Software Development for RGB Lighting Patterns

Brief Description

The KTD2061 is optimized to drive up to 36 LED channels (arranged as 12 RGB modules) in applications that benefit from lighting effects. The I²C register map is both compact and highly flexible to minimize activity on the I²C bus while still enabling sophisticated lighting patterns. Although the KTD2061 does not contain a flexible pattern generator, it does contain 36 independent, high-resolution, exponential fade-engines to eliminate the burden for implementing smooth ramping and fade-in/out effects in software. Therefore, system developers only need to write simple code to select global settings and fade rates, and then issue real-time commands for the on/off status and color-target of each RGB module.

This user guide shows how to write, run and test Python scripts with the KTD2061 EVKit. Numerous scripts are provided for various RGB lighting patterns. These scripts serve as a teaching tool for how to best utilize the registers in the KTD2061. The scripts may be modified, exported or translated to other software languages to meet the needs of any system using the KTD2061.

System Diagram

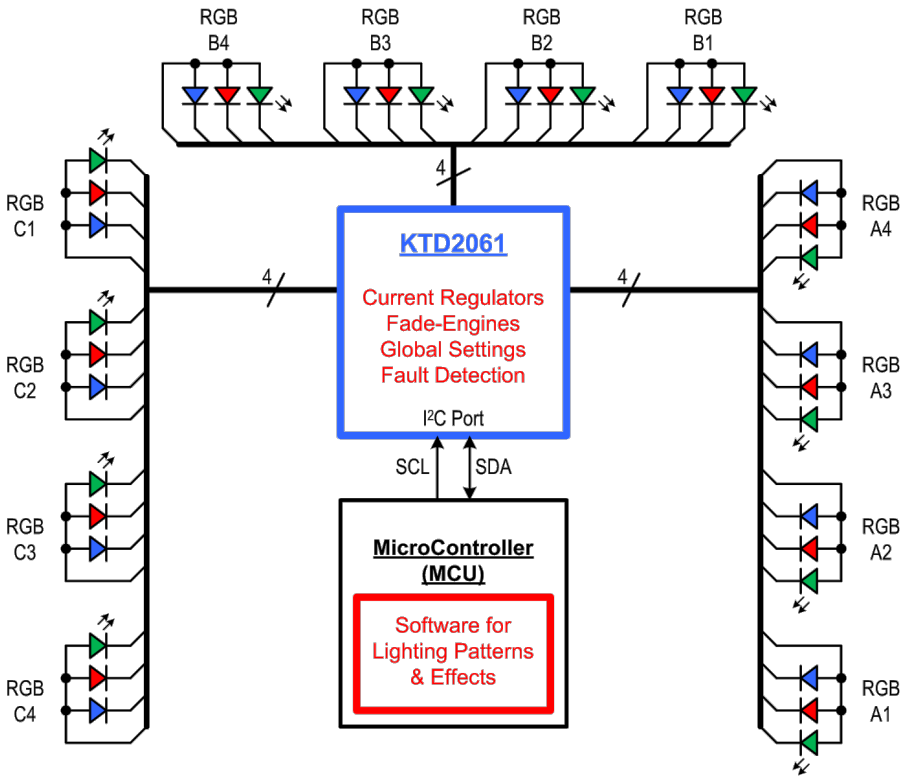

Figure 1. System Diagram for the KTD2061 RGB LED Driver IC

Table of Contents

I²C Register Map

The KTD2061 register map is shown in Figure 2. The register map is comprised of four sections:

1) ID [0x00] & MONITOR [0x01] – read only registers that confirm the chip's ID and status.
2) CONTROL [0x02] – read/write register to enable/disable the IC, set various features, and set the global fade rate.
3) Color/Current "Setting" Registers [0x03 to 0x08] – read/write registers to set two specific RGB colors or set a palette of 8 total colors.
4) Color/Current "Selection" Registers [0x09 to 0x0E] – read/write registers to control the individual RGB modules. Each register controls two RGBs with four bits of control per RGB. Of the four bits, the MSB controls the on/off of the assigned RGB, while the three remaining bits select the assigned RGB's color from the available colors as defined in 3) above.

I²C Register Map

Hex Address	Name	Type	Access	Default Reset	B7	B6	B5	B4	B3	B2	B1	B0
0x00	ID	Data	R	1010 0100	VENDOR[2:0]			DIE_ID[4:0]				
0x01	MONITOR	Status	R	0000 0000	DIE_REV[3:0]				SC STAT	BE _STAT	CE _STAT	UV/OT _STAT
0x02	CONTROL	Config	R/W	0000 0000	EN_MODE[1:0]		BE_EN	CE_TEMP[1:0]		FADE_RATE[2:0]		
0x03	IRED0	Config	R/W	0010 1000	IRED_SET0[7:0]							
0x04	IGRN0	Config	R/W	0010 1000	IGRN_SET0[7:0]							
0x05	IBLU0	Config	R/W	0010 1000	IBLU_SET0[7:0]							
0x06	IRED1	Config	R/W	0110 0000	IRED_SET1[7:0]							
0x07	IGRN1	Config	R/W	0110 0000	IGRN_SET1[7:0]							
0x08	IBLU1	Config	R/W	0110 0000	IBLU_SET1[7:0]							
0x09	ISELA12	Config	R/W	0000 0000	ENA1	RGBA1_SEL[2:0]			ENA2	RGBA2_SEL[2:0]		
0x0A	ISELA34	Config	R/W	0000 0000	ENA3	RGBA3_SEL[2:0]			ENA4	RGBA4_SEL[2:0]		
0x0B	ISELB12	Config	R/W	0000 0000	ENB1	RGBB1_SEL[2:0]			ENB2	RGBB2_SEL[2:0]		
0x0C	ISELB34	Config	R/W	0000 0000	ENB3	RGBB3_SEL[2:0]			ENB4	RGBB4_SEL[2:0]		
0x0D	ISELC12	Config	R/W	0000 0000	ENC1	RGBC1_SEL[2:0]			ENC2	RGBC2_SEL[2:0]		
0x0E	ISELC34	Config	R/W	0000 0000	ENC3	RGBC3_SEL[2:0]			ENC4	RGBC4_SEL[2:0]		

Figure 2. Register Map for the KTD2061 RGB LED Driver IC

EVKit Hardware

The KTD2061 EVKit hardware is shown in Figure 3. There are two versions of the EVKit. The square EVKit is optimized for evaluation with test-points and jumpers for voltage and current measurements. The round EVKit is optimized for demonstrations using a light-guide and diffuser – specifically the mechanical body of Amazon's Echo Dot, gen2. Both EVKits are useful for software development. But for systems that include a light-guide and diffuser, the round EVKit is more ideal.

Below the EVKits, Figure 3 shows two different interface boards. Either board can work with either EVKit. The larger interface board on the left is from Kinetic Technologies and is used to interface to the Windows EVKit GUI. (There is also a new version of the Kinetic interface board in a small black plastic box.) The smaller interface board is a MicroPython pyboard. When connected to a PC, the pyboard connects as a USB storage device and contains a file "main.py". This text file is Python script that auto-executes whenever the pyboard is powered up. When saving a modified main.py, a small red LED on the pyboard lights. Once the file is saved and the LED turns off, press the reset button on the pyboard to execute the updated script. Or remove the USB cable to de-power the board, and then reconnect the cable to the PC's USB port, any USB adapter, or a USB battery bank to execute the script.

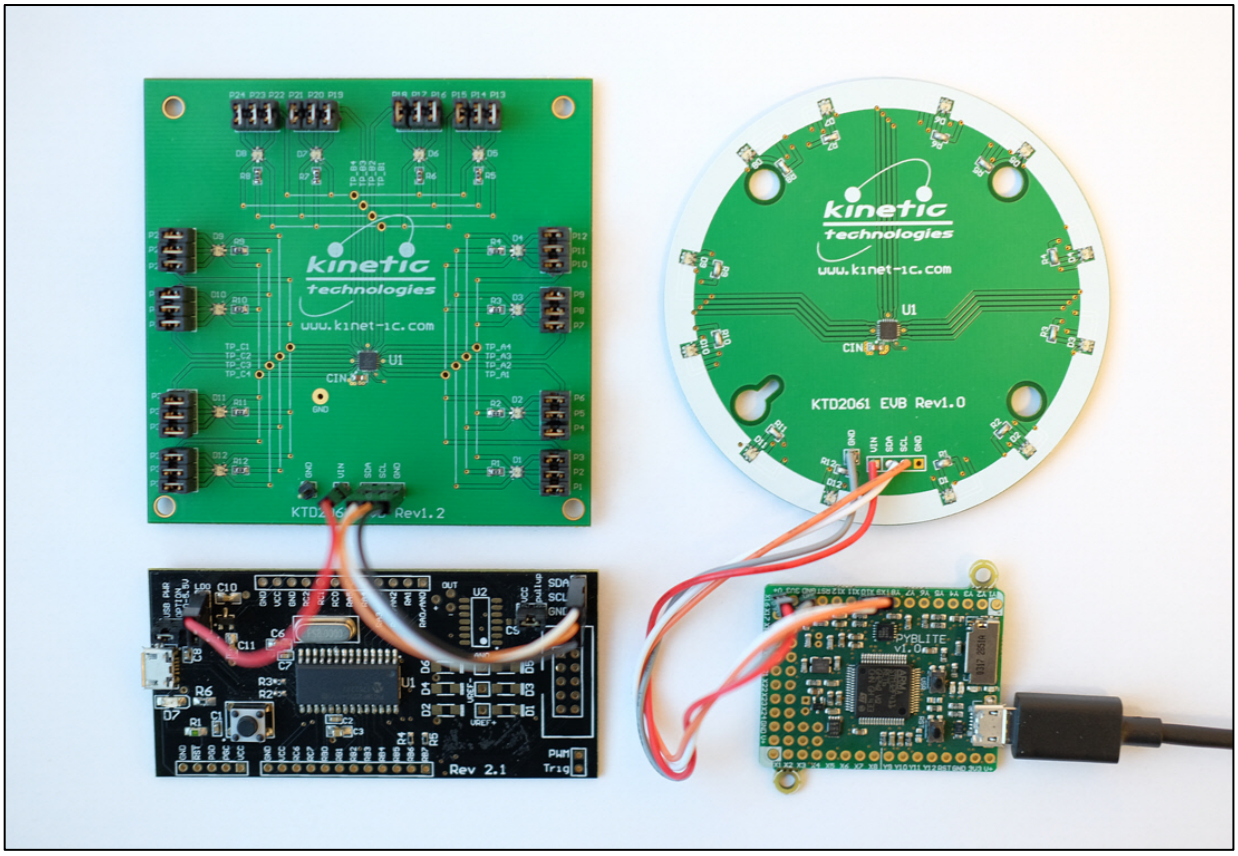

Figure 3. KTD2061 EVKit Hardware

EVKit GUI – Connection & Top Tab

When using the Kinetic interface board, connect the EVKit hardware to a PC using a USB micro-B cable. Open the GUI in Windows (see Figure 4) and click the red "Connect" GUI button in the upper left to connect to the hardware. By default, the GUI opens on the "TOP" tab, which shows the top-level ID (0x00) and MONITOR (0x01) read-only registers.

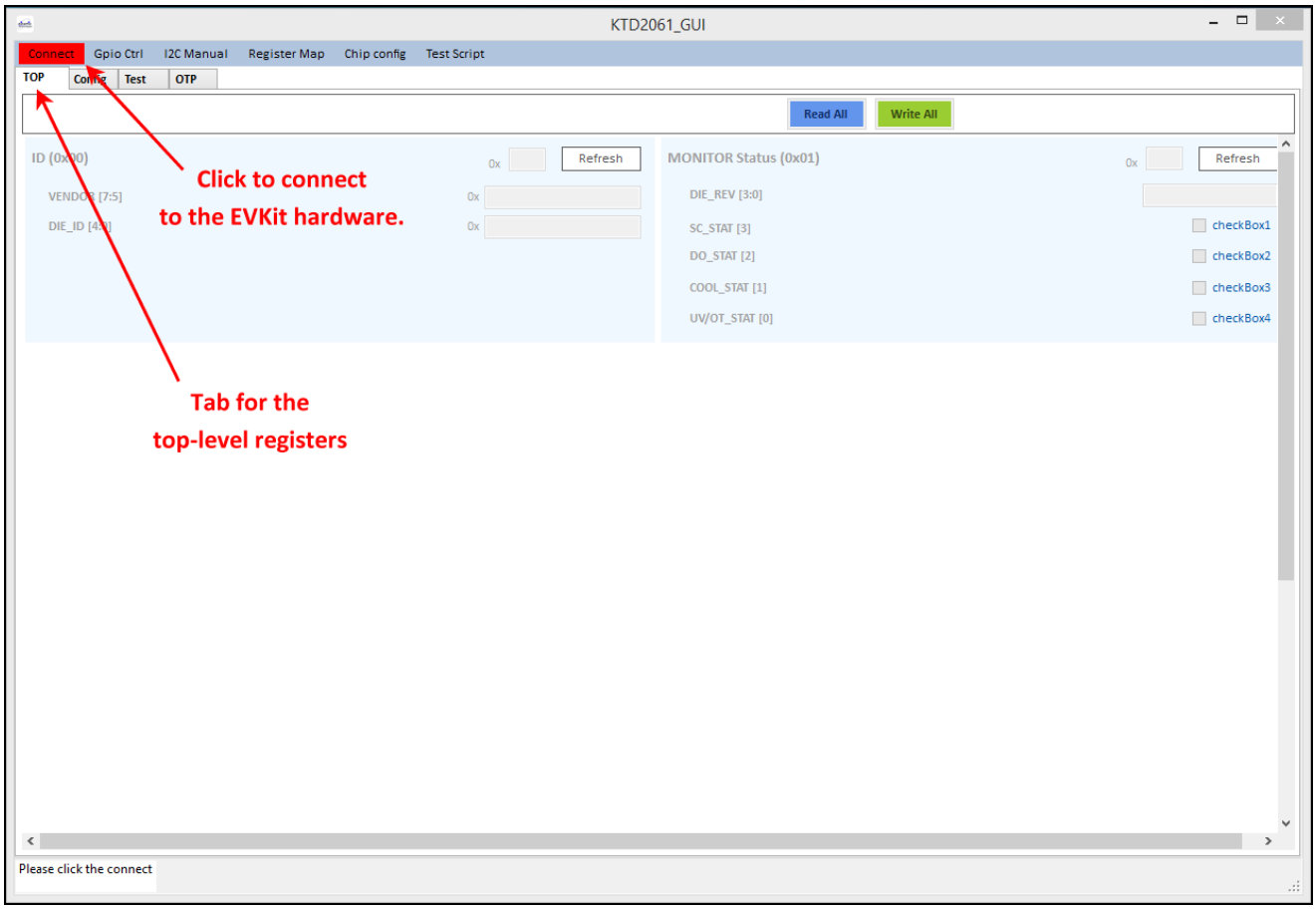

Figure 4. KTD2061 EVKit GUI opening window – click the red "Connect" button.

EVKit GUI – Config Tab

Select the "Config" tab (see Figure 5) to read, adjust, and write to the KTD2061's configuration registers. Alternatively, select the "Register Map" from the menu to access an interactive register map (similar to Figure 2). The GUI includes blue "Read All" and green "Write All" buttons to read (refresh) all register contents in the GUI and to write (update) all registers in the KTD2061, respectively.

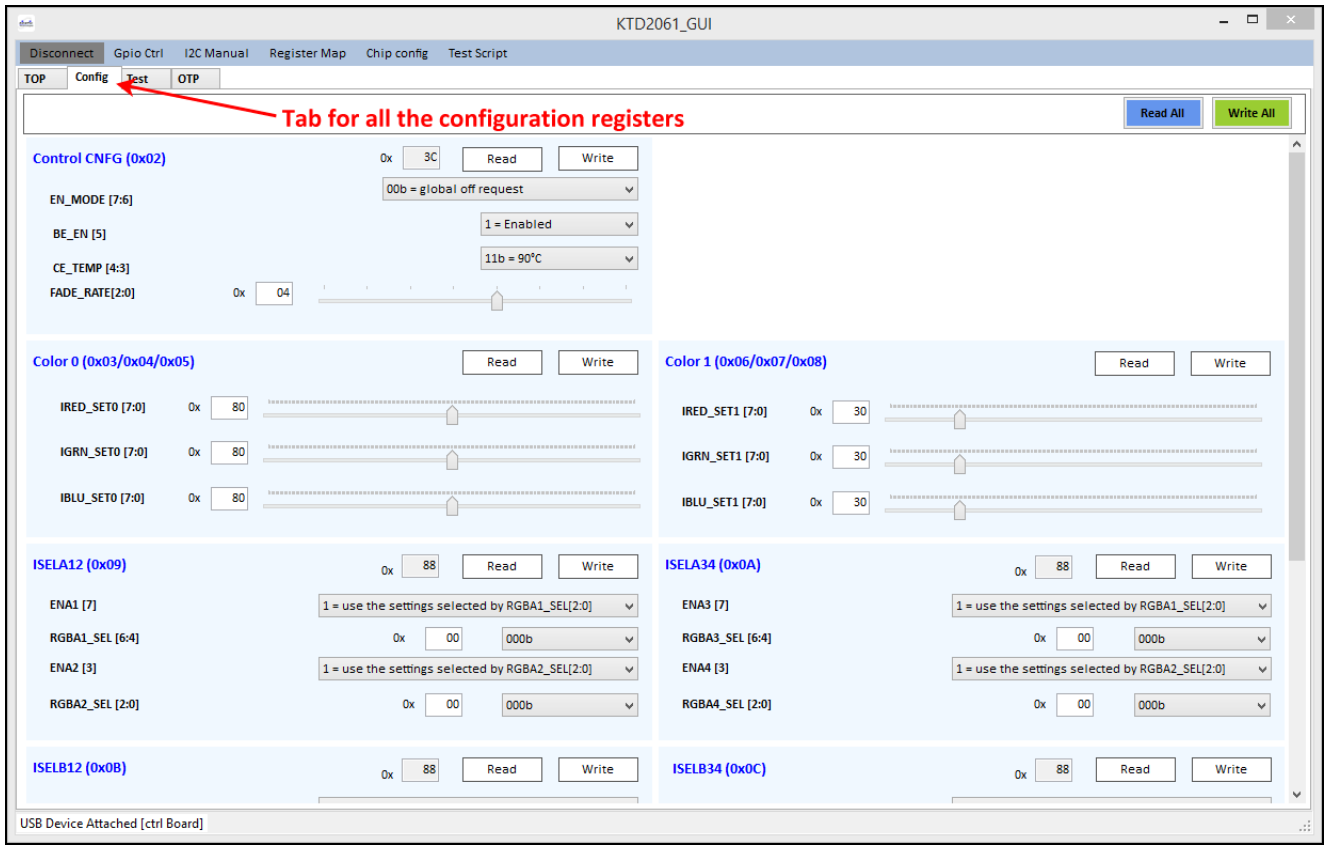

Figure 5. KTD2061 EVKit GUI Config tab – read, adjust, and write to the configuration registers here.

EVKit GUI – Python Script Window

The KTD2061 EVKit GUI contains a menu item “Test Script” which opens a Python scripting window (see Figure 6). Scripts may be written here by selecting the “Custom” radio button and composing in the text editor pane. However, it is often advantageous to use a more powerful text editor and subsequently load the scripts into this window by selecting the “From File” radio button. The Atom editor (https://atom.io) is highly recommended.

This document provides various scripts. They can be copied and pasted into the GUI’s scripting window. They may be easily modified and quickly tested on the EVKit hardware to see the resulting change to the RGB lighting patterns.

One limitation of the GUI’s Python scripting window is that the scripts are interpreted and executed within Microsoft Windows on the PC. Therefore, they run slightly slow with variable timing. Even so, the GUI provides a fairly useful platform for rapid software development. However, for complex patterns with more write commands, other platforms may serve better, such as MicroPython, Raspberry Pi, Arduino, BeagleBone, etc. The MicroPython pyboard is exceptionally convenient, but you will need to add a defined function to translate the GUI’s normal “i2c_write(SID, register, data)” into the pyboard’s “i2c.mem_write(data, SID, register)” command.

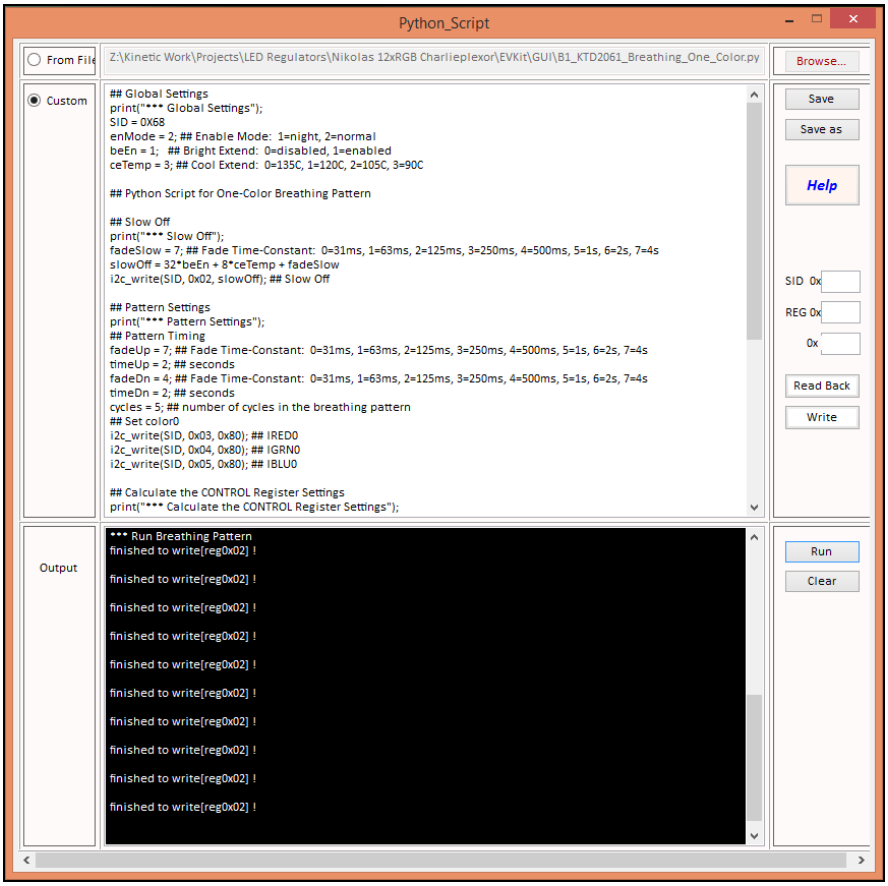

Figure 6. KTD2061 EVKit GUI Python Script Window – load, run, edit and test Python scripts here.

Global Core Function Definitions

The below core functions are defined to ease software development and reduce the size of the code. These functions map typical sequences of I^2C commands to the control register, color setting registers, and color selection registers. Software lighting pattern scripts call these functions to simplify their code. Example scripts in this document demonstrate how to utilize these functions.

```
# Global Core Function Definitions
import time
import random
SID = 0x68                                      # KTD2061 i2c address
en_mode = 2                                     # EnableMode:   1=night, 2=normal(day)
be_en = 1                                       # BrightExtend: 0=disabled, 1=enabled
ce_temp = 3                                     # CoolExtend:   0=135C, 1=120C, 2=105C, 3=90C
on = en_mode*64 + be_en*32 + ce_temp*8          # calculate global on
off = be_en*32 + ce_temp*8                      # calculate global off
def slow_off():
    i2c_write(SID, 0x02, off+7)
def global_reset():
    i2c_write(SID, 0x02, 0xC0)
def global_on(fade):
    i2c_write(SID, 0x02, on+fade)
def global_off(fade):
    i2c_write(SID, 0x02, off+fade)
def set_color0(ired0, igrn0, iblu0):
    i2c_write(SID, 0x05, iblu0)
    i2c_write(SID, 0x04, igrn0)
    i2c_write(SID, 0x03, ired0)
def set_color1(ired1, igrn1, iblu1):
    i2c_write(SID, 0x08, iblu1)
    i2c_write(SID, 0x07, igrn1)
    i2c_write(SID, 0x06, ired1)
def select_all(isel):
    for reg in range(0x09, 0x0F):
        i2c_write(SID, reg, isel)
def select_off():
    select_all(0x00)
def select_color0():
    select_all(0x88)
def select_color1():
    select_all(0xFF)
def select_colors(isela12, isela34, iselb12, iselb34, iselc12, iselc34, delay):
    i2c_write(SID, 0x09, isela12)
    i2c_write(SID, 0x0A, isela34)
    i2c_write(SID, 0x0B, iselb12)
    i2c_write(SID, 0x0C, iselb34)
    i2c_write(SID, 0x0D, iselc12)
    i2c_write(SID, 0x0E, iselc34)
    time.sleep(delay)
def select_one(reg, data, delay):
    i2c_write(SID, reg, data)
    time.sleep(delay)
def select_one_clear(reg, data, delay):
    select_one(reg, data, delay)
    i2c_write(SID, reg, 0x00)
```

Smooth Pattern-to-Pattern Transitions

The KTD2061's advanced fade engines make smooth pattern-to-pattern transitions exceptionally easy to encode in software. There are four benefits:

1) No sudden, unpleasant discontinuity in the RGB lighting.
2) No spurious lighting effects when reconfiguring the color palette settings or RGB color selection registers.
3) No annoying delay while waiting for one pattern to complete before executing the next pattern.
4) No need for the software to keep track of where it is within one pattern before transitioning to the next pattern.

The Python script "slow_off()" for smooth pattern-to-pattern transitions is a single write command to the CONTROL register (0x02) to request global off with a slow fade-out. While this fade-out is executed autonomously by the KTD2061, all the other configuration registers (0x03 to 0x0E) may be pre-configured without any effect on the LEDs or the slow fade-out. Then, once pre-configured, a single write to the CONTROL register (0x02) for global enable with fade-in is used to initiate the next pattern with a smooth fade-in transition. Note that the Slow Off fade-out does not need to complete before the next pattern is enabled and fades in.

The below "slow_off()" python script is for smooth pattern-to-pattern transitions. It can be inserted at the beginning of all patterns, and sometimes it is useful in the middle of patterns. For some patterns, it may make sense to use a faster fade-rate.

```
def slow_off():
    i2c_write(SID, 0x02, off+7)
```

For "global_reset()", use the following script. This writes a global reset (0xC0) to the CONTROL register (0x02). However, the global reset restores all registers to their power-on default values; therefore, the CONTROL register will read back as 0x00 (its POR default setting), which includes resetting the fade-rate to its fastest setting of 31ms. Effectively, the global reset translates to a request for global off with near-instant fade-out. Obviously, this is not for smooth pattern-to-pattern transitions. Instead, it is useful when needing to execute a new pattern quickly and starting from a known register status.

```
def global_reset():
    i2c_write(SID, 0x02, 0xC0)
```

Checking that All LEDs are Functional

The below Pythons script is useful to check the functionality of the KTD2061, the RGB LED modules, the PCB, and the solder connections. It simply turns on all LEDs at their Color0 POR default setting of 5mA. After execution, all RGBs should be on and showing a medium-bright white color.

```
# Python Script to check that all LEDs are working.  Displays mid white.
def check(fade, delay):
    global_reset()
    select_color0()
    global_on(fade)
    time.sleep(delay)

# Execute
check(5, 4)                              # functional check (mid white), fade=5, delay=4s
```

Choosing the Color Palette Script

The KTD2061's registers allow two specific RGB colors or a palette of 8 simultaneous colors. The below script is used to quickly display a palette of colors on the twelve RGB modules. Then the color settings are easily viewed, adjusted, and optimized to achieve the precise colors desired for the pattern.

By default, this script sets Color 0 to black and Color 1 to white. Then it selects Color 0 for the first three RGBs, a blend of Color 0 and Color 1 in binary sequence for the next six RGBs, and Color 1 for the final three RGBs.

To modify the colors, there are two ways:

1) Modify the Color 0 and Color 1 settings in the script and run it again, or…
2) Exit the scripting window and made adjustment in the Config tab of the GUI. First click the Read All button to refresh the GUI to match the registers in the IC. Then adjust and write Color 0 and Color 1.

As an exercise, set a pastel color palette by changing Color 0 to dim white (0x20, 0x20, 0x20). After that, change Color 1 to yellow by setting blue to zero. Then change Color 0 to dim cyan by setting red to zero.

```
# Python Script for choosing the color palette
def color_palette(fade, delay):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black
    set_color1(0x80, 0x80, 0x80)                # color1 = white
    select_colors(0x88, 0x89, 0xAB, 0xCD, 0xEF, 0xFF, 0)
    global_on(fade)
    time.sleep(delay)

# Execute
color_palette(5, 4)                             # displays color palette, fade=5, delay=4s
```

One-Color Breathing Pattern

The below script executes the one-color breathing pattern by utilizing the fade engines and the global on and off functions. This works because global off first fades all LEDs to zero current before turning off the KTD2061.

The breathing() function executes the timing, while the purple_breathing() function selects the colors and timing values to pass to the breathing() function.

While executing, only one I^2C write command is issued every 1.5 seconds. Competing solutions without fade engines would require 256 I^2C write commands for each smooth fade ramp, assuming they have a global "bank" function. Without a "bank" function, they would need 256 x 12 = 3072 I^2C write commands, provided they have a brightness register per RGB module.

```
# Python Script for Global Breathing Pattern
def breathing(cycles, fadeOn, timeOn, fadeOff, timeOff):
    for i in range(cycles):
        global_on(fadeOn)
        time.sleep(timeOn)
        global_off(fadeOff)
        time.sleep(timeOff)

# Python Script for Purple Breathing Pattern
def purple_breathing(cycles):
    slow_off()
    set_color0(0x60, 0x00, 0xC0)                # color0 = purple
    select_color0()
    breathing(cycles, 7, 1.5, 3, 1.5)           # fadeOn=7, timeOn=1.5s, fadeOff=3, timeOff=1.5s

# Execute
purple_breathing(3)                             # one-color breathing, cycles=3
```

Two-Color Breathing Pattern

In one-color breathing, the script made good use of the “global off request” fade-out feature to implement breathing in the simplest way possible. Two-color breathing is less simple because the RGB modules do not fade towards zero current, but instead fade between two different colors. The script sets Color 0 and Color 1, and then the loop alternates all the RGB modules between the two colors.

Depending upon the chosen colors and their brightness, some experimentation on the delays and fade-rates is necessary to achieve the most pleasing effect.

```
# Python Script for Two-Color Breathing Pattern
def two_color_breathing(cycles, fadeUp, timeUp, fadeDn, timeDn):
    slow_off()
    for i in range(cycles):
        select_color0()
        global_on(fadeUp)
        time.sleep(timeUp)
        slow_off()
        select_color1()
        global_on(fadeDn)
        time.sleep(timeDn)
        slow_off()

# Python Script for Tequila Sunset Two-Color Breathing Pattern
def tequila_sunset_breathing(cycles):
    slow_off()
    set_color0(0xC0, 0x40, 0x00)                # color0 = amber
    set_color1(0x08, 0x00, 0x00)                # color1 = dark red
    two_color_breathing(cycles, 7, 1.5, 3, 1.5)     # fadeUp=7, timeUp=1.5s, fadeDn=3, timeDN=1.5s

# Execute
tequila_sunset_breathing(3)                     # two-color breathing, cycles=3
```

Double-Rainbow Breathing Pattern

Breathing patterns can use multiple colors assigned to the RGBs. In this script, a palette of rainbow colors is constructed by setting Color 0 to black and Color 1 to white. There are six colors in a rainbow: magenta, blue, cyan, green, yellow, and red. The rainbow colors are in the correct sequence when mapped to the RGBs in gray code (rather than binary). With 12 RGB modules, the rainbow can repeat twice around the circumference of a ring-light, thereby enabling the full rainbow to be seen in 180 degrees of viewing angle.

While executing, only one I²C write command is issued every 1.5 seconds. Because each RGB is a different color, competing solutions without fade engines would require 256 x 12 = 3072 I²C write commands every 1.5 seconds, provided they have a brightness register per RGB module.

```
# Python Script for Global Breathing Pattern
def breathing(cycles, fadeOn, timeOn, fadeOff, timeOff):
    for i in range(cycles):
        global_on(fadeOn)
        time.sleep(timeOn)
        global_off(fadeOff)
        time.sleep(timeOff)

# Python Script for Double-Rainbow Breathing Pattern
def double_rainbow_breathing(cycles):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black (off)
    set_color1(0x80, 0x80, 0xC0)                # color1 = white (more balanced)
    select_colors(0xB9, 0xDC, 0xEA, 0xB9, 0xDC, 0xEA, 0)
    breathing(cycles, 7, 1.5, 3, 1.5)             # fadeOn=7, timeOn=1.5s, fadeOff=3, timeOff=1.5s

# Execute
double_rainbow_breathing(3)                     # many-color breathing, cycles=3
```

Rainbow Flower Breathing Pattern

Similar to the Double-Rainbow Breathing pattern, the Rainbow Flower Breathing uses a rainbow palette of colors. However, in this case, the colors are displayed on every other RGB, while the in between RGBs are turned off. The For-loop alternates between using the odd RGBs and the even RGBs, while the fade-engines allow a small overlap in the breathing of the even and odd RGBs. For competing devices without fade engines, it is very difficult to program overlapping fade effects.

```
# Python Script for Global Breathing Pattern
def breathing(cycles, fadeOn, timeOn, fadeOff, timeOff):
    for i in range(cycles):
        global_on(fadeOn)
        time.sleep(timeOn)
        global_off(fadeOff)
        time.sleep(timeOff)

# Python Script for Rainbow Flower Breathing Pattern
def rainbow_flower_breathing(cycles):
    slow_off()
    set_color0(0x00, 0x00, 0x00)         # color0 = black (off)
    set_color1(0x80, 0x80, 0xC0)         # color1 = white (more balanced)
    for i in range(round(cycles/2)):
        select_colors(0xC0, 0xE0, 0xA0, 0xB0, 0x90, 0xD0, 0)
        breathing(1, 6, 0.5, 3, 0.5)     # cycles=1, fadeOn=6, timeOn=0.5s, fadeOff=3, timeOff=0.5s
        select_colors(0x0E, 0x0A, 0x0B, 0x09, 0x0D, 0x0C, 0)
        breathing(1, 6, 0.5, 3, 0.5)     # cycles=1, fadeOn=6, timeOn=0.5s, fadeOff=3, timeOff=0.5s

# Execute
rainbow_flower_breathing(8)              # many-color breathing, cycles=8
```

One-Color Chasing Pattern

In chasing patterns, the lighting is animated to move across the sequential RGBs. This is done by defining a generic one_color_chasing function that steps through all the selection registers, turning the RGBs on and off in sequence with a parameterized delay and fade-rate. The below function also includes a "presel" variable to preselect off for all the RBGs when presel = 1. If presel does not equal 1, then the RGBs are simply left at their previous settings before the first chasing cycle, and then overwritten during the first chasing cycle.

The 36 independent fade-engines in the KTD2061 create a soft rising edge and long comet-tail falling edge to the chasing animation. The effect is visually pleasing. The fade-rate may be adjusted for more or less of this effect. Competing solutions without internal fade-engines would require highly complex software and very many I^2C write commands to create the "comet" effect.

Note that from the GUI, chasing runs slower than it would from a dedicated MCU, such as the pyboard.

```
# Python Script for Generic One-Color Chasing Pattern
def one_color_chasing(cycles, delay, fade, presel):
    slow_off()
    if presel == 1:                             # if presel = 1, pre-select off for all the RBGs
        select_off()
    global_on(fade)
    for i in range(cycles):                     # number of chasing cycles
        for reg in range(0x09, 0x0F):               # registers 0x09 to 0x0E
            select_one_clear(reg, 0x80, delay)
            select_one_clear(reg, 0x08, delay)

# Python Script for White One-Color Chasing Pattern
def white_chasing(cycles):
    slow_off()
    set_color0(0x80, 0x80, 0xC0)                # color0 = white (more balanced)
    one_color_chasing(cycles, 0.083, 2, 1)      # delay=83ms, fade=2, presel=1

# Execute
white_chasing(4)                                # one-color chasing, cycles=4
```

Two-Color Chasing Pattern

Two-color chasing is similar to one-color chasing, but with a background color. The foreground chasing color is Color 0, while the background color is Color 1. The “presel” variable is used to optionally pre-select all the RGB modules to Color 1 before the first chasing cycle, as opposed to simply over-writing the previous selections from a prior pattern.

Note that from the GUI, chasing runs slower than it would from a dedicated MCU, such as the pyboard.

```
# Python Script for Generic Two-Color Chasing Pattern
def two_color_chasing(cycles, delay, fade, presel):
    slow_off()
    if presel == 1:                             # if presel = 1, pre-select color1 for all the RBGs
        select_color1()
    global_on(fade)
    for i in range(cycles):                     # number of chasing cycles
        for reg in range(0x09, 0x0F):           # registers 0x09 to 0x0E
            select_one(reg, 0x80, delay)        # select color 0 for 1st RGB
            select_one(reg, 0xF8, delay)        # select color 1 for 1st RGB, color 0 for 2nd RGB
            i2c_write(SID, reg, 0xFF)           # select color 1 for 1st and 2nd RGB

# Python Script for White over Red Two-Color Chasing Pattern
def white_red_chasing(cycles):
    slow_off()
    set_color0(0x80, 0x80, 0xC0)                # color0 = white (more balanced)
    set_color1(0x10, 0x00, 0x00)                # color1 = dim red
    two_color_chasing(cycles, 0.083, 2, 0)      # delay=83ms, fade=2, presel=0
    global_off(2)
    time.sleep(5/32*2**2)                       # delay of 5 time-constants

# Execute
white_red_chasing(5)                            # two-color chasing, cycles=5
```

Dual Chasing Pattern

For dual-chasing, two RGBs chase each other from opposite sides of the ring-light.

Note that from the GUI, chasing runs slower than it would from a dedicated MCU, such as the pyboard.

```
# Python Script for Generic Dual-Chasing Pattern
def dual_chasing(cycles, delay, fade):
    global_on(fade)
    for i in range(cycles*2):
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0x80)
            select_one_clear(reg+3, 0x80, delay)
            i2c_write(SID, reg, 0x08)
            select_one_clear(reg+3, 0x08, delay)
            i2c_write(SID, reg, 0x00)

# Python Script for Dual-Chasing first red, then green, then blue
def RGB_dual_chasing():
    slow_off()
    select_off()
    set_color0(0xC0, 0x00, 0x00)                    # color0 = red
    dual_chasing(3, 0.083, 1)                       # cycles=3, delay=83ms, fade=1
    set_color0(0x04, 0xC0, 0x04)                    # color0 = green
    dual_chasing(3, 0.083, 1)                       # cycles=3, delay=83ms, fade=1
    set_color0(0x04, 0x04, 0xC0)                    # color0 = blue
    dual_chasing(3, 0.083, 1)                       # cycles=3, delay=83ms, fade=1

# Execute
RGB_dual_chasing()
```

Korean Flag Yin-Yang Chasing Pattern

The Korean flag contains a Yin-Yang symbol with red and blue. In this chasing pattern, red and blue chase each other with comet tails. The effect reminds one of the iconic symbol on the Korean flag.

Note that from the GUI, chasing runs slower than it would from a dedicated MCU, such as the pyboard.

```
# Python Script for Korean Yin-Yang Chasing Pattern
def Kor_YinYang(cycles, delay, fade):
    slow_off()
    set_color0(0xC0, 0x00, 0x00)
    set_color1(0x00, 0x00, 0xC0)
    select_off()
    global_on(fade)
    for i in range(cycles):
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0x80)
            select_one(reg+3, 0xF0, delay)
            i2c_write(SID, reg, 0x08)
            select_one_clear(reg+3, 0x0F, delay)
            i2c_write(SID, reg, 0x00)
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0xF0)
            select_one(reg+3, 0x80, delay)
            i2c_write(SID, reg, 0x0F)
            select_one_clear(reg+3, 0x08, delay)
            i2c_write(SID, reg, 0x00)
    global_off(fade)
    time.sleep(5/32*2**fade)                    # delay of 5 time-constants

# Execute
Kor_YinYang(5, 0.083, 2)                        # yin-yang chasing, cycles=5, delay=83ms, fade=2
```

Twelve-Color Patterns

The color setting registers of the KTD2061 are designed for a color palette of 8 colors. While the color palette can be changed mid-pattern, it is normally only possible to display up to 8 colors at any given moment. However, by using the fade-engines, it becomes possible to display 12 simultaneous unique colors on the 12 RGBs.

The twelve_color_rainbow() function uses 0mA, 24mA, and 6mA current settings to mix the 12 colors. First, the 0mA and 24mA settings are used to set some of the LEDs to their required brightness. Then the fade engines are used to hold these colors on those LEDs. While holding, the 0mA and 6mA settings are used to set other LEDs to their required brightness. Then the fade engines hold these colors and the For-loop returns to the 0/24mA LEDs. Due to the 50% duty, the average currents are half of the 0/24/6mA settings (namely 0/12/3mA average). The fade time-constant is very long compared to the delay in the For-loop; therefore, the LED currents are averaged to nearly constant current (DC).

In the twelve_colors() function, it is possible to set a completely unique current for each of the 36 LED channels, albeit at 1/6th duty (1/6th brightness when averaged by the fade engines). In the below example, the colors are still the 12 rainbow colors, but each RGB could be adjusted to any unique color (from 7 million possible colors) by modifying the code.

The chasing_rainbow() function is an exhibition pattern that calls the twelve_colors() function and steps the fade rate down and back up to reveal the pattern's internal chasing with and without the averaging of the fade engines.

```
# Python Script to make a rainbow with 12 independent colors using fade engines and 50% duty
def twelve_color_rainbow(cycles, delay, fade):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                    # color0 = black (off)
    select_off()
    global_on(fade)
    for i in range(cycles):
        set_color1(0xC0, 0xC0, 0xC0)                # color1 = white
        select_colors(0xCC, 0xEA, 0xAA, 0xB9, 0x99, 0xDC, delay/2)
        select_off()
        set_color1(0x30, 0x30, 0x30)                # color1 = grey (6mA)
        select_colors(0x8A, 0x8C, 0x89, 0x8A, 0x8C, 0x89, delay/2)
        select_off()

# Python Script to make 12 fully-independent colors using fade engines and 1/6th duty
# As set below, the colors are 12 rainbow colors.  It can also make chasing within the colors.
def twelve_colors(cycles, delay, fade):
    slow_off()
    select_off()
    global_on(fade)
    for i in range(cycles):
        set_color0(0xC0, 0x00, 0x00)
        set_color1(0xC0, 0x30, 0x00)
        select_colors(0x8F, 0x00, 0x00, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0xC0, 0xC0, 0x00)
        set_color1(0x30, 0xC0, 0x00)
        select_colors(0x00, 0x8F, 0x00, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0xC0, 0x00)
        set_color1(0x00, 0xC0, 0x30)
```

```
        select_colors(0x00, 0x00, 0x8F, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0xC0, 0xC0)
        set_color1(0x00, 0x30, 0xC0)
        select_colors(0x00, 0x00, 0x00, 0x8F, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0x00, 0xC0)
        set_color1(0x30, 0x00, 0xC0)
        select_colors(0x00, 0x00, 0x00, 0x00, 0x8F, 0x00, delay/6)
        select_off()
        set_color0(0xC0, 0x00, 0xC0)
        set_color1(0xC0, 0x00, 0x30)
        select_colors(0x00, 0x00, 0x00, 0x00, 0x00, 0x8F, delay/6)
        select_off()

# Python Script to execute the twelve_colors function and ramp the fade rate to reveal chasing
def chasing_rainbow():
    for fade in range(7, 0, -1):
        twelve_colors(1, 1, fade)
    for fade in range(0, 8):
        twelve_colors(1, 1, fade)
    global_off(4)
    time.sleep(3)

# Execute
twelve_color_rainbow(1, 1, 7)                   # cycles=1, delay=1s, fade=7
chasing_rainbow()                               # 12 color rainbow exhibition pattern
```

Matrix Water Flow Pattern

The “Matrix” or “Water Flow” pattern is popular for many applications. The below script mimics an automotive turn-signal application. First, it flashes white twice, then executes 3 cycles of the matrix water flow in a bright amber color. For example, this script could be applied when locking the doors via the key-fob.

Note that from the GUI, water flow runs slower than it would from a dedicated MCU, such as the pyboard.

```
# Python Script for Global Breathing Pattern
def breathing(cycles, fadeOn, timeOn, fadeOff, timeOff):
    for i in range(cycles):
        global_on(fadeOn)
        time.sleep(timeOn)
        global_off(fadeOff)
        time.sleep(timeOff)

# Python Script for Automotive-like turn-signal Matrix Water Flow Pattern
def matrix_water_flow(cycles):
    global_reset()
    set_color0(0x80, 0x30, 0x00)                # color0 = amber
    set_color1(0x40, 0x40, 0x60)                # color1 = white (more balanced)
    for j in range(cycles):                     # number of pattern cycles
        # Flash Twice using White
        select_color1()
        breathing(2, 0, 0.2, 0, 0.2)
        time.sleep(0.25)
        # Matrix Water Flow using Amber
        for i in range(3):                      # number of matrix cycles
            select_off()
            global_on(2)
            for reg in range(0x0E, 0x08, -1):   # registers 0x09 to 0x0E
                select_one(reg, 0x08, 0.03)     # select Color 0 for one RGB, delay=30ms
                select_one(reg, 0x88, 0.03)     # select Color 0 for both RGBs, delay=30ms
            time.sleep(0.25)                    # 250ms delay hold time
            global_off(0)
            time.sleep(0.35)

# Execute
matrix_water_flow(2)                            # automotive turn-signal, cycles=2
```

Arbitrary Animation Patterns

The select_colors() global core function may be used to define a pattern generator within the software. In the below rainbow_swirl() example, only red, green, and blue colors are each assigned to 1/3rd of the RGBs, and then rotated at a step-rate defined by the variable "delay". Although only three colors are written, the fade-rate and step-rate work together to blend the colors such that each of the 12 RGBs displays a unique color at any given moment. Unlike the previous twelve-color patterns, this pattern displays at full brightness (100% duty).

The rainbow_swirl() pattern is just one example. The select_colors() function may be used to realize a huge variety of other arbitrary patterns. It is especially useful when there isn't enough time to figure out a more clever way to realize a pattern.

```
# Python Script for Rotating Rainbow Pattern
# It only uses red, green, and blue, but fade engine creates intermediary colors.
def rainbow_swirl(cycles, delay, fade):
    slow_off()
    set_color0(0x04, 0x04, 0x04)            # color0 = black (off)
    set_color1(0x80, 0x70, 0xC0)            # color1 = white (more balanced)
    global_on(fade)
    for i in range(cycles):
        select_colors(0xCC, 0xCC, 0xAA, 0xAA, 0x99, 0x99, delay)
        select_colors(0x9C, 0xCC, 0xCA, 0xAA, 0xA9, 0x99, delay)
        select_colors(0x99, 0xCC, 0xCC, 0xAA, 0xAA, 0x99, delay)
        select_colors(0x99, 0x9C, 0xCC, 0xCA, 0xAA, 0xA9, delay)
        select_colors(0x99, 0x99, 0xCC, 0xCC, 0xAA, 0xAA, delay)
        select_colors(0xA9, 0x99, 0x9C, 0xCC, 0xCA, 0xAA, delay)
        select_colors(0xAA, 0x99, 0x99, 0xCC, 0xCC, 0xAA, delay)
        select_colors(0xAA, 0xA9, 0x99, 0x9C, 0xCC, 0xCA, delay)
        select_colors(0xAA, 0xAA, 0x99, 0x99, 0xCC, 0xCC, delay)
        select_colors(0xCA, 0xAA, 0xA9, 0x99, 0x9C, 0xCC, delay)
        select_colors(0xCC, 0xAA, 0xAA, 0x99, 0x99, 0xCC, delay)
        select_colors(0xCC, 0xCA, 0xAA, 0xA9, 0x99, 0x9C, delay)
    global_off(3)
    time.sleep(5/32*2**3)                   # delay of 5 time-constants

# Execute
rainbow_swirl(5, 0.2, 3)                    # rotating rainbow, cycles=5, delay=200ms, fade=3
```

Purple Raindrops Pattern

For almost any lighting pattern you can imagine, the KTD2061 is capable... Imagine walking home at night in the city. Rain begins to fall on the sidewalk. The droplets randomly hit the concrete and splatter. At the street corner, the splatter looks purple in the reflected light of a neon sign. You drank just enough at dinner to really appreciate how beautiful it looks. But now you're getting wet, so you hurry home.

This pattern really benefits from the fast execution of an MCU, such as the pyboard. The splash timing is very fast. Also, this pattern looks best with the round EVKit installed under diffuser.

```
# Python Script for Purple Raindrops Pattern
def purple_raindrops(cycles, delay, fade):
    slow_off()
    set_color0(0x60, 0x08, 0xC0)                # color0 = bright purple
    select_off()
    global_on(fade)
    for i in range(cycles):
        reg = random.randint(0x09, 0x0E)        # random selection register
        regp = reg + 1
        regm = reg - 1
        if regp == 0x0F:
            regp = 0x09
        if regm == 0x08:
            regm = 0x0E
        select_one(reg, 0x80, delay*2)          # select color 0 for reg 1st RGB
        select_one(reg, 0x08, 0)                # select color 0 for reg 2nd RGB
        select_one(regm, 0x08, delay/4)         # select color 0 for regm 2nd RGB
        select_one(reg, 0x00, 0)                # select off for reg 2nd RGBs
        select_one(regm, 0x00, delay)           # select off for regm 2nd RGB
        select_one(regp, 0x80, 0)               # select color 0 for regp 1st RGB
        select_one(regm, 0x80, delay/16)        # select color 0 for regm 1st RGB
        select_one(regp, 0x00, 0)               # select off for regp 1st RGB
        select_one(regm, 0x00, 0)               # select off for regm 1st RGB
        time.sleep(random.random())             # random delay
        reg = random.randint(0x09, 0x0E)        # random selection register
        regp = reg + 1
        regm = reg - 1
        if regp == 0x0F:
            regp = 0x09
        if regm == 0x08:
            regm = 0x0E
        select_one(reg, 0x08, delay*2)          # select color 0 for reg 2nd RGB
        select_one(reg, 0x80, 0)                # select color 0 for reg 1st RGB
        select_one(regp, 0x80, delay/4)         # select color 0 for regp 1st RGB
        select_one(reg, 0x00, 0)                # select off for reg 1st RGB
        select_one(regp, 0x00, delay)           # select off for regp 1st RGB
        select_one(regm, 0x08, 0)               # select color 0 for regm 2nd RGB
        select_one(regp, 0x08, delay/16)        # select color 0 for regp 2nd RGB
        select_one(regm, 0x00, 0)               # select off for regm 2nd RGB
        select_one(regp, 0x00, 0)               # select off for regp 2nd RGB
        time.sleep(random.random())             # random delay

# Execute
purple_raindrops(15, 0.05, 2)                   # cycles=15, delay=50ms, fade=2
```

Random Lighting Pattern

This generic random lighting pattern utilizes the random number module in Python. It randomly modifies the color palette and RGB color selections over time. The random current settings for the color palette are in octave increments to produce log-scale color coordinates, which appear more uniformly distributed to the human eye. The "mode" variable determines if the RGBs are randomly selected on/off (mode = 0) or forced always on (mode = 1).

With the "delay" and "fade" variables, the random lighting pattern can execute more quickly, such as for a music light-show, or more slowly, such as for meditation. To demonstrate this, the below script calls the random_pattern() function twice with different variables.

```
# Python Script for Random Lighting Pattern
def random_pattern(cycles, delay, fade, mode):
    global_on(fade)
    for i in range(cycles):
        iset = 2**random.randint(1, 7)              # random Iset in octaves
        reg = random.randint(0x03, 0x08)            # random color register
        i2c_write(SID, reg, iset)                   # random Iset to random reg
        if mode == 0:                               # mode = 0, random on/off + random color
            data = random.randint(0, 255)               # random selection data
        elif mode == 1:                             # mode = 1, fixed on only + random color
            data = 16*random.randint(8, 15) + random.randint(8, 15)
        reg = random.randint(0x09, 0x0E)            # random selection register
        i2c_write(SID, reg, data)                   # random data to random reg
        time.sleep(delay)                           # delay

# Execute
random_pattern(100, 0.2, 2, 0)              # for music, cycles=100, delay=200ms, fade=2, mode=0
random_pattern(40, 1, 5, 1)                 # for meditation, cycles=40, delay=1s, fade=5, mode=1
global_off(5)                               # global off at 1s fade-rate
time.sleep(5)                               # 5s delay
```

Log-Scale Breathing

The fade engines in the KTD2061 are exponential, which inherently produces consistent fade durations when ramping LED currents from any level to any new level. This relieves the software from having to make complicated fade calculations, as in competing devices.

Exponential fading is symmetrical with respect to measured LED current; however, it appears asymmetrical to the human eye's logarithmic response. To the human eye, exponential fading appears slow and linear for fading down, but much faster for fading up. For fading down, exponential-fading is already the same as log-fading. For this reason, the internal fade-engines easily produce long fade-downs up to 24 seconds (4s time-constant setting x 6 time-constants to reach zero).

For long-duration fade-ups with log shape and very smooth response, use 8 log-scale steps plus with the fade-engines to smooth out the steps. The below Python script is a demonstration of slow log-fade breathing. In this case, a function is defined to step up the current in 8 steps (in octave increments -- each with twice the current of the prior step). Fade-down is still just one step. As an example, the specific steps might be 0µA → 125µA → 250µA → 500µA → 1mA → 2mA → 4mA → 8mA → 16mA → 0µA. Note that competing solutions without integrated fade-engines would need 256 steps for smooth fades in either direction.

The pattern is executed multiple times from slow to fast. The breathing timing is optimized within the function to match to the selected fade-rate.

```
# Python Script for Log-Scale Breathing Pattern
# There are 8 log-steps to ramp up; fade engines smooth it out.
# There is only 1 step to ramp down; exponential ramp down is already log.
def log_breathing(ired0, igrn0, iblu0, cycles, fade):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black (off)
    select_color0()
    delay = 1/64*2**fade                        # adjust delay based on fade-rate
    global_on(fade)
    for i in range(cycles):
        for step in range(9):                       # 8 log-steps fading up
            gain = 2**(8-step)
            set_color0(round(ired0/gain), round(igrn0/gain), round(iblu0/gain))
            time.sleep(delay)
        time.sleep(3*delay)
        set_color0(0x00, 0x00, 0x00)                # 1 step fading down
        time.sleep(6/32*2**fade)                    # wait 6 time-constants

# Execute
log_breathing(0xC0, 0xC0, 0x00, 1, 2)       # color0 = yellow, cycles=1, fade=2
log_breathing(0xC0, 0x00, 0xC0, 1, 3)       # color0 = magenta, cycles=1, fade=3
log_breathing(0xC0, 0x20, 0x00, 1, 4)       # color0 = orange, cycles=1, fade=4
log_breathing(0x00, 0x30, 0xC0, 1, 5)       # color0 = azure, cycles=1, fade=5
log_breathing(0xC0, 0x08, 0x20, 1, 6)       # color0 = fushia, cycles=1, fade=6
log_breathing(0x60, 0xC0, 0x00, 1, 7)       # color0 = chartreuse, cycles=1, fade=7
```

MicroPython main.py Demonstration Script

The below pattern is a complete script for demonstrating the KTD2061 with the MicroPython pyboard. This code may be saved onto the pyboard as the main.py text file, which auto-executes at power-up. At the top of the code, there are commands to import the "time" and "random" modules, set the I^2C bus baudrate to 400kHz, and define a function to translate the i2c_write(SID, regAddr, regData) command into the syntax of the MicroPython's i2c.mem_write(regData, SID, regAddr) command.

While the KTD2061 includes a night mode with 1/16th global brightness, sometimes other levels of global dimming are desired. For this reason, the variable GlobalScale = 1 is used to multiple any I^2C command that writes to the current setting registers. It is applied at the translation function since all I^2C commands pass through this function. For example, set GlobalScale = 0.5 for 50% global dimming for the entire demo script.

```
# main.py -- put your code here!

## from pyb import I2C
##
## i2c = I2C(1, I2C.MASTER, baudrate=100000)
## i2c.scan() # returns list of slave addresses
## i2c.mem_read(2, 0x42, 0x10) # read 2 bytes from slave 0x42, slave memory 0x10
## i2c.mem_write('xy', 0x42, 0x10) # write 2 bytes to slave 0x42, slave memory 0x10

import time
import random

from pyb import I2C
i2c = I2C(1, I2C.MASTER, baudrate=400000)

# translate the i2c write command for Micro Python
GlobalScale = 1
def i2c_write(SID, regAddr, regData):
    if 0x03 <= regAddr <= 0x08:
        regData = round(regData*GlobalScale)
    i2c.mem_write(regData, SID, regAddr)

# Global Core Function Definitions
SID = 0x68                                      # KTD2061 i2c address
en_mode = 2                                     # EnableMode:   1=night, 2=normal(day)
be_en = 1                                       # BrightExtend: 0=disabled, 1=enabled
ce_temp = 3                                     # CoolExtend:   0=135C, 1=120C, 2=105C, 3=90C
on = en_mode*64 + be_en*32 + ce_temp*8          # calculate global on
off = be_en*32 + ce_temp*8                      # calculate global off
def slow_off():
    i2c_write(SID, 0x02, off+7)
def global_reset():
    i2c_write(SID, 0x02, 0xC0)
def global_on(fade):
    i2c_write(SID, 0x02, on+fade)
def global_off(fade):
    i2c_write(SID, 0x02, off+fade)
def set_color0(ired0, igrn0, iblu0):
    i2c_write(SID, 0x05, iblu0)
    i2c_write(SID, 0x04, igrn0)
    i2c_write(SID, 0x03, ired0)
def set_color1(ired1, igrn1, iblu1):
    i2c_write(SID, 0x08, iblu1)
    i2c_write(SID, 0x07, igrn1)
```

```
    i2c_write(SID, 0x06, ired1)
def select_all(isel):
    for reg in range(0x09, 0x0F):
        i2c_write(SID, reg, isel)
def select_off():
    select_all(0x00)
def select_color0():
    select_all(0x88)
def select_color1():
    select_all(0xFF)
def select_colors(isela12, isela34, iselb12, iselb34, iselc12, iselc34, delay):
    i2c_write(SID, 0x09, isela12)
    i2c_write(SID, 0x0A, isela34)
    i2c_write(SID, 0x0B, iselb12)
    i2c_write(SID, 0x0C, iselb34)
    i2c_write(SID, 0x0D, iselc12)
    i2c_write(SID, 0x0E, iselc34)
    time.sleep(delay)
def select_one(reg, data, delay):
    i2c_write(SID, reg, data)
    time.sleep(delay)
def select_one_clear(reg, data, delay):
    select_one(reg, data, delay)
    i2c_write(SID, reg, 0x00)

# Python Script to check that all LEDs are working.  Displays mid white.
def check(fade, delay):
    global_reset()
    select_color0()
    global_on(fade)
    time.sleep(delay)

# Python Script for choosing the color palette
def color_palette(fade, delay):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black
    set_color1(0x80, 0x80, 0x80)                # color1 = white
    select_colors(0x88, 0x89, 0xAB, 0xCD, 0xEF, 0xFF, 0)
    global_on(fade)
    time.sleep(delay)

# Python Script for mimic Amazon Echo Dot gen2 boot pattern
def amazin_boot(cycles):
    global_reset()
    time.sleep(1.7)                             # startup delay (3.7s on some Dots)
    set_color0(0x04, 0xC0, 0xC0)                # color0 = bright cyan
    set_color1(0x00, 0x00, 0x40)                # color1 = dim blue
    select_color1()
    global_on(7)
    time.sleep(3)
    two_color_chasing(cycles, 0.11, 2, 0)       # cyan over blue chasing (11 cycles on Dots)
    slow_off()
    set_color0(0xC0, 0x48, 0x00)                # color0 = amber
    one_color_chasing(cycles, 0.11, 3, 1)       # amber chasing (many cycles on Dots)
    global_off(3)
    time.sleep(5/32*2**3)                       # delay of 5 time-constants

# Python Script for Global Breathing Pattern
def breathing(cycles, fadeOn, timeOn, fadeOff, timeOff):
```

```
    for i in range(cycles):
        global_on(fadeOn)
        time.sleep(timeOn)
        global_off(fadeOff)
        time.sleep(timeOff)

# Python Script for Purple Breathing Pattern
def purple_breathing(cycles):
    slow_off()
    set_color0(0x60, 0x00, 0xC0)               # color0 = purple
    select_color0()
    breathing(cycles, 7, 1.5, 3, 1.5)          # fadeOn=7, timeOn=1.5s, fadeOff=3, timeOff=1.5s

# Python Script for Two-Color Breathing Pattern
def two_color_breathing(cycles, fadeUp, timeUp, fadeDn, timeDn):
    slow_off()
    for i in range(cycles):
        select_color0()
        global_on(fadeUp)
        time.sleep(timeUp)
        slow_off()
        select_color1()
        global_on(fadeDn)
        time.sleep(timeDn)
        slow_off()

# Python Script for Tequila Sunset Two-Color Breathing Pattern
def tequila_sunset_breathing(cycles):
    slow_off()
    set_color0(0xC0, 0x40, 0x00)               # color0 = amber
    set_color1(0x08, 0x00, 0x00)               # color1 = dark red
    two_color_breathing(cycles, 7, 1.5, 3, 1.5) # fadeUp=7, timeUp=1.5s, fadeDn=3, timeDN=1.5s

# Python Script for Double-Rainbow Breathing Pattern
def double_rainbow_breathing(cycles):
    slow_off()
    set_color0(0x00, 0x00, 0x00)               # color0 = black (off)
    set_color1(0x80, 0x80, 0xC0)               # color1 = white (more balanced)
    select_colors(0xB9, 0xDC, 0xEA, 0xB9, 0xDC, 0xEA, 0)
    breathing(cycles, 7, 1.5, 3, 1.5)          # fadeOn=7, timeOn=1.5s, fadeOff=3, timeOff=1.5s

# Python Script for Rainbow Flower Breathing Pattern
def rainbow_flower_breathing(cycles):
    slow_off()
    set_color0(0x00, 0x00, 0x00)        # color0 = black (off)
    set_color1(0x80, 0x80, 0xC0)        # color1 = white (more balanced)
    for i in range(round(cycles/2)):
        select_colors(0xC0, 0xE0, 0xA0, 0xB0, 0x90, 0xD0, 0)
        breathing(1, 6, 0.5, 3, 0.5)    # cycles=1, fadeOn=6, timeOn=0.5s, fadeOff=3, timeOff=0.5s
        select_colors(0x0E, 0x0A, 0x0B, 0x09, 0x0D, 0x0C, 0)
        breathing(1, 6, 0.5, 3, 0.5)    # cycles=1, fadeOn=6, timeOn=0.5s, fadeOff=3, timeOff=0.5s

# Python Script for Generic One-Color Chasing Pattern
def one_color_chasing(cycles, delay, fade, presel):
    slow_off()
    if presel == 1:                             # if presel = 1, pre-select off for all the RBGs
        select_off()
    global_on(fade)
    for i in range(cycles):                     # number of chasing cycles
```

```
        for reg in range(0x09, 0x0F):           # registers 0x09 to 0x0E
            select_one_clear(reg, 0x80, delay)
            select_one_clear(reg, 0x08, delay)

# Python Script for White One-Color Chasing Pattern
def white_chasing(cycles):
    slow_off()
    set_color0(0x80, 0x80, 0xC0)                # color0 = white (more balanced)
    one_color_chasing(cycles, 0.083, 2, 1)      # delay=83ms, fade=2, presel=1

# Python Script for Generic Two-Color Chasing Pattern
def two_color_chasing(cycles, delay, fade, presel):
    slow_off()
    if presel == 1:                             # if presel = 1, pre-select color1 for all the RBGs
        select_color1()
    global_on(fade)
    for i in range(cycles):                     # number of chasing cycles
        for reg in range(0x09, 0x0F):           # registers 0x09 to 0x0E
            select_one(reg, 0x80, delay)        # select color 0 for 1st RGB
            select_one(reg, 0xF8, delay)        # select color 1 for 1st RGB, color 0 for 2nd RGB
            i2c_write(SID, reg, 0xFF)           # select color 1 for 1st and 2nd RGB

# Python Script for White over Red Two-Color Chasing Pattern
def white_red_chasing(cycles):
    slow_off()
    set_color0(0x80, 0x80, 0xC0)                # color0 = white (more balanced)
    set_color1(0x10, 0x00, 0x00)                # color1 = dim red
    two_color_chasing(cycles, 0.083, 2, 0)      # delay=83ms, fade=2, presel=0
    # global_off(2)
    # time.sleep(5/32*2**2)                     # delay of 5 time-constants

# Python Script for Generic Dual-Chasing Pattern
def dual_chasing(cycles, delay, fade):
    global_on(fade)
    for i in range(cycles*2):
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0x80)
            select_one_clear(reg+3, 0x80, delay)
            i2c_write(SID, reg, 0x08)
            select_one_clear(reg+3, 0x08, delay)
            i2c_write(SID, reg, 0x00)

# Python Script for Korean Yin-Yang Chasing Pattern
def Kor_YinYang(cycles, delay, fade):
    slow_off()
    set_color0(0xC0, 0x00, 0x00)
    set_color1(0x00, 0x00, 0xC0)
    select_off()
    global_on(fade)
    for i in range(cycles):
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0x80)
            select_one(reg+3, 0xF0, delay)
            i2c_write(SID, reg, 0x08)
            select_one_clear(reg+3, 0x0F, delay)
            i2c_write(SID, reg, 0x00)
        for reg in range(0x09, 0x0C):
            i2c_write(SID, reg, 0xF0)
            select_one(reg+3, 0x80, delay)
```

```
            i2c_write(SID, reg, 0x0F)
            select_one_clear(reg+3, 0x08, delay)
            i2c_write(SID, reg, 0x00)
    global_off(fade)
    time.sleep(5/32*2**fade)                    # delay of 5 time-constants

# Python Script for Audi-like turn-signal Matrix Water Flow Pattern
# This might be a turn-signal pattern when unlocking the doors with keyfob.
def matrix_water_flow(cycles):
    global_reset()
    set_color0(0x80, 0x30, 0x00)                # color0 = amber
    set_color1(0x40, 0x40, 0x60)                # color1 = white (more balanced)
    for j in range(cycles):                     # number of pattern cycles
        # Flash Twice using White
        select_color1()
        breathing(2, 0, 0.2, 0, 0.2)
        time.sleep(0.25)
        # Matrix Water Flow using Amber
        for i in range(3):                      # number of matrix cycles
            select_off()
            global_on(2)
            for reg in range(0x0E, 0x08, -1):   # registers 0x09 to 0x0E
                select_one(reg, 0x08, 0.03)     # select Color 0 for one RGB
                select_one(reg, 0x88, 0.03)     # select Color 0 for both RGBs
            time.sleep(0.25)                    # 250ms delay hold time
            global_off(0)
            time.sleep(0.35)

# Python Script for Rotating Rainbow Pattern
# It only uses red, green, and blue, but fade engine creates intermediary colors.
def rainbow_swirl(cycles, delay, fade):
    slow_off()
    set_color0(0x04, 0x04, 0x04)                # color0 = black (off)
    set_color1(0x80, 0x70, 0xC0)                # color1 = white (more balanced)
    global_on(fade)
    for i in range(cycles):
        select_colors(0xCC, 0xCC, 0xAA, 0xAA, 0x99, 0x99, delay)
        select_colors(0x9C, 0xCC, 0xCA, 0xAA, 0xA9, 0x99, delay)
        select_colors(0x99, 0xCC, 0xCC, 0xAA, 0xAA, 0x99, delay)
        select_colors(0x99, 0x9C, 0xCC, 0xCA, 0xAA, 0xA9, delay)
        select_colors(0x99, 0x99, 0xCC, 0xCC, 0xAA, 0xAA, delay)
        select_colors(0xA9, 0x99, 0x9C, 0xCC, 0xCA, 0xAA, delay)
        select_colors(0xAA, 0x99, 0x99, 0xCC, 0xCC, 0xAA, delay)
        select_colors(0xAA, 0xA9, 0x99, 0x9C, 0xCC, 0xCA, delay)
        select_colors(0xAA, 0xAA, 0x99, 0x99, 0xCC, 0xCC, delay)
        select_colors(0xCA, 0xAA, 0xA9, 0x99, 0x9C, 0xCC, delay)
        select_colors(0xCC, 0xAA, 0xAA, 0x99, 0x99, 0xCC, delay)
        select_colors(0xCC, 0xCA, 0xAA, 0xA9, 0x99, 0x9C, delay)
    global_off(3)
    time.sleep(5/32*2**3)                       # delay of 5 time-constants

# Python Script to copy competitor's demo video
def texas_2_step():
    slow_off()
    # Breathing with diffent colors
    select_color0()
    set_color0(0xC0, 0x00, 0x00)        # color0 = red
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0x00, 0xC0, 0x00)        # color0 = green
```

```
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0x00, 0x00, 0xC0)        # color0 = blue
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0xC0, 0xC0, 0x00)        # color0 = yellow
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0xC0, 0x00, 0xC0)        # color0 = magenta
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0x00, 0xC0, 0xC0)        # color0 = cyan
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    set_color0(0xC0, 0xC0, 0xC0)        # color0 = white
    breathing(1, 7, 1, 3, 1.5)          # cycles=1, fadeOn=7, timeOn=1s, fadeOff=3, timeOff=1.5s
    # Color-to-color fades
    for i in range(2):                      # repeat these fades twice
        set_color0(0x00, 0x00, 0xC0)        # color0 = blue
        global_on(6)
        time.sleep(1)
        i2c_write(SID, 0x04, 0xC0)          # color0 = cyan (turn on the green)
        time.sleep(1)
        i2c_write(SID, 0x04, 0x00)          # color0 = blue (turn off the green)
        global_on(3)
        time.sleep(1.5)
        global_on(6)
        i2c_write(SID, 0x03, 0xC0)          # color0 = magenta (turn on the red)
        time.sleep(1)
        i2c_write(SID, 0x03, 0x00)          # color0 = blue (turn off the red)
        global_on(3)
        time.sleep(1.5)
        global_off(2)
        time.sleep(1)
    # Dual Chasing Patterns
    slow_off()
    select_off()
    set_color0(0xC0, 0x00, 0x00)            # color0 = red
    dual_chasing(3, 0.083, 1)               # cycles=3, delay=83ms, fade=1
    set_color0(0x04, 0xC0, 0x04)            # color0 = green
    dual_chasing(3, 0.083, 1)               # cycles=3, delay=83ms, fade=1
    set_color0(0x04, 0x04, 0xC0)            # color0 = blue
    dual_chasing(3, 0.083, 1)               # cycles=3, delay=83ms, fade=1
    # Rotating Multi-color
    slow_off
    set_color0(0x00, 0x00, 0x00)
    set_color1(0xC0, 0xC0, 0x40)
    delay=0.083
    global_on(1)
    for i in range(4):
        for reg in range(0x09, 0x0F):
            regp = reg + 3
            if regp > 0x0E:
                regp = reg - 3
            i2c_write(SID, reg, 0xA9)
            select_one(regp, 0xE9, delay)
            i2c_write(SID, reg, 0x9A)
            select_one(regp, 0x9E, delay)
            i2c_write(SID, reg, 0x99)
            i2c_write(SID, regp, 0x99)
    # Dual Chasing Red w/reverse
    global_off(2)
    select_off()
    set_color0(0xC0, 0x00, 0x00)             # color0 = red
```

```
    for i in range(3):
        dual_chasing(2, 0.083, 2)               # cycles=2, delay=83ms, fade=2
        i2c_write(SID, 0x09, 0x80)
        select_one_clear(0x0C, 0x80, delay)
        i2c_write(SID, 0x09, 0x00)
        for j in range(3):
            for reg in range(0x0B, 0x08, -1):
                i2c_write(SID, reg, 0x08)
                select_one(reg+3, 0x08, delay)
                i2c_write(SID, reg, 0x80)
                select_one_clear(reg+3, 0x80, delay)
                i2c_write(SID, reg, 0x00)
    time.sleep(0.5)
    # Water-Flow Dual Green w/reverse
    global_off(2)
    select_off()
    set_color0(0x00, 0x00, 0x00)
    set_color1(0xC0, 0xC0, 0xC0)
    global_on(1)
    for i in range(3):
        time.sleep(delay)
        select_colors(0x00, 0xAA, 0x00, 0x00, 0x00, 0x00, delay)
        select_colors(0x0A, 0xAA, 0xA0, 0x00, 0x00, 0x00, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0x00, 0x00, 0x00, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0xA0, 0x00, 0x0A, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0xAA, 0x00, 0xAA, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, delay*2)
        select_colors(0xAA, 0xAA, 0xAA, 0xAA, 0x00, 0xAA, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0xA0, 0x00, 0x0A, delay)
        select_colors(0xAA, 0xAA, 0xAA, 0x00, 0x00, 0x00, delay)
        select_colors(0x0A, 0xAA, 0xA0, 0x00, 0x00, 0x00, delay)
        select_colors(0x00, 0xAA, 0x00, 0x00, 0x00, 0x00, delay)
        select_off()
        time.sleep(4*delay)
    # Water-Flow Red w/reverse
    for i in range(3):
        time.sleep(delay)
        select_one(0x0A, 0x0C, delay)
        select_one(0x0B, 0xC0, delay)
        select_one(0x0B, 0xCC, delay)
        select_one(0x0C, 0xC0, delay)
        select_one(0x0C, 0xCC, delay)
        select_one(0x0D, 0xC0, delay)
        select_one(0x0D, 0xCC, delay)
        select_one(0x0E, 0xC0, delay)
        select_one(0x0E, 0xCC, delay)
        select_one(0x09, 0xC0, delay)
        select_one(0x09, 0xCC, delay)
        select_one(0x0A, 0xCC, delay*2)
        select_one(0x0A, 0x0C, delay)
        select_one(0x09, 0xC0, delay)
        select_one(0x09, 0x00, delay)
        select_one(0x0E, 0xC0, delay)
        select_one(0x0E, 0x00, delay)
        select_one(0x0D, 0xC0, delay)
        select_one(0x0D, 0x00, delay)
        select_one(0x0C, 0xC0, delay)
        select_one(0x0C, 0x00, delay)
        select_one(0x0B, 0xC0, delay)
```

```
        select_one(0x0B, 0x00, delay)
        select_one(0x0A, 0x00, delay*3)

# Python Script for Purple Raindrops Pattern
def purple_raindrops(cycles, delay, fade):
    slow_off()
    set_color0(0x60, 0x08, 0xC0)                    # color0 = bright purple
    select_off()
    global_on(fade)
    for i in range(cycles):
        reg = random.randint(0x09, 0x0E)            # random selection register
        regp = reg + 1
        regm = reg - 1
        if regp == 0x0F:
            regp = 0x09
        if regm == 0x08:
            regm = 0x0E
        select_one(reg, 0x80, delay*2)              # select color 0 for reg 1st RGB
        select_one(reg, 0x08, 0)                    # select color 0 for reg 2nd RGB
        select_one(regm, 0x08, delay/4)             # select color 0 for regm 2nd RGB
        select_one(reg, 0x00, 0)                    # select off for reg 2nd RGBs
        select_one(regm, 0x00, delay)               # select off for regm 2nd RGB
        select_one(regp, 0x80, 0)                   # select color 0 for regp 1st RGB
        select_one(regm, 0x80, delay/16)            # select color 0 for regm 1st RGB
        select_one(regp, 0x00, 0)                   # select off for regp 1st RGB
        select_one(regm, 0x00, 0)                   # select off for regm 1st RGB
        time.sleep(random.random())                 # random delay
        reg = random.randint(0x09, 0x0E)            # random selection register
        regp = reg + 1
        regm = reg - 1
        if regp == 0x0F:
            regp = 0x09
        if regm == 0x08:
            regm = 0x0E
        select_one(reg, 0x08, delay*2)              # select color 0 for reg 2nd RGB
        select_one(reg, 0x80, 0)                    # select color 0 for reg 1st RGB
        select_one(regp, 0x80, delay/4)             # select color 0 for regp 1st RGB
        select_one(reg, 0x00, 0)                    # select off for reg 1st RGB
        select_one(regp, 0x00, delay)               # select off for regp 1st RGB
        select_one(regm, 0x08, 0)                   # select color 0 for regm 2nd RGB
        select_one(regp, 0x08, delay/16)            # select color 0 for regp 2nd RGB
        select_one(regm, 0x00, 0)                   # select off for regm 2nd RGB
        select_one(regp, 0x00, 0)                   # select off for regp 2nd RGB
        time.sleep(random.random())                 # random delay

# Python Script for Random Lighting Pattern
def random_pattern(cycles, delay, fade, mode):
    global_on(fade)
    for i in range(cycles):
        iset = 2**random.randint(1, 7)              # random Iset in octaves
        reg = random.randint(0x03, 0x08)            # random color register
        i2c_write(SID, reg, iset)                   # random Iset to random reg
        if mode == 0:                               # mode = 0, random on/off + random color
            data = random.randint(0, 255)               # random selection data
        elif mode == 1:                             # mode = 1, fixed on only + random color
            data = 16*random.randint(8, 15) + random.randint(8, 15)
        reg = random.randint(0x09, 0x0E)            # random selection register
        i2c_write(SID, reg, data)                   # random data to random reg
        time.sleep(delay)                           # delay
```

```
# Python Script for Log-Scale Breathing Pattern
# There are 8 log-steps to ramp up; fade engines smooth it out.
# There is only 1 step to ramp down; exponential ramp down is already log.
def log_breathing(ired0, igrn0, iblu0, cycles, fade):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black (off)
    select_color0()
    delay = 1/64*2**fade                        # adjust delay based on fade-rate
    global_on(fade)
    for i in range(cycles):
        for step in range(9):                   # 8 log-steps fading up
            gain = 2**(8-step)
            set_color0(round(ired0/gain), round(igrn0/gain), round(iblu0/gain))
            time.sleep(delay)
        time.sleep(3*delay)
        set_color0(0x00, 0x00, 0x00)            # 1 step fading down
        time.sleep(6/32*2**fade)                # wait 6 time-constants

# Python Script to make a rainbow with 12 independent colors using fade engines and 50% duty
def twelve_color_rainbow(cycles, delay, fade):
    slow_off()
    set_color0(0x00, 0x00, 0x00)                # color0 = black (off)
    select_off()
    global_on(fade)
    for i in range(cycles):
        set_color1(0xC0, 0xC0, 0xC0)            # color1 = white
        select_colors(0xCC, 0xEA, 0xAA, 0xB9, 0x99, 0xDC, delay/2)
        select_off()
        set_color1(0x30, 0x30, 0x30)            # color1 = grey (6mA)
        select_colors(0x8A, 0x8C, 0x89, 0x8A, 0x8C, 0x89, delay/2)
        select_off()

# Python Script to make 12 fully-independent colors using fade engines and 1/6th duty
# As set below, the colors are 12 rainbow colors.  It can also make chasing within the colors.
def twelve_colors(cycles, delay, fade):
    slow_off()
    select_off()
    global_on(fade)
    for i in range(cycles):
        set_color0(0xC0, 0x00, 0x00)
        set_color1(0xC0, 0x30, 0x00)
        select_colors(0x8F, 0x00, 0x00, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0xC0, 0xC0, 0x00)
        set_color1(0x30, 0xC0, 0x00)
        select_colors(0x00, 0x8F, 0x00, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0xC0, 0x00)
        set_color1(0x00, 0xC0, 0x30)
        select_colors(0x00, 0x00, 0x8F, 0x00, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0xC0, 0xC0)
        set_color1(0x00, 0x30, 0xC0)
        select_colors(0x00, 0x00, 0x00, 0x8F, 0x00, 0x00, delay/6)
        select_off()
        set_color0(0x00, 0x00, 0xC0)
        set_color1(0x30, 0x00, 0xC0)
        select_colors(0x00, 0x00, 0x00, 0x00, 0x8F, 0x00, delay/6)
```

```
        select_off()
        set_color0(0xC0, 0x00, 0xC0)
        set_color1(0xC0, 0x00, 0x30)
        select_colors(0x00, 0x00, 0x00, 0x00, 0x00, 0x8F, delay/6)
        select_off()

# Python Script to execute the twelve_colors function, and then ramp the fade rate to reveal
chasing.
def chasing_rainbow():
    for fade in range(7, 0, -1):
        twelve_colors(1, 1, fade)
    for fade in range(0, 8):
        twelve_colors(1, 1, fade)
    global_off(4)
    time.sleep(3)

# Execute all the above defined patterns
while True:
    # check(5, 4)                             # functional check (mid white), fade=5, delay=4s
    # color_palette(5, 4)                     # displays color palette, fade=5, delay=4s
    amazin_boot(5)                          # Echo boot, but smoother, 11 cycles used on Echo
    purple_breathing(3)                     # one-color breathing, cycles=3
    tequila_sunset_breathing(3)             # two-color breathing, cycles=3
    double_rainbow_breathing(3)             # many-color breathing, cycles=3
    rainbow_flower_breathing(8)             # many-color breathing, cycles=8
    white_chasing(4)                        # one-color chasing, cycles=4
    white_red_chasing(5)                    # two-color chasing, cycles=5
    Kor_YinYang(5, 0.083, 2)                # yin-yang chasing, cycles=5, delay=83ms, fade=2
    twelve_color_rainbow(1, 1, 7)           # cycles=1, delay=1s, fade=7
    chasing_rainbow()                       # 12 color rainbow exhibition pattern
    texas_2_step()                          # similar to TI demo for LP5024/36
    matrix_water_flow(2)                    # automotive turn-signal, cycles=2
    rainbow_swirl(5, 0.2, 3)                # rotating rainbow, cycles=5, delay=200ms, fade=3
    purple_raindrops(15, 0.05, 2)           # cycles=15, delay=50ms, fade=2
    random_pattern(100, 0.2, 2, 0)          # for music, cycles=100, delay=200ms, fade=2, mode=0
    random_pattern(40, 1, 5, 1)             # for meditation, cycles=40, delay=1s, fade=5, mode=1
    global_off(5)                           # global off at 1s fade-rate
    time.sleep(5)                           # 5s delay
    log_breathing(0xC0, 0xC0, 0x00, 1, 2)   # color0 = yellow, cycles=1, fade=2
    log_breathing(0xC0, 0x00, 0xC0, 1, 3)   # color0 = magenta, cycles=1, fade=3
    log_breathing(0xC0, 0x20, 0x00, 1, 4)   # color0 = orange, cycles=1, fade=4
    log_breathing(0x00, 0x30, 0xC0, 1, 5)   # color0 = azure, cycles=1, fade=5
    log_breathing(0xC0, 0x08, 0x20, 1, 6)   # color0 = fushia, cycles=1, fade=6
    log_breathing(0x60, 0xC0, 0x00, 1, 7)   # color0 = chartreuse, cycles=1, fade=7
```