1 USB 应用之传输 AD7606 数据采集

导读

本节实验结合 ADI 公司的 16 位 8 通道并行采样 ADC 芯片 AD7606，并利用 AC608_7Z010 开发板上外接一个 USB 模块 ACM68013，实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并传输到电脑。FPGA 从 USB 下发的数据中解析出命令，最终实现对 AD7606 的采样频率、采样个数以及采样通道的合理配置。配置完成之后，AD7606 开始采集数据，并将 AD7606 采集的数据存储进 DDR3 中。再由 USB 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。

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 AC608_7Z010 开发板上外接的一个 USB 模块 ACM68013 接收数据，随后从 USB 下发的数据中解析出命令，最终实现对 AD7606 采样频率、采样个数以及采样通道的配置。配置完成之后，AD7606 开始采集数据，并将采集的数据存储进 PS 侧的 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。

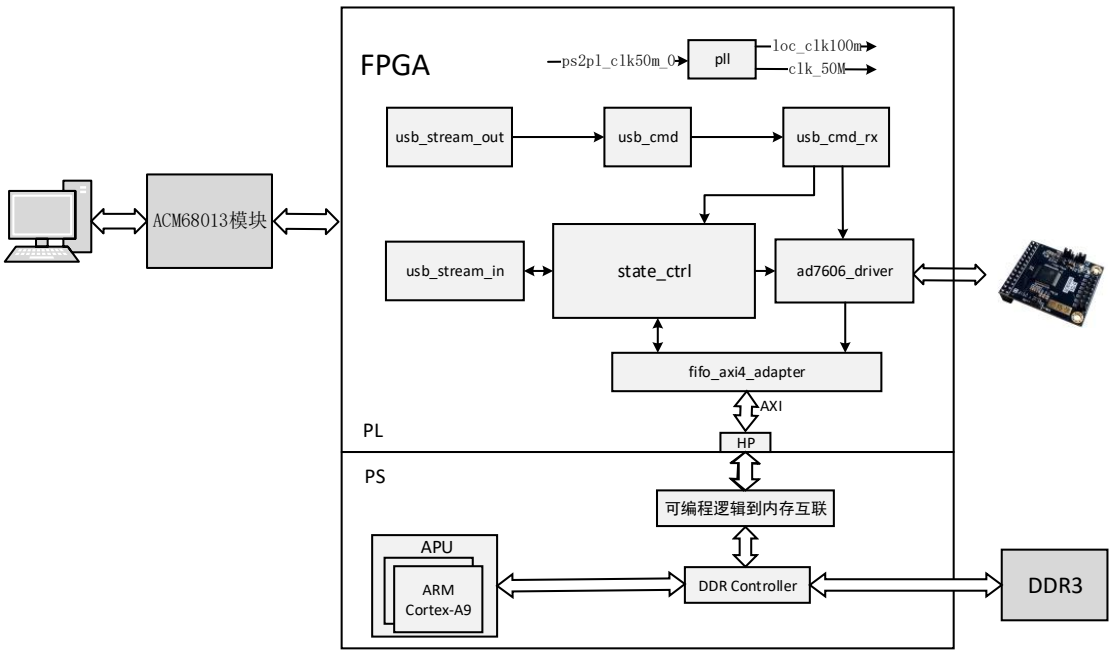

图 1-1 USB 应用之传输 AD7606 数据采集整体设计框图

本章实验仅针对于上述图中的 PL 部分的代码设计进行讲解，PS 部分的内容请参看：

【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

对于 PS 部分各个模块的功能介绍如下：

1. PLL 模块：锁相环模块，生成本次实验每个模块所需要的工作时钟，其中输入时钟为 50M 的系统时钟，由 PS 输出给 PL，输出 100M 的时钟给到 DDR3 控制器使用，输出 50M 的时钟给其他模块使用。
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. ad7606_driver 模块：AD7606 控制器驱动模块，该控制器实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出。使用该控制器时，用户无需关心 AD7606 的具体控制时序，一切都在控制器内部完成，用户只需要像使用并行 ADC 一样取用数据即可。

6. 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。
7. 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。
8. fifo_axi4_adapter 模块：fifo 接口到 AXI4 接口的转换模块(含 2 个 FIFO)。

1.2 ACM7606 模块简介

ACM7606 数据采集模块使用的是 ADI 公司的 16 位 8 通道同步采样模数转换器 AD7606，模块图如下图 1-2 所示。

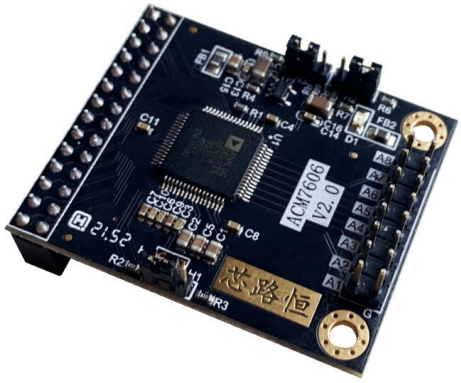

图 1-2 AD7606 模块图

AD7606 是 16 位 8 通道同步采样模数数据采集系统（DAS）。内置模拟输入箝位保护、二阶抗混叠滤波器、跟踪保持放大器、16 位电荷再分配逐次逼近型模数转换器（ADC）、灵活的数字滤波器、2.5V 基准电压源、基准电压缓冲以及高速串行和并行接口。AD7606 采用 5V 单电源供电，可以处理±10V 和±5V 真双极性输入信号。同时所有通道均能以高达 200kSPS 的吞吐速率采样。输入箝位保护电路可以耐受最高达±16.5V 的电压。无论以何种采样频率工作，其模拟输入阻抗均为 1MΩ。采用单电源工作方式，具有片内滤波和高输入阻抗，因此无需驱动运算放大器和外部双极性电源。AD7606 抗混叠滤波器的 3dB 截止频率为 22kHz；当采样频率为 200Ksps 时，它具有 40dB 的抗混叠抑制特性。

芯片对外提供 SPI 和并行的数字接口。当 AD7606 的 8 个通道全部以 200KPS 的最高速率进行转换时，数据输出速率达到 25.6Mbps，需要使用高性能 MCU 的 SPI 外设才能勉强该速率要求。因此可以使用 16 位并口来进行数据的传输，提高数据传输速率。当 AD7606 应用在 FPGA 系统中的时候，使用 SPI 串行接口和并行接口都能够轻松的满足数据传输的速率需求。当在 FPGA 系统上应用

AD7606 时，可以通过在 FPGA 上设计 AD7606 控制转换逻辑，将转换结果数据直接存储到片上的存储器如 FIFO 或者 RAM 中，也可以存储到 FPGA 片外的存储器如 SRAM 或 SDRAM 中，然后由其他主控芯片如 MCU 或 DSP 读出，或者直接在 FPGA 内部进行数据的运算和处理。当然，由于 FPGA 片上可以设计软核控制器，也可以直接使用软核控制器完成数据的处理和传输工作。

1.2.1 功能框图

AD7606 的功能框图如下图 1-3 所示：

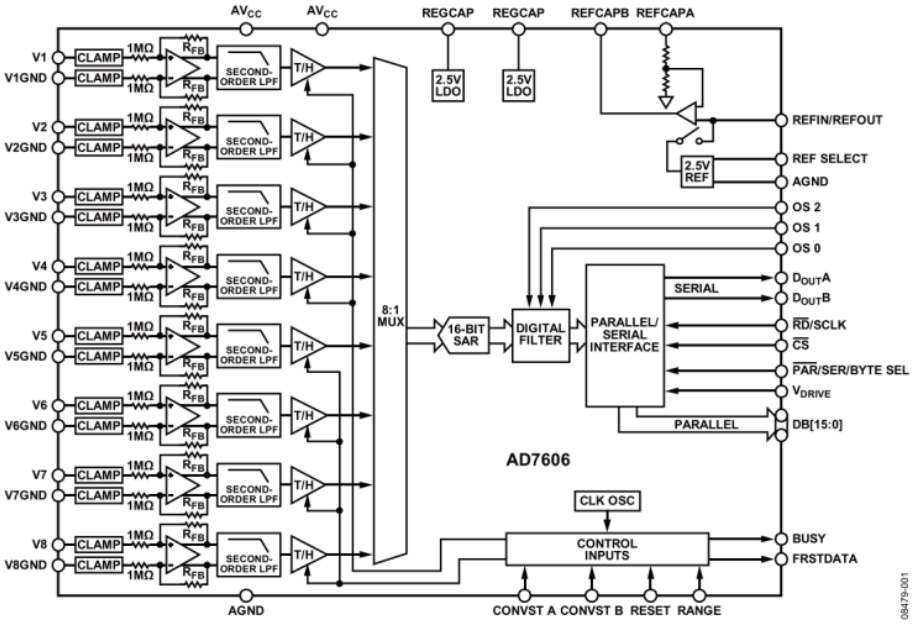

图 1-3 AD7606 功能框图

如上图所示，采集到的数据在经过稳压滤波和采样保持后通过 8 选 1 多路选择器被分别送入到 16 位逐次逼近型 ADC 芯片 AD7606 中进行转换，最后经由数字滤波后输出，当数据以串行模式输出时，数据会从 DoutA、DoutB 中输出，如果数据是以并行方式输出，那么数据将从 DB[15:0]中输出。

1.2.2 模拟输入

AD7606 可处理真双极性、单端输入电压。RANGE 引脚的逻辑电平决定所有模拟输入通道的模拟输入范围。如果此引脚与逻辑高电平相连，则所有通道的模拟输入范围为±10V。如果此引脚与逻辑低电平相连，则所有通道的模拟输入范围为±5V。AD7606 的模拟输入阻抗为 1MΩ。这是固定输入阻抗，不随 AD7606 采样频率而变化。AD7606 的输入结构如下图 1-4 所示，其各路模拟输

入均含有箝位保护电路。虽然采用 5V 单电源供电，但此模拟输入箝位保护允许输入过压达到±16.5V。

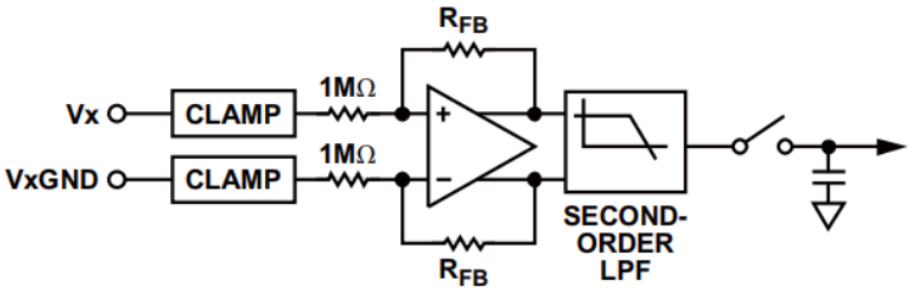

图 1-4 AD7606 模拟输入结构

1.2.3 数字滤波器与过采样

AD7606 内置一个可选的数字一阶 sinc 滤波器，在使用较低吞吐率或需要更高信噪比或更宽动态范围的应用中，应使用该滤波器。数字滤波器的过采样引脚 OS[2:0]控制（具体参考 AD7606 数据手册）。OS2 为 MSB 控制位。OS0 为 LSB 控制位，下表 1-1 提供了用来选择不同过采样倍率的过采样位解码。

表 1-1 不同过采样倍率的过采样位解码

OS[2:0]	过采样倍率	5V 范围 SNR(dB)	10V 范围 SNR(dB)	5V 范围 3dB 带宽(kHz)	10V 范围 3dB 带宽(kHz)	最大吞吐量 CONVST 频率(kHz)
000	No OS	89	90	15	22	200
001	2	91.2	92	15	22	100
010	4	92.6	93.6	13.7	18.5	50
011	8	94.2	95	10.3	11.9	25
100	16	95.5	96	6	6	12.5
101	32	96.4	96.7	3	3	6.25
110	64	96.9	97	1.5	1.5	3.125
111	无效					

OS 引脚在 BUSY 下降沿锁存，从而设置下一个转换的过采样倍率，如下图 1-5 所示，如果 OS 引脚选择过采样倍率 8，则下一个 CONVST x 上升沿采集各通道的第一个采样点，一个内部产生的采样信号采集所有通道的其余 7 个样点，然后对这些样点求平均值，以改进 SNR 性能。开启过采样时，CONVST A 和 CONVST B 引脚必须连在一起驱动，转换过程中 BUSY 保持高电平的时间会延长。BUSY 保持高电平的实际时间取决于所选的过采样倍率，过采样倍率越高，则 BUSY 保持高电平的时间或总转换时间越长。

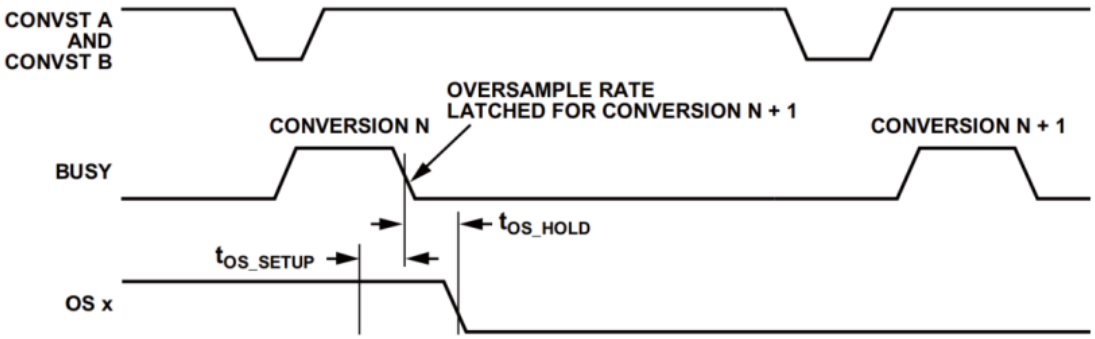

图 1-5 OS 引脚在 BUSY 下降沿锁存时序示意图

1.2.4 工作时序

AD7606 根据采样方式不同具有多种驱动时序，本次实验采用的为并行输出（即 8 个 16 位的数据通过 16 根并行线一个接着一个输出），转换后读取模式。其时序图由两部分组成：完成 AD 转换和读取 AD 数据。其中的时间可以参考 ADI 公司的手册。当 CONVST A 和 CONVST B 通道都变为上升沿时，BUSY 信号转变为高电平，代表转换开始，直到 BUSY 的下降沿到来，代表数据已经转换完成，正在锁存至输出数据寄存器中，当$\overline{CS}$变为下降沿时，数据将会被输送到总线上。并行工作模式下，当$\overline{CS}$和$\overline{RD}$都为低电平时，会使能总线，将转换结果输出到并行数据总线上，当 V1 转换结果开始输出之后，FRSTDATA 会随后转变为高电平，表示输出数据总线可以提供 V1 的结果。并行模式下，每次数据的输出为 16 位，对应一个通道。

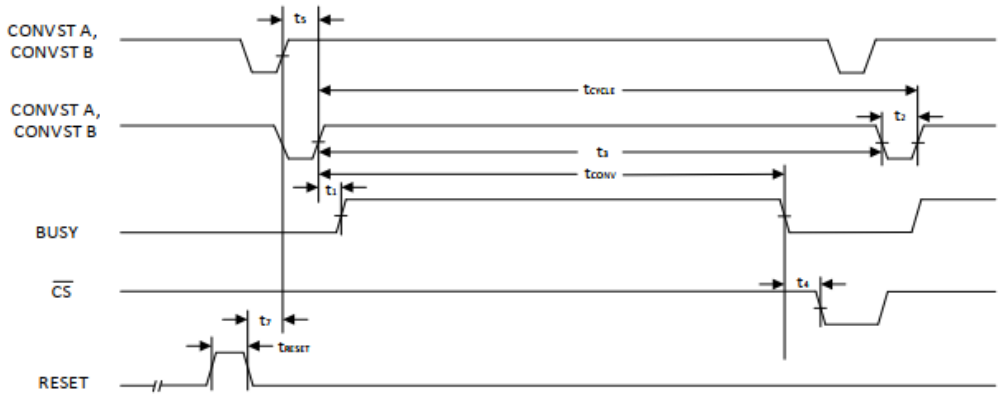

图 1-6 CONVST 时序—转换之后读取

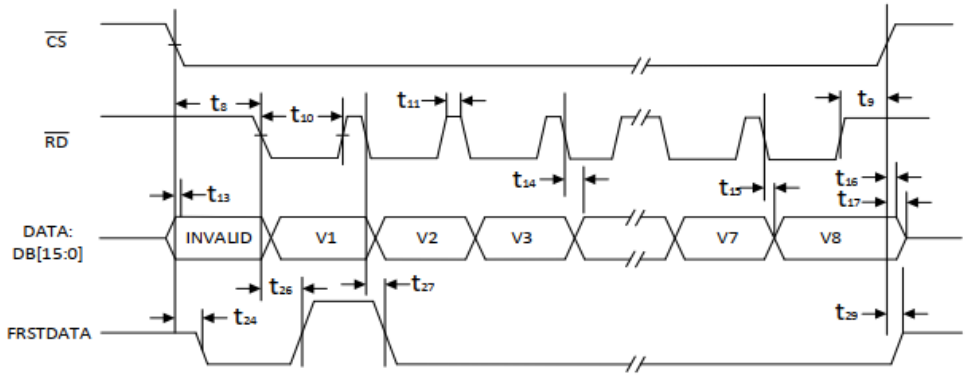

图 1-7 并行模式，独立的CS和RD脉冲

1.3 模块设计

下面给将对本次实验需要设计的模块进行介绍。

1.3.1 USB 数据流接收控制模块

USB 数据流接收控制模块（usb_stream_out）的功能是不断地将端点 2 中的数据读取出来，数据读取后直接作为端口输出，该模块的接口图如下图 1-8 所示。

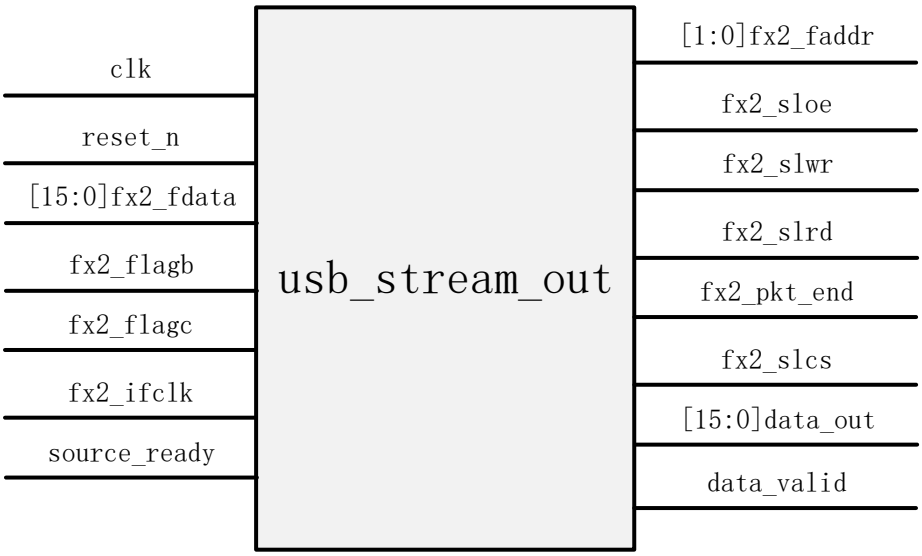

图 1-8 USB_Stream_OUT 模块接口图

该模块的接口信号说明如下表 1-2 所示：

表 1-2 usb_stream_out 模块接口信号表

信号名称	I/O	信号意义
clk	I	时钟信号
reset_n	I	复位信号，低电平有效
fx2_fdata[15:0]	I	FX2 型 USB2.0 芯片的 SlaveFIFO 的数据线

fx2_flagb	I	FX2 型 USB2.0 芯片的端点 2 空标志
fx2_flagc	I	FX2 型 USB2.0 芯片的端点 6 满标志
fx2_ifclk	I	FX2 型 USB2.0 芯片的接口时钟信号
source_ready	I	外部数据消费者数据接收允许信号，例如 FPGA 中的缓存 FIFO 中有足够的空间存储一帧 USB 数据，则允许从 Slave FIFO 中去读取数据
fx2_faddr[1:0]	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的 FIFO 地址线
fx2_sloe	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的输出使能信号，低电平有效
fx2_slwr	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的写控制信号，低电平有效
fx2_slrd	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的读控制信号，低电平有效
fx2_pkt_end	O	数据包结束标志信号
fx2_slcs	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的片选信号，当 SLCS 输出高时，不可进行数据传输
data_out	O	经过 FPGA 接收了的 USB 数据
data_valid	O	经过 FPGA 接收了的 USB 数据有效标志信号

上表中以 FX2 开头的信号，都是与 FX2 进行连接的信号，实现对 FX2 的 SlaveFIFO 读操作。data_out 和 data_valid 是读取到的数据与数据有效信号，使用这两个信号，可以非常方便地将从 FX2 中读取到的数据存储到 RAM、FIFO 中，以实现用户自定义应用。

同时，考虑到基于 FX2 和 FPGA 进行实际应用开发时，系统对 USB 的数据传输速率和传输时间有要求，所以必须在 FPGA 这边主动去控制整个通信的吞吐率。为此该逻辑特意设置了一个名为 source_ready 的信号，当该信号有效时，才允许从 SlaveFIFO 中读取一包数据。

我们使用状态机的方式实现该模块的功能，定义状态如下所示，分别为空闲状态、读状态、等待状态。

```
parameter [1:0] stream_out_idle        = 2'd0;
parameter [1:0] stream_out_read        = 2'd1;
parameter [1:0] stream_out_wait        = 2'd2;
```

整体的状态转移图如下图 1-9 所示。

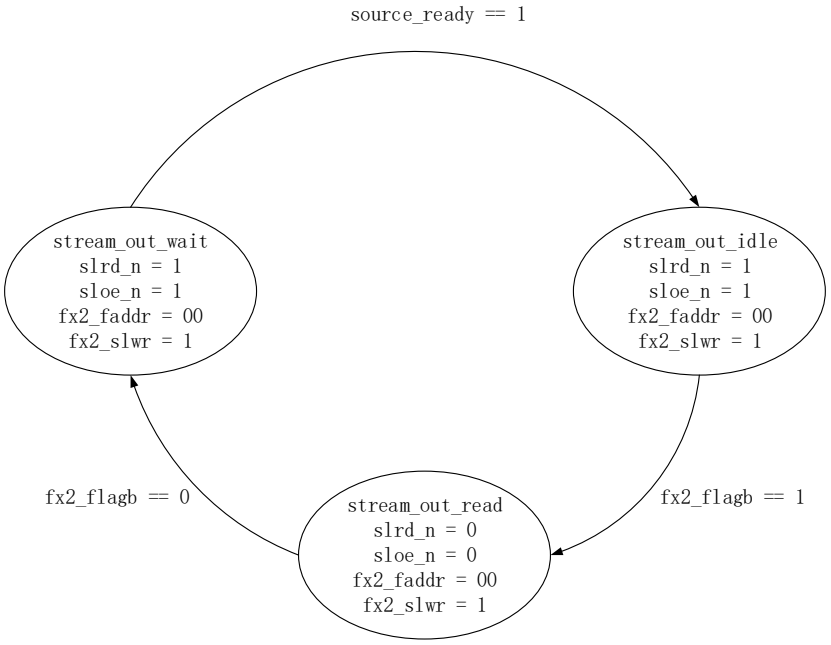

图 1-9 usb_stream_out 状态转移图

下面将对每个状态的代码设计进行简单的说明。

1. stream_out_wait 状态

在 stream_out_wait 状态的时候，等待数据使用逻辑允许从 USB 中读取新的数据，当 source_ready 信号为高电平的时候，也就是外部数据消费者数据允许接收数据，例如 FPGA 中的缓存 FIFO 中有足够的空间存储一帧 USB 数据，则允许从 Slave FIFO 中去读取数据，然后状态跳转至 stream_out_idle 状态等待读取数据，代码如下所示：

```
stream_out_wait:begin    //等待数据使用逻辑允许从 USB 中读取新的数据
    if(source_ready)
        next_stream_out_state = stream_out_idle;
    else
        next_stream_out_state = stream_out_wait;
end
```

2. stream_out_idle 状态

stream_out_idle 状态是指 slrd_n 和 sloe_n 均为高电平的闲置状态。只要端点 2 为空的标志 fx2_flagb 为低电平（被激活），状态机将处于 stream_out_idle 状态，fx2_flagb 变成高电平（端点 2 非空）之后，状态机将从 stream_out_idle 状态跳转至 stream_out_read 状态，开始读取数据，代码如下所示：

```
stream_out_idle:begin    //等待 FX2 的 Slave FIFO 中端点 2 非空
    if(fx2_flagb == 1'b1)
```

```
        next_stream_out_state = stream_out_read;
    else
        next_stream_out_state = stream_out_idle;
end
```

3. stream_out_read 状态

在 stream_out_read 状态中，FPGA 将连续从端点 2 的 FIFO 中读取数据。当 fx2_flagb 变成低电平，也就是端点 2 的 FIFO 为空之后，跳转回 stream_out_wait 状态，代码如下所示：

```
stream_out_read:begin    //从端点 2 中读取数据
    if(fx2_flagb == 1'b0)
        next_stream_out_state = stream_out_wait;
    else
        next_stream_out_state = stream_out_read;
end
```

在读取数据的时候，需要激活 slrd_n 信号和 sloe_n 信号才能成功读取数据，slrd_n 信号和 sloe_n 信号都是低电平有效，代码如下所示：

```
//产生读 Slave FIFO 数据请求信号
always@(*)begin
if((current_stream_out_state==stream_out_read)&(fx2_flagb==1'b1))
begin
        slrd_n = 1'b0;
        sloe_n = 1'b0;
    end else begin
        slrd_n = 1'b1;
        sloe_n = 1'b1;
    end
end
```

1.3.2 接收转命令模块

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，得到需要的指令数据传送给别的模块进行处理，该模块的结构框图如下图 1-10 所示。

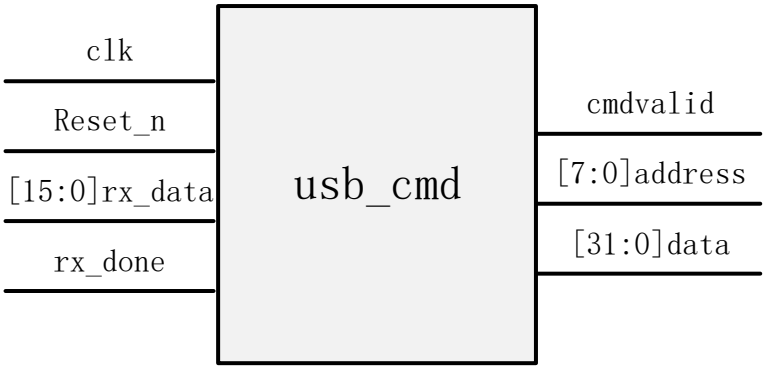

图 1-10 接收转命令模块框图

模块信号说明如下表 1-3 所示。

表 1-3 接收转命令模块信号说明表

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 AD7606 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 1-4 所示：

表 1-4 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

从上表中可以看出，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 为要写入寄存器的数据的 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 AD7606 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];
        cmdvalid <= #1 1;
    end
    else
        cmdvalid <= #1 0; end
else
    cmdvalid <= #1 0;
```

1.3.3 指令转控制模块

指令转控制模块（usb_cmd_rx）将从接收转命令模块（usb_cmd）接收到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 1-5 所示：

表 1-5 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，对应了 8 个通道的数据存储开关，如果某通道对应的设置为 1，则该通道的采样结果就会被存入 DDR 并通过 USB 发送，注意对应的 2 进制的位，不是 10 进制，比如设置通道 3，对应 01 地址需要写入的值为 04(100)，而不是 03(011)。
set_sample_num	2	32	数据个数寄存器，设定总共采集传输多少个数据。注意，该寄存器设置的是总共采集的数据个数， 假设设置采集 100 个数据，ChannelSel 为 0000_0011b， 则实际每个通道采样的次数就是 50，2 个通道的数据加起来是 100 个。假设设置采集 100 个数据，而且设置了 ChannelSel 为 0011_0011b，则实际每个通道采样的次数就是 25，4 个通道加起来采集 100 个数据
set_sample_speed	3	32	ADC 采样速率设置寄存器。该寄存器用来设置 ADC 每多久执行一

			次转换。 由于 ADC 的最大采样速率为 200Ksps，所以可以通过设置该寄存器的值来让 ADC 的采样速率在 1~200Ksps 范围内调整，以适应不同的应用场景。 ADC_Speed_Set=1000000000/20/speed-1，其中 speed 就是实际要设置的采样速率。

指令转控制模块的结构框图如下图 1-11 所示。

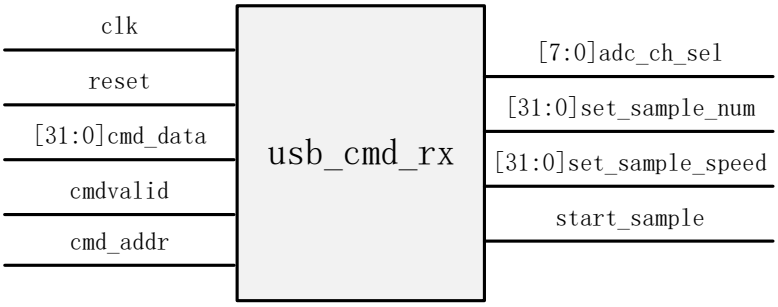

图 1-11 指令转控制模块结构框图

模块信号说明如下表 1-6 所示。

表 1-6 指令转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	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 [7:0]	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 1-5 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[7:0]；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，代码如下所示：

```
always@(posedge clk or posedge reset)
if(reset)begin
    adc_ch_sel <= 8'b1111_1111;
    set_sample_num <= 31'd256;
    start_sample <= 1'b0;
    set_sample_speed <= 32'd0;
end
else if(cmdvalid)begin
    case(cmd_addr)
        0: start_sample <= 1'b1;
        1: adc_ch_sel <= cmd_data[7:0];
        2: set_sample_num <= cmd_data[31:0];
```

```
        3: set_sample_speed <= cmd_data[31:0];
        4:
        begin
            adc_ch_sel <= cmd_data[7:0];
            set_sample_num <= cmd_data[23:8];
            start_sample <= 1'b1;
        end
    default:;
    endcase
end
else
    start_sample <= 1'b0;
```

1.3.4 AD7606 控制器驱动模块

AD7606 控制器驱动模块 ad7606_driver，该控制器实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出。使用该控制器时，用户无需关心 AD7606 的具体控制时序，一切都在控制器内部完成，用户只需要像使用并行 ADC 一样取用数据即可。该模块的结构框图如下图 1-12 所示。

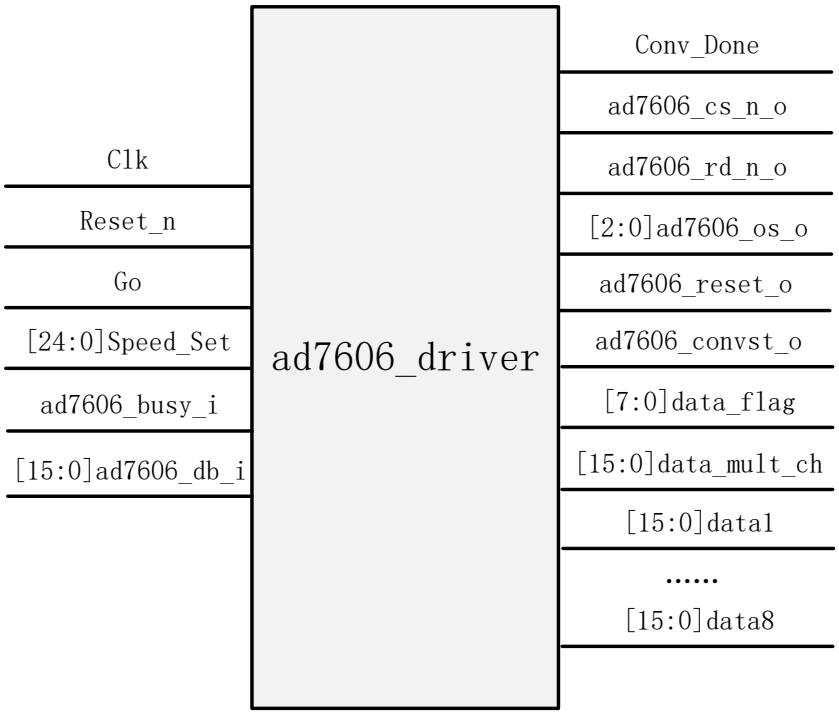

图 1-12 AD7606 控制器驱动模块结构框图

该控制器接口分为四个类，第一类为时序逻辑模工作所必须的时钟和复位信号（Clk、Reset_n），第二类是 AD7606 控制器与 AD7606 芯片管脚相连的各种功能控制和数据信号，第三类是设置控制器工作状态/工作数据的用户控制接口，第四类是控制器的结果输出接口。对于用户来说，只需要关注第三类和第

四类接口的使用，即可快速高效的使用该控制器来控制 AD7606 芯片完成数据转换。对于该模块的信号说明如下表 1-7 所示。

表 1-7 AD7606 控制器模块信号说明表

类	信号名称	I/O	信号意义
系统信号	Clk	I	模块系统时钟，为了让采样速率准确，要求为 50MHz
	Reset_n	I	模块复位信号，低电平复位
控制信号	Go	I	采样使能信号，为高电平就使能采样，低电平则在已经开始的一轮采样结束后，停止下一次采样
	Speed_Set[24:0]	I	采样速率控制端口，Speed_Set = 1000000000/20/speed - 1
	Conv_Done	O	一次采样完成标志信号，单时钟周期脉冲信号。每次 8 个通道结果都输出后，产生一个高脉冲信号
数据结果输出端口和标志信号	data_flag[7:0]	O	转换结果有效标志信号，因为 AD7606 有 8 个通道，转换结果是依次输出，并非同时的，所以设置 8 个 Flag 信号，每个通道的结果就绪之后，就产生一个 Flag 信号，通知外部可以取用。另外，如果只关心其中的部分通道，则只需要关心 data_flag 中对应的位即可
	data_mult_ch[15:0]	O	多通道数据输出端口，该通道 16 位，在不同的时刻，输出不同通道的转换结果，使用时，与 data_flag 信号配合，data_flag 的哪一位出现高脉冲，则代表当前 data_mult_ch 的值为该通道的转换结果。该端口设计的目的是用于往 FIFO、RAM 等存储器中存储结果时使用。
	data1[15:0]…data8[15:0]	O	8 个通道的采样结果输出端口，每个端口分别对应一个模拟通道的采样结果，使用时与 data_flag 信号配合，每当 data_flag 中的某一位为 1 时，则对应的通道上的 16 位采样结果已经就绪，可以使用。这些端口主要用于每个通道的数据需要分别应用的场合。
ADC 芯片控制信号	ad7606_busy_i	I	ad7606 转换状态标志信号，为高电平则表明 ad7606 当前仍处于转换状态，结果没有更新，如果此时读取，读取的结果就还是前一次的采样转换结果。需要待该信号变为低电平之后，再读取 ad7606 中的数据
	ad7606_db_i[15:0]	I	ad7606 的 16 位数据线，读取时，输出对应通道的转换结果
	ad7606_cs_n_o	O	ad7606 芯片选中控制信号，可以从 AD7606 中读取转换结果时，需要使该信号为低电平
	ad7606_rd_n_o	O	ad7606 转换结果读取信号，该信号的下降沿，AD7606 将特定通道的采样结果送到 16 位数据线上，供外部读取。外部可以在 rd_n 信号的上升沿读取 16 位数据线上的结果
	ad7606_os_o[2:0]	O	ad7606 过采样控制信号，使用过采样可以进一步提高 ad7606 的采样精度，使用过采样会降低 ad7606 的有效转换速率，关于过采样的功能和使用方法，可以参考 ad7606 的 datasheet，默认为 0，则表示不使用过采样。能够运行在最高的转换速率（200Ksps）
	ad7606_reset_o	O	ad7606 的复位信号，复位 ad7606 内部各个功能单元的工作状态

	ad7606_convst_o	O	ad7606 转换开始信号，该信号的上升沿启动 ad7606 内部的采样转换逻辑开始新一轮的采样转换

该控制器在工作时会根据主机的指令对采样频率进行修改，当信号转换完成后便对 ADC 写控制器发出 data_flag 信号，控制其将转换完成数据 data_mult_ch 写入 FIFO 或者 RAM 的同时，也指出了该信号来自哪个通道。通过这两个信号，我们可以实现让 ADC 以特定的采样速率多次采样一个或多个通道的数据。每当 data_flag 中任意一位为 1，则将 data_mult_ch 中的值写入 FIFO 或者 RAM 中。由于 FIFO 和 RAM 等存储器，只有一个数据输入接口，所以这个时候用一个 data_mult_ch 端口分时输出不同通道的采样结果，就比使用 data1~data8 这 8 个 16 位的数据端口分别输出各自通道的采样结果要方便。

例如，将通道 1、2、5、8 的采样结果存入 FIFO。就可以使用下面的写法：

```
module adc_wr_fifo(
  input Clk,
  input Reset_n,
  input [7:0]data_flag;
  input [15:0]data_mult_ch;
  output reg fifo_wrreq,
  output reg [15:0]fifo_data
);
always@(posedge Clk or negedge Reset_n)
if(!Reset_n)begin
  fifo_wrreq <= 0;
  fifo_data <= 0;
end
else if(data_flag == 8'b1001_0011)begin
  fifo_wrreq <= 1'd1;
  fifo_data <= data_mult_ch;
end
else begin
  fifo_wrreq <= 0;
  fifo_data <= fifo_data;
end
endmodule
```

1.3.5 state_ctrl 模块

fifo_axi4_adapter 模块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 1-13 所示：

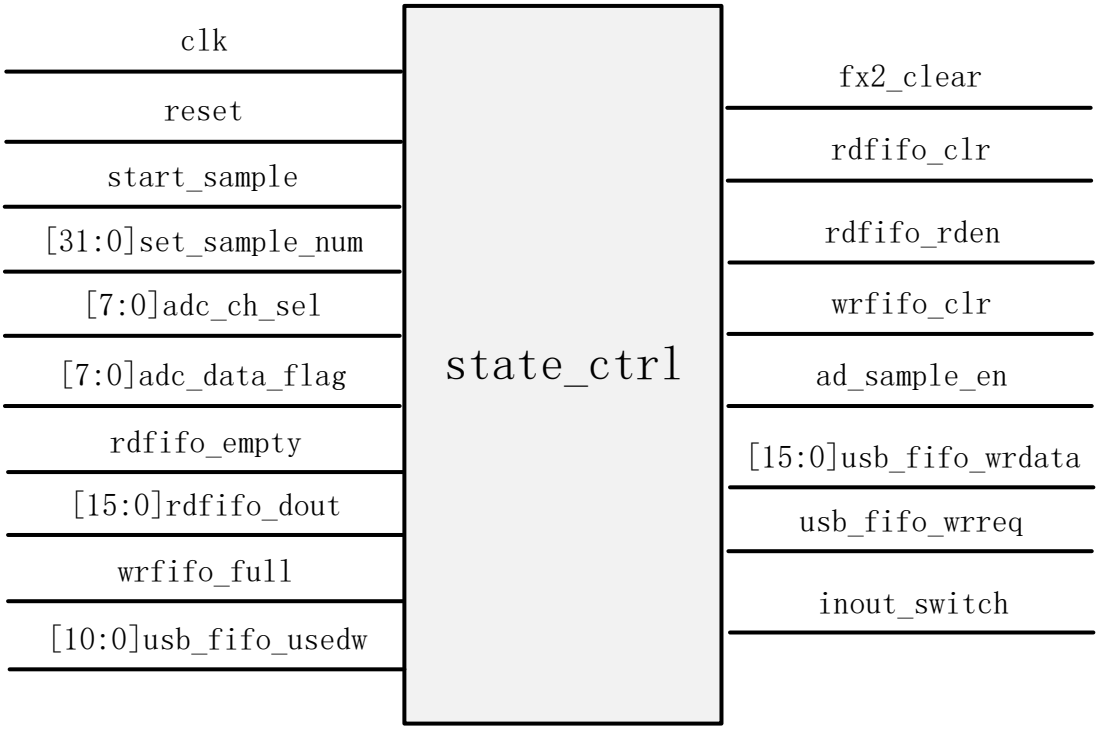

图 1-13 state_ctrl 模块结构框图

对于该模块的信号说明如下表 1-8 所示。

表 1-8 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
adc_ch_sel [7:0]	I	设置的 ADC 的采样通道
adc_data_flag [7:0]	I	ADC 模块转换结果有效标志信号
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
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_wrreq	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 ad7606_driver 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 模块产生对应通道数据输出标志信号（adc_data_flag & adc_ch_sel）与数据采集使能信号 ad_sample_en 信号相与得到；wrfifo_din 信号为 ADC 模块对应通道的输出数据信号 adc_data_mult_ch。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。定义状态如下表 1-9 所示。

表 1-9 state_ctrl 状态控制模块状态描述表

状态值	状态名称	状态意义
0	IDLE	空闲状态
1	DDR_WR_FIFO_CLEAR	DDR 写 FIFO 清除状态
2	ADC_SAMPLE	ADC 采样数据状态
3	DDR_RD_FIFO_CLEAR	DDR 读 FIFO 清除状态
4	RESET_USB	复位 USB 状态
5	DATA_SEND_START	数据发送状态
6	DATA_SEND_WORKING	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

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'b0;
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(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，每当 ADC 模块采集的数据有效并且是我们需要的通道产生的数据时（adc_data_flag& adc_ch_sel 有效），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_flag & adc_ch_sel)
        adc_sample_cnt<=adc_sample_cnt+1'b1;
    else
        adc_sample_cnt<=adc_sample_cnt;
end
else
    adc_sample_cnt<=32'd0;
```

当 adc_sample_cnt 达到设定的采样数据个数时，ADC 模块数据采集完成，跳转到 DDR 读 FIFO 清除状态，代码如下所示：

```
begin
    if(adc_sample_cnt>=set_sample_num)
        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'b0;
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(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 fifo_axi4_adapter 模块的读 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：

```
begin
    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
```

每个 send_data_cnt 在 rdfifo_rden 为 1 的状态下加 1，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从 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
```

1.3.6 fifo_axi4_adapter 模块

fifo 接口到 AXI4 接口的转换模块（fifo_axi4_adapter）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

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
```

该模块其它部分的设计和前面章节中一致，这里将不再进行说明，只需要将该模块例化进来使用即可，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_ddr1),
    //clock reset
    .clk                  (loc_clk100m         ),
    .reset                (reset               ),
    //wr_fifo Interface
    .wrfifo_clr           (wrfifo_clr          ),
    .wrfifo_clk           (clk_50M             ),
    .wrfifo_wren          ((adc_data_flag&adc_ch_sel)&&ad_sample_en ),
    .wrfifo_din           (adc_data_mult_ch    ),
    .wrfifo_full          (wrfifo_full         ),
    .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_awsize        (s_axi_awsize        ),
    .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       )
    );
```

1.3.7 USB 数据流发送模块

USB 数据流发送模块（usb_stream_in）主要用于实现 FPGA 通过 FX2 芯片发送数据给 PC，实现一些常见的输入型应用，比如本次实验 FPGA 实时采集 ADC 数据，然后经由 FX2 发送到 PC，再由 PC 进一步处理，在该模块内部添加了一个单时钟 FIFO IP 用于缓存需要通过 USB 传输的数据，FIFO 的深度设置为 1024，位宽为 16。usb_stream_in 模块的基本结构框图如下图 1-14 所示：

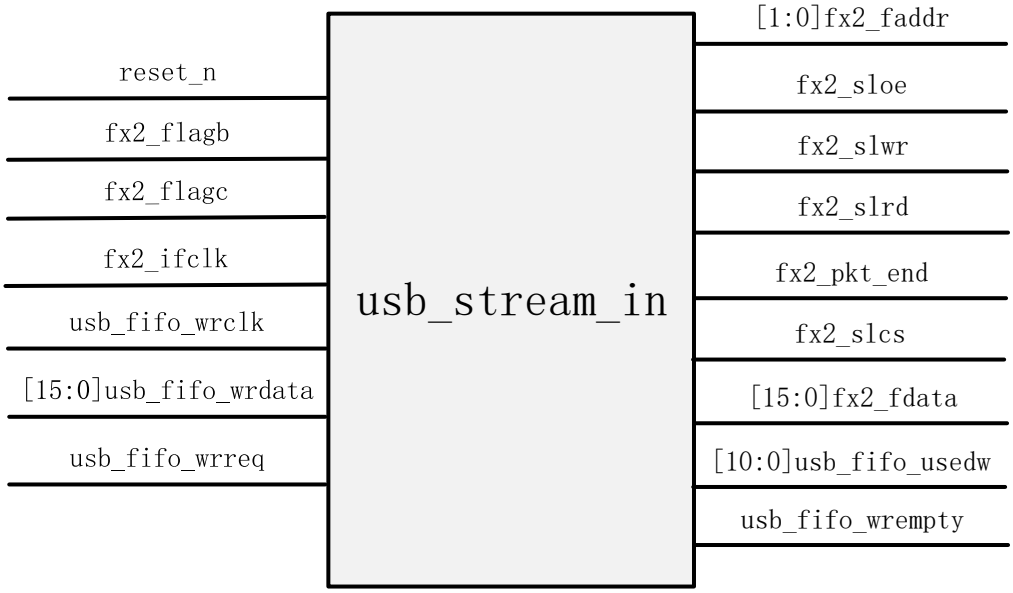

图 1-14 usb_stream_in 模块的基本结构框图

usb_stream_in 模块接口信号说明如下表 1-10 所示。

表 1-10 usb_stream_in 模块信号说明表

信号名称	I/O	信号意义
reset_n	I	复位信号，低电平有效
fx2_flagb	I	FX2 型 USB2.0 芯片的端点 2 空标志
fx2_flagc	I	FX2 型 USB2.0 芯片的端点 6 满标志
fx2_ifclk	I	FX2 型 USB2.0 芯片的接口时钟信号
usb_fifo_wrclk	I	usb_stream_in 模块内部 FIFO 的写时钟信号
usb_fifo_wrdata [15:0]	I	usb_stream_in 模块内部 FIFO 的写数据信号
usb_fifo_wrreq	I	usb_stream_in 模块内部 FIFO 的写使能信号
fx2_fdata[15:0]	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的数据线
fx2_faddr[1:0]	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的 FIFO 地址线

fx2_sloe	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的输出使能信号，低电平有效
fx2_slwr	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的写控制信号，低电平有效
fx2_slrd	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的读控制信号，低电平有效
fx2_clear	O	FX2 型 USB2.0 芯片的清除信号
fx2_pkt_end	O	数据包结束标志信号
fx2_slcs	O	FX2 型 USB2.0 芯片的 SlaveFIFO 的片选信号
usb_fifo_usedw[10:0]	O	usb_stream_in 模块内部 FIFO 的数据计数信号
usb_fifo_wrempty	O	usb_stream_in 模块内部 FIFO 为空的信号

usb_stream_in 模块功能的实现：FPGA 监控端点 6 的满标志（fx2_flagc），当 fx2_flagc 为高电平的时候，FPGA 会连续将数据写入到端点 6 的 FIFO 内，当 fx2_flagc 为低电平，也就是端点 6 的 FIFO 写满之后，FPGA 将暂停写操作。

本次设计我们可以通过状态机的方式实现 usb_stream_in 模块的功能，定义状态如下所示，分别是空闲状态和写状态。

状态转移图如下图 1-15 所示，下面我们对每个状态的代码设计和功能进行说明。

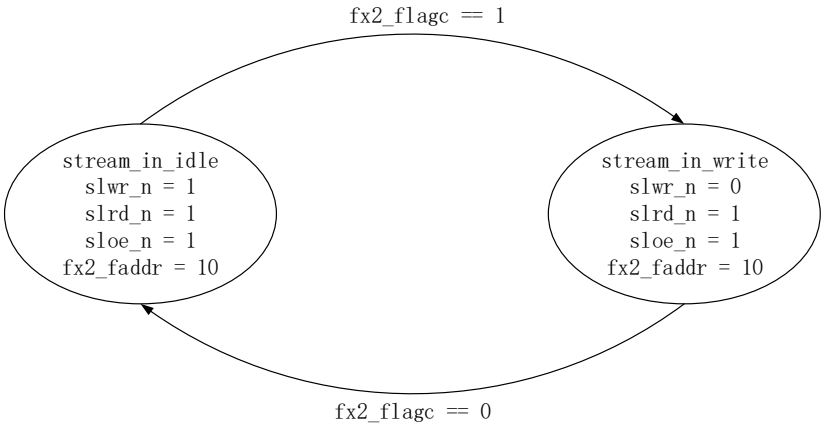

图 1-15 usb_stream_in 状态转移图

1. stream_in_idle 状态

stream_in_idle 是指 slwr_n 处在高电平时的闲置状态，只要端点 6 的满标志（fx2_flagc）并且 fifo 不为空时为低电平（被激活），则状态会处于 stream_in_idle 状态。slwr_n 信号产生的代码如下所示：

```
always@(*)begin
  if((current_stream_in_state==stream_in_write)&(fx2_flagc==1'b1)&(~empty))
    slwr_n <= 1'b0;
  else
    slwr_n <= 1'b1;
end
```

在 fx2_flagc 变成高电平的时候，代表此时端点 6 的 FIFO 还未写满，可以向 FIFO 中写入数据，状态机将从 stream_in_idle 状态跳转至 stream_in_write 状态，

代码如下所示：

```
stream_in_idle:begin
    if(fx2_flagc == 1'b1)
        next_stream_in_state = stream_in_write;
    else
        next_stream_in_state = stream_in_idle;
end
```

2. stream_in_write 状态

在 stream_in_write 状态中，将会激活 slwr_n 信号，FPGA 将会持续向端点 6 的 FIFO 中写入数据，在 fx2_flagc 变成低电平的时候（FIFO 被写满），状态机将返回到 stream_in_idle 状态，代码如下所示：

```
stream_in_write:begin
    if(fx2_flagc == 1'b0)
        next_stream_in_state = stream_in_idle;
    else
        next_stream_in_state = stream_in_write;
end
```

最后，例化我们添加的 FIFO IP，当~slwr_n 有效时，从 FIFO 中读出数据，并将数据交由 fx2_fdata 传输，如下所示：

```
fifo fifo(
    .rst (~reset_n),// input wire srst
    .wr_clk (usb_fifo_wrclk),            // input wire wr_clk
    .rd_clk (fx2_ifclk),            // input wire rd_clk
    .din (usb_fifo_wrdata),// input wire [15 : 0] din
    .wr_en (usb_fifo_wrreq),// input wire wr_en
    .rd_en (~slwr_n),// input wire rd_en
    .dout (data_out1),// output wire [15 : 0] dout
    .full ( ),// output wire full
    .empty (empty),// output wire empty
    .wr_data_count (usb_fifo_usedw),// output wire [10 : 0] data_count
    .wr_rst_busy ( ),  // output wire wr_rst_busy
    .rd_rst_busy ( )  // output wire rd_rst_busy
);
    assign fx2_fdata[15:0] = data_out1[15:0];
```

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，然后我们便可以进行板级验证了。

1.4 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 1-16 所示。

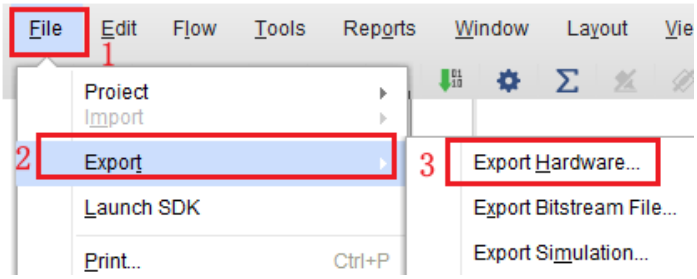

图 1-16 导出 bit 文件

2. 进入 SDK 软件，点击 File->Launch SDK，如下图 1-17 所示。

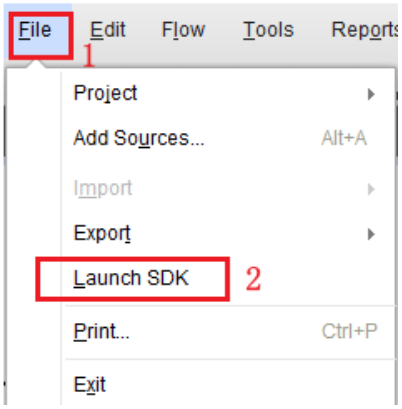

图 1-17 进入 SDK 软件

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 1-18 所示，然后给工程命名，比如命名为 ad7606_ddr3_usb，然后点击 Next，选择 Empty Application，就完成了工程的创建。

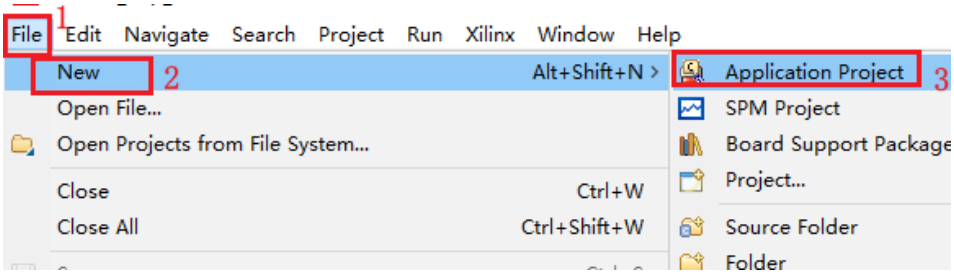

图 1-18 新建 SDK 工程

4. src 文件夹下，新建一个源文件，如下图 1-19 所示，然后在弹出的界面中给文件命名为 main.c。

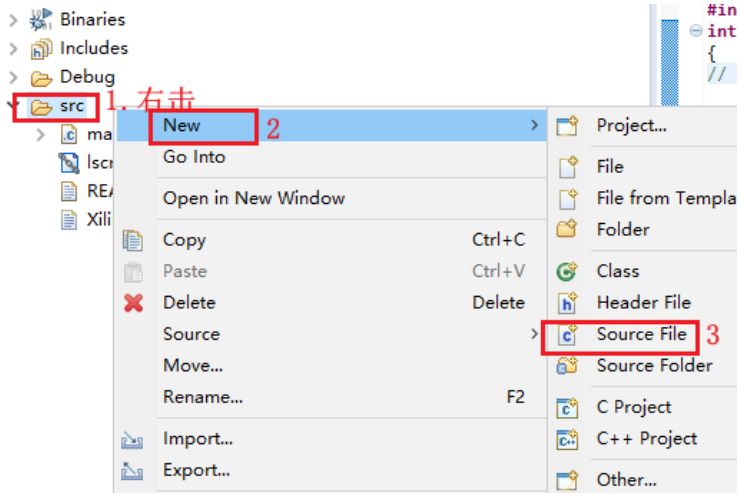

图 1-19 新建源文件

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 crtl+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。

1.5 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 AC608_7Z010_DEV 开发板上外接的 USB 模块 ACM68013 接收，随后从 USB 下发的数据中解析出命令，最终实现对 AD7606 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，AD7606 开始采集数据，将 AD7606 采集的数据通过 USB 传输到电脑。电脑端将接收到的数据进行保存，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。

1.5.1 系统所需硬件

1. AC608_7Z010_DEV 开发板一块
2. ACM68013 模块一个
3. USB 线一根
4. ACM7606 模块一个
5. 电源线一根
6. XILINX 下载器一个
7. 信号发生器一台

1.5.2 硬件连接

本次设计系统硬件连接如下图 1-20 所示：

1. 将下载器连接至开发板 JTAG 下载器口。
2. 使用 5V 的电源给开发板供电。
3. 将 ACM7606 模块连接至开发板的 GPIO1 的 40 pin 的引脚上，靠左连接，1 脚和 1 脚对应。
4. 将 ACM68013 模块连接至开发板的 GPIO0 的 40 pin 的引脚上，靠右连接，1 脚和 1 脚对应。
5. 使用 USB 线连接开发板和电脑。
6. 使用信号发生器输出 200hz，VPP 等于 5V 的正弦波连接至 ACM7606 的通道 1 上。

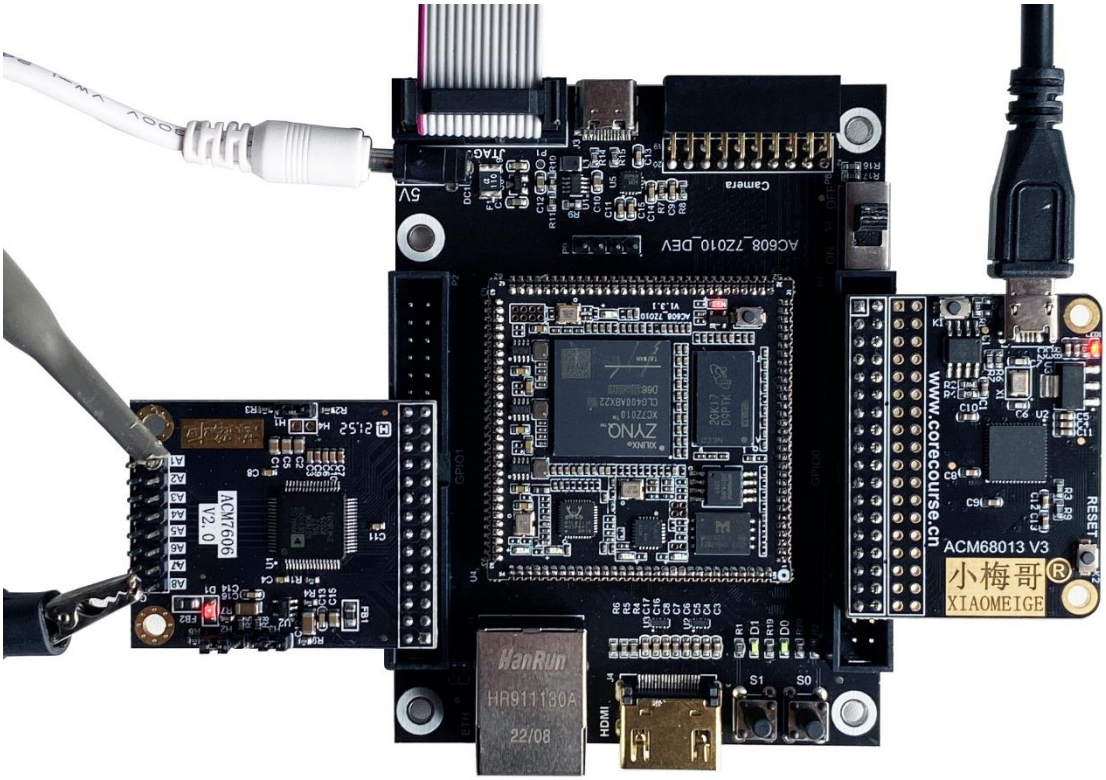

图 1-20 系统硬件链接图

1.5.3 烧录程序

在 SDK 软件中，依次点击 Run->Run Configurations，如下图 1-21 所示。

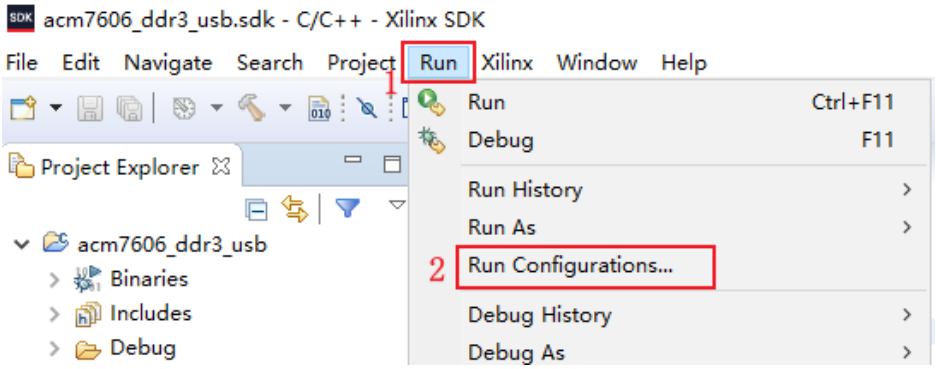

图 1-21 进入下载界面示意图

进入下载界面之后，下载 bit 文件，如下图 1-22 所示。

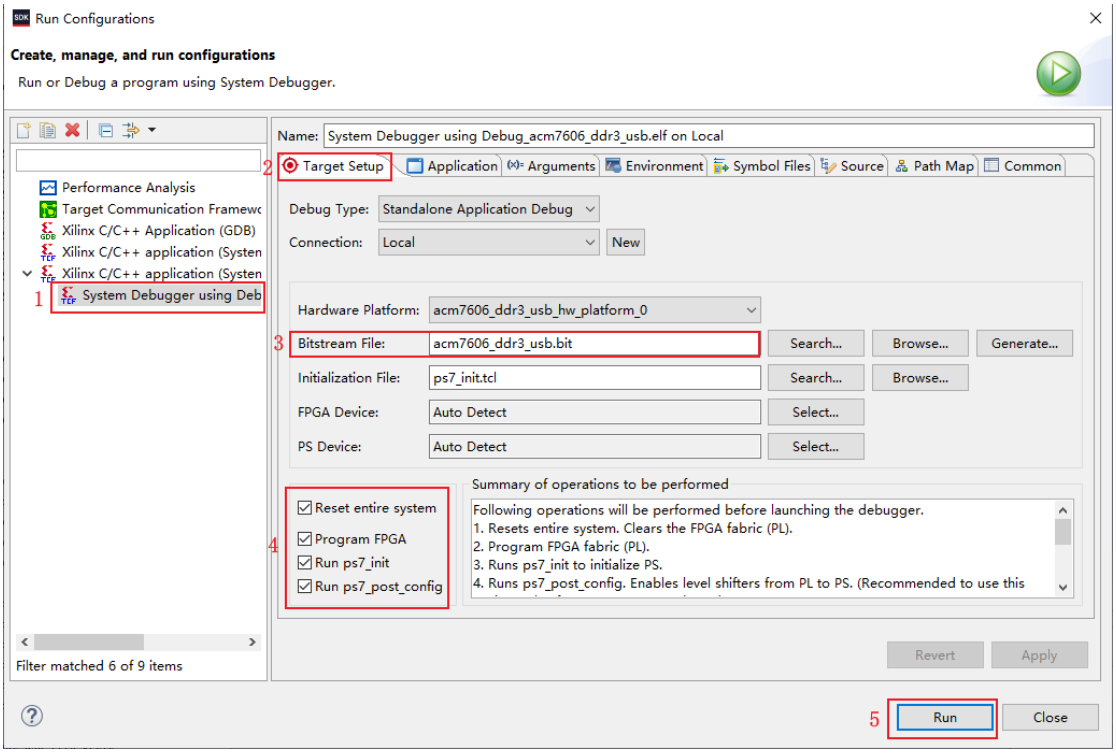

图 1-22 下载 bit 文件

下载成功之后，开发板右下脚的 D1 将会被点亮，D1 被点亮说明 PLL 工作正常，如下图 1-23 所示。

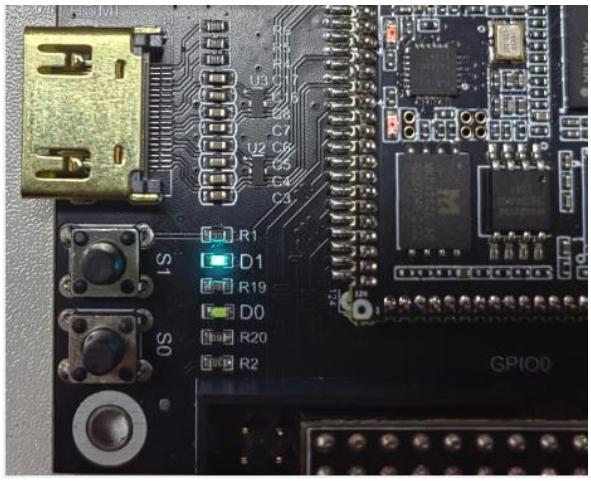

图 1-23 LED0 被点亮

1.5.4 烧写 USB 固件

打开 CyControl 软件，点击设备，烧写我们压缩包下的“slave_for_adc_clk_not.iic”文件至 EEPROM 中，如下图 1-24 所示。

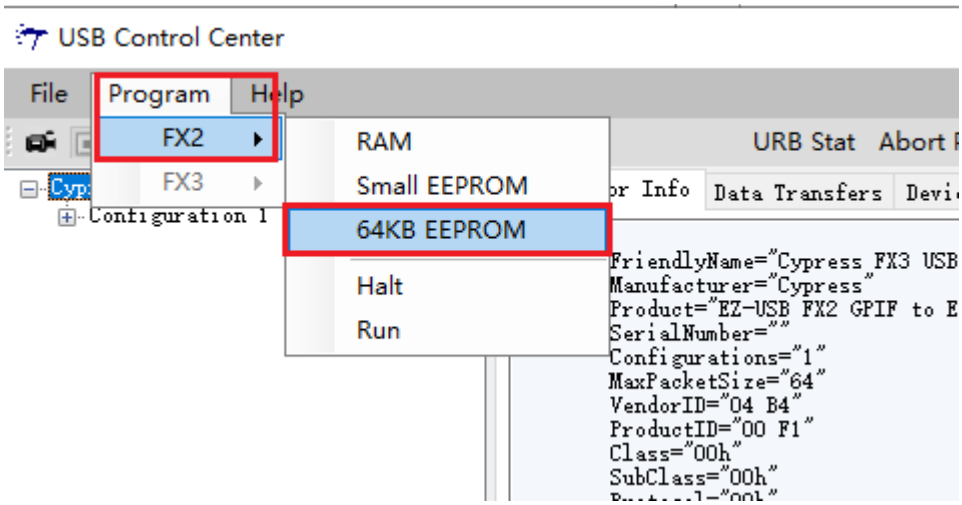

图 1-24 烧写固件

烧写成功之后，软件会显示“Programming successed”，如下图 1-25 所示，然后按一下 USB 模块上的 RESET 按键。

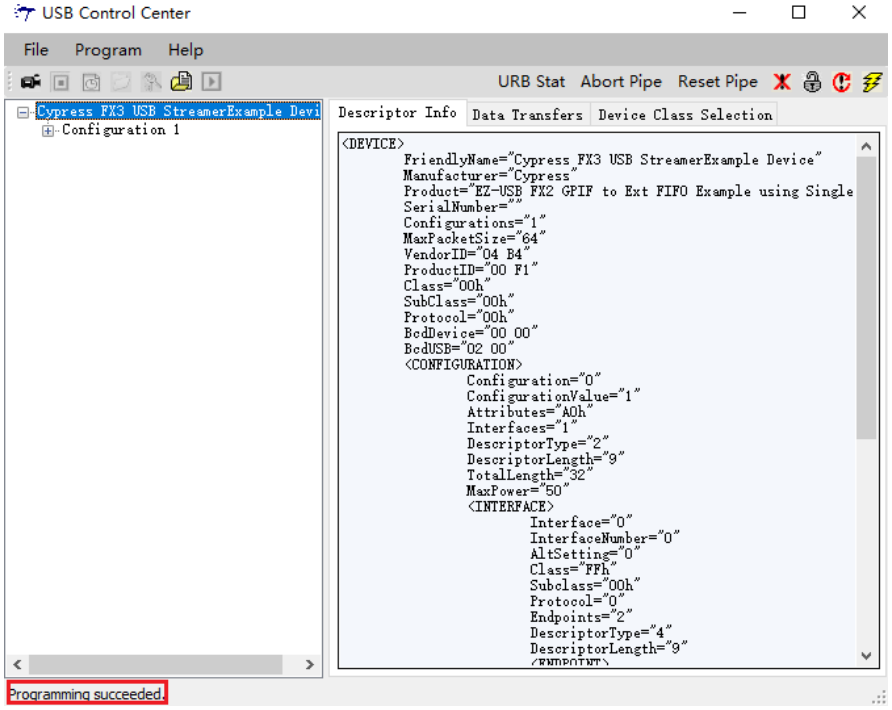

图 1-25 固件烧写成功之后示意图

1.5.5 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压缩包中找到 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 1-26 所示。

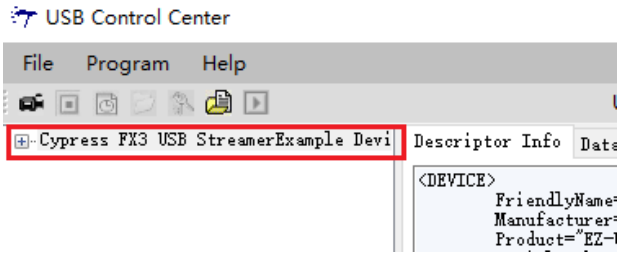

图 1-26 cypress 软件识别到 FX3

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 AD7606 的四个寄存器进行配置。例如 AD7606 以 200K 的采样速率，对 1 个通道进行采样（本次实验以通道 1 为例），共采集 16384 个数据，此时 Cypress 软件需要发送的指令串如下：

55A50200004000F055A50100000001F055A503000000F9F055A50000000000F0

4. 点击“Transfer Data-OUT”进行命令传输，传输完成之后如下图 1-27 所示。

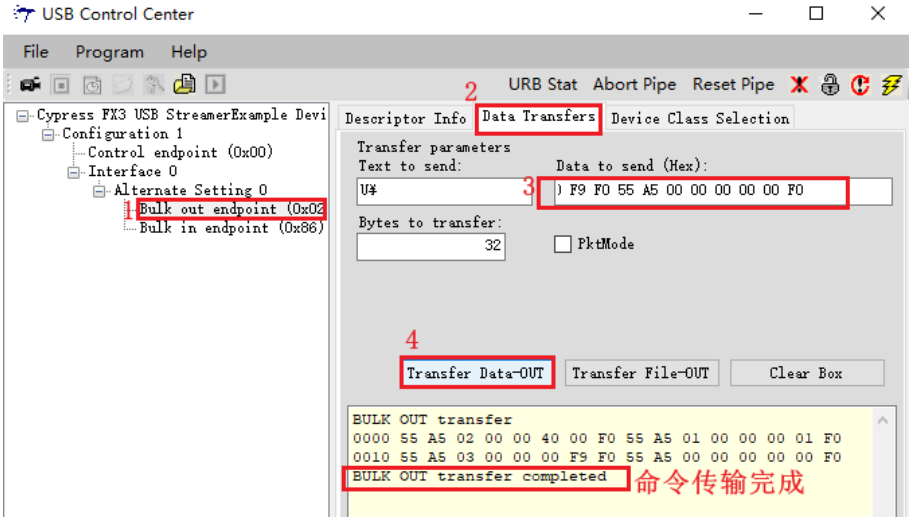

图 1-27 命令传输完成标志

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，AD7606 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 1-28 如下所示。

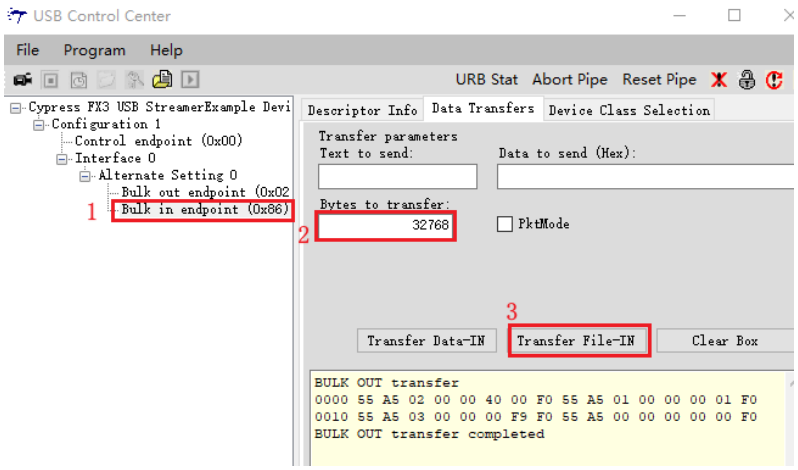

图 1-28 数据接收界面设置

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 E 盘并给文件命名为“ad7606_16384_usb”，然后点击保存。如下图 1-29 所示。

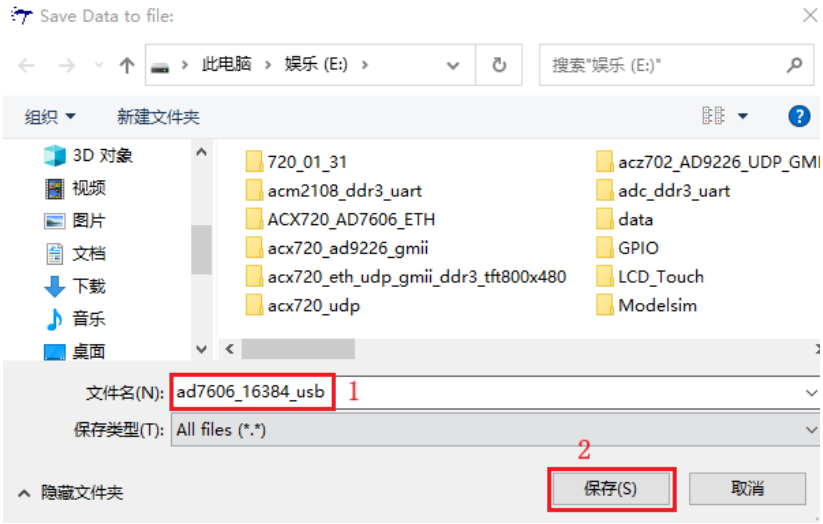

图 1-29 设置文件名称

9. Cypress 软件接收数据，并提示传输成功。如下图 1-30 所示。

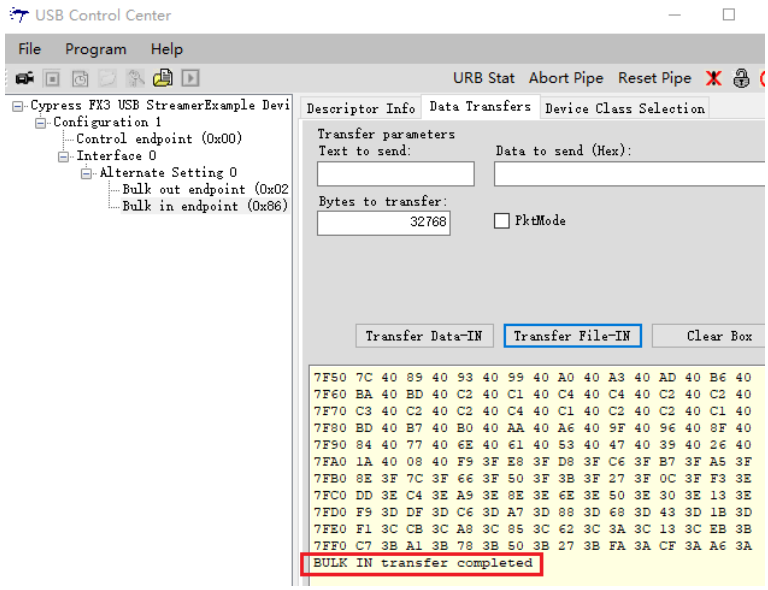

图 1-30 数据传输完成界面显示

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 1-31 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。

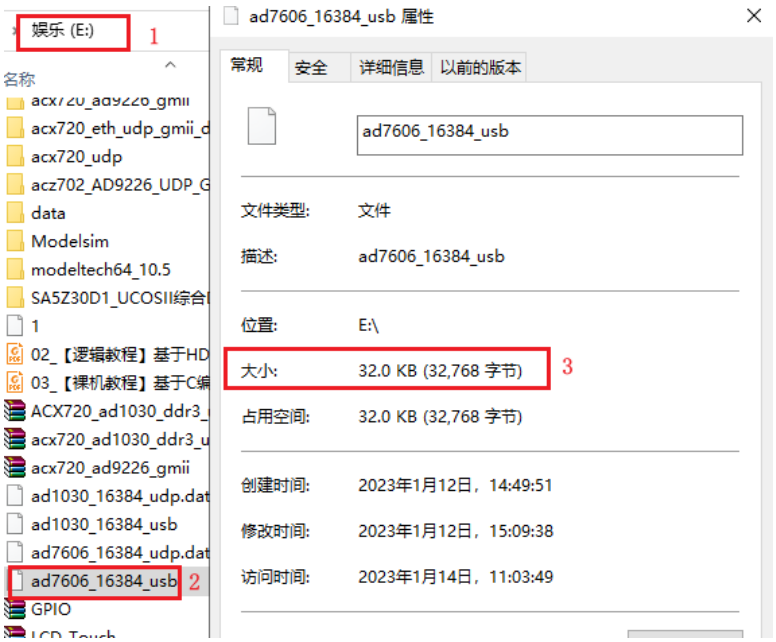

图 1-31 查看接收文件大小

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。这时候建议将开发板复位，按下按键 S0，并且按一下 USB 模块 ACM68013 上的 RESET 按钮，清除上次不正常读写的数据。

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。

1.5.6 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，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 1-32 所示，得到的波形图如下图 1-33 所示。

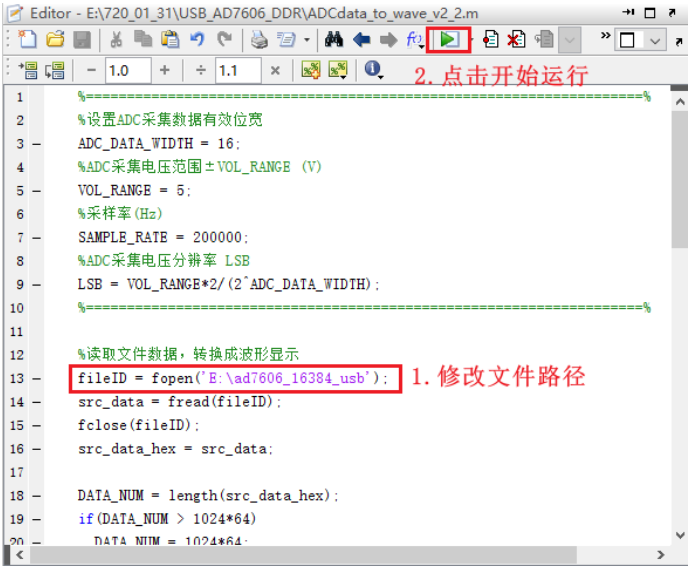

图 1-32 修改文件路径并运行

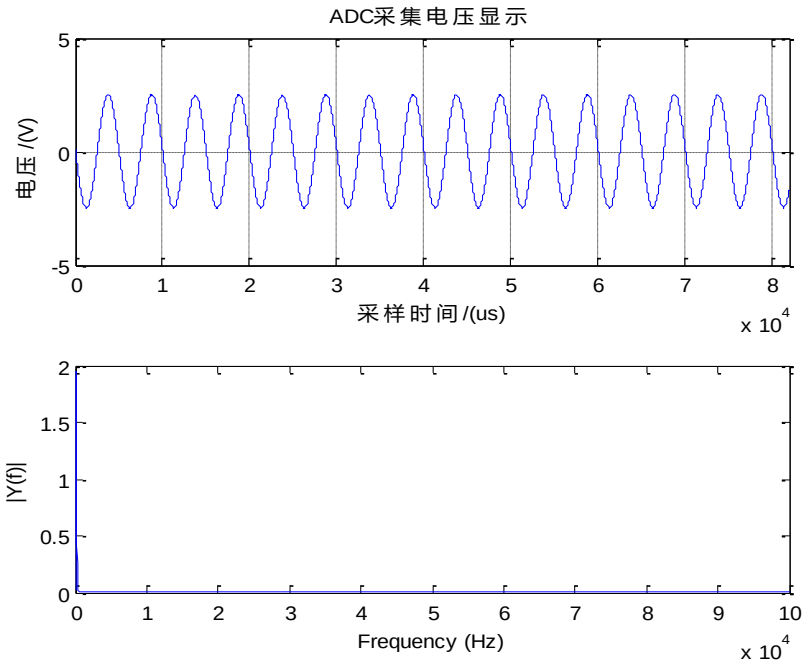

图 1-33 MATLAB 分析波形图

前面我们提到过本次实验提供的信号源为 200hz，Vpp 为 5V 的正弦波（正负 2.5V），与 MATLAB 分析出来的波形一致，说明我们本次实验成功。

1.5.7 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，上位机内部直接对命令进行了构建，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 1-34 所示。

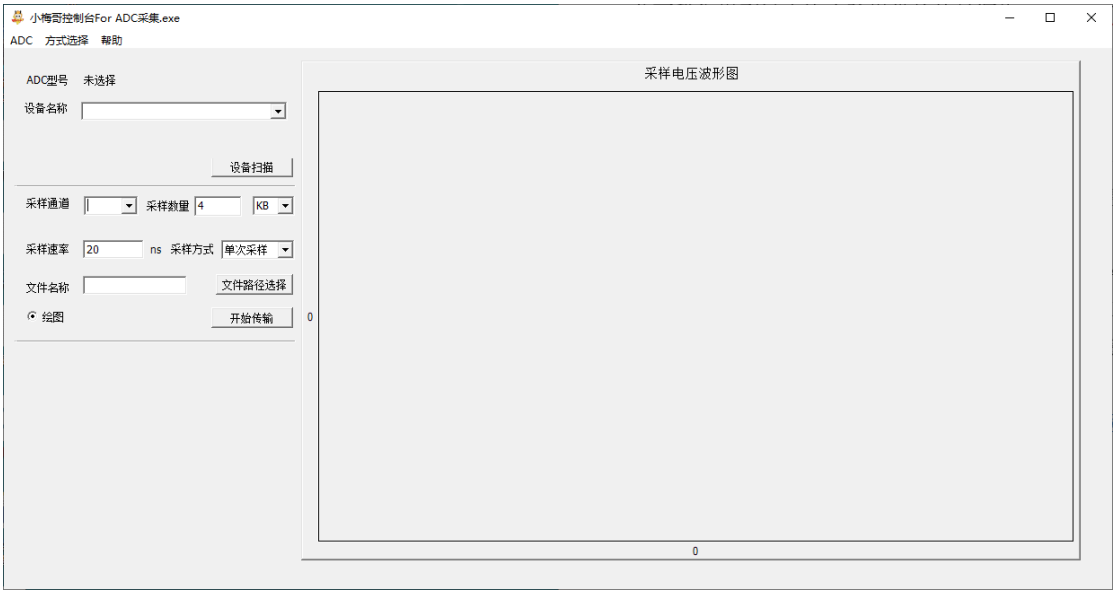

图 1-34 上位机软件初始界面显示

本次实验使用该软件的方式如下所示：

1. 点击 ADC，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（默认设置的采样率为ADC模块的最大采样率），用户可以根据自己的需求进行修改。
4. 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 1-35 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。

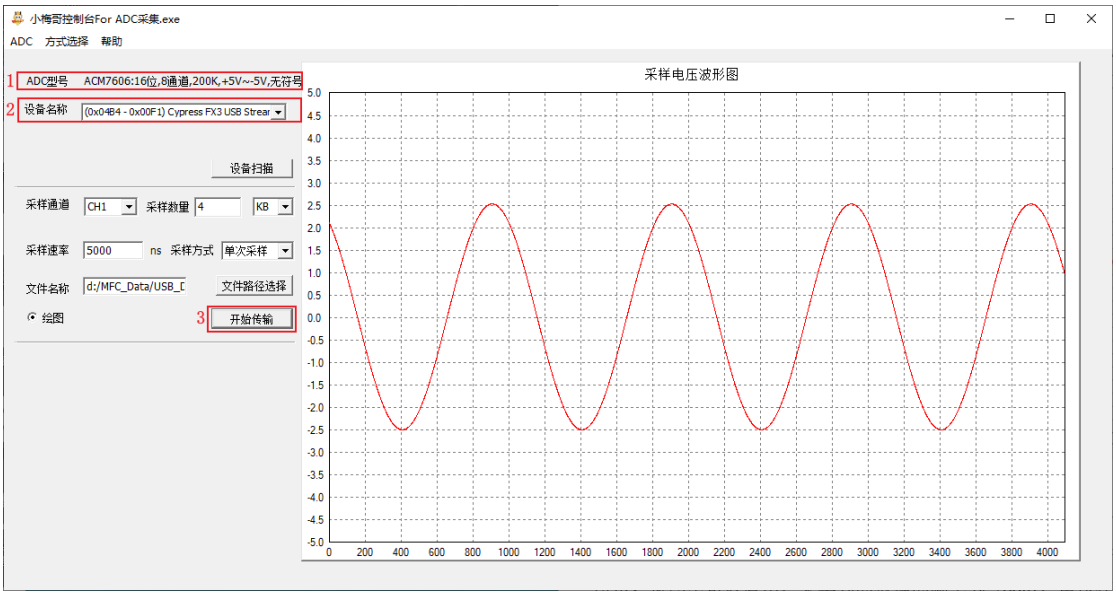

图 1-35 数据采集上位机显示图

通过上位机采集到的数据文件保存在 d:/ MFC_Data 文件夹下，后续可以通过 MATLAB 软件进行进一步的分析，通过 MATLAB 分析的波形图如下图 1-36 所示。从图中可以看出，采集到的数据的频率为 200hz，电压在正负 2.5V 左右，与我们输入的信号一致。

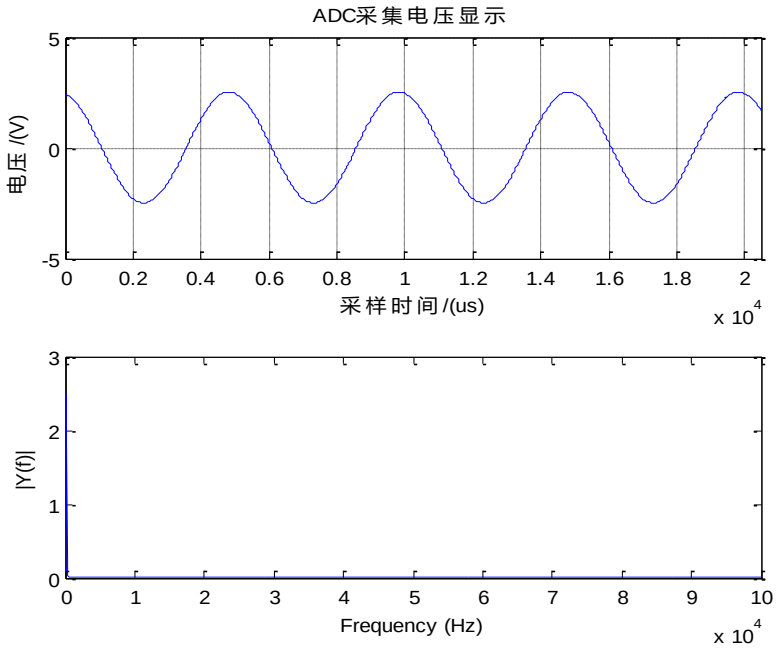

图 1-36 MATLAB 进一步波形分析图

1.6 思考与总结

本次实验介绍了基于 AD7606 的 USB 收发数据，用户通过 FX2_USB 调试

工具 CyControl 向开发板发送指令数据配置 AD7606 的四个寄存器，以此控制 ADC 进行采样，ADC 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 传输过来的数据以文件的形式进行保存，最终通过 MATLAB 对数据进行了进一步的分析。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请读者自行查看 ACM68013 模块资料的内容，模块资料可以去我们论坛中获取，模块资料链接如下所示：

【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