

# STATE MACHINES

copyright © 2008 Sergio Masci

All rights reserved

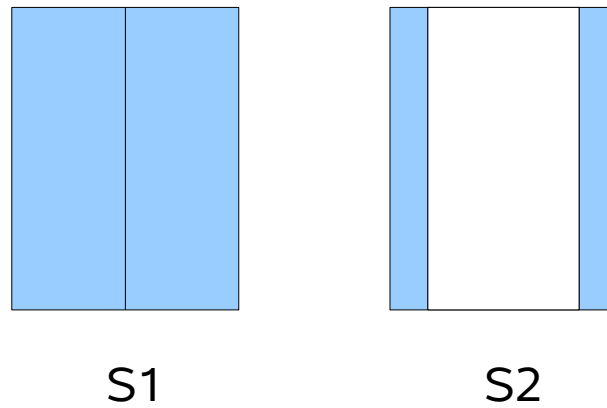
(draft revision date 2008-05-16)

## Table of Contents

<a href="#">Introduction to State Diagrams.....</a>	<a href="#">3</a>
<a href="#">Developing a state machine.....</a>	<a href="#">6</a>
<a href="#">Implementing state machines in software.....</a>	<a href="#">9</a>
<a href="#">Using groups of conditional statements.....</a>	<a href="#">9</a>
<a href="#">Table driven state machine.....</a>	<a href="#">12</a>
<a href="#">Benefits of using a small core dispatcher.....</a>	<a href="#">14</a>
<a href="#">Multiple interacting state machines.....</a>	<a href="#">15</a>
<a href="#">Table driven inefficiencies.....</a>	<a href="#">17</a>
<a href="#">Complete example of a table driven state machine.....</a>	<a href="#">18</a>
<a href="#">Critical event order.....</a>	<a href="#">21</a>
<a href="#">Step by step debugging using a state diagram.....</a>	<a href="#">30</a>
<a href="#">Animation of a real state machine trace.....</a>	<a href="#">35</a>
<a href="#">Reducing multiple states to a single state.....</a>	<a href="#">36</a>
<a href="#">The event queue.....</a>	<a href="#">38</a>
<a href="#">Generating events during an interrupt.....</a>	<a href="#">44</a>
<a href="#">Decreasing interrupt overheads.....</a>	<a href="#">48</a>
<a href="#">Multiple interacting state machines.....</a>	<a href="#">49</a>
<a href="#">Overall system design.....</a>	<a href="#">50</a>
<a href="#">CPU view.....</a>	<a href="#">50</a>
<a href="#">CPU component breakdown.....</a>	<a href="#">51</a>
<a href="#">Intra CPU mode interaction .....</a>	<a href="#">52</a>
<a href="#">part 5.....</a>	<a href="#">53</a>
<a href="#">Mode A / B Interaction.....</a>	<a href="#">53</a>
<a href="#">I/O Component Description.....</a>	<a href="#">55</a>
<a href="#">VCLK Component Description.....</a>	<a href="#">56</a>
<a href="#">I2C Component Description.....</a>	<a href="#">57</a>
<a href="#">Animated Message Flow Description.....</a>	<a href="#">58</a>
<a href="#">Mode B description.....</a>	<a href="#">59</a>
<a href="#">Appendix 1.....</a>	<a href="#">75</a>
<a href="#">Appendix 2.....</a>	<a href="#">76</a>
<a href="#">Appendix 3.....</a>	<a href="#">77</a>
<a href="#">Appendix 4.....</a>	<a href="#">78</a>
<a href="#">TBD.....</a>	<a href="#">91</a>

---

## Introduction to State Diagrams



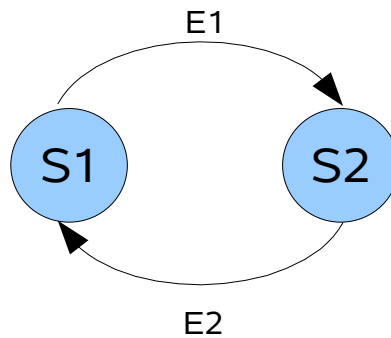
*Fig 1:*

Fig 1 shows two sets of elevator doors labelled S1 and S2. S1 shows the doors in a closed state while set S2 shows the same doors in an open state. Although this diagram shows two sets of doors it is actually only showing one set of doors in two different states (closed or open). The doors can only be in one of the two states at any one time.



*Fig 2:*

Fig 2 is a simplified representation of Fig 1. It shows the same two states S1 and S2 but does not bother trying to show any detail about those 2 states. It just shows that there are 2 states.



*Fig 3:*

Fig 3 shows the relationship between states S1 and S2 in terms of events E1 and E2. Here E1 is the open event and E2 is the close event. Note that E1 points from S1 to S2. This means that event E1 causes a transition from state S1 to S2 if the system is in state S1 when event E1 arrives. Event E1 has no effect when the system is in state S2. Likewise E2 points from S2 to S1 meaning that event E2 causes a transition from state S2 to S1 if the system is in state S2 when event E2 arrives. Event E2 has no effect when the system is in state S1. So the state diagram Fig 3 shows that when the elevator doors are closed, an open event causes them to go to the open state and when the doors are open a close event causes them to go to the close state. An open event is ignored if the doors are already open and a close event is ignored if the doors are already closed.

For completeness we would need to describe all the states and events shown on the state diagram.

S1 = doors closed

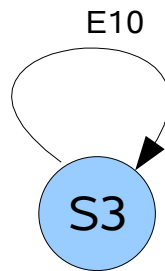
S2 = doors open

E1 = open

E2 = close

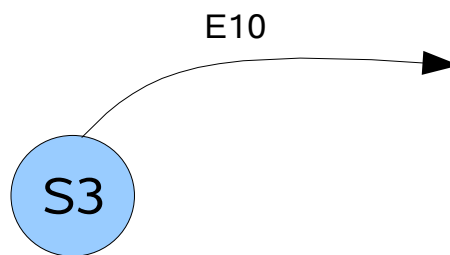
This is the essence of a state diagram. It simply shows all the possible states that a system can be in and all the events that can cause the system to change from one state to another.

**NOTE** although a state diagram shows all the possible states a system can be in, the system can only be in one state at any one time.



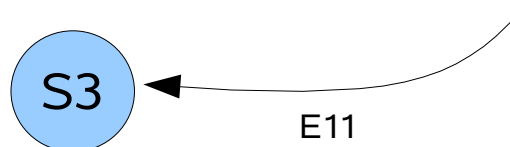
*Fig 4:*

Fig 4 shows a state S3 with an event E10 looping back to itself. This is valid. This kind of transition (a state acknowledging an event and staying in the same state) might be used to count the number of events of a specific type and possibly generate another event when a certain number is reached.



*Fig 5:*

Fig 5 shows a state S3 with an event E10 leading to nowhere. This is invalid. An event must always be drawn with a source state and destination state.



*Fig 6:*

Fig 6 shows a state S3 with an event E11 coming from nowhere. This is invalid. An event must always be drawn with a source state and destination state.

---

## Developing a state machine

Here we will develop a system for opening an elevator door using a switch.

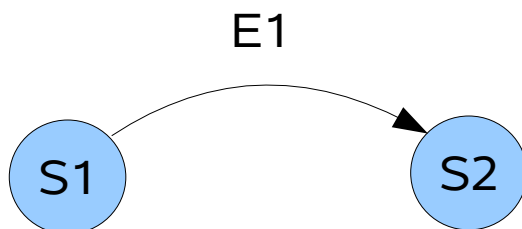
We start with a state diagram showing the possible states of the elevator door (Fig 7).



*Fig 7:*

S1 = door closed  
S2 = door opened

Now we need to add something that will cause the state to change. Let us say that pressing a switch is an event. This event will cause the state to change (Fig 8).

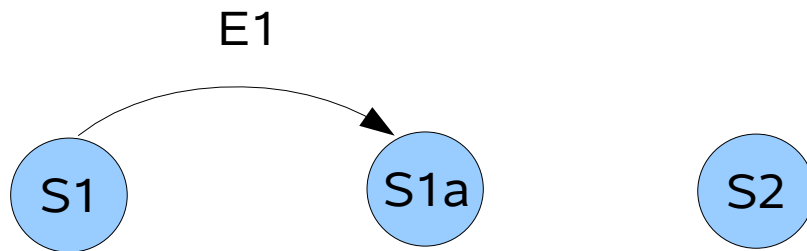


*Fig 8:*

So now we have

S1 = door closed  
S2 = door opened  
  
E1 = switch pressed

But elevator doors don't work like this in reality. Pressing a switch does not cause the doors to instantaneously be open – they take time to open. So we need to revise our state diagram (Fig 9).



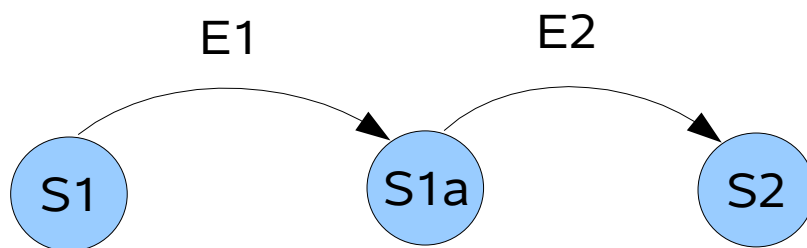
*Fig 9:*

So now we have

S1 = door closed  
S1a = door opening  
S2 = door opened

E1 = switch pressed

But we now have no way of getting from S1a to S2 – we need another event. This event will be when the doors are fully opened (Fig 10).



*Fig 10:*

So now we have

S1 = door closed  
S1a = door opening  
S2 = door opened

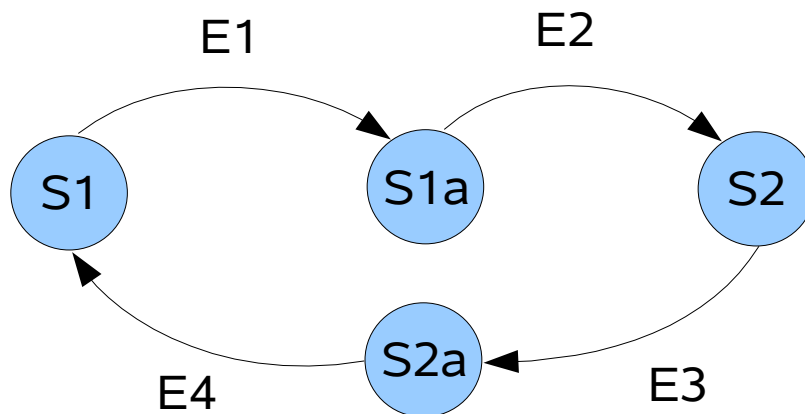
E1 = switch pressed

E2 = doors are fully opened

---

Great, so now we have the doors opening in response to a switch press. Looking at the state diagram we can see that there is no path from the doors being open to the doors being closed. This state diagram tells us that there is a fault with our system – we need a way to close the doors. We need to be able to press the switch again and for the doors to close in response to this.

We already have experience of opening the doors with a switch press and we have seen that we need a separate door opening state. The same is true for the door closing – we need a separate door closing state (Fig 11).



*Fig 11:*

So now we have

S1 = door closed  
S1a = door opening  
S2 = door opened  
S2a = door closing

E1 = switch pressed  
E2 = doors are fully opened  
E3 = switch pressed  
E4 = doors are fully closed

Now you may be wondering how you can press the switch twice without releasing it. The answer is you can't. So how does this affect the state machine. Actually even though the switch may generate a switch released event it doesn't matter because the state machine ignores that event (any event not shown for a state is ignored by default). So yes, a real user would need to release the switch before he could press it again but that is already accounted for by the state machine as it is drawn here.



---

## Implementing state machines in software

### *Using groups of conditional statements*

There are several ways to code a state machine. The simplest is to use a variable to hold the state ID and conditional statements to execute the code corresponding to that state.

e.g.

```
enum { STATE_DOOR_CLOSED,
      STATE_DOOR_OPENING,
      STATE_DOOR_OPEN,
      STATE_DOOR_CLOSING };

state = STATE_DOOR_CLOSED;

while (1)
{
    if (state == STATE_DOOR_CLOSED)
    {
        if ((PORTA & 1) != 0)
        {
            // switch pressed
            state = STATE_DOOR_OPENING;

            // turn on motor to open doors
            PORTB = 1;
        }
    }
    else
    if (state == STATE_DOOR_OPENING)
    {
        if ((PORTA & 2) != 0)
        {
            // door sensor indicates doors fully open
            state = STATE_DOOR_OPEN;

            // turn off motor to open doors
            PORTB = 0;
        }
    }
    else
    if (state == STATE_DOOR_OPEN)
    {
        if ((PORTA & 1) != 0)
        {
            // switch pressed
            state = STATE_DOOR_CLOSING;

            // turn on motor to close doors
            PORTB = 2;
        }
    }
    else
    if (state == STATE_DOOR_CLOSING)
    {
        if ((PORTA & 4) != 0)
        {
            // door sensor indicates doors fully closed
            state = STATE_DOOR_CLOSED;
        }
    }
}
```

---

```
        // turn off motor to close doors
        PORTB = 0;
    }
}
```

**NOTE** here there is no event mailbox, instead each state is responsible for moving to another state by setting the “state” variable.

Another way of writing the same thing is to use a “switch” statement instead of multiple “if” statements:

```
while (1)
{
    switch (state)
    {
        case STATE_DOOR_CLOSED:

            if ((PORTA & 1) != 0)
            {
                // switch pressed
                state = STATE_DOOR_OPENING;

                // turn on motor to open doors
                PORTB = 1;
            }

            break;

        case STATE_DOOR_OPENING:

            if ((PORTA & 2) != 0)
            {
                // door sensor indicates doors fully open
                state = STATE_DOOR_OPEN;

                // turn off motor to open doors
                PORTB = 0;
            }

            break;

        case STATE_DOOR_OPEN:

            if ((PORTA & 1) != 0)
            {
                // switch pressed
                state = STATE_DOOR_CLOSING;

                // turn on motor to close doors
                PORTB = 2;
            }

            break;

        case STATE_DOOR_CLOSING:

            if ((PORTA & 4) != 0)
            {
                // door sensor indicates doors fully closed
                state = STATE_DOOR_CLOSED;
            }
    }
}
```

---

```
        // turn off motor to close doors
        PORTB = 0;
    }
}
```

This type of implementation gets very hard to read as the state machine grows in complexity.

---

## ***Table driven state machine***

Another way to implement a state machine is to have a small core that dispatches functions depending on what state the "state machine" is actually in and what event has occurred. This type of implementation uses a table.

Each row of the table would look something like this:

```
current state
new state
event
state transition function
state monitor function
```

The core of the state machine would look something like this:

```
enum EVENT_ID
{
    EVENT_NONE,
};

enum STATE_ID
{
    STATE_RESET,
};

struct STATE_DESC
{
    int    cur_state,
          new_state,
          event;

    void (*transition_func)(STATE_ID, STATE_ID, EVENT_ID);
    void (*monitor_func)(STATE_ID);
};

void reset_STF(STATE_ID, STATE_ID, EVENT_ID);
void reset_SMF(STATE_ID);

struct STATE_DESC
state_tbl[] =
{
    { STATE_RESET,
      STATE_DOOR_OPENING,
      EVENT_SWITCH_PRESSED,
      reset_STF,
      reset_SMF },
};

EVENT_ID
event,
tmp_event;

STATE_ID
current_state;

int    state_tbl_indx;
```

---

```

current_state = STATE_RESET;
event = EVENT_NONE;
state_tbl_indx = 0;

while (1)
{
    if (event != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event == event)
            {
                tmp_event = event;

                event = EVENT_NONE;

                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;

                state_tbl_indx = j;

                break;
            }
        }

        (*state_tbl[state_tbl_indx].monitor_func)(current_state);
    }
}

```

In the above table driven state machine, the state machine is in the state identified by 'current\_state' the event that is to be processed while the state machine is in the state 'current\_state' arrives in the mailbox called 'event'. When an event arrives, the state machine dispatcher looks to find a match (state / event pair) in the event table. If it is not found then the event is ignored while in the 'current\_state'. If an entry is found in the event table (event for current\_state) then the event transition function for the state / event pair is triggered and the new state and state monitor function is remembered.

The state monitor function is repeatedly called by the dispatcher when the dispatcher is not processing events. This allows the state monitor function to monitor aspects of the system that are important to that state and generate events if something occurs.

State transition and monitor functions can be shared and may occur in many places in the state table. To make a state monitor function more general purpose it may generate many different events, one or more of which are ignored in the current state.

---

## ***Benefits of using a small core dispatcher***

A very important and useful consequence of using such a state machine dispatcher is that other tasks can be performed when the state machine is idle i.e. not executing state transition functions or state monitor functions. So the above simple dispatcher could cope with the watchdog.

e.g.

```
while (1)
{
    if (event != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event == event)
            {
                tmp_event = event;

                event = EVENT_NONE;

                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;

                state_tbl_indx = j;

                break;
            }
        }

        (*state_tbl[state_tbl_indx].monitor_func)(current_state);

        // note addition of watchdog processing
        reset_watchdog();
    }
}
```

---

## Multiple interacting state machines

Sometimes a state machine will have a large number of states and it will be obvious that large parts of the state machine are repeated in different places with small subtle differences. In such cases it is often possible to break up such a large state machine into two or more much smaller state machines that interact with each other by sending each other events – kind of like breaking down a huge program into a set of functions and calling the functions in many different places.

Another important benefit of using a table driven state machine is that it makes concurrently running two or more interacting state machines trivial.

e.g.

```
while (1)
{
    if (event_1 != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_1_len; j++)
        {
            if (state_tbl_1[j].cur_state == current_state_1 &&
                state_tbl_1[j].event == event_1)
            {
                tmp_event = event_1;

                event_1 = EVENT_NONE;

                (*state_tbl_1[j].transition_func)(
                    current_state_1,
                    state_tbl_1[j].new_state,
                    tmp_event);

                current_state_1 = state_tbl_1[j].new_state;

                state_tbl_1_idx = j;

                break;
            }
        }
    }
}
```

```
// NOTE addition of second state machine dispatcher

(*state_tbl_1[state_tbl_idx].monitor_func)(current_state_1);

if (event_2 != EVENT_NONE)
{
    for (j=0; j<max_state_tbl_2_len; j++)
    {
        if (state_tbl_2[j].cur_state == current_state_2 &&
            state_tbl_2[j].event == event_2)
        {
            tmp_event = event_2;

            event_2 = EVENT_NONE;

            (*state_tbl_2[j].transition_func)(
                current_state_2,
                state_tbl_2[j].new_state,
                tmp_event);

            current_state_2 = state_tbl_2[j].new_state;
```

---

```
                                state_tbl_2_indx = j;
                                break;
                            }
                        }
                    }
                (*state_tbl_2[state_tbl_2_indx].monitor_func)(current_state_2);
            }
        }
```



---

### ***Table driven inefficiencies***

It may appear that the table driven state machine is much more inefficient than the simpler equivalent group of conditional statements. Actually there are some inefficiencies but there is also one huge gain – the state monitor function can be computed and executed very efficiently when no events occur. In contrast, executing the equivalent state monitor code in the simpler state machine (using the group of conditional statements) takes longer because of all the wasted tests needed to get to the state monitor code.

Also when searches in the state table start to impact on performance it is possible to greatly reduce the search time by implementing more complex state tables. Such optimisations are very hard to do on the simpler group of conditional statements type of state machine.

---

## ***Complete example of a table driven state machine***

```
enum EVENT_ID
{
    EVENT_NONE,
    EVENT_SWITCH_PRESSED,
    EVENT_DOOR_FULLY_OPENED,
    EVENT_DOOR_FULLY_CLOSED
};

enum STATE_ID
{
    STATE_DOOR_CLOSED,
    STATE_DOOR_OPENING,
    STATE_DOOR_OPEN,
    STATE_DOOR_CLOSING
};

struct STATE_DESC
{
    int    cur_state,
          new_state,
          event;

    void (*transition_func)(STATE_ID, STATE_ID, EVENT_ID);
    void (*monitor_func)(STATE_ID);
};

void door_closed_STF(STATE_ID, STATE_ID, EVENT_ID);
void door_closed_SMF(STATE_ID);

void door_opening_STF(STATE_ID, STATE_ID, EVENT_ID);
void door_opening_SMF(STATE_ID);

void door_open_STF(STATE_ID, STATE_ID, EVENT_ID);
void door_open_SMF(STATE_ID);

void door_closing_STF(STATE_ID, STATE_ID, EVENT_ID);
void door_closing_SMF(STATE_ID);

struct STATE_DESC
    state_tbl[] =
    {
        { STATE_DOOR_CLOSED,
          STATE_DOOR_OPENING,
          EVENT_SWITCH_PRESSED,
          door_closed_STF,
          door_closed_SMF },

        { STATE_DOOR_OPENING,
          STATE_DOOR_OPEN,
          EVENT_DOOR_FULLY_OPENED,
          door_opening_STF,
          door_opening_SMF },

        { STATE_DOOR_OPEN,
          STATE_DOOR_CLOSING,
          EVENT_DOOR_FULLY_CLOSED,
          door_open_STF,
          door_open_SMF },
    }
```

---

```

        { STATE_DOOR_CLOSING,
          STATE_DOOR_CLOSED,
          EVENT_SWITCH_PRESSED,
          door_closing_STF,
          door_closing_SMF },
    };

void state_machine_dispatcher(void)
{
    EVENT_ID
        event,
        tmp_event;

    STATE_ID
        current_state;

    int    state_tbl_indx;

    current_state = STATE_DOOR_CLOSED;
    event = EVENT_NONE;
    state_tbl_indx = 0;

    while (1)
    {
        if (event != EVENT_NONE)
        {
            for (j=0; j<max_state_tbl_len; j++)
            {
                if (state_tbl[j].cur_state == current_state &&
                    state_tbl[j].event == event)
                {
                    tmp_event = event;

                    event = EVENT_NONE;

                    (*state_tbl[j].transition_func)(
                        current_state,
                        state_tbl[j].new_state,
                        tmp_event);

                    current_state = state_tbl[j].new_state;

                    state_tbl_indx = j;

                    break;
                }
            }

            (*state_tbl[state_tbl_indx].monitor_func)(current_state);
        }
    }

    void door_closed_STF(STATE_ID cur_state, STATE_ID new_state, EVENT_ID new_event)
    {
    }

    void door_closed_SMF(STATE_ID state)

```

---

---

```
{
    if ((PORTA & 1) != 0)
    {
        // switch pressed
        event = EVENT_SWITCH_PRESSED;

        // turn on motor to open doors
        PORTB = 1;
    }
}

void door_opening_STF(STATE_ID cur_state, STATE_ID new_state, EVENT_ID new_event)
{
}

void door_opening_SMF(STATE_ID state)
{
    if ((PORTA & 2) != 0)
    {
        // door sensor indicates doors fully open
        event = EVENT_DOOR_FULLY_OPENED;

        // turn off motor to open doors
        PORTB = 0;
    }
}

void door_open_STF(STATE_ID cur_state, STATE_ID new_state, EVENT_ID new_event)
{
}

void door_open_SMF(STATE_ID state)
{
    if ((PORTA & 1) != 0)
    {
        // switch pressed
        event = EVENT_SWITCH_PRESSED;

        // turn on motor to close doors
        PORTB = 2;
    }
}

void door_closing_STF(STATE_ID cur_state, STATE_ID new_state, EVENT_ID new_event)
{
}

void door_closing_SMF(STATE_ID state)
{
    if ((PORTA & 4) != 0)
    {
        // door sensor indicates doors fully closed
        event = EVENT_DOOR_FULLY_CLOSED;

        // turn off motor to close doors
        PORTB = 0;
    }
}
```

---

## Critical event order

Sometimes people complain that a state machine misses an event when it shouldn't or that the order in which events arrive is critical. They think that this is a fundamental problem with state machines. The reality is that the state machine concept is fine, it is the model of the system that they are trying to describe as a state machine that is wrong.

Consider the following:

The system has two switches (labelled SWA and SWB) and two LEDs (labelled LDA and LDB). when switch SWA is pressed LED LDA is to illuminate, when switches SWA and SWB are pressed both LEDs LDA and LDB are to illuminate.

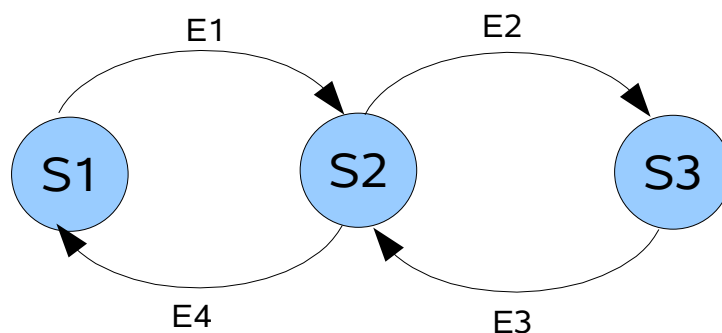
So we can describe the system as 3 states (Fig 12):



*Fig 12:*

- S1 = LDA and LDB are both extinguished
- S2 = LDA is illuminated and LDB is extinguished
- S3 = LDA and LDB are both illuminated

Now we need to add events to cause transitions between the states (Fig 13)



*Fig 13:*

- S1 = LDA and LDB are both extinguished

---

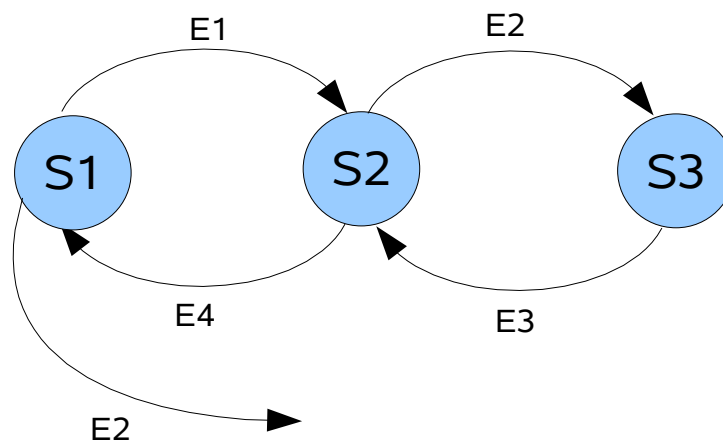
S2 = LDA is illuminated and LDB is extinguished  
S3 = LDA and LDB are both illuminated

E1 = SWA pressed  
E2 = SWB pressed  
E3 = SWB released  
E4 = SWA released

From this we can see that pressing switch SWA then SWB will cause both LEDs LDA and LDB to be illuminated (state S3).

So if both switches SWA and SWB are pressed simultaneously we just need to ensure that event E1 is seen before event E2 – wrong, we should not impose this requirement. How would we guarantee this anyway? Should we specify that there is always a delay after reading a switch just in case the other switch is also being pressed? If so how long should the delay be 0.5 seconds, 1.0 seconds? This just complicates the switch reading software and prevents the state machine responding to events while the CPU is tied up busy waiting for the switches to be read.

What we actually need to do is take advantage of the state machine and use its properties to solve the problem. We know that in state S1 either event E1 or event E2 can occur, so let us add that to the diagram (Fig 14):



*Fig 14:*

But all event transitions must be between states. In other words this new event must lead to a state.

Now we have a problem. How do we add a new state as we have already defined all possible states for this state machine. The answer is to refine the existing states and in the process make room for new states.

We will start by fully annotating the state diagram to make things easier to see (Fig 15)

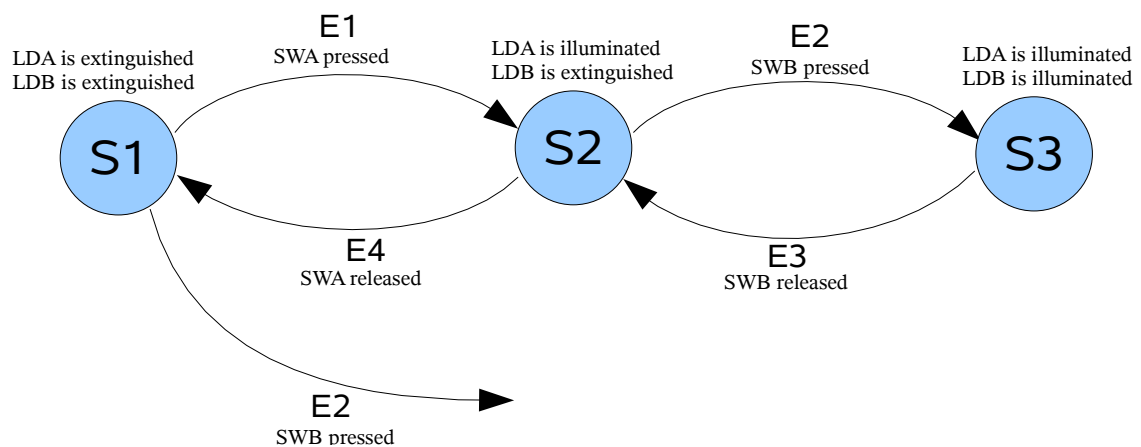


Fig 15:

We go from “only interested in condition of LEDs”:

S1 = LDA is extinguished  
LDB is extinguished

S2 = LDA is illuminated  
LDB is extinguished

S3 = LDA is illuminated  
LDB is illuminated

To “interested in condition of LEDs and switches”:

S1 = LDA is extinguished  
LDB is extinguished  
SWA is released  
SWB is released

S2 = LDA is illuminated  
LDB is extinguished  
SWA is pressed  
SWB is released

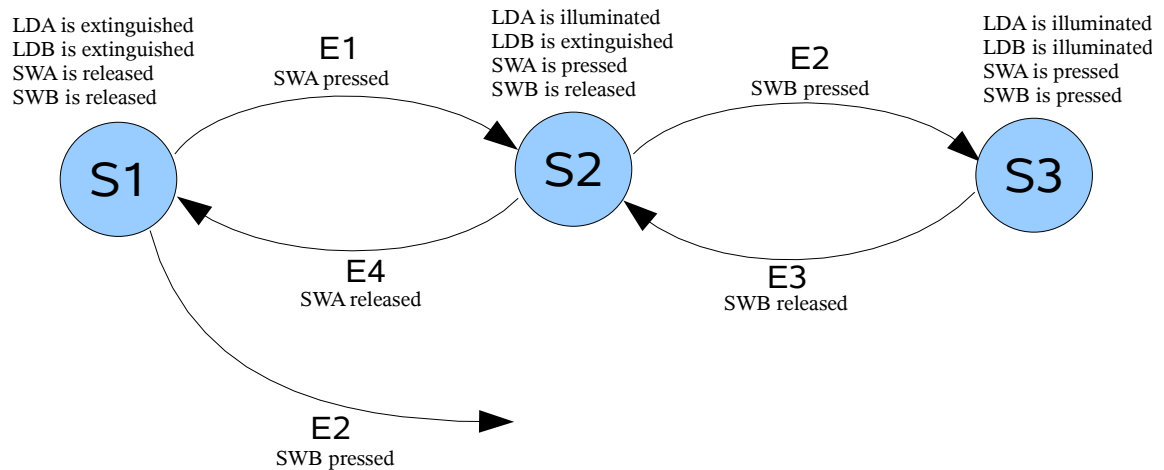
S3 = LDA is illuminated  
LDB is illuminated  
SWA is pressed  
SWB is pressed

**NOTE:** In state S1, SWA pressed is NOT the same as the event E1 SWA pressed. In the event E1 “SWA pressed” indicates a transition from released to pressed whereas in S1 “SWA pressed” indicated the switch is actually in the pressed conditional (or being held in the ON condition if you

---

prefer).

We can now redraw the fully annotated state diagram with the revised properties of each state.



*Fig 16:*

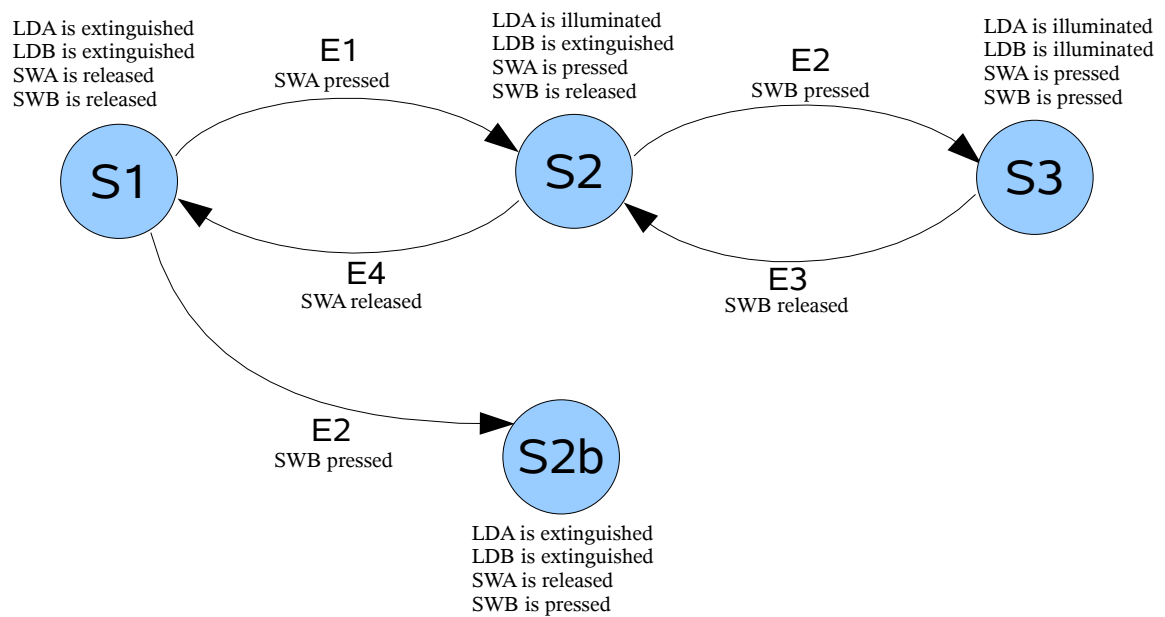
This gives us the possibility of a new state:

S2b = LDA is extinguished  
LDB is extinguished  
SWA is released  
SWB is pressed

**NOTE:** S1 and S2b can now coexist even though LDA and LDB are in an identical condition for both states because we have introduced a new property by which we can tell them apart.

We can now add this new state to our diagram (Fig 18) and as if by magic it fits exactly where we predicted a new state needs to be placed.





*Fig 17:*

S1 = LDA is extinguished  
LDB is extinguished  
SWA is released  
SWB is released

S2 = LDA is illuminated  
LDB is extinguished  
SWA is pressed  
SWB is released

S2b = LDA is extinguished  
LDB is extinguished  
SWA is released  
SWB is pressed

S3 = LDA is illuminated  
LDB is illuminated  
SWA is pressed  
SWB is pressed

E1 = SWA pressed

E2 = SWB pressed

E3 = SWB released

E4 = SWA released

Now it is immediately obvious that there is an event missing between states S2b and S3, so we add

that (Fig 18):

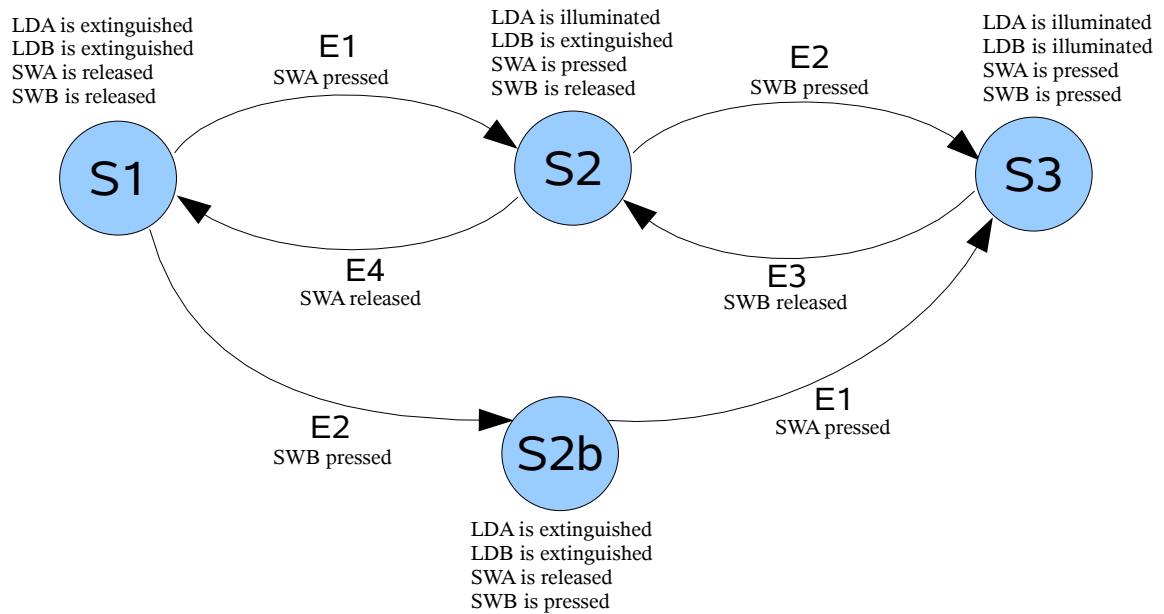


Fig 18:

Now the state diagram is starting to look a little crowded so we will remove the annotations to make it simpler to see the overall structure (too much detail can obscure the model) (Fig 19)

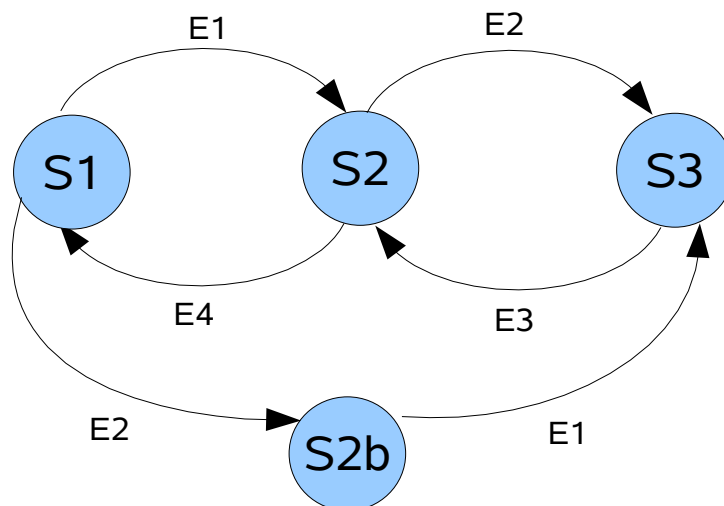


Fig 19:

Notice that we haven't added any new types of events but we have added a new state. Now the systems looks as though it will actually work. It seems to cope with the two switches being pressed

---

simultaneously regardless of the order in which the events occur.

Now comes the time to make use of the hidden power of the state machine diagram. Using the diagram, try simulating what happens when you press and release switches in different orders. For the two switches you will have the following combinations

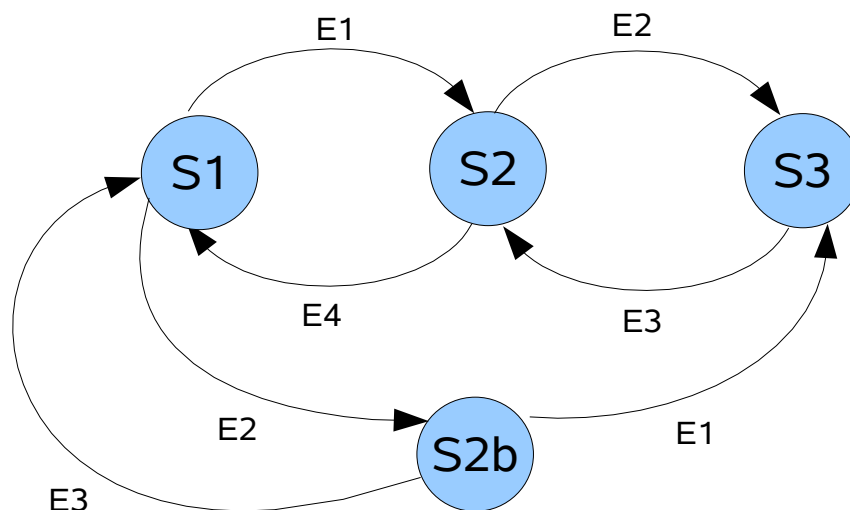
SWA = released, SWB = released  
SWA = pressed, SWB = released  
SWA = released, SWB = pressed  
SWA = pressed, SWB = pressed

If you apply all 4 of the above combinations to each of the states S1, S2, S2b and S3 you will be able to verify the correct operation of the system for all combinations of all defined inputs.

Doing this you will find that there is a problem if you start at S1 and you press and release SWB before SWA. Pressing SWB while in S1 takes you to S2b and subsequently releasing SWB leaves you still in S2b. Pressing SWA then takes you from S2b to S3. This is not what we intended.

We end up in S3 while SWA is released and our definition of S3 clearly states that both SWA and SWB are pressed while in this state.

The state machine diagram has highlighted the fact that we have not considered pressing and releasing SWB before SWA. Looking at the state diagram it is clear that if we release SWB while in S2b we need a state transition back to S1 (this is the state with both SWA and SWB released). In other words an E3 event occurring at S2b should cause a transition back to S1, thus (Fig 20):



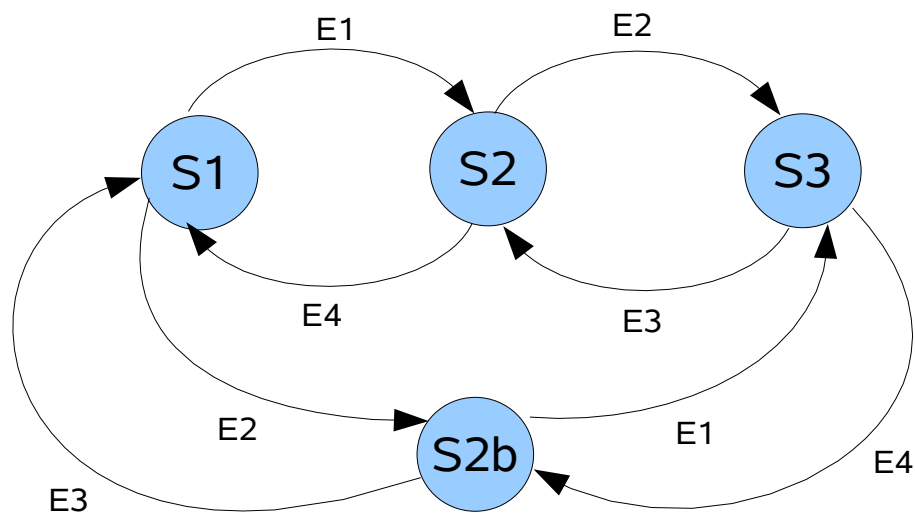
*Fig 20:*

If we go through the same debug process again we will find that this time, pressing and releasing

---

SWB will take us back to S1 and subsequently pressing SWA will make the system behave the way we intended it to.

Continuing with the debugging we now find that while in state S3, if we release SWA, the system again starts to behave in an unexpected manor. In this situation both LEDs are illuminated whereas we expect them to both be extinguished. We need the state machine to react to event E4 (switch SWA being released) while in state S3. So we need the state which has as its properties SWA and SWB both pressed to transition to the state which has as its properties SWA released and SWB pressed when an event E4 occurs. Looking at the diagram we can see that state S2b has the properties SWA released and SWB pressed. So we need a state transition from S3 to S2b when event E4 occurs, thus (Fig 21):



*Fig 21:*

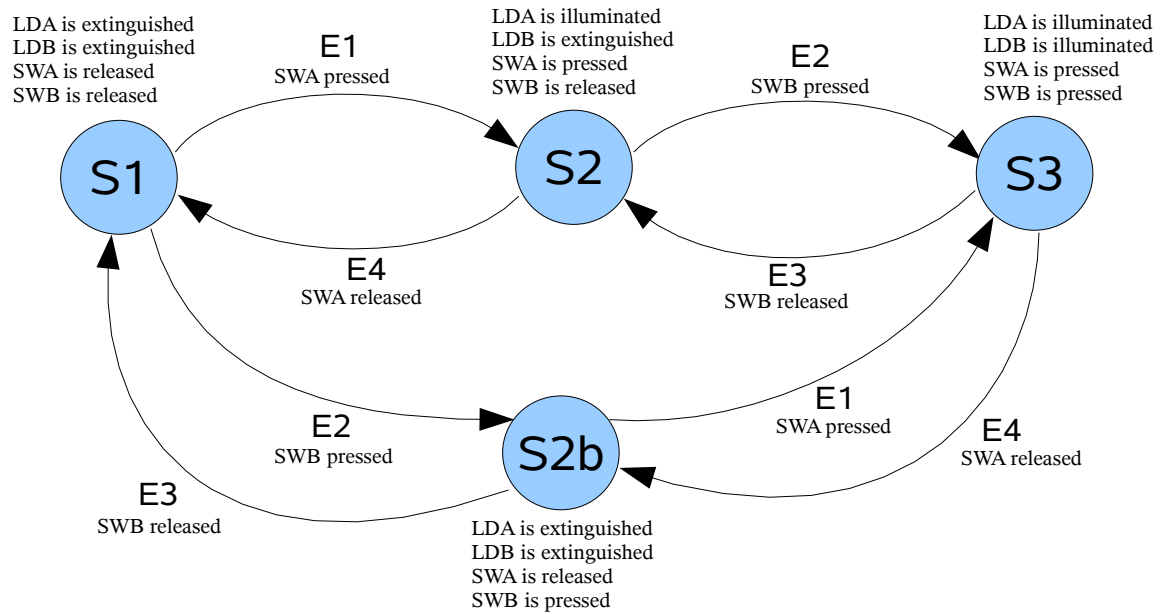
Again we can go through the debug process and this time we find that the system behaves as expected regardless of whether E1 or E2 arrives first during a simultaneous pressing of SWA and SWB or whether E3 or E4 arrives first during a simultaneous releasing of SWA and SWB.

The important point here is that the diagram has actually lead us to the correct solution based on:

1. the state we were in when the error occurred
2. the state we actually wanted to be in
3. the event that was needed to take us from the error state to the required correct state.

---

The fully annotated state diagram is shown in Fig 22



*Fig 22:*

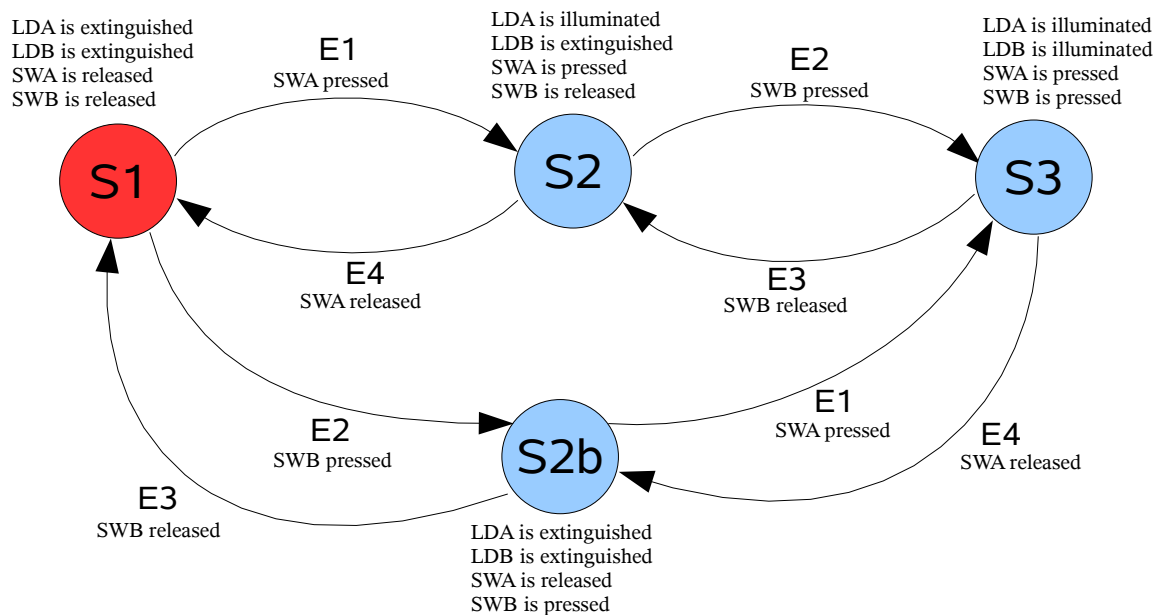
---

## Step by step debugging using a state diagram

Testing and debugging a state machine gets more complicated as the number of states increases. It gets harder to keep track of which states have been checked and in particular which events for a given state have been checked. The easiest way to do this is to build a trace path using the state diagram from which the state machine was produced.

How this works: we pick a colour to highlight the current state and last event - I prefer red.

To start, take a printed copy of your state machine (Fig 23) and highlight the initial (or start state) in red:



*Fig 23:*

No matter what happens now (whether the phone rings and you get dragged into a long tech support conference call or you simply end up trudging through a long assembly listing of your code) you will instantly know where you are in the state machine.

Next we apply a stimulus to the system that will generate an event. For this system we have only defined four events (E1, E2, E3 and E4) and they can only be generated by:

E1 = SWA pressed  
(this is a transition of the switch from released to pressed)

E2 = SWB pressed  
(this is a transition of the switch from released to pressed)

---

E3 = SWB released  
(this is a transition of the switch from pressed to released)

E4 = SWA released  
(this is a transition of the switch from pressed to released)

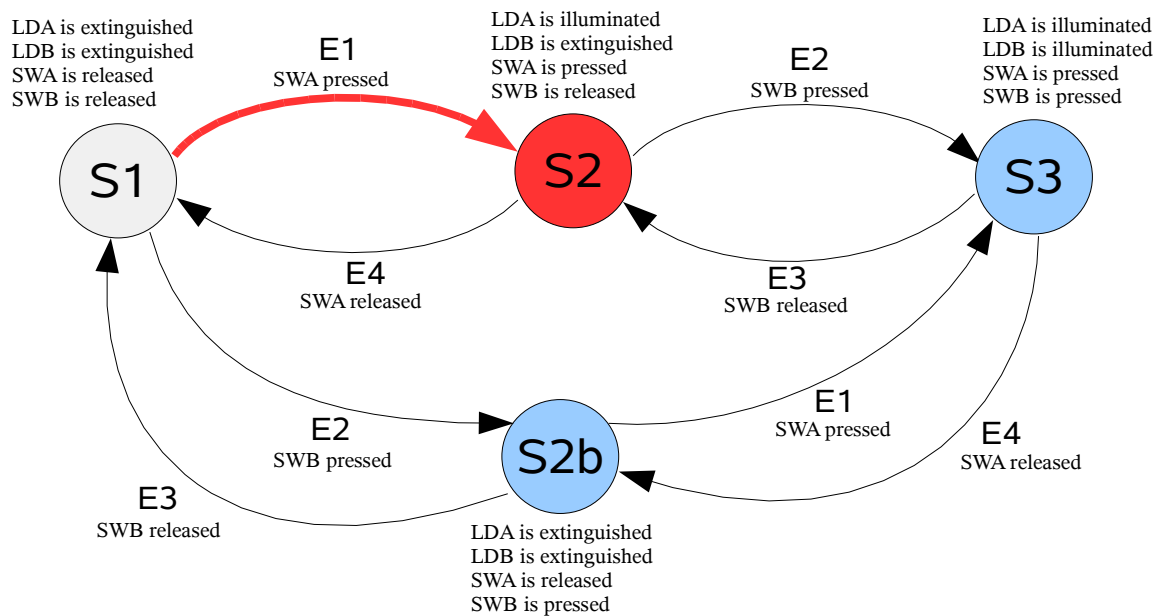
**NOTE:** we do not repeatedly get event E1 while the switch is held pressed, we only get one event during the transition from released to pressed.

So the stimulus we chose is “press SWA”. We note the new condition of SWA (pressed) and we trace the effect of the generated event E1. Tracing is archived by:

- (1) highlighting the event that has been generated
- (2) fading the state that was the current state
- (3) highlighting the new current state

**NOTE:** this process also works when the event transitions back to the same current state.

Fig 24 shows an example of a traced state transition from S1 to S2 caused by E1



*Fig 24:*

Now we need to look at the code of the state transition function that must be invoked when there is a state transition between S1 and S2 caused by event E1. This is part of the testing / debugging process. We need to see what effect this code will have on the system / state machine.

**NOTE:** it is possible for the state transition function to generate an event or change some other

---

property of the state (e.g. initialise a global variable).

Next we repeat the process. This time we will select “press SWB” as the stimulus. Again we make a note of the new condition of SWB (pressed) and we trace the new event E2 on the state diagram (Fig 25)

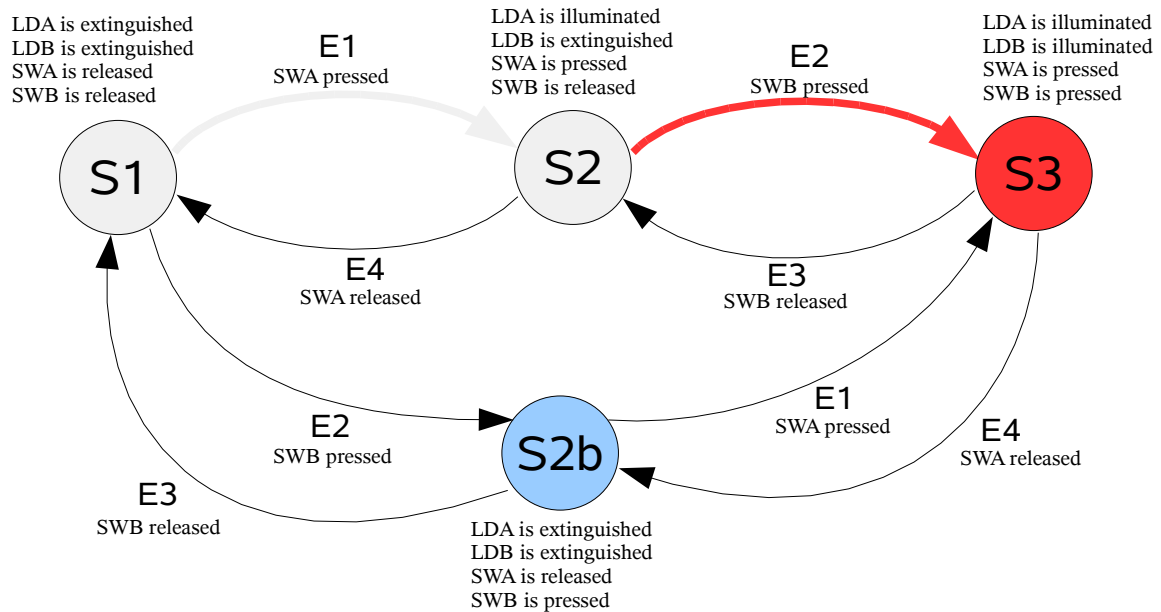


Fig 25:

**NOTE:** the previous highlighted state and event (the red ones in Fig 24) are now faded and the new current state and event leading to that state are highlighted (the red ones in Fig 25).

As before we need to look at the code of the state transition function that must be invoked when there is a state transition between S2 and S3 caused by event E2.

We must repeat this process until all events and all states have been covered and shown as faded on the state diagram.

While we are tracing through the state diagram we must also ensure that all the properties of the current state are in the condition defined for that state. In the case of our example system this means that the LEDs LDA and LDB are illuminated or extinguished as per the requirements of the state and that the switches SWA and SWB are in the pressed or released condition as per the requirements of the state. If any of the properties of the state is undefined or does not match the requirements of the state then we have discovered a bug in the system.

Now the only other thing to do is verify the code of the state monitor functions for each state. The smaller these functions the easier it is to verify them. State monitor functions should not maintain any state information that is hidden from the state diagram. Doing so makes it much harder to verify



---

and maintain the state machine.

e.g. if state S1 has a state monitor function that uses a global variable X and state S3 has another state monitor function that uses the same global variable X then we might enter S1 or S3 with invalid or undefined values of X. This makes it difficult to verify S1 and S3 since we have a property which S1 and S3 are dependent on which we cannot verify. To correct this fault we must specify X as a property of S1 and S3 and we must also specify the value or range of values which this variable must have when in this state.

When you have gone through every state and every event, your state diagram will look like Fig 26

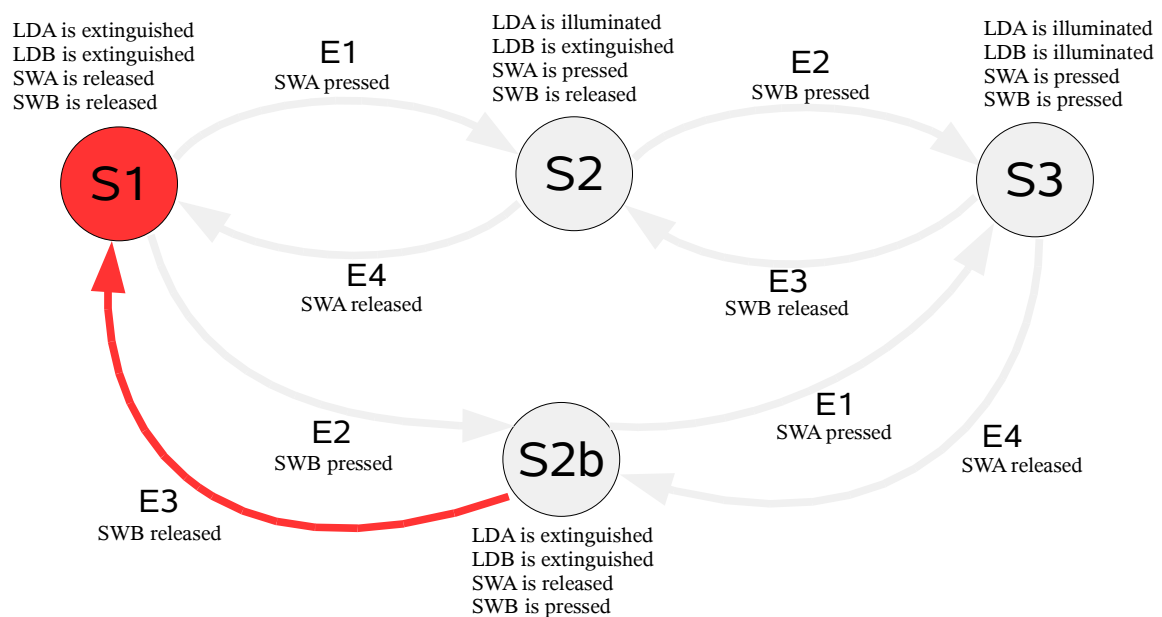


Fig 26:

Here you see that all the states and events have been covered at least once (they are either faded or highlighted). If any of the states is not covered then this indicates that the state is not reachable via any of the events that lead to it. If any of the events are not covered then this indicates that either:

1. the events have not been generated for the state from which they originate or
2. the events cannot be generated for the state from which they originate

To show how the state diagram would catch a faulty design let us remove the event transition E4 between S3 and S2b and start the trace from the beginning (Fig 27)

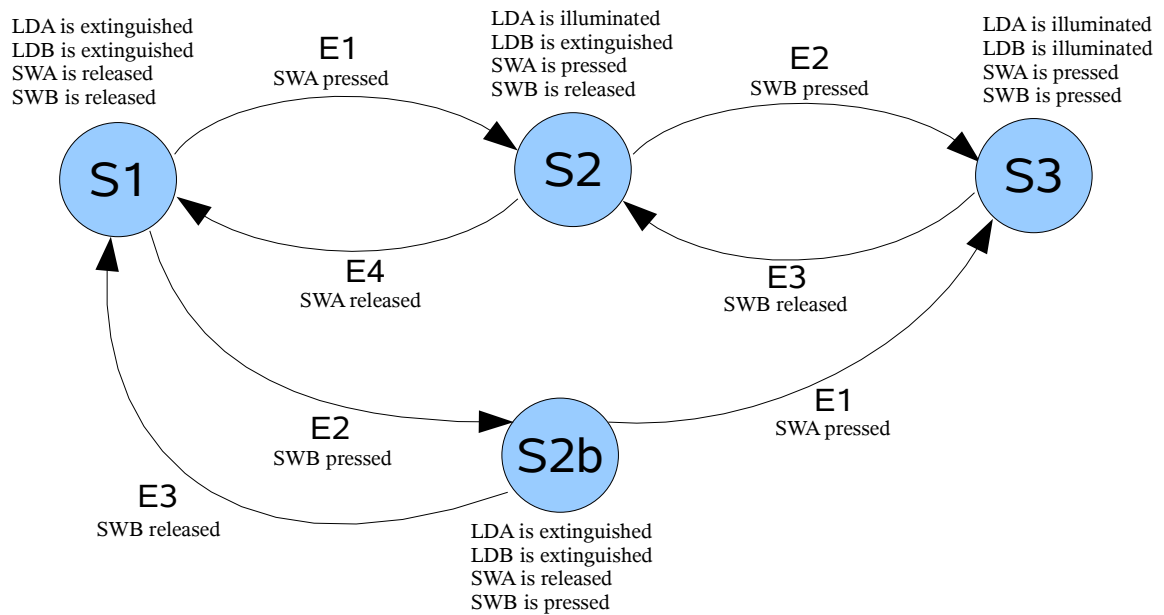


Fig 27:

Now we can apply stimuli selectively to each of the states in turn as we trace through the state machine and we will find that there is a path that will allow us to cover all of the states and events shown.

try this

SWA pressed :	S1(E1) -> S2
SWB pressed :	S2(E2) -> S3
SWB released :	S3(E3) -> S2
SWA released :	S2(E4) -> S1
SWB pressed :	S1(E2) -> S2b
SWA pressed :	S2b(E1) -> S3
SWB released :	S3(E3) -> S2
SWA released :	S2(E4) -> S1
SWB pressed :	S1(E2) -> S2b
SWB released :	S2b(E3) -> S1

However if we apply all the possible stimuli to the system while we are in each of the states, we will find that when we get to S3 releasing SWA generates an event E4 for which there is no state transition so the state machine does not respond to E4 (this is OK as we are allowed to ignore events while in a state) however after the event E4 has occurred, the state machine remains in S3 and one of the properties of S3 is now invalid i.e. SWA is in the released condition and it should be in the pressed condition.

---

## ***Animation of a real state machine trace***

An example of an animation showing a trace of a real state machine can be found at:

<http://www.xcprod.com/titan/DEMO/zmech001.gif>

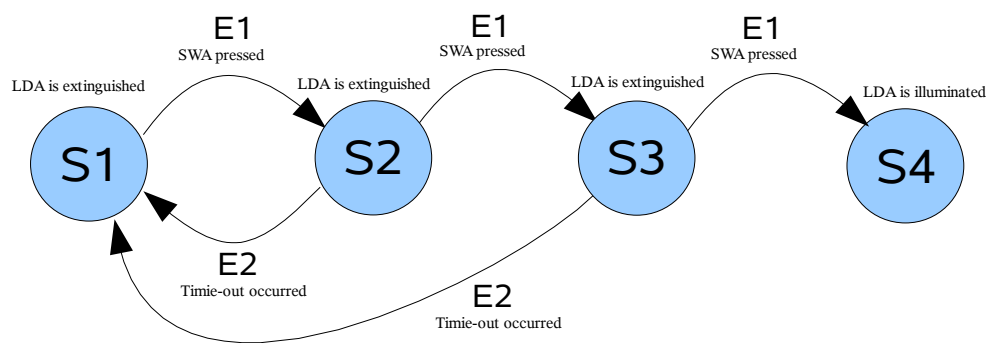
This animation shows an actual trace of a self contained PIC 16F628 machine code executable performed using the ZMech state machine development tool. On the left of the image are 8 virtual switches which the user can toggle using the mouse. In this way the user can provide a stimulus which the virtual PIC can interpret as an event. All event transitions that the state machine performs on the virtual PIC are shown on the state diagram. Once the user is satisfied that the executable is working correctly it can be downloaded to a PIC without modification and run in real time.

---

## Reducing multiple states to a single state

Here we will show how to combine a simple sequence of similar states into one state.

Let us start by defining the requirement. We need to see a switch pressed three times in quick succession and on the third press we need to illuminate an LED. We will call the switch SWA and the LED LDA

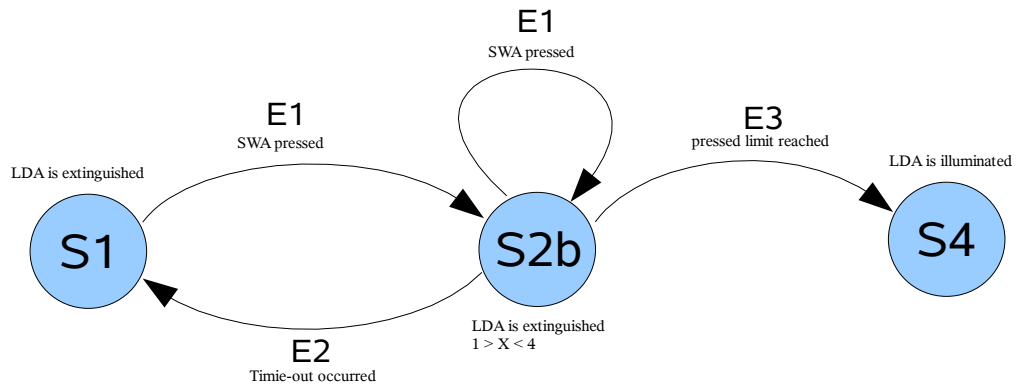


*Fig 28:*

S1 = LDA extinguished  
S2 = LDA extinguished  
S3 = LDA extinguished  
S4 = LDA illuminated

E1 = SWA pressed  
E2 = time-out occurred

**NOTE:** here the special event E2 (time-out occurred) would probably be generated by an interrupt routine periodically counting down. The state transition function invoked during a transition between states caused by event E1 would probably be used to reset this time-out.



*Fig 29:*

S1 = LDA extinguished  
 S2b = LDA extinguished  
        $1 > X < 4$   
 S4 = LDA illuminated

E1 = SWA pressed  
 E2 = time-out occurred  
 E3 = pressed limit reached ( $X = 3$ )

The following things have changed between Fig 28 and Fig 29

- (1) state S2 and S3 have gone
- (2) state S2b has been added
- (3) state S2b has the property (i.e.  $1 > X < 4$ )
- (4) state S2b is affected by event E1 such it causes a transition back to the same state S2b
- (5) event E3 has been added

We have also changed the state transition function invoked when S1 transitions to S2b due to event E1. This function must now initialise X to 1 so that it can be used by S2b.

The state transition function invoked when S2b transitions (back) to S2b due to event E1 must increment X and when  $X = 3$  it must generate an event E3.

Why is the state machine shown in Fig 29 better than the one shown in Fig 28? Answer: it's not. This state machine is an artificial example. Having an extra state and not needing the variable X is actually preferable. However if we increased the number of switch presses needed (to say 5 or above) then the simplified state machine would justify using a variable.

A situation which would definitely benefit from using variables is when a state machine is used to process packets of data to be sent or received over a communications channel.

---

## The event queue

The simplest event queue that can be implemented is a mailbox. This is simply a variable which holds the ID of the last event triggered or a special number which indicates “no new events have occurred since the last event was processed”. The event mailbox is very simple to implement and efficient to use. However it does have a major drawback: only one event can be generated at a time. Using an event mailbox means that the event handler must act on an event before the next event is generated otherwise events are lost.

In the example we gave where two switches were each capable of generating an independent event we did not consider the possibility that either switch could generate an event before the event of the other switch had been processed. If we used a simple input monitoring function to generate an event whenever it saw a change in the state of a switch we could be in trouble if we simply used an event mailbox. Consider the following switch reading function:

```
void read_switch(void)
{
    // NOTE: this example assumes that all
    // connected switches produce perfectly
    // debounced digital signals

    static int    old_port_a = 0;

    int    new_port_a;
    int    j;
    int    changes;

    new_port_a = PORTA;

    changes = old_port_a ^ new_port_a;

    for (j=0; j<8; j++)
    {
        if ((changes & (1 << j)) != 0)
        {
            // a change from pressed to released or
            // released to pressed was detected

            if ((new_port_a & (1 << j)) != 0)
            {
                // generate an event indicating switch j was pressed
                event = j | 0x80;
            }
            else
            {
                // generate an event indicating switch j was released
                event = j | 0x00;
            }
        }
    }

    old_port_a = new_port_a;
}
```

Now if we call this 'read\_switch' function at the start of our state machine dispatcher we will get an event whenever there is a change in a switch condition (i.e. it changes from pressed to released or

---

from released to pressed). This works great unless two switches change condition at exactly the same instant. In this situation we find that the first event is overwritten by the second event before the function has returned and given the state machine dispatcher the opportunity to process the first event. In other words the first event gets lost.

It would be possible to change the 'read\_switch' function such that it only ever returns one event at a time but this would complicate the function and would only defer the solution of the problem. Say we also needed to add another source of events – maybe temperature too low or too hot. We would then need to implement another function say 'read\_temperature', which we would need to call before or after 'read\_switch'. Now we start have problems since each function must be away of the others ability to generate events and be capable of holding off until the others events have been processed. This is a horrible solution since it is error prone, difficult to debug and maintain. A much simpler solution is to use a real queue maintained by other functions which can be used by the 'read\_switch', 'read\_temperature' and any other functions we may need to add to generate events and the state machine dispatcher which will process the events.

The following is a set of simple queue handling functions which will fulfil our requirements.

```
#define  max_event_queue_len 8

struct EVENT_QUEUE
{
    int    head,
          tail;

    int    buff[max_event_queue_len];
};

void event_queue_init(void)
{
    event_queue.head = 0;
    event_queue.tail = 0;
}

int event_queue_read(void)
{
    int    res, index;

    if (event_queue.tail == event_queue.head)
    {
        // return and indicate no events in queue
        return EVENT_NONE;
    }

    res = event_queue.buff[event_queue.tail];

    index = event_queue.tail + 1;

    if ( index == max_event_queue_len)
    {
        index = 0;
    }

    event_queue.tail = index;

    return res;
}
```

---

```
void event_queue_write(int val)
{
    int    index;

    index = event_queue.head + 1;

    if (index == max_event_queue_len)
    {
        index = 0;
    }

    if (index != event_queue.tail)
    {
        // the event queue is not full so append the event

        event_queue.buff[index] = val;

        event_queue.head = index;
    }
}
```

**NOTE:** 'max\_event\_queue\_len' is defined as 8. This means that our event queue can hold a maximum of 7 events. If we try to store more than 7 events at any one time, the newer events will be discarded.

So our 'read\_switch' function now becomes:

```
void read_switch(void)
{
    // NOTE: this example assumes that all
    // connected switches produce perfectly
    // debounced digital signals

    static int    old_port_a = 0;

    int    new_port_a;
    int    j;
    int    changes;
    int    tmp_event;

    new_port_a = PORTA;

    changes = old_port_a ^ new_port_a;

    for (j=0; j<8; j++)
    {
        if ((changes & (1 << j)) != 0)
        {
            // a change from pressed to released or
            // released to pressed was detected

            if ((new_port_a & (1 << j)) != 0)
            {
                // generate an event indicating switch j was pressed
                tmp_event = j | 0x80;
            }
            else
            {
                // generate an event indicating switch j was released
                tmp_event = j | 0x00;
            }
        }
    }
}
```



---

```

        // NOTE: we now use a FIFO as an event queue
        // instead of a mailbox
        event_queue_write(tmp_event);
    }
}

old_port_a = new_port_a;
}

```

and our state machine dispatcher now becomes:

```

while (1)
{
    read_switch();

    read_temperature();

    tmp_event = event_queue_read();

    if (tmp_event != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event == tmp_event)
            {
                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;

                state_tbl_indx = j;

                break;
            }
        }

        (*state_tbl[state_tbl_indx].monitor_func)(current_state);
    }
}

```

The small state machine core (the dispatcher loop) has allowed us to elegantly place the polling functions (switch, temperature and any other input reading functions) where they will be repeatedly invoked to produce a constant stream of data for the rest of the system to consume. This will reduce the overall reaction time to an external event without having to write complex spaghetti code to achieve the same effect as is sometimes the case.

However since most of the event transition and monitor functions are very short (they run to completion very quickly) we are now spending a disproportionate amount of time polling inputs that cannot physically change within the time it takes to execute the polling function several hundred times (switch change might take 20ms, polling function executed about 500 times in 20ms). This will often not be a problem but there will be the odd occasion when it is – primarily because the state monitor function of the current state needs a big slice of the CPU's time (maybe it's doing complex maths). A simple solution is to reduce the number of times the polling functions are called relative to the event transition and monitor functions.

e.g.

---

```

poll_cnt = 0;

while (1)
{
    poll_cnt++;

    if (poll_cnt >= 100)
    {
        poll_cnt = 0;

        read_switch();

        read_temperature();
    }

    tmp_event = event_queue_read();

    if (tmp_event != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event == tmp_event)
            {
                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;

                state_tbl_indx = j;

                break;
            }
        }

        (*state_tbl[state_tbl_indx].monitor_func)(current_state);
    }
}

```

An alternative to 'poll\_cnt' would be to use a software timer that is decremented to zero by the systems heartbeat interrupt handler. This has the advantage of compensating for both high and low demands on the CPU's time by both the event transition and state monitor functions.

e.g.

```

void main()
{
    sw_timer_1 = 0;

    while (1)
    {
        if ( sw_timer_1 == 0)
        {
            // reset the software timer

            // NOTE: sw_timer_1 is now in multiples
            // of the heartbeat time,
            // e.g. for heartbeat = 5ms
            //      sw_timer_1 = 4 * 5ms = 20ms

            sw_timer_1 = 4;
        }
    }
}

```

---

```

        read_switch();
        read_temperature();
    }

    tmp_event = event_queue_read();

    if (tmp_event != EVENT_NONE)
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event      == tmp_event)
            {
                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;
                state_tbl_indx = j;
                break;
            }
        }

        (*state_tbl[state_tbl_indx].monitor_func)(current_state);
    }
}

void interrupt_handler
{
    // this function is executed whenever an
    // interrupt occurs

    if (sw_timer_1 != 0)
    {
        // decrement software timer down to zero
        sw_timer_1--;
    }
}

```

---

## ***Generating events during an interrupt***

The event queue handling functions are interrupt safe provided the 'event\_queue\_write' function is not used in both the interrupt handler and the main line code. If reading from the event queue only ever occurs in the main line code (actually the state machine dispatcher) and writing to the event queue only occurs in the interrupt handler then there is no possibility of conflict and interrupts do not need to be disabled while the event queue is being written to.

If the main line code is also to generate events (concurrently with the interrupt handler) then interrupts should be disabled during writing to the event queue from within the main line code (actually the state monitor or transition functions).

However a much better solution would be to use two separate events queues, one that is only written to by the interrupt handler and one that is only written to by the main line code. In this way there is no need to disable interrupts during writing to the event queue and so interrupt response is not impacted by the implementation of the state machine.

e.g.

```
// NOTE: the use of functions dedicated to
// processing only the interrupt event queue

#define max_event_queue_interrupt_len 8

struct EVENT_QUEUE_INTERRUPT
{
    int    head,
          tail;

    int    buff[max_event_queue_interrupt_len];
};

struct EVENT_QUEUE_INTERRUPT
    event_queue_interrupt;

void event_queue_interrupt_init(void)
{
    event_queue_interrupt.head = 0;
    event_queue_interrupt.tail = 0;
}

int event_queue_interrupt_read(void)
{
    int    res, index;

    if (event_queue_interrupt.tail == event_queue_interrupt.head)
    {
        // return and indicate no events in queue
        return EVENT_NONE;
    }

    res = event_queue_interrupt.buff[event_queue_interrupt.tail];

    index = event_queue_interrupt.tail + 1;
```

---

```

        if ( index == max_event_queue_interrupt_len)
        {
            index = 0;
        }

        event_queue_interrupt.tail = index;

        return res;
    }
}

void event_queue_interrupt_write(int val)
{
    int    index;

    index = event_queue_interrupt.head + 1;

    if (index == max_event_queue_interrupt_len)
    {
        index = 0;
    }

    if (index != event_queue_interrupt.tail)
    {
        // the event queue is not full so append the event

        event_queue_interrupt.buff[index] = val;

        event_queue_interrupt.head = index;
    }
}

```

// NOTE: the use of functions dedicated to  
// processing only the main line event queue

```

#define max_event_queue_len 8

struct EVENT_QUEUE
{
    int    head,
          tail;

    int    buff[max_event_queue_len];
};

struct EVENT_QUEUE
event_queue;

void event_queue_init(void)
{
    event_queue.head = 0;
    event_queue.tail = 0;
}

int event_queue_read(void)
{
    int    res, index;

    if (event_queue.tail == event_queue.head)
    {
        // return and indicate no events in queue
        return EVENT_NONE;
    }

    res = event_queue.buff[event_queue.tail];
}

```

---

```

        index = event_queue.tail + 1;

        if ( index ==  max_event_queue_len)
        {
            index = 0;
        }

        event_queue.tail = index;

        return res;
    }

void event_queue_write(int val)
{
    int    index;

    index = event_queue.head + 1;

    if (index ==  max_event_queue_len)
    {
        index = 0;
    }

    if (index != event_queue.tail)
    {
        // the event queue is not full so append the event

        event_queue.buff[index] = val;

        event_queue.head = index;
    }
}

void interrupt_handler
{
    // this function is executed whenever an
    // interrupt occurs

    // these functions would now only write
    // events to the interrupt event queue

    read_switch();

    read_temperature();
}

void main()
{
    while (1)
    {
        // NOTE the use of two separate event queues

        // process all events generated by interrupts then
        // all events generated by main line code

        tmp_event = event_queue_interrupt_read();

        if (tmp_event == EVENT_NONE)
        {
            tmp_event = event_queue_read();
        }

        if (tmp_event != EVENT_NONE)

```

---

```
    {
        for (j=0; j<max_state_tbl_len; j++)
        {
            if (state_tbl[j].cur_state == current_state &&
                state_tbl[j].event      == tmp_event)
            {
                (*state_tbl[j].transition_func)(
                    current_state,
                    state_tbl[j].new_state,
                    tmp_event);

                current_state = state_tbl[j].new_state;

                state_tbl_indx = j;

                break;
            }
        }
        (*state_tbl[state_tbl_indx].monitor_func)(current_state);
    }
}
```

---

## ***Decreasing interrupt overheads***

Doing a lot of work while servicing an interrupt is usually a very bad idea. Consider a system that can execute 1,000,000 machine instructions per second – sounds like a huge number. Its interrupt handler consists of 5000 machine instructions and an interrupt occurs every 10ms. This means that the CPU spends  $(5000 * 1 / 0.01 = 500,000)$  machine instructions per second executing interrupt code. Since the CPU can only execute 1,000,000 machine instructions per second, this means that we only have 50% of the CPU's time for use in the main line. If we needed to increase the frequency of the interrupts to 5ms, our CPU would grind to a halt since it would be spending 100% of its time in the interrupt handler. Clearly spending as little time in the interrupt handler as possible is highly desirable.

To make the interrupt handler as lean as possible, developers often resort to generating packets of data within the interrupt handler and leaving these for the main line code to process. This might not seem like much of a gain because what we are gaining in the interrupt handler we are losing in the main line. However, in the main line we can choose whether to process data generated by the interrupt handler depending on how the main line is currently coping whereas we would have no choice but to process it if we doing so in the interrupt handler.

A state machine fits in very well with this interrupt handler generator / main line consumer approach. The state machine is executing in the main line and is the consumer, while the interrupt handler is the generator producing packets of data and events for the state machine to consume. Now the real elegance in all this is that the state machine only needs to consume packets determined by which state it is currently in and which events that state will respond to. Furthermore it becomes really trivial for the interrupt handler to change its behaviour depending on which state the main line is currently in. Think of this as though the state monitor function were split such that part of it executes in the main line and part in the interrupt handler. But remember, just because we can do this does not mean that we automatically should. Often it is better to let the state monitor function (executing in the main line) do the polling. We should only resort to splitting a state monitor function into main line / interrupt pair if we get a big gain.



---

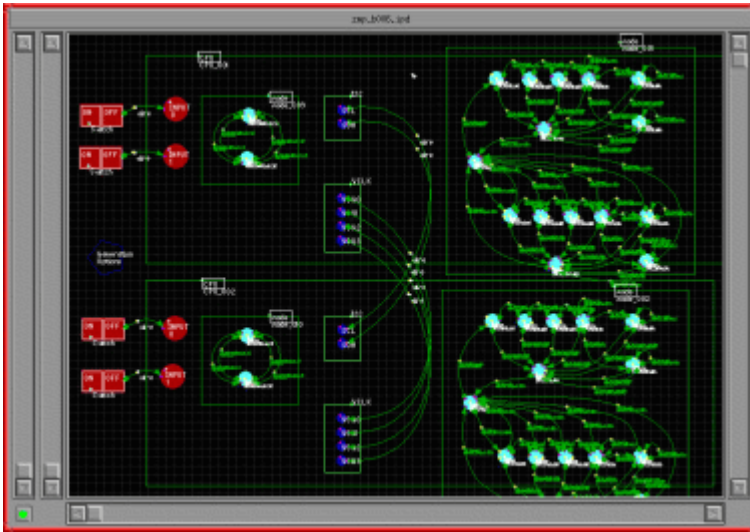
## Multiple interacting state machines

This System employs multiple interacting state machines to drive multi-master I2C communications as a low priority background task.

It was designed and debugged using the ZMech state machine development tool. The end result will run on real PIC MCUs. The same system could have been implemented using paper, pencil and an assembler but it would have taken a lot longer and the documentation wouldn't have looked as good. The pictures shown in this document are mostly unmodified screen shots. Any modification present are purely to highlight areas of interest.

---

## Overall system design



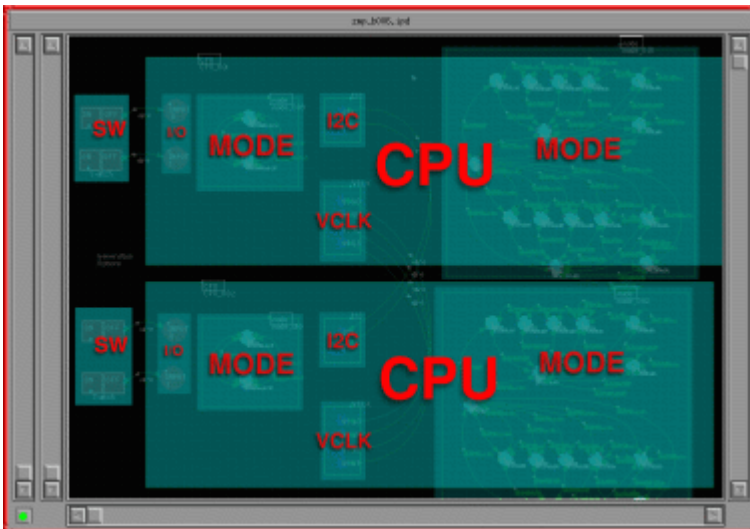
This image shows an exact snapshot of a complex system designed using ZMech.

To view this image in greater detail see [appendix 1](#)

To view a component state machine in greater detail see [appendix 3](#)

To view a detailed description of the high level comms state machine (MODE B) see [appendix 3](#)

## CPU view



This image is an annotated version of the above image. It has regions highlighted and labelled for the benefit of the user of this documentation.

Each highlighted region is a functional block. Some functional blocks contain one or more other functional blocks.

This diagram shows that, at the highest level, the system is made up of two CPU components and four SW (switch) components (two

groups of two).

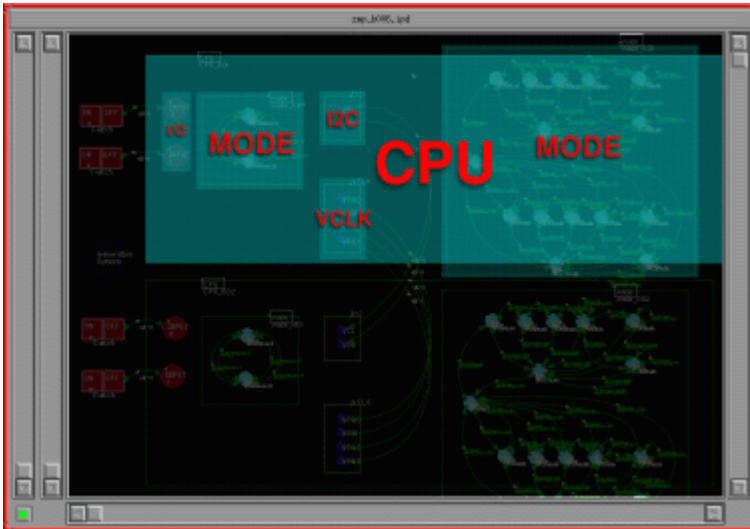
To view this image in greater detail see [appendix 2](#)

To view a detailed description of the high level comms state machine (MODE B) see [appendix 3](#)

In the HTML (WEB) version of this documentation this image as an **index to this documentation**, click on a functional block to see a description of that block

---

## CPU component breakdown



In this system both CPU components are identical and each is made up of

- two MODE components
- one I2C component
- one VCLK component
- two I/O components (one group of two)

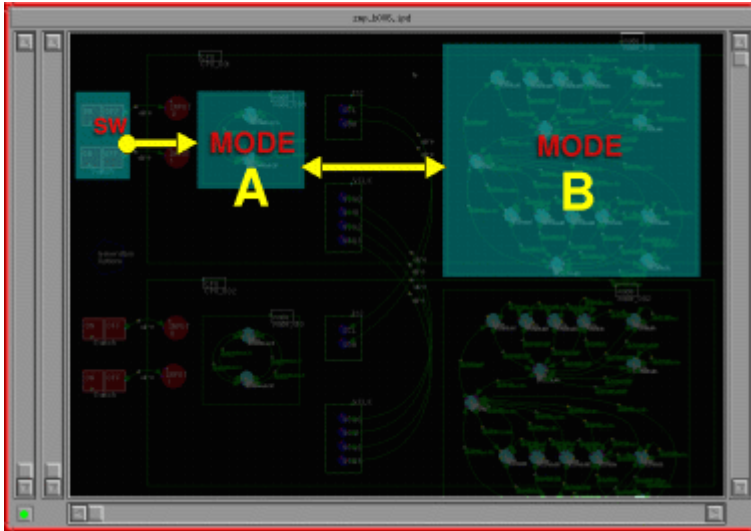


Each mode is a fully functional self contained state machine

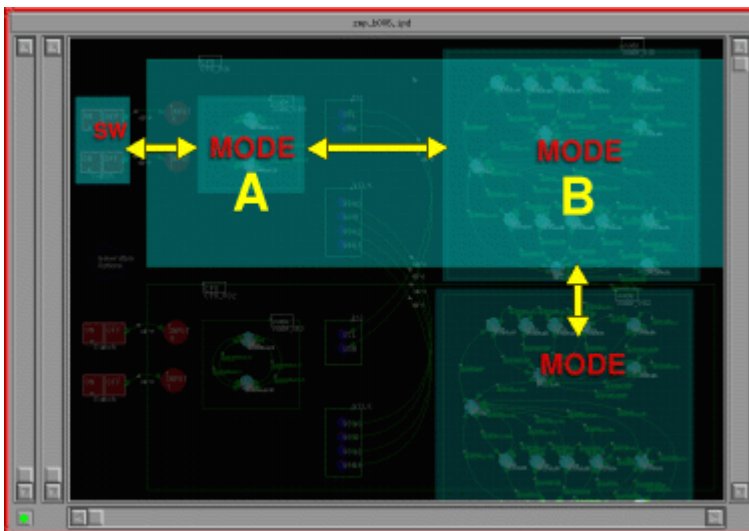
To view MODE A in greater detail [click here](#)

---

## ***Intra CPU mode interaction***



In this system, MODE A has been designed to handles I/O events from the external switches and comms events from MODE B. When an external switch changes from on to off or from off to on, MODE A generates a message and sends MODE B an event telling it a message is ready and waiting to be sent.



Each mode is a fully functional self contained state machine

In this system, MODE A has been designed to handles I/O events from the external switches, while MODE B has been designed to handle the high level communications protocol between the two CPUs.

MODE A generates events for MODE B in order to initiate a message transfer. MODE B responds by generating events to MODE A indicating when the message is

accepted for transmission, when transmission is complete, or if some error occurs. All the while MODE A is free to continue handling I/O events from the switches.

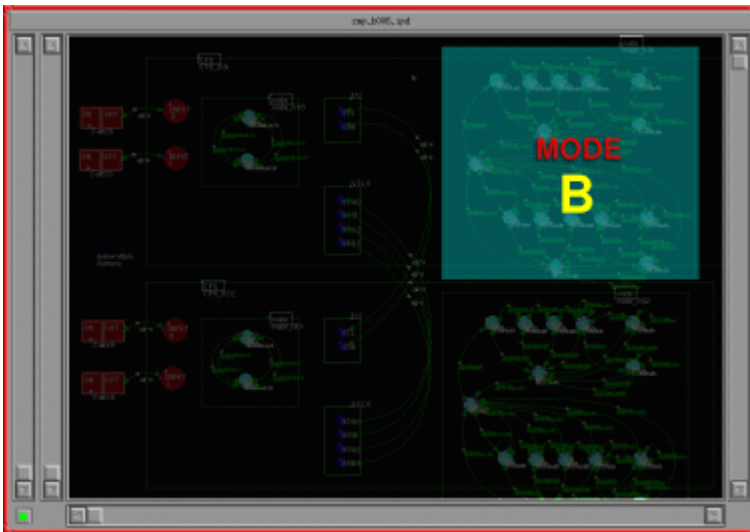
MODE B also generates events for MODE A when it receives a message from another CPU. These events are not responses to requests from MODE A they are initiated by MODE B which is monitoring comms traffic.

Both MODE A and B transfer data between each other via a shared buffer which is external to the event system. They know when it is safe to read from or write to the buffer due to their given state. This removes the need for shared resource locks (semaphores etc.) and complex polling schemes.

To view MODE B in greater detail see [appendix 3](#)

---

## part 5



machine see [Mode B Description](#)

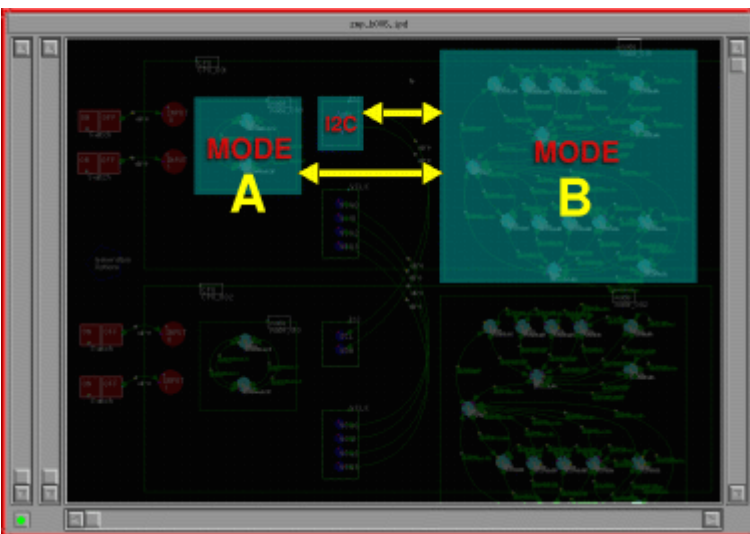
Each mode is a fully functional self contained state machine.

In this system, MODE B has been designed to handle the high level communications protocol between the two CPUs. Conceptually it communicates with its counterpart MODE B on another CPU.

To view MODE B in greater detail see [appendix 3](#)

To view a detailed description of the high level comms MODE B state

## Mode A / B Interaction



MODE B receives events from MODE A and also the I2C driver. It schedules bytes to be written to and read from the I2C driver. It operates in I2C MASTER mode when initiating a message transfer, and in I2C SLAVE mode when addressed by an I2C master.

MODE B receives events from the I2C driver when:

- the start of a message destined for this CPU has been detected
- a byte has been successfully sent
- a transmission error has been encountered
- a start or stop condition has been received
- a byte has been received
- a busy condition is detected (traffic on the I2C bus between two or more other CPUs is detected and is in progress).

---

MODE A generates events for MODE B in order to initiate a message transfer.

MODE B generates events for MODE A when

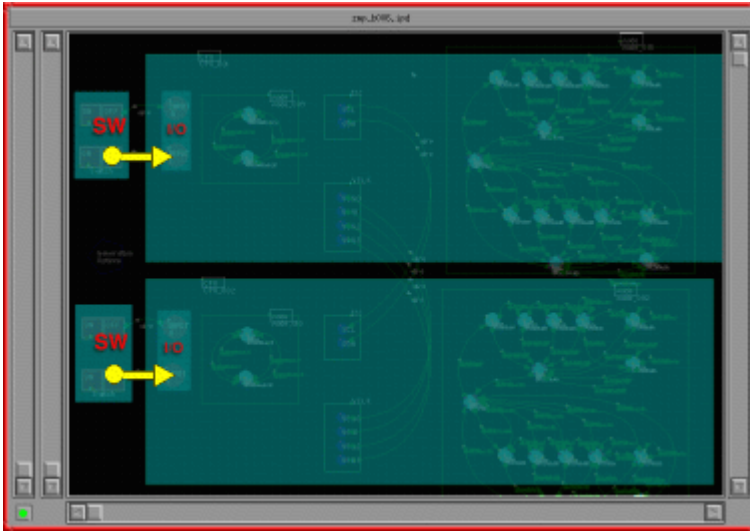
- a MODE A generated message has been accepted for transmission
- transmission of a message is complete
- a message transfer if aborted due to an I2C bus error
- a message is received from another CPU

Both MODE A and MODE B transfer data between each other via a shared buffer which is external to the event system. They know when it is safe to read from or write to the buffer due to their given state. This removes the need for shared resource locks (semaphores etc.) and complex polling schemes.

To view a detailed description of the high level comms MODE B state machine see [Mode B Description](#)

---

## I/O Component Description



Here we see the connection between the switch components and the I/O components

The switch components are used as external hardware mimics. While interactively debugging the system, the user can activate a switch component by clicking on it and the input component (connected to it) will see a switch toggle.

The I/O components generate events in response to input changes. These events are passed to any interested

state machine for processing by the state machine scheduler.

Other mimics such as LEDs are also available but not shown here. LED mimics change colour when a logic 0, 1 or high impedance is applied to them. They are normally connected to the output components of a CPU (not shown) or the outputs of some other component such as an I2C component.

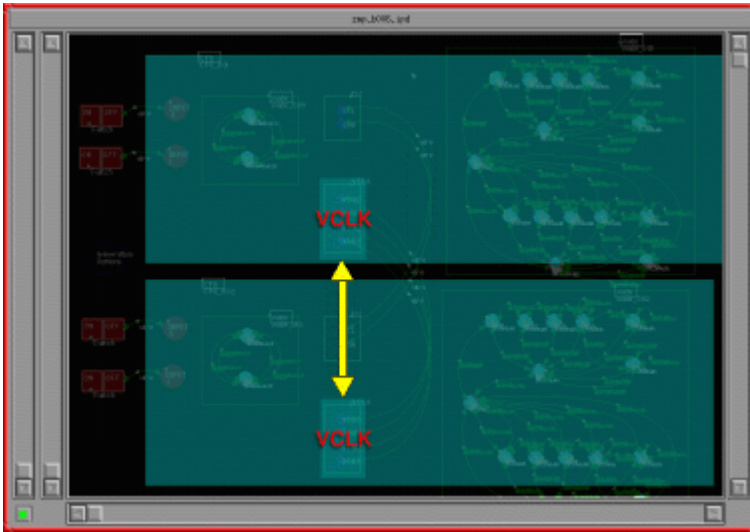
I/O components are mapped directly into the I/O ports of the simulated CPU. When the simulated CPU writes to an I/O port the corresponding I/O components route the signals to other connected I/O components and can be subsequently read by other simulated CPUs during an I/O port read.

I/O components generates machine code which is directly executable in the target system. No additional debug code is generated for the target when the simulator is used in place of a real CPU.



---

## VCLK Component Description



Here we see the VCLK components of both CPUs tied together.

The VCLK allows the generation of a system wide virtual clock. It allows several CPUs of varying speeds to be connected to a common free running clock signal. This clock ensures that all CPUs are never more than one virtual clock cycle apart.

A CPU with a VCLK component can opt into or out of the system wide virtual clock. By opting in, it ensures that it will keep pace with the rest of

the system. By opting out, it enables the rest of the system to run at maximum speed. A CPU may continually opt in and out depending on its work load and impact on the rest of the system.

All CPUs with a participating VCLK component have an associated VCLK counter plus offset. The offset is imaginary and once the VCLK starts participating in the system wide virtual clock, the offset is fixed and remains constant. If the whole system is stopped at any time the difference between any two participating vclk counters (plus associated offsets) will never be greater than one.

The VCLK is driven by the state machine scheduler as a background task.

When using the VCLK to synchronise message broadcasting, the software should wait for at least two VCLK cycles (the VCLK counter should change by two). This will guarantee that all other participating CPUs have seen the message.

The VCLK component generates machine code which is directly executable in the target system. No additional debug code is generated for the target when the simulator is used in place of a real CPU. Signalling between different CPUs is performed by reading from and writing to real CPU I/O ports. The simulator uses I/O components mapped to the CPU's I/O ports to route the signals between simulated CPUs. (see I/O components above for further information).



---

## I2C Component Description



In this example, comms between the two CPUs is provided by an I2C link.

Here the I2C bit banging is tied to the virtual clock (VCLK) but could easily be tied to a timer interrupt (or even polled if necessary).

VCLK driven bit banging, reduces CPU overhead at the cost of extra I/O interconnections. It allows several CPUs of varying speeds to be connected to a common free running clock signal. This clock ensures that all CPUs are always no more than

one clock cycle apart. The VCLK is driven by the state machine scheduler. Consequently tying the I2C bit banger to the VCLK causes comms to be driven as a background task. Tying the I2C bit banger to a timer interrupt causes comms to be driven as a foreground time critical task. Driving the comms in the background allows the CPU to drive other critical I/O with greater ease and accuracy (e.g. driving stepper motors or timing external events).

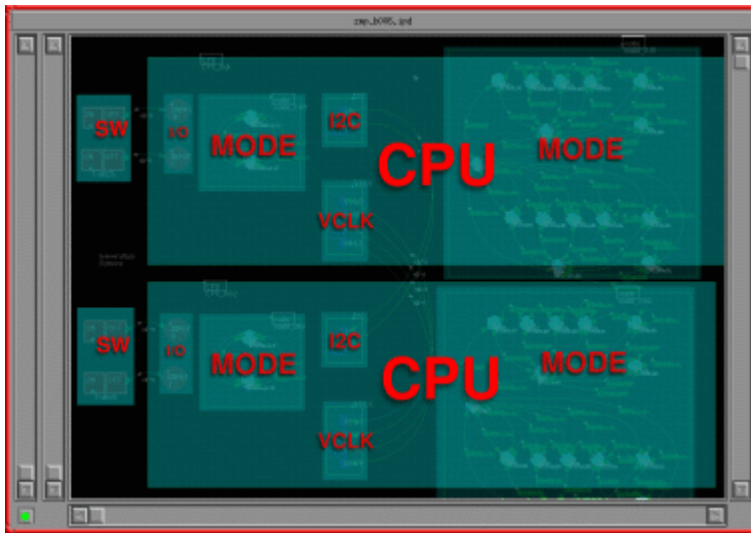
Driving the I2C comms (bit banger) via the timer interrupt could cause problems in systems consisting of several CPUs with large speed differences since a data producer could swamp a data consumer. The VCLK reduces the buffering requirements of each CPU since received data can be consumed by the state machine as it arrives i.e. VCLK driven comms is synchronous with the state machine scheduler.

I2C and VCLK are multi-drop buses. This means that several CPUs can be connected in parallel to these buses, and provided they stick to the defined protocol they will not interfere with each other.

The VCLK component and I2C component generates machine code which is directly executable in the target system. No additional debug code is generated for the simulator. Signalling between different CPUs is performed by reading from and writing to real CPU I/O ports. The simulator uses I/O components mapped to the CPU's I/O ports to route the signals between simulated CPUs. (see I/O components above for further information).

---

## Animated Message Flow Description



**NOTE:** this animation is only active if viewed as a WEB page ([click here for WEB view](#))

Here we see the flow of a message generated by an I/O event.

Once the code responsible for looking after the I/O has generated the message and passed it to the comms handler, it can get back to looking after its I/O. In this design, multi-tasking is effectively performed by the state machine scheduler. As a consequence the I/O

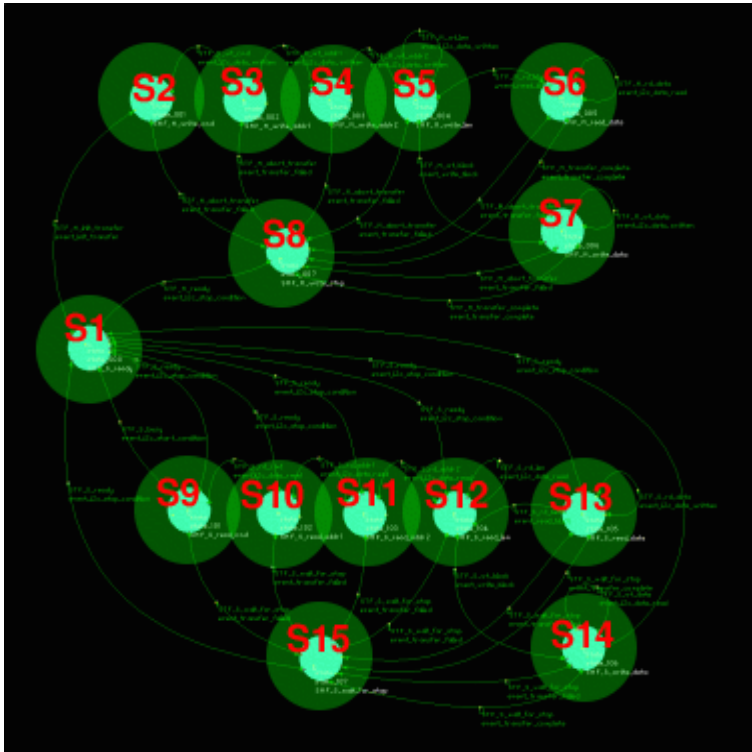
handling code is clean and lean and does not have to be interleaved with comms code.

This design uses just two state machines per CPU, but it is possible to use many interacting state machines per CPU. The exact number is dependent on the amount of physical memory available in the CPU.

Using background comms it is a simple matter to add additional CPUs to increase the computing power of the system.

By replacing the high level comms state machine in this example, with one that simply passes events, it is possible to build very large distributed state machines with very little effort using ZMech.

## Mode B description



To view this state machine in greater detail see [appendix 3](#)

**S1** Idle state

**S2** Init transfer in MASTER mode, send "I2C command" byte

**S3** Send "transfer address" low byte

**S4** Send "transfer address" high byte

**S5** Send "byte count" byte

**S6** Send block ("byte count" number of bytes, see **S5**)

**S7** Receive block ("byte count" number of bytes, see **S5**)

**S8** Finish transfer (set I2C stop condition)

**S9** Init transfer in SLAVE mode, receive "I2C command" byte

**S10** Receive "transfer address" low byte

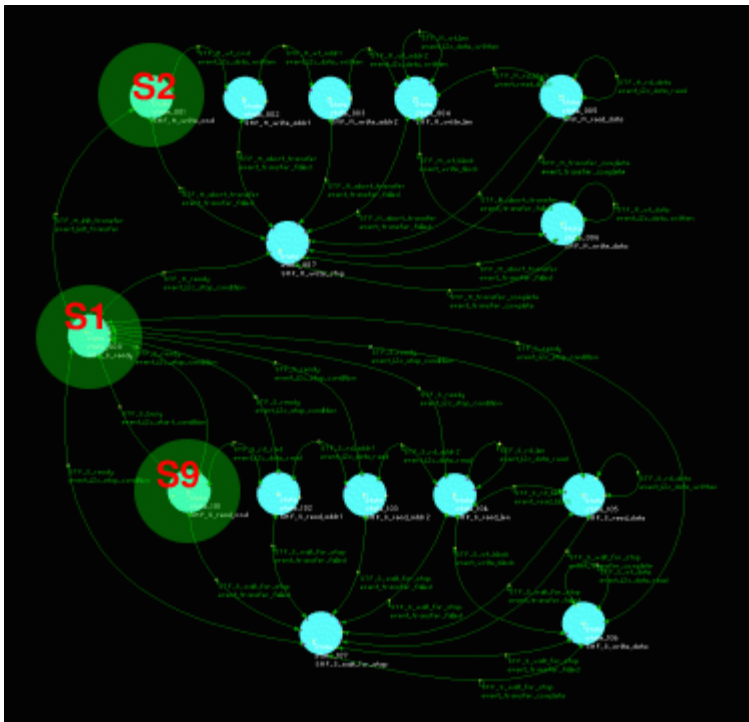
**S11** Receive "transfer address" high byte

**S12** Receive "byte count" byte

**S13** Receive block ("byte count" number of bytes, see **S12**)

**S14** Send block ("byte count" number of bytes, see **S12**)

**S15** Finish transfer (wait for I2C stop condition)



## S1

### Idle state

Event from MODE A requesting message transfer causes transition to **S2**. State transition function used is

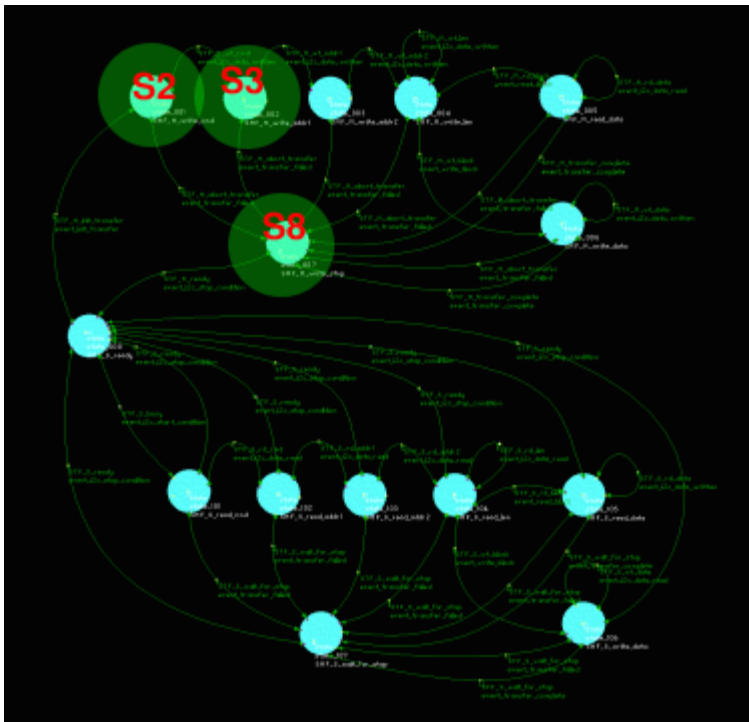
**"STF\_M\_init\_transfer"**

Event from I2C driver indicating incoming message (I2C start condition detected) causes transition to **S9**. State transition function used is **"STF\_S\_busy"**

To view this state machine in greater detail see [appendix 3](#)

jump to:

**S1** **S2** **S3** **S4** **S5**  
**S6** **S7** **S8** **S9** **S10**  
**S11** **S12** **S13** **S14**  
**S15**



## S2

Init transfer in MASTER mode, send "I2C command" byte

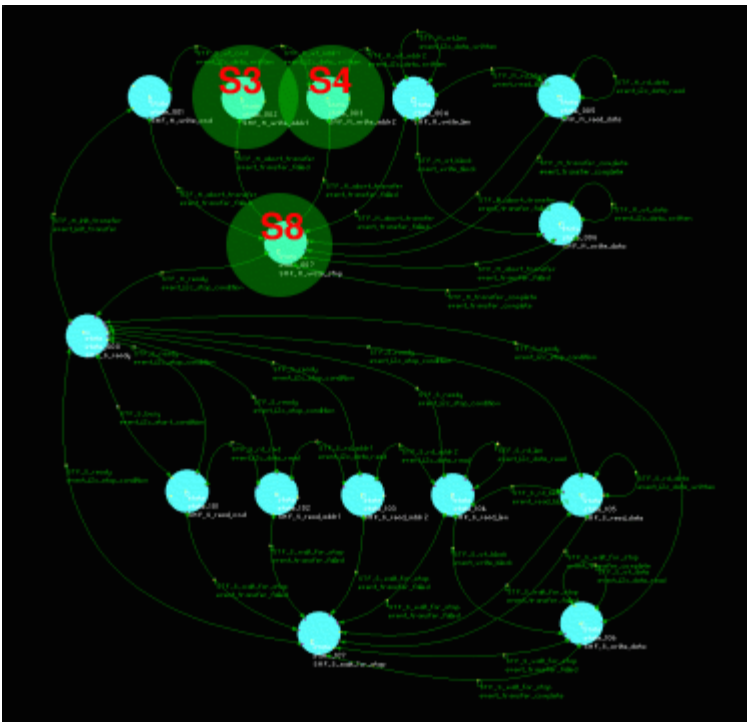
Event from I2C driver indicating successful transmission of command byte causes transition to [S3](#). State transition function used is ["STF\\_M\\_wt\\_block"](#)

Event from I2C driver indicating failure causes transition to [S8](#). State transition function used is ["STF\\_M\\_abort\\_transfer"](#)

To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)



### S3

Send "transfer address" low byte

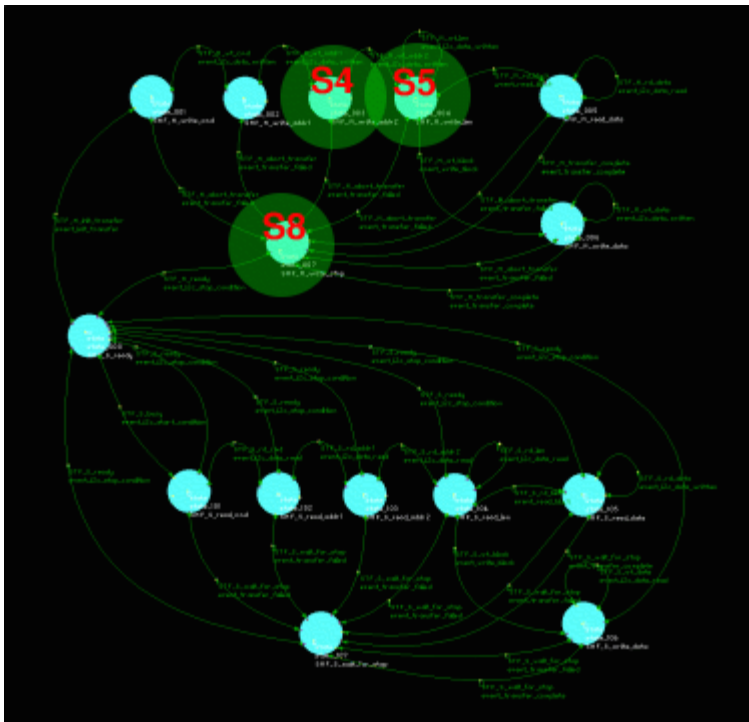
Event from I2C driver indicating successful transmission of byte causes transition to [S4](#). State transition function used is ["STF\\_M\\_wt\\_addr1"](#)

Event from I2C driver indicating failure in transmission causes transition to [S8](#). State transition function used is ["STF\\_M\\_abort\\_transfer"](#)

To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)



## S4

Send "transfer address" high byte

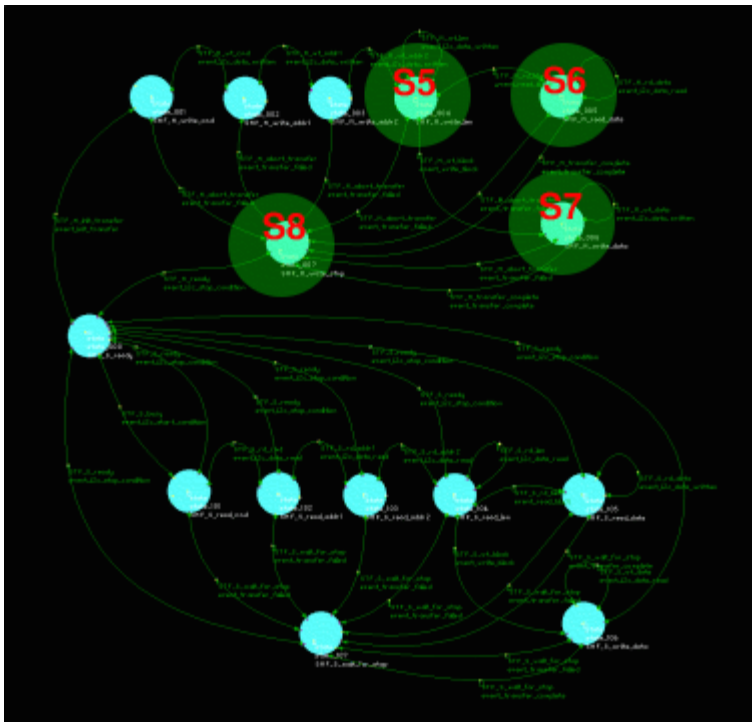
Event from I2C driver indicating successful transmission of byte causes transition to **S5**. State transition function used is **"STF\_M\_wt\_addr2"**

Event from I2C driver indicating failure in transmission causes transition to **S8**. State transition function used is **"STF\_M\_abort\_transfer"**

To view this state machine in greater detail see [appendix 3](#)

jump to:

**S1** **S2** **S3** **S4** **S5**  
**S6** **S7** **S8** **S9** **S10**  
**S11** **S12** **S13** **S14**  
**S15**



To view this state machine in greater detail see [appendix 3](#).

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

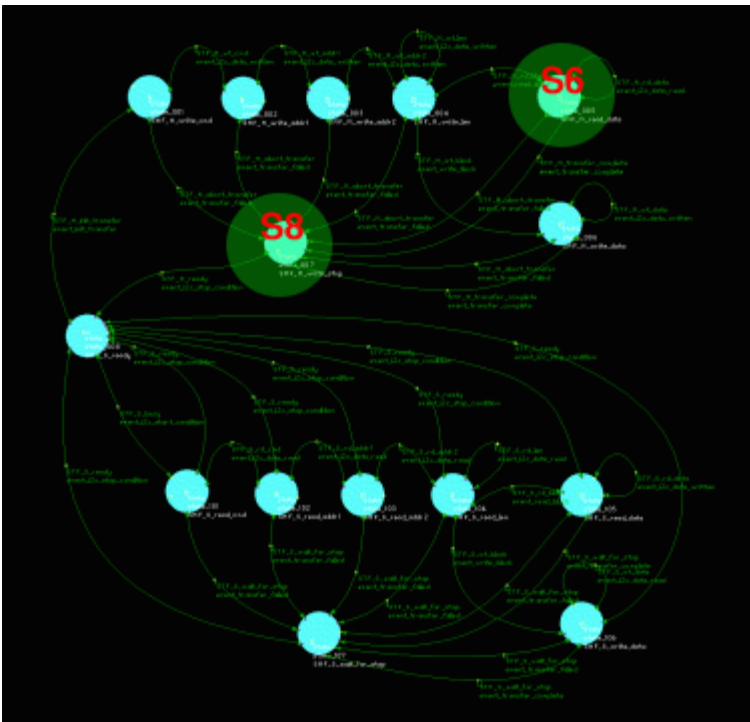
## S5

### Send "byte count" byte

Event from I2C driver indicating successful transmission of byte causes transition to [S6](#) or [S7](#) depending on the data direction bit in the I2C command already sent. State transition function used is ["STF\\_M\\_wt\\_len"](#). This state transition function generates one of two events (depending on the data direction bit in the I2C command) which causes a state transition to either [S6](#), using the state transition function ["STF\\_M\\_rd\\_block"](#) or [S7](#), using the state transition function ["STF\\_M\\_wt\\_block"](#).

Event from I2C driver indicating failure in transmission causes transition to [S8](#). State transition function used is ["STF\\_M\\_abort\\_transfer"](#)





To view this state machine in greater detail see [appendix 3](#).

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

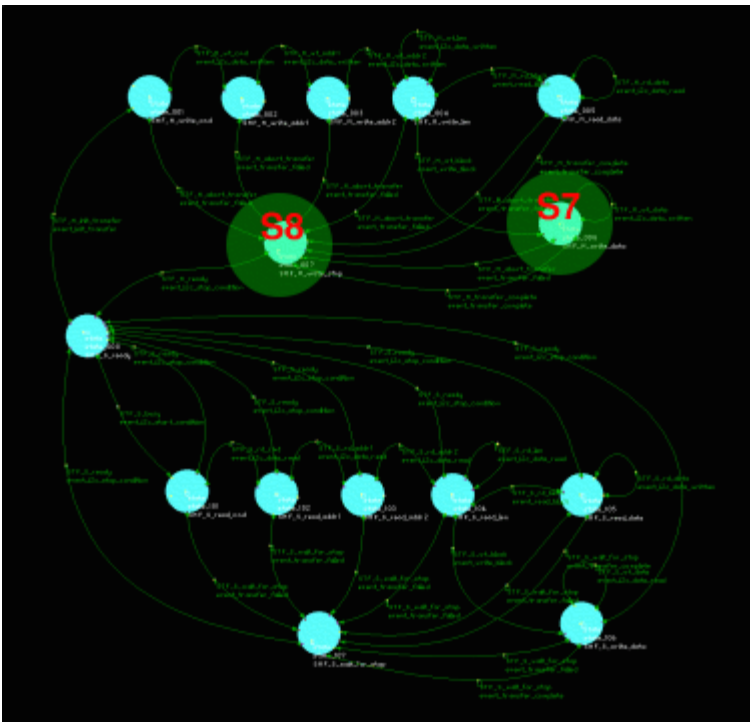
## S6

**Send block ("byte count" number of bytes, see [S5](#))**

Event from I2C driver indicating successful transmission of byte causes transition back to [S6](#) or forward to [S8](#) until the "byte count" (already sent) reaches zero. Each time a transition to [S6](#) is performed, the next byte in the output block is sent and the "byte count" is decremented. The state transition function used is ["STF\\_M\\_rd\\_data"](#).

When the "byte count" reaches zero the ["STF\\_M\\_rd\\_data"](#) state transition function generates an "event\_transfer\_complete" event that causes a transition to [S8](#). The state transition function used when going from [S6](#) to [S8](#) is ["STF\\_M\\_transfer\\_complete"](#).

Event from I2C driver indicating failure in transmission causes transition to [S8](#). State transition function used is ["STF\\_M\\_abort\\_transfer"](#)



To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

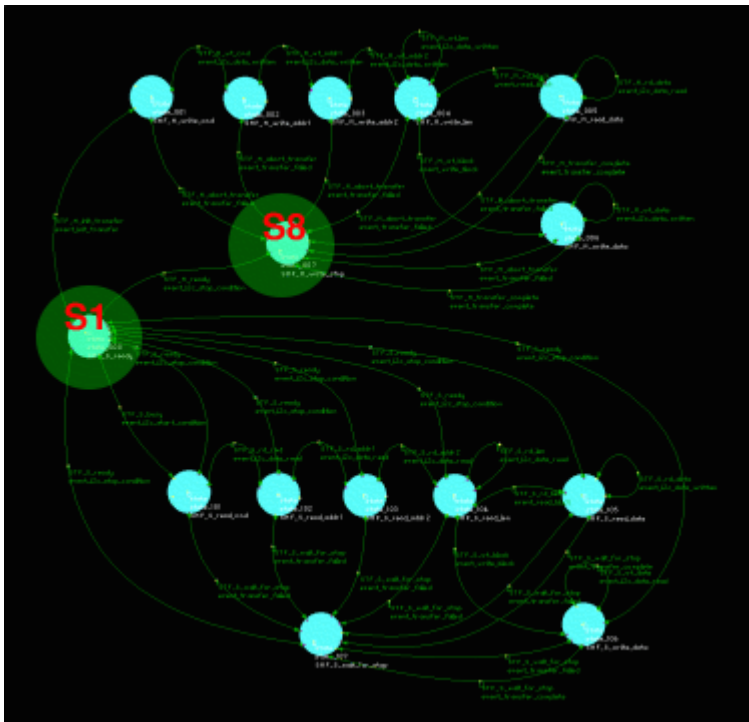
## S7

**Receive block ("byte count" number of bytes, see [S5](#))**

Event from I2C driver indicating successful receipt of byte causes transition back to [S7](#) or forward to [S8](#) until the "byte count" (already sent) reaches zero. Each time a transition to [S7](#) is performed, the byte received from the I2C driver is appended to the input block and the "byte count" is decremented. The state transition function used is ["STF\\_M\\_wt\\_data"](#).

When the "byte count" reaches zero the ["STF\\_M\\_wt\\_data"](#) state transition function generates an "event\_transfer\_complete" event that causes a transition to [S8](#). The state transition function used when going from [S7](#) to [S8](#) is ["STF\\_M\\_transfer\\_complete"](#).

Event from I2C driver indicating failure in transmission causes transition to [S8](#). State transition function used is ["STF\\_M\\_abort\\_transfer"](#)



## S8

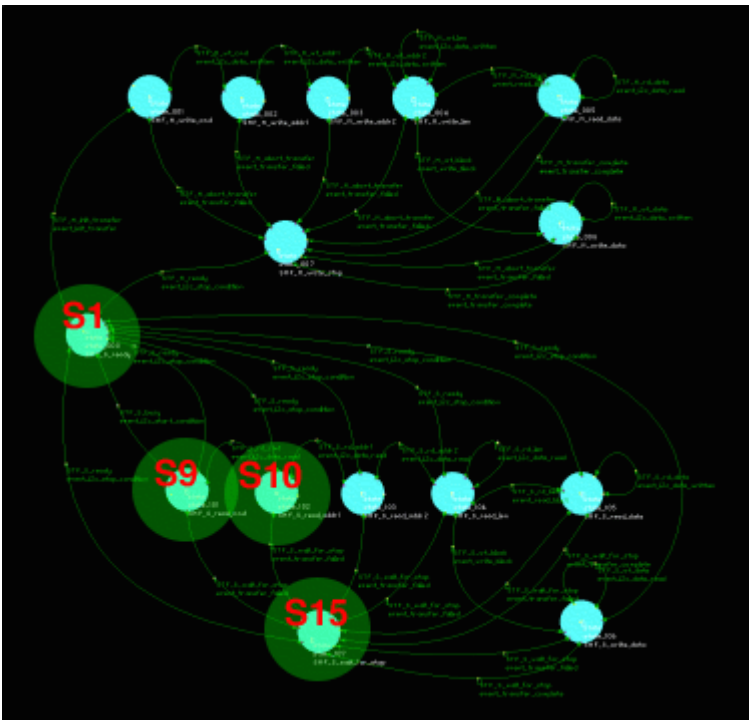
**Finish transfer (set I2C stop condition)**

Event from I2C driver indicating I2C stop condition established causes transition to [S1](#). State transition function used is ["STF\\_M\\_ready"](#).

To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)



To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

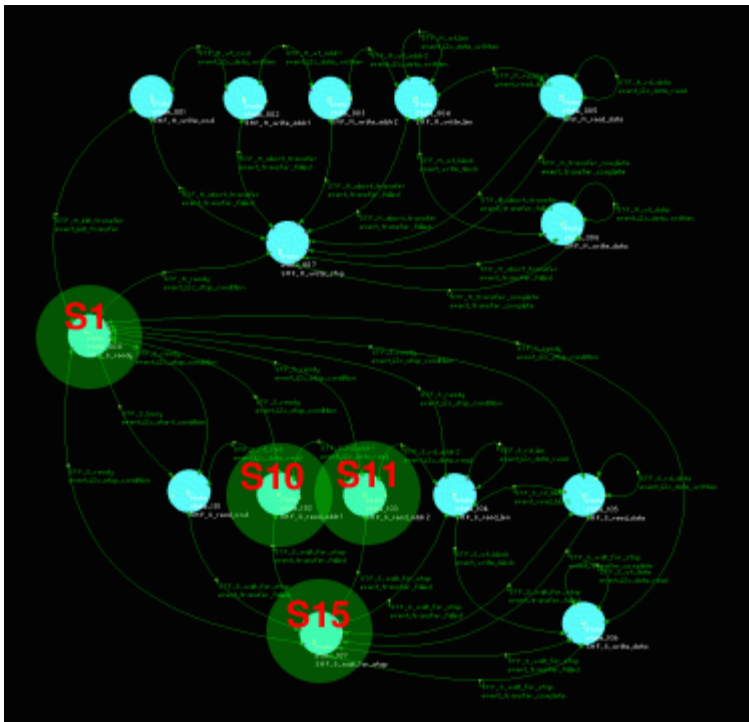
## S9

**Init transfer in SLAVE mode, receive "I2C command" byte**

Event from I2C driver indicating successful receipt of valid command byte causes transition to [S10](#). The state transition function used is ["STF\\_S\\_rd\\_cmd"](#).

Event from I2C driver indicating failure OR command byte received but addressed to other CPU causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to [S1](#). The state transition function used is ["STF\\_S\\_ready"](#)



## S10

### Receive "transfer address" low byte

Event from I2C driver indicating successful receipt of byte causes transition to [S11](#). The state transition function used is ["STF\\_S\\_rd\\_addr1"](#)

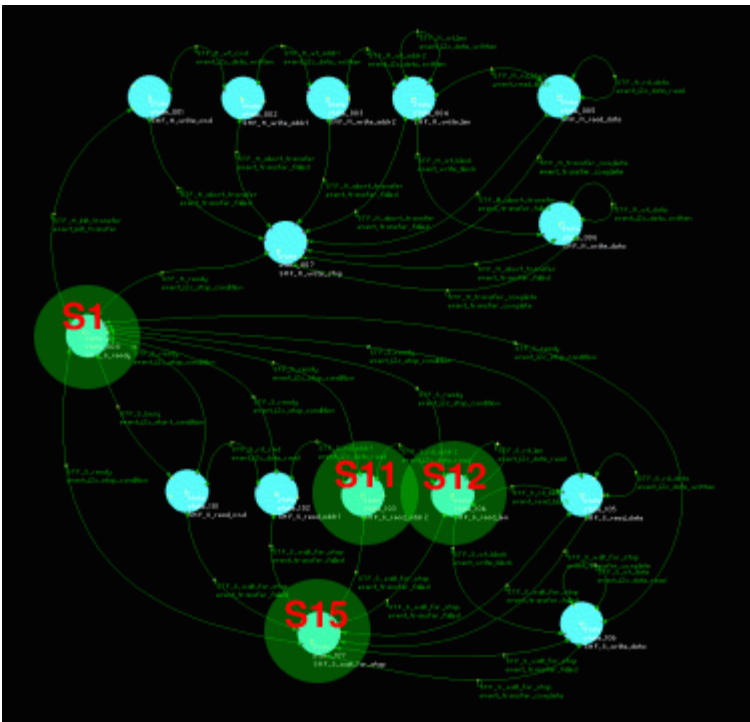
Event from I2C driver indicating failure in receipt causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to XS1. The state transition function used is ["STF\\_S\\_ready"](#)

To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)



To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

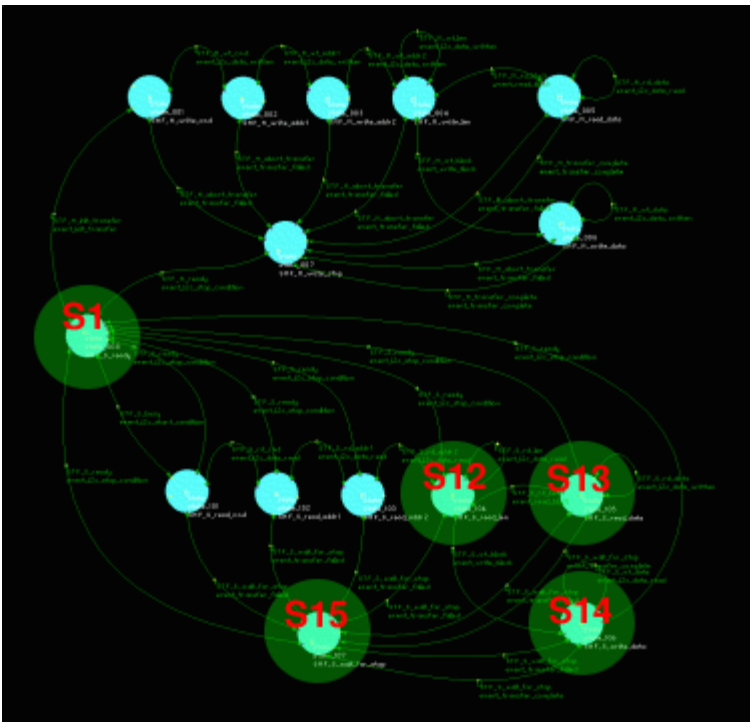
## S11

### Receive "transfer address" high byte

Event from I2C driver indicating successful receipt of byte causes transition to [S12](#). The state transition function used is ["STF\\_S\\_rd\\_addr2"](#)

Event from I2C driver indicating failure in receipt causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to [S1](#). The state transition function used is ["STF\\_S\\_ready"](#)



To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

## S12

### Receive "byte count" byte

Event from I2C driver indicating successful receipt of byte causes transition to [S13](#) or [S14](#) depending on the data direction bit in the I2C command already sent. The state transition function used is ["STF\\_S\\_rd\\_len"](#)

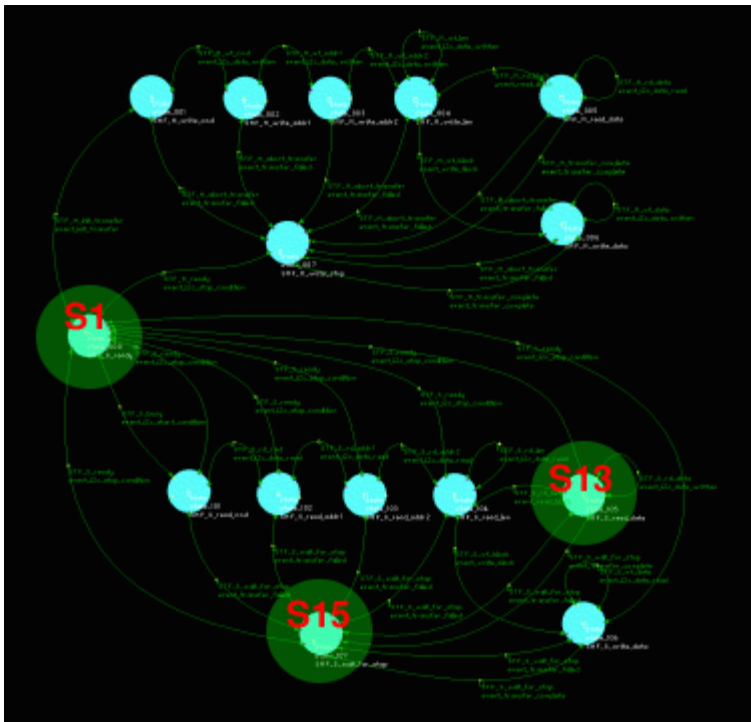
The state transition function ["STF\\_S\\_rd\\_len"](#) generates the event "event\_read\_block" if the data direction bit in the I2C command indicates a read, otherwise it generates the event "event\_write\_block" if the data direction bit in the I2C command indicates a write.

The "event\_read\_block" event causes a transition to [S13](#). The state transition function used is ["STF\\_S\\_rd\\_block"](#).

The "event\_write\_block" event causes a transition to [S14](#). The state transition function used is ["STF\\_S\\_wt\\_block"](#)

Event from I2C driver indicating failure in receipt causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to [S1](#). The state transition function used is ["STF\\_S\\_ready"](#)



To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

## S13

**Receive block ("byte count" number of bytes, see [S12](#))**

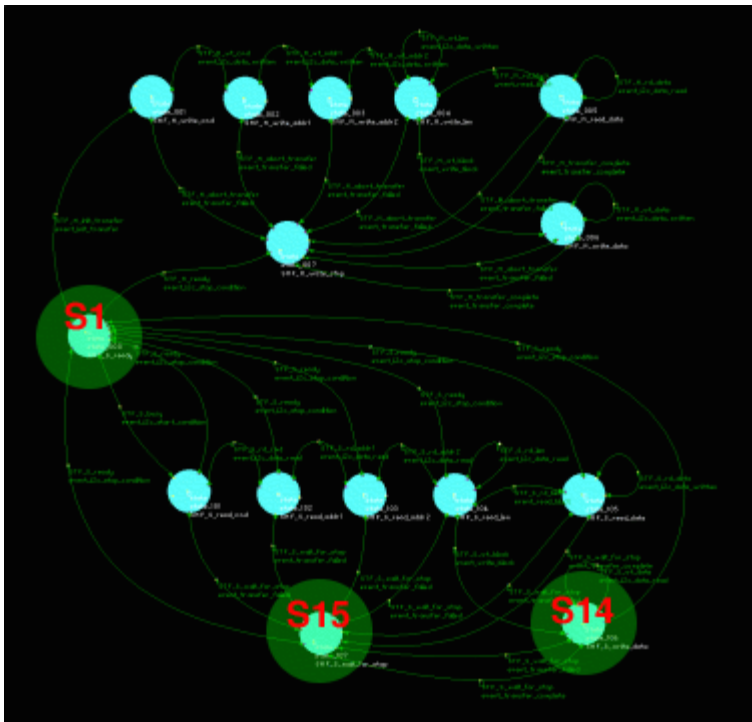
Event from I2C driver indicating successful receipt of byte causes transition back to [S13](#) or forward to [S15](#) until the "byte count" (already sent) reaches zero. Each time a transition to [S13](#) is performed, the byte received from the I2C driver is appended to the input block and the "byte count" is decremented. The state transition function used is ["STF\\_S\\_rd\\_data"](#).

When the "byte count" reaches zero the ["STF\\_S\\_rd\\_data"](#) state transition function generates an "event\_transfer\_complete" event that causes a transition to [S15](#). The state transition function used when going from [S13](#) to [S15](#) is STF\_S\_TRANSFER\_COMPLETE.

Event from I2C driver indicating failure in receipt causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to [S1](#). The state transition function used is ["STF\\_S\\_ready"](#)





To view this state machine in greater detail see [appendix 3](#).

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

## S14

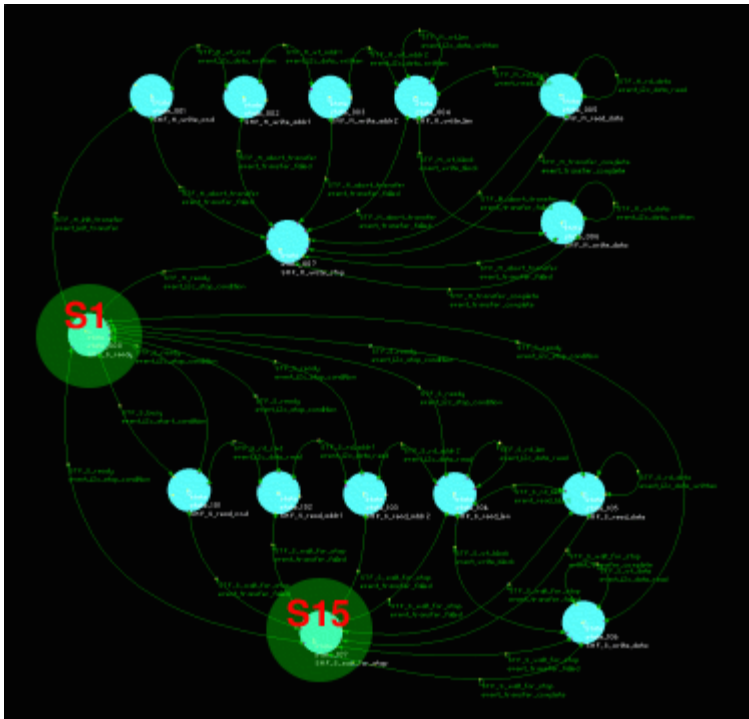
**Send block ("byte count" number of bytes, see [S12](#))**

Event from I2C driver indicating successful transmission of byte causes transition back to [S14](#) or forward to [S15](#) until the "byte count" (already sent) reaches zero. Each time a transition to [S14](#) is performed, the next byte in the output block is sent and the "byte count" is decremented. The state transition function used is ["STF\\_S\\_wt\\_data"](#).

When the "byte count" reaches zero the ["STF\\_S\\_rd\\_data"](#) state transition function generates an "event\_transfer\_complete" event that causes a transition to [S15](#). The state transition function used when going from [S14](#) to [S15](#) is [STF\\_S\\_TRANSFER\\_COMPLETE](#).

Event from I2C driver indicating failure in receipt causes transition to [S15](#). The state transition function used is ["STF\\_S\\_wait\\_for\\_stop"](#)

Event from I2C driver indicating unexpected stop condition seen causes transition to [S1](#). The state transition function used is ["STF\\_S\\_ready"](#)



## S15

**Finish transfer (wait for I2C stop condition)**

Event from I2C driver indicating I2C stop condition detected causes transition to S1. The state transition function used is "STF\_S\_ready"

To view this state machine in greater detail see [appendix 3](#)

jump to:

[S1](#) [S2](#) [S3](#) [S4](#) [S5](#)  
[S6](#) [S7](#) [S8](#) [S9](#) [S10](#)  
[S11](#) [S12](#) [S13](#) [S14](#)  
[S15](#)

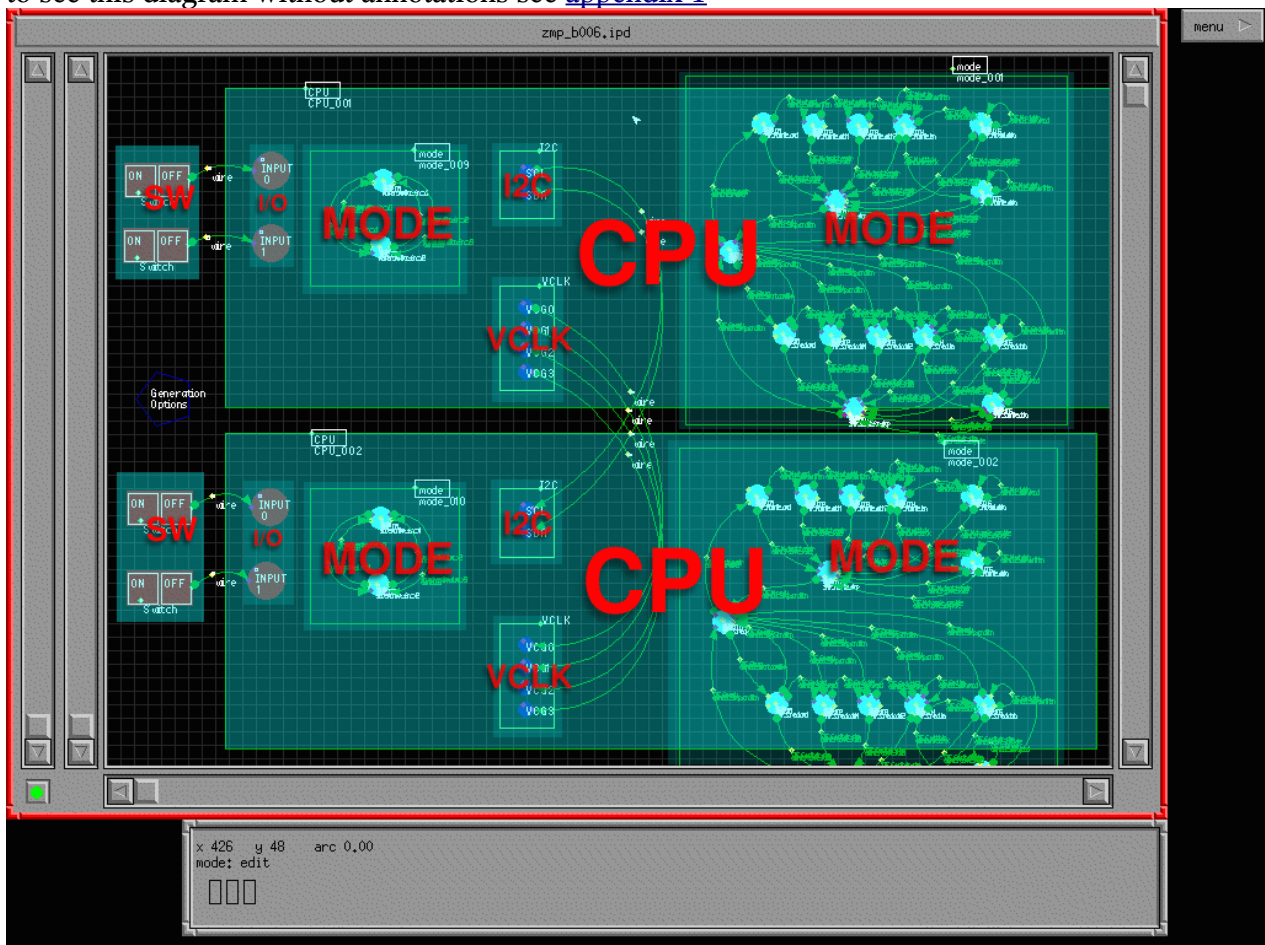
# Appendix 1

to see this diagram in annotated form see [appendix 2](#)

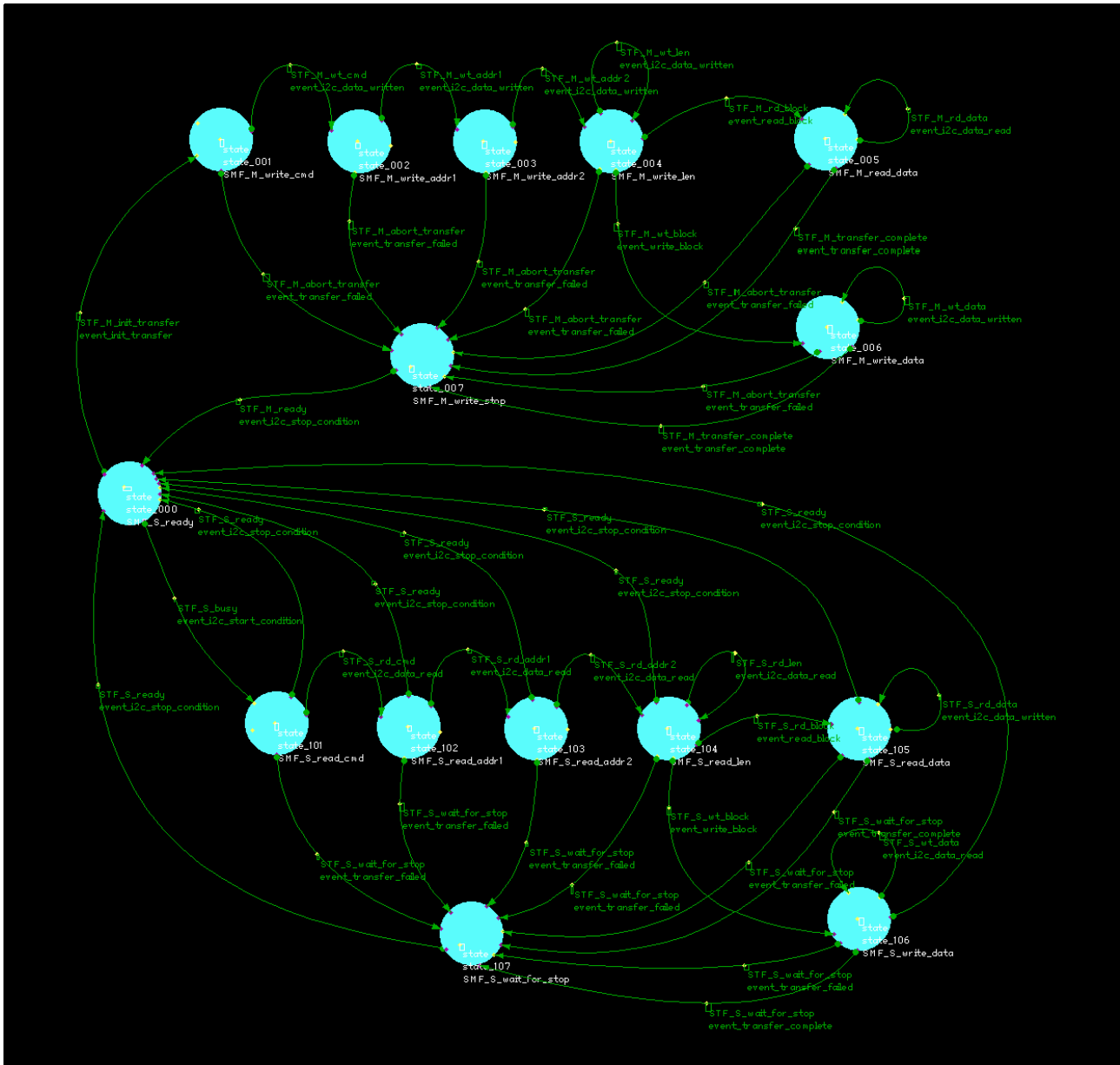


## Appendix 2

to see this diagram without annotations see [appendix 1](#)



## Appendix 3



---

## Appendix 4

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_stop  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_abort_transfer  
    call master_write_stop_entry_point  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_init_transfer  
    movfw    xfr_cmd  
    movwf    data_out  
    call master_write_cmd_entry_point  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_init_transfer  
    return
```

---

```
;-----  
;State Transition Function  
; I2C bus MASTER specific  
  
STF_M_rd_block  
    tstf xfr_len  
    bz mrdb_1  
  
        movlw event_transfer_complete  
        call server_event_queue_write  
  
        movlw event_transfer_complete  
        call client_event_queue_write  
        goto mrdb_9  
  
mrdb_1  
    call master_read_byte_entry_point  
  
mrdb_9  
    return
```

---

```

;-----
;State Transition Function
; I2C bus MASTER specific

STF_M_rd_data
    movfw    xfr_addr1
    movwf    fsr
    movfw    data_in
    movwf    INDF

    incf xfr_addr1
    decfsz   xfr_len
    goto mrdd_6
    goto mrdd_7

mrdd_6
    call master_read_byte_entry_point
    goto mrdd_9

mrdd_7
    movlw    event_transfer_complete
    call server_event_queue_write

    movlw    event_transfer_complete
    call client_event_queue_write

mrdd_9
    return

;-----
;State Monitor Function
; I2C bus MASTER specific

SMF_M_read_data
    return

;-----
;State Transition Function
; I2C bus MASTER specific

STF_M_ready
    call slave_read_cmd_entry_point
    return

```

---



---

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_ready  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_transfer_complete  
    call master_write_stop_entry_point  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_wt_addr1  
    movfw    xfr_addr2  
    movwf    data_out  
    call master_write_byte_entry_point  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_addr1  
    return
```

---

```
;-----  
;State Transition Function  
; I2C bus MASTER specific  
  
STF_M_wt_addr2  
    movfw    xfr_len  
    movwf    data_out  
    call master_write_byte_entry_point  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_addr2  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_wt_block  
    movlw    event_i2c_data_written  
    call server_event_queue_write  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_wt_cmd  
    movfw    xfr_addr1  
    movwf    data_out  
    call master_write_byte_entry_point  
    return
```

---

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_cmd  
    return
```

```
;-----  
;State Transition Function  
; I2C bus MASTER specific
```

```
STF_M_wt_data  
    tstf xfr_len  
    bnz mwtd_1  
  
    movlw    event_transfer_complete  
    call server_event_queue_write  
  
    movlw    event_transfer_complete  
    call client_event_queue_write  
    goto mwtd_9
```

```
mwtd_1  
    movfw    xfr_addr1  
    movwf    fsr  
    movfw    INDF  
    movwf    data_out  
    addlw    1  
    call master_write_byte_entry_point  
    incf xfr_addr1  
    decf xfr_len
```

```
mwtd_9    return
```

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_data  
    return
```

---

```
;-----  
;State Transition Function  
; I2C bus MASTER specific  
  
STF_M_wt_len  
    movlw    event_write_block  
    btfsc    xfr_cmd, 0  
    movlw    event_read_block  
  
    call server_event_queue_write  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus MASTER specific
```

```
SMF_M_write_len  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_busy  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_init_transfer  
    movlw    event_busy  
    call client_event_queue_write  
  
    movlw    event_i2c_start_condition  
    call server_event_queue_write  
    return
```

---

```
;-----  
;State Monitor Function  
; I2C bus SLAVE specific
```

```
SMF_S_init_transfer  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_rd_addr1  
    movfw    data_in  
    movwf    xfr_addr1  
    call slave_read_byte_entry_point  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus SLAVE specific
```

```
SMF_S_read_addr1  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_rd_addr2  
    movfw    data_in  
    movwf    xfr_addr2  
    call slave_read_byte_entry_point  
    return
```

---

```
;-----  
;State Monitor Function  
; I2C bus SLAVE specific
```

```
SMF_S_read_addr2  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_rd_block  
    movlw    event_i2c_data_written  
    call server_event_queue_write  
    return
```

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific
```

```
STF_S_rd_cmd  
    movfw    data_in  
    movwf    xfr_cmd  
    call slave_read_byte_entry_point  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus SLAVE specific
```

```
SMF_S_read_cmd  
    return
```

---

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific  
  
STF_S_rd_data  
    tstf xfr_len  
    bnz stf_srdd_1  
  
    movlw    event_transfer_complete  
    call server_event_queue_write  
  
    movlw    event_transfer_complete  
    call client_event_queue_write  
    goto stf_srdd_9  
  
stf_srdd_1  
    movfw    xfr_addr1  
    movwf    fsr  
    movfw    INDF  
    movwf    data_out  
    call slave_write_byte_entry_point  
  
    incf xfr_addr1  
    decf xfr_len  
  
stf_srdd_9  
    return  
  
  
;-----  
;State Monitor Function  
; I2C bus SLAVE specific  
  
SMF_S_read_data  
    return
```

---

```

;-----
;State Transition Function
; I2C bus SLAVE specific

STF_S_rd_len
    movfw    data_in
    movwf    xfr_len

    movlw    event_write_block
    btfsc    xfr_cmd, 0
    movlw    event_read_block

    call server_event_queue_write
    return

;-----
;State Monitor Function
; I2C bus SLAVE specific

SMF_S_read_len
    return

;-----
;State Transition Function
; I2C bus SLAVE specific

STF_S_ready
    call slave_read_cmd_entry_point
    return

;-----
;State Monitor Function
; I2C bus SLAVE specific

SMF_S_ready
    return

```



---

```

;-----
;State Transition Function
;  I2C bus SLAVE specific

STF_S_wait_for_stop
    call slave_wait_for_stop_entry_point
    return

;-----
;State Monitor Function
;  I2C bus SLAVE specific

SMF_S_wait_for_stop
    return

;-----
;State Transition Function
;  I2C bus SLAVE specific

STF_S_wt_block
    tstf xfr_len
    bnz srdb_1

    movlw    event_transfer_complete
    call server_event_queue_write

    movlw    event_transfer_complete
    call client_event_queue_write
    goto srdb_9

srdb_1
    call slave_read_byte_entry_point

srdb_9
    return

```

---

```
;-----  
;State Transition Function  
; I2C bus SLAVE specific  
  
STF_S_wt_data  
    movfw    xfr_addr1  
    movwf    fsr  
    movfw    data_in  
    movwf    INDF  
  
    incf xfr_addr1  
    decfsz   xfr_len  
    goto stf_swtd_1  
  
    movlw    event_transfer_complete  
    call server_event_queue_write  
  
    movlw    event_transfer_complete  
    call client_event_queue_write  
    goto stf_swtd_2  
  
stf_swtd_1  
    call slave_read_byte_entry_point  
  
stf_swtd_2  
    return
```

```
;-----  
;State Monitor Function  
; I2C bus SLAVE specific  
  
SMF_S_write_data  
    return
```

---

## TBD

- (2) breaking down a function into small units that can be processed as background separate tasks
- (7) example of two simple state machines interacting. One state machine monitoring a complex process, then signalling the other state machine when that process is complete. Think of a clever example that cuts down one big complex state machine into two much smaller simpler state machines.
- (8) explain how work flow is an example of a state machine
- (9) explain how a state machine development tool should allow the user to interactively generate events and observe the consequences on the state diagram. Show the benefits in term of time saved and errors detected
- (10) explain why having complex functions that use variables to store state info outside of state machine are actually detrimental to state machine (can cause hidden problems and are much harder to maintain, document and debug)
- (11) show a call graph of all inputs (events) against all states
- (12) when is it better not to use state machines
- (13) show how continuous execution and a software state machine are fundamentally similar and explain how the state machine just uses groups of instructions between changes of state.