```
1  // (c) 2011 Braden Licastro; FullForce Applications. All rights reserved.
2  //
3  // RGB Combination Doorlatch w. Idle Effects
4  //    Version 2.5; Build 87;
5  //    Nightly 20111115
6  //
7  // Color code ordering:
8  //    [r][g][b]
9  //
10 // Description:
11 //    On button press, the color is toggled, starting from blank. The password is entered
12 //    by pressing the buttons to set the color code.
13 //    When the code is entered, the pad will turn green and unlock for 10 seconds.
14 //    If the counter is exceeded, the pad will turn red for 5 seconds.
15 //    When the device is idle it will flash through a looping animation until input is
   detected.
16 //
17 // Credits:
18 //    Written by Braden Licastro. Based on code by Will O' Brien, and various others
   works.
19 //
20 // Modifications:
21 //    Added first start visual debugging (runs on reset or first power on)
22 //    Optimized and reorganized code
23 //    Changed the way that the LEDs are managed and controlled
24 //    Code modified from 4x4 support to 2x4 support
25 //    Animations 1 and 2 preserved, added 3 new variants
26 //        1. One-by-one board wipe (single color)
27 //        2. Whole board lit (single color)
28 //        3. One-by-one wipe (random RGB color each key)
29 //        4. Whole board lit (random RGB color per run)
30 //        5. Whole board random color fade
31 //    Device goes into sleep mode after 'sleep_time' minutes.
32 //    Code modified to allow for color mixing instead of R, G, or B.
33 //    Method of fading LEDs with digital variable resistor implemented.
34 //
35 // Known Bugs:
36 //    Color_Cycle() does not properly cycle through the colors.
37 //        First row only cycles once, every row to the right adds one more color to the
   next cycle.
38 //    Does not wake Braden up for class... crap.
39
40
41 // START DECLARATIONS
42 #define DATAOUT 11 // MOSI (pin 7 of AD5206)
43 #define DATAIN 12 // MISO - not used, but part of builtin SPI.
44 #define SPICLOCK 13 // sck (pin 8 of AD5206)
45 #define SLAVESELECT 10 // Removed the slave switching code entirely.
46 #define COLS 4 // x axis
47 #define ROWS 2 // y axis
48 #define H 255 // pot high
49 #define L 64 // pot low
50 #define effect_select 5 // Choose the idle effect.
```

```
51  // MIN .5 minutes, MAX 8.0 minutes in .25 minute increments.
52  #define sleep_time 8 // Duration of time animation is allowed to run before device goes
    into power saving mode.
53
54
55  // LED CODE
56
57  // Pins for led column grounding transistors.
58  const byte colpin[COLS] = {
59    14,15,16,17}; // Using the analog inputs as digital pins. (14=A0,15=A1,16=A2,17=A3)
60
61  // The pot register numbers for each of the red, green, and blue channels.
62  // Address map for AD5206:
63  // Pin bin dec
64  // 2   101 5
65  // 11  100 4
66  // 14  010 2
67  // 17  000 0
68  // 20  001 1
69  // 23  011 3
70  const byte red[2] = {
71    5, 0};
72  const byte green[2] = {
73    4, 1};
74  const byte blue[2] = {
75    2, 3};
76
77  // The number of oclors that will be available.
78  byte COLORS = 4;
79
80  // Set up each of the colors that will be available.
81  byte rColors[] = {
82    0, 255, 0,   0  }; // red
83  byte gColors[] = {
84    0, 0,   255, 0  }; // green
85  byte bColors[] = {
86    0, 0,   0,   255}; // blue
87
88  // Main data for the drawing routine.
89  byte rGrid[COLS][ROWS] = {
90    0};
91  byte gGrid[COLS][ROWS] = {
92    0};
93  byte bGrid[COLS][ROWS] = {
94    0};
95
96  byte trajectory[COLS][ROWS] = {
97    0};
98
99  // Map for the effects, is it right here?
100 boolean effect[COLS][ROWS] = {
101   0};
102
```

```
103  // Idle Effects
104  byte effect_color = 3; // The color the idle effect should be.
105  byte effect_color_rand = 0; // Placeholder to hold the random colors, won't overwrite
     effect color.
106  byte effect_state = 0; // What state the effect is currently in.
107  byte effect_count = 0; // How fast the idle effect is activated/refreshed.
108  byte randomized = 0; // Monitors whether or not the grid was randomized once.
109  byte first_boot = 1; // On first boot the device will check doorlatch and LED circuits.
110  int cycles = 0; // Tracks the number of cycles before turning off the idle animation
     and going to sleep.
111  int timeout_cycles = sleep_time * 4020; // Convert minutes until hibernation into
     program cycles @ 67 cycles/second.
112
113
114  // Store elapsed time, used to manage animations.
115  unsigned long time;
116
117
118  // BUTTON CODE
119
120  // Pins for the Vin of the buttons, y-axis.
121  const byte buttonWrite[ROWS] = {
122    2, 3};
123  // Pins for reading the state of the buttons, x-axis.
124  const byte buttonRead[COLS] = {
125    9, 8, 7, 6};
126  boolean pressed[COLS][ROWS] = {
127    0};
128
129  //  Entry code definition.
130  byte rCode[COLS][ROWS] = {
131    0};
132  byte gCode[COLS][ROWS] = {
133    0};
134  byte bCode[COLS][ROWS] = {
135    0};
136  byte colorcode[COLS][ROWS] = {
137    0 };
138  byte count;
139
140
141  // DOORLATCH CIRCUIT
142
143  // Set up the latch trigger and closed state.
144  const byte lock_pin = 4;
145  boolean lockState = 0;
146
147
148  // END DECLARATIONS, BEGIN PROGRAM
149
150
151  // SET UP EVERYTHING.
152  void setup(){
```

```
153     randomSeed(1);
154
155     // Initialize the counter.
156     count = 0;
157
158     // The entry code, right to left.
159     // O-OFF; 1-RED; 2-GREEN; 3-BLUE
160     //    ROW 1
161     colorcode[3][0] = 3;
162     colorcode[2][0] = 1;
163     colorcode[1][0] = 3;
164     colorcode[0][0] = 0;
165     //    ROW 2
166     colorcode[3][1] = 2;
167     colorcode[2][1] = 0;
168     colorcode[1][1] = 1;
169     colorcode[0][1] = 1;
170
171     // Start serial output.
172     Serial.begin(19200);
173
174     // Set up the latch circuit.
175     pinMode(lock_pin, OUTPUT);
176     digitalWrite(lock_pin, LOW);
177
178     // Set up the button inputs and outputs.
179     //    Set row lines to output and zero them.
180     for(int i = 0; i < ROWS; ++i){
181       pinMode(buttonWrite[i], OUTPUT);
182       digitalWrite(buttonWrite[i],LOW);
183     }
184     //    Set column lines to input.
185     for(int j = 0; j<COLS; ++j) {
186       pinMode(buttonRead[j], INPUT);
187     }
188
189     byte i;
190     byte clr;
191     pinMode(DATAOUT, OUTPUT);
192     pinMode(DATAIN, INPUT);
193     pinMode(SPICLOCK,OUTPUT);
194     pinMode(SLAVESELECT,OUTPUT);
195
196     // Set up the LED display grid.
197     //    Initialize rows and turn the mixer off until needed.
198     for(byte c = 0; c < COLS; ++c){
199       pinMode(colpin[c], OUTPUT);
200       digitalWrite(colpin[c], LOW);
201     }
202
203     digitalWrite(SLAVESELECT,HIGH); //disable device
204
205     // SPCR = 01010000
```

```
206     // Interrupt disabled,spi enabled,msb 1st,master,clk low when idle,
207     // Sample on leading edge of clk, system clock/4 (fastest).
208     SPCR = (1<<SPE)|(1<<MSTR);
209     clr=SPSR;
210     clr=SPDR;
211     delay(10);
212
213     // Clear all of the pot registers
214     for (i=0;i<6;i++)
215     {
216       write_pot(i,0);
217     }
218
219     // Zero the grid.
220     grid_init();
221
222     // Initialize the access code.
223     code_init();
224
225     // Initialize the effects
226     effect_init();
227
228     delay(10);
229
230     // Milliseconds since applet start - Used to time animation sequence.
231     time = millis();
232   }
233
234   // Running program (Infinite loop for hardware)
235   void loop(){
236
237     // Hardware visual self test.
238     if(first_boot==1){
239       Serial.print("First boot, checking device:\n");
240       Serial.print("Checking LEDs - RED...");
241       // Color test: RED
242       color_effect(255, 0, 0, 1);
243       Serial.print("Done!\nChecking LEDs - GREEN...");
244       // Color test: GREEN
245       color_effect(0, 255, 0, 1);
246       Serial.print("Done!\nChecking LEDs - BLUE...");
247       // Color test: BLUE
248       color_effect(0, 0, 255, 1);
249       Serial.print("Done!\nTesting Door Latch - Override Started...\nDoor latch
    unarmed.\n");
250       // Unlock the door.
251       open_door();
252       Serial.print("\n...Done!\n\nReturning to working state...Done!\n\n");
253       // Clear entered code and return to scanning mode.
254       grid_init();
255       first_boot--;
256     }
257
```

```
258     // Serial output, gives a way to debug things.
259     Serial.print(".");
260
261     //Restrict the counters, they dont need to run when it is in sleep mode.
262     if(cycles <= timeout_cycles){
263       //Count up, keeps the effect from moving too fast.
264       effect_count++;
265       //Tracks the number of cycles the program has gone through since idle_animation
    started.
266       cycles++;
267     }
268     else{
269       //Go into sleep mode, timers are dead, shut off the LED matrix.
270       grid_init();
271     }
272
273     // Check to see if you are guessing or messed up.
274     // Make sure to change this with your password, sum of color codes +1.
275     if(count > 12){
276       // Debug
277       Serial.print("Password incorrect. Lockout enabled.");
278       // Color: red
279       color_effect(255, 0, 0, 5);
280       // Reset grid and key press counter.
281       grid_init();
282       count = 0;
283     }
284
285     for(byte r = 0; r < ROWS; ++r){
286       // Bring button row high for reading presses.
287       digitalWrite(buttonWrite[r], HIGH);
288       // Clear pot regs between rows, otherwise the row not being read will be lit during
    other row reads.
289       clear_pot();
290       // Check and see if the password is correct.
291       if(code_check()){
292         count = 0;
293         // Another silly debug statment.
294         Serial.print("Code matched! Door latch unarmed.");
295         // Unlock the door.
296         open_door();
297         // Clear entered code and return to scanning mode.
298         grid_init();
299       }
300
301       for(byte c = 0; c < COLS; ++c){
302         //Read the button presses
303         if(pressed[c][r] != digitalRead(buttonRead[c])){
304           pressed[c][r] = digitalRead(buttonRead[c]);
305           if(pressed[c][r]){
306             //This is the first button pressed since last reset.
307             if(count == 0){
308               // Clear the button states once a code entry has begun.
```

```
309               grid_init();
310               // Reset the random checker for effects that need it.
311               randomized = 0;
312               // Reset the cycle counter so idle effects come out of hibernation.
313               cycles = 0;
314             }
315             // On button press, call on_press.
316             on_press(c, r);
317             // Count up for each button press encountered.
318             count++;
319           }
320           else {
321             // On release, call on_release.
322             on_release(r, c);
323           }
324         }
325         else {
326           if(pressed[r][c]){
327             // On button hold.
328             while_pressed(c, r);
329           }
330           else {
331             // On held button release.
332             while_released(c,r);
333           }
334         }
335
336         // There aren't any registered button presses and we aren't ready t ogo to sleep.
337         if(count == 0 && cycles < timeout_cycles){
338           //What effect is chosen? This determines what needs to run.
339           if(effect_select==1 || effect_select==2 || effect_select==3 || effect_select==4){
340             if(effect_count==50){
341               // Initialize the idle effect.
342               idle_effect();
343               effect_count = 0;
344             }
345           }
346           else{
347             //Did we randomize the grid for effects that need it? Only do it once though!
348             if(randomized==0){
349               grid_rand();
350               randomized++;
351             }
352             //Rapid fire LED triggering, for fading purposes.
353             if(effect_count==1){
354               // Initialize the idle effect.
355               idle_effect();
356               effect_count = 0;
357             }
358           }
359         }
360
361         // Writes state of colors to the digital pot register.
```

```
362         write_pot(red[r],rGrid[c][r]);
363         write_pot(green[r],gGrid[c][r]);
364         write_pot(blue[r],bGrid[c][r]);
365 
366         // Turn one column on while pot is written.
367         digitalWrite(colpin[c], HIGH);
368         // Display. Persistance of Vision makes things appear lit constantly.
369         delayMicroseconds(750);
370         // Turn the column back off.
371         digitalWrite(colpin[c], LOW);
372       }
373 
374       // Bring current button row low.
375       digitalWrite(buttonWrite[r], LOW);
376       delay(4);
377     }
378   }
379 
380   // Initialize the button grids with blank data.
381   void grid_init(){
382     for(byte c = 0; c < COLS; ++c){
383       for(byte r = 0; r < ROWS; ++r){
384         rGrid[c][r] = 0;
385         gGrid[c][r] = 0;
386         bGrid[c][r] = 0;
387         trajectory[c][r] = random(1,8);
388       }
389     }
390   }
391 
392   // Initialize the button grids with random data. (For random effects only)
393   void grid_rand(){
394     for(byte x = 0; x < COLS; ++x){
395       for(byte y = 0; y < ROWS; ++y){
396         rGrid[x][y] = random(0,256);
397         gGrid[x][y] = random(0,256);
398         bGrid[x][y] = random(0,256);
399         trajectory[x][y] = random(1,8);
400       }
401     }
402   }
403 
404   // Initialize the LED grids with blank data.
405   void effect_init(){
406     for(byte c = 0; c < COLS; ++c){
407       for(byte r = 0; r < ROWS; ++r){
408         effect[c][r] = 0;
409       }
410     }
411     effect[0][0] = 1;
412   }
413 
414   // Initialize the COLOR code.
```

```
415  void code_init(){
416    // C, R where 'c' is column, 'r' is row.
417    // Each is 0-255. Values are mixed to create colors.
418    // The code consists for four colors to be displayed on one row.
419    for(byte c=0; c < COLS; c++){
420      for(byte r=0; r < ROWS; r++){
421        rCode[c][r] = rColors[colorcode[c][r]]; // column 0, row 0
422        gCode[c][r] = gColors[colorcode[c][r]];
423        bCode[c][r] = bColors[colorcode[c][r]];
424      }
425    }
426  }
427
428  // Write to the potentiometer.
429  byte write_pot(byte address, byte value){
430    digitalWrite(SLAVESELECT, LOW);
431    // 2 byte opcode.
432    spi_transfer(address % 6);
433    spi_transfer(constrain(255-value,0,255));
434    // Release chip, signal end transfer.
435    digitalWrite(SLAVESELECT, HIGH);
436  }
437
438  // Clear all of the potentiometer registers.
439  void clear_pot(){
440    byte i;
441    for (i=0;i<6;i++)
442    {
443      write_pot(i,0);
444    }
445  }
446
447  char spi_transfer(volatile char data)
448  {
449    // Start the transmission.
450    SPDR = data;
451    // Wait the end of the transmission.
452    while (!(SPSR & (1<<SPIF)))
453    {
454    };
455    // Return the received byte
456    return SPDR;
457  }
458
459  // Called whenever a button is pressed.
460  void on_press(byte c, byte r){
461    Serial.print(c, DEC);
462    Serial.print(", ");
463    Serial.println(r, DEC);
464    color_cycle(c, r);
465  }
466
467  // Called whenever a button is released.
```

```
468  void on_release(byte c, byte r){
469  }
470
471  // Called while a button is pressed.
472  void while_pressed(byte c, byte r){
473  }
474
475  // Called after a held button is released.
476  void while_released(byte c, byte r){
477  }
478
479  // Color mixing per row/location.
480  void rgb(byte c, byte r, byte R, byte G, byte B){
481    rGrid[c][r] = R;
482    gGrid[c][r] = G;
483    bGrid[c][r] = B;
484  }
485
486  //Cycle through the colors (only one per press) every time the button is depressed D-: .
487  void color_cycle(byte c, byte r){
488    byte color = get_color(c, r);
489    Serial.print("got color");
490    Serial.print( color, DEC );
491    if(color < COLORS){
492      color++;
493    }
494    else {
495      // Skip the blank color. (0)
496      color = 1;
497    }
498    rgb(c, r, rColors[color], gColors[color], bColors[color]);
499  }
500
501  // Get the color at the current location.
502  byte get_color(byte c, byte r){
503    for(byte i=0; i < COLORS; i++){
504      if(rGrid[c][r] == rColors[i]){
505        if(bGrid[c][r] == bColors[i]){
506          if(gGrid[c][r] == gColors[i]){
507            return(i);
508          }
509        }
510      }
511    }
512  }
513
514  // Check color state of grid to see if it matches entry code.
515  boolean code_check(){
516    for(byte c=0;c<COLS;c++){
517      for(byte r=0; r < ROWS; r++){
518        if(rGrid[c][r] != rCode[c][r]){
519          return(0);
520        }
```

```
521         if(gGrid[c][r] != gCode[c][r]){
522           return(0);
523         }
524         if(bGrid[c][r] != bCode[c][r]){
525           return(0);
526         }
527       }
528     }
529     return(1);
530   }
531
532   // The entry code was apparently correct.
533   void open_door(){
534     // Activate the lock solenoid.
535     digitalWrite(lock_pin, HIGH);
536     if(first_boot==1){
537       //Just a test, unlock quickly!
538       color_effect(255, 255, 255, 1);
539     }
540     else{
541       // Change the keypad green for 5 seconds or soish.
542       color_effect(0, 255, 0, 5);
543     }
544     // Deactivate the door solenoid again.
545     digitalWrite(lock_pin, LOW);
546     // More silly debug code.
547     Serial.print("Door latch armed.");
548   }
549
550   // Change all of the keypad buttons to the provided color for n seconds and turn it off
      again.
551   void color_effect(byte dRed, byte dGreen, byte dBlue, byte time){
552     byte c;
553     // Leave the grid alone, just write the pot.
554     for(byte r=0; r<ROWS; r++){
555       write_pot(red[r],dRed);
556       write_pot(green[r],dGreen);
557       write_pot(blue[r],dBlue);
558     }
559     for(c=0;c<COLS;c++){
560       // Turn one col on while pot is written.
561       digitalWrite(colpin[c], HIGH);
562     }
563     // Turn time into secondsish.
564     delay(time*1000);
565     for(c=0;c<COLS;c++){
566       // Turn the col back off.
567       digitalWrite(colpin[c], LOW);
568     }
569     // Turn off everything in the pot.
570     clear_pot();
571   }
572
```

```
573   // This is he eye candy while the keypad isn't in use.
574   void idle_effect(){
575     //What effect was selected at the beginning?
576
577     // Only one light on at a time, one predetermined color.
578     if(effect_select==1){
579       //The grid needs cleared for this one.
580       grid_init();
581
582       for(byte r=0; r < ROWS; r++){
583         for(byte c=0;c < COLS;c++){
584           if(effect_state == 1){
585             effect[c][r] = 1;
586             effect_state = 0;
587             return;
588           }
589           if(effect[c][r]) {
590             rGrid[c][r] = rColors[effect_color];
591             gGrid[c][r] = gColors[effect_color];
592             bGrid[c][r] = bColors[effect_color];
593             effect[c][r] = 0;
594             effect_state = 1;
595           }
596         }
597       }
598     }
599
600     // Light the whole grid at once, one predetermined color.
601     if(effect_select==2){
602       //The grid needs cleared for this one.
603       grid_init();
604
605       byte red = rColors[effect_color];
606       byte blue = bColors[effect_color];
607       byte green = gColors[effect_color];
608       if(red != 0)
609         red = red - effect_state;
610
611       for(byte r=0; r < ROWS; r++){
612         for(byte c=0;c<COLS;c++){
613           rGrid[c][r] = rColors[effect_color];
614           gGrid[c][r] = gColors[effect_color];
615           bGrid[c][r] = bColors[effect_color];
616           effect[c][r] = 0;
617           effect_state = 1;
618         }
619       }
620     }
621
622     // Only one light on at a time, random color.
623     if(effect_select==3){
624       //The grid needs cleared for this one.
625       grid_init();
```

```
626
627       for(byte r=0; r < ROWS; r++){
628         for(byte c=0;c < COLS;c++){
629           if(effect_state == 1){
630             effect[c][r] = 1;
631             effect_state = 0;
632             return;
633           }
634           if(effect[c][r]) {
635             //Select our random color: R, G, B
636             effect_color_rand = random(3) + 1;
637
638             rGrid[c][r] = rColors[effect_color_rand];
639             gGrid[c][r] = gColors[effect_color_rand];
640             bGrid[c][r] = bColors[effect_color_rand];
641
642             effect[c][r] = 0;
643             effect_state = 1;
644           }
645         }
646       }
647     }
648
649     // Light the whole grid at once, random color.
650     if(effect_select==4){
651       //The grid needs cleared for this one.
652       grid_init();
653
654       //Select our random color: R, G, B
655       effect_color_rand = random(3) + 1;
656
657       byte red = rColors[effect_color_rand];
658       byte blue = bColors[effect_color_rand];
659       byte green = gColors[effect_color_rand];
660       if(red != 0)
661         red = red - effect_state;
662
663       for(byte r=0; r < ROWS; r++){
664         for(byte c=0;c<COLS;c++){
665           rGrid[c][r] = rColors[effect_color_rand];
666           gGrid[c][r] = gColors[effect_color_rand];
667           bGrid[c][r] = bColors[effect_color_rand];
668
669           effect[c][r] = 0;
670           effect_state = 1;
671         }
672       }
673     }
674
675     // Light the whole grid at once, block, randomly mixed color, with fade.
676     if(effect_select==5){
677       if((long)millis() - (long)time > 10){
678         time = millis();
```

```
679         for(byte x = 0; x < COLS; ++x){
680           for(byte y = 0; y < ROWS; ++y){
681             rGrid[x][y] = constrain(rGrid[x][y] + ((trajectory[x][y] & B001 ) ? 1 : -1),L,
      H);
682             gGrid[x][y] = constrain(gGrid[x][y] + ((trajectory[x][y] & B010 ) ? 1 : -1),L,
      H);
683             bGrid[x][y] = constrain(bGrid[x][y] + ((trajectory[x][y] & B100 ) ? 1 : -1),L,
      H);
684             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
      ) ) {
685               trajectory[x][y] = random(1,8);
686             }
687           }
688         }
689       }
690     }
691   }
692
693
```