TI Information — Selective Disclosure

Piccolo Software Programmer's Guide for the DLPC120 ASIC

User's Guide

Literature Number: DLPU061
March 2018

Contents

List of Figures

Introduction

Trademarks

DLP is a registered trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

Introduction

The Piccolo LED Controller is designed to cost effectively support automotive DLP® technology LED applications requiring precise control of color and brightness over a wide dimming range. The Piccolo LED Controller is a sub-system level reference design that leverages many unique features of the Piccolo TMS320F28023 MCU, plus some special companion functions designed into the automotive DLPC120 DMD controller.

The reference system block diagram is shown in Figure 1.

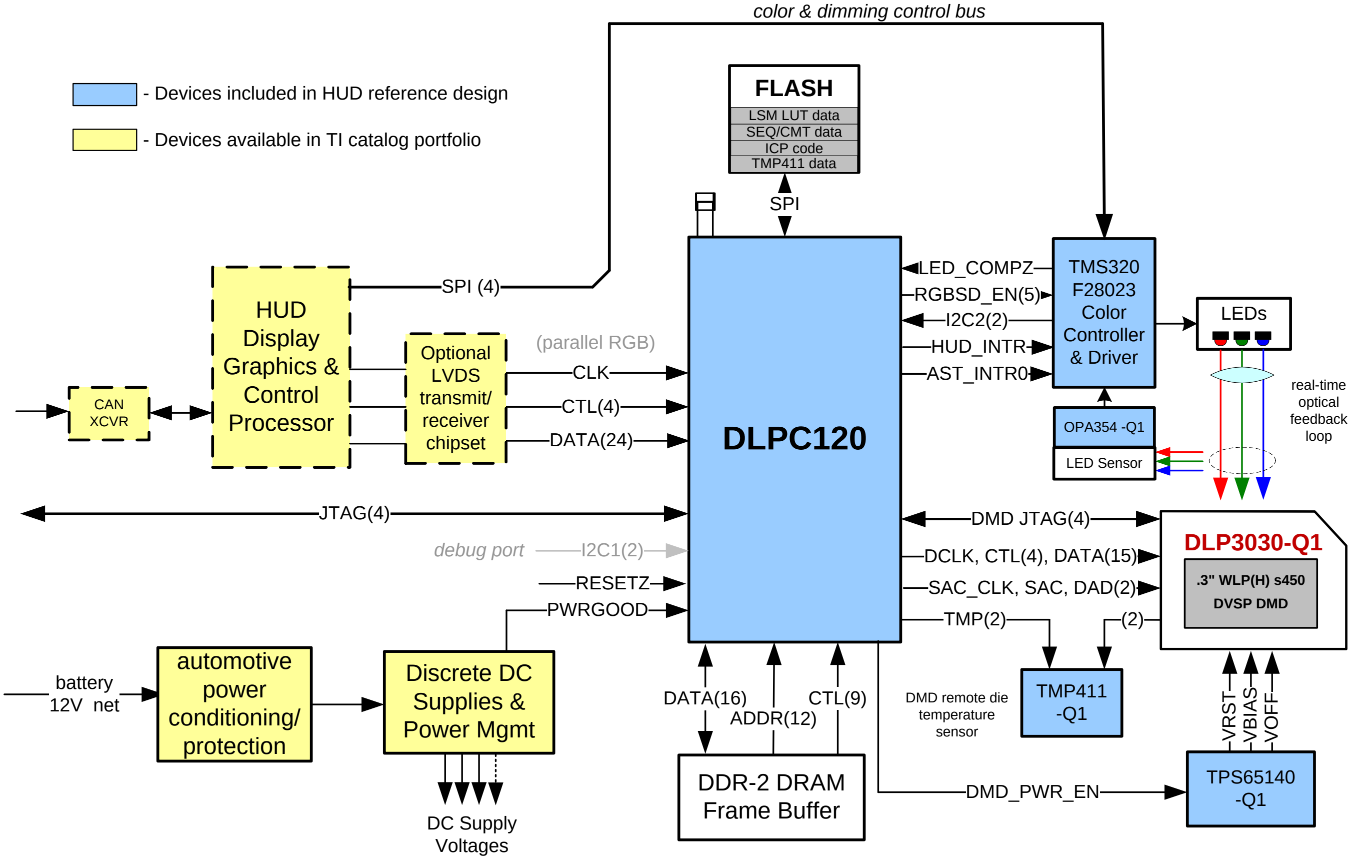


Figure 1. HUD System Block Diagram

The DLPC120 performs all image processing and display control to support the DLP3030-Q1 DMD. The DLPC120 accepts parallel video input from the HUD Display Graphics and Control Processor which acts as the display systems master.

The Piccolo MCU receives brightness/dimming commands from the display controller via a SPI bus input. Piccolo SW interprets these commands and coordinates all adjustments within the sub-system including setting the DLPC120 Index tables, adjusting the LED PWM level, and setting the sensor gain. The Piccolo configures the DLPC120 over a dedicated I2C bus (I2C2 in Figure 1).

Purpose & Scope

This document describes the operations of the Piccolo microcontroller software which accepts the dimming commands over the SPI bus and controls the DLPC120 ASIC and the LED PWMs to achieve requested brightness while maintaining the color point of the system. Also, this document describes other aspects of the SW implementation of the dimming algorithm that include system calibration, temperature compensation, and other peripheral system features.

Applicable Documents

The following documents and drawings should be referenced for more complete understanding the Piccolo based LED driver dimming solution.

1.	DLPA084	LED Driver for DLP3030-Q1 Displays Application Note
2.	DLPU057	DLP3030-Q1 Head-Up Display Piccolo SPI User's Guide
3.	2514132	HUD Controller board and LED driver Schematic
5.	2512174	0.3” WVGA DMD board Schematic
6.	DLPS076	DLP3030-Q1 Datasheet
7.	DLPU055	I2C Programmer’s Guide for DLPC120 ASIC

Terms and Abbreviations

DLPC – DLP Controller

DMD – Digital Micro-mirror Device

ASIC – Application Specific Integrated Circuit

CMT – Contour Mitigation Table

SEQ – Sequence

LUT – Look Up Table

LED – Light Emitting Diode

RGB – Red Green Blue

LDC – LED Drive Control

Piccolo Microcontroller Software

1.1 Piccolo Microcontroller Software

Figure 1-1 shows a model of the Piccolo software for the HUD Reference system. The Piccolo software has two parts: Bootloader and Main Application.

The bootloader enables the main application to be verified, programmed or reprogrammed in the system via the SPI port. Refer to Chapter 2 for more details on bootloader.

The Piccolo main application accepts external events such as commands coming from SPI and interrupt from ASIC. Based on these events it does a variety of actions to achieve the desired function. For example, when a dimming command is received through the SPI bus, the SPI command parser analyses and passes the value to dimming algorithm. On an ASIC interrupt, it applies the calculated set of dimming parameters through I2C back to ASIC. The LED PWMs and sensor gain GPIOs will then be updated upon indication from ASIC on completion of sequence update.

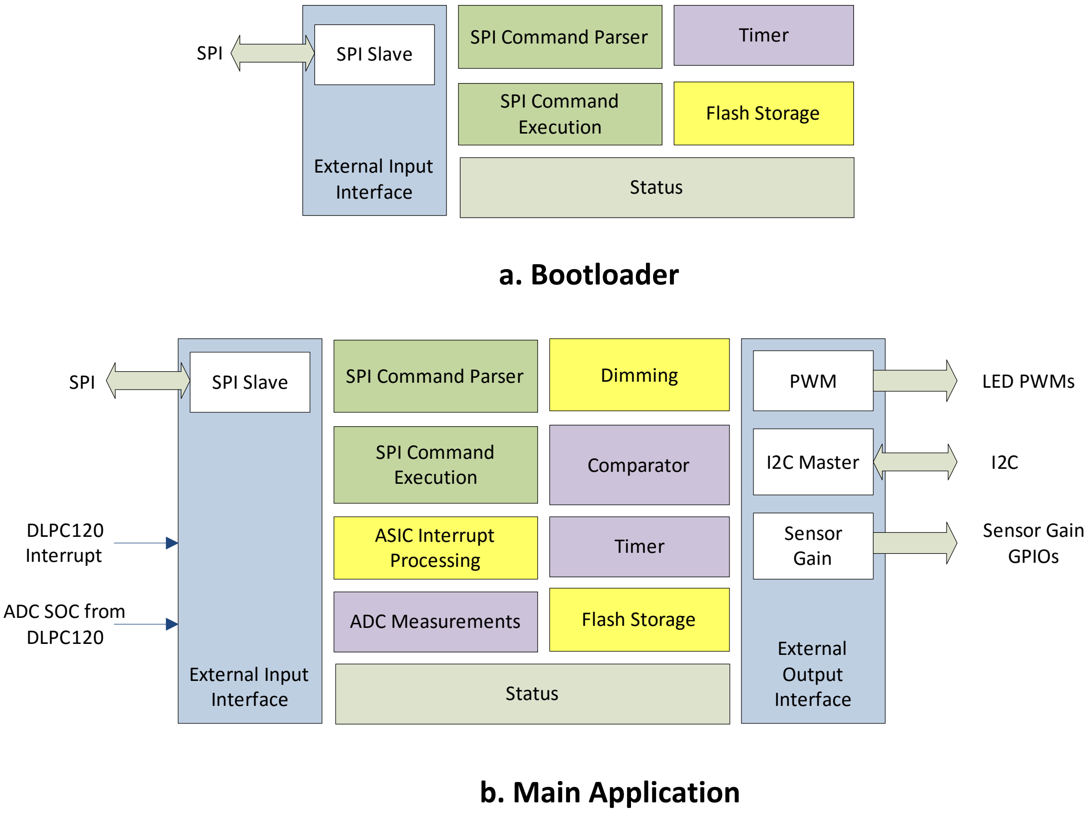


Figure 1-1. Piccolo Software Blocks

The Piccolo main application can be thought of having several sub-divisions depending on the type of task each section of code does.

- Initialization

- Voltage Supervision
- SPI Slave
- I2C Master
- Dimming Algorithm
- ASIC Interrupt Processing
- LED PWM control
- Sensor Gain Selection
- Flash Calibration Storage
- ADC Measurements
- Timer Control
- Status Indication

Chapter 2

DLPU061–March 2018

Bootloader Software

2.1 Bootloader Software

Piccolo bootloader allows verifying and programming the application. The primary function of the bootloader is to update the main application over SPI. The bootloader supports commands to erase, program and verify the application.

At startup, the bootloader configures the interrupt controller, vector table, clocks, GPIO, flash, timer and SPI. Once initialization is complete, the bootloader can be used to reprogram the main application, or to jump to the main application.

2.2 Initialization

This section of code does the initialization of the Piccolo at power on.

The Peripheral Interrupt Expansion (PIE) registers and vector table are initialized to their default state. The peripheral clocks are enabled next for the blocks which are used by the bootloader. The code then goes on to define the system by setting up the GPIOs for the SPI module to appropriate states. Following this, each of the peripherals (Flash, Timer and SPI) is initialized.

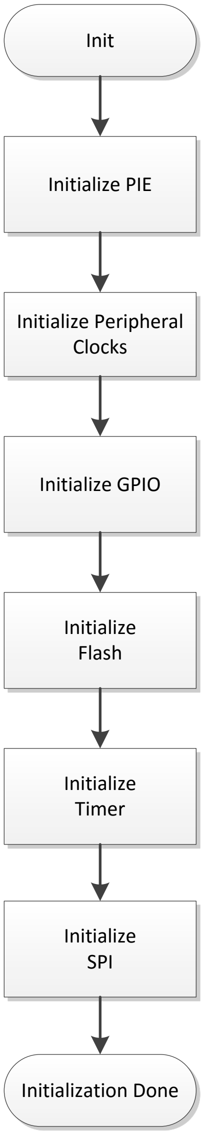


Figure 2-1. Initialization

The related C functions are named below:

- `void INIT_SetupHUDSystem(void)`
- `void DeviceInit(void)`
- `void PieCntrlInit(void)`
- `void PieVectTableInit(void)`
- `void INIT_SetupPeripheralClocks(void)`
- `void INIT_SetupGPIO(void)`
- `void INIT_Flash(void)`
- `void TMR_InitMeasurement(void)`
- `void TMR_StartMeasurement(void)`
- `void SPI_Init(void)`

2.3 Application Validation

After initialization, the bootloader performs validation of the main application. It does so by verifying the application signature and the checksum. If application signature is valid, it then performs a checksum validation. The checksum is calculated by the control program and programmed while programming the application code. If either validation step fails, then the bootloader sets “Stay in Bootloader” flag.

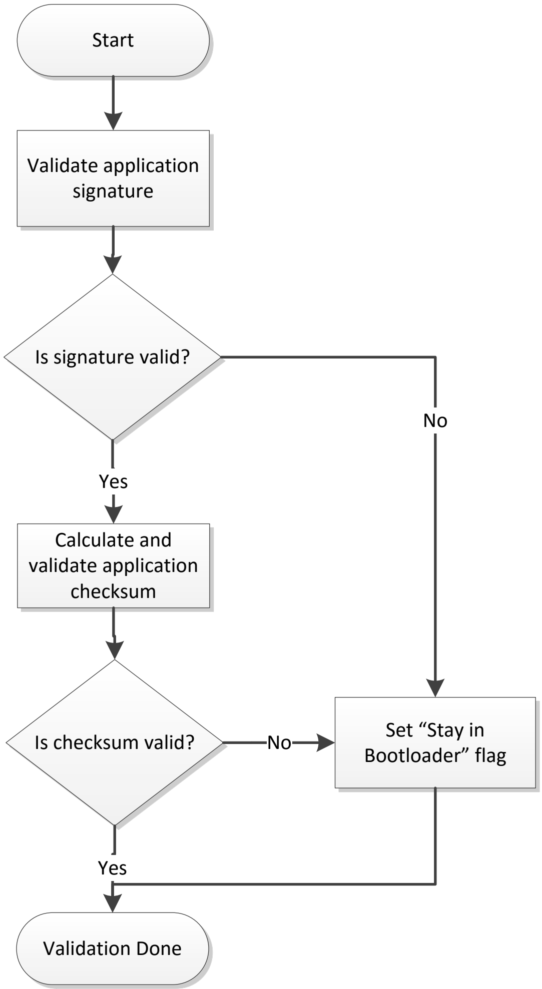

Start
Validate application signature
Is signature valid?
No
Yes
Calculate and validate application checksum
Is checksum valid?
No
Set “Stay in Bootloader” flag
Yes
Validation Done

Figure 2-2. Main Application Validation

2.4 Bootloader Main Loop

After validating the main application, the control enters the bootloader main loop where it waits for SPI commands. The control remains in the loop if “Stay in Bootloader” flag is set or if minimum time to stay in bootloader has not been reached.

For the normal case, when “Stay in Bootloader” flag is not set, the bootloader exits the loop after the minimum time to stay in bootloader is over. After exiting the loop, the program jumps to the main application start location.

“Stay in Bootloader” flag can be set because of the following reasons:

1. Invalid application: Refer to Section 2.3.
2. Stay in Bootloader command: Refer to Section 2.5.2.

“Stay in Bootloader” command is typically sent to reprogram the main application. This brings bootloader in a state where it will wait to receive commands to update the application (erase, program and verify). After the application is successfully updated, then send the “Toggle Program Mode” command to jump to application. This causes the control to jump to the beginning of the main application. Refer to Figure 2-3: Bootloader Flowchart for an overview of the bootloader execution flow.

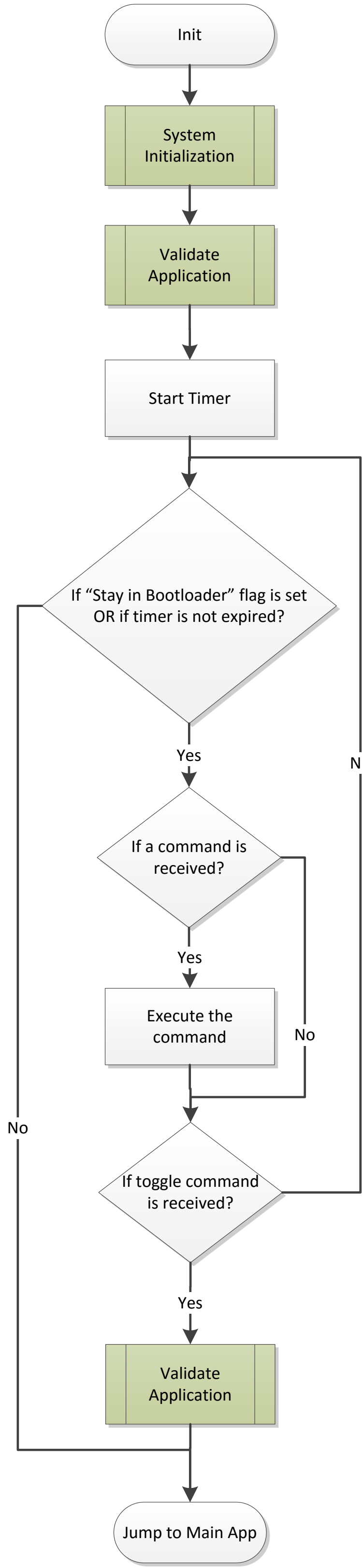


Figure 2-3. Bootloader Flowchart

2.5 Program Modes

There are two piccolo program modes: bootloader and main application. Software provides the mechanism to toggle between the two modes. Below is the method to toggle program modes:

2.5.1 Bootloader to Main Application

To jump from bootloader to main application, send toggle mode command to the bootloader. After receiving this command, the bootloader sends back the acknowledgment. It then verifies the main application, and if it is valid, the bootloader jumps to the start of the main application.

2.5.2 Main Application to Bootloader

To jump from main application to bootloader, send toggle mode command to the main application. After receiving this command, the main application sends back the acknowledgment. It then jumps to the start of the bootloader. If the acknowledgment is not received, that means, either the main application is corrupt or it doesn’t support “Toggle Mode”. In this case, reset the Piccolo to get it to the start of bootloader.

In the bootloader, after initialization and validating the application, if “Stay in Bootloader” pattern is not received, the bootloader jumps back to the main application. To stop the bootloader from jumping to the main application, after Toggle Mode command, start sending “Stay in Bootloader” pattern.

“Stay in Bootloader” pattern detection steps:

Host: Send the 4-byte write pattern repeatedly till the Piccolo sends back a 4-byte read pattern as a response. The read pattern indicates that the bootloader has detected the “Stay in Bootloader” write pattern.

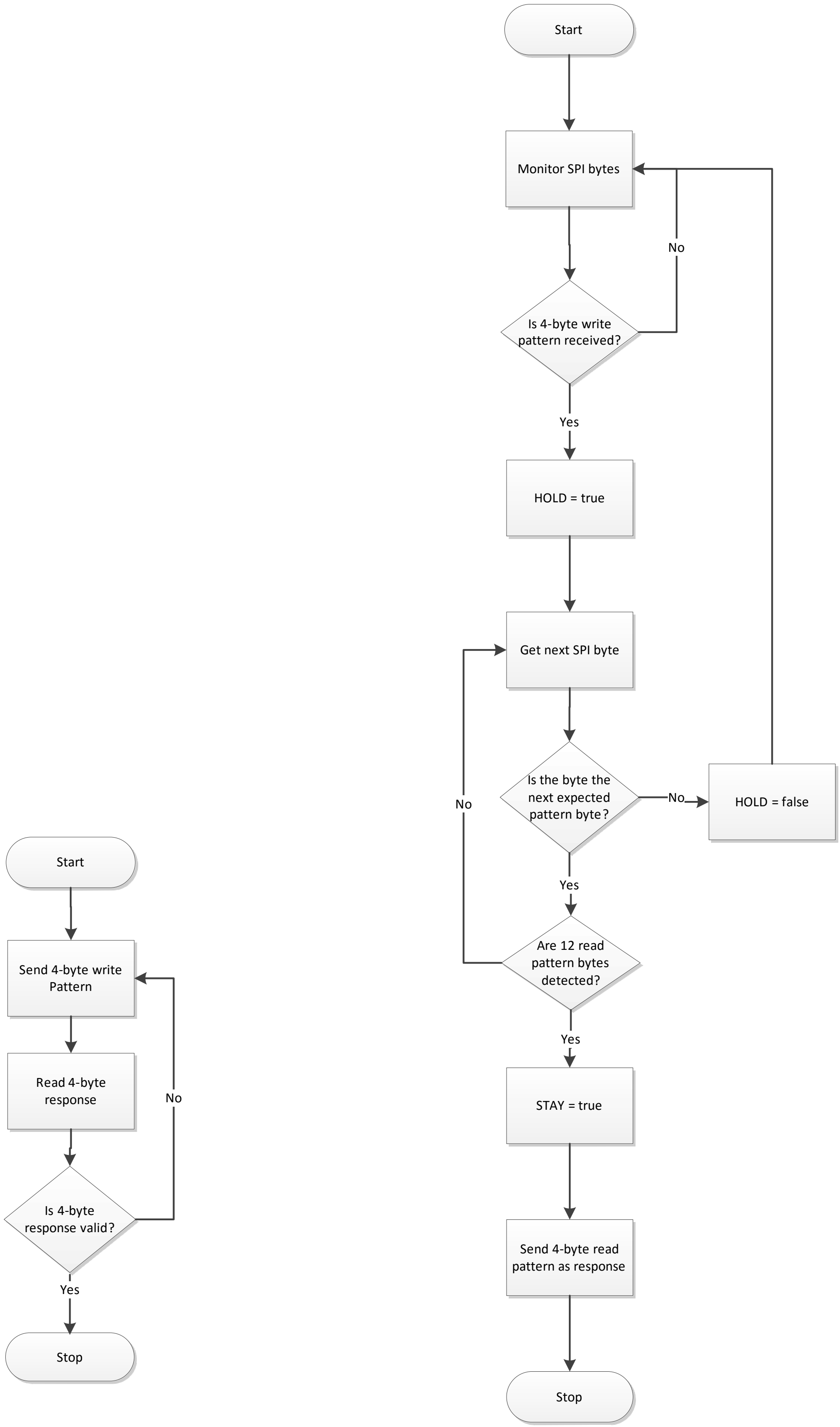
Piccolo: Piccolo remains in Bootloader main loop only for default wait time (10ms), provided the application validation was successful. In this duration, if the bootloader detects 4-byte write pattern, it comes to a “HOLD” state, in which it confirms the “Stay in Bootloader” write pattern by monitoring the following bytes. If the following bytes don’t match the write pattern, it jumps to the main application, indicating it was a false detection. If the following bytes match with the write pattern, it sets the “Stay in Bootloader” flag (STAY = true), and sends back the read pattern to acknowledge to the host that “Stay in Bootloader” write pattern was detected.

Now, the bootloader will not jump to the main application, and will wait for SPI commands.

The control remains in main loop if any of the below is true:

- Time elapsed is less than default wait time (10ms), or
- HOLD is set to true, or
- STAY is set to true.

Refer to Figure 2-4: Stay in Bootloader pattern detection for an overview of the pattern detection flow.


Start
Monitor SPI bytes
Is 4-byte write pattern received?
No
Yes
HOLD = true
Get next SPI byte
Is the byte the next expected pattern byte?
No
HOLD = false
Yes
No
Are 12 read pattern bytes detected?
Yes
STAY = true
Send 4-byte read pattern as response
Stop
Start
Send 4-byte write Pattern
Read 4-byte response
No
Is 4-byte response valid?
Yes
Stop

a. Host side **b. Piccolo side**

Figure 2-4. Stay in Bootloader Pattern Detection

2.6 Programming Main Application

To program the main application, a valid bootloader should be programmed in the piccolo flash, and piccolo should be in bootloader mode. The current state of piccolo can be either bootloader or main application. The following steps are used set the application into bootloader mode:

1. Bootloader – If this is the current state, nothing needs to be done.
2. Main application – Toggle mode to Bootloader. For more details, refer to the Section 2.5.2.

The bootloader is now ready to receive commands for reprogramming the application. To reprogram the application, send commands to perform following actions:

1. Erase the sectors to be programmed
2. Program the flash with main application data
3. Validate the application

Refer to the **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)** for command details.

After programming the application, “Toggle Mode” command can be sent to jump to the main application.

Chapter 3

DLPU061–March 2018

Software Initialization

3.1 Software Initialization

This section of code does the initialization of the Piccolo when it is powered on. The code copies the ADC and Oscillator calibration values from TI reserved one-time programmable (OTP) to appropriate trim registers. It then switches to internal oscillator and turns off all other clock sources to conserve power.

Following this, the Peripheral Interrupt Expansion (PIE) registers and vector table are initialized to their default state. The peripheral clocks are enabled next if the corresponding block is being used in the reference system.

The code then goes on to define the system by setting up the GPIOs to appropriate states. They are assigned either to a module (e.g. SPI, I2C etc.) or configured as a general purpose input or Output. The default values are also set for lines configured as general purpose output.

Following this, each of the peripherals (SPI, ADC, I2C, PWM, Comparator and Timer) is initialized one by one. The calibration data is then initialized as per the contents of sector D of the flash.

The Piccolo then performs the DLPC120 Initialization process as described in Section 3.2. Please note that this step is skipped if Voltage Supervision is enabled and the voltages are out of tolerance. In this case the DLPC120 initialization is done once the voltages are back within tolerance limits. Following DLPC120 initialization, the system initializes dimming and temperature compensation related data and applies the startup settings. The startup settings include applying the default backlight, dimming LUT index and executing the command lists mentioned in the configuration file. In addition to this, if there is no default display mode command list (Splash/Test Pattern/External Video) in the configuration file, the software assumes the system is in external video mode and executes external video detect BIST (see Section 15.4.1). After this the system goes into the MAIN loop where it waits for interrupts from ASIC, SPI, etc. See Chapter 4 for details on the MAIN loop.

The related C functions are named below:

- `void INIT_SetupHUDSystem(void)`
- `void DeviceInit(void)`
- `void INIT_SetupPeripheralClocks(void)`
- `void INIT_SetupGPIO(void)`
- `void SPI_Init(void)`
- `void I2C_Init(void)`
- `void COMP_Init(void)`
- `void PWM_Init(void)`
- `void ADC_Init(void)`
- `void DIM_Init(void)`
- `Uint16 CAL_Init(void)`
- `void DLPC120_Init(void)`
- `void TMPR_Init(void)`
- `void VMON_Init(void)`

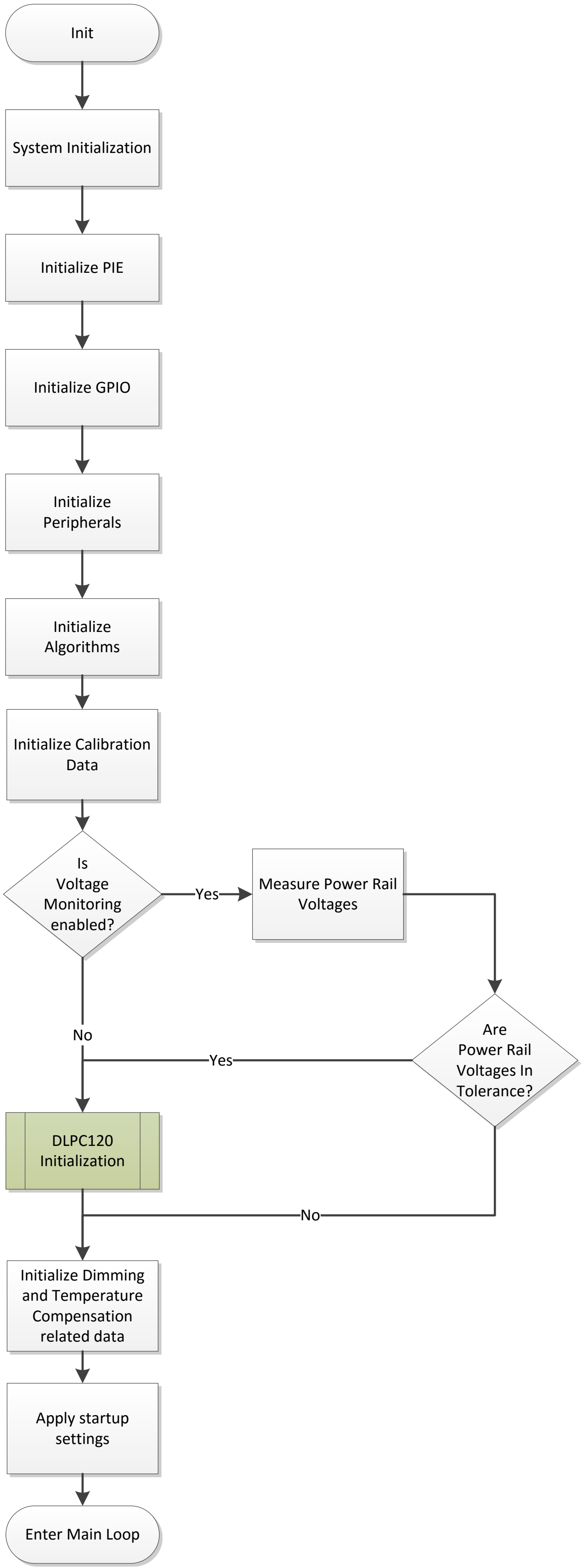

Init
System Initialization
Initialize PIE
Initialize GPIO
Initialize Peripherals
Initialize Algorithms
Initialize Calibration Data
Is Voltage Monitoring enabled?
Yes
Measure Power Rail Voltages
No
Yes
Are Power Rail Voltages In Tolerance?
DLPC120 Initialization
No
Initialize Dimming and Temperature Compensation related data
Apply startup settings
Enter Main Loop

Figure 3-1. Initialization Flow

3.2 DLPC120 ASIC Initialization

For proper dimming operation it is necessary that the DLPC120 ASIC is ready and running correctly. ASIC initialization comprises of 4 steps:

1. Checking if ASIC is used
2. PLL Spreading Enable
3. DDR initialization
4. ASIC BISTs

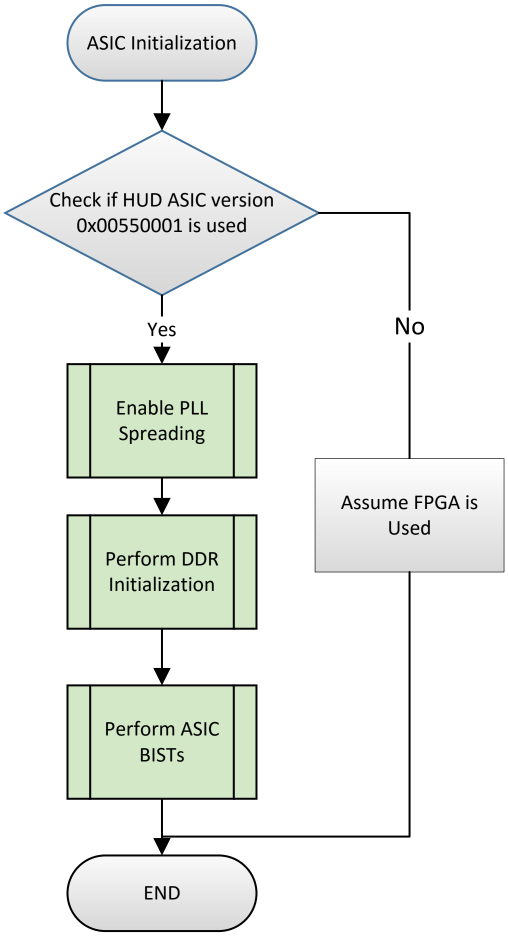


Figure 3-2. ASIC Initialization

3.2.1 *Checking if DLPC120 ASIC version 0x00550001 is used*

Before performing ASIC initialization, the Piccolo first checks if DLPC120 ASIC version 0x00550001 is being used. This is done by reading the HUD Version Info register and checking if it contains this value. If not, it is assumed that an FPGA is used and the next three initialization steps are not performed.

3.2.2 *PLL Spreading:*

Step 1: Enable PLL Spreading by performing appropriate register writes.

Step 2: Reset PLL. This will automatically kickoff DEFCONFIG again but will set up registers with PLL spreading enabled.

Step 3: Wait until DEFCONFIG finishes execution before moving on to initialize DDR.

3.2.3 DDR Initialization:

The DDR initialization is done by execution of a series of command lists. The Piccolo software uses an object of type DDRCmdListInfo that stores a list of setup data and expected status, which indicate the progress of the initialization. The expected status varies depending on whether the ASIC uses On Die Signal Termination (ODT) or External Termination. This can be known by writing specific setup data into the DPP_UMC_SIU_PUB_RDATA register after the execution of the first command list, and then reading bits 30 and 31 of the DPP_UMC_SIU_PUB_RDATA register. A warning is flagged if On Die Termination is used. Once we know the type of signal termination we can determine the expected status after each command list. The software executes the command lists one by one, writes the setup data to the appropriate registers, and waits for the corresponding expected status. In case the expected status is not received within a preconfigured time, the entire initialization process times out and an ASIC_INIT_FAILURE status is flagged. Once the initialization is complete the ASIC is ready to perform functions like dimming etc and the Piccolo enters the main loop.

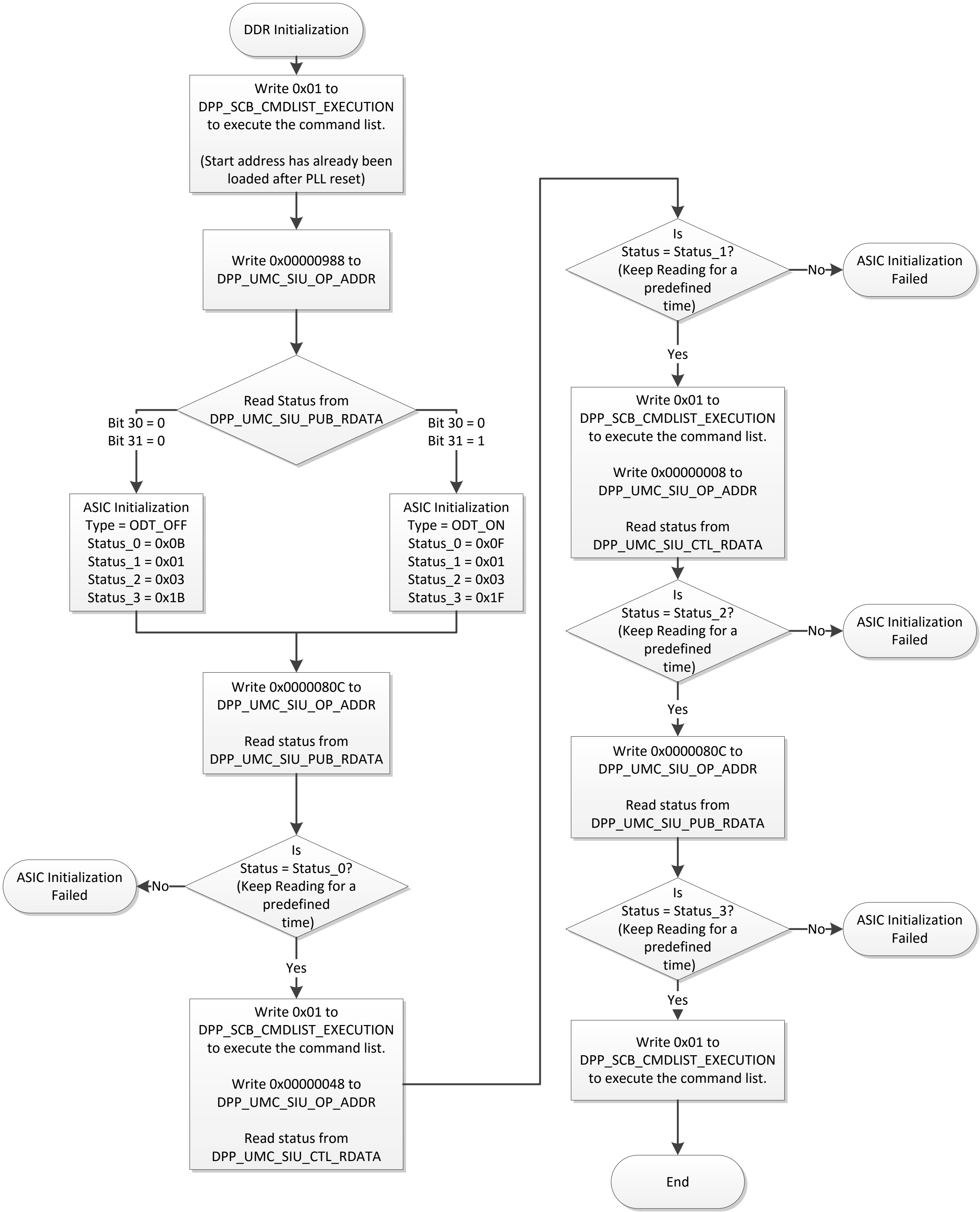

DDR Initialization
Write 0x01 to DPP_SCB_CMDLIST_EXECUTION to execute the command list.
(Start address has already been loaded after PLL reset)
Write 0x00000988 to DPP_UMC_SIU_OP_ADDR
Read Status from DPP_UMC_SIU_PUB_RDATA
Bit 30 = 0
Bit 31 = 0
Bit 30 = 0
Bit 31 = 1
ASIC Initialization Type = ODT_OFF
Status_0 = 0x0B
Status_1 = 0x01
Status_2 = 0x03
Status_3 = 0x1B
ASIC Initialization Type = ODT_ON
Status_0 = 0x0F
Status_1 = 0x01
Status_2 = 0x03
Status_3 = 0x1F
Write 0x0000080C to DPP_UMC_SIU_OP_ADDR
Read status from DPP_UMC_SIU_PUB_RDATA
Is Status = Status_0? (Keep Reading for a predefined time)
No
ASIC Initialization Failed
Yes
Write 0x01 to DPP_SCB_CMDLIST_EXECUTION to execute the command list.
Write 0x00000048 to DPP_UMC_SIU_OP_ADDR
Read status from DPP_UMC_SIU_CTL_RDATA
Is Status = Status_1? (Keep Reading for a predefined time)
No
ASIC Initialization Failed
Yes
Write 0x01 to DPP_SCB_CMDLIST_EXECUTION to execute the command list.
Write 0x00000008 to DPP_UMC_SIU_OP_ADDR
Read status from DPP_UMC_SIU_CTL_RDATA
Is Status = Status_2? (Keep Reading for a predefined time)
No
ASIC Initialization Failed
Yes
Write 0x0000080C to DPP_UMC_SIU_OP_ADDR
Read status from DPP_UMC_SIU_PUB_RDATA
Is Status = Status_3? (Keep Reading for a predefined time)
No
ASIC Initialization Failed
Yes
Write 0x01 to DPP_SCB_CMDLIST_EXECUTION to execute the command list.
End

Figure 3-3. DDR Initialization

3.2.4 ASIC Built in Self Tests (BIST's):

Once the DDR initialization is complete the Piccolo performs 4 Built In Self Tests (BIST's) on the ASIC. Another BIST – the Front End Video BIST (see Section 15.3) is not performed during initialization. It is triggered by SPI commands. The External Video Detect BIST (see Section 15.4) is executed while applying startup settings or when an external video command list is executed. It can also be externally setup and triggered using SPI commands. Some of these BIST's return a checksum or device ID which is stored by the Piccolo and can be viewed through SPI commands. The 4 BISTs that are executed on startup are:

1. DDR2 BIST: This BIST does not return any checksum.
2. Flash BIST: This BIST returns a checksum of a particular section of the memory as specified in the arguments to the function call. The checksum can be viewed along with the BIST results through SPI commands.
3. DMD JTAG BIST: This BIST returns a DMD Device ID, which can be viewed along with the BIST results through SPI commands.
4. System BIST: This BIST requires the execution of 2 command lists with a predefined type "Internal" and also predefined names. The first command list should have the name "SetupSystemBIST". This command list sets up the system BIST and should be executed before performing the test. The second command list, which reverts back from the setup operations performed by the first command list, should be called "EndSystemBIST". These predefined naming conventions are followed in the flash build files, provided by a TI Apps Engineer. As the name suggests this should be called after the BIST is complete. Please note that since this BIST requires the execution of some command lists, it is critical that it is performed only after calling the `CAL_Init ()` function, which stores the command list details in the appropriate data structures. This BIST also returns a checksum, which can be viewed along with the BIST results through SPI commands.

NOTE: The BISTs are performed only when the correct ASIC version is detected by the first step of DDR initialization.

For more detailed information on how the BIST's are performed please refer to the **I2C Programmer's Guide for DLPC120 ASIC (DLPA055)**.

Chapter 4

DLPU061–March 2018

MAIN Loop

4.1 MAIN Loop

The Piccolo software enters into idle state immediately after initialization. This idle state is implemented as an infinite while loop where the Piccolo software stays awake waiting for interrupts from DLPC120 or an SPI Command from Master Processor. Tasks that need to be run periodically are also performed in this state, including checking for power rail voltage quality, handling non-blocking delays, functions that check temperature changes in the DMD and compute the scale factor optimization (SFO) value for Piccolo High Resolution PWM. The control flow is shown in Figure 4-1.

The related C functions are:

- `int16 main(void)`
- `BOOL DLPC120_GetInterruptReceived(void)`
- `Uint16 DLPC120_ProcessInterrupt(void)`
- `BOOL SPI_GetCmdPending(void)`
- `void SPI_ExecuteCmd(void)`
- `Uint16 TMPR_CheckUpdate()`
- `Uint16 PWM_SFO(void)`
- `void VMON_ProcessPowerRailStatus(void)`
- `void TMR_HandleNonBlockingDelays(void)`

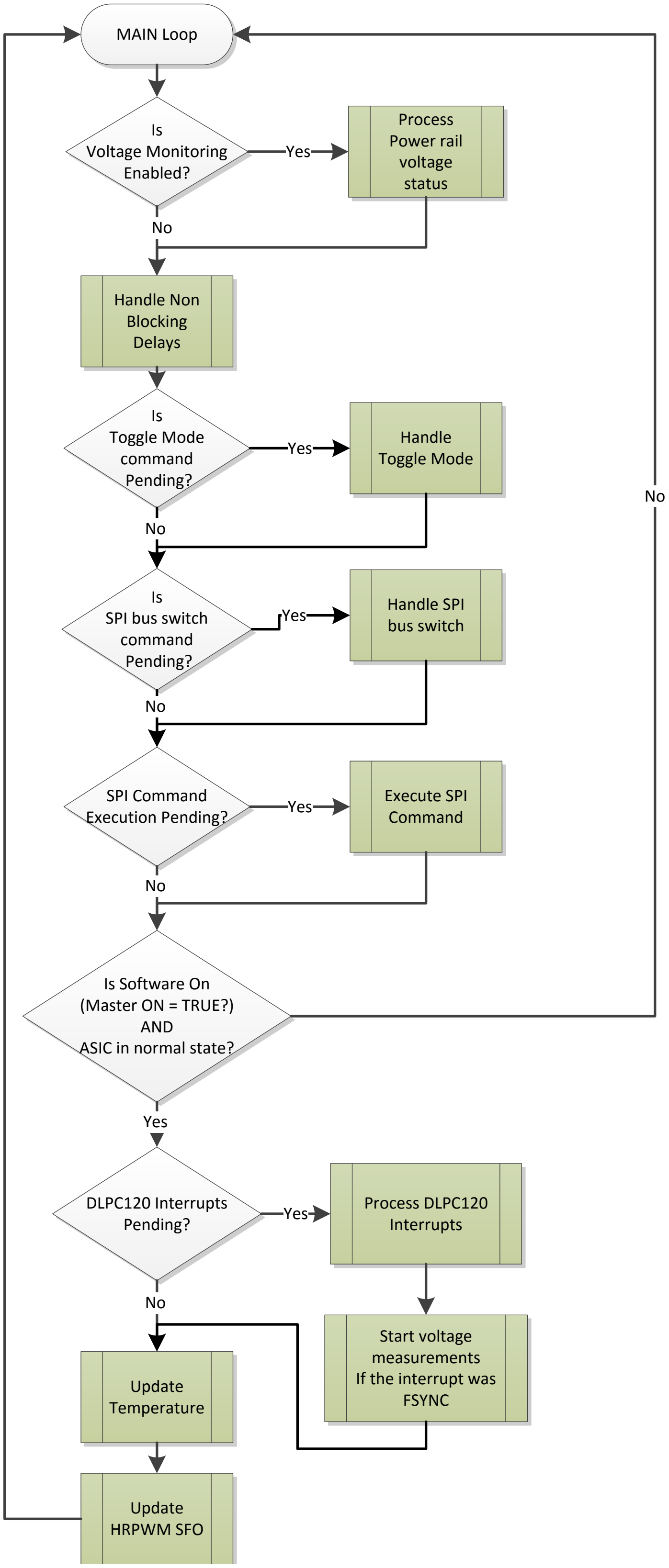

MAIN Loop
Is Voltage Monitoring Enabled?
Yes
Process Power rail voltage status
No
Handle Non Blocking Delays
Is Toggle Mode command Pending?
Yes
Handle Toggle Mode
No
Is SPI bus switch command Pending?
Yes
Handle SPI bus switch
No
SPI Command Execution Pending?
Yes
Execute SPI Command
No
Is Software On (Master ON = TRUE?) AND ASIC in normal state?
No
Yes
DLPC120 Interrupts Pending?
Yes
Process DLPC120 Interrupts
No
Start voltage measurements If the interrupt was FSYNC
Update Temperature
Update HRPWM SFO

Figure 4-1. MAIN Loop

Chapter 5

DLPU061–March 2018

ASIC Interrupt Handling

5.1 ASIC Interrupt Handling

The Piccolo communicates to the DLPC120 ASIC over I2C where Piccolo is the I2C master. In addition to the I2C, a GPIO line is connected between ASIC and Piccolo for the use of ASIC to interrupt Piccolo to indicate various conditions.

When an ASIC interrupt is received, the software will set a flag which will be processed by the MAIN loop. During the interrupt processing stage, the software issues I2C command to read out the ASIC interrupt register to identify which bit caused the interrupt. The interrupt bit is then cleared by writing to ASIC register. Please refer to **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)** for details on ASIC registers.

Piccolo software manages two interrupts from the ASIC

- FSYNC interrupt and
- Splash Load Done interrupt.

The FSYNC interrupt is issued by the ASIC to indicate the frame boundaries to the Piccolo. This interrupt is used by Piccolo to synchronize the sequence and PWM updates to achieve proper dimming. The Dimming Control Algorithm is explained in detail in Chapter 8.

The Splash Load Done Interrupt is issued by the ASIC when a splash load operation completes. Piccolo uses this interrupt to do special dimming processing that is needed when a splash screen is set up. Chapter 5 explains how this interrupt is used in detail.

The control flow is shown in Figure 5-1.

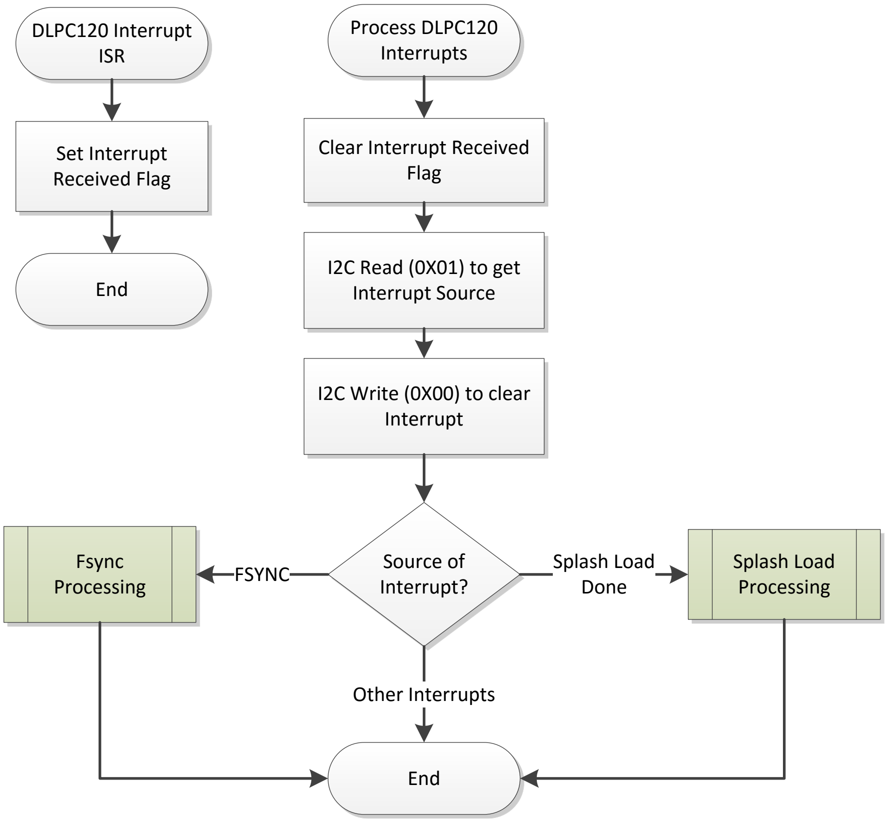

DLPC120 Interrupt ISR
Set Interrupt Received Flag
End
Process DLPC120 Interrupts
Clear Interrupt Received Flag
I2C Read (0X01) to get Interrupt Source
I2C Write (0X00) to clear Interrupt
Source of Interrupt?
FSYNC
Fsync Processing
Splash Load Done
Splash Load Processing
Other Interrupts
End

Figure 5-1. ASIC Interrupt Handling

The related C functions are:

- `Uint16 DLPC120_EnableInterrupt(Uint16 Bit, BOOL Enable)`
- `interrupt void DLPC120_GeneralInterrupt(void)`
- `Uint16 DLPC120_ClearInterruptRegister(void)`
- `BOOL DLPC120_GetInterruptReceived(void)`
- `Uint16 DLPC120_ProcessInterrupt(void)`
- `Uint16 DIM_ProcessASICInterrupt(BOOL Fsync, BOOL SSGLoadDone)`

Chapter 6

DLPU061–March 2018

SPI Slave

6.1 SPI Slave

The Piccolo color controller IC hosts an SPI slave interface that serves as the communication port to the display controller. The display controller generally will be an MCU with graphics capability that serves as the display systems master. The Piccolo accepts a variety of commands through this interface. These are described in a **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)** document available separately.

The Piccolo software utilizes two of the SPI interrupts – Receive Interrupt, Receiver Overrun Interrupt. The two interrupts share the same ISR. The receive interrupt will indicate that a character has been placed in the SPI receiver buffer and is ready to be read. The Receiver overrun interrupt on the other hand indicates a condition where a new character is received before the previous character was read from the receiver buffer. If a receiver overrun interrupt is received, the SW resets the SPI to recover from the error condition. Please note that during recovery, any SPI commands received will not be serviced. The control flow happening in ISR is shown in Figure 6-1.

The related C functions are:

- `void SPI_Init(void)`
- `interrupt void SPI_ReceiveISR(void)`
- `void SPI_Reset(void)`

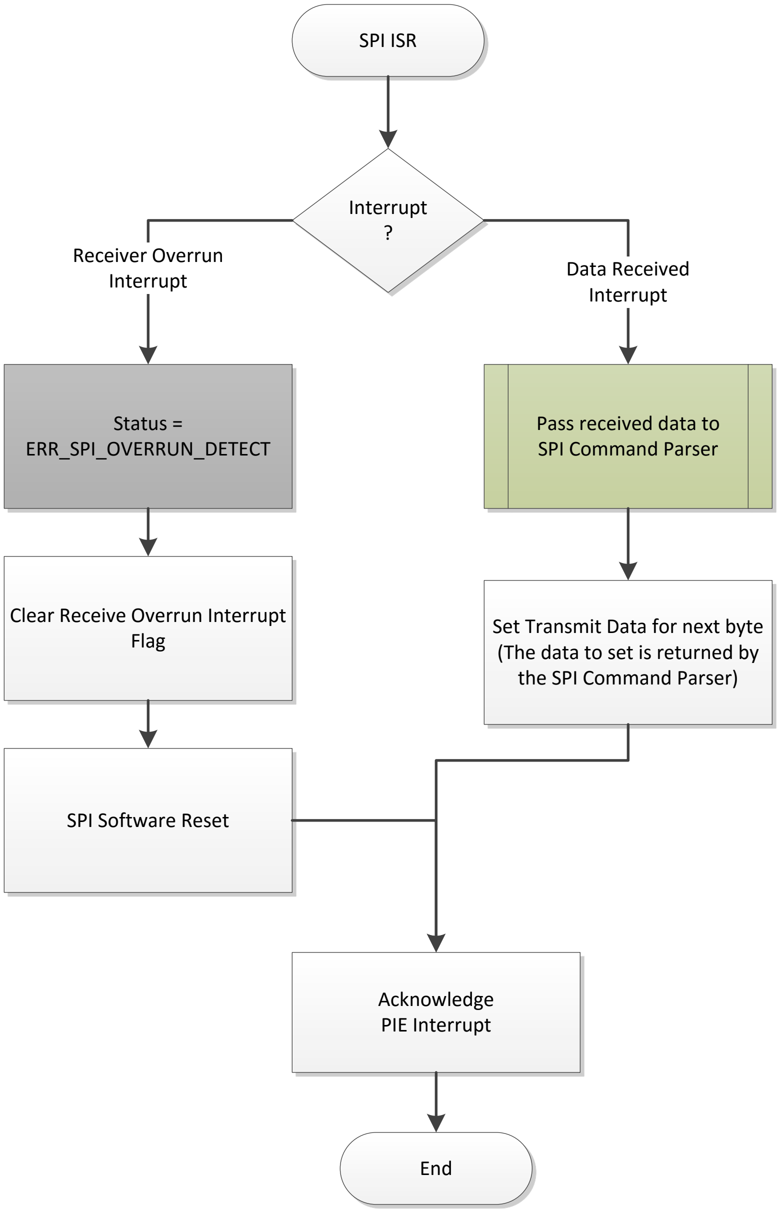

SPI ISR
Interrupt ?
Receiver Overrun Interrupt
Data Received Interrupt
Status = ERR_SPI_OVERRUN_DETECT
Pass received data to SPI Command Parser
Clear Receive Overrun Interrupt Flag
Set Transmit Data for next byte (The data to set is returned by the SPI Command Parser)
SPI Software Reset
Acknowledge PIE Interrupt
End

Figure 6-1. SPI ISR

6.2 SPI Command Parser

The SPI Command Parser analyses the received bytes and verifies that they confirm to the SPI Protocol used in the HUD reference system. The SPI commands should be sent as command packets. The packet structure is shown below:

Byte 1	Byte 2	Byte 3	…	Byte n+2	Byte n+3
Command Byte	Length(n)	Data[0]	…	Data[n-1]	Checksum

Figure 6-2. SPI Command Packet

Every command packet should be preceded by the byte 0xA5 (known as START_CHARACTER). Another special character 0x5A (known as ESCAPE_CHARACTER) should be utilized to escape any data byte 0xA5 coming in the data stream. Once the command packet is received completely, the software executes the command. It then prepares a response byte to indicate the success/failure of the command. Master should keep sending dummy bytes till the software prepares the response byte. Detailed explanations of the SPI command Packet is available in **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)**.

The high level control flow of the SPI command parser is shown below.

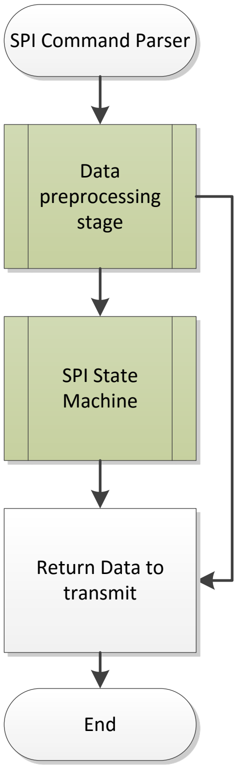


Figure 6-3. SPI Command Parser

6.3 SPI Preprocessing Stage

Two characters, START_CHARACTER and ESCAPE_CHARACTER, are reserved by the SPI protocol and have special meaning associated with them. A START_CHARACTER can only appear at the beginning of a command transmission and serves the purpose of indicating the start of a command packet to the Piccolo SW. The ESCAPE_CHARACTER is used to escape the START_CHARACTER. More details on these special characters can be found in the **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)**.

The SPI preprocessing stage removes these special characters from the stream. The preprocessing stage is shown in Figure 6-4. If a START_CHARACTER is received, the SPI state machine (Section 6.4) is reset to *Command Byte* state. Status bit is flagged if the previous command was not complete. On receiving an ESCAPE_CHARACTER, a flag is set to indicate that escape sequence is active. In the two cases above, the preprocessing stage ends the SPI command parser without going to the SPI state machine.

When characters other than START_CHARACTER and ESCAPE_CHARACTER are received, the preprocessing stage checks if an escape sequence is active. If an escape sequence is active, and if the data received is 0x00 (used for escaping START_CHARACTER), the SPI state machine is called with 0xA5 as the data byte. In all other cases, the received data is passed to the SPI state machine without modifications.

The related C function is:

- `Uint16 SPI_PreprocessData (Uint16 Data)`

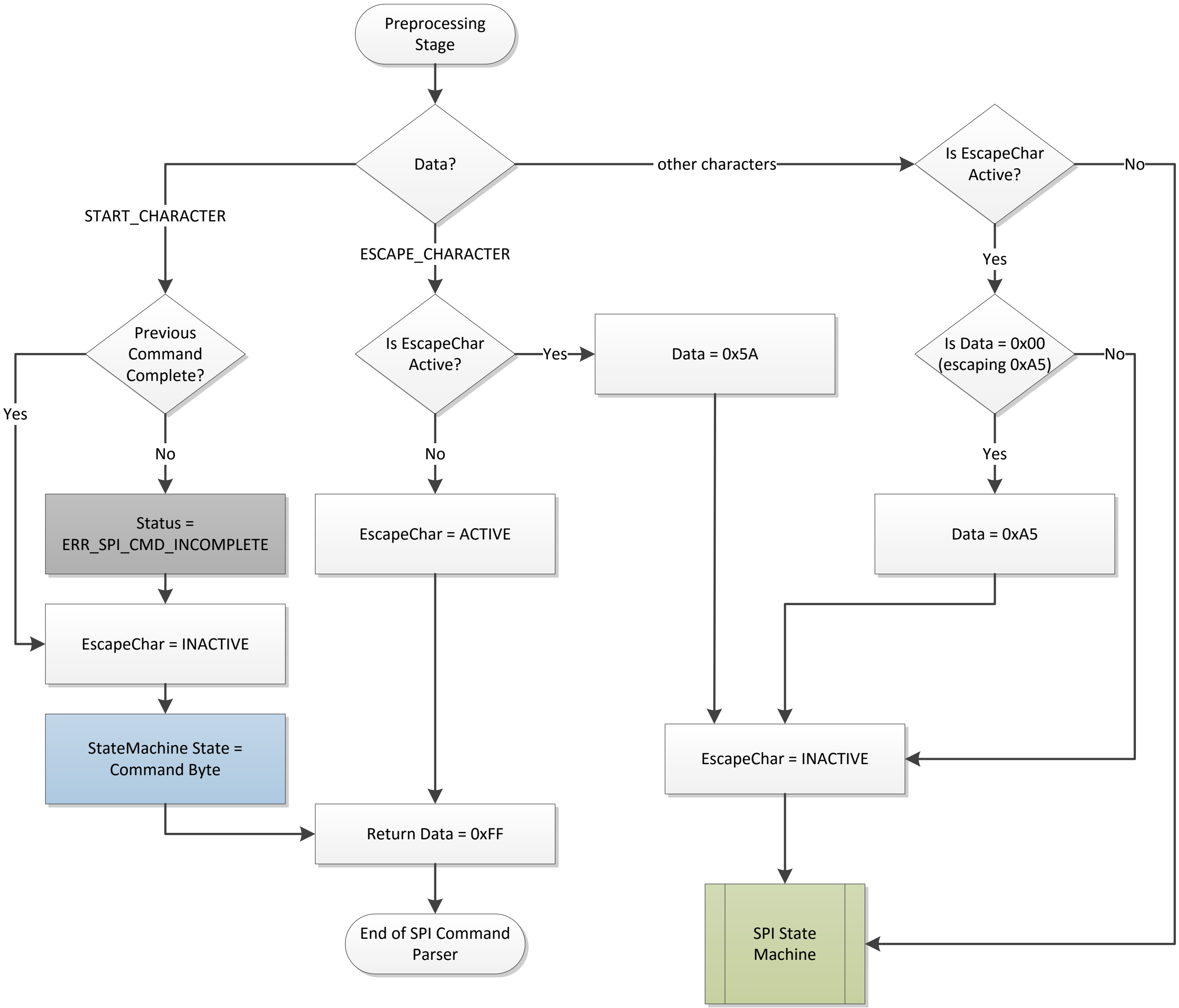


Figure 6-4. SPI Preprocessing Stage

6.4 SPI State Machine

After the preprocessing stage, the data byte received is passed on to the SPI state machine. The state machine is shown in Figure 6-5. SPI being a two-way protocol, both master and slave need to transmit data during every transaction. Piccolo being a slave thus needs to send a byte back to master transmitter at the same time it is receiving the byte from master. The values shown on the lines connecting the states indicates this data byte that will be transmitted back to the master in the next SPI transaction. 0xFF is used as a dummy byte (do not confuse with “Dummy” state) which is transmitted by Piccolo when there is no meaningful data to reply with. For example, between the “Command Byte” state and “Length Byte” state, the dummy byte 0xFF is transmitted from the Piccolo back to the master.

The user needs to send the START_CHARACTER to go to the first state which is the *Command Byte*. No checks are done at this point to verify if the command byte received is correct. The next state is *Length Byte*. If the length is zero, the next state is set to *Checksum Byte*. Otherwise, the state is in *Data Byte* state till length bytes of data are received. After enough data is received, the state is changed to *Checksum Byte*. At this stage, the checksum computed on the fly by the state machine is verified with the received checksum byte. If they do not match, an error is indicated by sending response byte 0x2 (CHECKSUM ERROR). The state then changes to *Ignore*, which will ignore all the bytes received hence forth till a START_CHARACTER is received again.

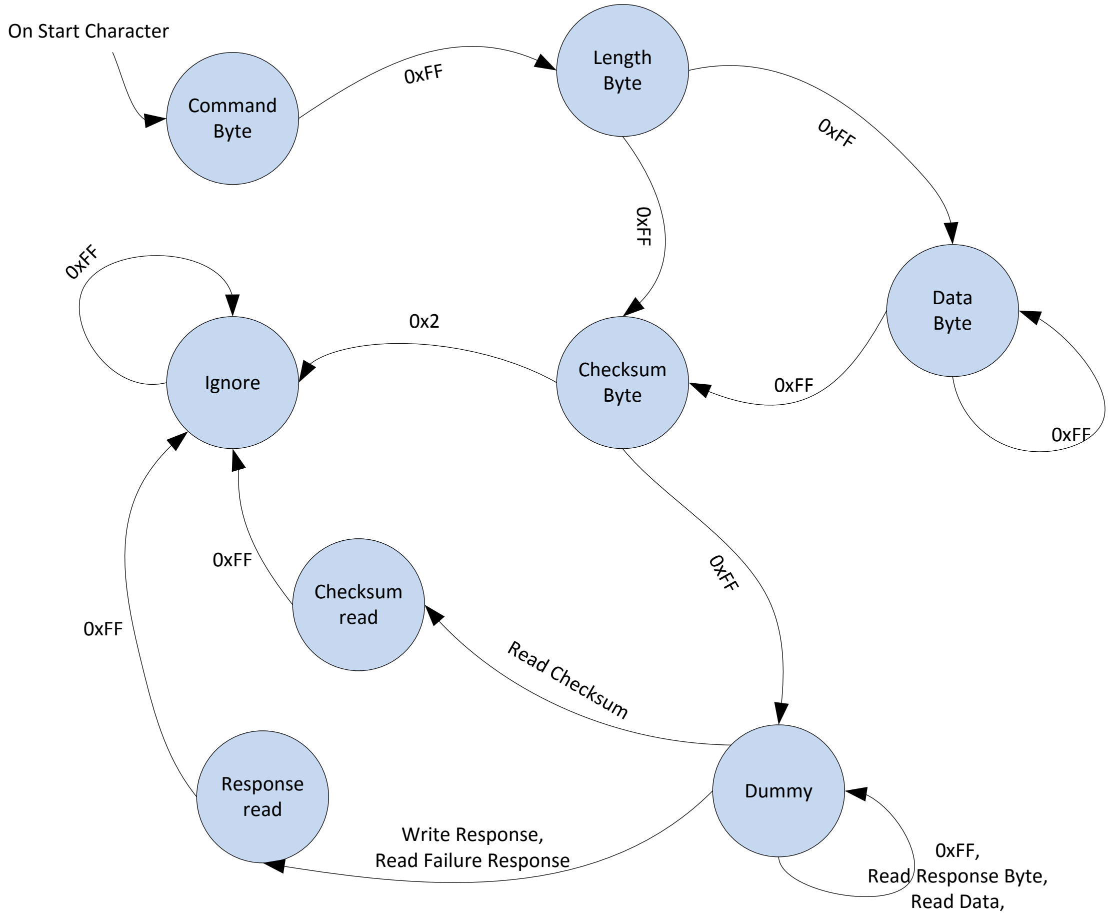


Figure 6-5. SPI State Machine

Once the command is received completely (State = *Checksum Byte*), the Piccolo state machine changes to *Dummy State*. Piccolo will remain in this state, sending only 0xFF to any request from the Master, till the command completes execution. This is analogous to slave holding the clock low in I2C to inform master it is busy. The master will send dummy bytes to Piccolo till it gets a non-0xFF response to the command it sent. A non-0xFF response indicates the slave completed execution, and is ready to accept another command. If the master needs to terminate a command execution mid-way, it can send another START_CHARACTER (0xA5), receiving which the SPI state machine is reset. This condition will be indicated in the system status. Please note, however, that, a command which was already in execution state may still execute in the background without the master getting the response.

For a **write command**, once the execution is complete and the execution response is available, the state is changed to *Response Read* where the master actually reads back the response byte. For a **read command**, the response byte, and the requested read data are all transmitted from inside the *Dummy State* itself. The last byte of the read data (Read Checksum) will be transmitted in the *Checksum Read* state.

The state then changes to *Ignore* till another START_CHARACTER is received.

An example from the SPI log will make this clearer:

A5FF 01FF 00FF 01FF 00FF 0001 0002 00FF 00FF 0001

Each word in the log has two bytes of data. The first byte is the byte transmitted by master, and the second one is the byte transmitted by slave. The above example is broken down below:

Bytes	Master Transmit	Slave Response	SPI - State Machine		Comments
			Current State	**New State**	
A5FF	Start Character	Dummy Byte	Ignore	Command Byte	
01FF	Command Byte	Dummy Byte	Command Byte	Length Byte	01 - stands for read backlight**
00FF	Length = 0	Dummy Byte	Length Byte	Checksum Byte	Switched to Checksum byte because Length = 0
01FF	Write Checksum	Dummy Byte	Checksum Byte	Dummy	
00FF	Master Dummy Byte (00)	Dummy Byte	Dummy	Dummy	started execution, but response not available
0001	Master Dummy Byte (00)	Response - 01 = SUCCESS	Dummy	Dummy	
0002	Master Dummy Byte (00)	Length = 2	Dummy	Dummy	
00FF	Master Dummy Byte (00)	Backlight Value (lsb)	Dummy	Dummy	
00FF	Master Dummy Byte (00)	Backlight Value (msb)	Dummy	Checksum Read	
00B8	Master Dummy Byte (00)	Read Checksum	Checksum Read	Ignore	

**Please check *SPI Command Interface* document for details on commands.

The related C function is:

- `static Uint16 SPI_ProcessData (Uint16 Data)`

6.5 Execution Stage

The command execution starts from MAIN loop. This stage begins with a series of validation checks. The command execution is not done if any of these validation tests fail. After a successful validation, the actual execution begins. The software will call the respective command handler functions based on the command byte (also known as CmdId). Once the execution is complete, the software prepares the response packet indicating success/failure. In case of a read command, the data requested by the master is transmitted back following the response packet. The execution stage is shown in Figure 6-6.

The related C functions are:

- `void SPI_ExecuteCmd(void)`
- `static Uint16 SPI_FindCommand(Uint16 Cmd, Uint16* Index)`
- `static void SPI_SetResponse (Uint16 Response)`
- `BOOL SPI_GetCmdPending(void)`

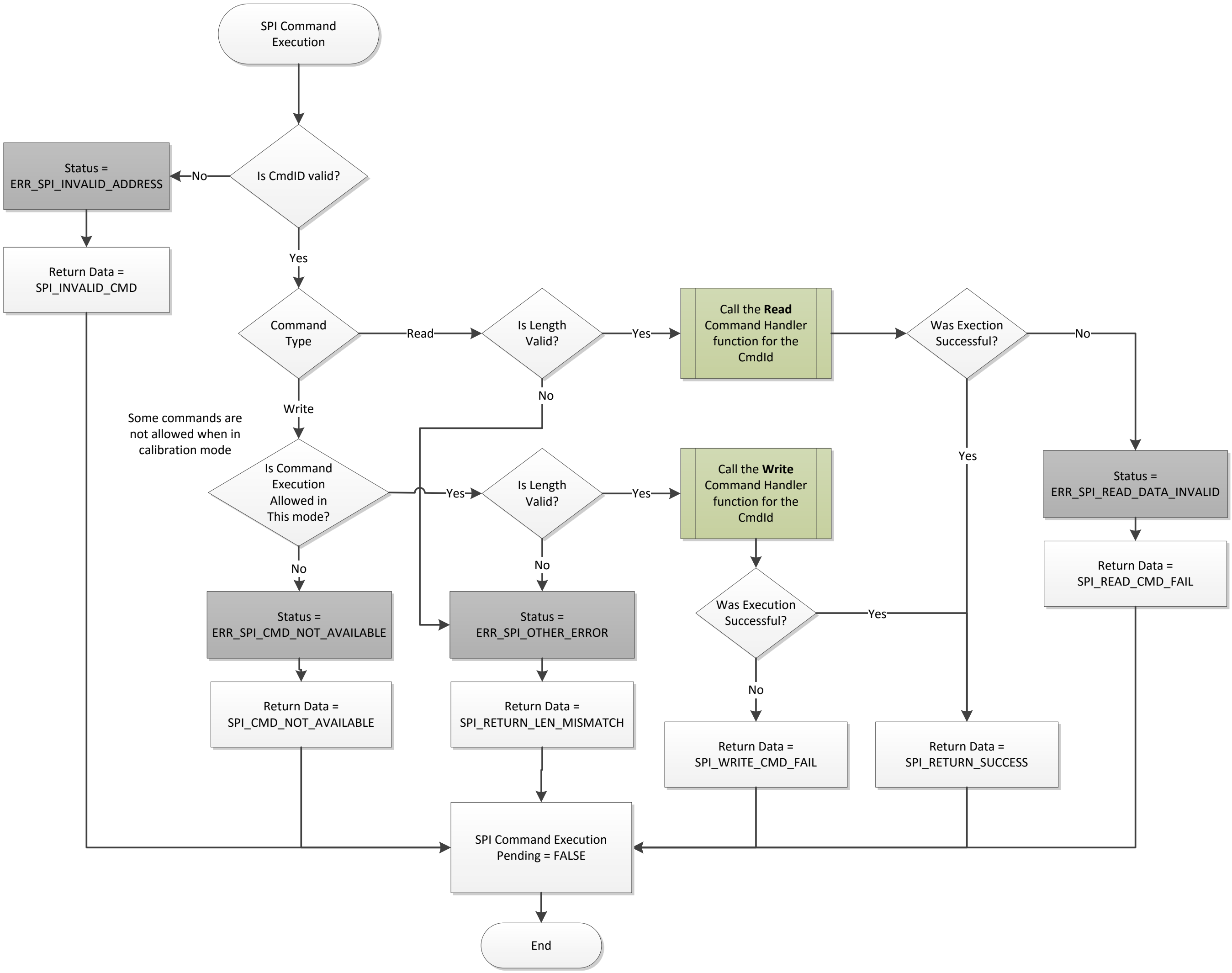

SPI Command Execution
Is CmdID valid?
No
Status = ERR_SPI_INVALID_ADDRESS
Return Data = SPI_INVALID_CMD
Yes
Command Type
Read
Is Length Valid?
Yes
Call the **Read** Command Handler function for the CmdId
Was Exection Successful?
No
Status = ERR_SPI_READ_DATA_INVALID
Return Data = SPI_READ_CMD_FAIL
Yes
Write
Some commands are not allowed when in calibration mode
Is Command Execution Allowed in This mode?
Yes
Is Length Valid?
Yes
Call the **Write** Command Handler function for the CmdId
No
Status = ERR_SPI_CMD_NOT_AVAILABLE
Return Data = SPI_CMD_NOT_AVAILABLE
No
Status = ERR_SPI_OTHER_ERROR
Return Data = SPI_RETURN_LEN_MISMATCH
Was Execution Successful?
Yes
No
Return Data = SPI_WRITE_CMD_FAIL
Return Data = SPI_RETURN_SUCCESS
SPI Command Execution Pending = FALSE
End

Figure 6-6. SPI Execution Phase

Command Lists

7.1 Command Lists

A Command List is a binary batch file that can be executed by the ASIC. Command Lists are stored in the ASIC flash. The ASIC can be configured to execute a command list by first loading the start address of that command list to the DPP_SCB_CL_BASE_ADDRESS register, and then writing 0x01 to the DPP_SCB_CMDLIST_EXECUTION register to kick-off execution. ASIC Command Lists are extensively used by the Piccolo SW for various functions such as DDR Initialization, System BISTs etc. Command Lists can be broadly divided into 4 categories:

- Splash
- Test Pattern
- External Video
- Generic/Other Command Lists

The details (including the start address in the ASIC Flash) of most command lists are stored in the Configuration file which enables the Piccolo SW to execute them without the user having to enter the address of each command list. The details can be viewed by opening the Configuration File in the Automotive Control Program. More details about the Configuration File can be found in Chapter 10.

Chapter 8

DLPU061–March 2018

Dimming Control and Temperature Compensation:

8.1 Dimming Control and Temperature Compensation:

The Piccolo color controller receives dimming input levels that range from 65535 to 0. The value is treated as an unsigned 0.16 fixed point number. The bit order and weighing for the backlight value relative to the full scale is shown below:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}

Figure 8-1. Bit order and weighing for the backlight value

8.2 Operating Modes:

The dimming control can operate in three modes based on how the brightness is achieved:

1. Continuous Mode:

   In *Continuous Mode* of operation, the on-board comparator in the Piccolo device is used to compare desired target LED light amplitude to actual amplitude. The *Continuous Mode* is used for high brightness situations.

2. Discontinuous Mode:

   For nominal and low light situations (up to a lower brightness limit), *Discontinuous Mode* is utilized. In this mode, during bit slice display periods, the LEDs are driven by individual pulses of current, rather than a continuous level of current.

3. VAC Mode:

To achieve very low dimming levels, we use *Video and Contrast Adjustment Control* (VAC). In the *VAC* mode, dimming is achieved by digitally attenuating the input video data. This can be used in conjunction with continuous mode or discontinuous mode. It is recommended for low light levels since it will decrease the bit depth of the displayed image.

More details on these modes are explained in **LED Driver for DLP3030-Q1 Displays Application Note (DLPA084)**.

VAC mode is explained in Section 8.3.

8.3 Video and Contrast Adjustment Control (VAC) Implementation:

VAC mode helps achieve lower brightness levels by using electronic contrast adjustment. The point below which VAC mode is turned on is called VAC Mode Switch Point. When VAC is enabled, continuous and discontinuous mode dimming are performed to achieve dimming up to the level of the VAC Mode Switch Point. Further dimming is achieved by contrast adjustment as illustrated in the example shown in Figure 8-2.

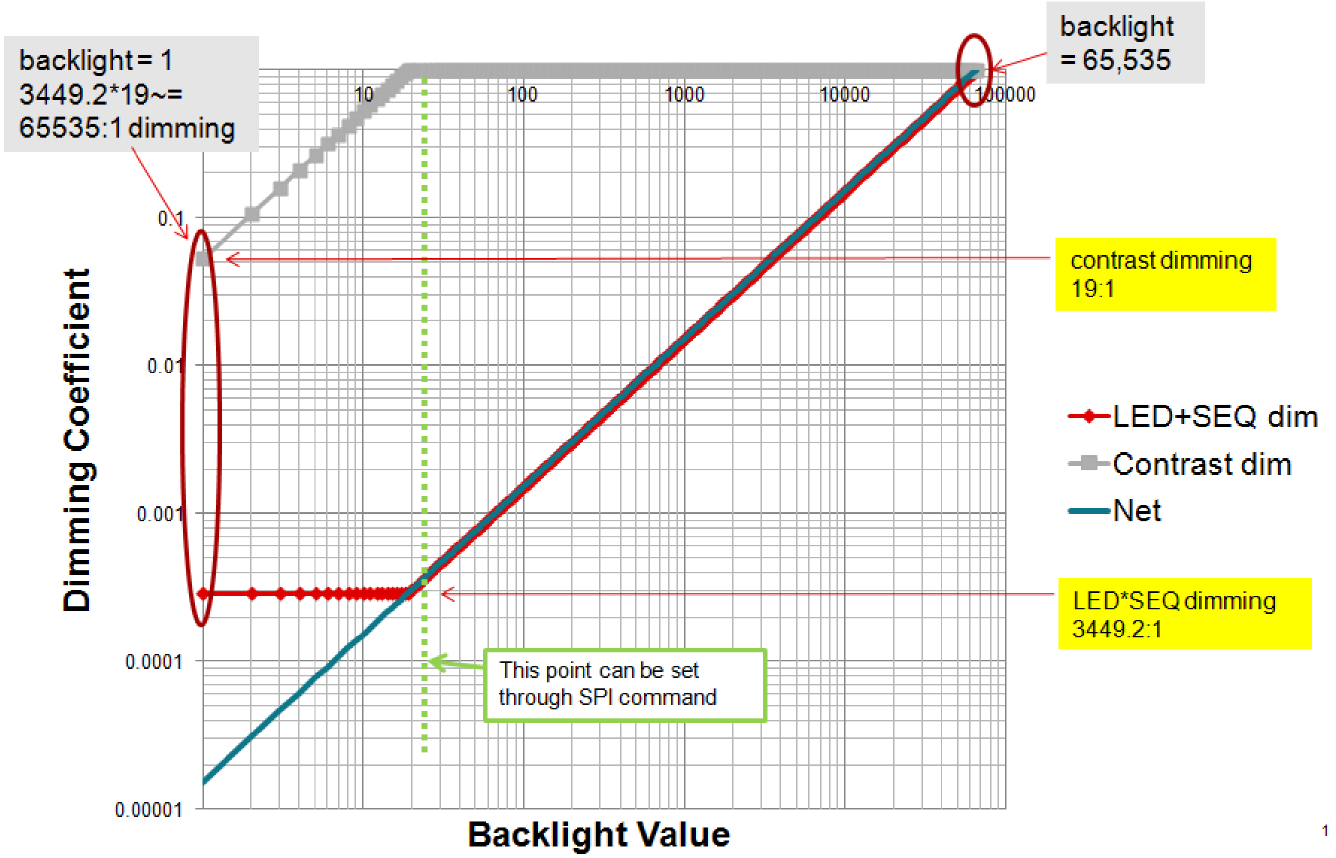


Figure 8-2. Illustration of VAC Dimming

The VAC dimming is on done on a scale of 256. The VAC Coefficient is computed as follows:

$$VAC\ Coefficient = \frac{\text{Required dimming}}{(DM2VACModeSwitchBrightness + 1)} * 256$$

8.4 Temperature Compensation:

Temperature compensation has two main functions:

1. To compensate for variations in LED characteristics with respect to temperature:

   The PWM values may be increased (or decreased) to compensate for the decrease (or increase) in brightness due to change in characteristics of LEDs.

2. To implement Transition Dimming and Low Pass Filter:

   This feature is not needed for operation with the DLP3030-Q1 chipset.

Base Temperature: It is the temperature at which calibration is done. It is also the default temperature used for dimming computations when temperature compensation is disabled.

Active temperature: It is the temperature used by the Piccolo to compute the dimming parameters.

Note: For temperature compensation to function properly, it is necessary that a calibration file with calibration tables for at least two different temperatures is loaded. It is also essential that the calibration tables are arranged in ascending order with respect to temperature, failing which, the software will not permit enabling temperature compensation and will also flag an error.

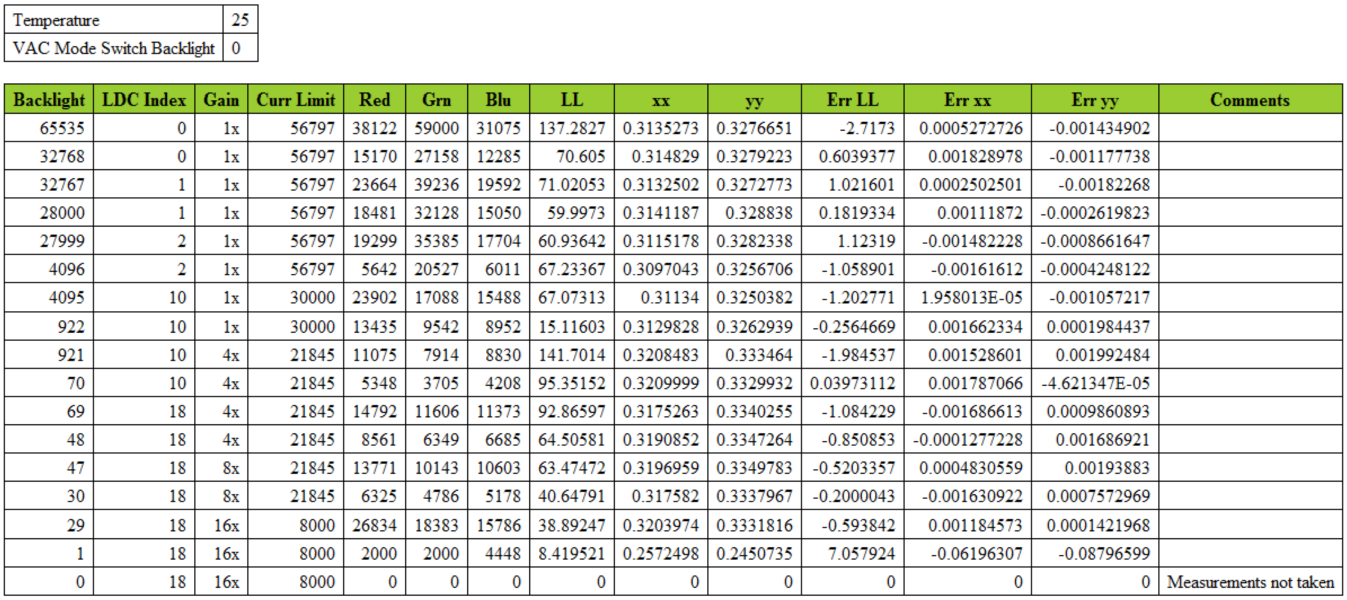
8.5 Calibration Tables:

The calibration tables store the LED PWM values, Sensor Gains, Current Limit PWMs and the LDC index (which also indicates the dimming mode) for different backlights at different temperatures. The Piccolo uses these to compute the PWMs required for achieving any dimming level at a given temperature. One table exists for each temperature at which calibration is done. For intermediate temperatures, the Piccolo interpolates between the closest temperatures available. However, if temperature compensation (see Section 8.4) is disabled, the calibration table for the base temperature is used.

8.5.1 Assumptions regarding calibration tables:

A set of calibration tables is considered valid only when the following conditions are satisfied:

1. The calibration tables must be arranged in ascending order of temperatures
2. For each calibration tables within the same transition region, the boundaries at which the LDC, Sensor Gain or Current Limit PWM vary must be clearly defined. This means that if consecutive rows in a calibration table have different values for any one of the previously parameters, their backlights must be consecutive. This is critical to prevent interpolation between values which were calibrated at different Gain/LDC/Current Limit PWM values. For example if LDC Index is to vary from 0 to 1 at a backlight of 32767 in the calibration table, then entries for backlights 32768 (corresponding to LDC 0) and 32767 (corresponding to LDC 1) must be present.
3. All calibration tables corresponding to the same transition region must have the same boundaries at which LDC, Sensor Gain or Current Limit PWM vary. For instance if 50C and 55C lie in the same transition region and LDC Index varies from 0 to 1 at a backlight of 32767 in the calibration table corresponding to 50C, then it must also change from 0 to 1 in the calibration table for 60C.



Temperature	25
VAC Mode Switch Backlight	0

Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	0	1x	56797	38122	59000	31075	137.2827	0.3135273	0.3276651	-2.7173	0.0005272726	-0.001434902	
32768	0	1x	56797	15170	27158	12285	70.605	0.314829	0.3279223	0.6039377	0.001828978	-0.001177738	
32767	1	1x	56797	23664	39236	19592	71.02053	0.3132502	0.3272773	1.021601	0.0002502501	-0.00182268	
28000	1	1x	56797	18481	32128	15050	59.9973	0.3141187	0.328838	0.1819334	0.00111872	-0.0002619823	
27999	2	1x	56797	19299	35385	17704	60.93642	0.3115178	0.3282338	1.12319	-0.001482228	-0.0008661647	
4096	2	1x	56797	5642	20527	6011	67.23367	0.3097043	0.3256706	-1.058901	-0.00161612	-0.0004248122	
4095	10	1x	30000	23902	17088	15488	67.07313	0.31134	0.3250382	-1.202771	1.958013E-05	-0.001057217	
922	10	1x	30000	13435	9542	8952	15.11603	0.3129828	0.3262939	-0.2564669	0.001662334	0.0001984437	
921	10	4x	21845	11075	7914	8830	141.7014	0.3208483	0.333464	-1.984537	0.001528601	0.001992484	
70	10	4x	21845	5348	3705	4208	95.35152	0.3209999	0.3329932	0.03973112	0.001787066	-4.621347E-05	
69	18	4x	21845	14792	11606	11373	92.86597	0.3175263	0.3340255	-1.084229	-0.001686613	0.0009860893	
48	18	4x	21845	8561	6349	6685	64.50581	0.3190852	0.3347264	-0.850853	-0.0001277228	0.001686921	
47	18	8x	21845	13771	10143	10603	63.47472	0.3196959	0.3349783	-0.5203357	0.0004830559	0.00193883	
30	18	8x	21845	6325	4786	5178	40.64791	0.317582	0.3337967	-0.2000043	-0.001630922	0.0007572969	
29	18	16x	8000	26834	18383	15786	38.89247	0.3203974	0.3331816	-0.593842	0.001184573	0.0001421968	
1	18	16x	8000	2000	2000	4448	8.419521	0.2572498	0.2450735	7.057924	-0.06196307	-0.08796599	
0	18	16x	8000	0	0	0	0	0	0	0	0	0	Measurements not taken

Figure 8-3. Calibration Table

8.6 Sequence Command Table Indexes:

Piccolo software computes three separate index values (LDC, SEQ and CMT) to pick the right set of sequences to be loaded by the DLPC120 ASIC. These indexes are then sent to the ASIC over I2C to start the sequence dimming process. The ASIC decodes the command table index to select the right set of sequences. These tables LDC (LED Drive Control), CMT (Contour Mitigation Table), and SEQ (DMD Sequence Table) are used by the DLPC120 to render a full color image via the DMD PWM timing.

The process of dimming starts with Piccolo receiving the backlight value through SPI. The software then computes the dimming parameters for LED PWM values, Sensor Gain and the sequence command table indexes to be set in the DLPC120 ASIC to achieve the requested dimming level. Once the parameters are computed, the software sends a request via I2C to the ASIC to start the sequence update process. The ASIC will complete the sequence update in one frame time. Piccolo will wait for the next frame sync and will apply the LED PWMs and set the sensor gain to complete the dimming process.

In order to streamline the dimming process and get the complete process to finish in a single frame, the sequence table updates are pipelined in hardware. The ASIC processes LDC, SEQ and CMT updates in single frame, but for different dimming values.

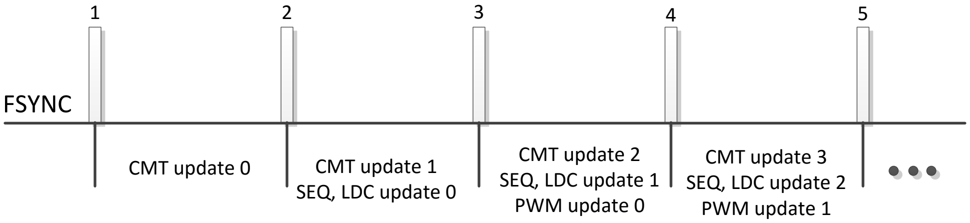


Figure 8-4. CMT, SEQ, PWM Pipelining

As seen in Figure 8-4, every frame processes both CMT and SEQ, LDC combination. The SEQ, LDC combination update always lags behind the CMT update by one frame. Or in other words, the SEQ, LDC update happening in current frame is for the dimming command received one frame earlier and whose CMT update happened in previous frame. The first frame (frame is counted after receiving a dimming command) however has only updates to CMT tables. The LED PWM update (done by Piccolo) lags one more frame to SEQ, LDC update, and happens immediately after the completion of the SEQ, LDC update. This pipelining is achieved through the help of three ASIC registers – CMT Table Index (0xD0), SEQ Table Index (0xD3) and LDC Table Index (0xD6). The software is responsible for updating these ASIC registers at appropriate frames to facilitate smooth pipelining. Please refer to **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**for more details on these ASIC registers.

8.7 Receive the Dimming Command

On receiving a valid dimming command (also known as Backlight command), the software first checks if calibration data is available in Piccolo flash memory. If it is present, the software goes computes the dimming parameters. The parameters computed are then stored to an internal software queue. The control flow is shown in Figure 8-5.

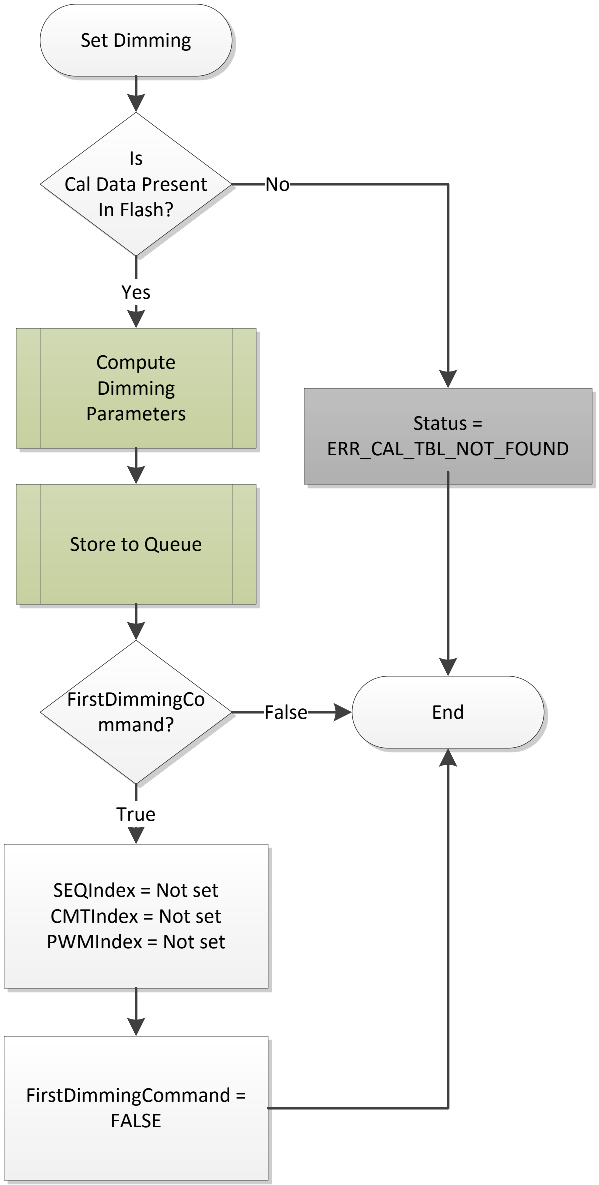


Figure 8-5. Dimming Command

The software uses several global variables to identify the first frame and manage the pipelining explained in Figure 8-4. `FirstDimmingCommand` variable is used to mark the case where no dimming commands were received by the software. This variable is set to TRUE during system power on and also when exiting from calibration mode. As soon as the first dimming command is received, the state of this variable is changed to FALSE.

Three variables - `SEQIndex`, `PWMIndex`, and `CMTIndex` - are used to manage the pipelining. These variables should not be confused with the sequence command table index registers referred to in earlier sections. These variables index into an array called `DIM_Params[]` which holds the dimming parameters fetched from queue for the last two frames. These variables are cleared when the first dimming command is received. To get a full understanding of these variables, please refer to Section 8.10 and Section 8.12.

The related C functions are:

- `Uint16 DIM_SetDimming(Uint16 DimVal)`
- `void DIM_GetDimming(Uint16 *DimVal)`

8.8 Computation of the Dimming Parameters

The Piccolo computes the following dimming parameters (`DIM_Params[]`):

- Operating Mode
- R, G, B LED PWMs
- Current Limit PWM
- VAC Coefficient
- Sensor Gain
- Sequence Command Table Indexes (SEQNo, LDCNo and CMTNo)

These parameters along with the dimming value are stored to the queue after the dimming computation.

8.9 Steps Involved in Computation of Dimming Parameters:

The dimming algorithm comprises of the following steps:

Step 1: Using the active temperature to decide which temperature tables need to be used for interpolation. The term ‘Temperature Table’ will be used to refer to the calibration table corresponding to any one temperature.

Step 2: Computation of VAC Mode Switch Point.

Step 3: Determination of target backlight and VAC Coefficient using VAC Mode Switch Point, if VAC is enabled.

Step 4: Determination of the rows to be used for interpolation for each temperature table selected.

The operating mode, LDC Index, Current Limit PWM values and Sensor Gain are also computed in this step.

Step 5: Interpolation to compute the Red, Green, and Blue PWM Values corresponding to the target backlight.

Step 6: Calculation of SEQ Index and CMT index using the LDC Index.

Each step is explained in further detail in the following sections. Figure 8-6 illustrates the steps involved in the computation of dimming parameters.

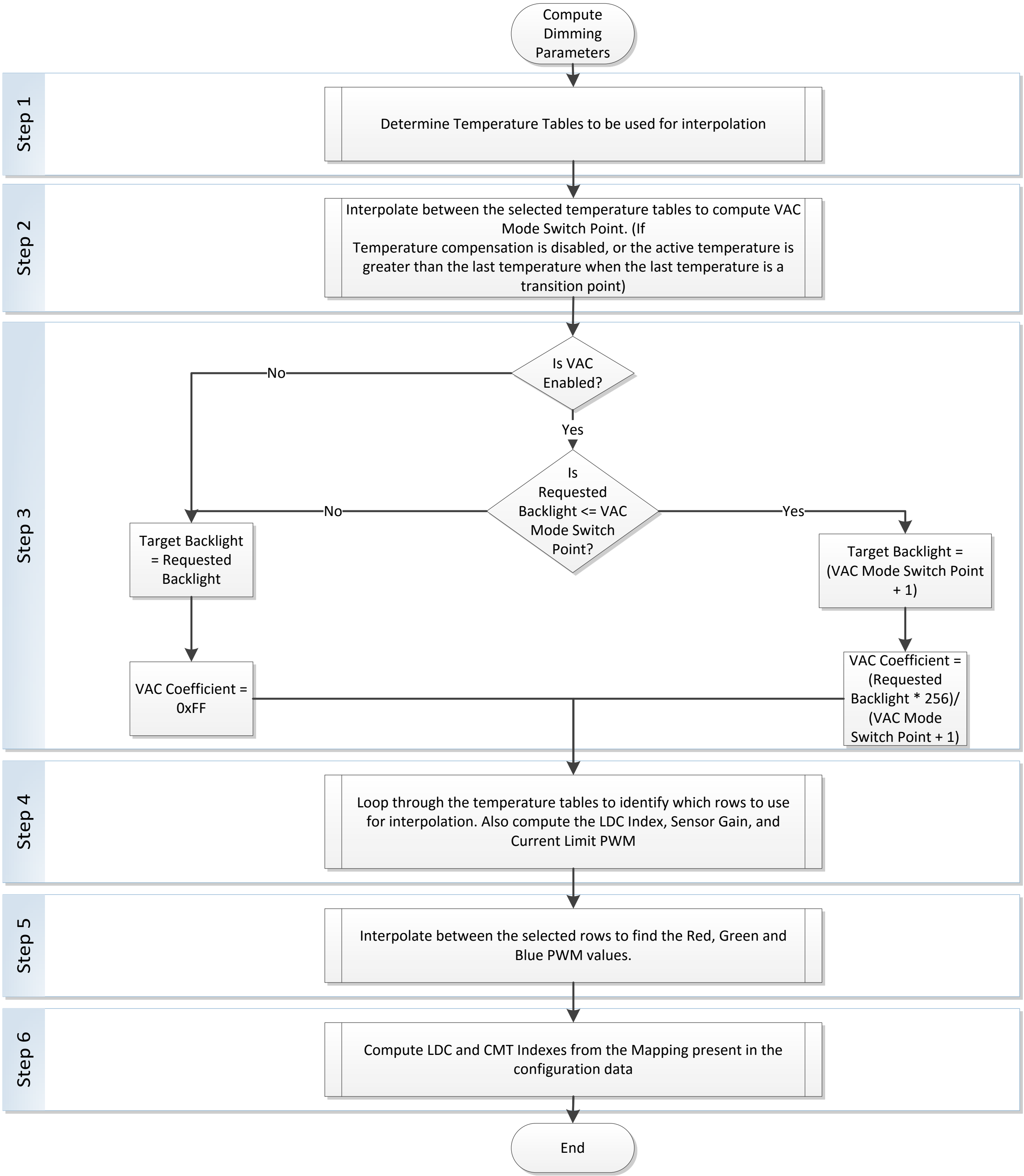

Compute Dimming Parameters
Step 1
Determine Temperature Tables to be used for interpolation
Step 2
Interpolate between the selected temperature tables to compute VAC Mode Switch Point. (If Temperature compensation is disabled, or the active temperature is greater than the last temperature when the last temperature is a transition point)
Step 3
Is VAC Enabled?
No
Yes
Is Requested Backlight <= VAC Mode Switch Point?
No
Yes
Target Backlight = Requested Backlight
Target Backlight = (VAC Mode Switch Point + 1)
VAC Coefficient = 0xFF
VAC Coefficient = (Requested Backlight * 256)/ (VAC Mode Switch Point + 1)
Step 4
Loop through the temperature tables to identify which rows to use for interpolation. Also compute the LDC Index, Sensor Gain, and Current Limit PWM
Step 5
Interpolate between the selected rows to find the Red, Green and Blue PWM values.
Step 6
Compute LDC and CMT Indexes from the Mapping present in the configuration data
End

Figure 8-6. Computation of dimming parameters

8.9.1 Step 1: Using the active temperature to decide which temperature tables need to be used for interpolation.

The active temperature refers to the ambient temperature filtered using a low pass filter. The temperature tables to be used by the consequent steps are chosen using the following conditions:

1. If temperature compensation is disabled, only the base temperature table is used. The Piccolo sets both temperature tables to point to the base temperature table and hence no interpolation is required.
2. If the temperature is greater than (or equal to) the highest temperature for which calibration table is available, the last two temperature tables are used. This will result in extrapolation rather than interpolation. However if the last temperature is a transition temperature, only the last temperature table (both selections point to the last temperature table) is used and no interpolation will be done.
3. If the temperature is less than (or equal to) the lowest temperature for which calibration table is available, the first two temperature tables are used. This will also result in extrapolation.
4. If the temperature lies within the range of temperatures for which data is available, the temperature corresponding to the nearest temperature less than the active temperature and the nearest temperature greater than the active temperature are chosen for interpolation.
5. Care should be taken that interpolation is not done across transition regions. This is because temperature tables belonging to different transition regions are likely to be calibrated using different LDC Index, Sensor Gain, etc., and hence interpolation may give wrong results. For the DLP3030-Q1, there is no requirement to have multiple transition regions. The same duty cycle can be used across the entire temperature range.

In the Piccolo software the conditions 2, 3 and 4 are evaluated by running a ‘for’ loop from the second lowest temperature to the second highest temperature and checking if active temperature is in the range bounded by the current (the one to which the control variable is pointing to) temperature and the adjacent temperature. However condition 5 and the sub-case in condition 2 involving the last temperature being a transition temperature are evaluated separately. Figure 8-7 illustrates this step.

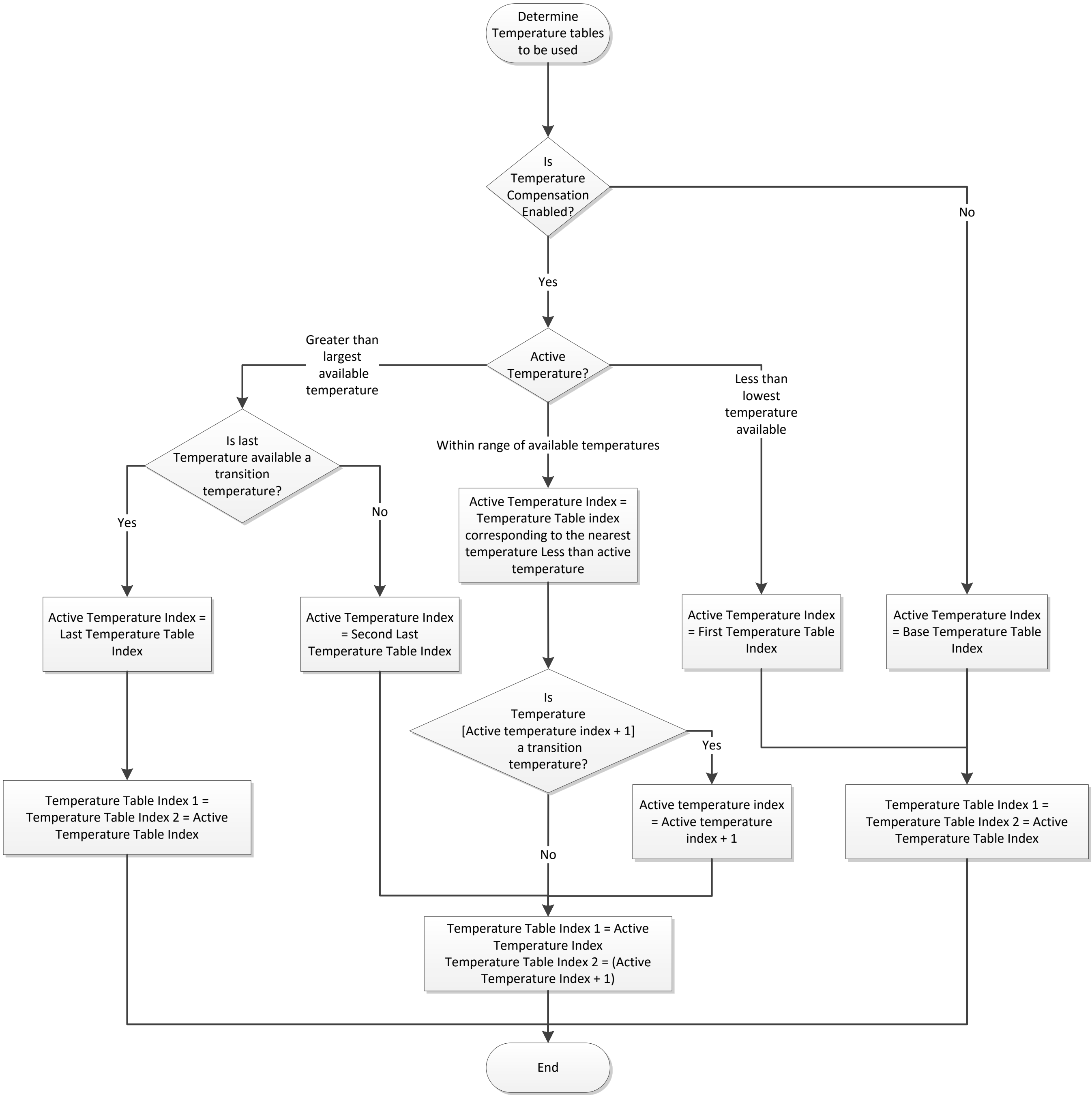

Determine Temperature tables to be used
Is Temperature Compensation Enabled?
No
Yes
Active Temperature?
Greater than largest available temperature
Less than lowest temperature available
Within range of available temperatures
Is last Temperature available a transition temperature?
Yes
No
Active Temperature Index = Temperature Table index corresponding to the nearest temperature Less than active temperature
Active Temperature Index = Last Temperature Table Index
Active Temperature Index = Second Last Temperature Table Index
Active Temperature Index = First Temperature Table Index
Active Temperature Index = Base Temperature Table Index
Is Temperature [Active temperature index + 1] a transition temperature?
Yes
No
Temperature Table Index 1 = Temperature Table Index 2 = Active Temperature Table Index
Active temperature index = Active temperature index + 1
Temperature Table Index 1 = Temperature Table Index 2 = Active Temperature Table Index
Temperature Table Index 1 = Active Temperature Index Temperature Table Index 2 = (Active Temperature Index + 1)
End

Figure 8-7. Determining the temperature tables to be used for interpolation

The related C functions are:

- `static Uint16 TMPR_UpdateActiveTmpr    (Uint16 TmprK10)`

8.9.2 Step 2: Computation of VAC Mode Switch Point

The VAC Mode Switch Point is the backlight at which the system switches (only when VAC is enabled) to VAC mode. The Mode Switch Point is a temperature dependent parameter. For a given active temperature, it is computed by interpolating/extrapolating between the Mode Switch Points corresponding to the temperature tables selected in the previous step. If temperature compensation is disabled, or if the active temperature is greater than the highest temperature for which data is available and this highest temperature is a transition temperature, no interpolation is required and the Mode Switch Point of the corresponding table can be used directly.

The related C functions are:

- `static Uint16    TMPR_UpdateActiveTmpr(Uint16 TmprK10)`

8.9.3 Step 3: Determination of target backlight and VAC Coefficient using VAC Mode Switch Point

If VAC is enabled and the requested backlight is less than the computed Mode Switch Point, the target backlight should be set as (Mode Switch Point + 1). The VAC Coefficient is determined by the formula:

$$VAC\ Coefficient = \frac{\text{Requested backlight}}{(VAC\ Mode\ Switch\ Point + 1)} * 256$$

In all other cases the target backlight is same as the requested backlight and the VAC coefficient is set to 255 (0xFF).

The related C functions are:

- `static Uint16    DIM_GetDimmingValues(Uint16 Backlight, DIM_PARAMETERS *DimParams)`

8.9.4 Step 4: Determination of the rows to be used for interpolation for each temperature table selected

For each temperature table selected, the row corresponding to the nearest backlight less than and the row corresponding to the nearest backlight greater (or equal to) than the target backlight are selected for interpolation.

If the assumptions 3 and 4 regarding calibration tables (discussed in Section 8.5.1) are correct all four rows chosen will have the same LDC, Sensor Gain and Current Limit PWM values except when the target backlight is equal to the lower boundary below which any of these parameters change. In this case the lower row indexes (rows corresponding to higher backlight) contain the correct PWM's. Hence in general applying the LDC, Sensor Gain, and the Current Limit PWM's corresponding to the row corresponding to the higher backlight for the lower temperature table should always give the right result.

The related C functions are:

- `Uint16    TMPR_GetInterpolationRowIndex(Uint16 Backlight, Uint16 TmprIndex, Uint16 *CalIndex)`
- `Uint16 TMPR_ComputeDimmingValues    (Uint16 Backlight, volatile CAL_CALDATA *DimCoeff)`

8.9.5 Step 5: Interpolation to compute the Red, Green and Blue PWM Values corresponding to the target backlight

The 4 selected rows are then used for interpolation to compute the Red, Green and Blue LED PWM's for the required backlight at the given temperature. Suppose T1 and T2 are the two temperatures whose tables are selected. The interpolation is done in two steps:

1. The PWMs are linearly interpolated for the given backlight for T1 and T2 individually
2. The PWM's at the required temperature are computed by linearly interpolating between values corresponding to T1 and T2.

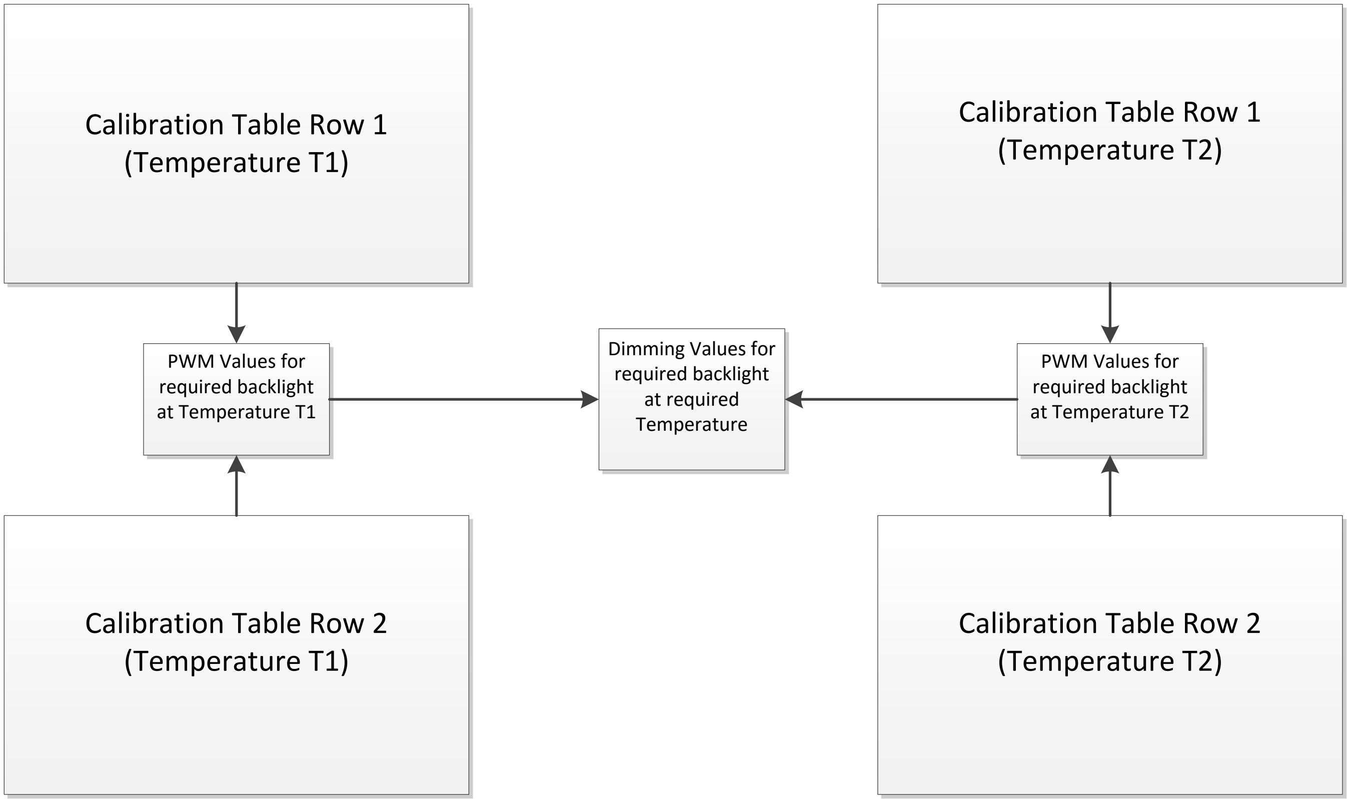

Calibration Table Row 1
(Temperature T1)
Calibration Table Row 1
(Temperature T2)
PWM Values for required backlight at Temperature T1
Dimming Values for required backlight at required Temperature
PWM Values for required backlight at Temperature T2
Calibration Table Row 2
(Temperature T1)
Calibration Table Row 2
(Temperature T2)

Figure 8-8. Illustrating the Interpolation Process

The related C functions are:

- `static void    TMPR_GetPWMCoefficients(Uint16 Backlight, Uint16 TmprCalIndex, PWM_VALUES *Pwm)`
- `Uint16 TMPR_ComputeDimmingValues(Uint16    Backlight, volatile CAL_CALDATA *DimCoeff)`

8.9.6 Step 6: Calculation of SEQ Index and CMT index using the LDC Index

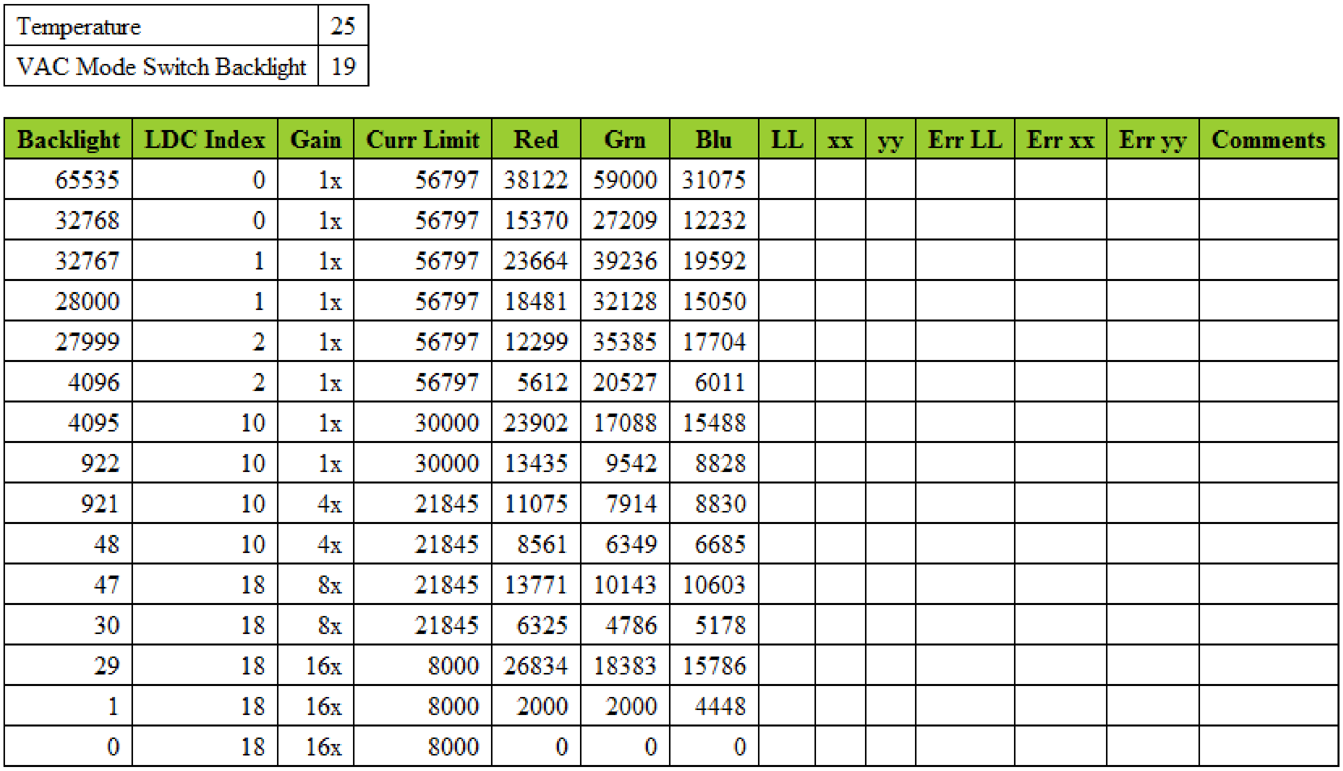
The SEQ and CMT index are calculated by the LDC-SEQ map, SEQ-CMT Map present in the Configuration file.

It is important to know that in the Piccolo software, all 6 steps are not necessarily carried out simultaneously. Steps 1 and 2 are carried out by the `TMPR_CheckUpdate()` function called in the main loop. The temperature is sampled at regular intervals, and the active temperature is determined. The parameters for the current dimming level are also updated and stored in queue. When a new dimming command is received, steps 1 and 2 are not repeated. The last calculated active temperature, VAC Mode Switch Point and the selected temperature tables for interpolation are directly used.

8.10 Examples of Dimming Computations:

8.10.1 Sample Calibration Tables:

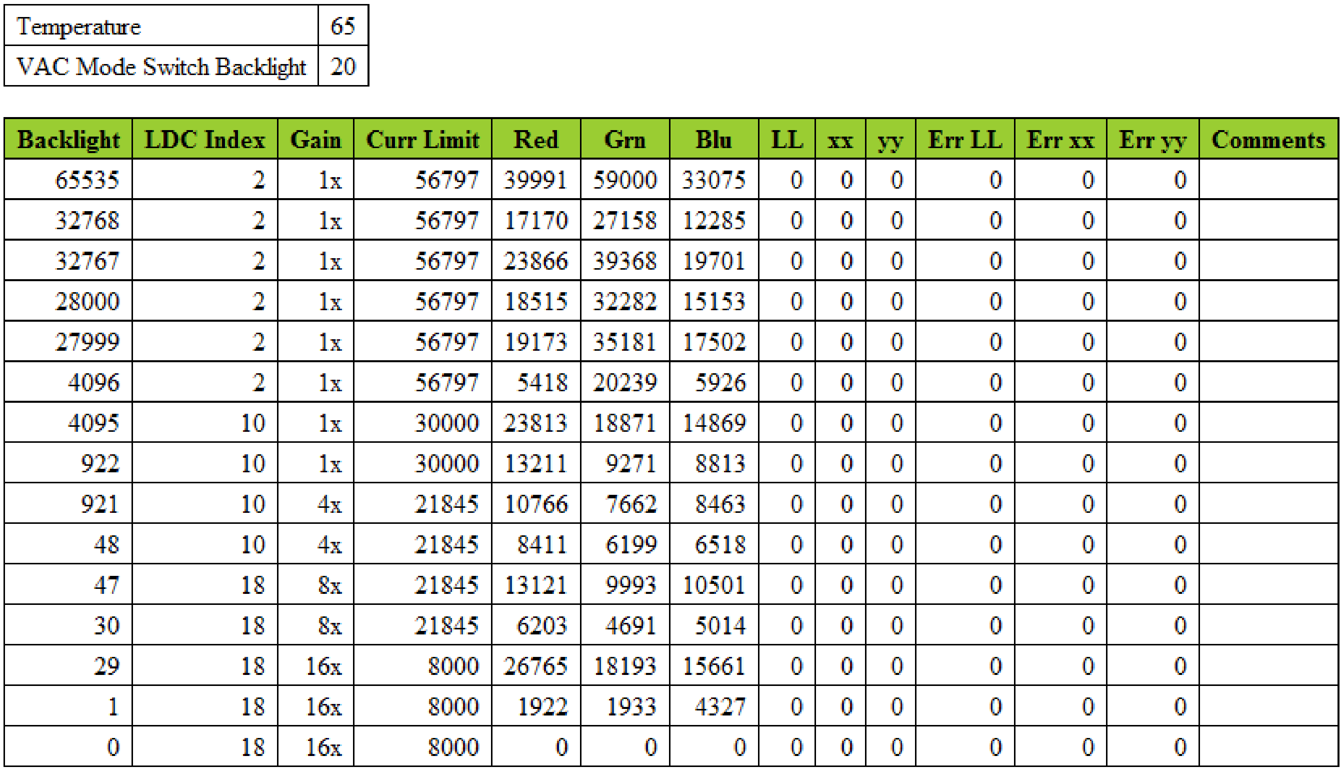
The following calibration tables will be used for computations in the following examples. Please note that these contain manually written values and might not be fit for use in real situations.



Temperature	25
VAC Mode Switch Backlight	19

Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	0	1x	56797	38122	59000	31075							
32768	0	1x	56797	15370	27209	12232							
32767	1	1x	56797	23664	39236	19592							
28000	1	1x	56797	18481	32128	15050							
27999	2	1x	56797	12299	35385	17704							
4096	2	1x	56797	5612	20527	6011							
4095	10	1x	30000	23902	17088	15488							
922	10	1x	30000	13435	9542	8828							
921	10	4x	21845	11075	7914	8830							
48	10	4x	21845	8561	6349	6685							
47	18	8x	21845	13771	10143	10603							
30	18	8x	21845	6325	4786	5178							
29	18	16x	8000	26834	18383	15786							
1	18	16x	8000	2000	2000	4448							
0	18	16x	8000	0	0	0							

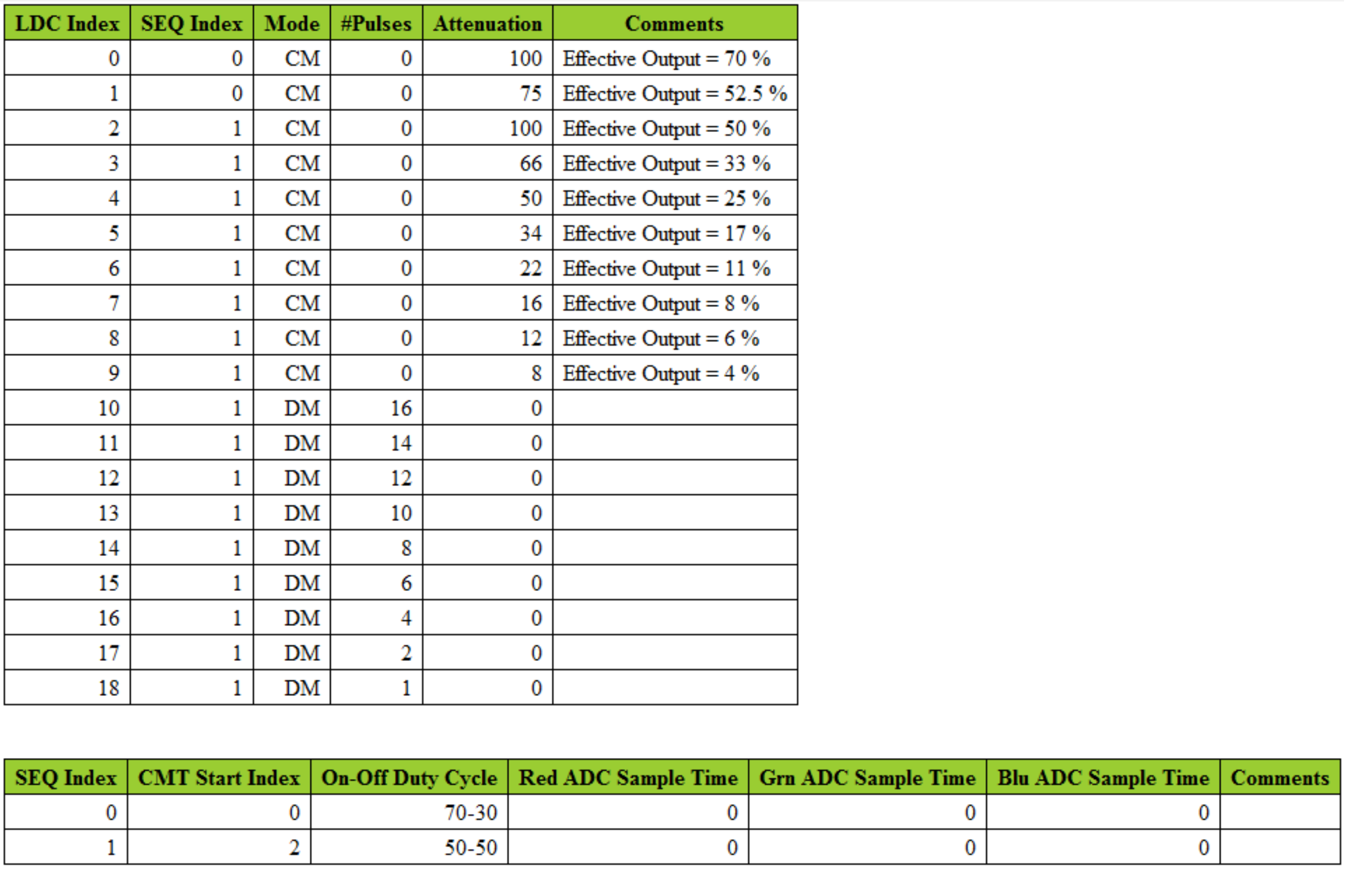
Figure 8-9. Sample Calibration Table for 25°C- Base Temperature



Temperature	65
VAC Mode Switch Backlight	20

Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	2	1x	56797	39991	59000	33075	0	0	0	0	0	0	
32768	2	1x	56797	17170	27158	12285	0	0	0	0	0	0	
32767	2	1x	56797	23866	39368	19701	0	0	0	0	0	0	
28000	2	1x	56797	18515	32282	15153	0	0	0	0	0	0	
27999	2	1x	56797	19173	35181	17502	0	0	0	0	0	0	
4096	2	1x	56797	5418	20239	5926	0	0	0	0	0	0	
4095	10	1x	30000	23813	18871	14869	0	0	0	0	0	0	
922	10	1x	30000	13211	9271	8813	0	0	0	0	0	0	
921	10	4x	21845	10766	7662	8463	0	0	0	0	0	0	
48	10	4x	21845	8411	6199	6518	0	0	0	0	0	0	
47	18	8x	21845	13121	9993	10501	0	0	0	0	0	0	
30	18	8x	21845	6203	4691	5014	0	0	0	0	0	0	
29	18	16x	8000	26765	18193	15661	0	0	0	0	0	0	
1	18	16x	8000	1922	1933	4327	0	0	0	0	0	0	
0	18	16x	8000	0	0	0	0	0	0	0	0	0	

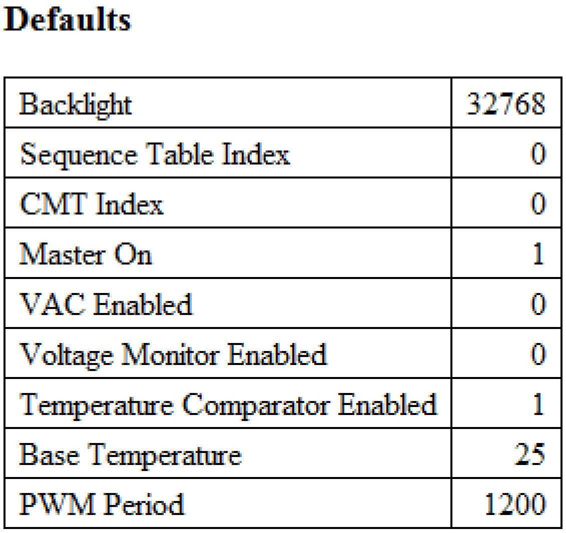
Figure 8-10. Sample Calibration Table for 65°C



LDC Index	SEQ Index	Mode	#Pulses	Attenuation	Comments
0	0	CM	0	100	Effective Output = 70 %
1	0	CM	0	75	Effective Output = 52.5 %
2	1	CM	0	100	Effective Output = 50 %
3	1	CM	0	66	Effective Output = 33 %
4	1	CM	0	50	Effective Output = 25 %
5	1	CM	0	34	Effective Output = 17 %
6	1	CM	0	22	Effective Output = 11 %
7	1	CM	0	16	Effective Output = 8 %
8	1	CM	0	12	Effective Output = 6 %
9	1	CM	0	8	Effective Output = 4 %
10	1	DM	16	0	
11	1	DM	14	0	
12	1	DM	12	0	
13	1	DM	10	0	
14	1	DM	8	0	
15	1	DM	6	0	
16	1	DM	4	0	
17	1	DM	2	0	
18	1	DM	1	0	

SEQ Index	CMT Start Index	On-Off Duty Cycle	Red ADC Sample Time	Grn ADC Sample Time	Blu ADC Sample Time	Comments
0	0	70-30	0	0	0	
1	2	50-50	0	0	0	

Figure 8-11. Sample LDC-SEQ and SEQ-CMT Map



Defaults

Backlight	32768
Sequence Table Index	0
CMT Index	0
Master On	1
VAC Enabled	0
Voltage Monitor Enabled	0
Temperature Comparator Enabled	1
Base Temperature	25
PWM Period	1200

Figure 8-12. Sample Default Data

8.10.2 Example 1: Backlight = 60000, Temperature Compensation Disabled, VAC Enabled

Step 1: Using the active temperature to decide which temperature tables need to be used for interpolation:

Since temperature compensation is disabled, only the base temperature is used. 25°C is the first temperature in the table (index 0).

- Temperature Table Index 1 = 0
- Temperature Table Index 2 = 0

Step 2: Computation of VAC Mode Switch Point:

For 25°C (no interpolation required as Temperature Compensation is disabled),

- VAC Mode Switch Point = 19

Step 3: Determination of target backlight and VAC Coefficient using VAC Mode Switch Point:

Though VAC is enabled, since Requested Backlight > VAC Mode Switch Point,

- Target Backlight = Requested Backlight = 60000
- VAC Coefficient = 255 (0xFF)

Step 4: Determination of the rows to be used for interpolation for each temperature table selected:

For the Temperature Table Index 1:

60000 lies between rows corresponding to backlights 65535 and 32768.

Backlight	LDC	Sensor Gain	Current Limit	Red PWM	Green PWM	Blue PWM
65535	0	1x	56797	38122	59000	31075
32768	0	1x	56797	15370	27209	12232

Rows 0 and 1 need to be used for interpolation.

These two rows are used for both Temperature Table Index 1 and 2.

- LDC Index = 0
  - Data corresponding to Row 0 of Temperature Table Index 1
- Sensor Gain = 1x
  - Data corresponding to Row 0 of Temperature Table Index 1
- Current Limit PWM = 56797
  - Data corresponding to Row 0 of Temperature Table Index 1

Step 5: Interpolation to compute the Red, Green and Blue PWM Values corresponding to the target backlight:

For each temperature table the two selected rows are used to compute the LED PWMs. In this case for both rows the values are:

Red PWM = Interpolation (38122, 15370) = 34279

Green PWM = Interpolation (59000, 27209) = 53630

Blue PWM = Interpolation (31075, 12232) = 27892

Step 6: Calculation of SEQ Index and CMT index using the LDC Index:

LDC Index = 0 (Computed in step 4)

From the LDC-SEQ Map in the Sample tables,

SEQ Index = 0

From the SEQ-CMT Map in the Sample tables,

CMT Index = 0

Result:

1. Red PWM = 34279
2. Green PWM = 53630
3. Blue PWM = 27892
4. Current Limit = 56797
5. Sensor Gain = 1x
6. LDC Index = 0
7. SEQ Index = 0
8. CMT Index = 0
9. VAC Coefficient = 255

8.10.3 Example 2: Backlight = 1200, Temperature Compensation Enabled, Active Temperature = 45°C, VAC Enabled

Step 1: Using the active temperature to decide which temperature tables need to be used for interpolation:

The active temperature lies within the range of available temperatures. The nearest bounding temperature tables are used for interpolation.

In this case 45°C lies between 25°C and 64.9°C.

- Temperature Table Index 1 = 0
  - Corresponding to 25°C
- Temperature Table Index 2 = 1
  - Corresponding to 64.9°C
- T1 = 25°C
- T2 = 64.9°C

Step 2: Computation of VAC Mode Switch Point:

Mode Switch Point for 45°C is obtained by interpolating between the selected tables.

VAC Mode Switch Point = Interpolation (19, 17) for temperature 45°C

- VAC Mode Switch Point = 18

Step 3: Determination of target backlight and VAC Coefficient using VAC Mode Switch Point:

Though VAC is enabled, since Requested Backlight > VAC Mode Switch Point,

Target Backlight = Requested Backlight = 1200

- VAC Coefficient = 255 (0xFF)

Step 4: Determination of the rows to be used for interpolation for each temperature table selected:

For the Temperature Table Index 1 (25°C):

1200 lies between rows corresponding to backlights 4095 and 922.

Hence rows 6 and 7 need to be used for interpolation.

Backlight	LDC	Sensor Gain	Current Limit	Red PWM	Green PWM	Blue PWM
4095	10	1x	30000	23902	17088	15488
922	10	1x	30000	13435	9542	8828

For the Temperature Table Index 2 (64.9°C):

1200 lies between rows corresponding to backlights 2000 and 922.

Hence rows 7 and 8 need to be used for interpolation.

Backlight	LDC	Sensor Gain	Current Limit	Red PWM	Green PWM	Blue PWM
2000	10	1x	30000	19022	14133	12462
922	10	1x	30000	13223	9272	8828

- LDC Index = 10
  - Data corresponding to Row 6 of Temperature Table Index 1
- Sensor Gain = 1x
  - Data corresponding to Row 6 of Temperature Table Index 1
- Current Limit PWM = 30000
  - Data corresponding to Row 6 of Temperature Table Index 1

Step 5: Interpolation to compute the Red, Green and Blue PWM Values corresponding to the target backlight:

For Temperature Table 1:

Red PWM = Interpolation (23902, 13435) for backlight 1200 = 14352

Green PWM = Interpolation (17088, 9542) for backlight 1200 = 10203

Blue PWM = Interpolation (15488, 8828) for backlight 1200 = 9412

For Temperature Table 2:

Red PWM = Interpolation (19022, 13223) for backlight 1200 = 14718

Green PWM = Interpolation (14133, 9272) for backlight 1200 = 10526

Blue PWM = Interpolation (12462, 8828) for backlight 1200 = 9765

Interpolating across temperatures,

Red PWM = Interpolation (14352, 14718) for temperature 45°C = 14535

Green PWM = Interpolation (10203, 10526) for temperature 45°C = 10365

Blue PWM = Interpolation (9412, 9765) for temperature 45°C = 9589

Step 6: Calculation of SEQ Index and CMT index using the LDC Index:

LDC Index = 10 (Computed in step 4)

From the LDC-SEQ Map in the Sample tables,

- SEQ Index = 1

From the SEQ-CMT Map in the Sample tables,

- CMT Index = 2

Result:

1. Red PWM = 14535
2. Green PWM = 10365
3. Blue PWM = 9589
4. Current Limit = 30000
5. Sensor Gain = 1x
6. LDC Index = 10
7. SEQ Index = 1
8. CMT Index = 2
9. VAC Coefficient = 255

8.10.4 Example 3: Backlight = 12, Temperature Compensation Enabled, Active Temperature = 45°C, VAC Enabled

Step 1: Using the active temperature to decide which temperature tables need to be used for interpolation:

The active temperature lies within the range of available temperatures. Hence the nearest temperature tables corresponding to the nearest lower and higher temperatures are used.

In this case 45°C lies between 25°C and 64.9°C.

- Temperature Table Index 1 = 0
  - Corresponding to 25°C
- Temperature Table Index 2 = 1
  - Corresponding to 64.9°C
- T1 = 25°C
- T2 = 64.9°C

Step 2: Computation of VAC Mode Switch Point:

Mode Switch Point for 45°C is obtained by interpolating between the selected tables.

VAC Mode Switch Point = Interpolation (19, 17) for temperature 45°C

- VAC Mode Switch Point = 18

Step 3: Determination of target backlight and VAC Coefficient using VAC Mode Switch Point:

VAC is enabled and Requested Backlight

Target Backlight = (VAC Mode Switch Point + 1) = 19

$$VAC\ Coefficient = \frac{\text{Requested backlight}}{(VAC\ Mode\ Switch\ Point + 1)} * 256$$

- $VAC\ Coefficient = \frac{12}{19} * 256$

VAC Coefficient = 162 (0xA2)

Step 4: Determination of the rows to be used for interpolation for each temperature table selected:

For the Temperature Table Index 1 (25°C):

19 lies between rows corresponding to backlights 29 and 1.

Hence rows 12 and 13 need to be used for interpolation.

Backlight	LDC	Sensor Gain	Current Limit	Red PWM	Green PWM	Blue PWM
29	18	16x	8000	26834	18383	15786
1	18	16x	8000	2000	2000	4448

For the Temperature Table Index 2 (64.9°C):

19 lies between rows corresponding to backlights 29 and 1.

Hence rows 13 and 14 need to be used for interpolation.

Backlight	LDC	Sensor Gain	Current Limit	Red PWM	Green PWM	Blue PWM
29	18	16x	8000	26773	18204	15666
1	18	16x	8000	1929	1944	4333

- LDC Index = 18
  - Data corresponding to Row 12 of Temperature Table Index 1
- Sensor Gain = 16x
  - Data corresponding to Row 12 of Temperature Table Index 1
- Current Limit PWM = 8000
  - Data corresponding to Row 12 of Temperature Table Index 1

Step 5: Interpolation to compute the Red, Green and Blue PWM Values corresponding to the target backlight:

For Temperature Table 1:

Red PWM = Interpolation (26834, 2000) for backlight 19 = 17965

Green PWM = Interpolation (18383, 2000) for backlight 19 = 12532

Blue PWM = Interpolation (15786, 4448) for backlight 19 = 11737

For Temperature Table 2:

Red PWM = Interpolation (26773, 1929) for backlight 19 = 17900

Green PWM = Interpolation (18204, 1944) for backlight 19 = 12397

Blue PWM = Interpolation (15666, 4333) for backlight 19 = 11619

Interpolating across temperatures,

Red PWM = Interpolation (17965, 17900) for temperature 45°C = 17932

Green PWM = Interpolation (12532, 12397) for temperature 45°C = 12464

Blue PWM = Interpolation (11737, 11619) for temperature 45°C = 11678

Step 6: Calculation of SEQ Index and CMT index using the LDC Index:

LDC Index = 18 (Computed in step 4)

From the LDC-SEQ Map in the Sample tables,

SEQ Index = 1

From the SEQ-CMT Map in the Sample tables,

CMT Index = 2

Result:

1. Red PWM = 17932
2. Green PWM = 12464
3. Blue PWM = 11678
4. Current Limit = 8000
5. Sensor Gain = 16x
6. LDC Index = 18
7. SEQ Index = 1
8. CMT Index = 2
9. VAC Coefficient = 162

8.11 Sequence Update

Once the parameters are stored to queue, the next step in the line to achieve the dimming is to update the sequences. The sequence update process happens inside the FSYNC interrupt processing after the application of PWMs and sensor gain for the previous dimming command.

To support pipelining of updates, the `DIM_Params[]` array is used to hold the last two set of dimming data fetched from Dimming queue. The three index variables namely `CMTIndex`, `SEQIndex` and `PWMIndex` are used to index into the `DIM_Params[]` array to select the parameters to apply in each frame. The first frame after a command will update CMT, Next frame will update SEQ and the third frame will update the PWM corresponding to the first dimming command.

Initially all three variables are set with invalid values (0xFF in code) which is mentioned as “Not Set” state in the diagrams and explanations shown in previous sections of this document. The valid values are 0 and 1 (array indexes). When the first dimming command is received, the state of all three index variables will be 0xFF (not-set/invalid). This indicates that we are in first frame of the first dimming command. The `CMTIndex` is set to 0 (first array index) and then the data is fetched from queue into the `DIM_Params[CMTIndex]` or in other words the first location of `DIM_Params[]` array. The CMTNo is then written to CMT ASIC register. After this the `SEQIndex` variable is updated to 0 (same as `CMTIndex`) Thus in the second frame, `SEQIndex` has a valid value (0), and `DIM_Params[SEQIndex]` contains the same dimming data that was used while doing CMT update. In this second frame, both SEQNo and LDCNo that were computed during dimming and stored in `DIM_Params[SEQIndex]` are written to ASIC registers. Once the SEQ and LDC update is also done in this second frame, the `PWMIndex` is set to `SEQIndex`. The third frame will update the PWMs from `DIM_Params[PWMIndex]`. This continues on and on. The Figure 8-13 shows how the values of all three indexes change.

The VAC register is also updated in the same frame as CMTNo update. This update also saves a flag to indicate whether the VAC register data was modified or not. This flag is used when the system is in splash dimming mode (Chapter 9).

The sequence update control flow is shown in Figure 8-14. The VAC update control flow is shown in Figure 8-15.

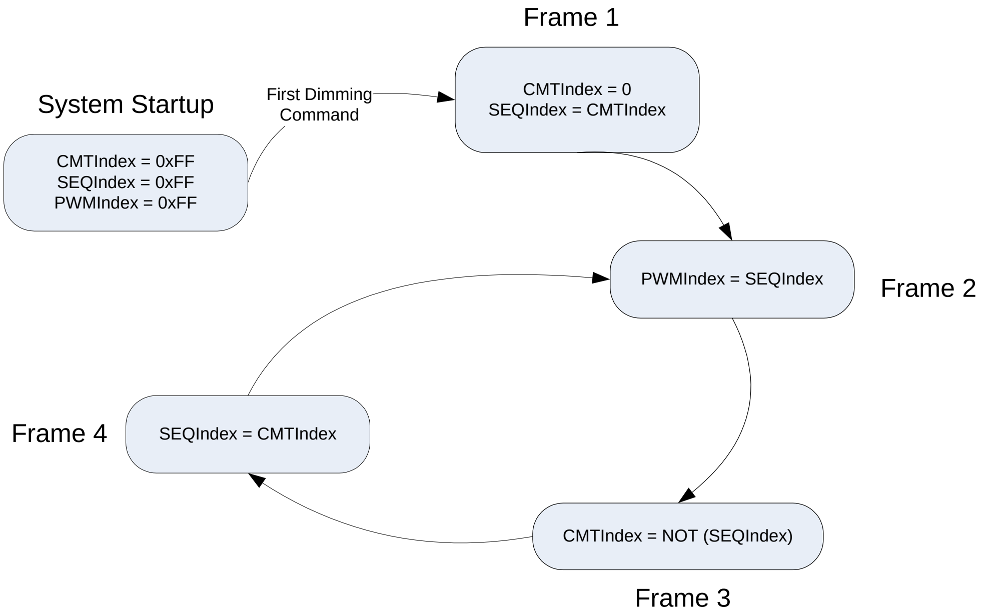


Figure 8-13. Index Variable

The related C functions are:

- `void DIM_UpdateSequences(void)`
- `Uint16 DIMQ_FetchParams(DIM_PARAMETERS *Value)`
- `Uint16 DIMQ_GetLastFetch(DIM_PARAMETERS *Value)`
- `static BOOL DIM_UpdateVAC(Uint16 Index)`

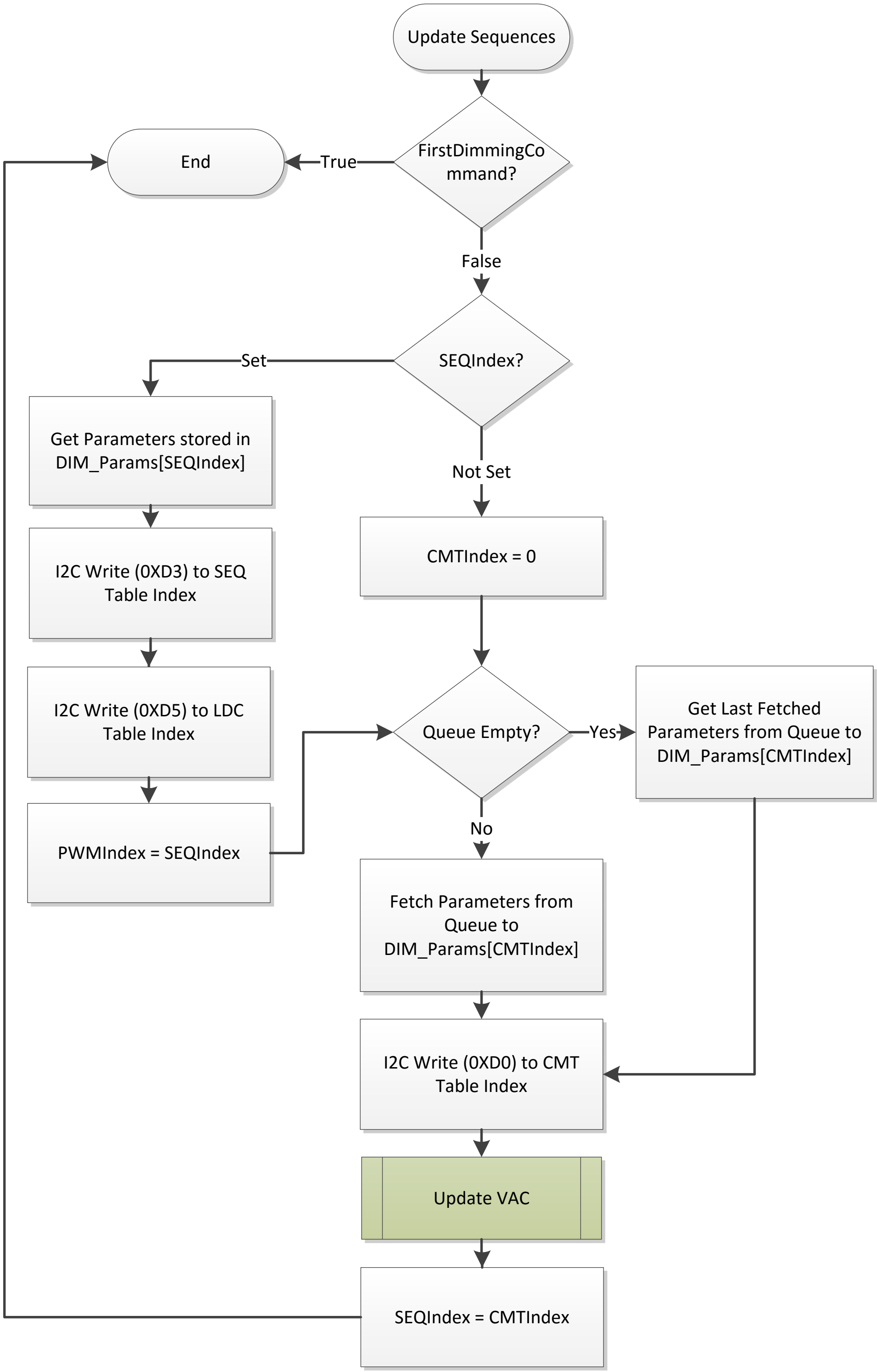

Update Sequences
FirstDimmingCommand?
True
End
False
SEQIndex?
Set
Get Parameters stored in DIM_Params[SEQIndex]
I2C Write (0XD3) to SEQ Table Index
I2C Write (0XD5) to LDC Table Index
PWMIndex = SEQIndex
Not Set
CMTIndex = 0
Queue Empty?
Yes
Get Last Fetched Parameters from Queue to DIM_Params[CMTIndex]
No
Fetch Parameters from Queue to DIM_Params[CMTIndex]
I2C Write (0XD0) to CMT Table Index
Update VAC
SEQIndex = CMTIndex

Figure 8-14. Sequence Update

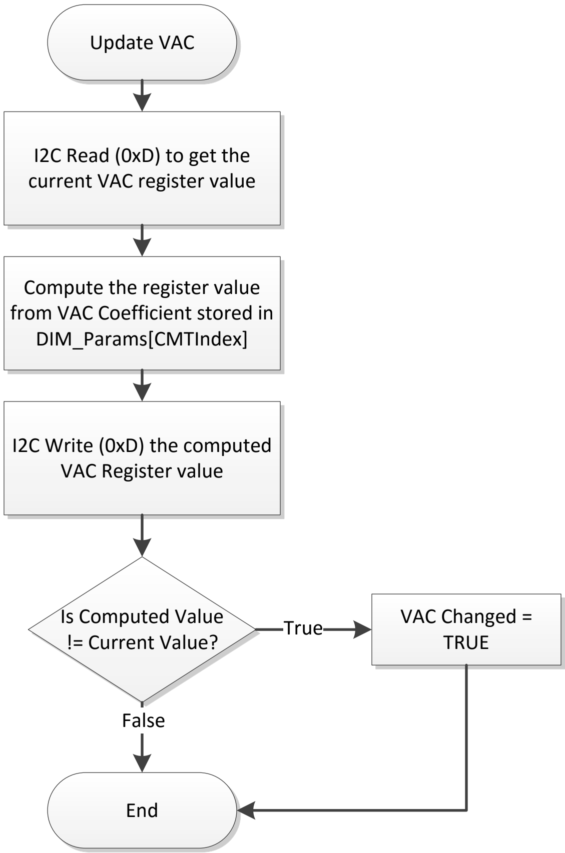

Update VAC
I2C Read (0xD) to get the current VAC register value
Compute the register value from VAC Coefficient stored in DIM_Params[CMTIndex]
I2C Write (0xD) the computed VAC Register value
Is Computed Value != Current Value?
True
VAC Changed = TRUE
False
End

Figure 8-15. VAC Update

8.12 FSYNC Interrupt

ASIC interrupts Piccolo on frame syncs to help Piccolo synchronize the dimming updates. The sequence command table index updates and PWM updates are all synchronized to this interrupt from the ASIC.

The control flow is shown in Figure 8-16. Please note that control flow reflect only the common dimming approach done when using external video input or internal test pattern generator. Special dimming processing is performed when a splash screen is displayed due to the irregular video processing of the splash image (Chapter 9).

The related C functions are:

- `interrupt void DLPC120_GeneralInterrupt(void)`
- `BOOL DLPC120_GetInterruptReceived(void)`
- `Uint16 DLPC120_ProcessInterrupt(void)`

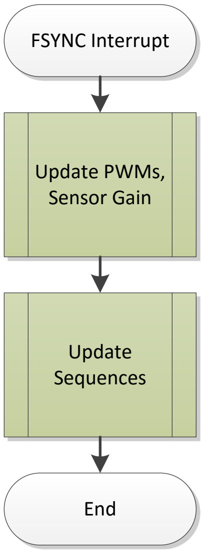


Figure 8-16. FSYNC Interrupt Processing

**This diagram shows only the basic framework on how dimming is tied to FSYNC interrupt from ASIC. For description on ASIC interrupts handled by Piccolo, please refer to Chapter 5.

The related C functions are:

- `interrupt void DLPC120_GeneralInterrupt(void)`
- `BOOL DLPC120_GetInterruptReceived(void)`
- `Uint16 DLPC120_ProcessInterrupt(void)`

8.13 PWM Update

The last step in completing the dimming process is setting the PWMs and sensor gains to match the sequence loaded by the ASIC. The PWM update process applies new values to the Piccolo PWM outputs which control LED light and/or current level settings, depending on operating mode. The process is initiated after receipt of an ASIC interrupt and determination of need of update from calling process.

The update PWM routine will first check if the frame is right to apply the PWMs. As seen in Figure 8-4, PWMs are applied only from the third frame onwards (frame count starts from frame after the first dimming command). To ensure this, the software uses the `FirstDimmingCommand` variable. If it is not the first dimming command, it checks if the `PWMIndex` is set to a valid value. This index variable is set during sequence update process. If the `PWMIndex` is valid, the respective parameters are fetched from the `DIM_Params[]` array. The `DIMParams[]` at `PWMIndex` will correspond to the sequence update that finished in the last frame, and for which the ASIC interrupted the Piccolo.

The last stage in PWM update is the actual application of the PWM and sensor gain values. Depending on the operating mode, there is a difference in how the current limit PWM is handled. In the *discontinuous* case, the current PWM is applied first so that the DC current established in the drive inductor, etc. begins to slew to the new value prior to application of the new LED amplitude settings. In cases where previous mode was *discontinuous* and next mode is *continuous*, the LED levels must be adjusted (lowered) prior to the application of the new current level to avoid potential momentary spiking of LED current to higher than necessary levels.

Following the application of PWMs, the sensor gains are adjusted by changing two GPIO lines of the Piccolo thereby completing the dimming process. These two GPIO lines can be found in the ***HUD LED Driver Schematics***. The control flow is shown in Figure 8-17.

The related C functions are:

- `void DIM_ApplyDimmingSettings(void)`
- `Uint16 DIM_SetSensorGain(DIM_SENSOR_GAIN GainLevel)`

- `Uint16 PWM_SetDutyCycle (PWM_NAMES Pwm, Uint16 Value)`

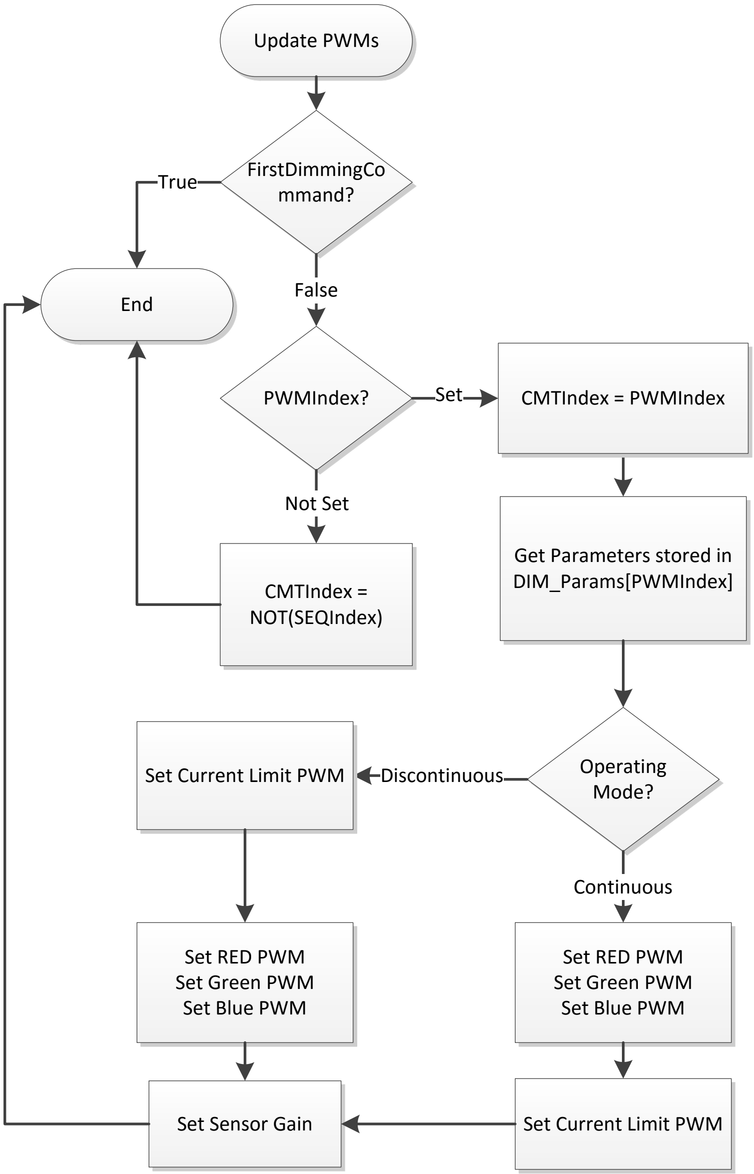


Figure 8-17. PWM Update Process

8.14 Temperature Compensation Modes:

Temperature compensation may be performed in 2 measurement modes, the custom mode and the Piccolo mode.

1. **Custom Mode:** In the custom mode, the ambient temperature, possibly measured using some external device is sent as an SPI command. This temperature is directly used for dimming without further processing. Low Pass Filter and Sequence Enforcement (explained in TMP411 mode) do not apply to this mode.
2. **Piccolo Mode:** In the Piccolo mode, the temperature is measured by the TMP411 temperature sensor monitor, which monitors the DMD temperature sensor. This temperature is periodically sampled by the Piccolo at a frequency of 8 Hz or less. The sampling frequency can also be set through an SPI command (only integers less than eight are permitted). Refer to the Temperature Compensation section in the **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)** for more information about the SPI commands. The Low Pass Filter is explained in the next section.

Sequence Enforcement: The Piccolo Software validates the temperature readings of the TMP411 after sampling. If 4 or more consecutive invalid readings are obtained**,** the software assumes that TMP411 sensor is malfunctioning and in order to protect the DMD, sets the active temperature to the highest transition state available so as to enforce the lowest sequence. A “50-50 Sequence Enforced” status is flagged in such a situation. Once a valid temperature is obtained again, the current active temperature is used and any further TMP411 readings are filtered and then applied. **NOTE:** for the DLP3030-Q1 DMD, 50-50 Sequence Enforce is not required.

The related C function is:

- `Uint16 TMPR_ReadDMDTmprK10(Uint16 *TmprK10)`
- `static Uint16 TMPR_GetTargetTmprK10()`

8.15 Low Pass Temperature Filter

The low pass filter is used to smooth sharp changes in temperature. This is not only to prevent rapid variations in the brightness but also to avoid undesirably quick changes in sequence that may occur when the temperature is oscillating about a transition point. The damping effect of the low pass filter is a function of its strength (g). In the Piccolo software the low pass filter is implemented by measuring the difference in consecutive temperature readings and using a damped value for temperature for dimming. The low pass filter approach comprises of 2 steps:

1. **Filtering and determination of output temperature of filter:**

   Let T1(n) be the last output temperature of the filter for dimming and T2(n + 1) the new reading from the TMP411.

   $\text{New output temperature T1(n + 1)} = \text{g} * \text{T2(n + 1)} - (1 - \text{g}) * \text{T1(n)}$

2. **Quantization:**

   Before using it for dimming, the output temperature can be quantized to a desired level as indicated by the ‘Quantization Step’ (Qstep) property of the Filter.

   If T1(n + 1) > Applied Low Pass Filter Output (Quantized) (round down)

   $Active\ Temperature\ (Used\ for\ dimming) = T1(n+1) - modulo(T1(n+1), Qstep\ )$

   Else (round up)

   $Active\ Temperature\ (Used\ for\ dimming) = T1(n+1) - modulo(T1(n+1), Qstep\ ) + Qstep$

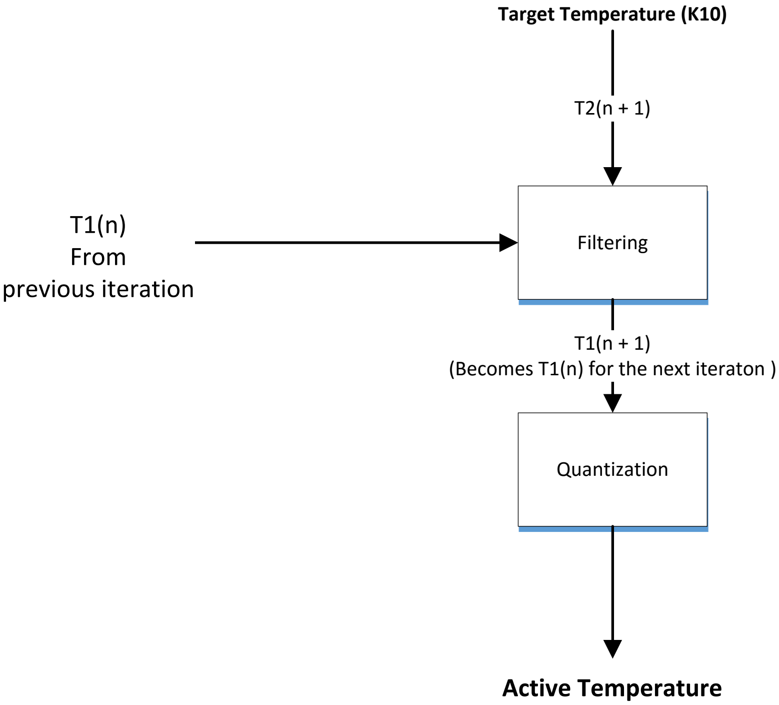


Figure 8-18. Low Pass Temperature Filter

Note that the Filter Strength and Qstep can be set and read through SPI commands. The software actually performs all calculations in K10 format. Any necessary conversions (for example, multiplying the Qstep by 10 to operate on K10 temperatures) on the factors are performed internally by the software and the user can enter all data with respect to temperatures in degrees Celsius.

The related C function is:

- `static Uint16 TMPR_LowPassFilter(Uint16 TmprK10)`

Splash Screen Control

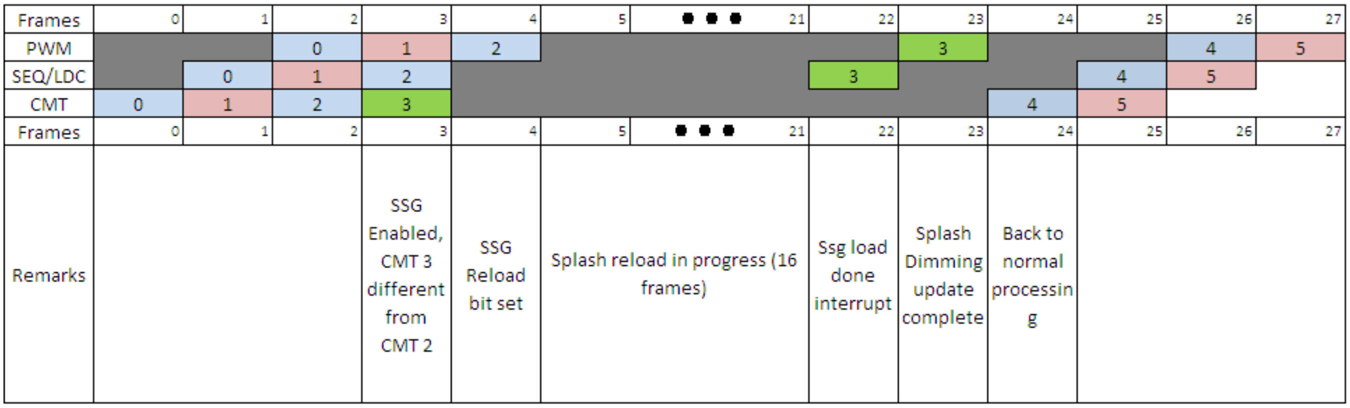
9.1 Splash Screen Control

The HUD system is capable of displaying preloaded splash images having resolutions 864x480 and 608x684. Due to the way splash is implemented in ASIC, changes in CMT, Bezel and VAC will not affect splash after it is loaded. We need to reload the splash whenever there is a change in CMT Table, the Bezel parameters or the VAC coefficient. Additionally, since the splash image is loaded from serial flash interface, it takes much longer to update than a standard video input. Therefore, this necessitates a special dimming mode when splash is loaded. A special SPI command is provided to force Piccolo into this Splash Screen Dimming mode. See **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)** for details on the SPI command.

The ASIC supports reloading of splash under the control of Piccolo. When a dimming command is received in splash dimming mode, the software performs the dimming parameter computation in the same manner as in normal dimming mode. The processing continues in normal dimming mode if the values for parameters (CMT, VAC and Bezel Offset) are same as current values. If they are different splash is reloaded by the Piccolo SW. Once the parameters computation is over, the software checks if the CMT Command Table Index already loaded to the ASIC is different from the newly computed CMT Table Index. If they are same, the processing continues as in normal dimming mode. The splash also reloads when there is a change in Bezel Offset. Please note that the reloading on change in CMT Index / Bezel Offset / VAC does occurs only if the splash is enabled through the Piccolo and not when it is enabled by direct register writes or the Pass/Fail command lists of the External Video Detect BIST (see Section 15.4)

However, if the CMT to be loaded is different than currently loaded CMT, we need to reload splash in addition to setting the CMT table index in the ASIC. For this, the Piccolo software sets the SSG Reload bit in ASIC register 0x98 (Refer to **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)** for details on registers), forcing ASIC to start the splash reload. The reloading of splash can take close to 16 frames. During these frames where ASIC is reloading splash, the pipelining of dimming updates are disabled in Piccolo. In other words, the software will not write to any sequence command index registers in the ASIC and will not edit the LED PWMs.

Once the splash load is completed, ASIC interrupts Piccolo through the Splash Load Done interrupt. On receiving this interrupt, the software finishes the current dimming update by setting the SEQ, LDC table indexes and LED PWMs in the next two frames. Following this the normal dimming processing is restarted. This process is indicated in Figure 9-1.



Frames	0	1	2	3	4	5	● ● ● 21	22	23	24	25	26	27
PWM	▨	▨	0	1	2	▨	▨	▨	3	▨	▨	4	5
SEQ/LDC	▨	0	1	2	▨	▨	▨	3	▨	▨	4	5	
CMT	0	1	2	3	▨	▨	▨	▨	▨	4	5		
Frames	0	1	2	3	4	5	● ● ● 21	22	23	24	25	26	27
Remarks				SSG Enabled, CMT 3 different from CMT 2	SSG Reload bit set	Splash reload in progress (16 frames)		Ssg load done interrupt	Splash Dimming update complete	Back to normal processing			

Figure 9-1. Splash Dimming Timeline

The software implementation of this splash dimming mode utilizes a small state machine as shown in Figure 9-2. The red colored labels near to the connectors indicate the ASIC interrupt after which the state transition is completed. The green colored labels indicate the actions performed by the software when in the respective state.

In this state machine, the default state is *CMTTest*. The software remains in this state as long as the CMT, index VAC and the Bezel Adjustment offset computed for the new dimming command remains the same. When the computed CMT index or VAC or the Bezel Adjustment offset value is different, it applies the new CMT parameters, Bezel Adjustment values and then switches to the *Refresh* state. The state change occurs only on next FSYNC interrupt. Once in this state the software sends I2C command to ASIC to initiate the splash reload. The state is then changed to *LDC-SEQ*, which gets activated when ASIC interrupt Splash Load Done is received by Piccolo. All FSYNC interrupts between these two states are ignored by the software. Once the *LDC-SEQ* mode is activated, the software updates the LDC and SEQ command table indexes computed for the dimming command. The state is then changed to *PWM* which gets activated after next FSYNC interrupt. The LED PWMs are applied in this state completing the splash dimming. After this the state is set to *CMT Test* which gets activated after next FSYNC.

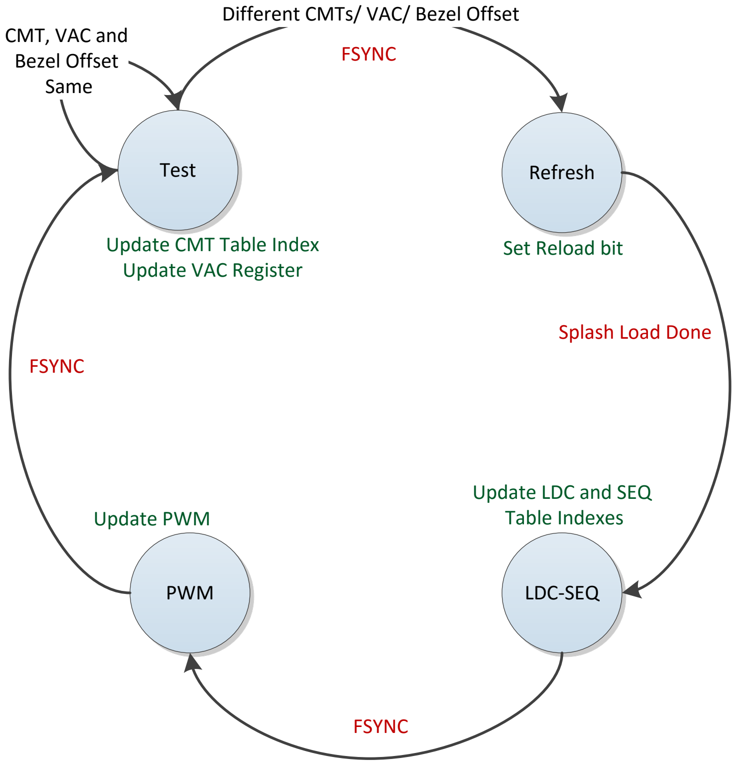


Figure 9-2. Splash Dimming State Machine

The software uses two ASIC interrupts – *FSYNC* and *Splash Load Done* to achieve the dimming in splash dimming mode. The software control flow during FSYNC interrupt processing is shown in Figure 9-3. In this figure, the *VAC Changed* field used is set during VAC Update (Figure 8-15). The control flow during Splash Load Done interrupt is shown in Figure 9-4.

The related C functions are:

- `Uint16 DIM_ProcessASICInterrupt(BOOL Fsync, BOOL SSGLoadDone)`
- `Uint16 DIM_SetSSGMode(BOOL Enable)`
- `BOOL DIM_GetSSGMode(void)`

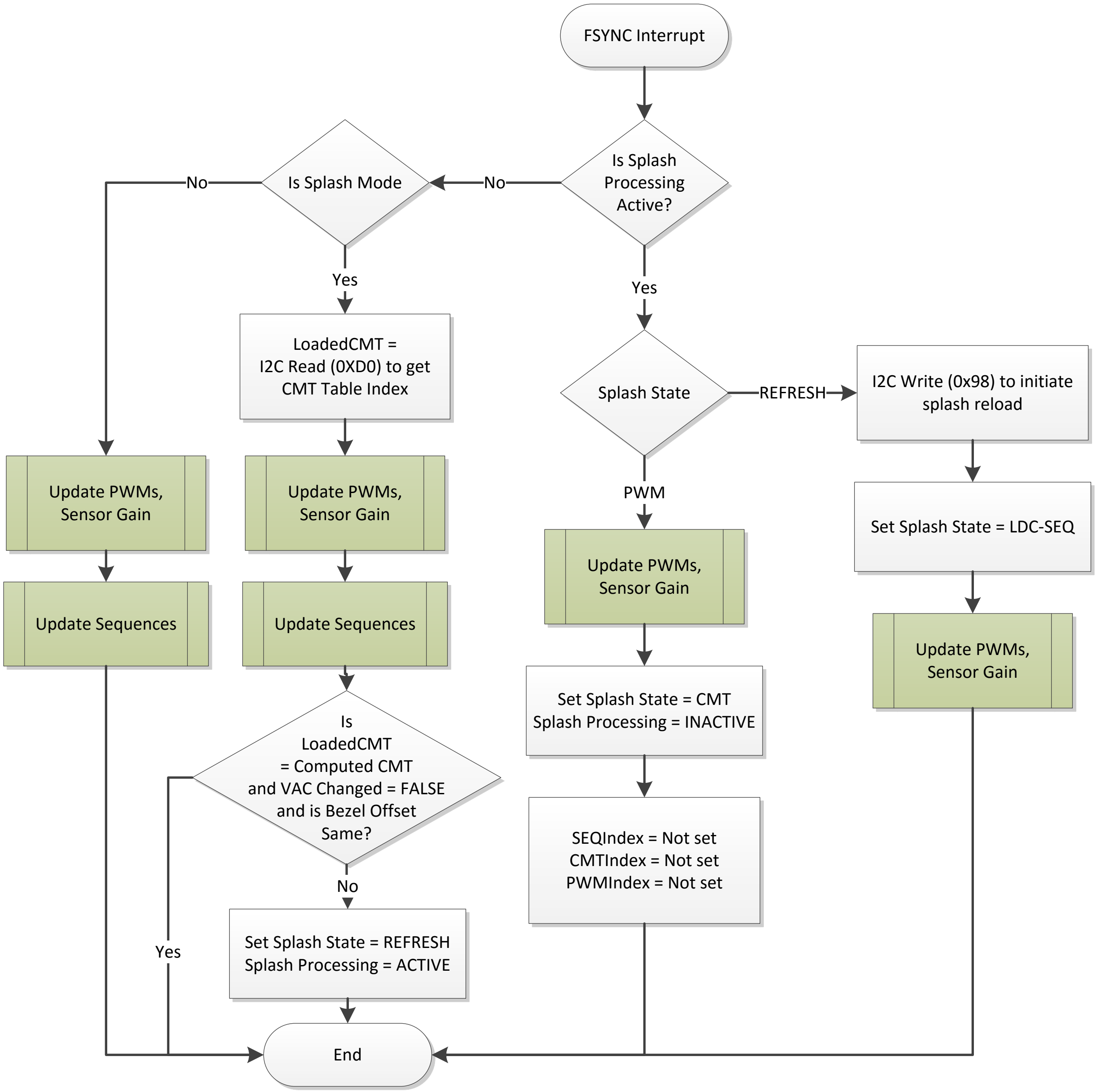

FSYNC Interrupt
Is Splash Processing Active?
No
Is Splash Mode
No
Yes
Yes
LoadedCMT = I2C Read (0XD0) to get CMT Table Index
Splash State
REFRESH
I2C Write (0x98) to initiate splash reload
Update PWMs, Sensor Gain
Update PWMs, Sensor Gain
PWM
Set Splash State = LDC-SEQ
Update PWMs, Sensor Gain
Update Sequences
Update Sequences
Update PWMs, Sensor Gain
Set Splash State = CMT
Splash Processing = INACTIVE
Is LoadedCMT = Computed CMT and VAC Changed = FALSE and is Bezel Offset Same?
SEQIndex = Not set
CMTIndex = Not set
PWMIndex = Not set
No
Yes
Set Splash State = REFRESH
Splash Processing = ACTIVE
End

Figure 9-3. FSYNC Interrupt Handling in Splash Dimming Mode

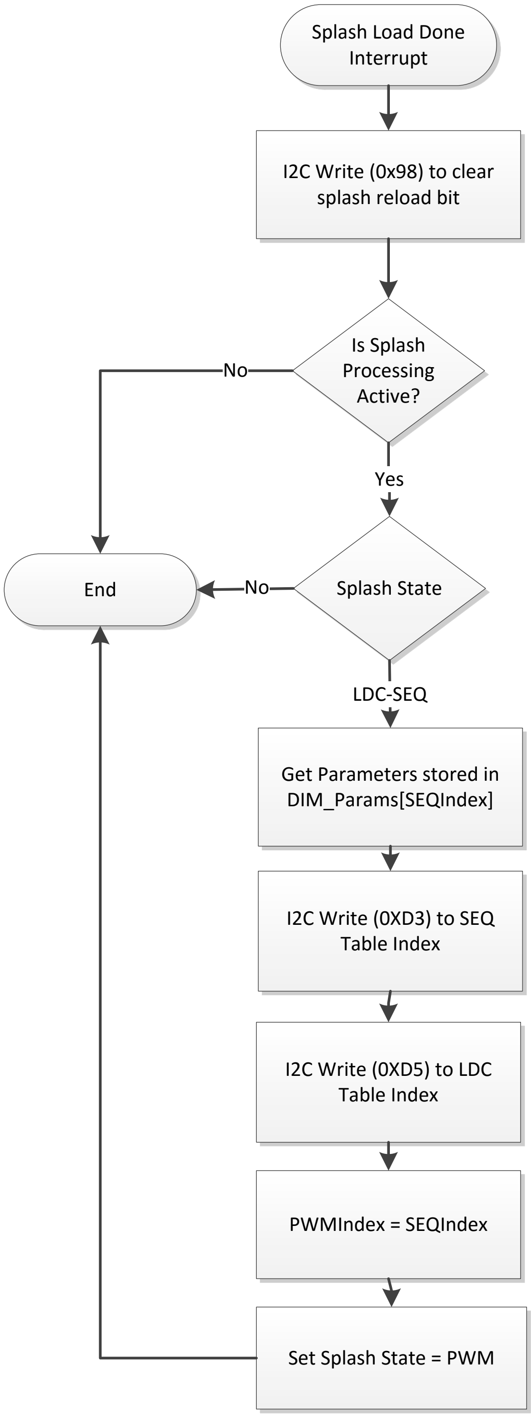

Splash Load Done Interrupt
I2C Write (0x98) to clear splash reload bit
Is Splash Processing Active?
No
Yes
End
No
Splash State
LDC-SEQ
Get Parameters stored in DIM_Params[SEQIndex]
I2C Write (0XD3) to SEQ Table Index
I2C Write (0XD5) to LDC Table Index
PWMIndex = SEQIndex
Set Splash State = PWM

Figure 9-4. Splash Load Done Interrupt Handling

Chapter 10

DLPU061–March 2018

Calibration Data and Configuration File

10.1 Calibration Data and Configuration File

The internal flash memory of Piccolo is divided into four sectors (A, B, C and D) each of 16 kilobytes. The sector D is reserved for storing calibration and command list information. The data stored in this sector comprises of two files – the calibration file and the configuration file.

Calibration Data contains characteristics of LEDs measured at various temperatures. These characteristics help the software compute the dimming parameters necessary for achieving proper brightness control. The configuration file stores data relating to Transition temperatures, Dimming LUTs and command lists (like splash, test patterns etc).

The calibration file is initially generated by a TI Apps engineer. This basic calibration file is then used to perform automatic calibration with the help of a PC based tool – Automotive Control Program. The auto-calibration algorithm performs several measurements to find the various parameters that accurately describe the LED characteristics over various dimming and temperature ranges. Once the automatic calibration finishes, a new calibration file is obtained.

The configuration file contains a map of locations of various command lists in the ASIC flash. The ASIC App Flash File is also embedded into this file. Before flashing the configuration data into the Piccolo, it is essential to ensure the corresponding ASIC flash file resides in the SPI Flash that is accessed by the ASIC. If not, there is a good chance that all command lists might dangerously malfunction or completely fail, including critical ones that perform ASIC Initialization and BIST’s. The calibration file must also be compatible with the configuration file especially in case of the temperature data. Any incompatibility between the two files can be found by opening both files in the Automotive Control Program and verifying.

The calibration and the configuration files are sent to Piccolo over SPI by the Automotive Control Program Software. The Piccolo software then writes the data to the internal flash memory. The structure of the two files is explained better in the Section 10.2.

The calibration data includes:

- Calibration Tables for different temperatures
- VAC Mode Switch Points at different temperatures
- Power-On Default Settings
- Sensor Gain mapping data

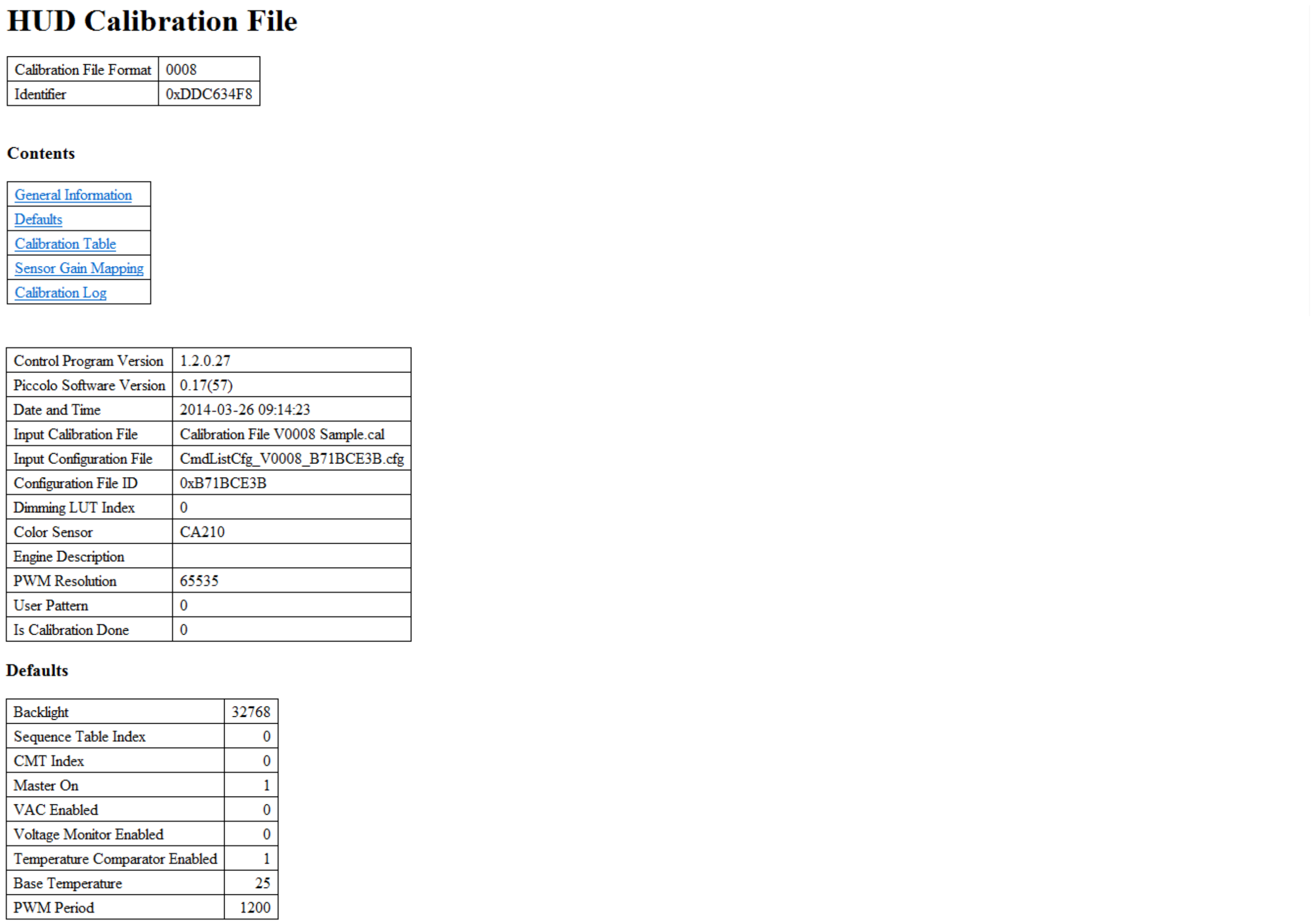
The configuration data includes:

- Transition Temperature Information (not needed for DLP3030-Q1)
- Locations and details of all Command Lists present in the corresponding ASIC Flash File, broadly classified as:
  - Splash
  - Test Patterns
  - External Video
  - Generic Command Lists
- Extra Information that can be stored in terms of 32 bit keys and corresponding 32 bit values
- Command List Information for Dimming Look Up Tables
  - LDC-SEQ Map
  - SEQ-CMT Map

- – CMT-Gamma Map

The calibration data also includes other information used to aid in automatic calibration of LEDs by the PC-based Automotive Control Program Software, such as information to self-verify its contents to ensure that valid calibration data is present in the flash.

Figure 10-1 and Figure 2-23 show a sample calibration and configuration file.



HUD Calibration File

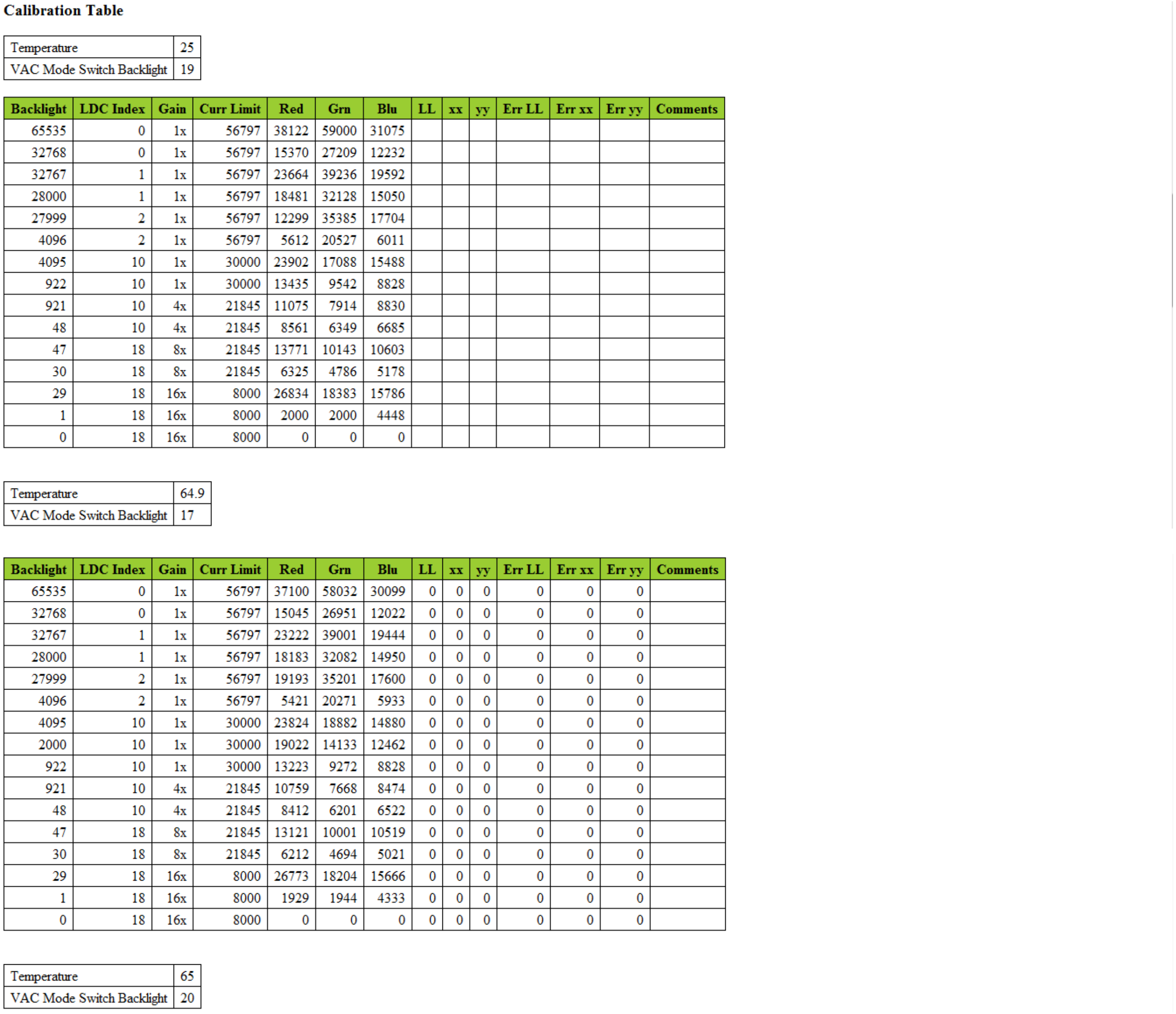
Calibration File Format	0008
Identifier	0xDDC634F8

Contents

Control Program Version	1.2.0.27
Piccolo Software Version	0.17(57)
Date and Time	2014-03-26 09:14:23
Input Calibration File	Calibration File V0008 Sample.cal
Input Configuration File	CmdListCfg_V0008_B71BCE3B.cfg
Configuration File ID	0xB71BCE3B
Dimming LUT Index	0
Color Sensor	CA210
Engine Description	
PWM Resolution	65535
User Pattern	0
Is Calibration Done	0

Defaults

Backlight	32768
Sequence Table Index	0
CMT Index	0
Master On	1
VAC Enabled	0
Voltage Monitor Enabled	0
Temperature Comparator Enabled	1
Base Temperature	25
PWM Period	1200



Calibration Table

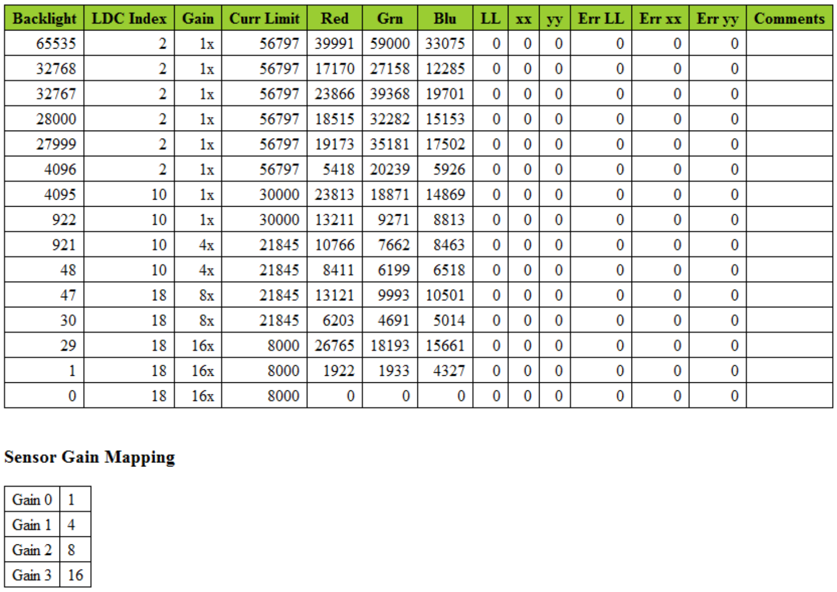
Temperature	25
VAC Mode Switch Backlight	19

Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	0	1x	56797	38122	59000	31075							
32768	0	1x	56797	15370	27209	12232							
32767	1	1x	56797	23664	39236	19592							
28000	1	1x	56797	18481	32128	15050							
27999	2	1x	56797	12299	35385	17704							
4096	2	1x	56797	5612	20527	6011							
4095	10	1x	30000	23902	17088	15488							
922	10	1x	30000	13435	9542	8828							
921	10	4x	21845	11075	7914	8830							
48	10	4x	21845	8561	6349	6685							
47	18	8x	21845	13771	10143	10603							
30	18	8x	21845	6325	4786	5178							
29	18	16x	8000	26834	18383	15786							
1	18	16x	8000	2000	2000	4448							
0	18	16x	8000	0	0	0							

Temperature	64.9
VAC Mode Switch Backlight	17

Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	0	1x	56797	37100	58032	30099	0	0	0	0	0	0	
32768	0	1x	56797	15045	26951	12022	0	0	0	0	0	0	
32767	1	1x	56797	23222	39001	19444	0	0	0	0	0	0	
28000	1	1x	56797	18183	32082	14950	0	0	0	0	0	0	
27999	2	1x	56797	19193	35201	17600	0	0	0	0	0	0	
4096	2	1x	56797	5421	20271	5933	0	0	0	0	0	0	
4095	10	1x	30000	23824	18882	14880	0	0	0	0	0	0	
2000	10	1x	30000	19022	14133	12462	0	0	0	0	0	0	
922	10	1x	30000	13223	9272	8828	0	0	0	0	0	0	
921	10	4x	21845	10759	7668	8474	0	0	0	0	0	0	
48	10	4x	21845	8412	6201	6522	0	0	0	0	0	0	
47	18	8x	21845	13121	10001	10519	0	0	0	0	0	0	
30	18	8x	21845	6212	4694	5021	0	0	0	0	0	0	
29	18	16x	8000	26773	18204	15666	0	0	0	0	0	0	
1	18	16x	8000	1929	1944	4333	0	0	0	0	0	0	
0	18	16x	8000	0	0	0	0	0	0	0	0	0	

Temperature	65
VAC Mode Switch Backlight	20

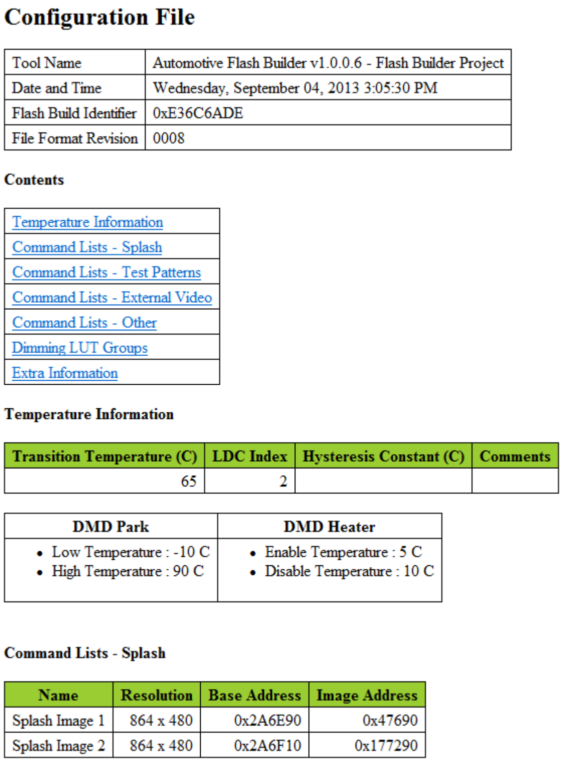


Backlight	LDC Index	Gain	Curr Limit	Red	Grn	Blu	LL	xx	yy	Err LL	Err xx	Err yy	Comments
65535	2	1x	56797	39991	59000	33075	0	0	0	0	0	0	
32768	2	1x	56797	17170	27158	12285	0	0	0	0	0	0	
32767	2	1x	56797	23866	39368	19701	0	0	0	0	0	0	
28000	2	1x	56797	18515	32282	15153	0	0	0	0	0	0	
27999	2	1x	56797	19173	35181	17502	0	0	0	0	0	0	
4096	2	1x	56797	5418	20239	5926	0	0	0	0	0	0	
4095	10	1x	30000	23813	18871	14869	0	0	0	0	0	0	
922	10	1x	30000	13211	9271	8813	0	0	0	0	0	0	
921	10	4x	21845	10766	7662	8463	0	0	0	0	0	0	
48	10	4x	21845	8411	6199	6518	0	0	0	0	0	0	
47	18	8x	21845	13121	9993	10501	0	0	0	0	0	0	
30	18	8x	21845	6203	4691	5014	0	0	0	0	0	0	
29	18	16x	8000	26765	18193	15661	0	0	0	0	0	0	
1	18	16x	8000	1922	1933	4327	0	0	0	0	0	0	
0	18	16x	8000	0	0	0	0	0	0	0	0	0	

Sensor Gain Mapping

Gain 0	1
Gain 1	4
Gain 2	8
Gain 3	16

Figure 10-1. Sample Calibration File



Configuration File

Tool Name	Automotive Flash Builder v1.0.0.6 - Flash Builder Project
Date and Time	Wednesday, September 04, 2013 3:05:30 PM
Flash Build Identifier	0xE36C6ADE
File Format Revision	0008

Contents

Temperature Information
Command Lists - Splash
Command Lists - Test Patterns
Command Lists - External Video
Command Lists - Other
Dimming LUT Groups
Extra Information

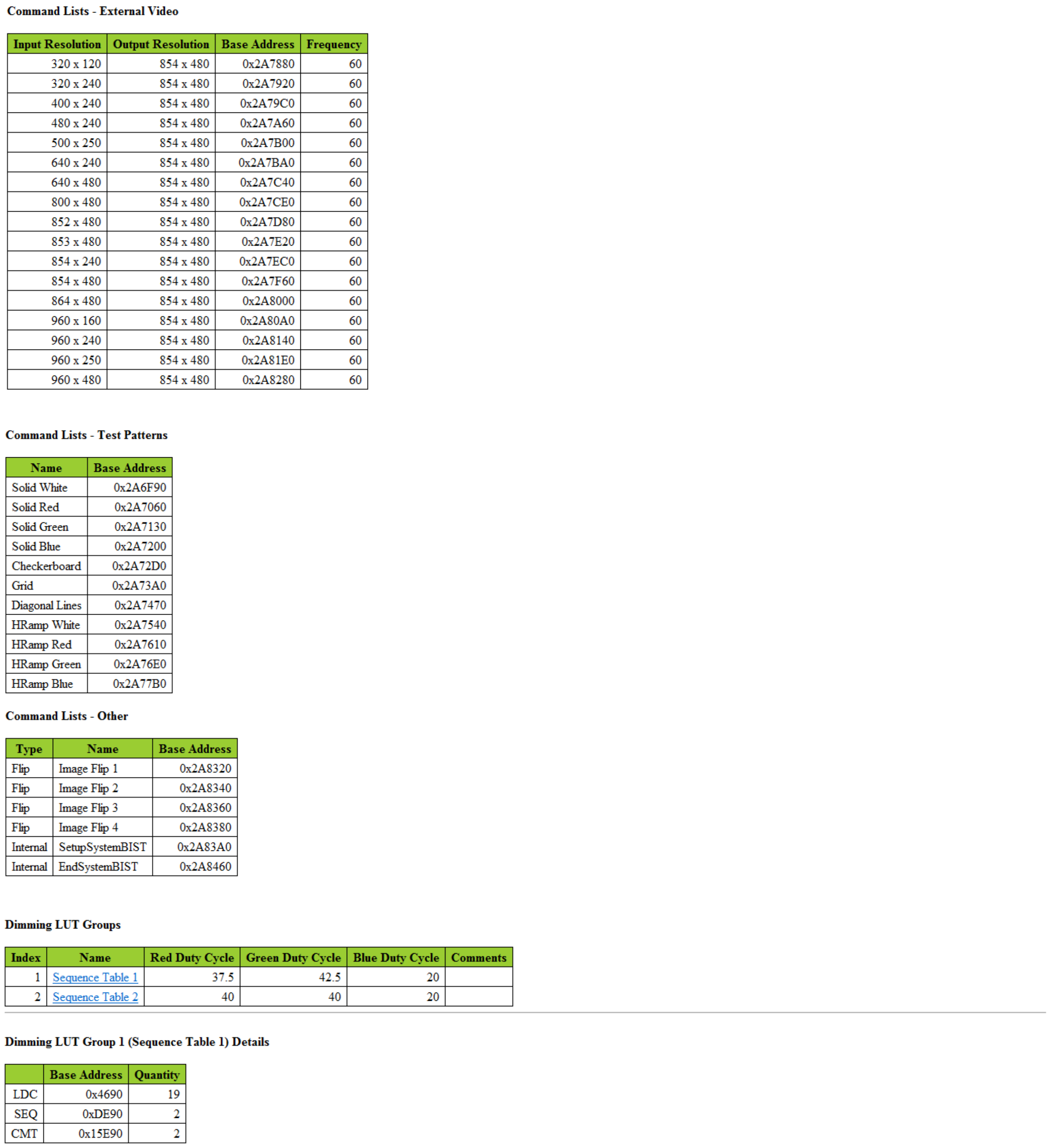
Temperature Information

Transition Temperature (C)	LDC Index	Hysteresis Constant (C)	Comments
65	2		

DMD Park	DMD Heater
• Low Temperature : -10 C • High Temperature : 90 C	• Enable Temperature : 5 C • Disable Temperature : 10 C

Command Lists - Splash

Name	Resolution	Base Address	Image Address
Splash Image 1	864 x 480	0x2A6E90	0x47690
Splash Image 2	864 x 480	0x2A6F10	0x177290



Command Lists - External Video

Input Resolution	Output Resolution	Base Address	Frequency
320 x 120	854 x 480	0x2A7880	60
320 x 240	854 x 480	0x2A7920	60
400 x 240	854 x 480	0x2A79C0	60
480 x 240	854 x 480	0x2A7A60	60
500 x 250	854 x 480	0x2A7B00	60
640 x 240	854 x 480	0x2A7BA0	60
640 x 480	854 x 480	0x2A7C40	60
800 x 480	854 x 480	0x2A7CE0	60
852 x 480	854 x 480	0x2A7D80	60
853 x 480	854 x 480	0x2A7E20	60
854 x 240	854 x 480	0x2A7EC0	60
854 x 480	854 x 480	0x2A7F60	60
864 x 480	854 x 480	0x2A8000	60
960 x 160	854 x 480	0x2A80A0	60
960 x 240	854 x 480	0x2A8140	60
960 x 250	854 x 480	0x2A81E0	60
960 x 480	854 x 480	0x2A8280	60

Command Lists - Test Patterns

Name	Base Address
Solid White	0x2A6F90
Solid Red	0x2A7060
Solid Green	0x2A7130
Solid Blue	0x2A7200
Checkerboard	0x2A72D0
Grid	0x2A73A0
Diagonal Lines	0x2A7470
HRamp White	0x2A7540
HRamp Red	0x2A7610
HRamp Green	0x2A76E0
HRamp Blue	0x2A77B0

Command Lists - Other

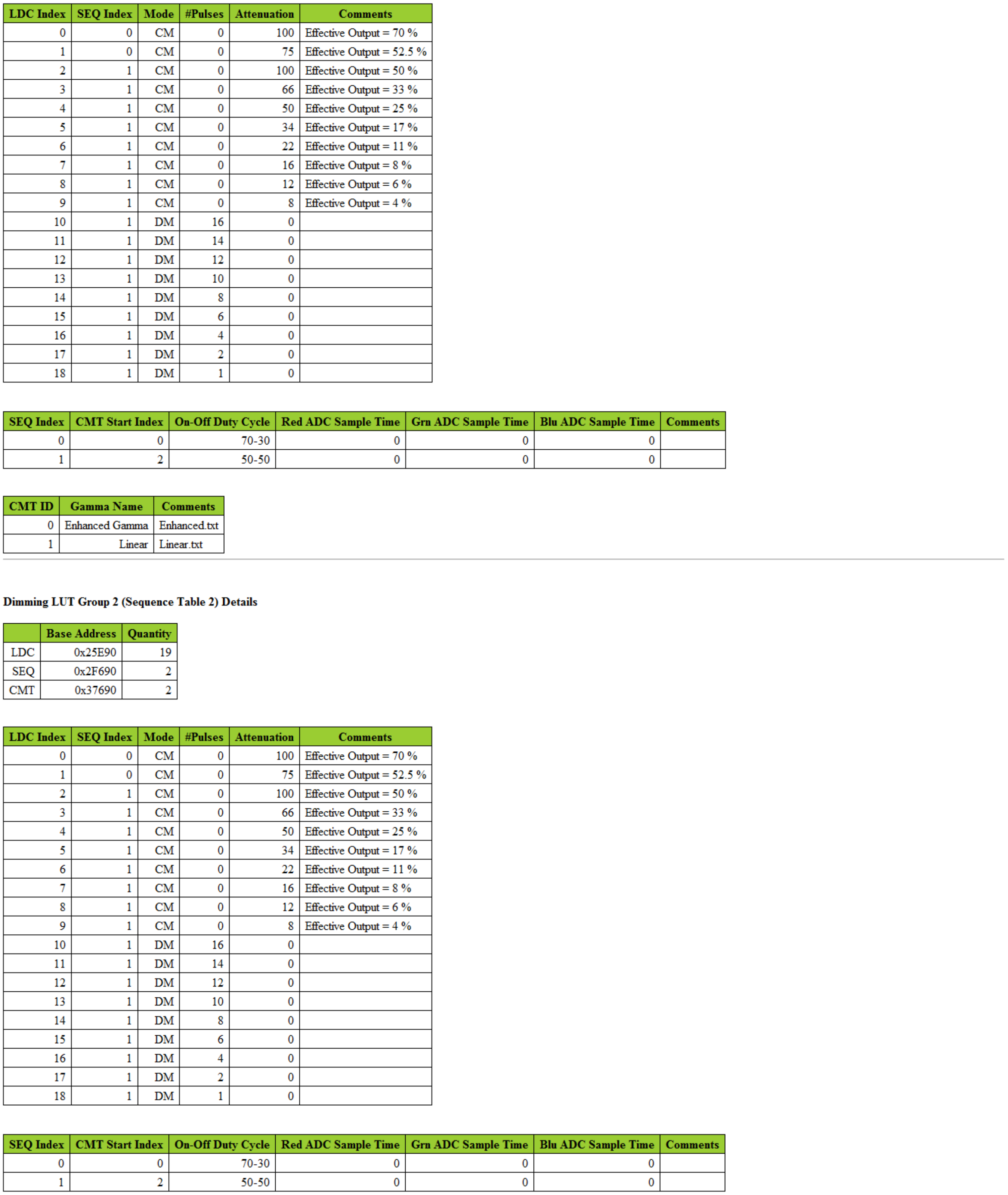
Type	Name	Base Address
Flip	Image Flip 1	0x2A8320
Flip	Image Flip 2	0x2A8340
Flip	Image Flip 3	0x2A8360
Flip	Image Flip 4	0x2A8380
Internal	SetupSystemBIST	0x2A83A0
Internal	EndSystemBIST	0x2A8460

Dimming LUT Groups

Index	Name	Red Duty Cycle	Green Duty Cycle	Blue Duty Cycle	Comments
1	Sequence Table 1	37.5	42.5	20	
2	Sequence Table 2	40	40	20	

Dimming LUT Group 1 (Sequence Table 1) Details

	Base Address	Quantity
LDC	0x4690	19
SEQ	0xDE90	2
CMT	0x15E90	2



LDC Index	SEQ Index	Mode	#Pulses	Attenuation	Comments
0	0	CM	0	100	Effective Output = 70 %
1	0	CM	0	75	Effective Output = 52.5 %
2	1	CM	0	100	Effective Output = 50 %
3	1	CM	0	66	Effective Output = 33 %
4	1	CM	0	50	Effective Output = 25 %
5	1	CM	0	34	Effective Output = 17 %
6	1	CM	0	22	Effective Output = 11 %
7	1	CM	0	16	Effective Output = 8 %
8	1	CM	0	12	Effective Output = 6 %
9	1	CM	0	8	Effective Output = 4 %
10	1	DM	16	0	
11	1	DM	14	0	
12	1	DM	12	0	
13	1	DM	10	0	
14	1	DM	8	0	
15	1	DM	6	0	
16	1	DM	4	0	
17	1	DM	2	0	
18	1	DM	1	0	

SEQ Index	CMT Start Index	On-Off Duty Cycle	Red ADC Sample Time	Grn ADC Sample Time	Blu ADC Sample Time	Comments
0	0	70-30	0	0	0	
1	2	50-50	0	0	0	

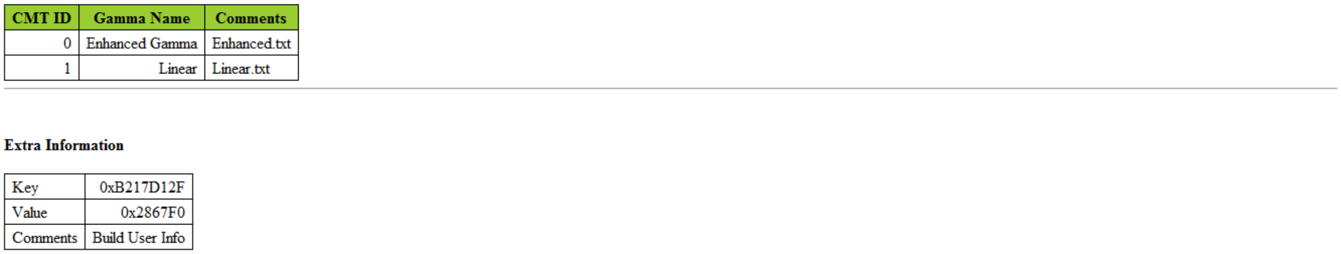
CMT ID	Gamma Name	Comments
0	Enhanced Gamma	Enhanced.txt
1	Linear	Linear.txt

Dimming LUT Group 2 (Sequence Table 2) Details

	Base Address	Quantity
LDC	0x25E90	19
SEQ	0x2F690	2
CMT	0x37690	2

LDC Index	SEQ Index	Mode	#Pulses	Attenuation	Comments
0	0	CM	0	100	Effective Output = 70 %
1	0	CM	0	75	Effective Output = 52.5 %
2	1	CM	0	100	Effective Output = 50 %
3	1	CM	0	66	Effective Output = 33 %
4	1	CM	0	50	Effective Output = 25 %
5	1	CM	0	34	Effective Output = 17 %
6	1	CM	0	22	Effective Output = 11 %
7	1	CM	0	16	Effective Output = 8 %
8	1	CM	0	12	Effective Output = 6 %
9	1	CM	0	8	Effective Output = 4 %
10	1	DM	16	0	
11	1	DM	14	0	
12	1	DM	12	0	
13	1	DM	10	0	
14	1	DM	8	0	
15	1	DM	6	0	
16	1	DM	4	0	
17	1	DM	2	0	
18	1	DM	1	0	

SEQ Index	CMT Start Index	On-Off Duty Cycle	Red ADC Sample Time	Grn ADC Sample Time	Blu ADC Sample Time	Comments
0	0	70-30	0	0	0	
1	2	50-50	0	0	0	



CMT ID	Gamma Name	Comments
0	Enhanced Gamma	Enhanced.txt
1	Linear	Linear.txt

Extra Information

Key	0xB217D12F
Value	0x2867F0
Comments	Build User Info

Figure 10-2. Sample Configuration File

NOTE: Transition temperatures are not required for DLP3030-Q1 operation. Only one transition region is required for full performance over the entire operating temperature range.

10.2 Calibration and Configuration Data Structure

The calibration data starts with a *File Header* residing at the beginning of sector D. The *File Header* is fixed size and takes 16 bytes of memory. Following the *File Header* the various data is organized in sections with each section containing a fixed 8 byte *Section Header* followed by the variable *Section Data*. This arrangement is shown in Figure 10-3.

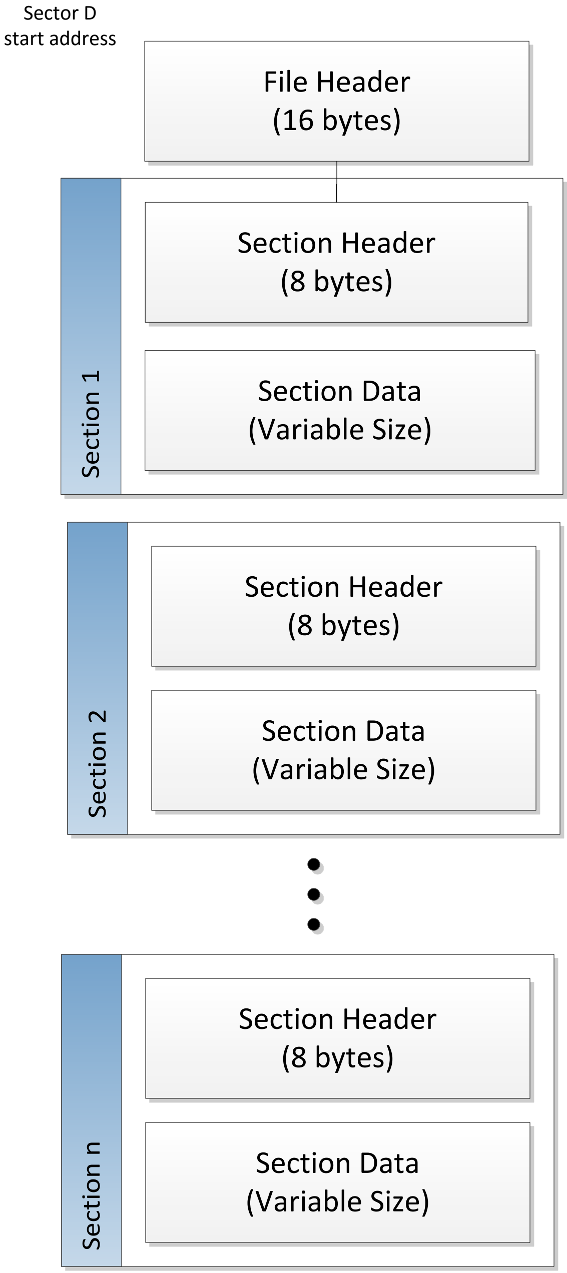


Figure 10-3. Calibration Data High level Memory Arrangement

10.2.1 File Header

The calibration *File Header* is structured as shown in Figure 10-4. The *File Header* helps the software verify if the data present in flash memory is valid. It also helps the software differentiate between various calibration file format revisions. Different versions of Piccolo software may support different revisions of calibration file. Figure 10-5 shows a sample file header section, which is also the section data corresponding to the example calibration file shown in Figure 10-1.

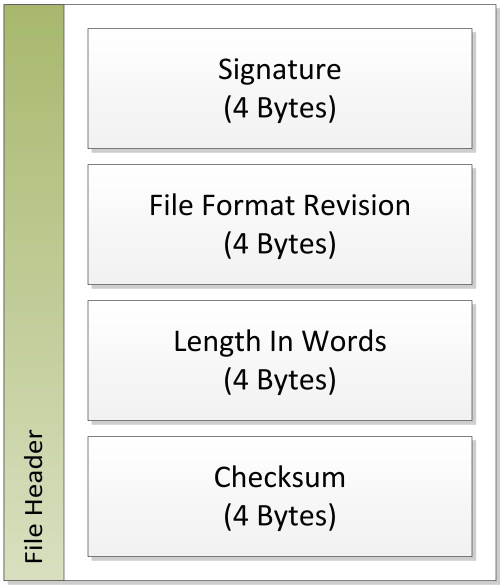


Figure 10-4. Calibration Data - Section File Header

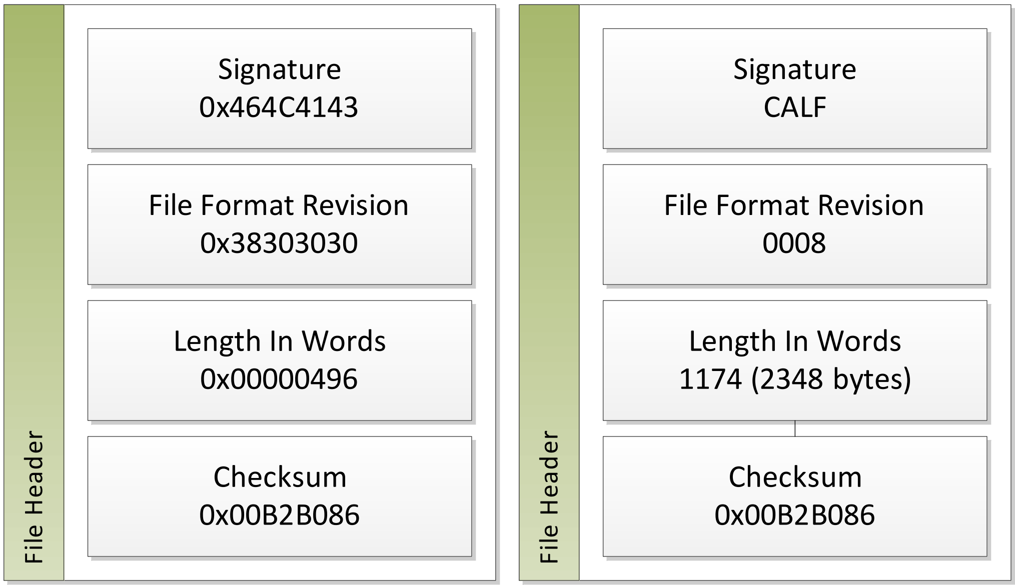


Figure 10-5. File Header: Sample Data with explanation

The fields are explained below:

- *Signature* – The Signature field is used to verify if the loaded file is indeed a calibration file. A valid calibration file will have the *Signature* value of ‘**0x464C4143’** which stands for letters ‘CALF’ in little endian format. Refer to Section A.2 for more details on this format.
- *File Format Revision* – This field provides the calibration file format revision information. Four characters are reserved for this purpose, with each revision changing these values. The present revision of calibration file is ‘0008’ stored in the flash as ‘**0x38303030’**. The following table lists the changes made to calibration file over various revisions.

File Format Revision	Description
0001	Supported only Calibration Table
0002	Added support for Dimming Table, XFactors, Calibration Tables for various Temperatures
0003	Added option to select Defaults while starting the HUD system
0004	Added support for command list information. Includes LDC-SEQ table, SEQ-CMT Table, CMT-Gamma Table and RGB Duty Cycle Information.
0005	Updated to support HRPWM in calibration tables.
0006	Added support for Temperature Transition Tables, DM Max Lumens, Mode Switch Brightness etc
0007	Splitting of the old calibration file: LED Brightness related data remains in the calibration file. All ASIC Flash File related data are now moved into the Configuration file. Support for Dimming LUT group.
0008**	• Combining of the LDC, Sensor Gain data and the PWM Values into a single table called the calibration table (in earlier versions LDC and Gain were part of a separate dimming table, which contained XFactors) • Using a different calibration table for each calibrated temperature instead of storing temperature coefficients in different tables

** The remaining descriptions in this file may be applicable only to calibration file format 0008.

Figure 10-6. Calibration File Format Revision History

- *Length In Words* – This field (4 bytes) stores the calibration data length in words after the *Checksum* field in the *File Header.*
- *Checksum* – This field stores the checksum of all words (4 bytes) after this field till the end of calibration data denoted by *Length In Words* field.

The File Header field of the calibration data is accessed in software through a pointer to `CAL_FILEHDR` structure as defined in calibration.h file.

```
typedef struct FileHeader
{
    Uint32 Signature;
    Uint32 FileVersion;
    Uint32 LengthInWords;
    Uint32 Checksum;
} CAL_FILEHDR;
```

10.2.2 Section Header

After the *File Header* the *calibration and configuration data sections* are sent one by one*.* Each section has a *Section Header* and variable amount of data that the section holds. Every *Section Header* has two fields as shown in Figure 10-7.

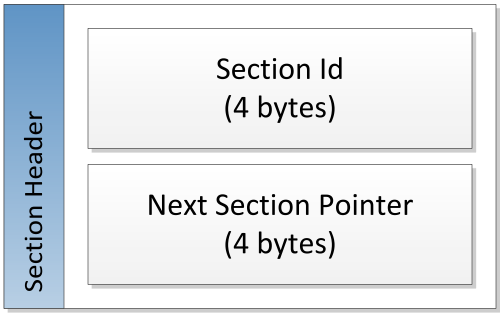


Figure 10-7. Calibration Data - Section Header

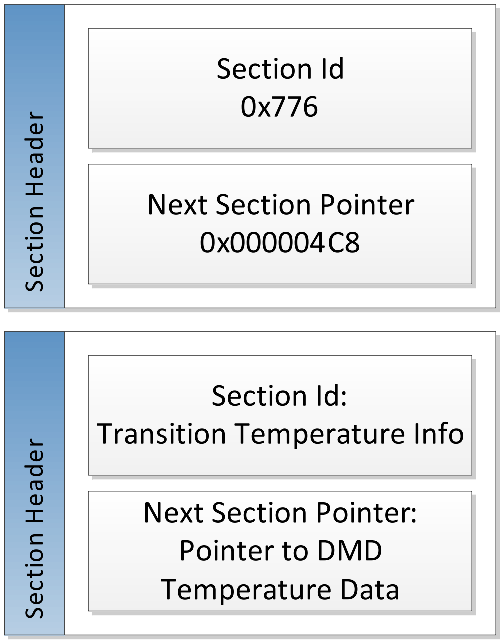


Figure 10-8. Calibration Data - Section Header: Sample Data with explanation

Section Id – This field stores a unique identifier for each section and helps identify the section data following the *Section Header.* The various sections supported and their corresponding Ids are shown in Figure 10-9.

Section	Id
Power On Defaults	0xDEF
Temperature Header Info	Ox608
Calibration Tables	0x6A2
Sensor Gain Map	0x5A1
Configuration File Defaults	0xCF5
Transition Temperature Info	0x776
DMD Temperature Data	0xD3D
Splash	0x5D2
Test Patterns	0x335
External/Video	0x387
Generic Command List Header Info	0x5E8
Generic/Other Command Lists	0x6E2
Command List LDC-SEQ Mapping	0x7DC
Command List SEQ-CMT Mapping	0x5E0
Command List CMT-Gamma Mapping	0xC37
Dimming LUT Groups	0x66B
Extra Information	0xE14
Version Info	0x674
Calibration Info	0x684

Figure 10-9. Calibration Data - Sections and Ids

- *Next Section Pointer* - This field marks the offset address from beginning of the calibration data to the start of the next *Section Header.* The value of this field should be set to 0 (zero) for the last *Section.*

The *Section Header* field in software is accessed through the following structure in calibration.h file in the software.

```
typedef struct SectionHeader
{
    Uint32 Id;
    Uint32 NextSecPtr;
} CAL_SECHDR;
```

10.2.3 Version Information

Figure 10-10 illustrates the structure of this section. Version Information has two Fields:

- *Calibration Data Identifier (or Calibration Version)*: This field stores the 32 bit CRC Value of the Calibration Data.
- *Flash Build Identifier*– This field stores the 32 bit CRC value of the entire ASIC hex file except the user data section. This field is compared against ASIC register 0x05 (Refer to ***I2C Programmer's Guide for DLPC120 ASIC (DLPU055)*** for details). A mismatch could mean the hex file in the ASIC flash may not be compatible with the configuration file used. This will result in an error being flagged and can cause serious malfunction in dimming or command lists.

Sample data for this section is shown in Figure 10-11.

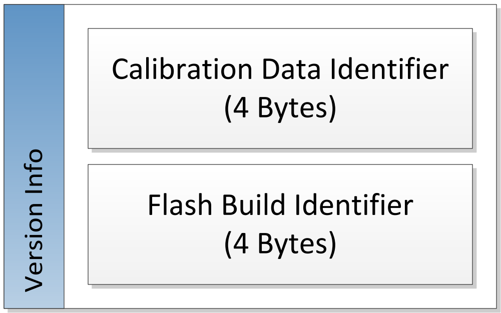


Figure 10-10. Calibration Data - Section Version Info

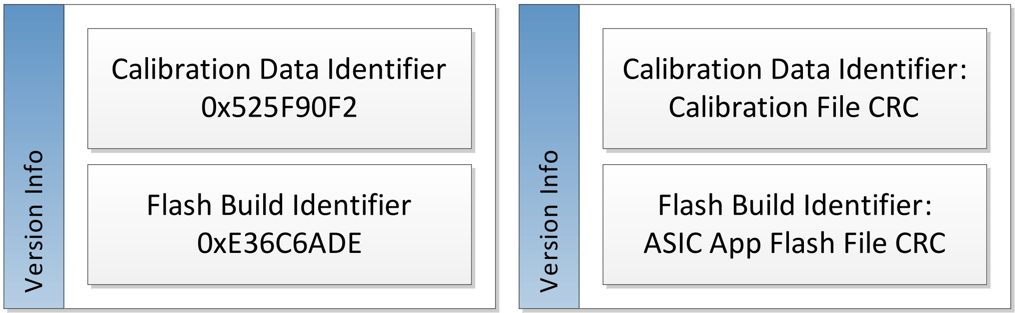


Figure 10-11. Calibration Data - Section Version Info: Sample Data with explanation

The Version Information field of the calibration data is accessed through the following structure in calibration.h file in the software:

```
typedef struct VersionInfo
{
    Uint32 CalVersion;
    Uint32 FlashBuildId;
} CAL_VERSIONINFO;
```

10.2.4 Temperature Header Information

The Temperature Header Information (or Calibration Header Information) acts as a guide to the Piccolo on how the data in the calibration tables is stored. It has entries for each calibrated temperature, its VAC Mode Switch Points and the row index in the calibration tables where calibration data corresponding to the temperature begins. In the calibration tables, data relating to all temperatures are stored in the form of rows one after the other. The Piccolo uses the header information to interpret these rows correctly. Figure 10-12 describes the structure of this section.

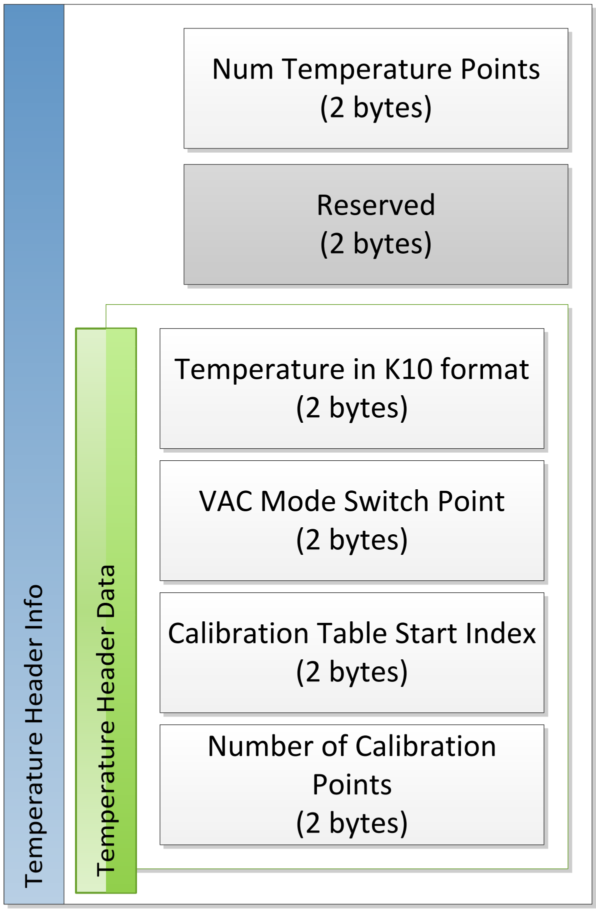

Temperature Header Info
Num Temperature Points (2 bytes)
Reserved (2 bytes)
Temperature Header Data
Temperature in K10 format (2 bytes)
VAC Mode Switch Point (2 bytes)
Calibration Table Start Index (2 bytes)
Number of Calibration Points (2 bytes)

Figure 10-12. Calibration Data - Section Temperature Header Info

This section has the following fields:

- *Num Temperature Points*: This field contains the number of temperatures for which data is present in the calibration file.
- *Temperature Header Data:* One of these fields exists for each calibrated temperature:
  - *Temperature in K10 Format:* This field contains the temperature (in K10 format) for which data is present in the rest of the Temperature Header Information field.
  - *VAC Mode Switch Point:* This field stores the backlight at which the system should switch to VAC mode (when VAC is enabled) at the corresponding temperature.
  - *Calibration Table Start Index:* This field contains the row index in the calibration tables where the data relating to this temperature begins.
  - *Number of calibration points*: Contains the number of rows or calibration data relating to this temperature.

The temperature table for any temperature can be described as the portion of calibration data starting from the row indicated by the *Calibration Table Start Index* and of size indicated by the *Number of calibration point's* field for that temperature. Figure 10-13 shows the sample data for this section.

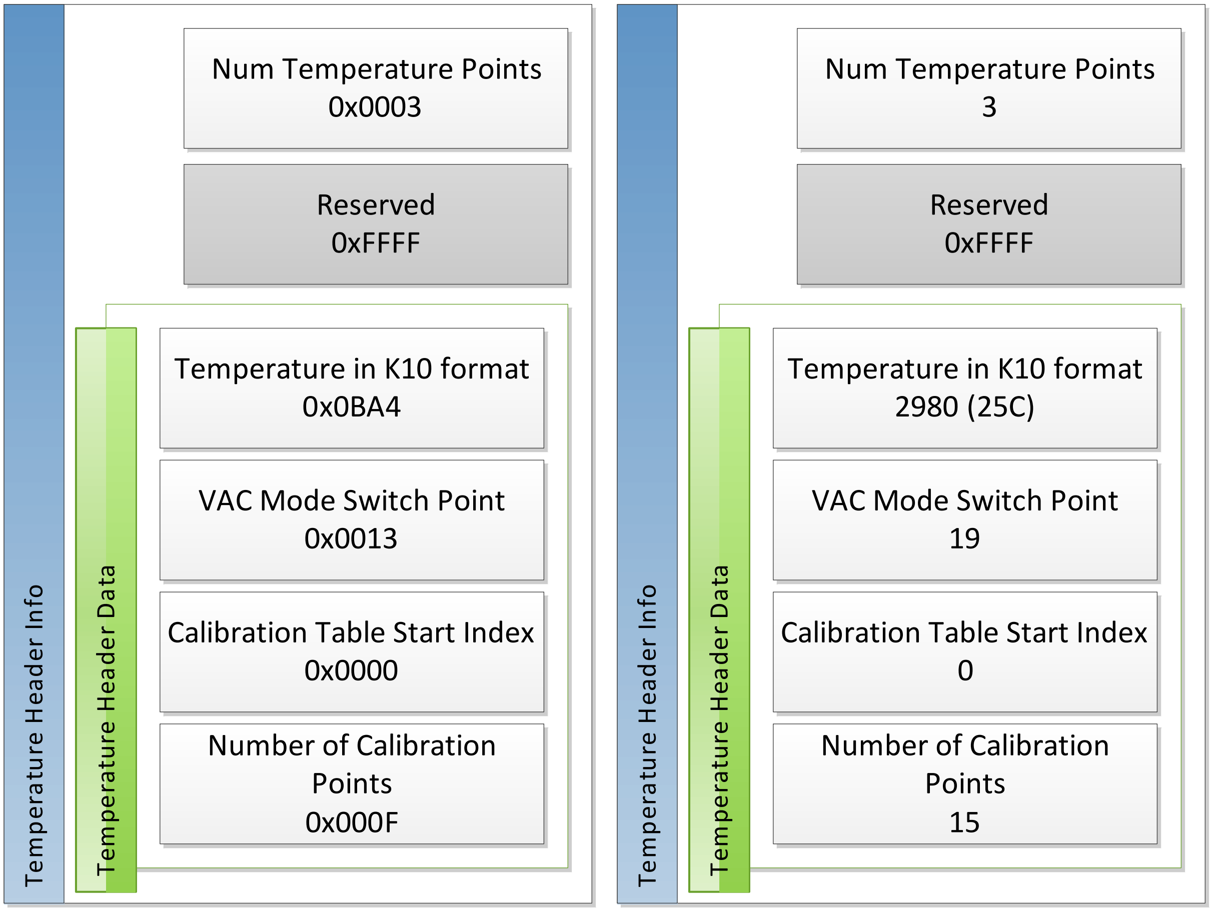


Figure 10-13. Calibration Data - Section Temperature Header Info: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct CalHeader
{
    Uint16 NumTmprPoints;
    Uint16 Reserved;
    struct TmprHeader
    {
        Uint16 TmprK10;
        Uint16 VACModeSwitchPoint;
        Uint16 CalDataIndex;
        Uint16 NumCalPoints;
    } Hdr[1];
} CAL_CALHDR;
```

10.2.5 Calibration Tables

This section contains the LED PWM values, LDC Index and Sensor Gain for various values of backlights at different temperatures. The data is arranged in the form of rows in ascending order of temperatures, and in descending order of backlights for each temperature. The Temperature Header Information discussed in Section 10.2.4, helps the Piccolo understand how the data is grouped. Figure 10-15 shows a sample Calibration Table section with valid data. The flash arrangement is shown in figure below:

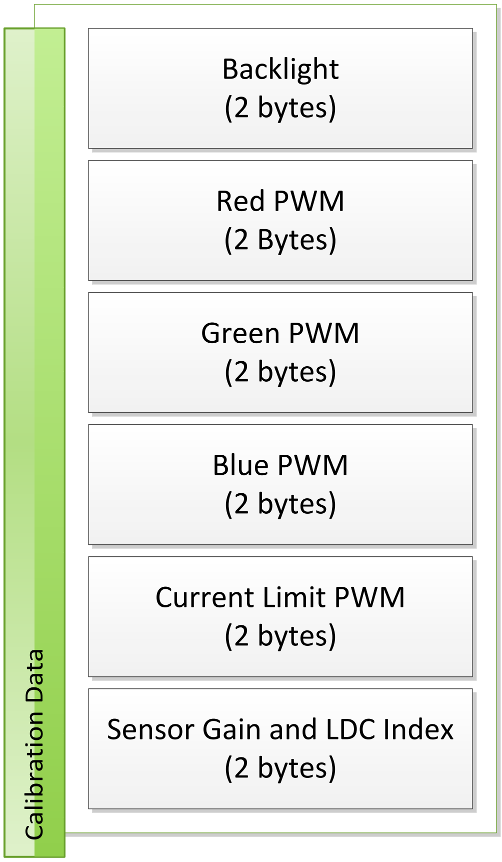


Figure 10-14. Calibration Data - Section Calibration Tables

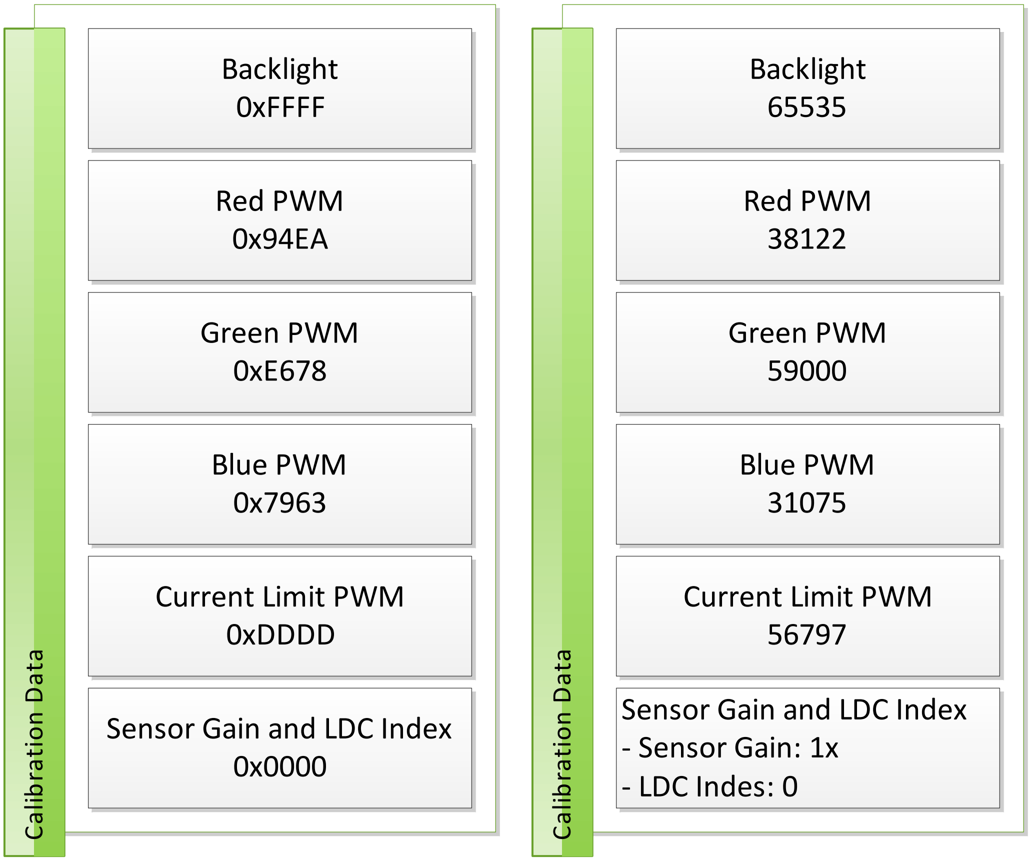


Figure 10-15. Calibration Data - Section Calibration Tables: Sample Data with explanation

The calibration data comprises of multiple entries of the structure described in Figure 10-14. Each entry has the following fields:

- *Backlight:* This field contains the backlight to which the rest of the calibration data for the corresponding row belong.
- *Red PWM:* The value for the Red PWM on a scale of 65535
- *Green PWM:* The value for the Green PWM on a scale of 65535
- *Blue PWM:* The value for the Blue PWM on a scale of 65535
- *Current Limit PWM:* The value for the Current Limit PWM on a scale of 65535
- *Sensor Gain and LDC index:* The most significant byte of this 2 byte data contains the Sensor Gain. This contains a value ranging from 0 to 3. The actual gain indicated by this value can be obtained from the sensor gain map. The least significant byte stores the LDC Index for the current backlight.

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct CalData
{
    Uint16 Backlight;
    Uint16 RedPwm;
    Uint16 GrnPwm;
    Uint16 BluPwm;
    Uint16 CurrentLimitPwm;
    Uint16 GainLDC;
} CAL_CALDATA;
```

10.2.6 Sensor Gain Map:

The Piccolo sets the sensor gain by setting two GPIO's which are inputs to the two select pins of a 4 to 1 mux. Hence the only valid values for gain are 0, 1, 2, and 3. The sensor gain map section maps these values to the actual gain that will be applied when a value is set. Figure 10-17 shows a sample mapping where 0, 1, 2 and 3 map to gain 1x, 4x, 8x and 16x respectively. The structure of this section is shown below:

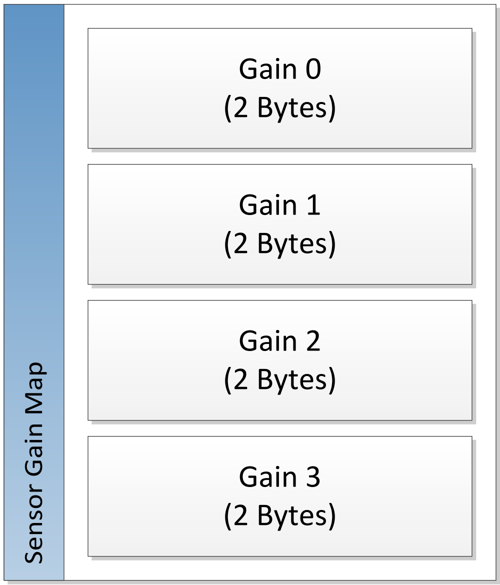


Figure 10-16. Calibration Data - Section Sensor Gain Map

The fields belonging to this section are described below:

- *Gain 0*: Actual gain applied when the Piccolo uses the value 0 for gain
- *Gain 1*: Actual gain applied when the Piccolo uses the value 1 for gain
- *Gain 2*: Actual gain applied when the Piccolo uses the value 2 for gain
- *Gain 3*: Actual gain applied when the Piccolo uses the value 3 for gain

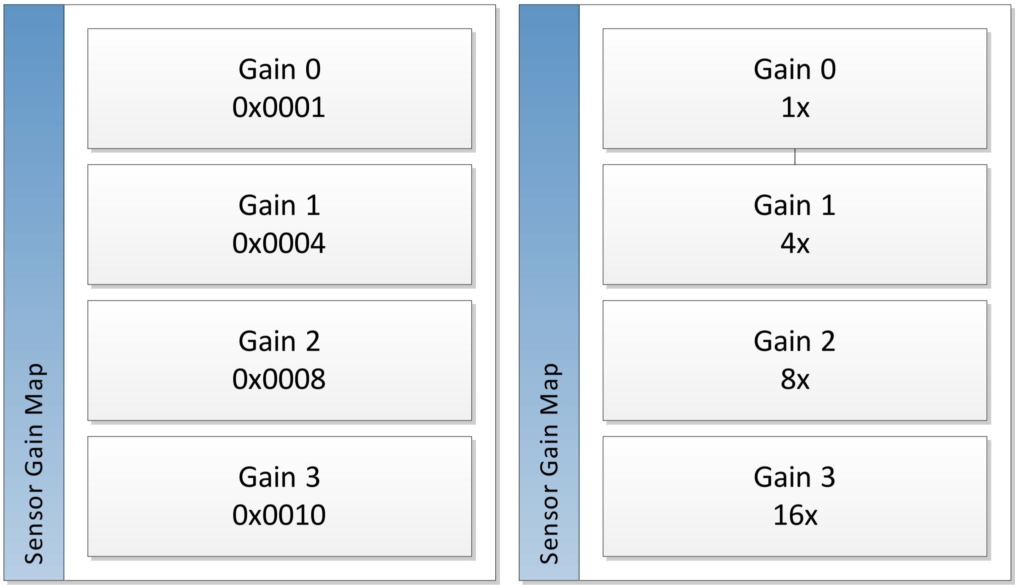


Figure 10-17. Calibration Data - Section Sensor Gain Map: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct SensorGainMap
{
    Uint16 Gain0;
    Uint16 Gain1;
    Uint16 Gain2;
    Uint16 Gain3;
} CAL_SENSORGAINMAP;
```

10.2.7 Power-Up Defaults:

This section contains default settings that will be applied at power-up. The flash arrangement is shown below:

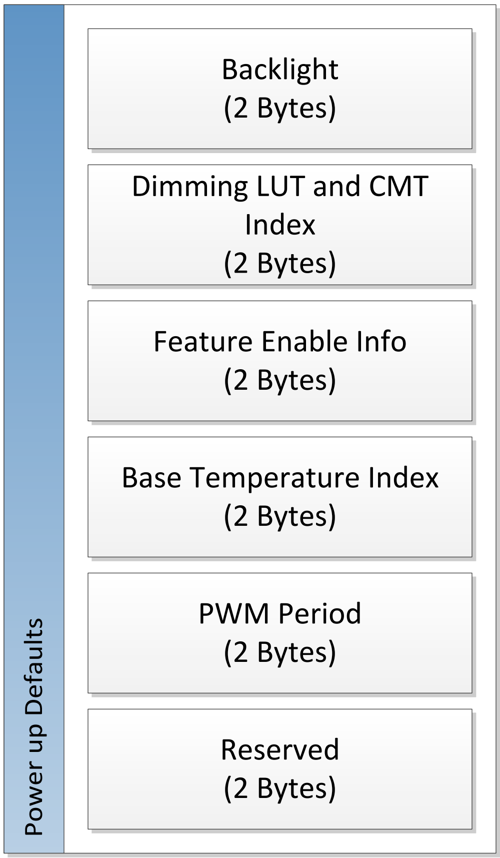

Power up Defaults
Backlight (2 Bytes)
Dimming LUT and CMT Index (2 Bytes)
Feature Enable Info (2 Bytes)
Base Temperature Index (2 Bytes)
PWM Period (2 Bytes)
Reserved (2 Bytes)

Figure 10-18. Calibration Data: Section Power-On Defaults

This section contains the following fields:

- *Backlight:* This field stores the default backlight to be applied at power on
- *Dimming LUT and CMT Index:* The MSB of this data contains the index of the default Dimming LUT group to be used. The LSB contains the default CMT Index.
- *Feature Enable Info:* This field contains the features default state of different features supported by the Piccolo Software. The table below lists the features that each bit corresponds to. For Master On, a ‘1’ indicates System On and ‘0’ indicates System Off. For other features, ‘1’ indicates enabled and ‘0’ indicates disabled.

Bits	15 – 4	3	2	1	0
Feature	*Reserved*	Voltage Supervision	VAC	Temperature Compensation	Master On

- *Base Temperature Index*: This stores the index of the entry in the Temperature Header Information Section) corresponding to the base temperature.
- *PWM Period:* This field stores the number of coarse low resolution PWM Pulses used in each PWM cycle.

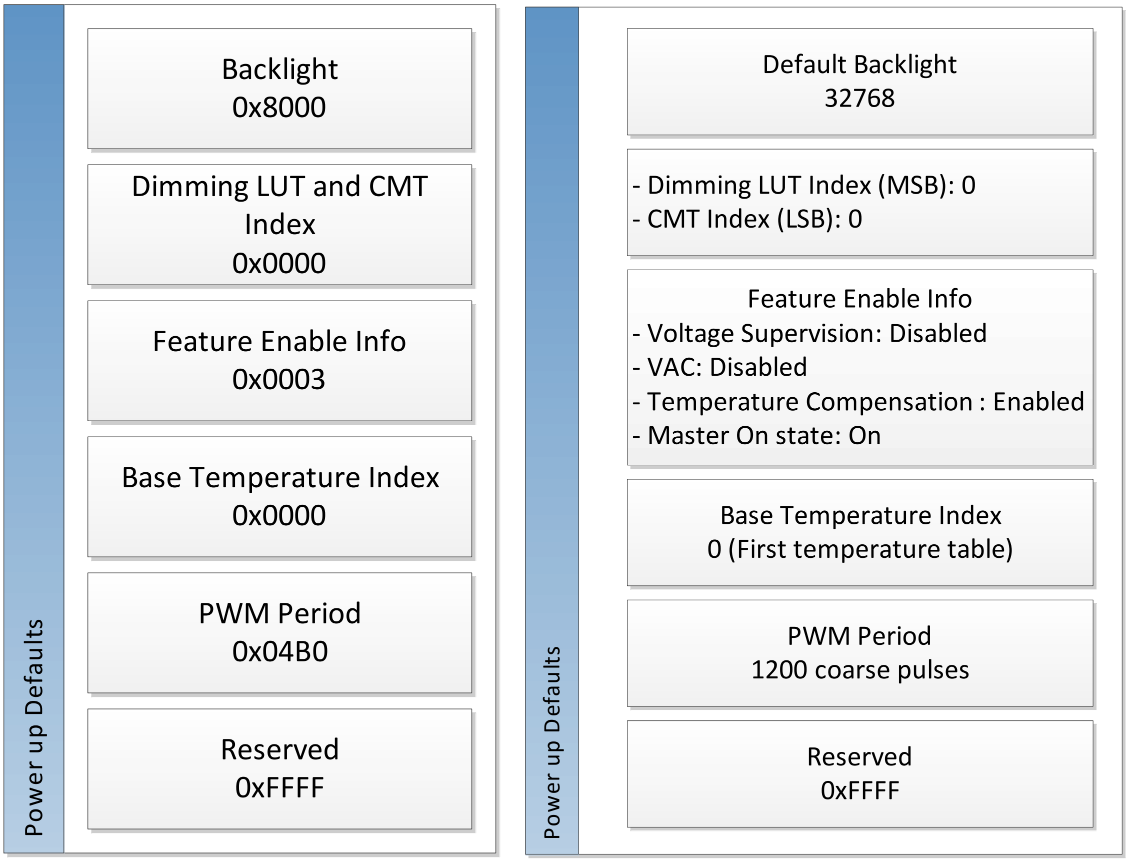


Figure 10-19. Calibration Data: Section Power-Up defaults: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct Defaults
{
    Uint16 Backlight;
    Uint16 DimLUTCMT;
    Uint16 FeatureEnables;
    Uint16 BaseTmprIndex;
    Uint16 PWMPeriod;
    Uint16 Reserved;
} CAL_DEFAULTS;
```

10.2.8 Calibration Information:

This section stores calibration related information. Figure 10-20 shows the flash arrangement of this section.

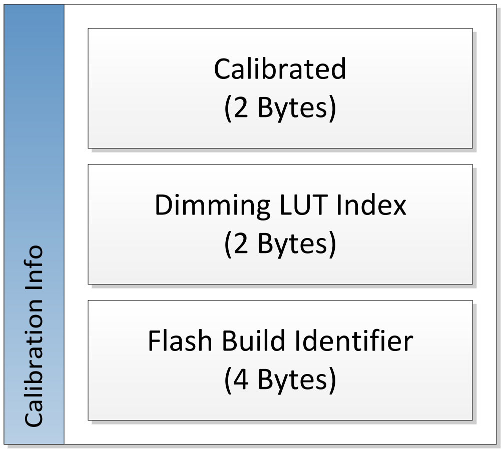


Figure 10-20. Calibration Data - Section Calibration Info

It contains the following fields:

- *Calibrated:* a ‘1’ in this field indicates that the calibration file being programmed is an unmodified product of calibration. A ‘0’ indicates that it is either manually written or modified.
- *Dimming LUT Index:* This field contains the index of the Dimming LUT Group that was used during calibration
- *Flash Build ID:* This field stores the 32 bit CRC Identifier of the Configuration file used during calibration.

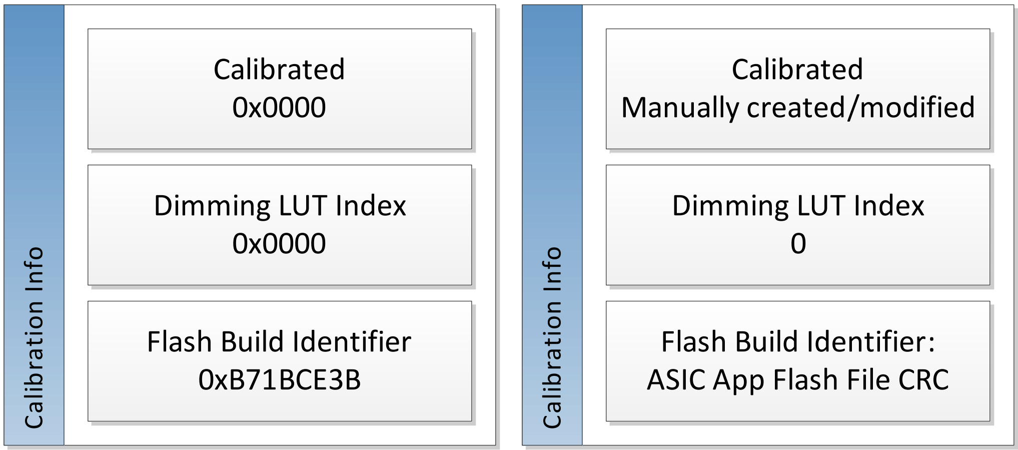


Figure 10-21. Calibration Data - Section Calibration Info: Sample Data with explanations

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct CalibrationInfo
{
    Uint16 Calibrated;
    Uint16 DimLUTIndex;
    Uint32 FlashBuildId;
} CAL_CALIBRATIONINFO;
```

10.2.9 Configuration File Defaults:

This section contains addresses of command lists that the Piccolo will execute on startup. The structure of this section is described below:

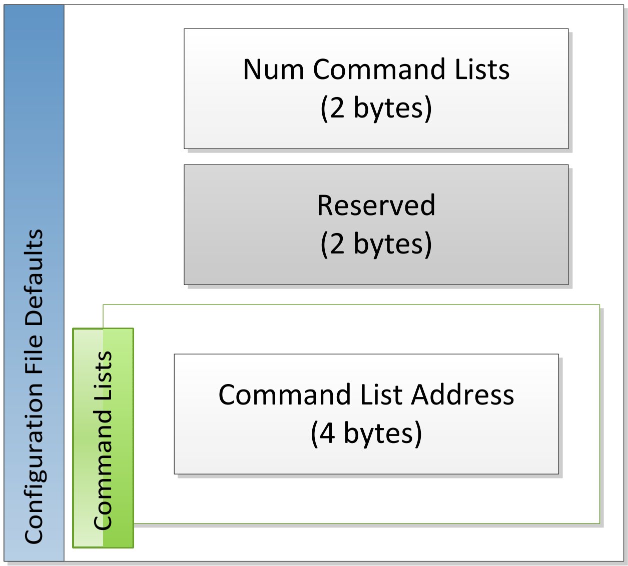


Figure 10-22. Configuration Data - Section Configuration File Defaults

- *Num Command Lists*: Stores the number of command lists that need to be executed on Power-On
- *Command Lists*: An entry of this field is present for each command list to be executed.
  - Command List Address: This field stores the start address of the command list.

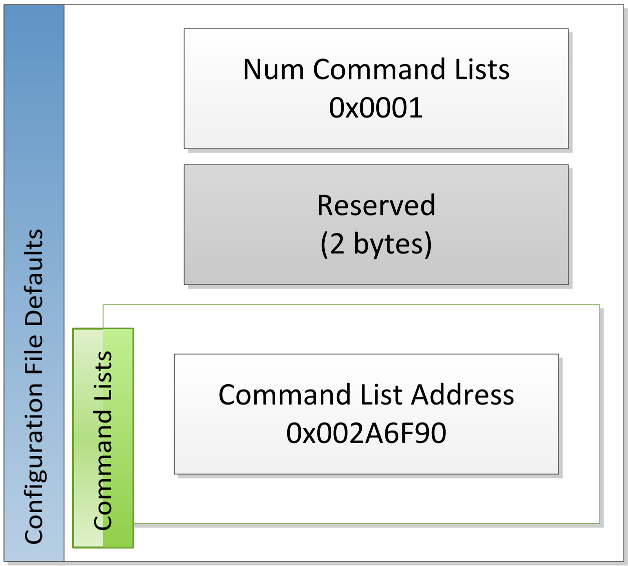


Figure 10-23. Section Configuration File Defaults: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct CfgDefaults
{
    Uint16 NumCmdLists;
    Uint16 Reserved;
    struct CmdLists
    {
        Uint32 CmdListAddr;
    }Cmd[1];
} CAL_CFGDEFAULTS;
```

10.2.10 Temperature Transition Tables:

NOTE: Transition temperatures are not needed for DLP3030-Q1 operation. The DLP3030-Q1 DMD requires no derating over the entire operating temperature range.

Transition temperatures are temperatures beyond which some sequences, which are permitted for lesser temperatures, are not allowed. Sequences are disallowed by restricting LDC values. The Temperature Transition Table contains the number of transition temperatures present, and the minimum permitted LDC index for each transition temperature. Figure 10-25 shows some sample Temperature Transition Data. This data is the same as that corresponding to the lowest transition temperature in the example Configuration File.

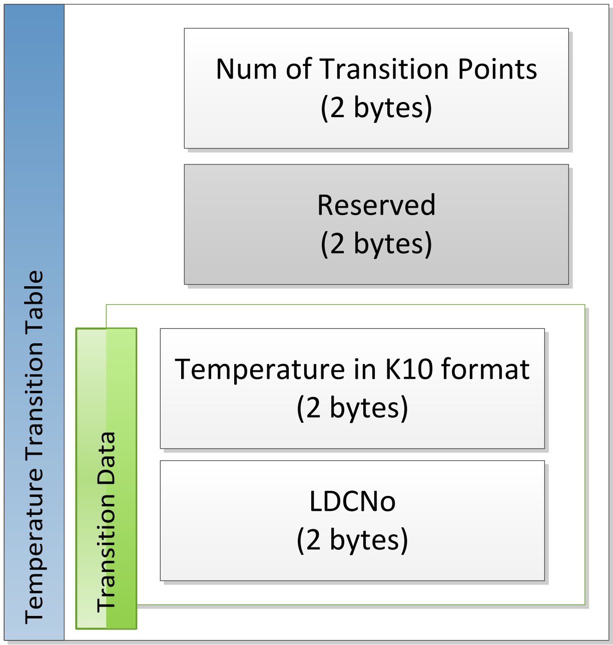


Figure 10-24. Configuration Data - Section Temperature Transition Table

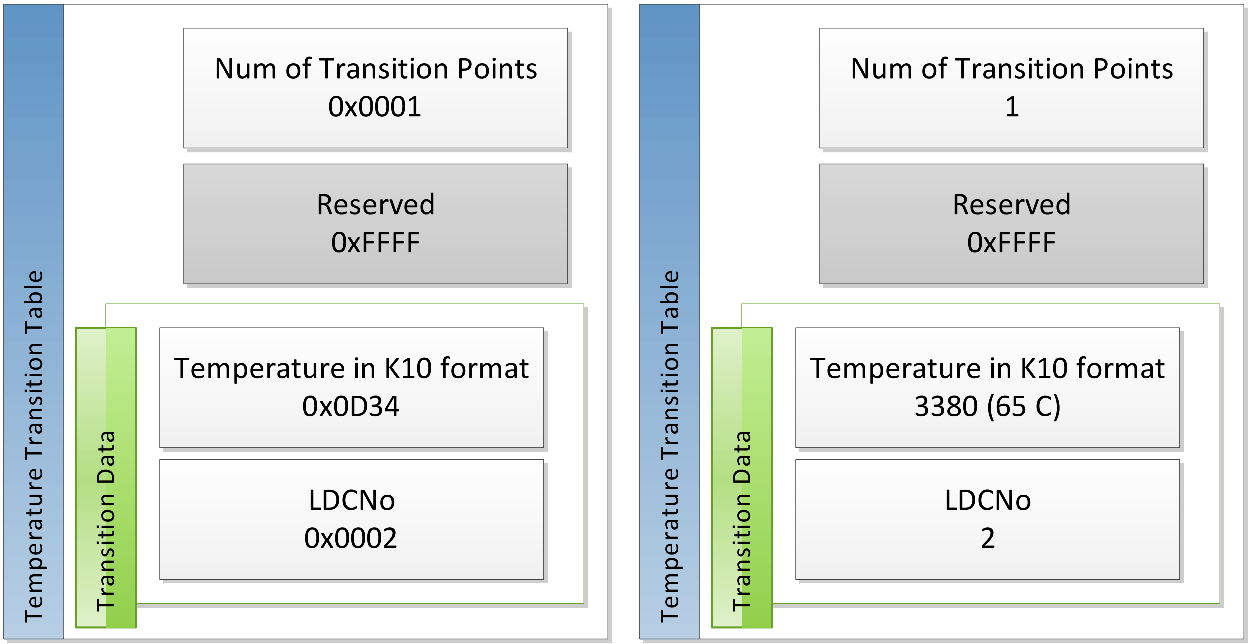


Figure 10-25. Configuration Data - Section Temperature Transition Table: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

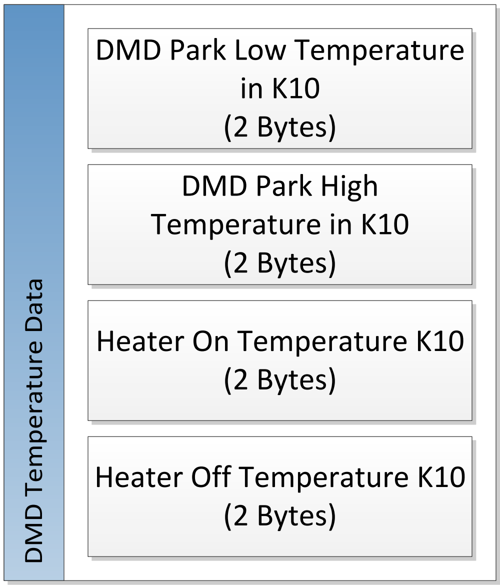
```
typedef struct TmprTransitionData
{
    Uint16 NumPoints;
    Uint16 Reserved;
    struct TmprTrans
    {
        Uint16 TmprK10;
        Uint16 LDCNo;
    } Trans[1];
} CAL_TMPRTRANS;
```

10.2.11 DMD Temperature

The DMD Temperature data contains upper and lower limits for the DMD temperature beyond which the DMD is parked. The heater is not needed for DLP3030-Q1 operation. It should remain off during normal use.

Figure 10-27 shows sample values for the DMD Temperature data section.

Below is the structure of this section:



DMD Temperature Data

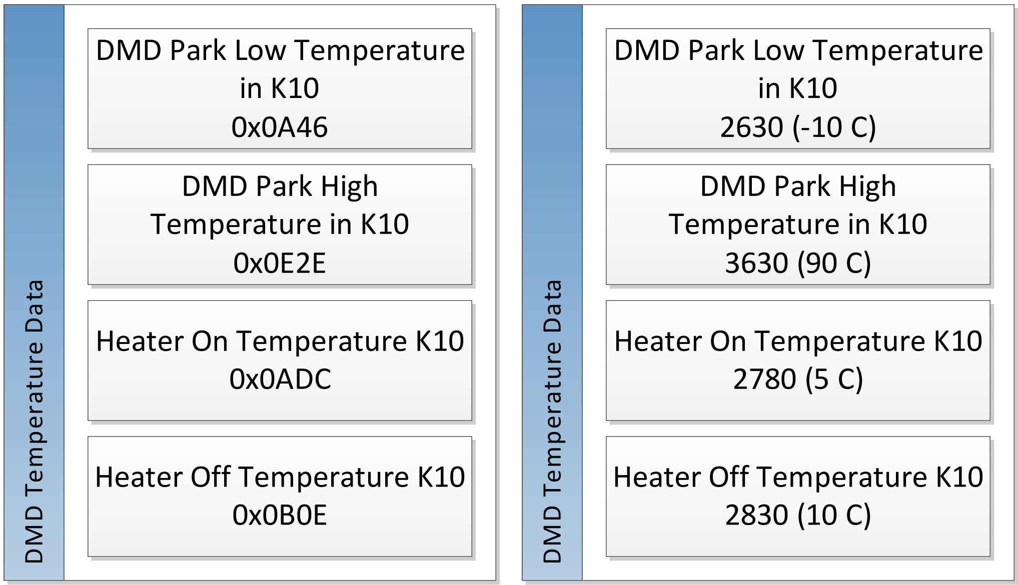
DMD Park Low Temperature in K10 (2 Bytes)

DMD Park High Temperature in K10 (2 Bytes)

Heater On Temperature K10 (2 Bytes)

Heater Off Temperature K10 (2 Bytes)

Figure 10-26. Configuration Data - Section DMD Temperature



DMD Temperature Data

DMD Park Low Temperature in K10 0x0A46

DMD Park High Temperature in K10 0x0E2E

Heater On Temperature K10 0x0ADC

Heater Off Temperature K10 0x0B0E

DMD Temperature Data

DMD Park Low Temperature in K10 2630 (-10 C)

DMD Park High Temperature in K10 3630 (90 C)

Heater On Temperature K10 2780 (5 C)

Heater Off Temperature K10 2830 (10 C)

Figure 10-27. Configuration Data - Section DMD Temperature: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct DMDTmprData
{
    Uint16 ParkLoTmprK10;
    Uint16 ParkHiTmprK10;
    Uint16 HeaterOnTmprK10;
    Uint16 HeaterOffTmprK10;
} CAL_DMDTMPRDATA;
```

10.2.12 Splash Command Lists

The splash command list data contains details of each splash present in the configuration file.

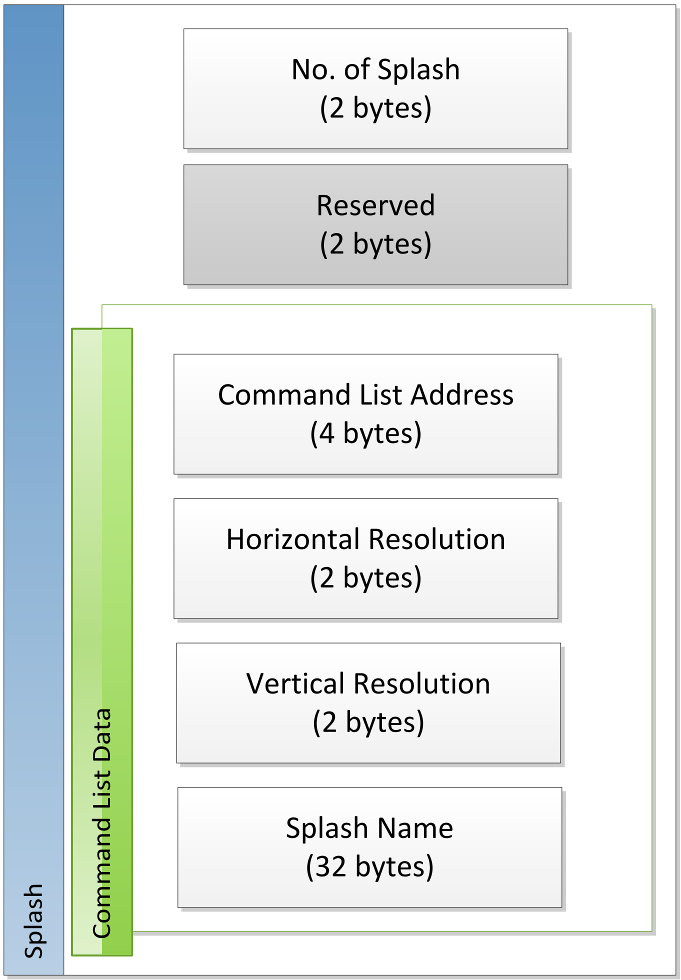


Figure 10-28. Configuration Data: Section Splash

This section has the following fields:

- *Num Splash:* stores the number of splash command lists in the configuration file
- *Command List Data*: one copy of this field exists for each Splash Command List present. This field contains the following data:
  - *Command List Address*: Start Address of the command list
  - *Horizontal Resolution*
  - *Vertical Resolution*
  - *Name:* Name of the splash command list. This is an array of 16 bit integers. Each element stores ASCII values of two characters in little endian format. Refer to for more details on this format. Shown below is an example of how “Splash Image 1” is represented in the array.

Array Index	Value	Characters Stored
[0x00000000]	0x7053	P (0x70), S(0x53)
[0x00000001]	0x616c	a, l
[0x00000002]	0x6873	h, s
[0x00000003]	0x4920	I, [space]
[0x00000004]	0x616d	a, m
[0x00000005]	0x6567	e, g
[0x00000006]	0x3120	1, [space]

[0x00000007]	0x0000	null, null
[0x00000008]	0x0000	null, null
[0x00000009]	0x0000	null, null
[0x0000000a]	0x0000	null, null
[0x0000000b]	0x0000	null, null
[0x0000000c]	0x0000	null, null
[0x0000000d]	0x0000	null, null
[0x0000000e]	0x0000	null, null
[0x0000000f]	0x0000	null, null

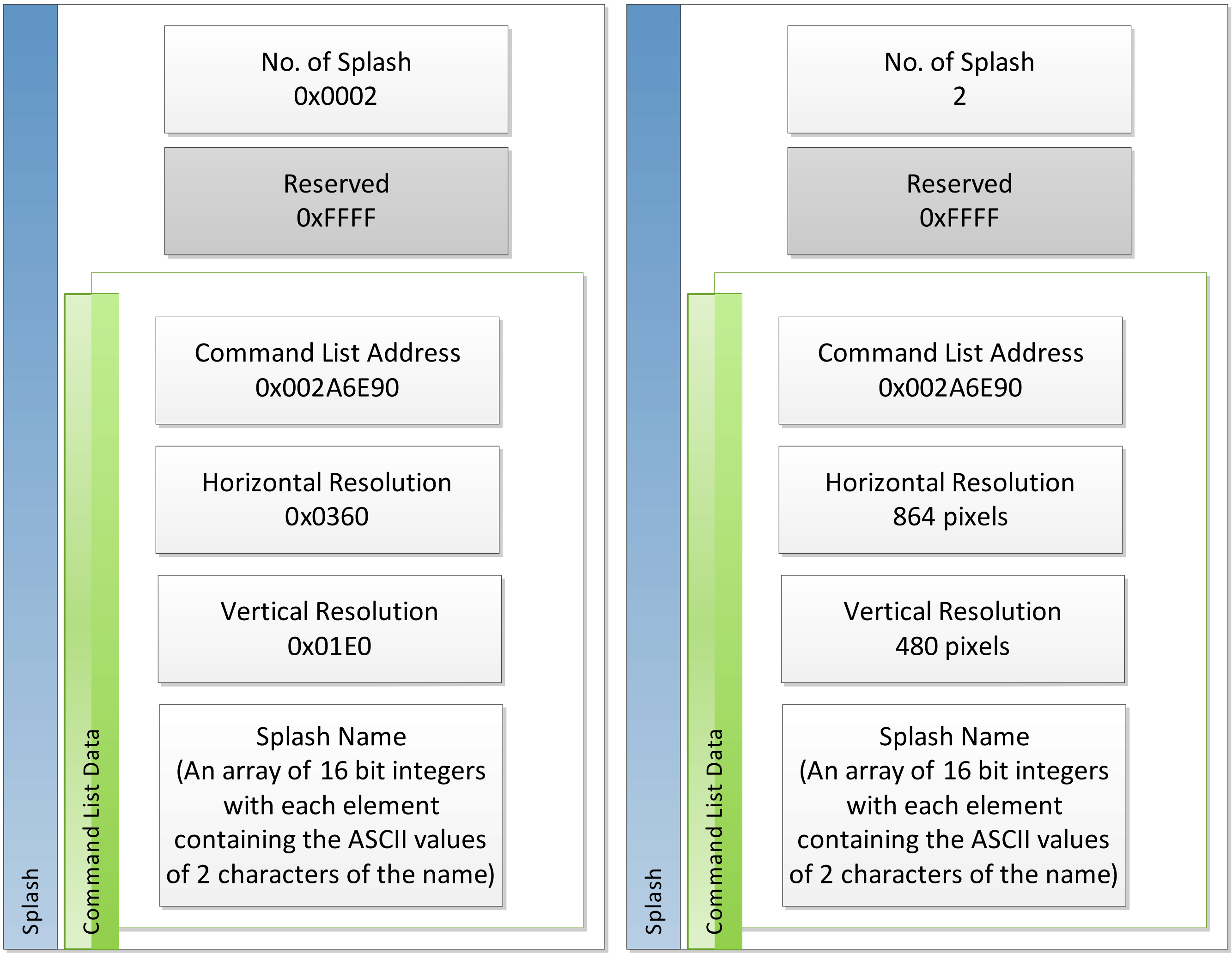


Figure 10-29. Configuration Data: Section Splash: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct SplashCommandList
{
    Uint16 NumSplash;
    Uint16 Reserved;
    struct SplashData
    {
        Uint32 CmdListAddr;
        Uint16 Horiz;
        Uint16 Vert;
        Uint16 Name[(MAX_SPLASH_NAME_LEN+1)/2];
    } Splash[1];
}CAL_SPLASHCMDLIST;
```

10.2.13 Test Pattern Command Lists

Test Pattern command list data contains following fields:

- *Num Test Patterns*: stores the number of Test Patterns in the configuration file
- *Command List Data*: one copy of this field exists for each Test Pattern Command List present. This field contains the following data:
  - *Command List Address*: Start Address of the command list
  - *Name*: Name of the Test Pattern. This is an array of 16 bit integers. Each element stores ASCII values of two characters in little endian format. Refer to Section A.2 for more details on this format. Shown below is an example of how “Solid White” is represented in the array.

Array Index	Value	Characters Stored
[0x00000000]	0x6f53	o(0x6f), S(0x53)
[0x00000001]	0x696c	i, l
[0x00000002]	0x2064	[space], d
[0x00000003]	0x6857	h, W
[0x00000004]	0x7469	t, i
[0x00000005]	0x0065	null, e
[0x00000006]	0x0000	null, null
[0x00000007]	0x0000	null, null
[0x00000008]	0x0000	null, null
[0x00000009]	0x0000	null, null
[0x0000000a]	0x0000	null, null
[0x0000000b]	0x0000	null, null
[0x0000000c]	0x0000	null, null
[0x0000000d]	0x0000	null, null
[0x0000000e]	0x0000	null, null
[0x0000000f]	0x0000	null, null

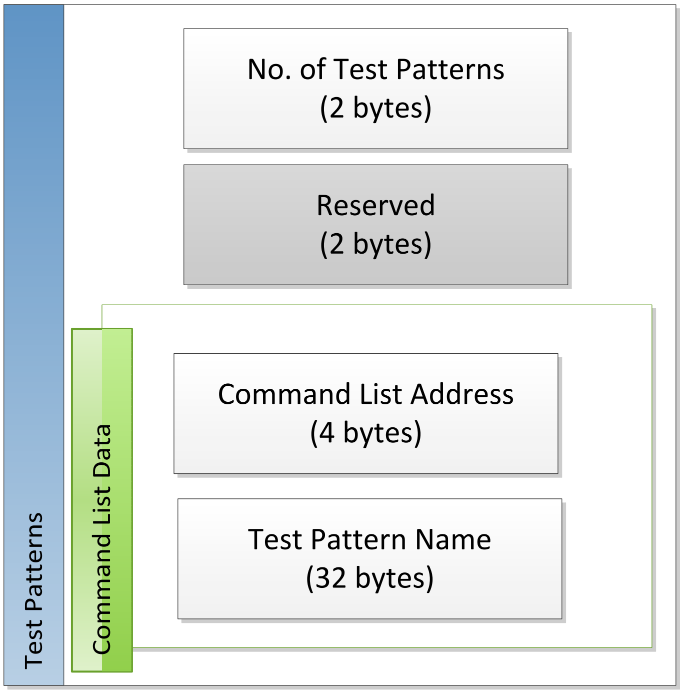


Figure 10-30. Configuration Data - Section Test Patterns

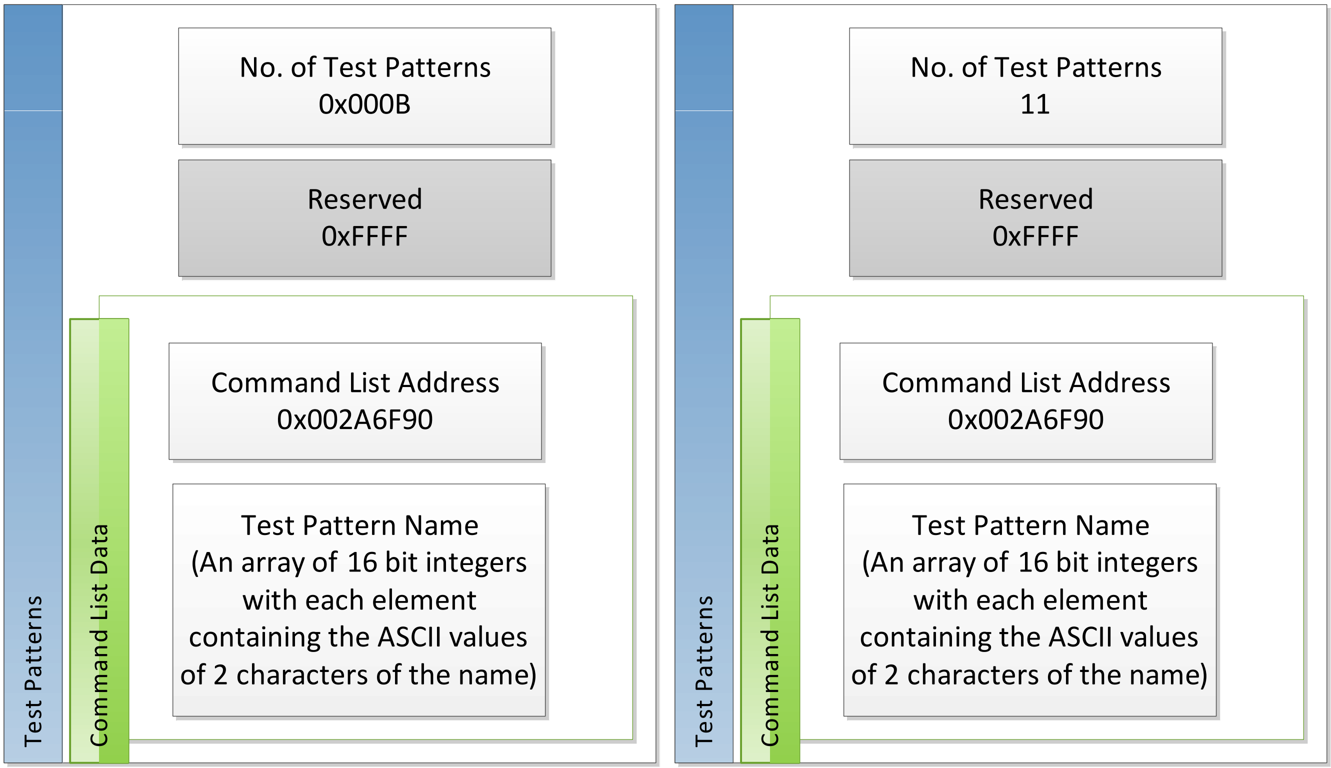


Figure 10-31. Configuration Data - Section Test Patterns: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct TestPatternList
{
```

```
          Uint16 NumTestPatterns;
          Uint16 Reserved;
          struct TestPatternData
          {
              Uint32 CmdListAddr;
              Uint16 Name[(MAX_TEST_PTN_NAME_LEN+1)/2];
           } TestPattern[1];
      }CAL_TESTPTNCMDLIST;
```

10.2.14 External Video Command Lists

The External Video command list data contains the following data:

- *Num External Video Command Lists:* stores the number of External Video command lists in the configuration file
- *Command List Data*: one copy of this field exists for each External Video command list present. This field contains the following data:
  - *Command List Address*: Start Address of the command list
  - *Input Horizontal Resolution*
  - *Input Vertical Resolution*
  - *Output Horizontal Resolution*
  - *Output Vertical Resolution*
  - *Frequency:* The frame rate in frames per second

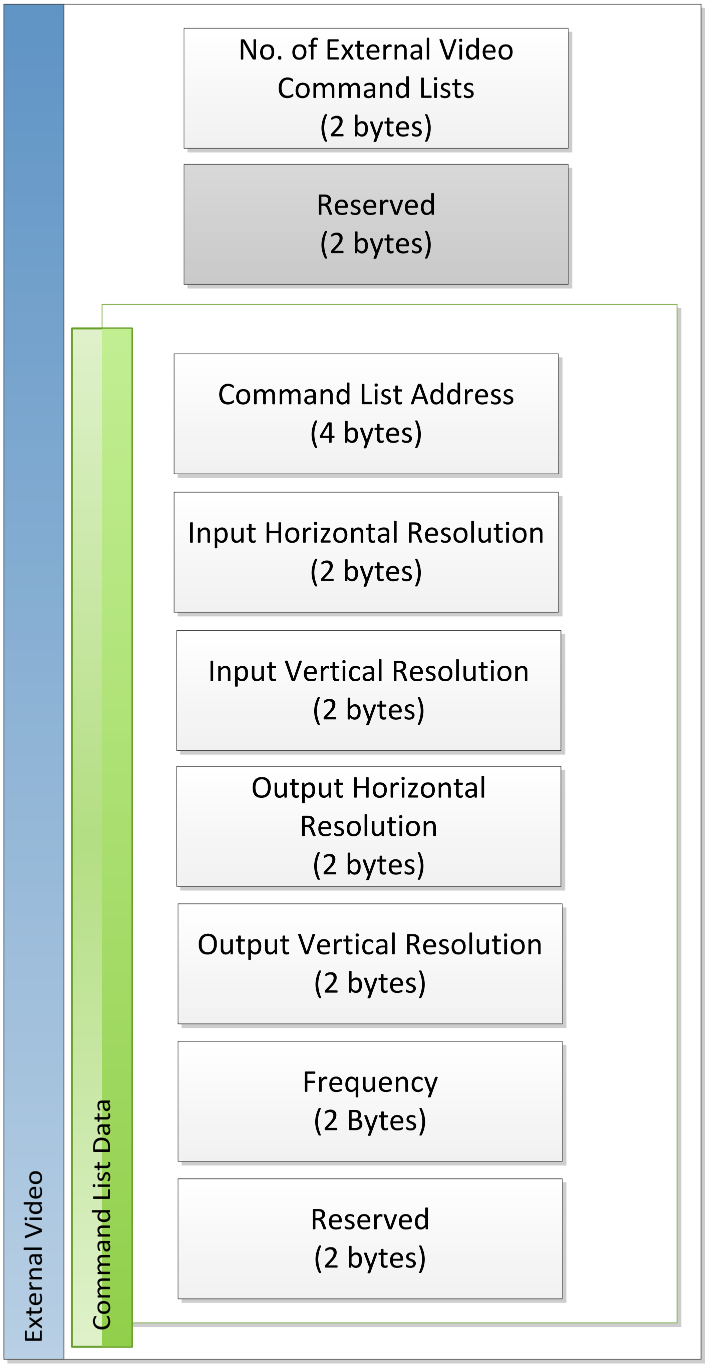

No. of External Video Command Lists (2 bytes)
Reserved (2 bytes)
Command List Address (4 bytes)
Input Horizontal Resolution (2 bytes)
Input Vertical Resolution (2 bytes)
Output Horizontal Resolution (2 bytes)
Output Vertical Resolution (2 bytes)
Frequency (2 Bytes)
Reserved (2 bytes)
External Video
Command List Data

Figure 10-32. Configuration Data - Section External Video

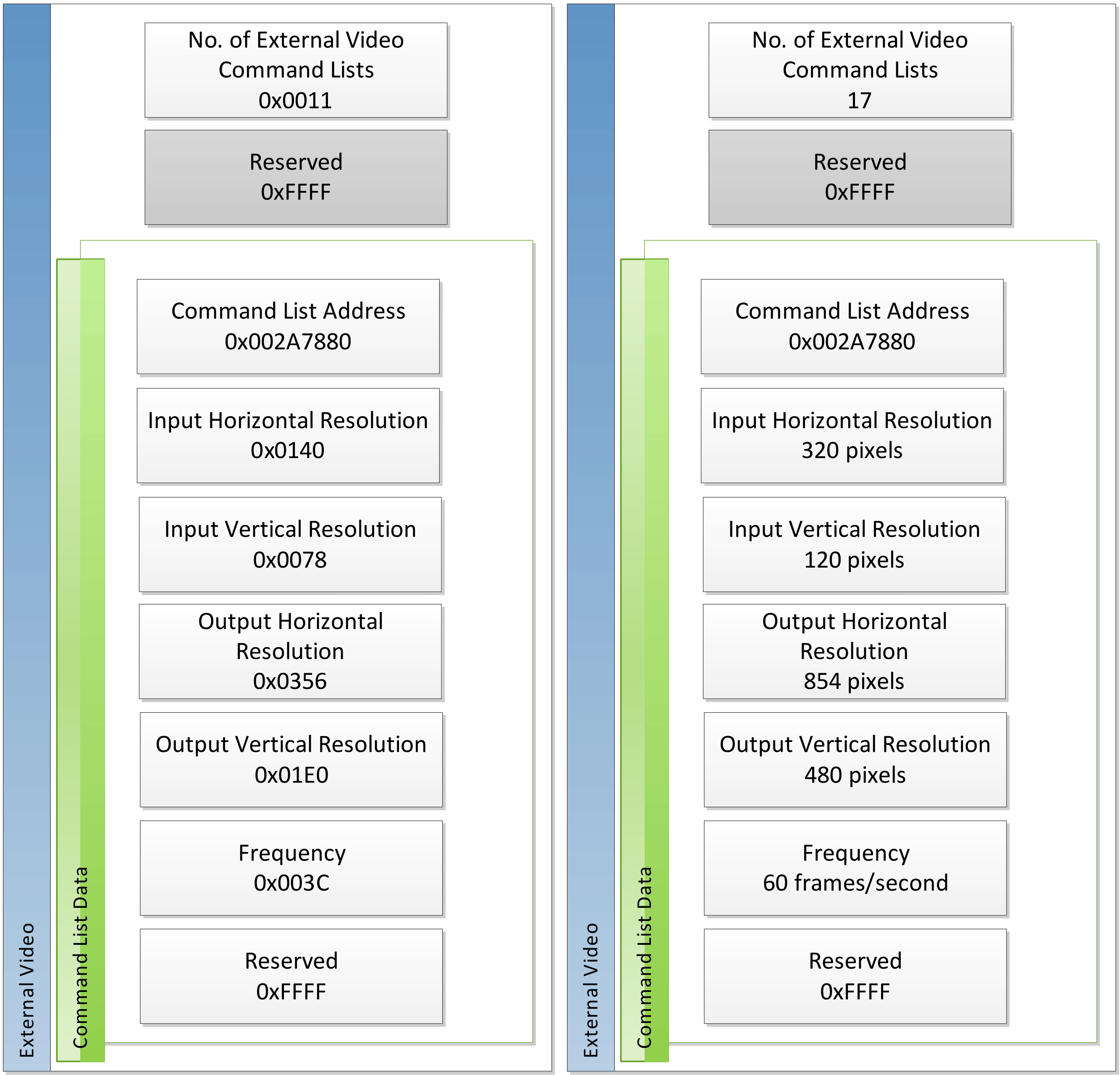


Figure 10-33. Configuration Data - Section External Video: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct ExternalVideoList
{
    Uint16 NumExtVideoLists;
    Uint16 Reserved;
    struct ExtVideoData
    {         Uint32 CmdListAddr;
        Uint16 Horiz;
        Uint16 Vert;
        Uint16 OutputHoriz;
        Uint16 OutputVert;
        Uint16 Frequency;
        Uint16 Reserved;
        } ExtVid[1];
}CAL_EXTVIDCMDLIST;
```

10.2.15 Generic/Other Command List Header Information:

The Generic/Other Command List Header Information stores data for each type of generic command list that is present in the configuration file. This helps the Piccolo locate any generic command list given a type and an index. All generic command lists are grouped by type and stored consecutively. The Piccolo uses the header information to see where the command list information for a particular type starts (Command List Start Index) and offsets this number by the required index. Figure 10-34 shows the structure of the header information.

This section contains the following fields:

- *Number of Generic Command List Types*
- *Generic Command List Header Data:* one copy of this field exists for every generic command list type. This field has the following data:
  - *Generic Command List Type:* This is an array of 16 bit integers. Each element stores ASCII values of two characters in little endian format. Refer to Chapter 16.2 for more details on this format.
  - *Command List Start Index:* The index (with respect to the list of all Generic Command Lists) at which data relevant to the type (Generic Command List Type) begins.
  - *Number of Generic Command Lists of this type*

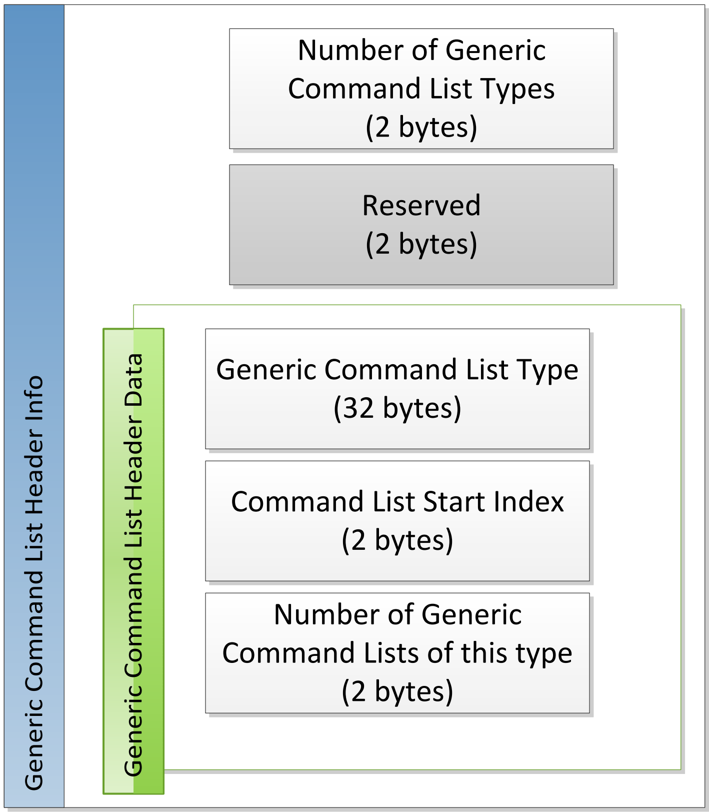

Generic Command List Header Info
Number of Generic Command List Types (2 bytes)
Reserved (2 bytes)
Generic Command List Header Data
Generic Command List Type (32 bytes)
Command List Start Index (2 bytes)
Number of Generic Command Lists of this type (2 bytes)

Figure 10-34. Configuration Data - Section Generic/Other Command List Header Info

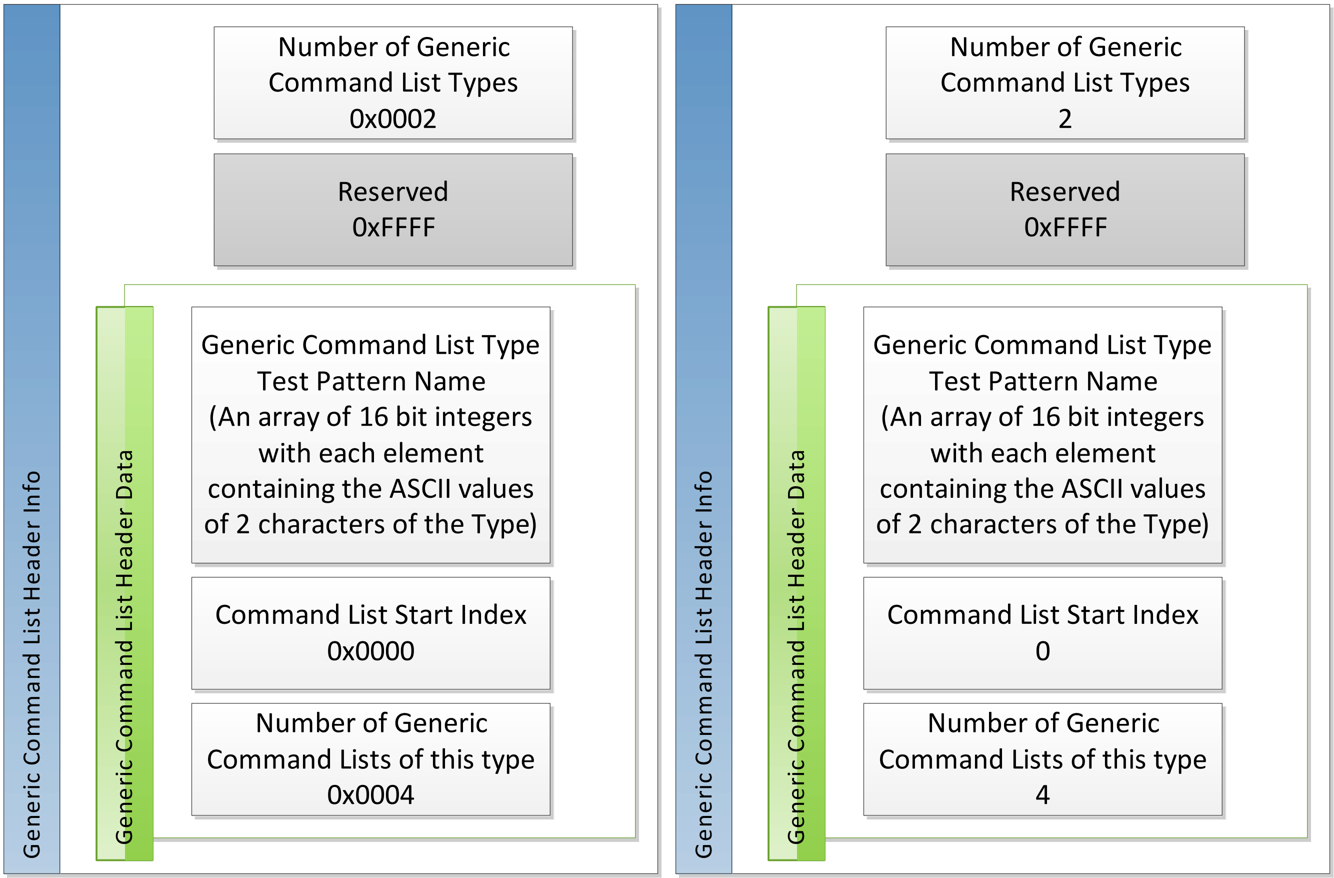

Number of Generic Command List Types
0x0002
Reserved
0xFFFF
Generic Command List Type Test Pattern Name (An array of 16 bit integers with each element containing the ASCII values of 2 characters of the Type)
Command List Start Index
0x0000
Number of Generic Command Lists of this type
0x0004
Generic Command List Header Info
Generic Command List Header Data
Number of Generic Command List Types
2
Reserved
0xFFFF
Generic Command List Type Test Pattern Name (An array of 16 bit integers with each element containing the ASCII values of 2 characters of the Type)
Command List Start Index
0
Number of Generic Command Lists of this type
4
Generic Command List Header Info
Generic Command List Header Data

Figure 10-35. Configuration Data - Section Generic/Other Command List Header Info: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct GenericListHeader
{
    Uint16 NumTypes;
    Uint16 Reserved;
    struct CmdListData
    {   Uint16 Type[(MAX_GENERIC_LIST_TYPE_LEN+1)/2];
        Uint16 StartIndex;
        Uint16 NumLists;
    } Hdr[1];
}CAL_GENERICLISTSHDR;
```

10.2.16 Generic/Other Command List Data:

This section contains all generic command lists stored one after the other. The flash arrangement is shown below:

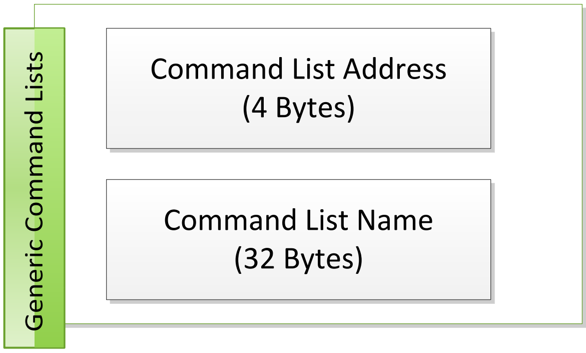


Figure 10-36. Configuration Data - Section Generic/Other Command Lists

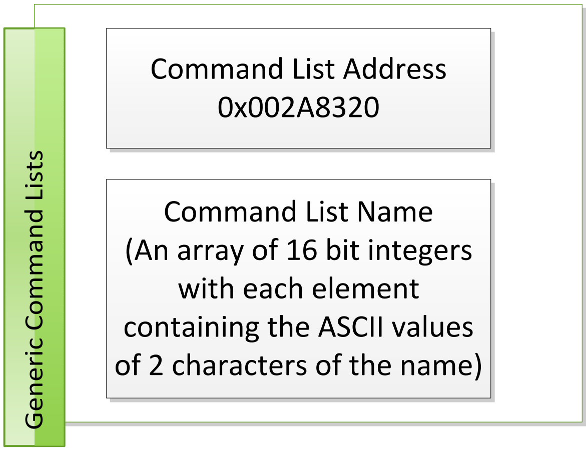


Figure 10-37. Configuration Data - Section Generic/Other Command Lists: Sample Data

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct GenericCmdList
{
    Uint32 CmdListAddr;
    Uint16 Name[(MAX_GENERIC_LIST_NAME_LEN+1)/2];
}CAL_GENERICLISTS;
```

10.2.17 Dimming LUT (Look Up Table) Groups

Each Dimming LUT group contains information about the sequences in the system. This includes:

1. RGB duty cycles:

   Only the red and green duty cycles are stored. The blue duty cycle is equal to (100 - red - green). The duty cycles are floating point values stored as per the IEEE 754 format (Refer to Section A.3)
2. On/Off duty cycles
3. Base address of LDC, CMT, SEQ
4. Start indexes for data relating to the current LUT on the global LDC-SEQ, SEQ-CMT, and CMT-Gamma Tables. In other words, the configuration file contains 3 global tables i.e. LDC-SEQ, SEQ-CMT, and CMT-Gamma Tables. These tables contain data for all the dimming LUT groups packed together. The individual Dimming LUT group tables contain the location (indexes) of its own information on the global tables.

Dimming LUT groups are referred to as Sequence Tables in the example configuration file shown in Figure 10-2**.**

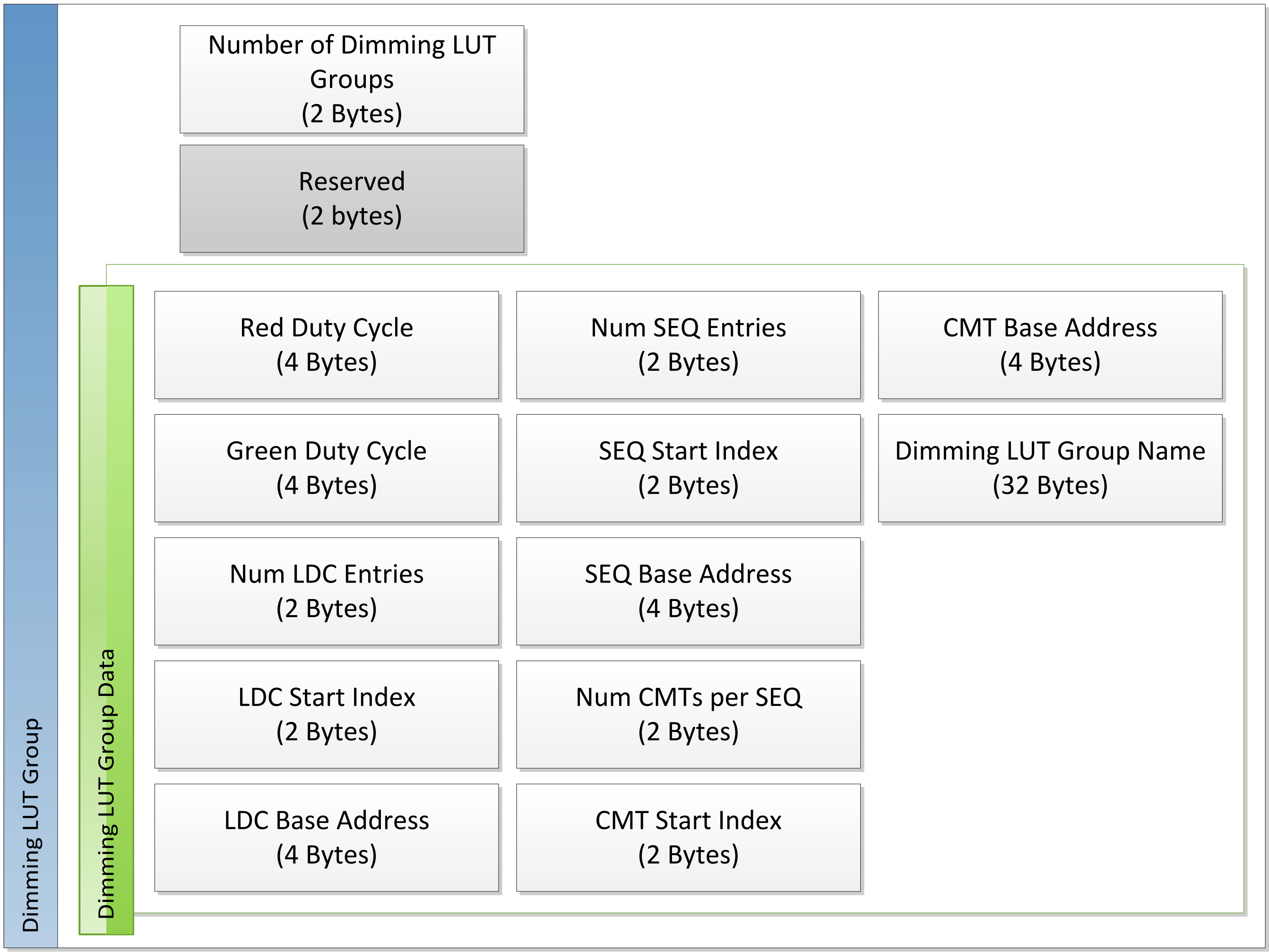


Figure 10-38. Configuration Data: Section Dimming LUT Groups

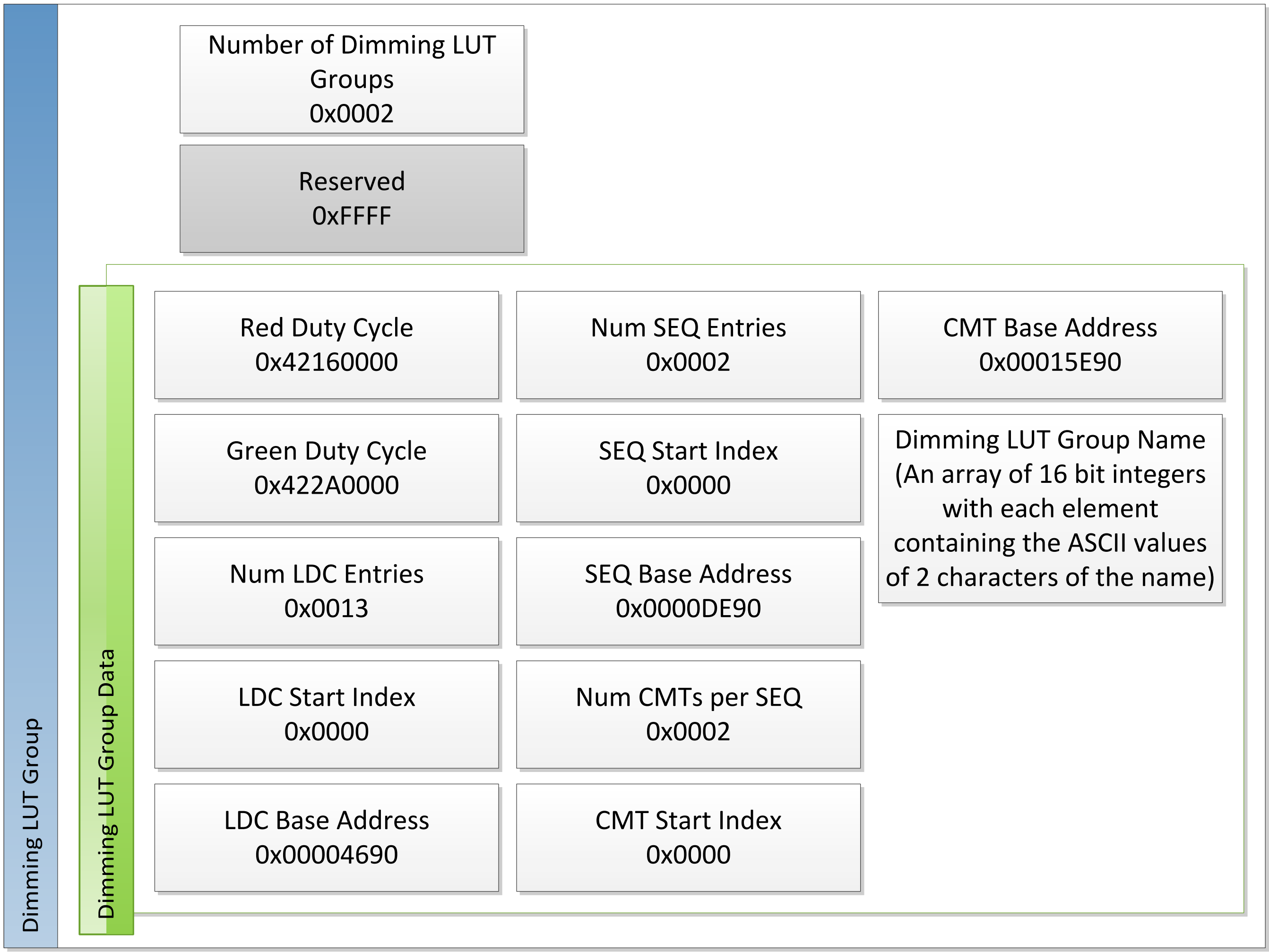


Figure 10-39. Configuration Data: Section Dimming LUT Groups: Sample Data

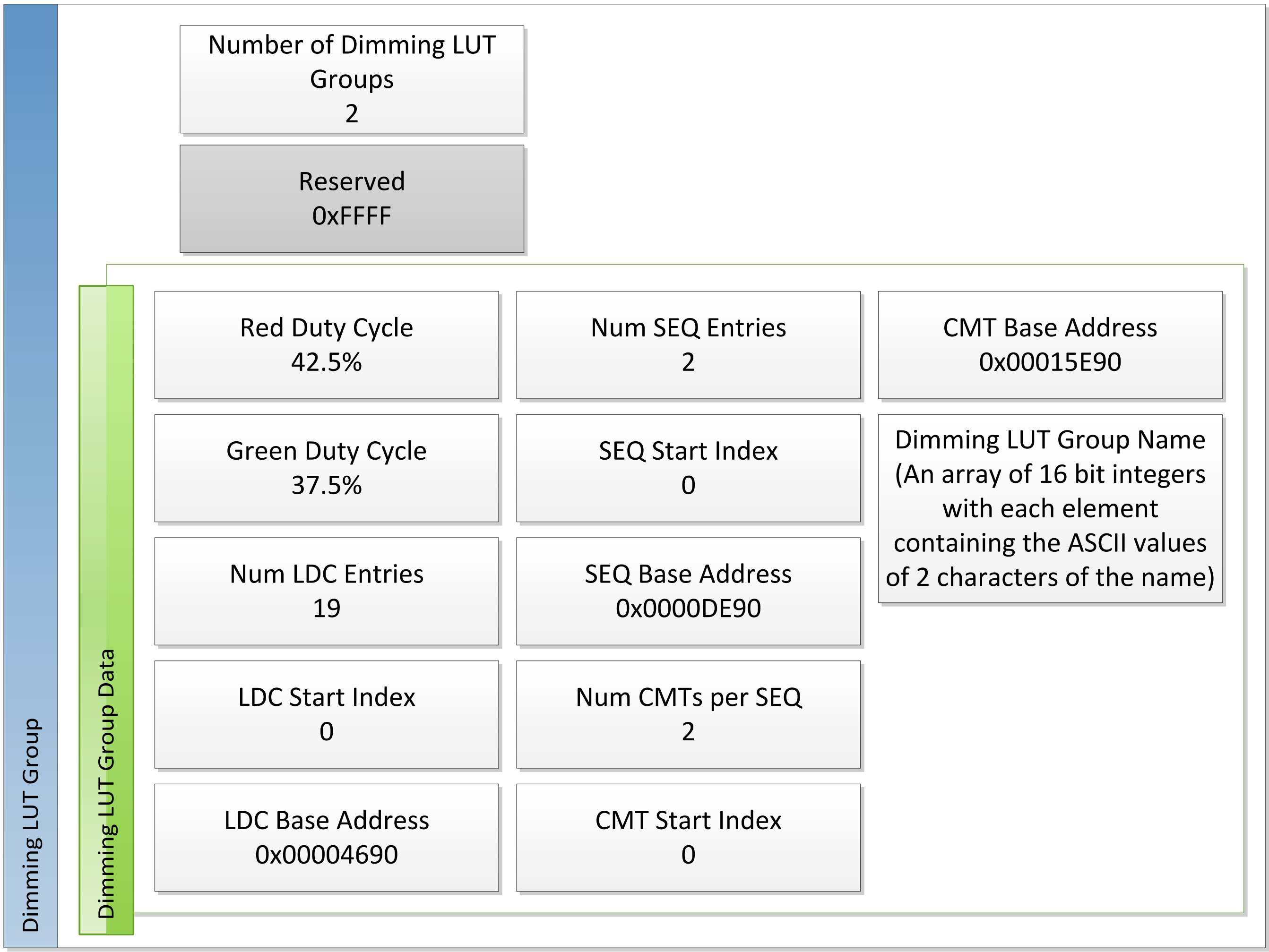


Figure 10-40. Dimming LUT Group Sample Data Explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct SequenceTable
{
    Uint16 NumSEQTables;
    Uint16 Reserved;
    struct SequenceTableData
    {
        float Red;
        float Grn;
        Uint16 NumLDCEntries;
        Uint16 LDCStartIndex;
        Uint32 LDCBaseAddr;
        Uint16 NumSEQEntries;
        Uint16 SEQStartIndex;
        Uint32 SEQBaseAddr;
        Uint16 NumCMTsPerSEQ;
        Uint16 CMTStartIndex;
        Uint32 CMTBaseAddr;
        Uint16 Name[(MAX_SEQ_TBL_NAME_LEN+1)/2];
    } SeqTbl[1];
} CAL_SEQTABLEINFO;
```

10.2.18 Section LDC-SEQ

The LDC-SEQ table contains a list of pairs of LDC Indexes and their associated Sequence Indexes for each Dimming LUT group. The LDC Start Index field for each Dimming LUT group points to the index number (starting from zero) on this table which marks the beginning of LDC-SEQ mapping data for that LUT group. Figure 10-42 shows the data corresponding to the 1st row of the LDC-SEQ table shown in the Example Configuration file. The Mode and LDC field has Mode as its MSB and LDC Index as its LSB. Value ‘0’ for mode indicates continuous mode and value ‘1’ indicates discontinuous mode.

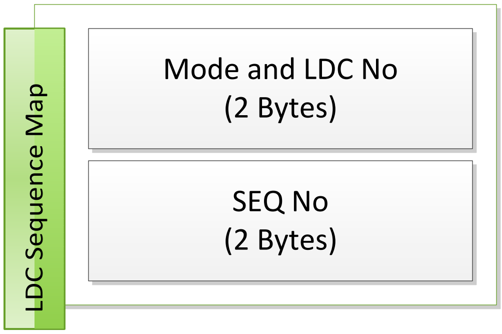


Figure 10-41. Configuration Data - Section LDC-SEQ

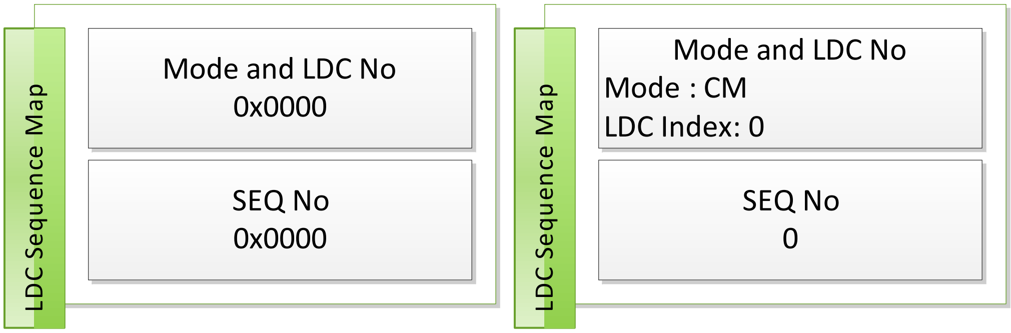


Figure 10-42. Configuration Data - Section LDC-SEQ: Sample Data with explanation

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct LDCSeqInfo
{
    Uint16 ModeLDCNo;
            Uint16 SEQNo;
} CAL_LDCSEQINFO;
```

10.2.19 Section SEQ-CMT

The SEQ-CMT table contains a list of pairs of Sequence Indexes, their associated CMT Start Indexes and the ADC sample times for Red, Green and Blue LED’s for each Dimming LUT group. The sample times are meant for future expansion. These are floating point numbers stored as per the IEEE 754 format (Refer to 17.3). The SEQ Start Index field for each Dimming LUT group maps points to the index number (starting from zero) on this table which marks the beginning of SEQ-CMT mapping data for that LUT group.

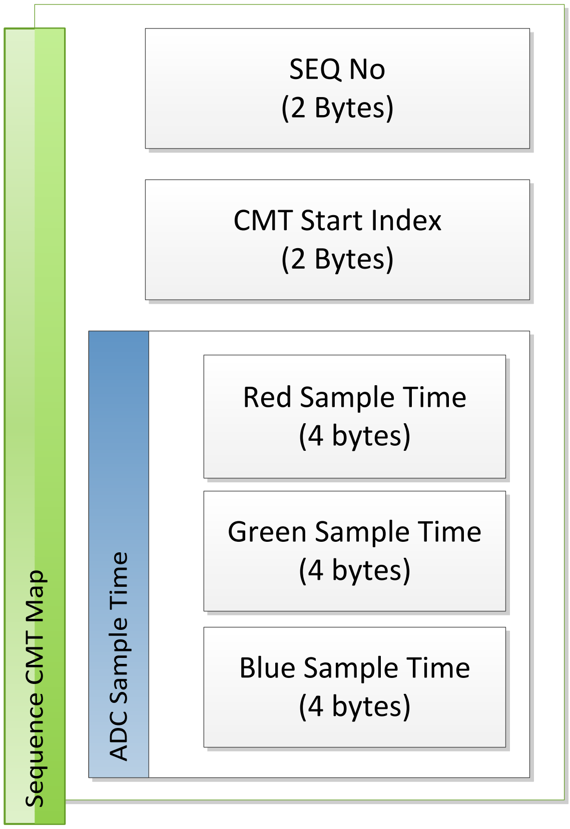


Figure 10-43. Configuration Data - Section SEQ-CMT

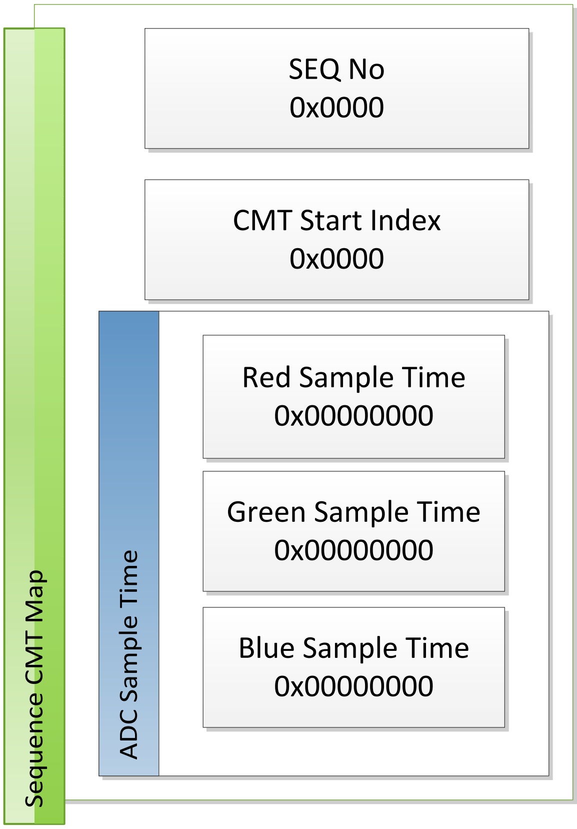


Figure 10-44. Configuration Data - Section SEQ-CMT: Sample Data

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct SeqCMTInfo
{
    Uint16 SEQNo;
    Uint16 CMTStartIndex;
    struct ADCSampleTimeInfo
          {
              float RedTime;
              float GrnTime;
              float BluTime;
          }ADCSampleTime;
} CAL_SEQCMTINFO;
```

10.2.20 Section CMT-Gamma

The CMT-Gamma table shows the gamma name for each CMT ID.

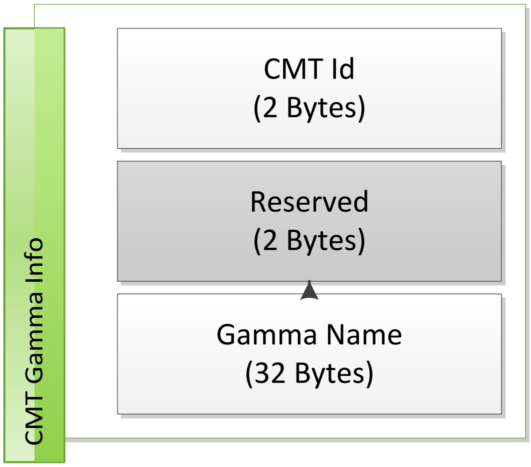


Figure 10-45. Calibration Data - Section CMT-Gamma

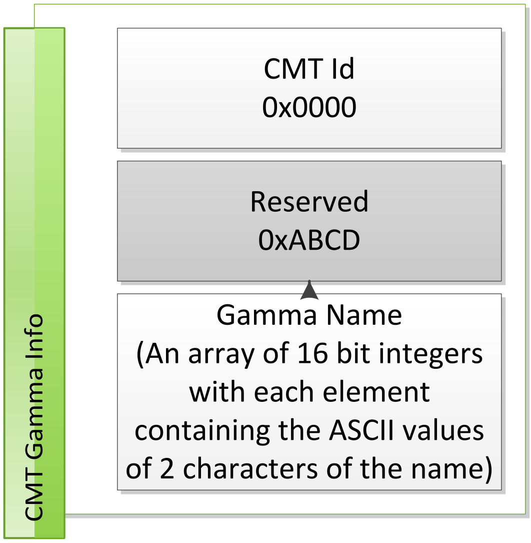

CMT Gamma Info
CMT Id
0x0000
Reserved
0xABCD
Gamma Name
(An array of 16 bit integers
with each element
containing the ASCII values
of 2 characters of the name)

Figure 10-46. Configuration Data- Section CMT-Gamma: Sample Data

This section is accessed in software through the following structure in calibration.h file in the software:

```
typedef struct CMTNameInfo
{
    Uint16 CMTNo;
    Uint16 Reserved;
    Uint16 Name[(MAX_GAMMA_NAME_LEN+1)/2];
} CAL_CMTNAMEINFO;
```

10.2.21 Extra Information

This section can be used to store any extra information in the Piccolo Software. This information is stored in the form of 32 bit key-value pairs. Given a key, its value can be retrieved through SPI commands. The key 0xB217D12F contains the address in the Piccolo where the ASIC Build Information is stored. The Automotive Control Program uses this value to read the build information. The structure of this section is shown below:

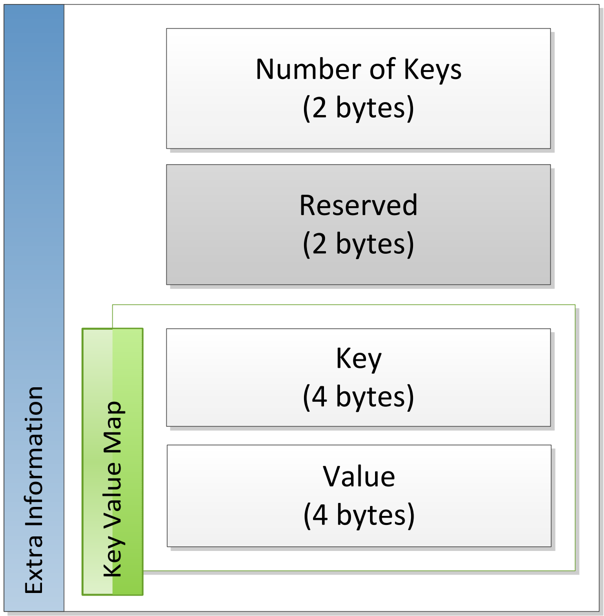


Figure 10-47. Configuration Data - Section Extra Information

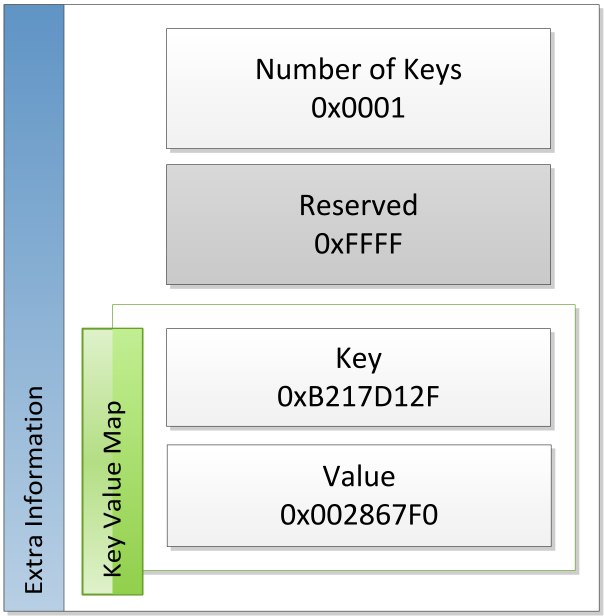


Figure 10-48. Configuration Data - Section Extra Information Sample data

This section is accessed in software through the following structure in calibration.h file in the software:

```
 typedef struct ExtraInformation
{
    Uint16 NumKeys;
    Uint16 Reserved;
       struct ExtraInfo
       {
              Uint32 Key; ;
              Uint32 Value; ;
          } Info[1];
} CAL_EXTRAINFO;
```

10.3 Power on Tests

The software performs several tests on the data stored in flash to ensure that a valid calibration file is present in the flash. These tests are repeated after a new calibration file is programmed via corresponding SPI command. The calibration and configuration defaults however, are applied only at power-up and not after a new calibration file is programmed. The software will not allow dimming without the presence of proper calibration data within flash.

The tests performed are:

1. Check if Signature is Valid
2. Compute the checksum of words following checksum field in File Header adding all words till the end of calibration data denoted by Length in words field. Verify that this checksum matches the stored value in checksum field.
3. Check if all sections listed in Figure 10-9 have data.

During the process of testing the software pointers are initialized to each section of the calibration data. This enables easily accessing the calibration data after power on.

10.4 Building the Calibration File

The calibration data to program to Piccolo flash is generated by PC software tool Automotive Control Program. The Calibration and Configuration Files are XML files that can be input to the Automotive Control Program. The Automotive Control Program performs verification on the received input calibration and configuration files, and on successful validation it packs the data as mentioned in Section 10.2. Following this, the packed data is send via SPI command to Piccolo to program the calibration data to internal flash memory.

Chapter 11

Master On/Off

11.1 Master On/Off

The Master On/Off feature can be used to turn on or turn off various software functionalities such as dimming and interrupt processing. The Master On/Off is set through an SPI command. When set to Off, the Piccolo instructs the ASIC to park the DMD and turn off all interrupt processing except for SPI. While the Piccolo is set to Master Off, some commands will be disabled. Once the system state is changed to On, all the above functionalities are enabled and the DMD is un-parked.

High Resolution Pulse Width Modulation (HRPWM)

12.1 High Resolution Pulse Width Modulation (HRPWM)

The Piccolo microcontroller operates at a clock frequency of 50 MHz or 60 MHz (depending on the device). When using PWM at a high frequency, the number of steps possible is limited by this clock period. As a result only a coarse control of the PWM duty cycle is possible, and this limitation results in low resolution brightness control.

In order to increase the resolution, the High Resolution PWM (HRPWM) feature of the microcontroller is utilized. The HRPWM uses the concept of Micro Edge Positioning (MEP), by which an edge can be positioned very finely by sub-dividing one coarse system clock of a conventional PWM generator. The time step accuracy is on the order of 150 ps. In short, the 50-MHz coarse clock which can only provide a time step accuracy of 20 ns is now divided into much finer pulses. Figure 12-1 illustrates HRPWM and various parameters associated with it. For more information on HRPWM, please see TMS320x2802x, 2803x Piccolo High Resolution Pulse Width Modulator (HRPWM) Reference Guide (SPRUGE8).

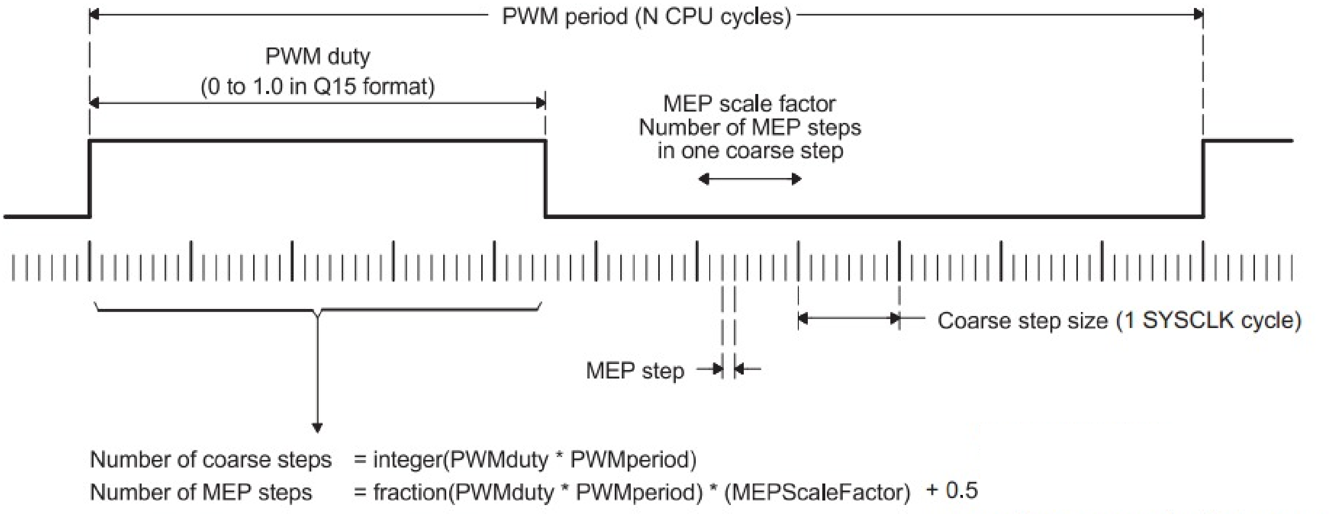

PWM period (N CPU cycles)
PWM duty
(0 to 1.0 in Q15 format)
MEP scale factor
Number of MEP steps
in one coarse step
Coarse step size (1 SYSCLK cycle)
MEP step
Number of coarse steps = integer(PWMduty * PWMperiod)
Number of MEP steps = fraction(PWMduty * PWMperiod) * (MEPScaleFactor) + 0.5

Figure 12-1. High Resolution PWM

12.2 HRPWM MEP Scale Factor

HRPWM Micro Edge Positioner (MEP) Scale Factor: The MEP scale factor is defined as the number of fine (HRPWM) pulses per one coarse (normal PWM) pulse. The higher the scale factor, the better the resolution that can be achieved.

12.3 Scale Factor Optimization (SFO) Function

The MEP Scale Factor however, is not a constant, and varies with operating conditions. TI provides an MEP scale factor optimizing (SFO) software C function, which uses the built in diagnostics in each HRPWM and returns the best scale factor for a given operating point. This SFO function is called in the main loop of the software as shown in Chapter 4.

At any instant the maximum resolution available can be determined by using the formula:

$$Max\ Resolution = PWM\ Period\ (in\ number\ of\ coarse\ system\ clocks) * MEP\ Scale\ Factor$$

The software supports SPI commands to vary the PWM Period and also receive the maximum resolution for the current period. Refer to the **DLP3030-Q1 Head-Up Display (HUD) Piccolo SPI User's Guide (DLPU057)** for more details.

Chapter 13

DLPU061–March 2018

Analog to Digital Converter (ADC) Setup

13.1 Analog to Digital Converter (ADC) Setup

The ADC API's in the Piccolo software allow simplified setup of ADC SOC's and channels. For details on various components (SOC (Start of Conversion configuration), trigger, channel, ADC interrupts etc) of the Piccolo ADC, refer to *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator* (SPRUGE5).

The software automatically manages ADC interrupt allocation. This has two advantages. First, the programmer does not have to be concerned with which interrupt to allocate to an ADC. Second, even if only one interrupt is free, any number of ADC's that require only polling can be used. At the time of polling the software allocates a free interrupt to the SOC, performs polling operations and then de-allocates the interrupt.

13.2 Guidelines for ADC setup

The `ADC_SetupSOC()` function is used to setup ADC for polling as well as interrupt based operations. This function takes five arguments: SOC number, channel, sample window length, trigger and interrupt routine. If only polling is required, NULL is to be passed for the interrupt routine argument. If an interrupt routine is required the function pointer should be passed.

Examples:

Polling:

```
ADC_SetupSOC(SOC04, ADCINA3, 8, 0, NULL);
```

The above line of code sets up SOC 4 with channel A3 as input channel and a sampling window length of 8 clocks. The conversion is software triggered (0 – Software triggered) and no interrupt routine is assigned.

Interrupt routine :

```
ADC_SetupSOC(SOC04, ADCINA3, 8, 0, &InterruptRoutine);
```

The above line of code sets up SOC 4 with channel A3 as input channel and a sampling window length of 8 clocks. The conversion is software triggered (0 – Software triggered) and calls the `InterruptRoutine()`function at the end of conversion. When using an interrupt routine, the software automatically sets up the PIE Interrupt registers required.

It is important to note that the `ADC_SetupSOC()`function overrides any previous settings for the concerned SOC. It is important that the programmer keeps track of the SOCs utilized in the software..

Note: The `ADC_SetupSOC()`function does not enable simultaneous sampling mode. This should be done separately by writing to the registers. For example:

```
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN14 = 1;
```

The above line of code enables simultaneous sampling mode for SOC 14 and SOC 15.

13.3 Measurement by polling

The `ADC_PolledMeasureSOC()` function should be used for polling an ADC channel. This function can only be used for SOCs that do not have an interrupt function setup.

13.4 Triggered conversions

For ADC channels which are associated with interrupt routines, polling is not permitted. For such SOC's, the conversion needs to be triggered using the `ADC_TriggerSOCMeasurement()` function. Once the conversion is complete, the routine will be called and the result can be obtained by using the `ADC_GETRESULT()` macro defined in adc.h. Please note that using the `ADC_GETRESULT()` before completion of conversion can result in wrong/inaccurate results.

The related C functions are named below:

- `Uint16 ADC_SetupSOC(Uint16 Soc, Uint16 Channel, Uint16 AcqPrescale, Uint16 Trigger, ADC_InterruptFn InterruptFn);`
- `Uint16 ADC_SetupSOC(Uint16 Soc, Uint16 Channel, Uint16 AcqPrescale, Uint16 Trigger, ADC_InterruptFn InterruptFn);`
- `int ADC_TriggerSOCMeasurement(Uint16 Soc);`

Chapter 14

DLPU061 – March 2018

Piccolo Voltage Supervision

14.1 Piccolo Voltage Supervision

The Piccolo can be made to monitor the voltages of the four power rails (1.2 V, 1.8 V, 2.5 V, 3.3 V). This is a feature that can be enabled or disabled through SPI commands. For this to work, the power rails need to be connected to the ADC Inputs A0, B3, B6 and B7 respectively. If these connections are not available, it is essential that the voltage supervision feature is disabled so that wrong voltages are not monitored. The ASIC is always reset at startup (the Mon-reset pin is held HIGH). If voltage monitoring is enabled in the calibration data stored in Piccolo flash then the power rail voltages will be measured at startup. If the power rail voltages are out of tolerance at this time then the Piccolo will not take the ASIC out of reset. Instead, the Piccolo regularly monitors the voltages and once the rails are back in tolerance, the mon-reset (GPIO 5) is set to LOW, bringing the ASIC out of reset. The software then reinitializes the ASIC. GPIO-5 must be connected to the reset pin of the ASIC for this feature to work correctly.

When voltage supervision (also called voltage monitoring) is enabled, the Piccolo monitors the voltages at the completion of every FSYNC processing or after a certain ‘Sampling Time’, whichever comes first. The Piccolo performs an interrupt based measurement rather than polling. In this approach, the ADC conversions are started by using the `VMON_StartVoltageMeasurements()`function in the `voltage_monitoring.c` file. The ADC conversions are done in a round-robin fashion. For more details on the ADC conversion process, refer to *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator* (SPRUGE5). Once the conversion is completed, the ADC interrupt causes the `VMON_ConvCompleteRoutine()` function to get called, which checks if the voltages are in tolerance.

When the ASIC is in reset state, the Piccolo stops responding to all SPI commands that do not have permission to be executed in this state.

If this feature is enabled at startup, the Piccolo performs ASIC initialization only if the voltages are within tolerance.

Even when voltage monitoring is disabled, the voltages can be manually read by using the `VMON_PolledVoltageMeasure()`function.

The related C functions are named below:

- `Uint16 VMON_Init(void)`
- `void VMON_UpdateVoltageStatus(void)`
- `void VMON_ProcessPowerRailStatus(void)`
- `Uint16 VMON_PolledVoltageMeasure(Uint16 *Voltage1p2, Uint16 *Voltage1p8, Uint16 *Voltage2p5, Uint16 *Voltage3p3)`
- `void VMON_StartVoltageMeasurements(void)`
- `Uint16 VMON_MeasurePowerRailVoltage_1p2(Uint16 *DigVal);`
- `Uint16 VMON_MeasurePowerRailVoltage_1p8(Uint16 *DigVal);`
- `Uint16 VMON_MeasurePowerRailVoltage_2p5(Uint16 *DigVal);`
- `Uint16 VMON_MeasurePowerRailVoltage_3p3(Uint16 *DigVal);`

DLPC120 ASIC Features Supported

15.1 Bezel Adjustment

The Bezel Offset sets the location of the first pixel of image data in any display mode. The Piccolo accepts the magnitude and sign (positive or negative) of the horizontal and vertical offsets through SPI, and sets the appropriate registers to perform bezel adjustment. The Horizontal offset is rounded to the closest multiple of 4 and the vertical offset to the nearest multiple of 2 and applied. For more details, refer to the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)** When in splash mode, the Piccolo constantly monitors the bezel offset, and reloads the splash if there is any change in the values. Please note that this reload takes place only when the Splash mode is set through the Piccolo, and not when done by writing directly to registers or through Pass/Fail command lists for External Video BIST.

The related C functions are:

- `Uint16 DLPC120_SetBezelOffset(Uint16 Horiz, BOOL HorizSign, Uint16 Vert, BOOL VertSign);`
- `Uint16 DLPC120_GetBezelOffset(Uint16 *HorizPix, BOOL *HorizSign, Uint16 *VertPix, BOOL *VertSign);`

15.2 DMD Drive Strength

The Piccolo can set the DMD Drive Strength to 6 mA, 10 mA or 12 mA. For more details, refer to the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**

The related C functions are:

- `Uint16 DLPC120_SetDMDDriveStrength(Uint16 DriveStrength);`
- `Uint16 DLPC120_GetDMDDriveStrength(Uint16 *DriveStrength);`

15.3 Front End Video Built In Self Test (BIST)

This test generates a checksum result using the data read directly from an external video source. The checksum can be compared from system to system to verify the video source. A static image must be displayed from an external video source in order to generate a consistent checksum on each system. The Piccolo performs a sequence of I2C register writes to execute the BIST and stores the checksum. The BIST is then disabled. For more details on the register writes refer to the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**. The coordinates of the start and end pixels of the region for which the checksum is to be computed are accepted as input. This BIST is not performed on startup and needs to be executed through SPI commands.

The related C functions is:

- `Uint16 DLPC120_FrontEndVideoBIST();`

15.4 External Video Detect Built In Self Test

A video detect test can be run to confirm that the external Vsync is within a specified range. Additionally, this test verifies that there is a nonzero pixel clock, input lines, and pixels per line.

The I2C write sequence for this BIST is explained in detail in the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**Like the Front End Video BIST this BIST can be executed through SPI commands. However the software also executes this BIST whenever an external video command list is executed (See Section 15.4.1). The user must disable the BIST before running any other BIST with different settings. The Piccolo can run the External Video BIST in two ways:

1. Continuous Enable Mode: In this mode the BIST is enabled and allowed to run continuously until disabled by the user. Addresses of Command Lists that will be executed on pass or fail are taken in as inputs. For the BIST to run in this mode, it is necessary that command list execution on Pass/Fail is enabled.
2. Update Status Mode: In this mode the BIST is executed 4 times with different mux select values so as to receive reports for Vsync, Pixel Clock, Active Lines, and Active Pixels per Line. The results are stored and can be read through SPI. Command List execution on Pass/Fail need not be enabled for this mode of execution.

Please note that if a splash command list is set as one of the Pass/Fail command lists and gets executed, the splash will **not** get reloaded on change in CMT Index/Bezel Offset.

The related C functions are:

-
  ```
  Uint16 DLPC120_ExternalVideoDetectBISTSetup(BOOL UpdateResult, BOOL CmdListEnable,
                                              Uint32 CmdListPass, Uint32 CmdListFail);
  ```

-
  ```
  Uint16 DLPC120_ExternalVideoDetectBIST(BOOL KeepEnabled, Uint16 MuxSelect,
                                            BOOL CmdListEnable, DLPC120_BIST_RESULT* Result,
  Uint32 *MuxReport);
  ```

15.4.1 Software Owned External Video Detect BIST

The Piccolo software runs the External Video BIST whenever an external video command list is executed. The second available test pattern (assumed to be a black test pattern) is set as the FAIL command list and the executed command list as the PASS command list. When the display mode is changed to splash or test pattern, the BIST is disabled and re-enabled only when it is changed back to external video. This feature can be disabled by initializing the `DLPC120_SWOwnedEVDBISTEnableState` variable to ‘FALSE’ in DLPC120.c. After applying startup settings and executing all the default command lists, if there were no default display mode command lists, the software assumes the system is in external video mode and enables the BIST with the first available 864 × 480 command list. The BIST is not enabled if none are available.

The related C functions are:

- `Uint16    DLPC120_RunSWOwnedEVDBIST(void);`
- `void    DLPC120_DisableCurtainandRunSWOwnedEVDBIST (void);`
- `void    DLPC120_SetupSWOwnedEVDBISTCmdLists(Uint32 PassAddress, Uint32 FailAddress);`

Texas Instruments

Chapter 16

DLPU061–March 2018

DMD Parking

16.1 DMD Parking

The DMD has a parking specification that is described in the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**

The Piccolo software executes parking as described in the **I2C Programmer's Guide for DLPC120 ASIC (DLPU055)**using the `DLPC120_DMDPark` function in `DLPC120.c`. This function can un-park or park the DMD. When the function is called, the park enable or disabled and park owner must be specified.

The park owner is the process that requested the park. The DLPC120_ParkOwners variable specifies which processes have requested a park. The park owners are listed below:

- `PARK_VMON` = park called by ADC voltage monitoring
- `PARK_SPI` = park called by SPI park command
- `PARK_BOOTLOADER` = park called before switching to bootloader
- `PARK_MASTER_OFF` = park called by MasterOn/Off SPI command

If no other park owner has already executed a park, the parking sequence is executed according to the specification in **I^{2}C Programmer’s Guide for DLPC120 ASIC.**

When un-park is requested, the park owner is cleared. To un-park the DMD, all owners must be cleared.

Appendix A

Appendix

A.1 K10 Format

This is a custom format used by the software to store the temperature values. In this format, the temperatures in Celsius scale is converted to Kelvin and then multiplied by 10. For example, 35°C in K10 format is 3080. This format is used so that any negative and floating point temperatures in Celsius scale gets converted to a positive 16-bit integer.

A.2 Little Endian Representation

Piccolo is a 16-bit MCU and the minimum addressable data is 16-bit. In other words, it does not understand bytes, but only 16-bit words. Each address it can access contains 1 word data. So, as per little endian definition, the lower addressable data should come first before the higher addressable data. This means, the data stored in little endian format will be reversed by word length and not by byte length.

For example, the Signature field to identify Calibration Data uses a value 0x464C4143 that represents the ASCII for letters CALF. This value when stored in little endian format in Piccolo will change to 0x4143 followed by 0x464C. Note that the reversal only occurred at the word boundary.

A.3 Floating Point IEEE 754 Format

Piccolo uses IEEE-754 format to represent single precision floating point numbers. Such a representation takes 4 bytes (32-bits) of data for storage. The 32-bits is divided into 1-bit sign, 8-bit exponent and 23-bit mantissa.

The bit layout is as shown below:

```
                                        meaning
seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm
31                              0       bit #
s       = sign bit, e = exponent, m = mantissa
```

In general,

$$floating\ point\ number = (sign\ ?\ -1{:}\ 1) * 2^{(exponent-127)} * 1.(mantissa\ bits)$$

Please refer to following link for more details on IEEE-754 format:

www.cprogramming.com/tutorial/floating_point/understanding_floating_point_representation.html

Floating point data is stored in calibration file in the same format, so that Piccolo can directly access the memory location using a float pointer. As an example, the duty cycle of Red and Green is stored in calibration data in IEEE-754 format.

Consider the number 0x42140000 in IEEE-754 format.

Sign (bit 31) = 0

Exponent (Bits 30:23) = 1000 0100 (binary) = 0x84 = 132

Mantissa (Bits 22:0) = 0x140000 = 0010 1000 0000 0000 (binary)

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated (‘TI”) technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, “TI Resources”) are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI’s provision of TI Resources does not expand or otherwise alter TI’s applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED “AS IS” AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI’s standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265