

## EduCake 使用矩阵键盘



### 一、 矩阵键盘原理介绍

关于使用者与 86Duino EduCake 的互动方式，除了前面章节提到的传感器应用、数字模拟转换器读取、Serial Port 传输数据等方式外，还有一种常见的输入方式，就是使用「矩阵键盘」这种装置，通常用在控制器需要让用户输入命令、数据等等用途上。其实矩阵键盘的原理，就跟先前章节介绍过的「微动开关按键」结合「digitalRead( ) 函式」类似，只是从单一按键变成了按键矩阵。不过也因为按键数量变多，控制程序撰写方式也变得不同了，本章节就从矩阵键盘的原理介绍开始，并使用 86Duino EduCake 来实作一些有趣的功能吧。

一般市面上可以买到的矩阵键盘种类多样化，有的是薄膜接触式导通，有的是机械式微动开关等，不过使用原理大同小异。本章节讨论的小型矩阵键盘而言，大小常见 3X4 或 4X4 等，上面可能标示 0~9、英文符号、其他符号如「\*」「#」等等，例如计

算器的键盘就是个很好的例子。读者如果想要制作属于自己的特殊键盘布局也可以，看完本章节后就可以自己动手做啰。

当项目里面使用的按键数量只需要手指就数得出来，实作上其实只要一个按键对应一个 digital 脚位就可以，但是当使用了 3X4 或更大的键盘时，需要的针脚数量一下子就增加很多了，当然也会占据控制器宝贵的针脚空间。因此，矩阵键盘设计上会使用「扫描」的概念，逐行逐列作按键状态的扫描，便可以节省针脚数量，就跟前面提到的 8X8 LED 矩阵扫描显示是一样的道理。以一个 4X4 矩阵键盘为例，电路接线如下面图 1 所示：

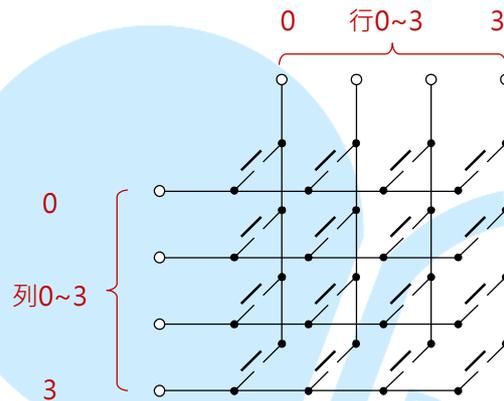
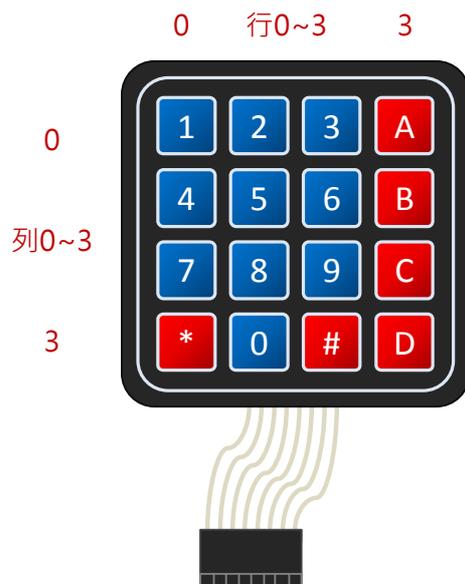
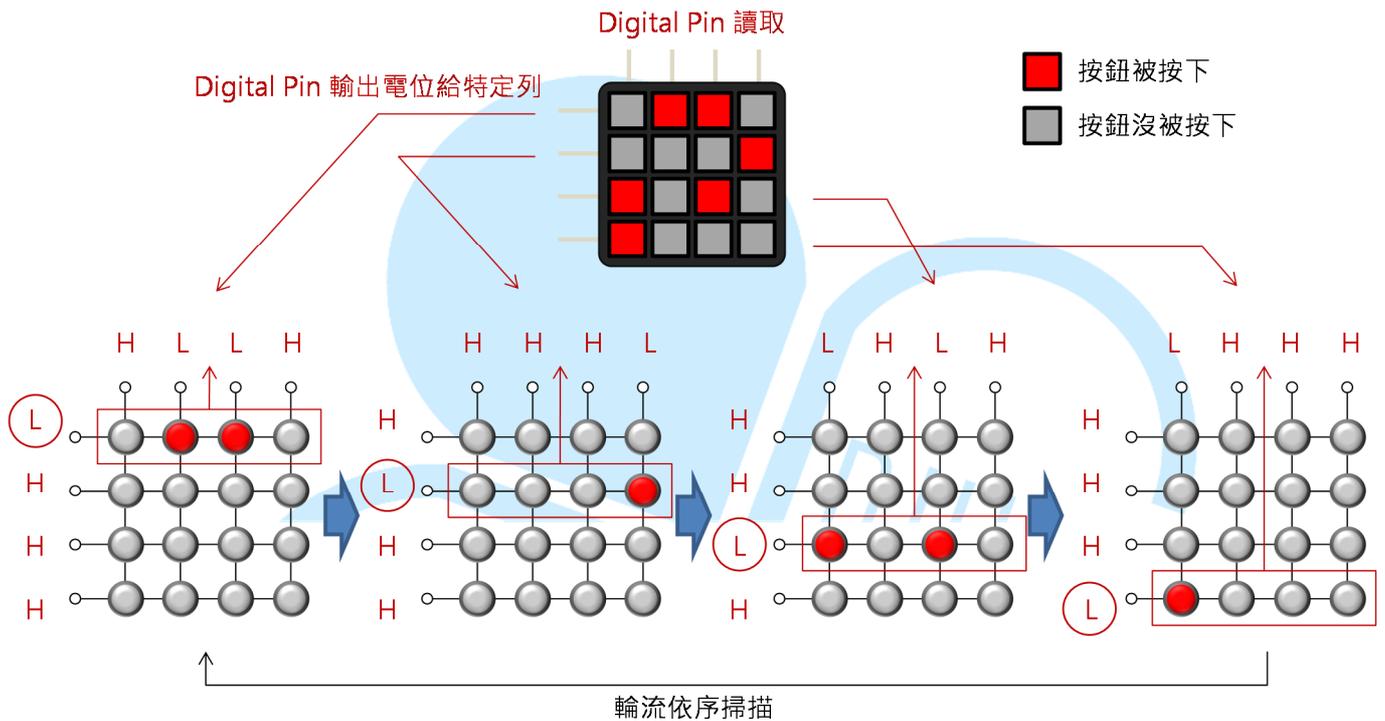


图 1. 4X4 矩阵键盘电路接线图

读者可以从电路图上看到，依照这样的连接方式，4X4 的矩阵键盘只需要占用控制器的 8 只针脚，而不是 16 只针脚，相对「一对一读取按键方法」可以节省一半针脚数量。对某一列的按键来说，按键一侧是全部相通的，不过这样的接线方式造成程序需要使用较复杂的流程来处理。扫描流程中，控制器程序会依序对某一列给予 HIGH 或 LOW 的电压值（依使用状况决定），然后读取这一列上每个行的按键电压，扫描流程如下图 2 所示。

图 2. 4X4 矩阵键盘扫描流程图

这边须注意，依序扫描时，一次只能有一列的电压是 HIGH 或 LOW，其他列必须给相反的电压，否则读取某行电压时会分辨不出是哪一列的按键被按下了。接下来的 86Duino EduCake 实作上，我们选用一般容易买到的 4X4 矩阵键盘模块，如下图 3 所示：



针脚定义，依序为：  
[列0 · 列1 · 列2 · 列3 · 行0 · 行1 · 行2 · 行3]

图 3. 4X4 矩阵键盘引脚定义

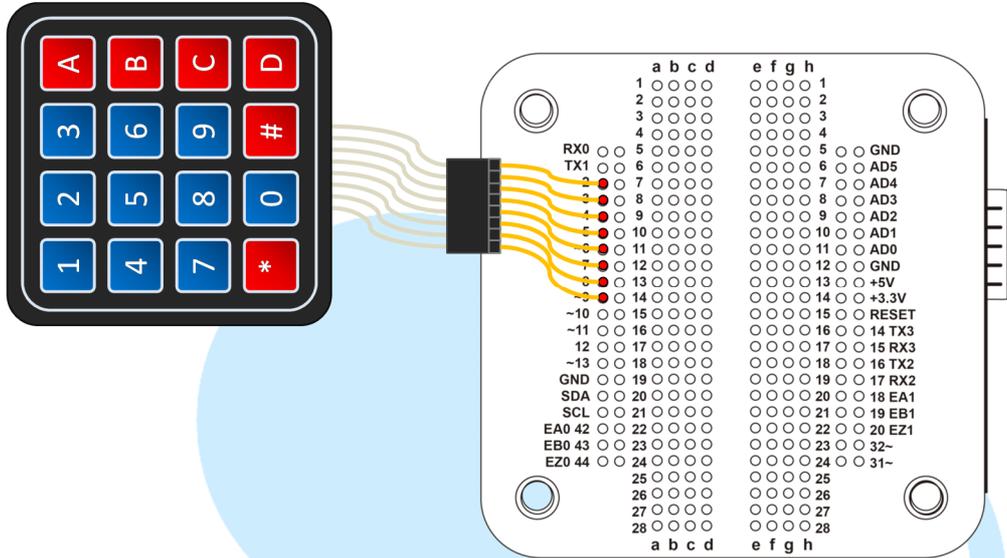
以这个矩阵键盘模块来说，按键按下是电路导通，各行列的接线已经被排列为 8 针脚插头方便安装，针脚定义由左到右依序是「列 0，列 1，列 2，列 3，行 0，行 1，行 2，行 3」。不过若读者实作时买的是其他键盘模块，最好先使用三用电表搭配手指按下特定按键，先测量好针脚的定义，这样接线与写程序时才能正常运作喔。

由于 86Duino EduCake 可用的脚位数量约有二十几个（AD 脚位可作为读取用途），因此本章节的实作方式，较适合用在这种中等按键数量的矩阵键盘上，如果读者希望自己动手做一个像计算机键盘那么多按键的装置，最好还是搭配专用的 IC 较佳。下面便让我们利用程序，实际练习矩阵键盘的原理，并使用 4X4 矩阵键盘模块做些实际应用的功能吧。



## 二、 第一个程序 – 键盘扫描原理练习

第一个范例程序先来练习如何使用 86Duino EduCake 实作上述矩阵键盘扫描原理，读者请先依下图接线：



列 0~3，行 0~3 依序接到数字脚位 9~2 即可，接着请打开 86Duino Coding IDE，输入以下程序代码：

```
const int Rows = 4; // 行数
const int Cols = 4; // 列数
// 按键对应符号
char keys[Rows][Cols] =
{
    {'1','2','3','A'},
    {'4','5','6','B'},
    {'7','8','9','C'},
    {'*','0','#','D'}
};
// 纪录按键上一次状态
bool keys_status_last[Rows][Cols] =
{
    {false,false,false,false},
    {false,false,false,false},
    {false,false,false,false},
    {false,false,false,false}
};
// 定义引脚编号
```

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

void setup()
{
    // 设定 Pin IO 模式
    // 用行的 pin 读取按键电压状态
    for(int col = 0; col < Cols; col++)
    {
        pinMode( col_pins[col], INPUT_PULLUP);
    }
    // 用列的 pin 当电压源
    for( int row = 0; row < Rows; row++) // 扫描行
    {
        pinMode( row_pins[row], OUTPUT );
        digitalWrite( row_pins[row], HIGH );
    }

    Serial.begin(115200);
}

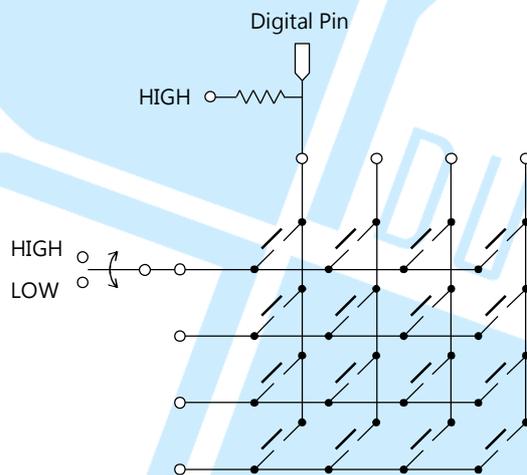
void loop()
{
    for( int row = 0; row < Rows; row++ ) // 扫描列
    {
        digitalWrite( row_pins[row], LOW ); // 此列电压变为 LOW
        for( int col = 0; col < Cols; col++ ) // 扫描行
        {
            // 读取这行的电压，如果按钮有按下导通，电压为 LOW
            boolean result = !digitalRead( col_pins[col] );

            // 这次按钮按下，上一次是按下，则按键为持续压按状态
            if( result == HIGH && keys_status_last[row][col] == true )
            {
                Serial.print("Button ");
                Serial.print(keys[row][col]);
                Serial.println(" hold");
            }
            // 这次按钮按下，上一次不是按下，则按键为刚被按下状态
```



程序一开始，先设定矩阵键盘的行数 `Cols` 与列数 `Rows`，`char` 矩阵 `keys[Rows][Cols]` 用在储存键盘对应字符符号，而 `bool` 矩阵 `keys_status_last[Rows][Cols]` 用在储存每个按键的上一次状态，按下为「true」，没按下为「false」。针脚编号矩阵则用在宣告键盘行列实际对应的针脚编号，以及方便后续的存取，例如使用语法 `row_pins[1]`，便可以取得列 1 的针脚编号。

`setup()` 阶段则设定了各个数字脚位的 I/O 模式，这里使用列 `pin` 当作电压来源，而行 `pin` 用在读取电压状态。另外执行了 `Serial Port` 的初始化。须注意数字输入的部分使用了 `INPUT_PULLUP` 模式，而扫描列时给予 `LOW` 的电压（其余列给 `HIGH`），因此当按键被按下时会读取到 `LOW`，没按下时都是 `HIGH`，这样可让按键电压状态的读取更为稳定。电路示意如下图：



`loop()` 循环里，使用了双层 `for` 循环，第一层为扫描列用，第二层用在扫描行。双层循环会对按键进行电压读取，达到前述的扫描原理。

由于按键按下时电压为 `LOW`，因此程序代码「`boolean result = !digitalRead( col_pins[col] );`」中有一个反向的动作，所以 `result` 为 `true` 时按键为按下，再将 `result` 的结果当作这次读取按键的状态。

这里需要注意的是，`keys_status_last` 矩阵用在记录各个按键的上一次状态，因此能够知道按键是刚被按下（上一次 `false`，这一次 `true`）、持续压着（上一次 `true`，这一次 `true`）、或是被放开（上一次 `true`，这一次 `false`），视用户需要的功能完整度来实现。这样便简单练习了键盘的扫描原理，如果读者使用了不同大小的键盘、不同的字符符号，只要从前面几个变量宣告做修改就可以啰。

### 三、 第二个程序 – 使用 Keypad 函式库

了解 86Duino EduCake 与扫描矩阵键盘的基本原理后，接着来做点小变化。这个范例电路不用更改，使用与范例一相同的接线。

依照键盘的机械结构不同，按下按钮与放开按钮时实际上会有「弹跳」的状况发生，接点在接触与未接触的状态间快速变化，如果没有处理的话，可能单击按钮，却发现电压其实起伏了好几下才稳定。去弹跳（也称为 **debounce**）的方法有电路式，也有程序的处理方式，不过上面第一个程序并没有处理去弹跳的问题，主要是因为 **loop** 间隔时间满长的（20ms），读取数字脚位的间隔已经超过了弹跳的变动时间，因此读取按钮电压时不大会变动。如果读者需要在高速读取按键状态的场合使用，就不太适合了。

还好，已经有现成的 **Keypad** 函式库可以使用，且内部已经处理了去弹跳、扫描侦测按键状态等的功能，这个范例，我们就使用这个函式库来做矩阵键盘读取的练习吧。

请读者先到网址：

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

下载 **Keypad** 的程序代码压缩文件后，将解压缩后的「**Keypad**」文件夹放到 86Duino IDE 所在文件夹路径「86Duino\_Coding\_版本号码\_WIN\libraries」内。再使用文本编辑器打开 **Keypad.h** 档案，然后找到「**#include "WProgram.h"**」修改成「**#include <Arduino.h>**」。然后对「**Keypad/utility** 文件夹」内的 **Key.h** 档案做相同的修改动作。完成后，请打开 86Duino Coding IDE，输入以下程序代码：

```
#include <Keypad.h>

const byte Rows = 4; // 行数
const byte Cols = 4; // 列数
// 按键对应符号
char keys[Rows][Cols] =
{
    {'1','2','3','A'},
    {'4','5','6','B'},
    {'7','8','9','C'},
    {'*','0','#','D'}
};
// 定义引脚编号
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++ ) // 逐一检查 keypad4X4 对象的按键列表
        {
            if( keypad4X4.key[i].stateChanged ) // 检查按键状态是否变动，如果有变动再取出内容
            {
                Serial.print("Button ");
                Serial.print(keypad4X4.key[i].kchar); // 目前被按下的按键字符符号
                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())

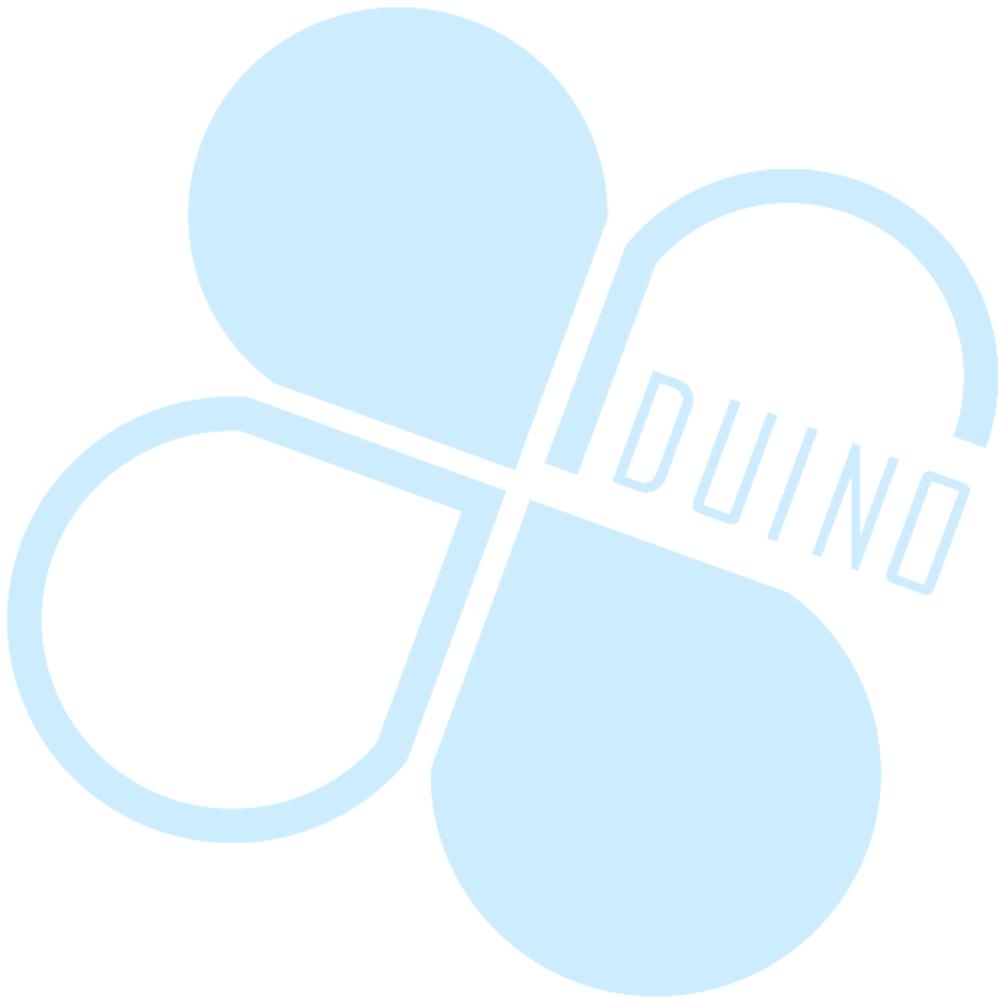
delay( 20);
```

编译并上传程序后，一样打开 Serial Monitor，并尝试按下键盘的按键，Serial Monitor 中将会显示被按下的按键对应符号与按键状态，程序的执行结果与范例一类似。

读者会注意到，这个程序代码一开始的行数、列数宣告、按键对应符号、脚位对应都跟范例一的程序差不多，但是开头多了「`#include <Keypad.h>`」，接着全局变量多了个 Keypad class 对象「`Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins, col_pins, Rows, Cols );`」，这便是 Keypad 函式库的方便之处，将所有矩阵键盘的处理程序、变量定义都包装到 Keypad 的 class 中，有效缩减.ino 档案内的程序行数，也方便在不同项目程序内重复使用相同的 Keypad 函式库。

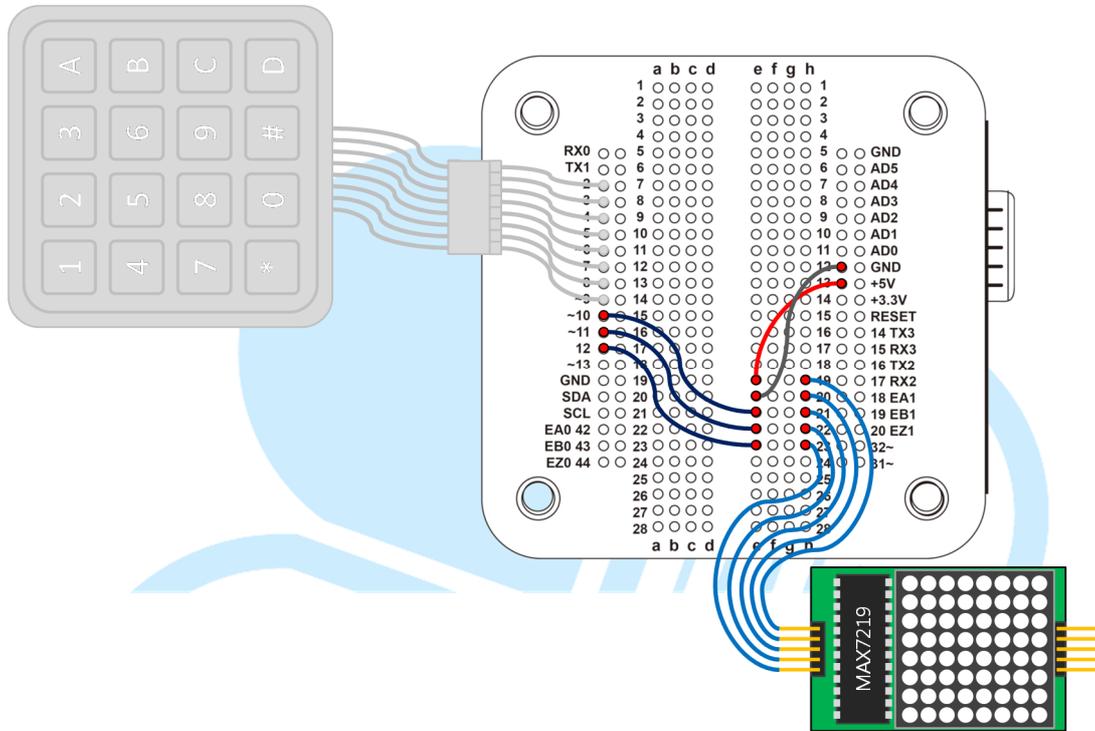
接着在 `setup()` 阶段，只做了 Serial Port 的初始化。`loop()` 循环内则使用「`keypad4X4.getKeys()`」语法做键盘状态的读取与更新，接着才能用「`for(int i = 0; i < LIST_MAX; i++)`」语法逐一检查 keypad4X4 对象的按键列表，判断各个按键的触发状态。Keypad 函式库提供：「`keypad4X4.key[i].stateChanged`」

用在检查按键状态是否改变、「`keypad4X4.key[i].kchar`」用在取得此按键对应的字符符号、「`keypad4X4.key[i].kstate`」用在取得此按键状态等。按键状态共有 **PRESSED**（被按下）、**HOLD**（持续压着）、**RELEASED**（放开）、**IDLE**（没被按着），`loop()`循环内便是用这几个函式与变量判断按键状态，然后印出讯息。使用了 **Keypad** 函式库之后，程序代码看起来是不是变整洁多了呢？



#### 四、 第三个程序 – 使用 Keypad 函式库+8X8LED 矩阵模块

接着这个范例程序继续使用 Keypad 函式库, 我们继续来加入其他较复杂的功能。此范例程序会用到之前已经介绍过的 MAX7219+8x8 LED 矩阵模块, 请读者依照下图加上 LED 矩阵的接线:



由于 MAX7219+8x8 LED 矩阵模块的功能之前已经讲过, 而且重复利用的机会很多, 因此这边把相关的程序代码包装成为 LEDmat8 函式库, 需要先做几个步骤:

1. 在 86Duino IDE 所在文件夹路径「86Duino\_Coding\_版本号\_WIN\libraries」内, 新增一「LEDmat8」文件夹。
2. 在「LEDmat8」文件夹内新增一文本文件, 改档名为「LEDmat8.h」(注意扩展名也要一起改), 然后在档案内容填入以下程序代码并存盘:

```
#ifndef LEDMAT8_H
#define LEDMAT8_H

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

    // MAX7219 缓存器定义
#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」文件夹内新增一文本文件，改档名为「LEDmat8.cpp」（注意扩展名也要一起改），然后在档案内容填入以下程序代码并存盘：

```
#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 的缓存器
    MAX7219_1Unit( max7219_REG_scanLimit, 0x07 ); // 设定为扫描
所有行
    MAX7219_1Unit( max7219_REG_decodeMode, 0x00 ); // 不使用
译码模式
    MAX7219_1Unit( max7219_REG_shutdown, 0x01 ); // 设定为不在
关闭模式
    MAX7219_1Unit( max7219_REG_displayTest, 0x00 ); // 设定为
不在测试模式

    for(int i = 1; i <= 8; i++) { // 先把所有 LED 矩阵变暗
        MAX7219_1Unit( i, 0 );
    }

    MAX7219_1Unit( max7219_REG_intensity, 0x0f ); // 设定亮度范
围, 0x00 ~ 0x0f

}

void LEDmat8::DrawLED( byte *LED_matrix ) // 绘制整个画面
```

```

    byte i = 8;
    byte mask;

    while( i > 0 )
    {
        mask = (0x01 << (i - 1)); // 制造位掩码，从最左边位开始
        digitalWrite( CLOCK_pin, LOW ); // 频率同步线 = LOW
        if ( data & mask ) { // 判断位掩码对应的位是 0 或 1
            digitalWrite( DIN_pin, HIGH ); // 如果对应位为 1，DIN 送出
HIGH
        }
        else{
            digitalWrite( DIN_pin, LOW ); // 如果对应位为 0，DIN 送出
LOW
        }
        digitalWrite( CLOCK_pin, HIGH ); // 频率同步线 = HIGH
        i = i - 1; // 移到下一个位
    }
}

void LEDmat8::MAX7219_1Unit ( byte reg_addr, byte reg_data ) //
控制单一个 MAX7219 模块
{
    digitalWrite( LOAD_pin, LOW ); // 传送前 LOAD 脚要先变成 LOW
    SPI_SendByte( reg_addr ); // 先送出要设定的缓存器地址
    SPI_SendByte( reg_data ); // 接着送出资料
    digitalWrite( LOAD_pin, HIGH ); // 传送完 LOAD 脚要变成 HIGH
}

```

接着请打开 86Duino Coding IDE，输入以下程序代码：

```

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

const byte Rows = 4; // 行数
const byte Cols = 4; // 列数
// 按键对应符号
char keys[Rows][Cols] =
{
    {'1','2','3','A'},
    {'4','5','6','B'},
    {'7','8','9','C'},
}

```

```
{ '*', '0', '#', 'D' }
};
// 定义引脚编号
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 模块控制脚位定义
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;

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

byte LED_Data_8X8[8] = { // 图样资料矩阵
  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 画面资料
{
  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 矩阵数据前，先清空数据
    // 检查键盘状态
    keypad4X4.getKeys(); // 更新键盘状态

    // 逐一检查 keypad4X4 对象的按键列表
    for( int i = 0; i < LIST_MAX; i++ )
    {
        // 检查按键状态是否为按键被按下
        if( keypad4X4.key[i].kstate == PRESSED )// HOLD
        {
            // 针对不同按键符号更新绘制数据，'D'对应左上角，'1'对应右下角
            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

// 绘制 LED 图案
LedMatrix.DrawLED( LED_Data_8X8 );
delay( 50 );
}
```

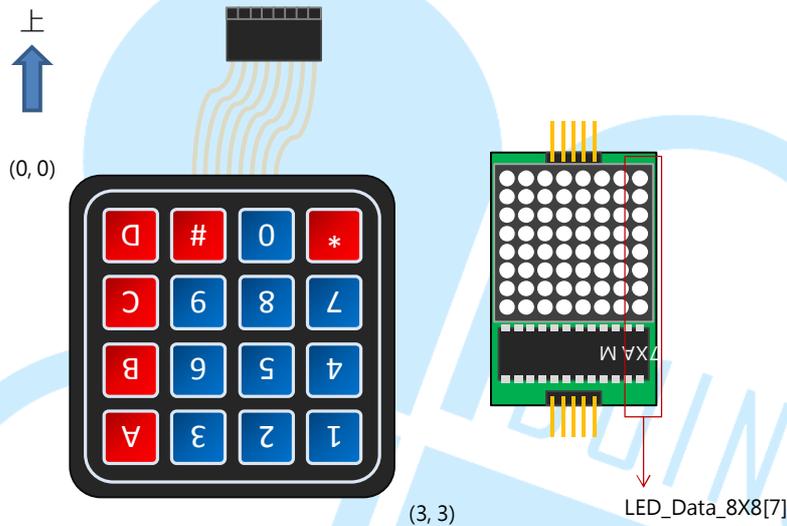
编译并上传程序后，读者可按下矩阵键盘上的按键，LED 矩阵便会亮起对应位置的灯号。

这个程序内结合了 Keypad 函式库与之前提过的 8x8 LED 矩阵显示的功能，只是将之前的 LED 矩阵程序代码包装成为 LEDmat8 函式库而已。程序前方的变量宣告与范例二一样，但多了几个 LED 矩阵用的变数，如 byte 矩阵「LED\_Data\_8X8[8]」，当作 LED 显示图样的数据。「ClearLED\_Data()」函式用在清空 LED\_Data\_8X8 的资料。「LEDmat8 LedMatrix = LEDmat8( DIN\_pin, LOAD\_pin, CLOCK\_pin );」为 LEDmat8 class 的对象，所有 LED 矩阵控制的相关功能皆包含在内。

setup()阶段呼叫「LedMatrix.Init()」，初始化 LED 矩阵控制的功能。loop() 循环每次都呼叫 ClearLED\_Data( )，在更新 LED 矩阵数据前，先清空其中的资料。接着一样须使用「keypad4X4.getKeys();」更新键盘的状态，再搭配「for( int

`i = 0; i < LIST_MAX; i++ )`」、「`keypad4X4.key[i].kstate`」取得每个按键的状态。

而 `switch case` 内则针对被按下的按键，改变 `LED_Data_8X8` 的内容。例如按下位于最右下角的按键「1」，则「`LED_Data_8X8[7] |= B11000000;`」、「`LED_Data_8X8[6] |= B11000000;`」将会让 `LED_Data_8X8` 的右下角四个点变亮。这里需要注意的是，按键对应 LED 位置是依据接线时的模块摆放方向而言，若读者的矩阵键盘、LED 矩阵摆放相对方向不同，须修改程序代码以符合两者的位置定义。本章节摆放的方向定义如下图：



修改好 LED 的显示内容数据后，最后使用「`LedMatrix.DrawLED( LED_Data_8X8 );`」来绘制 LED 图案。

读者也可以试试把「`if( keypad4X4.key[i].kstate == PRESSED )`」这边的条件改为其他的按键状态，观察 LED 矩阵发亮的时间点变化喔。

## 五、 第四个程序 – 使用 Keypad 函式库+8X8LED 矩阵模块,打地鼠游戏

最后这个范例程序以范例三做修改,来实作一个打地鼠游戏的功能。电路接线不用变动,读者请打开 86Duino Coding IDE,输入以下程序代码:

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

const byte Rows = 4; // 行数
const byte Cols = 4; // 列数
// 按键对应符号
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
// 定义引脚编号
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 模块控制脚位定义
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;

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

// 游戏资料
int score = 0; // 游戏计分
long gameTime = 0; // 游戏进行时间
```

```
boolean runGame = false; // 是否在游戏执行状态
int loopCount = 0; // 决定随机地图在几次循环后要更新
#define DELAY_TIME      50 // loop 间隔时间
#define LOOPCOUNT_MAX  30 // 决定随机地图在几次循环后
要重新产生，实际间隔时间(ms) = DELAY_TIME * LOOPCOUNT_MAX
#define GAME_TIME       30 // 游戏进行时间长度
#define MOLE_NUM_MAX    6 // 一次出现的最大老鼠数量

byte LED_Data_8X8[8] = { // 图样资料矩阵
    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] = { // 老鼠地图 [列][行]
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0}
};

boolean Key_Data[4][4] = { // 按键地图 [列][行]
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0}
};

void ClearLED_Data() // 清空 LED 画面资料
{
    for(int i = 0; i < 8; i++)
    {
        LED_Data_8X8[i] = B00000000;
    }
}
```

```
void ClearMoleData() { // 清空地图数据
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            Mole_Data[i][j] = false;
        }
    }
}

void ClearKeyData() { // 清空按键地图数据
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            Key_Data[i][j] = false;
        }
    }
}

void GameStart () { // 开始游戏，初始化游戏数据
    ClearMoleData(); // 清空地图数据
    ClearKeyData(); // 清空按键地图数据
    runGame = true; // 设定执行游戏的状态变量
    score = 0; // 游戏计分归零
    gameTime = millis(); // 游戏进行时间重置
}

void GameEnd () { // 停止游戏，印出讯息
    Serial.println("-----");
    Serial.println("Game end!");
    Serial.print(" Total Score : ");Serial.println(score);
    Serial.println(" - Press 'S' or 'R' to play again.");
    Serial.println("-----");

    // 游戏结束符号
    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) );// 初始化随机数字产生器

    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;
    }

    // 检查 COM PORT 传入的讯息
    if(Serial.available()){
        char ch = Serial.read();
        if( ch == 's' || ch == 'S' ) { // 如果输入 S 则开始游戏
            Serial.println("-----");
            Serial.println("Game is started!");
            Serial.println("-----");
            GameStart();
        }
        else if( ch == 'r' || ch == 'R' ) { // 如果输入 R 则重置游戏
            Serial.println("-----");
            Serial.println("Game is reset!");
        }
    }
}
```

```

Serial.println("-----");
    GameStart();
}
else if( ch == 'e' || ch == 'E' ) { // 如果输入 E 则停止游戏
    GameEnd();
}
}

// 检查游戏进行时间
if( runGame ){
    long time = millis() - gameTime; // 游戏已进行时间
    if( time < GAME_TIME*1000 ) { // 游戏还在进行中
        // 随机位置、时间产生地鼠地图
        if( loopCount == LOOPCOUNT_MAX ) { // 当 loopCount 到达最大值才更新地鼠地图
            ClearMoleData();
            long mole_num = random( 0, MOLE_NUM_MAX+1 ); // 一次出现的最大地鼠数量
            for( int i = 0; i < mole_num; i++ ) {
                long i_num = random(0,4); // 产生 [0,4) 之间的随机数字
                long j_num = random(0,4); // 产生 [0,4) 之间的随机数字
                //Serial.print(i_num);           Serial.print(",");
                Serial.println(j_num);
                Mole_Data[i_num][j_num] = true;
            }
        }

        // 读取按键状态，比对按键地图与地鼠地图，计算分数
        ClearLED_Data(); // 更新 LED 矩阵数据前，先清空数据
        ClearKeyData(); // 更新按键地图数据前，先清空数据
        keypad4X4.getKeys(); // 更新键盘状态

        for( int i = 0; i < LIST_MAX; i++ ) // 逐一检查 keypad4X4 对象的按键列表
        {
            // 检查按键状态是否为按键被按下
            if( keypad4X4.key[i].kstate == PRESSED )
            {
                // 针对不同按键符号更新绘制数据，'D'对应左上角，'1'对应
                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;

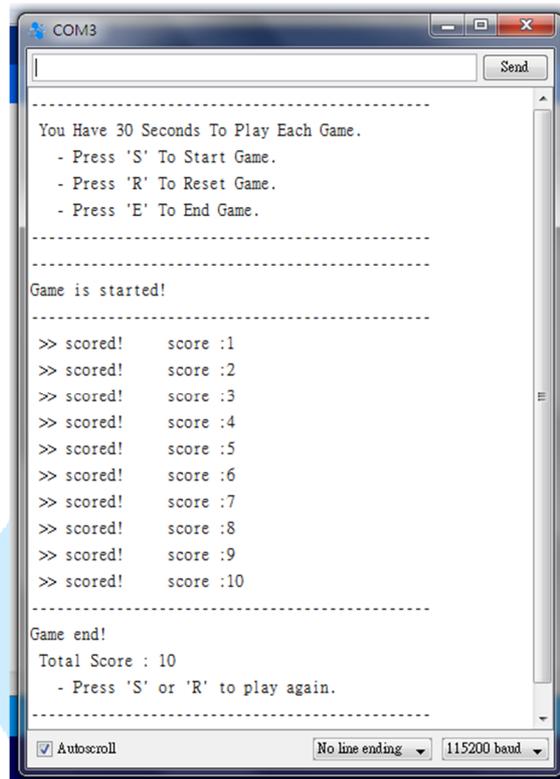
    default:
        break;
    }
}
} // end for
// 检查按键地图与地鼠地图
for( int i = 0; i < 4; i++ ) { // 列
    for( int j = 0; j < 4; j++ ) { // 行
        // 当按键按下的地方符合地鼠出现位置，则消除地鼠，分数
+1
        if( Mole_Data[i][j] == true && Key_Data[i][j] == true ) {
            Mole_Data[i][j]= false;
            score++;
            Serial.print("  >>  scored!           score  :");
Serial.println(score);
        }
    }
}
}
}

```

```
// 依据消除后的地图绘制画面
// 将 4x4 的地图数据扩充到 8x8 LED 矩阵所需数据
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 ); // 绘制 LED 图案
}
else{ // 时间到，结束游戏
  GameEnd();
  LedMatrix.DrawLED( LED_Data_8X8 ); // 绘制 LED 图案
}
}

delay( DELAY_TIME );
}
```

编译并上传程序后，请打开 Serial Monitor，上面会显示如何开始进行游戏的讯息，读者可用 Serial Monitor 送出字母「S」或「R」开始游戏。游戏时 LED 矩阵会随机亮起一些亮点区块（代表有地鼠的地方），需按下键盘对应位置的按键才能消除此处的地鼠，并增加得分。每回合游戏为 30 秒，当时间到就会停止游戏，显示得分，LED 矩阵出现笑脸图样。Serial Monitor 收到的讯息如下图：

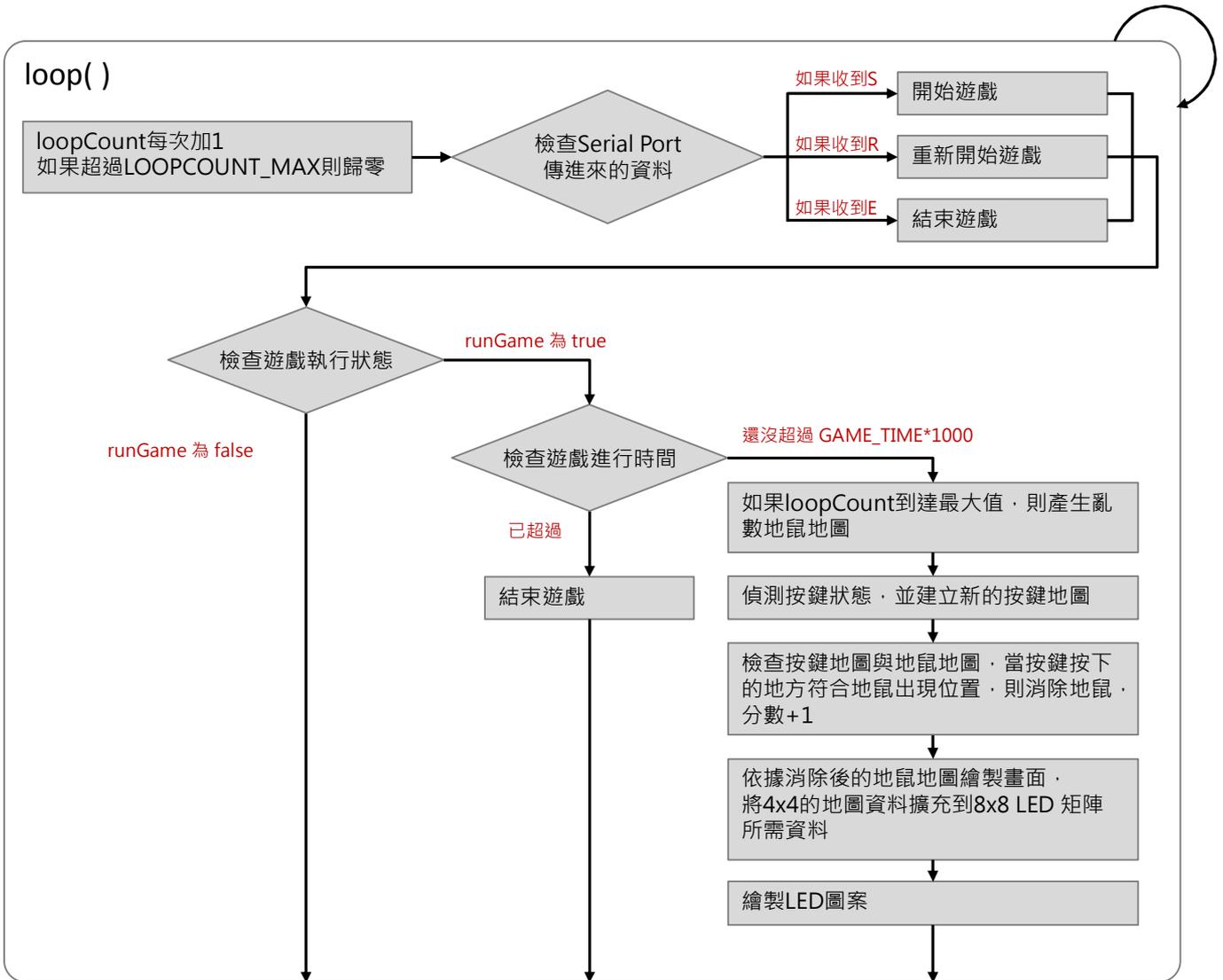


```
COM3
-----
You Have 30 Seconds To Play Each Game.
- Press 'S' To Start Game.
- Press 'R' To Reset Game.
- Press 'E' To End Game.
-----
Game is started!
-----
>> scored!   score :1
>> scored!   score :2
>> scored!   score :3
>> scored!   score :4
>> scored!   score :5
>> scored!   score :6
>> scored!   score :7
>> scored!   score :8
>> scored!   score :9
>> scored!   score :10
-----
Game end!
Total Score : 10
- Press 'S' or 'R' to play again.
-----
 Autoscroll   No line ending   115200 baud
```

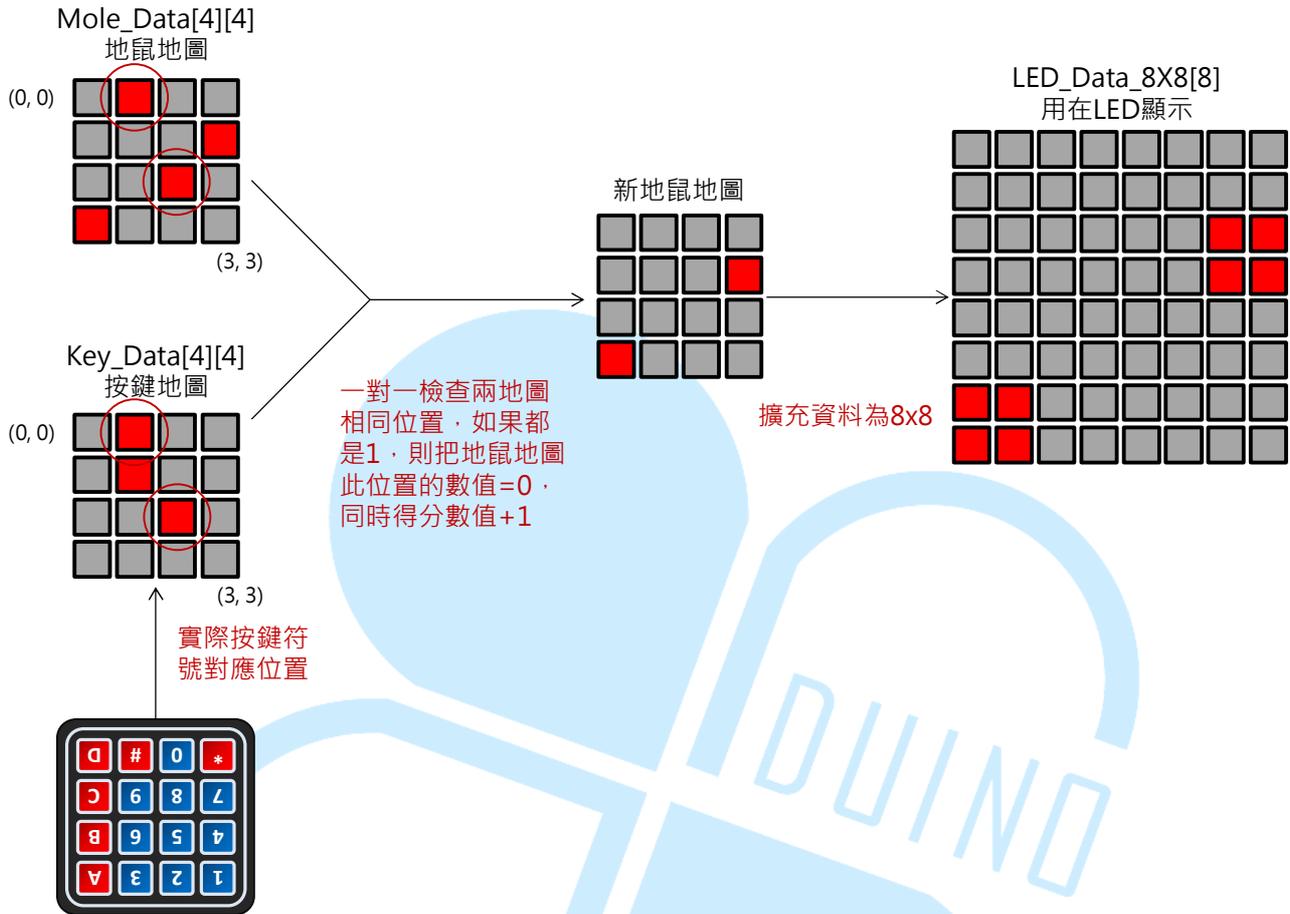
此范例程序增加了一些游戏执行时的变量，例如「score」为游戏计分、「gameTime」为游戏进行时间、「runGame」用在判断是否在游戏执行状态、「loopCount」决定随机地图在几次 loop 循环后要更新等等。另外，「#define DELAY\_TIME」设定 loop 间隔时间、「#define LOOPCOUNT\_MAX」决定随机地图在几次循环后要重新产生、「#define GAME\_TIME」决定游戏进行时间长度、「#define MOLE\_NUM\_MAX」决定一次出现的最大老鼠数量等等。函式则增加了「ClearMoleData()」、「ClearKeyData()」、「GameStart()」、「GameEnd()」等等，处理游戏的流程。

setup()阶段与前面范例类似，比较需要注意的是，由于此程序需要产生随机数，因此使用了「randomSeed( analogRead(0) );」语法，当作随机数产生器的初始化，参数栏内的「analogRead(0)」目的是使用一个无连接脚位，确保随机数产生出来够乱。

loop()内的流程较复杂，读者请参考下面概略流程图：



检查按键地图与地鼠地图的方式如下图：



产生随机地鼠地图的方法上，这里采用的是先随机产生地鼠数量  $N$ ，再随机产生  $N$  组范围在  $0\sim 3$  的行数、列数，并把这些位置的数值变成  $1$ （表示有出现地鼠），当然位置有可能会重迭，所以最多只会一次出现  $N$  只地鼠而已。语法「`long num = random( min, max )`」会让  $num$  的数值为介于「大于等于  $min\sim$  小于  $max$ 」之间的随机数字。

经由上述的流程以及矩阵键盘、LED 矩阵模块的配合，便可以完成一个简单的打地鼠游戏，而读者也可以在这个概念上加上其他 I/O 装置功能，例如用按钮启动游戏、RC 伺服马达驱动地鼠升降、用 7 段显示器显示游戏进行状态及得分数、打中地鼠时发出音效等等，有很大的发挥空间呢。

游戏的难度则可以藉由调整 `LOOPCOUNT_MAX`（调整地鼠地图的变动速度）、`MOLE_NUM_MAX`（调整地鼠一次出现的最大数量）的数值、`gameTime`（调整每回合时间长度），来达到不同的难度等级，这些就有待读者自行尝试改装啰。