Oversampling UART Reduces RF Noise

BY ELI FLAXER

Reap The Benefits Of Programmable Logic By Implementing Wireless Communications Based On An Oversampling Algorithm.

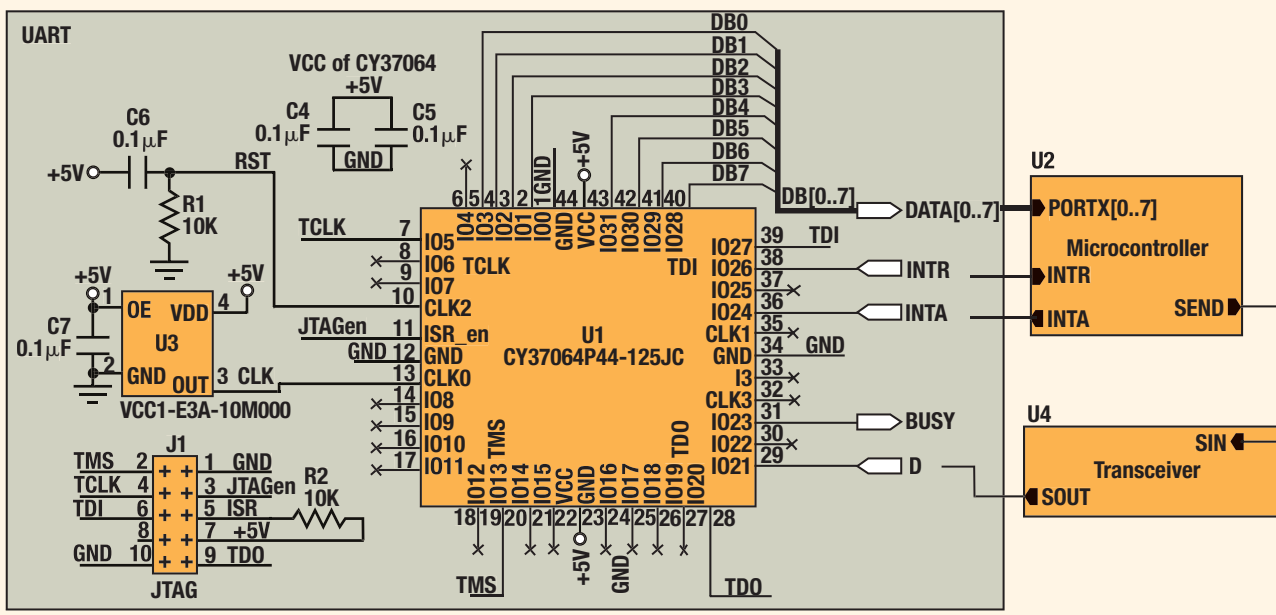
In recent years, wireless communications have grown to encompass applications like remote control, remote sensing, and wireless local-area networks (WLANs). The data rates of such implementations vary from a few hundred to several million bits per second. In addition, their range can be anywhere from a few to several hundred meters. Yet almost all of these wireless applications are the same in that they provide serial, asynchronous digital data formats.

On the analog side, the physical channels are shrouded in background noise. This noise comes from other transmitters and instruments in the surrounding environment. Moreover, some receivers will detect "fake data" when the transmitter is out of range. An example is a receiver that implements FSK modulation schemes. The "fake data" is actually signals with random frequency. These error challenges necessitate a different way to implement a physical-layer Universal Asynchronous Receiver Transmitter (UART) for the wireless communication line.

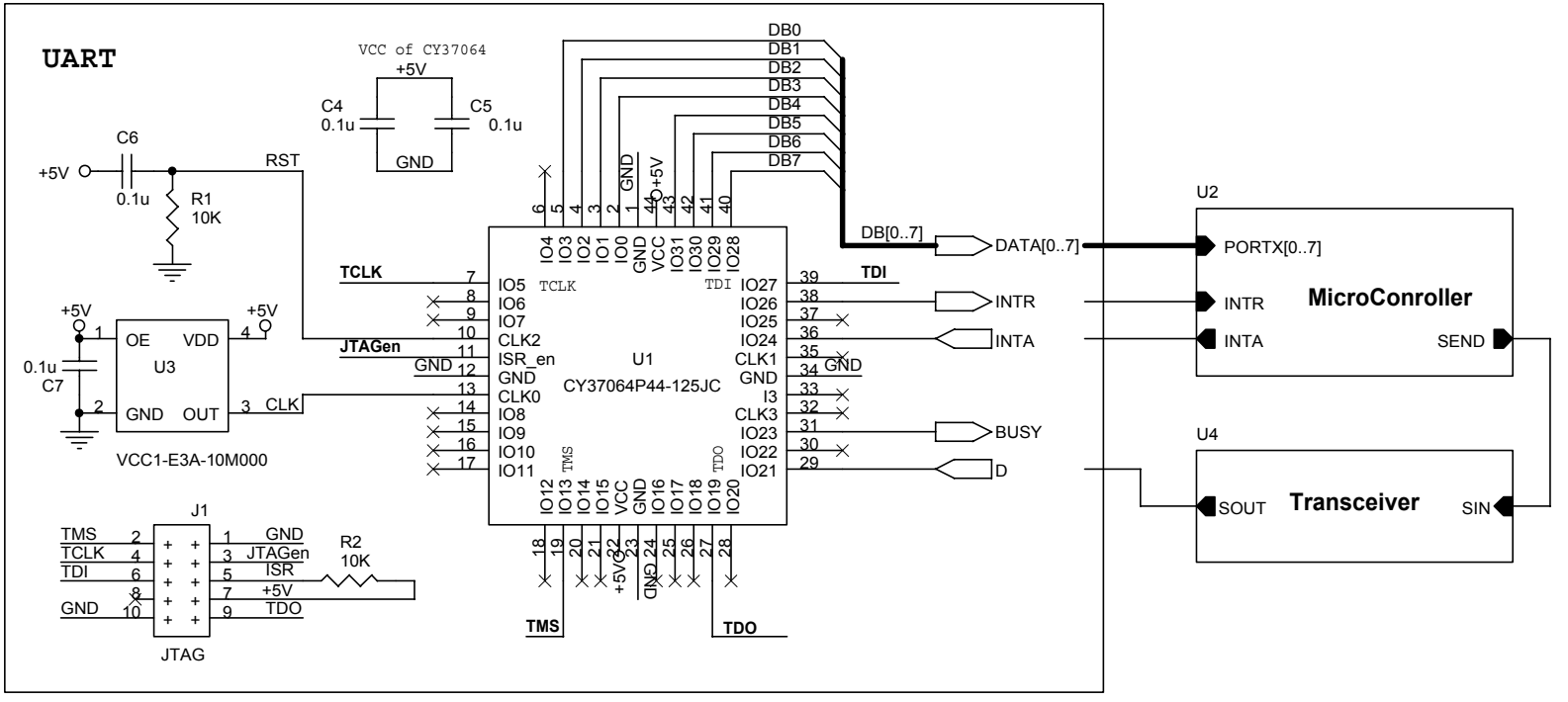
This article describes the use of an oversampling technique to transfer data from one microcontroller to another via a transceiver module. The module itself requires no data coding. But the wireless nature of the transmissions demands the use of a receiving algorithm. This algorithm separates valid data from undesirable noise. As a result, the data needs to be sent in packets or frames that have a well-defined format.

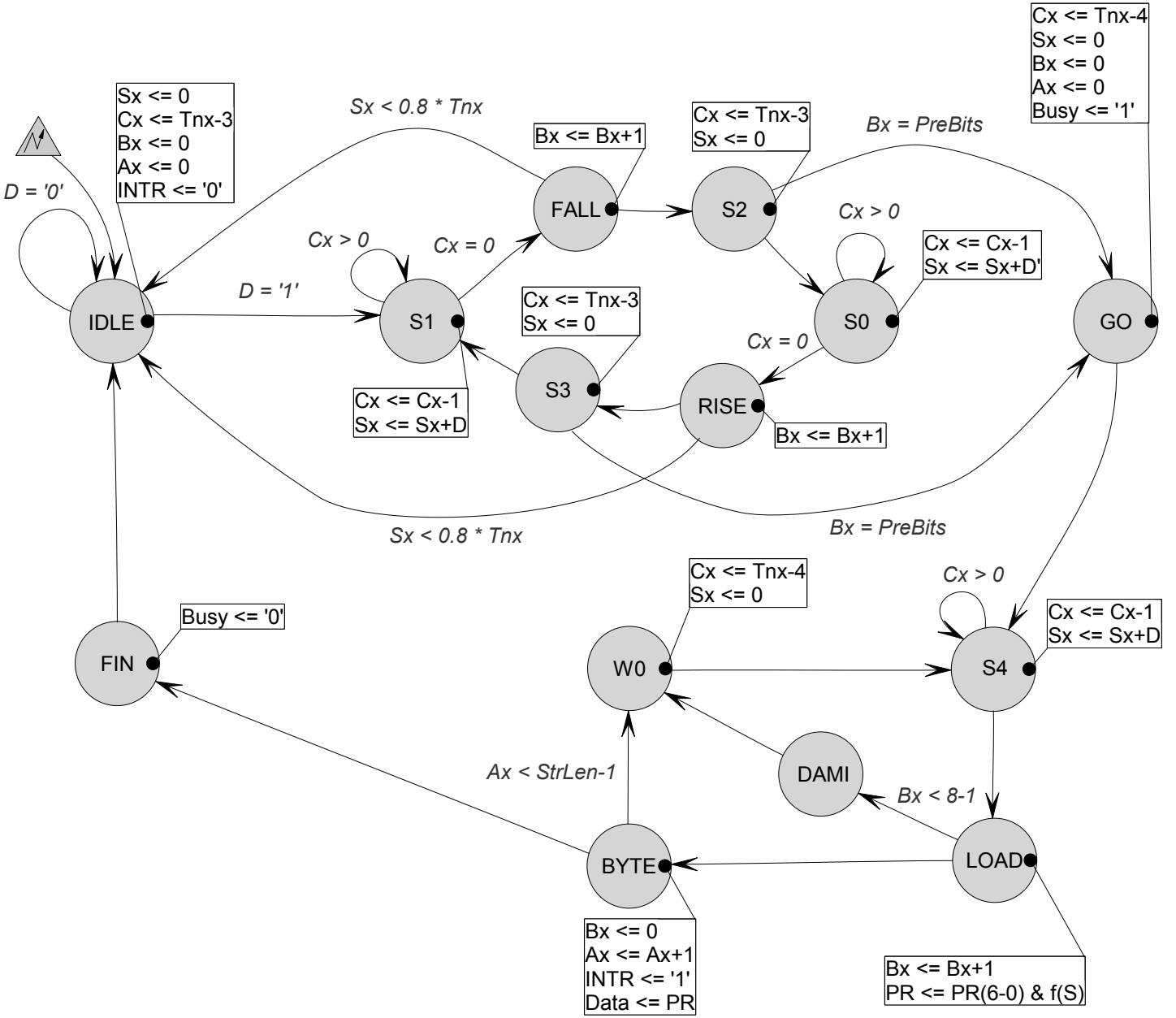
Each frame should begin with a preamble of 1 to 2 B (for example, "10101010…"). The second field of the frame should contain the start byte, which is an identification that indicates the beginning of a frame. The rest of the frame will follow the start byte. It should consist of 1 B of an address (ID). It will be followed by the frame data (1 to 100 B) and finally 2 B for a checksum. The general structure of the frame is as follows:

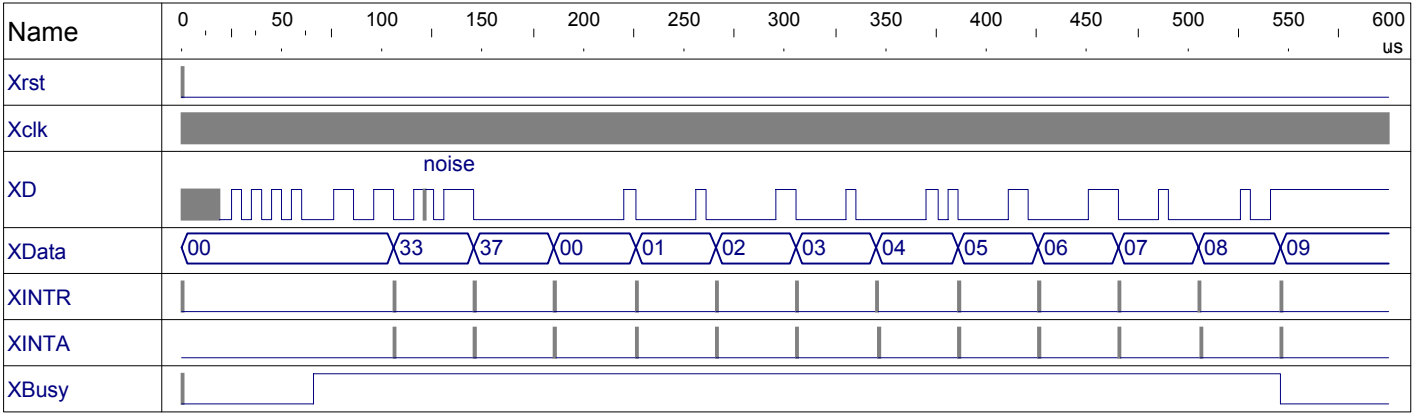
Preamble	Start Byte	Address	Data	CRC



1. This hardware-based oversampling UART receives bit-stream data from a transceiver module by a serial input (D). It then transfers the complete byte to the microcontroller via portx.


UART
VCC of CY37064
+5V
C4
0.1u
C5
0.1u
GND
C6
0.1u
R1
10K
RST
+5V
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
DB[0..7]
DATA[0..7]
TCLK
TDI
INTR
INTA
JTAGen
GND
CLK
BUSY
D
TMS
TDO
U1
CY37064P44-125JC
U2
PORTX[0..7]
MicroConroller
SEND
U3
VCC1-E3A-10M000
U4
SOUT
Transceiver
SIN
J1
JTAG
R2
10K
ISR


Sx <= 0
Cx <= Tnx-3
Bx <= 0
Ax <= 0
INTR <= '0'
D = '0'
IDLE
Sx < 0.8 * Tnx
Bx <= Bx+1
FALL
Cx <= Tnx-3
Sx <= 0
S2
Bx = PreBits
Cx <= Tnx-4
Sx <= 0
Bx <= 0
Ax <= 0
Busy <= '1'
Cx > 0
Cx = 0
D = '1'
S1
Cx <= Cx-1
Sx <= Sx+D'
S0
Cx > 0
GO
Cx <= Tnx-3
Sx <= 0
S3
Cx = 0
RISE
Cx <= Cx-1
Sx <= Sx+D
Bx <= Bx+1
Sx < 0.8 * Tnx
Bx = PreBits
Busy <= '0'
FIN
Cx <= Tnx-4
Sx <= 0
W0
Cx > 0
S4
Cx <= Cx-1
Sx <= Sx+D
Ax < StrLen-1
DAMI
Bx < 8-1
BYTE
LOAD
Bx <= 0
Ax <= Ax+1
INTR <= '1'
Data <= PR
Bx <= Bx+1
PR <= PR(6-0) & f(S)


Name
0
50
100
150
200
250
300
350
400
450
500
550
600
us
Xrst
Xclk
XD
noise
XData
00
33
37
00
01
02
03
04
05
06
07
08
09
XINTR
XINTA
XBusy

```vhdl
--  Over-Sampling Wireless UART Implementing by VHDL
--  Written By Eli Flaxer
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity WirelessUART is
  port (Clk:        in  STD_LOGIC;
        D:          in  STD_LOGIC;
        Rst:        in  STD_LOGIC;
        INTR:       out STD_LOGIC;
        INTA:       in  STD_LOGIC;
        Busy:       out STD_LOGIC;
        Data:       out STD_LOGIC_VECTOR (7 downto 0));
  end;

architecture WirelessUART_Arch of WirelessUART is

constant Freq:    INTEGER := 20000000;      -- Clock Freq of CPLD
constant BPS:     INTEGER := 200000;        -- Transciever Communication Rate

constant Tnx:     INTEGER := Freq / BPS;    -- OverSample per Data Bit
constant StrLen:  INTEGER := 12;            -- Byte per Frame without preamble
constant PreBits: INTEGER := 8;             -- Preamble Bits minimum 8

-- SYMBOLIC ENCODED state machine:  UartState
type    UartState_Type is (IDLE, S0, S1, S2, S3, FALL, RISE,
                           GO, S4, LOAD, DAMI, W0, BYTE, FIN);

subtype CunterType1  is INTEGER range 0 to StrLen+1;
subtype CunterType2  is INTEGER range 0 to PreBits+1;
subtype CunterType3  is INTEGER range -1 to Tnx+1;

-- Uart signal declarations
signal Ax: CunterType1;                     -- Byte Counter in the Frame
signal Bx: CunterType2;                     -- Bit Counter in the Byte
signal Cx: CunterType3;                     -- OverSampling Index in the Bit
signal Sx: CunterType3;                     -- OverSampling Data in the Bit
signal PR: STD_LOGIC_VECTOR (7 downto 0);   -- Packege Byte

signal  UartState:       UartState_Type;
signal  UartStateNext:   UartState_Type;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
function F1 (n: CunterType3 ) return STD_LOGIC is
    begin
        if(n > Tnx/2) then
            return('1');
        else
            return('0');
        end if;
    end function F1;
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
begin
--------------------------------------------------------------------------
    UartSignal_Update: process (Clk, Rst,  UartState)
        constant TnxC: integer := Tnx-1;
    begin
        if Rst = '1' then
            Sx <= 0;
            Cx <= Tnx-2;
            Bx <= 0;
            Ax <= 0;
            INTR <= '0';
            Busy <= '0';
            Data <= (others => '0');
        elsif (Clk'event and Clk = '1') then
              case  UartState is

                when IDLE =>
                    Sx <= 0;
                    Cx <= TnxC-2;
                    Bx <= 0;
                    Ax <= 0;
                    INTR <= '0';
```

```
 76
 77                 when S1 =>
 78                     Cx <= Cx-1;
 79                     if (D = '1') then
 80                         Sx <= Sx+1;
 81                     end if;
 82
 83                 when S0 =>
 84                     Cx <= Cx-1;
 85                     if (D = '0') then
 86                         Sx <= Sx+1;
 87                     end if;
 88
 89                 when FALL =>
 90                     Bx <= Bx+1;
 91
 92                 when RISE =>
 93                     Bx <= Bx+1;
 94
 95                 when S2 =>
 96                     Cx <= TnxC-2;
 97                     Sx <= 0;
 98
 99                 when S3 =>
100                     Cx <= TnxC-2;
101                     Sx <= 0;
102
103                 when GO =>
104                     Cx <= TnxC-3;
105                     Sx <= 0;
106                     Bx <= 0;
107                     Ax <= 0;
108                     Busy <= '1';
109
110                 when S4 =>
111                     Cx <= Cx-1;
112                     if (D = '1') then
113                         Sx <= Sx+1;
114                     end if;
115
116                 when LOAD =>
117                     Bx <= Bx+1;
118                     PR <= PR(6 DOWNTO 0) & F1(Sx);
119
120                 when BYTE =>
121                     Ax <= Ax+1;
122                     Bx <= 0;
123                     INTR <= '1';
124                     Data <= PR;
125
126                 when W0 =>
127                     Cx <= TnxC-3;
128                     Sx <= 0;
129 --                  INTR <= '0';
130
131                 when FIN =>
132                     Busy <= '0';
133
134                 when others =>
135                     null;
136               end case;
137
138             if INTA = '1' then
139                 INTR <= '0';
140             end if;
141
142         end if;
143     end process;
144 -------------------------------------------------------------------------
145     Uart_State_Update: process (Clk, Rst)
146     begin
147         if Rst = '1' then
148              UartState <= IDLE;
149         elsif (Clk'event and Clk = '1') then
150              UartState <=  UartStateNext;
```

```
151         end if;
152     end process;
153 -------------------------------------------------------------------------
154     Uart_State_Next: process( UartState, D, Ax, Bx, Cx, Sx)
155     begin
156       case  UartState is
157 
158         when IDLE =>
159             if D = '1' then
160                 UartStateNext <= S1;
161             else
162                 UartStateNext <= IDLE;
163             end if;
164 
165         when S1 =>
166             if Cx = 0 then
167                 UartStateNext <= FALL;
168             else
169                 UartStateNext <= S1;
170             end if;
171 
172         when S0 =>
173             if Cx = 0 then
174                 UartStateNext <= RISE;
175             else
176                 UartStateNext <= S0;
177             end if;
178 
179         when FALL =>
180             if (Sx < Integer((0.8 * Tnx))) then
181                 UartStateNext <= IDLE;
182             else
183                 UartStateNext <= S2;
184             end if;
185 
186         when RISE =>
187             if (Sx < Integer((0.8 * Tnx))) then
188                 UartStateNext <= IDLE;
189             else
190                 UartStateNext <= S3;
191             end if;
192 
193         when S2 =>
194             if (Bx = PreBits) then
195                 UartStateNext <= GO;
196             else
197                 UartStateNext <= S0;
198             end if;
199 
200         when S3 =>
201             if (Bx = PreBits) then
202                 UartStateNext <= GO;
203             else
204                 UartStateNext <= S1;
205             end if;
206 
207         when GO =>
208              UartStateNext <= S4;
209 
210         when S4 =>
211             if Cx = 0 then
212                 UartStateNext <= LOAD;
213             else
214                 UartStateNext <= S4;
215             end if;
216 
217         when LOAD =>
218             if (Bx < 8-1) then
219                 UartStateNext <= DAMI;
220             else
221                 UartStateNext <= BYTE;
222             end if;
223 
224         when DAMI =>
225              UartStateNext <= W0;
```

```
226
227         when BYTE =>
228             if (Ax < StrLen - 1) then
229                  UartStateNext <= W0;
230             else
231                  UartStateNext <= FIN;
232             end if;
233
234         when W0 =>
235              UartStateNext <= S4;
236
237         when FIN =>
238              UartStateNext <= IDLE;
239
240         when others =>
241              UartStateNext <= IDLE;
242             --null;
243       end case;
244     end process;
245 ----------------------------------------------------------------------------
246 end WirelessUART_Arch;
247
```

```vhdl
--  Over-Sampling Wireless UART Implementing by VHDL
--  Test Bench Simulation
--  Written By Eli Flaxer
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity WirelessUART_Sim is
end;

architecture TestBench of WirelessUART_Sim is

component WirelessUART
  port (Clk:        in  STD_LOGIC;
        D:          in  STD_LOGIC;
        Rst:        in  STD_LOGIC;
        INTR:       out STD_LOGIC;
        INTA:       in  STD_LOGIC;
        Busy:       out STD_LOGIC;
        Data:       out STD_LOGIC_VECTOR (7 downto 0));
end component;

signal Xclk:    STD_LOGIC;
signal Xrst:    STD_LOGIC;
signal XD  :    STD_LOGIC;
signal XINTR:   STD_LOGIC;
signal XINTA:   STD_LOGIC;
signal XBusy:   STD_LOGIC;
signal XData:   STD_LOGIC_VECTOR (7 downto 0);


begin

    UART: WirelessUART port map(Xclk, XD, Xrst, Xintr, Xinta, Xbusy, Xdata);

    Xrst <= '0', '1' after 100 ns, '0' after 200 ns;

    process
    begin
        Xclk <= '0';
        wait for 25 ns;
        Xclk <= '1';
        wait for 25 ns;
    end process;

    process
        constant preamble: STD_LOGIC_VECTOR(7 downto 0) := "10101010";
        variable tmpstr: UNSIGNED(7 downto 0);
    begin
        XD <= '0';
        wait for 500 ns;
        for k in 1 to 10 loop
            XD <= '1';
            wait for 1000 ns;
            XD <= '0';
            wait for 1000 ns;
        end loop;

        wait for 5000 ns;
        for k in 7 downto 0 loop
            XD <= preamble(k);
            wait for 5 us;
        end loop;

        tmpstr := To_Unsigned(16#33#,8);
        for k in 7 downto 0 loop
            XD <= tmpstr(k);
            wait for 5 us;
        end loop;

        tmpstr := To_Unsigned(16#37#,8);
        for k in 7 downto 0 loop
            if k = 4 then
                for i in 1 to 10 loop
                    XD <= '0';
```

```
76                      wait for 50 ns;
77                      XD <= '1';
78                      wait for 50 ns;
79                  end loop;
80                  XD <= tmpstr(k);
81                  wait for 4 us;
82              else
83                  XD <= tmpstr(k);
84                  wait for 5 us;
85              end if;
86          end loop;
87
88          for j in 0 to 9 loop
89              tmpstr := To_Unsigned(j,8);
90              for k in 7 downto 0 loop
91                  XD <= tmpstr(k);
92                  wait for 5 us;
93              end loop;
94          end loop;
95
96          wait;
97      end process;
98
99      process
100     begin
101         Xinta <= '0';
102         wait until Xintr = '1';
103         wait for 200 ns;
104         Xinta <= '1';
105         wait for 200 ns;
106     end process;
107 ------------------------------------------------------------------------
108 end TestBench;
```