21.DS18B20 温度传感器使用实验例程

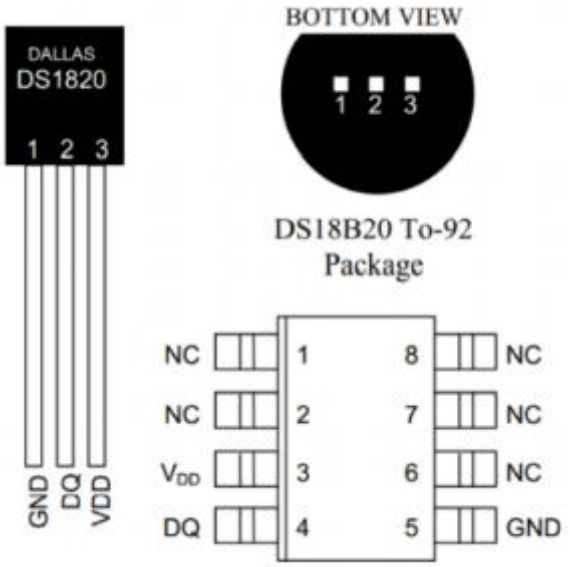
21.1. 实验目的

使用盘古 PGX-Nano 开发板驱动 DS18B20 模块读取温度并通过串口打印输出

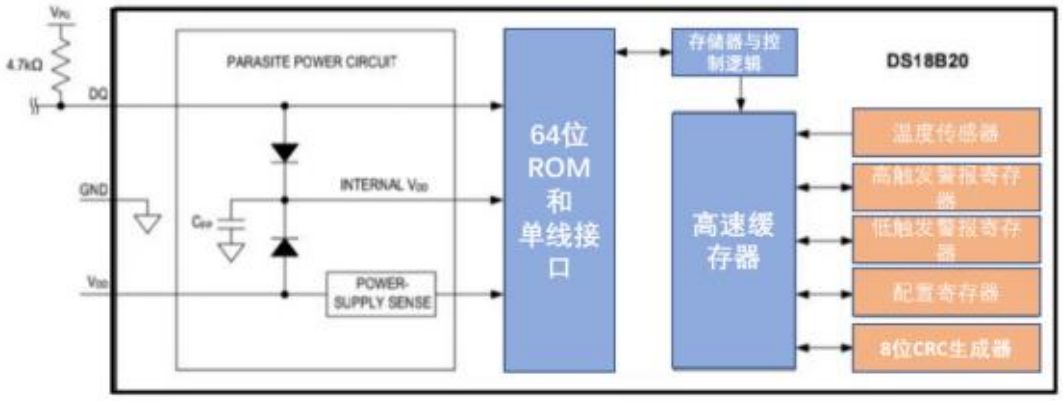
21.2. 实验简介

21.2.1. DS18B20 模块介绍

DS18B20 是一种数字温度传感器，支持单线通信协议，这意味着它只需要一根数据线与 FPGA 通讯仅需占用一个 I/O 端口，无须任何外部元件，直接将环境温度转化成数字信号，以数字码方式串行输出，从而大大简化了传感器与 FPGA 的接口设计。极大地简化了温度测量系统的连接复杂性。DS18B20 具有唯一的 64 位 ROM 地址，允许在单根数据线上连接多个传感器，实现多点温度测量。此外，它还支持寄生电源模式，可以在数据线供电的情况下工作，进一步简化了系统设计。由于其可靠的性能、低功耗、高精度以及易于集成的特点，DS18B20 被广泛应用于各种温度监控和数据采集系统中，如家庭自动化、工业控制和环境监测等。
模块引脚定义如下：
其中用户只需要对 **D0 口进行操作即可**



DS18B20 测温范围从-55° C 到+125° C，在-10° C 到+85° C 的范围内，精度可达±0.5° C。现场（实时）温度直接以“单总线” 的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~12 位的数字值读数方式。其芯片内部框图如下：



其中高速缓存器结构示意图如下：

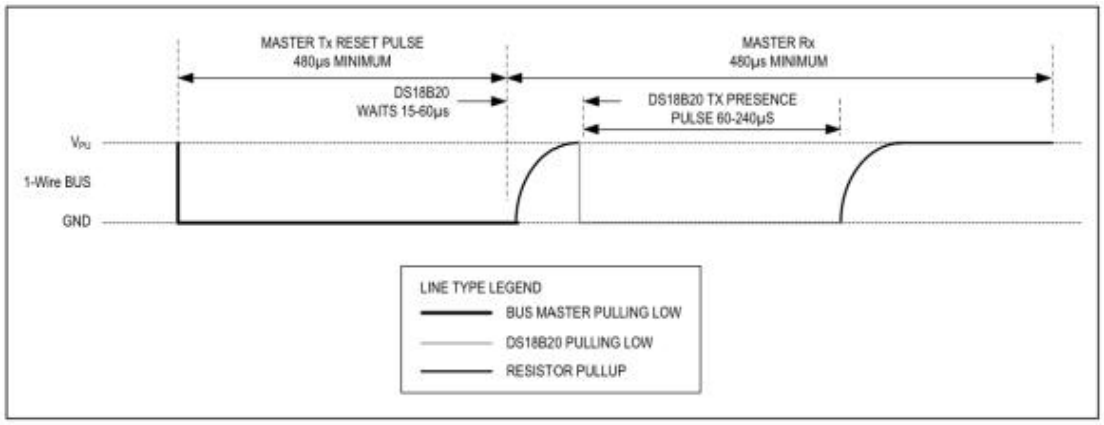
DS18B20 的高速缓冲器是一个临时存储温度测量数据和配置参数的小型存储器，用于在温度转换和数据通信过程中缓存必要的信息。由上图可知缓冲器由 9 个字节组成，其中包括测量的温度数据（前两个字节）、配置寄存器（第四个字节）、以及内部保留位等。用户可通过配置寄存器设置分辨率，而其他字节中的特定位被保留用于内部用途，不可写入。在配置寄存器的 8 个比特位中，只有第 5 位（R0）与第 6 位（R1）可以由用户配置，其他位都保留给 DS18B20 内部使用。R0 与 R1 的组合决定了传感器的分辨率，以及温度转换所需的时间。它们的作用可以理解为配置选项的二进制编码开关，通过不同的值组合实现分辨率的选择。详情请看下面的图片：

R1	R0	分辨率（位）	温度精度（℃）	典型转换时间（ms）
0	0	9 位	±0.5	93.75
0	1	10 位	±0.25	187.5
1	0	11 位	±0.125	375
1	1	12 位（默认）	±0.0625	750

21.2.2. DS18B20 通讯时序

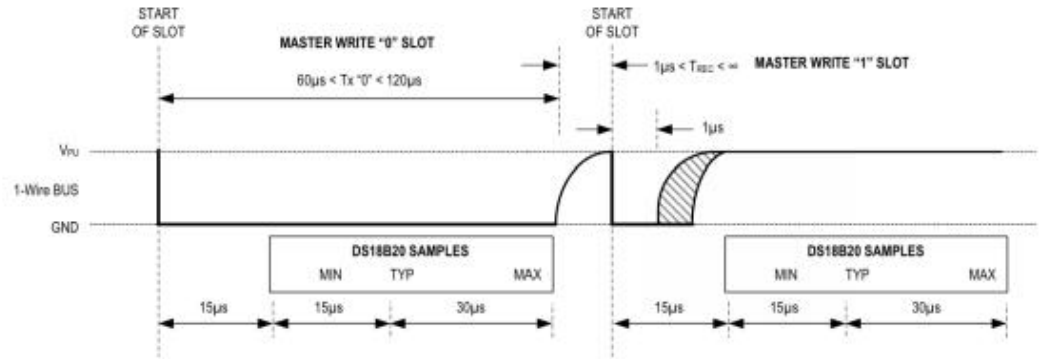
（1）DS18B20 总线复位时序

控制器与 DS18B20 所有的通信都是由初始化开始的，初始化由主设备发出的复位脉冲及 DS18B20 响应的存在脉冲组成。当 DS18B20 响应复位信号的后，向主设备表明其在该总线上，并且已经做好了执行命令的准备。在这个过程中，总线上的主设备需要拉低总线最少 480us 来表示发送复位脉冲。发送完之后，主设备要释放总线进入接收模式。当总线释放后，上拉电阻将总线拉至高电平。当 DS18B20 检测到该上升沿信号后，其等待 15us 至 60us 后将总线拉低 60us 至 240us 来实现发送回复。其具体时序图如下：



（2）DS18B20 总线写时序

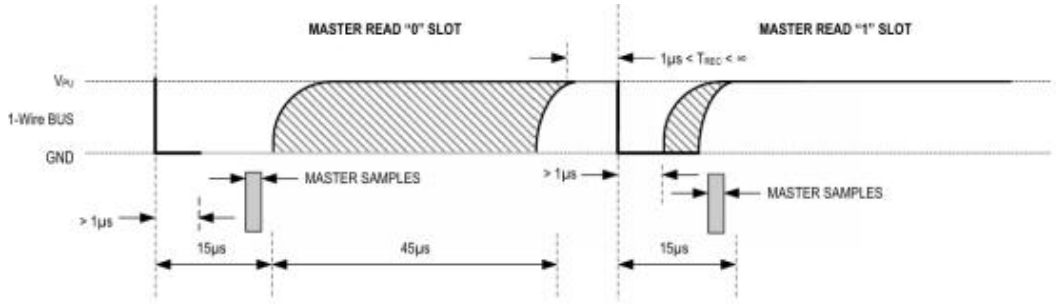
在写过程中写入数据有两种情况：写“1”和写“0”。写两种数据需要遵循不同的时序。当主设备将总线从高电平拉至低电平时，启动写操作，所有的写操作持续时间最少为 60us，每个写操作间的恢复时间最少为 1us。当总线（DQ）拉低后，DS18B20 在 15us 至 60us 之间对总线进行采样，如果采的 DQ 为高电平则发生写 1，如果为低电平则发生写 0，如下图所示（图中的总线控制器即为主设备）。如果主机要写 1，必须先将总线拉至逻辑低电平然后释放总线，允许总线在写操作开始后 15us 内上拉至高电平。若要写 0，必须将总线拉至逻辑低电平并保持不变最少 60us。



（3）DS18B20 总线读时序

同样的在读过程中也有有两种情况：读出“1”和读出“0”。每个读时序最小有 60us 的持续时间以及每个读时序之间有 1us 的恢复时间。当主设备将总线从高电平拉至低电平超过 1us，启动读操作。当启动读操作后，DS18B20 将会向主设备发送“0”或者“1”。DS18B20 通过将总线拉高来发送 1，将总线拉低

来发送 0。当读时隙完成后，DQ 引脚将通过上拉电阻将总线拉高至高电平的闲置状态。从 DS18B20 中输出的数据在启动读时隙后的 15us 内有效，所以，主设备在读时隙开始后的 15us 内必须释放总线，并且对总线进行采样。其具体时序图如下：



21.2.3.DS18B20 通讯过程

操作 DS18B20 需要三个步骤，分别是初始化、写 ROM 命令、写功能命令。其具体解释如下：

1.初始化

总线上的所有事件都必须以初始化为开始。初始化信号由总线上的主设备发出的复位脉冲以及紧跟着从设备回应的存在脉冲构成。该存在脉冲是让总线主设备知道 DS18B20 在总线上并准备好运行。具体的时序在 19.2.2 中 DS18B20 总线复位时序小节有具体介绍。

2.写 ROM 命令

在初始化完成后，可以执行 ROM 命令，这些命令用于操作每个设备的 64 位 ROM 编码，帮助主设备在总线上识别和管理多个从设备。ROM 命令共有 5 种，每种命令长度均为 8 位，具体如下：

(1)搜索 ROM [F0h]

在系统上电初始化后，主设备使用该命令识别总线上的所有从设备及其 ROM 编码，从而确定从设备的类型和数量。

(2)读 ROM [33h]

该命令允许主设备读取 DS18B20 的 64 位 ROM 编码，仅在总线上只有一个 DS18B20 时可用。若总线上存在多个从设备，使用此命令将导致所有设备同时响应，从而引起数据冲突。

(3)匹配 ROM [55h]

此命令后接 64 位 ROM 编码，用于在多点总线中定位特定的 DS18B20。只有编码与指定 ROM 完全匹配的设备会响应，其他设备将等待下一个复位脉冲。该命令可在单点或多点总线上使用。

(4)跳过 ROM [CCh]

该命令允许主设备跳过 64 位 ROM 编码直接执行下一步操作，适用于单点总线（只有一个 DS18B20）的情况（本次实验就是此种情况），可节省时间。但在

多点总线中，若发送跳过 ROM 命令后执行读操作，则所有从设备将同时响应，导致数据冲突。

(5)警报搜索 [ECh]

此命令功能类似跳过 ROM，但只有温度超出报警阈值（高于 TH 或低于 TL）的从设备会响应。报警状态会持续保留，直到温度回到正常范围或掉电为止。

3.写功能命令

当主设备通过 ROM 命令确认某个 DS18B20 可以通信后,便可向目标从设备发送功能命令，以执行特定操作。以下是 DS18B20 的功能命令及其作用：

(1)温度转换 [44h]

该命令用于启动单次温度转换，完成后，转换结果会存储在高速缓存器的 byte0（温度低 8 位）和 byte1（温度高 8 位）中，随后 DS18B20 进入低功耗闲置状态。若总线在命令后发出读时隙，DS18B20 会返回：

″0″：表示温度转换尚未完成。

″1″：表示温度转换已完成。

寄生电源模式注意事项：在发送该命令后，必须立即强制拉高总线，且拉高时间需满足时序要求。

(2)写入暂存器 [4Eh]

此命令允许主设备向高速缓存器写入 3 个字节数据，按顺序写入以下寄存器：byte2（高温触发值）：报警触发高温值。byte3（低温触发值）：报警触发低温值。byte4（配置寄存器）：分辨率配置等参数。数据写入按低位到高位顺序进行，可通过复位随时中断写入操作。

(3)读取高速缓存器 [BEh]

该命令从高速缓存器读取数据，读取从 byte0（温度低 8 位）开始，到 byte8（CRC 校验）结束。数据从低位开始传送，读取过程可通过复位随时终止。

(4)复制高速缓存器 [48h]

将高速缓存器中的高温触发值（byte2）、低温触发值（byte3）和配置寄存器（byte4）的值复制到非易失性存储器（EEPROM）中，若命令后主机发出读请求，DS18B20 会返回：

″0″：表示复制操作正在进行。

″1″：表示复制完成。

需要注意的是如果使用寄生电源模式,发送该命令后，必须立即强制拉高总线至少 10ms。

(5)召回 EEPROM [B8h]

将 EEPROM 中存储的高温触发值（byte2）、低温触发值（byte3）和配置寄存器（byte4）的数据恢复到高速缓存器中。上电后，召回操作会自动执行一次，确保缓存器中有有效数据。若执行该命令后主机发出读请求，DS18B20 会返回：

″0″：表示正在召回数据。

″1″：表示召回完成。

(6)读取供电模式［B4h］

该命令用于判断 DS18B20 的供电方式：返回 ″0″：表示使用寄生电源模式。返回 ″1″：表示使用外部电源模式。

21.3. 程序设计

根据以上对DS18B20的介绍，我们使用逻辑派Z1开发板对DS18B20进行初始化，并写入将模块测量的环境温度解析出来，使用串口发送到上位机。

DS18B20 驱动模块端口描述如下表：

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
cap_flag	input	1	触发采集信号（开始温度测量）
distance	output	8	计算得到的距离（单位厘米）
data_vld	output	1	distance 的有效标志
dq	inout	1	DS18b20 通讯总线
s_bf	output	4	温度数据的百分位
s_sf	output	4	温度数据的十分位
s_sw	output	4	温度数据的十位
s_gw	output	4	温度数据的个位
data_out	output	20	温度数据
data_symbol	output	1	温度正负指示

驱动部分代码如下，完整源码请查看 demo 源文件

```
always@(posedge clk_us or negedge rst_n)
   if(rst_n == 1'b0)
      state <= WAIT_TRI;
   else
      case(state)
            WAIT_TRI    : //等待触发采样
               if(cap_start)//检测到触发信号,跳转到 INIT
```

```
                state <= INIT;
            else
                state <= WAIT_TRI;
        INIT       ://初始化
            if(us_cnt == 20'd959 && flag == 1'b1)//初始化完成,跳转到 WR_CMD
                state <= WR_CMD;
            else
                state <= INIT;
        WR_CMD     ://写温度转换命令
            if(bit_cnt == 4'd15 && us_cnt == 20'd64)//写完 16 位数据，跳
转到 WAIT
                state <= WAIT;
            else
                state <= WR_CMD;
        WAIT       ://等待转换完成
            if(us_cnt == WAIT_MAX)//等待完转换时间，跳转到 RD_CMD
                state <= INIT_AGAIN;
            else
                state <= WAIT;
        INIT_AGAIN ://初始化
            if(us_cnt == 20'd959 && flag == 1'b1)//初始化完成,跳转到 RD_CMD
                state <= RD_CMD;
            else
                state <= INIT_AGAIN;
        RD_CMD     ://写温度读取命令
            if(bit_cnt == 4'd15 && us_cnt == 20'd64)//写完 16 位数据，跳
转到 RD_TEMP
                state <= RD_TEMP;
            else
                state <= RD_CMD;
        RD_TEMP    ://读温度数据
            if(bit_cnt == 4'd15 && us_cnt == 20'd64)//接收完 16 位数据，
跳转到 WAIT_TRI
                state <= WAIT_TRI;
            else
                state <= RD_TEMP;
        default:state <= WAIT_TRI;
    endcase
always@(posedge clk_us or negedge rst_n)
  if(rst_n == 1'b0)
    begin
    dq_en  <= 1'b0;
    dq_out <= 1'b0;
    end
```

```
else
   case(state)
         WAIT_TRI       :
            begin
               dq_en  <= 1'b1;
               dq_out <= 1'b1;
            end
         INIT       :
            if(us_cnt < 20'd499)
               begin
                     dq_en  <= 1'b1;
                     dq_out <= 1'b0;
               end
            else
               begin
                     dq_en  <= 1'b0;
                     dq_out <= 1'b0;
               end
         WR_CMD     :
            if(us_cnt > 20'd62)
               begin
                     dq_en  <= 1'b0;
                     dq_out <= 1'b0;
               end
            else    if(us_cnt <= 20'd1)
               begin
                     dq_en  <= 1'b1;
                     dq_out <= 1'b0;
               end
            else    if(WR_CC_44[bit_cnt] == 1'b0)
               begin
                     dq_en  <= 1'b1;
                     dq_out <= 1'b0;
               end
            else    if(WR_CC_44[bit_cnt] == 1'b1)
               begin
                     dq_en  <= 1'b0;
                     dq_out <= 1'b0;
               end
         WAIT       :
            begin
               dq_en  <= 1'b1;
               dq_out <= 1'b1;
            end
```

```
            INIT_AGAIN :
               if(us_cnt < 20'd499)
                  begin
                        dq_en  <= 1'b1;
                        dq_out <= 1'b0;
                  end
               else
                  begin
                        dq_en  <= 1'b0;
                        dq_out <= 1'b0;
                  end
            RD_CMD     :
               if(us_cnt > 20'd62)
                  begin
                        dq_en  <= 1'b0;
                        dq_out <= 1'b0;
                  end
               else    if(us_cnt <= 20'd1)
                  begin
                        dq_en  <= 1'b1;
                        dq_out <= 1'b0;
                  end
               else    if(WR_CC_BE[bit_cnt] == 1'b0)
                  begin
                        dq_en  <= 1'b1;
                        dq_out <= 1'b0;
                  end
               else    if(WR_CC_BE[bit_cnt] == 1'b1)
                  begin
                        dq_en  <= 1'b0;
                        dq_out <= 1'b0;
                  end
            RD_TEMP    :
               if(us_cnt <= 1)
                  begin
                        dq_en  <= 1'b1;
                        dq_out <= 1'b0;
                  end
               else
                  begin
                        dq_en  <= 1'b0;
                        dq_out <= 1'b0;
                  end
            default:
```

```
            begin
               dq_en  <= 1'b0;
               dq_out <= 1'b0;
            end
      endcase
```

因为我们使用的是默认配置，所以温度采样分辨率为 12 位，温度转换时间为 750ms。模块核心逻辑是一个三段式状态机，我们通过设计这个状态机来处理顶层触发信号与 DS18B20 的交互。状态机总共有七个状态：DS18B20 驱动模块的状态机共有 7 个状态，每个状态的功能如下：

WAIT_TRI：在此状态下，等待顶层模块的触发信号 cap_flag 启动信号的到来。如果接收到启动信号，状态将转移到 INIT。

INIT：初始化状态。在该状态中，按照总线时序发送发送初始化信号。如果初始化信号发送成功，且接收到 DS18B20 的应答，状态机将转移到 WR_CMD 状态。

WR_CMD：写命令状态。在此状态中，发送发送跳过 ROM 命令[CCh]和温度转换命令[44h]。发送完成后，状态机将转移到 WAIT 状态，等待 DS18B20 采集温度数据完成。

WAIT：等待状态。在此状态下，进行 750ms 的等待。DS18B20 温度转换完成，当等待完成后状态机将转移到 INIT_AGAIN 状态。

INIT_AGAIN：重新初始化状态。发送初始化命令，如果初始化信号发送成功，且接收到 DS18B20 的应答，状态机将转移到 RD_CMD 状态。

RD_CMD：读命令状态。在此状态下，向 DS18B20 发送跳过 ROM 命令[CCh]和温度读取命令[BEh]。读取命令发送完成后，状态机将转移到 RD_TEMP 状态。

RD_TEMP：读取温度状态。在此状态中，处理 DS18B20 返回的温度数据。处理完成后，状态机将重新回到等待启动状态(WAIT_TRI)。

分析这个模块需要清楚的理解 DS18B20 的操作过程，若读者对 DS18B20 的操作时序不太清楚，请务必返回理论部分进行学习。

串口发送代码如下：

```
// 串口发送数据控制
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      uart_tx_req <= 1'b0;
      uart_tx_data <= 8'b0;
      ascii_table[0] = 8'd48;                                    //'0'
      ascii_table[1] = 8'd49;                                    //'1'
      ascii_table[2] = 8'd50;                                    //'2'
      ascii_table[3] = 8'd51;                                    //'3'
```

```
    ascii_table[4] = 8'd52;                                    //'4'
    ascii_table[5] = 8'd53;                                    //'5'
    ascii_table[6] = 8'd54;                                    //'6'
    ascii_table[7] = 8'd55;                                    //'7'
    ascii_table[8] = 8'd56;                                    //'8'
    ascii_table[9] = 8'd57;                                    //'9'
    ascii_table[10] = 8'd43;                                    //'+'
    ascii_table[11] = 8'd45;                                    //'-'
  end else if (work_en && !uart_tx_busy) begin
    case (tx_byte_cnt)                                         // 根据当前字节计数
器发送数据
          7'd1:  uart_tx_data <= STR[127:120];                  // "当"
          7'd2:  uart_tx_data <= STR[119:112];                  // "当"
          7'd3: uart_tx_data <= STR[111:104];                  // "前"
          7'd4: uart_tx_data <= STR[103:96];                   // "前"
          7'd5: uart_tx_data <= STR[95:88];                    // "环"
          7'd6: uart_tx_data <= STR[87:80];                    // "环"
          7'd7: uart_tx_data <= STR[79:72];                    // "境"
          7'd8: uart_tx_data <= STR[71:64];                    // "境"
          7'd9: uart_tx_data <= STR[63:56];                    // "温"
          7'd10: uart_tx_data <= STR[55:48];                    // "温"
          7'd11: uart_tx_data <= STR[47:40];                    // "度"
          7'd12: uart_tx_data <= STR[39:32];                    // "度"
          7'd13: uart_tx_data <= STR[31:24];                    // "为"
          7'd14: uart_tx_data <= STR[23:16];                    // "为"
          7'd15: uart_tx_data <= STR[15:8];                     // ": "
          7'd16: uart_tx_data <= STR[7:0];                      // ": "
          7'd17: uart_tx_data <= ascii_table[data_symbol];       // 温度符号
          7'd18: uart_tx_data <= ascii_table[s_sw];              // 温度十位
          7'd19: uart_tx_data <= ascii_table[s_gw];              // 温度个位
          7'd20: uart_tx_data <= 8'd46;                          //小数点
          7'd21: uart_tx_data <= ascii_table[s_sf];              // 温度十分位
          7'd22: uart_tx_data <= ascii_table[s_bf];              // 温度百分位
          7'd23: uart_tx_data <= 8'h0A;                         // 换行符
          default: uart_tx_data <= 8'b0;
    endcase
    uart_tx_req <= 1'b1;                                       // 拉高请求信号
  end else begin
    uart_tx_req <= 1'b0;                                       // 请求信号拉高仅一
个时钟周期
  end
end
```

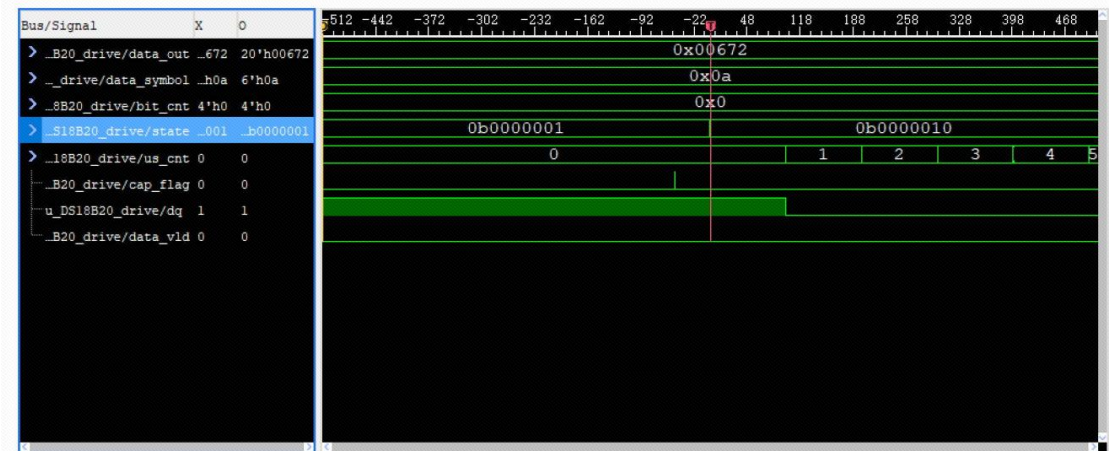
模块中初始化了一个 ascii_table 用来储存阿拉伯数字 0~9 和符号“+ -”的 ASCII 值，将 DS18B20 采样得到的温湿度数据在 ascii_table 索引到对应的 ASCII 值进行发送。

21.4. 代码仿真

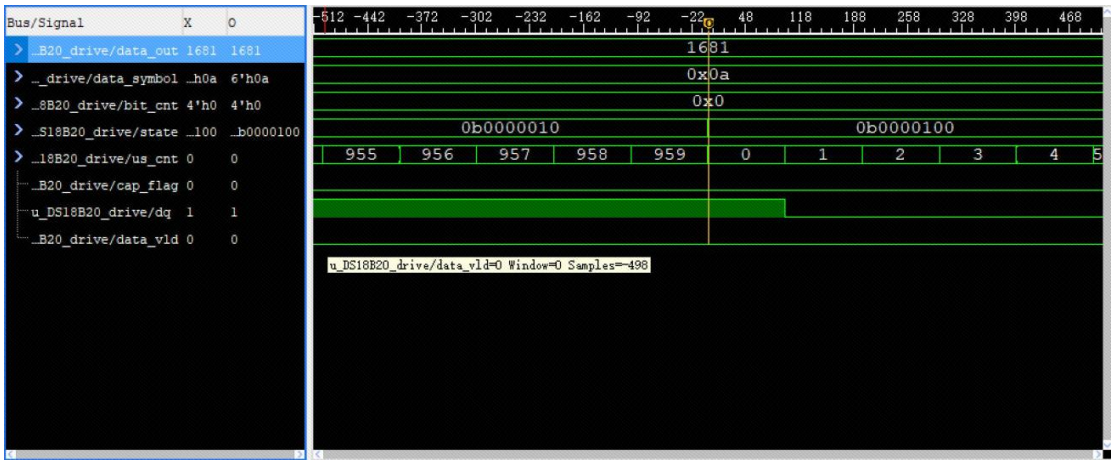
由于本实验需要与 DS18B20 通过总线交互信息，在测试文件中编写 DS18B20 的应答逻辑比较麻烦，我们可以将 DS18B20 驱动模块中的关键信号打上 debug 标记，分析 DS18B20 驱动模块是否正确工作，同时检查 DS18B20 是否正确应答我们通过总线发送的信息。

我们将顶层模块发送拉低采样触发信号 cap_flag、总线信号 dq、温度符号指示 data_symbol、温度有效标志 data_vld、温度数据 data_out、状态机信号 state、微秒计数器 us_cnt、接收数据比特计数器 bit_cnt 添加上 debug 标记进行分析。

首先我们设置触发模式为 cap_flag = 1，按下按键捕获到波形，观察状态机是否正确响应触发信号。波形如下图所示，状态机正确跳转到初始化状态，对 DS18B20 进行初始化。

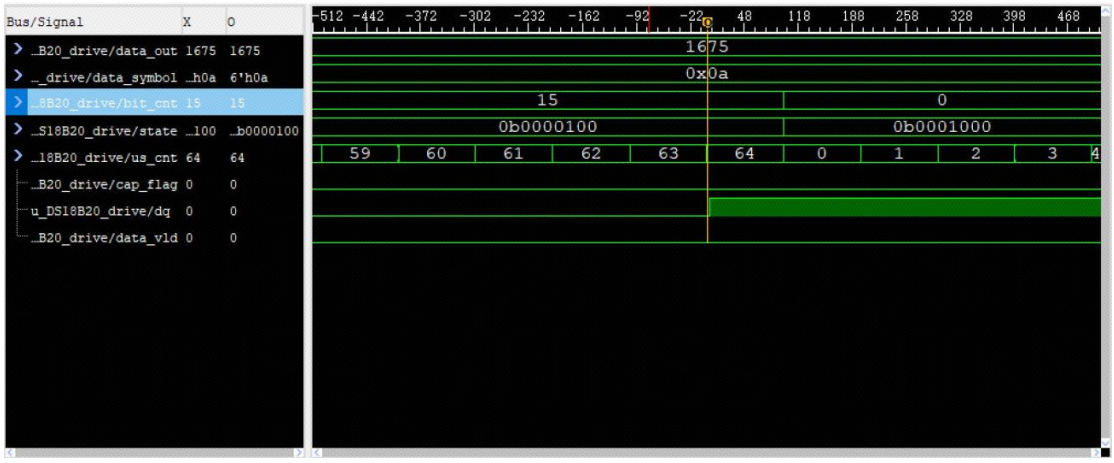


然后设置触发模式为（state = 7’b0000_010&&us_cnt = 959）（状态机处于 DS18B20 初始化状态，接收响应完成），DS18B20 总线复位时序中我们有说明，一整个复位操作包括主机发送复位与从机应答总共 960us，我们以此作为条件观察复位是否成功，按下按键观察捕获到的波形如下：

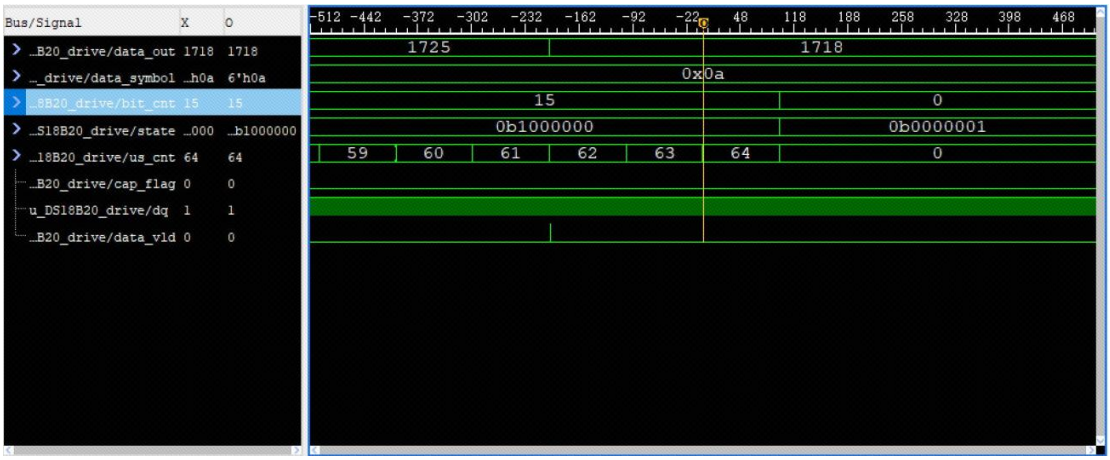


由上图可知当 us_cnt = 959 时，状态机跳转到下一状态，同时总线拉低，符合我们的复位时序。

接下来将触发模式设置为（state = 7’b0000_100&&bit_cnt = 15&&us_cnt =64）（状态机发送 16bit 命令状态，并且发送完成），捕获到的波形图如下图：



由上图可知在发送完成 16bit 命令[CCh][44h]之后状态机跳转到等待状态，等待 DS18B20 进行温度转换，共等待 750ms，此时总线拉高。由此可见状态跳转逻辑正确。接下来就是再次进行初始化，发送命令[CCh][BEh]，状态机跳转到接收温度数据状态，开始读数据。我们将触发模式设置为（state = 7’b1000_000&&bit_cnt = 15&&us_cnt =64）（状态机接收温度数据，并且接收完成），捕获到的波形图如下：



在这张图中，状态机接收数据完成，并计算出当前温度为 17.18 度，同时 data_vld 信号拉高，表示当前温度数据解析完成，数据有效。之后状态机跳转到等待触发状态，等待顶层触发信号再次到来。由此可见，模块功能正确。

21.5. 实验现象