TECHNICAL MEMORANDUM

Lafayette College

Department of Electrical and Computer Engineering

Title: WimpFi Project

Author(~~s~~): G. Flynn

Date: November 29, 2016

Abstract

This memo presents documentation demonstrating the functionality of the WimpFi base station. The station is capable of receiving and transmitting all types of packets required as well as performing backoff.

Introduction

The WimpFi station should be capable of receiving 4 different types of packets as well as perform backoff to reduce interference. All packets are formatted in a similar way: destination, source, type, data, then a FCS if appropriate. A type 0 (0x30) packet has no FCS. A type 1 (0x31) does. A type 2 (0x32) has a FCS as well as requires an acknowledgement. A type 3 (0x33) has a FCS and is the acknowledgement for a type 2 packet. The station also needs to ensure that radio waves are quiet for a specified amount of time.
The requirements ask for two units, a transmitter interface and a receiver interface. To demonstrate the functionality of the hardware additional wrappers are needed.

Design

For all aspects of the design there are block diagrams at the end of this document with FSM state diagrams providing information about the inner workings for the modules.

Receiver interface

The receiver interface was capable of parsing types 0-3. The first location that the manchester data hit was the manchester receiver. This converted the manchester data into a 8-bit parallel connection. From there the 8 bit connection went in many directions. The most important link was to the FSM_ADDR. This was the primary brains of the receiver interface. It would parse through the data and identify if the packet was for us. As the data is being received another FSM identifies the packet type. The FSM_ADDR uses this information to decide to check the CRC and activate the got_ack or send_ack signals. The FSM writes all incoming data into a modified FIFO which can also pop the last entry. This is used to pop the CRC after it has been verified. Another FSM, FCS_FSM clocks the data through the CRC generator. When the FSM_ADDR is done with the data if the packet is not type 0 it verifies that the CRC is 0. If it is not the FIFO is flushed and the error counter is incremented.

Transmitter interface

The core of the transmitter interface is the FSM_TX. This module loads a preamble, SFD and all data available in the FIFO. It also identifies the packet type and ensures that if a CRC is needed it is appended to the packet. Also if a type 2 packet is sent out then this FSM enters a state to wait for the acknowledgement or for a retry request. An important feature of the transmitter was to ensure that it did not jam the radio waves. To achieve this a watchdog timer was implemented that allowed the txen signal to only stay high for 512 byte periods.

Another major module was the backoff module. This module had a simple interface with the transmit FSM. It had a RTS, RTS_ack and CTS signal. Once the transmit module said that it would like to send a packet the FSM module would perform a variety of checks to ensure that the radio was clear to send. At this point the CTS signal would be asserted.

If a type 2 packet was received then an additional FSM would start up. This FSM would load the required type 3 packet to send. This module would also trigger the RTS_ack signal.

Transmitter side

To connect the transmitter to the UART receiver there was a simple FSM that would parse the data from the UART receiver and present it to the transmitter interface.

Receiver side

The receiver has a simple FSM that allows for the data from the receiver interface to be displayed on the screen.

Core Requirements

There will be a hardware and a simulation setup to test every single test.

code	Test	Setup	Evidence

rx1.0	T0 all call	Send 0x2f type 0	Check data
rx1.1	T1 all call	Send empty type 1 with data	Check data, Check CRC, check error
rx2.0	T0 MAC	Send MAC type 0	Check data
rx2.1	T1 MAC	Send MAC type 1	Check data, check CRC, check error
rx2.2	T0 not MAC	Send not MAC type 0	No data change, no error
rx2.3	T1 not MAC	Send not MAC type 1	No data change, no error
rx2.4	T1 MAC	Send MAC type 1 bad CRC	Check data, check CRC, check error
tx1.0	Load data	Load >255 bytes	Check first 255 bytes Check no more
tx2.0	Send data	Load and send data (xsnd)	Check xrdy fall Check data sent
tx2.1	Send data with traffic	Load and send data (xsnd) after cardet falls	Check data before cardet fall Check xrdy Check data sent Check back off
tx2.2	Send to MAC	Load data and send to specific address	Check data Check addr
tx2.3	Send data	Load and send	Check correct source
tx2.4	Send type 1	Load and send	Check CRC
ack1	Send type 2	Load and send	Check wait for ACK
ack2	Resend	Load and send with no ACK on first tx	Check retry
ack3	Timeout	Load and send with no ACK ever	Check retry. Check timeout
ack4	ACK	On rx of type 2 send ACK	Check address Check packet type Check CRC Check bo4

ack5	Bad ACK	On rx of type 2 send ACK wrong SRC	Check retry. Check timeout
bo1	DIFS spacing	Send a packet on quiet network	Check wait for DIFS
bo2	DIFS + Slots	Send a packet on noisy network	Check wait for quiet Then DIFS Then 1-63 slots
bo3	DIFS + Slots + Contention	Send packet on noisy network. Crash during contention	Check wait Then DIFS Then slots Then hold Then DIFS at idle Then slots
bo4	SIFS	Rx T2 (see ack4)	Check wait SIFS Check ack4
co1	Failsafe	Shutdown after 512 bytes (81920us)	Check txen to txen

Design verification

Hardware

Type 0

Requirement	Criteria	Evidence
Send type 0	Dest/Src/Type and data sent	Displayed on screen
Receive type 0 mac	Get data from mac address	Displayed on screen
Receive type 0 all call	Get data from all call	Display on screen
Ignore type 0 not us	Get no data	Nothing on screen

Type 1

Requirement	Criteria	Evidence

Send type 1	Dest/Src/Type and correct CRC	Displayed on screen
Receive type 1 mac	Get data from mac address	Displayed on screen
Receive type 1 all call	Get data from all call	Display on screen
Ignore type 1 not us	Get no data	Not displayed on screen
Ignore bad CRC	Get no data. Inc error count	Not displayed on screen Error count increased

Type 2 tx/ Type 3 rx

Requirement	Criteria	Evidence
Send correct packet	FCS correct Type = 0x32	ACK from demo core on scope
Retry with no ack on silent network	Wait 5.120ms +/- 256us Then retry	txen vs txen spacing
Retry timeout	Check retry 6 times including the first attempt	Packet count (txen on scope 5 times)
Get ack on retry (after 1...4 retries)	Check ack on retry	Txen vs cardet for ack package
Ignore bad FCS ack	FCS wrong on ACK Resend type 2	Packet count = 5

Type 3 tx/ Type 2 rx

Requirement	Criteria	Evidence
Ack valid personal packet	FCS correct MAC address not ALL CALL	Scope waveform of cardet then txen
Ack in SIFS	Timing 800us +/- 40us	Difference between cardet and txen
No ack all call	FCS correct MAC address ALL CALL	No ack sent on scope

No ack not our personal packet	FCS correct MAC address does not match ours	No ack sent on scope
No ack bad FCS	FCS wrong MAC matches ours	No ack sent on scope

Backoff

Requirement	Criteria	Evidence
Wait DIFS on quiet network	1600us +/- 80us	Scope xrdy -> txen
Wait slots on busy network	Cardet -> txen Min: 1760us (1 slot) Max: 11840us (64 slots)	Scope cardet -> txen
Wait slots random	Repeat last test. Check different times	Scope cardet -> txen
Wait and get stepped on	Send large packet with small gap	Cardet -> txen
Never exceed max	Predicted max +2*std_dev < 11840us	m-script
Accurate	Error on mean < 10%	m-script
LFSR in range	Never > 64 Never < 1	r-script
LFSR centered	Mean = 32 with >90% confidence	r-script

Statistical analysis of backoff

To characterise backoff 2 statistical tests were required. The first desired trait to identify was the average mean of the random generator. The second would be to characterize the maximum value of the LFSR.

Average mean

Since an LFSR was used there should be a slight bias towards slightly more than 32 since 0 can never appear. Using a python script the LFSR was modeled and ran for 2000 clock cycles

5000 times, each time starting from a random value. These values were read into a R-script which performed analysis on the values. It can be seen from the histogram plot that there are more frequent averages just above 32 than just below. The average is at 32 with a very slight lean towards the lower values. This is because there are more values further from 32 towards the lower values. The mean was between 31.99911 and 32.00088 with 95% confidence and with p equal to 0.9927 according to the R-script log.

Maximum value

To solve the maximum value problem the solution to the german tank problem was used. This involved predicting the maximum value by using the recorded values. A total of 113 timing values were sampled in hardware. These were all recorded into the MatLab script. After that the solution to the tank problem was found as well as the standard deviation. This allowed the maximum value to be identified with a 95% certainty. The script found that the maximum time was within limits and the mean was a little bit lower than expected as predicted by the R-script. With 95% certainty the maximum value was found to be 63 slots.

Backoff conclusion

The LFSR module did not allow for the full range. The specification asked for a numeric value between 1 and 64 but the LFSR can only get to 63. As a result all of the timing requirements were never exceeded.

Verifying requirements

1. The frame transmission format was verified with self checking test benches.
2. The transmitter was modified slightly to allow for clear signals for sending and receiving acknowledgements. Other than those cabled the ports were kept the same.
3. The receiver had the same modifications to connect the nets.
4. Throughout the hardware the bit rate is passed through as a variable. This allows the design to be parameterized to work at different rates.
5. The receiver was previously tested to work with up to 5ns of jitter and a variation of +/-1%.
6. The FIFO used to build the frame was made to be 255 bytes long to ensure that more than 255 bytes cannot be written in. Once the FIFO is full attempting to write additional bytes to it will not change the stored values.
7. The hardware used 1256 LUTs. The XC7S15 has 8000 LUTs so the hardware easily fits onboard.
8. The system only responds to packets that are addressed to it
9. The system also responds to all call packets
10. The MAC address was settable by the switches on the FPGA.
11. When demonstrating all types of packets were sent to multiple different users

12. The center button provides a GSR for all sequential circuits on the board
13. This Manchester Receiver does not violate the Nortel patent for a variety of reasons:
    a. The sample rate for this receiver is at 64x the baud rate vs 5x baud rate
    b. This receiver converts the data into a 8-bit parallel connection vs a serial data line
    c. This receiver uses a PLL to lock onto the bits
14. There is a fail safe that prevents more than 512 bytes being transmitted at one time
15. The Nexys4 DDR is RoHS compliant.
16. All of the system was designed using SystemVerilog
17. There are no latches in the design
18. The hardware was verified in simulation by using multiple test benches as documented above and in the test logs
19. The test logs print summaries stating the results of the tests
20. There are hardware wrappers that interface with the transmitter and the receiver
21. There is a requirements checklist in this document

Implications of design

This design was designed to be able to work on smaller boards. This would help to reduce the cost of manufacturing additional WimpFi units. These stations allow for long range indoor communication (largest test at about 15m). To prevent these units from creating harmful interference there are 2 main safety features. The first is a watchdog timer for the txen line. This prevents the system from jamming another signal. The second protection method is the backoff module. This module detects if someone else is trying to talk. If someone is talking the station waits until there is silence to attempt to prevent harmful interference.

Log files

R-script (2000)

```
> nSims <- 5000 #number of simulated experiments
> max_val <-numeric(nSims) #set up empty container for all simulated
max values
> min_val <-numeric(nSims) #set up empty container for all simulated
min values
> mean_val <-numeric(nSims) #set up empty container for all simulated
mean values
> values = read.table("data.txt")
>
> for(i in 1:nSims){ #for each simulated experiment
+   y<-values[[i]]
+
+   max_val[i]<-max(y)
+   min_val[i]<-min(y)
+   mean_val[i]<-mean(y)
+ }
>
> #now plot the histogram
> hist(max_val, main="Histogram of max values from LFSR", xlab=("Max
value"),breaks = c(60,61,62,63,64,65,66))
> hist(min_val, main="Histogram of min values from LFSR", xlab=("Min
value"),breaks = c(-1,0,1,2,3,4))
> hist(mean_val, main="Histogram of mean values from LFSR",
xlab=("Mean value"),
+      breaks =
c(31.90,31.91,31.92,31.93,31.94,31.95,31.96,31.97,31.98,31.99,
+
32.00,32.01,32.02,32.03,32.04,32.05,32.06,32.07,32.08,32.09,
+                 32.10))
>
> t.test(mean_val, mu=32)

      One Sample t-test

data:  mean_val
t = -0.0091187, df = 4999, p-value = 0.9927
alternative hypothesis: true mean is not equal to 32
```

```
95 percent confidence interval:
 31.99911 32.00088
sample estimates:
mean of x
       32
```

Matlab script (Tank problem)

```
Max time: 11400us, Limit: 11840us
Standard Deviation: 100.3876us
Max time (95% confidence): 11706us, Limit: 11840us
Mean time: 6421us, Expected: 6780us
Error on mean:  9.44%, Target: 10%
```

Type 0 Rx

```
START: Rx mac address type 0 at time 200.
OK: Rx all call type 0 (5 tests passed)
START: Rx mac address type 0 multi packet at time 2026436.
OK: Rx mac address type 0 multi packet (5 tests passed)
START: Rx not our mac address type 0 at time 4090386.
OK: Rx not our mac address type 0 (1 tests passed)

Tesbench Complete.
No errors in 16 tests. :)
```

Type 0 Tx

```
START: Send type 0 at time 100.
OK: Send type 0 (3 tests passed)

Tesbench Complete.
No errors in 3 tests. :)
```

Type 1 Rx

```
START: rx2.1 Type1 to mac address at time 100.
```

```
OK: rx2.1 Type1 to mac address (1 tests passed)
START: rx2.4 bad CRC at time 5120115.
OK: rx2.4 bad CRC (1 tests passed)
START: rx2.3 bad MAC at time 10277355.
OK: rx2.3 bad MAC (2 tests passed)

Tesbench Complete.
No errors in 4 tests. :)
```

Type 1 Tx

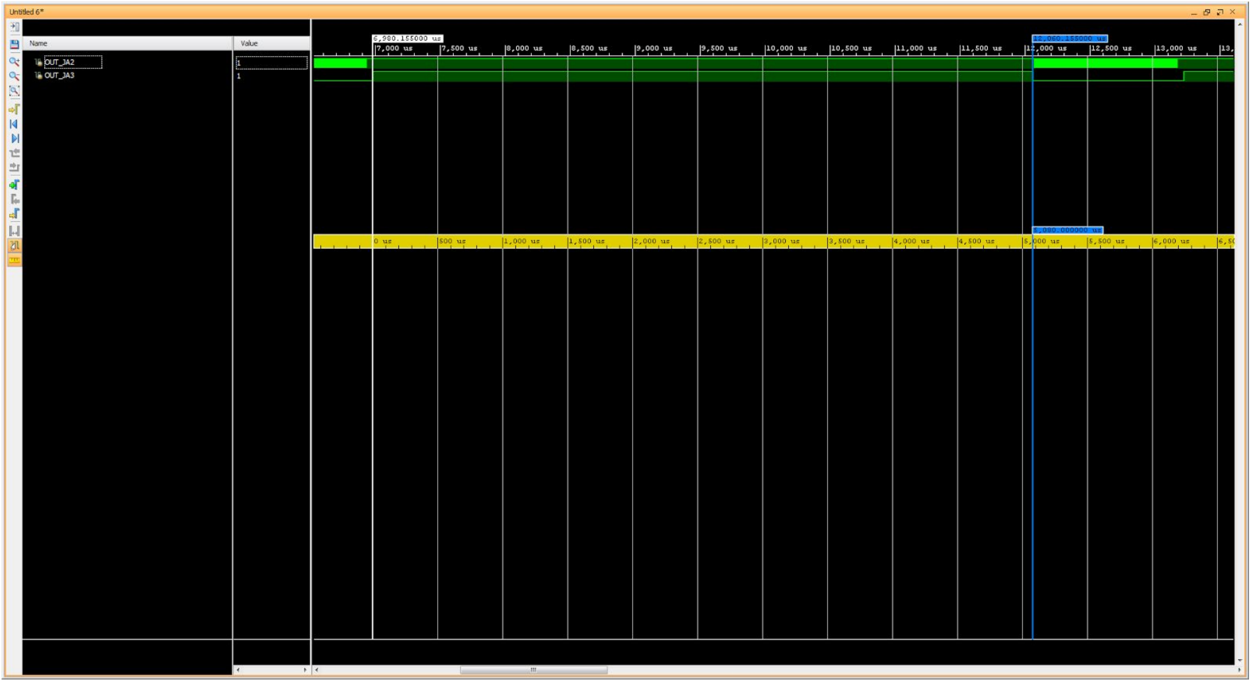
```
START: Send type 1 at time 100.
OK: Send type 1 (4 tests passed)

Tesbench Complete.
No errors in 4 tests. :)
```

Type 2 Tx/Type 3 Rx

```
START: Send type 2 nak at time 100.
OK: Send type 2 nak (20 tests passed)

Tesbench Complete.
No errors in 20 tests. :)
```



Timing of retry

START: Send type 2 ack at time 100.
OK: Send type 2 ack (5 tests passed)

Tesbench Complete.
No errors in 5 tests. :)

Type 2 Rx/Type 3 Tx

START: Get type 2/Send type 3 at time 100.
OK: Get type 2/Send type 3 (4 tests passed)

Tesbench Complete.
No errors in 4 tests. :)

START: Get type 2 all call at time 100.
OK: Get type 2 all call (1 tests passed)

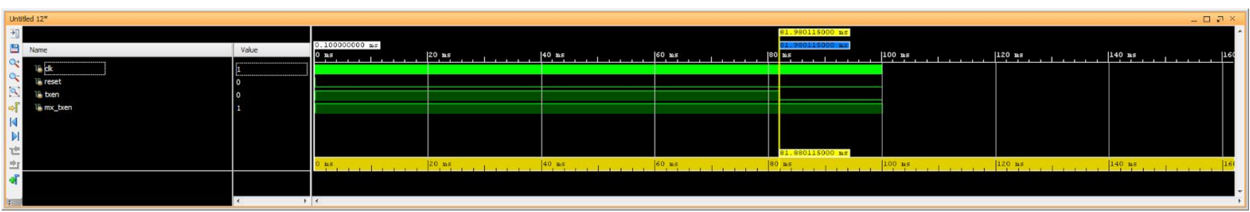
Tesbench Complete.
No errors in 1 tests. :)

Backoff

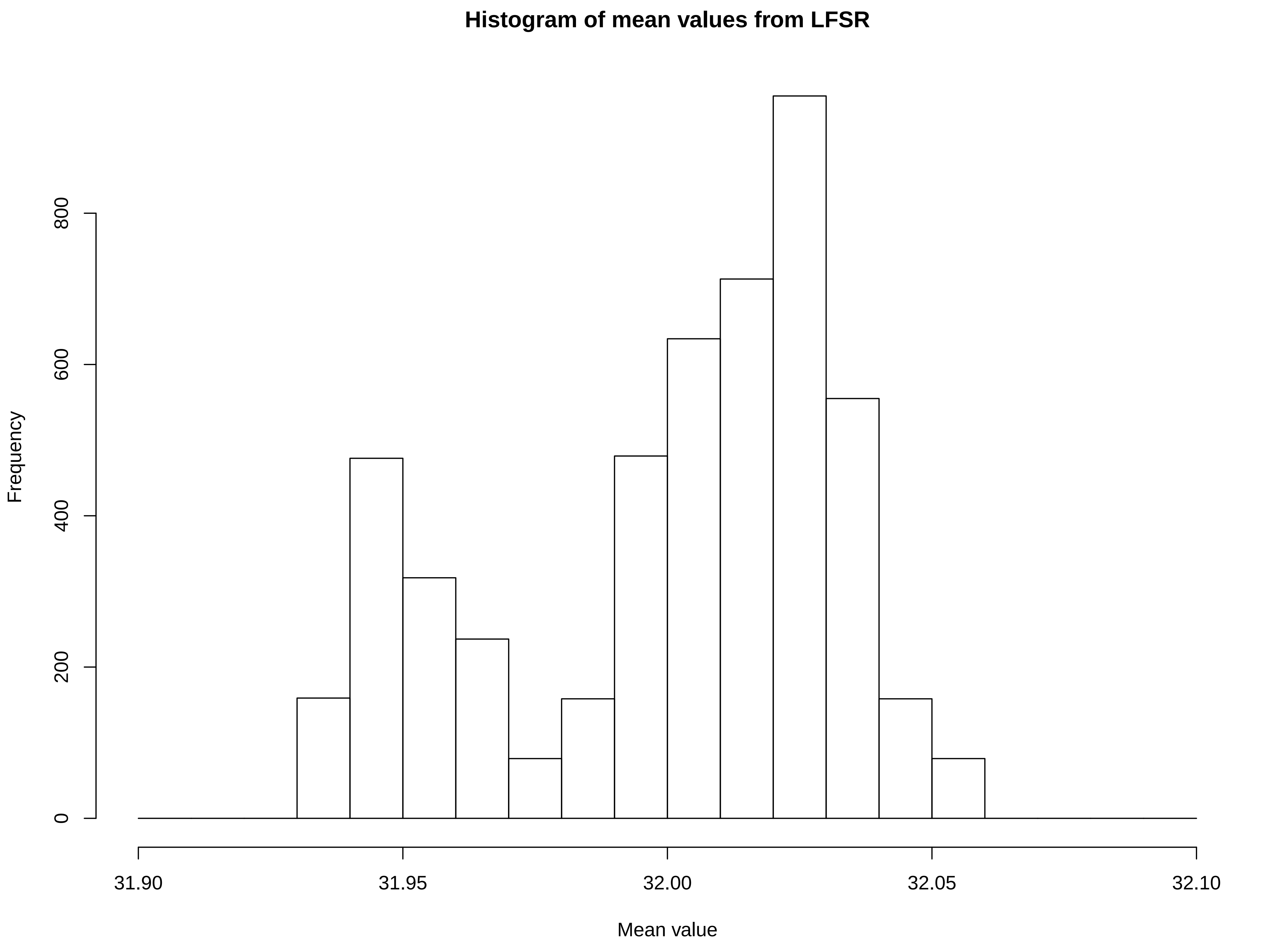
```
START: No traffic at time 200.
OK: No traffic (1 tests passed)
START: cardet for first try at time 1580215.
OK: cardet for first try (3 tests passed)
START: Glitchy traffic at time 11680115.
OK: Glitchy traffic (2 tests passed)

Tesbench Complete.
No errors in 6 tests. :)
```

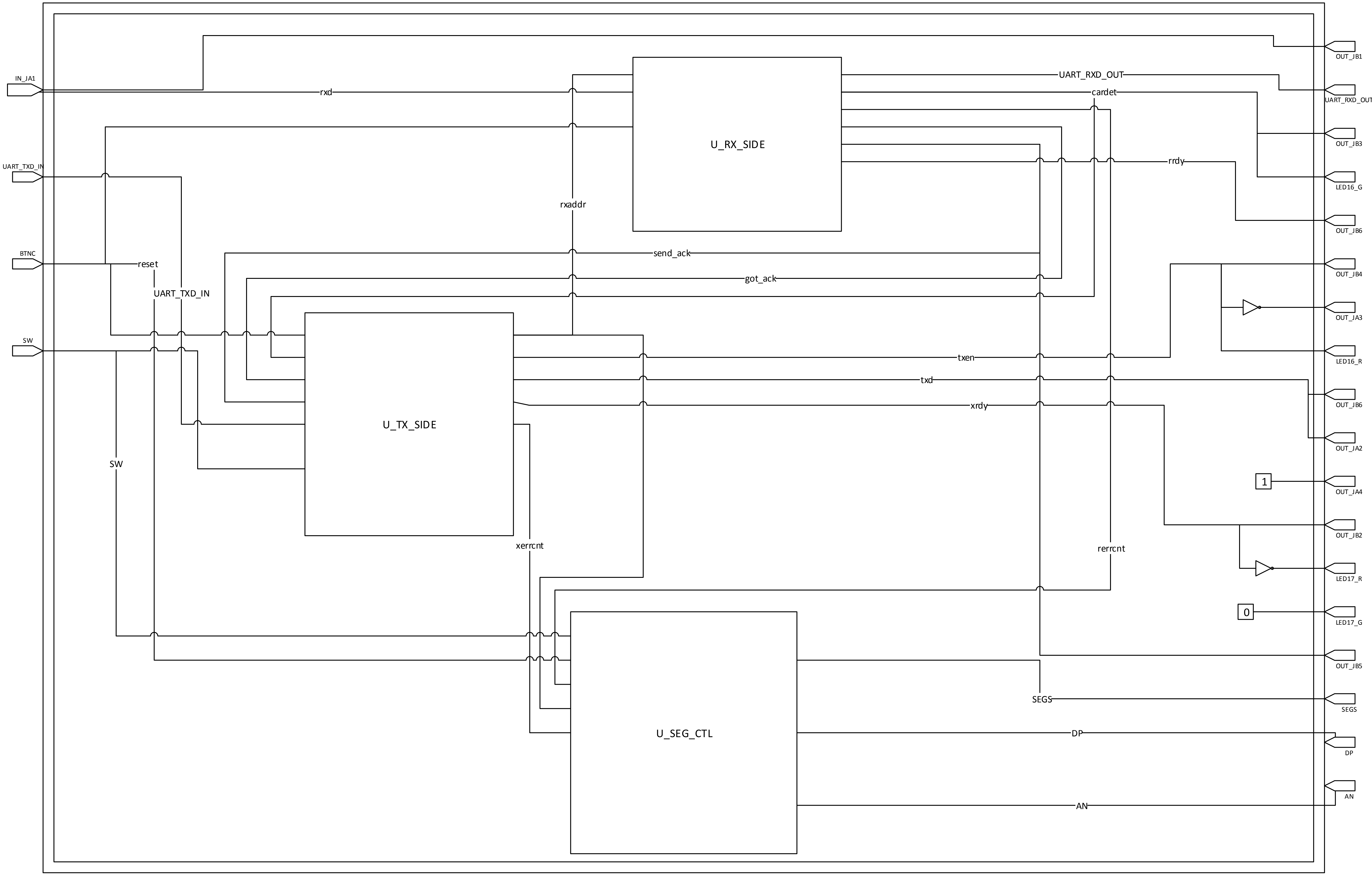
Safety cutout



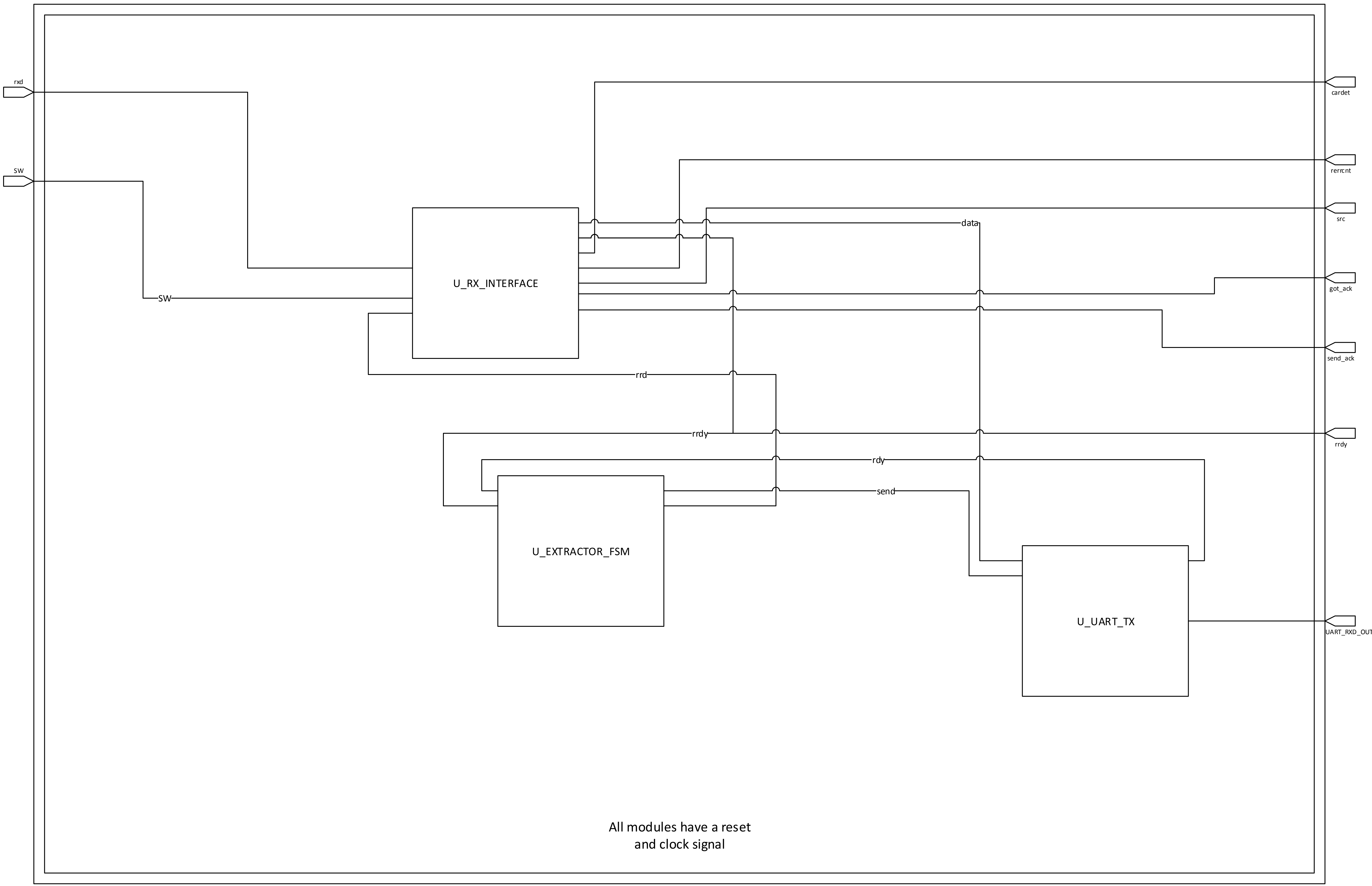
Timing for safety cutout

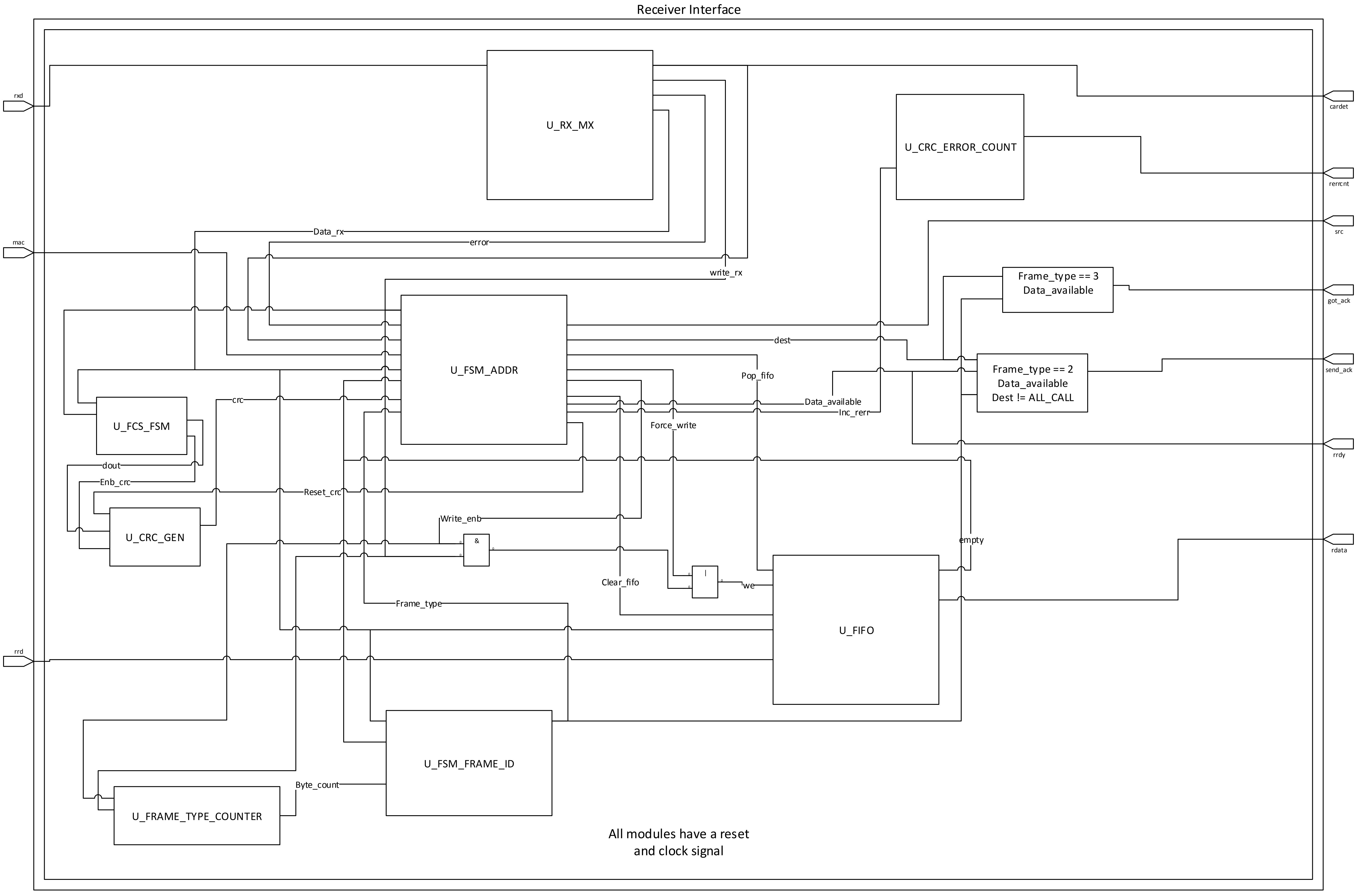

Histogram of mean values from LFSR
Frequency
0
200
400
600
800
31.90
31.95
32.00
32.05
32.10
Mean value

Top level Block Diagram



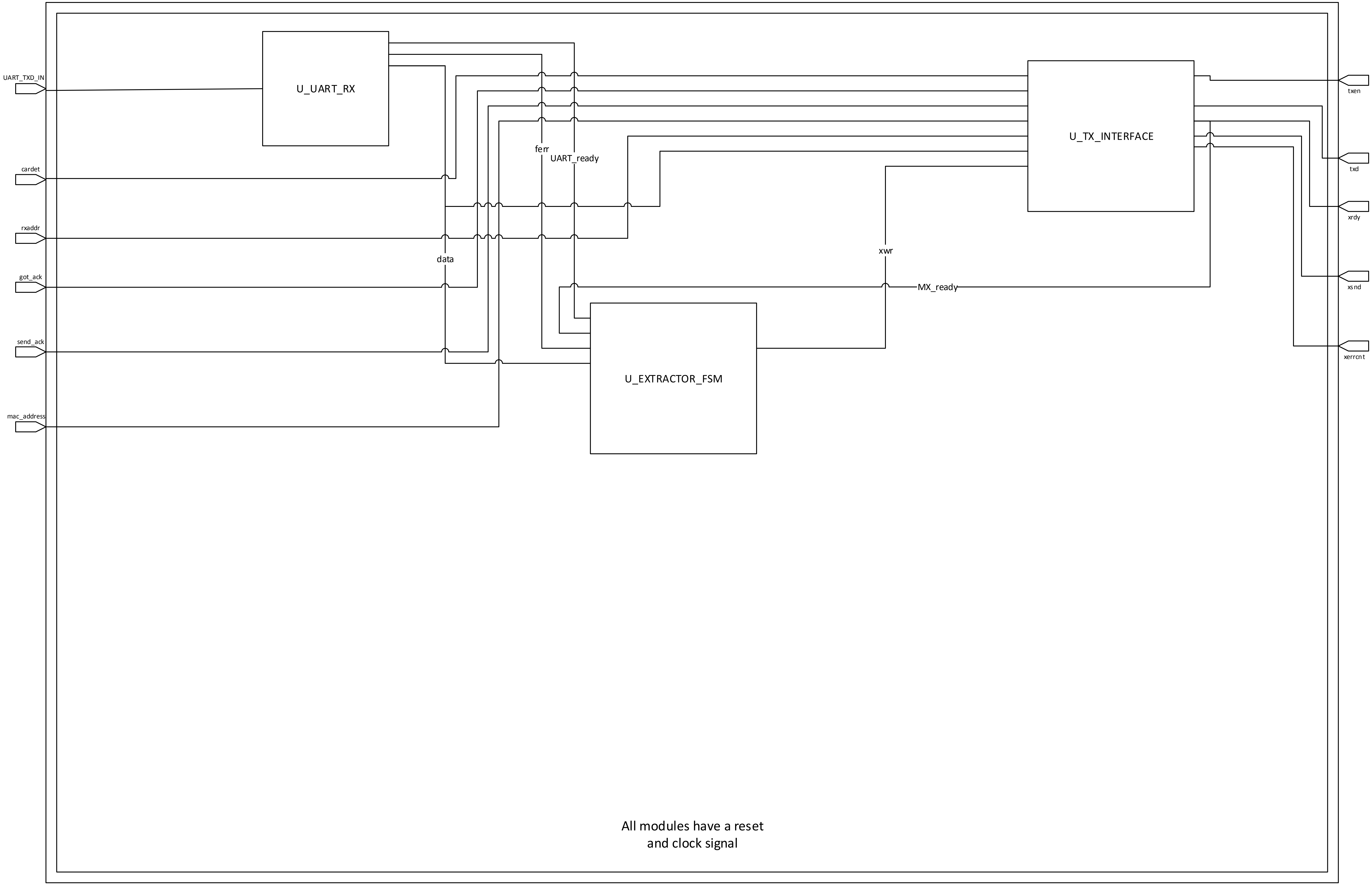
Receiver side



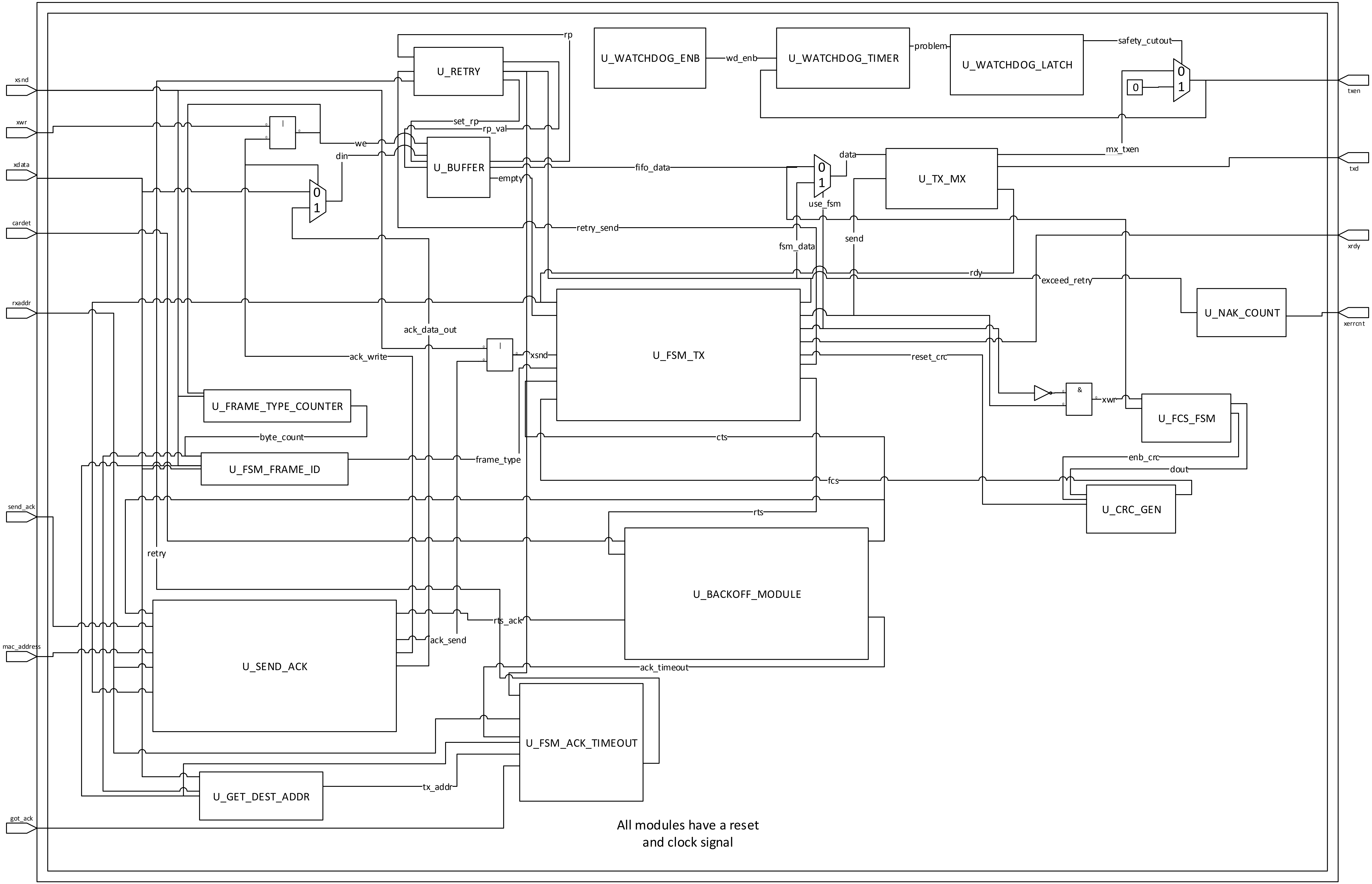


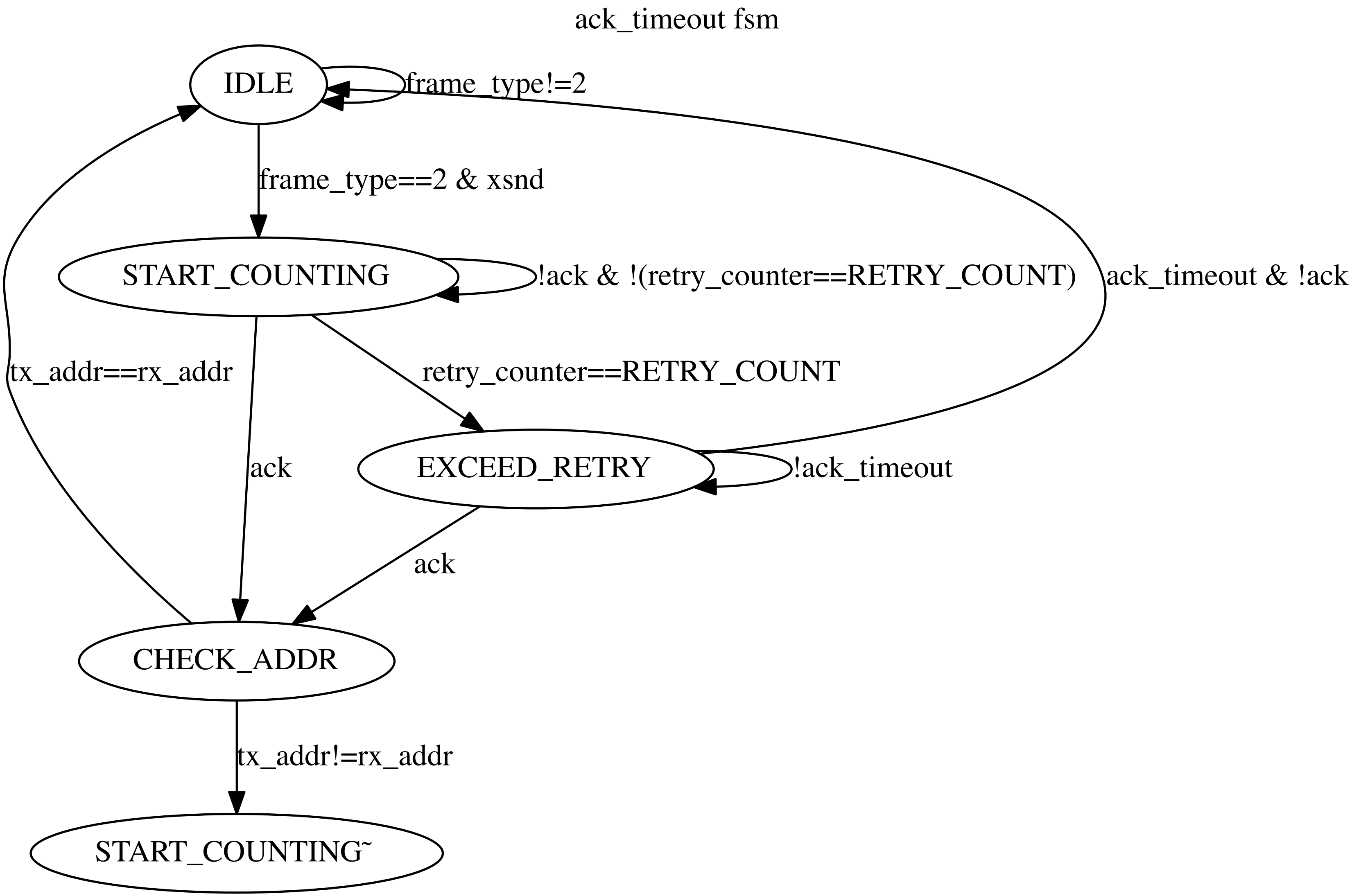
Receiver Interface
rxd
mac
rrd
cardet
rerrcnt
src
got_ack
send_ack
rrdy
rdata
U_RX_MX
U_CRC_ERROR_COUNT
Data_rx
error
write_rx
U_FSM_ADDR
Frame_type == 3
Data_available
dest
Frame_type == 2
Data_available
Dest != ALL_CALL
Pop_fifo
Data_available
Inc_rerr
Force_write
crc
U_FCS_FSM
dout
Enb_crc
Reset_crc
U_CRC_GEN
Write_enb
&
empty
we
Clear_fifo
Frame_type
U_FIFO
U_FSM_FRAME_ID
Byte_count
U_FRAME_TYPE_COUNTER
All modules have a reset
and clock signal

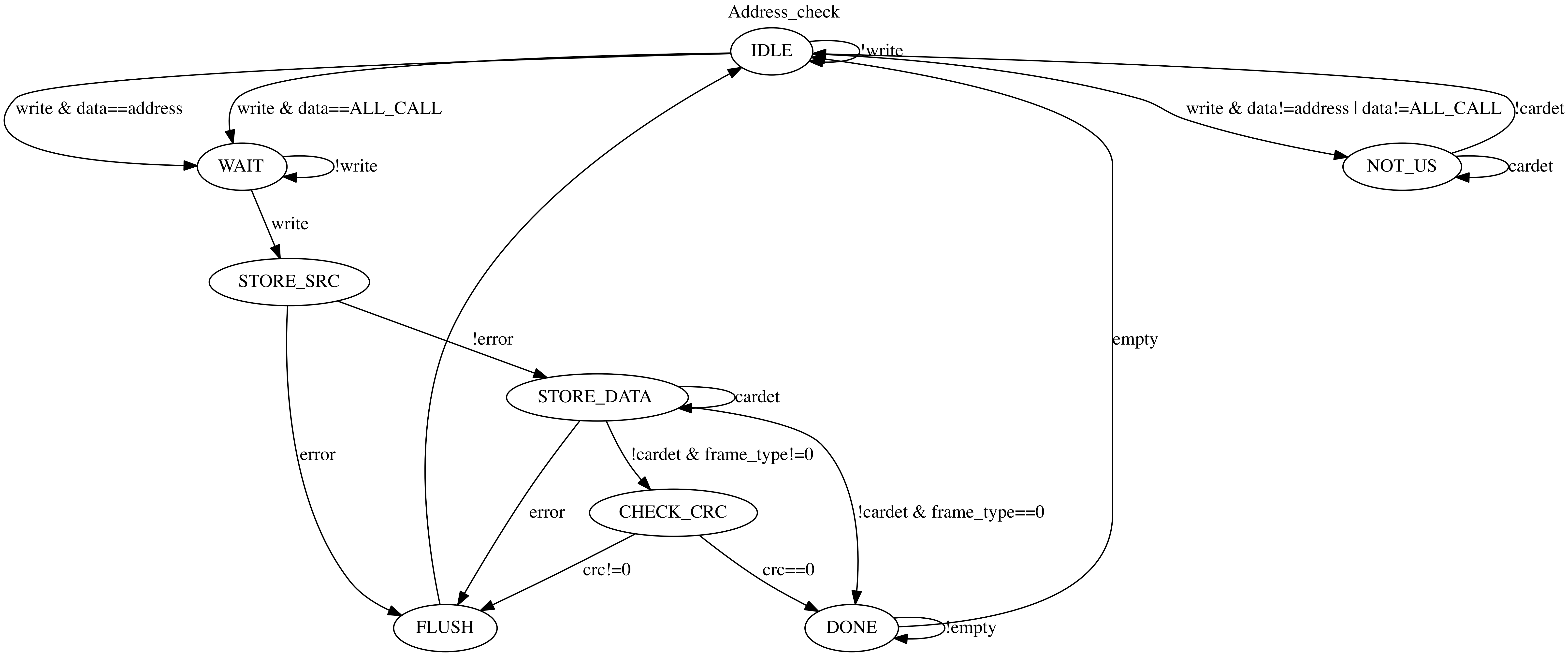
Transmitter side

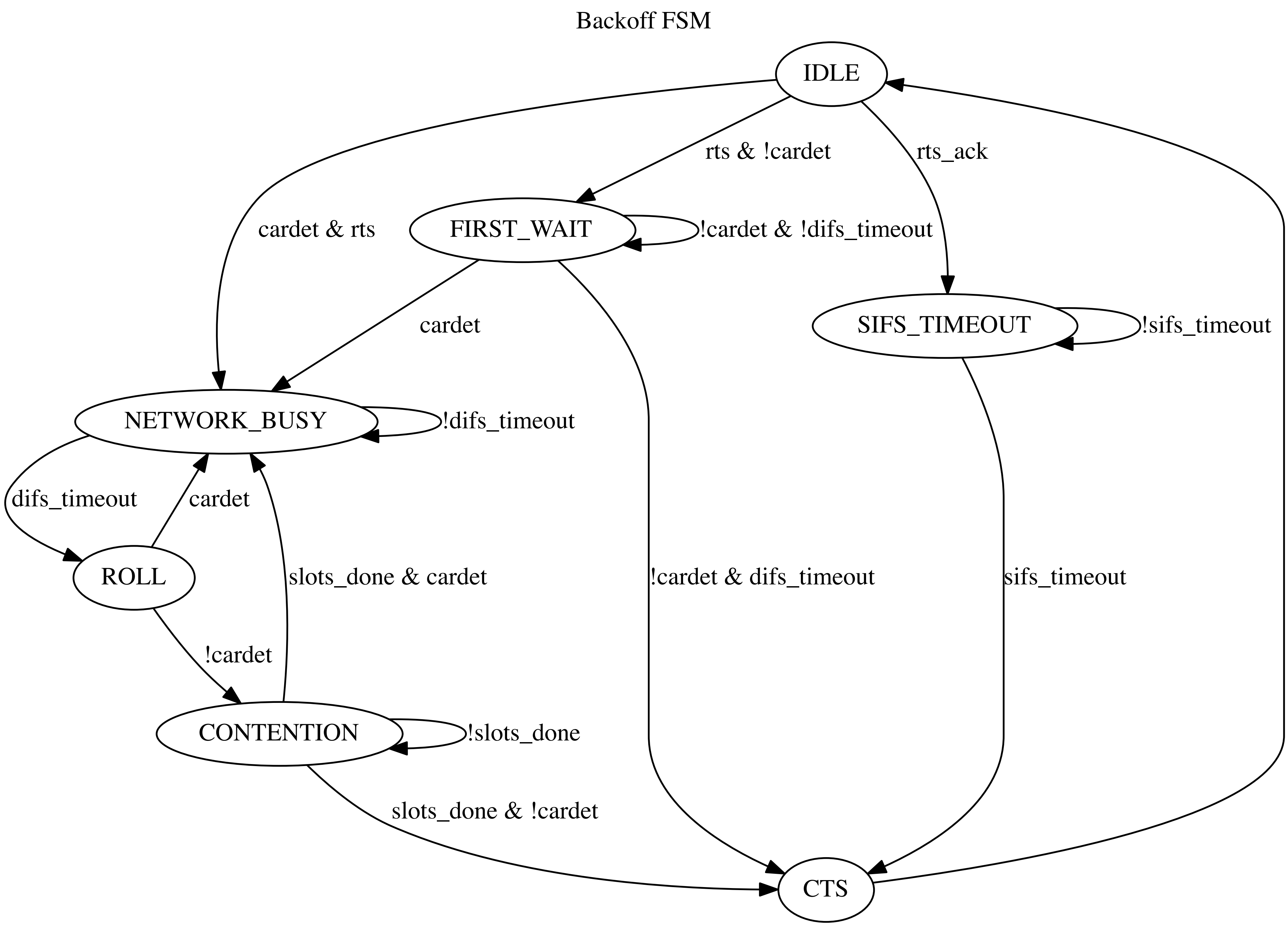


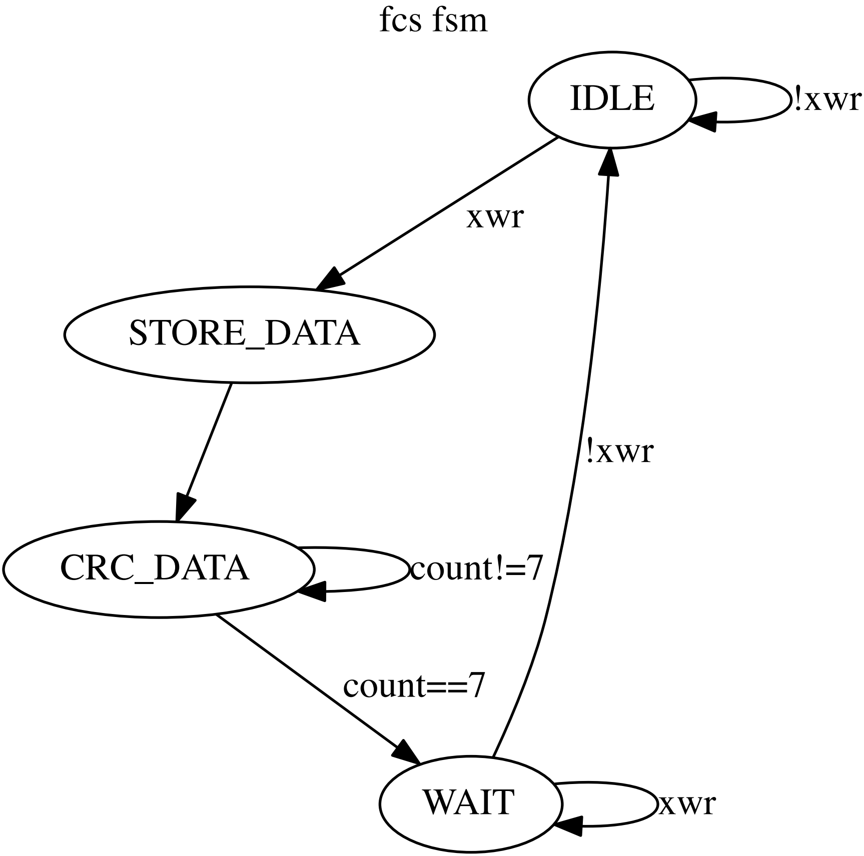
Transmitter interface



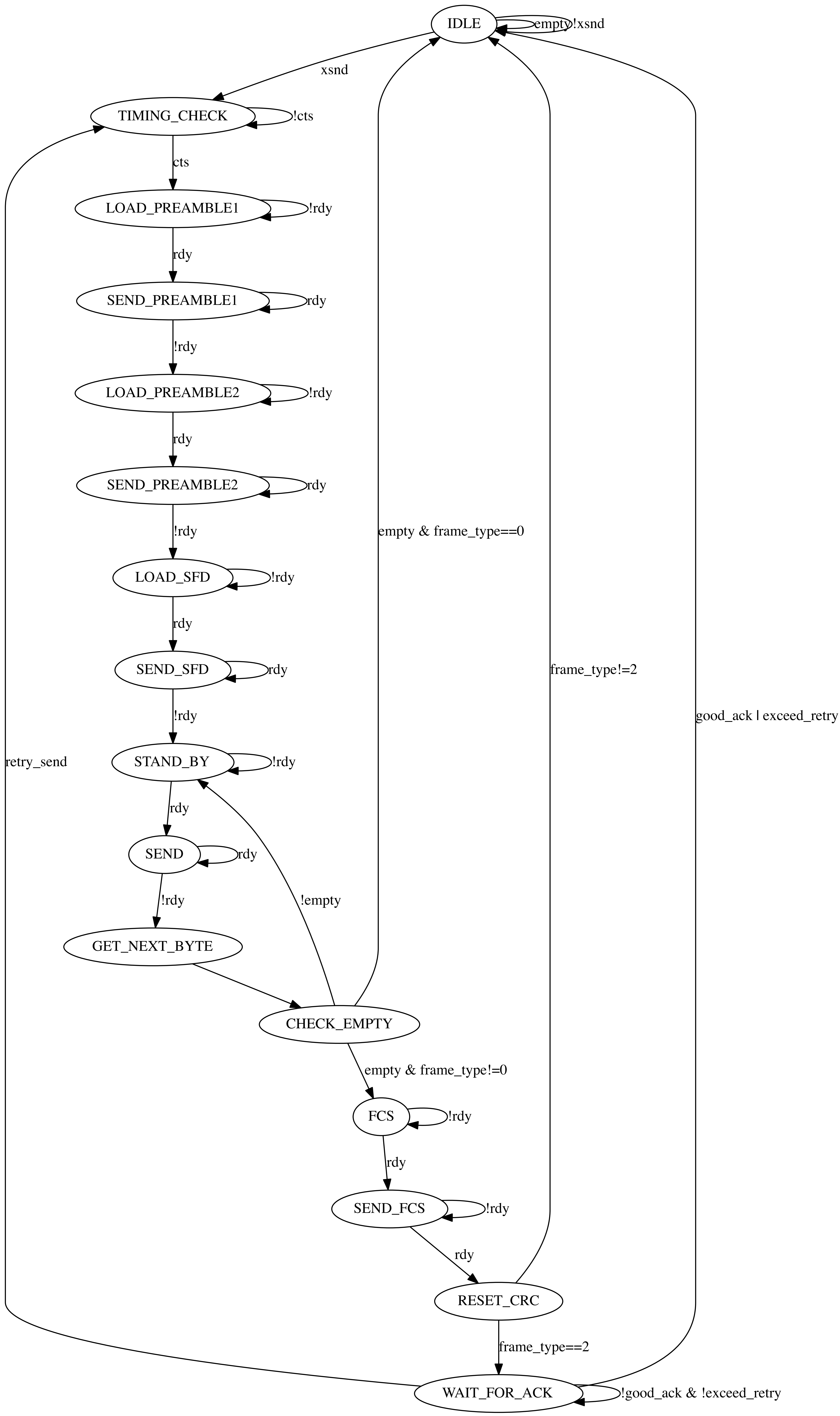

ack_timeout fsm
IDLE
frame_type!=2
frame_type==2 & xsnd
START_COUNTING
!ack & !(retry_counter==RETRY_COUNT)
ack_timeout & !ack
tx_addr==rx_addr
retry_counter==RETRY_COUNT
ack
EXCEED_RETRY
!ack_timeout
ack
CHECK_ADDR
tx_addr!=rx_addr
START_COUNTING~

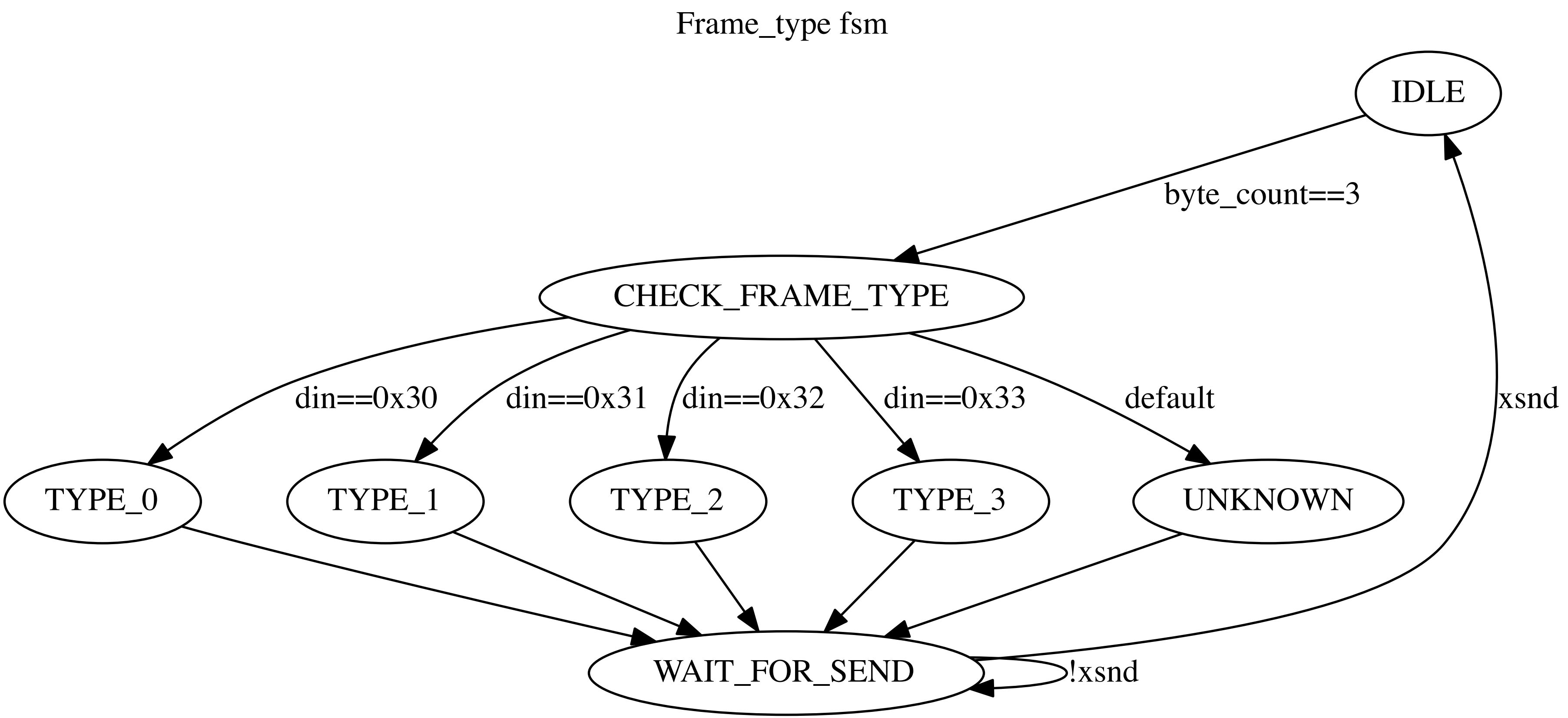

Address_check
IDLE
!write
write & data==address
write & data==ALL_CALL
write & data!=address | data!=ALL_CALL
!cardet
WAIT
!write
NOT_US
cardet
write
STORE_SRC
!error
empty
STORE_DATA
cardet
error
!cardet & frame_type!=0
error
CHECK_CRC
!cardet & frame_type==0
crc!=0
crc==0
FLUSH
DONE
!empty

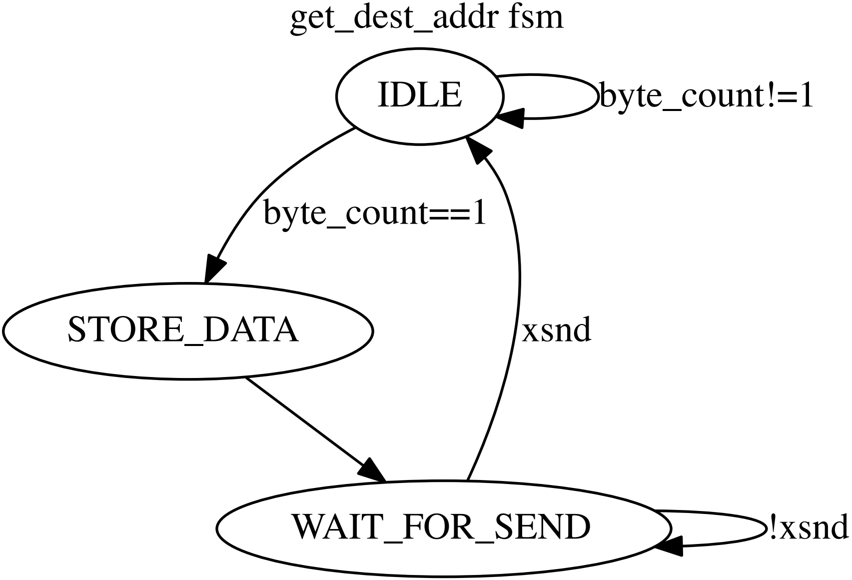

Backoff FSM
IDLE
rts & !cardet
rts_ack
cardet & rts
FIRST_WAIT
!cardet & !difs_timeout
cardet
SIFS_TIMEOUT
!sifs_timeout
NETWORK_BUSY
!difs_timeout
difs_timeout
cardet
ROLL
slots_done & cardet
!cardet & difs_timeout
sifs_timeout
!cardet
CONTENTION
!slots_done
slots_done & !cardet
CTS

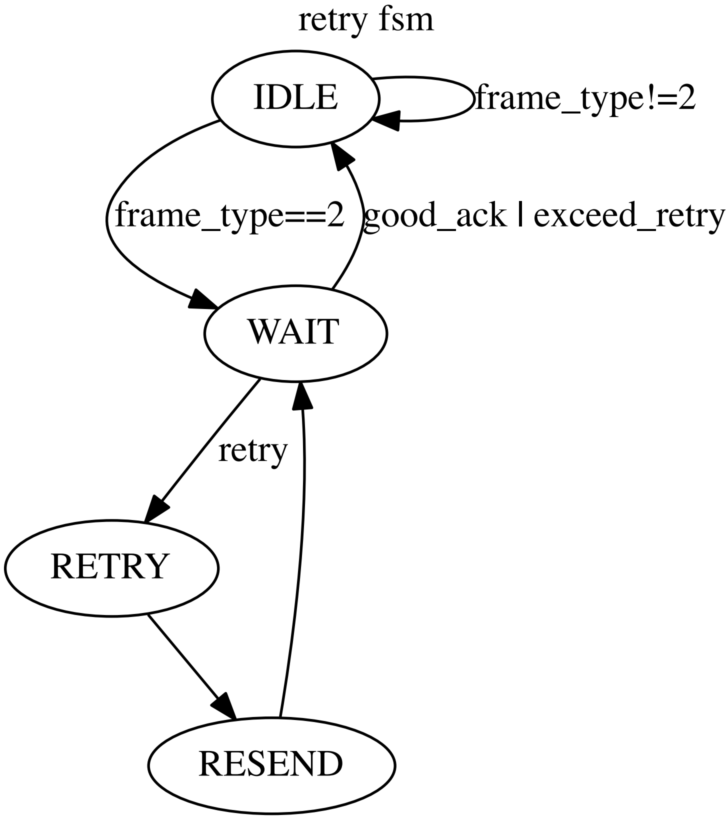

fcs fsm
IDLE
!xwr
xwr
STORE_DATA
CRC_DATA
count!=7
!xwr
count==7
WAIT
xwr

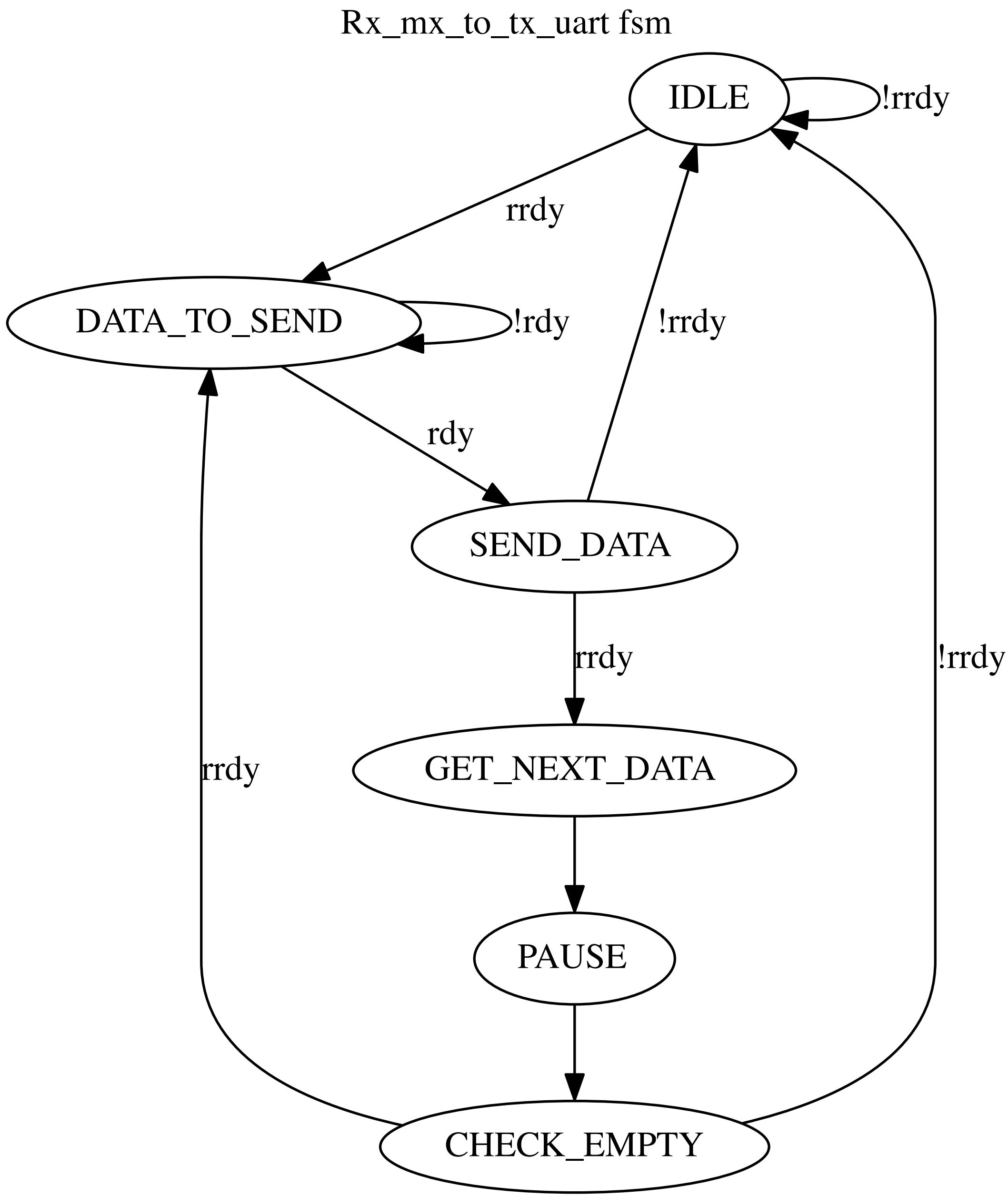
fifo_to_send fsm

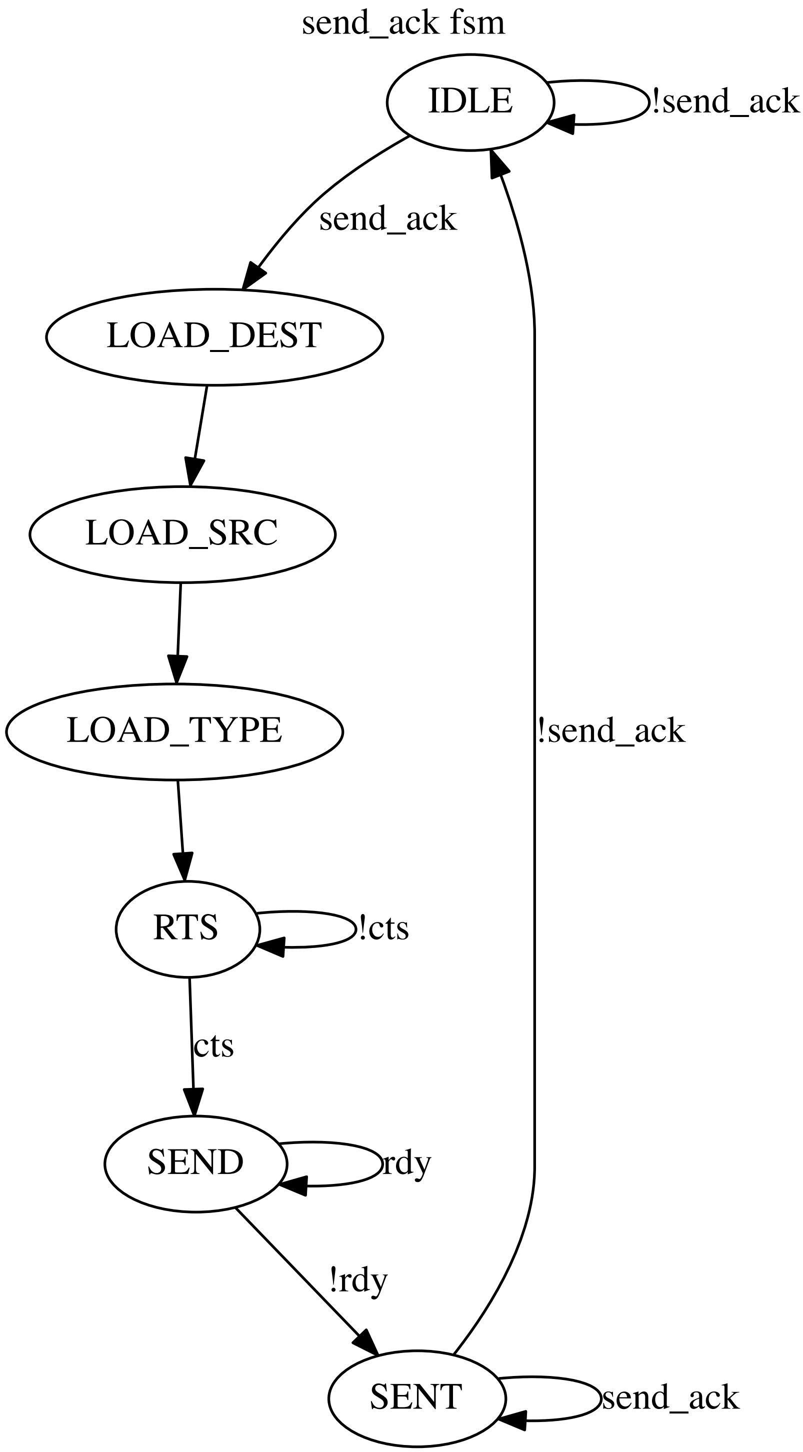


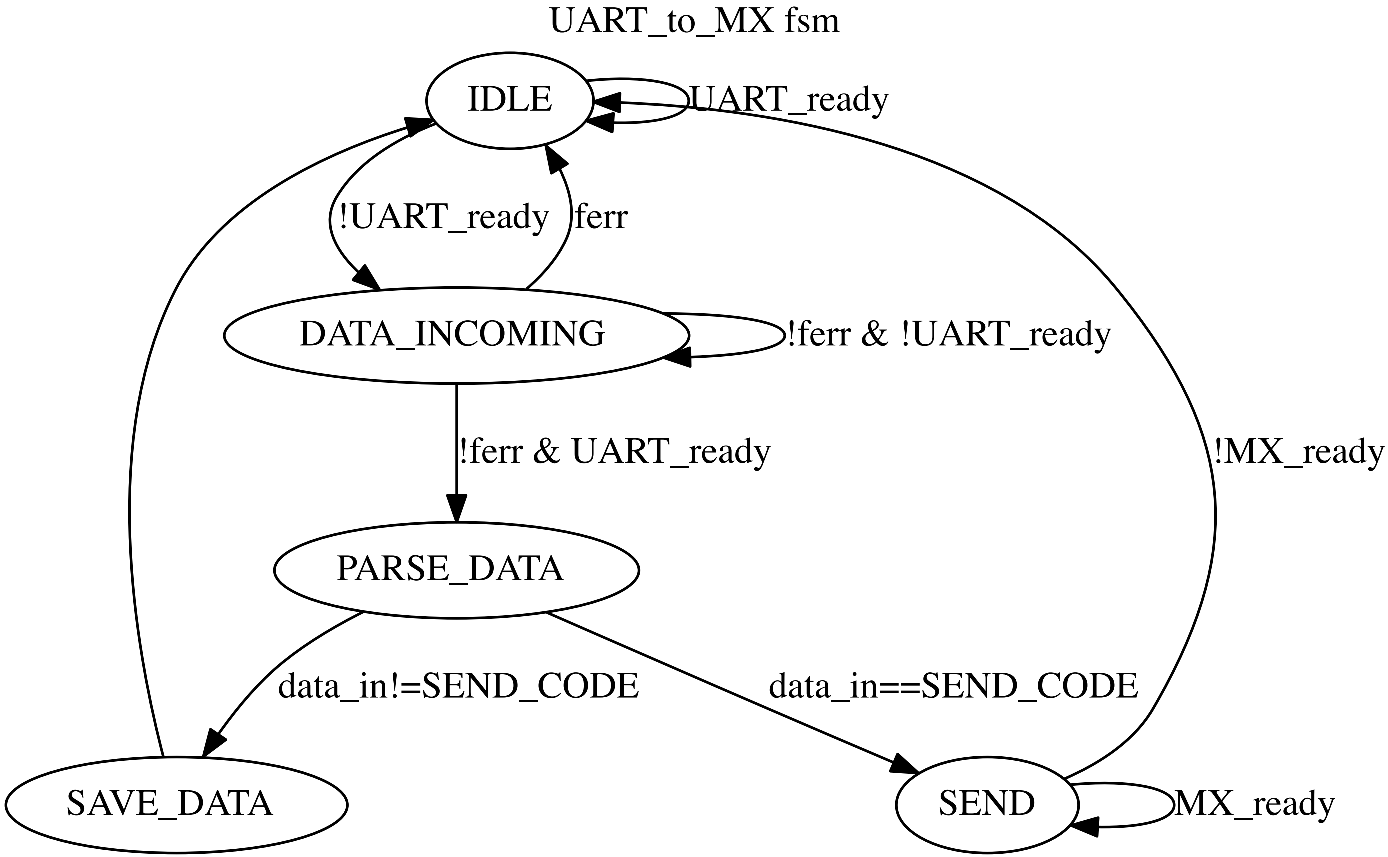

Frame_type fsm
IDLE
byte_count==3
CHECK_FRAME_TYPE
din==0x30
din==0x31
din==0x32
din==0x33
default
xsnd
TYPE_0
TYPE_1
TYPE_2
TYPE_3
UNKNOWN
WAIT_FOR_SEND
!xsnd


get_dest_addr fsm
IDLE
byte_count!=1
byte_count==1
STORE_DATA
xsnd
WAIT_FOR_SEND
!xsnd


retry fsm
IDLE
frame_type!=2
frame_type==2
good_ack | exceed_retry
WAIT
retry
RETRY
RESEND


Rx_mx_to_tx_uart fsm
IDLE
!rrdy
rrdy
DATA_TO_SEND
!rdy
!rrdy
rdy
SEND_DATA
rrdy
!rrdy
GET_NEXT_DATA
rrdy
PAUSE
CHECK_EMPTY


send_ack fsm
IDLE
!send_ack
send_ack
LOAD_DEST
LOAD_SRC
LOAD_TYPE
!send_ack
RTS
!cts
cts
SEND
rdy
!rdy
SENT
send_ack


UART_to_MX fsm
IDLE
UART_ready
!UART_ready
ferr
DATA_INCOMING
!ferr & !UART_ready
!ferr & UART_ready
PARSE_DATA
!MX_ready
data_in!=SEND_CODE
data_in==SEND_CODE
SAVE_DATA
SEND
MX_ready