

6. 串口收发实验例程

6.1 PGX-Nano 开发板简介

PGX-Nano 开发板集成了一路 USB 转串口模块, 采用的 USB-UART 芯片 CP2102, USB 接口采用 USB Type C 接口, 可以用一根 USB Type C 线连接到 PC 的 USB 口进行串口数据通信 (详情请查看“PGX-Nano 开发板硬件使用手册”)。

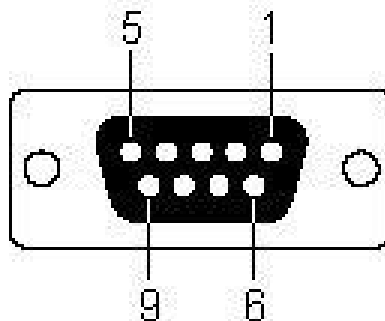
6.2 实验要求

串口通信时波特率设置为 115200bps, 数据格式为 1 位起始位、8 位数据位、无校验位、1 位结束位。板子 每隔 1s 向串口助手发送一次 ASCII 码的“www.meyesemi.com”, 通过串口助手向板子以十六进制形式发送数字 (00~FF), 开发板 LED 以二进制显示亮起。

6.3 实验原理

6.3.1 串口原理

从下图我们可以看到标准串口接口是 9 根线, 具体含义如下:



数据线:

TXD (pin 3): 串口数据输出 (Transmit Data)

RXD (pin 2): 串口数据输入 (Receive Data)

握手:

RTS (pin 7): 发送数据请求 (Request to Send)

CTS (pin 8): 清除发送 (Clear to Send)

DSR (pin 6): 数据发送就绪 (Data Send Ready)

DCD (pin 1): 数据载波检测 (Data Carrier Detect)

DTR (pin 4): 数据终端就绪(Data Terminal Ready)

地线:

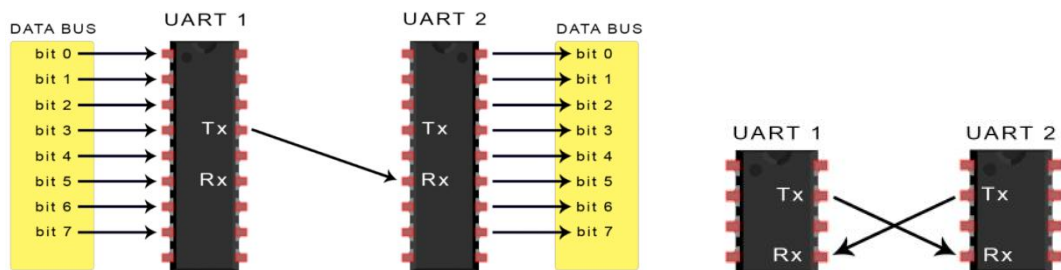
GND (pin 5): 地线

其它:

RI (pin 9): 铃声指示

通常我们用 RS232 串口仅用到了 9 根传输线中的三根: TXD, RXD, GND。但是对于数据传输, 双方必须对数据传输采用使用相同的波特率, 约定同样的传输模式(传输架构, 握手条件等)。尽管这种方法对于大多数应用已经足够, 但是对于接收方过载的情况这种使用受到限制。

RS232 的串口连接方式:



串口传输协议如下:

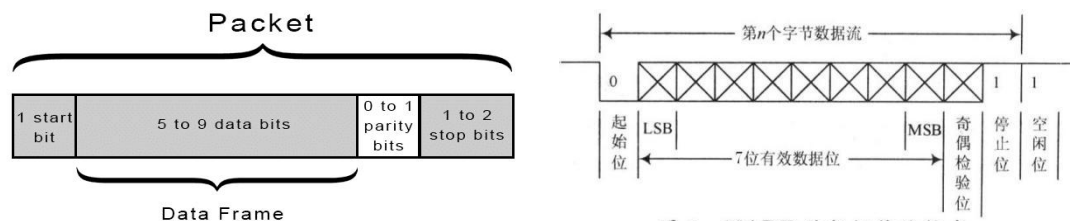


图 1 UART 的数据传输格式

起始位: 先发出一个逻辑”0”信号, 表示传输字符的开始。

数据位: 可以是 5~8 位逻辑”0”或”1”。如 ASCII 码(7 位), 扩展 BCD 码(8 位)。

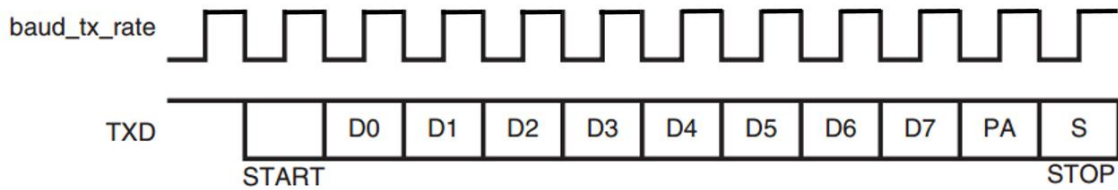
校验位: 数据位加上这一位后, 使得”1”的位数应为偶数(偶校验)或奇数(奇校验)。

停止位: 它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。

空闲位: 处于逻辑”1”状态, 表示当前线路上没有资料传送。

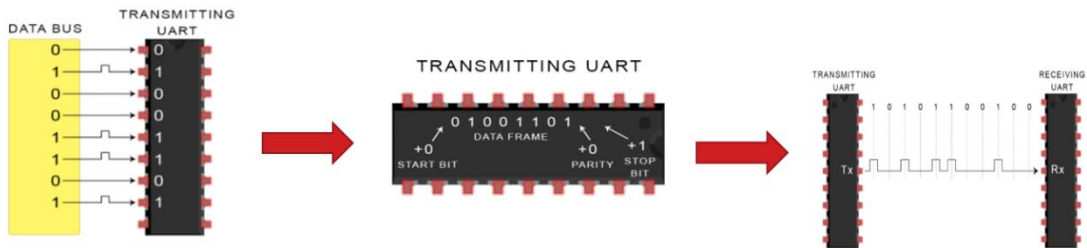
波特率: uart 中的波特率就可以认为是比特率, 即每秒传输的位数(bit)。一般选波特率都会有 9600, 19200, 115200 等选项。其实意思就是每秒传输这么多个比特位数(bit)。

引入波特率的概念后可得到串口的传输节奏如下:

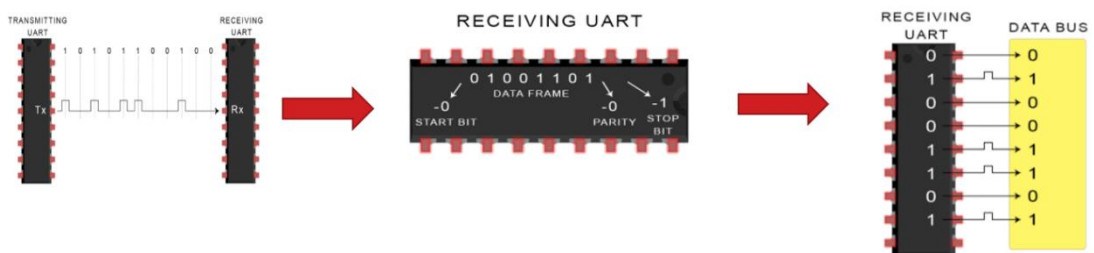


6.3.2 串口传输步骤

6.3.2.1 串口发送流程



6.3.2.2 串口接收流程



6.3.3 串口发送字符

从前面串口协议中可以了解到串口每次传输可以有 5~8bit 数据, 在计算机中字符通常用 ASCII 码 (7bit) 表示, 所以字符的发送可以用 ASCII 码发送。

查询 ASCII 码表格可得到: “www.meyesemi.com” 用到的字符对应 ASCII 码;

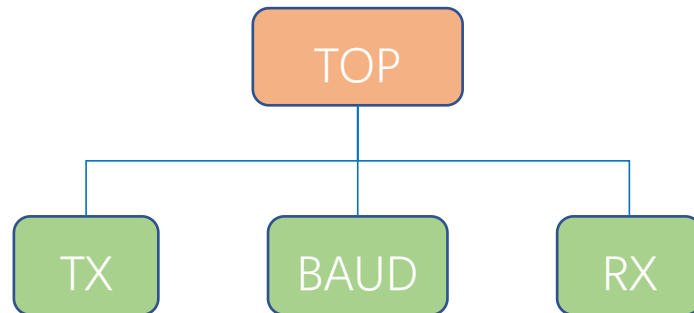
```

8'h1 : write_data <= `UD 8'h77; // ASCII code is w
8'h2 : write_data <= `UD 8'h77; // ASCII code is w
8'h3 : write_data <= `UD 8'h77; // ASCII code is w
8'h4 : write_data <= `UD 8'h2E; // ASCII code is .
8'h5 : write_data <= `UD 8'h6D; // ASCII code is m
8'h6 : write_data <= `UD 8'h65; // ASCII code is e
8'h7 : write_data <= `UD 8'h79; // ASCII code is y
8'h8 : write_data <= `UD 8'h65; // ASCII code is e
8'h9 : write_data <= `UD 8'h73; // ASCII code is s
8'ha : write_data <= `UD 8'h65; // ASCII code is e
8'hb : write_data <= `UD 8'h6D; // ASCII code is m
8'hc : write_data <= `UD 8'h69; // ASCII code is i
8'hd : write_data <= `UD 8'h2E; // ASCII code is .
8'he : write_data <= `UD 8'h63; // ASCII code is c
8'hf : write_data <= `UD 8'h6F; // ASCII code is o
8'h10: write_data <= `UD 8'h6D; // ASCII code is m

```

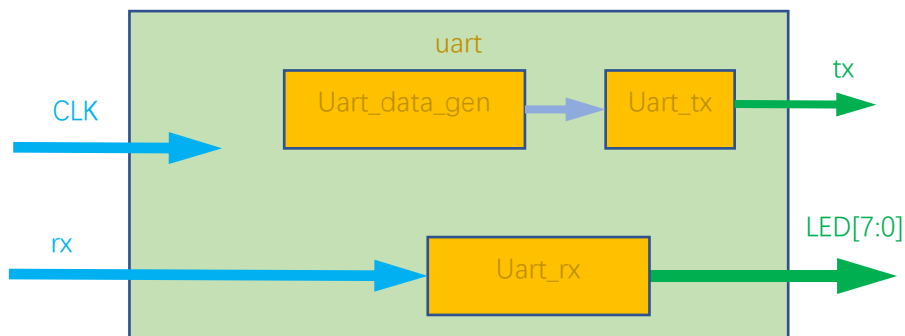
6.4 实验源码设计

从实验目的分析可将实验做如下划分：



从原理上分析波特率的计算是一个计数器，发射和接收可复用，我们在设计时为保持 TX，或 RX 的完整性，故将波特周期计数器集成在各自模块内部；

上述分析仅仅搭建好 PGX-Nano 的与 PC 通信的桥梁 UART，传输的数据没有体现。故而需要增加发送数据模块，与接收数据模块；



6.4.1 串口发射模块设计

目标：接收到一个发送命令信号时，将 data[7:0] -> 依次发出 {start, data[0:7], stop} 共 10bit 数据（无校验位，停止位 1bit）；

有两种方法可以将一个并行数据串行化；

方法一：通过 bit 计数与 baud 计数控制移位输出；

```

1  // transmit bit
2  always@(posedge clk)
3  begin
4      if(!rstn)
5          txd <= `UD 1'b1;
6      else
7          begin
8              if(trans_en)
9                  Begin
10 // 将开始标志和停止标志以及传输数据集成放到 trans_data 中可用下方语句
11 //          txd <= `UD trans_data[trans_bit];
12 // 单 bit 控制用下方语句
13             case(trans_bit)
14                 4'h0 :txd <= `UD 1'b0;
15                 4'h1 :txd <= `UD tx_data_reg[0];
16                 4'h2 :txd <= `UD tx_data_reg[1];
17                 4'h3 :txd <= `UD tx_data_reg[2];
18                 4'h4 :txd <= `UD tx_data_reg[3];
19                 4'h5 :txd <= `UD tx_data_reg[4];
20                 4'h6 :txd <= `UD tx_data_reg[5];
21                 4'h7 :txd <= `UD tx_data_reg[6];
22                 4'h8 :txd <= `UD tx_data_reg[7];
23                 4'h9 :txd <= `UD 1'b1;
24                 default :txd <= `UD 1'b1;
25             endcase
26         end
27     else
28         txd <= `UD 1'b1;
29     end
30 end
31

```

方法二：通过 bit 计数与 baud 计数控制状态跳转，在状态机中输出；

```

1  // logical ouput 状态机输出
2  always @ (posedge clk)
3  begin
4      if(tx_en)
5          begin
6              case(tx_state)
7                  IDLE      : uart_tx <= `UD 1'h1; //空闲状态输出高电平
8                  SEND_START : uart_tx <= `UD 1'h0; //start 状态发送一个波特周期的低电平
9                  SEND_DATA  : //发送状态每个波特周期发送一个 bit;
10                 begin
11                     case(tx_bit_cnt)
12                         3'h0 : uart_tx <= `UD trans_data[0];
13                         3'h1 : uart_tx <= `UD trans_data[1];
14                         3'h2 : uart_tx <= `UD trans_data[2];
15                         3'h3 : uart_tx <= `UD trans_data[3];
16                         3'h4 : uart_tx <= `UD trans_data[4];
17                         3'h5 : uart_tx <= `UD trans_data[5];
18                         3'h6 : uart_tx <= `UD trans_data[6];
19                         3'h7 : uart_tx <= `UD trans_data[7];
20                     default: uart_tx <= `UD 1'h1;
21                 endcase
22             end
23             SEND_STOP : uart_tx <= `UD 1'h1; //发送停止状态 输出 1 个波特周期高电平
24             default   : uart_tx <= `UD 1'h1; // 其他状态默认与空闲状态一致, 保持高电平输出
25         endcase
26     end
27 else
28     uart_tx <= `UD 1'h1;
29 end
30

```

方法一的 module 如下:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module uart_tx #(
4      parameter BAUND_RATE_CNT = 12'd1250
5      //115200 : 12MHz, 12000000/115200 = 10'd104
6      //9600   : 12000000/9600 = 11'd1250
7  )
8  (
9      input      clk,
10     input      rstn,
11     input      trig, // active posedge
12
13     input [7:0] tx_data,
14
15     output reg  txd,
16     output      tx_busy
17 );
18

```

```

19 //=====
20 // baud rate set
21 reg [11:0] baud_cnt;
22 always @(posedge clk)
23 begin
24     if(!rstn)
25         baud_cnt <= `UD 12'd0;
26     else
27         begin
28             if(baud_cnt == BAUND_RATE_CNT - 1'b1)
29                 baud_cnt <= `UD 12'd0;
30             else
31                 baud_cnt <= `UD baud_cnt + 12'd1;
32         end
33     end
34
35     wire baud_over = (baud_cnt == BAUND_RATE_CNT - 1'b1) ? 1'b1 : 1'b0;
36
37 //=====
38 //transmit start
39 reg trig_1d;
40 reg [7:0] tx_data_reg;
41 always @(posedge clk)
42 begin
43     trig_1d <= `UD trig;
44     end
45
46 reg start_en;
47 wire start;
48 always @(posedge clk)
49 begin
50     if(!rstn)
51         start_en <= `UD 1'b0;
52     else if(~trig_1d & trig & ~start_en)
53         start_en <= `UD 1'b1;
54     else if(baud_over)
55         start_en <= `UD 1'b0;
56     end
57 assign start = ~trig_1d & trig; //start_en & baud_over;
58
59 //将 data 在触发发送时进行锁存
60 always @(posedge clk)
61 begin
62     if(!rstn)
63         tx_data_reg <= `UD 8'h3f;
64     else if(~trig_1d & trig)
65         tx_data_reg <= `UD tx_data;
66     end
67

```

```

68 //=====
69 // trasmit data 将 start stop data 锁存在一个锁存器中。
70 reg [9:0] trans_data;
71
72 always @(posedge clk)
73 begin
74     if(!rstn)
75         trans_data <= `UD 10'h3f;
76     else if(~trig_1d & trig)
77         trans_data <= `UD {1'b1,tx_data,1'b0};
78 end
79
80 //=====
81 // transmit control
82 reg trans_en;
83 reg [3:0] trans_bit;
84 always @(posedge clk)
85 begin
86     if(!rstn)
87         trans_en <= `UD 1'b0;
88     else if(~trig_1d & trig)
89         trans_en <= `UD 1'b1;
90     else if(trans_bit == 4'd9 && baund_over)
91         trans_en <= `UD 1'b0;
92     else
93         trans_en <= `UD trans_en;
94 end
95
96 assign tx_busy = ~trans_en;
97
98 always @(posedge clk)
99 begin
100     if(!rstn)
101         trans_bit <= `UD 4'd0;
102     else
103     begin
104         if(trans_en && baund_over)
105         begin
106             if(trans_bit == 4'd9)
107                 trans_bit <= `UD 4'd0;
108             else
109                 trans_bit <= `UD trans_bit + 4'd1;
110         end
111     else if(!trans_en)
112         trans_bit <= `UD 4'd0;
113     end
114 end
115

```

```

116 //=====
117 // transmit bit
118 always@(posedge clk)
119 begin
120     if(!rstn)
121         txd <= `UD 1'b1;
122     else
123         begin
124             if(trans_en)
125                 begin
126 // 将开始标志和停止标志以及传输数据集成放到 trans_data 中可用下方语句
127 //             txd <= `UD trans_data[trans_bit];
128 // 单 bit 控制用下方语句
129                 case(trans_bit)
130                     4'h0 :txd <= `UD 1'b0;
131                     4'h1 :txd <= `UD tx_data_reg[0];
132                     4'h2 :txd <= `UD tx_data_reg[1];
133                     4'h3 :txd <= `UD tx_data_reg[2];
134                     4'h4 :txd <= `UD tx_data_reg[3];
135                     4'h5 :txd <= `UD tx_data_reg[4];
136                     4'h6 :txd <= `UD tx_data_reg[5];
137                     4'h7 :txd <= `UD tx_data_reg[6];
138                     4'h8 :txd <= `UD tx_data_reg[7];
139                     4'h9 :txd <= `UD 1'b1;
140                     default :txd <= `UD 1'b1;
141                 endcase
142             end
143         else
144             txd <= `UD 1'b1;
145         end
146     end
147

```

方法二的 module 设计如下:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module uart_tx #(
4      parameter      BPS_NUM = 16'd434
5      // 设置波特率为4800时, bit 位宽时钟周期个数:50MHz set 10417 40MHz set 8333
6      // 设置波特率为9600时, bit 位宽时钟周期个数:50MHz set 5208 40MHz set 4167
7      // 设置波特率为115200时, bit 位宽时钟周期个数:50MHz set 434 40MHz set 347 12M set 104
8  )
9  (
10     input          clk,          // clock                      时钟信号
11     input [7:0]    tx_data,      // uart tx data signal byte; 等待发送的字节数据
12     input          tx_pluse,     // uart tx enable signal,rising is active; 发送模块发送触发信号
13
14     output reg     uart_tx,      // uart tx transmit data line 发送模块串口发送信号线
15     output         tx_busy      // uart tx module work states,high is busy;发送模块忙状态指示
16 );
17

```

```

18 //=====
19 //wire and reg in the module
20 //=====
21 reg          tx_pluse_reg =0;
22 reg [7:0]    trans_data=0;
23
24 reg [2:0]    tx_bit_cnt=0; //the bits number has transmited.
25
26 reg [2:0]    tx_state=0; //current state of tx state machine.
27 reg [2:0]    tx_state_n=0; //next state of tx state machine.
28
29 reg [3:0]    pluse_delay_cnt=0;
30 reg          tx_en = 0;
31
32 // uart tx state machine's state
33 localparam IDLE      = 4'h0; //tx state machine's state.空闲状态
34 localparam SEND_START = 4'h1; //tx state machine's state.发送 start 状态
35 localparam SEND_DATA  = 4'h2; //tx state machine's state.发送数据状态
36 localparam SEND_STOP  = 4'h3; //tx state machine's state.发送 stop 状态
37 localparam SEND_END   = 4'h4; //tx state machine's state.发送结束状态
38
39 // uart bps set the clk's frequency is 50MHz
40 reg [15:0]    clk_div_cnt=0; //count for division the clock.
41
42 //=====
43 //logic
44 //=====
45 assign tx_busy = (tx_state != IDLE);
46 //some control single.
47
48 always @(posedge clk)
49 begin
50     tx_pluse_reg <= `UD tx_pluse;
51 end
52
53 // uart 锁存待发射数据
54
55 always @(posedge clk)
56 begin
57     if(~tx_pluse_reg & tx_pluse)
58         trans_data <= `UD tx_data;
59 end
60
61 // uart 模块发送工作使能标志信号
62 always @(posedge clk)
63 begin
64     if(~tx_pluse_reg & tx_pluse)
65         tx_en <= `UD 1'b1;
66     else if(tx_state == SEND_END)
67         tx_en <= `UD 1'b0;
68 end
69

```

```

70 //division the clock to satisfy baud rate.波特周期计数器
71 always @ (posedge clk)
72 begin
73     if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
74         clk_div_cnt <= `UD 16'h0;
75     else
76         clk_div_cnt <= `UD clk_div_cnt + 16'h1;
77 end
78
79 //count the number has transmited.发送数据状态中, 发送 bit 位数, 以波特周期累加
80 always @ (posedge clk)
81 begin
82     if(!tx_en)
83         tx_bit_cnt <= `UD 3'h0;
84     else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
85         tx_bit_cnt <= `UD 3'h0;
86     else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
87         tx_bit_cnt <= `UD tx_bit_cnt + 3'h1;
88     else
89         tx_bit_cnt <= `UD tx_bit_cnt;
90 end
91
92 //=====
93 //transmitter state machine
94 //=====
95 // state change 状态跳转
96 always @(posedge clk)
97 begin
98     tx_state <= tx_state_n;
99 end
100
101 // state change condition 状态跳转条件及规律
102 always @ (*)
103 begin
104     case(tx_state)
105     IDLE :
106     begin
107         if(~tx_pluse_reg & tx_pluse) //触发发送做 16 个 clock 延时后跳到发送 start 状态
108             tx_state_n = SEND_START;
109         else
110             tx_state_n = tx_state;
111     end
112     SEND_START :
113     begin
114         if(clk_div_cnt == BPS_NUM) //发送一个波特周期的低电平后进入, 发送数据状态
115             tx_state_n = SEND_DATA;
116         else
117             tx_state_n = tx_state;
118     end

```

```

119     SEND_DATA :
120     begin
121         if(tx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
122             //计时 8 个波特周期后（发送了 8bit 数据），跳转到发送 stop 状态
123             tx_state_n = SEND_STOP;
124         else
125             tx_state_n = tx_state;
126         end
127     SEND_STOP :
128     begin
129         if(clk_div_cnt == BPS_NUM)
130             //设置停止位宽为 1 个波特周期，计数发送一个波特周期的高电平，之后跳转到发送结束状态
131             tx_state_n = SEND_END;
132         else
133             tx_state_n = tx_state;
134         end
135     SEND_END : tx_state_n = IDLE;
136     default : tx_state_n = IDLE;
137 endcase
138 end
139
140 // logical ouput 状态机输出
141 always @ (posedge clk)
142 begin
143     if(tx_en)
144     begin
145         case(tx_state)
146             IDLE : uart_tx <= `UD 1'h1; //空闲状态输出高电平
147             SEND_START : uart_tx <= `UD 1'h0; //start 状态发送一个波特周期的低电平
148             SEND_DATA : //发送状态每个波特周期发送一个 bit;
149             begin
150                 case(tx_bit_cnt)
151                     3'h0 : uart_tx <= `UD trans_data[0];
152                     3'h1 : uart_tx <= `UD trans_data[1];
153                     3'h2 : uart_tx <= `UD trans_data[2];
154                     3'h3 : uart_tx <= `UD trans_data[3];
155                     3'h4 : uart_tx <= `UD trans_data[4];
156                     3'h5 : uart_tx <= `UD trans_data[5];
157                     3'h6 : uart_tx <= `UD trans_data[6];
158                     3'h7 : uart_tx <= `UD trans_data[7];
159                     default: uart_tx <= `UD 1'h1;
160                 endcase
161             end
162             SEND_STOP : uart_tx <= `UD 1'h1; //发送停止状态 输出 1 个波特周期高电平
163             default : uart_tx <= `UD 1'h1;
164             // 其他状态默认与空闲状态一致，保持高电平输出
165         endcase
166     end
167     else
168         uart_tx <= `UD 1'h1;
169     end
170
171 endmodule
172

```

6.4.2 串口接收模块设计

串口接收模块是发射模块的逆过程，设计思路区别不大，但是有如下几点需要注意：

1. 接收开始信号，当 rx 下降沿到来后保持几个时钟周期的低电平，表明进入接收 start；
2. 接收数据提取位置，前面讲发射的时候都是在波特周期开始的位置变更数据，接收数据提取时需要在 rx 稳定时刻取数，去波特周期的中间位置取数；
3. 最终输出数据锁存，在最后 1bit 存入寄存器后需要对接收数据锁存，并在之后需要给出数据使能信号，表示输出数据有效；

Module 设计如下：

```

1  `timescale 1ns / 1ps
2  `define UD #1
3
4  module uart_rx # (
5      parameter          BPS_NUM      =    16'd433
6      // 设置波特率为 4800 时,   bit 位宽时钟周期个数:50MHz set 10417   40MHz set 8333
7      // 设置波特率为 9600 时,   bit 位宽时钟周期个数:50MHz set 5208    40MHz set 4167
8      // 设置波特率为 115200 时, bit 位宽时钟周期个数:50MHz set 434     40MHz set 347
9  )
10 (
11     //input ports
12     input          clk,
13     input          rstn,
14     input          uart_rx,
15
16     //output ports
17     output reg [7:0] rx_data,
18     output reg      rx_en,
19     output          rx_finish
20 );
21
22     // uart rx state machine's state
23     localparam IDLE      = 4'h0;    //空闲状态，等待开始信号到来。
24     localparam RECEIV_START = 4'h1; //接收 Uart 开始信号，低电平一个波特周期。
25     localparam RECEIV_DATA  = 4'h2; //接收 Uart 传输数据信号，
26     localparam RECEIV_STOP  = 4'h3; //停止状态数据线是高电平，
27     localparam RECEIV_END   = 4'h4; //结束中转状态。
28
29     //=====
30     //wire and reg in the module
31     //=====
32     reg [2:0] rx_state=0; //current state of tx state machine. 当前状态
33     reg [2:0] rx_state_n=0; //next state of tx state machine. 下一个状态
34     reg [7:0] rx_data_reg; //接收数据缓冲寄存器
35     reg      uart_rx_1d; //save uart_rx one cycle. 保存 uart_rx 一个时钟周期
36     reg      uart_rx_2d; //save uart_rx one cycle. 保存 uart_rx 前两个时钟周期
37     wire      start; //active when start a byte receive. 检测到 start 信号标志
38     reg [15:0] clk_div_cnt; //count for division the clock. 波特周期计数器
39

```

```

40 //=====
41 //logic
42 //=====
43 //some control single.
44 always @ (posedge clk)
45 begin
46     uart_rx_1d <= `UD uart_rx;
47     uart_rx_2d <= `UD uart_rx_1d;
48 end
49
50 assign start      = (!uart_rx) && (uart_rx_1d || uart_rx_2d);
51 assign rx_finish = (rx_state == RECEIV_END);
52
53 //division the clock to satisfy baud rate.波特周期计数器
54 always @ (posedge clk)
55 begin
56     if(rx_state == IDLE || clk_div_cnt == BPS_NUM)
57         clk_div_cnt <= `UD 16'h0;
58     else
59         clk_div_cnt <= `UD clk_div_cnt + 16'h1;
60 end
61
62 // receive bit data numbers
63 //在接收数据状态中, 接收的 bit 位数, 每一个波特周期计数加 1
64 reg [2:0] rx_bit_cnt=0; //the bits number has transmtited.
65 always @ (posedge clk)
66 begin
67     if(rx_state == IDLE)
68         rx_bit_cnt <= `UD 3'h0;
69     else if((rx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
70         rx_bit_cnt <= `UD 3'h0;
71     else if((rx_state == RECEIV_DATA) && (clk_div_cnt == BPS_NUM))
72         rx_bit_cnt <= `UD rx_bit_cnt + 3'h1;
73     else
74         rx_bit_cnt <= `UD rx_bit_cnt;
75 end
76 //=====
77 //receive state machine
78 //=====
79 //状态机状态跳转
80 always @(posedge clk)
81 begin
82     rx_state <= rx_state_n;
83 end
84
85 //状态机状态跳转条件及跳转规律
86 always @ (*)
87 begin
88     case(rx_state)
89         IDLE :
90         begin
91             if(start) //监测到 start 信号到来, 下一状态跳转到 start 状态
92                 rx_state_n = RECEIV_START;
93             else
94                 rx_state_n = rx_state;
95         end

```

```

96     RECEIV_START    :
97     begin
98         if(clk_div_cnt == BPS_NUM)           //已完成接收 start 标志信号
99             rx_state_n = RECEIV_DATA;
100        else
101            rx_state_n = rx_state;
102        end
103     RECEIV_DATA     :
104     begin
105         if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
106             //已完成 8bit 数据的传输
107             rx_state_n = RECEIV_STOP;
108         else
109             rx_state_n = rx_state;
110        end
111     RECEIV_STOP     :
112     begin
113         if(clk_div_cnt == BPS_NUM) //已完成接收 stop 标志信号
114             rx_state_n = RECEIV_END;
115         else
116             rx_state_n = rx_state;
117        end
118     RECEIV_END      :
119     begin
120         if(!uart_rx_1d)
121             //数据线重新被拉低，表示新数据传输又发送 start 标志信号，需要跳转到 start 状态
122             rx_state_n = RECEIV_START;
123         else
124             //没有其他状况出现时，回到空闲状态，等待 start 信号的到来
125             rx_state_n = IDLE;
126        end
127     default         : rx_state_n = IDLE;
128 endcase
129 end
130
131 // 状态机输出
132 always @ (posedge clk)
133 begin
134     case(rx_state)
135         IDLE
136         ,
137         RECEIV_START :
138             //在空闲和 start 状态时将接收数据缓冲寄存器和数据使能置位；
139             begin
140                 rx_en <= `UD 1'b0;
141                 rx_data_reg <= `UD 8'h0;
142             end
143         RECEIV_DATA :
144             begin
145                 if(clk_div_cnt == BPS_NUM[15:1])
146                     //在一个波特周期的中间位置取数据线上传输的数据；
147                     rx_data_reg <= `UD {uart_rx , rx_data_reg[7:1]}; //以循环右移的方式将
148                     uart_rx 数据填入缓冲寄存器的最高位（Uart 传输低位在前，最后一个 bit 刚好是最高位）
149             end

```

```

148     RECEIV_STOP :
149     begin
150         rx_en  <= `UD 1'b1;      // 输出使能信号, 表示最新的数据输出有效
151         rx_data <= `UD rx_data_reg; // 将缓冲寄存器的值赋值给输出寄存器
152     end
153     RECEIV_END :
154     begin
155         rx_data_reg <= `UD 8'h0;
156     end
157     default:    rx_en <= `UD 1'b0;
158 endcase
159 end
160
161 endmodule
162

```

6.4.3 串口发射控制模块设计

目标: 产生 1S 间隔的触发信号并输出第一个发送字节, busy 的下降沿时输出下一个字节;

Module 如下:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module uart_data_gen(
4      input          clk,
5      input          rstn,
6      input  [7:0]   read_data,
7      input          tx_busy,
8      input  [7:0]   write_max_num,
9      output reg [7:0] write_data,
10     output reg      write_en
11 );
12     // set every second send a string, "===HELLO WORLD==="
13     // 设置约每秒发送一个字符串
14     reg [23:0] time_cnt=0;
15     reg [ 7:0] data_num;
16     always @(posedge clk)
17     begin
18         time_cnt <= `UD time_cnt + 24'd1;
19     end
20
21     // 设置串口发射工作区间
22     reg      work_en=0;
23     reg      work_en_1d=0;
24     always @(posedge clk)
25     begin
26         if(time_cnt == 25'd2048)
27             work_en <= `UD 1'b1;
28         else if(data_num == write_max_num-1'b1)
29             work_en <= `UD 1'b0;
30     end

```

```

32     always @(posedge clk)
33     begin
34         work_en_1d <= `UD work_en;
35     end
36
37     // get the tx_busy's falling edge  获取 tx_busy 的下降沿
38     reg          tx_busy_reg=0;
39     wire          tx_busy_f;
40     always @ (posedge clk) tx_busy_reg <= `UD tx_busy;
41
42     assign tx_busy_f = (!tx_busy) && (tx_busy_reg);
43
44     // 串口发射数据触发信号
45     reg write_pluse;
46     always @ (posedge clk)
47     begin
48         if(!rstn)
49             write_pluse <= `UD 1'b0;
50         else if(work_en)
51             begin
52                 if(~work_en_1d || tx_busy_f)
53                     write_pluse <= `UD 1'b1;
54                 else
55                     write_pluse <= `UD 1'b0;
56             end
57         else
58             write_pluse <= `UD 1'b0;
59     end
60
61     always @ (posedge clk)
62     begin
63         if(!rstn)
64             data_num <= `UD 8'h0;
65         else if(~work_en & tx_busy_f)
66             data_num <= 7'h0;
67         else if(write_pluse)
68             data_num <= data_num + 8'h1;
69     end
70
71     always @(posedge clk)
72     begin
73         write_en <= `UD write_pluse;
74     end
75
76     // 字符的对应 ASCII 码
77     // H:0x48    E:0x45    L:0x4C    O:0x4F    W:0x57    R:0x52    D:0x44
78     always @ (posedge clk)
79     begin
80         case(data_num)
81             8'h0 ,
82             8'h1 : write_data <= `UD 8'h77; // ASCII code is w
83             8'h2 : write_data <= `UD 8'h77; // ASCII code is w
84             8'h3 : write_data <= `UD 8'h77; // ASCII code is w
85             8'h4 : write_data <= `UD 8'h2E; // ASCII code is .
86             8'h5 : write_data <= `UD 8'h6D; // ASCII code is m

```

```

87      8'h6 : write_data <= `UD 8'h65; // ASCII code is e
88      8'h7 : write_data <= `UD 8'h79; // ASCII code is y
89      8'h8 : write_data <= `UD 8'h65; // ASCII code is e
90      8'h9 : write_data <= `UD 8'h73; // ASCII code is s
91      8'ha : write_data <= `UD 8'h65; // ASCII code is e
92      8'hb : write_data <= `UD 8'h6D; // ASCII code is m
93      8'hc : write_data <= `UD 8'h69; // ASCII code is i
94      8'hd : write_data <= `UD 8'h2E; // ASCII code is .
95      8'he : write_data <= `UD 8'h63; // ASCII code is c
96      8'hf : write_data <= `UD 8'h6F; // ASCII code is o
97      8'h10 : write_data <= `UD 8'h6D; // ASCII code is m
98      8'h11 ,
99      8'h12 : write_data <= `UD 8'h0d;
100     8'h13 : write_data <= `UD 8'h0a;
101     default : write_data <= `UD read_data;
102 endcase
103 end
104
105 endmodule
106

```

6.4.4 串口实验顶层模块设计

目标: 板子每隔 1s 向串口助手发送一次 ASCII 码的 “www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字，LED 以二进制显示亮起。

Uart_data_gen 模块产生一个间隔 1S 钟的触发信号，同时输出第一个发送字节，等待 uart_tx 输出的 busy 下降沿到来，获知 uart_tx 进入空闲状态可发送下一个 byte 时，再次给出串口发送的触发脉冲，并输出下一个字节；

Uart_rx 模块接收到数据后输出一个 rx_en 信号（接收数据使能信号）、一组接收数据信号；接收的数据信号是锁存的，可直接点亮 LED 灯；

具体的 module 实现如下：

```

1  `timescale 1ns / 1ps
2  `define UD #1
3
4  module uart_top(
5      //input ports
6      input      clk,
7      input      rstn,
8      input      uart_rx,
9
10     //output ports
11     output [7:0] led,
12     output      uart_tx
13 );
14

```

```

15     parameter      BPS_NUM = 16'd104;
16     // 设置波特率为 4800 时, bit 位宽时钟周期个数:50MHz set 10417 40MHz set 8333
17     // 设置波特率为 9600 时, bit 位宽时钟周期个数:50MHz set 5208 40MHz set 4167
18     // 设置波特率为 115200 时, bit 位宽时钟周期个数:50MHz set 434 40MHz set 347 12M set 104
19     //=====
20     //wire and reg in the module
21     //=====
22     wire            tx_busy;           //transmitter is free.
23     wire            rx_finish;        //receiver is free.
24     wire [7:0]      rx_data;          //the data receive from uart_rx.
25     wire [7:0]      tx_data;
26     wire            tx_en;            //enable transmit.
27     wire            rx_en;
28     //=====
29     //instance
30     //=====
31     reg [7:0] receive_data;
32     always @(posedge clk) receive_data <= led;
33     uart_data_gen uart_data_gen(
34         .clk          ( clk          ),//input          clk,
35         .rstn          ( rstn         ),//input          rstn,
36         .read_data     ( receive_data ),//input          [7:0] read_data,
37         .tx_busy       ( tx_busy      ),//input          tx_busy,
38         .write_max_num  ( 8'h14       ),//input [7:0] write_max_num,
39         .write_data     ( tx_data     ),//output reg [7:0] write_data
40         .write_en       ( tx_en       ) //output reg      write_en
41     );
42
43     //uart transmit data module.
44     uart_tx #(
45         .BPS_NUM       ( BPS_NUM     ) //parameter BPS_NUM = 16'd434
46     )
47     u_uart_tx(
48         .clk           ( clk          ),// input          clk,
49         .tx_data       ( tx_data     ),// input [7:0]      tx_data,
50         .tx_pluse      ( tx_en       ),// input          tx_pluse,
51         .uart_tx       ( uart_tx     ),// output reg      uart_tx,
52         .tx_busy       ( tx_busy     ) // output          tx_busy
53     );
54
55     //Uart receive data module.
56     uart_rx #(
57         .BPS_NUM       ( BPS_NUM     ) //parameter BPS_NUM = 16'd434
58     )
59     u_uart_rx (
60         .clk           ( clk          ),// input          clk,
61         .rstn          ( rstn         ),// input          rstn,
62         .uart_rx       ( uart_rx     ),// input          uart_rx,
63         .rx_data       ( rx_data     ),// output reg [7:0] rx_data,
64         .rx_en         ( rx_en       ),// output reg      rx_en,
65         .rx_finish     ( rx_finish   ) // output          rx_finish
66     );
67     assign led = rx_data;
68
69 endmodule
70

```

6.5 实验现象

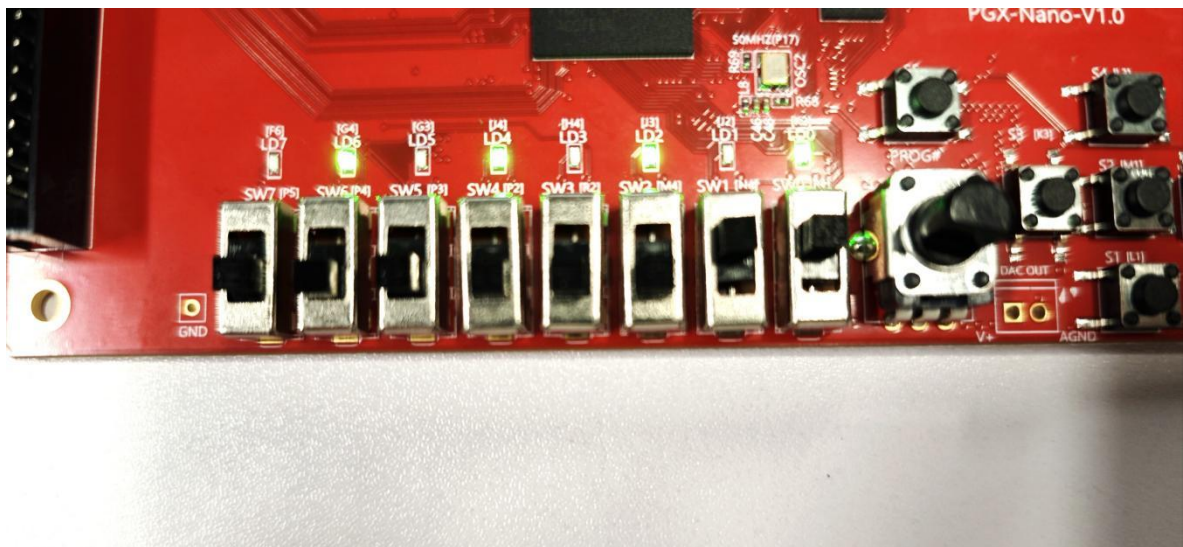
用 SSCOM 串口调试工具，波特率设置为 115200bps，数据格式为 1 位起始位、8 位数据位、无校验位、1 位结束位，用 Type-C 连接开发板与电脑后有如下现象：

实验现象一：在串口工具中每隔 1S 中打印一次：“www.meyesemi.com”；

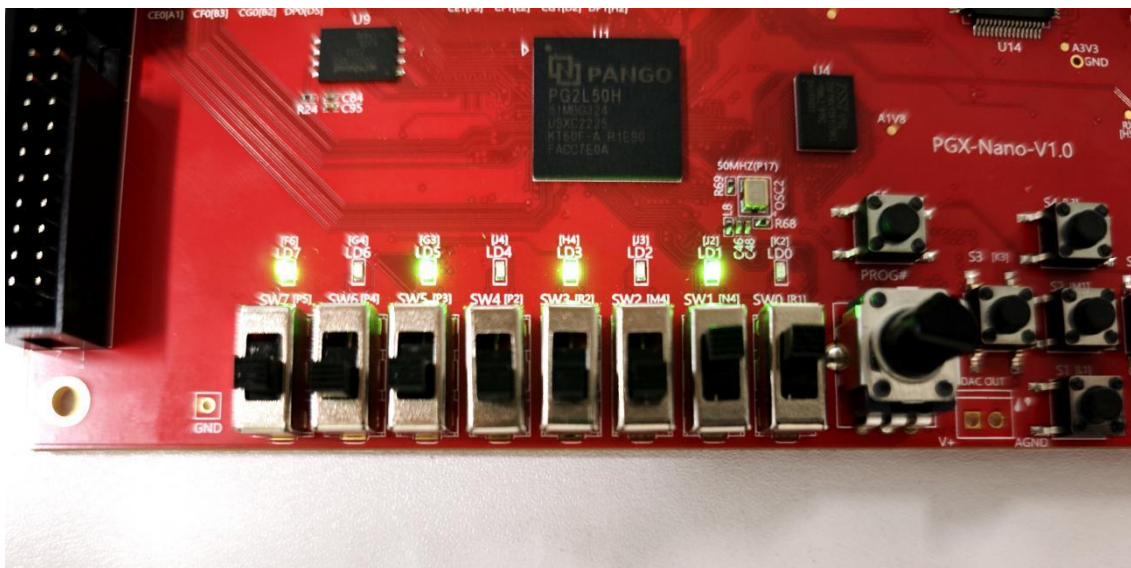


实验现象二：

在串口工具上以 Hex 格式发送 55；我们可看到 PGX-Nano 板卡上的 LED0, LED2, LED4, LED6 被点亮，LED1, LED3, LED5, LED7 为熄灭状态；



发送 AA；我们可看到 PGX-Nano 板卡上的 LED2, LED4, LED6, LED8 被点亮，LED1, LED3, LED5, LED7 为熄灭状态。



也可以试着发送其他数据（00~FF），看一下 LED 灯的变化；