

1 USB 应用之数据采集 DDR3 缓存实验

章节导读

本章实验将通过 ACZ702 开发板上外接的 USB 模块（ACM68013）接收电脑端发送过来的命令帧，然后 FPGA 从 USB 下发的数据中解析出命令，从而实现对数据采集模块的采样频率、数据采样个数以及采样通道的合理配置。配置完成之后，数据采集模块开始采集数据，并将采集的数据存储进 DDR3 中，再由 ACM68013 模块将 DDR3 中的数据传输至电脑。用户可以在电脑上通过 FX2_USB 调试工具 CyControl 进行指令的下发，并以文件的形式保存接收到的数据，然后使用 MATLAB 软件进行进一步的数据处理分析。需要注意的是，因为 ACZ702 开发板的 40pin 的排针上已经接了一个 USB 模块，就没有办法再外接一个 ADC 模块，所以本次实验数据采集的信号是 FPGA 内部通过计数器产生的一个三角波的模拟波形，读者如果想要同时使用 ACM68013 模块和 ADC 模块，可以使用小梅哥店里的另一款产品——AC608_7Z010_DEV。

1.1 系统整体设计

通过电脑上的 FX2_USB 调试工具 CyControl，将命令帧进行发送，然后通过 ACZ702 开发板上外接的 ACM68013 模块接收数据，随后从 USB 下发的数据中解析出命令，从而实现对数据采集模块采样频率、数据采样个数以及采样通道的配置。配置完成之后，数据采集模块开始采集数据，并将数据采集模块采集的数据存储进 DDR3 中。最后将 DDR3 中存储的数据通过 USB 传输至电脑，电脑端对采集到的数据使用 MATLAB 进行进一步的分析。系统的整体设计框图如下图 1-1 所示。

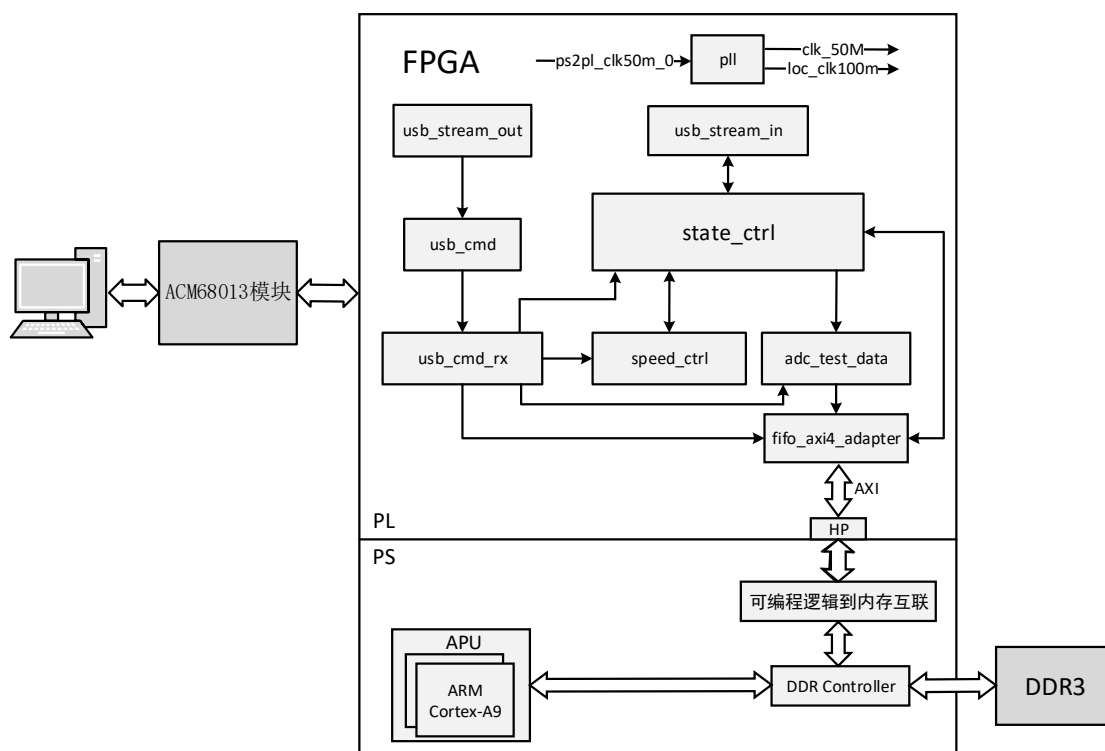


图 1-1 USB 应用之数据采集实验整体设计框图

本次实验仅针对上述图中 PL 部分的代码进行设计说明，PS 部分的设计请参看：

[【ACZ702】ZYNQ PL 读写 PS DDR3 双端口读写控制器设计](#)

<http://www.corecourse.cn/forum.php?mod=viewthread&tid=29312>

PL 部分的模块说明如下：

1. PLL 模块：锁相换 IP 模块，该模块的输入时钟为 50Mhz，由 PS 提供给 PL，输出 50Mhz 的时钟给其它模块使用，输出 100Mhz 的时钟给 DDR 控制器使用。
2. usb_stream_out 模块：USB 数据流接收控制模块，不断的将端点 2 中的数据读取出来，数据读取后直接作为端口输出。
3. usb_cmd 模块：接收转命令模块，对 USB 接收到的数据进行分析，提取出每个控制命令帧。
4. usb_cmd_rx 模块：指令转控制模块，将从接收转命令模块接收到的数据转换为相应的控制数据并分别输出到对应的模块。
5. speed_ctrl 模块：采样速率控制模块，控制数据采集模块的采样频率。
6. adc_test_data 模块：数据采集模块，该模块内部通过一个计数器产生模拟波形。

7. state_ctrl 模块: ADC 采集数据 DDR3 缓存 USB 发送状态控制模块, 协调各个模块的信号控制, 程序状态的总控制模块。
8. fifo_axi4_adapter 模块: fifo 接口到 AXI4 接口的转换模块(含 2 个 FIFO)。
9. usb_stream_in 模块: USB 数据流发送模块, 将最终采集到的数据通过 USB 发送出去。

本章实验需要设计的模块包括usb_cmd 模块、usb_cmd_rx 模块、speed_ctrl 模块、adc_test_data 模块、state_ctrl 模块和 fifo_axi4_adapter 模块。

1.2 模块设计

1.2.1 usb_cmd 模块

接收转命令模块 (usb_cmd) 将 USB 传输过来的指令数据帧进行拆解, 得到需要的指令输出传送给别的模块进行处理, 该模块的结构框图如下图 1-2 所示:

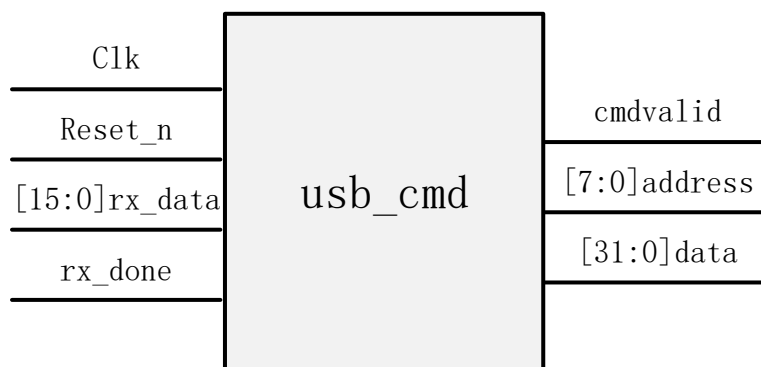


图 1-2 usb_cmd 模块的基本结构框图

模块的信号说明如下表 1-1 所示:

表 1-1 usb_cmd 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块工作时钟
Reset_n	I	模块复位信号, 低电平复位
rx_data [15:0]	I	USB 接收数据流模块接收到的 16 位数据
rx_done	I	USB 一次数据接收完成标志信号
cmdvalid	O	输出命令有效标志信号
address[7:0]	O	配置数据采集模块的寄存器地址信号
data[31:0]	O	写入到寄存器中的数据

USB 一次性发送的命令帧内容为 32 个字节, 为了实现通过 USB 修改这些寄存器的值, 需要对发送一次的数据进行拆解才能实现。对于设计的数据帧, 一帧数据一共 8 个字节, 包含帧头、帧尾、地址段、数据段。帧格式如下表 1-2

所示：

表 1-2 帧格式说明表

数据	D0	D1	D2	D3	D4	D5	D6	D7
功能	帧头 0	帧头 1	地址 address	data[31:24]	data[23: 16]	data[15:8]	data[7:0]	帧尾
值	0x55	0xA5	XX	XX	XX	XX	XX	0xF0

从上表可以看出，每帧数据一共 8 个字节，分别用 D0~D7 表示，其中，D0 和 D1 两个数据作为帧头，其值固定为 0x55、0xA5，D7 作为帧尾，其值固定为 0xF0。帧头和帧尾的作用是为了准确识别数据帧，确保接收的数据是我们需要分析的。D2 代表的是要操作的寄存器地址，D3 为要写入寄存器的数据的 24~31 位，D4 为要写入寄存器的数据的 16~24 位，D5 为要写入寄存器数据的 8~15 位，D6 为要写入寄存器的数据的 0~7 位。

该模块的作用就是将 USB 接收到的数据拆解成上述帧格式，将 D2 作为地址 address 输出，指定修改哪个寄存器，D3~D6 共 32 位作为数据 data 输出，控制数据采集模块进行相应的配置。下面将对模块中的部分代码进行说明：

首先，当检测到了 rx_done 信号，data_str 连续 4 次存储 USB 接收到的数据，组成 32 字节的命令帧。代码如下所示：

```
always@(posedge Clk)
if(rx_done)begin
    data_str[3] <= #1 rx_data;
    data_str[2] <= #1 data_str[3];
    data_str[1] <= #1 data_str[2];
    data_str[0] <= #1 data_str[1];
end
```

最后判断得到的帧命令数据是否正确，当数据符合 D0 为 8'h55，D1 为 8'hA5，D7 为 8'hF0，则代表该数据格式正确，会生成一个指令正确信号 cmdvalid 输出到指令转控制模块，并将数据进行输出，代码如下所示：

```
always@(posedge Clk or negedge Reset_n)
if(!Reset_n) begin
    address <= #1 0;
    data <= #1 0;
    cmdvalid <= #1 0;
end else if(r_rx_done)begin
    if((data_str[0][7:0]==8'h55)&&(data_str[0][15:8]==8'hA5)&&(data_str[3][15:8]==8'hF0))begin
        data[7:0] <= #1 data_str[3][7:0];
        data[15:8] <= #1 data_str[2][15:8];
        data[23:16] <= #1 data_str[2][7:0];
        data[31:24] <= #1 data_str[1][15:8];
        address <= #1 data_str[1][7:0];
    end
end
```

```
cmdvalid <= #1 1;  
end  
else  
    cmdvalid <= #1 0; end  
else  
    cmdvalid <= #1 0;
```

1.2.2 usb_cmd_rx 模块

指令转控制模块（usb_cmd_rx）将从 usb_cmd 模块接收的数据转换为相应的控制数据，首先将对寄存器进行说明，其功能和地址分别如下表 1-3 所示。

表 1-3 寄存器说明表

名称	地址	位宽	功能简介
start_sample	0	1	重新开始采集请求寄存器，向该寄存器写入任意值即可启动新一轮的采样存储传输
adc_ch_sel	1	8	通道设置寄存器，共 8 位。一共有 8 个通道可供选择，本次实验仅使用到了通道 1，当选择为通道 1 的时候才有数据输出。
set_sample_num	2	32	数据个数寄存器。如果采样 512 个数据，应该向寄存器中写入 01 00。
set_sample_speed	3	32	ADC 采样速率设置寄存器。如果设置为 0，采样和时钟保持一致 50M 时钟就是 50M 的采样速率，设置计数值后就可以改变采样频率，设置为 1 就是 25M。如果设置为 27 0F，换算成十进制是 9999，采样速率设置是 5k，计数值和采样频率之间的关系：设置计数值= Fclk/Fs - 1，Fs 是期望的采样率，Fclk 是系统时钟 50M。

usb_cmd_rx 模块的结构框图如下图 1-3 所示。

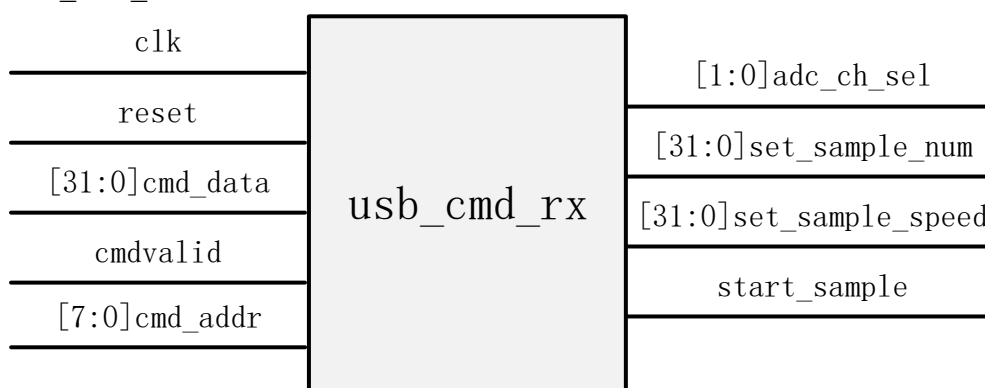


图 1-3 usb_cmd_rx 模块结构框图

模块的信号说明如下表 1-4 所示。

表 1-4 usb_cmd_rx 模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号
reset	I	模块复位信号，高电平有效
cmd_data[31:0]	I	写入到寄存器中的值

cmdvalid	I	命令有效标志信号
cmd_addr[7:0]	I	寄存器地址信号
adc_ch_sel [1:0]	O	通道设置寄存器
set_sample_num [31:0]	O	数据个数寄存器
set_sample_speed [31:0]	O	ADC 采样速率控制寄存器
start_sample	O	重新开始采集请求信号

根据表 1-3 中的内容，地址 cmd_addr 为 0 时，产生 RestartReq 信号；cmd_addr 为 1 时，得到通道设置数据 cmd_data[1:0]；cmd_addr 为 2 时，得到需要采样的数量 cmd_data[31:0]；cmd_addr 为 3 时，得到设置的采样速率的值 cmd_data[31:0]，代码如下所示：

```
always@(posedge clk or posedge reset)
if(reset)begin
    adc_ch_sel <= 2'b00;
    set_sample_num <= 32'd256;
    start_sample <= 1'b0;
    set_sample_speed <= 32'd0; //50M 采样率
end
else if(cmdvalid)begin
    case(cmd_addr)
        0: start_sample <= 1'b1;
        1: adc_ch_sel <= cmd_data[1:0];
        2: set_sample_num <= cmd_data[31:0];
        3: set_sample_speed <= cmd_data[31:0];
        4:
            begin
                adc_ch_sel <= cmd_data[1:0];
                set_sample_num <= cmd_data[23:8];
                start_sample <= 1'b1;
            end
        default;;
    endcase
end
else
    start_sample <= 1'b0;
```

1.2.3 speed_ctrl 模块

采样速率控制模块（speed_ctrl）用来控制数据采集模块的采样速率，该模块的结构框图如下图 1-4 所示：

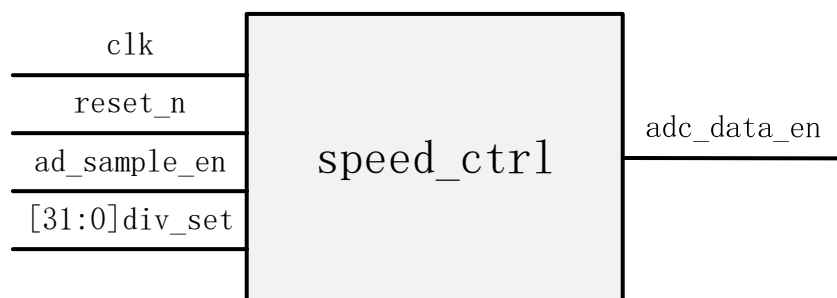


图 1-4 speed_ctrl 模块基本结构框图

对该模块的信号说明如下表 1-5 所示。

表 1-5 采样速率控制模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号
reset_n	I	模块复位信号，低电平复位
ad_sample_en	I	输入的启动采样标志信号
div_set[31:0]	I	采样频率数据控制信号， $\text{div_set} = \text{Fclk}/\text{Fs} - 1$ ， Fs 是期望的采样率， Fclk 是系统时钟 50M
adc_data_en	O	ADC 采样结果存储使能信号

speed_ctrl 模块内部通过对采样数据进行抽取重采样的方法实现对 ADC 数据采集模块的频率控制。如果 ADC 数据采集模块的最大采样速率为 50M，需要使用低于时钟频率的采样速率时，可以依旧给 ADC 提供 50M 的时钟信号，但在 FPGA 内部，对 50Msps 的采样结果数据进行抽取重采样的方法实现。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。下面我们将编写相应代码实现上述功能。

设置一个计数器 div_cnt，当产生采样使能信号 ad_sample_en 之后，计数器加 1，当计数值等于设置的 div_set 的时候，将计数器清零。代码如下所示：

```
always@(posedge clk or negedge reset_n)
if(!reset_n)
    div_cnt <= 0;
else if(ad_sample_en)begin
    if(div_cnt >= div_set)
        div_cnt <= 0;
    else
        div_cnt <= div_cnt + 1'd1;
end
else
    div_cnt <= 0;
```

计数器的计数值达到 div_set 的时候，使能 ADC 采样结果存储使能信号

adc_data_en，我们将该信号输出，最终实现每隔 div_set 个采样数据取一个结果存储或使用，从而达到对 ADC 采样频率的控制。代码如下所示：

```
always@(posedge clk or negedge reset_n)
if(!reset_n)
    adc_data_en <= 0;
else if(div_cnt == div_set)
    adc_data_en <= 1;
else
    adc_data_en <= 0;
```

1.2.4 adc_test_data 模块

ADC 测试数据模块（adc_test_data）的功能就是模拟 ADC 采集得到的数据，这里模拟的是数据位宽为 16 位的 ADC，该模块的基本结构如下图 1-5 所示：

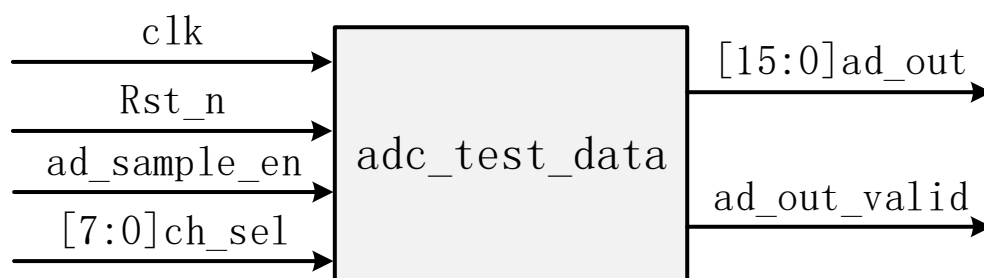


图 1-5 adc_test_data 模块基本结构框图

对该模块的信号说明如下表 1-6 所示。

表 1-6 adc_test_data 模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号
Rst_n	I	模块复位信号，低电平复位
ad_sample_en	I	输入的启动采样标志信号
ch_sel[7:0]	I	通道命令信号
ad_out[15:0]	O	输出的 16 位的数据信号
ad_out_valid	O	输出数据有效信号

为了用户能够看到变化的波形，每隔 0.5S 切换一次数据源，第一个数据源是根据 50Mhz 时钟进行计数的计数器，第二个数据源是根据 50Mhz 分频得到的 25Mhz 的时钟进行计数的计数器，下面我们将介绍具体的代码实现。

首先，设计一个计数器，计数时间为 0.5S，每隔 0.5S 波形切换标志 flag_change 反转一次，代码如下所示：

```
parameter wave_change_cnt = 25000000 - 1 ; //0.5S
always@(posedge clk or negedge Rst_n)
if(!Rst_n)
    cnt <= 26'd0;
```



```
else if(cnt >= wave_change_cnt)
    cnt <= 26'd0;
else
    cnt <= cnt + 1'd1;
always@(posedge clk or negedge Rst_n)
if(!Rst_n)
    flag_change <= 1'd0;
else if(cnt >= wave_change_cnt)
    flag_change <= ~flag_change;
```

然后将 50Mhz 的时钟分频得到 25M 的时钟信号 flag_data，代码如下所示：

```
parameter wave_data_cnt = 2 - 1;
always@(posedge clk or negedge Rst_n)
if(!Rst_n)
    cnt1 <= 9'd0;
else if(cnt1 >= wave_data_cnt)
    cnt1 <= 9'd0;
else
    cnt1 <= cnt1 + 1'd1;
always@(posedge clk or negedge Rst_n)
if(!Rst_n)
    flag_data <= 1'd0;
else if(cnt1 >= wave_data_cnt)
    flag_data <= ~flag_data;
```

然后是得到两个模拟信号源，当产生 ad_sample_en 信号之后，开始计数，一个根据 50Mhz 的时钟进行计数，一个根据 25Mhz 的时钟进行计数，代码如下所示：

```
always@(posedge clk)
    test_data <= ad_sample_en ? (test_data + 1'b1) : 16'd0;
always@(posedge flag_data)
    test_data1 <= ad_sample_en ? (test_data1 + 1'b1) : 16'd0;
```

当得到的通道命令是通道 1 时，将模拟数据信号进行输出，根据 flag_change 的值，定时切换信号源，代码如下所示：

```
always @(posedge clk)
if(ad_sample_en && ch_sel == 2'b01)
begin
    if(flag_change)
        ad_out <= test_data1;
    else
        ad_out <= test_data;
end
else
    ad_out <= 16'd0;
```

最后输出 ad_out_valid 信号，代码如下所示：

```
always @(posedge clk)
```

```
ad_out_valid <= ad_sample_en;
```

1.2.5 state_ctrl 模块

DDR3 双端口控制器的控制信号如何产生以及 USB 何时开始往外发送数据，这些问题都可以通过一个控制模块 state_ctrl 解决。该模块的结构框图如下图 1-6 所示：

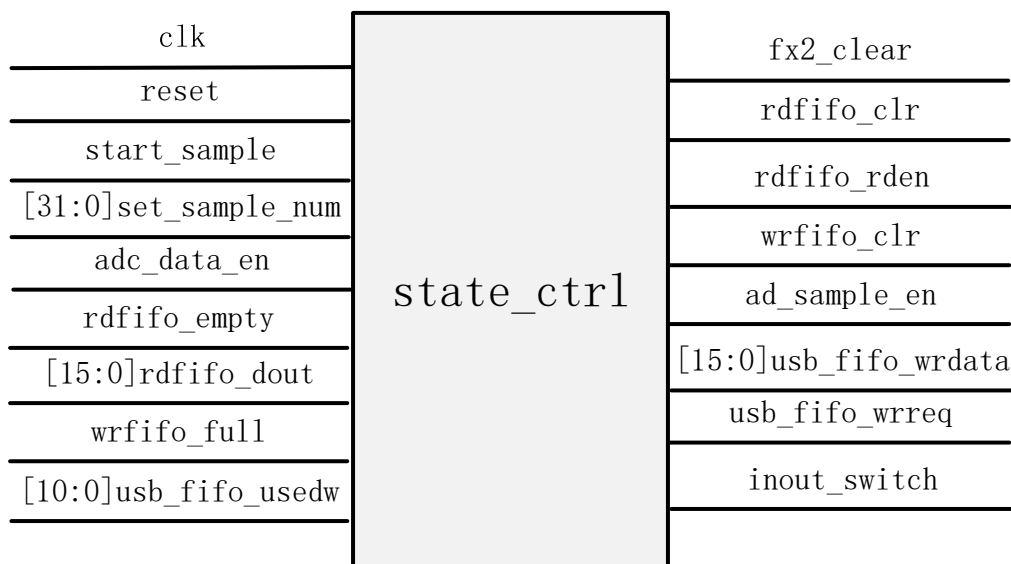


图 1-6 state_ctrl 模块基本结构框图

对于该模块的信号说明如下表 1-7 所示。

表 1-7 state_ctrl 模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号
reset	I	模块复位信号，高电平复位
start_sample	I	ADC 模块开始采样标志信号
set_sample_num [31:0]	I	设置的采样个数
rdfifo_empty	I	读 FIFO 的读空标识信号，用于标识当前 FIFO 是否为空（即 FIFO 内有无数据）
rdfifo_dout[15:0]	I	读 FIFO 的读数据输出，数据位宽为 16 位
wrfifo_full	I	写 FIFO 的写满标识信号，用于标识当前 FIFO 是否有被写满
adc_data_en	I	ADC 采样结果存储使能信号
usb_fifo_usedw[10:0]	I	USB 发送数据流的模块的写 FIFO 计数
fx2_clear	O	USB 清除信号
rdfifo_clr	O	读 FIFO 清空控制信号，给高电平表示执行清空，执行清空操作时，需保证给 3 个及以上个时钟（rdfifo_clk）周期的高电平
rdfifo_rden	O	读 FIFO 的读数据使能控制信号，给高电平表示往 FIFO 读数据，为避免读数据的丢失，确保在 FIFO 非空（rdfifo_empty = 0）情况下读数据
wrfifo_clr	O	FIFO 的写清除信号，wrfifo_clr 向外打三拍输出，保证 wrfifo 的清零信

		号的生效节拍数
ad_sample_en	O	ADC 采样使能标志信号
usb_fifo_wrdata[15:0]	O	USB FIFO 需要发送的 16 位数据
usb_fifo_wrrreq	O	USB FIFO 的写请求信号
inout_switch	O	0: read, 由 PC 向 FPGA 下发指令 1: write, 由 FPGA 向 fx2 芯片继而向 PC 上传数据

fifo_axi4_adapter 模块需要的控制信号有：wrfifo_clr、wrfifo_clk、wrfifo_wren、wrfifo_din、rdfifo_clr、rdfifo_clk、rdfifo_rden。上述信号中：wrfifo_clk 应该与数据采集模块的 clk 保持一致为 50M；rdfifo_clk 应该与 usb_stream_in 模块的 usb_fifo_wrclk 保持一致为 50M；wrfifo_wren 信号由 ADC 模块输出数据有效信号 ad_out_valid 与 adc_data_en 相与得到；wrfifo_din 信号为数据的 16 位的数据采集信号 ad_out。除去上述已经得到的控制信号外，state_ctrl 模块还需要产生的控制信号包括：wrfifo_clr、rdfifo_clr、rdfifo_rden。

USB 在每次发送数据之前，我们都应该先将其复位，这样做的目的是清除 USB 中遗留的数据，保证 USB 每次发送的都是 ADC 模块当前采集到的数据。USB 在发送数据时都是以 512 字节（USB 数据包大小）的整数倍进行发送的。

根据上述描述，我们可以通过状态机实现该模块的功能，定义状态如下：

localparam IDLE	= 4'd0;	//空闲状态
localparam DDR_WR_FIFO_CLEAR	= 4'd1;	//DDR 写 FIFO 清除状态
localparam ADC_SAMPLE	= 4'd2;	//ADC 采样数据状态
localparam DDR_RD_FIFO_CLEAR	= 4'd3;	//DDR 读 FIFO 清除状态
localparam RESET_USB	= 4'd4;	//复位 USB 状态
localparam DATA_SEND_START	= 4'd5;	//数据发送状态
localparam DATA_SEND_WORKING	= 4'd6;	//数据发送完成状态

下面编写每个状态对应的代码。

1. IDLE 状态

当处于空闲状态的时候，对 start_sample 采样起始位进行寄存，同时限定其只工作在状态 IDLE，代码如下所示：

```
always@(posedge clk or posedge reset)begin
  if(reset)
    start_sample_rm <= 1'b0;
  else if(state==IDLE)
    start_sample_rm <= start_sample;
  else
    start_sample_rm <= 1'b0;
end
```

当产生 start_sample_rm 信号之后将 inout_switch 至 1，代表由 FPGA 向 fx2 芯片继而向 PC 上传数据，跳转到 DDR_WR_FIFO_CLEAR 状态。空闲状态代码

代码如下所示：

```
begin
    if(start_sample_rm)begin
        state<=DDR_WR_FIFO_CLEAR;
        inout_switch<=1'b1;
    end
    else begin
        state<=state;
        inout_switch<=1'b0;
    end
end
```

2. DDR_WR_FIFO_CLEAR 状态

当进入写 FIFO 清零状态后，开始清除写 FIFO 内的原始数据。设置清除 DDR 写 FIFO 的计数器，保证至少 10 拍的延时，代码如下所示：

```
always@(posedge clk or posedge reset)begin
    if(reset)
        wrfifo_clr_cnt<=0;
    else if(state==DDR_WR_FIFO_CLEAR)//如果进入了清 fifo 状态
    begin
        if(wrfifo_clr_cnt==9)
            wrfifo_clr_cnt<=5'd9;
        else
            wrfifo_clr_cnt<=wrfifo_clr_cnt+1'b1;
    end
    else
        wrfifo_clr_cnt<=5'd0;
end
```

然后等待 wrfifo_full（写端 fifo 满信号）的信号拉低，拉低后，表示可以往 FIFO 里写入数据，此时进入下一个状态。在清空（复位）FIFO 的时候，FIFO 的 full 信号会变高，可以认为在复位 FIFO 时是不允许对 FIFO 进行写操作的，即使写也是不可靠的，等 FIFO 的复位结束后，full 信号会变低，就允许对 FIFO 进行写操作。DDR 写 FIFO 清除状态代码如下所示：

```
begin
    if(!wrfifo_full && (wrfifo_clr_cnt==9))
        state<=ADC_SAMPLE;
    else
        state<=DDR_WR_FIFO_CLEAR;
end
```

当处于 DDR_WR_FIFO_CLEAR 状态时，我们需要产生清除写 FIFO 的信号 wrfifo_clr，由三拍延时信号拉高提供，之所以提供的延迟信号时间为 3 拍，是为了给清 FIFO 信号足够的拉高时间，以保证清空指令的可靠，也就是在

wrfifo_clr_cnt 为 0、1 或 2 时，wrfifo_clr 置 1，否则 wrfifo_clr 为 0。

```
always@(posedge clk or posedge reset)begin
  if (reset)
    wrfifo_clr<=0;
  else if(DDR3_init_done==1'b0)
    wrfifo_clr<=1'b1;
  else if(state==DDR_WR_FIFO_CLEAR)
    begin
      if(wrfifo_clr_cnt==0||wrfifo_clr_cnt==1||wrfifo_clr_cnt==2)
        wrfifo_clr<=1'b1;
      else
        wrfifo_clr<=1'b0;
      end
  else
    wrfifo_clr<=1'b0;
end
```

3. ADC_SAMPLE 状态

进入 ADC 采样数据状态之后，首先设置 ADC 采样个数计数器 adc_sample_cnt，当产生 adc_data_en 信号之后，我们就将 adc_sample_cnt 计数值加 1，对 ADC 采集的数据进行计数。代码如下所示：

```
always@(posedge clk or posedge reset)
if(reset)
  adc_sample_cnt<=32'd0;
else if(state==ADC_SAMPLE)begin
  if(adc_data_en)
    adc_sample_cnt<=adc_sample_cnt+1'b1;
  else
    adc_sample_cnt<=adc_sample_cnt;
end
else
  adc_sample_cnt<=1'b0;
```

当 adc_sample_cnt 达到设定的采样数据个数，并且 adc_data_en 有效的时候，ADC 模块数据采集完成，跳转到读 FIFO 清除状态，代码如下所示：

```
begin
  if((adc_sample_cnt>=set_sample_num-1'b1)&& adc_data_en)
    state<=DDR_RD_FIFO_CLEAR;
  else
    state<=state;
end
```

当处于 ADC_SAMPLE 状态时，我们还需要产生采样使能信号 ad_sample_en 给到其他模块使用，代码如下所示：

```
always@(posedge clk or posedge reset)begin
```

```
if(reset)
    ad_sample_en<=0;
else if(state==ADC_SAMPLE)
    ad_sample_en<=1;
else
    ad_sample_en<=0;
end
```

4. DDR_RD_FIFO_CLEAR 状态

进入清 FIFO 状态之后，首先设置清除读 FIFO 的计数器 rdfifo_clr_cnt，保证至少 10 拍的延时。代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    rdfifo_clr_cnt<=0;
else if(state==DDR_RD_FIFO_CLEAR)//如果进入了清 fifo 状态
begin
    if(rdfifo_clr_cnt==9)
        rdfifo_clr_cnt<=5'd9;
    else
        rdfifo_clr_cnt<=rdfifo_clr_cnt+1'b1;
end
else
    rdfifo_clr_cnt<=5'd0;
end
```

然后等待 rdfifo_empty（读端 FIFO 的空信号）信号拉低，拉低后，表示 FIFO 里已经有被写入数据，此时进入下一个状态。在清空(复位)FIFO 的时候，FIFO 的 empty 信号会变高，可以认为在复位 FIFO 时是不允许对 FIFO 进行读操作的，即使读也是不可靠的，等 FIFO 的复位结束后，FIFO 被写入数据后，empty 信号会变低，就允许对 FIFO 进行读操作，然后跳转到 RESET_USB 状态，fx2_clear 信号拉高，开始清除 USB，读 FIFO 清除状态代码如下所示：

```
begin
    if(!rdfifo_empty && (rdfifo_clr_cnt==9))begin
        state<=RESET_USB;
        fx2_clear <= 1'b1;
    end
    else
        state<=state;
end
```

当处于 DDR_WR_FIFO_CLEAR 状态时，我们需要产生清除读 FIFO 的信号 rdfifo_clr，由三拍延时信号拉高提供，之所以提供的延迟信号时间为 3 拍，是为了给清 FIFO 信号足够的拉高时间，以保证清空指令的可靠，也就是在

rdfifo_clr_cnt 为 0、1 或 2 时，rdfifo_clr 置 1，否则 rdfifo_clr 为 0。

```
always@(posedge clk or posedge reset)begin
  if (reset)
    rdfifo_clr<=0;
  else if(DDR3_init_done==1'b0)
    rdfifo_clr<=1'b1;
  else if(state==DDR_RD_FIFO_CLEAR)begin
    if(rdfifo_clr_cnt==0||rdfifo_clr_cnt==1||rdfifo_clr_cnt==2)
      rdfifo_clr<=1'b1;
    else
      rdfifo_clr<=1'b0;
  end
  else
    rdfifo_clr<=1'b0;
end
```

5. RESET_USB 状态

复位 USB 状态是为了清除 USB 中遗留的数据，当进入该状态之后，USB 复位计数器 rst_usb_cnt 开始计数，计数到一定值之后，拉低 fx2_clear，使 USB 内遗留数据能够充分清除。代码如下所示：

```
begin
  if(rst_usb_cnt >= 20'hffff0)begin
    rst_usb_cnt <= 0;
    state<=DATA_SEND_START;
  end
  else if(rst_usb_cnt >= 20'h7fff0)begin
    rst_usb_cnt <= rst_usb_cnt + 1'd1;
    fx2_clear <= 1'b0;
  end
  else
    rst_usb_cnt <= rst_usb_cnt + 1'd1;
end
```

6. DATA_SEND_START 状态

进入DATA_SEND_START 状态之后，state 直接跳转到数据发送状态，USB 启动传输，代码如下所示：

```
begin
  state <= DATA_SEND_WORKING;
end
```

7. DATA_SEND_WORKING 状态

进入数据发送状态之后，当发送数据计数器 send_data_cnt 计数到需要采集的数据个数 set_sample_num 时，跳转到 IDLE 状态，完成一次数据采集发送；当

USB 发送 FIFO 计数小于 512 时，使 DDR3 双端口控制器的读 FIFO 使能信号 rdfifo_rden 为高电平，开始从 DDR3 中读数据；每发送一个 16bit 数据，如果不满足前两个条件，则重新回到本状态。代码如下所示：

```
if(send_data_cnt>=set_sample_num-1'b1)begin
    state <= IDLE;
    rdfifo_rden <= 1'b0;
end
else if(usb_fifo_usedw < 512) begin
    rdfifo_rden <= 1'b1;
    state <= DATA_SEND_WORKING;
end
else begin
    /**//每发送一个 16bit 数据，如果不满足 if 条件，则重新回到本状态
    rdfifo_rden <= 1'b0;
    state <= DATA_SEND_WORKING;
end
end
```

当 rdfifo_rden 为 1 的时候，每个时钟上升沿到来之后，send_data_cnt 计数值加 1，实现对 USB 发送的数据进行计数，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    send_data_cnt<=32'd0;
else if(state==IDLE)
    send_data_cnt<=32'd0;
else if(rdfifo_rden)
    send_data_cnt<=send_data_cnt+1;
else
    send_data_cnt<=send_data_cnt;
end
```

当 rdfifo_rden 信号到来之后，我们需要产生 USB 写 FIFO 请求信号并且需要将 DDR 读出的数据提取出来，最终交由 USB 数据流发送控制模块进行处理，代码如下所示：

```
always@(posedge clk or posedge reset)
if(reset) begin
    usb_fifo_wrreq <= 1'b0;
    usb_fifo_wrdata <= 16'd0;
end
else if(rdfifo_rden) begin
    usb_fifo_wrreq <= 1'b1;
    usb_fifo_wrdata <= rdfifo_dout;
end
else begin
    usb_fifo_wrreq <= 1'b0;
```

```
usb_fifo_wrdata <=16'd0;  
end
```

1.2.6 fifo_axi4_adapter 模块

fifo_axi4_adapter 模块的详细设计请参看：

[【ACZ702】ZYNQ PL 读写 PS DDR3 双端口读写控制器设计](#)

<http://www.corecourse.cn/forum.php?mod=viewthread&tid=29312>

需要注意的是，相较于之前的模块，新增加了一个起始信号，当接收到启动传输之后，启动 fifo2axi4 模块中的状态转移，也就是启动向 DDR 中写入数据，代码如下所示：

```
S_IDLE:  
begin  
    if(start)  
        next_state = S_ARB;  
    else  
        next_state = S_IDLE;  
end
```

该模块例化代码如下所示：

```
fifo_axi4_adapter #(  
    .FIFO_DW                (16),  
    .WR_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR + 1'b1),  
    .WR_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR + 16'd65535),  
    .RD_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR + 1'b1),  
    .RD_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR + 16'd65535),  
  
    .AXI_DATA_WIDTH         (64),  
    .AXI_ADDR_WIDTH         (32),  
    .AXI_ID                  (4'b0000),  
    .AXI_BURST_LEN          (8'd15)  
)fifo_axi4_adapter_inst  
(  
    .start                (RestartReq_dds1),  
    //clock reset  
    .clk                  (loc_clk100m),  
    .reset                 (reset),  
    //wr_fifo Interface  
    .wrfifo_clr            (wrfifo_clr),  
    .wrfifo_clk            (clk_50M),  
    .wrfifo_wren            (ad_out_valid && adc_data_en),  
    .wrfifo_din            (ad_out),  
    .wrfifo_full           (wrfifo_full),  
    .wrfifo_wr_cnt         (wrfifo_wr_cnt),  
    //rd_fifo Interface
```

```
.rdfifo_clr      (rdfifo_clr      ),
.rdfifo_clk      (clk_50M         ),
.rdfifo_rden     (rdfifo_rden     ),
.rdfifo_dout     (rdfifo_dout     ),
.rdfifo_empty    (rdfifo_empty    ),
.rdfifo_rd_cnt   (                 ),
// Master Interface Write Address Ports
.m_axi_awid      (s_axi_awid      ),
.m_axi_awaddr    (s_axi_awaddr    ),
.m_axi_awlen     (s_axi_awlen     ),
.m_axi_awsz      (s_axi_awsz      ),
.m_axi_awburst   (s_axi_awburst   ),
.m_axi_awlock    (s_axi_awlock    ),
.m_axi_awcache   (s_axi_awcache   ),
.m_axi_awprot    (s_axi_awprot    ),
.m_axi_awqos     (s_axi_awqos     ),
.m_axi_awregion  (s_axi_awregion  ),
.m_axi_awvalid   (s_axi_awvalid   ),
.m_axi_awready   (s_axi_awready   ),
// Master Interface Write Data Ports
.m_axi_wdata     (s_axi_wdata     ),
.m_axi_wstrb     (s_axi_wstrb     ),
.m_axi_wlast     (s_axi_wlast     ),
.m_axi_wvalid    (s_axi_wvalid    ),
.m_axi_wready    (s_axi_wready    ),
// Master Interface Write Response Ports
.m_axi_bid       (4'b0000        ),
.m_axi_bresp     (s_axi_bresp     ),
.m_axi_bvalid    (s_axi_bvalid    ),
.m_axi_bready    (s_axi_bready    ),
// Master Interface Read Address Ports
.m_axi_arid      (s_axi_arid      ),
.m_axi_araddr    (s_axi_araddr    ),
.m_axi_arlen     (s_axi_arlen     ),
.m_axi_arsize    (s_axi_arsize    ),
.m_axi_arburst   (s_axi_arburst   ),
.m_axi_arlock    (s_axi_arlock    ),
.m_axi_arcache   (s_axi_arcache   ),
.m_axi_arprot    (s_axi_arprot    ),
.m_axi_arqos     (s_axi_arqos     ),
.m_axi_arregion  (s_axi_arregion  ),
.m_axi_arvalid   (s_axi_arvalid   ),
.m_axi_arready   (s_axi_arready   ),
// Master Interface Read Data Ports
.m_axi_rid       (4'b0000        ),
.m_axi_rdata     (s_axi_rdata     ),
.m_axi_rresp     (s_axi_rresp     ),
```

```
.m_axi_rlast      (s_axi_rlast      ),  
.m_axi_rvalid     (s_axi_rvalid     ),  
.m_axi_rready     (s_axi_rready     )  
);
```

模块设计完成之后，只需要在顶层文件中对各个模块之间的接口信号进行连接，完整的顶层文件代码请自行查看例程文件。

1.3 建立 SDK 工程

经过以上工作，代码设计部分的任务已经全部完成，但是用到了 PS 部分的内容，所以我们需要新建一个空的 SDK 工程用于下载程序，步骤如下。

1. 编译工程，然后依次点击 File->Export->Export Hardware，导出 bit 文件，如下图 1-7 所示。

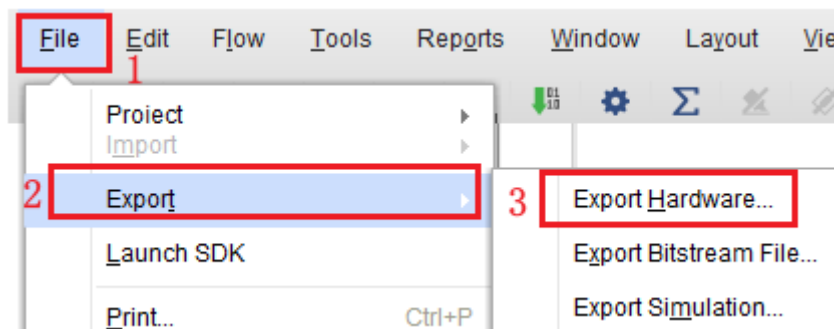


图 1-7 导出 bit 文件

2. 进入 SDK 软件，点击 File->Launch SDK，如下图 1-8 所示。

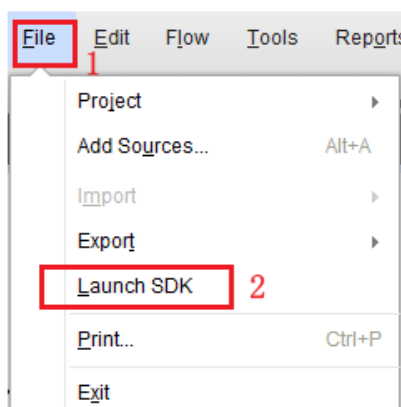


图 1-8 进入 SDK 软件

3. 进入 SDK 软件之后，依次点击 File->New->Application Project，建立一个 SDK 工程，如下图 1-9 所示，然后给工程命名，比如命名为 usb_data_capture，然后点击 Next，选择 Empty Application，就完成了工

程的创建。

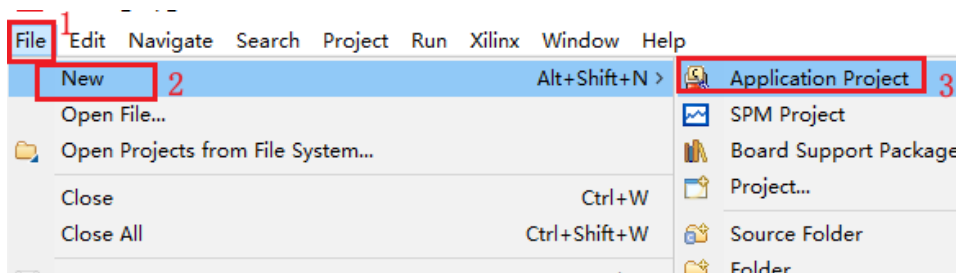


图 1-9 新建 SDK 工程

4. src 文件夹下，新建一个源文件，如下图 1-10 所示，然后在弹出的界面中给文件命名为 main.c，如下图 1-11 所示。

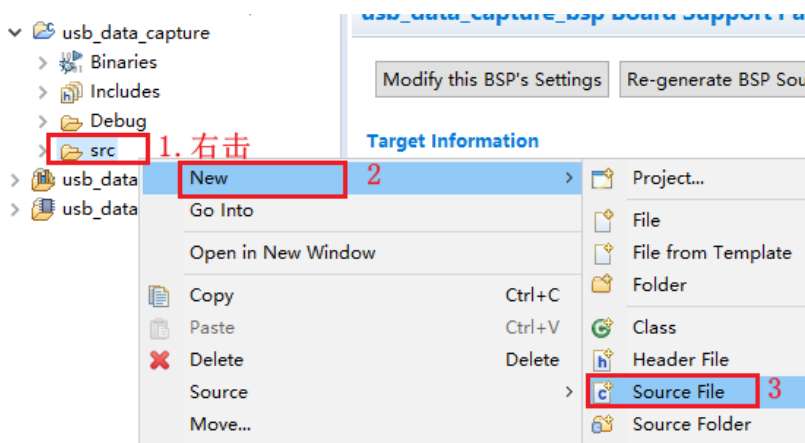


图 1-10 新建源文件

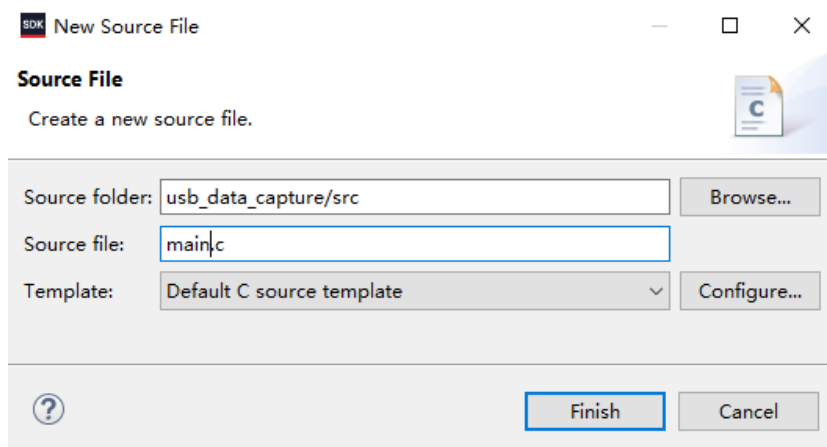


图 1-11 给源文件命名

5. 源文件中添加代码，如果不添加会报错，这里添加没有意义的一句代码，如下所示：

```
int main(void)
```

```
{  
    return 0;  
}
```

最后 `ctrl+b` 编译工程，没有报错，这样本次实验的设计就完成了，接下来便可以进行板级验证了，需要注意的是，如果你修改了 `vivado` 中的代码，你需要进行上面的步骤 1，导出 `bit` 文件，然后在 `SDK` 中重新编译工程，这样才算修改成功了。

1.4 板级验证

经过以上工作，代码设计部分的任务已经全部完成，接下来就可以进行板级验证了。本次实验的板级验证环节，主要验证：通过电脑上 `FX2_USB` 调试工具 `CyControl`，将命令帧进行发送，然后通过 `ACZ702` 开发板上的 `ACM68013` 模块接收，随后从 `USB` 下发的数据中解析出命令，最终实现对数据采集模块采样频率、数据采样个数以及采样通道的配置。配置完成之后，数据采集模块开始采集数据，将数据采集模块采集的数据通过 `USB` 传输到电脑。电脑端将接收到的数据以文件的形式进行保存，然后通过 `MATLAB` 进行进一步的分析。针对本次实验，我们也提供有专门的上位机软件，用户只需要在软件界面进行参数配置，便可以实时观察到数据波形变化，使用起来非常方便。

1.4.1 系统所需硬件

1. `ACZ702` 开发板一块
2. `USB` 线一根
3. `ACM68013` 模块一个
4. `Type-C` 下载线一根
5. 电源线一根（可选）

1.4.2 硬件连接

本次设计系统硬件连接如下图 1-12 所示：

1. 使用 `Type-C` 线连接开发板调试接口（靠近电源接口）和电脑 `USB` 口
2. 将开发板电源拨码开关拨到对应侧
3. `ACM68013` 模块连接至 40 pin 的排针上，靠右连接，1 脚和 1 脚对应

4. 使用 USB 线连接 ACM68013 模块和电脑

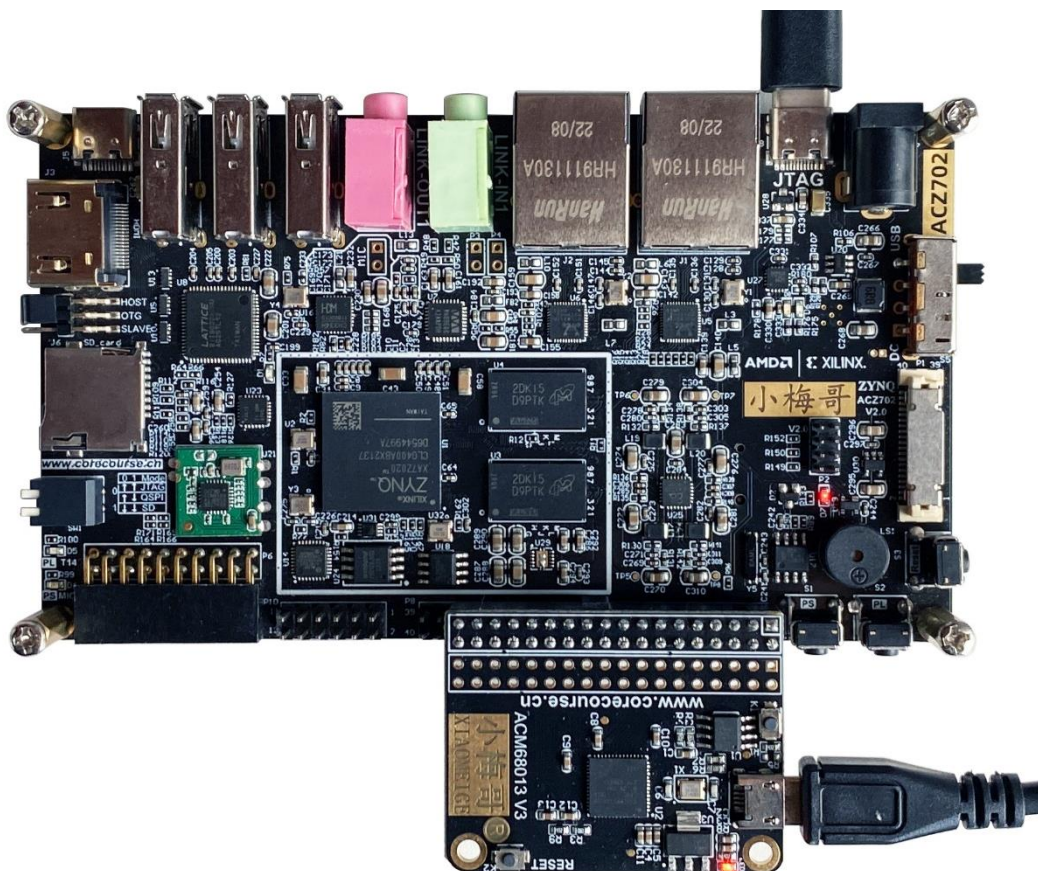


图 1-12 硬件连接图

1.4.3 烧录程序

在 SDK 软件中，依次点击 Run->Run Configurations，如下图 1-13 所示。

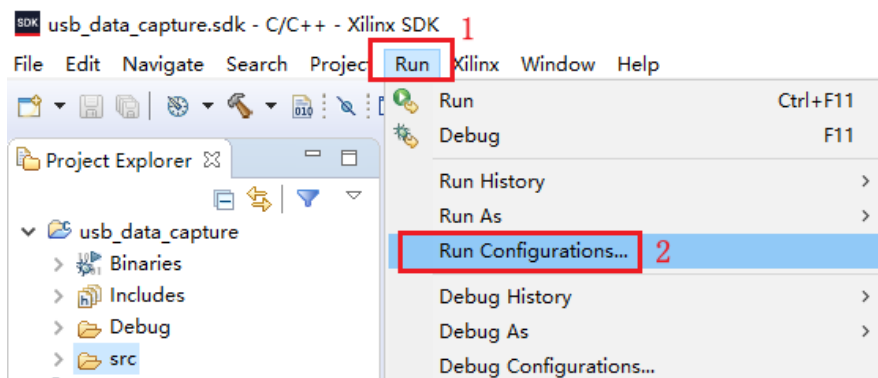


图 1-13 进入下载界面示意图

进入下载界面之后，下载 bit 文件，如下图 1-14 所示。

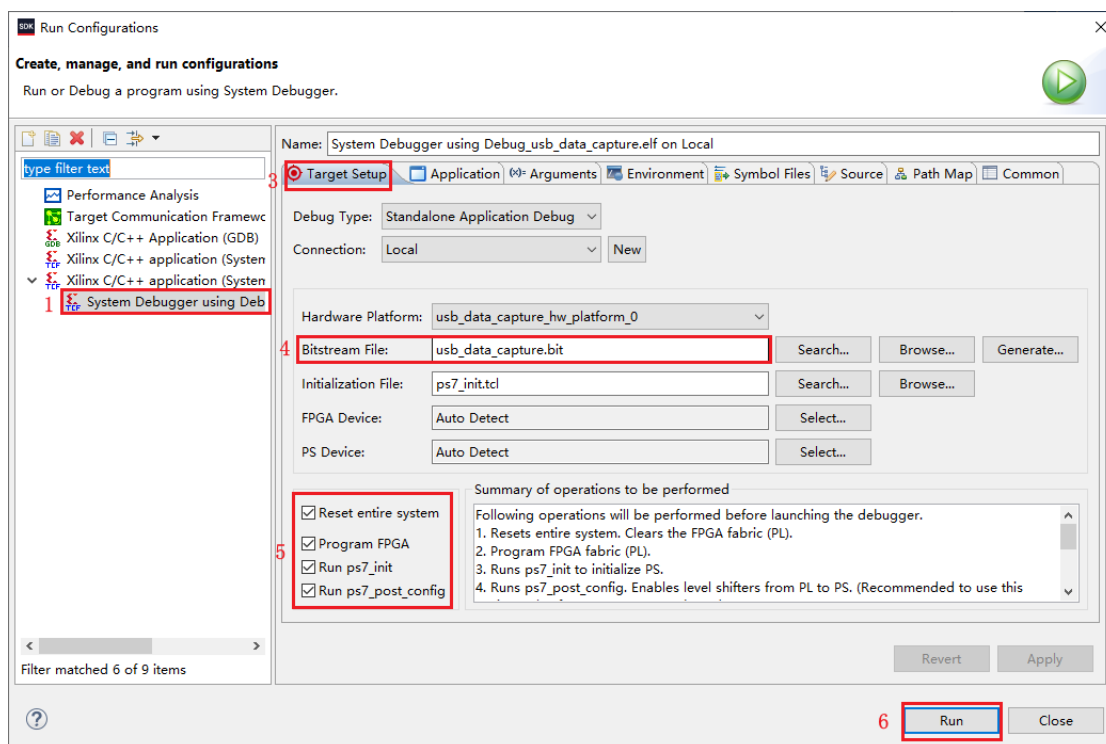


图 1-14 下载 bit 文件

下载成功之后，开发板右侧的 PL 侧的 LED 灯会被点亮，这时说明 PLL 锁相环工作正常，如下图 1-15 所示。

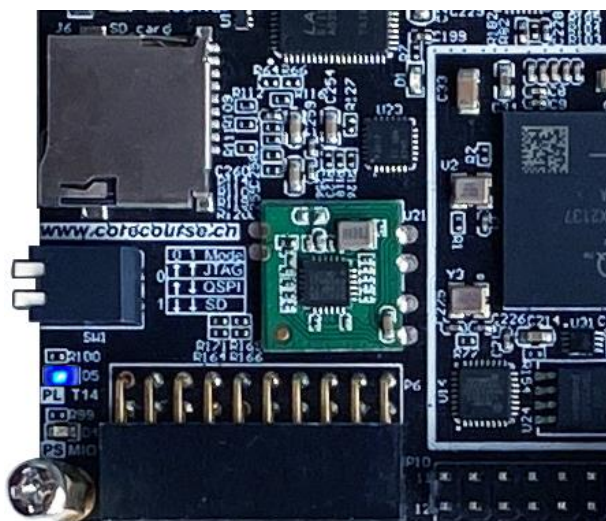


图 1-15 PL 侧 LED 灯被点亮

1.4.4 cypress 上位机数据通信

本节使用 cypress 上位机发送命令帧，并将接收的数据进行存储。读者首先在我们提供的 cypress 软件，然后双击打开软件。

打开软件之后，如果 USB 连接正常并且驱动安装成功，我们可以看到列表

框中有 “Cypress FX3 USB StreamerExample Device”，如下图 1-16 所示。

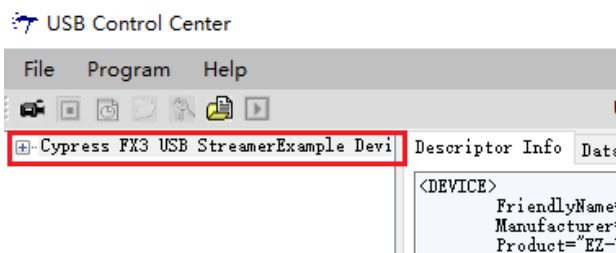


图 1-16 cypress 软件识别到 FX3

识别成功之后，我们使用该软件进行数据通信，软件的使用方式如下所示。

1. 点击 “Cypress FX3 USB StreamerExample Device” 前面的 “+” 找到 bulk out endpoint (0x02)。
2. 点击 “Data Transfers”。
3. 在 Data to Send 中输入指令串。在前面接收转命令模块中介绍到数据帧格式对数据采集模块的四个寄存器进行配置。例如以 1M 的采样速率，对 1 个通道进行采样（本次实验仅使用通道 1），共采集 16384 个数据，此时 Cypress 软件需要发送的指令串如下：

55A50200004000F055A50100000001F055A50300000031F055A50000000000F0

4. 点击 “Transfer Data-OUT” 进行命令传输，传输完成之后如下图 1-17 所示。

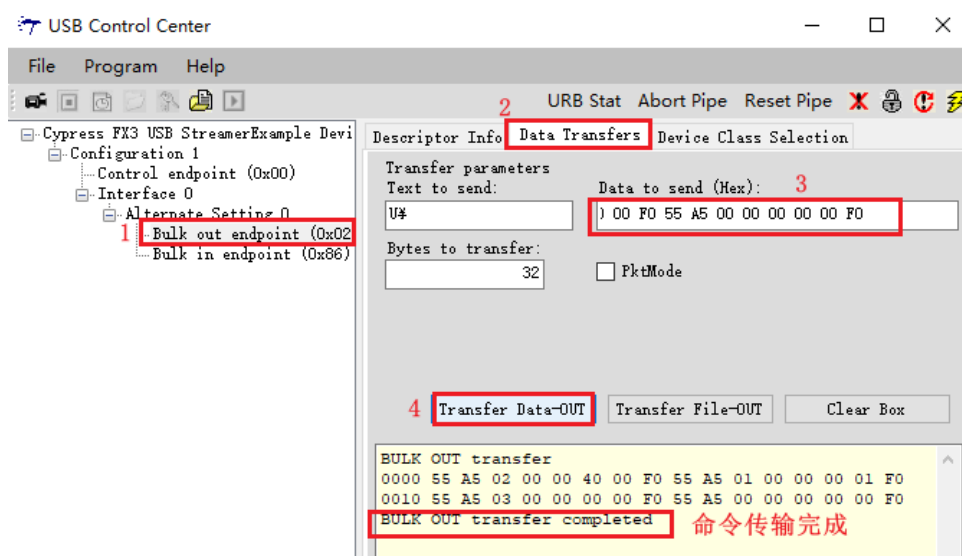


图 1-17 命令传输完成标志

5. 点击 “bulk in endpoint”。

6. 设置采样数量值，在“Bytes to transfer”一栏中设置需要的数据个数，本次实验采集的数据是 16 位的，而 cypress 软件是以字节为单位传输的，那么这里设置的数值应该是采样数量*2（16384*2=32768）。
7. 点击“Transfer File-IN”将采集到的数据以文件的形式保存，操作图 1-18 如下所示。

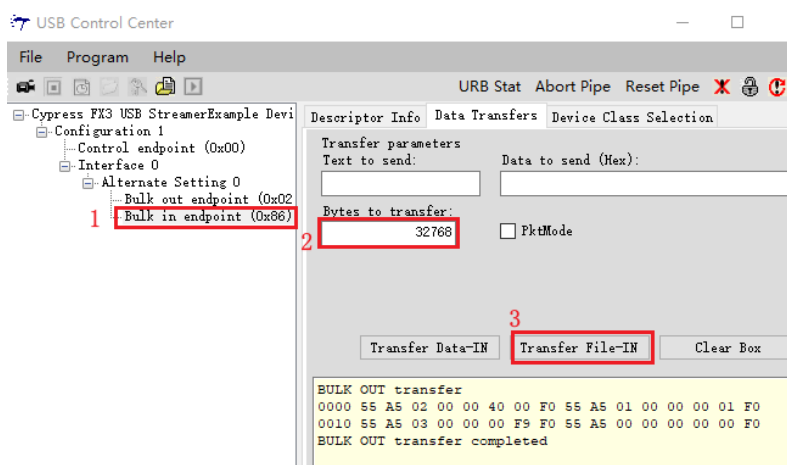


图 1-18 数据接收界面设置

8. 在弹出的文件保存界面中，读者需要设置文件保存的路径并给文件命名，比如我们这里保存在 F 盘并给文件命名为“usb_data”，然后点击保存。如下图 1-19 所示。

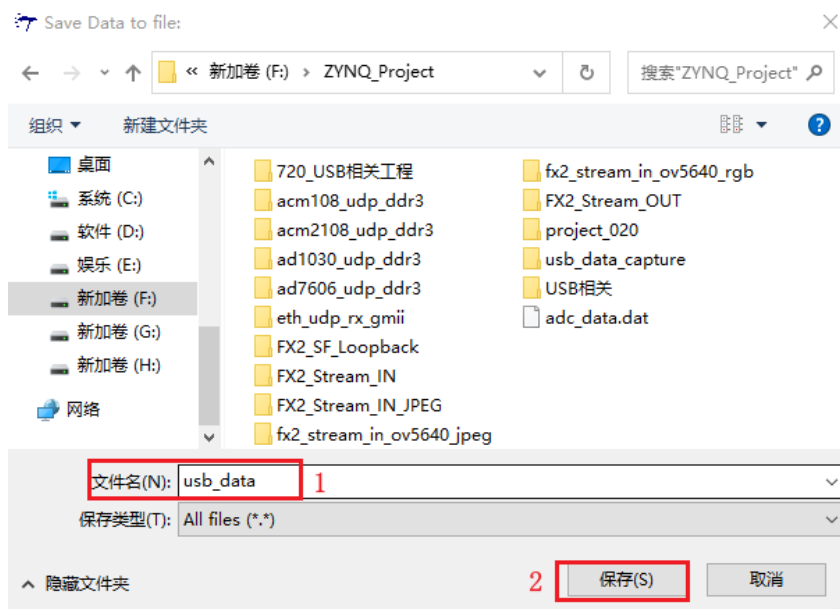


图 1-19 设置文件名称

9. Cypress 软件接收数据，并提示传输成功。如下图 1-20 所示。

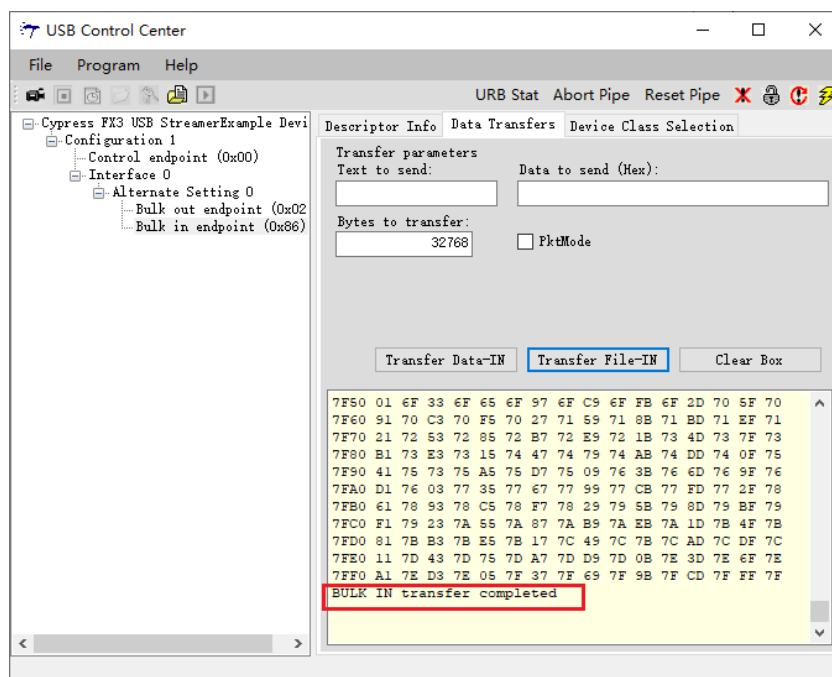


图 1-20 数据传输完成界面显示

10. 我们根据前面第 8 步设置的文件路径，找到数据文件，然后右击选中属性，查看文件大小，如下图 1-21 所示。从图中可以看出，文件大小为 32768 字节，符合我们之前设置的采样数量的大小。

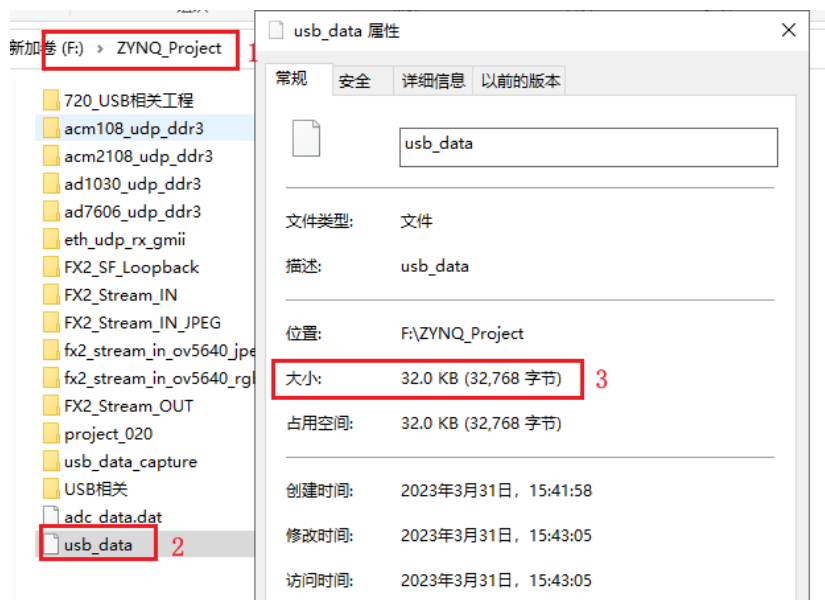


图 1-21 查看接收文件大小

通过上述步骤，我们就完成了一次数据采集，并将采集到的数据进行了保存，方便后续进行数据分析。在操作的过程中还需要注意以下几点：

1. 采样数量必须为 256 个 16bit 数的整数倍，即接收的 bytes to transfer，一

定是 512 个 8bit 数据的整数倍。如果操作失误要求读数据的数量大于指令码设置的写入数据故障，会出现输出框为 997 的故障码。这时候建议将开发板复位，清除上次不正常读写的数据。开发板复位，按下 S2 按钮即可，fx2 芯片复位，按一下 ACM68013 模块上的 RESET 按钮。

2. 如果是逐条发送指令，则必须确保启动指令在设置通道和设置采样数量指令之后发送，否则一旦发送采样启动指令，通道和采样数量设置将不会生效。

1.4.5 MATLAB 图像绘制

前面通过 Cypress 软件得到了 ADC 采集到的数据文件，我们需要对采集到的数据进行分析，本次实验使用 MATLAB 软件进行分析。使用 MATLAB 软件需要读者电脑安装了 MTALAB，如果已经安装好了 MTALAB 软件，则可以双击我们提供的 ADCdata_to_wave_v2_2.m 文件，在打开方式里选择以 MATLAB 打开，该文件位于工程文件文件夹下。文件打开之后，读者需要将代码中文件路径修改为你保存的数据文件路径，随后点击运行便可以直观的看到数据是否正确，MATLAB 操作如下图 1-22 所示，得到的波形图如下图 1-23 所示。

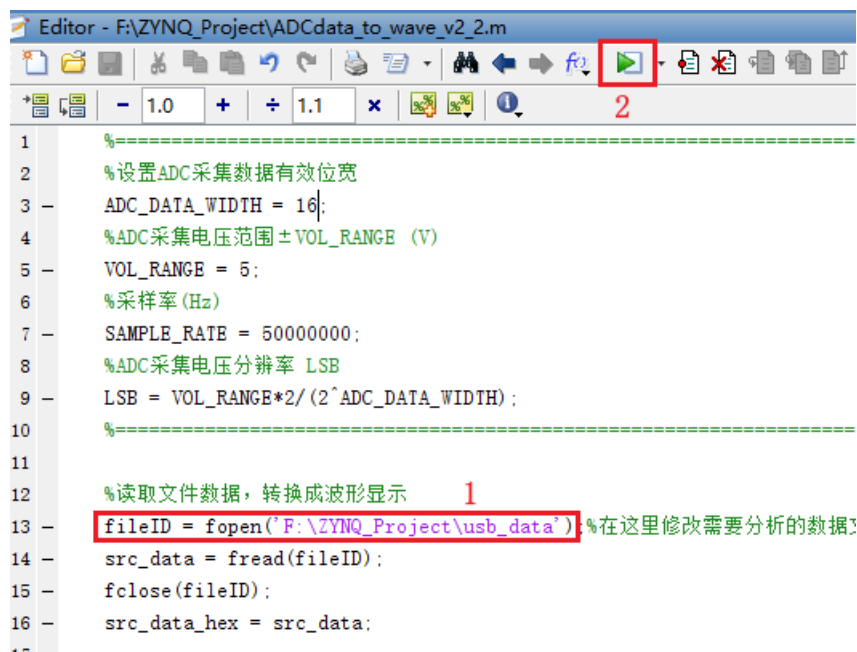


图 1-22 修改文件路径并运行

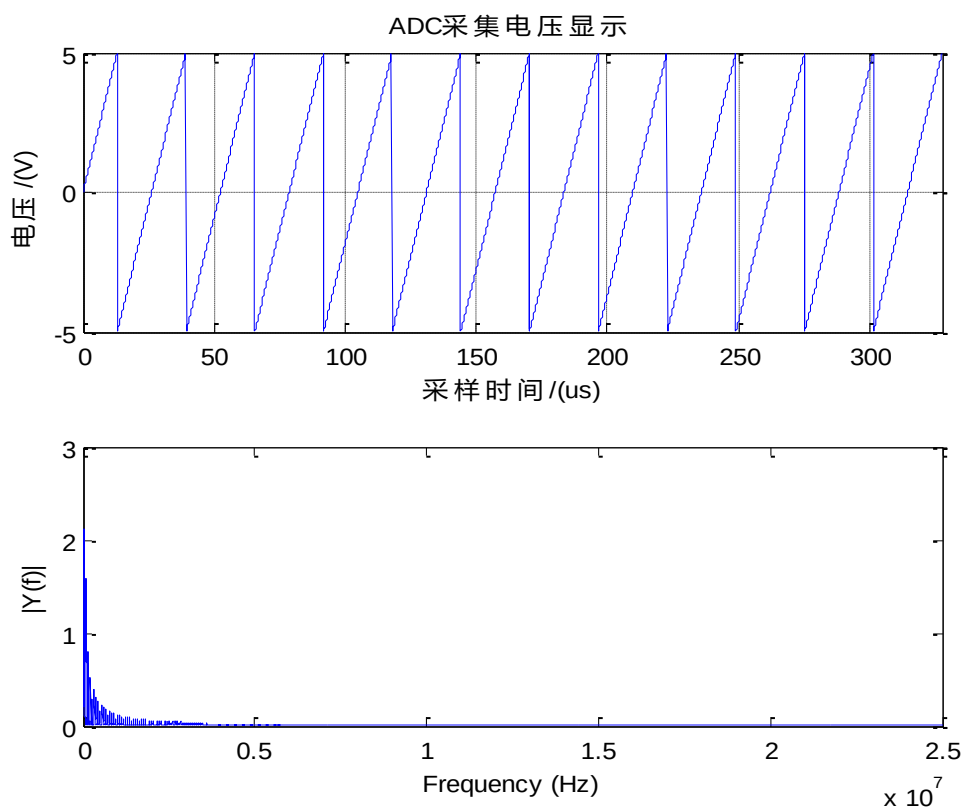


图 1-23 MATLAB 分析波形图

从上述图中可以看出，分析出的波形是一个三角波的波形，和我们设置的一致，说明实验成功。

1.4.6 数据采集上位机通信

前面通过 Cypress 软件采集数据时，每次保存数据都需要重新点击“Transfer File-IN”一栏，修改寄存器参数的时候，都需要重新计算，然后发送命令，修改之后也不能直接实时观察到数据波形，使用起来不是很方便。基于上述问题，我们设计了上位机软件“小梅哥控制台 For ADC 采集”进行数据采集，用户只需要在界面上对采样参数进行设置，便可以实时观测到数据变化，该软件的最新下载链接如下所示：

[数据采集上位机使用方法说明](#)

<http://www.corecourse.cn/forum.php?mod=viewthread&tid=29224>

双击上位机软件，初始界面如下图 1-24 所示。

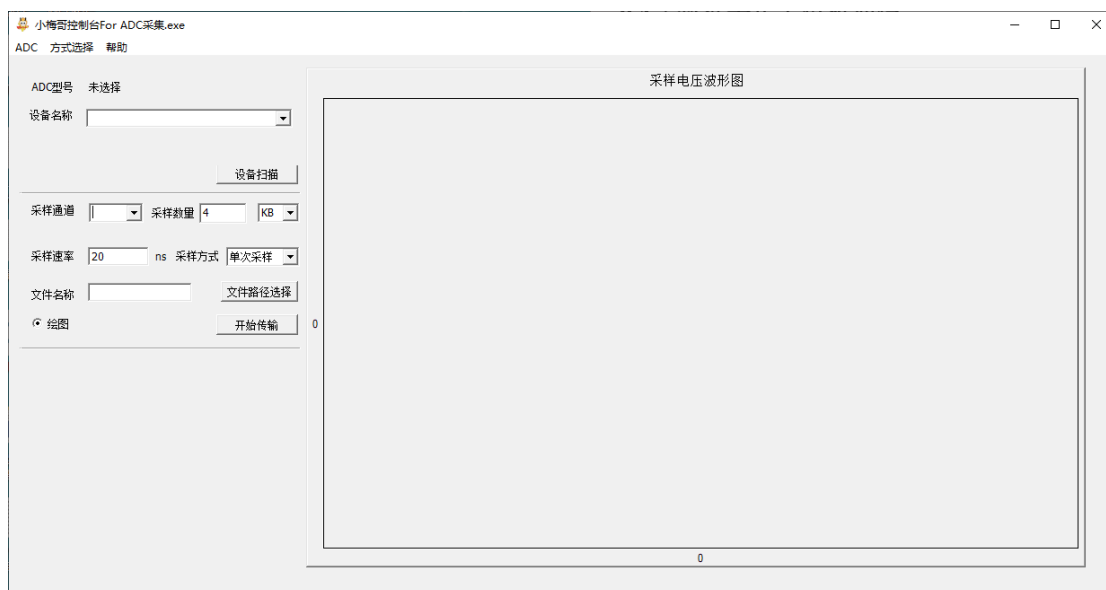


图 1-24 上位机软件初始界面显示

本次实验使用该软件的方式如下所示：

1. 点击 ADC，因为模拟的有效数据位宽是 16 位的，所以这里选择 ACM7606。
2. 点击方式，选择 USB，将会检测 USB 设备，设备检测成功之后，设备名称将会显示“(0x04B4 - 0x00F1) Cypress FX3 USB StreamerExample”。
3. 选择完成之后，可以看到对采样通道、采样数量等都已经设置了初始值，用户可以根据自己的需求进行修改。

点击开始传输之后，可以看到在右边采样电压波形图界面可以直观看到波形图，如下图 1-25 所示。需要注意的是波形图的横坐标对应的不是频率，而是采样数量。

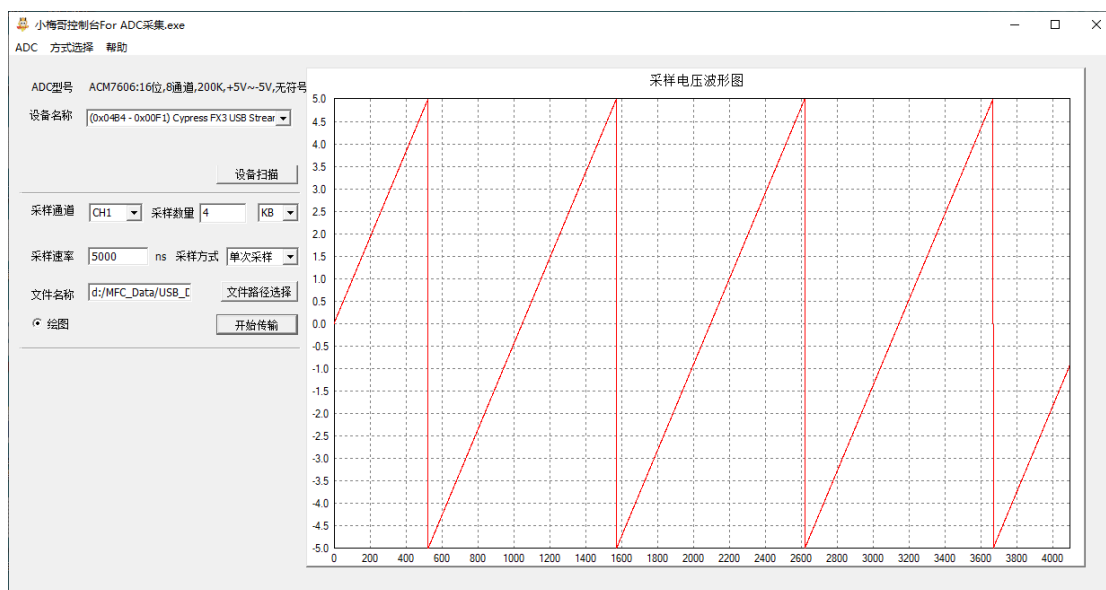


图 1-25 数据采集上位机显示图 1

从上述图中可以看出采样波形图是一个三角波的波形，符合实验中我们模拟给的数据波形，再次点击开始传输可以看到波形会发生变化，如下图 1-26 所示。这是因为在 FPGA 内部我们模拟产生了两种波形，如果选择循环采样可以看到明显的波形收缩的效果。

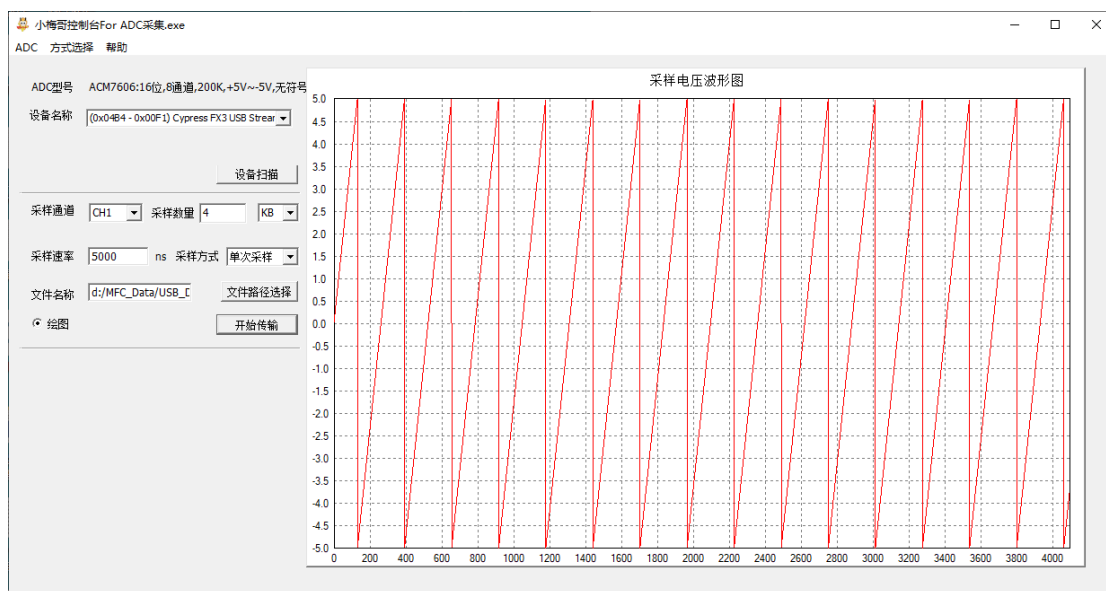


图 1-26 数据采集上位机显示图 2

1.5 思考与总结

本次实验介绍了基于 ACM68013 模块的数据采集实验，用户通过 FX2_USB 数据采集上位机工具 CyControl 向开发板发送指令数据配置数据采集模块的四个

寄存器，数据采集采样完成之后，将采集到的数据存储至 DDR3 中，最后将 DDR3 中的数据通过 USB 传输至 PC 机，电脑端通过 CyControl 软件将 USB 传输过来的数据进行保存，最终通过 MATLAB 对数据进行进一步的分析。如果使用我们提供的上位机软件，则不需要自己设置命令，只需要在界面上修改相关参数，便可以在右边的波形显示界面实时观察到波形变化。

本节实验使用 USB 实现数据的传输，关于 USB 通信以及驱动的安装和固件的烧写请读者自行查看 ACM68013 模块配套的资料内容：

[【产品资料】【扩展模块】ACM68013 USB2.0 模块资料和使用说明](#)

<http://www.corecourse.cn/forum.php?mod=viewthread&tid=28528>