ANALOG DEVICES

AN-2033
アプリケーション・ノート

ADPD188BI 煙およびエアロゾル検出モジュールのキャリブレーション

はじめに

ADPD188BI は、デュアル光学波長技術を使用したフル機能の煙検知用フォトメトリック・システムです。このモジュールには、高効率のフォトメトリック・フロント・エンド、青色と赤外（IR）の発光ダイオード（LED）、フォトダイオードが内蔵されています。これらの部品はカスタム・パッケージに収容され、LED からの光が煙検知チャンバを通ってフォトダイオードに照射されるように考案されています。ADPD188BI を EVAL-CHAMBER スモーク・チャンバと併用することで、住居用や工業用の煙検知器に実装可能なフル機能の光煙検知ソリューションが構成されます。この EVAL-CHAMBER は、EVAL-ADPD188BI の注文時に購入できます。

本アプリケーション・ノートでは、オンチップの不揮発メモリ（NVM）にプログラムされたキャリブレーション係数を使用してデバイスごとの差異を±10%未満に抑える、ADPD188BI のキャリブレーション方法について説明します。

ADPD188BI では、特定の LED 駆動設定とテスト／アプリケーション環境に対する LED の応答が、デバイスごとに異なります。LED の応答の傾き（ゲイン）およびインターセプト（オフセット）はデバイスによって異なるため、同じ環境に対する応答がデバイス間で異なるという結果が生じますが、これはゲインとオフセットのキャリブレーション係数を用いて補正することができます。このキャリブレーションは主に、エンド・アプリケーションで複数のデバイスの出力が生じている場合に、それらをより効率的に比較するために利用されます。このキャリブレーションによって、デバイスごとの光学的な差異が著しく減少し、アプリケーション環境に固有の差異を容易に調べることができるようになります。

Rev. 0

アナログ・デバイセズ株式会社

本　　社／〒105-6891　東京都港区海岸 1-16-1　ニューピア竹芝サウスタワービル 10F
電話 03（5402）8200
大　阪営業所／〒532-0003　大阪府大阪市淀川区宮原 3-5-36　新大阪トラストタワー 10F
電話 06（6350）6868
名古屋営業所／〒451-6038　愛知県名古屋市西区牛島町 6-1　名古屋ルーセントタワー 38F
電話 052（569）6300

目次

改訂履歴

12/2019—Revision 0: Initial Version

テスト方法

各 LED／ドライバ・ペアは、複数の LED 電流値で反射器に向けて動作し、反射器の応答が ADPD188BI モジュール内のフォトダイオードによって測定されます。応答の傾きが LED／ドライバ・ペアごとに計算され、インターセプトが線形回帰から求められます。次にキャリブレーション係数が計算されてオンチップ NVM（別名 eFuse レジスタ）に保存され、後の最終アプリケーションで使用されます。キャリブレーション係数は、特定デバイスのパルスあたりの測定に基づいて計算され、様々なデバイスから収集され幅広く分布したデータの平均値に正規化されます。この正規化によって、デバイスの母集団の中でデバイスごとの差異が最小になるようにできます。

eFuse レジスタからの読出し

オフセットとゲインのキャリブレーション係数は、オンチップの eFuse レジスタに保存されます。ゲインのキャリブレーション係数、LED1_GAIN_COEFF はレジスタ 0x71 に、LED3_GAIN_COEFF はレジスタ 0x72 に保存されます。オフセットのキャリブレーション係数、LED1_INT_COEFF はレジスタ 0x73 に、LED3_INT_COEFF はレジスタ 0x74 に保存されます。

eFuse レジスタにアクセスするには、以下の手順を実行します。

1. レジスタ 0x4B のビット 7 を 1 に設定して 32kHz 発振器をイネーブルします。
2. レジスタ 0x10 に 0x1 を書き込み、デバイスを強制的にプログラム（アイドル）モードに移行させます。
3. レジスタ 0x5F に 0x1 を書き込み、32MHz の先入れ先出し（FIFO）クロックをイネーブルします。
4. レジスタ 0x57 に 0x7 を書き込み、eFuse レジスタにアクセスできるようにします。
5. レジスタ 0x67 を読み出します。レジスタ 0x67 が 0x04 の場合、eFuse レジスタの更新が完了し、読出しアクセスが可能になっています。
6. 誤り訂正符号（ECC）機能を eFuse データに適用してからキャリブレーション係数を適用します（ECC を使用した eFuse 値の誤りの検出・訂正のセクションを参照してください）。
7. レジスタ 0x70 のコンテンツが、Module Type 30 用の 0x1E となっていることを確認します。
8. 目的の LED／ドライバ・ペアについて、ゲインとオフセットのキャリブレーション係数を読み出します。最終的なゲイン・キャリブレーション係数は、eFuse レジスタのデータを使用して、キャリブレーション係数の計算のセクションの定義に従って計算する必要があります。最終的なゲイン・キャリブレーション係数を計算した後、それらをユーザ・アクセス可能なメモリに保存し、今後の使用に備えます。
9. eFuse レジスタの読出しが完了したら、以下の手順で eFuse レジスタをディスエーブルします。
   a. レジスタ 0x57 に 0x0 を書き込み、eFuse レジスタにアクセスできないようにします。
   b. レジスタ 0x5F に 0x0 を書き込み、32MHz の FIFO クロックをディスエーブルします。

キャリブレーション係数の計算

最終的なキャリブレーション係数は、次式のとおり、レジスタ 0x71～0x74 のデータを使用して算出します。

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

ここで、
DEVICE_SCALAR = *x_GAIN* × *LEDx* + *x_INTERCEPT*、
x_GAIN は、青色 LED チャンネルの場合は BLUE_GAIN、IR LED チャンネルの場合は IR_GAIN、
BLUE_GAIN = (17/256)(LED1_GAIN_COEFF − 112) + 17、
IR_GAIN = (34/256)(LED3_GAIN_COEFF − 112) + 34、
LEDx は、ミリアンペアを単位とする LED 駆動電流で、例えば駆動電流が 200mA の場合は 200 と入力、*LEDx* は、青色 LED の場合は LED1、IR・LED の場合は LED3、
x_INTERCEPT は、青色 LED チャンネルの場合は BLUE_INTERCEPT、IR LED チャンネルの場合は IR_INTERCEPT、
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 は、青色 LED チャンネルの場合は 17、IR LED チャンネルの場合は 34、
x_MEAN_INTERCEPT は、青色 LED チャンネルの場合は 622 、IR LED チャンネルの場合は 128。

表 1. eFuse レジスタの内容

アドレス	名前	ビット	説明
0x70	MODULE_TYPE	[7:0]	Module Type 30 のみがキャリブレーションに使用できます
0x71	LED1_GAIN_COEFF	[7:0]	青色 LED のゲイン係数
0x72	LED3_GAIN_COEFF	[7:0]	IR LED のゲイン係数
0x73	LED1_INT_COEFF	[7:0]	青色 LED のインターセプト係数
0x74	LED3_INT_COEFF	[7:0]	IR LED のインターセプト係数
0x7E	ECC	[7:0]	ECC

キャリブレーション係数の適用

キャリブレーション係数を最終的なアプリケーションに適用するには、以下の手順を実行します。

1. 目的に応じて ADPD188BI デバイスを設定します。
2. アドレス 0x10 に 0x2 を書き込み、通常のサンプリング動作を開始します。
3. 目的の LED レベルで測定を行い、以下の計算を実行します。

正規化出力（LSB）= AFE_OUT/GAIN_CAL_x

ここで、
AFE_OUT は、LED がオン状態での未加工の出力測定値、
GAIN_CAL_x は、青色 LED チャンネルの場合は GAIN_CAL_BLUE、IR LED チャンネルの場合は GAIN_CAL_IR。

キャリブレーション係数を適用すると、デバイスごとの差異が大幅に減少します。青色 LED と IR LED のキャリブレーション前後のヒストグラムを図 1 および図 2 に示します。図 1 および図 2 より、どちらの場合もデバイスごとの差異の分布が±10%未満に縮小していることがわかります。

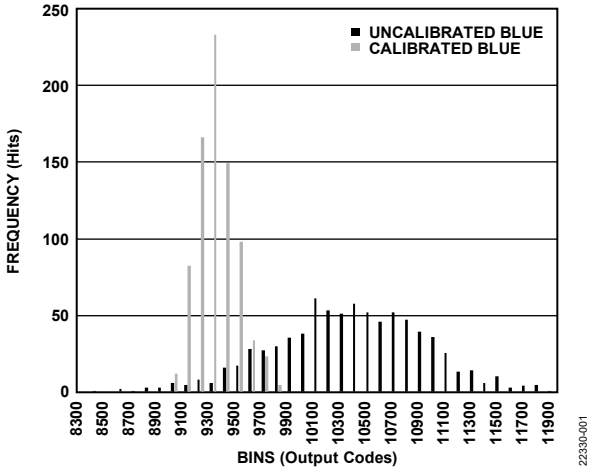

図 1. キャリブレーション前後での青色 LED の応答

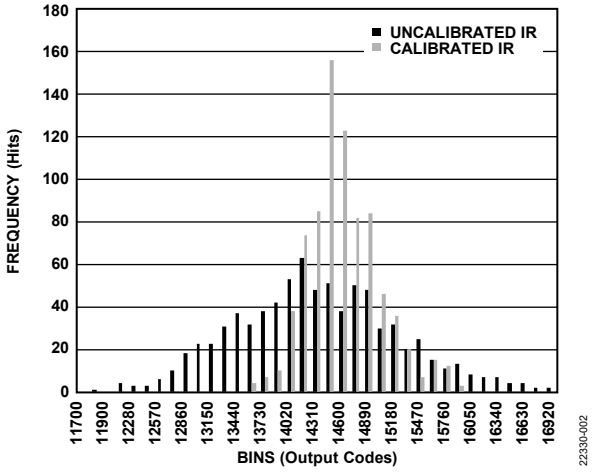

図 2. キャリブレーション前後での IR LED の応答

eFuse のデータが通常のデバイス動作に及ぼす影響

ADPD188BI の eFuse レジスタにキャリブレーション係数が書き込まれても、デバイスの性能や仕様に変更は全く生じません。本質的にデータシートの仕様やデバイスの性能はすべて、eFuse レジスタのプログラミングには影響されません。

キャリブレーション係数は、サンプリングしたデータに後処理を行いデバイスごとの光学特性の差異を補正するために使用することを目的としています。eFuse レジスタがプログラムされているか否かによって、ADPD188BI の性能に違いが生じることはありません。eFuse レジスタにキャリブレーション係数がプログラムされている状況で、eFuse レジスタに保存されているデータが影響する可能性があるのは、ソフトウェアがサンプリング・データに後処理のキャリブレーション・ルーチンを実施する場合のみです。

ECC を使用した eFuse 値の誤りの検出・訂正

ECC用C言語プログラムのセクションに示したCプログラムには、保存されたeFuseレジスタ値の誤りをハミング符号を使用して検出し補正するルーチンが含まれています。この機能では、伝統的な127,120ハミング符号を119,112に切り詰めて使用します。グローバル・パリティ・ビットを追加することで、1ビットの訂正と2ビットの誤り検出ができます。最終的に120,112の形となり、これは、8ビットのパリティ・コードが各112ビット（14バイト）ブロックに追加されたものです。

この符号により、各データ・ブロックの1ビットの誤りはすべて検出／訂正され、2ビットの誤りはすべて検出されます。

手順の最初は、eFuseのデータとパリティのバイトをローカル・メモリに読み出すことです。レジスタ0x70～0x7Eを読み出す必要があります。レジスタ0x70～0x7Dは入力ポインタであるデータに関連するもので、データ配列に読み出します。レジスタ0x7Eは入力ポインタであるパリティに関連するもので、パリティ値として読み出します。fix_hamm_parityコマンドを使用してブロックを検証します。この機能により、1つの破損ビットが適切に訂正されますfix_hamm_parityコマンドでエラーが返された場合、デバイスに問題があることを示すフラグを立てます。

この手順によって、すべての 1 ビット誤りを訂正でき、すべての 2 ビット誤りと 6%の 3 ビット誤りを検出できます。また、ほとんどの偶数誤りが検出できます。

キャリブレーション係数へのハンダ・リフローの影響

リフロー・オーブンでハンダ・リフローを行うと、残存酸素濃度が制御されていない場合、青色 LED に対するフォトダイオードの応答性を低下させる可能性があります。青色 LED に対するフォトダイオード応答のリフローごとの変化は、平均約 7%です。キャリブレーション係数は、ADPD188BI のリフローを行う前の最終テストでプログラムされているため、ハンダ・リフローを酸素濃度が制御されていないオーブンで行った場合、青色 LED の係数は正確ではなくなります。

図 3 に、酸素濃度不制御のオーブンでリフロー処理した後の青色 LED 応答の未加工データとキャリブレーション済みデータを示します。このグループのデバイスには 3 回のリフローを行いました。データには、チェックポイントでの各リフロー後の値も含まれます。データが示すように、各リフロー後には青色 LED 応答が約 7%変化しています。

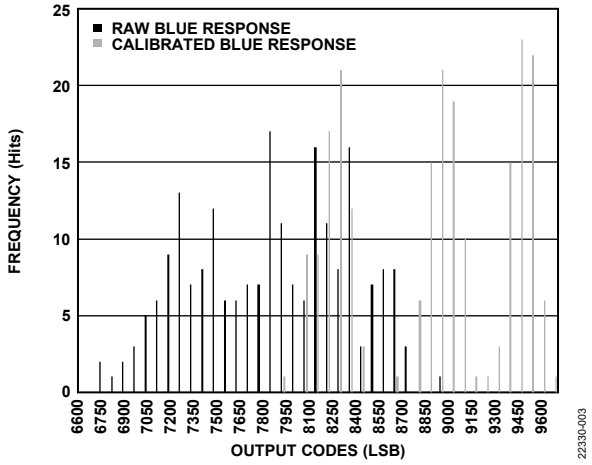

図 3. ハンダ・リフロー中の酸素濃度制御がない場合の青色 LED 応答の変化

応答の変化を防止するためには、窒素によってオーブン中の酸素濃度を減少させるリフロー・オーブンを使用します。窒素制御のリフロー・オーブンを使用して酸素濃度を 1000ppm 未満にした場合、青色 LED 応答の変化は生じません。

図 4 は、窒素パージによって酸素濃度を 1000ppm 未満に減少させたオーブンで 3 回のリフローを行ったデバイスの青色 LED 応答について、未加工データとキャリブレーション済みデータを示したものです。データには、チェックポイントでの各リフロー後の値も含まれます。図 4 に示すように、このような条件下ではリフローによる変化がありません。

IR の応答については、オーブンの酸素濃度制御の有無にかかわらず、リフローによる影響はありません。

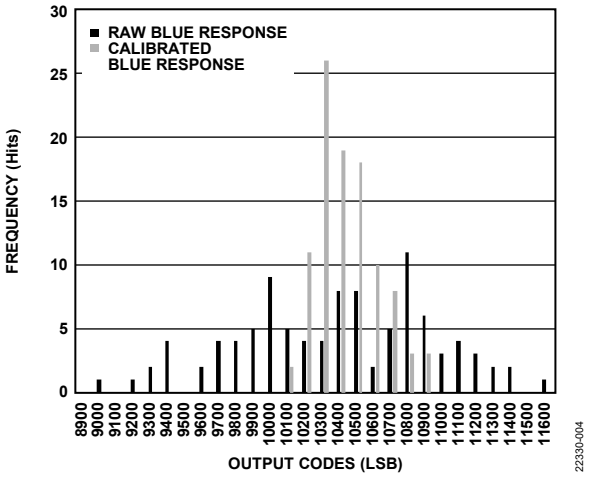

図 4. ハンダ・リフロー中に窒素パージによって酸素濃度を制御した場合の青色 LED 応答の変化

ECC 用 C 言語プログラム

```
int generate_hamm_block_parity( data )
     int data[];
{
// 従来の 127,120 を 120,112 に切り詰めたハミング符号を生成／検査するパリティ・バイトと
// SECDED 用に 120,112 符号を作成する追加パリティ・ビット
// のためのパリティ・マッピングを定義。
//
// この表は各データ・ビットに含まれるパリティ・ビットを定めるものです。
// MSB はグローバルな「全データのパリティ」です。
// この関数にはグローバル・ビットのパリティ・ビットは含まれません。
// そのため、必要に応じて、generate_hamm_parity 関数と
// generate_hamm_syndrome 関数に別個に追加することができます。

  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; // ポインタ
  int h;        // パリティ・バイト

  h=0; // 初期パリティ・バイト
  // マップに従って 112 個のデータ・ビットのパリティを計算
  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); // 112 ビットのブロックのパリティ・バイトのみを返します
}
//
//
int generate_hamm_syndrome( data, parity_in )
     int data[],*parity_in;
{
  //
```

```
  // 次の 2 ステップを使用して最終のハミング・パリティを生成
  // - 112 ビットのデータ・ブロックのパリティを生成
  // - 入力パリティをグローバル・パリティ・ビットに含める
  //
  int bit; // ポインタ
  int h;   // パリティ・バイト

  h=generate_hamm_block_parity(data);  // 112 ビットのパリティ・バイトを取得

  // 8 個の入力パリティ・ビットのパリティをグローバルに追加
  for(bit=0;bit<7;bit++) {
    if ((*parity_in&(1<<bit))==(1<<bit)) h^=0x80;
  }
  return(h); // 最終パリティを返します
}
//
// この関数はデータとパリティ・バイトの整合性を確認し
// 1 ビット問題を訂正します。
// 戻り値 :
//   - 0 : データ／パリティが正しい場合（訂正なし）
//   - 1 : データ領域に 1 ビット誤りがある場合（訂正）
//   - 2 : パリティ・バイトに 1 ビット誤りがある場合（訂正）
//   - 3 : 複数の誤りがある場合（訂正なし）
//
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); // 誤りなし（訂正不要）
    else return(3); // 2 か所の誤り（訂正不能）
  }
  else {
    if (syn>=120) return(3); // 同じく 2 か所の誤り
    switch (syn) { // 偶数パリティの誤り（ビットを訂正）
    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:        // データ・ブロックの誤り（それを訂正）
// この状態になった場合 1 ビットのデータ誤りがあります。
// まず、アドレスを調整し、パリティ・ビットが範囲外となっている
// 原因となったアドレスを調整します。
      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);  // データ・ビットを修正
      return(1);          // 単一のデータ誤り（訂正済み）
    }
  }
}
```