1 基于 ACM7606 的多通道简易示波器

章节导读

模数转换器即 A/D 转换器，或简称 ADC，通常是指一个将模拟信号转变为数字信号的电子元件。通常的模数转换器是将一个输入电压信号转换为一个输出的数字信号。由于数字信号本身不具有实际意义，仅仅表示一个相对大小。故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参考标准为最大的可转换信号大小。而输出的数字量则表示输入信号相对于参考信号的大小。

本章将基于前面章节设计的 RGB_TFT 工程，结合 ACM7606 模块，完成带截图功能的多通道简易示波器的设计。除了波形显示，用户可以通过触摸屏对采样率、采样通道、触发模式等参数进行设置，以及通过按键对当前内存中的数据截取，并以.bmp 格式存储进 SD 卡中。

1.1 ACM7606 概述

ACM7606 是小梅哥团队出品的一款 16 位 8 通道的高速 ADC 模块，该模块如下图 1-1 所示：

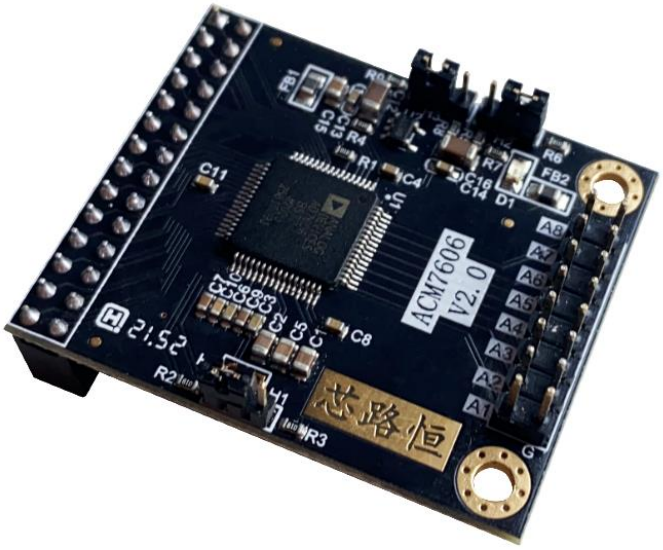

图 1-1 ACM7606 模块图

模块使用 ADI 公司的 16 位 8 通道同步采样模数转换器 AD7606 芯片。芯片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，具有 40dB 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 200KSPS 的最高速率进行转换时，数据输出速率达到了 25.6Mbps，需要使用高性能 MCU 的 SPI 外设才能勉强满足该速率要求。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 的转换控制逻辑，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，也可以直接使用软和控制器完成数据的处理和传输工作，如典型的，在 Intel FPGA 器件上使用 NIOS II 软核控制器，将 AD7606 的转换结果通过串口或以太网传输到电脑上，也可以直接将数据显示在液晶显示屏上。

1.2 功能框图

AD7606 芯片的功能框图如图 1-2 所示：

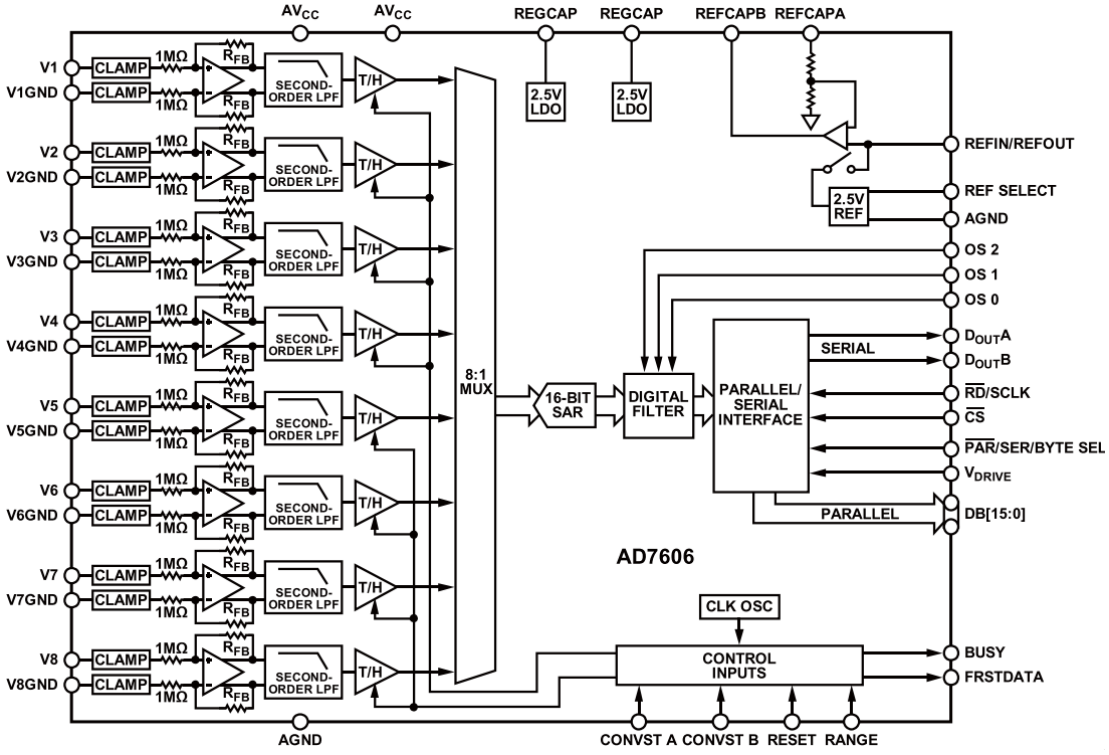

图 1-2 AD7606 功能框图

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，当数据是以串行模式输出时，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，下面为 AD7606 部分模块的说明介绍。

1.2.1 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 的模拟输入结构如下图，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。

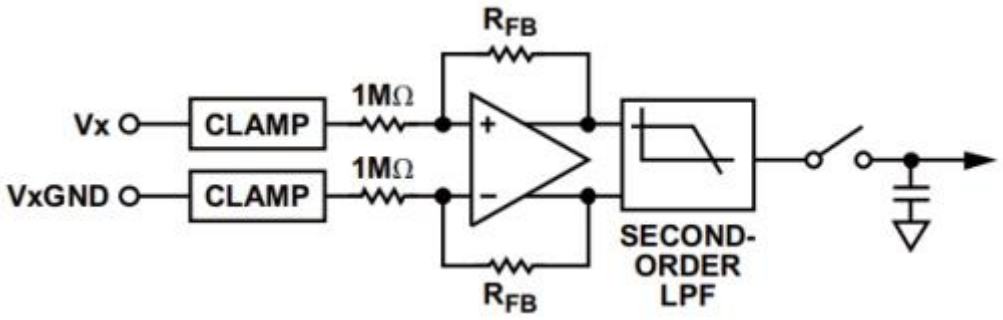

图 1-3　AD7606 模拟输入结构

1.2.2 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倍率由过采样引脚 OS[2:0]控制（具体参考 AD7606 数据手册）。OS[2]为 MSB 控制位，OS[0]为 LSB 控制位，下表提供了用来选择不同过采样倍率的过采样位解码。

表 1-1　不同过采样倍率下过采样位解码

OS[2:0]	过采样倍率	5V 范围 SNR(dB)	10V 范围 SN（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。

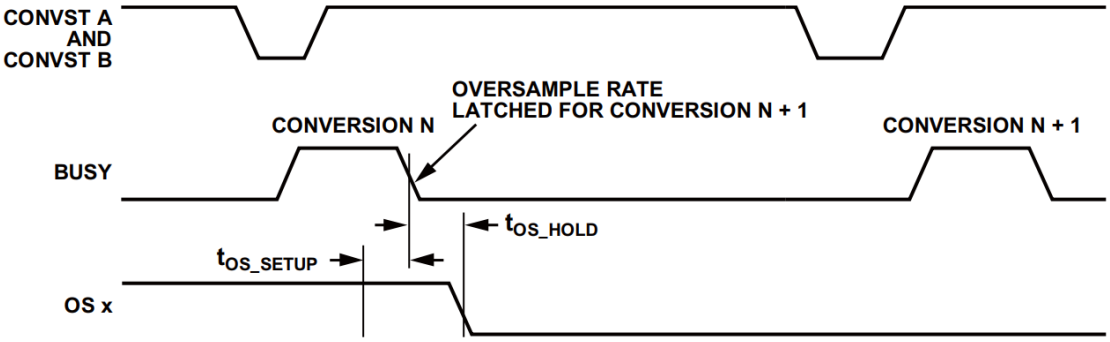

图 1-4　AD7606 过采样下时序图

1.2.3 工作时序

AD7606 根据采样方式不同具有多种驱动时序，本次采用的为并行输出（即 8 个 16 位的数据通过 16 根并行线一个接着一个输出），转化后读取模式。其时序图有两部分组成：1.完成 AD 转化；2.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。

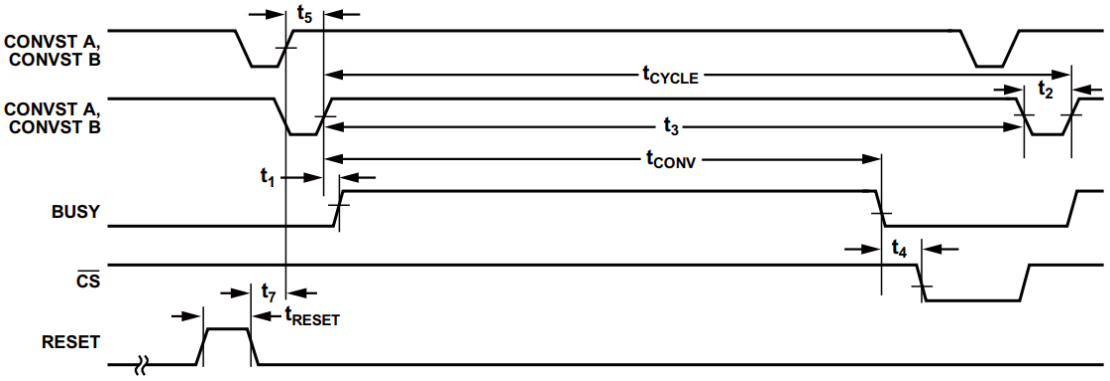

图 1-5　CONVST 时序--转换之后读取

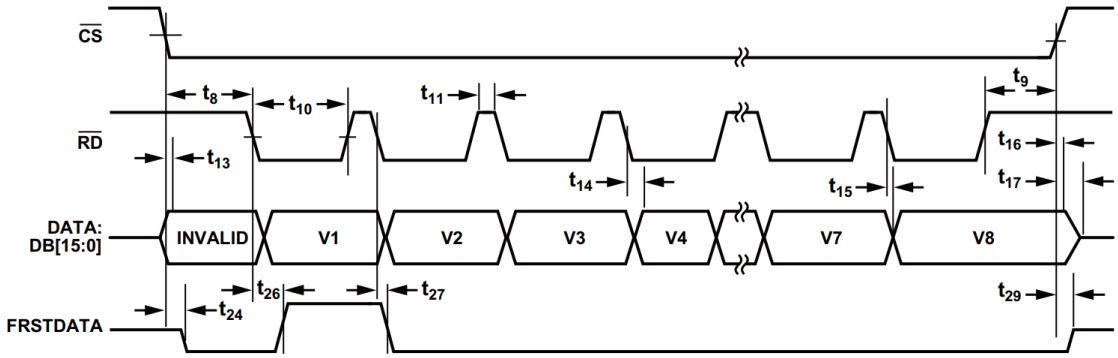

图 1-6　并行模式，独立的$\overline{CS}$和$\overline{RD}$

介绍完 AD7606 的工作原理，接下来即可开始进行本次设计。

1.3 应用实例

本次设计将会使用到独立模块 ACM7606，该模块通过开发板上 40pin 拓展接口连接，支持 8 通道电压数据采集。用户可以设置采集其中任一通道的数据，自定义的 IP 核会根据数据计算出其最大值、最小值、中间值等特殊值，这些数据会存储进 DDR 中，最终在触摸屏中显示。用户可以通过触摸屏中的按键设置不同的触发模式、采样频率、采样通道、触发电压等。

1.3.1 硬件逻辑系统设计

本次设计的显示部分与 RGB_TFT 工程相同，因此将会基于该工程搭建硬件逻辑系统。将 RGB_TFT 另存为一个新工程，作为本次硬件设计的基础。

1.3.1.1导入 IP 核

本次设计除了 RGB_LCD 实验所要用到的 dynclk 和 rgb2lcd 这两个自定义 IP 核，还需要添加 ADC_Acq2DDR_IP 和 rgb565to888 自定 IP 核。这些 IP 核存储在对应例程的 ip_repo 文件夹中，路径如下：*ACZ702 开发板资料\02_设计实例\03_[裸机例程]基于 C 编程的 Zynq 裸机例程*。添加完成后如图 1-7 所示：

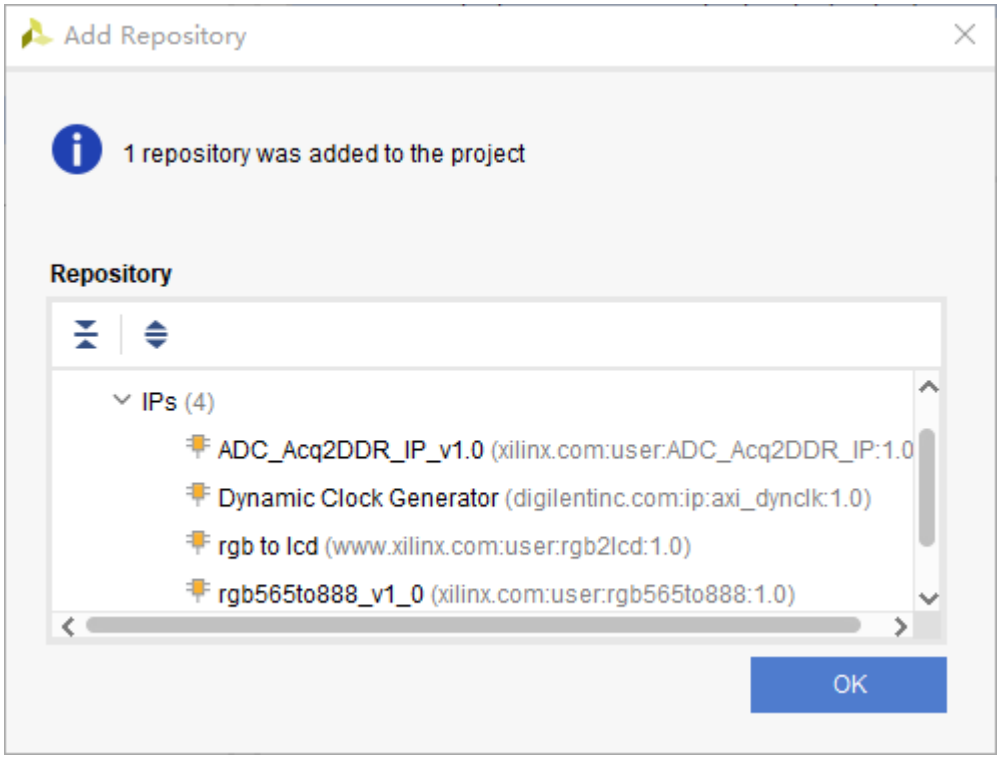

图 1-7　添加自定义 IP 核

添加完成后如果出现部分 IP 核被锁的情况，用户只需更新 IP 核即可。对于点击了稍后更新的用户可以关闭 Block Design，再重新打开便会再次看到更新提醒。

除了 ADC_Acq2DDR_IP 核外，其余 IP 核我们在前面章节中都有过介绍，而 ADC_Acq2DDR_IP 核则是我们自定义的一个用于驱动 ACM7606 采样并将符合触发条件的数据通过 HP 接口存入 DDR 的自定义 IP 核。该 IP 核在本次设计中扮演着及其重要的角色，因此，接下来我们先来了解下该 IP 的内部结构以及工作原理。

1.3.1.2 ADC_Acq2DDR_IP 核介绍

ADC_Acq2DDR_IP 核是我们自定义的一个带 AXI4 接口的 IP 核。该 IP 核能够实现对 ACM7606 模块触发模式、采样频率、采样通道、触发电压的输出和控制，同时能够将测量到的特殊值传输给 ARM 进一步处理。ADC_Acq2DDR_IP 核的结构框图如图 1-8 所示：

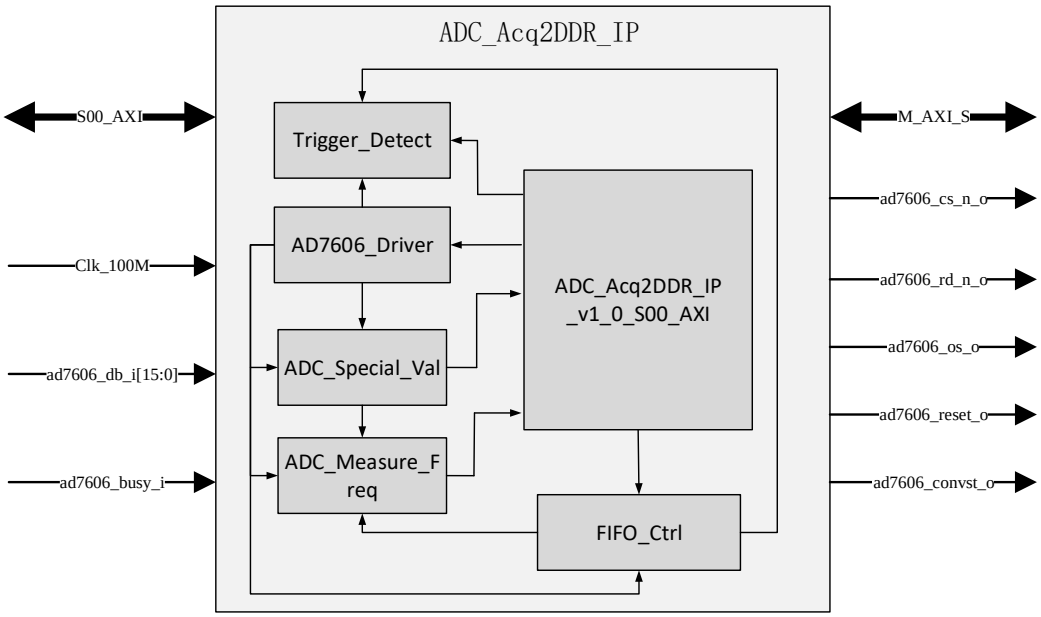

图 1-8　ADC_Acq2DDR_IP 模块结构

其中 S00_AXI 为 AXI4-Lite 总线接口，用户可以通过该接口，对 IP 核内部进行读写配置。M_AXI_S 为 AXI4-Stream 总线接口，用于将 ADC 采集到的满足触发条件的信号通过 HP 接口写入 DDR 中。Clk_100M 为 100M 输入时钟信号，ad7606_xxx 为 ad7606 相关的输入输出信号。

下面将对该 IP 中使用到的模块进行说明。

1.3.1.2.1　AD7606_Driver

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，该模块的基本框图如图 1-9 所示：

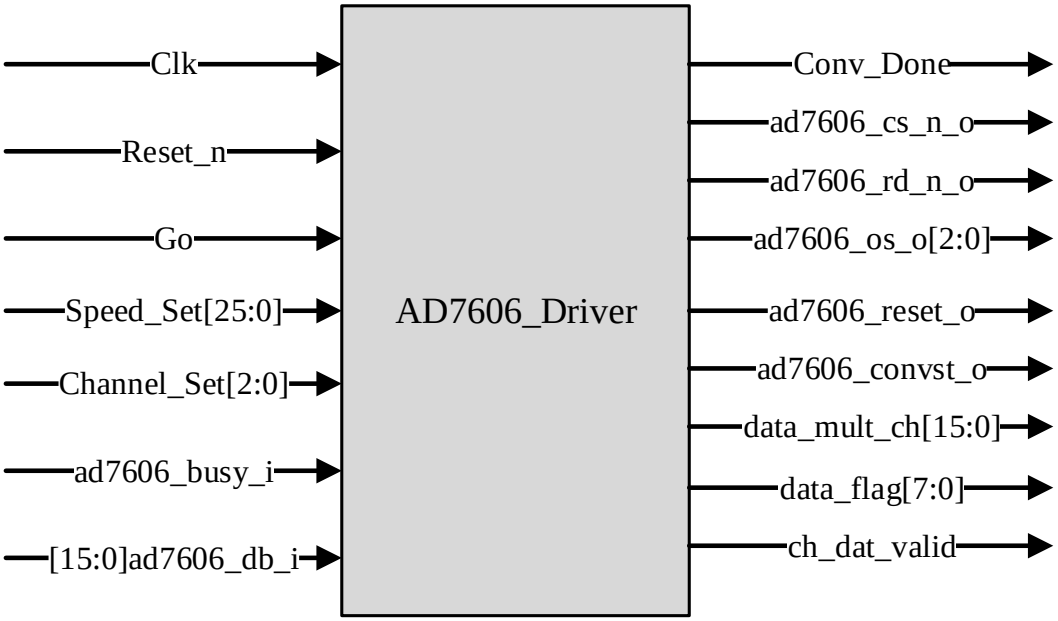

图 1-9　ad7606_driver 模块基本框图

该模块的接口信号功能如所示：

表 1-2　ad7606_driver 模块接口信号

信号名称	I/O	信号意义
Clk	I	时钟信号，为了让采样速率准确，要求为 100MHz
Reset_n	I	复位，低电平复位
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[25:0]	I	采样速率控制端口，Speed_Set = 100000000/speed - 1
Channel_Set[2:0]	I	采样通道设置，采集从 0 通道到设定通道 Channel_Set 的值
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
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。
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

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。
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
ch_dat_valid	O	输出通道有效信号

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;
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
```

1.3.1.2.2 ADC_Special_Val

ADC_Special_Val 模块用于得到 ADC 采集电压的最大、最小和中值，该模

块的基本结构如下图 1-10 所示。

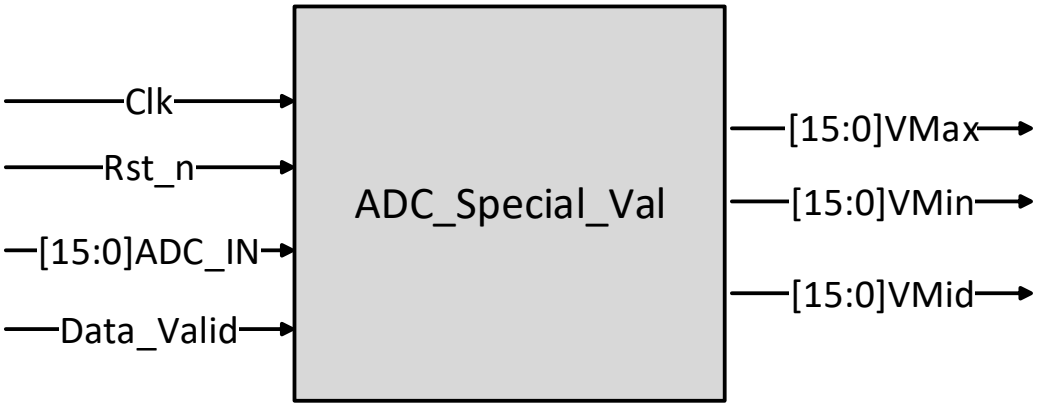

图 1-10 ADC_Special_Val 模块的基本框图

该模块的信号说明如表 1-3 所示。

表 1-3 ADC_Special_Val 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_IN[15:0]	I	ADC 采集得到的 16 位数据
Data_Valid	I	ADC 采集数据有效信号
VMaxt[15:0]	O	采集信号的最大值
VMin[15:0]	O	采集信号的最下值
VMid [15:0]	O	采集信号的中值

在获取 AD7606 采集电压的特殊值时，首先需要知道 AD7606 的输出编码方式为二进制补码的方式，也就是说用 0/1 来区分一个数的符号，为 0 则代表正数，为 1 代表复数，其电压对应的二进制值如下图 1-11 所示。

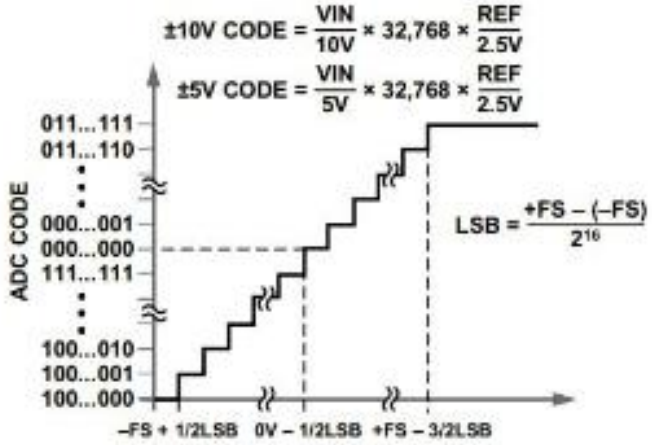

图 1-11 电压对应二进制数值图

从上图可以看出，AD7606 采集的数值最高位对应的就是符号位，我们在获取其采集电压的最大、最下值时，需要根据是正数还是复数进行判断。

首先我们定义一个计数器，计时达到 1S 时，重新开始计数，代码如下所示：

```
parameter UPDATERATE = 99999999;//计时一秒
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    counter <= 0;
else if(counter >= UPDATERATE)
    counter <= 0;
else
    counter <= counter + 1'b1;
```

然后需要得到采集电压的最大值，当计数达到 1S 时，使电压最大值 VMax_r 为-32767（1111 1111 1111 1111）；当输入的电压 ADC_IN 最高位 ADC_IN[15]小于 VMax_r 最高位 VMax_r[15]并且数据有效时，代表此时输入的 ADC_IN 为正数，VMax_r 为负数，最大值 VMax_r 为 ADC_IN；当输入电压 ADC_IN 最高位 ADC_IN[15]等于 VMax_r 最高位 VMax_r[15]时，代表此时两个数都是正数或者复数，这时只需要比较两个值的大小，大的那个对应的就是电压最大值，否则电压的最大值保持不变，代码如下所示：

```
always@(posedge Clk)
if(counter >= UPDATERATE)
    VMax_r <= -16'd32767;//-32767
else if((ADC_IN[15] < VMax_r[15]) && Data_Valid)//IN 为正，r 为负，IN>r
    VMax_r <= ADC_IN;
else if((ADC_IN[15]==VMax_r[15]) && (ADC_IN > VMax_r) && Data_Valid)
    VMax_r <= ADC_IN;
else
    VMax_r <= VMax_r;
```

然后需要得到采集电压的最小值，当计数达到 1S 时，使电压最小值 VMin_r 为-32767（1111 1111 1111 1111）；当输入的电压 ADC_IN 最高位 ADC_IN[15]大于 VMin_r 最高位 VMin_r [15]并且数据有效时，代表此时输入的 ADC_IN 为负数，VMax_r 为正数，最小值 VMin_r 为 ADC_IN；当输入电压 ADC_IN 最高位 ADC_IN[15]等于 VMin_r 最高位 VMin_r [15]时，代表此时两个数都是正数或者复数，这时只需要比较两个值的大小，小的那个对应的就是电压最小值 ，否则，电压的最小值保持不变，代码如下所示：

```
always@(posedge Clk)
if(counter >= UPDATERATE)
    VMin_r <= 16'd32767;
else if((ADC_IN[15] > VMin_r[15]) && Data_Valid)//IN 为负，r 为正，IN<r
    VMin_r <= ADC_IN;
else if((ADC_IN[15]==VMin_r[15]) && (ADC_IN < VMin_r) && Data_Valid)
    VMin_r <= ADC_IN;
else
    VMin_r <= VMin_r;
```

最后，当计数达到 1S 后，将最大值和最小值进行输出，否则保持不变，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)begin
    VMin <= 0;
    VMax <= 0;
end
else begin
    if(counter >= UPDATERATE)begin
        VMin <= VMin_r;
        VMax <= VMax_r;
    end
    else begin
        VMin <= VMin;
        VMax <= VMax;
    end
end
```

得到最大、最下值之后，首先将其相加，然后除以 2，便得到其中间值，代码如下所示：

```
assign VMid_r = $signed(VMax) + $signed(VMin);
assign VMid = VMid_r[16:1];
```

上述代码中的$signed（）作用是将 VMax 和 VMin 作为有符号数进行相加，在进行运算的时候就会按照有符号数进行扩位，在高位补符号位，然后取 VMid_r[16:1]，这样就得到了 16 位的有符号的电压中值数据，举个例子，当最大值为+29408（0111001011100000），最小值为-32520（1111111100001000），两者相加得到-29160(10111000111101000)，取高 16 位得到-14580（1011100011110100），也就是将-29160 除以 2 的数值，也就是采集电压的中值。

1.3.1.2.3 ADC_Measure_Freq

ADC_Measure_Freq 模块用来得到采集信号的频率值，该模块的基本结构框图如图 1-12 所示：

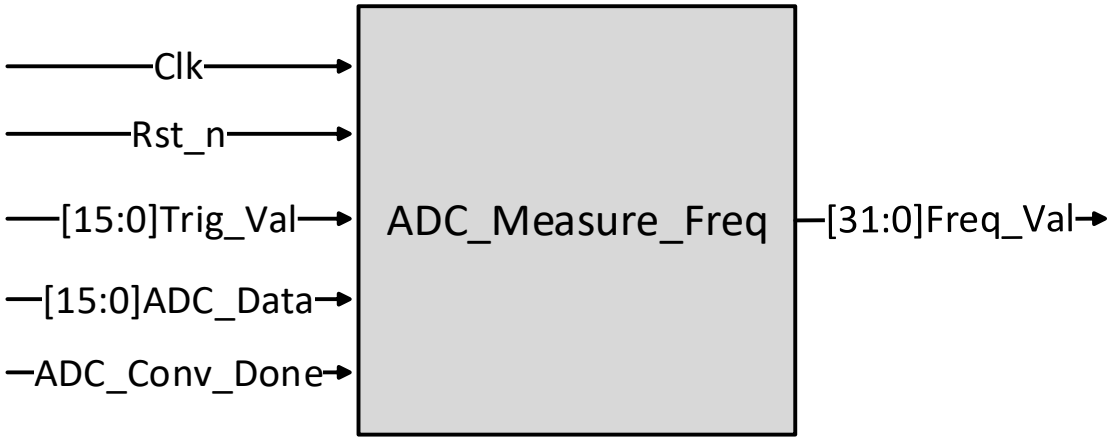

图 1-12 ADC_Measure_Freq 模块的基本结构框图

该模块的信号说明如下表 1-4 所示。

表 1-4 ADC_Measure_Freq 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
Trig_Val [15:0]	I	频率测量触发值
ADC_Data[15:0]	I	ADC 采集到的 16 位数据
ADC_Conv_Done	I	ADC 单次采集完成信号
Freq_Val [31:0]	O	采集信号的频率测量值

首先设计一个计数器，根据设定的计数值来控制采样时间，这里设定计数值为 99999999，也就是计数 1S，代码如下所示：

```
parameter TIME_CNT_VAL = 99999999; //设定计数值来控制采样时间
//计数 1 秒
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Time_Cnt <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Time_Cnt <= 0;
    else
        Time_Cnt <= Time_Cnt + 1;
end
```

保存触发设定值，防止中途被修改，每次测完一秒后才允许被修改，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Trig_Val_r <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Trig_Val_r <= Trig_Val;
    else
        Trig_Val_r <= Trig_Val_r;
end
```

保存上一次 ADC 采集到的值，代码如下所示：

```
//保存上一次 ADC 测量值
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        ADC_Data_Pre <= 0;
    end
    else if(ADC_Conv_Done) begin
```

```
        ADC_Data_Pre <= ADC_Data;
    end
    else begin
        ADC_Data_Pre <= ADC_Data_Pre;
    end
end
```

对 1S 内的触发点的个数进行计数，以此测得频率。当上次采样的电压值 ADC_Data_Pre 小于触发值 Trig_Val_r、本次采样的电压值 ADC_Data 大于 Trig_Val_r、上次采样电压值 ADC_Data_Pre 小于本次采样电压值 ADC_Data、单次采样完成，满足以上条件，触发周期计数加 1，这里以电压中值为触发，使用采集电压的中值作为触发值，是因为在中值两边的电压变化最明显，测得的频率最准，这里以正弦波为例，触发条件如下图 1-13 所示。

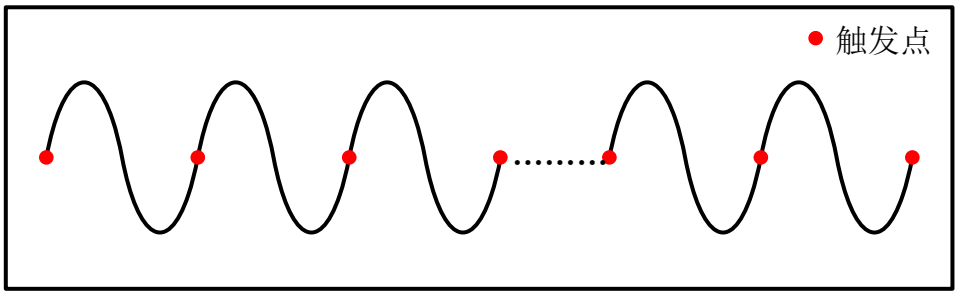

图 1-13 正弦波触发条件波形图

记录一次触发周期的时间，也就是满足触发条件所需时间，从上图可以看出，两次触发之间需要的时间刚好是一个周期的时间，代码如下所示：

```
//记录一次触发周期的时间
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Cycle_Cnt <= 0;
    else if((ADC_Data_Pre<=Trig_Val_r) && (ADC_Data>=Trig_Val_r) &&
(ADC_Data_Pre<ADC_Data) && ADC_Conv_Done)
        Cycle_Cnt <= 0;
    else
        Cycle_Cnt <= Cycle_Cnt + 1;
    end
```

保存上一次触发周期的时间，用于比较两次触发周期的时间，防止出现误触发的现象，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Cycle_Cnt_Pre <= 0;
```

```
    else if((ADC_Data_Pre <= Trig_Val_r) && (ADC_Data >= Trig_Val_r)
&& (ADC_Data_Pre < ADC_Data) && ADC_Conv_Done)
        Cycle_Cnt_Pre <= Cycle_Cnt;
    else
        Cycle_Cnt_Pre <= Cycle_Cnt_Pre;
end
```

记录 1S 内累加的触发计数，以此测得频率，如果当前周期计数小于等于上一个计数周期的一半，则此次触发异常，触发计数无效，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Freq_Add <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Freq_Add <= 0;
    else if((ADC_Data_Pre <= Trig_Val_r) && (ADC_Data >= Trig_Val_r)
&& (ADC_Data_Pre < ADC_Data) && ADC_Conv_Done) begin
        if(Cycle_Cnt <= Cycle_Cnt_Pre[31:1])
            Freq_Add <= Freq_Add;
        else
            Freq_Add <= Freq_Add + 1;
    end
    else
        Freq_Add <= Freq_Add;
end
```

最后将计数 1 后测出的频率值进行输出，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Freq_Val <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Freq_Val <= Freq_Add;
    else
        Freq_Val <= Freq_Val;
end
```

1.3.1.2.4 FIFO_Ctrl

本次实验将 AD7606 驱动模块 AD7606_Driver 的采样率设置为其最高采样频率 200Ksps，如果想要改变其采样频率，可以通过对采样数据进行抽取重采样的方法实现。比如希望以 40Ksps 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 40Ksps 的采样率。FIFO_Ctrl 模块就是用于实现该功能，该模块通过输入的采样分频值，最终产生数据有效标志，将需要数据写入至 FIFO 中，该模块的基本结构如图 1-14

下所示。

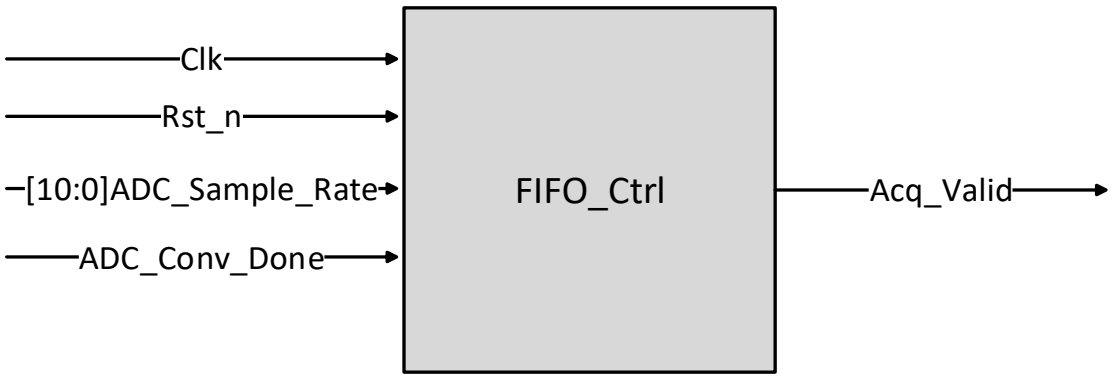

图 1-14 FIFO_Ctrl 模块的基本结构图

该模块的信号说明如下表 1-5 所示。

表 1-5 FIFO_Ctrl 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_Sample_Rate [10:0]	I	采样分频值
ADC_Conv_Done	I	ADC 单次采集完成信号
Acq_Valid	O	采集有效标志信号

首先设计一个采样分频计数器，根据采样分频值，当产生单次采集完成之后，计数器进行计数，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if (!Rst_n)
        Acq_Div_Cnt <= 0;
    else if(Acq_Div_Cnt >= ADC_Sample_Rate)
        Acq_Div_Cnt <= 0;
    else if(ADC_Conv_Done)
        Acq_Div_Cnt <= Acq_Div_Cnt + 1;
    else
        Acq_Div_Cnt <= Acq_Div_Cnt;
end
```

然后产生数据有效标志，当采样分频计数器达到设定值时，产生数据有效标志信号 Acq_Valid，否则，清除该标志信号，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if (!Rst_n)
        Acq_Valid <= 0;
    else if(Acq_Div_Cnt >= ADC_Sample_Rate)
        Acq_Valid <= 1;
    else
```

```
        Acq_Valid <= 0;
end
```

1.3.1.2.5 Trigger_Detect

触发检测模块（Trigger_Detect）根据设置的不同触发模式得到的采集触发值，最终输出传输有效标志信号，通过该信号，产生 RAM 的写使能信号，将数据写入至 RAM 中，该模块的基本结构如下图 1-15 所示。

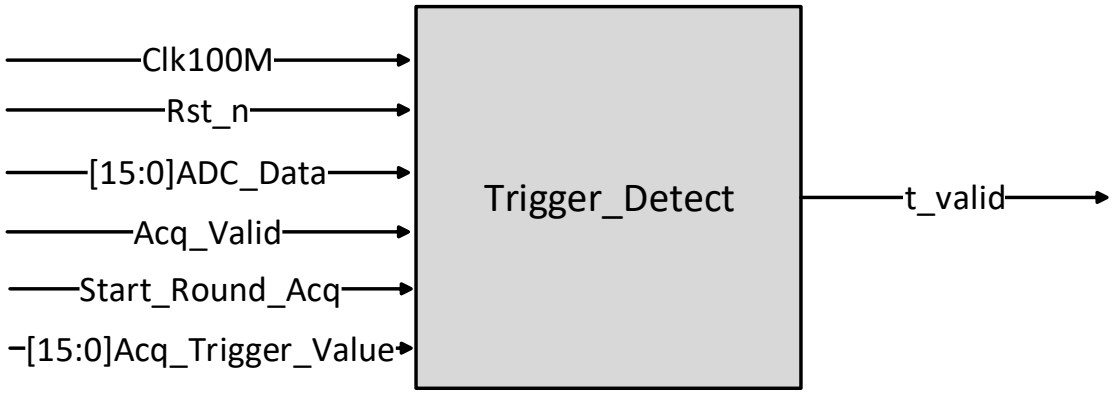

图 1-15 Trigger_Detect 模块的基本结构框图

对该模块的信号说明如下表 1-6 所示。

表 1-6 Trigger_Detect 模块的信号说明表

信号名称	I/O	信号意义
Clk100M	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_Data [15:0]	I	ADC 采集的 16 位的数据信号
Acq_Valid	I	采集有效标志信号
Start_Round_Acq	I	启动一轮采集标志信号
Acq_Trigger_Value[15:0]	I	采集触发值
t_valid	O	传输有效标志信号

首先设置 8 个 16 位的寄存器，当产生采集有效标志信号时，对 ADC 输出的 16 位的寄存器进行移位寄存，代码如下所示：

```
reg signed [15:0]ADC_Data_Reg[7:0];//寄存器
always@(posedge Clk100M)
if (Acq_Valid) begin
    ADC_Data_Reg[7] <= ADC_Data;
    ADC_Data_Reg[6] <= ADC_Data_Reg[7];
    ADC_Data_Reg[5] <= ADC_Data_Reg[6];
    ADC_Data_Reg[4] <= ADC_Data_Reg[5];
    ADC_Data_Reg[3] <= ADC_Data_Reg[4];
    ADC_Data_Reg[2] <= ADC_Data_Reg[3];
    ADC_Data_Reg[1] <= ADC_Data_Reg[2];
    ADC_Data_Reg[0] <= ADC_Data_Reg[1];
end
```

设置一个累加器，产生采集有效标志信号的时候，将 8 个 16 位的寄存器进行累加，代码如下所示：

```
reg signed [18:0]ADC_Data_Reg_Add; //累加器
always@(posedge Clk100M)
begin
    if(Acq_Valid)
    ADC_Data_Reg_Add <= ADC_Data_Reg[7] + ADC_Data_Reg[6] +
                        ADC_Data_Reg[5] + ADC_Data_Reg[4] +
                        ADC_Data_Reg[3] + ADC_Data_Reg[2] +
                        ADC_Data_Reg[1] + ADC_Data_Reg[0];
end
```

然后得到 ADC 输出数据的平均值，也就是累加器的[18:3]的数据，相当于除以 8，代码如下所示：

```
wire signed [15:0]ADC_Average_Val; //平均值
assign ADC_Average_Val = ADC_Data_Reg_Add[18:3];
```

当产生数据有效信号的时候，将得到的 ADC_Average_Val 数据进行寄存，这样在产生下一次的 ADC_Average_Val 信号的时候，使其保持不变，就相当于保存上次的 ADC 采集的数据平均值信号，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        ADC_Average_Val_Pre <= 0;
    else if(Acq_Valid)
        ADC_Average_Val_Pre <= ADC_Average_Val;
    else
        ADC_Average_Val_Pre <= ADC_Average_Val_Pre;
end
```

设计一个状态机，包含空闲状态和等待触发状态，如下所示：

```
localparam
    IDLE = 2'b01,//空闲态
    WAIT = 2'b10;//等待触发态
```

下面对这个状态的实现进行说明。

1. IDLE

空闲状态，当在该状态的时候，传输完成信号 Trigger_Done 为低电平，当产生开始信号的，进入等待触发状态，该状态的代码如下所示：

```
IDLE: begin
    Trigger_Done <= 0;
    if(Start_Signal)
        State <= WAIT;
    else
        State <= IDLE;
```

```
end
```

当产生启动开始信号 Start_Round_Acq 之后，将开始信号 Start_Signal 拉高，进入等待触发状态之后，将该信号拉低，否则保持不变，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if(!Rst_n)
        Start_Signal <= 0;
    else if(Start_Round_Acq)
        Start_Signal <= 1;
    else if(State == WAIT)
        Start_Signal <= 0;
    else
        Start_Signal <= Start_Signal;
end
```

2. WAIT

当处于等待触发状态的时候，如果上一次 ADC 输出数据的平均值 ADC_Average_Val_Pre 小于触发值 Acq_Trigger_Value 同时本次 ADC 输出数据的平均值 ADC_Average_Val 大于触发值 Acq_Trigger_Value，此时代表触发完成，将触发完成信号 Trigger_Done 拉低，并且进入空闲状态，该状态的代码如下所示：

```
WAIT: begin
    if((ADC_Average_Val_Pre<=Acq_Trigger_Value)&&(ADC_Average_Val>=
Acq_Trigger_Value)) begin
        Trigger_Done <= 1;
        State <= IDLE;
    end else
        Trigger_Done <= 0;
    end
```

在本次设计中，设置的一轮采集的长度 ONE_ROUND_LENGTH 为 1024，也就是 1024 个 ADC 输出的数据个数，根据采集长度，设计一个 Data_Valid，当产生触发完成信号 Trigger_Done 之后，将 Data_Valid 信号拉高，当计数器的值达到采集长度的时候，将 Data_Valid 信号拉低，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        Data_Valid <= 0;
    else if(Trigger_Done)
        Data_Valid <= 1;
    else if(Cnt >= ONE_ROUND_LENGTH)
        Data_Valid <= 0;
end
```

当 Data_Valid 为高电平的时候，并且 Acq_Valid 信号有效的情况下，计数器 Cnt 进行计数，当计数值达到 1024 个时候，将计数器清零，然后重新计数，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        Cnt <= 0;
    else if(Data_Valid)
        begin
            if(Cnt >= ONE_ROUND_LENGTH)
                Cnt <= 0;
            else if(Acq_Valid)
                Cnt <= Cnt + 1;
            else
                Cnt <= Cnt;
        end
    else
        Cnt <= 0;
end
```

最终当 Data_Valid 有效并且 Acq_Valid 有效的情况下，输出传输有效信号 t_valid，代码如下所示：

```
assign t_valid = Data_Valid && Acq_Valid;
```

1.3.1.2.6 ADC_Acq2DDR_IP_v1_0_S00_AXI

该模块的基本结构如图 1-16 所示：

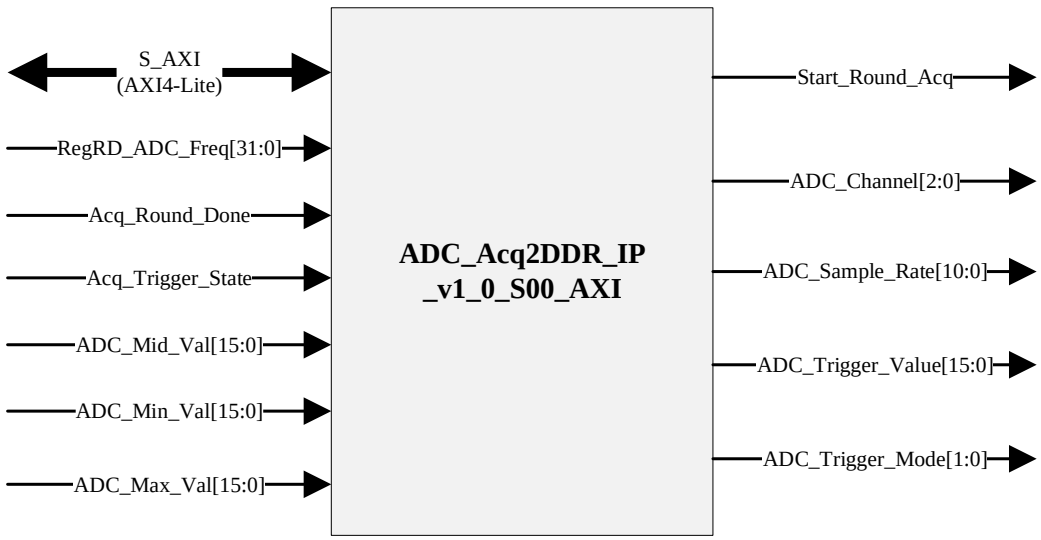

图 1-16 ADC_Acq2DDR_IP_v1_0_S00_AXI 模块基本结构图

用户可以通过 AXI4-Lite 总线，配置该模块内部 ADC 相关寄存器。模块会根据寄存器中对应位的值，结合自身内部逻辑代码，产生 AD7606 启动采样、采

样通道选择、采样速率、触发值以及触发模式等控制信号，输出给其他模块。

同时，其他模块输入的采样最大值、最小值、中值等特殊值，都会由该模块通过 AXI4-Lite 总线传输到 PS 端，最终经过转换后，用于 TFT 屏显示。

该模块的各个接口信号功能如表 1-7 所示：

表 1-7 模块接口信号功能

信号名称	I/O	信号意义
S_AXI_xxx (AXI4-Lite)	IO	AXI4-Lite 总线，用于配置模块内部 ADC 相关寄存器以及传输测量得到的最大值、最小值、中值等特殊值
RegRD_ADC_Freq[31:0]	I	AD7606 采集信号频率值
Acq_Round_Done	I	AD7606 一轮采集完成信号
Acq_Trigger_State	I	AD7606 触发状态控制信号
ADC_Mid_Val[15:0]	I	AD7606 采集信号电压的中值
ADC_Min_Val[15:0]	I	AD7606 采集信号电压的最小值
ADC_Max_Val[15:0]	I	AD7606 采集信号电压的最大值
Start_Round_Acq	O	AD7606 开始一轮开始标志信号
ADC_Channel[2:0]	O	AD7606 通道选择信号
ADC_Sample_Rate[10:0]	O	AD7606 采样分频值
ADC_Trigger_Value[15:0]	O	AD7606 触发有效信号
ADC_Trigger_Mode[1:0]	O	AD7606 触发模式信号

其中，模块内部的 ADC 相关寄存器一共有 4 个，每个寄存器位宽为 32 位，分别为 slv_reg0~slv_reg3，偏移地址分别为 0x00、0x04、0x08、0x0C。四个寄存器中我们只使用了其中的 slv_reg0，该寄存器的对应位描述如表 1-8 所示：

表 1-8 slv_reg0 寄存器位描述

寄存器	位	功能描述
slv_reg0 (偏移地址 0x00)	bit[31:30]	用于产生 AD7606 触发模式信号 ADC_Trigger_Mode
	bit[29:14]	用于产生 AD7606 触发有效信号 ADC_Trigger_Value
	bit[13:3]	用于产生 AD7606 采样分频信号 ADC_Sample_Rate
	bit[2:0]	用于产生 AD7606 通道选择信号 ADC_Channel

用户通过 AXI4-Lite 总线读写这四个寄存器，对于写操作，逻辑部分的代码实现主要如下：

```
assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready &&
S_AXI_AWVALID;

always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          slv_reg0 <= 0;
          slv_reg1 <= 0;
          slv_reg2 <= 0;
          slv_reg3 <= 0;
          Start_Round_Acq <= 0;
```

```
        end
      else begin
        if (slv_reg_wren)
          begin
            case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
              2'h0:
              begin
                Start_Round_Acq <= 1;//在寄存器 0 被写入时，启动一轮采集
                for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )

                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
                    // Slave register 0
                    slv_reg0[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              end
              2'h1:
                for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
                    // Slave register 1
                    slv_reg1[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h2:
                for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
                    // Slave register 2
                    slv_reg2[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h3:
                for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
                    // Slave register 3
```

```
                    slv_reg3[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                end
              default : begin
                          slv_reg0 <= slv_reg0;
                          slv_reg1 <= slv_reg1;
                          slv_reg2 <= slv_reg2;
                          slv_reg3 <= slv_reg3;
                        end
            endcase
          end
        else
            Start_Round_Acq <= 0;//写完后清除信号
      end
  end
```

这里的 S_AXI_WDATA 为用户通过 AXI4-Lite 总线写入的数据。可以看到，当写通道握手成功之后，逻辑代码会根据写地址的 bit[3:2]来判断所需写入的寄存器。随后，数据以字节为单位，根据写选通信号 WSTRB 的值，将有效数据写入寄存器的对应位中。

接着是控制信号的产生。写入 slv_reg0 寄存器的数据会根据对应位的值，被作为相关控制信号。相关代码实现如下：

```
assign ADC_Channel = slv_reg0[2:0];
assign ADC_Sample_Rate = slv_reg0[13:3];
assign ADC_Trigger_Value = slv_reg0[29:14];
assign ADC_Trigger_Mode = slv_reg0[31:30];
```

最后是寄存器的读取操作，用户通过 AXI4-Lite 总线，读取出其他模块测量出的最大值、最小值、中值等特殊值。相关代码实现如下：

```
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
    2'h0   : reg_data_out <= slv_reg0;
    2'h1   : reg_data_out <= {ADC_Max_Val,ADC_Min_Val};
    2'h2   : reg_data_out <=
{Acq_Round_Done,Acq_Trigger_State,ADC_Mid_Val};
    2'h3   : reg_data_out <= RegRD_ADC_Freq;
    default : reg_data_out <= 0;
    endcase
end

// Output register or memory read data
```

```
always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
    begin
        axi_rdata  <= 0;
    end
    else
    begin
        // When there is a valid read address (S_AXI_ARVALID) with
        // acceptance of read address by the slave (axi_arready),
        // output the read dada
        if (slv_reg_rden)
        begin
            axi_rdata <= reg_data_out;     // register read data
        end
    end
end
```

可以看到，代码中使用读地址（4 个 slv_reg 寄存器偏移地址）的 bit[3:2]作为条件，来判断待读出的值，并赋值给 reg_data_out。当读使能后，reg_data_out 中的值便会通过 AXI4-Lite 的读通道传输给 ARM 端处理。

通过这种方式，我们巧妙的利用寄存器的偏移地址，使用其他模块输入的测量信号代替 slv_reg1~slv_reg3 寄存器输出。这样，用户只需遵循 AXI4-Lite 总线协议，通过使用这四个寄存器地址，便能读取到测量模块采集到的特殊值。

至此，ADC_Acq2DDR_IP 中涉及的模块都已经介绍完成，对于顶层中对于各个模块的例化连接这里就不再进行说明，读者请自行查看源代码。了解完 ADC_Acq2DDR_IP 核的结构及实现原理，接下来我们继续硬件逻辑系统的设计，开始在设计中添加 IP 核并进行配置。

1.3.1.3IP 核添加与配置

打开 Block Design 点击“+”或者 Ctrl+i 添加 IP 核，本次所需添加 IP 具体 I 如下：

1、ADC_Acq2DDR_IP_v1.0

在添加栏中搜索并添加该 IP 核，如图 1-17 所示。

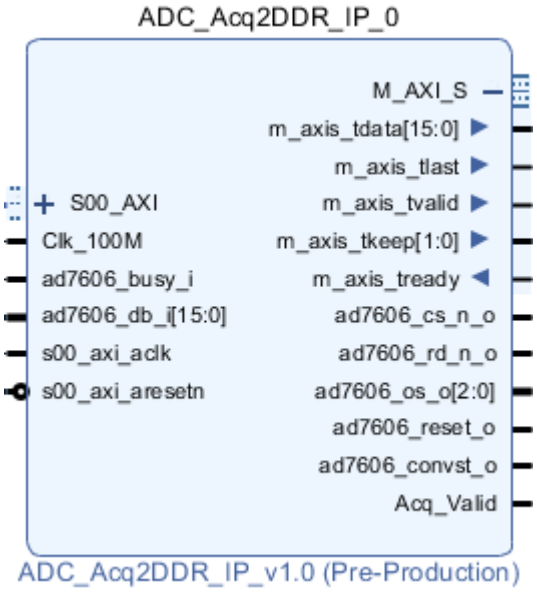

图 1-17　ADC_Acq2DDR_IP_v1.0

这里该 IP 核保持默认配置即可，如图 1-18 所示：

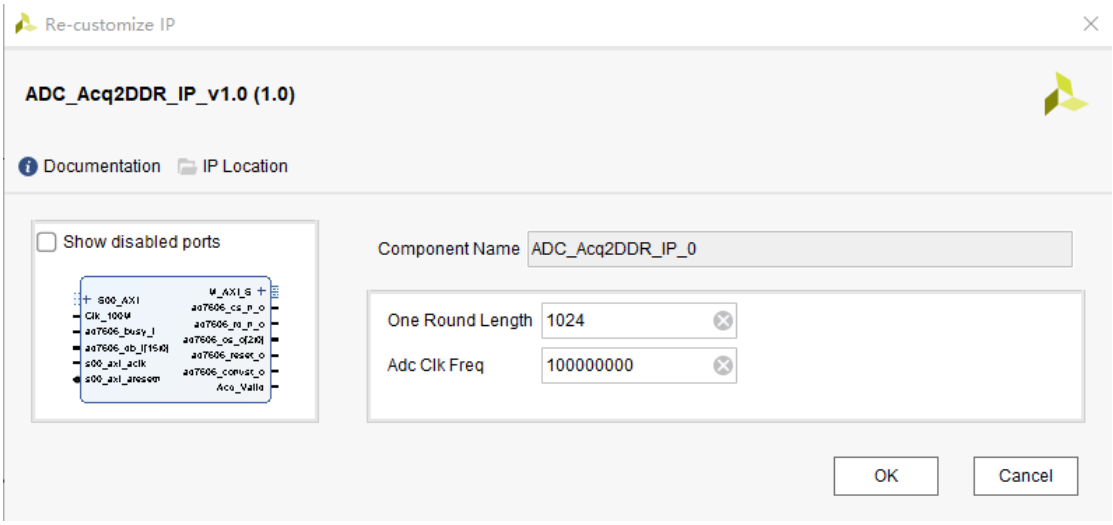

图 1-18　ADC_Acq2DDR_IP_0 核配置

2、RAM-based Shift Register

在搜索栏搜索并添加该 IP 核，添加后该核如图 1-19 所示：

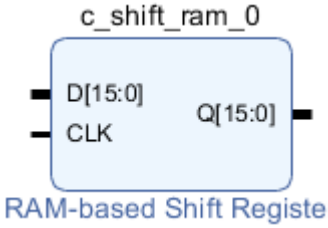

图 1-19　RAM-based Shift Register 寄存器

该核是一个非常高效的多位宽移位寄存器，用于 FIFO 类应用或作为延迟线。

用户可以通过该 IP 核创建固定长度或可变长度的移位寄存器。

双击打开该 IP 核的配置界面，本次设计配置如图 1-20 所示：

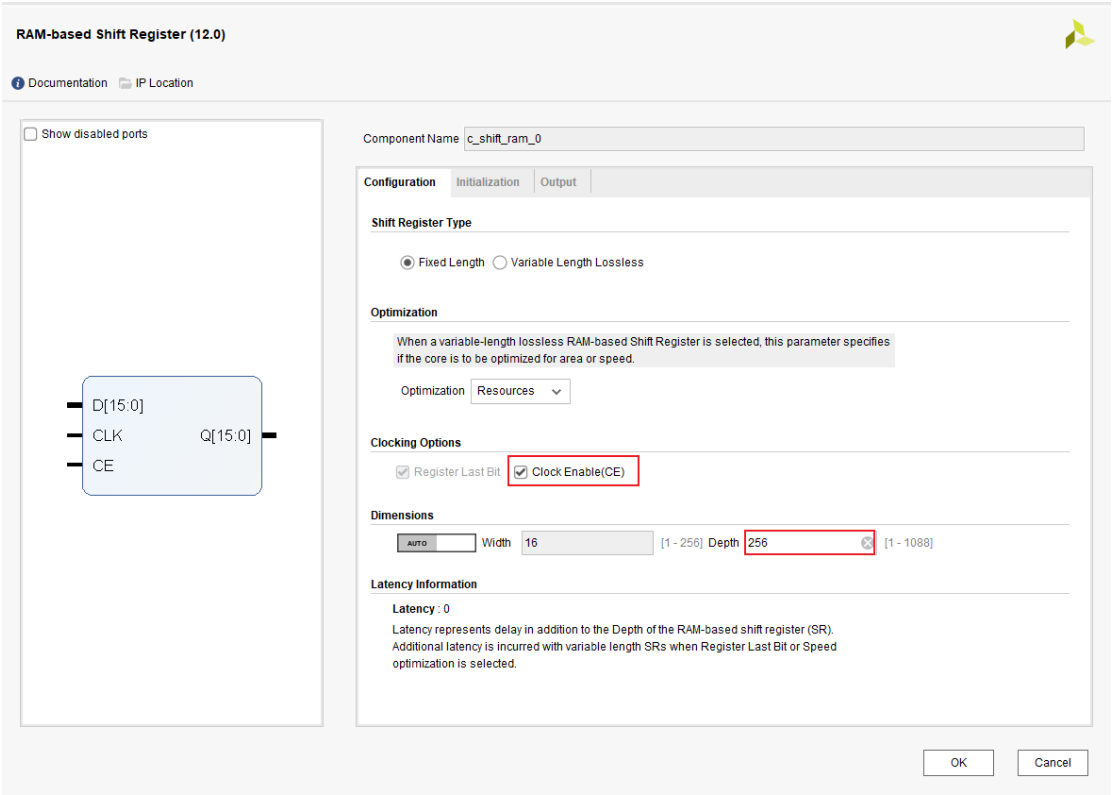

图 1-20　配置 IP 核

对于本次设计，ADC 每单位时间内会采集 1024 个数据，这些数据将会转换为图像数据显示。而 LCD 屏的分辨率为 800*480，为了方便显示，屏幕中最多只会显示一半的数据，而另外的数据用户则可以通过拖动滑块查看。为了使达到触发条件的波形图像数据能够显示在屏幕中央，我们需要将数据延时 1/4 个屏幕的数据量到来，也就是 256 个数据。因此这里将延时值设置为 256。

3、AXI4-Stream Data FIFO

在添加栏中搜索 AXI4-Stream Data FIFO 即可看到该 IP 核，双击添加，添加完成后打开 IP 核，对该 IP 核的常规项进行配置。首先是 FIFO 的深度，本次设计每个波形图像对应 512 个 32 位的 ADC 数据，即 512*4=2048 字节这里为了防止以后对工程修改时，FIFO 深度对工程造成影响，因此设置为了 4096（实际情况下读写是同时进行的，及时设置为 1024 也是可以的）；随后设置 TDATA 数据位宽为 4；最后使能 TKEEP 和 TLAST 信号点击 OK 即可完成配置。对 IP 核的配置操作如图 1-22 所示。

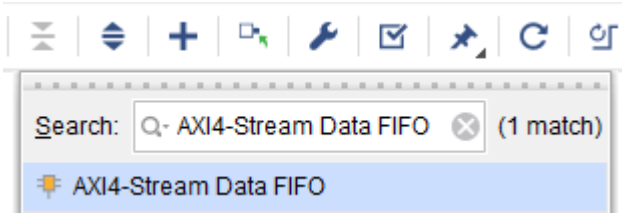

图 1-21　添加 IP 核

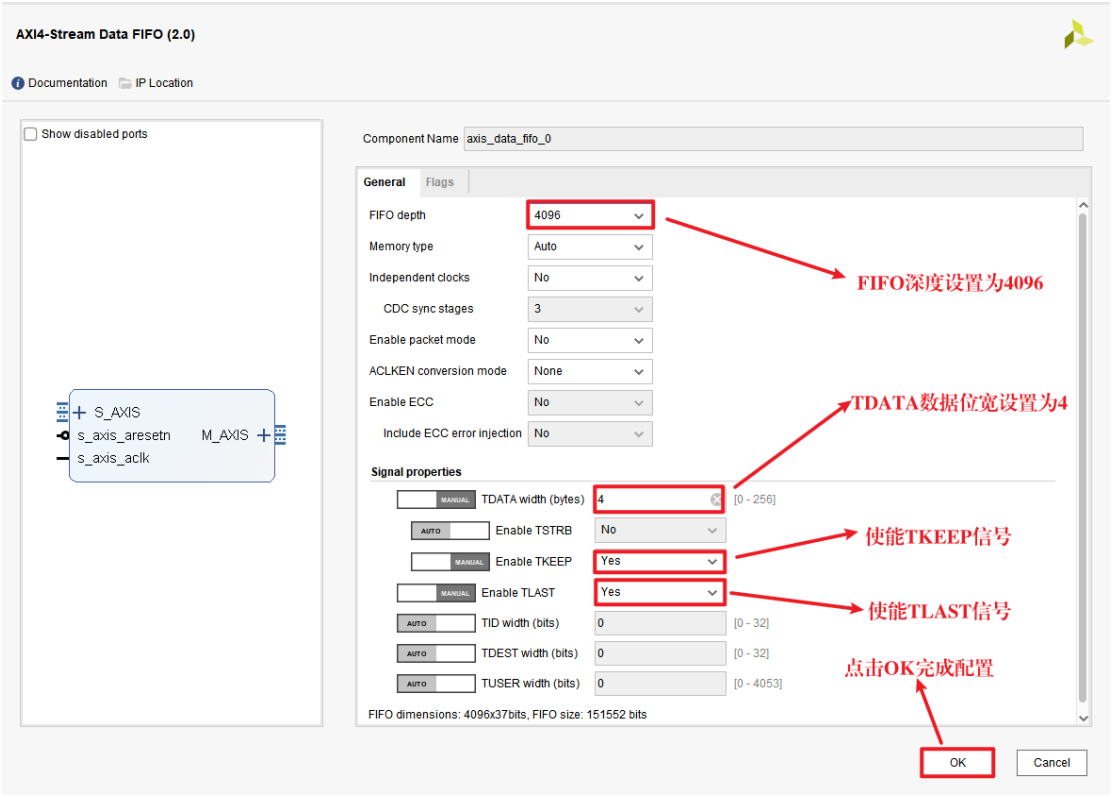

图 1-22　配置 IP 核

这里对 TDATA width、Enable TSTRB、Enable TKEEP、Enable TLAST 选项进行介绍，各个选项作用如下：

- TDATA width：该项用于设置 AXI4-Stream 接口上 TDATA 信号的宽度，单位为 BYTE，可以在 0 到 512 间变化，当该信号被忽略时，TKEEP 和 TSTRB 信号也会被忽略。
- Enable TKEEP：用于使能 TKEEP 信号，仅当 TDATA Width 大于 0 时，才可使能。TKEEP 信号用于提供辅助信息，当该信号为 1 时，代表该值有效，当该值为 0 时代表该值无效，该信号通常用于需要进行数据宽度转换时。
- Enable TSTRB：用于使能 TSTRB 信号，仅当 TDATA Width 大于 0 时，才可使能。TSTRB 信号也是用于提供辅助信息，当该信号为 1 时，表示

对应的数据有效。TSTRB 为 0 时，表示对应数据是占位数据，position data。在有的应用中，position data 可能被作为 sideband data 来使用。TSTRB 信号和 TKEEP 信号一起构成了对 TDATA 的描述，表明当前传输的数据是有效、或是占位符或无效。

- Enable TLAST：用于使能 TLAST 信号，对于 STREAM FIFO 来说，TLAST 信号的作用是指示一次传输数据流的最后一个数据，也指示着该数据流的结束。其会记录下 TLAST 信号的位置，如果 SLAVE 接口（SFIFO 的数据写入接口）的某一个数据写入的同时 TLASET 信号也为高的话，TLASET 信号会被记录。在 MASTER 接口（SFIFO 的数据读出接口）读出该数据的同时， TLAST 信号也会被拉高。总结起来就是，进的数据有 TLAST，该数据出的时候就会有 TLAST。

其他项由于本次并未使用到，这里暂不做介绍，想要了解的用户可以自行查阅该 IP 核的官方手册 pg085。

配置完成后该 IP 核如图 1-23 所示。该 FIFO 用来缓存 ACM7606 采集到的数据，等待 DMA 将数据搬运到 DDR 中。

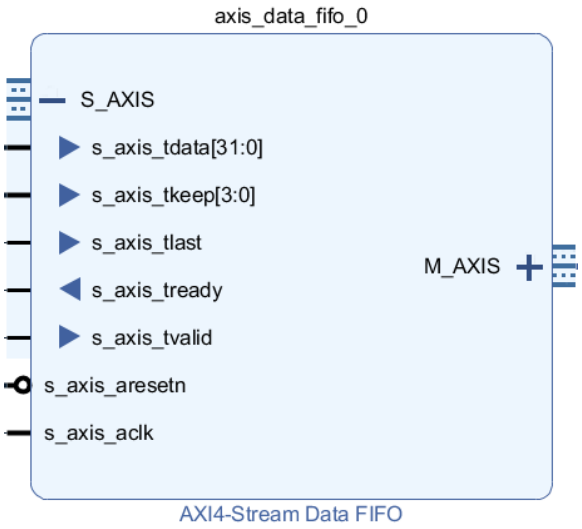

图 1-23　AXI4-Stream Data FIFO IP 核

4、AXI Direct Memory Access

在添加栏中搜索 AXI Direct Memory Access 即可看到该 IP 核，双击添加，添加完成后打开 IP 核进行配置。该 IP 核需要配置的地方有两处：取消 scatter gather engine 的使能以及将缓冲区长度寄存器位宽设置为 13 位，其余部分保持默认配置。

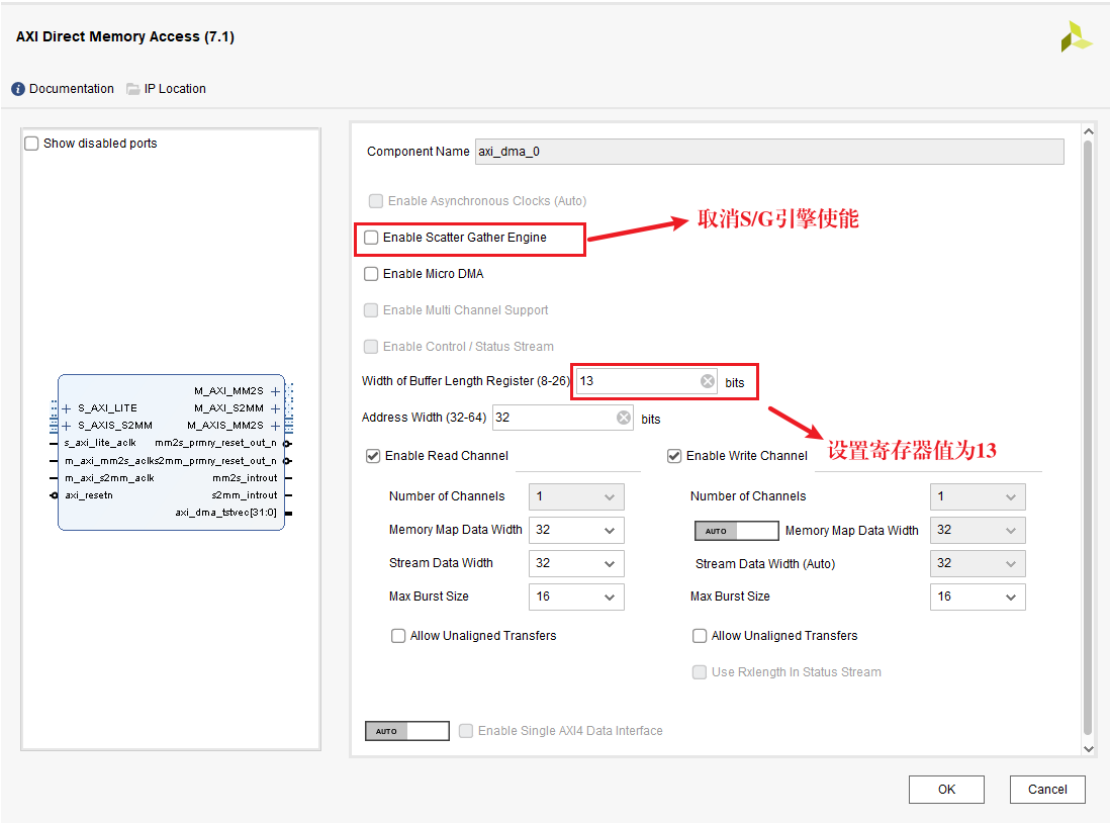

图 1-24　配置 IP 核

这里对这两个选项进行解释：

- Enable Scatter Gather Engine：勾选后会启用 S/G 模式，并在 AXI DMA 中包含 S/G 驱动。取消后将启用直接寄存器模式，此时 S/G 引擎所有的输出端口都被绑定到零，且所有输入端口保持打开状态。
- Width of Buffer Length Register：该项用来设置在 S/G 描述符中传输的控制字段缓冲区长度和状态字段字节的有效位数。对于直接寄存器模式，它指定 MM2S_LEGNTH 和 S2MM_LEGNTH 寄存器中的有效位数，即最大传输量。本次设计每次采集 512 个数据，所以最大传输量为 512x4=2048，即需要 12 位。为了防止意外，这里我们将寄存器的值设置为 13，也就是能容纳 8192 位数据。

该 IP 核对应的手册为 pg201，在 ACZ702 开发板资料中有提供，对应路径为：*小梅哥 ACZ702 型 Zynq 开发板资料\盘 A_ACZ702 开发板标准配套资料\10_Xilinx 原厂文档*。这里我们只对修改的两项进行了介绍，对于其他项，用户可以自行查阅对应手册。

配置完成后的 IP 核如图 1-25 所示，该 IP 主要起通信“桥梁”的作用，可以将所有外设映射的寄存器“连接”起来，这样就可以高速访问各寄存器。其传输不受 CPU 的支配，传输还是双向的；例如，从“表面”上看，它可以将 flash 中的数

据与储存器中变量建立通讯，还可以将一外设的寄存器或缓冲器与另外设的寄存器或缓冲器建立双向通讯。由于该 IP 核的通讯不会占用 CPU 资源，且传输速度较快，因此可以有效节省 CPU 资源。

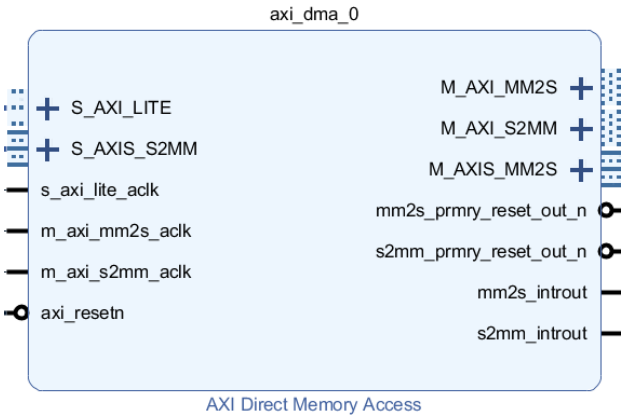

图 1-25　DMA IP 核

5、zynq 核配置

zynq IP 核在 RGB_LCD 工程中已经被创建，因此本次只需对该 IP 配置进行修改即可。在 TFT_LCD 工程基础上，我们需要使能 SD0（MIO40...45）用于存储我们从内存中截取的图像数据；使能 I2C0（MIO50...51）用于配置触摸屏以及读取触摸数据；使能 MIO 以及 EMIO 用于截图以及触摸屏复位等功能。

除了这些之外，为了方便设计我们还需要勾选 PL 的中断使能，具体修改如图 1-26 和图 1-27 所示：

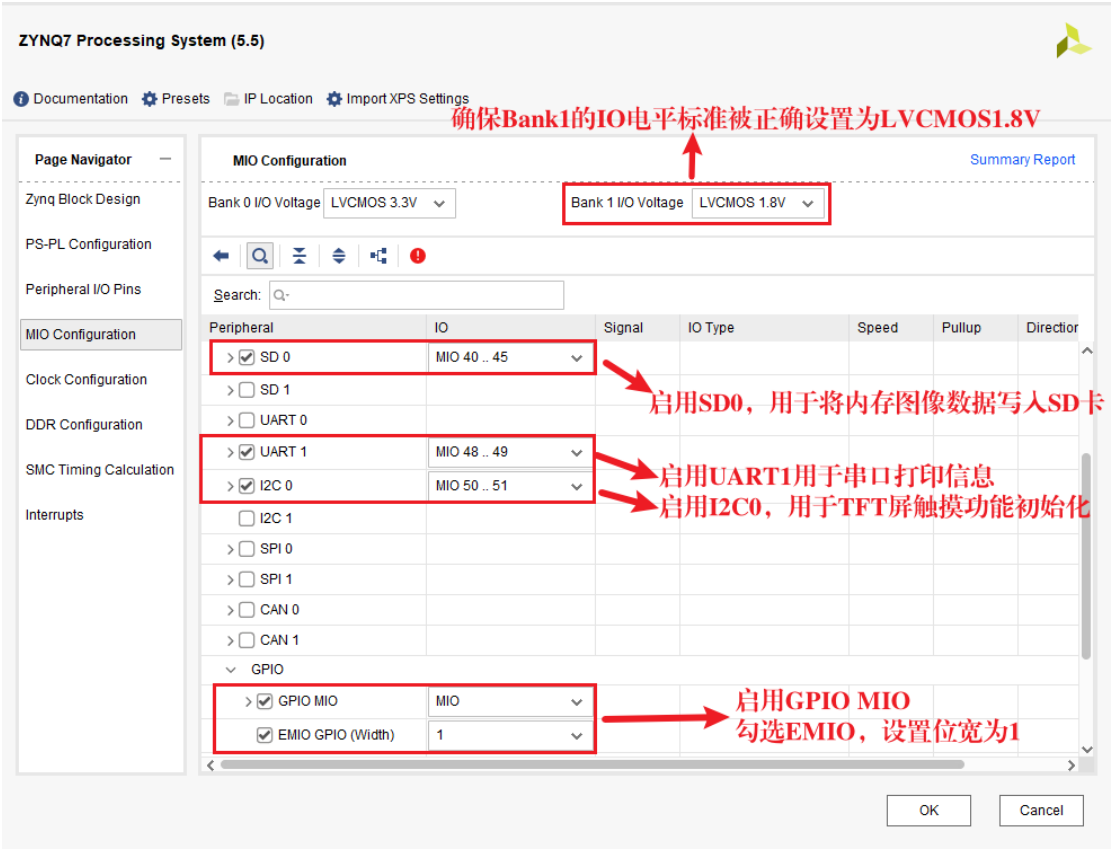

图 1-26　配置 IP 核

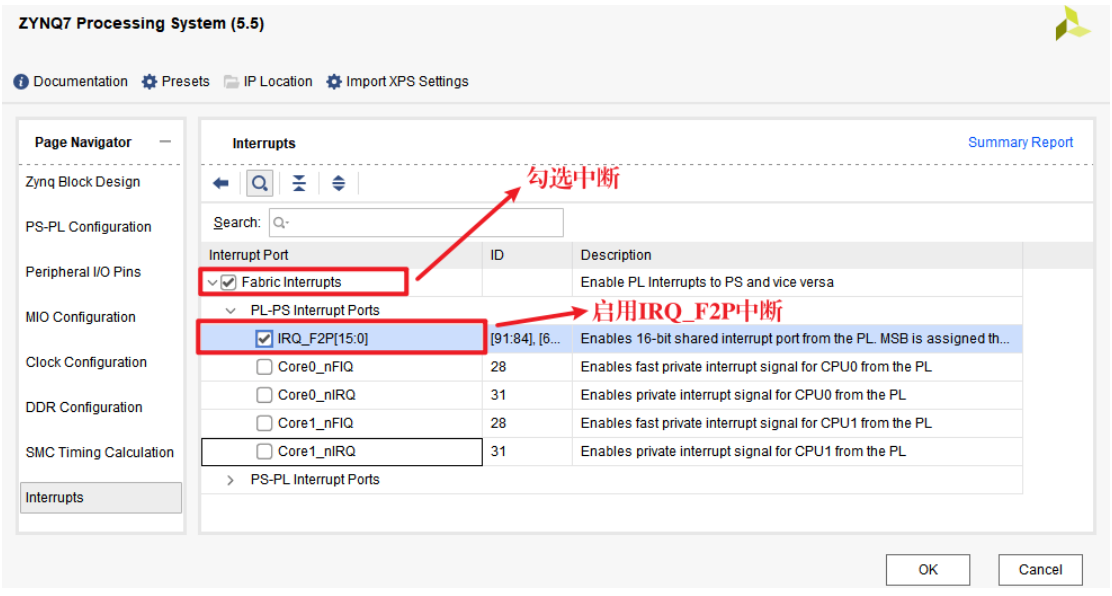

图 1-27　配置 IP 核

配置完成后点击 OK，此时我们会发现 zynq IP 核会多出来一个引脚，将其导出，并修改命名为 TP_Rst。

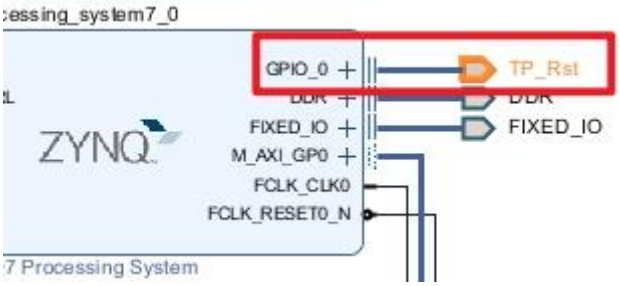

图 1-28　修改引脚名

该步骤完成后，IP 核的添加与配置也已经完成，接下来即可开始对模块进行连接。

1.3.1.4端口连接

首先点击“Run Connection Automation”让软件连接部分端口，如图 1-29 所示：

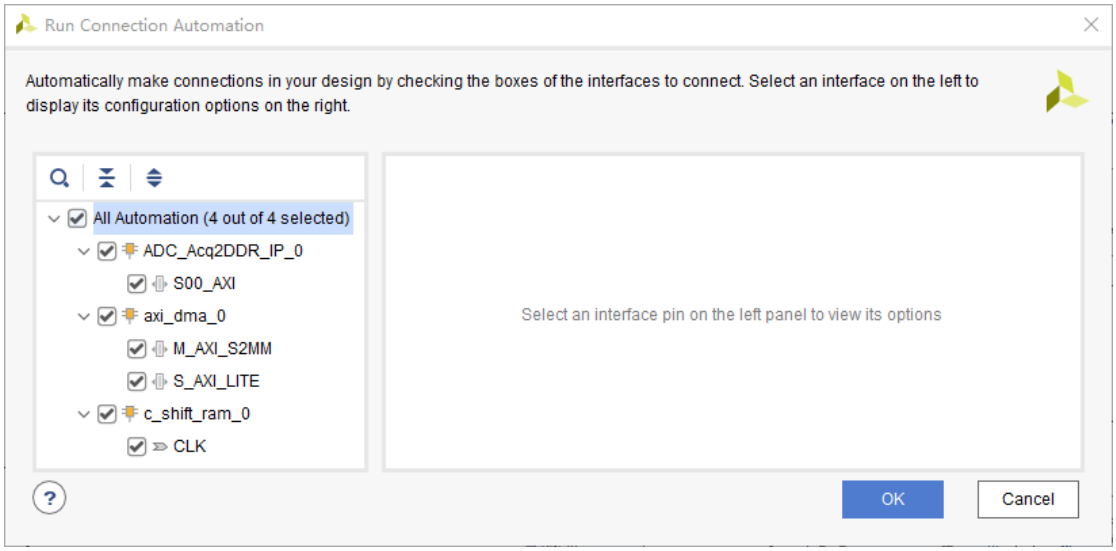

图 1-29　运行自动连接

此时的硬件逻辑系统如图 1-30 所示:

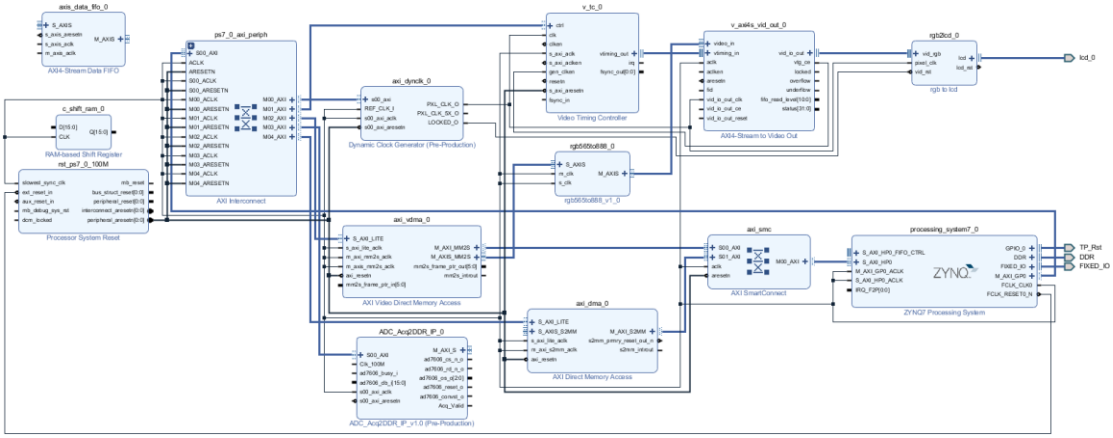

图 1-30　硬件逻辑系统

接下来我们需要手动连接其余部分端口，具体连接方式如下：

ADC_Acq2DDR_IP

① 导出 ad7606_busy_i、ad7606_db_i[15:0]、ad7606_cs_n_o、ad7606_rd_n_o、ad7606_os_o[2:0]、ad7606_reset_o、ad7606_convst_o 接口引脚

② 将 M_AXI_S 接口展开

③ m_axis_tdata[15:0] → D[15:0] (c_ shift_ram_0)

④ m_axis_tlast → s_axis_tlast (axis_data_fifo_0)

⑤ m_axis_tvalid → s_axis_tvalid (axis_data_fifo_0)

⑥ m_axis_tkeep[1:0] → s_axid_tkeep[1:0] (axis_data_fifo_0)

⑦ m_axis_tready → s_axid_tready (axis_data_fifo_0)

⑧ Acq_Valid → CE (c_ shift_ram_0)

⑨ Clk_100M → FCLK_CLK0 (ZYNQ)

⑩ s00_axi_aresetn → peripheral_aresetn[0:0] (rst_ps7_0_100M)

axis_data_fifo_0

① s_axis_tdata[15:0] → Q[15:0] (c_ shift_ram_0)

② s_axis_aresetn → peripheral_aresetn[0:0] (rst_ps7_0_100M)

③ M_AXIS → S_AXIS_S2MM (axi_dma_0)

④ m_axis_aclk → FCLK_CLK0 (ZYNQ)

⑤ s_axis_aclk → FCLK_CLK0 (ZYNQ)

axi_dma_0

① s2mm_introut → IRQ_F2P[0:0]

其中箭头左边为该模块内的信号，箭头所指的为所要连接的信号，箭头仅代表连接，不代表信号的方向，括号内为该信号所在的模块。连线完成后系统整体框图如图 1-31 所示。如果硬件逻辑系统图片不够清晰，用户可以点击图片跳转到网页查看。

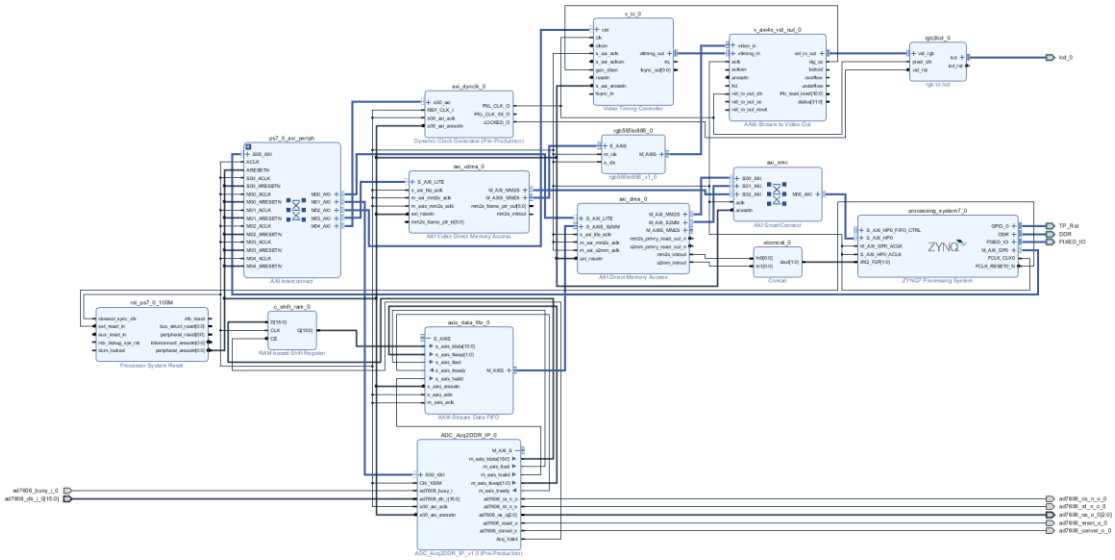

图 1-31　系统框图

接着点击 √验证设计是否正确，确认设计无误后封装完即可进行管脚约束。

1.3.1.5管脚约束

本次设计引脚约束表如表 1-9 所示，这里灰色部分为 RGB_TFT 章节中已经分配好的引脚。

表 1-9　管脚分配表

Pin Name	Signial Name	Pin NO.		Pin Name	Signial Name	Pin NO.
Display_R4	lcd_0_data[15]	W20		ad_data[15]	ad7606_db_i_o[15]	K18
Display_R3	lcd_0_data [14]	W19		ad_data[14]	ad7606_db_i_o[14]	H17
Display_R2	lcd_0_data [13]	V17		ad_data[13]	ad7606_db_i_o[13]	J18
Display_R1	lcd_0_data [12]	V16		ad_data[12]	ad7606_db_i_o[12]	K19
Display_R0	lcd_0_data [11]	T15		ad_data[11]	ad7606_db_i_o[11]	H18
Display_G5	lcd_0_data [10]	V20		ad_data[10]	ad7606_db_i_o[10]	J19
Display_G4	lcd_0_data [9]	U17		ad_data[9]	ad7606_db_i_o[0]	J20
Display_G3	lcd_0_data [8]	V18		ad_data[8]	ad7606_db_i_o[8]	H16
Display_G2	lcd_0_data [7]	T16		ad_data[7]	ad7606_db_i_o[7]	H20
Display_G1	lcd_0_data [6]	R16		ad_data[6]	ad7606_db_i_o[6]	G18
Display_G0	lcd_0_data [5]	U19		ad_data[5]	ad7606_db_i_o[5]	G20
Display_B4	lcd_0_data [4]	Y19		ad_data[4]	ad7606_db_i_o[4]	G19
Display_B3	lcd_0_data [3]	W18		ad_data[3]	ad7606_db_i_o[3]	G17
Display_B2	lcd_0_data [2]	Y18		ad_data[2]	ad7606_db_i_o[2]	E19
Display_B1	lcd_0_data [1]	W16		ad_data[1]	ad7606_db_i_o[1]	F17
Display_B0	lcd_0_data [0]	Y17		ad_data[0]	ad7606_db_i_o[0]	D20
Display _PCLK	lcd_0_pclk	U15		ad_os[2]	ad7606_os_o_0[2]	G14
Display _HSYNC	lcd_0_hs	U14		ad_os[1]	ad7606_os_o_0[1]	L15
Display _VSYNC	lcd_0_vs	W14		ad_os[0]	ad7606_os_o_0[0]	K14
Display _DE	lcd_0_de	W15		ad_busy	ad7606_busy_i_o	E17
Display _BL	lcd_0_bl	R17		ad_convstab	ad7606_convst_o_0	J14
TP_RST	TP_Rst_tri_io[0]	F19		ad_cs	ad7606_cs_n_o_0	D18

				ad_rd	ad7606_rd_n_o_0	H15
				ad_rsset	ad7606_reset _o_0	F16

管脚约束完成后按住 Ctrl+S 保存，接下来就可以生成比特流并导出硬件运行 SDK 了。由于本次设计是基于 RGB_LCD 工程创建的，为了避免在 SDK 中创建的工程收 RGB_LCD 硬件的影响。在导出硬件前，我们要先将工程的.sdk 文件夹删除再导出硬件然后才能 launch SDK。

1.3.2 CPU 软件程序设计

首先还是新建一个 SDK 工程，本次设计代码较为复杂，不适合教用户如何一步步实现，因此这里建议直接从例程中拷贝已经设计完成的代码。

本次例程名为 ACM7606_Scope，提供于开发板资料包中，相对路径如下：*小梅哥 ACZ702 型 Zynq 开发板资料\盘 A_ACZ702 开发板标准配套资料\02_设计实例\03_【裸机例程】基于 C 编程的 Zynq 裸机例程*。打开例程 sdk 文件夹下的 ACM7606_Scope 文件夹，将 ACZ702_Lib 文件夹拷贝到创建的工程中，将 src 文件夹下的.c 和.h 文件拷贝到工程的 src 文件夹下，并添加它们的头文件路径，完成后工程资源目录如图 1-32 所示：

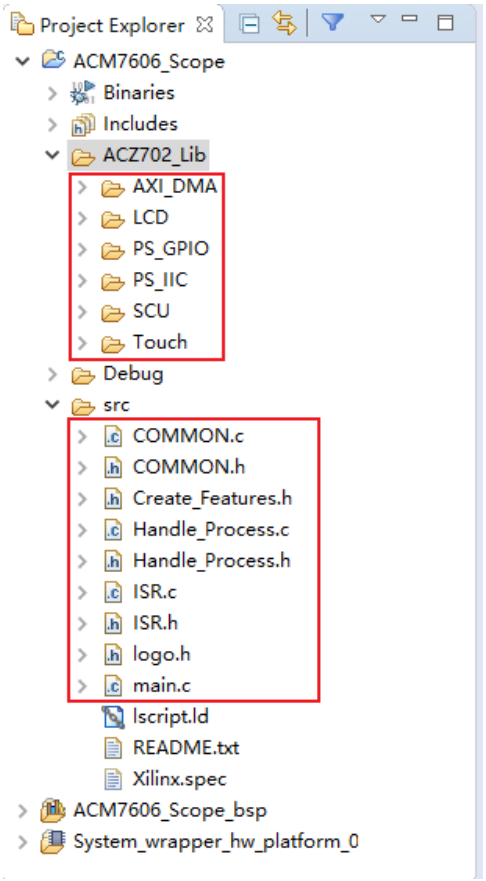

图 1-32 工程资源目录

ACZ702_Lib 文件夹下为本次设计所需要使用到的相关应用库，src 下为本次

设计相关文件，各个文件的功能如表 1-10 所示：

表 1-10　设计相关文件说明

文件	描述
COMMON.c	用于存放用户的一些通用宏定义和参数的源文件
COMMON.h	包含本次所使用到的相关头文件以及通用全局变量声明
ISR.c	中断处理函数源文件，本次设计对应私有定时器中断处理函数、GPIO 中断处理函数与 AXI DMA 接收中断处理函数。
ISR.h	中断处理函数头文件，包含对中断处理函数的声明以及相关标志的全局修饰
main.c	主函数源文件
logo.h	log 文件，替换该文件即可替换界面 log
Create_Features.h	属性创建头文件，对按键、文本、波形的属性进行设置，包括颜色、字体大小、对应坐标、长度、宽度等
Handle_Process.c	进程处理源文件，包含本次设计中所需要使用到的各项进程，如界面绘制、波形更新、触发处理等
Handle_Process.h	进程处理头文件，对进程处理源文件所需要的函数、标志进行声明

本次设计我们需要对 SD 卡进行写操作，因此我们需要右键板级支持包进行设置，让支持包包含 xilffs 库，以使用 FatFs 中各种 API 来操作 SD 卡，如图 1-33 所示：

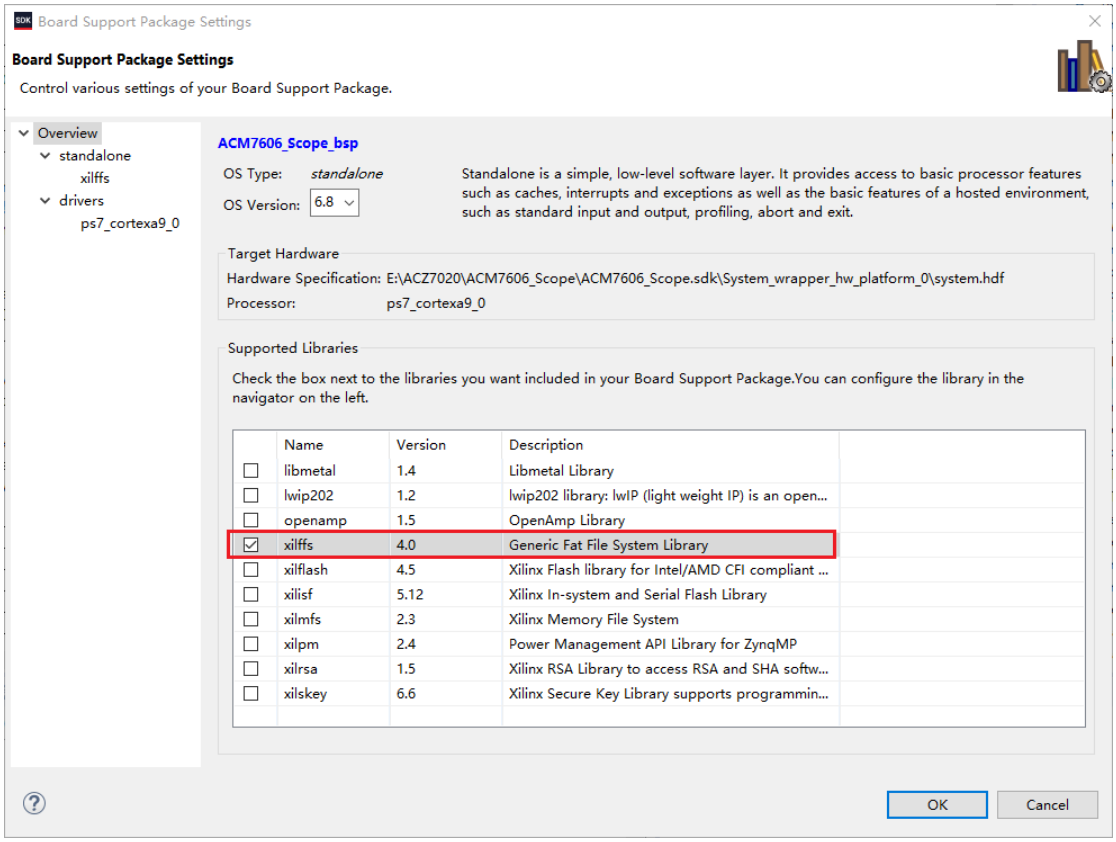

图 1-33　包含 xilffs 库

设置完成后重新编译 bsp，便完成了 SDK 部分工程的创建。接下来我们双击打开 main.c 文件，查看本次设计

打开 main.c 文件，其中代码如下：

```
#include "COMMON.h"

void Handle_Events(void);

//主函数
int main(void)
{
    Init_Homepage();//初始化主页和控制器

    //循环处理事件
    while(1) {
        Handle_Events();
    }
    return 0;
}

void Handle_Events(void)
{
    //如果开启了单次触发，触发成功则 STOP，取消触发则恢复原状
    if(Single_TriggerFlag && (ADC_ROUND_DONE || Cancel_Trigger)) {
        Single_TriggerFlag = 0; //清标志
        Handle_Single_Trigger();//处理单次触发事件
    }
    //每轮传输完成则刷新 DCache 并开启下一轮传输
    if(ADC_ROUND_DONE && Wave_Run && (!Single_TriggerFlag)) {
        Handle_Round_Done();
    }
    //定时刷新波形窗口，10ms 一次
    if(Flag_DrawWave) {
        Flag_DrawWave = 0;
        Refresh_WaveWindow();   //刷新波形窗口
    }
    //定时检测触摸，30ms 一次
    if(Flag_TouchScan) {
        Flag_TouchScan = 0;
        Touch_Scan();                   //触摸扫描
    }
    //定时刷网格背景，20ms 一次
    if(Flag_DrawGrid) {
        Flag_DrawGrid = 0;
        Draw_Grid_Background(ADC_Wave); //绘制网格背景
    }
    //定时刷测量值，500ms 一次
```

```
    if(Flag_Refresh_Val && Wave_Run) {
        Flag_Refresh_Val = 0;
        Refresh_Measure_Val();          //刷新测量值
    }
}
```

可以看到，主函数首先初始化主页和控制器，随后通过 While(1)循环处理事件 Handle_Events。当满足对应条件时便调用相关函数进行处理，这些被调用的函数全部定义于 Handle_Process.c 中，下面我们以其中几个为例，带大家一起看看函数的具体功能。

1.3.2.1初始化页面和控制函数

Init_Homepage 函数用来初始化页面和控制器，函数中主要包含 LCD 屏的初始化、I2C 的初始化、GPIO 及 GPIO 中断初始化、SD 卡挂载、触摸初始化、定时器初始化、以及一些背景、按键、显示波形窗口的绘制等，函数的实现的代码如下所示：

```
//初始化主页
void Init_Homepage()
{
    FRESULT rc;
    ScuGic_Init();  //初始化通用中断控制器
    LCD_Init();     //LCD 初始化
    GT9147_Init();  //GT9147 初始化
    //定时器初始化，每 10ms 触发一次
    ScuTimer_Int_Init(10000);
    //DMA 初始化，带中断功能
    AXI_DMA_Init(&AxiDma0,XPAR_AXIDMA_0_DEVICE_ID);
    AXI_DMA_RxInt_Init(&AxiDma0,XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR,AXI_DMARx
_IRQHandler);

    rc = f_mount(&fatfs, "0:/", 0);     //挂载 SD 卡
    if (rc != FR_OK) {
        printf("Mount Failed!\n");      //若挂载失败则发出警告
    }

    //GPIO 及 GPIO 中断初始化
    PS_GPIO_Init();
    PS_GPIO_Int_Init(PS_GPIO_IRQ_Handler);
    PS_GPIO_SetInt(47,XGPIOPS_IRQ_TYPE_EDGE_FALLING);

    LCD_Clear(LCD_BLACK);   //设置整体背景为黑色
    POINT_COLOR = LCD_WHITE;//笔画设定为白色
```

```
    BACK_COLOR = LCD_BLACK; //笔画背景设定为黑色

    LCD_ShowString(3,20,180,16,16,0,(uint8_t*)"www.corecourse.cn");//显示网址
    LCD_DisplayPic(20,50,gImage_logo);//画 LOGO

    Draw_Box(ADC_Wave.Window, LCD_WHITE, 0);
    Get_Waveform_Window_Parameters(&ADC_Wave);
    Draw_Waveform_Windows(ADC_Wave, Sample_Rate[Sample_Set]);//KHz

    Draw_Normal_Button(Button_RUN);
    Draw_Normal_Button(Button_AUTO);
    Draw_Normal_Button(Button_CH_INC);
    Draw_Normal_Button(Button_CH_DEC);
    Draw_Normal_Button(Button_SA_INC);
    Draw_Normal_Button(Button_SA_DEC);
    Draw_Normal_Button(Button_Tri_Mode);
    Draw_Normal_Button(Wave_Slider);
    Fill_Box(Slider,LCD_BLUE,5);

    sprintf(str,"%.3fV",Get_ADC_Vmid());
    strcpy(Button_TriggerVal.Text[0],str);
    Draw_Normal_Button(Button_TriggerVal);

    sprintf(str,"CH%d",CH_Set);
    Draw_Normal_Text(Text_CHANNEL, str);
    sprintf(str,"%dKHz",(int)Sample_Rate[Sample_Set]);
    Draw_Normal_Text(Text_SAMPLE, str);

    //先把 ADC 数据所处的内存清零
    memset(P_ADC_Data,0,ADC_DATA_LENGTH*2);

    //先采集一次
    Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
    //DMA 传输
    XAxiDma_SimpleTransfer(&AxiDma0,(UINTPTR) P_ADC_Data,
            ADC_DATA_LENGTH*2, XAXIDMA_DEVICE_TO_DMA);
    //测量点为 Y 轴线
     Measure_Point = ADC_Wave.Wave_Area.X1 + ADC_Wave.Wave_Area.Width/2;
    //触发点为 X 轴线
     Trigger_Point = ADC_Wave.Wave_Area.Y1 + ADC_Wave.Wave_Area.Height/2;
}
```

关于绘图和初始化 TFT、触摸的部分将不做介绍，下面将对该函数中比较重要的几个点进行介绍。

1. 定时器初始化

在函数中，使用了私有定时器，通过对定时器初始化，使其每 10ms 触发一次。通过计时器控制每个进程的触发间隔，启动对应的标志位，各个时间的标志位和各个进程标志位如下所示：

```
//各个时间标志位
static uint8_t Flag_20ms = 0;
static uint8_t Flag_30ms = 0;
static uint8_t Flag_50ms = 0;
static uint8_t Flag_100ms = 0;
static uint8_t Flag_200ms = 0;
static uint8_t Flag_500ms = 0;

//定义各个进程的标志位
uint8_t Flag_DrawWave = 0;      //绘制波形，10ms 一次
uint8_t Flag_DrawGrid = 0;      //绘制网格，20ms 一次
uint8_t Flag_TouchScan = 0;     //触摸扫描，30ms 一次
uint8_t Flag_Refresh_Val = 0;   //刷新电压和频率数值，500ms 一次
```

关于标志位的使用将在后面进行介绍，得到需要控制的标志位之后，在定时器中断函数中实现对标志位的控制，代码如下所示：

```
void ScuTimer_IRQ_Handler(void *CallBackRef)
{
    /* ↓↓↓用户处理↓↓↓ */
    //以下通过计时器控制每个进程的触发间隔，启动对应的标志位
    Flag_DrawWave = 1;

    if(Flag_20ms >= 1) {
        Flag_20ms = 0;
        Flag_DrawGrid = 1;
    }
    else
        Flag_20ms++;

    if(Flag_30ms >= 2) {
        Flag_30ms = 0;
        Flag_TouchScan = 1;
    }
    else
        Flag_30ms++;

    if(Flag_50ms >= 4) {
        Flag_50ms = 0;
```

```
    }
    else
        Flag_50ms++;

    if(Flag_100ms >= 9) {
        Flag_100ms = 0;

    }
    else
        Flag_100ms++;

    if(Flag_200ms >= 19) {
        Flag_200ms = 0;

    }
    else
        Flag_200ms++;

    if(Flag_500ms >= 49) {
        Flag_500ms = 0;
        Flag_Refresh_Val = 1;
    }
    else
        Flag_500ms++;


    /* ↑↑↑结束处理↑↑↑ */
    XScuTimer_ClearInterruptStatus(&ScuTimer);
}
```

2. DMA 初始化及第一轮传输

设计中 DMA 用来将 ACM9767 采集到的有效数据搬运到 DDR 中来，为了让初始化界面时能够显示出采集到的波形数据，我们就需要对 DMA 初始化，初始化完成后搬运采集到的有效数据，经过转化后用于显示。

代码中首先对 DMA 和 DMA 接收中断进行初始化，代码中的实现如下：

```
//DMA 初始化，带中断功能
AXI_DMA_Init(&AxiDma0,XPAR_AXIDMA_0_DEVICE_ID);
AXI_DMA_RxInt_Init(&AxiDma0,XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR,A
XI_DMARx_IRQHandler);
```

在 DMA 接收中断处理函数中会对中断事件进行判断，如果 DMA 接收错误便复位 DMA，重新使能 DMA 和 DMA 中断随后开始新一轮接收。如果接收成功便产生接收成功标志信号 RxDone，该信号的作用我们稍后再讲。这部分的代

码实现如下：

```
//DMA 的 Rx 传输的中断服务函数
void AXI_DMARx_IRQHandler(void *Callback)
{
    uint32_t IrqStatus;

    //读取挂起的中断
    IrqStatus = XAxiDma_IntrGetIrq(&AxiDma0, XAXIDMA_DEVICE_TO_DMA);

    //确认挂起的中断
    XAxiDma_IntrAckIrq(&AxiDma0, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    //如果没有发生中断，则不执行任何操作
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        return;
    }

    //如果发生错误则复位硬件，并重新启动传输
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        Error = 1;
        //printf("RX Error!\n");

        //复位
        XAxiDma_Reset(&AxiDma0);
        //等待复位完成
        while(!XAxiDma_ResetIsDone(&AxiDma0));
        //使能 DMA 的 Rx 中断
        XAxiDma_IntrEnable(&AxiDma0,
XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
        //启动新一轮传输
        XAxiDma_SimpleTransfer(&AxiDma0,(UINTPTR) P_ADC_Data,
                ADC_DATA_LENGTH*2, XAXIDMA_DEVICE_TO_DMA);

        return;
    }

    //如果完成中断被断言，将 RxDone 置 1
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

        RxDone = 1;
        //printf("RX Done!\n");
    }
}
```

初始化完 DMA 后，需要传输一次 ADC 采集的数据，为了确保数据的准确性，首先用 memset 将用来存储 ADC 数据的内存清零，随后发送指令控制 ADC 完成一次数据采集。接着再通过 DMA 将 stream 类型的 ADC 数据（stream 类型数据没有地址）存储到刚刚清零的地址中。相关代码实现如下：

```
//先把 ADC 数据所处的内存清零
memset(P_ADC_Data,0,ADC_DATA_LENGTH*2);

//先采集一次
Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value,
Trigger_Mode);
//DMA 传输
XAxiDma_SimpleTransfer(&AxiDma0,(UINTPTR) P_ADC_Data,
       ADC_DATA_LENGTH*2, XAXIDMA_DEVICE_TO_DMA);
```

3. 设置 ADC 相关模式以及获取特殊值

设定 ACM7606 工作模式以及获取特殊值，都是通过操作寄存器 slv_reg0~slv_reg3 实现。这些寄存器的地址以及相关控制函数定义于 ADC_Acq2DDR_IP 库中，四个寄存器对应地址以及位含义如表 1-11 所示：

表 1-11 ADC 相关寄存器

寄存器名称	寄存器地址	寄存器意义
REG_ADC_CONTROL	ADC_IP_BASEADDR+0x00	ADC 控制寄存器 [2:0] ADC 通道选择 Channel [13:3]ADC 采样分频 Sample_Div [29:14] ADC 触发电压值 Trigger_Val [31:30] ADC 触发模式 Trigger_Mode
REG_ADC_MINMAX	ADC_IP_BASEADDR+0x04	ADC 采集电压的最大最小值寄存器 [15:0]ADC 采样电压的最小值 [31:16]ADC 采样电压的最大值
REG_ADC_STATE	ADC_IP_BASEADDR+0x08	ADC 采集电压的中值 [15:0] ADC 采集信号中值 ADC_Mid_Val
REG_ADC_FREQ	ADC_IP_BASEADDR+0x0C	ADC 采样频率寄存器，该寄存器中存储采样信号的频率值

在 Init_Homepage 函数中，我们使用以下函数配置 ADC 进行一次数据采集：

```
Set_ADC_Mode(Channel, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
```

对上述变量说明如下所示：

表 1-12 Set_ADC_Mode 函数变量说明表

变量名称	变量含义	变量值获取
Channel	通道设置	一共有 8 个通道可供选择（0~7），默认通道 0，可以通过触摸切换

Sample_Rate	采样分频设置	采样分频值=原始采样率 200K/设定采样率，可以通过触摸设置
Trigger_Value	触发电压值	默认触发值为 0V，可以通过设置
Trigger_Mode	触发模式设置	0 表示自动触发，1 表示手动触发，2 表示单次触发，通过触摸设置

设置完 ADC 的控制器之后，我们需要读取电压、频率相关寄存器中的值，读完之后，将这些值显示在 TFT LCD 显示屏上，这里以读取电压最大值为例，通过 Get_ADC_Vmax 函数实现读取电压的功能，电压最大值对应 GET_ADC_REG 的高 16 位，读取完成后，将得到的数值转换成电压值，转换公式如下所示：

$$V = \frac{V_{Tmp}}{32768} \times 5$$

上述公式中的 V_Tmp 是从寄存器中读取出来的值，V 是实际电压值，也就是显示屏上需要显示的值，由此，我们便得到了 Get_ADC_Vmax 函数的实现方式，如下所示：

```
float Get_ADC_Vmax()
{
    uint32_t Vmax_Val;
    int16_t Vmax_Tmp;
    float Vmax;
    Vmax_Val = (GET_ADC_REG(REG_ADC_MINMAX) >> 16);
    Vmax_Tmp = Vmax_Val;    //无符号转有符号
    Vmax = Vmax_Tmp/32768.0*5.0;
    return Vmax;
}
```

获取电压的最小值、频率值、中值的方式都可以参考上述方法实现。

4. GPIO 初始化及写 SD 卡

设计中使用 PS 侧的按键控制截图功能，因此需要使用 MIO 读取按键状态，一旦按键电平发生变化，就进入到 GPIO 中断，开始截取内存中的图像数据存储到 SD 卡中。设计中我们使用的 xilffs 库，通过 FatFs 中各种 API 来操作 SD 卡，在写 SD 卡之前我们需要先对 SD 卡挂载。因此，相关代码实现如下：

```
rc = f_mount(&fatfs, "0:/", 0);      //挂载 SD 卡
if (rc != FR_OK) {
    printf("Mount Failed!\n");       //若挂载失败则发出警告
}

//GPIO 及 GPIO 中断初始化
PS_GPIO_Init();
PS_GPIO_Int_Init(PS_GPIO_IRQ_Handler);
PS_GPIO_SetInt(47,XGPIOPS_IRQ_TYPE_EDGE_FALLING);
```

在 GPIO 中断处理函数中，我们需要对按键进行消抖，以确认按键确实被按

下。随后使用 bmp_write()函数将内存中的图像数据按照 bmp 格式写入 SD 卡中。代码实现如下：

```
volatile uint8_t Cnt = 0;
char Pic_Name[30];
void PS_GPIO_IRQ_Handler(void *CallBackRef, uint32_t Bank, uint32_t
Status)
{
    uint32_t Key_State;
    XGpioPs *Gpio = (XGpioPs *)CallBackRef;

    //添加判断来消除其它 Bank 的干扰
    if(Bank == 1) {
        usleep(10000);  //按键消抖 10ms
        Key_State = XGpioPs_ReadPin(Gpio, PS_KEY);
        if(Key_State == 0) {
            //截图，存入 SD 卡
            sprintf(Pic_Name,"Screenshot(%02d).bmp",Cnt);
            bmp_write(Pic_Name, (char *)&BMODE_800x480, (char
*)frame);
            Cnt++;
        }
    }
}
```

设计中 TFT 显示使用的是 RGB565 格式数据，bmp_write()函数在从传入的 BmpMode 中获取图像尺寸后，便会基于该尺寸读取内存中图像数据。同时，函数会对数据各分量低位补零，从而扩充为 RGB888 格式后，将数据以 bmp 格式写入 SD 卡中。代码实现如下：

```
void bmp_write(char * name, char *head_buf, char *data_buf)
{
    short y,x;
    short Ximage;
    short Yimage;
    uint32_t iPixelAddr = 0;
    FRESULT res;
    unsigned int br;        // File R/W count


    memset(&Write_line_buf, 0, 1920*3) ;


    res = f_open(&fil, name, FA_CREATE_ALWAYS | FA_WRITE);
    if(res != FR_OK)
    {
        return ;
    }
```

```
    res = f_write(&fil, head_buf, 54, &br) ;
    if(res != FR_OK)
    {
        return ;
    }
    Ximage=(unsigned short)((head_buf[19] << 8) | head_buf[18]);
    Yimage=(unsigned short)((head_buf[23] << 8) | head_buf[22]);
    iPixelAddr = (Yimage-1)*Ximage*2 ;
    for(y = 0; y < Yimage ; y++)
    {
        for(x = 0; x < Ximage; x++)
        {
            Write_line_buf[x*3 + 0] = (data_buf[x*2 + iPixelAddr + 0] & 0x1F) << 3;
            Write_line_buf[x*3 + 1] = ((data_buf[x*2 + iPixelAddr + 1] & 0x07)<<5)
                                      |((data_buf[x*2 + iPixelAddr + 0]& 0xE0)>> 3);
            Write_line_buf[x*3 + 2] = data_buf[x*2 + iPixelAddr + 1] & 0xF8;
        }
        res = f_write(&fil, Write_line_buf, Ximage*3, &br) ;
        if(res != FR_OK)
        {
            printf("Write BMP Failed!\n");
            return;
        }
        iPixelAddr -= Ximage*2;
    }

    f_close(&fil);
    printf("Write BMP Successfully!\n");
}
```

1.3.2.2单次触发函数

Handle_Single_Trigger 主要用来处理单次触发事件，在该函数中，首先通过 ADC_DataTransfer 函数读取 RAM 中存储的数据，当处于单次触发时，恢复按钮的颜色，绘制按钮，触发取消则不再 STOP，否则执行 STOP 操作，其实其中涉及的都是按钮颜色背景的切换，比较容易理解，这里就不再进行详细说明，函数的实现代码如下所示：

```
void Handle_Single_Trigger()
{
    ADC_DataTransfer(P_ADC_Data,ADC_DATA_LENGTH);
    //按钮恢复原色
    if(Trigger_Mode == 0x03) {
        Button_TriggerVal.TextColor = LCD_BLACK;
```

```
        Button_TriggerVal.BackColor = LCD_GREEN;
    } else {
        Button_TriggerVal.TextColor = LCD_WHITE;
        Button_TriggerVal.BackColor = LCD_GRAY;
    }
    Draw_Normal_Button(Button_TriggerVal);

    Trigger_Enable_Press();

    if(!Cancel_Trigger)//触发取消则不再 STOP
        Perform_STOP();//执行 STOP 操作
    Cancel_Trigger = 0;     //清标志
}
```

1.3.2.3启动下一次传输

启动下一次传输函数（Handle_Round_Done）首先通过 Set_ADC_Mode 函数设置 ADC 的控制寄存器，然后通过 ADC_DataTransfer 函数读取 RAM 中存储的数据，代码如下所示：

```
void Handle_Round_Done()
{
    //开启下一次传输
    Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
    ADC_DataTransfer(P_ADC_Data,ADC_DATA_LENGTH);
}
```

1.3.2.4刷新波形窗口

刷新波形窗口（Refresh_WaveWindow）函数首先读波形数据、然后执行刷新电压值、刷新时间显示等操作，函数的实现代码如下所示：

```
void Refresh_WaveWindow()
{
    Read_Wave_Data();//读波形数据
    Draw_Waveform(ADC_Wave,Pre_Wave_Data,Wave_Data);    //画波形
    if(Wave_Run) {
        if(Trigger_Mode)
            POINT_COLOR=LCD_ORANGE;     //画笔颜色
        else
            POINT_COLOR=LCD_BRRED;     //画笔颜色
            //画横标线
            Draw_Mark_Line(0,Trigger_Point, ADC_Wave.Wave_Area.X1,
                    ADC_Wave.Wave_Area.X1 + ADC_Wave.Wave_Area.Width);
    } else if(!Wave_Run) {
```

```
        POINT_COLOR=LCD_CYAN;      //画笔颜色
        //画竖标线
        Draw_Mark_Line(1,Measure_Point, ADC_Wave.Wave_Area.Y1,
                ADC_Wave.Wave_Area.Y1 + ADC_Wave.Wave_Area.Height);

        //刷新电压显示
        Point_Voltage=Wave_Data[Measure_Point - ADC_Wave.Wave_Area.X1]*5.0/32768;
        if(Point_Voltage >= 0)
            sprintf(str,"Voltage = +%.3fV",Point_Voltage);
        else
            sprintf(str,"Voltage = %.3fV",Point_Voltage);
        Draw_Normal_Text(Text_Point_V, str);

        //刷新时间显示
                        Point_Time=(float)((Measure_Point-ADC_Wave.Wave_Area.X1)-
250)/Sample_Rate[Sample_Set];
        if(Point_Time >= 0)
            sprintf(str,"Time = +%.3fms",Point_Time);
        else
            sprintf(str,"Time = %.3fms",Point_Time);
        Draw_Normal_Text(Text_Point_T, str);
    }
    POINT_COLOR=LCD_BLACK;      //画笔颜色
}
```

上述函数中主要都是关于绘图的说明，这里将不做介绍，其中比较重要的就是读取波形数据 Read_Wave_Data 函数的实现，函数波形的显示一共分为两页显示，一页最多显示 500 个数据，首先保存上一次的波形数据，然后根据界面上的滑动按钮，切换 Wave_Data 保存的数据的起始地址，也就是设置的 P_ADC_Data 中的起始第一个需要显示的数据，代码如下所示：

```
void Read_Wave_Data()
{
    uint16_t i ;
    memcpy(Pre_Wave_Data,Wave_Data,ADC_DATA_LENGTH*2);
    for(i = 0 ;i<ADC_DATA_LENGTH;i++)
    {
        Wave_Data[i] = P_ADC_Data[i+ADC_Wave_Offset];
    }
}
```

至此，对于函数库中比较重要的部分，也就是数据处理部分我们已经有了较为深入的了解。至于余下的触摸以及画图部分，请用户自行查看源代码进行理解。

接下来保存设计，软件自动编译无误后便可以进行板级验证了。

1.4 板级验证

1.4.1 系统所需硬件

1. ACZ702 开发板一个
2. 电源线一根（可选）
3. ADC7606 模块一个
4. 5 寸 800*480 分辨率 LCD 屏
5. 下载线一根
6. 信号发生器一个
7. 软排线一根
8. SD 一个
9. 读卡器一个

1.4.2 硬件连接

本次系统设计硬件连接如图 1-34、图 1-35 和图 1-36 所示，连接步骤如下：

1. 用软排线连接好 LCD 屏，连接方法如图 1-34 所示
2. 连接好 AD7606 模块（右侧对齐），如图 1-36 所示
3. 将信号发生器的输出信号连接到 AD7606 上
4. 将 SD 卡插入 SD 卡卡槽
5. 连接下载线

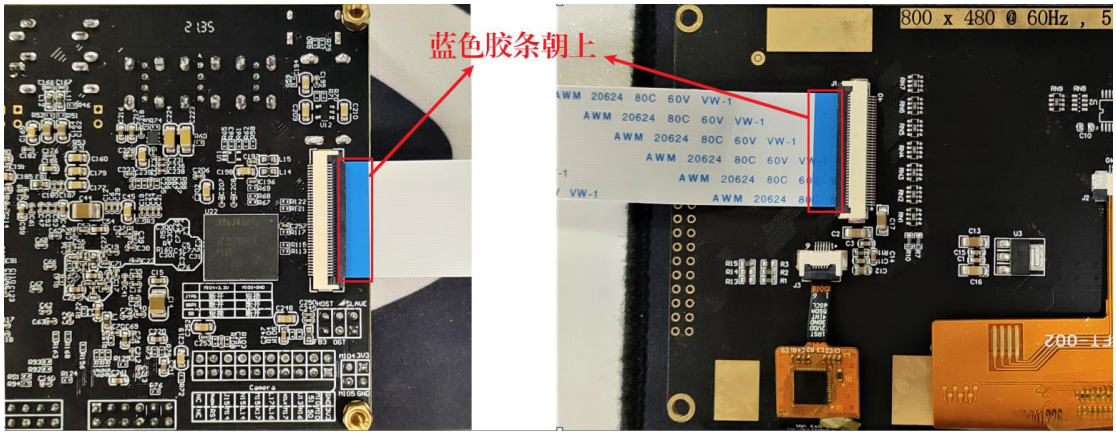

图 1-34　LCD 屏连接图

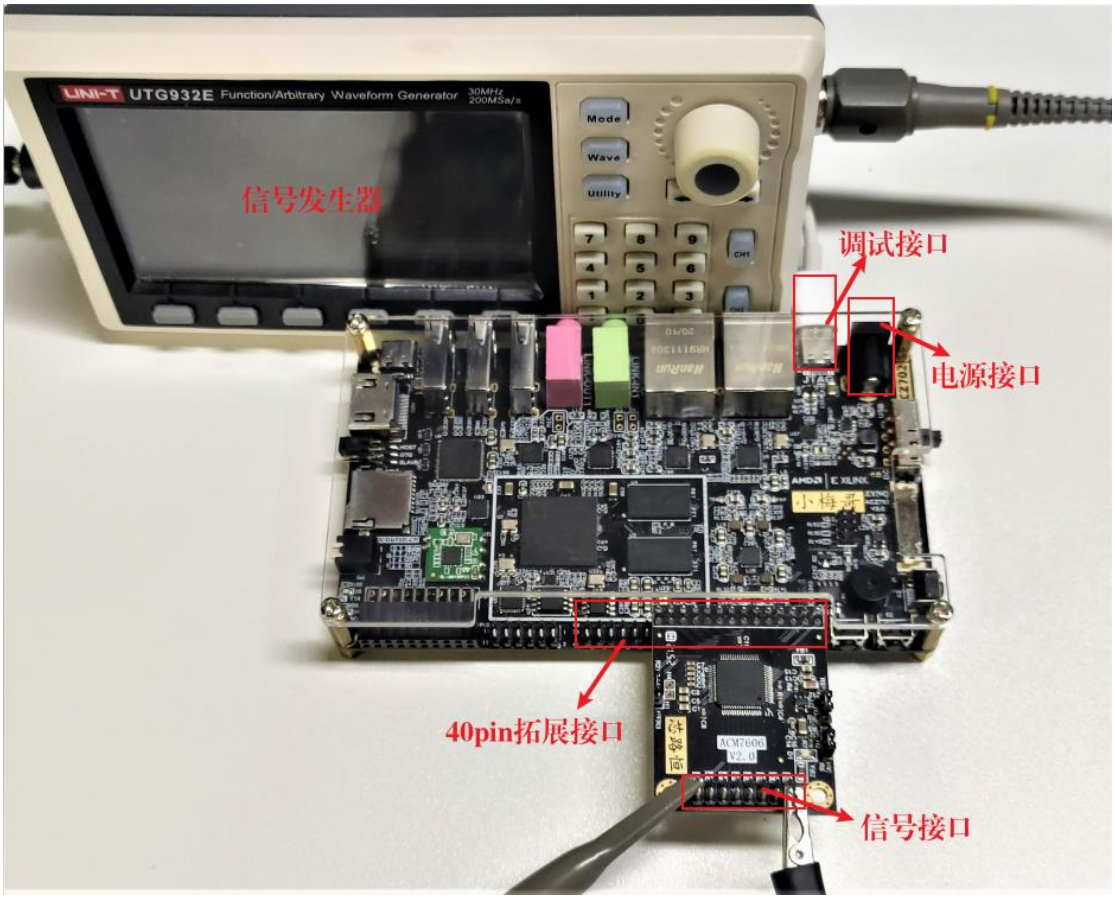

图 1-35　开发板正面连接图

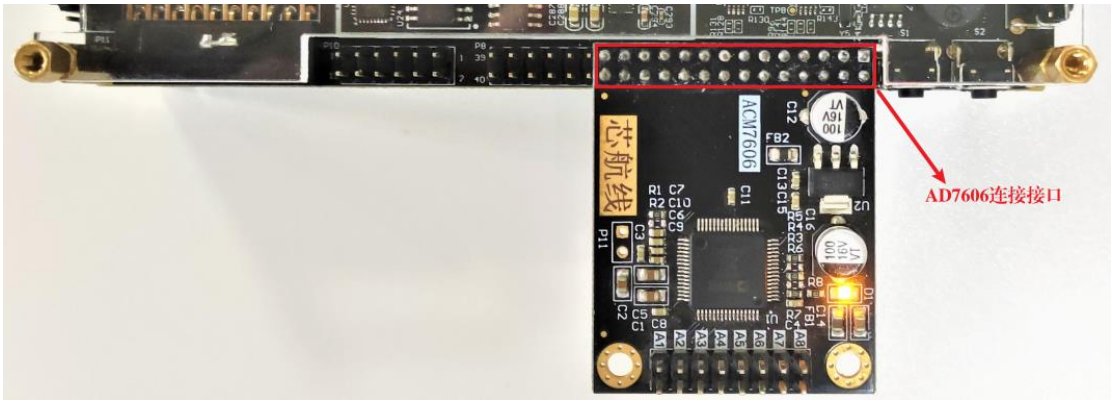

图 1-36　AD7606 连接图

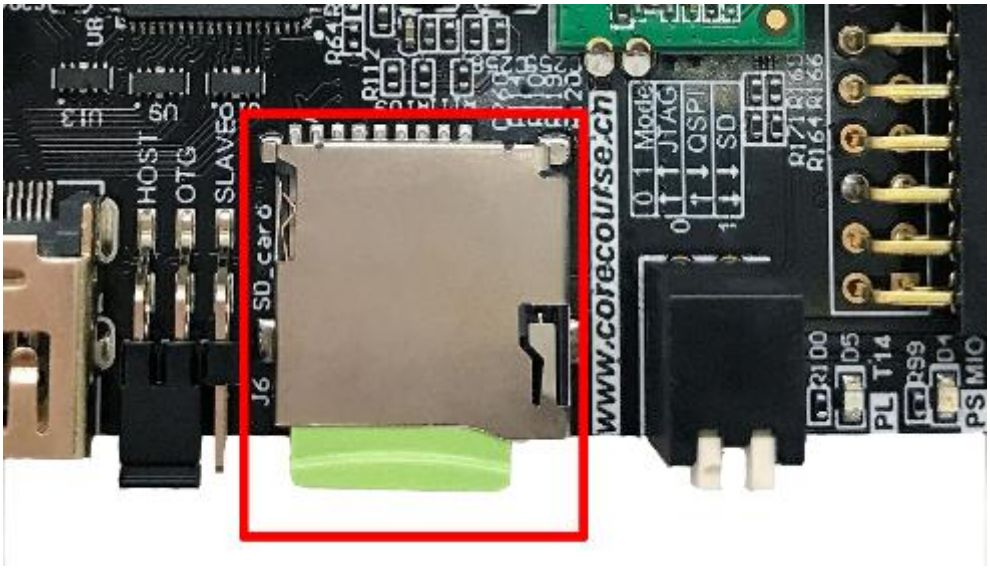

图 1-37　SD 卡插入

本次板级验证中，我们使用信号发生器作为信号源。有条件的用户也可以使用另一块开发板自制一个 DDS 信号发生器作为信号源。在连接硬件时，需要注意 AD7606 模块的 1 号引脚与开发板上 40pin 拓展口的 1 号引脚相连。连接完成后拨通开发板电源即可开始进行下载验证了。

1.4.3 串口终端连接

打开设备管理器，查看 PS 侧串口端口号，笔者电脑上 ACZ702V2.0 与 V1.0 的串口标识如图 1-38 所示：

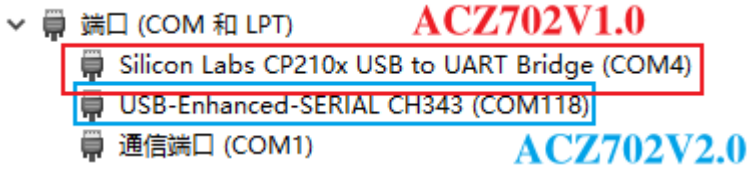

图 1-38　端口列表

以 V2.0 开发板为例，笔者电脑上 PS 端串口端口号为 COM118，打开 SDK 终端，连接 COM118。连接时将波特率设置为 115200，其余项保持默认即可，如图 1-39 所示：

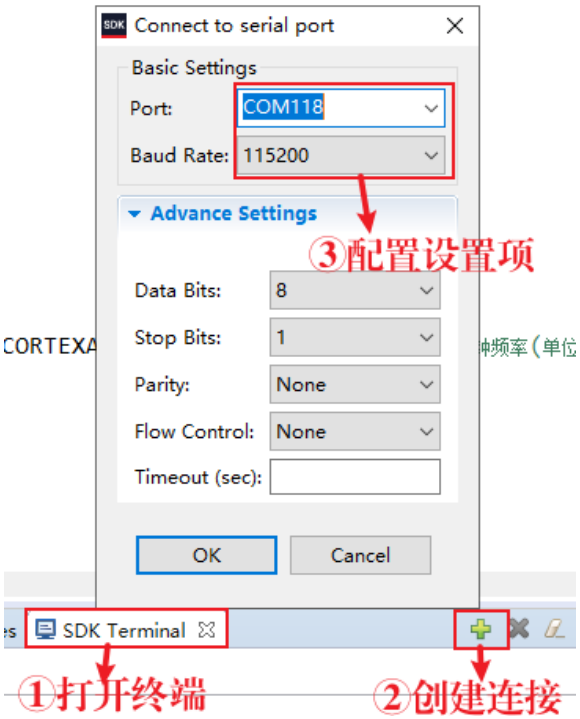

图 1-39　串口终端连接

连接完成后终端会打印连接成功提醒，如图 1-40 所示：

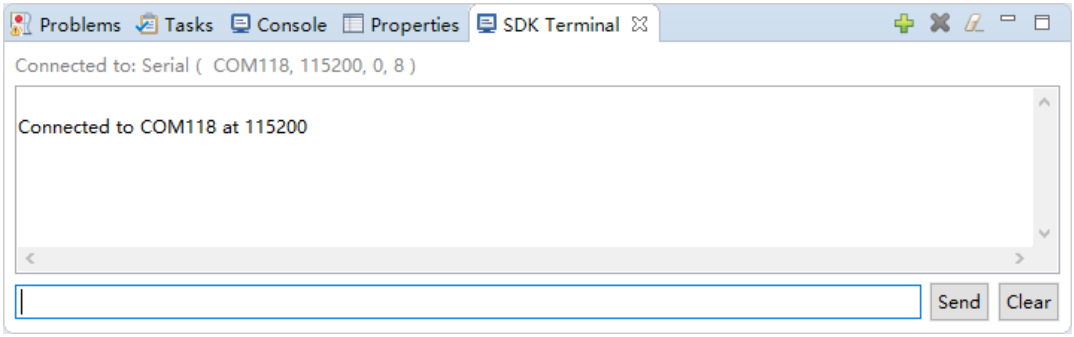

图 1-40　串口终端连接成功

连接完串口终端，接下来便可以将烧录文件下载到开发板验证设计了。

1.4.4 下载与验证

点击本次设计的工程资源文件，创建烧录任务，将生成的烧录文件下载至开发板中，如图 1-41 所示，具体流程如下：

1. 创建烧录任务
2. 确认添加比特流和 PS 初始化脚本
3. 勾选“Reset entire system”和“Program FPGA”，确认勾选了“Run ps7_init”和“Run ps7_post_config”
4. 在 Application 界面中检查是否添加.elf 文件并勾选下载任务。

5. 点击“Run”开始烧录

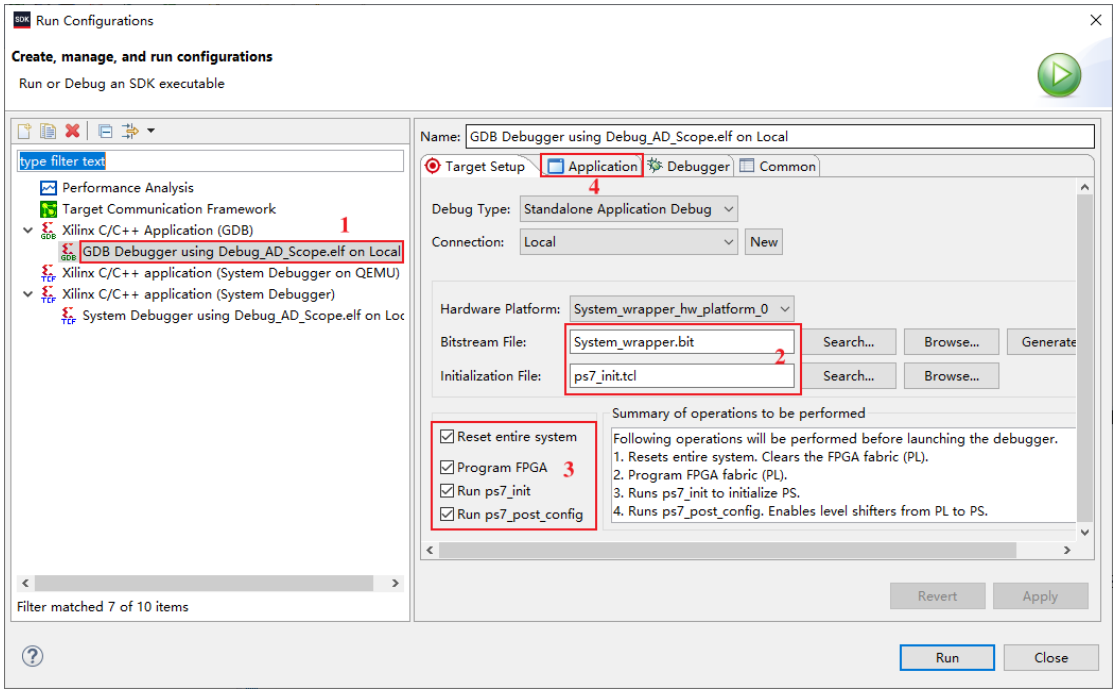

图 1-41 创建烧录任务

将本次烧录文件下载至开发板后，打开信号发生器，当软件右下角的下载进度条消失且无报错时代表下载成功，稍等片刻即可看到 LCD 屏上显示出了波形界面，如图 1-42 所示，且 SDK 终端中也会打印相关版本信息，如图 1-43。

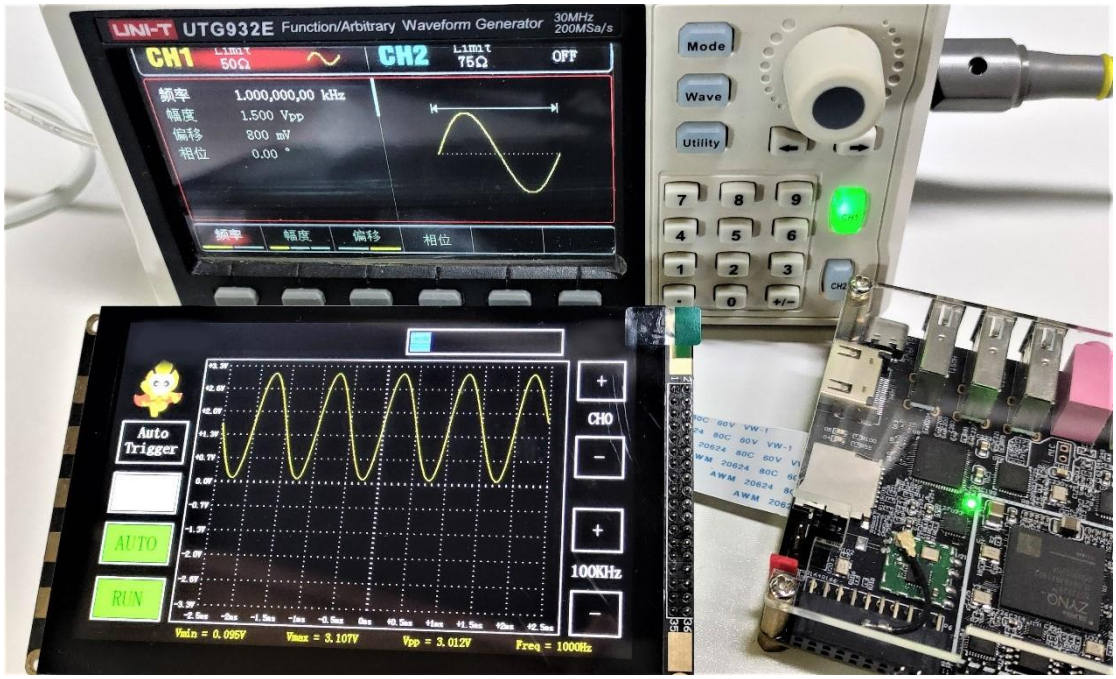

图 1-42 波形界面

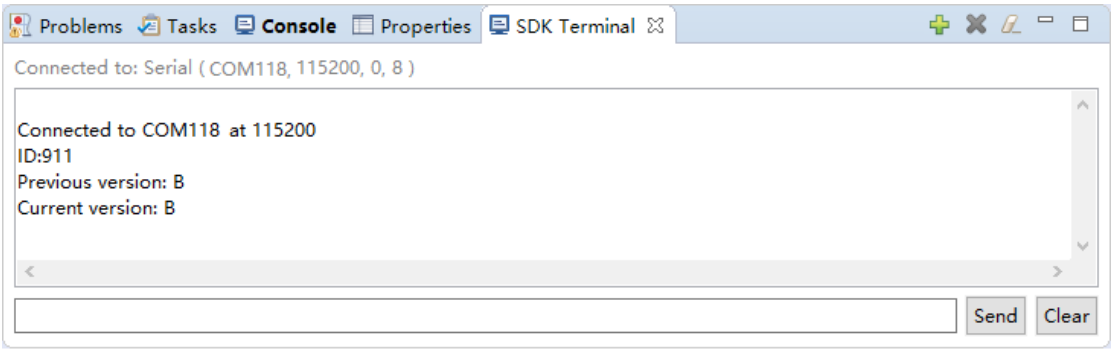

图 1-43　版本信息

1.4.5 功能介绍

程序下载成功之后，初始界面如下图 1-44 所示（未连接信号源）。

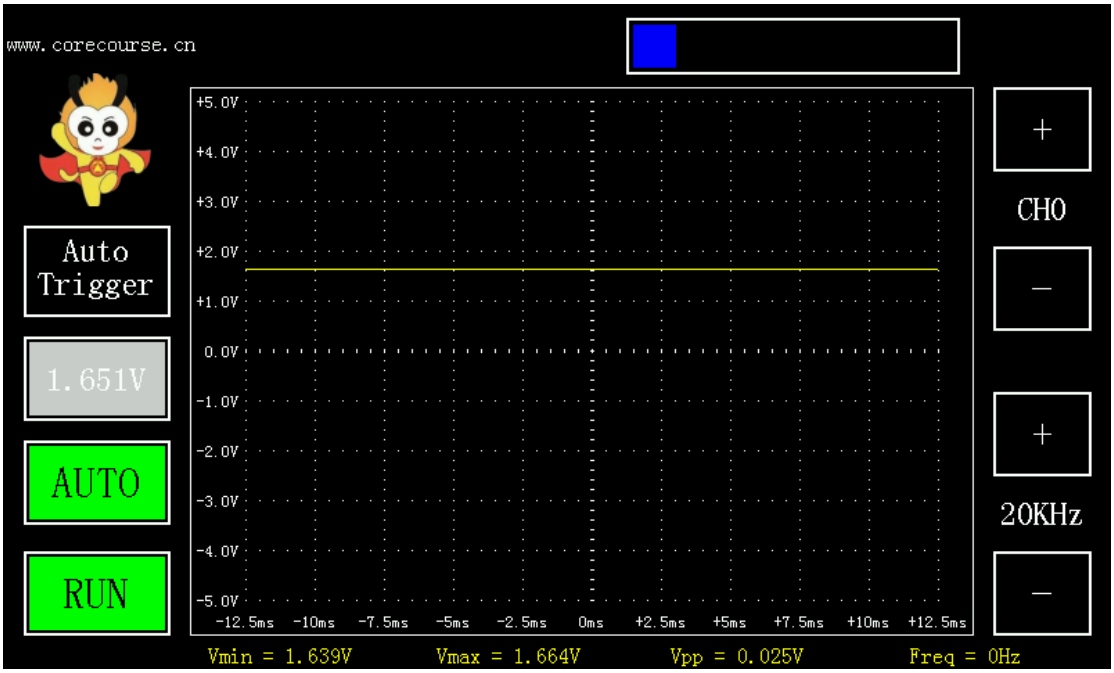

图 1-44 初始界面图

然后给 AD7606 的通道 1 输入一个 200hz，Vpp=5V 的正弦波，显示如下图 1-45，从图中最下一行的显示中，可以看到，其采集电压最大、最小值都为 2.5V 左右，频率为 200hz，与我们输入的信号一致。

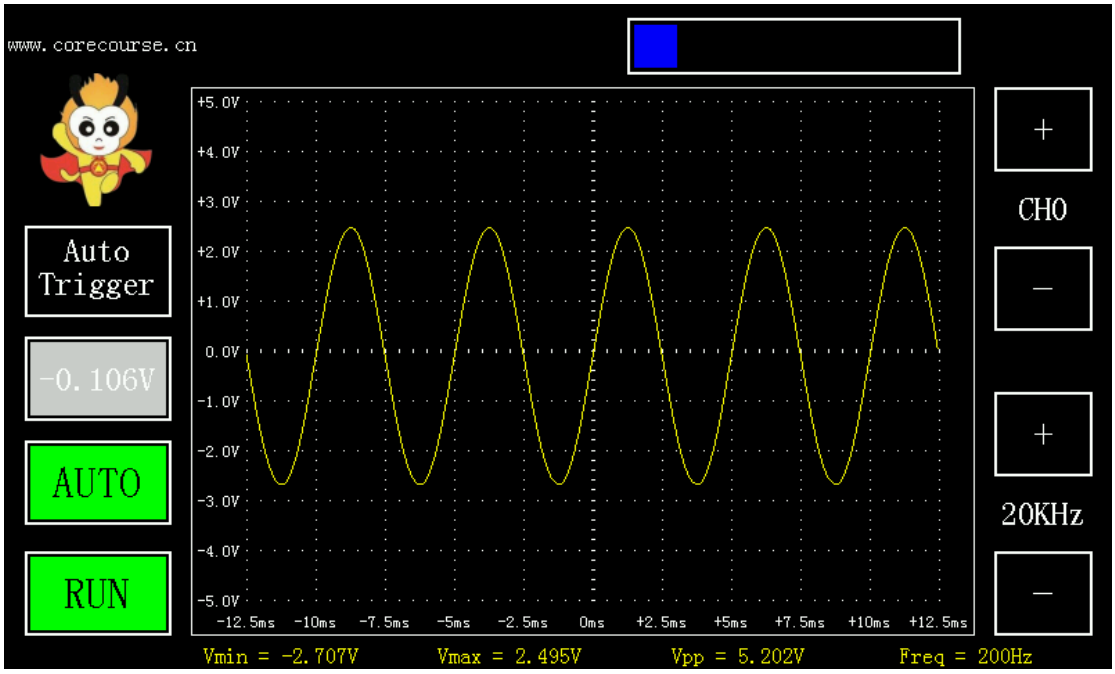

图 1-45 输入信号源之后，显示波形图

下面将对主界面中设计的一些功能进行说明。

1.4.5.1主界面说明

主界面如下图 1-46 所示。

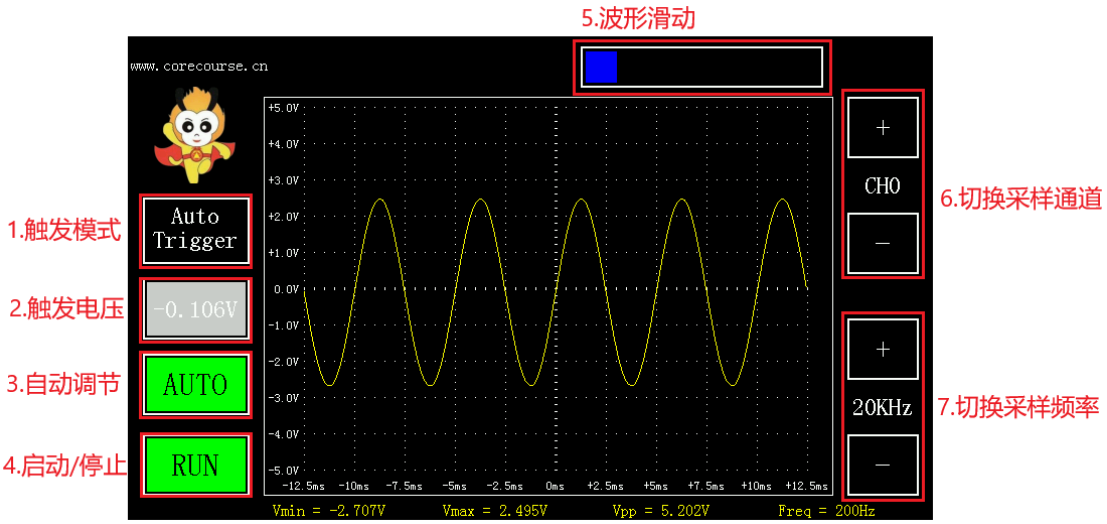

图 1-46 主界面示意图

1. 触发模式：可选自动触发、普通触发、单次触发三种模式；
2. 触发电压：手动模式下不可更改，在其他模式下，可通过滑动波形窗口的橙色标线来选择当前的触发电压；
3. 自动调整：会根据当前输入的波形频率，自动调整到最佳的采样率，并

将触发模式切换为自动模式；

4. 启动/停止：启动/停止波形采集；
5. 波形移动：本工程存储了长度为 1024 的波形，但窗口只显示 500，拖动滑块即可移动波形；
6. 切换采样通道：点击“+”、“-”可切换 CH0~CH7 共 8 个采样通道；
7. 选择采样率：点击“+”、“-”可切换 1KHz~1000KHz 共 10 种采样率；
8. 显示输入电压的最值、峰峰值、频率值。

1.4.5.2 测量界面介绍

测量界面如下图 1-47 所示，在点击“RUN”按钮后，“RUN”会变为“STOP”表示停止波形采集，并进入测量界面在测量界面会有一条青色竖标线，触摸可以移动标线，在上方会显示标线与波形交点处的电压值、相对时间值

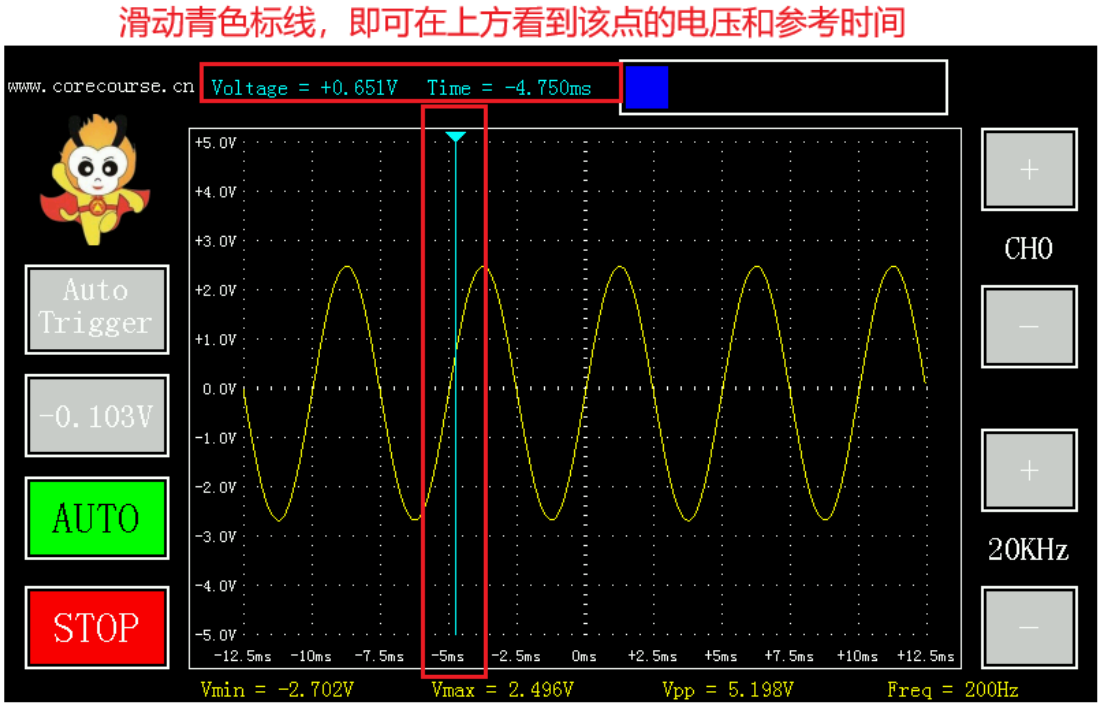

图 1-47 测量界面示意图

1.4.5.3 触发模式介绍

本工程支持三种触发方式：自动触发、普通触发、单次触发

1. 自动触发：触发值自动选择为输入电压的中值，如下图 1-48 所示。

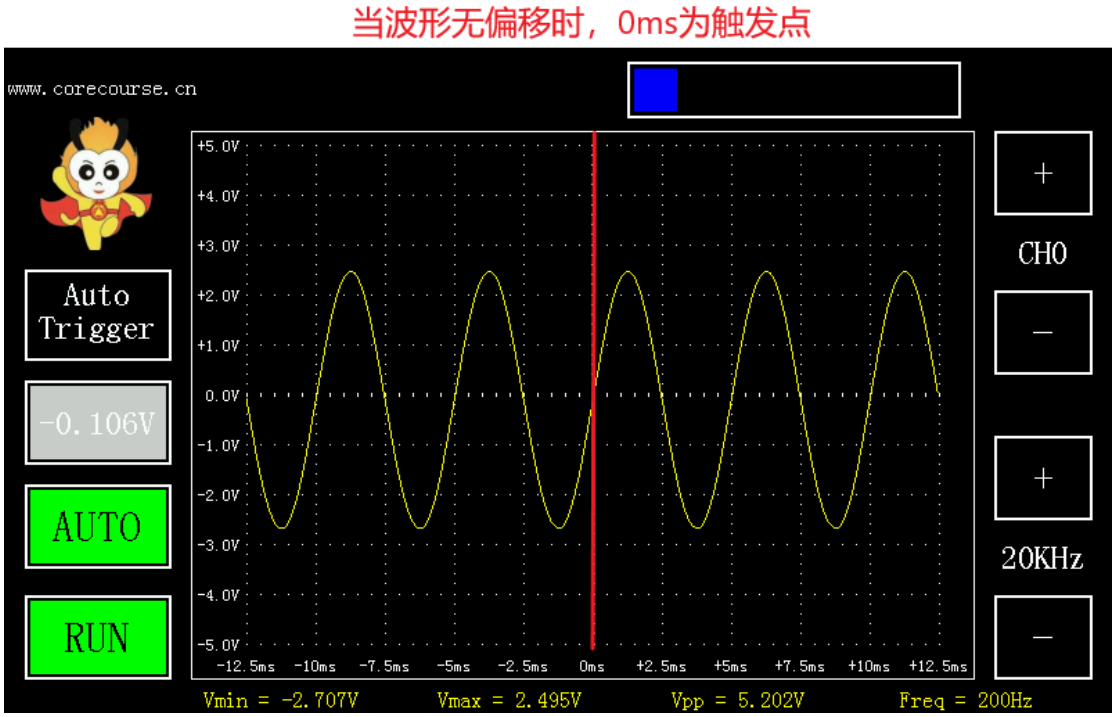

图 1-48 自动触发显示示意图

2. 普通触发：触发值由用户自己选择。如下图 1-49，点击触发模式切换按键，切换到普通触发模式，然后设置触发电压为 0.917V，可以看到波形触发点变为 0.917V。

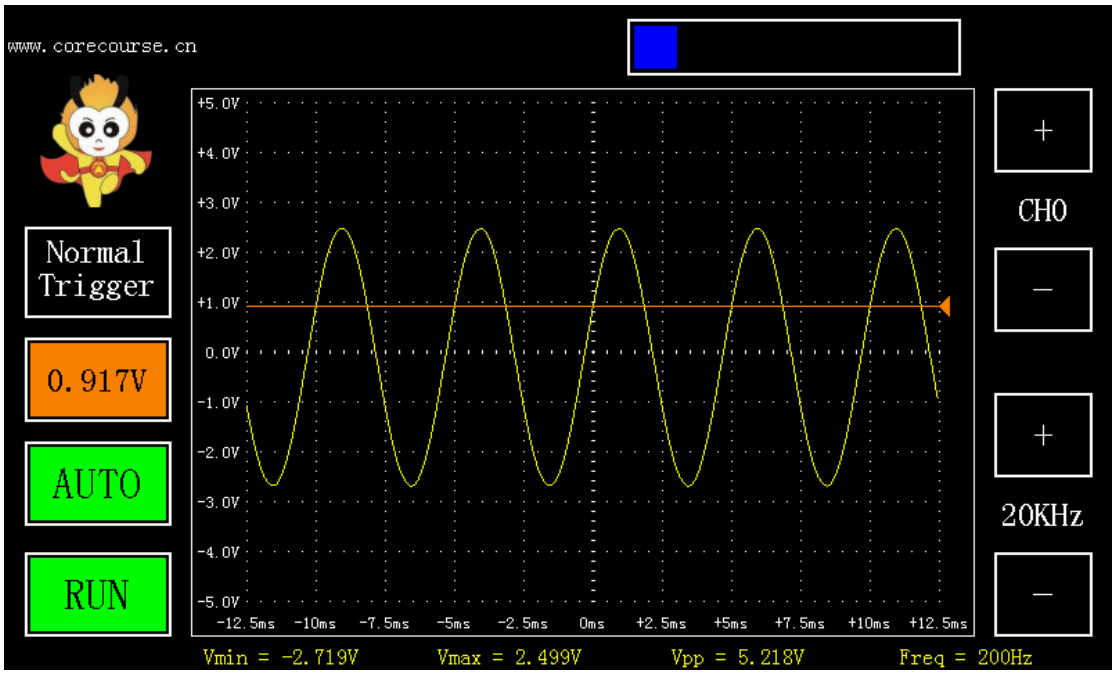

图 1-49 普通触发示意图

3. 单次触发：点击触发模式切换按键，切换到单次触发模式，“触发电压”按钮会变为绿色可触摸，然后设置触发电压为 0.750V，点击“0.750V”

启动单次触发，在等待触发时，“触发电压”按钮为红色，再次点击可退出等待，如下图 1-50 所示。

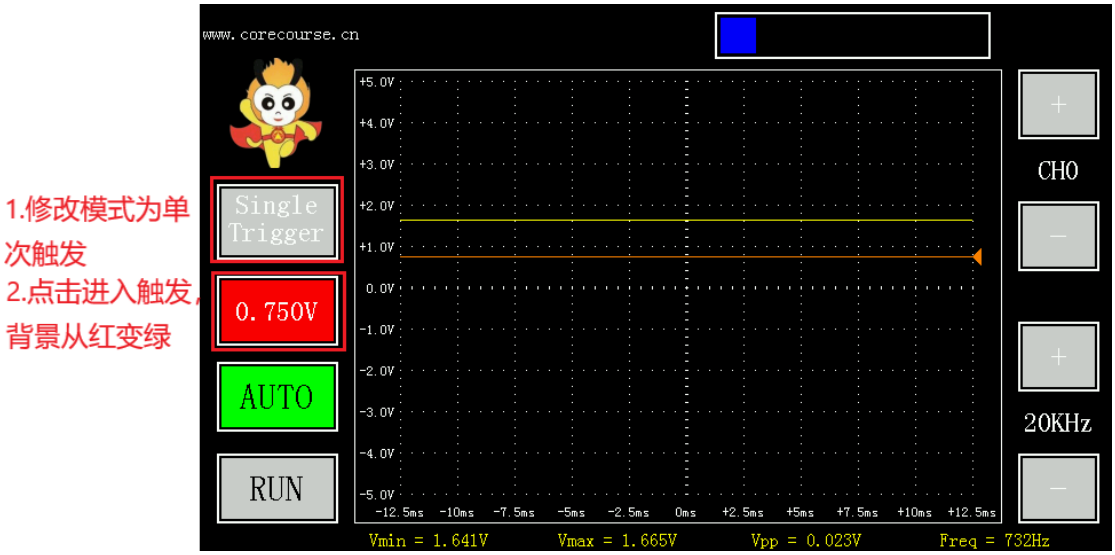

图 1-50 单次触发波形图

触发完成之后，会进入停止状态，方便观测波形，如下图 1-51 所示。

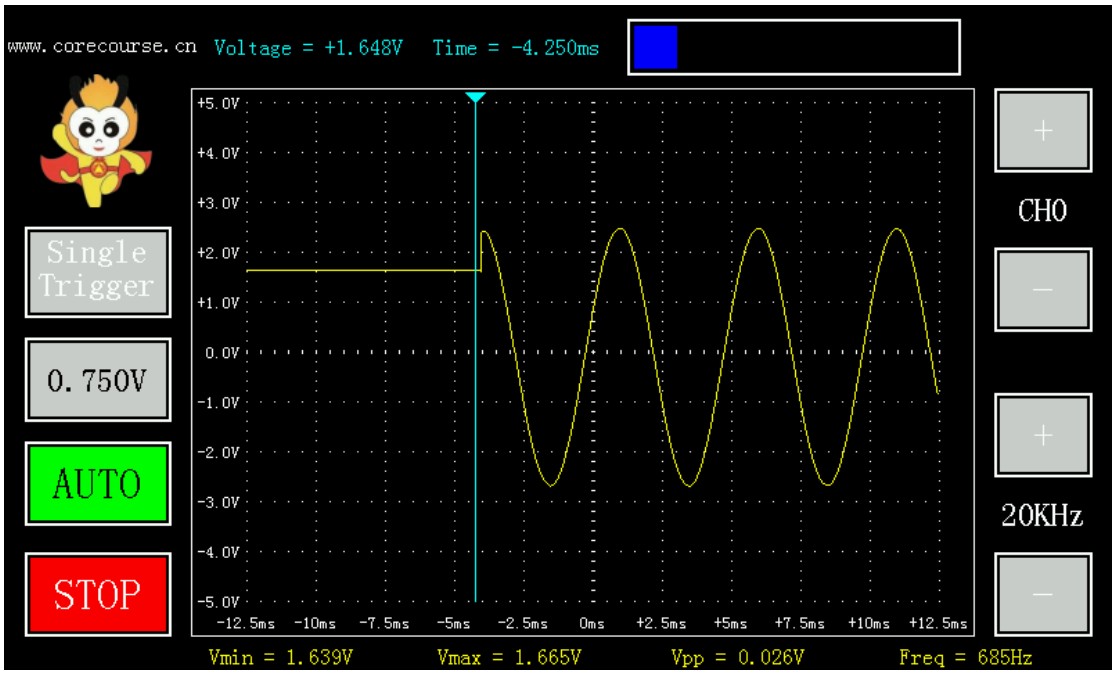

图 1-51 触发完成波形图

1.4.5.4截图功能介绍

设计使用 ACZ702 开发板 PL 侧按键实现截图功能，用户按下按键后，程序会拷贝内存数据，在处理后以.bmp 的命名方式写入 SD 卡中，并通过串口打印出写入结果。这里笔者截图了四次，串口打印情况如图 1-52 所示：

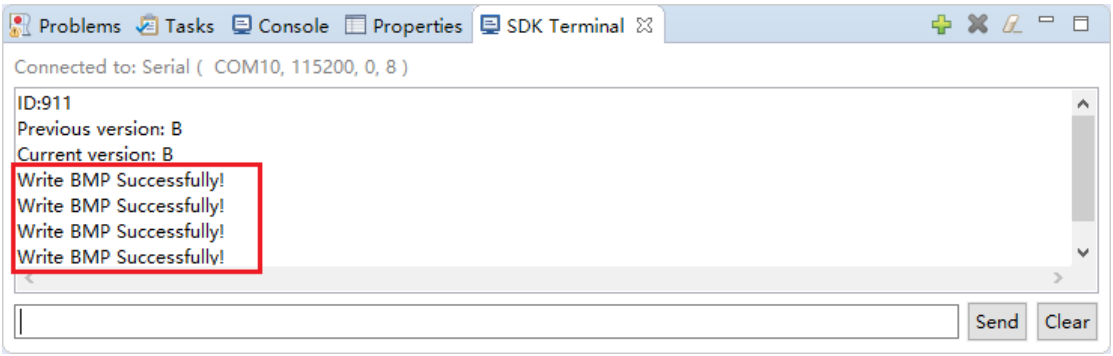

图 1-52　串口打印

接下来将 SD 卡从开发板中弹出，插入到读卡器内，再将读卡器插到电脑主机的 USB 接口上。接下来我们打开 windows 资源管理器，查看 SD 卡中的内容，如图 1-53 所示：

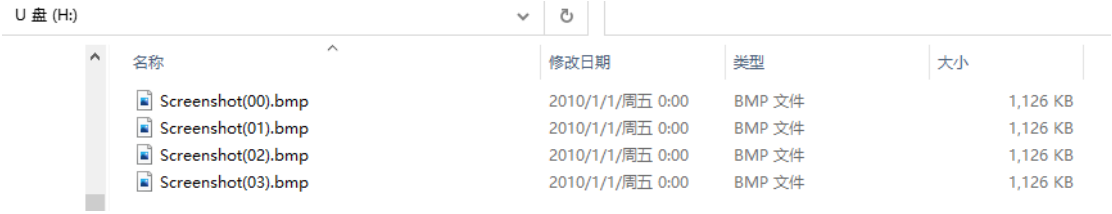

图 1-53　使用资源管理器查看截图保存文件

双击截图文件便能直接查看截图内容，如图 1-54 所示：

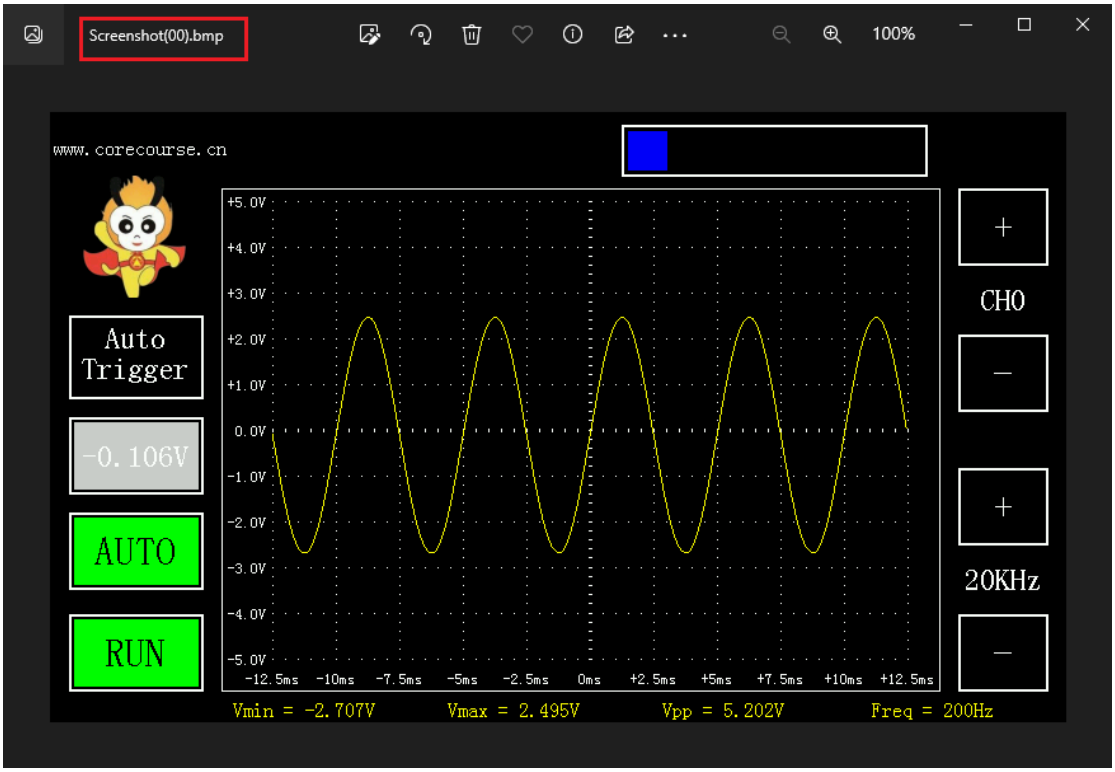

图 1-54　查看截图内容

至此，设计所有功能介绍完成，本次设计结束。

1.5 章节总结

1. 本章设计以 RGB_LCD 实验为基础，结合 AD7606 模块完成了多通道数据采集系统的搭建，用户在使用时可以直接通过触摸 LCD 屏上对应按钮，实现不同功能的切换/控制以及采集通道的选择。
2. 在进入单次触发之后，我们可以看到波形并不是从设定的触发电压之后开始显示波形，而是在设定的触发电压之前就已经有波形数据了，这是因为我们在程序中添加了一个移位寄存器，将 AD 采集的数据进行移位之后，才送到 RAM 中进行显示，所以这里才能看到在触发之后，还会显示触发之前的数据。