# 1/3.2-Inch System-On-A-Chip (SOC) CMOS Digital Image Sensor

**MT9D111 Data Sheet**

## Features

- Superior low-light performance
- Ultra-low-power, low-cost
- Internal master clock generated by on-chip phase-locked loop oscillator (PLL)
- Electronic rolling shutter (ERS), progressive scan
- Integrated image flow processor (IFP) for single-die camera module
- Automatic image correction and enhancement, including lens shading correction
- Arbitrary image decimation with anti-aliasing
- Integrated real-time JPEG encoder
- Integrated microcontroller for flexibility
- Two-wire serial interface providing access to registers and microcontroller memory
- Selectable output data format: ITU-R BT.601 (YCbCr), 565RGB, 555RGB, 444RGB, JPEG 4:2:2, JPEG 4:2:0, and raw 10-bit
- Output FIFO for data rate equalization
- Programmable I/O slew rate
- Xenon and LED flash support with fast exposure adaptation
- Flexible support for external auto focus, optical zoom, and mechanical shutter

## Applications

- Cellular phones
- PC cameras
- PDAs

## General Description

Aptina™ MT9D111 is a 1/3.2 inch 2-megapixel CMOS image sensor with an integrated advanced camera system. The camera system features a microcontroller (MCU) and a sophisticated image flow processor (IFP) with a real-time JPEG encoder. It also includes a programmable general purpose I/O module (GPIO), which can be used to control external auto focus, optical zoom, or mechanical shutter.

**Table 1: Key Performance Parameters**

| Parameter | | Typical Value |
|---|---|---|
| Optical format | | 1/3.2-inch (4:3) |
| Full resolution | | 1,600 x 1,200 pixels (UXGA) |
| Pixel size | | 2.8µm x 2.8µm |
| Active pixel array area | | 4.73mm x 3.52mm |
| Shutter type | | Electronic Rolling Shutter (ERS) with Global Reset |
| Maximum frame rate | | 15 fps at full resolution, 30 fps in preview mode, (800 x 600) |
| Maximum data rate/ master clock | | 80 MB/s 6 MHz to 80 MHz |
| Supply voltage | Analog | 2.5V–3.1V |
| | Digital | 1.7V–1.95V |
| | I/O | 1.7V–3.1V |
| | PLL | 2.5V–3.1V |
| ADC resolution | | 10-bit, on-die |
| Responsivity | | 1.0/lux-sec (550nm) |
| Dynamic range | | 71dB |
| $SNR_{MAX}$ | | 42.3dB |
| Power consumption | | 348mW at 15 fps, full resolution |
| | | 223mW at 30 fps, preview mode |
| Operating temperature | | -30°C to +70°C |
| Package | | Bare die |

The microcontroller manages all components of the camera system and sets key operation parameters for the sensor core to optimize the quality of raw image data entering the IFP. The sensor core consists of an active pixel array of 1668 x 1248 pixels, programmable timing and control circuitry including a PLL and external flash support, analog signal chain with automatic offset correction and programmable gain, and two 10-bit A/D converters (ADC). The entire system-on-a-chip (SOC) has ultra-low power requirements and superior low-light performance that is particularly suitable for mobile applications.

## Table of Contents

Features . . . 1
Applications . . . 1
General Description . . . 1
Feature Overview . . . 8
Typical Connection . . . 9
Ballout and Interface . . . 10
Architecture Overview . . . 12
  Sensor Core . . . 12
  Color Pipeline . . . 13
    Test Pattern . . . 13
    Black Level Conditioning and Digital Gain . . . 14
    Lens Shading Correction . . . 14
    Line Buffers . . . 14
    Defect Correction . . . 14
    Color Interpolation and Edge Detection . . . 14
    Color Correction and Aperture Correction . . . 15
    Gamma Correction . . . 15
    YUV Processing . . . 15
    Image Cropping and Decimation . . . 15
    YUV-to-RGB/YUV Conversion and Output Formatting . . . 16
  JPEG Encoder and FIFO . . . 16
    JPEG Encoding Highlights . . . 16
    Output Buffer Overflow Prevention . . . 17
  Output Interface . . . 18
    Control (Two-Wire Serial Interface) . . . 18
    Data . . . 18
  Auto Focus . . . 19
    Algorithm . . . 19
    Modes . . . 20
    Lens Actuator Interface . . . 21
    Context and Operational Modes . . . 22
  Auto Exposure . . . 22
    Preview Mode . . . 23
    Scene Evaluative Algorithm . . . 23
  Auto White Balance . . . 23
  Flicker Detection . . . 23
Registers and Variables . . . 24
  How to Access . . . 24
    Registers . . . 24
    Variables . . . 24
    Special Function Registers (SFR) and MCU SRAM . . . 25
Registers . . . 26
  Sensor Core Registers . . . 26
Output Format and Timing . . . 28
  YUV/RGB Uncompressed Output . . . 28
  Uncompressed YUV/RGB Data Ordering . . . 29
  Uncompressed 10-Bit Bypass Output . . . 30
  JPEG Compressed Output . . . 30
  Color Conversion Formulas . . . 33
    Y'Cb'Cr' ITU-R BT.601 . . . 33
    Y'U'V' . . . 33

Y'Cb'Cr Using sRGB Formulas . . . . . 33
Y'U'V' Using sRGB Formulas . . . . . 33
Sensor Core. . . . . 35
Introduction . . . . . 35
Pixel Array Structure . . . . . 36
Default Readout Order . . . . . 37
Raw Data Format . . . . . 37
Raw Data Timing . . . . . 38
Registers . . . . . 40
Double-Buffered Registers . . . . . 40
Bad Frames . . . . . 40
Changes to Integration Time . . . . . 40
Changes to Gain Settings . . . . . 41
Feature Description . . . . . 42
PLL Generated Master Clock . . . . . 42
PLL Setup . . . . . 42
PLL Power-up . . . . . 42
Window Control . . . . . 43
Window Start . . . . . 43
Window Size . . . . . 43
Pixel Border . . . . . 43
Readout Modes . . . . . 43
Readout Speeds and Power Savings . . . . . 43
Column Mirror Image. . . . . 43
Row Mirror Image . . . . . 43
Column and Row Skip . . . . . 44
Digital Zoom . . . . . 46
Binning . . . . . 46
Binning Limitations . . . . . 46
Frame Rate Control . . . . . 47
Minimum Horizontal Blanking . . . . . 47
Minimum Row Time Requirement . . . . . 47
Context Switching . . . . . 48
Integration Time. . . . . 48
Maximum Shutter Delay . . . . . 49
Flash STROBE . . . . . 49
Global Reset. . . . . 50
Analog Signal Path . . . . . 51
Stage-by-Stage Transfer Functions . . . . . 52
VREFD . . . . . 52
Gain Settings: G1, G2, G3 . . . . . 52
Offset Voltage: VOFFSET. . . . . 53
Recommended Gain Settings . . . . . 53
Analog Inputs AIN1–AIN3 . . . . . 53
Firmware . . . . . 55
Overview of Architecture . . . . . 55
Bootstrap Routine . . . . . 55
Drivers . . . . . 55
Extending and Overriding Drivers . . . . . 56
Variables . . . . . 56
Uploading Custom Code . . . . . 56
Hardware Resource Programming . . . . . 56
. . . . . DMA56

Watchdog . . . 56
Sleep and Wakeup . . . 57
Internal Register Bus Access . . . 57
GPIO and Waveform Generator on Local Bus . . . 57
High-Precision Timer . . . 57
Sleep and Wakeup Programming . . . 57
Two-Wire Serial Interface Programming . . . 57
DMA Programming . . . 57
Warm Restarts and Watchdog Programming . . . 57
High-Precision Timer . . . 58
Safe and Test Modes . . . 58
Application Scheduling . . . 58
Monitor Driver . . . 58
Sequencer Driver . . . 59
Flash Types and Settings . . . 60
Low-Light Operation . . . 61
Firmware . . . 62
Auto Exposure Driver . . . 62
Evaluative Auto Exposure . . . 62
Auto White Balance Driver . . . 62
Auto Focus Driver . . . 63
Auto Focus Mechanics Driver . . . 64
Mode Driver . . . 64
Histogram Driver . . . 65
Flicker Avoidance Driver . . . 65
JPEG Driver . . . 66
Usage . . . 66
Table Programming . . . 66
Error Handling and Handshaking . . . 67
Start-Up and Usage . . . 67
Power-Up . . . 68
Hard Reset Sequence . . . 68
Soft Reset Sequence . . . 68
Enable PLL . . . 68
Configure Pad Slew . . . 69
Configure Preview Mode . . . 69
Configure Capture Mode . . . 69
Perform Lock or Capture . . . 69
Standby Sequence . . . 69
To Enter Standby . . . 70
To Exit Standby . . . 70
Standby Hardware Configuration . . . 71
Output Enable Control . . . 71
Power-Down . . . 71
Contrast and Gamma Settings . . . 71
S-Curve . . . 73
Auto Exposure . . . 75
Preview . . . 75
Evaluative . . . 76
Exposure Control . . . 76
Lens Correction Zones . . . 77
Lens Correction Procedure . . . 78
Color Correction . . . 80

Decimator . . . . . .80
General Purpose I/O . . . . . .81
Introduction . . . . . .81
GPIO Output Control Overview . . . . . .81
Waveform Programming . . . . . .82
Notification Signals . . . . . .84
Digital and Analog Inputs . . . . . .85
GPIO Software Drivers . . . . . .85
Auto Focus . . . . . .85
Algorithm Description . . . . . .85
Evaluation of Image Sharpness . . . . . .91
Spectral Characteristics . . . . . .94
Die Outline . . . . . .95
Electrical Specifications . . . . . .96
I/O Timing . . . . . .98
Appendix A: Two-Wire Serial Register Interface . . . . . .99
Protocol . . . . . .99
Sequence . . . . . .99
Bus Idle State . . . . . . 100
Start Bit . . . . . . 100
Stop Bit . . . . . . 100
Slave Address . . . . . . 100
Data Bit Transfer . . . . . . 101
Acknowledge Bit . . . . . . 101
No-Acknowledge Bit . . . . . . 101
Page Register . . . . . . 101
Sample Write and Read Sequences . . . . . . 101
16-Bit Write Sequence . . . . . . 101
16-Bit Read Sequence . . . . . . 102
8-Bit Write Sequence . . . . . . 102
8-Bit Read Sequence . . . . . . 102
Revision History . . . . . . 105

## List of Figures

Figure 1: Typical Configuration (Connection) . . . . . 9
Figure 2: Ball Assignment . . . . . 10
Figure 3: Block Diagram . . . . . 12
Figure 4: Color Pipeline . . . . . 13
Figure 5: JPEG Encoder Block Diagram . . . . . 17
Figure 6: Timing of Decimated Uncompressed Output Bypassing the FIFO . . . . . 28
Figure 7: Timing of Uncompressed Full Frame Output or Decimated Output Passing Through the FIFO . . . 28
Figure 8: Details of Uncompressed YUV/RGB Output Timing . . . . . 29
Figure 9: Example of Timing for Non-Decimated Uncompressed Output Bypassing Output FIFO . . . . . 30
Figure 10: Timing of JPEG Compressed Output in Free-Running Clock Mode . . . . . 32
Figure 11: Timing of JPEG Compressed Output in Gated Clock Mode . . . . . 32
Figure 12: Timing of JPEG Compressed Output in Spoof Mode . . . . . 32
Figure 13: Sensor Core Block Diagram . . . . . 35
Figure 14: Pixel Array . . . . . 36
Figure 15: Pixel Color Pattern Detail (Top Right Corner) . . . . . 36
Figure 16: Imaging a Scene . . . . . 37
Figure 17: Spatial Illustration of Image Readout . . . . . 37
Figure 18: Pixel Data Timing Example . . . . . 38
Figure 19: Row Timing and FRAME_VALID/LINE_VALID Signals . . . . . 38
Figure 20: 6 Pixels in Normal and Column Mirror Readout Modes . . . . . 44
Figure 21: 6 Rows in Normal and Row Mirror Readout Modes . . . . . 44
Figure 22: 8 Pixels in Normal and Column Skip 2x Readout Modes . . . . . 45
Figure 23: 16 Pixels in Normal and Column Skip 4x Readout Modes . . . . . 45
Figure 24: 32 Pixels in Normal and Column Skip 8x Readout Modes . . . . . 45
Figure 25: 64 Pixels in Normal and Column Skip 16x Readout Modes . . . . . 46
Figure 26: Xenon Flash Enabled . . . . . 50
Figure 27: LED Flash Enabled . . . . . 50
Figure 28: LED Flash Enabled, Using Restart . . . . . 50
Figure 29: GLOBAL RESET Operation . . . . . 51
Figure 30: Analog Readout Channel . . . . . 52
Figure 31: Timing Diagram AIN1–AIN3 Sample . . . . . 54
Figure 32: Sequencer Driver . . . . . 59
Figure 33: Gamma Correction Curve . . . . . 72
Figure 34: Contrast “S” Curve . . . . . 73
Figure 35: Tonal Mapping . . . . . 74
Figure 36: Contrast Diagram . . . . . 75
Figure 37: Gain vs. Exposure . . . . . 77
Figure 38: Lens Correction Zones . . . . . 78
Figure 39: Examples of GPIO-Generated Waveforms . . . . . 83
Figure 40: Search for Best Focus . . . . . 87
Figure 41: Scene with Two Potential Focus Targets at Different Distances from Camera . . . . . 89
Figure 42: Dependence of Luminance-Normalized Local Sharpness Scores on Lens Position . . . . . 90
Figure 43: Example of Position Weight Histogram Created by AF Driver . . . . . 90
Figure 44: Auto Focus Windows . . . . . 92
Figure 45: Computation of Sharpness Scores and Luminance Average for an AF Window . . . . . 93
Figure 46: Typical Spectral Characteristics . . . . . 94
Figure 47: Optical Center Offset . . . . . 95
Figure 48: I/O Timing Diagram . . . . . 98
Figure 49: WRITE Timing to R0x09:0—Value 0x0284 . . . . . 101
Figure 50: READ Timing from R0x09:0; Returned Value 0x0284 . . . . . 102
Figure 51: WRITE Timing to R0x09:0—Value 0x0284 . . . . . 102
Figure 52: READ Timing from R0x09:0; Returned Value 0x0284 . . . . . 103
Figure 53: Two-Wire Serial Bus Timing Parameters . . . . . 103

## List of Tables

Table 1: Key Performance Parameters . . . . 1
Table 2: Signal Description . . . . 11
Table 3: Sensor Core Register Defaults . . . . 26
Table 4: YCrCb Output Data Ordering . . . . 29
Table 5: RGB Ordering in Default Mode . . . . 29
Table 6: 2-Byte RGB Format . . . . 30
Table 7: Frame Time . . . . 39
Table 8: Frame—Long Integration Time . . . . 39
Table 9: Frequency Parameters . . . . 42
Table 10: Skip Values . . . . 44
Table 11: Row Addressing . . . . 46
Table 12: Column Addressing . . . . 46
Table 13: Minimum Horizontal Blanking Parameters . . . . 47
Table 14: Minimum Row Time Parameters . . . . 48
Table 15: Offset Gain . . . . 53
Table 16: Recommended Gain Settings . . . . 53
Table 17: Flash Types and Settings . . . . 60
Table 18: Output Enable Control . . . . 71
Table 19: Gamma Settings . . . . 73
Table 20: Contrast Values . . . . 73
Table 21: Examples of AF Filters that can be Programmed into the MT9D111 . . . . 93
Table 22: AC Electrical Characteristics . . . . 96
Table 23: DC Electrical Definitions and Characteristics . . . . 97
Table 24: Absolute Maximum Ratings . . . . 98
Table 25: Slave Address Options . . . . 100
Table 26: Two-wire Serial Bus Characteristics . . . . 104

## Feature Overview

The MT9D111 is a color image sensor with a Bayer color filter arrangement. Its basic characteristics are described in Table 1, "Key Performance Parameters," on page 1.

The excellent low-light performance of MT9D111 is one of the hallmarks of Aptina's breakthrough low-noise CMOS imaging technology that achieves near-CCD image quality (based on signal-to-noise ratio and low-light sensitivity) while maintaining the inherent size, cost, power consumption, and integration advantages of CMOS.

The MT9D111 has an embedded phase-locked loop oscillator (PLL) that can be used with the common wireless system clock. When in use, the PLL adjusts the incoming clock frequency up, allowing the MT9D111 to run at almost any desired resolution and frame rate. To reduce power consumption, the PLL can be bypassed and powered down.

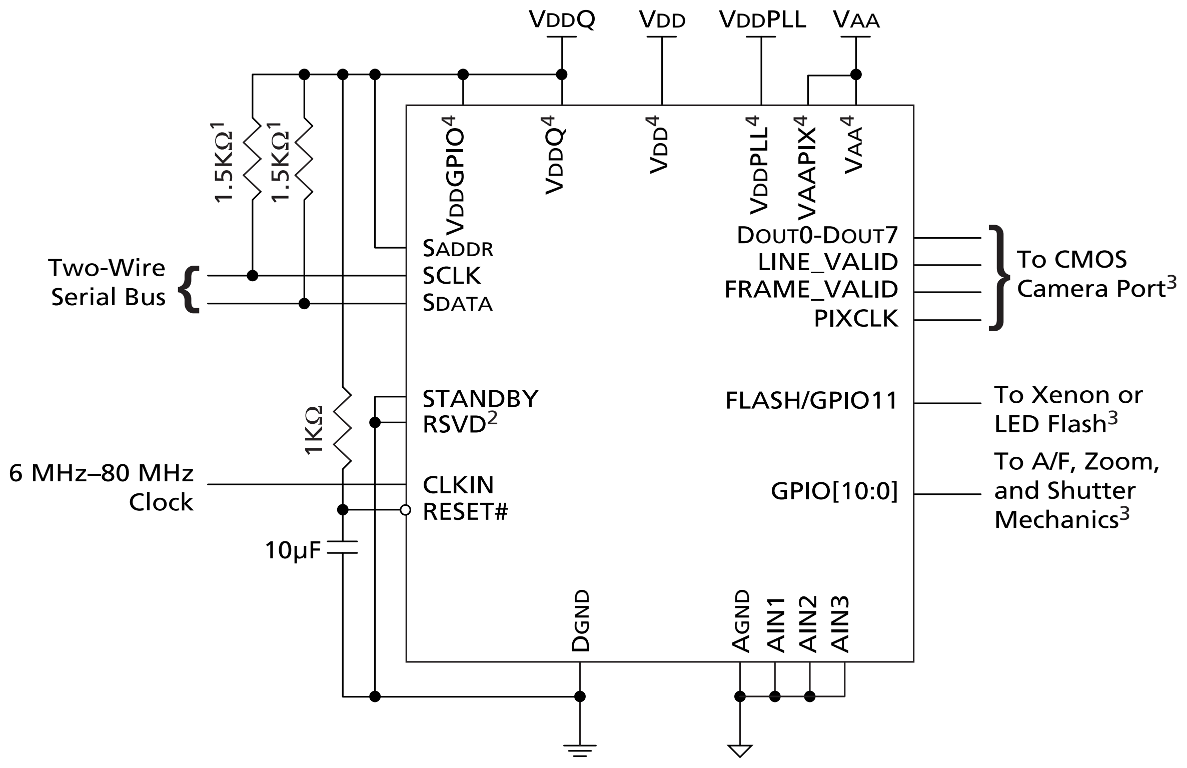
Low power consumption is a very important requirement for all components of wireless devices. The MT9D111 has numerous power conserving features, including an ultra-low power standby mode and the ability to individually shut down unused digital blocks.

Another important consideration for wireless devices is their electromagnetic emission or interference (EMI). The MT9D111 has a programmable I/O slew rate to minimize its EMI and an output FIFO to eliminate output data bursts.

The advanced IFP and flexible programmability of the MT9D111 provide a variety of ways to enhance and optimize the image sensor performance. Built-in optimization algorithms enable the MT9D111 to operate at factory settings as a fully automatic, highly adaptable camera. However, most of its settings are user-programmable.

## Typical Connection

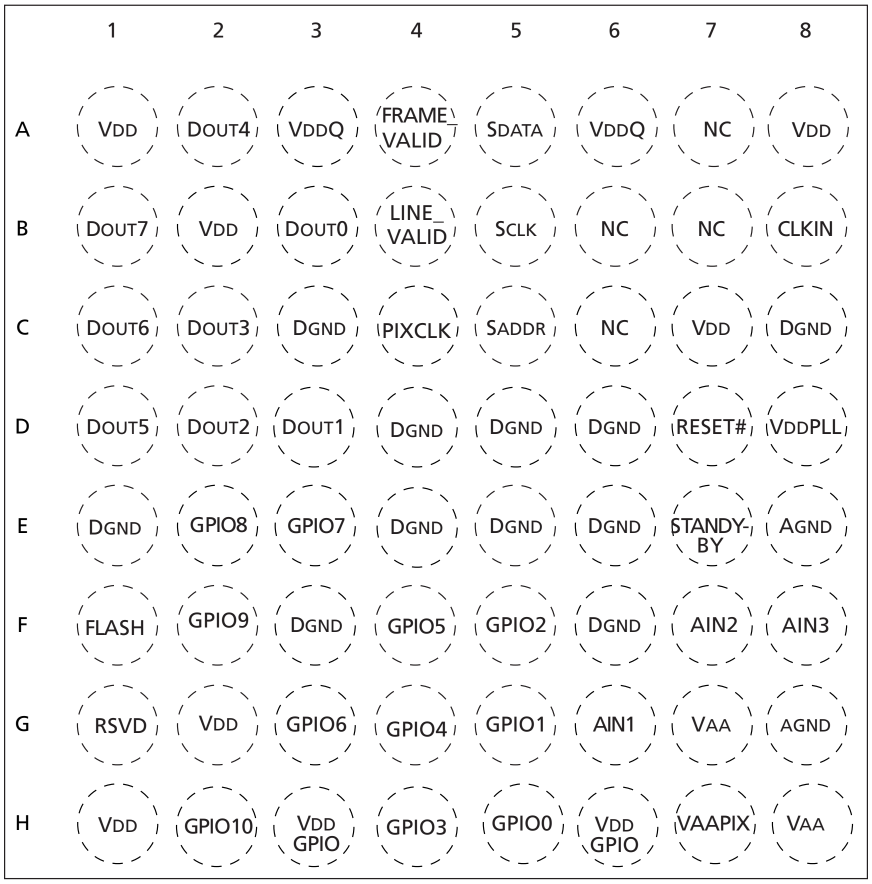
**Figure 1: Typical Configuration (Connection)**



Note:
1. Resistor value 1.5KΩ is recommended, but may be greater for slower two-wire speed.
2. RSVD must be connected to digital ground for normal device operation.
3. See “Standby Hardware Configuration” on page 71.
4. All power supply pads must be used.

## Ballout and Interface

**Figure 2: Ball Assignment**



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| A | VDD | DOUT4 | VDDQ | FRAME_VALID | SDATA | VDDQ | NC | VDD |
| B | DOUT7 | VDD | DOUT0 | LINE_VALID | SCLK | NC | NC | CLKIN |
| C | DOUT6 | DOUT3 | DGND | PIXCLK | SADDR | NC | VDD | DGND |
| D | DOUT5 | DOUT2 | DOUT1 | DGND | DGND | DGND | RESET# | VDDPLL |
| E | DGND | GPIO8 | GPIO7 | DGND | DGND | DGND | STANDBY | AGND |
| F | FLASH | GPIO9 | DGND | GPIO5 | GPIO2 | DGND | AIN2 | AIN3 |
| G | RSVD | VDD | GPIO6 | GPIO4 | GPIO1 | AIN1 | VAA | AGND |
| H | VDD | GPIO10 | VDD GPIO | GPIO3 | GPIO0 | VDD GPIO | VAAPIX | VAA |

Top View
(Ball Down)

Note: All power supply pads must be used.

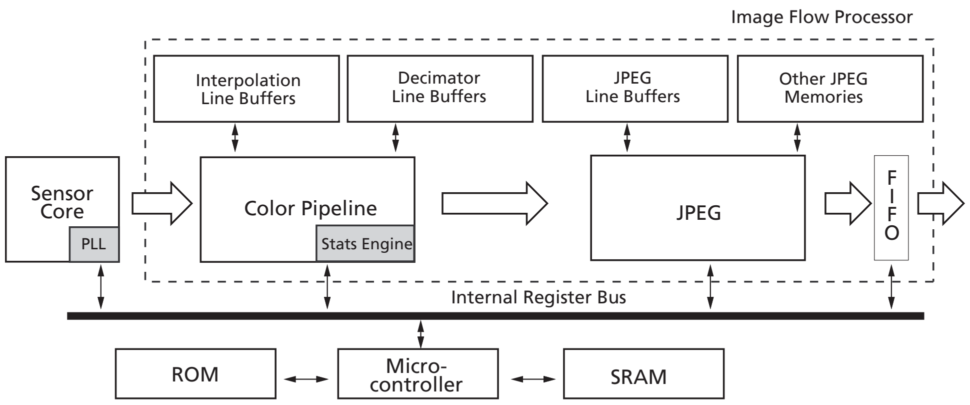
**Table 2: Signal Description**

| Name | Type | Description | Note |
|---|---|---|---|
| CLKIN | Input | Master clock signal (can either drive the on-chip PLL or bypass it). | |
| RESET# | Input | Master reset signal, active LOW. | |
| STANDBY | Input | Controls sensor's standby mode. | |
| RSVD | Input | Reserved for factory test. Tie to digital ground during normal operation. | |
| AIN1 | Input | Analog sampling and test. During normal operation, can be used to feed an external analog signal to an ADC in the sensor core, in order to have the signal sampled during horizontal blanking times and stored in a register. | |
| AIN2 | Input | Analog sampling and test. Can be used like AIN1 during normal operation. | |
| AIN3 | Input | Analog sampling and test. Can be used like AIN1 during normal operation. | |
| SCLK | Input | Two-wire serial interface clock. | |
| SADDR | Input | Selects device address for the two-wire serial interface. The address is 0x90 when SADDR is tied LOW, 0xBA if tied HIGH. See also R0x0D:0[10]. | |
| SDATA | I/O | Two-wire serial interface data. | |
| GPIO[7:0] | I/O | General purpose digital I/O. Each bit can be independently configured as an input or output. Outputs are controlled by register-programmable waveform generator or by writing to registers GPIO_DATA_L and GPIO_DATA_H. Inputs can be sensed by reading the same registers. | 1 |
| FLASH/GPIO11 | I/O | GPIO11 or signal to control Xenon or LED flash. | 1 |
| GPIO10/STROBE | I/O | GPIO10 or signal to control mechanical shutter. | 1 |
| GPIO9/DOUT_LSB | I/O | GPIO9 during normal IFP operation or data bit 1 in 10-bit sensor bypass mode. | 1 |
| GPIO8DOUT_LSB0 | I/O | GPIO8 during normal IFP operation or data bit 0 in 10-bit sensor bypass mode. | 1 |
| DOUT0-DOUT7 | Output | Eight-bit image data output or most significant bits (MSB) of 10-bit sensor bypass mode. | 1 |
| FRAME_VALID | Output | Identifies rows in the active image. | 1 |
| LINE_VALID | Output | Identifies lines in the active image. | 1 |
| PIXCLK | Output | Pixel clock. To be used for sampling DOUT, FRAME_VALID, and LINE_VALID. | 1 |
| VDD | Supply | Digital power (1.8V). | |
| VDDPLL | Supply | PLL power (2.8V). | |
| VAA | Supply | Analog power (2.8V). | |
| VAAPIX | Supply | Pixel array power (2.8V). | |
| VDDQ | Supply | I/O power (nominal 1.8V or 2.8V). | |
| VDDGPIO | Supply | I/O power for GPIO (nominal 1.8V or 2.8V). | |
| AGND | Supply | Analog ground. | |
| DGND | Supply | Digital, I/O, and PLL ground. | |
| NC | — | No connect. | |

Note: 1. See "Standby Hardware Configuration" on page 71.

## Architecture Overview

**Figure 3: Block Diagram**

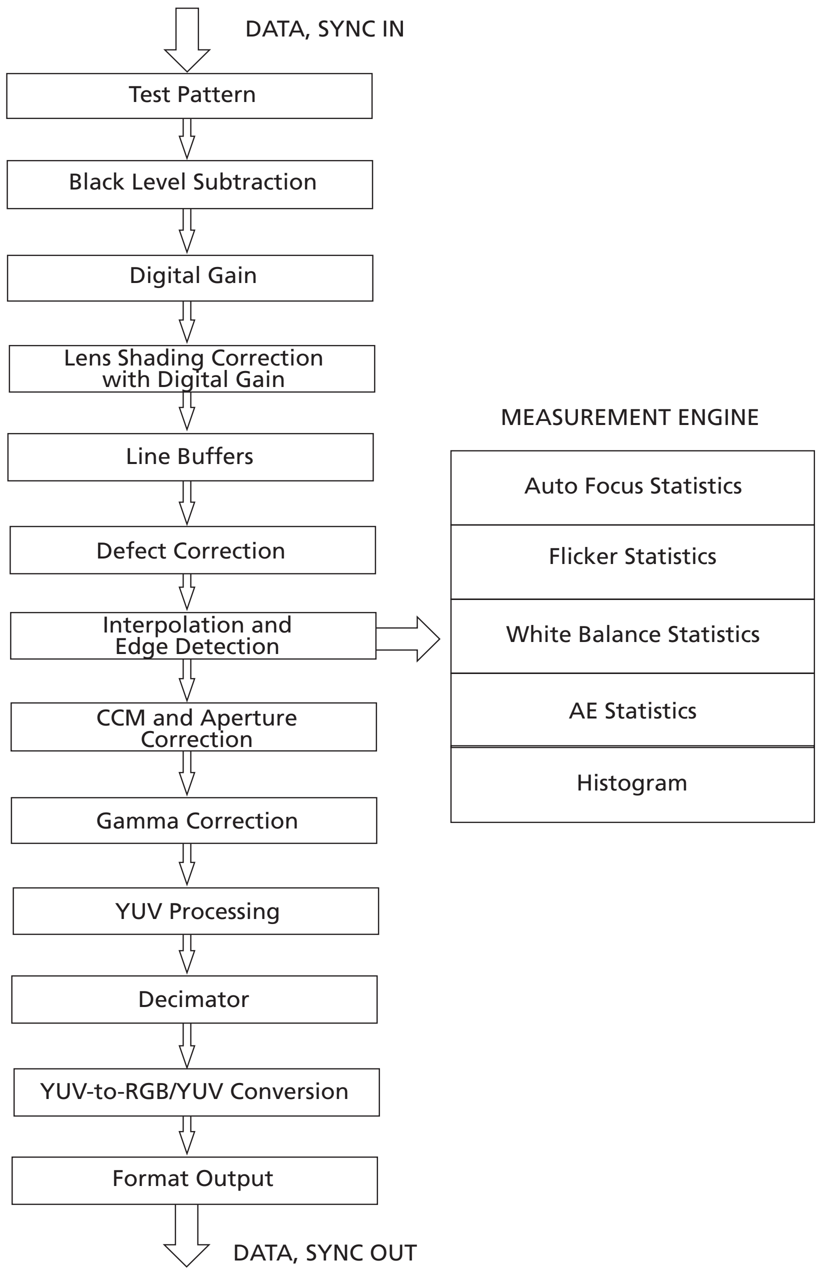


## Sensor Core

The MT9D111 sensor core is based on Aptina's MT9D011, a stand-alone, 2-megapixel CMOS image sensor with a 2.8µm pixel size. Both image sensors have the same optical size (1/3.2 inches) and maximum resolution (UXGA). Like the MT9D011, the MT9D111 sensor core includes a phase-locked loop oscillator (PLL) to facilitate camera integration and minimize the system cost for wireless and mobile applications. When in use, the PLL generates internal master clock signal whose frequency can be set higher than the frequency of external clock signal CLKIN. This allows the MT9D111 to run at any desired resolution and frame rate up to the specified maximum values, irrespective of the CLKIN frequency.

## Color Pipeline

**Figure 4: Color Pipeline**



## Test Pattern

During normal operation of MT9D111, a stream of raw image data from the sensor core is continuously fed into the color pipeline. For test purposes, this stream can be replaced with a fixed image generated by a special test module in the pipeline. The module provides a selection of test patterns sufficient for basic testing of the pipeline.

## Black Level Conditioning and Digital Gain

Image stream processing starts with black level conditioning and multiplication of all pixel values by a programmable digital gain.

## Lens Shading Correction

Inexpensive lenses tend to produce images whose brightness is significantly attenuated near the edges. Chromatic aberration in such lenses can cause color variation across the field of view. There are also other factors causing fixed-pattern signal gradients in images captured by image sensors. The cumulative result of all these factors is known as lens shading. The MT9D111 has an embedded lens shading correction (LC) module that can be programmed to precisely counter the shading effect of a lens on each RGB color signal. The LC module multiplies RGB signals by a 2-dimensional correction function F(x,y), whose profile in both x and y direction is a piecewise quadratic polynomial with coefficients independently programmable for each direction and color.

## Line Buffers

Several data processing steps following the lens shading correction require access to pixel values from up to 8 consecutive image lines. For these lines to be simultaneously available for processing, they must be buffered. The IFP includes a number of SRAM line buffers that are used to perform defect correction, color interpolation, image decimation, and JPEG encoding.

## Defect Correction

The IFP performs on-the-fly defect correction that can mask pixel array defects such as high-dark-current ("hot") pixels and pixels that are darker or brighter than their neighbors due to photoresponse non uniformity. The defect correction algorithm uses several pixel features to distinguish between normal and defective pixels. After identifying the latter, it replaces their actual values with values inferred from the values of nearest same-color neighbors.

## Color Interpolation and Edge Detection

In the raw data stream fed by the sensor core to the IFP, each pixel is represented by a 10-bit integer number, which, to make things simple, can be considered proportional to the pixel's response to a one-color light stimulus, red, green or blue, depending on the pixel's position under the color filter array. Initial data processing steps, up to and including the defect correction, preserve the 1-color-per-pixel nature of the data stream, but after the defect correction it must be converted to a 3-colors-per-pixel stream appropriate for standard color processing. The conversion is done by an edge-sensitive color interpolation module. The module pads the incomplete color information available for each pixel with information extracted from an appropriate set of neighboring pixels. The algorithm used to select this set and extract the information seeks the best compromise between maintaining the sharpness of the image and filtering out high-frequency noise. The simplest interpolation algorithm is to sort the nearest 8 neighbors of every pixel into 3 sets, red, green, and blue, discard the set of pixels of the same color as the center pixel (if there are any), calculate average pixel values for the remaining 2 sets, and use the averages in lieu of the missing color data for the center pixel. Such averaging reduces high-frequency noise, but it also blurs and distorts sharp transitions (edges) in the image. To avoid this problem, the interpolation module performs edge detection in the neighborhood of every processed pixel and, depending on its results, extracts color information from neighboring pixels in a number of different ways. In effect, it does low-pass filtering in flat-field image areas and avoids doing it near edges.

## Color Correction and Aperture Correction

In order to achieve good color fidelity of IFP output, interpolated RGB values of all pixels are subjected to color correction. The IFP multiplies each vector of three pixel colors by a 3 x 3 color correction matrix. The three components of the resulting color vector are all sums of three 10-bit numbers. Since such sums can have up to 12 significant bits, the bit width of the image data stream is widened to 12 bits per color (36 bits per pixel). The color correction matrix can be either programmed by the user or automatically selected by the auto white balance (AWB) algorithm implemented in the IFP. Color correction should ideally produce output colors that are independent of the spectral sensitivity and color cross-talk characteristics of the image sensor. The optimal values of color correction matrix elements depend on those sensor characteristics and on the spectrum of light incident on the sensor.

To increase image sharpness, a programmable aperture correction is applied to color corrected image data, equally to each of the 12-bit R, G, and B color channels.

## Gamma Correction

Like the aperture correction, gamma correction is applied equally to each of the 12-bit R, G, and B color channels. Gamma correction curve is implemented as a piecewise linear function with 19 knee points, taking 12-bit arguments and mapping them to 8-bit output. The abscissas of the knee points are fixed at 0, 64, 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, and 4095. The 8-bit ordinates are programmable via IFP registers or public variables of mode driver (ID = 7). The driver variables include two arrays of knee point ordinates defining two separate gamma curves for sensor operation contexts A and B.

## YUV Processing

After the gamma correction, the image data stream undergoes RGB to YUV conversion and optionally further corrective processing. The first step in this processing is removal of highlight coloration, also referred to as “color kill.” It affects only pixels whose brightness exceeds a certain pre-programmed threshold. The U and V values of those pixels are attenuated proportionally to the difference between their brightness and the threshold. The second optional processing step is noise suppression by 1-dimensional low-pass filtering of Y and/or UV signals. A 3- or 5-tap filter can be selected for each signal.

## Image Cropping and Decimation

To ensure that the size of images output by MT9D111 can be tailored to the needs of all users, the IFP includes a decimator module. When enabled, this module performs “decimation” of incoming images, i.e. shrinks them to arbitrarily selected width and height without reducing the field of view and without discarding any pixel values. The latter point merits underscoring, because the terms “decimator” and “image decimation” suggest image size reduction by deleting columns and/or rows at regular intervals. Despite the terminology, no such deletions take place in the decimator module. Instead, it performs “pixel binning”, i.e. divides each input image into rectangular bins corresponding to individual pixels of the desired output image, averages pixel values in these bins and assembles the output image from the bin averages. Pixels lying on bin boundaries contribute to more than one bin average: their values are added to bin-wide sums of pixel values with fractional weights. The entire procedure preserves all image information that can be included in the downsized output image and filters out high-frequency features that could cause aliasing.

The image decimation in the IFP can be preceded by image cropping and/or image decimation in the sensor core. Image cropping takes place when the sensor core is programmed to output pixel values from a rectangular portion of its pixel array - a window - smaller than the default 1600 x 1200 window. Pixels outside the selected cropping window are not read out, which results in narrower field of view than at the default sensor settings. Irrespective of the size and position of the cropping window, the MT9D111 sensor core can also decimate outgoing images by skipping columns and/or rows of the pixel array, and/or by binning 2 x 2 groups of pixels of the same color. Since decimation by skipping (i.e. deletion) can cause aliasing (even if pixel binning is simultaneously enabled), it is generally better to change image size only by cropping and pixel binning.

The image cropping and decimator module can be used to do digital zoom and pan. If the decimator is programmed to output images smaller than images coming from the sensor core, zoom effect can be produced by cropping the latter from their maximum size down to the size of the output images. The ratio of these two sizes determines the maximum attainable zoom factor. For example, a 1600 x 1200 image rendered on a 160 x 120 display can be zoomed up to 10 times, since 1600/160 = 1200/120 = 10. Panning effect can be achieved by fixing the size of the cropping window and moving it around the pixel array.

### YUV-to-RGB/YUV Conversion and Output Formatting

The YUV data stream emerging from the decimator module can either exit the color pipeline as-is or be converted before exit to an alternative YUV or RGB data format. See “Color Conversion Formulas” on page 33 and the description of register R151:1 for more details.

## JPEG Encoder and FIFO

The JPEG compression engine in the MT9D111 is a highly integrated, high-performance solution that can provide sustained data rates of almost 80MB/s for image sizes up to 1600 x 1200. Additionally, the solution provides for low power consumption and full programmability of JPEG compression parameters for image quality control.

The JPEG encoding block is designed for continuous image flow and is ideal for low power applications. After initial configuration for a target application, it can be controlled easily for instantaneous stop/restart. A flexible configuration and control interface allows for full programmability of various JPEG-specific parameters and tables.

### JPEG Encoding Highlights

1. Sequential DCT (baseline) ISO/IEC 10918-1 JPEG-compliant
2. YCbCr 4:2:2 format compression
3. Programmable quantization tables

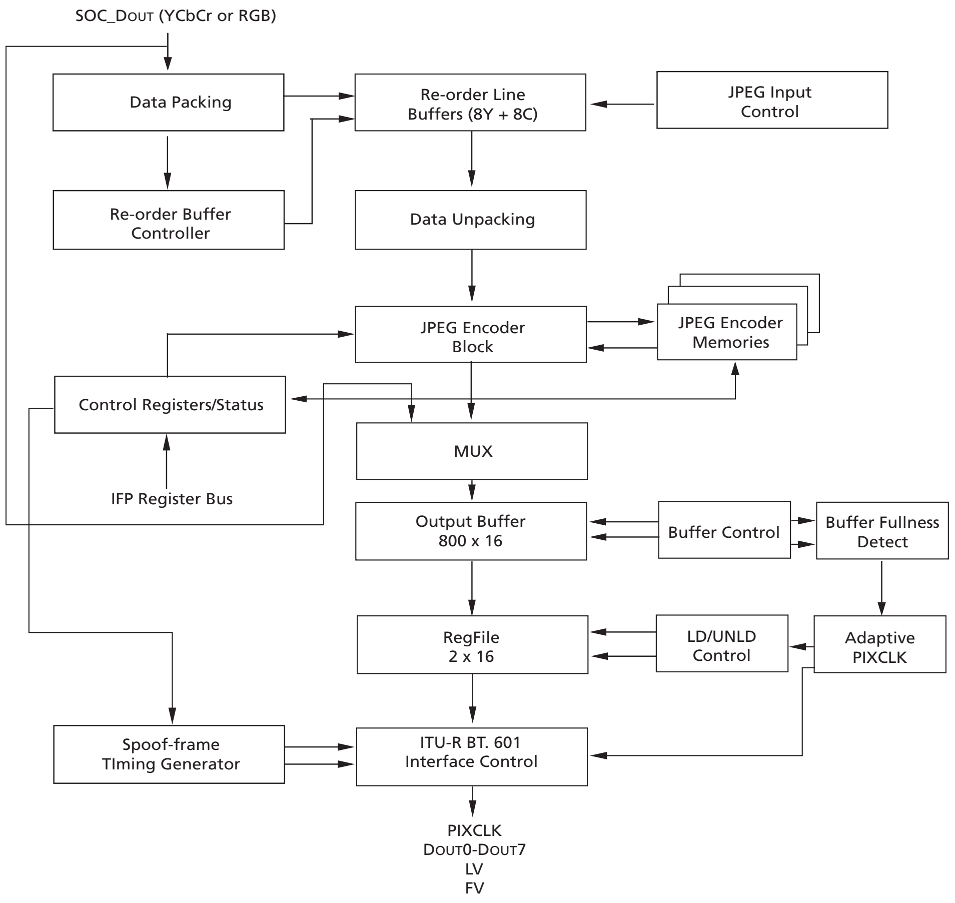
- One each for luminance and chrominance (active)
- Support for three pairs of quantization tables—two pairs serve as a backup for buffer overflow

4. Programmable Huffman Tables

- 2 AC, 2 DC tables—separate for luminance and chrominance

5. Quality/compression ratio control capability
6. 15 fps MJPEG capability (header processing in external host processor)

Figure 5: JPEG Encoder Block Diagram



## Output Buffer Overflow Prevention

The MT9D111 integrates SRAM for the storage of JPEG data. In order to prevent output buffer overflow, the MT9D111 implements an adaptive pixel clock (PIXCLK) rate scheme. When the adaptive pixel clock rate scheme is enabled, PIXCLK can run at clock frequencies of (CLKIN freq/n1), (CLKIN freq/n2), (CLKIN freq/n3), where n1, n2, n3 are register values programmed by the host via the two-wire serial interface. A clock divider block from the master clock CLKIN generates the three clocks, PCLK1, PCLK2, and PCLK3.

At the start of the frame encode, PIXCLK is sourced by PCLK1. The buffer fullness detection block of the SOC switches PIXCLK to PCLK2 and then to PCLK3, if necessary, based on the watermark at the output buffer (i.e., percentage filled up). When the output buffer watermark reaches 50 percent, PIXCLK switches to PCLK2. This increase in PIXCLK rate unloads the output buffer at a higher rate. However, depending on the image complexity and quantization table setting, the compressed image data may still be generated by the JPEG encoder faster than PIXCLK can unload it. Should the output buffer watermark equal 75 percent or higher, PIXCLK is switched to PCLK3. When the output buffer watermark drops back to 50 percent, PIXCLK is switched back to PCLK2. When the output buffer watermark drops to 25 percent, PIXCLK is switched to PCLK1.

When a decision to adapt PIXCLK frequency is made, LINE_VALID, which qualifies the 8-bit data output (DOUT), is de-asserted until PIXCLK is safely switched to the new clock. LINE_VALID is independent of the horizontal timing of the uncompressed imaged. Its assertion is strictly based on compressed image data availability.

Should an output buffer overflow still occur with PIXCLK at the maximum frequency, the output buffer and the small asynchronous FIFO is flushed immediately. This causes LINE_VALID to be de-asserted. FRAME_VALID is also de-asserted.

In addition to the adaptive PIXCLK rate scheme, the MT9D111 also has storage for 3 sets of quantization tables (6 tables). In the event of output buffer overflow during the compression of the current frame, another set of the preloaded quantization tables can be used for the encoding of the immediate next frame. Then, the MT9D111 starts compressing the next frame starting with the nominal PIXCLK frequency.

## Output Interface

### Control (Two-Wire Serial Interface)

Camera control and JPEG configuration/control are accomplished via a two-wire serial interface. The interface supports individual access to all camera function registers and JPEG control registers. In particular, all tables located in the JPEG quantization and Huffman memories are accessible via the two-wire interface. To write to a particular register, the external host processor must send the MT9D111 device address (selected by SADDR or R0x0D:0[10]), the address of the register, and data to be written to it. [1] for a description of read sequence and for details of the two-wire serial interface protocol.

### Data

JPEG data is output in a BT656-like 8-bit parallel bus DOUT0-DOUT7, with FRAME_VALID, LINE_VALID, and PIXCLK. JPEG output data is valid when both FRAME_VALID and LINE_VALID are asserted. When the JPEG data output for the frame completes, or buffer overflow occurs, LINE_VALID and FRAME_VALID are de-asserted. The output clock runs at frequencies selected by frequency divisors N1, N2, and N3 (registers R0x0E:2 and R0x0F:2), depending on output buffer fullness.

1.

## Auto Focus

### Algorithm

The auto focus (AF) algorithm implemented in the MT9D111 firmware seeks to maximize sharpness of vertical lines in images output by the sensor by guiding an external lens actuator to the position of best lens focus. The algorithm is actuator-independent: it provides guidance by means of an abstract one dimensional position variable, leaving the translation of its changes into physical lens movements to a separate AF mechanics (AFM) driver. The AF algorithm relies on the AFM driver to generate digital output signals needed to move different lens actuators and to correctly indicate at all times if the lens is stationary or moving. The latter is required to prevent the AF algorithm from using line sharpness measurements distorted by concurrent lens motion.

For measuring line sharpness, the AF algorithm relies on focus measurement engine in the color pipeline, which is a programmable vertical-edge-filtering module. The module convolves two pre-programmed 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 in addition divided into up to 16 equal-size 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 (R[77:84]:2 and R[87:94]:2), as are Y averages from all the AF windows (in R[67:74]:2). The AF algorithm reduces these data to one normalized sharpness score per AF window, by calculating for each window the ratio (S1+S2)/<Y>, where <Y> is the average Y and S1 and S2 are the raw sharpness scores from the two filters multiplied by 128. Programming 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 algorithm permits unequal weighting of the windows, but window weights are not included in the normalized sharpness scores, for a reason that will soon become clear.

There are several motion sequences through which the MT9D111 AF algorithm can bring a lens to best focus position. 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 AF algorithm stops the lens at each preselected position for long enough to obtain valid sharpness scores. The first normalized score from each AF window 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 from one position to the next 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 at the same time 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 and therefore its observed extrema are likely determined by random noise. On the other hand, if the score varies a lot with the lens position, its maximum is much more likely to be valid, i.e. close to the true sharpness maximum for the corresponding AF window. Due to these considerations, the AF algorithm implemented in MT9D111 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 build 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, i.e. 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 algorithm arbitrarily chooses the start position as the best.

What happens after the first scan is user-programmable—the AF algorithm gives the user a number of ways to proceed with final lens positioning. The user should select a way 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 and either jump or retrace the steps of the first scan to the best scanned position. 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 voted best after the first scan. This second scan is done in the same way as the first, except that the positions it covers are not pre-selected. Instead, the AF algorithm user must set step size and number of steps for the second scan. The second scan must be followed by the same hysteresis-matching motion sequence as the first scan, e.g. a third flyback and jump to the best position.

### Modes

There are four AF camera modes that the MT9D111 can fully support if it controls the position of the camera lens.

1. Snapshot mode
   In this mode, a camera performs auto focusing upon a user command to do so. When the auto focusing is finished, a snapshot is normally taken and there is no further AF activity until next appropriate user command. The MT9D111 can do the auto focusing using its own AF algorithm described above or a substitute algorithm loaded into its RAM. It can then wait or automatically proceed with other operations required to take a snapshot.
2. Locked mode
   The MT9D111 can be commanded to lock the lens in its current position. Between the command to lock the lens and another to release it, the lens does not respond to other commands or scene changes.
3. Focus-free mode
   In many situations, e.g. under low light or during video recording, it may be impossible or undesirable to focus the lens prior to every image capture. Instead, the lens can be locked in a position most likely to produce satisfactory images, e.g. the hyperfocal position. This position can be programmed into the MT9D111, and it can move and hold the lens there on command.

4. Manual mode
   In this mode there is no AF activity—focusing the camera is left to its user. The user typically can move the camera lens in steps, by manually issuing commands to the lens actuator, and observe the effect of his actions on a preview display. The MT9D111 can provide 30 fps image input for the display and simultaneously translate user commands received via two-wire serial interface into digital waveforms driving the lens actuator.

## Lens Actuator Interface

Actuators used to move lenses in AF cameras can be classified into several broad categories that differ significantly in their requirements for driving signals. These requirements also vary from one device to another within each category. To ensure its compatibility with many different actuators, the MT9D111 includes a general purpose input/output module (GPIO).

In essence, the GPIO is a programmable rectangular waveform generator, with 12 individually controllable output pads (GPIO0 through GPIO11), a separate power supply pad ($V_{DD}$GPIO), and a separate clock domain that can be disconnected from the master clock to save power when the GPIO is not in use. The GPIO can toggle its output pads as fast as half the master clock frequency (every 25ns at 80 MHz).

An external host processor and the embedded microcontroller (MCU) of the MT9D111 have two ways to control the voltages on the GPIO output pads:

1. Setting or clearing bits in a control register
   The state of the GPIO pads is updated immediately after writing to the register is finished. Since writing via the two-wire serial interface takes some time, this way does not give the host processor a very precise control over GPIO output timing.
2. Waveform programming
   The second way to obtain a desired output from the GPIO is to program into its registers a set of periodic waveforms and initialize their generation. The GPIO then generates the programmed waveforms on its own, without waiting for any further input, and therefore with the best attainable timing precision. If necessary, the GPIO can notify the MCU and the host processor about reaching certain points in the waveforms generation, e.g., the end of a particular waveform. Every GPIO notification has two components: the GPIO sends a wakeup signal to the MCU and sets a bit in its status register that can be polled by the MCU and/or the host processor. The wakeup signals have an effect only when the MCU is in sleep mode.

The MT9D111 can be set up not only to output digital signals to a lens actuator and/or other similar devices, but also to receive their digital and analog feedback. All GPIO output pads are reconfigurable as high-impedance digital inputs. The logical state of each GPIO pad is mirrored by the state of a bit in a dedicated register, which allows the MCU and host processor to sample digital input signals at intervals equal to their respective register read times. Analog input signals (0.1V to 1.0V) can be sampled using one of the 10-bit ADCs in the sensor core. During horizontal blanking periods, when it does not digitize the sensor signal, this ADC samples voltages on AIN1, AIN2, and AIN3 input pads. The results are stored in dedicated registers. The maximum signal sampling rate permitted by this scheme is about 36000 samples per second.

### Context and Operational Modes

The MT9D111 can operate in several modes, including preview, still capture (snapshot), and video. All modes of operation are individually configurable and are organized as two contexts—context A and context B. A context is defined by sensor image size, frame rate, resolution and other associated parameters. The user can switch between the two contexts by sending a command via the two-wire serial interface.

#### Preview

Context A is primarily intended for use in the preview mode. During preview, the sensor usually outputs low resolution images at a relatively high frame rate, and its power consumption is kept to a minimum. Context B can be configured for the still capture or video mode, as required by the user. For still capture configuration, the user typically specifies the desired output image size, if JPEG compression and flash should be enabled, how many frames to capture, etc. For video, the user might select a different image size and a fixed frame rate.

#### Snapshot and Flash

To take a snapshot, the user must send a command that changes the context from A to context B. Typical sequence of events after this command is as follows. First, the camera may turn on its LED flash, if it has one and is required to use it. With the flash on, the camera exposure and white balance is automatically adjusted to the changed illumination of the scene. Next, the camera performs auto focusing. Once in focus, it enables JPEG compression and capture one or more frames of desired size. A camera equipped with a Xenon flash strobes it during the capture. Completing the sequence, the camera automatically returns to context A and resume running preview.

#### Video

To start video capture, the user has to change relevant context B settings, such as capture mode, image size and frame rate, and again send a context change command. Upon receiving it, the MT9D111 switches to the modified context B settings, while continuing to output YUV-encoded image data. Auto exposure and auto focus automatically switches to smooth continuous operation. To exit the video capture mode, the user has to send another context change command causing the sensor to switch back to context A.

## Auto Exposure

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

Two auto exposure algorithm modes are available:

1. preview
2. scene evaluative

Auto exposure is implemented by means of a firmware driver that analyzes image statistics collected by exposure measurement engine, makes a decision and programs the sensor core and color pipeline to achieve the desired exposure. The measurement engine subdivides the image into 16 windows organized as a 4 x 4 grid.

### Preview Mode

This exposure mode is activated during preview or video capture. It relies on the exposure measurement engine that tracks speed and amplitude of the change of the overall luminance in the selected windows of the image.

The backlight compensation is achieved by weighting 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 speed of response, and control of the sensitivity to the small changes. While the default settings are adequate in most situations, the user can program target brightness, measurement window, and other parameters described above.

### Scene Evaluative Algorithm

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 most difference for backlight and bright outdoor conditions.

## Auto White Balance

The MT9D111 has a built-in auto white balance (AWB) algorithm designed to compensate for the effects of 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 and a driver performing the selection of the optimal color correction matrix, digital, and sensor core analog gains. While default settings of these algorithms are adequate in most situations, the user can re-program base color correction matrices, place limits on color channel gains, and control the speed of both matrix and gain adjustments. Unlike simple white balancing algorithms found in many PC cameras, the MT9D111 AWB does not require the presence of gray or white elements in the image for good color rendition. The AWB does not attempt to locate "brightest" or "grayest" element of the image but instead performs sophisticated image analysis to differentiate between changes in predominant spectra of illumination and changes in predominant colors of the scene. While defaults are suitable for most applications, a wide range of algorithm parameters can be overwritten by the user via the serial interface.

## Flicker Detection

Flicker occurs when the integration time is not an integer multiple of the period of the light intensity. The automatic flicker detection block does not compensate for the flicker, but rather avoids it by detecting the flicker frequency and adjusting the integration time. For integration times below the light intensity period (10ms for 50Hz environment), flicker cannot be avoided.

## Registers and Variables

Four types of configuration controls are available:

1. Hardware registers
2. Driver variables
3. Special function registers (SFR)
4. MCU SRAM

The following convention is used in the text below to designate registers and variables:

R0x12:1, R0x12:1[3:0] or R18:1, R18:1[3:0]

These refer to two-wire accessible register number 18, or 0x12 hexadecimal, located on page 1. [3:0] indicate bits. Registers numbers range 0...255 and bits range 15...0.

- ae.Target
  This refers to variable 'Target" in the AE driver.
- SFR 0x1080 or SRAM 0x0400
  This refers to special function register or SRAM located at address 0x1080 in MCU memory space.

## How to Access

Registers, variables, and SFRs are accessed in different ways.

### Registers

Hardware registers are organized into several pages. Page 0 contains sensor controls. Page 1 contains color pipeline controls. Page 2 contains JPEG, output FIFO and more color pipeline controls. The desired page is selected by writing the desired value to R0xF0. After that all READs and WRITEs to registers 0...255 except R0xF0 and R0xF1, is directed to the selected page. R0xF0 and R0xF1 are special registers and are present on all pages. [1] for description of two-wire register access.

### Variables

Variables are located in the microcontroller RAM memory. Each driver, such as auto exposure, white balance, auto focus, etc., has a unique driver ID (0...31) and a set of public variables organized as a structure. Each variable in this structure is uniquely identified by its offset from the top of the structure and its size. The size can be 1 or 2 bytes, while the offset is 1 byte.

All driver variables (public and private) can be accessed via R198:1 and R200:1. While two access modes are available-access by physical address and by logical address, the public variables are typically accessed by the logical method. The logical address, which is set in R198:1, consists of a 5-bit driver ID number and a variable offset. Examples are provided below.

To set the variable ae.Target=50:

- The variable has a driver ID of 2. Therefore, set R198:1[12:8]=2
- The variable has an offset of 6. Therefore, set R198:1[7:0]=6
- This is a logical access. Therefore, set R198:1[14:13]=01
- The size of the variable is 8 bits. Therefore, set R198:1[15]=1
- By combining these bits, R198:1=0xA206.
- Set R200:1=50 for the value of the variable

1.

To read the variable ae.Target:
- Since this is the same variable as the above example, R198:1=0xA206
- Read R200:1 for the current variable value

To set the variable mon.arg1=0x1234:
- The variable has a driver ID of 0. Therefore, set R198:1[12:8]=0
- The variable has an offset of 3. Therefore, set R198:1[7:0]=3
- This is a logical access. Therefore, set R198:1[14:13]=01
- The size of the variable is 16 bits. Therefore, set R198:1[15]=0
- By combining these bits, R198:1=0x2003
- Set R200:1=0x1234 for the value of the variable

To read the variable mon.arg1:
- Since this is the same variable as the above example, R198:1=0x2003
- Read R200:1 for the current variable value

Please see description for R198:1 and R200:1 in the MT9D111 register reference for more details.

## Special Function Registers (SFR) and MCU SRAM

Special function registers (SFR) are registers connected to the local bus of the micro-controller. These registers include GPIO, waveform generator, and those important for firmware operation. SFR are accessed by physical address. MCU SRAM consists of 1K system memory and 1K user memory. Examples of access:
- Write into user SRAM. Use to upload code
  - a. R198:1 = 0x400 // address
  - b. R200:1 = 0x1234 // write 16-bit value
- Read from user SRAM
  - c. R198:1 = 0x400 // address
  - d. Read R200:1 // read 16-bit value
- Write to 8-bit GPIO register
  - e. R198:1 = 0x9079 // GPIO_DIR_L at 0x1079
  - f. R200:1 = 0x00FE // Configure GPIO[0] as output
- Read from 8-bit GPIO register
  - g. R198:1 = 0x9079 // GPIO_DIR_L at 0x1079
  - h. Read R200:1 //Check GPIO[7:0] pad state

See R198:1 and R200:1 description in the MT9D111 register reference for more detail.

## Registers

### Sensor Core Registers

Table 3: Sensor Core Register Defaults

| ADDR | Register Description | Default PRE MCU BOOT | Default AFTER MCU BOOT |
|---|---|---|---|
| 0x00 | Reserved | 0x1519 | 0x1519 |
| 0x01 | Row Start | 0x001C | 0x001C |
| 0x02 | Column Start | 0x003C | 0x003C |
| 0x03 | Row Width | 0x04B0 | 0x04B0 |
| 0x04 | Col Width | 0x0640 | 0x0640 |
| 0x05 | Horizontal Blanking B | 0x015C | 0x0204 |
| 0x06 | Vertical Blanking B | 0x0020 | 0x002F |
| 0x07 | Horizontal Blanking A | 0x00AE | 0x00FE |
| 0x08 | Vertical Blanking A | 0x0010 | 0x000C |
| 0x09 | Shutter Width | 0x04D0 | N/A |
| 0x0A | Row Speed | 0x00011 | 0x0001 |
| 0x0B | Extra Delay | 0x0000 | 0x0000 |
| 0x0C | Shutter Delay | 0x0000 | 0x0000 |
| 0x0D | Reset | 0x0000 | 0x0000 |
| 0x1F | Frame Valid Control | 0x0000 | 0x0000 |
| 0x20 | Read Mode B | 0x0000 | 0x0300 |
| 0x21 | Read Mode A | 0x0490 | 0x8400 |
| 0x22 | Dark Col/Rows | 0x010F | 0x010F |
| 0x23 | Flash | 0x0608 | 0x0608 |
| 0x24 | Extra Reset | 0x8000 | 0x8000 |
| 0x25 | Line Valid Control | 0x0000 | 0x0000 |
| 0x26 | Bottom Dark Rows | 0x0007 | 0x0007 |
| 0x2B | Green1 Gain | 0x0020 | N/A |
| 0x2C | Blue Gain | 0x0020 | N/A |
| 0x2D | Red Gain | 0x0020 | N/A |
| 0x2E | Green2 Gain | 0x0020 | N/A |
| 0x2F | Global Gain | 0x0020 | N/A |
| 0x30 | Row Noise | 0x042A | 0x042A |
| 0x59 | Black Rows | 0x00FF | 0x00FF |
| 0x5B | Dark G1 average | N/A | N/A |
| 0x5C | Dark B average | N/A | N/A |
| 0x5D | Dark R average | N/A | N/A |
| 0x5E | Dark G2 average | N/A | N/A |
| 0x5F | Calib Threshold | 0x231D | 0x231D |
| 0x60 | Calib Control | 0x0080 | 0x0080 |
| 0x61 | Calib Green1 | 0x0000 | 0x0000 |
| 0x62 | Calib Blue | 0x0000 | 0x0000 |
| 0x63 | Calib Red | 0x0000 | 0x0000 |
| 0x64 | Calib Green2 | 0x0000 | 0x0000 |
| 0x65 | Clock Control | 0xE000 | 0xE000 |
| 0x66 | PLL Control 1 | 0x2809 | 0x1000 |

**Table 3: Sensor Core Register Defaults (continued)**

| ADDR | Register Description | Default PRE MCU BOOT | Default AFTER MCU BOOT |
|---|---|---|---|
| 0x67 | PLL Control 2 | 0x0501 | 0x0500 |
| 0xC0 | Global Shutter Control | 0x0000 | 0x0000 |
| 0xC1 | Start Integration (T1) | 0x0064 | 0x0064 |
| 0xC2 | Start Readout (T2) | 0x0064 | 0x0064 |
| 0xC3 | Assert Strobe (T3) | 0x0096 | 0x0096 |
| 0xC4 | De-assert Strobe (T4) | 0x00C8 | 0x00C8 |
| 0xC5 | Assert Flash | 0x0064 | 0x0064 |
| 0xC6 | De-assert Flash | 0x0078 | 0x0078 |
| 0xE0 | External Sample 1 | 0x0000 | 0x0000 |
| 0xE1 | External Sample 2 | 0x0000 | 0x0000 |
| 0xE2 | External Sample 3 | 0x0000 | 0x0000 |
| 0xE3 | External Sampling Control | 0x0000 | 0x0000 |
| 0xF0 | Page Register | 0x0000 | 0x0000 |
| 0xF1 | Bytewise Address | 0x0000 | 0x0000 |
| 0xF2 | Context Control | 0x000B | 0x000B |
| 0xFF | Reserved | 0x1519 | 0x1519 |

## Output Format and Timing

### YUV/RGB Uncompressed Output

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**

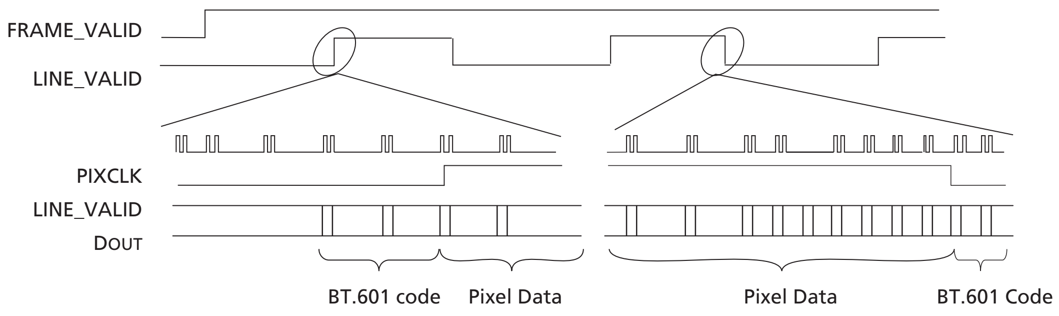
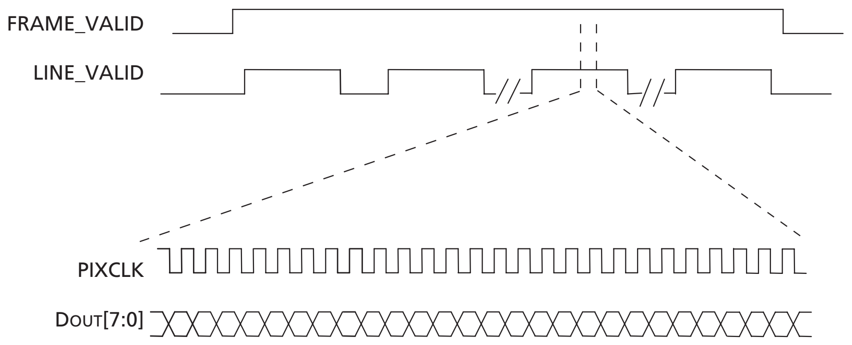


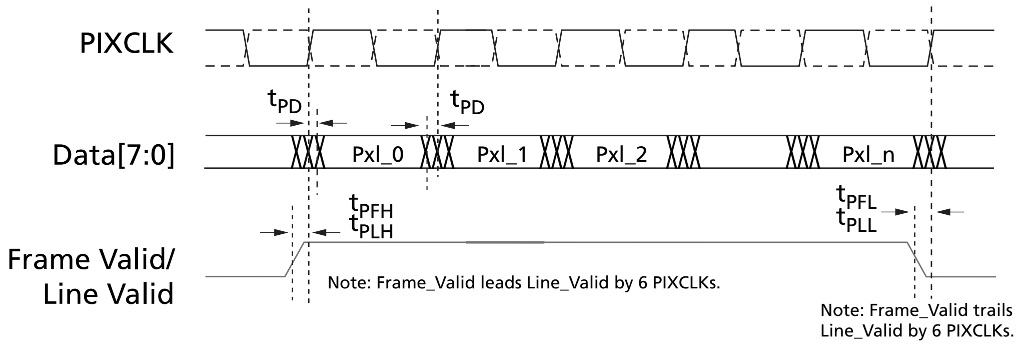
Figure 7 depicts the output timing of uncompressed YUV/RGB when a decimated data stream is equalized by buffering or when no decimation takes place. The pixel clock frequency remains constant during each LINE_VALID high period. Decimated data are output at a lower frequency than full size frames, which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

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



Timing details of uncompressed YUV/RGB output are shown in Figure 8 on page 29.

**Figure 8: Details of Uncompressed YUV/RGB Output Timing**



| Symbol | Definition | Conditions | MIN | MAX | Units |
|---|---|---|---|---|---|
| $f_{PIXCLK}$ | PIXCLK frequency | Default | | 80 | MHz |
| $t_{PD}$ | PIXCLK to data valid | Default | -3 | 3 | ns |
| $t_{PFH}$ | PIXCLK to FV high | Default | -3 | 3 | ns |
| $t_{PLH}$ | PIXCLK to LV high | Default | -3 | 3 | ns |
| $t_{PFL}$ | PIXCLK to FV low | Default | -3 | 3 | ns |
| $t_{PLL}$ | PIXCLK to LV low | Default | -3 | 3 | ns |

## Uncompressed YUV/RGB Data Ordering

The MT9D111 supports swapping YCrCb mode, as illustrated in Table 4.

**Table 4: 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 5. 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 5: 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 | $0\ R_7R_6R_5R_4R_3G_7G_6$ |
| | Even | $G_4G_3G_2B_7B_6B_5B_4B_3$ |
| RGB 444x | Odd | $R_7R_6R_5R_4G_7G_6G_5G_4$ |
| | Even | $B_7B_6B_5B_4\ 0\ 0\ 0\ 0$ |
| RGB x444 | Odd | $0\ 0\ 0\ 0\ R_7R6_6R_5R_4$ |
| | Even | $G_7G_6G_5G_4B_7B_6B_5B_4$ |

## Uncompressed 10-Bit Bypass Output

Raw 10-bit Bayer data from the sensor core can be output in bypass mode in two ways:

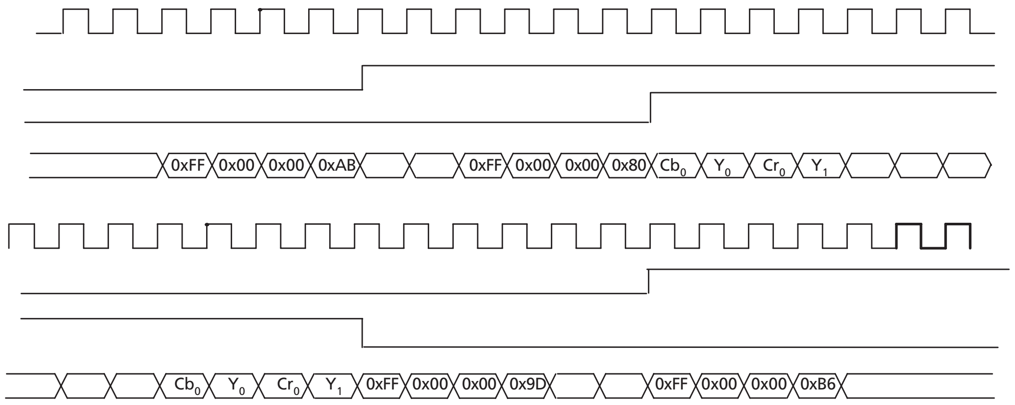
1. Using 10 data output pads (DOUT0–DOUT9), or
2. Using only 8 pads (DOUT0–DOUT7) and a special 8 + 2 data format, shown in Table 6.

The timing of 10-bit or 8-bit data stream output in the bypass mode is qualitatively the same as that depicted in Figure 7.

**Table 6: 2-Byte RGB Format**

| | | |
|---|---|---|
| Odd bytes | 8 data bits | $D_9D_8D_7D_6D_5D_4D_3D_2$ |
| Even bytes | 2 data bits + 6 unused bits | 0 0 0 0 0 0 $D_1D_0$ |

**Figure 9: Example of Timing for Non-Decimated Uncompressed Output Bypassing Output FIFO**



## JPEG Compressed Output

JPEG compression of IFP output produces a data stream whose structure 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 10 on page 32, Figure 11 on page 32, and Figure 12 on page 32. These timing diagrams are merely three typical examples of many variations of JPEG output. Description of output configuration register R13:2 in Table 6, “IFP Registers, Page 2,” on page 49 provides more information on different output interface configuration options.

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 output 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 spoof 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 registers R14:2 and R15:2. 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 10: Timing of JPEG Compressed Output in Free-Running Clock Mode**

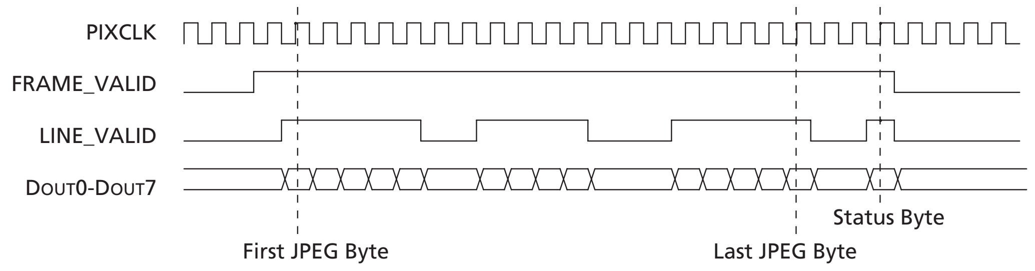


**Figure 11: Timing of JPEG Compressed Output in Gated Clock Mode**

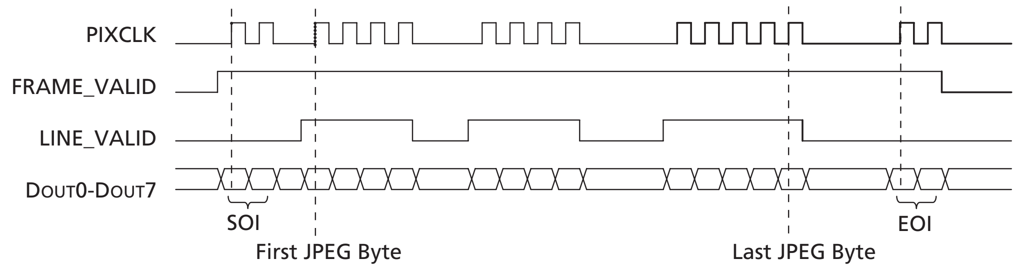
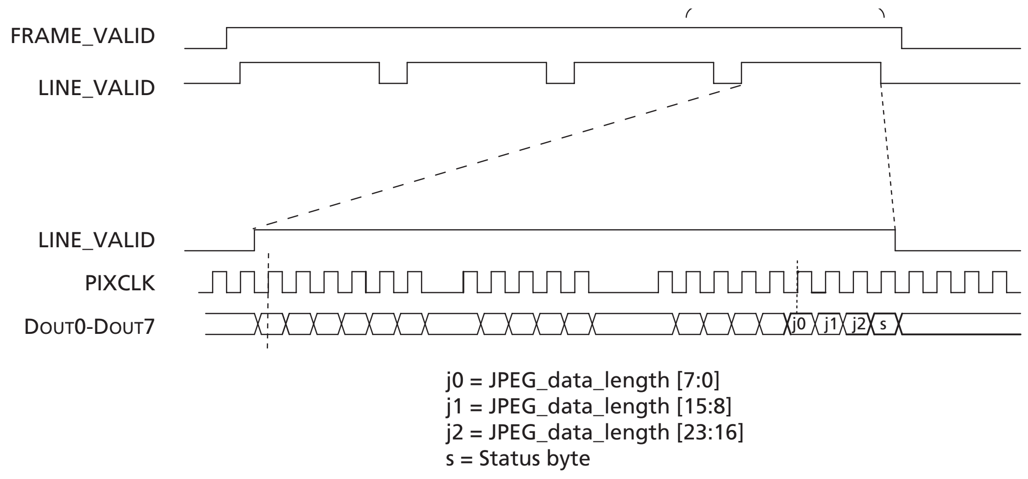


**Figure 12: Timing of JPEG Compressed Output in Spoof Mode**


FRAME_VALID
LINE_VALID
LINE_VALID
PIXCLK
DOUT0-DOUT7
j0
j1
j2
s
j0 = JPEG_data_length [7:0]
j1 = JPEG_data_length [15:8]
j2 = JPEG_data_length [23:16]
s = Status byte

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.

## Color Conversion Formulas

### Y'Cb'Cr' ITU-R BT.601

A widely known color conversion standard. Note that $16 < Y_{601} < 235$ and $16 < Cb, Cr < 240$. $0 <= RGB <= 255$.

$Y_{601}' = Y'*219/256 + 16$

$Cr' = 0.713 (R' - Y')*224/256 + 128$

$Cb' = 0.564 (B' - Y')*224/256 + 128$

where $Y' = 0.299 R' + 0.587 G' + 0.114 B'$

The reverse formulas to convert YCbCr into RGB, $0 <= RGB <= 255$:

$R' = 1.164(Y_{601}' - 16) + 1.596(Cr' - 128)$

$G' = 1.164(Y_{601}' - 16) - 0.813(Cr' - 128) - 0.391(Cb' - 128)$

$B' = 1.164(Y_{601}' - 16) + 2.018(Cb' - 128)$

### Y'U'V'

This conversion is BT 601 scaled to make YUV range from 0 through 255. This setting is recommended for JPEG encoding and is the most popular, although it is not well defined and often misused in Windows.

$Y' = 0.299 R' + 0.587 G' + 0.114 B'$

$U' = 0.564 (B' - Y') + 128$

$V' = 0.713 (R' - Y') + 128$

There is an option where 128 is not added to U'V'.

### Y'Cb'Cr Using sRGB Formulas

The MT9D111 implements the sRGB standard. This option provides YCbCr coefficients for a correct 4:2:2 transmission. Note that 16 < Y60 1< 235, 16 < Cb, Cr < 240 and 0 < = RGB < = 255.

$Y' = (0.2126*R' + 0.7152*G' + 0.0722*B')*219/256 + 16$

$Cb' = 0.5389*(B '- Y')*224/256 + 128$

$Cr' = 0.635*(R' - Y')*224/256 + 128$

### Y'U'V' Using sRGB Formulas

Similar to the previous set of formulas, but has YUV spanning a range of 0 through 255.

$Y' = 0.2126*R' + 0.7152*G' + 0.0722*B'$

$U' = 0.5389*(B' - Y') + 128 = -0.1146*R' - 0.3854*G' + 0.5*B + 128$

$V' = 0.635*(R' - Y') + 128 = 0.5*R' - 0.4542*' - 0.0458*B + 128$

There is an option to disable adding 128 to U'V'. The reverse transform is as follows:

$R' = Y + 1.5748*V$

$G' = Y - 0.1873*(U - 128) - 0.4681*(V - 128)$

$B' = Y + 1.8556*(U - 128)$

## Sensor Core

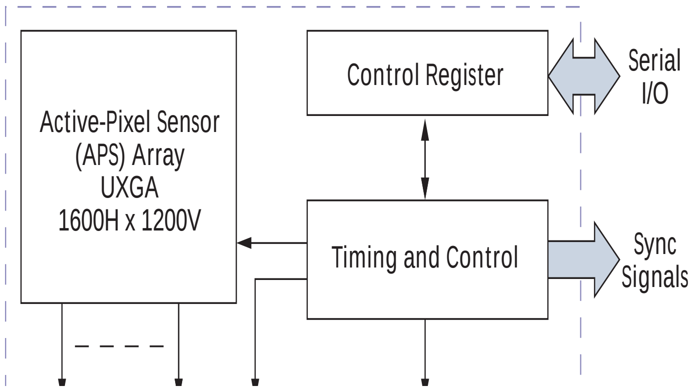
This section describes the sensor core. The core is based entirely on Aptina's MT9D011 sensor.

The SOC firmware controls a key sensor core registers, such as exposure, window size, gains, and contexts. When firmware or MCU are disabled, the sensor core can be programmed directly.

### Introduction

The sensor core is a progressive-scan sensor that generates a stream of pixel data qualified by LINE_VALID and FRAME_VALID signals. An on-chip PLL generates the master clock from an input clock of 6 MHz to 40 MHz. In default mode, the data rate (pixel clock) is the same as the master clock frequency, which means that one pixel is generated every master clock cycle. The sensor block diagram is shown in Figure 13.

**Figure 13: Sensor Core Block Diagram**



The core of the sensor is an active-pixel array. The timing and control circuitry sequences through the rows of the array, resetting and then reading each row. In the time interval between resetting a row and reading that row, the pixels in that row integrate incident light. The exposure is controlled by varying the time interval between reset and readout. After a row is read, the data from the columns is sequenced through an analog signal chain (providing offset correction and gain), and then through an ADC. The output from the ADC is a 10-bit value for each pixel in the array. The pixel array contains optically active and light-shielded "black" pixels. The black pixels are used to provide data for on-chip offset correction algorithms (black level control).

The sensor contains a set of 16-bit control and status registers that can be used to control many aspects of the sensor operations. These registers can be accessed through a two-wire serial interface. In this document, registers are specified either by name (Column Start) or by register address (R0x04:0). Fields within a register are specified by bit or by bit range (R0x20:0[0] or R0x0B:0[13:0]). The control and status registers are described in Table 4, "Sensor Register Description," on page 28.

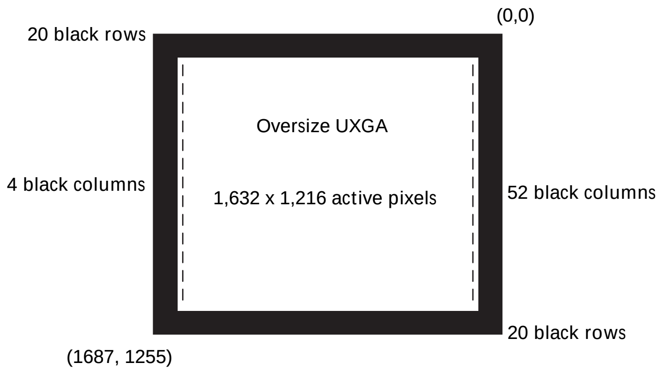
The output from the sensor is a Bayer pattern: alternate rows are a sequence of either green/red pixels or blue/green pixels. The offset and gain stages of the analog signal chain provide per-color control of the pixel data.

## Pixel Array Structure

The sensor core pixel array is configured as 1,688 columns by 1,256 rows (shown in Figure 14). The first 52 columns and the first and the last 20 rows of pixels are optically black and are used for the automatic black level adjustment. The last 4 columns are also optically black.

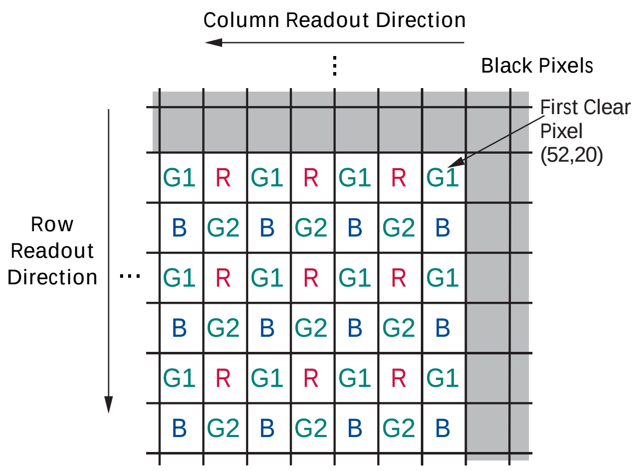
The optically active pixels are used as follows: In default mode a UXGA image (1,600 columns by 1,200 rows) is generated, starting at row 28, column 60. A 4-pixel boundary of active pixels can be enabled around the image to avoid boundary effects during color interpolation and correction. During mirrored readout, the region of active pixels used to generate the image is offset by 1 pixel in each mirrored direction so that the readout always starts on the same color pixel.

**Figure 14: Pixel Array**



The sensor core uses a Bayer color pattern, as shown in Figure 15. The even-numbered rows contain green and red color pixels; odd-numbered rows contain blue and green color pixels. Even-numbered columns contain green and blue color pixels; odd-numbered columns contain red and green color pixels. The color order is preserved during mirrored readout.

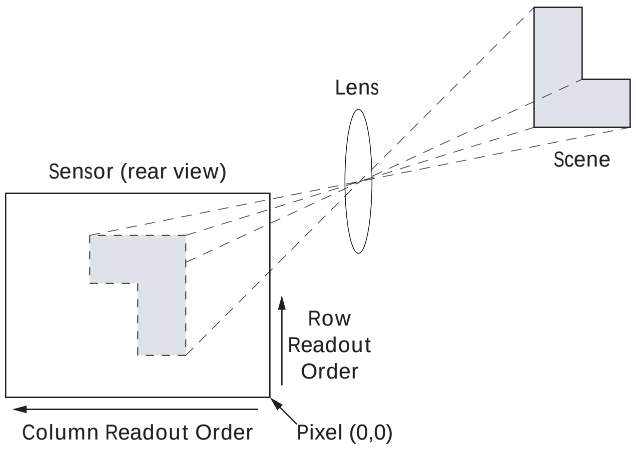
**Figure 15: Pixel Color Pattern Detail (Top Right Corner)**



### Default Readout Order

By convention, the sensor core pixel array is shown with pixel (0,0) in the top right-hand corner (see Figure 15). This reflects the actual layout of the array on the die. When the sensor is imaging, the active surface of the sensor faces the scene as shown in Figure 16. When the image is read out of the sensor, it is read one row at a time, with the rows and columns sequenced as shown in Figure 14. By convention, data from the sensor is shown with the first pixel read out—pixel (52,20) in the case of the sensor core—in the top left-hand corner.

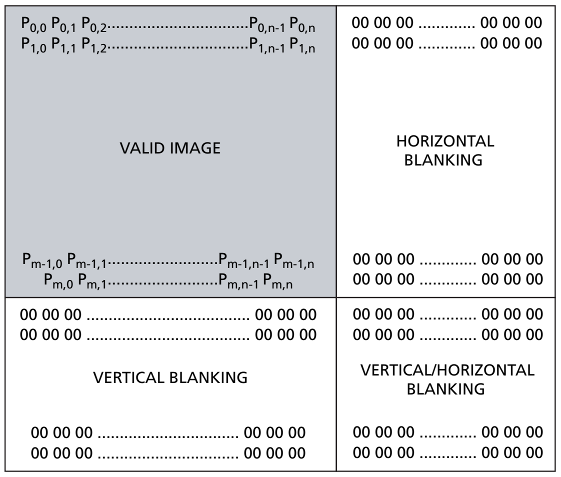
**Figure 16: Imaging a Scene**



## Raw Data Format

The sensor core image data is read out in a progressive scan. Valid image data is surrounded by horizontal blanking and vertical blanking as shown in Figure 17. The amount of horizontal blanking and vertical blanking is programmable. LINE_VALID is HIGH during the shaded region of the figure. FRAME_VALID timing is described in the next section.

**Figure 17: Spatial Illustration of Image Readout**


$P_{0,0}$ $P_{0,1}$ $P_{0,2}$.....$P_{0,n-1}$ $P_{0,n}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$.....$P_{1,n-1}$ $P_{1,n}$
VALID IMAGE
$P_{m-1,0}$ $P_{m-1,1}$.....$P_{m-1,n-1}$ $P_{m-1,n}$
$P_{m,0}$ $P_{m,1}$.....$P_{m,n-1}$ $P_{m,n}$
00 00 00 ............. 00 00 00
HORIZONTAL BLANKING
VERTICAL BLANKING
VERTICAL/HORIZONTAL BLANKING

## Raw Data Timing

The sensor core output data is synchronized with the PIXCLK output. When LINE_VALID is HIGH, one pixel datum is output on the 10-bit DOUT output every PIXCLK period. By default, the PIXCLK signal runs at the same frequency as the master clock, and its rising edges occur one-half of a master clock period after transitions on LINE_VALID, FRAME_VALID, and DOUT (see Figure 18). This allows PIXCLK to be used as a clock to sample the data. PIXCLK is continuously enabled, even during the blanking period. The sensor core can be programmed to delay the PIXCLK edge relative to the DOUT transitions from 0 to 3.5 master clocks, in steps of one-half of a master clock. This can be achieved by programming the corresponding bits in R0x0A:0. The parameters P, A, and Q in Figure 19 are defined in Table 7 on page 39.

**Figure 18: Pixel Data Timing Example**

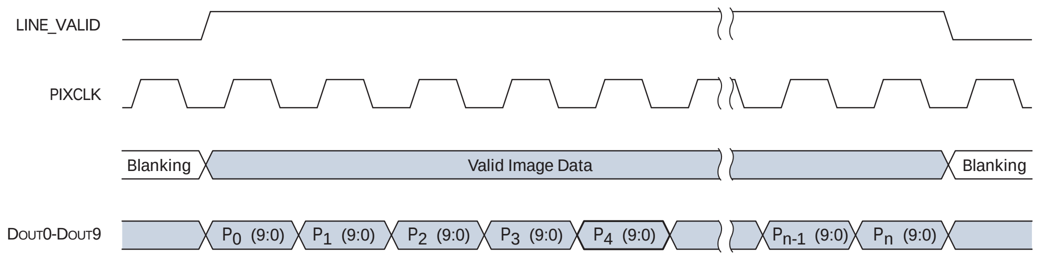
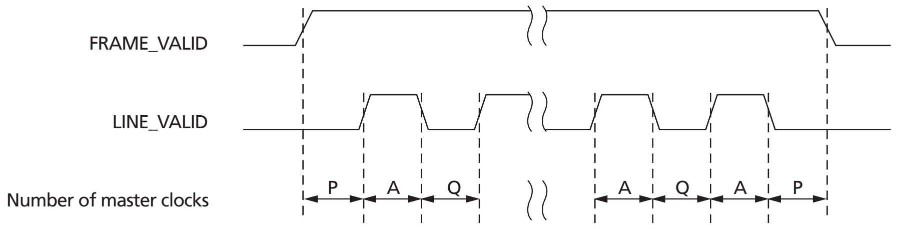


**Figure 19: Row Timing and FRAME_VALID/LINE_VALID Signals**



The sensor timing is shown in terms of pixel clock and master clock cycles (see Figure 18 on page 38). The recommended master clock frequency is 36 MHz. Increasing the integration time to more than one frame causes the frame time to be extended. The equations in Table  assume integration time is less than the number of rows in a frame (R0x09:0 < R0x03:0/S + BORDER + VBLANK_REG). If this is not the case, the number of integration rows must be used instead to determine the frame time, as shown in Table 8.

**Table 7: Frame Time**

| Parameter | Name | Equation | Default Timing at 36 MHz Dual ADC Mode |
|---|---|---|---|
| HBLANK_REG | Horizontal Blanking Register | R0x07:0 if R0xF2:0[0] = 0<br>R0x05:0 if R0xF2:0[0] = 1 | 0x15C = 348 pixels |
| VBLANK_REG | Vertical Blanking Register | R0x8:0 if R0xF2:0[1] = 0<br>R0x6:0 if R0xF2:0[1] = 1 | 0x20 = 32 rows |
| ADC_MODE | ADC mode | R0xF2:0[3] = 0: R0x20:0[10]<br>R0xF2:0[3] = 1: R0x21:0[10] | |
| PIXCLK_PERIOD | Pixel clock period | ADC_MODE = 0: R0x0A:0[2:0]<br>ADC_MODE = 1: R0x0A:0[2:0]*2 | 1 ADC_MODE: 55.556ns<br>2 ADC_MODE: 27.778ns |
| S | Skip Factor | For skip 2x mode: S = 2<br>For skip 4x mode: S = 4<br>For skip 8x mode: S = 8<br>For skip 16x mode: S = 16<br>otherwise, S = 1 | 1 |
| A | Active Data Time | (R0x04:0/S) * PIXCLK_PERIOD | 1,600 pixel clocks<br>= 1,600 master<br>= 44.44µs |
| P | Frame Start/End Blanking | 6 * PIXCLK_PERIOD (can be controlled by R0x1F:0) | 6 pixel clocks<br>= 12 master<br>= 0.166µs |
| Q | Horizontal Blanking | HBLANK_REG * PIXCLK_PERIOD | 348 pixel clocks<br>= 348 master<br>= 9.667µs |
| A + Q | RowTime | ((R0x04:0/S) + HBLANK_REG) * PIXCLK_PERIOD | 1,948 pixel clocks<br>= 1,948 master<br>= 54.112µs |
| V | Vertical Blanking | VBLANK_REG * (A + Q) + (Q - 2*P) | 62,672 pixel clocks<br>= 62,672 master<br>= 1.741ms |
| Nrows * (A + Q) | Frame Valid Time | (R0x03:0/S) * (A + Q) - (Q - 2*P) | 2,337,264 pixel clocks<br>= 2,337,264 master<br>= 64.925ms |
| F | Total Frame Time | ((R0x03:0/S) + VBLANK_REG) * (A + Q) | 2,399,936 pixel clocks<br>= 2,399,936 master<br>= 66.665ms |

Note: Skip factor should be multiplied by 2 if binning is enabled.

**Table 8: Frame—Long Integration Time**

| Parameter | Name | Equation (Master Clock) |
|---|---|---|
| V’ | Vertical Blanking (long integration time) | (R0x09:0 – (R0x03:0)/S) * (A + Q) + (Q - 2*P) |
| F’ | Total Frame Time (long integration time) | (R0x09:0) * (A + Q) |

## Registers

Table 4, “Sensor Register Description,” on page 28 provides a detailed description of the registers. Bit fields that are not identified in the table are read only.

### Double-Buffered Registers

Some sensor settings cannot be changed during frame readout. For example, changing row width R0x03:0 part way through frame readout results in inconsistent LINE_VALID behavior. To avoid this, the sensor core double buffers many registers by implementing a “pending” and a “live” version. READs and WRITEs access the pending register. The live register controls the sensor operation.

The value in the pending register is transferred to a live register at a fixed point in the frame timing, called “frame start.” Frame start is defined as the point at which the first dark row is read out. By default, this occurs 10 row times before FRAME_VALID goes HIGH. R0x22:0 enables the dark rows to be shown in the image, but this has no effect on the position of frame start.

To determine which registers or register fields are double-buffered in this way, see Table 4, “Sensor Register Description,” on page 28, the “sync’d-to-frame-start” column.

R0x0D:0[15] can be used to inhibit transfers from the pending to the live registers. This control bit should be used when making many register changes that must take effect simultaneously.

### Bad Frames

A bad frame is a frame where all rows do not have the same integration time, or where offsets to the pixel values changed during the frame.

Many changes to the sensor register settings can cause a bad frame. For example, when row width R0x03:0 is changed, the new register value does not affect sensor behavior until the next frame start. However, the frame that would be read out at that frame-start has been integrated using the old row width. Consequently, reading it out using the new row width results in a frame with an incorrect integration time.

By default, most bad frames are masked: LINE_VALID and FRAME_VALID are inhibited for these frames so that the vertical blanking time between frames is extended by the frame time.

To determine which register or register field changes can produce a bad frame, see Table 4, “Sensor Register Description,” on page 28, the “bad frame” column, and these notations:

- N—No. Changing the register value does not produce a bad frame
- Y—Yes. Changing the register value might produce a bad frame
- YM—Yes; but the bad frame is masked out unless the “show bad frames” feature (R0x0D:0[8]) is enabled

### Changes to Integration Time

If the integration time (R0x09:0) is changed while FRAME_VALID is asserted for frame *n*; the first frame output using the new integration time is frame (*n* + *2*). The sequence is as follows:

1. During frame *n*, the new integration time is held in the R0x09:0 pending register.
2. At the start of frame (*n* +*1*), the new integration time is transferred to the R0x09:0 live register.

Integration for each row of frame ($n + 1$) has been completed using the old integration time. The earliest time that a row can start integrating using the new integration time is immediately after that row has been read for frame ($n + 1$). The actual time that rows start integrating using the new integration time is dependent on the new value of the integration time.

3. When frame ($n + 1$) is read out, it is integrated using the new integration time.

If the integration time is changed (R0x09:0 written) on successive frames, each value written is applied to a single frame; the latency between writing a value and it affecting the frame readout remains at two frames.

### Changes to Gain Settings

When the gain settings (R0x2B:0, R0x2C:0, R0x2D:0, R0x2E:0, and R0x2F:0) are changed, the gain is usually updated on the next frame start. When the integration time and the gain are changed simultaneously, the gain update is held off by one frame so that the first frame output with the new integration time also has the new gain applied.

## Feature Description

### PLL Generated Master Clock

The PLL embedded in the sensor core can generate a master clock signal whose frequency is up to 80 MHz (input clock from 6 MHz through 64 MHz). Registers R0x66:0 and R0x67:0 control the frequency of the PLL-generated clock. It is possible to bypass the PLL and use CLKIN as master clock. In order to do so, one must set R0x65:0[15] to 1. If power consumption is a concern, R0x65:0[14] should be also set to 1 a short time later, to put the bypassed PLL in power down mode. To enable the PLL again, the two bits must be set to 0 in the reverse order. By default, the PLL is bypassed and powered down.

### 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:

$$f_{PLL}, f_{OUT} = f_{PLL}, f_{IN}, \text{ x } M \text{ / } [2 \text{ x } (N+1) \text{ x } (P+1)]$$

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 9: Frequency Parameters**

| Frequency | Equation | Min (MHz) | Max (MHz) |
|---|---|---|---|
| $f_{PFD}$ | $f_{IN}$/ (N+1) | 2 | 13 |
| $f_{VCO}$ | $f_{PFD}$ x M | 110 | 240 |
| $f_{OUT}$ | $f_{VCO}$ / [2 x (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.

### PLL Power-up

The PLL takes time to power up. During this time, the behavior of its output clock signal is not guaranteed. The PLL is in the power down mode by default and must be turned on manually. When using the PLL, the correct power-up sequence after chip reset is as follows:

1. Program PLL frequency settings (R0x66:0 and R0x67:0)

2. Power up the PLL (R0x65:0[14] = 0)
3. Wait for a time longer than PLL locking time (> 1ms)
4. Turn off the PLL bypass (R0x65:0[15] = 0)

## Window Control

### Window Start

The row and column start address of the displayed image can be set by R0x01:0 (Row Start) and R0x02:0 (Column Start).

### Window Size

The size of displayed image is controlled by R0x03:1 (Row Width) and R0x04:0 (Column Width). The default image size is 1,600 columns and 1,200 rows (UXGA).

The window start and size registers can be used to change the number of columns in the image from 17 through 1,632 and the number of rows from 2 through 1,216.

## Pixel Border

When bits R0x20:0[9:8] are both set, a 4-pixel border is added around the specified image. This border can be used as extra pixels for image processing algorithms. The border is independent of the readout mode, which means that even in skip, zoom, and binning modes, a 4-pixel border is output in the image. When enabled, the row and column widths are 8 pixels larger than the values programmed in R0x03:0 and R0x04:0. If the border is enabled but not shown in the image (R32[9:8] = 01), the horizontal blanking and vertical blanking values are 8 pixels larger than the values programmed in the blanking registers.

## Readout Modes

### Readout Speeds and Power Savings

The sensor core has two ADCs to convert the pixel values to digital data. Because the ADCs run at half the master clock frequency, it is possible to achieve a data rate equal to the master clock frequency. By turning off one of the ADCs, the power consumption of the sensor is reduced. The pixel clock is then reduced by a factor of two.

In R0x20:0 or R0x21:0, bit 10 chooses between the two modes:

- 0: Use both ADCs and read out at the set pixel clock frequency (R0x0A:0[3:0])
- 1: Use 1 ADC and read out at half the set pixel clock frequency (R0x0A:0[3:0])

This can be used, for instance, when the camera is in preview mode. To make the transitions between two sensor settings easier, some simple context switching is described in “Context Switching” on page 48.

### Column Mirror Image

By setting R0x20:0[1] = 1 (R0x21:0 in context A), the readout order of the columns are reversed as shown in Figure 20. The starting color is preserved when mirroring the columns.

### Row Mirror Image

By setting R0x20:0[0] = 1 (R0x21:0 in context A), the readout order of the rows are reversed as shown in Figure 21. The starting color is preserved when mirroring the rows.

**Figure 20: 6 Pixels in Normal and Column Mirror Readout Modes**

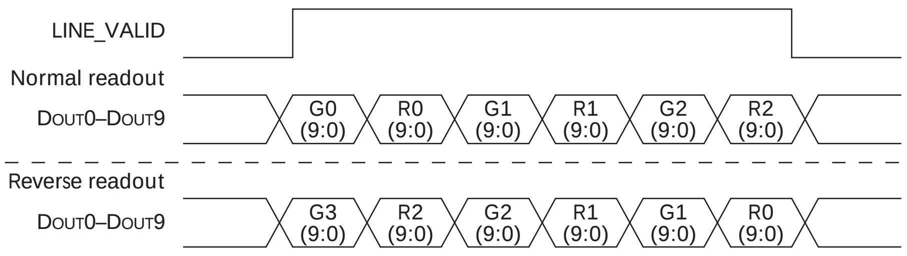
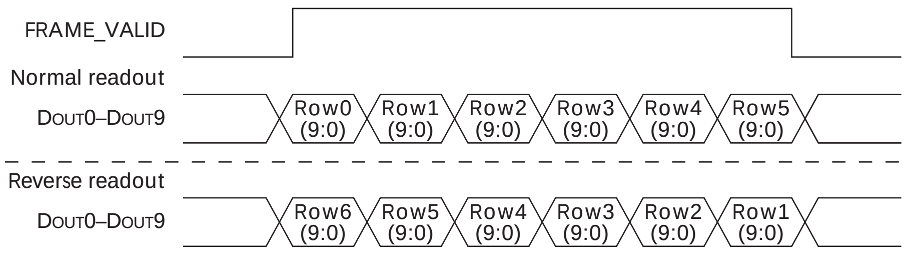


**Figure 21: 6 Rows in Normal and Row Mirror Readout Modes**



## Column and Row Skip

This section assumes context B. If context A is used, replace all references to R0x20:0 with R0x21:0.

By setting R0x20:0[4] = 1 or R0x20:0[7] = 1, skip is enabled for rows or columns, respectively. When skip is enabled, the image is subsampled. The amount of skipping is set by R0x20:0[3:2] (rows) and R0x20:0[6:5] (columns) according to Table 10.

**Table 10: Skip Values**

| Bit Values | Skip Value |
|---|---|
| 00 | 2 |
| 01 | 4 |
| 10 | 8 |
| 11 | 16 |

The number of rows or columns read out is what is set in R0x03:0 or R0x04:0, respectively, divided by the skip value in this table.

In all cases, the row and column sequencing ensures that the Bayer pattern is preserved.

Column skip examples are shown in Figures 22 through 26.

**Figure 22: 8 Pixels in Normal and Column Skip 2x Readout Modes**

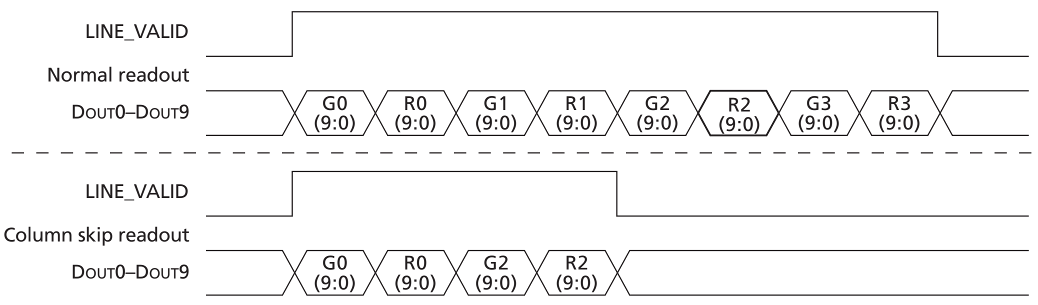


**Figure 23: 16 Pixels in Normal and Column Skip 4x Readout Modes**

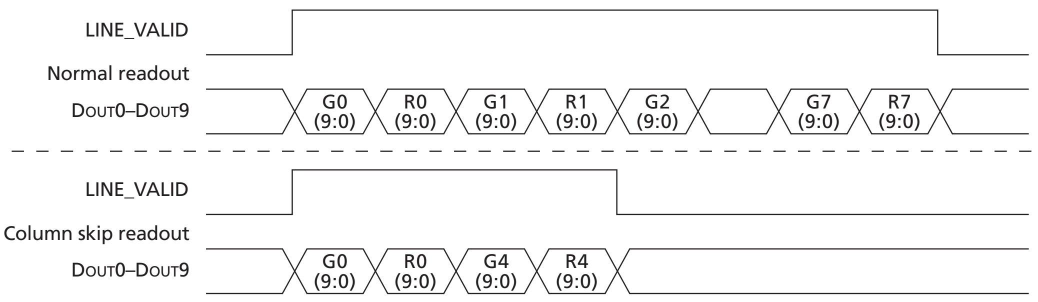


**Figure 24: 32 Pixels in Normal and Column Skip 8x Readout Modes**

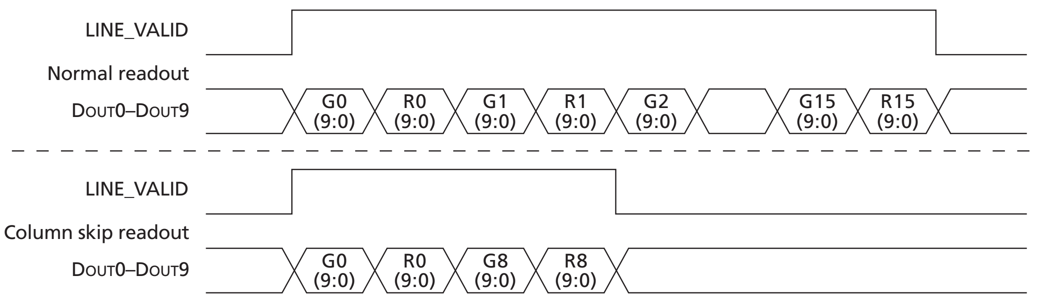
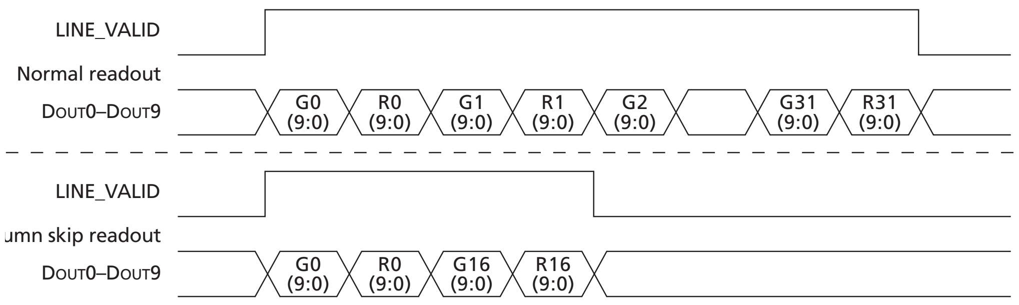


Figure 25: 64 Pixels in Normal and Column Skip 16x Readout Modes



## Digital Zoom

Digital zoom is not used in SOC.

## Binning

The sensor core supports 2 x 2 binning of 4 pixels of the same color. This mode can be activated by asserting R0x20:0[15] (R0x21:0 if context A is used).

Binning is primarily used instead of 2x skip as a way of decimating the picture without losing information. The effect of aliasing in preview mode is eliminated when binning is used instead of just skipping rows and columns.

Activating binning has a few implications:
- It adds a level of skip, so the picture that comes out has the same dimensions as a picture read out with the next higher skip setting.
- It increases the minimum hblank and minimum row time requirements (see Table 13 and Table 14).

Table 11: Row Addressing

| Number of ADCs | Binning | Row Addressing (not considering start address) |
|---|---|---|
| 1 | No | 0, 1, 2, 3, 4, 5, 6, 7,... |
| 2 | No | 0, 1, 2, 3, 4, 5, 6, 7,... |
| 1 | Yes | 0/2, 1/3, 4/6, 5/7,... |
| 2 | Yes | 0/2, 1/3, 4/6, 5/7,... |

Table 12: Column Addressing

| Number of ADCs | Binning | Column Addressing (not considering start address) |
|---|---|---|
| 1 | No | 0, 1, 2, 3, 4, 5, 6, 7,... |
| 2 | No | 0/1, 2/3, 4/5, 6/7,... |
| 1 | Yes | 0/2, 1/3, 4/6, 5/7,... |
| 2 | Yes | 0/1/2/3, 4/5/6/7,... |

## Binning Limitations

To achieve correct operation, the following conditions must be met:
- 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 = 1,200. 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 1,200 / 8 = 150, which is not divisible by 4.

- Binning can be seen as an extra level of skip. The combination binning/16x skip is therefore not legal.

## Frame Rate Control

For a given window size, the blanking registers (R0x05:0 - R0x08:0) along with the row speed register (R0x0A:0) can be used to set a particular frame rate.

The frame timing equations (Table 7 and Table 8 on page 39) can be rearranged to express the horizontal blanking or vertical blanking values as a function of the frame rate:

HBLANK_REG = master clock freq / (frame rate* ((R0x03:0/S + BORDER) + VBLANK_REG)*PIXCLK_PERIOD) - (R0x04:0/S + BORDER)

VBLANK_REG = master clock freq / (frame rate* ((R0x04:0/S + BORDER) + HBLANK_REG)*PIXCLK_PERIOD) - (R0x03:0/S + BORDER)

The HBLANK_REG value allows the frame rate to be adjusted with a minimum resolution of one PIXCLK_PERIOD multiplied by the total number of rows (displayed plus blanking). When finer resolution is required, R0x0B:0 (extra delay) can be used. R0x0B:0 allows the frame time to be changed in increments of pixel clocks.

### Minimum Horizontal Blanking

The minimum horizontal blanking value is constrained by the time used for sampling a row of pixels and the overhead in the row readout. This is expressed in Table 13.

**Table 13: Minimum Horizontal Blanking Parameters**

| Parameter | Default / 2 ADC Mode, No Binning | 1 ADC Mode, No Binning | 2 ADC Mode, Binning | 1 ADC Mode, Binning |
|---|---|---|---|---|
| HBLANK(MIN) | 286 mclks | 324 mclks<br>= 162 pixclks | 470 mclks | 508 mclks<br>= 254 pixclks |

### Minimum Row Time Requirement

The total row time must be sufficient to allow all row operations (readout and shutter operations). The row time is the sum of column width (halved during binning divided by column skip factor) and horizontal blanking, and can therefore be adjusted by programming these.

Table 14 shows minimum row time as a function of mode of operation.

This is a particularly strict requirement during binning because twice as many row operations are required per row and the column width is halved.

**Table 14: Minimum Row Time Parameters**

| Parameter | Default / 2 ADC Mode, No Binning | 1 ADC Mode, No Binning | 2 ADC Mode, Binning | 1 ADC Mode, Binning |
|---|---|---|---|---|
| ROW_TIME(MIN) | 473 mclks | 488 mclks = 244 pixclks | 931 mclks | 946 mclks = 473 pixclks |
| pointer_operations | 461 mclks | 464 mclks | 919 mclks | 922 mclks |

## Context Switching

R0xF2:0 is designed to enable easy switching between sensor modes. Some key registers and bits in the sensor have two physical register locations, called contexts. Bits 0, 1, and 3 of R0xF2:0 control which context register context is currently in use. A "1" in a bit selects context B, while a "0" selects context A for this parameter. The select bits can be used in any combination, but by default are setup to allow easy switching between preview mode and full resolution mode:

**Context B (Default context)**

| | | |
|---|---|---|
| R0xF2:0 | = 0x000B | (Context B) |
| R0x05:0 | = 0x015C | (Horizontal blanking, context B) |
| R0x06:0 | = 0x0020 | (Vertical Blanking, context B) |
| R0x20:0 | = 0x0000 | (2 ADCs, no column or row skip) |

**Description:** Full resolution UXGA (1,600 x 1,200) image at 15 fps

**Context A (Alternate context, preview mode)**

| | | |
|---|---|---|
| R0xF2:0 | = 0x0000 | (Context A) |
| R0x07:0 | = 0x00AE | (Horizontal blanking, context A) |
| R0x08:0 | = 0x0010 | (Vertical blanking, context A) |
| R0x21:0 | = 0x0490 | (1 ADC, 2x column and row skip) |

**Description:** Half-resolution SVGA (800 x 600) image at 30 fps

The horizontal blanking and vertical blanking values for the two contexts are chosen so that row time is preserved between contexts. This ensures that changing contexts does not affect integration time. A few more control bits are also available through the context register (R0xF2:0) so that flash and restarting the sensor can be done simultaneously with changing contexts. See Table 4, "Sensor Register Description," on page 28 for more information.

Settings for skip, 1 ADC mode, and binning can be set separately for context B and context A using R0x20:0 and R0x21:0, respectively. When these settings are referred to in this document, the register is dependent on what context is set in R0xF2:0.

## Integration Time

Integration time is controlled by R0x09:0 (shutter width in multiples of the row time) and R0x0C:0 (shutter delay, in PIXCLK_PERIOD/2). R0x0C:0 is used to control sub-row integration times and only has a visible effect for small values of R0x09:0. The total integration time, $^{t}$INT, is shown in the equation below:

| | | |
|---|---|---|
| $^t$INT | = | R0x09:0 * Row Time - Integration Overhead - Shutter Delay |
| where: | | |
| Row Time | = | (R0x04:0/S + BORDER + HBLANK_REG)*PIXCLK_PERIOD master clock periods (from Table on page 39) |
| S | = | Skip Factor, multiplied by 2 if binning is enabled |
| Overhead Time | = | 260 master clock periods (262 in 1 ADC mode) |
| Shutter Delay | = | R0x0C:0 * PIXCLK_PERIOD master clock periods (/2 in 1 ADC mode) |
| | | |
| with default settings: | | |
| $^t$INT | = | (1,232 * (1,600 + 348)) - 260 - 0 |
| | = | 2,399,676 master clock periods = 66.66ms at 36 MHz |

In the equation, the integration overhead corresponds to the delay between the row reset sequence and the row sample (read) sequence.

Typically, the value of R0x09:0 is limited to the number of rows per frame (which includes vertical blanking rows), so that the frame rate is not affected by the integration time. If R0x09:0 is increased beyond the total number of rows per frame, the sensor adds blanking rows as needed. Additionally, $^t$INT must be adjusted to avoid banding in the image caused by light flicker. Therefore, $^t$INT must be a multiple of 1/120 of a second under 60Hz flicker, and a multiple of 1/100 of a second under 50Hz flicker.

### Maximum Shutter Delay

The shutter delay can be used to reduce the integration time. A programmed value of *N* reduces the integration time by *N* master clock periods. The maximum shutter delay is set by the row time and the sample time, as shown in the equation below:

| | | |
|---|---|---|
| Maximum shutter delay | = | (Row Time - pointer_operations) |
| where: | | |
| Row Time | = | (R0x04:0/S + BORDER + HBLANK_REG)*PIXCLK_PERIOD master clock periods (from Table on page 39) |
| S | = | Skip Factor, multiplied by 2 if binning is enabled |
| pointer_operations | = | see Table 14 on page 48. |
| | | |
| with default settings: | | |
| Maximum shutter delay | = | (1,600 + 348) - 461 |
| | = | 1,487 (master clock periods) |

If the value in this register exceeds the maximum value given by this equation, the sensor may not generate an image.

## Flash STROBE

The sensor core supports both Xenon and LED flash through FLASH output. The timing of FLASH with the default settings is shown in Figure 26, Figure 27, and Figure 28. R0x23:0 allows the timing of the flash to be changed.

The flash can be programmed to:

- fire only once
- be delayed by a few frames when asserted
- and (for Xenon flash) the flash duration can be programmed

When Xenon flash is enabled, an integration time significantly smaller than one frame causes uneven exposure of the image, as does setting a flash pulse width larger than vertical blanking.

Enabling the LED flash causes one bad frame in which several rows have the flash on during only part of their integration time. This can be avoided by forcing a restart (write R0x0D:0[1] = 1) immediately after enabling the flash; the first bad frame is then masked out as shown in Figure 28. Read-only bit R0x23:0[14] is set during frames that are correctly integrated; the state of this bit is shown below.

**Figure 26: Xenon Flash Enabled**

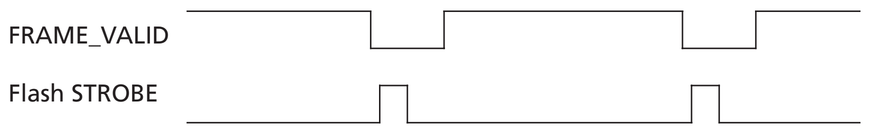
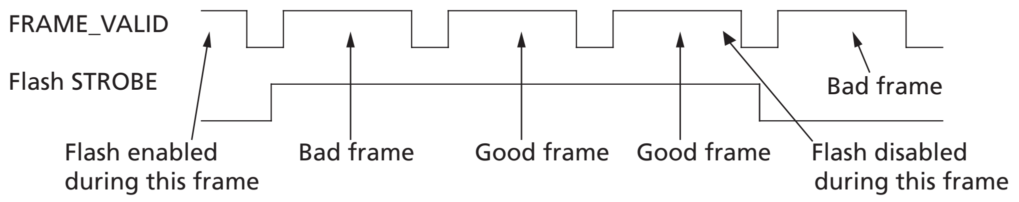
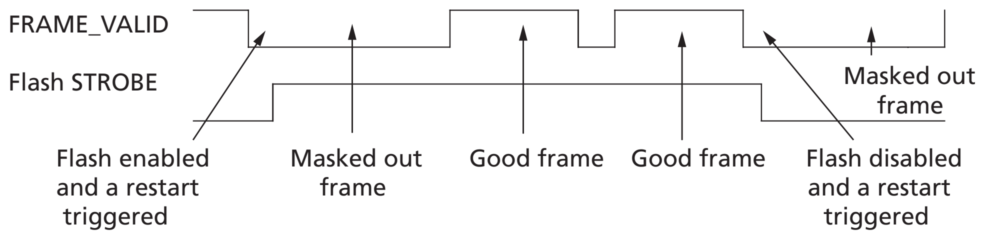


**Figure 27: LED Flash Enabled**


FRAME_VALID
Flash STROBE
Bad frame
Flash enabled during this frame
Bad frame
Good frame
Good frame
Flash disabled during this frame

Note: Integration time = number of rows in a frame.

**Figure 28: LED Flash Enabled, Using Restart**


FRAME_VALID
Flash STROBE
Masked out frame
Flash enabled and a restart triggered
Masked out frame
Good frame
Good frame
Flash disabled and a restart triggered

Note: Integration time = number of rows in a frame.

## Global Reset

The sensor core provides a global reset mode in which the pixel integration time is controlled by an external mechanical shutter. The sensor can then operate on a lower clock frequency, reducing the bandwidth on the interface between the sensor and the host processor without losing image quality.

The basic operation is as follows:

1. The sensor operates in either preview or full-frame mode (electronic rolling shutter [ERS]).
2. A rising edge on the signal GRST_CTR or a WRITE to an internal register starts the global reset sequence.

3. The sensor now enters the snapshot mode and after a certain time, all the lines in the sensor array are reset and kept in a reset state until the integration starts.

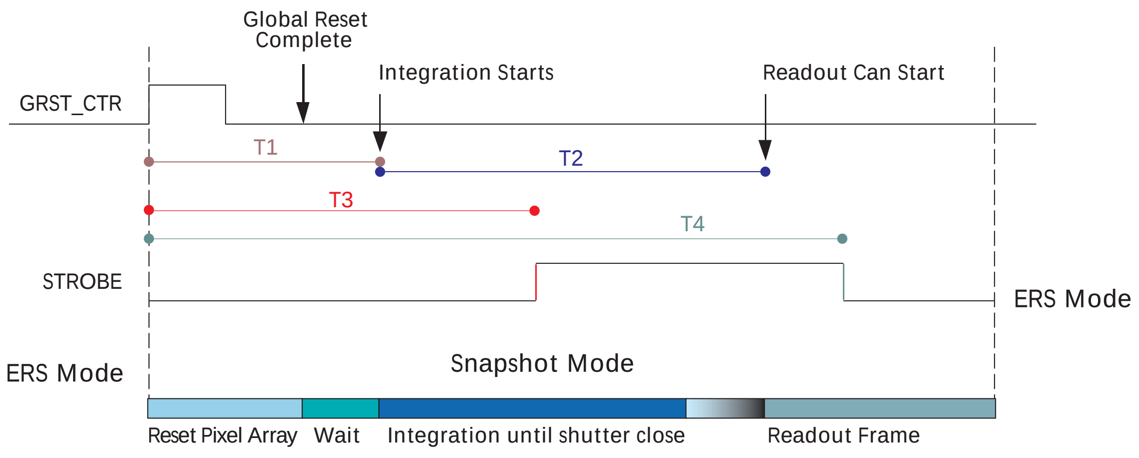
The start of the integration (exposure) period, the assertion of STROBE, the start of the readout, and the de-assertion of STROBE can be controlled by internal registers (T1, T2, T3, and T4, as shown in Figure 29).

The sensor core provides an output signal, STROBE, that can be used to control the mechanical shutter. This signal can be programmed to occur in a specified window around the actual start of integration.

During global reset, FLASH is programmed in a different way than during normal ERS operation. Normally, the FLASH behavior is programmed using R0x23:0. In global reset mode, FLASH is programmed in the same way as STROBE, showed in Figure 29, using registers R0xC5:0 and R0xC6:0.

R0xC0:0[0] controls the mechanism for starting the readout after a GLOBAL RESET operation. If this bit is HIGH, the integration time is directly controlled by GRST_CTR. Very long integration times can be achieved this way.

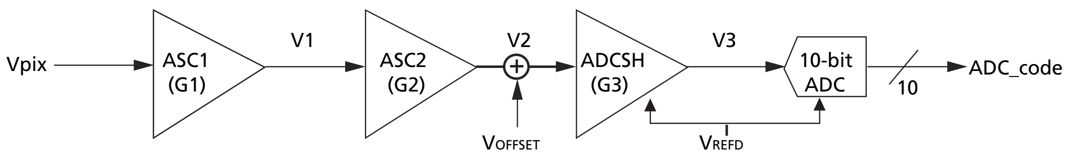
**Figure 29: GLOBAL RESET Operation**



## Analog Signal Path

The sensor core features two identical analog readout channels. A block diagram for one channel is shown in Figure 30. The readout channel consists of two gain stages (ASC1 and ASC2), a sample-and-hold (ADCSH) stage with black level calibration capability (VOFFSET), and a 10-bit ADC.

**Figure 30: Analog Readout Channel**



## Stage-by-Stage Transfer Functions

Transfer functions proceed stage-by-stage, as follows:

| | | |
|---|---|---|
| Let VPIX be the input of the signal path: | VPIX = pixel output voltage = signal path input voltage, | |
| The output voltage of ASC 1st stage is: | V1 = -1*G1*VPIX | (1) |
| The output voltage of ASC 2nd stage is: | V2 = -1*G2*V1 | (2) |
| The output voltage of ADC Sample-and-Hold stage is: | V3 = 2*G3*V2 - VREFD + VOFFSET | (3) |
| and the ADC output code is: | ADC output code = 511*(1 + (V3 / VREFD)) | (4) |
| From (1) to (4), the ADC output code can also be written as: | ADC code = (1022/VREFD)*[G1*G2*G3*VPIX + (Voffset/(2*G3))] | (5) |

Where G1, G2, and G3 are the gain settings, VOFFSET is the offset (calibration) voltage, and VREFD is the reference voltage of the ADC. The gain setting G3 is applied to the signal but is not applied to VOFFSET. The parameters VREFD, G1, G2, G3, and VOFFSET are described next.

### VREFD

The VREFD parameters are as follows:

| | | |
|---|---|---|
| The ADC reference voltage VREFD IS: | VREFD = VREF_HI - VREF_LO | (6) |
| where | VREF_HI = 55.5mV*(R0x41:0[7:4] + 23) | (7) |
| using default register values: | VREF_HI = 55.5mV*(13 + 23) = 1.998V | |
| and | VREF_LO = 55.5mV*(R0x41:0[3:0] +11) | (8) |
| using default register values: | VREF_LO = 55.5mV*(7 +11) = 0.999V | |
| so | VREFD = 55.5mV*(R0x41:0[7:4] - R0x41:0[3:0] + 12) | (9) |
| using default register values | VREFD = 1.998 - 0.999 = 0.999V | |

## Gain Settings: G1, G2, G3

The gains for green1, blue, red, and green2 pixels are set by registers R0x2B:0, R0x2C:0, R0x2D:0, and R0x2E:0, respectively. Gain can also be globally set by R0x2F:0. The analog gain is set by bits 8:0 of the corresponding register as follows:

| | |
|---|---|
| G1 = bit 7 + 1 | (10) |
| G2 = bit 6:0 / 32 | (11) |
| G3 = bit 8 + 1 | (12) |

Digital gain is set by bits 11:9 of the same registers.

### Offset Voltage: VOFFSET

The offset voltage provides a constant offset to the ADC to fully utilize the ADC input dynamic range. The offset voltages for green1, blue, red, and green2 pixels are manually set by registers R0x61:0, R0x62:0, R0x63:0, and R0x64:0, respectively. The offset voltages also can be automatically set by the black level calibration loop.

For a given color, the offset voltage, VOFFSET, is determined by:

VOFFSET = 0.50V*offset_gain*offset_sign*offset_code[7:0]/ (13)
255
where:
"offset_sign" is determined by bit 8 as:
if bit 8 = 0, offset_sign = +1 (14)
if bit 8 = 1, offset_sign = -1 (15)
"offset_code" is the decimal value of bit<7:0>

OFFSET_GAIN is determined by the 2-bit code from R0x5A:0[1:0], as shown in Table 15. These step sizes are not exact; increasing the stage0 ADC gain from 2 to 4 decreases the step size significance; decreasing the ADC VREFD increases the step size significance.

**Table 15: Offset Gain**

| R0x5A:0[1:0] | OFFSET_GAIN |
|---|---|
| 00 | OFFSET_GAIN = 0 (no calibration voltage is applied) |
| 01 | OFFSET_GAIN = 0.25 (1 calibration LSB is equal to 0.5 ADC LSB when VREFD = 1V) |
| 10 | OFFSET_GAIN = 0.50 (1 calibration LSB is equal to 1 ADC LSB when VREFD = 1V) |
| 11 | OFFSET_GAIN = 1 (1 calibration LSB is equal to 2 ADC LSB when VREFD = 1V) |

### Recommended Gain Settings

The analog gain circuitry in the sensor core provides signal gains from 1 through 15.875.

**Table 16: Recommended Gain Settings**

| Desired Gain | Recommended Gain Register Setting |
|---|---|
| 1–1.969 | 0x020–0x03F |
| 2–7.938 | 0x0A0–0x0FF |
| 8–15.875 | 0x1C0–0x1FF |

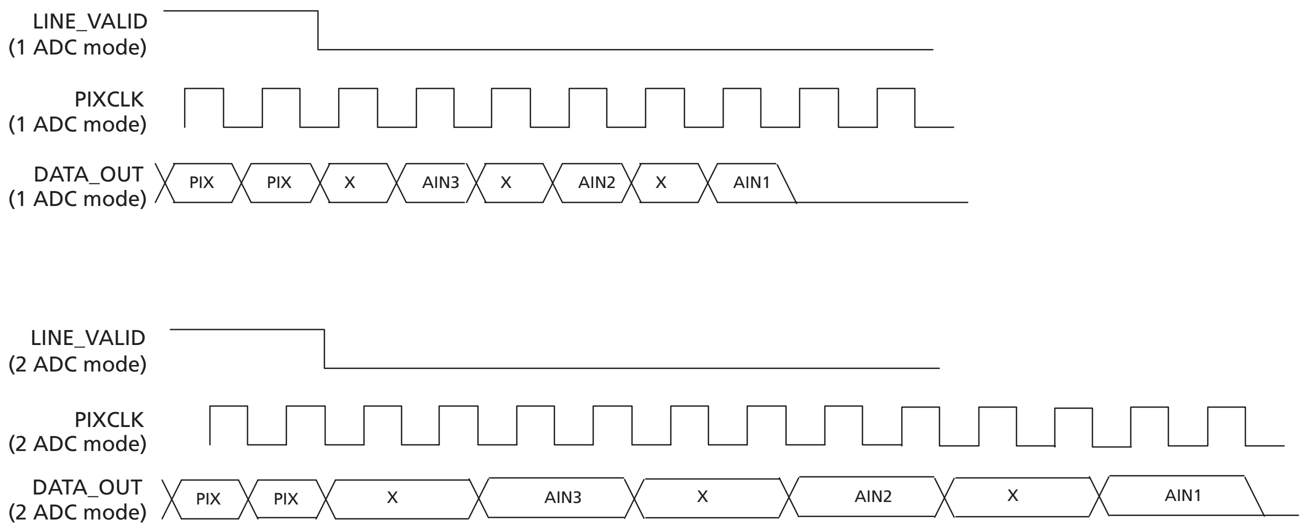
## Analog Inputs AIN1–AIN3

Since ADC resources in the sensor core are not used to digitize pixel array output 100 percent of the time, they can be intermittently used to sample external analog signals coming from a variety of sources—e.g., a flash charging circuit or an auto focus lens actuator. If R0xE3:0[15] = 1, the chip samples AIN1–AIN3 once per every pixel array row (after reading out the row). Digital data produced by this sampling are available to the user in two ways:

- They can be read from registers R0xE0:0 through R0xE2:0.
- If R0xE3:0[14] = 1, the data are additionally inserted into image data stream each time LINE_VALID goes LOW.

The nominal range of AIN are 0V + VOFFSET to VREFD + VOFFSET. VREFD is the ADC reference voltage (nominally 1V), but can be programmed. (See "Analog Signal Path" on page 51.) VOFFSET is the offset in the ADC and is typically ±10mV to 20mV. If required, the offset can be measured by converting a calibrated reference voltage, which can be used to compensate at the input. The ADC is designed to operate with differential inputs. Since AIN1–AIN3 are used as single-ended inputs to the ADC, it is recommended to average values from several samples (if possible, a whole frame) to cancel out noise.

**Figure 31: Timing Diagram AIN1–AIN3 Sample**



## Firmware

Firmware implements all automatic functionality of the camera, including auto exposure, white balance, auto focus, flicker detection and so on. The firmware consists of drivers, generally one driver per each major control function. The firmware runs on a 6811-compatible microcontroller with a mathematical co-processor. 32K of metal mask-programmable PROM is available for non-volatile code. 2K of volatile SRAM is available, where typically the first 1K of SRAM is used by the PROM drivers for data and stack, while to the second 1K is reserved for loadable custom code.

The firmware and MCU subsystem targets to achieve the following high-level goals and features:

- execute applications from ROM and RAM
- execute applications in real-time, synchronized to IFP
- allow loading custom executable programs
- firmware controls IFP by reading and writing ring bus registers
- firmware consists from drivers and bootstrap code
- user can extend or override functionality of ROM drivers
- user can add all new drivers
- drivers have variables; user configures drivers by setting variables via two-wire serial interface
- when idle, puts MCU into low-power sleep mode
- a minimal code overhead due to flexibility
- transparency from firmware build version
- stability, diagnostics and recovery from software crashes
- framework for application and driver development

The firmware is written in C with some inline assembler where extra performance is required. It runs on 6811-compatible microcontroller with a mathematical co-processor.

## Overview of Architecture

### Bootstrap Routine

The firmware consists of a bootstrap code, drivers and a test module. The bootstrap code initializes internal low-level MCU registers, performs a mini-self-test, and sets up a 128-bytes deep stack. The main routine is then invoked.

### Drivers

Firmware applications are organized into drivers. Each driver performs a specific function. Each driver has its own data structure with various variables. The user can access the data structure via R198:1 and R200:1 to read or write variables. The access is implemented using hardware direct memory access (DMA).

Drivers are static objects with virtual methods. Each driver has an associated data structure and a driver ID. To access a driver variable, the user must know driver ID and variable offset in the data structure. Driver methods are implemented using a virtual method table (VMT). VMT is a table of pointers to driver methods. The object data structure has a pointer to VMT. Software routines must call methods indirectly, via VMT pointer. This allows overriding of methods.

### Extending and Overriding Drivers

VMTs are necessary to be able to override a driver. To override a driver, the software developer has to perform the following steps:

1. Load executable code
2. Construct new VMT
   a. use pointers to the new functions
   b. use old pointers to call old methods
3. New methods can call inherited functions using old VMT pointers
4. Point driver data structure VMT pointer to the new VMT

The main routine continues firmware initialization by calling driver initialization functions. If all drivers initialization completes without error, the main function indicates a successful firmware start by setting a code in R195:1. The user can request the firmware to start in a special test mode. The bootstrap code reads R195:1 and invoke the corresponding routine.

### Variables

The main routine sets up a system table of pointers to drivers. The address of the table is recorded in SFR 0x1050. The hardware DMA uses this register to implement logical access to variables.

MCU SRAM can be accessed in two ways:

1. By specifying a physical address (physical mode)
2. By specifying driver ID and variable offset in driver's data structure (logical mode)

In physical mode, the user writes address to be accessed (read or written) in R198:1 and reads or writes data in R200:1. PROM data is not accessible using this mechanism.

In logical mode, the user writes driver ID and variable offset combination in R198:1 and reads or writes data in R200:1. This mode allows accessing driver variables regardless of their physical location, which can change from one firmware version build to another. In this sense variables are very much like hardware registers.

### Uploading Custom Code

To upload custom code

1. Write application data to SRAM using physical variable access
2. Use monitor driver to invoke the code

See “Monitor Driver” on page 58 for details.

## Hardware Resource Programming

The MCU subsystem provides a number of hardware resources to the software developer. These resources include

### DMA

To enable user access variables.

### Watchdog

The watchdog monitors execution of firmware and resets MCU when a time out is detected. During normal execution the firmware must reset the watchdog periodically or disable it.

## Sleep and Wakeup

Logic is available to suspend MCU clock and enter low-power mode. The MCU can wake up synchronously with IFP when IFP is processing a line with a specified number.

## Internal Register Bus Access

The firmware has master access to the ring bus and can read and write any hardware register.

## GPIO and Waveform Generator on Local Bus

For better performance GPIO and waveform generator are connected to the local 6811 SFR bus.

## High-Precision Timer

A 32-bit master clock (80 MHz) counter for code profiling or time measurements.

## Sleep and Wakeup Programming

To sleep and wake up on a certain line, program the following:

1. Set line number to wake up at SFR 0x1042.
   a. User can choose to wake up on IFP line, referred to interpolation, or frame enable of IFP, sensor, JPEG, and FIFO. The edge and level are selectable.
2. Clear SFR 0x1040[0].
   b. This clears previous sleep event notification.
3. Read from SFR 0x1040 to attempt sleeping.
   c. MCU tries entering sleep mode. MCU does not sleep if DMA is happening or the wakeup line number is reached or GPIO notification signal is asserted.
4. Read SFR 0x1040 to check if sleep succeeded

**Note:** If bit 0 is set, sleeping was successful. Otherwise repeat step 3.

A library function is available for easy sleep mode usage.

## Two-Wire Serial Interface Programming

The microcontroller can access all registers on the ring bus. The access is enabled by custom SFRs located at addresses $1060 through $1066. To carry out a bus read, set desired I/O page in READ_IOPAGE and write desired register address to READ_IOADR to initiate the READ transaction. Poll IOSTATUS until it returns a "0." The read data is available in IODATA.

To carry out a bus write, set the desired I/O page in WRITE_IOPAGE. and the register address in WRITE_IOADR. Load write data into IODATA to initiate a WRITE transaction. Poll IOSTATUS until it returns a "0" to signal the completion of the WRITE transaction.

A library function is available for easy sleep usage.

## DMA Programming

Set up table of variables for logical access. Driver ID corresponds to the pointer offset in the table. Program table address into SFR 0x1050.

## Warm Restarts and Watchdog Programming

The watchdog register can detect a software crash and have the system recover from it by issuing a hardware reset. The watchdog resets the microcontroller subsystem if SFR 0x1040 register have not been cleared for *N* frames.

Number *N* is programmed in the WATCHDOG_REG. Setting *N* to “0” disables the watchdog. The user may need this option when the microcontroller runs a lengthy task, e.g., calculation of CRC or RAM testing.

When the watchdog resets the microcontroller, OS restarts and checks for restart reason. Register RESTART_CODE specifies the probable code. The user can also see restart code by reading R195:1. A restart caused by resetting the chip is called a "cold" restart. Restart caused by the watchdog, software, or illegal code trap are called "warm" restarts.

### High-Precision Timer

SFR 0x1048 is a high-precision timer is available for use in applications, and profiling code during development. It is a 32-bit counter incremented every master clock, except when in standby. To profile, read and store the timer before executing a routine. After the routine exits, read the timer again and subtract from the initial value.

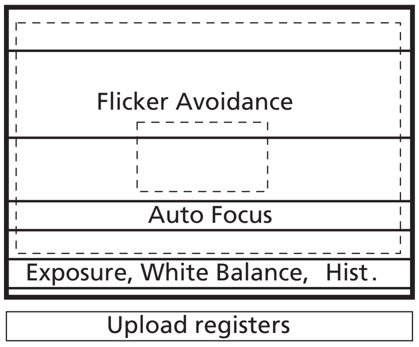
### Safe and Test Modes

A safe mode is available to start MCU without running drivers. Set R195:1 and reset MCU to enable this mode. Only monitor runs allowing to start ROM or load and start RAM code.

R195:1 can also invoke the MCU and memory test mode.

## Application Scheduling

Drivers run in synchrony with IFP image processing. As the frame comes through the color pipeline in real time, certain statistics are calculated and become available. Typically, the flicker avoidance driver is active at the top of the frame. Auto focus measurement window is located in the center of the image. Once its result is ready, the auto focus and auto focus mechanics drivers execute. The exposure, white balance, and histogram statistics windows nearly cover the entire image. Their result is available at the end of the frame, when AE, WB, and histogram drivers are invoked. Finally, during the vertical blanking, drivers upload calculated settings to sensor and SOC.

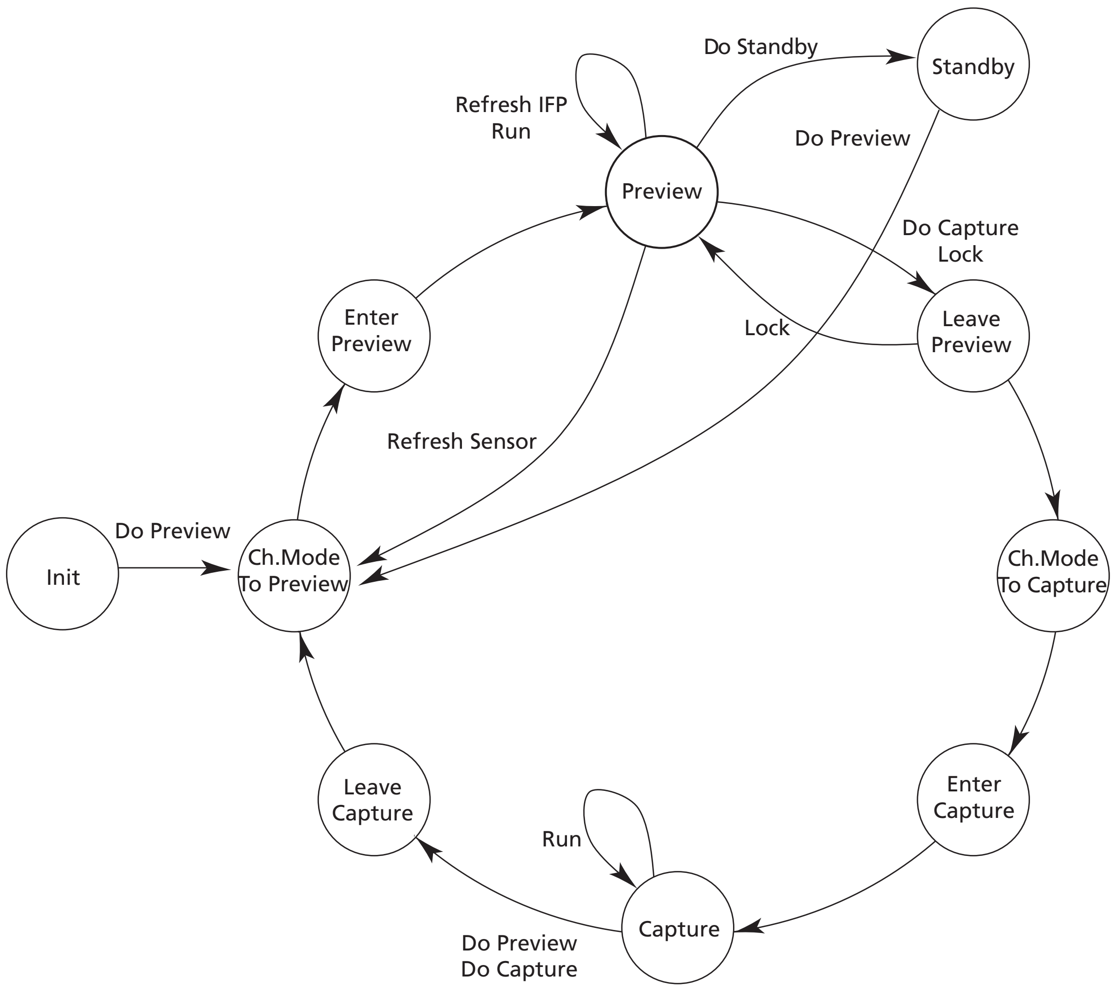


## Monitor Driver

The main monitor function is to run code. Set the "arg1" to the address of the function to be called, set “arg2” to an optional 16-bit parameter to be passed and set "cmd" to 0x01. The sequencer driver calls the monitor driver typically once per frame. The desired function is invoked in one frame time or less. The monitor also provides CRC calculation capability to check if the uploaded firmware was transmitted correctly.

Variable mon.ver contains firmware version number. mon.ver[7] = 1 indicates uploaded firmware.

**Figure 32: Sequencer Driver**



## Sequencer Driver

The sequencer is a finite state machine that controls the operation of main functions and the switching between modes. Camera operation is organized as states, such as preview or capture. The sequencer carries out a number of programs, such as previewing, preview lock, capture, and so on.

The state of the sequencer is indicated in variable state. To have the sequencer execute a certain program, set the program number in cmd. The host then should monitor state to know when to change resolution and/or capture frames.

Each state has its configuration (see "Firmware Driver Variables" on page 63). Before executing programs, set up state configurations to customize the program. For example, to capture compressed frames, enable compression in capture state configuration.

A typical scenario includes the following:

1. Configure mode variables after hardware reset

   a. Set up sensor image size for preview
   b. Set up displayed image size
   c. Set up FIFO to smooth data rate
2. Configure preview mode
   a. Set auto exposure, white balance and auto focus speed
3. Execute sensor REFRESH command
4. Run in preview until shutter button is depressed
   a. If the shutter button is depressed halfway, execute the lock program
      i. Configure preview state to disable necessary functions to be locked, e.g., auto exposure, white balance, and auto focus
      ii. Configure PreviewLeave state for fast settling of those functions
      iii. Execute the lock program
      iv. The lock program transits PreviewLeave state, perform fast settling, and return to Preview state, which was configured to freeze (lock)
   b. If the shutter button is depressed all the way, execute the capture program
      i. If already in lock mode, disable PreviewLeave state and execute capture program
      ii. If not in lock state and some settling is required (auto focus), configure PreviewLeave state for fast settling
      iii. Execute capture program. The program transitions to PreviewLeave, perform required settling and proceed through mode change to capture
5. Capture frame
   a. Preconfigure capture mode variables for correct image size, compression, and select video/still
   b. Monitor the "state" variable. When the state is capture, grab the frame(s)

Optionally, configure Capture Enter or Preview Leave to have output blanked. This helps grabbing correct frame for capture.

## Flash Types and Settings

The sequencer supports three types of flash light source:

- LED
- Xenon
- Xenon continuous

Flash type is selected using seq.sharedParams.flashType.

States in the sequencer diagram have an associated flash setting. That setting is specified using variables such as seq.prevParEnter.flash.

The four settings available are shown in Table 17.

**Table 17: Flash Types and Settings**

| Type | Setting |
|---|---|
| MODE_FLASH_OFF | Turns flash off |
| MODE_FLASH_ON | Turns flash on |
| MODE_FLASH_AUTO | Checks for low-light condition and turns flash on if necessary |
| MODE_FLASH_LOCKED | Turns flash on or off to put it into the state selected by MODE_FLASH_AUTO |

### Using LED Flash

To perform CM_LOCK

1. Set PreviewLeave flash configuration to MODE_FLASH_AUTO. Set the state's AE and WB to fast settling

2. Set Preview flash configuration to MODE_FLASH_OFF
3. Perform CM_LOCK. That turns flash on in low-light conditions, perform fast AE/WB settling, return to Preview and turn flash off

To continue with capture after CM_LOCK

1. Disable all automatic functions in PreviewLeave. Set flash mode to MODE_FLASH_LOCKED
2. Disable CaptureEnter state
3. Perform CM_CAPTURE. That restores flash state in PreviewLeave and make a snap-shot

To perform CM_CAPTURE without CM_LOCK

1. Set PreviewLeave flash configuration to MODE_FLASH_AUTO. Set the state's AE and WB to fast settling.
2. Disable CaptureEnter state
3. Perform CM_CAPTURE. That automatically turns flash on, perform fast settling and make a snapshot

**Note:** When the LED flash is turned on, the current frame coming out has not seen the effect of the LED; therefore, use the SkipFrame variable to skip a frame when turning the LED on.

### Using Xenon Burst Flash

This type of flash fires on every frame. In many respects it is similar to LED, except that the sensor integration time must always be more than 1 frame.

### Using Xenon Flash

The three flash types have their own specifics.

To capture images using one-shot Xenon, do the following.

1. If using CM_LOCK, carry out the lock operation as usual
2. Before issuing CM_CAPTURE, re-program the sequencer to disable automatic advancement from state to state, seq.StepMode
3. Reprogram CaptureEnter state to have all automatic functions off (AE, AF, WB, FD)
4. Advance to ModeChange and load Xenon-specific parameters directly into the color pipeline

- color matrix for Xenon R96-R102:1
- integration time R9:0 (1 full frame or more)
- analog gains, R0x2B–0x2E:0
- digital gains R78:1, R106–109:1

## Low-Light Operation

A number features are available to improve image quality in low-light conditions:

- increased noise suppression in interpolation
- reduced color saturation
- reduced aperture correction
- increased aperture correction threshold
- Y-filter

The seq.sharedParams.LLmode variable enables individual low-light features. seq.sharedParams.LLvirtGain1 and 2 specify gain thresholds to enable some of these features. Values are given by seq.sharedParams.LLInterpThresh1/2, LLSat1/2, LLApCorr1/2 and LLApThresh1/22.

# Firmware

## Auto Exposure Driver

The AE driver works to achieve desirable exposure by adjusting sensor core's integration time, analog, and digital gains. The driver can be configured with respect to desired AE speed, maximum and minimum frame rate, the range of gains, brightness, backlight compensation, and so on.

AE driver typically runs in one of these modes. The modes are set in the sequencer driver for each sequencer state.

- fast settling preview—reach target exposure as fast as possible
- continuous preview—a slow-changing mode good for video capture
- evaluative—evaluate current scene and adjust exposure for still capture

Some key variables affecting all modes:

- ae.Target— controls target exposure for all modes. Increasing or decreasing this variable makes the image brighter or darker
- ae.Gate—how accurate AE driver tracks the target exposure
- ae.weights—specify weights for central and peripheral backlight compensation in preview modes

AE driver adjusts exposure by programming sensor gains, R43–46:0 and R65:0, sensor integration time R9:0 and R12:0 and IFP digital gains R78:2 and R110:1.

### Evaluative Auto Exposure

Evaluative AE (EAE) selects optimum exposure for scenes where conventional AE does not give good results:

- scenes involving sun
- back light scenes
- strong contrast scenes and so on

EAE breaks down input image into a 4 x 4 grid of subwindows and analyzes their exposure value (EV) readings. It evaluates brightness and contrast in the image, the scene and adjusts exposure appropriately. Variables ae.mmEVZone1/2/3/4 keep programmable thresholds defining brightness classes. Variable ae.mmShiftEV is used to calibrate EV readings for particular module type.

## Auto White Balance Driver

AWB detects the temperature of the light source in the scene and adjusts color correction to always produce image for sRGB display. In other words, it makes the gray areas in the raw image look also gray in the output image. To detect the illuminant temperature, the driver reads results generated by the statistics block in the color pipeline, R48–50:1. Color correction is achieved via adjusting sensor analog RGB gains R43–46:0, IFP digital RGB gains R106–109:1, and coefficients of the color correction matrix R96–102:1.

The adjustment is done in two ways:

The driver adjusts RGB gains to achieve a gray output. When gains are too large, the driver also adjusts the CCM. During part calibration, two sets of CCM coefficients are obtained, one for red-rich, incandescent illumination and one for blue-rich daylight illumination. These two sets are specified in the AWB driver using awb.ccmL and awb.ccmRL arrays of variables. Variable awb.ccmL corresponds to the red-rich set, also called left. Variable awb.ccmRL is the delta between blue-rich—or right—and the left sets. CCM programmed into the color pipeline is available in variables awb.ccm. The current setting is calculated by interpolating between the left and right sets. The awb.CCM position indicates the position, from 0–left, red-rich to 127–right, blue-rich. For example, cool-white office illumination tends to have a position near 64. Variables awb.CCMpositionMin, awb.CCMpositionMax, awb.GainMin, awb.GainMax specify CCM position and digital RGB gain limits. When changing these settings, issue a REFRESH command to put new values into effect.

AWB speed is controlled by awb.JumpDivisor and awb.GainBufferSpeed. AWB statistics window typically covers the entire image. Its size is set by awb.windowPos and awb.windowSize. Issue a REFRESH_MODE command for settings to take effect. AWB can be forced to daylight when AE detects outdoor condition, see seq.mode[5].

AWB can be operated manually in two ways:

1. Set awb.mode[5] = 1. This forces digital RGB gains to unity. Program desired CCM position into awb.CCMposition.
2. Set awb.CCMpositionMin and awb.CCMpositionMax limits to desired position. Issue REFRESH command. CCM moves into specified position. To fix gains to unity, set awb.GainMin and awb.GainMax to 128. Otherwise digital gains continue changing automatically.

## Auto Focus Driver

The AF driver works to find best lens position providing maximal image sharpness. The driver operates in two modes: manual and auto mode. In manual mode (af.mode[7] = 1) focus position is defined by variable af.bestPosition. User can change lens position from af.posMin to af.posMax.

Automatic mode is started by af.mode=1, The driver automatically scans af.numSteps focal zones to find best position. AF driver makes search of the best focus position based on average sharpness and luma information received from 16 programmable windows. Each filter kernel has seven user-programmable elements specified in registers R75:2, R75:2, R85:2 and R86:2. User can specify window size and position (af.windowPos, af.windowSize), number of the focal zones to scan (af.numSteps), weight for each window (af.zoneWeights).

Public variables of the AF driver listed in Table 14, “Driver Variables-Auto Focus Driver (ID = 5),” on page 77 and register settings defining the AF filters (see page 14) 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 instead of 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 Sequencer driver, the master firmware driver continuously running on 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 4x4 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.

## Auto Focus Mechanics Driver

The auto focus mechanics (AFM) driver comprises functions and variables that enable the AF driver to control different types of lens actuators through the general purpose I/O module (GPIO). The AFM driver provides the AF driver with a single, abstract, hardware-independent control interface hiding from it all details of GPIO programming and GPIO interactions with external devices. Each type of lens actuator requires different input signals and produces different response (lens movements and feedback signals), but can be controlled by the AF driver using only 4 variable function pointers (afm.vmt->pInit, afm.vmt->pSetPos, afm.vmt->pExecCmd, afm.vmt->pGetStatus) and three 1-byte variables (afm.curPos, afm.prePos, and afm.backlash) supplied by the AFM driver.

The function pointers are variable (by changing the value of afm.vmt) because they must point to different AFM driver functions depending on the current selection of lens actuator type indicated by the variable afm.type. By default, afm.type equals 0 and the pointers point to dummy functions that do nothing. The AF driver can safely use them, but using them has no effect on the GPIO - it generates no output signals. This is appropriate when no external AF hardware is connected to it. If there is a lens actuator to control, the AF driver can do so only after the afm.vmt is changed to give it access to the device-matching control functions in the AFM driver. Proper simultaneous change of afm.type and afm.vmt can be requested at any time from the MT9D111 sequencer, by first setting afm.type to desired new value plus 128 and then setting seq.cmd to 5.

The AFM driver supports two types of lens actuators, one of which (stepper motor type, afm.type=2) includes a variety of devices from different vendors, while the other (helimorph type, afm.type=1) as of this writing encompasses just one actuator. However, this actuator is representative of a broad category of devices that are controlled by writing simple commands through their $I^2C$ interfaces and provide no feedback on their status. The AFM driver functionality developed to handle one such device can be partly reused to handle other similar devices.

## Mode Driver

The mode driver reduces integration efforts by managing most aspects of switching between preview (mode A) and capture (mode B) modes. It remembers vital register values for each image acquisition mode and uploads these values to the appropriate registers upon each mode switch. In addition to remembering the register data, it also creates preloaded and custom gamma and contrast tables for each mode.

To change the mode driver parameters, upload the new values to the mode driver table (ID = 7). Upon the next mode change or sequencer REFRESH (CM_REFRESH) command, these mode driver values are uploaded to the appropriate sensor and system registers, and the image processing then reflects the new values (at the beginning of the next frame acquired).

To control the output image size, upload the dimensions to the mode driver variables: output width_A, output height_A, output width_B and output height_B. The mode driver automatically applies any appropriate downsizing filter to achieve this output image size as well as updating the FIFO output size when in JPG bypass (RAW) mode. It is important so set-up the imager to output an image equal to or larger than the target output image. It is also important for the crop window (crop_left_A/B and crop_right_A/B) to be equal to or larger than the target output image.

To upload a custom gamma table, upload the values to the appropriate mode driver locations (see Table 17, "Driver Variables-JPEG Driver (ID = 9)," on page 90), and select the gamma table type to be three (user defined) for the particular mode. If a contrast level is selected, it is applied to the user uploaded gamma table.

The driver contains settings for raw and output image size and pan for each mode. See variables such as mode.sensor_col_width_A/B, mode.sensor_row_height_A/B and mode.fifo_width_A/B, mode.fifo_height_A/B. Blanking and readout mode— the use of skip, binning, and 1ADC modes— is configured using sensor core registers R5:1-8:1 and R32:1, R33:1.

To change overall image brightness by changing adding luma offset at output, set
- mode.y_rgb_offset_A for preview
- mode.y_rgb_offset_B for capture

To change output format, set
- mode.output_format_A for preview
- mode.output_format_B for capture

Similarly, the user can set special effects for each mode using mode.spec_effects_A/B.

## Histogram Driver

The histogram driver continually works to reduce image flare and continually analyzes input image histogram and dynamically adjusts the black level, R59:1. When flare is present the histogram does not contain dark tones, causing the driver to subtract off a higher black level thus regaining the lost contrast. In certain situations the scene may contain no dark tones without flare. The histogram driver cannot distinguish this situation and alters the black level just the same, causing the image to have more contrast, which looks acceptable in many situations.

Variable hg.maxDLevel sets the maximum level that can be subtracted off the input data. Set this value to match lens flare percentage. For example, if a lens typically has a 5 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. hg.percent indicates how many percent of histogram dark tones need to be clipped. The recommended value is 0. The hg.DLevelBufferSpeed controls the speed of adjustment, 32 being fastest and 1 being the slowest.

## Flicker Avoidance Driver

Indoor light sources powered from the wall AC power supply often exhibit fast oscillations. The oscillations are too fast to be seen; their frequency often being the frequency of the wall power supply. The oscillations interact with the sensor and can show in the image as standing or rolling horizontal bars. Such bars are visible when sensor integration time is not a natural multiple of the flicker period. Since there are two AC frequencies in common use, 50Hz and 60Hz, the imager integration time must match to ambient indoor flicker frequency to produce good images.

The MT9D111 can automatically detect the presence of flicker and adjust its integration time to avoid it.

To set up flicker avoidance program the following variables:

- fl.R9_step60 and fl.R9_step50 are flicker periods for 50 and 60Hz illumination expressed in row time, fl.R9_step60=Tflk/Trow. For 60Hz (1/120)/Trow and for 50Hz (1/100)/Trow. The light flicker frequency is twice the power frequency due to rectification.
- fl.search_f1_50 and fl.search_fl are ranges of frequencies to look for, fl.search_f1_50=0.2*fl.R9_step50-1; fl.search_f2_50=0.2*fl.R9_step50+1; fl.search_f1_60=0.2*fl.R9_step60-1; fl.search_f1_60=0.2*fl.R9_step50+1
- fl.stat_min and fl.stat_max are detection thresholds; for flicker to be detected fl.stat_max continuous frames must contain the searched frequency at least fl.stat_min times

Other variables are as follows:

fl.windowPosH indicates current sampling window position. Variable fl.windowHeight indicate window size. Do not change these variables.

Flicker can operate in automatic and manual modes. Automatic mode is enabled by default. fl.mode[5] indicates current 50Hz/60Hz mode. Make sure to set flicker detection bits in sequencer states. To choose manual mode, set fl.mode[7]=1. Bits fl.mode[6]=!fl.mode[5] indicate 50Hz/60Hz frequency selection. To set 50Hz manual mode, set fl.mode=0x80. To set 60Hz manual mode, set fl.mode=0xC0. As frequencies are selected, the auto exposure integration time step is assigned the appropriate fl.R9_step50/60 variable value.

For the flicker avoidance driver to operate correctly, the sensor frame rate may not be exactly 15 fps or 30 fps. Frame rate must be tuned off the exact frame rate. This is a necessary condition to have the flicker bars rolling. For this reason the preview mode frame rate is typically deliberately set not to exactly match 15 fps or 30 fps. However, in capture mode, the frame rate is set exactly, assuming that the flicker has already been detected and compensated during preview.

## JPEG Driver

The JPEG driver programs Huffman table and quantization table memories, sets up the MT9D111 to output JPEG compressed data, and handles error checking and handshaking with the host processor.

### Usage

JPEG is enabled using mode.mode_config. For example, to enable compression for capture set mode.mode_congif=16. jpeg.qscale1/2/3, and specify the quantization factor to control compression ratio. Bigger number indicates more compression. mode.fifo* configure spoof and non-spoof output and specify FIFO output clock divider. If necessary, use jpeg.config for error handshaking with the host.

### Table Programming

At power up initialization, the JPEG driver loads standard Huffman tables into Huffman memory. Scaled versions of standard luma and chroma quantization tables and are loaded into quantization memory. The calculation of the scaled quantization table is as follows:

Scaled_Q = (scaling_factor * standard_Q + 16) >> 5

Scaling factor is bit 6:0 of JPEG driver variables qscale1, qscale2, and qscale3. The standard quantization and Huffman tables are Tables K.1–K.4 of the ISO/IEC 10918-1 Specification. Host processor can override Huffman and quantization memory with any arbitrary Huffman and quantization tables through indirect register access.

If the host chooses to use the scaled standard quantization tables, bit 4 of JPEG driver variable config must be set to "1." Scaling factor can be changed at any time. Whenever it is changed, bit 7 of the corresponding JPEG driver variable qscale must be set to "1," and the new value takes effect in the next frame JPEG encoding.

Since the quantization memory stores 3 sets (luma and chroma) of quantization tables, the one that is used for the current frame JPEG encoding is indicated in bit 7:6 of jpeg.config (quantization table ID). Bit 5 of jpeg.config determines who is responsible for setting the quantization table ID. If it is "0," the host processor must program quantization table ID for every JPEG compressed frame (no need to reprogram it if subsequent JPEG frames use the same quantization tables). If it is "1," the JEPG driver sets quantization table ID to "0" at the start of JPEG capture (be it still or video) and automatically select next set of quantization tables for the subsequent frames when the current JPEG frame is unsuccessful.

Bit 7:6 of jpeg.config = 0, 1, and 2 indicates first, second, and third set of quantization tables, respectively.

### Error Handling and Handshaking

When the capturing of a JPEG snapshot is unsuccessful, the JPEG driver can be configured to enable encoding subsequent frames until it is successful by setting bit 2 of jpeg.config to "1." For capturing JPEG video, MT9D111 always encodes subsequent frames no matter what value bit 2 of jpeg.config is set to.

If bit 1 of jpeg.config is "1," handshaking with the host processor at every erroneous JPEG frame is required. When JPEG status register indicates that there is an error in the current JPEG frame, JPEG encoding is stopped until the host processor sets bit 3 of jpeg.config to "1" to indicate it is ready to receive next JPEG frame. If the host processor does not respond to an erroneous JPEG frame within jpeg.timeoutFrames frames, JPEG capture is terminated. If bit 1 of jpeg.config is "0" or if there is no error in JPEG status register, no handshaking is required.

The handshaking mechanism is provided so that the host processor has sufficient time to handle the erroneous JPEG frame and react upon the error condition. For example, if the JPEG status register indicates FIFO overflow, the host processor should increase quantization value by changing quantization table scaling factor or selecting another set of the preloaded quantization tables. If spoof oversize error occurs, the host processor should increase the spoof frame size by programming registers R16 and R17 on IFP Register Page 2 and/or increase quantization value.

## Start-Up and Usage

The start-up sequence consists of the following:

1. Power-up
2. Hardware reset
3. Configure and enable PLL
4. Configure pad slew rate
5. Configure preview mode
6. Configure and enable auto focus

7. Configure capture mode
8. Perform lock or capture

To start the part, power up power supplies, provide an input clock, and perform a hardware reset.

## Power-Up

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 ±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) sets 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.

A hard reset sequence to the camera can be activated by the following steps:

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

## Soft Reset Sequence

A soft reset to the camera can be activated by the following procedure:

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 MT9D111 registers—both page 1 and page 2—is possible during soft reset.

## Enable PLL

Since the input clock frequency is unknown, the part starts with PLL disabled. The default MNP values are for 10 MHz. For other frequencies, calculate and program appropriate MNP values. PLL programming and power-up sequence is as follows:

1. Program PLL frequency settings, R0x66-67:0
2. Power up PLL, R0x65:0[14] = 0

3. Wait for PLL settling time >100µs–150µs
4. Turn off PLL bypass, R0x65:2[15] = 0

Allow one complete frame to effect the correct integration time after enabling PLL.

**Note:** Until PLL is enabled the two-wire serial interface is slow. After PLL is enabled, the two-wire serial interface master can increase its communication speed.

## Configure Pad Slew

Program with desired slew rate for DOUT, PIXCLK, FRAME_VALID, and LINE_VALID. Program R10:1 with desired GPIO slew rate and slew rate for two-wire serial interface SDATA and SCLK

## Configure Preview Mode

The default preview image size is 800 x 600, running at up to 30 fps at 80 MHz internal clock. To change the default size, program mode driver variables mode.output_width_A and mode.output_height_A and issue a REFRESH command, seq.cmd=5.

For example, to configure 160x120 LCD RGB preview, program
- mode.output_width_A=160
- mode.output_width_B=120
- mode.out_format_A=0x20
- seq.cmd=5

Preview contrast, brightness, gamma, frame rate, and many other parameters can be loaded here as well. If known at this time, the user can also program capture parameters. If necessary, set up the AE/WB/AF lock command here.

## Configure Capture Mode

When desired capture parameters are known (video, still, compression, and resolution), program the mode and other drivers correspondingly.

## Perform Lock or Capture

See sequencer driver for details.

## Standby Sequence

Standby mode can be activated by two methods. 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 100mA at 55°C. There is no serial interface access for hard standby.

The second method is activated through the serial interface by setting R13:0[2]=1 to the register, known as the soft standby. As long as the input clock remains on, the chip allows 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. Preparing 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. Preventing additional leakage current during standby
   a. Set R10:1[7]=1 to prevent elevated standby current. It controls 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 drives 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 are HIGH while some are LOW. For dual camera systems, at least one camera has to be driving the bus at any time so that the outputs are not 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 0x9078, 0x9079, 0x9070, and 0x9071 (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 has 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. Putting 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
3. If necessary, reconfigure the GPIOs back to the desired state by GPIO variables 0x9078 and 0x9079. Also set R10:1[7]=0 if any GPIOs are used as inputs.
4. Go to preview
5. Issue a GO_PREVIEW command to the firmware by setting seq.cmd=1
6. 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

## Standby Hardware Configuration

While in standby, floating IO signals may cause the standby current to rise significantly. Therefore, it is recommended that the following signals be maintained high or low during standby and not floating: GPIO[11:0], DOUT[7:0], PIXCLK, FRAME_VALID, and LINE_VALID.

## Output Enable Control

When the sensor is configured to operate in default mode, the DOUT, FRAME_VALID, LINE_VALID, and PIXCLK outputs can be placed in High-Z under hardware or software control, as shown in Table 18.

**Table 18: Output Enable Control**

| Standby | R0x0D:0[4] (output_dis) | R0x0D:0[6] (drive_pins) | Output State |
|---|---|---|---|
| 0 | 0 (default) | 0 (default) | Driven |
| 1 | 0 (default) | 0 (default) | High-Z |
| “Don’t Care” | 0 (default) | 1 | Driven |
| “Don’t Care” | 1 | “Don’t Care” | High-Z |

The pin transition between driven and High-Z always occurs asynchronously. Output-enable control is provided as a mechanism to allow multiple sensors to share a single set of interface pins with a host controller.

There is no benefit in placing the pins in a High-Z while the part is in its low-power standby state. Therefore, in single-sensor applications that use STANDBY to enter and leave the standby state, programming R0x0D:0[6] = 1 is recommended.

GPIO outputs can also be tri-stated. See “General Purpose I/O” on page 81 description for details.

## Power-Down

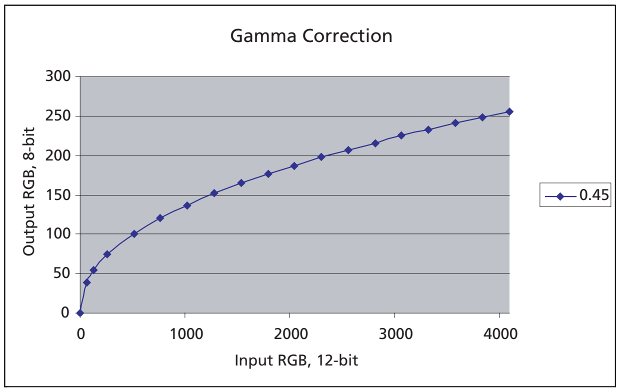
There are no specific requirements to power down the part.

## Contrast and Gamma Settings

The MT9D111 IFP includes a block for gamma and contrast correction. A custom gamma/contrast correction table may be uploaded, or pre-set gamma and contrast settings may be selected.

The gamma and contrast correction block uses the following 12-bit input data points to form a piecewise linear transformation curve: 0, 64, 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, and 4095. These input points have been selected to provide more detail to the low end of the curve where gamma correction changes are typically the greatest. These points correspond to 8-bit output values that can be uploaded to the appropriate registers.

**Figure 33: Gamma Correction Curve**



For simplicity, predefined gamma and contrast tables may be selected, and the MT9D111 automatically combines these tables and upload them to the appropriate gamma correction registers.

The gamma and contrast tables may be selected at mode driver (ID = 7) offsets 67 and 68 (decimal) for mode A and mode B, respectively. The gamma settings are established at bits 0–2, and the contrast settings are established at bits 4–6.

The gamma setting values are shown in Table 19 on page 73.

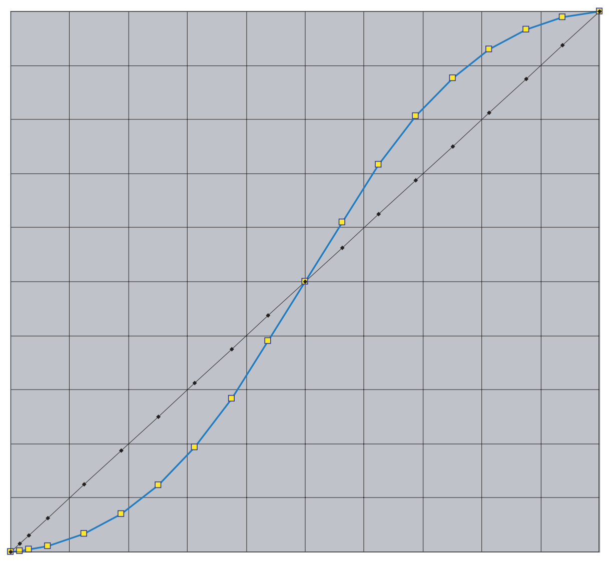
**Table 19: Gamma Settings**

| Gamma Setting | Definition |
|---|---|
| 0 | Gamma = 1.0 (no gamma correction) |
| 1 | Gamma = 0.56 |
| 2 | Gamma = 0.45 |
| 3 | Use user-defined gamma table |

## S-Curve

The predefined contrast table values have been established by creating an "S" curve with highlight and shadow regions that blend smoothly with a linear midtone region. The slope and value of the highlight and shadow regions match the linear region at these transitions. In addition, the slope of the "S" curve is zero at the top (white) and bottom (black) points. The slope of the linear region determines how much contrast is applied; more contrast corresponds to a higher, midtone linear slope.

**Figure 34: Contrast "S" Curve**



The contrast values are shown in Table 20.

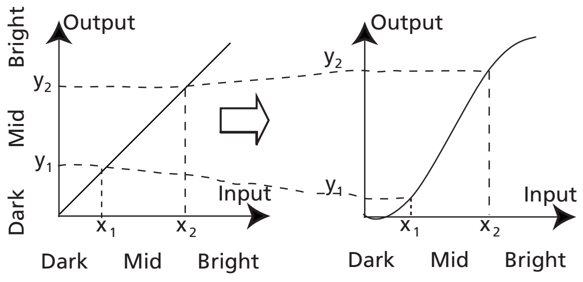
**Table 20: Contrast Values**

| Contrast Setting | Definition |
|---|---|
| 0 | No contrast correction |
| 1 | Contrast slope = 1.25 |
| 2 | Contrast slope = 1.50 |
| 3 | Contrast slope = 1.75 |
| 4 | Noise reduction contrast |

The contrast curve function is applied to the gamma curve points used (whether the gamma curve points are predefined or user-uploaded).

S-curve is a function to correct image pixel values. When applied to pixel values, it typically compresses dark and bright tones, while stretching the midtones. Figure 35 shows how input tone range is remapped to output tone range. Suppose we categorize input pixel values as dark, midlevel, and bright. When no S-curve is applied, pixel values are unchanged. That is, dark tones 0...$x_1$ map to same dark tones 0...$x_1$=$y_1$, and midlevel maps to identical midlevel $x_1$=$x_2$, and bright maps to identical bright. When an S-curve is applied, the mapping is changed. In particular, midtones are stretched $(y_1-y_2)>(x_1-x_2)$, causing increase of contrast. Here $(y_1-y_2)/(x_1-x_2)$ is a measure of contrast. Value of 1 corresponds to no change; >1 and <1 indicate contrast increase and decrease, respectively. Dark tones are compressed, $y_1<x_1$, causing suppression of noise.

**Figure 35: Tonal Mapping**



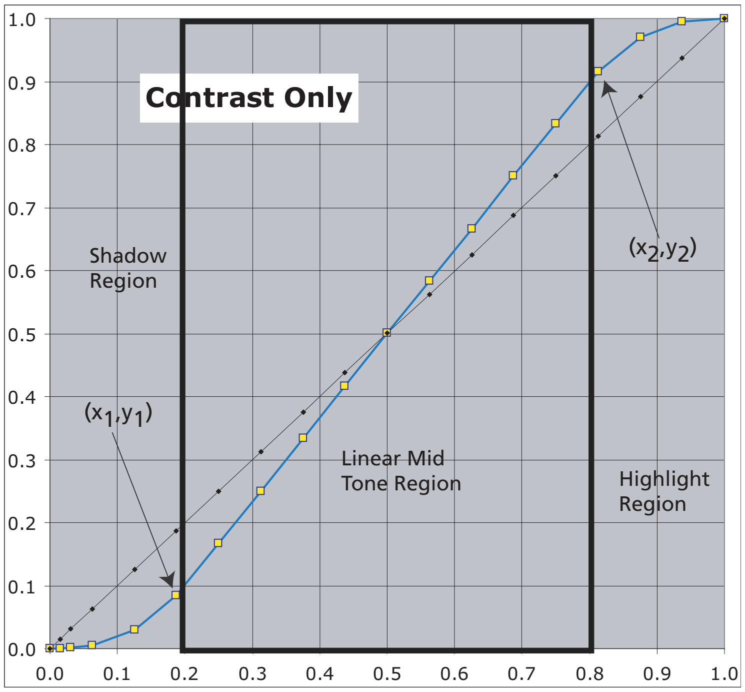
The contrast settings on the MT9D111 are implemented by applying an S-Curve function on to the data points of the gamma curve. These resulting points then replace the existing gamma curve points, and the image is processed using this new contrast-enhanced gamma curve. Identical S-curve is applied to all three RGB components.

The S-curve is created by joining a linear midsection (often with a slope greater than one) to curved sections for highlights and shadows. The following constraints apply to the overall curve to ensure a smooth curve appropriate for increased contrast:

1. The first/lowest point must be (0,0) and the last/highest point must be (1,1) (normalized).
2. The midsection and each of the end-curves must intersect on the same points.
3. The slope of the midsection and each of the end-curves must be equal at the intersection points.

The midsections a simple linear function of the form: $y = mx + b$. Each of the end-curves (shadow and highlights) must be a trinomial to satisfy all boundary conditions.

**Figure 36: Contrast Diagram**



## Auto Exposure

Two types of auto exposure are available—preview and evaluative.

### Preview

In preview AE, the driver calculates image brightness based on average luma values received from 16 programmable equal-size rectangular windows forming a 4 x 4 grid. In preview mode, 16 windows are combined in 2 segments: central and peripheral. Central segment includes four central windows. All remaining windows belong to peripheral segment. Scene brightness is calculated as average luma in each segment taken with certain weights. Variable ae.weights[3:0] specifies central zone weight, ae.weights[7:4] - peripheral zone weight.

The driver changes AE parameters (IT, Gains, and so on) to drive brightness (ae.CurrentY) to programmable target (ae.Target). Value of one step approach to target is defined by ae.JumpDivisor variable. Expected brightness is

Ynew = ae.CurrentY+(ae.Target-ae.CurrentY)/ ae.JumpDivisor.

To avoid unwanted reaction of AE on small fluctuations of scene brightness or momentary scene changes, the AE driver uses temporal filter for luma and gate around AE luma target. The driver changes AE parameters only if buffered luma outsteps AE target gates. Variable ae.lumaBufferSpeed defines buffering level.

32*Ybuf1=Ybuf0*(32-ae.lumaBufferSpeed)+Ycurr* ae.lumaBufferSpeed;

Values ae.lumaBufferSpeed=32 and ae.JumpDivisor=1 specify maximal AE speed.

### Evaluative

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 most difference for backlight and bright outdoor conditions.

### Exposure Control

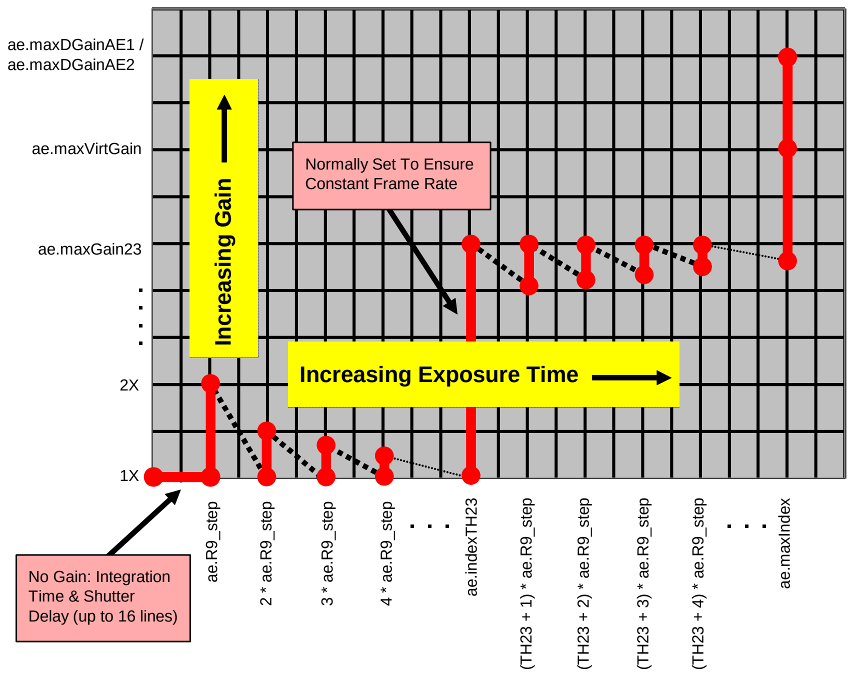
To achieve the required amount of exposure, the AE driver adjusts the sensor integration time R9:0, R12:0, gains, ADC reference, and IFP digital gains. To reject flicker, integration time is typically adjusted in increments of ae.R9_step. ae.R9_step specifies duration in row times equal to one flicker period. Thus, flicker is rejected if integration time is kept a natural factor of the flicker period.

Exposure is adjusted differently depending on illumination situation.

- In extremely bright conditions, the exposure is set using R12:0, R9:0 and analog gains. R12:0 is used to achieve very short integration times. In this situation, R9:0<ae.R9_step and flicker are not rejected.
- In bright conditions where R9:0>=ae., R9_step R9:0 is set as a natural factor of ae.R9_step. Analog gains are also used, but the green gain, also called virtual gain, does not exceed 2x. ae.minVirtGain limits minimal integration time and is expressed in flicker periods. ae.Index indicates the current integration time expressed in the same form.
- Under medium-intensity illumination, the integration time can increase further. For any given exposure, the best signal-to-noise ratio can be typically obtained by using the longest exposure and the smallest gain setting. However, a long exposure time can slow down the output frame rate if the former exceeds the default frame rate, R9:0 > R3:0 + R6:0 + 1. Integration ae.IndexTH23 specifies the breakpoint where AE scheme, giving preference to increasing the shutter width, is replaced with another scheme giving preference to increase in gain. ae.maxGain23 specifies maximum allowed gain in this situation. ae.VirtGain indicates current green channel gain.
- In darker situations, the gain achieves ae.maxGain23 and the integration time is allowed to increase again up to ae.maxIndex.
- In yet darker situations, once the integration time achieves ae.maxIndex, the analog gain is allowed to increase up to ae.maxVirtGain.
- In very dark conditions, the digital IFP gains are allowed to increase up to ae.maxDGainAE1 and ae.maxDGainAE2.

ADC is used as an additional gain stage by adjusting reference levels. See ae.ADC* variables.

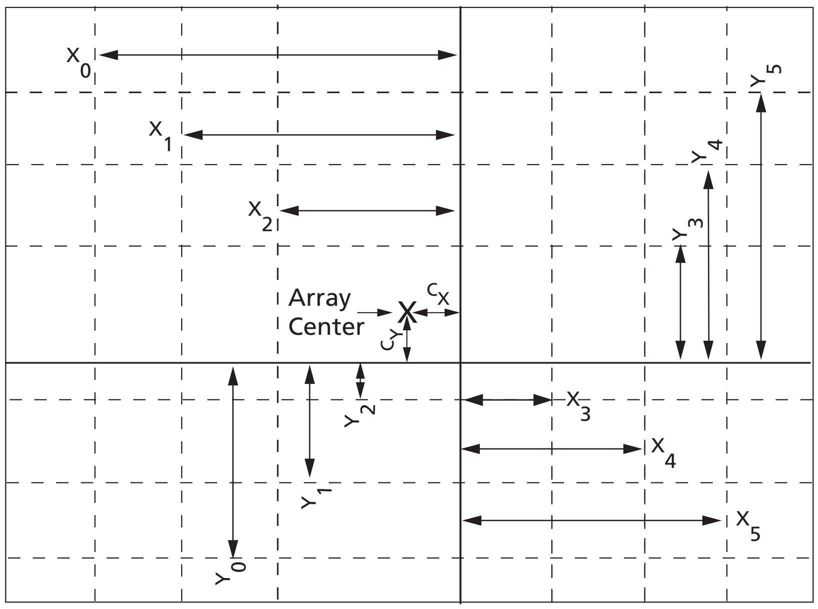
**Figure 37: Gain vs. Exposure**



## Lens Correction Zones

In order to increase the precision of the correction function, the image plane is divided into 8 zones in each dimension. The coordinates of zone boundaries are referenced with respect to the lens center, C. Each boundary as well as C (Cx, Cy) coordinate is stored as a byte, which represents the coordinate value divided by 4. There always three boundaries to the left (top) of the center and three to the right (bottom) of the center. These boundaries apply uniformly for each color channel. However, the correction functions are programmable independently for each color component. Boundary and lens center positions are also valid for the preview mode. Figure 38 illustrates the lens correction zones.

**Figure 38: Lens Correction Zones**



## Lens Correction Procedure

The goal of the lens shading correction is to achieve a constant sensitivity across the entire image area after the correction is applied. In order to accomplish this, each incoming pixel value is multiplied with a correction function F (x, y), which is dependent on the location of the pixel. The corrected pixel data, $P_{OUT}$, can be expressed in the following term:

$$P_{OUT}(x, y) = P_{IN}(x, y) * F(x, y) \qquad (1)$$

Within each zone described above, the correction function can be expressed with a following equation, as follows:

$$F(x,y) = \phi(x,x^2) + \varphi(y,y^2) + k * \phi(x,x^2) * \varphi(y,y^2) - G \qquad (2)$$

where

$\phi(x,x^2)$ and $\varphi(y,y^2)$

are piecewise quadratic polynomial functions that are independent in each x and y dimension

k and G are constants that can be used to increase lens correction at the image corners (k) and to offset the LC magnitude for all zones and colors (G)

The expressions

$\phi(x, x^2)$ and $\varphi(y, y^2)$

and are defined independently for each dimension. These can be expressed further as

$$\phi(x, x^2) = a_i x^2 + b_i x + c_i$$

$$\varphi(y, y^2) = d_i y^2 + e_i y + f_i$$

In order to implement the function F (x, y) for each zone the MT9D111 provides a set of registers that allow flexible definition of the function F (x, y). These registers contain the following parameters:

- operation mode R128:2
- zone boundaries and center offset R129:2-R135:2
- initial conditions of

  $\phi(x, x^2)$ and $\varphi(y, y^2)$

  for each color (12-bit wide) R136:2-R141:2
- initial conditions of the first derivate of

  $\phi(x, x^2)$ and $\varphi(y, y^2)$

  for each color (12-bit wide) R142:2-R147:2
- second derivatives of

  $\phi(x, x^2)$ and $\varphi(y, y^2)$

  for each color and each zone (8-bit wide) R148:2-R171:2
- values for k(10-bit wide) and G(8-bit wide) in (2) for all colors and zones are specified in R173:2 and R174:2. Sign for k is specified in R R128:2.

There are x2 factor that can be applied to the second derivatives inside each zone (R172:2) if correction curvature in the particular zone is to be increased. There are also global x2 factors for all zones in X and Y direction located in R128:2. The first derivative (for both X and Y directions) can be divided by a number (devisor) specified in R128:2 before it is applied to the F function. Higher numbers result in more precise curve (full-scale expanse).

## Color Correction

Color correction in the color pipeline is achieved by

1. Apply digital gain to raw RGB, R106:110:1
2. Subtract second black level D from raw RGB, see R59:1
3. Multiply result by CCM, R96-102:1
4. Clip the result

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \text{CLIP}\left( \begin{vmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{vmatrix} \bullet \begin{vmatrix} R_{raw} * G_R - D \\ G_{raw} * G_G - D \\ B_{raw} * G_B - D \end{vmatrix} \right)$$

MCU controls both CCM and D; see AWB and histogram driver description.

## Decimator

In order to fit image size to customer needs, the image size of the SOC can be scaled down. The decimator can reduce image to arbitrary size using filtering. The scale-down procedure is performed by transferring an incoming pixel from the image space into a decimated (scaled down) space. The procedure can be performed in both X and Y dimension. All the standard formats with resolution lower than 2 megapixels as well as customer specified resolutions are supported. Transfer of pixel from the image into the decimated space is done in the following way, which is the same in X and Y directions:

Each incoming pixel is split in two parts. The first part (P1) goes into the currently formed output-space pixel while part two (P2) goes to the next pixel. P1 could be equal or less than value of the incoming pixel while P2 is always less. These two parts are obtained by multiplying the value of incoming pixel by scaling factors f1 and f2, which sum is always constant for the given decimation degree and proportional to X1/X0 where X1 is size of the output image and X0 is the size of the input image. It is denoted as "decimation weight." Coefficients f1 and f2 are calculated in the microcontroller transparently for user based on the specified output image size and mode of SOC operation. At large decimation degrees, several incoming pixels may be averaged into one decimated pixel. Averaging of the pixels during decimation provides a low pass filter, which removes high-frequency components from the incoming image, and thus avoids aliasing in the decimated space. The decimator has two operational modes—normal and high-precision. Since the intermediate result for Y decimated pixels has to be stored in a memory buffer with certain word width, there is a need for additional precision at larger decimation degrees when scaling factors are small. This is done by increasing the number of digits for each stored value when decimation is greater than 2.

# General Purpose I/O

## Introduction

Actuators used to move lenses in AF cameras can be classified into several categories that differ significantly in their requirements for driving signals. These requirements vary also from one device to another within each category. The MT9D111 has been designed to meet the needs of many different lens actuators without having a lot of hardware resources dedicated solely to that purpose. Internal resources that MT9D111 brings together to control a lens actuator can be, and at least in part are, used for many other tasks, which amply justifies their presence on the chip even when no AF support is required from it. This is particularly true of the embedded 6811 microcontroller (MCU) with associated memory, but also of the general purpose input/output module (GPIO), whose description is given below.

The GPIO is, in essence, a programmable digital waveform generator with 12 individually controllable output pads (GPIO0 through GPIO11), a separate power supply pad (VDDGPIO), and a separate clock domain. By default, the GPIO clock domain is connected to the master clock, but it can be disconnected to save power, by writing 0 to R11:1[7]. The maximum rate at which the GPIO outputs can be toggled is 1/2 of the master clock frequency. In other words, when the master clock frequency is 80 MHz, the GPIO can change the state of its output pads every 25ns. This maximum rate is attainable only when the GPIO outputs pre-programmed periodic waveforms—a discussion of these and of differently generated arbitrary output patterns follows in the next two sections.

Since some lens actuators provide feedback signals that can be used to ascertain their position, direction of motion, etc., the MT9D111 has the capability to sense such signals via digital and analog inputs. All of the GPIO output pads can be individually reversed to become high-impedance digital inputs. Section “Digital and Analog Inputs” on page 144 explains how to change the direction of the GPIO pads and how to sense both digital and analog input signals.

The GPIO is programmed via 77 8-bit registers mapped to hexadecimal addresses 0x1070–0x10B6 and 0x10B8–0x10BD in the memory space of the MCU. Sixty-five of these registers and two additional registers with hexadecimal addresses 0x10BE and 0x10BF return information about GPIO status when read. Like driver variables, the GPIO registers can be accessed by an external host processor through registers R198:1 and R200:1. Since the two-wire serial interface is relatively slow, the host processor can never access the GPIO registers as fast as the embedded MCU. In GPIO applications requiring exact output timing and/or fast response to input signals, all time-critical writes and reads to the GPIO registers should be done by the MCU. If the only task of the GPIO is to move an AF lens, Aptina strongly recommends leaving the control of the GPIO entirely to AFM driver (ID = 6), a part of MT9D111 firmware dedicated to that task. MT9D111 users have the option to substitute their own code for the AFM driver or part of it, if it does not meet their needs.

## GPIO Output Control Overview

There are two ways to control voltages on the GPIO output pads. One way, direct but not guaranteeing high timing precision, is to set or clear bits in GPIO_DATA_L and GPIO_DATA_H registers. Both the MCU and the host processor can do it by writing to the hexadecimal addresses 0x1070–0x1077. The first two of these provide normal, "what-you-write-is-what-you-get" access to the registers, while the remaining six provide selective access to individual register bits. This selective access facilitates changing voltages on some GPIO output pads without affecting the state of other pads—an output manipulation expected to be routine. For example, to change voltage on the output pad GPIO3, the user only has to write 8 to the address 0x1073 (also known as register GPIO_OUTPUT_TOGGLE_L). To do the same by writing to the register GPIO_DATA_L, the user must know in advance the state of its third bit and all other bits corresponding to output pads.

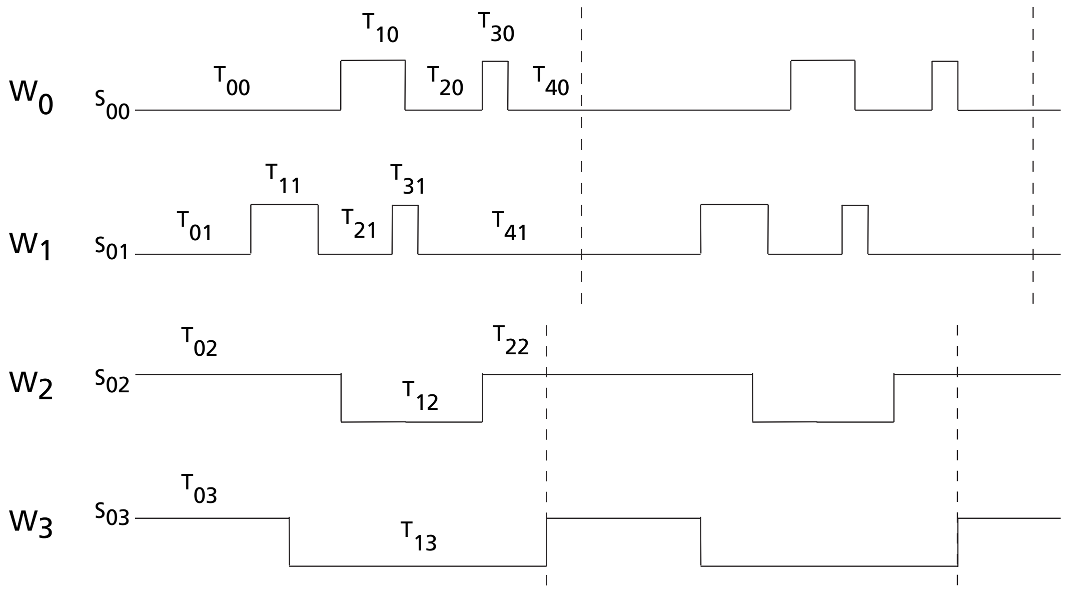
In general, writing a positive number to one of the GPIO_OUTPUT_* registers has the following effect: the GPIO output pads corresponding to ones in the binary representation of the number are toggled, set, or cleared, while the output pads corresponding to zeros and all input pads are left alone (masked). Once the GPIO outputs and the GPIO_DATA_* registers are updated, all bits in the GPIO_OUTPUT_* registers are automatically cleared.

The second way to obtain a desired output from the GPIO is to program into it a set of periodic waveforms, initialize their generation, and optionally monitor its progress. The advantage of using this way is that the GPIO, once programmed and activated, generates the desired waveforms on its own, without waiting for any external stimuli, and therefore with the best attainable timing accuracy. It is possible to override, suspend or abort this autonomous waveform generation by writing to appropriate GPIO registers, but no register writing is necessary for the GPIO to continue. If necessary, the GPIO can notify the MCU about reaching certain points on the waveform generation timeline, e.g., the first rising edge or end of a selected waveform. See "Notification Signals" on page 84 for more details.

## Waveform Programming

A large subset of the GPIO registers is dedicated to programming periodic waveforms. In designing this subset, the main concern has been to meet the requirements of the most demanding AF actuators known to date. All registers belonging to this subset have names starting with GPIO_WG_, where WG stands for waveform generator. The GPIO_WG_* registers allow one to individually specify up to 8 waveforms, to be output through the GPIO[7:0] pads. It is not possible to output preprogrammed waveforms through the GPIO[11:8] outputs. These outputs are controlled only by the GPIO_DATA_H and GPIO_OUTPUT_*_H registers.

**Figure 39: Examples of GPIO-Generated Waveforms**



Of the four waveforms depicted in Figure 39, the first two are examples of the most complex waveforms that the GPIO can generate. Periods of these waveforms consist of five different time intervals (subperiods), the first four of which end with a transition to an opposite state (LOW-to-HIGH or HIGH-to-LOW). Each waveform, $W_p$ (p = 0, 1), is completely described by its initial state, $S_{0p}$, the lengths of its five subperiods, $T_{ip}$ (i = 0,…, 4), and the number of periods (repetitions) from the start to the end, $N_p$. The simpler waveforms W2 and W3 can be described using the same set of numbers, only with some of the subperiod lengths equal to 0. A valid waveform description must include $N_p > 0$ and at least one $T_{ip} > 0$, (i.e. it must set a nonzero duration for the waveform). Just one $T_{ip} > 0$ gives a constant function of time, $W_p(t) = S0p$. Generating such a waveform means keeping a particular GPIO output at LOW or HIGH for the specified time. To toggle an output between LOW and HIGH, one has to assign to it at least two nonzero $T_{ip}$ values.

The $S_{0p}$ values are set in the GPIO_DATA_L register. There are two ways to store the $N_p$ and $T_{ip}$ values in the GPIO registers GPIO_WG_N* and GPIO_WG_T*. The first is to allocate 8 bits for each value, which allows one to store values for eight waveforms. The second is to allocate 16 bits per value. When this is done across the board, one can define only up to four waveforms, to be generated at the GPIO0, GPIO2, GPIO4, and GPIO6 pads. However, the GPIO allows users to select “8-bit counter mode” or “16-bit counter mode” individually for each of the following pairs of outputs: GPIO[1:0], GPIO[3:2], GPIO[5:4], and GPIO[7:6].

These pairs are put in the 8- or 16-bit counter mode by setting bits [4:7] in register GPIO_WG_CONFIG to 0 or 1, respectively. The term counter mode is used here because these bits control the width of counters used in waveform generation. What is described above as allocating 8 or 16 bits per $N_p$ or $T_{ip}$ value in the GPIO_WG_N* and GPIO_WG_T* registers, is in fact done by changing counter widths and the way register values are loaded into the counters. Also, strictly speaking, there is a proportionality, not equality, relation between $T_{ip}$ values and single or coupled GPIO_WG_T* register settings that represent them. Hence, the statement that $T_{ip}$ values are stored in 8-bit or16-bit "cells" is a bit inaccurate. Perhaps we should say "encoded" instead of "stored."

In any case, the $N_p$ values occupy up to 8 registers (GPIO_WG_N0 through GPIO_WG_N7). Writing an invalid $N_p = 0$ to any of these registers is interpreted as setting this particular $N_p$ to infinity. The $T_{ip}$ values are fully encoded in 42 registers named GPIO_WG_T*, GPIO_WG_CLKDIV, and GPIO_WG_CLKDIV_SEL. This last two registers contain, respectively, two 4-bit settings for two clock dividers and eight 1-bit switches assigning one or the other divider to each of the GPIO[7:0] pads. Each clock divider divides the GPIO clock frequency by $2^{d+1}$, where d is its 4-bit setting. The GPIO_WG_T* registers contain natural numbers obtained by dividing the $T_{ip}$ values by master clock period and by the appropriate power of 2. For example, suppose that the master clock period is 12.5 ns. If one decides to use the registers (and counters) in the 16-bit mode and sets the clock divider for the GPIO2 output to $2^{2+1} = 8$, one has to write 39 to GPIO_WG_T03 and 16 to GPIO_WG_T02 to get $T_{02} = 1$ ms. To obtain approximately the same $T_{02}$ in the 8-bit mode with GPIO2 clock divider set to $2^{8+1} = 512$, one has to set the GPIO_WG_T02 to 156 (since 1ms/12.5ns/512 = 156.25).

Bit values in registers GPIO_WG_FRAME_SYNC, GPIO_WG_STROBE_SYNC and GPIO_WG_CHAIN determine the conditions that must be met for waveform generation to begin on each of the GPIO[7:0] pads. It always has to be enabled first by clearing an appropriate bit in GPIO_WG_SUSPEND register. Depending on selections made in the GPIO_WG_*_SYNC and GPIO_WG_CHAIN registers, the enabling is the signal to either start waveform generation immediately or on one of the following events: next falling edge of FRAME_VALID, next raising edge of STROBE or end of waveform generation on another pad. No more than one of these events should be chosen to trigger waveform generation on each particular pad. If more than one event is selected, only selection made in the highest priority register has an effect. The order of priority is, from highest to lowest, GPIO_WG_CHAIN, GPIO_WG_FRAME_SYNC, GPIO_WG_STROBE_SYNC.

Waveform generation at any pad can be suspended and resumed at will using the GPIO_WG_SUSPEND register. Suspending is like stopping the time on a particular pad. If generation on other pads continues in the meantime, synchronization between the suspended waveform and others are lost. Register GPIO_WG_RESET contain reset bits for the GPIO[7:0] outputs. Setting any of them to 1 aborts ongoing waveform generation at the corresponding pad and resets the counters used in it. The bit must be cleared before the waveform generation can resume.

## Notification Signals

The GPIO can send signals to the MCU to notify it about two types of events: the end of waveform generation at a particular output pad (GPIO[7:0]) or a transition of interest (LOW-to-HIGH or HIGH-to-LOW) on a particular input or output pad (GPIO[11:0]).

The GPIO uses two kinds of notification signals in parallel: on each event of interest, it sends a wake-up signal to the MCU and it sets to 1 the appropriate bit in register GPIO_NS_STATUS_L or GPIO_NS_STATUS_H. The bit remains set until acknowledged by writing 1 to it. The wake-up signal has no effect unless the MCU is in sleep mode.

To enable notification signals from GPIO, some bits in register GPIO_NS_MASK_L and/or register GPIO_NS_MASK_H must be set to 0. By writing to the corresponding bits in registers GPIO_NS_TYPE, GPIO_NS_EDGE_L, and GPIO_N_EDGE_H, one can choose events that triggers the signals.

## Digital and Analog Inputs

All GPIO pads are configurable as high-impedance digital inputs. Setting or clearing bits in the GPIO_DIR_* registers turns the corresponding pads into outputs or inputs, respectively. The logical state of each input pad is mirrored by the state of the corresponding bit in the GPIO_DATA_* registers, enabling the MCU or external host processor to receive digital feedback.

One of the 10-bit ADCs in the MT9D111 sensor core is available to sample external voltage signals (0.1 to 1.0V) during horizontal blanking periods, when it does not digitize sensor signal. The external signals that need to be sampled must be connected to AIN1, AIN2, and/or AIN3 input pads. By default, these are test pads that give access to different points in the sensor analog signal chain. To connect them directly to the ADC and enable signal sampling during horizontal blanking, R0xE3[15] must be set to 1. When this bit is set, 10-bit values of digitized AIN3, AIN2, and AIN1 signals are put in registers R0xE0:0, R0xE1:0, and R0xE3:0, respectively. The maximum signal sampling rate provided by this scheme is about 36000 samples per second.

## GPIO Software Drivers

It is likely that some MT9D111 users wants to develop their own software for controlling the GPIO and, through it, their own devices. All MT9D111 firmware, in particular the AFM driver controlling the GPIO, has been designed to facilitate user modifications and additions. Public driver methods (functions) have been made easily replaceable by means of jump tables, also referred to as virtual method tables (VMT). Public firmware variables include pointers to these tables, which in turn include pointers to driver methods that users may want to replace with their own. To replace one or more public method of, say, the AFM driver, a user must do the following:

1. Load code containing replacement methods into the MCU RAM memory
2. Create in the RAM a copy AFM driver VMT to replace the original VMT
3. Populate the replacement VMT with pointers to the replacement methods and to those original methods that need not be replaced
4. Change the pointer to the AFM driver VMT included among the public variables of the driver so that it points to the replacement VMT.

The public VMTs also make it easy for the users to call firmware methods from their own code. Utilizing these methods may reduce user code size and shorten development time.

A detailed discussion of driver requirements of various lens actuators will be added to this document later.

## Auto Focus

### 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, 1-dimensional position variable, leaving the translation of its changes into physical lens movements to a separate AF 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.

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 pre-programmed 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 in addition divided into up to 16 equal-size 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 1 normalized sharpness score per AF window, by calculating for each window the ratio (S1+S2)/<Y>, where <Y> is the average Y and S1 and S2 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, for a reason that will soon become clear.

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 40. 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, so 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 their execution and determines how soon after each command the AF driver gets a signal to proceed. All inclined sections of the lens position plot in Figure 40 are therefore of unknown duration—unless the AF algorithm discussion is narrowed to a specific use case.

**Figure 40: Search for Best Focus**

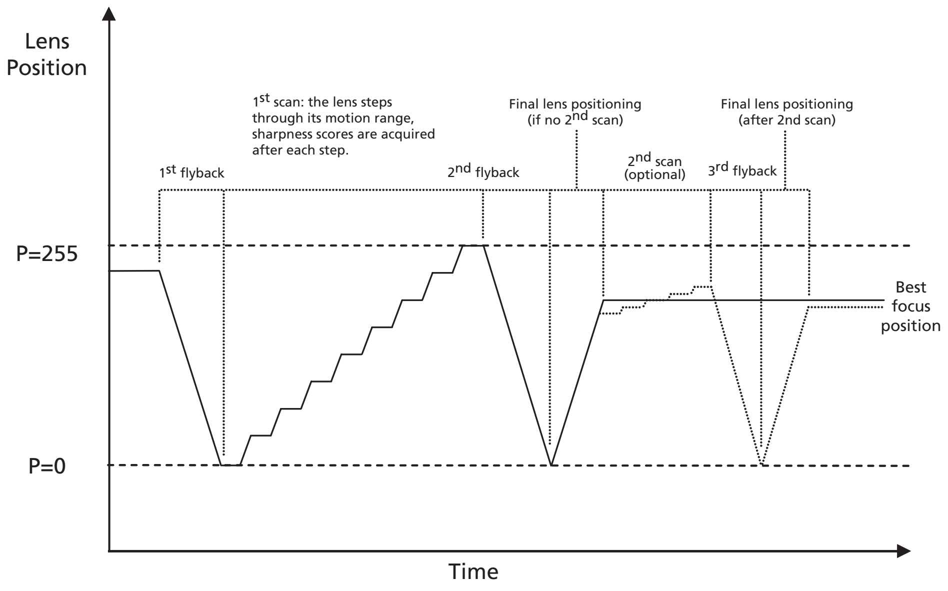

Lens Position
1st scan: the lens steps through its motion range, sharpness scores are acquired after each step.
Final lens positioning (if no 2nd scan)
Final lens positioning (after 2nd scan)
1st flyback
2nd flyback
2nd scan (optional)
3rd flyback
P=255
P=0
Best focus position
Time

Figure 40 shows lens movements during dual/triple-flyback auto focusing sequence. The depicted sequence is just an example and 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.

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 at the same time 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 and therefore its observed extrema are likely determined by random noise. On the other hand, if the score varies a lot with the lens position, its maximum is much more likely to be valid, i.e. 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 build 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, i.e. 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.

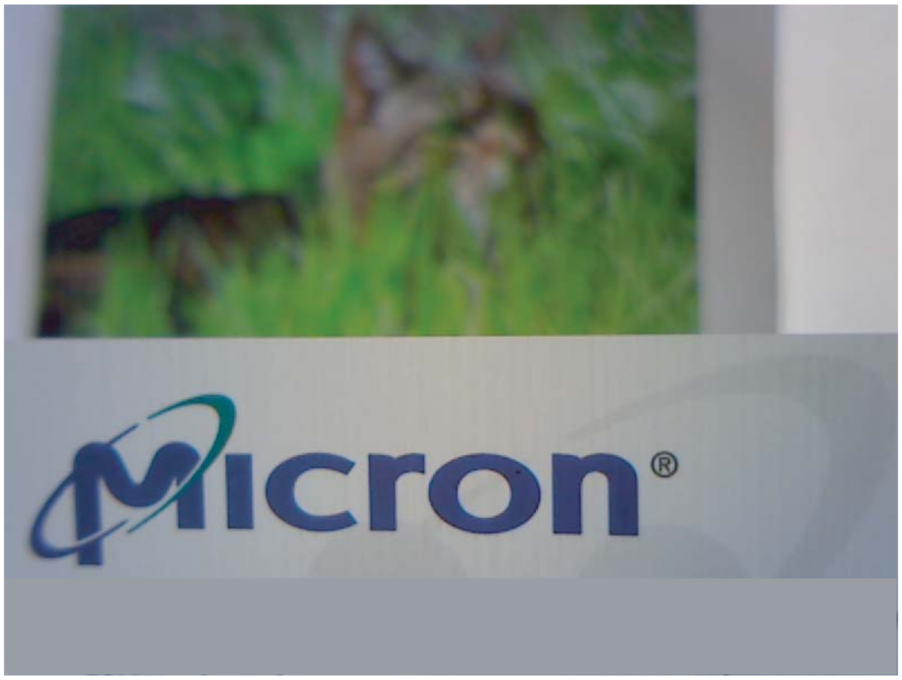
Figure 41, Figure 42 and Figure 43 illustrate selection of best lens position when there are several objects in imaged scene to focus on, each at a different distance from the lens. Each lens position bringing one or more of these objects into sharp focus within the AF window grid can be potentially 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 gives the user a number of ways to proceed with final lens positioning. The user should select a way 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, i.e. 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 pre-programmed 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 2 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 2 unidirectional scans can be also seen as a single scan with 2 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, e.g. a flyback to the start position of the latter and a jump to the position found best in the former.

Figure 41: Scene with Two Potential Focus Targets at Different Distances from Camera



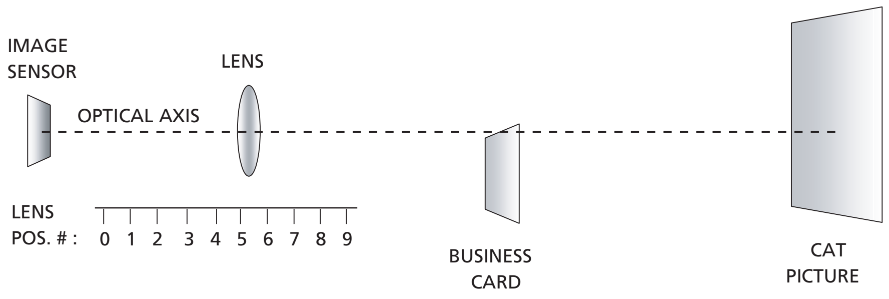


Figure 41 shows a simple scene with two potential focus targets: a business card in front and a picture of a cat far in the background. Distances in the diagram are not to scale. Red and yellow rectangles in the middle of the image represent 16 AF windows, each of which yields a separate Y-normalized 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. See Figure 43.

**Figure 42: Dependence of Luminance-Normalized Local Sharpness Scores on Lens Position**

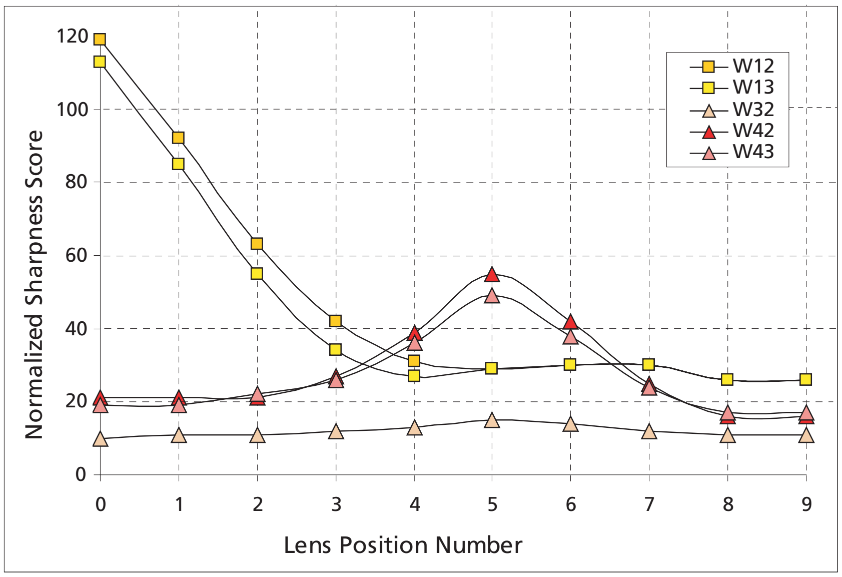


Figure 42 shows luminance-normalized sharpness scores from AF windows W12, W13, W32, W42, and W43 in Figure 41. Lens position 0 is the position of best focus for windows W12 and W13, while windows W42 and W43 are in sharpest focus at lens position 5. Relatively featureless window W32 is in sharpest focus at the same position, but it is difficult to determine this from its sharpness score, which varies very little with lens position.

**Figure 43: Example of Position Weight Histogram Created by AF Driver**

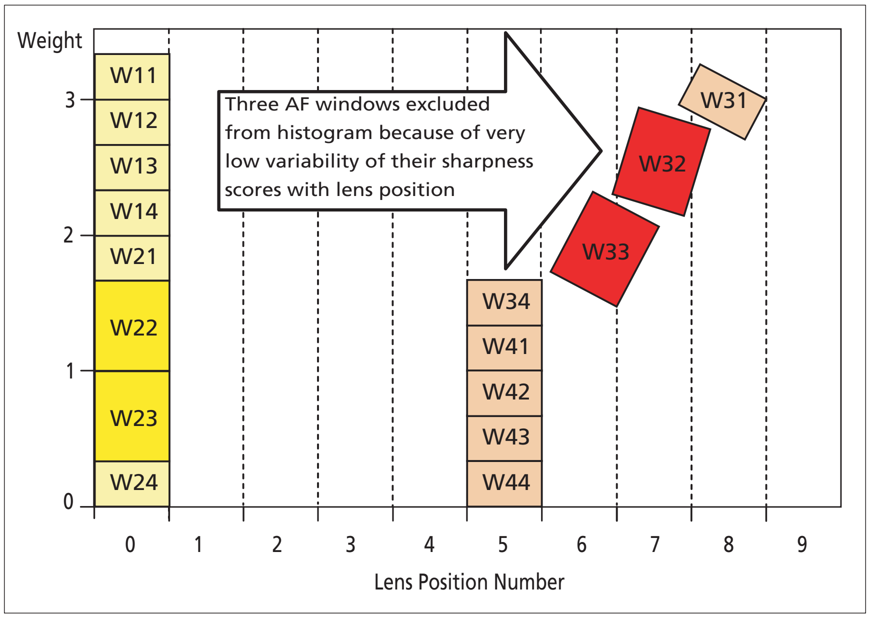

Weight
Three AF windows excluded from histogram because of very low variability of their sharpness scores with lens position
W11
W12
W13
W14
W21
W22
W23
W24
W34
W41
W42
W43
W44
W31
W32
W33
0
1
2
3
Lens Position Number
0
1
2
3
4
5
6
7
8
9

Figure 43 shows an example of position weight histogram created by AF driver to select best lens position. This histogram corresponds to the situation depicted in Figure 41 and Figure 42. After scanning 10 lens positions numbered 0 through 9, 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 2 positions, the AF driver summed preprogrammed weights of the AF windows being clearly in focus at that position, thus obtaining 2 position weights. These weights would have been equal if not 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 pre-programmed 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.

## 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 Image Flow Processor (IFP). The FME convolves 2 pre-programmed 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 size 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 [R64:66]: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 3x3 portion of the grid and place the remaining 7 AF windows almost entirely outside of it, as shown in Figure 44. Whenever a portion of an AF window is inside the frame, the FME calculates 2 sharpness scores and average 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 outputting 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. Thinking this way about the AF windows, refer to a portion of a frame row belonging to any AF window 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 2 AF filters. The AF filters are user-programmable within the following constraints: each can have 8 or 9 integer coefficients with values from -15 to 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 than 4 columns away from the left and right side of the frame.

**Figure 44: Auto Focus Windows**

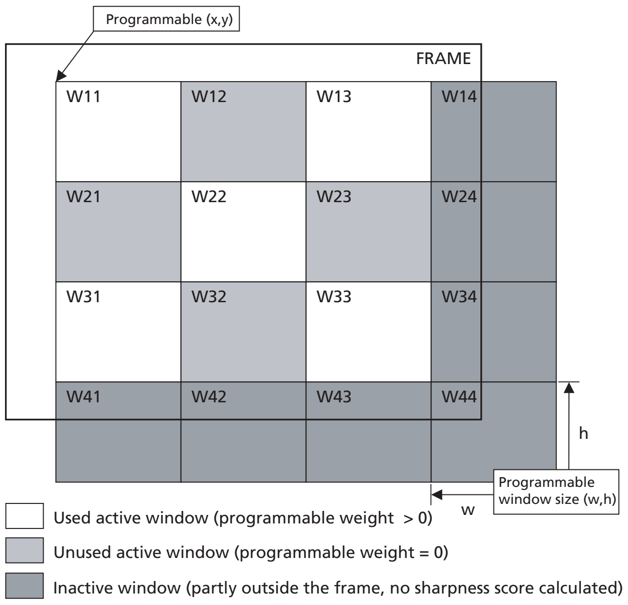

Programmable (x,y)
FRAME
W11
W12
W13
W14
W21
W22
W23
W24
W31
W32
W33
W34
W41
W42
W43
W44
h
w
Programmable window size (w,h)
Used active window (programmable weight > 0)
Unused active window (programmable weight = 0)
Inactive window (partly outside the frame, no sharpness score calculated)

Figure 44 shows an array of 16 equal-size AF windows configured to work like a centered quincunx pattern of 5 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 45, each AF window not located fully outside the frame yields 2 sharpness scores (the sums of convolution results from the 2 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 and only 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 44—hence these windows are inactive. Results from each active AF window are output immediately after its last row is processed.

**Figure 45: Computation of Sharpness Scores and Luminance Average for an AF Window**

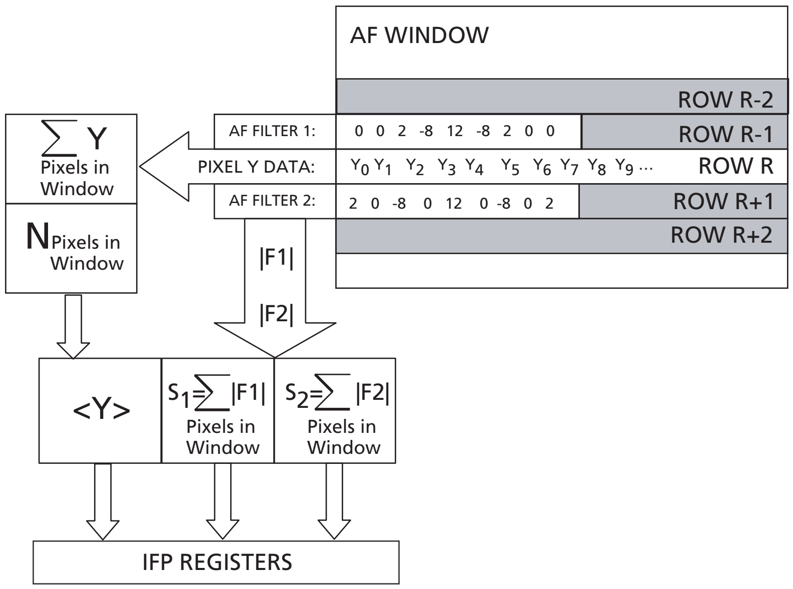


Figure 45 shows the computation of sharpness scores and average luminance in an AF window. Coefficients of the two AF filters are programmable. The filters shown here as an example yield convolution results $F_1 = 2Y_2 - 8Y_3 + 12Y_4 - 8Y_5 + 2Y_6$ and $F_2 = 2Y_0 - 8Y_2 + 12Y_4 - 8Y_6 + 2Y_8$.

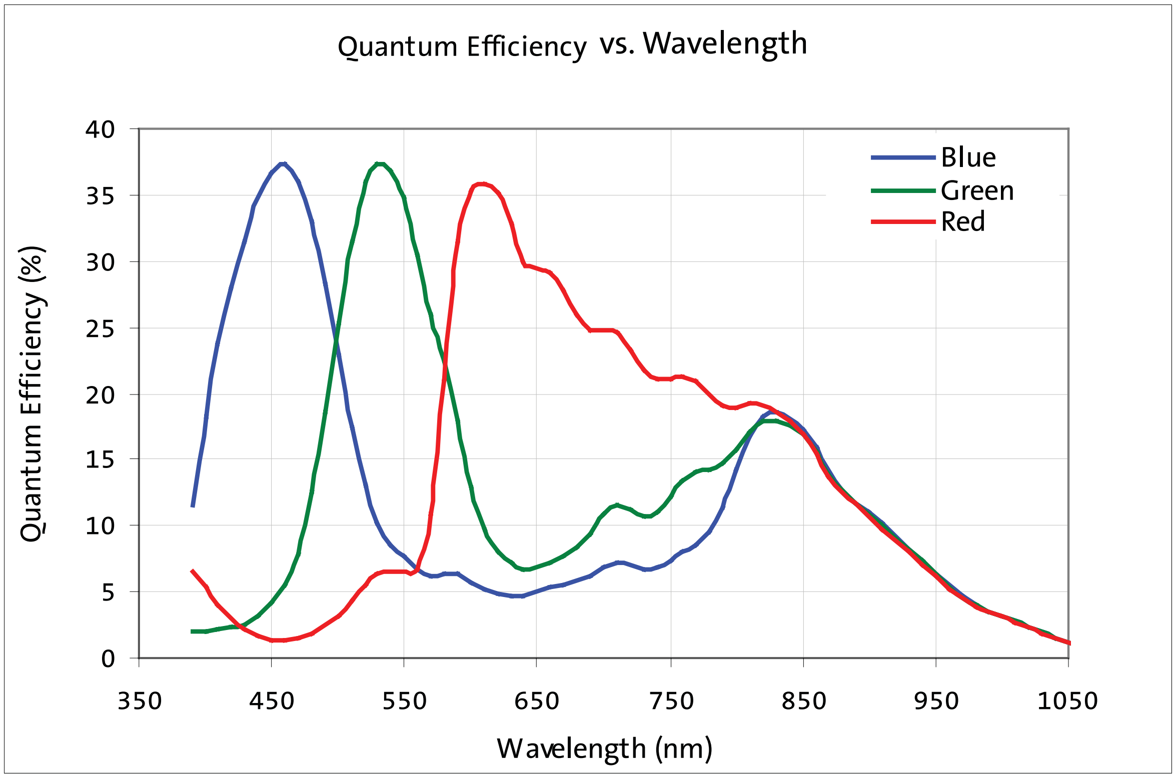
The symmetry constraint placed on the AF filters reduces the number of coefficient values needed to define them. Symmetric 8- or 9-coefficient filters are defined by specifying 4 or 5 coefficient values, respectively. Only 4 coefficients are needed to define an 8- or 9-coefficient antisymmetric filter. Examples of AF filters that can be programmed into MT9D111 are given in Table 21. To program the first AF filter, one must 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 21: Examples of AF Filters that can be Programmed into the MT9D111**

| Filter Parameters Programmed into Registers | | | | | | | | Filter |
|---|---|---|---|---|---|---|---|---|
| Filter Size | Filter Symmetry | Filter Coefficients | | | | | Filter Weight | |
| | | 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] |

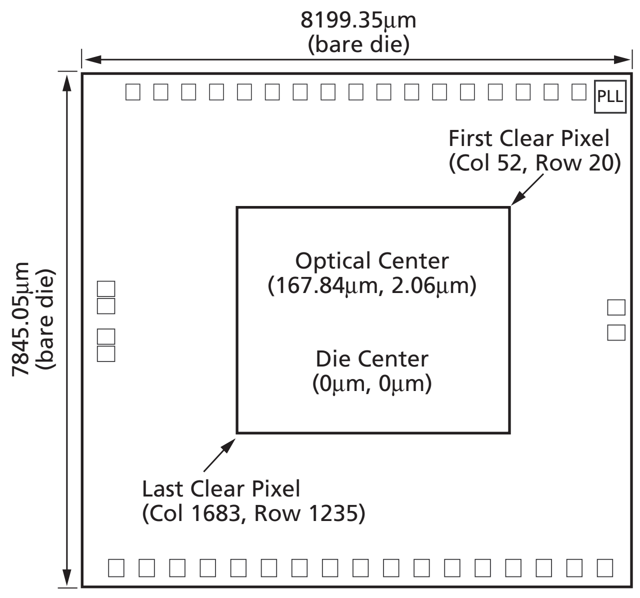
## Spectral Characteristics

**Figure 46: Typical Spectral Characteristics**



## Die Outline

**Figure 47: Optical Center Offset**



Note: Figure not to scale.

## Electrical Specifications

Recommended die operating temperature range is from -20° to +55°C. The sensor image quality may degrade above +55°C.

**Table 22: AC Electrical Characteristics**

| Symbol | Definition | Conditions | MIN | TYP | MAX | Units |
|---|---|---|---|---|---|---|
| $f_{CLKIN1}$ | Input clock frequency | PLL enabled (MCLK max = 80 MHz) | 6 | 10 | 64 | MHz |
| $t_{CLKIN1}$ | Input clock period | PLL enabled (MCLK max = 80 MHz) | 166.7 | 100 | 15.625 | ns |
| $f_{CLKIN2}$ | Input clock frequency | PLL disabled | 6 | | 80 | MHz |
| $t_{CLKIN2}$ | Input clock period | PLL disabled | 166.7 | | 12.5 | ns |
| $t_R$ | Input clock rise time | | 0.5 | | 1 | V/ns |
| $t_F$ | Input clock fall time | | 0.5 | | 1 | V/ns |
| | Clock duty cycle | | 40 | 50 | 60 | % |
| $f_{PIXCLK}$ | PIXCLK frequency | Default | | | | MHz |
| $t_{PD}$ | PIXCLK to data valid | Default | -3 | | 3 | ns |
| $t_{PFH}$ | PIXCLK to FV high | Default | -3 | | 3 | ns |
| $t_{PLH}$ | PIXCLK to LV high | Default | -3 | | 3 | ns |
| $t_{PFL}$ | PIXCLK to FV low | Default | -3 | | 3 | ns |
| $t_{PLL}$ | PIXCLK to LV low | Default | -3 | | 3 | ns |
| CIN | Input pin capacitance | | | 3.5 | | pF |
| CLOAD | Load capacitance | | | 15 | 20 | pF |
| AC Setup Conditions | | | | | | |
| | $f_{CLKIN1}$ | | 6 | | 64 | MHz |
| | VDD | | 1.7 | 1.8 | 1.95 | V |
| | VDDQ | | 1.7 | 2.8 | 3.1 | V |
| | VAA | | 2.5 | 2.8 | 3.1 | V |
| | VAAPIX | | 2.5 | 2.8 | 3.1 | V |
| | VDDPLL | | 2.5 | 2.8 | 3.1 | V |
| | Output load | | | 15 | | |

**Table 23: DC Electrical Definitions and Characteristics**

| Symbol | Definition | Conditions | MIN | TYP | MAX | Units | Note |
|---|---|---|---|---|---|---|---|
| VDD | Core digital voltage | | 1.7 | 1.8 | 1.95 | V | |
| VDDQ | I/O digital voltage | | 1.7 | 2.8 | 3.1 | V | |
| VDDGPIO | GPI/O digital voltage | | 1.7 | 2.8 | 3.1 | V | |
| VAA | Analog voltage | | 2.5 | 2.8 | 3.1 | V | |
| VAAPIX | Pixel supply voltage | | 2.5 | 2.8 | 3.1 | V | |
| VDDPLL | PLL supply voltage | | 2.5 | 2.8 | 3.1 | V | |
| VIH | Input high voltage | VDDQ = 2.8V | 2.4 | | VDDQ+0.3 | V | |
| | | VDDQ = 1.8V | 1.4 | | VDDQ+0.3 | | |
| VIL | Input low voltage | VDDQ = 2.8V | GND-0.3 | | 0.8 | V | |
| | | VDDQ = 1.8V | GND-0.3 | | 0.5 | | |
| IIN | Input leakage current | No pull-up resistor; VIN = VDDQ or DGND | -10 | +/-0.5 | 10 | µA | |
| VOH | Output high voltage | At specified IOH | VDDQ-0.4 | | | V | |
| VOL | Output low voltage | At specified IOL | | | 0.4 | V | |
| IOH | Output high current | At specified VOH = VDDQ~400MV AT 1.7V VDDQ | -7 | | x | mA | |
| IOL | Output low current | At specified VOL~400MV at 1.7V VDDQ | 7 | | x | mA | |
| IOZ | Tri-state output leakage current | Vin =VDDQ or GND | -10 | +/-0.5 | 10 | µA | |
| IDD1 | Digital operating current | Context B, 1600x1200, JPEG on, MCLK=Max, PIXCLK=Max | | 92 | 110 | mA | |
| IDDQ1 | I/O digital operating current | Context B, 1600x1200, JPEG on, MCLK=Max, PIXCLK=Max | | 1.5 | | mA | 1 |
| IAA1 | Analog operating current | Context B, 1600x1200, JPEG on, MCLK=Max, PIXCLK=Max | | 43 | 55 | mA | |
| IAAPIX1 | Pixel supply current | Context B, 1600x1200, JPEG on, MCLK=Max, PIXCLK=Max | | 2.1 | 3.5 | mA | |
| IDDPLL1 | PLL supply current | Context B, 1600x1200, JPEG on, MCLK=Max, PIXCLK=Max | | 2.3 | 3 | mA | |
| IDD2 | Digital operating current | Context A, 800x600, No JPEG, MCLK=Max, PIXCLK=Max | | 48 | 60 | mA | |
| IDDQ2 | I/O digital operating current | Context A, 800x600, No JPEG, MCLK=Max, PIXCLK=Max | | 15 | | mA | 1 |
| IAA2 | Analog operating current | Context A, 800x600, No JPEG, MCLK=Max, PIXCLK=Max | | 27 | 3.5 | mA | |
| IAAPIX2 | Pixel supply current | Context A, 800x600, No JPEG, MCLK=Max, PIXCLK=Max | | 4 | 5.5 | mA | |
| IDDPLL2 | PLL supply current | Context A, 800x600, No JPEG, MCLK=Max, PIXCLK=Max | | 2.3 | 3 | mA | |
| ISTDBY1 | Standby current PLL enabled | PLL enabled (MCLK = 0Hz, held at either VIL or VIH) | | 35 | 100 | µA | |
| ISTDBY2 | Standby current PLL disabled | PLL disabled (MCLK = 0Hz, held at VIL or VIH) | | 35 | 100 | µA | |

Notes: 1. Due to the influence of several variables (scene illumination, output load) max values are not available.

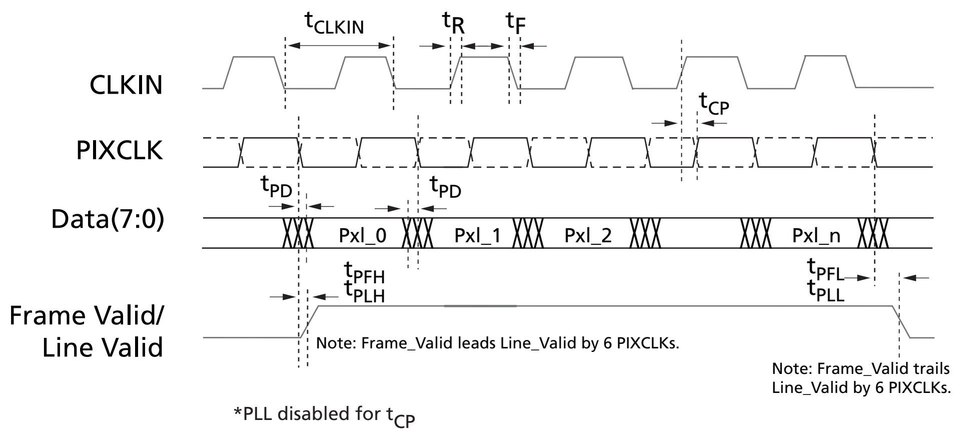
Table 24: Absolute Maximum Ratings

| Symbol | Parameter | Rating | | Unit |
|---|---|---|---|---|
| | | MIN | MAX | |
| VDD | Digital power | -0.3 | 2.4 | V |
| VDDQ | I/O power | -0.3 | 4.0 | V |
| VDDPLL | PLL power | -0.3 | 4.0 | V |
| VAA | Analog power (2.8V) | -0.3 | 4.0 | V |
| VAAPIX | Pixel array power | -0.3 | 4.0 | V |
| VIN | DC input voltage | -0.3 | VDDQ+0.3 | V |
| VOUT | DC output voltage | -0.3 | VDDQ+0.3 | V |
| TOP | Operation temperature | -30 | 70 | °C |
| TSTG[1] | Storage temperature | -40 | 85 | °C |

**Note:** [1]Stresses above those listed may cause permanent damage to the device. This is a stress rating only, and functional operation of the device at these or any other conditions above those indicated in the product specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

## I/O Timing

Figure 48: I/O Timing Diagram



# Appendix A: Two-Wire Serial Register Interface

This section describes the two-wire serial interface bus that can be used in any functional sensor mode.

The two-wire serial interface bus enables R/W access to control and status registers within the sensor core.

The interface protocol uses a master/slave model in which a master controls one or more slave devices. The sensor acts as a slave device. The master generates a clock (SCLK) that is an input to the sensor and used to synchronize transfers. The master is responsible for driving a valid logic level on SCLK at all times. Data is transferred between the master and the slave on a bidirectional signal (SDATA). Both the SDATA AND SCLK signal are pulled up to VDD off-chip by a 1.5KΩ resistor. Either the slave or master device can drive the SDATA line low—the interface protocol determines which device is allowed to drive the SDATA line at any given time.

## Protocol

The two-wire serial interface bus defines the transmission codes as follows:

- a start bit
- the slave device 8-bit address
- a(an) (no) acknowledge bit
- an 8-bit message
- a stop bit

## Sequence

A typical read or write sequence is executed as follows:

1. The master sends a start bit.
2. The master sends the 8-bit slave device address. The last bit of the address determines if the request is a read or a write, where a "0" indicates a write and a "1" indicates a read.
3. The slave device acknowledges receipt of the address by sending an acknowledge bit to the master.
4. If the request is a write, the master then transfers the 8-bit register address, indicating where the write takes place.
5. The slave sends an acknowledge bit, indicating that the register address has been received.
6. The master then transfers the data, 8 bits at a time, with the slave sending an acknowledge bit after each 8 bits.

The sensor core uses 16-bit data for its internal registers, thus requiring two 8-bit transfers to write to one register. After 16 bits are transferred, the register address is automatically incremented so that the next 16 bits are written to the next register address. The master stops writing by sending a start or stop bit.

A typical read sequence is executed as follows.

1. The master sends the write-mode slave address and 8-bit register address, just as in the write request.
2. The master then sends a start bit and the read-mode slave address, and clocks out the register data, 8 bits at a time.
3. The master sends an acknowledge bit after each 8-bit transfer. The register address is auto-incremented after every 16 bits is transferred.

4. The data transfer is stopped when the master sends a no-acknowledge bit.

## Bus Idle State

The bus is idle when both the data and clock lines are high. Control of the bus is initiated with a start bit, and the bus is released with a stop bit. Only the master can generate start and stop bits.

## Start Bit

The start bit is defined as a HIGH-to-LOW data line transition while the clock line is HIGH.

## Stop Bit

The stop bit is defined as a LOW-to-HIGH data line transition while the clock line is HIGH.

## Slave Address

The 8-bit address of a two-wire serial interface device consists of seven bits of address and one bit of direction. A "0" in the LSB (least significant bit) of the address indicates write mode, and a "1" indicates read mode. The default slave addresses used by the sensor core are 0xBA (write address) and 0xBB (read address). R0x0D:0[10] or the SADDR pin can be used to select the alternate slave addresses 0x90 (write address) and 0x91 (read address).

Writes to R0x0D:0[10] are inhibited when the standby pin is asserted (all other writes proceed normally). This allows two sensors to co-exist as slaves on this interface, but they must be addressed independently. Enable this capability as follows:

After RESET, both sensors use the default slave address. Reads or writes on the serial register interface to the default slave address are decoded by both sensors simultaneously.

1. After RESET, assert the STANDBY signal to one sensor and negate the STANDBY signal to the other sensor.
2. Perform a write to R0x0D:0 with bit 10 set. The sensor with STANDBY asserted ignores the write to bit 10 and continues to decode at the default slave address.

The sensor with STANDBY negated has its R0x0D:0[10] set and responds to the alternate slave address for all subsequent READ and WRITE operations, as shown in Table 25.

**Table 25: Slave Address Options**

| SADDR | R0xD:0[10] | Slave Address | |
|---|---|---|---|
| | | WRITE | Read |
| 0 | 0 | 0x090 | 0x091 |
| 0 | 1 | 0x0BA | 0x0BB |
| 1 | 0 | 0x0BA | 0x0BB |
| 1 | 1 | 0x090 | 0x091 |

## Data Bit Transfer

One data bit is transferred during each clock pulse. The serial interface clock pulse is provided by the master. The data must be stable during the high period of the two-wire serial interface clock—it can only change when the serial clock is low. Data is transferred eight bits at a time, followed by an acknowledge bit.

## Acknowledge Bit

The master generates the acknowledge clock pulse. The transmitter (which is the master when writing, or the slave when reading) releases the data line, and the receiver indicates an acknowledge bit by pulling the data line low during the acknowledge clock pulse.

## No-Acknowledge Bit

The no-acknowledge bit is generated when the data line is not pulled down by the receiver during the acknowledge clock pulse. A no-acknowledge bit is used to terminate a read sequence.

## Page Register

The MT9D111 two-wire serial interface and its associated protocols support an address space of 256 16-bit locations. This address space is extended by a 3-bit page prefix, and controlled through accesses to R0xF0:0.

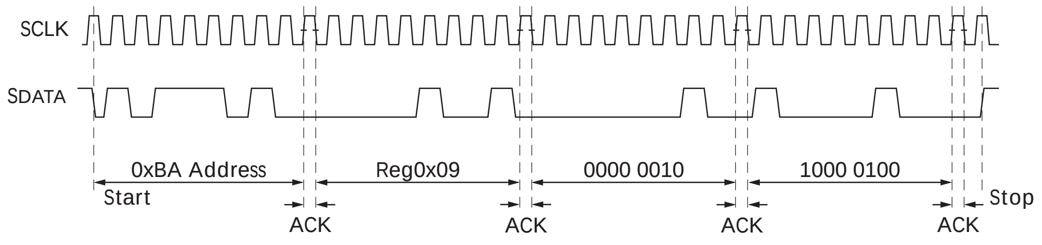
The paging mechanism is intended to allow access to other sets of registers when the sensor is embedded as part of a more complex integrated subsystem, for example, in an SOC. All registers within the sensor core are accessible on page 0 (the default page).

## Sample Write and Read Sequences

### 16-Bit Write Sequence

A typical write sequence for writing 16 bits to a register is shown in Figure 49. A start bit given by the master starts the sequence, followed by the write address. The image sensor then sends an acknowledge bit and expects the register address to come first, followed by the 16-bit data. After each 8-bit transfer, the image sensor sends an acknowledge bit. All 16 bits must be written before the register is updated. After 16 bits are transferred, the register address is automatically incremented so that the next 16 bits are written to the next register. The master stops writing by sending a start or stop bit.

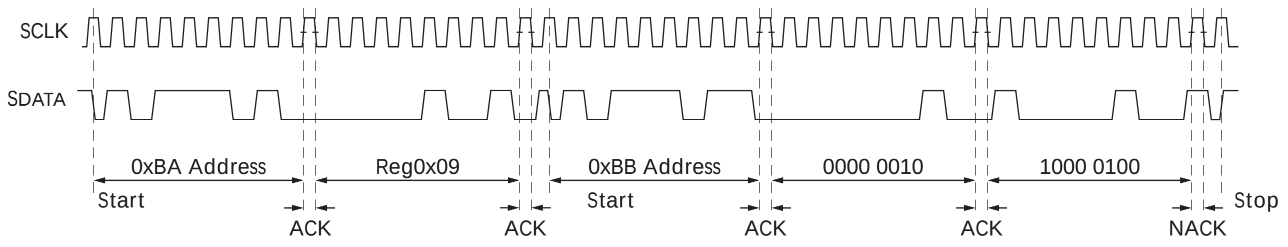
**Figure 49: WRITE Timing to R0x09:0—Value 0x0284**



## 16-Bit Read Sequence

A typical read sequence is shown in Figure 50. First the master writes the register address, as in a write sequence. Then a start bit and the read address specify that a read is about to happen from the register. The master clocks out the register data, eight bits at a time. The master sends an acknowledge bit after each 8-bit transfer. The register address should be incremented after every 16 bits is transferred. The data transfer is stopped when the master sends a no-acknowledge bit.

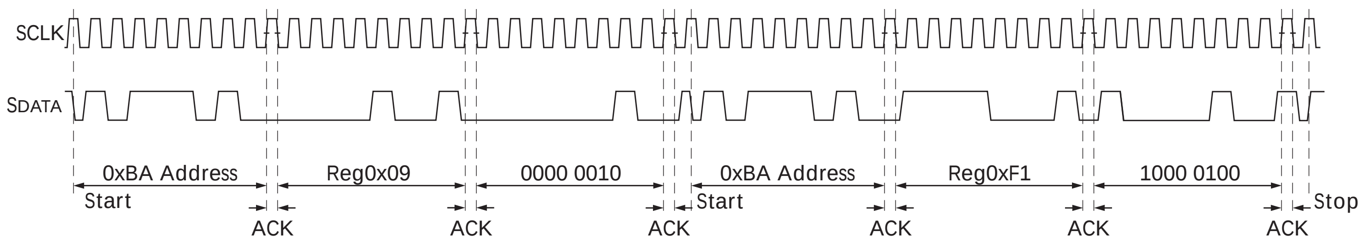
**Figure 50: READ Timing from R0x09:0; Returned Value 0x0284**



## 8-Bit Write Sequence

To be able to write one byte at a time to the register, a special register address is added. The 8-bit write is done by writing the upper 8 bits to the desired register, then writing the lower 8 bits to the special register address (R0xF1:0). The register is not updated until all 16 bits have been written. It is not possible to update just half of a register. In Figure 51, a typical sequence for 8-bit writes is shown. The second byte is written to the special register (R0xF1:0).

**Figure 51: WRITE Timing to R0x09:0—Value 0x0284**



## 8-Bit Read Sequence

To read one byte at a time, the same special register address is used for the lower byte. The upper 8 bits are read from the desired register. By following this with a read from the special register (R0xF1:0), the lower 8 bits are accessed (Figure 52). The master sets the no-acknowledge bits.

**Figure 52: READ Timing from R0x09:0; Returned Value 0x0284**

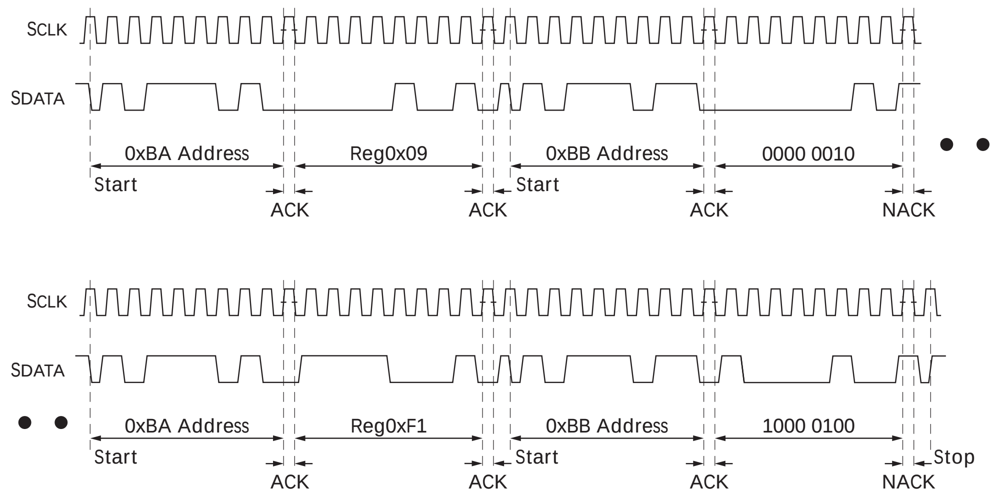
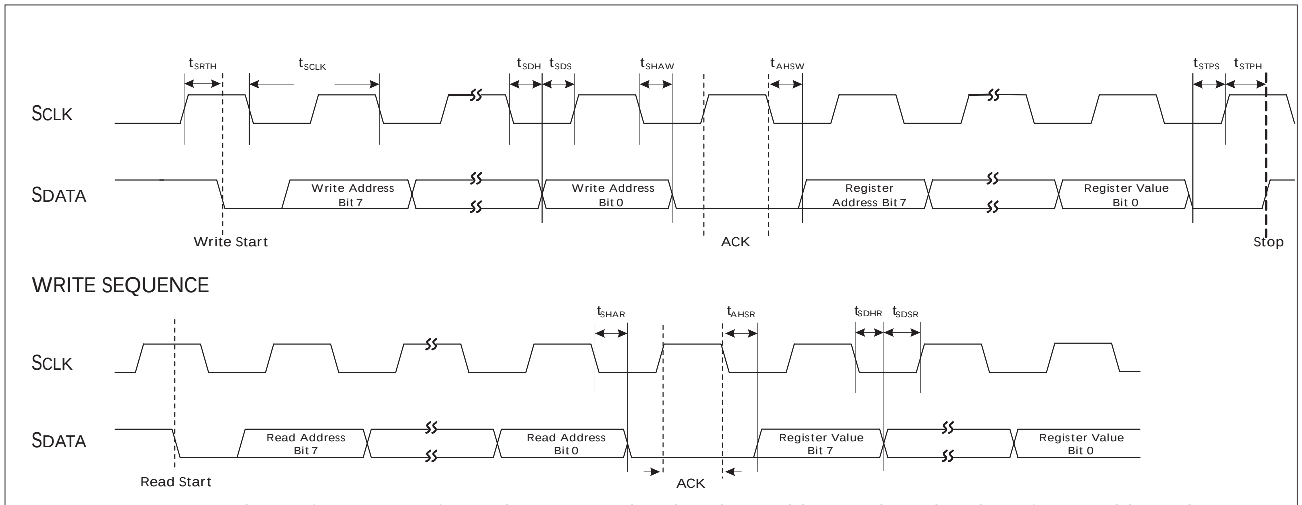


**Figure 53: Two-Wire Serial Bus Timing Parameters**



READ SEQUENCE - read waveforms start after write command and register addres are issued and are for an 8-bit read.

**Table 26: Two-wire Serial Bus Characteristics**

| Symbol | Definition | Conditions | MIN | TYP | MAX | Units |
|---|---|---|---|---|---|---|
| $f_{SCLK}$ | Serial interface input clock frequency | | | | $f_{CLK}/16$ | kHz |
| $t_{SCLK}$ | Serial interface input clock period | | | $1/f_{SCLK}$ | | ns |
| | SCLK duty cycle | | 40 | 50 | 60 | % |
| $t_{SRTH}$ | Start hold time | WRITE/READ | $4^*t_{CLK}$ | | | ns |
| $t_{SDH}$ | SDATA hold | WRITE | $4^*t_{CLK}$ | | | ns |
| $t_{SDS}$ | SDATA setup | WRITE | $4^*t_{CLK}$ | | | ns |
| $t_{SHAW}$ | SDATA hold to ACK | WRITE | $4^*t_{CLK}$ | | | ns |
| $t_{AHSW}$ | ACK hold to SDATA | WRITE | $4^*t_{CLK}$ | | | ns |
| $t_{STPS}$ | Stop setup time | WRITE/READ | $4^*t_{CLK}$ | | | ns |
| $t_{STPH}$ | Stop hold time | WRITE/READ | $4^*t_{CLK}$ | | | ns |
| $t_{SHAR}$ | SDATA hold to ACK | READ | $4^*t_{CLK}$ | | | ns |
| $t_{AHSR}$ | ACK hold to SDATA | READ | $4^*t_{CLK}$ | | | ns |
| $t_{SDHR}$ | SDATA hold | READ | $4^*t_{CLK}$ | | | ns |
| $t_{SDSR}$ | SDATA setup | READ | $4^*t_{CLK}$ | | | ns |
| CIN_SI | Serial interface input pin capacitance | | | 3.5 | | pF |
| CLOAD_SD | SDATA max load capacitance | | | 15 | | pF |
| RSD | SDATA pull-up resistor | | | 1.5 | | KΩ |

Note: Either the slave or master device can drive the SCLK line low—the interface protocol determines which device is allowed to drive the SCLK line at any given time.

## Revision History

**Rev. C** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10/11/12**

- Removed iCSP information from Table 1, "Key Performance Parameters," on page 1
- Updated "Y'U'V' Using sRGB Formulas" on page 33
- Deleted Figure 17, "64-Ball iCSP Package Mechanical Drawing," from page 106

**Rev B, Production** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **9/10**

- Updated to Aptina template

**Rev A, Advance.,** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7/05**

- Initial release