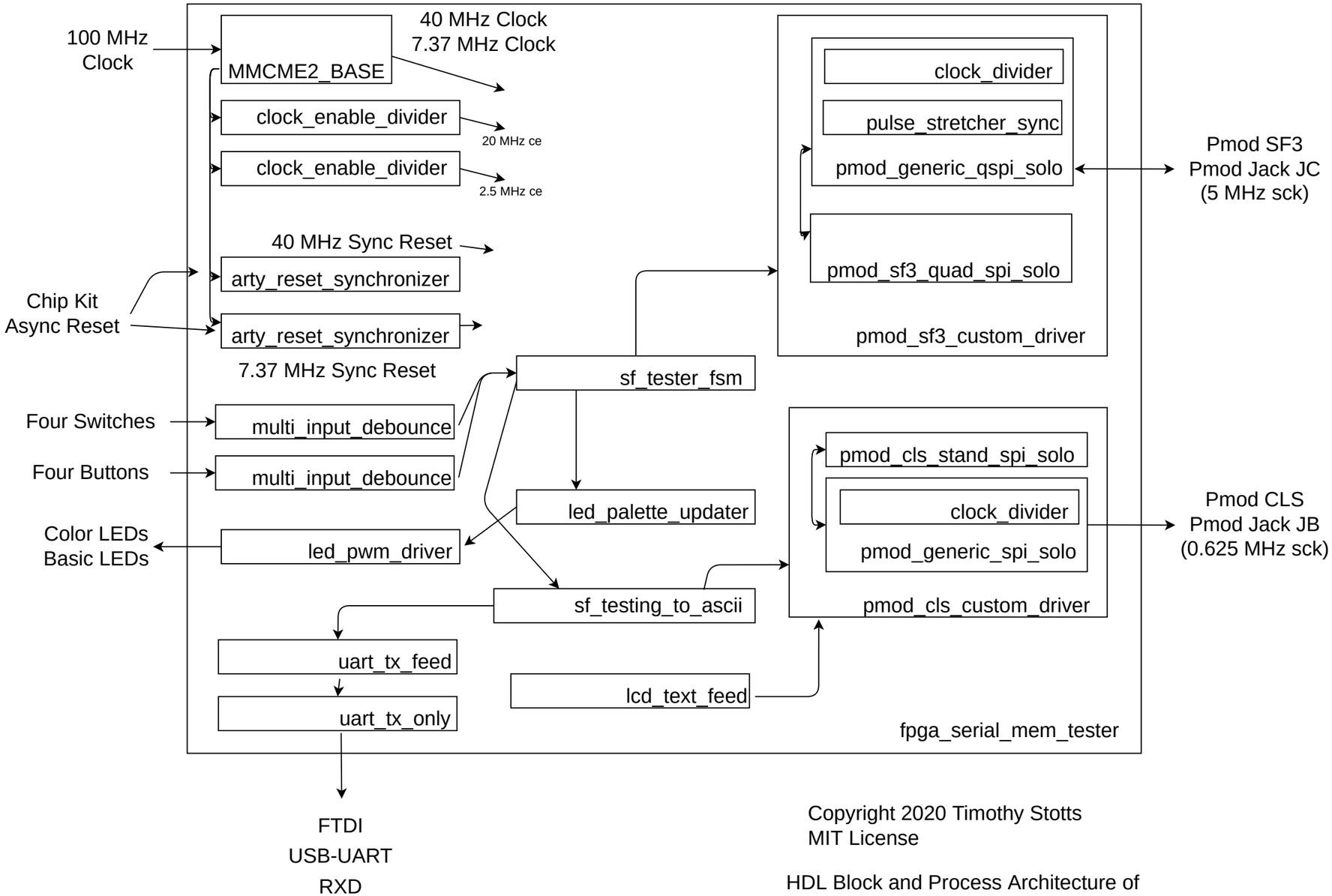
FPGA Serial Mem Tester, Version 1

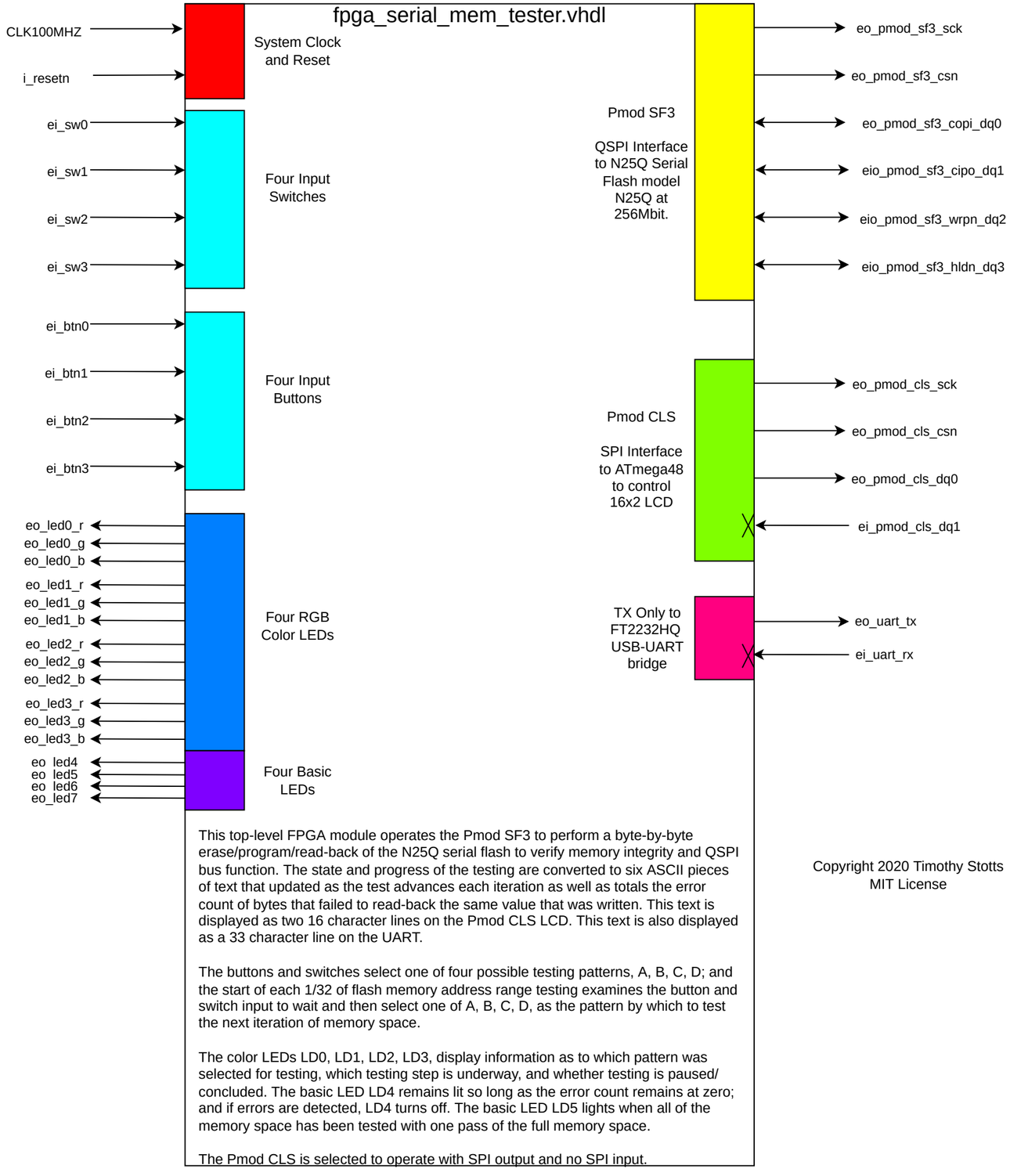
by Timothy Stotts

Hosted at:
https://github.com/timothystotts/fpga-serial-mem-tester-1

SF-Tester-Design-Diagrams document revision 12A



HDL Block and Process Architecture of the fpga-serial-mem-tester-1 design.

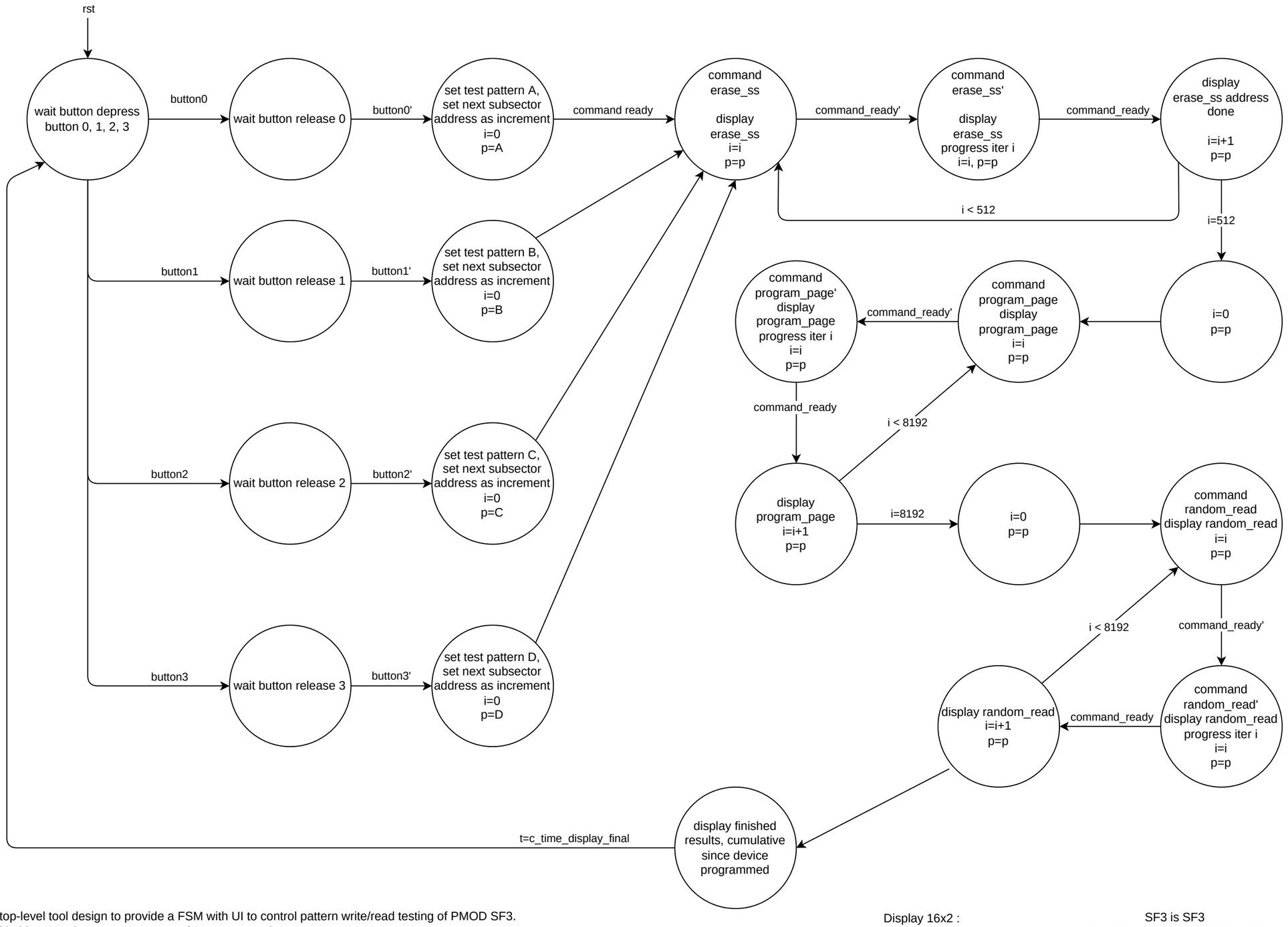


This top-level FPGA module operates the Pmod SF3 to perform a byte-by-byte erase/program/read-back of the N25Q serial flash to verify memory integrity and QSPI bus function. The state and progress of the testing are converted to six ASCII pieces of text that updated as the test advances each iteration as well as totals the error count of bytes that failed to read-back the same value that was written. This text is displayed as two 16 character lines on the Pmod CLS LCD. This text is also displayed as a 33 character line on the UART.

The buttons and switches select one of four possible testing patterns, A, B, C, D; and the start of each 1/32 of flash memory address range testing examines the button and switch input to wait and then select one of A, B, C, D, as the pattern by which to test the next iteration of memory space.

The color LEDs LD0, LD1, LD2, LD3, display information as to which pattern was selected for testing, which testing step is underway, and whether testing is paused/ concluded. The basic LED LD4 remains lit so long as the error count remains at zero; and if errors are detected, LD4 turns off. The basic LED LD5 lights when all of the memory space has been tested with one pass of the full memory space.

The Pmod CLS is selected to operate with SPI output and no SPI input.



A top-level tool design to provide a FSM with UI to control pattern write/read testing of PMOD SF3.
With this FSM, there are 32 groups of 32 sectors a piece.

Refer to project source code for more details of the FSM.
This drawing is a simplified conceptual representation of a more complex machine.

Display 16x2 :

```
SF3 PA hADDRESS_
ERS ERR 67108864
```

SF3 is SF3
PA = P* or PA or PB or PC or PD
hADDRESS = h00100000
ERS = GO_ or ERS or PRO or TST or END
ERR is ERR
67108864 is decimal in range 00000000 to 67108864

rst,
dat_wr_cntidx=0
dat_rd_cntidx=0
test_pass='0'
test_done='0'
err_count=0
pattern_start=pattern_startval_a
pattern_incr=pattern_incrval_a
pattern_track=h00
addr_start=h00000000
start_at_zero='1'
i=0

WAIT_BUTTON_DEP
len_random_read=0
address_of_cmd=h00000000
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'

dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=((addr_start < last_starting_byte_addr)?'0':'1')

err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i

addr_start >= last_starting_bytet_addr |
((buttons_debounced /= a single one bit) &
(switches_debounced /= a_single_one_bit))

addr_start < last_starting_byte_addr &
(buttons_debounced="0001" |
switches_debounced="0001") -> WAIT_BUTTON_0_REL

addr_start < last_starting_byte_addr &
(buttons_debounced="0010" |
switches_debounced="0010") -> WAIT_BUTTON_1_REL

addr_start < last_starting_byte_addr &
(buttons_debounced="0100" |
switches_debounced="0100") -> WAIT_BUTTON_2_REL

addr_start < last_starting_byte_addr &
(buttons_debounced="1000" |
switches_debounced="1000") -> WAIT_BUTTON_3_REL

WAIT_BUTTON[0-3]_REL
len_random_read=0
address_of_cmd=h00000000
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'

dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i

buttons_debounced/="0000"

buttons_debounced="0000"

if BUTTTON0 -> PATTERNA
if BUTTON1 -> PATTERNB
if BUTTON2 -> PATTERNC
if BUTTON3 -> PATTERND

SET_PATTERN_[A-D]
len_random_read=0
address_of_cmd=h00000000
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'

dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_startval_[a-d]
pattern_incr=pattern_incrval_[a-d]
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i

command_ready='0'

command_ready='1'

if PATTERNA -> START_ADDR_A
if PATTERNB -> START_ADDR_B
if PATTERNC -> START_ADDR_C
if PATTERND -> START_ADDR_D

start_at_zero='0' &
addr_start >= last_starting_byte_addr

SET_START_ADDR_[A-D]
len_random_read=0
address_of_cmd=h00000000
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'

dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done*=(start_at_zero?'0':(addr_start<last_starting_byte_addr?'0':'1'))

addr_start*=(start_at_zero?h00000000:(addr_start<last_starting_byte_addr?
addr_start + per_iteration_byte_count:addr_start))

err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
start_at_zero*='0'
i=0

start_at_zero='1' |
addr_start<last_starting_byte_addr

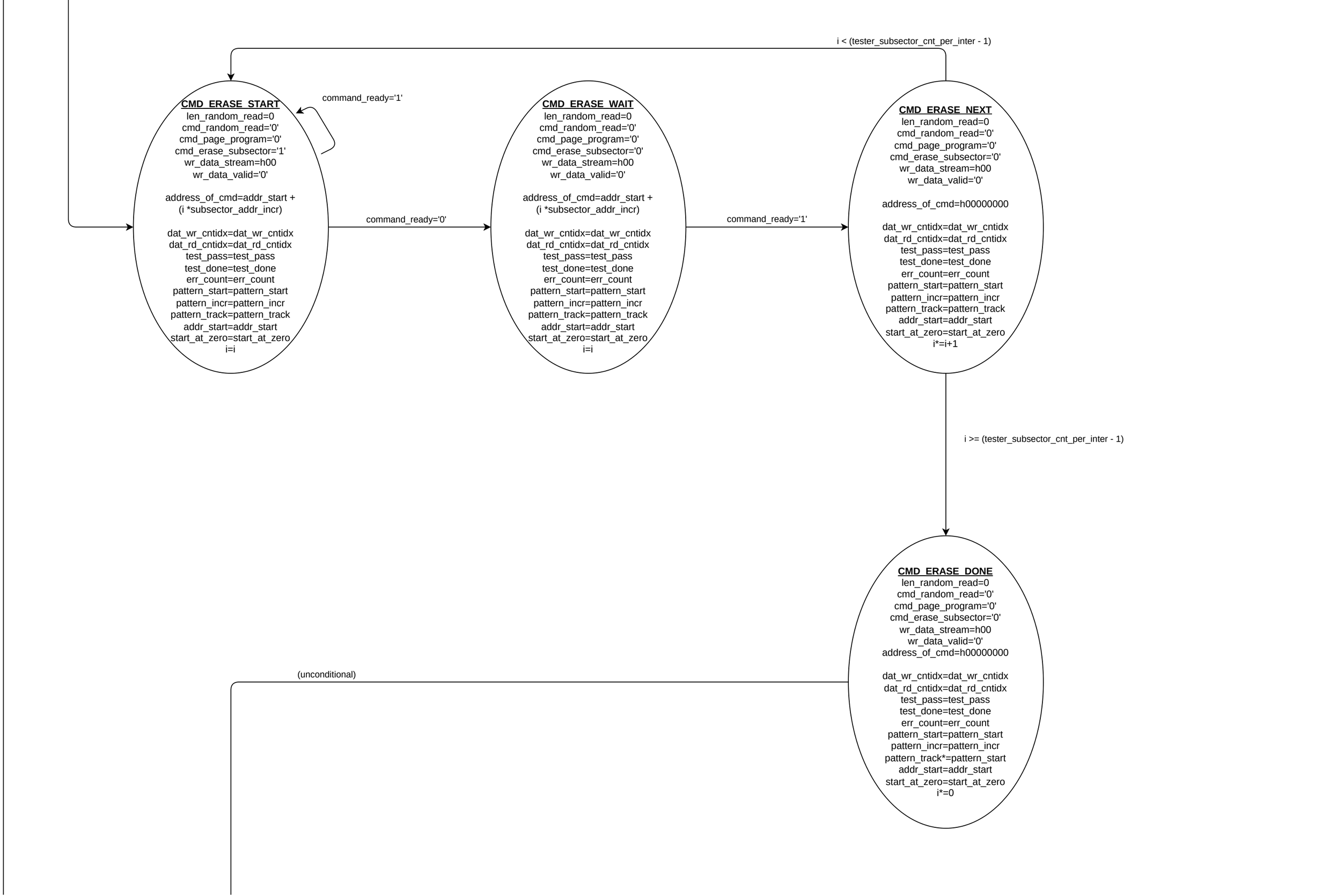
if START_ADDR_A -> SET_START_WAIT_A
if START_ADDR_B -> SET_START_WAIT_B
if START_ADDR_C -> SET_START_WAIT_C
if START_ADDR_D -> SET_START_WAIT_D

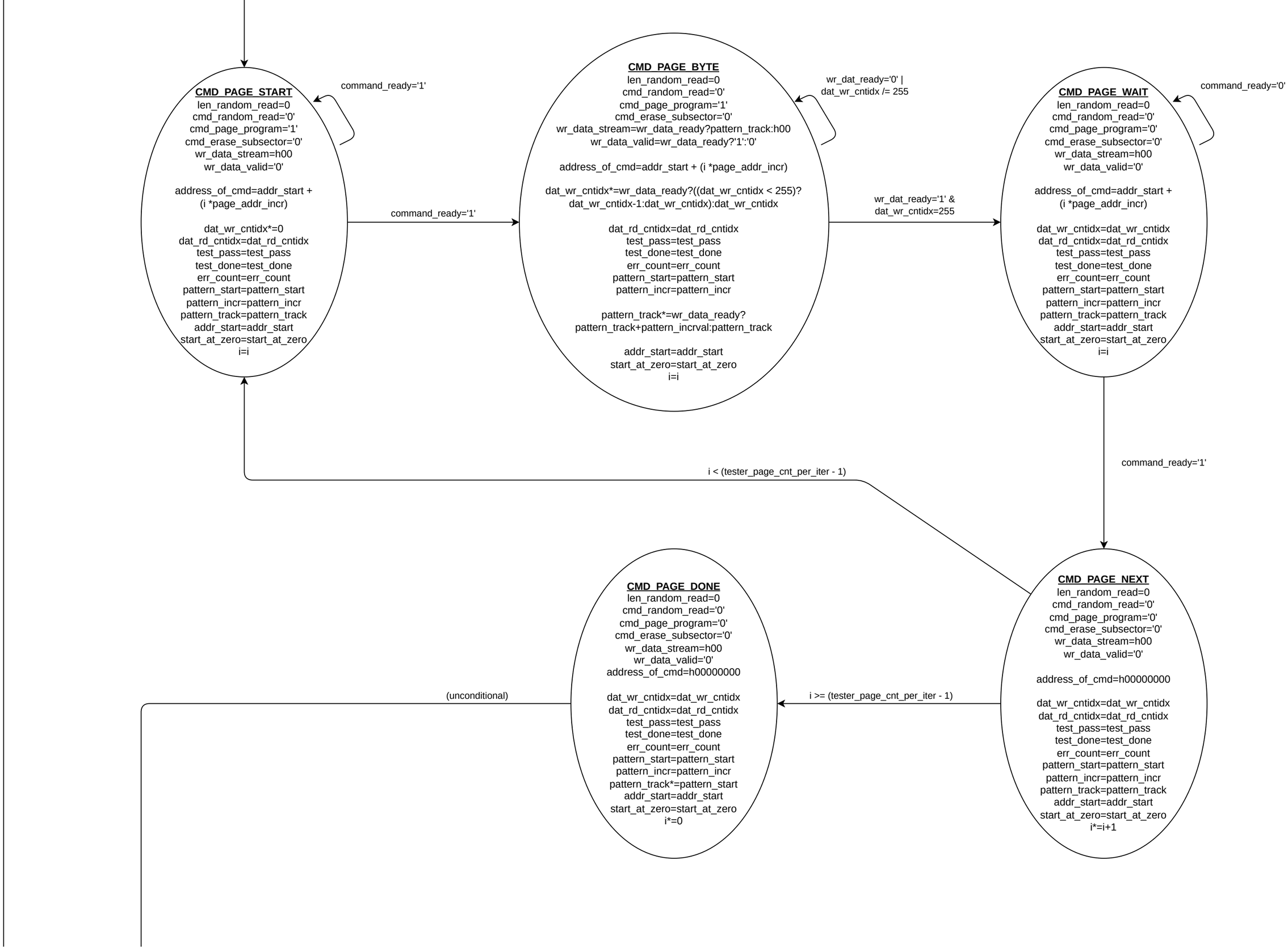
t/=t_max / 2

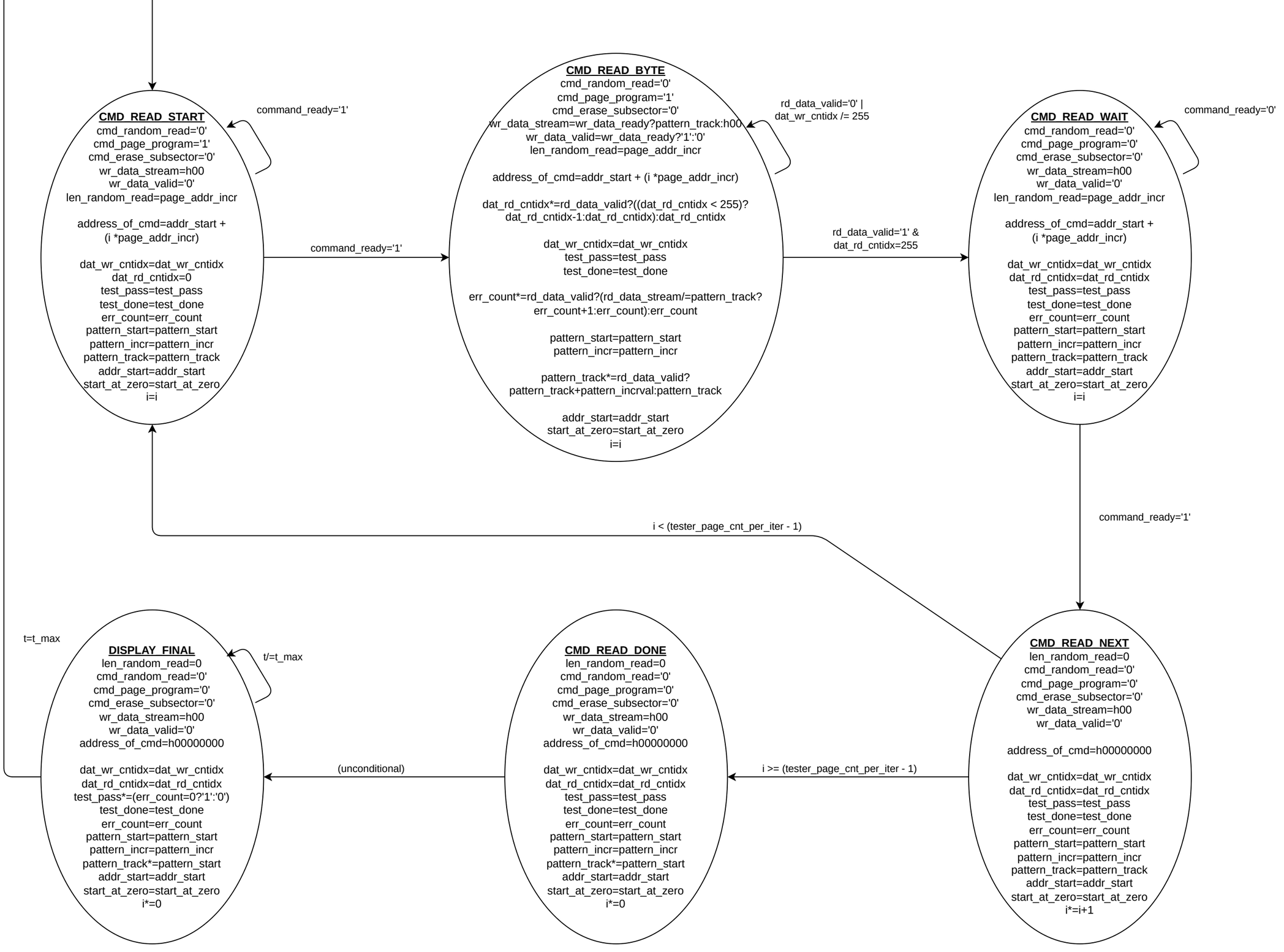
SET_START_WAIT_[A-D]
len_random_read=0
address_of_cmd=h00000000
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'

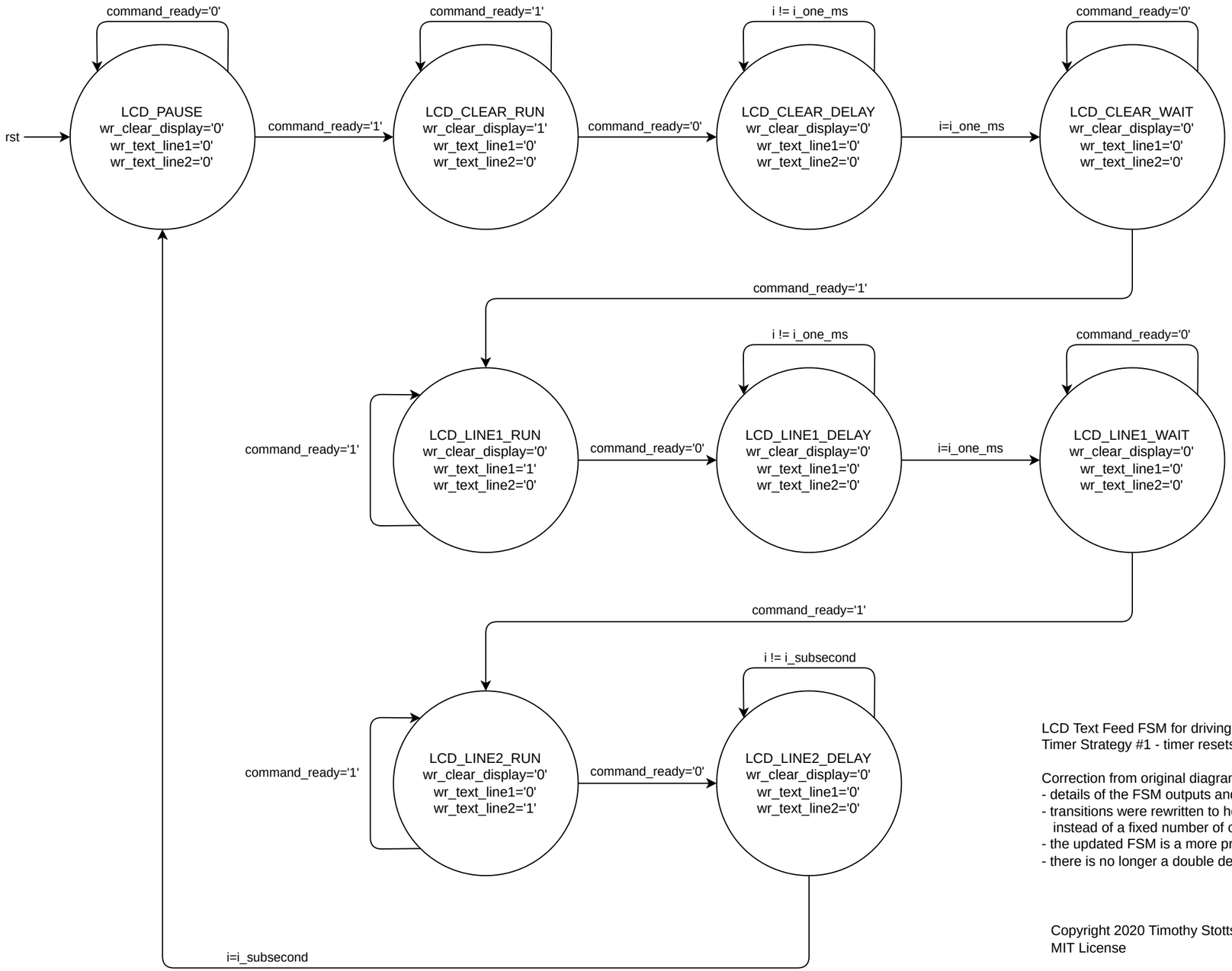
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i

t=t_max / 2


i < (tester_subsector_cnt_per_inter - 1)
CMD_ERASE_START
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='1'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=addr_start +
(i *subsector_addr_incr)
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='1'
command_ready='0'
CMD_ERASE_WAIT
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=addr_start +
(i *subsector_addr_incr)
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='1'
CMD_ERASE_NEXT
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i*=i+1
i >= (tester_subsector_cnt_per_inter - 1)
CMD_ERASE_DONE
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=pattern_start
addr_start=addr_start
start_at_zero=start_at_zero
i*=0
(unconditional)

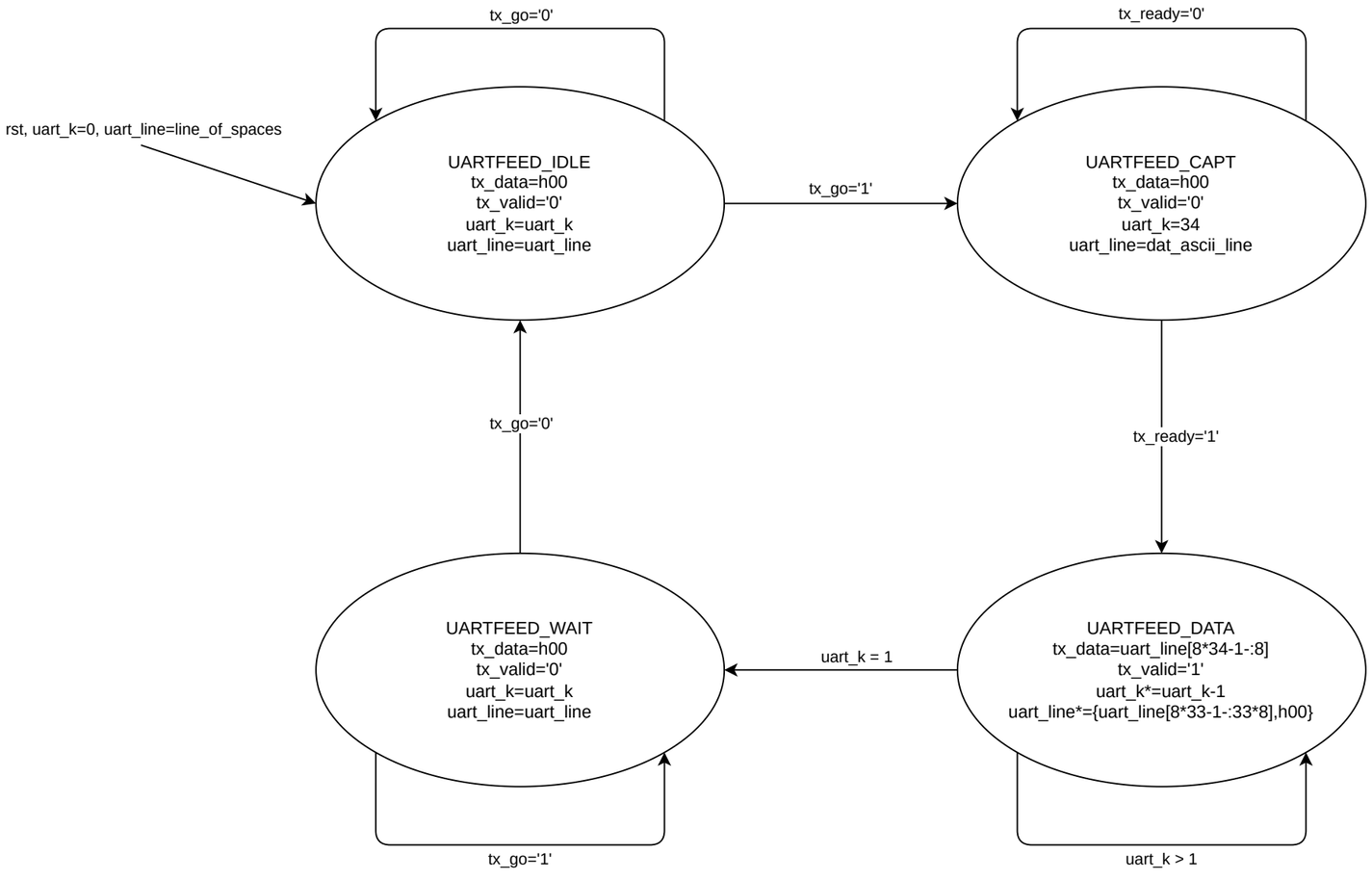

CMD_PAGE_START
len_random_read=0
cmd_random_read='0'
cmd_page_program='1'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=addr_start +
(i *page_addr_incr)
dat_wr_cntidx*=0
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='1'
command_ready='1'
CMD_PAGE_BYTE
len_random_read=0
cmd_random_read='0'
cmd_page_program='1'
cmd_erase_subsector='0'
wr_data_stream=wr_data_ready?pattern_track:h00
wr_data_valid=wr_data_ready?'1':'0'
address_of_cmd=addr_start + (i *page_addr_incr)
dat_wr_cntidx*=wr_data_ready?((dat_wr_cntidx < 255)?
dat_wr_cntidx-1:dat_wr_cntidx):dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=wr_data_ready?
pattern_track+pattern_incrval:pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
wr_dat_ready='0' |
dat_wr_cntidx /= 255
wr_dat_ready='1' &
dat_wr_cntidx=255
CMD_PAGE_WAIT
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=addr_start +
(i *page_addr_incr)
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='0'
command_ready='1'
i < (tester_page_cnt_per_iter - 1)
CMD_PAGE_NEXT
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i*=i+1
i >= (tester_page_cnt_per_iter - 1)
CMD_PAGE_DONE
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=pattern_start
addr_start=addr_start
start_at_zero=start_at_zero
i*=0
(unconditional)


CMD_READ_START
cmd_random_read='0'
cmd_page_program='1'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
len_random_read=page_addr_incr
address_of_cmd=addr_start +
(i *page_addr_incr)
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=0
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='1'
command_ready='1'
CMD_READ_BYTE
cmd_random_read='0'
cmd_page_program='1'
cmd_erase_subsector='0'
wr_data_stream=wr_data_ready?pattern_track:h00
wr_data_valid=wr_data_ready?'1':'0'
len_random_read=page_addr_incr
address_of_cmd=addr_start + (i *page_addr_incr)
dat_rd_cntidx*=rd_data_valid?((dat_rd_cntidx < 255)?
dat_rd_cntidx-1:dat_rd_cntidx):dat_rd_cntidx
dat_wr_cntidx=dat_wr_cntidx
test_pass=test_pass
test_done=test_done
err_count*=rd_data_valid?(rd_data_stream/=pattern_track?
err_count+1:err_count):err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=rd_data_valid?
pattern_track+pattern_incrval:pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
rd_data_valid='0' |
dat_wr_cntidx /= 255
rd_data_valid='1' &
dat_rd_cntidx=255
CMD_READ_WAIT
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
len_random_read=page_addr_incr
address_of_cmd=addr_start +
(i *page_addr_incr)
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i=i
command_ready='0'
command_ready='1'
i < (tester_page_cnt_per_iter - 1)
t=t_max
DISPLAY_FINAL
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass*=(err_count=0?'1':'0')
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=pattern_start
addr_start=addr_start
start_at_zero=start_at_zero
i*=0
t/=t_max
(unconditional)
CMD_READ_DONE
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track*=pattern_start
addr_start=addr_start
start_at_zero=start_at_zero
i*=0
i >= (tester_page_cnt_per_iter - 1)
CMD_READ_NEXT
len_random_read=0
cmd_random_read='0'
cmd_page_program='0'
cmd_erase_subsector='0'
wr_data_stream=h00
wr_data_valid='0'
address_of_cmd=h00000000
dat_wr_cntidx=dat_wr_cntidx
dat_rd_cntidx=dat_rd_cntidx
test_pass=test_pass
test_done=test_done
err_count=err_count
pattern_start=pattern_start
pattern_incr=pattern_incr
pattern_track=pattern_track
addr_start=addr_start
start_at_zero=start_at_zero
i*=i+1



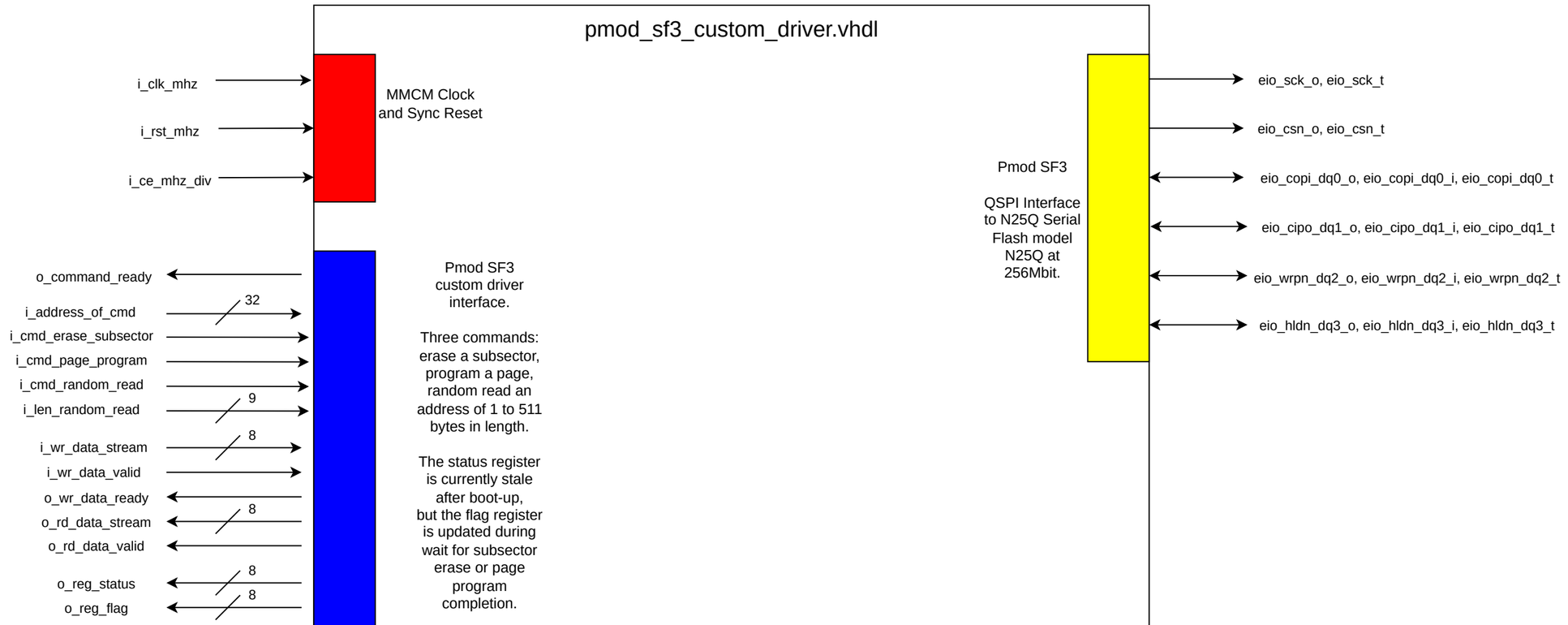
LCD Text Feed FSM for driving the commands of the custom Pmod CLS driver.
Timer Strategy #1 - timer resets on state change and runs from zero to i_subsecond .

Correction from original diagram:
- details of the FSM outputs and timer "i"
- transitions were rewritten to hold commands according to command_ready handshake instead of a fixed number of clock cycles during a state
- the updated FSM is a more proper FSM diagram design
- there is no longer a double delay between the end of Line 2 and the start of Clear.



UART TX Feed FSM

This FSM feeds the TX FIFO of the uart_tx_only module.
The data to feed to the TX FIFO is always a 34 8-bit character line of ASCII text.
The tx_go input is triggered by the corresponding wr_clear_display pulse on the Pmod CLS custom driver, such that the UART TX Feed occurs when the LCD is starting to update on the FSM cycle of that driver.



The Pmod SF3 custom driver operates by booting the N25Q and waiting for a strobe on one of the three possible operation commands.

Upon receiving a pulse on the erase subsector command, the driver reads the address of command and issues a 32-bit address subsector erase. Refer the to the N25Q datasheet as to how the address is interpreted in 32-bit addressing mode.

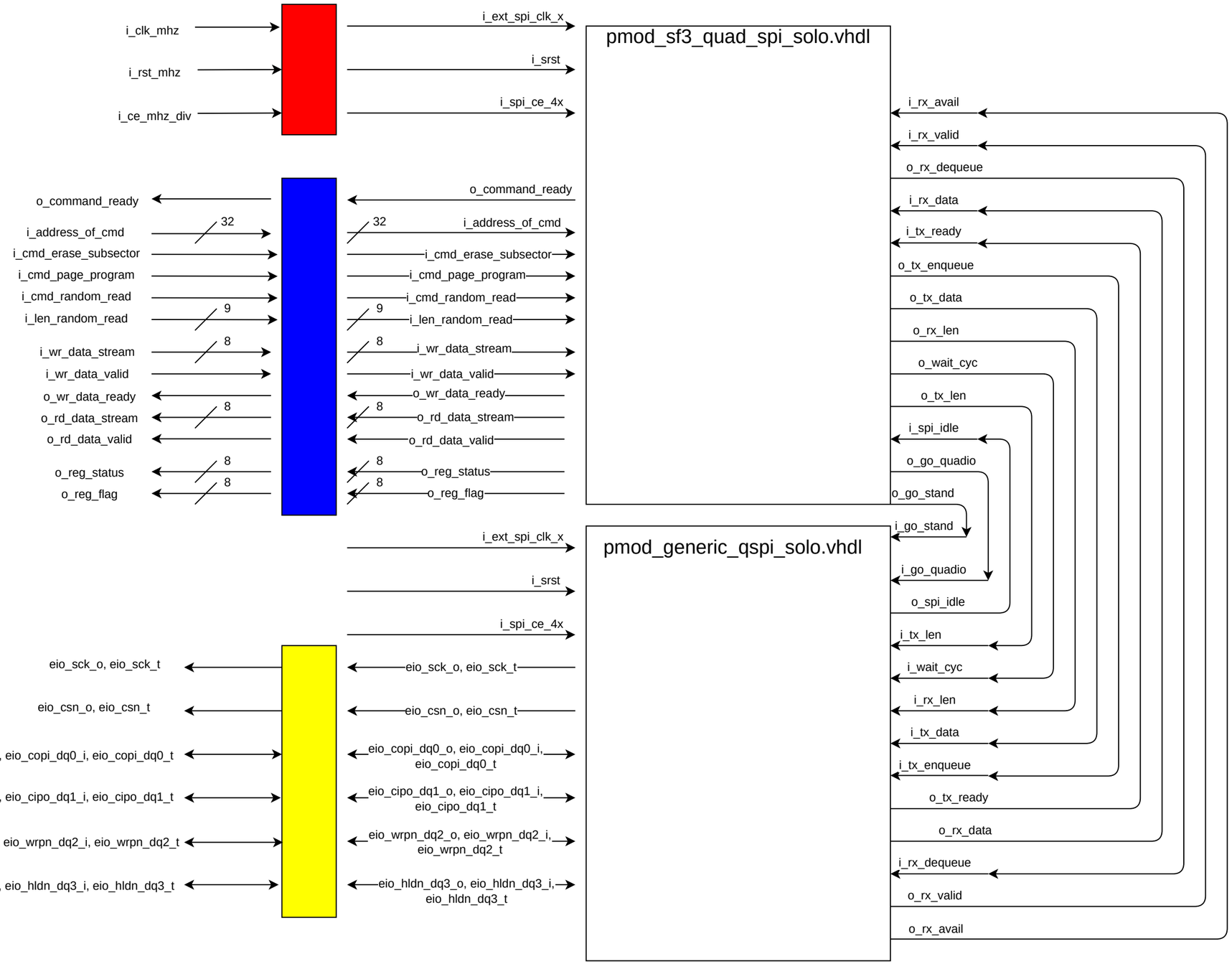
Upon receiving a pulse on the page program command, the driver reads the address of command and issues a 32-bit address page program of 256 bytes of data. The data must be enqueued in advance on the write stream signals. Refer the to the N25Q datasheet as to how the address is interpreted in 32-bit addressing mode.

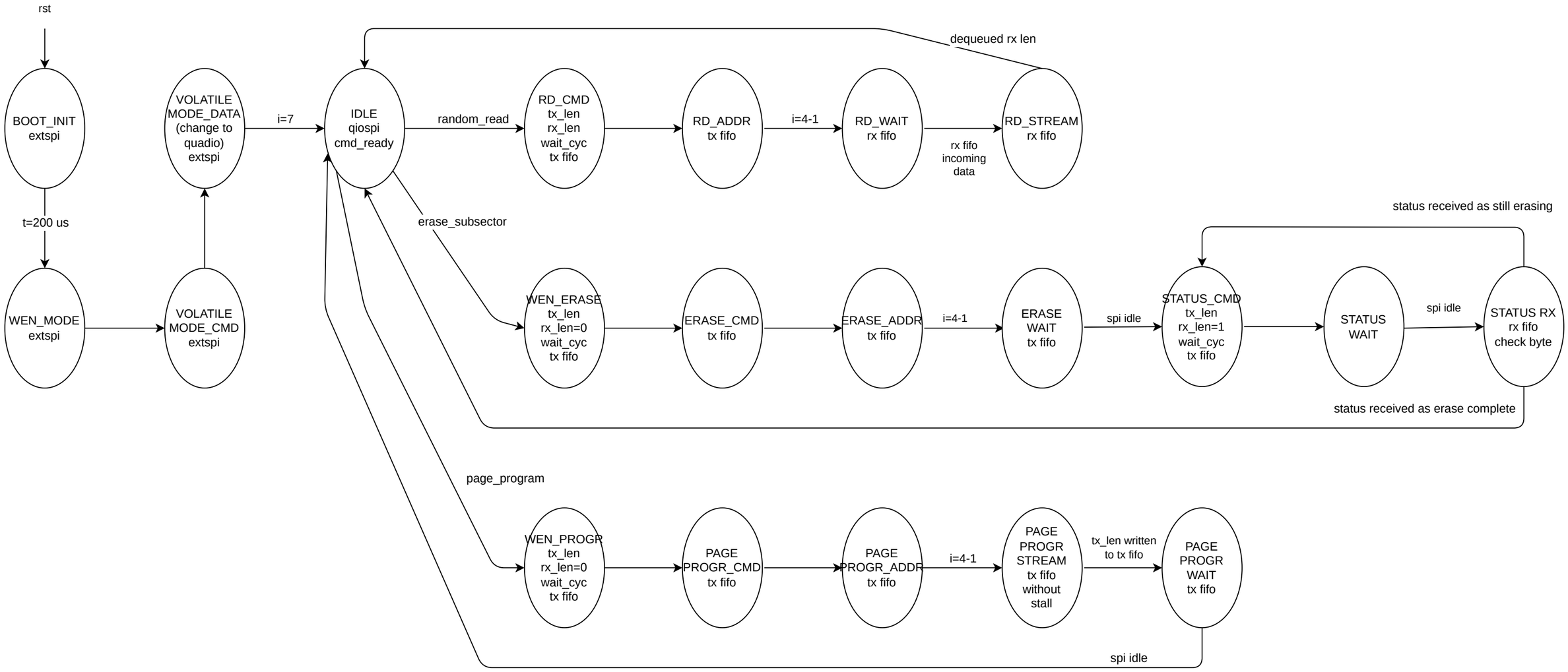
Upon receiving a pulse on the random read command, the driver reads the address of command and issues a 32-bit address random read of random read length bytes, ranging from 1 to 511. The data is output on the read data stream byte by byte as received from the memory.

During the wait for completion of an erase command or page program command, the N25Q flag register is updated. Refer to the N25Q datasheet as to the meaning of each of the eight bits of this register.

Note that the SPI machine operated by this driver is a generic QSPI driver operating in Enhanced SPI mode. Quad IO SPI mode is implemented but incompletely coded, and is thus not used with this design revision.

pmod_sf3_custom_driver.vhdl

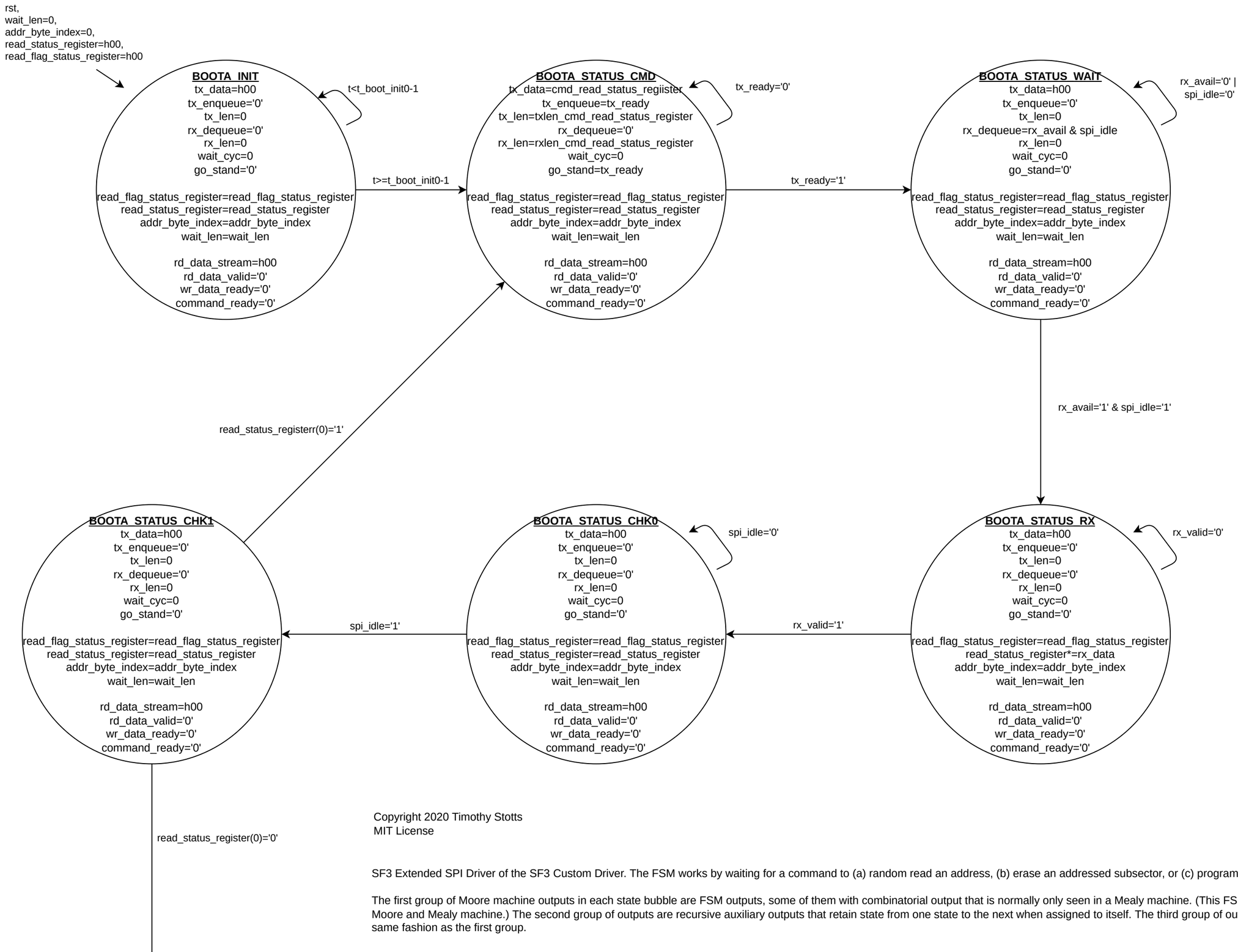




In each transition, tx_len and rx_len are to be multiplied by 8 from the FSM input signals,
as it only makes sense to input into the FSM a byte count, while the FSM requires transitioning
based upon a bit count.

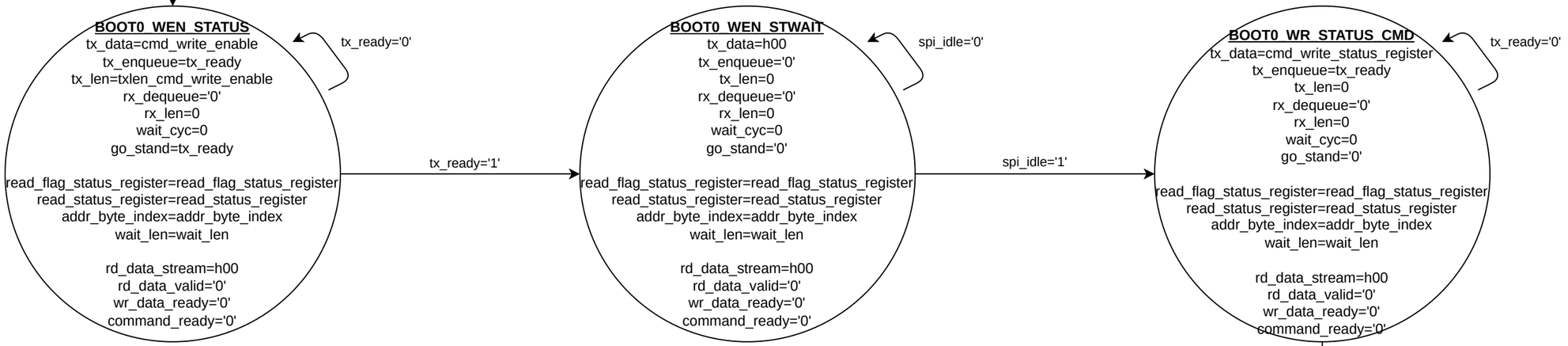
N25Q chip FSM for communicating wtih the SPI Machine, with the N25Q as only SPI slave on the bus.

Refer to project source code for more details of the FSM.
This drawing is a simplified conceptual representation of a more complex machine.

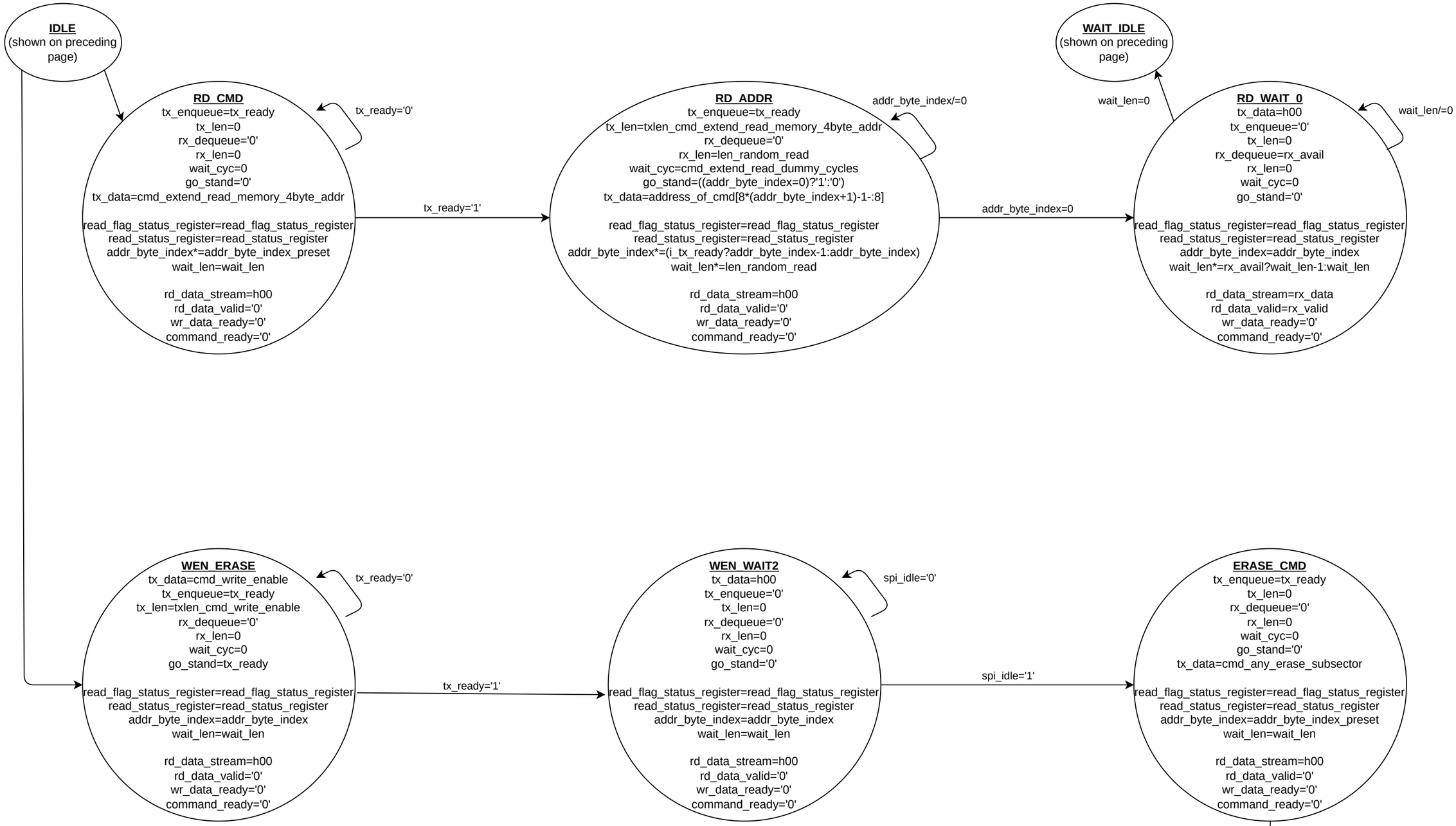


SF3 Extended SPI Driver of the SF3 Custom Driver. The FSM works by waiting for a command to (a) random read an address, (b) erase an addressed subsector, or (c) program an addressed page.

The first group of Moore machine outputs in each state bubble are FSM outputs, some of them with combinatorial output that is normally only seen in a Mealy machine. (This FSM could be said to be a hybrid Moore and Mealy machine.) The second group of outputs are recursive auxiliary outputs that retain state from one state to the next when assigned to itself. The third group of outputs are FSM outputs in same fashion as the first group.


BOOT0_WEN_STATUS
tx_data=cmd_write_enable
tx_enqueue=tx_ready
tx_len=txlen_cmd_write_enable
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=tx_ready
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
BOOT0_WEN_STWAIT
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
BOOT0_WR_STATUS_CMD
tx_data=cmd_write_status_register
tx_enqueue=tx_ready
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
BOOT0_FLAGST_CMD
tx_data=cmd_read_flag_status_register
tx_enqueue=tx_ready
tx_len=txlen_cmd_read_flag_status_register
rx_dequeue='0'
rx_len=rxlen_cmd_read_flag_status_register
wait_cyc=0
go_stand=tx_ready
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
BOOT0_IDLE_STWAIT
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
BOOT0_WR_STATUS_DAT
tx_data=dat_status_reg_as_pmod_st3
tx_enqueue=tx_ready & spi_idle
tx_len=txlen_cmd_write_status_register
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=tx_ready & spi_idle
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0' |
spi_idle='0'
tx_ready='1' & spi_idle='1'


BOOT0_FLAGST_WAIT
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue=rx_avail & spi_idle
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
rx_avail='0' | spi_idle='0'
rx_avail='1' & spi_idle='1'
BOOT0_FLAGST_RX
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register*=rx_data
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
rx_valid='0'
rx_valid='1'
BOOT0_FLAGST_CHK0
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
read_flag_status_register(7)='0'
BOOT0_FLAGST_CHK1
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
WAIT_IDLE
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
IDLE
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready=spi_idle
cmd_random_read='0 & cmd_erase_subsector='0' & cmd_page_program='0'
cmd_random_read='1
cmd_random_read='0' & cmd_erase_subsector='1'
cmd_random_read='0' & cmd_erase_subsector='0' & cmd_page_program='1'
RD_CMD
(shown on following page)
WEN_ERASE
(shown on following page)
WEN_PROG
(shown on following page)


IDLE
(shown on preceding page)
RD_CMD
tx_enqueue=tx_ready
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
tx_data=cmd_extend_read_memory_4byte_addr
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index*=addr_byte_index_preset
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
RD_ADDR
tx_enqueue=tx_ready
tx_len=txlen_cmd_extend_read_memory_4byte_addr
rx_dequeue='0'
rx_len=len_random_read
wait_cyc=cmd_extend_read_dummy_cycles
go_stand=((addr_byte_index=0)?'1':'0')
tx_data=address_of_cmd[8*(addr_byte_index+1)-1-:8]
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index*=(i_tx_ready?addr_byte_index-1:addr_byte_index)
wait_len*=len_random_read
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
addr_byte_index/=0
addr_byte_index=0
WAIT_IDLE
(shown on preceding page)
wait_len=0
RD_WAIT_0
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue=rx_avail
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len*=rx_avail?wait_len-1:wait_len
rd_data_stream=rx_data
rd_data_valid=rx_valid
wr_data_ready='0'
command_ready='0'
wait_len/=0
WEN_ERASE
tx_data=cmd_write_enable
tx_enqueue=tx_ready
tx_len=txlen_cmd_write_enable
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=tx_ready
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
WEN_WAIT2
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
ERASE_CMD
tx_enqueue=tx_ready
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
tx_data=cmd_any_erase_subsector
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index_preset
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'


ERASE_ADDR
tx_enqueue=tx_ready
tx_len=txlen_cmd_any_erase_subsector
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=((addr_byte_index=0)?'1':'0')
tx_data=address_of_cmd[8*(addr_byte_index+1)-1-:8]
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index*=(i_tx_ready?addr_byte_index-1:addr_byte_index)
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
addr_byte_index/=0
addr_byte_index=0
ERASE_WAIT
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'
FLAGST_CMD
tx_enqueue=tx_ready
rx_dequeue='0'
wait_cyc=0
go_stand=tx_ready
tx_len=txlen_cmd_read_flag_status_register
rx_len=rxlen_cmd_read_flag_status_register
tx_data=cmd_read_flag_status_register
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
tx_ready='0'
tx_ready='1'
FLAGST_WAIT
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue=rx_avail & spi_idle
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
rx_avail='0' |
spi_idle='0'
rx_avail='1' & spi_idle='1'
FLAGST_RX
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register*=rx_data
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
rx_valid='0'
rx_valid='1'
FLAGST_CHK0
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
spi_idle='0'
spi_idle='1'

WAIT_IDLE
(shown on preceding page)

read_flag_status_register(7)='1'

FLAGST_CHK1
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'

read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len

rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'

read_flag_status_register(7)='0'

FLAGST_CMD
(shown on preceding page)

IDLE
(shown on preceding page)

WEN_PROG
tx_data=cmd_write_enable
tx_enqueue=tx_ready
tx_len=txlen_cmd_write_enable
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=tx_ready

read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len

rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'

tx_ready='0'

tx_ready='1'

WEN_WAIT3
tx_data=h00
tx_enqueue='0'
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'

read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
wait_len=wait_len

rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'

spi_idle='0'

spi_idle='1'

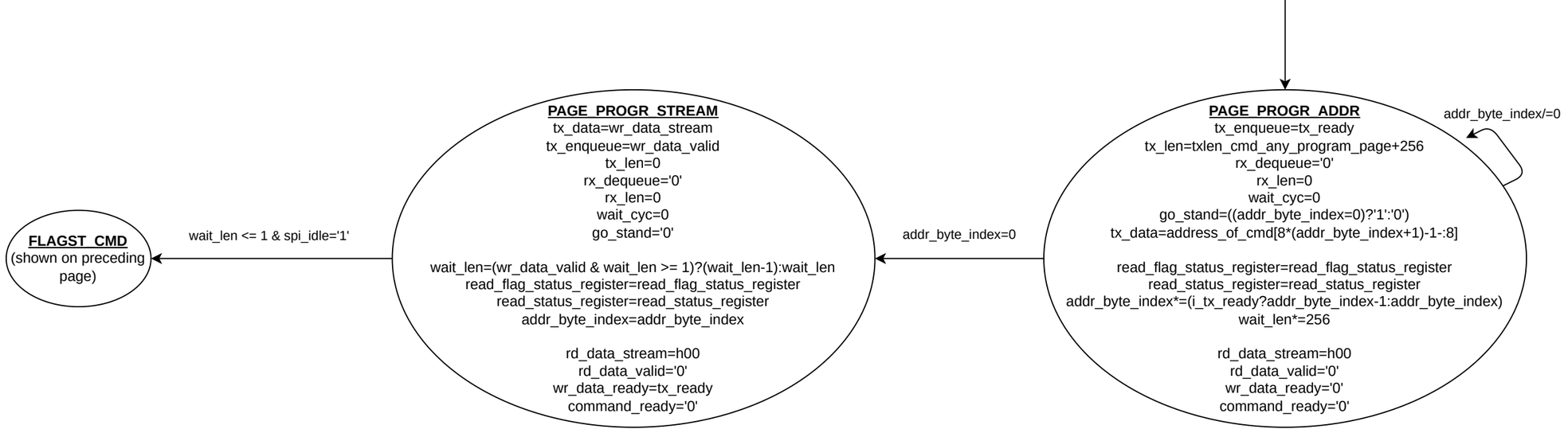
PAGE_PROGR_CMD
tx_enqueue=tx_ready
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
tx_data=cmd_any_page_program

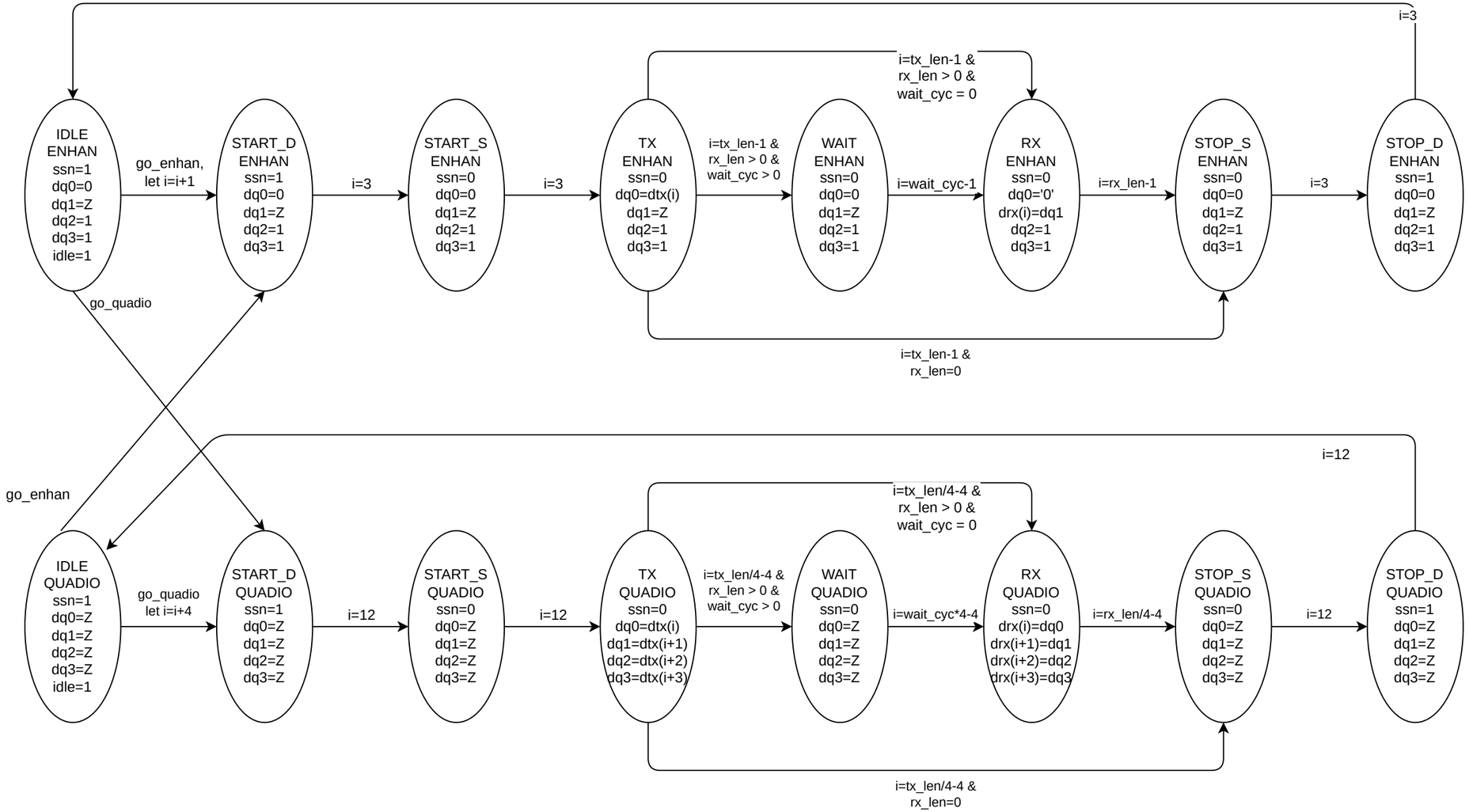
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index_preset
wait_len=wait_len

rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'

tx_ready='0'

tx_ready='1'

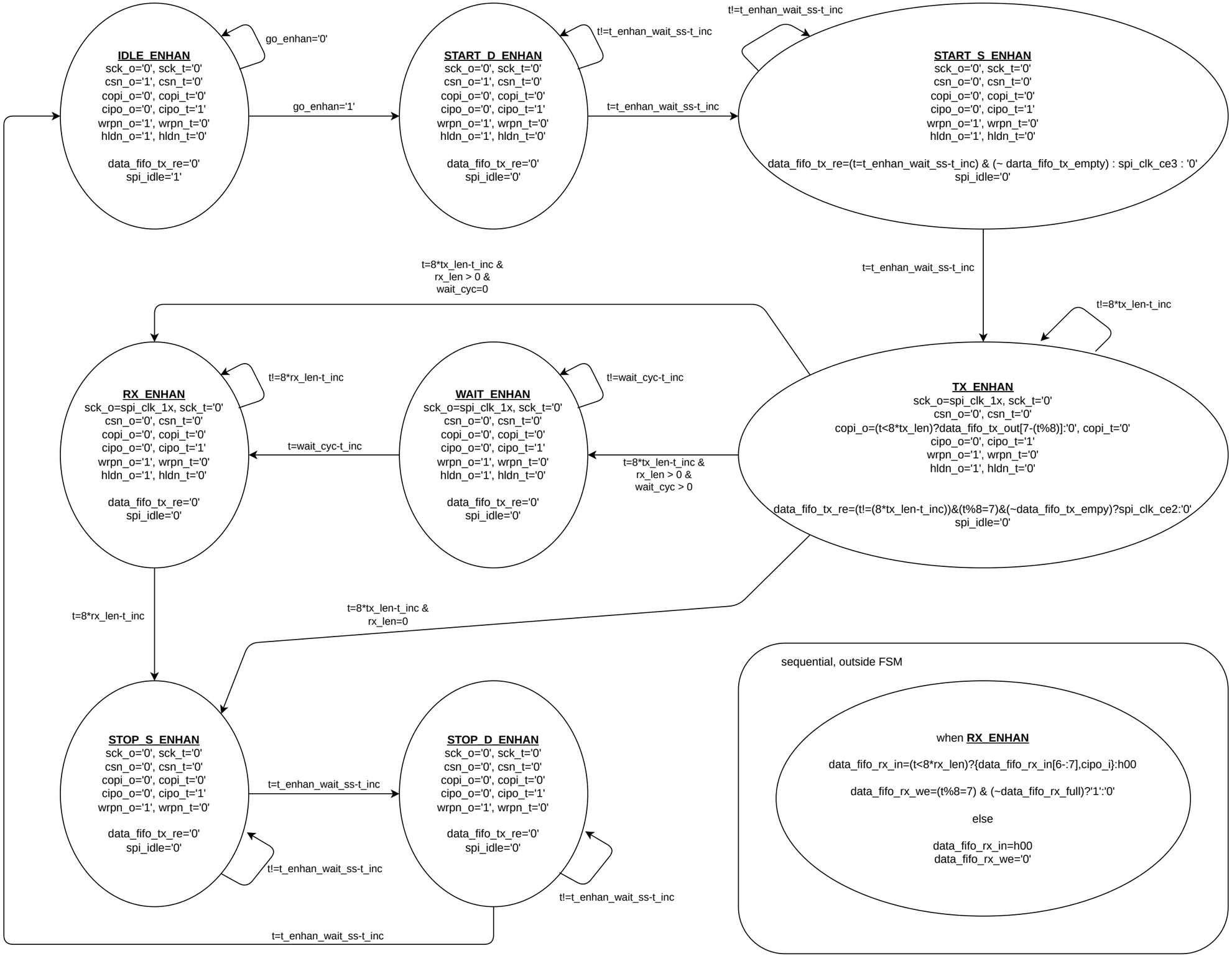

PAGE_PROGR_ADDR
tx_enqueue=tx_ready
tx_len=txlen_cmd_any_program_page+256
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand=((addr_byte_index=0)?'1':'0')
tx_data=address_of_cmd[8*(addr_byte_index+1)-1-:8]
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index*=(i_tx_ready?addr_byte_index-1:addr_byte_index)
wait_len*=256
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready='0'
command_ready='0'
addr_byte_index/=0
addr_byte_index=0
PAGE_PROGR_STREAM
tx_data=wr_data_stream
tx_enqueue=wr_data_valid
tx_len=0
rx_dequeue='0'
rx_len=0
wait_cyc=0
go_stand='0'
wait_len=(wr_data_valid & wait_len >= 1)?(wait_len-1):wait_len
read_flag_status_register=read_flag_status_register
read_status_register=read_status_register
addr_byte_index=addr_byte_index
rd_data_stream=h00
rd_data_valid='0'
wr_data_ready=tx_ready
command_ready='0'
wait_len <= 1 & spi_idle='1'
FLAGST_CMD
(shown on preceding page)



In each transition, tx_len and rx_len are to be multiplied by 8 from the FSM input signals,
as it only makes sense to input into the FSM a byte count, while the FSM requires transitioning
based upon a bit count.

Generic SPI FSM, with the only one SPI slave on the bus.

Refer to project source code for more details of the FSM.
This drawing is a simplified conceptual representation of a more complex machine.
Note that this drawing indicates operation of either Extended SPI or Quad IO SPI.
The source code only implemented Extended SPI due to the Quad IO for N25Q
requiring some transmission parts to only output data on MOSI and other parts
to be output using all four bits. This diagram shows the original design intent, but
ony Extended SPI was implemented. In other words, this diagram is incorrect.



Generic Enhanced SPI FSM, with only one SPI slave on the bus.

In each transition, tx_len and rx_len are to be multiplied by 8 from the FSM input signals, as it only makes sense to input into the FSM a byte count, while the FSM requires transitioning based upon a bit count.

The main FSM combinatorial operates all states, but processes data in the TX state when TX data is written out the copi port.

A side sequential process processes data only on the RX state when RX data is read from the cipo input port.

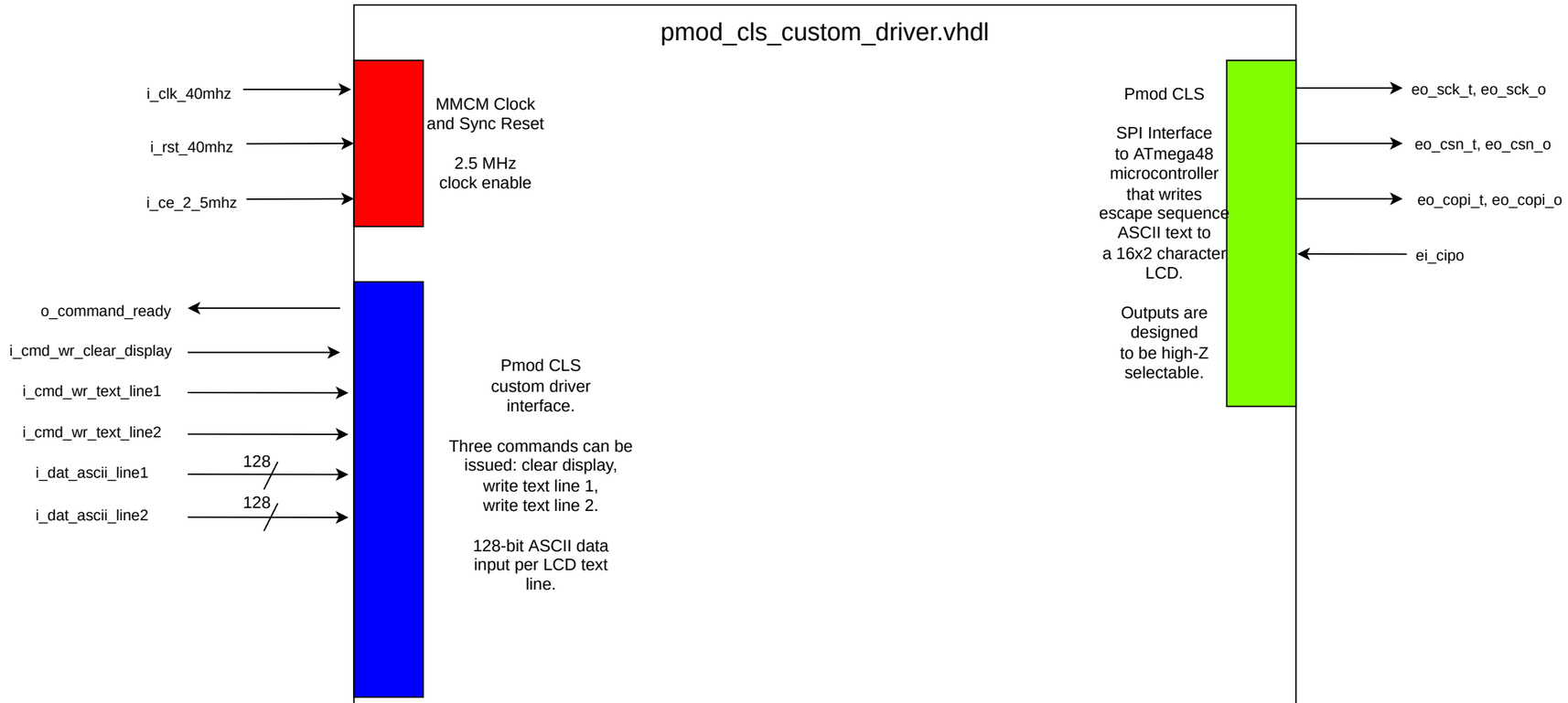
A general timer is used to time the SPI clock cycles (and thus bits). The timer is reset to zero at the transition from a state to a different state.

This machine could be considered a hybrid Moore and Mealy machine. The outputs during a state can change based upon timer and FIFO control inputs; but the diagram is drawn as a Moore machine.

The TX state controls reading byte by byte from a TX fifo and writes its bits to COPI until the timer has reached 8*tx_len cycles.

The RX standalone sequential controls reading a byte bit by bit from CIPO and then writing the full byte to a RX fifo when the full byte has been received.
The main FSM still controls when the FSM is in the **RX** state or has transitioned to **STOP_S** state.

Note that clock enables such as spi_clk_ce2 are omitted from most of the diagram, but are still required for proper function of the design to write data on the falling edge of the SPI clock output and read data on the rising edge of the SPI clock output. Refer to the source code.



The Pmod CLS custom driver operates by waiting for the ATmega48 microcontroller to boot. It then indicates when a command is ready by outputting '1' on the Command Ready port.

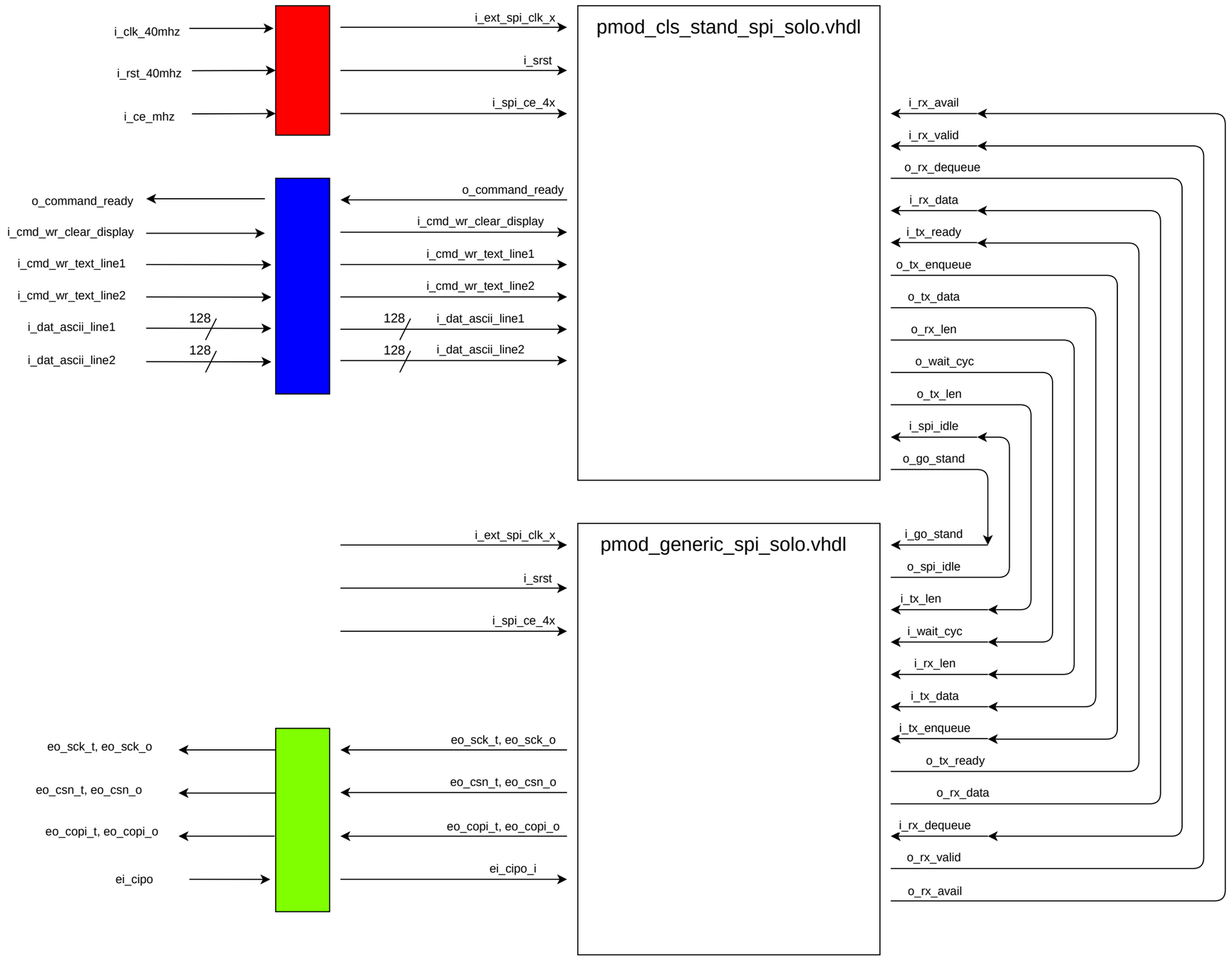
A pulse on Command Write Clear Display instructs the driver to transmit via SPI an ASCII text escape sequence to the ATmega48. The ASCII text escape sequence is a hard-coded text value that instructs that the LCD display be cleared of text on its two lines.

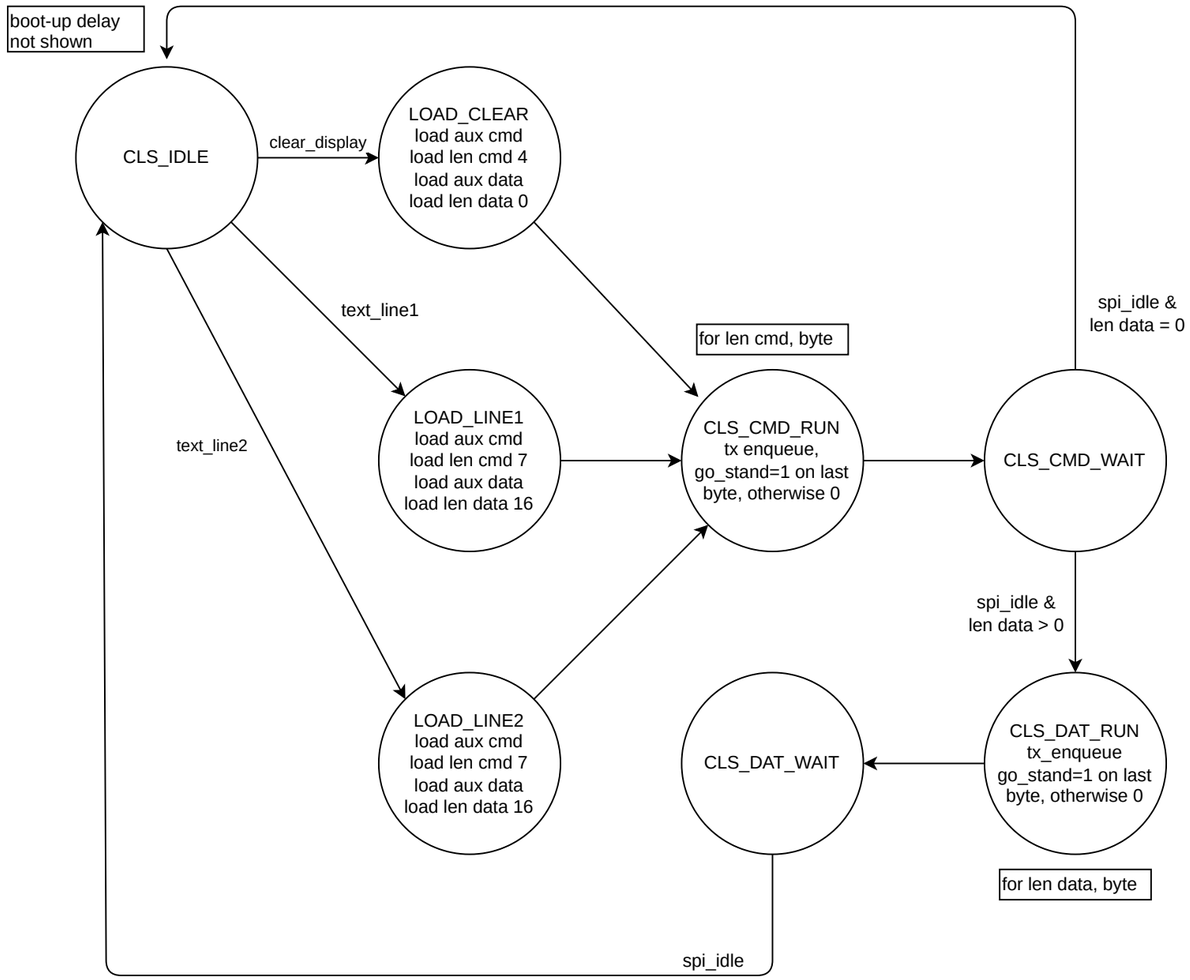
A pulse on Command Write Text Line 1 instructs an FSM to load the Data ASCII Line 1 input and then issue a sequence of sending two ASCII text pieces to the microcontroller. The first text piece instructs that the Pmod CLS be controlled to position its text writing cursor at the start of the LCD's first text line. The second text piece is the 16 character data value that is to be written to line 1 of the LCD.

A pulse on Command Write Text Line 2 instructs an FSM to load the Data ASCII Line 2 input and then issue a sequence of sending two ASCII text pieces to the microcontroller. The first text piece instructs that the Pmod CLS be controlled to position its text writing cursor at the start of the LCD's second text line. The second text piece is the 16 character data value that is to be written to line 2 of the LCD.

Other than the Command Write Clear Display which clears the display; there is no soft reset function to reboot the microcontroller and/or reset the display..

pmod_cls_custom_driver.vhdl

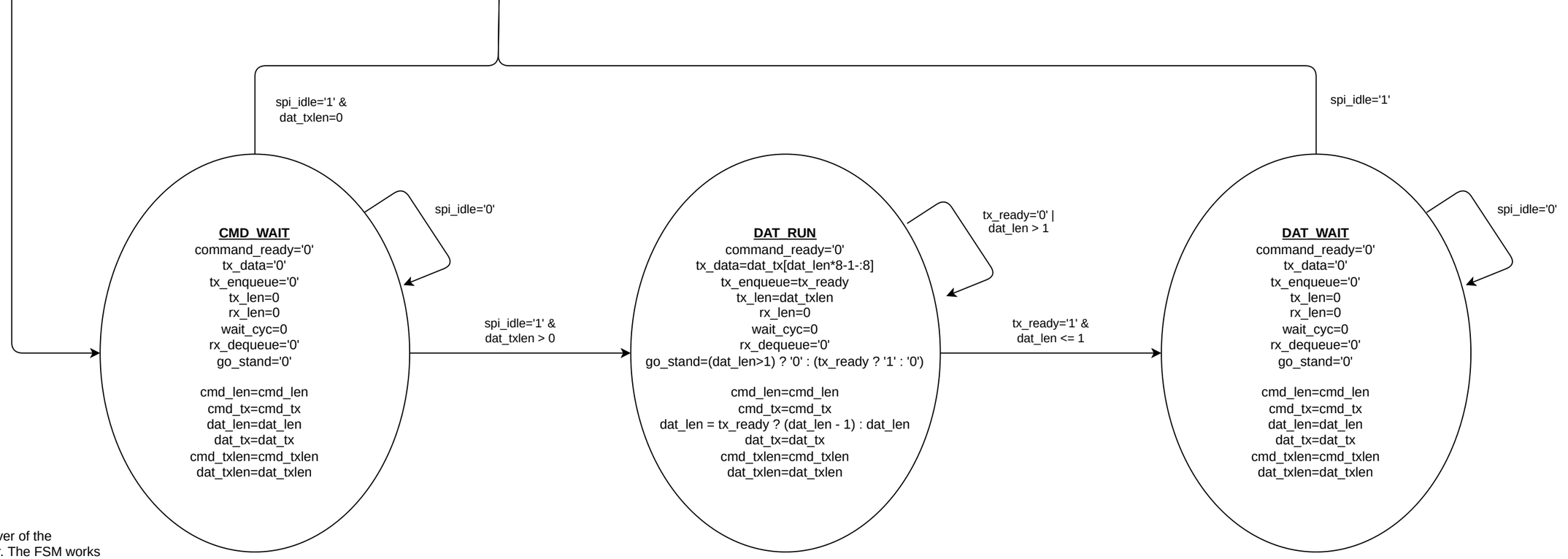




A FSM to operate the Digilent Inc. PMOD CLS LCD display communication via the single slave SPI-machine FSM of this document.

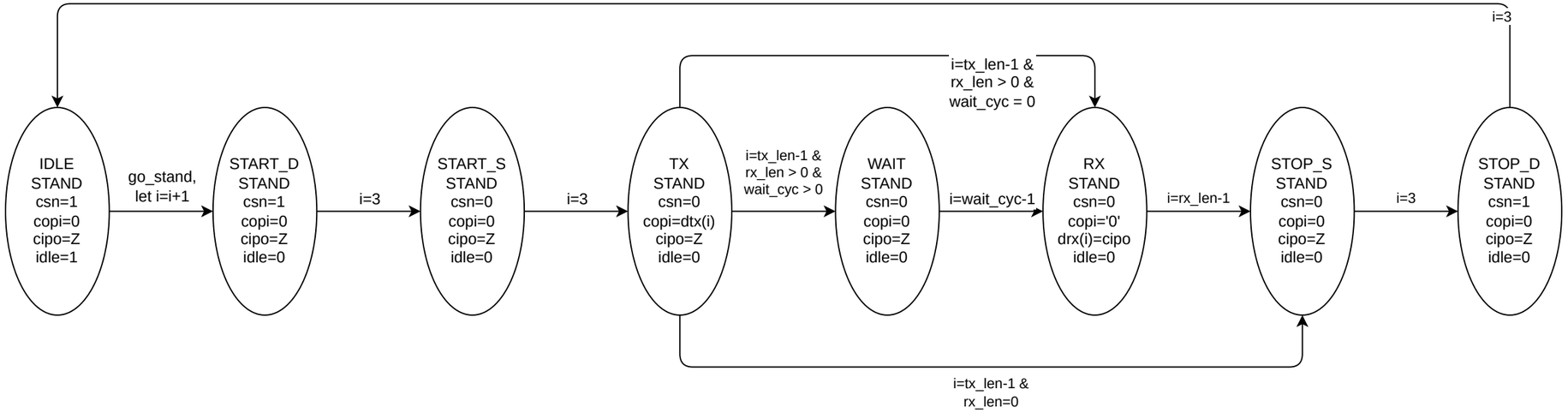
This diagram is
incomplete and does not
show boot-time delay. Also, some
state-bypass preventions
and iterations may not be shown.
This is the first design draft,
and the complete FSM diagram
is shown on another page.


rst,
cmd_len=0
dat_len=0,
cmd_tx=h00000000000000
dat_tx=h00000000000000000000000000000000,
cmd_txlen=0,
dat_txlen=0
BOOT0
command_ready='0'
tx_data='0'
tx_enqueue='0'
tx_len=0
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand='0'
cmd_len=cmd_len
cmd_tx=cmd_tx
dat_len=dat_len
dat_tx=dat_tx
cmd_txlen=cmd_txlen
dat_txlen=dat_txlen
t != pmodcls_boot
t = pmodcls_boot
IDLE
command_ready='1'
tx_data='0'
tx_enqueue='0'
tx_len=0
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand='0'
cmd_len=cmd_len
cmd_tx=cmd_tx
dat_len=dat_len
dat_tx=dat_tx
cmd_txlen=cmd_txlen
dat_txlen=dat_txlen
cmd_wr_clear_display='0' &
cmd_wr_text_line1='0'
cmd_wr_text_line2='0'
cmd_wr_clear_display=''1'
LOAD_CLEAR
command_ready='0'
tx_data='0'
tx_enqueue='0'
tx_len=0
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand='0'
cmd_len=4
cmd_tx=CLR_CMD
dat_len=0
dat_tx=0
cmd_txlen=4
dat_txlen=0
cmd_wr_clear_display='0' &
cmd_wr_text_line1='0'
cmd_wr_text_line2='1'
cmd_wr_clear_display='0' &
cmd_wr_text_line1='1'
tx_ready='0' |
cmd_len > 1
CMD_RUN
command_ready='0'
tx_data=cmd_tx[cmd_len*8-1-:8]
tx_enqueue=i_tx_ready
tx_len=cmd_txlen
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand=(cmd_len>1) ? '0' : (tx_ready ? '1' : '0')
cmd_len = tx_ready ? (cmd_len - 1) : cmd_len
cmd_tx=cmd_tx
dat_len=dat_len
dat_tx=dat_tx
cmd_txlen=cmd_txlen
dat_txlen=dat_txlen
LOAD_LINE2
command_ready='0'
tx_data='0'
tx_enqueue='0'
tx_len=0
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand='0'
cmd_len=7
cmd_tx=POS_LINE2
dat_len=16
dat_tx=dat_ascii_line2
cmd_txlen=7
dat_txlen=16
LOAD_LINE1
command_ready='0'
tx_data='0'
tx_enqueue='0'
tx_len=0
rx_len=0
wait_cyc=0
rx_dequeue='0'
go_stand='0'
cmd_len=7
cmd_tx=POS_LINE1
dat_len=16
dat_tx=dat_ascii_line1
cmd_txlen=7
dat_txlen=16
tx_ready='1' &
cmd_len <= 1



CLS Stand SPI Driver of the CLS Custom Driver. The FSM works by waiting for a command to (a) write a clear display command to the CLS, (b) write a 16-character line to the first line of the CLS, or (c) write a 16-character line to the second line of the CLS. The clear display command only writes an ANSI escape sequence with no textual data after it. The write line 1 and write line 2 commands write an ANSI escape sequence to position the cursor at the beginning of one of the two lines, and then 16 characters of text. The CLS microcontroller processes each command and line data. The clear display clears the 16x2 LCD; and the write line writes new text to the specified line of the 16x2 LCD.

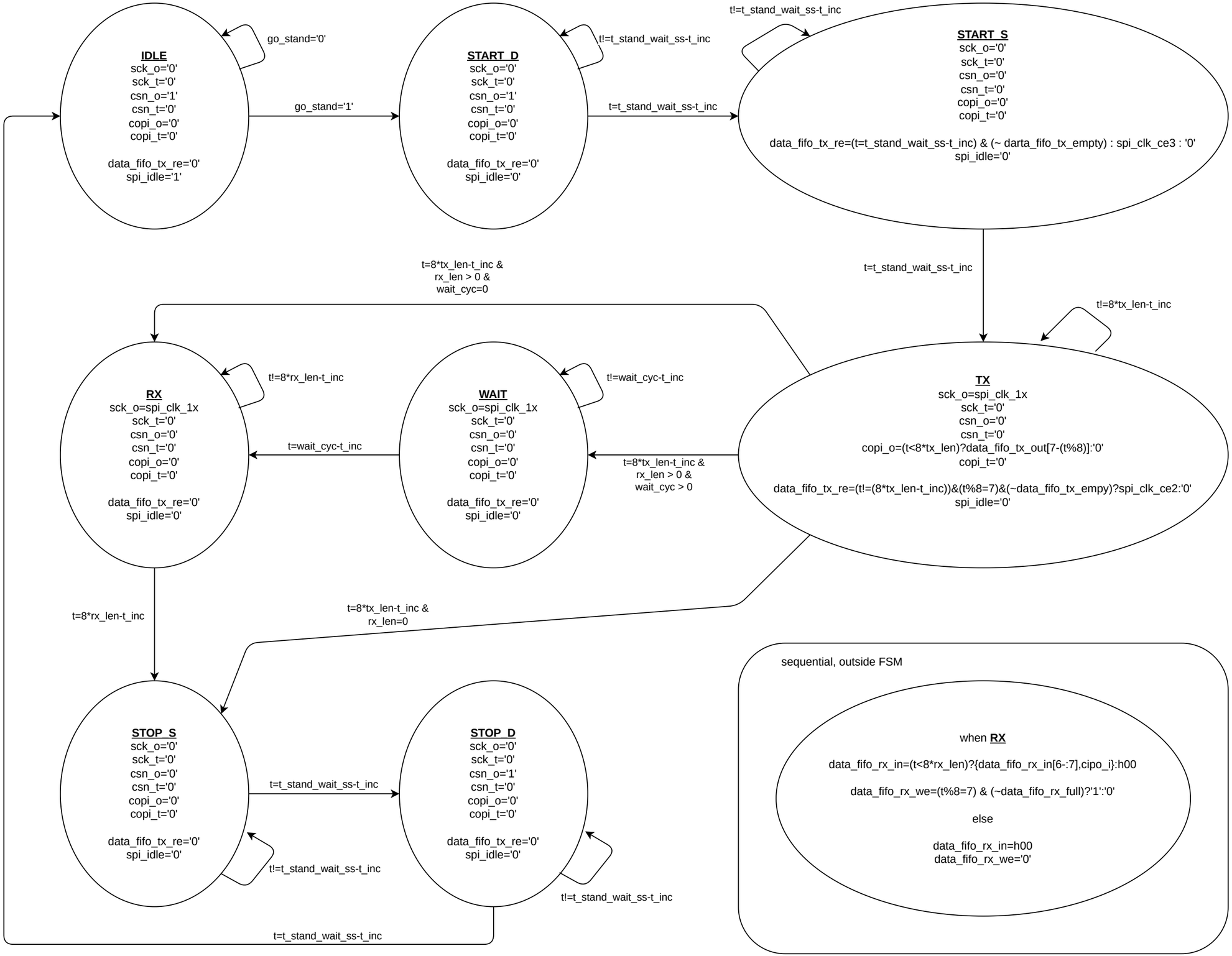
The first group of Moore machine outputs in each state bubble are FSM outputs, some of them with combinatorial output that is normally only seen in a Mealy machine. (This FSM could be said to be a hybrid Moore and Mealy machine.) The second group of outputs are recursive auxiliary outputs that retain state from one state to the next when assigned to itself.



In each transition, tx_len and rx_len are to be multiplied by 8 from the FSM input signals, as it only makes sense to input into the FSM a byte count, while the FSM requires transitioning based upon a bit count.

Generic SPI FSM, with only one SPI slave on the bus.

This diagram is incomplete and is shown as the first draft of designing the Standard SPI Single Slave Device driver.



Generic SPI FSM, with only one SPI slave on the bus.

In each transition, tx_len and rx_len are to be multiplied by 8 from the FSM input signals, as it only makes sense to input into the FSM a byte count, while the FSM requires transitioning based upon a bit count.

The main FSM combinatorial operates all states, but processes data on in the TX state when TX data is written out the copi port.

A side sequential process processes data only on the RX state when RX data is read from the cipo input port.

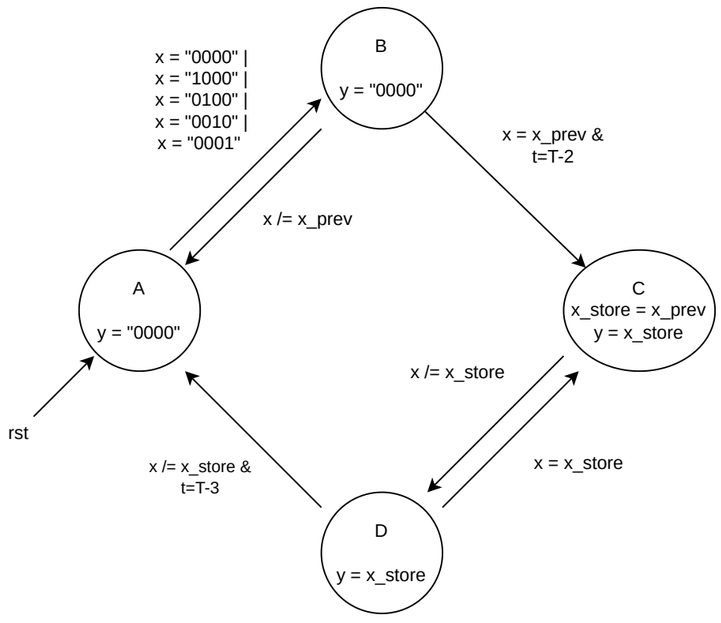
A general timer is used to time the SPI clock cycles (and thus bits). The timer is reset to zero at the transition from a state to a different state.

This machine could be considered a hybrid Moore and Mealy machine. The outputs during a state can change based upon timer and FIFO control inputs; but the diagram is drawn as a Moore machine.

The TX state controls reading byte by byte from a TX fifo and write its bits to COPI until the timer has reached 8*tx_len cycles.

The RX standalone sequential controls reading a byte bit by bit from CIPO and then writing the full byte to a RX fifo when the full byte has been received.
The main FSM still controls when the FSM is in the **RX** state or has transitioned to **STOP_S** state.

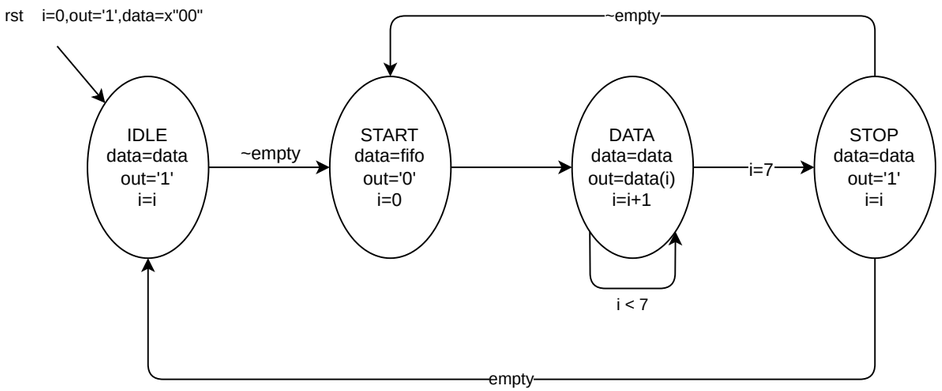
Note that clock enables such as spi_clk_ce2 are omitted from most of the diagram, but are still required for proper function of the design to write data on the falling edge of the SPI clock output and read data on the rising edge of the SPI clock output. Refer to the source code.



Full 4-button combined debouncer.

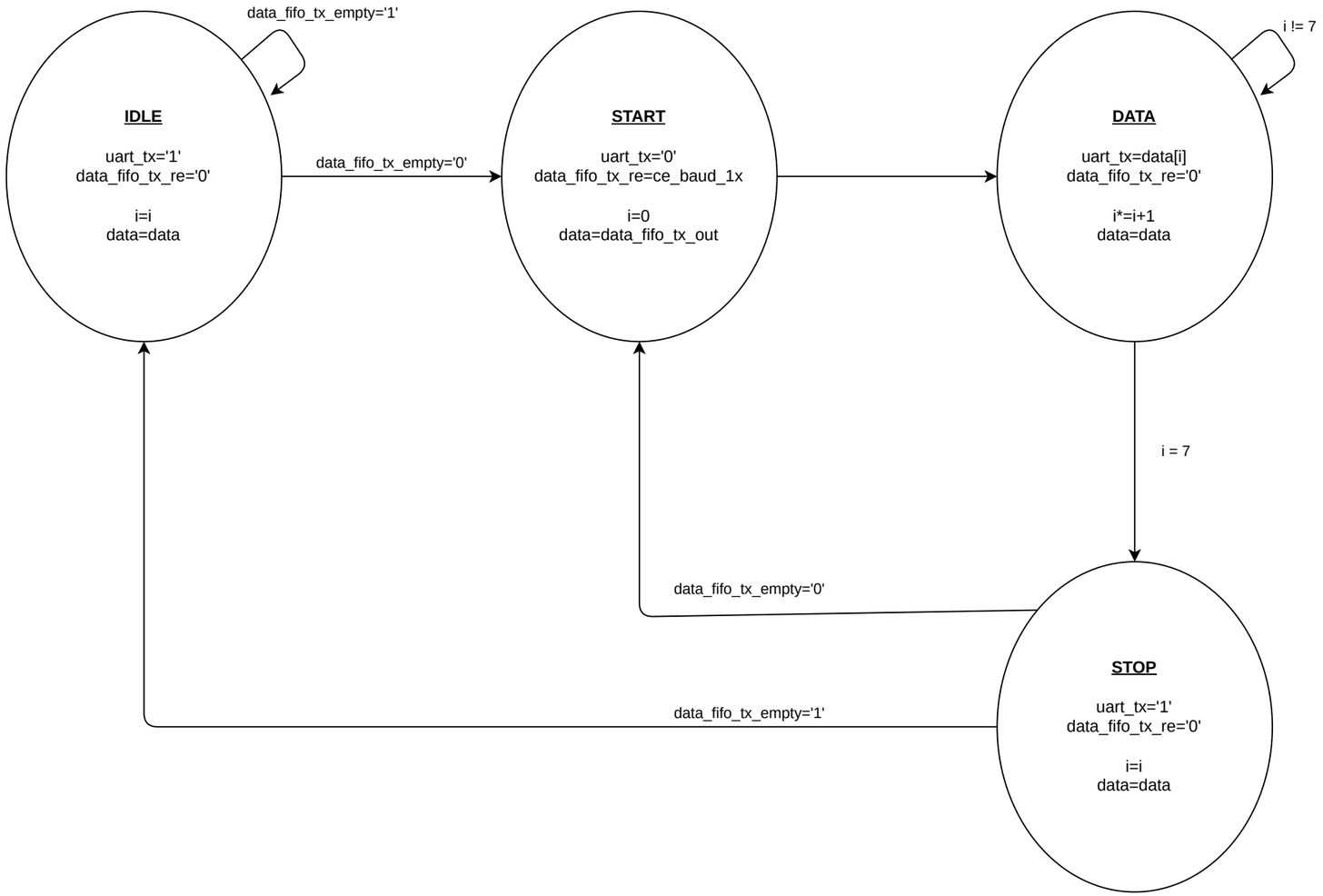
x is defined as a four-bit value.
x_prev is defined as a four-bit value that holds the previous clock cycle value of x.
x_store is defined as a four-bit value that holds the value of x and updates the
debouncer FSM entered state C during the transition BC..

The registers x_prev and x_store could be combined into one register, with its capture of X
being a clock-enable during transitions and states of a more complex diagram.



A TX ONLY UART output to UART chip from the FPGA, with the FSM executing at BAUD rate as its clock enable.

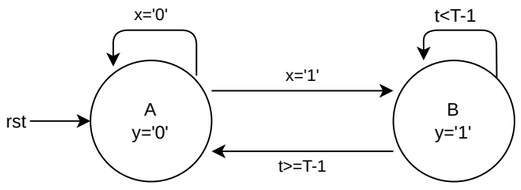
This is the first design draft,
and the complete FSM diagram
is shown on another page.



A TX ONLY UART output to UART chip from the FPGA, with the FSM executing at BAUD rate as its clock enable.

The FSM dequeues the UART TX FIFO when there is at least one more byte in the FIFO. No extra STOP/IDLE bits are generated except when the UART TX FIFO reaches empty.

The TX ONLY UART FSM generates a single START bit, 8 bits of DATA, a single STOP bit, and no parity bits.



Moore FSM for a synchronous pulse stretcher of signal X that lasts for a duration less than T, with Y lasting exactly T cycles.

Textbook Figure 8.28a. quoted from Chapter 8, page 174, of:

Finite State Machines in Hardware: Theory and Design (with VHDL and SystemVerilog)
by Volnei A. Pedroni,
reprinted courtesy of The MIT Press