说　明

1、本内容节选自《FPGA 自学笔记——设计与验证》一书 6.4 节，内容仅供学习使用，不得用于任何其他商业目的。如果您喜欢，建议购买正版纸质书。

2、本节内容由小梅哥于 2019 年 4 月 30 日修订，增加了系统测试部分的细节内容。

3、如果有更新，我们将会发布在我们的官方论坛 www.corecourse.cn，如需关注本节内容的更新，可至我们的官方网站以本文档标题为关键词进行检索，以查找相关内容。

4、本教程对应的 FPGA 工程源码名为“29_DDS2”，位于“【书本源码】FPGA 自学笔记——设计与验证.rar”压缩包的“chapter6”中。

6.4 双通道幅频相可调 DDS 信号发生器

本节导读

本节将结合第三章中串口收发小节实现的串口收发模块、第四章中 ROM 使用小节讲解的 ROM 使用知识以及第 5 章中的 DAC 驱动小节实现的 DAC 驱动模块实现一个双通道幅度、频率、相位均可调的 DDS 信号发生器。

其中，输出信号的频率、相位、幅度的调节通过 PC 端串口发送命令实现，最终由 DAC 芯片 TLV5618 输出对应的模拟信号。PC 端通过串口发出控制命令，由串口接受命令并解析转换得到相应的幅度、频率、相位控制字，这些控制字直接作用在两个 DDS 信号发生器电路上，从而设定信号发生器输出信号的幅度、频率、相位，然后经过 DAC 控制模块将两个通道的数据分别经由 DAC 驱动模块写入 TLV5618 型 DAC 中， DAC 即可输出对应的模拟电压信号。其系统工作框图如图 6.4-1 所示。通过本节，我们将实现一个通过上位机可控的双通道幅频相信号发生器，具体信号发生器输出信号的最高频率与 DAC 的转换速率相关。

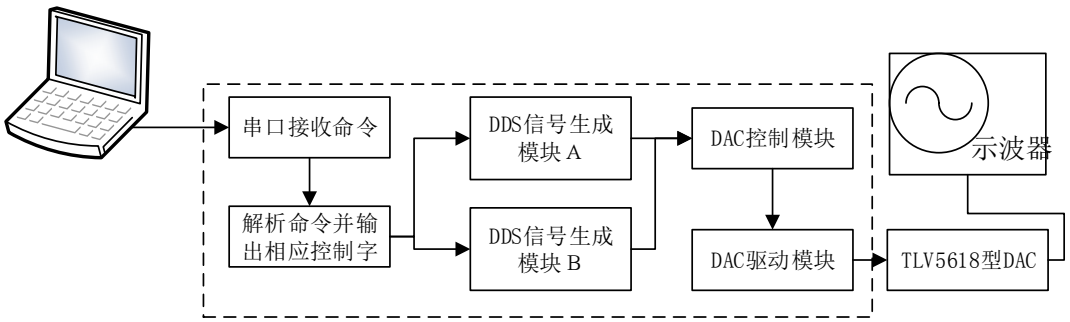

图 6.4 -1　基于 AC620 开发板的双通道 DDS 系统架构图

由工作原理图可以暂时将本系统划分为串口接收模块、命令解析模块、DDS 信号生成模块、DAC 控制模块和 DAC 驱动模块。

6.4.1 DDS 原理与实现

6.4.1.1 DDS 基本原理

DDS(Direct Digital Synthesizer)即数字合成器，是一种新型的频率合成技术，具有相对带宽大，频率转换时间短、分辨率高和相位连续性好等优点。较容易实现频率、相位以及幅度的数控调制，广泛应用于通信领域。

DDS 的基本结构图如图 6.4 -2 所示。

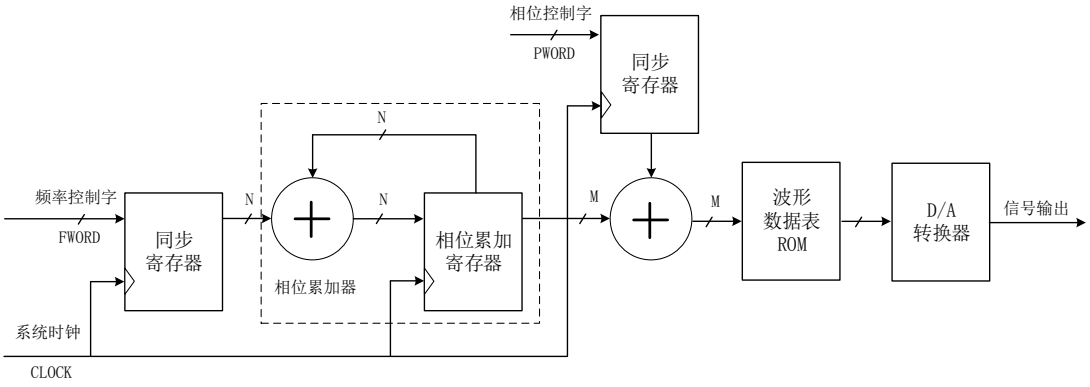

图 6.4 -2 DDS 基本结构图

由图 8.6.2 可以看出，DDS 主要由相位累加器、相位调制器、波形数据表以及 D/A 转换器构成。

其中相位累加器由 N 位加法器与 N 位寄存器构成。每个时钟周期的时钟上升沿，加法器就将频率控制字与累加寄存器输出的相位数据相加，相加的结果又反馈至累加寄存器的数据输入端，以使加法器在下一个时钟脉冲的作用下继续与频率控制字相加。这样，相位累加器在时钟作用下，不断对频率控制字进行线性相位累加。即在每一个时钟脉冲输入时，相位累加器便把频率控制字累加一次。

相位累加器输出的数据就是合成信号的相位。相位累加器的溢出频率，就是 DDS 输出的信号频率图 2.30 相位累加器输出的数据，作为波形存储器的相位采样地址，这样就可以把存储在波形存储器里的波形采样值经查表找出，完成相位到幅度的转换。波形存储器的输出数据送到 D/A 转换器，由 D/A 转换器将数字信号转换成模拟信号输出。

DDS 信号流程示意图如图 6.4 -3 所示。

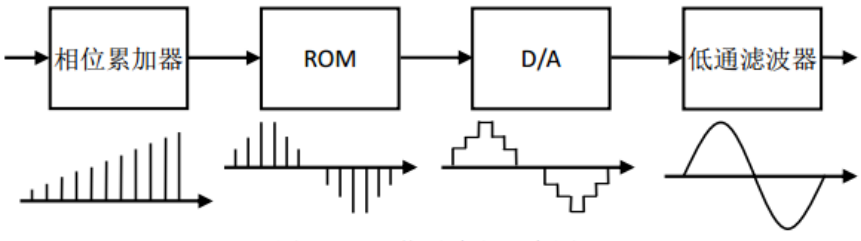

图 6.4 -3 DDS 原理流程图

这里相位累加器位数为 N 位（N 的取值范围实际应用中一般为 24~32），相当于把正弦信号在相位上的精度定义为 N 位，所以其分辨率为$1/2^N$。

若 DDS 的时钟频率为F_{clk}，频率控制字 fword 为 1，则输出频率为$F_{out}=\frac{F_{clk}}{2^N}$，这个频率相当于“基频”。若 fword 为 B，则输出频率为$F_{out}=B\times\frac{F_{clk}}{2^N}$。

因此理论上由以上三个参数就可以得出任意的f_o输出频率。且可得出频率分辨率由时钟频率和累加器的位数决定的结论。当参考时钟频率越高，累加器位数越高，输出频率分辨率就越高。

从上式分析可得，当系统输入时钟频率F_{clk}不变时，输出信号频率由频率控制字 B 所决定，由上式可得：$B=2^N\times\frac{F_{out}}{F_{clk}}$。其中 B 为频率字且只能取整数。为了合理控制 ROM 的容量，此处选取 ROM 查询的地址时，可以采用截断式，即只取 32 位累加器的高 M 位。这里相位寄存器输出的位数一般取 10~16 位。

以上通过理论计算加数据变换的形式对 DDS 原理进行了较为严谨的解释，但是 DDS 究竟是怎么实现频率和相位的控制的呢，以下通过一个简化的实例来描述 DDS 实现频率和相位控制的过程。

图 6.4 -4 为一个完整周期的正弦信号的波形，总共有 33 个采样点，其中第 1 点和第 33 点的值相同，第 33 点为下一个周期的起始点，因此，实际一个周期为 32 个采样点（1~32）。

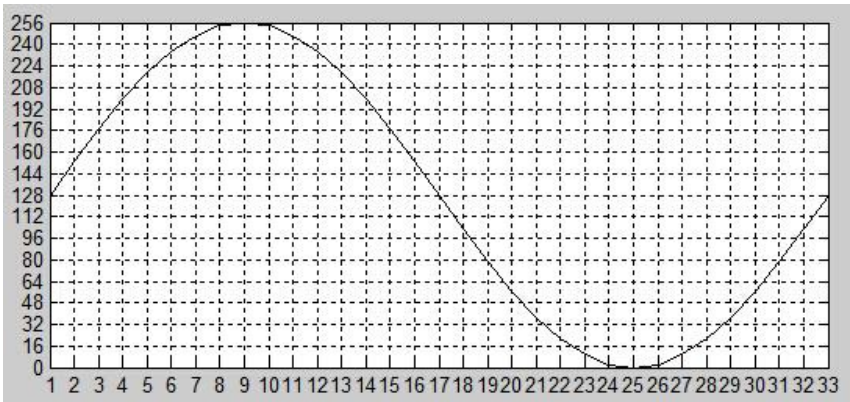

图 6.4 -4　32 个采样点的正弦信号波形

当使用 FPGA 控制 DAC 输出一个周期的正弦信号时，每 1ms 输出一个数值。如果每个点都输出，则总共输出这一个完整的周期信号需要输出 32 个点，因此输出一个完整的信号需要 32ms，可知输出信号的频率为 1000/32Hz。

如果需要用这一组数据来输出一个 2*（1000/32）Hz 的正弦信号，因为输出信号频率为 2*（1000/32）Hz，那么输出一个完整的周期的正弦波所需要的时间为 32/2，即 16ms。为了保证输出信号的周期为 16ms，我们需要对我们的输出策略进行更改，上面输出周期为 32ms 的信号时，我们采用的为逐点输出的方式，以 32 个点来输出一个完整的正弦信号，而我们 FPGA 控制 DAC 输出信号的频率固定为 1ms。因此，我们要输出周期为 16ms 的信号，只能输出 16 个点来表示一个完整的周期。我们就选择以每隔一个点输出一个数据的方式来输出即可。我们可以选择输出（1、3、5、7……29、31）这些点，因为采用这些点，我们还是能够组成一个完整的周期的正弦信号，而输出时间缩短为一半，即频率提高了一倍。最终结果如图 6.4 -5 所示。

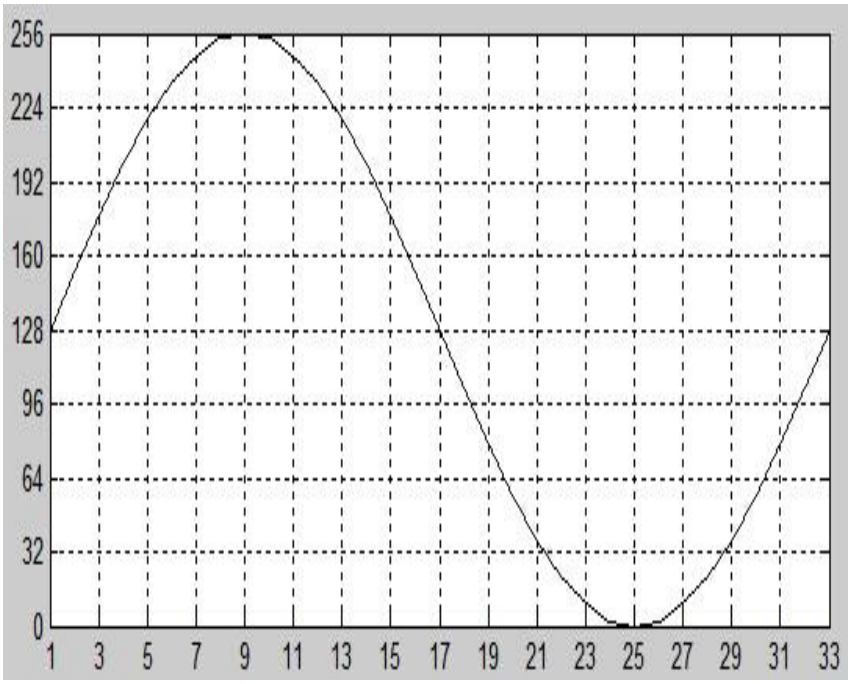

图 6.4 -5　16 个采样点的正弦信号波形

如果需要使用该组波形数据输出频率为(1/2)*（1000/32）Hz 的信号，即周期为 64ms，则只需要以此组数据为基础，每 2ms 输出一个数据即可，例如第 1ms 和第 2ms 输出第一个点，第 3ms 和第 4ms 输出第二个点，以此类推，第 63ms 和第 64ms 输出第 32 个点，即可实现周期加倍，即频率减半的效果。

对于相位的调整，则更加简单。只需要在每个取样点的序号上加上一个偏移量，便可实现相位的控制。例如，上面默认的是第 1ms 时输出第一个点的数据，假如我们现在在第 1ms 时从第 9 个点开始输出，则将相位左移了 90 度，这就是控制相位的原理。

实现 DDS 输出时，将横坐标上的数据作为 ROM 的地址，纵坐标上的数据作为 ROM 的输出，那么指定不同的地址就可实现对应值的输出。而我们 DDS 输出控制频率和相位，归结到底就是控制 ROM 的地址。

6.4.1.2 DDS 模块功能设计

在本设计中参考时钟F_{clk}频率为 50 MHz，相位累加器位数 N 取 32 位，频率控制字位数 M 取 12 位。

经过以上的分析，可以得出 DDS 模块的端口模块图如图 6.4 -6 所示。

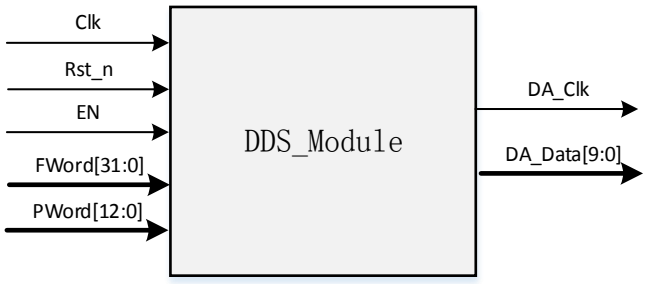

图 6.4 -6 DDS 模块接口示意图

其中，每个端口的功能描述如表 6.4-1 DDS 模块信号功能描述表 6.4-1 所示。

表 6.4-1 DDS 模块信号功能描述

端口名称	I/O	功能描述
Clk	I	为本模块的工作时钟，频率为 50MHz
Rst_n	I	控制器复位，低电平复位
EN	I	DDS 模块使能信号
FWord	I	频率控制字
PWord	I	相位控制字
DA_Clk	O	DA 数据输出时钟
DA_Data	O	DA 数据输出

新建 DDS_Module.v 保存至 rtl 文件夹下。从图 6.4 -6 以及表 6.4-1 就得到了端口列表：

```
input Clk;
input Rst_n;
input EN;
input [31:0]Fword;
input [11:0]Pword;

output DA_Clk;
```

```
output [9:0]DA_Data;
```

以下只需按照图 6.4 -2 进行编写。相位累加器此处即为一个 32bit 的加法器。

```
reg [31:0]Fre_acc;

always @(posedge Clk or negedge Rst_n)
if(!Rst_n)
    Fre_acc <= 32'd0;
else if(!EN)
    Fre_acc <= 32'd0;
else
    Fre_acc <= Fre_acc + Fword;
```

查找表地址生成，此处即为 12bit 的加法器。这里直接截取 32 位累加器结果中的高 12 位作为 ROM 的查询地址，这样产生的误差会对频谱纯度有影响，但是对波形的精度的影响是可以忽略的。

```
reg [11:0]Rom_Addr;

always @(posedge Clk or negedge Rst_n)
if(!Rst_n)
    Rom_Addr <= 12'd0;
else if(!EN)
    Rom_Addr <= 12'd0;
else
    Rom_Addr <= Fre_acc[31:20] + Pword;
```

DA 数据输出时钟模块使能，通过选择器来进行控制。

```
assign DA_Clk = (EN)?Clk:1'b1;
```

6.4.1.3 制作波形数据存储器

现在只需要例化存有波形文件的 ROM 即可。其中单端口的 ROM 主要配置数据如图 6.4 -77 所示，其初始化文件选为已经生成的正弦 mif 文件。此处 mif 位宽为 10，深度为 4096。

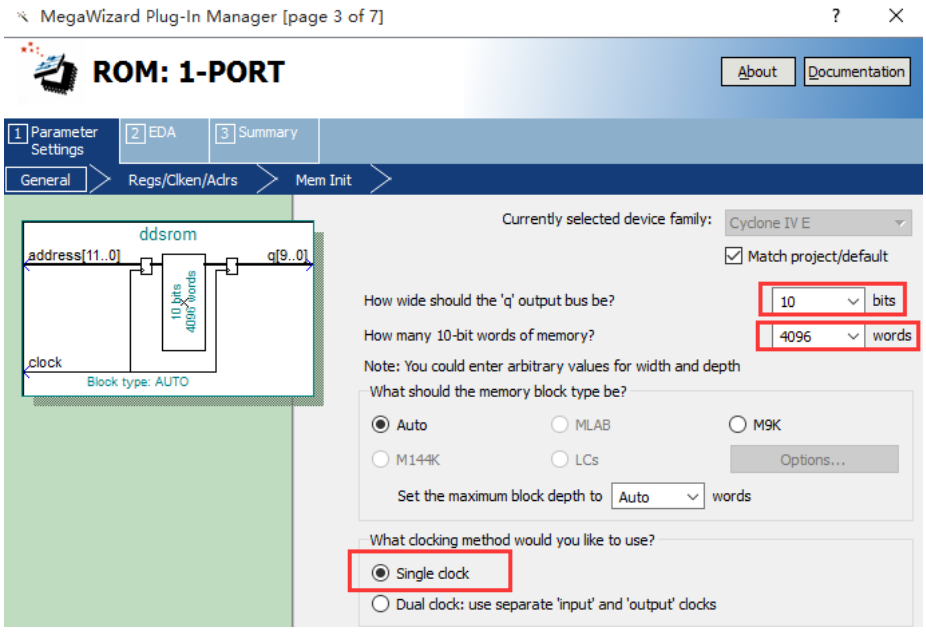

图 6.4 -7　ROM 主要配置参数

这样例化到 DDS_Module 中即为如下内容。

```
ddsrom ddsrom(
        .address(Rom_Addr),
        .clock(Clk),
        .q(DA_Data)
);
```

本模块编译无误后，点击 RTL Viewer 后可以看到如图 6.4 -8 所示的模块逻辑电路图，可与 DDS 原理图对比。这里有两个选择器的原因是加了使能端的缘故，如果将使能端去掉即可看到如图 6.4 -9 所示的电路图

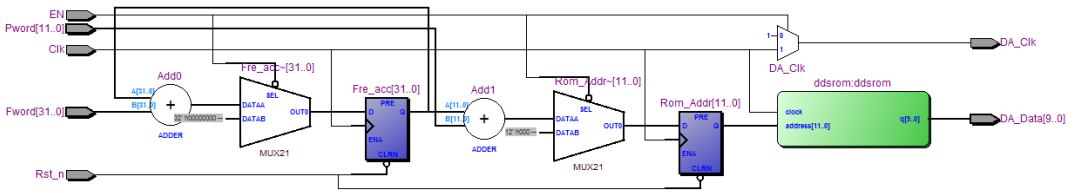

图 6.4 -8　DDS 模块设计逻辑电路图

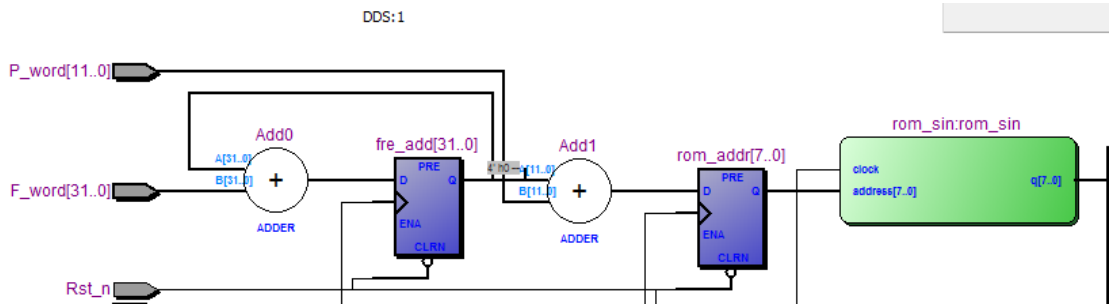

图 6.4 -9　DDS 模块简化逻辑电路图

设置此文件为顶层进行功能仿真。激励文件中除了产生正常的时钟以及模块例化调用，还需使能模块以及设置自加字。此处简化将 Fword 设置为固定值'd5000,。

```
initial begin
    Rst_n = 1'b0;
    EN = 1'b0;
    Fword = 32'd0;
    Pword = 12'd0;

    #(`clk_period*20)
    Rst_n = 1'b1;
    #(`clk_period*20)
    EN = 1'b1;
    Fword = 32'd5000;

    #(`clk_period*2000000)
    $stop;
end
```

6.4.1.4 DDS 模块仿真验证

编译无误后设置好仿真脚本并进行仿真，可以看到如图 6.4 -10 所示的功能仿真波形文件，可以看出波形数据正常。放大仿真开始位置，如图 6.4 -11 所示，可以看出输出 DA 时钟使能设计也正常。初始值为'd511 也就是'h1ff，与初始化 mif 文件一致。

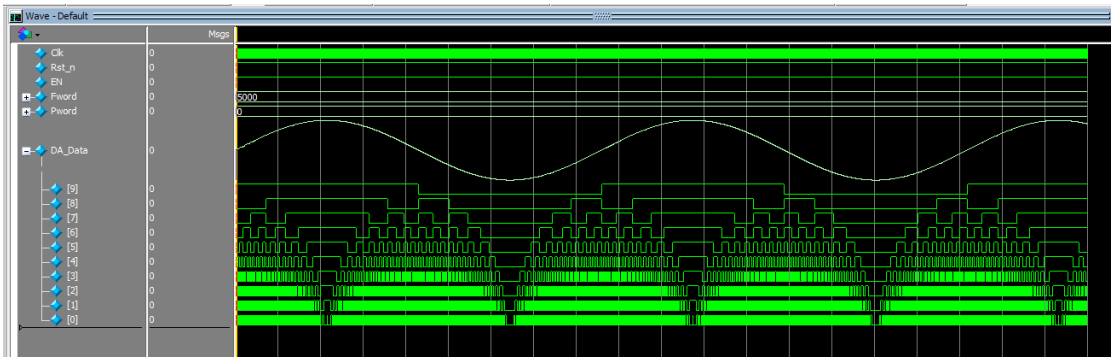

图 6.4 -10　DDS 模块功能仿真波形

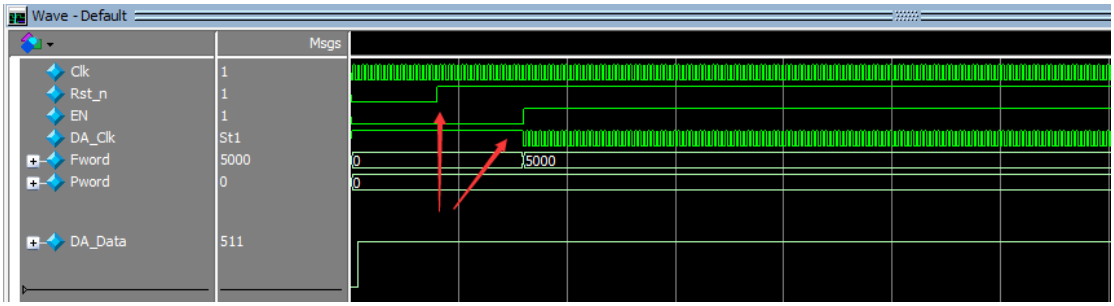

图 6.4 -11　DDS 模块部分放大波形文件

这样一个 DDS 模块即设计完成，这里可以自行修改 Pword 的值进行观察波形相位是否发生相应变化。

6.4.2 数模转换器（DAC）驱动模块设计

6.4.2.1 TLV5618 DAC 驱动模块

这里采用 DAC 芯片为 AC620 开发套件上的 TLV5618。其中 TLV5618 模块的设计与实现在第五章已经详细阐述，此处不再对本部分进行解释。其模块接口示意图 6.4 -12 所示，其接口对应的功能描述如表 6.4-2 所示。

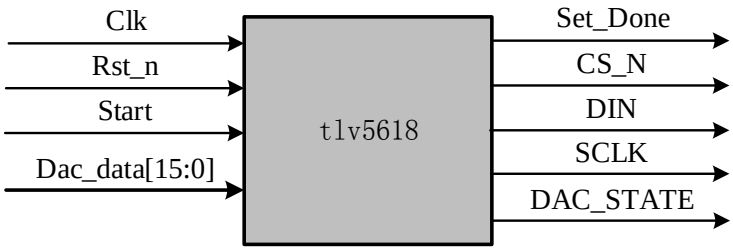

图 6.4 -12　DA 模块端口接口示意图

表 6.4-2　DA 模块接口功能描述

端口名称	/O	端口功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz，
Rst_n	I	控制器复位，低电平复位
Start	I	转换启动信号，每次该信号的一个时钟周期的高脉冲启动一次数据到 DAC 芯片的传输。注意，该信号只能为 1 个时钟周期的高电平脉冲。
Dac_data[15:0]	I	控制字和 DAC 数据端口，高 4 位为控制字，低 12 位为数据
Set_Done	O	更新 DAC 完成标志，每次完成更新产生一个高电平脉冲，脉冲宽度为 1 个时钟周期
CS_N	O	TLV5618 的 CS_N 接口
DIN	O	TLV5618 的 DIN 接口
SCLK	O	TLV5618 的 SCLK 接口
DAC_STATE	O	模块状态标识，低电平时为忙标志，高电平为空闲状态

6.4.2.2 多通道数据输出实现

本节需实现的是双通道的信号发生器。而 DAC 的两个通道的数据是通过一个数据通道+不同的控制字实现的，因此这里就要轮转控制两个通道并输入相应的控制字。将此模块命名为 DAC_2CH，该模块例化了 TLV5618 模块作为 DAC 的实际驱动模块。

另外需要说明的是，本例中的 DDS 模块生成的数据为 10 位，而 DAC 是 12 位数据位宽的，因此在实际连接时，将 DDS 输出的数据左移了两位，转换为 12 位的数据，然后再送给 DAC 进行数模转换。其模块接口示意图如图 6.4 -13 所示。

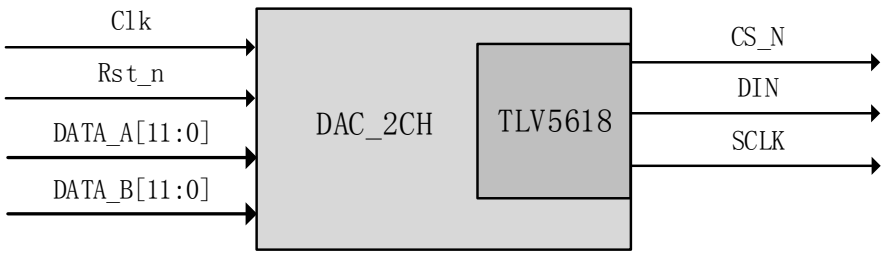

图 6.4 -13　DAC 通道选择模块接口示意图

其每个接口功能描述如表 6.4-3 所示。

表 6.4-3　DAC 通道选择模块接口功能描述

端口名称	I/O	功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz
Rst_n	I	控制器复位，低电平复位
DATA_A	I	DDS 产生通道 A 的数据
DATA_B	I	DDS 产生通道 B 的数据
CS_N	O	TLV5618 的 CS_N 接口
DIN	O	TLV5618 的 DIN 接口
SCLK	O	TLV5618 的 SCLK 接口

由以上表分析可知其端口列表如下：

```
input Clk;
input Rst_n;
input [11:0]DATA_A,DATA_B;//每个通道的数据
output CS_N;
output DIN;
output SCLK;
```

这里在调用 TLV5618 时，设计器件一直处于工作状态，定义 state 信号来确定 TLV5618 驱动模块的工作状态，state 为 0 表示当前 DAC 空闲，state 为 1 表示当前正在进行数据的传输。每当 DAC 控制器处于空闲时，就切换状态到数据

传输状态，然后根据 state 的值来控制转换启动信号。每当 state 为 0，即 DAC 控制器为空闲时，就设置 Start 信号为高电平，启动一次传输，该部分代码如下所示。

```
    tlv5618 tlv5618(
        .Clk(Clk),
        .Rst_n(Rst_n),

        .DAC_DATA({CtrlWord,DATA}),
        .Start(Start),
        .Set_Done(UpdateDone),
        .DIV_PARAM(8),

        .CS_N(CS_N),
        .DIN(DIN),
        .SCLK(SCLK),
        .DAC_State(DAC_State)
    );

    reg state;//state 为 0 表示当前 DAC 空闲，state 为 1 表示当前正在进行数据的传输
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        state <= 1'b0;
    else if(state == 0)//每当 DAC 空闲时，就切换状态到数据传输状态
        state <= 1'b1;
    else if(Set_Done)
        state <= 1'b0;

    //根据 DAC 控制状态，设置 DAC 传输数据启动信号
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        Start <= 1'b0;
    else if(state == 0)
        Start <= 1'b1;
    else
        Start <= 1'b0;
```

由于 TLV5618 采用的是串行接口与 FPGA 进行数据传输的，一次只能传输一个通道的数据，那么要想实现双通道都能够输出波形，就需要通过轮询的方式分时给 DAC 的每个通道写入需要输出的值。实现轮询，就是在时钟上升沿到来时，每当一次 TLV5618 转换完成后，就开始选通下一个通道进行数据输出。

```
    /*每完成一次转换，通道编号加 1，由于总共为 2 两个通道，所以，Current_CH 将循环在 0 和 1 之间变化*/
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        Current_CH <= 0;
    else if(Set_Done)
        Current_CH <= Current_CH + 1'b1;

    //根据通道编号从两个 DDS 数据源中选择 DAC 数据
    always@(*)
    begin
        case(Current_CH)
            0: DATA = DATA_A;
            1: DATA = DATA_B;
            default:DATA = DATA_A;
        endcase
    end
```

在选通对应通道后，最终传递给 DAC 控制模块进行传输的数据应该是 16 位，高四位为控制字，包含通道信息，低 12 位为 DAC 期望输出的电压编码，而控制字 CtrlWord 则根据当前选择的通道不通，其值不同。具体为，当通道选择为 1 时，选择通道 B，此时控制字为 4'b1100，即写数据到通道 B 缓冲区，当通道选择为 0 时，选择通道 A 并同时更新缓冲器中的数据到 DAC 输出，此时控制字为 4'b0101。

```
assign CtrlWord = Current_CH?4'b0101:4'b1100;
```

最终传递给 DAC 驱动模块 TLV5618 的数据应该是由 4 位的 CrtlWord 和 12 位的 DATA 拼接而成的 16 位数据，如下所示：

```
.DAC_DATA({CtrlWord,DATA}),
```

至此，我们就完成了两个 DDS 生成的数据分别传递给 DAC 芯片两个输出通道的功能。后续，DDS 模块输出什么信号，该信号就将直接传递给 DAC 对应通道并最终输出在 DAC 的模拟电压输出管脚。

6.4.3 串口命令接收与解析

这里使用的串口接收模块波特率为 9600，具体实现方式在第三篇已经阐述，此处不再详细解释。其模块接口示意图如图 6.4 -14 所示，接口对应功能描述如表 6.4-4 所示。

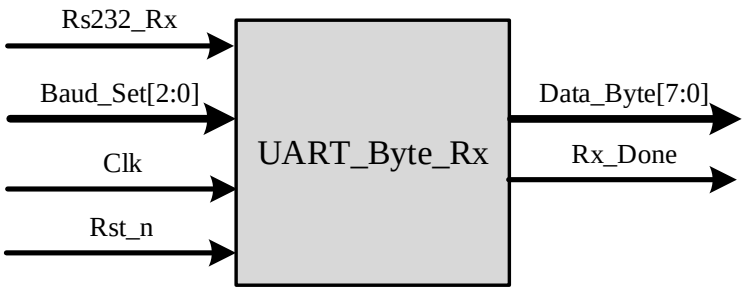

图 6.4 -14　串口接收模块接口示意图

表 6.4-4　串口接收模块接口功能描述

信号名称	I/O	功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz
Rst_n	I	控制器复位，低电平复位
Rs232_Rx	I	串行数据输入
Band_Set	I	波特率选择信号
Data_Byte	O	并行数据输出
Rx_Done	O	接收结束标志信号

6.4.3.1 信号发生器之自定义帧

由于 DDS 输出信号的频率、相位、幅度都需要能够调节，因此系统需要有一个较为灵活的输入设备来完成对应控制信号的接收。传统的按键在面对如此多的输入量时，略显力不从心。而且，信号发生器设备一般都需通过上位机来控制，所以本系统中直接使用串口来接收控制信号，控制信号的产生可以是 PC 机或其他带有串口设备的智能控制器。

在通信行业，存在各种符合一定标准的通信协议，如最经典的 MODBUS 协议，通过收发双方遵循相同的协议，可以实现两者间数据的可靠传输。当然，在某些情况下，也可以根据实际应用情况，定义一些简化的自定义协议。

本系统采用自定义帧的方式来进行数据传输。其中串口命令数据的格式如表 6.4-5 所示。一帧数据为 8 个字节，包含帧头+地址+数据+帧尾四个组成部分，无数据帧校验部分。通过发送不同地址和数据部分，可以实现对特定地址寄存器写入对应的数据。

表 6.4-5　自定义帧数据格式

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
帧头 0	帧头 1	寄存器地址	数据低字节	数据中低字节	数据中高字节	数据高字节	帧尾
0xAA	0x03	Addr	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]	0x88

不同的寄存器地址对应不同通道的不同的被控制数。系统设计了 6 个实体寄存器分别存储每个通道的相关控制参数。6 个寄存器分为 2 组，分别对应了 2 个通道。每组包含 3 个寄存器，即频率控制字寄存器、相位控制字寄存器和幅度控制字寄存器。具体的各实体控制寄存器地址与对应功能如表 6.4-6 所示。当想控制某个通道的信号参数时，直接写实体寄存器，控制模块就会立即更新控制参数到对应通道上。

表 6.4-6 实体寄存器地址与功能对应表

地址（16 进制）	寄存器名	寄存器位宽	寄存器功能
10	Fword0	32	通道 0 频率控制字寄存器
11	Fword1	32	通道 1 频率控制字寄存器
12	Pword0	12	通道 0 相位控制字寄存器
13	Pword1	12	通道 1 相位控制字寄存器
14	Aword0	4	通道 0 幅度控制字寄存器
15	Aword1	4	通道 1 幅度控制字寄存器

现在举例说明如何使用实体寄存器控制对应控制字。

6.4.3.1.1 使用实体寄存器控制某通道信号频率

要求：设置通道 0 的频率为 100Hz

地址：0x10

控制字：$Fword = 2^N \times \frac{F_{out}}{F_{clk}} = 2^{32} \times \frac{100}{50\_000\_000}$ =≒8590

8590 换算成 16 进制就是 0x218E。因此指令的数据段为 8E 21 00 00

这样最终发送的指令为：AA 03 10 8E 21 00 00 88

6.4.3.1.2 使用实体寄存器控制某通道信号相位

要求：设置通道 1 的相位为 90 度

地址：0x13

控制字：$Pword = 2^M \times \frac{P_{out}}{360} = 2^{12} \times \frac{90}{360} = 1024$

1024 换算成 16 进制就是 0x400。因此数据段为 00 04 00 00

这样最终发送的指令为：AA 03 13 00 04 00 00 88

6.4.3.1.3 使用实体寄存器控制某通道信号幅度

系统设置支持 8 级信号幅度调节，此 8 级幅度的信号控制字与满幅的对应关系如表 6.4-7 所示。

表 6.4-7 控制字与幅度关系

控制字	信号缩减比例	最大输出电压
0	满幅	3.3
1	半幅	1.65
2	1/4 满幅	0.825
3	1/8 满幅	0.413
4	1/16 满幅	0.206

5	1/32 满幅	0.103
6	1/64 满幅	0.052
7	1/128 满幅	0.026

(注意：虽然系统设计支持 8 级调幅，但是由于本系统的调幅原理为简单的将数据右移，以达到除 2 的效果，从而减小实际送给 DAC 的数据值。因此，数据在按照倍数减小的同时，其精度也会降低，输出模拟信号也会因为精度的丢失而失真。实测幅度衰减到 1/8 时，波形失真就已经无法接受了。在实际的调幅电路中，应该采用模拟电路对信号进行放大或衰减。)

例如：设置通道 1 的幅度为半幅

地址：0x15

控制字：01 因此数据段为 01 00 00 00

这样最终发送的指令为 AA 03 15 01 00 00 00 88

6.4.3.1.4 使用影子寄存器实现多通道参数同步更新

另外，为了实现同时控制所有通道的所有参数同时更新，特为以上所有实体寄存器设置了影子寄存器。当需要预先设置某通道的数据，但并不立即执行时，可使用影子寄存器功能。

影子寄存器会预先存储所有的设置数据，这些数据不会立即更新到每个模块，而是在收到更新指令后，系统同时更新指令选择的所有影子寄存器中的值到实体寄存器，这样所有通道的参数就是被同时更新的。在操作时只需要先将要更新的数据逐次写入每个通道需要更新的寄存器的影子寄存器中，然后发出更新指令，则系统会自动将影子寄存器中的数据同时更到实体寄存器。

更新指令由更新寄存器完成，更新寄存器每一位对应了一个功能寄存器，这样，就可以通过对更新寄存器写入不同的值来选择具体更新哪些影子寄存器中的值到实体寄存器中。影子寄存器地址、名称与功能对应表如表 6.4-8 所示。

表 6.4-8 影子寄存器名称与功能对应表

地址（16 进制）	寄存器名	寄存器位宽	寄存器功能
00	s_Fword0	32	通道 0 频率控制字影子寄存器
01	s_Fword1	32	通道 1 频率控制字影子寄存器
02	s_Pword0	12	通道 0 相位控制字影子寄存器
03	s_Pword1	12	通道 1 相位控制字影子寄存器
04	s_Aword0	4	通道 0 幅度控制字影子寄存器
05	s_Aword1	4	通道 1 幅度控制字影子寄存器

更新寄存器以及通道控制地址与功能对应关系，如表 6.4-9 所示

表 6.4-9 更新寄存器以及通道控制地址与功能对应表

地址（16 进制）	寄存器名	寄存器位宽	寄存器功能
06	CH_Sync	4	通道使能（兼同步功能）寄存器
07	Ctrl	8	影子寄存器更新控制寄存器

Ctrl 寄存器每一位对应的功能如表 6.4-10 所示。

表 6.4-10 更新控制寄存器位功能

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
X	X	更新 Aword1	更新 Aword0	更新 Pword1	更新 Pword0	更新 Fword1	更新 Fword0

当该位为 1 时，使能更新对应的影子寄存器到实体寄存器中，为 0 时则不使能更新。

在使用影子寄存器更新多个通道多个寄存器的操作顺序如下：

1. 写一个或多个通道中一个或多个影子寄存器的值

2. 发出更新指令

例如，需要设置通道 0 的频率为 100Hz，相位为 90 度，幅度不变，设置通道 1 的频率为 100Hz，相位为 180 度，幅度为半幅。由于需要保持通道 0 的幅度不变，因此该寄存器不可直接从影子寄存器中更新，所以更新寄存器指令时数据应为 8'b00101111，则可按照如下指令顺序进行发送：

AA 03 00 8E 21 00 00 88 (设置通道 0 的频率为 100Hz)

AA 03 02 00 04 00 00 88 (设置通道 0 的相位为 90 度)

AA 03 01 8E 21 00 00 88 (设置通道 1 的频率为 100Hz)

AA 03 03 00 08 00 00 88 (设置通道 1 的相位为 180 度)

AA 03 05 01 00 00 00 88 (设置通道 1 的幅度为半幅)

AA 03 07 2F 00 00 00 88 (更新影子寄存器中的数据到实体寄存器，其中 Aword0 不更新)

6.4.3.1.5 使用控制寄存器实现通道的使能和关闭

通过以上寄存器，我们能够控制 DDS 模块的频率控制字和幅度控制字，以及简单的控制输出信号幅度。此外，我们还需要能够控制 DDS 的输出和关闭。为此，特设置了一个控制寄存器，地址为 06。该寄存器控制 2 个通道的运行和停止，共有 2 个有效位，每一位对应一个通道的开关。当需要同步两个通道输出时，也可使用这个寄存器来实现，实现方式为首先设置这 2 个通道的寄存器对应的控制位为 0，即关闭两个通道的输出，然后重新设置这 2 个通道的寄存器对应位为 1，即可实现同步。

所以，在上述 5 个步骤操作完成后，我们还需要进行同步操作，所谓同步就是先关闭两个通道，然后再同时打开。同步时的指令如下所示：

AA 03 06 00 00 00 00 88(关闭两个通道)

AA 03 06 03 00 00 00 88 (开启两个通道)

6.4.3.2 串口数据帧接收

使用前面定义的数据帧格式，就需要一次发送多个字节的数据，这样就不排除多个字节发送错误或者多发、少发。例如，本来一帧数据应该是 8 个字节，但

是 PC 端在发送了 2 个字节的数据后被人为中断，然后重新启动该帧数据的发送。对于 FPGA 的接收端来说，由于之前已经接收了 2 个字节的数据，如果不做合理的判断的话，那么新接收到的一帧数据会被认为是之前一帧数据的从第三个数据开始的后续内容，此时 FPGA 端接收的数据就被误判 了，所以使用相邻字节间间隔时长来作为一帧数据传输完成或失败的依据。例如，设置相邻两字节间的最大间隔时间不能超过 3.5 个字节传输时间长度，如果超过 3.5 个字节的时间传输长度，则认为一帧数据传输完成。通过此种方式，就能强制的判断帧的结束或者中断了。

这里新建 uart_rx_frameend.v，来进行帧格式判断，该模块接口示意图如图 6.4 -15 所示。

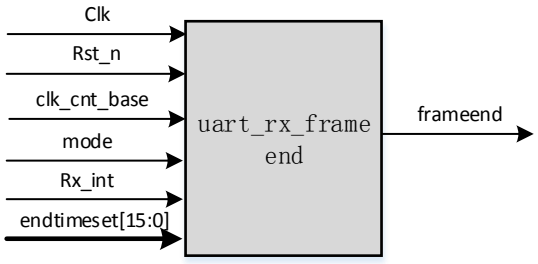

图 6.4 -15　帧判断模块接口示意图

帧判断模块的接口功能描述如表 6.4-11 所示。

表 6.4-11 帧判断模块接口功能描述

端口名称	I/O	功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz，
Rst_n	I	控制器复位，低电平复位
clk_cnt_base	I	基本计数时钟，帧结束计数器计数时基
mode	I	模式 0，使用内部 1K 的基准时钟，模式 1，使用外部计数时钟
Rx_int	I	字节接收成功信号

endtimeset	I	帧结束判定时间设置
frameend	O	帧结束标志信号

帧结束判断模块实际是一个定时器，该定时器的计时基准时间以由其他模块输入，也可以在内部通过对全局时钟分频得到。模块内部默认产生一个 1KHz 的单时钟周期脉冲时钟，并使用一个 mode 信号来选择内部产生的 1K 频率时钟或外部输入时钟信号作为计数器计数时钟。对于常见的 9600 波特率的串口传输，使用内部 1KHz 的时钟信号即可，如果串口传输波特率大于 9600，则需要使用外部基准时钟。

```
    reg [15:0]internal_base_cnt;
    reg internal_base_clk;   //内部计数基准时钟
    wire base_clk; //定时器计数基准时钟

    //内部 1KHz 基准计数时钟计数器
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        internal_base_cnt <= 16'd0;
    else if(internal_base_cnt == 49_999)
        internal_base_cnt <= 16'd0;
    else
        internal_base_cnt <= internal_base_cnt + 1'd1;

    //   产生内部 1KHz 基准时钟
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        internal_base_clk <= 1'd0;
    else if(internal_base_cnt == 49_999)
        internal_base_clk <= 1'd1;
    else
        internal_base_clk <= 1'd0;

//通过模式选择位选择使用内部计数基准时钟或外部计数基准时钟
//模式 0，使用内部 1K 的基准时钟，模式 1，使用外部计数时钟
    assign base_clk = mode?clk_cnt_base:internal_base_clk;
```

串口数据标志接收部分，每当接收到一字节的数据后均会产生一个时钟周期的高电平标志信号。此处以 cnt_state 为标志来定义状态进而使能计数器。每当接收到 1 字节数据即开始计数，当帧超时则停止计数。

```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    cnt_state <= 1'b0;
else if(Rx_int) //接收到数据。开始计数
    cnt_state <= 1'd1;
else if(frameend)    //帧超时，停止计数
    cnt_state <= 1'b0;
```

计数器使能时每当到来一个参考时钟就将计数器加一。当接收到新的字节数据时或者当达到预设的帧结束判定时间就将计数器清零。且当到达帧结束判定时间时产生一帧数据接收完成标志信号。

```
//计数进程
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    cnt <= 16'd0;
//接收到数据或者字节间间隔时长已达设定超时值，清零计数器
else if(Rx_int || (base_clk && (cnt == endtimeset)))
    cnt <= 16'd0;
else if(base_clk && cnt_state)
    cnt <= cnt + 16'd1;

//字节间时间间隔时长已经达到设定值，产生帧结束信号
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    frameend <= 1'b0;
else if(base_clk && (cnt == endtimeset))
    frameend <= 1'b1;
else
    frameend <= 1'b0;
```

为了简化测试激励，这里先使用外部时钟进行功能仿真。因此在激励文件中例化待测试文件如下所示。

```
uart_rx_frameend uart_rx_frameend(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .clk_cnt_base(clk_cnt_base),
    .mode(1),
    .Rx_int(Rx_int),
    .endtimeset(endtimeset),
    .frameend(frameend)
```

```
);
```

激励文件中除产生正常的系统时钟 50MHz 以外，还产生了 4.5MHz 的外部时钟。

```
initial clk_cnt_base = 0;
always begin
    clk_cnt_base = 1;
    #20;
    clk_cnt_base = 0;
    #200;
end
```

循环产生接收到的数据标志信号 36 次，且最后一个个延迟足够长的时间。这里也就是用时钟 clk_cnt_base 计数 20 以上即可（大约 5000ns）。

```
integer i;

initial begin
    Rst_n = 0;
    endtimeset = 19;
    Rx_int = 0;
    #201;
    Rst_n = 1;
    #200;
    for(i=0;i<36;i=i+1)begin
        Rx_int = 1;
        #20;
        Rx_int = 0;
        #2000;
    end
    #200000;
    for(i=0;i<36;i=i+1)begin
        Rx_int = 1;
        #20;
        Rx_int = 0;
        #2000;
    end
    #200000;
    $stop;
end
```

设置好仿真脚本以后，开始运行仿真，大致在图 6.4 -16 可以看出 Rx_int 信号产生正常，且计数使能信号正常，在一次帧数据传输结束后经过一定时间后 frameend 有一个标志信号。放大部分数据，在图 6.4 -17 查看此时的 cnt 信号可以看出 19 时产生信号，符合预期设计。

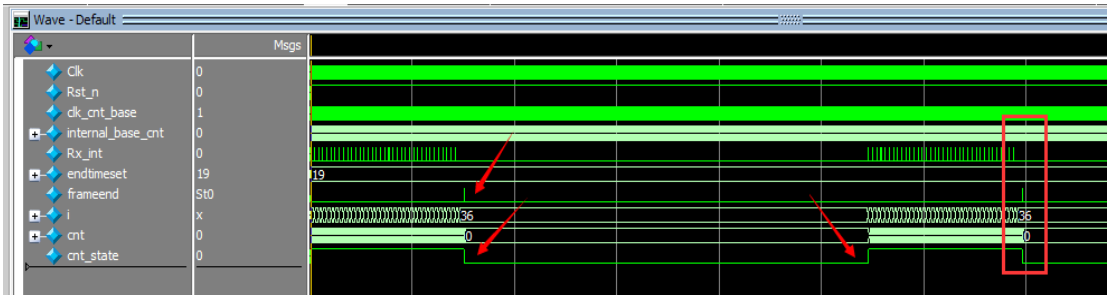

图 6.4 -16　帧判断模块功能仿真波形

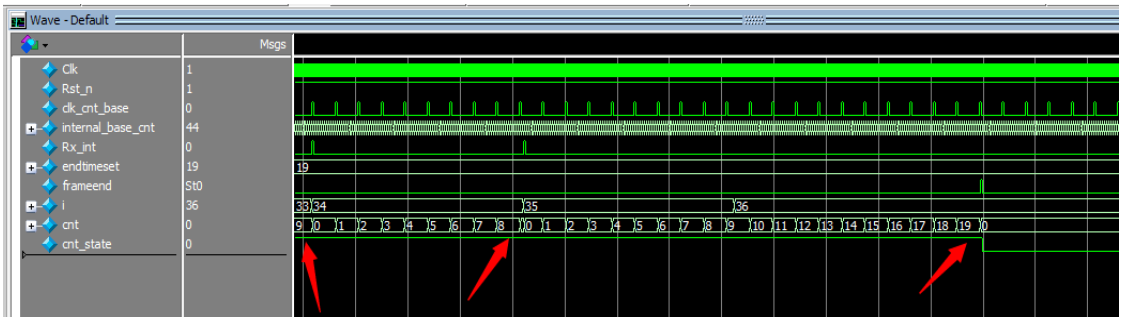

图 6.4 -17　帧判断模块部分波形

6.4.3.3 串口数据解析

由上面自定义了数据格式，总共为 8 个 8 位数据，因此先用移位寄存器把八个数据移入进来。串口数据解析模块接口示意图如图 6.4 -18 所示。

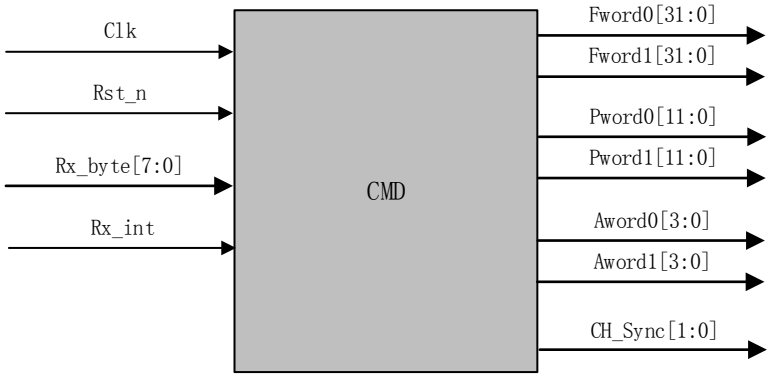

图 6.4 -18 CMD 模块接口示意图

因此其端口名称及功能描述如表 6.4-12 所示。

表 6.4-12 模块接口及功能描述

端口名称	I/O	功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz，
Rst_n	I	控制器复位，低电平复位
Rx_Byte	I	串口接收模块接收到的数据字节
Rx_Int	I	串口接收模块接收到数据标志
Fword0	O	通道 0 频率控制字
Fword1	O	通道 1 频率控制字
Pword0	O	通道 0 相位控制字
Pword1	O	通道 1 相位控制字
Aword0	O	通道 0 幅度控制字
Aword1	O	通道 1 幅度控制字
CH_Sync	O	通道选择控制字

首先利用移位寄存器将接收到的数据放到一个寄存器中。这里定义了一个清零端，其作用是在一帧数据接收完成后将计数器以及移位寄存器数据清零。

```
reg [63:0]shift_data;
reg clr;

//移位寄存数据，接收串口发送的数据并存入移位寄存器中
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    shift_data <= 64'd0;
else if(clr)
```

```
        shift_data <= 64'd0;
    else if(Rx_Int)
        shift_data <= {Rx_Byte,shift_data[63:8]};
```

对接收到的数据个数进行计数，直到产生清零信号。

```
    //计数接收到的数据总数
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        data_cnt <= 4'd0;
    else if(clr)
        data_cnt <= 4'd0;
    else if(Rx_Int)
        data_cnt <= data_cnt + 1'b1;
```

每当 frameend 变为高电平，即代表一次数据传输结束。

注意此处一次数据传输结束，可能是正常的一帧数据，也可能是不正常的数据接收。这样就需要对此次接受的数据进行一定的分析判断是否为正常的一帧数据。本实验中定义的正常一帧数据为表 6.4-5 所示，数据个数为 8 且帧头为'haa、'h03，帧尾分别为'h88。只有接收到的数据帧中对应位置的数据与定义的帧格式相匹配才认为是正常的数据，可以根据寄存器地址来进行相对应的操作。一次传输结束后就将计数器清零等待下一次传输。

```
always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)begin
        Fword0 <= 0; Fword1 <= 0; Pword0 <= 0; Pword1 <= 0;
        Aword0 <= 0; Aword1 <= 0; s_Fword0 <= 0; s_Fword1 <= 0;
        s_Pword0 <= 0; s_Pword1 <= 0; s_Aword0 <= 0; s_Aword1 <= 0;
        CH_Sync <= 2'b00; clr <= 1'b0;
    end
    else if(frameend)begin //帧结束，将接收到的数据提取出来，根据其中的地址段和数据段，写数据到对应寄存器
        clr <= 1'b1;
        if(data_cnt == 8 && shift_data[7:0] == Header0 && shift_data[15:8] == Header1 && shift_data[63:56] == Tail)begin
            case(shift_data[23:16])
                8'h10:Fword0 <= shift_data[55:24];
                8'h11:Fword1 <= shift_data[55:24];
```

```
                8'h12:Pword0 <= shift_data[35:24];
                8'h13:Pword1 <= shift_data[35:24];

                8'h14:Aword0 <= shift_data[27:24];
                8'h15:Aword1 <= shift_data[27:24];

                8'h00:s_Fword0 <= shift_data[55:24];
                8'h01:s_Fword1 <= shift_data[55:24];

                8'h02:s_Pword0 <= shift_data[35:24];
                8'h03:s_Pword1 <= shift_data[35:24];

                8'h04:s_Aword0 <= shift_data[27:24];
                8'h05:s_Aword1 <= shift_data[27:24];

                8'h06:CH_Sync <= shift_data[25:24];

                8'h07:begin

                    Fword0 <= shift_data[24]?s_Fword0:Fword0;
                    Fword1 <= shift_data[25]?s_Fword1:Fword1;

                    Pword0 <= shift_data[26]?s_Pword0:Pword0;
                    Pword1 <= shift_data[27]?s_Pword1:Pword1;

                    Aword0 <= shift_data[28]?s_Aword0:Aword0;
                    Aword1 <= shift_data[29]?s_Aword1:Aword1;
                end
                default:;
            endcase
        end
    end
    else
        clr <= 1'b0;
```

在调用写好的一次数据传输结束模块时，使用内部的 1K 时钟，且将帧结束判定时间设置为 9。这里的时钟速度选择以及判定时间设置需根据不同的情况采用不同的设置。

```
    uart_rx_frameend uart_rx_frameend(
        .Clk(Clk),
        .Rst_n(Rst_n),
```

```
        .clk_cnt_base(1'b0),
        .mode(1'b0),
        .Rx_int(Rx_Int),
        .endtimeset(4'd9),
        .frameend(frameend)
    );
```

由待测试文件可以看出如果产生帧结束标志信号会至少经过 9/1000s，所以激励文件发送一次正常的数据时先发送 AA 03 10 8E 21 00 00 88，然后延时 0.01s，再发送 83 00 数据后再延迟 0.01s， 0.01s 刚好大于 0.009s。具体实现代码如下所示。

```
    initial begin
        Rst_n = 1'b0;
        Rx_Byte = 8'd0;
        Rx_Int = 1'b0;
        #(`clk_period*20 + 1 )
        Rst_n = 1'b1;

        #(`clk_period*50);
        Rx_Byte = 8'haa;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*5000);
        Rx_Byte = 8'h03;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*1500);
        Rx_Byte = 8'h10;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500);
```

```
        Rx_Byte = 8'h8e;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500);
        Rx_Byte = 8'h21;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500);
        Rx_Byte = 8'h00;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500);
        Rx_Byte = 8'h00;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500);
        Rx_Byte = 8'h88;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500000);
    //////////////////////////
        #(`clk_period*1000);
        Rx_Byte = 8'h83;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;
```

```
        #(`clk_period*15000);
        Rx_Byte = 8'h00;
        #`clk_period;
        Rx_Int = 1'b1;
        #`clk_period;
        Rx_Int = 1'd0;

        #(`clk_period*500000);
        $stop;
    end
```

编译无误后设置好仿真脚本，可以看到如图 6.4 -19 所示的现象，每接收一个字节数据，Rx_Int 信号均会产生一个系统周期的高电平，且数据计数器 cnt 能够正常的自加及清零。一帧数据接收成功后控制字 Fword0 值被正确更新。当一帧数据接收完成后，再次发送一字节数据时 cnt 又开始重新计数。

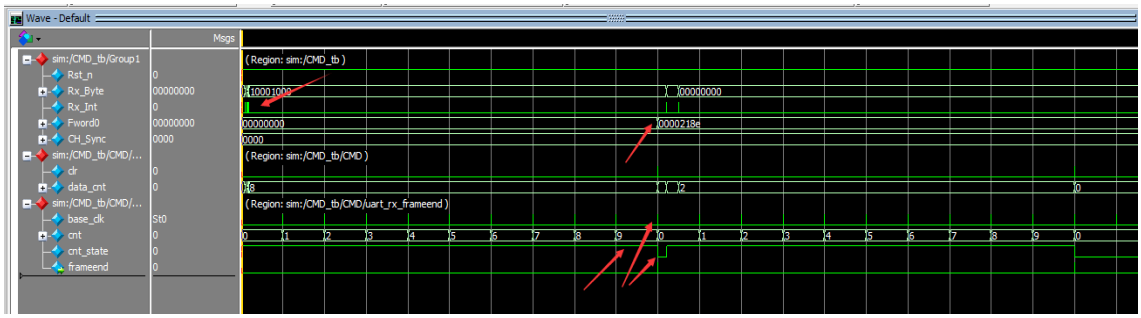

图 6.4 -19 串口接收即命令解析模块功能仿真波形

如果想看到一次数据发送中 data_cnt 以及 shift_data 细节，则可以增加 base_clk 频率并修改计数器，以便于仿真查看现象。修改后运行仿真，可以看到如图 6.4 -20 的结果。这里修改频率控制字的同时也清零了计数器值，从仿真波形中看出该种设置下同样可以正常实现帧结束位的判定。

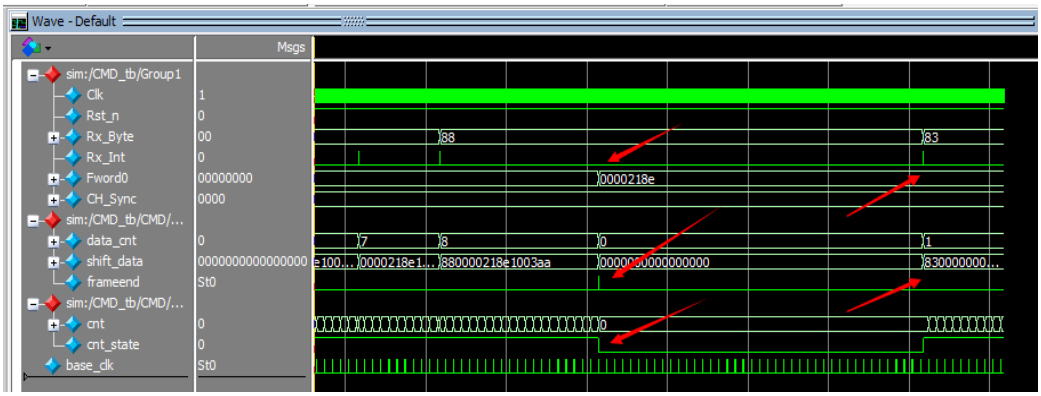

图 6.4 -20　模块部分仿真波形

6.4.4 信号发生器顶层设计

综合以上的分析，可以得出其中模块顶层如下图 6.4 -21 所示，连接时将内部信号全部定义为 wire 即可，其接口功能描述如表 6.4-13 所示。

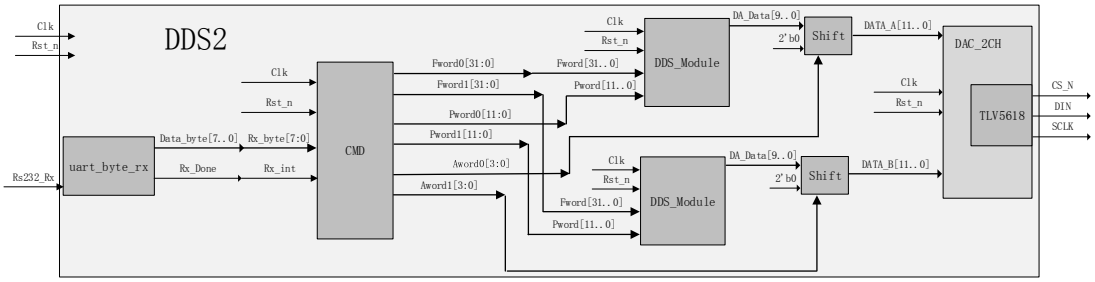

图 6.4 -21　顶层模块接口示意图

表 6.4-13 顶层模块接口及功能描述

端口名称	I/O	功能描述
Clk	I	为控制器的工作时钟，频率为 50MHz
Rst_n	I	控制器复位，低电平复位
Rs232_Rx	I	串口接收接口，接收串口助手发送数据
CS_N	O	TLV5618 的 CS_N 接口
DIN	O	TLV5618 的 DIN 接口

SCLK	O	TLV5618 的 SCLK 接口

前面提到，通过将 DDS 的输出以右移的方式实现除 2，以达到输出信号幅度减半的效果，从而实现调幅的目的，该方法实现代码如下所示。

```
wire [9:0]DA_Data0,DA_Data1;      //DDS 输出的波形数据
wire [11:0]DATA_A,DATA_B;     //送给 DAC 通道的波形数据（幅度受控）

wire DA_Clk0,DA_Clk1;

assign DATA_A = {DA_Data0,2'b00} >> Aword0;
assign DATA_B = {DA_Data1,2'b00} >> Aword1;
```

根据实际使用情况将各个模块例化进来，例化后的顶层应为如图 6.4 -22 所示。

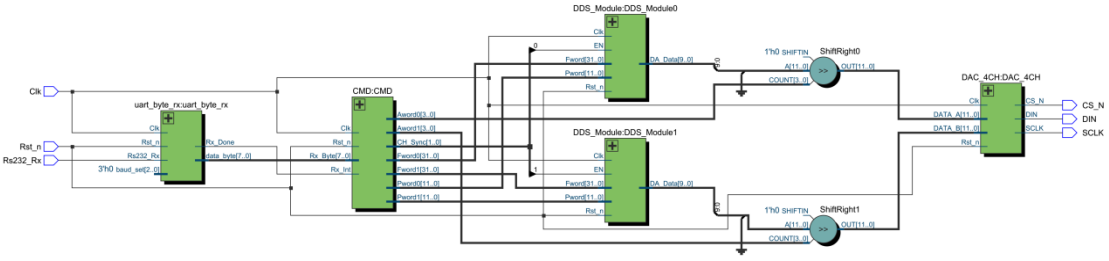

图 6.4 -22　顶层文件 RTL Viewer 视图

6.4.5 系统测试

经过上述工作，DDS2 的所有代码都已经设计完毕由于之前已经对子模块分别完成了仿真验证，这里对顶层模块暂不进行仿真。接下来介绍板级测试和验证方法。

1、按照 AC620 开发板的引脚分配表分配正确的引脚，本设计实例的硬件分配如下表所示：

信号名称	信号方向	对应 FPGA 管脚
Clk	Input	PIN_E1
Rst_n	Input	PIN_E16
Rs232_Rx	Input	PIN_B5
SCLK	Output	PIN_E7
CS_N	Output	PIN_E8

DIN	Output	PIN_C8

2、将整个工程修改编译至没有错误，下载 sof 文件到 FPGA 中。

3、使用 MicroUSB 数据线连接 PC 的 USB 口和 AC620 开发板的 USB 串口电路，如图 6.4 -23 所示。

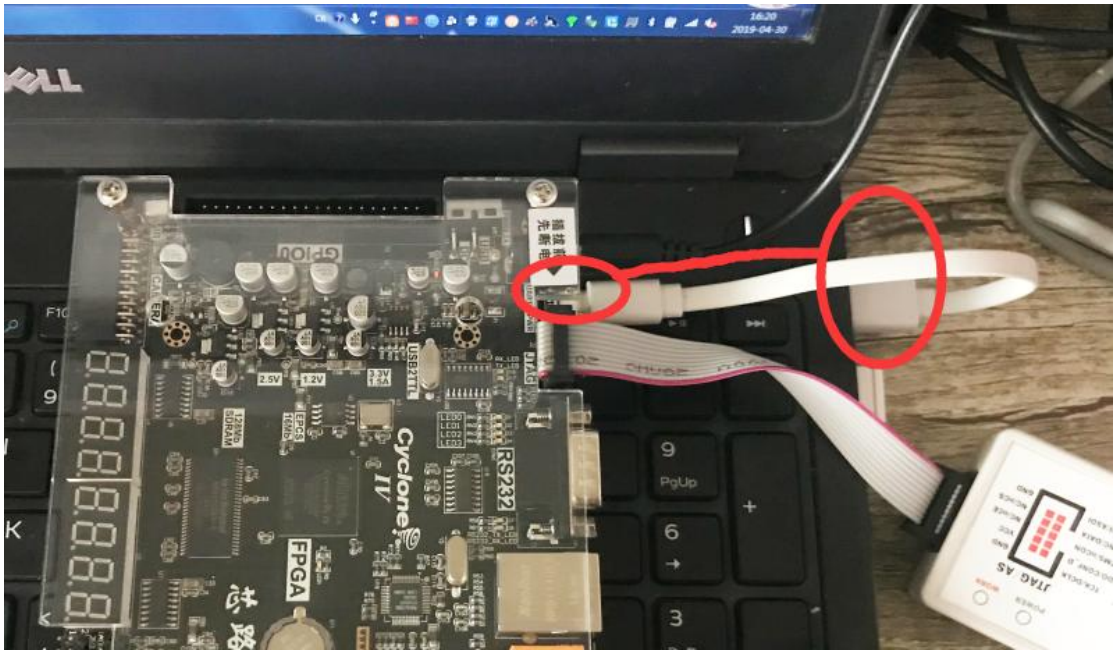

图 6.4 -23　AC620 开发板串口连接

3、打开串口猎人调试工具，设置正确的端口号，选择波特率为 9600，如所示。

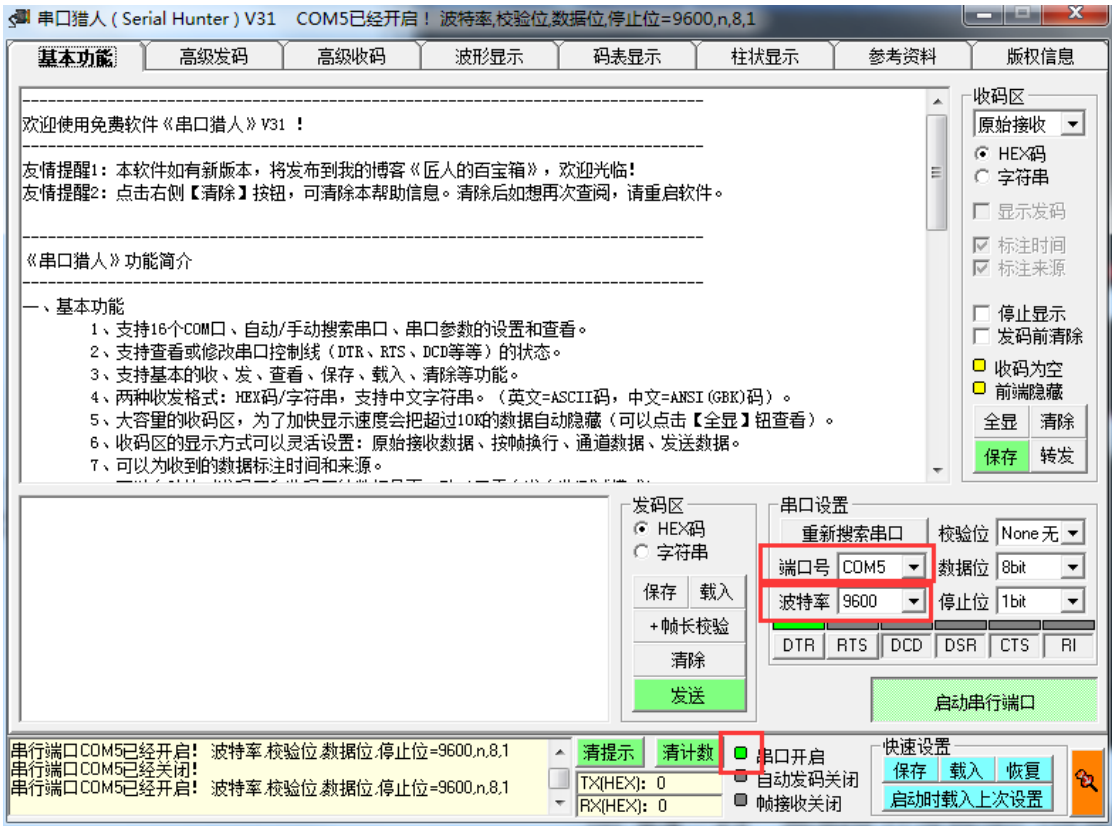

图 6.4 -24　串口猎人参数设置

4、设置通道 0 输出 100Hz，相位为 90 度，信号幅度为满幅；通道 1 输出 100Hz，相位为 180 度，信号幅度为满幅的一半，可以按照如下命令依次发送：

AA 03 10 8E 21 00 00 88 (设置通道 0 的频率为 100Hz)

AA 03 12 00 04 00 00 88 (设置通道 0 的相位为 90 度)

AA 03 14 00 00 00 00 88 (设置通道 0 的幅度为满幅)

AA 03 11 8E 21 00 00 88 (设置通道 1 的频率为 100Hz)

AA 03 13 00 08 00 00 88 (设置通道 1 的相位为 180 度)

AA 03 15 01 00 00 00 88 (设置通道 1 的幅度为半幅)

AA 03 06 03 00 00 00 88 (打开 2 个通道的输出)

由于需要发送的数据内容较多，为了方便操作，可以借助“串口猎人”软件的高级发码操作实现。在“高级发码”选项卡中，依次将上述命令输入到每一组中，

然后设置循环次数为 1，然后点击启动自动发码，串口猎人软件就能够将所有数据帧依次发送到板卡了，如图 6.4 -25 所示。

图 6.4 -25　使用串口猎人发送命令帧

5、使用示波器的两个探头分别连接开发板上的 DA 和 DB 输出连接针，就能够在示波器上看到 DAC 输出的波形了，如图 6.4 -26 和图 6.4 -27 所示。

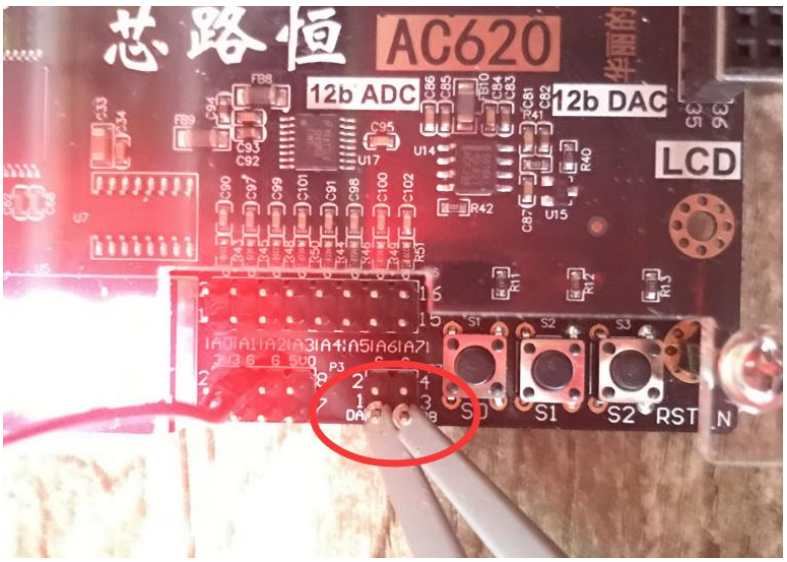

图 6.4 -26　使用示波器连接 AC620 开发板 DAC 输出

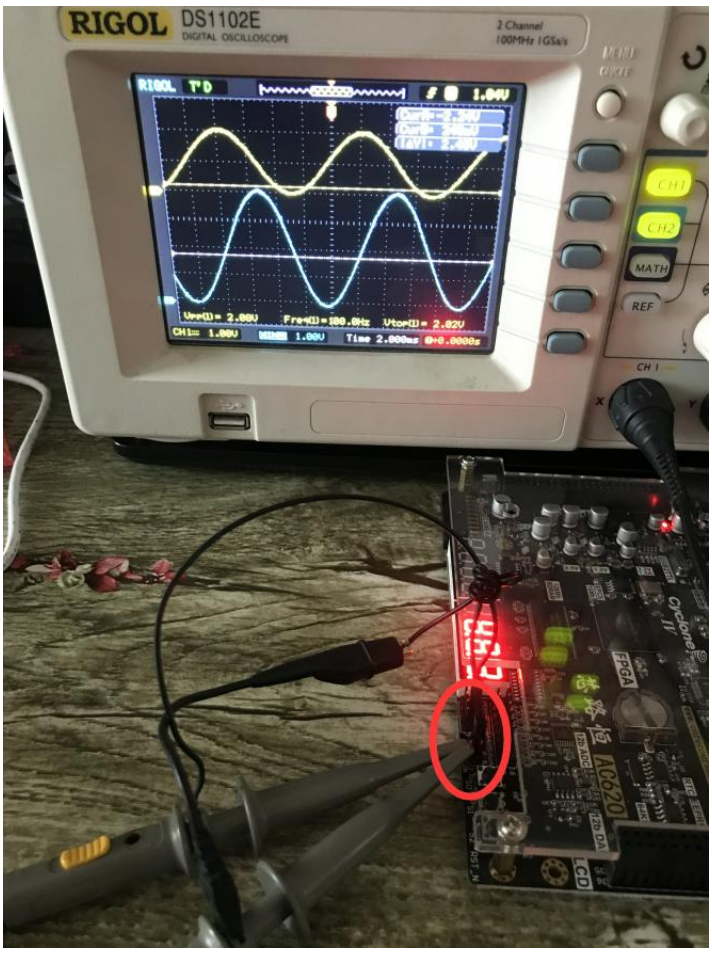

图 6.4 -27　示波器测量 DAC 输出波形

6、为了方便使用，本实例另外配套了一个上位机软件来代替串口猎人发送命令，该软件名称为“AC620_DDS2_控制台.exe”，该软件界面如图 6.4 -28 所示。

图 6.4 -28　DDS2 控制台软件

在该软件中，用户可以直接输入期望输出的信号频率、相位和幅度，然后更新指定通道的参数，或者同步更新两个通道的参数。其中，频率一栏输入每个通道期望输出的指定信号频率值，相位一栏输入每个通道输出信号的初始相位，值为 0~359，对应了 0°到 359°，而幅度一栏，还是和本手册中讲解的一致，为 0 就是满幅输出，为 1 就是半满幅输出。

图 6.4 -29 为网友使用该软件，基于本设计实例设置两个通道输出频率不相同的实例。

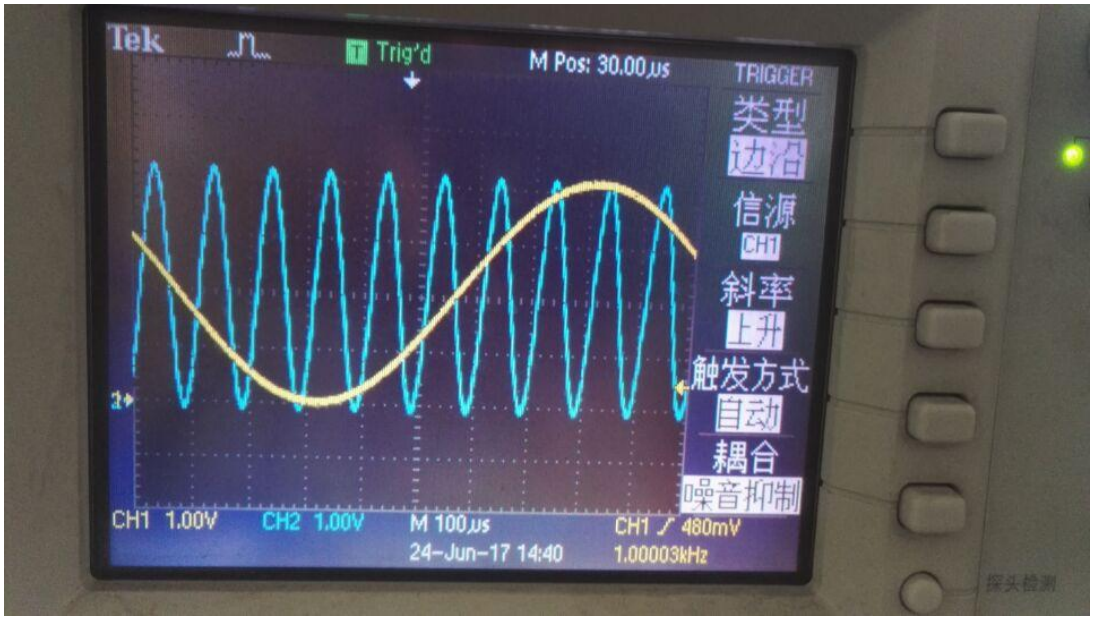

图 6.4 -29　双通道正弦波波形显示