

第五章

認識硬體

本章重點：

- * 大方向
- * 集中焦點
- * 如何溝通
- * 認識處理器
- * 認識外部週邊
- * 初始化硬體

硬體：名詞，電腦系統中會被腳踢到的部份。

一個嵌入式系統的工程師的職業生涯當中，將會有許多的機會接觸到五花八門的硬體。我將會提供我的經驗，讓你很快熟悉一個新的電路板。在程序上，我將引導你建立一個描述電路板關鍵特性的標頭檔，以及將硬體初始化成明確狀態的程式片段。

大方向

在撰寫一個嵌入式系統的軟體之前，你必須了解此硬體的用途為何。首先，你只需了解此系統的一般操作，不需要深入太多的細節，因為這些知識並不會立刻用到，而且會耗費許多的時間。

當你拿到一個新的電路板時，應該花一些時間閱讀電路板的技術文件。電路板應該會附有使用手冊，或是由軟體研發人員所寫的程式設計手冊。無論如何，如果這個電路板是針對於你的計劃所設計的，說明文件可能就不夠齊全，因為主要是說明是給硬體設計者參考用的。但無論是哪一種方式，閱讀手冊都是一個好的開始。

當你閱讀說明文件時，請將此電路板放置於手邊，這將會幫助你做圖文對照。當你閱讀完後，還將需要充分的時間實際上去了解這電路板。當你拿起這塊電路板之前，你應該能回答下列兩個問題：

1. 這個電路板的整個作用為何？
2. 資料的處理程序為何？

例如：假設你是數據機設計團隊中的一員，並擔任軟體研發的任務，你剛拿到由硬體研發人員所設計的電路板。因為你已經對於數據機相當熟悉，對於整個電路板的功能及資料流向都相當清楚。這個電路板的功能就是在類比的電話線路上收發數位信號。硬體從一組電子電路讀取數位信號，轉換成相關的類比信號後送出至電話線上。另一方面，當從電話線接收到資料之後，依照相反的處理方式輸出數位信號。

雖然系統大多數的功能都相當清楚，但是資料的處理方式卻不是如此。如果有一個描繪資料流向的圖表，將會更有助於快速了解。如果你夠幸運，你可以在說明手冊中發現這樣的一個圖解。無論如何，你可以自己建立這樣的圖解；如有必要自己動手，不妨可以略去與資料處理不相關的元件，如此更容易瞭解。

在 Arcom 的電路板中，硬體部分並不是被設計用於特殊的功能。所以，在本章的其餘部分，我們需要想像它擁有某種功能。在此，我們把它想像成是印表機分享器。印表機分享器允許兩部電腦分享同一部印表機。裝置的使用者透過串列埠連接兩部電腦，並且將印表機連接至並列埠。兩部電腦都可以傳送文件至印表機列印，但一次只能有一部電腦能執行列印功能。

為了說明印表機分享器的資料流向，我有繪製一份圖解，請參考圖 5-1。（在 Arcom 的電路板中的其餘不相關部份都被省略了）。經由這份圖解，你可以很快了解資料的流向。要被列印的資料從單一的串列埠接收，並儲存於記憶體中，當印表機準備列印時，經由並列埠傳送至印表機上，而執行的軟體則儲存於唯讀記憶體中。

一但建立了圖解，不要將它揉掉，應該要好好保存起來，在研發的過程中隨時做為參考。我建議在研發的過程中能準備一本筆記簿，將資料流的圖解做為第一頁，在你研發硬體的每一部份時，寫下每一個你所學習到的東西，你也可以記錄軟體的研發及製作過程。這本筆記簿不僅在研發的過程有用，在研發完成後也具有相當的價值。你將會發現，當你需要改變軟體，或是硬體需要更改時，這本筆記會發揮許多效用。

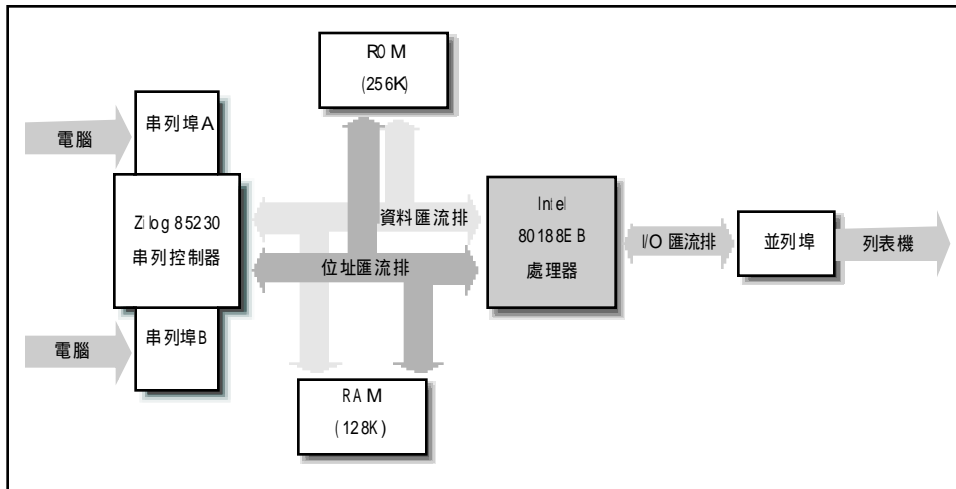


圖 5-1：印表機分享器的資料流向圖解

在你研讀硬體說明之後仍有疑問的話，你需要向硬體設計者尋求幫助。如果你不認識設計硬體的工程師，你需要花點時間介紹自己。如果必要，不妨可以和硬體工程師出去吃個飯或是下班之後喝個啤酒。我發現許多的軟體工程師和硬體工程師有溝通上的困難。在嵌入式系統中，硬體團隊與軟體團隊的溝通尤其重要。

集中焦點

了解處理器的動作方式是很重要的；畢竟，處理器只會依照你所撰寫的軟體來動作。想像一下處理器像什麼：處理器的世界像什麼？如果你從透視的觀點來看的話，你可以很快了解到處理器有許多的同胞，它們也屬於電路板上硬體的一部份，處理器能直接存取它們。在本章節，你將會學習到這些裝置的名稱，以及存取的方法。

首先，你要了解這些裝置有兩種基本的型態：記憶體及週邊裝置。很明顯的，記憶體用於資料、程式碼的儲存及讀取。至於週邊幾乎都是特殊的硬體裝置，用來與外界做互相溝通的動作，或是執行一個特殊的硬體功能。例如：兩種在嵌入式系統最常用的週邊裝置就是串列埠及計時器，前者屬於輸入輸出裝置，而後者基本上屬於一種計數器。

在 Intel 80x86 系列與其它的處理器家族中，它們有分離的位址空間來存取記憶體與週邊裝置。第一種定址空間稱為「記憶體空間」(Memory space)，主要是用於存取記憶體裝置的內容。第二種是保留給週邊裝置使用，稱為「I/O 空間」(I/O space)。當然，這些週邊也可以佔用記憶體空間，這完全取決於硬體的設計者。當使用這種設計方式時，我們稱這些週邊裝置為「記憶體映射式裝置」(memory-mapped device)。

由處理器的觀點來看，記憶體映射式裝置的動作就像是一般的記憶體。無論如何，它們的功能完全不同於記憶體。週邊不是單純的提供資料儲存的功能，它會將收到的資訊當成命令或是資料來作某些處理。如果週邊存在於記憶體空間中，我們稱該系統為使用「記憶體映射式的 I/O」(memory-mapped I/O)。

嵌入式系統的硬體設計人員常喜歡用 memory-mapped I/O 的方式，因為這對於硬體及軟體的研發都相當有利。在硬體方面，工程師可以省去部份 I/O 的空間，因為許多的線路都可以整合在一起。這種方式可能無法有效降低整個電路板的成本，但是能降低硬體設計的複雜度。使用記憶體映射式的週邊也對於程式設計者較有利，因為這樣就可以利用指標、資料結構 ... 等等軟體技術來與週邊互動，讓程式比較有效率，並容易設計。【註】

記憶體對應圖

所有的處理器都把程式及資料儲存在記憶體中。在某些狀況下，這些資料會存放在與處理器相同的晶片之內（像是 Intel 的 8051/8751），但大多數都儲存於外部的記憶體晶片內。這些記憶體裝置位於處理器的記憶體空間內，而處理器利用資料匯流排與位址匯流排與這些記憶體溝通。要存取某個位於記憶體中的位址，處理器首先送出要存取的位址到位址匯流排，然後透過資料匯流排上做資料傳輸。

當你在研究一個新的電路板時，你可以建立一個圖表，標示出每一個記憶體裝置及週邊裝置的名稱，並標示出所佔用的記憶體的位址。在圖表中，低位址繪於底層，高位址繪於上層。每次當你加入一個新的裝置時，你就將它加入這個表格中，並且標示對應的記憶體位址，用十六進制的數字標出開始的位址與結束的位址。

註 如果 P2LTCH 的位址映射到記憶體空間的話，之前章節提到的 toggleLED 函式就可以不用使用 in-line assembly。

如果你回頭看圖 5-1，你將會在 Arcom 電路板看見有三個裝置連接至位址匯流排及資料匯流排。這三裝置分別為 RAM、ROM 與 Zilog 85230 串列控制器。依照 Arcom 所提供的文件，RAM 的起始位址在於記憶體空間的底部，並且向上延伸 128K 的記憶體空間。ROM 位於記憶空間的頂部，向下延伸 256K，在這段記憶空間中有兩種記憶體裝置，一種是 EPROM，另一種是 Flash，各 128K 的容量。第三個裝置是 Zilog 85230 串列控制器，它是一個 memory-mapped I/O 的裝置，映射區間是 70000h 與 72000h 之間的記憶空間。

圖 5-2 顯示了這三個裝置在所佔用的記憶空間位址，我們常稱這種圖叫「記憶體對應圖」(memory map)，可以視為處理器的“address book”【譯註】。這個圖包含了實際記憶體所在的位址與週邊佔用的記憶體位址，記憶體對應圖的資訊對於整個系統相當重要，尤其是程式設計人員而言更是息息相關，應該隨時依最新狀況同步更新，並且永久保存。

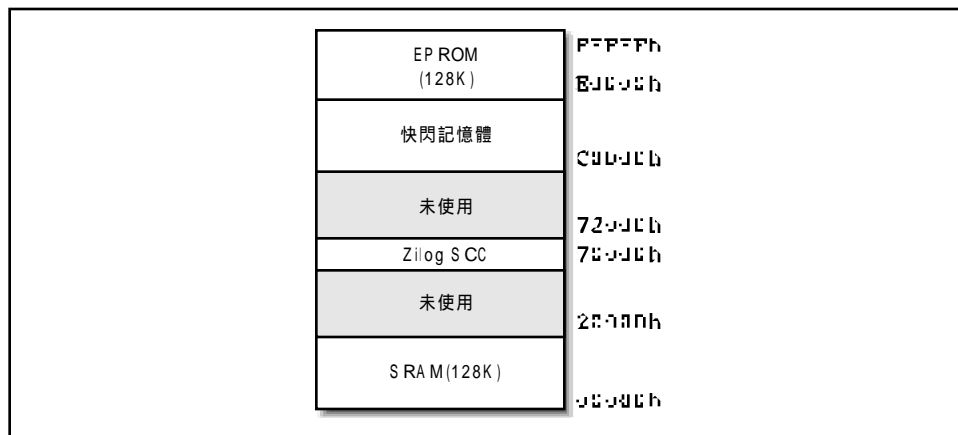


圖 5-2：Arcom 電路板的記憶體對應圖

對於每一片新的電路板，妳應該建立一個標頭檔，說明電路板上最重要的特性。這個檔案對於硬體提供一個抽象的資訊。在效用上，對於電路板上裝置的存取只需要用名稱而不需要去擔心這個裝置位於哪一個記憶體位址。加入標檔也會讓你的軟體更加有彈性，如果記憶體映射有改變，例如：如果 128K 的記憶體位址改變，你只要改變標頭檔中的記憶體位址，然後重新組譯一次。

註 這是個雙關語，address book 在在一般生活上是通訊錄的意思，在此處的另一個意義是「位址簿」。

依照本章進行的程序，我將教你如何對於 A rcom 的電路板建立一個標頭檔 (header file)。這個檔案的第一個部份如下所示。此部份的標頭檔描述了記憶體映射方式。它是把圖 5-2 予以程式化後的結果，如果你對程式中的位址描述格式有疑問，請參考「指標 vs. 位址」的附帶說明。

```

/*****
 *
 * 記憶體對應圖
 *
 *          基底位址      容量  描述
 *          -----
 *          0000:0000h    128K  SRAM
 *          2000:0000h                未用到
 *          7000:0000h                Zilog SCC 暫存器
 *          7000:1000h                Zilog SCC 的中斷認可
 *          7000:2000h                未用到
 *          C000:0000h    128K  Flash
 *          E000:0000h    128K  EPROM
 *
 *****/

#define SRAM_BASE      (void *) 0x00000000
#define SCC_BASE       (void *) 0x70000000
#define SCC_INTACK     (void *) 0x70001000
#define FLASH_BASE     (void *) 0xC0000000
#define EPROM_BASE     (void *) 0xE0000000

```

指標 vs. 位址

在 C 與 C++ 中，一個指標 (pointer) 的值就是一個位址。所以，當我們說有一個指標指向某些資料，也就是說資料儲存於該位址。但是程式設計人員通常不會直接設定該位址。只有作業系統的研發人員、設計裝置驅動程式的人、或嵌入式系統的軟體設計人員才需要直接指定指標的值。

不幸的是，位址的表示法每種處理器可能都不一樣，甚至依據編譯程式本身而有不同的方式；換言之，如果有一個如 12345h 的實體位址，其存放方式不見得就是你所想像的那樣，不同的編譯器有不同的作法【註】。所以，程式設計人員需要知道如何明確的表達實際位址，才能確定會讓指標指到正確的地方。

多數的針對 80x86 處理器的 C/C++ 編譯程式會使用 32 位元的指標。問題是，舊的處理器沒有線性的 32 位元定址空間。例如 Intel 80188EB 只有 20 位元的定址空間，除此之外，處理器內部的暫存器不會超過 16 位元，所以在這類處理器上使用兩個暫存器：一個區段暫存器 (Segment register) 以及一個偏移位址暫存器 (offset register)，利用這兩個 16 位元的暫存器來建立 20 位元的實際定址空間。(實際位址空間的計算方式，是將區段暫存器左移 4 個位元然後加上偏移位址而成，若相加的結果會進位到第 21 位元，原則是直接忽略)

要宣告並初始化一個指標，讓指標指向某個位於 12345h 實體位址的記憶體，我們宣告成：

```
int* pRegister = (int *) 0x10002345;
```

該宣告的最左邊 16 位元 (1000) 為區段值，最右邊的 16 位元 (2345) 為偏移位址。

為了方便起見，80x86 的程式設計者有時候撰寫位址會用 segment:offset 的格式來表示。若用這種方式，一個實體位址 12345h 將會需要寫成 0x1000:2345 — 但這只是其中一種寫法而已，事實上，每個實體記憶位址都有 4096 種等效的寫法，因為 0x1200:0345、0x1234:0005、與其它 4093 種可能，都指向相同的實體位址。

註 某些處理器提供數種記憶體模型 (memory model)，若加以考慮，情況將更為複雜。本書所有的範例都使用 80188EB 的大型記憶體模型 (large memory model)，在此模型下，我所告訴你的狀況都成立，但是在其它模型下，存放於指標的位址格式得要看所指到的資料或程式碼的型態而定！

I/O 對應圖

如果存在一個分離的 I/O 空間，同樣的，也需要建立一個 I/O 裝置與 I/O 位址對應的圖表，其程序大致相同。將週邊的名稱與 I/O 位址範圍描述於表格中，I/O 位址的部署方式就如同記憶體對應圖的方式。在絕大多數的情況下，通常大部分的 I/O 位址根本沒有用到，這些週邊裝置頂多只有幾個暫存器而已。

圖 5-3 是 Arcom 電路板的 I/O 對應圖。當中包含了三個裝置：週邊控制區 (Peripheral Control Block, PCB)，並列埠 (parallel port) 以及一個偵錯埠 (debugger port)。PCB 是一組位於 80188EB 中的暫存器，用於控制晶片內部的週邊裝置。這晶片用來控制位於該處理器外部的並列埠及偵錯埠，分別用於連接印表機，與在主機上的除錯器。

周邊控制區	FFFFh
未使用	FF00h
未使用	FE00h
並列埠	FC00h
偵錯埠	FC00h
未使用	E000h

圖 5-3 Arcom 電路板的 I/O 對應圖

I/O 對應圖對於在建立該電路板的標頭檔也相當有用。每一個 I/O 區間對應到一個常數上，稱為基底位址。以下就是依據 I/O 對應圖所建立的標頭檔：

```

/*****
*
* I/O 對應圖
*

```

```

*          基底位址      描述
*          -----
*          0000h          未用到
*          FC00h          SourceVIEW 的偵錯埠 (SVIEW)
*          FD00h          串列埠 (PIO)
*          FE00h          未用到
*          FF00h          週邊控制區塊 (PCB)
*
*****/

#define SVIEW_BASE  0xFC00
#define PIO_BASE    0xFD00
#define PCB_BASE    0xFF00

```

如何溝通

現在你知道連接至處理器的記憶體以及週邊的名稱及位址，該是學習如何與這些玩意兒溝通的時機了！

有兩種基本的通訊方式：輪詢 (polling) 與中斷 (interrupt)。不管是哪一種情形，處理器都會送出一些命令至裝置上 (透過直接 I/O 或 memory-mapped I/O)，並等待裝置給予回應。例如：處理器也許會要求一個計時器從 1000 倒數至 0。當倒數開始，處理器只會對一件事情感興趣：計時器是否完成計數的任務？

若使用輪詢的方式，處理器會重複確認是否計時器已經完成計數。這彷彿一位小孩長途旅行時，他會不斷問你「我們到了嗎？」；就像小孩的問題一樣，處理器會花相當多的時間去詢問計數結果，但都會得到負面的回應 - 計數仍未完成。要完成輪詢的軟體設計，你需要建立一個迴圈，不斷讀取該裝置相關的暫存器狀態，以下是一個例子：

```

do
{
    // 玩遊戲，閱讀，聽音樂 ... 等等
    ...
    // 確認一下狀態
    status = areWeThereYet();
} while (status == NO);

```

第二個溝通方式是使用中斷。中斷是一種非同步的電子信號，由週邊裝置發出並傳向處理器。當使用中斷時，處理器查詢週邊狀態的方法還是一樣，但不會主動去詢問，只會保持正常的運作並注意中斷訊號的到來。當週邊發出中斷訊號時，處理器會暫時放下手邊的工作，然後將各暫存器的狀態加以保存，並開始執行中斷服務常式（Interrupt SubRoutine, ISR）。當 ISR 完成之後，處理器又會繼續剛剛在中斷前未完成的工作。

當然，這並非全自動的。程式設計師需要撰寫自己的 ISR、安裝並進行相關的設定，如此 ISR 才能在相關的事件發生時正確執行。若你是新手，這些工作對你將是一種挑戰。儘管如此，使用中斷的方式有助於降低程式的複雜度，並且會改善程式的結構。對照之下，當使用輪詢的方式時，輪詢的程式碼可能要四處安插 — 包括那些不相關的程式碼，如此一來，程式的結構很可能被搞得七零八落！

以整體來看，使用中斷方式的處理器，其使用效率高於輪詢方式 — 因為處理器不用耗掉一堆時間做一些神經兮兮的動作，三不五時就去拷問週邊的狀態！無論如何，每一個中斷都會有一些額外的工作要做；當中斷產生時，處理器需要將目前程式執行的狀況加以儲存，然後才能開始執行 ISR，此時處理器中的許多暫存器內容就需要暫時存放到記憶體中，而較低優先權的中斷也會被取消。當週邊有許多的裝置需同時監控的時候，或事件的發生有優先順序的狀況，此時是中斷方式最有效率的時候。但碰到某些特殊場合，中斷的方式無法使處理器產生夠快的回應，此時就必須考慮使用輪詢的方式。

中斷對應表

大多數的嵌入式系統只有少數的中斷，每個中斷都有關連的中斷腳位（位於處理器晶片外頭的 PIN 腳）以及負責處理相關工作的 ISR。為了讓處理器能執行到正確的 ISR，必須讓中斷腳位對應到 ISR，這個對應關係通常由中斷向量表（Interrupt Vector Table）來負責。向量表只是一個指標陣列，用來指向 ISR 的起始執行位址，處理器使用中斷型態（每個中斷腳位有一個中斷型態）以及索引的方式來進入指標陣列搜尋，然後找出相對應的 ISR 來執行。

你必須正確的初始化中斷向量表（如果沒搞對，很可能對應到錯誤的位址，或是無法執行）。首先要做的是建立一個中斷對應表，並填入相關資訊。此表列出了會用到的中斷型式，以及中斷的來源裝置。這些資訊都常都包含在該電路板所提供的文件之中。表 5-1 列出了 Arcom 這個電路板上的中斷。

Table 5-1 : Arcom 電路板的上中斷對應表

中斷型式	產生中斷的裝置
8	計時器/計數器 #0
17	Zilog 85230 SCC
18	計時器/計數器 #1
19	計時器/計數器 #2
20	串列埠接收端
21	串列埠傳送端

要再次強調的是，我們的目的是將資訊轉換成一個表格，這對於程式設計人員很有幫助。建立出此表格之後，你應該利用它來撰寫一個標頭檔，將每個中斷向量都用 #define 加以定義：

```

/*****
*
* 中斷對應表
*
*****/

/*
* Zilog 85230 SCC
*/
#define SCC_INT      17
/*
* 晶片上的計時器 (Timer), 計數器 (Counter)
*/
#define TIMER0_INT   8
#define TIMER1_INT   18
#define TIMER2_INT   19
/*
* 晶片上的串列埠
*/
#define RX_INT       20
#define TX_INT       21

```

認識處理器

如果你從未接觸過電路板上的處理器，必須花一些時間來熟悉它。如果你使用 C/C++ 語言，事情就好辦了。對於使用高階語言的人，處理器的型態其實不是關鍵。當然，如果你想用組合語言來設計程式，就得熟悉處理器的架構及指令集。

你想知道的處理器資訊，都能在廠商提供的 databook (技術手冊) 內找到。如果你沒有技術手冊或程式設計師手冊，應該立即去找一份。如果你想要成為一位成功的嵌入式系統程式設計師，就必須能閱讀技術手冊，從中找到你要的資訊。處理器的技術手冊通常寫得很好，可以從這個地方開始。快速翻閱整本技術手冊，標出需要的章節，然後回頭仔細研讀整個章節。

一般的處理器

許多常用的處理器都是相關的系列。在某些情況下，這些處理器家族的成員依照演進的路徑不斷創新，最明顯的例子是 Intel 的 80x86 系列，從最初的 8086 到 Pentium III，80x86 推展了另一波工業革命的浪潮。

本書所用到的「處理器」，指的是以下三種裝置中的任何一種：微處理器 (microprocessor)，微控制器 (microcontroller) 以及數位信號處理器 (Digital Signal Processor, DSP)。微處理器通常指的是一個晶片，包含一個功能強大的 CPU，此處理器未著重於任何特殊運算的功能。這些晶片通常用於 PC 和高階工作站。常用的微處理器家族是 Motorola 的 68K 系列 - 用在舊型的 Mac 電腦；另一個無所不在的是 Intel 的 80x86 系列。

微控制器非常類似微處理器，它適用於特殊的嵌入式系統。微控制器通常包含一個 CPU、記憶體 (少量的 RAM、ROM) 以及其它的週邊，全都塞在一顆晶片中。只用一顆晶片就能代替這些電路裝置可以大幅降低嵌入式系統的成本。最常用的微控制器是 Intel 的 MCS-51 系列 (8051、8751 ...) 與 Motorola 的 68HCxx 系列。你也可以在常用的微處理器中發現微控制器的版本，例如：Intel 的 386EX 就是 80386 微處理器的一個特殊版本。

最後一種型態是數位信號處理器，簡稱 DSP。在 DSP 中的 CPU 被設計用於執行特殊的信號計算；例如：用於影像與聲音通訊的計算。這類計算需要非常高速的運算能力。因為 DSP 能執行這類快速的運算，所以比其它兩種的處理器更快。DSP 為設計師提供了功能強大且低成本的微處理器，能當作數據機、通訊設備與多媒體裝置的替代品。DSP 領域裡的兩大家族是分別是德州儀器公司 (TI) 的 TMS320Cxx 與 Motorola 的 5600x 系列。

Intel 80188EB 處理器

在 Arcom 電路板上的處理器是 Intel 的 80188EB，它是修改自 80186 的一顆微控制器。除了 CPU 之外，80188EB 包含了一個中斷控制單元，兩個可程式控制的 I/O 埠 (programmable I/O)，三個計時器/計數器 (Timer/Counter)，兩個串列埠，一個 DRAM 控制器，以及一個晶片選擇單元 (chip-select)。這些額外的週邊裝置都塞進同一顆晶片中，可視為晶片中的週邊裝置。CPU 可透過內部的匯流排直接與這些在晶片中的週邊裝置溝通。

雖然這些都是個別的硬體裝置，但是它們工作起來就像是一個經過擴充的 80186 CPU。軟體可以透過一個 256 位元組的暫存器區塊來控制這些週邊裝置，此區塊稱為週邊控制區塊 (Peripherals Control Block, PCB)。你應該記得，當我們頭一次討論這電路上的記憶體以及 I/O 對應時，曾談到這個控制區塊。PCB 的預設狀態是位於 I/O 空間中，起始位置是 FF00h。無論如何，PCB 能被設定於任何適當的地址，不論是 I/O 空間或是記憶體空間內。

對於晶片內的每個週邊裝置，其控制與狀態暫存器都位於 PCB 的基底位址加上固定的偏移位址。每一個暫存器的實際偏移位址都能在 80188EB 的 databook 中找到。為了從應用軟體中獨立出這些細節，你最好將這些暫存器的偏移位址放入標頭檔中。我已經做了 Arcom 這塊電路板的標頭檔，但以下只列出在這本書中談到的一些暫存器：

```

/*****
*
* 晶片內的週邊
*
*****/
/*
* 中斷控制單元 (Interrupt Control Unit)

```

```
*/
#define EOI      (PCB_BASE + 0x02)
#define POLL     (PCB_BASE + 0x04)
#define POLLSTS  (PCB_BASE + 0x06)

#define IMASK    (PCB_BASE + 0x08)
#define PRIMSK   (PCB_BASE + 0x0A)

#define INSERV   (PCB_BASE + 0x0C)
#define REQST    (PCB_BASE + 0x0E)
#define INSTS    (PCB_BASE + 0x10)

/*
* 計時器 / 計數器 (Timer/Counter)
*/

#define TCUCON   (PCB_BASE + 0x12)

#define T0CNT    (PCB_BASE + 0x30)
#define T0CMPA   (PCB_BASE + 0x32)
#define T0CMPB   (PCB_BASE + 0x34)
#define T0CON    (PCB_BASE + 0x36)

#define T1CNT    (PCB_BASE + 0x38)
#define T1CMPA   (PCB_BASE + 0x3A)
#define T1CMPB   (PCB_BASE + 0x3C)
#define T1CON    (PCB_BASE + 0x3E)

/*
* 可程式規劃的 I/O 埠 (Programmable I/O Ports)
*/

#define P1DIR    (PCB_BASE + 0x50)
#define P1PIN    (PCB_BASE + 0x52)
#define P1CON    (PCB_BASE + 0x54)
#define P1LTCH   (PCB_BASE + 0x56)

#define P2DIR    (PCB_BASE + 0x58)
#define P2PIN    (PCB_BASE + 0x5A)
#define P2CON    (PCB_BASE + 0x5C)
#define P2LTCH   (PCB_BASE + 0x5E)
```

其它你可以從 databook 找出的東西包括：

- 中斷向量表應該放置於何處？它應該放置於記憶體的特殊位址嗎？如果不是，處理器該如何知道其位址？
- 中斷向量表的格式為何？它只是一個指標指向 ISR 嗎？
- 有任何特殊的中斷嗎？有時我們稱之為陷阱（trap），由處理器本身所產生。必須另寫一個 ISR 來處理這類的中斷嗎？
- 如何啟用（Enable）某個中斷？該如何停用（Disable）它？若想一次啟用所有中斷該怎麼做？如何一次就停用所有的中斷？
- 中斷如何被認可（Acknowledged）或清除（cleared）？
- 中斷的優先權如何決定？又該如何變更？

認識外部週邊

聊到這裡，除了外部的週邊，相信你已經對於硬體本身有相當了解。顧名思義，外部週邊就是位於處理器外部的硬體裝置，處理器可以透過中斷的方式，以 I/O 或 memory-mapped I/O 的方法來與這些裝置溝通。

一開始，我們先製作外部週邊的清單。不同的應用會有不同的清單內容，這份清單包含了 LCD、鍵盤控制器、類比/數位轉換器（Analog/Digital Converter, ADC），網路介面晶片，或是依特殊應用所訂製的 IC（Application-Specific Generated Circuit，或稱為 ASIC）。在 Arcom 的電路板上，這份清單表包含三種裝置：Zilog 85230 串列控制器（Serial Controller，後文簡稱 SCC），並列埠以及偵錯埠。

你應該為清單中的每項裝置都取得一份類似 databook 的資料。在開發計畫的初期，不要盲目的急著想寫程式【譯註】，應該花點時間好好研究這些文件，了解每個裝置的基本功能。搞清楚每項裝置的用途、與處理器溝通的方式、有哪些是控制暫存器？發送控制命令的方法與命令格式為何？哪些是狀態暫存器？如何查詢裝置的狀態？在暫存器中的每個位元代表的意義？該裝置產生中斷的時機？中斷如何確認？又如何回應中斷？

譯註 不要懷疑，這是大多數工程師的通病，包括我自己。

當你正在設計嵌入式系統，應該試著將程式依照裝置加以區分。能依不同的裝置而分離出來的程式，稱做外部裝置的驅動程式（driver）。驅動程式的作法很簡單，只要將控制此週邊裝置的副程式加以結合，並把它從應用程式中分離出來。關於驅動程式，我有很多話不吐不快，我保留到第七章討論週邊裝置時再加以說明。

初始化硬體

最後一個步驟是要讓你知道如何撰寫初始化（initialization）硬體的程式。這是在研發的過程中最接近硬體的機會，尤其是使用高階語言發展軟體時。在硬體初始化的過程中難免會用到組合語言。無論如何，完成這個步驟之後，你將要開始用 C/C++ 撰寫程式。



如果你是第一個接觸到新電路板的軟體工程師 — 尤其是雛形（prototype）電路板 — 此時硬體可能還無法正常運作。所有以處理器為基礎的電路都需要一些軟體來加以測試，以確定硬體的機能與週邊是否正確。當某些功能無法正常運作時，會讓你覺得份外棘手 — 因為你將很難搞清楚到底是硬體還是軟體的問題。除非你非常熟悉硬體，而且可以用 ICE 來檢驗新電路板各部份的功能，否則，就應該要請硬體工程師加入除錯的工作。

硬體的初始化動作需要在起始碼（startup code）開始執行之前就要完成，第三章曾談過起始碼，當時我們假設硬體都已經過初始化了，因為目的是建立一個適合高階語言執行的環境。圖 5-4 提供一個完整初始化的過程，從處理器的重置（reset），經由硬體的初始化以及 C/C++ 的起始碼，然後到主程式。

第一個階段的初始化程序是啟動碼（reset code），這是一小段的組合語言程式（通常只有兩三個指令），當處理器在開機或重置時會立即執行。這麼做的目的是要將控制權轉移至硬體初始化的程式（hw_init）中。啟動程式的第一個指令需要放於記憶體的一個特殊位址，通常稱為啟動位址（reset address），你得自己查閱處理器的 databook，看看啟動位址到底在哪個實際位址。在 80188EB 的啟動位址為 FFFF0h。

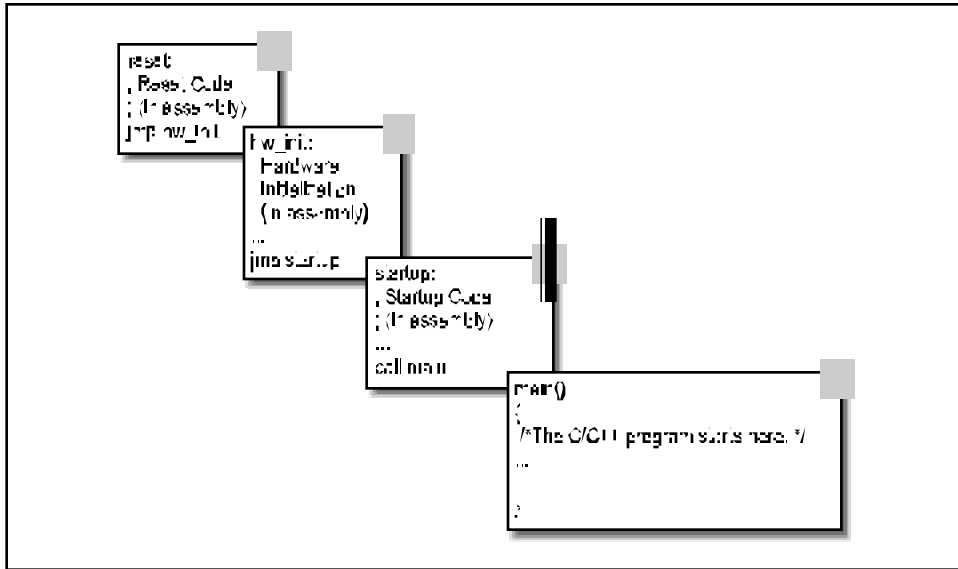


圖 5-4 硬體與軟體的初始化程序

實際上，大部分的硬體初始化都在第二個步驟完成。此時，我們需要去提醒處理器它的運作環境為何，通常是先去初始化中斷控制器以及其它較首要的週邊設備，其它次要的硬體裝置，可以等待裝置的驅動程式啟動之後再做，通常都是在主程式當中。

在 Intel 80188EB 中，有幾個內部的暫存器必須先予以適當規劃，然後才能開始執行有用的工作。這些暫存器負責設定記憶體及 I/O 的映射，以及微處理器內部的晶片選擇 (chip-select) 單元。藉由規劃晶片選擇暫存器，你就可以選用與處理器連接的記憶體或 I/O 裝置，每個晶片選擇暫存器都對應到一條啟用晶片 (chip-enable) 的線，處理器利用這些線來啟用連接在其周圍的裝置，至於詳細的對應關係則由硬體設計人員所決定。所以，你要做的就是硬體工程師那裡拿到這些對應關係的列表，然後依照這份關連性清單去設定相關的晶片選擇暫存器。

當重置時，80188EB 會假設一個最差的情況，它會假設目前只有 1024 bytes 的 ROM — 位於 FFC00h 到 FFFFFh 之間的位址，而且沒有其它的記憶體以及 I/O 裝置。這是處理器的初始情況 (fetal position)，而且 hw_init 常式必須位於 FFC00h (或更高的位址)，還不能用到任何的 RAM！硬體初始化常式應該由先規劃晶片選擇暫存器，讓處理器了解目前安裝在這塊電路板上的還有哪些裝置；當這個動作完成之後，所有的 ROM 與 RAM 就能夠運作了，這時軟體才可以開始使用這些位址。

第三個初始化的步驟就是起始碼 (startup code)。這是我們在第三章曾看過的組合語言程式。這個程式的工作是準備一套軟體環境，讓使用高階語言的程式可以執行。我們通常稱這段起始碼為主程式 (main)，從這裡開始，就可以使用 C 或 C++ 來撰寫軟體。

順利的話，你應該對嵌入式系統的軟體 — 從處理器重置到開始執行主程式 — 有基本的認識。不可否認的，如果你是第一次將許多的元件結合在一起 (啟動程式、硬體初始化、C/C++ 的起始程式、以及應用程式)，難免會有一些問題，這是很正常的，通常會需要耗用一些時間來除錯。其實，這是整個過程中最難的部份。以後你就知道，這個 LED 的閃爍程式雖然再簡單不過，卻可能成為未來大型計畫的骨架。

談到這裡，我們已經為嵌入式系統的程式設計建立的一個基本架構，接下來要討論到進階主題：記憶體測試、裝置的驅動程式、作業系統與確實有用的程式。這些片段的軟體你可能早已經在其它的電腦系統看過了，但是要把它們應用到嵌入式系統的架構上，還需要費一番手腳，才能順利的“嵌”進來。