Real Time OS 下的多工系統設計

作者：廖勝泰

1 前言

對於在微控制器下從事產品開發的系統工程師而言，如果所開發的產品只是作簡單的 I/O 控制，往往在硬體方面是選用簡單的 8 位元微控制器，再搭配以組合語言撰寫的軟體程式來完成系統整合。然而隨著時代的進步，嵌入式系統本身已不再是簡單的作些數位 I/O 控制或著是控制 LCD 顯示面板這樣簡單，一個功能複雜的系統往往在硬體方面採用的是高速的 DSP 控制器或者是耗電量低的 ARM CPU；而在軟體撰寫上也都改用 C 語言來完成。由於嵌入式系統的特性使然，在軟體的架構上和一般 PC 上跑的應用程式有很大的差異。在這篇文章中，我所要探討的課題就是著重於在嵌入式系統下如何藉由 Real Time OS（即時作業系統）來幫助我們設計多工程式。

2 Real Time OS 簡介

Real Time OS 的主要功用在於可支援多工要求，對於外界的輸入可以即時作出反應。舉例而言：假定目前在執行的系統程式常常要作一段很非常耗時的複雜計算，而我們又要求這個系統在進行這個計算的過程中，如果外界有某個數位訊號進來時，必須能夠立刻跳出去處理這個外界信號。這樣的要求在單工作業系統如 MS-DOS 下，除非透過中斷否則是無法達到即時處理的要求；然而在 Real Time OS 中，系統工程師就可以將這個應用程式切割成兩部份，由作業系統去針對它們分別加以排程執行。在這樣的情況下，對於使用者的感覺而言，就會覺得 CPU 本身在作運算的同時，也像是隨時在監控外界的輸入訊號。而事實上，在只使用一個 CPU 的情況下，系統是沒有辦法同時處理兩件事情。我們只是藉由 RTOS 巧妙的排程處理，讓這兩件事情像是「同時」被處理。

談完了 Real Time OS 所能帶給我們的好處後，接下來我們就會想到要如何選擇一個適合的 Real Time OS 來使用呢？基本上可以從產品規格和成本上來作選擇。在產品規格上，廠商所強調的在於 task 間切換所需的時間以及優先權的層級數目。而在成本方面，有些商業化的 RTOS 產品除了在購買開發系統時要付費之外，每個使用此作業系統的產品在販售時還要收取所謂的 run-time license，也就是每賣出一個產品就會被抽取權利金（最有名的例子就是微軟的 embedded NT

及 Windows CE）；相反地，有些產品則是以一次買斷的方式來授權使用。

3 系統韌體架構

對於熟悉 C 語言的讀者而言，必然對於 C 語言中的指標印象非常深刻。指標最常用來作爲動態配置記憶體時使用，因此在一般的使用情況下指標都是指向程式中的資料段。然而當我們將指標用來指向自定函式時，在某些場合下是非常有用的。尤其是在 super-loop 的程式架構中，如果搭配 Finite-State Machine 時，我們往往會迷失在龐大複雜的 switch-case 中，不但程式碼難以閱讀，而且也很容易因此產生很多 bug。這個時候如果能夠善用函式指標陣列，就可以將這個原本複雜的 switch-case 簡化爲一兩行的函式呼叫，而原本混在一起的程式碼就可以打散爲各自獨立的使用者自定函式。更重要的是在實際的 Finite-State Machine 控制程式中，往往在每個不同的 state 中有其獨特使用的區域變數要處理，如果單純只採用 super-loop 的程式架構的話，每個 state 中個別使用到的區域變數宣告會混在一起，在沒有個別 namespace 的保護下會造成程式碼的嚴重難以維護（光爲一堆變數取不同的名字就傷透腦筋）。而這個問題在使用函式指標陣列後，可以完全被妥善地解決，因爲個別 state 所使用到的區域變數只要放在各自的 state 處理函式中宣告即可，不再有原本雜亂無章的情況出現。

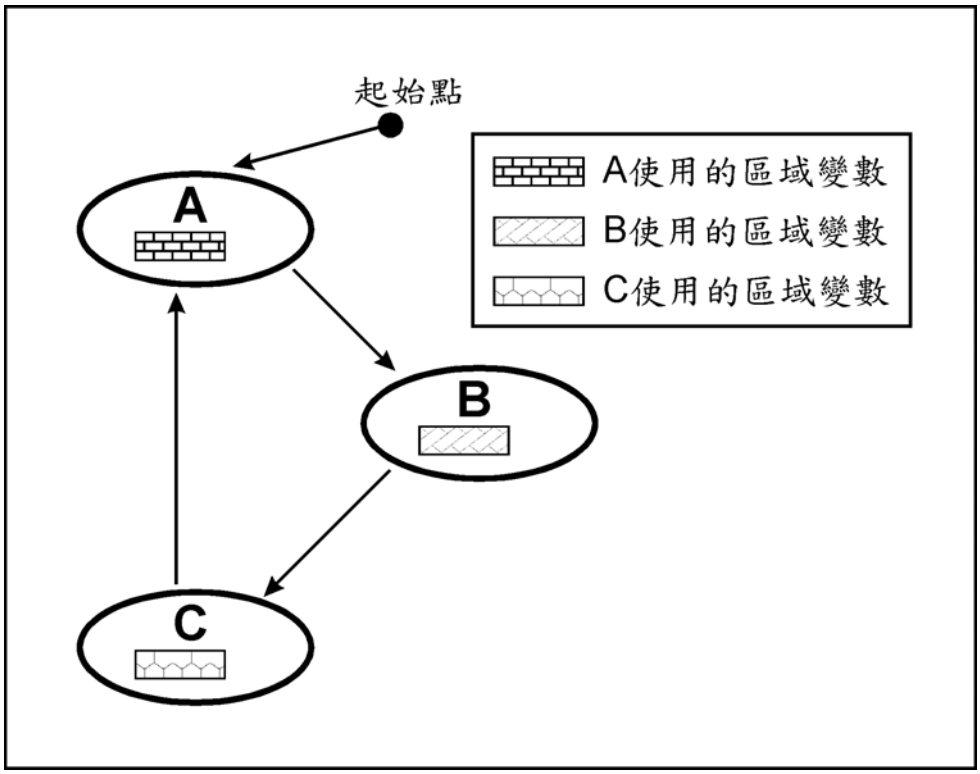

Fig.1 一個具備 3 個不同狀態的系統示意圖，其中每個不同的狀態都使用到一些獨特的區域變數。

在 Fig.1 和 Fig.2 中，我們以實際的示意圖來說明以上的概念。其中在 Fig.1 所顯示的是一個具備 3 個子狀態的系統概圖，而在 Fig.2 中則清楚地比較了這兩種設計方式的優缺點。可以很明顯地看出使用函式指標陣列可以得到一個清楚簡潔的系統，可惜的是在一般的教科書中較少去強調這種設計方法。

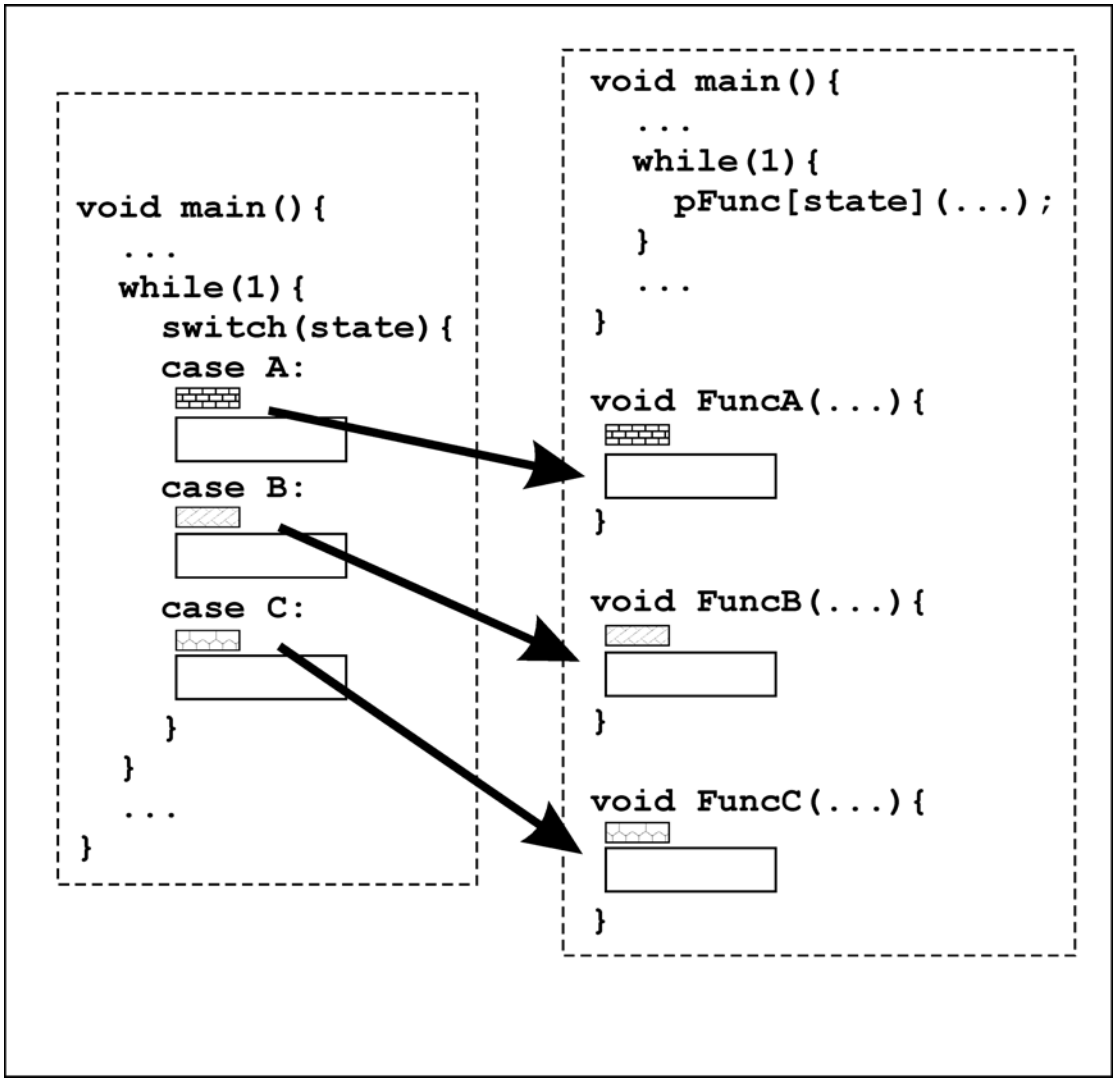

Fig.2 左邊是以 super-loop 的方式所建構的系統概圖，而右邊則是以函式指標陣列的方式所建構的系統概圖。可以很明顯地看出原本繁雜的 switch-case 變得很乾淨，而各個 state 所使用到的區域變數也分開在各自的處理函式內作宣告。

在一些不考慮任何時序問題的系統中，使用以上這兩種程式架構就足以應付。然而，在一個實際的應用系統中，時序問題是非常重要的一件事。系統工程師總是必須去面對一些和時間有關的要求：諸如「每隔 5 分鐘，將溫度昇高至...每隔 2 分鐘，將機器手臂從 A 點移到 B 點...」...。除了這些各式各樣與時間相關的要求之外，別忘了系統本身還得具備一個對於使用者輸入有著「即時反應」能力的使

用介面。考慮到以上的這些時序問題之後，即使是一個簡單的 Finite-State Machine，也需要一個非常有經驗的系統工程師才能妥善處理。然而一旦系統後續有任何需要改變或新增功能的要求，可不是程式碼加上去就好了，而是要再一次妥善分析彼此的時序關係後，才能夠找到一個適當的「點」把程式碼放進去。RTOS 的出現，最重要的目的就是要解決系統工程師所面對的複雜時序問題，而且降低系統程式碼的複雜度和後續維護成本。

4 基本觀念與術語

在所有的商業化 RTOS 產品規格中，我們最常看到的一個字眼就是所謂的「Preemptive」多工排程能力。這個術語所代表的意義及多工系統中的優先權概念就是我們在這一節中所要說明的主題。

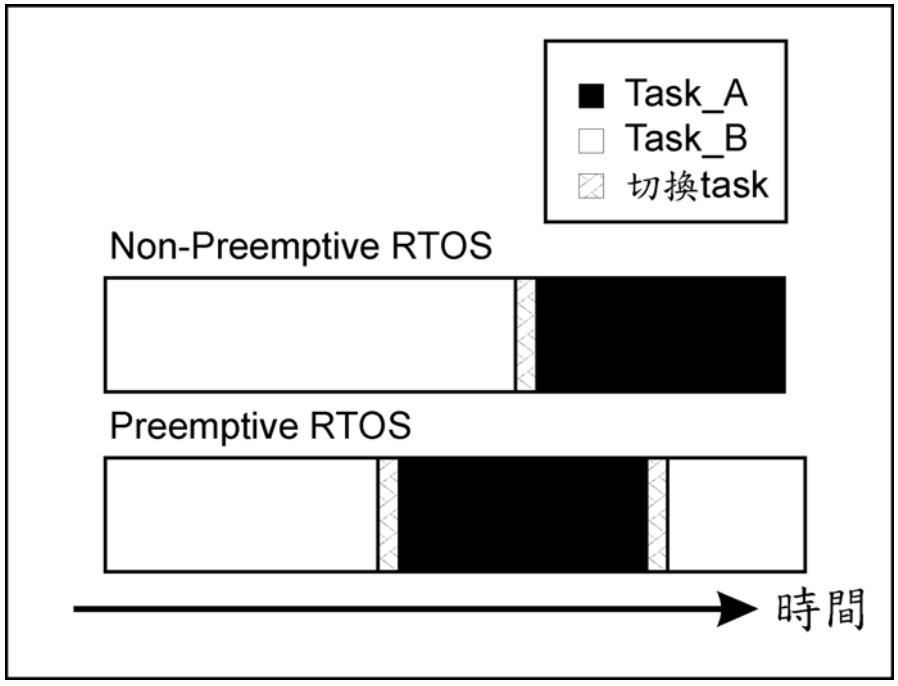

Fig.3 比較在兩種不同的排程架構下，RTOS 如何切換 CPU 去執行不同的 task。在 Preemptive RTOS 中，RTOS 會視情況而將目前執行中但優先權較低的 task（Task_B）予以中斷，而先去執行優先權較高的 task（Task_A）。然而在 Non-Preemptive RTOS 中，除予目前執行中的 task 主動放棄 CPU 的佔有權，否則是沒有辦法在它執行的過程中加以中斷的。

一個 RTOS 的核心工作就是安排系統中所有的 task 分時分段交由 CPU 加以執行，也就是所謂的排程管理。根據各個 task 所實際負責的工作性質，我們可以賦予它們不同的執行優先順序。這個優先順序稱爲優先權，通常是用一個數字來

代表，數字越小的代表其執行優先順序越高。所謂的「Preemptive」和「Non-Preemptive」的分別，就在於 RTOS 是否會「主動打斷目前執行中的 task」，而跳去執行另一個等待執行且具備更高優先權的 task。舉例而言：假定在我們的系統中總共有 3 個 task，分別是 Task_A、Task_B 和 Task_C，且其優先權的順序爲 Task_A 高於 Task_B、Task_B 又高於 Task_C。CPU 在執行 Task_B 的途中，Task_A 和 Task_C 分別被外界的訊號所喚醒而進入等待被執行的狀態，這個時候在「Preemptive」的 RTOS 中，Task_B 會被先行擱置而跳去執行優先權較其來得高的 Task_A；然而，如果在「Non-Preemptive」的 RTOS 中，Task_A 就必須一直等到 Task_B 放棄 CPU 的控制權之後，RTOS 才會去執行它。另外要注意的地方是雖然 Task_A 和 Task_C 都進入等待被執行的狀態，然而由於 Task_A 的優先權高於 Task_C，因此 Task_A 會先被執行。我們用 Fig.3 來說明以上的觀念，讀者可以很明顯地看出，Preemptive RTOS 的反應能力會比 Non-Preemptive 的 RTOS 來得好，因爲它總是會優先跳去處理優先權最高的 task。以微軟的 Windows 作業系統而言，它們在多工排程上，Windows 3.1 就是屬於 Non-Preemptive，而 Windows 95/98 則是屬於 Preemptive。

5 RTOS 設計原理

以本文中所使用的μc/OS [1]而言，它在 DOS 環境下運作的原理非常簡單：整個系統與 PC 的第 8 號中斷串接起來 [2]，而第 8 號中斷是每秒鐘會被觸發 18.2 次，也就是每隔系統的 1 個 tick 之後就會被觸發一次。只要把排程器（scheduler）和這個中斷串接起來之後，scheduler 就可以每隔一段固定的時間去對所有的 task 作排程。原理雖然簡單，但是讀者可以想見中間有多少複雜的 stack 處理工作在進行，而且爲了講求排程的速度，這一部份的碼在實用上都必須用組合語言來撰寫，除錯的難度非常高。

由於不論在任何的硬體平台上，CPU 都有提供週期性的硬體中斷，因此只要透過此週期性的硬體中斷，就可以以一個固定的 time slice 去作時間分割、分別執行不同的 task。而另外一個 RTOS 設計上的重點就是必須妥善規畫一個記錄 task 狀態（目前執行狀態、優先權…）的資料結構，使用者每 create 一個新的 task，就將使用者對這個 task 的設定值記錄到這個資料結構中，所有 task 的對應資料結構最後以串列的型式存在於記憶體內，RTOS 在起動後，就會每隔一個固定的週期被喚醒一次，而在被喚醒之後要作的重要工作就是走遍這個串列，找尋下一個要被執行的 task 爲何，接著就是複雜的 stack 處理與暫存器切換，將 CPU 下一個要執行的指令位址設定到此 task 中。

6 應用實例

我將透過一個在 MS-DOS 下執行的範例程式來說明如何使用一個 RTOS 來幫助我們達成多工的要求。爲何選用 MS-DOS 呢？因爲我的目的只是要實際示範 RTOS 的 Programming Model，讓讀者們能夠比較有感覺。MS-DOS 本身不具備多工能力，適合展示 RTOS 的功能，而且 PC 下的執行測試環境是最方便的，我們不用去採買任何特別的電子零件或 Evaluation Board。

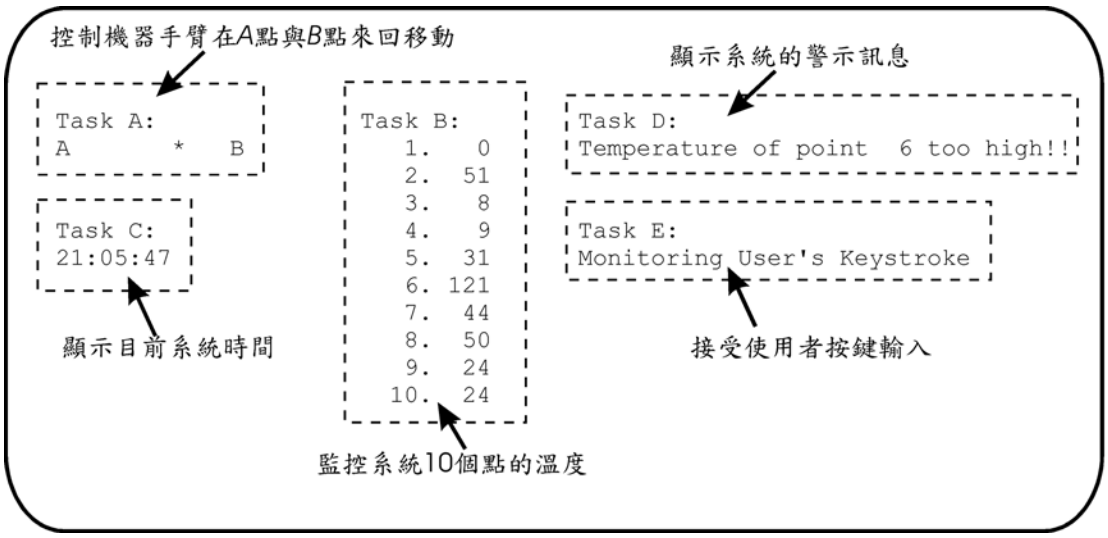

Fig.4 這是整個範例程式執行時的螢幕顯示。整個系統由 5 個 Task 組成，分別負責處理不同的工作。在不同的 Task 間也可以互相交換資料，如本系統中的 Task B 與 Task D。

整個系統由 5 個 task 組成，分別負責不同的工作，如 Fig.4 所示。在我們的這個假想系統中，總共要處理 5 件完全不相干的事情。如果不使用 RTOS 來幫我們作 task 間的自動切換的話，使用傳統的 super-loop 架構會使得整個系統變得很複雜而且很難維護。更重要的一點是，如果系統後續要增加新的功能時，維護人員必須很小心地避免新增加的程式碼破壞了原先的時序設計。相反地，在 RTOS 的協助下，我們可以很輕鬆地將 5 件 task 分別交給 5 個不同的函式（Task_A~Task_E）來負責處理：

- Task_A：控制機器手臂在兩點之間來回移動。
- Task_B：監控系統 10 個點的溫度。
- Task_C：顯示目前系統時間。
- Task_D：顯示系統的警示訊息。
- Task_E：接受使用者按鍵輸入。

撰寫好這 5 個處理程序之後，系統一啓動時就呼叫 OSCreateTask 分別建立 5 個不同的 task 來分別執行它們即可。完整的範例程式碼列在 List.1 中，讀者可自行參考之。

```
#include   "includes.h"

// 設定 Task 所使用的堆疊空間大小
#define TASK_STK_SIZE 768

UBYTE Task_A_Stk[TASK_STK_SIZE];
UBYTE Task_B_Stk[TASK_STK_SIZE];
UBYTE Task_C_Stk[TASK_STK_SIZE];
UBYTE Task_D_Stk[TASK_STK_SIZE];
UBYTE Task_E_Stk[TASK_STK_SIZE];
UWORD OldSS;
UWORD OldSP;
UWORD OldBP;

OS_EVENT *pMbox;

void interrupt (*OldTickISR)(void);

// 5 個 Task 的函式原型宣告。
void far Task_A(void *data); // Task A
void far Task_B(void *data); // Task B
void far Task_C(void *data); // Task C
void far Task_D(void *data); // Task D
void far Task_E(void *data); // Task E

void DispChar(UBYTE x, UBYTE y, char c);
void DispStr(UBYTE x, UBYTE y, char *s);

void main(void)
{
  // 清除螢幕。
  clrscr();
  // 儲存系統設定。
  OldBP     = _BP;
```

```
  OldSS     = _SS;
  OldSP     = _SP;
  OldTickISR = getvect(0x08);

  setvect(0x81, OldTickISR);
  setvect(uCOS, (void interrupt (*)(void))OSCtxSw);
  OSInit();
  pMbox = OSMboxCreate(NULL);
  OSTaskCreate(Task_E, NULL, (void *)&Task_E_Stk[TASK_STK_SIZE], 4);
  OSStart();
}

// Task A: 控制機器手臂在 A 點與 B 點之間來回移動
void far Task_A(void *data)
{
  UBYTE x=10;
  UBYTE y=10;
  int iDirection=0;

  DispStr(9, 9, "Task A:");
  DispChar(9, 10, 'A');
  DispChar(21, 10, 'B');
  DispChar(x, y, '*');
  while (1) {
    OSTimeDly(5);
    if (x>=20)
      iDirection=1;
    if (x<=10)
      iDirection=0;
    DispChar(x, y, ' ');
    if (iDirection==0)
      x++;
    else
      x--;
    DispChar(x, y, '*');
  }
}
```

```
// Tack B: 每 2 秒鐘掃描一次系統 10 個點的溫度，超過 100 度時發出警報。
void far Task_B(void *data)
{
  int iTemperature[10], i;
  char cBuffer[20];
  char *pAlarmText;

  DispStr(30, 9, "Task B:");
  while (1) {
    for (i=0; i<10; i++) {
      // 以亂數來模擬溫度變化。
      iTemperature[i]=random(125);
      DispStr(32, 10+i, "      ");
      sprintf(cBuffer, "%2d. %3d", i+1, iTemperature[i]);
      DispStr(32,10+i, cBuffer);
      if (iTemperature[i]>100) {
     pAlarmText=(char*)malloc(50);
     sprintf(pAlarmText, "Temperature of point %2d too high!!", i+1);
     OSMboxPost(pMbox, pAlarmText);
      }
    }
    // 休息 2 秒。
    OSTimeDly(36);
  }
}

// Tack C: 顯示系統時間。
void far Task_C(void *data)
{
  time_t timer;
  struct tm *tblock;
  char cBuffer[20];

  DispStr(9, 13, "Task C:");
  while (1) {
    timer = time(NULL);
    tblock = localtime(&timer);
    sprintf(cBuffer, "%02d:%02d:%02d", tblock->tm_hour,
```

```
     tblock->tm_min, tblock->tm_sec);
    DispStr(9, 14, cBuffer);
    OSTimeDly(18);
  }
}

// Task D: 負責記錄顯示系統的各種異常狀況。
void far Task_D(void *data)
{
  char *pAlarmText;
  unsigned char err;

  DispStr(45, 9, "Task D:");
  while (1) {
    pAlarmText=(char*)OSMboxPend(pMbox, 0, &err);
    DispStr(45, 10, pAlarmText);
    free(pAlarmText);
  }
}

// Task E: 監視使用者的按鍵輸入
void far Task_E(void *data)
{
  // 安裝 Tick ISR 以達到 Preemptive 多工的目的。
  OS_ENTER_CRITICAL();
  setvect(0x08, (void interrupt (*)(void))OSTickISR);
  OS_EXIT_CRITICAL();

  // 啓動其餘的 4 個 Task。
  OSTaskCreate(Task_A, NULL, (void *)&Task_A_Stk[TASK_STK_SIZE], 0);
  OSTaskCreate(Task_B, NULL, (void *)&Task_B_Stk[TASK_STK_SIZE], 1);
  OSTaskCreate(Task_C, NULL, (void *)&Task_C_Stk[TASK_STK_SIZE], 2);
  OSTaskCreate(Task_D, NULL, (void *)&Task_D_Stk[TASK_STK_SIZE], 3);

  DispStr(45, 13, "Task E:");
  DispStr(45, 14, "Monitoring User's Keystroke");

  // 監視使用者按鍵輸入。
```

```
  while (!kbhit()) {
    OSTimeDly(18);
  }
  OS_ENTER_CRITICAL();

  // 在結束程式前，恢復成原先的系統設定。
  setvect(0x08, OldTickISR);
  OS_EXIT_CRITICAL();
  _BP = OldBP;
  _SS = OldSS;
  _SP = OldSP;
  exit(0);
}

// uCOS 所附的範例程式中所提供的顯示字元函式。
void DispChar(UBYTE x, UBYTE y, char c)
{
  UBYTE far *pscr;
  UWORD     offset;

  // Calculate position of character on the screen
  offset  = (UWORD)y * 160 + (UWORD)x * 2;
  pscr    = MK_FP(0xB800, offset);
  OS_ENTER_CRITICAL();
  // Put character in video RAM
  *pscr++ = c;
  // Put video attribute in video RAM
  *pscr   = 0x07;
  OS_EXIT_CRITICAL();
}

void DispStr(UBYTE x, UBYTE y, char *s)
{
  UBYTE far *pscr;
  UWORD     offset;

  // Calculate position of character on the screen
  offset  = (UWORD)y * 160 + (UWORD)x * 2;
  pscr    = MK_FP(0xB800, offset);
  while (*s) {
```

```
    OS_ENTER_CRITICAL();
    // Put character in video RAM
    *pscr++ = *s++;
    // Put video attribute in video RAM
    *pscr++ = 0x07;
    OS_EXIT_CRITICAL();
  }
}
```

List.1 完整的範例程式碼。

7 結語

在本篇文章中，我們了解透過即時作業系統RTOS的幫助，在設計多工系統時可以讓系統工程師透過一個比較直覺的觀點來將系統切割爲數個不同的task，由RTOS依據我們給定的優先權來排程執行。相較於以往一個系統的程式碼全部都放在一個super-loop中由程式設計師以人工的方式去微調整個系統的執行順序，我想使用RTOS對於較爲複雜的多工系統而言，它的好處是可以有效減輕系統工程師的負擔，而且比較容易交付給其它人員作後續的維護。至於在系統設計之前的多工系統分析工作要如何進行、系統的功能要如何有效地切割爲數個不同的task再搭配RTOS來執行…這些課題在我個人時間允許的情況下，在未來也會以專文來討論。如果讀者對本篇文章有任何指教之處，可以透過我的電子信箱：stliaw@ms7.url.com.tw 和我連絡，另外我個人的網址是：http://home.pchome.com.tw/computer/stliaw。

8 參考資料

[1] http://www.ucos-ii.com
[2] IBM 8086 組合語言實務/旗標出版社/施威銘著