
MT9D111 Developer Guide

1/3.2-Inch, 2-Megapixel SOC CMOS Digital Image Sensor

About this Guide

The MT9D111 is a complete system-on-a-chip solution. It incorporates sophisticated, on-chip camera functions and is programmable through a simple two-wire serial interface. The developer guide is a thorough reference for engineers who wish to develop applications for the MT9D111. The guide provides a detailed information on working with chip registers and variables and explains how to use Aptina's developer software—DevWare. The MT9D111 data sheet should be used along with this guide as a reference for specific register and programming information.

Table of Contents

Two-Wire Serial Interface	9
Overview	9
Example: 16 Bit Register Read	9
Example: 16 Bit Register Write	10
Accessing the Firmware Drivers' Variables	11
Initializing the MT9D111	13
Power-up Sequence	13
Hard Reset Sequence	13
Soft Reset Sequence	13
Standby Sequence	13
PLL Setup	15
Identifying Chip Version	15
Initializing FAQs	16
Context Switching and Output Configuration	17
Context Switch and Setup	17
Changing the Output Resolution	17
Selecting Output Data Formats	17
Raw Bayer Data Output	18
Output Format and Timing	18
Decimation, Zoom, and Pan	22
Enabling Special Effects	23
Mirroring the Image	24
Column and Row Skip	24
Binning	25
Configuring Pad Slew	25
Capturing Still Pictures	25
Capturing Videos	26
Enabling and Capturing JPEG	27
Switching Between JPEG 4:2:2, 4:2:0, and Monochrome	27
Context Switching and Output Configuration FAQs	28
Gamma and Contrast	30
Gamma	30
Contrast	31
Gamma and Contrast FAQs	32
Lens Shading and Correction	33
Introduction	33
Lens Shading Approach	33
Setup	34
Preset and Load	35
Setup Conditions	35
Calibration	38
Result	41
Verification	42
Related Register List	43
Lens Shading and Correction FAQs	45
Auto Exposure	46
Overview	46
Preview Mode	46
Scene Evaluative Mode	46
AE Sport Mode	46
How to Calibrate the AE Exposure Value (EV) Reference	47

How to Modify the Image Brightness.	47
How to Speed Up and Slow Down AE Adjustments	47
How to Maintain Specific Frame Rates	48
How to Use Manual Exposure and Manual Gain	49
Auto Exposure FAQs	50
Flicker Avoidance	51
Background	51
How to Use the Flicker Detection Driver	51
How to Fine Tune the Anti-flick Driver Setting	53
How to Verify the Setting	53
How to Modify the Setting for Specific Applications	54
Flicker Avoidance FAQs	55
Color Correction	56
Auto White Balance	56
How to Change the Color Saturation.	56
How to Speed Up/Slow Down AWB.	57
How to use a Static CCM.	57
How to Perform Color Calibration	58
Related Register List.	68
Color Correction FAQs	69
Auto Focus Driver	70
Background	70
Scan Auto Focus Algorithm	70
Evaluation of Image Sharpness.	76
Algorithm Flowchart	82
Creep Compensation.	84
Public Variables of AF Driver	84
Public Functions of AF Driver and Corresponding VMT Pointers.	87
Auto Focus Driver FAQs.	92
Auto Focus Mechanism	94
Introduction	94
HD80 (I2C)	94
AD5398 (I2C)	95
LB1935T/CL (Stepper).	97
Interface Protocol Waveform.	97
LV8071LP (Piezo)	99
Interface Protocol Waveform.	99
MD115/ID9701 (PWM)	101
Interface Protocol Waveform.	101
Configuration	101
ID9701(PWM)	102
Lens Actuator Control	103
Managing Lens Actuator Hysteresis.	105
Timer.	106
Serial Interface	112
Initial Positioning of Stepper Motors.	113
Public Functions of the AFM Driver and Corresponding VMT Pointers	127
Currently Supported AFM Mechanics	130
Mode Driver-Setting up Preview (A) and Capture (B) Modes	131
MT9D111 Register Wizard	131
Procedure.	131
MT9D111 Developer Guide Mode Driver Preview and Driver FAQs	137
How to Set Up the Histogram Driver Variable for Operation.	138

Flash Strobe, Mechanical Shutter, and Global Reset.	139
Still Capture using Xenon/LED Flash with User-defined Image Quality Settings	139
Still Capture using LED Flash with Automatic White Balance and Exposure Control	140
Flash Strobe, Mechanical Shutter, and Global Reset FAQs	142
GPIOs	143
Programming GPIO Outputs	143
Reading GPIO Inputs	143
Outputting Flash and/or Strobe from GPIO	143
Waveform Generator Programming Example	143
GPIO FAQs	145
Using the Test Patterns	146
Disabling All Firmware Drivers	146
JPEG Functionality	147
How to Enable/Disable the JPEG Output	147
How to Set the JPEG Color Format	147
How to Set the Restart Marker Interval	147
How to Get the JPEG Status	147
How to Get the JPEG Data Length	148
How to Handle the JPEG Errors.	148
How to Read/Write the JPEG Quantization/Huffman Table Memories	148
How to Program the Quantization Table	148
How to Translate between Qscale and Quality Factor.	151
How to Program the Customized Huffman Table.	151
How to Append the JPEG Header	153
Sample C Code	153
JPEG Power Saving.	160
JPEG Functionality FAQs	161
Appendix A—How to Update Demo2 Firmware	163
Appendix B—Miscellaneous FAQs	164
Appendix C—Glossary of Terms	169
Revision History.	171

List of Figures

Figure 1:	Register Legend	8
Figure 2:	Firmware Variable Legend	8
Figure 3:	Example of 16-Bit Register Read from Chip Version Register (Reg0x00:0), Value=0x1519	10
Figure 4:	Example of 16-Bit Register Write to Register (Reg0x20:1), Value=0xA5F0	11
Figure 5:	PLL Setting Change Flow Chart	16
Figure 6:	Timing of Decimated Uncompressed Output Bypassing the FIFO	19
Figure 7:	Timing of Uncompressed Full Frame or Decimated Output Passing through the FIFO	19
Figure 8:	Example of Timing for Non-Decimated Uncompressed Output Bypassing Output FIFO	19
Figure 9:	Timing of JPEG Compressed Output in Free-Running Clock Mode	21
Figure 10:	Timing of JPEG Compressed Output in Gated Clock Mode	22
Figure 11:	Timing of JPEG Compressed Output in Spoof Mode	22
Figure 12:	Gamma Correction Curve	30
Figure 13:	Signal	33
Figure 14:	Lens Correction Zones	34
Figure 15:	DevWare Toolbar	34
Figure 16:	Presets Dialog Box	35
Figure 17:	Setting Gamma to 1.0.	36
Figure 18:	Enable Auto Exposure	37
Figure 19:	Disable AWB and Color Correction	38
Figure 20:	Setting the Row and Column Line	39
Figure 21:	Sensor Control Dialog Box Showing Lens Correction Settings.	40
Figure 22:	Settled Analysis Graph	40
Figure 23:	Adjusting K Factor	41
Figure 24:	Correlation Between Percentage and Curvature	41
Figure 25:	Lens Correction Result, Before and After	42
Figure 26:	Intensity Graph (horizontal) Before and After	42
Figure 27:	Intensity Graph (vertical) Before and After	42
Figure 28:	Locating the Lens Correction.ini File	43
Figure 29:	Block Diagram of a Basic AF Camera Built Around the MT9D111 Image Sensor.	71
Figure 30:	Search for Best Focus	74
Figure 31:	Scene with Two Potential Focus Targets at Different Distances From Camera	76
Figure 32:	Dependence of Luminance-Normalized Local Sharpness Scores on Lens Position	76
Figure 33:	Example of Position Weight Histogram Created by AF Driver	77
Figure 34:	Auto Focus Windows	79
Figure 35:	Computation of Sharpness Scores and Luminance Average for AN AF Window	80
Figure 36:	Flowchart of Scan AF Algorithm Implemented in the MT9D111	83
Figure 37:	Flowchart of Scan AF Algorithm Implemented in the MT9D111	84
Figure 38:	HD80C Complete WRITE	96
Figure 39:	Configuration Schematic	96
Figure 40:	Timing Diagram	97
Figure 41:	Configuration Schematic	98
Figure 42:	Timing Diagram	99
Figure 43:	Configuration Schematic	99
Figure 44:	Timing Diagram	101
Figure 45:	Configuration Schematic	102
Figure 46:	Timing Diagram	103
Figure 47:	Configuration Schematic	103
Figure 48:	Configuration Schematic	104
Figure 49:	Example of Hysteresis-affected Relation Between Physical and Logical Lens Position	106
Figure 50:	Hysteresis Loop Typical for Simple Mechanical Gears	108
Figure 51:	Time Needed to Increase Voltage on Helimorph by 10V as a Function of Lens Position	113
Figure 52:	Piecewise Linear Function Used by AFM Driver to Estimate Lens Travel Time	115
Figure 53:	Typical Relation Between Photointerrupter Output Signal and Lens Position.	118
Figure 54:	Lens Movements During Initial Positioning of a Stepper Motor (Example 1)	119
Figure 55:	Lens Movements During Initial Positioning of a Stepper Motor (Example 2)	120
Figure 56:	Lens Movements During Initial Positioning of a Stepper Motor (Example 3)	121

Figure 57:	Flowchart of AFM Driver Function Used in Initial Positioning of Stepper Motors (Page 1)	123
Figure 58:	Flowchart of AFM Driver Function Used in Initial Positioning of Stepper Motors (Page 2)	126
Figure 59:	Input Clock and PLL Output Frequencies	138
Figure 60:	Image Timing Section	140
Figure 61:	State Parameters Tab	141
Figure 62:	State Diagram and Transitions of the MT9D111	142
Figure 63:	Gamma and Contrast Tab.	143
Figure 64:	Register Output Tab	144
Figure 65:	LED Flash Timing Diagram	148
Figure 66:	Xenon Flash Timing Diagram	149
Figure 67:	LED Flash Timing Diagram with Automatic Exposure and White Balance	150
Figure 68:	Chief Ray Angle Requirement for 2MP MT9D111	176

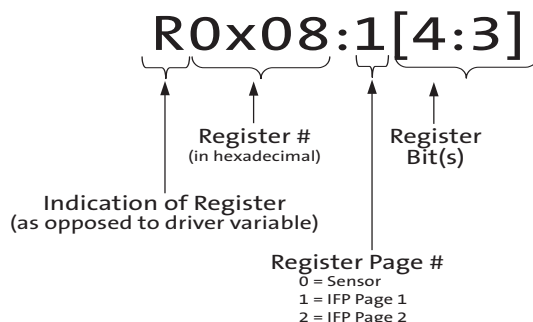
List of Tables

Table 1:	Selecting Values in the Bus Address (Read/Write)	9
Table 2:	Register Page Description	9
Table 3:	Reg0x00C6:1[15:0], Indirect Access Address Register	11
Table 4:	Reg0x001E:2[15:0], JPEG Indirect Access Control Register	12
Table 5:	Frequency Parameters	15
Table 6:	Changing Output Format Variables	17
Table 7:	Output Format Option Configuration Settings	18
Table 8:	YCrCb Output Data Ordering	20
Table 9:	RGB Ordering in Default Mode	20
Table 10:	Enabling Special Effects	24
Table 11:	Driver-Variables Auto Exposure Driver (ID = 2)	72
Table 12:	Possible AF Filters	79
Table 13:	Public Variables of the Auto Focus Driver	85
Table 14:	Actuators Supported	94
Table 15:	Configuration List	95
Table 16:	Configuration List	96
Table 17:	Truth Table	97
Table 18:	Configuration List 1	97
Table 19:	Configuration List 2	98
Table 20:	Configuration List 1	99
Table 21:	Configuration List 2	99
Table 22:	Configuration List 1	101
Table 23:	Configuration List	102
Table 24:	Programmable Parameters of Stepper Motor Positioning Function	119
Table 25:	Public Variables of the AFM Driver	122
Table 26:	AFM Mechanics Supported	130
Table 27:	PLL Specifications	132
Table 28:	Luminance Quantization	149
Table 29:	Chrominance Quantization	149
Table 30:	Quantization Address Map	150
Table 31:	Huffman Memory Map	152
Table 32:	Structure of Huffman Code in Huffman Memory	152
Table 33:	Location of AC Huffman Codes in Huffman Memory	152
Table 34:	Location of DC Huffman Codes in Huffman Memory	153
Table 35:	Glossary of Terms	169

Introduction to Registers

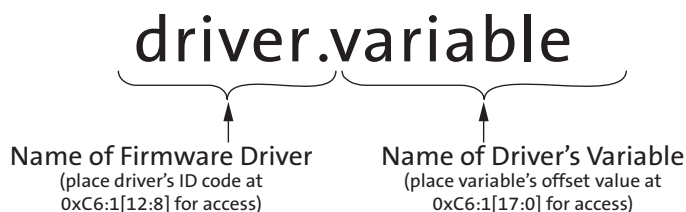
This developer guide refers to various memory locations and registers that the user reads from or writes to for altering the MT9D111 operation. Hardware registers appear as follows and may be read or written by sending the address and data information over the two-wire serial interface.

Figure 1: Register Legend



Other memory locations are within the microcontroller block and may be accessed by utilizing hardware registers from 0xC6:1 through 0xD1:1 (see the MT9D111 data sheet for further details on how to use these registers). These are denoted below:

Figure 2: Firmware Variable Legend



The MT9D111 was designed to facilitate customizations to optimize image quality processing. As the image data travels through the various stages of image processing, the user can adjust the parameters in these stages to affect the images' appearances. This section describes most of these available adjustments.

Two-Wire Serial Interface

Overview

The only external control interface to the MT9D111 is a two-wire serial interface. This chapter shows how to access the MT9D111 registers. For the complete specification, refer to the MT9D111 data sheet.

The MT9D111 contains 3 pages of registers as well as the firmware-driver variables. Each page has 256 address locations, and each location is 16 bits wide. Not all locations and bits are accessible (refer to the register table for detailed information on each register). Included in these three pages is the indirect access for MCU (drivers) and JPEG memory.

The bus address of the two-wire serial interface is selectable between two sets of values. Changing the state of the hardware pin, SADDR, or of R0x0D:0[10] selects between them as follows:

Table 1: Selecting Values in the Bus Address (Read/Write)

	SADDR Set High	SADDR Set Low
R0x0D:0[10] = 0 (default)	0xBB/0xBA	0x91/0x90
R0x0D:0[10] = 1	0x91/0x90	0xBB/0xBA

Table 2: Register Page Description

Page	
Page 0: Sensor	Sensor and PLL control.
Page 1: SOC 1	SOC Image processing, and MCU register, MCU memory indirect access.
Page 2: SOC 2	JPEG control and Soc control. JPEG indirect memory access.

Page Selection Register: R0xF0:0

Register 0xF0:0 is a unique register; it is used to select which of the 3 pages are active when reading or writing. Physically there is only one register—regardless of which page is selected it accesses the same content.

Reg 0xF0:0 = 0x0000 => Page 0, Sensor

Reg 0xF0:0 = 0x0001 => Page 1, SOC 1

Reg 0xF0:0 = 0x0002 => Page 2, SOC 2

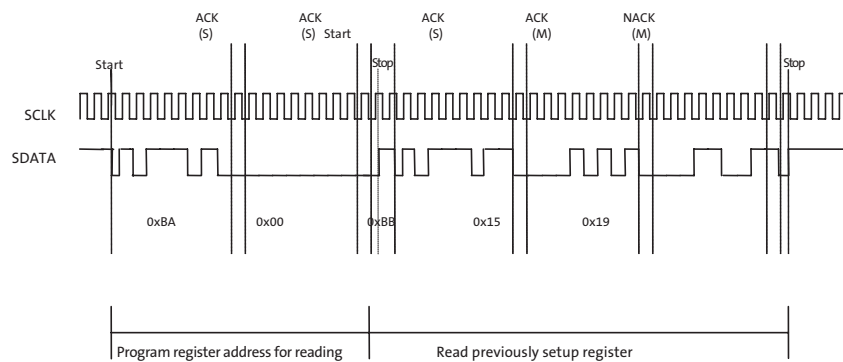
Example: 16 Bit Register Read

This is an example of a 16 Bit Register read from Chip Version register (Reg0x00:0), expected value = 0x1519

1. Send Start
2. Send Device Address
 - a. 0xBA
3. Wait for ack
4. Send register address (8-bit)
 - a. 0x00
5. Wait for ack
6. Send stop
7. Send start

8. Send device address for read
 - a. 0xBB
9. Wait for ack
10. Slave device sends 8-bit data (MSB byte)
 - a. 0x15
11. Master send ack
12. Slave device sends another 8-bit data (LSB byte)
 - a. 0x19
13. Master send nack
14. Send stop

Figure 3: Example of 16-Bit Register Read from Chip Version Register (Reg0x00:0), Value=0x1519



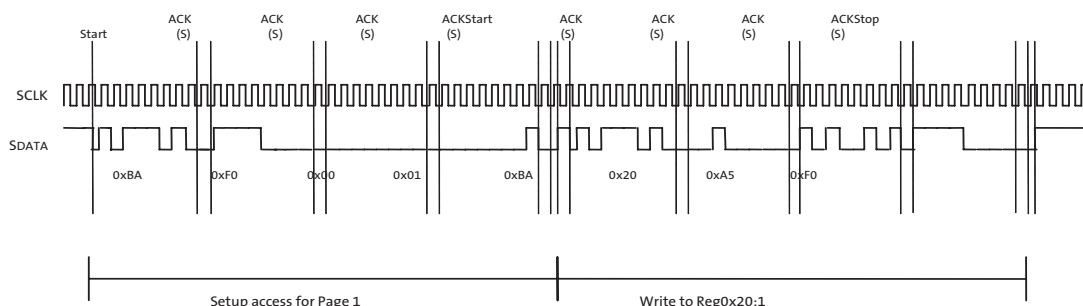
Example: 16 Bit Register Write

This is an example of a 16 bit register write to register (Reg0x20:1), value = 0xA5f0

1. Send Start
2. Send Device Address
 - a. 0xBA
3. Wait for ACK
4. Send register address (8-bit)
 - a. 0xF0
5. Wait for ACK
6. Send 8-bit data (MSB byte)
 - a. 0x00
7. Wait for ACK
8. Send another 8-bit data (LSB byte)
 - a. 0x01
9. Wait for ACK
10. Send stop
11. Send Start
12. Send Device Address
 - a. 0xBA
13. Wait for ACK
14. Send register address (8-bit)
 - a. 0x20

15. Wait for ack
16. Send 8 bit data (MSB byte)
 - a. 0xA5
17. Wait for ack
18. Send another 8-bit data (LSB byte)
 - a. 0xF0
19. Wait for ack
20. Send stop

Figure 4: Example of 16-Bit Register Write to Register (Reg0x20:1), Value=0xA5F0



Accessing the Firmware Drivers' Variables

One register (R198:1) is used for the memory address and another (R200:1) is used for data in the address.

Write Access

A write to the indirect access data register triggers a write to the targeted memory after the two-wired serial interface has completed the WRITE cycle.

Read Access

Data is pre-fetched once the indirect access address register is updated; therefore, when user reads from the indirect access data register, the data is available.

MCU Memory

Reg0x00C6:1[15:0], Indirect Access Address Register

Reg0x00C8:1[15:0], Indirect Access Data Register

Table 3: Reg0x00C6:1[15:0], Indirect Access Address Register

Bit	
7:0	Bits 7:0 of address for physical access; driver variable offset for logical access.
12:8	Bits 12:8 of address for physical access; driver ID for logical access.
14:13	Bits 14:13 of address for physical access; R198:1[14:13] = 01 select logical access.
15	1 = 8-bit access; 0 = 16-bit

JPEG Memory

Reg0x001E:2[15:0], Indirect Access Address Register

Reg0x001F:2[15:0], Indirect Access Data Register

Table 4: Reg0x001E:2[15:0], JPEG Indirect Access Control Register

Bit	
10:0	Indirect access address register: This 11-bit register contains the address of the register or memory to be accessed indirectly.
12:11	Unused.
13	Enable two-wire serial interface burst: When this bit is set, the two-wire serial interface decoder operates in burst mode for the indirect data register (READ burst and WRITE burst). The longest burst supported is 16 (128 READ or WRITE cycles).
14	Enable indirect writing: When set, data from the indirect data register is written to the Indirect address location specified by [10:0] of this register except when auto-increment is set. Reading the same address location when this bit is reset to "0."
15	Address auto-increment: When this bit is set, the value in the indirect access address register is automatically incremented after every read or write, to the JPEG indirect access data register. This feature is used to emulate a burst access to memory or registers being accessed indirectly.

Initializing the MT9D111

Power-up Sequence

There are no specific requirements to the order in which different supplies are turned on. Once the last supply is stable within the valid ranges specified below, follow the hard reset sequence to complete the power-up sequence.

- Analog Voltage: 2.8V for best image performance
- Digital Voltage: 1.8V \pm 0.1V (1.7V–1.9V)
- I/O Voltage: 1.7V–3.1V

Hard Reset Sequence

After power-up, a hard reset is required. Assuming all supplies are stable, the assertion of RESET# (active LOW) will set the device in reset mode. The clock is required to be active when RESET# is released. Hence, leaving the input clock running during the reset duration is recommended. After 24 clock cycles (CLKIN), the two-wire serial interface is ready to accept commands on the two-wire serial interface.

Note: Reset should not be activated while STANDBY is asserted.

To activate a hard reset sequence to the camera:

1. Wait for all supplies to be stable
2. Assert RESET# (active LOW) for at least 1 μ s
3. De-assert RESET# (input clock must be running)
4. Wait 24 clock cycles before using the two-wire serial interface

Soft Reset Sequence

To activate a soft reset to the camera:

1. Bypass the PLL, R0x65:0=0xA000, if it is currently used
2. Perform MCU reset by setting R0xC3:1=0x0501
3. Enable soft reset by setting R0x0D:0=0x0021. Bit 0 is used for the sensor core reset while bit 5 refers to SOC reset.
4. Disable soft reset by setting R0x0D:0=0x0000
5. Wait 24 clock cycles before using the two-wire serial interface

Note: No access to the MT9D111 registers—both page 1 and page 2—is possible during soft reset.

Standby Sequence

Standby mode can be activated by two methods:

1. The first method is to assert STANDBY, which places the chip into hard standby. Turning off the input clock (CLKIN) reduces the standby power consumption to the maximum specification of 100 μ A at 55°C. There is no serial interface access for hard standby.
2. The second method is activated through the serial interface by setting R13:0[2]=1 known as the soft standby. As long as the input clock remains on, the chip will allow access through the serial interface in soft standby.

Standby should only be activated from the preview mode (context A), and not the capture mode (context B). In addition, the PLL state (off/bypassed/activated) is recorded at the time of firmware standby (seq.cmd=3) and restored once the camera is out of firmware standby. In both hard and soft standby scenarios, internal clocks are turned off and the analog circuitry is put into a low power state. Exit from standby must go through the same interface as entry to standby. If the input clock is turned off, the clock must be restarted before leaving standby.

To Enter Standby

1. To prepare for standby:
 - a. Issue the STANDBY command to the firmware by setting seq.cmd=3
 - b. Poll seq.state until the current state is in standby (seq.state=9)
 - c. Bypass the PLL if used by setting R101:0[15]=1
2. To prevent additional leakage current during standby:
 - a. Set R10:1[7]=1 to prevent elevated standby current. It will control the bidirectional pads DOUT, LINE_VALID, FRAME_VALID, PIXCLK, and GPIO.
 - b. If the outputs are allowed to be left in an unknown state while in standby, the current can increase. Therefore, either have the receiver hold the camera outputs HIGH or LOW, or allow the camera to drive its outputs to a known state by setting R13:0[6]=1. R13:0[4] needs to remain at the default value of "0." In this case, some pads will be HIGH while some will be LOW. For dual camera systems, at least one camera has to be driving the bus at any time so that the outputs will not be left floating.
 - c. For each GPIO that is left floating (which are set as inputs by default), configure as outputs and drive LOW by the setting the respective bit to "0" in the GPIO variables 0x1078, 0x1079, 0x1070, and 0x1071 (accessed via R198:1 and R200:1). For example, if all GPIOs are floating inputs, the following settings can be used:
 - i. R198:1=0x9078
 - ii. R200:1=0x0000
 - iii. R198:1=0x9079
 - iv. R200:1=0x0000
 - v. R198:1=0x9070
 - vi. R200:1=0x0000
 - vii. R198:1=0x9071
 - viii. R200:1=0x0000
3. Check if other devices sharing the GPIO bus will have conflicts with this arrangement
 - a. If a GPIO configured as an input is not allowed to be set as output during standby, have the external source hold its output HIGH or LOW during standby.
4. To put the camera in standby:
 - a. Assert STANDBY=1. Optionally, stop the CLKIN clock to minimize the standby current specified in the MT9D111 data sheet. For soft standby, program standby R13:0[2]=1 instead.

To Exit Standby

1. De-assert standby:
 - a. Provide CLKIN clock, if it was disabled when using STANDBY
 - b. De-assert STANDBY=0 if hard standby was used. Or program R13:0[2]=0 if soft standby was used
2. Reconfiguring output pads:
 - a. If necessary, reconfigure the GPIOs back to the desired state by GPIO variables 0x1078 and 0x1079. Also set R10:1[7]=0 if any GPIOs are used as inputs.
3. Go to preview

4. Issue a GO_PREVIEW command to the firmware by setting seq.cmd=1
5. Poll seq.state until the current state is preview (seq.state=3)

The following timing requirements should be met to turn off CLKIN during hard standby:

1. After the asserting standby, wait 10 clock cycles before stopping the clock
2. Restart the clock 24 clock cycles before de-asserting standby

PLL Setup

The PLL output frequency is determined by three constants (M, N, and P) and the input clock frequency. These three values are set in:

R102:0 // [15:8] for M; [5:0] for N

R103:0 // [6:0] for P

Their relations can be shown by the following equation:

$$f_{OUT} = f_{IN} \times M / [2 \times (N+1) \times (P+1)]$$

However, since the following requirements must be satisfied, then not all combinations of M/N/P are valid:

M must be 16 or higher

f_{PFD} , f_{VCO} , f_{OUT} ranges are satisfied

Table 5: Frequency Parameters

Frequency	Equation	Min (MHz)	Max (MHz)
f_{PFD}	$f_{IN} / (N+1)$	2	13
f_{VCO}	$f_{PFD} \times M$	110	240
f_{OUT}	$f_{VCO} / [2 \times (P+1)]$	6	80
f_{IN}	—	6	64

After determining the proper M, N, and P values and setting them in R102:0/R103:0, the PLL can be enabled by the following sequence:

R101:0[14] = 0 // powers on PLL

R101:0[15] = 0 // disable PLL bypass (enabling PLL)

Note: If PLL is used, bypass the PLL (R101:0[15]=1) before going into hard standby. It can be enabled again (R101:0[15]=0) once the sensor is out of standby.

Identifying Chip Version

The sensor version as well as the firmware version can be determined by reading its respective register/variable.

R0:0 (i.e. =0x1519) // sensor chip version #
mon.ver (ID=0, Offset=12, 8-bit variable) // firmware version

Initializing FAQs

- Is there any other way to make entering Standby Mode faster?
- If the PLL settings must change during operation, which registers and which order should we set? Could you show me sequence flow charts of context switch with PLL setting changes?

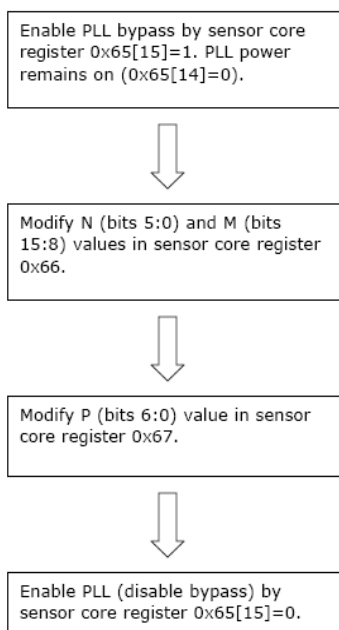
Is there any other way to make entering Standby Mode faster?

We recommend that you follow the standby sequence we have outlined in the document because the MCU needs to prepare for standby (seq.cmd = 3), then the sensor acknowledges standby (seq.state = 9) and then asserts STANDBY. To make the camera enter standby mode quicker, you can set the PLL to run at maximum frequency (80 MHz).

If the PLL settings must change during operation, which registers and which order should we set? Could you show me sequence flow charts of context switch with PLL setting changes?

In order to change PLL settings, sensor core registers 0x65, 0x66, 0x67 are used. For the order of settings, see the flow chart in Figure 5.

Figure 5: PLL Setting Change Flow Chart



Context Switching and Output Configuration

Context Switch and Setup

There are two contexts (or modes) available, A and B. Context A is known as the preview mode and has a default resolution of 800 x 600, while context B is called the capture mode with a default resolution of 1600 x 1200.

To switch from preview to the capture state, set the following variables:

- seq.captureParams.mode[1]=1// ID=1, Offset=0x20
- seq.cmd=2// ID=1, Offset=0x03

To switch from capture back into preview mode, the following settings are used:

- seq.captureParams.mode[1]=0// ID=1, Offset=0x20
- seq.cmd=1// ID=1, Offset=0x03

The current context mode can also be read via the serial interface from the variable mode.context (DriverID=7, Offset=0x02). If mode.context is read back as logic "1", the capture mode is active. When the bit is cleared, the current mode is preview.

For each context, there is a set of variables that enables the user to configure its properties. These settings are automatically put into effect by the firmware during context switching. Examples of configurable options are: output resolution, crop sizes, data format, output FIFO, spoof mode, slew rate, special effects, and gamma table. For more information, see the related description in this chapter or "Mode Driver-Setting up Preview (A) and Capture (B) Modes" on page 131."

Changing the Output Resolution

In order to change the output size of a context, the associated context image height and width values can be updated before switching to the targeted context. If the size is changed for the current (active) context, then a refresh command (seq.cmd=5) needs to be executed additionally.

For context A, use the variable "mode.Output Width A" (ID=7, offset=3) and "mode.Output Height A" (ID=7, offset=5). For context B, use the variable "mode.Output Width B" (ID=7, offset=7) and "mode.Output Height B" (ID=7, offset=9).

Selecting Output Data Formats

The MT9D111 can output several different formats. They are YCbCr, RGB565, RGB555, RGB444, JPEG, and raw data.

To select between YUV and RGB, bit 5 of variables "mode.output_format_A" (ID=7, Offset=0x7D) and "mode.output_format_B" (ID=7, Offset=0x7E) should be used, depending on which context mode is of interest. Within RGB, 565/555/444x/x444 modes can be chosen by bits 6–7 of the same variable:

Table 6: Changing Output Format Variables

Bits[7:6]	RGB Mode
00	16-bit RGB565
01	15-bit RGB555
10	12-bit RGB444x
11	12-bit RGBx444

A refresh is needed (seq.cmd=5) before the new settings are effective. As for JPEG images, only context B has support for it. See “Enabling and Capturing JPEG” on page 27 for more details.

Other bits for "mode.output_format_A" (ID=7, Offset=0x7D) and "mode.output_format_B" (ID=7, Offset=0x7E) are used for:

Table 7: Output Format Option Configuration Settings

Bit	
0	In YUV output mode, setting this bit high would swap Cb and Cr channels. In RGB mode, it will swap the R and B channel. This bit is subject to synchronous update.
1	Setting this bit high would swap the chrominance byte with luminance byte in YUV output. In RGB mode, it will swap odd and even bytes. This bit is subject to synchronous update.
2	Progressive Bayer.
3	Monochrome output.

To select the 10-bit raw data from the sensor core, see “Raw Bayer Data Output.”

Raw Bayer Data Output

There are two ways to obtain raw Bayer data. In both cases, the data from the sensor core bypasses the color pipeline. Hence, it does not go through the any of its image processing units (however, AE, AWB, etc. may still be active).

The first option is to output the all 10 bits in parallel. In this case, DOUT0–DOUT7 represents sensor core data D[9:2] and GPIO[9:8] representing sensor core data D[1:0].

In order to put the sensor in this mode, the MCU (microcontroller unit) should first be disabled by setting: R195:1=1 // disable MCU

Next, enable the bypass mode with: R9:1=0 // enable sensor core bypass

The pad slew while in bypass can be set using R10:1[2:0]. With the color pipeline and MCU disabled, the sensor core parameters (integration time, gains, image size, power mode, etc.) can be programmed manually.

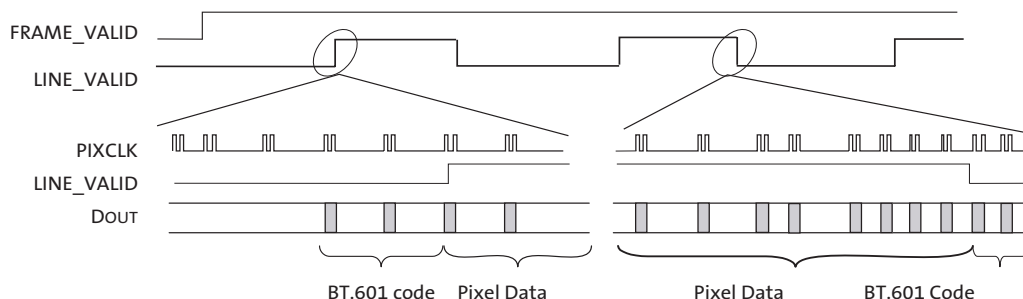
The second option is to enable "8+2 bypass" by using DOUT0–DOUT7 only. In this mode, the data bits are sent out in two bytes: D9–D2 in the first byte, and D1–D0 (with 0s padded in the more significant bit positions) for the second byte.

To enable this mode, first program register R9:1[2:0]=1 to set the proper data flow, then turn on "8+2" with R152:1[6]=1.

Output Format and Timing

Uncompressed YUV or RGB data can be output either directly from the output formatting block or via a FIFO buffer with a capacity of 1,600 bytes, enough to hold one half uncompressed line at full resolution. Buffering of data is a way to equalize the data output rate when image decimation is used. Decimation produces an intermittent data stream consisting of short high-rate bursts separated by idle periods. Figure 6 depicts such a stream. High pixel clock frequency during bursts may be undesirable due to EMI concerns.

Figure 6: Timing of Decimated Uncompressed Output Bypassing the FIFO



Note: PIXCLK default inverted.

Figure 7 depicts the output timing of uncompressed YUV/RGB when a decimated data stream is equalized by buffering or when no decimation takes place. The pixel clock frequency remains constant during each LINE_VALID high period. Decimated data are output at a lower frequency than full size frames, which helps to reduce EMI.

Figure 7: Timing of Uncompressed Full Frame or Decimated Output Passing through the FIFO

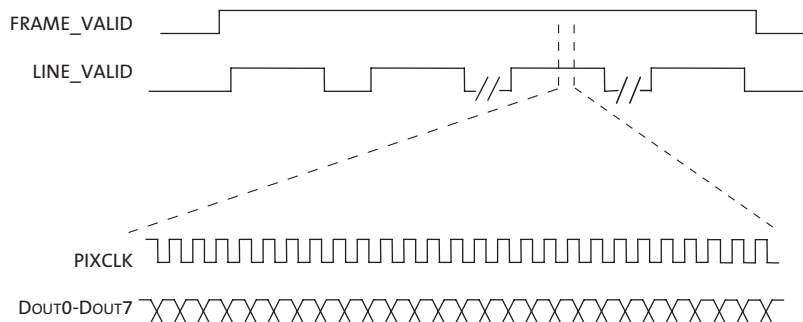
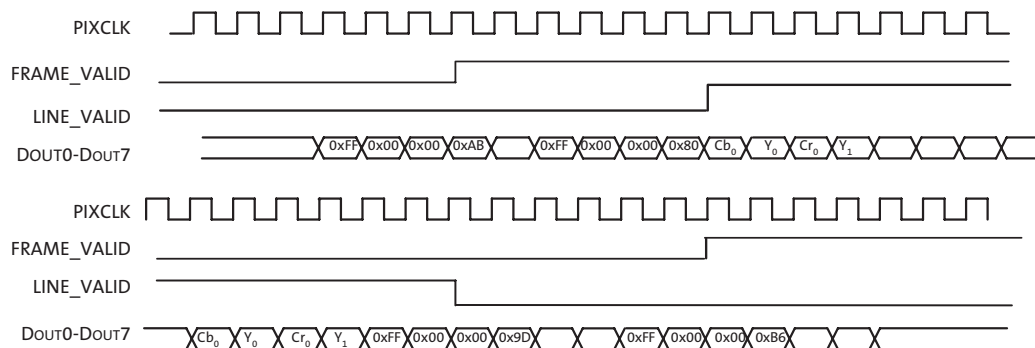


Figure 8: Example of Timing for Non-Decimated Uncompressed Output Bypassing Output FIFO



The MT9D111 supports swapping YCrCb mode as illustrated in Table 8, “YCrCb Output Data Ordering,” on page 20.

Table 8: YCrCb Output Data Ordering

Mode				
Default (no swap)	Cb_i	Y_i	Cr_i	Y_{i+1}
Swapped CrCb	Cr_i	Y_i	Cb_i	Y_{i+1}
Swapped YC	Y_i	Cb_i	Y_{i+1}	Cr_i
Swapped CrCb, YC	Y_i	Cr_i	Y_{i+1}	Cb_i

The RGB output data ordering in default mode is shown in Table 9. The odd and even bytes are swapped when luma/chroma swap is enabled. R and B channels are bit-wise swapped when chroma swap is enabled.

Table 9: RGB Ordering in Default Mode

Mode (swap disabled)	Byte	$D_7D_6D_5D_4D_3D_2D_1D_0$
RGB 565	Odd	$R_7R_6R_5R_4R_3G_7G_6G_5$
	Even	$G_4G_3G_2B_7B_6B_5B_4B_3$
RGB 555	Odd	$0R_7R_6R_5R_4R_3G_7G_6$
	Even	$G_4G_3G_2B_7B_6B_5B_4B_3$
RGB 444x	Odd	$R_7R_6R_5R_4G_7G_6G_5G_4$
	Even	$B_7B_6B_5B_40000$
RGB x444	Odd	$0000R_7R_6R_5R_4$
	Even	$G_7G_6G_5G_4B_7B_6B_5B_4$

JPEG compression of IFP output produces a data stream structure that differs from that of an uncompressed YUV/RGB stream. Frames are no longer sequences of lines of constant length. This difference is reflected in the timing of the LINE_VALID signal. When JPEG compression is enabled, logical HIGHs on LINE_VALID do not correspond to image lines, but to variably sized packets of valid data. In other words, the LINE_VALID signal is in fact a DATA_VALID signal. It is a good analogy to compare the JPEG output of the MT9D111 to an 8-bit parallel data port wherein the LINE_VALID signal indicates valid data and the FRAME_VALID signal indicates frame timing.

The JPEG compressed data can be output either continuously or in blocks simulating image lines. The latter output scheme is intended to spoof a standard video pixel port connected to the MT9D111 and for that purpose treats JPEG entropy-coded segments as if they were standard video pixels. In the continuous output mode, JPEG output clock can be free running or gated. In all, three timing modes are available and are depicted in Figure 7 on page 19, Figure 8 on page 19, and Figure 9 on page 21. These timing diagrams are merely three typical examples of many variations of JPEG output. The “continuous” and spoof JPEG output modes differ primarily in how the LINE_VALID output is asserted. In the continuous mode, LINE_VALID is asserted only during output clock cycles containing valid JPEG data. The resulting LINE_VALID signal pattern is non-uniform and highly image dependent, reflecting the inherent nature of JPEG data stream. In the spoof mode, LINE_VALID is asserted and de-asserted in a more uniform pattern emulating uncompressed video output with horizontal blanking intervals. When LINE_VALID is de-asserted, available JPEG data are not output, but instead remain in the FIFO until LINE_VALID is asserted again. During the time when LINE_VALID is asserted, the output clock is gated off whenever there is no valid JPEG data in the FIFO.

Note: As a result, spoof “lines” containing the same number of valid data bytes may be output within different time intervals depending on constantly varying JPEG data rate.

The host processor configures the spoof pattern by programming the total number of LINE_VALID assertion intervals, as well as the number of output clock periods during and between LINE_VALID assertions. In other words, the host processor can define a temporal "frame" for JPEG output, preferably with "size" tailored to the expected JPEG file size. If this frame is too large for the total number of JPEG bytes actually produced, the MT9D111 either de-asserts FRAME_VALID or continues to pad unused "lines" with zeros until the end of the frame. If the frame is too small, the MT9D111 either continues to output the excess JPEG bytes until the entire JPEG compressed image is output or discards the excess JPEG bytes and sets an error flag in a status register accessible to the host processor.

In the continuous mode, the JPEG output clock can be configured to be either gated off or running while LINE_VALID is de-asserted. To save extra power, the JPEG output clock can also be gated off between frames (when FRAME_VALID is de-asserted) in both continuous and spool output mode. In the continuous output mode, there is an option to insert JPEG SOI (0xFFD8) and EOI (0xFFD9) markers respectively before and after valid JPEG data. SOI and EOI can be inserted either inside or outside the FRAME_VALID assertion period, but always outside LINE_VALID assertions.

The output order of even and odd bytes of JPEG data can be swapped in the spoof output mode. This option is not supported in the continuous mode.

Output clock speed can optionally be made to vary according to the fullness of the FIFO, to reduce the likelihood of FIFO overflow. When this option is enabled, the output clock switches at three fixed levels of FIFO fullness (25 percent, 50 percent and 75 percent) to a higher or lower frequency, depending on the direction of fullness change. The set of possible output clock frequencies is restricted by the fact that its period must be an integer multiple of the master clock period. The frequencies to be used are chosen by programming three output clock frequency divisors in the mode driver FIFO variables (mode.fifo_conf1_A/B and mode.fifo_conf2_A/B). Divisor N1 is used if the FIFO is less than 50 percent full and last fullness threshold crossed has been 25 percent. When the FIFO reaches 50 percent and 75 percent fullness, the output clock switches to divisor N2 and N3, respectively. When the FIFO fullness level drops to 50 percent and 25 percent, the output clock is switched back to divisor N2 and N1, respectively.

The host processor can read registers containing JPEG status flags and JPEG data length (total byte count of valid JPEG data) via a two-wire serial interface. In addition, the JPEG data length and JPEG status byte are always appended at the end of JPEG spoof frame. JPEG status byte can be optionally appended at the end of JPEG continuous frame. JPEG data stream sent to the host does not have a header.

Figure 9: Timing of JPEG Compressed Output in Free-Running Clock Mode

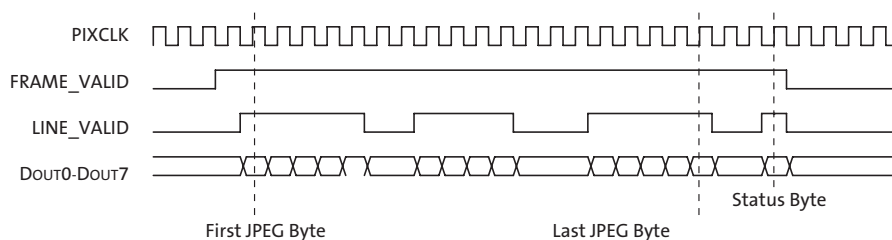
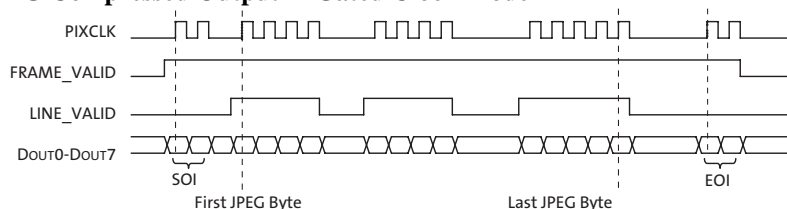
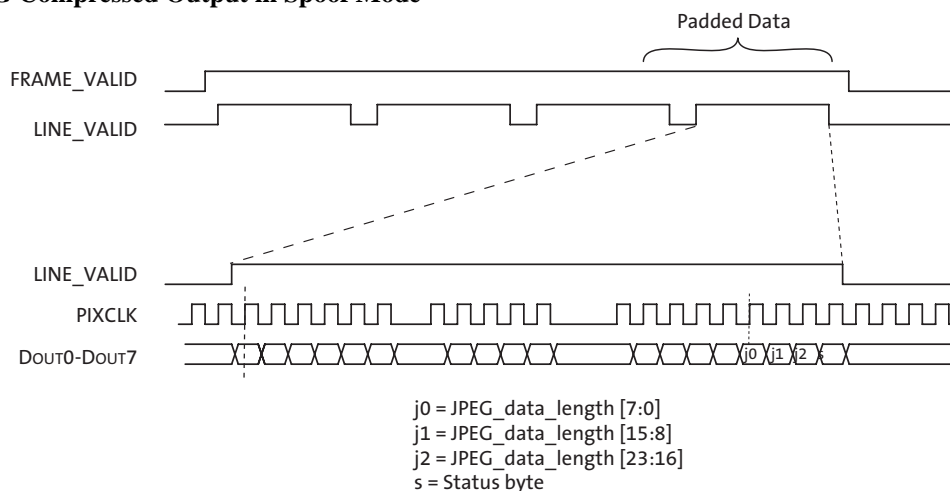


Figure 10: Timing of JPEG Compressed Output in Gated Clock Mode


Note: PIXCLK default inverted.

Figure 11: Timing of JPEG Compressed Output in Spoof Mode


Note: PIXCLK default inverted.

In the spoof mode, the timing of FRAME_VALID and LINE_VALID mimics an uncompressed output. The timing of PIXCLK and DOUT within each LINE_VALID assertion period is variable and therefore unlike that of uncompressed data. Valid JPEG data are padded with dummy data to the size of the original uncompressed frame.

Decimation, Zoom, and Pan

Decimation

Registers linking to decimation can be automatically adjusted by setting the related firmware variables. In preview mode, these variables need to be adjusted to 800 x 600 or lower resolution:

- mode.Output Width A // ID=7, Offset=0x03
- mode.Output Height A // ID=7, Offset=0x05

As for capture mode, these variables need to be adjusted to a resolution of 1600 x 1200 or lower:

- mode.Output Width B // ID=7, Offset=0x07
- mode.Output Height B // ID=7, Offset=0x09

Keep in mind that at all times, the output sizes above needs to be equal or smaller than the input size to the decimator (crop sizes).

If the modification in decimation occurs for the current (active) context, a refresh command (seq.cmd=6) is needed to reflect the new values. Otherwise, a context switch to the targeted context automatically refreshes in these values.

Zoom

The zoom feature can be enabled only if the output size is smaller than the cropped size for each context mode. Once the output size is reduced, the crop size can be reduced also to provide a “zooming” effect. The variables that are involved are:

For context A (preview):

```
mode.Output Width A// ID=7, Offset=0x03
mode.Output Height A// ID=7, Offset=0x05
mode.crop_X0_A// ID=7, Offset=0x27
mode.crop_X1_A// ID=7, Offset=0x29
mode.crop_Y0_A// ID=7, Offset=0x2B
mode.crop_Y1_A// ID=7, Offset=0x2D
seq.cmd=6// to refresh in the new settings
```

For context B (capture):

```
mode.Output Width B// ID=7, Offset=0x07
mode.Output Height B// ID=7, Offset=0x09
mode.crop_X0_B// ID=7, Offset=0x35
mode.crop_X1_B// ID=7, Offset=0x37
mode.crop_Y0_B// ID=7, Offset=0x39
mode.crop_Y1_B// ID=7, Offset=0x3B
seq.cmd=6// to refresh in the new settings
```

Pan

The Pan feature can be used once the image is zoomed in as describe above. In order to move (or “Pan”) the image view horizontally or vertically, an offset is applied to the corresponding crop variables:

In preview mode, apply the same offset to “mode.crop_X0_A” and “mode.crop_X1_A” to pan horizontally. Apply the same offset to “mode.crop_Y0_A” and “mode.crop_Y1_A” to pan vertically.

As for capture mode, apply the same offset to “mode.crop_X0_B” and “mode.crop_X1_B” to pan horizontally. Apply the same offset to “mode.crop_Y0_B” and “mode.crop_Y1_B” to pan vertically.

Enabling Special Effects

Special effect (monochrome, sepia, negative, solarization) formats can be selected by variables “mode.spec_effects_A” (ID=7, Offset=0x7F) in context A or “mode.spec_effects_B” (ID=7, Offset=0x81) in context B. Their settings are shown in Table 10.

Table 10: Enabling Special Effects

Value of Bits[2:0]	Effect
0	disabled (no special effects)
1	monochrome
2	sepia
3	negative
4	solarization with unmodified UV
5	solarization with UV

Note: Seq.cmd=5 is also necessary to refresh the new settings for special effects.

Mirroring the Image

An image can be mirrored horizontally or vertically or both in the current context (A/B). The registers associated to these functions are:

- R32:0[0] // set to mirror rows
- R32:0[1] // set to mirror columns

Column and Row Skip

Column and row Skipping are available in both context A and B. When enabled, the image is subsampled, which results in a smaller dimension. This is one method used to reduce image size and have less constraint on the row time requirements. In all cases, the row and column sequencing ensures that the Bayer pattern is preserved. Skipping can be enabled and configured by:

Context A

R33:0[3:2] // Row skip level (00=2x, 01=4x, 10=8x, 11=16x)
R33:0[4] // Row skip enable
R33:0[6:5] // Column Skip level (00=2x, 01=4x, 10=8x, 11=16x)
R33:0[7] // Column skip enable

Context B

R32:0[3:2] // Row skip level (00=2x, 01=4x, 10=8x, 11=16x)
R32:0[4] // Row skip enable
R32:0[6:5] // Column Skip level (00=2x, 01=4x, 10=8x, 11=16x)
R32:0[7] // Column skip enable

Note: For skip level 4x or higher, the MCU must be disabled by setting R195:1[0]=1. In addition, the proper image output and crop sizes must be updated beforehand. As an example, the following values are set to enable 4x skipping in context A.

mode.Output Width A=400 // ID=7, Offset=3
mode.Output Height A=300 // ID=7, Offset=5
mode.crop_X1_A=400 // ID=7, Offset=41
mode.crop_Y1_A=300 // ID=7, Offset=45
seq.cmd=5 // ID=1, Offset=3

R195:1[0]=1// disable MCU

R33:0=1204// 4x skipping, no binning

Binning

The MT9D111 supports a 2 x 2 binning mode, which is used primarily instead of 2x skip to decimate a picture without losing information. The effect of aliasing in preview mode is eliminated when binning is used instead of just skipping rows and columns. To enable binning in preview mode, R33:0[15] (or R32:0[15] for capture mode) should be set high.

For proper operation, the following to binning applies:

- Start address must be divisible by four (row and column)
- Window size must be divisible by four in both directions, after dividing by zoom factor and skip factor (because they both reduce the effective window size from the sensor's point of view)

Example

Default row size = 1200. 8x zoom means the actual window on the sensor is divided by 8, so 8x zoom and binning is not allowed with default window size, because $1200/8 = 150$, which is not divisible by 4.

- Since binning can be seen as an extra level of skip. The combination binning/16x skip is not possible.

Configuring Pad Slew

During normal operation (no bypass), the slew rate for the DOUT0-DOUT7, PIXCLK, FRAME_VALID, and LINE_VALID are set by mode variables. In context A, they are defined by:

- mode.fifo_conf1_A[7:5]// PCLK1 slew rate (ID=7, Offset=109)
- mode.fifo_conf1_A[15:13]// PCLK2 slew rate (ID=7, Offset=109)
- mode.fifo_conf2_A[7:5]// PCLK3 slew rate (ID=7, Offset=111)
- seq.cmd=5// refresh for new settings to be effective

As for context B, they are set by:

- mode.fifo_conf1_B[7:5]// PCLK1 slew rate (ID=7, Offset=116)
- mode.fifo_conf1_B[15:13]// PCLK2 slew rate (ID=7, Offset=116)
- mode.fifo_conf2_B[7:5]// PCLK3 slew rate (ID=7, Offset=118)
- seq.cmd=5// refresh for new settings to be effective

The slew rate for GPIO[11:0] pads are set by R10:1[6:4] and SDATA pad by R10:1[10:8]. A value of 7 is designated as the fastest slew while a value 0 is defined as the slowest slew.

Note: The actual slew depends on load, temperature, and I/O voltage. Hence the proper slew rate should be tested and determined for each system. The default slew value will not work for all setups.

For bypass mode-such as sensor or SOC bypass, as set by R9:1[2:0]-the slew rate for DOUT0-DOUT7, PIXCLK, FRAME_VALID, and LINE_VALID is set by R10:1[2:0].

Capturing Still Pictures

In order to capture still images, the capture video mode bit must first be cleared:

seq.captureParams.mode[1]=0

Next, the specify the output image size for context B by using the variables "mode.Output Width B" and "mode.Output Height B". If the image is cropped from the original size, the following variables should also be set appropriately:

- mode.crop_X0_B // ID=7, Offset=0x35
- mode.crop_X1_B // ID=7, Offset=0x37
- mode.crop_Y0_B // ID=7, Offset=0x39
- mode.crop_Y1_B // ID=7, Offset=0x3B

For example, if the user wants to capture a full resolution image (no cropping), these values should be set:

- mode.Output width_B= 1600 // ID=7, Offset=0x07
- mode.Output height_B=1200 // ID=7, Offset=0x09
- mode.crop_X0_B=0 // ID=7, Offset=0x35
- mode.crop_X1_B=1600 // ID=7, Offset=0x37
- mode.crop_Y0_B=0 // ID=7, Offset=0x39
- mode.crop_Y1_B=1200 // ID=7, Offset=0x3B

As for an 800x600 capture with no cropping:

- mode.Output width_B= 800
- mode.Output height_B=600
- mode.crop_X0_B=0
- mode.crop_X1_B=1600
- mode.crop_Y0_B=0
- mode.crop_Y1_B=1200

As for a VGA capture with cropping of 800 x 600:

- mode.Output Width B=640
- mode.Output Height B=480
- mode.crop_X0_B=0
- mode.crop_X1_B=800
- mode.crop_Y0_B=0
- mode.crop_Y1_B=600

If the image size is less than or equal to 800 x 600, binning mode with 1ADC must be enabled by:

- R32:0[10]=1 and R32:0[15]=1

In addition, set the horizontal blanking for context B (R5:0) such that the integration time is the same as preview mode.

Next, set how many frames to be issued in capture mode before sequencer goes back to preview mode:

- seq.captureParams.numFrames=2 // 2 for this example (ID=1, Offset=0x21)

Lastly, call the "CAPTURE" command:

- seq.cmd=2 // ID=1, Offset=0x03

Capturing Videos

First, set the capture video mode bit by:

```
seq.captureParams.mode[1]=1 // ID=1, Offset=0x20
```

Next, specify the output size with "mode.Output Width B" and "mode.Output Height B". Similar to the "Capturing Still Pictures" on page 25, the appropriate variables (ID=7, Offset=0x35/37/39/3B) also need to be adjusted if cropping is used.

If the image size is less than or equal to 800x600 turn on power (1ADC) and binning mode for context_B:

```
R32:0[10]=1, R32:0[15]=1
```

Set H_Blanking for context_B (R5:0) to keep the same integration time as in a preview mode. Set mode.sensor_x_delay_B (ID=7, Offset=0x23) to adjust frame timing accurately. Set V_Blanking for context_B (R6:0) to obtain 30 fps or another target frame rate.

Lastly, call the "CAPTURE" command:

- seq.cmd=2 // ID=1, Offset=0x03

To stop video capture and return back to preview, call "PREVIEW" command:

- seq.cmd=1

Enabling and Capturing JPEG

In order to enable JPEG output, the user needs to confirm the following value:

```
mode.mode_config[5] = 0 // ID=7, offset=0x0B
```

JPEG is supported in context B, not A. If the JPEG size needs to be changed, the context B image height and width needs to be updated. For example, to change the JPEG image to 800x600 requires the following:

- VAR=7, 0x07, 0x0320 // MODE_OUTPUT_WIDTH_B
- VAR=7, 0x09, 0x0258 // MODE_OUTPUT_HEIGHT_B

After enabling JPEG output, a still JPEG image can be captured by following the procedure in the "Capturing Still Pictures" on page 25. For JPEG video, enable JPEG output, then follow "Capturing Videos" on page 26.

Switching Between JPEG 4:2:2, 4:2:0, and Monochrome

Before going into the capture mode for JPEG images, the desired resolution and format should be selected. To set the output size, these variables need to have the target values:

- mode.Output Width B // output width for capture (ID=7, Offset=0x07)
- mode.Output Height B // output height for capture (ID=7, Offset=0x09)

As for the JPEG format, the user can select between 4:2:2, 4:2:0 or monochrome by setting the variable:

- jpeg.format // 0=4:2:2, 1=4:2:0, 2 = monochrome (ID=9, Offset=0x06)

Note: For the 4:2:0 format, the maximum width is 384 pixels (no special restriction on height). Next, switch to capture mode (see "Context Switch and Setup" on page 17) to output the new JPEG settings.

Context Switching and Output Configuration FAQs

- What are the recommended settings if we want to share 1 ADC in contexts A and B because brightness changes when mode change?
- Is it possible to have the same pclk in Context A and Context B?
- Is it possible to use only 1 ADC in context B? How do we disable the default JPEG setting to use YUV4:2:2?
- Is it required to write seq.cmd=1 (Do preview) but only seq.cmd=5 (Refresh) when the camera starts up?
- Is there seq.cmd writing timing limitation? In our evaluation when I wrote seq.cmd=5 after seq.cmd=1, the setting of seq.cmd=5 was not reflected.

What are the recommended settings if we want to share 1 ADC in contexts A and B because brightness changes when mode change?

When context A and context B share 1 ADC, it is best to enable the AE driver in the captureEnter state. One may be trying to minimize the state transition time by skipping previewLeave and captureEnter states. However, there is a trade-off here between quality and time. Therefore, we recommend having the following settings:

- VAR8=1, 0x37, 0x0001 // SEQ_PREVIEW_3_AE (fast AE in captureEnter state)
- VAR8=1, 0x3D, 0x0000 // SEQ_PREVIEW_3_SKIPFRAME (no skipping captureEnter state)

The previewLeave state may be skipped, but in order to have same brightness in context A and B, captureEnter state should not be skipped.

Is it possible to have the same pclk in Context A and Context B?

Yes, it is possible to have the same PCLK frequency in context A and context B. If this setup is used, both contexts will need to have the same number of ADC. Horizontal blanking will also need adjustments otherwise column FPN might appear.

Is it possible to use only 1 ADC in context B? How do we disable the default JPEG setting to use YUV4:2:2?

It is possible to use only 1 ADC in context B. Rev2 and Rev3 have JPEG as the default setting for context B. To disable JPEG and use YUV422, set the following settings:

- Mode.mode_config: ID=7, Offset=11, bit 5 = 1 // bypass JPEG
- Mode.fifo_conf1_B: ID=7, Offset=116, bit[3:0] = 1 // N1

Is it required to write seq.cmd=1 (Do preview) but only seq.cmd=5 (Refresh) when the camera starts up? At startup, the MT9D111 will automatically go into preview mode after initialization. Hence, no seq.cmd=1 (go to preview) or seq.cmd=5 (refresh) is needed. However, if settings for context A is changed (such as image size), then the following command is needed:

if firmware settings for context A is changed while the current context is A, then seq.cmd=5 (refresh is needed).

if firmware settings for context A is changed while the current context is B, then seq.cmd=1 (go to preview) will go to context A and automatically refresh (no seq.cmd=5 is needed).

For example, to set an image size of 160 x 120 (with RGB format) while in context A, the following sequence is needed:

VAR=7, 0x03, 0x00A0 // MODE_OUTPUT_WIDTH_A

VAR=7, 0x05, 0x0078 // MODE_OUTPUT_HEIGHT_A

VAR8=7, 0x7D, 0x0020 // MODE_OUTPUT_FORMAT_A

VAR8=1, 0x03, 0x0005 // SEQ_CMD (refresh in the new settings above)

Crop settings are used for zooming and panning purposes. They are not needed if you simply want to resize the original image.

Is there seq.cmd writing timing limitation? In our evaluation when I wrote seq.cmd=5 after seq.cmd=1, the setting of seq.cmd=5 was not reflected?

Setting seq.cmd=1 automatically refreshes for the new settings, hence seq.cmd=5 is not needed in this case. The context switch will automatically check for new settings. In terms of timing, seq.state variable (which is a status variable located in ID=1, offset=4) should be monitored (polled) for the status of the firmware.

For example, if the user wants to switch from context A to B, and back to A IMMEDIATELY, then one should follow the timing:

(while in preview: seq.state=3)

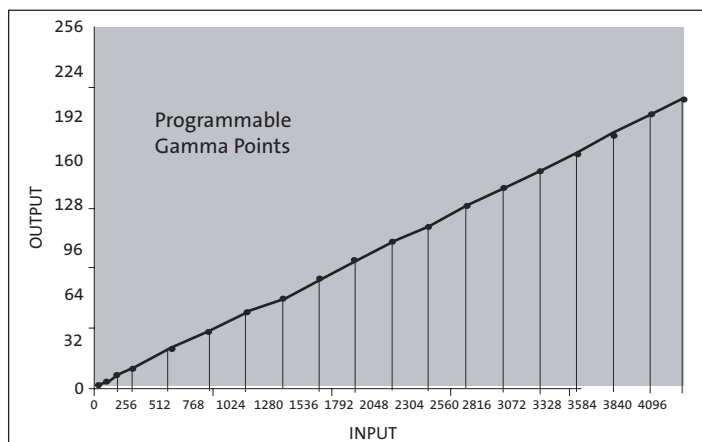
issue seq.cmd=2 (go to context B)

poll seq.state, make sure it is equal to 7 before issuing seq.cmd=1 (go back to context A)

The above is just a sample, in most cases, the user will not switch back and forth from A to B immediately. While the firmware switches from context A to B, the seq.state value will travel from 3 to 7. If a command to “go back to context A” is called before seq.state=7, it will not work because the current state is not context B yet. Note that the time required to switch from A to B is small (several frames) and depends on the clock frequency.

Gamma and Contrast

Figure 12: Gamma Correction Curve



Gamma

Because most video systems have been designed to compensate for the non-linear light-intensity response of the CRT, the camera should output a gamma-corrected image. This means that an exponential gamma curve should be applied to the image before it leaves the camera system so that when it is viewed on a CRT or other modern imaging system, it does not appear too dark.

The MT9D111 image processing chain contains a gamma correction stage. All pixel values (in RGB color coordinates) are remapped to new values based on a piecewise linear extrapolation from a 19 point lookup table. The image pixel data coming in to the gamma correction stage is 12 bits (0–4095) and is remapped to 8-bit values (0–255). The 19 input points of the gamma table correspond to the following values: 0, 64, 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, and 4095. (There are more points at the low pixel values where gamma curves usually change the most.)

Each input point is mapped to an output point (0–255). Pixel values that fall between two points are calculated based on a linear extrapolation between the nearest two points. Each of the three colors (R, G, and B) use the same input gamma table. Either the mode driver can calculate the values that make up this lookup table, or the user can upload a custom table.

The mode driver remembers the identity of two separate gamma tables:

- one gamma table for preview (A) mode
- one gamma table for capture (B) mode

Upon each mode change, these tables are calculated and uploaded to the appropriate hardware registers at the end of the current frame. For changes to be immediately apparent in the image during development, the user must issue a refresh command (5) to the sequencer.cmd variable so that the new values are recalculated and uploaded without changing modes.

For each mode (A or B), the user can select from three predefined gamma tables, or upload and select a custom table. The predefined tables can be selected by changing the mode driver variables gam_cont_A, bits[2:0], or gam_cont_B, bits[2:0], depending on the

intended mode (see mode driver variable description). Setting this value to 3 causes the mode driver to use the values stored in the mode driver variables `gamma_table_A/B_0` through `gamma_table_A/B_18` to construct the gamma table.

Contrast

Contrast enhancement can be mathematically applied to the gamma curve to give the image the appearance of wider dynamic range, but at the cost of slight errors in the color hues. The mode driver allows for predefined levels of contrast enhancement to be applied to each mode's gamma setting (or table) before updating the gamma-correction image processor. The contrast level may be set using the mode driver variables `gam_cont_A`, bits[6:4], or `gam_cont_B`, bits[6:4], depending on the intended mode (see mode driver variable description).

For changes to be immediately apparent in the image during development, the user must issue a refresh command (5) to the `sequencer.cmd` variable so that the new values are recalculated and uploaded without changing modes. Otherwise, the new settings take effect upon the next sequencer state change. The mode driver applies an s-curve function (see "S-Curve" section in documentation for more details) to the selected gamma table values and this result is uploaded to the gamma-correction image processor.

Gamma and Contrast FAQs

- How are brightness, contrast, saturation, sharpening, and color tones adjusted?
- Can contrast be adjusted by the predefined S-curves?
- How are Color tones set?
- Does aperture correction mean sharpening? Is it programmable, so that the desired amount of sharpening can be done?
- Is it possible increase or decrease the color saturation?
- The sensor output does not correspond to normal bayer output when binning is used. Do you use different bayer interpolation algorithm for binned and full output modes?

How are brightness, contrast, saturation, sharpening, and color tones adjusted?

There are different methods of changing brightness of the image—a luminance offset can be applied at the output or the exposure target can be changed from the default value.

LCD display brightness should be changed by programming luma offset. Snapshot/picture brightness should be changed using `ae.target`. Changing gamma to change brightness is not recommended—just set the gamma to match your display device.

Can contrast be adjusted by the predefined S-curves?

Yes.

How are Color tones set?

Color tones such as sepia, B&W, and negative are programmable via variables `mode.spec.effects_A` and `mode.spec.effects_B`.

Does aperture correction mean sharpening? Is it programmable, so that the desired amount of sharpening can be done?

Yes, aperture correction sharpen edges and the desired amount of sharpening is programmable.

Is it possible increase or decrease the color saturation?

Yes, by interpolation CCM between 100 percent saturated and unit matrix.

The sensor output does not correspond to normal bayer output when binning is used. Do you use different bayer interpolation algorithm for binned and full output modes?

Sensor is outputting the data in the same bayer pattern configuration during binning and therefore no other interpolation algorithm is necessary. The sensor output is the same for both modes.

Lens Shading and Correction

Introduction

This section outlines the lens shading basics featured in the MT9D111 and shows how to adjust the lens shading settings automatically. Miniature camera modules have signal degradation on sensor periphery due to optical and geometrical factors. Lens shading correction compensates the signal degradation by digitally gaining pixels on the image periphery.

Lens Shading Approach

The digital gain to correct signal degradation can be expressed as the following:

$$\text{Gain}(x, y, \text{color}) = 1 - G + F_{\text{horizontal}}(x, \text{color}) + F_{\text{vertical}}(y, \text{color}) + k * F_{\text{horizontal}}(x, \text{color}) * F_{\text{vertical}}(y, \text{color})$$

Where:

color = R, G, or B

K is the corner parameter, defined by register 0xAE.

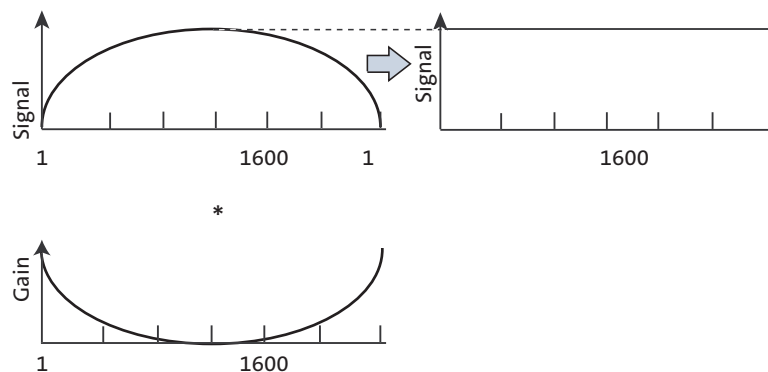
G is a global constant, defined by register 0xAD, which offsets the maximum gain of 1 (G is set to 0 for lens shading Auto-Adjust function).

The signal of each pixel is gained as follows:

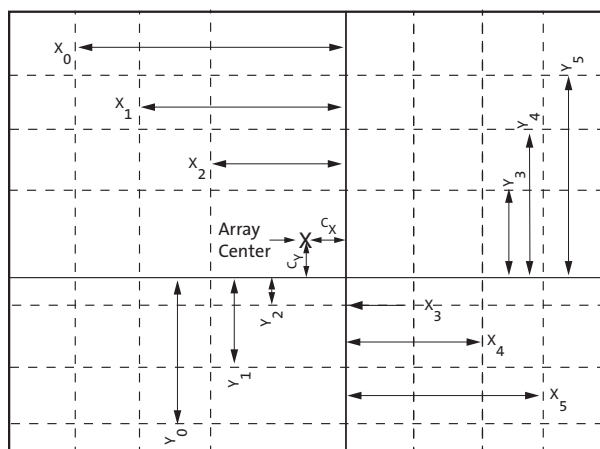
$$\text{Signal}_{\text{AFTER_LC}} = \text{Signal}_{\text{BEFORE_LC}} * \text{Gain}(x, y, \text{color})$$

The relationship of the signal before and after is shown in Figure 13.

Figure 13: Signal



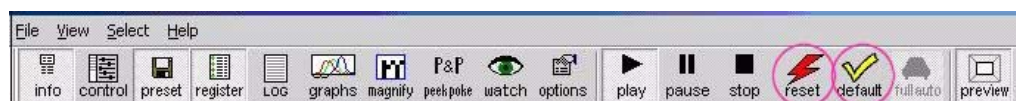
In the MT9D111, $F_{\text{horizontal}}(x, \text{color})$ and $F_{\text{vertical}}(y, \text{color})$ are piecewise quadratic (PWQ). The sensor is divided into 8 zones in the x and y directions (see Figure 14). Therefore, each function for each color is represented by 10 numbers—8 numbers for the 8 zone values in the corresponding direction and two initial conditions used to iteratively calculate function values across the entire pixel array.

Figure 14: Lens Correction Zones


Note: The array center parameters, C_x and C_y , are defined by Register 0x87 (bit 0~7 for X coordinate, 8~15 for Y coordinate).

Setup

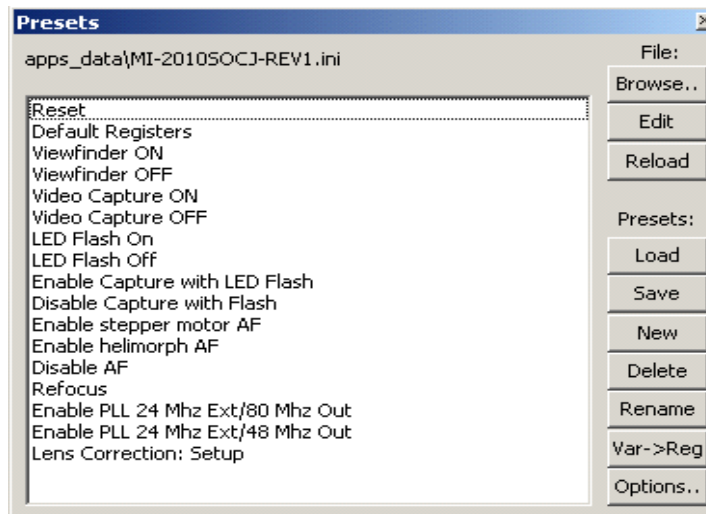
1. Start up DevWare version 2.6 Beta 5 (or above) and camera in the default state
2. Point camera at a flat and uniformly illuminated calibration target (the variation in light intensity should be no more than 5 percent, target under Daylight (6500K) produces the best result). For best result, the light source and sensor demo system should be shielded from external light sources.

Figure 15: DevWare Toolbar


3. RESET—Reset all when loaded DevWare
4. Default—Sets all registers to default
5. The automatic lens correction algorithm operates in either 800 x 600 viewfinder mode (preview) or 1600 x 1200 full resolution mode (full auto). The user can simply leave at the default preset mode, or change to full auto mode and click on zoom out once to see the full image on the screen.

Preset and Load

Figure 16: Presets Dialog Box



Select “Lens Correction: Setup” in Preset window and click “Load.” This loads all relevant register conditions that are needed to perform lens correction. The user can also choose not to load this setup setting and manually set all initial conditions listed in the following section.

Setup Conditions

1. Select Sensor Control
2. Select Gamma, Saturation and set the gamma to 1.0
3. Select Sensor Control and then Auto Exposure. Open Graphs to look at the intensity graph before calibration. Adjust Brightness Target so peak brightness is about 75 percent of the maximum intensity scale on the Analysis Graph (this correspond to about 180), then turn off Auto Exposure.
4. Select Sensor Control and then White Balance and turn off Auto White Balance.
5. Go to Sensor Control -> White Balance -> IF Settings and turn off Color Correction

Figure 17: Setting Gamma to 1.0

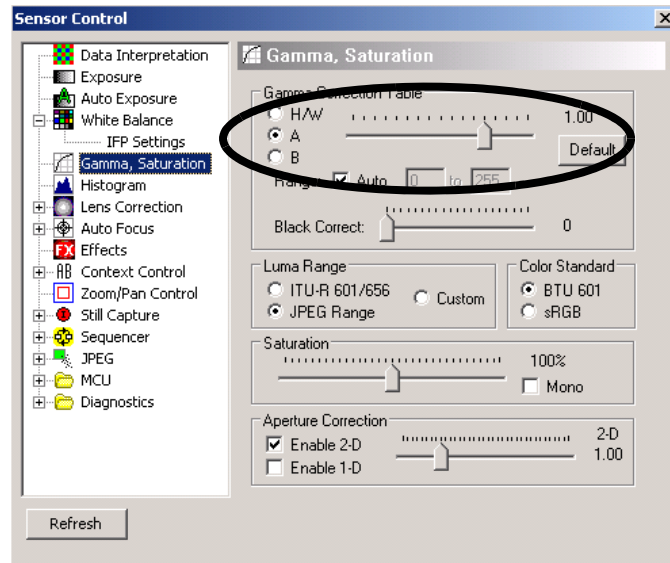


Figure 18: Enable Auto Exposure

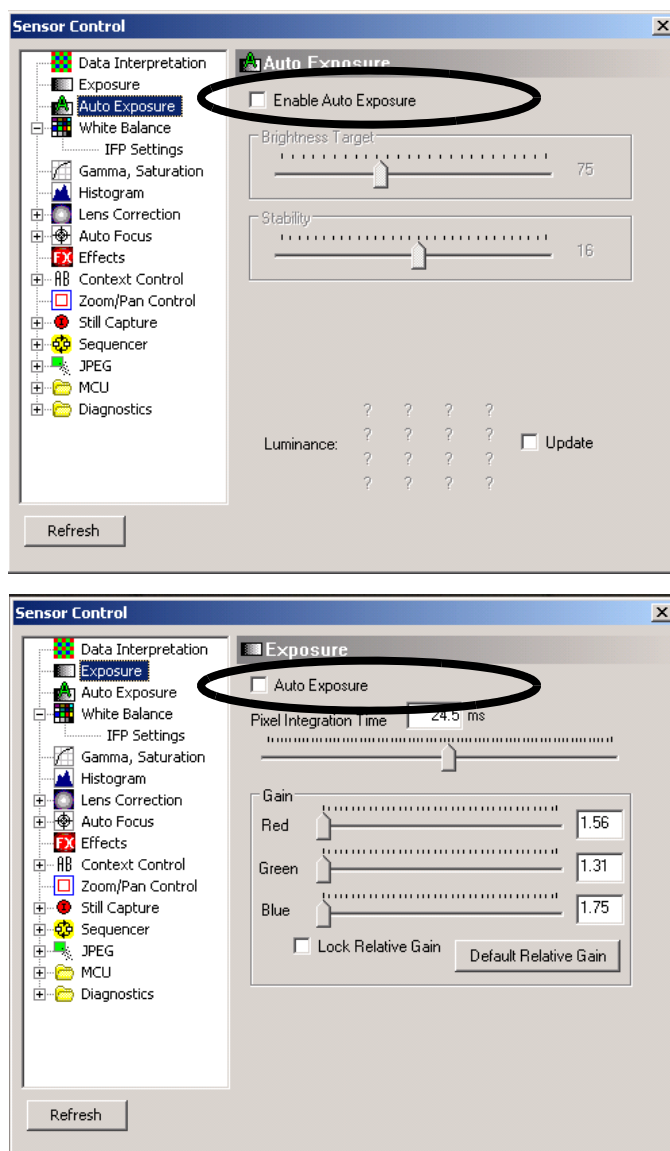
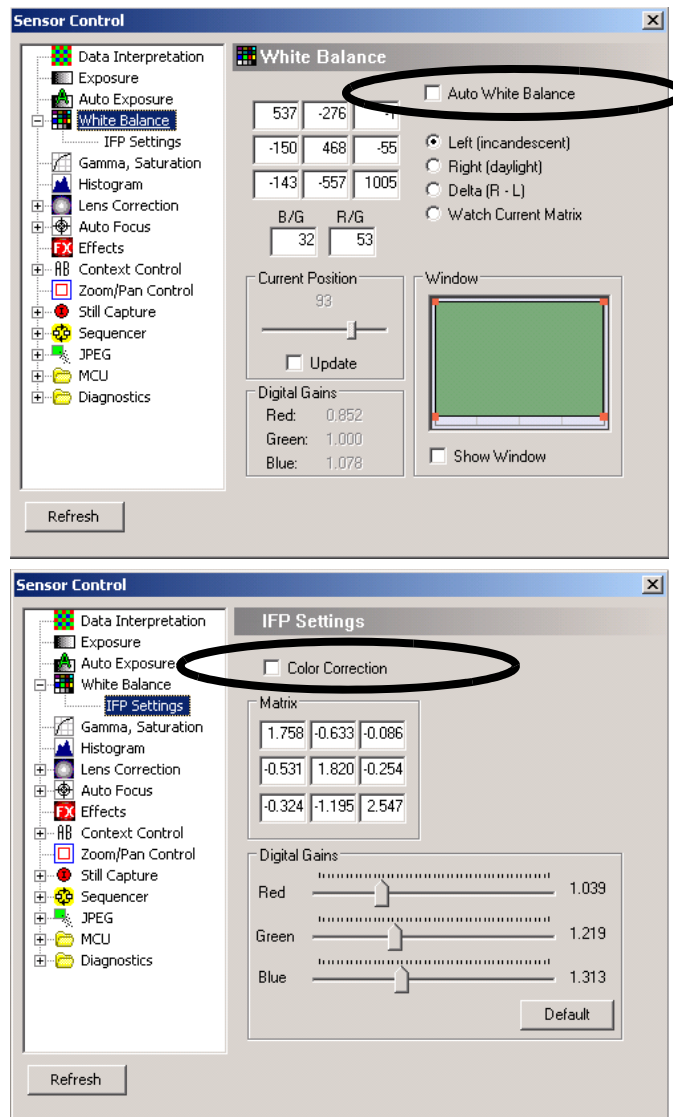


Figure 19: Disable AWB and Color Correction


Calibration

1. Set the row and column line at the middle row/column of the image. (In preview mode set row line at 300, set column line at 400. In full auto mode with 0.5x zoom, set row line at 600, column line at 800.)
2. Open Analysis Graph and view the Intensity plot of the middle row/column of the image. This will produce an analysis graph.
3. Go to Sensor Control -> Lens Correction and follow the examples in Figure 21.
4. Observe the progress on the Analysis Graph as in Figure 22. When the graph has settled, unchecked Auto-Adjust.
5. Observe the progress on the Analysis Graph as in Figure 22. When the graph has settled, unchecked Auto-Adjust.
6. Slide the horizontal scroll bar all the way to the right so the "k factor" slider becomes visible. Adjust k factor for the best appearance of the corners.

7. Make other final adjustments as you judge necessary. The user has the option to choose not to completely calibrate the lens to obtain flat intensity curves by selecting different percentage for curvature. 100 percent correspond to a completely flat curve. See Figure 24 for correlation between percentage and curvature.
8. Click Save As... to save the lens correction as an.ini file.
9. Load the new settings in DevWare and check the image quality.

Figure 20: Setting the Row and Column Line

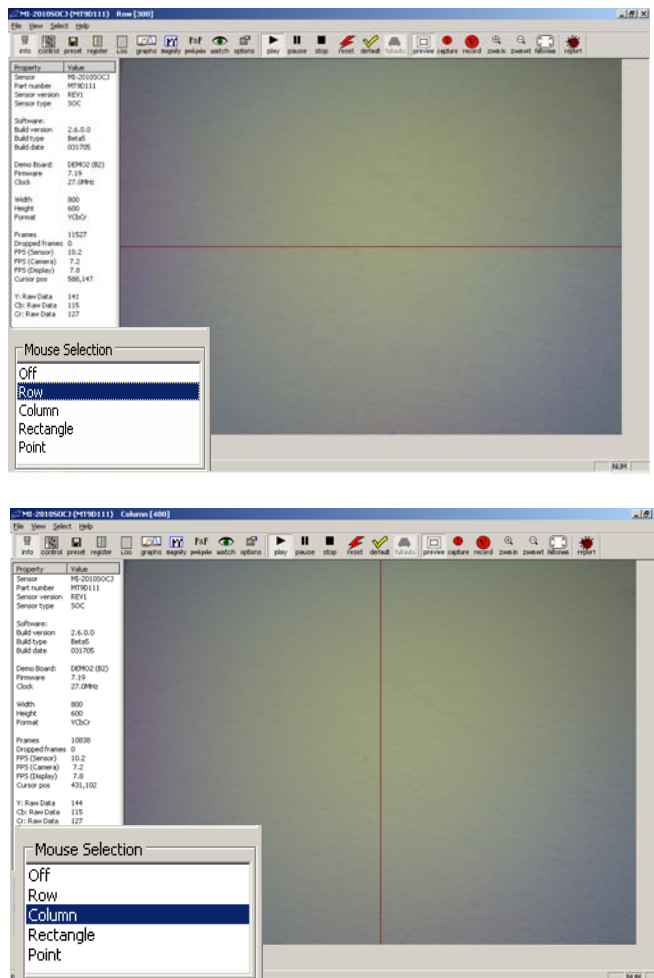


Figure 21: Sensor Control Dialog Box Showing Lens Correction Settings

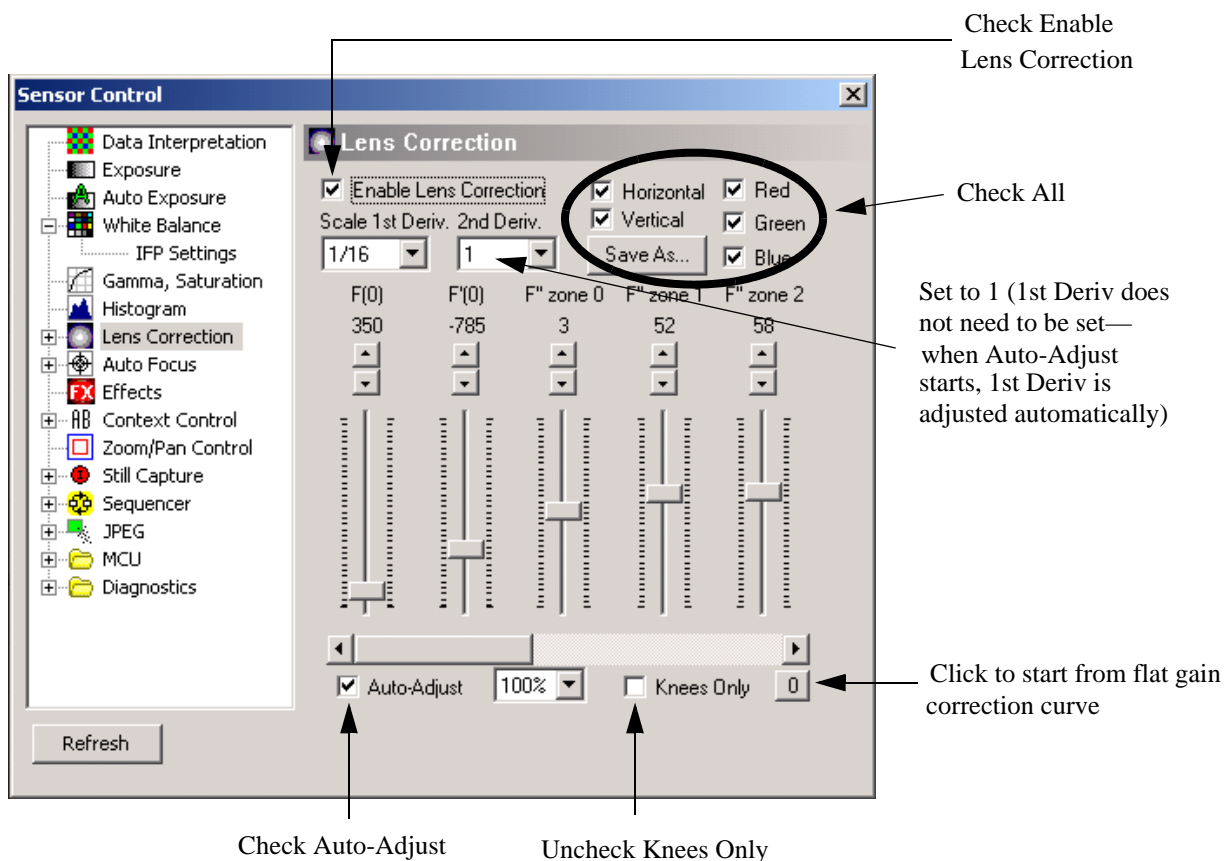


Figure 22: Settled Analysis Graph

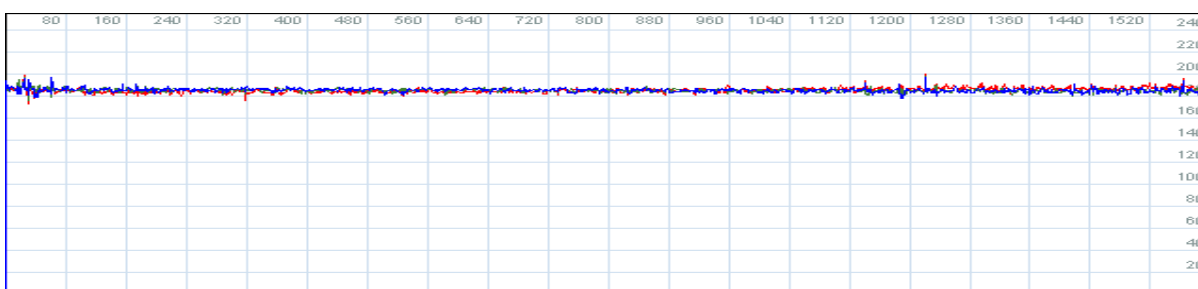


Figure 23: Adjusting K Factor

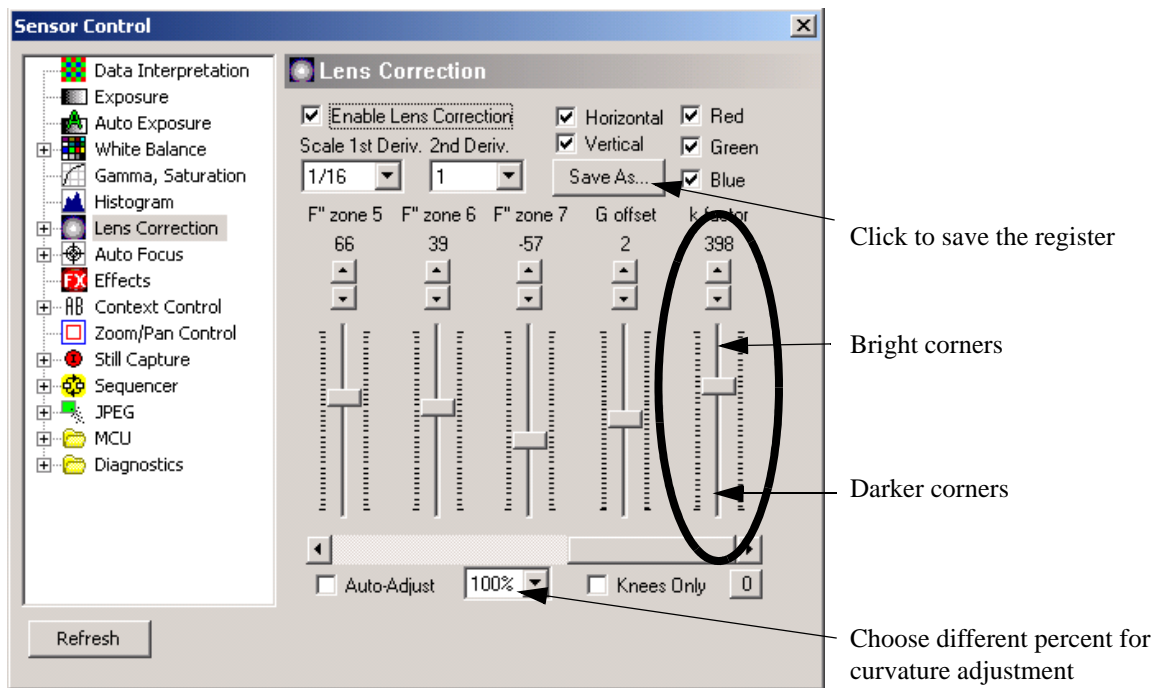
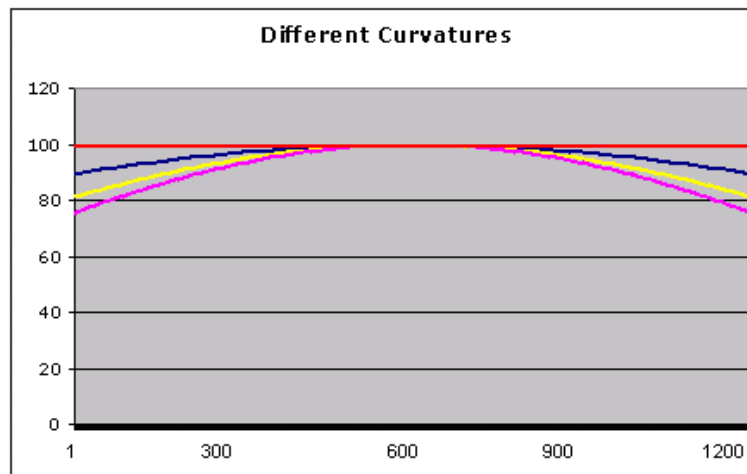


Figure 24: Correlation Between Percentage and Curvature



Result

Figure 25 shows a set of data and images BEFORE (on the left) and AFTER (on the right) the lens correction calibration procedure.

Figure 25: Lens Correction Result, Before and After

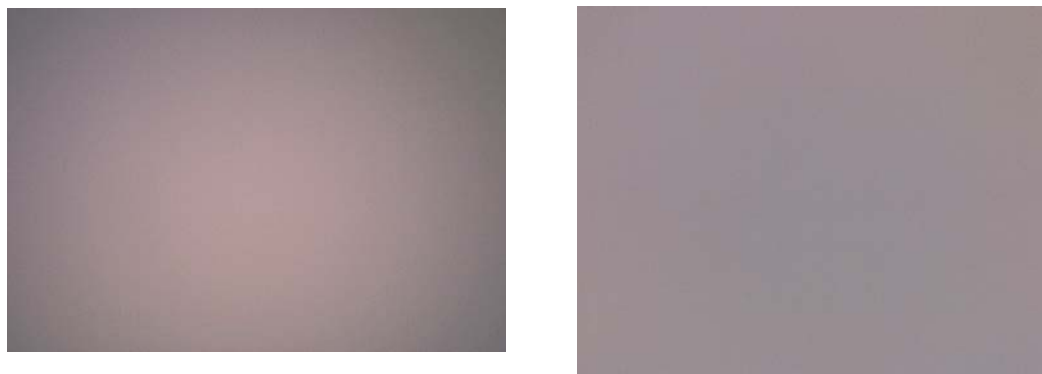


Figure 26: Intensity Graph (horizontal) Before and After

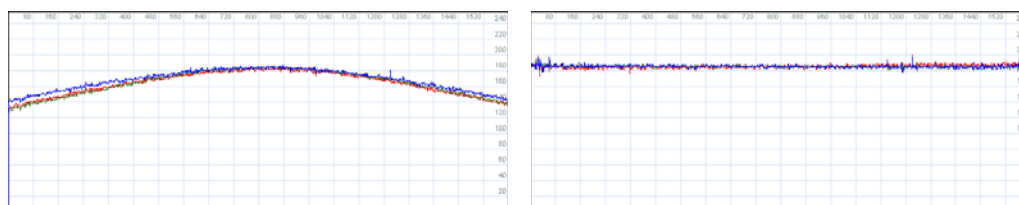
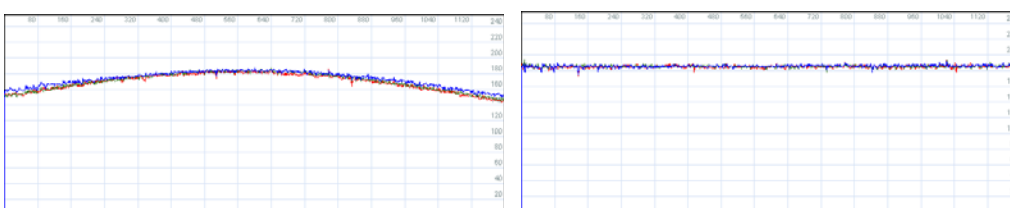


Figure 27: Intensity Graph (vertical) Before and After



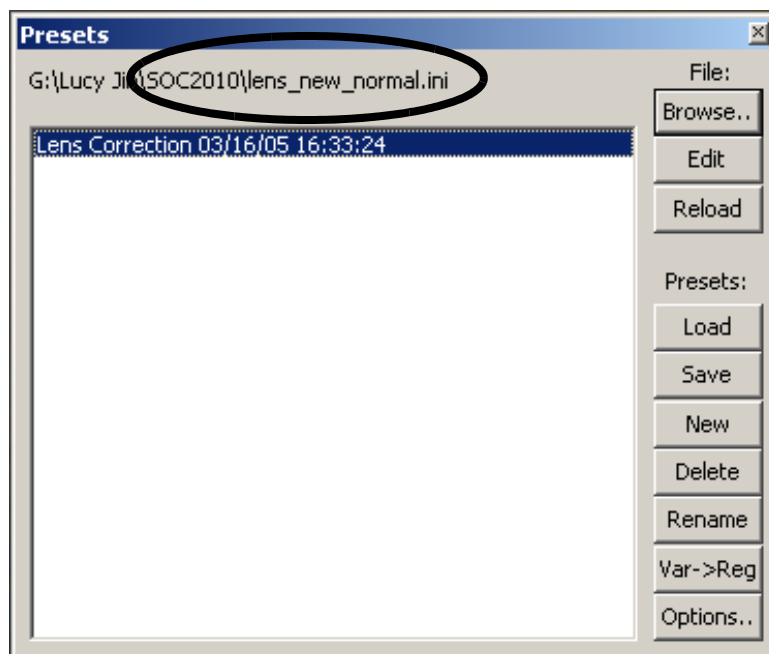
Verification

For a thorough verification of lens corrected image results, the following steps are recommended:

1. Place the MT9D111 camera module in front of flat and uniformly illuminated targets for inspection. The following scenarios are suggested: Daylight (6500K), Incandescent (2850K), Cool White Florescent (5300K).
2. Load DevWare and reset all registers (this enables AWB, Color correction, AE, Gamma Correction).
3. Open Preset window, click on "Browse..." and locate the lens correction.ini file.
4. For each of the scenario, look at the picture for visual inspection and the intensity graph to see if a flat curve for each color (R,G,B) is obtained.
5. Point the camera module at a well-lit scenery and inspect the picture as a last check on image quality.

6. If needed, repeat the Lens correction procedure to obtain improved register setting values and repeat the verification process to check image quality.

Figure 28: Locating the Lens Correction.ini File



Related Register List

```

REG=2, 0x80 //LENS_CORRECTION_CONTROL
REG=2, 0x81 //ZONE_BOUNDS_X1_X2
REG=2, 0x82 //ZONE_BOUNDS_X0_X3
REG=2, 0x83 //ZONE_BOUNDS_X4_X5
REG=2, 0x84 //ZONE_BOUNDS_Y1_Y2
REG=2, 0x85 //ZONE_BOUNDS_Y0_Y3
REG=2, 0x86 //ZONE_BOUNDS_Y4_Y5
REG=2, 0x87 //CENTER_OFFSET
REG=2, 0x88 //FX_RED
REG=2, 0x89 //FX_GREEN
REG=2, 0x8A //FX_BLUE
REG=2, 0x8B //FY_RED
REG=2, 0x8C //FY_GREEN
REG=2, 0x8D //FY_BLUE
REG=2, 0x8E //DF_DX_RED
REG=2, 0x8F //DF_DX_GREEN

```

```
REG=2, 0x90 //DF_DX_BLUE
REG=2, 0x91 //DF_DY_RED
REG=2, 0x92 //DF_DY_GREEN
REG=2, 0x93 //DF_DY_BLUE
REG=2, 0x94 //SECOND_DERIV_ZONE_0_RED
REG=2, 0x95 //SECOND_DERIV_ZONE_0_GREEN
REG=2, 0x96 //SECOND_DERIV_ZONE_0_BLUE
REG=2, 0x97 //SECOND_DERIV_ZONE_1_RED
REG=2, 0x98 //SECOND_DERIV_ZONE_1_GREEN
REG=2, 0x99 //SECOND_DERIV_ZONE_1_BLUE
REG=2, 0x9A //SECOND_DERIV_ZONE_2_RED
REG=2, 0x9B //SECOND_DERIV_ZONE_2_GREEN
REG=2, 0x9C //SECOND_DERIV_ZONE_2_BLUE
REG=2, 0x9D //SECOND_DERIV_ZONE_3_RED
REG=2, 0x9E //SECOND_DERIV_ZONE_3_GREEN
REG=2, 0x9F //SECOND_DERIV_ZONE_3_BLUE
REG=2, 0xA0 //SECOND_DERIV_ZONE_4_RED
REG=2, 0xA1 //SECOND_DERIV_ZONE_4_GREEN
REG=2, 0xA2 //SECOND_DERIV_ZONE_4_BLUE
REG=2, 0xA3 //SECOND_DERIV_ZONE_5_RED
REG=2, 0xA4 //SECOND_DERIV_ZONE_5_GREEN
REG=2, 0xA5 //SECOND_DERIV_ZONE_5_BLUE
REG=2, 0xA6 //SECOND_DERIV_ZONE_6_RED
REG=2, 0xA7 //SECOND_DERIV_ZONE_6_GREEN
REG=2, 0xA8 //SECOND_DERIV_ZONE_6_BLUE
REG=2, 0xA9 //SECOND_DERIV_ZONE_7_RED
REG=2, 0xAA //SECOND_DERIV_ZONE_7_GREEN
REG=2, 0xAB //SECOND_DERIV_ZONE_7_BLUE
REG=2, 0xAC //X2_FACTORS
REG=2, 0xAD //GLOBAL_OFFSET_FXY_FUNCTION
REG=2, 0xAE //K_FACTOR_IN_K_FX_FY
STATE=Lens Correction Center X, 800
STATE=Lens Correction Center Y, 600
BITFIELD=1, 0x08 //LENS_CORRECTION
```


Lens Shading and Correction FAQs

- Where are the lens shading correction control registers located?
- Can the lens shading calibration be used for AF calibration?

Where are the lens shading correction control registers located?

The lens shading correction control register is located in R128:2. The lens shading feature can be turned on/off by R8:1[2]. Other lens correction settings can be found in R129:2 to R174:2. For more details on the registers, refer to the latest MT9D111 data sheet.

Can the lens shading calibration be used for AF calibration?

Our lens shading calibrations are for only one position as there is only one set of values. This may introduce some minor deviation from ideal correction values if the focus movement causes the field of view to change, but it is likely undetectable.

Auto Exposure

Overview

The auto exposure (AE) algorithm performs automatic adjustments of the image brightness by controlling the exposure time and analog gains of the sensor core as well as the digital gains applied to the image.

Two auto exposure algorithm modes are available:

- preview
- scene evaluative

Auto exposure is implemented by means of a firmware driver that analyzes image statistics collected by exposure measurement engine, decides the best exposure and gain settings, and programs the sensor core and color pipeline accordingly. The measurement engine subdivides the image into 16 windows organized as 16 programmable equal-size rectangular windows forming a 4 x 4 grid.

Preview Mode

In preview mode, 16 windows are combined in two segments: central and peripheral. The central segment includes the four central windows. All remaining windows belong to the peripheral segment. Scene brightness is calculated as average luma in each segment taken with certain weights. The variable `ae.weights[3:0]` specifies the central zone weight, and `ae.weights[7:4]` specifies the peripheral zone weight. The preview exposure mode is activated during preview or video capture. It relies on the AE measurement engine that tracks the speed and amplitude of the change of the overall luminance in the selected windows of the image. The backlight compensation is achieved by weighing the luminance in the center of the image higher than the luminance on the periphery. Other algorithm features include the rejection of fast fluctuations in illumination (time averaging), control of the response speed, and control of the sensitivity to the small changes. While the default settings are adequate in most situations, the user can program the target brightness, the measurement window, and other parameters described above.

Scene Evaluative Mode

A scene evaluative AE algorithm is available for use in “snapshot” mode. The algorithm performs scene analysis and classification with respect to its brightness, contrast and composure, and then decides to increase, decrease or keep original exposure target. It makes the most difference for backlight and bright outdoor conditions.

AE Sport Mode

Digital sensor “sport mode” is essentially reducing motion-blur by limiting the exposure time to a small amount while allowing increased gain to compensate. The MT9D111 AE algorithm allows for advanced control over its parameters.

Based on the “Gain vs. Exposure” in the specification sheet, the host can upload register values to limit the exposure time. These limits may be uploaded to `ae.IndexTH23` or `ae.maxIndex`. Using `ae.IndexTH23` allows for slower shutter speeds if the exposure/gain limits are insufficient to capture the scene. Using `ae.maxIndex` to limit the exposure time sets the absolute AE limit, and thus low-light scenes will appear dark.

The value of either the `ae.maxIndex` or `ae.IndexTH23` is the integer multiple of the 50 or 60Hz period respectively, (depending on current flicker frequency) in the units of # of sensor lines. For example, if the required sport shutter speed is less than 1/100 seconds then, taking the longest of the two flicker periods 50Hz (rectified = 100Hz), the maximum AE index should be set to 1. Issue a REFRESH Sequencer command after uploading new driver parameters for them to take effect.

How to Calibrate the AE Exposure Value (EV) Reference

The AE driver translates the luminance data in to EV for some decisions (e.g., `seq.sharedParams.FlashTH`). This translation must be calibrated to ensure accurate results. This calibration should occur after the lens module has been calibrated for roll-off.

To calibrate:

- place an 18 percent reflectance grey card in full sunlight
- point the camera at the card so that the entire frame is filled with the grey card image
- allow the AE algorithm to settle on a new exposure time
- change the value of `ae.mmShiftEV` until the `ae.mmMeanEV` variable shows a value of 15 (EV)

How to Modify the Image Brightness

The appearance of the final image can be adjusted by a variety of controls each with a different specific effect on brightness.

When using the auto exposure driver, the target luminance of the MT9D111 can easily be changed by modifying the variable `ae.Target`. This causes the AE to select a different exposure time/gain combination to capture a brighter or dimmer image histogram.

To attain a relative exposure value (EV) number from the `ae.Target` value, use the relation: $\text{relative EV} = \text{LOG}(\text{ae.Target}/\text{reference ae.Target})/\text{LOG}(2)$ where each log is base-10. Reference `ae.Target` is the normal (EV=0) operating target, and `ae.Target` is the new position. For example, if the normal `ae.Target` is 75, then to get an EV value of +1, the `ae.Target` should be set to 150.

Modifying the gamma curve characteristics is another method to affect the final image's brightness. These controls are found in the mode driver's `gam_cont_A/B` variables (see data sheet and/or "Gamma and Contrast"). A preloaded gamma table can be selected or a user-defined gamma table may be uploaded. Using the gamma table to affect brightness allows brightening some shades in the image while leaving others as it is. This maintains the image's dynamic range while boosting the overall brightness.

The mode-driver variables `y_rgb_offset_A/B` (these overwrite hardware register 0xBF:1) allow for an offset to be applied to the image data. Changing these values alters the apparent brightness whether the system is outputting RGB data or YUV data. This adjustment only shifts the image histogram data and is not a good option for increasing the image quality.

How to Speed Up and Slow Down AE Adjustments

Three variables in the AE driver influence the AE speed:

- `ae.SkipFrames`—Specifies how many frames AE driver has to skip between AE register calculations. Larger numbers slow the AE.

- **ae.JumpDivisor**—Specifies how much luminance that the AE jumps in one calculation. 1: full distance, 2: half-distance, 3: 1/3rd of the distance, and so on. The smaller the number, the faster the AE (1 is minimal number). The bigger the number, the smoother the AE. (The expected brightness after one AE step is $Y_{new} = ae.CurrentY + (ae.Target - ae.CurrentY) / ae.JumpDivisor$)
- **ae.lumaBufferSpeed**—Speed of luma buffering (32=fastest, 1=slowest). To avoid unwanted reactions of the AE on small fluctuations of scene brightness or of momentary scene changes, the AE driver uses a temporal filter for the luminance.

Values: **ae.lumaBufferSpeed** = 32, **ae.SkipFrames** = 0, and **ae.JumpDivisor**=1 specify the maximum AE speed, but the image may appear to oscillate in certain dynamic lighting conditions.

The Sequencer controls AE speed in every state by the variables:

seq.sharedParams.aeContBuff

seq.sharedParams.aeContStep

seq.haredParams.aeFastBuff

seq.sharedParams.aeFastStep

For example, to achieve the fastest AE during the "Leave Preview" state, set:

seq.sharedParams.aeFastBuff=32

seq.sharedParams.aeFastStep=1,

and select the fast AE mode for the "Leave Preview" state:

Seq.previewParLeave.ae=1

The AE requires more time to achieve a luminance target if the scene suddenly becomes bright as compared to if the scene suddenly becomes dark. To quicken the light to dark AE adjustment time, set the variable **ae.status[7]** to 1.

How to Maintain Specific Frame Rates

Specific frame rates are attained by establishing a set of image timing variables (i.e., **hblank**, **vblank**, PLL timing, extra delay, etc.). However, the integration time (also known as the shutter width) may be longer than a frame time for low light scenes. This causes the frame rate to be dependent on the lighting conditions. To avoid drop in the frame rate, the AE driver provides several optional controls.

To reject flicker, integration time is adjusted in increments of **ae.R9_step**. **ae.R9_step** specifies the duration in row-times equal to one flicker period. The AE driver sets integration time as an integer multiple of **ae.R9_step** (integration time, in rows = **ae.R9_step** * **ae.Index**) and these multiples are labeled "index." Adjusting the parameters: **ae.IndexTH23** and **ae.maxIndex** control the relationships between integration time and gain used to achieve the luminance targets. Refer to the specification sheet for a more detailed diagram of the relationships between shutter width and gain. (If the AE index control variables are changed, the sequencer command "Do Refresh" (**seq.cmd=5**) must be called to activate the new settings.)

How to Use Manual Exposure and Manual Gain

If the AE driver is active (`seq.mode[0] = 1`), then the user cannot change some of the registers related to auto exposure (e.g., R9:0–shutter width, R12:0–shutter delay, R43–47:0–analog gains, R78:1, R106–110:1–digital gains, etc.) because the AE driver continuously overwrites them. The user can manually set the AE parameters in two configurations:

1. Disable the microcontroller (see “Using the Test Patterns” on page 146). Once disabled, the sensor and image processing registers may be set without being overwritten by the firmware.
2. Disable the AE driver (set to manual mode in the sequencer states), and change the AE driver's variables to affect the exposure time and the gain (see `ae.R9`, `ae.VirtGain`, `ae.DGainAE1`, `ae.DGainAE2`).

Auto Exposure FAQs

- To implement EV shift, the target exposure must be changed, is there a register to set for over or under exposure?
- Can we use single AE measurement window, whose size could be freely programmed?
- Manual exposure time & ISO setting. Is it possible to override the auto exposure and set the integration time and analog gain manually?
- Can we lock integration time and analog gain to their current values? Can we read the values of the integration time and analog gain from the sensor and set them manually afterwards?

To implement EV shift, the target exposure must be changed, is there a register to set for over or under exposure?

There is a variable named `ae.target` under the AE driver (ID=2) for the customer to modify. Here is a description from the data sheet that explains how the MCU variables are accessed:

Microcontroller variables are similar to two-wire serial interface registers, except that they are located in MCU memory. Variables are accessed by specifying their address in R198:1 and reading/writing the value to R200:1. Variables can be accessed as 8-bit (byte) and 16-bit (word) at a time. Variable address can be specified as physical or logical. Use logical address to configure driver variables. A logical address consists of a driver ID (0 = monitor, 1 = sequencer, etc.) and an offset into the driver data structure.

Can we use single AE measurement window, whose size could be freely programmed?

Yes, you can use one window (read register which specifies average luma in this window and use it as current luma. Or average all 16 windows and consider it as one big window). This is the way how AE works in preview mode. You program size and position for Top/Left window, another window positions are calculated automatically making grid 4 x 4.

Manual exposure time & ISO setting. Is it possible to override the auto exposure and set the integration time and analog gain manually?

By disabling AE and AWB, integration time and analog gains will be frozen, then they can be modified.

Can we lock integration time and analog gain to their current values? Can we read the values of the integration time and analog gain from the sensor and set them manually afterwards?

By disabling the AE and AWB the integration time and analog gains will be frozen and can then be read and written to manually in the sensor core.

Flicker Avoidance

Background

Most low-power fluorescent lights flicker quickly on and off to reduce power consumption. While this fast flickering is marginally detectable by the human eye, it is very noticeable in digital images. This is because the flicker period of the light source is very close to the range of digital images' exposure times.

Many CMOS sensors use a "rolling shutter" readout mechanism that greatly improves sensor data readout times. This allows pixel data to be read out much sooner than other methods that wait until the entire exposure is complete before reading out the first pixel data. The rolling shutter mechanism exposes a range of pixel rows at a time. This range of exposed pixels starts at the top of the image and then "rolls" down to the bottom during the exposure period of the frame. As each pixel row completes its exposure, it is ready to be read out. If the light source oscillates (flickers) during this rolling shutter exposure period, the image appears to have alternating light and dark horizontal bands.

If the sensor uses the traditional snapshot readout mechanism, in which all pixels are exposed at the same time and then the pixel data is read out, then the image may appear overexposed or underexposed due to light fluctuations from the flickering light source.

The rate of light flicker is most often either 100Hz or 120Hz; the value is determined by the adopted power specifications of different nations around the world.

To avoid this flicker effect, the exposure times must be multiples of the light source flicker periods. For example, in a scene lit by 120Hz lighting, the available exposure times are 8.3ms, 16.67ms, 25ms, 33.33ms, etc. (The need for an exposure time less than 8.3ms under artificial light is extremely rare.)

The camera designer must first detect whether there is a flickering light source in the scene, and if so, determine its flickering frequency. In this case, the auto exposure must limit the integration time to an integer multiple of the light's flicker period.

By default, the MT9D111 does all of this automatically, ensuring that all exposure times avoid any noticeable light flicker in the scene. The MT9D111 auto exposure algorithm is *always* setting exposure times to be integer multipliers of *either* 100Hz or 120Hz. The flicker detection microcontroller driver keeps monitoring the incoming frames to detect whether the scene's lighting has changed to the other of the two light sources.

How to Use the Flicker Detection Driver

The flicker detection hardware registers are located between 0x7A:1 and 0x7D:1, but these are merely statistical controls used by the flicker detection driver (ID=4) in the microcontroller code and their values should not be changed.

The intelligence and adjustability of the flicker detection algorithm resides in the microcontroller code (ID=4). The flicker detection driver's actions may be controlled by the sequencer driver (ID=1). For each of the sequencer's programmable states (preview enter, preview, preview leave, capture; all variables' names end in ".fd"), the flicker detection driver can be set to off (0), continuous (1), or manual (2).

Upon setting these values, the sequencer must be commanded to refresh the state parameters by issuing a "refresh mode" command (sequencer.cmd = 6) for the settings to be reflected in the image stream. Changing the value of the flicker detection's "mode" variable is not recommended because the sequencer driver, upon the next state change, will likely overwrite this value.

Off (0) mode completely disables the flicker detection and the auto exposure driver uses the current value stored in the R9_step as the exposure step size.

Continuous (1) mode constantly analyzes the image statistics to detect whether the lighting source has changed from the current frequency to the other frequency. If it detects the change, it uploads the new flicker period step size to the auto exposure driver (R9_step) so that all auto exposure times are integer multipliers of this period.

Manual (2) mode disables the automatic analysis of the light source, but allows the user to alter the flicker detection driver variable “mode” bit 6 to select the light source manually. Once changed, the new value, at either R9_step50 or R9step60, is uploaded to the auto exposure driver's R9_step variable.

To set up the flicker detection driver, first know the time required for each sensor line to read out. Then, calculate how many lines is in a 100Hz flicker period and a 120Hz flicker period. For example, if the line time is 75µsec, then the 120Hz (for 60Hz lighting sources) flicker period is 111 lines ($1/(120 * 75\mu\text{sec})$).

The resulting values for 50Hz and 60Hz flicker period lines should be uploaded to the flicker detection driver's R9_step50 and R9_step60 variables, respectively.

Next, calculate the search window for the flicker detection window for each lighting source. This is the flicker period search range that the flicker detection driver uses (in units of lines/5) to identify whether flickering is occurring in the scene. For most applications, the values are simply the flicker period (calculated above) divided by 5, plus or minus 1. For example, if the flicker period lines were calculated to be 111, then the flicker search period is from 21 (“f1”) to 23 (“f2”) using the formula: $\text{lines}/5 \pm 1$. These search period values should be uploaded to the variables search_f1_50, search_f2_50, search_f1_60, and search_f2_60.

These driver variable settings should be sufficient for reliable flicker detection. However, if there is a further need to fine tune the algorithm, some additional (advanced) variables are available for adjustment.

stat_min and stat_max help control the sensitivity of the algorithm to flickering light in a scene. If the algorithm detects stat_min instances of flickering in stat_max measurements, then the flicker detection driver responds by changing the flickering frequency (if different than the existing detected frequency). Altering the ratio between the two values affects the sensitivity of the algorithm and the time required to decide whether to change the detected light flicker period.

Increasing ae.SkipFrame increases the number of frames that are skipped between each measurement. However, because the algorithm depends on detecting differences in light intensity given a constant scene, setting this value too high may allow the camera user to change the scene framing between flicker detection frames, making this algorithm less effective overall.

How to Fine Tune the Anti-flick Driver Setting

1. Use the Register Wizard to generate the basic anti-flicker setting for the specific timing conditions, including the following:


```
VAR8 = 4, 8, 0x0F      //search_f1_50 = 15
VAR8 = 4, 9, 0x11      //search_f2_50 = 17
VAR8 = 4, 10, 0x12     //search_f1_60 = 18
VAR8 = 4, 11, 0x14     //search_f2_60 = 20
VAR = 4, 17, 0x0053    //R9_Step_60_A = 83
VAR = 4, 19, 0x0063    //R9_Step_50_A = 99
VAR = 4, 21, 0x0053    //R9_Step_60_B = 83
VAR = 4, 23, 0x0063    //R9_Step_50_B = 99
```
2. Fine tune the search window for 50Hz.
 Take the search_f1_50 value and minus 1, and plus 1 for search_f2_50.
3. Fine tune the search window for 60Hz in the same way as for 50Hz.
4. Increase the sensitivity of the flicker detection.
 stat_min and stat_max help control the sensitivity of the algorithm to flickering light in a scene. If the algorithm detects stat_min instances of flickering in stat_max measurements, then the flicker detection driver responds by changing the flickering frequency. Altering the ratio between the two values affects the sensitivity of the algorithm and the time required to decide whether to change the detected light flicker period. For most applications, the following should produce enough sensitivity:


```
VAR8=4, 0x0D, 0x02     // FD_STAT_MIN
VAR8=4, 0x0E, 0x05     // FD_STAT_MAX
```
5. Increase the signal strength sensitivity of the flicker detection.
 MinAmplitude specifies the signal threshold, below which signals are ignored. The following should be suitable for most cases:


```
VAR8=4, 0x10, 0x01     // FD_MIN_AMPLITUDE
```
6. Call REFRESH command to make the above changes effective.

How to Verify the Setting

1. Verify the setting for 50Hz manual mode.
 - a. Connect the light source with 50Hz power supply.
 - b. Load the following to enable 50Hz manual mode.


```
VAR8=4, 0x04, 0xC0      // FD_MODE
VAR8=1, 0x03, 0x05      // SEQ_CMD
```
 - c. Check the output image to see if flicker disappears.
 - d. Monitor fd.mode[5] to see if it goes to "1."
2. Verify the setting for 60Hz manual mode.
 - a. Connect the light source with 60Hz power supply.
 - b. Load the following to enable 60Hz manual mode.


```
VAR8=4, 0x04, 0x80      // FD_MODE
VAR8=1, 0x03, 0x05      // SEQ_CMD
```
 - c. Check the output image to see if flicker disappears.
 - d. Monitor fd.mode[5] to see if it goes to "0."

3. Verify the setting for auto mode.
 - a. Connect the light source with 60Hz power supply.
 - b. Load the following to enable auto mode.
`VAR8=4, 0x04, 0x02 // FD_MODE`
`VAR8=1, 0x03, 0x05 // SEQ_CMD`
 - c. Check the output image to see if flicker disappears.
 - d. Monitor fd.mode[5] to see if it goes to "0."
 - e. Switch the power supply to 50Hz.
 - f. Check the output image to see if flicker disappears.
 - g. Monitor fd.mode[5] to see if it goes to "1."

How to Modify the Setting for Specific Applications

If any of the above verifications fail, fine tune the setting again to increase the sensitivity and search window. During this process, use debug mode to assist the tuning.

```
VAR8=4, 0x04, 0x10      // FD_MODE
REG=1, 0xA3, 0x0242      // HIGHLIGHT_COLOR
```

After loading the above, on the left top corner, the current flick frequency will appear.

Flicker Avoidance FAQs

- What are the instructions to enable flicker detection (and presumably suppression) for the rev 1 MT9D111 samples?

What are the instructions to enable flicker detection (and presumably suppression) for the rev 1 MT9D111 samples?

The flicker detection feature is enabled by default in context A (preview). It is capable of detecting 50Hz and 60Hz frequencies. To turn it on or off, the following settings can be used:

- ID=1, Offset=2: seq.mode[1] = 1 for flicker detection enable; = 0 to disable

If the above is turned off, upon the REFRESH command, the flicker detection is turned back on, unless the feature is specifically turned off in each of these states:

- ID=1, Offset=35 : seq.previewParEnter.fd (FD mode in the PreviewEnter state)
- ID=1, Offset=42 : seq.previewPar.fd (FD mode in the Preview state)
- ID=1, Offset=49 : seq.previewParLeave.fd (FD mode in the PreviewLeave state)
- ID=1, Offset=56 : seq.captureParEnter.fd (FD mode in the CaptureEnter state)

Color Correction

Auto White Balance

In order to achieve good color rendition and color saturation, interpolated colors of all pixels are subjected to color correction. The color correction is a linear transformation of the image with a 3 x 3 color correction matrix. The optimal values of the correction matrix elements depend on the spectrum of light incident on the sensor. They can be either programmed by the user or automatically selected by the auto white balance (AWB) algorithm.


The AWB algorithm is designed to compensate the effects of the changing spectra of the scene illumination on the quality of the color rendition. This sophisticated algorithm consists of two major parts:

- a measurement engine performing statistical analysis of the image (R48–50:1)
- and a firmware driver performing the selection of the optimal color correction matrix, digital, and analog gains

The driver keeps the values for the analog gain ratio of two matrices corresponding to two opposite illuminations red-rich (incandescent) and blue-rich (daylight). The variables `awb.ccmL[0]–awb.ccmL[10]` keep the values for left (incandescent) matrix. The variables `awb.ccmRL[0]–awb.ccmRL[10]` keep the difference between the right (daylight) and the left matrices. The AWB driver analyzes the measurement engine data and sets appropriate digital AWB gains (`awb.GainR`, `awb.GainG`, `awb.GainB`) and the matrix position (`awb.CCMposition`). The matrix position defines current matrix coefficients and the analog gain ratio (`awb.ccm` array).

$$\text{Awb.ccm}[i] = \text{awb.ccmL} + \text{awb.ccmRL} \times \left(\frac{\text{awb.CCMposition}}{127} \right) \quad (\text{EQ 1})$$

(EQ 2)

$$\text{Total WB gain} = \frac{\text{Red Gain (R45:0)}}{\text{GlobalGain (R47:0)}} \times \text{awb.GainR}$$


How to Change the Color Saturation

The variable `awb.saturation` defines the color saturation. `awb.saturation = 128` corresponds to 100 percent of saturation when `awb.ccm` is calculated in EQ 1. `awb.saturation = 0` corresponds to unit matrix. Any other value of `awb.saturation` corresponds to a matrix which is a linear interpolation between 100 percent saturation and the unit CCM.

To set a 30-percent saturated matrix:

- enable AWB `seq.mode[2]=1`
- set `awb.saturation = 30 * (128 / 100) = 38`

How to Speed Up/Slow Down AWB

Two variables are responsible for the AWB speed:

1. `awb.GainBufferSpeed`—Speed of the AWB digital gains buffering (32 = fastest, 1 = slowest)
2. `awb.JumpDivisor`—specifies how much the AWB parameters (gains, CCM, etc.) are traversed in one jump (1 = whole way, 2 = half way, 3 = 1/3rd of the way, and so on)

The smaller the number the faster AWB (1 is minimal number). The sequencer controls the AWB speed in every state by these variables:

- `seq.sharedParams.awbContBuff`
- `seq.sharedParams.awbContStep`
- `seq.sharedParams.awbFastBuff`
- `seq.sharedParams.awbFastStep`

For example, to make the AWB as fast as possible on “Leave Preview,” set the fastest parameters for fast AWB mode:

1. `seq.sharedParams.awbFastBuff=32`
2. `seq.sharedParams.awbFastStep=1`

Then select the fast AWB mode for the “Leave Preview” state.

3. `seq.previewParLeave.awb=1`

How to use a Static CCM

There are two ways to use a static CCM:

1. Use default matrices
 - a. turn AWB On
 - i. `seq.mode[2]=1`
 - b. set digital WB gain to 1
 - ii. `awb.mode[5]=1`
 - c. set WB position to select the desired CCM.
 - iii. `awb.CCMposition=0` corresponds incandescent CCM
 - iv. `awb.CCMposition=127` corresponds daylight CCM
 - d. `CCMPosition` is decided from B/G
 If `awb.GainB` is outside the range of `SteadyBGainOutMin` and `SteadyBGainOutMax`, the `CCMPosition` would be changed, and analog gain is adjusted. `CCMPosition` keeps adjusting until `awb.GainB` is in range of `SteadyBGainInMin` and `SteadyBGainInMax`. If CCM Position reaches one end, but AWB is not satisfied, digital gains would be increased/decreased until it reaches `GainMax/GainMin`.
2. Download user defined CCM
 - a. turn AWB On
 - i. `seq.mode[2]=1;`
 - b. set digital WB gain to 1
 - ii. `awb.mode[5]=1`
 - c. write coefficients of new CCM into variables `awb.ccmL[0] – awb.ccmL[10]` (256 corresponds to 1.0)
 - iii. `awb.ccmL[0] = CCM11*256`
 - iv. `awb.ccmL[1] = CCM12*256`
 - ...
 - v. `awb.ccmL[8] = CCM33*256`
 - vi. `awb.ccmL[9] = RedAnalogGain/GreenAnalogGain *32`
 - vii. `awb.ccmL[10] = BlueAnalogGain/GreenAnalogGain *32`

- d. set variables awb.ccmRL[0] – awb.ccmRL[10] to 0
- viii. awb.ccmRL[0] = 0
- ...
- ix. awb.ccmRL[10] = 0

How to Perform Color Calibration

1. Setup

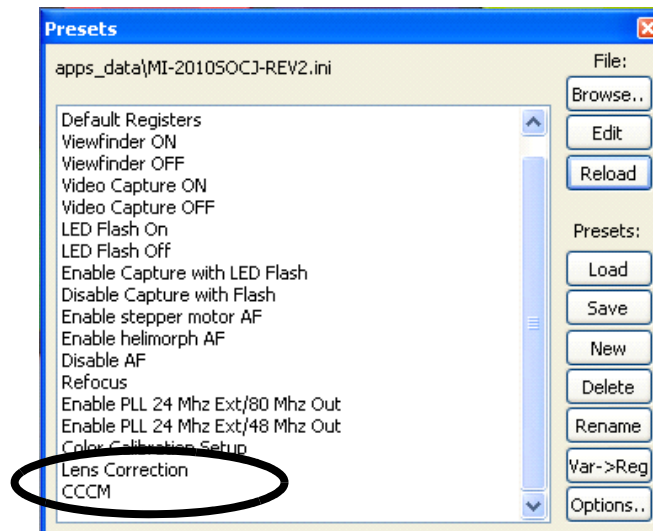
Hardware

- a. Make sure the lens and sensor module are completely shielded from external light entering the module from the side. If needed, shield the lens module from external light using black tape.
- b. Place the color rendition chart in the middle of the image screen to avoid corner color shading effect to the chart. Make sure the entire chart is in the image screen.
- c. Turn on one of the light source (Blue Daylight at 6500 Kelvin or Red Incandescent light at 2850 Kelvin). For the procedure described below, we show the calibration for Blue Daylight first, then ask the user to repeat the process for Red light.

Software

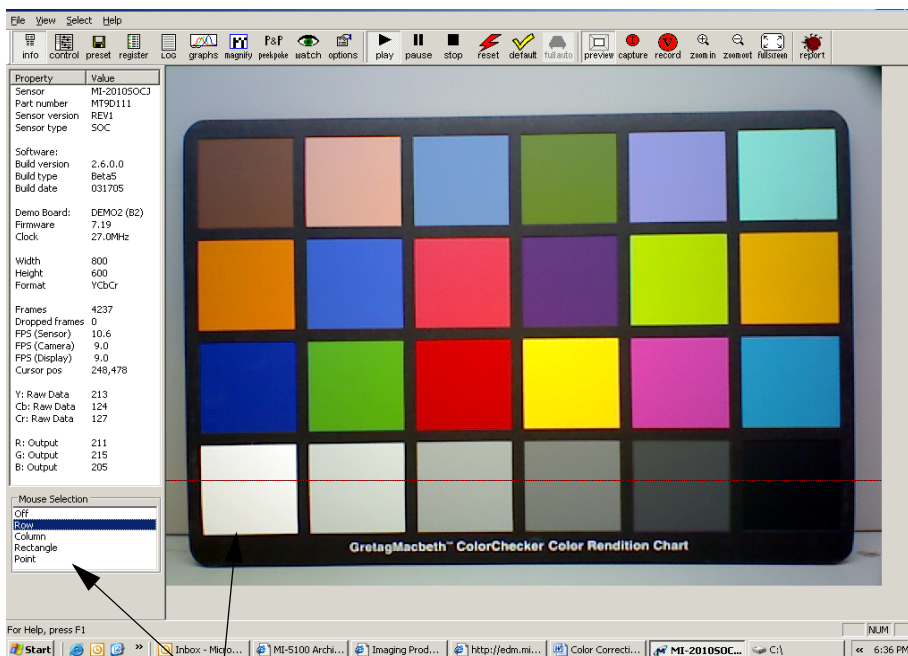
- d. Start up DevWare version 2.6 Beta 5 (or above) and camera module
- e. Do not load Register Default settings upon starting the software

2. Preset and Load



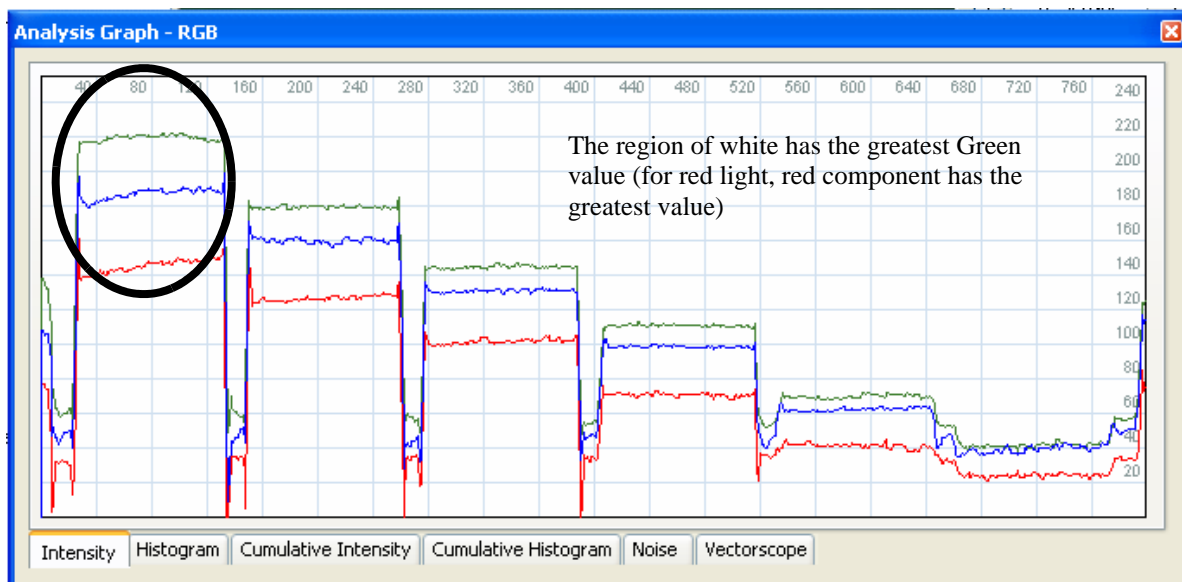
- a. Select “Color Calibration Setup” in Preset window and click on “Load”—this loads all relevant register setting that are needed to perform the tuning in DevWare.
- b. Select “Lens Correction” to load the settings for lens calibration. The user should calibrate the lens before calibrating color. For procedure on lens calibration, see “Lens Shading and Correction” on page 33.

3. Calibration in DevWare



Since the light hits vertically from top, select the ROW line which comes across the top of the white color square. Thus, the maximum Green needed for white balance can be obtained.

- a. Open the Cumulative Intensity Graph and adjust the shutter width so the Green in white patch is at 220



7) Mode 9) JPEG 11) HG FPGA
 1) Seq 2) AE 3) AWB 4) FD 5) AF 6) AFM
 0: Sensor 1: SOC1 2: SOC2 6: Analog 0) Mon

Address (hex)	Description	Value (hex)
0x0000	Chip Version	0x1519
0x0001	Row Window Start	0x001C
0x0002	Column Window Start	0x003C
0x0003	Row Width	0x04B0
0x0004	Column Width	0x0640
0x0005	Horizontal Blanking B	0x0204
0x0006	Vertical Blanking B	0x002F
0x0007	Horizontal Blanking A	0x00FE
0x0008	Vertical Blanking A	0x000C
0x0009	Shutter Width	0x0231
0x000A	Extra Delay	0x0001
0x000B	Shutter Delay	0x0000
0x000C	Reset	0x0000
0x000D	Frame Valid Control	0x0000
0x000E	Read Mode B	0x0303
0x000F	0: Mirror rows	0x0001
0x0010	1: Mirror columns	0x0001
0x0011	2-3: Row Skip Context B	0x0000
0x0012	4: Row Skip Enable Context B	0x0000
0x0013	5-6: Column Skip Context B	0x0000
0x0014	7: Column Skip Enable Context B	0x0000
0x0015	8: Over read	0x0001

Value: 0x0231 Set Refresh

Register Info: Bitmask 0xFFFF Default 0x04D0

Bitfield Info: Bitmask 0x0002 Maximum 0x0001 Default 0x0000

Reg bits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

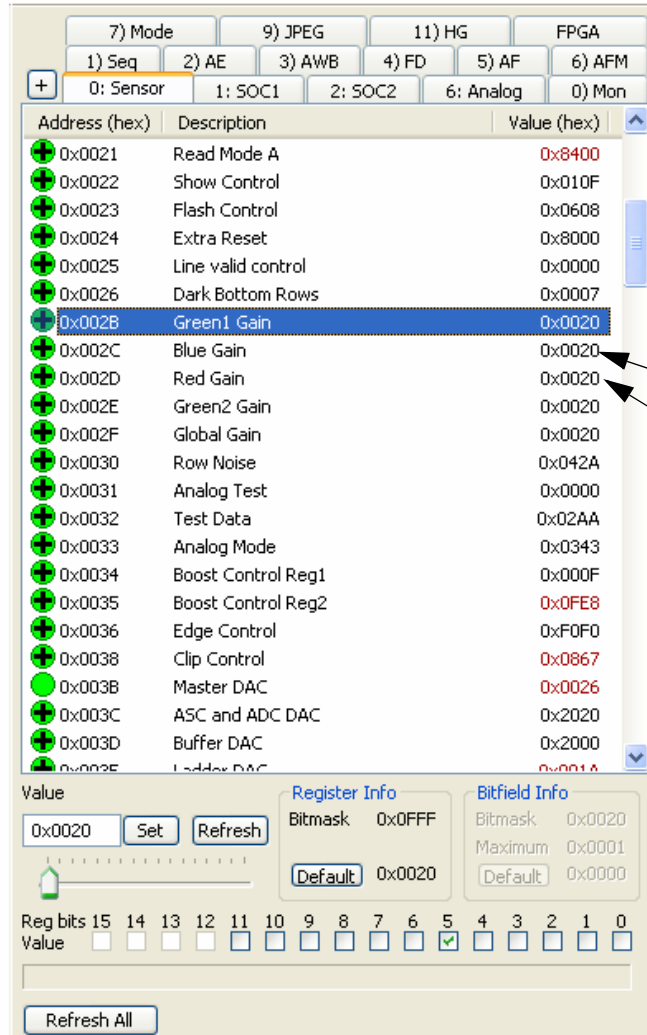
Integration time in Number of rows

Refresh All

Adjust "Shutter Width" register values so the maximum Green in the intensity graph reaches "220," as shown in the graph above.

Adjust the "Shutter Width" register by changing the binary bits of gain. Start from left to right to approximate values.

- b. Adjust the Red and Blue Gain, so that the curves overlaps each other approximately (start from Red gain and then Blue gain)

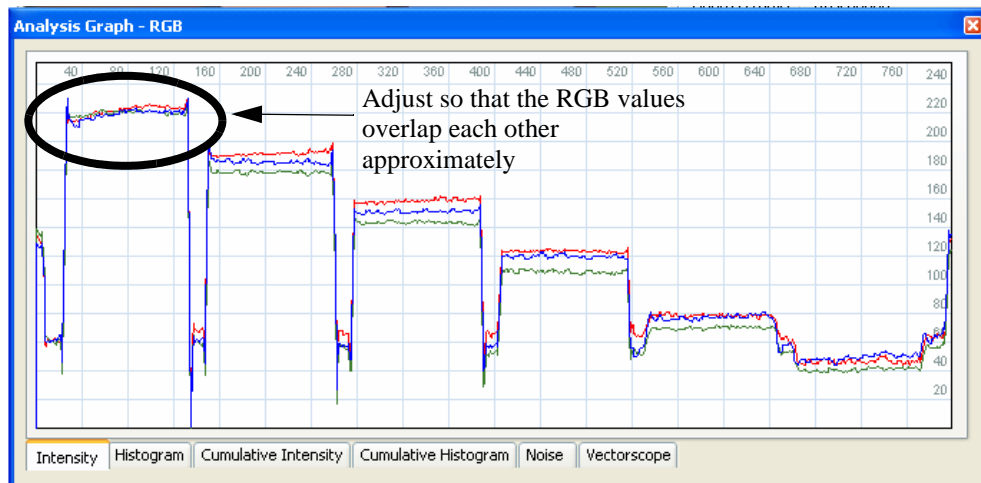


The screenshot shows the MT9D111 Register Editor interface. The '0: Sensor' tab is selected. The register list shows 'Green1 Gain' at address 0x002B with a value of 0x0020. Below the list, the 'Value' field is set to 0x0020. The 'Register Info' section shows a bitmask of 0x0FFF and a default value of 0x0020. The 'Bitfield Info' section shows a bitmask of 0x0020, a maximum value of 0x0001, and a default value of 0x0000. The 'Reg bits' section shows bits 15 through 0, with bit 5 checked.

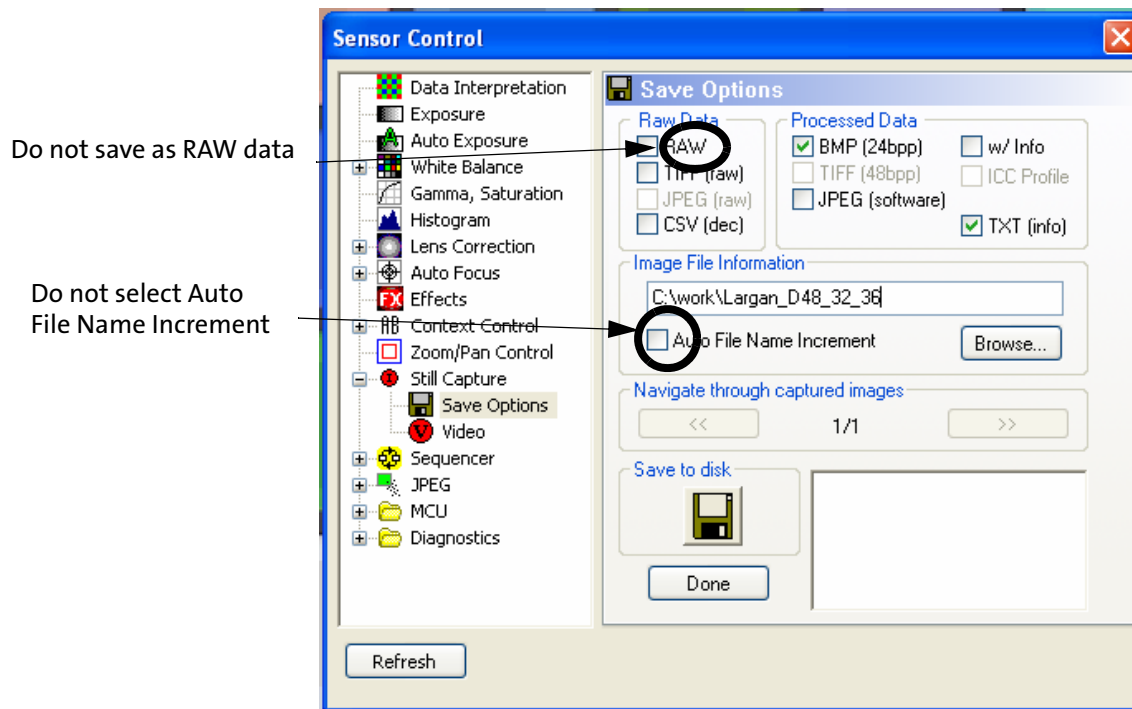
Address (hex)	Description	Value (hex)
0x0021	Read Mode A	0x8400
0x0022	Show Control	0x010F
0x0023	Flash Control	0x0608
0x0024	Extra Reset	0x8000
0x0025	Line valid control	0x0000
0x0026	Dark Bottom Rows	0x0007
0x002B	Green1 Gain	0x0020
0x002C	Blue Gain	0x0020
0x002D	Red Gain	0x0020
0x002E	Green2 Gain	0x0020
0x002F	Global Gain	0x0020
0x0030	Row Noise	0x042A
0x0031	Analog Test	0x0000
0x0032	Test Data	0x02AA
0x0033	Analog Mode	0x0343
0x0034	Boost Control Reg1	0x000F
0x0035	Boost Control Reg2	0x0FE8
0x0036	Edge Control	0xF0F0
0x0038	Clip Control	0x0867
0x003B	Master DAC	0x0026
0x003C	ASC and ADC DAC	0x2020
0x003D	Buffer DAC	0x2000
0x003E	Ladder DAC	0x001A

Start from adjusting Red gain and move to Blue gain. (Green gain should not be adjusted it should be the default 0x0020).

For red light, start from adjusting blue gain. Green gain should not be adjusted (0x0020). If Red is higher than Green, then lower Red gain so it overlaps with Green.

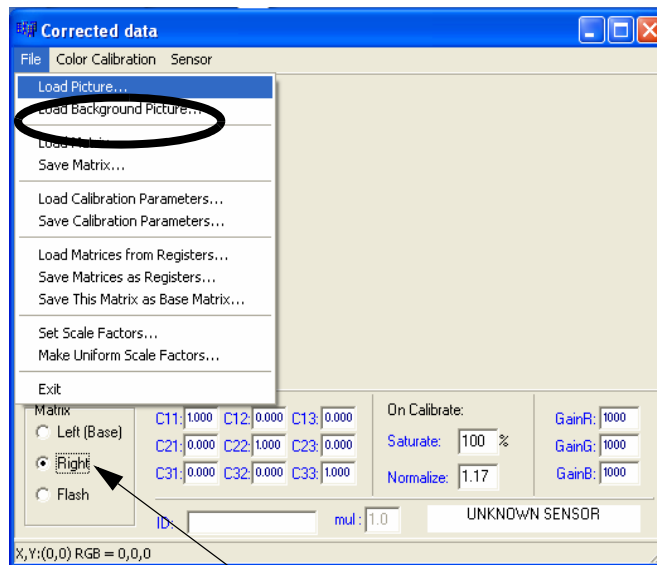


- c. Capture the picture and save it. The naming convention is important here. The image should be named as: Module ID _ Light Condition and Red Gain _ Green Gain _ Blue Gain. For the light condition field, the user should enter "D" for D65 light and "A" for A28 light. For example, in the GUI below, we show capturing an image taken by the Largan module with D65 and Red Gain = 48, Green Gain = 32, and Blue Gain = 36. This example is explained throughout this document.



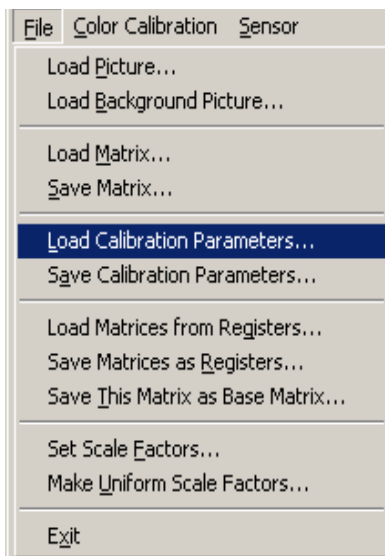
- d. With all settings staying the same, take away the Color Rendition Chart from the light box and take an image of the background. Capture this image and save it as "Largan_D48_32_36_bk". Note the "_bk" stands for background
- e. Repeat Step 3 (Calibration in DevWare) for incandescent light and save the color chart picture and the background picture as a different file name

4. Calibration of Color Correction Matrix



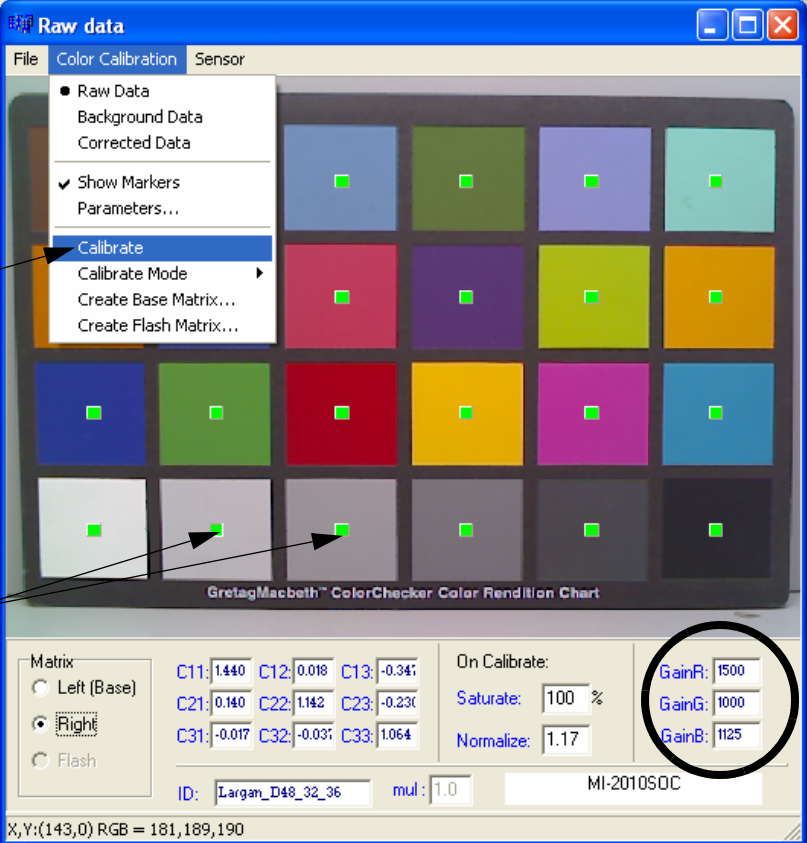
Make sure Daylight picture is for the Right matrix

- Load the 'D_48_32_36.bmp' as the "Right" Picture. Note that 'D_48_32_36_bk.bmp' is loaded automatically as the background picture
- Load the calibration target (it is located in C:\Program Files\Aptina Imaging\cccm). The target is called DAY.cal for D65 lighting and A.cal for A28 lighting



- Manipulate and adjust so that the Matrix dots are approximately located at the middle of each color square. If the GainR is less than 1,000 for Incandescent

Light, change the number to 1,000; then go to the Color Calibration and Select Calibrate



The screenshot shows the 'Raw data' window with the 'Color Calibration' tab selected. The 'File' menu is open, and the 'Calibrate' option is highlighted. A color calibration chart (GretagMacbeth ColorChecker) is displayed in the center. Below the chart, the 'Matrix' section shows the 'Right' matrix selected, with values for C11 through C33. The 'On Calibrate' section shows 'Saturate' at 100% and 'Normalize' at 1.17. The 'Gain' section shows 'GainR' at 1500, 'GainG' at 1000, and 'GainB' at 1125. The 'ID' field shows 'Largan_D48_32_36' and the 'mul' field shows '1.0'. The 'MI-2010SOC' sensor is selected.

Select Calibrate

Try to place the matrix dots at the center of the color squares

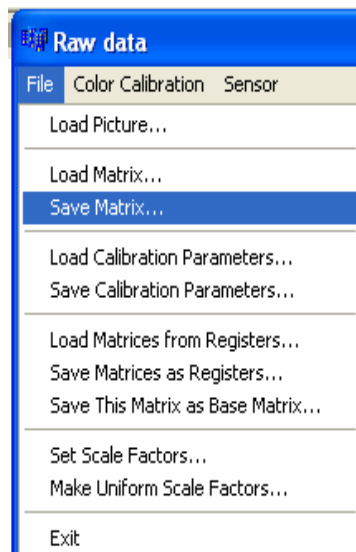
These gain ratios are automatically loaded when the picture is loaded.

GainR = 1000*(Red Gain/Green Gain)

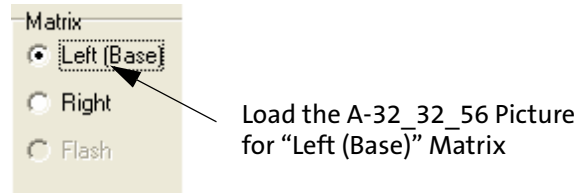
GainG = 1000*(Green Gain/Green Gain)

GainB = 1000*(Blue Gain/Green Gain)

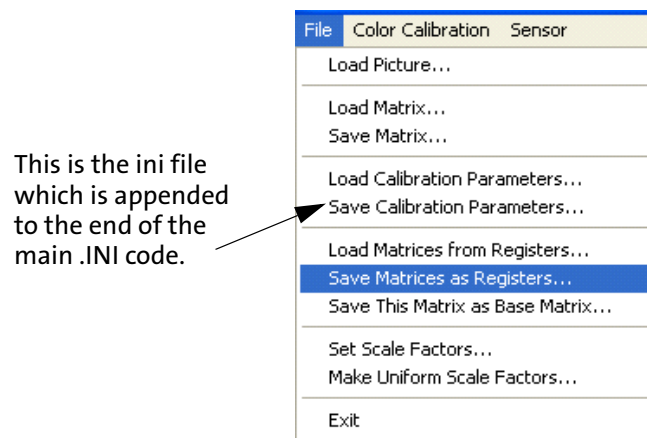
d. Save the calibrated Daylight picture as a Matrix data file



- e. After Calibrated the picture, repeat steps [a]–[d]. loading the Incandescent light picture as the “Left (Base)” Matrix

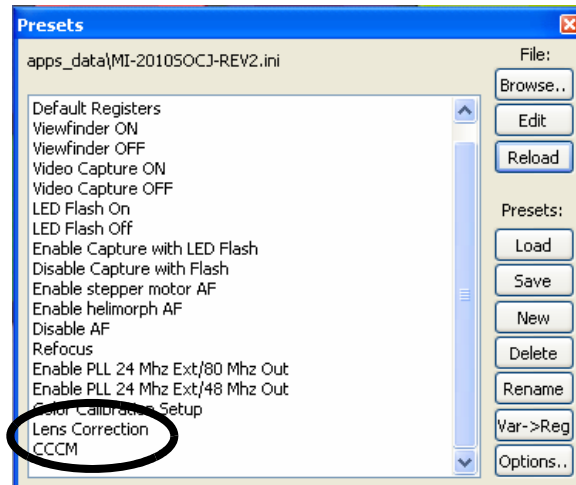


- f. After both pictures have been calibrated and the corresponding matrix data file has been saved, save a combine .ini file

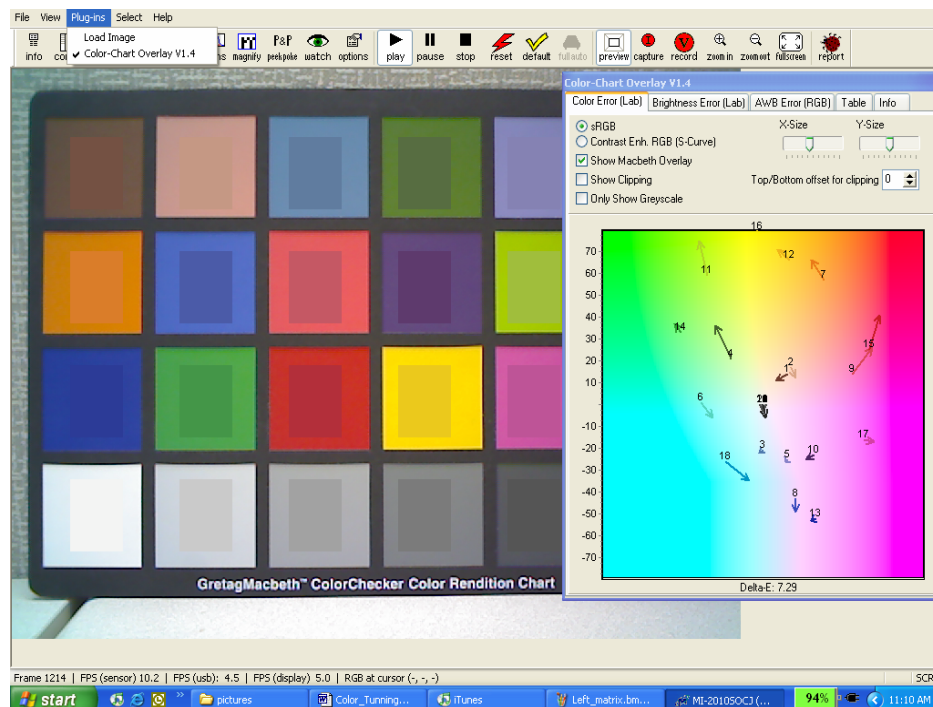


5. Verification

- a. Hardware setup is the same as described above
- b. Load DevWare and reset all registers (this enables AWB, Color Correction, AE, and Gamma Correction)
- c. Load the Lens Correction, then load CCCM setting in the Preset window



- d. Go to Plug-ins --> Color-char Overlay v1.4. Make sure the patches are right on top of the color bars in the image. If not, adjust X-Size and Y-Size to have the patches positioned
- e. The Delta-E parameter on Color-Chat Overlay V1.4 should be no more than 8.0



- f. If needed, repeat the color correction procedure to obtain improved register setting values and repeat the verification process to check image quality

Related Register List

REG=1, 0x60 //COLOR_CORR_MATRIX_SCALE_14
REG=1, 0x61 //COLOR_CORR_MATRIX_SCALE_11
REG=1, 0x62 //COLOR_CORR_MATRIX_1_2
REG=1, 0x63 //COLOR_CORR_MATRIX_3_4
REG=1, 0x64 //COLOR_CORR_MATRIX_5_6
REG=1, 0x65 //COLOR_CORR_MATRIX_7_8
REG=1, 0x66 //COLOR_CORR_MATRIX_9
REG=1, 0x6A //DIGITAL_GAIN_1_RED
REG=1, 0x6B //DIGITAL_GAIN_1_GREEN1
REG=1, 0x6C //DIGITAL_GAIN_1_GREEN2
REG=1, 0x6D //DIGITAL_GAIN_1_BLUE

Color Correction FAQs

- How can I load a new Color Correction Matrix?

How can I load a new Color Correction Matrix?

The CCM can be loaded using the AWB driver (ID=3). The offsets for the left (A) and right (daylight) CCMs and gain ratios are as follows:

- ID=3, Offset=6-26 (ccmL) contains the CCM values and gain ratios for the red-light (left color correction matrix)
- ID=3, Offset=28-48 (ccmRL) contains the delta values between the left and right matrices

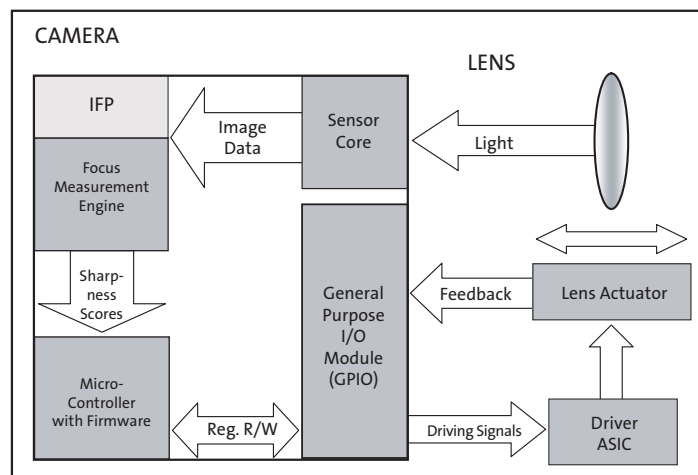
The AWB driver will automatically adjust the current CCM values (ID=3, Offset=50-70) to the proper values based on the two extreme cases above.

Auto Focus Driver

Background

This section details the auto focus (AF) drivers included in the firmware of the MT9D111 image sensor. These drivers encapsulate all firmware code developed to date by Aptina to support AF function in cameras built around the sensors.

Figure 29: Block Diagram of a Basic AF Camera Built Around the MT9D111 Image Sensor



Note: A block diagram of a basic AF camera built around the MT9D111 image sensor. It shows system components essential for auto focusing and interactions between them.

Scan Auto Focus Algorithm

Algorithm Description

The AF algorithm implemented in the MT9D111 seeks to maximize sharpness of vertical lines in the sensor's image output by guiding an external lens actuator to the position of best lens focus. The algorithm's implementation has a hardware component called focus measurement engine (FME) and a firmware component called AF driver. The algorithm is lens-actuator-independent: it provides guidance by means of an abstract 8-bit variable called logical lens position, leaving the translation of its changes into physical lens movements to a separate auto focus mechanics (AFM) driver. The AF algorithm relies on the AFM driver and the GPIO to generate digital output signals needed to move different lens actuators. The AFM driver must also correctly indicate at all times if the lens it controls is stationary or moving. This is required to prevent the AF driver from using line sharpness measurements distorted by concurrent lens motion and from issuing new commands to move the lens while previous one is still being executed.

Line sharpness measurements are performed continuously (in every frame) by the FME, which is a programmable edge-filtering module in image flow processor (IFP). The FME convolves two preprogrammed 1-dimensional digital filters with luminance (Y) data it receives row-by-row from the color interpolation module. In every interpolated image, the pixels whose Y values are used in the convolution form a rectangular block that can be arbitrarily positioned and sized, and then divided into up to 16 equal-sized sub-blocks, referred to as AF windows or zones. The absolute values of convolution results are summed separately for each filter over each of the AF windows, yielding up to 32

sums per frame. As soon as these sums or raw sharpness scores are computed, they are put in dedicated IFP registers, as are Y averages from all the AF windows. The AF driver reduces these data to one normalized sharpness score per AF window, by calculating for each window the ratio $(S_1 + S_2) / \langle Y \rangle$, where $\langle Y \rangle$ is the average Y and S_1 and S_2 are the raw sharpness scores from the 2 filters multiplied by 128. Programming of the filters into the MT9D111 includes specifying their relative weights, so each ratio can be called a weighted average of two equally normalized sharpness scores from the same AF window. In addition to unequal weighting of the filters, the AF driver permits unequal weighting of the windows, but window weights are not included in the normalized sharpness scores. Reasons for this are detailed below.

There are several motion sequences through which the AF driver can bring a lens to best focus position. An example sequence is depicted in Figure 30. All these sequences begin with a jump to a preselected start position, e.g. the infinity focus position. This jump is referred to as the first flyback. It is followed by a unidirectional series of steps that puts the lens at up to 19 preselected positions different from the start position. This series of steps is called the first scan.

Before and during this scan, the lens remains at each preselected position long enough for the AF driver to obtain valid sharpness scores. Typically, the time needed is no longer than 1 frame, but there is an option to skip 1 frame before the AF driver grabs the scores, which means that the total time spent at each position can reach 2 frames. The timing of lens movements between the preselected positions is lens-actuator-dependent and not controlled by the AF driver. Though the AF driver gives commands to move the lens, it is the AFM driver that takes care of the execution and determines how soon after each command the AF driver gets a signal to proceed. All inclined sections of the lens position (plots in Figure 30) are therefore of unknown duration—unless the discussion on the AF is narrowed to a specific use case.

Second Auto Focus Scan

The purpose of auto focus is to find the best lens position which provides the highest sharpness of the picture. A typical auto focus process is shown in the Figure 30. For the MT9D111, the process must include the 1st flyback and 1st scan, in which, the system will try to make the lens steps through its motion range and sharpness scores are calculated after each step. Then we can pick up the best focus position and move the lens to this position. However, the total step number AF can scan is up to 20, so for better resolution of the lens position, we can choose to make the second finer focus scan. What happens after the first scan and ensuing selection of best position is user-programmable. The selection make should be the one that best fits the magnitude of lens actuator hysteresis and desired lens proximity to the truly optimal position.

Defining the Second Scan

The MT9D111 is designed to use the certain registers to define the second scan. Here we note that the total scan step number, including 1st and 2nd scan, is 20, i.e. $af.numSteps + af.numSteps2 \leq 20$. Below we list the relevant registers:

Table 11: Driver Variables-Auto Exposure Driver (ID = 2)

Name	Default Value	Description
af.modeEx[5]	0	0: disable the second scan. 1: enable the second scan.
af.numSteps	10	Number of steps in the first scan.
af.numSteps2[3:0]	6	Desired number of steps in second scan, maximum 14.
af.numSteps2[7:4]	0	Actual number of steps in the second scan, calculated by the AF driver at the beginning of the scan.
af.stepSize	6	Logical step size for the second scan.

Example 1: Figure 30.

```
VAR8=5, 0x05, 0x00A0    // af.modeEx, enable the second scan
VAR8=5, 0x08, 0x0004    // af.numSteps2 = 5
VAR8=5, 0x09, 0x0004    // af.stepSize = 4
```

After the first scan, suppose the best logical position is 119, then the second scan will begin from $119 - (af.numSteps2 - 1) * af.stepSize / 2 = 119 - (4) * 4 / 2 = 119 - 8 = 111$ and then go through the steps of 111, 115, 119, 123, 127.

Example 2: even number of step 2.

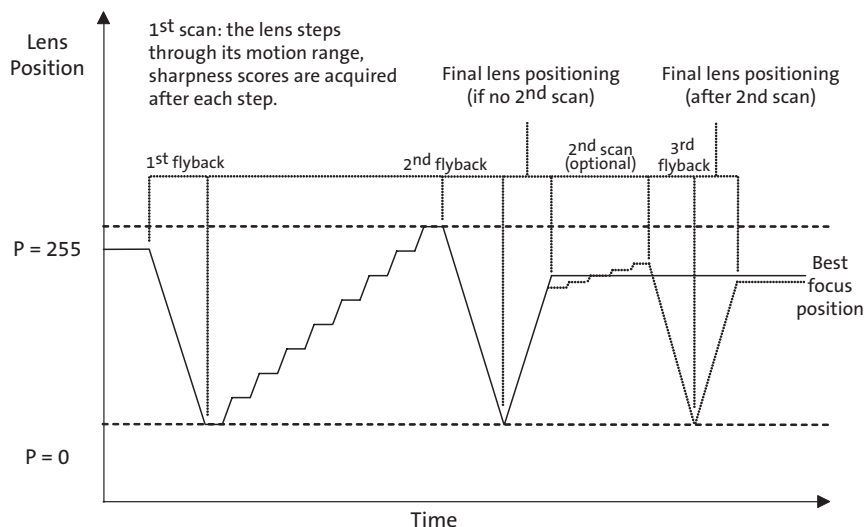
```
VAR8=5, 0x05, 0x00A0    // af.modeEx, enable the second scan
VAR8=5, 0x08, 0x0004    // af.numSteps2 = 4
VAR8=5, 0x09, 0x0004    // af.stepSize = 4
```

After the first scan, if the best logical position is 119, then the second scan will begin from $119 - (af.numSteps2 - 1) * af.stepSize / 2 = 119 - (3) * 4 / 2 = 119 - 6 = 113$ and then go through the steps of 113, 117, 121, 125.

Note: The original best position 119 is skipped.

Lens Movements and Focus

Figure 30 illustrates lens movements during a dual/triple-flyback auto focusing sequence. The sequence is an example only; it can be changed in a number of ways. Second scan, as well as second and third flyback, are optional—final lens positioning can be a direct jump from last position tried in a scan to best focus position. Number of steps in each scan, lens positions stepped through during the first scan, and step size in the second scan are all individually programmable.

Figure 30: Search for Best Focus


The first normalized score from each AF window, acquired at the start position, is stored as both the worst (minimum) and best (maximum) score for that window. These two extreme scores are then updated as the lens moves to subsequent positions and a new maximum position is memorized at every update of the maximum score. In effect, the preselected set of lens positions is scanned for maxima of the normalized sharpness scores, while information needed to validate each maximum is being collected. This information is in the difference between the maximum and the minimum of the same score. A small difference in their values indicates that the score is not sensitive to the lens position; therefore, its observed extrema are likely determined by random noise. On the other hand, if the score varies greatly with the lens position, its maximum is much more likely to be valid—close to the true sharpness maximum for the corresponding AF window. Due to these considerations, the AF driver ignores the maxima of all sharpness scores whose peak-to-trough variation is below a preset percentage threshold. The remaining maxima, if any, are sorted by position and used to build a weight histogram of the scanned positions. The histogram is built by assigning to each position the sum of weights of all AF windows whose normalized sharpness scores peaked at that position. The position with the highest weight in the histogram is then selected as the best lens position.

This method of selecting the best position may be compared to voting. The voting entities are the AF windows, that is, different image zones. Depending on the imaged scene, they may all look sharp at the same lens position or at different ones. If all the zones have equal weight, the lens position at which a simple majority of them looks sharp is voted the best. If the weights of the zones are unequal, it means that making some zones look sharp is more important than maximizing the entire sharp-looking area in the image. If there are no valid votes, because sharpness scores from all the AF windows vary too little with the lens position, the AF driver arbitrarily chooses the start position as the best.

Figure 31, Figure 32, and Figure 33 illustrate selection of the best lens position when there are several objects in imaged scene to focus on, each at a different distance from the lens. Each lens position bringing one or more of these objects into sharp focus within the AF window grid can be potentially voted the best. The actual result of the vote is determined by the extent and texture of each object and the weights of the overlying AF windows.

What happens after the first scan and ensuing selection of best position is user-programmable—the AF algorithm contains a number of ways to proceed with final lens positioning. The selection made should be the one that best fits the magnitude of lens actuator hysteresis and desired lens proximity to the truly optimal position. Actuators with large, unknown or variable hysteresis should do a second flyback—jump back to the start position of the first scan, and then either retrace the steps made during the scan or directly jump to the best of the scanned positions. Actuators with constant hysteresis (like gear backlash) can be moved to that position directly from the end position of the scan—the AF algorithm offers an option to automatically increase the length of this move by a preprogrammed backlash-compensating step. Finally, if the first scan is coarse, relative to the positioning precision of the lens actuator and depth of field of the lens, an optional second fine scan can be performed around the lens position selected as best after the first scan.

The second scan is done in the same way as the first, except that the positions it covers are not preset. Instead, the AF algorithm user must preset step size and number of steps for the second scan and enable its execution by setting the appropriate control bit in one of AF driver variables. Finding this bit equal to “1” at the end of the first scan, the AF driver calculates lens positions to be tried in the second scan from its two user-set parameters and the position found best in the first scan. The calculation takes into account where that position is, relative to the limits of the lens motion range and how it would be reached if the second scan were not enabled.

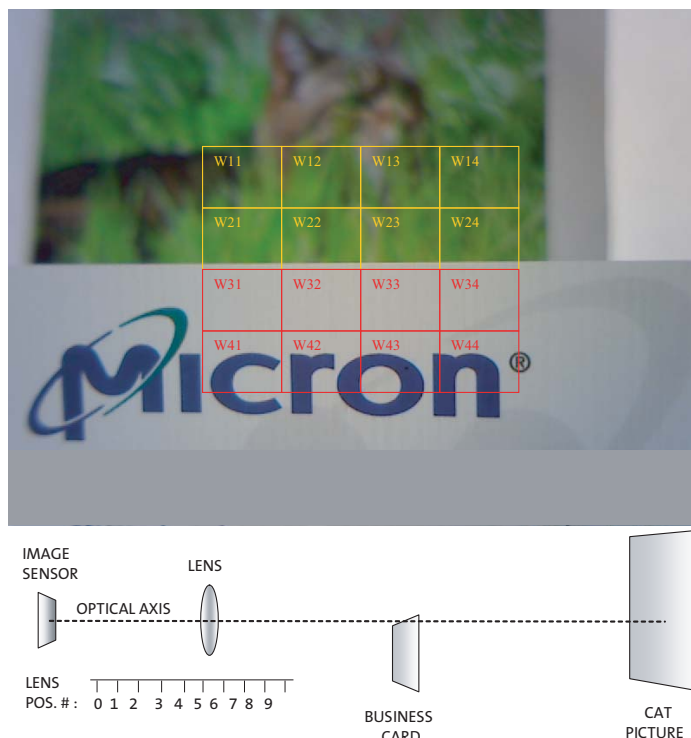
If the user-selected way to reach it includes the second flyback, the AF driver assumes that the start position of the second scan must likewise be reached not directly from end position of the first scan, but via logical position 0, the default start position of that scan. An extra zero is therefore put at the beginning of the list of positions calculated for the second scan—unless this list already starts with logical position 0. If the second flyback is not enabled, no extra zero is prepended to the list. In every case, the list is then appended to the list of positions already scanned in the first scan. The combined list cannot have more than 20 entries, due to fixed 20-byte size of memory buffer used by the AF driver to store lens positions. This means that the first scan of, say, 15 positions can be followed by a flyback to 0 and second scan of no more than 4 non-zero positions or, alternatively, a second scan of up to 5 non-zero positions if the second flyback is not enabled. In both cases, the two unidirectional scans can be also seen as a single scan with two changes of direction. If the lens actuator has significant hysteresis, the effect of those changes should be carefully considered. The only way to alleviate it is to do the flyback to 0 prior to the second change.

The second scan is always followed by the user-selected final positioning sequence that in the absence of the second scan would follow the first scan—a flyback to the start position of the latter and a jump to the position found best in the former.

Focus Targets at Different Distances

Figure 31 depicts a simple scene with two potential focus targets: a business card in front and a picture of a cat far in the background. Distances in the diagram are not to scale. Red and yellow rectangles in the middle of the image represent 16 AF windows, each of which yields a separate Y-normalized sharpness score. Sharpness scores from the lower 8 windows are highest when the business card is in focus, which happens when the lens is at position 5. Scores from the upper 8 windows peak when the lens is at position 0.

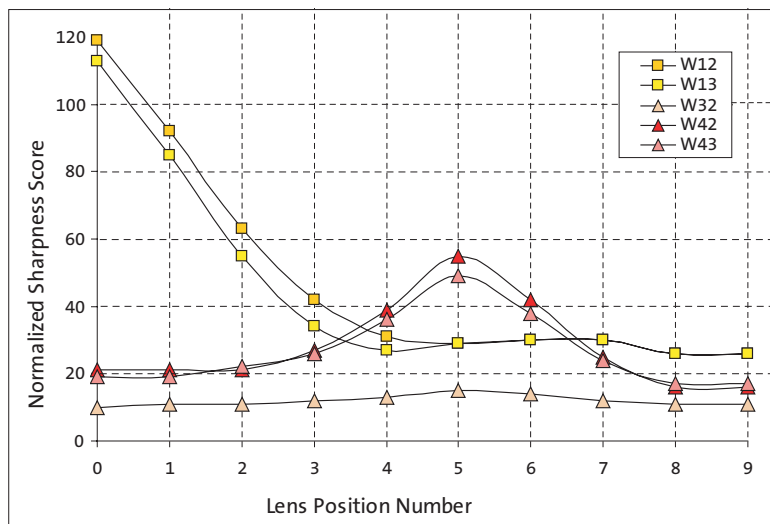
Figure 31: Scene with Two Potential Focus Targets at Different Distances From Camera



Luminance-Normalized Local Sharpness

Luminance-normalized sharpness scores from AF windows W12, W13, W32, W42, and W43 are reflected in Figure 32. Lens position 0 is the position of best focus for windows W12 and W13, while windows W42 and W43 are in sharpest focus at lens position 5. The relatively featureless window W32 is in sharpest focus at the same position, but it is difficult to determine this from its sharpness score, which varies very little with lens position.

Figure 32: Dependence of Luminance-Normalized Local Sharpness Scores on Lens Position



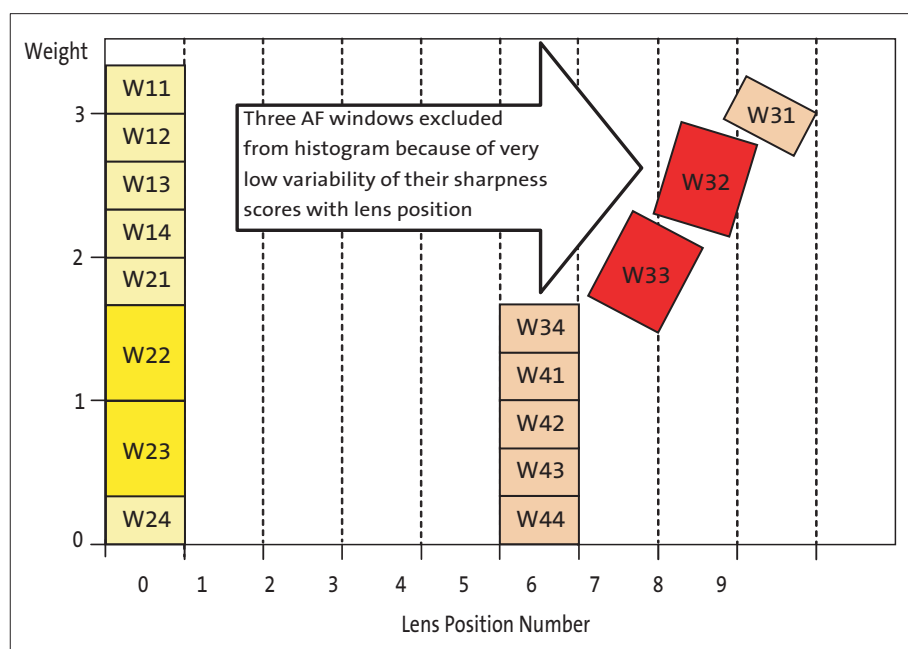
Position Weight Histogram Created by AF Driver

Figure 33 shows a position weight histogram created by AF driver to select best lens position. This histogram corresponds to the situation depicted in Figure 32 and Figure 33. After scanning ten lens positions numbered from zero through nine, the AF driver determined that Y-normalized sharpness scores from the upper 8 of 16 AF windows (W11 through W24) peak at lens position 0, while the scores from the lower 8 windows (W31 through W44) at lens position 5.

For each of the two positions, the AF driver summed preprogrammed weights of the AF windows being clearly in focus at that position, thus obtaining two position weights. These weights would have been equal except for very weak dependence of sharpness scores from windows W31, W32, and W33 on lens position.

Finding peak-to-trough variability of these scores lower than preprogrammed threshold, the AF driver concluded that for W31, W32, and W33, no lens position was clearly optimal, and therefore the weights of these windows should not be added to the weight of position 5. This gave position 0 a higher weight and decided its selection as the best position. Note unequal weighting of the AF windows, increasing the importance to the 4 central ones.

Figure 33: Example of Position Weight Histogram Created by AF Driver



Evaluation of Image Sharpness

Information on image sharpness that the AF algorithm uses to find best focus position is provided by focus measurement engine (FME), a programmable edge-filtering module in IFP. The FME convolves two preprogrammed 1-dimensional digital filters with luminance (Y) data that it receives row by row from the color interpolation module. For each interpolated frame, the convolution of the AF filters with Y produces up to 32 local sharpness scores reflecting the density and sharpness of vertical edges in up to 16 user-selected areas of the frame. The FME outputs these sharpness scores once every frame to

IFP registers R[77:84]:2 and R[87:94]:2 (where “[:]” denotes a range of register numbers and “:2” means page 2). In addition, the FME calculates and writes to IFP registers R[67:74]:2 up to 16 local averages of Y that can be used to normalize the sharpness scores and thus make them nearly independent of scene brightness.

Since each sharpness score and Y average has only 8 register bits allocated for it, care should be taken in programming the AF filters to ensure that the sharpness scores they produce never exceed 255. Otherwise, the content of registers R[77:84]:2 and R[87:94]:2 may not match actual sharpness scores computed by the FME.

The exact method of computing the sharpness scores is as follows. Sixteen equal-sized rectangular windows forming a 4 x 4 grid are superimposed on each color-interpolated frame. The size of these AF windows and the position of the upper-left corner of the grid are programmable (via IFP registers R[64:66]:2). The grid does not have to be entirely inside the frame.

For example, it is perfectly legal to cover most of the frame with a 3 x 3 portion of the grid and place the remaining 7 AF windows almost entirely outside of it, as shown in Figure 34. Whenever a portion of an AF window is inside the frame, the FME calculates two sharpness scores and averages Y for this portion. However, it can write these results to registers only if the bottom boundary of the window is partly or fully inside the frame. Placing this boundary entirely outside the frame makes the window inactive, in the sense that the FME stops the output of new sharpness scores and Y averages for it. Although no AF window having some part of the bottom boundary inside the frame can be deactivated in the same sense, any AF window can be made irrelevant in the AF algorithm by giving it a weight of zero.

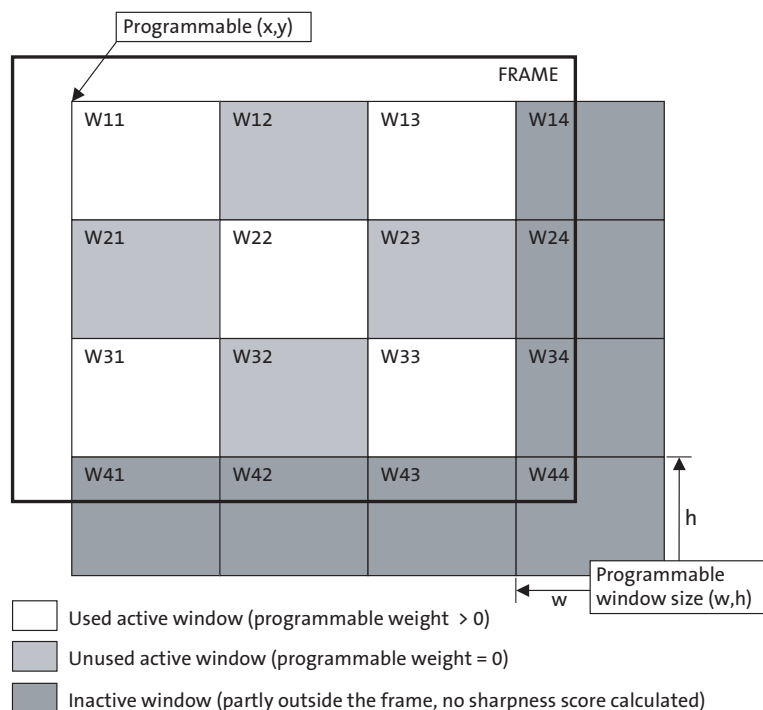
Each frame is read out from the sensor core and processed by the IFP row by row. Every rectangular block of pixels in the frame can be considered as a separate, smaller frame, also read out and processed row by row. A portion of a frame row belonging to any AF window will be referred to as a window row. As the color interpolation module processes each window row and makes Y values of its successive pixels available to the FME, the FME convolves those values with the two AF filters.

The AF filters are user-programmable within the following constraints: each can have 8 or 9 integer coefficients with values from -15 through 15, can be symmetric or antisymmetric, and can be multiplied by a power-of-2 weight factor ranging from 1/512 to 32. By default, both are programmed to detect sharp edges, but the first filter is more high-pass than the second. Each filter is applied to successive locations in a window row, starting at the first pixel and ending at the last. This requires using Y values from outside the window, specifically from the 4 columns to the left and 4 columns to the right of the window. Hence, when programming the size and position of the AF window grid, one should make sure that every AF window intended to have non-zero weight is at least 4 columns away from the left and right side of the frame.

Auto Focus Windows

Figure 34 shows an array of 16 equal-sized AF windows configured to work like a centered quincunx pattern of 5 windows.

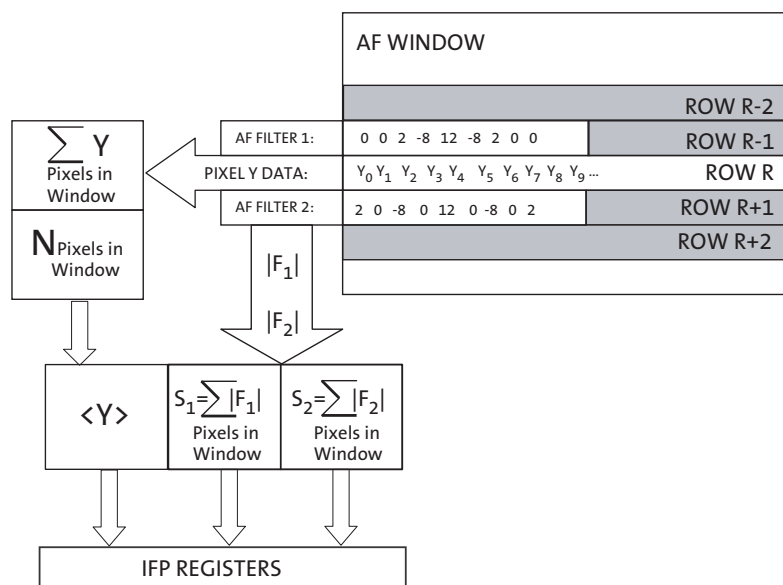
Figure 34: Auto Focus Windows



As the convolution of each AF filter with Y progresses along a window row, then to the next row, and so on, absolute values of its successive results are added to a sum that ultimately becomes a sum over the whole portion of the window located inside the frame. At the same time, the pixels in the window are counted and their Y values are added up to get the average Y for the window.

In this way, schematically depicted in Figure 35, each AF window not located fully outside the frame yields two sharpness scores (the sums of convolution results from the two AF filters) and 1 average Y. The number of window rows processed to obtain these results can be equal to or less than the common AF window height programmed into the register R65:2.

If the window row count matches that height, the results are output to registers. This never happens for AF windows positioned like W41 or W44 in Figure 34—hence, these windows are inactive. Results from each active AF window are output immediately after its last row is processed.

Figure 35: Computation of Sharpness Scores and Luminance Average for AN AF Window


- Notes:
1. This figure shows the computation of sharpness scores, S_1 and S_2 , and average luminance, $\langle Y \rangle$, in an AF window.
 2. Coefficients of each of the two AF filters are independently programmable. The filters shown here as an example yield convolution results:
 $F_1 = 2Y_2 - 8Y_3 + 12Y_4 - 8Y_5 + 2Y_6$ and $F_2 = 2Y_0 - 8Y_2 + 12Y_4 - 8Y_6 + 2Y_8$.

The symmetry constraint placed on the AF filters reduces the number of coefficient values needed to define them. Symmetric 8- or 9-coefficient filters are defined by specifying 4 or 5 coefficient values, respectively. Only four coefficients are needed to define an 8- or 9-coefficient antisymmetric filter. Examples of AF filters that can be programmed into the MT9D111 are given in Table 12. To program the first AF filter, write its parameters to IFP registers R75:2 and R76:2. The parameters of the second AF filter must be written to IFP registers R85:2 and R86:2.

Table 12: Possible AF Filters

Filter Parameters Programmed into Registers								
Filter Size	Filter Symmetry	Filter Coefficients					Filter Weight	Filter
		C0	C1	C2	C3	C4		
8	Symmetric	n/a	6	-7	2	0	1	[0 2 -7 6 6 -7 2 0]
8	Antisymmetric	n/a	1	0	0	0	1/4	[0 0 0 -1/4 1/4 0 0 0]
9	Symmetric	6	0	-4	0	1	2	[2 0 -8 0 12 0 -8 0 2]
9	Antisymmetric	0	15	5	0	0	1/8	[0 0 -5/8 -15/8 0 15/8 5/8 0 0]

Edge Detection

Fermi-function:

$f(E) = 1 / (\exp((E - E_f) / KT) + 1)$ to simulate edges and empirically pick a set of parameters that can make the most distinguishable results between edge and non-edge on luminance (Y).

Note: Saturated color blocks might have different luminance such as Red (255 0 0) versus Yellow (255 255 0).

Filters

1. Sobel filter [2 2 2 2 0 -2 -2 -2 -2]
It is used to find the max gradient in an input grey scale image. The advantage is this anti-symmetric filter can average out temporal noise with the same coefficient for each side. The difference with saturated color blocks in the MacBeth color charts are acceptable.
2. Laplace filter [-2 -2 -2 -2 16 -2 -2 -2 -2]
This symmetric filter is to get the second derivative. They are very sensitive to noise.
3. Gaussian filter [-8 -8 5 5 12 5 5 -8 -8] or [-8 -6 3 5 12 5 3 -6 -8]
This is a modification of the Laplace filter which can reduce the noise effect. However, the score looks different for different saturated colors.
4. Default filter [2 0 -8 0 12 0 -8 0 2]
Often the filter produces similar scores versus noise when there are only a few edges in a plain block.

It is suggested to use a Gaussian filter and choose the proper weight according to scenes. For example: choose [-12 -12 8 9 14 9 8 -12 -12] and [-4 -4 2 3 6 3 2 -4 -4] to prevent score saturation.

Note: The coefficients are integers within [-15 15].

Sharpness Threshold

$\Delta[i] = (\max_sha[i] - \min_sha[i]) * 256 / \max_sha[i]$ // check the sha among AF positions.

If $(\Delta[i] > shaTH)$ $score[best_position[i]] += weight$;

If the max-min is small, the delta would be small. This means that if the sharpness is not changing with the AF positions, it should be noise. However, it is scene/condition dependent whether it should be normalized with max_sha or not.

The default value is 10. While in low light condition, if the max_sha is small, it would be larger than threshold; therefore, one should choose a larger shaTH instead. The window with noise but have passed the threshold might vote for the wrong position. But in normal light, shaTH might be good to stay with 10.

Note: The scale factor 2 in the Sobel filter should be chosen so the sharpness score before luma normalization is approximately 20 according to f# and transparency of the lens. Hence, there would be more information in the raw scores. The sharpness threshold must be adjusted accordingly for the best results.

However, the best way is not to use the noisy windows or automatically change weights according to the real sharpness information. It can not be done in patch because the function is inside a larger function so the patch would exceed 1K RAM that customer uses to override the original function.

Hysteresis

Always check to be sure that the filter still works when reversing the physical position mapping. The actual positions of stops are different due to hysteresis.

This implies that the positions for lens stops should be optimized also. While the linear default positions might work in most cases, for better results pick up some best positions with frequently used scenes for specific lenses.

Algorithm Flowchart

When configured to do a single scan, the scan AF algorithm searches for best focus position according to the flowchart in Figure 36 and Figure 37.

Figure 36: Flowchart of Scan AF Algorithm Implemented in the MT9D111

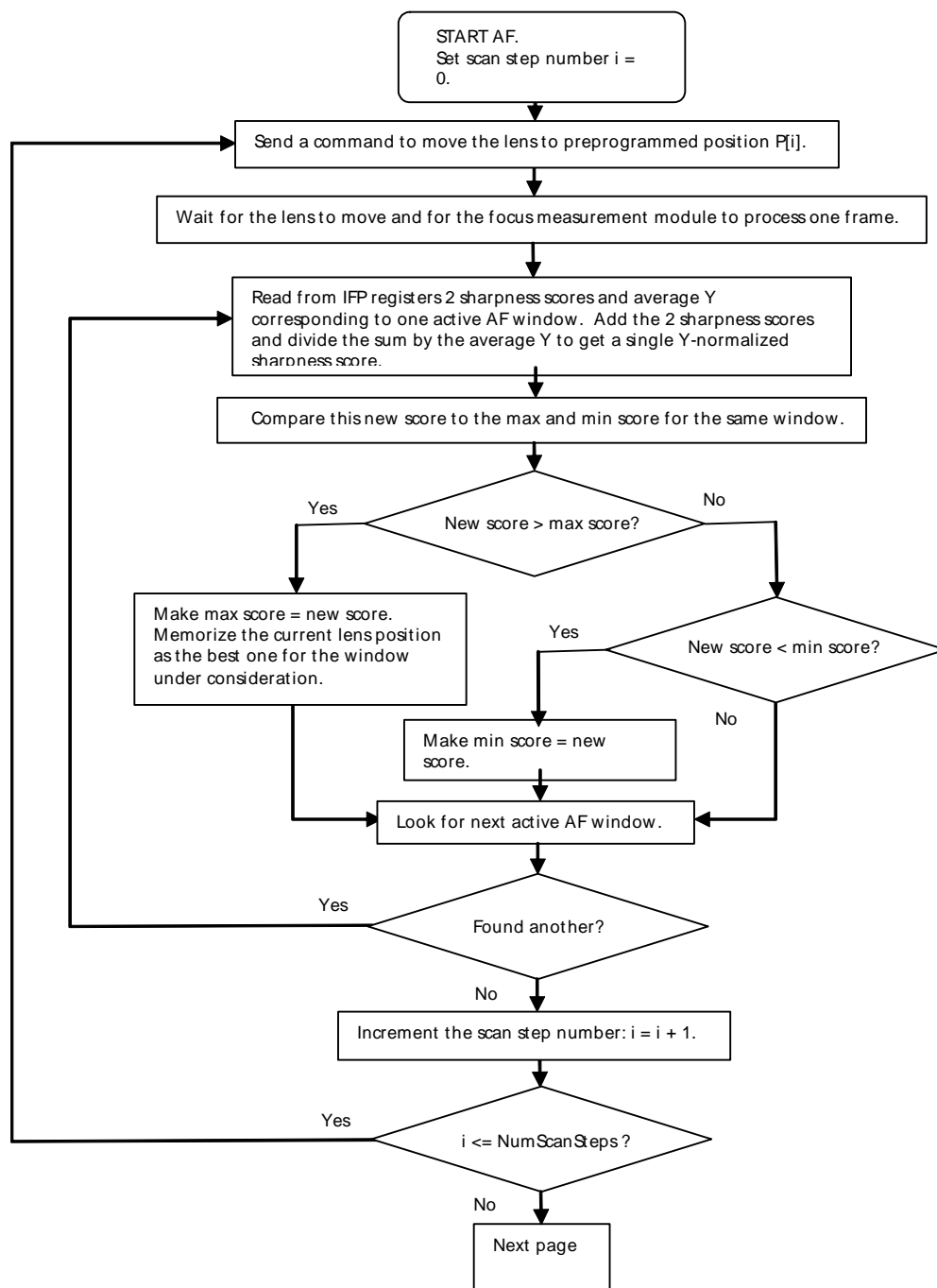
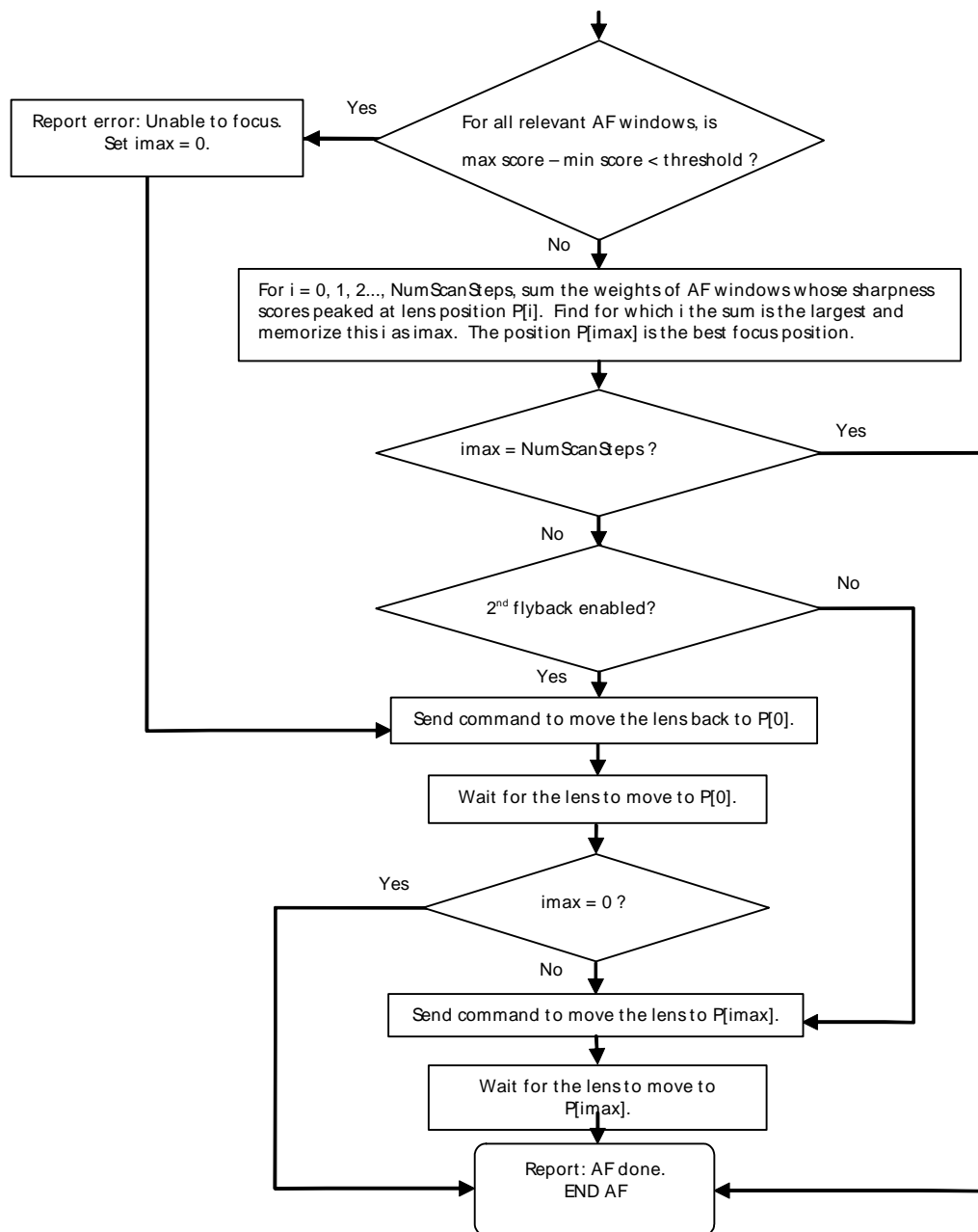


Figure 37: Flowchart of Scan AF Algorithm Implemented in the MT9D111



Creep Compensation

In addition to an implementation of the flowchart given in the previous section, the AF driver includes a function whose task is to return an AF lens to previously found best focus position if the lens actuator is unable to keep it there over long periods of time (e.g. over a 30-second focus lock period). According to external sources and tests carried out at Aptina, helimorph actuators exhibit this problem, referred to as position creep. Creep compensation currently implemented in the AF driver would be more accurately called brute force re-positioning: to make the lens actuator “forget” the position creep, the lens is moved from its current logical position, `afm.curPos`, to `af.positions[0]` and then back. If `afm.curPos` equals 0, which is the default value of `af.positions[0]`, `af.positions[af.numSteps-1]` is chosen instead as the destination of the first move. The creep compensation function is enabled by switching the AF driver to creep compensation mode, i.e. by setting bits 6 and 7 of `af.mode` to 1 and 0, respectively. The creep compensation function does not move the lens as long as bits [5:0] of `af.mode` all equal 0. To correctly trigger lens re-positioning in the creep compensation mode, one must set bit 0 of `af.mode` to 1.

Public Variables of AF Driver

Public variables of the AF driver listed in Table 13 and register settings defining the AF filters are all the parameters that one needs to pay attention to when customizing the built-in AF algorithm. The AF driver variables include two unsigned characters (bytes) named `af.windowPos` and `af.windowSize` that should be used to adjust position and size of the AF windows, rather than direct writes to registers `R[64:66]:2`. It is certainly possible to access these registers directly, and new values written to them have immediate effect. However, these values remain in effect only as long as the sequencer driver, the master firmware driver continuously running on the MT9D111 microcontroller unit (MCU), does not call AF driver function `AF_SetSize` (or its user-supplied substitute).

The sequencer calls this function at its initialization, at every change of sensor operation mode (e.g. from preview mode to capture mode), and also whenever `sq.cmd` variable is set to 6 in the preview mode. The function `AF_SetSize` translates the current settings of `af.windowPos` and `af.windowSize` to corresponding settings of registers `R[64:66]:2` and writes these settings over the previous values of the registers. There is no way to change the precedence of `af.windowPos` and `af.windowSize` over the register settings other than by overriding the function `AF_SetSize`.

In addition to programming the registers `R[64:66]:2`, the function `AF_SetSize` automatically sets `af.wakeUpLine` in accordance with `af.windowPos` and `af.windowSize`, so that during every frame readout the AF driver is activated 2 rows below the bottom of the 4 x 4 array of AF windows. If `af.windowPos` and `af.windowSize` are such that the bottom of the array is outside the frame, the value given to `af.wakeUpLine` by `AF_SetSize` is invalid (greater than frame height) and must be changed to something less than the frame height, otherwise AF will not work.

Table 13: Public Variables of the Auto Focus Driver

Off	Name	Type	Default	RW	Description
0	Vmt	void*	0xE9D2	RW	Pointer to virtual method table (VMT)
2	windowPos	uchar	0x44	RW	Position of the upper left corner of the first AF window (W11): Bits [3:0]—x coordinate (horizontal offset from the upper left corner of the frame) in units of 1/16th of frame width, Bits [7:4]—y coordinate (vertical offset from the upper left corner of the frame) in units of 1/16th of frame height. New position written to this variable takes effect after REFRESH_MODE command is given to the sequencer driver (sq.cmd is set to 6).
3	windowSize	uchar	0x77	RW	Dimensions of the 4 x 4 array of AF windows: Bits [3:0]—width (in units of 1/16th of frame width) decremented by 1, Bits [7:4]—height (in units of 1/16th of frame height) decremented by 1. New dimensions written to this variable take effect only after REFRESH_MODE command is given to the sequencer (sq.cmd is set to 6).
4	mode	uchar	0	RW	Two mode switches and 5 bits reserved for use in default snapshot AF mode: Bit 7—manual mode switch (0—manual mode disabled, 1—enabled) Bit 6—creep compensation mode switch (0—creep compensation mode disabled, 1—enabled) Bits [4:0]—reserved for use in snapshot AF mode If AF is enabled in the Sequencer (sq.mode bit 4 = 1) and manual mode is disabled (af.mode bit 7 = 0), a snapshot AF sequence can be triggered at any time by setting af.mode bit 0 to 1. Bits [4:0] are used in the sequence and automatically cleared at its end.
5	modeEx	uchar	128	RW	Four option switches and 4 status indicators: Bit 7—switch enabling the second flyback (jump to the start position of the first scan) and then jump to best focus position, Bit 6—switch enabling the second flyback and retracing of scan steps to best focus position (0—option disabled, 1—enabled), Bit 5—switch enabling the second scan (0—disabled, 1—enabled), Bit 4—status indicator, reserved to indicate AF algorithm failure, currently set to 1 only when auto focusing is attempted while auto exposure is settling, otherwise cleared at the start of the first flyback Bit 3—if bit 7 of af.mode equals 0, this bit enables skipping 1 extra frame after detecting that bit 1 of afm.status has been cleared (i.e. after the end of every lens movement) Bit 2—status indicator, set to 1 when the extra frame is being skipped Bit 1—status indicator, set to 1 when the second scan is in progress Bit 0—status indicator, set to 1 when sharpness scores are ready.
6	numSteps	uchar	10	RW	Number of steps (lens positions tried) in the first scan.
7	initPos	uchar	0	RW	Number (index) of start position, af.positions[af.initPos], used in the first scan and optional second scan. Must be 0 at the beginning of the first scan if the second is enabled. Otherwise, can be set before the first scan to any value below 20-af.numSteps. The AF driver makes af.initPos equal to af.numSteps at the beginning of the second scan and equal to 0 at the end of last scan (first or second, whichever is last).
8	numSteps2	uchar	6	RW	Bits [3:0]—desired number of steps in second scan (max. allowed number is 14) Bits [7:4]—actual number of steps in the second scan (calculated by the AF driver at the beginning of the scan).

Table 13: Public Variables of the Auto Focus Driver (continued)

Off	Name	Type	Default	RW	Description
9	stepSize	uchar	6	RW	Logical step size for the second scan. Because the logical range of motion is from 0 to 255, af.stepSize=6 means that during the second scan the AF driver tries to move then lens in increments equal to 6/255 of the length of its entire motion range. However, the lens actuator may or may not be able to move the lens in steps of precisely that size. It is important to ascertain that lens movements requested by the AF driver during the second scan can be at least reasonably approximated by the lens actuator. The quality of the approximation may depend on how well the physical limitations of the actuator are accounted for in the source code and /or configuration of the AFM driver.
10	wakeUpLine	uint	448	RW	Number of image row at which the MCU wakes up to execute AF driver code. When the function AF_SetSize resizes the 4 x 4 array of AF windows according to new values of af.windowPos and af.windowSize, it automatically makes af.wakeUpLine equal to the number of the second row below the bottom of the array. If af.windowPos and af.windowSize are such that the bottom of the array is outside the frame, the value given to af.wakeUpLine by AF_SetSize is greater than frame height, i.e. invalid. It must be changed to some-thing less than the frame height, otherwise AF does not work.
12	zoneWeights	ulong	0xFFFFFFFF	RW	Weights of the AF windows or zones. Bits [1:0] of this variable represent the weight of window W11, bits [3:2] the weight of W12, and so on to bits [31:30] that represent the weight of W44. Since each weight is represented by just 2 bits, it is allowed to have only 4 values, 0, 1/3, 2/3 or 1. Value stored in each 2 bits equals window weight times 3, so the value of 1 signifies the weight of 1/3, 2 stands for 2/3, and 3 for 1.
16	distanceWeight	uint	0xFF	RW	Reserved.
17	bestPosition	uchar	0	RW	This variable is used in three different ways depending on values of bits 6 and 7 of af.mode. When bit 7 equals 1 (in manual lens control mode), the position of AF lens can be changed by changing the value of af.bestPosition, which is interpreted by the AF driver as logical lens position desired by its user. The AF driver reads af.bestPosition once every frame, and if it differs from current logical lens position (afm.curPos), the AF driver gives the AFM driver a command to make these variables equal by moving the lens. Physical movement of the lens corresponding to the change of afm.curPos to af.bestPosition always takes some time, during which it is best not to change the value of af.bestPosition to avoid possible errors. When af.mode bits [7:6] are both 0, af.bestPosition serves to store AF algorithm output rather than user input. After both first and second scan, the AF algorithm outputs to this variable the offset of programmable logical lens position found to be best relative to the start position of the scan. In other words, after each scan, the best lens position found is af.positions[af.initPos+af.bestPosition]. When af.mode bit 7 is 0 and bit 6 is 1 (creep compensation mode is enabled) af.bestPosition is used during lens re-positioning triggered by setting bit 1 of af.mode to 1. It is used to store the desired final lens position, which is assumed to be the same as afm.curPos before the re-positioning. As a result after every successful re-positioning, af.bestPosition equals afm.curPos.
18	shaTH	uchar	10	RW	Sharpness score variability threshold. Only AF windows whose MIN and MAX normalized sharpness scores satisfy the condition (max.score-min.score)/max.score >= af.shaTH/256 are used to select best focus position.
19	positions[0]	uchar	0	RW	Programmable logical lens position 0.
20	positions[1]	uchar	28	RW	Programmable logical lens position 1.
21	positions[2]	uchar	56	RW	Programmable logical lens position 2.
22	positions[3]	uchar	85	RW	Programmable logical lens position 3.

Table 13: Public Variables of the Auto Focus Driver (continued)

Off	Name	Type	Default	RW	Description
23	positions[4]	uchar	113	RW	Programmable logical lens position 4.
24	positions[5]	uchar	141	RW	Programmable logical lens position 5.
25	positions[6]	uchar	170	RW	Programmable logical lens position 6.
26	positions[7]	uchar	198	RW	Programmable logical lens position 7.
27	positions[8]	uchar	226	RW	Programmable logical lens position 8.
28	positions[9]	uchar	255	RW	Programmable logical lens position 9.
29	positions[10]	uchar	27	RW	Programmable logical lens position 10.
30	positions[11]	uchar	55	RW	Programmable logical lens position 11.
31	positions[12]	uchar	84	RW	Programmable logical lens position 12.
32	positions[13]	uchar	112	RW	Programmable logical lens position 13.
33	positions[14]	uchar	140	RW	Programmable logical lens position 14.
34	positions[15]	uchar	169	RW	Programmable logical lens position 15.
35	positions[16]	uchar	197	RW	Programmable logical lens position 16.
36	positions[17]	uchar	225	RW	Programmable logical lens position 17.
37	positions[18]	uchar	254	RW	Programmable logical lens position 18.
38	positions[19]	uchar	26	RW	Programmable logical lens position 19.

Public Functions of AF Driver and Corresponding VMT Pointers

Public functions of the AF driver are those functions that can be called via pointers in the driver's virtual method table (VMT) and are called in this way by other functions in the AF and sequencer drivers.

The reason for using indirect function calls in the firmware is to permit selective function replacement, which may be needed to economically update or customize the firmware. Since the MT9D111 users are expected to do most of the firmware customization, it is necessary to give them at least some information on the firmware functions that they can replace with their own. Hence, all these functions are designated as public.

Because there are disadvantages to having many indirect function calls and large jump tables, only selected functions in each firmware driver have corresponding pointers in the driver's VMT. They can be viewed as the top layer of the driver's functionality, under which there may be one or more layers of functions used by the top layer to perform certain tasks recurring more than once in the firmware code. Those lower level functions are always called directly and are all designated as private. While knowing and using the private functions can help in development of substitutes for the public functions, brief descriptions of the latter is all that most readers of this document are likely to need. Such brief descriptions of the public functions of the AF driver are given in the list below, which also shows the names of pointers to these functions that comprise the AF driver VMT. The order of the pointers in the VMT is the same as in this list. All the pointers are of type void* and therefore in indirect function calls each must be cast to the function pointer type matching the corresponding function.

void AF_Init (void)

Pointer.

af.Vmt->plnit

Description.

Initializes the public variables of the AFM and AF drivers and calls function AF_SetSize (indirectly, via pointer af.Vmt->pSetSize). The first variable initialized by AF_Init is afm.type. It is set to 0, which means that the AF driver is always initialized as if there is no lens actuator for it to control.

The initialization of all other AFM driver variables is taken care of by calling (directly) AFM driver function `AFM_Init`. When `afm.type=0`, this function fills the RAM segment holding the AFM driver variables with zeros and then writes positive default values to 3 of the variables whose type is `void*`. `AF_Init` then initializes the AF driver variables, using explicit assignments of default values, which means that these values can be changed only by overriding `AF_Init`. The final jump to the function pointed to by `af.Vmt->pSet-Size` is preceded by two lines of code that fix the argument of this function, expected frame size, at 800 x 600.

Use. Called once, indirectly, by Sequencer driver function `SEQ_Init`, one of whose tasks is to initialize all firmware drivers.

`void AF_DownloadRegs_16 (unsigned char* sharpness)`

Pointer . `af.Vmt->pDownloadRegsSnapshot`

Description. Reads 32 raw sharpness scores from IFP registers `R[77:84]:2` and `R[87:94]:2` and 16 average luminances from registers `R[67:74]:2`. Reduces these data to 16 normalized sharpness scores by calculating for each AF window the ratio $(S_1+S_2)/\langle Y \rangle$, where $\langle Y \rangle$ is average luminance and S_1 and S_2 are raw sharpness scores multiplied by 128. Replaces all normalized scores greater than 255 with 255. Stores all the normalized scores in a 16-byte memory segment with starting address equal to the argument, unsigned `char*` sharpness.

Use. Called once, indirectly, by function `AF_Run_snapshot`, which tracks the normalized sharpness scores while performing the first and optionally second scan of lens positions.

`void AF_ParseCmd (void)`

Pointer . `af.Vmt->pParseCmd`

Description. Uses the pointer `afm.vmt->pSetPos` to indirectly call AFM driver function whose task is to move lens actuator to a new logical position given to it as an argument. The argument passed to that function by `AF_ParseCmd` is `af.bestPosition`.

Use. This function is called once, indirectly, by function `AF_Run`, when bit 7 of `af.mode` equals 1. The purpose of this is to enable AF driver users to “manually” control their lens actuators, once the setting of `af.mode` bit 7 to 1 has taken control away from the AF algorithm.

The user can exercise control by writing to AF driver variable `af.bestPosition`, whose value is passed by `AF_ParseCmd` to the AFM driver as desired logical lens position. The AFM driver compares the desired position to the current logical lens position, `afm.curPos`, and if they differ, attempts to move the lens to the desired position. If a functioning lens actuator is connected to the GPIO and the AFM driver is properly configured to control it, the attempt should result in a physical lens movement taking certain amount of time. Duration of physical lens movements and the intervals at which the Sequencer calls the function `AF_Run` must be considered when timing successive writes to `af.bestPosition`. Writing to it more than once per frame makes no sense, because the `AF_Run` and `AF_ParseCmd` functions are executed only once per frame. Changing the value of `af.bestPosition` before previously commanded lens movement is completed may disrupt that movement if the change is communicated by `AF_ParseCmd` to the AFM driver and it sends a new set of signals to the lens actuator.

Neither AF_Run nor AF_ParseCmd check the status of the AFM driver before passing the value of af.bestPos to it, so it is the responsibility of the user to prevent timing-related errors in the manual lens control mode. In the snapshot AF and creep compensation modes (when bit 7 of af.mode is 0), the status of the AFM driver is always checked before it receives a command from the AF driver—see description of AF_Run_snapshot below.

void AF_SetSize (t_size)

Pointer .

af.Vmt->pSetSize

Description.

Translates current settings of af.windowPos and af.windowSize to corresponding settings of registers R[64:66]:2 and writes these settings to the registers. Sets af.wakeUpLine in accordance with af.windowPos and af.windowSize, so that during every frame readout the AF driver is activated two rows below the bottom of the 4 x 4 array of AF windows. If af.windowPos and af.windowSize are such that the bottom of the array is outside the frame, the value given to af.wakeUpLine by AF_SetSize is invalid (greater than frame height) and must be changed to something less than the frame height, otherwise AF does not work.

Use.

Called indirectly by AF driver function AF_Init and Sequencer function SEQ_ModeChange. The latter function is executed at Sequencer initialization, at every change of sensor operation mode (e.g. from preview mode to capture mode), and also whenever sq.cmd variable is set to 6 in the preview mode.

void AF_Run (void)

Pointer .

af.Vmt->pRun

Description.

Calls one of three other AF driver functions depending on the values of bits 6 and 7 of af.mode. If bit 7 equals 1 (manual lens control mode is enabled) the function called is the one pointed to by the pointer af.Vmt->pParseCmd (AF_ParseCmd or its user-supplied substitute). If bit 7 equals 0, the called function is determined by bit 6 (creep compensation mode switch).

If bit 6 is 0 (creep compensation mode is disabled, snapshot AF mode enabled), AF_Run calls the function pointed to by pointer af.Vmt->pRunSnapshot (AF_Run_snapshot or its user-supplied substitute). If bit 6 is 1, AF_Run calls a private AF driver function whose task is to return an AF lens to previously found best focus position if the lens actuator is unable to keep it there over long periods of time (for example, over a 30-second focus lock).

According to external sources and tests carried out at Aptina, helimorph actuators exhibit this problem, referred to as position creep. Creep compensation currently implemented in the AF driver would be more accurately called “brute force repositioning.” To make the lens actuator “forget” the position creep, the lens is moved from its current logical position, afm.curPos, to af.positions[0] and then back. If afm.curPos equals 0, which is the default value of af.positions[0], af.positions[af.numSteps-1] is chosen instead as the destination of the first move. The creep compensation function does not move the lens as long as bits [5:0] of af.mode all equal 0. To correctly trigger lens repositioning in the creep compensation mode, one must set bit 0 of af.mode to 1.

Use.

Called once, indirectly, by the Sequencer driver, when bit 4 of sq.mode equals 1 (AF is enabled) and Auto Exposure driver indicates that it is done setting sensor exposure level. The Sequencer does not allow the AF driver to do snapshot mode auto focusing while the exposure is not settled. Any violation of this prohibition results in an error signal: bit 4 of af.modeEx is set to 1.

void AF_Run_snapshot(void)

Pointer . af.Vmt->pRunSnapshot

Description.

This function guides AF lens through the motion sequence illustrated in Figure 30, Search for Best Focus, on page 73. During the first and optional second scans, it keeps track of extreme normalized sharpness scores from every AF window and uses them to select best lens position at the end of each scan. Every time AF_Run_snapshot is called (which happens once every frame, if bits [7:6] of af.mode are 0), the first thing it does is an indirect function call using pointer afm.vmt->pGetStatus. This pointer points to a lens-actuator-dependent status checking function of the AFM driver, whose task is to check current status of the driver and lens actuator, update the variable afm.status and return its updated value.

After receiving this value, AF_Run_snapshot checks if its bit 1 equals 1, which indicates that the lens actuator is busy moving the lens. If it is, AF_Run_snapshot immediately returns, in effect putting the AF driver to sleep until the row count reaches af.wakeUpLine in the next frame. If updated afm.status indicates that the lens is stationary, AF_Run_snapshot checks if the option to skip one extra frame after the end of every lens movement is enabled (bit 3 of af.modeEx is 1) and if the current frame should therefore be skipped. If the answer to both questions is yes, AF_Run_snapshot sets bit 2 of af.modeEx to 1 and immediately returns, causing the AF driver to skip the frame. At the next frame, the set bit tells AF_Run_snapshot that no more frame skipping is needed. The function clears this bit whenever it equals 1.

Next, AF_Run_snapshot checks the current value of af.mode. A 0 value here also causes AF_Run_snapshot to immediately return, because it indicates that no auto focusing has been requested since the completion of last AF sequence. A correct request for auto focusing (or creep compensation) is to set bit 0 of af.mode to 1 when bits 7 and [5:0] are 0. AF_Run_snapshot is called only in snapshot AF mode, when bits [7:6] of af.mode are 0. This means that at the first execution of AF_Run_snapshot after the correct request for auto focusing is made, af.mode equals 1. The function takes this as a signal to clear bit 4 of af.modeEx, clear a memory buffer for sharpness scores, command the AFM driver to do the first flyback, and increment af.mode by 1. The incrementing of af.mode signals that the first or second scan is in progress, and it continues until the scan is completed, at which point AF_Run_snapshot resets af.mode back to 0.

Every time AF_Run_snapshot finds af.mode greater than 1, it proceeds to do something with the lens that presumably is stationary at its current position. At every position but the last in the sequence shown in Figure 54 on page 116, there are some calculations to be done and then AF_Run_snapshot must give the AFM driver a command to move the lens to a new position. During the scans, calculations at each position are preceded by reading sharpness scores from registers. The reading is done by the function AF_DownloadRegs_16, which AF_Run_snapshot calls via the associated pointer af.Vmt->pDownloadRegsSnapshot.

To command the AFM driver to move the lens, AF_Run_snapshot uses logical positions stored in the array af.positions[] and function pointer afm.vmt->pSetPos. This pointer points to a lens-actuator-dependent AFM driver function that takes a logical lens position as an argument and acts to make the current logical lens position—afm.curPos—equal to that argument. If afm.curPos already equals the argument, the function does nothing. If the two are unequal, but there is no physical lens to move (afm.type=0), the function simply makes afm.prePos equal to afm.curPos and then afm.curPos to its argument. If in addition to changing these variables, the function is required to move a physical lens, it programs the GPIO to generate appropriate signals to

the lens actuator. The time required to program the GPIO is usually much shorter than the time required to move the lens, so in effect the AFM driver just sends the lens on its way to the new position and then quickly gives MCU control back to the AF driver (AF_Run_snapshot), which in turn quickly hands it over to the Sequencer. This quick return from AF-related activities to routine frame-by-frame maintenance of proper exposure, white balance, etc. is essential for the viability of MCU-controlled AF. The MCU cannot spend much time moving an AF lens. The time required should correspond to a few image rows near the bottom of every frame.

Use. Called once, indirectly, by AF_Run, if bits [7:6] of af.mode are 0.

void AF_Set2ndScan(void)

Pointer . af.Vmt->pSet2ndScan

Description.

The sole purpose of this function is to change a number of AF driver variables so that the function AF_Run_snapshot can do the second scan after completing the first. When considering the seemingly awkward features of second scan setup described below, keep in mind the fact that both scans are done using the same firmware code.

The first thing that AF_Set2ndScan does is set bit 1 of af.modeEx to 1. A non-zero value for this bit tells the function AF_Run_snapshot that it must do some things differently than the first scan—for example, it must take the value of bits [7:4] of af.numSteps2, not the value of af.numSteps, as the number of lens positions to scan.

The second preparatory change done by AF_Set2ndScan is to make af.initPos equal to af.numSteps. This change enables AF_Run_snapshot to locate the set of logical positions that it must step through in the second scan within the array af.positions[].

AF_Set2ndScan then generates this set of positions, using as input bits [3:0] of af.numSteps2, af.stepSize and af.positions[af.bestPosition], the position found best in the first scan. AF_Set2ndScan takes into account where that position is relative to the limits of the lens motion range and how it would be reached if the second scan were not enabled. If the user-selected way to reach it includes the second flyback (bit 7 af.modeEx is set to 1), AF_Set2ndScan assumes that the start position of the second scan must likewise be reached not directly from the end position of the first scan, but via logical position 0, the default start position of the first scan. AF_Set2ndScan therefore adds an extra zero to the list of positions calculated for the second scan—unless this list already starts with logical position 0. If the second flyback is not enabled, no extra zero is added to the list. In every case, the list is then appended to the list of positions already scanned in the first scan (for example, written to af.positions[af.initPos], af.positions[af.initPos+1], etc.).

The combined list of positions cannot have more than 20 entries, due to fixed 20-byte size of af.positions[] array. This means that the first scan of, say, 15 positions can be followed by a flyback to 0 and second scan of no more than 4 non-zero positions or, alternatively, a second scan of up to 5 non-zero positions if the second flyback is not enabled.

The last thing that AF_Set2ndScan does is to write the number of positions to be stepped through in the second scan (including the extra zero, if necessary) to bits [7:4] of af.numSteps2.

Use. Called once, indirectly, by AF_Run_snapshot, if bit 5 of af.modeEx equals 1.

Auto Focus Driver FAQs

- Is one second a reasonable adjustment time to get stable AF data?
- What settings adjust the AF ROI (Region of Interest) so that it only looks at the central part of the image to determine the optimal focus position? The primary ROI is positioned horizontally and vertically between 1/3 and 2/3 of the image width/height.
- We tried to change the AF zone by DevWare. We think that “IFP Register Page2 REG#129-135” are registers for setting AF zone, but it seems that the register values are not reflected to AF zone. Would you please explain the reasons? We would like to know the default AF zone. It seems that the AF zone for a 1600x1200 image is different from the 800x600 image used by DevWare.

Is one second a reasonable adjustment time to get stable AF data?

One second seems to be too long because the statistics update every frame. Even with 3 fps, you will not need a full second. Is the positioning done by hand or machine? It is possible that the hand/machine shakes while in position, which leads to initial AF statistic register instability.

What settings adjust the AF ROI (Region of Interest) so that it only looks at the central part of the image to determine the optimal focus position? The primary ROI is positioned horizontally and vertically between 1/3 and 2/3 of the image width/height.

The AF variables related to the sizing of ROI have units of 1/16. This means that you can evenly divide the region into half, quarter, etc., but not 1/3. If you do not need exactly 1/3 size, you can use the following settings:

- VAR8=5, 0x02, 0x0055 // AF_WINDOW_POS (determines the x and y coordinates of the starting coordinates)
- VAR8=5, 0x03, 0x0055 // AF_WINDOW_SIZE (determines the size of the ROI based on the frame size)
- VAR8=1, 0x03, 0x0006 // SEQ_CMD (refresh mode)

With the above values, instead of $1/3=3.3333$, you will get $5/16=0.3125$. If you need exactly $1/3$, you may overwrite an AF function (for these two variables) and use the register (with more resolution) instead.

We tried to change the AF zone by DevWare. We think that “IFP Register Page2 REG#129-135” are registers for setting AF zone, but it seems that the register values are not reflected to AF zone. Would you please explain the reasons? We would like to know the default AF zone. It seems that the AF zone for a 1600x1200 image is different from the 800x600 image used by DevWare.

The 16 windows used for AF are configured by the variables af.windowPos (ID=5, offset=2) and af.windowSize (ID=5, offset=3). See the description below:

Variable: af.windowPos

ID: 5, Offset: 2

Default value: 68

Description:

This variable is divided into two fields: bits [7:4] which specifies the scaled Y coordinate where the frame starts, and bits [3:0] which specify the scaled X coordinate where the frame starts. For example, if the value is set to 0xBC, then the starting coordinate of the

frame is $12/16 \times$ frame width, $11/16 \times$ frame height. Hence, an offset to the starting coordinate can be easily set. A refresh command (seq.cmd=5) is needed for the new values to be effective.

Variable: af.windowSize

ID: 5, Offset: 3

Default value: 119

Description:

This variable is divided into two fields: bits [7:4] which specify the scaling factor of the window height of the entire (not the individual 16) window, and bits [3:0] which specify the scaling factor of the entire window width. A factor x is defined as $[(x+1)/16] \times$ frame height or width. For example, if the value is set to 0xEE, then the programmable window size has a height and width of $15/16 \times$ frame height and $16/16 \times$ frame width, respectively. A refresh command (seq.cmd=5) is needed for the new values to be effective. This window size will be the area that AF uses for its analysis. The individual 16 windows will be divided equally from the total window size.

Auto Focus Mechanism

Introduction

This section describes the auto focus mechanics (AFM) drivers included in the firmware of the MT9D111 image sensor. These drivers encapsulate all firmware code developed to date by Aptina to support AF function in cameras built around our sensors.

Table 14 lists the lens actuator and associated driver IC currently supported.

Table 14: Actuators Supported

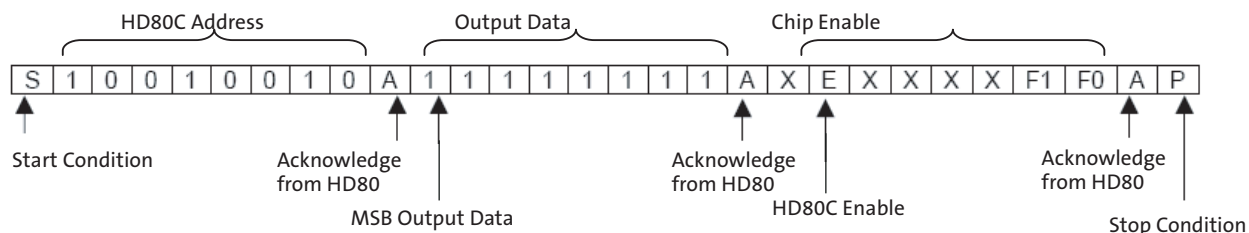
Driver IC	Interface Protocol	Lens Actuator	Patch Needed?	Actuator Performance
HD80 (One Limited)	I ² C (8-bit)	Helimorph	No	Simple configuration, less power consumption, but poor precision, and bad shock resistance.
AD5398 (Analog Device)	I ² C (10-bit)	VCM	No	
AD5398 (Analog Device)	I ² C (10-bit)	MEMS	No	Excellent precision, good optical performance, but complicated configuration.
LB1935T (Sanyo)	Stepper	Stepper	No	Better optical performance and precision, but more power consumption and complicated configuration.
LB1935CL (Sanyo)	Stepper	Stepper	No	
LV8071LP (Sanyo)	Stepper Like	Piezo	Yes	Simple configuration, less power consumption, but poor precision, and bad shock resistance.
MD115 (Silicon Touch)	PWM	VCM	Yes	Low cost, and simple configuration, but poor precision and loud noise.
ID9701 (Interpion)	PWM	VCM	Yes	

HD80 (I²C)

Interface Protocol Waveform

The HD80C serial input is standard-mode and fast-mode I²C compatible, from DC to 400 kHz. The minimum time between successive writes to the I²C is 100μs. SCL and SDA inputs have hysteresis to improve noise immunity. Eight bit unsigned output data and 1 bit shutdown/power up data may be sent to the HD80C in each serial packet. A complete write to the HD80C is shown in Figure 38.

Figure 38: HD80C Complete WRITE



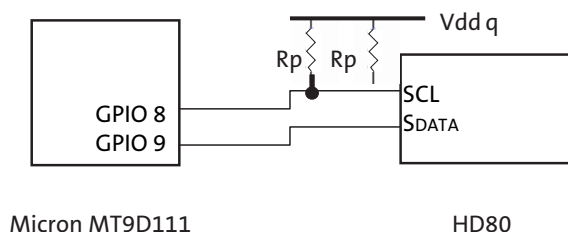
Bit	Function	Default Value	FO	F1	PSU Frequency
E	Enables HD80C	0	0	0	80 kHz
F0	Lower PSU freq set bit	X	1	0	120 kHz
F1	Upper freq set bit	X	0	1	160 kHz
			1	0	200 kHz

Configuration

Table 15: Configuration List

Pin (Driver IC)	Name (Driver IC)	GPIO	Function
1	SCL	8	I ² C compatible input clock
8	SDA	9	I ² C compatible input data stream

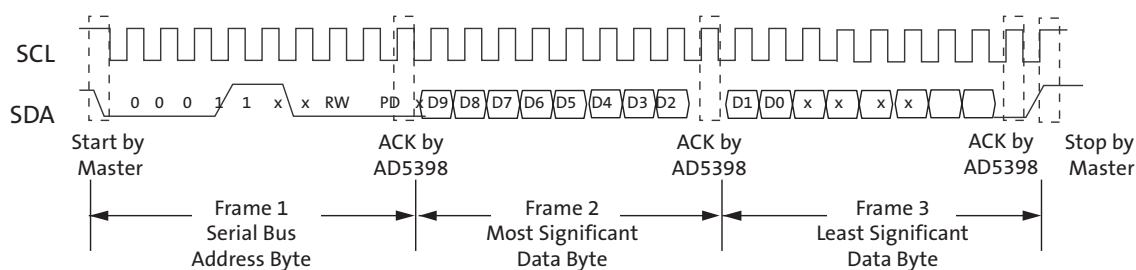
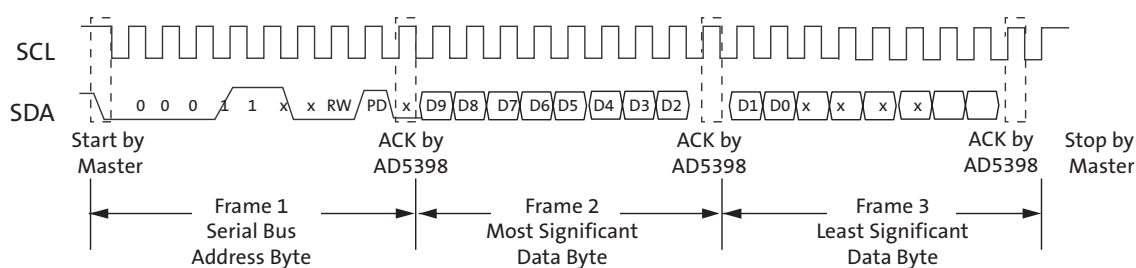
Figure 39: Configuration Schematic



AD5398 (I²C)

Interface Protocol Waveform

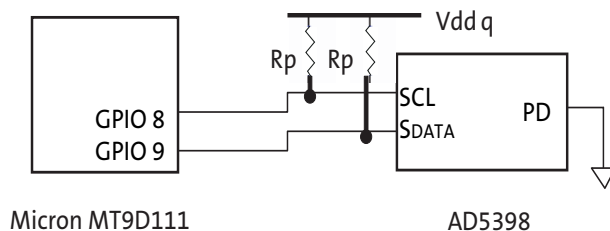
Data is written to the AD5398 high byte first, MSB first, and is shifted into the 16-bit input register. After all data is shifted in, data from the input register is transferred to the DAC register. Because the DAC requires only 10 bits of data, not all bits of the input register data are used. The MSB is reserved for an active-high, software-controlled, power-down function. Bit 14 is unused; bits 13 to 4 are DAC data; bits 9 to 0 and bits 3 to 0 are unused. During a read operation, data is read back in the same bit order.

Figure 40: Timing Diagram

AD5398 WRITE OPERATION

AD5398 READ OPERATION

Configuration

Table 16: Configuration List

Pin Driver IC)	Name (Driver IC)	GPIO	Function
3	SCL	8	I2C Interface Signal
4	SDA	9	I2C Interface Signal

Figure 41: Configuration Schematic


LB1935T/CL (Stepper)

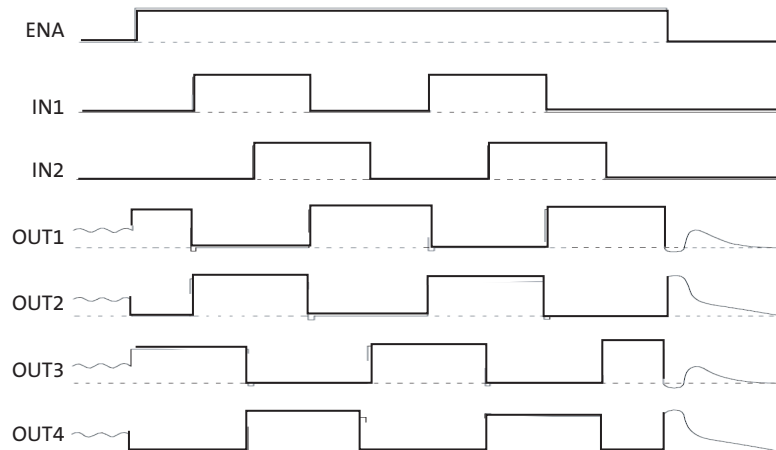
Interface Protocol Waveform

LB1935T/LB1935CL is IC with forward/reverse motor drives 2-channel in which low saturation voltage and low voltage operation are possible. Its small sized package is optimal for 2 phase excitation drive of 2 phase bipolar stepping motors for various portable devices such as digital still cameras. The photointerrupter is SG290 by KODENSHI.

Table 17: Truth Table

INPUT			OUTPUT				REMARKS
ENA	IN1	IN2	OUT1	OUT2	OUT3	OUT4	
L	—	—	OFF	OFF	OFF	OFF	Stdby
H	L	L	H	L	H	L	2-phase excitation
	L	H	H	L	L	H	
	H	H	L	H	L	H	
	H	L	L	H	H	L	

Figure 42: Timing Diagram



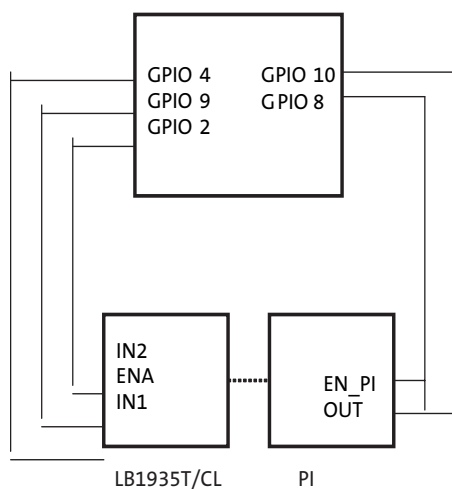
Configuration

Table 18: Configuration List 1

Pin (Driver IC)	Name (Drive IC)	GPIO	Function
2	IN2	2	Input signal 1
3	ENA	9	Enable signal
4	IN1	4	Input signal 2

Table 19: Configuration List 2

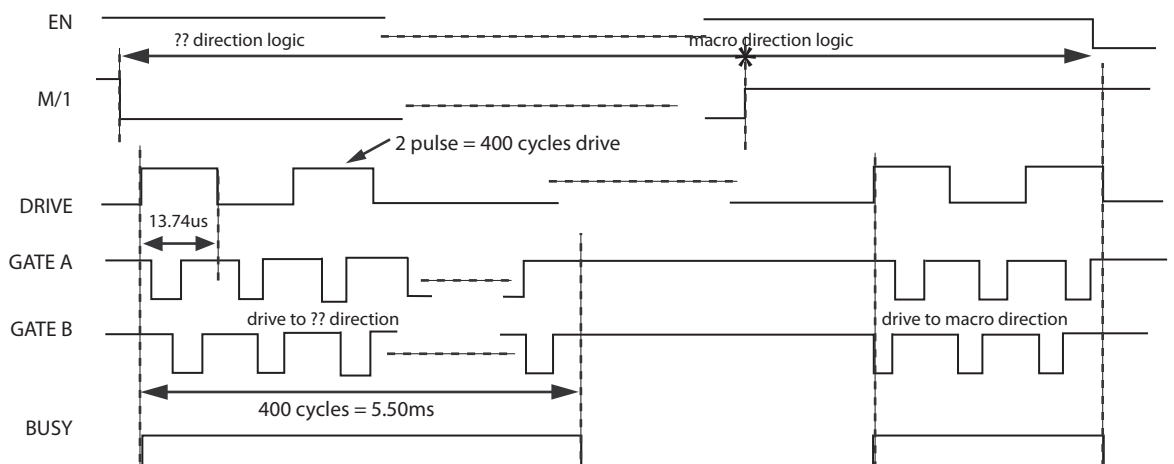
Pin (PI)	Name (PI)	GPIO	Function
1	EN_PI	8	Input signal
4	OUT	10	Output signal

Figure 43: Configuration Schematic


LV8071LP (Piezo)

Interface Protocol Waveform

Figure 44: Timing Diagram



Configuration

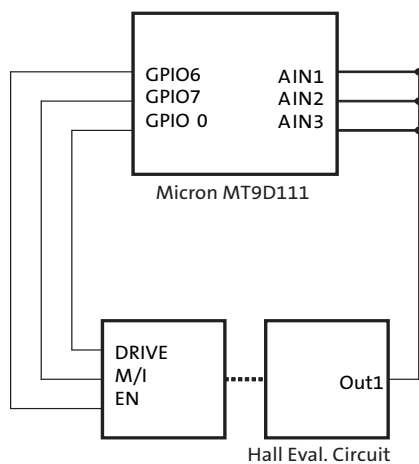
Table 20: Configuration List 1

Pin (Driver IC)	Name (Driver IC)	GPIO	Function
1	DRIVE	0	Set the operation waveform
15	EN	6	Enable/disable the function of IC
16	M/I	7	Set the actuator operation direction
6	BUSY	1	Indicate if the actuator is operating or not

Table 21: Configuration List 2

Pin (Evaluation Circuit)	Name (Evaluation Circuit)	Pin (Sensor)	Function
1	Hall-out	AIN1	Hall element evaluation output
1	Hall-out	AIN2	Hall element evaluation output
1	Hall-out	AIN3	Hall element evaluation output

Figure 45: Configuration Schematic

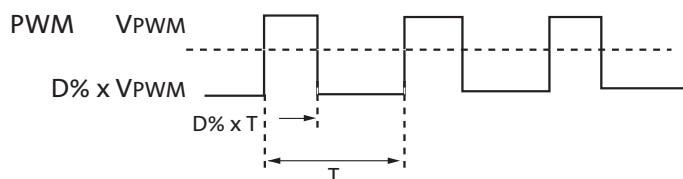


Note: The registers corresponding to analog input AIN1–3 are located at R0×E2:0, R0×E1:0, and R0×E0:0. The control bit to enable the sampling those signals is at R0×E3:0[15]. Certain average function is needed to achieve 7-bit resolution.

MD115/ID9701 (PWM)

Interface Protocol Waveform

Figure 46: Timing Diagram



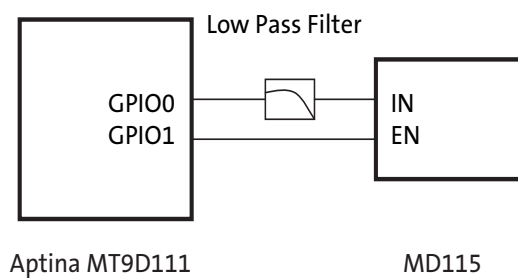
Configuration

MD115 (PWM)

Table 22: Configuration List 1

Pin Driver IC	Name (Driver IC)	GPIO	Function
2	IN	0	Motor input
6	PD	1	IC power down

Figure 47: Configuration Schematic

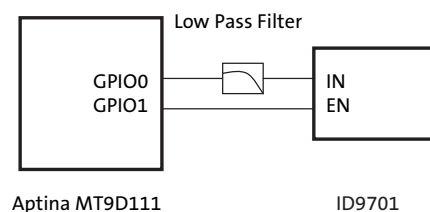


ID9701(PWM)

Table 23: Configuration List

Pin (Driver IC)	Name (Driver IC)	GPIO	Function
2	IN	0	IC input
6	PD	1	IC enable

Figure 48: Configuration Schematic



Lens Actuator Control

Logical and Physical Lens Position

As previously mentioned, the main task of the AFM driver is to translate every change in the logical lens position requested by the AF driver into an appropriate change of physical lens position. It is somewhat difficult to generally specify what physical position change is an appropriate response to AF driver's command to move the lens from one logical position to another. Perhaps the best way to do it is to define an ideal relation between the logical and physical lens position and then make this relation more “fuzzy” and thus a better approximation of the real world. Ideally, we would like the physical lens position, P , to be a fixed monotonic function of the logical position, L . This postulate can be written as an equation

$$P = f_m(L) \quad (\text{EQ 3})$$

where the subscript m indicates the monotonicity of the function. If this equation were always true, then at any time t , after any arbitrarily long and complicated sequence of lens movements, knowing the logical position would be sufficient to know the physical position, since

$$P(t) = f_m(L(t)) \quad (\text{EQ 4})$$

Unfortunately, due to the existence of hysteresis and other real-world complications, such simple 1-to-1 relation between the logical and physical position is impossible to maintain over time. As a lens is moved here and there by a real actuator, every physical position at which it stops, P_n ($n = 0, 1, 2, \dots$), is a function of the history of its movements since at least actuator initialization ($n = 0$). For theoretical manageability, this history can be reduced to a list (vector) of logical positions to which the AFM driver was commanded to move the lens:

$$H_n = (L_n, L_{n-1}, L_{n-2}, \dots, L_0) \quad (\text{EQ 5})$$

Naturally, the latest element of this vector is the current logical position, L_n . The dependence of the current physical position, P_n , on each component of the vector can be in general arbitrarily significant, hence we should write

$$P_n = g(H_n) = g(L_n, L_{n-1}, L_{n-2}, \dots, L_0) \quad (\text{EQ 6})$$

where g is a sort of “black box” function combining the effects of control exercised by the AF driver with the effects of “forces of chaos:” hysteresis, friction, etc. However, for the physical lens position to be controllable by the AF driver, its dependence on past positions (history) must be always much weaker than the dependence on the current logical position. Contribution of the history to the current physical position must therefore be inherently non-cumulative or cancellable by means of certain sequence of movements. This postulate can be written as:

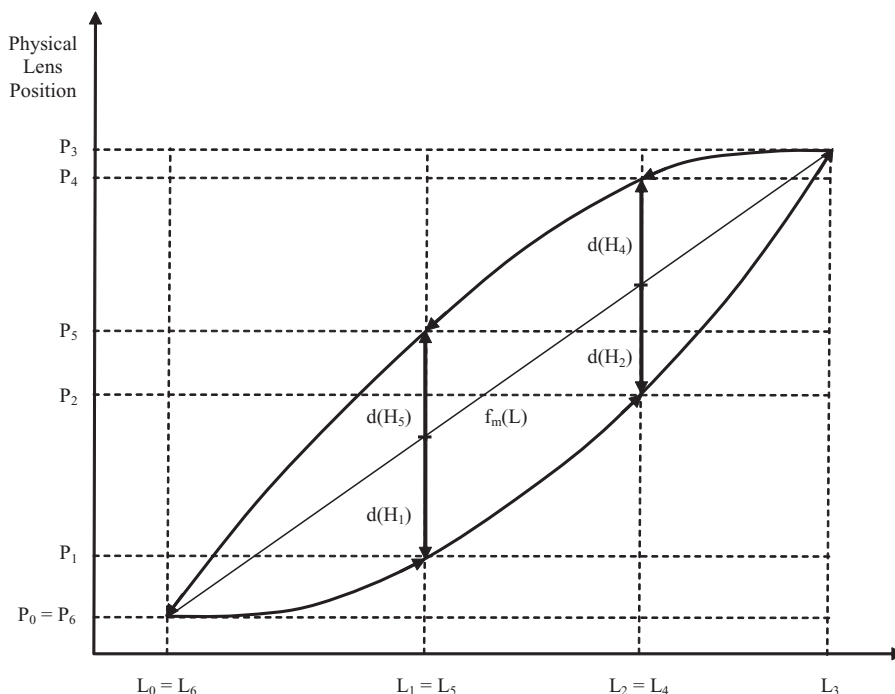
$$P_n = f_m(L_n) + d(H_n) \quad (\text{EQ 7})$$

where $d(H_n)$ is small compared to the range of physical positions

$$P_n = f_m(L_n) + d(H_n) \ll |f_m(L_{min}) - f_m(L_{max})| \quad (\text{EQ 8})$$

Not much more can be said about the relation between the logical and physical lens position without reducing the generality of this discussion. To make the discussion a bit less abstract, Figure 49 illustrates some of the concepts using a typical hysteresis loop.

Figure 49: Example of Hysteresis-affected Relation Between Physical and Logical Lens Position



In Figure 49, the physical position, P , is not a simple function of the logical position, L , but depends also on history of previous lens movements, H . This dependence may be expressed by adding a history-dependent correction, $d(H)$, to a function $f_m(L)$ representing an ideal relation between P and L . What the function $f_m(L)$ should be, aside from being monotonic, is largely a matter of choice. One possible choice of $f_m(L)$ is depicted by the straight line connecting points (L_0, P_0) and (L_3, P_3) . Four examples of corresponding history-dependent corrections are depicted by vertical arrows. Instead of this linear function, one could choose as $f_m(L)$ one of the branches of the hysteresis loop—the ascending one. With this choice of $f_m(L)$, $d(H)$ would be nonzero on the descending branch only and would equal the difference between the branches.

In REV5, the control over AD5398 involves mapping 8-bit logical lens position used internally by the AF and AFM drivers to 10-bit setting of the AD5398 DAC. The mapping can be done by adding a user-programmable offset to the logical lens position and/or left-shifting it by up to 3 bits. The offset is programmable via the variable `afm.posMacro` and bits [5:3] of `afm.custCtrl`. The size of the left-shift (always done after adding the offset) equals the value of bits [7:6] of `afm.custCtrl`. By default, `afm.custCtrl` and `afm.posMacro` are set to "0."

The equation for the physical position (accepted by AD5398) P (0~1023)

1) When `afm.custCtrl` [3] =1

$$P = (L + \text{afm.posMacro} + m) \cdot 2^n$$

2) When `afm.custCtrl[3] = 0`

$$P = (L + m) * 2^n$$

where, $m = \text{afm.custCtrl}[4:5] * 256$

$$n = \text{afm.custCtrl}[6:7]$$

L is the logic position (0 ~ 255)

According to the above formula, you can set `custCtrl` and `posMacro` properly to change the travel distance.

Managing Lens Actuator Hysteresis

Like all macroscopic mechanical devices, lens actuators have intrinsic hysteresis, whose effect is to make physical position of the lens a function of not only the current logical position set by the AF driver, but also of previous lens movements. The problem that the dependence of the physical position on history poses for the scan AF algorithm can be summarized thus: Knowing which logical lens position has been found best in the first or second scan, how to return the lens to the physical position that produced the winning sharpness scores?

As can be seen in Figure 49 on page 104, unless a logical position is very close to the low or high of the logical position range, there are two very different physical positions corresponding to it, one on the ascending branch of the hysteresis loop and another on the descending branch. For example, logical position L_1 corresponds to physical positions P_1 and P_5 . Suppose that L_1 has been found to be the best logical position in a scan that started at L_0 and ended at L_3 . The physical position corresponding to L_1 was P_1 , because physical movements of the lens during the scan corresponded to moving up the ascending branch of the hysteresis loop. The lens ended at P_3 and now the question is: how to return it to P_1 or close enough to get focus as good as at P_1 ?

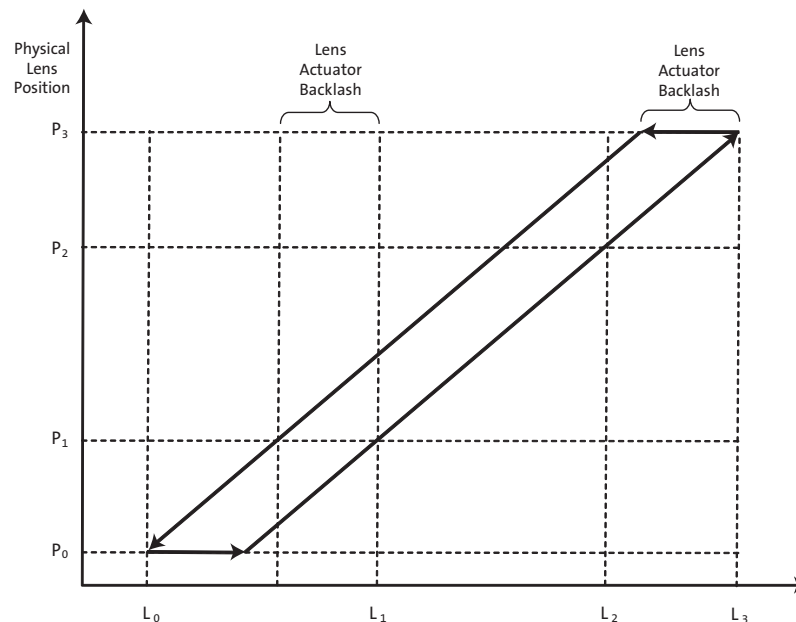
The AF driver has essentially two ways to do it. The first way is to command the AFM driver to move the lens to back L_0 and then from there to L_1 . This should bring the lens first to P_0 down the descending branch of the hysteresis loop, and then back to P_1 along the ascending branch. The assumption on which this “flyback” procedure relies is that the hysteresis loop is closed, i.e. the end point of its descending branch, (L_6, P_6) , is the same as the starting point of the ascending branch, (L_0, P_0) . If this assumption is correct or at least the difference between P_0 and P_6 is very small, then the flyback procedure allows one to bring the lens back to P_1 blindly, without knowing the exact shape of the hysteresis loop.

It is necessary to know the shape of the loop to use the second method of returning the lens from P_3 to P_1 , which has been mentioned previously as the direct move to best focus position involving a backlash-compensating step. The direct move means going from P_3 to P_1 down the descending branch of the hysteresis loop. The problem that the AF driver has in commanding such a move is that it can only give a command to go to a certain logical position, but the logical position corresponding to P_1 on the descending branch is impossible to calculate without knowing the shape of the branch. The AF driver does not have any means to memorize, much less determine, the exact shape of the hysteresis loop of the lens actuator it works with, so it estimates the logical position it needs by subtracting from the position L_1 a fixed value stored in the variable `afm.backlash`.

This estimate is generally crude because the subtracted value should vary with P_1 to match the varying horizontal width of the hysteresis loop. However, it can be reasonably accurate if `afm.backlash` is correctly set to the average horizontal width of the loop and the loop is much slimmer than the one shown in Figure 49. It is completely accurate only

if the hysteresis loop has a parallelogram shape, like that shown in Figure 50 and its horizontal width equals `afm.backlash`. If these conditions are met, then going from L_3 to the logical position $(L_1 - \text{afm.backlash})$ along the descending branch of the loop can be viewed as taking two distinct logical steps: a step of length `afm.backlash` that compensates for lens actuator backlash and has no physical effect on the lens, and another step of length $(L_3 - L_1)$ that brings the lens to the desired physical position P_1 . Hence, `afm.backlash` is referred to below as the logical length of backlash-compensating step.

Figure 50: Hystereis Loop Typical for Simple Mechanical Gears



Note: The loop above has a parallelogram shape and therefore a constant horizontal (logical) width. This width is often called backlash.

Timer

As previously mentioned, one of the tasks of the AFM driver critical for proper functioning of the AF algorithm is to correctly and constantly communicate to the AF driver whether the lens it controls is stationary or moving. This is required to prevent the AF driver from using sharpness scores distorted by lens motion and from issuing new commands to move the lens while a previous one is still being executed. The AFM driver indicates that the lens is moving by setting bit 1 of the variable `afm.status`, which also carries information about lens actuator errors and other aspects of its status. Obviously, the value of this variable must be updated to correctly reflect true actuator status. The AFM driver includes lens-actuator-specific functions named `AFM_GetStatus*` (where * stands for a lens actuator identifier) that update `afm.status` and return its updated value when called. The AF driver calls one of them indirectly (using the pointer `afm.vmt->pGetStatus`), whenever it wakes up and needs to know if the lens is in motion.

A function updating `afm.status` must have some source of current information about lens movement or, at least, about the time elapsed since the lens actuator last received a command to move the lens and time required to execute this command. The most reli-

able source of current information would be a lens actuator outputting a digital busy signal while moving the lens. Unfortunately, most actuators do not provide such feedback. However, the waveform generator in the GPIO module can provide such a signal, if it is the source of waveforms driving lens movements from start to finish, not just starting them. The time when these waveforms are generated can be identified with the time of lens motion, even though the motion may really die down a while after the waveforms are finished. A busy signal from the waveform generator is therefore as good as one from the lens actuator—unless the actuator is not responding to the driving waveforms as expected.

Lens actuators that make it most difficult to wait through lens movements meet two criteria:

1. They are command-driven (meaning their movements are started, not sustained, by input signals)
2. They provide no motion-indicating feedback

Helimorph or VCM lens actuators incorporating two-wire serial interface-enabled digital-to-analog converters (DACs) are good representatives of this type. If a command-driven lens actuator does not provide any feedback about lens motion after receiving a command to move the lens from position A to position B, the AFM driver must somehow predict how long the execution of this command will take. After making a prediction, it must keep track of the time elapsed since the command was given, and until this time reaches the predicted value, answer all inquiries about lens actuator status by setting bit 1 of `afm.status` to 1. As long as the AFM driver does not underestimate the actual duration of lens movements, interactions of the AF driver with the FME and lens actuator are safely timed. To time them optimally and thus achieve the quickest possible AF, one must refine the AFM driver's predictions of lens travel time until they perfectly match the real behavior of the lens actuator. In practice, some balance must be struck between the AF speed, safety, and size of code required to model the lens actuator behavior.

The need to keep track of time after initiating lens movements has been the primary reason for including an object-like set of public variables and functions called `timer` in the AFM driver. It could also be called a counter of master clock cycles, but this would lead to confusion with 32-bit master clock cycle counter implemented in hardware that the timer functions simply put to a particular use. The use is to mark a certain moment in time and then periodically check if a pre-set amount of time has elapsed since the marked moment. Alternatively, this can be seen as putting a mark at a point in the future, at a pre-set distance from now, and then checking as time goes by if the mark is still ahead or has been left behind.

Two timer functions, `AFM_TimerSetDelay` and `AFM_TimerIsStopped`, utilizing three 16-bit timer variables—`afm.timer.startTime`, `afm.timer.stopTime`, and `afm.timer.hiWordMclkFreq`—provide a convenient way to do it, albeit with a somewhat limited accuracy. To minimize code size, both functions read only the upper word (16 most significant bits) of the master clock cycle counter, which limits their time resolution to $dt = 2^{16}/f_{mclk}$, where f_{mclk} is master clock frequency. At nominal $f_{mclk} = 80$ MHz, dt equals approximately 0.82ms. The function `AFM_TimerSetDelay`, which marks a point in the future for the other function to look for, takes as an argument the time interval that should separate that point from the moment of its selection. The interval should be given in milliseconds and must be greater than zero for any point in the future to be marked. This latter requirement is verified immediately after `AFM_TimerSetDelay` is called.

Upon finding that its argument, unsigned integer `wDelay`, is positive, the function does the following:

1. Makes `wDelay = wDelay*afm.timer.hiWordMclkFreq/1000`, to convert it from a time interval in milliseconds to a multiple of `dt`
2. Copies the upper word of the current master clock cycle count to `afm.timer.startTime`
3. Checks if `wDelay < 0xFFFF-afm.timer.startTime`. If it is, then the function makes `afm.timer.stopTime = afm.timer.startTime+wDelay`. Otherwise, `afm.timer.stopTime` is set to `wDelay-(0xFFFF-afm.timer.startTime)`, which makes it less or equal to `afm.timer.startTime`

Upon finding that `wDelay = 0`, `AFM_TimerSetDelay` does not go through the three steps noted above, but instead immediately sets `afm.timer.stopTime = afm.timer.startTime = 0` and returns.

What `AFM_TimerSetDelay` does in step 3 above makes the timer immune to overflows of the master clock cycle counter. Since `afm.timer.startTime` can be anywhere between 0 and 0xFFFF, and so can `wDelay`, their sum can be larger than 0xFFFF. When it is, it means that the clock cycle counter will overflow between the present call to `AFM_TimerSetDelay` and the point in the future that `AFM_TimerSetDelay` is supposed to mark. Upon an overflow, the counter “wraps around”—it automatically resets to 0 and keeps on counting (in other words, what can be read from it at any time is clock cycle count modulo 2^{32}). This behavior is matched by setting `afm.timer.stopTime` to `wDelay-(0xFFFF-afm.timer.startTime)` instead of `afm.timer.startTime+wDelay` when the latter sum exceeds 0xFFFF.

Whatever the value of this sum is, if `wDelay` is greater than 0, `AFM_TimerSetDelay` sets `afm.timer.startTime` and `afm.timer.stopTime` to two different values, usually also greater than 0. By doing so, `AFM_TimerSetDelay` in effect activates the timer and programs it to “tick” until the number of milliseconds received by the function as an argument passes by. At the end of this time—the point in time previously marked by `AFM_TimerSetDelay`—the timer will stop “ticking” and will remain active, but “stopped.” In similar figurative terms, the effect of calling `AFM_TimerSetDelay(0)` can be described as de-activating and resetting the timer. When de-activated, the timer can also be thought of as “stopped.”

The argument-less Boolean function `AFM_TimerIsStopped` answers whether the timer is presently “stopped” or “ticking” (in other words, if the delay set by `AFM_TimerSetDelay` is over or not). It also automatically de-activates and resets the timer if it is “stopped.” The C code of this function is so short and simple that it seems better to give it verbatim below rather than translate it into plain English.

```
BYTE AFM_TimerIsStopped(void)
{
    WORD t;

    if (afm.timer.startTime < afm.timer.stopTime) {
        t = sys.ClockCnt.Word.Hi;
        if ((t > afm.timer.startTime)&&(t < afm.timer.stopTime)) return 0; // timer is ticking
    }
    else if (afm.timer.startTime > afm.timer.stopTime) {
        t = sys.ClockCnt.Word.Hi;
        if ((t > afm.timer.startTime)|| (t < afm.timer.stopTime)) return 0; // timer is ticking
    }
    afm.timer.startTime = 0;
    afm.timer.stopTime = 0;
    return 1; // timer is stopped
}
```



```
} // End of AFM_TimerIsStopped
```

A few words and symbols in this code may require clarification. The variable types BYTE and WORD are equivalent to unsigned char and unsigned int, respectively. The value assignment `t = sys.ClockCnt.Word.Hi` makes the local variable `t` equal to the upper word of the current count of master clock cycles. The symbols `&&` and `||` represent logical operations AND and OR, respectively. Everything that follows `//` in a line is a comment.

To complete the discussion of the functions, `AFM_TimerSetDelay` and `AFM_TimerIsStopped`, consider an example of their use to track lens motion time. Suppose that the AF driver has given the AFM driver a command to move an AF lens from its current logical position, `afm.curPos`, to logical position `af.positions[0]`. The AFM driver has received this command, estimated that executing it will take 80 milliseconds, passed it to a helimorph lens actuator via a serial interface, called `AFM_TimerSetDelay(80)`, set bit 1 of `afm.status` to 1, and returned control to the AF driver. The AF driver now goes to sleep until `af.wakeUpLine` in the next frame, about 33ms. When it wakes up again, the first thing it will need is the answer to the question: is the lens moving or not? To give the AF driver an answer, the AFM driver includes the following public function:

```
BYTE AFM_GetStatusHelimorph(void)
{
    if (AFM_TimerIsStopped()) {    // lens is not moving
        afm.status &= ~2;    // clear bit 1 of afm.status
    }
    return afm.status;
}

// End of AFM_GetStatusHelimorph
```

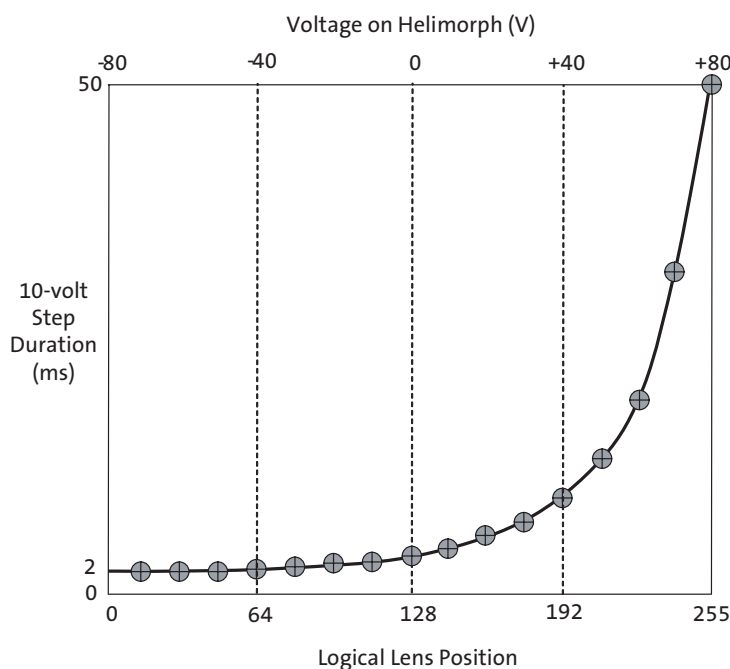
When the AF driver calls this function after a 33ms sleep, it will return `afm.status` with bit 1 equal to 1, because the return value of `AFM_TimerIsStopped` after 33ms will be 0. Upon finding bit 1 of `afm.status` set, the AF driver will immediately go back to sleep for another 33ms. After waking up and calling `AFM_GetStatusHelimorph` again, it will find bit 1 of `afm.status` unchanged. Another 33ms sleep will follow. Finally, on the third call, 99ms after issuing the command to move the lens, the AF driver will find the timer stopped, which will result in clearing bit 1 of `afm.status`, which the AF driver will take as a signal that the lens has stopped moving. Of course, this signal will not reflect actual lens status, but the status of the timer, determined by the AFM driver's prediction that the lens movement commanded by the AF driver should take 80ms.

The final topic to discuss here is how the AFM driver can make such predictions. In its current version, the need for predicting lens travel time is met in a rather simple way. The driver includes a public function `AFM_TimerSetTimeToMove` that takes as arguments two logical lens positions and roughly estimates the time required to move the lens between them. The function can use two different estimation methods, both of which rely on three user-set parameters, `afm.timer.maxShortDelay`, `afm.timer.maxLongDelay`, and `afm.timer.maxQuickMove`, as a sole source of information about how fast the lens actuator moves the lens. The default method of piecewise linear estimation is used when bit 0 of `afm.timer.config` is cleared. Setting this bit to 1 enables the alternative bipolar method. The bipolar method is very simple: if the distance between the two logical positions given to `AFM_TimerSetTimeToMove` as arguments exceeds `afm.timer.maxQuickMove`, then `afm.timer.maxLongDelay` is selected as the proper lens travel time estimate. Otherwise, unless the two logical positions are the same, the estimate equals `afm.timer.maxShortDelay`. If the two positions are the same,

the estimate should be 0, and indeed is 0 if bit 1 of `afm.timer.config` is cleared. However, if this bit is set to 1 and the positions are the same, the function `AFM_TimerSetTimeToMove` outputs `afm.timer.maxShortDelay` instead of 0.

The slightly more complicated piecewise linear estimation method has been developed to model temporal characteristics of helimorph lens actuators. The position of a lens mounted on one of those actuators is a fairly linear function of DC voltage applied to its helimorph. In other words, changing this voltage causes the lens to move a distance proportional to the voltage change. The voltage can range from negative to positive, e.g. from -80 V to 80 V. Helimorph response to voltage adjustments in the negative and positive directions is roughly symmetrical, so it is enough to consider only changes in the positive direction. Lens movements caused by decreasing the absolute value of the voltage are generally faster than movements caused by increasing it, especially when the absolute value is already high. As a result, when the lens moves across its logical position range, from 0 to 255, in steps of equal length, $l_s \ll 255$, the time needed to take one step, t_s , is a function of lens position, $P = i \cdot l_s$, where $i = 1, 2, 3, \dots$. The plot of this function, $t_s(p)$, looks like Figure 51.

Figure 51: Time Needed to Increase Voltage on Helimorph by 10V as a Function of Lens Position



Note: Time needed to step up voltage applied to helimorph as a function of the voltage and of the corresponding logical lens position. The curve shows that it does not take long to go, for example, from -80V to -70V, from -70V to -60V, etc., because the absolute value of the voltage is reduced with each step until it becomes 0. However, stepping up from 0 to 10V, from 10V to 20V, and so on gets progressively more difficult and slow. The time required to move the lens between any two logical positions can be estimated by summing up the durations of 10-volt steps connecting the two positions. For example, a jump from lens position 0 (voltage of -80V) to lens position 64 (-40V) should take about 8ms, because it corresponds to four +10V steps, each of which takes about 2ms. On the other hand, the jump back from position 64 to position 0 (-40V to -80V) takes much longer, because the portion of the curve corresponding to it is between lens positions 192 and 255 (+40V to +80V).

What makes $t_s(p)$ interesting is that by integrating it over an appropriate range of p one can estimate the duration of every conceivable lens movement (see the note below Figure 51 for an example). The integration can be done individually for every pair of positions between which the lens may have to be moved, but this approach is not very efficient. Instead of computing definite integrals of a function (especially in real time) it is usually better to take a look at its indefinite integral and see if that function can be conveniently approximated. Since $t_s(p)$ is defined only for $p = i \cdot l_s$, where $i = 1, 2, 3, \dots$, what we really want to look at is a function $t_0(p)$ defined by

$$t_0(p) = \sum_{i=0}^{\text{int}\left(\frac{p}{l_s}\right)} t_s(i+1) \quad (\text{EQ 9})$$

As it turns out, when $t_s(p)$ is like that depicted in Figure 52, $t_0(p)$ can be reasonably well approximated by two pieces of linear functions, as shown in Figure 52. All that is needed to define these two pieces are just three parameters like those of the function `AFM_TimerSetTimeToMove`. In other words, a piecewise linear function $T(p)$ approximating $t_0(p)$ can be computed for every p using only very simple arithmetic and four integer numbers: p , `afm.timer.maxShortDelay`, `afm.timer.maxLongDelay`, and `afm.timer.maxQuickMove`. This makes estimating lens travel times rather trivial. For every two lens positions, p and q , the time required to move the lens from p to q approximately equals

$$t_+(p, q) = T(q) - T(p), \text{ if } p < q \quad (\text{EQ 10})$$

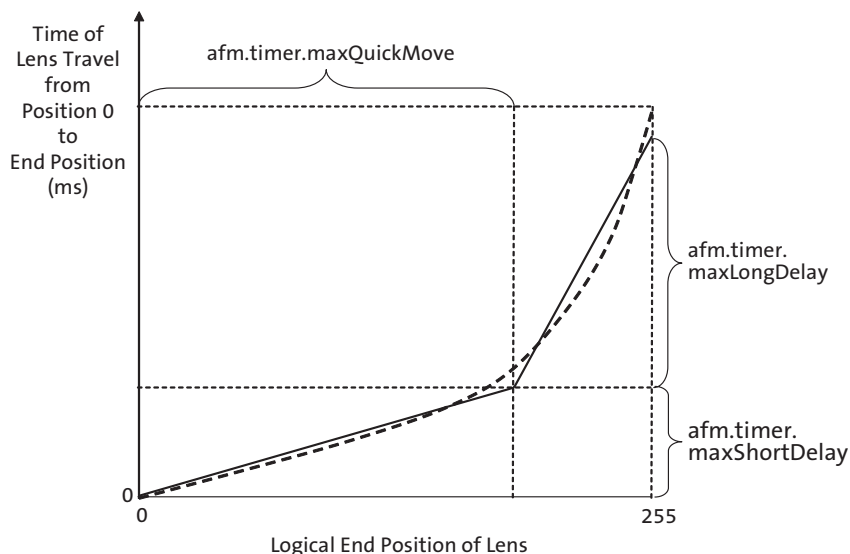
and

$$t_-(p, q) = t_+(255 - p, 255 - q), \text{ if } p > q \quad (\text{EQ 11})$$

The function `AFM_TimerSetTimeToMove` uses Equation 10 and Equation 11 to estimate lens travel time when the positions p and q given to it as arguments differ from each other and bit 0 of `afm.timer.config` is 0. When $p = q$, the function produces a travel time estimate of 0 or `afm.timer.maxShortDelay`, depending on the current value of bit 1 of `afm.timer.config`.

It is worth underscoring that the piecewise linear estimation method described above is not intended for use with helimorph lens actuators only. The three programmable parameters of the function $T(p)$ make it very adaptable. Adapting it for use with other lens actuators should definitely be considered before developing any alternative solution.

Regardless of the method it employs to estimate lens travel time, the function `AFM_TimerSetTimeToMove` always programs its estimate into the timer using a direct call to the function `AFM_TimerSetDelay`. Hence, a single call to `AFM_TimerSetTimeToMove` is all that is required to start the timer “ticking” after commanding a lens actuator to move.

Figure 52: Piecewise Linear Function Used by AFM Driver to Estimate Lens Travel Time


Note: Piecewise linear approximation of the function $t_0(p)$ defined by Equation 9 on the preceding page. The dashed curve is a smoothed plot of this function that does not show its steps, but only the overall shape that must be approximated. The solid line shows one possible approximating function $T(p)$.

Serial Interface

The AFM driver includes a small library of public functions supporting two-way serial communication between the MT9D111 microcontroller and any two-wire serial interface external devices through two user-selected GPIO pads. The following functions belong to this library:

- void AFM_SiSendCmd(WORD wCmd)
- void AFM_SiSetActvFlag(BYTE bOn)
- BYTE AFM_SiSendByte(BYTE bByte)
- void AFM_SiRecvByte(BYTE *pDestByte)
- They can be called indirectly using the following pointers:
- afm.si.vmt->pSendCmd
- afm.si.vmt->pSetActvFlag
- afm.si.vmt->pSendByte
- afm.si.vmt->pRecvByte

The functions use the following variables as configuration parameters:

- afm.si.clkMask
- afm.si.dataMask
- afm.si.clkQtrPrd
- afm.si.needsAck
- afm.si.slaveAddr

The first two of the above variables select the GPIO pads that all the functions use as the clock and data lines. The third variable sets the rate of the two-wire serial interface clock: the rate is approximately inversely proportional to `afm.si.clkQtrPrd`. The fourth variable enables/disables detection of ACK bits. The fifth variable specifies the device address that the functions must use to communicate with the external device of interest.

The function `AFM_SiSendByte` sends its 1-byte argument to the external device and returns 1 or 0, depending on whether the device has responded with an ACK bit and whether such a response is needed. If `afm.si.needsAck = 1`, it means that ACK is needed. `AFM_SiSendByte` returns 1 upon receiving it. If it is missing, the function returns 0. If `afm.si.needsAck = 0`, `AFM_SiSendByte` always returns 1.

The function `AFM_SiRecvByte` receives a byte of data from an external two-wire serial interface transmitter and stores it at the memory location pointed to by its argument. It acknowledges receiving the data by sending an ACK bit back to the transmitter.

The function `AFM_SiSetActvFlag` is for generating START and STOP bits that bracket every two-wire serial interface transmission. If its argument is 0, it sends a STOP bit. Any other argument produces a START bit.

The function `AFM_SiSendCmd` can send a 2- or 3-byte long command, the first byte of which is always equal to `afm.si.slaveAddr`. The second command byte equals the lower byte of the function's 16-bit argument, `wCmd`. The upper byte of `wCmd` is sent next if it differs from `0xFF`. If it equals `0xFF`, it is discarded. The entire command is bracketed by START and STOP bits generated by `AFM_SiSetActvFlag`.

Each command byte is sent using the function `AFM_SiSendByte`. If after sending some byte `AFM_SiSendByte` returns 0 (indicating the absence of a needed ACK bit), the transmission of the command is aborted and restarted from the first byte. Up to two restarts can occur if the problem of missing ACK bits persists. The function `AFM_SiSendCmd` uses `afm.status` variable to count the restarts and report the final result of the command transmission. A successful transmission is indicated by `afm.status = 0`. If `afm.status > 0`, all three attempts to send the command have failed due to the lack of acknowledgement from the intended command recipient.

Initial Positioning of Stepper Motors

Lens actuators powered by stepper motors convert rotational (or periodic) motion of a stepper motor shaft to linear motion of a lens, usually by means of a helical gear. As the name stepper motor indicates, rotation of the shaft is not smooth and continuous, but occurs by stepping between several fixed angular positions distributed evenly within the 0-to-360-degrees range. Each of these positions corresponds to a unique polarization of stepper motor windings, brought about by applying a unique set of voltages to motor inputs. Likewise, each position of the lens corresponds to a single state of stepper motor inputs. However, unless the motor is permitted to rotate its shaft by no more than 360 degrees, each angular position of the shaft corresponds to many positions of the lens—even if we assume that the gear connecting the shaft to the lens is completely backlash-free.

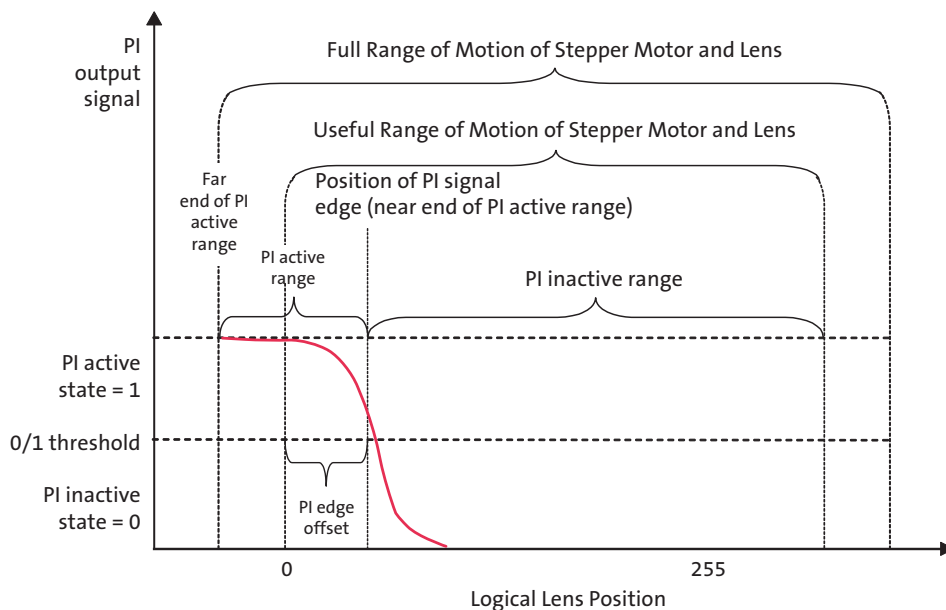
One consequence of this is that when a stepper motor in a lens actuator is powered up and its windings polarized in a certain way, the lens position cannot be determined from the state of the motor. If the initial lens position is unknown, it is impossible to tell how far the lens can be moved forward and backward without crossing the limits of its useful motion range. Going outside this range can cause mechanical problems which negatively affect further lens movements (jamming of the helical gear, for example). Just as important, an uncertain lens position may also prevent the AF algorithm from moving

the lens over the whole range and finding best focus position within it. In order to avoid both overshooting and never reaching the limits of the useful motion range, at least one fixed lens position in the range must have a unique “signature” allowing the device controlling the lens actuator to find it after actuator power-up, irrespective of the initial position of the lens. In other words, the controlling device must receive from the actuator some feedback signal depending on lens position and having at least two values (0 and 1, for example).

The position signature provided by this signal can be a certain signal value or signal change that occurs only at one lens position. The position with the signature must be fixed in the sense that the distances between it and the ends of the useful motion range must not change in time, must not change from one actuator to another and/or must be made known to the actuator controller before actuator initialization. If these conditions are met, once the lens is brought sufficiently close to the position with the signature, it can be moved anywhere within its useful motion range without losing track of its position. All that is required to keep the lens position known, even after a very long sequence of movements, is to count stepper motor steps and take certain measures to avoid or compensate backlash. Of course, this is true under the assumption that the lens actuator is mechanically sound and lens movements are not hindered in any way.

The device most often used to produce position signature in stepper-motor-powered lens actuators is called photointerrupter (PI). The photointerrupter consists of a light-emitting diode (LED) and a phototransistor positioned in such a way that the light from the LED can alternatively fall on the phototransistor or be blocked by a designated part of the moving lens mount, depending on lens position. Typically, the light is blocked over most of the lens motion range and no light throughput results in low output signal from the PI. As depicted in Figure 53, the light throughput begins to increase gradually as the lens crosses certain point near one end of its motion range and reaches 100 percent before that end is reached. As a result, over a short range of lens positions, the PI output signal is higher than elsewhere. Its exact value depends on phototransistor bias and can be set above the digital low/high threshold of the device controlling the lens actuator. A digital input of the device can then be used to sense the PI output. Sensed in this way, the PI output signal is reduced to two values, 0 and 1, between which the signal jumps only once as the lens is moved one way or the other across its motion range. The single PI signal transition or edge occurring at a fixed point is the simplest signature sufficient to disambiguate and then track lens position linked to angular position of stepper motor.

Figure 53: Typical Relation Between Photointerrupter Output Signal and Lens Position



Note: Typical relation between photointerrupter (PI) output signal and lens position in a camera module where the PI is used for initial positioning of the lens.

The AFM driver in the MT9D111 includes a function called `AFM_ResetStMotor`, whose main task is to perform initial positioning of stepper-motor-powered, PI-equipped lens actuators. The initial positioning involves disambiguation of physical lens position by finding the PI signal edge, moving the lens a pre-set number of stepper motor steps forward or backward from this edge, and equating the physical position thus reached with the logical position 0 or 255. More precisely, `AFM_ResetStMotor` accomplishes its main task in the following steps:

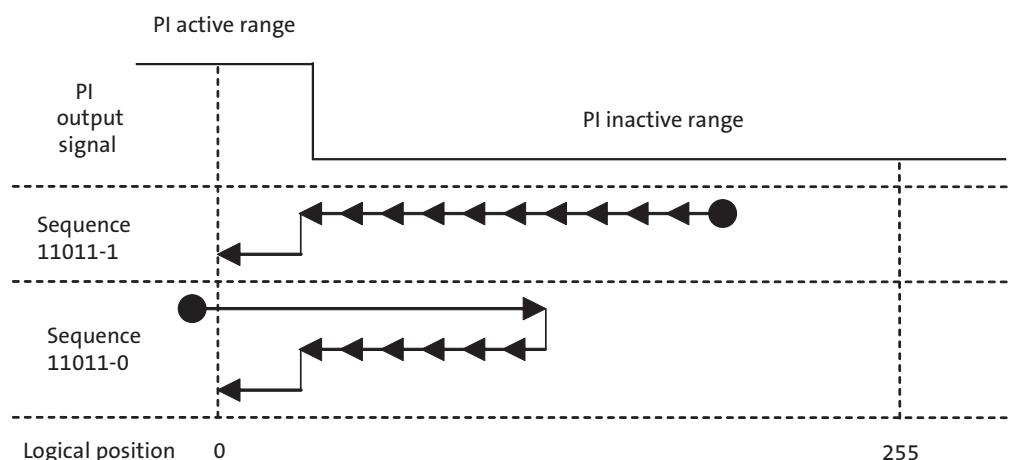
1. Powers up the PI by setting high the GPIO output selected by `afm.sm.piEnabMask`.
2. Waits a short time (proportional to bits [7:5] of `afm.sm.piConfig`) for the PI output signal to settle after power-up.
3. Reads the PI output signal through the GPIO input selected by `afm.sm.piOutMask`.
4. Compares current digital value of the PI output to bit 2 of `afm.sm.piConfig` (this bit identifies PI output value expected in lens position range referred to in Figure as PI active range).
5. If the current value of the PI output matches the value of bit 2 of `afm.sm.piConfig`, moves the lens by half the width of its useful motion range in the direction opposite to that indicated by bit 1 of `afm.sm.piConfig` (this move takes the lens from the PI active range to PI inactive range—see Figure 53).
6. Repeats steps 3 and 4. If the value of the PI output differs from bit 2 of `afm.sm.piConfig` (which indicates that current lens position is in the PI inactive range), goes to step 7. Otherwise, goes to step 8.
7. Moves the lens by 1, 2 or 4 stepper motor steps (according to the value of bits [1:0] of `afm.sm.drvsGenMode`) in the direction of the PI signal edge (indicated by bit 1 of `afm.sm.piConfig`).

8. Moves the lens by `afm.sm.piEdgeOffset` times 1, 2 or 4 stepper motor steps (depending on the value of bits [1:0] of `afm.sm.drvsGenMode`) in the direction indicated by bit 3 of `afm.sm.piConfig` (this move takes the lens from the PI signal edge to its desired initial position).
9. Makes `afm.curPos` equal to 255 times bit 4 of `afm.sm.piConfig` (this links the initial lens position reached in step 8 with logical position 0 or 255).
10. Powers down the PI by setting low the GPIO output selected by `afm.sm.piEnabMask`.

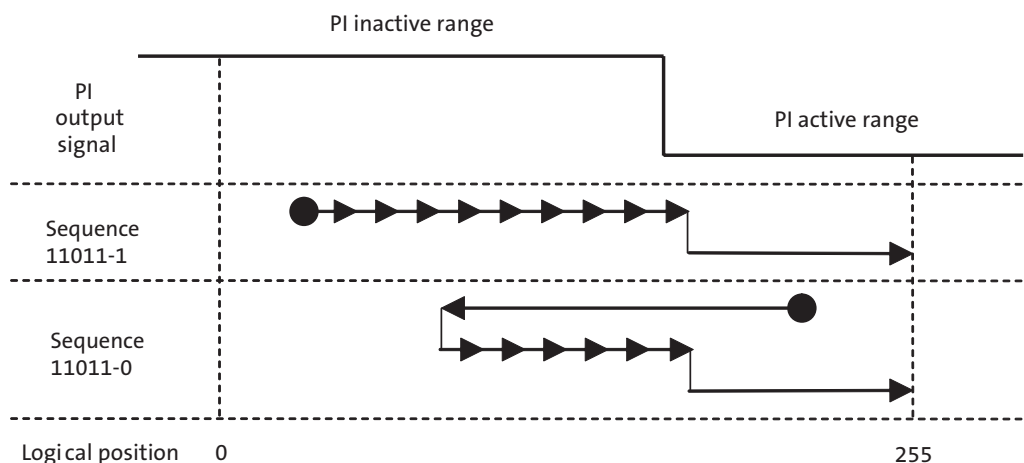
Figure 54, Figure 55, and Figure 56 schematically depict several sequences of lens movements possible during initial stepper motor positioning done by `AFM_ResetStMotor`. Input parameters of this function allow users to choose between different positioning sequences are listed in Table 24, “Programmable Parameters of Stepper Motor Positioning Function,” on page 119. A flowchart of the function follows in Figure 57 and Figure 58.

It should be noted that though by default the function `AFM_ResetStMotor` is executed only at initialization of the AFM driver with `afm.type = 2 + 128 = 130`, it can also be called right before the MT9D111 enters standby mode with `afm.type = 2` and/or right after it exists this mode. The optional pre- and post-standby execution of `AFM_ResetStMotor` is enabled by setting, respectively, bits 4 and 5 of `afm.custCtrl` to 1. Reinitializing a stepper motor at standby may be a way to periodically cancel any slow drift of the lens motion range that may occur, for example because of accumulation of small round-off errors in lens position calculations.

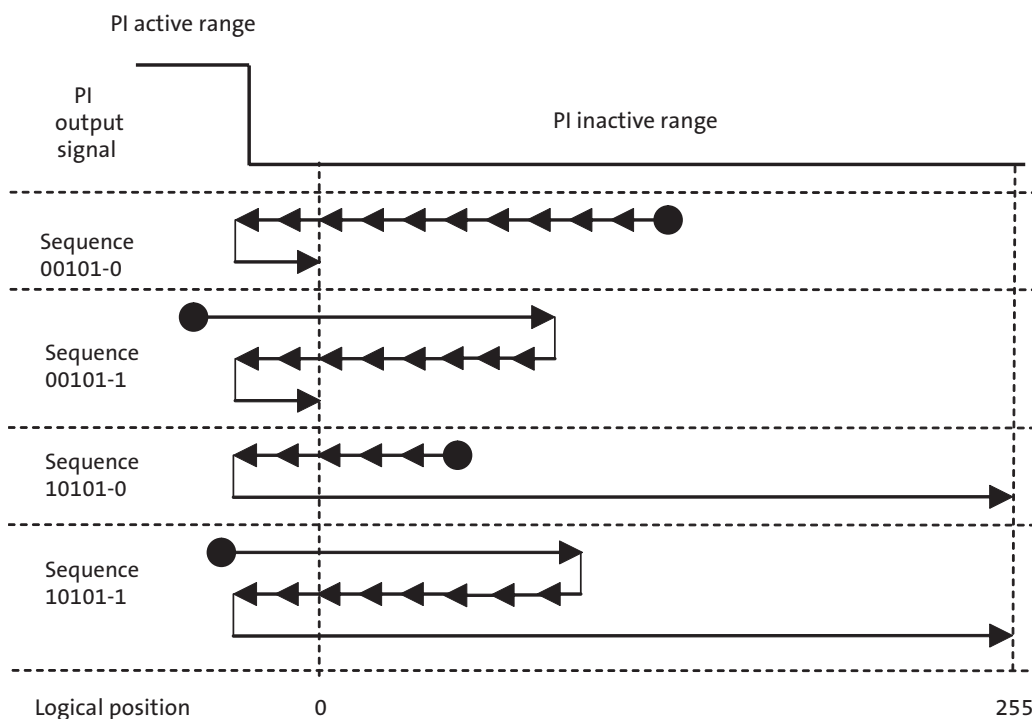
Figure 54: Lens Movements During Initial Positioning of a Stepper Motor (Example 1)



Note: Two sequences of lens movements used for initial stepper motor positioning when bits [4:0] of `afm.sm.piConfig` are set to 13 (binary '01101'). The second sequence results from the requirement that search for PI signal edge must always begin from a lens position where the PI is inactive. The output signal from the PI typically is higher when the device is active (detects some light), but the AFM driver does not require that. The polarity of the output signal is selectable by means of bit 2 of `afm.sm.piConfig`.

Figure 55: Lens Movements During Initial Positioning of a Stepper Motor (Example 2)


Note: Two sequences of lens movements possible when bits [4:0] of `afm.sm.piConfig` are set to 27 (binary '11011'). These sequences differ from the sequences depicted in Figure 54 in the following ways: direction of search for the PI edge is reversed (because bit 1 of `afm.sm.piConfig` is set to 1), PI active state is low (bit 2 equals 0), and `afm.curPos` is set to 255 rather than 0 at the end of each sequence (because bit 4 of `afm.sm.piConfig` equals 1).

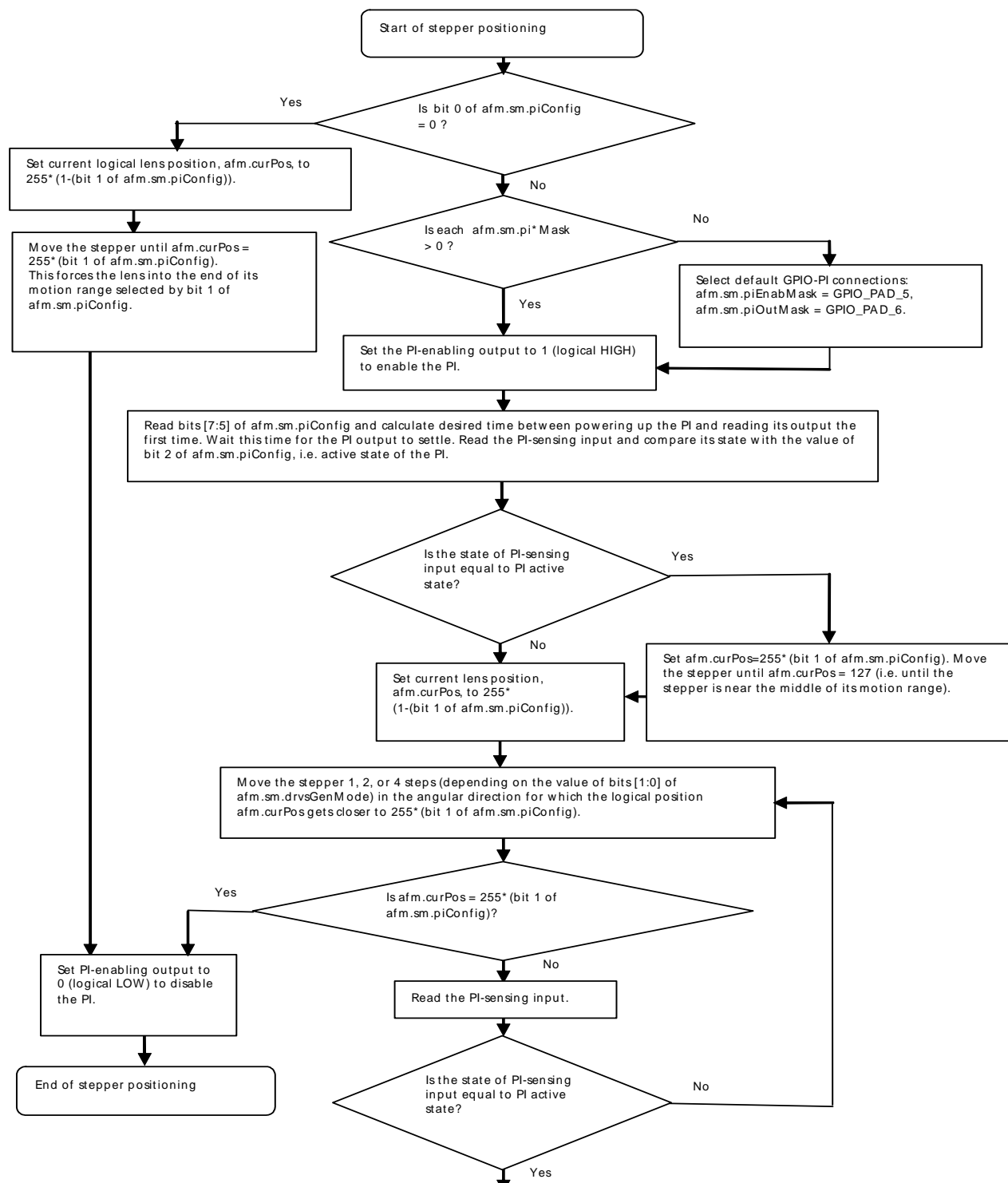
Figure 56: Lens Movements During Initial Positioning of a Stepper Motor (Example 3)


Note: Four lens motion sequences are possible when bits [3:0] of `afm.sm.piConfig` are set to 5 (binary '0101'). The only difference between the top two sequences and the sequences shown in Figure 54 is reversed direction of lens jumps from PI signal edge to logical position 0. The direction of this jump is determined by bit 3 of `afm.sm.piConfig` and its length by `afm.sm.piEdgeOffset`. If `afm.sm.piEdgeOffset` is greater than 0 and bit 3 equals 0, the AFM driver moves the lens back into the PI inactive range after detecting the PI signal edge. If bit 3 is 1, the lens ends up at a position where the PI is active, as shown in Figure 54.

To switch between the top two and bottom two sequences above, one must toggle bit 4 of `afm.sm.piConfig` and change the value of `afm.sm.piEdgeOffset`. The maximum value of the latter variable is 255, which means that at maximum motion precision (when bit 0 of `afm.sm.drvsGenMode` equals 1), the lens cannot be moved more than 255 stepper motor steps away from the PI signal edge. When the initial lens positioning is done with minimum precision (bits [1:0] of `afm.sm.drvsGenMode` equal 0), `afm.sm.piEdgeOffset` = 255 corresponds to 1020 stepper motor steps.

Table 24: Programmable Parameters of Stepper Motor Positioning Function

#	Variable Name	Bit(s)	Valid Values	Parameter Description
1	afm.sm.piConfig	0	0, 1	Bit selecting one of two possible ways to do initial positioning of a stepper-motor-powered AF lens: Setting this bit to 1 selects initial positioning with a photointerrupter. Clearing this bit selects positioning without a PI, by forcing the lens into a preselected end of its motion range (see afm.sm.piConfig bit 1 below).
2	afm.sm.piConfig	1	0, 1	Bit telling the AFM driver which limit of lens position range is closer to the position marked by PI output signal edge. The state of the PI output in the position range between this limit and the signal edge is referred to as PI active state and the range itself is called PI active range. When bit 1 of afm.sm.piConfig equals 0, the PI signal edge is closer to logical position 0 than 255. When the bit equals 1, the edge is closer to logical position 255. Since the relation between the logical and physical lens positions is flexible, the logical position 0 can correspond either to infinity focus position or near focus position. The same is true of logical position 255.
3	afm.sm.piConfig	2	0, 1	Bit identifying the PI active state. The AFM driver expects the active state to be low (0) or high (1) according to the value of this bit.
4	afm.sm.piConfig	3	0, 1	Bit indicating the direction from the PI signal edge to desired initial position of the lens. If this bit is 1, the desired initial position is in the PI active range; otherwise, it is on the other side of the PI signal edge, in the range where the PI is inactive.
5	afm.sm.piConfig	4	0, 1	Bit selecting the value to be given to afm.curPos once the lens is at the desired initial position. Bit values of 0 and 1 correspond to afm.curPos values of 0 and 255, respectively.
6	afm.sm.piConfig	[7:5]	0,...,7	Delay between powering up the PI and reading its output the first time is proportional to the value stored in these 3 bits.
7	afm.sm.piEnabMask	[15:0]	1,...,4096	Mask selecting a GPIO pad as an output enabling the PI (for example, afm.sm.piEnabMask = 4 selects GPIO2).
8	afm.sm.piOutMask	[15:0]	1,...,4096	Mask selecting a GPIO pad as an input for sensing the output of the PI.
9	afm.sm.piEdgeOffset	[7:0]	0,...,255	Distance between PI signal edge and the desired initial position of the lens given as a multiple of the smallest increment of lens position selected by bits [1:0] of afm.sm.drvsGenMode.

Figure 57: Flowchart of AFM Driver Function Used in Initial Positioning of Stepper Motors (Page 1)


Note: The flowchart of the AFM driver function `AFM_ResetStMotor` whose task is to do initial positioning of stepper-motor-powered lens actuators (with or without a photointerrupter).

Figure 58: Flowchart of AFM Driver Function Used in Initial Positioning of Stepper Motors (Page 2)

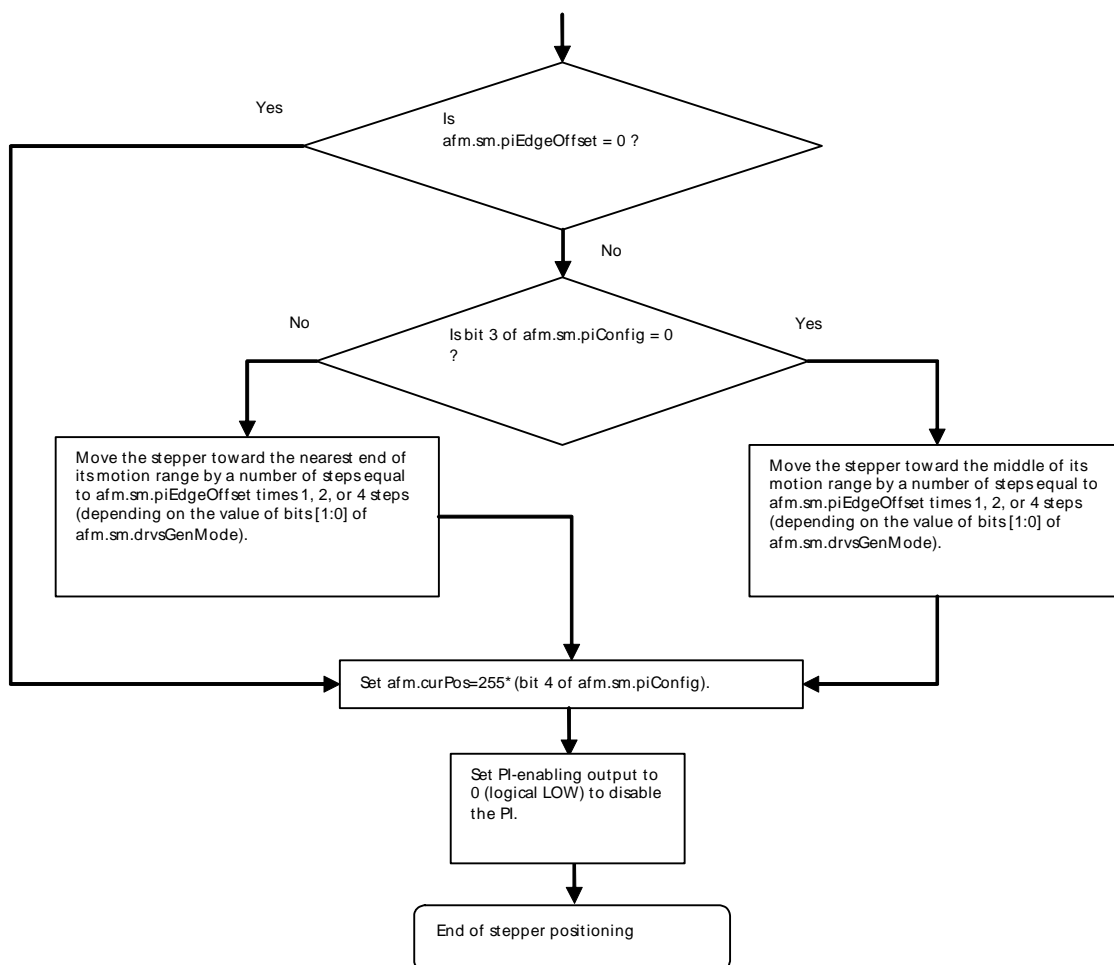


Table 25: Public Variables of the AFM Driver

Offset	Name	Type	Default	R/W	Description
0	vmt	void*	E9AE	RW	Pointer to the driver's virtual method table (VMT). The AFM driver includes a separate set of control methods for each supported type of auto focus mechanism (see afm.type below). Each set of methods is assessable via a separate VMT. Pointing afm.vmt to one of those VMTs either enables the AF driver to control the corresponding type of AF mechanism via the GPIO, or takes control of the GPIO away from the AF driver.
2	type	uchar	0	RW	Type of AF mechanism (lens actuator) used: 0—none, 1—helimorph, 2—stepper motor. At sequencer initialization, this variable is set to 0 and the afm.vmt is pointed to the default VMT of the AFM driver, which makes the GPIO inaccessible to the AF driver. Enabling the AF driver to control a lens actuator via the GPIO involves two steps. First, afm.type must be set to t+128, where t is 1 or 2. Second, the sequencer must be given REFRESH command by setting seq.cmd to 5. The nonzero seventh bit in afm.type forces the sequencer to call AFM_Init function upon that command. The function makes afm.type equal to t and points afm.vmt to the VMT through which the AFM driver methods for controlling actuator type t can be called.
3	curPos	uchar	0	RW	Current logical position.
4	prePos	uchar	0	RW	Previous logical position.
5	status	uchar	0	RW	Lens actuator status: Bit 0—0 if all is OK, 1 if the actuator reported an error Bit 1—0 if the lens is stationary, 1 if it is moving Bit 2—0 if direction of last lens movement was forward (+), 1 if the direction was backward (-) Bits [4:3]—number of current stepper motor position if afm.type=2; otherwise unused Bits [7:5]—unused After sending a “change lens position” command to the lens actuator, the AFM driver sets bit 1 of afm.status to 1. The value of the bit remains 1 until the AFM driver gets information that the lens is not moving (either has stopped at the desired new position or has failed to reach it). The AF driver waits through all times when the lens is moving by having afm.status updated once every frame, reading its bit 1 and immediately going back to sleep if it equals 1.
6	posMin	uchar	0	RW	Lower limit of physical position range.
7	posMax	uchar	0	RW	Upper limit of physical position range. Can be set below afm.posMin to swap forward (+) and backward (-) directions of lens motion.
8	posMacro	uchar	0	RW	Logical macro position.

Table 25: Public Variables of the AFM Driver (continued)

Offset	Name	Type	Default	R/W	Description
9	backlash	uchar	0	RW	<p>Logical size of backlash-compensating step that the AF driver can optionally use in lens positioning after the first scan. If bits [7:6] of <code>af.mode</code> are set to 0, positioning is done by moving the lens directly from the end position of the scan (<code>af.positions[af.initPos+af.numSteps-1]</code>) to the logical position found best (<code>af.positions[af.initPos+af.bestPosition]</code>). The direction of this move is opposite to the direction of the scan, and therefore the move may not bring the lens to its intended physical destination, unless its logical length is adjusted upward to compensate for lens actuator backlash.</p> <p>To make this adjustment, the AF driver subtracts <code>afm.backlash</code> from the value of <code>af.positions[af.initPos+af.bestPosition]</code> and gives the result to the AFM driver as the logical position to move the lens to. Negative results of the subtraction are replaced with 0. Note that subtracting <code>afm.backlash</code> makes sense only when <code>af.positions[af.initPos+af.numSteps-1] > af.positions[af.initPos+af.bestPosition]</code>; otherwise addition is required. Since the AF driver always subtracts <code>afm.backlash</code>, backlash compensation using this variable is not recommended after scans done in the negative direction (e.g. from logical position 255 to logical position 0).</p>

Table 25: Public Variables of the AFM Driver (continued)

Offset	Name	Type	Default	R/W	Description
10	custCtrl	char	0	RW	<p>Custom controls: 1-bit option switches and fine-tuning parameters for actuator control methods. The function of different bits of this variable depends on the current value of afm.type.</p> <p>If afm.type = 1, then:</p> <p>Bit 0—selects the length of commands sent to HD80 helimorph driver by function AFM_SetPosHelimorph (0–2 bytes, 1–3 bytes including enable byte)</p> <p>Bit 1—selects one of two possible relations between the argument of the function AFM_SetPosHelimorph, bPos, and position byte sent to HD80 helimorph driver (0 means send bPos, 1—send 255-bPos, to reverse the direction of lens movement)</p> <p>Bit 2—selects one of two positions that helimorph can assume upon command to exit standby (0—afm.posMin, 1—afm.posMax),</p> <p>Bits [7:3]—unused</p> <p>If afm.type = 2, then:</p> <p>Bit 0—selects direction of lens motion if bit 1 is set to 1,</p> <p>Bit 1—determines how function AFM_SetPosStMotor interprets its 1-byte argument (0—as desired logical lens position, 1—as number of physical steps to make in the direction indicated by bit 0),</p> <p>Bit 2—enables periodic forcing of stepper-motor-driving outputs into calculated logical states (0—forcing disabled, out puts are only toggled as needed, 1—forcing enabled),</p> <p>Bit 3—when set to 1, enables powering stepper motor down after every movement (the motor is always powered up before movements, but powering down is optional),</p> <p>Bit 4—enables repositioning of stepper motor by function AFM_ResetStMotor upon command to enter standby (1—enable, Bit 5—enables repositioning of stepper motor by function AFM_ResetStMotor upon command to exit standby (1—enable, 0—disable),</p> <p>Bits [7:6]—allow one to slow down initial portions of stepper-motor-driving waveforms that cannot be entirely generated by the MT9D111 waveform generator (higher value = slower waveforms). 0—disable),</p> <p>If afm.type = 0, afm.custCtrl is unused.</p>
11	timer.vmt	void*	E9C6	RW	<p>Pointer to timer VMT. Default timer VMT located in ROM contains pointers to the following public functions:</p> <p>AFM_Wait AFM_TimerSetDelay, AFM_TimerSetTimeToMove AFM_TimerIsStopped</p> <p>The pointers are all of type void* and have the following names: pWait, pSetDelay, pSetTimeToMove, pTimerIsStopped.</p>
13	timer.startTime	uint	0	RW	Timer start time.
15	timer.stopTime	uint	0	RW	Timer stop time.

Table 25: Public Variables of the AFM Driver (continued)

Offset	Name	Type	Default	R/W	Description
17	timer.hiWordMclkFreq	uint	0	RW	Master clock frequency in Hz divided by 65536. Used to convert delay times in milliseconds (for example, the values of <code>afm.timer.maxShortDelay</code> and <code>afm.timer.maxLongDelay</code>) to corresponding counts of master clock cycles that can be programmed into the timer.
19	timer.maxShortDelay	uint	0	RW	Maximum expected duration of a short lens move in milliseconds. Used by the AFM driver function <code>AFM_TimerSetTimeToMove</code> to compute lens travel time estimates.
21	timer.maxLongDelay	uint	0	RW	Maximum expected duration of a long lens move in milliseconds. Used by the AFM driver function <code>AFM_TimerSetTimeToMove</code> to compute lens travel time estimates.
23	timer.maxQuickMove	uchar	0	RW	Maximum length of short lens move (or threshold between short and long moves). Used by the AFM driver function <code>AFM_TimerSetTimeToMove</code> to compute lens travel time estimates.
24	timer.config	uchar	0	RW	<p>Bits [1:0] of this variable determine how <code>afm.timer.maxShortDelay</code>, <code>afm.timer.maxLongDelay</code>, and <code>afm.timer.maxQuickMove</code> are used to estimate duration of lens movements. Bits [7:2] are unused.</p> <p>If a command-driven lens actuator does not provide any feedback about its status after receiving a command to move an AF lens, the AFM driver must somehow predict how long the lens will be moving to prevent the AF driver from collecting sharpness scores and issuing new commands during its movement. The need for predictions of lens travel time is satisfied rather inexpensively by the AFM driver function <code>AFM_TimerSetTimeToMove</code>, which takes as arguments two logical lens positions and estimates the time required to move the lens between them. The function can use two different estimation methods, both of which rely on three user-set parameters, <code>afm.timer.maxShortDelay</code>, <code>afm.timer.maxLongDelay</code>, and <code>afm.timer.maxQuickMove</code>, as a sole source of information about how fast the lens actuator moves the lens. The default method of piecewise linear estimation is used when bit 0 of <code>afm.timer.config</code> is cleared.</p> <p>Setting this bit to 1 enables the alternative bipolar method. The bipolar method is very simple: if the distance between the two logical positions given to <code>AFM_TimerSetTimeToMove</code> as arguments exceeds <code>afm.timer.maxQuickMove</code>, then <code>afm.timer.maxLongDelay</code> is selected as the proper lens travel time estimate. Otherwise, unless the two logical positions are the same, the estimate equals <code>afm.timer.maxShortDelay</code>. If the two positions are the same, the estimate should be 0, and indeed is 0 if bit 1 of <code>afm.timer.config</code> is cleared. However, if this bit is set to 1 and the positions are the same, the function <code>AFM_TimerSetTimeToMove</code> outputs <code>afm.timer.maxShortDelay</code> instead of 0.</p> <p>See “Timer” on page 106 or a description of the piecewise linear estimation method.</p>

Table 25: Public Variables of the AFM Driver (continued)

Offset	Name	Type	Default	R/W	Description
25	si.vmt	void*	E9CE	RW	Pointer to serial interface VMT. Default serial interface VMT located in ROM contains pointers to the following public functions: AFM_SiSendCmd, AFM_SiSetActvFlag, AFM_SiSendByte, AFM_SiRecvByte. The pointers are all of type void* and have the following names: pSendCmd, pSetActvFlag, pSendByte, pRecvByte.
27	si.clkMask	uint	0	RW	Mask selecting one of the GPIO pads as the clock line of dedicated two-wire serial interface between the MT9D111 and a lens actuator (for example, helimorph).
29	si.dataMask	uint	0	RW	Mask selecting one of the GPIO pads as the data line of the dedicated two-wire serial interface to a lens actuator.
31	si.clkQtrPrd	uint	0	RW	Delay for slowing down serial interface transmissions. The period of serial interface clock is asymptotically proportional to si.clkQtrPrd.
33	si.needsAck	uchar	0	RW	Switch enabling detection of ACK bits from the lens actuator (0—detection disabled, 1—enabled).
34	si.slaveAddr	uchar	0	RW	Lens actuator address used in serial interface transmissions.
35	sm.enabMask	uint	0	RW	Mask selecting one of the GPIO pads as stepper-motor-enabling output.
37	sm.driv0Mask	uchar	0	RW	Mask selecting one of the GPIO pads as first stepper-motor-driving output.
38	sm.driv1Mask	uchar	0	RW	Mask selecting one of the GPIO pads as second stepper-motor-driving output.
39	sm.driv2Mask	uchar	0	RW	Mask selecting one of the GPIO pads as third stepper-motor-driving output.
40	sm.driv3Mask	uchar	0	RW	Mask selecting one of the GPIO pads as fourth stepper-motor-driving output.
41	sm.drvsQtrPrd	uchar	0	RW	Delay for lengthening the period of stepper motor driving waveforms. The number of master/GPIO clock cycles in this period asymptotically approaches 8 times the sm.drvsQtrPrd.
43	sm.drvsGenMode	uchar	0	RW	This variable tells the AFM driver how to program the GPIO to generate stepper motor driving waveforms. Bits [1:0]—size of smallest stepper motor move (0—4 steps, 1—1 step, 2—2 steps) Bit 2—if 0, use the waveform generator in 8-bit counter mode, if 1, use it in 16-bit counter mode Bit 3—if 0, use clock divider 1, if 1, use clock divider 2 Bits [7:4]—clock divider setting (used by the AFM driver only if it is higher than the setting the driver has automatically calculated).
44	sm.piEnabMask	uint	0	RW	Mask selecting one of GPIO pads as photointerrupter-enabling output.
46	sm.piOutMask	uint	0	RW	Mask selecting one of GPIO pads as photointerrupter-sensing input.
48	sm.piEdgeOffset	uchar	0	RW	Distance (in units of smallest stepper motor move) between the position of photointerrupter signal edge and desired initial position of stepper motor.

Table 25: Public Variables of the AFM Driver (continued)

Offset	Name	Type	Default	R/W	Description
49	sm.piConfig	uchar	0	RW	Photointerrupter (PI) configuration: Bit 0—if 0, do not use PI, if 1, use it for initial stepper positioning Bit 1—if 0, PI signal edge is near logical position 0, if 1—near 255, Bit 2—PI active state (0—low, 1—high) Bit 3—if 1, PI is in the active state at initial stepper position, if 0—it is not Bit 4—if 0, initial logical stepper position is 0, if 1—255 Bits [7:5]—delay between powering up the PI and sensing its output the first time.

Public Functions of the AFM Driver and Corresponding VMT Pointers

void AFM_Init (void)

Pointer: afm.vmt->pInit

Description/Use: Initializes the AFM driver

void AFM_SetPos*(unsigned char bPos)

Pointer: afm.vmt->pSetPos

void AFM_ExecCmd*(BYTE bCmd)

Pointer: afm.vmt->pExecCmd

unsigned char AFM_GetStatus*(void)

Pointer: afm.vmt->pGetStatus

Description: Determines the current status of the AFM driver and lens actuator, updates the variable afm.status accordingly and returns its value

Use: Called once, indirectly, by the AF driver function AF_Run_snapshot. AFM_ResetStMotor and one other AFM driver function use the code line while (AFM_GetStatusStMotor()&2); to put the MCU on hold during certain short movements of the stepper motor.

void AFM_Wait(unsigned int wDelay)

Pointer: afm.timer.vmt->pWait

Description: Timer library function. Contains only 1 line of code:

```
if (wDelay) do { asm nop; asm nop; } while (--wDelay);
```

Each repetition of the “do” loop below takes 25 master clock cycles. The code preceding and following the loop is executed in 26 master clock cycles. Direct JSR (jump to subroutine) with address > 255 takes 6 clock cycles. The total delay caused by this function is therefore (32+25*wDelay)*(master clock cycle).

Use: Used whenever there is a need to keep the MCU idle for a short while. Several AFM_Si* functions (see below) contain multiple calls to AFM_Wait(afm.si.clkQtrPrd) that make the rate of serial interface clock adjustable. The function AFM_ResetStMotor calls AFM_Wait(32<<(afm.sm.piConfig>>5)) to allow the photointerrupter output signal to

settle after power-up. AFM_Wait is also used to time initial portions of stepper-motor-driving waveforms that cannot be entirely generated by the MT9D111 waveform generator.

void AFM_TimerSetDelay(unsigned int wDelay)

Pointer: afm.timer.vmt->pSetDelay

Description: Timer library function. If wDelay > 0, activates the timer and programs it to “tick” until the number of milliseconds equal to wDelay passes by. This is done by setting afm.timer.startTime and afm.timer.stopTime to two different values dependent on the current value of special function register sys.ClockCnt.Word.Hi (the upper word of 32-bit master clock cycle counter).

If wDelay = 0, the function stops and de-activates the timer by setting afm.timer.stopTime and afm.timer.startTime to 0.

AFM_TimerSetDelay uses afm.timer.hiWordMclkFreq to convert non-zero wDelay (in milliseconds) to the corresponding number of master clock cycles. For further details, see “Timer” on page 106.

Use: Called once, directly, by the function AFM_TimerSetTimeToMove.

void AFM_TimerSetTimeToMove(unsigned int wPos)

Pointer: afm.timer.vmt->pSetTimeToMove

Description: Timer library function. Takes as arguments two logical lens positions, bCurPos and bNewPos, combined into a 16-bit word ($wPos = 256 * bNewPos + bCurPos$) and estimates the time required to move the lens between them. The function can use two different estimation methods, both of which rely on three user-set parameters, afm.timer.maxShortDelay, afm.timer.maxLongDelay, and afm.timer.maxQuickMove, as a sole source of information about how fast the lens actuator moves the lens. The default method of piecewise linear estimation is used when bit 0 of afm.timer.config is cleared. Setting this bit to 1 enables the alternative bipolar method.

The bipolar method is very simple: if the distance between the two logical positions given to AFM_TimerSetTimeToMove as arguments exceeds afm.timer.maxQuickMove, then afm.timer.maxLongDelay is selected as the proper lens travel time estimate. Otherwise, unless the two logical positions are the same, the estimate equals afm.timer.maxShortDelay. If the two positions are the same, the estimate should be 0, and indeed is 0 if bit 1 of afm.timer.config is cleared. However, if this bit is set to 1 and the positions are the same, the function AFM_TimerSetTimeToMove outputs afm.timer.maxShortDelay instead of 0.

See “Timer” on page 106 for a description of the piecewise linear estimation method.

Use: Called once, indirectly, by function AFM_SetPosHelimorph.

BYTE AFM_TimerIsStopped(void)

Pointer: afm.timer.vmt->pTimerIsStopped

Description: Timer library function. Answers whether the timer is presently “stopped” or “ticking” (in other words, if the delay set by AFM_TimerSetDelay is over or not). It also automatically de-activates and resets the timer if it is stopped. Deactivation/reset is done by setting afm.timer.startTime and afm.timer.stopTime to 0.

Use: Called once, directly, by the function AFM_GetStatusHelimorph.

void AFM_SiSendCmd(unsigned int wCmd)

Pointer: afm.si.vmt->pSendCmd

Description: Serial interface library function. Can send a 2- or 3-byte long command, the first byte of which is always equal to afm.si.slaveAddr. The second command byte equals the lower byte of the function's 16-bit argument, wCmd. The upper byte of wCmd is sent next if it differs from 0xFF. If it equals 0xFF, it is discarded. The entire command is bracketed by START and STOP bits generated by AFM_SiSetActvFlag.

Each command byte is sent using the function AFM_SiSendByte. If after sending some byte AFM_SiSendByte returns 0 (indicating the absence of a needed ACK bit), the transmission of the command is aborted and restarted from the first byte. Up to two restarts can occur if the problem of missing ACK bits persists. The function AFM_SiSendCmd uses afm.status variable to count the restarts and report the final result of the command transmission. A successful transmission is indicated by afm.status = 0. If afm.status > 0, all three attempts to send the command have failed due to the lack of acknowledgement from the intended command recipient.

Use: Used to give two-wire serial interface commands to helimorph driver ASIC. Called once, indirectly, by AFM_SetPosHelimorph and once, likewise indirectly, by AFM_ExecCmdHelimorph.

void AFM_SiSetActvFlag(unsigned char bOn)

Pointer: afm.si.vmt->pSetActvFlag

Description: Serial interface library function. Generates START and STOP bits that bracket every two-wire serial interface transmission. If its argument **bOn** equals 0, it sends a STOP bit. Any other argument value produces a START bit.

Use: Called 2 times, directly, by the function AFM_SiSendCmd.

BYTE AFM_SiSendByte(unsigned char bByte)

Pointer: afm.si.vmt->pSendByte

Description: Serial interface library function. Sends its 1-byte argument, bByte, to an external device via two GPIO pads selected to be two-wire serial interface clock and data lines. Returns 1 or 0, depending on whether device has responded with an ACK bit and whether such a response is needed. If afm.si.needsAck = 1, it means that ACK is needed. AFM_SiSendByte returns 1 upon receiving it. If it is missing, the function returns 0. If afm.si.needsAck = 0, AFM_SiSendByte always returns 1.

Use: Called directly three times in the function AFM_SiSendCmd.

void AFM_SiRecvByte(unsigned char *pDestByte)

Pointer: afm.si.vmt->pRecvByte

Description: Serial interface library function. Receives a byte of data from an external two-wire serial interface transmitter and stores it at the memory location pointed to by its argument, pDestByte. It acknowledges receiving the data by sending an ACK bit back to the transmitter.

Use: Not currently used in the MT9D111 firmware.

Currently Supported AFM Mechanics

Table 26 lists the lens actuator and associated driver two-wire serial interface currently supported.

Table 26: AFM Mechanics Supported

Driver IC	Interface Protocol	Lens Actuator	Patch Needed	Actuator Performance
One Limited HD80	Two-wire serial interace (8-bit)	Helimorph	No	Simple configuration, less power consumption, but poor precision, and bad shock resistance.
Analog Device AD5398	Two-wire serial interace (10-bit)	VCM	Yes*	
Analog Device AD5398	Two-wire serial interace (10-bit)	MEMS	Yes*	Excellent precision, good optical performance, but complicated configuration.
Sanyo LB1935T	Stepper	Stepper	No	Better optical performance and precision, but more power consumption and complicated configuration.
Sanyo LB1935CL	Stepper	Stepper	No	
Sanyo LV8071LP	Stepper Like	Piezo	Yes	Simple configuration, less power consumption, but poor precision, and bad shock resistance.
SiTi (Silicon Touch) MD115	PWM	VCM	Yes	Low cost, and simple configuration, but poor precision and loud noise.
Interpion ID9701	PWM	VCM	Yes	

Note: * The MT9D111, Rev 5, supports AD5398. Therefore, patch may not be necessary.

Mode Driver-Setting up Preview (A) and Capture (B) Modes

The mode driver (ID=7) serves two major functions:

1. Storing local copies of image sensor and pipeline registers for both preview and capture modes so that they may be uploaded at the appropriate time between frames (to avoid mid-frame changes)
2. Generating a gamma correction table that may be selected to include a predefined level of contrast enhancement, thus adding contrast control to this part

The mode driver contains shadow registers for context A (preview) and context B (capture). These shadow registers upload their values to the corresponding image sensor and pipeline control hardware registers at the proper time so as not to introduce mid-frame changes that would cause frame corruption. These shadow registers have been chosen to provide all foreseeable changes that a user would want to make between preview and captures modes (see data sheet for affected registers for each mode driver variable). There are additional preview-specific and capture-specific registers in the sensor core (R0x05:0, R0x06:0, R0x07:0, R0x08:0, R0x20:0, R0x21:0), which are not included in the mode driver's values.

Upon power-up the user should upload/change all non-default register values desired, including the mode driver values. The user does not need to upload to any hardware registers that correspond to the mode driver values because the registers are overwritten upon initialization, a context change (preview to capture or vice versa), or a sequencer REFRESH or REFRESH_MODE command.

The user may also change these mode driver variables at any time, but their changes are not reflected in the images until either: (a) the user issues a REFRESH or REFRESH_MODE command to the sequencer, (b) the user changes the sequencer state from preview to capture or vice versa, or (c) the user issues a STANDBY sequencer command and then returns. This allows the user to change values without affecting the immediate image processing, avoiding image corruption.

“Gamma and Contrast” on page 30 describes the mode driver's gamma and contrast functionality.

MT9D111 Register Wizard

The MT9D111 Register Wizard tool is a software program that allows a user to generate the proper settings in regards to timing, PLL, state parameters, and gamma/contrast parameters for the sensor.

After specifying the desired operating frequency, frame rate, resolution, and other parameters, the user can save the resulting register/variable values in an INI file format that can easily be loaded in Aptina's DevWare demo software.

The tool is currently in its pre-release form and has not yet been extensively tested. Additional features may be added in the future. Any bug reports should be reported to your local Aptina Field Applications Engineer (FAE).

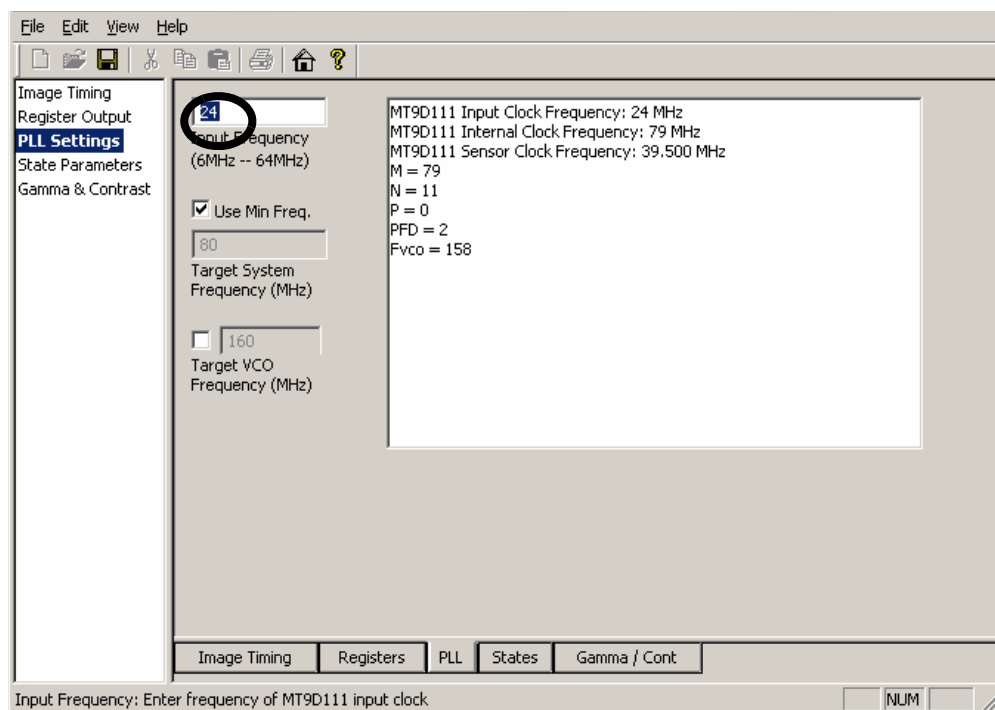
Procedure

After opening the MT9D111 Register Wizard tool, the user should first go to the PLL settings section to specify the input clock and PLL output frequencies (see Figure 59.)

In the first text box, enter the input clock frequency to the sensor (CLKIN). Next, enter the targeted output frequency of the PLL in the second text box. If the text box is disabled, uncheck the “Use Min Freq” checkbox. Alternatively, you can leave the “Use Min Freq” box checked and let the tool to select the minimum PLL output frequency needed based on the “Image Timing” section.

The window on the right will then show the required PLL settings—set by the M, N, and P values—to achieve the named configuration.

Figure 59: Input Clock and PLL Output Frequencies



The “Target VCO Frequency (MHz)” field allows the user to specify a target VCO frequency—the frequency of one of the stages of the PLL, which has a valid range of 110 to 240 MHz. The tool limits its calculations based on this range, as well as other requirements—see Table 27 for more details.

Table 27: PLL Specifications

Specification	Equation	Min	Max
M	—	16	—
f_{PFD}	$f_{IN}/(N+1)$	2 MHz	13 MHz
f_{VCO}	$f_{PFD} * M$	110 MHz	240 MHz
f_{OUT}	$f_{VCO}/(2*(P+1))$	6 MHz	80 MHz
f_{IN}	—	6 MHz	64 MHz

The tool assumes the use of PLL. However, if the sensor is operating from the CLKIN frequency directly (no PLL), follow the steps below:

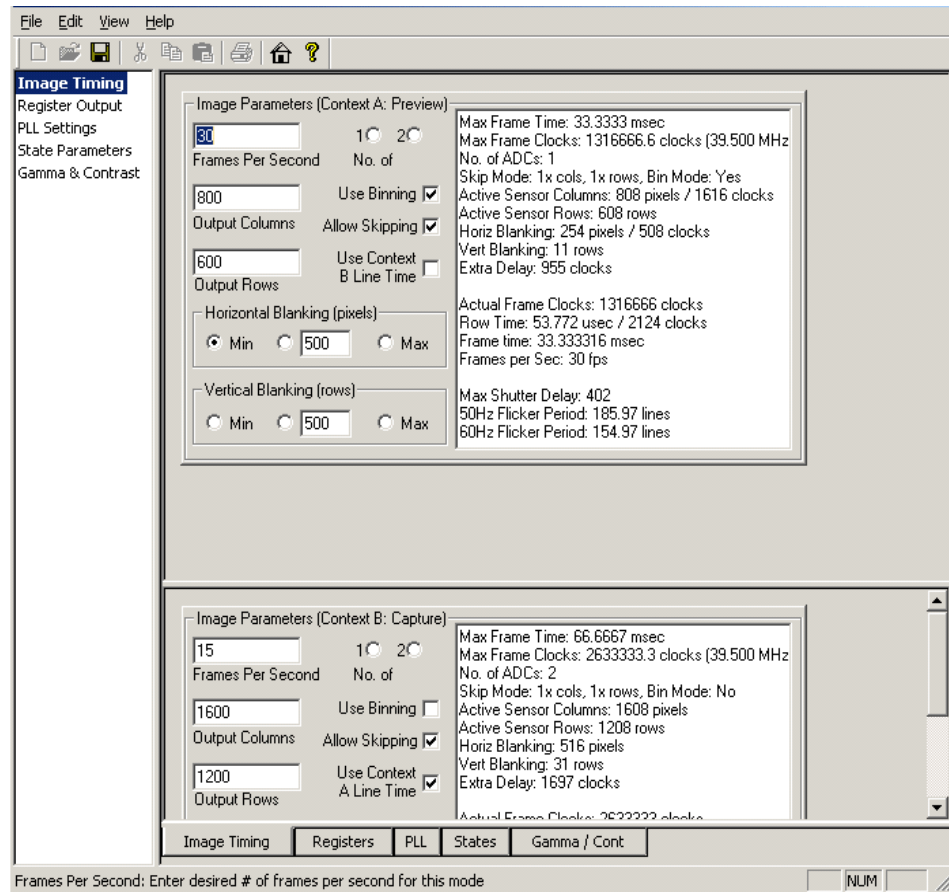
1. Enter the same frequency in the “Input Frequency” and “Target System Frequency” text boxes. The “Use Min Frequency” checkbox should be unchecked.

- After all the parameters are set from the other sections (Image Timing, State Parameters, and Gamma & Contrast), save the settings as an INI file.
- Use a text editor to remove the following lines from the INI file.

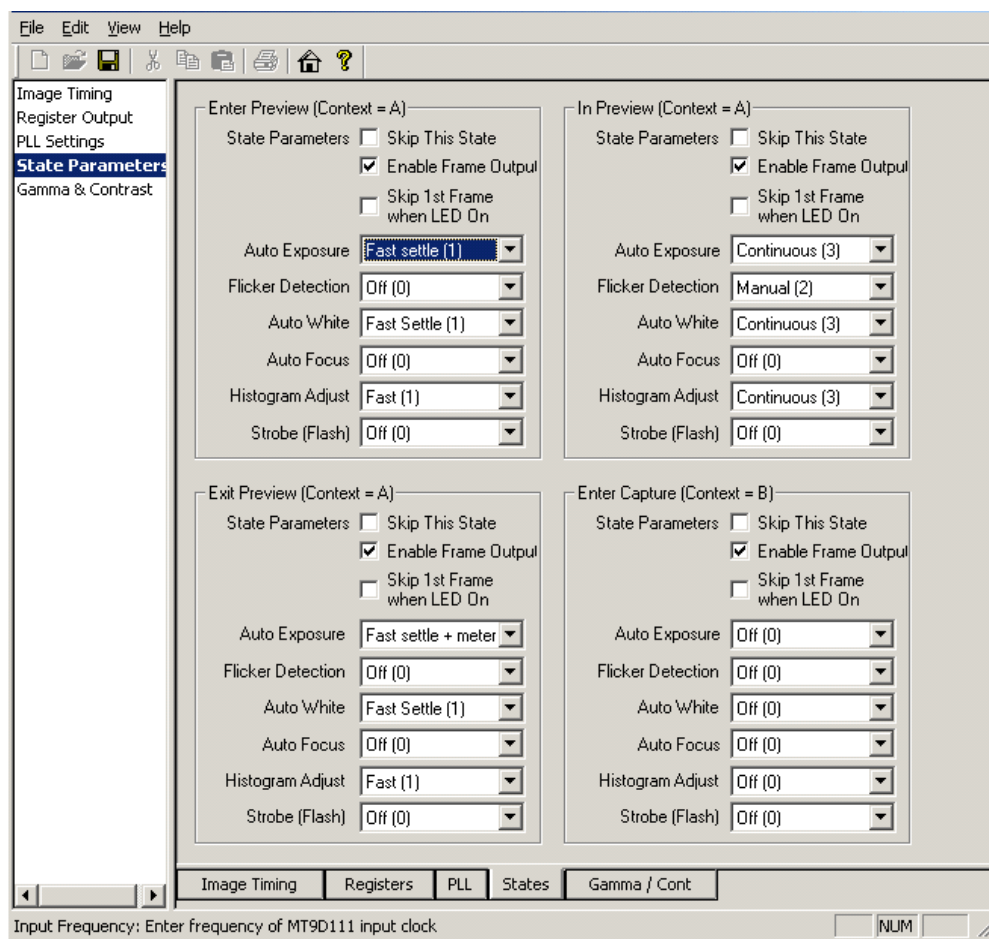
```

REG = 0, 0x66, 0x**** //PLL Control 1 = ****
REG = 0, 0x67, 0x**** //PLL Control 2 = ****
REG = 0, 0x65, 0xA000 //Clock CNTRL: PLL ON = 40960
REG = 0, 0x65, 0x2000 //Clock CNTRL: USE PLL = 8192
  
```

Figure 60: Image Timing Section



The Image Timing section allows the user to configure the frame rate, resolution, ADC mode, binning option, skipping option, and blanking option for each context. If the input values are beyond specifications, the corresponding text box will be highlighted in yellow. A warning message will also appear in the window on the right-hand side. For example, if a user enters a frame rate that is too high, the warning message will specify the maximum frame rate achievable based on the current operating frequency. By default, the “Use Context A Line Time” checkbox is selected. It is necessary to match the line (row) time for both contexts in order for the firmware drivers (such as auto exposure) to function optimally.

Figure 61: State Parameters Tab


In the State Parameters section, the user can configure the mode for each driver in the following four states: Enter Preview, Preview, Leave Preview, and Enter Capture. The configurable drivers are: Auto Exposure, Flicker Detection, Auto White Balance, Auto Focus, Histogram, and Flash. There is also an option to skip a particular state by using the associated checkbox.

Figure 62: State Diagram and Transitions of the MT9D111

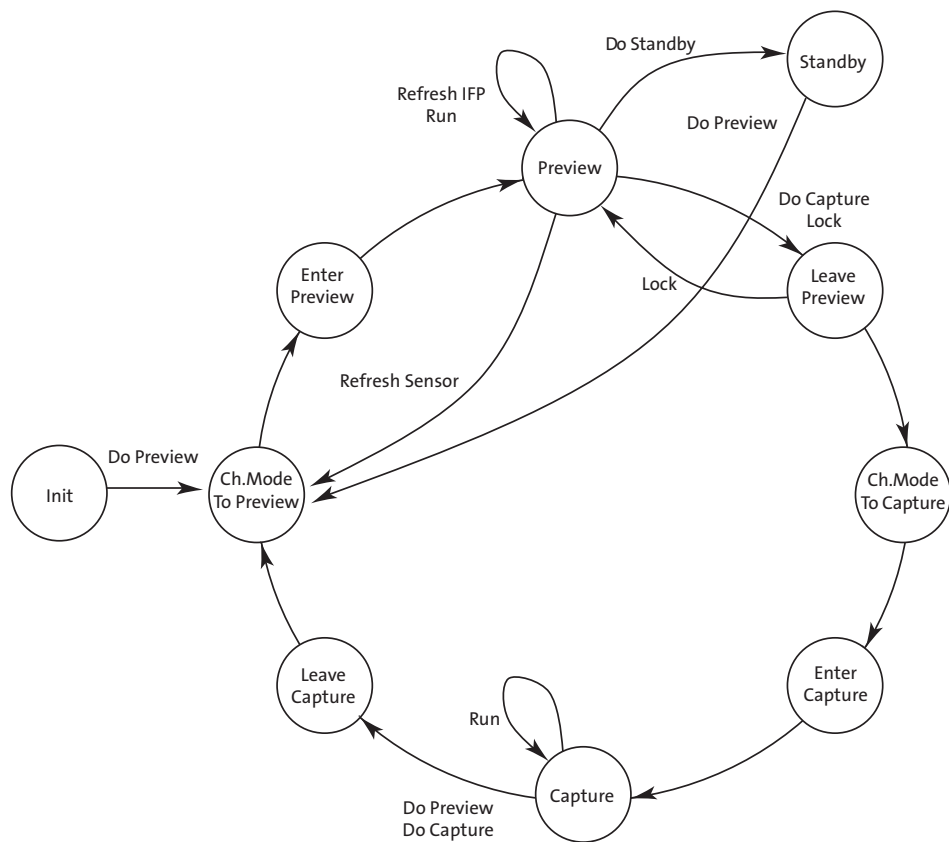
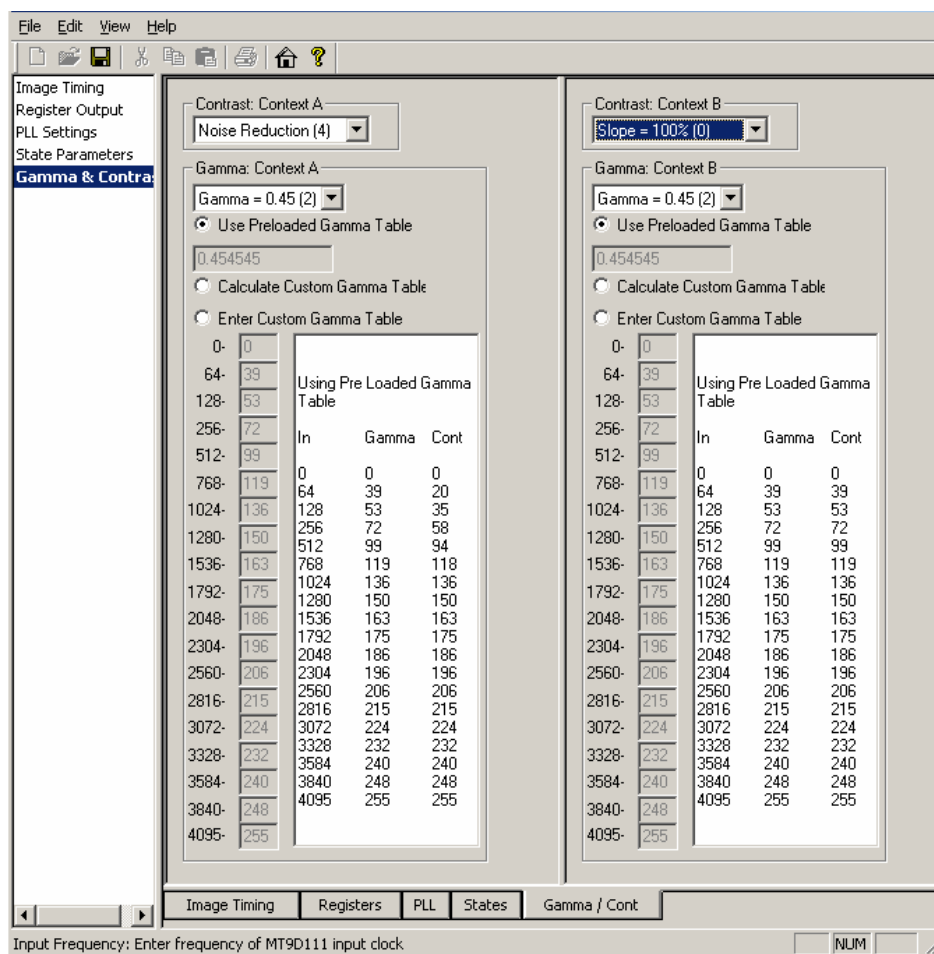
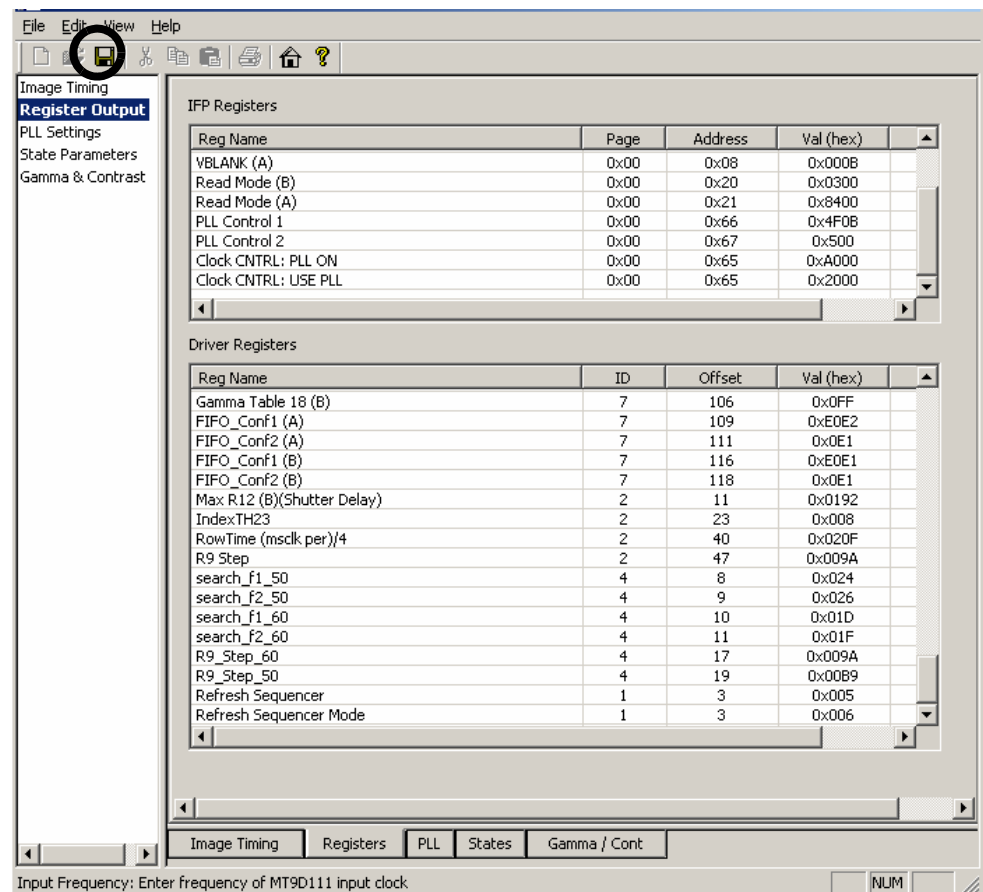


Figure 63: Gamma and Contrast Tab


In the final section—Gamma & Contrast—the user can select pre-defined gamma and contrast settings. The user also has the option to program the gamma/contrast table manually.

The available gamma options are: 1.0, 0.56, 0.45, or user-defined. Contrast options are: 100 percent (no contrast increase), 125 percent, 150 percent, 175 percent, and noise-reduction.

For additional gamma or contrast settings, the user can modify individual data points—see Figure 63. Data point values may range from 0 to 255.

Figure 64: Register Output Tab


Once all the parameters are selected, the user can review all the associated register/variable values in the Register Output section of the tool, as shown in Figure 64.

In order to save these settings, click the diskette icon in the tool bar. Once the INI file is generated, it may be loaded into DevWare or converted to a different system format.

Note: DevWare/demo2 may not operate in the frequency specified by the user in the PLL Settings section. In this case, supply an external oscillator to the demo2 system with the same frequency that was entered in the tool.

The Register Wizard tool currently outputs register/variable values even if they have default values, so not all settings in the INI file are necessary to program the sensor. If an option is not modified, the user can remove the associated default register/variable setting in the INI file.

Note: This tool is currently under development. The information in this manual and the features in the Register Wizard tool are subject to change without notice. Report any typos, errors, or bugs to your local FAE.

MT9D111 Developer Guide Mode Driver Preview and Driver FAQs

- What is the maximum frame rate that can be achieved if I use 65 MHz clock?

What is the maximum frame rate that can be achieved if I use 65 MHz clock? For 65 MHz, you can do 25 fps for QVGA (without skipping) and 12.8 fps for UXGA capture. (The register wizard will show this information.) Histogram Driver

How to Set Up the Histogram Driver Variable for Operation

The histogram driver works to reduce image flare by continually analyzing the input image histogram and dynamically adjusting the black level, R0x59:1. When flare is present, the image histogram does not contain dark tones, causing the driver to subtract a higher black level, thus regaining the lost contrast (at the expense of dynamic range). In certain situations, the scene may contain no dark tones without flare. The histogram driver cannot distinguish this condition and alters the black level just the same, causing the image to have more contrast, which looks acceptable in many situations.

The variable `hg.maxDLevel` sets the maximum level that can be subtracted from the input data (set this value to match the lens flare percentage). For example, if a lens typically has a five percent flare, set this value to $0.05 * 1024 = 51$. To disable flare subtraction in all modes, set this value to 0. The maximum allowed value is 128. Read variable `hg.DLevel` to see the current subtracted value. The variable `hg.percent` indicates the percentage of histogram dark tones which need to be clipped. The recommended value is 0. The `hg.DLevelBufferSpeed` controls the speed of adjustment, and has a range of 32 (fastest) to 1 (slowest).

The histogram driver operates with two sets of bins:

- The first set of bins is programmed to calculate low signal distribution in the image and is used to estimate black level, which should be subtracted to reduce image flare as described previously. Variables `hg.lowerLimit1` and `hg.binSize1` define the first set of bin. Variable `hg.lowerLimit1 = 0` sets offset for bin0 (divided by 4 on a 10-bit scale). Variable `hg.binSize1` sets bin width (0–4LSB, 1–8LSB, 2–16LSB, 7–512LSB on a 10-bit scale), so the first set of bins covers input signals 0–64.
- The second set of bins is programmed to calculate high signal distribution in the image and is used to estimate the percent of oversaturated pixels. Variables `hg.lowerLimit2=192` and `hg.binSize2=4` define the signal range from 768 to 1024, covered by the second histogram.

Variable `hg.scaleGFactor` sets the precision of histogram. If `hg.scaleGFactor=1`, then `reg216:2[15:18]` has a maximal value = 255, if all pixels have the same value and hit into bin1. This variable defines minimal bin resolution as 1/256 of the total number of pixels in histogram window. If `hg.scaleGFactor=2`, bin resolution is 1/512, and so on.

If the user wants to change one from the variables `hg.scaleGFactor`, `hg.lowerLimit1`, `hg.binSize1`, `hg.lowerLimit2`, or `hg.binSize2`, command `DO_REFRESH (seq.cmd=5)` has to be called for new values have effect.

The histogram driver uses the same window as the AE driver.

Flash Strobe, Mechanical Shutter, and Global Reset

Still Capture using Xenon/LED Flash with User-defined Image Quality Settings

The MT9D111 supports flash mode with user predefined settings for auto exposure and auto white balance for both Xenon and LED flashes. To capture images using predefined settings for Xenon or LED flash, the user (host):

1. Must program the sequencer to select the appropriate flash type
2. Turn the flash on
3. Select the “load user defined settings” modes for the CaptureEnter state
4. Disable all auto functionality for the PreviewLeave, CaptureEnter, and Capture states
5. Transfer the sequencer into step-by-step mode (`sequencer.stepMode[0] = 1`) and call the `DO_CAPTURE` command (`sequencer.cmd`). Wait until the sequencer comes to the PreviewLeave state and load the white balance and exposure settings to the corresponding AWB, AE, and histogram (HG) driver variables. Release the step-by-step sequencer mode (`sequencer.stepMode[0] = 0`) so that the sequencer automatically passes all states from PreviewLeave back to the Preview state.

While in the CaptureModeChange state, the sequencer changes the imaging frame size from preview to capture. In the CaptureEnter state, the settings loaded to the driver variables in the PreviewLeave state are automatically loaded into SOC registers and the flash is engaged.

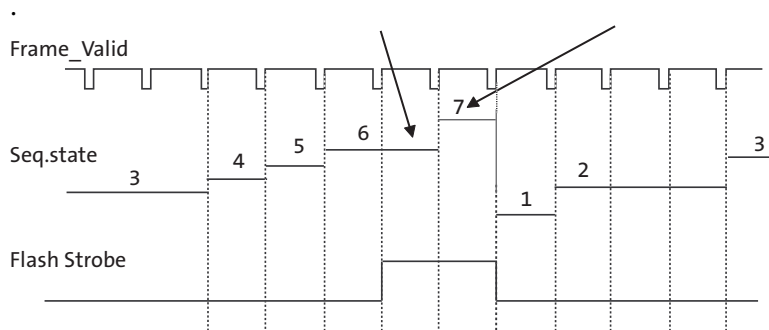
After this, the sequencer comes back to the Capture state. During N frames (see variable `sequencer.captureParams.numFrames`) when the sequencer stays in Capture state, the user must grab a frame. After N frames in the Capture state or the `DO_CAPTURE` command (whichever comes first), the sequencer turns off the flash pin (if flash was enabled for every frame), and goes back to the Preview state.

Command Sequence

- `seq.sharedParams.flashType = 1` (LED) or `2` (Xenon)
- `seq.capParEnter.flash = 129` —enable flash and loading of the user defined settings for “CaptureEnter” state.
- `seq.capParEnter.skipframe = 32` —skip one frame after LED flash is enabled (optional, for LED flash only)
- `seq.previewParLeave.ae=0`
- `seq.previewParLeave.fd=0`
- `seq.previewParLeave.awb=0`
- `seq.previewParLeave.hg=0`
- `seq.previewParLeave.flash=0` —disable all auto functionality for PreviewLeave states
- `seq.capParEnter.ae=0`
- `seq.capParEnter.fd=0`
- `seq.capParEnter.awb=0`
- `seq.capParEnter.hg=0` —disable all auto functionality for “CaptureEnter” states
- `seq.captureParams.mode = 0` —disable all auto functionality for Capture states
- `seq.captureParams.numFrames = 1` —specify how many frames sequencer should stay in Capture state
- `seq.stepMode = 3` —set step-by-step sequencer mode and let to do next step
- `seq.cmd = 2` —call `DO_CAPTURE` command
- wait until `seq.mode = 4` —wait until sequencer comes to PreviewLeave state.
- for (`i=0; i<11; i++`)

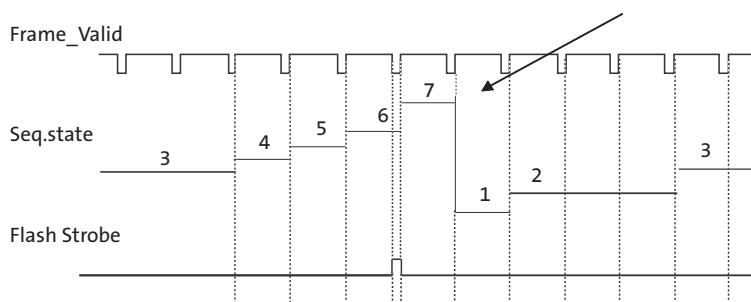
- awb.ccm[i] = ud_ccm[i]; —load user defined (ud) color correction matrix
- awb.GainR = ud_GainR;
- awb.GainG = ud_GainG;
- awb.GainB = ud_GainB; —load ud digital WB gains
- ae.VirtGain = ud_VirtGain; —load ud virtual analog gain
- ae.DGainAE1 = DGainAE1; —AE digital gain1
- ae.DGainAE2 = DGainAE2; —AE digital gain2
- ae.R9 = ud_IT; —integration time
- ae.R65 = ud_R65; —ADC reference
- hg.DLevel = ud_DLevel; —dark level
- seq.stepMode = 2 —release step-by-step sequencer mode
- wait until seq.mode == 7 and capture frame

Figure 65: LED Flash Timing Diagram



Note: Parameters: integration time = one frame. Skip one frame after LED is ON. One frame in Capture state. Same frame size for Preview and Capture modes.

Figure 66: Xenon Flash Timing Diagram



Note: Parameters: integration time = one frame. One frame in Capture state. Same frame size for Preview and Capture modes.

Still Capture using LED Flash with Automatic White Balance and Exposure Control

The MT9D111 supports LED flash mode with automatic white balance and exposure control. The sequencer can be programmed to make fast AE and AWB calculations for a scene illuminated by an LED flash in the PreviewLeave and CaptureEnter states. (Calculating AE and AWB in PreviewLeave state rather than CaptureEnter state is recommended because the frame rate is usually faster in preview mode.)

Use the following steps to capture an image using AWB and AE:

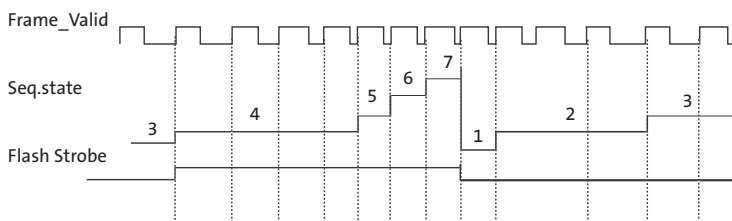
1. Program the sequencer to select the LED flash type
2. Turn on the flash in the PreviewLeave and CaptureEnter states
3. Enable the desired automatic functions in the PreviewLeave state, and disable them in the CaptureEnter and Capture states
4. Specify how many frames the sequencer should stay in the Capture state. Call the DO_CAPTURE sequencer command, wait until sequencer comes to the Capture state, and grab the appropriate frame.

After receiving the DO_CAPTURE command, the sequencer changes to the PreviewLeave state (step 4), enables the LED flash, and calculates the AE and AWB values. These calculations take one to seven frames.

Command sequence

- seq.sharedParams.flashType = 1 (LED)
- seq.previewParLeave.ae = 1
- seq.previewParLeave.fd = 0
- seq.previewParLeave.awb = 1
- seq.previewParLeave.hg = 1 —enable all auto functionality for PreviewLeave states
- seq.previewParLeave.flash = 1 —enable flash for PreviewLeave state.
- seq.previewParLeave.skipframe = 32 —skip one frame after LED flash is enabled (optional, for LED flash only)
- seq.capParEnter.ae = 0
- seq.capParEnter.fd = 0
- seq.capParEnter.awb = 0
- seq.capParEnter.hg = 0 —disable all auto functionality for CaptureEnter states
- seq.capParEnter.flash = 1 —enable flash for CaptureEnter state.
- seq.captureParams.mode = 0 —disable all auto functionality for Capture states
- seq.captureParams.numFrames = 1 —specify how many frames sequencer should stay in Capture state
- seq.cmd = 2 —call DO_CAPTURE command
- wait until seq.mode == 7 and capture frame

Figure 67: LED Flash Timing Diagram with Automatic Exposure and White Balance



Note: Same frame size for Preview and Capture modes.

Flash Strobe, Mechanical Shutter, and Global Reset FAQs

- For flash LED applications, what is the proper sequence for the flash light to be turned on before a frame is captured?
- We are considering using the CMOS sensor to detect the ambient light for the purpose of auto-flash application. Can you tell me which register contains the luminance value from the sensor? How do we convert the luminance to lux and what's the accuracy?

For flash LED applications, what is the proper sequence for the flash light to be turned on before a frame is captured?

Assuming that the customer is going from preview to capture mode for the flash capture, the proper sequence should be:

1. In Preview (seq.state=3)
2. Turn on flash
3. Send command to switch to context B
4. Turn off flash when seq.state=8 or 1, 2, 3

We are considering using the CMOS sensor to detect the ambient light for the purpose of auto-flash application. Can you tell me which register contains the luminance value from the sensor? How do we convert the luminance to lux and what's the accuracy?

There is a firmware variable (ae.mmMeanEV, ID=2 Offset=78, 8-bit variable) that provides an "EV" score. It is the mean of five EV zones. There is no straight conversion from this score to lux. Determining when the host should turn on the flash (that is, which EV values require flash) calls for some experimentation and optimization on the customer's side.

GPIOs

Programming GPIO Outputs

To program the GPIO, set the desired GPIOs to outputs and then assign the values for them to output. The following code shows how to drive a logic 1 out of GPIO[0]. In this example, the GPIO registers are programmed via the two-wire serial interface.

```
// Example: set GPIO[0]=1

R0xC6:1=0x9079// GPIO_DIR_L at 0x1079
R0xC8:1=0x00FE// Configure GPIO[0] is output

R0xC6:1=0x9071// GPIO_DATA_L at 0x1071
R0xC8:1=0x0001// Set GPIO[0]=1
```

Reading GPIO Inputs

To read the values of the GPIO (outputs and inputs), read from GPIO registers GPIO_DATA_H/L at 0x1070/1. The status of the output for the GPIOs are defined as outputs, and the input values for the GPIOs defined as inputs are read.

Outputting Flash and/or Strobe from GPIO

In 10-bit sensor bypass mode (register 9:1[1:0] = 00), strobe and flash are output from GPIO[10] and GPIO[11] pins, respectively. Also, flash is output from GPIO[11] if register 9:1[4:3] = 01. Strobe is output on GPIO[10] if register 81:1[0] = 1.

Waveform Generator Programming Example

This example uses physical access to variables to program the waveform generator. The waveform generator can also be programmed directly from the microcontroller.

```
// Example: set GPIO[0] to be a waveform with five parts per period

640 clocks high 1280 clocks low 1280 clocks high 1280 clocks low 640 clocks high

        ┌─── 640 ──┐ ┌─── 1280 ──┐ ┌─── 1280 ──┐ ┌─── 1280 ──┐ ┌─── 640 ──┐
        └──────────┘ └──────────┘ └──────────┘ └──────────┘ └──────────┘

R0xC6:1=0x907F// GPIO_DIR_OUT_L at 0x107F
R0xC8:1=0x0001// Configure GPIO[0] as output

R0xC6:1=0x9071// GPIO_DATA_L at 0x1071
R0xC8:1=0x0001// Set GPIO[0]=1 (set initial value of waveform)

//by default, clock divider is 21=2
R0xC6:1=0x90B2// GPIO_WG_CLKDIV at 0x10B2
R0xC8:1=0x0002// Set clock divider to 2^3=8

R0xC6:1=0x9081// GPIO_WG_T00
R0xC8:1=0x0050// set count of first waveform subperiod to be 0x50 (divided) clocks

R0xC6:1=0x9083// GPIO_WG_T10
R0xC8:1=0x00A0// set count of second waveform subperiod

//having more than two waveform subperiods is optional
R0xC6:1=0x9085// GPIO_WG_T20
R0xC8:1=0x00A0// set count of third waveform subperiod

R0xC6:1=0x9087// GPIO_WG_T30
R0xC8:1=0x00A0// set count of fourth waveform subperiod
```

```
R0xC6:1=0x9089// GPIO_WG_T40
R0xC8:1=0x0050// set count of fifth waveform subperiod

//If this register is set to 0, waveform will run forever
R0xC6:1=0x908B// GPIO_WG_N0
R0xC8:1=0x0005// waveform will repeat for five periods

R0xC6:1=0x90B6// GPIO_WG_SUSPEND
R0xC8:1=0x0001// suspend the waveform for GPIO[0]

R0xC6:1=0x90B5// GPIO_WG_RESET
R0xC8:1=0x0001// reset the waveform state machine for GPIO[0]

R0xC6:1=0x90B0// GPIO_WG_CONFIG
R0xC8:1=0x0001// enable the waveform for GPIO[0]
// This enables the waveform in manual mode. Instead of a manual trigger, the waveform gen-
erator can be set up to trigger on each falling edge of frame valid by setting the proper
bits in register 0x10B4 (GPIO_WG_FRAME_SYNC), each rising edge of strobe by setting the
proper bits in register 0x10BD (GPIO_WG_STROBE_SYNC), or at the end of the higher GPIO wave-
form by setting the proper bits of 0x10B1 (GPIO_WG_CHAIN)

R0xC6:1=0x90B5// GPIO_WG_RESET
R0xC8:1=0x0000// deassert reset

R0xC6:1=0x90B6// GPIO_WG_SUSPEND
R0xC8:1=0x0000// deassert suspend

R0xC6:1=0x908B// GPIO_WG_N0
read R0xC8:1// read back current value of the period counter
```

GPIO FAQs

- How do you set the GPIO to output pin of the following registers:
GPIO_DIR_IN_H(ADDR 0x107C), GPIO_DIR_IN_L(ADDR 0x107D),
GPIO_DIR_OUT_H(ADDR 0x107E), GPIO_DIR_OUT_L(ADDR 0x107F)?
- If GPIO ports are not used: 1. Is the power supply to VDDGPIO needed or not? 2. If power supply to VDDGPIO is not needed, should the VDDGPIO pin (pad) be connected to GND or not? 3. Should the GPIO pin (pad) be pull-up / pull-down / open?

**How do you set the GPIO to output pin of the following registers:
GPIO DIR IN H(ADDR 0x107C), GPIO DIR IN L(ADDR 0x107D),
GPIO DIR OUT H(ADDR 0x107E), GPIO DIR OUT L(ADDR 0x107F)?**

To enable GPIO pads as input or output, use:

GPIO_DIR_H (address 0x1078):

- Controls GPIO[11:8] pads direction. 1=input, 0=output.
- Upon power-on reset, all GPIO are inputs.

GPIO_DIR_L (address 0x1079):

- Controls GPIO[7:0] pads direction. 1=input, 0=output.
- Upon power-on reset, all GPIO are inputs.

For example, to set GPIO[0] as an output, set:

- R0xC6:1=0x9079// GPIO_DIR_L at 0x1079
- R0xC8:1=0x00FE// Configure GPIO[0] as output

If GPIO ports are not used: 1. Is the power supply to VDDGPIO needed or not? 2. If power supply to VDDGPIO is not needed, should the VDDGPIO pin (pad) be connected to GND or not? 3. Should the GPIO pin (pad) be pull-up / pull-down / open?

We recommend leaving VDDGPIO connected to power to avoid ESD problems. If the customer does not use any GPIO pins, doing so will not consume much power. Therefore, do not connect VDDGPIO to ground.

As for the GPIO pins, they are set as inputs by default. The customer can either configure them all as outputs (leave floating) or just tie them all to ground (as inputs).

Using the Test Patterns

The MT9D111 allows predefined test images to be loaded into the beginning of the image processor, replacing the live pixel readout of the imager. This provides a static image for testing the various image processing algorithms and function blocks independent of the scene data.

The test patterns are enabled by working with registers 72:1 through 75:1. Manipulating 72:1[0:2] allows for the selection of the following test patterns:

001—flat field; RGB values are specified in R73–75:1

010—vertical monochrome ramp

011—vertical color bars

100—vertical monochrome bars; set bar intensity in R73:1 and R74:1

101—pseudo-random test pattern

110—horizontal monochrome bars; set bar intensity in R73:1 and R74:1

111—white background

Disabling All Firmware Drivers

Although the firmware drivers may be disabled in the sequencer, their values may still automatically upload to the image processing hardware registers at the end of each frame, thus overwriting attempts to manually control the image processing hardware or the sensor's timing controls (shutter time, for example).

To disable all firmware drivers from changing the hardware registers, the microcontroller must be commanded to stop executing any firmware:

- Set R195:1[3] to 1 (puts microcontroller in “safe mode”)
- Set R195:1[1] to 1 (resets current execution of firmware)

Use the following procedure to resume firmware execution (reboot):

- Set R195:1[3] to 0 (puts microcontroller in “normal mode”)
- Set R195:1[1] to 0 (reboots execution of firmware)

JPEG Functionality

How to Enable/Disable the JPEG Output

JPEG output is only available in capture mode (Context B). To enable JPEG output, set `mode.mode_config[5] = 0`. To disable JPEG output, set `mode.mode_config[5] = 1`. Enabling or disabling JPEG takes effect only when the MT9D111 is switched to capture mode.

Due to the instantaneous change nature of JPEG data rate, we recommend enabling the variable output clock rate when capturing JPEG compressed data by setting `mode.fifo_conf0_B[6] = 1`. This makes the output clock run at a lower frequency when output FIFO occupancy is low and at a higher frequency when output FIFO occupancy is high, avoiding FIFO overflow.

However, when capturing uncompressed data, we recommend disabling the variable output clock rate (`mode.fifo_conf0_B[6] = 0`) and setting PCLK divisor N1 (`mode.fifo_conf1_B[3:0]`) to less than or equal to the image width decimating ratio, due to the constant data rate. For example, if output image width is decimated from 1600 to 800, set `mode.fifo_conf1_B[3:0] = 2`.

How to Set the JPEG Color Format

The MT9D111 supports YCbCr 4:2:2, YCbCr 4:2:0, and monochrome for JPEG. The color format is specified by `jpeg.format` (0 = YCbCr 4:2:2, 1 = YCbCr 4:2:0, 2 = monochrome).

The minimum image resolution for all color formats is 8 x 8. The maximum image width for YCbCr 4:2:0 is 384. There is no other limitation on image resolution.

How to Set the Restart Marker Interval

The MT9D111 can insert restart markers into the JPEG data stream. The restart marker interval is specified by `jpeg.restartInt`. Setting this variable to 0 disables restart marker insertion.

How to Get the JPEG Status

There are two ways to get JPEG status:

1. In spoof mode, the JPEG status byte is always appended to the end of the JPEG data stream. In continuous mode, the appending of JPEG status byte can be optionally enabled by setting `mode.fifo_conf0_B[9] = 1`.
2. Read `R2:2[7:0]`

Care should be taken when reading `R2:2`. The JPEG status in `R2:2` is cleared by the JPEG driver shortly after JPEG data transfer from output FIFO is completed if the next frame is going to be JPEG encoded. However, the user can set `jpeg.config[1] = 1` to keep the JPEG driver from clearing the status of an unsuccessful JPEG frame until `jpeg.config[3]` is set by the user.

Bit 7:6 of status indicates which set of quantization tables is used for the current frame JPEG encoding. Bit 7:6 = 0 means first set, 1 means second set, and 2 means third set.

How to Get the JPEG Data Length

There are two ways to get JPEG data length:

1. In spoof mode, the three JPEG data length bytes (in the order of LSB to MSB) are output right before JPEG status byte at the end of the JPEG data stream.
2. Read `jpeg.dataLengthMSB` and `jpeg.dataLengthLSBs`; they are bit 23:16 and bit 15:0 of the JPEG data length, respectively.

How to Handle the JPEG Errors

If any of the JPEG error flags (bit 3:1 of JPEG status) are set, the received JPEG frame should be discarded. In single frame still JPEG capture, the MT9D111 can be configured to encode the next frame when an error occurs by setting `jpeg.config[2] = 1`. In multiple frame still JPEG capture or video JPEG capture, it continually encodes subsequent frames until the specified number of still JPEG frames are successfully captured or video JPEG capture is terminated by a user command.

If `jpeg.config[5] = 1`, the JPEG driver automatically selects another set of quantization tables for next frame JPEG encoding when an error occurs by rolling through the three sets of quantization tables in the order of first, second, third, first, and so on. The first set of quantization tables is always used for the first frame of each JPEG capture command.

If `jpeg.config[5] = 0`, the user is responsible to specify the quantization table by setting `jpeg.config[7:6]` for every frame.

If desired, the user can enable handshaking at every erroneous JPEG frame by setting `jpeg.config[1] = 1`; which halts JPEG encoding after an erroneous JPEG frame until `jpeg.config[3]` is set to 1. This provides time for the user to handle the erroneous JPEG frame and react to the error condition, such as setting new quantization table scaling factor, loading new customized quantization tables, changing spoof frame size, etc. This could also cause frame loss if the user does not set `jpeg.config[3]` quickly enough, or terminate JPEG capture altogether if `jpeg.config[3]` is not set within the number of frames specified in `jpeg.timeoutFrames`.

How to Read/Write the JPEG Quantization/Huffman Table Memories

JPEG quantization and Huffman table memories can be accessed through the JPEG Indirect Access Control Register R30:2 and the JPEG Indirect Access Data Register R31:2.

Example: read quantization table 0 memory starting from address 0x80

- Set R30:2 = 0x8080 (bit 15 = 1 indicates address auto-increment)
- Read R31:2 repeatedly returns contents of memory address 0x80, 0x81, 0x82, ...

Example: write quantization table 0 memory starting from address 0x80

- Set R30:2 = 0xC080
- Write R31:2 repeatedly writes to memory address 0x80, 0x81, 0x82, ...

How to Program the Quantization Table

The MT9D111 can store up to three sets of quantization tables; each set consists of one table for luma and one for chroma. There are two ways to program quantization tables: scaled version of standard quantization tables in JPEG specification and customized quantization tables.

Scaled Standard Quantization Table

To have the JPEG driver program scaled version of standard quantization tables, write the desired scaling factor to bit 6:0 of the JPEG driver variables jpeg.qscale1, jpeg.qscale2, jpeg.qscale3, and 1 to bit 7 of the qscale variable. Quantization table scaling factor can be changed any time, even during JPEG capture (the new value takes effect after the current frame JPEG encoding is finished).

Example: set scaling factor of the first set of quantization tables to 8

- set jpeg.config[4] = 1
- set jpeg.qscale1 = 0x88

The calculation of the scaled version of standard quantization tables is as follows:

- $\text{scaled_Q} = (\text{scaling_factor} * \text{sandard_Q} + 16) \gg 5$

The value of scaled_Q is clipped to [1, 255]. The standard quantization tables are tables K.1 and K.2 of the ISO/IEC 10918-1 Specification (duplicated in Table 28 and Table 29 for convenience).

Table 28: Luminance Quantization

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 29: Chrominance Quantization

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Customized Quantization Table

Customized quantization tables can be loaded into quantization memory through JPEG Indirect Access Control Register R30:2 and JPEG Indirect Access Data Register R31:2. The address of quantization memory is mapped to indirect registers 0x080 through 0x1FF.

Table 30: Quantization Address Map

Indirect Register Address	Quantization Table
0x080 – 0x0BF	1st set luminance quantization table (table 0)
0x0C0 – 0x0FF	1st set chrominance quantization table (table 1)
0x100 – 0x13F	2nd set luminance quantization table (table 2)
0x140 – 0x17F	2nd set chrominance quantization table (table 3)
0x180 – 0x1BF	3rd set luminance quantization table (table 4)
0x1C0 – 0x1FF	3rd set chrominance quantization table (table 5)

The 14-bit floating-point reciprocal of the 8-bit quantization table value should be loaded into quantization memory in zigzag order. The reciprocal value can be found from the following lookup table.

```
const unsigned int Quant_Reciprocal_Lookup[256] =
{
    0x0000, 0x0001, 0x0400, 0x02ab, 0x0200, 0x0b33, 0x0155, 0x0a49,
    0x0100, 0x09c7, 0x00cd, 0x12e9, 0x0955, 0x093b, 0x1249, 0x0911,
    0x0080, 0x08f1, 0x11c7, 0x11af, 0x08cd, 0x08c3, 0x08ba, 0x08b2,
    0x1155, 0x251f, 0x113b, 0x1a5f, 0x1a49, 0x1a35, 0x1111, 0x2421,
    0x0880, 0x23e1, 0x10f1, 0x10ea, 0x19c7, 0x19bb, 0x19af, 0x10d2,
    0x10cd, 0x231f, 0x10c3, 0x197d, 0x10ba, 0x10b6, 0x10b2, 0x22b9,
    0x1955, 0x229d, 0x10a4, 0x2d05, 0x193b, 0x1935, 0x225f, 0x1095,
    0x2249, 0x223f, 0x2235, 0x2c57, 0x1911, 0x2219, 0x1084, 0x1082,
    0x1080, 0x18fc, 0x18f8, 0x21e9, 0x18f1, 0x21db, 0x18ea, 0x2b9b,
    0x21c7, 0x21c1, 0x21bb, 0x21b5, 0x21af, 0x2b53, 0x18d2, 0x367b,
    0x18cd, 0x2b29, 0x18c8, 0x218b, 0x18c3, 0x2b03, 0x217d, 0x2af1,
    0x18ba, 0x18b8, 0x18b6, 0x18b4, 0x18b2, 0x2ac1, 0x2ab9, 0x2159,
    0x2155, 0x3547, 0x2a9d, 0x214b, 0x18a4, 0x2a89, 0x2141, 0x189f,
    0x213b, 0x189c, 0x2135, 0x34c9, 0x2a5f, 0x2a59, 0x1895, 0x349d,
    0x2a49, 0x1891, 0x2a3f, 0x211d, 0x2a35, 0x188c, 0x2a2b, 0x344d,
    0x2111, 0x343b, 0x2a19, 0x2a15, 0x1884, 0x3419, 0x1882, 0x1881,
    0x1880, 0x20fe, 0x20fc, 0x33e9, 0x20f8, 0x3fb3, 0x29e9, 0x33cb,
    0x20f1, 0x33bd, 0x29db, 0x33af, 0x20ea, 0x29d1, 0x20e7, 0x3395,
    0x29c7, 0x20e2, 0x29c1, 0x20df, 0x29bb, 0x20dc, 0x29b5, 0x20d9,
    0x29af, 0x3359, 0x3353, 0x29a7, 0x20d2, 0x3343, 0x299f, 0x20ce,
    0x20cd, 0x2997, 0x3329, 0x20c9, 0x20c8, 0x298d, 0x298b, 0x3311,
    0x20c3, 0x3e0f, 0x3303, 0x3dfd, 0x297d, 0x297b, 0x2979, 0x32ed,
    0x20ba, 0x3dc9, 0x20b8, 0x20b7, 0x20b6, 0x20b5, 0x20b4, 0x20b3,
    0x20b2, 0x3d89, 0x20b0, 0x32bd, 0x295d, 0x3d6b, 0x2959, 0x2957,
}
```

```

0x2955, 0x32a7, 0x20a9, 0x20a8, 0x20a7, 0x3299, 0x294b, 0x3293,
0x20a4, 0x20a3, 0x3289, 0x2943, 0x2941, 0x3cff, 0x209f, 0x3279,
0x293b, 0x3273, 0x209c, 0x326d, 0x2935, 0x3ccf, 0x2099, 0x3cc3,
0x325f, 0x2097, 0x3259, 0x3cad, 0x2095, 0x3251, 0x2927, 0x2093,
0x3249, 0x3c8d, 0x2091, 0x3c83, 0x323f, 0x3c79, 0x291d, 0x3c6f,
0x3235, 0x3c65, 0x208c, 0x2917, 0x208b, 0x3229, 0x3227, 0x3c49,
0x2911, 0x2088, 0x290f, 0x3c37, 0x3219, 0x3217, 0x3215, 0x3c25,
0x2084, 0x3c1d, 0x2083, 0x2905, 0x2082, 0x2903, 0x2081, 0x2901
};

```

How to Translate between Qscale and Quality Factor

Quantization tables are computed as follows:

```
Qtable = (standard_qtable * qscale + 16) / 32;
```

In the ubiquitous JPEG quality factor case:

```

if (quality < 50)
    quality = 5000/quality;
else
    quality = 200 - 2*quality;
Qtable = (standard_qtable * quality + 50) / 100;

```

So from there, we can derive:

```

if (quality < 50)
    qscale = 1600 / quality;
else
    qscale = 16 * (100 - quality) / 25;

```

Since qscale is 7-bit, the maximum value is 127, which implies the minimum quality is 13 (very poor quality—such a low setting should never be necessary).

The quality factor for the default qscale is as follows:

- qscale1 = 6 --> quality = 90.625 (or 91)
- qscale2 = 9 --> quality = 85.937 (or 86)
- qscale3 = 12 --> quality = 81.25 (or 81)

How to Program the Customized Huffman Table

The MT9D111 can store two sets of Huffman DC and AC tables (one each for luma and chroma). The standard Huffman tables K.3 and K.4 in the ISO/IEC 10918-1 Specification are programmed into Huffman memory by the JPEG driver. These standard Huffman tables can be overwritten by customized Huffman tables through the JPEG Indirect Access Control Register R30:2 and the JPEG Indirect Access Data Register R31:2.

Each AC table contains 176 12-bit words; each DC table contains 16 12-bit words. The memory map of the Huffman memory is shown in Table 31.

Table 31: Huffman Memory Map

First Address	Last Address	Table
0	175	AC Huffman Table 0
176	351	AC Huffman Table 1
352	367	DC Huffman Table 0
368	383	DC Huffman Table 1

Each Huffman code is stored as a record containing the actual code and its length, as shown in the Table 32.

Table 32: Structure of Huffman Code in Huffman Memory

11	10	9	8	7	6	5	4	3	2	1	0
HLEN	HCODE										

HLEN is the number of bits in the Huffman code—HCODE—minus 1.

HCODE are the eight least-significant bits of the Huffman code. If the Huffman code is less than 8 bits long, the bits not used must be 0.

Although Huffman codes used in the JPEG encoding can be up to 16 bits long, when the code is more than 8 bits long, the most significant bits are always 1. Thus it is unnecessary to specify more than 8 bits for any code, as the most significant bits are generated internally within the MT9D111.

In the case of the JPEG baseline algorithm, 162 Huffman codes are required for the encoding of the AC run-length codes and 12 Huffman codes are required for the encoding of the DC coefficients. The location of the Huffman codes for the 162 run-length codes in an AC table is shown in Table 33.

Table 33: Location of AC Huffman Codes in Huffman Memory

Address	Value
0 – 9	Huffman code of run lengths 0/1 to 0/A
10 – 19	Huffman code of run lengths 1/1 to 1/A
20 – 29	Huffman code of run lengths 2/1 to 2/A
30 – 39	Huffman code of run lengths 3/1 to 3/A
40 – 49	Huffman code of run lengths 4/1 to 4/A
50 – 59	Huffman code of run lengths 5/1 to 5/A
60 – 69	Huffman code of run lengths 6/1 to 6/A
70 – 79	Huffman code of run lengths 7/1 to 7/A
80 – 89	Huffman code of run lengths 8/1 to 8/A
90 – 99	Huffman code of run lengths 9/1 to 9/A
100 – 109	Huffman code of run lengths A/1 to A/A
110 – 119	Huffman code of run lengths B/1 to B/A
120 – 129	Huffman code of run lengths C/1 to C/A
130 – 139	Huffman code of run lengths D/1 to D/A
140 – 149	Huffman code of run lengths E/1 to E/A
150 – 159	Huffman code of run lengths F/1 to F/A
160	Huffman code of EOB

Table 33: Location of AC Huffman Codes in Huffman Memory (continued)

Address	Value
161	Huffman code of ZRL
162 – 167	0xFF
168 – 175	0xFD0 – 0xFD7

The location of the Huffman codes for the 12 codes in each DC table is shown in Table 34.

Table 34: Location of DC Huffman Codes in Huffman Memory

Address	Value
0 – 11	Huffman code of DC codes 0 to A
12 – 15	Not used

The address of Huffman memory 0 through 383 is mapped to indirect registers 0x200 through 0x37F.

How to Append the JPEG Header

The MT9D111 outputs the entropy-coded data segments of JPEG data without a header. The user is responsible for adding all JPEG markers except the restart marker. To generate a complete JPEG file in JFIF format, the following JPEG header markers must be added to the beginning of JPEG data:

- Start of image
- JFIF APP0
- Define quantization table
- Start of frame
- Define Huffman table
- Define restart interval
- Start of scan

For the details of JPEG header markers, see the following section on Sample C Code. At the end of JPEG data, an end of image marker (0xFFD9) should be appended.

Sample C Code

The following sample C code is Aptina Proprietary and Confidential.

DO NOT DISTRIBUTE. All or part of the code subject to change
©2005, Aptina Imaging, Inc. All rights reserved.

```
#define FORMAT_YCBCR4220
#define FORMAT_YCBCR4201
#define FORMAT_MONOCHROME2

// JPEG tables
unsigned char JPEG_StdQuantTblY_ZZ[64] =
{
    16, 11, 12, 14, 12, 10, 16, 14,
    13, 14, 18, 17, 16, 19, 24, 40,
    26, 24, 22, 22, 24, 49, 35, 37,
    29, 40, 58, 51, 61, 60, 57, 51,
    56, 55, 64, 72, 92, 78, 64, 68,
```

```

87, 69, 55, 56, 80, 109, 81, 87,
95, 98, 103, 104, 103, 62, 77, 113,
121, 112, 100, 120, 92, 101, 103, 99
};

unsigned char JPEG_StdQuantTblC_ZZ[64] =
{
    17, 18, 18, 24, 21, 24, 47, 26,
    26, 47, 99, 66, 56, 66, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99
};

unsigned int JPEG_StdHuffmanTbl[384] =
{
    0x100, 0x101, 0x204, 0x30b, 0x41a, 0x678, 0x7f8, 0x9f6,
    0xf82, 0xf83, 0x30c, 0x41b, 0x679, 0x8f6, 0xaf6, 0xf84,
    0xf85, 0xf86, 0xf87, 0xf88, 0x41c, 0x7f9, 0x9f7, 0xbf4,
    0xf89, 0xf8a, 0xf8b, 0xf8c, 0xf8d, 0xf8e, 0x53a, 0x8f7,
    0xbf5, 0xf8f, 0xf90, 0xf91, 0xf92, 0xf93, 0xf94, 0xf95,
    0x53b, 0x9f8, 0xf96, 0xf97, 0xf98, 0xf99, 0xf9a, 0xf9b,
    0xf9c, 0xf9d, 0x67a, 0xaf7, 0xf9e, 0xf9f, 0xfa0, 0xfa1,
    0xfa2, 0xfa3, 0xfa4, 0xfa5, 0x67b, 0xbf6, 0xfa6, 0xfa7,
    0xfa8, 0xfa9, 0xfaa, 0xfab, 0xfac, 0xfad, 0x7fa, 0xbf7,
    0xfae, 0faf, 0xfb0, 0xfb1, 0xfb2, 0xfb3, 0xfb4, 0xfb5,
    0x8f8, 0xec0, 0xfb6, 0xfb7, 0xfb8, 0xfb9, 0xfba, 0xfbb,
    0xfbc, 0xfbd, 0x8f9, 0xfbe, 0xfbf, 0xfc0, 0xfc1, 0xfc2,
    0xfc3, 0xfc4, 0xfc5, 0xfc6, 0x8fa, 0xfc7, 0xfc8, 0xfc9,
    0xfca, 0xfcb, 0fcc, 0fcd, 0fce, 0fcf, 0x9f9, 0xfd0,
    0xfd1, 0xfd2, 0xfd3, 0xfd4, 0xfd5, 0xfd6, 0xfd7, 0xfd8,
    0x9fa, 0xfd9, 0xfda, 0xfdb, 0xfdc, 0xfdd, 0xfde, 0xfdf,
    0xfe0, 0xfe1, 0xaf8, 0xfe2, 0xfe3, 0xfe4, 0xfe5, 0xfe6,
    0xfe7, 0xfe8, 0xfe9, 0xfea, 0xfeb, 0xfec, 0xfed, 0fee,
    0xfef, 0xff0, 0xff1, 0xff2, 0xff3, 0xff4, 0xff5, 0xff6,
    0xff7, 0xff8, 0xff9, 0xffa, 0xffb, 0xffc, 0xffd, 0xffe,
    0x30a, 0xaf9, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xfd0, 0xfd1, 0xfd2, 0xfd3, 0xfd4, 0xfd5, 0xfd6, 0xfd7,
    0x101, 0x204, 0x30a, 0x418, 0x419, 0x538, 0x678, 0x8f4,
    0x9f6, 0xbf4, 0x30b, 0x539, 0x7f6, 0x8f5, 0xaf6, 0xbf5,
    0xf88, 0xf89, 0xf8a, 0xf8b, 0x41a, 0x7f7, 0x9f7, 0xbf6,
    0xec2, 0xf8c, 0xf8d, 0xf8e, 0xf8f, 0xf90, 0x41b, 0x7f8,
    0x9f8, 0xbf7, 0xf91, 0xf92, 0xf93, 0xf94, 0xf95, 0xf96,
    0x53a, 0x8f6, 0xf97, 0xf98, 0xf99, 0xf9a, 0xf9b, 0xf9c,
    0xf9d, 0xf9e, 0x53b, 0x9f9, 0xf9f, 0xfa0, 0xfa1, 0xfa2,
    0xfa3, 0xfa4, 0xfa5, 0xfa6, 0x679, 0xaf7, 0xfa7, 0xfa8,
    0xfa9, 0faa, 0xfab, 0xfac, 0xfad, 0fae, 0x67a, 0xaf8,
    0xaf9, 0xfb0, 0xfb1, 0xfb2, 0xfb3, 0xfb4, 0xfb5, 0xfb6,
    0x7f9, 0xfb7, 0xfb8, 0xfb9, 0xfba, 0xfbb, 0xfbc, 0xfbd,
    0xfbe, 0xfbf, 0x8f7, 0xfc0, 0xfc1, 0xfc2, 0xfc3, 0xfc4,
    0xfc5, 0xfc6, 0xfc7, 0xfc8, 0x8f8, 0xfc9, 0xfca, 0xfcb,
    0fcc, 0fcd, 0fce, 0fcf, 0xfd0, 0xfd1, 0x8f9, 0xfd2,
    0xfd3, 0xfd4, 0xfd5, 0xfd6, 0xfd7, 0xfd8, 0xfd9, 0xfda,
    0x8fa, 0xfdb, 0fdc, 0fdd, 0xfde, 0xfdf, 0xfe0, 0xfe1,
    0xfe2, 0xfe3, 0xaf9, 0xfe4, 0xfe5, 0xfe6, 0xfe7, 0xfe8,
    0xfe9, 0xfea, 0xfeb, 0xfec, 0xde0, 0xfed, 0fee, 0fef,
    0xff0, 0xff1, 0xff2, 0xff3, 0xff4, 0xff5, 0xec3, 0xff6,
    0xff7, 0xff8, 0xff9, 0xffa, 0xffb, 0xffc, 0xffd, 0xffe,
    0x100, 0x9fa, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xfd0, 0xfd1, 0xfd2, 0xfd3, 0xfd4, 0xfd5, 0xfd6, 0xfd7,

```

```

    0x100, 0x202, 0x203, 0x204, 0x205, 0x206, 0x30e, 0x41e,
    0x53e, 0x67e, 0x7fe, 0x8fe, 0xfff, 0xfff, 0xfff, 0xfff,
    0x100, 0x101, 0x102, 0x206, 0x30e, 0x41e, 0x53e, 0x67e,
    0x7fe, 0x8fe, 0x9fe, 0xafe, 0xfff, 0xfff, 0xfff, 0xfff
};

int JfifApp0Marker(char *pbuf)
{
    *pbuf++ = 0xFF; // APP0 marker
    *pbuf++ = 0xE0;
    *pbuf++ = 0x00; // length
    *pbuf++ = 0x10;
    *pbuf++ = 0x4A; // JFIF identifier
    *pbuf++ = 0x46;
    *pbuf++ = 0x49;
    *pbuf++ = 0x46;
    *pbuf++ = 0x00;
    *pbuf++ = 0x01; // version
    *pbuf++ = 0x02;
    *pbuf++ = 0x00; // units
    *pbuf++ = 0x00; // X density
    *pbuf++ = 0x01;
    *pbuf++ = 0x00; // Y density
    *pbuf++ = 0x01;
    *pbuf++ = 0x00; // X thumbnail
    *pbuf++ = 0x00; // Y thumbnail

    return 18;
}

int FrameHeaderMarker(char *pbuf, int width, int height, int format)
{
    int length;

    if (format == FORMAT_MONOCHROME)
        length = 11;
    else
        length = 17;

    *pbuf++ = 0xFF; // start of frame: baseline DCT
    *pbuf++ = 0xC0;
    *pbuf++ = length >> 8; // length field
    *pbuf++ = length & 0xFF;
    *pbuf++ = 0x08; // sample precision
    *pbuf++ = height >> 8; // number of lines
    *pbuf++ = height & 0xFF;
    *pbuf++ = width >> 8; // number of samples per line
    *pbuf++ = width & 0xFF;

    if (format == FORMAT_MONOCHROME) // monochrome
    {
        *pbuf++ = 0x01; // number of image components in frame
        *pbuf++ = 0x00; // component identifier: Y
        *pbuf++ = 0x11; // horizontal | vertical sampling factor: Y
        *pbuf++ = 0x00; // quantization table selector: Y
    }
    else if (format == FORMAT_YCBCR422) // YCbCr422
    {
        *pbuf++ = 0x03; // number of image components in frame
        *pbuf++ = 0x00; // component identifier: Y
        *pbuf++ = 0x21; // horizontal | vertical sampling factor: Y
        *pbuf++ = 0x00; // quantization table selector: Y
        *pbuf++ = 0x01; // component identifier: Cb
    }
}

```

```

    *pbuf++ = 0x11;// horizontal | vertical sampling factor: Cb
    *pbuf++ = 0x01;// quantization table selector: Cb
    *pbuf++ = 0x02;// component identifier: Cr
    *pbuf++ = 0x11;// horizontal | vertical sampling factor: Cr
    *pbuf++ = 0x01;// quantization table selector: Cr
  }
else// YCbCr420
{
  *pbuf++ = 0x03;// number of image components in frame
  *pbuf++ = 0x00;// component identifier: Y
  *pbuf++ = 0x22;// horizontal | vertical sampling factor: Y
  *pbuf++ = 0x00;// quantization table selector: Y
  *pbuf++ = 0x01;// component identifier: Cb
  *pbuf++ = 0x11;// horizontal | vertical sampling factor: Cb
  *pbuf++ = 0x01;// quantization table selector: Cb
  *pbuf++ = 0x02;// component identifier: Cr
  *pbuf++ = 0x11;// horizontal | vertical sampling factor: Cr
  *pbuf++ = 0x01;// quantization table selector: Cr
}

return (length+2);
}

int ScanHeaderMarker(char *pbuf, int format)
{
  int length;

  if (format == FORMAT_MONOCHROME)
    length = 8;
  else
    length = 12;

  *pbuf++ = 0xFF;// start of scan
  *pbuf++ = 0xDA;
  *pbuf++ = length>>8;// length field
  *pbuf++ = length&0xFF;

  if (format == FORMAT_MONOCHROME)// monochrome
  {
    *pbuf++ = 0x01;// number of image components in scan
    *pbuf++ = 0x00;// scan component selector: Y
    *pbuf++ = 0x00;// DC | AC huffman table selector: Y
  }
  else// YCbCr
  {
    *pbuf++ = 0x03;// number of image components in scan
    *pbuf++ = 0x00;// scan component selector: Y
    *pbuf++ = 0x00;// DC | AC huffman table selector: Y
    *pbuf++ = 0x01;// scan component selector: Cb
    *pbuf++ = 0x11;// DC | AC huffman table selector: Cb
    *pbuf++ = 0x02;// scan component selector: Cr
    *pbuf++ = 0x11;// DC | AC huffman table selector: Cr
  }
  *pbuf++ = 0x00;// Ss: start of predictor selector
  *pbuf++ = 0x3F;// Se: end of spectral selector
  *pbuf++ = 0x00;// Ah | Al: successive approximation bit position

  return (length+2);
}

int DefineQuantizationTableMarker(char *pbuf, int qscale, int format)
{
  int i, q, length;

```



```

if (format == FORMAT_MONOCHROME) // monochrome
    length = 67;
else
    length = 132;

*pbuff++ = 0xFF; // define quantization table marker
*pbuff++ = 0xDB;
*pbuff++ = length >> 8; // length field
*pbuff++ = length & 0xFF;

*pbuff++ = 0; // quantization table precision | identifier
for (i = 0; i < 64; i++)
{
    q = (JPEG_StdQuantTblY_ZZ[i] * qscale + 16) >> 5;
    *pbuff++ = q; // quantization table element
}

if (format != FORMAT_MONOCHROME)
{
    *pbuff++ = 1; // quantization table precision | identifier
    for (i = 0; i < 64; i++)
    {
        q = (JPEG_StdQuantTblC_ZZ[i] * qscale + 16) >> 5;
        *pbuff++ = q; // quantization table element
    }
}

return (length+2);
}

int DefineHuffmanTableMarkerDC(char *pbuff, unsigned int *htable, int class_id)
{
    int i, l, count;
    char *plength;
    int length;

    *pbuff++ = 0xFF; // define huffman table marker
    *pbuff++ = 0xC4;
    plength = pbuff; // place holder for length field
    *pbuff++;
    *pbuff++;
    *pbuff++ = class_id; // huffman table class | identifier
    for (l = 0; l < 16; l++)
    {
        count = 0;
        for (i = 0; i < 12; i++)
        {
            if ((htable[i] >> 8) == 1)
                count++;
        }
        *pbuff++ = count; // number of huffman codes of length l+1
    }
    length = 19;
    for (l = 0; l < 16; l++)
    {
        for (i = 0; i < 12; i++)
        {
            if ((htable[i] >> 8) == 1)
            {
                *pbuff++ = i; // HUFFVAL with huffman codes of length l+1
                length++;
            }
        }
    }
}

```

```

    }
  }
  *plength++ = length>>8; // length field
  *plength = length&0xFF;

  return (length + 2);
}

int DefineHuffmanTableMarkerAC(char *pbuf, unsigned int *htable, int class_id)
{
  int i, l, a, b, count;
  char *plength;
  int length;
  int eob = 1;
  int zrl = 1;

  *pbuf++ = 0xFF; // define huffman table marker
  *pbuf++ = 0xC4;
  plength = pbuf; // place holder for length field
  *pbuf++;
  *pbuf++;
  *pbuf++ = class_id; // huffman table class | identifier
  for (l = 0; l < 16; l++)
  {
    count = 0;
    for (i = 0; i < 162; i++)
    {
      if ((htable[i] >> 8) == l)
        count++;
    }
    *pbuf++ = count; // number of huffman codes of length l+1
  }
  length = 19;
  for (l = 0; l < 16; l++)
  {
    // check EOB: 0|0
    if ((htable[160] >> 8) == l)
    {
      *pbuf++ = 0; // HUFFVAL with huffman codes of length l+1
      length++;
    }
    // check HUFFVAL: 0|1 to E|A
    for (i = 0; i < 150; i++)
    {
      if ((htable[i] >> 8) == l)
      {
        a = i/10;
        b = i%10;
        *pbuf++ = (a<<4)|(b+1); // HUFFVAL with huffman codes of length l+1
        length++;
      }
    }
    // check ZRL: F|0
    if ((htable[161] >> 8) == l)
    {
      *pbuf++ = 0xF0; // HUFFVAL with huffman codes of length l+1
      length++;
    }
    // check HUFFVAL: F|1 to F|A
    for (i = 150; i < 160; i++)
    {
      if ((htable[i] >> 8) == l)
      {

```

```

        a = i/10;
        b = i%10;
        *pbuf++ = (a<<4)|(b+1); // HUFFVAL with huffman codes of length l+1
        length++;
    }
}
}
*plength++ = length>>8; // length field
*plength = length&0xFF;

return (length + 2);
}

int DefineRestartIntervalMarker(char *pbuf, int ri)
{
    *pbuf++ = 0xFF; // define restart interval marker
    *pbuf++ = 0xDD;
    *pbuf++ = 0x00; // length
    *pbuf++ = 0x04;
    *pbuf++ = ri >> 8; // restart interval
    *pbuf++ = ri & 0xFF;

    return 6;
}

/*
Purpose: Create JPEG header in JFIF format
Parameters: header - pointer to JPEG header buffer
            width - image width
            height - image height
            format - color format (0 = YCbCr422, 1 = YCbCr420, 2 = monochrome)
            restart_int - restart marker interval
            qscale - quantization table scaling factor
Return: length of JPEG header (bytes)
*/
int CreateJpegHeader(char *header, int width, int height, int format, int restart_int, int
qscale)
{
    char *pbuf = header;
    int length;

    // SOI
    *pbuf++ = 0xFF;
    *pbuf++ = 0xD8;
    length = 2;

    // JFIF APP0
    length += JfifApp0Marker(pbuf);

    // Quantization Tables
    pbuf = header + length;
    length += DefineQuantizationTableMarker(pbuf, qscale, format);

    // Frame Header
    pbuf = header + length;
    length += FrameHeaderMarker(pbuf, width, height, format);

    // Huffman Table DC 0 for Luma
    pbuf = header + length;
    length += DefineHuffmanTableMarkerDC(pbuf, &JPEG_StdHuffmanTbl[352], 0x00);

    // Huffman Table AC 0 for Luma
    pbuf = header + length;

```

```
length += DefineHuffmanTableMarkerAC(pbuf, &JPEG_StdHuffmanTbl[0], 0x10);

if (format != FORMAT_MONOCHROME) // YCbCr
{
    // Huffman Table DC 1 for Chroma
    pbuf = header + length;
    length += DefineHuffmanTableMarkerDC(pbuf, &JPEG_StdHuffmanTbl[368], 0x01);

    // Huffman Table AC 1 for Chroma
    pbuf = header + length;
    length += DefineHuffmanTableMarkerAC(pbuf, &JPEG_StdHuffmanTbl[176], 0x11);
}

// Restart Interval
if (restart_int > 0)
{
    pbuf = header + length;
    length += DefineRestartIntervalMarker(pbuf, restart_int);
}

// Scan Header
pbuf = header + length;
length += ScanHeaderMarker(pbuf, format);

return length;
}
```

JPEG Power Saving

Turning off JPEG clock Reg 11:1[4] during preview mode will reduce about 25 percent of current draw from VDD (50mA -> 38mA) and power on VDD is about 50 percent of the total power consumption for the MT9D111. Manually change this bit between preview/capture context switches. Turn on the clock just before DO_CAPTURE, then turn off the clock once the sequencer state is back to Preview or PreviewEnter.

JPEG Functionality FAQs

- Do I have to capture JPEGs in 422 Mode for a full-resolution image?
- MT9D111 spec does not provide information about the status byte that is at the end of each LINE_VALID stream in SPOOF mode. Please provide a description of the field.
- From MT9D111 Spec: “The timing of PIXCLK and Dout within each LINE_VALID assertion period is variable and therefore unlike that of uncompressed data”. Why is the timing different? Spoof mode is supposed to emulate the uncompressed data, which will require no changes on PIXCLK or Dout behavior, right? I was expecting that filler bytes will be placed after JPEG bytes.
- What are the specific recommendations that map the most popular SNAPSHOT resolutions with a preferred programming for the Spoof Frame Width and Spoof Frame Height (IFP Register's Page2 Reg 0x10 and 0x11)? Does Aptina recommend a FIXED Spoof Frame Width/Height regardless of resolution?
- Does the JPEG data length include the 4 bytes used for J0, J1, J2, and status bytes at the end of each LINE_VALID sync?
- Does the JPEG compression algorithm support compressed 565 output?
- How does the use of the JPEG engine affect the frame rate? Use of JPEG should reduce the amount of data transmitted, but does it affect the frame rate?

Do I have to capture JPEGs in 422 Mode for a full-resolution image? Yes, for full-resolution JPEG, you will need 4:2:2 format rather than 4:2:0. For full resolution in JPEG, you can capture in 4:2:2 and monochrome format.

MT9D111 spec does not provide information about the status byte that is at the end of each LINE_VALID stream in SPOOF mode. Please provide a description of the field.

In spoof mode, the JPEG status byte is always appended to the end of the JPEG data stream. In continuous mode, the appending of the JPEG status byte can be optionally enabled by setting mode.FIFO_config0_B[9]=1. The status byte is the value of R2:2[7:0].

From MT9D111 Spec: “The timing of PIXCLK and DOUT within each LINE_VALID assertion period is variable and therefore unlike that of uncompressed data”. Why is the timing different? Spoof mode is supposed to emulate the uncompressed data, which will require no changes on PIXCLK or DOUT behavior, right? I was expecting that filler bytes will be placed after JPEG bytes.

Spoof mode is used such that the LINE_VALID (which in this case is DATA_VALID) are in regular periods, instead of being non-uniform and dependent on when the compressed data is available. The timing of PIXCLK and DOUT are based on when the valid data is available.

What are the specific recommendations that map the most popular SNAPSHOT resolutions with a preferred programming for the Spoof Frame Width and Spoof Frame Height (IFP Register's Page2 Reg 0x10 and 0x11)? Does Aptina recommend a FIXED Spoof Frame Width/Height regardless of resolution?

Spoof frame width and height should be set according to the expected JPEG frame size. We recommend enabling ignore spoof frame height to avoid spoof frame error when the spoof frame size is set too small, and to avoid excessive padding of dummy data when the spoof frame size is set too large. Spoof frame width should be much larger than the

horizontal blanking—spooof LINE_VALID lead plus spooof LINE_VALID trail—to prevent FIFO overflow. The user should set the desired spooof frame width and height to mode.spooof_width_B and mode.spooof_height_B.

Does the JPEG data length include the 4 bytes used for J0, J1, J2, and status bytes at the end of each LINE_VALID sync?

The JPEG data length does not include J0, J1, or J2. Status bytes are appended only at the end of the last LINE_VALID of the JPEG spooof frame, *not* every LINE_VALID.

Does the JPEG compression algorithm support compressed 565 output?

No. We can only output RGB565 format in uncompressed mode.

How does the use of the JPEG engine affect the frame rate? Use of JPEG should reduce the amount of data transmitted, but does it affect the frame rate?

No, the use of JPEG engine would not affect the frame rate. The limiting factor for frame rate is the AE algorithm and image size, blanking, system clock, etc. Regardless of the compression ratio, AE needs to collect enough luminance information from the image and change exposure time and gain to achieve the new target luminance, and therefore limit the frame rate.

Appendix A—How to Update Demo2 Firmware

To run the MT9D111, the firmware in the demo2 board to B3_7.1C must be updated.

1. Install the new DevWare 2.6 Beta x
2. Go to C:\Program File\Aptina Imaging\
3. Run HardwareUpdate.exe
4. Select “Check for FPGA update”
5. Select “Pick customer xvf file” ~\fpga\
Program: 100700b3.xvf (without v)
Verify: v100700b3.xvf (starts with v)
6. Wait until FPGA update is complete, then click “Finish”
7. Run HardwareUpdate.exe again
8. Check for firmware update
9. Pick customer xvf file ~\firmware\1007001C.bin
10. Wait until the firmware update is complete, then click “Finish”

Appendix B—Miscellaneous FAQs

- Fixed Pattern Noise (FPN) appears in Preview mode in low light conditions when 2 ADC are used in Context A. How do we reduce the FPN?
- When should REFRESH be used? When not? What does REFRESH do? When does REFRESH start to affect the system, after how many frames, etc.? What is the difference between REFRESH and REFRESH_MODE?
- When can the registers be written? When not?
- Is there a need for a delay after the REFRESH command?
- Does the MT9D111 contain flash memory? Can the user access the MCU on the chip for general computing?
- What is the chief ray angle requirement for the 2 MP MT9D111?
- What does “PRNU” stand for?
- What is TCLKIN? Is this the period of master clock?
- Is it possible to have an ISO setting in the MT9D111?
- How do I change the PLL setting after the firmware is running?
- Why does random noise appear in low light conditions?
- How do you adjust several functions (AE target, AWB parameter) after power-on and initialization? I think that it is not possible to set the parameter while the firmware is running...
- Why does random noise appear in low light conditions?
- Do I have to wait after a REFRESH command?
- Can VDDPLL be left unconnected if PLL is not used (by default, we knew the PLL is bypassed and powered down)?
- Can VDDGPIO be left unconnected if GPIO pins all are not used?
- Are the registers REG0x61:0, REG0x62:0, REG0x63:0, and REG0x64:0 related to a row noise? When row noise occurs, these register values seem to change a lot.
- According to the specifications sheet, I have to issue the REFRESH command (seq.cmd = 5) when changing the image resolution, effect, gamma and so on. In this timing, it seems that hardware registers were changed by drivers. See the following example for an explanation:
- How should we treat the STANDBY terminal? Should it be connected to GND or open?

Fixed Pattern Noise (FPN) appears in Preview mode in low light conditions when 2 ADC are used in Context A. How do we reduce the FPN?

The AE settings show that it was in the highest zone number, which means that all the gains are at maximum values due to the dark lighting conditions. In such a scenario (almost complete darkness), we do expect some column FPN. However, the column FPN can be reduced by:

- Limiting all the maximum gain settings in the AE driver (ID=2)
- Use 1 ADC. Since the frame rate will be reduced in dark conditions, 1 ADC is enough for the slower FPS.

When should REFRESH be used? When not? What does REFRESH do? When does REFRESH start to affect the system, after how many frames, etc.? What is the difference between REFRESH and REFRESH_MODE?

REFRESH and REFRESH_MODE both upload image processing parameters (from drivers to IP hardware) for the current context. However, only REFRESH_MODE uploads sensor size and timing parameters and only REFRESH_MODE updates the drivers' windows (IP statistics). REFRESH does not upload the sensor size and timing parameters.

ters, but REFRESH does update/reset all of the drivers' operating values (for example, AE limits, Low-light parameters, etc.). For crop/pan changes (like digital zoom), use REFRESH_MODE. These changes are applied at the end of the current frame in which the REFRESH/REFRESH_MODE command is issued.

When can the registers be written? When not?

Registers may be written any time the MT9D111 is awake and running via the two-wire serial interface. However, to avoid mid-frame changes in an image's appearance, avoid changing Sensor Core registers (page 0) or SOC registers (page 1, 2) during the FRAME_VALID time (these may be changed at any time if appearance is not a factor, for example, upon startup). Most changes to image parameters may be written to the driver variables at any time. Driver variables are internally programmed to update the IP hardware at the correct time between frames.

Is there a need for a delay after the REFRESH command?

REFRESH should only be issued once per frame. REFRESH_MODE may be issued as often as possible.

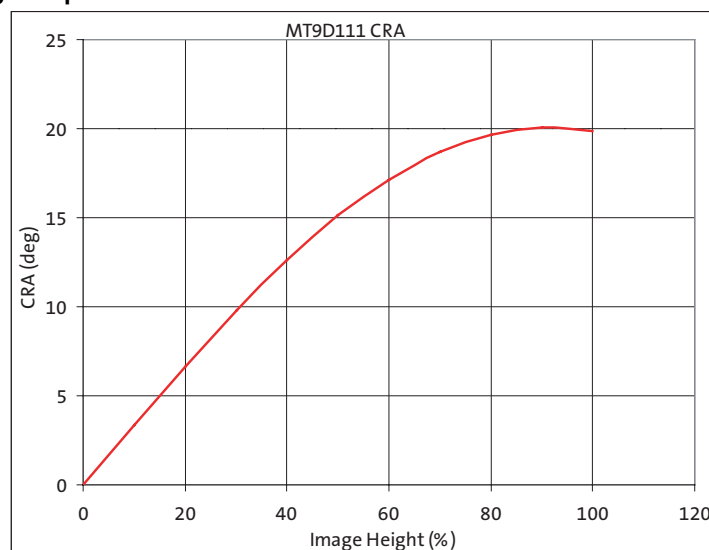
Does the MT9D111 contain flash memory? Can the user access the MCU on the chip for general computing?

This part does not contain flash memory. Also, we have not attempted to use the MT9D111 to drive an external flash part. The MCU contained in the MT9D111 is not designed to be used for general computing.

What is the chief ray angle requirement for the 2 MP MT9D111?

The chief ray angle (CRA) requirement for the 2MP MT9D111 is captured in Figure 68.

Figure 68: Chief Ray Angle Requirement for 2MP MT9D111



What does "PRNU" stand for?

PRNU stands for "pixel response non-linearity."

What is TCLKIN? Is this the period of master clock?

TCLKIN is the period of master clock cycle (if PLL is not used) or PLL cycle (if PLL is enabled).

Is it possible to have an ISO setting in the MT9D111?

The MT9D111 does not have registers that map directly to an ISO-equivalent setting. Since our sensors employ an electronic rolling shutter (ERS), the meaning and application of ISO is different (and less beneficial) than it is for a traditional camera with mechanical shutters. ISO is essentially a control of sensitivity to light. For the MT9D111, it means to have different exposure time and gains. Our parts have automatic gain control (hence auto ISO speed). In order to have fixed sensitivity, you would have to either:

1. Turn off the firmware drivers (AE, AWB, etc.) to manually control gain and exposure time. However, this might lead to incorrect exposure and flicker detection.
2. Fix the max/min gains and max/min indexes (“zones”) settings in the AE driver. By locking the camera into a certain zone and gains, you will have fixed frame rate and gains.

Again, the MT9D111 does not provide direct ISO settings. You may control the gain/exposure time to control the sensitivity (to simulate ISO settings), but doing so is not likely to work as well as the auto mode.

How do I change the PLL setting after the firmware is running?

You may change the PLL settings (M/N/P) during operation by bypassing the PLL first. For example:

```
REG=0, 0x65, 0xA000 // bypass PLL
...change PLL settings here...
REG=0, 0x65, 0x2000 // disable PLL bypass
```

How do you adjust several functions (AE target, AWB parameter) after power-on and initialization? I think that it is not possible to set the parameter while the firmware is running.

We are considering the following sequence:

1. Preview mode set
2. Firmware stop
3. Parameter setting (ex. AE target...)
4. REFRESH command (#Invoke Sequencer Refresh[B103 0005])

Firmware parameters (AE, AWB, etc.) may be adjusted after initialization. You should note that some firmware variables are updated immediately, while others require a REFRESH/REFRESH_MODE command after the change in values. For example, if ae.target (ID=2, Offset=6) is changed from the default value of 60 to 10, the image brightness changes suddenly (the image becomes very dark).

Why does random noise appear in low light conditions?

At under 5 lux, we do expect some noise in the image. However, there are several ways to improve the image quality in low-light conditions, including the “night mode” preset below:

```
[Increased Gain/Lower Saturation (Night Mode)]
VAR8=2, 0x10, 0x0080 // AE_MAX_VIRTGAIN
```

```
VAR8=2, 0x18, 0x0080 // AE_MAXGAIN23
VAR8=2, 0x0E, 0x0018 // AE_MAX_INDEX
VAR8=2, 0x16, 0x0060 // AE_MAX_DGAIN_AE2
VAR8=7, 0x43, 0x0042 // MODE_GAM_CONT_A
VAR8=7, 0x44, 0x0042 // MODE_GAM_CONT_B
VAR8=1, 0x18, 0x0040 // SEQ_LLSAT1
VAR8=1, 0x03, 0x0005 // SEQ_CMD
```

If the above settings do not provide improvement, you can also configure the low-light (“LL”) variables in the sequencer driver. For example, you may use:

- seq.sharedParams.LLmode (ID=1, Offset=21)
- Bit 0-change interpolation threshold
- Bit 1-reduce color saturation
- Bit 2-reduce aperture correction
- Bit 3-increase aperture correction threshold
- Bit 4-enable Y filter

For details related to low-light variables, refer to the “Driver Variable-Sequencer Driver (ID=1)” table in the MT9D111 data sheet.

The above settings are only an example and can be tuned to your needs. The maximum gain was increased for low-light conditions, which will introduce more noise. For experimental purposes, you can reduce the maximum allowed gain by the following variables in the AE driver (ID=2):

- ae.maxVirtGain (offset=16) –maximum allowed virtual gain
- ae.maxDGainAE1 (offset=20) –maximum digital gain pre-LC (lens correction)
- ae.maxDGainAE2 (offset=22) –maximum digital gain post-LC (lens correction)
- ae.mazGain23 (offset=24) –maximum gain to increase in low-light before dropping frame rate

Do I have to wait after a REFRESH command?

Changes to the firmware variables would be effective starting the next frame after the REFRESH command is called, except for a few cases related to changes to integration time (which takes two frames instead of one). Hence there is no significant delay. If you are using DevWare, you will notice longer delays, but these delays are due to the software, not the chip itself.

Can VDDPLL be left unconnected if PLL is not used (by default, we knew the PLL is bypassed and powered down)?

For ESD reasons, we do not recommend leaving VDDPLL floating even if they are not used. If they are not used, they will not consume much power.

Can VDDGPIO be left unconnected if GPIO pins all are not used?

For ESD reasons, we do not recommend leaving VDDGPIO floating even if they are not used. Unused VDDGPIO won't consume much power.

**Are the registers REG0x61:0, REG0x62:0, REG0x63:0, and REG0x64:0 related to a row noise?
When row noise occurs, these register values seem to change a lot.**

An offset can be provided to the ADC for green1, blue, red, and green2 (determined by 0x61-64:0). When 0x60:0[0]=0 (default), the values in R0x61-64:0 are read-only; they show the offset values determined by the black-level calibration algorithm. The purpose of this is to fully utilize the ADC input dynamic range.

According to the specifications sheet, I have to issue the REFRESH command (seq.cmd = 5) when changing the image resolution, effect, gamma and so on. In this timing, it seems that hardware registers were changed by drivers. See the following example for an explanation:

First, I changed the slew rate control register in R14:2 and R15:2 (hardware registers), but in this timing, I do not change the slew rate control variables in the mode driver(ID=7). Next, I issued the REFRESH command to change the image resolution. After issuing the command, the hardware registers were changed by the drivers.

I think that there are many registers like this. I would like to know which registers were changed by drivers when I issued the seq.cmd =5. I cannot understand how to set the register correctly without this information. The REFRESH command (seq.cmd=5) is only needed when variable (not register) settings are changed. As you have noted, some variables are mapped to registers, so the variable value overwrites the associated register. Since the MT9D111 has many features and registers, the firmware is designed such that the user can make programming easier (because the firmware calculates and programs multiple registers automatically).

Therefore, we recommend programming via variables instead of registers whenever possible. For example, slew rate values should be programmed from the firmware (mode driver) instead of the SOC page2 register. Most frequently used settings related to the color pipeline can be configured using firmware variables.

How should we treat the STANDBY terminal? Should it be connected to GND or open?

STANDBY should not be left open. If it is not used, it should be tied to ground.

Appendix C—Glossary of Terms

Table 35: Glossary of Terms

Term	Definition
ADC	Analog-to-digital converter
APS	Active-pixel sensor. The CMOS active-pixel sensor is a second-generation solid state sensor technology that was invented and developed at JPL. CMOS APS technology utilizes active transistors in each pixel to buffer the photo signal. The performance of this technology is comparable to charge-coupled devices (CCDs). Because CMOS APS is inherently CMOS-compatible, it is easy to integrate on-chip timing, control, and drive electronics, reducing system cost and complexity.
AWB	Auto white balance
Bayer color filter array	Color space jointly developed by Microsoft and Hewlett Packard as a color standard. Refer to http://www.w3.org/Graphics/Color/sRGB .
Bi Level	An image with only two colors
Bitmap	An image containing only raster information
BMP	A bitmap format
CCD	Charge-coupled device—one of the two main types of image sensors used in digital cameras
CCM	Color correction module or color correction matrix
Chrominance	Comprises the two components of a television signal that encode color information. Chrominance defines the difference between a color and a chosen reference color of the same luminous intensity.
CMOS	Complementary metal-oxide semiconductor
CMYK	Cyan, magenta, yellow and black. A color printing system that uses these colors. See RGB.
Dithering	A method of displaying colors that are not available on a printer, monitor, etc., by mixing available colors.
DRAM	Dynamic RAM
ERS	Electronic rolling shutter
FIFO	First in first out
fps	Frames per second
Gamma	Different platforms such as PC, Mac, and UNIX interpret color values slightly differently. Any images that look dark on a PC might look bright on a Mac. The gamma correction is a way to explain how an image should be displayed.
Gamma characteristic	A gamma characteristic is a power-law relationship that approximates the relationship between the encoded luminance in a television system and the actual desired image brightness. With this nonlinear relationship, equal steps in encoded luminance correspond to subjectively approximately equal steps in brightness.
GIF	Graphics interchange format
Halftone	A gray-scale image represented by bi-level information
IFP	Image flow processor—performs color recovery and correction, sharpening, gamma correction, lens shading correction, and on-the-fly defect correction
Interlaced	Graphic data is split (usually into two parts), and displayed alternately line by line.
Interlacing	Also known as progressive display—GIF, JPG and TIFF have supported this feature since the early 1990s.
JPG	Joint photographic experts group—a file format.
LC	Lens shading correction
LED	Light emitting diode
LSB	Least significant bit
Luma	Intensity or brightness component of pixel information

Table 35: Glossary of Terms (continued)

Term	Definition
Luminance	The quality of being luminous—emitting or reflecting light. Luminosity is measured relative to that of our sun.
MSB	Most significant bit.
MTF	Modulation transfer function—the sharpness of a photographic imaging system or of a component of the system (lens, film, scanner, enlarging lens, etc.) is characterized by MTF.
NTSC	National Television Standards Committee—the North American standard (525-line interlaced raster-scanned video) for the generation, transmission, and reception of television signals.
Output Resolutions	Includes, but is not limited to, VGA, QVGA, CIF, and QCIF.
PAL	Phase Alternating Line, Phase Alternation by Line or Phase Alternation Line—mainly a European standard of displaying analog television signals. It consists of 625 horizontal lines of resolution at 50Hz. Also see NTSC.
Palette	A set of colors that can be used for a spec output device.
Pixels	A short name for picture element. The smallest part that can be displayed on a monitor.
PPS	Passive-pixel sensors (1960s).
Progressive Scan	An image sensor that gathers its data and processes each scan line one after another in sequence. Compare to Interlaced.
Resolution	A measure on how much information is stored in an image.
RGB	Red, green and blue. A way to represent color on a monitor. Also see CMYK.
RST	Reset
SADDR	Sensor address
SCLK	Serial clock
SOC	System-on-a-chip
SRAM	Static random access memory. SRAM is a type of memory that is faster and more reliable than the more common DRAM (dynamic RAM). The term static is derived from the fact that it does not need to be refreshed like dynamic RAM. Due to its expense, SRAM is often used only as a memory cache.
TIFF	Tagged image file format
True Color	24-bit color. 16,777,216 colors.
TWAIN	A software use to control the communication between scanners and image processing software.
V _{REF}	Voltage reference

Revision History

Rev D, Production	12/09
<ul style="list-style-type: none"> Updated to Aptina template 	
Rev C, Advance?	11/06
<ul style="list-style-type: none"> Update section "Flicker Avoidance" on page 51 (pages 50-53) (add How to Fine Tune the Anti-Flick Driver Setting, How to Verify the Setting, and How to Modify the Setting for Specific Application. Update section "Auto Focus Mechanism" on page 94, after Introduction (HD80([I²C] through MD115 [PWM]) (pages 92-100 and 102-103) 	
Rev B, Advance? , 11/05	
<ul style="list-style-type: none"> Update Table 13, "Public Variables of the Auto Focus Driver," on page 85 (Offset 18) Add "Second Auto Focus Scan" on page 71 Add "Edge Detection" on page 79 Add "Currently Supported AFM Mechanics" on page 130 Add Table 26, "AFM Mechanics Supported," on page 130 	
Rev A, Advance, Draft	6/05