EduCake 使用矩陣鍵盤

一、 矩陣鍵盤原理介紹

關於使用者與 86Duino EduCake 的互動方式，除了前面章節提到的感測器應用、數位類比轉換器讀取、Serial Port 傳輸資料等方式外，還有一種常見的輸入方式，就是使用「矩陣鍵盤」這種裝置，通常用在控制器需要讓使用者輸入命令、資料等等用途上。其實矩陣鍵盤的原理，就跟先前章節介紹過的「微動開關按鍵」結合「digitalRead( ) 函式」類似，只是從單一按鍵變成了按鍵矩陣。不過也因為按鍵數量變多，控制程式撰寫方式也變得不同了，本章節就從矩陣鍵盤的原理介紹開始，並使用 86Duino EduCake 來實作一些有趣的功能吧。

一般市面上可以買到的矩陣鍵盤種類多樣化，有的是薄膜接觸式導通，有的是機械式微動開關等，不過使用原理大同小異。本章節討論的小型矩陣鍵盤而言，大小常見

3X4 或 4X4 等，上面可能標示 0~9、英文符號、其他符號如「*」「#」等等，例如計算機的鍵盤就是個很好的例子。讀者如果想要製作屬於自己的特殊鍵盤布局也可以，看完本章節後就可以自己動手做囉。

當專案裡面使用的按鍵數量只需要手指就數得出來，實作上其實只要一個按鍵對應一個 digital 腳位就可以，但是當使用了 3X4 或更大的鍵盤時，需要的針腳數量一下子就增加很多了，當然也會佔據控制器寶貴的針腳空間。因此，矩陣鍵盤設計上會使用「掃描」的概念，逐行逐列作按鍵狀態的掃瞄，便可以節省針腳數量，就跟前面提到的 8X8 LED 矩陣掃描顯示是一樣的道理。以一個 4X4 矩陣鍵盤為例，電路接線如下面圖 1 所示：

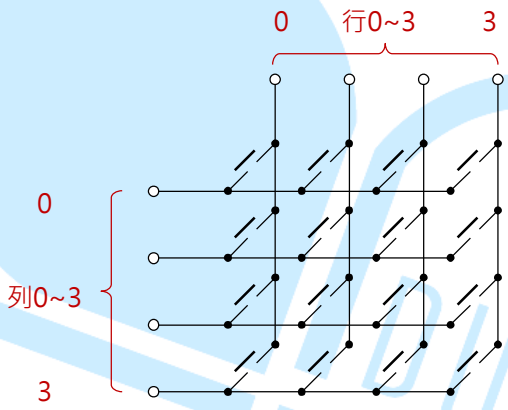


圖 1. 4X4 矩陣鍵盤電路接線圖

讀者可以從電路圖上看到，依照這樣的連接方式，4X4 的矩陣鍵盤只需要占用控制器的 8 隻針腳，而不是 16 隻針腳，相對「一對一讀取按鍵方法」可以節省一半針腳數量。對某一列的按鍵來說，按鍵一側是全部相通的，不過這樣的接線方式造成程式需要使用較複雜的流程來處理。掃描流程中，控制器程式會依序對某一列給予 HIGH 或 LOW 的電壓值（依使用狀況決定），然後讀取這一列上每個行的按鍵電壓，掃描流程如下圖 2 所示。

圖 2. 4X4 矩陣鍵盤掃描流程示意圖

這邊須注意，依序掃描時，一次只能有一列的電壓是 HIGH 或 LOW，其他列必須給相反的電壓，否則讀取某行電壓時會分辨不出是哪一列的按鍵被按下了。接下來的 86Duino EduCake 實作上，我們選用一般容易買到的 4X4 矩陣鍵盤模組，如下圖 3 所示：

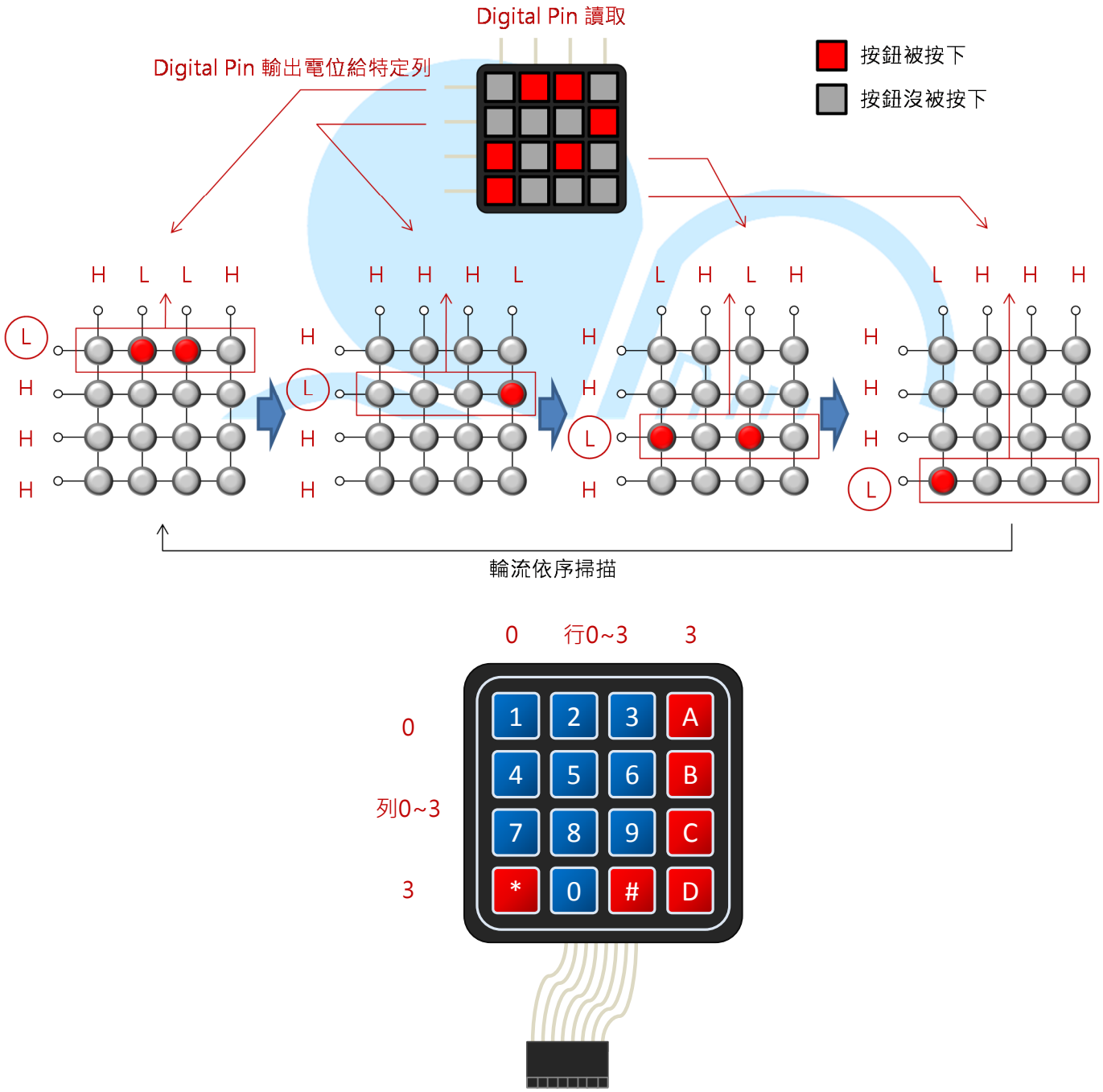


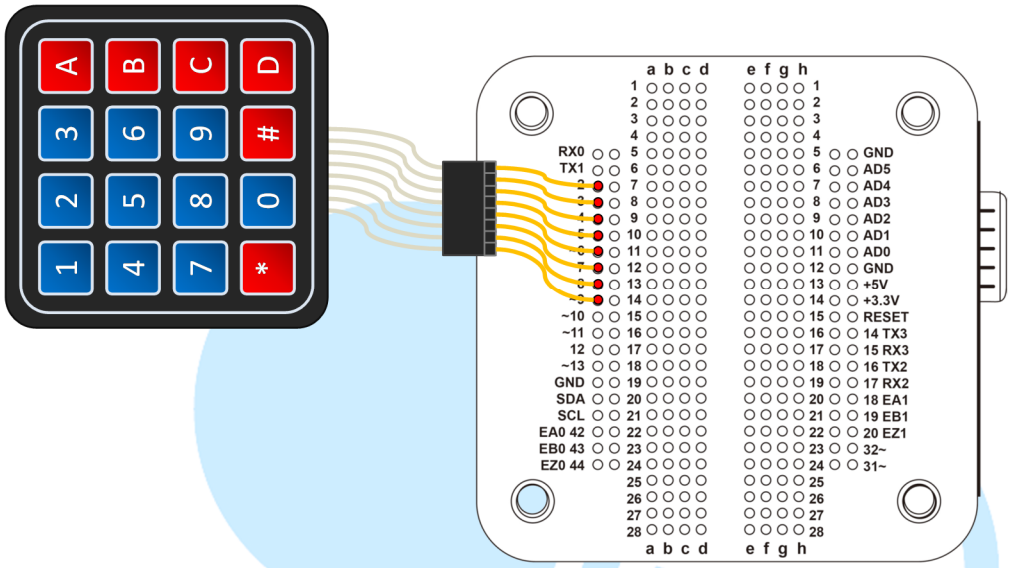
圖 3. 4X4 矩陣鍵盤針腳定義

以這個矩陣鍵盤模組來說，按鍵按下是電路導通，各行列的接線已經被排列為 8 針腳插頭方便安裝，針腳定義由左到右依序是「列 0，列 1，列 2，列 3，行 0，行 1，行 2，行 3」。不過若讀者實作時買的是其他鍵盤模組，最好先使用三用電表搭配手指按下特定按鍵，先測量好針腳的定義，這樣接線與寫程式時才能正常運作喔。

由於 86Duino EduCake 可用的腳位數量約有二十幾個（AD 腳位可作為讀取用途），因此本章節的實作方式，較適合用在這種中等按鍵數量的矩陣鍵盤上，如果讀者希望自己動手做一個像電腦鍵盤那麼多按鍵的裝置，最好還是搭配專用的 IC 較佳。下面便讓我們利用程式，實際練習矩陣鍵盤的原理，並使用 4X4 矩陣鍵盤模組做些實際應用的功能吧。

二、 第一個程式 – 鍵盤掃描原理練習

第一個範例程式先來練習如何使用 86Duino EduCake 實作上述矩陣鍵盤掃描原理，讀者請先依下圖接線：



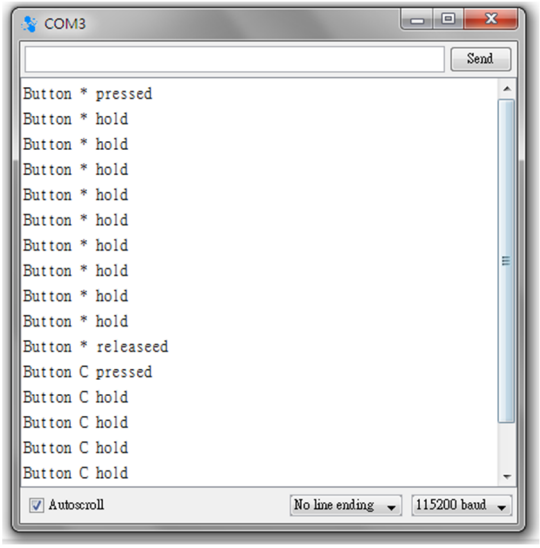
列 0~3，行 0~3 依序接到數位腳位 9~2 即可，接著請打開 86Duino Coding IDE，輸入以下程式碼：

```
const int Rows = 4; // 行數
const int Cols = 4; // 列數
// 按鍵對應符號
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
// 紀錄按鍵上一次狀態
bool keys_status_last[Rows][Cols] =
{
  {false,false,false,false},
  {false,false,false,false},
  {false,false,false,false},
  {false,false,false,false}
};
      // 定義針腳編號
```

```
int row_pins[Rows] = {9, 8, 7, 6};// 列 0~3
int col_pins[Cols] = {5, 4, 3, 2};// 行 0~3

void setup()
{
  // 設定 Pin IO 模式
  // 用行的 pin 讀取按鍵電壓狀態
  for(int col = 0; col < Cols; col++)
  {
    pinMode( col_pins[col], INPUT_PULLUP );
  }
  // 用列的 pin 當電壓源
  for( int row = 0; row < Rows; row++) // 掃描行
  {
    pinMode( row_pins[row], OUTPUT );
    digitalWrite( row_pins[row], HIGH );
  }

  Serial.begin(115200);
}


void loop( )
{
  for( int row = 0; row < Rows; row++ ) // 掃描列
  {
    digitalWrite( row_pins[row], LOW ); // 此列電壓變為 LOW
    for( int col = 0; col < Cols; col++) // 掃描行
    {
      // 讀取這行的電壓，如果按鈕有按下導通，電壓為 LOW
      boolean result = !digitalRead( col_pins[col] );

      // 這次按鈕按下，上一次是按下，則按鍵為持續壓按狀態
      if( result == HIGH && keys_status_last[row][col] == true )
      {
        Serial.print("Button ");
        Serial.print(keys[row][col]);
        Serial.println(" hold");
      }
      // 這次按鈕按下，上一次不是按下，則按鍵為剛被按下狀態
```

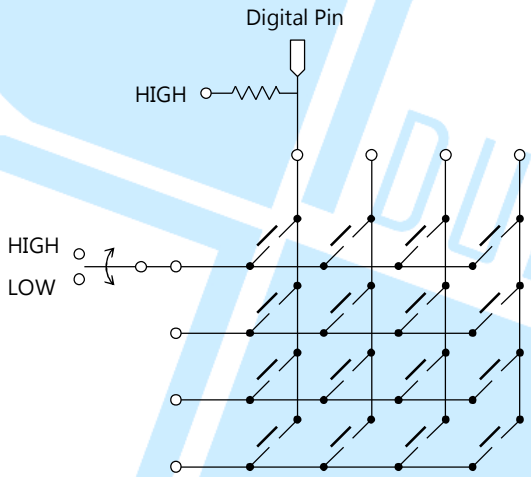
```
      else if( result == HIGH && keys_status_last[row][col] == false )
            {
               Serial.print("Button ");
               Serial.print(keys[row][col]);
               Serial.println(" pressed");
            }
            // 這次按鈕不是按下，上一次是按下，則按鍵為剛被放開狀態
            else if( result == LOW && keys_status_last[row][col] ==
true )
            {
               Serial.print("Button ");
               Serial.print(keys[row][col]);
               Serial.println(" releaseed");
            }
            keys_status_last[row][col] = result; // 更新按鍵狀態
          }
          digitalWrite( row_pins[row], HIGH ); // 此列電壓變為 HIGH
        }

        delay( 20 );
          }
```

編譯並上傳程式後，請讀者打開 Serial Monitor，並嘗試按下鍵盤的按鍵，Serial Monitor 中將會顯示被按下的按鍵對應符號，如下圖：



程式一開始，先設定矩陣鍵盤的行數 Cols 與列數 Rows，char 矩陣 keys[Rows][Cols] 用在儲存鍵盤對應字元符號，而 bool 矩陣 keys_status_last[Rows][Cols]用在儲存每個按鍵的上一次狀態，按下為「true」，沒按下為「false」。針腳編號矩陣則用在宣告鍵盤行列實際對應的針腳編號，以及方便後續的存取，例如使用語法 row_pins[1]，便可以取得列 1 的針腳編號。

setup( )階段則設定了各個數位腳位的 I/O 模式，這裡使用列 pin 當作電壓來源，而行 pin 用在讀取電壓狀態。另外執行了 Serial Port 的初始化。須注意數位輸入的部分使用了 INPUT_PULLUP 模式，而掃描列時給予 LOW 的電壓(其餘列給 HIGH)，因此當按鍵被按下時會讀取到 LOW，沒按下時都是 HIGH，這樣可讓按鍵電壓狀態的讀取更為穩定。電路示意如下圖：



loop( )迴圈裡，使用了雙層 for 迴圈，第一層為掃描列用，第二層用在掃描行。雙層迴圈會對按鍵進行電壓讀取，達到前述的掃描原理。

由於按鍵按下時電壓為 LOW，因此程式碼「boolean result = !digitalRead(col_pins[col]);」中有一個反向的動作，所以 result 為 true 時按鍵為按下，再將 result 的結果當作這次讀取按鍵的狀態。

這裡需要注意的是，keys_status_last 矩陣用在記錄各個按鍵的上一次狀態，因此能夠知道按鍵是剛被按下 (上一次 false，這一次 true) 、持續壓著 (上一次 true，這一次 true) 、或是被放開 (上一次 true，這一次 false)，視使用者需要的功能完整度來實現。這樣便簡單練習了鍵盤的掃描原理，如果讀者使用了不同大小的鍵盤、不同的字元符號，只要從前面幾個變數宣告做修改就可以囉。

三、 第二個程式 – 使用 Keypad 函式庫

了解 86Duino EduCake 與掃描矩陣鍵盤的基本原理後，接著來做點小變化。這個範例電路不用更改，使用與範例一相同的接線。

依照鍵盤的機械結構不同，按下按鈕與放開按鈕時實際上會有「彈跳」的狀況發生，接點在接觸與未接觸的狀態間快速變化，如果沒有處理的話，可能按一下按鈕，卻發現電壓其實起伏了好多下才穩定。去彈跳（也稱為 debounce）的方法有電路式，也有程式的處理方式，不過上面第一個程式並沒有處理去彈跳的問題，主要是因為 loop 間隔時間滿長的（20ms），讀取數位腳位的間隔已經超過了彈跳的變動時間，因此讀取按鈕電壓時不大會變動。如果讀者需要在高速讀取按鍵狀態的場合使用，就不太適合了。

還好，已經有現成的 Keypad 函式庫可以使用，且內部已經處理了去彈跳、掃描偵測按鍵狀態等等的功能，這個範例，我們就使用這個函式庫來做矩陣鍵盤讀取的練習吧。

請讀者先到網址：

http://playground.arduino.cc/uploads/Code/keypad.zip

下載 Keypad 的程式碼壓縮檔後，將解壓縮後的「Keypad」資料夾放到 86Duino IDE 所在資料夾路徑「86Duino_Coding_版本號碼_WIN\libraries」內。再使用文字編輯器打開 Keypad.h 檔案，然後找到「#include "WProgram.h"」修改成「#include <Arduino.h>」。然後對「Keypad/utility 資料夾」內的 Key.h 檔案做相同的修改動作。完成後，請打開 86Duino Coding IDE，輸入以下程式碼：

```
#include <Keypad.h>

const byte Rows = 4; // 行數
const byte Cols = 4; // 列數
// 按鍵對應符號
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
// 定義針腳編號
byte row_pins[Rows] = {9, 8, 7, 6}; // 列 0~3
byte col_pins[Cols] = {5, 4, 3, 2}; // 行 0~3

// Keypad lib 物件
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins, col_pins, Rows, Cols );
void setup( ){
 Serial.begin(115200);
}

void loop( ){

  if( keypad4X4.getKeys( ) )
  {
    for( int i = 0; i < LIST_MAX; i++) // 逐一檢查 keypad4X4 物件的按鍵列表
    {
      if( keypad4X4.key[i].stateChanged ) // 檢查按鍵狀態是否變動，如果有變動再取出內容
      {
        Serial.print("Button ");
        Serial.print(keypad4X4.key[i].kchar); // 目前被按下的按鍵字元符號
        switch( keypad4X4.key[i].kstate )
```

```
            {
              case PRESSED:
                Serial.println(" pressed.");
                break;
              case HOLD:
                Serial.println(" hold.");
                break;
              case RELEASED:
                Serial.println(" released.");
                break;
              case IDLE:
                Serial.println(" idle.");
            }
          }
        } // end for
      } // end if ( keypad4X4.getKeys( ) )

      delay( 20 );
```

編譯並上傳程式後，一樣打開 Serial Monitor，並嘗試按下鍵盤的按鍵，Serial Monitor 中將會顯示被按下的按鍵對應符號與按鍵狀態，程式的執行結果與範例一類似。

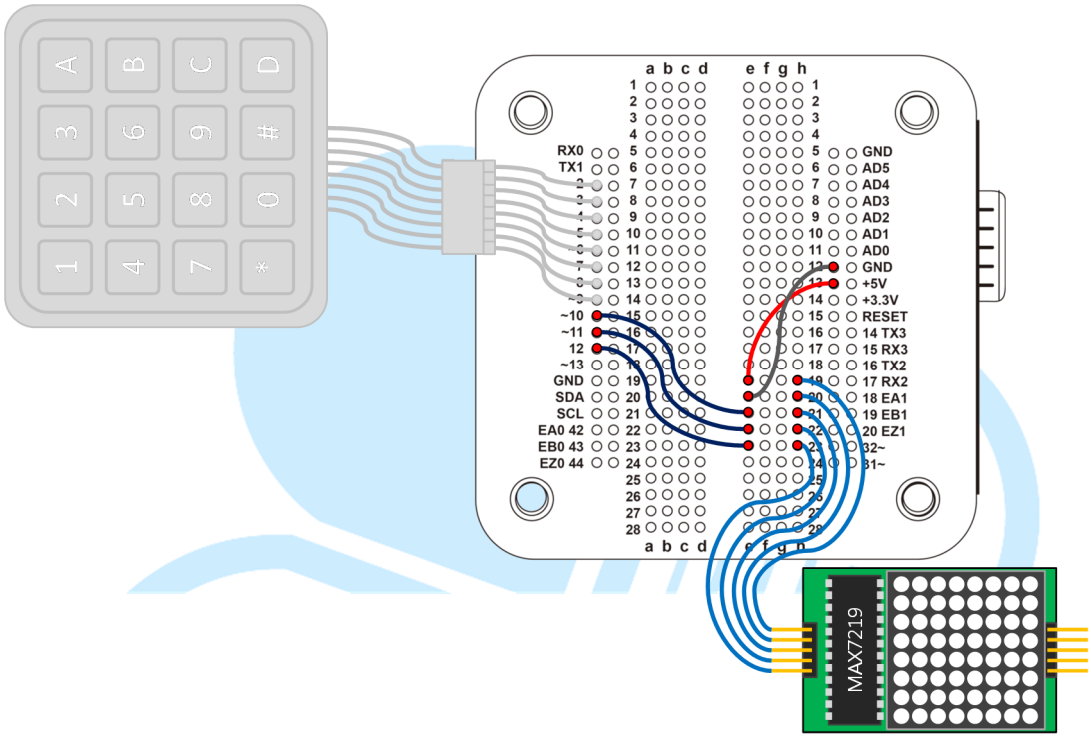
讀者會注意到，這個程式碼一開始的行數、列數宣告、按鍵對應符號、腳位對應都跟範例一的程式差不多，但是開頭多了「#include <Keypad.h>」，接著全域變數多了個 Keypad class 物件「Keypad keypad4X4 = Keypad(makeKeymap(keys), row_pins, col_pins, Rows, Cols);」，這便是 Keypad 函式庫的方便之處，將所有矩陣鍵盤的處理程式、變數定義都包裝到 Keypad 的 class 中，有效縮減.ino 檔案內的程式行數，也方便在不同專案程式內重複使用相同的 Keypad 函式庫。

接著在 setup( )階段，只做了 Serial Port 的初始化。loop( )迴圈內則使用「keypad4X4.getKeys( )」語法做鍵盤狀態的讀取與更新，接著才能用「for(int i = 0; i < LIST_MAX; i++)」語法逐一檢查 keypad4X4 物件的按鍵列表，判斷各個按鍵的觸發狀態。Keypad 函式庫提供：「keypad4X4.key[i].stateChanged」

用在檢查按鍵狀態是否改變、「keypad4X4.key[i].kchar」用在取得此按鍵對應的字元符號、「keypad4X4.key[i].kstate」用在取得此按鍵狀態等。按鍵狀態共有 PRESSED（被按下）、HOLD（持續壓著）、RELEASED（放開）、IDLE（沒被按著）。loop()迴圈內便是用這幾個函式與變數判斷按鍵狀態，然後印出訊息。使用了 Keypad 函式庫之後，程式碼看起來是不是變整潔多了呢？

四、　第三個程式 – 使用 Keypad 函式庫+8X8LED 矩陣模組

接著這個範例程式繼續使用 Keypad 函式庫，我們繼續來加入其他較複雜的功能。此範例程式會用到之前已經介紹過的 MAX7219+8x8 LED 矩陣模組，請讀者依照下圖加上 LED 矩陣的接線：



由於 MAX7219+8x8 LED 矩陣模組的功能之前已經講過，而且重複利用的機會很多， 因此這邊把相關的程式碼包裝成為 LEDmat8 函式庫，需要先做幾個步驟：

1. 在 86Duino IDE 所在資料夾路徑「86Duino_Coding_版本號碼_WIN\libraries」內，新增一「LEDmat8」資料夾。

2. 在「LEDmat8」資料夾內新增一文字檔案，改檔名為「LEDmat8.h」（注意副檔名也要一起改），然後在檔案內容填入以下程式碼並存檔：

```
#ifndef LEDMAT8_H
#define LEDMAT8_H

#if defined(ARDUINO) && ARDUINO >= 100
#include "Arduino.h"
#else
//#include "WProgram.h"
#include <Arduino.h>
#endif

    // MAX7219 暫存器定義
#define max7219_REG_noop        0x00
#define max7219_REG_digit0      0x01
#define max7219_REG_digit1      0x02
#define max7219_REG_digit2      0x03
#define max7219_REG_digit3      0x04
#define max7219_REG_digit4      0x05
#define max7219_REG_digit5      0x06
#define max7219_REG_digit6      0x07
#define max7219_REG_digit7      0x08
#define max7219_REG_decodeMode  0x09
#define max7219_REG_intensity   0x0a
#define max7219_REG_scanLimit   0x0b
#define max7219_REG_shutdown    0x0c
#define max7219_REG_displayTest 0x0f


class LEDmat8{
public:
    LEDmat8( int DIN, int LOAD, int CLOCK );
    void Init( );
    void DrawLED( byte *LED_matrix );
    //~LEDmat8( );
    void SPI_SendByte( byte data );
    void MAX7219_1Unit( byte reg_addr, byte reg_data );

private:
    int DIN_pin;
    int LOAD_pin;
    int CLOCK_pin;
};
```

3. 在「LEDmat8」資料夾內新增一文字檔案，改檔名為「LEDmat8.cpp」（注意副檔名也要一起改），然後在檔案內容填入以下程式碼並存檔：

```
#include <LEDmat8.h>

LEDmat8::LEDmat8( int DIN, int LOAD, int CLOCK )
{
        DIN_pin = DIN;

LOAD_pin = LOAD;
  CLOCK_pin = CLOCK;
}

void LEDmat8::Init( )
{
  pinMode( DIN_pin, OUTPUT );
  pinMode( CLOCK_pin, OUTPUT );
  pinMode( LOAD_pin, OUTPUT );

  digitalWrite( CLOCK_pin, HIGH );

  // 初始化 MAX7219 的暫存器
  MAX7219_1Unit( max7219_REG_scanLimit, 0x07 ); // 設定為掃描所有行
  MAX7219_1Unit( max7219_REG_decodeMode, 0x00 ); // 不使用解碼模式
  MAX7219_1Unit( max7219_REG_shutdown, 0x01 ); // 設定為不在關閉模式
  MAX7219_1Unit( max7219_REG_displayTest, 0x00 );  // 設定為不在測試模式

  for(int i = 1; i <= 8; i++) { // 先把所有 LED 矩陣變暗
    MAX7219_1Unit( i, 0 );
  }

  MAX7219_1Unit( max7219_REG_intensity, 0x0f ); // 設定亮度範圍，0x00 ~ 0x0f

}

void LEDmat8::DrawLED( byte *LED_matrix ) // 繪製整個畫面
```

```
    byte i = 8;
    byte mask;

    while( i > 0 )
    {
      mask = (0x01 << (i - 1)); // 製造位元遮罩，從最左邊位開始
      digitalWrite( CLOCK_pin, LOW ); // 頻率同步線 = LOW
      if ( data & mask ) { // 判斷位元遮罩對應的位是 0 或 1
        digitalWrite( DIN_pin, HIGH ); // 如果對應位為 1，DIN 送出
HIGH
      }
      else{
        digitalWrite( DIN_pin, LOW ); // 如果對應位為 0，DIN 送出
LOW
      }
      digitalWrite( CLOCK_pin, HIGH ); // 頻率同步線 = HIGH
      i = i - 1; // 移到下一個位
    }
  }

  void LEDmat8::MAX7219_1Unit ( byte reg_addr, byte reg_data ) //
控制單一個 MAX7219 模組
  {
    digitalWrite( LOAD_pin, LOW ); // 傳送前 LOAD 腳要先變成 LOW
    SPI_SendByte( reg_addr ); // 先送出要設定的暫存器位址
    SPI_SendByte( reg_data ); // 接著送出資料
    digitalWrite( LOAD_pin, HIGH ); // 傳送完 LOAD 腳要變成 HIGH
  }
```

接著請打開 86Duino Coding IDE，輸入以下程式碼：

```
  #include <LEDmat8.h>
  #include <Keypad.h>

  const byte Rows = 4; // 行數
  const byte Cols = 4; // 列數
  // 按鍵對應符號
  char keys[Rows][Cols] =
  {
    {'1','2','3','A'},
    {'4','5','6','B'},
        {'7','8','9','C'},
```

```
      {'*','0','#','D'}
    };
    // 定義針腳編號
    byte row_pins[Rows] = {9, 8, 7, 6}; // 列 0~3
    byte col_pins[Cols] = {5, 4, 3, 2}; // 行 0~3

    // Keypad lib 物件
    Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins,
col_pins, Rows, Cols );

    // LED 模組控制腳位定義
    int DIN_pin = 10;
    int LOAD_pin = 11;
    int CLOCK_pin = 12;

    // 8X8 LED 矩陣 物件
    LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin, CLOCK_pin );

    byte LED_Data_8X8[8] = { // 圖樣資料矩陣
      B00000000,// 左->右 = 第 1 行由下而上
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000 // 左->右 = 第 8 行由下而上
    };

    void ClearLED_Data( ) // 清空 LED 畫面資料
    {
      for(int i = 0; i < 8; i++)
      {
        LED_Data_8X8[i] = B00000000;
      }
    }

    void setup ( ) {
      LedMatrix.Init();

          delay(1000);
  }
```

```
void loop ( ) {
  ClearLED_Data( ); //  更新 LED 矩陣資料前，先清空資料
  //  檢查鍵盤狀態
  keypad4X4.getKeys( ); //  更新鍵盤狀態

    //  逐一檢查  keypad4X4  物件的按鍵列表
    for( int i = 0; i < LIST_MAX; i++ )
    {
      //  檢查按鍵狀態是否為按鍵被按下
      if( keypad4X4.key[i].kstate == PRESSED )// HOLD
      {
        //  針對不同按鍵符號更新繪製資料，'D'對應左上角，'1'對應右下角
        switch( keypad4X4.key[i].kchar )
        {
          case '1': // LED r3 c3
            LED_Data_8X8[7] |= B11000000;
            LED_Data_8X8[6] |= B11000000;
            break;

          case '2': // LED r3 c2
            LED_Data_8X8[5] |= B11000000;
            LED_Data_8X8[4] |= B11000000;
            break;

          case '3': // LED r3 c1
            LED_Data_8X8[3] |= B11000000;
            LED_Data_8X8[2] |= B11000000;
            break;

          case 'A': // LED r3 c0
            LED_Data_8X8[1] |= B11000000;
            LED_Data_8X8[0] |= B11000000;
            break;

          case '4': // LED r2 c3
            LED_Data_8X8[7] |= B00110000;
            LED_Data_8X8[6] |= B00110000;
                  break;
```

```
        case '5': // LED r2 c2
                    LED_Data_8X8[5] |= B00110000;
                    LED_Data_8X8[4] |= B00110000;
                    break;

                  case '6':// LED r2 c1
                    LED_Data_8X8[3] |= B00110000;
                    LED_Data_8X8[2] |= B00110000;
                    break;

                  case 'B': // LED r2 c0
                    LED_Data_8X8[1] |= B00110000;
                    LED_Data_8X8[0] |= B00110000;
                    break;

                  case '7': // LED r1 c3
                    LED_Data_8X8[7] |= B00001100;
                    LED_Data_8X8[6] |= B00001100;
                    break;

                  case '8': // LED r1 c2
                    LED_Data_8X8[5] |= B00001100;
                    LED_Data_8X8[4] |= B00001100;
                    break;

                  case '9': // LED r1 c1
                    LED_Data_8X8[3] |= B00001100;
                    LED_Data_8X8[2] |= B00001100;
                    break;

                  case 'C': // LED r1 c0
                    LED_Data_8X8[1] |= B00001100;
                    LED_Data_8X8[0] |= B00001100;
                    break;

                  case '*': // LED r0 c3
                    LED_Data_8X8[7] |= B00000011;
                    LED_Data_8X8[6] |= B00000011;
                        break;
```

```
        case '0': // LED r0 c2
                    LED_Data_8X8[5] |= B00000011;
                    LED_Data_8X8[4] |= B00000011;
                    break;

                  case '#': // LED r0 c1
                    LED_Data_8X8[3] |= B00000011;
                    LED_Data_8X8[2] |= B00000011;
                    break;

                  case 'D': // LED r0 c0
                    LED_Data_8X8[1] |= B00000011;
                    LED_Data_8X8[0] |= B00000011;
                    break;

                  default:
                    break;
                }
              }
            } // end for

          // 繪製 LED 圖案
          LedMatrix.DrawLED( LED_Data_8X8 );
          delay( 50 );
            }
```

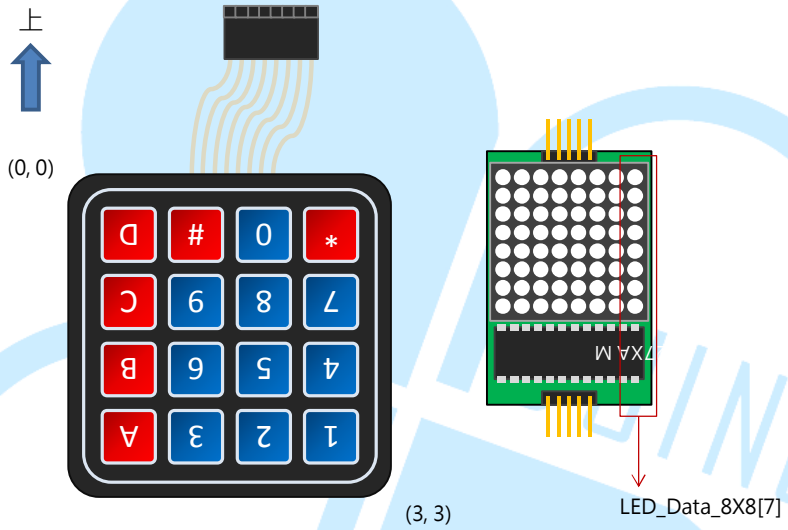
編譯並上傳程式後，讀者可按下矩陣鍵盤上的按鍵，LED 矩陣便會亮起對應位置的燈號。

這個程式內結合了Keypad函式庫與之前提過的8x8 LED矩陣顯示的功能，只是將之前的 LED 矩陣程式碼包裝成為 LEDmat8 函式庫而已。程式前方的變數宣告與範例二一樣，但多了幾個 LED 矩陣用的變數，如 byte 矩陣「LED_Data_8X8[8]」，當作 LED 顯示圖樣的資料。「ClearLED_Data( )」函式用在清空 LED_Data_8X8 的資料。「LEDmat8 LedMatrix = LEDmat8(DIN_pin, LOAD_pin, CLOCK_pin);」為 LEDmat8 class 的物件，所有 LED 矩陣控制的相關功能皆包含在內。

setup( )階段呼叫「LedMatrix.Init( )」，初始化 LED 矩陣控制的功能。loop( )迴圈每次都呼叫 ClearLED_Data( )，在更新 LED 矩陣資料前，先清空其中的資料。接著一樣須使用「keypad4X4.getKeys( );」更新鍵盤的狀態，再搭配「for(int

i = 0; i < LIST_MAX; i++)」、「keypad4X4.key[i].kstate」取得每個按鍵的狀態。

而 switch case 內則針對被按下的按鍵，改變 LED_Data_8X8 的內容。例如按下位於最右下角的按鍵「1」，則「LED_Data_8X8[7] |= B11000000;」、「LED_Data_8X8[6] |= B11000000;」將會讓 LED_Data_8X8 的右下角四個點變亮。這裡需要注意的是，按鍵對應 LED 位置是依據接線時的模組擺放方向而言，若讀者的矩陣鍵盤、LED 矩陣擺放相對方向不同，須修改程式碼以符合兩者的位置定義。本章節擺放的方向定義如下圖：



修改好 LED 的顯示內容資料後，最後使用「LedMatrix.DrawLED(LED_Data_8X8);」來繪製 LED 圖案。

讀者也可以試試把「if(keypad4X4.key[i].kstate == PRESSED)」這邊的條件改為其他的按鍵狀態，觀察 LED 矩陣發亮的時間點變化喔。

五、 第四個程式 – 使用 Keypad 函式庫+8X8LED 矩陣模組，打地鼠遊戲

最後這個範例程式以範例三做修改，來實作一個打地鼠遊戲的功能。電路接線不用變動，讀者請打開 86Duino Coding IDE，輸入以下程式碼：

```
#include <LEDmat8.h>
#include <Keypad.h>

const byte Rows = 4; //  行數
const byte Cols = 4; //  列數
//  按鍵對應符號
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
//  定義針腳編號
byte row_pins[Rows] = {9, 8, 7, 6}; //  列  0~3
byte col_pins[Cols] = {5, 4, 3, 2}; //  行  0~3

// Keypad lib  物件
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins, col_pins, Rows, Cols );


// LED  模組控制腳位定義
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;

// 8X8 LED 矩陣  物件
LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin, CLOCK_pin );

//  遊戲資料
int score = 0; //  遊戲計分
long gameTime = 0; //  遊戲進行時間
```

```
    boolean runGame = false; // 是否在遊戲執行狀態
    int loopCount = 0; // 決定隨機地鼠地圖在幾次迴圈後要更新
    #define DELAY_TIME        50 // loop 間隔時間
    #define  LOOPCOUNT_MAX   30  // 決定隨機地鼠地圖在幾次迴圈後
要重新產生，實際間隔時間(ms) = DELAY_TIME * LOOPCOUNT_MAX
    #define GAME_TIME          30 // 遊戲進行時間長度
    #define MOLE_NUM_MAX       6 // 一次出現的最大地鼠數量

    byte LED_Data_8X8[8] = { // 圖樣資料矩陣
      B00000000, // 左->右 = 第 1 行由下而上
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000   // 左->右 = 第 8 行由下而上
    };

    boolean Mole_Data[4][4] = { // 地鼠地圖 [列][行]
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0}
    };

    boolean Key_Data[4][4] = { // 按鍵地圖 [列][行]
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0}
    };

    void ClearLED_Data( ) // 清空 LED 畫面資料
    {
      for( int i = 0; i < 8; i++ )
      {
        LED_Data_8X8[i] = B00000000;
      }
        }
```

```
void ClearMoleData( ) { // 清空地鼠地圖資料
  for(int i = 0; i < 4; i++) {
    for(int j = 0; j < 4; j++) {
      Mole_Data[i][j] = false;
    }
  }
}

void ClearKeyData( ) { // 清空按鍵地圖資料
  for( int i = 0; i < 4; i++ ) {
    for( int j = 0; j < 4; j++ ) {
      Key_Data[i][j] = false;
    }
  }
}

void GameStart ( ) { // 開始遊戲，初始化遊戲資料
  ClearMoleData( ); // 清空地鼠地圖資料
  ClearKeyData( ); // 清空按鍵地圖資料
  runGame = true; // 設定執行遊戲的狀態變數
  score = 0; // 遊戲計分歸零
  gameTime = millis( ); // 遊戲進行時間重置
}

void GameEnd ( ) {// 停止遊戲，印出訊息
  Serial.println("-----------------------------------------");
  Serial.println("Game end!");
  Serial.print(" Total Score : ");Serial.println(score);
  Serial.println("     - Press 'S' or 'R' to play again.");
  Serial.println("-----------------------------------------");

  // 遊戲結束符號
  LED_Data_8X8[0] = B01111110;
  LED_Data_8X8[1] = B10000001;
  LED_Data_8X8[2] = B10010101;
  LED_Data_8X8[3] = B10100001;
  LED_Data_8X8[4] = B10100001;
  LED_Data_8X8[5] = B10010101;
  LED_Data_8X8[6] = B10000001;
      LED_Data_8X8[7] = B01111110;
```

```
    runGame = false;
  }

  void setup ( ) {

    LedMatrix.Init( );
    randomSeed( analogRead(0) );// 初始化隨機數字產生器

    Serial.begin( 115200 );

    delay( 4000 );

    Serial.println("-----------------------------------------------");
    Serial.print("    You    Have    ");    Serial.print(GAME_TIME);
Serial.println(" Seconds To Play Each Game.");
    Serial.println("    - Press 'S' To Start Game.");
    Serial.println("    - Press 'R' To Reset Game.");
    Serial.println("    - Press 'E' To End Game.");
    Serial.println("-----------------------------------------------");

  }

  void loop ( ) {
    loopCount++;
    if(loopCount>LOOPCOUNT_MAX){
      loopCount = 0;
    }

    // 檢查 COM PORT 傳入的訊息
    if(Serial.available( )){
      char ch = Serial.read( );
      if( ch == 's' || ch == 'S' ) { // 如果輸入 S 則開始遊戲
        Serial.println("-----------------------------------------------");
        Serial.println("Game is started!");
        Serial.println("-----------------------------------------------");
        GameStart( );
      }
      else if( ch == 'r' || ch == 'R' ) { // 如果輸入 R 則重置遊戲
        Serial.println("-----------------------------------------------");
            Serial.println("Game is reset!");
```

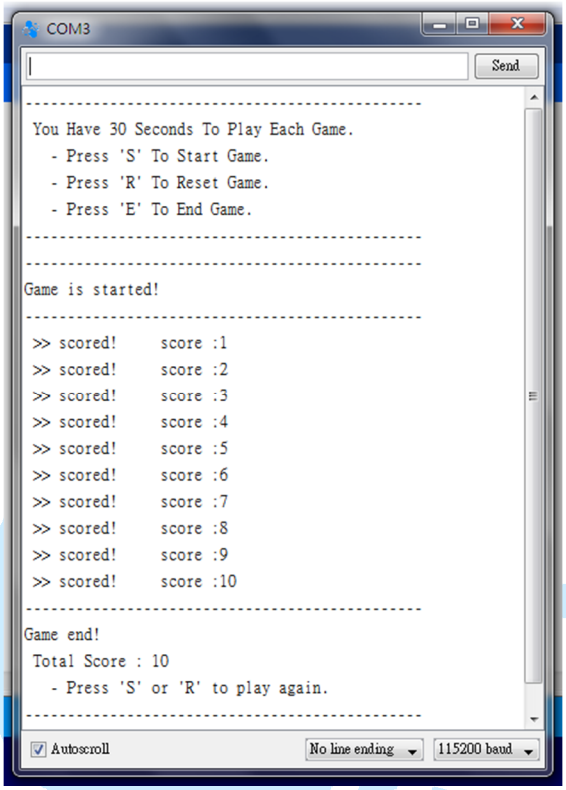
```
      Serial.println("--------------------------------------------");
            GameStart( );
          }
          else if( ch == 'e' || ch == 'E' ) { // 如果輸入 E 則停止遊戲
            GameEnd( );
          }
        }

        // 檢查遊戲進行時間
        if( runGame ){
          long time = millis( ) - gameTime; // 遊戲已進行時間
          if( time < GAME_TIME*1000 ) { // 遊戲還在進行中
            // 隨機位置、時間產生地鼠地圖
            if( loopCount == LOOPCOUNT_MAX ) { // 當 loopCount 到達最大值才更新地鼠地圖
              ClearMoleData( );
              long mole_num = random( 0, MOLE_NUM_MAX+1 ); // 一次出現的最大地鼠數量
              for( int i = 0; i < mole_num; i++ ) {
                long i_num = random(0,4); // 產生 [0,4) 之間的隨機數字
                long j_num = random(0,4); // 產生 [0,4) 之間的隨機數字
                //Serial.print(i_num);                Serial.print(",");Serial.println(j_num);
                Mole_Data[i_num][j_num] = true;
              }
            }

            // 讀取按鍵狀態，比對按鍵地圖與地鼠地圖，計算分數
            ClearLED_Data( ); // 更新 LED 矩陣資料前，先清空資料
            ClearKeyData( ); // 更新按鍵地圖資料前，先清空資料
            keypad4X4.getKeys( ); // 更新鍵盤狀態

            for( int i = 0; i < LIST_MAX; i++ ) // 逐一檢查 keypad4X4 物件的按鍵列表
            {
              // 檢查按鍵狀態是否為按鍵被按下
              if( keypad4X4.key[i].kstate == PRESSED )
              {
                // 針對不同按鍵符號更新繪製資料，'D'對應左上角，'1'對應右下角
                switch( keypad4X4.key[i].kchar )
                {
                        case '1':// LED r3 c3
```

```
      Key_Data[3][3] = true;
      break;

  case '2':// LED r3 c2
      Key_Data[3][2] = true;
      break;

  case '3':// LED r3 c1
      Key_Data[3][1] = true;
      break;

  case 'A':// LED r3 c0
      Key_Data[3][0] = true;
      break;

  case '4':// LED r2 c3
      Key_Data[2][3] = true;
      break;

  case '5':// LED r2 c2
      Key_Data[2][2] = true;
      break;

  case '6':// LED r2 c1
      Key_Data[2][1] = true;
      break;

  case 'B':// LED r2 c0
      Key_Data[2][0] = true;
      break;

  case '7':// LED r1 c3
      Key_Data[1][3] = true;
      break;

  case '8':// LED r1 c2
      Key_Data[1][2] = true;
      break;

        case '9':// LED r1 c1
```

```
                    Key_Data[1][1] = true;
                    break;

                case 'C':// LED r1 c0
                    Key_Data[1][0] = true;
                    break;

                case '*':// LED r0 c3
                    Key_Data[0][3] = true;
                    break;

                case '0':// LED r0 c2
                    Key_Data[0][2] = true;
                    break;

                case '#':// LED r0 c1
                    Key_Data[0][1] = true;
                    break;

                case 'D':// LED r0 c0
                    Key_Data[0][0] = true;
                    break;

                default:
                    break;
            }
        }
      }// end for
      // 檢查按鍵地圖與地鼠地圖
      for( int i = 0; i < 4; i++ ) { // 列
        for( int j = 0; j < 4; j++ ) { // 行
          // 當按鍵按下的地方符合地鼠出現位置，則消除地鼠，分數
+1
          if( Mole_Data[i][j] == true && Key_Data[i][j] == true ) {
            Mole_Data[i][j]= false;
            score++;
            Serial.print("    >>    scored!                score    :");
Serial.println(score);
          }
        }
          }
```

```
        // 依據消除後的地鼠地圖繪製畫面
            // 將 4x4 的地圖資料擴充到 8x8 LED 矩陣所需資料
          for( int i = 0; i < 4; i++ ) { // 列
            for( int j = 0; j < 4; j++ ) { // 行
              if( Mole_Data[i][j] == true ) {
                byte data = B00000011;
                data = data << (i*2);
                LED_Data_8X8[j*2] |= data;
                LED_Data_8X8[j*2+1] |= data;
              }// end if
            }
          }
          LedMatrix.DrawLED( LED_Data_8X8 ); // 繪製 LED 圖案
        }
        else{ // 時間到，結束遊戲
          GameEnd( );
          LedMatrix.DrawLED( LED_Data_8X8 );// 繪製 LED 圖案
        }
      }

      delay( DELAY_TIME );
        }
```

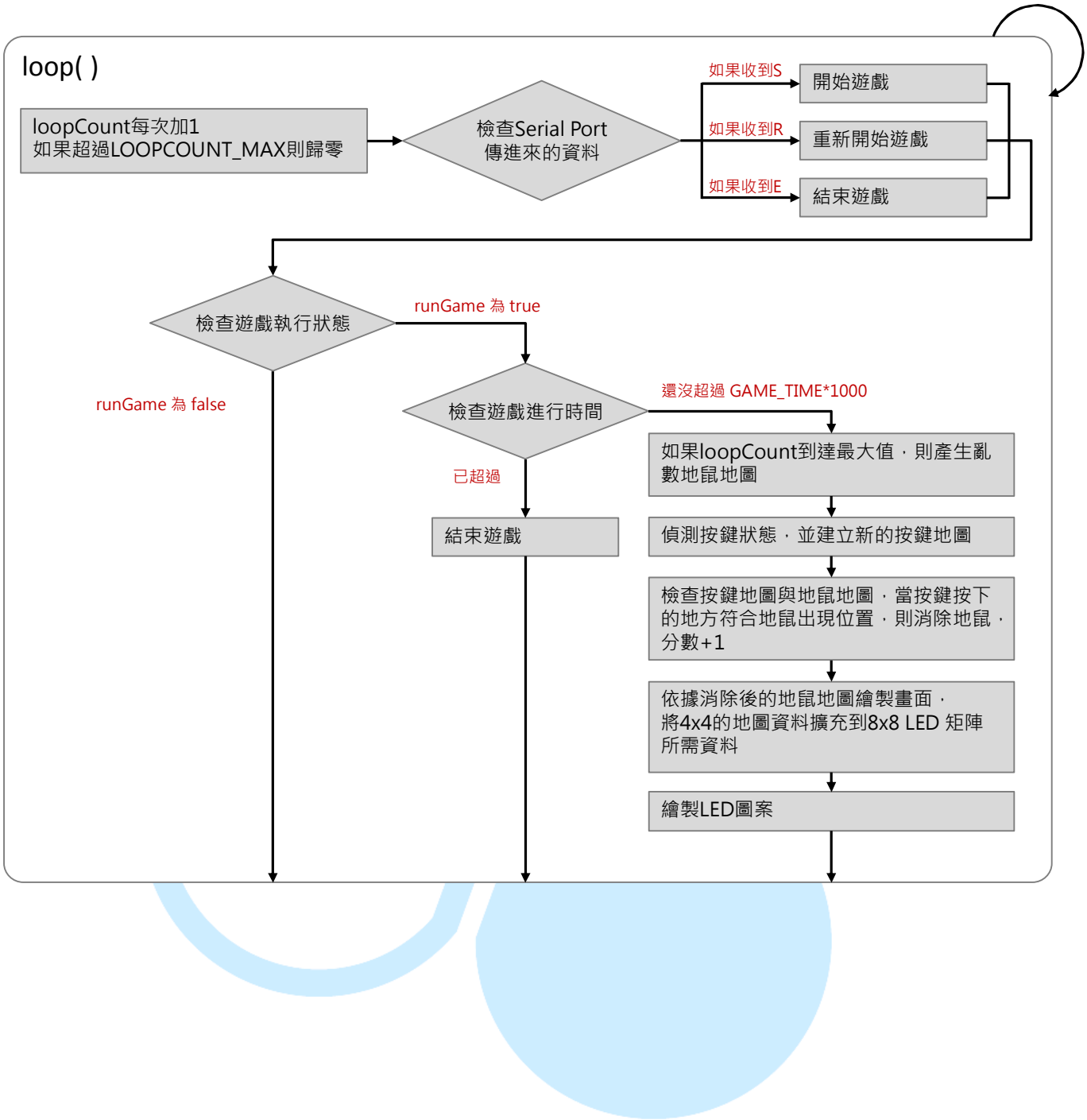
編譯並上傳程式後，請打開 Serial Monitor，上面會顯示如何開始進行遊戲的訊息，讀者可用 Serial Monitor 送出字母「S」或「R」開始遊戲。遊戲時 LED 矩陣會隨機亮起一些亮點區塊（代表有地鼠的地方），需按下鍵盤對應位置的按鍵才能消除此處的地鼠，並增加得分。每回合遊戲為 30 秒，當時間到就會停止遊戲，顯示得分，LED 矩陣出現笑臉圖樣。Serial Monitor 收到的訊息如下圖：



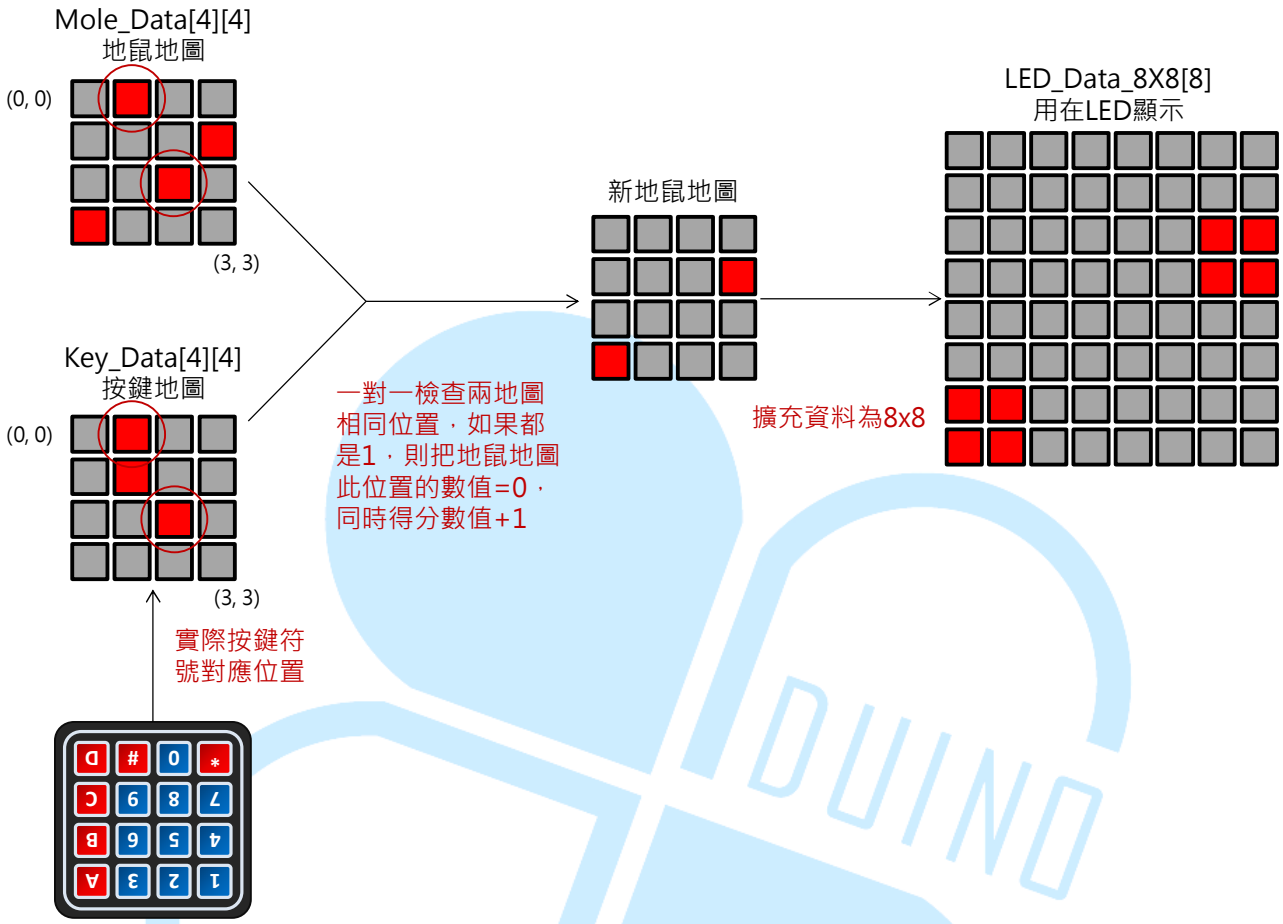
此範例程式增加了一些遊戲執行時的變數，例如「score」為遊戲計分、「gameTime」為遊戲進行時間、「runGame」用在判斷是否在遊戲執行狀態、「loopCount」決定隨機地鼠地圖在幾次 loop 迴圈後要更新等等。另外，「#define DELAY_TIME」設定loop間隔時間、「#define LOOPCOUNT_MAX」決定隨機地鼠地圖在幾次迴圈後要重新產生、「#define GAME_TIME」決定遊戲進行時間長度、「#define MOLE_NUM_MAX」決定一次出現的最大地鼠數量等等。函式則增加了「ClearMoleData( )」、「ClearKeyData( )」、「GameStart ( )」、「GameEnd ( )」等等，處理遊戲的流程。

setup( )階段與前面範例類似，比較需要注意的是，由於此程式需要產生隨機數，因此使用了「randomSeed(analogRead(0));」語法，當作隨機數產生器的初始化，參數欄內的「analogRead(0)」目的是使用一個無連接的腳位，確保隨機數產生出來夠亂。

loop( )內的流程較複雜，讀者請參考下面概略流程圖：


loop( )
loopCount每次加1
如果超過LOOPCOUNT_MAX則歸零
檢查Serial Port
傳進來的資料
如果收到S
開始遊戲
如果收到R
重新開始遊戲
如果收到E
結束遊戲
檢查遊戲執行狀態
runGame 為 true
runGame 為 false
檢查遊戲進行時間
還沒超過 GAME_TIME*1000
已超過
結束遊戲
如果loopCount到達最大值，則產生亂數地鼠地圖
偵測按鍵狀態，並建立新的按鍵地圖
檢查按鍵地圖與地鼠地圖，當按鍵按下的地方符合地鼠出現位置，則消除地鼠，分數+1
依據消除後的地鼠地圖繪製畫面，將4x4的地圖資料擴充到8x8 LED 矩陣所需資料
繪製LED圖案

檢查按鍵地圖與地鼠地圖的方式如下圖：



產生隨機地鼠地圖的方法上，這裡採用的是先隨機產生地鼠數量 N，再隨機產生 N 組範圍在 0~3 的行數、列數，並把這些位置的數值變成 1（表示有出現地鼠），當然位置有可能會重疊，所以最多只會一次出現 N 隻地鼠而已。語法「long num = random(min, max)」會讓 num 的數值為介於「大於等於 min~小於 max」之間的隨機數字。

經由上述的流程以及矩陣鍵盤、LED 矩陣模組的配合，便可以完成一個簡單的打地鼠遊戲，而讀者也可以在這個概念上加上其他 I/O 裝置功能，例如用按鈕啟動遊戲、RC 伺服馬達驅動地鼠升降、用 7 段顯示器顯示遊戲進行狀態及得分數、打中地鼠時發出音效等等，有很大的發揮空間呢。

遊戲的難度則可以藉由調整 LOOPCOUNT_MAX（調整地鼠地圖的變動速度）、MOLE_NUM_MAX（調整地鼠一次出現的最大數量）的數值、gameTime（調整每回合時間長度），來達到不同的難度等級，這些就有待讀者自行嘗試改裝囉。