Oversampling UART Reduces RF Noise

BY ELI FLAXER

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

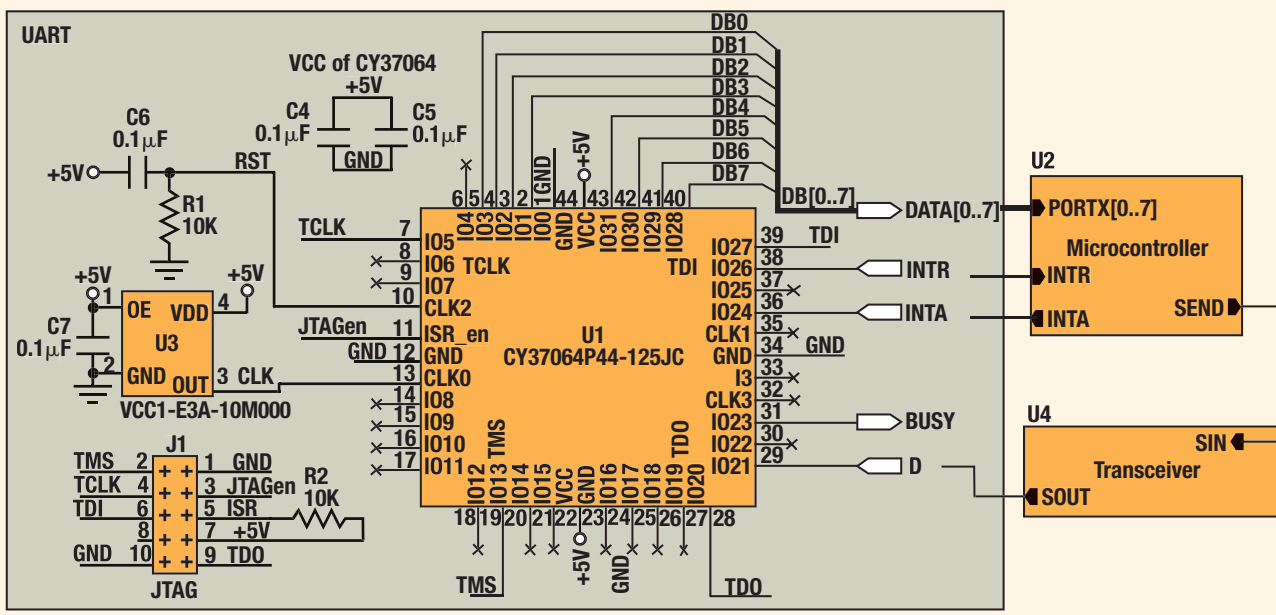
In recent years, wireless communications have grown to encompass applications like remote control, remote sensing, and wireless local-area networks (WLANs). The data rates of such implementations vary from a few hundred to several million bits per second. In addition, their range can be anywhere from 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.

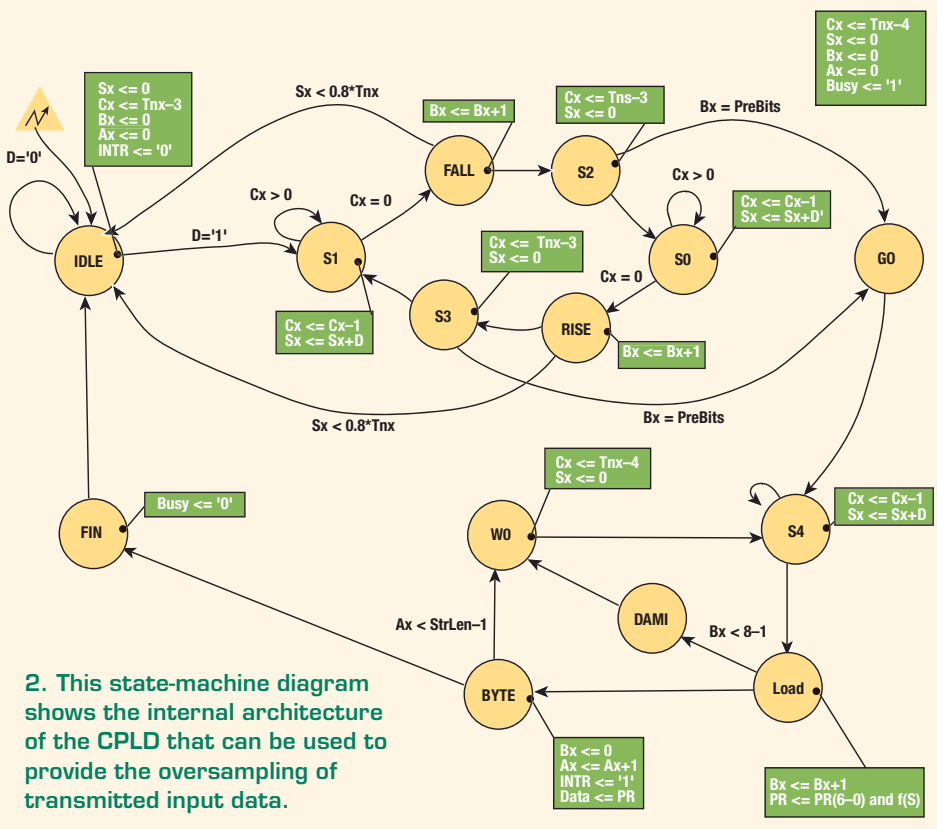
This article describes the use of an oversampling technique to transfer data from one microcontroller to another via a transceiver module. The module itself requires no data coding. But the wireless nature of the transmissions demands the use of a receiving algorithm. This algorithm separates valid data 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.



2. This state-machine diagram shows the internal architecture of the CPLD that can be used to provide the oversampling of transmitted input data.

The transmission procedure is simple: The interrupt handler in the microcontroller should send the frame, bit by bit, at a constant rate. This constant data stream is known as the communication rate.

The receiver is on the other end of the transmission system. The receiving routine is more complex than its counterpart on the transmitting side. For example, the receiver interrupt handler may be triggered by an edge-level rise or fall. Such triggering might cause the interrupt handler to sample the input stream in the middle of each bit. The result would be a reduced data rate of only one sample per bit. This rate would produce an output that is much more sensitive to noise than the oversampling approach.

When using oversampling, the receiving routine should sample the received bit stream at several times the bit rate. After being acquired, the sam-

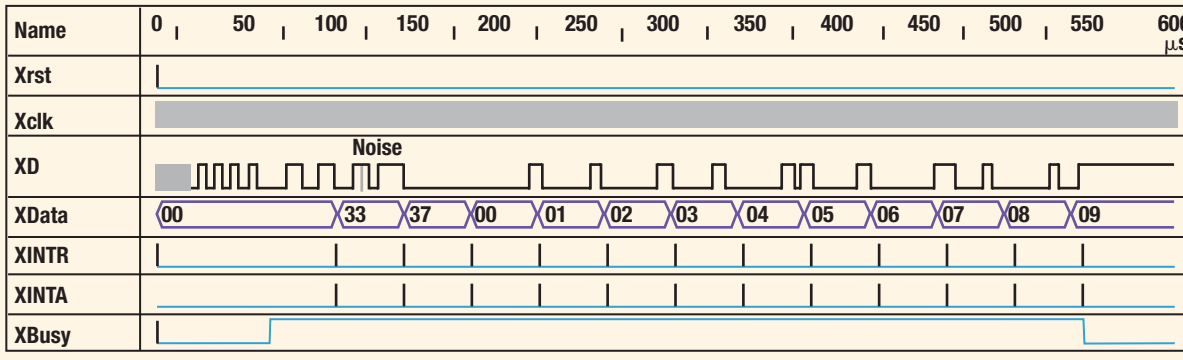
```
1 -- Oversample A Wireless-
UART Implementation With
VHDL
2 -- Written By Eli Flaxer
3 library IEEE;
4 use
IEEE.STD_LOGIC_1164.all;
5 use IEEE.NUMERIC_STD.all;
6
7 entity WirelessUART is
8 port (Clk: in STD_LOGIC;
9 D: in STD_LOGIC;
10 Rst: in STD_LOGIC;
11 INTR: out STD_LOGIC;
12 INTA: in STD_LOGIC;
13 Busy: out STD_LOGIC;
14 Data: out
STD_LOGIC_VECTOR (7 downto
0));
15 end;
16
17 architecture
WirelessUART_Arch of
WirelessUART is
18
19 constant Freq: INTEGER :=
20000000; -- Clock Freq of
CPLD
20 constant BPS: INTEGER :=
200000; -- Transceiver
Communication Rate
21
22 constant Tnx: INTEGER :=
Freq / BPS; -- OverSample per
Data Bit
23 constant StrLen: INTEGER :=
12; -- Byte per Frame without
preamble
24 constant PreBits: INTEGER
:= 8; -- Preamble Bits minimum
8
25
26 -- SYMBOLIC ENCODED
state machine: UartState
27 type UartState_Type is (IDLE,
S0, S1, S2, S3, FALL, RISE,
28 G0, S4, LOAD, DAMI, W0,
BYTE, FIN);
29
30 subtype CunterType1 is
INTEGER range 0 to StrLen+1;
31 subtype CunterType2 is
INTEGER range 0 to PreBits+1;
32 subtype CunterType3 is
INTEGER range -1 to Tnx+1;
33
34 -- Uart signal declarations
35 signal Ax: CunterType1; --
Byte Counter in the Frame
36 signal Bx: CunterType2; --
Bit Counter in the Byte
37 signal Cx: CunterType3; --
OverSampling Index in the Bit
38 signal Sx: CunterType3; --
OverSampling Data in the Bit
39 signal PR:
STD_LOGIC_VECTOR (7 downto
0); -- Package Byte
40
41 signal UartState:
UartState_Type;
42 signal UartStateNext:
UartState_Type;
43
44 --
++++++++++++++++++++
++++++++++++++++++++
++++++++++++++++++++
+++++++++++++++
45 function F1 (n: CunterType3 )
return STD_LOGIC is
46 begin
47 if(n > Tnx/2) then
48 return('1');
49 else
50 return('0');
51 end if;
52 end function F1;
53 --
++++++++++++++++++++
++++++++++++++++++++
++++++++++++++++++++
+++++++++
54 begin
55 ------------------------------
--------------------------------
-----------
56 UartSignal_Update: process
(Clk, Rst, UartState)
57 constant TnxC: integer :=
Tnx-1;
58 begin
59 if Rst = '1' then
60 Sx <= 0;
61 Cx <= Tnx-2;
62 Bx <= 0;
63 Ax <= 0;
64 INTR <= '0';
65 Busy <= '0';
66 Data <= (others => '0');
67 elsif (Clk'event and Clk = '1')
then
68 case UartState is
69
70 when IDLE =>
71 Sx <= 0;
72 Cx <= TnxC-2;
73 Bx <= 0;
74 Ax <= 0;
75 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 G0 =>
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 <= G0;
196 else
197 UartStateNext <= S0;
198 end if;
199
200 when S3 =>
201 if (Bx = PreBits) then
202 UartStateNext <= G0;
203 else
204 UartStateNext <= S1;
205 end if;
206
207 when G0 =>
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
```

ples must be weighted according to a weighting table. All of these computations result in a very reliable transmission link. Oversampling at a rate of 10 times the bit rate with sample weighting, for example, will produce a result that is noticeably resistant to noise.

Designers must note, however, that timing is critical because the sampling period must be consistent. If one uses the microcontroller interrupt to implement an oversampling, all of the system resources will be at the mercy of the interrupt handler. A better method would be to have a dedicated chip—using programmable logic—that performs the task of oversampling. This method won't tie up valuable system resources. At the same time, it will enable much higher oversampling rates (e.g., sample rates of tens of megasamples per second).

Consider the use of a hardware-based oversampling UART **(FIG. 1)**. Component U1-CY37064P44 is a programmable-logic device (CPLD) from Cypress Semiconductor (www.cypress.com). This component boasts 64 macro cells. It has 44 pins that are triggered by



3. In the results from a VHDL simulation testbench, high-frequency noise is rejected by the UART.

a 20-MHz clock. Of course, any other CPLD with similar capabilities also could be used. One example might be the ispMACH4A from Lattice Semiconductor (www.latticesemi.com).

The designer can program the CPLD with any industrial, parallel CPLD programmer. But a preferred method is to use In-System Reprogramming (ISR). J1 is a JTAG connector, which is designed for this purpose. The designer connects the UltraISR or C3ISR programming cable between J1 and PC. After loading the appropriate JED file, he or she can then begin programming the device.

In this UART circuit, one serial input (D) receives the bit stream from the transceiver module. A data bus (DATA) transfers a complete byte to the microcontroller. That microcontroller contains the traditional control ports for interrupt request (INTR): trigger the microcontroller, interrupt acknowledge (INTA), and a busy line (BUSY).

The internal architecture of the CPLD can be described using state-machine diagrams **(FIG. 2)**. The state machine includes several constants: PreBits (number of bits in the preamble); StrLen (number of bytes in the frame excluding the preamble); and Tnx (number of samplings per bit). In addition, there are four counters: Ax, Bx, Cx, and Sx. They count bytes, bits, oversampling index, and oversampling data, respectively. The CPLD contains one register (PR) to store the received bits. At IDLE state, the machine is waiting for a rising edge in the input stream D, which is the beginning of the preamble.

The next six states act as a preamble detector. At states S1 and S0, the machine oversamples bits '1' and '0' of the preamble. At states FALL and RISE, the machine checks if the average sample is correct. The machine checks for the end of the preamble at states S2 and S3. The busy line is set at GO, which means that real data is received.

At state S4, the machine oversamples and averages the data bits. At LOAD, eight average data bits are packed into 1 B. An interrupt request (INTR) is sent to the microcontroller at state W0 to collect the byte. At state BYTE, the machine checks for the end of the frame. Finally, at FIN, it clears the busy line to indicate that the entire frame was received. The DAMI state is for balancing the path in the machine. If INTA occurs at any state, the interrupt acknowledge (INTR) will clear.

> ■ ■ ■
> …THE SAMPLING PERIOD MUST BE CONSISTENT. IF ONE USES THE MICROCONTROLLER INTERRUPT TO IMPLEMENT AN OVERSAMPLING, ALL OF THE SYSTEM RESOURCES WILL BE AT THE MERCY OF THE INTERRUPT HANDLER.

Several methods are commonly used to design, synthesize, and simulate the content of the complex programmable-logic device. Typically, VHDL code is written according to the state-machine flow to synthesize and simulate the design. In VHDL, the designer can search for the best tradeoffs between the clock frequency, data transfer rate, and number of bytes per frame. For example, the VHDL source file created for this oversampling project was based on a 20-MHz clock, 200-kbps data transfer rate, 12 B per frame, and a preamble of 8 b **(SEE SIDEBAR)**. Naturally, the designer can change those values as necessary to examine other tradeoff conditions.

A testbench is needed to verify the VHDL simulation. Given the preceding example, the simulation would start with high-frequency noise (500 kHz) followed by a correct preamble and 12 B of data (a complete frame). The resulting simulation would help to test the design **(FIG. 3)**.

Here, the testbench indicated that the UART rejects high-frequency noise (or a preamble with a different frequency). The correct preamble is detected and the BUSY line is set. The 12 bytes that follow the preamble (complete frame) are 33, 37, 00, 01, 02, 03, 04, 05, 06, 07, 08, and 09. Although the second byte (37) includes high-frequency noise at the fourth bit, the UART rejects this noise. It sends the corrected data to the bus. One can see that the UART packs eight serial bits to one parallel byte and triggers the microcontroller by the INTR command. When a complete frame is received, the BUSY line turns to zero. The UART is then ready to receive the next frame.

This article presents one way to implement a physical layer (UART) for wireless communications. By using an oversampling technique, designers can follow this very approach. It should lead them to develop a subsystem that is resistant to noise and other interference. ■

Dr. Eli Flaxer, *Senior Lecturer, Electrical Engineering and Computer Science, Tel-Aviv Academic College of Engineering, 61533 Tel-Aviv, Israel; e-mail: flaxer@mail.tace.ac.il, www.tace.ac.il.*