基于 AXI DMA 的双通道数据采集 TCP 传输系统

AD9238 ADC

3 章节导读

作为一款双通道的高速 ADC，ACM9238 能够同时进行双通道的模拟数据采集和转换。本节内容便是介绍基于 ACM9238 的双通道 ADC 采集 DMA 搬运 TCP 传输系统。

用户可以通过以太网上位机下发指令，控制 ADC 的采样频率、采样个数、采样通道。采集到的数据会通过 DMA 搬运到 PS 端的 DDR3 中，并在采样结束后，通过 PS 端网口，使用 TCP 协议发送到电脑端，用户可以使用小梅哥上位机对波形进行绘制，也可以使用 matlab 等软件对数据做进一步处理。

3.1 ACM9238 模块简介

ACM9238 模块如下图 3-1 所示，可用于小梅哥全系列 FPGA、SOC、Zynq 开发板。模块上使用的是 ADI 公司的 AD9238BSTZ 芯片，该 ADC 芯片是一款高性能的、低功耗的双通道 12 位模数转换器，能够支持最高 65Msps 的采样率。配合上 ACM9238 模块的前端模拟信号调理电路，实现了±5V 电压范围内信号的高速采样。

模块使用 2 路完全相同的 AD 采样和信号调理电路，构成了双通道高速 AD 采样电路。两路 ADC 电路完全独立，结构和元器件参数相同，确保了两个通道有较高的一致性。模块与 FPGA 的连接采用并行接口，每路 ADC 包括 12 位数据信号（ADC_DATA），1 位时钟信号（ADC_CLK），1 位超量程指示信号（ADC_OTR），该模块接口图如下图 3-2 所示。

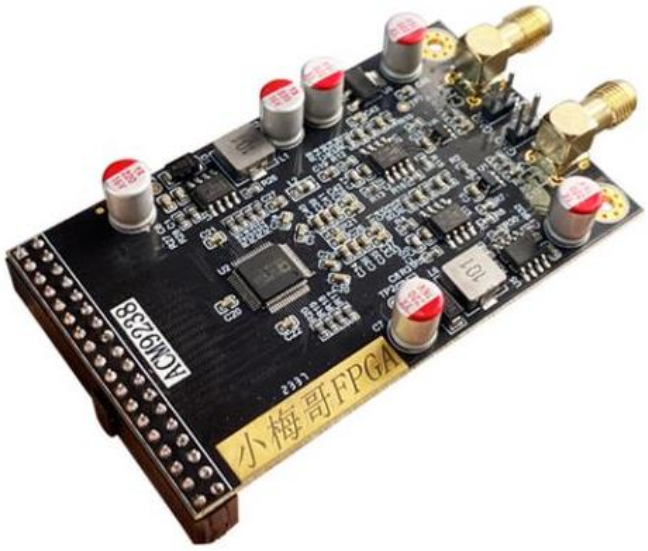

图 3-1 ACM9238 模块图

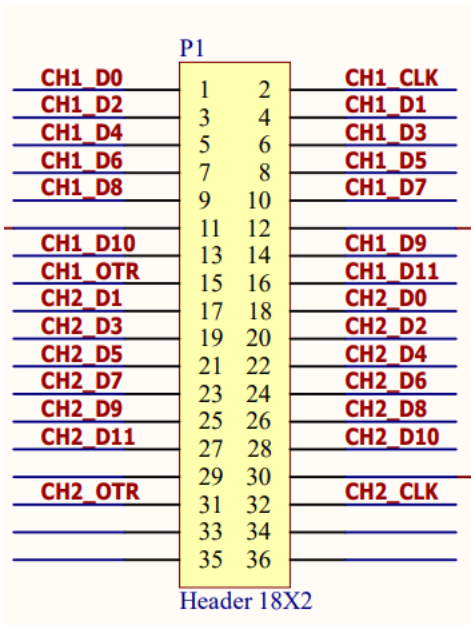

图 3-2 ACM9238 模块接口图

使用该模块时， FPGA 仅需为每路 ADC 提供一路时钟信号，ADC 便会在每个时钟周期输出一个 12 位的采样结果。当 9238BSTZ 模拟输入端接-5V 至+5V 之间变化的正弦波电压信号时，其转换后的数据也是成正弦波波形变化，转换波形如下图 3-3 所示，从图中可以看出 9238BSTZ 采集到的数据是无符号数据。

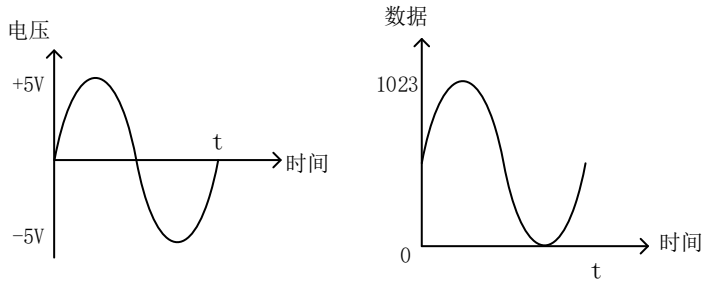

图 3-3 9238BSTZ 正弦波模拟电压值（左）、数据（右）

虽然 9238BSTZ 芯片的采样率上限为 65Msps，但实际使用的时候，模块最高只能实现 60Msps 的采样率。在本次设计中，我们将让 ADC 工作在 50Msps 的采样率下。由于 9238BSTZ 的采样率等于 FPGA 提供给 ADC 的时钟频率，因此，当需要使用低于 50Msps 的采样率时，可以依旧给 ADC 提供 50MHz 的时钟信号。然后，在 FPGA 内部，对 50Msps 的采样结果数据进行抽取重采样。比如期望以 1Msps 的采样速率采样，则只需要每间隔 50 个采样数据取一个结果存储或使用，其他 49 个数据直接舍弃，这样就能实现 1MSPS 的采样率了。

十分不建议通过大幅降低 ADC 工作时钟频率的方法来降低采样率，因为时钟太低，会影响 ADC 芯片内部采样保持电路的工作情况，导致采样误差偏大。

3.2 实验介绍

本节实验将会通过 ACM9238 模块，实现双通道 ADC 数据采集。用户可以通过以太网上位机，使用 TCP 协议下传指令，控制 ADC 的采样数量、采样频率、采样通道。ADC 采集到的数据会通过 AXI DMA 核搬运到 PS 端的 DDR3 中存储，在采集结束后，DDR3 中的数据会被读出，通过 PS 端网口发送给电脑端，用户可以通过上位机对数据做进一步处理。

设计整体会由 PL 与 PS 两部分构成，PL 端负责搭建硬件逻辑系统，实现对 ACM9238 模块的驱动控制、采样数据重采样以及搭建 DMA 通路。PS 端负责通过 C 代码，解包用户下发的 TCP 指令交由 PL 端去驱动 ACM9238 模块；在 AXI DMA 中断触发时，驱动 AXI DMA 完成数据搬运；采样结束后，从 DDR 中读出数据，通过 TCP 协议传输给 PC 端。

为了确保指令的准确性，指令需要按照下表 3-1 所示的帧格式发送：

表 3-1 指令帧格式

帧头		寄存器	数据				帧尾
0x55	0xA5	reg[7:0]	data[31:24]	data[23:16]	data[15:8]	data[7:0]	0xF0

其中，指令类型 Addr 用于表明指令的作用，不同的值代表不同功能，如下：

表 3-2 指令类型

指令类型	功能
0	restart，写入任意值使能采样
1	ChannelSel，通道选择，每一位对应一个通道
2	DataNum，采样数据个数
3	ADC_Speed_Set，ADC 采样速率设置
4	ADC_Soft_Mode，配置 ADC 内部寄存器，部分 ADC 有效
5	Pll_Reconf，PLL 输出时钟重配置
6~254	预留
255	devinfo，设备信息查询

实际运行时，只需要根据需求发送对应指令即可，当所有指令发送完毕后，发送指令 0，写入任意值，便能驱使 ADC 以新的配置方式工作。

3.2.1 PL 端逻辑系统设计

PL 端硬件逻辑系统通过 BD 搭建，完整结构如下图 3-4：

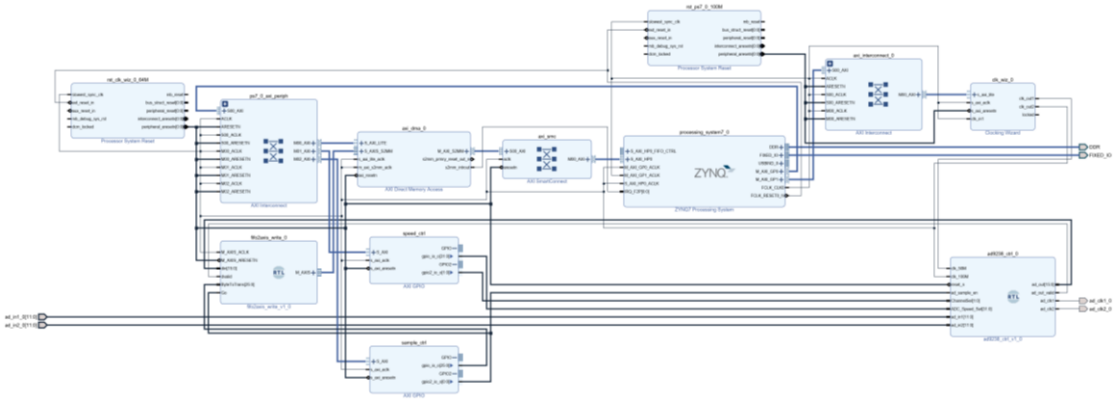

图 3-4 硬件逻辑系统

上图如果不清晰，用户可直接参看例程。在该逻辑系统设计中，ad9238_ctrl 和 fifo2axis_write 是通过 Add Module 方式添加的 IP，即使用 verilog 模块设计转换而来的 IP，可以在 Sources 中查看到这两个 IP 的源码，如下图 3-5：

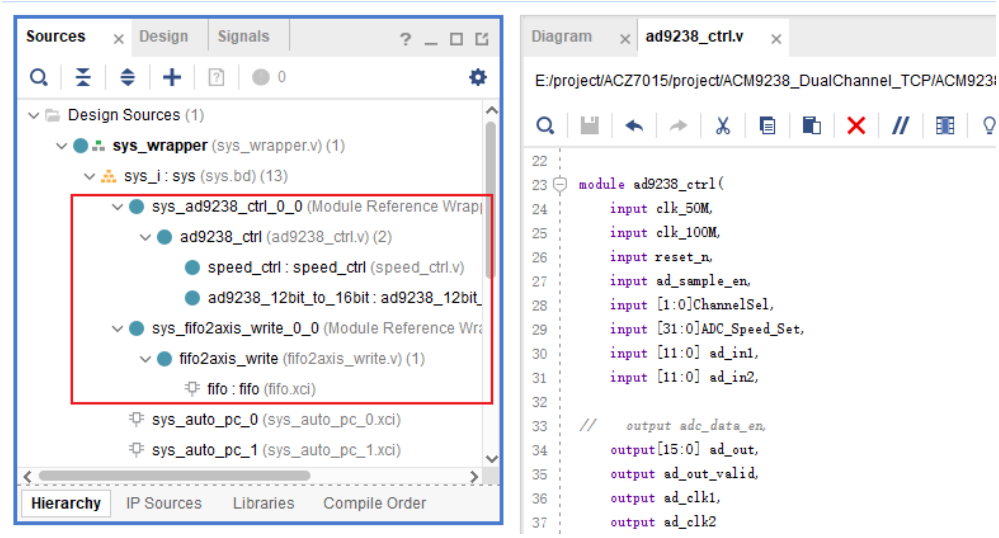

图 3-5 ad9238_ctrl 与 fifo2axis_write 结构

这里，ad9238_ctrl 由 speed_ctrl 和 ad9238_12bit_to_16bit 模块构成，fifo2axis_write 则是还包含了一个 fifo，它们的功能如下表：

表 3-3 ad9238_ctrl 与 fifo2axis_write 模块功能

模块	功能
ad9238_ctrl	AD9238 控制模块，为双通道 adc 提供时钟；通过 AXI4-lite 接口接收指令，控制 ADC 采样频率以及采样通道；接收 ADC 采集的数据，进行有无符号转换后输出，并产生数据有效信号
speed_ctrl	ADC 采样频率控制模块，对 50Msps 的采样结果数据进行抽取重采样
ad9238_12bit_to_16bit	ADC 通道控制以及位宽转换模块。①根据通道选择信号控制采样通道，2'b00 为 A 通道测试数据，01 为 A 通道，10 为 B 通道，11 为双通道。②对采样数据进行有无符号和位宽转换，将采样的数据+2048 转换为有符号数，再对高位补零转换为 16 位后输出
fifo2axis_write	Fifo 转 AXI4-Stream 模块，将 ad9238_ctrl 模块输出的数据存进 FIFO 中，并通过 AXI4-Stream 接口，交由 AXI DMA。
fifo	标准读 FIFO，用来缓存待搬运数据

硬件逻辑系统的分析，我们需要拆分为三个部分，这里我们逐个来看。

3.2.1.1 时钟生成与 ADC 数据接收策略

在介绍 ACM9238 模块时，我们曾介绍过，只需要为 AD9238 的两路通道都提供时钟，便能让两个通道同时工作。因此，设计中使用 PLL 生成了一路 50MHz 的时钟提供给 ad9238_ctrl，ad9238_ctrl 模块再将该时钟作为 ADC 的两路工作时钟提供给 ACM9238 模块，如图 3-6 所示：

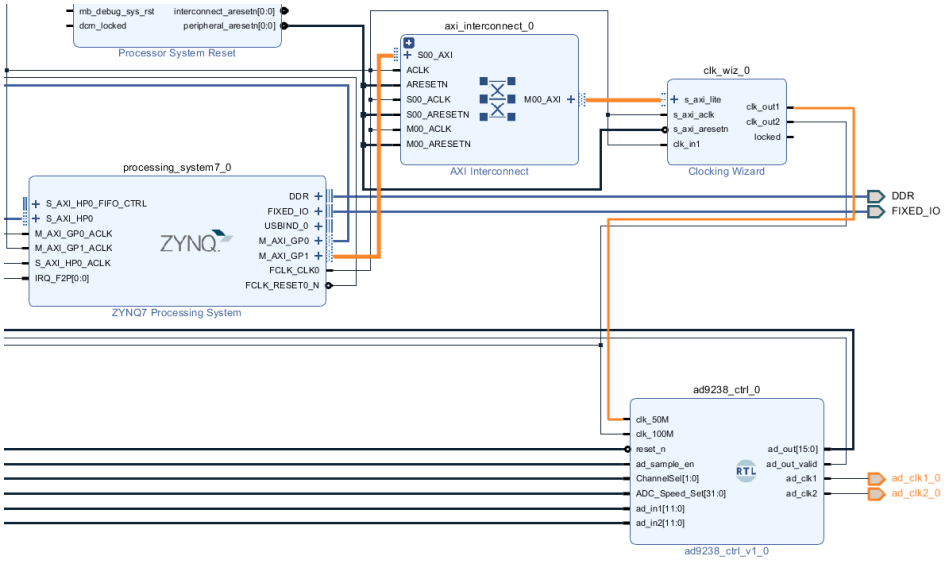

图 3-6 ADC 工作时钟产生路径

为了能够接收到 ADC 双通道的数据，这里系统中又通过 PLL 产生了一路 100MHz 的时钟给 ad9238_ctrl。ad9238_ctrl 模块内部的 ad9238_12bit_to_16bit 模块会生成一个一位位宽的 ch_flag 信号，在时钟上升沿对采样进行计数，这样便可以得到一个在 0 和 1 之间循环跳变的选通信号。

```
reg ch_flag;

always @(posedge clk)
if(ad_sample_en)
    ch_flag <= ch_flag +1;
else
    ch_flag <= 0;
```

然后，结合通道选择信号，便可以让 ad9238_ctrl 模块在双通道采样时，轮流接收两个通道的数据。而在单通道采样时，每两个时钟周期采样一次对应通道的数据。

```
always @(posedge clk)
if(ch_flag == 1)begin
    if(ad_sample_en && ch_sel == 2'b01)
        ad_out<={4'd0,s_ad_in1};//{{4{ad_in1[11]}},ad_in1};
    else if(ad_sample_en && ch_sel == 2'b10)
        ad_out<={4'd0,s_ad_in2};//{{4{ad_in2[11]}},ad_in2};
    else if(ad_sample_en && ch_sel == 2'b00)
        ad_out<={4'd0,adc_test_data};//{{4{adc_test_data[11]}},adc_tes
t_data};
    else if(ad_sample_en && ch_sel == 2'b11)
        ad_out<={4'd0,s_ad_in1};//;
end
else if(ch_flag == 0)begin
```

```
    if(ad_sample_en && ch_sel == 2'b11)
        ad_out<={4'd0,s_ad_in2};//{{4{ad_in2[11]}},ad_in2};
```

这样，便能确对每个通道数据的接收仍是按照 50MHz 的时钟频率进行的。同时，为了确保能稳定采样到数据，这里我们还需要对 100MHz 的时钟调相，具体的相位关系则会跟硬件存在一定关系（例程中为 120 度）。因此，考虑到方便性，这里的 clock wizard 勾选了动态重配置和相位占空比配置，如图 3-7 所示：

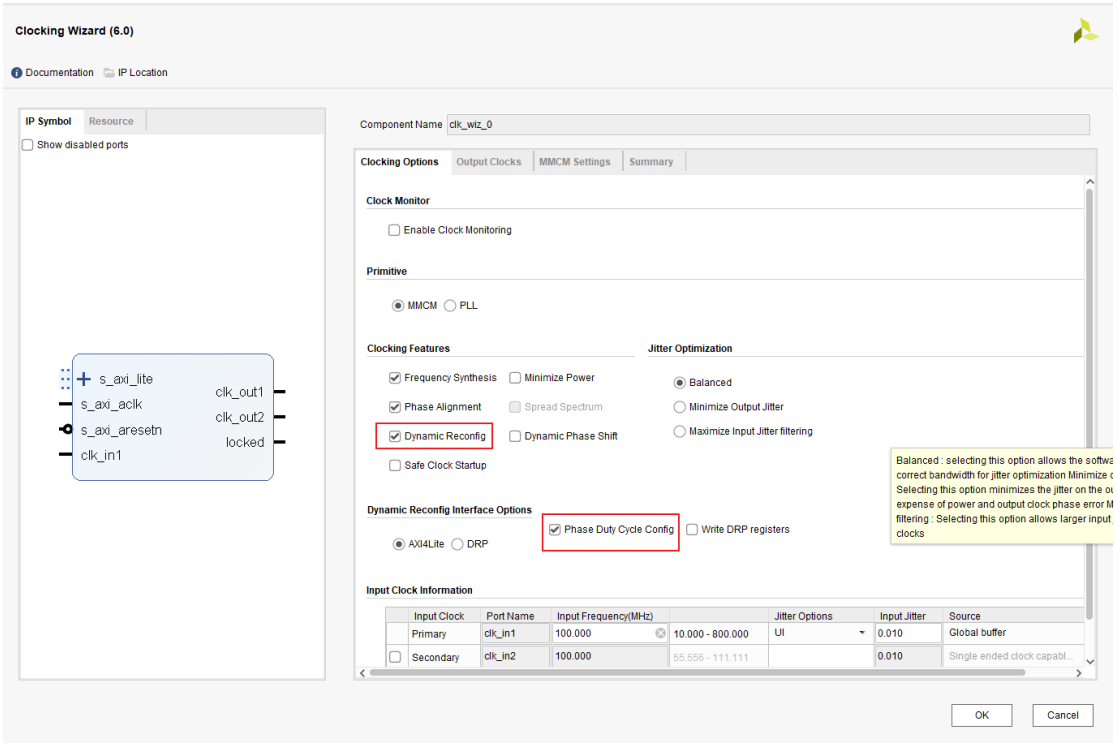

图 3-7 动态重配置与相位占空比配置

这样，我们就能够如图 3-6 中所示，在 PS 端，通过 GP1 接口动态调整输出的两路时钟的频率、相位、占空比。

3.2.1.2 ADC 采样模式控制

至于 ADC 的采样模式控制，则是通过 GP0 接口和两个 AXI GPIO 实现，如图 3-8 所示：

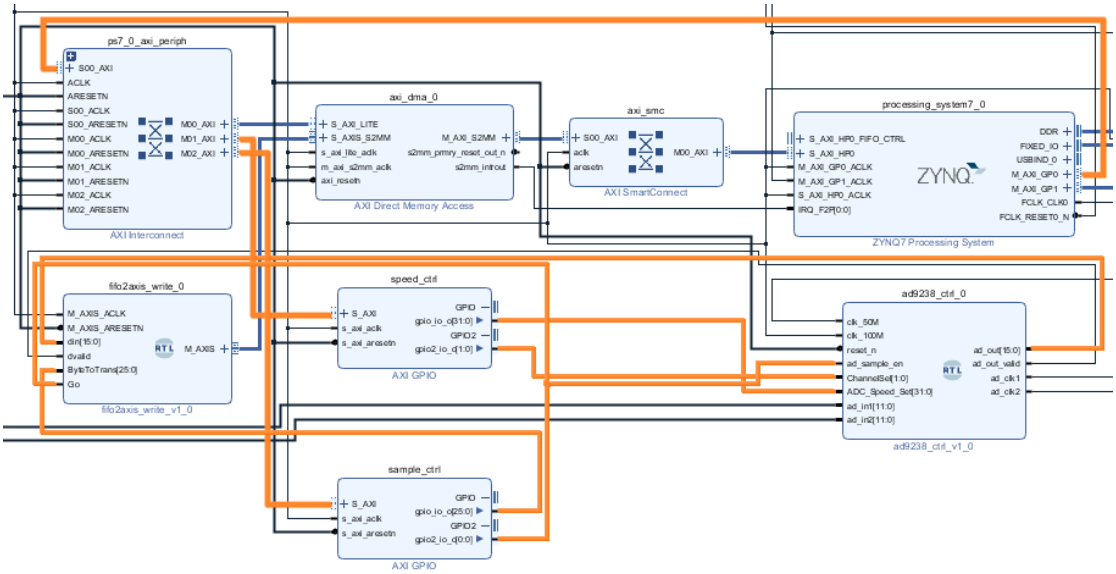

图 3-8 ADC 采样模式控制

这里的两个 AXI GPIO 都被设置为了双通道，并且每个通道的 GPIO 全部设置为了输出。PS 端通过 GP0 接口，配置 speed_ctrl 和 sample_ctrl 两个 AXI GPIO 核的通道输出不同值来进行 ADC 采样通道、采样频率、采样个数、采样状态（是否接收 ADC 采集的数据）的控制。

需要注意的是，在整个系统中，ADC 始终是以 50MHz 的采样率同时采样双通道的数据。这里的采样模式控制并不是直接控制 ADC，而是控制 ACM9238_ctrl 模块，也就是控制 ACM9238_ctrl 模块对于 ADC 采样数据的重采样以及通道的选择。

两个 AXI GPIO 各自对应通道的功能如下：

表 3-4 AXI GPIO 通道功能

通道	位宽	功能
Speed_ctrl		
GPIO	32	采样频率控制，对采样结果数据进行抽取重采样
GPIO2	2	采样通道控制，2'b00 为 A 通道测试数据，01 为 A 通道，10 为 B 通道，11 为双通道。
Sample_ctrl		
GPIO	26	采样个数，需要采样的字节数
GPIO2	1	采样使能，控制是否接收 ADC 采集的数据

当采样使能有效的时候，ad9238_ctrl 模块会根据配置信号，获取对应通道的数据并进行重采样，最终的数据会被存入 fifo2axis_write 的 FIFO 中。当 FIFO 中的数据首次满足 256 长度的 AXI4-Stream 突发后，会将 en_send 信号置 1，随后，fifo2axis_write 模块便会通过 AXI-Stream（M_AXIS）接口开始将数据发送出去，直至发送的数据量与采样个数一致。

```
case (mst_exec_state)
    IDLE:
        mst_exec_state  <= INIT_COUNTER;

    INIT_COUNTER:
    if ( (count == C_M_START_COUNT - 1) && en_send )
        mst_exec_state  <= SEND_STREAM; // 计数够时间后跳转到下一个状态
SEND_STREAM
    else
        begin
        count <= count + 1;
        mst_exec_state  <= INIT_COUNTER;
        end

    SEND_STREAM:
    if (tx_done)begin
        mst_exec_state <= IDLE;    //发送完成后回到 IDLE 状态
    end
    else
        mst_exec_state <= SEND_STREAM;
endcase

assign axis_tvalid = ((mst_exec_state == SEND_STREAM) && (read_pointer
< ByteToTrans[25:2]) && (rd_data_count >=2) && en_send);
assign axis_tlast = (read_pointer == ByteToTrans[25:2]-1);


case(state)
    ......
    2:if(Go)begin
        if(rd_data_count >= 256)
            state <= 3;
        end
        else
        state <= 0;

    3: state <= 4;

    4:begin
        en_send <= 1;
        state <= 5;
        end

    5:
    if(tx_done)begin
        en_send <= 0;
    end
```

```
    else if(!Go)begin
        state <= 0;
    end
endcase
```

3.2.1.3 AXI DMA 数据搬运

fifo2axis_write 通过 AXI-Stream 接口发送的数据会给到 AXI DMA，由 AXI DMA 通过 AXI4 接口（M_AXI_S2MM），经由 HP0 搬运到 PS 侧的 DDR3 中存储。

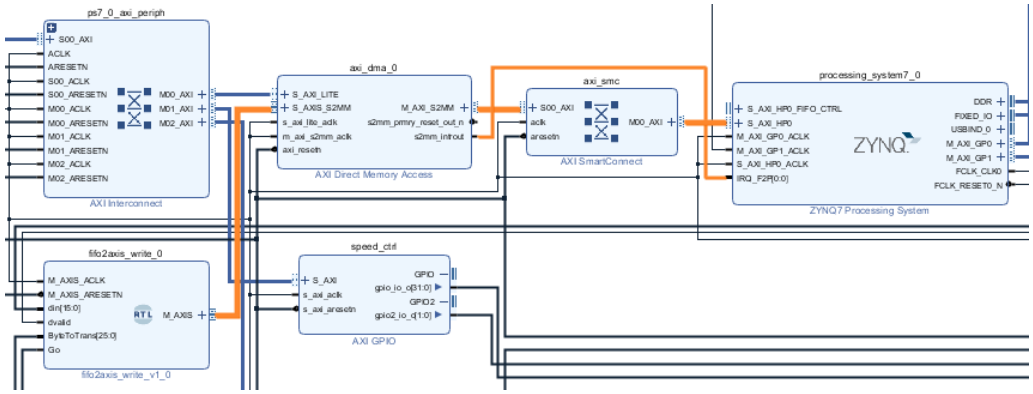

图 3-9 AXI DMA 数据搬运和中断连接

由于设计中只需要 AXI DMA 将数据搬运到 DDR 中，不需要进行读出，因此，在设置 AXI DMA 时，仅使能写通道。

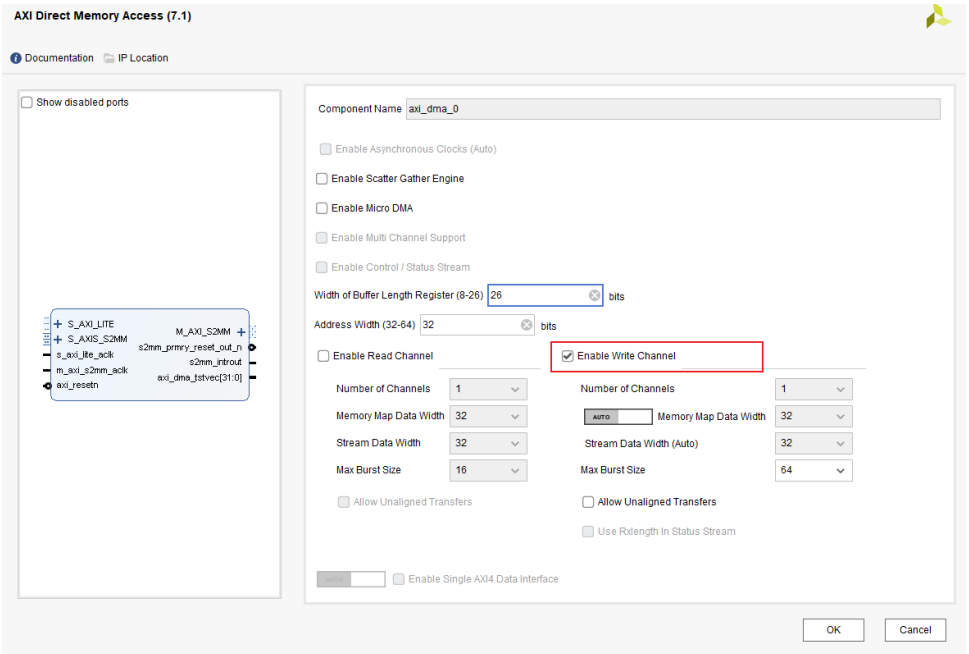

图 3-10 AXI DMA 配置

由于 03 教程中对于 AXI DMA 有专门的介绍，这里就不对其他项做过多讲解。

除了这些之外，设计中还需要使用到 AXI DMA 的中断信号，用来实现 AXI DMA 对数据的搬运。因此可以看到，在图 3-9 中，将 AXI DMA 的中断信号连接到了 IRQ_F2P[15:0]上（PL-PS 中断）。

综上，对于 ZYNQ 核的配置便有如下需求：

- 使能 PS 网口，用于接收用户 TCP 指令以及将采样的数据传输给 PC 端
- 使能 PS 串口，用于方便代码调试
- 使能 SD 与 QSPI（方便在需要时，进行程序固化）
- 使能 M_AXI_GP0 和 GP1（存在 50M 和 100M 两个时钟域）
- 使能 HP0 接口，用于 DMA 数据的高速搬运
- 设置 100MHz 的时钟输出，作为 clock wizard 的时钟输入源
- 配置 DDR，型号 MT41K256M16 RE-125，总线位宽 32bit
- 配置 Bank1 电压 LVCMOS1.8V（硬件决定）
- 使能 PL-PS 中断（IRQ_F2P[15:0]）

对应到 ZYNQ 核的 GUI 界面，配置方法如下：

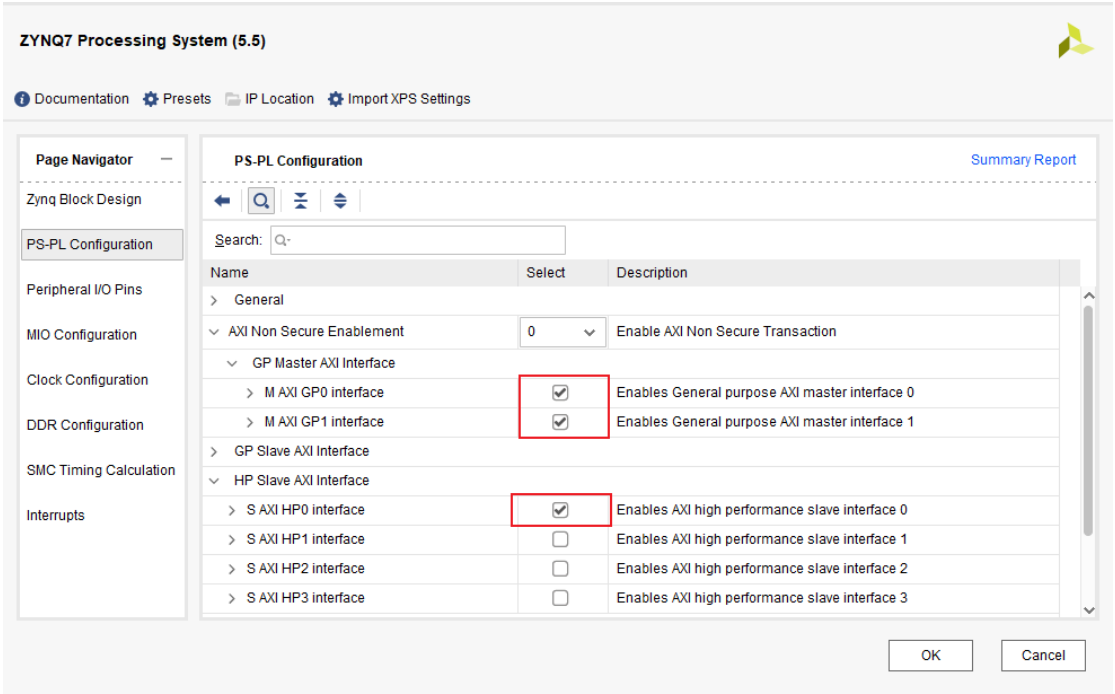

图 3-11 ZYNQ 核 AXI 接口配置

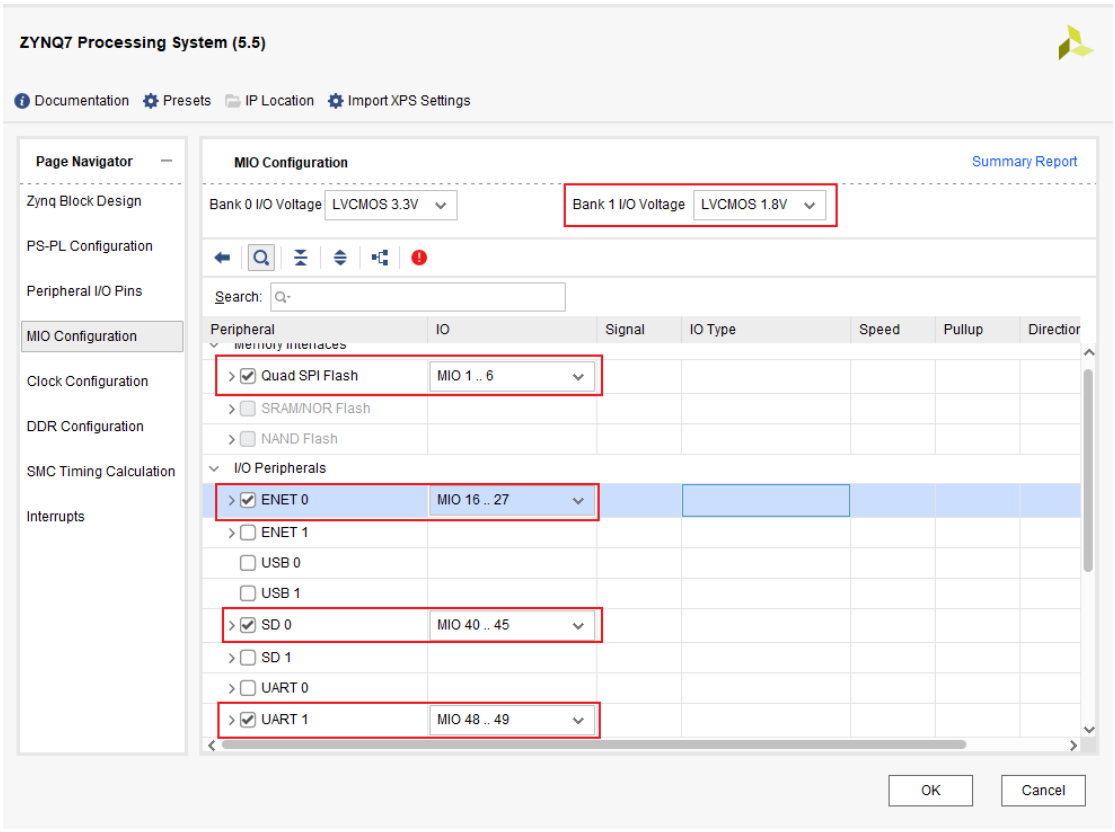

图 3-12 ZYNQ 核外设接口配置

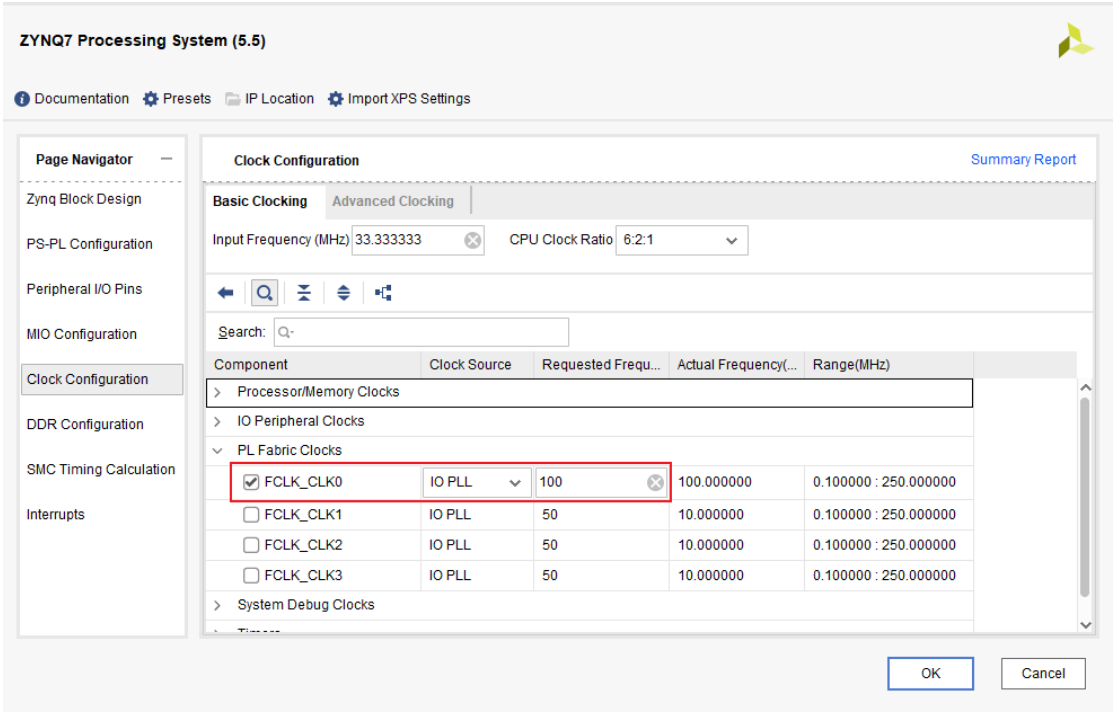

图 3-13 ZYNQ 核输出时钟配置

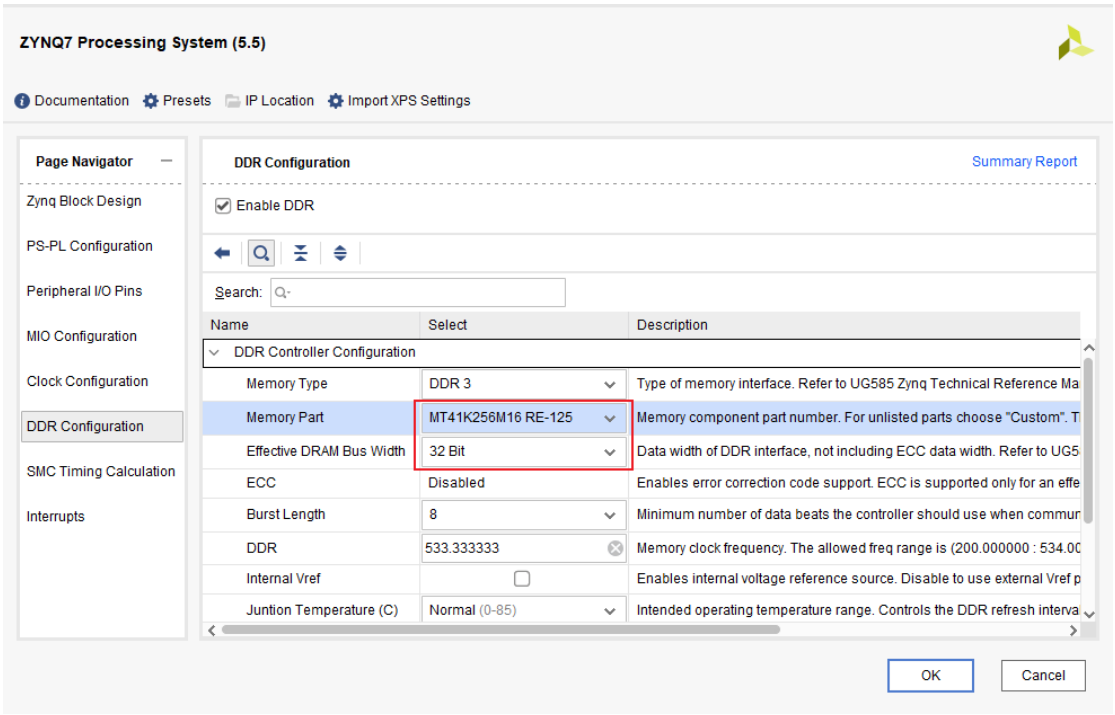

图 3-14 ZYNQ 核 DDR 配置

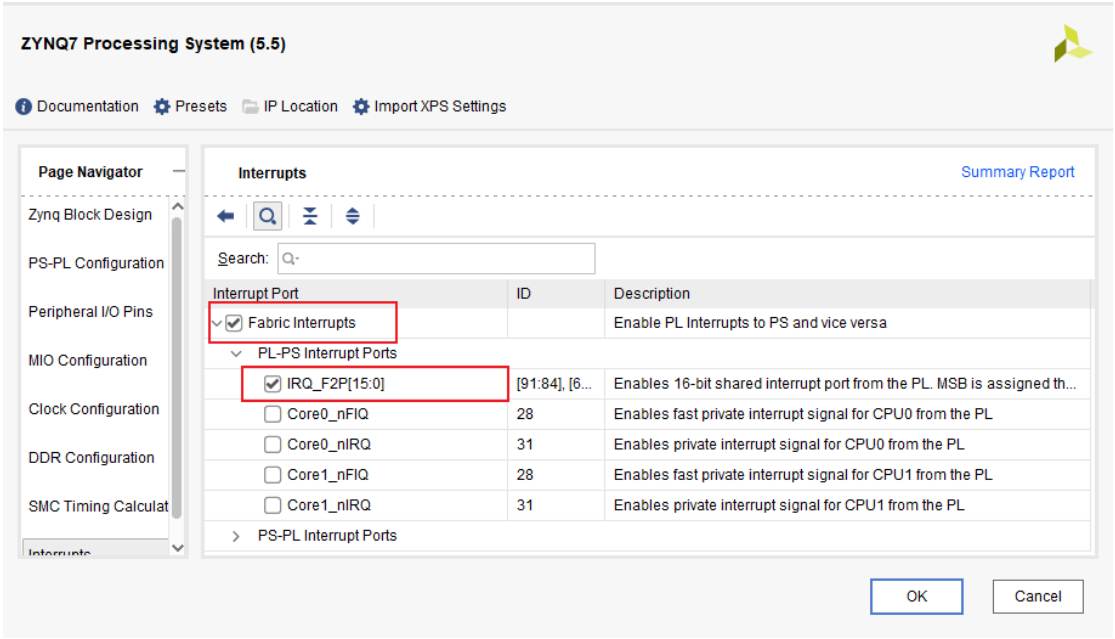

图 3-15 使能 PL 中断

以上，便是整个 PL 端的硬件逻辑系统搭建和配置。随后是管脚约束，设计只需要分配 ACM9238 的管脚。由于 ACZ7015 有两个通用 GPIO 接口，所以，插接的接口不同，对应管脚分配也不相同。例程的 xdc 文件中，以注释的方式给出了两种插接方式下的约束文件，用户可以直接拷贝。

完成以上操作后，为设计生成 bitstream，随后将 bit 与硬件一起导出，便可以打开 SDK，开始应用程序设计。

3.2.2 PS 端应用程序设计

PS 端的应用程序需要基于 xilinx 提供的 LWIP Echo Server 模板进行搭建，也就是在创建 application project 时，选择 LWIP Echo Server 模板，如图 3-16 所示：

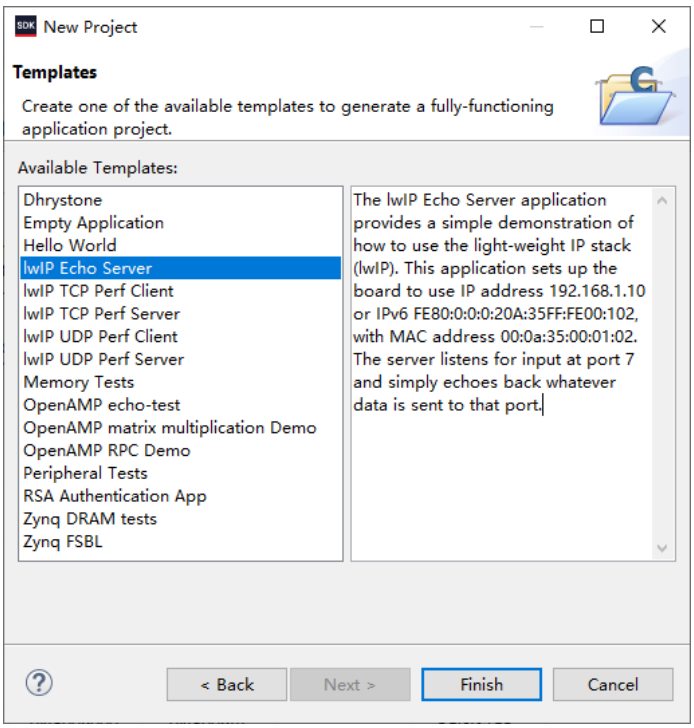

图 3-16 选择工程模板

该模板工程会将开发板设置为服务端，并设置开发板的 IP 地址和 MAC 地址，侦听指定端口上的 TCP 数据包，回环发送。

为了实现设计目标，需要对其中的 echo.c、echo.h、main.c、platform_zynq.c 四个文件内容进行修改，由于涉及到的内容较多，用户可以直接从例程中拷贝替换。修改后的四个文件其功能如下表 3-5：

表 3-5 源文件功能说明

源文件	功能
main.c	硬件以及 LWIP 初始化、接收指令处理、采样数据的发送等等
echo.c	TCP 端口绑定以及接收处理
echo.h	echo.c 中相关变量和函数声明以及宏定义
plantform_zynq.c	ZYNQ 硬件初始化

这里，我们就按照程序运行的顺序，来逐步分析代码。首先是主函数中关

于 LWIP 的初始化。

3.2.2.1 LWIP 功能初始化

LWIP 功能的初始化，该部分内容涉及较多，首先是 plantform 的初始化，通过 init_platform()函数完成。该函数位于 plantform_zynq.c 中，函数内容如下：

```
void init_platform()
{
    platform_setup_interrupts();
    platform_setup_timer();
    platform_setup_axidma();

    return;
}
```

该函数中调用了以下三个函数：

表 3-6 硬件初始化相关函数

函数	功能
platform_setup_interrupts()	初始化 GIC，注册异常中断、私有定时器中断和 AXI DMA 接收中断的处理函数，开启 GIC 中对于私有定时器和 AXI DMA 接收中断的使能
platform_setup_timer()	初始化私有定时器，设置装载值
platform_setup_axidma()	初始化 AXI DMA，设定中断触发极性和优先级

这里的私有定时器，用于定时处理 TCP 超时事件，因此设计会用到私有定时器中断。而 AXI DMA 的数据搬运也会用到中断，为了避免出现多中断冲突的问题，设计中将 AXI DMA 中断的初始化也放在了 platform_setup_interrupts 函数中。这里 AXI DMA 的接收中断处理函数被链接为 RxIntrHandler，函数内容会在采样数据发送小节讲解。

硬件初始化完成后，代码中通过 lwip_init()函数初始化 LWIP，然后使能硬件中断，初始化网络接口。该过程中会修改开发板的 IP 地址和网关地址，为了方便实验，这里将默认的 IP 地址修改为 192.168.0.2，网关地址修改为 192.168.0.1。

```
xil_printf("Configuring default IP of 192.168.0.2\r\n");
IP4_ADDR(&(echo_netif->ip_addr),  192, 168,   0, 2);
IP4_ADDR(&(echo_netif->netmask), 255, 255, 255,  0);
IP4_ADDR(&(echo_netif->gw),      192, 168,   0,  1);
```

除了 IP 地址之外，代码中也将 MAC 地址修改为了 00_0a_35_01_fe_c0。

```
/* the mac address of the board. this should be unique per board */
unsigned char mac_ethernet_address[] =
{ 0x00, 0x0a, 0x35, 0x01, 0xfe, 0xc0 };
```

3.2.2.2 建立 TCP 通信

TCP 通信的建立由 start_application()函数完成，函数具体代码如下：

```
int start_application()
{
    err_t err;
    unsigned port = 5000;

    /* create new TCP PCB structure */
    pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    /* bind to specified @port */
    err = tcp_bind(pcb, IP_ANY_TYPE, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", port,
err);
        return -2;
    }

    /* we do not need any arguments to callback functions */
    tcp_arg(pcb, NULL);

    /* listen for connections */
    pcb = tcp_listen(pcb);
    if (!pcb) {
        xil_printf("Out of memory while tcp_listen\n\r");
        return -3;
    }

    /* specify callback to use for incoming connections */
    tcp_accept(pcb, accept_callback);

    xil_printf("TCP echo server started @ port %d\n\r", port);

    return 0;
}
```

函数主要是实现 TCP 端口的绑定以及将服务端设置为监听模式，

代码中涉及到的主要函数如表 3-7 所示：

表 3-7　建立 TCP 通信相关函数

函数	功能

tcp_new_ip_type()	创建新的 TCB_PCB
tcp_bind()	将本地 IP 地址、端口号与 PCB 绑定，将绑定完毕的控制块插入 pcb 链表中
tcp_arg()	设置 TCP 连接的回调函数参数
tcp_listen()	将服务端设置为监听模式，等待 TCP 客户端的连接。当有 TCP 客户端连接时，将会调用 tcp_accept()函数进行处理
tcp_accept()	监听模式下，TCP 客户端连接时的回调函数

结合表格可知，start_application()函数依次进行了以下四个操作:

- 通过 tcp_new_ip_type ()函数创建新 PCB
- 使用 tcp_bind ()绑定 IP 地址和端口号
- 使用 tcp_listen()对连接进行监听
- 使用 tcp_accept ()设置连接回调函数。

这也是 TCP 服务端与客户端建立连接的一般步骤。值得注意的是，这里的 TCP_PCB 又被称之为 TCP 控制块，是一个十分重要的结构体。其内部定义了大量的成员变量，基本定义了整个 TCP 协议运作过程的所有需要的东西，如发送窗口、接收窗口、数据缓冲区、超时处理、拥塞控制、滑动窗口等等。

除了该控制块外，为了减少资源占用，设计中还有一个专用于监听状态的控制块 tcp_pcb_listen()，这种控制块只包含 TCP_PCB 的部分字段信息。上述的 tcp_listen()在执行时，便会新建一个 tcp_pcb_listen()，从原来的 TCP_PCB 中拷贝需要用于监听的字段后，便会删除并释放原来的 TCP_PCB。当有客户端进行连接时，再将 tcp_pcb_listen()中的内容拷贝到新建的 TCP_PCB 中，并补充空缺的字段。

当有客户端进行连接时，accept_callback()便会调用 recv _callback()函数，该函数的代码如下：

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    static int connection = 1;

    /* set the receive callback for this connection */
    tcp_recv(newpcb, recv_callback);

    /* just use an integer number indicating the connection id as the
       callback argument */
    tcp_arg(newpcb, (void*)(AL_UINTPTR)connection);

    /* increment for subsequent accepted connections */
    connection++;
    serpcb=newpcb; // 将建立的 pcb 赋予全局变量 serpcb;
```

```
    return ERR_OK;
}
```

这段代码主要实现以下两个功能：

- 设置了接收回调函数 recv_callback（），并将连接 ID 作为回调参数。
- 拷贝当前 TCP 控制块，用于主函数中服务端将 ADC 采样的数据发送到客户端

3.2.2.3 接收数据处理

当接收到客户端的数据包后，回调函数 recv_callback()会被调用。此时，我们需要它根据接收到的数据还原出指令。该函数的代码内容如下：

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                                struct pbuf *p, err_t err)
{
    int i;
    uiIdx = 0;
    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);
    unsigned char cmd_num = (p->len)/8;//计算指令数
    /* echo back the payload */
    /* in this case, we assume that the payload is < TCP_SND_BUF */
    if (tcp_sndbuf(tpcb) > p->len) {
        for(i=0;i<cmd_num;i++)
        {
            cmd_analysis(p->payload + 8*i);
        }

    } else
        xil_printf("no space in tcp_sndbuf\n\r");

    pbuf_free(p);

    return ERR_OK;
}
```

其中涉及到的函数，功能如下：

表 3-8 函数功能表

函数	功能
tcp_close()	关闭 PCB 连接
tcp_recv()	更新接收窗口，并返回已处理的数据
tcp_sndbuf()	获取发送 buffer 的大小
pbuf_free()	释放数据包 pbuf 资源
cmd_analysis()	指令分析，将接收到的数据按照帧格式转换为对应的指令

回调函数会确认接收数据的长度，并根据长度计算出包含多少个完整的指令，只要数据不超过接收缓冲区，就会通过 cmd_analysis()函数对数据进行解析。解析完成后，通过 pbuf_free()释放掉 pbuf 所占用的内存，以免造成内存资源浪费。

这里 cmd_analysis()函数代码内容如下：

```
/*
 * 根据指令格式 55 A5 addr data data data data F0 进行解析
 * addr 目前有效的值有 0~5 以及 255，其余值保留；data 为配置参数
 * addr = 0,控制 ADC 传输开始
 * addr = 1,ADC 通道选择
 * addr = 2,采样个数
 * addr = 3,采样速率设置
 * addr = 4,ADC 寄存器配置（部分 ADC 有效）
 * addr = 5,PLL 输出时钟分频设置
 * addr = 255,上位机查询 ADC 参数
*/
void cmd_analysis(uint8_t *cmd_buffer)
{
    unsigned char cmd[8] = {0};
    for(int i=0;i<8;i++){
        cmd[i] = *(cmd_buffer+i);
    }

    if((cmd[0] == 0x55) && (cmd[1] == 0xA5) && (cmd[7] == 0xF0))
    {
        switch(cmd[2])
        {
            case 0: restart = 1;break;
            case 1: ChannelSel = cmd[6];break;
            case 2: DataNum = (cmd[3] << 24) | (cmd[4] << 16) |
(cmd[5] << 8) | cmd[6];break;
            case 3: Speed_Set = (cmd[3] << 24) | (cmd[4] << 16) |
(cmd[5] << 8) | cmd[6];break;
            case 5: PLL_Reconfig = (cmd[3] << 24) | (cmd[4] << 16) |
(cmd[5] << 8) | cmd[6];
                    Reconfig = 1;
                    break;
```

```
            case 255: devinfo = 1; break;
            default:break;
        }
    }
}
```

函数会以每 8 个字节为一组，按照表 3-1 中的指令格式对数据进行分析，还原出指令内容和对应的指令数据，并给出对应的标志信号。这里支持的指令共有 6 个，指令 0~3 为一组，用来控制采样使能、采样通道、采样个数、采样速率，当采样使能为 1 时指令才会被执行；指令 5 为一组，主要是用来配置输出给 ADC 通道工作时钟的频率与相位；指令 255 为一组，用于上位机查询当前 ADC 的相关参数，从而方便上位机进行参数设定。

最终，这些指令和指令数据会在主函数的 while 循环中进行处理。

3.2.2.4 指令处理

首先是指令 255 的处理，当上位机与开发板上的通讯端口进行连接时，会下发指令查询 ADC 的参数。对应到本次设计，所用的通讯端口便是网口 tcp 协议。设计中使用结构体的形式对 ADC 参数进行存储，该部分内容位于 ADC_Info.h 中，对应代码如下：

```
typedef struct{
    unsigned int ID;//ID 序列号
    char Label[16];//产品型号
    unsigned int Max_Sample;//当前最大采样率
    unsigned char Channel_Num;//最大通道数
    unsigned char Data_Type;//数据类型,1 为补码，0 为无符号
    unsigned char Resolution;//分辨率
    unsigned char mode;//模式
    unsigned int reserved;//预留

    float Max_Volta;//最大输入电压
    float Min_Volta;//最小输入电压
    unsigned int Clk_Freq;//工作时钟 Hz
    float Cycle;//周期 ns
}ADC_info;


static const ADC_info ACM9238_info = {
    .ID = 0x00000001,
    .Label = "ACM9238",
    .Max_Sample = 50000000,
    .Channel_Num= 2,
    .Data_Type   = 1,
```

```
    .Resolution    = 12,
    .mode = 1,
    .reserved = 0,
    .Max_Volta   = 5.0,
    .Min_Volta   = -5.0,
    .Clk_Freq    = 50000000,
    .Cycle    = 20,
};
```

在主函数的 while 循环中，当检测到上位机的查询指令时，便会将该结构体中的内容通过 tcp 协议发送给上位机，对应代码如下：

```
if(devinfo == 1)
{
    unsigned char buffer[sizeof(ADC_info)];
    memcpy(buffer, &ACM9238_info, sizeof(ADC_info));
    tcp_write(serpcb,&ACM9238_info,sizeof(ADC_info),1);
    tcp_output(serpcb);
    pbuf_free(serpcb->refused_data);
    devinfo = 0;
}
```

需要注意的是，这里的 tcp_write()函数只是用于组建 TCP 报文，实际上数据的发送是通过 tcp_output()函数实现。发送完成后，通过 pbuf_free()函数释放掉内存，然后将 devinfo 标志信号置 0。

接着是 5，也就是 PLL_Reconfig，该指令用来读取 PLL 内部寄存器或修改输出时钟寄存器，其对应指令执行代码如下：

```
//指令格式：【31:28】读写控制 【27:24】输出时钟选择
//        写寄存器：【23:16】频率 【15:8】相位 【7:0】占空比
//        读寄存器：【23:0】寄存器地址
if(Reconfig)
{
    //如果是读寄存器
    if(PLL_Reconfig >>28){
        pll_read_data = Xil_In32(XPAR_CLK_WIZ_0_BASEADDR +
(PLL_Reconfig & 0xffffff));
        printf("寄存器值为%x\n",pll_read_data);
    }
    //如果是配置输出时钟
    else {
        //输出时钟按照 CLK1、CLK2....计算
        Xil_Out32(XPAR_CLK_WIZ_0_BASEADDR + 8+
12*(((PLL_Reconfig >>24)&0xf)-1),
                PLL_Reconfig & 0xffffff);
        Xil_Out32(XPAR_CLK_WIZ_0_BASEADDR + 0x25C,0x03);//updata
configuration
    }
```

```
    Reconfig = 0;
    PS_Uart_RecvData(&UartPs1, Receive_Buffer, 8);
}
```

指令对应的 32 位数据中，bit[31:28]用于表示读写，1 为读，0 为写。如果指令为读 PLL 指令，那么 bit[23:0]便用来表示要读取的寄存器。如果指令为写 PLL 指令，那么 bit[27:24]用来表示要配置的是哪路输出时钟；bit[23:16]用来设置输出时钟的分频值；bit[15:8]用来设置相位；bit[7:0]用来设置占空比。

需要注意的是，这里 PLL 的输出时钟是按照 CLK_OUT1、CLK_OUT2.....CLK_OUT6 来进行计算，时钟、频率、相位的值只支持整数。

然后是 ADC 的配置指令，当指令 0，也就是 restart 指令有效时，代表已经完成了对 ADC 的配置，此时，需要按照指令 1~3 的内容配置 ADC 的采样通道、采样个数、采样速率，然后使能 ADC 重采样。同时，还需要使能 AXI DMA 接收中断，用以接收重采样的数据。对应的代码如下：

```
if(restart)
{
    num = DataNum*2;

    Xil_Out32(XSLCR_UNLOCK_ADDR, XSLCR_UNLOCK_CODE);
    Xil_Out32(XSLCR_FPGA_RST_CTRL_ADDR, 0x0F);
    usleep(50);
    // and release the FPGA Reset Signal
    Xil_Out32(XSLCR_FPGA_RST_CTRL_ADDR, 0x00);
    Xil_Out32(XSLCR_LOCK_ADDR, XSLCR_LOCK_CODE);

    usleep(50);

    start_trans_data(num);
    Xil_DCacheDisable();

    XGpio_WriteReg((ADC_GPIO_BASEADDR),  ((ADC_CH_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_TRI_OFFSET, (~0xFFFFFFFF));
    XGpio_WriteReg((ADC_GPIO_BASEADDR),  ((ADC_SPEED_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_TRI_OFFSET, (~0xFFFFFFFF));
    XGpio_WriteReg((SAMPLE_GPIO_BASEADDR),  ((SAMPLE_EN_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_TRI_OFFSET, (~0x1));
    XGpio_WriteReg((SAMPLE_GPIO_BASEADDR),  ((SAMPLE_BTT - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_TRI_OFFSET, (~0xFFFFFFFF));

    XGpio_WriteReg((ADC_GPIO_BASEADDR),  ((ADC_SPEED_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_DATA_OFFSET, (Speed_Set));
    XGpio_WriteReg((ADC_GPIO_BASEADDR),  ((ADC_CH_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_DATA_OFFSET, (ChannelSel));
```

```
    XGpio_WriteReg((SAMPLE_GPIO_BASEADDR),  ((SAMPLE_BTT - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_DATA_OFFSET, (num));
    XGpio_WriteReg((SAMPLE_GPIO_BASEADDR),  ((SAMPLE_EN_CHANNEL - 1) *
XGPIO_CHAN_OFFSET) + XGPIO_DATA_OFFSET, (1));

    restart = 0;

    busy = 1;
}
```

由于每个 ADC 数据占两字节（12 位按两字节处理），所以这里的采样个数 DataNum 需要乘 2 得到对应的字节数。然后为了避免 FIFO 中有残余的数据对传输造成影响，这里会先通过 Xil_Out32()复位 PS 提供给 PL 的 FCLK_CLK0 时钟，从而复位整个 PL 系统，以清除 FIFO 内部。

再然后，通过 start_trans_data()函数使能 AXI DMA 接收中断，设定要接收的字节数为 num。函数的本体位于 echo.c 中，内容如下：

```
int start_trans_data(unsigned int len)
{
    int Tries = NUMBER_OF_TRANSFERS;
    int Index;
    u8 *RxBufferPtr;
    int Status;
    RxBufferPtr = (u8 *)RX_BUFFER_BASE;

    /* Disable all interrupts before setup */

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    /* Initialize flags before start transfer test  */
    RxDone = 0;
    Error = 0;

    for(Index = 0; Index < Tries; Index ++) {
```

```
        Status = XAxiDma_SimpleTransfer(&AxiDmaInstance,(UINTPTR)
RxBufferPtr,
                len, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
    }

    return XST_SUCCESS;
}
```

其中的 XAxiDma_SimpleTransfer()函数便是用于向系统提交一次 AXI DMA 的传输申请，函数中，接收数据的存储起始地址被设定为 0x1800000。

设定完 AXI DMA 中断后，通过 XGpio_WriteReg()函数，使能 sample ctrl 和 speed_ctrl 两个 AXI GPIO 的通道输出。然后根据指令，设定通道输出对应的值从而配置 ADC 的重采样。

3.2.2.5 采样数据发送

由于我们已经使能了 AXI DMA 的接收中断并设定了接收字节数 num，当对 ADC 的重采样开始时，数据会被 AXI DMA 搬运到 DDR3 中的 0x1800000 为起始地址的存储空间中。当搬运数量达到 num 时，便会触发接收完成中断。

此时需要关闭重采样使能，也就是控制 sample_ctrl 的采样使能通道输出 0，同时，给出接收完成标志。因此，AXI DMA 的接收中断处理函数中代码如下：

```
static void RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
//  int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        return;
    }
```

```
    /*
     * If completion interrupt is asserted, then set RxDone flag
     */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        XGpio_WriteReg((SAMPLE_GPIO_BASEADDR),  ((SAMPLE_EN_CHANNEL -
1) * XGPIO_CHAN_OFFSET) + XGPIO_DATA_OFFSET, (0));
        RxDone = 1;
    }
}
```

这里的 RxDone 便是接收完成标志信号。而在主函数的 while 循环中，当检测到 RxDone 标志信号有效时，便会从 DDR3 中读出数据并通过 tcp 协议发送给 PC 端。代码如下：

```
while (1) {
    if (TcpFastTmrFlag) {
        tcp_fasttmr();
        TcpFastTmrFlag = 0;
    }
    if (TcpSlowTmrFlag) {
        tcp_slowtmr();
        TcpSlowTmrFlag = 0;
    }
    xemacif_input(echo_netif);

    if(RxDone) {
        busy = 0;
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
        buff = tcp_sndbuf(serpcb);
        if(buff >= num){
            tcp_write(serpcb,(void*)(0x1800000 + uiIdx),num,1);
            tcp_output(serpcb);
            RxDone = 0;
            uiIdx = 0;
            pbuf_free(serpcb->refused_data);
        }
        else{
            tcp_write(serpcb,(void*)(0x1800000 + uiIdx),buff,1);
            tcp_output(serpcb);
            num = num - buff;
            uiIdx += buff;
            pbuf_free(serpcb->refused_data);
        }
    }
```

```
    transfer_data();

}
```

发送前需要先判断待发送数据与 tcp 发送 buffer 容量的关系，如果数据大于发送 buffer 容量，需要分多次读取发送。这里使用 uiIdex 来记录当前发送了多少字节数据，因此待发送数据的地址可以用数据起始地址 0x1800000+uiIdx 组成。

当数据发送完成后，将接收完成标志位 RxDone 拉低，将 uiIdex 清零。同时，为了避免内存的占用，还需要通过 pbuf_free()函数释放掉 tcp 发送 buffer 所占的内存。

以上，便是 PL 端逻辑系统设计和 PS 端应用程序设计的内容，接下来，我们通过实际的硬件来对设计进行验证。

3.3 板级验证

3.3.1 实验所需硬件

1. ACZ7015 开发板 x1
2. ACM9238 模块 x1
3. 信号发生器 x1
4. 千兆网线 x1
5. DC 电源线 x1
6. Type-c 下载线 x1

3.3.2 硬件连接

实验硬件连接如图 3-17 所示：

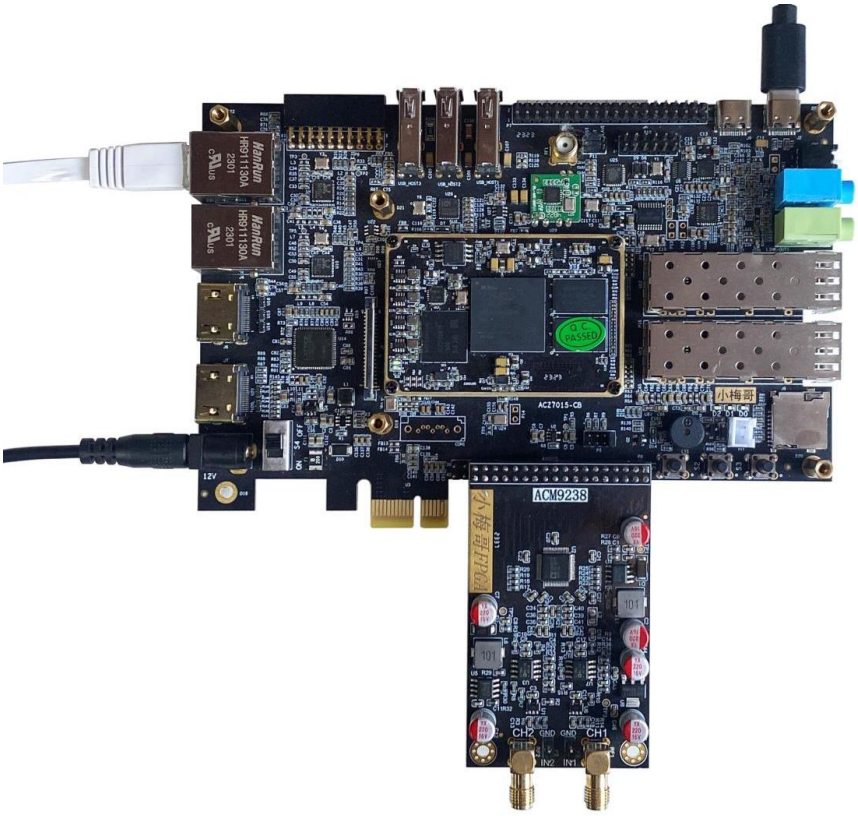

图 3-17 硬件连接

设计使用到的是 PS 侧网口，因此使用千兆网线时，注意网线一端连接开发板 PS 网口（开发板背面丝印 PS_ETH），一端连接电脑网口。ACM9238 模块的插接需要根据引脚分配来确定是插接在 GPIO0 还是 GPIO1，插接时请确保模块的 1 脚（焊盘上为方形，或引脚丝印为 1）与开发板上 GPIO 的 1 脚相接。

3.3.3 修改电脑 IP 地址

在 LWIP 功能初始化小节中，我们通过代码将开发板 IP 地址设置为了 192.168.0.2，因此，我们需要将电脑 IP 地址设为同一网段。

在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态。需要看到在活动网络中有以太网连接存在，才表明开发板和电脑的网络才已经连通。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意，如图 3-18 所示：

图 3-18 查看网络连接状态

点击“以太网”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如图 3-19 所示：

图 3-19 查看网络速度

在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址（默认网关可以不设置）。如图 3-20 所示：

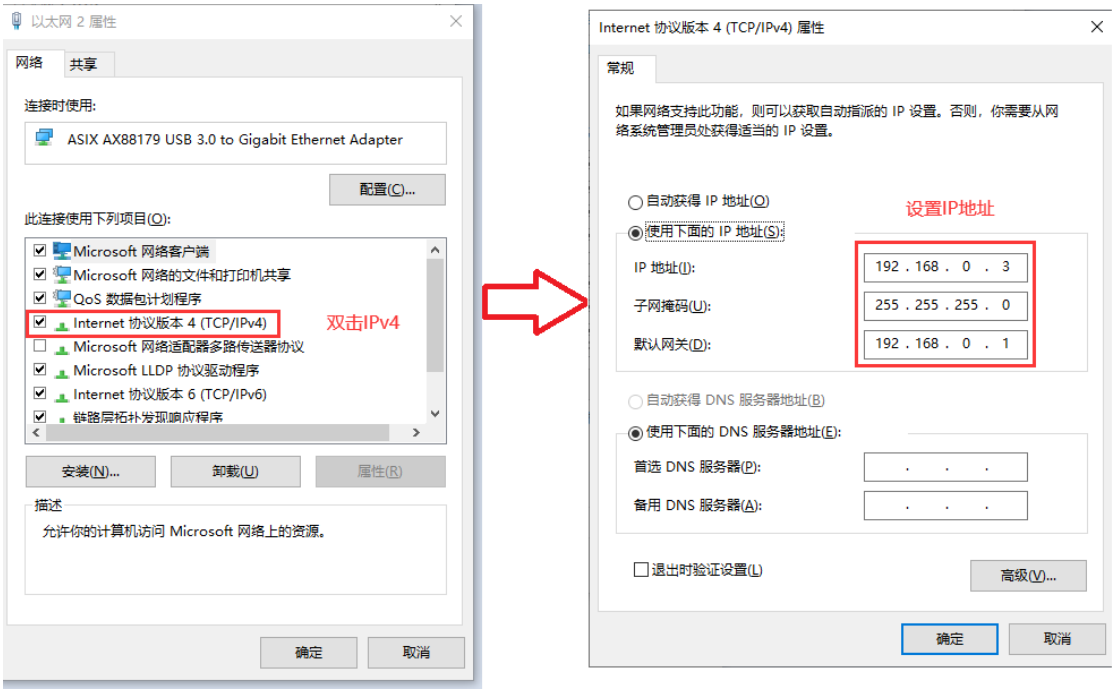

图 3-20 修改 PC 的 IP 地址

如果用户电脑中相关驱动缺失文件，会出现弹窗报错“出现了一个意外的情况，不能完成所有你在设置中所要求的更改。”导致修改失败，对于该情况用户可以参考下帖：

设置静态 IP 时提示“出现了一个意外的情况，不能完成所有你在设置中所要求的更改”

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29229

(出处: 芯路恒电子技术论坛)

更改完 IP 地址后，还需要检查是否有 arp 绑定，以管理员权限打开 CMD，输入以下命令查询 arp：

```
arp -a
```

查询界面如图 3-21 所示：

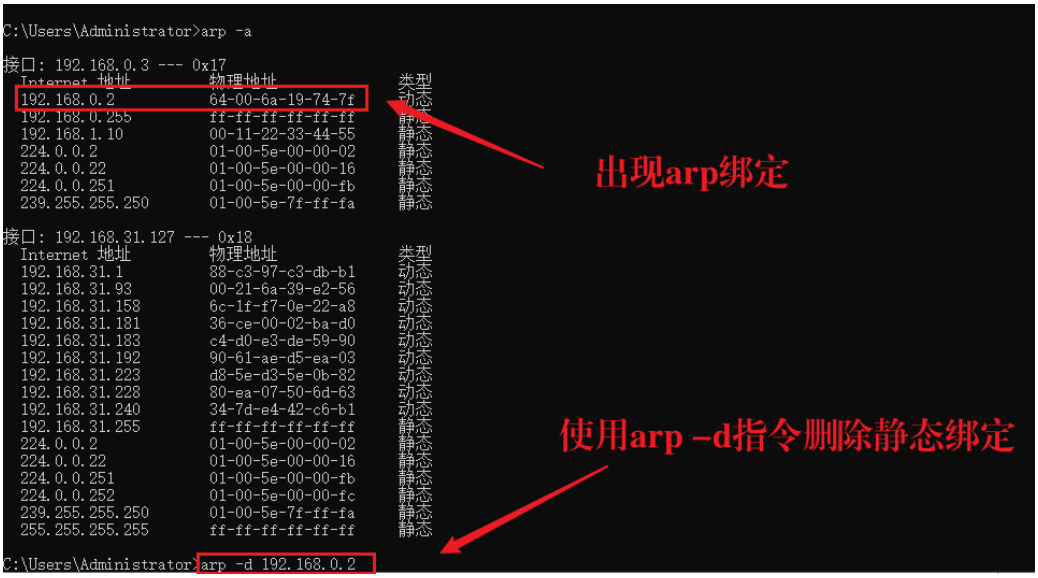

图 3-21 arp 查询与删除

如果出现了图中的静态绑定，则需要使用以下指令进行删除:

```
arp -d 192.168.0.2
```

3.3.4 代码烧录

由于 LWIP 的初始化需要一段时间，因此在烧录代码之前，我们需要在 SDK 终端中连接串口，通过串口打印的信息观察。串口的端口号需要通过设备管理查询，在端口栏中，名字为 CH9102 的便是 PS 端串口。譬如，这里笔者电脑上分配给开发板 PS 端串口的端口号为 COM20。

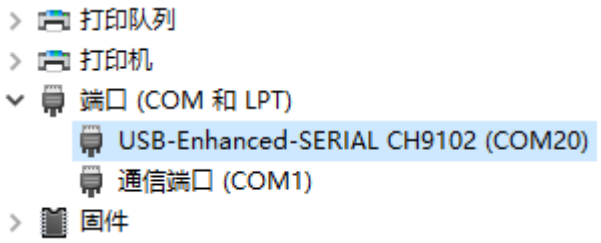

图 3-22 查询端口号

读者连接串口时，请以自己电脑上分配的端口号为准。连接完串口后，便可以烧录程序，LWIP 的初始化需要等待一段时间，当串口打印出图 3-23 所示信息时，便代表初始化完成，此时便可以连接以太网上位机进行功能验证。

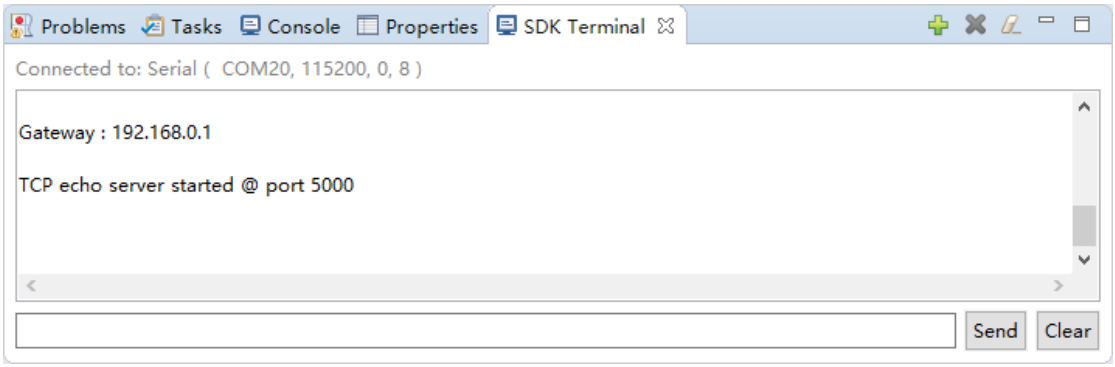

图 3-23 LWIP 初始化完成

3.3.5 功能验证

这里我们需要通过信号发生器为 ACM9238 的 A 通道输入一路 100KHz，幅度 2.5V 的正弦波，B 通道输入一路 50KHz，幅度 5V 的三角波。接着，通过我们提供的上位机软件“ 小梅哥数据采集仪”采集数据。该软件的最新下载链接如下所示：

基于 QT 的小梅哥数据采集系统上位机软件使用说明

https://www.corecourse.cn/forum.php?mod=viewthread&tid=29728

双击上位机软件，初始界面如下图 3-24 所示。

图 3-24 上位机软件初始界面显示

这里我们需要点击齿轮图标，选择通信方式为以太网 TCP 并连接，如图 3-25 所示：

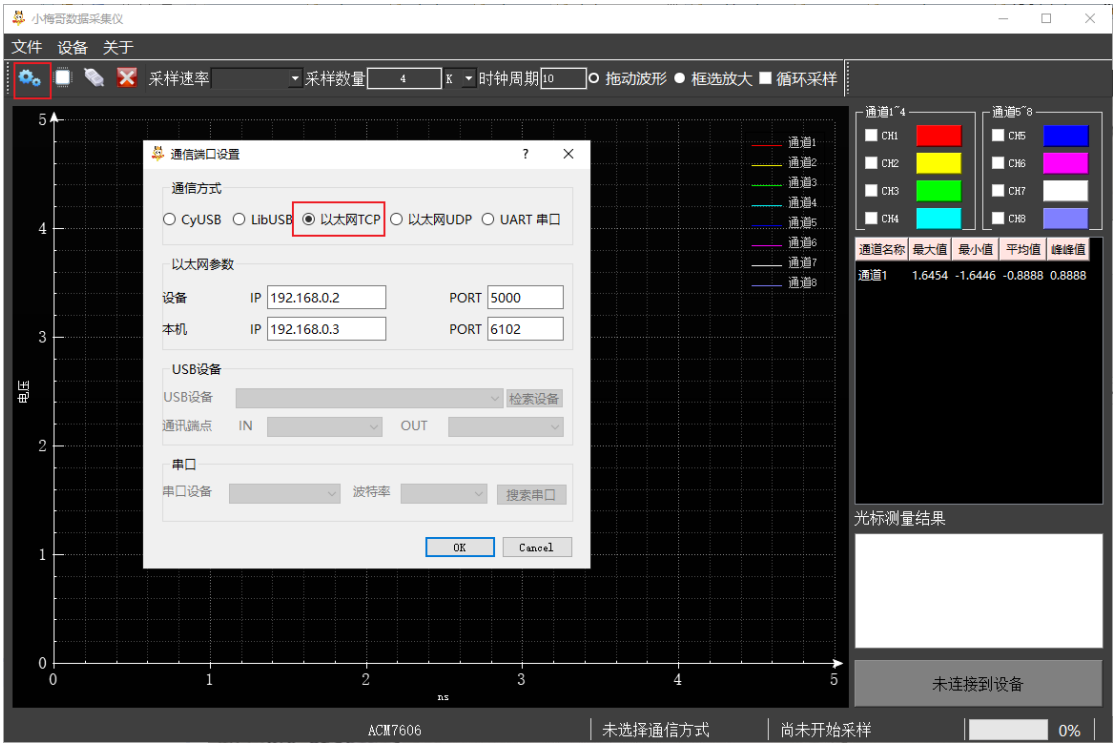

图 3-25 连接通信端口

随后，点击连接，软件会下发指令 devinfo 来获取设备信息，并用于设置上位机。如图 3-26 所示：

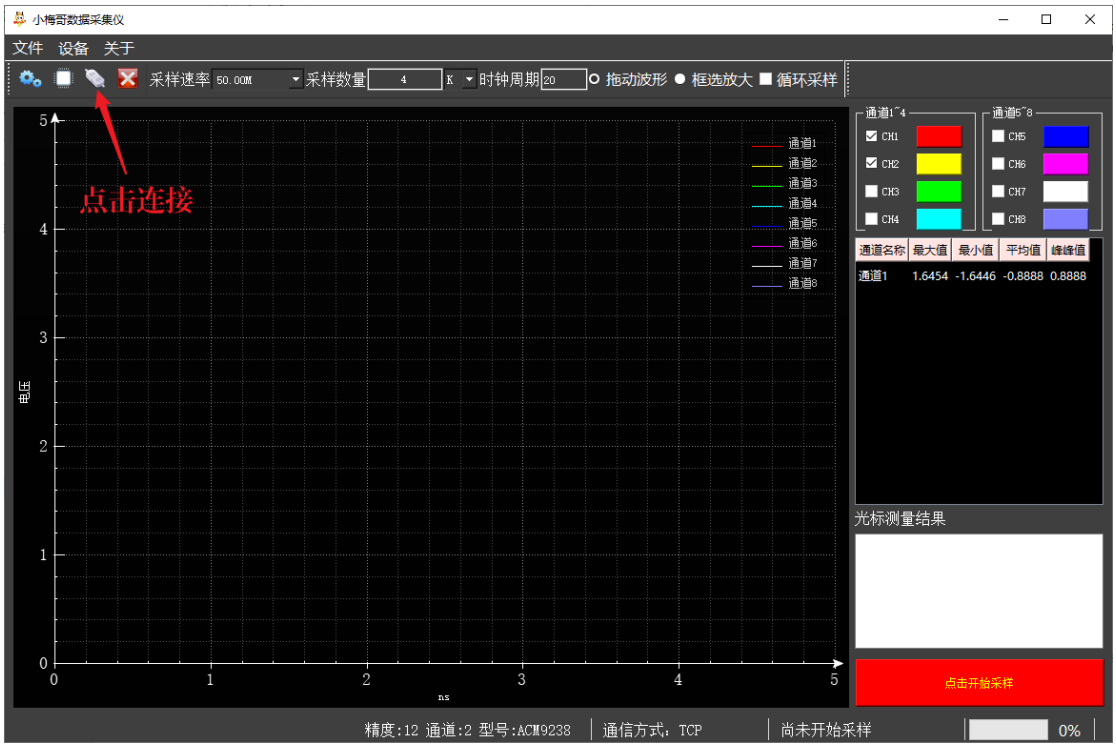

图 3-26 上位机获取 ADC 信息

这里采样速率和采样通道默认都基于获取的信息进行设置，采样数量默认为 4K，用户也可以手动进行更改。

接下来，点击开始采样，采集到的波形如图 3-27 所示：

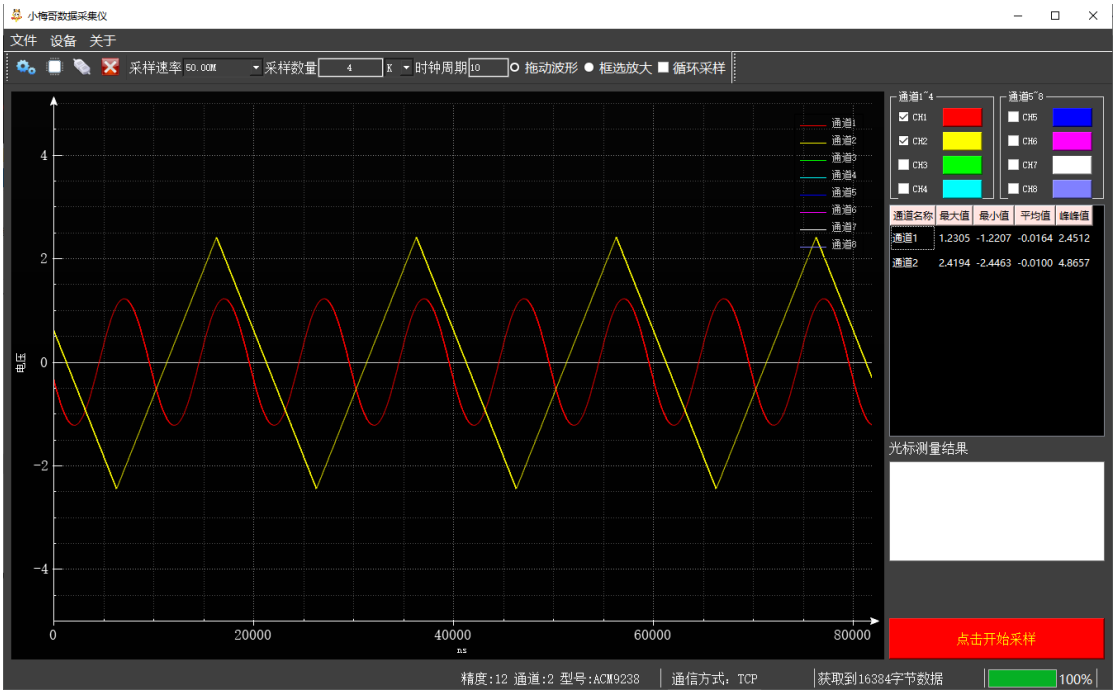

图 3-27　上位机波形显示

可以看到，通道 1 采集到的是正弦波，通道 2 采集到的是三角波，对应的峰峰值分别为 2.4512V 和 4.8657V，属于正常误差范围内。同时，默认设置的采样数量为 4K，每个数据大小为 2 字节，因此双通道情况下一共需要采集 4*1024*2*2=16384 字节，实际获取到的字节数与该数量一致。

因此，无论是采集到的波形还是总的数据量都与预期中一致，也就说明设计没有问题，代码能够按照预期运行。

3.4 思考与总结

本次实验介绍了基于 ACM9238 的双通道采集以太网 TCP 收发设计，通过 AXI DMA+DDR+以太网 TCP+上位机的方式，实现了对 ADC 的采样控制、数据存储、波形显示。通过这样一套方案，用户可以灵活简便地控制 ADC 以预期的模式运行，并且直观地观察到采样结果。

设计中只是 ACM9238 以及以太网 TCP 为例进行讲解，实际使用时，用户也可以基于该架构，根据自身需求，修改 ADC 驱动以及通讯端口，从而应用于更多的场景。