第三十八章 摄像头实验

ALIENTEK 精英 STM32 开发板板载了一个摄像头接口（P6），该接口可以用来连接 ALIENTEK OV7670 摄像头模块。本章，我们将使用 STM32 驱动 ALIENTEK OV7670 摄像头模块，实现摄像头功能。本章分为如下几个部分：

38.1 OV7670 简介

38.2 硬件设计

38.3 软件设计

38.4 下载验证

38.1 OV7670 简介

OV7670 是 OV（OmniVision）公司生产的一颗 1/6 寸的 CMOS VGA 图像传感器。该传感器体积小、工作电压低，提供单片 VGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、取窗口等方式的各种分辨率 8 位影像数据。该产品 VGA 图像最高达到 30 帧/秒。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、度、色度等都可以通过 SCCB 接口编程。OmmiVision 图像传感器应用独有的传感器技术，通过减少或消除光学或电子缺陷如固定图案噪声、托尾、浮散等，提高图像质量，得到清晰的稳定的彩色图像。

OV7670 的特点有：

- 高灵敏度、低电压适合嵌入式应用
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(GBR4:2:2，RGB565/RGB555/RGB444)，YUV(4:2:2)和 YCbCr（4:2：2）输出格式
- 支持 VGA、CIF，和从 CIF 到 40*30 的各种尺寸输出
- 支持自动曝光控制、自动增益控制、自动白平衡、自动消除灯光条纹、自动黑电平校准等自动控制功能。同时支持色饱和度、色相、伽马、锐度等设置。
- 支持闪光灯
- 支持图像缩放

OV7670 的功能框图图如图 38.1.1 所示：

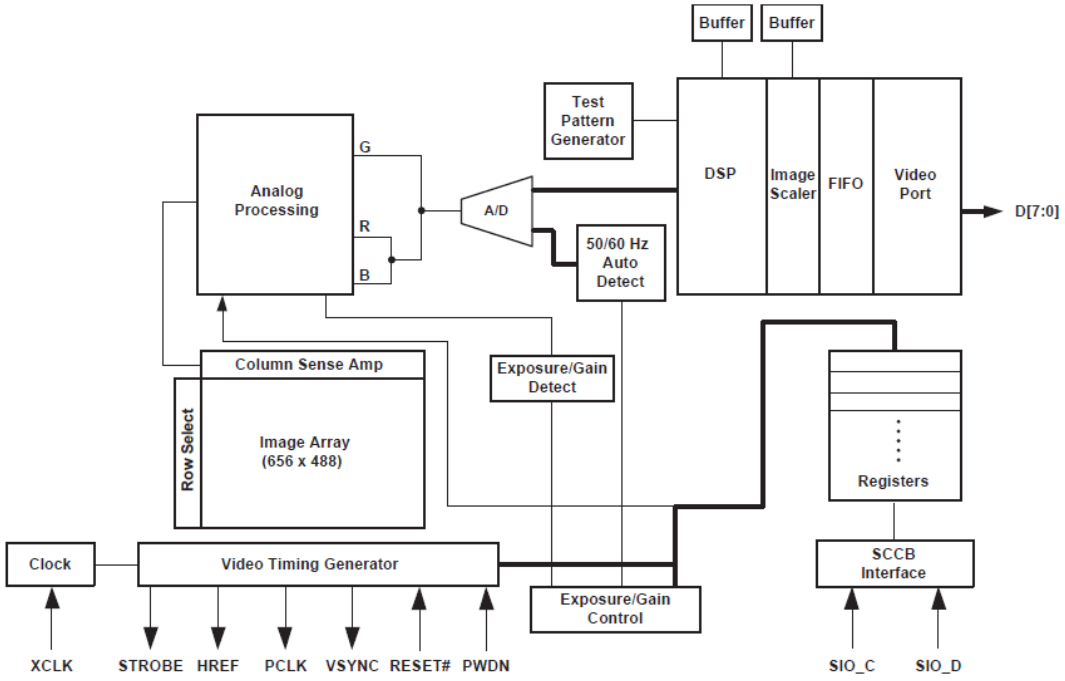

图 38.1.1 OV7670 功能框图

OV7670 传感器包括如下一些功能模块。

1.感光整列（Image Array）

OV7670 总共有 656*488 个像素，其中 640*480 个有效（即有效像素为 30W）。

2.时序发生器（Video Timing Generator）

时序发生器具有的功能包括：整列控制和帧率发生（7 种不同格式输出）、内部信号发生器和分布、帧率时序、自动曝光控制、输出外部时序（VSYNC、HREF/HSYNC 和 PCLK）。

3.模拟信号处理（Analog Processing）

模拟信号处理所有模拟功能，并包括：自动增益（AGC）和自动白平衡（AWB）。

4.A/D 转换（A/D）

原始的信号经过模拟处理器模块之后 ，分G和BR两路进入一个10 位的A/D 转换器，A/D 转换器工作在 12M 频率，与像素频率完全同步（转换的频率和帧率有关）。

除 A/D 转换器外，该模块还有以下三个功能：

- 黑电平校正（BLC)
- U/V 通道延迟
- A/D 范围控制

A/D 范围乘积和 A/D 的范围控制共同设置 A/D 的范围和最大值，允许用户根据应用调整图片的亮度。

5.测试图案发生器（Test Pattern Generator）

测试图案发生器功能包括:八色彩色条图案、渐变至黑白彩色条图案和输出脚移位“1”。

6.数字处理器（DSP）

这个部分控制由原始信号插值到 RGB 信号的过程，并控制一些图像质量：

- 边缘锐化（二维高通滤波器）
- 颜色空间转换（ 原始信号到 RGB 或者 YUV/YCbYCr)
- RGB 色彩矩阵以消除串扰
- 色相和饱和度的控制
- 黑/白点补偿
- 降噪
- 镜头补偿
- 可编程的伽玛
- 十位到八位数据转换

7.缩放功能（Image Scaler）

这个模块按照预先设置的要求输出数据格式，能将 YUV/RGB 信号从 VGA 缩小到 CIF 以下的任何尺寸。

8.数字视频接口（Digital Video Port）

通过寄存器 COM2[1:0]，调节 IOL/IOH 的驱动电流，以适应用户的负载。

9.SCCB 接口（SCCB Interface）

SCCB 接口控制图像传感器芯片的运行，详细使用方法参照光盘的《OmniVision Technologies Seril Camera Control Bus(SCCB) Specification》这个文档

10.LED 和闪光灯的输出控制（LED and Storbe Flash Control Output）

OV7670 有闪光灯模式，可以控制外接闪光灯或闪光 LED 的工作。

OV7670 的寄存器通过 SCCB 时序访问并设置，SCCB 时序和 IIC 时序十分类似，在本章我们不做介绍，请大家参考光盘的相关文档。

接下来我们介绍一下 OV7670 的图像数据输出格式。首先我们简单介绍几个定义：

VGA，即分辨率为 640*480 的输出模式；

QVGA，即分辨率为 320*240 的输出格式，也就是本章我们需要用到的格式；

QQVGA，即分辨率为 160*120 的输出格式；

PCLK，即像素时钟，一个 PCLK 时钟，输出一个像素(或半个像素)。

VSYNC，即帧同步信号。

HREF /HSYNC，即行同步信号。

OV7670 的图像数据输出（通过 D[7:0]）就是在 PCLK，VSYNC 和 HREF/ HSYNC 的控制下进行的。首先看看行输出时序，如图 38.1.2 所示：

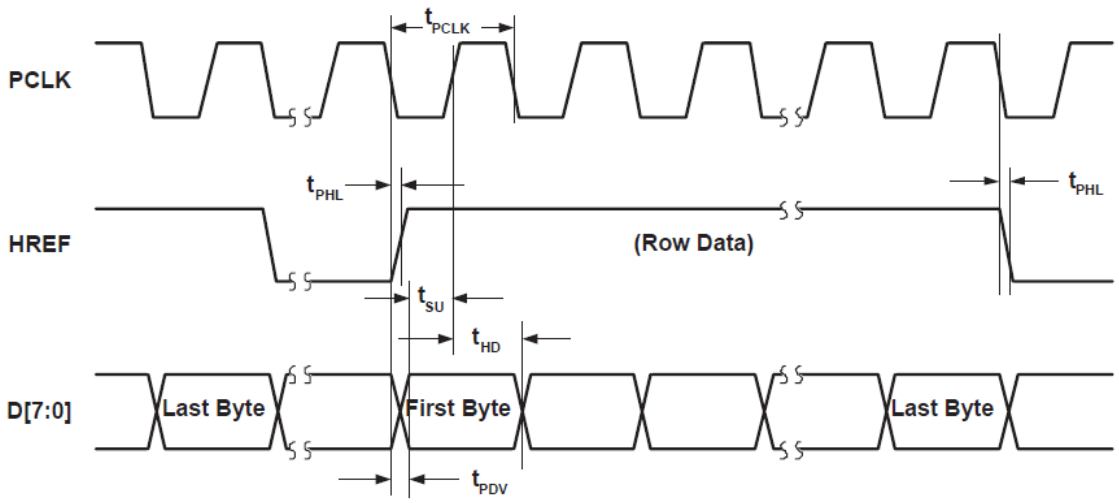

图 38.1.2 OV7670 行输出时序

从上图可以看出，图像数据在 HREF 为高的时候输出，当 HREF 变高后，每一个 PCLK 时钟，输出一个字节数据。比如我们采用 VGA 时序，RGB565 格式输出，每 2 个字节组成一个像素的颜色（高字节在前，低字节在后），这样每行输出总共有 640*2 个 PCLK 周期，输出 640*2 个字节。

再来看看帧时序（VGA 模式），如图 38.1.3 所示：

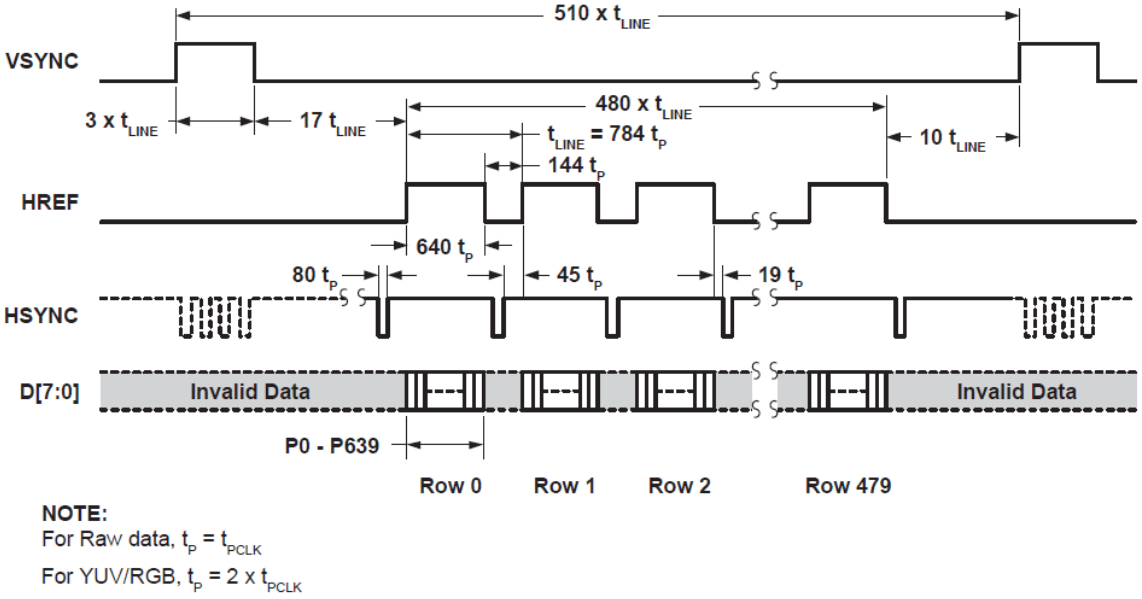

图 38.1.3 OV7670 帧时序

上图清楚的表示了 OV7670 在 VGA 模式下的数据输出，注意，图中的 HSYNC 和 HREF 其实是同一个引脚产生的信号，只是在不同场合下面，使用不同的信号方式，我们本章用到的是 HREF。

因为 OV7670 的像素时钟（PCLK）最高可达 24Mhz，我们用 STM32F103ZET6 的 IO 口直接抓取，是非常困难的，也十分占耗 CPU（可以通过降低 PCLK 输出频率，来实现 IO 口抓取，但是不推荐）。所以，本章我们并不是采取直接抓取来自 OV7670 的数据，而是通过 FIFO 读取，ALIENTEK OV7670 摄像头模块自带了一个 FIFO 芯片，用于暂存图像数据，有了这个芯片，我们就可以很方便的获取图像数据了，而不再需要单片机具有高速 IO，也不会耗费多少 CPU，可以说，只要是个单片机，都可以通过 ALIENTEK OV7670 摄像头模

块实现拍照的功能。

接下来我们介绍一下 ALIENTEK OV7670 摄像头模块。该模块的外观如图 38.1.4：

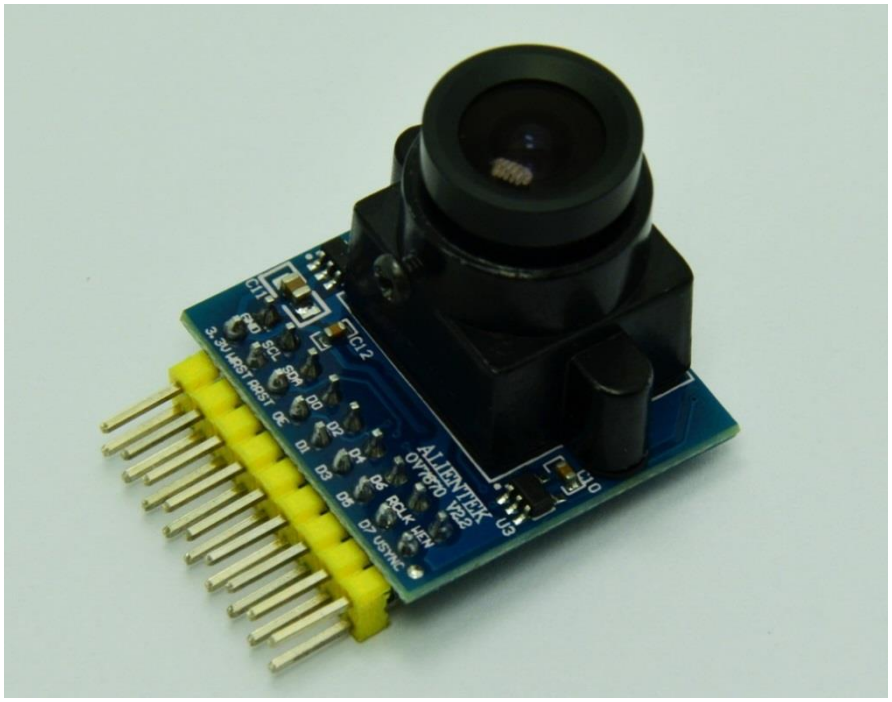

图 38.1.4 ALIENTEK OV7670 摄像头模块外观图

模块原理图如图 38.1.5 所示：

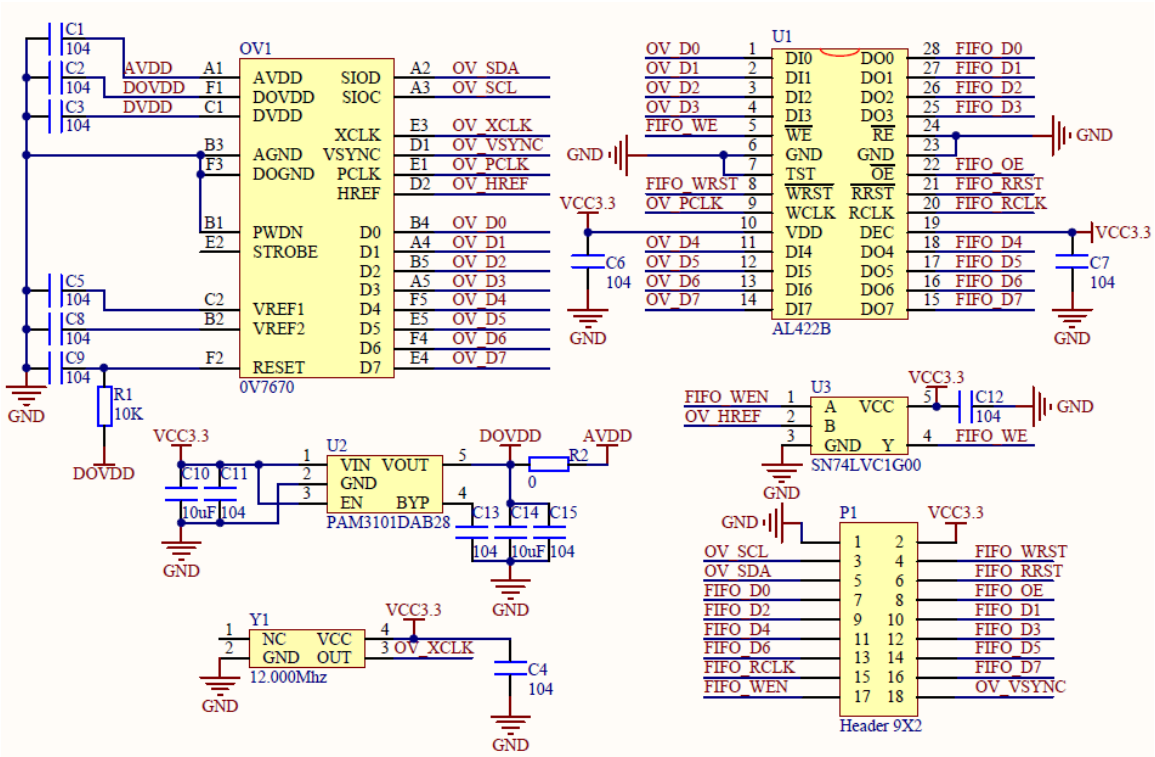

图 38.1.5 ALIENTEK OV7670 摄像头模块原理图

从上图可以看出，ALIENTEK OV7670 摄像头模块自带了有源晶振，用于产生 12M 时钟作为 OV7670 的 XCLK 输入。同时自带了稳压芯片，用于提供 OV7670 稳定的 2.8V 工作电压，并带有一个 FIFO 芯片（AL422B），该 FIFO 芯片的容量是 384K 字节，足够存储 2

帧 QVGA 的图像数据。模块通过一个 2*9 的双排排针（P1）与外部通信，与外部的通信信号如表 38.1.1 所示：

信号	作用描述	信号	作用描述
VCC3.3	模块供电脚，接 3.3V 电源	FIFO_WEN	FIFO 写使能
GND	模块地线	FIFO_WRST	FIFO 写指针复位
OV_SCL	SCCB 通信时钟信号	FIFO_RRST	FIFO 读指针复位
OV_SDA	SCCB 通信数据信号	FIFO_OE	FIFO 输出使能（片选）
FIFO_D[7:0]	FIFO 输出数据（8 位）	OV_VSYNC	OV7670 帧同步信号
FIFO_RCLK	读 FIFO 时钟		

表 38.1.1 OV7670 模块信号及其作用描述

下面我们来看看如何使用 ALIENTEK OV7670 摄像头模块（以 QVGA 模式，RGB565 格式为例）。对于该模块，我们只关心两点：1，如何存储图像数据；2，如何读取图像数据。

首先，我们来看如何存储图像数据。

ALIENTEK OV7670 摄像头模块存储图像数据的过程为：等待 OV7670 同步信号→FIFO 写指针复位→FIFO 写使能→等待第二个 OV7670 同步信号→FIFO 写禁止。通过以上 5 个步骤，我们就完成了 1 帧图像数据的存储。

接下来，我们来看看如何读取图像数据。

在存储完一帧图像以后，我们就可以开始读取图像数据了。读取过程为：FIFO 读指针复位→给 FIFO 读时钟（FIFO_RCLK）→读取第一个像素高字节→给 FIFO 读时钟→读取第一个像素低字节→给 FIFO 读时钟→读取第二个像素高字节→循环读取剩余像素→结束。

可以看出，ALIENTEK OV7670 摄像头模块数据的读取也是十分简单，比如 QVGA 模式，RGB565 格式，我们总共循环读取 320*240*2 次，就可以读取 1 帧图像数据，把这些数据写入 LCD 模块，我们就可以看到摄像头捕捉到的画面了。

OV7670 还可以对输出图像进行各种设置，详见光盘《OV7670 中文数据手册 1.01》和《OV7670 software application note》这两个文档，对 AL422B 的操作时序，请大家参考 AL422B 的数据手册。

了解了 OV7670 模块的数据存储和读取，我们就可以开始设计代码了，本章，我们用一个外部中断，来捕捉帧同步信号（VSYNC），然后在中断里面启动 OV7670 模块的图像数据存储，等待下一次 VSHNC 信号到来，我们就关闭数据存储，然后一帧数据就存储完成了，在主函数里面就可以慢慢的将这一帧数据读出来，放到 LCD 即可显示了，同时开始第二帧数据的存储，如此循环，实现摄像头功能。

本章，我们将使用摄像头模块的 QVGA 输出（320*240），刚好和精英 STM32 开发板使用的 LCD 模块分辨率一样，一帧输出就是一屏数据，提高速度的同时也不浪费资源。注意：ALIENTEK OV7670 摄像头模块自带的 FIFO 是没办法缓存一帧的 VGA 图像的，如果使用 VGA 输出，那么你必须在 FIFO 写满之前开始读 FIFO 数据，保证数据不被覆盖。

38.2 硬件设计

本章实验功能简介：开机后，初始化摄像头模块（OV7670），如果初始化成功，则在 LCD 模块上面显示摄像头模块所拍摄到的内容。我们可以通过 KEY0 设置光照模式（5 种模式）、通过 KEY1 设置色饱和度，通过 KEY_UP 设置对比度，通过 TPAD 设置特效（总共 7 种特效）。通过串口，我们可以查看当前的帧率（这里是指 LCD 显示的帧率，而不是指 OV7670 的输出帧率），同时可以借助 USMART 设置 OV7670 的寄存器，方便大家调试。DS0 指示程序运行状态。

本实验用到的硬件资源有：

1） 指示灯 DS0

2） KEY0/KEY1/KEY_UP 和 TPAD 按键

3） 串口

4） TFTLCD 模块

5） 摄像头模块

ALIENTEK OV7670 摄像头模块在 38.1 节已经有详细介绍过，这里我们主要介绍该模块与 ALIETEK 精英 STM32 开发板的连接。

在开发板的左下角的 2*9 的 P6 排座，是摄像头模块/OLED 模块共用接口，在第十七章，我们曾简单介绍过这个接口。本章，我们只需要将 ALIENTEK OV7670 摄像头模块插入这个接口（P4）即可，该接口与 STM32 的连接关系如图 38.2.1 所示：

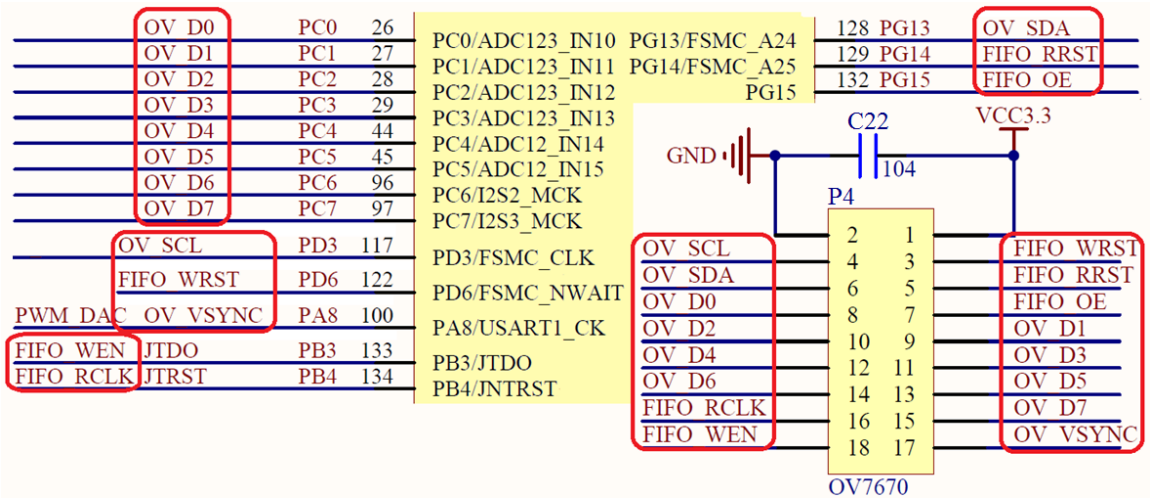

图 38.2.1 摄像头模块接口与 STM32 连接图

从上图可以看出，OV7670 摄像头模块的各信号脚与 STM32 的连接关系为：

OV_SDA 接 PG13；

OV_SCL 接 PD3；

FIFO_RCLK 接 PB4；

FIFO_WEN 接 PB3；

FIFO_WRST 接 PD6；

FIFO_RRST 接 PG14；

FIFO_OE 接 PG15；

OV_VSYNC 接 PA8；

OV_D[7:0]接 PC[7:0]；

这些线的连接，精英 STM32 的内部已经连接好了，我们只需要将 OV7670 摄像头模块插上去就好了。注意，其中有几个信号线和其他外设共用了，OV_SCL 与 JOY_CLK 共用 PD3，所以摄像头和手柄不可以同时使用；FIFO_WEN 和 FIFO_RCLK 则和 JTAG 的信号线 JTDO 和 JTRST 共用了，所以使用摄像头的时候，不能使用 JTAG 模式调试，而应该选择 SW 模式（SW 模式不需要用到 JTDO 和 JTRST）；OV_VSYNC 和 PWM_DAC 共用了 PA8，所以他们也不可以同时使用。

实物连接如图 38.2.2 所示：

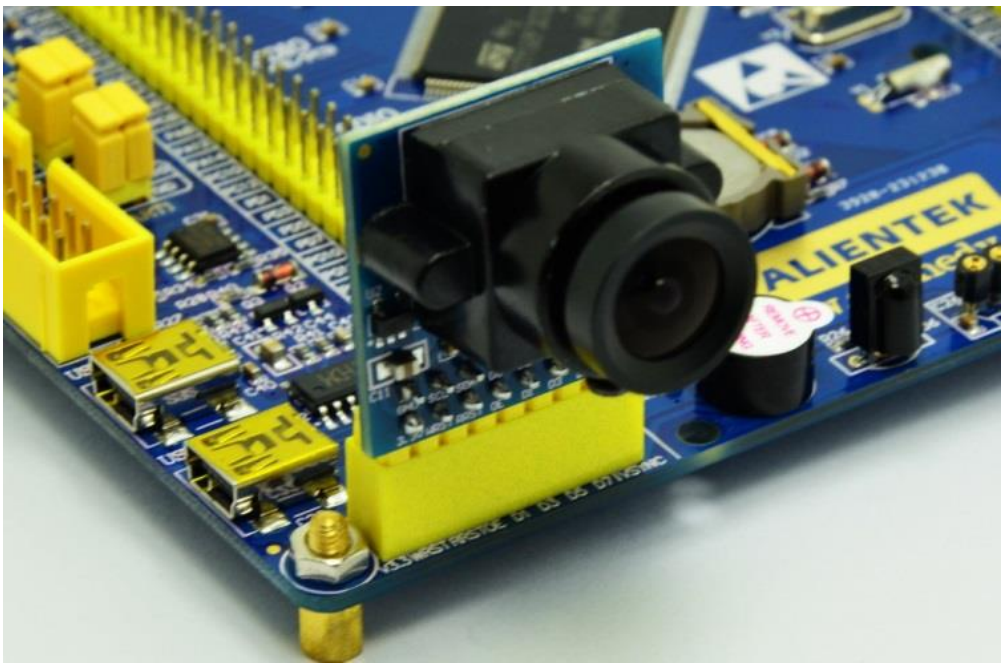

图 38.2.2 OV7670 摄像头模块与开发板连接实物图

38.3 软件设计

打开我们摄像头实验的工程，可以看到我们的工程中多了 ov7670.c 和 sccb.c 源文件，以及头文件 ov7670.h、sccb.h 和 ov7670cfg.h 等 5 个文件。

本章总共新增了 5 个文件，代码比较多，我们就不一一列出了，仅挑两个重要的地方进行讲解。首先，我们来看 ov7670.c 里面的 OV7670_Init 函数，该函数代码如下：

```
u8 OV7670_Init(void)
{
    u8 temp;
    u16 i=0;
    GPIO_InitTypeDef    GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB|
    RCC_APB2Periph_GPIOC|RCC_APB2Periph_GPIOD|
    RCC_APB2Periph_GPIOG, ENABLE);          //使能相关端口时钟

    GPIO_InitStructure.GPIO_Pin    = GPIO_Pin_8;        //PA8  输入  上拉
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);              //初始化 GPIOA.8

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4;        // 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;          //推挽输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_SetBits(GPIOB,GPIO_Pin_3|GPIO_Pin_4);     //初始化 GPIO

    GPIO_InitStructure.GPIO_Pin    = 0xff; //PC0~7  输入  上拉
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
```

```
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    GPIO_SetBits(GPIOD,GPIO_Pin_6);

    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_14|GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOG, &GPIO_InitStructure);
    GPIO_SetBits(GPIOG,GPIO_Pin_14|GPIO_Pin_15);

    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable,ENABLE);   //SWD
    SCCB_Init();                    //初始化 SCCB 的 IO 口
    if(SCCB_WR_Reg(0x12,0x80))return 1;     //复位 SCCB
    delay_ms(50);
    //读取产品型号
    temp=SCCB_RD_Reg(0x0b);
    if(temp!=0x73)return 2;
    temp=SCCB_RD_Reg(0x0a);
    if(temp!=0x76)return 2;
    //初始化序列
    for(i=0;i<sizeof(ov7670_init_reg_tbl)/sizeof(ov7670_init_reg_tbl[0])/2;i++)
    {
        SCCB_WR_Reg(ov7670_init_reg_tbl[i][0],ov7670_init_reg_tbl[i][1]);
        delay_ms(2);
    }
    return 0x00;   //ok
}
```

此部分代码先初始化 OV7670 相关的 IO 口（包括 SCCB_Init），然后最主要的是完成 OV7670 的寄存器序列初始化。OV7670 的寄存器特多（百几十个），配置特麻烦，幸好厂家有提供参考配置序列（详见《OV7670 software application note》），本章我们用到的配置序列，存放在 ov7670_init_reg_tbl 这个数组里面，该数组是一个 2 维数组，存储初始化序列寄存器及其对应的值，该数组存放在 ov7670cfg.h 里面。

接下来，我们看看 ov7670cfg.h 里面 ov7670_init_reg_tbl 的内容，ov7670cfg.h 文件的代码如下：

```
//初始化寄存器序列及其对应的值
const u8 ov7670_init_reg_tbl[][2]=
{
    /*以下为 OV7670 QVGA RGB565 参数  */
    {0x3a, 0x04},//
    {0x40, 0x10},
    {0x12, 0x14},//QVGA,RGB 输出
    ……省略部分设置
```

```
        {0x6e, 0x11},//100
        {0x6f, 0x9f},//0x9e for advance AWB
        {0x55, 0x00},//亮度
        {0x56, 0x40},//对比度
        {0x57, 0x80},//0x40,  change according to Jim's request
    };
```

以上代码，我们省略了很多（全部贴出来太长了），我们大概了解下结构，每个条目的第一个字节为寄存器号（也就是寄存器地址），第二个字节为要设置的值，比如{0x3a, 0x04}，就表示在 0X03 地址，写入 0X04 这个值。

通过这么一长串（110 多个）寄存器的配置，我们就完成了 OV7670 的初始化，本章我们配置 OV7670 工作在 QVGA 模式，RGB565 格式输出。 在完成初始化之后，我们既可以开始读取 OV7670 的数据了。

OV7670 文件夹里面的其他代码我们就不逐个介绍了，请大家参考光盘该例程源码。

因为本章我们还用到了帧率（LCD 显示的帧率）统计和中断处理，所以我们还需要修改 timer.c、timer.h、exti.c 及 exti.h 这几个文件。

在 timer.c 里面，我们新增 TIM6_Int_Init 和 TIM6_IRQHandler 两个函数，用于统计帧率，增加代码如下：

```
    u8 ov_frame;  //统计帧数
    //定时器 6 中断服务程序
    void TIM6_IRQHandler(void)
    {     if (TIM_GetITStatus(TIM6, TIM_IT_Update) != RESET) //更新中断发生
          {
                printf("frame:%dfps\r\n",ov_frame);     //打印帧率
                ov_frame=0;
          }
              TIM_ClearITPendingBit(TIM6, TIM_IT_Update    );  //清中断标志位

    }
    //基本定时器 6 中断初始化
    //这里时钟选择为 APB1 的 2 倍，而 APB1 为 36M
    //arr：自动重装值。
    //psc：时钟预分频数
    //这里使用的是定时器 3!
    void TIM6_Int_Init(u16 arr,u16 psc)
    {
          TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
          NVIC_InitTypeDef NVIC_InitStructure;
          RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE); //时钟使能

          TIM_TimeBaseStructure.TIM_Period = arr; //自动重装载周期值
          TIM_TimeBaseStructure.TIM_Prescaler =psc; //预分频值
          TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTS = Tck_tim
          TIM_TimeBaseStructure.TIM_CounterMode  =  TIM_CounterMode_Up;   //向上计数
模式
```

```
        TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure); //根据指定的参数初始化TIMx

        TIM_ITConfig( TIM6,TIM_IT_Update|TIM_IT_Trigger,ENABLE);//使能更新触发中断
        TIM_Cmd(TIM6, ENABLE);  //使能 TIMx 外设
        NVIC_InitStructure.NVIC_IRQChannel = TIM6_IRQn;          //TIM3 中断
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;  //先占优先级 0 级
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;          //从优先级 3 级
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;          //IRQ 通道被使能
        NVIC_Init(&NVIC_InitStructure);  //根据指定的参数初始化外设 NVIC 寄存器

    }
```

这里，我们用到基本定时器 TIM6 来统计帧率，也就是 1 秒钟中断一次，打印 ov_frame 的值，ov_frame 用于统计 LCD 帧率。

再在 timer.h 里面添加 TIM6_Int_Init 函数的定义，就完成对 timer.c 和 timer.h 的修改了。

在 exti.c 里面添加 EXTI8_Init 和 EXTI9_5_IRQHandler 函数，用于 OV7670 模块的 FIFO 写控制，exti.c 文件新增部分代码如下：

```
    //ov_sta:0,开始一帧数据采集
    u8 ov_sta;      //帧中断标记
     //外部中断 5~9 服务程序
    void EXTI9_5_IRQHandler(void)
    {
        if(EXTI_GetITStatus(EXTI_Line8)==SET) //是 8 线的中断
        {
            OV7670_WRST=0;      //复位写指针
            OV7670_WRST=1;
            OV7670_WREN=1;      //允许写入 FIFO
            ov_sta++;           //帧中断加 1
        }
        EXTI_ClearITPendingBit(EXTI_Line8);  //清除 EXTI8 线路挂起位

    }
    //外部中断 8 初始化
    void EXTI8_Init(void)
    {
        EXTI_InitTypeDef EXTI_InitStructure;
        NVIC_InitTypeDef NVIC_InitStructure;
        GPIO_EXTILineConfig(GPIO_PortSourceGPIOA,GPIO_PinSource8);//PA8 对中断线8

        EXTI_InitStructure.EXTI_Line=EXTI_Line8;
        EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
        EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
```

```
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);        //根据指定的参数初始化外设 EXTI 寄存器

    NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;      //使能外部中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //抢占优先级 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;          //子优先级 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //使能外部中断通道
    NVIC_Init(&NVIC_InitStructure);          //根据指定的参数初始化外设 NVIC 寄存器
}
```

因为 OV7670 的帧同步信号（OV_VSYNC）接在 PA8 上面，所以我们这里配置 PA8 作为中端输入，因为 STM32 的外部中断 5~9 共用一个中端服务函数（EXTI9_5_IRQHandler），所以在该函数里面，我们需要先判断中断是不是来自中断线 8 的，然后再做处理。

中断处理部分流程：每当帧中断到来后，先判断 ov_sta 的值是否为 0，如果是 0，说明可以往 FIFO 里面写入数据，执行复位 FIFO 写指针，并允许 FIFO 写入，此时，AL422B 将从地址 0 开始，存储新一帧的图像数据。然后设置 ov_sta++即可，标记新的一帧数据正在存储中。如果 ov_sta 不为 0，说明之前存储在 FIFO 里面的一帧数据还未被读取过，直接禁止 FIFO 写入，等待 MCU 读取 FIFO 数据，以免数据覆盖。

然后，STM32 只需要判断 ov_sta 是否大于 0，来读取 FIFO 里面的数据，读完一帧后，设置 ov_sta 为 0，以免重复读取，同时还可以时能 FIFO 新帧的写入。

再在 exti.h 里面添加 EXTI8_Init 函数的定义，就完成对 exti.c 和 exti.h 的修改了。

最后，打开 main.c 文件，代码如下：

```
const  u8*LMODE_TBL[5]={"Auto","Sunny","Cloudy","Office","Home"};//5  种光照模式

const u8*EFFECTS_TBL[7]={"Normal","Negative","B&W","Redish","Greenish",
                                                    "Bluish","Antique"};//7 种特效
extern u8 ov_sta;   //在 exit.c 里  面定义
extern u8 ov_frame;      //在 timer.c 里面定义
//更新 LCD 显示
void camera_refresh(void)
{
    u32 j;
    u16 color;
    if(ov_sta)//有帧中断更新？
    {
        LCD_Scan_Dir(U2D_L2R);        //从上到下,从左到右
        if(lcddev.id==0X1963)
            LCD_Set_Window((lcddev.width-240)/2,(lcddev.height-320)/2,240,320);

        else if(lcddev.id==0X5510||lcddev.id==0X5310)
            LCD_Set_Window((lcddev.width-320)/2,(lcddev.height-240)/2,320,240);
        LCD_WriteRAM_Prepare();        //开始写入 GRAM
        OV7670_RRST=0;                 //开始复位读指针
        OV7670_RCK_L;
        OV7670_RCK_H;
```

```
            OV7670_RCK_L;
            OV7670_RRST=1;                  //复位读指针结束
            OV7670_RCK_H;
            for(j=0;j<76800;j++)
            {
                OV7670_RCK_L;
                color=GPIOC->IDR&0XFF; //读数据
                OV7670_RCK_H;
                color<<=8;
                OV7670_RCK_L;
                color|=GPIOC->IDR&0XFF; //读数据
                OV7670_RCK_H;
                LCD->LCD_RAM=color;
            }
            ov_sta=0;                       //清零帧中断标记
            ov_frame++;
            LCD_Scan_Dir(DFT_SCAN_DIR);     //恢复默认扫描方向
        }
    }


 int main(void)
 {
      u8 key;
      u8 lightmode=0,saturation=2,contrast=2;
      u8 effect=0;
      u8 i=0;
      u8 msgbuf[15];                    //消息缓存区
      u8 tm=0;
      delay_init();                 //延时函数初始化
      NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置中断优先级分组为组 2
      uart_init(115200);        //串口初始化为 115200
      usmart_dev.init(72);            //初始化 USMART
      LED_Init();                       //初始化与 LED 连接的硬件接口
      KEY_Init();                       //初始化按键
      LCD_Init();                       //初始化 LCD
      TPAD_Init();                  //触摸按键初始化
      POINT_COLOR=RED;              //设置字体为红色
      LCD_ShowString(30,50,200,16,16,"ELITE STM32F103 ^_^");
      LCD_ShowString(30,70,200,16,16,"OV7670    TEST");
      LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
      LCD_ShowString(30,110,200,16,16,"2015/1/18");
      LCD_ShowString(30,130,200,16,16,"KEY0:Light Mode");
      LCD_ShowString(30,150,200,16,16,"KEY1:Saturation");
```

```
LCD_ShowString(30,170,200,16,16,"KEY_UP:Contrast");
LCD_ShowString(30,190,200,16,16,"TPAD:Effects");
LCD_ShowString(30,210,200,16,16,"OV7670 Init...");
while(OV7670_Init())//初始化 OV7670
{
    LCD_ShowString(30,210,200,16,16,"OV7670 Error!!");
    delay_ms(200);
    LCD_Fill(30,210,239,246,WHITE);
    delay_ms(200);
}
LCD_ShowString(30,210,200,16,16,"OV7670 Init OK");
delay_ms(1500);
OV7670_Light_Mode(lightmode);
OV7670_Color_Saturation(saturation);
OV7670_Contrast(contrast);
OV7670_Special_Effects(effect);

TIM6_Int_Init(10000,7199);          //10Khz 计数频率,1 秒钟中断

EXTI8_Init();                       //使能定时器捕获
OV7670_Window_Set(12,176,240,320);  //设置窗口
OV7670_CS=0;
LCD_Clear(BLACK);
while(1)
{
    key=KEY_Scan(0);//不支持连按
    if(key)
    {
        tm=20;
        switch(key)
        {
            case KEY0_PRES: //灯光模式 Light Mode
                lightmode++;
                if(lightmode>4)lightmode=0;
                OV7670_Light_Mode(lightmode);
                sprintf((char*)msgbuf,"%s",LMODE_TBL[lightmode]);
                break;
            case KEY1_PRES: //饱和度 Saturation
                saturation++;
                if(saturation>4)saturation=0;
                OV7670_Color_Saturation(saturation);
                sprintf((char*)msgbuf,"Saturation:%d",(signed char)saturation-2);
                break;
            case WKUP_PRES:     //对比度 Contrast
```

```
                        contrast++;
                        if(contrast>4)contrast=0;
                        OV7670_Contrast(contrast);
                        sprintf((char*)msgbuf,"Contrast:%d",(signed char)contrast-2);
                        break;
                }
            }
            if(TPAD_Scan(0))//检测到触摸按键
            {
                effect++;
                if(effect>6)effect=0;
                OV7670_Special_Effects(effect);//设置特效
                sprintf((char*)msgbuf,"%s",EFFECTS_TBL[effect]);
                tm=20;
            }
            camera_refresh();//更新显示
            if(tm)
            {

LCD_ShowString((lcddev.width-240)/2+30,(lcddev.height-320)/2+60,200,16,16,msgbuf);
                tm--;
            }
            i++;
            if(i==15)//DS0 闪烁.
            {
                i=0;
                LED0=!LED0;
            }
        }
    }
```

此部分代码除了 mian 函数，还有一个 camera_refresh 函数，该函数用于读取摄像头模块自带 FIFO 里面的数据，并显示在 LCD 上面，对分辨率大于 320*240 的屏幕，则通过开窗函数（LCD_Set_Window）将显示区域开窗在屏幕的正中央。注意，为了提高 FIFO 读取速度，我们将 FIFO_RCK 的控制，采用快速 IO 控制，关键代码如下（在 ov7670.h 里面）：

```
#define OV7670_RCK_H     GPIOB->BSRR=1<<4   //设置读数据时钟高电平
#define OV7670_RCK_L     GPIOB->BRR=1<<4    //设置读数据时钟低电平
```

OV7670_RCK_H 和 OV7670_RCK_L 就用到了 BSRR 和 BRR 这两个寄存器，以实现快速 IO 设置，从而提高读取速度。

main 函数的处理则比较简单，我们就不细说了。

前面提到，我们要用 USMART 来设置摄像头的参数，我们只需要在 usmart_nametab 里面添加 SCCB_WR_Reg 和 SCCB_RD_Reg 这两个函数，就可以轻松调试摄像头了。

最后，为了得到最快的显示速度，我们将 MDK 的代码优化等级设置为-O2 级别（在 C/C++选项卡里面设置），这样 LCD 的显示帧率可以达到 18 帧。注意：这里是因为 TPAD_Scan 扫描占用了很多时间（>15ms/次），帧率才是 18 帧，如果屏蔽掉 TPAD_Scan，则可以达到

30 帧。

38.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32 开发板上，得到如图 38.4.1 所示界面：

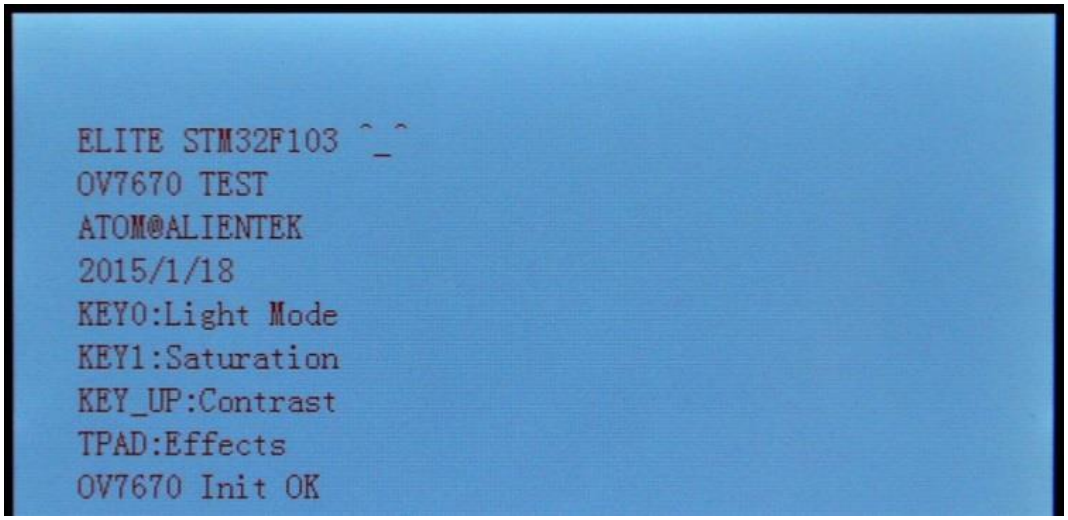

图 38.4.1 程序运行效果图

随后，进入监控界面。此时，我们可以按不同的按键（KEY0、KEY1、KEY_UP、TPAD 等），来设置摄像头的相关参数和模式，得到不同的成像效果。 同时，你还可以在串口，通过 USMART 调用 SCCB_WR_Reg 等函数，来设置 OV7670 的各寄存器，达到调试测试 OV7670 的目的，如图 38.4.2 所示：

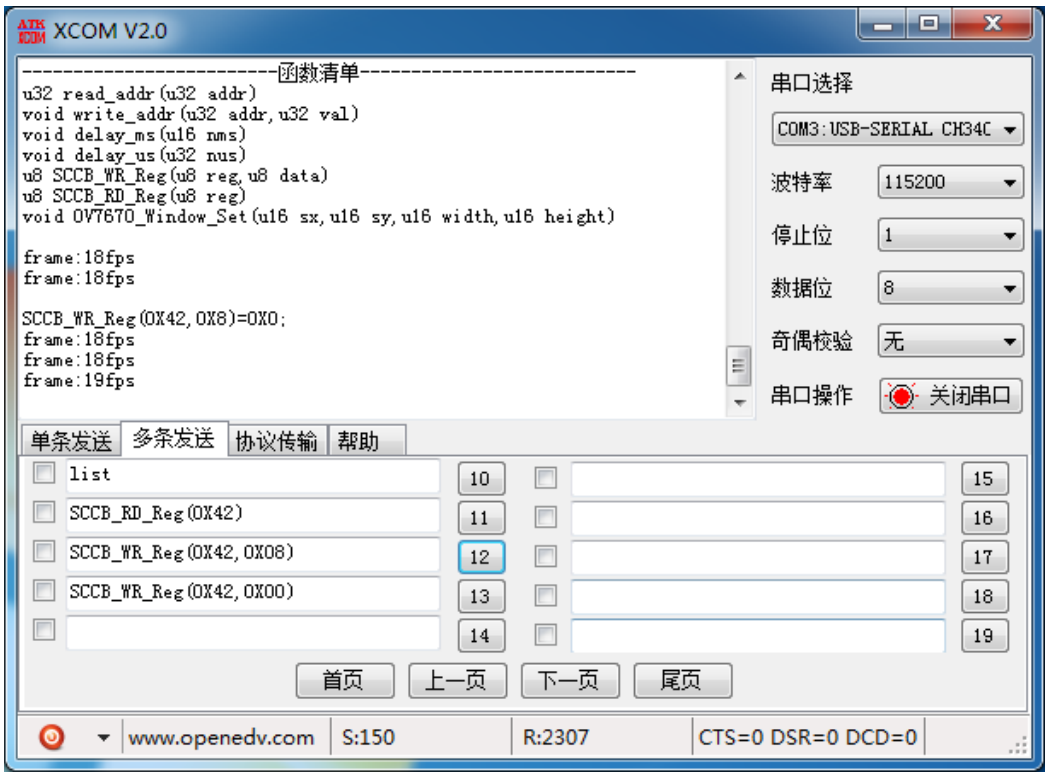

图 38.4.2 USMART 调试 OV7670

从上图还可以看出，LCD 显示帧率为 19 帧左右，而实际上 OV7670 的输出速度是 30

帧（即 OV_VSYNC 的频率）。图中，我们通过 USMART 发送 SCCB_WR_Reg(0X42,0X08)，即可设置 OV7670 输出彩条，方便大家测试。

第四十四章 照相机实验

上一章，我们学习了图片解码，本章我们将学习 BMP 编码，结合前面的摄像头实验，实现一个简单的照相机。本章分为如下几个部分：

44.1 BMP 编码简介

44.2 硬件设计

44.3 软件设计

44.4 下载验证

44.1 BMP 编码简介

上一章，我们学习了各种图片格式的解码。本章，我们介绍最简单的图片编码方法：BMP 图片编码。通过前面的了解，我们知道 BMP 文件是由文件头、位图信息头、颜色信息和图形数据等四部分组成。我们先来了解下这几个部分。

1、BMP 文件头（14 字节）：BMP 文件头数据结构含有 BMP 文件的类型、文件大小和位图起始位置等信息。

```
//BMP 文件头
typedef __packed struct
{
    u16    bfType ;          //文件标志.只对'BM',用来识别 BMP 位图类型
    u32    bfSize ;          //文件大小,占四个字节
    u16    bfReserved1 ;     //保留
    u16    bfReserved2 ;     //保留
    u32    bfOffBits ;       //从文件开始到位图数据(bitmap data)开始之间的偏移量
}BITMAPFILEHEADER ;
```

2、位图信息头（40 字节）：BMP 位图信息头数据用于说明位图的尺寸等信息。

```
typedef __packed struct
{
    u32 biSize ;              //说明 BITMAPINFOHEADER 结构所需要的字数。
    long    biWidth ;         //说明图象的宽度，以象素为单位
    long    biHeight ;        //说明图象的高度，以象素为单位
    u16    biPlanes ;         //为目标设备说明位面数，其值将总是被设为 1
    u16    biBitCount ;       //说明比特数/象素，其值为 1、4、8、16、24、或 32
    u32 biCompression ;       //说明图象数据压缩的类型。其值可以是下述值之一：
    //BI_RGB：没有压缩；
    //BI_RLE8：每个象素 8 比特的 RLE 压缩编码，压缩格式由 2 字节组成
    //BI_RLE4：每个象素 4 比特的 RLE 压缩编码，压缩格式由 2 字节组成
    //BI_BITFIELDS：每个象素的比特由指定的掩码决定。
    u32 biSizeImage ;//说明图象的大小,以字节为单位。当用 BI_RGB 格式时,可设置为
0
    long    biXPelsPerMeter ;//说明水平分辨率，用象素/米表示
    long    biYPelsPerMeter ;//说明垂直分辨率，用象素/米表示
    u32 biClrUsed ;           //说明位图实际使用的彩色表中的颜色索引数
    u32 biClrImportant ;      //说明对图象显示有重要影响的颜色索引的数目，
                              //如果是 0，表示都重要。
}BITMAPINFOHEADER ;
```

3、颜色表：颜色表用于说明位图中的颜色，它有若干个表项，每一个表项是一个 RGBQUAD 类型的结构，定义一种颜色。

```
typedef __packed struct
{
    u8 rgbBlue ;          //指定蓝色强度
    u8 rgbGreen ;         //指定绿色强度
    u8 rgbRed ;           //指定红色强度
```

```
    u8 rgbReserved ;  //保留，设置为 0
}RGBQUAD ;
```

颜色表中 RGBQUAD 结构数据的个数由 biBitCount 来确定：当 biBitCount=1、4、8 时，分别有 2、16、256 个表项；当 biBitCount 大于 8 时，没有颜色表项。

BMP 文件头、位图信息头和颜色表组成位图信息（我们将 BMP 文件头也加进来，方便处理），BITMAPINFO 结构定义如下:

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
}BITMAPINFO;
```

4、位图数据：位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右，扫描行之间是从下到上。位图的一个像素值所占的字节数:

当 biBitCount=1 时，8 个像素占 1 个字节;

当 biBitCount=4 时，2 个像素占 1 个字节;

当 biBitCount=8 时，1 个像素占 1 个字节;

当 biBitCount=16 时，1 个像素占 2 个字节;

当 biBitCount=24 时，1 个像素占 3 个字节;

当 biBitCount=32 时，1 个像素占 4 个字节;

biBitCount=1 表示位图最多有两种颜色，缺省情况下是黑色和白色，你也可以自己定义这两种颜色。图像信息头装调色板中将有两个调色板项，称为索引 0 和索引 1。图象数据阵列中的每一位表示一个像素。如果一个位是 0，显示时就使用索引 0 的 RGB 值，如果位是 1，则使用索引 1 的 RGB 值。

biBitCount=16 表示位图最多有 65536 种颜色。每个像素用 16 位（2 个字节）表示。这种格式叫作高彩色，或叫增强型 16 位色，或 64K 色。它的情况比较复杂，当 biCompression 成员的值是 BI_RGB 时，它没有调色板。16 位中，最低的 5 位表示蓝色分量，中间的 5 位表示绿色分量，高的 5 位表示红色分量，一共占用了 15 位，最高的一位保留，设为 0。这种格式也被称作 555 16 位位图。如果 biCompression 成员的值是 BI_BITFIELDS，那么情况就复杂了，首先是原来调色板的位置被三个 DWORD 变量占据，称为红、绿、蓝掩码。分别用于描述红、绿、蓝分量在 16 位中所占的位置。在 Windows 95（或 98）中，系统可接受两种格式的位域：555 和 565，在 555 格式下，红、绿、蓝的掩码分别是：0x7C00、0x03E0、0x001F，而在 565 格式下，它们则分别为：0xF800、0x07E0、0x001F。你在读取一个像素之后，可以分别用掩码“与”上像素值，从而提取出想要的颜色分量（当然还要再经过适当的左右移操作）。在 NT 系统中，则没有格式限制，只不过要求掩码之间不能有重叠。（注：这种格式的图像使用起来是比较麻烦的，不过因为它的显示效果接近于真彩，而图像数据又比真彩图像小的多，所以，它更多的被用于游戏软件）。

biBitCount=32 表示位图最多有 4294967296(2 的 32 次方)种颜色。这种位图的结构与 16 位位图结构非常类似，当 biCompression 成员的值是 BI_RGB 时，它也没有调色板，32 位中有 24 位用于存放 RGB 值，顺序是：最高位—保留，红 8 位、绿 8 位、蓝 8 位。这种格式也被成为 888 32 位图。如果 biCompression 成员的值是 BI_BITFIELDS 时，原来调色板的位置将被三个 DWORD 变量占据，成为红、绿、蓝掩码，分别用于描述红、绿、蓝分量在 32 位中所占的位置。在 Windows 95(or 98)中，系统只接受 888 格式，也就是说三个掩码的值将只能是：0xFF0000、0xFF00、0xFF。而在 NT 系统中，你只要注意使掩码之间不产生

重叠就行。（注：这种图像格式比较规整，因为它是 DWORD 对齐的，所以在内存中进行图像处理时可进行汇编级的代码优化（简单））。

通过以上了解，我们对 BMP 有了一个比较深入的了解，本章，我们采用 16 位 BMP 编码（因为我们的 LCD 就是 16 位色的，而且 16 位 BMP 编码比 24 位 BMP 编码更省空间），故我们需要设置 biBitCount 的值为 16，这样得到新的位图信息（BITMAPINFO）结构体：

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    u32 RGB_MASK[3];              //调色板用于存放 RGB 掩码.
}BITMAPINFO;
```

其实就是颜色表由 3 个 RGB 掩码代替。最后，我们来看看将 LCD 的显存保存为 BMP 格式的图片文件的步骤：

1）创建 BMP 位图信息，并初始化各个相关信息

这里，我们要设置 BMP 图片的分辨率为 LCD 分辨率、BMP 图片的大小（整个 BMP 文件大小）、BMP 的像素位数（16 位）和掩码等信息。

2）创建新 BMP 文件，写入 BMP 位图信息

我们要保存 BMP，当然要存放在某个地方（文件），所以需要先创建文件，同时先保存 BMP 位图信息，之后才开始 BMP 数据的写入。

3）保存位图数据。

这里就比较简单了，只需要从 LCD 的 GRAM 里面读取各点的颜色值，依次写入第二步创建的 BMP 文件即可。注意：保存顺序（即读 GRAM 顺序）是从左到右，从下到上。

4）关闭文件。

使用 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！这个要特别注意，写完之后，一定要调用 f_close。

BMP 编码就介绍到这里。

44.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡根目录是否存在 PHOTO 文件夹，如果不存在则创建，如果创建失败，则报错（提示拍照功能不可用）。在找到 SD 卡的 PHOTO 文件夹后，开始初始化 OV7670，在初始化成功之后，就一直在屏幕显示 OV7670 拍到的内容。当按下 KEY0 按键的时候，即进行拍照，此时 DS1 亮，拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功，同时 DS1 灭。DS0 还是用于指示程序运行状态。

所要用到的硬件资源如下：

1） 指示灯 DS0 和 DS1

2） KEY0 按键

3） 蜂鸣器

4） 串口

5） TFTLCD 模块

6） SD 卡

7） SPI FLASH

8） OV7670 摄像头模块

这几部分，在之前的实例中都介绍过了，我们在此就不介绍了。

44.3 软件设计

打开本章实验工程，由于本章要用到 OV7670、蜂鸣器、外部中断和定时器等外设，所以，先添加 ov7670.c、sccb.c、beep.c、exti.c 和 timer.c 等文件到 HARDWARE 组下。

然后，我们来看下 PICTURE 组下的 bmp.c 文件里面的 bmp 编码函数：bmp_encode，该函数代码如下：

```
//BMP 编码函数
//将当前 LCD 屏幕的指定区域截图,存为 16 位格式的 BMP 文件  RGB565 格式.
//保存为 rgb565 则需要掩码,需要利用原来的调色板位置增加掩码.这里我们增加了掩码.
//保存为 rgb555 格式则需要颜色转换,耗时间比较久,所以保存为 565 是最快速的办法.
//filename:存放路径
//x,y:在屏幕上的起始坐标
//mode:模式.0,仅创建新文件;1,如果存在文件,则覆盖该文件.如果没有,则创建新的文件.
//返回值:0,成功;其他,错误码.
u8 bmp_encode(u8 *filename,u16 x,u16 y,u16 width,u16 height,u8 mode)
{
    FIL* f_bmp; u8 res=0;
    u16 bmpheadsize;            //bmp 头大小
    BITMAPINFO hbmp;            //bmp 头
    u16 tx,ty;                  //图像尺寸
    u16 *databuf;               //数据缓存区地址
    u16 pixcnt;                 //像素计数器
    u16 bi4width;               //水平像素字节数
    if(width==0||height==0)return PIC_WINDOW_ERR;           //区域错误
    if((x+width-1)>lcddev.width)return PIC_WINDOW_ERR;      //区域错误
    if((y+height-1)>lcddev.height)return PIC_WINDOW_ERR;    //区域错误
#if BMP_USE_MALLOC == 1     //使用 malloc
    databuf=(u16*)pic_memalloc(1024);
    //开辟至少 bi4width 大小的字节的内存区域 ,对 240 宽的屏,480 个字节就够了.
    if(databuf==NULL)return PIC_MEM_ERR;        //内存申请失败.
    f_bmp=(FIL *)pic_memalloc(sizeof(FIL));     //开辟 FIL 字节的内存区域
    if(f_bmp==NULL){pic_memfree(databuf); return PIC_MEM_ERR; }//内存申请失败.
#else
    databuf=(u16*)bmpreadbuf;
    f_bmp=&f_bfile;
#endif
    bmpheadsize=sizeof(hbmp);                   //得到 bmp 文件头的大小
    mymemset((u8*)&hbmp,0,sizeof(hbmp));        // 申请到的内存置零.
    hbmp.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);//信息头大小
    hbmp.bmiHeader.biWidth=width;               //bmp 的宽度
    hbmp.bmiHeader.biHeight=height;             //bmp 的高度
    hbmp.bmiHeader.biPlanes=1;                  //恒为 1
    hbmp.bmiHeader.biBitCount=16;               //bmp 为 16 位色 bmp
    hbmp.bmiHeader.biCompression=BI_BITFIELDS;//每个象素的比特由指定的掩码决定。
```

```
        hbmp.bmiHeader.biSizeImage=hbmp.bmiHeader.biHeight*hbmp.bmiHeader.biWidth*
                                    hbmp.bmiHeader.biBitCount/8;//bmp 数据区大小
        hbmp.bmfHeader.bfType=((u16)'M'<<8)+'B';      //BM 格式标志
        hbmp.bmfHeader.bfSize=bmpheadsize+hbmp.bmiHeader.biSizeImage;//整个 bmp 的大
小
        hbmp.bmfHeader.bfOffBits=bmpheadsize;         //到数据区的偏移
        hbmp.RGB_MASK[0]=0X00F800;                    //红色掩码
        hbmp.RGB_MASK[1]=0X0007E0;                    //绿色掩码
        hbmp.RGB_MASK[2]=0X00001F;                    //蓝色掩码
        if(mode==1)res=f_open(f_bmp,(const TCHAR*)filename,FA_READ|FA_WRITE);
        //尝试打开之前的文件
        if(mode==0||res==0x04)res=f_open(f_bmp,(const TCHAR*)filename,FA_WRITE|
        FA_CREATE_NEW);//模式 0,或者尝试打开失败,则创建新文件
        if((hbmp.bmiHeader.biWidth*2)%4)//水平像素(字节)不为 4 的倍数
        {
            bi4width=((hbmp.bmiHeader.biWidth*2)/4+1)*4;//实际像素,必须为 4 的倍数.

        }else bi4width=hbmp.bmiHeader.biWidth*2;            //刚好为 4 的倍数
        if(res==FR_OK)//创建成功
        {
            res=f_write(f_bmp,(u8*)&hbmp,bmpheadsize,&bw);//写入 BMP 首部
            for(ty=y+height-1;hbmp.bmiHeader.biHeight;ty--)
            {
                pixcnt=0;
                for(tx=x;pixcnt!=(bi4width/2);)
                {
                    if(pixcnt<hbmp.bmiHeader.biWidth)databuf[pixcnt]=LCD_ReadPoint(tx,
ty);
                    //读取坐标点的值
                    else databuf[pixcnt]=0Xffff;//补充白色的像素.
                    pixcnt++; tx++;
                }
                hbmp.bmiHeader.biHeight--;
                res=f_write(f_bmp,(u8*)databuf,bi4width,&bw);//写入数据
            }
            f_close(f_bmp);
        }
    #if BMP_USE_MALLOC == 1    //使用 malloc
        pic_memfree(databuf); pic_memfree(f_bmp);
    #endif
        return res;
    }
```

该函数实现了对 LCD 屏幕的任意指定区域进行截屏保存，用到的方法就是 44.1 节我们所介绍的方法，该函数实现了将 LCD 任意指定区域的内容，保存个为 16 位 BMP 格式，存

放在指定位置（由 filename 决定）。注意，代码中的 BMP_USE_MALLOC 是在 bmp.h 定义的一个宏，用于设置是否使用 malloc，本章我们选择使用 malloc。

头文件 bmp.h 主要是一些函数申明，这里我们不做过多讲解。

最后我们来看看 main.c 源文件内容，代码如下：

```
extern u8 ov_sta;   //在 exit.c 里面定义
extern u8 ov_frame;      //在 timer.c 里面定义
//更新 LCD 显示
void camera_refresh(void)
{
    ……//省略代码，此部分代码同第 40 章一模一样。
}
//文件名自增（避免覆盖）
//组合成:形如"0:PHOTO/PIC13141.bmp"的文件名
void camera_new_pathname(u8 *pname)
{
    u8 res;
    u16 index=0;
    while(index<0XFFFF)
    {
        sprintf((char*)pname,"0:PHOTO/PIC%05d.bmp",index);
        res=f_open(ftemp,(const TCHAR*)pname,FA_READ);//尝试打开这个文件
        if(res==FR_NO_FILE)break;       //该文件名不存在=正是我们需要的.
        index++;
    }
}
int main(void)
{
    u8 res; u8 i;
    u8 *pname;                  //带路径的文件名
    u8 key;                     //键值
    u8 sd_ok=1;                 //0,sd 卡不正常;1,SD 卡正常.
    delay_init();               //延时函数初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置中断优先级分组为组 2
    uart_init(115200);          //串口初始化为 115200
    usmart_dev.init(72);        //初始化 USMART
    LED_Init();                 //初始化与 LED 连接的硬件接口
    KEY_Init();                 //初始化按键
    LCD_Init();                 //初始化 LCD
    BEEP_Init();                //蜂鸣器初始化
    W25QXX_Init();              //初始化 W25Q128
    my_mem_init(SRAMIN);        //初始化内部内存池
    exfuns_init();              //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);      //挂载 SD 卡
    f_mount(fs[1],"1:",1);      //挂载 FLASH.
```

```
	POINT_COLOR=RED;
	while(font_init())		//检查字库
	{
		LCD_ShowString(30,50,200,16,16,"Font Error!"); delay_ms(200);
		LCD_Fill(30,50,240,66,WHITE);//清除显示
	}
	Show_Str(30,50,200,16,"精英 STM32F1 开发板",16,0);
	Show_Str(30,70,200,16,"照相机实验",16,0);
	Show_Str(30,90,200,16,"KEY0:拍照",16,0);
	Show_Str(30,110,200,16,"正点原子@ALIENTEK",16,0);
	Show_Str(30,130,200,16,"2015 年 1 月 20 日",16,0);
	res=f_mkdir("0:/PHOTO");		//创建 PHOTO 文件夹
	if(res!=FR_EXIST&&res!=FR_OK)		//发生了错误
	{
		Show_Str(30,150,240,16,"SD 卡错误!",16,0); delay_ms(200);
		Show_Str(30,170,240,16,"拍照功能将不可用!",16,0);
		sd_ok=0;
	}else
	{
		Show_Str(30,150,240,16,"SD 卡正常!",16,0); delay_ms(200);
		Show_Str(30,170,240,16,"KEY0:拍照",16,0);
		sd_ok=1;
	}
	pname=mymalloc(SRAMIN,30);		//为带路径的文件名分配 30 个字节的内
存
	while(pname==NULL)			//内存分配出错
	{
		Show_Str(30,190,240,16,"内存分配失败!",16,0); delay_ms(200);
		LCD_Fill(30,190,240,146,WHITE); delay_ms(200);  //清除显示
	}
	while(OV7670_Init())//初始化 OV7670
	{
		Show_Str(30,190,240,16,"OV7670 错误!",16,0); delay_ms(200);
		LCD_Fill(30,190,239,206,WHITE); delay_ms(200);
	}
	Show_Str(30,190,200,16,"OV7670 正常",16,0);
	delay_ms(1500);
	TIM6_Int_Init(10000,7199);		//10Khz 计数频率,1 秒钟中断
	EXTI8_Init();				//使能定时器捕获
	OV7670_Window_Set(12,176,240,320);	//设置窗口
	OV7670_CS=0;
	LCD_Clear(BLACK);
	while(1)
	{
```

```
            key=KEY_Scan(0);//不支持连按
            if(key==KEY0_PRES)
            {
                if(sd_ok)
                {
                    LED1=0;                         //点亮 DS1,提示正在拍照
                    camera_new_pathname(pname);   //得到文件名
                    if(bmp_encode(pname,(lcddev.width-240)/2,(lcddev.height-320)/2,
                                  240,320,0))       //拍照有误
                    {
                        Show_Str(40,130,240,12,"写入文件错误!",12,0);
                    }else
                    {
                        Show_Str(40,130,240,12,"拍照成功!",12,0);
                        Show_Str(40,150,240,12,"保存为:",12,0);
                        Show_Str(40+42,150,240,12,pname,12,0);
                        BEEP=1; //蜂鸣器短叫，提示拍照完成
                        delay_ms(100);
                    }
                }else           //提示 SD 卡错误
                {
                    Show_Str(40,130,240,12,"SD 卡错误!",12,0);
                    Show_Str(40,150,240,12,"拍照功能不可用!",12,0);
                }
                BEEP=0;     //关闭蜂鸣器
                LED1=1;     //关闭 DS1
                delay_ms(1800);//等待 1.8 秒钟
                LCD_Clear(BLACK);
            }else delay_ms(10);
            camera_refresh();   //更新显示
            i++;
            if(i==20)           //DS0 闪烁.
            {
                i=0;
                LED0=!LED0;
            }
        }
    }
```

此部分代码，和第四十章的代码有点类似，只是这里我们多了一个 camera_new_pathname 函数，用于获取新的 bmp 文件名字（不覆盖旧的）。在 main 函数里面，我们通过 KEY0 按键控制拍照（调用 bmp_encode 函数实现），其他部分我们就不多介绍了，至此照相机实验代码编写完成。

最后，本实验可以通过 USMART 来测试 BMP 编码函数，将 bmp_encode 函数添加到 USMART 管理，即可通过串口自行控制拍照，方便测试。

44.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F103 上，得到如图 44.4.1 所示界面：

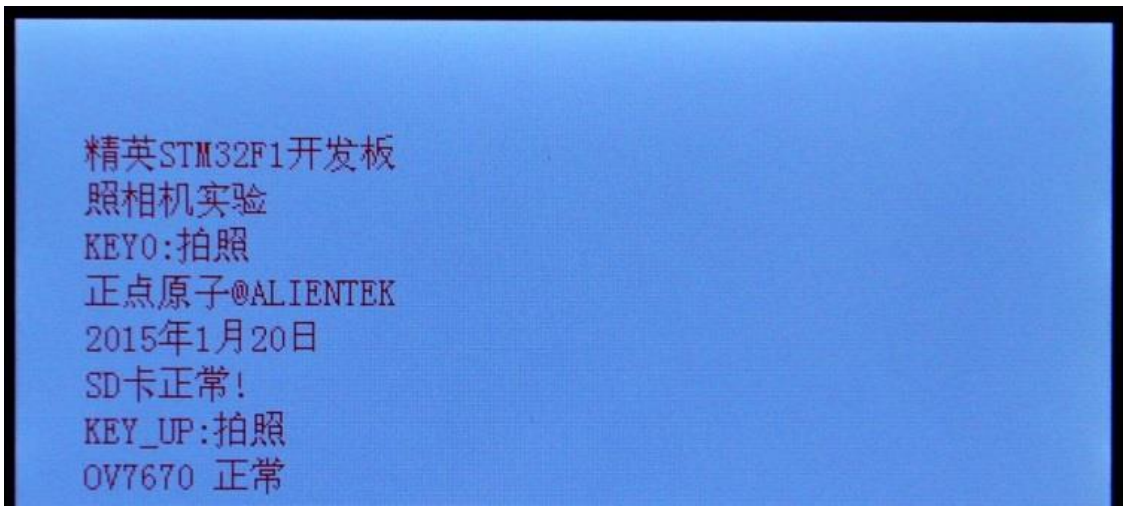

图 44.4.1 程序运行效果图

随后，进入监控界面。此时，我们可以按下 KEY0 即可进行拍照。拍照得到的照片效果如图 47.4.2 所示：

图 44.4.2 拍照样图（bmp 拍照样图）

最后，我们还可以通过 USMART 调用 bmp_encode 函数，实现串口控制拍照，还可以拍成各种尺寸哦（不过必须小于 240*320）!