TECHNICAL MEMORANDUM

Lafayette College

Department of Electrical and Computer Engineering

Title: Manchester Code Transmitter
Author(s): G. Flynn and R. Birru
Date: October 2nd, 2016

Abstract

This memo describes the design and implementation of an Asynchronous Serial Receiver on a Nexys 4 FPGA board. The design was parameterized to receive data at varying baud-rates as necessary. A functional design was implemented to be fully operational.

Introduction

The goal of this memo is to describe one approach to the design of a Serial Receiver. The receiver was to be designed to run off the 100MHz system clock on a Nexys 4 FPGA board. Spurious starts were to be detected and ignored. Framing errors were to be detected and an alert signal be activated. The reception baud-rate was to be parametrized. And finally the design was to be put under heavy scrutiny with self-checking test benches as well as the usual testing mechanisms.

The Design

After the requirements for the design were provided, we threw some ideas back and forth as to how the Receiver was to be designed and we came up with some ideas before settling onto our final design was decision.

The first idea we were thinking of employing was a design centered around a Shift Register that clocked 10 bits of our serial input line and checked for a start bit. If found, it would then check for the presence of a stop bit to detect a framing error if one is present. If no errors are found during reception, the middle 8-bits are loaded onto the data output.

The next design idea we were thinking about employing was a nested FSM. The high level FSM would decide the specific mode the receiver would operate in. Idle, Spurious Start Check, Preamble, Receiving, and EOF check could be the different modes the receiver could operate in

at any given time. This design was particularly interesting as it could be easily expanded for receivers with different reception protocol.

The last idea we came up with was a design similar to our previous designs; one centered around one multifunctional FSM that would control other smaller blocks in our design in a sort of brain-body relationship.

Selected Design

After careful analysis of all design options, we settled on the last of the design ideas listed above. We proceeded to flesh it out and realize it into an operational design. Below is a high level and detailed description of of the operation of our design.

The design's functions were divided into several blocks to provide seamless operation and modularity for the future, the most important block being the FSM. The many blocks work in tandem to provide the required functionality. The following blocks are included in the design. A block diagram of the entire design is also attached to the Appendix A-1 section of this document for further reference.

1. Input Data Synchronizer

This relatively simple block has a relatively simple function. Since the incoming serial data is asynchronous, it syncs it up to our system clock.

- It is named 'rxd_synchronizer' on the the block diagram.

2. Bit Counter

The Bit Counter is used to count how many of our expected 8 data bit we have received. It has a ‘bit_counter_rst’ input which is used to reset it; this is handled by our state machine. It has a ‘bit_count’ output that’s a count of the bits. The FSM uses this count to know when all data bits have been received.

3. Delay Timer

The Delay Timer block is a modified version of the clock divider module we employ. It has two functions. This first is to count to half our baud and throw up a flag. This is employed by our FSM to find the start bit. The second function is a similar function but a flag is thrown up after a whole baudd has been counted. This will be used by the FSM when sampling our data bits.

The Delay Timer has two outputs ‘half_timer’ and ‘full_timer’ which are the half and full bit width identifier flags already mentioned. These are sent to the FSM. The ‘full_timer’ is additionally sent to the Bit Counter block. There is a single ‘delay_timer_rst’ output for resetting the block when necessary.

4. FERR Counter

This block handles how long we throw up our framing error flag. Our design decision for handling framing errors is waiting for 10 clock cycles and start receiving again. In accordance with this, our FERR Counter block counts the number of cycles we have been waiting and relays the information to the FSM.

There is a single input ‘ferr_counter_rst’ to reset the count when necessary and a ‘ferr_delay_count’ which is the actual count.

5. Start Pulser

The Start Pulser’s function is to find the start bit so that the receiver can begin receiving. It samples the incoming serial data 16 times per baud and monitors for the idle high signal to go low (a start bit) and when it does, its only output, ‘start_check’ is set high to signify to our FSM that the start bit found wasn't spurious.

6. Data Blocks

There are two data blocks in our design. Both are 8-bit registers. Our data bits are first stored in the first of these blocks, the Temporary Data Block, until the framing error is guaranteed to not have been found. If one is found, the data is thrown out. The data is kept and moved to the Final Data Block if no framing error is found.

- The Temporary Data Block is labeled 'temp_data' on the block diagram. The Final Data Block is labeled 'data'.

7. Framing Error Block

The Framing Error Block has a simple enough function. Upon instruction from the FSM it sets the ferr output to high if a framing error is detected and low when correct reception resumes.

8. FSM

The FSM is the brain of our design. It is in charge of managing all other blocks in our design. The basic functionality of the FSM is detailed below.

- Initially the FSM monitors the ‘start_check’ signal from the Start Pulser module and when it’s flagged high, uses the half timer from the Delay TImer block to verify whether we have a spurious start or not. If the ‘rxd’ has gone high and we have found a spurious start bit, it is ignored.
- If a valid start bit is found the FSM proceeds to the next step. It sets the ‘rdy’ signal low and uses the full timer signal from the Delay Timer and starts receiving data bits using the Bit Counter to keep track of how many bits have been received. The received data bits are stored in the Temporary Data Block.
- The last step in the receiving process is to check for a valid stop bit. If one is found, the receiver gets ready for the next frame by setting ‘rdy’ and some other signals. These are shown in detail in the ASM State diagram provided.
- If a stop bit isn’t seen, the FSM sets the ‘ferr’ output high and alerts for a framing error. It then uses the full timer once again to wait 10 cycles to make sure nothing is getting sent and then re-synchronizes. Once it has verified that the rxd is an idle high, it once again readies the receiver for reception.

A high level ASM state diagram is provided for our FSM in Appendix A-2.

Design Verification

A part of the requirements for the design was intensive verification on both the hardware and simulation fronts. To this end we set up an intensive test plan to cover for both. Hardware was tested through RealTerm on a PC and our previous verified design for a Serial Transmitter.

Diagrams of the hardware test setups are included in the Appendix section of the document.

- Appendix A-3: RealTerm hardware test setup
- Appendix A-4: Transmitter to Receiver hardware test setup

Simulation Tests

CODE	TEST	TEST SETUP	PASS CRITERIA	PASS EVIDENCE
1	Baud rate bounds	Send 1 byte, formatted correctly at 9300 baud and at 9900 baud	All data is received correctly and no ferr at a baud rate of 9600 +/- 300 baud.	Verification Test Log: Test 1 Appendix C-1
2	Start bit width	Send 0x00 formatted correctly. Cause a glitch for 6.51us *[1:16] (1/16*baud rate) Then hold the line high	No data should be read if the glitch is less than 40us (6/16 baud rate). Data should be read if the glitch is more than 60us (9/16 baud rate)	Verification Test Log: Test 2 Appendix C-2
3.1	Frame error set single byte	Send 1 byte with the EOF low then idle	Ferr comes high Data does not change	Verification Test Log: Test 3
3.2	Frame error reset	Trigger ferr. Wait for ready. Send correctly formatted byte	Ferr drops with falling edge of rxd	Verification Test Log: Test 3
3.3	Frame error, multibyte	Trigger ferr. Wait for ready. Send malformatted byte	Ferr drops with falling edge of rxd. Ferr comes high at EOF	Verification Test Log: Test 3
3.4	Frame error ignore additional byte	Clear ferr Send five bytes with the third malformatted.	Check first two bytes read correctly. Check ferr trigger on third byte. Check data is still byte 2 not 3-5	Verification Test Log: Test 3
4.1	Ready on idle	Leave rxd high for 20 baud	Verify rdy stays high	Verification Test Log: Test 4
4.2	Ready on good byte	Send correctly formatted byte	Verify rdy high Verify rdy drops low within 65us of start of start bit (10/16 *baud	Verification Test Log: Test 4

			rate) Verify rdy comes high within 1.1ms (10 baud)	
4.3	Ready on bad byte	Send malformatted byte.	Verify rdy high Verify rdy drops low within 65us of start of start bit (10/16 *baud rate) Verify rdy low after 1.1ms (10 baud due to DATA_BAD state) Verify rdy comes high within 2.2ms (20 baud)	Verification Test Log: Test 4
4.4	Ready on false start	Drive rxd low for 40us (longest glitch that should be ignored)	Verify rdy stays high	Verification Test Log: Test 4
4.5	Ready on multibyte	Send correctly formatted bytes	Verify rdy drops low with start bit Verify rdy pulses with middle of stop bit Verify rdy width is 1 baud (104us +/- 10us)	Verification Test Log: Test 4
5.1	Data update on good rx	Send correctly formatted byte	Verify data update at end of rxd	Verification Test Log: Test 5 Appendix C-3
5.2	Data update sequential	Send correctly formatted bytes	Verify data updated by every posedge of rdy Verify data doesn't update during rx	Verification Test Log: Test 5 Appendix C-3
5.3	Data not updated on frame error	Send malformatted byte	Verify data does not change	Verification Test Log: Test 5
5.4	Talk to tx	Attach tx to rx Send one byte Send multi bytes	Verify data matches old data	Verification Test Log: Test 5

Hardware Tests

CODE	TEST	TEST SETUP	PASS CRITERIA	PASS EVIDENCE
1.0	Baud rate bounds	Send 1 byte from realterm, formatted correctly at 9300 baud and at 9900 baud	All data is received correctly and no ferr at a baud rate of 9600 +/- 300 baud.	Data displayed on the 7-seg as expected. Appendix B-1
2.1	Framing error Option a	Send \x00 with a baud rate of less than 9300 baud. Suggest 4800	Ferr is detected	Ferr LED lights up and stays on. Appendix B-2
2.2	Framing error Option b	Get rxd on to JA[0]. Create a simple circuit that can be driven low with a button press/wire connection	Fferr is detected Ferr stays detected	Ferr LED lights up and stays on after button released. Appendix B-2
3.0	Framing error reset	Trigger ferr. Send correct formatted byte	Ferr is cleared	Ferr LED goes out Appendix B-3
4.0	Ready LED on idle	Program the board. Do nothing	Rdy is high	Rdy LED is on
5.1	Ready drops on data	Get rdy on to JA[2]. Send one good byte. Trigger on falling edge of rdy. Check time to rising edge of rdy	Low should be 937.5us +/- 10.4us (9 baud +/- 0.1 baud) (8 baud for the data, 0.5 baud for start bit, 0.5 for stop bit)	Appendix B-3
5.2	Ready pulses on multiple byte	Get rdy on to JA[2]. Send many good bytes with one stop bit. Trigger on falling edge of rdy. Check time to rising edge of rdy	High pulse between each bit should be 104 us +/- 10.4us (1 baud +/- 10%)	Visually inspect scope outputs
6.0	Verify 1 byte	Using RealTerm and inspect 7 seg display for values	Data on display = data transmitted	Visually inspect 7-seg display

7.0	False start	User RealTerm to send a glitch @250k	Ready stays high Data stays at old value	Visually inspect scope outputs

Conclusion

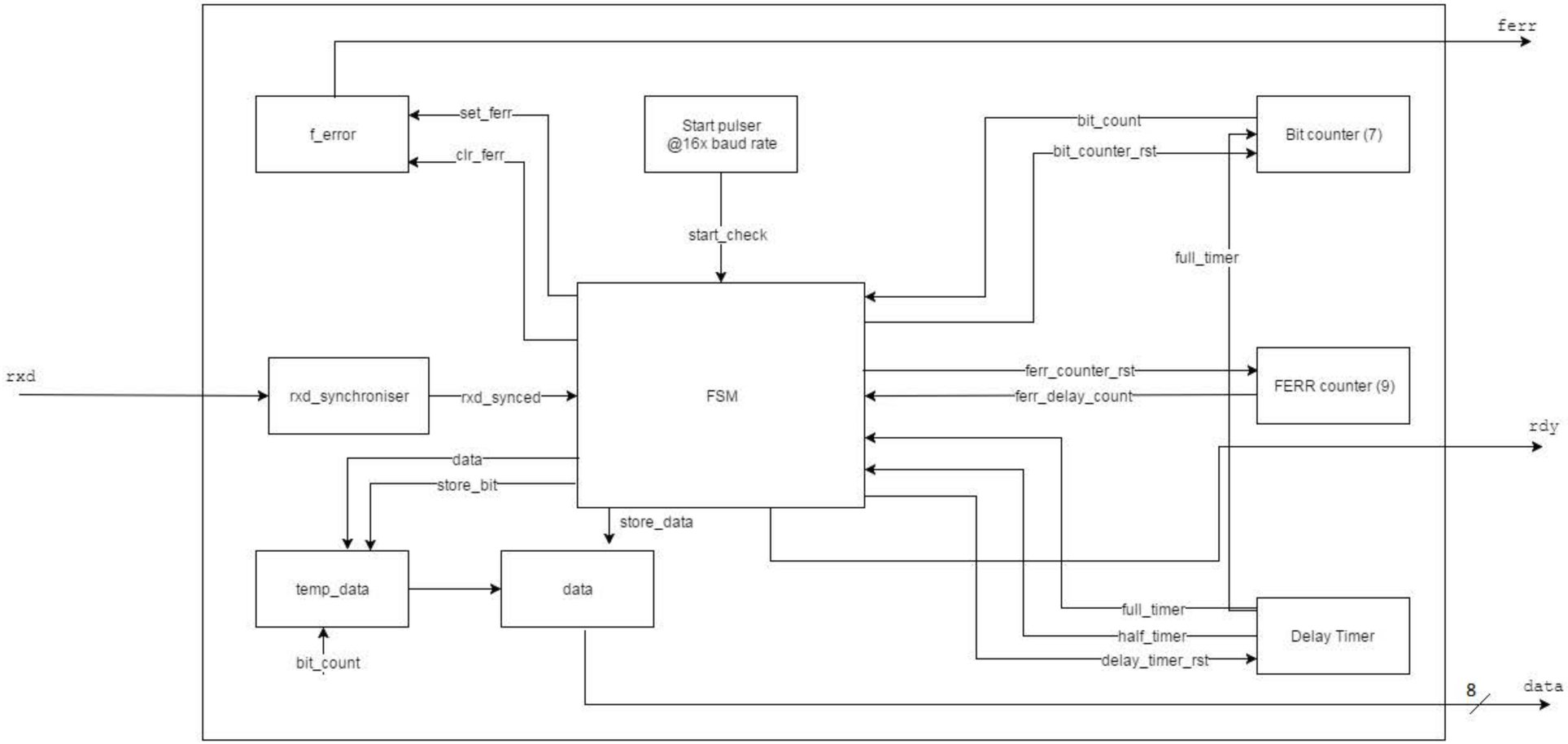
The design was implemented and hardware and passed all demonstration tests with flying colors. The implementation was highly modular and very easy to understand at a high level. We would readily suggest this approach to anyone looking to design an asynchronous serial receiver.

Appendix

A Diagrams
B Hardware Test Outputs
C Simulation Outputs

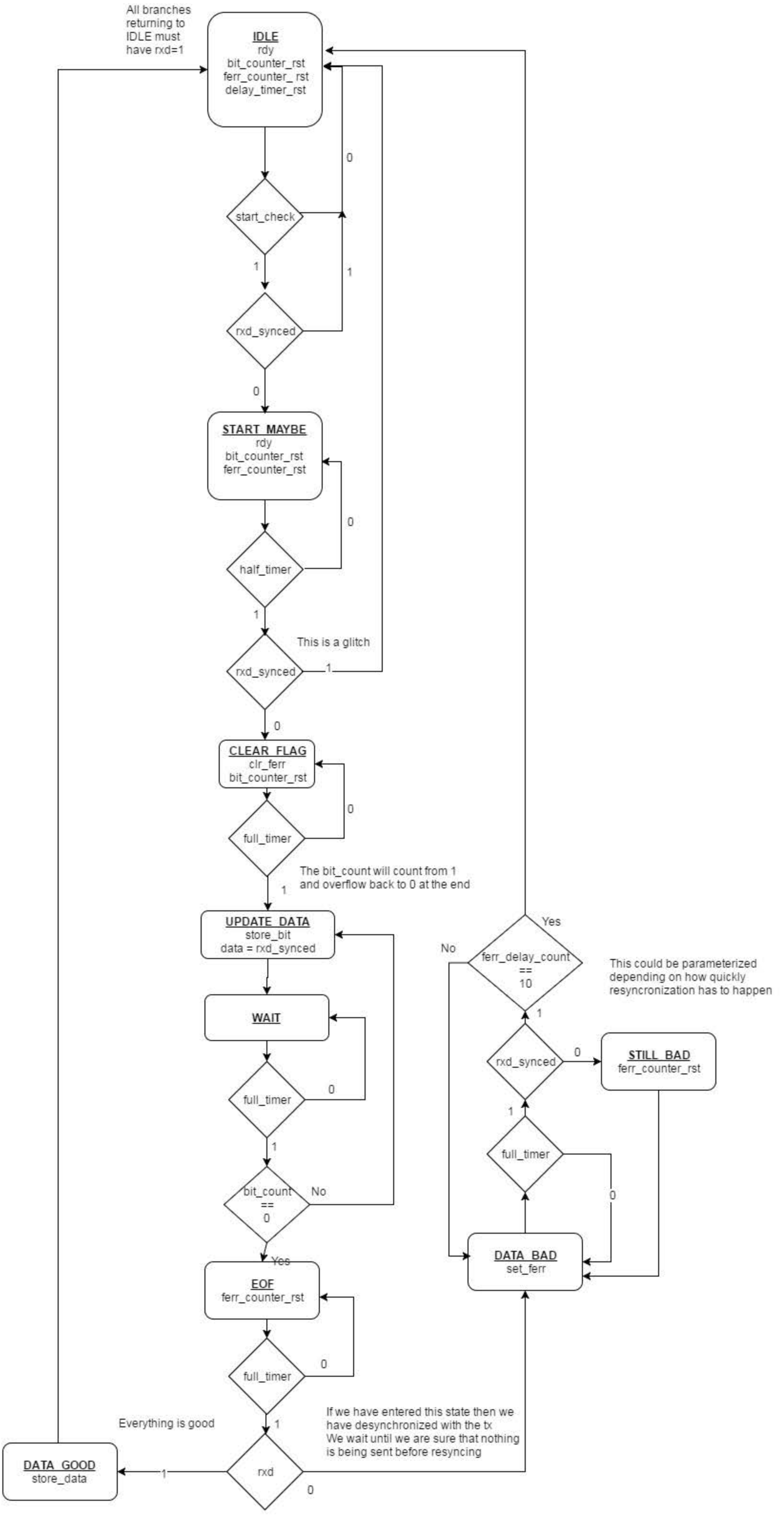
APPENDIX A

DESIGN BLOCK DIAGRAM



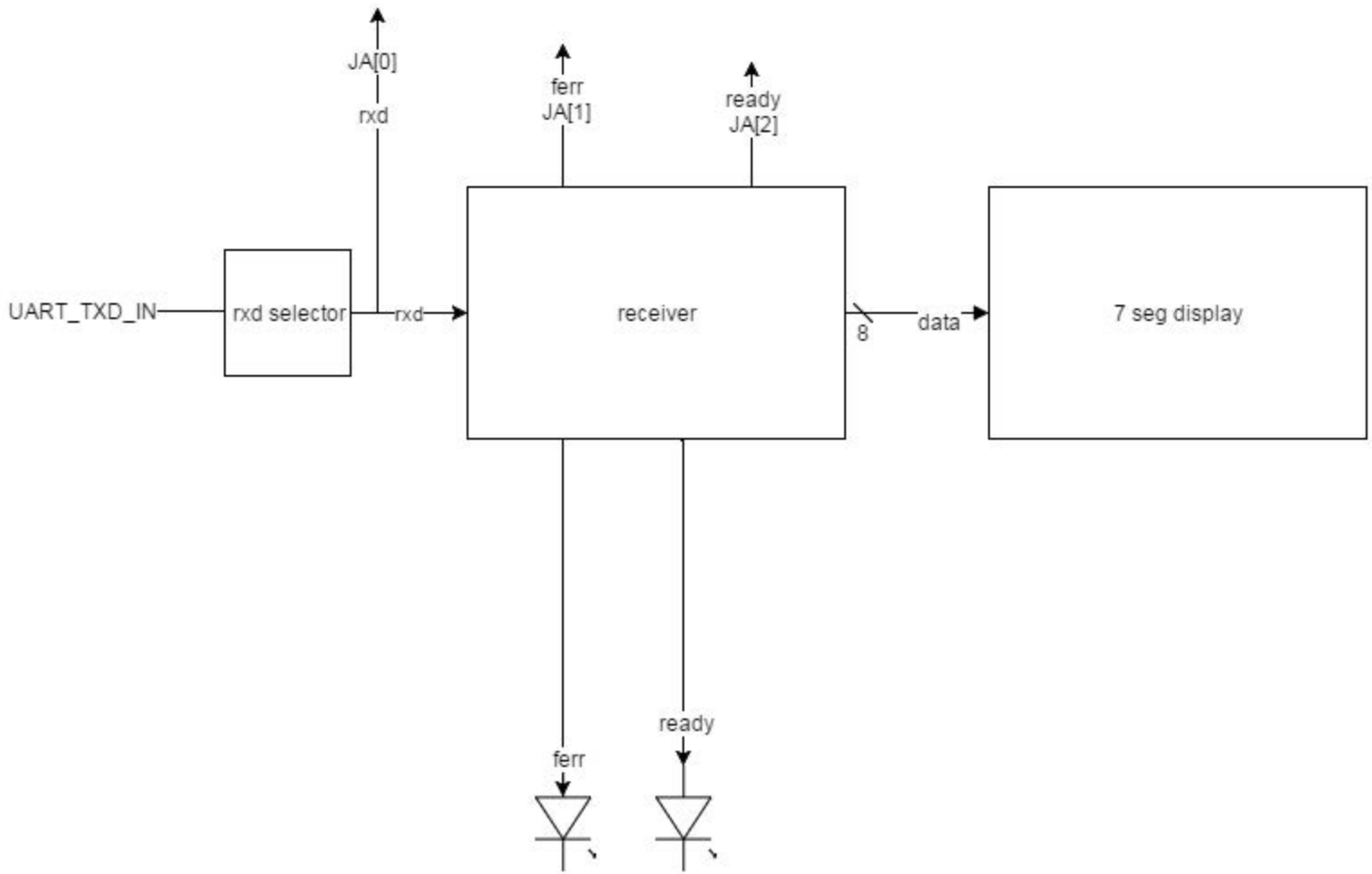
APPENDIX A-1

ASM State Diagram for FSM in design



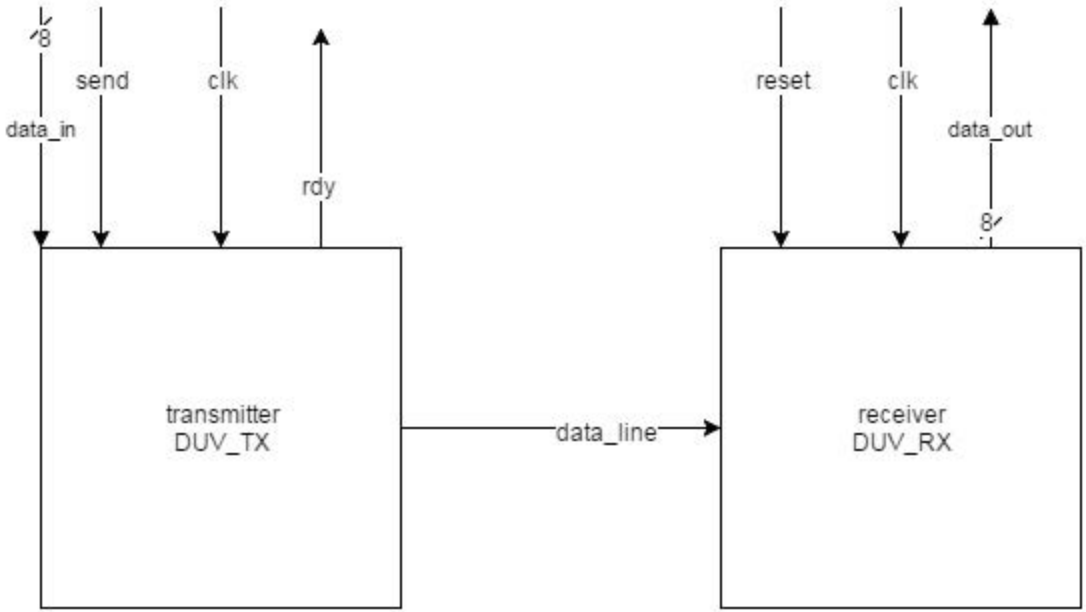
APPENDIX A-2

RealTerm Hardware Test Setup



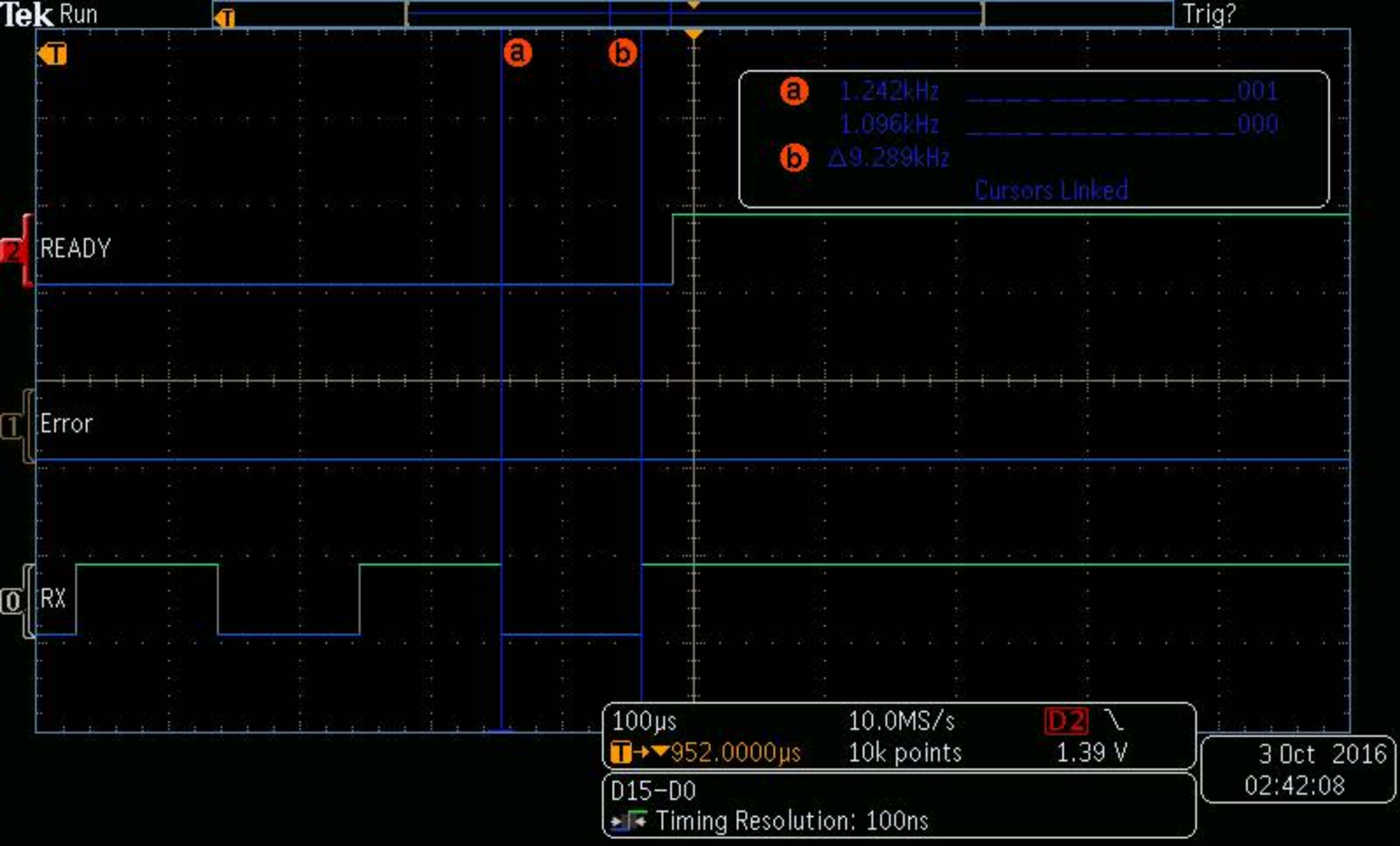
APPENDIX A-3

Transmitter to Receiver Hardware Test Setup

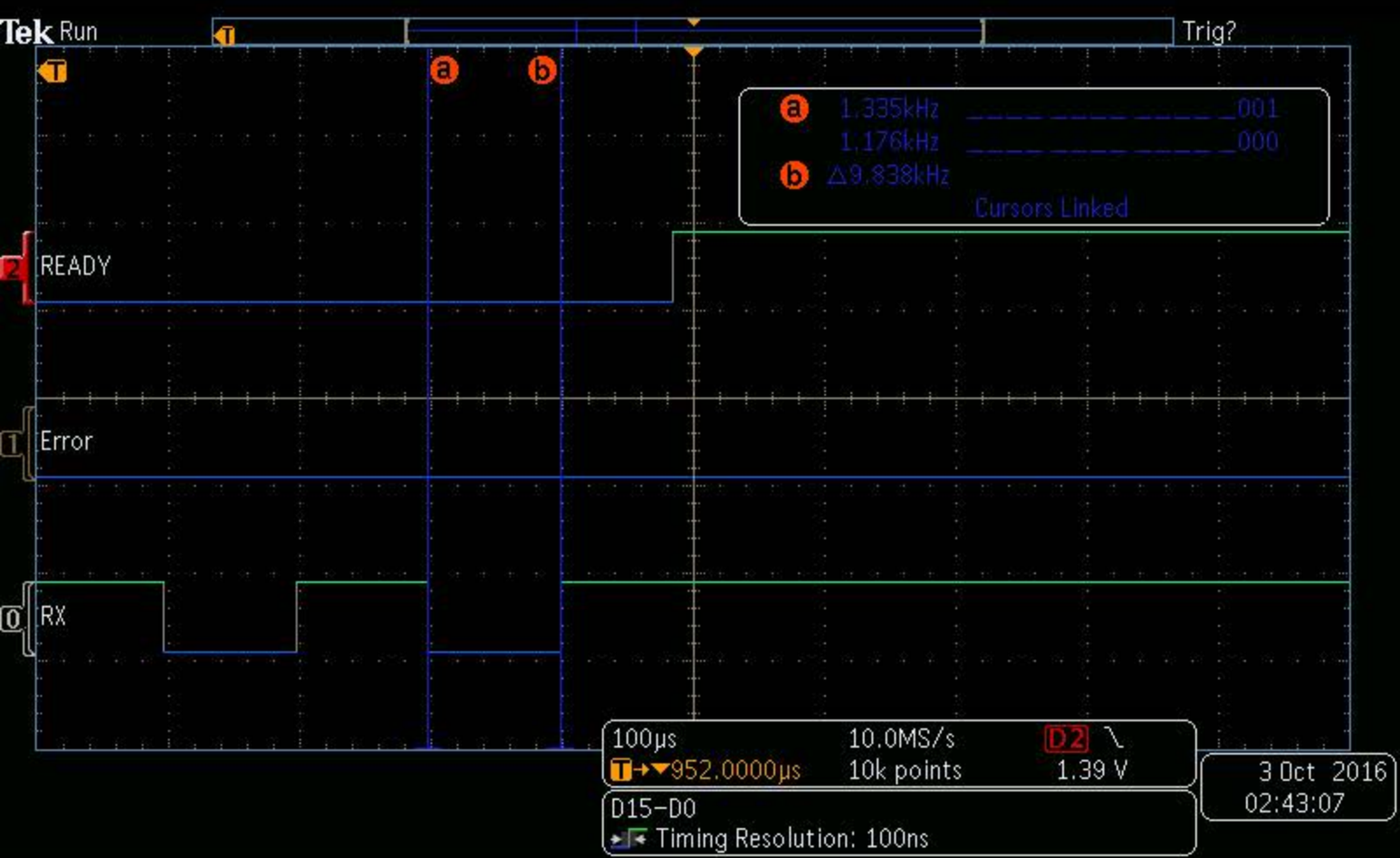


APPENDIX A-4

APPENDIX B

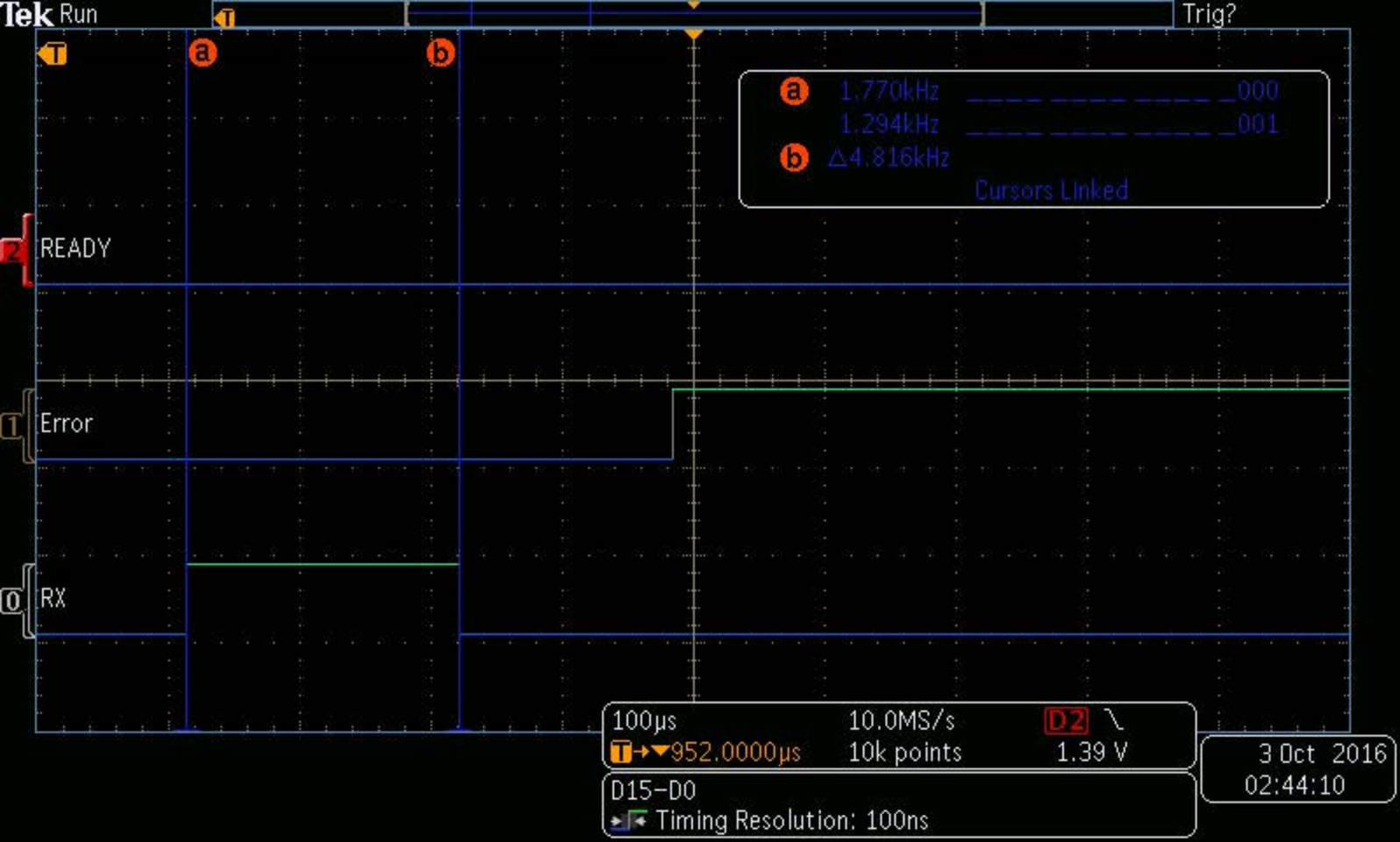


> Reciever operational at ~9300 MHz lower limit

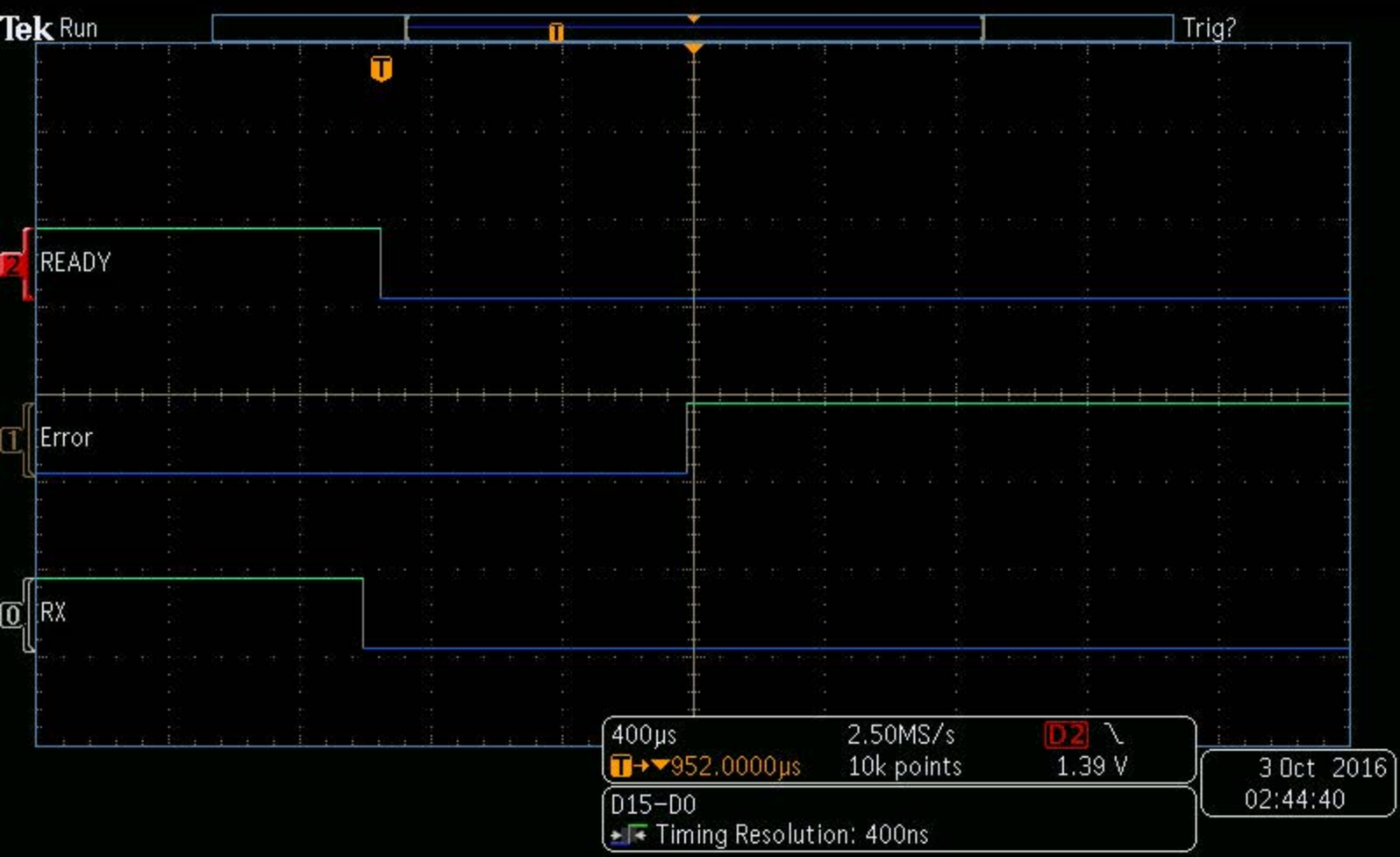


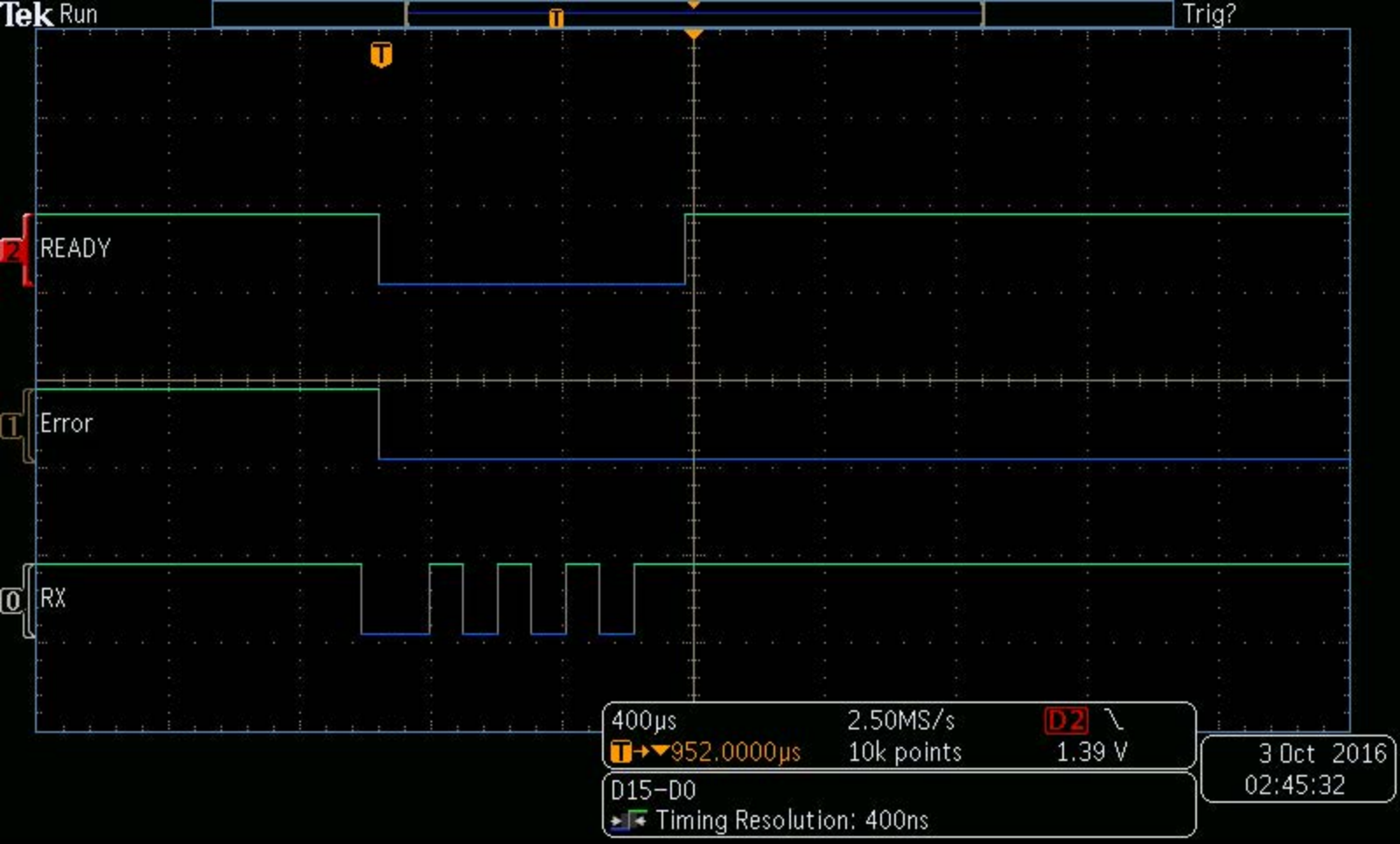
> Receiver operational at ~9900Mhz upper limit.

APPENDIX B-1



> Framing Error working as expected. Flag goes high when stop bit not found.

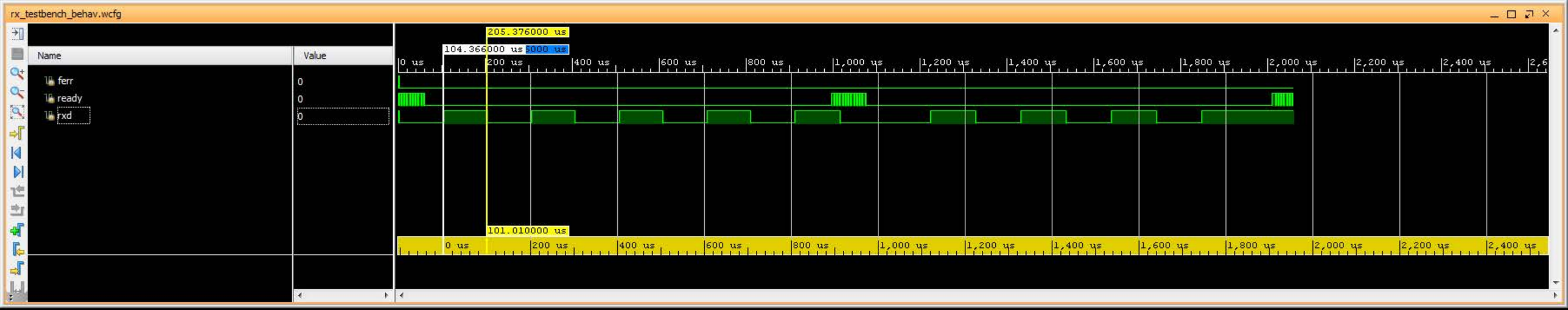




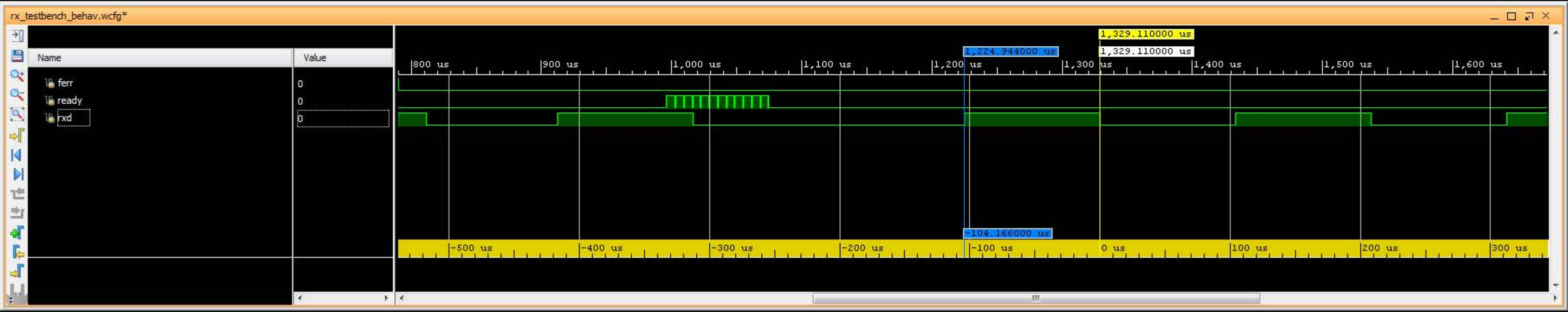
> error flag drops upon valid reception of new data. ready also drops when valid data beings being received.

APPENDIX B-3

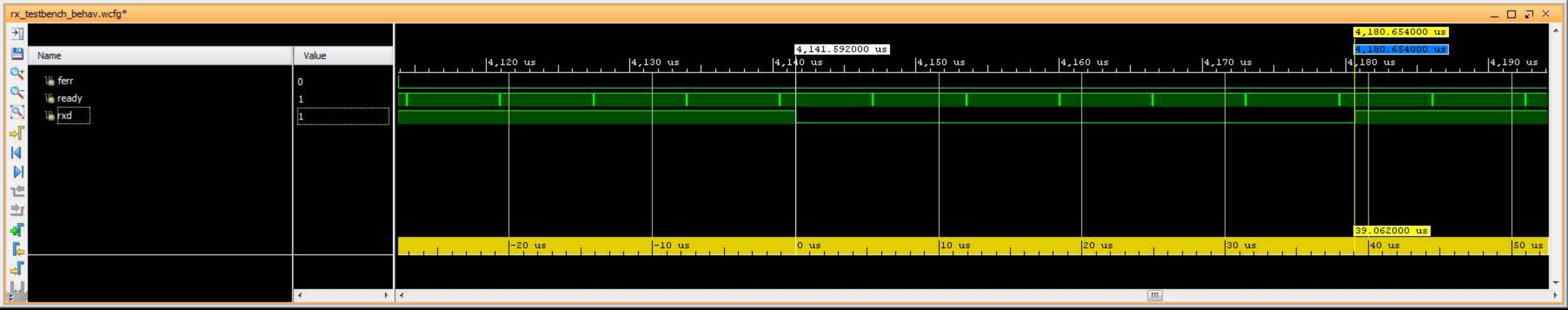
APPENDIX C



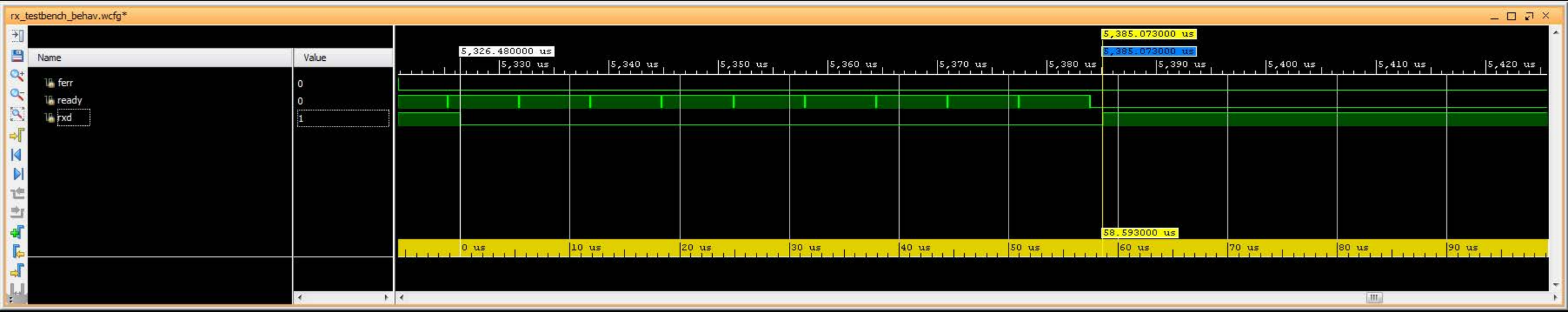
> Reciever functions properly at upper limit of 9600+300 = 9900 MHz baud rate.



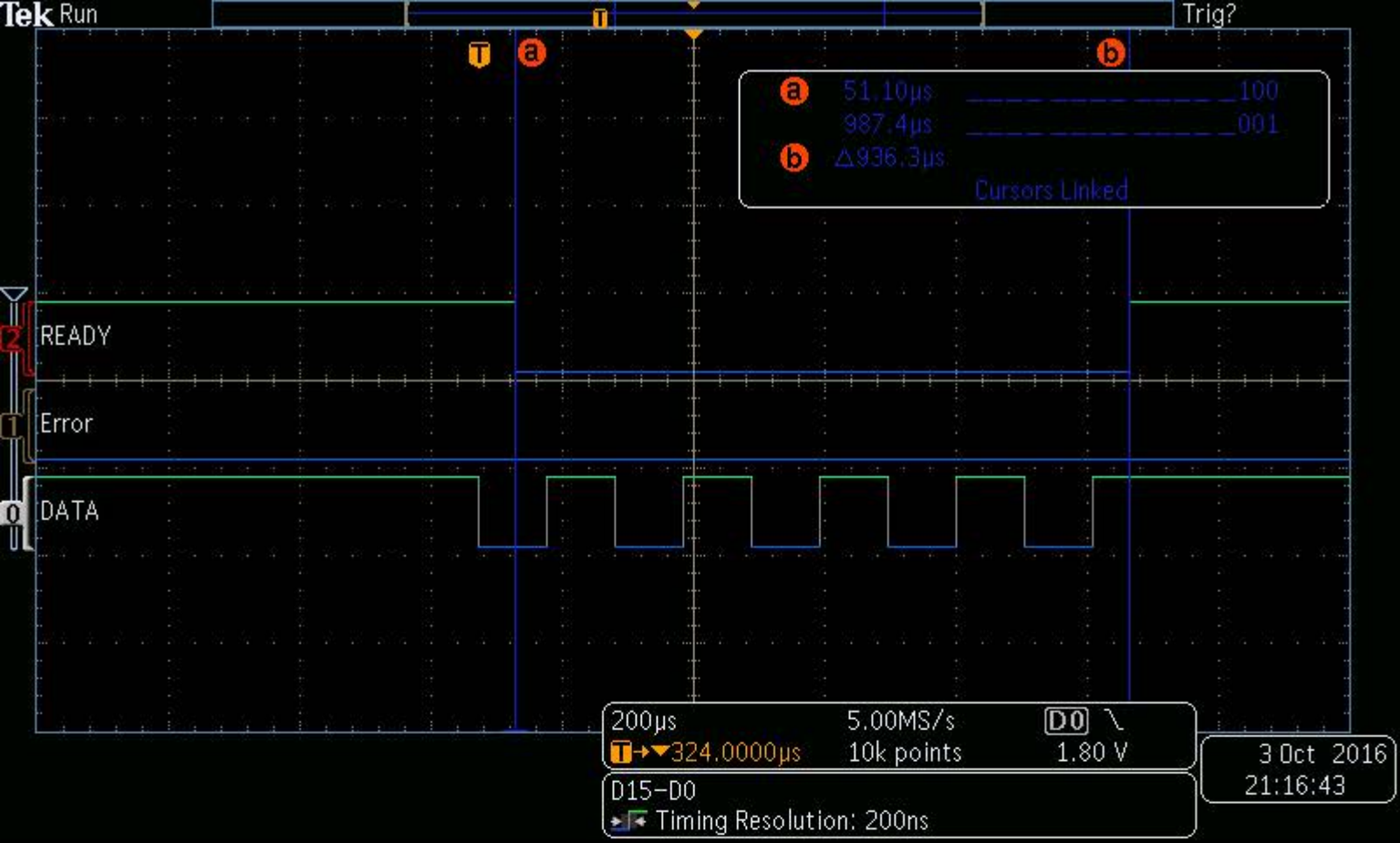
> Receiver operational at lower baud limit of 9600-300 = 9600 MHz



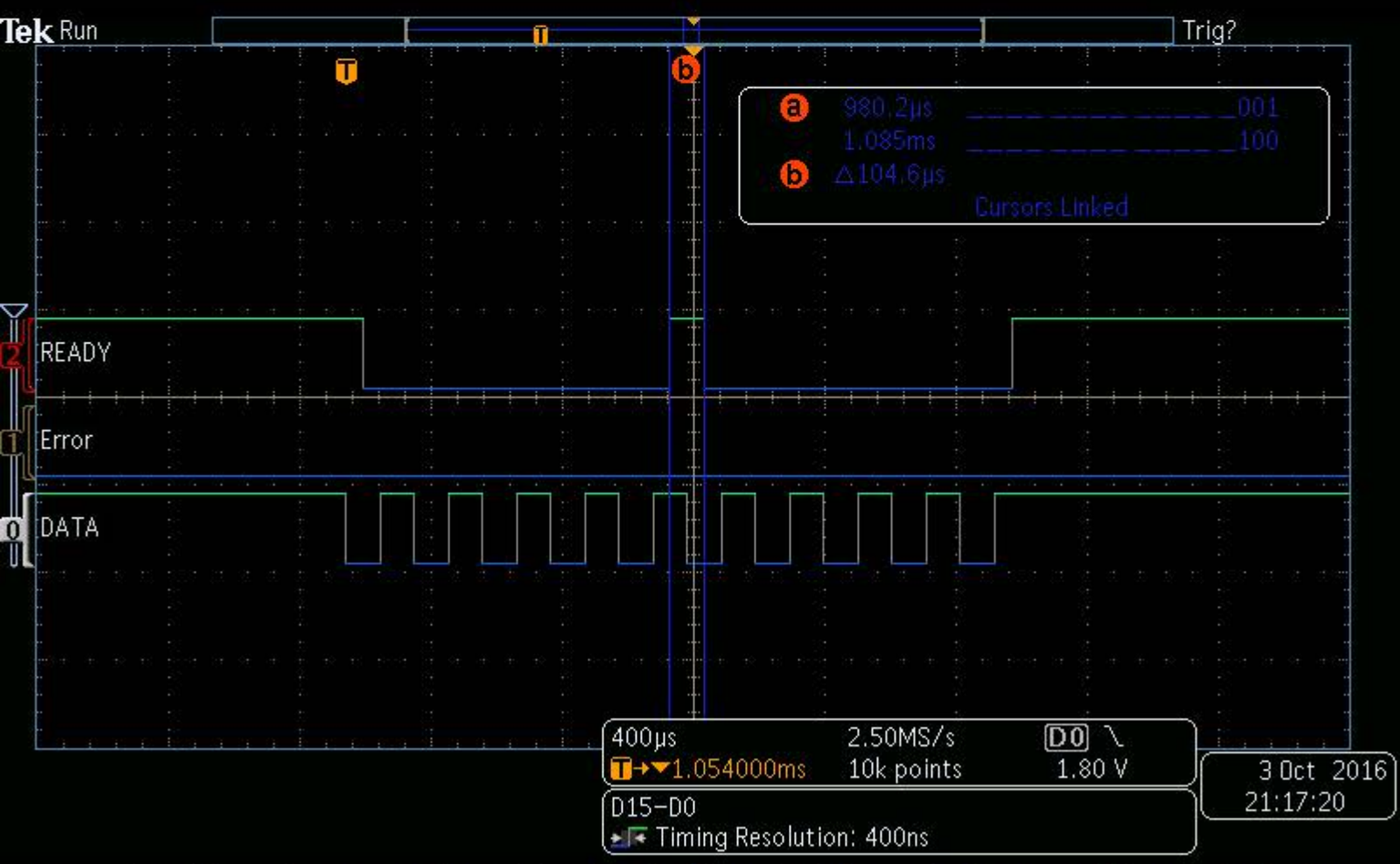
> Start bit width not sufficient (Spurious Start). ready does not drop low. No recieving occurs.



> Start bit width wider than 40us. Valid start bit. Receiver is receiving data. ready drops low.



> Single byte reception. ready performance as expected. Baudrate as expected.



> Reception of 2 bytes in a row. ready goes high after first bit received; low again when second bit reception begins. Data as expected.

APPENDIX C-3