NQM UV LIGHT ENGINE
SOFTWARE PROGRAMMING GUIDE

Version 1.1

Date: 01/07/2019

Content

Revision History

Date	Revision	Changes and Additions	Page	By
2018/11/20	1.0	First Draft	all	Richard
2019/01/04	1.1	1. modify the recommended display timing of 3840x2160 2. supplement the processing time before the system is ready to use 3. modify the index of the I2C command of setting LED DAC 4. modify the LED Current range that the DAC values are mapped to 5. modify the way to set projector source and test pattern 6. modify the parameter of the motor control 7. modify the way to read the Application version 8. revise the SPI Flash memory configuration	5, 9, 10, 12, 14, 15, 18	Richard

1 OVERVIEW

This software programming guide is based on our control utility and introduces all the functions we used and their respective implementation.

2 PRELIMINARY REQUIREMENT

Our control utility is connecting to the mainboard of NQM UV Light Engine via the USB interface. On the mainboard, we use a Cypress chip to transfer the incoming USB commands into the I2C interface, to communicate with TI DLP Controller DDP4422. In additions, we use some GPIO pins provided by this Cypress chip, to communicate with DDP4422 as well. Therefore, our functions are all built on Cypress API functions, and you have to install Cypress SDK Library first if you'd like to develop your own control software. For the details, please visit Cypress Semiconductor website at http://www.cypress.com/documentation/software-and-drivers/usb-serial-software-development-kit.

3 IMPORTANT NOTICE BEFORE PROGRAMMING

3.1 Input Signal

The resolution of projection image will depend on the input signal, which is accepted as only the following two sets of display timings at this moment：

	Resolution	Pixel Clock	H.Active Pixels	H.Blank	H.Front Porch	H.Sync Width	H.Clock (kHz)	V.Active Lines	V.Blank	V.Front Porch	V.Sync Width	V.Clock (Hz)
1	3840x2160	148.50	3840	160	48	32	32.62	2160	16	3	5	14.99
2	2712x1528	133.50	2712	160	48	32	46.48	1528	22	3	5	29.99

4 CYPRESS API FUNCTION INTRODUCTION

4.1 GPIO

CypressGpio mCypressGpio;

int mCpressGpio.Open()

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCypressGpio.Read(byte GpioNum, ref byte Value)

input:

GpioNum : the index of the GPIO pin provided by Cypress (0~18)[2]
Value : the value of the GPIO pin provided by Cypress (0 or 1)

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCypressGpio.Write(byte GpioNum, byte Value)

input:

GpioNum : the index of the GPIO pin provided by Cypress (0~18)[2]
Value : the value of the GPIO pin provided by Cypress (0 or 1)

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressGpio.Close()

return:

0 means the execution is successful
-N[1] means the execution is failed

[1] *The negative numbers show different error cases, please refer to Cypress SDK documentation for the details.*

[2] *The indexes of the GPIO pins that we have to read are only 1 (I2C_BUSY) and 15 (ASIC_READY). The indexes of the GPIO pins that we have to written are only 0 (RESETZ) and 14 (LED_ON_OFF).*

4.2 I2C

CypressI2c mCypressI2c;

int mCpressI2c.Open()

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressI2c.Read(byte cSlaveAddress7bit, ref byte[] recvBuf, int cReadsize)

input:

cSlaveAddress7bit : the I2C slave address to be taken access to
sendBuf : the buffer of the data to be read back
cWriteSize : the buffer size of the data to be read back

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressI2c.Write(byte cSlaveAddress7bit, byte[] sendBuf, int cWriteSize)

input:

cSlaveAddress7bit : the I2C slave address to be taken access to
sendBuf : the buffer of the data to be sent out
cWriteSize : the buffer size of the data to be sent out

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressI2c.Close()

return:

0 means the execution is successful
-N[1] means the execution is failed

[1] *The negative numbers show different error cases, please refer to Cypress SDK documentation for the details.*

4.3 SPI

CypressSpi mCypressSpi;

int mCpressSpi.Open()

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressSpi.ReadWrite(ref byte[] rdbuf, byte[] wrbuf, int len)

input:

rdbuf : the buffer of the data to be read back
wrbuf : the buffer of the data to be sent out
len : the buffer size of the data to be read back or to be sent out

return:

0 means the execution is successful
-N[1] means the execution is failed

int mCpressSpi.Close()

return:

0 means the execution is successful
-N[1] means the execution is failed

[1] *The negative numbers show different error cases, please refer to Cypress SDK documentation for the details.*

5 FUNCTION IMPLEMENTATION (GPIO/I2C INTERFACE)

5.1 Projector On/Off

Set GPIO-0 (*RESETZ*) to 1 to turn on the projector (controller), then we have to keep polling GPIO-15 (*ASIC_READY*) until its value is turned into 1, before taking any further access to the GPIO/I2C interface. The processing time normally takes about 14 seconds.

Clear GPIO-0 (*RESETZ*) to 0 to turn off the projector (controller). It is not equal to power off the projector, and we ***have to*** clear it off before shutdown, to make sure that the DMD is safely parked.

5.2 LED On/Off

Set GPIO-14 (*LED_ON_OFF*) to 1 to turn on the UV LED; clear it to 0 to turn off LED.

5.3 LED DAC

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

(1) Set

- Apply the I2C Write command (**0xD1**)：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]

  1st Byte = Command = 0xD1

  2nd Byte = Data [0] = (the MSB of the LED DAC value)

  3rd Byte = Data [1] = (the LSB of the LED DAC value)

  The data size is 3.

Note: The Brightness of NQM UV Light Engine is proportional to the LED Current, which is represented by the LED DAC value of 0 and 50 ~ 1000, mapping to 0 and 1.25 ~ 25.0 (Amp). The LED DAC value is calculated by the formula:

dac = 1000 * (LED Current value) / 25.0

For example, to set the LED Current value into 15 Amp, you have to set the DAC value into 1000*15.0/25.0 = 600 (0x0258).

(2) Get

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]

  1st Byte = Command = 0x15

  2nd Byte = Data [0] = 0xD1

  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → to read back the MSB of the LED DAC value
  2nd Byte = Data [1] → to read back the LSB of the LED DAC value
  The data size is 2.

Note: The LED Current value is calculated by the following formula：

dac = (Data[0] << 8) + Data[1]
current = 25.0 * dac / 1000

5.4 Projector Source and Test Pattern

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

(1) HDMI

Apply the I2C Write command (**0x01**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0x01
2nd Byte = Data [0] = 0x00 (0x00：External source)
The data size is 2.

(2) Ramp (256-level horizontal ramp)

- Apply the I2C Write command (**0x01**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x01
  2nd Byte = Data [0] = 0x01 (0x01：Test pattern)
  The data size is 2.

- Apply the I2C Write command (**0x11**) next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x11
  2nd Byte = Data [0] = 0x01 (0x01：Horizontal Ramp)
  The data size is 2.

(3) Checkerboard

- Apply the I2C Write command (**0x01**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x01
  2nd Byte = Data [0] = 0x01 (0x01：Test pattern)
  The data size is 2.

- Apply the I2C Write command (**0x11**) next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x11
  2nd Byte = Data [0] = 0x07 (0x07：Checkerboard)

The data size is 2.

(4) SolidField

- Apply the I2C Write command (**0x13**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x13
  2nd Byte = Data [0] = 0xFF
  3rd Byte = Data [1] = 0x03
  4th Byte = Data [2] = 0xFF
  5th Byte = Data [3] = 0x03
  6th Byte = Data [4] = 0xFF
  7th Byte = Data [5] = 0x03
  The data size is 7.

- Apply the I2C Write command (**0x01**) next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x01
  2nd Byte = Data [0] = 0x02 (0x02：Solid Field)
  The data size is 2.

5.5 Fan Speed

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

(1) DMD Fan

Apply the I2C Write command (**0xEB**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xEB
2nd Byte = Data [0] = (DMD Fan Speed (Duty) Setting, 0~100)
The data size is 2.

(2) LED Fan 1

Apply the I2C Write command (**0xEC**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xEC
2nd Byte = Data [0] = (LED Fan 1 Speed (Duty) Setting, 0~100)
The data size is 2.

(3) LED Fan 2

Apply the I2C Write command (**0xED**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xED
2nd Byte = Data [0] = (LED Fan 2 Speed (Duty) Setting, 0~100)
The data size is 2.

5.6 Motor Control

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

(1) Front Group Motor

(1.1) Direction

Apply the I2C Write command (**0xB5**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xB5
2nd Byte = Data [0] = 0 or 1 (0: Up, 1: Down)
The data size is 2.

(1.2) Step Size

Apply the I2C Write command (**0xB6**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xB6
2nd Byte = Data [0] = (the LSB of the step size)
3rd Byte = Data [1] = (the MSB of the step size)
4th Byte = Data [2] = 0x32
5th Byte = Data [3] = 0x00
The data size is 5.

(2) Rear Group Motor

(2.1) Direction

Apply the I2C Write command (**0xBA**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xBA
2nd Byte = Data [0] = 0 or 1 (0: Up, 1: Down)
The data size is 2.

(2.2) Step Size

Apply the I2C Write command (**0xBB**)：

Command format:

Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
1st Byte = Command = 0xBB
2nd Byte = Data [0] = (the LSB of the step size)
3rd Byte = Data [1] = (the MSB of the step size)
4th Byte = Data [2] = 0x64
5th Byte = Data [3] = 0x00
The data size is 5.

5.7 Sequence Revision

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0xBE
  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → (to read back the LSB of the sequence revision)
  2nd Byte = Data [1]
  3rd Byte = Data [2]
  4th Byte = Data [3] → (to read back the MSB of the sequence revision)
  The data size is 4.

  Note: The 4 bytes of data are stored by the little-endian format.

5.8 Application Version

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0x85
  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → (to read back Patch.LSB of the application version)
  2nd Byte = Data [1] → (to read back Patch.MSB of the application version)
  3rd Byte = Data [2] → (to read back Minor of the application version)
  4th Byte = Data [3] → (to read back Major of the application version)
  The data size is 4.

5.9 Temperature Sensor

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

(1) DMD

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0x9C
  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → (to read back the DMD temperature value)
  The data size is 1.

(2) LED (Junction)

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0x9F
  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data[0] → (to read back the fraction/LSB of LED temperature)
  2nd Byte = Data[1] → (to read back the fraction/MSB of LED temperature)
  3rd Byte = Data[2] → (to read back the integer/LSB of LED temperature)
  4th Byte = Data[3] → (to read back the integer/MSB of LED temperature)
  The data size is 4.

Note: The LED temperature value is a floating number, which is composed of 2 bytes of the integral number and 2 bytes of the fractional number. It is calculated as：

$$T_{TH} = (Data[3] << 8) + Data[2] + (float)((Data[1] << 8) + Data[0]) / 65536$$

(3) LED Driver Board

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0x9E
  The data size is 2.

- Apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → (to read back the LSB of the temperature value)
  2nd Byte = Data [1] → (to read back the MSB of the temperature value)
  The data size is 2.

Note: The LED Driver temperature value is a floating number, which is composed of 2 bytes of data with the following configuration:

MSB (Data[1])								LSB (Data[0])							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
all-zero				decimal				decimal				fraction			

This floating number is calculated as

((Data[1] & 0x0F) << 4) + ((Data[0] & 0xF0) >> 4) + (float)(Data[0] & 0x0F) / 16

5.10 Light Sensor

Read GPIO-1 (I2C_BUSY) first. If its value is 0, which means the I2C bus is busy and we cannot send any I2C command now -- must retry it later; if its value is 1, then we can go on the following I2C command.

- Apply the I2C Write command (**0x15**) first：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Command = 0x15
  2nd Byte = Data [0] = 0xF7
  The data size is 2.

- Wait for 1 second, and then apply the I2C Read command next：

  Command format:

  Slave device address = 0x1A [0x35 (Read) / 0x34 (Write)]
  1st Byte = Data [0] → (to read back the LSB of the light sensor value)
  2nd Byte = Data [1] → (to read back the MSB of the light sensor value)
  The data size is 2.

6 SPI SERIAL FLASH MEMORY INTERFACE

6.1 Uniformity Mask File

- The mask file of native resolution (2712x1528) is saved as the PNG file format, and stored in the serial flash memory, starting at offset 0x10000, with the following configuration:

	starting offset	size
the length of the mask file[1]	0x10000	4
the checksum of the mask file	0x10004	4
the mask file itself	0x10008	TBD[1]

[1] The length of the mask file is stored at offset 0x10000, 4 bytes in total, with the little-endian format, and it will be used to retrieve the mask file from 0x10008.

- The mask file of 4K_UHD resolution (3840x2160) is also saved as the PNG file format, and stored in the same flash memory, starting at offset 0x30000, with the following configuration:

	starting offset	size
the length of the mask file[2]	0xA0000	4
the checksum of the mask file	0xA0004	4
the mask file itself	0xA0008	TBD[2]

[2] The length of the mask file is stored at offset 0xA0000, 4 bytes in total, with the little-endian format, and it will be used to retrieve the mask file from 0xA0008.

6.2 Optical Examination Statistics

Besides the mask files, we recorded some optical examination statistics in the same flash memory, with the following configuration:

	starting offset	size
Serial Number	0x00000	20
LED Current (DAC) value	0x00030	2
LED (Junction) Temperature Sensor Readings	0x00034	2
Light Sensor Readings	0x00038	2