Calibration Process change for ADPD188BI Optical Smoke and Aerosol Detection Module

INTRODUCTION

The ADPD188BI is a complete photometric system for smoke detection using optical, dual-wavelength technology. The module integrates a highly efficient photometric front end, blue and infrared (IR) light emitting diodes (LEDs), and a photodiode, housed in a custom package that prevents light going directly from the LED to the photodiode without first entering the smoke detection chamber. The ADPD188BI is used with an EVAL- CHAMBER smoke chamber to create a complete, optical smoke detection solution for implementation in residential and industrial smoke detectors.

For specific LED drive settings and test/application environments, the ADPD188BI exhibits device to device variability in LED response. The LED response has a slope (gain) and intercept (offset) that varies from device to device, which results in device to device variation in response to a common environment and can be calibrated with gain and offset calibration coefficients. The primary application of ADPD188BI calibration is to allow more effective comparison of multiple device outputs as they are instantiated in the end application. The calibration significantly reduces any device to device optical variation and allows simplified observation of the variations specific to the application environment.

The calibration of the ADPD188BI is done using calibration coefficients that are programmed into the on- chip, nonvolatile memory (NVM). The existing calibration process is being changed to further reduce device to device variation. This document describes the existing and new calibration process information and how users can integrate the new calibration process information into their existing systems.

TEST METHOD

Each LED/driver pair operates into a reflector at multiple LED currents, and the reflector response is measured by the photo- diode inside the ADPD188BI module. The slope of the response is calculated for each LED/driver pair and the intercept is derived from a linear regression. Calibration coefficients are then calculated and stored in the on-chip NVM, otherwise known as the eFuse registers, for later use in the final application. The calibration coefficients are calculated based on a per pulse measurement for a specific device, and are normalized to the mean of a large distribution of collected data from different devices. This normalization ensures that device to device variability is minimized in a population of devices.

READING eFuse REGISTERS

The offset and gain calibration coefficients are stored in on-chip, eFuse registers. The gain calibration coefficients, LED1_GAIN_ COEFF and LED3_GAIN_COEFF, are stored in Register 0x71 and Register 0x72, respectively. The offset calibration coefficients, LED1_INT_COEFF and LED3_INT_COEFF, are stored in Register 0x73 and Register 0x74, respectively.

To access the eFuse registers, take the following steps:

1. Set Register 0x4B, Bit 7 = 1 to enable the 32 kHz oscillator.
2. Write 0x1 to Register 0x10 to force the device into program (idle) mode.
3. Write 0x1 to Register 0x5F to enable the 32 MHz first in, first out (FIFO) clock.
4. Write 0x7 to Register 0x57 to enable access to the eFuse registers.
5. Read Register 0x67. When Register 0x67 = 0x04, the refresh of the eFuse registers is complete and they are ready to be accessed for reading.
6. Apply the error correction code (ECC) function to the eFuse data before applying the calibration coefficients (see the Using ECC to Detect and Correct Errors in eFuse Values section).
7. **Confirm that the contents of Register 0x70 are valid (see Existing Calibration Process section and New Calibration Process section)**
8. Read gain and offset calibration coefficients for desired LED/driver pair(s). The final gain calibration coefficients must be calculated as defined in the Calculating Calibration Coefficients section, using the contents of the eFuse register. When the final gain calibration coefficients are calculated, load them into a user-accessible memory for future use.
9. When reading of the eFuse registers is complete, disable the eFuse registers as follows:
    a. Write 0x0 to Register 0x57 to disable access to the eFuse registers
    b. Write 0x0 to Register 0x5F to disable the 32 MHz FIFO clock.

CALCULATING CALIBRATION COEFFICIENTS

For best operation, users must read eFuse register 0x70 to determine the module type and apply the appropriate equations.

An example case statement is shown here that can be a part of user's software.

```
// check module type
Case (Module type):
          // existing calibration process
          Case 30, 31:
                GAIN_CAL_BLUE = (use equations shown in EXISTING CALIBRATION PROCESS section)
                GAIN_CAL_IR = (use equations shown in EXISTING CALIBRATION PROCESS section)
          // new calibration process
          Case 32:
                GAIN_CAL_BLUE = (use equations shown in NEW CALIBRATION PROCESS section)
                GAIN_CAL_IR = (use equations shown in NEWCALIBRATION PROCESS section)
          Case TBD1: leave for future expansion
          Case TBD2: leave for future expansion
          Default: Raise error
```

EXISTING CALIBRATION PROCESS: Module type = 30 or 31 only

The calibration coefficients must be calculated using the contents of Register 0x71 to Register 0x74, as shown in the following equation:

$$GAIN\_CAL\_X = DEVICE\_SCALAR/NOMINAL\_SCALAR$$

where:

DEVICE_SCALAR = *x_GAIN* × *LEDx* + *x_INTERCEPT*.
x_GAIN is BLUE_GAIN for the blue LED channel and is IR_GAIN for the IR LED channel.
BLUE_GAIN = (17/256)(LED1_GAIN_COEFF -112) + 17.
IR_GAIN = (34/256)(LED3_GAIN_COEFF – 112) + 34.

LEDx is the LED drive current in milliamperes, for example, if the drive current = 200 mA, enter 200. *LEDx* is LED1 for the blue LED channel and is LED3 for the IR LED channel.

x_INTERCEPT is BLUE_INTERCEPT for the blue LED channel and is IR_INTERCEPT for the IR LED channel.
BLUE_INTERCEPT = 8(LED1_INT_COEFF – 128).
IR_INTERCEPT = 5(LED3_INT_COEFF – 128).

NOMINAL_SCALAR = x_MEAN_GAIN × LEDx + x_MEAN_INTERCEPT.
x_MEAN_GAIN is 17 for the blue LED channel and is 34 for the IR LED channel.
x_MEAN_INTERCEPT is 622 for the blue LED channel and is 128 for the IR LED channel.

Table 1. Contents of eFuse Registers for existing calibration process

Address	Name	Bits	Description
0x70	**MODULE_TYPE**	**[7:0]**	**Only modules of Type 30 or 31 can be used**
0x71	LED1_GAIN_COEFF	[7:0]	Blue LED gain coefficient
0x72	LED3_GAIN_COEFF	[7:0]	IR LED gain coefficient
0x73	LED1_INT_COEFF	[7:0]	Blue LED intercept coefficient
0x74	LED3_INT_COEFF	[7:0]	IR LED intercept coefficient
0x7E	ECC	[7:0]	ECC

Users should calibrate the 32kHz and 32MHz on chip clocks (see datasheet sections Calibrating the 32kHz clock and Calibrating the 32MHz clock). 32kHz clock affects the overall sample rate of ADPD188BI. 32MHz clock affects the system gain of ADPD188BI.

NEW CALIBRATION PROCESS: Module type = 33 only

The calibration coefficients must be calculated using the contents of Register 0x71 to Register 0x74, as shown in the following equation:

$$GAIN\_CAL\_X = DEVICE\_SCALAR/NOMINAL\_SCALAR$$

where:
DEVICE_SCALAR = *x_GAIN* × *LEDx* + *x_INTERCEPT*.
x_GAIN is BLUE_GAIN for the blue LED channel and is IR_GAIN for the IR LED channel.
BLUE_GAIN = (21/256)(LED1_GAIN_COEFF -112) + 21.
IR_GAIN = (42/256)(LED3_GAIN_COEFF – 112) + 42.

LEDx is the LED drive current in milliamperes, for example, if the drive current = 200 mA, enter 200. *LEDx* is LED1 for the blue LED channel and is LED3 for the IR LED channel.

x_INTERCEPT is BLUE_INTERCEPT for the blue LED channel and is IR_INTERCEPT for the IR LED channel.
BLUE_INTERCEPT = 8(LED1_INT_COEFF – 80).
IR_INTERCEPT = 5(LED3_INT_COEFF – 80).

NOMINAL_SCALAR = x_MEAN_GAIN × LEDx + x_MEAN_INTERCEPT.
x_MEAN_GAIN is 21 for the blue LED channel and is 42 for the IR LED channel.
x_MEAN_INTERCEPT is 753 for the blue LED channel and is 156 for the IR LED channel.

Table 1. Contents of eFuse Registers for existing calibration process

Address	Name	Bits	Description
0x70	**MODULE_TYPE**	**[7:0]**	**Only modules of Type 33 can be used**
0x71	LED1_GAIN_COEFF	[7:0]	Blue LED gain coefficient
0x72	LED3_GAIN_COEFF	[7:0]	IR LED gain coefficient
0x73	LED1_INT_COEFF	[7:0]	Blue LED intercept coefficient
0x74	LED3_INT_COEFF	[7:0]	IR LED intercept coefficient
0x77	32kHz_TRIM	[7:0]	32kHz clock optimum trim code
0x78	32MHz_TRIM	[7:0]	32MHz clock optimum trim code
0x7E	ECC	[7:0]	ECC

Users should calibrate the 32kHz and 32MHz on chip clocks (see datasheet section Calibrating the 32kHz clock and Calibrating the 32MHz clock). 32kHz clock affects the overall sample rate of ADPD188BI. 32MHz clock affects the system gain of ADPD188BI. Users can read eFuse registers 0x77 and 0x78 and write those values into device registers 0x4B and 0x4D respectively. Alternatively, users can manually calibrate the clocks themselves.

APPLYING CALIBRATION COEFFICIENTS

To apply the calibration coefficients in the final application, take the following steps:

1. Configure the ADPD188BI device as desired.
2. Write 0x2 to Address 0x10 to start normal sampling operation.
3. Take a measurement at the desired LED level and perform the following calculation:

$$Normalized\ Output\ (LSBs) = AFE\_OUT/GAIN\_CAL\_x$$

where:
AFE_OUT = Raw output measurement with LED on. *GAIN_CAL_x* = GAIN_CAL_BLUE for the blue LED channel and is GAIN_CAL_IR for the IR LED channel.

Applying the calibration coefficients results in greatly reduced device to device variation. Figure 1 and Figure 2 show histograms from before and after calibration for the blue LED and IR LED. Figure 1 and Figure 2 illustrate that in both cases, the distribution of device to device variation is narrowed to <±10%.

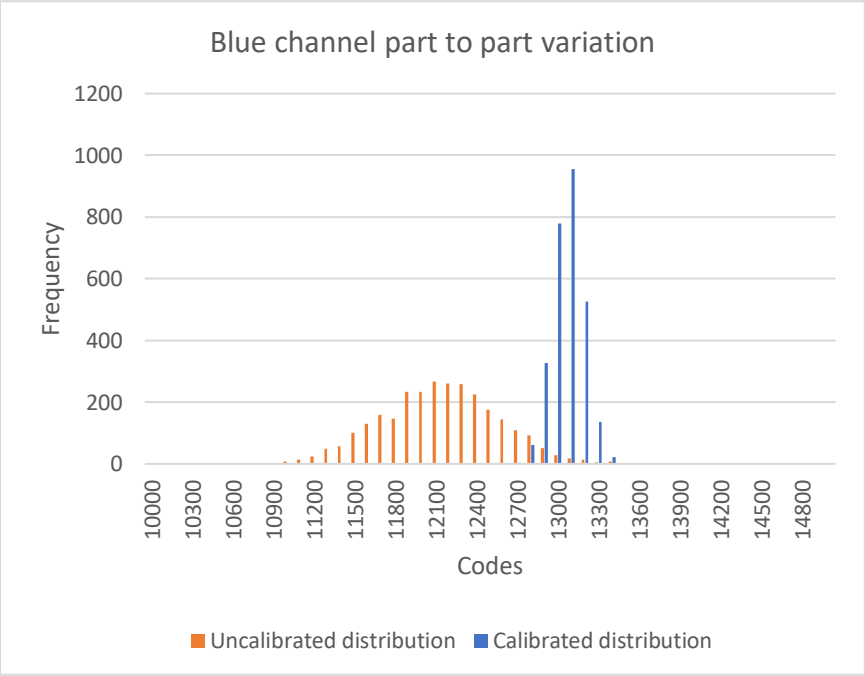

Figure 1. Blue LED Response Before and After Calibration

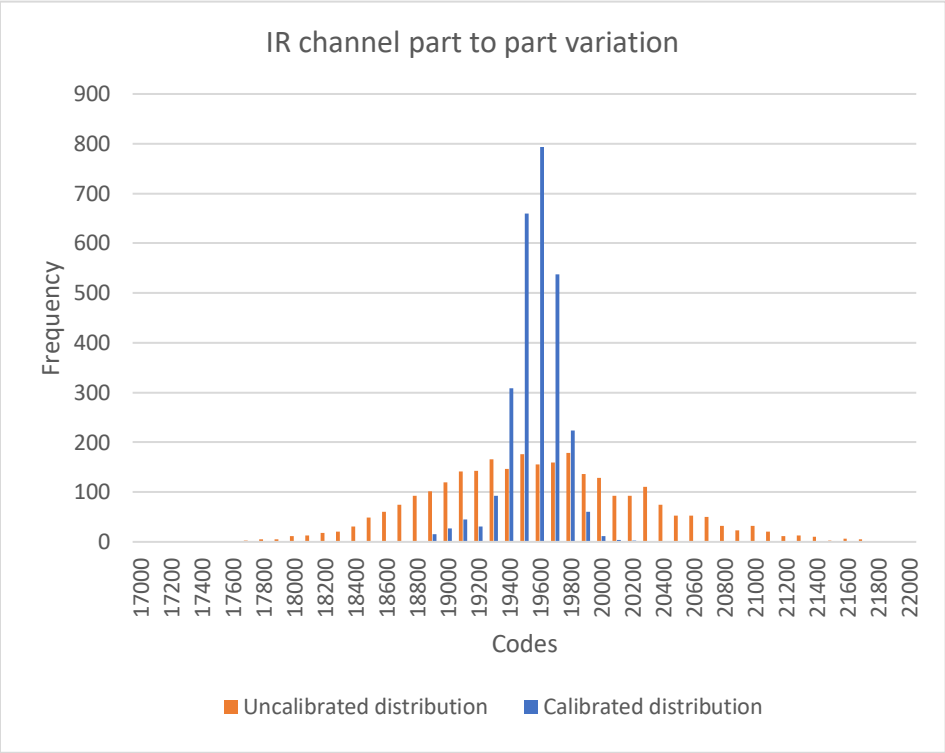

Figure 2. IR LED Response Before and After Calibration

EFFECT OF eFuse CONTENTS ON NORMAL DEVICE OPERATION

The calibration coefficients that are written into the eFuse registers of the ADPD188BI do not modify any device performance or specification. All data sheet specifications and device performance are inherently unaffected by the programming of the eFuse registers.

The calibration coefficients are intended to be used in post- processing of the sampled data in order to calibrate variations in device to device optical characteristics. There is no difference in the performance of the ADPD188BI whether the eFuse registers are programmed or not. In situations where the eFuse registers are programmed with calibration coefficents, the data stored in the eFuse registers can only have an impact if a postprocessing calibration routine is implemented on the sampled data in software.

USING ECC TO DETECT AND CORRECT ERRORS IN eFuse VALUES

The C code shown in the C Code for ECC section contains routines to utilize Hamming codes to detect and correct errors in stored eFuse register values. These functions use a traditional, 127,120 Hamming code, truncated to 119,112. An additional global parity bit is added to provide single-bit correction with 2-bit fail detection. The final form is 120,112 that adds an 8-bit parity code to each 112-bit (14-byte) block.

This code detects and repairs 100% of single-bit errors in each data block and detect 100% of 2-bit fails in each data block.

The methodology is as follows: read the eFuse data and parity bytes into local memory. The user must read Register 0x70 to Register 0x7E. Register 0x70 to Register 0x7D are associated with input pointer, data, and must be read into a data array. Register 0x7E is associated with input pointer, parity, and must be read in as a parity value. Use the fix_hamm_parity command to verify the block. This function repairs single broken bits in place. If the fix_hamm_parity command returns an error, flag the device as bad.

This process fixes all single-bit failures, detects all 2-bit failures and about 6% of 3-bit failures, and detects most even number failures.

C code for ECC

```
int generate_hamm_block_parity( data )
     int data[];
{
// Define parity mapping for parity byte generation/testing
// traditional hamming coding for 127,120 truncated to 120,112
// plus extra parity to make 120,112 code for SECDED.
//
// this table determines which parity bits are involved in each data bit.
// MSB is global "all data parity"
// this function does not include the parity bits in the global bit
// so it can be added differently in the generate_hamm_parity
// and generate_hamm_syndrome functions as needed


  const int paritymap[112]={
    131,  133,  134,  135,  137,  138,  139,  140,  141,  142,
    143,  145,  146,  147,  148,  149,  150,  151,  152,  153,
    154,  155,  156,  157,  158,  159,  161,  162,  163,  164,
    165,  166,  167,  168,  169,  170,  171,  172,  173,  174,
    175,  176,  177,  178,  179,  180,  181,  182,  183,  184,
    185,  186,  187,  188,  189,  190,  191,  193,  194,  195,
    196,  197,  198,  199,  200,  201,  202,  203,  204,  205,
    206,  207,  208,  209,  210,  211,  212,  213,  214,  215,
    216,  217,  218,  219,  220,  221,  222,  223,  224,  225,
    226,  227,  228,  229,  230,  231,  232,  233,  234,  235,
    236,  237,  238,  239,  240,  241,  242,  243,  244,  245,
    246,  247 };
```

```
  //
  int bit,byte; // pointers
  int h;        // parity byte

  h=0; // init parity byte
  // calculate parity for the 112 data bits according to map
  for(byte=0; byte < 14; byte++) {
    for(bit=0x0; bit < 8 ; bit++) {
      if( (data[byte] & (1<<bit) )!=0){ h ^= paritymap[(byte<<3)+bit];
      }
    }
  }
  return(h); // return the parity byte for the 112bit block only
}
//
//
int generate_hamm_syndrome( data, parity_in )
     int data[],*parity_in;
{
  //
      // generate final hamm parity using two steps
      // - generate parity for 112 bit data block
      // - include input parity into global parity bit
      //
      int bit; // pointer
      int h;   // parity byte

      h=generate_hamm_block_parity(data);  // get parity byte for 112 bits

      // add the parity of the 8 input parity bits into the global
      for(bit=0;bit<7;bit++) {
        if ((*parity_in&(1<<bit))==(1<<bit)) h^=0x80;
      }
      return(h); // return the final parity
    }
    //
    // This function checks the data and parity byte
    // for consistency and corrects single bit problems
    // Return Values:
    //   - 0 if the data/parity is correct. (NO REPAIR DONE)
    //   - 1 if there is a single bit error in the data region (REPAIRED)
    //   - 2 if there is a single bit error in the parity byte (REPAIRED)
    //   - 3 if there are multiple errors (NO REPAIR DONE)
    //
    int fix_hamm_parity (data, parity)
         int data[]; int *parity;
```

```
   {
     int calculated_parity;
     int syn, glob;
     int bit, byte;

     calculated_parity=generate_hamm_syndrome(data,parity);
     syn=(*parity^calculated_parity)&0x7f;
     glob=(*parity^calculated_parity)&0x80;
     if(glob==0) {
       if (syn==0) return(0); // no errors (no fix needed)
       else return(3); // double error (can't fix)
     }
     else {
       if (syn>=120) return(3); //also double error
       switch (syn) { // error in lower parity (fix the bit)
       case 0:  *parity=*parity ^ 0x80; return(2);
       case 1: *parity=*parity ^ 0x01; return(2);
       case 2: *parity=*parity ^ 0x02; return(2);
       case 4: *parity=*parity ^ 0x04; return(2);
       case 8: *parity=*parity ^ 0x08; return(2);
       case 16: *parity=*parity ^ 0x10; return(2);
      case 32: *parity=*parity ^ 0x20; return(2);
      case 64: *parity=*parity ^ 0x40; return(2);
      default:     // error in data block (fix it)
// if it gets here there is a single bit data error
// first adjust the address to account for the
// parity bits being outside the data region
      syn =
        (syn>64) ? syn - 8 :
        (syn>32) ? syn - 7 :
        (syn>16) ? syn - 6 :
        (syn>8)  ? syn - 5 :
        (syn>4)  ? syn - 4 : 0;

      byte = syn >> 3;
      bit = syn & 0x7;
      data[byte]=data[byte]^(1<<bit);  // fix the data bit
      return(1);         // single data error (fixed)
    }
  }
}
```