國 立 高 雄 師 範 大 學
高雄師大學報 47, 2019. 12, 頁 53-72

即時作業系統 tinyOS1：以 ARM Cortex-M0 微控制器實測

蔡長達*

摘 要

在作業系統領域，很少發現有國人可以提供完整的原始程式來作為授課教材。本文參考 uC/OS-II 的理論、ARMv6-M 指令集與 context switch 的前人研究，成功開發出一個可運作施行的即時作業系統命名為 tinyOS1。實驗是以 ARM Cortex-M0 微控制器來做為執行平台，展示了 31 個任務之間的順序同步之功能。

關鍵詞：即時作業系統、微控制器

投稿日期：2019/05/19；接受日期：2019/09/24

* 東南科技大學資訊科技系助理教授

54 Kaohsiung Normal University Journal
No.47, Dec. 2019, pp.53-72

The TinyOS1 Real-Time Operating System Demonstrated by ARM Cortex-M0 Processor Based Microcontroller

Chang-da Tsai*

Abstract

In the field of operating system, it is exceedingly rare to find out a complete teaching material including source code proposed by Taiwanese. After investigating the uC/OS-II theory, ARMv6-M instruction set, and previous research in context switch, a real-time operating system naming tinyOS1 was developed and successfully implemented on ARM Cortex-M0 processor based microcontroller to demonstrate the functionality of sequential synchronization with 31 tasks.

Keywords: real-time operating system, microcontroller

Submitted: 2019/05/19；Accepted: 2019/09/24

* Assistant Professor, Department of Information Technology, Tungnan University

壹、前言

以 ARM Cortex-M 為 CPU 核心的微控制器一直廣泛應用於產業界，而在教學的應用上，NXP 半導體公司有提供一款 Cortex-M0 等級的 DIP 封裝微控制器 LPC1114FN28/102(4KB RAM)，非常適合在麵包板上施行軟體應用程式與硬體電路設計的基礎教學，但對於作業系統在 4KB RAM 以下之 Cortex-M0 平台的教學與實作，全世界基本上是毫無原始程式碼的教材可供施行，故作者(tinyOS 作者即本文作者)自行開發一套可在 Cortex-M0 平台運作的即時作業系統來作為授課教材，以利學生順利接軌產業界的需求，本文所發表的版本是 tinyOS1。

tinyOS1 是 tinyOS 三個版本中記憶體需求最少的版本，而代價就是最多只能夠執行 31 個任務(task)，教材設計最主要的理念是要讓作業系統可以在 Cortex-M0 低記憶體容量等級(4KB RAM 以下)之微控制器正常運作施行，在參考流行的 uC/OS-II 即時作業系統的理論基礎[1]，結合 Cortex-M0 的 ARMv6-M 指令集[2]以及 Adam Heinrich 對於內文交換(context switch)的研究[3]之後，作者成功開發出針對 Cortex-M0 這種資源等級可以順利運作的即時作業系統，命名為 tinyOS。

tinyOS1 是設計在極低 RAM 容量的平台之應用而開發，故不支援執行時期動態記憶體配置的功能，所有的應用程式在編譯時期就必須確定記憶體需求。

貳、版本

依據微控制器的硬體規格以及軟體功能目的之不同需求，tinyOS 共有四種不同應用版本，所有版本皆不收費，廠商使用 tinyOS 開發的產品也不必公開程式碼。

一、tinyOS1

最精簡版的 tinyOS，可編譯出最小記憶體需求的程式，1KB RAM 之微控制器即可順暢執行一般應用程式，非常適合物聯網作業系統的應用。tinyOS1 只提供信號量惟一一個事件功能，該事件功能可實現任務之間的順序同步或是共享資源的管理等應用，但 tinyOS1 最多只允許 31 個使用者開發的任務來運行，系統核心不提供任務之間的訊息傳遞，若是任務之間需要傳遞訊息必須使用者應用程式本身來實現。

二、tinyOS2

解除 tinyOS1 的 31 個使用者任務的限制條件，其餘功能與 tinyOS1 皆相同。程式設計師可隨心所欲使用任意多個任務，只需考慮微控制器的 RAM 容量限制，另 tinyOS2m 可提供同時等待多個信號量的服務。

三、tinyOS3

除了與 tinyOS2 一樣無使用任務數量的限制之外，系統核心還提供任務之間傳遞訊息與旗

標事件的功能，是 tinyOS 功能較齊全的作業系統版本。

四、tinyOS4

以功能服務完備為最高設計考量，除了結合 tinyOS2m 與 tinyOS3 之外，再提供佇列、動態記憶體配置與記憶體管理等服務。

雖然以功能性而言是 tinyOS4> tinyOS3 > tinyOS2 > tinyOS1，但若是比較相同的應用程式所搭配的作業系統在編譯後的 RAM 記憶體需求，記憶體需求大小依序是tinyOS4 > tinyOS3 ≥ tinyOS2 ≥ tinyOS1。

參、特色與理論

一、啟動

tinyOS1 非常容易啟動不需要繁瑣的設定程序與使用規則，只須將 SIZE.h 檔案中的兩個常數設定好就可以成功啟動，如此在教學上學生即可將心思專注在作業系統的核心程式並且非常容易實習實作，而一般使用者只需專注於應用程式的開發不太須要在意作業系統的程序性要求，所以應用程式與作業系統核心之間的溝通介面非常簡潔。

二、狀態

每一個任務可以存在就緒、等待與執行三種狀態之一，任何時候只有一個任務可以處於 CPU 執行狀態。作業系統啟動時，預設是所有的任務皆處於就緒狀態，任務就緒就是將就緒表中代表該任務之位元值設定為 1，使用者必須指定一個即將執行的任務以供 CPU 執行，任務在執行狀態獲得局部滿足之後將主動進入等待狀態或被動進入就緒狀態。任務於等待狀態可因獲取事件或等待逾時而進入就緒狀態，不可由等待狀態直接進入執行狀態。任務於就緒狀態進入執行狀態是因為具備相對最高優先權而被排程器所挑選執行，就緒狀態的任務也不能直接進入等待狀態。

三、優先權與就緒表

多任務切換的常用技巧是賦予每一個任務一個優先權值，排程器再依據就緒狀態任務的優先權值來決定提供給 CPU 執行的任務，一個任務的優先權值與就緒狀態的映射技巧會決定這個作業系統的設計架構與執行效率。tinyOS1 的就緒表只允許 31 個使用者任務，由於優先權值與就緒表的映射邏輯非常直覺簡潔，tinyOS1 只需極少量的記憶體便可順利正常運作，簡潔的映射邏輯是重要原因之一。

四、任務

任務是一種特殊的函數，是由程式設計師所負責開發的應用程式，任務函數不可以輸出回傳值，函數的主體必須是無窮迴圈，tinyOS1 所定義的任務並無輸入，如果需要輸入資料就要使用全域變數來傳遞，任務的型式如下所示：

```
void task(void)
{
   .................
   .................
   while(1)
   {
   .................
   .................
   }
}
```

五、任務函數指標陣列

一個任務的優先權值必須由程式設計師所賦予，tinyOS1 定義優先權值必須是正整數或 0 且不可以重複，優先權值越小則優先權越高，所以優先權值為 0 的任務將具有最高優先權，而任務的優先權值之指定是藉由函數指標陣列所指定的初始值順序來實現，也就是說任務的優先權值其實就是函數指標陣列的索引值。tinyOS1 使用這個創新技巧大幅度地簡化了作業系統的啟動設定程序，也大幅度降低了人為指定優先權值的錯誤機率，舉例而言若應用程式有 31 個任務其名稱依序為 task0, task1, ···, task30 且其優先權值依序為 0, 1, ···, 30，則任務函數指標陣列應如下給予初始值：

```
void (*taskName[ ])(void)={task0, task1, task2, task3, task4, task5, task6, task7,
task8, task9, task10, task11, task12, task13, task14, task15, task16, task17, task18,
task19, task20, task21, task22, task23, task24, task25, task26, task27, task28, task29,
task30};
```

如此並不需要額外程式來指定任務的優先權值，tinyOS1 這個獨創技巧就是指定了陣列初始值的順序就等同於指定了優先權值，這個技巧大幅度地降低了核心預防錯誤優先權值的程式複雜度。

六、事件

tinyOS1 只定義信號量事件可供資源存取或任務順序同步之控制，所有的任務使用事件函數

並不需要前置設定的程序。

七、等待函數

tinyOS1 規定每一個使用者任務都必須呼叫等待函數以使自己進入等待狀態而讓其他任務有機會獲得 CPU 的執行權，tinyOS1 所提供的等待函數有以下三個：pendSemOS( )、delayTimeOS( )與 delayTickOS( )。

肆、程式內容

tinyOS1 包含三個檔案：SIZE. h、OS. h 與 os. c，以及實現內文交換的中斷向量程式 PendSV_Handler，本文範例中 tinyOS1 所有原始程式碼如下所示：

一、SIZE.h

```
#define   TASKSIZE      (int)31
#define   PADDING       (char) 7
```

二、OS.h

```
#define   OSCLOCK_1S          SystemCoreClock
#define   OSCLOCK_500mS       SystemCoreClock/2
#define   OSCLOCK_200mS       SystemCoreClock/5
#define   OSCLOCK_100mS       SystemCoreClock/10
#define   OSCLOCK_10mS        SystemCoreClock/100
#define   OSCLOCK_1mS         SystemCoreClock/1000
#define   OSCLOCK_100uS       SystemCoreClock/10000
#define   OSCLOCK_10K         10000
#define   OSCLOCK_100K        100000
#define   OSCLOCK_1M          1000000

#define   clockOS    (unsigned int) OSCLOCK_100mS

#define   TIMEOUT_INFINITE (int) -1

char startOS(void (*[ ])(void), int, int, unsigned int);
int   findOptimalPaddingOS(void);
```

```
void delayTickOS(int);
void delayTimeOS(int, int, int, int);
char chechOwnEventOS(void);
void postSemOS(char);
void deleteSemNumberSelfOS(void);
void pendSemOS(char, int);
```

三、startup_LPC11xx.s

startup_LPC11xx.s 是 Keil MDK 所提供之晶片啟動檔案，PendSV_Handler 是該晶片啟動檔案所定義之中斷向量程式名稱，將 startup_LPC11xx.s 內之 PendSV_Handler 內部程式碼以下方之組合語言程式改寫即可。

PendSV_Handler 主要的功能是要處理內文交換(context switch)，所謂內文交換是指將當下正在執行的任務其在 CPU 暫存器內的資料壓入該任務所專屬的堆疊記憶體空間儲存，而將下一個即將執行的任務其在專屬的堆疊記憶體空間內的資料取回至 CPU 內所對應的暫存器。使用中斷向量程式來處理內文交換的一個重大的好處就是：ARM 微處理機的 NVIC 硬體控制器在接獲中斷要求後會自動執行 8 個敏感暫存器(R0、R1、R2、R3、R12、LR、PC 與 XPSR)的壓入與取回動作，PendSV_Handler 中斷向量程式僅須處理剩餘相關的另外 8 個暫存器資料即可。

```
PendSV_Handler PROC
```

以下程式碼是將 CPU 的 8 個暫存器資料儲存至任務堆疊。

```
MRS      R0, PSP
SUBS R0, #16
STMIA R0!, {R4-R7}
MOV      R4, R8
MOV      R5, R9
MOV      R6, R10
MOV      R7, R11
SUBS R0, #32
STMIA R0!, {R4-R7}
SUBS R0, #16
IMPORT CurrentTaskOS
LDR      R2,=CurrentTaskOS
LDR      R1, [R2]
STR      R0, [R1]
```

以下程式碼是從任務堆疊取回 CPU 的 8 個暫存器資料。

```
IMPORT    NextTaskOS
LDR       R2,=NextTaskOS
LDR       R1, [R2]
LDR       R0, [R1]
LDMIA     R0!, {R4-R7}
MOV       R8, R4
MOV       R9, R5
MOV       R10, R6
MOV       R11, R7
LDMIA     R0!, {R4-R7}
MSR       PSP, R0

LDR R3,=0xFFFFFFFD
BX        R3
ENDP
```

C 語言非常適合處理周邊界面的控制，對於龐大連續記憶體資料的存取也是非常有效率，但是 C 語言並無法對 CPU 內部的暫存器作存取，內文交換就是將正在 CPU 執行的當下任務之暫存器資料移至外部堆疊記憶體，且將下一個要執行的任務資料從外部堆疊記憶體移至 CPU 對應的暫存器，所以內文交換並無法以 C 語言來實現而必須使用該微控制器所支援的指令集以組合語言來實現，tinyOS1 的內文交換是參考 Adam Heinrich 的作法在晶片啟動檔案內的中斷向量程式 PendSV_Handler 內來實現[3]。

每一個任務的堆疊至少須有 16 個空間來儲存 CPU 暫存器的資料，每一個堆疊空間必須是 4bytes 大小因為暫存器是 32 位元長度，任務堆疊是由高位址往低位址壓入而由低位址往高位址取回進而實現先進後出的功能。

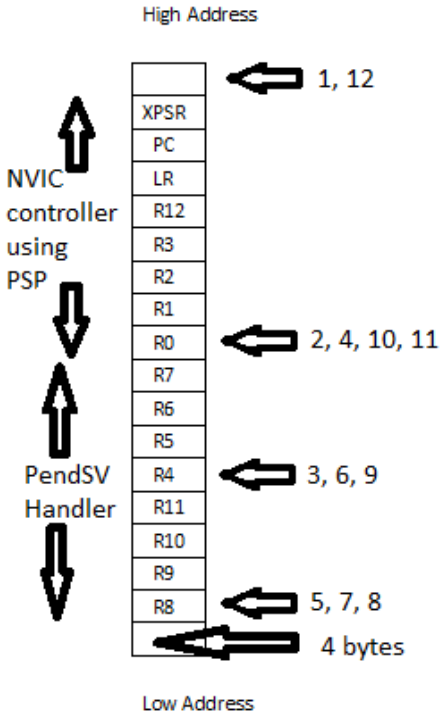
ARM Cortex-M0 微處理機的架構設計是在接獲中斷要求後，在執行中斷向量程式之前，NVIC 控制器硬體會依據 PSP 所指定的位址自動壓入 R0、R1、R2、R3、R12、LR、PC 與 XPSR 等 8 個暫存器的資料，因為 M0 微控制器是先減 4bytes 再壓入資料，因此在執行 PendSV_Handler 之前，PSP 必須是先指向 XPSR 所儲存位址再加 4 才能執行壓入之動作；而在 NVIC 控制器作取回動作之前，因為 M0 微控制器是從 PSP 所指的位址直接取回資料，故 PSP 必須是指向 R0 所儲存的位址。

PendSV_Handler 程式就是在執行所剩下的 R4 至 R11 等 8 個暫存器資料的壓入與取回，壓入指令是使用 STMIA 而取回指令是使用 LDMIA，該二指令並不需要使用到 PSP，但仍須指定壓入與取回的位址給該二指令，且由於在 ARMv6-M 指令集中該二指令並無法存取 R8 以上的高階暫存器，所以必須先將 R4 至 R7 等 4 個低階暫存器先壓入堆疊，以便騰出空間將 R8 至 R11 等高階暫存器移至 R4 至 R7 後才能夠繼續讓 STMIA 來執行。

一個任務的堆疊，要執行壓入的起始位址是在 XPSR 所在位址再加 4，該位址就是當下 PSP

所指定的位址，當該執行狀態之任務主動或被動進行內文交換時，NVIC 控制器便可以立即執行壓入動作來保存此執行狀態任務的資料;而取回的堆疊起始位址是在 R8 資料所儲存的位址，每一個任務都必須維護保存自己的堆疊起始位址，以便一旦該任務成為就緒狀態之最高優先權時，就能夠立即從就緒狀態順利轉換為執行狀態。

PendSV_Handler 依時間順序的程序說明如圖 1 所示，⇦ 代表所指到之 SRAM 堆疊的位址。



1： PSP before PendSV_Handler

2： PSP after (MRS R0, PSP) for push

3： R0 = PSP-16

4： R0 after(STMIA R0!, {R4-R7})firstly

5： R0 = PSP - 32 (SUBS R0, #32)

6： R0 after (STMIA R0!, {R4-R7}) secondly

7： CurrentTaskOS (SUBS R0, #16), SP (TaskOS [current]. sp)

8： NextTaskOS (pop), SP (TaskOS [next]. sp)

9： R0 after (LDMIA R0!, {R4-R7}) firstly

10： R0 after (LDMIA R0!, {R4-R7}) secondly

11： PSP afte r(MSR PSP, R0) for pop

12： PSP after (BX R3)

圖 1 PendSV_Handler 依時間順序的程序說明

四、os.c

使用 NXP LPC1114 系列的 Cortex-M0 平台作測試。

```
#include <LPC11xx.h>
#include "SIZE.h"
#include "OS.h"
#define   DISABLE_INTERRUPT__ASM{ cpsid i }
#define   ENABLE_INTERRUPT__ASM{ cpsie i }

typedef struct

{
    unsigned int      sp;
    int               pack [PADDING];
    unsigned int      registerStack [16];
    int               priority;
    int               waitTick;
}   OStask;

OStask            *CurrentTaskOS;
OStask            *NextTaskOS;
OStask            TaskOS [TASKSIZE+1];
unsigned int      ReadyTableOS=0x0;
int               CurrentPriorityOS;
unsigned int      TickPerSecondOS;
char              SemNumberTaskOS[TASKSIZE];
char              PriorityOwnEventOS[TASKSIZE];
char              SysTickCountOS=0;
char              ReadSysTickCountOS=0;

void set ReadyTableOS(int priority)

{
    ReadyTableOS|=(1<< priority);
}

void clearReadyTableOS(int priority)
```

```
    {
        ReadyTableOS &=～(1<<priority);
    }

__ASM   int findLeastBit(unsigned int Table)

    {
        MOVS R3, #0
    loop
        LSRS R0, R0, #1
        BCS   return
        ADDS   R3, R3, #1
        CMP   R3, #32
        BLT   loop
    return
        MOV   R0, R3
        BX     LR
    }

void executeHighestPriorityTaskOS( )

    {
        int highestPriority;
        if
    ( (PriorityOwnEventOS[CurrentPriorityOS]<1)||(CurrentPriorityOS==TASKSIZE) )
        {
          DISABLE_INTERRUPT;
                highestPriority=findLeastBit(ReadyTableOS);
                if ( highestPriority!=CurrentPriorityOS)
                {
                    CurrentTaskOS=&TaskOS[ CurrentPriorityOS];
                    NextTaskOS=&TaskOS[highestPriority];
                    CurrentPriorityOS=highestPriority;
                    SCB->ICSR|=1<<28;
        }
    ENABLE_INTERRUPT;
        }
    }
```

```
void initializeTaskOS( void (*handler)(void), int priority)

    {
      DISABLE_INTERRUPT;
        if ( handler !=0x0)
        {
            PriorityOwnEventOS[priority]=0;
            setReadyTableOS(priority );
            TaskOS[priority].priority=priority;
            TaskOS[priority].waitTick=0;
            TaskOS[priority].sp=(unsigned int)( &TaskOS[priority].registerStack[0]);
            TaskOS[priority].registerStack[15]=0x01000000;
            TaskOS[priority].registerStack[14]=(unsigned int)handler;
        } // if
      ENABLE_INTERRUPT;
    }

void idleTaskOS(void)

    {
        while(1)
     {
     }
    }

char startOS (void (*taskName[ ])(void), int arraySize, int startPriority, unsigned int OS_clock)

    {
        char i;
        if (   arraySize !=TASKSIZE )
        {
              return 1;
        }
        if (   arraySize>=32 ) // maximum TASKSIZE is 31
        {
              return 2;
       }
        for ( i=0; i <=(TASKSIZE-1); i++)
```

```
        {
            SemNumberTaskOS[i]=(char)-1;
            initializeTaskOS(taskName[i], i);
          }
            initializeTaskOS(idleTaskOS, TASKSIZE);
            NVIC_SetPriority(PendSV_IRQn, 0xff );
            NVIC_SetPriority(SysTick_IRQn,0x0 );
            SysTick_Config( OS_clock);
            TickPerSecondOS=SystemCoreClock / OS_clock;
            CurrentPriorityOS=startPriority;
            CurrentTaskOS=&TaskOS[CurrentPriorityOS];
            __set_PSP(CurrentTaskOS->sp+64);
            __set_CONTROL(0x03);
            __ISB( );
            taskName[CurrentPriorityOS]( );
            return 0;
    }// st artOS

int findOptimalPaddingOS( )

    {
        int pack;
        int i;
        int maximum=-1;
        for ( i=0; i< TASKSIZE; i++)
        {
            pack=PADDING-( (int)TaskOS[i].sp-(int)&TaskOS[i].pack[0] ) / 4;
            if ( pack>maximum)
            {
                maximum=pack;
            }
        }
        return maximum;
    }
```

```
char chechOwnEventOS( )
    {
        return PriorityOwnEventOS[CurrentPriorityOS];
    }

void postSemOS(char semNumber)
    {
        char i;
        if (   (int)semNumber>=0 )
        {
            DISABLE_INTERRUPT;
            PriorityOwnEventOS[CurrentPriorityOS]=0;
            SemNumberTaskOS[CurrentPriorityOS]=(char)-1;
            for ( i=0; i <TASKSIZE; i++ )
            {
                if ( SemNumberTaskOS[i]=semNumber )
                {
                    SemNumberTaskOS[i]=(char)-1;
                    PriorityOwnEventOS[i]=1;
                    TaskOS[i].waitTick=0;
                    setReadyTableOS(i);
                }
            }
            ENABLE_INTERRUPT;
        } //   if ( (int)semNumber>=0 )
        executeHighestPriorityTaskOS( );
    }

void deleteSemNumberSelfOS( )
    {
        SemNumberTaskOS[CurrentPriorityOS]=(char)-1;
    }
```

```
void pauseTaskOS(int timeout)

    {
        PriorityOwnEventOS[CurrentPriorityOS]=0;
        TaskOS[CurrentPriorityOS].waitTick=timeout;
        clearReadyTableOS(CurrentPriorityOS );
    }

void pendSemOS(char semNumber, int timeout)

    {
        while( SysTickCountOS=ReadSysTickCountOS )
        {
        }
        ReadSysTickCountOS=SysTickCountOS;
        DISABLE_INTERRUPT;
                SemNumberTaskOS[CurrentPriorityOS]=semNumber;
                pauseTaskOS(timeout);
        ENABLE_INTERRUPT;
        executeHighestPriorityTaskOS( );
    }

void SysTick_Handler(void)

    {
        int i;
        char schedule=0;
        DISABLE_INTERRUPT;
        for( i=0; i<=TASKSIZE-1; i++ )
        {
            if ( TaskOS[i].waitTick>=1 )
            {
                TaskOS[i].waitTick--;
            }
            if ( TaskOS[i].waitTick=0 )
            {
                setReadyTableOS(i);
                schedule=1;
            }
```

```
            } // for
            SysTickCountOS++;
        ENABLE_INTERRUPT;
        if ( schedule )
            {
                executeHighestPriorityTaskOS( );
            }
    }

void delayTickOS(int tick)

    {
        if ( tick>0 )
        {
            DISABLE_INTERRUPT;
                pauseTaskOS(tick);
            ENABLE_INTERRUPT;
        }
        executeHighestPriorityTaskOS( );
    } // DelayTick

void delayTimeOS( int hour, int minute, int second, int mS)

    {
        int tick;
        if ( (hour>=0) && (minute>=0) && (second>=0) && (mS>=0) )
        {
            tick=TickPerSecondOS * ( hour*3600 + minute*60 + second );
            tick += TickPerSecondOS * mS / 1000;
            if ( tick<1 )
            {
                tick=1;
            }
            DISABLE_INTERRUPT;
                pauseTaskOS(tick);
            ENABLE_INTERRUPT;
        }
        executeHighestPriorityTaskOS( );
    }
```

五、main.c

main.c 檔案是作者所示範的應用程式範例並不屬於作業系統的一部分，作者是以 LPC1114FN28/102(4KB SRAM)在麵包板上實作 Cortex-M0 微控制器的應用教學來作為教材的呈現方式，開發環境是使用 Keil uVision5。

在教學方法上，tinyOS1 的程式邏輯設計是課程的核心，除了強調 C 語言的程式設計技巧之外，教學過程中可以先不啟動作業系統(不要呼叫 startOS( ))，而在 main( )函數內依據作業系統函數的輸入要求，逐一呼叫 os.c 檔案內的核心函數並利用 LPC1114FN28/102 微控制器的周邊功能來輸出結果(例如使用 GPIO 或是 UART 通信)，如此學生便能夠深刻體會核心函數的功能。

啟動的步驟僅有以下 2 項，非常簡潔容易，第 3 項步驟可選擇性更改：

1. 建立應用程式專屬 C 檔案如本例之 main.c 檔案，主體是任務程式的功能設計，對於 C 語言而言每一個任務都是一個函數，將 main( )函數內真正使用到的任務數量指定給 SIZE. h 中的 TASKSIZE。
2. 適當指定 SIZE. h 中的 PADDING，該值若太小則作業系統將誤動作或無法執行，該值若太大則會浪費記憶體空間，可由最後執行的任務呼叫 findOptimalPaddingOS( )函數，最佳 PADDING 值會在該函數回傳值附近。
3. 可在 OS. h 中更改預設的作業系統時脈 clockOS。

主程式所展示的功能是一個 M0 微控制器能夠同步依序執行 31 個任務，每一個任務都有自己的代碼(代碼是 0～9 與 a～u 總共 31 個)，每一個任務藉由 sendByte( )函數與 PC 通信然後將自己的代碼顯示在螢幕上的通信軟體 AccessPort，sendByte( )函數是作者所開發之應用程式暫略其程式碼，該函數將會輸出一個 ASCII 碼至螢幕而實現微控制器與 PC 之間的 UART 通信。順序同步的功能是使用信號量讓一個任務去啟動執行另一個特定任務，然後自己釋出 CPU 執行權而進入等待狀態，測試過程將額外展示 PADDING 設定稍微不足時，應用程式所執行的瑕疵與錯誤結果。

```
#include <LPC11xx.h>

#include "OS.h"

void task0(void)

    {
        while(1)
        {
            sendByte('0');
            pendSemOS(0, TIMEOUT_INFINITE);
            postSemOS(1);
        }
    }
```

```
void task1(void)
    {
        while(1)
        {
            sendByte('1');
            pendSemOS(1, TIMEOUT_INFINITE);
            postSemOS(2);
        }
    }
```

略過 task2 到 task29。

```
    .....
    .....
    .....
    .....
    .....
```

task30是等待0.4秒之後送出信號量0來啟動task0，task0至task29皆在等待信號量，故task30必須以逾時而主動規律性啟動。

```
void task30(void)
    {
        while(1)
        {
            sendByte('u');
            postSemOS(0);
            delayTimeOS(0,0,0, 400);
        }
    }

    void (*taskName[ ])(void)={task0, task1, task2, task3, task4, task5, task6, task7,
    task8, task9, task10, task11, task12, task13, task14, task15, task16, task17, task18,
    task19, task20, task21, task22, task23, task24, task25, task26, task27, task28, task29,
    task30};

    // errorCode: 1- arraySize !=TASKSIZE
    // errorCode: 2- arraySize>=32
```

```
int main( )
    {
        char errorCode;
        int startTaskIndex;
        int arraySize;
        initializeUART(9600);
        arraySize = sizeof(taskName)/sizeof(taskName[0]);
        startTaskIndex=0;
        errorCode=startOS(taskName, arraySize, startTaskIndex, clockOS);
        sendByte('0'+ errorCode);
    } // main
```

伍、實驗測試

先在 PC 上執行 AccessPort 軟體，且正確設定該軟體所需的鮑率(Baud rate)與 UART 通信連接埠，在 main( )函數執行後電腦螢幕上將會出現 31 個任務的代碼 0 1 2 3 4 5 6 7 8 9 a b c d e f g h I j k l m n o p q r s t u 且一再循環如圖 2 所示。測試的條件是更改 SIZE.h 中的 PADDING 數值以觀察臨界條件附近的執行結果，下圖依序是 PADDING 數值為 7、6 與 4 之執行結果，圖 2 是將 PADDING 數值設為 7 是可正常執行的最小值，若將 PADDING 數值改為 6 時可發現執行結果會出現兩個 t 的微小瑕疵如圖 3 所框，若將 PADDING 數值改為 4 時，執行時會出現記憶體堆疊空間錯置而執行順序紊亂的結果如圖 4 所示。

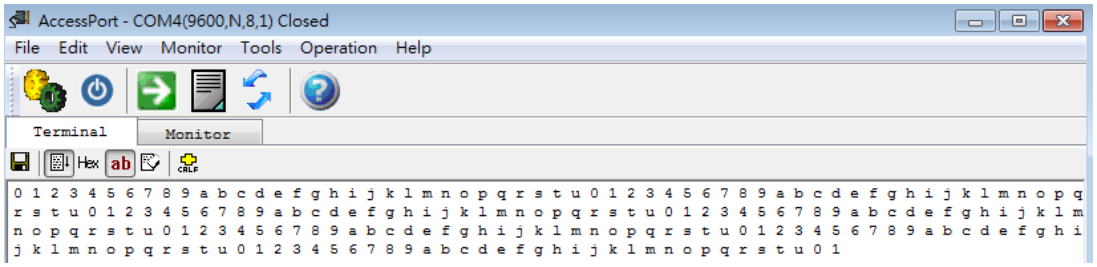


圖 2　PADDING 數值為 7 是本例可正常執行的最小值

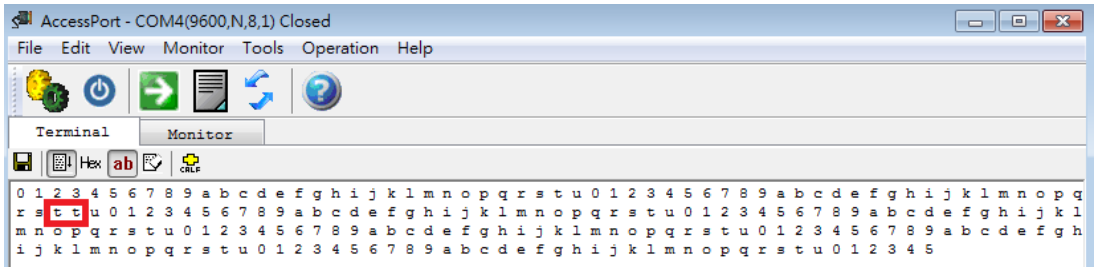


圖 3　PADDING 數值為 6 時會出現兩個 t

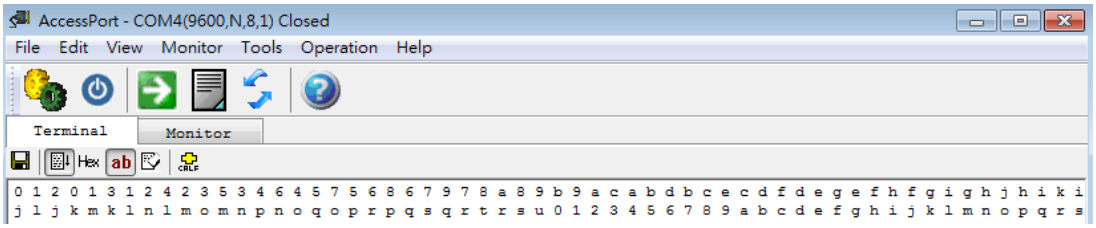


圖 4　PADDING 數值為 4 時會執行順序紊亂

陸、結論

tinyOS 是目前唯一可取得原始程式碼能夠在 4KB RAM 之 ARM Cortex-M0 微控制器順暢執行的即時作業系統，tinyOS1 的實驗結果已經證明這個結論，本文所展示的核心與應用程式之原始程式碼電子檔已提供給《高雄師大學報》，tinyOS 也提供四種不同的版本，使用者可依據微控制器的硬體規格與應用程式的功能需求來選擇嵌入作業系統的版本。

參考文獻

[1] JEAN J. LABROSSE 著，黃文增譯(2005)。MicroC/OS-II：即時作業系統核心。新北：全華。

[2] Joseph Yiu 著，阮聖彰、莊文維譯(2014)。ARM Cortex-M0 嵌入式系統設計入門。臺北：旗標。

[3] Adam Heinrich (2016). A simple context switcher for Cortex-M0 processors. Retrieved from https://github.com/adamheinrich/os.h/tree/blog_2016_07