第11章嵌入式系統

在前面的章節中，我們已經介紹了電腦的硬體架構、組合語言、組譯器、連結器、載入器、C 語言等主題。現在，是讓我們將這些概念全部整合起來的時候了，在本章中，我們將利用嵌入式系統這個主題，整合前述的所有內容，完整的說明如何使用 C 語言、組合語言、連結器等工具，建構出電腦的軟硬體系統。

在嵌入式系統當中，通常沒有作業系統可以使用，此時，系統程式設計師必須能從硬體開始，以最原始的方式，逐步建構出整個系統。在這樣的過程當中，我們可以進一步理解軟硬體系統，這是為何在本章探討此一主題的原因。

目前，開發嵌入式系統程式時，通常是將嵌入式的硬體，連接到個人電腦上，然後透過跨轉編譯器 (Cross Compiler)，產生出可在嵌入式的硬體上執行的二進位執行檔。再將這個二進位檔燒錄到該電腦的記憶體 (EPROM 或 Flash) 當中，然後，按下啟動鍵，重新啟動電腦以執行該系統。

嵌入式程式設計師所必須做的工作，包含撰寫啟動程式 (11.5 節)、設定中斷向量函數 (11.4 節)、撰寫驅動程式 (11.2 節) 等等。然後，撰寫專案編譯檔 (make 檔)，以整合系統中所有的程式 (11.6 節)，形成一個完整的嵌入式系統。

11.1 輸出入

對當今的 CPU 而言，存取輸出入裝置的方法，通常可以分為兩類，第一類是採用專用的輸出入指令，第二類是使用記憶體映射的輸出入方式。以下，我們將分別介紹這兩種輸出入方法。

專用的輸出入指令

在某些處理器中會提供專用的輸出入指令，像是測式裝置 TD (Test Device)、讀取裝置 RD (Read from Device)、寫入裝置 WD (Write to Device) 等等。我們可以使用 TD 指令測試輸入裝置是否有讀到資料，當發現資料進來後，再利用 RD 指令將資料讀入暫存器當中，以完成輸入工作。

舉例而言，假如輸入裝置 0x09 代表鍵盤，那麼，範例 11.1 將會不斷的用 TD 指令測試鍵盤是否有資料輸入。一但有按鍵資料進入時，再用 RD R1, inDev 將資

料讀入暫存器 R1 當中。最後，再利用 STB R1, key 將讀入的資料存到變數 key 當中。

範例 11.1 從輸入裝置讀入一個位元組的程式

組合語言 (輸入資料)	說明
inLoop: TD inDev JEQ inLoop RD R1, inDev STB R1, key inDev BYTE 0x09 key RESB 1	inLoop 標籤 測試輸入裝置 0x09。 若沒有輸入，跳回到 inLoop 繼續測試 讀取輸入值到暫存器 R1 當中。 將讀到的值存入變數 data

同樣的，我們也可以利用 TD 指令測試輸出裝置是否已準備好，當裝置準備就緒後，再利用 WD 指令將資料寫入該輸出裝置中，以完成輸出工作。

舉例而言，假如輸出裝置 0xF3 代表具備字元顯示能力的螢幕，那麼，範例 11.2 將會不斷的用 TD 指令測試螢幕是否已準備就緒，當螢幕準備就緒後，再用 LDB R1, ch 將字元 'A' 載入到暫存器 R1 當中。最後，再利用 WD R1, ch 將 R1 中的字元輸出到螢幕中，如此，螢幕上將會顯示出字元 'A'。

範例 11.2 將字元 A 顯示到螢幕的程式

組合語言 (輸出資料)	說明
oLoop: TD oDev JEQ oLoop LDB R1, ch WD R1, oDev oDev BYTE 0xF3 ch WORD 'A'	oLoop 標籤 測試輸出裝置 0xF3 若該裝置未就緒，則跳回 oLoop，直到就序為止 將欲輸出的資料 (字元 'A') 載入到暫存器 R1 中 將字元 'A' 輸出到輸出裝置 0xF3 當中

雖然，範例 11.1 與範例 11.2 分別代表輸入與輸出，但是，兩者都利用了等待迴圈，以等候輸入裝置的資料進入，或等候輸出裝置就緒。這種等待迴圈會讓 CPU 陷入忙碌狀態，而無法進行其他計算，因此稱為忙碌等待 (Busy Waiting)。

當然，我們也可以在等待迴圈當中，不斷檢查所有的輸入裝置，一但發現有輸入進來，就執行對應的動作，這種利用等待迴圈等候裝置的方法，稱為輪詢法 (Polling)。

舉例而言，假如系統中有三個輸入裝置，包含鍵盤 (0x09)、滑鼠 (0x0A) 與網路卡 (0x0B)。那麼，我們就可以利用程式輪流詢問這三個裝置，一但有輸入事件發生，則將資料讀入變數 inData，並將該輸入裝置的代號儲存於 inDev 變數中，以利後續進行輸入處理。

範例 11.3 輪流詢問鍵盤、滑鼠與網路卡三個裝置的程式

組合語言 (輸入資料)	說明
inLoop:	輪詢迴圈開始
LD R1, 0	清除 R1 中的值
TestKeyboard:	檢查鍵盤輸入
TD keyboard	測試鍵盤是否有按下
JNE TestMouse	如果沒有則測試下一個裝置
LD R1, keyboard	(如果有)則將按鍵資料放入 R1
TestMouse:	檢查滑鼠輸入
TD mouse	測試滑鼠是否有輸入
JNE TestNet	如果沒有則測試下一個裝置
LD R1, mouse	(如果有)則將滑鼠資料放入 R1
TestNet:	檢查網路輸入
TD net	測試網路是否有輸入
JNE EndTest	如果沒有則結束輸入偵測
LD R1, net	(如果有) 則將網路資料放入 R1
EndTest:	結束輸入偵測
ST R1, inDev	將輸入裝置代號放入 inDev 變數中
RD R2, inDev	讀取輸入值，放入 R2 暫存器中
ST R2, inData	將輸入值存入 inData 變數中
… 處理輸入 (省略) …	… 處理輸入的程式碼 (省略) …
JMP inLoop	輪詢迴圈結束，回到 inLoop
keyboard BYTE 0x09	鍵盤的裝置代號為 0x09
mouse BYTE 0x0A	滑鼠的裝置代號為 0x0A
net BYTE 0x0B	網路的裝置代號為 0x0B
inDev RESB 1	變數 inDev 用來儲存輸入裝置代號
inData RESW 1	變數 inData 用來儲存輸入值

對於簡易的嵌入式系統而言，輪詢是一種常見的作法。然而，如果希望電腦的效能可以發揮得淋漓盡致，那麼，被輪詢迴圈卡住 CPU 的作法，就不是一個好的解決方式了。因此，對於較注重效能的電腦而言，往往會利用所謂的中斷機制代替輪詢，以避免輪詢所造成的效能問題，增進電腦的效率，有關中斷機制的說明，

我們將於後文中詳細的說明。

記憶體映射輸出入

所謂的記憶體映射，是利用記憶體存取指令進行輸出入的方法，在採用這類方式的 CPU 當中，很可能沒有專用的輸出入指令，於是利用記憶體存取指令替代輸出入指令，以達成輸出入的功能。

目前，使用記憶體映射輸出入的方式越來越常見，甚至，在許多現今的 CPU 當中，往往將輸出入指令完全取消，在本書當中的 CPU0 即是如此。那麼，這類的電腦要如何進行輸出入呢？

當然，在這類的電腦當中，還是必須能進行輸出入，否則該電腦就無法與鍵盤、螢幕等裝置溝通，如此一來，電腦的功能也就所剩無幾，可以說是無法使用了。

在這類沒有輸出入指令的電腦，要進行輸出入時，會分配給每一個周邊裝置一些記憶體位址空間。當程式使用 ST、STB 等記憶體寫入指令，將資料寫入該周邊裝置的記憶體位址時，對應的輸出裝置就會進行輸出動作。同樣的，當程式使用 LD、LDB 等記憶體讀取指令，讀取這些位址時，就會讀取到該周邊裝置的輸入資料，如此，即可透過記憶體存取的指令，存取這些周邊裝置。

對於程式設計師而言，『將周邊裝置視為記憶體的一部分』，這個方法聽起來很奇怪，但是，若程式師能改以硬體設計人員的立場，去思考這件事，就會覺得不那麼奇怪了。

對於硬體設計人員而言，CPU、記憶體與輸出入裝置，都透過匯流排被連接在一起。對於 CPU 而言，輸出入裝置與記憶體都被視為是『外部裝置』，CPU 只不過是透過將位址傳入到位址匯流排上，然後透過資料匯流排傳送或接收資料，以與這些外部裝置溝通。因此，不論是記憶體或輸出入裝置，對 CPU 而言其實運作原理都一樣。

既然如此，那為何不將輸出入裝置當成是記憶空間的一部分，給這些輸出入裝置一些『記憶體位址』呢？也就是說，將記憶體空間分成兩區，其中一區是真正的記憶體位址，另一區則是輸出入裝置的『記憶體映射位址』。如此、只要 CPU 指定的位址是輸出入的映射位址，則會進行輸出入動作，如果 CPU 指定的位址是記憶體位址，則會進行記憶體存取動作。根據這樣的設計，我們就能利用記憶體存取指令進行輸出入裝置的存取了。

記憶體映射通常是輸出入控制器的工作。對於輸出裝置而言，當輸出控制器發現控制匯流排上，出現記憶體寫入訊號，且位址匯流排上的記憶體位址，代表某個輸出暫存器時 (也就是該暫存器被映射到輸出位址上時)，就將資料匯流排上的資料存入該暫存器當中。同樣的，對於輸入裝置而言，當有資料輸入時，會被暫時儲存在輸入介面卡上的暫存器當中，等到輸入控制器發現控制匯流排上，出現記憶體讀取訊號，而且位址匯流排上的位址，代表某輸入暫存器時，才將該暫存器中的資料取出，傳送到資料匯流排上，讓 CPU 取得該記憶體映射後的暫存器資料。這就是記憶體映射的硬體實作方法。圖 11.1 顯示了利用裝置控制器進行記憶體輸出入映射控制的方式。

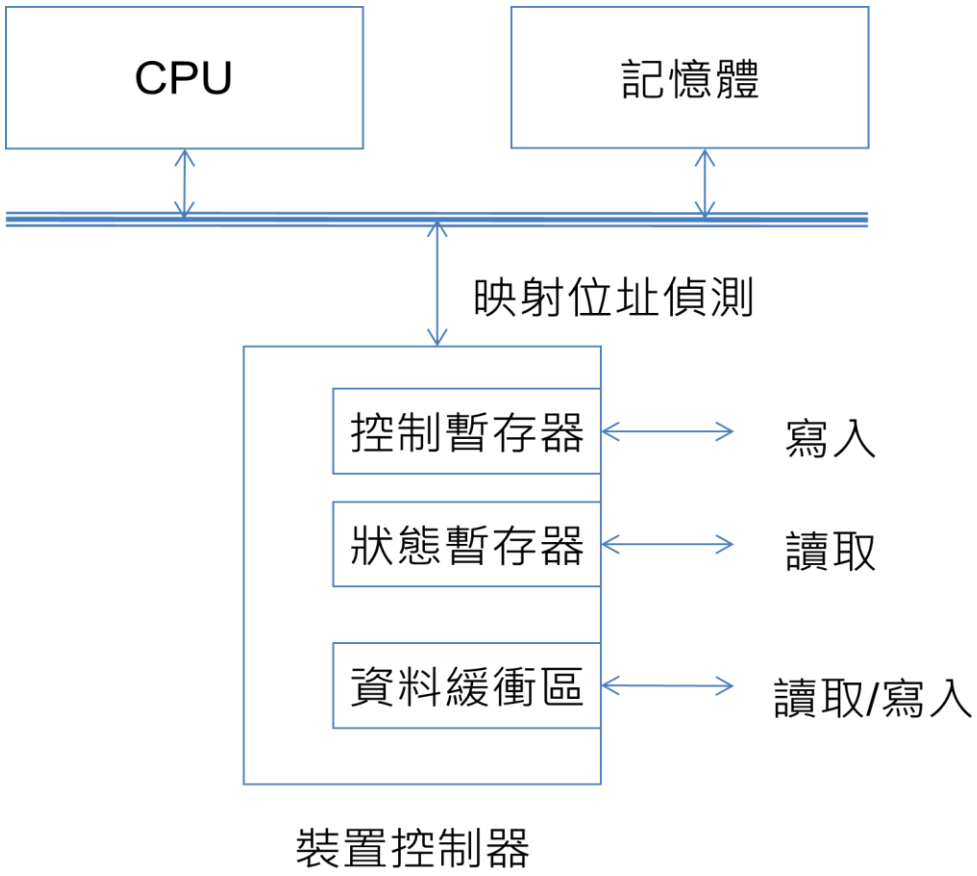

圖 11.1 裝置控制器與記憶體映射機制

在本書當中所使用的處理器 CPU0，並沒有專用的輸出入指令，因此，CPU0 只能透過記憶體映射的方式進行輸出入。在 CPU0 當中，組合語言程式設計師可以透過 LD、ST、LDB、STB 等指令，進行周邊裝置的存取，其前提是在硬體接線上，必須以記憶體映射的方式設計線路，也就是硬體工程師在設計輸出入控制器或介面卡時，就必須以記憶體映射的方式『解釋』位址匯流排上的資料。

既然 CPU0 是透過記憶體映射的方式進行輸出入動作，那麼，其輸出入程式設計方式就與一般的記憶體存取無異。只是這些記憶體位址，對輸出入控制器而言，具有特別意義而已。

接下來，我們將以組合語言範例，實際說明 CPU0 的記憶體映射輸出入方式。然而，光是有 CPU 卻沒有輸出入裝置，撰寫出來的輸出入程式將毫無意義。因此，筆者只好繼續扮演硬體工程師，親自利用 CPU0 設計一台陽春型的電腦，稱為 Machine 0，簡稱 M0。在下一節當中，我們將介紹 M0 的硬體結構，然後，利用 CPU0 的組合語言，導引讀者寫出 M0 上的鍵盤控制程式。

簡易電腦 **M0**

簡易電腦 M0 是一種單晶片實驗板，採用 CPU0 作為處理器，包含 16 K 的 FLASH 與 64K 的 RAM。16 K 的 FLASH 扮演了唯獨記憶體的角色，之所以選用 FLASH 而不採用一般的 ROM 是為了測試方便。因為可以利用燒錄器將程式與資料燒錄到 Flash 中，圖 11.2 顯示了簡易電腦 M0 的外觀與基本架構。

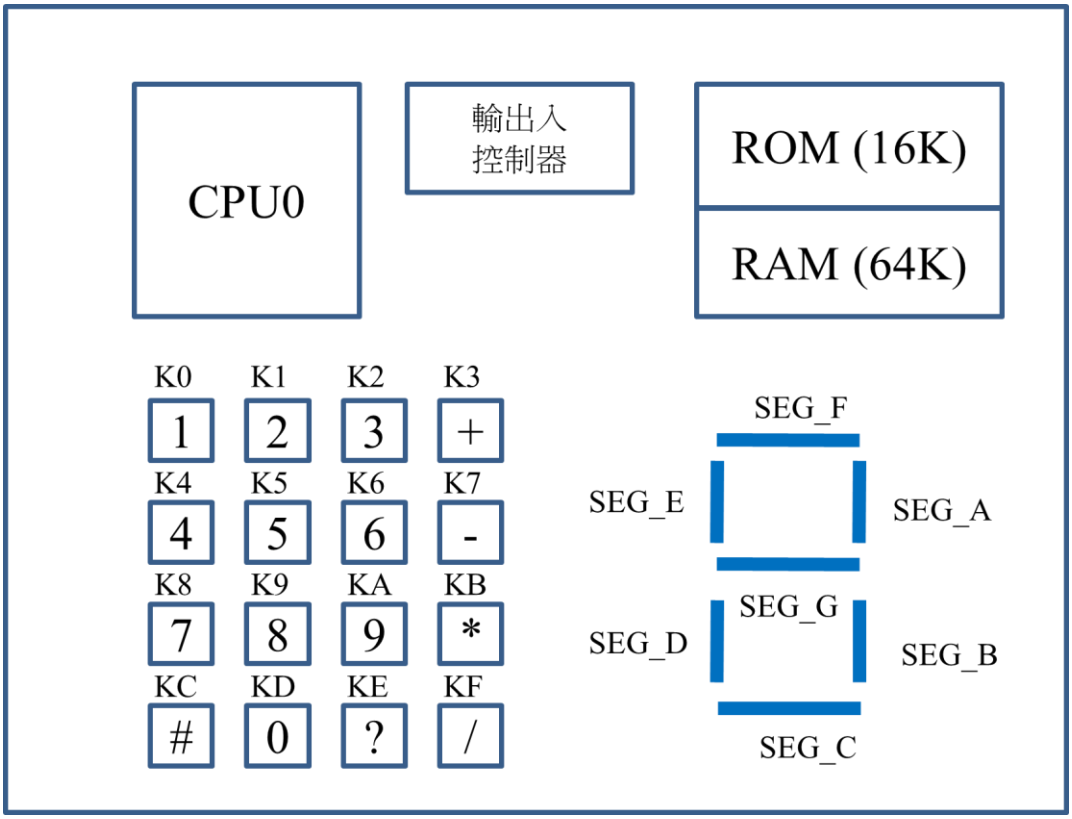

圖 11.2 簡易電腦 M0 的基本架構

程式設計師可以透過個人電腦所控制的燒錄裝置，將程式燒錄到 M0 的 Flash 當中，只要啟動程式正確的燒錄到其中，M0 就可以順利的開機，M0 當中的 RAM

則可用來儲存堆疊與變數等動態資料。

另外，M0 上還有一組鍵盤，共有 16 個按鍵，還有一個七段顯示器，可用來顯示一個數字。鍵盤上的每一個按鍵都有編號，從 K0 開始直到 KF，代表 Key 0、Key 1、…、Key F 的縮寫，而七段顯示器的每一根光棒 (LED) 也都有編號，從 SEG_A 開始編到 SEG_G 為止。

簡易電腦 M0 完全採用記憶體映射的方式進行輸出入，輸出入的映射位址位於定址空間的最後部分，其中的 0xFFFFFF00 是七段顯示器的映射區域，而 0xFFFFFF01 與 0xFFFFFF02 則是鍵盤按鈕的映射區域。這些映射方式均記載於其硬體手冊當中。表格 11.1 是 M0 的硬體對應手冊。請讀者仔細對照圖 11.2 與表格 11.1，就可以找到每個按鍵與光棒各被映射到哪一個位元上。

表格 11.1 簡易電腦 M0 的硬體對映手冊 (Data Sheet)

Reg bit	7	6	5	4	3	2	1	0
IO_REG0 0xFFFFFF00		SEG_G	SEG_F	SEG_E	SEG_D	SEG_C	SEG_B	SEG_A
IO_REG1 0xFFFFFF01	KF	KE	KD	KC	KB	KA	K9	K8
IO_REG2 0xFFFFFF02	K7	K6	K5	K4	K3	K2	K1	K0

M0 上的所有按鈕，都是在按下去時，對應的位元才會變成 1，在未按鈕的情況下該位元會是 0。同樣的，七段顯示器上的光棒也是以 1 作為點亮的狀態，而 0 作為熄滅的狀態。

根據圖 11.2 與表格 11.1，我們就可以開始撰寫簡易電腦的程式了。假如，我們希望寫一個程式，可以讓七段顯示器顯示出 0 這個數字，那應該怎麼做呢？

根據我們的目標，我們必須讓七段顯示器外圍的六根亮棒都亮起來，然後讓中心的亮棒熄滅。因此，我們必須將 SEG_A ...SEG_F 都設成 1 ，而 SEG_G 則設成 0。根據這樣的想法，程式將出奇的簡單，只要將記憶體位址 0xFFFFFF00 之處的 byte 設定為 0x3F，就能讓該七段顯示器顯示出字元 0，該程式如範例 11.4 所示。

範例 11.4 讓 M0 的七段顯示器顯示數字 0 的程式 (絕對定址版-錯誤示範)

組合語言	C 語言 (對照版)	C 語言 (真實版)
LDI R1, 0x3F	R1 = 0x3F;	(*(unsigned char *)0xFFFFFF00) =

STB R1, [0xFFFFFF00]	[0xFFFFFF00] = R1	0x3F;

在範例 11.4 當中，組合語言的部分並不難，其原理是利用 LDI R1, 0x3F 指令，先將 0x3F 的數值載入到 R1 暫存器當中。然後再利用 STB R1, 0xFFFFFF00 指令，將 R1 的內容 (0x3F) 存入到七段顯示器的記憶體映射位址 0xFFFFFF00 中。如此，即可讓該七段顯示器顯示出數字 0。

然而，上述組合語言程式對 CPU0 而言，卻是錯誤的做法，原因是 CPU0 的 LD 與 ST 指令，其格式中的位移 Cx 部分只有 16 個位元。因此，最大的定址範圍是 0 到 65535，也就是十六進位的 0x0000000 到 0x0000FFFF 而已。然而， STB R1, 0xFFFFFF00 這個指令，其範圍已超出 Cx 可定址範圍，因此，必須加上某些暫存器當作基底。為了解決定址範圍的問題，我們將範例 11.4 改寫成範例 11.5，利用 R8 當作 STB 指令的基底，以解決 Cx 範圍過小的問題。

範例 11.5 讓 M0 的七段顯示器顯示數字 0 的程式 (絕對定址版-正確示範)

組合語言	C 語言 (對照版)	C 語言 (真實版)

LD R8, IO_BASE
LDB R1, [0x3F]
STB R1, [R8+0x00]
IO_BASE WORD 0xFFFFFF00
	R8=IO_BASE
	R1 = [0x3F];
	[R8+0x00] = R1
	IO_BASE = 0xFFFFFF00
		(*(unsigned char *)
		0xFFFFFF00)= 0x3F;

上述的組合語言程式即可達成記憶體映射輸出的功能，並不難，是嗎？

假如您尚未使用 C 語言寫過單晶片或嵌入式系統上的程式，那麼，範例 11.4 的 C 語言部分反而顯得匪夷所思了，筆者有必要在此多作解釋。

首先，(unsigned char *) 0xFFFFFF00 的意思是，將 0xFFFFFF00 這個數字，型態轉換為記憶體位址。接著，最前面的 * ，則代表要存取這個記憶體位址的內容。因此，整個指令的意義就是，將 0x3F 指定給 0xFFFFFF00 這個 unsigned char 型態的指標變數當中，也就是將 0xFFFFFF00 記憶體位址的內容設定為 0x3F 了。

當然，這樣的程式真的很難讀懂。還好，C 語言當中有 #define 這個巨集指令，可以讓我們將上述程式改寫，以利程式閱讀者理解。範例 11.6 就顯示了這個修改的過程，將直接存取的方式改用定義的方式，這會讓程式的可讀性變得較高，也會更加容易維護。

範例 11.6 讓 M0 的七段顯示器顯示數字 0 的程式 – 修改後容易讀的版本

C 語言

```
#define BYTE unsigned char
#define SEG7_REG (*(BYTE*) 0xFFFFFF00)
SEG7_REG = 0x3F;
```

上述範例可說是使用 C 語言實作記憶體映射寫入的最簡單版本，但是卻有可能導致錯誤，這個錯誤主要是編譯器的最佳化動作所造成的，我們將在後續進行解說。

讓我們將範例 11.6 改寫為正確的版本，其方法是加上 volatile 這個關鍵字，將程式修改為範例 11.7 的形式，以告知編譯器該變數是揮發性的 (volatile)，要求編譯器不可以對該變數進行最佳化的動作，這是規格 ISO C 99 所定義的語法。

範例 11.7 讓 M0 的七段顯示器顯示數字 0 的程式 – 使用 volatile 關鍵字

C 語言

```
#define BYTE unsigned char
#define SEG7_REG (*(volatile BYTE*) 0xFFFFFF00)
SEG7_REG = 0x3F;
```

舉例而言，在範例 11.8 當中，我們在 while 迴圈中使用 inBit 這個變數，但是卻在中斷服務函數中修改了這個變數的值，這樣的程式看來並沒有甚麼大問題，但卻會導致編譯器在最佳化時產生錯誤的組合語言碼。

在範例 11.8 (a) 的 C 語言當中，編譯器看到 inBit 在 while 迴圈當中並沒有被改變，因此就可能認為 inBit 可以事先被載入到暫存器 R1 當中，而不會造成任何問題，於是就利用最佳化的機制，將 (b) 欄中將 inBit 載入到 R1 的動作提出到迴圈之外，因而導致該迴圈永遠不會結束，成為無窮迴圈。

範例 11.8 一個未使用 volatile 關鍵字而造成錯誤的範例

(a) C 語言

```
static int inBit=0;

int main() {
   ...
   while (1) {
      if (inBit) dosomething();
   }
}
```

(b) 組合語言

```
            LD R1, inBit



LOOP:
            CMP R1, R0
            JNE LOOP
```

// 中斷服務常式.
void ISR() {
  inBit=1;
}
	
	ISR:
	  STI inBit, 1
	  RET
	inBit WORD 0

但是，在範例 11.8 (b) 的組合語言中，即使 ISR() 被觸發，程式也不會離開 LOOP 迴圈了，因為該組合語言檢查的是 R1 而非 inBit。

在範例 11.9 當中，我們將 static int inBit=0 改為 volatile static int inBit=0，關鍵字 volatile 告訴編譯器 inBit 是揮發性的，於是編譯器就不會將 inBit 事先載入，而是在需要的時候才載入到 R1 進行比較。因此，一但中斷發生呼叫 ISR() 之後，inBit 就會被設定為 1，於是程式檢查到 R1 = 1 之後就會離開 LOOP 迴圈，而不會造成無窮迴圈的情況了。

範例 11.9 一個使用 volatile 關鍵字而讓中斷機制正確運作的案例

C 語言	組合語言

volatile static int inBit=0;
 	
int main() {
  ...
  while (1) {
  if (inBit) dosomething();
  }
}
 	
// 中斷服務常式.
void ISR() {
  inBit=1;
}
	
	…
	LOOP:
	  LD R1, inBit
	  CMP R1, R0
	  JNE LOOP
	…
	
	ISR:
	  STI inBit, 1
	  RET
	inBit WORD 0

如果我們要為 M0 撰寫輸入程式，也只要利用記憶體存取指令即可。方法不難，請讀者參考圖 11.2 與表格 11.1，會發現可以從 0xFFFFFF01 與 0xFFFFFF02 兩個記憶體位址中，取出鍵盤的按鍵值。但是由於總共有 16 個按鍵，分為兩個 byte，因此，應分兩次的讀取，分別儲存在兩個 byte 中。然而，若要方便使用，則可將兩個 byte 合併後放入一個暫存器當中，將會比較容易使用。因此，在範

例 11.10 的組合語言版當中，這兩個 byte 被合併後放入暫存器 R3 當中，以便後續處理使用。

範例 11.10 讀取 M0 鍵盤暫存器的組合語言程式

組合語言版	C 語言 (對照版)

LD R8, IoBase
LDB R1, [R8+1]
LDB R2, [R8+2]
SHL R2, 8
OR R3, R1, R2
…
IoBase WORD 0xFFFFFF00
	R8 = IoBase;
	R1 = [R8+1];
	R2= [R8+2];
	R2 = R2 << 8;
	R3 = R2 | R1;
	…
	int IoBase= 0xFFFFFF00;

在 C 語言當中，要儲存 16 個位元的資料，可以利用 unsigned short 型態的變數，這是因為 unsigned short 占據兩個 byte，而且是屬於無正負號的 16 位元整數型態。

由於 unsigned short 的宣告實在太長了，因此，在 C 語言當中，常常會用 #define UNIT16 unsigned short 這樣語句定義為較短的 UNIT16 寫法，如範例 11.11 所示，如此會讓程式更加清楚

範例 11.11 讀取 M0 鍵盤暫存器的 C 語言程式

C 語言 (真實版)
#define BYTE unsigned char #define UINT16 unsigned short #define KEY (*(volatile UNIT16*) 0xFFFFFF01)

一旦取得按鍵資料後，即可檢查看看哪些按鈕被按下。舉例而言，若我們想知道對應到數字 5 的按鈕是否被按下，那應該怎麼做呢？

由於在 M0 的架構圖 11.2 當中，數字 5 按鈕的標示為 K5，而 K5 在硬體對映手冊 (表格 11.1) 中，對應到 IO_REG1 (0xFF01) 中的位元 5。因此，只要檢查位元 5 是否為 1 ，就可以知道該按鈕是否被按下了，範例 11.12 即分別示範了如何以組合語言與 C 語言檢查按鍵是否被按下的程式片段

範例 11.12 檢查按鍵 5 是否被按下的程式片段

組合語言版	C 語言 (對照版)	C 語言 (真實版)
LD R8, IoBase	R8 = IoBase	UNIT16 isK5hit=KEY&0x20;

LD R3, [R8+1]
LDI R7, 0x20
AND R4, R3, R7
	R3 = [R8+1]
	R7 = 0x20
	R4 = R3 & R7	

在範例 11.13 當中，更是利用這種檢查方法，以決定是否要在七段顯示器上顯示數字 5，於是這個程式，透過記憶體映射的方式，將輸入與輸出結合再一起了。

範例 11.13 以程式檢查按鍵 5 是否被按下，若是，則顯示數字 5

組合語言版	C 語言 (對照版)	C 語言 (真實版)

…
LD R8, IoBase
LD R3, [R8+1]
LDI R7, 0x20
AND R4, R3, R7
CMP R4, 0
JNE L2
LDB R1, 0x76
STB R1, [R8+0x00]
L2:
…
IoBase WORD 0xFFFFFF00
	…
	R8=IoBase
	R3=[R8+1]
	R7= 0x20
	R4 = R3 & 0x20;
	If (R4 != 0)
	goto L2;
	R1 = 0x76;
	[R8+0x00] = R1;
	L2:
	…
	int IoBase= 0xFFFFFF00;
		…
		UNIT16 isK5hit=key&0x20;
		If (isK5hit != 0)
		SEG7_REG = 0x76;
		…

利用此種方法，我們就可以寫出一個完整的程式，讓 M0 實驗板在數字按鍵被按下時，於七段顯示器當中顯示對應的數字，我們將在下一節中說明這個程式的寫法。

在本節中，我們介紹了兩種存取輸出入裝置的方式，組合語言可以使用『專用指令』或『記憶體映射』的方式，進行輸出入的動作，而 C 語言則只能用記憶體映射的方式進行輸出入，這也是為何『記憶體映射』的方式越來越受歡迎的原因之一。

11.2 驅動程式

在現代的電腦中，通常周邊裝置都會附有驅動程式，對使用者而言，驅動程式是一個需要安裝的軟體，而對於程式設計師而言，驅動程式則是用來控制特定輸出入裝置的程式。

通常，對於一個周邊裝置，程式設計人員就會寫出一組驅動程式，這組程式負責設定周邊裝置，並進行低階的輸出入動作，在本節中，我們將說明驅動程式的撰寫方式。

對於沒有作業系統的電腦而言，驅動程式只是一堆控制特定輸出入裝置的程式，驅動程式的撰寫者必須負責設計這些程式，將輸出入的功能包裝成函數，然後其他的程式設計人員只要呼叫這些函數進行輸出入即可。如此，就不需要讓每個人都透過低階的指令直接控制輸出入裝置，讓專案得以順利的分工與撰寫。

舉例而言，如果我們針對 M0 電腦中的七段顯示器撰寫驅動程式，那麼，我們可以利用範例 11.14 中的 seg7_show() 函數對該裝置的輸出動作進行包裝，該函數就是一個最簡單的驅動程式。

範例 11.14 M0 電腦的驅動程式 (七段顯示器+鍵盤)

C 語言程式檔 (driver.h)	說明
#define BYTE unsigned char #define UINT16 unsigned short #define BOOL unsigned char #define SEG7_REG (*(volatile BYTE*)0xFFFFFF00) #define KEY_REG1 (*(volatile BYTE*) 0xFFFFFF01) #define KEY_REG2 (*(volatile BYTE*) 0xFFFFFF02) #define KEY (KEY_REG2 << 8 | KEY_REG1)	定義 BYTE 型態 定義 UNIT16 型態 定義 BOOL 型態 七段顯示器的映射位址 鍵盤暫存器的映射位址 七段顯示器的映射位址
C 語言程式檔 (driver.c)	說明
#define BYTE seg7map[]={ /*0*/ 0x3F, /*1*/ 0x18, /*2*/ 0x6D, /*3*/ 0x67, /*4*/ 0x53, /*5*/ 0x76, /*6*/ 0x7E, /*7*/ 0x23, /*8*/ 0x7F, /*9*/ 0x77 };	七段顯示器的顯示表，例如：顯示 0 時 SEG_G 應熄滅，其他應點亮，因此應顯示二進位的 00111111，也就是 0x3F。
#define char keymap[]={'1', '2', '3', '+', '4', '5', '6', '-', '7', '8', '9', '*', '#', '0', '?', '/'};	keymap 是鍵盤的字元地圖，在 keyboard_getkey() 中可用來查出對應字元。

// 七段顯示器驅動程式
void seg7_show(char c) {
SEG7_REG = map7seg[c-'0'];
}
	在七段顯示器中輸出 b 數字

// 鍵盤驅動程式
char keyboard_getkey() {
  UNIT16 key = KEY;
  for (int i=0; i<16; i++) {
  UNIT 16 mask = 0x0001 << i;
  if (key & mask !=0)
  return keymap[i];
  }
  return 0;
}
 	
BOOL keyboard_ishit() {
  return (KEY != 0)
}
	取得按下的鍵
	取得按鍵暫存器
	從 K0 開始掃描，
	
	看看到底哪個鍵被按下
	傳回第一個被按下的鍵
	(假設：不會同時有兩個鍵被按下)
	
	檢查是否有按鍵按下

範例 11.14 中的 keyboard_getkey()、keyboard_ishit() 等函數，實作了 M0 的鍵盤驅動程式。其中，keyboard_ishit() 可用來檢查是否有按鍵被按下，而 keyboard_getkey() 則可用來取得所按下的字元。

範例 11.15 M0 電腦的主程式 (按下數字鍵後顯示在螢幕上)

C 語言程式 (driverTest.c)	說明

#include <driver.h>
 	
int main() {
  while (1) {
  while (!keyboard_ishit()) {}
  char key = keyboard_getkey();
  if (key >='0' && key <='9')
  seg7_show(key);
  }
}
	引用 driver.h
	
	主程式開始
	無窮迴圈
	  等待鍵盤被按下
	  取得按鍵
	  檢查是否為數字鍵
	顯示數字於七段顯示器

在上述範例中，我們已經說明了撰寫驅動程式的方法，我們針對 M0 電腦的鍵盤與七段顯示器撰寫了兩組『驅動程式』，分別『驅動』這兩個裝置，我們希望透過這個範例可以清楚的說明輸出入程式的設計原理。

11.3 輪詢機制

輸出入系統的設計是嵌入式系統的主要工作之一，在現代的電腦中，進行輸出入的方法有兩種，第一種是採用輪詢的機制，輪流詢問各個輸出入裝置。第二種是採用中斷的機制，由裝置主動通知 CPU 輸出入的狀況。我們將在下列兩節當中分別探討這兩種機制。

在前一節當中，我們已經介紹過輪詢輸出入的實際案例，但是，設計不良的輪詢程式可能讓整個系統陷入無窮迴圈，或者讓其他程式難以順利執行。因此，嵌入式系統通常會使用一個主要的輪詢迴圈，以訊息傳遞 (Message Passing)的方式控制程式的執行順序，這是嵌入式系統常見的一種程式設計模式 (Design Pattern)。

採用訊息傳遞的輪詢機制，可以在整個系統的最上層，撰寫一個大迴圈 (通常是一個無窮迴圈)，這個迴圈不斷的詢問各個裝置的狀態，一旦發現輸出入裝置有資料進入或有狀態改變時，就呼叫對應的函數進行處理。這也是『輪詢』一詞的由來，因為該程式輪流詢問各個裝置是否有資料進來。

範例 11.16 顯示了上層的訊息傳遞架構，該程式使用輪詢機制，利用 msg 變數傳遞訊息，然後利用 mouse, keyboard 等變數記錄各裝置的輸入與狀態。透過這種方式，嵌入式系統也可以管理許多輸出入程式，而不容易出現錯誤。

範例 11.16 採用訊息傳遞的輪詢機制-

MessagePassing.c

```
#include <MessagePassing.h>
msg_t msg;
keyboard_t keyboard;
mouse_t mouse;

int main() {
   while (1) {
      if (getMessage(&msg))
         processMessage(&msg);
   }
}

void processMessage(msg_t *msg) {
   if (msg->source == KEYBOARD) { ...
```

MessagePassing.h

```
#ifndef MESSAGE_PASSING_H

#define KEYBOARD 1
#define MOUSE 2

typedef struct {
int source;
} msg_t;

typedef struct {
char key;
} keyboard_t;

typedef struct {
```

```
    } else if (msg->source ==MOUSE) { ...
    }
}

int getMessage(msg_t *msg) {
    if (keyboardHit()) {
        msg->source = KEYBOARD; …
    } else if (mouseHit()) {
        msg->source = MOUSE; …
    }
    return 0;
}

int keyboardHit() {…}
int mouseHit() {…}
```

```
int x, y;
} mouse_t;

extern msg_t msg;
extern keyboard_t keyboard;
extern mouse_t mouse;

void processMessage(msg_t *msg);
int getMessage(msg_t *msg);

int keyboardHit();
int mouseHit();

#endif
```

如果採用上述的輪詢方式，那麼，確實可以不需要一個作業系統，因為輪詢迴圈會扮演協調角色，所有的程式都必須按照這種規定進行撰寫，在偵測到輸入訊息時才進行取得的動作，如此就不會有程式因為輪詢迴圈而霸佔了整個 CPU 的執行權，整個系統因此而可以順暢的運作。

但是，在範例 11.16 的架構下，只要有一個程式不按照規矩辦事，例如利用輪巡迴圈進行輸出入動作，那麼整個系統可能就會陷入崩潰的命運，這也正是輪詢機制的缺點之一。

輪詢機制的另一個缺點，是浪費了許多時間在輪流詢問的動作上，這對執行效能有不利的影響。因此，現代的 CPU 通常會支援中斷的機制，使用中斷機制，除了讓電腦更有效率之外，還可以透過中斷機制實作作業系統中的行程切換系統，讓程式得以在互不干擾的情況下，有效率的使用輸出入裝置，在下一節當中，我們將會說明中斷機制的原理。

11.4 中斷機制

中斷機制是由輸出入裝置，利用中斷訊號，主動回報輸出入裝置情況給 CPU 的一種技術。這種技術必須依靠硬體的配合，當輸出入裝置想要回報訊息時，可透過匯流排，傳遞中斷訊號給 CPU。此時，CPU 會暫停目前正在執行的程式，跳到對應的中斷向量上，該中斷向量內會包含一個跳向中斷函數的指令，讓 CPU 開

始執行該中斷函數。

CPU0 的中斷機制

中斷機制的設計必須仰賴 CPU 的配合，舉例而言，假如我們為 CPU0 加入中斷控制線，並且加入中斷控制電路，就可以在 CPU0 上實作中斷機制。

當裝置需要回報訊息給 CPU0 時，會將中斷代號放入到中斷控制線上，此時，CPU 就會根據此中斷代號，跳到對應的中斷向量位址中，引發中斷機制。

範例 11.17 顯示了一個 CPU0 的中斷設定程式，這個程式會被燒錄到 CPU0 的中斷向量位址 0x0000 開頭之處，當 CPU0 接收到中斷訊號時，就會跳到對應的位址中。舉例而言，假如 CPU0 收到代碼 4 的中斷請求訊號時，就會跳到記憶體位址 0x000C 之處，然後再跳轉到 IrqHnd 這個程式區，開始處理軟體中斷。

範例 11.17 CPU0 的中斷向量

記憶體位址	中斷向量		說明
	InterruptVector :		中斷向量開始
0000	JMP	ResetHandler	1. 重開機 (Reset)
0004	JMP	Unexpected	2. 非預期中斷 (Unexpected)
0008	JMP	SwiHnd	3. 軟體中斷 (Software Interrput)
000C	JMP	IrqHnd	4. 中斷請求 (Interrupt Request)

範例 11.18 顯示了 CPU0 的中斷處理函數，在這個程式中，由於未對 Unexpected 中斷進行處理，因此，乾脆讓 Unexpected 中斷進入無窮迴圈，在該中斷出現時直接當機，以免產生不可預期的情況，這是嵌入式系統常見的一個技巧。

範例 11.18 CPU0 的中斷處理程式 (組合語言)

中斷處理的呼叫端 (組合語言)	說明
Unexpected:	未預期中斷
JMP Unexpected	不處理、無窮迴圈
SwiHnd:	軟體中斷
PUSH {R1..R14}	保留暫存器 R1..R14
CALL CSwiHandler	跳到 CSwiHandler 函數
POP {R14..R1}	恢復暫存器 R1..R14
RET	處理完後返回原程式
	…

IrqHnd: PUSH {R1..R14} CALL CIrqHandler POP {R14..R1} RET …	中斷請求 保留暫存器 R1..R14 跳到 CIrqHandler 函數 恢復暫存器 R1..R14 處理完後返回原程式 …

在 CPU0 的組譯器中，PUSH {R1..R14}這樣的指令是一種簡便的寫法，實際上會被展開為 PUSH R1, PUSH R2, …., PUSH R14，同樣的，POP {R14..R1} 也會被展開為 POP R14, POP R13, … POP R1。

在範例 11.18 中處理未定義事件 (UndefHnd) 中斷時，首先利用 PUSH {R1..R14} 保留除了 PC 之外的暫存器。然後，就利用 CALL CUdefHandler 指令，跳到 CUdefHandler 函數中，以處理未定義事件中斷。等到處理完後，再利用 POP {R1..R14} 恢復暫存器的內容。

範例 11.19 顯示了中斷處理函數 CIrqHandler() 的 C 語言程式，我們可以利用連結器將範例 11.19 與範例 11.18 連結在一起，讓範例 11.18 的 CALL CIrqHandler 指令可以順利的連結到 C 語言的 CIrqHandler() 函數上。

範例 11.19 CPU0 的中斷處理函數 (C 語言)

	中斷處理函數 (C 語言)	
1	void CIrqHandler(void){	軟體中斷的處理函數
2	int id = rINT;	取得中斷代號 id
3		rINT 是中斷暫存器
4	if ((id>=0)&&(id<MAX_IRQ)) {	如果是合理的中斷代號
5	if(irq_table[id].handler){	如果請求表中有函數
6	irq_table[id].handler();	呼叫該函數
7	} else {	否則
8	error("IRQ 函數尚未設定!");	處理錯誤
9	}	
10	}	
11	}	
12		
13	void CSwiHandler(void) {	
14	…	
15	}	
16	…	

在 CPU0 的中斷向量中，IRQ 是由硬體觸發的中斷，而 SWI 則是由軟體觸發的中斷。例如，鍵盤、滑鼠、時間中斷等，是由硬體引發的，屬於 IRQ 中斷。而作業系統的系統呼叫，像是開檔、讀檔等，皆可以透過軟體中斷 SWI 達成。

由於 IRQ 是硬體引發的中斷，因此，必須在電腦啟動後就先設定好 irq_table，如此，在硬體中斷被觸發時 (例如，鍵盤被按下時)，才能在 irq_table[id].handler 中找到函數指標，以順利呼叫硬體中斷函數。

要設定這些函數，可以透過註冊的機制，將函數指標填入到中斷註冊表 (irq_table) 中，範例 11.20 顯示了一個中斷註冊函數的實作範例。其中，register_irq() 函數可用來註冊中斷函數，而 unregister_irq() 則可用來取消註冊。

範例 11.20 中斷函數的註冊函數(C 語言)

	中斷處理函數 (C 語言)	
1	void register_irq(unsigned int id,	中斷註冊函數，參數為
2	void (*handler)(void)) {	代號(id), 函數(handler)
3	if ((id >= 0) && (id < MAX_IRQ)) {	檢查中斷代號是否合理
4	if (!irq_table[id].handler) {	如果該中斷尚未定義
5	irq_table[id].handler = handler;	設定該中斷為 handler
6	}else{	否則
7	error("IRQ 函數重複定義!");	處理錯誤
8	}	
9	}	
10	}	
11		
12	void unregister_irq(unsigned int id) {	取消中斷註冊
13	irq_table[id].handler=NULL;	清除該中斷
14	}	

一但有了這些中斷設定之後，我們就可以透過註冊的方式，先設定對應的中斷函數，然後在程式的執行過程中，如果有中斷發生時，該函數就會被執行。舉例而言，時間中斷是一個很有用的硬體中斷，假如我們希望能讓 M0 電腦中的七段顯示器能夠反覆的從 0 數到 9，那麼，就可以利用先註冊一個時間中斷函數 ISR_TIMER() [1]，然後在該函數中利用全域變數 count 計數，接著將該數值輸出到七段顯示器當中即可。

[1] ISR 是 Interrupt Service Routine 的簡稱，也就是中斷服務函數。

範例 11.21 利用時間中斷讓 M0 電腦反覆從 0 數到 9

	主程式 (C 語言)	說明
1	#include <driver.h>	引用 driver.h
2		
3	int count = 0;	計數器，從 0 開始不停向上數
4		
5	void timer_ISR() {	時間中斷的服務函數
6	int num = count % 10;	中斷時會執行此函數
7	seg7show(num);	顯示 count 除 10 後的餘數
8	count ++;	繼續將 count 向上數
9	}	
10		
11	int main() {	主程式
12	register_irq(TIMER_IRQ_ID, timer_ISR);	註冊 timer_ISR 為時間中斷函數
13	enable_irq();	啟動中斷機制
14	while(1) { }	無窮迴圈
15	}	

如果讀者仔細閱讀範例 11.21，應該會發現一件奇怪的事，第 13 行的 while (1) {} 實際上是一個無窮迴圈，因此，該程式在啟動中斷機制之後，就進入了一個無窮迴圈。因此，理論上這個程式應該會直接當機，甚麼事都不會做才對，但是，由於中斷機制是由硬體直接驅動的，因此，每當時間中斷發生時，timer_ISR() 函數仍然會被呼叫，因此，七段顯示器上就會顯示下一個數字。透過中斷機制，七段顯示器會反覆的從 0 數到 9，造成類似電子錶秒數跳動的感覺。

中斷機制與輪詢是嵌入式系統中經常使用的兩種輸出入方式，中斷機制由於仰賴硬體的配合，因此，只能在支援中斷的處理器中實現。目前，除了非常低階且原始的單晶片處理器之外，大部分的處理器都能支援中斷機制，像是 ARM、IA32、MIPS 等處理器，都能支援中斷機制。

有了中斷機制，作業系統才有可能實現真正的多行程執行環境，作業系統可以利用時間中斷，收回處理器的控制權，以便安排另一個程式開始執行。接著，我們先來理解如何讓電腦啟動，以及如何設定中斷向量等主題，這將是下一節啟動程式的內容。

11.5 啟動程式

啟動程式就是一個相當特殊的程式，其功能是建立電腦的程式執行環境，讓其他程式得以順利執行。撰寫啟動程式的設計師，必須依靠各種感覺器官，去感覺程式是否正常，因為在啟動時，包含螢幕在內的各種裝置不見得能正常運作 (甚至，許多嵌入式系統根本就沒有螢幕)，因此、可能會使用『嗶一聲』等原始的方法，代表程式還在正常的執行，這也是嵌入式系統常用的方法。

電腦開機後，第一個被執行的記憶體位址，稱為啟動位址，啟動程式的第一個指令，必須被正確的燒錄到該位址中，才能正確的啟動。然而，程式設計師對啟動程式通常會感到迷惑。因為啟動程式必須在電腦一開機時就存在記憶體中，因此，只能將啟動程式放到 ROM 或 FLASH 等永久性儲存體當中。如果將啟動程式放到揮發性記憶體，像是 DRAM 或 SRAM 中，那麼，當使用者關閉電源後重新開機時，啟動程式將消失無蹤，電腦也就無法順利啟動了。

對嵌入式系統而言，啟動程式中除了包含機器指令之外，也會包含資料區域，像是 .data 段與 .bss 段。對於 .data 段而言，這些資料一開始會被儲存在 ROM 當中，但是，在啟動之後必須被搬入到 RAM 當中，否則，程式將無法修改這些資料。

因此，嵌入式的啟動程式會將資料區從 ROM 搬移到 RAM，才能讓這些具有初值的變數進入可修改狀態。因此，啟動程式必須將自己先從 ROM 搬到 RAM 之後，才能開始執行其主要功能。

啟動程式必須建立程式的執行環境，讓其他程式得以順利執行，其主要任務包含下列四項：

1. 設定中斷向量，啟動中斷機制。
2. 設定 CPU 與主機板的各項參數，讓 CPU 與主機板得以進入正確的狀態。
3. 將存放在永久儲存體中的程式與資料搬到記憶體中。
4. 設定高階語言的執行環境，包含設定堆疊 (Stack)、堆積 (Heap) 等區域。

為了更詳細的說明啟動程式的功能，我們將再度以基於 CPU0 處理器的 M0 電腦為例，以範例的方式說明啟動程式的設計方式。

M0 電腦的記憶體配置

M0 電腦擁有 16K 的 ROM 與 48K 的 RAM，ROM 的存取位址為 0x0000~0x3FFF，而 RAM 的存取位址為 0x4000~0xFFFF。如圖 11.3 (a) 所示。

在啟動時，ROM 當中已經燒錄有整個系統的程式與資料，其中，啟動程式段 (*.stext) 被燒錄在 ROM 開頭的 0x0000 區域。M0 電腦在重開機時，會從啟動位址 0x0000 開始執行，因此，啟動程式當中的重開機中斷必須被燒錄在 0x0000 的位址上，如此才能順利開機。

由於 M0 電腦是嵌入式系統，沒有硬碟，所有程式都被燒錄在 ROM 當中，因此，除了啟動程式之外，所有的程式區 (*.text)、資料區 (*.data) 都被燒錄在 ROM 當中。但是，BSS 段 (*.bss) 並不需要燒錄，因為這個區段的變數沒有設定初值，燒錄或不燒錄都無所謂，只要能記住 BSS 段的大小即可。所以，BSS 段被放在 ROM 的最後部分，其空間可以一直延伸到 RAM 的區域，如圖 11.3 (b) 所示。

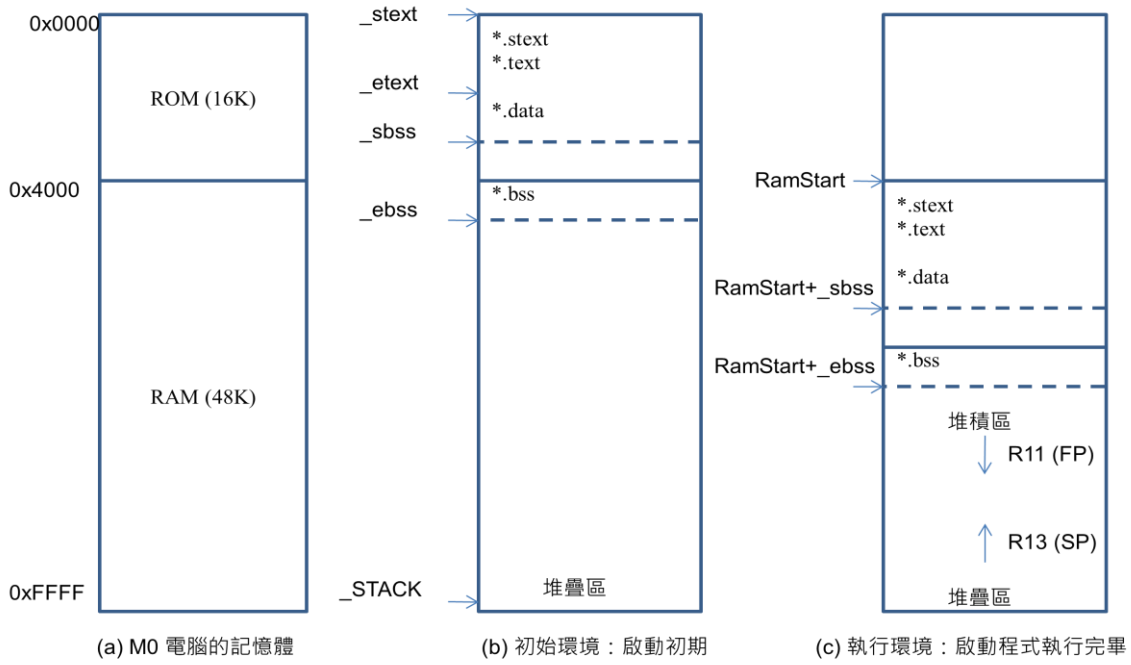

圖 11.3 M0 電腦的記憶體配置圖

另外，堆疊區被放置在 RAM 的最後部分，從 0xFFFF 開始向低位址的方向延伸，如此，堆疊區與堆積區可共用 RAM 的最後的一個區塊，只要兩者不發生重疊的情形即可。

根據圖 11.3 (b)，我們可以撰寫 M0 電腦的連結檔，如範例 11.22 的檔案 M0.ld 所示。其中，程式區塊 .text 0x0000 : { … } 代表程式區會從 0x0000 位址開始，

而 _stext = . 當中的點符號『.』代表目前位址，該指令要求連結器創建一個 _stext 符號，該符號的值為目前位址 (0x0000)。然後，開始將程式放入這個區域，首先是啟動程式 (*.stext)，接著是一般程式 (*.text)，然後，利用 『.=ALIGN(4)』 這個指令，讓點符號『.』所代表的目前位址，向後移動到 4 的倍數之處，以便進行 32 位元電腦的對齊動作。

接著，由於資料區塊 .data : {…} 沒有指定起始位址，因此會緊接在上一個區塊之後。然後，BSS 區塊 .bss: {…} 會跟在資料區塊之後。由於 BSS 段是未設初值的資料，因此，就算 BSS 段很大，無法被放入 ROM 當中也無所謂，只要能放入 RAM 區域當中即可。

範例 11.22 M0 電腦的連結檔

M0 電腦的連結檔 (M0.ld)	說明
SECTIONS	
{	
.text 0x0000 : {	程式段：起始位址為 0x00000000
_stext = .;	設定程式段起點 _stext 為目前位址
*(.stext)	插入所有的 .stext 段的目的碼
*(.text)	插入所有的 .text 段的目的碼
. = ALIGN(4);	以 4 byte 為單位進行對齊
_etext = .;	設定程式段終點 _etext 為目前位址
}	
.data : {	資料段：緊接在程式段之後
_sdata = .;	設定資料段起點 _sdata 為目前位址
*(.data)	插入所有的 .data 段的目的碼
. = ALIGN(4);	以 4 byte 為單位進行對齊
_edata = .;	設定資料段終點 _edata 為目前位址
}	
.bss : {	BSS 段：
_sbss = .;	設定 BSS 段起點 _sbss 為目前位址
*(.bss)	插入所有的.bss 段的目的碼
. = ALIGN(4);	以 4 byte 為單位進行對齊
_ebss = .;	設定 BSS 段終點 _ebss 為目前位址
}	
_end = .;	設定終點 _end 為目前位址

.stack 0xFFFF : {
_STACK = .;
}
}
	堆疊段：位址為 0xFFFF
	設定使用者堆疊起點為 0xFFFF

在這些區塊中，我們創建了 _stext, _etext, _sdata, _edata, _sbss, _ebss, 等符號，並設定其位址，以便在組合語言或 C 語言中可以取得這些符號的位址，提供啟動程式使用。

最後，我們設定堆疊區塊 .stack 0xFFFF : {…}，並將堆疊符號 _STACK 設定為該區塊的位址，也就是 0xFFFF，這樣，我們就能在啟動程式中正確的設定堆疊暫存器 R13 (SP) 的值了。

利用範例 11.22 的 M0.ld 連結檔，我們可以建構出如圖 11.3 (b) 的映像檔，然後燒錄到 M0 電腦的 ROM 之中。然後，我們必須撰寫啟動程式，利用該程式設定電腦的基本執行環境，讓後續的程式可以正確的執行。

M0 電腦的啟動程式

在 M0 電腦的啟動程式執行前，其記憶體映像就如同圖 11.3 (b) 的映像檔所示，ROM 區域燒錄有 1. 啟動程式 (*.stext) 2. 其他程式 (*.text) 3. 資料 (*.data) 等區塊。在啟動程式執行完畢後，會將所有的程式與資料都搬到 RAM 當中，如此，才能讓這些資料得以被寫入或修改，並且分配出 BSS 區段、堆積段 (heap) 與堆疊段 (stack) 等空間，這些空間可為 C 語言與組合語言提供一個良好的執行環境。

其中的資料段可以存放『具有初值的變數』，BSS 段可以存放『無初值變數』使用，堆疊段可以存放函數的『參數』、『返回點』與『區域變數』等[2]，而堆積段則可以儲存 C 語言當中的動態配置記憶體，像是 malloc 指令所需要的記憶空間。

必須注意的是，堆疊段是由高位址空間向低位址的方向增長的，而堆積段則是由低位址空間向高位址方向增長的，因此，整個系統在執行時期的記憶體配置會如圖 11.3 (c) 所示。M0 電腦啟動程式的主要功能，就是改造圖 11.3 (b) 的『初始環境』，以便形成如圖 11.3 (c) 的『執行環境』。

[2] 關於 C 語言函數呼叫的堆疊配置方式，請參考第 3 章的實務案例-『編譯器與副程式』一節。

範例 11.23 顯示了該啟動程式的組合語言部分，程式的開頭就是中斷向量表，包含了四個跳躍指令。接著，該程式定義了這些中斷的處理函數，包含重開機中斷(ResetHandler)、非預期中斷(Unexpected) 、軟體中斷 (SwiHandler)、中斷請求(IrqHandler) 等，其中，後三個中斷已經在 11.4 節中介紹過，在此不再重複敘述，我們將焦點放在啟動中斷的 ResetHandler 程式上。

範例 11.23 M0 電腦的啟動程式

	M0 電腦的啟動程式 (組合語言) 檔案：boot.s	說明
1	.global stext, main	
2	.global CSwiHandler	
3	.global CIrqHandler	
4		
5	RamStart EQU 0x4000	
6		
7	.section ".stext"	
8	InterruptVector :	中斷向量開始
9	JMP ResetHandler	中斷 1：重開機 (Reset) 啟動中斷
10	JMP Unexpected	中斷 2：非預期中斷 (Unexpected)
11	JMP SwiHandler	中斷 3：軟體中斷 (Software Interrput)
12	JMP IrqHandler	中斷 4：中斷請求 (Interrupt Request)
13		
14	Unexpected:	中斷 2：非預期中斷
15	JMP Unexpected	不處理，因此進入無窮迴圈
16		
17	SwiHandler:	中斷 3：軟體中斷
18	PUSH {R1..R14}	保存暫存器
19	CALL CSwiHandler	呼叫軟體中斷處理函數 (C 語言)
20	POP {R14..R1}	恢復暫存器
21	RET	返回原程式
22		
23	IrqHandler:	中斷 4：中斷請求
24	PUSH {R1..R14}	保存暫存器
25	CALL CIrqHandler	呼叫中斷請求處理函數 (C 語言)
26	POP {R14..R1}	恢復暫存器
27	RET	返回原程式
28		

```
29  ResetHandler:                       重開機處理程式
30      LDI R12, 0xD0                   設定狀態暫存器，同時禁止所有中斷
31
32  MoveToRam:                          將機器碼從 ROM(或 Flash)搬到 RAM
33      LD   R1, RamStart               從 0x0000 搬到 RamStart = 0x4000
34      LDI  R2, 0x0000
35  LOOP1:  LD   R3,[R0+R2]
36      ST   R3,[R1+R2]
37      ADD R2, R2, 1
38      CMP R2,R1
39      JNE  LOOP1
40  JumpToRam:
41      LD   R15, RamCinit              跳到 RAM 版本的 Cinit 標記中。
42
43  Cinit:
44      LD       R1, RamSbss            C 語言環境設定
45      LD       R2, RamEbss            首先清除 BSS 段
46  LOOP2:  CMP      R1, R2             設定 RAM 中 BSS 段的內容為 0
47      JNE      InitStacks
48      ST       R0, [R1]
49      JMP      LOOP2
50
51  InitStacks:
52      LD       SP, StackBase          堆疊初始化
53      LDI      R12, 0x00              允許中斷
54  MainLoop:
55      CALL     main                   無窮迴圈
56      JMP      MainLoop               進入 C 語言的主程式。
57
58  RamSbss WORD RamStart+_sbss
59  RamEbss WORD RamStart+_ebss
60  RamCinit WORD RamStart+Cinit
61  StackBase WORD _STACK
```

當 M0 電腦一開機後，會先執行位址 0000 的 JMP ResetHandler 指令，然後跳入 ResetHandler 標記的程式區。接著，啟動程式會使用 LDI R12, 0xD0 這樣一個指令，禁止所有中斷發生。但是，這個指令可能會令讀者費解，為何該指令會禁止中斷的發生呢？

這牽涉到 CPU0 的架構，請讀者參考的 CPU0 的狀態暫存器位元圖 (圖 11.4)。由於在 CPU0 當中，R12 就是狀態暫存器 SW，其中第 7, 6 兩個位元是 I, T 位元，代表中斷的禁止位元，因此，LDI R12, 0xD0 這樣的指令會使 I, T 兩個位元變成 0，於是禁止了中斷的發生，以免在重開機時還有中斷產生，讓開機程序無法順利完成。

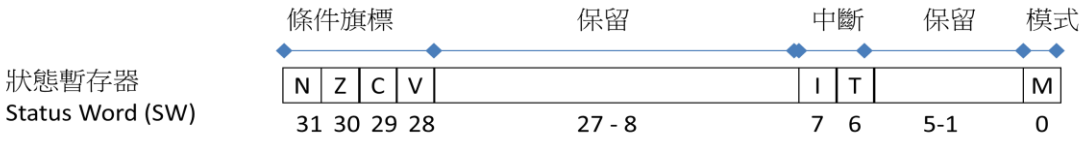

圖 11.4 CPU0 的狀態暫存器 R12 之位元圖

接著，在一般的 CPU 處理程序上，可能會有一連串的設定動作，例如設定暫存器的初值、清除快取、設定記憶體控制暫存器、設定時間中斷頻率等等。但是由於我們假設 M0 當中這些參數都是直接燒錄在電路當中，因此，不需要在開機時用程式設定，這可以省掉許多動作，但相對也缺少了許多彈性。

然後，在 MoveToRam 標記的程式區段中，程式會將原先位於 ROM 當中的程式與資料，原封不動的搬到 RAM 當中。然後，在 JumpToRam 標記的區塊中，利用 LD R15, RamCinit 這個指令設定程式計數器，讓程式從 ROM 跳入到 RAM 區域的 Cinit 標記中，開始執行 RAM 中的 Cinit 標記後的指令。

在 Cinit 標記的區段中，程式會先清除 BSS 區段為 0，然後利用 LD SP, StackBase 這個指令設定堆疊指標。接著，在 MainLoop 標記的區段中，使用 CALL main 呼叫 C 語言的主程式，完成整個啟動過程。

在啟動完畢之後，就可以進入 C 語言的主程式 main 當中，如果主程式如同上一節的範例 11.21 所示，那麼，M0 電腦的七段顯示器就會反覆的從 0 數到 9，如同電子錶般的跳動著。

11.6 系統整合

在一個嵌入式系統開發的過程當中，開發人員會建構出許多相關檔案，包含組合語言、C 語言程式、資料檔、連結檔等等，要能有效率的編譯與連結這些檔案，最好是使用專案整合工具 (例如 GNU 的 Make 工具)。

GNU 的 Make 是專案建置實相當常見的工具，在本節中，我們將說明如何利用 make 工具整合專案的建置過程，讓專案的建置自動化。當嵌入式專案開始時，最好能撰寫一個 Makefile 檔，整合專案中的所有檔案。

舉例而言，假如 GNU 已經針對 CPU0 開發了一組專用工具，其編譯器名稱為 cpu0gcc，連結器名稱為 cpu0ld。而且我們已經撰寫了 driver.h、driver.c、main.c, boot.s、M0.ld 等檔案，那麼，我們就可以撰寫如範例 11.24 的 Makefile 檔，以便對這些檔案進行專案編譯、連結的動作。

範例 11.24 M0 電腦的專案檔

專案檔：Makefile	說明
CC=cpu0gcc	使用 cpu0gcc 編譯器
LD=cpu0ld	使用 cpu0ld 連結器
OBJCOPY=cpu0objcopy	使用 cpu0objcopy
OBJS=driver.o boot.o main.o	目的檔列表
FLAGS=-I .	編譯參數 (-I .代表在目前路徑下搜尋 *.h 檔)
all : $(OBJS)	全部編譯連結
$(CC) -TM0.ld -o M0.o $(OBJS)	使用 M0.ld 連結，輸出 M0.o
$(OBJCOPY) -O binary -S M0.o M0.bin	將目的檔轉換為二進位檔
.c.o:	編譯 C 語言程式
$(CC) $(FLAGS) -c -o $@ $<	
.s.o:	編譯組合語言程式
$(CC) $(FLAGS) -c -o $@ $<	
clean:	清除上一次的輸出檔
rm *.bin *.o	

利用 Makefile 檔與 GNU make 工具，我們可以將所有程式編譯、連結成目的檔後，再透過 objcopy 工具將目的檔轉換成二進位檔 (像是範例 11.24 中的 M0.bin)，然後，就可以利用燒錄工具，將該二進位檔燒錄到嵌入式系統中，接著，按下重新開機鍵，看看程式是否能正常執行。

嵌入式系統在開發時，通常會先在測試板上開發，這樣可以方便開發人員撰寫與測試程式。開發人員可以利用個人電腦上的操作介面，像是 MS. Windows 上的『超級終端機』程式，利用簡易的通訊介面 (例如 UART)，透過 COM 連接埠與測試板溝通，利用指令或選單，將編譯完成的二進位檔傳送到測試板的 RAM 當中，以便執行該程式。

當開發人員想要將程式燒錄到測試板的 ROM (在測試板上通常是 Flash) 上的時候，通常必須透過 ICE (Integrated Circuit Environment) 裝置，將該二進位檔燒錄到測試板的 Flash 中。由於 Flash 是永久儲存體，電源關閉後資料仍然會存在，因此，當下次重開機時，就會透過該 Flash 中的程式啟動該嵌入式系統，於是該系統就變成了目標系統的原型機，可以在量產之前先提供給客戶使用，以確認功能是否正確，系統的運作是否正常等。因此，測試板在嵌入式系統的開發上是相當有用的，特別是針對系統開發人員而言。

嵌入式裝置是系統程式設計的一個進階領域，該領域整合了軟體、韌體與硬體的背景知識，是學習系統程式的絕佳戰場。對於有心學習系統程式的讀者而言，可以進一步參考嵌入式系統的專業書籍，或者購買嵌入式實驗板，學習期程式設計方法，相信會有更多的收穫。

11.7 實務案例：新華 Creator S3C2410 實驗板

為了說明嵌入式系統的開發過程，筆者使用新華電腦的 Creator S3C2410 實驗板作為範例，重點式的說明開發的流程。

該實驗板的範例主要架構在 Cygwin 環境下，舉例而言，如果我們想編譯其中的 LCD 這個範例，可以進入 /usr/var/creator/LCD 這個資料夾後，執行 make 指令，以便重建整個專案，範例 11.25 顯示了該建置過程。

範例 11.25 新華 Creator S3C2410 實驗板範例 LCD 的建置過程

在 Cygwin 中的編譯執行過程 (TIMER 範例)	說明
ccc@ccc-kmit2 /usr/var/creator/ LCD $ ls Makefile demo.c demo_ram.map demo_rom.map lcd.c common demo_ram.ld demo_rom.ld gnu	列出資料夾中的檔案
ccc@ccc-kmit2 /usr/var/creator/ LCD $ make clean rm *.bin *.axf *.o *.s	清除專案 (上次的輸出)
ccc@ccc-kmit2 /usr/var/creator/ LCD $ make /usr/local/bin/arm-elf-gcc -nostartfiles -g -I/usr/var/creator/ LCD/common -I/u	重建專案

…略… /usr/local/bin/arm-elf-gcc -Tdemo_ram.ld –Wl , -M, -Map=demo_ram.map -o "demo_ram.axf" demo.o sbrk.o driver.o irq.o mmu.o 2410slib.o lcd.o head_ram.o arm-elf-objcopy -O binary -S demo_ram.axf demo_ram.bin …略…	製作 demo_ram.axf 目的檔 將該目的檔轉為二進位的 demo_ram.bin

一但建置完畢後，我們就可以開啟 MS. Windows 中位於『開始/附屬應用程式/通訊/超級終端機』程式，以便將 demo_ram.bin 檔案，傳送到該實驗板中，以下是我們的操作過程。

首先，當超級終端機啟動後，按下實驗板的重開機按鈕，會進入圖 11.5 的起始畫面，這個畫面是由實驗板中一個預先燒錄的啟動程式，利用 UART 協定傳送給超級終端機所顯示出來的。此時，使用者可以選擇功能 1 – Download to Ram & Go 這個功能，以便將方才所建置的 demo_ram.bin 檔案上傳。

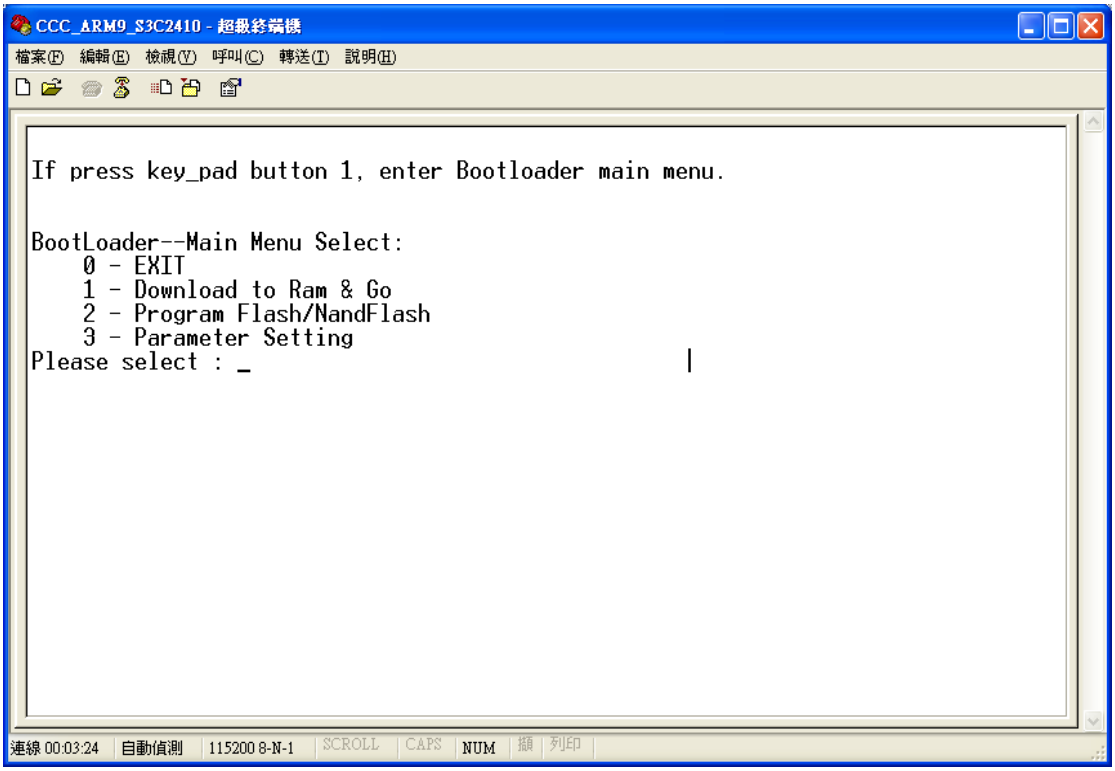

圖 11.5 透過超級終端機，顯示該實驗板的起始功能表

接著，我們必須按下『傳送/傳送檔案』的功能，然後選擇所要上傳的檔案 demo_ram.bin，接著，按下開始按鈕，確定選取該檔案。

圖 11.6 按下傳送功能，選取欲上傳的檔案 demo_ram.bin

接著，會顯示傳送檔案的對話框，我們必須將通訊協定設為 Zmodem，然後按下傳送按鈕，將檔案上傳並且開始執行。

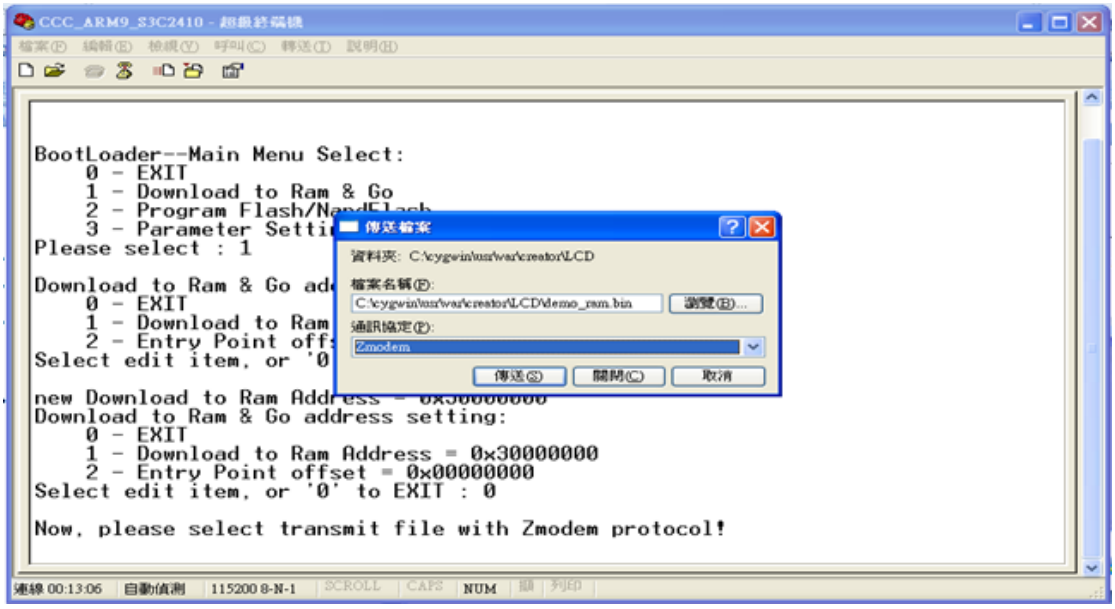

圖 11.7 選擇通訊協定為 Zmodem，按下傳送鈕，開始上傳

上傳完畢後，實驗板上預設的啟動程式就會直接執行該上傳程式，若一切正常，我們會在實驗板的 LCD 螢幕中看到該範例印出了 "Hello!" 的字串，這代表程式正確的被執行了。

透過這個範例，我們希望讀者能較為實際的感受到嵌入式系統的開發流程，如果您手上也有嵌入式的實驗板，也可以趁現在操作看看。當然，每一張實驗板的設

計會有所不同，操作的過程也會不同，您必須根據該實驗板製造商的說明文件進行操作，才能順利的執行這些程式，然後才能進一步開發自己的程式。

習題

11.1 請說明何謂專用輸出入指令？

11.2 請說明何謂記憶體映射輸出入？

11.3 請說明何謂中斷機制？中斷發生時程是會跳到哪裡呢？

11.4 請說明啟動程式應該做些甚麼事呢？

11.5 請寫出一個完整的 C 語言程式，讓 M0 實驗板在數字按鍵被按下時，於七段顯示器當中顯示對應的數字。

11.6 同上題，但是請以 CPU0 的組合語言撰寫。