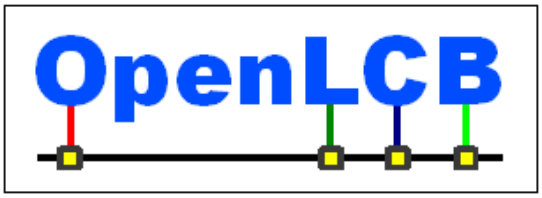


OpenLCB Technical Note	
Time Broadcast Protocol	
Apr 7, 2013	Preliminary

1 Introduction

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.

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

This Standard does not completely solve the use cases in this section.

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, none of which cover all needs:

- 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, but it's not possible to automatically which events are consumed for state changes (light on), and which for instantaneous occurrences (toll bell).

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.

2 Annotations to the Standard

2.1 Introduction

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

Clock events are produced by a clock generator, and consumed by clock faces and other devices that want information about time. The events themselves are called “clock events”, as they carry information about the current clock time.

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 upper parts can be separated into well-known “public” ones, and self-allocated “private” ones. Much of the first part of this section is to ensure that the self-allocated upper parts result in unique event IDs that cannot inadvertently collide.

The “Default Fast Clock” and “Default Real-Time Clock” terms are not further defined in the Standard. They have no operational effect. They're provided to make it easier for a clock face manufacturer to set up the clock face node to automatically find the default fast clock generator, without user involvement, when attached to an OpenLCB installation. One fast clock system is expected to be the most common installation, and this makes it very convenient for the user.

The values are packed into two bytes to make it easier to allocate the unique “Specific Upper Part” by using an OpenLCB unique ID.

2.4.1 Set/Report Time Event ID

2.4.2 Set/Report Date Event ID

2.4.3 Set/Report Year Event ID

(It's been proposed that the year instead be in some BCD format, which represents 999 years around some epoch)

2.4.4 Set/Report Rate Event ID

A rate of zero is valid. 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.

(should one bit be reserved, perhaps in the high nibble, to carry the run/stop state when reporting the rate? That was in an earlier proposal, makes it easier to look for the entire state in a single event, but it's got messy semantics)

2.4.5 Stop/Start 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

The “independent” in the definition of states means that clock generators work independently of each other, and don't have linked states.

It's up to the node implementor to decide how to implement the internal clock time. It could be an internal time generator of some kind, or linked to some external time reference.

2.6 Interactions

The reply to an enquiry about “Delivers Clock Protocol” depends on whether the clock generator is operating, not running. It's operating if it's serving as a clock generator and will produce and consume the appropriate event IDs. This is provided to allow a clock generator to be placed on or removed from an operating layout.

Status events are not produced while the clock is not running so that it's possible to stop all traffic on the network.

The Set/Report Rate interaction is written in a way that makes options available for e.g. low-bandwidth clocks, see below. A simple implementation can just produce them always.

Nodes that want the date, e.g. to initialize a display, can ask for the Set/ReportDate event status from the clock-producer node using IdentifyEvents.

Nodes that receive an event ID that would set the rate, date or year, but do not actually set to the specific value, send an event ID with the value they do set. There may be other consumers listening and tracking the state of the clock, and producing these event IDs keeps all those in synch.

Note that nodes should not assume 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.

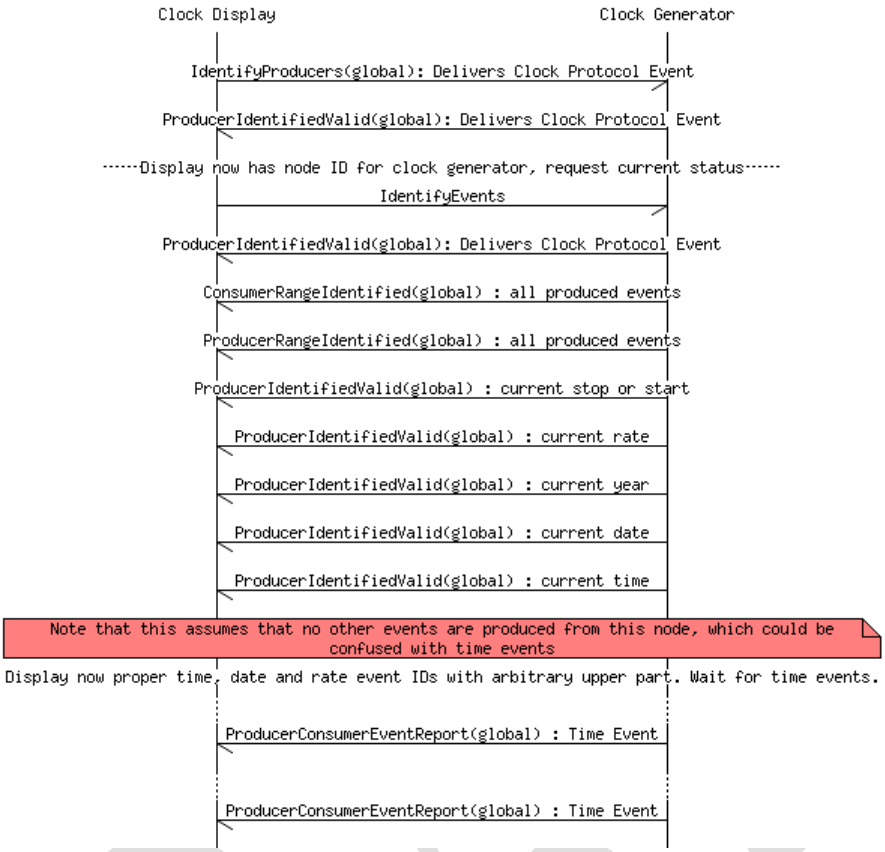
The clock set event ID sets the time exactly, including the seconds and fraction of seconds. Repeated production of that event will reset the time to the same value.

OpenLCB doesn't have latency guarantees, so different nodes may have slightly different ideas of the time. In practice, the differences should be much less than a second for operating OpenLCB installations.

Clock generators may also respond to other events to set their time. For example, to allow slaving of a clock generators events to time events from another clock generator, a clock generator may react to time event IDs with another high six bytes. How, when, whether to do that is an internal configuration option of the clock generator node, and is not discussed by the Standard.

2.6.1 Event Identification and Reporting

The event transfer protocol combines with these event definition to let devices rapidly establish information about a clock, even if they don't know it exists.



To find the current clock:

- Send a global IdentifyProducers for well-known event ID “Delivers Clock Protocol”
- If there's no reply, no clock exists. If there is a reply, extract the source node ID (alias) of the clock producer node and send an IdentifyEvents.
- The reply will contain IdentifyProducedEvent messages that show the current clock producer state via their active/valid status, along with IdentifyProducedEvent messages for all other producers on the node.
- Clock events from the four well-known clocks can be identified by the well-known upper six bytes of the event ID. Clock events from clock generators with other upper-six-byte values must be identified by another means, see “Further Work” section below.

3 Background Information

This protocol doesn't appear in the Protocol Identification message, because it's totally self-identifying. There's no formal, functional need to check for it. IdentifyProducers can find the clock producer node(s) on the layout.

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 via multiple clock generators with unique upper parts in their event IDs, but it's up to the user to allocate clock generators and configure the clock producers and consumers to use the proper clock number as part of their event range.

Multiple clock generators using the same event ID range will not function properly. The source node IDs in the Producer-Consumer Event Report messages produced by the multiple clocks can help diagnose and correct this.

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 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 isn't specifically part of the clock 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. 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.

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 Event IDs in the following Range” due to buffer-management and network-load-scaling issues with it.

3.5 Slaving Clock Generators

For some modular layout use cases (still being identified), it's useful to have the time events be created in synch with a local modular 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 generator's values to the slaved modular clock generator's 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.

3.6 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. Sparse clocks would help with this, but are significantly more complex for the clock generator, which may need a lot of event storage to keep rates low. This has led to the choice of minute events, with seconds reserved for possible future consideration.

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 of the day.

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.

4 Future Work

Perhaps this should be added to the Protocol Identification mechanism to make it unambiguous that the node manufacturer intends this to be supported, trigger automated tests, etc.

There is no fully-reliable solution to finding time event IDs with a non-common upper part. A node that produces its own time events can be found by asking for producers of the “Delivers Clock Protocol” event. The clock events will then appear in the set of ProducerIdentified and ProducerRangeIdentified replies to an IdentifyEvents message to the node. But it may not be possible to disentangle the clock events from other events produced by the node, particularly if the node produces multiple ranges of events. A better solution for this would be useful. This doesn't interfere with operating and configuring clock producers and clock faces manually. In that case, the events are known by the person doing the configuration, or a configuration tool is passing an event ID from the clock generator to the clock.

255

Table of Contents

