感測網路作業系統動態元件及自我維護機制之設計與實作
Design and Implementation of a Dynamic Module and Self-Maintaining Mecahnism in Wireless Sensor Network Operating System

馮立琪、姚松男、王昱堯　　沈仲九
長庚大學資訊工程研究所　工研院電通所
lcfeng@mail.cgu.edu.tw

摘要

感測節點的功用在於感測周圍的環境，並對這些感測到的資料做初步的處理，之後透過感測節點形成的感測網路傳遞給資料收集主機。

感測節點被佈置到廣大的環境裡，當感測節點的應用轉變或系統發生異常時，很難回收感測節點來維護。如何讓感測節點擁有自我維護的特性是一個重要的需求。此外目前感測網路作業系統的功能大多稍嫌簡略，可能不足以應付未來許多新興的感測網路應用。

本論文透過 eCos 作業系統的擴充與修改，設計實作出一個擁有自我維護能力的感測網路作業系統。我們的系統主要有下列特色：輕量的動態模組機制、具自我復原及自我更新的功能。藉由實驗證明，本系統已能正確有效的運作，效能並未因新增的自我維護機制而受到影響。

關鍵詞: 感測網路、感測節點、動態模組機制、作業系統、自我維護

1. 序論

感測節點的功用在於感測周圍的環境，並對這些感測到的資料做初步的處理，之後透過感測節點形成的感測網路傳遞給資料收集主機。

感測網路的應用很廣。在環境應用方面，有生物棲息地的監控、環境污染的監控和火災的監控；在軍事應用裡，有戰場的監控、軍隊的辨識和敵方陣地的監察；在醫療應用裡，有病患的健康監控和病患藥品的辨識；在家庭應用裡，有安全的監控和溫度的調節。

感測網路的節點個數可能從數百個至數萬個，並且會被佈置到廣大的環境裡，當感測節點的應用轉變或系統發生異常狀況時，很難回收感測節點來維護。例如鳥類棲息地的感測節點，回收感測節點會驚動到鳥類，影響到監控的效果。如何讓感測節點擁有自我維護的特性是一個重要的需求。

目前感測節點的硬體資源[1]非常有限，多為 8-bit Micro–controller，100KB 的 ROM 和少於 20KB 的 RAM，所以只能處理簡單的應用。而感測網路作業系統為了能在這種硬體資源上跑，也多趨向於簡單的 Event-Driven System，使得作業系統在功能上顯得不足。Adam Dunkels 和 Björn Grönvall 等人在 2004 年的研究結果也認為，在感測網路的應用上 Event-Driven System 太過簡單，應用範圍有限，所以在之上又加了 Preemptive Multi-Thread 的能力[4]。

我們認為未來的感測節點處理能力將會向上提升，硬體技術的進步可以讓許多硬體設備被放進微小的體積裡。許多現有的資訊應用技術也會被放到感測節點上執行，如在 TinyOS 裡就有使用 Virtual Machine 的需求。許多新興的應用如老人的健康照護，可能需要有影像判斷的能力。因此感測網路作業系統的功能也應加強，以符合這些資訊應用的需求。

eCos(embedded Configurable operating system)[10]是一個開放源碼、功能完整、支援多種硬體架構的嵌入式作業系統。因為系統簡潔，支援廣泛且擴充性良好，近年來逐漸受到大家的重視。

在本論文中，我們透過對 eCos 作業系統的修改，設計實作出一個擁有自我維護能力的感測網路

作業系統。我們新增了一個高效能且輕量(light-weight)的動態模組機制，並以此為基礎，達成系統執行中自我復原以及自我更新的能力。透過實驗証明，本系統已能正常的運作，效能亦未因新增的自我維護機制而受到影響。

本篇論文共分為八節，第二節將探討感測節點作業系統與模組機制的相關研究；第三節將介紹 eCos 作業系統；第四節說明模組格式的設計；第五節介紹我們的動態模組機制；第六節為系統整合；第七節為系統評量與比較；最後一節為結論與未來工作。

2. 相關研究

在本節中我們將先簡介幾個比較著名的感測網路作業系統，接著描述目前在感測網路作業系統上軟體更新的技術與其不足之處。

2.1. 感測網路作業系統

TinyOS[2] 是 Berkeley 大學所設計的 Event-Driven System。為了減少 RAM 的使用，TinyOS 的 Process 沒有 Stack，Process 沒有 Context Switch 的機制，也就無法做 Blocking I/O 的動作。Event-Handler 不會被 Blocking，所以執行的時間不能太長，否則會系統會喪失處理其他中斷的時機。為了能處理長時間運算的工作，TinyOS 將需要長時間運算的工作放置於 Job Queue 裡，等到 Event-Handler 處理完畢，再去消耗 Job Queue 裡的工作。

Adam Dunkels 和 Björn Grönvall 等人認為 Event-Driven System 在處理長時間運算的能力不足，所以在 2005 年提出了 Contiki 感測網路作業系統[4]。主要的特色是在 Event-Driven System 之上加入了 Preemptive Multi-Thread 的機制。需要做長時間運作的處理可以被放置在 Preemptive Multi-Thread queue 上。當 Event-Handler 處理完畢後，就會開始執行 Thread。

2.2. 軟體更新技術

TinyOS 對於軟體更換機制採取的是 Script Language 的方式，每一個要更換的軟體會被包裹成膠囊，每一個膠囊擁有 24 個 instruction，透過 Virtual Machine－Maté[3]來執行，而 Maté 會被當作一個 TinyOS 的元件放置在架構的最上層。使用 Virtual Machine 的方法對於縮小軟體的設計原則有很大的幫助。不過最大的缺點在於它的效能低落，而且 Maté 被放置於 TinyOS 的最上層，無法更換 TinyOS 底層的元件。

Chih-Chieh Han 和 Ram Kumar 等人在 2005 年提出的 SOS 感測網路作業系統[7](為 Event-driven System)，可以在系統執行時更換軟體。每一個可以更換的軟體稱為 Module，並且可以在系統執行時期更換。Module 裡含有軟體的 Binary Image，會依據 Module ID 來更換作業系統裡的 Binary Image。缺點在於 Kernel 與 Module 間的函式的呼叫採取間接的方式，使得系統的效能較為低落。

3. eCos 作業系統

eCos 作業系統的完整性和擴充性的優勢讓我們選擇它為我們修改的對象。我們將於本節描述 eCos 所使用的元件管理機制與不足之處。最後再解說 eCos 的 I/O 與中斷機制以作為動態元件管理機制之設計考量。

3.1. eCos 的元件管理機制

eCos(embedded Configurable operating system)[10]為一個開放源碼且適合於嵌入式環境的作業系統， eCos 本身沒有動態的模組機制，但提供了劃分清楚、可重新組態的靜態元件，每一個元件稱為 Package。

eCos 將各種功能與套件分離為各 Package，透過組態配置工具建立出自己想要的組態設定檔，再使用組態配置工具建立出這些 Package 的 Makefile。之後利用這些 Makefile 來編譯 Package，製造出 eCos 的核心映像檔。

3.2. 動態元件機制

eCos Package 的觀念只能應用在 Compile Time 時使用，對於感測網路作業系統自我維護的應用並不足夠。要能達到自我維護的應用，感測網路作業系統必須有動態模組的機制，以致於能在 Run-Time 時個別的維護各個元件。

因此我們將在 eCos 上設計實作這些機制，使 eCos 更符合感測網路作業系統之需求。

3.3. eCos 的 I/O Sub-System

為了便利後面系統設計上的討論，我們在此小節中將介紹 eCos 的 I/O 及中斷機制。

eCos 的 I/O Control System 控制著所有 I/O 的處理，分為兩個部份，I/O Sub System 和 Device Drivers。

Device Driver 控制它之下的硬體，I/O Sub-System 則提供一套介面，讓 application 能透過 Device 來傳輸資料。每一個 Device Driver 必須在 Device Table 裡註冊一個 *DEVTAB_ENTRY*，此 Entry 的結構如圖 1。

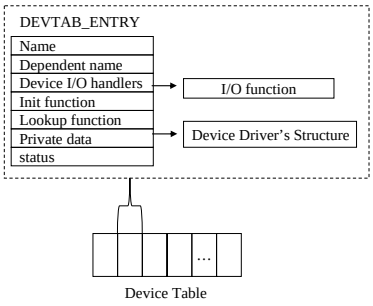

圖 1. eCos 的 Device Table 和 *DEVTAB_ENTRY* 結構

I/O Sub-System 會提供一組 API 讓 Application Level 使用並與底層的 Device Driver 傳輸資料。

當 AP 要使用 Device 時，會以 Device 的位置名稱來呼叫 Open Function 取得 File Descriptor。之後使用此 File Descriptor 來呼叫 *read()*或 *write()*函式來讀取或傳送資料。

File Descriptor 可以找到 Device Driver 註冊時提供的 *DEVTAB_ENTRY*。之後藉由 *DEVTAB_ENTRY* 與底層的硬體做溝通，完成資料傳輸的動作。

3.4. eCos 的中斷機制

eCos 的中斷處理分為 ISR、DSR 和 Deliver Thread。ISR 的特性就是處理時間很快，主要考慮的因素是不想霸佔中斷處理的時間太久以免喪失服務其他中斷的時機。而需要長時間處理的中斷程序則可以放到 DSR 裡或是 Deliver Thread 處理。

DSR 處理的時機為 Scheduler 做 Context Switch 之前，而 Deliver Thread 跟系統的 Thread 一樣，不過執行的時機得等待 DSR 的喚醒之後，透過 Scheduler 來排程。

4. 模組執行檔格式(LW-ELF)

為了不以回收的方式更換感測節點上的模組，所以必須過感測節點的無線網路設備來做更換的動作。所以模組的大小與耗電量息息相關。在本節中，我們先簡述 ELF 格式，再解說我們設計的 LW-ELF 格式。

4.1. ELF 格式

大部分的開放原始碼的作業系統都採用 ELF 格式，所以我們針對此種格式進行分析，以裁減出 Light-Weight 的格式。ELF 的格式如圖 2：

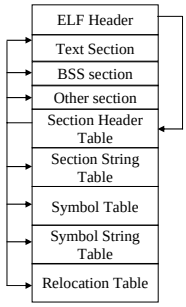

圖 2. ELF 格式

Object Code 位於 ELF Header 和 Section Header Table 之間，此段資料最終會被載入到作業系統裡，所以無法做裁減。以下針對其他資料做分析：

ELF Header 包含 ELF 格式的版本資訊、放置 Object Code 的位置、Program Header 和 Section Header Table 的位置。Section Header Table 裡存有

每一個 section 的 header 來說明 section 的資訊。主要有此 section 的名稱、位置、種類、特性、大小以及此 section 在 Object File 裡的位置。

Symbol Table 儲存著 symbol entry，每一個 symbol entry 記載著名稱位置、symbol 的值以及此 symbol 位在哪一個 section。Relocation Table 裡存著 relocation entry，entry 裡記載著 machine Code 的位置，以及它對應到的 symbol 之 index。

4.2. Light-Weight ELF 格式

我們針對 Section、Relocation 和 Symbol 的資料提出裁減的方法，並在最後提出完整的 Light-Weight ELF 格式。

4.2.1. Section 資料的裁減方法

Section 資料包括 Section Header Table 和 Section String Table。Section Header Table 可以找出 Object Code 裡的 Section 資料、Section String Table、Symbol Table、Symbol String Table 和 Relocation Table。

為了裁減 Section 資料，我們將 Object Code 裡的 Section 視為同一個區塊，這樣可以減少 Section Header Entry 和 Section String Table 使用。如圖 3 所示，將 Text Section、BSS Section 以及其他會用到的 Section 放到 LW-ELF 的 Object Code 區塊裡。

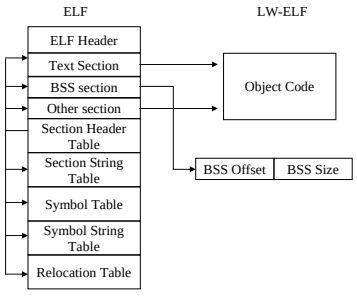

圖 3. Section 資料的裁減方法

4.2.2. Relocation 的裁減方法

此種資料可以分成四種形式。分別是 Defined 的 PC、ABS 和 Undefined 的 PC、ABS。Defined 的 PC 和 ABS 表示這些 Machine Code 用到 Object Code 裡的資料，可以在 Object Code 裡找到 Target Address，而 Undefined 的 PC 和 ABS 表示這些 Object Code 裡的 Machine Code 用到資料沒有定義在 Object Code，得透過作業系統的 Linker 來得知 Target Address。

以下為 LW_ELF 使用的裁減方法：

- Defined PC

一般的 Branch Machine Code 所放的 Operand 為 Target Section 為起始的 Offset。此 Section Based Offset 需要藉由 Relocation Entry 以及 Symbol Entry 才能得知 Object Code Based Offset。

對於此種類的資料，我們算出 Object Code Based Offset，並更新到 Branch Machine Code 裡。以減少 Relocation Entry 的數量、Symbol 的數量和 Relocate 的時間。

- Defined ABS

Defined ABS 的位置裡會存著以 Target Section 為起始的 Offset，將 Object Code 載入作業系統裡的記憶體時，必須依據 Relocate 的資料算出絕對記憶體位置。我們計算出以 Object Code 為起始的 Offset，並紀錄於 ABS Defined Relocate Entry 中。這樣可以節省掉尋找 Symbol Table 的時間和 Symbol Table 所佔的空間。

- Undefined PC

Undefined PC 代表 Target 為 Undefined 的 Symbol，必須依賴作業系統以此 Symbol 的名稱來找尋此 Symbol 的記憶體位置，此種資料通常為一個 Function 的記憶體位置，所以底下的敘述會直接以 Undefined Function 稱之。

為了減少 Symbol 所佔的空間，我們以 ID 的方式來代替 Symbol String。如圖 4 所示，A 要 Branch 到作業系統裡的 B 的位置，我們以 Fun Undefined Relocate Entry 來紀錄 A 的 Relocate 資料，以 Fun Undefined ID Entry 來記錄 B 的資料。

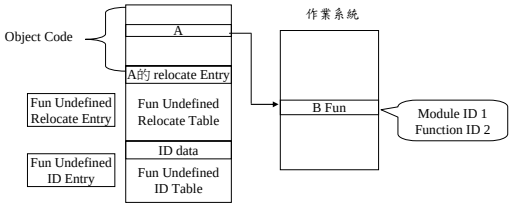

圖 4. Undefined Function 的裁減方式

Fun Undefined Relocate Entry 的結構記載著 A 與以 Object Code 為起始的 Offset 和存在於 Fun Undefined ID Table 的 Target 資料的 Index。

Fun Undefined ID Entry 的結構記載著此 Symbol 的 Fun ID 和所屬的 Module ID。

- Undefined ABS

Undefined ABS 相對於 Undefined Function，我們使用 GV Undefined Relocate Entry 與 GV Undefined ID Entry 紀錄 Undefined ABS 所需要的資料。

- Module export 的 Function

Module 會提供給作業系統呼叫的 Function，如 Module 的初始化函式，必須讓作業系統知道。我們以 Fun Reg. Entry 來提供此種資訊。

- Module export 的 GV(Global Variable)

Module 會提供給作業系統使用的 Global Variable，我們以 GV Reg. Entry 來提供此種資訊。

最後裁減過後的 ELF 格式如下圖：

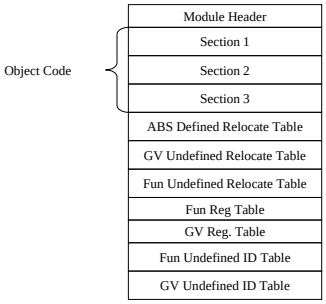

圖 5. LW-ELF 格式

4.3. 自動格式轉換工具

我們實作出兩個工具，一個是將 ELF 格式轉換成 LW-ELF 的轉換工具，另一個是讀取 LW-ELF 的工具來得知此格式裡的資料。

轉換工具裡的流程如圖 6。一開始將 ELF 檔案讀到 buffer 裡，並解析 Header 裡的資料來得知各個資料。之後讀取作業系統的 Fun ID Table 和 GV ID Table，然後裁減 Section、Relocation 和 Symbol 資料，並轉換 Relocation 的資料成我們訂製出的格式，最後產生出 LW-ELF 格式。

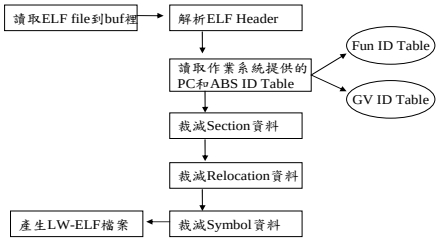

圖 6. 轉換工具的處理流程

讀取工具的流程如圖 7，主要的目的是檢視產生出的 LW-ELF 格式是否符合我們的需求。工具一開始會讀取 LW-ELF File 到 buffer 裡，然後解析 LW-ELF Header 來找出其他資訊的位置。最後印出 LW-ELF Header、Object Code、Relocation、Fun Reg 和 GV Reg 的資料。

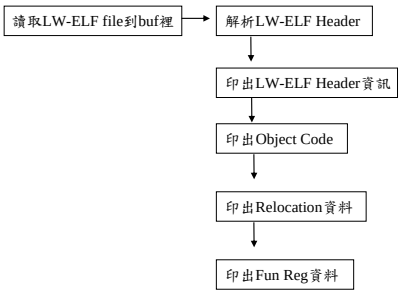

圖 7. 讀取工具的處理流程

5. 動態 Module 機制之設計

我們於本節中先描述動態 Module 機制之設計方式，再解釋 Module 如何做到自我復原與更新。

5.1. Module Manger 之設計

在我們的系統中，Module 可以為 Device Driver 或 AP，會透過 Module Manager 來載入 eCos 裡。圖 8 為我們系統的架構圖。

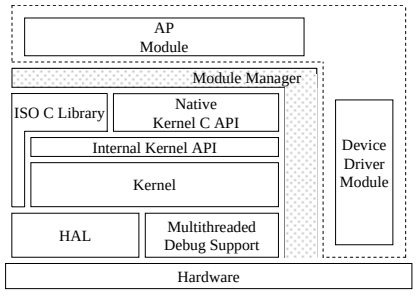

圖 8. 系統架構圖

底下描述 Module Manager 之設計方式與重要資料結構，最後再解說 Module Manager 之初始化流程。

5.1.1. Module 結構

Module 的資訊會以 Module 結構來記錄，如圖 9 所示。為了能迅速的找到 Module 結構，所以使用 Array 的方式來儲存，再透過 Module ID 做搜尋。為了讓 Module 能透過此機制來與 Kernel 溝通，所以我們設計 kernel 的 Module ID 為 0，其他 Module ID 則交由系統開發人員來指定。下面對每一個欄位做解釋：

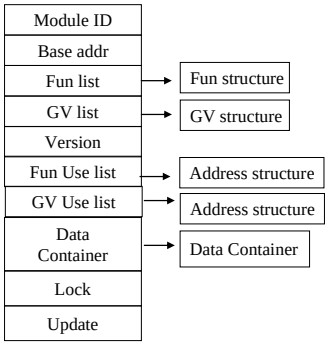

圖 9. Module 結構

Module ID 存著 Module 的 ID，為尋找到此 Module 結構的依據。Base Address 存著此 Module 載入到 eCos 後 Object Code 的起始位置。Fun List 指向 *Fun* 結構的指標，存著此 Module 釋放出去的 Function 資料，會使用 Linking List 的方式串連起這些 Fun 結構。GV List 相對於 Fun List，存著此 Module 釋放出去的 Global Variable 資料。

Version 欄位存著此 Module 的版本，為更換此 Module 的依據。Fun Use List 欄位指向 *Address* 結構，存著此 Module 會使用到的 Function 資料。GV Use List 意義同於 Function Use List 差別在於其內存著此 Module 會使用到的 Global Variable 資料。

Data_container 結構之目的在存著此 Module 欲保留的資料，當自我更新或自我初始化處理時，會使用此資料結構來找到先前被保留的資料。Data Container 欄位則為指向 *Data_container* 結構的指標。

Lock 是在自我更新或自我復原時使用的同步機制。Update 的作用在於告知 Kernel 此 Module 有做更新或復原動作。*Fun* 結構、Address 結構、Data Container 結構會於下一小節再加詳述。

5.1.2. *Fun*、*Address*、*Data_Container* 之結構

圖 10 為 *Fun* 結構，主要的用途是記錄此 Module 釋放出去的 Function 資料。ID 欄位記載著此 Function 的 ID；Base address 記載著此 Function 所在的記憶體位置；Be Used list 記載著一串 address 結構的 Linking List，主要是記載呼叫此 Function 的記憶體位置；next 欄位會記載著下一個 Fun 結構。

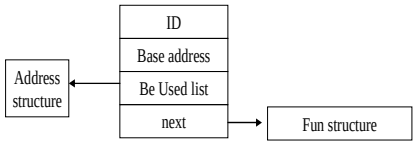

圖 10. *Fun* 結構

圖 11 為 *Address* 結構，主要是記載記憶體位置的資訊，提供給 Fun Use list 和 *Fun* 結構記錄記憶體位置資訊。

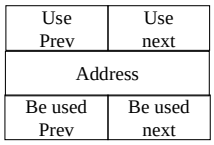

圖 11. *Address* 結構

圖 12 為 *Data_Container* 結構，主要的功用是紀錄 Module 使用到的資料，它所使用的記憶體空間由 Module Manager 負責分配和管理，所以在 Module 自我更新或是復原時，不會影響到這些資料。藉由此一機制，系統可以順利取得並回復到先前的執行環境而不會影響到系統的運作。

ID 欄位記載著此 *Data_Container* 結構的 ID；Base address 記載著由 Module 機制配出的記憶體位置；next 會指向下一個 *Data_Container* 結構。

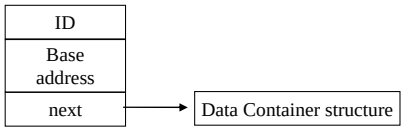

圖 12. *Data_Container* 結構

5.2. Module Manager 的初始化流程

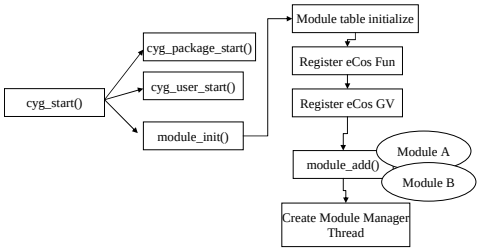

圖 13. Module Manager 初始化的流程圖

圖 13 為 Module Manager 初始化的流程圖，觸發的時機為 eCos 初始化的時候。*module_init()*負責初始化整個 Module Manager。各個細節會在以下說明：

(1) Module table initialize

此階段會初始化每一個在 Table 裡的 Module 結構。

(2) Register eCos Function、Register eCos Global Variable

將 eCos 所提供的 Function 和 Global Variable 資料作註冊的動作，以提供給 Module 使用。eCos 會被視為 Module ID 0，所以註冊的 Function 和 Global Variable 資料會註冊在 Module ID 為 0 的 Module 結構裡。

(3) 呼叫 *module_add()*

此階段會新增 Module 進系統裡。一開始會分析 LW-ELF 檔案來得知此 Module 的 ID、Version、其他資料位在此檔案的位置和 Object Code 的大小並拷貝 Object Code 至配置好的記憶體空間裡。依據 LW-ELF 格式裡的 ABS Defined、Fun Undefined 和 GV Undefined 資料找到 Target 的記憶體位置來更新 Object Code，讓 Kernel 和 Module 之間的函式呼叫與全域變數之使用，藉由存取 Target Address 的方式，達到直接的效果，以加快呼叫的速度。之後註冊此 Module 所提供的 Function 和 Global Variable 資料。最後執行 Module 的 init Function 來初始化 Module。

(4) Create Module Manager Thread

Module Manager Thread 主要的功用是接收 Module，並判斷是否必須更新 Module，圖 14 為此 Module Thread 的流程圖。當接收到完整的 LW-ELF 檔案，會進到 process Module 的階段。一開始會依據 LW-ELF 檔案的 Module ID 來判斷是否如要做新增 Module 或是更新 Module 的動作。如果為新增 Module，則依據上一小節的 *module_add()*函式新增一個 Module。如果為更新 Module，則判斷 Module Version 來決定是否要更新，更新 Module 的詳細細節會在下一節裡解說。

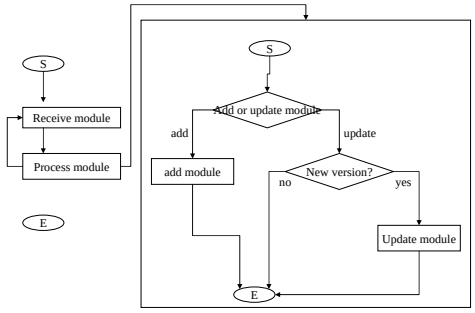

圖 14 Module Manager Thread 的流程

5.3. Module 自我復原及更新

我們所設計的自我維護功能包含二個部份: 自我復原及自我更新。

Module自我復原的時機是在系統發現到可能有的錯誤發生時所進行的初步處理程序，主要的作法是針對該Module進行局部的自我的初始化動作，以快速的回到正常的工作狀態。近來有些研究顯示，此一方式常可快速有效的解決一些暫時性的錯誤(transient failures) [12]。

Module 自我更新的時機會在自我復原無法解決錯誤和當 Module Manager Thread 收到新版的 Module 時，其流程如圖 15 所示。Module Manager Thread 每隔十分鐘會去檢查是否有接收到新版的 Module，當接收到新版的 Module 會判斷是否可以更新。判斷的依據是接收的 LW-ELF 檔案的 Module id 和 version，如果系統內此 Module 的 version 比接收進來的 Module 舊，就會進入更新的程序。

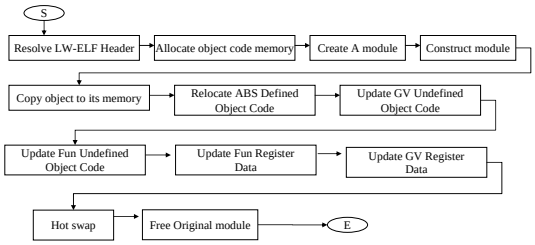

圖 15. Module 自我更新的流程

Module 自我更新的流程一開始跟 *module_add()*函式的流程一樣，會解析 LW-ELF Header 並配置 Object Code 的記憶體空間。之後會建立一個新的 Module 結構，並將之前 Module 結構的資料移至新的 Module 結構裡。接下來會將 Object Code 放置到已配置好的記憶體空間裡，並處理 Relocation 的動作。

此部分的核心問題是 Module 更新或復原之後如何還原當初的執行環境。Module 裡的 Object Code 隨時有被更新或復原的可能，所以 Module 用到資料不能放在 Object Code 裡，而是必須放在我們設計實作的 *Data_Container* 結構裡，如圖 16 所示。

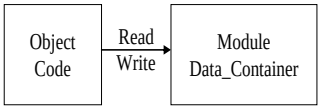

圖 16 Object Code 與 Module *Data_Container* 的關係

自我更新與自我復原流程中最重要的部分，就是 Hot Swap 的動作。在系統運作時將之前舊 Module 的環境移植到新的 Module 裡，並用新 Module 來取代舊 Module。在此步驟必須處理同步，會運用 Module 結構裡的 lock 來控制只有一方能使用此 Module。

舊的 Module 已被 Kernel 或其他 Module 所使用，Module thread 要更換舊 Module 時，必須等待 Kernel 或其他 Module 釋放此 Module 的 lock 才能更新；或是 Module 在更新時，Kernel 或其他 Module 得等待 Module thread 釋放此 Module 的 lock 才能使用此 Module。

當 Hot Swap 處理完畢，則將 Module 結構裡的 update 欄位設為 1，告知 Kernel 使用此 Module 時，必須執行系統提供的 reload Function 讀取在 *Data_container* 裡的資料，以完成資料更新的動作。最後釋放舊 Module 使用的記憶體空間，整個自我更新機制到此結束。

6. 系統整合

在這一節裡我們簡介我們系統所使用的硬體平台，說明如何把 Device Driver 轉換成 Module，以及 eCos 與 Device Driver 必需考量的同步機制。並解說我們如何撰寫一個精簡的 Shell。最後，在描述我們目前所使用的網路通訊協定與移植方式。

6.1. SCAN II 硬體平台簡介

SCAN II 為工研院所研發的感測節點，圖 17 為此硬體的架構。此硬體使用 Hynix 公司[15]所出的 SoC 晶片，型號為 7202，使用的 processor 為 ARM 720T。SCAN II 週邊的設備有 ZigBee 無線網路和感測裝置。

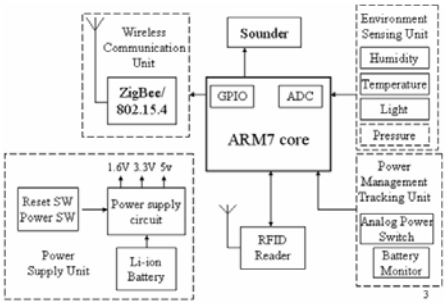

圖 17 SCAN II 的硬體架構

6.2. Device Driver Module 的實做

我們將 eCos 的 Serial port Driver 改寫為 Module，來證實系統能成功的運作。

6.2.1. Serial Port Driver 與 eCos 的關係

Device Driver 必須提供四種資訊，分別是 *DEVTAB_ENTRY*、ISR、DSR 和 Private Data，Private data 的用途在於讓 Kerenl 在處理 DSR 時，能找到此 Device 的設定資料，對於 Serial Port 而言是 Serial Channel 結構。Serial Channle 結構如圖 18 所示。

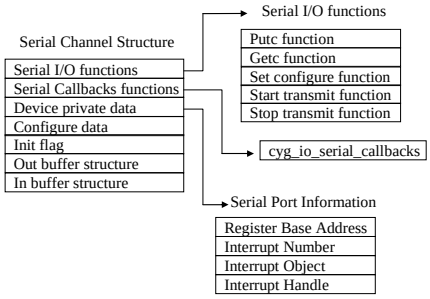

圖 18　Serial Channel 結構

Serial Channel 主要的資料有 Serial Port 提供的 I/O Functions、callbacks Functions、private data、configure data 還有 out、in buffer。其中 private data 存有 Serial port information 結構，記載著 Serial port 暫存器的位置和 Interrupt number、Object、Handle。Serial Channel 結構會記載於 *DEVTAB_ENTRY* 裡的 Private Data。

Serial Port Driver 必須被保留的資料有 Serial Channel 結構和其結構裡的 Serial Port Information。我們將資料保存於我們設計的資料保存機制－*Data_Container* 中，並把此記憶體空間記載在 ID 為 0 的 *Data_Container* 結構裡。

6.2.2. Kernel 與 Serial Port Driver Module 同步的方法

同步的目的在於讓 Module Manager 更新 Module 時，不會干擾到 eCos 使用 Module 的行為。當 AP 要使用 Serial Port 時，會以 Serial Port 的位置名稱－/dev/ser0 來呼叫 Open Functions 產生 *cyg_file* 結構。之後使用此結構呼叫 Read 或 Write Functions 並藉由 Serial Port Driver 所提供的 *DEVTAB_ENTRY* 來讀取或傳送資料。所以在此點必須新增同步處理以避免 Module Manager 更換此 Module 時，影響到 eCos 的運作行為。

當 AP 使用 Module 時會取得 lock，如果 Module 機制在處理更新時取得此 Module 的 lock，此 AP 就會進入睡眠狀態等待 Module 機制釋 lock。在 AP 取得 Module 的 lock 時，會去檢查 Module 結構裡的 update 資料。如果為 1，表示 Module Manager 已經更新過 Module，Kernel 就必須更新 *cyg_file* 結構裡的 File Private Data，也就是新 Module 所註冊的 *DEVTAB_ENTRY*。

6.3.精簡的 Shell

精簡的 Shell 主要的原理是使用虛擬的 Terminal Device 來達到終端機畫面的呈現，以及使用 Read Function 來接收使用者所輸入的字串和使用 Write Function 來輸出字串，底層的溝通媒介則會透過 Serial Port Device。所以虛擬的 Terminal Device 會使用 Serial Port Driver Module 來控制 Serial Port Device 上的資料傳輸。

當使用者輸入字串按 Enter 鍵後，Shell 會做字串的解析，如果結果符合某一個指令，則會去呼叫那一個指令的 Function。

6.4. 移植 IEEE 802.15.4/Zigbee 通訊協定

本系統使用 IEEE 802.15.4/Zigbee 作為感測網路的傳輸協定。IEEE 802.15.4/Zigbee 主要是應用在一個電源必須持久但 throughput 不高的設備上，可以使其在一個很簡單且花費甚低的環境下使用無線網路與其他設備溝通。主要的幾個優點為：易於安裝、可靠的資料傳送、短距離的資料傳輸、非常低的花費及合理的電力持久性。

本實驗室已經與工研院合作完成 IEEE 802.15.4/Zigbee 的實做[21]，但之前完成的版本是執行於 Linux 之上，必需將其移植到 eCos 上。移植的步驟大致如下：(1)調整硬體的 Memory Mapping，使 eCos 的 Memory Mapping 能對應到正確的硬體裝置。(2)將 IEEE 802.15.4/Zigbee 中所使用的 Linux 函式，全部轉為呼叫 eCos 的對應函式。(3)在 eCos 上設計簡化的 Linux Bottom Half 機制。由於我們實做的 IEEE 802.15.4/Zigbee 中大量使用 Linux 的 Bottom Half 機制，所以必須在 eCos 上藉由 DSR 設計類似於 Linux Bottom Half 的機制，以完成 IEEE 802.15.4/Zigbee 所需的工作。

目前已大致完成 MAC 與 PHY 層的移植工作。後續會繼續完成 Network 層(Zigbee)的部份。

7. 系統評量與比較

我們將在第一小節解說我們如何在 SCAN II 硬體平台上做測試的工作，在第二節小節裡描述我們設計的實驗和實驗結果。

7.1. 測試方法

SCAN II 上的 ZigBee 協定實做是本實驗室的一個計畫，因為 ZigBee 通信協定堆疊在 eCos 上的開發尚未完成，所以我們無法使用無線網路裝置進行模組傳送的測試。我們假設 LW-ELF 檔案已經存在於 SCAN II 的記憶體上，藉由讓 Module Manager 知道其記憶體位置的方式進行實驗測試。

我們將 Multi-ICE 設備連接至 SCAN II 上，並使用 ADS 工具將 eCos 的 Image、Serail Port Driver Module 放在記憶體上，並執行 eCos Image。

當系統初始化時， Module Manager 會讀取放在記憶體上格式為 LW-ELF 的 Serial Port Driver Module，並新增此 Module 進 Module Manager 裡，之後會製造出 Module Manager Thread 來負責 Module 更新的處理。最後，會製造出執行精簡 Shell 的 Thread 來與使用者互動。

7.2. 實驗設計與實驗結果

我們先比較 LW-ELF 與 ELF 的大小，來證實我們的裁減方式可以有效的縮減 ELF 的大小。接著則是自我復原與自我更新機制的實驗。最後將比較我們的系統與其他感測網路作業系統。

7.2.1. LW-ELF 與 ELF 的大小比較

裁減 ELF 主要的目的是減少 Module 的大小，希望能在無線網路裝置傳送時減少耗電量。我們會比較 Serial Port Driver Module 的 ELF 與 LW-ELF 的大小，比較的結果在表 1。從表 1 裡得知，原先 ELF 的 Serial Port Driver Module 為 5.3K，而經過我們轉換工具轉換出的 LW-ELF 為 2.4K，大小縮減了原先 ELF 的 45%。而精簡 Shell 也能縮減至原先 ELF 的 47%。顯示 LW-ELF 有效低減少了 ELF 的大小，有助於無線網路裝置的省電。

表 1. ELF 與 LW-ELF 的大小比較

應用	ELF	LW-ELF
Serial Port Driver	5.3K	2.4K
精簡Shell	2.3K	1.1K

7.2.2. Module 自我復原

Module 自我復原的時機會在系統發現可能有錯誤時啟動，我們會測試自我復原的處理是否正常? 以及對系統運作效能產生的影響。我們測試的方法是在傳檔的過程中，啟動自我復原的機制，然後觀察傳檔的時間與沒有 Module 機制的 eCos 的差別，也查看檔案的傳送是否正確。

使用的測試程式為我們所撰寫的 X Modem 應用程式來做資料傳輸，底層會使用 Serial Port Driver。Host 端會使用 X Modem 的資料傳輸程式並透過 Serial Port 傳輸 200K 的資料到感測節點上，而感測節點則會執行 X Modem 的應用程式來接收資料。

對照組為未修改過且沒有 Module 機制的 eCos，而實驗組為擁有 Module 機制和 Serial Port Driver Module 的 eCos。

因為系統目前並無自我檢測的功能，實驗中我們設計實驗組在傳輸資料十秒後，啟動自我復原的程序。圖 19 為實驗十次的平均結果，從實驗數據得知，對照組會在 40.02 秒接收完 200K 的資料，而實驗組同樣會在 40.02 秒接收完 200K 的資料，且資料正確的接收。會有這種結果是因為自我復原機制的設計極為精簡，使得系統的 Overhead 極低。

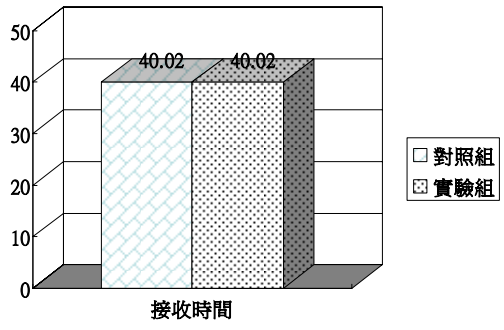

圖 19. Module 自我復原的系統效能測試

7.2.3. Module 自我更新

Module 自我更新的時機會在自我復原無法解決錯誤和 Module Manager Thread 每隔十分鐘接收到新版的 Module 時，底下會針對 Module Manager Thread 接收新版的 Module 的應用來實驗。

為了證明我們所實做的 Module 自我更新能運作，我們將利用我們寫的精簡 Shell 進行測試。在精簡 Shell 運作時，我們將更換 Serial Port Driver Module 來證明 Module 自我更新能正確的回復之前的環境。

十分鐘過後 Module Manager Thread 接收到新版的 Module，Module Manager 更新 Serial Port Driver，精簡 Shell 還能夠正常的運作。證實了 Module 自我更新機制的正確性。

為了瞭解 Module 自我更新所花的時間，我們在自我更新前印出當時系統的時間，以及在自我更新後印出當時系統的時間，兩個時間相減則是 Module 自我更新的時間。計算的結果為 0.01 秒。

7.2.4. 與其他感測網路作業系統之比較

由於無法找到共通的平臺，將各個感測網路作業系統進行實測的比較，我們只能進行靜態的功能性比對。

表 2 為比較我們的系統與 SOS 和 TinyOS 的差異。SOS 與 TinyOS 皆為 Event-Driven System，對於未來的應用稍有不足。TinyOS 的 Virtual Machine 機制有無法更新系統元件的缺點，我們的系統則可以做到更換系統元件的功能。SOS 雖然可以更換系統元件，但是 Module 與 Kernel 之間的函式呼叫是間接的，造成了一些 Overhead，我們的系統則為直接呼叫，效能較佳。最後本系統有做出自我復原的功能，這是 SOS 與 TinyOS 所缺乏的。

表 2. 本系統於其他感測網路作業系統比較

	我們的系統	SOS	TinyOS
架構	Multi-Threading	Event-Driven	Event-Driven
軟體更新機制	Module	Module	Script Language (Mate)
更換系統元件	Yes	Yes	No
Module與Kernel之間函式呼叫的關係	直接	間接	None
自我復原功能	Yes	No	No

8. 結論與未來工作

感測網路在近幾年來蓬勃發展，應用的領域也越來越多，在未來的人類社會裡，將扮演幫助人類感測環境的重要助手。由於感測網路的特性，感測節點被佈置在廣大的環境裡，使得維護工作的困難度增加。

本論文針對此問題提出一套可行的方法，透過 eCos 的擴充，完成一個具自我維護能力的感測節點作業系統，可以不用透過回收的方式進行更新與維護的工作，減少了大量的回收人力。藉由實驗證實，本系統未因新增的自我維護機制使得系統效能有顯著的影響。

在未來工作方面，我們將設計一個自我檢測的子系統，將自我檢測、自我復原、自我更新的能力連成一氣，達到完善的自我維護能力。

此外，我們也將完成 ZigBee 無線網路協定整合至我們的系統中，並針對無線通訊協定以及作業系統的省電機制進行研究與實作，使我們的感測網路作業系統更趨完善。

致謝：

本文部份係工研院電通所執行經濟部委託之特殊具時效工業技術發展計畫"Embedded Linux 共通規格訂定及設計計畫"之成果，特此致謝。

9. 參考文獻

[1] Berkeley Sensor node. http://www.tinyos.net/scoop/special/hardware

[2] Berkeley TinyOS.

http://www.tinyos.net

[3] Phil Levis and David Culler, "Maté : a Virtual Machine for Tiny Networked Sensors " **,** *ASPLOS*, Dec 2002.

[4] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors", *In Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Tampa, Florida, USA, November 2004.

[5] A. Boulis, C.C. Han, and M. B. Srivastava, " Design and Implementation of a Framework for Programmable and Efficient Sensor Networks" , *MobiSys* 2003.

[6] Jaein Jong, David Culler, "Incremental Network Programming for Wireless Sensors", *IEEE SECON 2004* (Oct 2004).

[7] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava, **"**A dynamic operating system for sensor networks" ,*NESL Tech Report TR-UCLA-NESL-200502-01*, 2005.

[8] Linux Loadable Kernel Module HOWTO. http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Module-HOWTO.pdf

[9] ELF. http://www.x86.org/ftp/manuals/tools/elf.pdf

[10] eCos. http://sources.redhat.com/ecos/

[11] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, and Robert W. Wisniewski, Providing Dynamic Update in an Operating System", *USENIX 2005* , pp. 279-291, Anaheim California April 2005

[12] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, Henry M. Levy. *Recovering Device Drivers* , in *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[13] GCC, http://gcc.gnu.org/

[14] ARM, http://www.arm.com/

[15] Hynix HMS30C7202, http://www.magnachip.com/ENG/Products/MCU/32Bit/HMS30C7202_ref.html

[16] Stefan Dulman and Paul Havinga, "Operating System Fundamentals for the EYES Distributed Sensor Network", *Progress 2002, Utrecht, the Netherlands, October 2002*

[17] EYES Project, http://www.eyes.eu.org

[18] Texas Instruments. MSP430F149 datasheet.http://wwws.ti.com/sc/ds/msp430f149.pdf.

[19] David E. Culler, Wei Hong, "Wireless sensor networks," COMMUNICATIONS OF THE ACM June 2004/Vol. 47, No. 6

[20] Eduardo Souto, Germano Guimaraes, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz "A message-oriented middleware for sensor networks," *2nd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, October 18th - 22nd, 2004, Toronto, Ontario, Canada.

[21] 陳信養、李宗憲、馮立琪、沈仲九、李俊賢 "Design and Implementation of IEEE 802.15.4 Protocol", *National Computer Symposium, NCS, 2005*