```
// (c) 2011 Braden Licastro; FullForce Applications. All rights reserved.
//
// RGB Combination Doorlatch w. Idle Effects
//    Version 2.5; Build 87;
//    Nightly 20111115
//
// Color code ordering:
//    [r][g][b]
//
// Description:
//    On button press, the color is toggled, starting from blank. The password is entered
//    by pressing the buttons to set the color code.
//    When the code is entered, the pad will turn green and unlock for 10 seconds.
//    If the counter is exceeded, the pad will turn red for 5 seconds.
//    When the device is idle it will flash through a looping animation until input is detected.
//
// Credits:
//    Written by Braden Licastro. Based on code by Will O' Brien, and various others works.
//
// Modifications:
//    Added first start visual debugging (runs on reset or first power on)
//    Optimized and reorganized code
//    Changed the way that the LEDs are managed and controlled
//    Code modified from 4x4 support to 2x4 support
//    Animations 1 and 2 preserved, added 3 new variants
//        1. One-by-one board wipe (single color)
//        2. Whole board lit (single color)
//        3. One-by-one wipe (random RGB color each key)
//        4. Whole board lit (random RGB color per run)
//        5. Whole board random color fade
//    Device goes into sleep mode after 'sleep_time' minutes.
//    Code modified to allow for color mixing instead of R, G, or B.
//    Method of fading LEDs with digital variable resistor implemented.
//
// Known Bugs:
//    Color_Cycle() does not properly cycle through the colors.
//        First row only cycles once, every row to the right adds one more color to the next cycle.
//    Does not wake Braden up for class... crap.


// START DECLARATIONS
#define DATAOUT 11 // MOSI (pin 7 of AD5206)
#define DATAIN 12 // MISO - not used, but part of builtin SPI.
#define SPICLOCK 13 // sck (pin 8 of AD5206)
#define SLAVESELECT 10 // Removed the slave switching code entirely.
#define COLS 4 // x axis
#define ROWS 2 // y axis
#define H 255 // pot high
#define L 64 // pot low
#define effect_select 5 // Choose the idle effect.
```

```
// MIN .5 minutes, MAX 8.0 minutes in .25 minute increments.
#define sleep_time 8 // Duration of time animation is allowed to run before device goes into power saving mode.


// LED CODE

// Pins for led column grounding transistors.
const byte colpin[COLS] = {
  14,15,16,17}; // Using the analog inputs as digital pins. (14=A0,15=A1,16=A2,17=A3)

// The pot register numbers for each of the red, green, and blue channels.
// Address map for AD5206:
// Pin bin dec
// 2   101 5
// 11  100 4
// 14  010 2
// 17  000 0
// 20  001 1
// 23  011 3
const byte red[2] = {
  5, 0};
const byte green[2] = {
  4, 1};
const byte blue[2] = {
  2, 3};

// The number of oclors that will be available.
byte COLORS = 4;

// Set up each of the colors that will be available.
byte rColors[] = {
  0, 255, 0,   0  }; // red
byte gColors[] = {
  0, 0,   255, 0  }; // green
byte bColors[] = {
  0, 0,   0,   255}; // blue

// Main data for the drawing routine.
byte rGrid[COLS][ROWS] = {
  0};
byte gGrid[COLS][ROWS] = {
  0};
byte bGrid[COLS][ROWS] = {
  0};

byte trajectory[COLS][ROWS] = {
  0};

// Map for the effects, is it right here?
boolean effect[COLS][ROWS] = {
  0};

```

```
// Idle Effects
byte effect_color = 3; // The color the idle effect should be.
byte effect_color_rand = 0; // Placeholder to hold the random colors, won't overwrite effect color.
byte effect_state = 0; // What state the effect is currently in.
byte effect_count = 0; // How fast the idle effect is activated/refreshed.
byte randomized = 0; // Monitors whether or not the grid was randomized once.
byte first_boot = 1; // On first boot the device will check doorlatch and LED circuits.
int cycles = 0; // Tracks the number of cycles before turning off the idle animation and going to sleep.
int timeout_cycles = sleep_time * 4020; // Convert minutes until hibernation into program cycles @ 67 cycles/second.


// Store elapsed time, used to manage animations.
unsigned long time;


// BUTTON CODE

// Pins for the Vin of the buttons, y-axis.
const byte buttonWrite[ROWS] = {
  2, 3};
// Pins for reading the state of the buttons, x-axis.
const byte buttonRead[COLS] = {
  9, 8, 7, 6};
boolean pressed[COLS][ROWS] = {
  0};

//  Entry code definition.
byte rCode[COLS][ROWS] = {
  0};
byte gCode[COLS][ROWS] = {
  0};
byte bCode[COLS][ROWS] = {
  0};
byte colorcode[COLS][ROWS] = {
  0 };
byte count;


// DOORLATCH CIRCUIT

// Set up the latch trigger and closed state.
const byte lock_pin = 4;
boolean lockState = 0;


// END DECLARATIONS, BEGIN PROGRAM


// SET UP EVERYTHING.
void setup(){
  randomSeed(1);

  // Initialize the counter.
  count = 0;

  // The entry code, right to left.
  // O-OFF; 1-RED; 2-GREEN; 3-BLUE
  //    ROW 1
  colorcode[3][0] = 3;
  colorcode[2][0] = 1;
  colorcode[1][0] = 3;
  colorcode[0][0] = 0;
  //    ROW 2
  colorcode[3][1] = 2;
  colorcode[2][1] = 0;
  colorcode[1][1] = 1;
  colorcode[0][1] = 1;

  // Start serial output.
  Serial.begin(19200);

  // Set up the latch circuit.
  pinMode(lock_pin, OUTPUT);
  digitalWrite(lock_pin, LOW);

  // Set up the button inputs and outputs.
  //    Set row lines to output and zero them.
  for(int i = 0; i < ROWS; ++i){
    pinMode(buttonWrite[i], OUTPUT);
    digitalWrite(buttonWrite[i],LOW);
  }
  //    Set column lines to input.
  for(int j = 0; j<COLS; ++j) {
    pinMode(buttonRead[j], INPUT);
  }

  byte i;
  byte clr;
  pinMode(DATAOUT, OUTPUT);
  pinMode(DATAIN, INPUT);
  pinMode(SPICLOCK,OUTPUT);
  pinMode(SLAVESELECT,OUTPUT);

  // Set up the LED display grid.
  //    Initialize rows and turn the mixer off until needed.
  for(byte c = 0; c < COLS; ++c){
    pinMode(colpin[c], OUTPUT);
    digitalWrite(colpin[c], LOW);
  }

  digitalWrite(SLAVESELECT,HIGH); //disable device

  // SPCR = 01010000
  // Interrupt disabled,spi enabled,msb 1st,master,clk low when idle,
  // Sample on leading edge of clk, system clock/4 (fastest).
  SPCR = (1<<SPE)|(1<<MSTR);
  clr=SPSR;
  clr=SPDR;
  delay(10);

  // Clear all of the pot registers
  for (i=0;i<6;i++)
  {
    write_pot(i,0);
  }

  // Zero the grid.
  grid_init();

  // Initialize the access code.
  code_init();

  // Initialize the effects
  effect_init();

  delay(10);

  // Milliseconds since applet start - Used to time animation sequence.
  time = millis();
}

// Running program (Infinite loop for hardware)
void loop(){

  // Hardware visual self test.
  if(first_boot==1){
    Serial.print("First boot, checking device:\n");
    Serial.print("Checking LEDs - RED...");
    // Color test: RED
    color_effect(255, 0, 0, 1);
    Serial.print("Done!\nChecking LEDs - GREEN...");
    // Color test: GREEN
    color_effect(0, 255, 0, 1);
    Serial.print("Done!\nChecking LEDs - BLUE...");
    // Color test: BLUE
    color_effect(0, 0, 255, 1);
    Serial.print("Done!\nTesting Door Latch - Override Started...\nDoor latch unarmed.\n");
    // Unlock the door.
    open_door();
    Serial.print("\n...Done!\n\nReturning to working state...Done!\n\n");
    // Clear entered code and return to scanning mode.
    grid_init();
    first_boot--;
  }

  // Serial output, gives a way to debug things.
  Serial.print(".");

  //Restrict the counters, they dont need to run when it is in sleep mode.
  if(cycles <= timeout_cycles){
    //Count up, keeps the effect from moving too fast.
    effect_count++;
    //Tracks the number of cycles the program has gone through since idle_animation started.
    cycles++;
  }
  else{
    //Go into sleep mode, timers are dead, shut off the LED matrix.
    grid_init();
  }

  // Check to see if you are guessing or messed up.
  // Make sure to change this with your password, sum of color codes +1.
  if(count > 12){
    // Debug
    Serial.print("Password incorrect. Lockout enabled.");
    // Color: red
    color_effect(255, 0, 0, 5);
    // Reset grid and key press counter.
    grid_init();
    count = 0;
  }

  for(byte r = 0; r < ROWS; ++r){
    // Bring button row high for reading presses.
    digitalWrite(buttonWrite[r], HIGH);
    // Clear pot regs between rows, otherwise the row not being read will be lit during other row reads.
    clear_pot();
    // Check and see if the password is correct.
    if(code_check()){
      count = 0;
      // Another silly debug statment.
      Serial.print("Code matched! Door latch unarmed.");
      // Unlock the door.
      open_door();
      // Clear entered code and return to scanning mode.
      grid_init();
    }

    for(byte c = 0; c < COLS; ++c){
      //Read the button presses
      if(pressed[c][r] != digitalRead(buttonRead[c])){
        pressed[c][r] = digitalRead(buttonRead[c]);
        if(pressed[c][r]){
          //This is the first button pressed since last reset.
          if(count == 0){
            // Clear the button states once a code entry has begun.
```

```
              grid_init();
              // Reset the random checker for effects that need it.
              randomized = 0;
              // Reset the cycle counter so idle effects come out of hibernation.
              cycles = 0;
            }
            // On button press, call on_press.
            on_press(c, r);
            // Count up for each button press encountered.
            count++;
          }
          else {
            // On release, call on_release.
            on_release(r, c);
          }
        }
        else {
          if(pressed[r][c]){
            // On button hold.
            while_pressed(c, r);
          }
          else {
            // On held button release.
            while_released(c,r);
          }
        }

        // There aren't any registered button presses and we aren't ready t ogo to sleep.
        if(count == 0 && cycles < timeout_cycles){
          //What effect is chosen? This determines what needs to run.
          if(effect_select==1 || effect_select==2 || effect_select==3 || effect_select==4){
            if(effect_count==50){
              // Initialize the idle effect.
              idle_effect();
              effect_count = 0;
            }
          }
          else{
            //Did we randomize the grid for effects that need it? Only do it once though!
            if(randomized==0){
              grid_rand();
              randomized++;
            }
            //Rapid fire LED triggering, for fading purposes.
            if(effect_count==1){
              // Initialize the idle effect.
              idle_effect();
              effect_count = 0;
            }
          }
        }

        // Writes state of colors to the digital pot register.
```

```
      write_pot(red[r],rGrid[c][r]);
      write_pot(green[r],gGrid[c][r]);
      write_pot(blue[r],bGrid[c][r]);

      // Turn one column on while pot is written.
      digitalWrite(colpin[c], HIGH);
      // Display. Persistance of Vision makes things appear lit constantly.
      delayMicroseconds(750);
      // Turn the column back off.
      digitalWrite(colpin[c], LOW);
    }

    // Bring current button row low.
    digitalWrite(buttonWrite[r], LOW);
    delay(4);
  }
}

// Initialize the button grids with blank data.
void grid_init(){
  for(byte c = 0; c < COLS; ++c){
    for(byte r = 0; r < ROWS; ++r){
      rGrid[c][r] = 0;
      gGrid[c][r] = 0;
      bGrid[c][r] = 0;
      trajectory[c][r] = random(1,8);
    }
  }
}

// Initialize the button grids with random data. (For random effects only)
void grid_rand(){
  for(byte x = 0; x < COLS; ++x){
    for(byte y = 0; y < ROWS; ++y){
      rGrid[x][y] = random(0,256);
      gGrid[x][y] = random(0,256);
      bGrid[x][y] = random(0,256);
      trajectory[x][y] = random(1,8);
    }
  }
}

// Initialize the LED grids with blank data.
void effect_init(){
  for(byte c = 0; c < COLS; ++c){
    for(byte r = 0; r < ROWS; ++r){
      effect[c][r] = 0;
    }
  }
  effect[0][0] = 1;
}

// Initialize the COLOR code.
```

```
void code_init(){
  // C, R where 'c' is column, 'r' is row.
  // Each is 0-255. Values are mixed to create colors.
  // The code consists for four colors to be displayed on one row.
  for(byte c=0; c < COLS; c++){
    for(byte r=0; r < ROWS; r++){
      rCode[c][r] = rColors[colorcode[c][r]]; // column 0, row 0
      gCode[c][r] = gColors[colorcode[c][r]];
      bCode[c][r] = bColors[colorcode[c][r]];
    }
  }
}

// Write to the potentiometer.
byte write_pot(byte address, byte value){
  digitalWrite(SLAVESELECT, LOW);
  // 2 byte opcode.
  spi_transfer(address % 6);
  spi_transfer(constrain(255-value,0,255));
  // Release chip, signal end transfer.
  digitalWrite(SLAVESELECT, HIGH);
}

// Clear all of the potentiometer registers.
void clear_pot(){
  byte i;
  for (i=0;i<6;i++)
  {
    write_pot(i,0);
  }
}

char spi_transfer(volatile char data)
{
  // Start the transmission.
  SPDR = data;
  // Wait the end of the transmission.
  while (!(SPSR & (1<<SPIF)))
  {
  };
  // Return the received byte
  return SPDR;
}

// Called whenever a button is pressed.
void on_press(byte c, byte r){
  Serial.print(c, DEC);
  Serial.print(", ");
  Serial.println(r, DEC);
  color_cycle(c, r);
}

// Called whenever a button is released.
```

```
void on_release(byte c, byte r){
}

// Called while a button is pressed.
void while_pressed(byte c, byte r){
}

// Called after a held button is released.
void while_released(byte c, byte r){
}

// Color mixing per row/location.
void rgb(byte c, byte r, byte R, byte G, byte B){
  rGrid[c][r] = R;
  gGrid[c][r] = G;
  bGrid[c][r] = B;
}

//Cycle through the colors (only one per press) every time the button is depressed D-: .
void color_cycle(byte c, byte r){
  byte color = get_color(c, r);
  Serial.print("got color");
  Serial.print( color, DEC );
  if(color < COLORS){
    color++;
  }
  else {
    // Skip the blank color. (0)
    color = 1;
  }
  rgb(c, r, rColors[color], gColors[color], bColors[color]);
}

// Get the color at the current location.
byte get_color(byte c, byte r){
  for(byte i=0; i < COLORS; i++){
    if(rGrid[c][r] == rColors[i]){
      if(bGrid[c][r] == bColors[i]){
        if(gGrid[c][r] == gColors[i]){
          return(i);
        }
      }
    }
  }
}

// Check color state of grid to see if it matches entry code.
boolean code_check(){
  for(byte c=0;c<COLS;c++){
    for(byte r=0; r < ROWS; r++){
      if(rGrid[c][r] != rCode[c][r]){
        return(0);
      }
      if(gGrid[c][r] != gCode[c][r]){
        return(0);
      }
      if(bGrid[c][r] != bCode[c][r]){
        return(0);
      }
    }
  }
  return(1);
}

// The entry code was apparently correct.
void open_door(){
  // Activate the lock solenoid.
  digitalWrite(lock_pin, HIGH);
  if(first_boot==1){
    //Just a test, unlock quickly!
    color_effect(255, 255, 255, 1);
  }
  else{
    // Change the keypad green for 5 seconds or soish.
    color_effect(0, 255, 0, 5);
  }
  // Deactivate the door solenoid again.
  digitalWrite(lock_pin, LOW);
  // More silly debug code.
  Serial.print("Door latch armed.");
}

// Change all of the keypad buttons to the provided color for n seconds and turn it off again.
void color_effect(byte dRed, byte dGreen, byte dBlue, byte time){
  byte c;
  // Leave the grid alone, just write the pot.
  for(byte r=0; r<ROWS; r++){
    write_pot(red[r],dRed);
    write_pot(green[r],dGreen);
    write_pot(blue[r],dBlue);
  }
  for(c=0;c<COLS;c++){
    // Turn one col on while pot is written.
    digitalWrite(colpin[c], HIGH);
  }
  // Turn time into secondsish.
  delay(time*1000);
  for(c=0;c<COLS;c++){
    // Turn the col back off.
    digitalWrite(colpin[c], LOW);
  }
  // Turn off everything in the pot.
  clear_pot();
}

```

```
// This is he eye candy while the keypad isn't in use.
void idle_effect(){
  //What effect was selected at the beginning?

  // Only one light on at a time, one predetermined color.
  if(effect_select==1){
    //The grid needs cleared for this one.
    grid_init();

    for(byte r=0; r < ROWS; r++){
      for(byte c=0;c < COLS;c++){
        if(effect_state == 1){
          effect[c][r] = 1;
          effect_state = 0;
          return;
        }
        if(effect[c][r]) {
          rGrid[c][r] = rColors[effect_color];
          gGrid[c][r] = gColors[effect_color];
          bGrid[c][r] = bColors[effect_color];
          effect[c][r] = 0;
          effect_state = 1;
        }
      }
    }
  }

  // Light the whole grid at once, one predetermined color.
  if(effect_select==2){
    //The grid needs cleared for this one.
    grid_init();

    byte red = rColors[effect_color];
    byte blue = bColors[effect_color];
    byte green = gColors[effect_color];
    if(red != 0)
      red = red - effect_state;

    for(byte r=0; r < ROWS; r++){
      for(byte c=0;c<COLS;c++){
        rGrid[c][r] = rColors[effect_color];
        gGrid[c][r] = gColors[effect_color];
        bGrid[c][r] = bColors[effect_color];
        effect[c][r] = 0;
        effect_state = 1;
      }
    }
  }

  // Only one light on at a time, random color.
  if(effect_select==3){
    //The grid needs cleared for this one.
    grid_init();
```

```

      for(byte r=0; r < ROWS; r++){
        for(byte c=0;c < COLS;c++){
          if(effect_state == 1){
            effect[c][r] = 1;
            effect_state = 0;
            return;
          }
          if(effect[c][r]) {
            //Select our random color: R, G, B
            effect_color_rand = random(3) + 1;

            rGrid[c][r] = rColors[effect_color_rand];
            gGrid[c][r] = gColors[effect_color_rand];
            bGrid[c][r] = bColors[effect_color_rand];

            effect[c][r] = 0;
            effect_state = 1;
          }
        }
      }
    }

    // Light the whole grid at once, random color.
    if(effect_select==4){
      //The grid needs cleared for this one.
      grid_init();

      //Select our random color: R, G, B
      effect_color_rand = random(3) + 1;

      byte red = rColors[effect_color_rand];
      byte blue = bColors[effect_color_rand];
      byte green = gColors[effect_color_rand];
      if(red != 0)
        red = red - effect_state;

      for(byte r=0; r < ROWS; r++){
        for(byte c=0;c<COLS;c++){
          rGrid[c][r] = rColors[effect_color_rand];
          gGrid[c][r] = gColors[effect_color_rand];
          bGrid[c][r] = bColors[effect_color_rand];

          effect[c][r] = 0;
          effect_state = 1;
        }
      }
    }

    // Light the whole grid at once, block, randomly mixed color, with fade.
    if(effect_select==5){
      if((long)millis() - (long)time > 10){
        time = millis();
      for(byte x = 0; x < COLS; ++x){
        for(byte y = 0; y < ROWS; ++y){
          rGrid[x][y] = constrain(rGrid[x][y] + ((trajectory[x][y] & B001 ) ? 1 : -1),L, H);
          gGrid[x][y] = constrain(gGrid[x][y] + ((trajectory[x][y] & B010 ) ? 1 : -1),L, H);
          bGrid[x][y] = constrain(bGrid[x][y] + ((trajectory[x][y] & B100 ) ? 1 : -1),L, H);
          if (rGrid[x][y] == ( (trajectory[x][y] & B001) ? H : L ) && gGrid[x][y] == ( (trajectory[x][y] & B010) ? H : L ) && bGrid[x][y] == ( (trajectory[x][y] & B100) ? H : L ) ) {
            trajectory[x][y] = random(1,8);
          }
        }
      }
    }
  }
}


```