MLX90640 开发笔记

目录

目录

一、概述及开发资料准备

MLX90640 有两个型号，A 型和 B 型焦距和尺寸略有不同。
首先是上 MLX 的官网下载几个必备文件。

（1）MLX90640 数据手册

下载地址是
https://www.melexis.com/-/media/files/documents/datasheets/mlx90640-datasheet melexis.pdf

（2）MLX90640 驱动库和说明文档

下载地址是
https://github.com/melexis/mlx90640-library/archive/master.zip
中文翻译，MLX90640 中文手册下载地址为：
https://download.csdn.net/download/scurobot/87691585

MLX90640 简要介绍说明

（1）A 型和 B 型的区别

区别主要有以下几点

视场角不同：A 型为 110*75°，B 型为 55*35°，通俗一点讲就是 A 型是广角，所以镜头矮一些，视野更宽，但对远处物体的捕捉能力更低，B 型更适于拍摄稍远的物体。

精度不同：A 型的噪声比 B 型大，所以 B 型的绝对温度和灵敏度都好一些。

（2）供电电压和数字接口

MLX90640 共有 4 个引脚，两个电源 3.3V 供电，两个通讯 I2C 接口，I2C 支持最高 1MHz 的通讯速率（实际测试发现 1.2MHz 也是可以的，只是偶尔会出错，还是老老实实 1MHz 吧）。

I2C 完全是经典的时序，而且通讯速率范围特别宽，从几十赫兹到兆赫兹都通讯正常，所以它的 I2C 接口程序还是很好写的。

功耗大约是 25mA，实测没有问题。

供电必须是 3.3V，但 I2C 的两根引脚可以 2.5~5V 兼容。

（3）灵敏度、测温范围和精度

MLX90640 的测量速率最高可以达到 64Hz，但越快的速率时的噪声会越大，导致灵敏度下降，手册上给出的指标是 1Hz 时可以区分出 0.1℃。

测温范围是-40~300℃。

测温精度和成像的区域有关，靠近中间位置是±0.5~1.0℃，最外侧 4 个角是±2.0℃，其它区域约是±1.0℃。

还有就是传感器上电后有个热平衡时间，大约是 5 分钟，未达到热平衡时精度会差一些。

（4）坏点

手册里特别提到了每个 MLX90640 传感器可能存在最多 4 个不能使用或者精度达不到要求的像素，这也许和传感器的生产工艺有关吧，坏点都会在出厂时记录到传感器的 EEPROM 里，实际使用时记得要读取一下并且在成像时特殊处理这种可能存在的像素点数据。

在后面的实际测试过程中，并没有发现坏点的存在。 MLX90640 传感器坏点故障率约为 5%。

在官方的 API 库里，坏点和未达到精度要求的点是不做区分的，都是同样的处理方法

（用相邻的好的点做平均值做为坏点的值）。

二、API 移植 IIC 和接口函数

API 说明文件里面有官方的移植指导，但我觉得可以把重点放在与 MLX90640 具体操作有关的几个函数上，而与标准 I2C 相关的函数和文件结构还是按照自己习惯的套路实现。这样更符合我们开发人员的可控性的习惯。步骤如下：

（1）建立标准 I2C 文件 IIC.h 和 IIC.c

用自己的方法实现如下几个函数（硬件也好，GPIO 模拟也好），函数名称用下面建议的。

```
void IIC_Init(void);
//I2C 接口初始化
void IIC_Start(void);
//发送开始信号
void IIC_Stop(void);
//发送结束信号
void IIC_SendACK(void);
//发送应答信号
void IIC_SendNAK(void);
//发送非应答信号
unsigned char IIC_RecvACK(void);
//读取应答信号
unsigned char IIC_RecvData(void);
//读取 1 个字节
void IIC_SendData(char dat);
//发送 1 个字节
```

（2）在工程中引入 MLX90640_API.c

并做如下几处修改

第一行#include <MLX90640_I2C_Driver.h>改为#include <IIC.h>

（3）添加 3 个函数

```
void MLX90640_I2CInit(void)
unsigned char MLX90640_I2CRead (unsigned short startAddress, unsigned short nWordsRead,
unsigned short *datas)
unsigned char MLX90640_I2CWrite (unsigned short writeAddress, unsigned short word)
void MLX90640_I2CInit(void)
{
  IIC_Stop();
}
//从指定地址读取 n 个字（每个字占用 2 个字节）
unsigned char MLX90640_I2CRead(unsigned short startAddress, unsigned short nWordsRead,
unsigned short *datas)
{
  unsigned char c1,c2;
  unsigned short i;
```

```
    unsigned char Msb,Lsb;
    Msb=(unsigned char)(startAddress>>8);
    Lsb=(unsigned char)(startAddress&0x00FF);
    IIC_Start(); //发送起始命令
    IIC_SendData(0x66); //发送设备地址+写命令
    IIC_RecvACK();
    IIC_SendData(Msb); //发送要操作的地址值 2 字节    IIC_RecvACK();
    IIC_SendData(Lsb);
    IIC_RecvACK();
    IIC_Start(); //发送起始命令
    IIC_SendData(0x67); //发送设备地址+读命令
    IIC_RecvACK();
    for (i=0;i<nWordsRead;i++)
    {
        c1=IIC_RecvData();
        IIC_SendACK();
        c2=IIC_RecvData();
        if (i==(nWordsRead-1))
            IIC_SendNAK();
        else
            IIC_SendACK();
        datas[i]=c1;
        datas[i]<<=8;
        datas[i]|=c2;
}
IIC_Stop(); //发送停止命令
return 0;
}
//向指定地址写入 1 个字（2 字节）
unsigned char MLX90640_I2CWrite(unsigned int writeAddress, unsigned int word)
{
    IIC_Start(); //发送起始命令
    IIC_SendData(0x66); //发送设备地址+写命令
    IIC_RecvACK();
    IIC_SendData(writeAddress>>8); //发送要操作的地址值 2 字节
    IIC_RecvACK();
    IIC_SendData(writeAddress&0x00FF);
    IIC_RecvACK();
    IIC_SendData(word>>8);
    IIC_RecvACK();
    IIC_SendData(word&0x00FF);
    IIC_RecvACK();
    IIC_Stop();
    return 0;    }
```

（4）修改 2 个函数

```
unsigned char MLX90640_DumpEE(unsigned short *eeData)
{
   return MLX90640_I2CRead(0x2400, 832, eeData);
}
unsigned char MLX90640_GetFrameData(unsigned short *frameData)
{
   unsigned short statusRegister,controlRegister1;
   MLX90640_I2CRead(0x8000, 1, &statusRegister);
   if (statusRegister&0x0008)//有测量完成的 Frame
   {
      MLX90640_I2CRead(0x800D, 1, &controlRegister1);
      MLX90640_I2CWrite(0x8000, statusRegister&(~0x0018));
      MLX90640_I2CRead(0x0400, 832, frameData);
      frameData[832] = controlRegister1;
      frameData[833] = statusRegister & 0x0001;
      return 0;
   }
   Return -1;
}
```

至此移植完成 编译工程，若没有错误提示则基本上没有问题了，下一篇开始讲述如何操作 MLX90640。

三、工作流程和操作步骤

默认参数时

MLX90640 的工作流程

（1）上电，内部初始化（约 40ms）

（2）读取工作参数到控制和状态寄存器

（3）开始以 2Hz 的速率测量实时数据并更新到 RAM，自动更新状态寄存器。

测量帧解释

MLX90640 共有 768 个测量像素点，每次测量其中的一半，称为 1 帧，故此完成 768 像素需要测量 2 帧，用帧 0 和帧 1 来表示。即：所谓的 1 帧数据其实是完整像素的一半。

可以修改的参数

可以修改的参数有以下几个方面：

■ **自动测量**：默认为自动测量，即自动循环测量帧 0 和帧 1 更新到 RAM 中。与其对应的是手动测量，即：用指令来控制测量帧 0 还是帧 1。手动测量已经在官方的数据手册中被删除，看来 MLX 也不喜欢别人用，所以我们也就别用了。自动测量保持默认值，不要改就好。

■ **帧分布**：前面已经说了，1 帧实际上是测量完成了一半的像素点，这一半像素有两种分布模式，手册上称为 TV 模式和 Chess 模式，TV 模式以行为单位，是指每帧只测量奇数行或者偶数行，Chess 模式是指以像素为单位，每次交错着像素测 384 个像素点。我们可以称之为“行交错模式”和“像素交错模式”。

在这方面，手册上又说了，出厂时是以 Chess 模式校准的，具有最好的精度（言下之意就是说如果修改为了 TV 模式时会不准），鉴于此，这个参数也不要动。

■ **测量分辨率**：可选的有 16~19 位 AD 转换精度，默认是 18 位，转换位数当然是越高越好了，但 18 和 19 位经过测试也没有发现有什么实际区别，这个参数可改可不改。

■ **测量速率**：每秒测量几帧数据，这个参数很有用处，毕竟我们希望成像后是连续的动画，每秒 2 次一定是不好的，我们可以调用 API 将这个参数修改为 8Hz 或者 16Hz 甚至 32Hz，64Hz 是不建议的，因为测量速率太快时噪声特别大，图像特别乱。普通相机的刷新速率也就 15Hz 左右，所以建议最高设置为 16Hz 吧。

所以，虽然数据手册上写的感觉好像可修改的参数挺多，这么一分析，其实只有 1 个测量速率是有用处的，其它都是浮云（鸡肋）。

参数修改方法讨论：

有两种修改方法，修改寄存器和修改 EEPROM。

（1）修改寄存器（推荐）

传感器上电后会自动从 EEPROM 读取参数到寄存器，寄存器内的参数值是运行时实际执行的参数，直接通过 I2C 修改寄存器值即可，随用随改、立即生效。寄存器的值是掉电遗失的，所以每次上电后都要修改一次。

（2）修改 EEPROM

EEPROM 是掉电不丢失的，所以修改 EEPROM 内的运行参数只需要一次，下次启动生效。但 EEPROM 内存储的不仅只是同步到运行寄存器的几个参数，大部分的是 768 个像素的校准参数，这些参数是出厂时写入的，特别重要，所以我的建议还是不要对 EEPROM 有任何的写操作，以免发生意外，EEPROM 里的像素校正参数一旦被意外修改就再也找不回来了。

建议的操作流程

```
unsigned short EE[832];
unsigned short Frame[834];
paramsMLX90640 MLXPars;
float Vdd,Ta,Tr;
float Temp[768];
IIC_Init();
//I2C 初始化
MLX90640_I2CInit();
//MLX 传感器初始化
Delay_ms(50);
//预留一点时间让 MLX 传感器完成自己的初始化
MLX90640_SetRefreshRate(0);
//测量速率 1Hz(0~7 对应 0.5,1,2,4,8,16,32,64Hz)
MLX90640_I2CRead(0x2400, 832, EE);
//读取像素校正参数
MLX90640_ExtractParameters(EE, &MLXPars);
//解析校正参数（计算温度时需要）
while (1)
{
  Delay_ms(5);
  if (MLX90640_GetFrameData(Frame)==0) //有转换完成的帧
  {
    Vdd=MLX90640_GetVdd(Frame, MLXPars); //计算 Vdd（这句可有可无）
    Ta=MLX90640_GetTa(Frame, MLXPars);
    //计算实时外壳温度
    Tr=Ta-8.0;
    //计算环境温度用于温度补偿
    //手册上说的环境温度可以用外壳温度-8℃
    MLX90640_CalculateTo(Frame, MLXPars, 0.95, Tr, Temp); //计算像素点温度
    /*
    Temp 数组内即是转换完成的实时温度值，单位℃
    可以在这里对得到的 32*24=768 个温度值进行处理、转换为颜色值、显示
    关于温度转颜色方法，在后续的文章中会有专门介绍
    */
  }
}
```

一点疑问

校正参数存储于传感器内部的 EEPROM，实时数据也来自传感器，如何利用实时数据和校正参数计算温度的方法也是事先规定好的，MLX 为什么不直接在内部完成这个温度计算让用户直接读取温度值？为了体现这个传感器的复杂性或者是让用户有成就感吗？本来可以在传感器内部解决的问题被厂家要求在外部完成，对 MCU 的性能要求是特别高的，大量的浮点运算，大量的 RAM 消耗，较低的效率。

四、坏点处理

如前“开发笔记（一）”所说，MLX90640 可能存在不超过 4 个像素的损坏或者不良像素，在温度计算过程完成后，这些不良像素点会得到错误的温度数据，对于处理这些不良数据 MLX 也给出了推荐方法和具体的函数。（其实就是找相邻的正常的温度数据取平均来代替不良数据）

前面开发笔记（一）的内容中所说的 API 库，里面缺少了对不良像素点的处理函数，在这里补上。

```
int CheckAdjacentPixels(uint16_t pix1, uint16_t pix2)
{
int pixPosDif;
pixPosDif = pix1 - pix2;
if(pixPosDif > -34 && pixPosDif < -30)
{
return -6;
}
if(pixPosDif > -2 && pixPosDif < 2)
{
return -6;
}
if(pixPosDif > 30 && pixPosDif < 34)
{
return -6;
}
return 0;
}
float GetMedian(float *values, int n)
{
float temp;
for(int i=0; i<n-1; i++)
{
for(int j=i+1; j<n; j++)
{
if(values[j] < values[i])
{
temp = values[i];
values[i] = values[j];
values[j] = temp;
}

}
}
if(n%2==0)
```

```
{
return ((values[n/2] + values[n/2 - 1]) / 2.0);
}
else
{
return values[n/2];
}
}
int IsPixelBad(uint16_t pixel,paramsMLX90640 *params)
{
for(int i=0; i<5; i++)
{
if(pixel == params->outlierPixels[i] || pixel == params->brokenPixels[i])
{
return 1;
}
}
return 0;
}
void MLX90640_BadPixelsCorrection(uint16_t *pixels, float *to, int mode, paramsMLX90640
*params)
{
float ap[4];
uint8_t pix;
uint8_t line;
uint8_t column;
pix = 0;
while(pixels[pix] != 0xFFFF)
{
line = pixels[pix]>>5;
column = pixels[pix] - (line<<5);
if(mode == 1)
{
if(line == 0)

{
if(column == 0)
{
to[pixels[pix]] = to[33];
}
else if(column == 31)
{
to[pixels[pix]] = to[62];
}
```

```
else
{
to[pixels[pix]] = (to[pixels[pix]+31] + to[pixels[pix]+33])/2.0;
}
}
else if(line == 23)
{
if(column == 0)
{
to[pixels[pix]] = to[705];
}
else if(column == 31)
{
to[pixels[pix]] = to[734];
}
else
{
to[pixels[pix]] = (to[pixels[pix]-33] + to[pixels[pix]-31])/2.0;
}
}
else if(column == 0)
{
to[pixels[pix]] = (to[pixels[pix]-31] + to[pixels[pix]+33])/2.0;
}
else if(column == 31)
{
to[pixels[pix]] = (to[pixels[pix]-33] + to[pixels[pix]+31])/2.0;
}
else
{
ap[0] = to[pixels[pix]-33];
ap[1] = to[pixels[pix]-31];
ap[2] = to[pixels[pix]+31];
ap[3] = to[pixels[pix]+33];
to[pixels[pix]] = GetMedian(ap,4);

}
}
else
{
if(column == 0)
{
to[pixels[pix]] = to[pixels[pix]+1];
}
```

```
else if(column == 1 || column == 30)
{
to[pixels[pix]] = (to[pixels[pix]-1]+to[pixels[pix]+1])/2.0;
}
else if(column == 31)
{
to[pixels[pix]] = to[pixels[pix]-1];
}
else
{
if(IsPixelBad(pixels[pix]-2,params) == 0 && IsPixelBad(pixels[pix]+2,params) ==
0)
{
ap[0] = to[pixels[pix]+1] - to[pixels[pix]+2];
ap[1] = to[pixels[pix]-1] - to[pixels[pix]-2];
if(fabs(ap[0]) > fabs(ap[1]))
{
to[pixels[pix]] = to[pixels[pix]-1] + ap[1];
}
else
{
to[pixels[pix]] = to[pixels[pix]+1] + ap[0];
}
}
else
{
to[pixels[pix]] = (to[pixels[pix]-1]+to[pixels[pix]+1])/2.0;
}
}
}
pix = pix + 1;
}
}
```

用法很简单，在开发笔记（三）MLX90640_CalculateTo(Frame, MLXPars, 0.95, Tr, Temp);之后添加两行即可。如下（斜体是添加的内容）：

```
    ……
MLX90640_CalculateTo(Frame, MLXPars, 0.95, Tr, Temp);
MLX90640_BadPixelsCorrection(MLXPars.brokenPixels, Temp, 1, MLXPars);
MLX90640_BadPixelsCorrection(MLXPars.outlierPixels, Temp, 1, MLXPars);
……
/*
经过上面的处理后，Temp 中的损坏和不良像素点已经处理，Temp 数组中是处理完成后的
768 个温度值。
*/
```

五、阵列插值

MLX90640 的 32*24=768 像素虽然比以往的 8*8 或者 16*8 像素提高了很多，但若直接用这些像素还是不能很好的形成热像图，为了使用这些像素点平滑成像就需要对其进行插值，使用更多的像素来绘制图像。

看了一些别人的算法，感觉主要就是多项式插值，仅是插值方法的组合方式不同。

算法依据

比较有代表性的是杭州电子科技大学杨风健等《基于 MLX90620 的低成本红外热成像系统设计》，使用三次多项式+双线性插值，将原 16*4 像素扩展为 256*64 像素。双线性插值的

本质就是一次函数（一次多项式）。该文章得到的结论是：

（1）双线性插值法计算量小、速度快，但对比度低、细节模糊。

（2）三次多项式插值，图像效果较清晰，对比度较高，但计算量较大。

（3）先双线性插值再三次多项式插值，效果优于上两种单一插值方法。

（4）先三次多项式插值再双线性插值，高低温分布更加明显，图像效果更接趋于真实。

同时，该文章还使用了一种对图像质量的评估方法---熵&平均梯度。

熵，热力学中表征物质状态的参量之一，用符号 S 表示，其物理意义是体系混乱程度的度量。用于图像评价表示图像表达信息量的多少。图像熵越高信息量越大。

平均梯度，指图像的边界或影线两侧附近灰度有明显差异，即灰度变化率大，这种变化率的大小可用来表示图像清晰度。它反映了图像微小细节反差变化的速率，即图像多维方向上密度变化的速率，表征图像的相对清晰程度。值越大表示图像越清晰。

插值实现

每行或者列的首个像素在前面插值 2 个点

1~n-1 像素，每个像素后面插值 3 个点

最后一个像素，在后面插值 1 个点

n+2+(n-1)*3+1=n+2+n*3-1*3+1=4n+2-3+1=4n，即：像素变为原来的 4 倍

上面的处理方法，首个像素之前插入 2 个点，最后一个像素之后插入 1 个点，下次插值时，应首个之前插值 1 个点，末个像素之后插值 2 个点，以达到图像平衡。

每次插值后像素为插值前的 4 倍，经过两次插值，即可将 32*24 改变为 512*384 像素。下面是已经实际使用的插值算法，不过是用 Pascal（Delphi）写的，有兴趣的可以改为 C 语言的，语句对应直接改就行，语言本来就是相通的嘛。

```
//这是一维数组插值算法
//SourceDatas:TDoubles;插值前的一维数组
//Dir:Integer;在哪个方向和末尾插入 2 个值（
0：前面；1：末尾）
//times:Integer 多项式的项数，一次多项式是 2 项，二次多项式是 3 项
//返回值：插值后的一维数组（数量是插值前*4）
function PolynomialInterpolationArr(
SourceDatas:TDoubles;
Dir:Integer;
times:Integer):TDoubles;//一维数组插值
var
i,j,k:Integer;
```

```
arrCount:Integer;
startIndex:Integer;
OriginDatas,TargetDatas:ArrayOf2D;
tempStr:string;
tempDou:Double;
coes:array[0..5] of Double;
begin
arrCount:=Length(SourceDatas);
SetLength(Result,arrCount*4);
if Dir=0 then startIndex:=2
else startIndex:=1;
//源数据复制到目标数组 Result
for i := 0 to arrCount-1 do
begin
Result[startIndex+i*4]:=SourceDatas[i];
end;
SetLength(OriginDatas,2,times);
//插值，插值完成后是*4 像素
for i := 0 to arrCount-times do
begin
for j := 0 to times-1 do//初始化拟合原始数据
begin
OriginDatas[0][j]:=j*4;
OriginDatas[1][j]:=SourceDatas[i+j];
end;
GetPolyData_U(OriginDatas,times,coes);
//插值
for j := 1 to 4-1 do
begin
if times>=2 then tempDou:=coes[0]+j*coes[1];
if times>=3 then tempDou:=tempDou+j*j*coes[2];
if times>=4 then tempDou:=tempDou+j*j*j*coes[3];
Result[startIndex+i*4+j]:=tempDou;
end;
end;
//两端插值，两端插值直接使用线性插值（一次多项式）
SetLength(OriginDatas,2,2);
//前端插值
OriginDatas[0][0]:=0;
OriginDatas[1][0]:=SourceDatas[0];
OriginDatas[0][1]:=4;
OriginDatas[1][1]:=SourceDatas[1];

GetPolyData_U(OriginDatas,2,coes);
```

```
if Dir=0 then
begin
tempDou:=coes[0]+(-1)*coes[1];
Result[1]:=tempDou;
tempDou:=coes[0]+(-2)*coes[1];
Result[0]:=tempDou;
end
else
begin
tempDou:=coes[0]+(-1)*coes[1];
Result[0]:=tempDou;
end;
//末端插值
for i := (arrCount-times) to (arrCount-2) do
begin
for j := 0 to 2-1 do//初始化拟合原始数据
begin
OriginDatas[0][j]:=j*4;
OriginDatas[1][j]:=SourceDatas[i+j];
end;
GetPolyData_U(OriginDatas,2,coes);
//插值
for j := 1 to 4-1 do
begin
tempDou:=coes[0]+j*coes[1];
Result[startIndex+i*4+j]:=tempDou;
end;
end;
if Dir=0 then
begin
tempDou:=coes[0]+(5)*coes[1];
Result[arrCount*4-1]:=tempDou;
end
else
begin
tempDou:=coes[0]+(5)*coes[1];
Result[arrCount*4-2]:=tempDou;
tempDou:=coes[0]+(6)*coes[1];
Result[arrCount*4-1]:=tempDou;
end;
end;
```

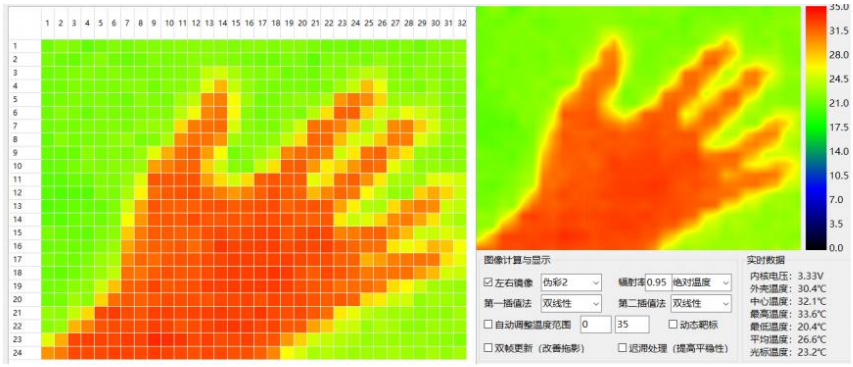
上面函数里用到的一个系数求解函数如下

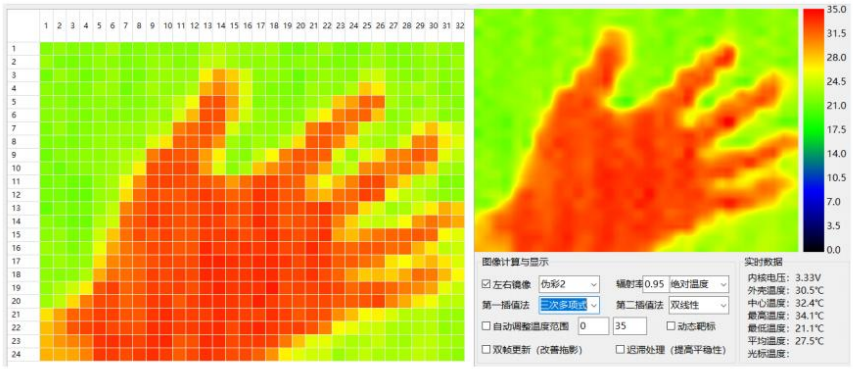
```
function GetPolyData_U(OriginData: ArrayOf2D;times:Integer;var coes:array of Double):
```

```
ArrayOf2D;//times 为项数，1 次多项式有 ab 两项，以些类推
var
x1,x2,x3,x4:Double;
y1,y2,y3,y4:Double;
begin
//1 次多项式：a+bx=y
//2 次多项式：a+bx+cx^2=y
//3 次多项式：a+bx+cx^2+dx^3=y
if ((times<2) or (times>4)) then
times:=2;
if times=2 then
begin
x1:=OriginData[0][0];
x2:=OriginData[0][1];
y1:=OriginData[1][0];
y2:=OriginData[1][1];
coes[1]:=(y2-y1)/x2;
coes[0]:=y1;
end
else if times=3 then
begin
x1:=OriginData[0][0];
x2:=OriginData[0][1];
x3:=OriginData[0][2];
y1:=OriginData[1][0];
y2:=OriginData[1][1];
y3:=OriginData[1][2];
coes[2]:=((y3-y1)*x2-(y2-y1)*x3)/(x2*x3*x3-x2*x2*x3);
coes[1]:=(y2-y1)/x2-coes[2]*x2;
coes[0]:=y1;
end
else if times=4 then
begin
x1:=OriginData[0][0];
x2:=OriginData[0][1];
x3:=OriginData[0][2];
x4:=OriginData[0][3];
y1:=OriginData[1][0];
y2:=OriginData[1][1];
y3:=OriginData[1][2];
y4:=OriginData[1][3];
coes[3]:=((y4-y1)*x2-(y2-y1)*x4)/x2-((y3-y1)*x2-(y2-y1)*x3)/(x2*x3*x3-
x2*x2*x3)*(x2*x4*x4-x2*x2*x4)/x2;
coes[3]:=coes[3]/((x2*x4*x4*x4-x2*x2*x2*x4)/x2-(x2*x3*x3*x3-
```

```
    x2*x2*x2*x3)/(x2*x3*x3-x2*x2*x3)*(x2*x4*x4-x2*x2*x4)/x2);
coes[2]:=((y3-y1)*x2-(y2-y1)*x3)/(x2*x3*x3-x2*x2*x3)-coes[3]*((x2*x3*x3*x3-
x2*x2*x2*x3)/(x2*x3*x3-x2*x2*x3));
coes[1]:=((y2-y1)-coes[2]*x2*x2-coes[3]*x2*x2*x2)/x2;
coes[0]:=y1;
end;
end;
```

下面是几个效果图





我并没有体会到文章开始时提到的那篇论文所说的“先三次多项式插值再双线性插值，高低温分布更加明显，图像效果更接趋于真实”

六、红外图像伪彩色编码

什么是红外成像伪彩编码

红外成像的最终目的是用图像来表现温度变化，并且可以通过颜色来区分出不同热量的物体轮廓和形状。那么，到底用什么颜色来表示什么温度呢？是否有什么标准规范呢？

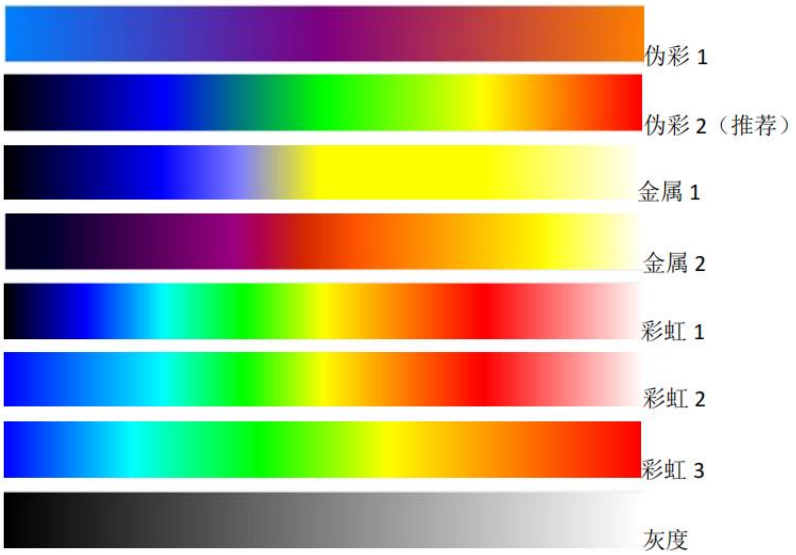
这个问题一开始也是心里没底的，因为我不是专业做红外成像的，只能到处查资料了解温度和颜色之间的关系，基本得到以下几点结论：

（1）温度和颜色之间没有绝对的对应关系，没有人要求红外成像必须要用什么颜色来表示某个温度，这种对应关系完全是由设计人员自己决定的。

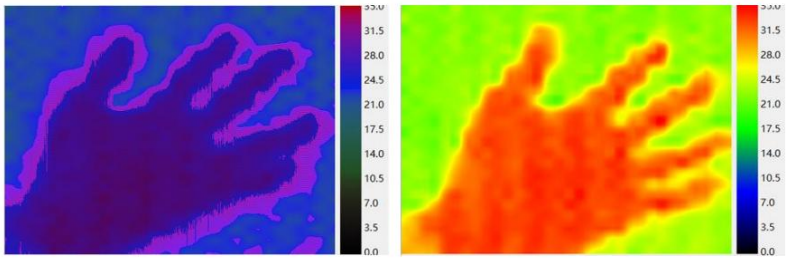
（2）不同的应用领域和行业出于不同的目的，会进行一些温度和颜色的研究，进而用一种适用的渐变色来突出显示某些特别关心的热元素。

（3）颜色编码绝大多数是渐变色。

以下是几种不同的颜色编码



另外，还有人提出了“符合人的生理”让人看着更加“舒服”的 HIS 彩色



温度转颜色的方法

（1）首先假设温度范围的上下限并将实际的温度数据转换为 0~255 之间的数值

（2）使用转换后的数值代入下面的伪彩编码计算函数，生成伪彩色

```
//伪彩 1
procedure GrayToPseColor(grayValue:Integer; var colorR,colorG,colorB:Ingeger);
Begin
```

```
colorR:=Abs(0-grayValue);
colorG:=Abs(127-grayValue);
colorB:=Abs(255-grayValue);
End;
```

七、注意事项

（1）硬件设计注意事项

电源：MLX90640 使用 3.3V 供电，并且使用供电电压做为温度测量的参考电压来使用，所以对电源的要求比较高，尽量使用 LDO 稳压元件，并且 10uF 和 0.1uF 的退耦电容不能省，一定要靠近 VDD 管脚放置。电源电流没太多要求，能够平衡的输出 100mA 就足够。

通讯：I2C 的两个管脚到 MCU 的距离不能过长（小于 5CM 吧），由于通讯速率可以 1MHz，所以上拉电阻不能太大，推荐使用 1k~2k 的电阻。

（2）软件设计注意事项

I2C 部分：尽量使用已经经过验证的 I2C 驱动程序，可以从其它项目中复制过来稍加改动，这部分是通讯的基础，一定要可靠。

接口层：主要是指读和写 MLX90640 的两个函数，可以先读写 MLX90640 的寄存器（地址 0x8000~0x8016）对读写函数进行验证。

计算层：这部分就直接用 API 库中的对应函数就可以，基本不会有问题。

（3）数据正确性验证

在操作过程中必须注意解算出的 Vdd 的值，如果这个值与实际值相差超过 0.1V 就应该检查问题。

（4）水平方向问题

如果镜头向前，MLX90640 的像素排列规则是从右向左（和我们的习惯相反），即：有点像手机的自拍摄像头，图像会是左右颠倒的，在成图之前记得把每行的像素前后颠倒一下，再显示就对了。

八、辐射率、灵敏度、精度、探测距离

辐射率

是描述面辐射源特性的物理量。它表示某物体的单位面积辐射的热量和黑体在相同温度、相同条件下的辐射热量之比。

即：辐射率通俗的说就是某物体会将自身温度转换为辐射扩散出去的能力，1 表示可以将自身温度转换为 100%的辐射，0.9 表示可以将自身温度的 90%转换为热辐射扩散出去。实际上辐射率为 1 的物质（黑体）是不存在的，所以任何材料的辐射率均是 0~1 之间的数值。

任何物体在高于绝对零度（-273.15℃）的时候，其物体表面就会有红外能量也就是红外线发射出来，温度越高，发射的红外能量越强。红外线测温仪和红外热像仪就是根据这个特点来测量物体表面的温度的，因为红外线测温仪和红外热像仪是测量物体表面的温度，所以在测量时会被物体表面的光洁度所影响。实验证明：物体表面越接近于镜面（反射越强），其表面所发出的红外能量衰减越厉害，就需要对不同物体的表面对红外能量的衰减情况做出补偿，即设置一个补偿系数，这个补偿系数就是辐射率。

不同材料的辐射率差别还是很大的，比如人体是 0.95~0.98，而光滑的不锈钢是 0.16，生锈的铁是 0.65 左右，当探测过程中想要得到较为准确的温度值时，就需要设置被探测物体的辐射系数。（若仅是为了成像则无需这么严格）

灵敏度与温度探测精度

噪声等效温差（NETD）是指红外探测器能探测到的最小温差，即：当被测物体的温度变化多少时红外探测器可以探测出来。衡量红外探测器性能的主要指标之一。

热探测器的噪声等效温差在 100mK 左右（0.1℃）。

第二代光探测器在 20mK 左右（0.02℃）。

第三代探测器目标在 1mK（0.001℃）。

红外探测计算出的绝对温度值与被测物体的辐射率参数有直接关系，不同材料的辐射率值是不同的，更为严重的是即便是同种材料，表面光洁度、含水率、温度高低等因素的影响也会直接改变辐射率，这就导致了红外探测绝对温度无法绝对准确，问题不在于红外探测器的对辐射量的感知准确度而在于材料的辐射率是随时在小范围变化的。所以，衡量红外探测器的性能指标一般不能用绝对温度，而应该用温度灵敏度，即：噪声等效温差（NETD）。这也可以得出一个结论，红外热成像仪的主要作用是尽量区分出不同区域的温度差异，用数字表现出来，进而展示为带有颜色的图像，只有温度区分开以后，图像才会细致分明。

红外探测距离

红外热成像仪是用光学镜头来收集被测物体的热辐射能量的，故此探测距离会受镜头视场角和热成像像素分辨率有关。

假如某成像仪的成像分辨率为 32*32 像素，视场角为 75 度，则可以理解为从镜头发射出 32*32=1024 条激光来探测 1024 个点的温度（32 行*32 列），每行 32 个点，每列 32 个点。则每相邻两条激光线的夹角为 75/31=2.4193° 发散出去。随着距离的增长，两条激光线的间距会变大，当被测物体足够小时，有可能处于两条激光线之间未被探测到，这就是探测距离的问题。

即：当成像仪的像素数量和视场角一定时，它的有效探测距离就与被测物体的大小有关。当被测物体尺寸已知时，对其进行探测的理论最远距离为：

$$S = \frac{D}{2 \times \tan\alpha}$$

$$D = 2 \times S \times \tan\alpha$$

$$\alpha = \tan^{-1}\left(\frac{D}{2 \times S}\right)$$

上式中：

S：探测距离

D：被测物体尺寸

α：相邻探测线之间的夹角

例 1：被测物体尺寸为 0.5 米，线夹角为 2.4193°，则理论上的最远探测距离为：

$$S = \frac{D}{2 \times \tan\alpha} = \frac{0.5}{2 \times \tan 2.4193} = \frac{0.5}{0.0845} = 5.92\ 米$$

例 2：探测距离为 10 米，线夹角为 2.4193°，则理论上可探测到最小的物体尺寸为：

$$D = 2 \times S \times \tan\alpha = 2 \times 10 \times \tan 2.4193 = 0.845\ 米$$

例 3：希望探测到 100 米外的一辆小汽车（长 2 米），需要的测线夹角为：

$$\alpha = \tan^{-1}\left(\frac{D}{2 \times S}\right) = \tan^{-1}\left(\frac{2}{2 \times 100}\right) = \tan^{-1}(0.01) = 0.573\ 度$$

由于线夹角=视场角/(行或者列像素-1)，为了得到小的线夹角，要么减少视场角度，要么增大像素数，或两者兼有。比如：视场角不变，选择 160*160 点阵的成像仪，则测线夹角为：

$$\alpha = \frac{75}{(160 - 1)} = 0.472\ ^\circ$$

九、EEPROM、RAM、寄存器说明

EEPROM

地址范围为 0x2400~0x273F，共 832 个字（1664 字节），前 16 个字包含了唯一 ID 码、工作参数（上电后自动同步到寄存器）、MLX90640 的 I2C 地址。后面的 816 个字全部是每个像素的校正或者测量参数，数据手册也没有写每个数据到底是什么意思，直接用就是了，不要问为什么。

RAM

地址范围为 0x0400~0x073F，共 832 个字（1664 字节），前 768 个字是实时的 768 像素的测量数据，后面 64 个字是与当前刚刚测量完成的一半像素有关的计算因数。RAM 是只读的。

寄存器

地址范围为 0x8000~0x8010，共 16 个字（32 字节），其中用户可以访问的有状态寄存器 0x8000、控制寄存器 0x800D，改变控制寄存器可以直接控制 MLX90640 的运行行为，是即有读又有写的部分。

三部分建议的操作

上电后读取一下 EEPROM，扔给 API 函数 MLX90640_ExtractParameters 得到参数项变量。根据需要修改控制寄存器的值。循环读取状态寄存器，当有新的数据测量完成时读取全部 RAM 扔给 API 函数 MLX90640_CalculateTo 得到每个像素的温度值。

状态寄存器说明

状态寄存器从字面来理解应该是只读的，但数据手册里却定义了一些位是参数（可修改的），不管寄存器叫什么了，根据参数功能来用吧。

STA[15:5]：保留，只能写入 0

STA[4]：RAM 是否可写，0：不可写；1：可写。在帧测量完成后，是否允许 MLX90640 将测量的数据写入（更新）到 RAM 里，这个功能可以在读取一帧数据的过程中设置为不允许，即：当上位机正在读取 RAM 的过程中，不允许再更新 RAM。这个位同时还受控制寄存器中的 bit2 的限制，当 CTR[2]=0 时，无论这个位怎样设置，都会自动更新 RAM，仅当 CTR[2]=1 时，STA[4]参数才会起作用。

STA[3]：子页测量完成标志位。0：表示没有完成；1 表示已经完成了一帧的测量。

STA[2:0]：刚刚完成的是帧 0 还是帧 1。

控制寄存器说明

CTR[15:13]：保留，只能写入 0

CTR[12]：帧交错模式，默认为 1（像素交错模式），数据手册上说了只有这种模式才能保证精度，那就不要改这一位，保持默认 1。

CTR[11:10]：测量精度，00~11 表示 16 位~19 位，默认为 10（18 位）。改成 11 也没什么效果，所以这个参数也保持默认吧。

CTR[9:7]：设置测量速率，0~7 表示 0.5、1、2、4、8、16、32、64Hz，默认为 010（2Hz），这个参数是唯一有用的参数。

CTR[6:4]：手动测量时，指定要测量哪个子页（帧 0 还是帧 1）。数据手册已经把手动测量部分删除了，所以手动测量相关的参数可以忽略。

CTR[3]：手动测量还是自动测量，默认为 0（自动测量），不要使用手动测量（原因同上）。

CTR[2]：如何更新 RAM。0：测量完成后自动更新；1：根据 STA[4]参数。

CTR[1]：保留，只能写入 0。

CTR[0]：0：所有数据更新在一页里；1：使能子页模式（页 0 和页 1），默认