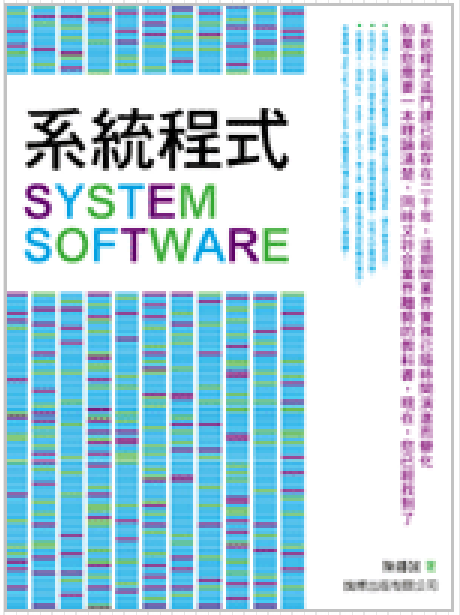
第 11 章、嵌入式系統

作者：陳鍾誠

旗標出版社



第 11 章、嵌入式系統

- 11.1　輸出入
- 11.2　驅動程式
- 11.3　輪詢機制
- 11.4　中斷機制
- 11.5　啟動程式
- 11.6　系統整合
- 11.7　實務案例：新華 Creator S3C2410 實驗板

為何探討嵌入式系統？

- 在嵌入式系統當中, 通常沒有作業系統可以使用
- 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

嵌入式系統的開發過程

- 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 2 進位執行檔。
- 再將這個 2 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。

11.1 輸出入

- 兩種輸出入的方式
  - 第一類是採用「專用的輸出入指令」
  - 第二類是使用「記憶體映射輸出入」

專用的輸出入指令

- 何謂專用的輸出入指令
  - 某些 CPU 指令集本身即具有輸出入專用指令

- 專用輸出入指令的範例
  - TD : 測式裝置 (Test Device)
  - RD : 讀取裝置 (Read from Device)
  - WD: 寫入裝置 (Write to Device)

專用的輸出入指令的運作方式

- 輸入
  - 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。
- 輸出
  - 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。

輸入的範例

- 假如輸入裝置 0x09 代表鍵盤, 則範例 11.1 可以從鍵盤中讀入一個字元。

▶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:`	inLoop 標籤
`  TD  inDev`	測試輸入裝置 0x09
`  JEQ inLoop`	若沒有輸入, 跳回到 inLoop 繼續測試
`  RD  R1, inDev`	讀取輸入值到暫存器 R1 當中
`  STB R1, key`	將讀到的值存入變數 key
`inDev BYTE 0x09`	
`key   RESB 1`	

輸出的範例

- 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 將會輸出字元 A 到螢幕上。

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

輪詢法 (Polling) 的範例

- 輪流詢問
  - 鍵盤
  - 滑鼠
  - 網路

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

記憶體映射輸出入

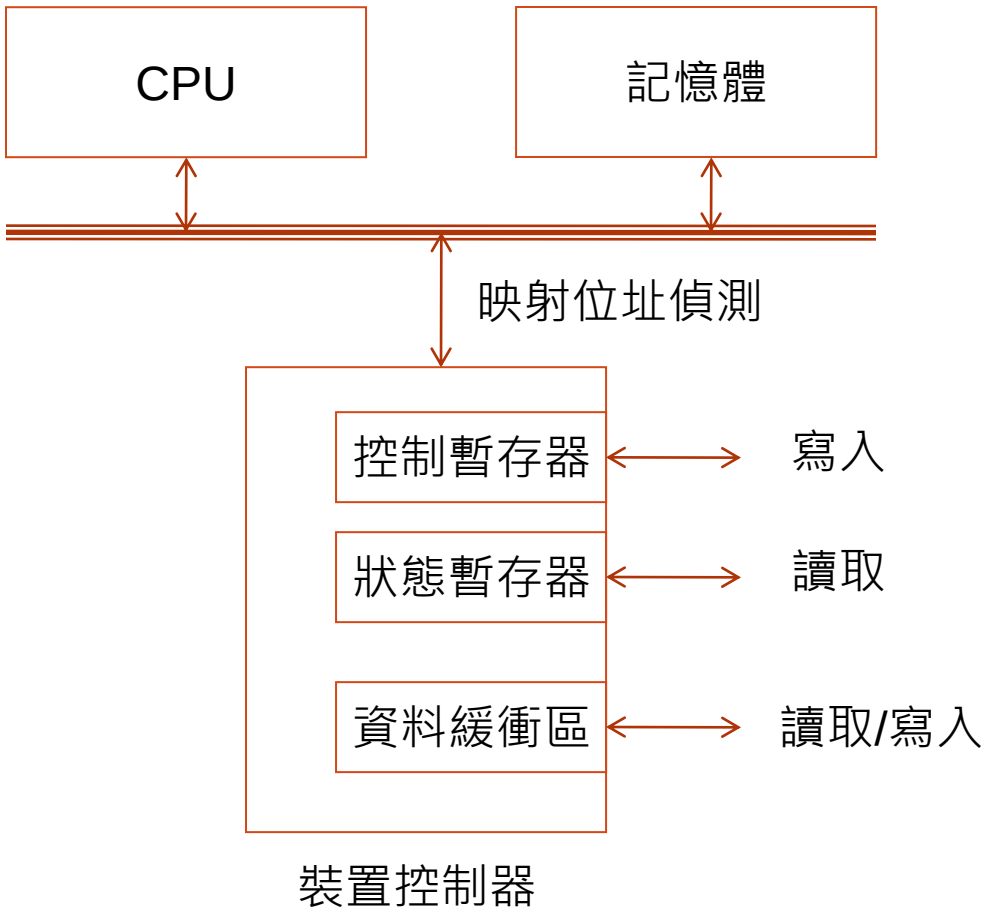
- 說明
  - 將周邊裝置視為記憶體的一部分
    - 一部分位址真正對應到記憶體
    - 另一部分則對應到周邊裝置
  - 利用記憶體存取指令進行輸出入的方法。

- 範例
  - 輸出：
    - 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。
  - 輸入：
    - 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。

記憶體映射輸出入的硬體設計

- 輸出入控制器
  - 記憶體映射通常是輸出入控制器的工作。
- 輸出
  - 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。
- 輸入
  - 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。

圖 11.1裝置控制器與記憶體映射機制

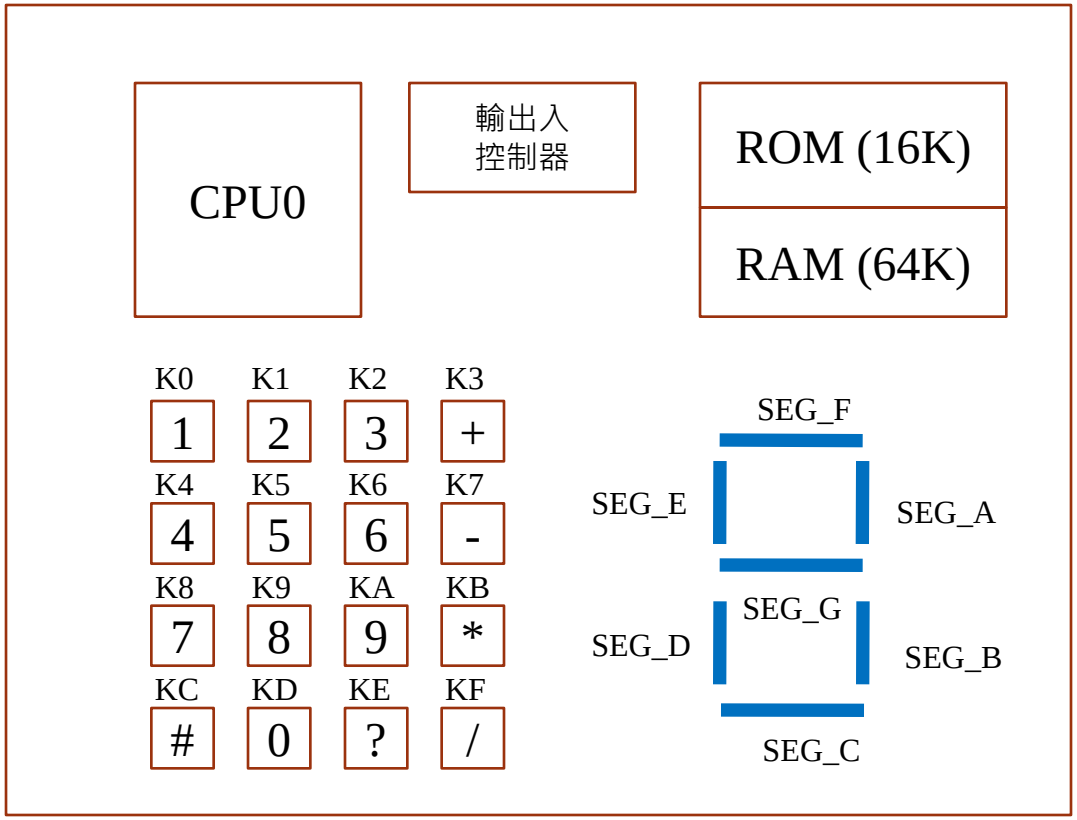


簡易電腦 M0

- M0 (Machine 0 的簡寫)
  - 為了說明記憶體映射輸出入而設計的電腦。
  - 使用 CPU0 做為處理器

- 架構
  - 是一種單晶片實驗板, 採用 CPU0 作為處理器
  - 包含 16 K 的 FLASH 。
    - 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。
  - 包含 64K 的 RAM

圖 11.2 簡易電腦 M0 的基本架構



M0 電腦中的記憶體映射區域

- M0 採用記憶體映射輸出入
  - 0xFFFFFF00 ：七段顯示器的映射區域
  - 0xFFFFFF01 與 0xFFFFFF02 ：鍵盤映射區

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 的輸出 – 七段顯示器

- 方法：將位元資料填入到七段顯示器的記憶體映射區，即可顯示該位元樣式。

- 範例：讓該七段顯示器顯示出字元 0

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 STB R1, [0xFFFFFF00]	R1 = 0x3F; [0xFFFFFF00] = R1	(*(unsigned char *)0xFFFFFF00) = 0x3F;

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)

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

使用 C 語言進行記憶體映射輸出 (簡潔版)

- 範例：讓該七段顯示器顯示出字元 0

- 簡潔的寫法
  - 寫法一：未加 volatile，可能會錯
    - (* (unsigned char *) 0xFFFFFF00) = 0x3F;
  - 寫法二：加上 volatile，正確無誤
    - (* (volatile unsigned char *) 0xFFFFFF00) = 0x3F;

使用 C 語言進行記憶體映射輸出 (易讀版)

- 範例：讓該七段顯示器顯示出字元 0

未加 volatile ，可能會因為最佳化而出錯

▶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;
```

加上 volatile ，正確

▶範例 11.7 讓 M0 的七段顯示器顯示數字 0 的程式 - 使用 volatile 關鍵字個裝置的程式

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

C 語言中 volatile 關鍵字的用途

- volatile 的功能：
  - 告訴編譯器該變數是揮發性的 (volatile), 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
  - (* (volatile unsigned char *) 0xFFFFFF00) = 0x3F;

▶範例 11.7 讓 M0 的七段顯示器顯示數字 0 的程式 - 使用 volatile 關鍵字個裝置的程式

C 語言

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

範例 11.8 一個未使用 volatile 關鍵字而造成錯誤的範例

▶範例 11.8 一個未使用 volatile 關鍵字而造成錯誤的範例

C 語言

```
static int inBit=0;

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

// 中斷服務常式.
void ISR() {
    inBit=1;
}
```

組合語言

```
                LD    R1, inBit

LOOP:
                CMP   R1, R0
                JNE   LOOP

ISR:
                STI   inBit, 1
                RET
inBit           WORD  0
```

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

▶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

```
volatile static int inBit=0;

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


// 中斷服務常式.
void ISR() {
  inBit=1;
}
```

組合語言

```
                    
                    
                    
...
LOOP:
            LD    R1, inBit
            CMP   R1, R0
            JNE   LOOP
...
                    
                    
                    
ISR:
            STI   inBit, 1
            RET
inBit       WORD  0
```

範例 11.10 讀取 M0 鍵盤暫存器的組合語言程式

▶範例 11.10 讀取 M0 鍵盤暫存器的組合語言程式

組合語言版

```
    LD   R8, IoBase
    LDB  R1, [R8+1]
    LDB  R2, [R8+2]
    SHL  R2, 8
    OR   R3, R1, R2
...
IoBase WORD  0xFFFFFF00
```

C 語言 (對照版)

```
R8 = IoBase;
R1 = [R8+1];
R2= [R8+2];
R2 = R2 << 8;
R3 = R2 | R1;
...
int IoBase= 0xFFFFFF00;
```

範例 11.11 讀取 M0 鍵盤暫存器的 C 語言程式

▶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)
```

範例 11.12 檢查按鍵 5 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

```
LD R8, IoBase
LD R3, [R8+1]
LDI R7, 0x20
AND R4, R3, R7
```

C 語言 (對照版)

```
R8 = IoBase
R3 = [R8+1]
R7 = 0x20
R4 = R3 & R7
```

C 語言 (真實版)

```
UNIT16 isK5hit=KEY&0x20;
```

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

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

```
...
   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
```

C 語言 (對照版)

```
...
   R8=IoBase
   R3=[R8+1]
   R7= 0x20
   R4 = R3 & 0x20;
   if (R4 != 0)
      goto L2;
   R1 = 0x76;
   [R8+0x00] = R1;
L2:
...
int IoBase= 0xFFFFFF00;
```

C 語言 (真實版)

```
...
UNIT16 isK5hit=key&0x20;
if (isK5hit != 0)
   SEG7_REG = 0x76;
...
```

11.2 驅動程式

- 用途
  - 對使用者而言
    - 驅動程式是一個需要安裝的軟體
  - 對程式設計師而言
    - 驅動程式則是用來控制特定輸出入裝置的程式。

- 特性
  - 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。

範例 11.14 M0 電腦的驅動程式(檔頭)

▶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`	定義 BYTE 型態
`#define UINT16 unsigned short`	定義 UNIT16 型態
`#define BOOL unsigned char`	定義 BOOL 型態
`#define SEG7_REG(*(volatile BYTE*)0xFFFFFF00)`	七段顯示器的映射位址
`#define KEY_REG1 (*(volatile BYTE*) 0xFFFFFF01)`	鍵盤暫存器的映射位址
`#define KEY_REG2 (*(volatile BYTE*) 0xFFFFFF02)`	七段顯示器的映射位址
`#define KEY (KEY_REG2 << 8 | KEY_REG1)`	

範例 11.14 M0 電腦的驅動程式(資料結構)

C 語言程式檔 (driver.c)

```
#define BYTE seg7map[]={ /*0*/ 0x3F,
   /*1*/ 0x18, /*2*/ 0x6D, /*3*/ 0x67,
   /*4*/ 0x53, /*5*/ 0x76, /*6*/ 0x7E,
   /*7*/ 0x23, /*8*/ 0x7F, /*9*/ 0x77 };


#define char keymap[]={
   '1', '2', '3', '+',
   '4', '5', '6', '-',
   '7', '8', '9', '*',
   '#', '0', '?', '/'};
```

說明

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 2 進位的 00111111，也就是 0x3F。

keymap 是鍵盤的字元地圖，在 keyboard_getkey() 中可用來查出對應字元。

範例 11.14 MO 電腦的驅動程式(函數)

```
// 七段顯示器驅動程式
void seg7_show(char c) {
   SEG7_REG = map7seg[c-'0'];
}

// 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)
}
```

在七段顯示器中輸出 b 數字

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 11.15 M0 電腦的主程式 (按下數字鍵後顯示在螢幕上)

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)

```
#include <driver.h>

int main() {
   while (1) {
      while (!keyboard_ishit()) {}
      char key = keyboard_getkey();
      if (key >='0' && key <='9')
            seg7_show(key);
   }
}
```

說明

```
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
```

輪詢 v.s. 中斷

- 輸出入的程式設計模式有兩種
  - 輪詢 v.s 中斷
- 輪詢
  - 輪流詢問各個輸出入裝置 (由 CPU 主控)。
- 中斷
  - 在輸出入完成時，利用硬體機制通知系統。

11.3 輪詢機制

- 輪詢與訊息傳遞
  - 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)。

- 輪詢的實作方法
  - 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。

- 用途
  - 常用在嵌入式系統當中
  - 也曾被用在 Windows 3.1 當中實作協同式多工機制

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) { ...
   } 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() {…}
```

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 {
  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
```

11.4 中斷機制

- 何謂中斷機制？
  - 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

- 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。

CPU0 的中斷設定程式

- 原理
  - 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。

- 範例
  - 假如 CPU0 收到代碼 4 的中斷請求訊號時, 就會跳到第4個中斷的記憶體位址 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 CPU0 的中斷處理程式 (組合語言)

▶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: JMP Unexpected	未預期中斷 不處理、無窮迴圈

SwiHnd:
  PUSH {R1..R14}
  CALL CSwiHandler
  POP {R14..R1}
  RET
	軟體中斷
	保留暫存器 R1..R14
	跳到 CSwiHandler 函數
	恢復暫存器 R1..R14
	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

…	…

範例 11.19 CPU0 的中斷處理函數

▶範例 11.19 CPU0 的中斷處理函數 (C 語言

中斷處理函數 (C 語言)

```
void CIrqHandler(void){
   int id = rINT;

   if ((id>=0)&&(id<MAX_IRQ)) {
     if(irq_table[id].handler){
      irq_table[id].handler();
     } else {
      error("IRQ 函數尚未設定!");
     }
   }
}

void CSwiHandler(void) {
…
}
…
```

說明

軟體中斷的處理函數
取得中斷代號 id
　rINT 是中斷暫存器
如果是合理的中斷代號
 如果請求表中有函數
　呼叫該函數
 否則
　處理錯誤

範例 11.20 中斷函數的註冊函數

▶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)	說明

`void register_irq(unsigned int id,`
`  void (*handler)(void)) {`
`  if ((id >= 0) && (id < MAX_IRQ)) {`
`  if (!irq_table[id].handler) {`
`  irq_table[id].handler = handler;`
`  }else{`
`  error("IRQ 函數重複定義!");`
`  }`
`  }`
`}`
	中斷註冊函數, 參數為
	代號(id), 函數(handler)
	檢查中斷代號是否合理
	如果該中斷尚未定義
	設定該中斷為 handler
	否則
	處理錯誤

`void unregister_irq(unsigned int id) {`
`  irq_table[id].handler=NULL;`
`}`
	取消中斷註冊
	清除該中斷

範例 11.21 利用時間中斷讓 M0 電腦反覆從 0 數到 9

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

11.5 啟動程式

- 功能
  - 建立電腦的程式執行環境, 讓其他程式得以順利執行
- 測試方式
  - 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。

啟動程式的任務

- 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。

啟動位址

- 功能
  - 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啟動位址中, 才能正確的啟動。
- 設計方式
  - 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。

嵌入式系統的啟動程式

- 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。
- 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。
- 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。

啟動程式的任務

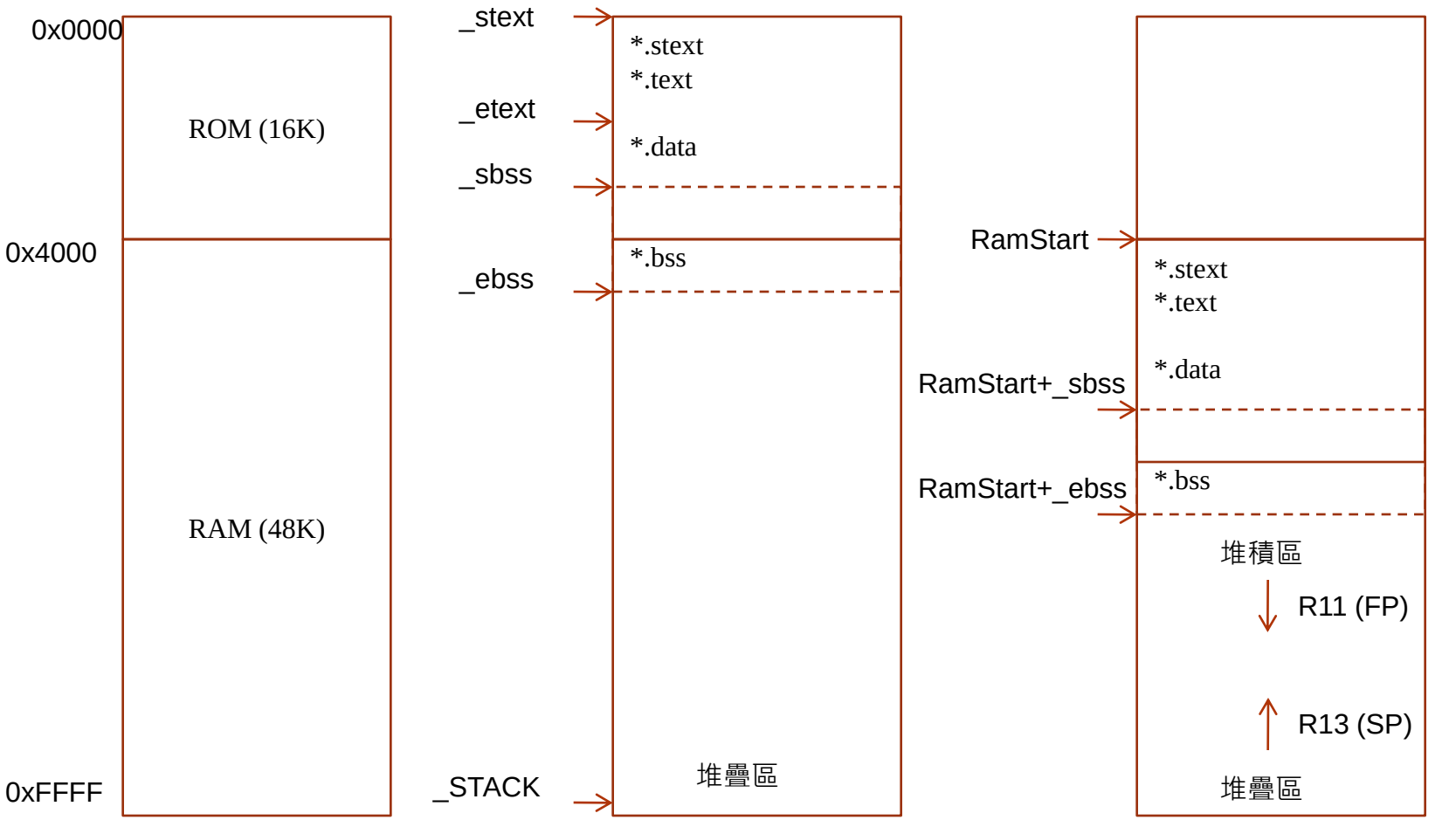
- 1. 設定中斷向量, 啟動中斷機制。
- 2. 設定 CPU 與主機板的各項參數, 讓 CPU 與主機板得以進入正確的狀態。
- 3. 將存放在永久儲存體中的程式與資料搬到記憶體中。
- 4. 設定高階語言的執行環境, 包含設定堆疊 (Stack)、堆積 (Heap) 等區域。

M0 電腦的啟動過程

- M0 電腦擁有
  - 16K 的 ROM：0x0000~0x3FFF
  - 48K 的 RAM：0x4000~0xFFFF

- 啟動過程
  - 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。

圖 11.3 M0電腦的記憶體配置圖



(a) M0 電腦的記憶體　(b) 初始環境：啟動初期　(c) 執行環境：啟動程式執行完畢

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

M0 電腦的啟動程式

- 1. 設定中斷向量, 啟動中斷機制。
- 2. 將 ROM 中的程式與資料搬到 RAM 中。
- 3. 設定高階語言的執行環境, 包含設定堆疊 (Stack)、堆積 (Heap) 等區域。

範例11.23 M0 電腦的啟動程式（中斷向量設定）

行號	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		

範例11.23 MO 電腦的啟動程式（將程式與資料搬移到RAM中，並進行起始化布局）

```
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:
36              LD      R3, [R0+R2]
37              ST      R3, [R1+R2]
38              ADD     R2, R2, 1
39              CMP     R2, R1
40              JNE     LOOP1
41          JumpToRam:                      跳到 RAM 版本的 Cinit 標記中。
42              LD      R15, RamCinit
43
44          Cinit:                          C 語言環境設定
45              LD      R1,  RamSbss        首先清除 BSS 段
46              LD      R2,  RamEbss        設定 RAM 中 BSS 段的內容為 0
47          LOOP2:
48              CMP     R1, R2
49              JNE     InitStacks
50              ST      R0, [R1]
51              JMP     LOOP2
52
53          InitStacks:                     堆疊初始化
54              LD      SP, StackBase
55
56          MainLoop:                       無窮迴圈
57              CALL    main                進入 C 語言的主程式。
58              JMP     MainLoop
59
60          RamSbss WORD RamStart+_sbss
61          RamEbss WORD RamStart+_ebss
62          RamCinit WORD RamStart+Cinit
63          StackBase WORD _STACK
```

中斷的禁止與恢復

- 禁止中斷：
  - LDI R12, 0xD0
- 允許中斷：
  - LDI R12, 0x00

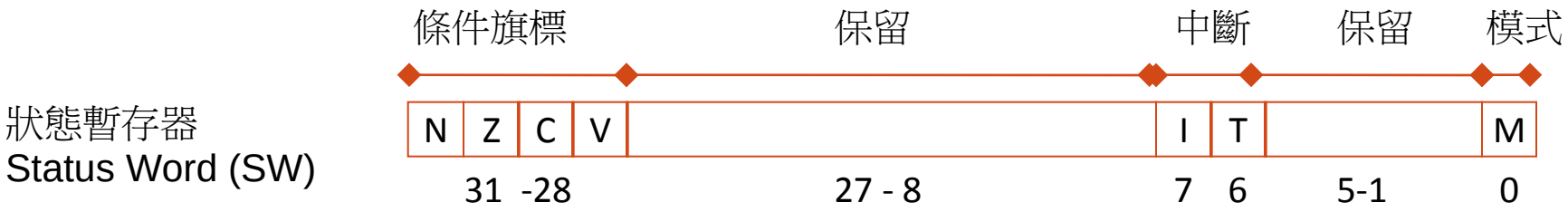


圖 11.4 CPU0 的狀態暫存器 R12 之位元圖

11.6 系統整合

- 專案建置工具
  - 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

- GNU 的 Make 工具
  - 是專案建置實相當常見的工具, 可以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。

- 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 2 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`	

11.7 實務案例：新華 Creator S3C2410 實驗板

- 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。
- 軟體環境
  - 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。

新華 Creator S3C2410實驗板範例 LCD 的建置過程

▶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 …略…	重建專案
/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	製作 demo_ram.axf 目的檔
arm-elf-objcopy -O binary -S demo_ram.axf demo_ram.bin …略…	將該目的檔轉為 2 進位的 demo_ram.bin

圖 11.5透過超級終端機，顯示該實驗板的起始功能表

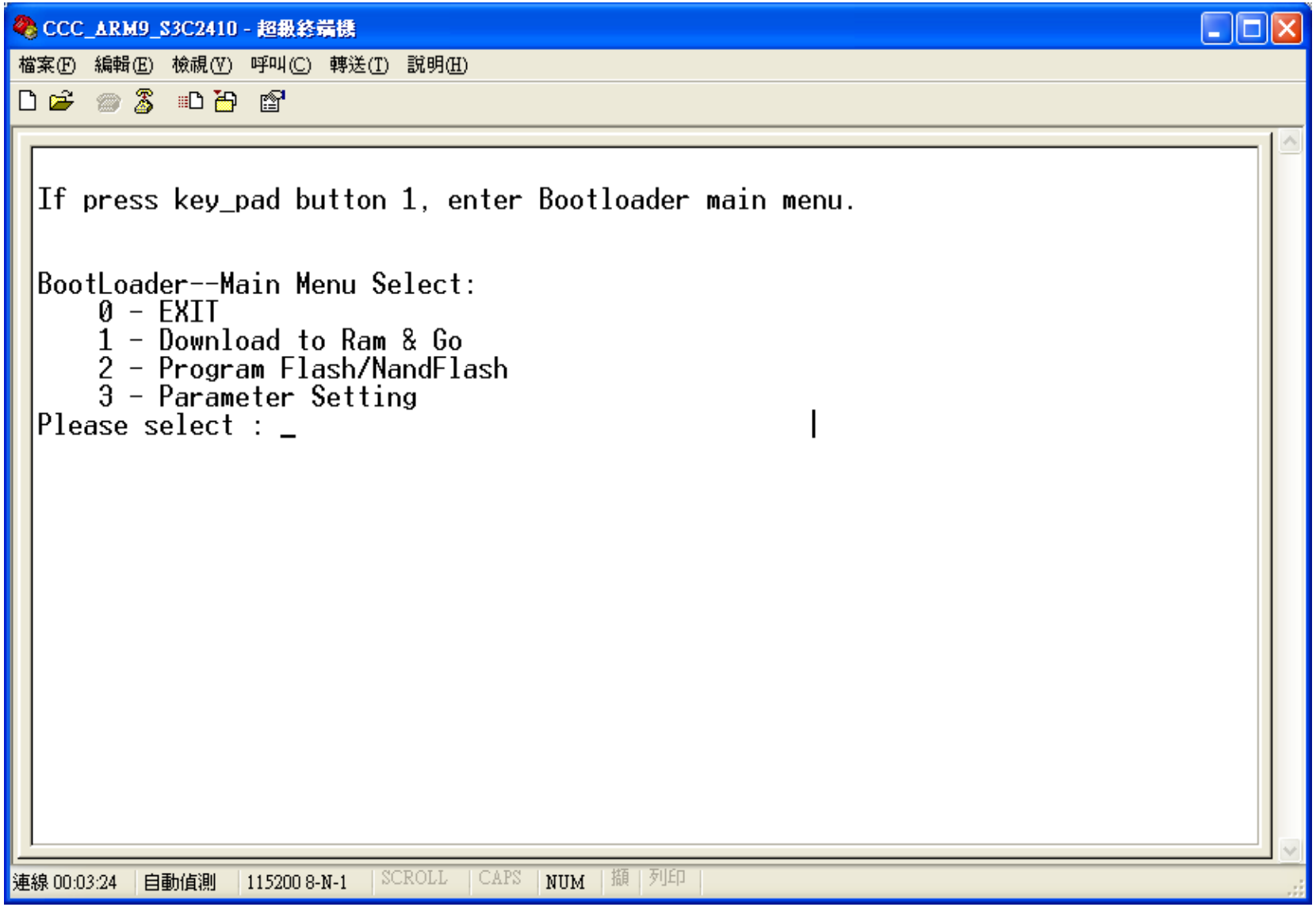


圖 11.6按下傳送功能，選取欲上傳的檔案 demo_ram.bin

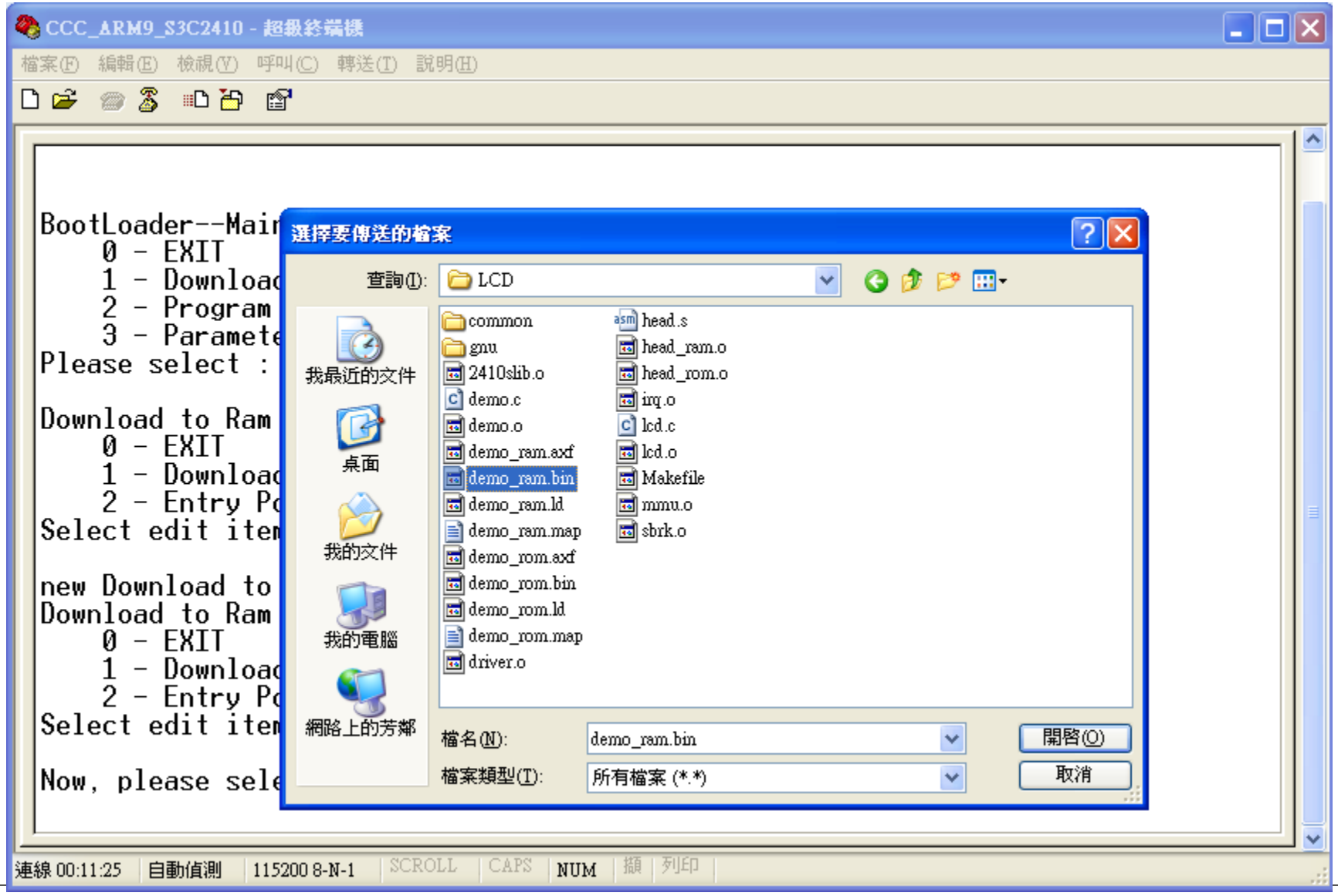
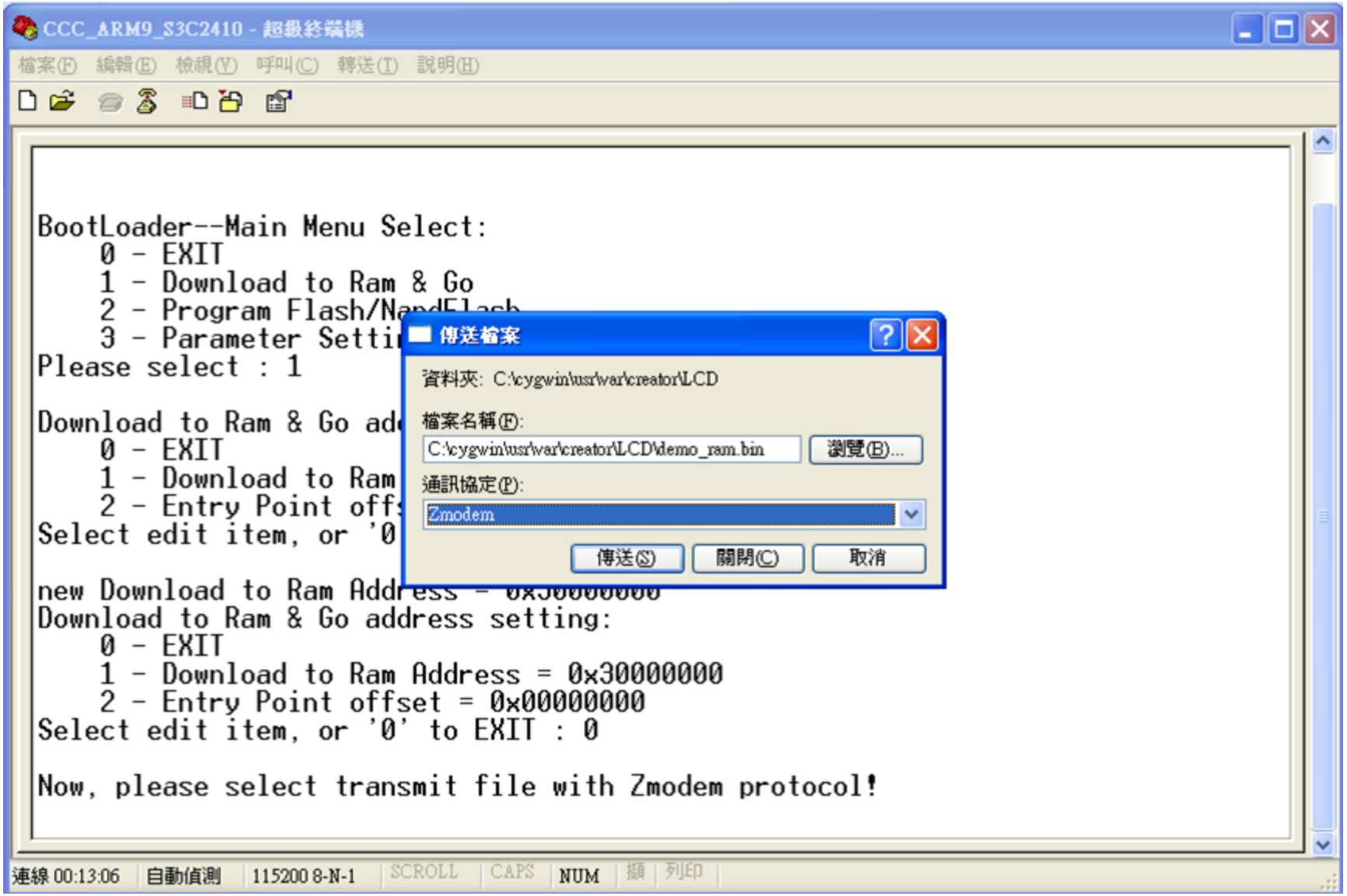


圖 11.7選擇通訊協定為 Zmodem，按下傳送鈕，開始上傳



開始執行 LCD 程式範例

- 執行過程
  - 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

採用實驗板的注意事項

- 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。
- 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。

結語 (1)

- 輸出入
  - 專用的輸出入指令 v.s. 記憶體映射輸出入
  - 輪詢 v.s. 中斷
- 驅動程式
  - 將輸出入功能包裝成函數，以方便程式呼叫。
  - 通常採「註冊-反向呼叫機制」(callback)
- 輪詢機制
  - 輪流詢問，訊息傳遞架構。
- 中斷機制
  - 搭配硬體，在輸出入完成時主動中斷 CPU，執行中斷函數。

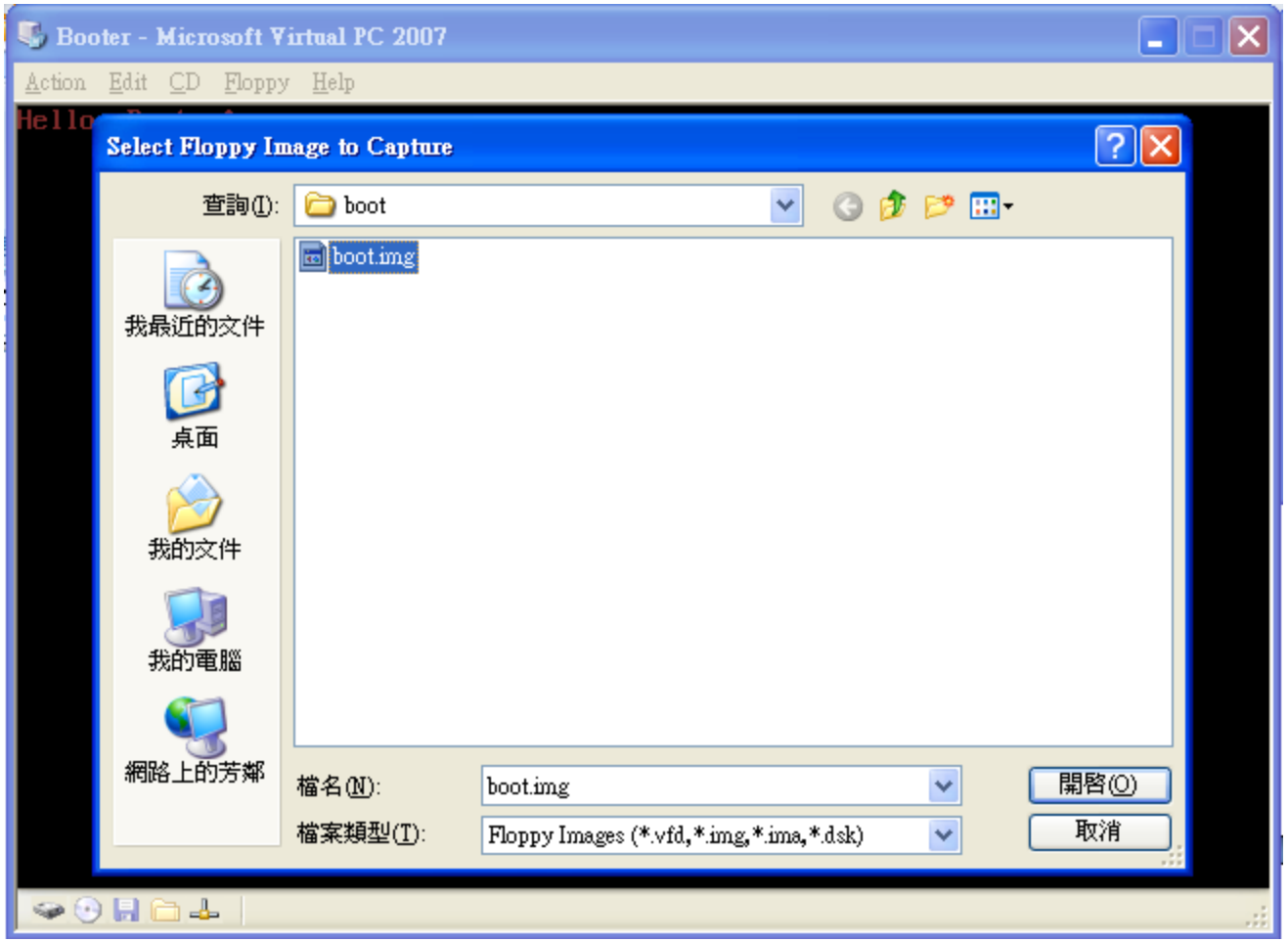
結語 (2)

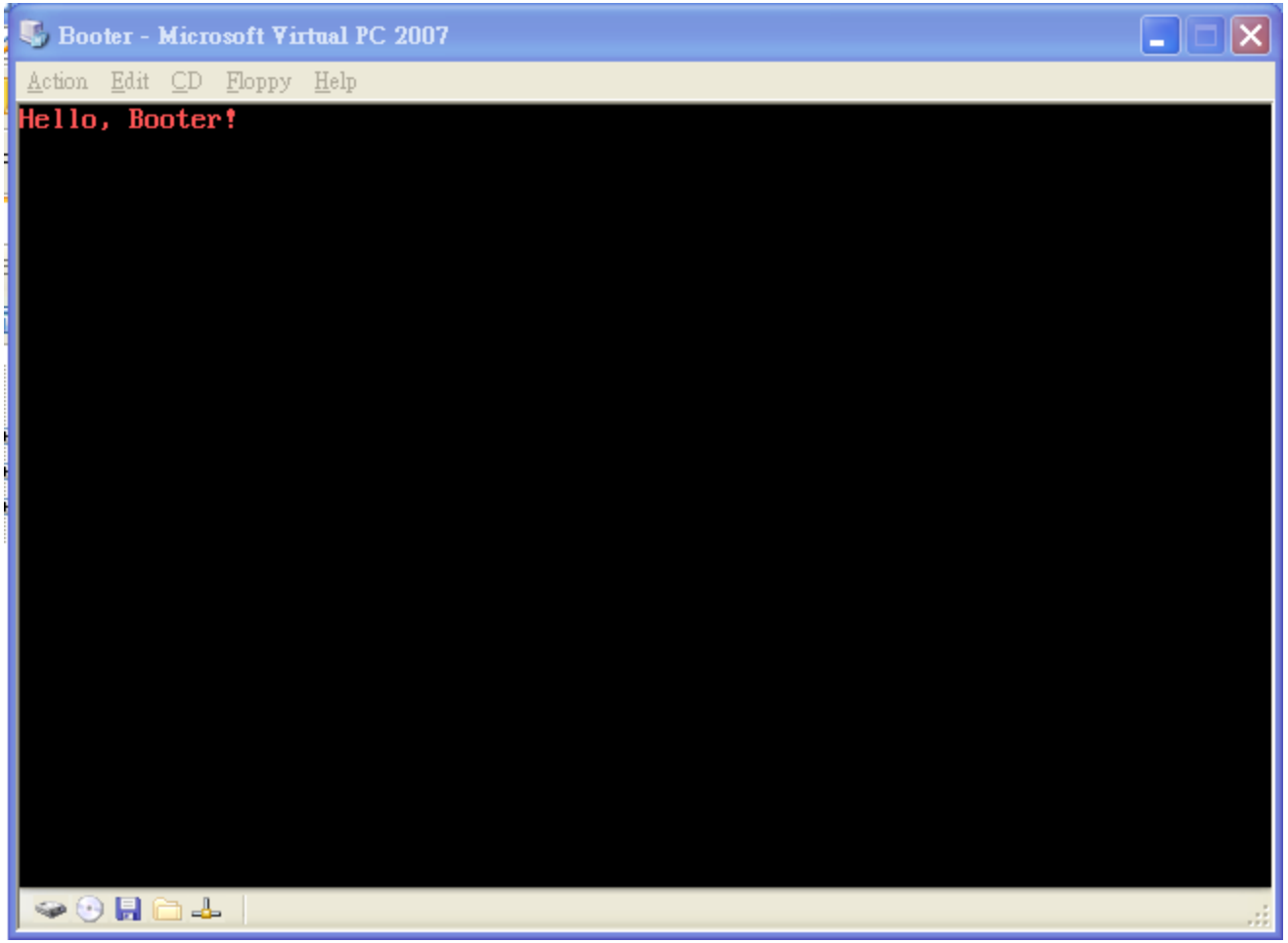
- 啟動程式
  - 1. 設定中斷向量與函數。
  - 2. 設定 CPU 與主機板的各項參數。
  - 3. 將存放在永久儲存體中的程式與資料搬到記憶體中。
  - 4. 設定高階語言的執行環境, 包含堆疊與堆積。
- 系統整合
  - 使用專案建置工具
  - GNU Make 的使用
  - 連結檔的用途 (例如 M0.ld)
- 實務案例
  - 新華 Creator S3C2410 實驗板

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。

未收錄於書中的圖


Booter - Microsoft Virtual PC 2007
Action Edit CD Floppy Help
Select Floppy Image to Capture
查詢(I):
boot
boot.img
我最近的文件
桌面
我的文件
我的電腦
網路上的芳鄰
檔名(N):
boot.img
開啓(O)
檔案類型(T):
Floppy Images (*.vfd,*.img,*.ima,*.dsk)
取消


Booter - Microsoft Virtual PC 2007
Action Edit CD Floppy Help
Hello, Booter!