核心在特殊工作處理的功能性和速度都有所不足，意即 GPP 沒辦法勝任 DSP 的工作。並且過去的嵌入式應用程式環境通常是單純的前景/背景(foreground/ background)工作模式，所以不需用到複雜的動態排程。

但是新一代多媒體應用會拓展到更寬廣的層面，再加上硬體裝置上有了新的提升。首先，多媒體應用程式已經複雜到一個境界，爲了提升系統效能和減低能量的消耗，必需用動態調整兩顆核心的工作量取代系統設計時做好的工作分配。其次是 GPP 的能力已被大幅提高，可以幾乎和 DSP 等速地處理某些多媒體資料，換句話說， 在這些情況下，GPP 可以用來分擔 DSP 的工作負載。接著是多媒體應用程式在記憶體和計算量的需求已經大大超越過往，多媒體資料經常會被包裝成運輸串流(transport stream)，往返於兩顆核心之間，但是在執行時核心之間溝通的成本並非固定，譬如傳輸時電力的消耗與總電量的關係、工作有沒有完成時間的限制(deadline)等……，有太多因素要考量，是不可能在系統設計時就預測到並且做好資料傳輸的設定。著眼於系統效能，靜態式分工系統設計不再合適，即時作業系統排程器在設計上要有新的突破。

考慮以上種種原因，我們便提出一種新的動態精細分工式(tightly-coupled)作業系統排程器[21]，[22]，這種新的排程法會由單一排程器監控各顆異質核心的工作狀態，並能動態地分配工作給當下最合適的處理器核心。排程視野的廣度上，由系統設計時就定好的靜態工作分配延伸到執行時的動態分配；而深度上，考量整個系統即時的狀況，做出最適當的工作分配並減少微處理器的閒置浪費，取得比靜態式分工系統更大的效能發揮。

2. 相關研究

這一章將會介紹此領域的相關研究。依順會介紹多核心平台著名的排程演算法，對稱多核心平台及非對稱式多核心平台，非對稱式多核心平台系統的排程，動態排程，共享資源的控制，非對稱式系統晶片 SoC，和靜態式分工系統。

2.1. 多核心平台排程演算法

多核心處理器架構排程器的研究越來越受到重視。過去十多年來，在實用上，多核心排程演算法的發展重點是放在對稱式多核心系統(symmetric multiprocessor system)上。多核心排程技巧在同質多核心平台部份可以被分成兩類，partition scheduling 和 global scheduling[23]。 Partition scheduling 是指每一個核心有自己的工作駐列(task queue)，包括 ready 駐列和 wait 駐列。工作排程的考慮會以各自區域的 priority 爲主，與其他處理器獨立。每一個工作一旦被分配到一個處理器，在其生命週期內都不會移到別的處理器。 Global scheduling 則是將所有準備完成的工作放在一個共同的 priority 駐列。最高 priority 的工作會被挑選放到一個工作量較低的處理器執行。這種 scheduling 模式在同質多核心的系統上表現較前者佳。

以下簡單列出一些常用的多核心排程演算法[3]:

- Rate monotonic： 每一個週期性的工作有固定的 priority，priority 的順序是根據該工作的執行頻率高低而定，例如要等待 interrupt 的工作，其 priority 相對較低。在 1973 年，Liu 和 Laylan 證明這個演算法是固定 priority 演算法中最理想的一種。

- Earliest Deadline First： 這種演算法可將週期性和非週期性的工作一起排程，主要概念是越早結束的工作越先執行。M. L. Dertouzos 在 1974

年證實當瞬間有許多工作等待執行時，此演算法是最有效率的。

- Deadline Monotonic： D.M.結合上述兩種演算-- priority 的給定除了根據該工作的執行頻率外，另外會再考慮 deadline 越早，priority 越高。
- Background Scheduling： 此種演算法同時處理 soft real-time aperiodic task 和 hard real-time periodic task。兩種型式的工作分別置入兩個不同的駐列。此演算法實用上雖沒有很高的利用性，但其優點在於實作很簡單。
- Pooling Server： P.S.可處理非週期性的工作。每個時間區塊一過，server 便服務下一個時間區塊可以執行的工作。若沒有工作在等待被執行，則會閒置 server，等到下一個時間間隔再甦醒。
- Deferrable Server： 此演算法類似上一個，但是若下一個時間區塊沒有等待被執行的工作，則 server 服務可能被服務的工作，而不是閒置 sever。
- Sporadic Server： S.S.使用於非週期性工作，可以增進其反應時間，使得非週性工作的效能追上週期性工作的效能。
- Dynamic Sporadic Server： 這個演算法利用 deadline 調整 priority，增進 Sporadic Server 的效能。
- Robust Earliest Deadline： 這是 1995 年 Buttazzo 和 Stankovic 發展的演算法，作用於 over loading 環境中的非週期性工作。此演算法不只可以減少 deadline 預測錯誤，也可降低系統 over loading 的程度。
- Constant Bandwidth Server： 這是在 1998 年 Buttazzo 和 Abeni 發展的演算法，用來解決即時多媒體應用的問題。例如在串流影音的系統中，對串流資料的傳輸和處理的 delay 和 jitter，必須要控制在一定的範圍內。
- Adaptative Bandwidth Reservation： Abeni 和 Buttazzo 在 1999 年提出

對 constant bandwidth server 的改良。對於執行時間未知的工作所能分配到的處理器的頻寬可以經由 Adaptative Bandwidth Reservation 來控制。在這裡，頻寬(bandwidth)一詞指的是處理器分配給工作的時間或是工作被執行的週期。

2.2. 對稱式多核心平台與非對稱式多核心平台

前面提到目前實用上多核心作業系統的排程演算法大部份都是以對稱式的多核心平台爲目標，比方說，Satoshi Kaneko et al在 2004提出的一個多核心平台[4]。這個 600MHz單晶片多核心平台包括兩個M32R 32-bit CPU核心，一個 512-KB共用的SRAM，和一個內部分享的pipeline bus。

這個平台是由 0.15um CMOS 製程製造，適用於嵌入式系統。此多核心平台是對稱式的多程序處理平台，並且支援 modified-exclusive-shared-invalid (MESI)的快取統一協定。該系統繼承了先前單晶片多核心平台的諸項優點，並針對嵌入式處理器做了最佳化，以使得系統效能增加的同時也能減低電力的消耗。爲了增加核心的效能，他們在平台內部置入一個共享的 pipeline bus。此 bus 的特性是低延遲和每秒 4.8 G-bit 的大頻寬。此外也用多個低耗電技術，例如擁有不同使用電力的模型選擇: 睡眠模式、工作模式、和等待模式。不同系統情況下，不同核心甚至週邊有不同的模式選擇，以達到最高的省電約 18.4%。使得此多核心平台在 600MHz 1.5V 之下功作僅消耗 800 mW，待機時更只耗 1.5mW。

有些應用，如 3G 通訊和嵌入式多媒體應用，會同時執行控制的工作和大量資料處理的工作。一般實作上，爲了達到最佳的性能／耗電量比值，異質多核心(Heterogeneous Multi-Processor)的架構是一般業界常用的設計方法[1], [13]。例如飛利浦半導體部門發展了一套 Silicon System Platform (SSP)。SSP 是零件的工具箱，是一種一般性、開放性和可程式化的架構。主要用來產生有軟體和硬體 IP

blocks 的特定應用產品領域。過往研發新產品，可能必需打造整個新平台架構，付多相當的成本花費。利用 SSP 概念，爲新應用產品而修改的架構會比試著去產生整個新架構更有實作的效率。使用 SSP 設計產品的速度很快而且技術風險低，因爲架構中軟體硬體的功能性已經驗證過，而且還可以結合其他工作元件更容易達到設計的目標。同時其中有很大的空間讓設計團隊創造不同市場需求的產品；一系列的產品由入門到進階的產品，只需在平台上增減功能區塊，就可有效地減少開發時間及成本花費。日後使用者甚至可以隨著更新軟體的版本來增強或增加產品的功能性。飛利浦的 Nexperia 平台是一個單晶片系統的 SSP，用來開發數位視訊產品。Nexperia 平台上主要包括 MIPS 處理器和飛利浦的 TriMedia VLIW 媒體處理器，及其他 IP 元件。結合 MIPS 及 TriMedia 兩種不同的計算核心，整合成單晶片系統。飛利浦利用此平台創造出多功能的機上盒，它可以即時解碼多個視訊串流、執行數位錄製、壓製訊號用於視訊電話、瀏覽網站和收發電子郵件等多項功能。其他如德州儀器(TI),飛思卡爾(Freescale)和 Toshiba 等知名大廠都有自己的異質多核心平台，本論文在 TI OMAP 上實作，稍後章節將會詳細介紹 OMAP 平台。

在軟體的開發過程中，軟體測試是很重要而且很昂貴的一部份。有一種軟體測試的方法稱之爲資料流測試(Data Flow Testing)，使用資料流測試可以決定一個軟體的測試是否充份且完整。Harrold 提出一個新的方法把整個資料流測試工作量切割成適當的大小[5]。這些測試的工作量可以靜態地也可以動態地接受排程。也可以改變成適合共用式記憶體或分散式記憶體的環境。在[5]中把資料流測試演算法實作出單一核心平台的版本和多核心平台的版本，並根據大量的軟體實驗來驗證資料流演算法的正確性。另外，這些實驗也可證實多核心平台的效能優於單核心平台。平均效能上多核心平台比單核心平台加速 1.7 倍。

Annavaram 等人討論過非對稱式多核心系統的效能優於對稱式多核心系統的看法[6]。激發此篇論文的研究動機有下例三點：首先，單晶片多核心平台上

CPU 核心的數目增加，同時間可以執行的運算量上升。第二，可以利用單晶片多核心架構優點的多執行緒軟體變得更流行。因為演算法的性質，這些多執行緒程式被分階段連續的執行。然而 Amdahl's law 指出平行化程式的加速將會被計算的連續部份限制。第三，不斷增加的晶片整合層級和逐步降低使用的電壓結合使得如何減少電量的耗損成為首要注重的設計限制。此論文的目標是最小化多執行緒程式的執行時間。該執行緒包含平行處理和連續處理的階段，同時也保要有多核心單晶片的電力消耗限制。為了減少 Amdahl' s law 影響，在論文中對於電量花費的計算是根據可獲得的平行度來決定處理的指令數，並以這些指令花費的電量為準。使用該等式，電力 ＝ 每個指令的能量(Energy per Instruction: EPI) * 每秒指令數(Instructions per second: IPS)。假設電力固定的情況下，因此限制平行量的多核心單晶片是低 IPS，會花較多的 EPI。相反地，高平行量時，會花較少的 EPI。根據[6]的實驗，在相同的耗電量前題下，一個複雜的系統在使用非對稱式多核心、多執行緒執行時，會比對稱式多核心系統增加百分之三十八的效能。

隨著近年來多媒體裝置的流行，多執行緒平台研究關注的焦點已由對稱式多核心系統轉移到非對稱式多核心系統。非對稱式多核心系統比對稱式多核心系統有更佳的效能/時脈比，因此在多不同工作執行時非對稱式多核心系統更適合嵌入式裝置。

2.3. 同質多核心平台系統下的非對稱式排程

前面提過，異質多核心平台在處理通訊及多媒體相關工作時可以得到最佳的效能/時脈比，但目前並沒有論文是針對異質多核心平台探討動態自動排程的設計。不過倒是有不少論文是針對同質多核心平台研究非對稱式動態自動排程的可行性。Wendorf 等人 提出多個工作分配和排程方法[7]，範圍由非對稱 master/slave 排程到對稱式排程。他們在許多情況下測試這些分配和排程方法。對於非對稱系統，結果顯示 OS Preempt 策略幾乎在所有的清況下都有最高的效

能。作業系統的工作的 priority 相對高於一般應用程式，而在兩者有相同 priority 時，作業系統的工作可以較優先得到處理器的使用權，稱之為 OS Preempt。相對於其他策略 OS Preempt 可以減少時間耗損百分之三十到百分之六十。在許多測試情況下非對稱的系統和對稱式的系統幾乎有一樣的效能，甚至前者有優於後者的情況。重要的是，在對稱式系統中，作業系統工作因 functionality partition 仍需在全部可以使用的處理器中選擇執行者，相較之下非對稱式系統指定單一處理器微理作業系統工作，更容易實作。結果也指出，在不同工作分配和排程演算法下，process switch overhead 和多處理器之間的對於分享資源的競爭是決定系統效能的因素中相對較不重要的。

Greenberg 提出了一個簡單的 master-slave 架構[8]。在一些電腦作業系統下，一個程序(process)可以在 user mode 或是 system mode 的模式下執行。一個 user mode 的程序可以在執行中進行一個系統呼叫(system call)變成 system mode。這個程序在結束這些呼叫後便回到 user mode。在 master-slave 的多核心架構下，系統呼叫如 kernel call 只可以在 master 核心執行，剩下的呼叫就被視為如 user call，和其它工作一樣可以在 master 核心或 slave 核心執行。當 slave 核心上的 user mode 程序欲使用 kernel call，slave 核心會將該程序交給 master 核心處理，而非由 slave 核心處理。在 Greenberg 提出的設計中，工作會先在兩個駐列等待。一個駐列稱為 master 駐列，另一個則稱為 slave 駐列。Master 駐列的工作都是在系統模式，而 salve 駐列上的工作都是在使用者模式。

如前所述 master 駐列是只在 master 核心執行的駐列，slave 駐列卻是可在 master 核心或 salve 核心執行。此論文利用兩種簡單又實作的排程演算法來平衡排程的彈性和 queue-switching 的成本花費。最後並提出一個分析公式，用來測量硬體和 work load 參數，同時考量 master-slave 系統的電力和限制，並進而尋找到非對稱式多核心系統中最佳 slave 核心的數量。

2.4. 動態式分工及排程

許多對時間有嚴格要求的應用都需要動態排程方能達到預定的效能。Manimaran 等人把一個系統的效能定義為該系統能在 deadline 之前完成的工作所佔的百分比[9]。在這篇論文中，他們提出在多核心系統上使用的一種演算法，可以動態地對可執行的即時工作進行排程，並具有容錯的功能。系統的運作是基於以下兩個限制條件: 一、每一個工作一旦分配給處理器以後，是不會被打斷的(non-preemptive)。二、每個工作有兩種版本，這個假設是用來改善處理器錯誤的問題並可以得到較高的效能。

系統的內有 N 個處理器和 N+1 個駐列，其中包含了 N 個 local 駐列和一個 global 駐列。每一個處理器和一個 local 駐列為一個組合。排程器自 global 駐列中取得最高 priority 的工作，動態地依系統狀態和各個處理器的狀態，決定將置入哪一個 local 駐列。提出的演算法有下列三個技巧:

1)距離概念：決定 task 駐列中兩個工作版本的相對位置。

2)彈性的系統復原：在效能和容錯等級的取捨。

3)資源的回收：回收被判定為 deadlock 的工作和已完成的工作所分配到的資源。

利用動態排程方法和上述技巧系統的效能和容錯性達成應用上時間的限制。

Avritzer 等人發展出一個效能分析模組[10]。該模組對使用 load sharing 演算法的高度非對稱系統做效能的評估。load sharing 演算法是基於全系統的狀態進行排程工作。load sharing 演算法有兩種實作的層級，第一種層級在作業系統內部，稱為 kernel 層級或 shell 層級。第二種層級為使用者層級，在 shell 的前端。前者雖有效率上的優點，但是異質機器之間的相容性使得實作上十分困難。雖然後者的 overhead 比前者大，但有三個理由令此論文決定使用後者實作。第一、

不必考慮異質機器的相容性，容易實作。第二、對機器和使用者 load sharing 會透明化。第三、使用者利用 shell 前端控制可以決定要不要加入負載分享的機制當中。

Load sharing 的主心概念是要儘可能縮短整個系統的反應時間，執行方法是把工作分配給利用率低的機器。動態的 load sharing 可分成由傳送者初始化的型態和由接收者初始化的型態兩種。傳送者初始化的型態使用時機是系統負載不高時，接收者初始化的型態是系統呈現高負載時使用。此論文提出了一個分界型(threshold type)的 load sharing 演算法，此演算法會隨著某些分界值的變動而調整最適當的工作參數，例如每個機器上的工作數量。實作上該演算法的模型是建立以全系統為視野的全系統狀態馬克夫鏈和並計算出能在最差狀況下達到最小 latency 的系統。此論文的結論指出在非對稱式的環境下，小心地動態調整 load sharing 的演算法，會比靜態設定 load sharing 的演算法的效能有大幅增進。

2.5. 共享資源控制

Majumdar 提到多核心系統上程序之間會有競爭分享的資源[11]，例如變數就會儲存在分享記憶體上。保持資料一致性的機制不可缺少，如此才能確保系統的正確性。可是這種機制又通常會降低系統效能。這篇論文研究以多核心平台為基礎的應用程式為對象，如電話交換器和即時資料庫，控制分享資源的競爭以達到高度 throughput 及高度 scalability。將已存在的程式改成 re-entrance 或是將程序做適當的排程是兩種可實行的控制記憶體競爭方法。此論文著重於第二種方法。對數種控制資源競爭的排程演算法量化其結果，可以了解系統內部的行為和每一種演算法最重要的特性。結合數種排程演算法的特性的優點，衍生出混合式的控制資源競爭排程演算法： Hybrid-K。Hybrid-K 可以把所有程序執行時間縮短為依序執行每個程序所花費執行時間的 1/K。參數 K 代表系統增加的處理器數目。因此增進的效能會依 K 的增加而上升。然而需要注意的一點是實驗使用的處理

器數目最大只到 10，因此 K 大於 10 的情況尚待驗證。

Saewong 等人指出如何安排同時存取多個資源[12]，這是眾所皆知的一個 NP complete 的問題。在分散式即時系統之中，通常都是用 Decoupling 的方法來管理點對點延遲的系統。不幸地，當利用單獨的核心來管理多個資源時，Decoupling 的方法就會失敗。利用單獨核心管理資源方式的優點是可以減少衝突以全系統的觀點來分配資源的使用。例如控制核心可以利用裝置驅動程式、檔案系統或協定服務(protocol service)來控制相關的資源。控制核心我們稱之為 host 核心。Host 核心具有兩個角色：其一，host 核心如一般的核心可執行應用程式。其二，host 核心可以控制和管理其他 time-shared 的資源。此論文研究協同排程的控制和受控制資源的問題，提出合作排程伺服器(Cooperative Scheduling Server :CSS)。

CSS 是一個專用的伺服器，利用固定的一個處理器來控制眾多可以分享的資源，例如： 磁碟機和程序之間的溝通。下列兩個概念是 CSS 的目標基礎。首先，在一個控制器上(如 CPU)先執行一個非週期性的伺服器，該伺服器可以處理所有局部資源的使用要求。這表示 conjunctive admission control 是在控制端和受控制端一起實行的。接著，在應用程式層級的時間限制被分割進入多個階段，每一個階段都會被保證在一個特定的資源上完成。Real time file system (RTFS)是一個即時的檔案系統，它可以提供在 CPU 低負載時，對磁碟頻寬的保證。有了 file system CSS (FSCSS)，磁碟頻寬的保證也可以在高 CPU 負載和高磁碟工作量下達成。以下列出協同排程演算法設計需要考慮到的因素。第一、資源異質性產生的排程失誤問題。依受控資源的觀點，host 核心必須確保這些資源相對活動不能被其他更高 priority 的活動過份地延遲。依 CPU 的觀點，native CPU application 又必須保有完成的時間限制。因此會有 confliction 和 scheduling miss。第二、conjunctive admission control；每一個受控制資源的 admission control 必須不只是考慮自己擁有資源的存取，還有 host 核心的可獲得性。因此，要保證即時的服

務，協同排程的允許控策略需要搭配資源存取資料的反應時間和處理器排程器去分配 CSS 程序的反應時間。第三、分享資源的同步問題。資源的存取可以平行化處理。資源和 host 核心做好同步，可以允許每個資源達到最大平行化。 第四、有效的資源利用。即時排程的主要目標是達到高利用率和對於應用程式的 deadline 保證。因此，除了保證多資源存取的 deadline 之外，系統應該提供整個系統資源的高利用率。

2.6. 靜態式分工

一般的異質雙核心系統架構(比如由 Ferrari 等人提出的 The Janus system[14]，是由一個一般功能處理器和一個特殊功能處理器所組成。這兩個運算單元共同使用一個公用匯流排(bus)，而且可以自由地使用 RAM 和 ROM 等記憶體。而其他週邊輸出輸入設備則由一般功能處理器控制。通常這兩顆處理器是建構在單一晶片上，可以完全分享整個架構上的記憶體空間，也可以將處理器之間的溝通所需的成本忽略成極小。如此的設計通常會將一般功能處理器視爲 master 處理器，而特殊功能處理器便視爲 DSP。

然而這些系統大多是設計成靜態式分工的方法。Gai 等人曾討論由 GPP 和 DSP 非對稱架構多核心排程的問題[15]。在這篇論文中，DSP 被當成是類似有計算能力的資源，在 DSP 上執行的工作，都是由 GPP 一次一個分配過去。等到 DSP 完成工作，再回到 GPP 繼續下一個工作。如此設計是因爲 DSP 對某些工作的能力比 GPP 有效率很多，DSP 在這些工作上所省下的時間和單獨由 GPP 執行整個工作所花的時間相較，會大於 GPP 的閒置和兩個核心的溝通所需的時間。這種方式的實作方法是由兩個 task 駐列來完成。一個是 GPP 駐列，存放一般的工作，並由 GPP 負責執行。另一個則是 DSP 駐列，存放給 DSP 執行的工作。當 DSP 閒置時即是可以接受新工作，排程器選擇在這兩個駐列的頂端有最高 priority 的工作。若是選擇到 GPP 駐列，就由 GPP 來執行工作，反之 GPP 便將

DSP 駐列上的工作傳給 DSP 執行。而當 DSP 正在工作，排程器只選擇 GPP 駐列上最高 priority 的工作交由 GPP 執行。

由過去的研究顯示，非對稱式多核心平台的優點及可行性十分明顯，而且同一個工作如果能動態根據不同核心來排程，也會大大提昇效能。下一章，我們將提出精細分工的工作模型和相關背景。

3. 理論與實作背景

我們以動態精細分工工作模型爲概念，實作出非對稱式異質多核心平台排程器。所謂動態精細分工系統和目前廣爲使用的靜態式分工(statically partitioned)系統是相對的。在靜態式分工系統中，一項工作會分配到哪一個處理器是在系統設計時就決定好的。爲了提高整體系統的效能，我們提出了動態精細分工工作模式。假設在系統平台上有兩個處理器核心，分別是 GPP 核心以及 DSP 核心。新的工作被執行前，在 GPP 上的排程器將監看每個處理器核心的執行時期狀態，和決定哪一個核心較適合執行該工作，再動態地分配給 GPP 或 DSP 執行，減少處理器核心閒置的時間，提高處理器核心利用率，進而縮短全部工作執行時間，增加整個系統平台的效能，這種工作模式我們稱之爲精細分工工作模型。

本篇論文提出的排程器是實作於 OMAP5912 OSK 平台上，使用的作業系統在 ARM 處理器核心部分是以 eCos 2.0 版本爲基礎進行修改，在 DSP 處理器核心的排程核心是由我們自行設計的。在本章中，我們會介紹 OMAP 5912 應用處理器 (OMAP 5912 Application Processor)和 OMAP 59120 發展板(OMAP 5912 Starter Kit: OSK 5912)，以及嵌入式作業系統 eCos 2.0 版本。過去，本驗室也曾開發過在 Linux 下利用 DSP Gateway 及 TI 發展的 DSP/BIOS 排程器[21]。根據過去的實驗結果，利用 DSP Gateway 的溝通機制成本太高，每秒傳輸只有 3 MBytes，不合乎精細分工系統的需求，因此我們在本論文中改用較爲精簡的 eCos 作業系統，並提出有效率的 mailbox 和 shared memory 的溝通機制，以證實精細分工系統可以得較高的效能。所有 eCos 移植到 5912 OSK 的過程將在下一章說明。本論文研究實作的細節會在第五章詳細介紹。

3.1. OMAP 5912 Application Processor

OMAP5912 應用處理器是一塊高度整合的 SoC，包括的重要元件有:

GPP-ARM 核心、DSP 核心、和 Traffic controller 等等。OSK 5912 爲使用 OMAP 5912 應用處理器的發展平台。

OMAP5912 應用處器整合 ARM 926 EJ-S RISC 核心和 TI TMS320C55x DSP 核心。ARM9 RISC 核心在嵌入式系統被廣爲使用，C55x DSP 核心對於數位訊號處理展現高效能和低耗電的特性。因此 OMAP5912 應用處理器適合多媒體嵌入式裝置，經由切割每個應用程式爲眾多工作和適切地分配工作給兩個處理器核心執行可以有優秀的效能表現。

圖 1 爲 OMAP 5912 功能區塊圖[1]。MPU(ARM9)、MPU peripheral bridge、Memory traffic controller 以及 system DMA 四者透過 MPU BUS 溝通。MPU 由 MPU bridge 透過 public/ private peripheral bus 和其週邊溝通。DSP 透過 public/ private peripheral bus 和其 peripheral 溝通。此外 DSP 可藉由 DSP MMU 或是 MPU Interface 和系統其他部份做溝通。

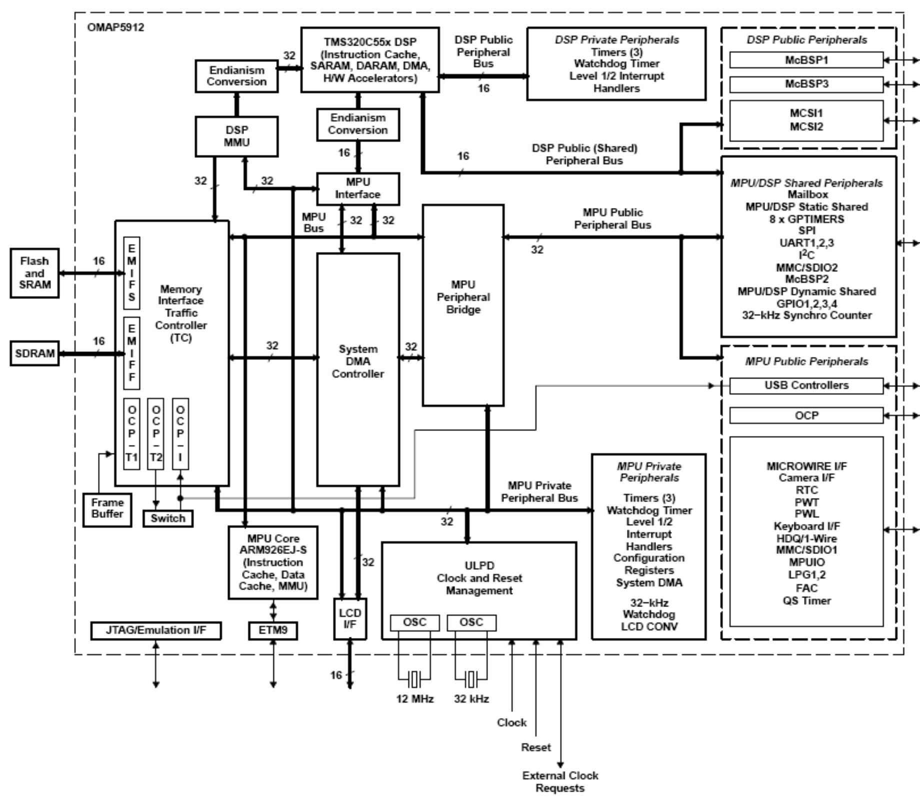


Figure 1. OMAP 5912 功能區塊圖

ARM 926 EJ-S　RISC 核心如下：

- Support for 32-Bit (Thumb Mode) Instruction Sets
- 16K-Byte Instruction cache
- 8K-byte data cache
- Data and program memory management unit (MMU)
- 17-word write buffer
- Two 64-Entry Translation Look-Aside buffers (TLBs) for MMUs

TMS320C55x 核心 technical feature 如下：

- One/two instructions executed per cycle
- Dual multipliers (two multiply-accumulates per cycle)
- Two arithmetic / logic units
- Five internal data/ operand buses (3 read buses and 2 write buses)
- 32k x 16-bit on-chip dual-access ram(DARAM, 64k bytes)
- 48k x 16-bit on-chip single-access ram (SARAM, 96k bytes)
- Instruction cache (24K bytes)
- Video hardware accelerators for DCT, iDCT, pixel interpolation, and motion estimation for video compression

表 1 列出 OMAP5912 應用處理器的記憶體配置[1]：

Table 1. OMAP5912 應用處理器記憶體配置

GBA BALL # Device Name	Start Address	End address	Signal Size	Data access Type
EMIFS				
CS0	0000 0000	03FF FFFF	64MB	
Boot ROM	0000 0000	0000 FFFF	64KB	32-bit Ex/R
Rserved boot ROM	0001 0000	0003 FFFF	192KB	33-bit Ex/R
Reserved	0004 0000	01FF FFFF		
NOR flash	0200 0000	03FF FFFF	32MB	8/16/32-bit Ex/R/W
CS1	0400 0000	07FF FFFF	64MB	
NOR flash	0400 0000	07FF FFFF	64MB	8/16/32-bit Ex/R/W
CS2	0800 0000	0BFF FFFF	64MB	
NOR flash	0800 0000	0BFF FFFF	64MB	8/16/32-bit Ex/R/W
CS3	0C00 0000	0FFF FFFF	64MB	
NOR flash	0C00 0000	0FFF FFFF	64MB	8/16/32-bit Ex/R/W
EMIFF				
SDRAM external	1000 0000	13FF FFFF	64MB	16-bit Ex/R/W
Reserved	1400 0000	1FFF FFFF		
L3 OCP T1				
Frame buffer	2000 0000	2003 E7FF	250KB	32-bit Ex/R/W
Reserved	2003 E800	2007 D7FF		
TI Camera I/F	2007 D800	2007 DFFF	2KB	32-bit Ex/R/W
L3 OCP T2				
Reserved	3000 0000	3000 D7FF		
TI Camera I/F	3007 D800	3007 DFFF	2KB	32-bit Ex/R/W
Reserved	3007 E000	7FFF FFFF		
DSP MPUI Interface				
MPUI memory + MPUI peripheral	E000 0000	E101 FFFF		
Reserved	E102 0000	EFFF FFFF		
TIPB Peripheral and Control Registers				
Reserved	F000 0000	FFFA FFFF		
OMAP5912 peripherals	FFFB 0000	FFFE FFFF		
Reserved	FFFF 0000	FFFF FFFF		

表 2 列出 OMAP5912 應用處理器 DSP 的記憶體對應關係(mapping)[1]，包括內部記憶體 DARAM 和 SARAM。ARM 定義一個 word 等於 4 個 byte，採 byte addressing，所有週邊和擴充的 memory 以及 control register 都由 32 位元來定位。DSP 定義一個 word 等於 2 個 byte，是採 word addressing。

Table 2. 記憶體對應關係

BYTE ADDRESS RANGE	WORD ADDRESS RANGE	INTERNAL MEMORY	EXTERNAL MEMORY†
0x00 0000 − 0x00 FFFF	0x00 0000 − 0x00 7FFF	DARAM 64K bytes	
0x01 0000 − 0x02 7FFF	0x00 8000 − 0x01 3FFF	SARAM 96K bytes	
0x02 8000 − 0x04 FFFF	0x01 4000 − 0x02 7FFF	Reserved	
0x05 0000 − 0xFF 7FFF	0x02 8000 − 0x7F BFFF		Managed by DSP MMU
0xFF 8000 − 0xFF FFFF	0x7F C000 − 0x7F FFFF	PDROM (MPNMC = 0)	Managed by DSP MMU (MPNMC =1)

0x050000-0xFF7FFFF(ARM: byte addressing)/ 0x028000-0x7FBFFF(DSP: word addressing)這塊區域，若 DSP 的 MMU 是 off：就把這塊空間直接對應到 MPU/DSP shared memory space（直接把 24bit line 對應即可），而此對應的區塊原來是保留區。若 DSP 的 MMU 是 on：這 24bit line 就當成是 virtual address，由 MPU 控制 DSP 之 MMU 來重新配置這塊空間，放到 MPU/DSP shared memory space 的某一段記憶體空間。

當 ARM 對應一塊實體記憶體到 DSP 的記憶體空間，DSP 可以透過 DSP MMU 來存取該塊記憶體，同時在 ARM 的虛擬記憶中有一塊配置為 DSP 記憶體空間，也會被對應到該塊實體記憶體。

在 OMAP5912 應用處理器上 Memory traffic controller 是一個很重要的內外部記憶體存取元件。Memory traffic controller 可以令 DSP 和 ARM 利用 TI OCP (Open Core Protocol)存取內部共用記憶體或週邊裝置，存取外部記憶體可利用兩種高速記憶介面來完成，分別為 External Memory Interface Fast(EMIFF)和 External Memory Interface Slow(EMIFS)。

EMIFF 相較 EMIFS 是較快速的記憶體裝置，在 OSK 5912 發展板上對應

EMIFF 配置的記憶體是 SDRAM，最大可支援到 64 M Bytes。存取資料的寬度和位址的寬度都是 16 bits，也提供了兩個 bank 選擇位元，亦即可以將 SDRAM 分成四個區域來使用。使用者的應用程式預設是諸存到此 SDRAM。

EMIFS 所連接的外部裝置記憶是 NOR FLASH。透過介面可以 8 bits / 16 bits / 32bits 的寬度在每個 NOR FLASH 晶片上存取資料，其使用的位址寬度爲 25 bits。OSK 5912 發展板上共有四塊外部 FLASH 晶片，每塊晶片最大容量爲 64 M Bytes，所以可使用的總記憶體容量爲 128 M Bytes。此四塊 NOR FLASH 分別爲 CS0, CS1, CS2, 和 CS3。Boot ROM 位於 CS0，系統開發者設計的 boot-loader 或作業系統則是存放在 CS3。經過設定，啓動的模式可以利用 CS0 的 boot ROM 或是 CS3 的 boot-loader 開機。

3.2. OMAP 5912 Starter Kit：OSK 5912 (OMAP 5912 OSK)

OSK 5912 是對軟體和硬體做高度整合的平台，主要可做爲視訊 圖片訊號處理裝置和行動溝通裝置。可以使用一般的嵌入式作業系統做爲 OSK 5912 上 ARM 處理器的作業系統，而 TI 提供 DSP/BIOS 做爲 DSP 處理器的即時核心(real-time kernel)。Figure 2 爲 OSK 5912 正視圖[19]：

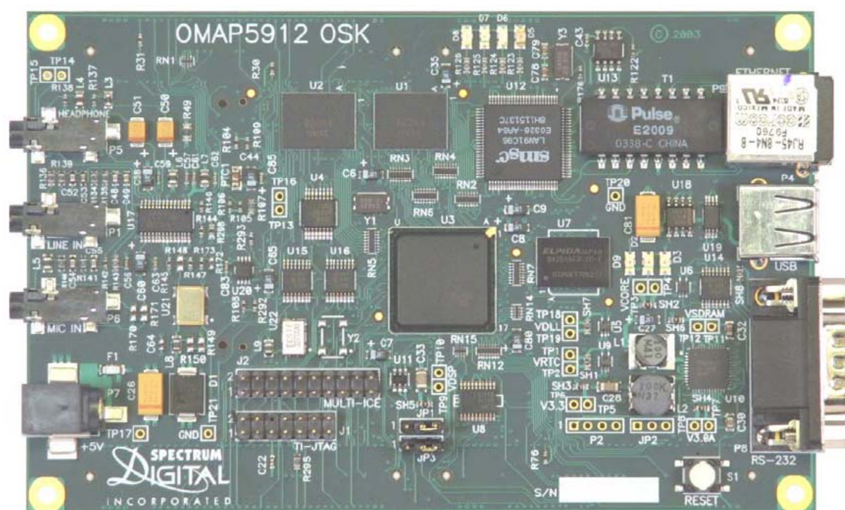


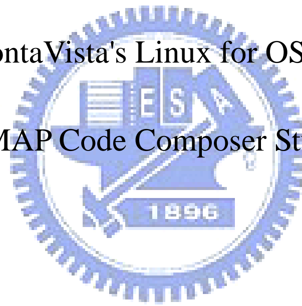
Figure 2. OSK 5912 正視圖

Hardware Features 如下：

- ARM 926EJS 處理器核心運行於頻率 192 MHz。
- Texas Instruments TMS320C55x 運行於頻率 192 MHz。
- 內建音訊編碼解碼器 TLV320AIC23 codec
- 64 Mega Bytes DDR RAM
- 256 Mega Bytes on board Flash ROM
- 10 MBPS Ethernet port
- On board IEEE 1149.1 JTAG connector for optional emulation

Software Features 如下：

- Compatible with MontaVista's Linux for OSK5912
- Compatible with OMAP Code Composer Studio from Texas Instruments



3.3. eCos

OSK 5912 所採用的原始作業系統是 MontaVista Linux，但是根據我們去年的經驗，Linux 配合 DSP Gateway 的效能表現無法達到精細分工系統所需的要求，故本論文沒有採用原始發展系統所採用的整合軟體。接下來介紹 ARM 端採用的 eCos 作業系統。在下一章我們會討論如何將 eCos 移植到 OSK5912 的平台下。

3.3.1. eCos Overview

eCos 是一個開放程式碼，可設定(configurable)，可移植和免費的嵌入式即時作業系統。eCos 的一項重大的技術革新是設定系統(configuration system)。設定系統允許應用程式設計者對 run time 元件加入或調整所需的功能和實作方式。傳統上，作業系統會限制實作的方法，無法選擇。設定系統使得 eCos 開發者創造符合特定應用程式的特定作業系統，也使得 eCos 適合更大範圍的嵌入式應用。

設定系統的使用可以保證資源的最小化，和其他不需要的功能和特徵都可以被移除。如此便利性的因素是 eCos 它是一個元件架構的系統。eCos 被設計爲可以移植到許多目標架構和目標平台，包括 16 32 64 位元架構和 MPU, MCU, DSP。eCos 支援許多不同平台架構，如 ARM、Intel StrongARM 及 XScale、Fujitsu FR-V、Hitachi SH2/3/4、Hitachi H8/300H、Intel x86、MIPS、Matsushita AM3x、Motorola PowerPC、Motorola 68k/Coldfire、NEC V850 和 Sun SPARC，其他尙包括許多流行的架構和發展板。

3.3.2. Configure Tool

嵌入式系統正被推動朝著更小更快更便宜更精緻，所以更需方便地控制系統內所有的軟體。有不同的方法可以控制應用程式內元件的特性。eCos 元件控制的哲學是爲了減少系統大小，對資源最自由的配置。持著此設計哲學，最小化的系統不必支援某些複雜系統上才有的強大功能。有一種在 run time 控制軟體元件的方法，例如動態連結程式庫(Dynamic Link Libraries)，不必預先對元計做設定，但是這個方法會會導致程式大小增加。另一種方法是在 link time 時，當需要某個特殊功能元件就會被包入，反之則除去，例如 GNU linker。這方法的特性是擁有某元件的全部功能或都不擁有。Compile time 的元件控制，使得系統開發者可以建立特定應用程式需要的元件，可以保持所有的程式碼都是系統所需要的。對於嵌入式系統來說，這是解決程式碼多寡的好方法。eCos 有一套十分方便的 configure tool，讓系統開發者在 compile time 決定所需的元件和元件的能力，而不必動手修改元件的程式本體。

3.3.3. Component

要了解 eCos，則了解元件的基礎架構非常重要。元件基礎架構專爲滿足嵌入式系統和嵌入式設計的相關需求而存在。設計的 eCos 元件基礎架構可以控制元件達到最小記憶體使用、允許使用者控制時間行爲以符合即時性、和使用一般

的程式語言，如 HAL 的實作就包括 C、C++、和組語。大部份嵌入式系統本身所支援的功能比特定應用程式需要的功能更多。通常系統內多餘的程式碼支援的功能，卻是嵌入式程式開發者所不關心也不需要的，而且更多的程式碼使產生錯誤的機會更大。舉 Hello world 程式爲例。很多即時作業系統支援了 mutexes, task switch，卻是在此簡單程式所不需要的。eCos 給開發者最終 run time 元件的控制權，沒必要的功能可以輕易地被移除。eCos 系統的大小可由幾百 Byte 到幾百 K-Byte 彈性地增減。

即時嵌入式系統的標準功能包括包括插斷處理，例外處理，錯誤處理，執行緒同步，排程，計時器，和裝置驅動程式。這些標準的功能在 eCos 中由各個元件負責，稱之爲標準元件。這些標準元件由即時 kernel 爲核心組成。列出元件如下：

- Hardware Abstraction Layer (HAL)—硬體抽象層隱藏每一個支援的 CPU 和平台的特別特徵，以至於 kernel 和其他 run time 元件都是在一個容易移植的形式。
- Kernel—Kernel，包括記憶體、快取、插斷處理、例外處理，執行緒、同步機制，排程器，計時器，計數器和鬧鐘。
- ISO C and math libraries—常用標準程式庫。
- u-ITRON and POSIC —μ ITRON and POSIX 應用程式者介面
- Device drivers—裝置驅動程式，支援廣泛的裝置標準序列埠控制器，Ethernet 網路控制器，PCMCIA 控制器，USB 控制器，PCI 控制器，和 Flash Rom 等等。
- RebBoot ROM monitor — RedBoot ROM monitor 是一個 Bootstrap 應用程式，爲了方便移植它使用 eCos HAL，而且他可以透過序例埠和網路兩種 booting 方式。

- GNU debugger (GDB) support—GNU 除錯器，提供目標平台軟體可以和 GDB host 溝通，對應用程式除錯。
- 其它軟體元件 — SNMP,HTTP,TFTP 和 FTP。

eCos kernel 或是應用程式都是在 supervisor 模式下執行，所以在 eCos 系統中，user 模式和 kernel 模式並沒有區分。

圖 3 為 eCos 系統區塊圖[18]：

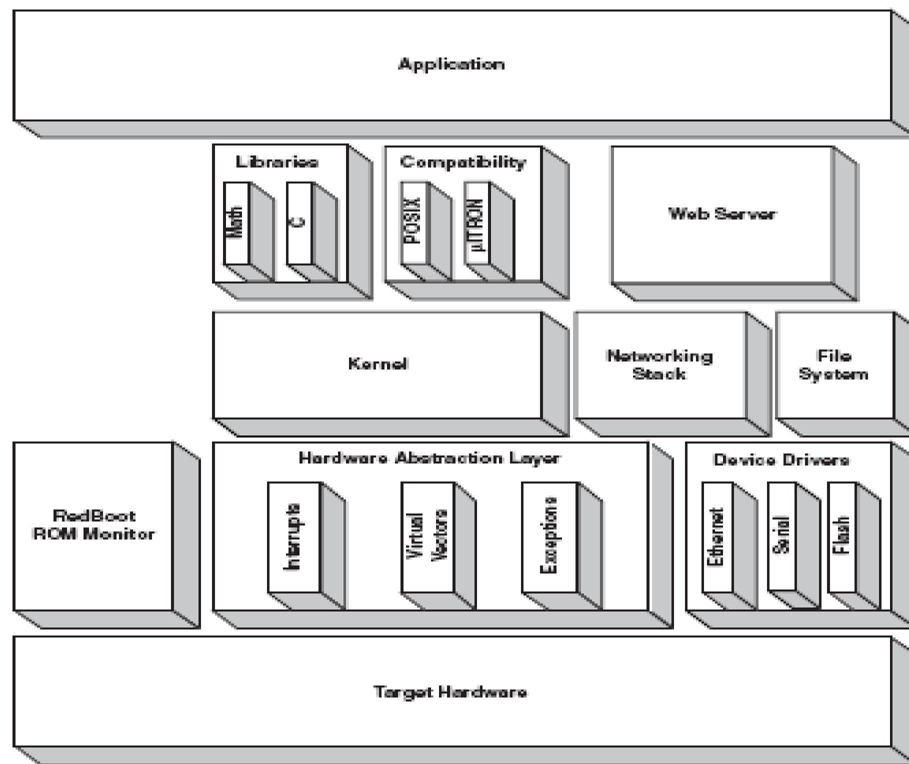


Figure 3. eCos 系統區塊圖

下面段落將介紹三個重要的 eCos 元件，分別是 HAL、kernel、及 RedBoot。

3.3.4. HAL

HAL 將處理器架構的底層硬體和平台的底層硬體抽象化可以有效率地移植 eCos kernel 到別的平台。為了移植 eCos 到新的平台，主要的工作是修改 HAL 以適應支援新的硬體平台，因此了解 HAL 這個軟體元件的架構相當重要。HAL 有一般化的 API，把特殊的硬體行為封裝起來，由硬體來完成設計的功能，並且

允許應用層直接存取硬體和任何架構特徵。例如有同樣意義的 interrupt，實際執行插斷的程序依不同架構而相異。為了使 HAL 的可用性達到最廣的範圍，HAL 是用 C 語言和組合語言實作。另外為了 HAL 介面實作上的效率，HAL 是用 C/ CPP 巨集實作，可以使用 inline C 語言，inline 組合語言，或外部呼叫 C 語言和外部呼叫組合語言。

HAL 含概三個不同的模組：

- Architecture
- Variant
- Platform

第一個 HAL 模組定義 architecture。每個被 eCos 支援的處理器家族都被認定為不同的 architecture。每個 architecture 模組會包含所需的 CPU 啟動程式碼，interrupt delivery 程式碼，context switching 程式碼和其他指令集架構特殊功能的程式碼。第二個 HAL 模組定義了 variant。一個 variant 是一個處理器家族當中的一個特定處理器。例如定義不同的 on-chip MMU 或是快取，也處理任何晶片上的週邊，如記 memory controller 和 interrupt controller。第三個 HAL 副模組是定義 platform。一個 platform 是一塊特別的硬體平台，它包括選擇處理器的 architecture 和 variant。傳統上這個模組包括平台啟動，晶片選擇設定，interrupt controller 和計時裝置。這三層的介定不必很清楚，因為各模組的功能性在不同硬體平台有不同的設定。例如快取和 MMU 可以在 architecture HAL 或 variant HAL。

圖 4 是 HAL 啟動期間副程式被引用的流程圖[18]。啟動程序可能會依不同架構和平台的使用有些微的不同。此外，啟動程序可能也因為 HAL 的一些設定選項的不同而與此流程圖有所差異。

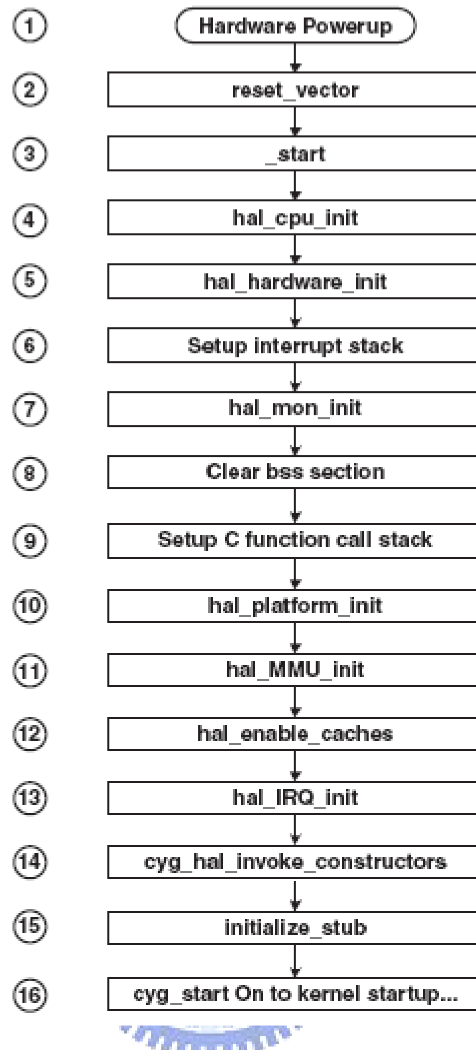


Figure 4. HAL 啓動流程圖

1. 系統在電力週期啓動之後開始運作。此啓動程序也是 soft reset 的啓動方式。

2. 在 hard reset 或 soft reset 發生後，處理器會跳到 reset vector。Reset vector 對處理器會用到的最少數量暫存器做設定，使得系統可以繼續初始化程序。

3. 接著 reset vector 會跳到 _start，是 HAL 初始化的主要開啓點。

4. 呼叫 hal_cpu_init，這個程式設定處理器的暫存器，如關閉指令和資料快取。確保對於剩下的初使化程序而言，處理器在一個正常的狀態。

5. 下一個被呼叫的副程式叫 hal_hardware_init，硬體設定包含快取設定，

設定插斷暫存器爲預設狀態，關閉處理器的 watchdog，設定即時時鐘暫存器，和設定晶片選擇暫存器，這些都是基於平台特有的硬體而不同。

6. 接著是設定 interrupt stack 的區域，在 interrupt 發生時可以儲存處理器狀態。在整個初始化程序中，都是利用這一段 stack 做爲呼叫 C 副程式會用到的 stack。因爲此時 interrupt 是關閉的，不會有衝突發生。
7. 下一步執行 hal_mon_init 程式，確保預設的 exception 處理器安裝給每一個處理器支援的例外情況。
8. 接著清理 BSS 部份，它包含所有未初始化的區域和全域變數。
9. 然後設定 stack，以致於 C 程式呼叫可以被實行，不再使用 interrupt stack。
10. 呼叫 hal_platform_init 或呼叫 hal_if_init，初始 virtual vector table。
11. 初始化 MMU，處理 logical addresses 和 physical addresses 的轉換，同時提供保護和快取機制。
12. 接著啓動指令快取和資料快取。
13. 執行 hal_IRQ_Init，設定 Communications Processor Module (CPM)，它接受和按優先順序處理內外部 interrupt。
14. 下一步，cyg_hal_invoke_constructors 呼叫所有 global C++ constructors。Linker 會提供 global constructor 的名單，cyg_type.h 則用巨集定義這份名單上 constructor 被呼叫的順序。
15. 如果設定當中有除錯環境而且 ROM monitor 沒有提供除錯支援，下一個要呼叫的是 initialize_stub。initialize_stub 安裝 standard trap handler 並把硬體設定在適合除錯的狀態。
16. 最後一個步驟是把控制權轉給 kernel。cyg_start 就是控制權由 HAL 轉入 kernel 的點。

3.3.5. The Kernel

Kernel 是 eCos 系統的中樞。Kernel 提供即時作業中的標準功能例如插斷和例外處理、排程、執行緒和同步。在 eCos 系統下可以對這些組成 kernel 的標準功能元件完全地設定，以達到特殊的需要。eCos kernel 以 C++實作，允許用 C++寫成的應用程式直接透過 C kernel API 介面和 kernel 溝通。eCos kernel 也支援標準 u-ITRON 和 POSIX compatibility layers 介面。爲了符合即時性需求，eCos kernel 依循下列準則發展：

- Interrupt latency－interrupt 回應和開始執行 ISR 的時間要少而且 deterministic。
- Dispatch latency—執行緒準備完成可以執行的狀態到開始執行的時間要少並且 deterministic。
- Memory footprint—對一個設定完成的系統，程式或資料所需求的記憶體資源要保持最小化和 deterministic。而且要確保嵌入式系統的動態記憶體配置不會使用超出所有記憶體的量。
- Deterministic kernel primitives—所有 kernel 的行爲都要是可預期的且符合即時性的需求。

eCos kernel API 不回傳錯誤訊息。在嵌入式系統當中，處理錯誤回傳訊息會導致許多問題，如消耗貴重的執行週期和程式空間來檢查回傳的訊息。爲了程式發展的便利性 eCos kernel 提供 assertion，它可以被 eCos package 開啓或關閉。傳統上，assertion 在除錯階段會開啓，允許 kernel 程式顯示錯誤檢查。如果錯誤產生了，就會回傳一個 assertion 失敗並且會中止應用程式。除錯程序完畢之後，assertion 就在 kernel package 之中關閉。此方法有很多好處，如限制程式中錯誤檢查的 overhead，消除應用程式錯誤檢查的需要；如果有一個錯誤發生，應用就被暫停，可以立即知道發生錯誤的地點，而不是依賴回傳訊息再去檢查。

kernel 提供開發多執行緒應用所需的重要功能。

在硬體都啓動完成之後，HAL 中呼叫 Kernel 啓動的程序。cyg_start 是 kernel 啓重程序的啓始點。它呼叫其他預設的啓動程式來處理不同初始化的任務。只要在應用程式之中提供相同程式名稱，預設 kernel 啓動程式可以被簡單的替換，以完成使用者特殊的初始化工作。Kernel 啓動程序如圖 5 [18]。

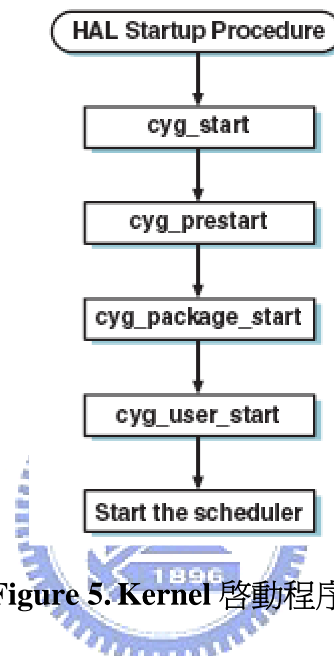


Figure 5. Kernel 啓動程序

Core kernel 啓動程式，cyg_star。Cyg_start 之後下一個被呼叫的程式是 cyg_prestart。它預設不完成任何初始任務，讓使用者自行決定在其他系統初始化之前該被完成的初始化都可以在這裡執行。接著呼叫 cyg_package_start，在應用程式開始之前初始化將用到的 package，如 u-ITRON 和 ISO C 程式庫。之後呼叫 cyg_user_start，它是應用程式的進入點。建議 cyg_user_start 被使用來完成任何應用指定的初始化、產生執行緒、產生 synchronization primitives、設定鬧鐘、和註冊任何需要的 interrupt handlers。當 cyg_user_start return 時會自動執行呼叫排程器。

3.3.5.1. The Scheduler

eCos kernel 的核心是排程器。排程器的工作是去選擇最適合的執行緒執行，提供執行中執行緒同步的機制和控制 interrupt 在執行緒執行的影響。在排程器程

式程式碼執行期間，不會關閉 interrupt，因爲 interrupt latency 十分短。排程器內存在一個計數器，它決定排程器是自由地執行或是關閉。如果計數器非零，排程器就被關閉，當計數器回到零，排程就會啓動。在 ISR 執行期間，HAL 預設 interrupt handler 會修改計數器來停止排程動作。執行緒也有能力開關排程器。

3.3.5.2. Multilevel Queue Scheduler

MLQ 排程器允許同一個 priority 有多個執行緒可執行。priority 可由 1 設定到 32，對應到數字是 0(最高)到 31(最低)。MLQ 排程器允許不同 priority 可有 preemption。Preemption 是指低 priority 執行緒的被 context switch 暫停執行，因此高 priority 的執行緒開始執行。在同一個 priority 中，MLQ 排程器有 time-slicing 功能。每一個執行緒在一段特定的時間內執行稱之爲 time-slicing，統系開發者可以設定 time-slicing 的長度。MLQ 排程器的駐列實作上是用 double linked circular list 來連結同一個層級的不同執行緒和不同層級的執行緒。圖 6 看到 MLQ 排程器的功能[18]。

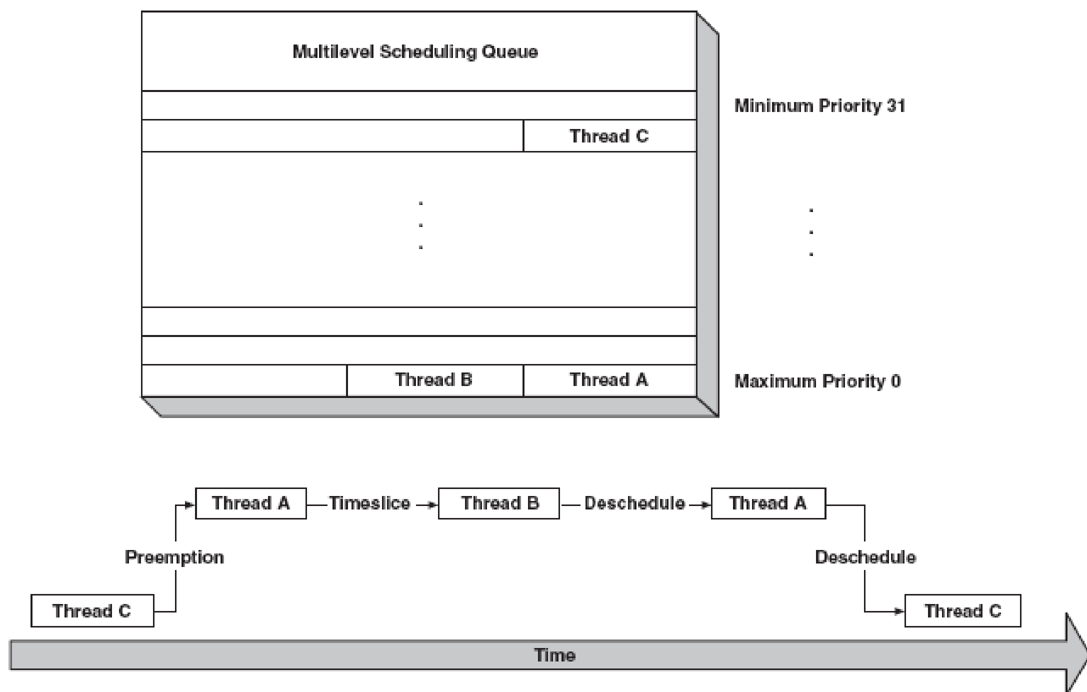


Figure 6. MLQ 排程器運做圖

三個執行緒，執行緒 ABC，priority 分別爲 0, 0, 30。當 C 開始執行之後，A 準備完畢可以執行，於是 C 發生 context switch，A 開始執行。接著 B 也準備完成，在 A 持續到它分配的時間用完，A 發生 context switch，讓 B 開始執行。AB 輪流執行完畢，才再由 C 繼續執行。

3.3.5.3. Bitmap Scheduler

Bitmap 排程器的執行緒也是有多個 priorities；然而每層級只能有一個執行緒。這種設計簡化排程演算法，也令 bitmap 排程器非常有效率。Priority 的數量和設定和 MLQ 相同；駐列的實作有三種，可以是 8,16 或 32 位元值，依設定的 priority 而定。因此駐列中一個位元代表一個 priority。Bitmap 排程器可以有 preemption，但是因爲每個 priority 只有一個執行緒，故 time-slicing 會失去功能。使用 bitmap 排程器時就會關閉 time-slicing。圖 7 是 bitmap 排程器的例子[18]：

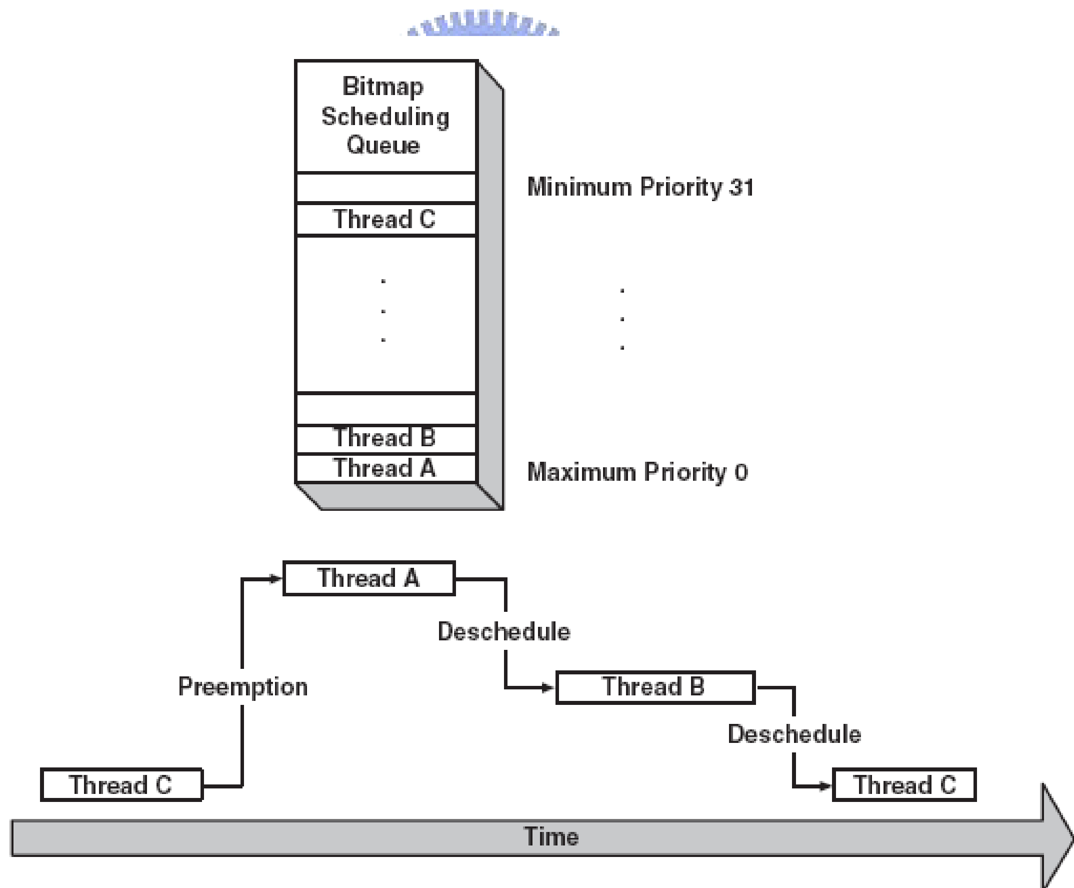


Figure 7. Bitmap 排程圖運做圖

有三個不同 priority 的執行緒 ABC，其 priority 分別爲 0、1、30。開始於 C

執行。接著 A 和 B 都可以執行，使得 context switch 發生，C 被置換出去。A 有最高的 priority，所以 A 先執行，完成之後，發生 context switch，B 開始執行。在 B 完成之後，C 才能繼續它的程序。

比較這兩種排程器，bitmap 排程器是較簡單的排程策略。但是，MLQ 排程器可提供更多的選擇給執行緒操作，也可容納更多的執行緒，只要記憶體夠，就沒有執行緒數量的限制。系統開發者可依應用程式的特殊需求而決定使用哪一種排程器。

3.3.5.4. Synchronization Mechanisms

eCos kernel 提供系統中執行緒溝通機制和執行緒對分享資源存取的同步機制。提供的機制有 Mutexes、Semaphores、Condition variables、Flags、Message boxes 和 Spinlock (for SMP systems)。

Kernel 提供同步 API，應用程式可以便利地使用這些同步機制。API 程式提供有 blocking 功能或 non-blocking 功能。Blocking 程式呼叫，如 cyg_semaphore_wait，暫停執行緒的執行，直到 API 程式可以成功地完成。Non-blocking 程式呼叫有兩種，第一種如 cyg_semaphore_trywait，不論程式有沒有成功地完成，會回傳訊息指出呼叫的狀態，所以執行可以繼續執行。第二種 blocking 呼叫，必需先設定等待的時間，用時間長度爲暫停長短的依據，如 cyg_semaphore_timed_wait。

Mutex 的目標和其他的都不同。Mutex 允許多個執行緒安全地存取共享的資源。它只有兩種狀態: locked 和 unlocked。並且 Mutex 有擁有者的概念，只有擁有者(上鎖者)可以解開 mutex。其他同步機制都是被使用來做爲執行緒之間互相溝通或從 DSR 連接到相關的 interrupt handler 到一個執行緒。

Semaphore 是一種用計數來指示資源已上鎖或可以獲得的同步機制。Semaphore 有兩種型態，分別是 counting semaphore 和 binary semaphore。Binary semaphore 很像 counting semaphore，但是它的計數決不會大於一，所以它的狀態

只有 locked 和 unlocked；和 mutex 不同的是，它沒有擁有者的概念。意即每一個執行緒都可以對 semaphore 上鎖和解鎖。Counting semaphore 依照計數值可以有多種狀態。當執行緒 post semaphore 時增加的數值，同時也是當一個執行緒 wait 一個 semaphore 減少的數值。當 semaphore 回到零時，只有最高 priority 的執行緒可以執行。

Condition variables 和 mutex 搭配使用允許多執行緒存取共享資源。傳統上，一個執行緒產生資料，一個或多個執行緒等待這筆資料。當資料備妥時，產生資料的執行可以發出 Condition variable 通知喚醒一個或喚醒全部執行緒。等待中的執行緒就可以拿到需要的資料並處理之。

Flag 用 32 位元表現的同步機制。每一個位元代表一個狀態，允許執行緒去等待一個或多個組成的狀態。當狀態符合，該執行緒就會被喚醒。

Message box 又叫 mail box，提供兩個執行緒交換資訊。因爲 mailbox 的容積有限，交換的資訊一般都是資料結構的指標，或是有特別設計過的訊息。

在對稱式多核心平台上，eCos kernel 提供另一種額外的同步機制。同時其他的同步機制可以在對稱式多核心平台上運作。Spinlock 基本上是一個 flag，和其它同步機制比較起來是更底層的操作，會依不同的硬體有不同的實作方式。有些處理器提供 test-and-set 指令來實作 spinlock。在處理器執行一段特殊程式碼之前要去檢查此 flag。如果 spinlock 沒有上鎖，處理器可以設定 flag 和繼續執行此執行緒。如果 spinlock 被鎖上，執行緒就會持續地檢查 flag 直到它被解鎖，而沒有被 suspend。

3.3.5.5. Threads and Interrupt Handling

Kernel 利用一種 two-level 方法處理 interrupt。連結每個 interrupt vector 是一個 Interrupt Service Routine (ISR)，它會盡可能快速的執行來回應硬體 interrupt。ISR 只可以做少數的 kernel 呼叫，都是和 interrupt 系統相關，但不能做喚醒執行緒的動作。如果 ISR 偵測到 I/O 完成，一個執行應該被喚醒，ISR 就會讓連結的

DSR 去執行。DSR 可以使用更多 kernel 呼叫，如發 condition variable 訊號或 post a semaphore。關閉 interrupt 可以阻止 ISR 執行，但系統中很少機會關閉 interrupt，如果有也是很短一段時間。執行緒關閉 interrupt 的主要理由是改變一些 ISR 之間共享的狀態設定。例如，如果一個執行緒會增加 linked list 的 node，而且 ISR 可能在任何時候自 linked list 移除一個 node，此執行緒就要在操作 list 時關掉 interrupt。

3.3.6. RedBoot

RedBoot 是 Red Hat Embedded Debug and Bootstrap 的縮寫。它是一個爲嵌入式系統提供一個除錯和 bootstrap 環境而設計的程式。RedBoot 是一個以 eCos 爲基礎的應用程式，並且使用 eCos HAL 爲它的基礎。下例爲 RedBoot 的特性:

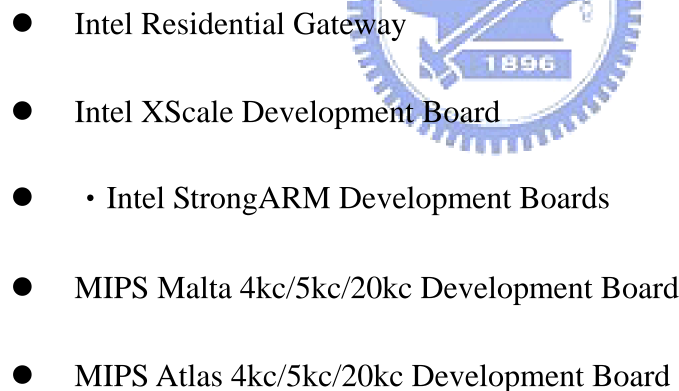
- 支援 boot scripting
- Command line 介面，用於監視和控制
- 可經序列埠或網路存取
- 支援 GDB
- 支援 Flash image 系統
- 支援 X/ Y 模式傳輸
- 支援網路 bootstrap，利用 BOOTP 或靜態 IP 位址設定

RedBoot 包括 GDB stub，可以從 PC 上的 GDB host 連接到平台上除錯。連接方式可以是透過 serial port 或是 Ethernet port。除了除錯外，RedBoot 主要的功能是 booting，支援三種 booting 方式，分別稱爲 ROM, RAM, 和 ROMRAM。其含意爲 RedBoot 的 binary image 放置於何種記憶和由何種記憶體執行。ROM 和 RAM 即代表 image 放置和執行都在 ROM 或 RAM，ROMRAM 指的是放置在 ROM，欲執行前拷貝到 RAM，才在 RAM 執行。在 RAM 資源有限的情況下，

經常使用 ROM 模式的 booting。Image 是被放置在 flash 或 EPROM。由於 flash 命令不能寫入 RedBoot image 本身存放的區塊，所以爲了更新 RedBoot，必需使用 RAM 模式來達成。可以使用 ROM 和 ROMRAM 模式直接 booting，除非配合其他 ROM monitor，否則不能直接使用 RAM 模式 booting。

RedBoot 會在 memory map 的底部，保留 RAM 空間結 run time 的資料和 CPU exception/ interrupt tables。在 memory map 頂空的空間則保留給 net stack、zlib 解壓空間、等等平台的特殊需求。

依設定選項和 package，可以建立最經濟的 RedBoot，只包含該平台所需的最小數元件。圖 8 顯示出 RedBoot ROM monitor 部份特徵的區塊圖[16]。由此圖，可以看到 RedBoot 提供的功能全包含在 RedBoot 程式 image 之中。RedBoot 可以只提供簡單的 image 載入和執行功能，提供簡易 flash file system，或是透過命令列對應用程式做監控和除錯功能。在市面上有很多產品使用 RedBoot：



- Intel Residential Gateway
- Intel XScale Development Board
- ・Intel StrongARM Development Boards
- MIPS Malta 4kc/5kc/20kc Development Board
- MIPS Atlas 4kc/5kc/20kc Development Board

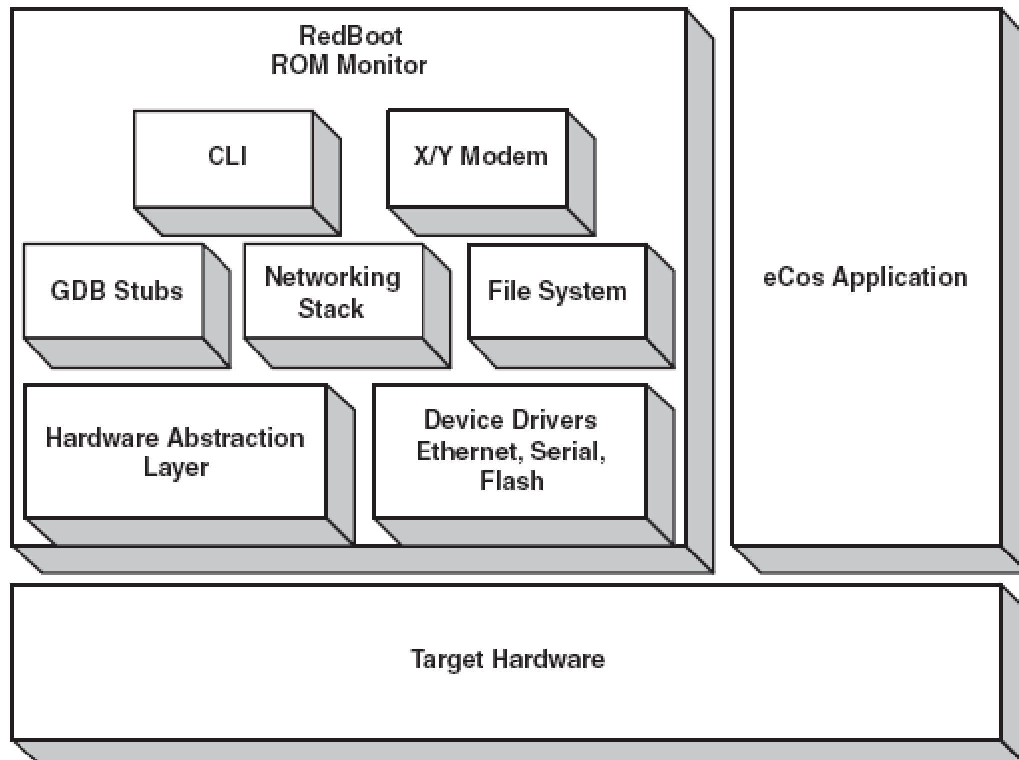


Figure 8. RedBoot ROM monitor 特徵區塊圖

由於 eCos 目前尚未被其它的單位移植到 5912 OSK，在下一章我們會介紹如何將它移植到我們的平台上的細節。

4. eCos 至 OMAP 5912 OSK 的移植方法

因為本論文所設計的動態分工異質雙核心排程器是架構在 eCos 系統之下，所以實作之前必須先把 eCos 移植到目標平台上。目前 eCos 的公開程式碼計畫並沒有支援 OMAP 5912 OSK，所以本章先描述如何將 eCos 移植到目標平台上。移植的流程分為三個階段，分別是移植 architecture HAL、variant HAL、和 platform HAL。每個 HAL 的功能及含義已在上一章敘述。這裡大略介紹移植流程，之後段落將每個細節再詳加說明。

Architecture HAL 的移植：這部份的程式是根據處理器核心的大體架構而設計的。因為 5912 OSK 的核心處理器是 ARM，由於 ARM 的 architecture HAL 已經存在現有的 eCos 中，所以我們可以直接延用。

Variant HAL 的移植：Variant HAL 主要是處理每一類嵌入式處理器核心在架構上的小調整。比如同樣是 ARM 9 的處理核心，有些有 MMU，有些沒有，有些有訊號處理指令集，有些則支援 Java 加速。由於目前在 eCos 下還沒有 ARM 926EJS 核心的移植，所以我們以 ARM 925T 的程式碼為基礎開始進行移植工作。首先取得需要的硬體資料（詳見下一節），接著修改各部份的內容設定，例如 instruction cache 和 data cache。還要在 eCos.db 檔之中新增 ARM926EJS package。

Platform HAL 的移植：由於 OMAP 5912 OSK 是承續 OMAP Innovator 的設計，因此可以用 OMAP Innovator 為基礎來移植。但仍舊必需取得所需之硬體資料。5912 OSK 的 Ethernet 是 LAN91C96，與 Innovator 相同，故可以使用，但必需新增 Ethernet_5912 package 於 eCos.db 之中，並指出其使用 LAN91C96，此外還得給定其 base address。Flash 是兩顆 Micron Q Flash MT 28F128J3，並沒有現存的 package，但是它和 Intel Strata Flash 28F128J3 相容，可以借用其 package，之後我們必需新增 Flash_5912 package 於 eCos.db 之中，並指定使用 Intel Strata 28F128J3。Serial port 使用 UART， Innovator package 中沒有檔案可以直接使用，

但我們可以用相近的Integrator做爲參考，但需做些許修改，當然也要加入eCos.db的package。Platform start up包括很多的項目，如 internal memory, external memory, MMU, register 等等的設定，由於平台的改變，因此能延用innovator的部份有限，在這邊，我們做了很大的修改。

上面三個 HAL 的設定若有衝突，將以 platform HAL， variant HAL，architecture HAL的優先順序來決定那一個設定是有效的。例如architecture HAL可能設定了instruction cache的大小，variant HAL也可以設定之，此時就以 variant HAL的設定爲準。

4.1. 搜集硬體資料

硬體資料的搜集十分重要，必需了解有哪些硬體，其型號和數量。下面是porting時需要的資料，擷取自 OMAP 5912 OSK[1] 和OSK 5912 Hard SPEC [19]。

- ARM926EJS
  - ✧ Support for 32-bit and 16-bit instruction set
  - ✧ L1 16K-byte, four-way, set-associative instruction cache
  - ✧ L1 8K-byte, four-way, set-associative data cache with 17-word write buffer
- Serial port
  - ✧ TI MAX3221 RS232 介面
  - ✧ Base address 0xFFFB 0000
  - ✧ 使用 UART 1 (UART 16550 晶片)
  - ✧ 115200 bps，8 data bits, No parity bit, 1 stop bit, 和 no flow control。
- Ethernet port

- ✧ LAN91C96 Controller
- ✧ Address 0x 0480 0000 使用 CS1 memory space

- Memory traffic controller
  - ✧ External memory interface slow (EMIFS);兩顆 128MB Micron MT Q-Flash， 16-bit data access， 4 chip-select，每個 chip-select 有 25-bit address bus，64MB。.
  - ✧ External memory interface fast (EMIFF): 一顆 64MB 的 SDRAM。16-bit data access，16-bit address，包含 2 個 bank selection bit。

4.2. eCos configtool 使用方法

要使用 eCos configtool 必需先設定好 eCos.db。它是一個 data base，用來記錄 build 時會用到的 package 的相關內容，例如名稱、說明、程式路徑等。位於 C:\cygwin\opt\ecos\ecos-2.0\packages

下面列出 build 出 5912Redboot 所新增 package: 參考資料為 eCos component writer's guide [15]。

```
package CYGPKG_HAL_ARM_ARM9_OMAP5912 {
    alias  { "OMAP (ARM)" hal_arm_arm9_omap5912 }
    directory     hal/arm/arm9/omap5912
    script hal_arm_arm9_omap5912.cdl
    hardware
    description "This HAL platform package provides generic support for the OMAP5912 platform." }


package CYGPKG_HAL_OSK5912 {
    alias  { "TI OSK5912 board" hal_OSK5912 }
```

directory hal/arm/arm9/OSK5912

script hal_OSK5912.cdl

hardware

description "The DTB HAL package provides the support needed to run on the ARM processor of the DTC Digital Transceiver Board." }

上二者爲板子對 HAL 的設定，可以合併，以後者爲說明例子。名稱爲 CYGPKG_HAL_OSK5912，CYGPKG 爲 CYG 之中的命名規則 CYG Package 簡稱爲 CYGPKG。CYGHWR、CYGNUM 和 CYGINT 則分別代表平台特有硬體、數值選項和介面。顯示名稱 alias 爲 TI OSK5912 board。script 檔案路徑爲 hal/arm/arm9/OSK5912，而 script 檔案名稱爲 hal_OSK5912.cdl，script 檔案紀錄這個 package 的 compile 資訊。Hardware 指出此 package 是述描硬體。Description 爲說明文。

在此簡介何爲 script 檔案? 每一個 package 都有 script 檔案，其功能是向 component framework 描述這個 package，可以有多個 script 檔案描述同一個 package。

描述的內容有下列四項: 首先是使用 package 的 dependences，其次是 configuration options, 第三是 associated value range，是後是如何去 build 這個 package。每一個 package 可以包含下例四種項目分別是 cdl_package, cdl_component, cdl_option, 和 cdl_interface。前三項大略是階層的關係，cdl_package 是描述的主體，其下可以有多個平行的 cdl_component 及 cdl_option。而 cdl_component 之下也可以有多個 cdl_option。Cdl_interface 則是定義這個 pacakge 的介面。使用的預設值含義如表 3:

Table 3. script file 預設值

no_define	不將 define 加入.h 檔案之中
define	將 define 加入.h 檔案之中
require <expression>	<expression>爲 true 才正確

user_value	數值會依數據的 reference 而改變
inferred_value	數值由 cdl 之中寫定不可更改

package CYGPKG_IO_SERIAL_OMAP5912 {

alias { "TI OMAP5912 serial device drivers " devs_serial_omap5912 omap5912_serial_driver omap5912_serial }

hardware

directory devs/serial/arm/omap5912

script ser_omap5912.cdl

description "TI OMAP5912 serial drivers." }

上爲 serial port package 的描述。

package CYGPKG_DEVS_ETH_ARM_OSK5912 {

alias { "OMAP OSK5912 onboard ethernet support" devs_eth_arm_OSK5912 }

hardware

directory devs/eth/arm/OSK5912

script OSK5912_eth_drivers.cdl

description "This package contains hardware support for onboard SMC91C96 ethernet device on the OSK5912 board." }

上爲 Ethernet package 的描述。

package CYGPKG_DEVS_FLASH_OSK5912 {

alias { "FLASH memory support for the TI OSK5912 Board" flash_OSK5912 }

directory devs/flash/OSK5912

script flash_OSK5912.cdl

hardware

description "This package contains hardware support for FLASH memory on the TI OSK5912 Board." }

上爲 Flash package 的描述。

```
target OSK5912 {
        alias { "OMAP5912 OSK Board"    }
        packages { CYGPKG_HAL_ARM
          CYGPKG_HAL_ARM_ARM9
          CYGPKG_HAL_ARM_ARM9_OMAP5912
          CYGPKG_IO_SERIAL_OMAP5912
          CYGPKG_HAL_OSK5912
          CYGPKG_DEVS_FLASH_STRATA
          CYGPKG_DEVS_FLASH_OSK5912
          CYGPKG_DEVS_ETH_SMSC_LAN91CXX
          CYGPKG_DEVS_ETH_ARM_OSK5912}
       description "The OSK HAL package provides the support needed to run           on the
OMAP5912 processor of the TI OSK5912 Board." }
```

上爲平台的設定，package 爲該平台預設會 include 的 packages。可以看到 Ethernet 有兩個一個是 CYGPKG_DEVS_ETH_SMSC_LAN91CXX，另一個是 CYGPKG_DEVS_ETH_ARM_OSK5912。前者爲板子上晶片的 package，或者可以看爲晶片與板子的介面。

這個 configtool 有發現兩個 bugs。第一: 新工作的順序必需是先於 template 中選擇 target 平台及 build 模式，再 import .ecm 檔。若不選 template，即使.ecm 檔內已經設定好 target 平台，configtool 依舊會使用預設的平台。第二: template 會有預設的 processor，import .ecm 檔案載入使用者設定的 processor，會產生兩個 processors 並存的問題,沒辦法自動移除前者,要手動 disable 預設的 processor。對於 configtool 的使用流程請見附錄 1。

4.3. 系統修改

這一小節會說明系統需要修改的部份檔案。各個需要修改的檔案和該檔案的用途，請見附錄 5。移植的過程之中，第一步驟是選定 based board，就是用來修改的基礎，選用 Omap 1510(5910) innovator，先將所有的檔案複製一份在新的資料夾，並將所有檔案之中含有 innovator 的字眼改成 omap 5912 OSK，牽扯到的用途包括路徑和 package 的名稱。下面介紹修改的部份時，若只是名稱的更改將不再贅述。CDL 的命名規則如下<arch>_<variant>_<platform>，<arch>爲 arm，<variant>爲 arm9，<platform>是 omap5912。OMAP5912 爲 OMAP Innovator 的進化版本，故 register 和 interrupt vector 都有所不同，register 修改表請見附錄 2[1][20]；interrupt vector 修改表請見附錄 3[1][20]。工作路徑 c:\cygwin\opt\ecos\ecos-2.0\。移植的過程中，最好是每次測試都把電源重接，按電源鍵只會執行 reset 程序，不一定會重設所有 register，故某些 register 仍會保留其值，可能造成不可預期的錯誤。

4.3.1. har_arm9.cdl

加入 ARM 926EJS package:

```
cdl_option CYGPKG_HAL_ARM_ARM9_ARM926EJ {
        display "ARM ARM926EJ microprocessor"
        implements CYGINT_HAL_ARM_ARM9_VARIANT
        default_value 0
        no_define
        define -file=system.h CYGPKG_HAL_ARM_ARM9_ARM926EJ
        description "  The ARM926EJ has 16k data cache, 16k instruction cache, 16 word write buffer and an MMU." }
```

4.3.2. hal_cache.c

加入 cache 設定；

```
#elif defined(CYGPKG_HAL_ARM_ARM9_ARM926EJ)
# define HAL_ICACHE_SIZE                 0x4000
# define HAL_ICACHE_LINE_SIZE            32
# define HAL_ICACHE_WAYS                 4
#define HAL_ICACHE_SETS (HAL_ICACHE_SIZE/(HAL_ICACHE_LINE_SIZE*HAL_ICACHE_WAYS))
# define HAL_DCACHE_SIZE                 0x2000
# define HAL_DCACHE_LINE_SIZE            32
# define HAL_DCACHE_WAYS                 4
#define HAL_DCACHE_SETS (HAL_DCACHE_SIZE/(HAL_DCACHE_LINE_SIZE*HAL_DCACHE_WAYS))
# define HAL_WRITE_BUFFER                64
#define CYGHWR_HAL_ARM_ARM926EJ_CLEAN_DCACHE
```

line size 等於一個 block 大小，ways 指每個 set 有幾個 block，所以一個 set size 等於 line size X ways，於是 set 數量是 cache size / set size。Write buffer 是 16-word data 以及 4-address，故 size 為 64。

4.3.3. basetype.h

Endianess 設定:

```
#ifdef __ARMEB__
    #define CYG_BYTEORDER CYG_MSBFIRST // big endian
#else
    #define CYG_BYTEORDER CYG_LSBFIRST // little endian
#endif
```

4.3.4. mtl_OSK5912_rom.h

調整 memory layout 以符合平台的 layout。在 eCos-2.0 之前有工具可以使用，但 eCos-2.0 開始就沒有這個功能，只能手動設定。

下三項是 RAM 相關的設定，依序為 address, size, 和 access 限制；

#define CYGMEM_REGION_ram (0x00000000)

#define CYGMEM_REGION_ram_SIZE (0x04000000)

#define CYGMEM_REGION_ram_ATTR (CYGMEM_REGION_ATTR_R | CYGMEM_REGION_ATTR_W)

接著三項是 RoM 相關的設定，依序為 address, size, 和 access 限制；

#define CYGMEM_REGION_rom (0x0C000000)

#define CYGMEM_REGION_rom_SIZE (0x04000000)

#define CYGMEM_REGION_rom_ATTR (CYGMEM_REGION_ATTR_R | CYGMEM_REGION_ATTR_W)

最後這三項是設定可用的 heap 大小；

extern char CYG_LABEL_NAME (__heap1) [];

#define CYGMEM_SECTION_heap1 (CYG_LABEL_NAME (__heap1))

#define CYGMEM_SECTION_heap1_SIZE (0x04000000 - (size_t) CYG_LABEL_NAME (__heap1))

4.3.5. Osk5912_misc.c

MMU 的設定在 Osk5912_misc.c 裡面的 hal_mmu_init()。以下面第一段為例子，MMU 的 function name 為 X_ARM_MMU_SECTION()，其中參數，頭一個為 actual base address，接著是 virtual base address，第三個是以 MB 為單位的 size，第四個還第五個指示是否 cache 或 buffer，最後一個表明 access 的形態。第一段是 Chip select 0 的設定，本身是屬於 External memory，actual base memory 在 0x0000 0000，virtual base address 在 0x0C00 0000，有 cache 但沒有 buffer。

X_ARM_MMU_SECTION (0x000, 0x0C0, 64, ARM_CACHEABLE, ARM_UNBUFFERABLE, ARM_ACCESS_PERM_RW_RW); // CS0

X_ARM_MMU_SECTION (0x100, 0x000, 64, ARM_CACHEABLE, ARM_BUFFERABLE, ARM_ACCESS_PERM_RW_RW); // CS3

X_ARM_MMU_SECTION (0x200, 0x200, 1, ARM_UNCACHEABLE, ARM_UNBUFFERABLE, ARM_ACCESS_PERM_RW_RW); // Internal SRAM

X_ARM_MMU_SECTION (0xE00, 0xE00, 512, ARM_UNCACHEABLE,

ARM_UNBUFFERABLE, ARM_ACCESS_PERM_RW_RW); // Internal Peripherals

4.3.6. hal_arm_arm9_omap5912.cdl / hal_OSK5912.cdl

OMAP 5912OSK 的 GPP 處理器和 OMAP Innovator 不同，改成 ARM926EJ，5912 OSK 發展板需要更改 requires CYGPKG_HAL_ARM_ARM9_ARM926EJ，以及新增 register 的 default vakye 設定，形式為 cdl option，並在 CYGPKG_HAL_OSK5912 package 之中加入 require 的限制。但是處理器編號延用 innovator 即可；#define HAL_PLATFORM_MACHINE_TYPOE 234。表 4 為 register default value:

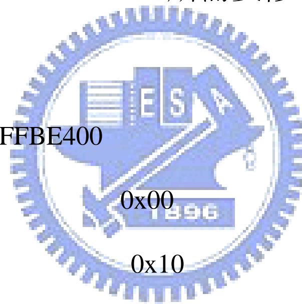
Table 4. 5912 register default value

VAL_FUNC_MUX_CTRL_3_WORD	0x09249FFF
VAL_FUNC_MUX_CTRL_4_WORD	0x3FE00001
VAL_FUNC_MUX_CTRL_5_WORD	0x3F4BFFFF
VAL_FUNC_MUX_CTRL_6_WORD	0x00000001
VAL_FUNC_MUX_CTRL_7_WORD	0x00001000
VAL_FUNC_MUX_CTRL_8_WORD	0x00001200
VAL_FUNC_MUX_CTRL_9_WORD	0x00201008
VAL_FUNC_MUX_CTRL_A_WORD	0x00000000
VAL_FUNC_MUX_CTRL_B_WORD	0x00000000
VAL_FUNC_MUX_CTRL_C_WORD	0x09000000
VAL_FUNC_MUX_CTRL_D_WORD	0x09249438
VAL_FUNC_MUX_CTRL_E_WORD	0x09249249
VAL_FUNC_MUX_CTRL_F_WORD	0x00000049
VAL_FUNC_MUX_CTRL_10_WORD	0x00000000
VAL_FUNC_MUX_CTRL_11_WORD	0x00000000
VAL_FUNC_MUX_CTRL_12_WORD	0x00000000
VAL_PULL_DWN_CTRL_0_WORD	0x00000000
VAL_PULL_DWN_CTRL_1_WORD	0x00040000
VAL_PULL_DWN_CTRL_2_WORD	0x00000000
VAL_PULL_DWN_CTRL_3_WORD	0x00000000
VAL_PULL_DWN_CTRL_4_WORD	0x00000000
VAL_PU_PD_SEL_0_WORD	0x00000000

VAL_PU_PD_SEL_1_WORD	0x00000000
VAL_PU_PD_SEL_2_WORD	0x00000000
VAL_PU_PD_SEL_3_WORD	0x00000000
VAL_PU_PD_SEL_4_WORD	0x00000000
VAL_VOLTAGE_CTRL_0_WORD	0x00000000
VAL_MOD_CONF_CTRL_0_WORD	0x000000000
VAL_ARM_CKCTL_WORD	0x000a
VAL_ARM_SYSST_WORD	0x0000
VAL_DPLL1_CTL_WORD	0x2413

4.3.7. Omap5912.h

修正更改或新增的 register address，所需要修改的 register 如附錄 2。截取一段如下



```
#define GPIO1_BASE      0xFFFBE400
#define _GPIO_REVISION              0x00
#define _GPIO_SYSCONFIG             0x10
#define GPIO1_REVISION          (vcyg16 *)(GPIO1_BASE + _GPIO_REVISION)
#define GPIO1_SYSCONFIG         (vcyg16 *)(GPIO1_BASE + _GPIO_SYSCONFIG)
```

4.3.8. hal_omap5912_setup.h / hal_platform_setup.h

定義 stack pointer base；#define OMAP5912_PLATFORM_SP_BASE 0x10000000。Register 有新增或移除，相對它的 initial value 必須要修改。例如 GPIO 的 configuration；

```
ldr   r0,=GPIO_BASE+_DIRECTION_CONTROL_REG
ldr   r1,=CYGOPT_SET_GPIO_CONF
str   r1,[r0]
```

EMIFF 的 MRS(mode register set) EMIFF_MRS，已經不再使用，而新增 EMIFF_MRS_NEW，可以更彈性的設定，於是更改如下，前一段是 innovator

code，第二段是 5912 OSK code:

```
ldr     r0, REG_TC_EMIFF_MRS
ldr     r1, VAL_TC_EMIFF_MRS
str     r1, [r0]


ldr r1, =VAL_TC_EMIFF_MRS_NEW_WORD
ldr r0, =EMIFF_MRS_NEW
str r1, [r0]
```

4.3.9. hal_platform_ints.h

新增及刪除 peripheral 的因素，必須修正 interrupt vector，以及 isr 數量也需修改。Interrupt vector 的數量由 54 個擴增到 93 個，請見附錄 3。最後再修改 ISR 相關定義:

```
#define CYGNUM_HAL_ISR_MIN        1
#define CYGNUM_HAL_ISR_MAX        93
```

4.3.10. omap5912_redboot_comds.c / OSK5912_redboot_comds.c

這兩個檔案是 redboot 的一部份，可以加入新的指令在 redboot 中使用。譬如 5912 新增 GPIO，我們就可以新增對 GPIO pin 塡値的指令，利用 HAL_API 來實作出 access 動作：

```
RedBoot_cmd(
"gpio",
        "Read, Write, or Configure a GPIO pin",
        "-p pin_num [-i|-o] [-s val] [-r] [-R] [-c] [-I] [-O] [-C]",
        do_gpio);


static void do_gpio(int argc, char *argv[]){
HAL_WRITE_UINT16();
```

HAL_READ_UINT16(););

4.3.11. omap5912_diag.c / hal_diag.h

diagnostic output support 是利用 UART 經過 serial port 傳輸，可以想像它是在 HAL 層對 serial port 的使用，比 serial device driver 更高一個層級。所以在這裡我們只需確定 UART 1 based address 的正確即可，其它方面不必修改。

4.3.12. OSK5912_misc.c

由於發展板上記憶體配置使用上的不同，在 hal_mmu_init()裡面如同前述 MMU 的設定做出我們欲使用的配置。在 hardware initial 部份，新增 disable watchdog timer 和 mux initial 分別是 plf_disable_wdt()和 plf_init _mux()，下列是 initial mux 的例子。若要使用 LCD 功能 FUNC_MUX_CTRL_A 到 FUNC_MUX_CTRL_12 必需保留硬體設定值。下面舉一個例子：

```
HAL_WRITE_UINT32(FUNC_MUX_CTRL_3, VAL_FUNC_MUX_CTRL_3_WORD);
```

4.3.13. Plf_io.h

設定 virtual address 與 physical address 的 mapping。5912 與 innovator 之間 Flash 的 mapping 不同，因此改成 5912 的設定，先列出 innovator 的設定，再列出 5912 的設定：

```
#define HACK_FLASH_VIRT_BASE 0x10000000
#define HACK_FLASH_SIZE      0x00400000
#define HACK_FLASH_MASK      0x003fffff
#define HACK_FLASH_PHYS_BASE 0x00000000


#define HACK_FLASH_VIRT_BASE 0x0c000000
#define HACK_FLASH_SIZE      0x02000000
#define HACK_FLASH_MASK      0x01ffffff
```

```
#define HACK_FLASH_PHYS_BASE 0x00000000
```

4.3.14. plf_stub.h

除了 5912 命名，其他不需修改。

4.3.15. Redboot_ROM-net.ecm

這個是設定最後 redboot 的功能，要包含所須的 package：

```
cdl_configuration eCos {
    description "" ;
    hardware    OSK5912 ;
    template    redboot ;
    package -hardware CYGPKG_HAL_ARM current ;
    package -hardware CYGPKG_HAL_ARM_ARM9 current ;
    package -hardware CYGPKG_HAL_ARM_ARM9_OMAP5912 current ;
    package -hardware CYGPKG_DEVS_FLASH_STRATA current ;
    package -hardware CYGPKG_DEVS_FLASH_OSK5912 current ;
    package -hardware CYGPKG_DEVS_ETH_SMSC_LAN91CXX current ;
    package -hardware CYGPKG_DEVS_ETH_ARM_OSK5912 current ;
    package -template CYGPKG_HAL current ;
    package -template CYGPKG_INFRA current ;
    package -template CYGPKG_REDBOOT current ;
    package -template CYGPKG_ISOINFRA current ;
    package -template CYGPKG_LIBC_STRING current ;
    package -template CYGPKG_NS_DNS current ;
    package -template CYGPKG_CRC current ;
    package CYGPKG_MEMALLOC current ;
    package CYGPKG_IO_FLASH current ;
    package CYGPKG_IO_ETH_DRIVERS current ;
    package CYGPKG_COMPRESS_ZLIB current ;
```

package -hardware CYGPKG_HAL_OSK5912 current ;

};

package 之後接 hardware 代表該 package 是個硬體，若接 template 則是包含了內部的部份設定。若有 conflict，configtool 會自動修正，但要注意的是只會修正一層，例如加入 package CYGPKG_LIBC_TIME 會自動加入需要的 CYGSEM_LIBC_TIME_TIME_WORKING，但是 CYGSEM_LIBC_TIME_TIME_WORKING 又需要 CYGPKG_IO_WALLCLOCK，這就不會自動加入了。

4.3.16. ser_omap5912.cdl

由於 innovator 沒有序列埠的使用程式，於是由相同晶片的 integrator 修改。UART 晶片型號: NS16550 (National Semiconductor)。總共有三個 UART，UART1 以 UART3 是由 ARM 來使用，UART2 是由 DSP 來使用，UART 定址如表 5。需要設定 name, baud-rate, and buffer size:

```
cdl_option CYGDAT_IO_SERIAL_OMAP5912_SERIAL0_NAME: {"\"/dev/ser0\""}
cdl_option CYGNUM_IO_SERIAL_OMAP5912_SERIAL0_BAUD: 115200
cdl_option CYGNUM_IO_SERIAL_OMAP5912_SERIAL0_BUFSIZE: 128
```

Table 5. UART 位址

UART	Address
UART 1	0xFFFB 0000
UART 2	0xFFFB 0800
UART 3	0xFFFB 9800

此外還要注意 baud-rate divisor 的設定，OMAP5912OSK 使用的設定是 26。baud-rate divisor = RS232 時脈 / (baud-rate * 16。最後，每個 device 都要 device table 註冊才可以使用，DEVTAB_ENTRY(l, name, dep_name, handlers, init, lookup, priv)，需要的資料如下:

- l: device table entry 的 C label。
- name: C string device name。
- dep_name: 下一層 device 的 device name。
- Handlers: I/O device 的 handler。
- Init: eCos 初始化時的 function call，其作用是 query the device 和設定硬體。
- Lookup: 當 cyg_io_lookup()對該 device 執行時，回應的 function call。
- Priv: device 的 object。

4.3.17. devs_eth_OSK5912.inl

設定 5912 OSK 上 Ethernet base address，使用 CS1 bank:

```
static lan91cxx_priv_data lan91cxx_eth0_priv_data = {
    base : (unsigned short *) 0x0480 0300, … 。
```

controller 和 5910 相同爲 smsc 91c96，所以延用 innovator 的程式。Controller space 定址在CS1。爲什麼不在 I/O device區? Ethernet與Compact Flash Card 的彈性使用爲原因：可以設定 CF card 使用 CS1，而 Ethernet 使用 CS2；或是只有 Ethernet，其使用 CS1。

經由對 output register 填入 data，若有順利傳送出去，control status register 會被填值，以聲名 ethernet 的連線成功。但並非一次就可以完成，於是我們得多等幾次，甚至無限迴圈等到傳送成功爲止。所以可能在 boot 過程的前面會出現 no link 的 message，直到 link 成功爲止。我們在 OSK5912_misc.c 之中，plf_hardware_init()加入這一段 code，包括相關設定以及達到上述效果：

```
#define ETH_CONTROL_REG 0x0480000b
```

```
*((volatile unsigned short *) 0xfffece08) = 0x03FF;
*((volatile unsigned short *) 0xfffb3824) = 0x8000;
*((volatile unsigned short *) 0xfffb3830) = 0x0000;
*((volatile unsigned short *) 0xfffb3834) = 0x0009;
*((volatile unsigned short *) 0xfffb3838) = 0x0009;
*((volatile unsigned short *) 0xfffb3818) = 0x0002;
*((volatile unsigned short *) 0xfffb382C) = 0x0048;
*((volatile unsigned short *) 0xfffb3824) = 0x8603;
hal_delay_us(3);
*((volatile unsigned short *) 0xfffb381C) = 0x6610;
hal_delay_us(30);
*((volatile unsigned char *) ETH_CONTROL_REG) &= ~0x01;
hal_delay_us(3);
```

Tftp 的使用，在 redboot 之下，使用指令可以取得在 pc 端 tftp root 資料夾的檔案: load －v －m tftp filename。在 kernel 之下，依照 embedded software development with eCos[18]，之中步驟 build 出 kernel。欲使用 tftp 必需再加入幾個 package:

```
package CYGPKG_NET current ;
    package CYGPKG_NET_FREEBSD_STACK current ;
    package CYGPKG_IO_FILEIO current ;
```

和填入 eth0 相關網路設定，如下:

```
cdl_component CYGHWR_NET_DRIVER_ETH0_ADDRS {user_value 1};
cdl_option CYGHWR_NET_DRIVER_ETH0_ADDRS_IP {user_value 140.113.208.18};
cdl_option CYGHWR_NET_DRIVER_ETH0_ADDRS_BROADCAST
  {user_value 140.113.208.255};
cdl_option CYGHWR_NET_DRIVER_ETH0_ADDRS_GATEWAY
  {user_value 140.113.208.254};
cdl_option CYGHWR_NET_DRIVER_ETH0_ADDRS_SERVER
```

{user_value 140.113.208.183 };

並使用 tftp_get() / tftp_put(); 在程式之中即可。

4.3.18. OSK5912_eth_drivers.cdl

CYGSEM_DEVS_ETH_ARM_OSK5912_ETH0_SET_ESA 可以設定 MAC address 由板子上的 EEPROM 讀取，或由使用者設定。我們已知板子的 MAC address，就直接使用使用者設定。

CYGDAT_DEVS_ETH_ARM_OSK5912_ETH0_ESA 可以設定 MAC address

4.4. Redboot 移植完成

圖 9 為移植完成，以 Redboot 開機的畫面:

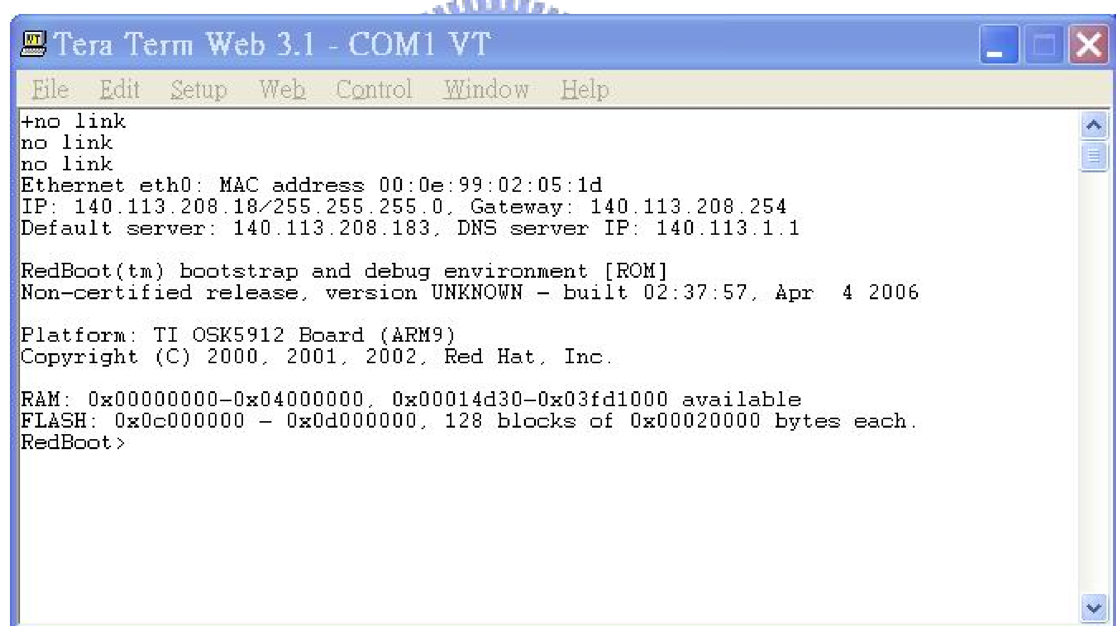
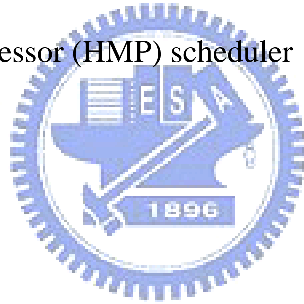


Figure 9. Redboot 開機畫面

為了測試移植 eCos 的正確性，可以移用 eCos 內附的測試程式來做驗證的工作，詳見附錄 4。

5. 異質雙核心動態分工排程器實作

本論文研究的中心概念為異質雙核心平台上的動態分工。如之前所述，動態精細分工是以同時考量整個平台的各異質核心的即時計算狀態為排程的依據，以期達到最高的效能。Figure 10 是我們設計的系統架構。原本利用 eCos 為 kernel 的系統架構包含有：應用程式 application(APP)，應用程式可以透過 eCos kernel 和 HAL 利用底層平台的硬體，如 GPP 和 DSP。其中 eCos kernel 最重要的元件是 MLQ scheduler。在 eCos kernel 中加入 Dispatcher、Service Registrar、和 core service table。應用程式經過 Scheduler API 可以和 kernel 或 Service Registrar 溝通，包括註冊 dual-core services，和下達執行 services 的命令。這些新加入的單元，即是 scheduler 的核心。以下，我們把這個異質雙核心動態分工排程器稱做 Heterogeneous Multi-Processor (HMP) scheduler。


E S A
1896

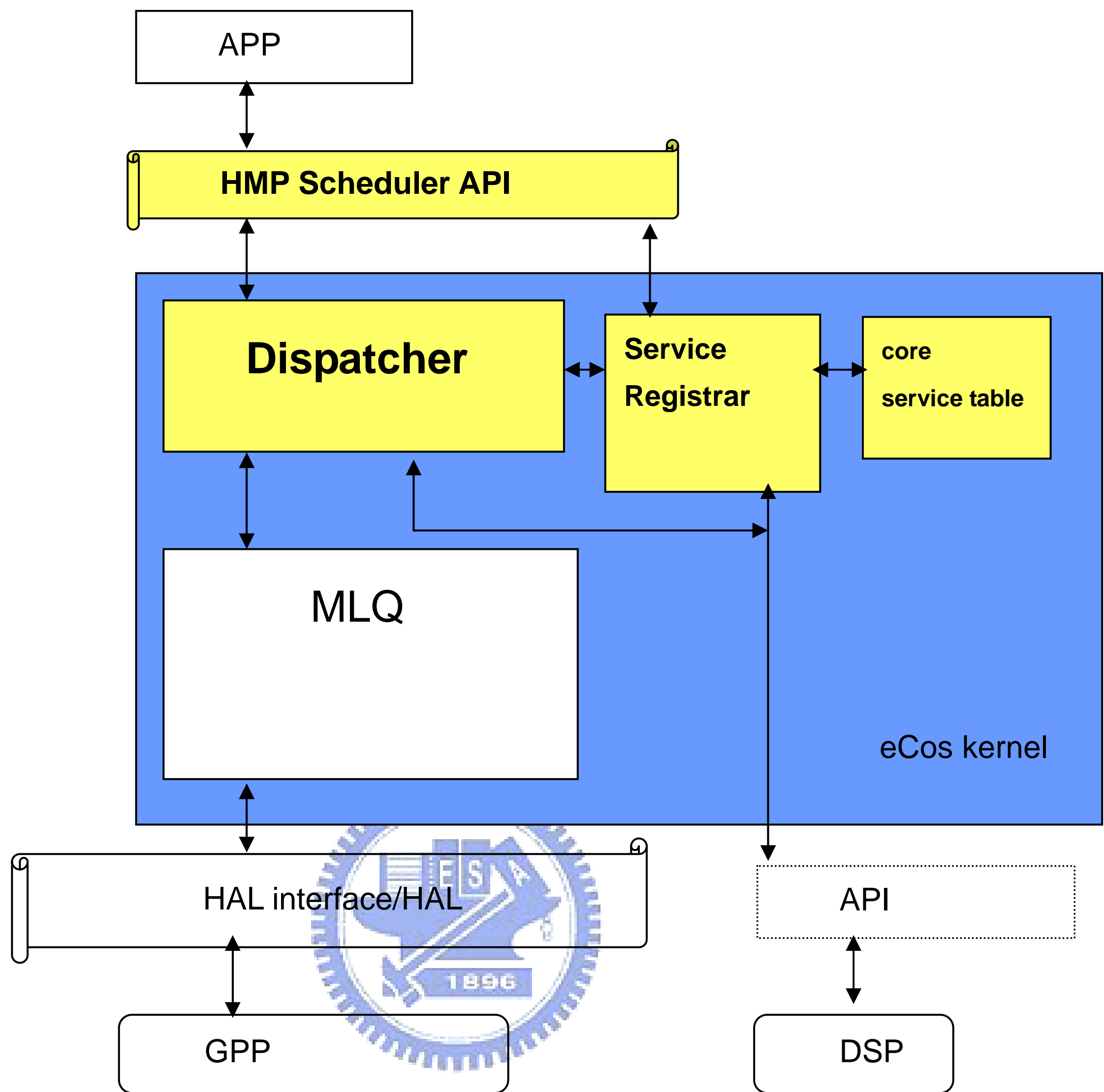


Figure 10. 系統架構

接下來會逐一介紹 scheduler 的每一個元件、它們各別的功能、以及運作方式。scheduler 的每個元件溝通都是雙向的動作，例如 Dispatcher 可能會向 Service Registrar 索取特定 service 的執行指標， Service Registrar 就會回覆之。圖中 eCos kernel 會透過一組 API 和 DSP 溝通，這一組 API 是同一研究團隊成員提供的介面，因為它不是本研究的重點，僅在最後做簡單介紹。

5.1. Scheduler API

這個介面提供 scheduler 使用上的便利性。Scheduler API 是新的 system

call，它的目標就是令使用者在移植單核心的應用程式爲異質雙核心應用程式時，只需做最小的修改，便能使用 scheduler。因此 Scheduler Interface 提供兩類三個系統呼叫的程式，它們是註冊和執行這兩類：

- HMP_register_service()
- HMP_invoke_service()
- HMP_remove_service()

HMP_register_service()和 HMP_remove_service()歸屬於註冊類。它們扮演的功能是向 kernel 註冊某項 service 爲 dual-core service 或是移除之。一旦註冊了某項 service 爲 dual-core service，之後使用這個 service 時，就會被引入 HMP scheduler，由 GPP 或由 DSP 負責執行都有可能。使用 service 的方法，在註冊後，HMP Scheduler API 會給使用者一個 ID。使用 service 就要呼叫 HMP_invoke_service()，填入 ID 指定註冊的 service 和填入原 service 的參數。HMP_invoke_service()被設計爲可容納不同數量參數的介面，因此各種數量參數皆可以處理。舉一個簡單的例子，在之後的實驗裡，測試的應用程式爲 H. 263 decoder。圖 11 中 Idct 是 decoder 其中一個程式，接受的參數是一個 short。欲使用 HMP scheduler 只需把程式改爲圖 12 的寫法。

```
void idct(short *block)
{ … }

void main()
{  …
   idct(block);
   …
}
```

Figure 11. 一般單核心 **idct** 函數的使用方法

Idct 程式不必做任何修改。在主程式中呼叫 idct 之前，先用 HMP_register_service(idct_image)註冊取得 id。之後要使用 idct 就改呼叫 idct(block)爲呼叫 HMP_invoke_service(id, block)。id 是向 kernel 取得的資料，而 block 是原本主程式的指標資料。全部就只要做到這些修改，應用程式幾乎看不到有單核心雙核心程式的不同，移植的便利性非常大。

```
void idct(short *data)
 { … }

void main()
{
   int    id;
   …
  id = HMPHMP_register_service(idct_image);
   …
   HMPHMP_invoke_service(id, block);
   …
}
```

Figure 12. HMP scheduler 下 idct 使用方法

OMAP 5912 是屬於雙核心的晶片，因此 HMP Scheduler 在執行排程工作之外，要負責 DSP 的啓動。當然 DSP 的啓動可以在 booting 時完成，考量並非每一種應用程式都需要用到 DSP，只在 HMP Scheduler 被啓動時才自動啓動 DSP，以減少 booting 的 overhead。另外有一個做法也是可以實行的，那就是將 DSP 啓動轉換成 bootstrap (Redboot)的指令，欲使用 HMP Scheduler 時，要先下該指令啓動 DSP，再執行應用程式。

5.2. Service Registrar 和 Core Service Table

應用程式向 Service Registrar 註冊 service，Service Registrar 收到 service 的資訊如下：

- Service function pointer
- Service DSP binary image pointer

得到所需資訊的 Service Registrar 開始進行註冊動作。首先在 core service table 內紀錄 function pointer。接著透過 DSP API 將 DSP binary image 傳入 DSP internal RAM，完成之後向 DSP 註冊該 service，得到 DSP 回傳的 service DSP ID。在 core service table 紀錄此 ID。之後都用此 ID 向 DSP 做指定 service 的動作。未來 function pointer 會改成 ARM 的 image。ARM 的 image 和 DSP 的 image 會包成一個資料結構。

完成上述程序的 Service Registrar 會產生一個 service kernel ID，回傳給應用程式。Service kernel ID 和 service DSP ID 是不同的。前者是應用程式和 kernel 互動時，用來指明 service 的依據。應用程式要執行 service 時，若 Dispatcher 判斷該 service 爲 GPP 執行，kernel 會根據 service kernel ID 到 core service table 找到正確的 function pointer，由 GPP 端執行 service；若 service 被判斷爲 DSP 執行，kernel 依 service kernel ID 在 core service table 內找對應的 service DSP ID。在通知 DSP 執行某個 service 時，一併傳入 service DSP ID，如此一來 DSP 便了解該執行哪一個 service。所以 service kernel ID 和 service DSP ID 是完全不相同。

Service Register 另外一項功能爲移除 service。接受移除指令的 Service Register 會自 core service table 中移除紀錄。並將 service image 從 DSP 內移除。

5.3. Dispatcher

Dispatcher 負責在應用程式要求執行 dual-core service 之後，判斷由哪一

顆處理器來執行會得到最短的執行時間，進而決定由哪一顆處理器執行。判斷由GPP 執行，就直接將 service 交給 MLQscheduler，依原本 eCos kernel 的路徑執行，但結束時會有些許改變；或是由 DSP 執行該 service。Dispatcher 的架構圖如下：

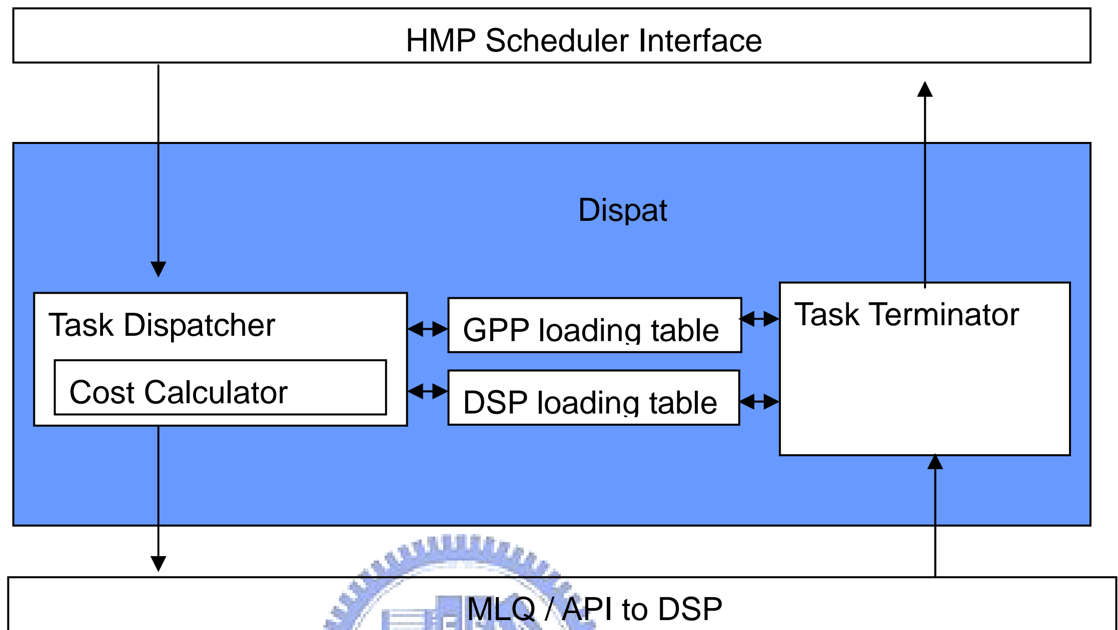


Figure 13. Dispatcher 架構圖

Dispatcher 是由三大單元組成，它們分別是 Task Dispatcher、Task Terminator、和 loading tables。執行 dual-core service 的資訊經 HMP Scheduler Interface 傳入 Dispatcher。一旦 Task Dispatcher 接收到資訊，會喚醒 Cost Calculator。Cost Calculator 開始計算 cost value，接著便回傳 Cost value 給 Task Dispatcher。Task Dispatcher 就以 cost value 判定由哪一處理器執行 service。Task Dispatcher 判定 service 的執行處理器，在屬於該處理器的 loading table 新增 service loading 資料。只要 service 執行完畢，就由 Task Terminator 來結束整個程序，包括通知應用程式 service 執行完成，和自處理器的 loading table 移除紀錄。由 loading table 的觀點來看，其 producer 是 Task Dispatcher，而 Task Terminator 爲 consumer。此外，當 service 必須對大量資料做運算使用到 pointer，而此 service 爲被決定爲 DSP 執行時，Dispatcher 得負責 DSP 可以得到整筆資料，之後有段落會專門介紹 Dispatcher 如何令 DSP 得到完整資料的實作方式。

5.3.1. Task Dispatcher

Task Dispatcher 得自應用程式的 service ID 是 service kernel ID。Task Dispatcher 的工作主要是依處理器的 loading 狀態，分配 service 給某一顆處理器負責執行，我們所使用的動態排程方式，基本上是根據過去的執行狀態來預測何者未來的 loading 比較低，便將工作交由該處理器執行。Task Dispatcher 會從 Cost Calculator 得到被呼叫的 service 在不同處理器所需的執行時間的估測，分別記錄在 time_ARM_prediction 和 time_DSP_prediction 這兩個變數中。此動態排程的判斷方法是將估測出的時間相比，來決定誰該執行該工作：

- time_ARM_prediction > time_DSP_prediction： service 判定給 DSP 執行。
- time_ARM_prediction < time_DSP_prediction： service 判定給 ARM 執行。
- time_ARM_prediction = time_DSP_prediction： service 判定給 DSP 執行。

在相等的情況下，service 會由 DSP 負責執行。其原因在於本平台系統專爲處理多媒體資料而設計，dual-core service 都是數位訊號處理的形式，在 DSP 和 ARM 的速度和耗電力比較，和忽略雙核心溝通的 overhead 的前提之下，DSP 更適合擔任執行角色，因此判定由 DSP 執行。

在 Cost Calculator 的部份，是利用指數平均數(exponential average)的方法來預測 service 的執行時間。依公式：

$$T_{n+1} \equiv \alpha \times t_n + (1-\alpha) \times T_n$$

T_{n+1} ： 預估這一次執行時間。

t_n ：上一次實際執行時間。

T_n ：上一次預估的執行時間。

α ：比例常數，常用 0.5；這裡使用 0.7 加重實際時間的比重。

每一類 service 執行時間的預測都是獨立的，換言之，每一類 service 都有專屬的一套預測變數。而且每一個 service 實際上是有兩份時間紀錄，一份爲 ARM 時間紀錄，一份爲 DSP 時間紀錄。一個 service 第一次執行時，t_0 及 T_0 都是預設值，所以前幾次的預測值不十分準確，但隨著執行次數的增加，不斷有實際執行時間可以修正預測的結果，提升精準度。

但是這種方式會出現一個嚴重的缺點。假設平均 DSP 在正常情況下處理 service idct 比 ARM 處理還快，其速度分別爲 500 個 time tick 與 800 個 time tick。一般情況下，也許有時候 DSP 慢一點而 ARM 又快一點，Cost Calculator 計算的結果是 ARM 較小，該 idct 就由 ARM 來執行；如果又回到 DSP 快的情況下，Cost Calculator 計算的值 DSP 會較小，變由 DSP 來執行 idct。實作過程中，發現一種特殊的情況，會極端地使效能變差。有些不明因素讓 DSP 執行 idct 的時間由平均的 500 個 time tick 大大提升到 2000 個 time tick，自此之後都不會由 DSP 執行 idct，也不會更新 t_n 和 T_n 。這麼一來，系統上的 idct 開始就會判定給 ARM 來執行。每一次 idct 平均執行的花費就會增加 300 個 time tick。

爲了解決這個問題，當上述情形發生時，Cost Calculator 會自動將此次的預測值除以特定值。以便讓下一次的預測值縮小。當我們假設 α 爲 0.7，而假設預測十分精準使得實際值和預測值都是 k。如下列公式：

$$
\begin{aligned}
T_{n+1} &= \alpha \times t_n + (1-\alpha) \times T_n \\
&= \alpha \times k + (1-\alpha) \times k \\
&= k
\end{aligned}
$$

經過實驗當預測值被除以 2 之後，公式改變如下。每次新值就會依 0.85 的比例向下修正，可以達到解決這個問題的需求。

$$
\begin{aligned}
& T_{n+1}' = \alpha \times t_n + (1-\alpha) \times T_n / 2 \\
& = \alpha \times k + (1-\alpha) \times k / 2 \\
& = 0.7k + 0.3k / 2 \\
& = 0.7k + 0.15k \\
& = 0.85k \\
& = 0.85 T_{n+1}
\end{aligned}
$$

利用上述方法完成執行處理器的指定之後，Task Dispatcher 要在對應處理器 loading table 增加 service 紀錄，和更新處理器全部執行時間。接著交付 service 給處理器執行。

5.3.2. Task Terminator

Task Dispatcher 和 Task Terminator 在系統中的角色是相反的。它接受 ARM 或是 DSP 結束 service 執行的訊息，自 service loading table 移除 service 紀錄，和更新處理器全部執行時間。下一步再通知應用程式”已經完成 service”。

5.3.3. Loading Tables

Loading table 用來紀錄各處理器的 loading 狀態，和各處理器上執行的 services。意義上 ARM(DSP) service table 是紀錄 service 種類，而 ARM(DSP) loading table 是紀錄每一個處理器上正在執行的 service。換句話說，同一種 service 種類在 ARM(DSP) service table 只有一筆資料，但在 ARM(DSP) loading table 上，只要應用程式對同一種 service 要求執行幾次，同一種 service 種類就會有幾筆資料，可是這些資料互相是不同。圖 14 為 loading table 的結構：

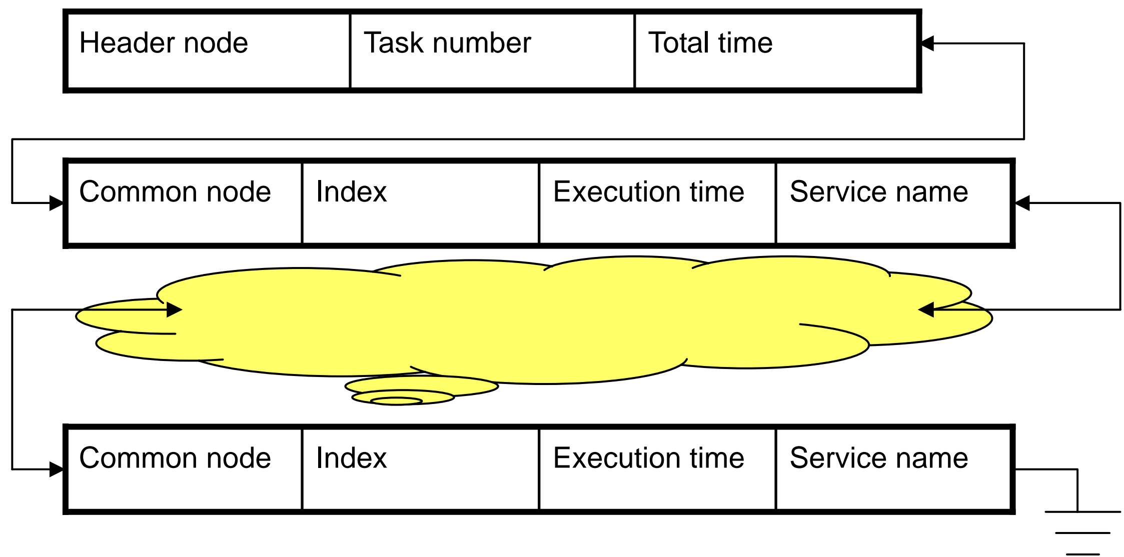


Figure 14.　Loading table

Loading table 分爲 header node 和 common node。Header node 紀錄其負責處理器上所有執行中 task 的數目(task number)，和所有的執行時間(total time)。未來我們會增加不同 cost function 考慮的因素，概括說來是不同種類的核心 loading 狀態，如電力消耗。在 header node 預留一些欄位可以用來記錄新增的 loading 狀態。每一個 common node 代表一個執行中的 service，紀錄著在 loading table 上的位置(index)、執行時間(execution time)、和名稱(service name)。這是一個 double direction linked list 資料結構，有鑑於不同的 service 有不同的執行時間，亦即 service 加入 table 的順序不會和離開 table 的順序相同，使用 linked list 處理這個特性。

5.4. 雙核心溝通方法

使用 mailbox 和 shared memory 的組合。OHMP 5912 上有四組 mailbox，其中兩組爲 ARM to DSP，另兩組爲 DSP to ARM。顧名思義，ARM to DSP 爲 ARM 發給 DSP 的 mailbox，DSP to ARM 是 DSP 發給 ARM 的 mailbox。每組 mailbox 上有兩個 16 位元 register，一個設定爲 command register，另一個爲 data

register。在 ARM to DSP mailbox 上，ARM 有完全的讀寫權利，DSP 只有讀取的權限；反之亦然。ARM 填完第二個 register 時，硬體會自動設定一個 flag，因此 DSP 會有 interrupt 產生。DSP 開始讀取兩個 register 的資料，讀完第二個 register 之後，flag 會再由硬體設回 0，表示 DSP 讀取完畢。利用 mailbox 傳遞做爲雙核心 service 執行的開端。

設計三種 ARM to DSP mailbox action command，如下：

- Register： 註冊 service。
- Invoke： 執行 service。
- Remove：移除 service。

以及 DSP to ARM mailbox action command，是 ARM to DSP 的回應指令如下：

- Return register： 完成 service 註冊。
- Return invoke： 完成 service 執行。
- Return remove： 完成 service 移除。

ARM 發 register mailbox action command 註冊 service 之前，ARM 利用 DSP API 把 service image 放入 DSP space internal RAM。完成註冊動作，DSP 會回 return register mailbox action command，並利用 data register 儲存 service DSP ID。再由 Service Registrar 存入 DSP service table。相同的運做模式，但功能相反是 remove。

最後一種指令爲 invoke 指令。ARM 在 command register 填入 invoke 指令，data register 填入自 DSP service table 取得的 service DSP ID。依此行爲 DSP 便會開始執行要求的 service。著眼於 mailbox 只提供兩個 16 位元的 register，沒有辦法利用 mailbox 傳入其它參數或資料。爲了解決這個問題，shared memory 正可派上用場。Shared memory 指得是 OMAP 5912 on-chip SRAM，共有 250 k-byte

可以使用。設計了 parameter table，結構如下：

Parameter 1
Parameter 2
Parameter 3
Parameter 4
Source data
Destination data

Figure 15. Parameter table

Parameter 1 到 parameter 4，每一個佔用 4 位元組。接下來 source data 及 destination data 每一個佔 4000 位元組。這兩塊大記憶體是選擇性使用，如果使用這兩塊記憶體，可以利用前面的 parameter 1 或 parameter 2 分別指到 source data 和 destination data。給應用程式設計者很大的彈性空間。如前述 H.263 idct(short block)的例子，可以把整個 short block 從 ARM 的 heap 搬到 source data，把 Parameter 1 指到 source data，並另用 Parameter 2 指定一塊記憶體。DSP 端的 idct.image 就可以從 Parameter 1 拿到整筆資料，運算完畢後，再將結果存到 Parameter 2 指到的位置。

每一個 service 會有一份此 parameter table。使用完之後可立即釋放記憶體空間。ARM 的定址模式爲 byte addressing，而 DSP 的定址模式爲 word addressing(每個 word 2 bytes)。兩種位址的轉換如下：

DSP_address = (ARM_address – ARM_base_address) / 2 + DSP_base_address

因此 ARM 在選擇 parameter table 的起始位址，必需在 16 位元制下以 0、4、8、或 C 結尾。DSP 才能正確讀取。否則會有不可預期的錯誤。

5.5. DSP API

表 6 簡介 DSP API 的名稱與其功能。

Table 6. DSP API

API name	description
dsp_init()	啓動 DSP 處理器
dspmmu_for_sdram()	啓動 DSP MMU
pmem_init()	啓動 DSP internal ram
pmem_allocate()	取得 DSP internal ram
pmem_free()	釋放 DSP internal ram

這部份的設計是配合 DSP 的 real-time kernel 設計的，是由本實驗室另一位研究生負責，因此不在這邊討論其細部設計。

6. 實驗結果

本論文一直闡述想的法在第五章介紹了實作的過程。在這一章將會做五組不同的實驗來驗證 HMP Scheduler 的動態排程功能和系統效能。首先我們會介紹實驗的環境，接著再介紹各組實驗，包括實驗的方法和討論。

6.1. 實驗環境

在這一段落先介紹實驗使用的的應用程式，它是 H.263 的 decoder。 H.263 decoder 是用來將已經壓製好的 m4v 檔案解回 yuv 檔案。主要的 function 也就是將在實驗中註冊為 service 的 function 如下：

- idct：對所有 frame 做 inverse DCT。
- dequant_inter：對 inter frame 做 inverse quantization。
- deqaunt_intra：對 intra frame 做 inverse quantization。
- interpolation：對 inter frame 做垂直點之間、水平點之間、和對角線點之間的 interpolation。

GPP 核心和 DSP 核心都設定為 96MHz。使用的 bit-stream 是 foreman。我們準備了三種不同的 bit rate，其特徵如表 7 所示：

Table 7. 實驗 bit-stream

Name	Type	Frame	Frame rate	Bit rate
Foreman	QCIF	300	30	192k
Foreman	QCIF	300	30	128k
Foreman	QCIF	300	30	64k

測定用的 timer 是 OMAP 5912 平台上 ARM private peripheral timer 2，它的頻率為 12MHz。time tick 與 micro second 的轉換如下：

(time tick +1) * 2 / Frequency * 1000 = micro second

Table 8. ARM Private Timer 2 Registers

BYTE ADDRESS	REGISTER NAME	DESCRIPTION	ACCESS WIDTH	ACCESS TYPE
FFFE:C600	MPU_CNTL_TIMER_2	Timer 2 Control Timer Register	32	RW
FFFE:C604	MPU_LOAD_TIM_2	Timer 2 Load Timer Register	32	W
FFFE:C608	MPU_READ_TIM_2	Timer 2 Read Timer Register	32	R

表 8 爲 timer register 列表。第一欄紀錄每個 register 在 ARM 記憶體空間的 address。接著兩欄位是 register 名稱和描敘。第四欄代表每個 register 的長度是 32 位元。第一列是 control register，用來開啓或關上 timer，名爲 MPU_CNTL_TIMER_2。第二列的 register 是 MPU_LOAD_TIM2，使用 timer 前可以填入自訂的值，由該值開始倒數，等於是控制 timer 計時的全部長度。第三列是 timer 被啓動之後，依時間 MPU_LOAD_TIM_2 遞減所得的新值會出現在這個 register 中。此 register 稱爲 MPU_READ_TIM_2。此外 MPU_CNTL_TIMER_2 尙可控制當 timer 倒數到零時的動作，共有兩種可以選擇，其一是狀態自動全部重設，再開始倒數；另一種是直接停止。本實驗將 MPU_LOAD_TIM_2 設爲最大値 2^32，足夠每一次實驗的時間，故當 timer 倒數到零時就直接停止，不再動作。自下節開始有各組實驗目的和結果的探討。

6.2. 動態排程實驗

這個實驗的目的在於觀察動態排程的實作結果。在相同環境下，使用 bit rate 192 K-bps 的 bit-stream，執行 HMP Scheduler。預測每一種 service 各自分佈在 ARM 和 DSP 上的比例會相差無幾，但是不會每次的數字都不變。下面表 9 到表 11 是相同的實驗執行三次的結果：

Table 9. 動態排程實驗 1

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3888	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	455	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
app time	14256			
sum(次)		61331	13562	74893

Table 10.動態排程實驗 2

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3886	18059	3643	21702
dequant inter	2220	14699	3379	18078
dequant intra	454	2947	677	3624
interpolation	4162	25625	5864	31489
service time	10723			
app time	14258			
sum(次)		61330	13563	74893

Table 11.動態排程實驗 3

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3894	18033	3669	21702
dequant inter	2218	14693	3385	18078
dequant intra	455	2941	683	3624
interpolation	4164	25632	5857	31489
service time	10733			
app time	14271			
sum(次)		61299	13594	74893

每張表前四列代表不同 service，第一欄是每個 service 所花費的時間，單位是毫秒。第二和第三欄分別是該 service 在 ARM 和 DSP 上處理的次數。而最後一欄是每個 service 總共會處理的次數，因爲所用的 bit stream 相同，故這個值不

會改變。第五列是所有 service 耗去的時間。第六列代表整個 decoder 執行的時間。第七列則總和 ARM 和 DSP 的執行次數。

以 interpolation 為例，ARM 上執行的次數和 DSP 執行的次數三個實驗下來都是約為 5：1；ARM 和 DSP 執行的總次數為 6：1。首先看到動態排程在執行時期確實達到動態分配工作的能力。然而以執行時間為分配工作的依據，在這個實驗中可以看到 ARM 所得到的工作量是 DSP 的 6 倍。這是不是意味著 ARM 處理這些 service 的速度較快呢？要探討這個問題，設計了下面一個相異處理器實驗。

6.3. 相異處理器實驗

對於 bit rate 為 192 K-bps 的 bit-stream，實驗在 Pure ARM、Pure DSP、和 Dual-core 時的效率。在每個處理器上跑 10 次 decoder 再做平均，觀察其效能的不同。本實驗可以達到兩個目的。第一，探討 ARM 和 DSP 對 decoder service 的相對速度。第二，驗證 dual-core 是否可以達到雙核心加成的能力：

Table 12.Pure ARM

	service time(ms)	application time(ms)
1	12454	15855
2	12451	15853
3	12459	15850
4	12451	15853
5	12452	15845
6	12452	15855
7	12451	15855
8	12450	15853
9	12454	15857
10	12452	15852
average	12453	15853

Table 13.Pure DSP

	service time(ms)	application time(ms)
1	14510	17911
2	14638	18039
3	14892	18301
4	14775	18184
5	14560	17970
6	14475	17876
7	15252	18652
8	13558	16966
9	13577	16987
10	14739	18039
average	14498	17893

Table 14.Dual core

	service time(ms)	application time(ms)
1	11003	14446
2	11055	14502
3	12471	15924
4	14868	18365
5	10170	13598
6	10581	14030
7	11041	14496
8	11803	15287
9	10708	14117
10	11494	14381
average	11520	14915

上方三張表分別是 pure ARM、pure DSP 和 dual-core 三種執行的結果。看到 pure ARM 與 pure DSP 的比較。大家都知曉 DSP 就是爲了處理數位訊號而特別設計的處理器，其效能應該會比 ARM 來得好。在這組比較中得知卻是相反的。其中的原因在於 HMP Scheduler 系統的 overhead。細數 overhead 有下例幾項：

- DSP invocation overhead

- Memory copy overhead

由第一項開始說明。ARM 如何通知 DSP 開始工作以及執行哪一個 service？靠著 mailbox 來達成。ARM 如何得知工作已結束？也是靠 mailbox 來達成。因此每個 service 交由 DSP 處理無可避免得承受 mailbox overhead。另外在 DSP 端 kernel 要處理 DSP 上排程問題，也必須執行 context switch，這些都是 overhead。測量之下得到一次 DSP invocation overhead 平均時間要 450 個 time tick。比較 ARM 和 DSP 對 idct 最長的執行時間，ARM 為 600 個 time tick；DSP 為 2080 個 time tick。DSP invocation overhead 佔了 DSP 執行時間的五分之一。

另外，第二項 overhead，是 memory copy overhead。DSP 被限制無法存取 ARM 的記憶體空間，除了共用記憶體外。所以 HMP Scheduler 需負責將資料由 ARM heap 區搬到共用記憶提供 DSP 使用；相同地，DSP 運算完畢後，HMP Scheduler 也要負責填回到 heap 區。當然這一段時間包含在 service 給 DSP 處理的執行時間內。就 idct 而言，必需搬動 64 筆 short 資料兩次，共要花 100 個 time tick。

表 12、13 和 14 記錄了所有 service 執行時間的總和以及應用程式執行的時間。現在讓我們檢視各 sercive 在不同處理器上的執行時間，將表 12 和表 13 各自第一筆實驗的細部資料列出如下二表；

Table 15.service time

ARM	time(ms)	DSP	time(ms)
idct	3795	idct	4788
inter	2895	inter	3088
intra	591	intra	635
interpolation	5172	interpolation	5998
service time	12454	service time	14510
app time	15855	app time	17911

Table 16.Reference profile

	ARM clock cycles per MB	DSP clock cycles per MB	ratio
(Y)DC pred. & comp.	44268	8465	5.229533373
(Y)Transform	188250	23178	8.121925964
(Y)Quant	247201	39521	6.25492776
(Y)Inv. Quant	238948	30002	7.964402373
(Y)Inv. Transform	202184	27863	7.256361483
(Y)Reconstruct	106392	17835	5.965349033
(DC)DC pred. & comp.	50458	10142	4.97515283
(DC)Transform	88928	11379	7.815097988
(DC)Quant	134689	20814	6.47107716
(DC)Inv. Quant	124131	15410	8.055223881
(DC)Inv. Transform	100497	13907	7.226360825
(DC)Reconstruct	51913	8861	5.858593838
cavlc	396022	46204	8.57116267
ILF	420018	37209	11.28807547
Total	2393899	310790	

在表 15 裡面可得到各項 service 的執行時間、全部 service 總和執行時間、和應用程式執行時間。我們著重於各項 service 在不同處理器上執行時間的討探，藉此可以了解不同處理器對同一個 service 的處理速度。以 idct 為例，ARM 和 DSP 的比例是 1：1.26(3.7：4.7)。但是根據研究室學長過去的的測試[24] ARM 和 DSP 對 idct 的執行時間比為 7.226：1(Inv. Transform 欄位)，這個數字是在 ARM 沒有開啓 cache 的情況下產生的。估計 ARM 的 cache 打開後，ARM 的處理時間可能會下降到為 DSP 的 2 到 4 倍。它意味著 DSP 對 idct 的執行速度比 ARM 快。比對我們的實驗是 ARM 比 DSP 快。這個差別的來源是因為我們的 service 是引用 reference software 用 C 寫成，但是[24]的 DSP 部份是特別以組語寫成，效率上佔極大優勢。此外 OMAP 5912 的 DSP 是 C55 系例，它的乘法運算是 16 位元 X16 位元的運算。在 reference software 中卻有 32 位元 X32 位元的乘法運算，這使得 DSP 必須利用兩次 16 位元乘法運算來模擬 32 位元乘法運算。因此我們測量的結果 DSP 效能下降很多。

何以我們不也改用組語來寫 DSP 部份的程式和改變乘法的使用方式？其原因是 HMP Scheduler 設計的重點在於減少 platform dependence 以及方便程式的移植。忠重這個想法，我們對 reference software 只做最少且必要的修改，其他一律依原程式的寫法忠實呈現-- 不特別將 C 語言改為組合語言也不特別對程式做最佳化。所以在這裡 DSP 的效能不如[24]所測量出來的效能。

6.4. 相異 bit rate 實驗

在這個實驗中，考量不同 bit rate 對 HMP Scheduler 效能的影響。以 idct 為例，在 bit rate 較低的情況下，會有較多的 0 出現；反之在 bit rate 較高的情況下，0 就比較少。對於底層硬來說，執行乘以 0 和乘以非 0 的動作所花費的時間也有所不同。希望藉由這個實驗探討不同 bit rate 對 HMP Scheduler 的效能有何不同。下面三個表格依順由 bit rate 高到底排列：

Table 17.bit rate 實驗：192k bps

	service time(ms)	application time(ms)
1	11004	14447
2	11056	14502
3	12471	15925
4	14869	18365
5	10171	13598
6	10582	14031
7	11042	14496
8	11804	15288
9	10708	14117
10	11494	14381
average	11520	14915

Table 18.bit rate 實驗：128k bps

	service time(ms)	application time(ms)
1	9514	11967

2	9139	11565
3	10204	12660
4	9598	12039
5	9525	11969
6	8682	11084
7	8760	11163
8	8633	11033
9	8526	10936
10	9031	11455
average	9161	11587

Table 19.bit rate 實驗：64 bps

	service time(ms)	application time(ms)
1	7082	8541
2	8555	10050
3	7320	8788
4	6367	7789
5	6496	7925
6	7371	8836
7	6356	7781
8	7658	9127
9	8444	9935
10	9074	10559
average	7472	8933

每一張表各執行 10 次 decoder，然後再取平均值。第一欄 service time 是每一個 service 所花費的時間的總和；第二欄 application time 代表整個 decoder 完成的時間，但不包括 I/O 的部份。192 K-bps 的 service time 是 11520 毫秒；128 K-bps 的 service time 是 9161 毫秒；64 K-bps 的 service time 則爲 7472. 毫秒。很明顯地解 bit rate 較低的 bit-stream 所使用的時間比 bit rate 較高的 bit-stream 少。以這個 300 張 frame bit-stream 爲例，decoder 處理 192 K-bps 的效能爲每秒 26 張 frame；128 K-bps 是每秒 33 張 frame；最後，最高效率的 64 k-bps 則是每秒 40 張 frame。

可以想像得到對於 bit-rate 較高的 bit-stream，其運算量會較高。對於較高的運算量，相較之下 DSP invocation overhead 和 memory copy overhead 這兩個 overhead 和運算量的比例會降低。可以減低 overhead 對較能的影響。

6.5. DSP delay 實驗

HMP Scheduler 發展的概念是動態的精細分工排程。換言之，HMP Scheduler 會動態地分辨處理器的狀態。理論上當 DSP 處理 service 的速度快於 ARM，service 會被分配到 DSP。但如果 DSP 同時又因爲有其他工作在做--不是屬於 decoder 的工作在執行中。在 DSP 上會發生 context switch 輪流處理 service 和另一項工作。這麼一來 service 的處理，從開始到完成的時間就會變長。這個影響會使得 HMP Scheduler 將 service 分配給 ARM 執行，以取得相較之下最好的效能。

我們在 DSP 端隨機地增加不同工作，分別會使 DSP 上的 service 延遲 1000~5000 的 time ticks 不等。因爲測試用的 services，單次 service 執行的時間在正常情況下最大不會超過 2500 個 time tick。選擇延遲最大 5000 個 time tick 就會對 DSP 造成很大的影響，可以達到模擬的目的。

在實驗中爲了使 DSP 延遲，設計除了 decoder 的 service 之外的另一個給 DSP 執行的延遲用 service，稱之爲 D-service。Kernel 利用隨機的方式決定是否使 DSP 延遲，如果決定延遲，就同時將 decoder service 和 D-service 一併透過雙核心溝通機制通知 DSP 執行。D-service 會到上一節介紹過的 parameter table 讀取 kernel 準備的參數。此參數告知 D-service 要做多少次 loop，換言之可以決定延遲的 time tick。

經過測量，一次 mailbox 的往返會花費平均 450 個 time tick。通知 D-service 延遲的時間必需減掉 mailbox 往返的花費，所以是 550~4550 個 time tick。D-service 內每個 service 設計爲消耗 550 個 time tick，於是 550~4550 個 time tick 延遲分別是 loop 1 ~ 9 次。實驗結果如表 18、19、20、21 和 22：

Table 20.dual-core without delay

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3889	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	456	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
application time	14257			
sum(次)		61331	13562	74893

表 18 為對照組，數據內容和表 9 相同。下兩組表格則是本實驗的實驗組。Dual-core with delay 和 delay distribution 兩張表格為一組。首先看到 dual-core with delay，第一欄為 service 單獨的執行時間、service time、和 application time。接著兩欄分別是各 service 執行在不同處理器的次數。再來看到 delay distribution 這張表，delay 這一列顯示的數字是 delay 的 tick 數，如 1000 是指 delay 1000 個 time tick 到 1999 個 time tick。2000 是指 delay 2000 個 time tick 到 2999 個 time tick。但是 5000 的欄是不同的，只有到 5050 個 time tick。因為 5000 對於系統已經是很大的 delay，故只使用到 5050 個 time tick。

Table 21.dual-core with delay 1

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3889	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	456	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
application time	14257			
sum(次)		61331	13562	74893

Table 22.Delay distribution 1

delay(tick)	1000	2000	3000	4000	5000

次數	945	1038	788	893	40

Table 23.Dual-core with delay 2

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	4348	18247	3455	21702
dequant inter	2608	15039	3039	18078
dequant intra	532	3006	618	3624
interpolation	4823	26206	5283	31489
service time	12310			
application time	15837			
sum(次)		62498	12395	74893

Table 24.Delay distribution 2

delay(tick)	1000	2000	3000	4000	5000
次數	918	1033	824	865	60

比較加入 DSP delay 和沒有加入 DSP delay 的數據，可以看到 DSP 處理的次數都下降。Dequant inter 的數字由 3385 降到 3039 和 3045，約有 350 次的減少。但是看到 dequant intra 的下降卻只有 70 左右，這是因爲會呼叫到 dequant intra 的時機很少，只有 intra frame 上的 macro block 和 inter frame 上的 I macro block 會呼叫。這個 foreman bit-stream 的 intra frame 只有第一張，其他都是 inter frame。所有呼叫 dequant intra 的總次數也只有 3624 次，和其他 service 呼叫量有一個位數之差，所以加入 DSP delay 改變的幅度相比之下不大。

在之前的實驗結果 dual-core 的效能大於 pure ARM 和 pure DSP。當 DSP delay 發生，HMP Scheduler 就認爲 DSP 上 service 的執行時間增加，若執行時間大於 ARM 的時間，就會判定 service 由 ARM 執行。既便是原本由 DSP 執行較快的 service 也因 DSP delay 的影響，就不再交由 DSP 負責。Dual-core 加入 DSP delay 因素使得 service time 由 10724 1 毫秒上升到 12325 毫秒，效能降低了。換另一方面看，若系統測試到 DSP 方面的速度下降後，仍執意由 DSP 執行，效能肯定

更下降。爲了避免這情形發生，HMP Scheduler 會判斷由 ARM 負責執行 service。再者，看到 pure ARM 的平均 service time 是 12453 毫秒，dual-core 加入 DSP delay 的效能下降到和 pure ARM 差不多，但又比 pure ARM 好一些。因爲由 ARM 負責的 service 量上升，更接近於 pure ARM 的工作清形。

原始期望的效能是 dual-core > pure DSP > pure ARM。由於前述的 overhead 尚未克服，使得效能成爲 dual-core > pure ARM > pure DSP。倘若上述 overhead 都能克服，直接影響到 DSP 在系統中的能力會大幅上升，進而使得 dual-core 也得到幫助。

7. 結論與展望

本篇論文目的在證實一個概念，這個概念就是動態精細分工的排程方式可以在多媒體雙核心平台上有較好的運作效能。為了證實此想法我們提出一個全新的 HMP Scheduler。有鑑於 eCos 作業系統是以元件組成的作業系統，有極高的彈性和修改的便利性，系統開發者可以自由地依需要的功能選擇對應的元件。利用此特點本論文在 eCos kernel 中加入新的 HMP Scheduler 元件來實作。我們將 HMP Scheduler 設計成依據整個系統的狀態來動態地分配 task 給 GPP 處理器或是 DSP 處理器。這裡考慮的系統狀態是一個抽象的處理器 loading 狀態，排程的標竿是儘量做出使整個系統 loading 最小的排程。因此考量使用預測出最小處理器執行時間來分配工作：只要工作預測出來在 GPP 處理器有比在 DSP 處理器有更短的執行時間，就把工作分配給 GPP 處理器；反之亦然。這裡只用了最短執行時間的因素作為排程依據，未來將會考慮更多的因素來控制動態的排程處理，例如電力消耗和 deadline 等等，以達到更精確更多功能性的考量。在 HMP Scheduler API 方面，HMP Scheduler API 的設計使得現存的單核心程式可以做最少的修改，便可以移植到使用 HMP Scheduler 的雙核心平台上。

除了動態精細分工排程，我們也提出一個雙核心溝通的方法。以有效率的溝通為設計目標，儘量簡化一切可以略去的資訊來往。利用 mailbox 和共用記憶體的組合來實現這個設計。除了一道通知 DSP 執行 service 的 mailbox 動作，其他不能省略的資料都以設計好的資料結構方式，放置在共用的記憶體上。如此一來，便可以不必靠好幾次的 mailbox 來回傳遞資訊。此外，eCos 在本論文實作之前，沒有移植到 5912 OSK 的版本。這個移植工作完成之後，eCos 便多了一個可以直接利用的 5912 OSK 版本。

然而本論文實作之後，發現了一個大障礙，那即是雖然本論文提出的雙核心溝通方式已經減去許多不必要的雙核心資訊往返，但其它的 overhead 沒辦法有

效地最小化或甚至消除。此 overhead 包含 DSP invocation 的花費，記憶體搬移的花費。第一個 DSP invocation 的花費在 mailbox 方面是無可避免的支出，但在 DSP kernel 的部份可以再做最佳化。第二個問題可以利用新的 compiler 來改善。記憶體搬移問題，可以將會共同使用的資料就直接配置在共用的 SRAM 上，如此可以減少搬動的時間花費；其他尚有必需與應用程式分開事先建立給 DSP 使用的 service image 的麻煩，但是利用新的 compiler 可以將應用程式和 service image 一併產生，全自動化增加便利性。

一旦能有效地解決上述提出的 overhead，在雙核心平台上精細分工的排程方式將會革命性地取代目前廣泛被做用的鬆散式結合排程方式。進而為日益複雜的嵌入式應用提供更高品質更高效能的環境。

8. 參考文獻

[1] C-N Chiu, C-T Tseng, and C-J Tsai, "Tightly-Coupled MPEG-4 Vedeo Encoder Framework on Assymetric Dual-Core Platforms," *IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS* Vol. 3, 2005.

[2] “OMAP5912 Application Processor Data Manual,” Texas Instruments, Dallas, Texas, [Online], Available: http://www.ti.com.

[3] R. L. Cancian, L. F. Friedrich, “Performance Evaluation of Real Time Schedulers for a Multicomputer,” *IEEE International Workshop on Distributed Simulation and Real-Time Applications,* 2002.

[4] Satoshi Kaneko et al, “A 600-MHz Single-Chip Multiprocessor With 4.8-GB/s Internal Shared Pipeline Bus and 512-kB Internal Memory,” *IEEE Journal of Solid-State Circuits,* Vol. 39, NO. 1, January 2004.

[5] M. J. Harrold, “Performing Data Flow Testing in Parallel,” *IEEE,* 1994.

[6] M. Annavaram, E.Grochowski, J. Shen, “Mitigating Amdahl’s Law Through EPI Throttling,” *IEEE International Symposium on Computer Architecture,* 2005.

[7] J. W. Wendorf, R. G. Wendorf, Hideyuki Tokuda, “Scheduling Perating System Processing on Small-Scale Multiprocessor,” *IEEE,* 1989.

[8] A. G. Greenberg, “Design and Analysis of Master/ Slave Multiprocessor,” *IEEE Transactions on Computers,* Vol. 40, No. 8. August 1991.

[9] G. Manimaran, C. Siva Ram Murthy, “A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis,” *IEEE Transactions on Parallel and Distributed Systems,* Vol. 9, No. 11, November 1998.

[10] A. Avritzer et al, “The Advantage of Dynamic Tuning in Distributed Asymmetric Systems,” *IEEE,* 1990.

[11] S. Majumdar, “Performance Scalability in Multiprocessor Systems with Resource Contention,” *IEEE,* 2000.

[12] S. Saewong, R. Rajkumar, “Cooperative Scheduling of Multiple Resources,” *In Proceedings of 20th IEEE Real-Time Systems Symposium,* 1999.

[13] K. K. P. Research, “Increasing Functionality in Set-Top Boxes,” *In Proceeding of IIC-Korea, Seoul,* 2001.

[14] A. Ferrari et al, “The Design and Implementation of a Dual-Core Platform for Power-Train Systems,” *Convergence 2000,* Detroit (MI), USA, October 2000.

[15] Paolo Gai, Luca Abeni, G. Guttazzo, “Multiprocessor DSP Scheduling in System-on-a-chip Architectures,” *IEEE Proceedings of the 14th Euromicro Conference on Real-Time Systems,* 2002.

[16] Bart Veer, John Dallaway, “The eCos Component Writer’s Guide,” Red Hat, 2002.

[17] N. Garnett, J. Larmour, A. Lunn, G. Thomas, “eCos Reference Manual,” Red Hat, 2003.

[18] “eCos User Guide,” eCosCentric, 2003.

[19] A. J. Massa, “Embedded Software Development with eCos,” PERNTICE HALL, 2002.

[20] “OMAP Starter Kit (OSK) OMAP5912 Target Module Hardware Design Specification,” OMPA, Revision 2.3, July 2004.

[21] “OMAP5910 Dual-Core Processor Data Manual,” Texas Instruments, Dallas, Texas, [Online], Available: http://www.ti.com.

[22] C-P Wang, *Design and analysis of a Unified Asymmetric Multiprocessors Scheduler*, master thesis, NCTU, June 2005

[23] The University of North Carolina, “Feasibility Analysis of Preemptive Real-Time Systems upon Heterogeneous Multiprocessor Platform,” *Prceedings of the 25th IEEE Internal Real-Time Systems Symposium,* 2004.

[24] Cheng-Nan Chiu, *H.264 Video Encoding Optimization on Dual-Core Platform*, master thesis, NCTU, June 2005.

9. 附錄

9.1. 附錄 1: configtool 使用流程

首先得設定好路徑:

ecos repository: opt/ecos/ecos-2.0/package

gnutool: /opt/ecos/gnutools/arm-elf/bin

cygwin 介面下打入指令 configtool 將出現下圖畫面。

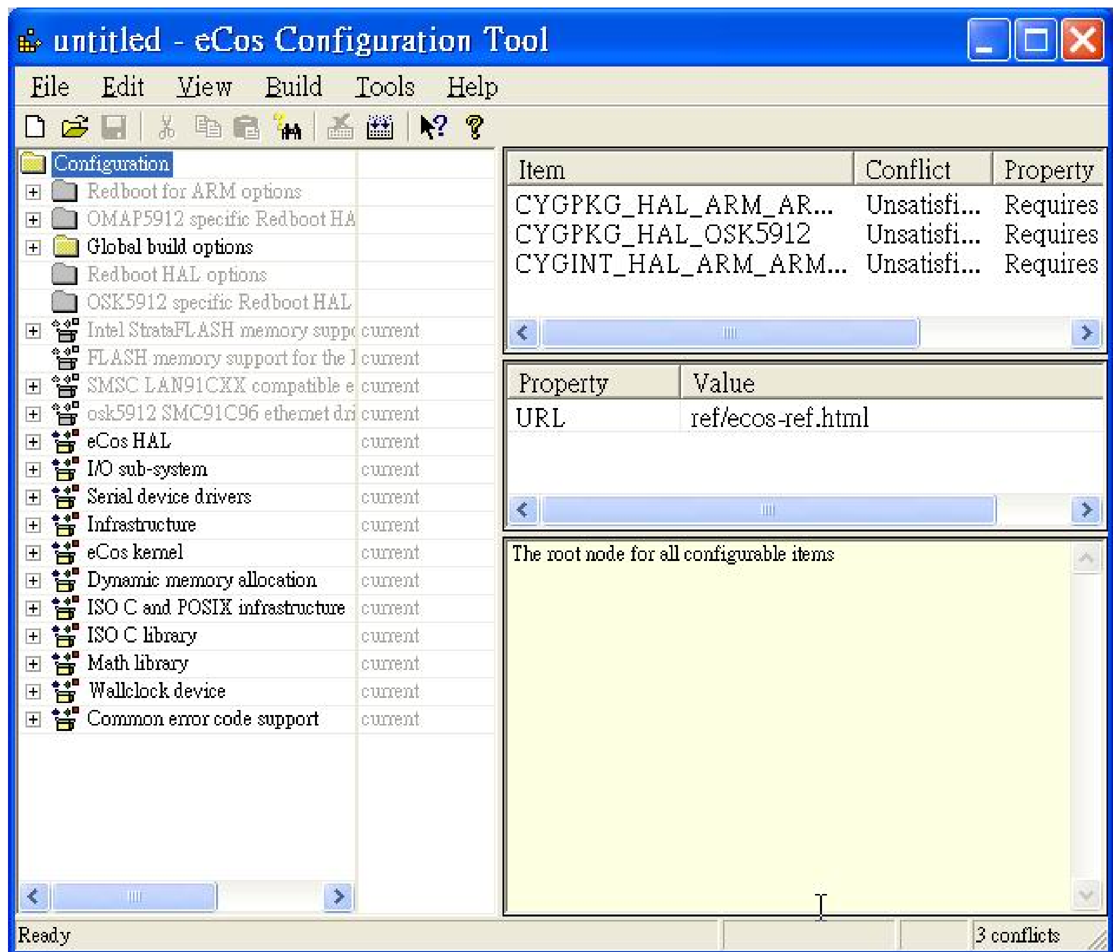


Figure 16. configtool 1

如果 repository 不正確會跳出如下視窗，需填入正確路徑。也可經由 build->repository 啓動此畫面:

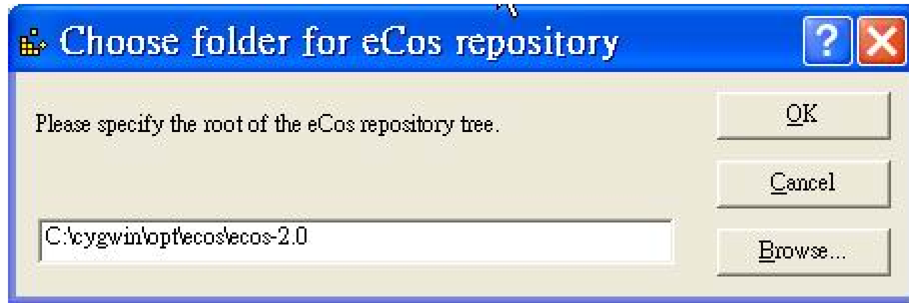


Figure 17. configtool 2

build->template 出現下列畫面，Hardware 部份要選出發展板 OMAP5912 OSK Board，Packages 則選想要產生的目標，例如產生 redboot，就選擇 redboot:

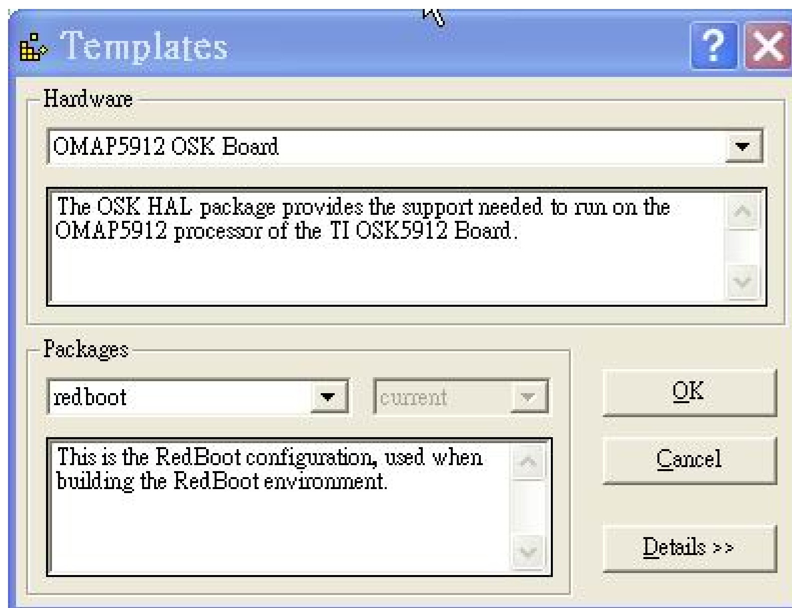


Figure 18. configtool 3

按下 OK 之後出現 Resolve conflicts 自動解決 conflicts 的視窗，按下 continue 即可:

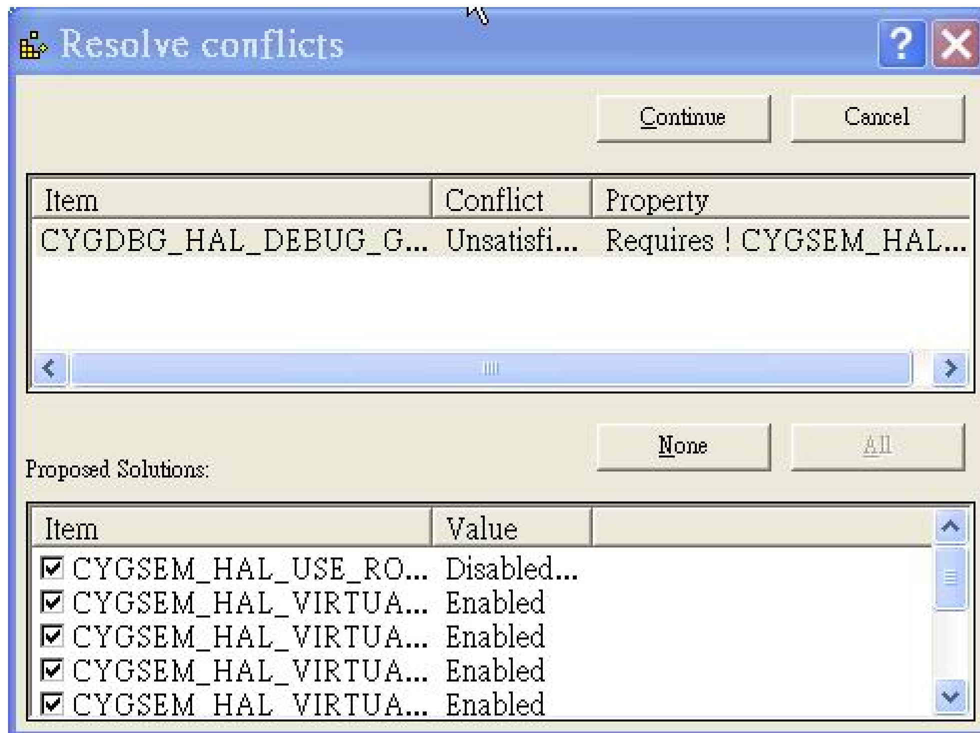


Figure 19. configtool 4

File->Import 選擇使用的.ecm 檔案(記錄想要產生的目標包含哪些 package)，一般平台有 Rom.ecm, Ram.ecm, 以及 Rom-Ram.ecm，主要分別是 bootloader 依附的位置，Rom.ecm 和 Ram.ecm 代表程式放在 Rom 或 Ram 執行，Redboot 是用 Rom.ecm 而 eCos kernel 屬於 Ram.ecm。第三種 Rom-Ram 很少平台使用，這種模式 bootloader 儲存在 Rom，開機後讀到 Ram 執行。接著 File->save 儲存這個工作(.ecc 檔)，之後 Build->library 開始 build 就可以開始產生 redboot.bin 的 image 檔案了。產生之後的 image 檔，會在路徑_install/bin 找到(ex. Redboot_install/bin)。

9.2. 附錄 2: Register 修改

下表爲新增的 register:

Table 25.新增 Register

REG_ARM_CKCTL	0xFFFECE00
REG_ARM_IDLECT3	0xFFFECE24
EMIFF_DLL_WRD_CTRL	0xFFFECC64
EMIFF_MRS_NEW	0xFFFECC70
EMIFF_EMRS1	0xFFFECC78
EMIFF_OP	0xFFFECC80
EMIFF_CMD	0xFFFECC84
EMIFF_DLL_URD_CTRL	0xFFFECCC0
EMIFF_DLL_LRD_CTRL	0xFFFECCCC
I2C_SYSS	0xFFFB3810
I2C_SYSC	0xFFFB3820
MCBSP1_RCERG	0xE1011836
MCBSP1_RCERH	0xE1011838
MCBSP1_XCERG	0XE101183a
MCBSP1_XCERH	0XE101183c
MCBSP1_REV	0XE101183e
MCBSP2_REV	0XFFFB103E
MCBSP3_REV	0XE101703E
MPUI_DSP_MISC_CONFIG	0xFFFEC920
MPUI_ENHANCED_CTL	0xFFFEC924
CLKM_ARM_CKCTL	0xFFFECE00
CLKM_ARM_CKOUT1	0xFFFECE1C
CLKM_ARM_CKOUT2	0xFFFECE20
CLKM_ARM_CKOUT3	0xFFFECE24
MMC_IOSR	0xFFFB7860
MMC_SYSC	0xFFFB7864
MMC_SYSS	0xFFFB7868
SPI_REV	0xFFFB0C00
SPI_SCR	0xFFFB0C10
SPI_SSR	0xFFFB0C14

SPI_ISR	0xFFFB0C18
SPI_IER	0xFFFB0C1C
SPI_SET1	0xFFFB0C24
SPI_SET2	0xFFFB0C28
SPI_CTRL	0xFFFB0C2C
SPI_DSR	0xFFFB0C3C
SPI_TX	0xFFFB0C34
SPI_RX	0xFFFB0C38
SPI_TEST	0xFFFB0C3C
GATE_CONF_REV	0XFFFE1058
USB_TRANSCEIVER_CTRL	0XFFFE1064
LDO_PWRDN_CTRL	0XFFFE1068
FUNC_MUX_CTRL_E	0XFFFE1090
FUNC_MUX_CTRL_F	0XFFFE1094
FUNC_MUX_CTRL_10	0XFFFE1098
FUNC_MUX_CTRL_11	0XFFFE109C
FUNC_MUX_CTRL_12	0XFFFE10A0
PULL_DWN_CTRL_4	0XFFFE10AC
PU_PD_SEL_0	0XFFFE10B4
PU_PD_SEL_1	0XFFFE10B8
PU_PD_SEL_2	0XFFFE10BC
PU_PD_SEL_3	0XFFFE10C0
PU_PD_SEL_4	0XFFFE10C4
FUNC_MUX_DSP_DMA_A	0XFFFE10D0
FUNC_MUX_DSP_DMA_B	0XFFFE10D4
FUNC_MUX_DSP_DMA_C	0XFFFE10D8
FUNC_MUX_DSP_DMA_D	0XFFFE10DC
FUNC_MUX_ARM_DMA_A	0XFFFE10EC
FUNC_MUX_ARM_DMA_B	0XFFFE10F0
FUNC_MUX_ARM_DMA_C	0XFFFE10F4
FUNC_MUX_ARM_DMA_D	0XFFFE10F8
FUNC_MUX_ARM_DMA_E	0XFFFE10FC
FUNC_MUX_ARM_DMA_F	0XFFFE1100
FUNC_MUX_ARM_DMA_G	0XFFFE1104
MOD_CONF_CTRL_1	0XFFFE1110
SECCTRL	0XFFFE1120

CONF_STATUS	0XFFFE1130
MOD_CONF_CTRL_1	0XFFFE1110
RESET_CTRL	0XFFFE1140
MOD_CONF_CTRL_2	0XFFFE1150
IH1_ENHANCED_CNTL	0XFFFECBA0
IH2_STATUS	0XFFFE00A0
IH2_OCP_CFG	0XFFFE00A4
IH2_INTH_REV	0XFFFE00A8
GPIO1_REVISION	0XFFFBE400
GPIO1_SYSCONFIG	0XFFFBE410
GPIO1_SYSSTATUS	0XFFFBE414
GPIO1_IRQSTATUS1	0XFFFBE418
GPIO1_IRQENABLE1	0XFFFBE41C
GPIO1_IRQSTATUS2	0XFFFBE420
GPIO1_IRQENABLE2	0XFFFBE424
GPIO1_WAKEUPENABLE	0XFFFBE428
GPIO1_DATAIN	0XFFFBE42C
GPIO1_DATAOUT	0XFFFBE430
GPIO1_DIRECTION	0XFFFBE434
GPIO1_EDGE_CTRL1	0XFFFBBE438
GPIO1_EDGE_CTRL2	0XFFFBE43C
GPIO1_CLEAR_IRQENABLE1	0XFFFBE49C
GPIO1_CLEAR_IRQENABLE2	0XFFFBE4A4
GPIO1_CLEAR_WAKEUPENA	0XFFFBE4A8
GPIO1_CLEAR_DATAOUT	0XFFFBE4B0
GPIO1_SET_IRQENABLE1	0XFFFBE4DC
GPIO1_SET_IRQENABLE2	0XFFFBE4E4
GPIO1_SET_WAKEUPENA	0XFFFBE4E8
GPIO1_SET_DATAOUT	0XFFFBE4F0
GPIO2_REVISION	0XFFFBEC00
GPIO2_SYSCONFIG	0XFFFBEC10
GPIO2_SYSSTATUS	0XFFFBEC14
GPIO2_IRQSTATUS1	0XFFFBEC18
GPIO2_IRQENABLE1	0XFFFBEC1C
GPIO2_IRQSTATUS2	0XFFFBEC20
GPIO2_IRQENABLE2	0XFFFBEC24

GPIO2_WAKEUPENABLE	0XFFFBEC28
GPIO2_DATAIN	0XFFFBEC2C
GPIO2_DATAOUT	0XFFFBEC30
GPIO2_DIRECTION	0XFFFBEC34
GPIO2_EDGE_CTRL1	0XFFFBEC38
GPIO2_EDGE_CTRL2	0XFFFBEC3C
GPIO2_CLEAR_IRQENABLE1	0XFFFBEC9C
GPIO2_CLEAR_IRQENABLE2	0XFFFBECA4
GPIO2_CLEAR_WAKEUPENA	0XFFFBECA8
GPIO2_CLEAR_DATAOUT	0XFFFBECB0
GPIO2_SET_IRQENABLE1	0XFFFBECDC
GPIO2_SET_IRQENABLE2	0XFFFBECE4
GPIO2_SET_WAKEUPENA	0XFFFBECE8
GPIO2_SET_DATAOUT	0XFFFBECF0
GPIO3_REVISION	0XFFFBB400
GPIO3_SYSCONFIG	0XFFFBB410
GPIO3_SYSSTATUS	0XFFFBB414
GPIO3_IRQSTATUS1	0XFFFBB418
GPIO3_IRQENABLE1	0XFFFBB41C
GPIO3_IRQSTATUS2	0XFFFBB420
GPIO3_IRQENABLE2	0XFFFB424
GPIO3_WAKEUPENABLE	0XFFFBB428
GPIO3_DATAIN	0XFFFBB42C
GPIO3_DATAOUT	0XFFFBB430
GPIO3_DIRECTION	0XFFFBB434
GPIO3_EDGE_CTRL1	0XFFFBB438
GPIO3_EDGE_CTRL2	0XFFFBB43C
GPIO3_CLEAR_IRQENABLE1	0XFFFBB49C
GPIO3_CLEAR_IRQENABLE2	0XFFFBB4A4
GPIO3_CLEAR_WAKEUPENA	0XFFFBB4A8
GPIO3_CLEAR_DATAOUT	0XFFFBB4B0
GPIO3_SET_IRQENABLE1	0XFFFBB4DC
GPIO3_SET_IRQENABLE2	0XFFFBB4DE4
GPIO3_SET_WAKEUPENA	0XFFFBB4E8
GPIO3_SET_DATAOUT	0XFFFBB4F0
GPIO4_REVISION	0XFFFBBC00

GPIO4_SYSCONFIG	0XFFFBBC10
GPIO4_SYSSTATUS	0XFFFBBC14
GPIO4_IRQSTATUS1	0XFFFBBC18
GPIO4_IRQENABLE1	0XFFFBBC1C
GPIO4_IRQSTATUS2	0XFFFBBC20
GPIO4_IRQENABLE2	0XFFFBBC24
GPIO4_WAKEUPENABLE	0XFFFBBC28
GPIO4_DATAIN	0XFFFBBC2C
GPIO4_DATAOUT	0XFFFBBC30
GPIO4_DIRECTION	0XFFFBBC34
GPIO4_EDGE_CTRL1	0XFFFBBC38
GPIO4_EDGE_CTRL2	0XFFFBBC3C
GPIO4_CLEAR_IRQENABLE1	0XFFFBBC9C
GPIO4_CLEAR_IRQENABLE2	0XFFFBBCA4
GPIO4_CLEAR_WAKEUPENA	0XFFFBBCA8
GPIO4_CLEAR_DATAOUT	0XFFFBBCB0
GPIO4_SET_IRQENABLE1	0XFFFBBCDC
GPIO4_SET_IRQENABLE2	0XFFFBBCE4
GPIO4_SET_WAKEUPENA	0XFFFBBCE8
GPIO4_SET_DATAOUT	0XFFFBBCF0

下表爲移除的 register:

Table 26.新增 Register

I2C_IV I2C	0XFFFB380C
EMIFF_SDRAM_CONFIG_2	0XFFFECC3C
RHSW_ARM_CNF3	0XFFFBC880
RHSW_ARM_STA3	0XFFFBC884
RHSW_ARM_STA2	0XFFFBC844
RHSW_ARM_STA1	0XFFFBC804
TEST_DBG_CTRL_0	0XFFFE1070
EP0	0XFFFB4080
I2C_IV I2C	0XFFFB380C

9.3. 附錄 3: Interrupt vector 修正

下表為原有的 interrupt vector 或修改的 interrupt vector:

Table 27.修改 interrupt vector

#define CYGNUM_HAL_INTERRUPT_CAMERA	1
#define CYGNUM_HAL_INTERRUPT_reserved02	2
#define CYGNUM_HAL_INTERRUPT_EXTERNAL_FIQ	3
#define CYGNUM_HAL_INTERRUPT_MCBSP2_TX	4
#define CYGNUM_HAL_INTERRUPT_MCBSP2_RX	5
#define CYGNUM_HAL_INTERRUPT_RTDX	6
#define CYGNUM_HAL_INTERRUPT_DSP_MMU_ABORT	7
#define CYGNUM_HAL_INTERRUPT_HOST_INT	8
#define CYGNUM_HAL_INTERRUPT_ABORT	9
#define CYGNUM_HAL_INTERRUPT_DSP_MAILBOX1	10
#define CYGNUM_HAL_INTERRUPT_DSP_MAILBOX2	11
#define CYGNUM_HAL_INTERRUPT_LCD_LINE	12
#define CYGNUM_HAL_INTERRUPT_reserved13	13
#define CYGNUM_HAL_INTERRUPT_GPIO1	14
#define CYGNUM_HAL_INTERRUPT_UART3	15
#define CYGNUM_HAL_INTERRUPT_TIMER3	16
#define CYGNUM_HAL_INTERRUPT_GPTIMER1	17
#define CYGNUM_HAL_INTERRUPT_GPTIMER2	18
#define CYGNUM_HAL_INTERRUPT_DMA_CH0_6	19
#define CYGNUM_HAL_INTERRUPT_DMA_CH1_7	20
#define CYGNUM_HAL_INTERRUPT_DMA_CH2_8	21
#define CYGNUM_HAL_INTERRUPT_DMA_CH3	22
#define CYGNUM_HAL_INTERRUPT_DMA_CH4	23
#define CYGNUM_HAL_INTERRUPT_DMA_CH5	24
#define CYGNUM_HAL_INTERRUPT_DMA_CH_LCD	25
#define CYGNUM_HAL_INTERRUPT_TIMER1	26
#define CYGNUM_HAL_INTERRUPT_WD_TIMER	27
#define CYGNUM_HAL_INTERRUPT_PERIPHERAL_BRIDGE_PUBLIC	28

#define CYGNUM_HAL_INTERRUPT_reserved29	29
#define CYGNUM_HAL_INTERRUPT_TIMER2	30
#define CYGNUM_HAL_INTERRUPT_LCD_CTRL	31
#define CYGNUM_HAL_INTERRUPT_IH1_IH2_BREAK	32
#define CYGNUM_HAL_INTERRUPT_FAC	32
#define CYGNUM_HAL_INTERRUPT_KEYBOARD	33
#define CYGNUM_HAL_INTERRUPT_MICROWIRE_TX	34
#define CYGNUM_HAL_INTERRUPT_MICROWIRE_RX	35
#define CYGNUM_HAL_INTERRUPT_I2C	36
#define CYGNUM_HAL_INTERRUPT_MPUIO	37
#define CYGNUM_HAL_INTERRUPT_USB_HHC_1	38
#define CYGNUM_HAL_INTERRUPT_USB_HHC_2	39
#define CYGNUM_HAL_INTERRUPT_USB_OTG	40
#define CYGNUM_HAL_INTERRUPT_reserved41	41
#define CYGNUM_HAL_INTERRUPT_MCBSP3_TX	42
#define CYGNUM_HAL_INTERRUPT_MCBSP3_RX	43
#define CYGNUM_HAL_INTERRUPT_MCBSP1_TX	44
#define CYGNUM_HAL_INTERRUPT_MCBSP1_RX	45
#define CYGNUM_HAL_INTERRUPT_UART1	46
#define CYGNUM_HAL_INTERRUPT_UART2	47
#define CYGNUM_HAL_INTERRUPT_MCSI1	48
#define CYGNUM_HAL_INTERRUPT_MCSI2	49
#define CYGNUM_HAL_INTERRUPT_reserved50	50
#define CYGNUM_HAL_INTERRUPT_reserved51	51
#define CYGNUM_HAL_INTERRUPT_USB_FUNCTION_GEN1	52
#define CYGNUM_HAL_INTERRUPT_1WIRE	53
#define CYGNUM_HAL_INTERRUPT_32KHZ_TIMER	54

下表爲新增的 interrupt vector:

Table 28.新增 interrupt vector

#define CYGNUM_HAL_INTERRUPT_OS_TIMER	54
#define CYGNUM_HAL_INTERRUPT_MMC	55
#define CYGNUM_HAL_INTERRUPT_32KUSB	56
#define CYGNUM_HAL_INTERRUPT_RTC_PERIODICAL_TIMER	57
#define CYGNUM_HAL_INTERRUPT_RTC_ALARM	58

#define CYGNUM_HAL_INTERRUPT_reserved59	59
#define CYGNUM_HAL_INTERRUPT_DSP_MMU	60
#define CYGNUM_HAL_INTERRUPT_USB_FUNCTION_ISO	61
#define CYGNUM_HAL_INTERRUPT_USB_FUNCTION_NON_ISO	62
#define CYGNUM_HAL_INTERRUPT_MCBSP2_RX_OVERFLOW	63
#define CYGNUM_HAL_INTERRUPT_reserved64	64
#define CYGNUM_HAL_INTERRUPT_reserved65	65
#define CYGNUM_HAL_INTERRUPT_GPTIMER3	66
#define CYGNUM_HAL_INTERRUPT_GPTIMER4	67
#define CYGNUM_HAL_INTERRUPT_GPTIMER5	68
#define CYGNUM_HAL_INTERRUPT_GPTIMER6	69
#define CYGNUM_HAL_INTERRUPT_GPTIMER7	70
#define CYGNUM_HAL_INTERRUPT_GPTIMER8	71
#define CYGNUM_HAL_INTERRUPT_IRQ1_GPIO2	72
#define CYGNUM_HAL_INTERRUPT_IRQ1_GPIO3	73
#define CYGNUM_HAL_INTERRUPT_MMC2	74
#define CYGNUM_HAL_INTERRUPT_COMPACTFLASH	75
#define CYGNUM_HAL_INTERRUPT_COMMRX	76
#define CYGNUM_HAL_INTERRUPT_COMMTX	77
#define CYGNUM_HAL_INTERRUPT_PERIPHERAL_WAKEUP	78
#define CYGNUM_HAL_INTERRUPT_reserved79	79
#define CYGNUM_HAL_INTERRUPT_IRQ1_GPIO4	80
#define CYGNUM_HAL_INTERRUPT_SPI	81
#define CYGNUM_HAL_INTERRUPT_reserved82	82
#define CYGNUM_HAL_INTERRUPT_reserved83	83
#define CYGNUM_HAL_INTERRUPT_DMA_CH6	84
#define CYGNUM_HAL_INTERRUPT_DMA_CH7	85
#define CYGNUM_HAL_INTERRUPT_DMA_CH8	86
#define CYGNUM_HAL_INTERRUPT_DMA_CH9	87
#define CYGNUM_HAL_INTERRUPT_DMA_CH10	88
#define CYGNUM_HAL_INTERRUPT_DMA_CH11	89
#define CYGNUM_HAL_INTERRUPT_DMA_CH12	90
#define CYGNUM_HAL_INTERRUPT_DMA_CH13	91
#define CYGNUM_HAL_INTERRUPT_DMA_CH14	92
#define CYGNUM_HAL_INTERRUPT_DMA_CH15	93

9.4. 附錄 4: 應用程式測試

以下範例是 eCos 資料夾之中，ecos/ecos2.0/exHMPles 內附的測試程式。

Makefile 裡 INSTALL_DIR=opt/kernel_install，INSTALL_DIR 變數指到 kernel package 的 install 路徑，其中包含 include 以及 lib 路徑。當程式 build 完成，將產生.srec 檔案，藉由 serial port： load －v －m xmodem 再由 terminal 介面下拉選單選擇檔案，可載入檔案到板子上。若由 tftp: load －v －m tftp filename，也可以載入檔案，而且更快速。 載入完成至 default address entry point 0x00020040，鍵入 go，便會執行程式。

下面介紹名程式以及執行結果:

1. hello.c: 最簡單的程範例程式，透過 c library 印出 hello eCos。驗證 serial port 的正確性。

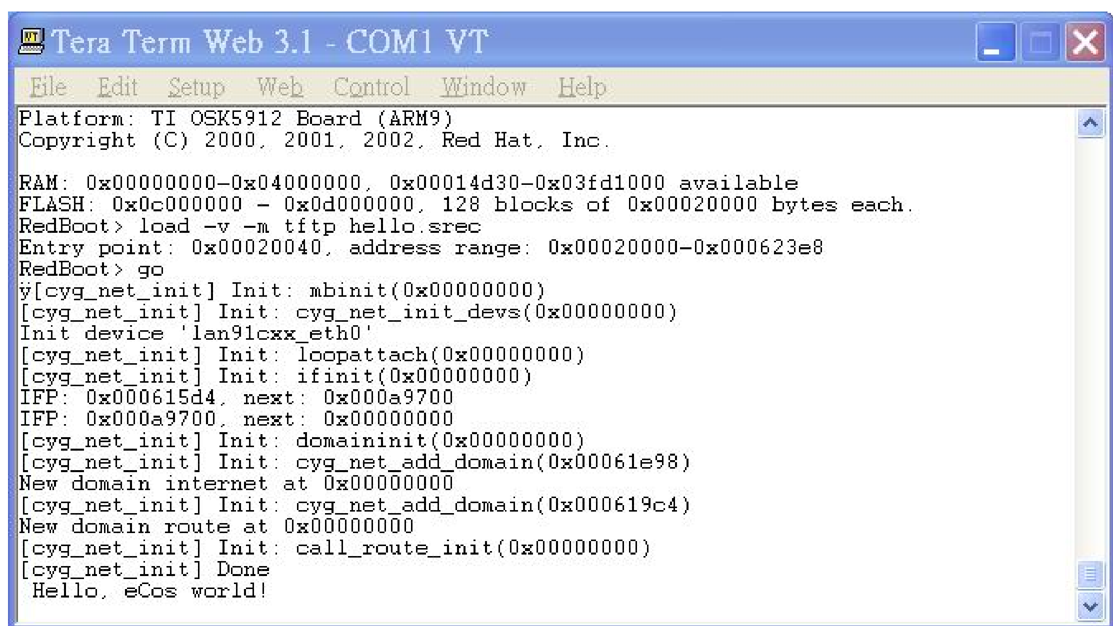


Figure 20. 範例程式: hello.c

2. serial.c: 和上例一樣印出字串，不同的是它建立 thread，在 thread 裡面印出字串。也就是驗證 serial port 和 thread 的使用。

Figure 21. 範例程式: serial.c

3. simple-alarm.c: 一個 alarm 的應用，配合 scheduler，定期印出字串。

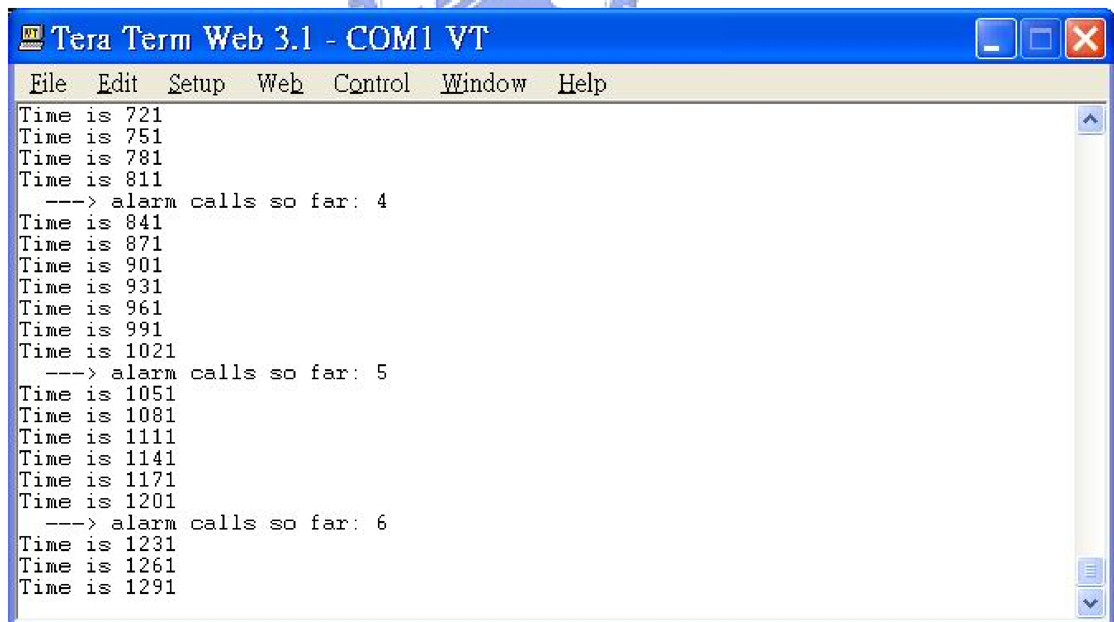


Figure 22. 範例程式: simple-alarm.c

4. twothreads.c: 建立兩個 threads，相同的 priority，加上 mutex 的使用，確保印出字串時，只能由一個 thread 印出自己的 thread data。驗證 thread

及 mutex 的正確性。

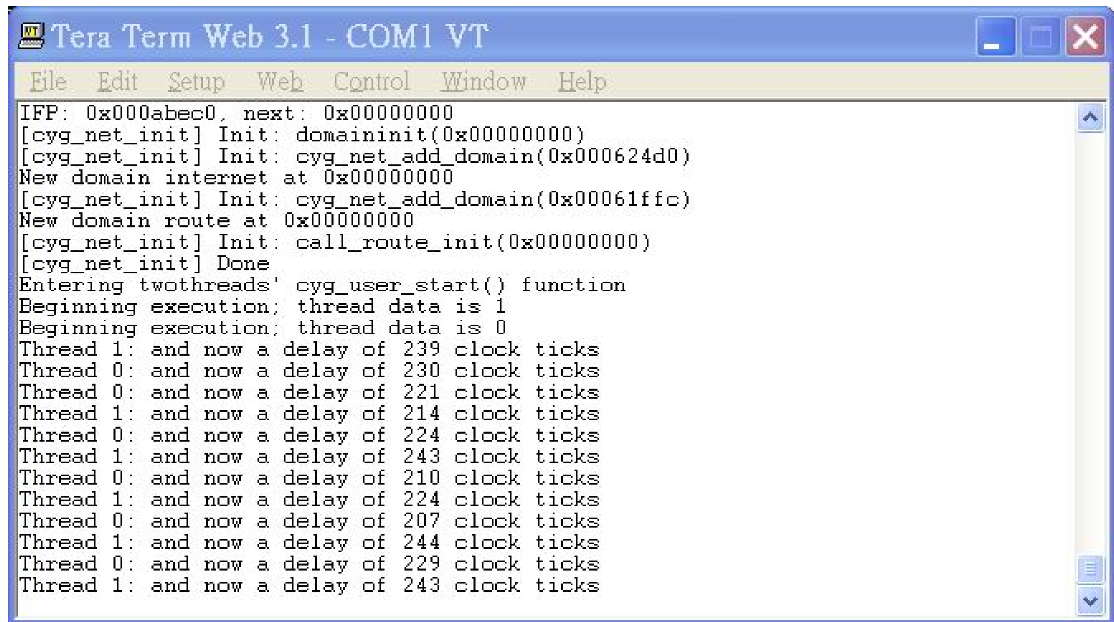


Figure 23. 範例程式: twothreads.c

9.5. 附錄 5: 需修改的系統檔案

9.5.1. Flash

packages\devs\flash\osk5912\current\cdl\flash_osk5912.cdl：記錄 Flash package 的 requirements 和 features。

packages\devs\flash\osk5912\current\include\osk5912_strataflash.inl ： Flash header file。

packages\devs\flash\osk5912\current\srcosk5912_flash.c：定義 Flash properties。

packages\devs\flash\intel\strata 已存在。

9.5.2. Ethernet

packages\devs\eth\arm\osk5912\current\cdl\osk5912_eth_drivers.cdl：記錄 Ethernet package 的 requirements 和 features。

packages\devs\eth\arm\osk5912\current\include\devs_eth_osk5912.inl ： Ethernet header flie。

packages\devs\eth\smsc 已存在。

9.5.3. Serial port

packages\devs\serial\arm\omap5912\current\cdl\ser_omap5912.cdl：記錄 Serial port package 的 requirements 和 features。

packages\devs\serial\arm\omap5912\current\src\omap5912_serial.c： Serial port API。

packages\devs\serial\arm\omap5912\current\src\omap5912_serial.h ： 定 義 register address offset, 及 register status 等。

9.5.4. HAL

packages\hal\arm\arm9\omap5912\current\cdl\hal_arm_arm9_omap5912.cdl ： 記錄 OMAP5912 package 的 requirements 和 features。

packages\hal\arm\arm9\omap5912\current\include\hal_omap5912_setup.h：調整或實作 平始的初使化。

packages\hal\arm\arm9\omap5912\current\include\omap5912.h： OMAP 5912 header file。

packages\hal\arm\arm9\omap5912\current\src\omap5912_diag.c ： diagnostic mode API。

packages\hal\arm\arm9\omap5912\current\src\omap5912_redboot_cmds.c ： 實作 Redboot commands。

packages\hal\arm\arm9\osk5912\current\cdl\hal_osk5912.cdl：記錄 osk5912 package 的 requirements 和 features。

packages\hal\arm\arm9\osk5912\current\include\hal_diag.h： diagnostic mode header file。

packages\hal\arm\arm9\osk5912\current\include\hal_platform_ints.h：定義 interrupt/ exception vector 數量及種類，還有操作的 macro 如 enable/ disable, attach/ detach , 和 mask/ unmask。

packages\hal\arm\arm9\osk5912\current\include\hal_platform_setup.h ：調整或實作平始的初使化。

packages\hal\arm\arm9\osk5912\current\include\plf_io.h：定義 platform I/O macros，I/O registers，及 memory mapping。

packages\hal\arm\arm9\osk5912\current\include\plf_stub.h ： Debugger 介面 GDB

header file。

packages\hal\arm\arm9\osk5912\current\misc\redboot_ROM-net.ecm ： 記錄 build Redboot 所需要的 package。

packages\hal\arm\arm9\osk5912\current\src\osk5912_misc.c： 調整或實作 平台的初使化。

packages\hal\arm\arm9\osk5912\current\src\osk5912_redboot_cmds.c：

實作 Redboot commands。

9.5.5. ARM 926EJS

packages\hal\arm\arm9\var\current\cdl\ hal_arm_arm9.cdl：記錄 ARM9 package 的 requirements 和 features。

packages\hal\arm\arm9\var\current\include\ var_io.h： IO header file。

packages\hal\arm\arm9\var\current\include\ hal_cache.h： instruction cache 和 data cache 設定。

packages\hal\arm\arm9\var\current\src\ arm9_misc.c： ARM926EJS 的初使化。