

Using Matrix Keyboard with EduCake



1. Matrix Keyboard Introduction

In addition to the sensor, digital to analog conversion and serial port applications in the earlier application notes, matrix keyboard is another common interface we can use to interact with the 86Duino EduCake.

For an automation-control application, one of the key component is the user interface to the system that enables the user to send command and control signals to the system. The matrix keyboard with multiple inputs is an useful interface to send command and control signals. Matrix keyboard function is similar to the push button

input combines with the `digitalRead()` function, covered in the earlier application note. Programming a single push button control is quite different from a matrix keyboard with multiple input.

Matrix keyboard has been around for quite sometimes, and is readily available in the market in different sizes, configurations and mechanical form factors. There are off the shelf smaller matrix keyboards, 3x4 and 4x4, that are fabricated with 12 and 16 keys with 0 to 9, along with *, # and other marking.

For project with small number of simple control, such as to turn the device on/off, reset the device and to change the device's operating mode, where only a few buttons are needed, you can simply attach the required number of push button and associate each button to a digital I/O to capture user input.

For project with complex control that requires large number of input signals that are different, matrix keyboard is a good solution that can provide relatively large variation of input that utilize small number of I/O pins.

Programming a 4x4 matrix keyboard is similar to programming a 4x4 LED matrix, which we covered in an earlier application notes. Instead of using a group of I/O pins to control signal output to the LED matrix, programming for matrix keyboard is focusing on capturing I/O pins input signal and translate these signals into commands, as shown in figure-1.

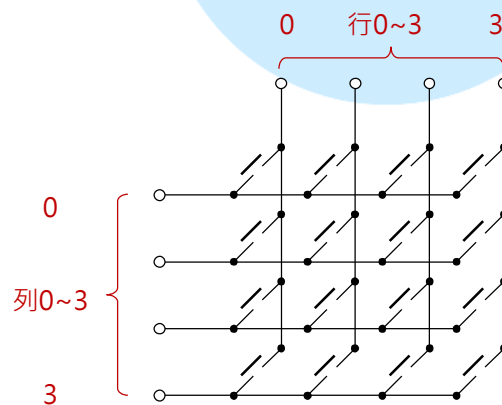


Figure-1. 4X4 matrix keyboard wiring.

Using push button control, which requires one I/O pin for each button, it takes 16 I/O pins to provide 16 button control. A 4x4 matrix keyboard with 16 input controls only occupies 8 I/O pins. The electronic circuitry to implement matrix keyboard is different from a simple push button control. Each of the 16 buttons on a 4x4 matrix keyboard has two pins, where one of the pin is connected to 3 other buttons on the same row and the other pin is connected to 3 other buttons on the same column, as shown in Figure-1.

The process and codes to read the status for each button on a matrix keyboard is more complex than the simple push button control, where each button is linked to one I/O pin. To read the status of the buttons on a matrix keyboard, you need to write codes to scan and read button status one row at a time or one column at a time. Figure-2 shows the process to scan through and read button status, one row at a time.

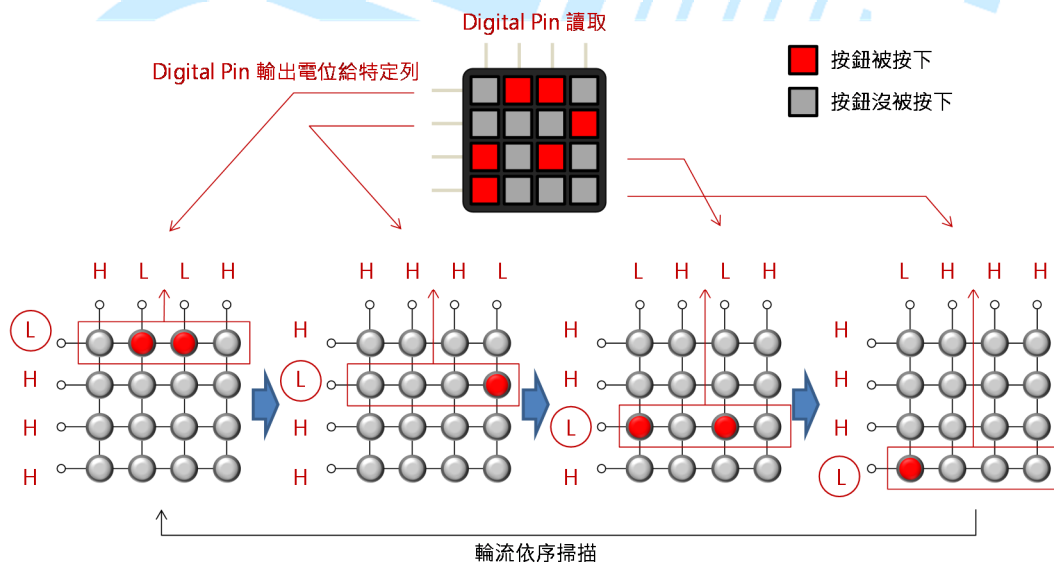


Figure-2. Scan and read button status on a 4x4 matrix keyboard

To scan through each row and read button status, as indicate in Figure-2, 4 digital input pins from the 86Duino EduCake are connected to the matrix keyboard to read the data with another 4 digital output pin connected to the keyboard to control the active row (or column) to be scanned and read button status. Depending on the circuitry, orientation for the matrix keyboard and the associated application

codes, you can scan through each row or each Column to read button status. For the exercise in this application note, we will use a 4x4 matrix keyboard, as shown in figure-3.

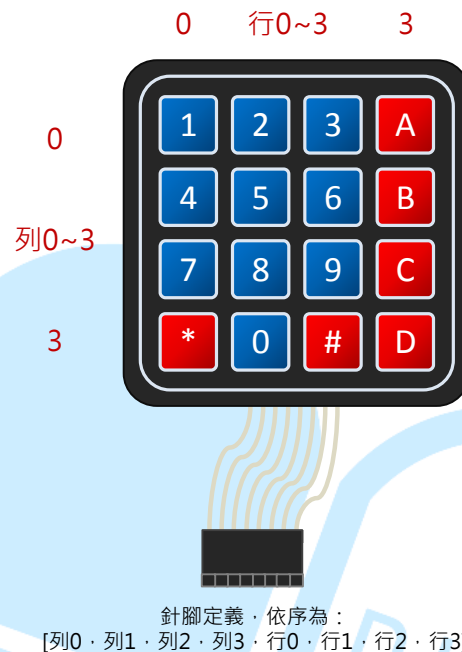


Figure-3. 4X4 matrix keyboard pin definition

For this matrix keyboard module, when a button is pressed, the corresponding pins for the row and column linked to the button are shorted. The wiring connections for the matrix key board is linked to the attached connector, as shown in figure-3, from left to right are as follow:

[Row-0, row-1, row-2, row3, column-0, column-1, column-2, column-3]

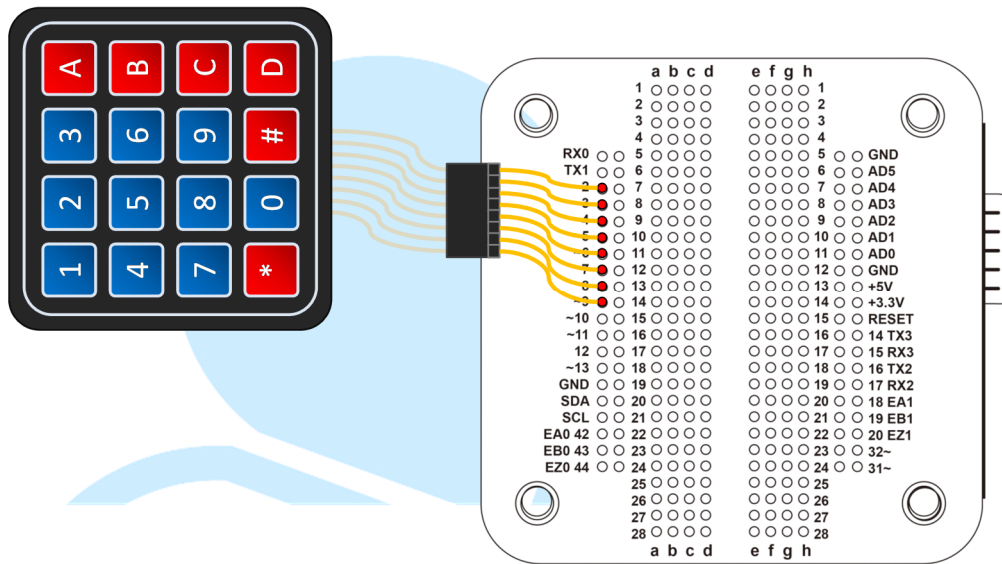
If you are using a different type of module, you need to check and identify how the pins are connected to the buttons.

The 86Duino EduCake has more than 20 usable I/O pins, where AD pin can be used as input to read pin status, suitable for I/O application that does not require high number of I/O pins. For application scenario that requires large number of keys, it's best to utilize IC chip designed to provide keyboard function.

2. First exercise: Keyboard scanning principle

In this section, we will work through an exercise using an EduCake and the 4x4 matrix keyboard mentioned in the previous section, to show how to scan through and read button status.

Attach the matrix keyboard to the EduCake as shown in the following figure:



Connection from the matrix keyboard for row 0~3 and column 0~3 are connected to Pin 9~2 on the EduCake. Launch 86Duino Coding IDE and enter the following codes:

```
const int Rows = 4; // Number of rows in the matrix
const int Cols = 4; // Number of columns in the matrix

// Map corresponding button to the matrix
char keys[Rows][Cols] =
{
    {'1', '2', '3', 'A'},
    {'4', '5', '6', 'B'},
    {'7', '8', '9', 'C'},
    {'*', '0', '#', 'D'}
};

// Previous button status
bool keys_status_last[Rows][Cols] =
{
    {false, false, false, false},
    {false, false, false, false},
    {false, false, false, false},
    {false, false, false, false}
};
```

```
// Associate pins from EduCake to the matrix keyboard
// Row 0~3 on the keyboard
int row_pins[Rows] = {9, 8, 7, 6};
// Column 0~3 on the keyboard
int col_pins[Cols] = {5, 4, 3, 2};

void setup()
{
    // Configure I/O mode for the pin attached to the keyboard
    // Read the voltage stage for the pins on the column
    for(int col = 0; col < Cols; col++)
    {
        pinMode( col_pins[col], INPUT_PULLUP );
    }

    // Use the pins associate with row as voltage source
    for( int row = 0; row < Rows; row++) // Scan row
    {
        pinMode( row_pins[row], OUTPUT );
        digitalWrite( row_pins[row], HIGH );
    }

    Serial.begin(115200);
}

void loop( )
{
    for( int row = 0; row < Rows; row++ ) // Scan column
    {
        // Voltage for this column goes LOW
        digitalWrite( row_pins[row], LOW );
        for( int col = 0; col < Cols; col++) // Scan column
        {
            // Read voltage level from column
            // It's Low when the button is pressed.
            boolean result = !digitalRead( col_pins[col] );

            // Button press is detected.
            // If previous button status is pressed, voltage
            // unchanged.
            if( result == HIGH && keys_status_last[row][col] ==
true )
            {
                Serial.print("Button ");
                Serial.print(keys[row][col]);
                Serial.println(" hold");
            }
        }
    }
}
```

```

// Previous button status is not pressed.
// Indicates a new button press event is detected.
else if( result == HIGH && keys_status_last[row][col] ==
false )
{
    Serial.print("Button ");
    Serial.print(keys[row][col]);
    Serial.println(" pressed");
}

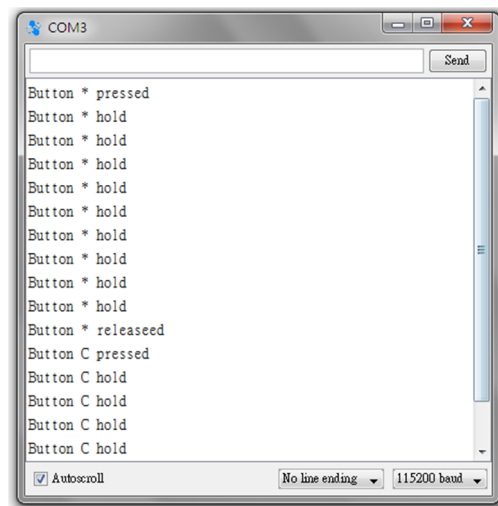
// Current scan: Button press is not detected
// Button previous status is pressed
// Indicate the button just been released
else if( result == LOW && keys_status_last[row][col]
== true )
{
    Serial.print("Button ");
    Serial.print(keys[row][col]);
    Serial.println(" released");
}
keys_status_last[row][col] = result; // Change
button status
}

// Switch row voltage to high
digitalWrite( row_pins[row], HIGH ); }

delay( 20 );
}

```

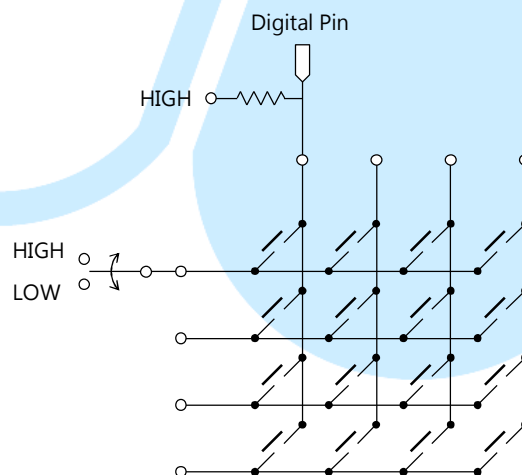
After the above sketch (code) is deployed to the EduCake, launch the serial monitor and press some buttons on the matrix keyboard to see the associated output to the serial monitor, as shown in the following figure:



In the beginning, there are code entries to configure and set the size of columns and rows for the matrix keyboard and associate the "char keys[]" with the button layout on the keyboard, initialize the "keys_status_last[]" array which is used to cache the previous button status and initialize the "row_pins[]" and "col_pins[]" arrays to link the pins from the matrix keypad to the I/O pins on the 86Duino EduCake.

In the "setup()" function, the I/O pins associated with the "col_pins[]" array are configured as input with internal pullup resistor, and the I/O pins associated with the "row_pins[]" array are configured as output. The last line of code configure the Baud Rate for the serial port.

Please note the I/O pins associated with the col_pins[] array are configured as INPUT_PULLUP mode, and the pins associated with the row_pins[] array are configured as OUTPUT and initialized to HIGH. Each of the pins associated with the row_pins[] array is pull low to scan and detect button status. When a button is pressed, a voltage LOW condition is created, as shown in the following figure:



In the main program loop, there are two nested For loops. The first For loop is used to scan the rows. The second For loop, nested within the first For loop, is used to scan the columns, to read voltage status associate with each of the button.

In the first For loop, as the code scan through each of the rows, the following line of code set voltage LOW condition to the row being scanned:

```
digitalWrite(row_pins[row], LOW);
```

When the button is not pressed, the column which the button is attached to is in voltage HIGH condition. When a button is pressed, it bridges the column which the button is attached to the row currently scan and cause voltage LOW condition to the column.

In the second For loop, it loops through the following line of code to detect button press status for the button attached to each of the column:

```
Boolean result = !digitalRead(col_pins[col]);
```

Then, the following line of code compare the current button status with the previously scanned status to detect the following condition:

- **Press and hold:** When the current status indicates the button is pressed and the previously scanned status is also pressed, it indicates the button has been pressed continuously.
- **New button pressed event:** When the current status indicates the button is pressed and the previously scanned status is not pressed, it indicates a new button press event.
- **Button released event:** When the current status indicates the button is not pressed and the previously scanned status is pressed, it indicates the button was pressed and just released.

While it's simple, the above button status scanning technique are useful in different type of application, to enter information needed by the program, to control program execution, such as code that control a motor. When press and hold the button, keep the motor running. When button is released, stop the motor.

3. Second exercise: Using the Keypad library

In this exercise, we will talk about the keypad library, using the same circuitry from the previous exercise.

When a button on the matrix keypad is pressed, it does not generate a clean one time transition from voltage High from voltage Low.

Instead, a series of Debounce signal is generated. Depending on the mechanical design and build quality for the matrix keypad, instead of a single transition event from voltage high to voltage low when the button is pressed, multiple transition events between voltage high and voltage low will take place within a very short period of time before settling the line signal stabilize at voltage low condition. Debounce or De-bouncing is an expected condition, when working with mechanical push button. Debounce makes it appear the button has been pressed, released and pressed multiple times, rapidly within a small fraction of second between each event, which can be in the 10ms range or faster. It's humanly not possible for us to repeatedly press the button at such rapidly rate.

The Keypad library, used for this exercise, includes codes to handle Debounce condition and help simplify the code we write to read and detect button press event. The Keypad library is available for download from the following URL:

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

To use the keypad library, unzip the downloaded file and copy the complete "Keypad" folder (including sub-folders and content) to the "\\86Duino_Coding_xxx\\Libraries" folder, where

"\\86Duino_Coding_xxx\\" is the directory where you installed the 86Duino development IDE.

Use a text or source code editor to edit the "Keypad.h" header file, in the "\\Keypad" folder, find the [#include "WProgram.h"] entry and change it to [#include <Arduino.h>]. Next, repeat the same task on the "Key.h" header file, locate in the "\\Keypad\\utility" folder.

Launch 86Duino Coding IDE and enter the following code:

```
#include <Keypad.h>

const byte Rows = 4; // number of rows in the matrix
const byte Cols = 4; // number of column in the matrix
// Associate keypad buttons to the matrix
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
// Associate EduCake I/O pins to the matrix
byte row_pins[Rows] = {9, 8, 7, 6}; // 列 0~3
byte col_pins[Cols] = {5, 4, 3, 2}; // 行 0~3

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

  if( keypad4X4.getKeys( ) )
  {
    // Check each of the button within the 4x4 keypad object
    for( int i = 0; i < LIST_MAX; i++)
    {
      // If button status changed, output to serial monitor
      if( keypad4X4.key[i].stateChanged )
      {
        Serial.print("Button ");
        // output character symbol for the pressed button
        // to serial monitor
        Serial.print(keypad4X4.key[i].kchar);
        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 );

}
```

Compile and upload the above sketch to the EduCake and launch the serial monitor.

As you press a button on the keypad, you can see corresponding output from the serial monitor match the pressed key, similar to the example in the first exercise.

However, in addition to the matrix size declaration, associating button symbol to the matrix array and I/O pins from the EduCake to the matrix keypad that are similar to the code in the first exercise, the `[#include <Keypad.h>]` statement is added along with the following line of code that create the Keypad class object to take advantage of the function provided by the Keypad library, enabling us to simplify the code needed to work with matrix keypad:

```
[ Keypad keypad4x4 = Keypad(makeKeymap(keys), row_pins, col_pins,
Rows, Cols); ]
```

The only code needed in the `setup()` section is to initialize the serial port.

In the main program loop, after using the `" keypad4x4.getKeys() "` function to read button status from the keypad object, the codes within the following For loop iterate through the keypad object to read the status for each of the button and output to the serial monitor:

```
For(int i = 0; i < LIST_MAX; i++)
```

Within the above For loop, the Keypad library provide the following functions that help simplify the code:

- The `" keypad4x4.key[i].stateChanged "` function is used to detect when the status of the button has changed.

- The " keypad4x4.key[i].kchar " function is used to identify which button on the keypad is currently being processed.
- The " keypad4x4.key[i].kstate " function is used to read the button status.

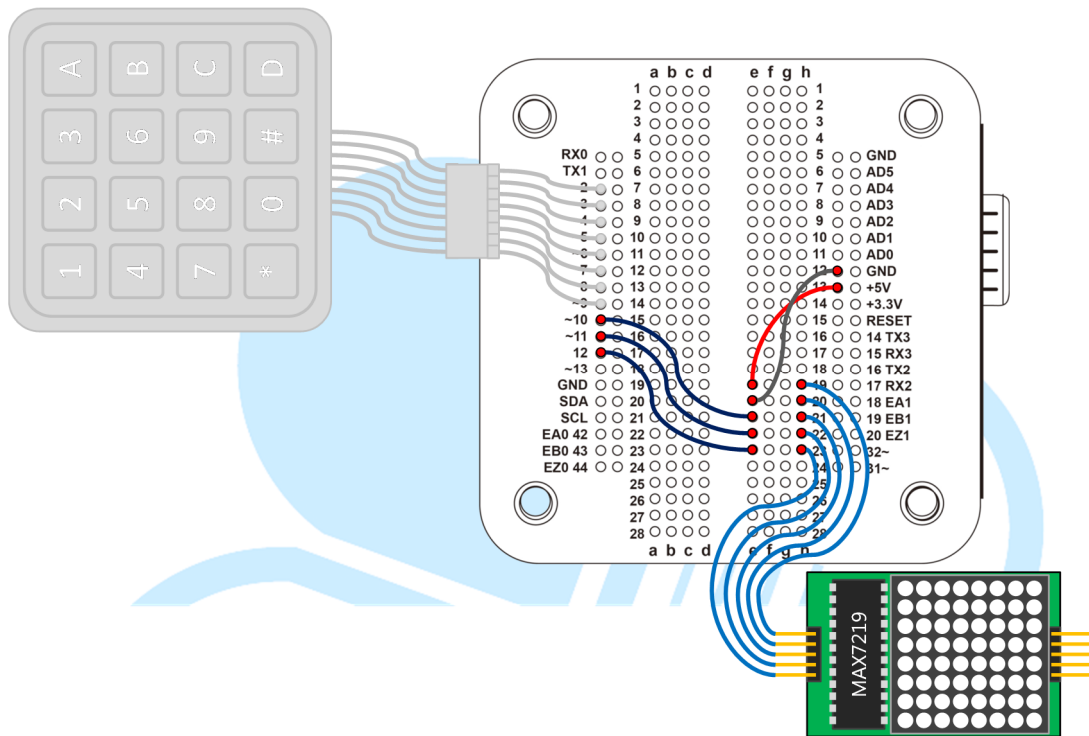
In addition to detect which button on the keypad is pressed, the code in this exercise also identify the following conditions:

- The button is pressed.
- The button is pressed and hold (the button was pressed during the previous scan cycle).
- The button is released (the button was pressed during the previous scan cycle).
- Idle (Button is not pressed).

Without the help from the Keypad library, it requires much more lengthy and complicated codes to accomplish the same result.

4. Third exercise: Keypad Library & 8x8 LED Matrix

For the exercise in this section, the MAX7219+8x8 LED matrix is used to demonstrate a more complex application scenario, as shown in the following figure:



In an earlier application note, we've talked about matrix LED, using the MAX7219+8x8 module. Since working with matrix LED is fairly common for Arduino and 86Duino developer, it's good to compose a library that encapsulate common function to help simplify application development.

Let's work through the following steps to create the LEDmat8 library to support the MAX7219+8x8 matrix LED module:

1. Create a new folder and name the folder "LEDmat8" under the following directory:

`\86Duino_Coding_xxx_Win\Libraries`

Note: "`\86Duino_Coding_xxx_Win`" is the directory where you installed the 86Duino development tool, 86Duino Coding IDE, where "`xxx`" is the installed version.

2. In the newly created “\LEDmat8” folder, create a new file, name the file as “LEDmat8.h” and enter the code from the following listing into this file:

```
#ifndef LEDMAT8_H
#define LEDMAT8_H

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

// MAX7219 register
#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. In the “\LEDmat8” folder, create a new file, name the file as “LEDmat8.cpp” and enter the code from the following listing into this file:

```

#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 );

    // Initialize MAX7219 register
    MAX7219_1Unit( max7219_REG_scanLimit, 0x07 );
    MAX7219_1Unit( max7219_REG_decodeMode, 0x00 );
    MAX7219_1Unit( max7219_REG_shutdown, 0x01 );
    MAX7219_1Unit( max7219_REG_displayTest, 0x00 );

    for(int i = 1; i <= 8; i++) { // Turn off all LED
        MAX7219_1Unit( i, 0 );
    }
    // Set brightness range, 0x00 ~ 0x0f
    MAX7219_1Unit( max7219_REG_intensity, 0x0f );
}

// Draw the whole LED display
void LEDmat8::DrawLED( byte *LED_matrix )
{
    byte i = 8;
    byte mask;

    while( i > 0 )
    {
        mask = (0x01 << (i - 1)); // Bitmask, starting from
        left
        digitalWrite( CLOCK_pin, LOW ); //
        if ( data & mask ) { // Use Bitmask to determine
        corresponding bit
            digitalWrite( DIN_pin, HIGH ); // If it's 1, DIN
            output HIGH
        }
        else{
            digitalWrite( DIN_pin, LOW ); // If it's 0, DIN output
            LOW
        }
        i--;
    }
}

```

```

    }
    digitalWrite( CLOCK_pin, HIGH ); //
    i = i - 1; // move to next bit
  }
}

// Control a MAX7219 module
void LEDmat8::MAX7219_1Unit ( byte reg_addr, byte
reg_data )
{
  // Before sending data, set LOAD_pin to LOW
  digitalWrite( LOAD_pin, LOW );

  SPI_SendByte( reg_addr ); // Send register address
  SPI_SendByte( reg_data ); // send data

  // After data is sent, set LOAD_pin to HIGH
  digitalWrite( LOAD_pin, HIGH );
}

```

Launch 86Duino Coding IDE and enter the following code:

```

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

const byte Rows = 4; // number of rows
const byte Cols = 4; // number of columns
// Corresponding symbols mapped to the keypad
char keys[Rows][Cols] =
{
  { '1', '2', '3', 'A' },
  { '4', '5', '6', 'B' },
  { '7', '8', '9', 'C' },
  { '*', '0', '#', 'D' }
};

// Associate EduCake I/O Pin# to the matrix
byte row_pins[Rows] = { 9, 8, 7, 6 }; // Row 0~3
byte col_pins[Cols] = { 5, 4, 3, 2 }; // Column 0~3

// Keypad library object
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins,
col_pins, Rows, Cols );

// LED module control pins
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;
};

// Associate EduCake I/O Pin# to the matrix

```

```
byte row_pins[Rows] = {9, 8, 7, 6}; // Row 0~3
byte col_pins[Cols] = {5, 4, 3, 2}; // Column 0~3

// Keypad library object
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins,
col_pins, Rows, Cols );

// LED module control pins
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;

// 8X8 LED matrix object
LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin,
CLOCK_pin );

byte LED_Data_8X8[8] = { // data matrix for LED display
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000
};

void ClearLED_Data( ) // Clear LED display data
{
    for(int i = 0; i < 8; i++)
    {
        LED_Data_8X8[i] = B00000000;
    }
}

void setup ( ) {
    LedMatrix.Init();
    delay(1000);
}

void loop ( ) {
    ClearLED_Data( );

    // check keypad status
    keypad4X4.getKeys( );

    // Check each of the item in the keypad4X4 array
    for( int i = 0; i < LIST_MAX; i++ )
    {
```

```
// Check whether the button is pressed
if( keypad4X4.key[i].kstate == PRESSED )// HOLD
{

    // Update display corresponding to the key pressed
    // "D" correspond to top-left corner
    // "1" correspond to lower-right corner
    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

// Draw LED display
LedMatrix.DrawLED( LED_Data_8X8 );
delay( 50 );

}
```

After the above code is compiled and uploaded, you can press a button on the keypad to turn on the corresponding LED.

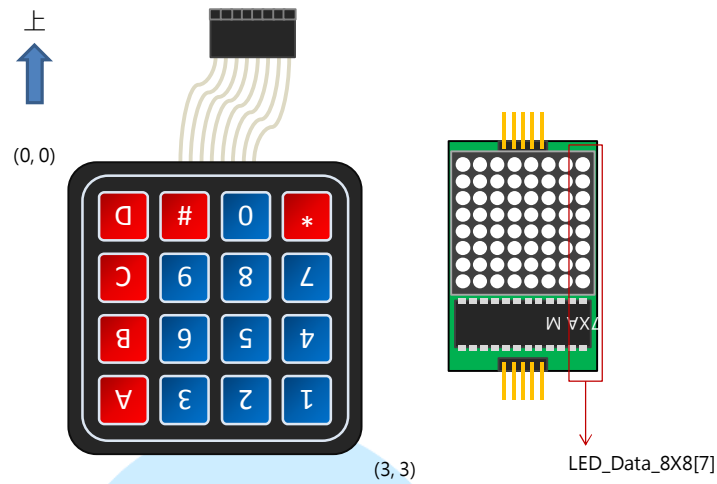
The above example uses the Keypad library and 8x8 LED application code from a previous application note, encapsulate the code from the 8x8 LED application into the LEDmat8 library.

The variable declaration in the beginning section is the same as in the 2nd exercise with the following variables for the LED matrix added:

- LED_Data_8x8[8] byte array to hold LED display data.
- ClearLED_Data() function to clear data in the LED_Data_8x8[] byte array.
- LedMatrix variable that represent the LED matrix object.

Within the setup() function, the LedMatrix.Init() function is called to initialize the LED matrix, looping through the LED matrix and call the ClearLED_Data() function to clear data. The keypad4x4.getKeys() function is called to retrieve update keypad button status. Then, the keypad4x4.key[i].kstate function inside a For loop to read button status for each of the button, follow by a series of switch statements to set the corresponding display status to the LED_Data_8x8[] array. The LedMatrix.DrawLED() function is call last within the Setup() function to draw the LED display (turning on LED corresponding to the button pressed).

The LED display corresponding to the button in the above example is based on the orientation of the matrix keypad and LED matrix, as shown in the figure below:



If the LED display on the LED matrix display does not correspond to the button press on the keypad, you need to change the orientation of the keypad, LED matrix or modify the code to get the expected result.

You can experiment and change the button press condition in the following statement to see different result on the LED matrix:

- If (keypad4x4.key[i].kstate == PRESSED)

5. Fourth exercise: Whac-A-Mole Game using Matrix Keypad & 8x8 LED Modules

In this last exercise, we will create a whac-a-mole game using the same circuitry with matrix keypad and 8x8 LED modules.

Launch 86Duino Coding IDE and enter the code from the following listing:

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

const byte Rows = 4; // declare number of rows for the matrix
const byte Cols = 4; // declare number of columns for the matrix
// Associate keypad symbols to the keys[] array
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};

// Associate EduCake I/O pins to the keypad
byte row_pins[Rows] = {9, 8, 7, 6}; // rows 0~3
byte col_pins[Cols] = {5, 4, 3, 2}; // columns 0~3

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

// Define LED module control pins
int DIN_pin = 10;
int LOAD_pin = 11;
int CLOCK_pin = 12;

// 8X8 LED matrix object
LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin, CLOCK_pin );

// variables for the game
int score = 0;
long gameTime = 0;
boolean runGame = false; // Variable to indicate active game
int loopCount = 0;
#define DELAY_TIME      50 // time delay between loop
#define LOOPCOUNT_MAX  30 //
#define GAME_TIME       30 //
#define MOLE_NUM_MAX    6 // max number of mole at same time
```

```
byte LED_Data_8X8[8] = { // LED matrix display data
    B00000000, // Left -> Right = row 1
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000,
    B00000000 // Left -> Right = row 8
};

// mole location array, [column] [row]
boolean Mole_Data[4][4] = {
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0}
};

// keypad array [column] [row]
boolean Key_Data[4][4] = {
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0}
};

void ClearLED_Data( ) // Clear LED display
{
    for( int i = 0; i < 8; i++ )
    {
        LED_Data_8X8[i] = B00000000;
    }
}

void ClearMoleData( ) { // Clear mole array data
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            Mole_Data[i][j] = false;
        }
    }
}

void ClearKeyData( ) { // Clear keypad array data
    for( int i = 0; i < 4; i++ ) {
        for( int j = 0; j < 4; j++ ) {
            Key_Data[i][j] = false;
        }
    }
}
```

```

void GameStart ( ) { // Start game
  ClearMoleData( ); // initialize mole display
  ClearKeyData( ); // initialize keypad data
  runGame = true; // set game to active mode
  score = 0; // Initialize game score
  gameTime = millis( ); // initialize game time.
}

void GameEnd ( ) { // Stop game, output game score to serial
  // monitor

  Serial.println("-----");
  ;
  Serial.println("Game end!");
  Serial.print(" Total Score : "); Serial.println(score);
  Serial.println(" - Press 'S' or 'R' to play again.");

  Serial.println("-----");
  ;

  // display that indicate end of the game
  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) ); // Initialize random number generator

  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( );

        // Start the game when the S key is pressed
        if( ch == 's' || ch == 'S' ) {
            Serial.println("-----");
            Serial.println("Game is started!");
            Serial.println("-----");
            GameStart( );

        }

        // When R is pressed, restart the game
        else if( ch == 'r' || ch == 'R' ) {
            Serial.println("-----");
            Serial.println("Game is reset!");
            Serial.println("-----");
            GameStart( );

        }

        // When E is pressed, end the game
        else if( ch == 'e' || ch == 'E' ) {
            GameEnd( );
        }

    }
}
```

```

// Check elapsed game time
if( runGame ){
    long time = millis( ) - gameTime; // elapsed game time

    // Check whether max game time elapsed
    if( time < GAME_TIME*1000 ) {
        // Randomly generate position for mole to appear
        // Refresh mole data after reaching max loopCount
        if( loopCount == LOOPCOUNT_MAX ) {
            ClearMoleData( );
            // randomly generate the number of mole to appear
            long mole_num = random( 0, MOLE_NUM_MAX+1 );
            for( int i = 0; i < mole_num; i++ ) {
                // random number between 0 to 4
                long i_num = random(0,4);
                // random number between 0 to 4
                long j_num = random(0,4);
                // Serial.print(i_num);
                // Serial.print(",");
                // Serial.println(j_num);
                Mole_Data[i_num][j_num] = true;
            }
        }

        // Read button status from keypad
        // Compare keypad status with mole location array
        // and adjust game score
        ClearLED_Data( ); // Clear LED data before update
        ClearKeyData( ); // Clear key data before update
        keypad4X4.getKeys( ); // update keypad status

        // check each button in the keypad4X4 array
        for( int i = 0; i < LIST_MAX; i++ )
        {
            // Check whether button is pressed
            if( keypad4X4.key[i].kstate == PRESSED )
            {
                // update array data based on key pressed
                // button "D" is at upper-Left corner
                // button "1" is at lower-right corner
                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;
```

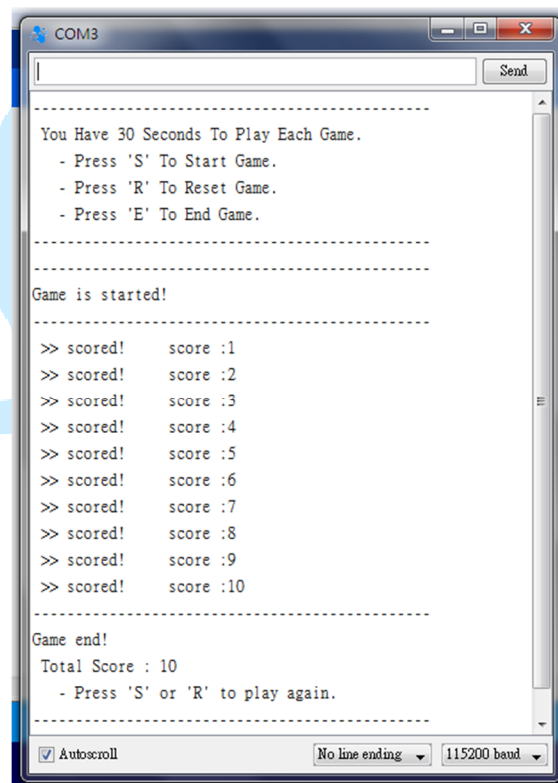
```
// end for

// Compare data between keypad and mole array
for( int i = 0; i < 4; i++ ) { // Row
  for( int j = 0; j < 4; j++ ) { // Column
    // When button pressed and mole appeared location match
    // Clear location data in the mole array
    // and increase game score by 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);
    }
  }
}

// Use mole array as display data for the LED matrix
// to show location where the mole has been whacked
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
  }
}

// Draw display to LED matrix
LedMatrix.DrawLED( LED_Data_8X8 );
```

Compile and upload the sketch to EduCake. Then, launch the serial monitor. As the code execute, the serial monitor display information showing how to start, reset and end the game. Each game session goes on for 30 seconds. When you start the game, the serial monitor display score for the game as you play, as shown in the following figure:



```
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
```

The code for this exercise includes variables such as 「`score`」 to keep track of game score, 「`gameTime`」 to keep track of elapsed game time, 「`runGame`」 to indicate whether a game is active and 「`loopCount`」 to keep track of number of game loop and update data accordingly. In addition to these variables, the following `#define` statements were used to define game parameters:

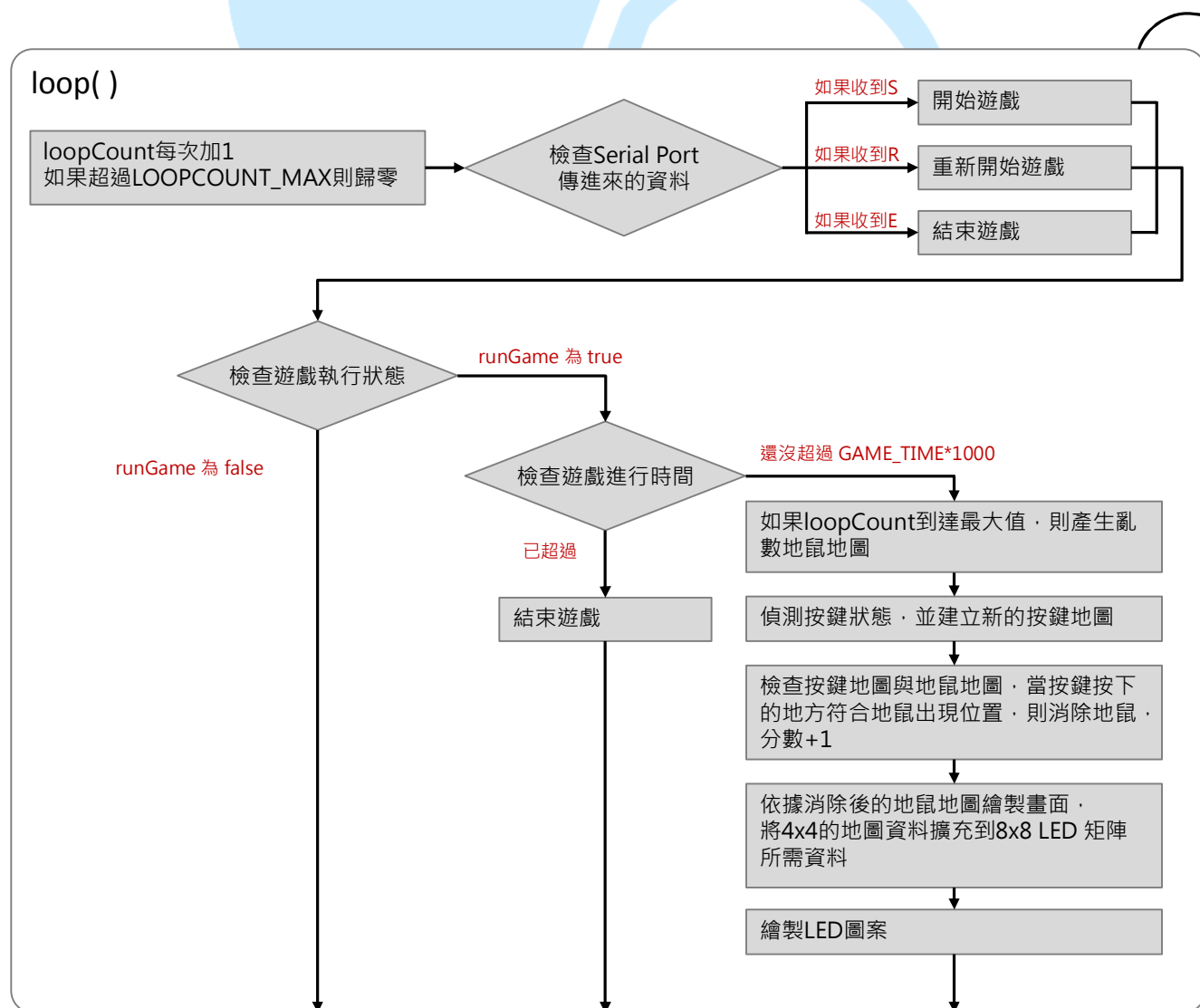
- `#define DELAY_TIME` (To define the delay time between loop)
- `#define LOOPCOUNT_MAX` (To define the maximum loop before refresh the game)

- #define MOLE_NUM_MAX (To define max number of Mole that can appear at the same time)

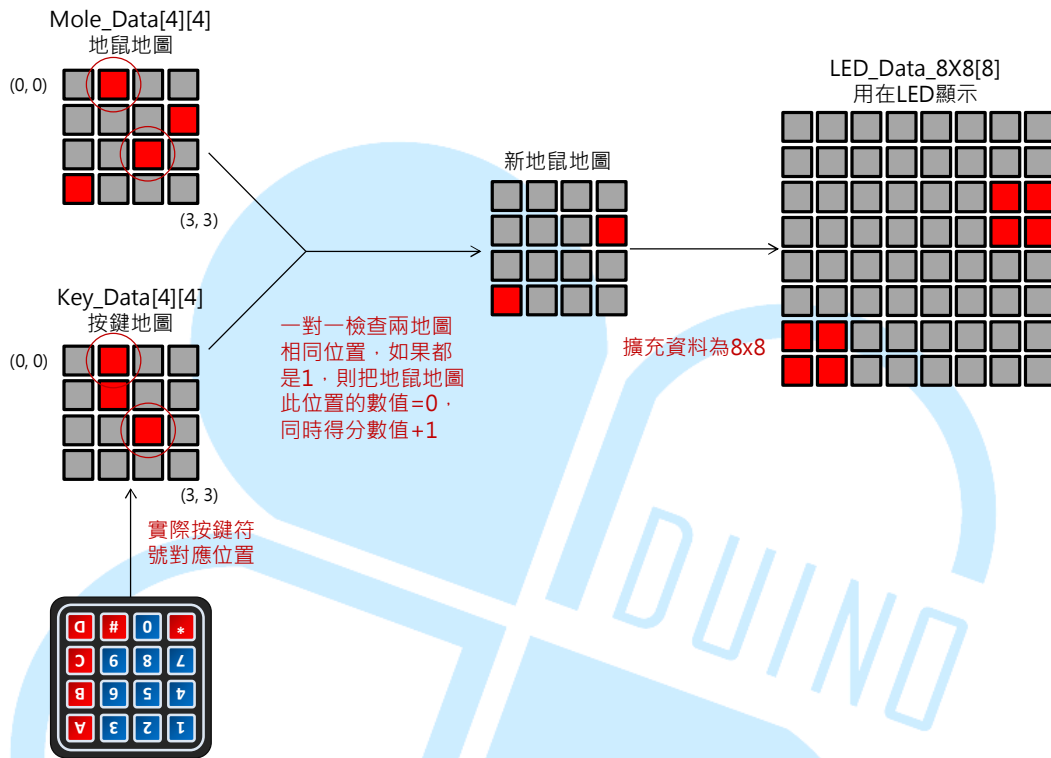
The「ClearMoleData()」,「ClearKeyData()」,「GameStart()」and「GameEnd()」 functions are used to control game flow.

Within the「setup()」function, to support the random variables needed for the game, the 「randomSeed(analogRead(0));」 statement is used to initialize random variable generator. The「analogRead(0)」function, used as the parameter for the randomSeed() function, is link to an I/O pin that is not connected, to insure randomness.

Following is the flow chart for the game:



Following is the flow to check button pressed on the keypad to the Mole matrix array:



To randomly generate moles to appear on the Mole matrix array, we first randomly generate N number of mole to appear. Then, randomly assign these moles to the Mole matrix array.

Based on the flow in the previous figure, display on the LED matrix is used to represent the location where the mole appear. To make the game more interesting, you can further expand the game by adding more complex components, such as servo to raise and lower the moles, 7 segment LEDs to display game progress, audio output when a mole is whac and etc.

The game complexity can be changed by changing the value for **LOOPCOUNT_MAX** (which affect how quickly the mole display data is updated), **MOLE_NUM_MAX** (which control the max number of moles

that can appear at the same time) and `gameTime` (which control available time to play the game).

