1 基于 LWIP TCP 的 OV5640 传图实验

章节导读

本章实验将 OV5640 采集到的图像数据通过 TCP 传输给电脑端，在电脑端通过上位机软件显示图像数据，这里我们主要讲解的是如何实现上位机和下位机之间的通信，给读者提供整个设计思路，使读者可以基于我们提供的程序进行修改，使其实现自己想要的功能。

1.1 整体设计思路

在进行设计之前，我们首先需要明确需要实现的功能：

1. 初始化 OV5640 摄像头，为了方便修改寄存器中的值，交由 CPU 进行初始化。
2. DDR 存储 OV5640 采集到的图像数据。
3. LWIP TCP 传输存储在 DDR 的数据。
4. 上位机接收 TCP 传输过来的数据并显示。
5. 上位机修改分辨率之后，直接修改摄像头输出图像分辨率，最终通过上位机显示。
6. 上位机可以修改 OV5640 摄像头某个寄存器中的值。

整体的设计流程图如下所示：

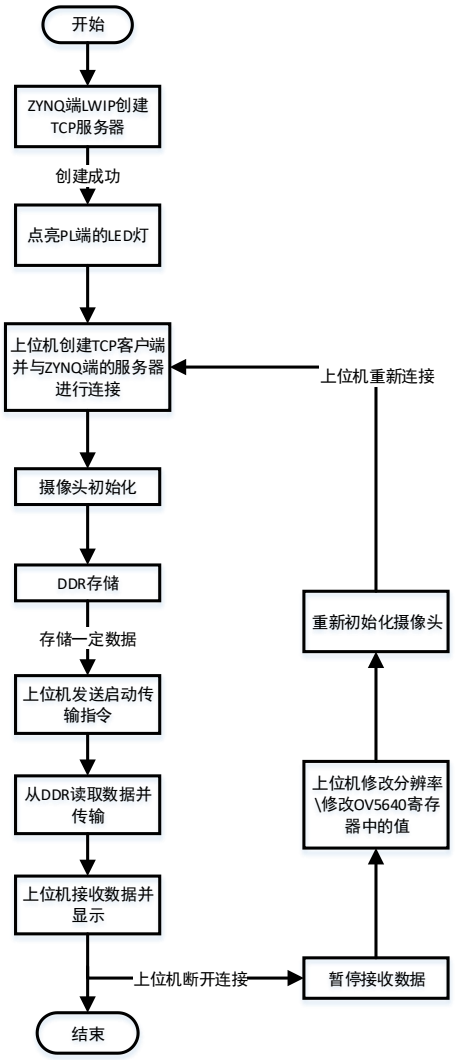


图 1-1 整体设计流程图

下面我们根据上述图对整个工程进行搭建，这里我们是基于 ACZ702 逻辑教程中的“ov5640_ddr3_tft_ic_hdmi”这个工程修改得到的，代码详细内容，请自行查看源代码，这里不在进行说明，主要对代码中比较重要的地方进行说明。

1.2 主要功能实现说明

1.2.1 摄像头初始化

我们这里为了方便修改摄像头寄存器的值，将摄像头初始化交由 CPU 处理，这里使能 I2C 0 进行摄像头初始化，如下所示。

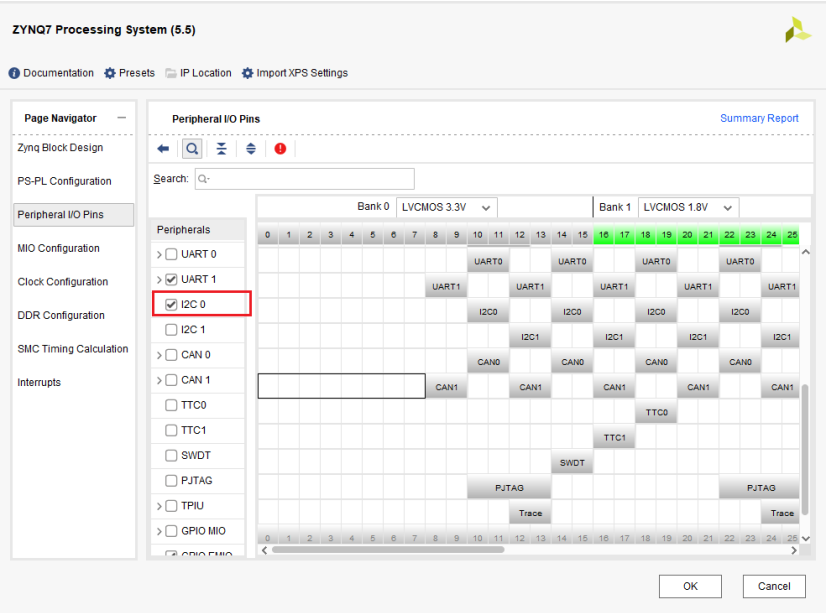


图 1-2 使能 I2C 进行摄像头初始化

我们使用之前的 OV5640 寄存器配置，分辨率最大只能到 1280*720，为了适配不同的分辨率，我们提供两种不同的初始化函数，如下所示，OV5640_p1080_Init()函数支持 1920*1080 的分辨。

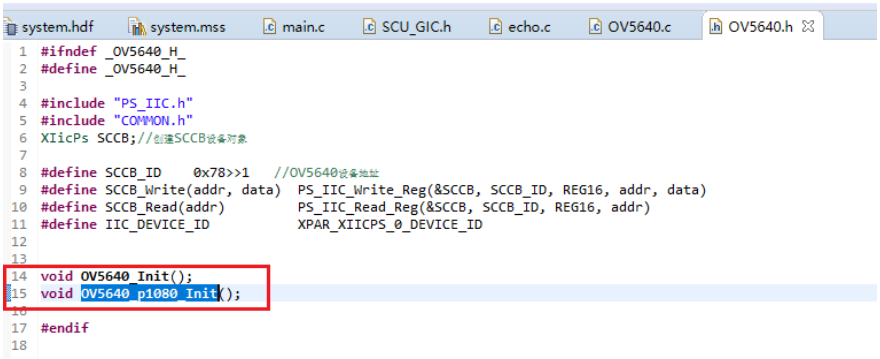


图 1-3OV5640 初始化函数

1.2.2 设置 DDR 存储起始地址

摄像头采集的数据需要保存到 DDR 中之后，再从 DDR 将数据读出并传输，按照我们之前的方式，是在 PL 端给出起始地址和结束地址，然后交由 PS 端进行读取，按照这种方式，每次修改分辨率之后，还需要手动在 PL 端修改起始地址，然后编译，导出等一系列操作。本次实验我们需要实现在不修改程序的情况下，直接通过上位机修改分辨率并显示，所以我们这里需要修改 fifo_to_axi4 模块的读写地址控制，并使能 ZYNQ 核的 M AXI GP0 接口，在 PS 接收到修改分辨率的指令之后，将相关参数传输至 PL 侧，交由 fifo_to_axi4 模块进行处理，

还需要注意的是，修改分辨率之后，还需要清除上一次的数据，防止影响下一次数据传输，主要修改地方如下所示：

1. 使能 ZYNQ 核的 M AXI GP0 端口

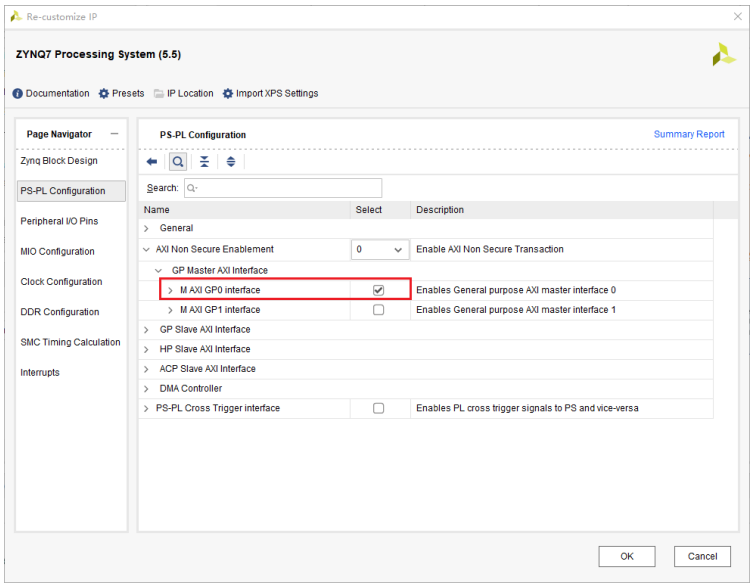


图 1-4 使能 ZYNQ 核的 M AXI GP0 端口

2. 设置相关寄存器

该功能的实现对应代码自定义的 IP 核 axi_lite_8reg，设置相关参数如下所示：

```
411     // Slave register read enable is asserted when valid address is available
412     // and the slave is ready to accept the read address.
413     assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
414     always @(*)
415     begin
416           // Address decoding for reading registers
417           case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
418             3'h0   : reg_data_out <= Trans_Length;//此处修改
419             3'h1   : reg_data_out <= slv_reg1;
420             3'h2   : reg_data_out <= slv_reg2;
421             3'h3   : reg_data_out <= slv_reg3;
422             3'h4   : reg_data_out <= slv_reg4;
423             3'h5   : reg_data_out <= slv_reg5;
424             3'h6   : reg_data_out <= slv_reg6;
425             3'h7   : reg_data_out <= slv_reg7;
426             default : reg_data_out <= 0;
427           endcase
428     end
449     // Add user logic here
450     assign  WR_DDR_Addr_Begin   =   slv_reg1;
451     assign  WR_DDR_Addr_End     =   slv_reg2;
452     assign  RD_DDR_Addr_Begin   =   slv_reg3;
453     assign  RD_DDR_Addr_End     =   slv_reg4;
454     assign  fifo_clear          =   slv_reg5;
455     // User logic ends
456
```

图 1-5 设置相关寄存器

对上述寄存器说明如下所示：

表 1-1 寄存器说明表

寄存器名称	寄存器地址	寄存器含义
WR_DDR_Addr_Begin	0x40000004	DDR 写数据的起始地址
WR_DDR_Addr_End	0x40000008	DDR 写数据的结束地址
RD_DDR_Addr_Begin	0x4000000C	DDR 读数据的起始地址
RD_DDR_Addr_End	0x40000010	DDR 读数据的结束地址
fifo_clear	0x40000014	清除信号，清除 FIFO 中存储的上一次的数据，复位相关模块
Trans_Length	0x40000000	当前写入的数据长度

上述表中的 DDR 读数据相关的寄存器在本次实验中，实际并不需要使用，这里是防止后续需要使用，Trans_Length 这个寄存器可以告诉 PS 端此时 DDR 中存储了多少数据，方便我们读取数据，寄存器的基地址，我们是在 Address Editor 中进行设置的。

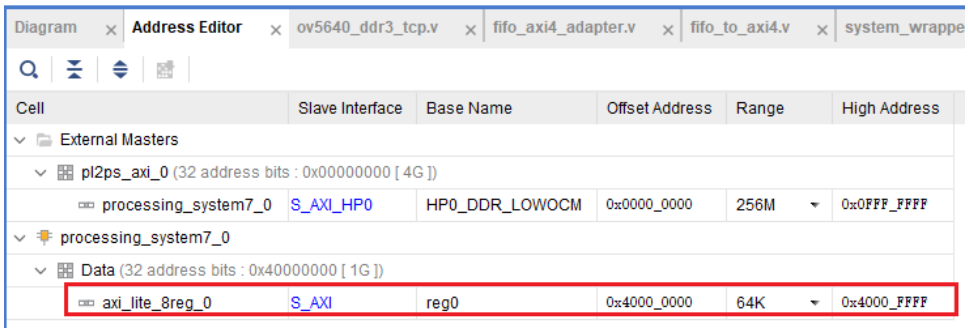


图 1-6 寄存器基地址设置

3. 修改 fifo_to_axi4 模块

PL 端获取到了 DDR 写数据的起始地址之后，将相关参数传输给 fifo_to_axi4 模块进行处理，首先是将起始地址从参数变化改成端口输入，如下所示：

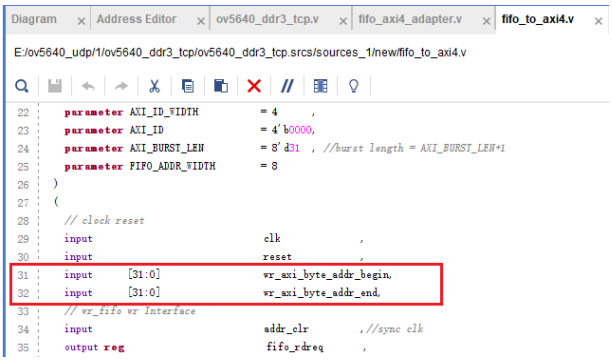


图 1-7 DDR 起始地址输入端口

然后是读写地址的控制，如下所示：

```
116   //m_axi_awaddr
117   always@(posedge clk or posedge reset)
118   begin
119     if(reset) begin
120       m_axi_awaddr <= wr_axi_byte_addr_begin;
121       wr_axi_byte_addr_begin_r <= wr_axi_byte_addr_begin;
122       wr_axi_byte_addr_end_r <= wr_axi_byte_addr_end;
123     end
124     else if(addr_clr || axi_awaddr_clr) begin
125       m_axi_awaddr <= wr_axi_byte_addr_begin;
126       wr_axi_byte_addr_begin_r <= wr_axi_byte_addr_begin;
127       wr_axi_byte_addr_end_r <= wr_axi_byte_addr_end;
128     end
129     else if(m_axi_awaddr >= wr_axi_byte_addr_end_r) begin
130       m_axi_awaddr <= wr_axi_byte_addr_begin;
131       wr_axi_byte_addr_begin_r <= wr_axi_byte_addr_begin;
132      wr_axi_byte_addr_end_r <= wr_axi_byte_addr_end;
133     end
134     else if((curr_wr_state == S_WR_RESP) && m_axi_bready && m_axi_bvalid && (m_axi_bresp == 2'b00) && (m_axi_bid == AXI_ID[AXI_ID_WIDTH-1:0]))
135       m_axi_awaddr <= m_axi_awaddr + ((m_axi_awlen + 1'b1)*(AXI_DATA_WIDTH/8));
136     else
137       m_axi_awaddr <= m_axi_awaddr;
138   end
139
```

图 1-8 读写地址控制

然后是将实时传输数据长度进行输出，如下所示：

```
288
289     always@(posedge clk or posedge reset)
290     begin
291       if(reset)
292         Trans_Length <= 0;
293       else
294         Trans_Length <= m_axi_awaddr - wr_axi_byte_addr_begin_r;
295     end
```

图 1-9 实时传输数据长度

1.2.3 相关指令的定义

在前面我们说过我们需要通过上位机发送启动传输指令、修改分辨率指令、修改寄存器指令，首先我们就需要确定每个功能对应的指令是什么，这样我们下位机在处理的时候，才能根据不同的指令去执行不同的功能，本次实验我们定义指令如下所示：

表 1-2 指令说明表

	byte0	byte1	byte2	byte3	byte4	byte5	byte6	byte7
启动传输	55	A5	00	00	00	00	00	F0
改分辨率	55	A5	01	image_weight [15:8]	image_weight [7:0]	image_height [15:8]	image_height [7:0]	F0
改寄存器	55	A5	02	00	camera_addr [15:8]	camera_addr [7:0]	camera_data [7:0]	F0

后续我们上位机需要按照上表中的内容构建指令发送至下位机，然后下位机根据指令解析相关参数并实现对应功能。

1.2.4 上位机实现流程

上位机我们通过 MFC 来实现，这里我们不会讲解如何编写上位机，想要了解的读者请自行学习相关的内容。我们这里对上位机基本实现流程进行简要说明，特别需要注意的是指令发送的先后顺序，基本流程图如下所示：

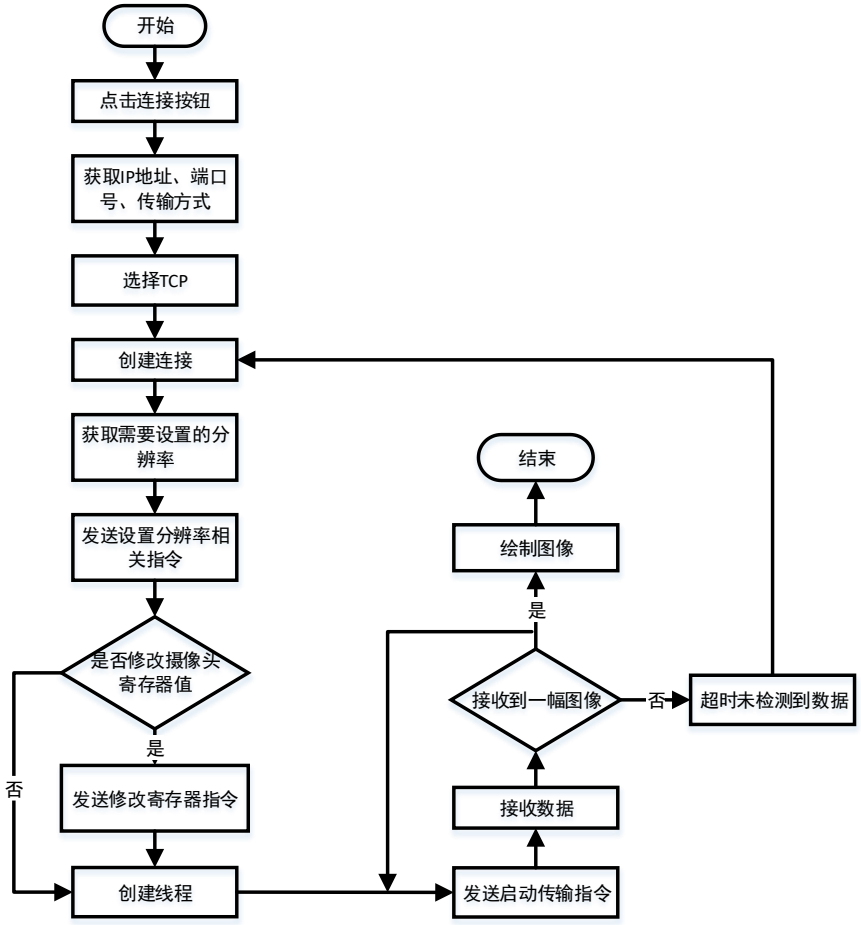


图 1-10 上位机实现流程图

根据上述所示流程图设计上位机时，需要注意以下几点：

1. 指令发送的先后顺序，首先启动传输指令一定要在最后，因为我们需要到设置的图像分辨率之后，才会获取到 DDR 的写的起始地址，才会向 DDR 中写入数据，此时才能开始传输数据。设置分辨率的指令需要在修改寄存器指令之前，因为我们需要根据不同的分辨率对 OV5640 进行初始化，如果修改寄存器指令在前面的话，发出分辨率指令之后，又对 OV5640 进行初始化，寄存器等于没改，当然读者可以对比两个寄存器表，只对某些对应的寄存器进行修改，我们这里为了方便，直接进行不同的初始化。

2. 当接收到一幅图像之后，绘制图像，并发送启动传输指令，进行下一幅图像的传输，如果长时间接收不到下一幅图像，那么我们将重现连接，发送相关指令进行数据传输，这样就防止我们传输过程完全卡死不动。

1.2.5 下位机代码实现

打开 SDK 之后，建立 APP 工程，以 LWIP TCP Perf Server 为模板建立工程，在此模板上进行修改将会方便很多。

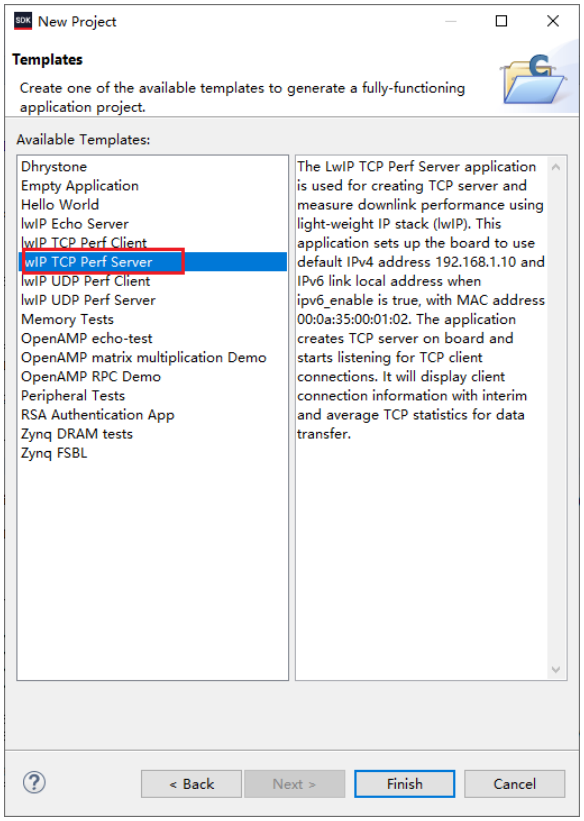


图 1-11 建立 LWIP TCP Perf Server 工程

我们的下位机需要实现的功能就是创建 TCP 服务器，接收上位机传输过来的相关配置指令，通过解析相关指令实现对应功能，基本实现流程图如下所示。

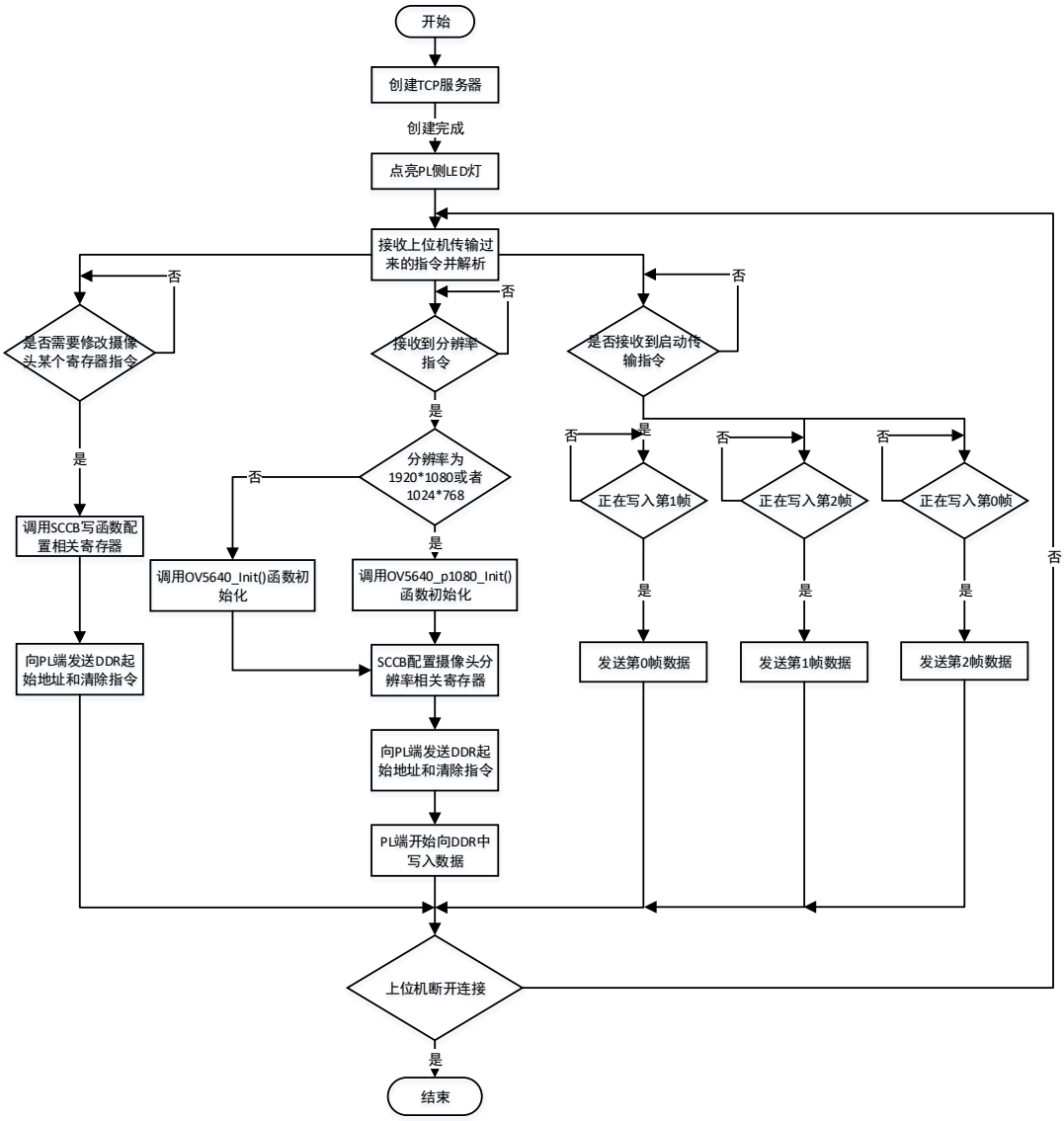


图 1-12 下位机基本实现流程图

通过上述图中的内容，我们便可以编写下位机代码，下面我们将对实现代码中的部分内容进行讲解。

1.2.5.1 TCP 服务器的创建

建立的工程模板中已经将建立 TCP 服务器的方式进行了说明，这里不对此部分进行详细说明，我们需要注意就是服务器的 IP 地址和端口号的设置，因为我们的上位机需要根据 IP 和端口号去进行连接，本次实验服务器的 IP 地址为 192.168.0.3，端口号为 6102，MAC 地址为 00-0a-35-01-fe-c0，如下所示：

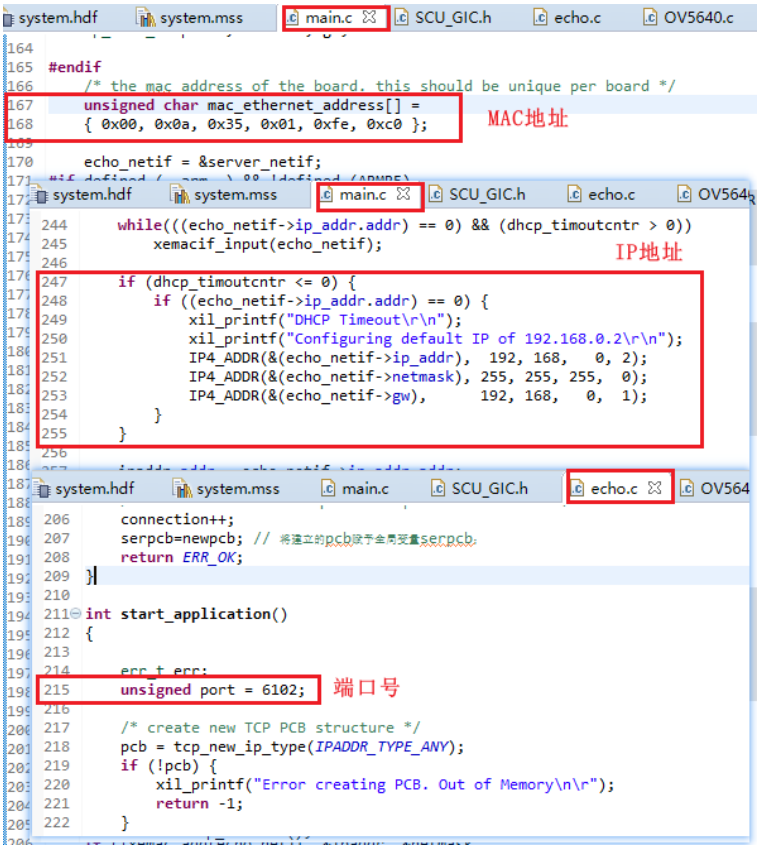


图 1-13 设置的服务器的 MAC 地址、IP 地址和端口号

需要注意的是，TCP 服务器的创建需要一定时间，程序下载完成之后，还需要等待一端时间，等待创建完成之后，上位机才能连接，进行数据传输，连接过程中，我们可以打开串口调试软件，将会看到如下所示的打印信息：

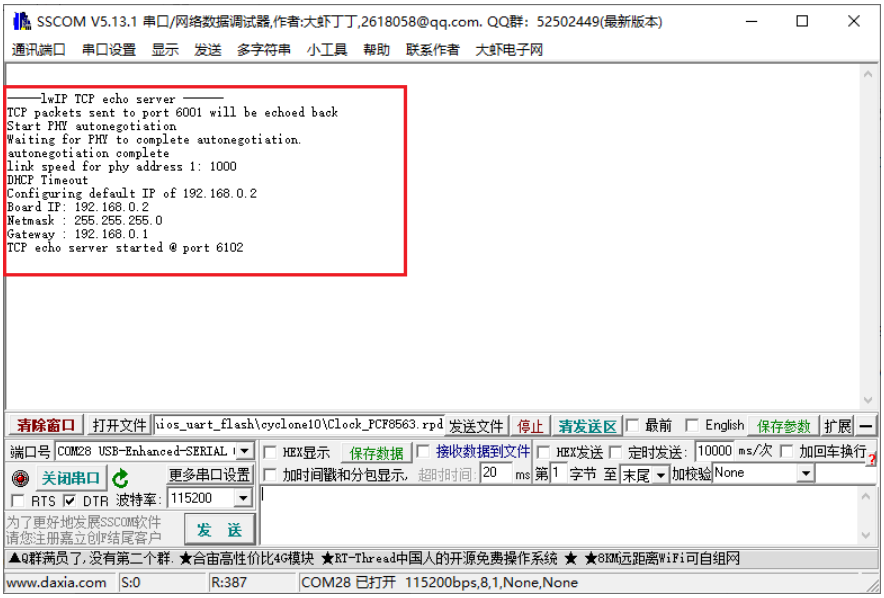


图 1-14 串口打印 TCP 服务器创建信息

当打印出“TCP echo server started @ port 6102”之后，说明 TCP 服务器创建完成，上位机此时才能进行连接，不然上位机连接之后将会卡住，连接不上服务器，这里为了方便用户观察，添加一个 LED 灯，当创建完成之后，点亮 PL 侧的 LED 灯，代码如下：

```
240
241     /* specify callback to use for incoming connections */
242     tcp_accept(pcb, accept_callback);
243
244     xil_printf("TCP echo server started @ port %d\n\r", port);
245     PS_GPIO_SetPort(PL_LED,1);
246     return 0;
247 }
248
```

图 1-15 服务器建立完成点亮 LED 灯

当程序下载完成之后，当我们看到 PL 侧 LED 灯被点亮之后，此时我们上位机就可以进行连接了，如下所示：

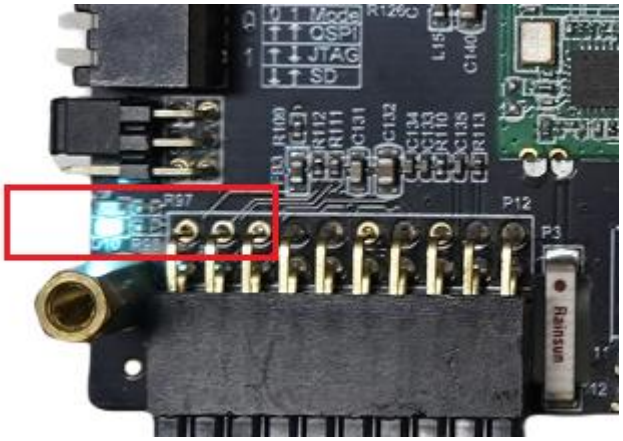


图 1-16 PL 侧的 LED 灯被点亮

1.2.5.2 解析指令

在 echo.c 文件中的 recv_callback 函数实现对命令的解析，通过 tcp_recved 接收接收数据，第一步需要判断帧头帧尾是否正确，关于指令的定义可以查看“相关指令的定义”一节中的内容，如下所示：

图 1-17 判断帧头帧尾

根据上位机发送指令的先后顺序，首先应该接收到分辨率的指令，并给出对应的标志信号，解析出对应的分辨率，根据不同的分辨率初始化，初始化完成之后，还需要根据不同的分辨率，向改变分辨率的寄存器中写入最新修改的分辨率，最后给出 DDR 存储的起始地址以及清除信号，交由 PL 端去控制写入 DDR 的数据，如下所示：

```
        }
        else if(recdata[2] == 0x01) //设置分辨率相关
        {
            change_image_size_flag = 1;
            image_weight = recdata[3] <<8 | recdata[4];
            image_height = recdata[5] <<8 | recdata[6];

            if((image_weight*image_height == 1920*1080) || (image_weight*image_height == 1024*768))
            {
                OV5640_p1080_Init();
            }
            else
            {
                OV5640_Init();
            }
            SCCB_Write(0x3808,recdata[3]); // DVPHO  (<1280>500) (<640>280)IMAGE_WIDTH
            SCCB_Write(0x3809,recdata[4]); // DVPHO
            SCCB_Write(0x380a,recdata[5]); // DVPVO  (<720>2d0)  (<480>1e0)IMAGE_HEIGHT
            SCCB_Write(0x380b,recdata[6]); // DVPHO

            Xil_Out32(WAB_REG_ADDR, FRAME_DATA_BASE_ADDR);  //起始地址
            Xil_Out32(WAE_REG_ADDR, FRAME_DATA_BASE_ADDR +  image_weight*image_height*2*3  - 1);//结束地址，3帧数据

            //需要更新DDR中的数据，给对应标志寄存器写相关的
            Xil_Out32(FIFO_CLEAR_ADDR, 0x1);    //起始地址
        }
```

图 1-18 解析出分辨率指令

上图所示的 DDR 起始地址、结束地址以及清除相关的地址，定义如下，这个是根据表 1-1 中的内容进行设置的。

```
volatile uint16_t image_height = 0;
#define FRAME_DATA_BASE_ADDR    0xF800000   //帧图像数据的起始地址


#define GTL_REG_ADDR            0x40000000  //获取传输长度寄存器的地址
#define WAB_REG_ADDR            0x40000004  //写地址起始寄存器的地址
#define WAE_REG_ADDR            0x40000008  //写地址结束寄存器的地址
#define RAB_REG_ADDR            0x4000000C  //读地址起始寄存器的地址
#define RAE_REG_ADDR            0x40000010  //读地址结束寄存器的地址
#define FIFO_CLEAR_ADDR         0x40000014  //清除DDR写FIFO中存储的数据

int data_buf[32768/4];
```

图 1-19 定义相关寄存器地址

我们可以看到，在给结束地址的时候，并不是在起始地址的基础上加上存储一幅图像所需要的空间，而是在此基础之上乘以三，这是因为在我们调试的过程中发现，当刚选择为一幅图像的存储空间时，移动摄像头的时候，上位机显示的图像会存在很明显的刷新分界线，也就是图像撕裂，看到一幅图像上下发生错位，出现的这种情况的原因是因为，当我们上位机还绘制一幅图像的时候，还未绘制完成，但是新的一帧图像已经接收到了，导致我们肉眼看到明显的断层现象，即我们看到的一张图片其实是两张图片组合而来，为了解决这一问题，我们加入“三帧缓存”的原理，也就是在 DDR 中设置三个缓存区，三重帧缓冲区组成了一个前缓冲区，两个后缓冲区的规格。程序来回向两个后缓冲区写入图像，每次传输数据显示时，前缓冲区就和最近完成写入的那个后缓冲区交换。

如果接收到修改寄存器指令之后，也就是 recdata[2]为 0x02，需要调用 SCCB_Write 函数根据接收到的寄存器地址和数据进行配置，并给出 DDR 存储的起始地址以及清除信号，为什么这个还需要再发送一遍，是因为改变寄存器指令之后，DDR 应该保存最新的寄存器配置输出的图像数据，代码如下所示：

```
136         else if(recdata[2] == 0x02)    //设置某个寄存器中的值
137         {
138             SCCB_Write(recdata[4] <<8 | recdata[5],recdata[6]); // DVPHO
139
140             Xil_Out32(WAB_REG_ADDR, FRAME_DATA_BASE_ADDR);  //起始地址
141             Xil_Out32(WAE_REG_ADDR, FRAME_DATA_BASE_ADDR +  image_weight*image_height*2*3  - 1);//结束地址，3帧数据
142
143             //需要更新DDR中的数据，给对应标志寄存器写相关的
144             Xil_Out32(FIFO_CLEAR_ADDR, 0x1);    //起始地址
145
146         }
147     }
```

图 1-20 修改寄存器指令

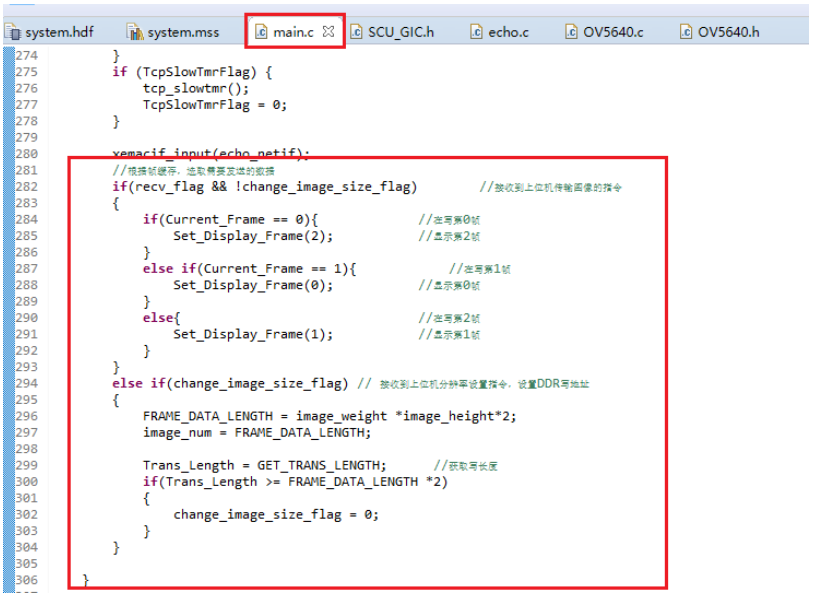
最后是接收到传输起始指令，接收到之后，将清除指令寄存器拉低，表明要开始传输数据，并拉高对应的标志信号。代码如下所示：

```
106         if(recdata[2] == 0x00)    //传输图像指令
107         {
108             Xil_Out32(FIFO_CLEAR_ADDR, 0x0);
109             recv_flag = 1;
110         }
```

图 1-21 启动传输指令

1.2.5.3 主函数实现

主函数中除了 LWIP 相关配置以外，还需要添加的就是根据修改分辨率标志信号以及启动传输标志信号，进行数据传输，主要代码如下所示：




```
274         }
275         if (TcpSlowTmrFlag) {
276             tcp_slowtmr();
277             TcpSlowTmrFlag = 0;
278         }
279
280         xemacif_input(echo_netif);
281         //读指针缓存，选取需要发送的数据
282         if(recv_flag && !change_image_size_flag)        //接收到上位机传输图像的指令
283         {
284             if(Current_Frame == 0){             //写写第0帧
285                 Set_Display_Frame(2);           //显示第2帧
286             }
287             else if(Current_Frame == 1){            //写写第1帧
288                 Set_Display_Frame(0);           //显示第0帧
289             }
290             else{                               //写写第2帧
291                 Set_Display_Frame(1);           //显示第1帧
292             }
293         }
294         else if(change_image_size_flag) // 接收到上位机分辨率设置指令，设置DDR写地址
295         {
296             FRAME_DATA_LENGTH = image_weight *image_height*2;
297             image_num = FRAME_DATA_LENGTH;
298
299             Trans_Length = GET_TRANS_LENGTH;      //获取写长度
300             if(Trans_Length >= FRAME_DATA_LENGTH *2)
301             {
302                 change_image_size_flag = 0;
303             }
304         }
305
306     }
```

图 1-22 主函数部分实现

从上图所示的代码可以看到，体现了三帧缓存的功能，再写第 0 帧的时候，显示第 2 帧，写第 1 帧的时候，显示第 0 帧，写第 2 帧的时候，显示第 1 帧，这样就减少了图像出现撕裂的情况，还需要注意的一点就是，在修改分辨率指令之后，我们此时需要写入一部分数据之后，才进行图像图像，防止还未向 DDR 写入数据，就开始传输，导致传输的数据不是我们想要的，这里我们是写入 2 幅图像之后再拉低标志信号，此时才允许进行数据传输，获取传输长度的函数为 GET_TRANS_LENGTH，实现方式如下：

```
65
66  //读取当前已传输的数据长度
67  #define GET_TRANS_LENGTH        Xil_In32(GTL_REG_ADDR);
68
```

图 1-23 获取当前已传输的数据长度

1.2.5.4 传输数据函数实现

主函数中的 Set_Display_Frame 函数的功能就是将 DDR 中存储的数据通过 TCP 协议传输给上位机，代码如下所示：

```
int Set_Display_Frame(uint8_t Frame_Num)
{
    u32 buff;
    pbuf_free(serpcb->refused_data);
    buff = 1446;    //TCP一次最长传输的包的大小
    if(buff > image_num){
        tcp_write(serpcb,(void*)(FRAME_DATA_BASE_ADDR+Frame_Num*FRAME_DATA_LENGTH + uiIdx),image_num,1);
        tcp_output(serpcb);
        recv_flag = 0;
        uiIdx = 0;
        image_num = FRAME_DATA_LENGTH;
        send_frame_finish = 1;
        pbuf_free(serpcb->refused_data);
        //获取当前传输长度
        Trans_Length = GET_TRANS_LENGTH;
        //计算当前传输帧
        Current_Frame = Trans_Length / FRAME_DATA_LENGTH;

    }
    else{
        tcp_write(serpcb,(void*)(FRAME_DATA_BASE_ADDR+Frame_Num*FRAME_DATA_LENGTH + uiIdx),buff,1);
        tcp_output(serpcb);
        image_num = image_num - buff;
        uiIdx += buff;
        send_frame_finish = 0;
        pbuf_free(serpcb->refused_data);
    }

    return send_frame_finish;
}
```

图 1-24Set_Display_Frame 函数的实现方式

从上图可以看出，每次传输的图像数据为 1446，这是因为 TCP 最大的一包数据长度就是 1446，这样每次传输按照最长包进行传输，电脑不用开启巨型帧，也能接收数据，按照之前的方式，每包以一行数据传输的话，增大分辨率之后，电脑还需要开始巨型帧才能接收到数据，以我们本次实验的方式进行数据传输，不管分辨率是多大，我们电脑都不需要再开始巨型帧进行数据接收，方便电脑没有巨型帧功能的用户使用。

还需要注意的是，再每次一幅图像传输完成之后，还需要获取此时写入到了哪帧数据，以此来判别我们下一帧需要传输哪一帧图像，并且需要拉低对应的启动传输标志信号，等待下一次传输指令的到来，防止图像显示错位。

1.3 板级验证

1.3.1 系统所需硬件

1. ACZ702 开发板 x1
2. OV5640 摄像头 x1
3. Type-C 线 x1
4. 网线 x1

1.3.2 硬件连接

将摄像头、网线和 Type-C 线依次连接到 ACZ702 开发板上，连接方式如下

所示：

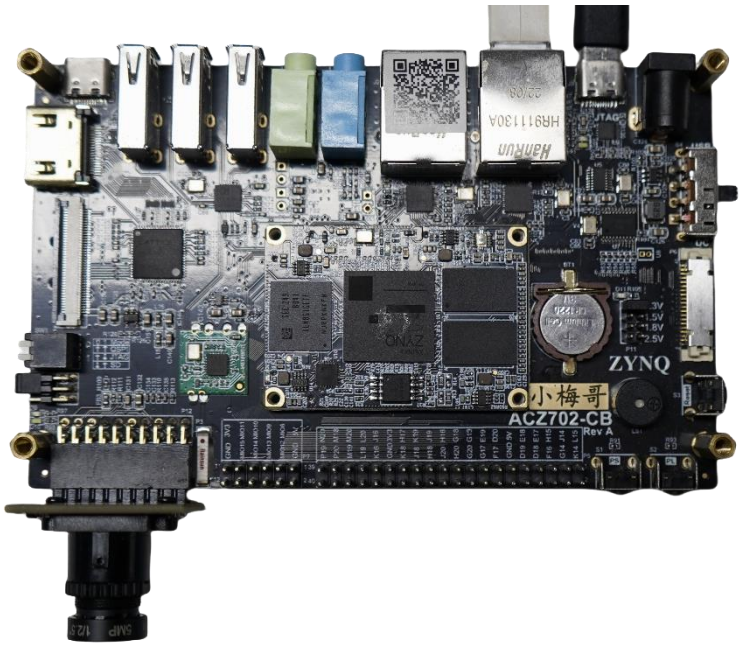


图 1-25 硬件连接

连接完成之后，给开发板上电，下载程序。

1.3.3 功能测试

1. 程序下载完成之后，等待 PL 侧 LED 灯被点亮，如下所示。

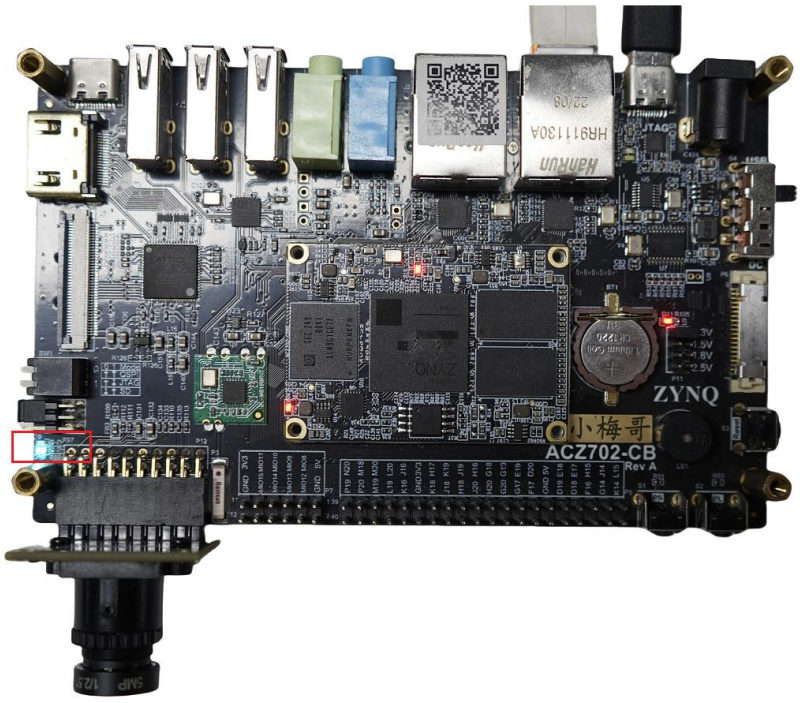


图 1-26 PL 侧 LED 灯被点亮

2. 打开上位机软件，传输方式选择为 TCP，如下所示。

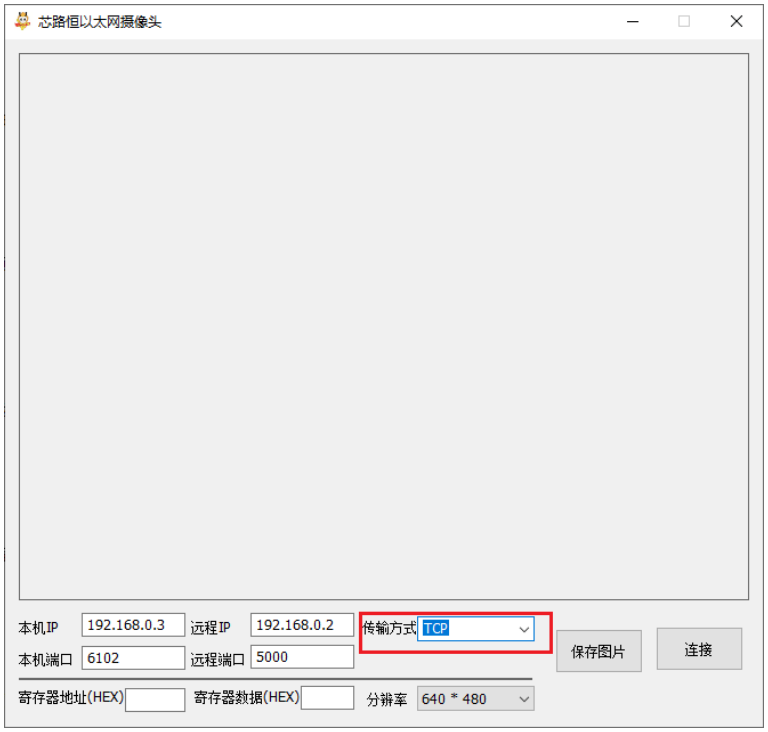


图 1-27 选择传输方式

3. 点击连接，此时我们便可以看到上位机显示了摄像头采集到的图像数据，如下所示。

图 1-28 上位机显示图像

4. 点击停止按钮之后，我们可以修改分辨率，这里以 800*480、1280*720 显示为例，上位机显示图像如下所示。

图 1-29 修改分辨率为 800*480

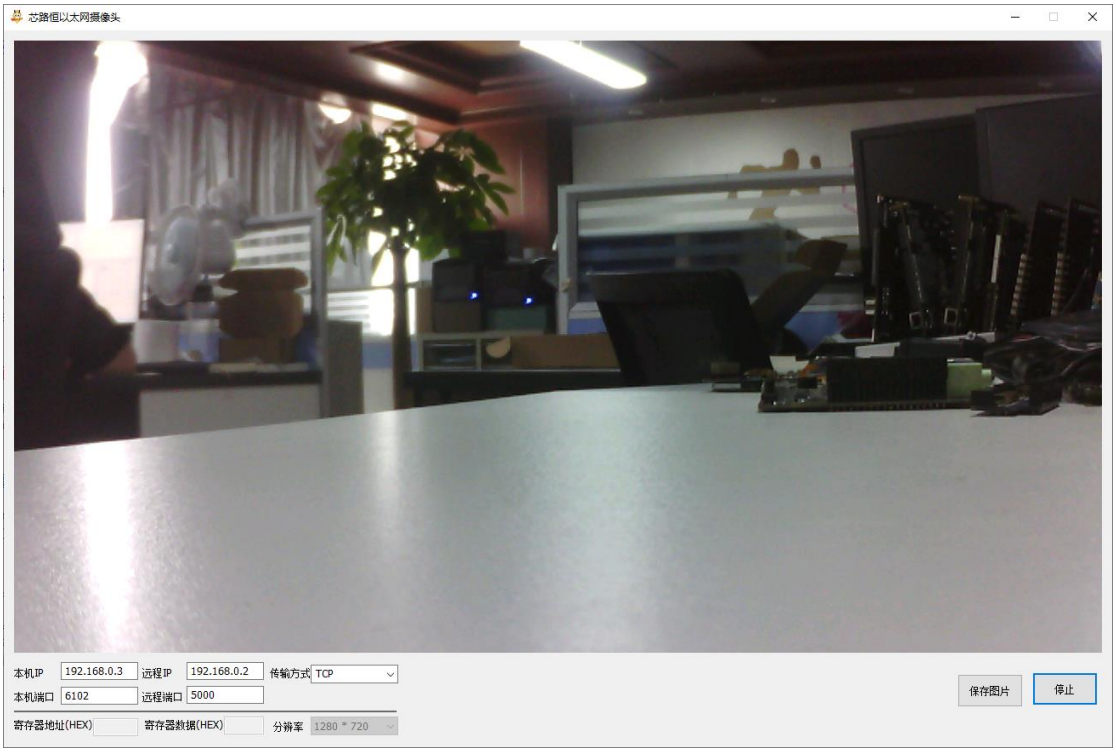


图 1-30 修改分辨率为 1280*720

5. 修改摄像头某个寄存器中的值，修改之前首先停止传输，然后给出寄存器地址和数据，为了我们方便观察，我们这里使用测试寄存器来进行测试观察，上位机配置界面如下所示，这里给寄存器地址和数据时都是 HEX 格式的数据。

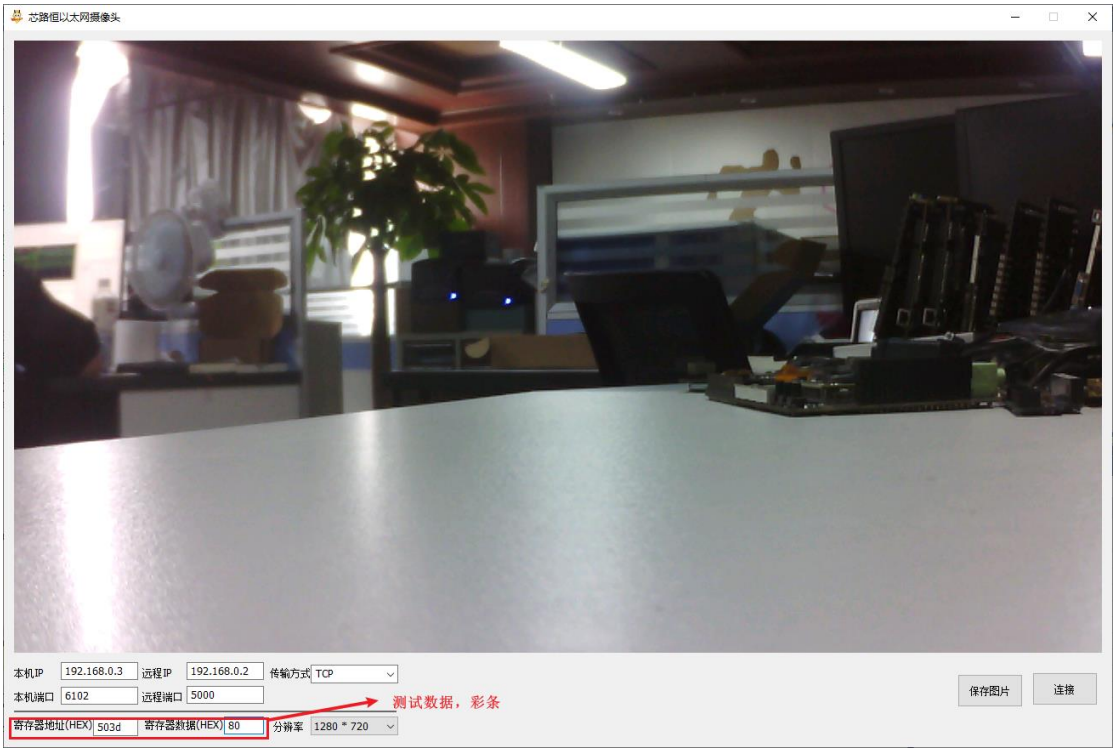


图 1-31 摄像头测试数据寄存器设置

然后点击连接，这时我们可以看到上位机显示彩条，如下所示，说明通过上位机修改摄像头寄存器值的功能测试 成功。

图 1-32 修改寄存器测试显示

1.4 总结

本章实验我们实现了通过上位机显示 OV5640 采集到的摄像头数据，并且可以通过上位机修改分辨率以及重新设置摄像头某个寄存器的值，对于代码的讲解比较少，主要是提供一个设计思路，读者可以根据自己的需求进行功能的优化等操作。