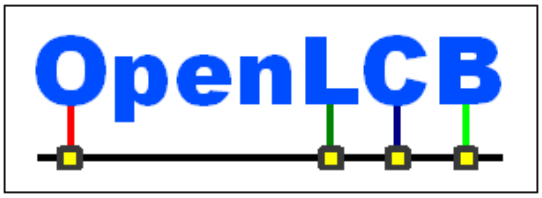

OpenLCB Working Note	
Time Broadcast Protocol	
Mar 29, 2013	Preliminary

1 Introduction

A Working Note is an intermediate step in the documentation process. It gathers together the content from various informal development documents, discussions, etc into a single place. One or more Working Notes form the basis for the next step, which is writing one or more Standard/TechNote pairs.

This Working Note is intended to become the Standard and Technical Note on the OpenLCB Time Broadcast Protocol.

Model railroad layouts sometime have their own time displayed via visible clocks that people refer to when operating the layout. These might display a different time of day from the current real clock time, so that an evening train can be run during the day. They might run at a different rate, so that the shorter distances of the model are covered by a train in the right number of "fast minutes".

Layouts that have multiple clock faces might want to synchronize them by running them from a single time generator. OpenLCB should provide a way to do that over existing OpenLCB infrastructure.

Producer/Consumer events, one per second or minute, could certainly meet this need. Since event IDs are arbitrary, in the absence of any other support, the clocks would have to have 24*60 or 24*60*60 separate event IDs configured between clock time producers and consumers. This is untenable. Hence, the primary reason for this Standard is to create well-defined ranges of event IDs that can be used to represent time for driving clock displays.

A secondary purpose is to allow producers on the layout to control the clock operation by producing certain events.

A tertiary purpose is to allow other consumers on the layout to be controlled via the time signal by interpreting the events or event ranges specified here. Not all timing-related purposes will be met by the contents of this Standard, and other ways of handling timing formation will be needed for those.

1.1 Served use cases

Fast clocks distributed around the layout all show the fast time. They start and stop together. Start and stop controls are distributed around the layout. The rate can be varied as needed. The display includes a modeled date.

In addition to fast time, a "correct wall-clock time" can also be distributed around the layout room. Further, a "duration so far" time is distributed around the layout. It starts and stops independently of the fast clock.

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

A node connected to model building lights turns them on and off at specific fast-clock times. A control for lights or ventilation can turn it on and off at specific wall-clock times.

Time acts in multiple ways:

- Action at a specific time: “The bell rings at noon”
- Condition during an interval: “The porch light should be on from 5PM to midnight”
- As a state: “The clock shows 10:15AM”

Model building lights and layout room lights turn on and off in a fast cycle to simulate a day every 10 minutes.

The model railroad can run through a scenario: Start at 17:00 (5PM) and run to 20:00 (8PM), then go back to 17:00 and repeat.

A town contains 100 separate lights, each connected to an output pin on OpenLCB nodes. They turn on and off on a detailed cycle.

Simple nodes should be able to respond to timed events without any additional cost.

Modular layout sections developed and configured in separate locations can be brought together and operated without prior management/allocation of magic numbers, and with only minimal (at most one node and one operation) reconfiguration.

1.2 Unserved use cases

The model building lights turn on and off only a few seconds apart - this proposal is for minute-granularity timing. Faster timing can be controlled from the node using a script that gets its initial trigger from the fast clock signal, or some other node can produce closely-timed events that are consumed to drive the lights.

Rapidly moving from one time to another, e.g from 20:00 back to 17:00 in the use case above, can be done in several ways:

- By rapidly stepping through the intervening minutes. This will ensure that e.g. a light that turns on at 14:00 and off at 18:00 will properly be on when the sequence restarts at 17:00. But it will also cause the church bell that's scheduled to strike at 08:00 and 12:00 to ring between “20:00” and “17:00”.
- By going directly to 17:00. This will skip ringing the church bells in the point above, but might not get the state at 17:00 correct. The state can be addressed by optionally producing the necessary events as part of going directly.

Layout lights execute a pattern that's different when the modeled day is Monday vs Saturday – Day of week, or calendar date in general, does not result in different events being produced. This use case should be solved the regular producer-consumer techniques and a smarter scheduler to produce the appropriate events.

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

2 Specified Sections

This is the usual sections for a Standard, to accumulate the Standard and Technical Note content in its eventual order.

2.1 Introduction

Note that this section of the Standard is informative, not normative.

A layout control bus can do a number of useful things with fast-time information:

- Connect a number of clock displays to keep them synchronized.
- Provide time information to plug in devices, e.g. throttles.
- Provide cueing for time-based occurrences, such as lights turning on and off at specific modeled times.

Generally, existing fast clock systems have one unit that ~~generates~~ produces time information, and one or more units that consume it. Existing fast clocks typically only report minutes, not seconds or finer time divisions. Some existing fast clock systems track a day/date, in addition to time.

Fast clocks run at various rates, and can be controlled by the user either at the ~~generator~~producer node or from other locations. Some fast clock systems broadcast run/stop and rate information, which can also be useful when interpolating within a fast-minute.

OpenLCB broadcasts time information by producing Event IDs. Specific Event IDs correspond to specific times with the day, for example “08:10:00~~13~~”, so that consumers can be taught to react to time-of-day. The date is handled separately, for those installations that choose to use it.

Simple nodes can then use specific EventIDs to trigger their actions at specific times. For example, lights in buildings in a model town can be sequenced to come on at various times by configuring consumers in a node to react to time events by changing output lines.

Since remote control of the fast clock is desired, a set-protocol using produced and consumed events is defined. (This makes it possible for e.g. throttles to have a general fast-clock-control capability built in)

2.2 Intended Use

Note that this section of the Standard is informative, not normative.

2.3 Reference and Context

2.4 Message Formats

In addition to the following Event IDs, the well-known event ID “Delivers Clock Protocol” is defined.

~~Separately documented to reduce change of inconsistency, as some cost in readability.~~

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

~~Existing Event Range definition is 0x01.01.99.01.01.01.xx.xx but that is in the Lenz XpressNet range now, and needs to change; easy change to existing code. Existing coding for content is hours, minutes in two lowest bytes; ditto.has been reserved for these messages.~~

~~Time Event Upper Part refers to~~

~~Date Event Upper Part refers to~~

~~Rate Event Upper Part refers to~~

~~NNN and NNNN event ranges The~~The upper (number TBD) bytes of the

~~Clock number is 0 to 15 to allow multiple clocks. Two upper bits reserved: Send as zero, ignore; to be used as flags. 16, 32 bit is send as zero, check; can be used either as either clock number expansion or for coding changes.~~following messages comes from an OpenLCB range that is allocated to the creator (or perhaps configurer?) of the clock generator node.

Note that there are no well-defined values of the clock event IDs. You locate the clock generators and their event spaces via the usual IdentifyEvent/IdentifyProducer mechanisms starting with the "Delivers Clock Protocol" event ID.

The details of the following are still being worked out. In particular, reducing the number of bytes needed down to three or even better two would simplify allocation.

2.4.1 Set/Report Time Event ID

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
~~Time Event~~Specific Upper Part				~~Clock Number~~(some key value)	Hours	Minutes	0

Hours are 0-23, minutes are 0-59~~, seconds are 0-59~~. ~~The top three bits in the hours field are reserved, must be ignored upon receipt so they can be used as flags~~. Other codes points in Hours~~;~~ and Minutes~~, Seconds~~ are reserved~~., may not be sent, shall result in ignoring the event upon receipt.~~ The low byte is reserved for eventual use as seconds. All reserved values shall not be produced, use shall result in the event not being consumed.

2.4.2 Set/Report Date Event ID

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
~~Date Event Upper Part~~Specific Upper Part				~~Clock Number~~(some key	Year since 1800	Month	Day

DRAFT

 Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
				value)			

How to code the year is an open question. Railways were invented around 1820. OpenLCB is meant to last a long time. 1820+256 = 2076 is enough? And what about people who want to model Roman chariot roads? Maybe should use some of the upper bits in the month byte for a 2047 year span.

Month is 1-12, Day is 1-31, other values are reserved, shall not be produced, use shall result in the event being not consumed.

2.4.3 Set/Report Rate Event ID

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
~~Rate Event Upper Part~~Specific Upper Part				~~Clock Number~~(some key value)	1/0 Run/Stop	Rate	

Rate is 12(?) bit fixed point rrrrrrrrrr.rr, 0.00, 0.25, 0.50, ..., 999.75 (Is that the right range? Enough granularity? Would something simpler be easier? A 999X clock is so fast that it saturates the bus)

Run/Stop = 1 clock is running; Run/Stop = 0, clock is stopped. Higher bits in that byte shall be zero.

Run/Stop bit is provided so that a single report tells you what the clock is doing. The stop/start clock event should be used to stop and start the clock, because that doesn't require resetting (or even knowing) the current rate.

~~Regardless of the rate, seconds may not be skipped due to the rate calculation.~~

2.4.4 Stop/Start Clock

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
~~Rate Event Upper Part~~Specific Upper Part				~~Clock Number~~(some key value)	05/04 Start/Stop	0	0

Start/Stop = 5, start clock. Start/Stop = 4, stop clock.

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

Having a separate start/stop mechanism, instead of just setting rate to zero, makes distributed start/stop control easier. There's no need for everybody to remember the current rate while the clock is stopped.

2.5 States

Each clock has an independent running/stopped state and an independent rate.

When the clock is in stopped state, it's internal time is not changing.

When the clock is in running state, it's internal time is advancing (rate X) as fast as normal time.

2.6 Interactions

The well-known event ID "Delivers Clock Protocol" shall be produced by every node when it first starts to operate as a clock generator. When enquired about, the node has to return "valid" if the clock is operating, and "invalid" if it is not. (Note that this is operating, not running; if it's serving as a clock generator and will produce and consume the appropriate event IDs)

A Set/Report Rate event for which there are one or more consumers are ~~is~~ produced every real 60(?) seconds while the clock is running to update the layout. (Should the current one~~it~~ also be ~~sent~~produced when the clock is not running? Sometimes nice to be able to stop all traffic on the network, which you can do if it doesn't send while the clock is stopped)(This is written in a way that makes options are available for e.g. low-bandwidth clocks, see below.)

The Set/Report Date event is produced how often? Every 60 seconds? Nodes that want it, e.g. to initialize a display, can ask for it from the (known) clock-producer node using IdentifyEvents. There's no real point in too long a delay, people waiting will get annoyed. Clock ~~generator~~producer nodes shall~~will~~ produce a Set/Report Date event if they receive one but do not update their internal date.

A Set/Report Time event is produced every time the current time changes, e.g. every fast ~~second~~minute. (Should it be periodically ~~sent~~produced, e.g. every real minute~~10 seconds~~, when the clock is stopped? That updates recent joiners, but it's nice to be able to stop all network traffic).

Note that nodes should not assumed that they will receive any time event only once. As time is set, it's possible for a node to receive the current time more than once.

When ~~If~~ a Set/Report Time event is received at a clock-producer node, it (optionally) sets the time in the clock producer node. If the time is not set, the current time event is produced immediately after.

If a Set/Report Rate event is received, the clock-producer's ~~clocks's~~ rate is set to the rate embedded in the event. The run/stop bits are ignored. If the clock does not support the requested rate, it moves to the closest non-zero supported rate, and ~~sends~~produces a Set/Report Rate event containing the current rate. Rate can be set while the clock is running or stopped. The clock ~~generator~~producer node may, but is not required to, produce a ~~send the a~~ Set/Report Time immediately after the rate event is ~~sent~~produced to ensure that all agree on the exact time. ~~This is more important if the clock ticks only every minute instead of every second.~~

The use of the same event to broadcast the rate and to set the rate should perhaps be revisited. Is that consistent with event semantics? Or should a "rate set event" do the set and cause the clock source to emit a "current rate" event? There's no security issue here, because we receive the Node ID of the set-

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

sending node either way. And a clock receiving a set command, no matter what the format, can still decide not to do it (if the protocol allows rejecting that, see above).

2.6.1 Event Identification and Reporting

When a clock ~~generator~~producer node receives an Identify Events message, it replies with:

- An IdentifyProducedRange and an IdentifyConsumedRange that covers the possible Set/Report Time events
- An IdentifyProducedRange and an IdentifyConsumedRange that covers the Set/Report Date events
- An IdentifyProducedRange and an IdentifyConsumedRange that covers the Set/Report Rate and Stop/Start Clock events.

In addition, the node will reply with:

- An IdentifyProducedEvent message for the current date, showing valid & active
- An IdentifyProducedEvent message for the current time, showing valid & active
- An IdentifyProducedEvent message for the current rate and start/stop state, showing valid & active

All of the preceding may be in any order.

(There's actually an standards issue here – with the current coding, the Date events don't occupy more than half of the reported range)

When a clock ~~generator~~producer node receives an IdentifyProducers message that covers any of the events it handles (Set/Report Time, Set/Report Date, Set/Report Rate, Stop/Start Clock) it will reply with a ProducerIdentified message showing valid. If the queried event is the current state (same time, same date, same rate & start/stop status, or same start/stop status respectively), the reply will be marked active. Otherwise, it will be marked inactive.

When a clock ~~generator~~producer node receives an IdentifyConsumers message that covers any of the events it handles (Set/Report Time, Set/Report Date, Set/Report Rate, Stop/Start Clock) it will reply with a ConsumerIdentified message showing valid. If the queried event is the current state (same time, same date, same rate & start/stop status, or same start/stop status respectively), the reply will be marked active. Otherwise, it will be marked inactive.

This lets devices rapidly establish information about a clock, even if they don't know it exists. To find the current clock (you have to know the clock number, but you can start with the default)

- Send a global IdentifyProducers for the Start/Stop event.
- If there's no reply, no clock exists. If there is a reply, extract the source node ID (alias) of the clock ~~generator~~producer node and send an IdentifyEvents.
- The reply will contain IdentifyProducedEvent messages that show the current clock ~~generator~~producer state via their active/valid status.

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

2.6.2 Slaving Clock Generators

For modular layout use cases (still being identified), it's useful to have a set of time events be created in synch with some other clock generator. For example, a modular segment has a clock generator and multiple nodes that are configured to together do complex automation. You want to include it in a larger modular layout without having to reconfigure this.

One way to do that is to slave the modular clock generator to another master clock generator, so that the master clock generator controls and specifies the time, and the modular clock generator produces its own events to represent that time. The consuming nodes then act on the modular clock events they are configured with.

Operationally, this slaving consists of reissuing the events with different top-end bytes, translating the top-end bytes from the master clock generators values to the slaved modular clock generator values.

Mechanically, this can be done by having a mode in the clock generator node that does this, having a gateway onto the modular OpenLCB segment that does the translation (and removing the original modular clock generator), or some other mechanism. The standard is about protocol, not implementation, and so is silent about this.

If a modular clock generator is to be configured to follow some other clock event range, that raises the question of how to do it. The Standard is silent about this too, but it's important that there be a way to do it within the Standard. Choices include:

- Configure it via the memory configuration protocol to listen for a particular value of the top bytes.
- Many layouts just have one clock generator. If that's true, a button on the clock that says “Slave to master” will get the right one. If there's more than one, pushing that button could just go to the next one.
- Use teach/learn or blue/gold to teach an event from the master clock, or some producer or consumer of the master clock, to the slave. For example, if there's a master “clock stop” button, teach that to the slave so that it knows that's the range to listen to. This can be done even if there are multiple clocks operating on the layout.

3 Background Information

This section will go directly into the Technical Note.

This protocol doesn't appear in PIP, because it's totally self-identifying. There's no formal, functional need to check for it. IdentifyProducers can find the clock ~~generator~~producer node(s) on the layout. Perhaps it should be added to the PIP mechanism to make it unambiguous that the node manufacturer intends this to be supported, trigger automated tests, etc.

3.1 Multiple Clocks

A user might want multiple clocks, for example a real-time clock to show current time, and a fast clock to show railroad time.

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

The protocol provides for this, but it's up to the user to allocate clock numbers and configure the clock producers~~generators~~ and consumers to use the proper clock number as part of their event range.

Since most railroads only use one fast clock, by convention products using this protocol should come configured for the "0" clock number. (Or should that be "1", as it's less confusing to non-digital people, and we have space?)

3.2 Configuration via a simple teach-learn UI

Blue and Gold configuration does not have to be implemented, perhaps many devices won't, but important to show it can be done. Configuring consumers to react to a particular time event can be done by putting a programming button on the master time ~~generator~~producer that sends the learn message for the current clock time.

Better (more powerful) UIs can then use the same underlying teach/learn implementation.

To Teach: gold; blue to select: hour, minute, rate; up/down to change; gold

To nominate: blue, blue; gold to select: h,m,r; up/down; blue

From the Arduino docs:

> The User Interface (UI) uses the Blue / Gold buttons:
>
> A (Blue/Gold)+ sequence Sets or Nominates the Time, Rate or Alarms to learn a Teach-event.
>
> Gold(Blue/Gold)+ sequence sends a Teach-event which includes the event associated with the current-Time, the Rate-, or any of the Alarms.

3.3 Alarms

Early OpenLCB clock prototypes contained support for "Alarms", special events that could be configured to be produced at a specific time. By configuring consumers to listen for the alarm event, instead of a particular time, it becomes easier to reconfigure the start or stop time for layout activities: Change one thing, the association between producing the alarm event and the time that it happens, and all the other nodes will follow without reconfiguration.

This protocol does not contain any specific support for alarms, because it doesn't need to. Simplicity is important, and the production of alarms can proceed completely independently. Any event ID could be produced by any node, upon receipt of the particular time event, and that don't have to be part of this protocol. An alarm-producer~~generator~~ isn't specifically part of the clock ~~generator~~producer. It could be anywhere. And no protocol needs to be defined for alarm events, as they're just regular produced events: When A is produced, produce B and C. That's a generally-useful thing to have, even for non-time events.

3.4 Low-Bandwidth Clock Producers

The Standard specifies that only time events with consumers need to be produced. Small clock-producing nodes can use this to reduce the bandwidth used by the clock events, particularly at high rate. The producing node can automatically generate and maintain a list of consumed event IDs, and produce only those.

DRAFT

 Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

Once the clock-producing node is up and running, it can listen for ConsumerIdentified messages and include any Set/Report Time event IDs seen in its list. Nodes that newly attach or restart will emit these automatically. So long as the clock-producing node isn't marked as a Simple node, these will be forwarded to it.

The process to find all consumers when the clock-producing node becomes active on an already existing OpenLCB network is similar, it just involves querying the entire network:

- Send a Verify Node Global, and construct a list of nodes from the response
- At a suitably metered interval, send a IdentifyEvent to each node.
- Construct the list of consumed event IDs from the response.

There is currently no support to do a network-parallel query such as "Identify All Consumers of EventIDs in the following Range" due to buffer-management and network-load-scaling issues with it.

3.5 History and Numerology

There are slightly more than 2^{16} seconds in a day, and just less than 2^{11} minutes. A 17-bit "seconds since midnight" format could be used, but it would still take three bytes, and is somewhat harder to read and display. It would keep the non-used code points together, perhaps better for expansion, but it's not clear how much time is going to expand.

By-seconds events allow much more precise timing, particularly at low fast-clock multipliers (clocks near real-time), but they result in too much traffic at high fast-clock multipliers (really fast clocks). A 10 minute fast 24 hour cycle is 144 messages per second, more than 15% of CAN bandwidth. Even faster rates, for example to do a fast recycle to prepare for a new sequence, would clearly saturate the bus. This has led to the choice of minute events, with seconds reserved for possible future consideration.

A denser format would be 5 bits hour, 6 bits minutes, 6 bits seconds all put consecutively, instead of in separate bytes. That still takes three bytes, but keeps the seven spare bits together.

Commands to change the run/stop state and clock rate are broadcast as separate events. That's better than coding them in the time events themselves, because if e.g. rate was separate bits in the event, simple nodes could not be configured to recognize specific times.

We're not sending events-with-only-minutes, or -only-seconds. Those make it easy to do something once a minute or once a second via consumer, but those are not to the only interesting intervals: Just do the calculation.

Regardless of whether the clock sends seconds or minutes, small times can be maintained via a local timebase synched to the time broadcast and the rate messages. There's a limit to how precise this can be, but it is probably worth a factor of 10 and perhaps 100 in resolution. Using received time is much simpler, and the smallest time commonly used by fast clocks is the second. Since OpenLCB networks can handle 3-4 messages/second for a fast clock with second resolution, it seems simplest to just broadcast the seconds.

3.6 Future Work

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.

Table of Contents

DRAFT

Copyright 2013. All rights reserved. See http://openlcb.org/Licensing.html for license terms.