EduCake はマトリックスキーボードを必要とする

一、　マトリックスキーボード原理紹介

使用者と 86Duino EduCake のインタクラティブ方式についてだが、先の章で述べたセンサー応用、A-D コンバーター読み取り、Serial Port データ送信等の方式を除いて、その他よく見られる入力方式は、「マトリックスキーボード」という装置を用い、通常コントローラーにおいて使用者が命令やデーター等を入力することを必要とする。

実際、マトリックスキーボードの原理は、先の章で紹介した「マイクロスイッチボタン」と「digitalRead()関数」を組合せたものによく似ており、単一のキーからボタンマトリックスを変更するだけである。

しかしながら、キーの数は多くなり、制御プログラミングコーディング方式も異なってしまっているので、本章では、マトリックスキーボードの原理から紹介することとし、同時に 86Duino EduCake を使用して興味深い機能を実作してみたいと思う。

通常マーケットで購入可能なマトリックスキーボードの種類は多様化し、あるものは膜接触導通、あるものはメカニカル膜接触導通とあるが、使用原理は大体一緒である。

本章では小型マトリックスボードについて述べることとするが、良く目にする 3X4 或いは 4X4 等は、上部に 0~9、英文符合、その他の符合例えば「*」「#」等が表示される。

パソコンのキーボードはとても良い例である。

読者が、もし自分の特殊なキーボード設定を行いたいと思うのであれば、本章を読んだ後、自身で行ってみるとよい。

プロジェクトに使用されるキーの数は指で数えられるほどしか必要としないが、実作では実際の所、キーボード一つが digital ピン一つに対応しているだけである。しかしながら、3X4 或いは更に大きなキーを使用する場合、必要と

なるピンの数量は大きく増加すし、当然、コントローラの貴重なピン空間を占めてしまう。

マトリックスキーボードは「スキャン」の概念を使用しているため、行ごと列ごとにキーボードの状態をスキャンし、ピンの数量を節約することを可能としている。先に挙げた 8X8 LED マトリックススキャンのディスプレイは同様の道理となっている。一つの 4X4 マトリックスキーボードを例とし、電気回路を下記の図 1 の通りとする：

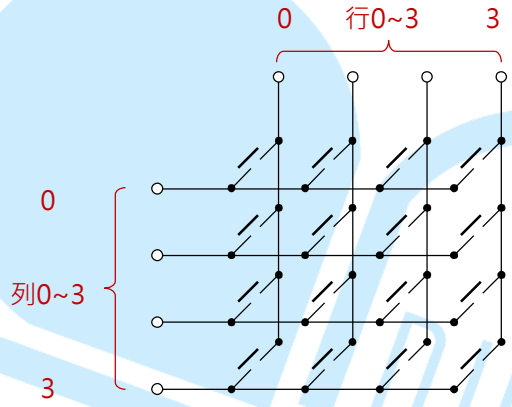


図 1. 4X4 矩陣鍵盤電路接線図

読者は電気回路図から、この様な接続方式により、4X4 のマトリックスキーボードが、コントローラーに必要とするのは 8 個のピンで、16 個では無いことが見て取れると思う。これは「一つが一つのことを読み取るキーボード方法」に相対し、ピンの数量を半減することが可能となっている。

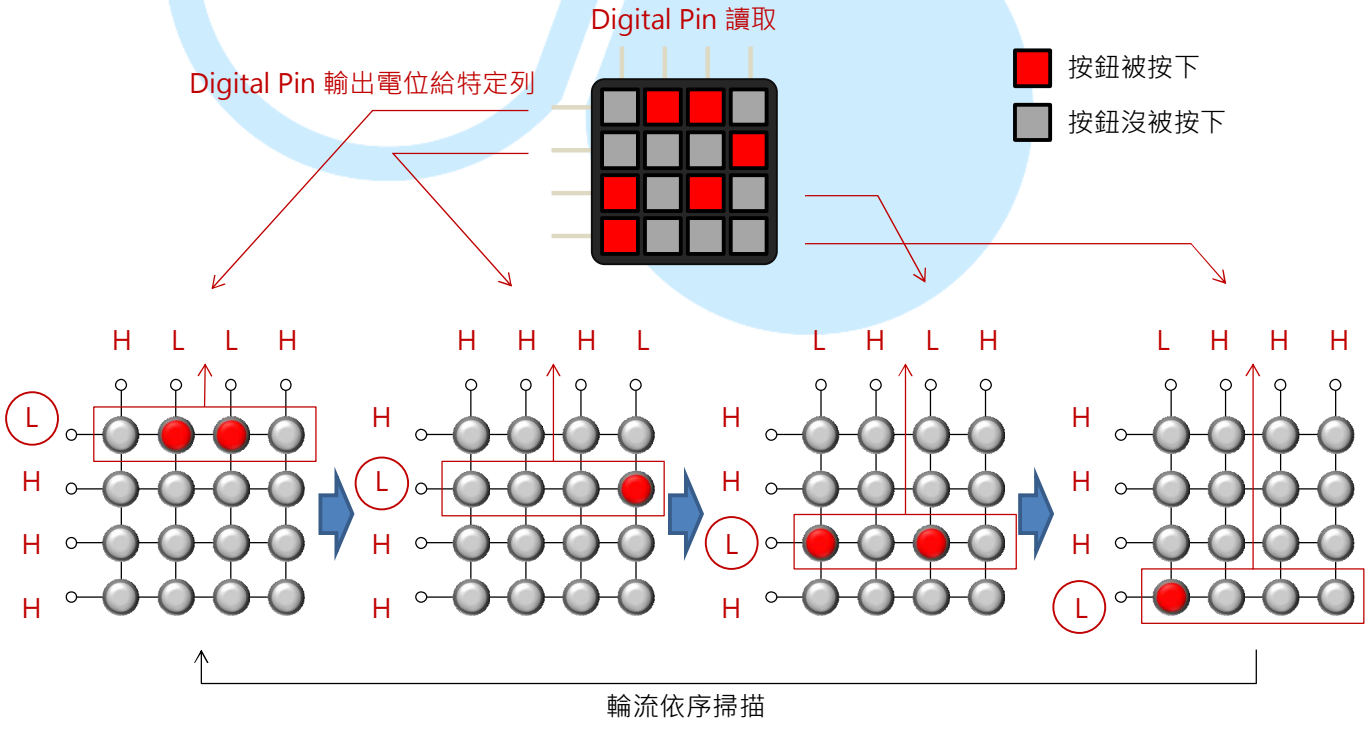
ある一列のキーについては、キーサイドに全て密接しているものの、この様な配線方式はプログラミングがやや複雑なプロセスにより処理を行うことを必要とする。スキャンプロセス中、コントローラープログラミングは連続してある一列に対し HIGH 或いは LOW の電圧値（状況決定に基づく）を与え、その後この一列のそれぞれの業のキー電圧を読み降り、下記の図 2 に示したようなプロセスをスキャンする。

図 2. 4X4 マトリックスキーボードのスキャンプロセス図

ここで注意しなくてはならないのは、次々にスキャンしていく時、一度ある一列の電圧が HIGH 或いは LOW であれば、その他の列にはそう反する電圧を必ず加えなくてはならない。

さもなければ、ある行の電圧を読み取った時、どの列のキーなのか区別することができなくなる。

続く 86Duino EduCake の実作において、私たちは一般的に容易に手に入る、下記に挙げる図 3 の様な 4X4 マトリックスキーボードモジュールを選択することにしよう：



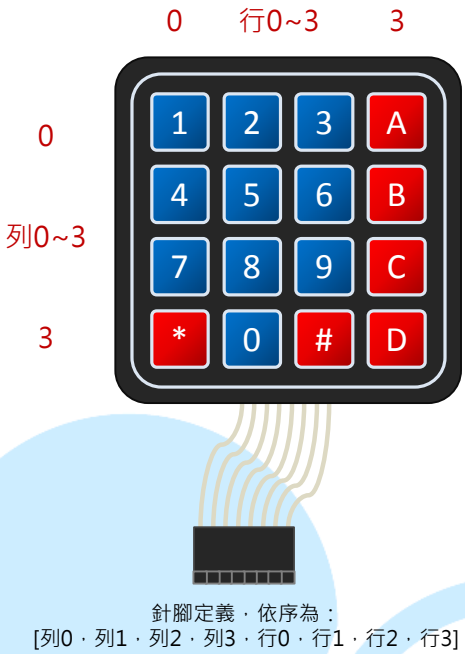


図 3. 4X4 マトリックスキーボードステッチ定義

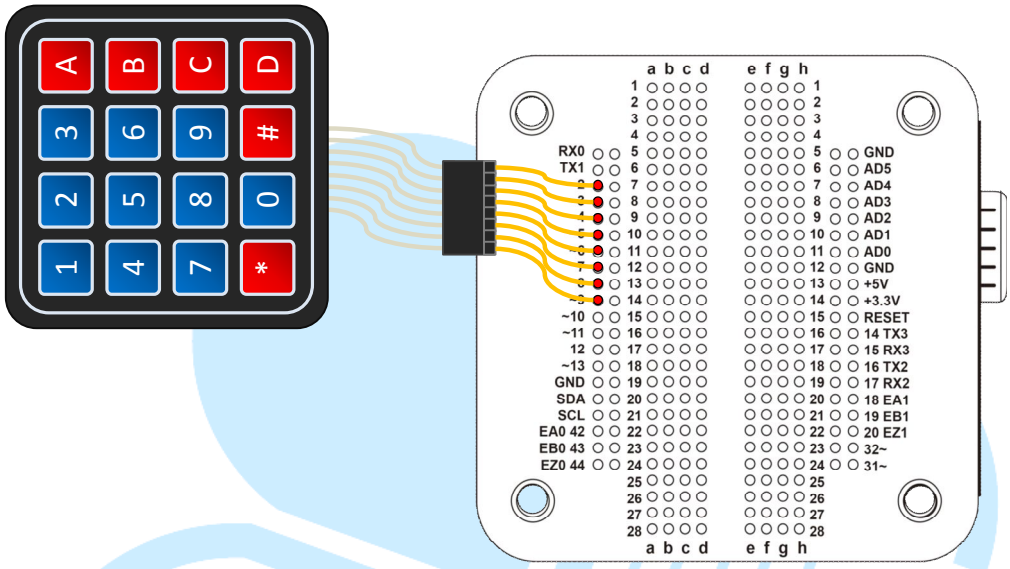
このマトリックスキーボードモジュールについては、キーで入力するのは電気回路電通であり、各行の配線は既に 8 ピンプラグで配列され、ピンの定義は左から右へと順番に「列 0，列 1，列 2，列 3，行 0，行 1，行 2，行 3」である。しかしながら、もし、実作時に購入したのが他のキーモジュールだとしたら、一番良いのはまずマルチメーターを搭載し、指で特定のキーを押して、ピンの定義を測ることだ。この様にして配線とコーディングはようやく正常に機能する。

86Duino EduCake に用いられるピンの数量は、およそ二十数個であり（AD ピンを読み取り用途とする）、その為、本章の実作方式は、比較的このような中等キー数量に適合するマトリックスキーボードにおいて、もし、読者が自身の手でパソコンキーの様に多くのキー装置を作りたいのであれば、最良のなのはやはり専用の IC を搭載することである。

下は私たちが利用するプログラミングであり、実際マトリックスキーボードの原理を練習する際に、同時に 4X4 マトリックスキーボードモジュールを使用して実際に応用機能を作ってみよう。

二、　第一プログラミング-キーボードスキャン原理練習

第一の先例プログラミングを用いて、86Duino EduCake を如何に使用して上述したマトリックスキーボードスキャンの原理を実作するか練習しよう。読者にはまず、下図に基づいて配線してもらいたい：



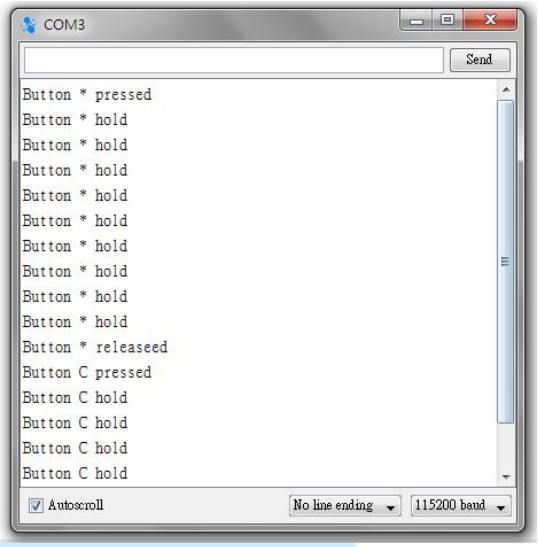

```
constint Rows = 4;// 行数
constint Cols = 4;// 列数
// 符合に対応するキー入力
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
{'*','0','#','D'}
};
//先の状態のキーを記録
boolkeys_status_last[Rows][Cols] =
{
  {false,false,false,false},
  {false,false,false,false},
  {false,false,false,false},
  {false,false,false,false}
};
      // ピンコードを定義
```

列列 0~3，行 0~3 は接するデジタルピンが 9~2 であればよく、続いて 86Duino Coding IDE を開き、下記のコードを入力する：

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

void setup()
{
  // Pin IO モジュール設定
  //行の pin を用いてキー電圧状態
for(int col = 0; col < Cols; col++)
  {
pinMode(col_pins[col], INPUT_PULLUP);
}
  //列の pin 当電圧源を用いる
for(int row = 0; row < Rows; row++)// 行をスキャン
  {
pinMode(row_pins[row], OUTPUT);
digitalWrite(row_pins[row], HIGH);
  }

Serial.begin(115200);
}


void loop()
{
  for(int row = 0; row < Rows; row++)// 列をスキャン
  {
digitalWrite(row_pins[row], LOW);// この列の電圧を LOW に変更
for(int col = 0; col < Cols; col++)// 行をスキャン
    {
// この業の電圧を読み取る・もしキーが電通を押していたら・電圧を LOW とする
boolean result = !digitalRead(col_pins[col]);

// このキーを入力し・先のものを入力する・これはすなわちキーの持続電圧状態である
if(result == HIGH &&keys_status_last[row][col] == true)
      {
```

```
        Serial.print("Button ");
        Serial.print(keys[row][col]);
        Serial.println(" hold");
                }
        // ここで入力し・さきでは入力しない・これは先に入力された状態である
        else if(result == HIGH &&keys_status_last[row][col] == false)
                {
        Serial.print("Button ");
        Serial.print(keys[row][col]);
        Serial.println(" pressed");
                }
        // ここでキーは入力しない・先に入力したものが・すなわちキーが開かれたばかりの
状態である
        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;// キー状態更新
              }
        digitalWrite(row_pins[row], HIGH);// この列の電圧を HIGH に変更
         }

          delay(20);
             }
```

コンパイル並びにアップローダーの後、読者に Serial Monitor を開いてもらい、試しにキーボードのキーを入力すると、Serial Monitor において入力したキーが対応する符合が、下図の様に表示される。

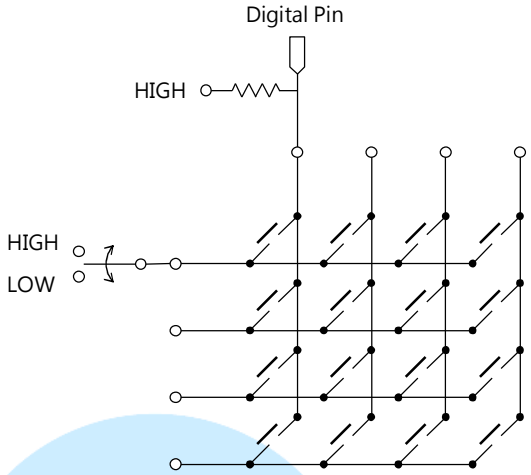


プログラミング開始し、まず、マトリックスキーボードの行数 Cols と列数 Rows, char マトリックスは keys[Rows][Cols]キーをする文字符号を保存する時に用いられ、bool マトリックスは keys_status_last[Rows][Cols]を各キーのひとつ前の状態を保存する時用いられる。入力には「true」を、入力しないには「false」を用いる。

ピンの数マトリックスはキーの行列が実際に対応するピン数をアナウンスすることで、後に続くアクセスを容易にする。例えば、文法 row_pins[1]を使用すれば、列 1 のピン数を取得することが可能である。

setup()段階で各デジタルピンの I/O モジュールの設定を行う。ここでは列 pin を用いて電圧ソースとし、行 pin を電圧状態読み取りに使用する。その他、Serial Port の初期化を実行する。

デジタル出力の部分に INPUT_PULLUP モジュールを使用し、列をスキャンする際 LOW の電圧を加え（その他の列は HIGH）ることに注意する必要がある。なぜなら、キーが入力された際 LOW が読み取られ、入力されないときは全て HIGH となるからである。この様にしてキー電圧状態の読み取りを更に安定したものにすることが可能である。電気回路図のディスプレイは下図の通り。



loop( )ループにおいて、二つの for ループを使用し、第一番目のものは、列をスキャンする際に用い、第二番目のものを行読み取りに用いる。二つのループはキーに対し電圧の読み取りを行って、前述したスキャン原理を達成する。

キー入力した際の電圧を LOW としたことで、コード行が「boolean result = !digitalRead(col_pins[col]);中にリバース動作が起こるので、result は true の際キーが入力され、再び result の結果を今回読み取ったキーの状態とみなす。

ここで注意しなくてはならないのは、keys_status_last マトリックスは各キーのひとつ前の状態を記録することに用いられる。なぜなら、キーはたった今押されたもの（前回は false、今回は true）を知ることができ、持続して押し続け（前回は true，今回は true）、或いはオープンされたもの（前回は true、今回は false）で、使用者が必要とする機能が完全に実現されているかをみる。この様に容易にキーボードのスキャン原理を練習し、もし読者が異なる大きさのキーボードや文字記号を使用するのであれば、先のいくつかの変数アナウンスを修正すれば、問題ない。

三、　第二のプログラミング-Keypad ライブラリ使用

86Duino EduCake とマトリックスキーボードスキャンの基本原理を理解したら、続いて小さな変更を行おう。このパラダイムは変更の必要はなく、使用とパラダイムは同じ配線である。

キーボードの機械構造が異なる為、入力するキーとボタンを離した時、実際は「バウンス」が発生する可能性があり、接点は、接触と未接触の状態において高速で変化する。もし、処理を終わらせていないと、ボタンを押したとき、電圧が上下した後、ようやく安定するのが発見できるだろう。

バウンス（またの名を debounce）する方法には回路タイプとプログラミングの処理方式が存在するが、上記の第一プログラミングはバウンスの処理をしていないという問題がある。主に loop 間隔時間が長すぎ（20ms）る為、デジタルピンを読み取る感覚がバウンスの変動時間を超過してしまい、その為、ボタン電圧を読み取るとき大きな変動は起こらない。もし、読者がキーの状態を高速読み取りしたい場合にしようすると、あまり適合はしない。

まあ、既に規制の Keypad ライブラリが使用でき、内部は既にバウンスとキーボード状態等の検索をスキャンする機能の処理が済んでいるので、このパラグラムで、私たちはこのライブラリを使用してマトリックスキーボードの読み取りの練習を行ってみよう。

読者は先にアクセスして欲しい：

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

下記に掲載した Keypad コードアーカイブの後、解凍した「Keypad」データを 86Duino IDE がフォルダパスした「86Duino_Coding_バージョン番号 WIN¥libraries」に入れる。

もう一度テキストエディタを用いて Keypad.h アーカイブを開き、「#include ″WProgram.h″」を探して「#include <Arduino.h>」に修正する。

その後「Keypad/utility フォルダ」内の Key.h アーカイブに対し、同じ修正動作を行う。完成後、86Duino Coding IDE, を開き、以下のコードを入力する：

```
#include <Keypad.h>

const byte Rows = 4;// 行数
const byte Cols = 4;// 列数
// 対応符号入力
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
{'*','0','#','D'}
};
// ピンコード定義
byte row_pins[Rows] = {9, 8, 7, 6};// 列 0~3
byte col_pins[Cols] = {5, 4, 3, 2};// 行 0~3

// Keypad lib オブジェクト
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins, col_pins, Rows, Cols );
void setup(){
 Serial.begin(115200);
}

 void loop(){

   if( keypad4X4.getKeys() )
   {
      for(inti = 0; i< LIST_MAX; i++)//keypad4X4 オブジェクトのキー列表を逐一検査
      {
        if( keypad4X4.key[i].stateChanged )// キー状態が変動していないか検査し、もし変動していた場合は再度内容を読み取る
        {
 Serial.print("Button ");
 Serial.print(keypad4X4.key[i].kchar);//目下入力された文字符号
          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);
```

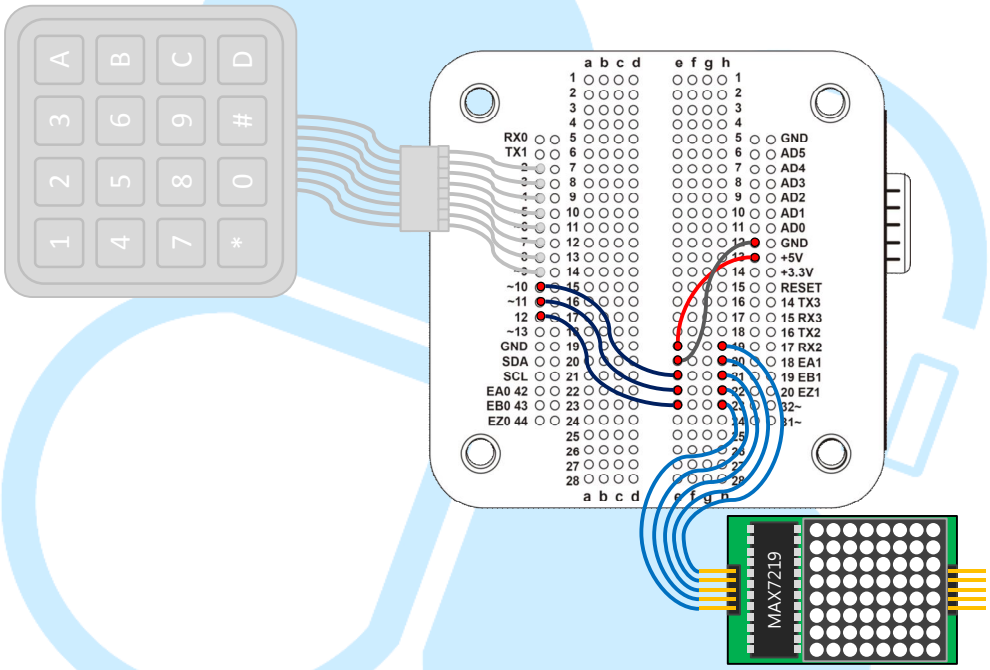
コンパイル及びアップローダー後、同様に Serial Monitor を開き、入力したキーボードのキーを試すと、Serial Monitor 内に入力されたキーに対応する符合とキーの状態プログラミングの執行結果とパラダイム類例が表示される。これはつまり Keypad ライブラリの便利なところであり、全てのマトリックスキーボードの処理プログラミング、変数定義を Keypad の class の中に包み込み、.ino アーカイブ内のプログラミング行数を有効的に削減し、同時に異なるプロジェクトプログラミング内に置いて重複使用される Keypad ライブラリを容易とする。続いて setup()ステージにおいて、Serial Port の初期化のみ行う。loop()ループ内で「keypad4X4.getKeys()」を使用しキーボード状態の読み取りと更新を行い、続いて「for(inti = 0; i< LIST_MAX; i++)」語法を用いて keypad4X4 フォルダのキー列表を逐一検査し、各キーのトリガー状態を判断する。

Keypad ライブラリ提供：「keypad4X4.key[i].stateChanged」はキー状態が変更されていないか検査することに用いられ、「keypad4X4.key[i].kchar」はそのキーが対応する文字符号を読み取ることに用いられ、「keypad4X4.key[i].kstate」はそのキー状態等を読み取る際に用いられる。キー状態は PRESSED（入力される）、HOLD（キーを押した状態持続）、RELEASED（離す）、IDLE（押されていない）を共有し、loop()ループ内ではこのいくつかの関数を変数を用いてキーの状態を判断し、データを出力する。Keypad ライブラリ使用後、コードを見てみると、かなり整理されたことがお分かりだろうか？

四、　第三のプログラミング-Keypad ライブラリ+8X8LED マトリックスモジュール使用

続いてこのプログラム例は Keypad ライブラリを継続使用し、私たちはその他の比較的複雑な機能を引き続き加えていくことにしよう。

このプログラミング例を用いる前に、既に紹介した MAX7219+8x8 LED マトリックスモジュールを、読者は下図を参照して、LED マトリックスの配線を加えて頂きたい：



MAX7219+8x8 LED マトリックスモジュールの機能の前に既に述べたので、重複利用の機会は大変多く、ここでは相関するコードを LEDmat8 に収めるため、まずいくつかのステップを踏む必要がある。

1. 86Duino IDE が存在するフォルダパス「86Duino_Coding_バージョンコード_WIN¥libraries」内に、新たに「LEDmat8」データを加える。

2. 「LEDmat8」フォルダ内にテキストファイルを一つ増やし、名称を「LEDmat8.h」に変更し（副ファイルの名称も共に変更することに注意すること）、その後フォルダ内に以下のコードを加え、保存する：

```
#ifndef LEDMAT8_H
#define LEDMAT8_H

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

    // MAX7219 スクラッチパッドの定義
#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:
    intDIN_pin;
    intLOAD_pin;
    intCLOCK_pin;
};
```

3. 「LEDmat8」フォルダ内にテキストファイルを新に作り、名称を「LEDmat8.cpp」に変更（副ファイルの名称も同時に変更することに注意）し、アーカイブ内に以下のコーディングを入力し、保存する：

```
#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);

  // 初期化した MAX7219 のスクラッチパッド
MAX7219_1Unit(max7219_REG_scanLimit, 0x07);// スキャンの設定
  MAX7219_1Unit(max7219_REG_decodeMode,  0x00);// 複合モード不使用
  MAX7219_1Unit(max7219_REG_shutdown, 0x01);// オフモード設定せず
  MAX7219_1Unit(max7219_REG_displayTest, 0x00); // テキストモードに設定せず

  for(inti=1; i<=8; i++) {// まず全ての LED マトリックスを暗くする
    MAX7219_1Unit(i,0);
  }

  MAX7219_1Unit(max7219_REG_intensity,  0x0f);// 明るさ範囲設定・0x00 ~ 0x0f

}

void LEDmat8::DrawLED(byte *LED_matrix)//全ての書面ドロー
```

```
    byte i = 8;
    byte mask;

    while(i> 0)
  {
      mask = (0x01 << (i - 1));// ビットマスク製造，一番左より開始
  digitalWrite(CLOCK_pin, LOW);// 周波数同期ライン = LOW
      if (data & mask){// ビットマスク対応の位が 0 か 1 か判断
  digitalWrite(DIN_pin, HIGH);// 対応位置が 1 の場合，DIN が HIGH
を送出
      }
      else{
  digitalWrite(DIN_pin, LOW);// もし対応位置が 0 の場合，DIN が
LOW を送出
      }
  digitalWrite(CLOCK_pin, HIGH);// 周波数同期ライン = HIGH
  i=i-1;// 一つ下の位へ移動
    }
  }

  void LEDmat8::MAX7219_1Unit(byte reg_addr, byte reg_data)//
MAX7219 モジュール制御
  {
  digitalWrite(LOAD_pin, LOW);// LOAD ピンが LOW へ変更を送信
  SPI_SendByte(reg_addr);//先に設定のスクラッチパッド位置を送信
  SPI_SendByte(reg_data);// 続いてデータ送信
  digitalWrite(LOAD_pin,HIGH);// LOAD ピン HIGH へ変更のデータ送
信完了
  }
```

86Duino Coding IDE を開き，下記のコード入力：

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

  const byte Rows = 4;// 行数
  const byte Cols = 4;// れっすう
  // 対応記号入力
  char keys[Rows][Cols] =
  {
    {'1','2','3','A'},
```

```
      {'*','0','#','D'}
      };
      //  ピン番号定義
      byte row_pins[Rows] = {9, 8, 7, 6};//  列  0~3
      byte col_pins[Cols] = {5, 4, 3, 2};//  行  0~3

      // Keypad lib  フォルダ
      Keypad  keypad4X4  =  Keypad(  makeKeymap(keys),  row_pins,
col_pins, Rows, Cols );

      // LED  ピンモジュールがピン定義制御
      intDIN_pin = 10;
      intLOAD_pin = 11;
      intCLOCK_pin = 12;

      // 8X8 LED マトリックスフォルダ
      LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin, CLOCK_pin );

      byte LED_Data_8X8[8] = {//  パターンデータ行列
        B00000000,//  左->右  =第 1 行下から上
        B00000000,
        B00000000,
        B00000000,
        B00000000,
        B00000000,
        B00000000,
        B00000000 //  左->右  =  第 8 行下から上
      };

      void ClearLED_Data()// LED 画面データクリア
      {
        for(inti = 0; i< 8; i++)
        {
          LED_Data_8X8[i] = B00000000;
        }
      }

      void setup ( ) {
      LedMatrix.Init();

            delay(1000);
    }
```

```
void loop ( ) {
ClearLED_Data();// LED データ行列更新の前に・データクリア
  /キー状態検査
  keypad4X4.getKeys( );//  キー状態更新

     // keypad4X4 フォルダのキー列表をチェック
for( inti = 0; i< LIST_MAX; i++ )
{
       //  キー状態が入力されているか検査
if( keypad4X4.key[i].kstate == PRESSED )// HOLD
{
          //異なるキー符合新しい描画データ更新に対し、'D'は左上角に対応し、'1'は右下角に対応する

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

        // LED 図案ドロー
      LedMatrix.DrawLED( LED_Data_8X8 );
        delay( 50 );
          }
```

コンパイル並びにアップローダー後、読者はマトリックスキーボードのキーを入力可能であり、LED マトリックスが対応位置のライトを点灯させる。

このプログラミング内に Keypad ライブラリと先に挙げた 8x8 LED マトリックスに機能を結合し、先のマトリックスコードを LEDmat8 ライブラリ内に入れる。プログラム前方の変数アナウンスとパラダイム二は一緒だが、いくつかの LED マトリックスの変数が、byte マトリックス「LED_Data_8X8[8]」の様に増加し、LED ディスプレイの資料とみなされる。

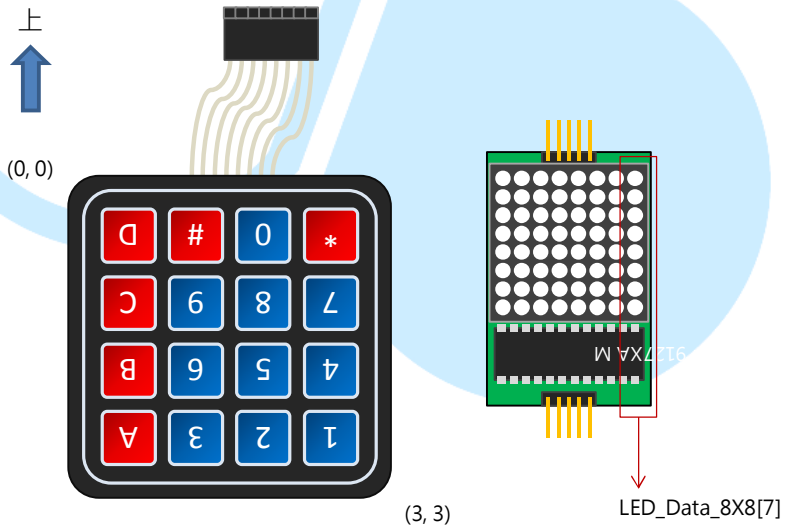
このプログラム内に Keypad ライブラリと先に挙げた 8x8 LED マトリックスディスプレイの機能を結合し、先の「ClearLED_Data()」関数を空にした LED_Data_8X8 のデータに用いる。「LEDmat8 LedMatrix = LEDmat8(DIN_pin, LOAD_pin, CLOCK_pin);」は LEDmat8 class のフォルダとし、全ての LED マトリックス制御の相関機能をその内部に入れる。

setup()段階を「LedMatrix.Init()」と言い、LED マトリックス制御機能を初期化する。loop()ループは毎回 ClearLED_Data()と呼ばれ、LED マトリックス資料更新の前に、その内部のデータを空にする。

続いて同様に「keypad4X4.getKeys();」を使用してキーボードの状態を更新し、「for(inti = 0; i< LIST_MAX; i++)」、「keypad4X4.key[i].kstate」を搭載し各キーの状態を取得する。

switch case 内は入力されたキーに対し、LED_Data_8X8 の内容を変更する。例えば、プレスは右下のキー「1」に位置しており、すなわちそれは「LED_Data_8X8[7] |= B11000000;」、「LED_Data_8X8[6] |= B11000000;」が LED_Data_8X8 の右下角四個の点を点灯させる。

ここで注意する必要があるのは、キーが対応する LED 位置は配線時のモジュールの配置方向に依拠し、もし読者のマトリックスキーボード、LED マトリックスの配置方向が相対する方向が異なる場合には、コードを修正し両者の位置定義を符合させる必要がある。本章では配置方向の定義を下図の通りとする：



LED のディスプレイ内容のデータ修正後、最後に「LedMatrix.DrawLED(LED_Data_8X8);」を使用して LED の図案をドローする。

読者は「if(keypad4X4.key[i].kstate == PRESSED)というこの条件をその他のキー状態に試しに変更して、LED マトリックス発行の時間変化を観察してみることもできる。

五、　第四のプログラミング—Keypad ライブラリ+8X8LED マトリックスモジュールを使用し、ホリネズミゲームを行う

最後にこのプログラム例で例 3 を修正し、ホリネズミゲームで遊ぶ機能を実作してみよう。電気回路図の配線は変動する必要はなく、読者は 86Duino Coding IDE を開き、以下のコーディングを入力してほしい：

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

const byte Rows = 4; // 行数
const byte Cols = 4; // 列数
// キー対応符号
char keys[Rows][Cols] =
{
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
// ピンコード定義
byte row_pins[Rows] = {9, 8, 7, 6}; // 列 0~3
byte col_pins[Cols] = {5, 4, 3, 2}; // 行 0~3

// Keypad lib フォルダ
Keypad keypad4X4 = Keypad( makeKeymap(keys), row_pins, col_pins, Rows, Cols );


// LED モジュール制御ピン定義
intDIN_pin = 10;
intLOAD_pin = 11;
intCLOCK_pin = 12;

// 8X8 LED マトリックスフォルダ
LEDmat8 LedMatrix = LEDmat8( DIN_pin, LOAD_pin, CLOCK_pin );

// ゲームデータ
int score = 0; // ゲーム特典
long gameTime = 0; // ゲーム進行時間
```

```
    booleanrunGame = false; // ゲーム執行状態であるか
    intloopCount = 0; // ホリネズミ地図を何回かのループ後更新することをランダムに決定
    #define DELAY_TIME        50 // loop 間隔時間
    #define LOOPCOUNT_MAX   30 //ホリネズミ地図を何回かのループ後再生後ランダムに決定、実際間隔
    時間(ms) = DELAY_TIME * LOOPCOUNT_MAX

    #define GAME_TIME         30 // ゲーム進行時間長さ
    #define MOLE_NUM_MAX      6 // 一回に出現する最大ホリネズミ数

    byte LED_Data_8X8[8] = { // パターンデータ
      B00000000, // 左->右 = 第 1 行下から上へ
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000,
      B00000000   // 左->右 = 第 8 行下から上へ
    };

    booleanMole_Data[4][4] = { // ホリネズミ地図 [列][行]
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0}
    };

    booleanKey_Data[4][4] = { // キー地図 [列][行]
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0},
      {0,0,0,0}
    };

    void ClearLED_Data() // LED 画面データクリア
    {
      for( inti = 0; i< 8; i++ )
      {
        LED_Data_8X8[i] = B00000000;
```

```
    }
      }
void ClearMoleData( ) { //  ホリネズミ地図データ空に
   for(inti = 0; i< 4; i++) {
      for(int j = 0; j < 4; j++) {
Mole_Data[i][j] = false;
      }
   }
}

void ClearKeyData( ) { //  キー地図データ空に
   for( inti = 0; i< 4; i++ ) {
      for( int j = 0; j < 4; j++ ) {
Key_Data[i][j] = false;
      }
   }
}

void GameStart ( ) { //  ゲーム開始・ゲームデータ初期化
ClearMoleData( ); //  ホリネズミ地図データクリア
ClearKeyData( ); //  キー地図データクリア
runGame = true; //ゲームの状態変数執行設定
   score = 0; //  ゲームゼロをスコアリング
gameTime = millis(); //  ゲーム進行時間リセット
}

void GameEnd ( ) {//ゲーム停止、データ出力
Serial.println("-----------------------------------------");
Serial.println("Game end!");
Serial.print(" Total Score : ");Serial.println(score);
Serial.println("     - Press 'S' or 'R' to play again.");
Serial.println("-----------------------------------------");

   //  ゲーム終了記号
   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) );// 乱数発生器初期化

    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;
    }

      // COM PORT チェック、データ入力
    if(Serial.available()){
        char ch = Serial.read();
        if( ch == 's' || ch == 'S' ) { // S 入力すれば、ゲーム開始
    Serial.println("-----------------------------------------------");
    Serial.println("Game is started!");
    Serial.println("-----------------------------------------------");
    GameStart();
        }
        else if( ch == 'r' || ch == 'R' ) { // 入力すれば、ゲームリセット
    Serial.println("-----------------------------------------------");
        Serial.println("Game is reset!");
```

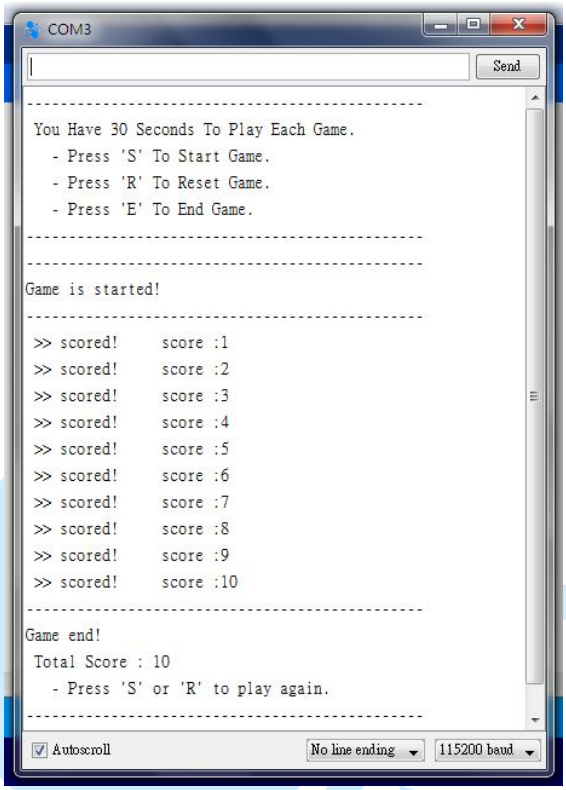
```
      Serial.println("-------------------------------------------");
      GameStart();
            }
            else if( ch == 'e' || ch == 'E' ) { //E 入力すればげ 0 無停止
      GameEnd();
            }
          }

          // ゲーム進行時間チェック
          if( runGame ){
            long time = millis() - gameTime; // ゲーム進行済時間
            if( time < GAME_TIME*1000 ) { // ゲーム進行中
              // ランダムに位置、時間設定しホリネズミ地図作成
              if( loopCount == LOOPCOUNT_MAX ) { // loopCount が最大
値到達後ホリネズミ地図再作成 ClearMoleData();
                long mole_num = random( 0, MOLE_NUM_MAX+1 );// 一
度に出現する最大ホリネズミ数
                for( inti = 0; i<mole_num; i++ ) {
                  long i_num = random(0,4); // [0,4) 間のランダム数字作成
                  long j_num = random(0,4); // [0,4) 間のランダム数字作成
                  //Serial.print(i_num);                    Serial.print(",");
Serial.println(j_num);
      Mole_Data[i_num][j_num] = true;
                }
              }

              // キー状態読み取り、キー地図と折りネズミ地図に対し、分数
計算
      ClearLED_Data( ); // LED マトリックスデータ更新前に・データクリア
      ClearKeyData( ); // キー地図データ更新前に・データクリア
      keypad4X4.getKeys( ); // キーボード状態更新

              for( inti = 0; i< LIST_MAX; i++ )// keypad4X4 フォルダのキー
列表を逐次検査
      {
                // キーが押されているか状態検査
      if( keypad4X4.key[i].kstate == PRESSED )
                {
      // 異なるキー符合がドローデータを更新することに対して、'D'は左上
角に対応し、'1'は右下角に対応する
      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
            // キーとホリネズミの地図検査
    for( inti = 0; i< 4; i++ ) { //  列
              for( int j = 0; j < 4; j++ ) { //  行
    // 入力した場所がホリネズミ出現の位置に一致すれば、ホリネズミ削除、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);
                }
              }
                }
```

```
        //削除後のホリネズミ地図に基づいて画面ドロー
                // 4x4 の地図データを 8x8 LED マトリックスが必要とするデ
ータに拡張
      for( inti = 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 );// LED 図案ドロー
          }
          else{// 時間到達・ゲーム終了
      GameEnd( );
      LedMatrix.DrawLED( LED_Data_8X8 );// LED 図案ドロー
          }
        }

        delay( DELAY_TIME );
          }
```

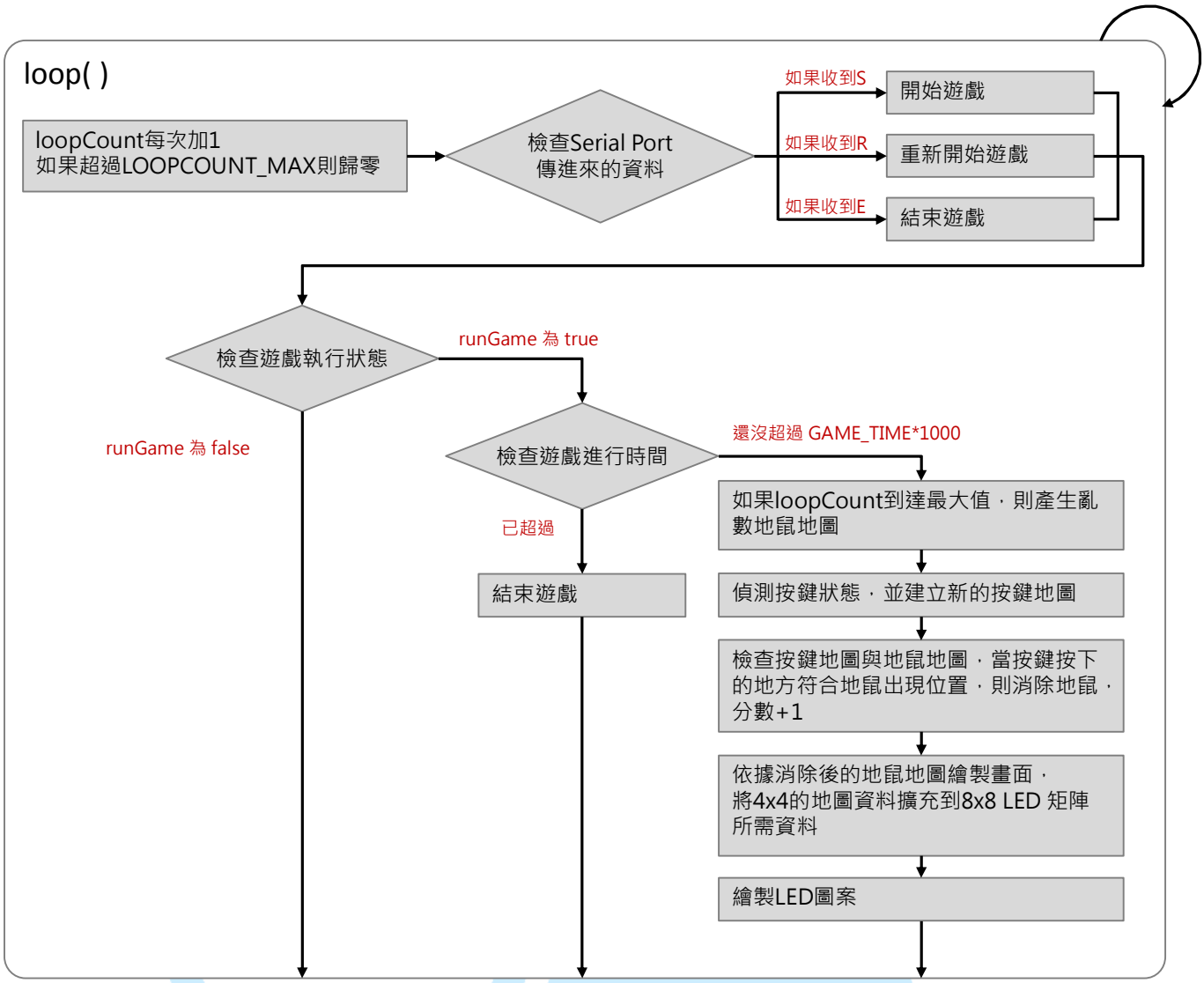
コーディング及びアップロード後、Serial Monitor を開き、上部にどのようにゲームのデータが進行開始されたかが表示され、読者は Serial Monitor を用いてアルファベット「S」或いは「R」を送信するとゲームが開始する。ゲーム時 LED マトリックスはランダムに点灯し（ホリネズミの場所を表）、キーボードの対応位置のキーを入力するとそのホリネズミを削除し、得点が増加する。毎回 30 秒で、時間に到達するとゲーム停止し、得点が表示され、LED マトリックスにスマイルデザインが出現する。Serial Monito g が受け取るデータは下記の図の通り：



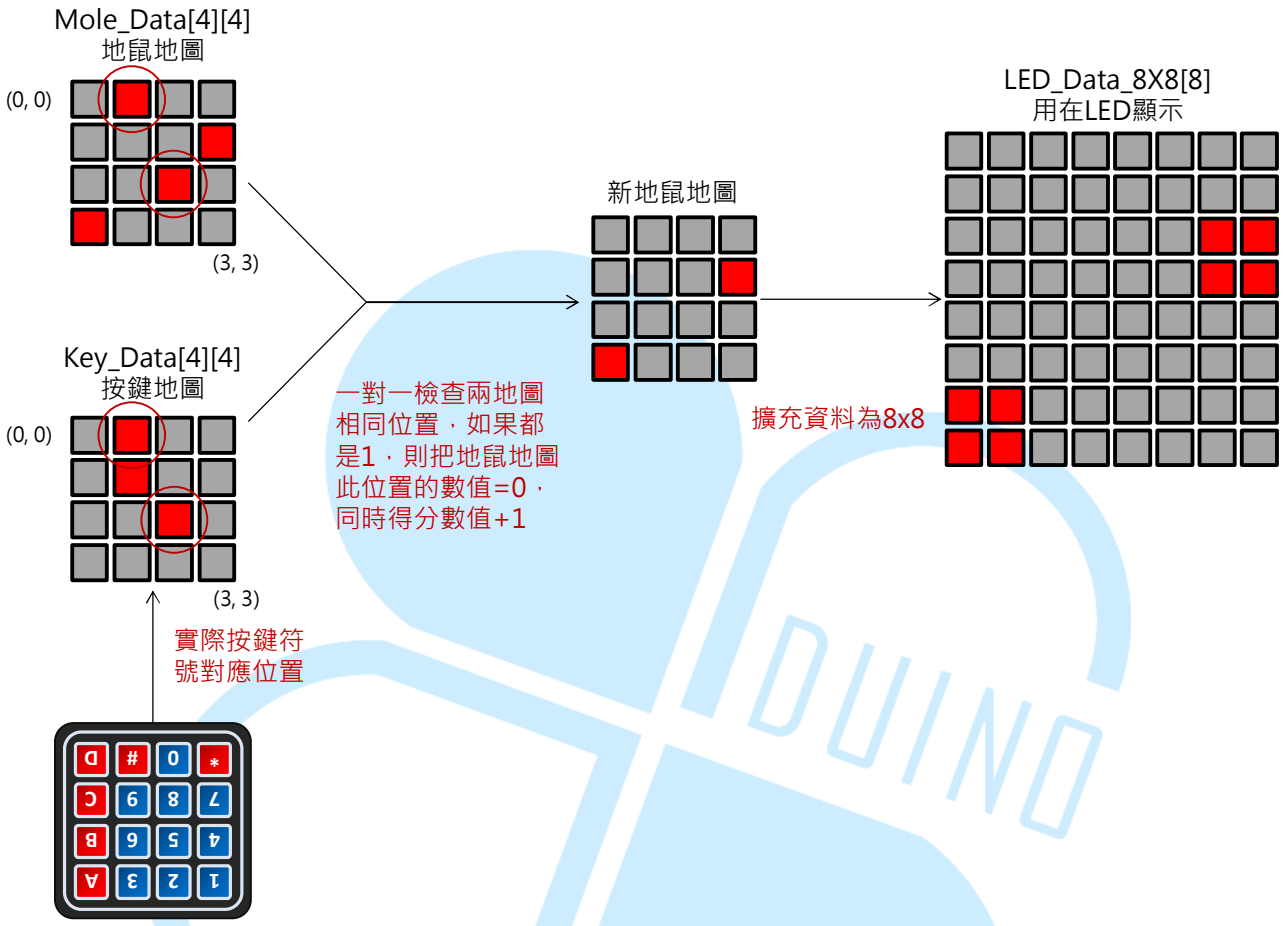
このプログラム例にゲーム執行時の変数を加える。たとえば、「score」はゲーム得点、「gameTime」はゲーム進行時間、「runGame」はゲームの執行状態を判断することに用いられ、「loopCount」はランダムなホリネズミ地図が幾度かのループ後更新されるなどである。その他、「#define DELAY_TIME」はloop 間隔時間を設定、「#define LOOPCOUNT_MAX」はランダムなホリネズミ地図が幾度かのループ後されることを決定し、「#define GAME_TIME」はゲーム進行時間の長さを設定、「#define MOLE_NUM_MAX」は一度に出現する最大ホリネズミ量を決定する等がある。関数は「ClearMoleData()」、「ClearKeyData()」、「GameStart ()」、「GameEnd ()」等を加え、ゲームプロセスを処理する。

loop()内的流程較複雜，讀者請參考下面概略流程圖：

setup()ステージと前回の例は類似しているので、注意が必要であり、このプログラムは乱数を生じるので、「randomSeed(analogRead(0));」語法を使用し、乱数発生器の初期化として、パラメータ欄の「analogRead(0)」をコネクションレスピンとして使用し、乱数発生器が混乱しないようにする。


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

地図とホリネズミ地図の方式を下図の様に入力し検査：



ランダムにホリネズミ地図を発生させる方法として、ここで採用するのはまずランダムに発生するホリネズミ数をNとし、ランダム発生するNの範囲を0~3の行数、列数とし、これらの位置を1に変更するが（出現するホリネズミを表示）、当然、位置は重複する可能性が有るる。最も多いのは、N匹のホリネズミが一度に出現することである。語法「long num = random(min, max)」はnumの変数値を「大をmin~小をmax」の間のランダム数字とする。

上述したプロセスとマトリックスキーボード、LED、マトリックスモジュールを組合せたので、容易にハリネズミゲームで遊ぶことができる。読者はこの概念その他のI/O機能、例えばキーを押すとゲーム開始、RCサーボモーターがホリネズミを昇降させる、7段のディスプレイがゲーム進行状態と分数を表す、

ホリネズミを打ったとき効果音発生等を加えることができるなど、とても大きな発揮空間がある。

ゲームの難易度は LOOPCOUNT_MAX（ホリネズミ地図の変数速度を調整）、MOLE_NUM_MAX(ホリネズミが一度に出現する最大数量を調整)の数値、gameTime（毎回の時間の長さを調整）を調整することで、異なる難易度レベルにすることができるので、ここで読者自身に改造してもらいたいと思う。