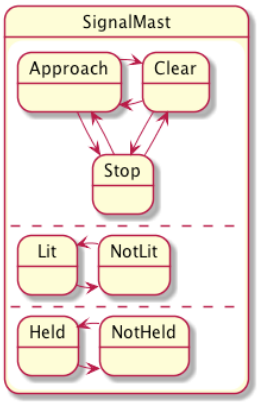
OpenLCB, JMRI and Signal Masts



Consider one complete signal on a railroad. It might have multiple lamps, multiple things that look like "heads", etc. Generally, a single mast by the side of the track contains one complete signal, though they can also be implemented in other ways. At any given moment, it's characterized by:

- its current aspect: Stop, Clear, Approach, etc[1] – this determines the appearance[2] that will be presented
- whether it's set to be Lit or NotLit – sometimes used for approach lighting; may not be present
- whether it's set to be Held or NotHeld – sometimes used for the logic controlling the aspect; may not be present
- and perhaps other information.

A model railroad might have multiple representations of one signal:

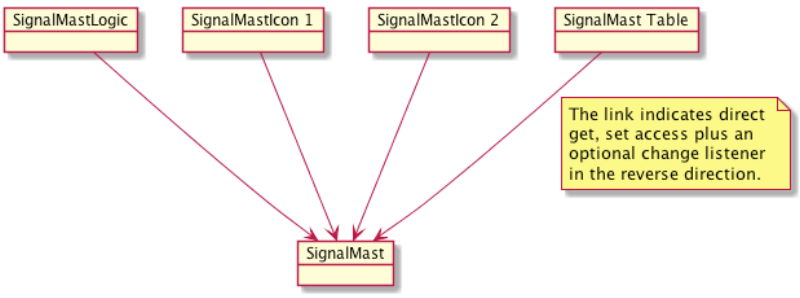
- A model mast with LEDs by the model trackwork
- A set of LEDs on the layout fascia nearby the track
- A drawing on the computer screen of a dispatcher workstation
- A drawing on the screen of a phone being used as a throttle
- A text entry in a computer program window, i.e. a table of signals
- State bits in the memory of a computer program that determines the proper current aspect as trains move and the dispatcher changes layout status.

To behave properly, all of these have to be kept consistent.

SignalMasts in JMRI

JMRI uses objects of "SignalMast" type[3] to represent a signal. A SignalMast object provides access to the current state via "get" methods, notification of changes via property listeners, and the ability to change the current state via "set" methods.

Other JMRI objects interact with the specific SignalMast object and its state to do various things. For example, a SignalMastIcon object is responsible for showing the state of the signal on a particular panel. To do this, the SignalMastIcon object listens to the SignalMast object for state changes, and displays the proper state when it changes.

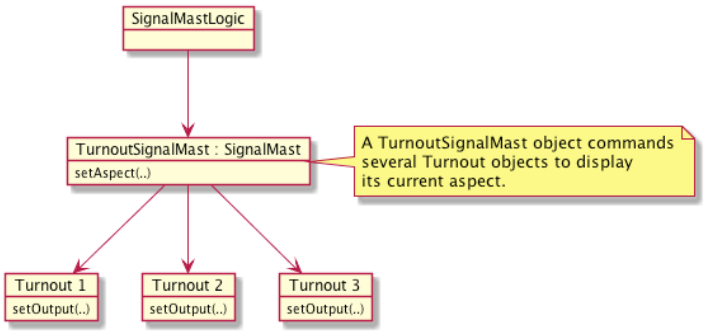


1 This note will use just those three aspects in examples without loss of generality.

2 Although the "aspect" vs "appearance" distinction and necessary mappings are important in signaling, they're not so important for our discussion here. Just assume that there are mechanisms that handle that, along with any necessary configuration.

3 See http://jmri.org/JavaDoc/doc/jmri/SignalMast.html for the interface specification.

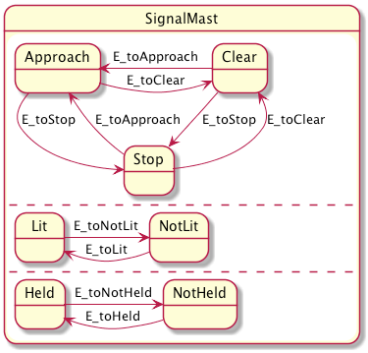
JMRI can control many different kinds hardware on the layout. It does this by having multiple SignalMast subclasses that do it in specific ways: By directly commanding "turnout" controllers that in turn operate lamps on the layout; by sending DCC signaling packets; by commanding "SignalHead" objects which in turn operate the one or more heads that make up the entire signal; etc.[4]



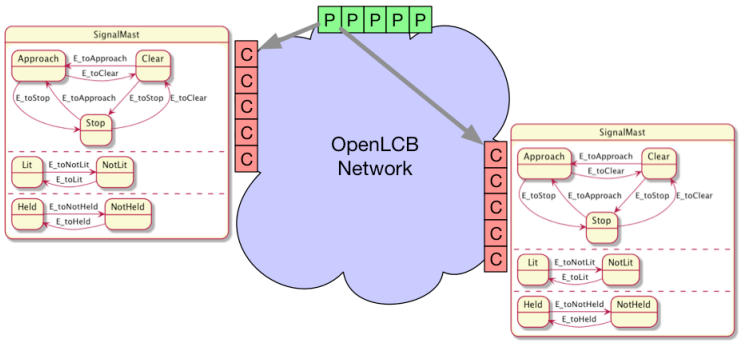
In general, JMRI does not listen to the layout for changes to the signal state.[5] Until recently, there generally wasn't a way to do that, and even when there were limited possibilities, the complete state of the signal wasn't present on the layout: The hardware didn't contain a copy of Held vs Not Held, Lit vs Not Lit or even some aspects. OpenLCB changed that.

OpenLCB

OpenLCB provides a reliable way to create fully coherent distributed state. It does this by using PCER messages to exchange events corresponding to state transitions[6], delivering them to every concrete implementation of the state machine in the various hardware and software nodes. It's up to each implementation to maintain its own state information properly when it receives those events. During design and configuration of the control system, specific numeric EventIDs are associated with each of those transitions and configured into all the various components of the system.



An OpenLCB network connects producers for an event to consumers for that event. To take part in the distributed state machine, each local state machine needs to have a consumer for every event.[7] One or more producers can them emit events when it's time to transition the entire distributed state machine. The OpenLCB network handles all the communication and coordination necessary to make this work properly.



4 If more than one type of hardware needs to be controlled, e.g. SignalHeads on the layout and Turnouts for a separate display on the fascia, JMRI provides a repeater capability that's not relevant here.

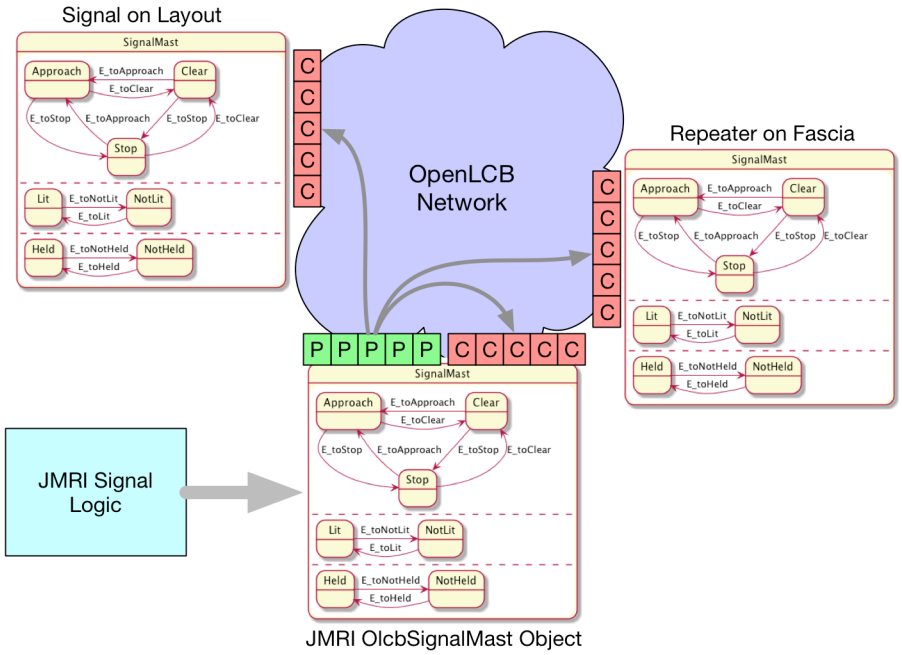
5 This is a statement about the state of the signal: current aspect, lit or not, etc. It's not a statement about the state of other things, such as train position, that some logic might use to command the current state of the signal to change.

6 Note that OpenLCB does not exchange state names or numbers; it exchanges state transitions. It is a mistake to consider them as states because in some cases there is no one-to-one mapping between events and states. Consider a "Lights Brighter" event, where the resulting state depends on how bright the lights are now. Our current discussion of signals, however, relies on the simpler case of every state being reachable when each event arrives.

7 The image shows five producers and five consumers for visibility, but seven of each are actually needed: two for Held/NotHeld, two for Lit/NotLit, and three for the aspects.

JMRI SignalMasts and OpenLCB

JMRI has an OlcbSignalMast type to implement SignalMast via OpenLCB. It maintains a copy of the common state machine, and attaches to the OpenLCB system connection within JMRI to communicate via the OpenLCB network.



Normal Operation

JMRI's control flow for signals starts with signal logic, which commands changes to aspects in a SignalHead object. The specific type of SignalHead knows how to work with its external connection(s) to control the actual layout. Displays on panels, on phones and web browsers, etc (not shown) simultaneously work with the SignalMast object to get the information they need.

In the case of an OpenLCB signal, the sequence is:

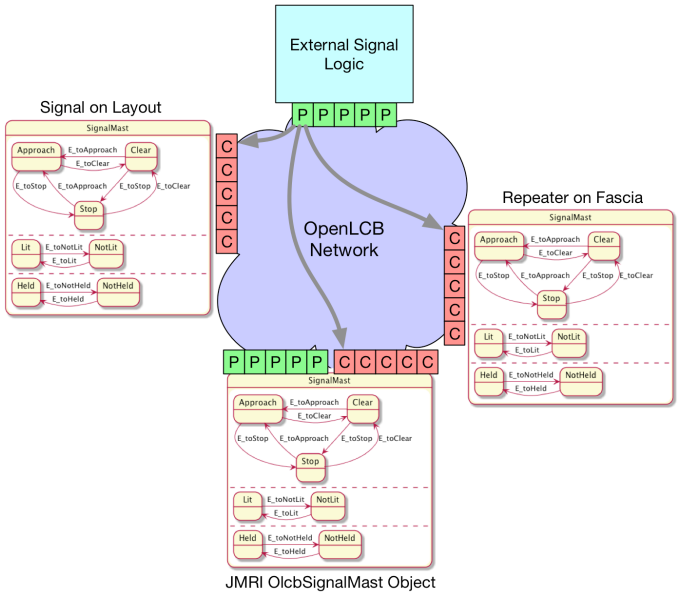
- Signal logic sets the current aspect in the OlcbSignalMast using a setAspect(..) call
- The OlcbSignalMast object produces the corresponding OpenLCB event via a PCER message
- The OpenLCB network propagates that message to all the relevant consumers, including the OlcbSignalMast itself
- Upon receipt of the EventID in the PCER message, each state machine implementation makes the appropriate transition, going to the same final state.
- Each implementation commands it's downstream connections as needed: Redisplay screens, change output voltages, etc.

Note that the state in the OlcbSignalMast only changes in response to receipt of the message at the consumer. This is a key point: *Only consumed events change the distributed state, including any specific elements of that state*.

This is different from other JMRI SignalMast implementations, which transition their local state in response to setAspect(..) in addition to sending commands to hardware. That's due to the limited

nature of other model railroad hardware connections. If JMRI could handle that other hardware as it does OpenLCB, it would.

This JMRI implementation for OpenLCB allows the signal logic to be anywhere on the network. Producers in some other component can produce the same events, which will propagate through the OpenLCB network in the same way, and drive the same behavior both within JMRI (including panel displays, etc) and on the layout. Although the OlcbSignalMast object creates producers for the events, without any logic in JMRI to drive local changes in the aspect, those will never send any messages. Control will entirely rest with the external signal logic.



Is it possible to combine JMRI and external logic? There are several reasons you might want to do this:

- Set held / not held from a CTC control panel, with the rest of the logic living controlling the aspect and lit / lit from somewhere else.[8]
- Allow a click on a control panel to command a particular aspect, e.g. for debugging or display running.
- Provide a button that sends the “go to lit” event so that you can see the current aspect on the layout even when using approach lighting (though not prototypical, sometimes people want to see the signals they put so much effort into)

Having multiple producers controlling the state, and hence the performance of the signals, works fine in the sense that OpenLCB and JMRI will happily respond when events are produced from any one of multiple sources. It’s up to the user and the external logic to arrange things so that the results make sense; neither JMRI’s OlcbSignalMast nor OpenLCB itself does anything about that.

For example, if you have a button that sets Lit so you can see signals, they’ll stay on until the next time any approach lighting logic decides to turn them off. Coordinating those two is up to you.

State Inquiry

OpenLCB provides a way, using the IdentifyConsumer message and its ConsumerIdentified reply, to learn about the status of a particular consumer, specifically the one that’s listening for a particular EventID on a particular node. When queried, each consumer of that event must[9] reply with one of:

- Currently valid – the internal state of the consumer & associated devices is known to be the same as if this was the last event consumed
- Currently invalid – the internal state of the consumer & associated devices is known to not be the same as if this was the last event consumed

8 See the Appendix for more on this.

9 This and later comments about what OpenLCB comments “must do” refer to the protocols as defined by the OpenLCB Standards, augmented with the corresponding Technical Notes, as of August 2015.

- Currently unknown – the consumer cannot determine whether either of the previous conditions is true

Note that these are not solely about the status of the event. An event has no status per-se: It's a transition, not a state. In our signal case, the "consumer & associated devices" language refers to the entire state machine. The "same as if this was the last event consumed" language, because each event transitions to a specific state, means the "Valid if in the state this event transitions to; invalid if in any other state".[10] For example, if events leading to "Lit", "Not Held" and "Approach" have most recently been consumed by a signal state machine implementation, so that it's in those states, a query of its consumers would find those three valid, and the rest invalid.

Perhaps an example will help. Consider just the Lit/NotLit two-way substate[11] of an OlcbSignalMast:

- At the start of our example, it is currently in the "Lit" state: Querying consumers would clearly find "to Lit" valid and "to Not Lit" invalid.
- Internal logic does a setLit(false), so the OlcbSignalMast produces a "to Not Lit" event on the OpenLCB network. At this point, the local state machine still in the "Lit" state, so querying consumers would find "to Lit" valid and "to Not Lit" invalid, same as the previous state.
- The "to Not Lit" event arrives from the network, is consumed, and the internal state becomes "Not Lit". Now, querying consumers would find "to Not Lit" valid and "to Lit" invalid.

Every signal implementation has consumers, so will take part in the above interactions.

A similar inquiry mechanism exists for producers, who must reply with one of:

- Currently valid - "the internal state of the producer & associated devices is known to be that which would cause them to produce the event"
- Currently invalid - "the internal state of the producer & associated devices is known to not be the same as that which would cause them to produce the event"
- Currently unknown - "the producer cannot determine whether either of the previous conditions is true"

Note that the language for producers is somewhat different: "which would have caused them to produce" reads as more hypothetical. The difference is because, as we saw in the discussion of the JMRI OlcbSignalMast, producing events doesn't directly change the state, only consuming them does. Never-the-less, that state matters to producers when replying to this queries.

Not every signal implementation has producers: Some just gather state for display. But some, like the JMRI OlcbSignalMast class or the dedicated logic implementation in the earlier figure, do have producers, so must reply to inquiries about the events they produce.

Consider again the example of just the Lit/NotLit two-way substate of an OlcbSignalMast when it is currently in the "Lit" state: Querying consumers would clearly find "to Lit" valid and "to Not Lit" invalid. A query to producers requires considering whether "the internal state of the producer &

10 It's permitted by the OpenLCB standard to reply "unknown" here, but that's not a good thing to do. In these examples, the right reply is known, not unknown, because the internal state cleanly indicates which should get "valid" and which "invalid".

11 For simplicity and compactness, with no loss of generality.

associated devices", i.e. the state machine, "is known to be that which would cause them to produce the event". But in general this can be considered in several different ways:

- This is about the state of whatever is driving the producers, so that upstream logic determines which producers reply "valid" now. If we assume that logic keeps the producers up to date, this is the same as saying that only the last event produced shows "valid" now.
- This is a query about the status of the state machine so the event associated with the current state should show as "valid".

In practice, these are both the same: The OlcbSignalMast will have a state machine that, after produced events are consumed, fully tracks the produced events, hence the commanded state. So either possibility can be implemented:

- Keep a record of the last event produced. The producer for that event will reply with "valid" and all others will reply with "invalid". If no event has ever been produced, i.e. the command logic has never setAspect(..) and no PCER messages have occurred on the network, all will reply with invalid.

  Note that this includes events produced elsewhere[12]. If the OlcbSignalMast sees event "to Lit" produced elsewhere by consuming a PCER message for it, the state is now "known to be that which would cause them to produce the event", so that's the one that gets the "valid" reply and the OlcbSignalMast starts replying to "to Not Lit" with invalid.

- When a producer inquiry comes in, check the state machine and reply appropriately. Optionally, add an "uninitialized" state that would allow both "to Lit" and "to Not Lit" to get "invalid" replies.

The second approach is perhaps preferred because the state machine is already tracking the PCER messages, and this allows some common implementation for the producer and consumer reply sequence. But either is fine, and will reach the same final states.

System Initialization

Traditionally, model railroad control systems haven't made a lot of allowance for how the system is brought up, or whether items can properly be added to a running system. They take the approach of "just keep cycling things until everything reaches the right state", but particularly when modeling complex prototypical signaling systems this can feel like a random walk.

Here, we consider several cases of a JMRI-OpenLCB signal implementation starting up, showing how they work with a full implementation as described above, and in some cases how they'll function is some of the inquiry features are missing.

For this section, a signal inquires for producers of all its EventIDs when it starts up and acts on the replies by setting its state machine from replies marked "valid"[13]. This is done to get as much information as possible about the state intended by the logic driving the control system. We discuss alternatives below.

12 One consequence of this is that a signal logic node needs to have a consumer for every producer, even if it's only implementing signal logic and not signal display. This is not true in general; there are P/C applications which don't have this requirement because of how their state machines are structured and interact with the layout. But it's true of a lot of them.

13 See the final section for more detail.

JMRI Logic Starts Last

In this case, there are multiple signal implementations that completely start up before JMRI starts. JMRI contains an OlcbSignalMast and the associated logic to drive it.

- Each signal starts, inquires for producers, and gets no replies because the logic isn't running yet. Because the layout isn't being controlled yet, the internal state of each signal is whatever commodious default has been selected, perhaps "Stop, Lit, Not Held". Alternately, an "uninitialized" state can be used. The signal performs its function based on the internal state: Showing lamps, drawing on a screen, etc.
- To start control of the layout, the JMRI program is started which creates a OlcbSignalMast. It inquires for producers, but hears no replies. The signal logic starts and gathers necessary input from the layout. At some point, it completes its first calculation and does setAspect(..) on the OlcbSignalMast. This produces the necessary event(s), which in turn sets the state in each of the signals. The system is now behaving coherently.

JMRI Starts Before Logic

In this case, there's one or more JMRI instances that contain OlcbSignalMast(s), but these are not providing the logic. Some signals come up before the OlcbSignalMast, some after. The logic comes up last.

- Some signals start, inquire for producers, and get no replies because the logic isn't running yet. Because the layout isn't being controlled yet, the internal state of each signal is whatever commodious default has been selected, perhaps "Stop, Lit, Not Held". Alternately, an "uninitialized" state can be used. The signal performs its function based on the internal state: Showing lamps, drawing on a screen, etc.
- JMRI with its OlcbSignalMast starts and inquires for producers. It hears no reply and sets a default state.
- Other signals start and inquire for producers. The JMRI OlcbSignalMast replies based on its internal state[14], which in turn is used to initialize these newly-started signals.
- The signal logic starts and gathers necessary input from the layout. At some point, it completes its first calculation and produces the necessary event(s), which in turn sets the state in each of the signals. The system is now behaving coherently.

In both of these examples, there's a time before the logic is running where the state of the system is unconstrained and different signals may have different results. OpenLCB provides mechanisms that can be used to force the system to a single, coherent state[15] earlier but how much trouble is that worth?

14 What, exactly it replies depends on details of the default state, use of uninitialized, etc, but that doesn't effect the results here once the layout is under control.

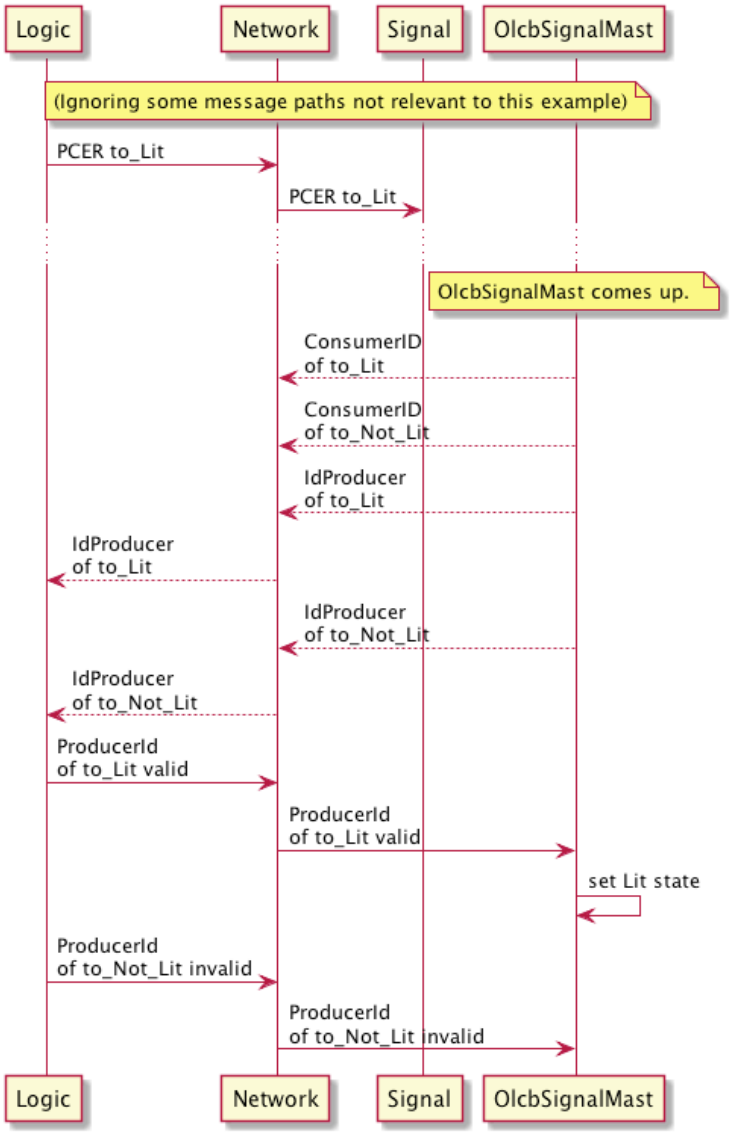
15 Basically, inquire of consumer states, and then use application logic to resolve any conflicts via e.g. "prefer Lit".

Normal operation has some logic element(s) that drive the signals to specific states so that they can help operate the railroad; if the logic[16] isn't operating, the signals are not operating.

Adding A Signal After Logic Is Running

Consider that one of the above sequences has run and the layout is up and being controlled. Then, add a signal to that running layout. Perhaps it's newly installed, perhaps it rebooted after a power hit, perhaps it's on a cell phone with limited connectivity. Many older systems would just say to wait for the next update, operating in a default state until that comes along.[17] The OpenLCB signal sequence is:

- The new signal starts and inquires for producers for its EventIDs.
- The logic replies, showing exactly one EventID as "valid" for each of the three substates, e.g. Lit, Not Held, Clear.
- The new signal sets its state machine from there, and operates based on that data.



16 Or a stand-in or manual mode or maintenance mode or some form of logic.

17 In some cases, the control logic provides a "run an update now" mechanism to force this to happen promptly.

Adding A Second Set Of Logic

As discussed above, it's possible for more than one element to control the signal state.

It's beyond the scope of the signals and OpenLCB to handle the consistency of more than one set of control logic: That's done in the control logic or a higher level. If it's useful, logic that's aware that it may not be unique may optionally query for producer replies, hence states, to see what it's peer logic elements have already decided. Otherwise, it just comes up and starts producing events, which communicates with the other logic element(s) as previously described.

If the second set of logic is being added to the system as a replacement, i.e. the first set was removed from the running railroad and a new one is added, the new one may not hear replies to its optional producer inquiries. In that case, it can just go ahead and do it's own calculation. Alternately, it can do consumer inquiries to try to figure out what the current state is; this works if one or more of the signal implementations is replying with valid/invalid to the inquiries. But it probably much easier to just gather input and drive the current correct state.

Alternate Approaches

It's certainly possible to define alternate approaches in addition to the above scenarios. First, we consider components with less capability:

- Simpler signals that don't inquire about producers or pay attention to the results of that inquiry process[18]: These will just wait for the first logic update to come along.
- Simpler signals that always reply "unknown" to an inquiry[19] about their consumers: Except in the most complex cases, signal logic doesn't inquire about consumers, so this has no effect. It's still a bad practice, though, because who knows what'll be done in the future?
- Logic that doesn't bother with the producer inquiry[20] process: These may or may not coordinate well with other logic implementations, but will work fine by themselves.
- Logic that replies "unknown" to producer enquiries: Signals added to this logic after it starts up will have to wait for the next update, but will otherwise operate fine.

It's possible to make up for the weaker implementations of some components by making others smarter/stronger:

- When a signal node initialized, even if it doesn't do anything else, it is required by the OpenLCB standard to send a ConsumerIdentified message for the events it consumes. A logic unit could consider this a request for a state update, and produce the event for its current output state, taking the place of waiting for the next routine update. The question is when to do this. It's burdensome to track which node(s) are already being driven so that the appearance of a new node can be detected. If some node inquires for one of the relevant events with IdentifyConsumer, there will replies from each of the signal nodes; that could be a lot of updates if one was driven by each reply.

18 This is a global process, so a node hears the results of other nodes' inquiries.

19 Although not recommended, this is permitted.

20 OpenLCB implementations must reply to producer and consumer inquiries with Producer/ConsumerIdentified messages and nodes must emit Producer/ConsumerIdentified messages at startup so that the routing mechanism works, but there's no general requirement to pay attention to these.

- Similarly, a newly-connected logic element could use received IdentifyConsumer messages to learn about the existing signals and their state[21]. Depending on past history, the results will be either a clean and consistent representation of the distributed state, or inconsistent replies could indicate that no logic has ever driven the state to consistency. It would take some care to handle all the cases well, but perhaps even handling a few common ones might be progress.

Although those alternatives could serve as workarounds, it seems desirable to have all signals and signal logic provide a consistent, complete implementation.

Recommendations

The following are the most basic recommendations for consistent implementations of OpenLCB aspect-based signals, most important first:

- Signal and Logic elements for a particular individual signal should have a three-part state machine, with a unique EventID associated with a transition to each state:
    - A consistent set of aspects
    - Lit / Not Lit
    - Held / Not Held
- A logic element desiring a change to the state should produce the relevant EventID and wait for it to return from the network; it should not change the state locally. Signals and logic element (and combined versions such as a JMRI OlcbSignalMast) should change the local state when an associated EventID is consumed.
- Signal element consumers and logic element producers & consumers should set the valid/invalid bits in Producer/ConsumerIdentified replies based on the current state of their state machine.
- A newly-up signal element should send IdentifyProducer requests for all its EventIDs and set the local state from any ProducerIdentified replies that are marked “valid”.

21 This might be a valuable way for the logic to get timely information about other signals, for example the signal that this one protects.

Appendices

A: Why doesn't this discuss having an "authoritative" source for information?

At least in signaling (but not always in other cases) there really should be just one set of logic controlling the normal case. If there were two sets attempting to fight each other for control, that would mean they weren't doing the same thing: how can that be a model of a prototype railroad? "Let's have Canyon Dispatch trying to drive this stretch of track as speed signaling, and Central Dispatch as route signaling; it'll be fun to see how the engineers react!"

People want the OpenLCB (part of) the system to know about "authoritative" as a way of saying "who's in control". That's already known as "who produced the last relevant event?" and already available as "Who's replying with a valid producer state for this event?" Adding a separate "authoritative" bit somewhere is just a recipe for inconsistency.

What would it mean if box A was "authoritative" and Box B was the last to produce a relevant event?[22] In OpenLCB, being the last to produce a relevant event means that B really is the boss, and that the system has been misconfigured: Either B was misconfigured and shouldn't have produced, or "authoritative" was set wrong, or it isn't even a valid concept for this system.

"Authoritative" is a distributed/emergent property of the overall system design. The time to get it right when you're figuring out what controls what. It's not a bit in some operational memory.

B: On "unknown" reports when gathering input for signal logic[23]

When it first comes up, the component who's calculating the signal logic has to gather its initial set of inputs before it can produce the new aspect(s). This can include multiple kinds of information:

- Block occupancy (Sensor) status
- Track configuration (Turnout) status
- Adjacent signal aspect
- Overall controls: Whether this signal is held by CTC, directional-of-travel information, etc

If the logic is within a signal JMRI instance, all of this information is available via get(..) operations and/or triggering listeners, so long as the relevant layout objects (Sensors, Turnouts) have been able to

22 The "authoritative bit" idea has come up in multiple fora, but so far nobody has ever provided an answer to this question that shows they've worked through the details. That also says something about how hard it is to make a useful, implementable "authoritative bit".

23 This appendix is about the consequences of "unknown" vs "valid"/"invalid" in ProducerIdentified reports. For discussion of a possible "unknown state", see appendix D.

interact with the hardware. Because the signals depend on each other (i.e. for protecting each other), the initial round of computation is iterative, but it rapidly converges[24].

How does this work if the signal logic is being computed by some independent element that's only connected through the OpenLCB?

- The simplest possibility is for the logic element to wait to hear from its inputs, not producing (hence leaving at default) the signal aspect until that's been done. But that's hardly desirable, as it may require actually moving trains before the signals start to work.
- It's also possible to phase the bring-up of the layout, bringing the various inputs up after the logic is ready to listen to their initial status. This can be very complicated and inconvenient, and still requires that:
    - All the layout device controllers be well behaved in the sense of advertising their current status on their "ProducerIdentified" initial messages.[25]
    - Requires very careful consideration of how turnout controls operate[26] at startup.

    This is in not in way a recipe for "easy user installation, configuration and operation".

The best solution is to have the signal logic go out and get the current state of the layout. OpenLCB's "IdentifyProducer" mechanism, along with the associated event valid/invalid information in the reply, is the way to do this[27]. That in turn requires meaningful reply contents: A blanket "we always reply with 'unknown'" policy is deeply anti-social for the control system. "Unknown" really means "You have to wait around until we next produce a relevant event, sorry". That wait will work, but it can be very undesirable for the user.

To put it another way: Although there are valid cases where a control element must reply with "unknown" in a ProducerIdentified reply[28], none of those are relevant for signal system inputs. The devices that are used for signal systems, such as block occupancy detectors, turnout controllers, etc should implement valid/invalid correctly.

---

24 There can be pathological cases, but they're rare and fixed by reconfiguring the signals.

25 They could also produce their relevant events at startup, but that's not recommended.

26 An OpenLCB-connected turnout has a controller that commands the turnouts position by producing events. At system startup, the turnout might be remembering its position, or it might be waiting for the controller; the controller in turn might be commanding a remembered position, or itself looking for input before calculating and sending commands. There are lots of possibilities, and it's hard to make them all part of one, always-reliable start-up sequence.

27 An OpenLCB-connected "sensor" produces events, so by querying the producer you get the status. Turnouts are more complicated, as the OpenLCB-connected "turnout" may or may not produce events (depending on its ability to provide feedback), and the original command to move it is generally produced somewhere else. See previous note "Features in Event-Based Turnout and Ladder Control". But somewhere, there's at least one producer of the relevant event responsible for replying to the inquiry.

28 For example, a momentary push button that sends an "All Lights Off" event, forcing a transition in multiple light-specific state machines.

C. Why should individual signals know about lit/not-lit and held/not-held?

Many control systems have concepts like “lit” and “held” that live only in the control logic. For example, if you’re controlling a signal via three digital outputs for “red”, “yellow” and “green” LEDs, then you can use logic like:

- If not-lit or aspect is “dark”, turn off all lamps.
- If held or aspect is “stop”, turn on red and turn others off
- …

The problem is that some other device, e.g. some other computer, watching layout communications can’t tell exactly what’s happening. “OK, I see all lamps off, but is that ‘dark’ or ‘stop and unlit’ or ‘clear and unlit’?” It’s no longer possible to e.g. put a signal repeater on a remote screen based on that information. Even using aspect-related commands, instead of appearance related ones, doesn’t fix the problem[29]. The right thing to do is to have the full set of state information available to anybody on the layout.

OpenLCB makes any properly-implemented state available to anybody. What does properly implemented mean?

- If you want to use the state information, you have to track the state machine, consume events that drive transitions, and otherwise completely take part.

  For example, if the controller for a physical head wants to use Lit/NotLit to decide what lamps to light, it needs to track that state. On the other hand, Held/NotHeld doesn’t generally change what physical lamps are lit, so that doesn’t need to be tracked.

  A software controller, which might want to separately display Lit/NotLit and Held/NotHeld on the screen, would have to track them both.

- If you want to control, i.e. change, a part of the state, you have to produce the relevant events and track the state machine, including consuming events, enough to be able to report its current state.

  For example, there might be several components that want to set the signal as Lit: The core logic controller, plus a “Light the signals so they look good” pushbutton, plus a clickable icon that can be used to show an engineer what’s further ahead. Each of them has to track the Lit/NotLit state completely to ensure that everything works as intended.

What does this mean for component designers?

29 One could suggest quadrupling the number of aspects with “stop-lit-held”, “stop-unlit-held”, “stop-lit-unheld”, “stop-unlit-unheld”, but that’s just a tricky coding of the separate state discussed here: Better to make it a separate state.

- The designer of a hardware board just has to select whether Lit/UnLit and Held/UnHeld are needed for the boards function, and implement those.

  Note that this is fully future-proof: A board doesn't have to implement Held/UnHeld if it doesn't know about it. Similarly, if some other state comes along in the future, say Painted/UnPainted, since the hardware board by definition doesn't have behavior that depends on that new state, it's perfectly fine that it doesn't know anything about it. The system will still work fine (though that particular board won't be using the new information).

- Software element designers face a similar decision, but because the resource balance is different, they may choose to approach it from the other end: Because JMRI wants to be able to drive all those states, full tracking of all of them will be built into JMRI's support. Then it's up to the user to configure whether some JMRI instance actually controls those states or not.

  For example, an OpenLCB-based ABS signal system may never use the Held/NotHeld state; a system without approach lighting may never use Lit/NotLit. That's perfectly fine; JMRI will track an unchanging state, and the cost of doing that is completely reasonable. Should the user ever later desired more function, it's there to be used.

Each component only needs to implement the parts that it needs to do it's current or future job: if it's producing output from the states, it has to track the transitions and states it cares about. If it's driving the states, it has to provide information on the transitions and states it wants to be able to drive. So long as they fully implement all of that, it'll all work.

Example of minimal support:

- Hardware signal head doesn't know about Held. Layout isn't using Held/UnHeld. JMRI instance X is providing ABS signaling, understanding that a Held state drives signals to Stop, but defaulting to UnHeld if it doesn't see any OpenLCB traffic to transition the state.

  This works great.

- Add a second JMRI instance Y. It can see the full signal state from JMRI X so that it can display it on its screen.

- Now, it's time to allow clicking on a signal on JMRI Y's screen to be used as a simple dispatcher function: "Hold the next train here". The click sets that signal to Held, and a second click restores it to NotHeld. JMRI Y's implementation[30] has to track the Held/NotHeld state and produce events to change it.

  Since JMRI X is already tracking the Held/UnHeld distributed state, it gets that information automatically; no configuration needed. JMRI X uses it in its aspect computation to display the right thing. The hardware head doesn't have to listen to Held/NotHeld, which is good because it's not built to do that; it just gets new aspect information from the controller in JMRI X.

30 Which would probably be through a SignalMastIcon and OlcbSignalHead, but that's not the key point here.

So long as every component is properly and completely implementing the relevant states, nothing needs to be reconfigured to add or remove new controls. The information all gets where it needs to go.

Certainly things can go wrong if the layout is operated improperly. For example, if JMRI Y has set the signal to Held and then goes offline, that signal is going to stay at Held. That's a problem, but it's not an OpenLCB or even a JMRI problem[31], it's a problem with mismatch between layout design expectations ("JMRI Y will control Held") and how the layout is operated ("JMRI Y went away, but it's still controlling Held"). The fix is at a higher level, in how JMRI and/or operations are set up. ("We have to decide what to do when John takes his dispatching laptop home")

D. Tediously working through start-up cases

There seems to be a general concern that something about start-up doesn't work right, particularly around default states. This section presents diagrams for several possible cases when two JMRI instances start up, in the artificial case that they have different ideas about proper defaults. Background and context points:

- This only considers the Held/NotHeld state for simplicity; it can be generalized to other state machines easily, but the diagrams get busy.
- These examples are specific to the signaling case, where there's a direct mapping between states and events. Other state-event mappings may behave differently.
- There many be other objects listening to this, e.g. signals on the layout, but they don't take part in these particular sequences.
- In all cases, OlcbSignalMast 1 and OlcbSignalMast 2 are in separate JMRI instances, communicating only through a OpenLCB network. OlcbSignalMast 1 always starts first and gets through it's initialization before OlcbSignalMast 2 starts initialization.[32]

There are three cases to consider:

- OlcbSignalMast 2 has control logic which really cares about its state, and OlcbSignalMast 1 does not so it willing to accept 2's state.
- OlcbSignalMast 1 has control logic which really cares about its state, and OlcbSignalMast 2 does not so is willing to accept 1's state.
- Neither cares about their states, so both are willing to accept the others.

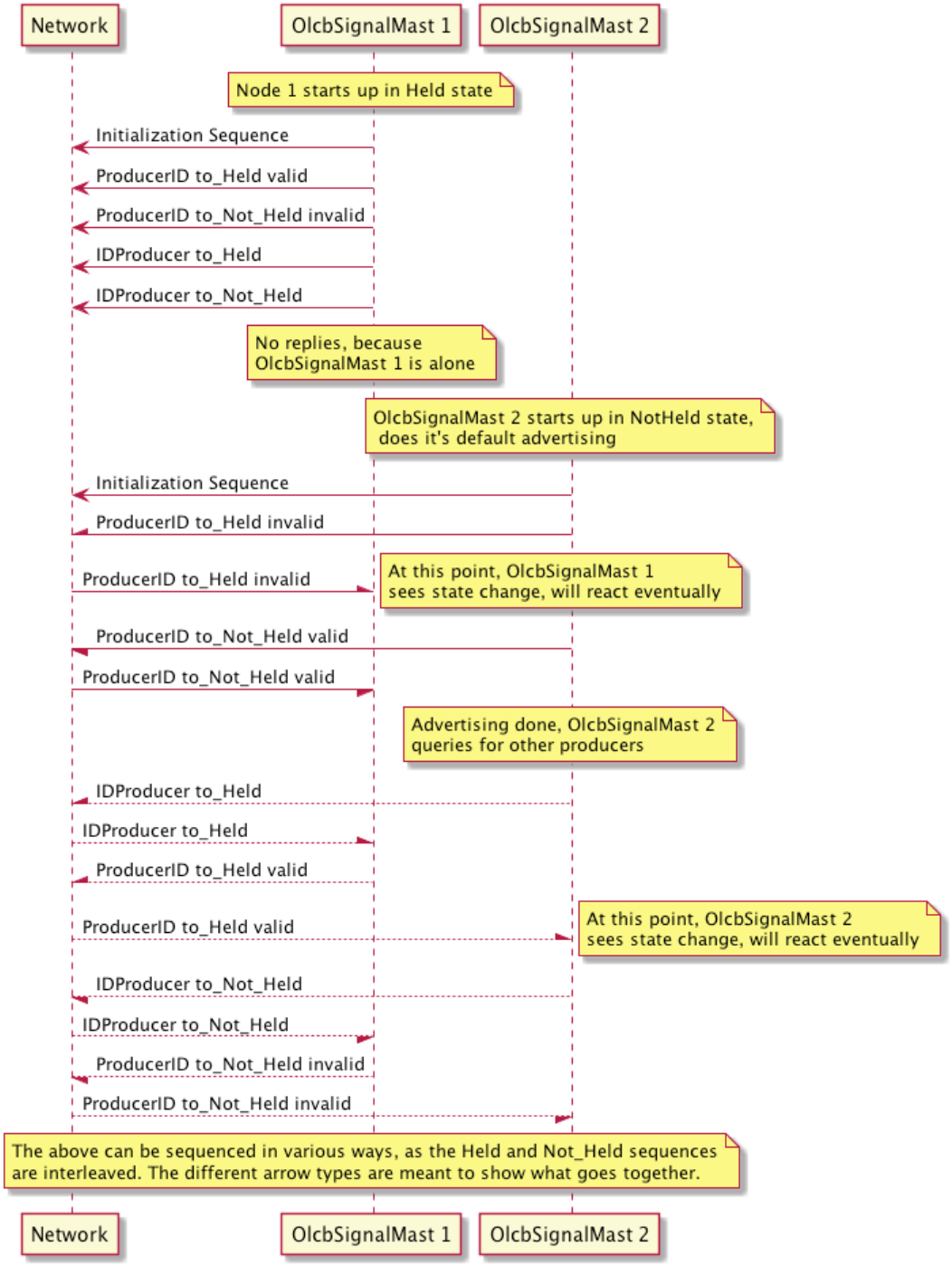
The fourth case, where both care and want to assert their state, isn't considered here because it's not at all clear what the outcome should be, other than declaring this a configuration error. If there are two

31 Unless it went away because the JMRI instance crashed.

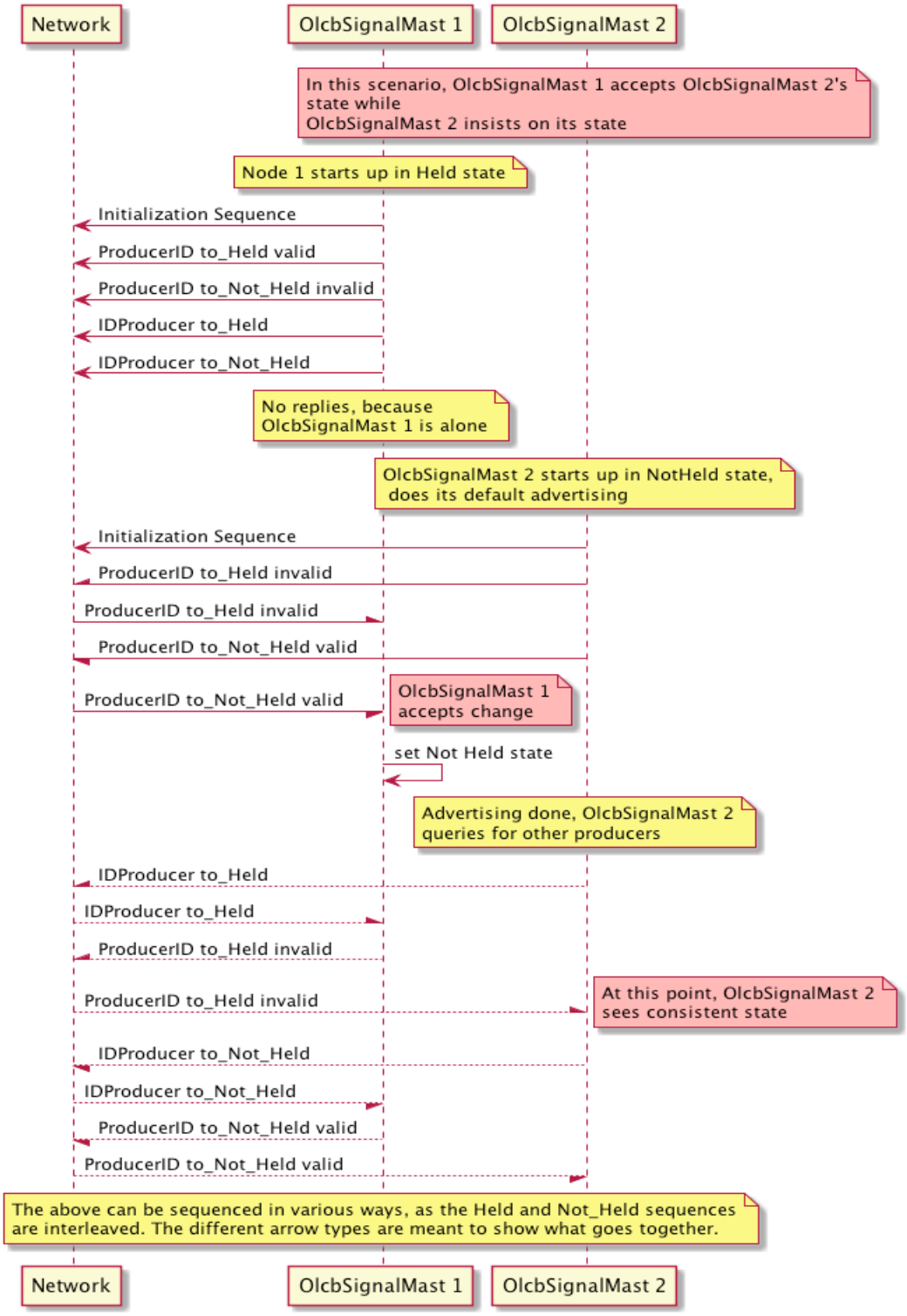
32 Overlapping initialization can certainly be considered, but it doesn't add any value to the discussion here because all the interactions are serialized in the same way; it just gets more complicated to refer to "1" and "2".

things insisting on transitioning the same state machine to different states, what happens depends a lot on which one gives up first, etc.

First, we present the sequence when neither reacts to the messages from the other. This is not a correct sequence; the messages are supposed to have consequences on the internal state. This is just to show the sequence of events when neither reacts, so we can compare to it later.

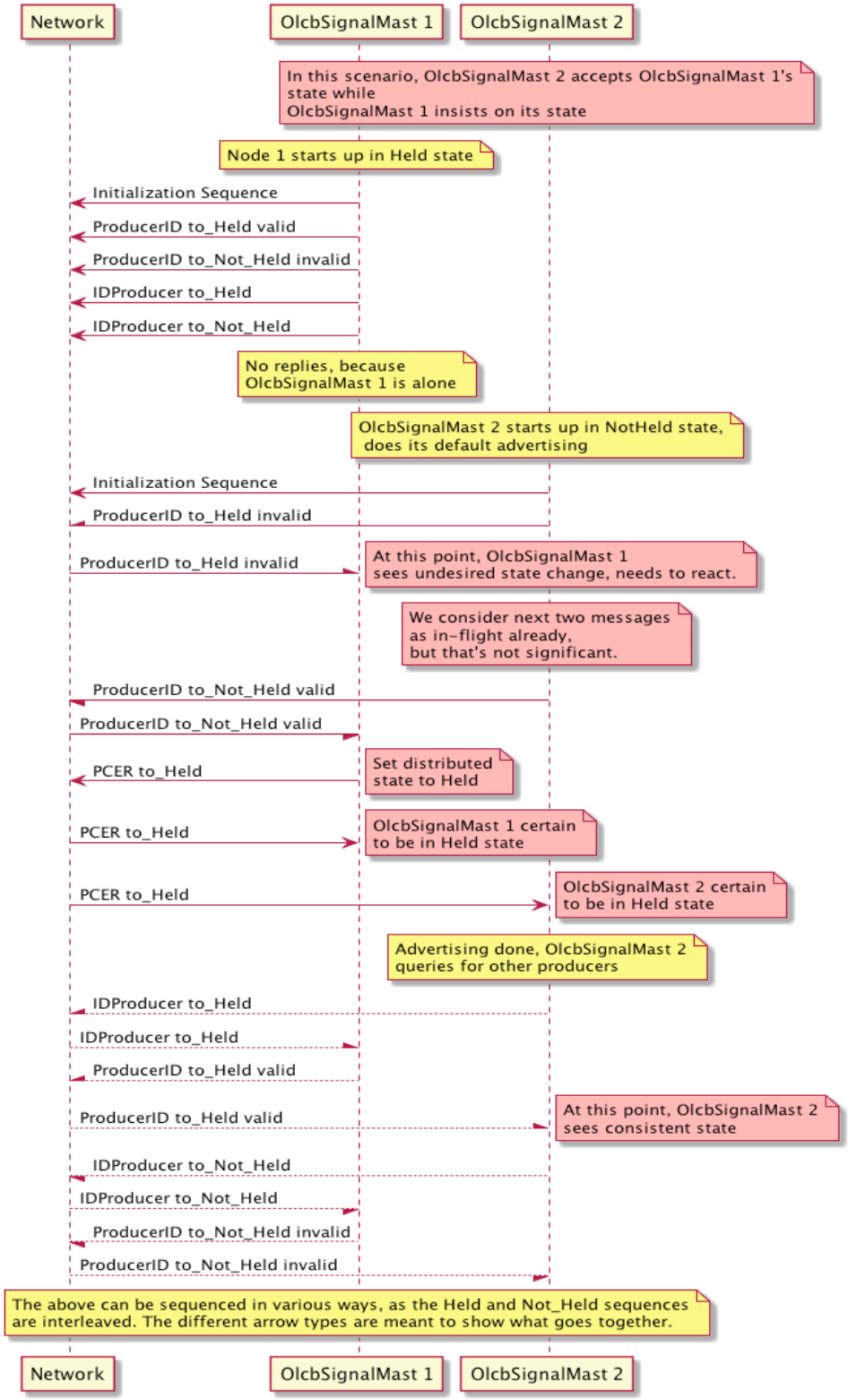


The first case is where OlcbSignalMast 1 is willing to accept the word of OlcbSignalMast 2 on what the state should be. When OlcbSignalMast 1 sees the initial state report, it changes its internal state:



Note that this is the same sequence for the “neither cares” case: If neither cares, the update from the new node is accepted.

The remaining case is where OlcbSignalMast 1 wants to insist on its state, and OlcbSignalMast 2 is willing to accept that. It's possible to sequence the exchange in several ways, but they always end up with the same state:

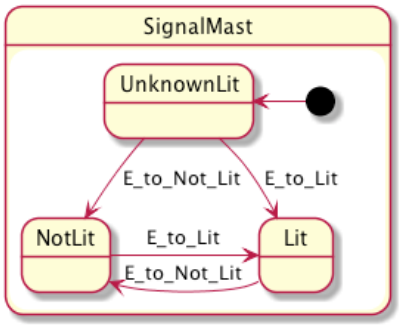


E: Why an “unknown” state isn’t a thing

“Why doesn’t OlcbSignalMast have an unknown state?” is a common question. The answer is discussed in this section. First we clarify some terminology, and then apply it to the state structure of OlcbSignalMast.

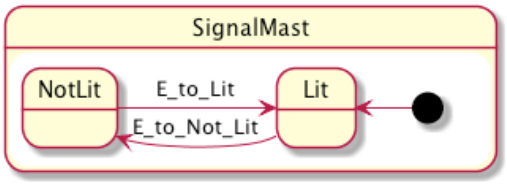
A distributed state machine’s own status can be “consistent” or “inconsistent”; ‘inconsistent’ only happens at the start, and it’s a transient condition so it’s sometimes called “initializing”. Once through initializing to consistent operation, the distributed state machine then conveys specific states like “the state called Held”.

There certainly can be a “state called unknown”. You can have a sub-state machine that starts in the “state called unknown” and then goes to either “the state called Held” or “the state called Not Held”.



But there is no such thing as unknown state, in the sense that the machine is not in a specific, named state. Should a consistent distributed state machine ever find itself not knowing its specific state, that’s a Really Bad Thing that indicates Something Is Badly Broken.

OlcbSignalMast has three independent sub-state machines: Aspect, Held and Lit. Let’s use Lit/NotLit as an example. Clearly, it needs the two meaningful states: Lit and Unlit. Because signals are often permanently lit, it seems reasonable to start in the Lit state. That gives a state machine like below left. Or, we could add an explicit state named “UnknownLit”[33] and start in that state, as shown above right.



In order to decide which of these to implement, we should consider a number of questions about the behavior of both approaches. Reasonable people will differ as to their relative importance, but at least we can settle which are pro and which are con.

1. Does a signal know how to display all the states?

   An icon on the screen or a line in a table can certainly show all three[34], but what do the lamps on the layout do? They have to be either on (lit) or off (not lit); they can’t be in a superposition of the two[35]. In other words, the signal has to behave as if its state machine had only two states.

2. Is there value to having the ability to display on the screen? Maybe, in that it indicates that this initial state is still present, and nothing has commanded another state. But that means that no node can e.g. store a state for display after power-up, as that would cause a negotiation away

33 It’s called “UnknownLit” instead of just “Unknown” because it’s just one components of the three sub-machines. If we generalize, we’ll need to distinguish from UnknownHeld and UnknownAspect too.

34 Though an icon space is going to get a bit cluttered if it has to represent two unknown states for Lit/NotLit/UnknownLit and Held/NotHeld/UnknownHeld.

35 Well, quantum mechanically they can be, but that would be a different document.

from UnknownLit. Nor that one can stay powered up and initialized on the layout. If any state information is available from when the layout was previously operating, it'll drive away from "UnknownLit" and you'll never see it once the OpenLCB state machines are consistently initialized.

"UnknownLit" is a state of the system that, by itself, seems to be used to indicate inconsistency; OpenLCB drives that out of the distributed state when it initializes, so you can't rely on "UnknownLit" to display that on the screen.

3. Maybe we should have events that can take the state from Lit to UnknownLit and/or NotLit to UnknownLit. Then the control system can command the hardware to remember that the state is that it doesn't know what the state is.

   Can one even say that with a straight face?

4. Is there anything else?

Fundamentally, people seem to want "unknown" because

a) they don't trust the components to be able to reliably achieve consistency via the OpenLCB protocols: "I worry that we'll end up with signals showing wrong aspects until we force a reset"

or

b) They want to have some control over the state of the system *before* the control elements of the system come up: "If the computer is off, I want the signals to show (insert favorite result here)".

Adding additional complexity is not the right answer for (a): If the protocols are incorrect or the components don't implement them properly, that has to be tracked down and fixed. The additional complexity of the extra states just allows more surface for problems to manifest[36], not less.

The right solution to (b), the desire for proper defaults, is to have proper defaults. If there really is a need to have some specific behavior before some part of the network comes up, make sure there's a component that does come up to do the job: Have some signal element come up and produce the desired state you want. But note that OpenLCB works very hard to achieve consistency: If you want your on-layout signals to show one thing, your repeaters to show another, and your screen icons a third, you don't get to ask OpenLCB to allow inconsistent operation sometimes and ensure consistency at other times. That's a different set of control system requirements.

Bottom line: Adding "UnknownLit" doesn't add enough to be worth it, so it's not present in the (prototype) JMRI OlcbSignalMast. But remembert that JMRI is an collaborative development project, so if somebody wants to do the work to add it and make it work right, more power to them.

36 Remember, in this case the reason for adding the 3rd state is because synchronization isn't reliable; synchronizing three is no easier than two.