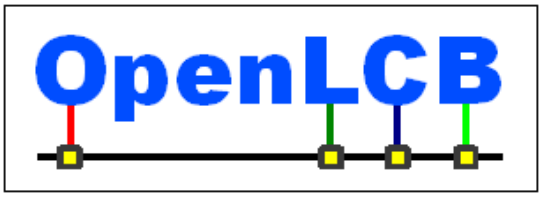

OpenLCB Technical Note	
Broadcast Time Protocol	
Apr 25, 2021	Adopted

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 world time, so that an evening train can be run during the day. Model railroad clocks might run at a different rate than real time (e.g., faster), 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 the 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 with specific meanings 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, such as starting and stopping the clock, or setting the rate or current time 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 information may be defined in the future.

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 can be distributed around the layout. It starts and stops independently of the fast clock.

A node connected to drive lights in modeled buildings 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. A node controlling a speaker can play a church bell sound at 12 noon (based on either model or real world time).

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. A simple input node can be configured to control a fast clock with a few common configuration-defined settings, such as buttons or a toggle switch to stop and start the clock, a few buttons to select a rate, and a button to set time to 8:00am. The clock running/stopped state can be displayed with a LED on a simple node.

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 twice 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 determine 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.

Both of these use cases should be solved using 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

This specification is in the context of the following OpenLCB-CAN Standards:

- OpenLCB Event Transport Standard, which defines messages for transporting Event IDs and identifying producers and consumers.
- The OpenLCB Event Identifiers Standard, which defines the format and content of Event IDs including the class of Well-Known Event IDs and Automatically-Routed Event IDs.
- OpenLCB Unique Identifiers Standard, which defines the allocation of OpenLCB 48-bit unique identifiers

For more information on format and presentation, see:

- OpenLCB Common Information Technical Note

2.4 Message Formats

The “Specific Upper Parts” can be separated into well-known “public” Identifiers, and self-allocated “private” Identifiers.

The “Default Fast Clock”, “Default Real-Time Clock”, “Alternate Clock 1”, and “Alternate Clock 2” 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. A single 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 Identifier.

While the terms “Fast Clock” and “Real-Time Clock” are intentionally undefined, one valid, and likely common, interpretation of these terms may be as follows.

- Fast Clock – A clock that runs at some ratio, typically faster, to real world time.
- Real-Time Clock – A clock that runs at a 1:1 ratio to real world time.

2.4.1 Report Time Event ID

While time is always represented in 24-hour form (00:00 is midnight) in the transmitted messages and clock state, this is independent to how time is displayed to the user. For products that have a clock display, it is up to the manufacturer to decide whether to use 12-hour or 24-hour time display, or provide a user-controllable setting for this. There is no provision in this Standard for centrally establishing formatting settings; manufacturer could provide configuration via physical buttons or the established configuration standards like the Memory Configuration Protocol and Configuration Description Information standards.

2.4.2 Report Date Event ID

These two event types are structured in bytes for ease of programming. It is not a goal for a hexadecimal dump of the packet be human-readable (hence not using BCD encoding).

2.4.3 Report Year Event ID

The range chosen, 0 AD to 4095 AD, is chosen to be sufficiently inclusive of the whole relevant span of railroad history typically represented in model form both in the past and for the foreseeable future.

2.4.4 Report Rate Event ID

A rate of zero means the clock is stopped, though it is preferred to use the Stop/Start Events for starting and stopping the clock. Negative rates mean the modeled time is moving backwards. Negative rates are represented using standard signed integer formats (i.e., 2's complement notation), for example 0x4800 is -512.00, 0x4801 is -511.75, … 0x4FFE is -0.5, and 0x4FFF is -0.25. The rate is intended to be a ratio measured against real world time.

2.4.5 Set Time Event ID

These event IDs are exactly 0x8000 shifted from the matching Report event IDs for ease of programming.

2.4.6 Set Date Event ID

These event IDs are exactly 0x8000 shifted from the matching Report event IDs for ease of programming.

2.4.7 Set Year Event ID

These event IDs are exactly 0x8000 shifted from the matching Report event IDs for ease of programming.

2.4.8 Set Rate Event ID

These event IDs are exactly 0x8000 shifted from the matching Report event IDs for ease of programming.

2.4.9 Query Event ID

This event ID will cause the clock generator to report it's entire internal state.

2.4.10 Stop/Start Event ID

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.

2.4.11 Date Rollover Event ID

This event ID works around race conditions involving changes in date and time due to midnight rollover.

2.4.12 Undefined/Reserved Event ID's

Any Event IDs not explicitly defined by the standard are reserved for future use.

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

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.

Note that the time is only advertised with minute granularity. If a Set Time is received when the current clock is 20 seconds past the minute, setting the time will set the time to exactly the minute received, with zero as seconds value.

It is not expected that it will be routine for clock generators to refuse to set their time, nor that other nodes will attempt to set it when not allowed to. The Standard is silent as to whether the seconds counter inside the clock should be reset to zero at this point. However, it can be implied that a Report Time Event is sent when the seconds count is equal to zero, and when an Event Producer Identified Valid message for the Report Time Event is sent, the seconds count is not assumed to be equal to zero.

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. It is allowable, though not required, for a clock consumer to perform an average on any observed jitter of of the incoming Report Time Events such that it can factor out that jitter on the node's internal tracking of the passage of time. The exact algorithm is not prescribed.

Clock generators may also set their time in response to other events, not specified here. For example, to allow slaving of a clock generator's events to time events from another clock generator, a clock generator may react to time event IDs with another high six byte Specific Upper Part. How, when, whether to do that is an internal configuration option of the clock generator node, and is not discussed by the Standard.

It is also permissible for a clock generator to have other mechanisms of setting its configuration including the current time or rate; one example would be pushbuttons on the generator node; another to have a consumed event for re-setting to the start of an operating session with configuration-determined values (e.g. 6:00am on October 3, 1953, rate 2.5x).

2.6.1 Startup

As stated in section 1.1 above, there are use cases, such as a clock face display, where a node will be consuming either all or the vast majority of clock events. In this case, it is preferable to use the Consumer Range Identified message in order to announce consumption of clock events.

There is another use case where a node may only be consuming a single or few clock events. In this case, it is preferable to use the Consumer Identified message to announce consumption of clock events. Some clock event consumers do not know anything about the concept of time, or make any distinction

between clock events and other non-clock events. From their point of view, they are generic event consumers.

The distinction of these two sets of messages can (and shall) be used to filter messages that need to be sent to the bus vs those that can be suppressed.

2.6.2 Clock Report

The reason to limit the Report Time Event generation to once per real world minute is not motivated by conserving bandwidth. Even with no filtering the fastest clock rate of 511.75:1 would result in about 1% of the bandwidth of a CAN bus segment being used up. Rather, the motivation behind limiting the Report Time Event generation is to prevent adding too much clutter in bus capture logs. The desirable property is that a layout bus should have no messages passing by in quiescent state.

It is important to note that, unless following one of the noted exceptions, a clock generator is not required to produce any time events for which a specific event consumer has not been identified – a consumer specifying the entire event range does not count here.

The process for a clock generator to identify all the clock event consumers is not explicitly specified. Two possible options include:

1. Send an Identify Events Addressed message to each node in the network as described in section 3.10 below.
2. Send an Identify Events Global message.

Clock consumers that display the time are required, in the absence of all event IDs being produced, to keep track of time internally. It is recommended that a clock display have an internal clock of sufficient accuracy to correctly extrapolate fast time into at least one real world hour without divergence between different units visible on the display. A common crystal oscillator has much better accuracy than this requirement.

A separate Date Rollover Event is required so that a clock consumer which tracks and progresses time internally can distinguish between a Report Time Event as a request to set the clock back in time within the same day versus a progression in time into the next day. The reasoning behind the three second delay for producing the Report Year and Report Date events is such that network delays do not result in smart clock consumers becoming confused about what the actual year and date is when their internal implementation of time rolls over asynchronous to clock Report Time Events.

2.6.3 Clock Synchronization

The requirement to always produce the next minute event is added so that clock consumers can learn the second counter of the clock producer and synchronize to each other.

2.6.4 Clock Query

When a clock consumer wants to know the current time, it may also want to know the Start or Stop state, Rate, Year, Date, and Time. The sequence prescribed allows all of this information to be provided in a short burst sequence.

This interaction can be used by a clock consumer to inquire time at it’s startup in case it has missed the announcements of the clock producer, or during running when it is uncertain about its internal clock’s accuracy because it hasn’t seen a time report for a long while. It is desirable however, that layouts with

many clock consumers to be able to start up and operate without each of them individually requesting and receiving clock updates. Instead,

1. clock consumers should listen to each others' requests for clock query and suppress their own query when another query is seen;
2. clock producers may delay the reply to the clock query by a few hundred milliseconds in order to catch any duplicate clock query commands and respond with a single announcement to all of them.

2.6.5 Clock Set

The Start or Stop state, Rate, Year, Date, or Time may each be set independently. If the desire is to change more than one of these settings at one time, it is recommended, though not required, to first stop the clock, then request the Rate, Year, Date, and/or Time settings, and finally (if desired) start the clock.

A race condition is possible between two nodes simultaneously requesting to adjust the Rate, Year, Date, and/or Time of a clock. While it is unpredictable (and possibly diverging) in which order the individual nodes of the layout will see the Set events, the clock generator is required to echo the effective value after setting; these echoed messages, even if multiple of them are sent in a short period of time, will always be received in the same order by all nodes on the bus. Any possible remaining ordering problem (e.g. on Start/Stop events) will be fixed to convergence three seconds later when the current state is announced.

It is not required that a clock generator support the setting of its time through events. A clock generator may, for example, only accept the setting of its time through a local push button interface, or by using the memory configuration protocol. This would fall under the category of an "out of band" mechanism.

3 Background Information

This section is general information of interest to the reader, implementers, etc.

3.1 Multiple Clocks

A user might want multiple clocks, for example a real-time clock display to show current time, and a fast clock display 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 Time Zones

The standard is purposely silent on the topic of time zones. It is up to the user to determine if and how time zones are represented. There are two options that are proposed here, and other solutions may also be possible.

1. The clock generator sends out time representing one coordinated universal time (UTC). Each time consumer is then programmed to add or subtract an offset of the universal time appropriate to their specific location. This is similar to how modern computer systems get and display their local time.

2. Create a separate clock generator for each local time to be represented. One of the clock generators is designated the “master” to which all the other clock generators synchronize to, adding or subtracting their “local” offset appropriately. Section 3.11 below has more information about how this can be achieved. A special case of this option is when only one time zone is in effect for the entire railroad: the clock generator may be set to railroad local time and no time offset needs to be configured on the clock listeners / displays. This will be the majority use-case.

3.3 Daylight Savings Time

The standard is purposely silent on topic of daylight savings time. This is in part because of the hyper locality specific nature of Daylight Savings Time. This is not something that is easily predictable by simple machines and can even change based on local legislation. It is possible for a time consumer implementation to provide a means for the user to enable/disable Daylight Savings Time and adjust what is displayed accordingly.

3.4 Day of the Week

There are a number of use cases where a day of the week may be relevant.

1. Church bells that have a special ring schedule for Sundays.

2. Commercial buildings that may not turn on as many lights on a weekend.

3. Display the day of the week on a clock face in order to indicate the time table being used in a given operating session.

There are no events defined to represent the day of the week. A time consumer can use standard language library API’s in order to convert a date to day of the week. These libraries exist for most modern programming languages (C/C++, C#, Objective-C, Java, Python, etc…).

Simple nodes will need to use facilities from a time cue-node to schedule events being produced at certain days only. Day-of-week events would not help simple nodes to achieve use-cases 1. and 2. above, since those would require a complex logic operations on the day-of-week and time events, including keeping track of a 7-way state machine and intersecting its state to the time events. A 7-way state machine exceeds the capabilities of currently available simple nodes. A much more robust solution is to use a pair of events output from a time cue-node (see Section 3.9) for covering precisely the desired time interval, which could be “Sunday 10am” vs “everything else” for case 1; “Friday 10pm – Saturday 5am” and “Saturday 10pm – Sunday 5am” for case 2; and “Wednesday” vs “not Wednesday” for one light output of case 3. This solution can easily be made compatible with time jumps.

3.5 Jumps in Time

Through configuration, it is possible that a clock generator would make a large jump, forward or backward, in time from its previous set time. The only requirement about this jump is that the sequence described in section 2.6.5 above follows the jump.

It is allowable, though not required, for a a clock generator to produce intermediate Report Time Events, as well as Report Year and Date events, at a highly accelerated rate. One possible use case for this is to “reset” time for nodes that consume clock events but otherwise have no knowledge of the concept of time. If either of these, or similar, techniques are deployed, the sequenced described in section 2.6.5 above is still required once the application of the technique is complete and the new set time is reached.

It is not prescribed how this might occur, but one can imagine a number of different scenarios:

1. Generic event consumers are setup to turn on/off lights at certain time throughout a simulated day. The end of an operating session occurs asynchronous to a full cycle. The operator wants to reset time to the beginning of the day for the next session. In order to reset the interactions, the clock generator replays the new set time’s previous 24-hours in the form of events before settling on the new time.
2. A jump forward in time occurs. In order for generic event consumers to progress their state to this new time, intermediate clock events are sent at an accelerated rate until the new set time is reached.
3. Generic event consumers are taught to consume one additional event, the “start of operating session” event, re-setting their state to the desired state at that point. This does not allow arbitrary jumps however.

3.6 Date Rollover

In some cases it may not be desirable to have a clock generator rollover at the end of a day into the next day, but rather start all over at the beginning of the current day. This would prevent the progression of the date, but allow the continuous 24 hour cycle of time. The standard is quiet on this use case, but it is certainly a possible implementation option in a clock generator.

3.7 Configuration via a Simple Teach-Learn User Interface

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.

For Example:

> 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

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

3.8 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 method is also sometimes called “cueing”.

This protocol does not contain any specific support for alarms or cueing, 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 the details of that do not have to be part of this protocol. An alarm-producer isn't specifically part of the clock producer. It could be also shipped with a clock consumer or a generic consumer configured to a specific time event. And no protocol needs to be defined for alarm events, as they're just regular produced events: When event A is consumed, produce B and C. That's a generally-useful thing to have, even for non-time events.

3.9 Time-range cues

A not-well-served case of the Standard is the operation of simple consumer nodes for turning on their outputs for desired periods of time, e.g. a street light to turn on during night time. Learning the direct event ID of the time when night should start and stop works in simple progression, but requires workarounds presented in section 1.2 and 3.5 to support jumps in time.

A way to correctly support jumps in time is by an advanced clock cue node. This cue node keeps track of the time internally by being a clock consumer. By configuration (e.g. via the CDI) it knows which time range the user is interested in; automatically produces the on and off events when the time passes these boundaries, and specifically produces an on or off event in case it detects a jump.

Such a clock cue node could also be shipped optionally as an added feature of a clock generator.

3.10 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 maintain a list of consumed event IDs, and produce only those.

Once the clock-producing node is up and running, it can listen for Consumer Identified and Consumer Range Identified messages and include any Report Time Events 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 an Identify Event Addressed 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.

3.11 Slaving Clock Generators

For some modular layout use cases (still being identified), it's useful to have the time events be created synchronous 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 a different “Specific Upper Part”, translating the “Specific Upper Part” 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), using a node that can do alarms/cueing for the needed events, 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.

3.12 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.

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.

Regardless of whether the clock sends seconds or minutes, small times can be maintained via a local timebase synchronized 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.

The choice of filtering as a requirement to the clock producer came from the wish to keep the bus quiescent for monitoring/logging/debugging purposes when there is no human interaction with the layout.

4 Future Work

The existence of this Simple Time Protocol does not preclude the existence of other protocols for the conveyance of time. It is possible that future time related protocols could be developed in order to realize the unserved use cases identified in section 1.2 above.

420

Table of Contents