Implementing a Reframe Controller for the CY7B933 HOTLink™ Receiver in a CY37032 CPLD

Introduction

This application note describes a reframe controller for the Cypress CY7B933 HOTLink™ Receiver. The primary function of the controller is to monitor the Receive Violation Symbol output, RVS, from the CY7B933 in order to detect framing errors and, under the correct conditions, assert the Reframe signal, RF, to the CY7B933. The controller function is designed with a state machine, a few counters, and some decode logic. All are implemented in VHDL and fit into a Cypress CY37032 32-macrocell CPLD. The exact implementation in this application note makes several assumptions about the next-higher-level controller that may not be universally applicable. However, the source code for the design is provided in Appendix A at the end of this application note so that modification and customization for other interfaces is easily possible.

Why Reframing is Necessary

The CY7B923 and CY7B933 HOTLink Transmitter and Receiver are a pair of chips for high-speed point-to-point serial data communication. The CY7B923 is the transmitter, and the CY7B933 is the receiver. The CY7B923 takes in an 8-bit byte at a frequency between 16 and 33 MHz, encodes it into 10 bits, does a parallel-to-serial conversion, and then transmits the serial data at ten times the byte-rate clock (about 160 to 330 Megabits per second (Mbps)). At the other end of the link, the CY7B933 receives the serial data, does a serial-to-parallel conversion, unencodes the data back into its original form, and shifts the 8-bit parallel data out at the same byte-rate clock frequency used by the transmitter. (Note: the chips can also transmit and receive 10 bits of unencoded data. For a full description of the encoding and decoding functions, see the CY7B923/933 datasheet.)

The key element in the data-and-clock-recovery circuit on the receiver is the phase-locked loop (PLL) on the chip. It is triggered by the transitions in the incoming data stream, and it is used to both separate the data stream into individual bits and to generate the byte-rate clock going out of the chip. Once the PLL achieves synchronization with the incoming serial data stream and is receiving bits properly, the receiver must be given a reference point that will set the byte boundaries in the bit stream. This is done by the framing circuitry. Whenever the receiver's RF (reframe) input is asserted, the receiver's framing logic will check the incoming bit stream for the special pattern that defines a byte boundary. When this is found, the receiver logic sets a reference point and simply counts bits from that point on so it can properly execute the serial-to-parallel conversion on subsequent byte boundaries, and properly align the byte-rate clock rising edge.

Thus, framing is always required when the receiver begins receiving data for the first time, either at power-up or after switching from one transmitter source to another. Periodic reframing may also be necessary due to other conditions. If the PLL goes out of lock—that is, if it loses its synchronization with the incoming serial bit stream for any reason—the recovered data will be erroneous and the framing boundary information will be lost. Once the PLL gets back into synchronization with the incoming bit stream, it will be necessary to force the receiver to reframe in order to re-establish the proper byte boundary point.

Using RVS to Know When to Reframe

The PLL out-of-lock condition can be detected by the behavior of the RVS output of the CY7B933 receiver. The CY7B933 asserts RVS when it detects an error in the bit stream. Infrequent errors, due to random noise in the environment or attenuation by the transmission medium, for example, are expected and do not necessarily mean that the PLL is out of lock or that the data needs to be reframed. Too many errors in too short a time indicates that the PLL has lost lock and reframing is necessary. The benchmark chosen in this controller is 16 errors occurring in a period of 64 bytes. If the controller counts RVS asserted 16 times during a 64-byte period, it will assume the PLL has lost lock and will assert RF to the receiver to force it to reframe.

The 16-out-of-64 benchmark is somewhat arbitrarily chosen, but it is justified by the fact that when the PLL is in lock, you would normally expect to see significantly fewer errors. The fact that 16 out of 64 is the criteria used does not mean that 15 out of 64, or 14 out of 64, etc., are acceptable error rates and that the PLL is not out of lock in these cases as well. But, it is fairly certain that if the PLL does go out of lock, you will get at least 16 errors in 64 byte-times, very quickly. Furthermore, there are counters inside the HOTLink Receiver that detect this same condition (16 errors in a 64-byte period) and when this detection occurs inside the CY7B933, it forces the PLL to re-lock onto the serial input data stream. Even if the PLL *is* out of lock, if fewer than 16 errors are detected in a 64-byte period, the PLL will not be forced to re-synchronize with the data stream and will stay out-of-lock until that condition is detected. Therefore, for consistency, the same criteria was selected for the reframe controller.

Additional Functionality of the Reframe Controller

The reframe controller itself interfaces to a higher-level controller that controls the entire receiver system. That higher-level controller can force the reframe controller to initiate framing in the CY7B933, regardless of any errors. There are two ways to do this. The first is with the DO_REFRAME signal, which the higher-level controller asserts when it wants the reframe controller to go through the same procedure it goes through to initiate framing when an out-of-lock condition occurs. If the reframe controller sees this signal asserted, it acts just like it had detected an out-of-lock condition. The oth-

er way the higher-level controller can force a reframe is by asserting its FORCE_RF output. This simply forces the reframe controller's RF output HIGH and does not cause the internal logic or state machine to change. The reframe controller's RF output will stay asserted as long as its FORCE_RF input remains asserted.

The higher-level controller will normally assert DO_REFRAME on power-up or when the transmitter source is switched on in order to find the initial byte-boundary, as described above. The FORCE_RF signal could be used for any reason depending on specific system requirements. The most likely reason to use it is to force multibyte framing. When the receiver does multibyte framing, instead of looking for a single byte-boundary-indicating character, the receiver looks to detect two of these special characters within any four-byte sequence. This is a more reliable way of finding the byte boundary, simply because it causes the framing circuitry to verify its first find with another one. This may be useful in particularly noisy environments. To cause the receiver to do multibyte framing, you must assert its RF input for 2048 consecutive cycles; this is something the reframe controller would not ordinarily do. The higher-level controller can cause this to happen by asserting FORCE_RF to the reframe controller for 2048 cycles, thus causing its RF output to be asserted for the same length of time.

The reframe controller also implements a basic handshake with the higher-level controller to make sure the two controllers' operations stay consistent after forced reframes. Whenever the higher-level controller uses the DO_REFRAME signal to force the reframe controller to initiate framing, it will keep that signal asserted until the reframe controller asserts RFDONE_HS. This signal from the reframe controller indicates that the receiver has finished its reframing. The higher-level controller will then assert RFDONE_ACK, which acknowledges receipt of RFDONE_HS, and both the reframe controller and the higher-level controller will return to the state they normally return to following a reframe.

In addition to the operations described above, the reframe controller also provides a decoding function. When the HOTLink Receiver detects a data error and asserts RVS, it also puts the code for the type of error on its eight data outputs, D7–D0. The reframe controller decodes these signals and asserts one of two outputs, UNDEF_CHAR or RDISP_ERR, depending on the exact type of error decoded. The two types of errors are an undefined-character error and a running-disparity error. A running-disparity error means that the character received had too many consecutive 1s or 0s to be a valid byte of data (the purpose of the eight-bit-to-ten-bit encoding mentioned earlier is to encode the data in such a way as to minimize the imbalance of 1s and 0s in the bit stream). If the reframe controller detects the code for a running-disparity error, it will assert the RDISP_ERR output. If the received character has the correct running disparity but is not a valid code for any character, then it is an undefined-character error, and the reframe controller will assert the UNDEF_CHAR output instead.

Design and Implementation

The out-of-lock detection, RF control, higher-level controller interface, and error-type decoding are implemented with a simple state machine, a few internal counters, and some decoding logic, and it is all fit into a 32-macrocell CY37032 CPLD (for more information on this CPLD, please refer to other application notes in the PLD section of the Cypress website (www.cypress.com) and to the CY37032 datasheet). The design was done in VHDL and compiled with Cypress's *Warp*™ PLD design tool. The receiver system, the reframe controller's interface, and the details of the design of the internal state machine, counters, and logic are described in detail in the rest of this section.

Receiver System

Figure 1 shows where this reframe controller fits into the overall system. The CY7B933 receiver connects (through a physical connector) to the actual transmission medium, which can be either twisted pair, coaxial cable, or fiberoptic cable. The reframe controller interfaces to the receiver, and it also interfaces to the higher-level system controller.

Controller Interface

The complete set of reframe controller inputs and outputs is shown in *Figure 2*, and their source or destination, polarity, and functionality are described below.

Inputs

RF_ENABLE. Overall enable. It comes from a higher-level controller. When asserted (HIGH), reframe controller is enabled. When deasserted, reframe controller is disabled and does not operate.

CLK. Clock signal to the reframe controller that comes from the recovered byte-rate-clock output, RCLK, of the CY7B933, and is also used in the rest of the system as the system clock.

RESET. Resets the state machine and the internal counters and status registers (HIGH = asserted).

RVS. Received Violation Symbol. It comes from RVS output of the CY7B933 (HIGH = asserted).

FORCE_RF. When asserted, this forces the RF output to also be asserted regardless of other conditions. It comes from a higher-level controller (HIGH = asserted).

DO_REFRAME. When asserted, it causes internal state machine to initiate framing in the receiver just as if it had detected an out-of-lock condition. It comes from higher-level controller (HIGH = asserted).

RFDONE_ACK. Handshake signal from the higher-level controller acknowledging that it received confirmation that the reframe controller completed the framing procedure. The handshake is only done when the framing was triggered by the DO_REFRAME signal, not by an out-of-lock condition (HIGH = asserted).

$\overline{\text{RDY}}$. Ready signal. It comes from the CY7B933 $\overline{\text{RDY}}$ output and indicates to the reframe controller that the receiver has completed the reframe operation (LOW = asserted).

D[7:0], SC/$\overline{\text{D}}$. Eight-bit data byte and control/data indicator bit from the CY7B933 receiver. The information on these lines can be decoded during a receive violation to determine the error type.

Outputs

RF. Reframe output. It goes to the RF input of the CY7B933 receiver and causes the HOTLink Receiver to begin a framing operation on the incoming data stream (HIGH = asserted).

RFDONE_HS. This is the handshake signal to the higher-level controller telling it that the reframe it requested with the

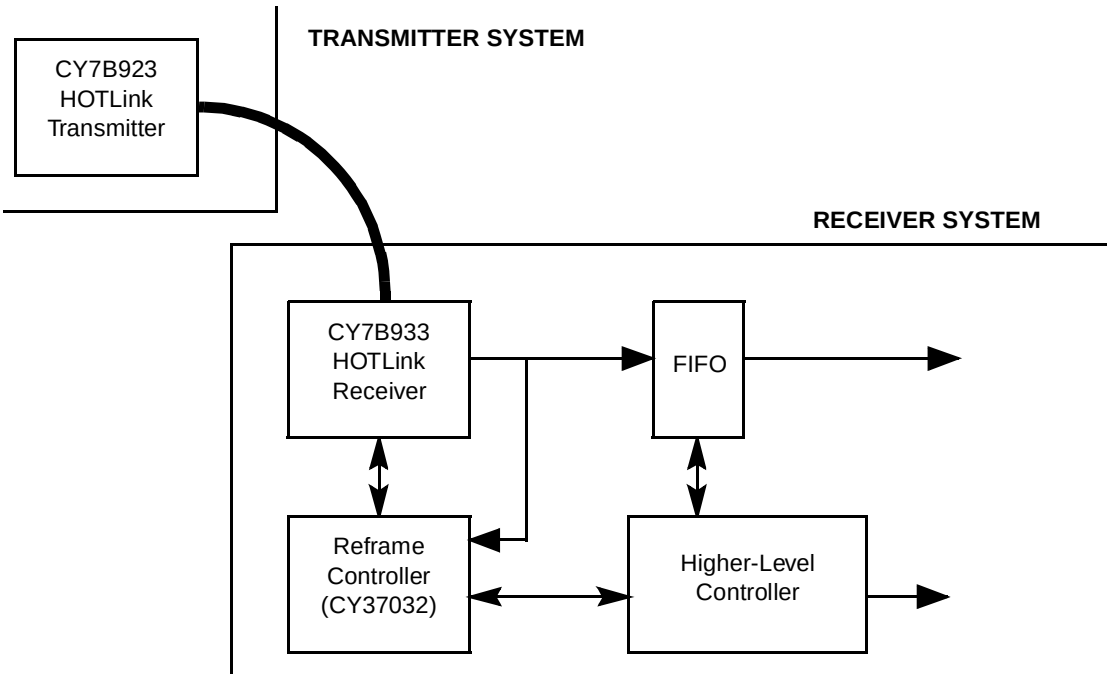

Figure 1. Block Diagram

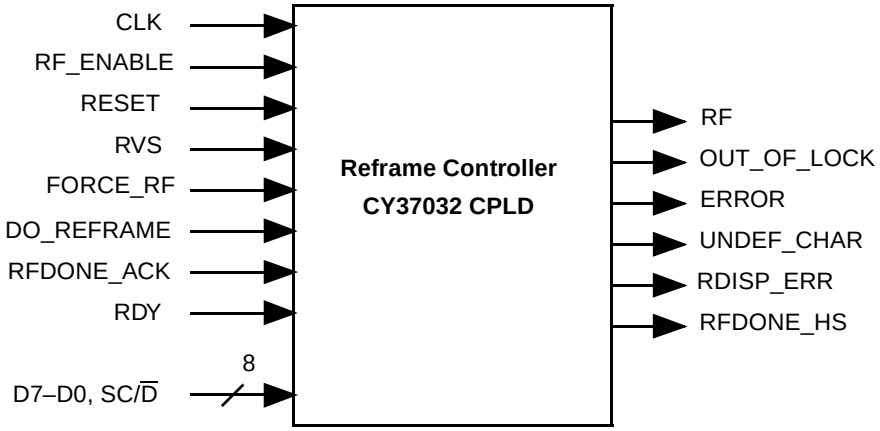

Figure 2. Controller Inputs and Outputs

DO_REFRAME signal has been completed (HIGH = asserted).

OUT_OF_LOCK. This signal indicates that the HOTLink Receiver's PLL has gone out of lock with the incoming serial bit stream. This is inferred by counting sixteen or more RVS assertions in a single 64-byte period. Once asserted, it remains asserted until the PLL regains lock and reframing has been accomplished (HIGH = asserted).

ERROR. When asserted (HIGH) it indicates to the higher-level controller that an error of some type (as indicated by the RVS signal from the receiver) has occurred.

UNDEF_CHAR. This is an undefined-character-error signal, one of two types of errors that can be decoded from the D7–D0, SC/$\overline{D}$ inputs during receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when RDISP_ERR is deasserted (HIGH = asserted).

RDISP_ERR. Running-disparity-error signal. This is the other of the two types of errors that can be decoded from the D7–D0, SC/$\overline{D}$ inputs during data-receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when UNDEF_CHAR is deasserted (HIGH = asserted.)

Counters

The primary function of the controller, which is to detect the out-of-lock condition by monitoring RVS and initiate a reframe when necessary, is implemented through the use of two counters. The VHDL for this function is shown in *Figure 3*. The

```
-- relevant VHDL code for counter functions

signal count2: bit_vector(0 to 1);          -- 2-bit counter
signal error_count: bit_vector(0 to 4);     -- 5-bit counter
signal rcvdbyts_count: bit_vector(0 to 6);   -- 7-bit counter

counters: process (CLK) begin

  if (clk'event and clk = '1') then

    if (reset = '1') then
       fb_out_of_lock    <= '0';
       rcvdbyts_count <= "0000000";
       error_count    <= "00000";
    elsif (error_count = "10000") then
       fb_out_of_lock    <= '1';
       rcvdbyts_count <= "0000000";
       error_count    <= "00000";
    elsif (rcvdbyts_count = "1000000") then
       rcvdbyts_count <= "0000000";
       error_count    <= "00000";
    else
       rcvdbyts_count  <= rcvdbyts_count + 1;
       if (RVS = '1') then
         error_count <= error_count + 1;
       end if;
    end if;

    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

    end if;

end process; --counters
```

Figure 3. VHDL for Counter Functions

first counter, rcvdbyts_count, is a seven-bit counter that counts the number of bytes received (0 to 64) and the second counter, error_count, is a five-bit counter that counts the number of times that RVS is asserted. If error_count reaches 16 before rcvdbyts_count reaches 64, then the out-of-lock condition will be declared. If rcvdbyts_count reaches 64 before error_count reaches 16, then fewer than 16 errors occurred in the given 64-byte window and out-of-lock is not declared. If rcvdbyts_count reaches 64 before error_count reaches 16, both rcvdbyts_count and error_count are set back to zero and a new 64-byte window begins. If the out-of-lock condition is declared (error_count = 16 and rcvdbyts_count $\leq$ 64), then the out-of-lock flip-flop is set to HIGH and a reframe operation is initiated. The out-of-lock flip-flop stays HIGH until the receiver successfully reframes. At that point, the out-of-lock flip-flop is set back to LOW and the search for the out-of-lock condition is started again.

State Machine

The state machine is described by the diagram in *Figure 4*, and the VHDL code that implements it is shown in *Figure 5*.

IDLE state

The normal, quiescent state of the state machine, and the state it enters upon reset, is IDLE. In this state, the RF output is deasserted and the state machine waits for either a DO_REFRAME input from the outside or for the counters to set the out-of-lock flip-flop. If neither of these conditions occur, the state machine simply stays in the IDLE state. Once either one of these conditions occurs, the state machine must initiate a reframe, so it will go to the START_REFRAME state.

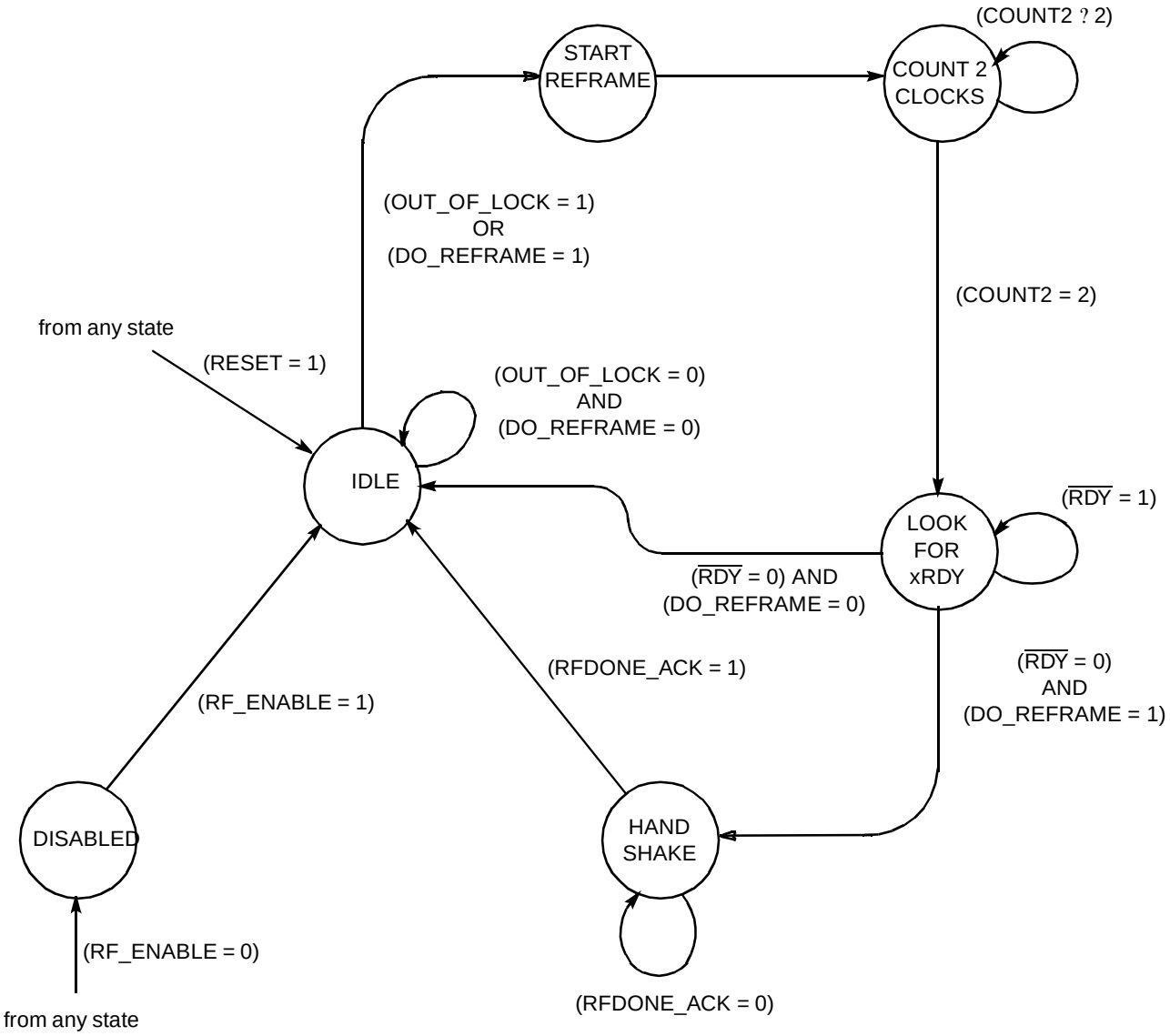

Figure 4. State Diagram

START_REFRAME state

In the START_REFRAME state, RF is asserted, and the state machine unconditionally transitions to the COUNT_2_CLOCKS state.

COUNT_2_CLOCKS state

The COUNT_2_CLOCKS state enables a two-bit counter to start counting incoming clock cycles. After two clock cycles have been counted, the state machine transitions to the LOOK_FOR_xRDY state. Two clock cycles must be counted before looking for the $\overline{RDY}$ signal from the outside because a total of three clocks must pass after RF is asserted until the value of $\overline{RDY}$ can be guaranteed valid (see the "HOTLink CY7B933 $\overline{RDY}$ Pin Description" application note for more details on this). One clock cycle passed during the START_REFRAME state, so the COUNT_2_CLOCKS state is used to count two more clock cycles to get to the requirement of three. RF is asserted throughout this state.

LOOK_FOR_xRDY state

On the fourth clock cycle from the start of RF, the value of $\overline{RDY}$ is guaranteed to be valid and the state machine, in the LOOK_FOR_xRDY state, continues to assert RF and waits until the HOTLink Receiver asserts $\overline{RDY}$. Once the receiver asserts RDY, it has successfully reframed and is ready to resume normal receiver operation. Thus, once an asserted $\overline{RDY}$ is detected in the LOOK_FOR_xRDY state, the state machine exits that state and goes back to the IDLE state. If the reframe was started by an out-of-lock detection, the transition back to the IDLE state is immediate; if the reframe was started by the DO_REFRAME input, then the state machine goes to the HANDSHAKE state first.

```
-- Relevant VHDL code for state machine

subtype StateType is bit_vector(0 to 2);      -- State Type
constant       DISABLED: StateType := b“111"; -- State Defns.
constant           IDLE: StateType := b“000";
constant  START_REFRAME: StateType := b“001";
constant COUNT_2_CLOCKS: StateType := b“010";
constant  LOOK_FOR_xRDY: StateType := b“011";
constant      HANDSHAKE: StateType := b“100";
signal current_state, next_state : StateType; --State declaration

-- State Machine Description

if (RESET = ’1’) then
  next_state <= IDLE;
elsif (RF_ENABLE = ’0’) then
  next_state <= DISABLED;
else
  case current_state is

  when IDLE =>
    if (fb_OUT_OF_LOCK = ’1’) or (DO_REFRAME = ’1’) then
      next_state <= START_REFRAME;
    else
      next_state <= current_state;
    end if;

  when START_REFRAME =>
    next_state <= Count_2_Clocks;

  when COUNT_2_CLOCKS =>
    if (count2 = “10") then
      next_state <= LOOK_FOR_xRDY;
    else
      next_state <= current_state;
    end if;

  when LOOK_FOR_xRDY =>
    if (xRDY = ’0’) and (DO_REFRAME = ’1’) then
      next_state <= HANDSHAKE;
    elsif (xRDY = ’0’) and (DO_REFRAME = ’0’) then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;

  when HANDSHAKE =>
    if (RFDONE_ACK = ’1’) then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;
```

Figure 5. VDHL Code for State Machine

```
  when DISABLED =>
    if (RF_ENABLE = '0') then
      next_state <= current_state;
    else
      next_state <= IDLE;
    end if;

  end case;
end if;

if (clk'event and clk = '1') then
    current_state <= next_state;
end if;
```

Figure 5. VDHL Code for State Machine (continued)

HANDSHAKE state

The HANDSHAKE state is used to make sure the reframe controller and the higher-level controller are consistent with each other. The only way this state will ever be entered is if the higher-level controller initiated a reframe by asserting DO_REFRAME to the reframe controller. Once that reframe has been completed by the receiver, the reframe controller communicates this to the higher-level controller by asserting RFDONE_HS. Once the higher-level controller acknowledges this assertion and is ready to proceed with normal receiving operation, it will assert RFDONE_ACK as confirmation to the reframe controller. It will simultaneously deassert DO_REFRAME so that once the state machine goes back to the IDLE state, that input is deasserted and does not erroneously cause another immediate pass into the reframe procedure. Once the state machine detects the RFDONE_ACK assertion, it exits the HANDSHAKE state and returns to the IDLE state. The RF operation is deasserted throughout the HANDSHAKE state.

DISABLED state

There is one more state, the DISABLED state, which is treated separately. As long as RF_ENABLE, the overall controller enable, is asserted, the state machine will never enter this state. If RF_ENABLE gets deasserted, the state machine will transition to the DISABLED state no matter what state it was in, and it will stay there until RF_ENABLE is once again asserted. Once RF_ENABLE is reasserted, the state machine goes to the IDLE state and resumes normal operation.

It was mentioned previously that the out-of-lock flip-flop is set when the out-of-lock condition is detected, and it stays set until the reframe has been completed. The exact time when the OUT_OF_LOCK flip-flop gets cleared is at the rising clock edge when the state machine exits the LOOK_FOR_xRDY state. This is because that is the exact point where the receiver has signalled to the controller, with $\overline{\text{RDY}}$, that it has successfully competed the reframe.

Decode Logic

The error-decode logic is very straightforward, and the VHDL code for it is shown in *Figure 6*. The ERROR output is a reg-

```
-- relevant VHDL code for Decode Logic

if (clk'event and clk = '1') then

  if (RVS = '1') then
    ERROR <= '1';
    if (D = x"E4" or D = x"E2" or D = x"E1") then
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '1';
    else
      UNDEF_CHAR <= '1';
      RDISP_ERR  <= '0';
    end if;
  else
    ERROR <= '0';
    UNDEF_CHAR <= '0';
    RDISP_ERR  <= '0';
  end if;

end if;
```

Figure 6. VHDL Code for Decode Logic

istered version of the RVS input. The RDISP_ERR and UNDEF_CHAR outputs are decoded from the D7–D0, SC/$\overline{D}$ inputs. These outputs are also registered.

When the receiver asserts RVS, it will also put a code for the error type on its eight data outputs. If this code is E4, E2, or E1 (hex), it indicates the error is a running-disparity error, (explained earlier), and the RDISP_ERR output is asserted. If it is any other hex code, the receiver has detected some kind of illegal or undefined character, and the UNDEF_CHAR output will be asserted instead. These outputs are mutually exclusive: if one is asserted, the other must be deasserted. However, it is only meaningful to decode the data outputs when an error condition is detected, so the ERROR signal must be examined by the higher-level controller as well. If ERROR is not asserted, the output from RDISP_ERR and UNDEF_CHAR is no longer valid.

VHDL, CY37032 Utilization, and CY37032 Speed Considerations

The complete VHDL description for this design is given in Appendix A. The full source code consists of the fragments shown throughout this application note along with the other code necessary to mesh it together, (process declarations, signal declarations, and package-entity declarations). As the fragments and complete source file show, VHDL is a very simple, efficient way for describing PLD designs. For example, the counter functions are simply bit vectors that are used in the manner: COUNT <= COUNT + 1. Upper limits for the counters, clearing functions, resets, and presets are all implemented with a few simple IF-THEN-ELSE statements. The entire state machine is implemented with a CASE statement and IF-THEN-ELSE statements that have a straightforward, natural, one-to-one correspondence with the bubble diagram shown in *Figure 4*. The entire set of decode logic is implemented in a single IF-THEN-ELSE clause. Furthermore, the VHDL code provided is easy to understand and can be very easily modified. For example, it can be modified to interface to different higher-level-controller interfaces than the one assumed in this application note, or it could be incorporated into the higher-level controller design, with that design consisting of other VHDL code and implemented in a larger FLASH370™ CPLD or even a gate array.

This design used all 32 of the CY37032's macrocells and 37 of its 37 I/O and input pins. It could have used fewer pins if necessary, by making the various counters be internal counters only. The outputs of the counters were brought out to output pins in this example, however, for easier simulation and debugging. The speed of the CY37032 ranges from 125 MHz (with a 10-ns combinatorial propagation delay and a 6.5-ns clock-to-output time) to 222 MHz (with a 5-ns combinatorial propagation delay and a blazing 4-ns clock-to-output time). For this application, the maximum byte-rate clock of the CY7B933 is 33 MHz, and this and the corresponding set-up and hold times on the CY7B933 make the CY37032-125 quite sufficient. The higher-level controller may have tighter timing requirements, but there is plenty of speed to be gained by going to the faster speed bins of the CY37032. The design can, thus, easily meet much faster system timing requirements.

Conclusion

The serial data received by the CY7B933 needs to be framed, i.e., aligned to the proper byte boundaries. This must always be done when the serial communication first begins, and it must always be redone if the PLL loses lock on the incoming serial bit stream. This application note described a controller that will manage this operation and provided some guidelines for determining when the periodic reframing is necessary. It assumed a particular interface to a higher-level controller, but the design was done in VHDL, which is provided in the appendix, to make it very easily modifiable and adaptable to any other specific interface. The controller itself is implemented in a CY37032 32-macrocell CPLD, which had sufficient resources and routability to implement this fairly substantial function. It was able to do this exceeding system speed requirements even in its slowest speed bin.

Appendix A. VHDL Description

```
-- Application Note
-- Using a CY37032 as a HOTLink Reframe Controller
-- Cypress Semiconductor

entity CONTROLLER is port (
    CLK, RVS, RESET, xRDY, DO_REFRAME, FORCE_RFOUT, RFDONE_ACK,
      RF_ENABLE : in bit;
    D           : in bit_vector(0 to 7);
    curr_st     : out bit_vector (0 to 2);
    rb_cntr     : out bit_vector (0 to 6);
    err_cntr    : out bit_vector (0 to 4);
    RF, RFDONE_HS, OUT_OF_LOCK, UNDEF_CHAR, RDISP_ERR, ERROR : out bit
);
end CONTROLLER;

architecture CNTRL933 of CONTROLLER is

    subtype StateType is bit_vector(0 to 2);        -- State Type
    constant       DISABLED: StateType := b“111";   -- State Definitions
    constant           IDLE: StateType := b“000";
    constant  START_REFRAME: StateType := b“001";
    constant COUNT_2_CLOCKS: StateType := b“010";
    constant  LOOK_FOR_xRDY: StateType := b“011";
    constant      HANDSHAKE: StateType := b“100";

    signal current_state, next_state : StateType;
    signal fb_OUT_OF_LOCK : bit;

    signal count2: bit_vector(0 to 1);          -- 2-bit counter
    signal error_count: bit_vector(0 to 4);     -- 5-bit counter
    signal rcvdbyts_count: bit_vector(0 to 6);  -- 7-bit counter

begin

counters: process (CLK) begin

  if (clk’event and clk = ’1’) then

    if (reset = ’1’) then
      fb_out_of_lock    <= ’0’;
      rcvdbyts_count <= “0000000";
      error_count    <= “00000";
    elsif (error_count = “10000") then
      fb_out_of_lock    <= ’1’;
      rcvdbyts_count <= “0000000";
      error_count    <= “00000";
    elsif (rcvdbyts_count = “1000000") then
      rcvdbyts_count <= “0000000";
      error_count    <= “00000";
    else
      rcvdbyts_count  <= rcvdbyts_count + 1;
      if (RVS = ’1’) then
        error_count <= error_count + 1;
      end if;
    end if;
```

Appendix A. VHDL Description (continued)

```
    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

  end if;

end process; --counters

next_st_comb:  process (fb_OUT_OF_LOCK, DO_REFRAME, COUNT2, xRDY,
                        RFDONE_ACK, RESET, RF_ENABLE, current_state) begin

  if (RESET = '1') then
    next_state <= IDLE;
  elsif (RF_ENABLE = '0') then
    next_state <= DISABLED;
  else
    case current_state is

    when IDLE =>

      if (fb_OUT_OF_LOCK = '1') or (DO_REFRAME = '1') then
        next_state <= START_REFRAME;
      else
        next_state <= current_state;
      end if;

    when START_REFRAME =>

      next_state <= Count_2_Clocks;

    when COUNT_2_CLOCKS =>

      if (count2 = "11") then
        next_state <= LOOK_FOR_xRDY;
      else
        next_state <= current_state;
      end if;

    when LOOK_FOR_xRDY =>

      if (xRDY = '0') and (DO_REFRAME = '1') then
        next_state <= HANDSHAKE;
      elsif (xRDY = '0') and (DO_REFRAME = '0') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;
```

Appendix A. VHDL Description (continued)

```
    when HANDSHAKE =>

      if (RFDONE_ACK = '1') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;

    when DISABLED =>

      if (RF_ENABLE = '0') then
        next_state <= current_state;
      else
        next_state <= IDLE;
      end if;

    end case;
  end if;
end process; --next_st_comb

outp_comb:  process (current_state, FORCE_RFOUT) begin

  if (FORCE_RFOUT = '1') then
    RF <= '1';
  else
    case current_state is

    when IDLE =>
      RF        <=  '0';
      RFDONE_HS <=  '0';

    when START_REFRAME =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when COUNT_2_CLOCKS =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when LOOK_FOR_xRDY =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when HANDSHAKE =>
      RF        <=  '0';
      RFDONE_HS <=  '1';


    end case;
  end if;
end process; --outp_comb
```

Appendix A. VHDL Description (continued)

```
seq_assgnmnt:  process (clk) begin

  if (clk'event and clk = '1') then

    current_state <= next_state;

    if (RVS = '1') then
      ERROR <= '1';
      if (D = x"E4" or D = x"E2" or D = x"E1") then
        UNDEF_CHAR <= '0';
        RDISP_ERR  <= '1';
      else
        UNDEF_CHAR <= '1';
        RDISP_ERR  <= '0';
      end if;
    else
      ERROR <= '0';
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '0';
    end if;

  end if;

end process; --seq_assgnmnt

-- concurrent assignment statements
-- outputs and local feedback signals made the same

curr_st     <=  current_state;
rb_cntr     <=  rcvdbyts_count;
err_cntr    <=  error_count;
OUT_OF_LOCK <=  fb_out_of_lock;

end CNTRL933;  -- end architecture
```

HOTLink, *Warp*, and FLASH370 are trademarks of Cypress Semiconductor Corporation.