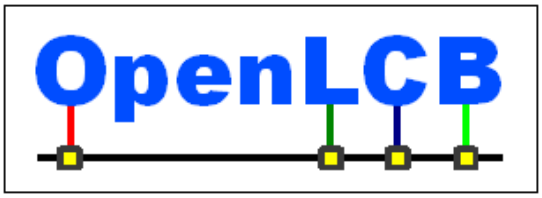

OpenLCB Working Note	
Time Broadcast Protocol	
Mar 24, 2013	Preliminary

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.

1.1 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. (Does it include seconds?)

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

A node connected to model building lights turns them on and off at specific times.

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 repeat directly without running through the rest of the day.

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

Layout lights execute a pattern that's different when the modeled day is Monday vs Saturday.

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

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 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 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 “08:10:13”, 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

The event ranges NNN and NNNN has been reserved for these messages.

Time Event Upper Part refers to

Date Event Upper Part refers to

Rate Event Upper Part refers to

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

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.

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.

2.4.1 Set/Report Time

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Time Event Upper Part				Clock Number	Hours	Minutes	Seconds

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, Minutes, Seconds are reserved, may not be sent, shall result in ignoring the event upon receipt.

2.4.2 Set/Report Date

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

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.

2.4.3 Set/Report Rate

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Rate Event Upper Part				Clock Number	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.

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				Clock Number	05/04 Start/Stop	0	0

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

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 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

A Set/Report Rate event is produced every real 60(?) seconds while the clock is running to update the layout. (Should it also be sent 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)

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. There's no real point in too long a delay, people waiting will get annoyed. Clock generator nodes 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. (Should it be periodically sent, e.g. every real 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.

If a Set/Report Time event is received, it (optionally) sets the time. If the time is not set, the current time is produced immediately after.

If a Set/Report Rate event is received, the 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 a Set/Report Rate containing the current rate. Rate can be set while the clock is running or stopped. The clock generator may, but is not required to, send the a Set/Report Time immediately after the rate is sent 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-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 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.

When a clock generator 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 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 and send an IdentifyEvents.
- The reply will contain IdentifyProducedEvent messages that show the current clock generator state via their active/valid status.

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 nodes 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.

The protocol provides for this, but it's up to the user to allocate clock numbers and configure the clock 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 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-generator isn't specifically part of the clock generator. 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 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.

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 procise 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.

3.5 Future Work

Table of Contents

