RR-CirKits

Specializing in Affordable Electronics for Model Railroads

Installation Guide

Revision-b

Jan 2024

Manual for Firmware Rev-1.04

Tower LCC+Q

LCC (Layout Command and Control

16 line Input Output board

Plus STL Logic

This PDF is designed to be read on screen, two pages at a time. If you want to print a copy, your PDF viewer should have an option for printing two pages on one sheet of paper, but you may need to start with page 2 to get it to print facing pages correctly. (Print this cover page separately.)

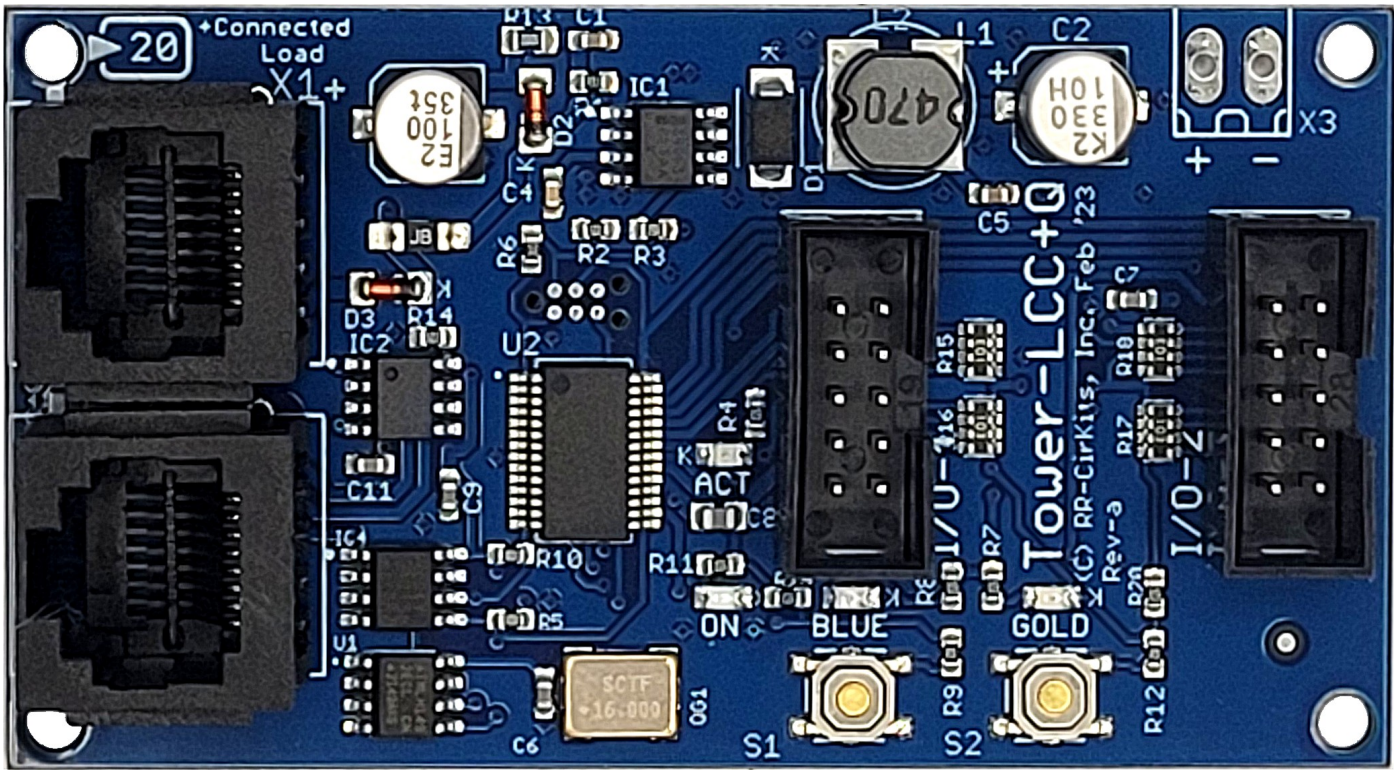
Table of Contents

You can download an editable version of this document from
http://www.rr-cirkits.com/manuals/TowerLCC+Q-manual-b.odt

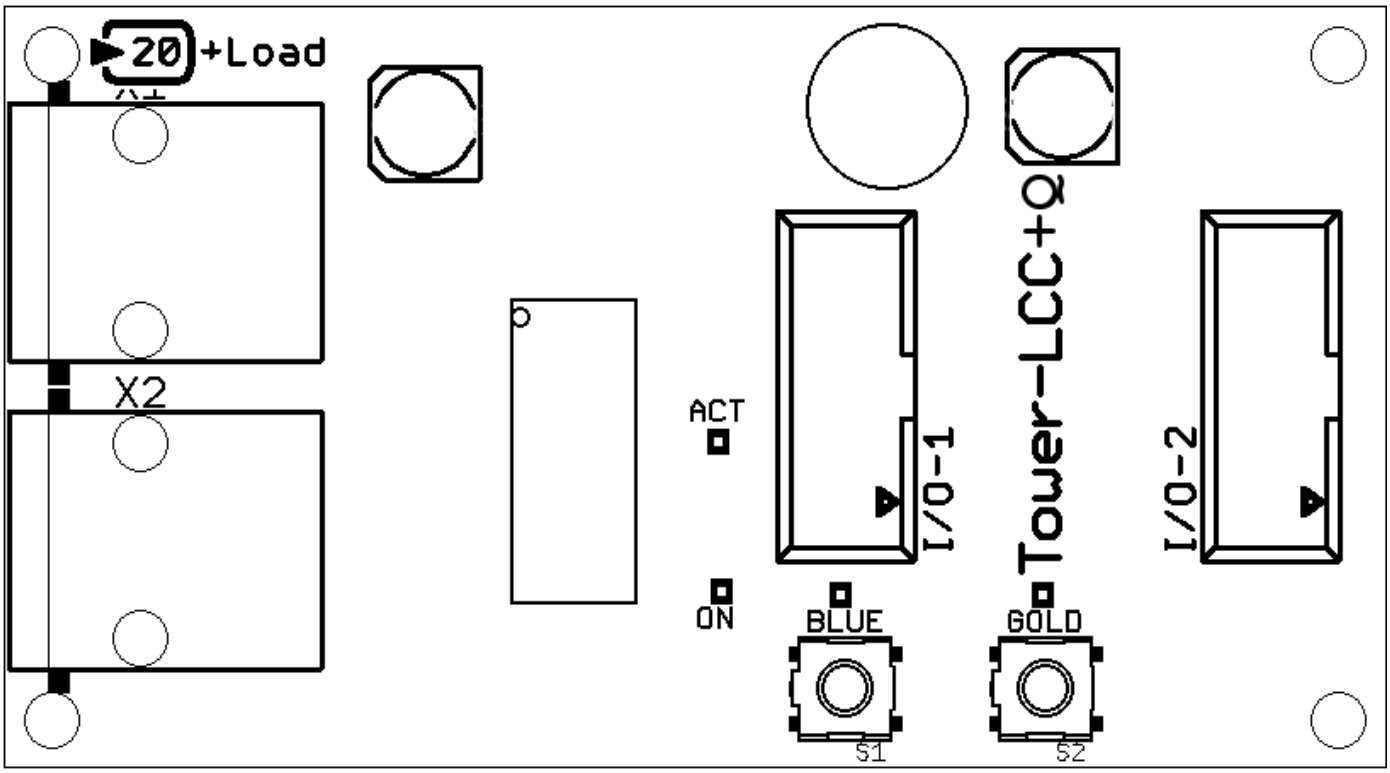
Overview

The Tower LCC+Q (Layout Command & Control) interface provides a simple and easy way to connect between the NMRA LCC® CAN bus and the layout. The Tower LCC+Q may be connected at any convenient point on the NMRA LCC® CAN bus.

LCC® is a registered trademark of the NMRA. www.nmra.org



Tower LCC+Q Image



Tower LCC+Q Connectors

1 About LCC

The NMRA LCC® is a subset of the OpenLCB specifications created by the OpenLCB group for Layout Command and Control. http://www.openlcb.org/

NMRA LCC® devices are controlled by events. Each event has a unique value that will never be repeated by any other LCC® event in use anyplace on your system, nor even on anyone else's system. The only meaning given to any specific event is that which you give it.

This event uniqueness is a key differences between the LCC and legacy systems. You can always create a link between any two points in the LCC world without needing to know anything else about the system. No more address conflicts, no more dedicating addresses to specific hardware, no more address space size limits.

The LCC uses 64 bit numbers to represent events, (18,446,744,073,709,551,615 possibilities) so we are not planning to run out of event numbers anytime soon. Events are created by event 'Producers' and used by event 'Consumers'. The same event may be created by one or more Producers, and may be used by any number of Consumers. (or none at all)

Events happen, they are not states nor the status of indicators. The only memory of events past exist in hardware. An event can tell you to turn a light 'on'. A different event can tell you to turn a light 'off'. Different events can tell you to turn the same light 'on'. However there is no event that tells you that the light is 'on'. That is a state, and only resides in the hardware that controls the light, or hardware that watches the state of the light. In this node we call these memory items ‘Variables’, and they are used to store the information about layout items (actual or imaginary) for use at some later time. In order to do logic operations The Tower LCC+Q node includes groups of internal memory items (called Variables) to remember the states that have resulted from the various past EventIDs.

1.1 Some Definitions

1.1.1 Node

We use the term ‘Node’ to indicate a single device or board that has both an electrical and logical connection to an LCC network. Some nodes may have multiple logical connections to the network, but only count as one node because there is only a single electrical connection. (transceiver) Some devices may have an electrical connection to the network, but not interact with the LCC logically in any way. An example might be a Repeater. It is electrically connected, but other nodes can not interact with it in any way. It does not count as an LCC node, but must be accounted for when counting the number of devices on a segment.

Please note that this may not be exactly the same usage of the term ‘node’ as is documented in the NMRA LCC specifications.

1.1.2 Segment

On a CAN based LCC network there are electrical limitations on the total number of devices connected to the same cable, or ‘Segment’. These limitations take several forms. Electrical limits may be overcome by the use of a repeater.

- **Electrical current limits.** The CAT5 cable used has a limitation of 1A per conductor. The user is responsible to assure that sufficient power is supplied to the cable to supply all nodes within 20’ of a power injection point without exceeding this amount. Each node is marked with the amount of current required or supplied to assist the user in this calculation. Be sure to count external loads such as signal lamps, etc.
- **Propagation delay.** The speed of any CAN network is inversely proportional to its total length. The LCC CAN network was chosen to run a maximum of 1000’/300m at 125K bits per second. This is a good compromise for most layouts.
- **Length.** The maximum cable length of 1000’/300m is reduced by 20’/6m for each physical node attached to the segment. This limits any segment to about 48 nodes. The maximum cable length is also shortened by the use of a Repeater due to the delay inherent between segments.
- **Topology.** Each CAN segment should be a single serial string of nodes with a termination at each end. Short branches are allowed, but they count as double their length when subtracted from the segment total. This *required* termination serves both as the current source for a ‘0’ bit, and the cable termination to prevent prevent signal distortion.

In the CDI itself each major section is also called a “Segment”. This use of the word has nothing at all to do with the electrical segment discussed above. It is simply a way to divide different (normally related) portions of the CDI from each other to make it simpler to deal with.

1.1.3 Line

Each Tower LCC+Q contains 16 I/O lines. Each line has the ability to watch for 6 events (consumers) and to send out 6 events. (producers) Each line has two registers. One register remembers the 'State' of the line. (on or off) The second register remembers the state of the 'Veto' option. The veto option allows or disallows some of the events used to control or respond to the line.

1.1.4 Consumer (Output Function)

The Output Functions are called **Consumers**, because they “Consume” or “Read” messages to tell them what to do. The Output Line may be configured in 5 ways. The hardware has an internal function generator that may be configured to create different types of actual outputs.

Output Type:

- **None** (the output line is disabled)
- **Steady** (output line follows the output state),
- **Pulse** (output line alternates one time when the output state is first ‘on’),
- **Blink A** (output line alternates on/off while output state is 'on').

- **Blink B** (output line alternates off/on while output state is 'on').

Drive Polarity:

- **Low (0V)** The output line is low when true. (default)
- **High (5V)** The output line is high when true.

Delay:

The function generator also includes delay settings for both 'on' and 'off' transitions. These same delays are used to control the blink rate and pulse length, or to simply delay the output action for some interval after the controlling event is seen. (e.g. to simulate “running time” on a CTC panel)

Action:

Each consumer event can be configured to control the line's output or veto register state in one of 9 ways:

- **None**
- **On** (Line Activate), **Off** (Line Inactivate)
- **Change** (Toggle)
- **Veto on** (Active), **Veto off** (Inactive)
- **Gated On** (Non Veto Output), and **Gated Off** (Non Veto Output)
- **Gated Change** (Non Veto Toggle)

The consumer events may also control a Veto state. If the veto state is 'on', then the consumer 'Gated on' (activate) and 'Gated off' (inactivate) events are ignored.

The producer 'on' (non vetoed input), and 'off' (non vetoed input) are also ignored (blocked) when the veto state is 'on'.

1.1.5 Producer (Input Function)

The Input functions are called **Producers** because when they are activated they “Produce” or “Create” messages in response to their activity.

The input line may be configured in 3 ways:

- **None** is no response.
- **Normal** response is used when an input change directly controls the sending of events.
- **Alternating** action is used when a single line needs to produce alternating control events. (e.g. turnout normal, reverse)

Each producer event can be configured to trigger in one of 10 ways:

- **None**
- **Input On**, **Input Off**, allow you to create events simply based on a change of the input line. This is the normal use for a producer.
- **Gated On (Non Veto Input)**, **Gated Off (Non Veto Input)**, allow events to respond to, or ignore, any input changes based on the veto state. For example this would allow you to enable/disable fascia buttons for local control of a turnout by using the output events from a panel switch to control the veto.

- **Cascade** is the trigger option that allows you to create a new producer event based on the commands to both activate and inactivate the output state. E.g. to cascade a yard ladder. (Cascade combines the following two commands)
- **Output State On Command** (activate), **Output State Off Command** (inactivate), are trigger options that allow you to create a new producer event based on the command to activate or inactivate the output. E.g. to cascade a yard ladder.
- **Output On (function hi)**, **Output Off (function lo)**, create a new event when the output of the function generator changes. This might be used to build a realistic traffic light controller. Be careful with this option because it can create a lot of traffic continuously, especially if the function output is blinking rapidly.

The 'on'-'off' time delays are used as function output delay or input debounce delay depending on the line status.

1.1.6 Sample Mode

Sample mode is used for Berrett Hill Touch Toggles or other dual mode situations where the input and output states of a line may not necessarily be the same. In Sample mode the line is normally driven by its output state, but it briefly disables the output drive and reads the un-driven (pull-up) state of the line. The input must be current limited for the case where the input and output are not the same. The normal output load must also be tolerant of brief changes during the input sample interval. The output must not load the line with more than 10K to prevent the load from causing false inputs.

Sample mode is automatically enabled if both output and input functions are enabled on the same line. Several different I/O modules are now compatible with Sample mode. This allows you to connect both input modules and output modules at the same time on the same port. This can save costs by combining dissimilar functions on the same hardware.

2 Tower LCC+Q Features

- The Tower LCC+Q uses the CAN bus implementation of the NMRA LCC.
- Communicates over the LCC CAN bus at 125Kb.
- Support for a total of 16 Input/Output lines:
  - Up to 16 Input Lines. (internal pull-up termination on all lines)
  - Up to 16 Output Lines.
- Internal Logic Engine with up to 4096 characters of logic space.
- Support for up to 16 virtual code lines. (8 aspects per Track Circuit)
- CDI (Configuration Description Information) controlled programming via Software.
- Lines may be configured as individual input, output, or shared, (sampled) lines.
- Automatically saves input/output and logic states during power down.
- Boot Loader allows contact less user firmware upgrades over the LCC® (Layout Command & Control) connection.
- Power is supplied over the LCC® bus. The TowerLCC requires 20mA. plus whatever load may be imposed by the I/O modules that you choose.
- Efficient switcher regulated 5VDC is available on each I/O port connector to power external modules or lamps.

2.1 Electrical Specifications

I/O Port 1:
Pin 1 - 5 volt logic level at 25mA.
Pin 2 - 5 volt logic level at 25mA.
Pin 3 - 5 volt logic level at 25mA.
Pin 4 - 5 volt logic level at 25mA.
Pin 7 - 5 volt logic level at 25mA.
Pin 8 - 5 volt logic level at 25mA.
Pin 9 - 5 volt logic level at 25mA.
Pin 10 - 5 volt logic level at 25mA.

I/O Port 2:
Pin 1 - 5 volt logic level at 25mA.
Pin 2 - 5 volt logic level at 25mA.
Pin 3 - 5 volt logic level at 25mA.
Pin 4 - 5 volt logic level at 25mA.
Pin 7 - 5 volt logic level at 25mA.
Pin 8 - 5 volt logic level at 25mA.
Pin 9 - 5 volt logic level at 25mA.
Pin 10 - 5 volt logic level at 25mA.

Maximum current to be supplied by all I/O lines combined is 200mA. This 200mA total limit means that not over 8 lines may supply their maximum current at any one time.

3 Line Details

3.1 Consumer (Output Function)

3.1.1 The Output Line

The hardware has an internal function generator that may be configured in 5 ways to create different types of actual outputs:

- **None** (the output line is disabled)
- **Steady** (output line follows the output state),
- **Pulse** (output line alternates one time when the output state is first ‘on’), the pulse to either high or low level is based on the timing delay intervals, then return to normal. Delay ‘Interval 1’ sets the time delay before the pulse occurs, and delay ‘Interval 2’ sets the pulse length itself.
- **Blink A** (output line alternates on/off while output state is 'on'). Normally used to control devices such as crossing gate flashers directly from the output lines. ‘A’ and ‘B’ are the two phases of the flashing. Blink A or Blink B refers to which phase starts the action.
- **Blink B** (output line alternates off/on while output state is 'on'). Delay ‘Interval 1’ sets the length of phase A, and delay ‘Interval 2’ sets the length of phase B. Normally it will be more realistic to use the LED drivers available on a Signal LCC node that can also add in the proper fade effects.

3.1.2 Drive Polarity:

- **Low (0V)** The output line is low when true. (default)
- **High (5V)** The output line is high when true.

3.1.3 Delay:

The function generator also includes delay settings for both 'on' and 'off' transitions. These same delays are used to control the blink rate and pulse length, or to simply delay the output action for some interval after the controlling event is seen. (e.g. to simulate “running time” on a CTC panel)

3.1.4 Action:

Each consumer event can be configured to control the line's output or veto register state in one of 9 ways:

- **None**
- **On** (Line Activate), **Off** (Line Inactivate)
- **Change** (Toggle)
- **Veto on** (Active), **Veto off** (Inactive)
- **Gated On** (Non Veto Output), and **Gated Off** (Non Veto Output)
- **Gated Change** (Non Veto Output)

In addition to controlling the Output state, the consumer events may also control a Veto state. If the veto state is 'on', then the consumer 'Gated on' (activate) and 'Gated off' (inactivate) events are ignored.

The producer's 'on' (non vetoed input), and 'off' (non vetoed input) are also ignored (blocked) when the veto state is 'on'.

3.2 Producer (Input Function)

3.2.1 The input line

An input line may be configured in 3 ways:

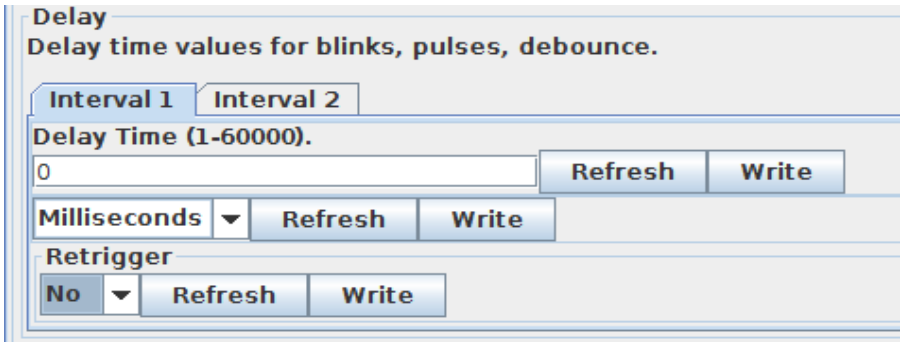
- **None** is no response to the input.
- **Normal** response is used when an input change directly controls the sending of events.
- **Alternating** action is used when a single line needs to produce alternating control events. (e.g. turnout normal, reverse)

3.2.2 Input Trigger:

Each producer event can be configured to trigger in one of 10 ways:

- **None**
- **Input On, Input Off,** allow you to create events simply based on a change of the input line. This is the normal use for a producer.
- **Gated On (Non Veto Input)**, **Gated Off (Non Veto Input)**, allow events to respond to, or ignore, any input changes based on the veto state. For example this would allow you to enable/disable fascia buttons for local control of a turnout by using the output events from a panel switch to control the veto.
- **Cascade command**, Combines the 'Output State On' and 'Output State Off' commands to always send the selected EventID if the consumer for this line is activated or deactivated. This is a shortcut to cascade a yard ladder.
- **Output State On Command** (activate), **Output State Off Command** (inactivate), are trigger options that allow you to create a new producer event based on the command to activate or inactivate the output. This is used to specify a cascaded command in a single direction.
- **Output On (function hi)**, **Output Off (function lo)**, create a new event when the output of the function generator actually changes. This might be used to build a realistic traffic light controller. Be careful with this option because it can create a lot of traffic continuously, especially if the function output is blinking rapidly.

3.2.3 Delay



Each line includes two delay timers that are used to control blinks, pulses, and input debounce times.

Interval 1 controls the ‘on’ delay or time, and Interval 2 controls the ‘off’ delay or time. The count may be set from 0-60,000 and the base interval may be set to Milliseconds, Seconds, or Minutes. This allows for delays from 1ms. to over 41 days. Probably the extremes will never be required, but this gives you a good range to choose from. Actual accuracy is ½% or better. Millisecond times are calculated to the nearest 8 ms.

Retrigger allows the time interval to be reset if the line state is set to ‘true’ again prior to the end of the delay time.

If the line state is set to ‘false’ prior to the end of the delay time, then the output does not occur.

The same timers are used for both input and outputs, so there are some combinations that are not allowed.

3.3 Sample Mode

Sample mode is used for Touch Toggles or other dual mode situations where the input and output states of a line may not necessarily be the same. In Sample mode the line is normally driven by its output state, but it briefly disables the output drive in order to read the un-driven (pull-up) state of the line.

The restrictions are:

- Any input must include a 1K series resistor to prevent shorting out any output that may be active at the same time.
- Any output must not load the line with more than a 10K load to prevent the load from creating a false input.
- Also note that the output function must be tolerant of the brief sample times when the output may change state during the sample period.

Sample mode is automatically enabled if both output and input functions are enabled on the same line. Several different RR-CirKits I/O modules are now compatible with Sample mode. This allows you to connect both input modules and output modules at the same time on the same port. This can save costs by combining dissimilar functions on the same hardware. (e.g. occupancy and turnout control)

4 Power and Serial Connections

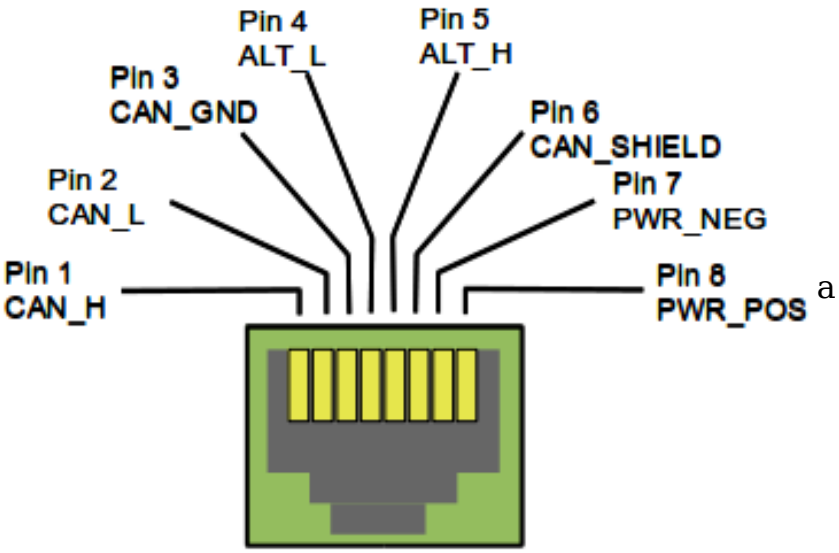
The Tower LCC+Q (16 Line I/O Board) has four connectors and four status indicators. Two of these connectors are for connections to the LCC bus network. The other two are used as connections to the I/O lines. This section covers the system connections consisting of the CAN bus port connectors, power connections, I/O port connections and Status indicators.

4.1 CAN LCC® Compatible Connector

The data connection is made to the Tower LCC+Q via a standard RJ-45 CAT5 cable connected to either of the two RJ-45 connectors. The LCC wiring passes straight through both connectors.

The LCC specification requires minimum of 1’ of cable length between connectors. Slightly shorter cables (10”) should not significantly impact operation.

These cables are commonly sold for wired Ethernet use.



a

Pin outs for the CAN LCC RJ-45 data connector:

Pin	Description
1	CAN H
2	CAN L
3	CAN GND
4	Alt L (DCC negative)
5	Alt H (DCC positive)
6	GND
7	GND
8	+Power 12-27V

LCC power is supplied on Pin 7 and Pin 8. Power can be from +12VDC to +27VDC. The RR-CirKits LCC Power-Point delivers approximately 15VDC to the bus.

The LCC connectors accept standard Ethernet style CAT5 (or better) cables. 4 pair cables are required by the Tower-LCC+Q. For any but the smallest networks it is recommended that you choose AWG 24 CAT-5 wiring or AWG 23 CAT6 wiring. The use of AWG 26 wiring reduces the maximum length of your network to approximately 40% of its specified length. Especially avoid using copper clad aluminum wire or AWG 28 low profile wiring as they have even higher than normal resistance at the relatively low frequencies used by the LCC. This higher resistance shortens the maximum distance for reliable communications even more than using AWG 26 wiring does.

A note on connectors: RJ-45 crimp connectors are made with three blade styles. (the end of the contact that crimps into/onto the wire) Single 'U', double 'UU', and triple 'VVV' points. Stranded cables may be made with any of the three blade styles because the points crimp into and between the individual wire strands. However if you are using solid wire, then you must only use the three point style of blade. ('VVV') It is designed to trap the solid wire between the three points, two on one side, and the center one on the other side, for a corrosion tight connection. The single or double pointed blades will simply press against the side of the solid wire, and will fail in time. (usually the morning of your open house)

4.2 Power Connections

The Tower LCC+Q requires an external power source of between 7.5 and 27 volts DC from the LCC cable.

The LCC Power-Point unit is a convenient way to supply the required power to the Tower LCC+Q and other LCC boards over standard RJ45 cables.

LCC Power-Point shown with Terminator

Each segment of LCC® cable requires a terminator at each end. Power can also be supplied by other powered LCC modules, or with the RR-CirKits LCC Repeater.

4.3 Status Indicators

The Tower LCC+Q has two status indicators located near to the LCC connectors. The green ON status indicator shows the power status of the Tower LCC+Q itself. The red ACT (activity) indicator normally shows all data activity on the bus, and also any activity/error status during a boot loader firmware upgrade. (see section 10.0)

4.4 Blue/Gold Buttons and LEDs

A limited amount of configuration may be accomplished on some manufacturer's nodes by using the Blue and Gold push buttons and indicators. The primary use is to link up producer and consumer lines. The Tower LCC+Q does not support this option.

The Gold LED can indicate two different error messages. If it is flashing (10% duty cycle) it indicates that it is idling in forced boot loader mode. If the Gold LED is blinking (50% duty cycle) it indicates that the board was unable to initialize itself on the network, most likely because it could not establish an alias.

4.4.1 Setting up Virtual Code Lines

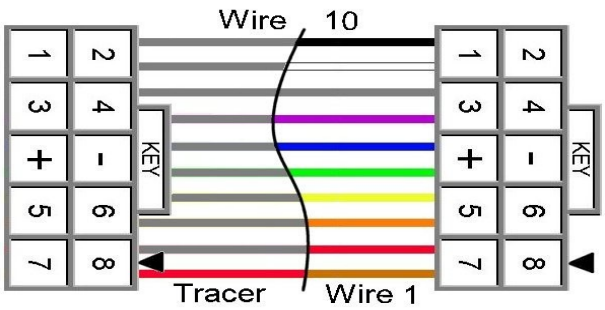
Use the CDI tools to setup links. The EventID for the TX code set will always come from the transmitting node and be entered into the receiving node to avoid accidental reuse of EventID numbers from setup to setup.

Each node can setup one or more virtual code lines to any other node. For simplicity these virtual links are named 'Circuit Y1', ‘Circuit Y2’, etc. It is incumbent upon the user to keep track of which 'Coded' virtual links are created between nodes. Be sure to record which block (1-16) is used for each side of the virtual links if you have not standardized these connections. There is no need to use the same 'block' number on both sides of any virtual coded track circuits, and in fact they will not normally be matching. Normally these virtual links will follow along with the rails, but there is no actual requirement that they do so.

Each track circuit links the logic with one out of a group of eight line states. Normally these are used to represent the track speed allowed at arrival to the next mast. This is based on the ‘Aspect’ shown by the mast.

4.5 Tower LCC+Q I/O Connector Wiring

The two port connector's wiring is as follows. Note that the pin numbers and I/O line numbers are NOT the same, and actually run opposite to each other.

Pin number	Connection name	Image
1	line 8	
2	line 7	
3	line 6	
4	line 5	
5	Ground	
6	+5VDC	
7	line 4	
8	line 3	
9	line 2	
10	line 1	

10 position IDC cable numbering and description

5 Getting Started

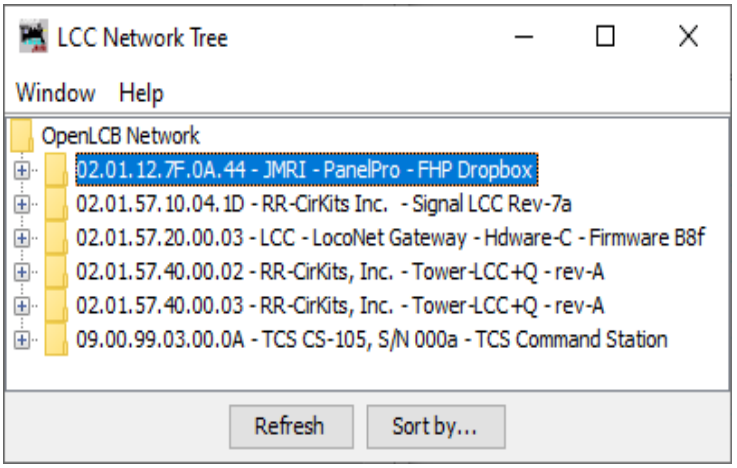
To properly display the Tower LCC+Q CDI you will need to use JMRI DecoderPro 5.6 or later <http://www.jmri.org/> to configure the Tower LCC+Q. This "point and click" interface will save you much time and frustration while setting the many possible options that you will need to configure, and in fact are the only way that we suggest for configuring the Tower LCC+Q node.

Node Address: Each Tower LCC+Q has a single node address that is used for CDI programming on the layout. Each individual Tower LCC+Q has its node address imprinted on a label on the back side of the board. It is recommended that you name your node with a friendly user name as the first step in configuration.

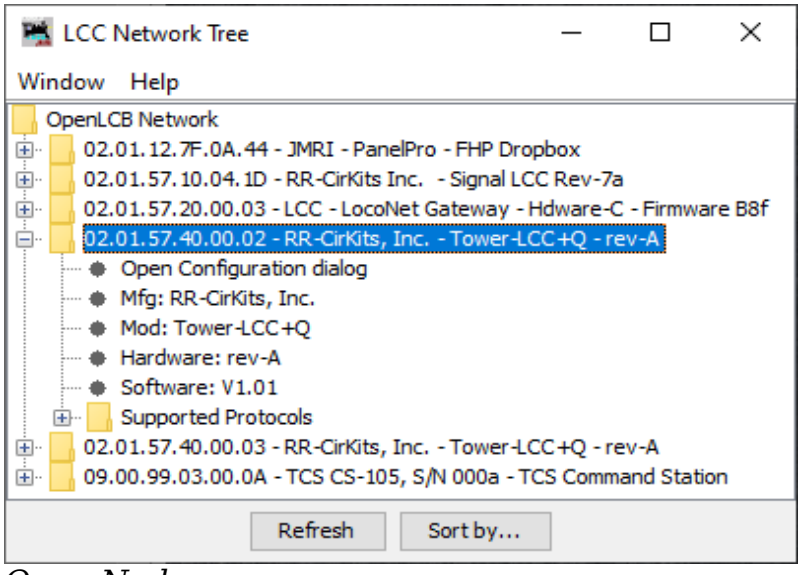
5.1 CDI (Configuration Description Information)

The CDI is the tool used to access the LCC node's internal configuration options. Instead of relying on printed manuals or volunteer created files to present the various decoder options, (like DCC devices have for the past 20 years) the LCC specification expects the manufacturer of the LCC node itself to present its capabilities and options in a standardized manner from an internal file. This allows any LCC configuration tool to be used interchangeably, and not need to be updated to support new hardware or firmware upgrades.

Start up the CDI tool. Once the tool is monitoring the LCC network you will be presented with a list of nodes similar to this.



The list should include all the nodes that are currently visible on your LCC network. In this example the first entry in the list (02.01.12.7F.0A.44) is the configuration program itself as seen through the interface.



Open Node

Select the node that you desire to configure and click on its '+' to open it. This will open up further options for that node. In this example there are various options including 'Open Configuration dialog' and a list of 'Protocols Supported'.

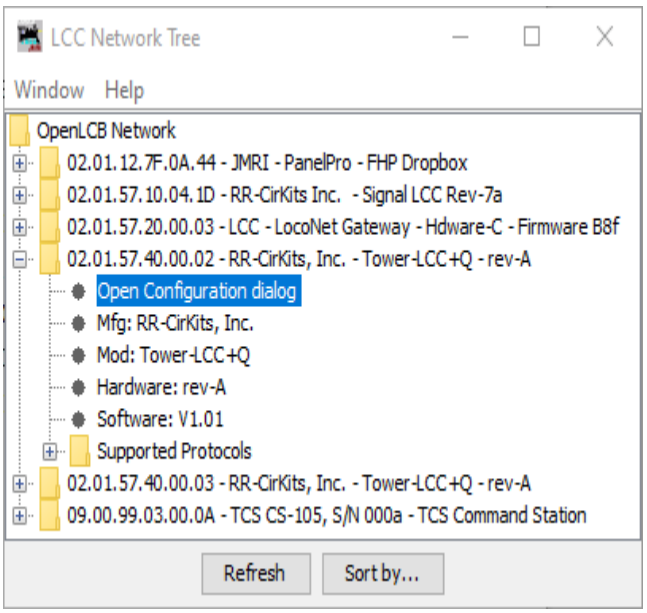
The Node Information will help you to be sure that you have chosen the correct node.

With the JMRI CDI program you simply highlight an item to open it in a new window.

CDI is the supported Protocol that you will need for configuration purposes.

Remember that with the LCC this display information is provided by the manufacturer and stored in the node itself rather than in some piece of paper, external file or program.

Once the CDI window opens up you can modify its contents by using the 'Refresh' and 'Write' buttons found near to each item.

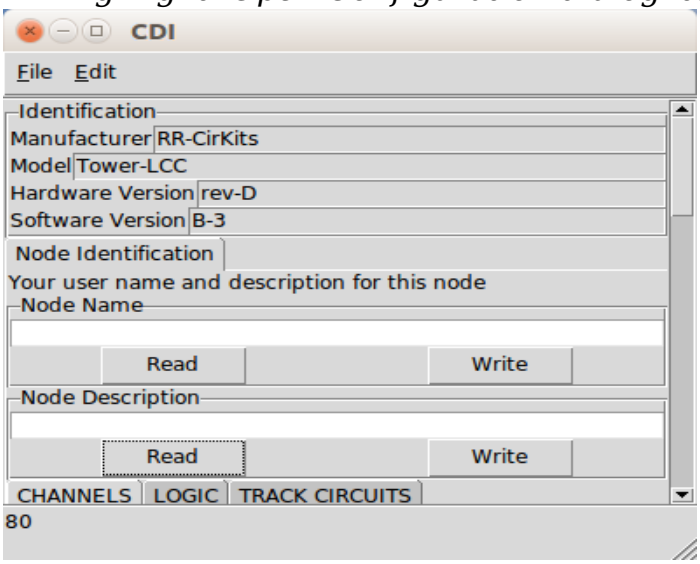


Highlight 'Open Configuration dialog' to

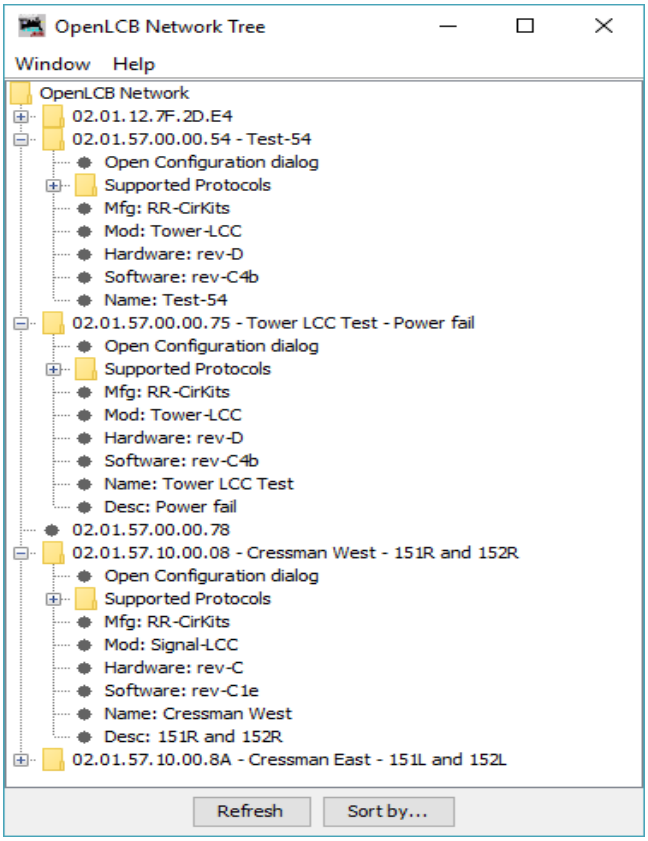
The **Refresh** button will present a new, unused, default value if one has not been previously stored.

There is also a '**Refresh All**' button located at the bottom of the window. This will reload all the EventID current values from the device's internal storage.

The **Write** button will store the currently displayed value or selection into the



Deepsoft CDI Window



Open the JMRI CDI Window

node's memory. If you have changed any value you must always then do a 'Write' to store it into the node before it can take effect.

The JMRI CDI tool will highlight the entry with orange until it has been written to the board. This is a helpful reminder that the change has not yet been stored into the board where it can take effect.

5.2 Input/Output Configuration

We suggest that the user take advantage of the JMRI CDI tool or a similar program to set the Tower LCC+Q configuration values.

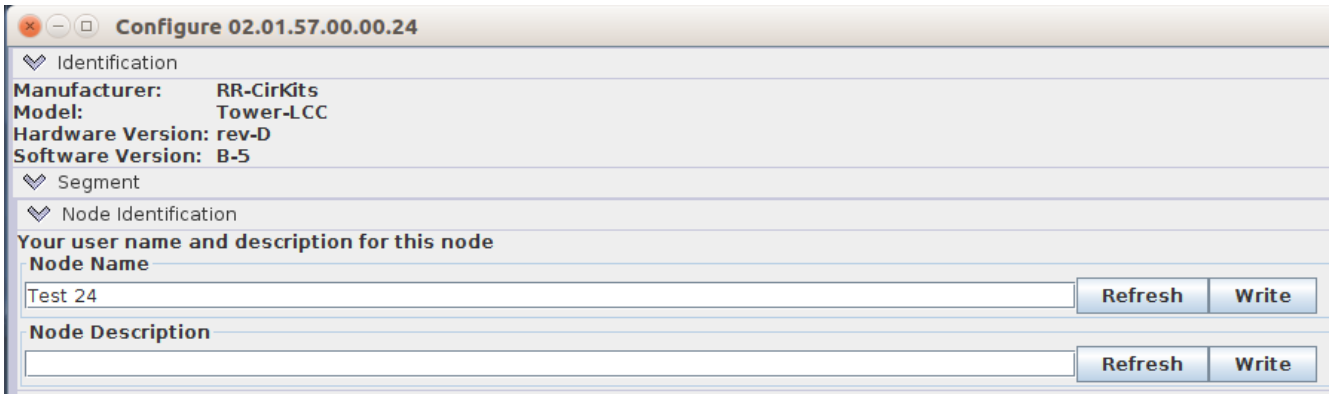
The following examples are using the JMRI CDI tool for the Tower LCC. Select the 'LCC' drop down list and click on 'Configure Nodes'. When the node selection window opens, choose the node to configure, click on 'Open Configuration dialog'.

This will open the CDI tool and automatically read in the basic information for the node.

This information is presented in a tabular format to allow a reasonably compact display but still have easy access to the vast amount of configuration information.

5.3 Identification

The first section shown will be the **Identification**. It includes the manufacturers



JMRI CDI Window

name and node model plus any version information.

5.4 Node Identification

The next item is the **Node Identification**. It contains the Name and Description that you give to the node. The name of this example node is 'TEST 24'. This name will appear in the node selection window to make it easier to select the correct node for configuration. There is a 63 character limit to the node name and description items.

5.5 Line (I/O Ports)

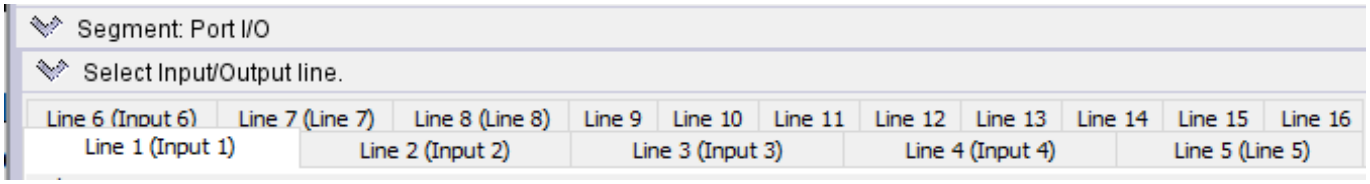
The Tower LCC+Q has two 8 bit Input/Output ports for a total of 16 lines. Each port is normally configured to be either all Inputs, or all Outputs, to be compatible with the various RR-CirKits I/O modules. However each line may also be individually set as either input or output for special purposes.

For the special case of one wire I/O a line may even be configured as both input and output at one time. (Sample Mode section 1.2) The Berrett Hill Touch Trigger is an example of a one line device. The Tower LCC+Q can also control both the indicator's color, using consumers, and report the output, using producers in the same channel.

Any special effects may be applied differently for each line. E.g. one line may be held steady while another sends a pulse or is blinking.

5.5.1 Lines

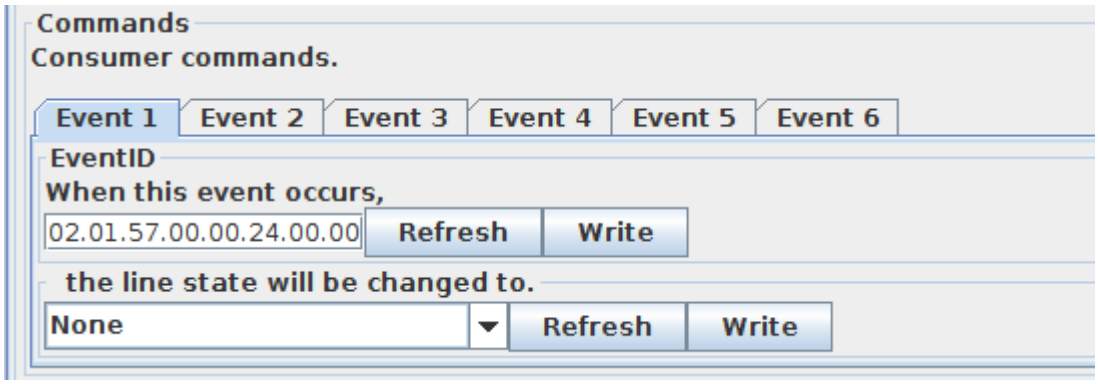
Each I/O line is called a 'Line' and is selected by tabs and presented separately. Note that lines 1-8 appear on Port 1, and lines 9-16 appear on Port 2 of the node.



5.3.4 Commands

Commands are consumer events that are sent to the line. A command can directly control the line, for example by turning it on or off. A command may also indirectly control the line, for example by controlling its veto status. Each line has 6 consumer events associated with it. Each Consumer event is associated with a single task or action.

For example event 1 could be used to turn an output 'on', and event 2 could be used to turn an output 'off'. The consumer events are:



- **None** –
- **On (Line Active)** – Sets the output state to 'true'
- **Off (Line Inactive)** – Sets the output state to 'false'
- **Change (Toggle)** – Changes the output state to its opposite state
- **Veto On (Active)** – Sets the veto state to 'true'
- **Veto Off (Inactive)** – Sets the veto state to 'false'

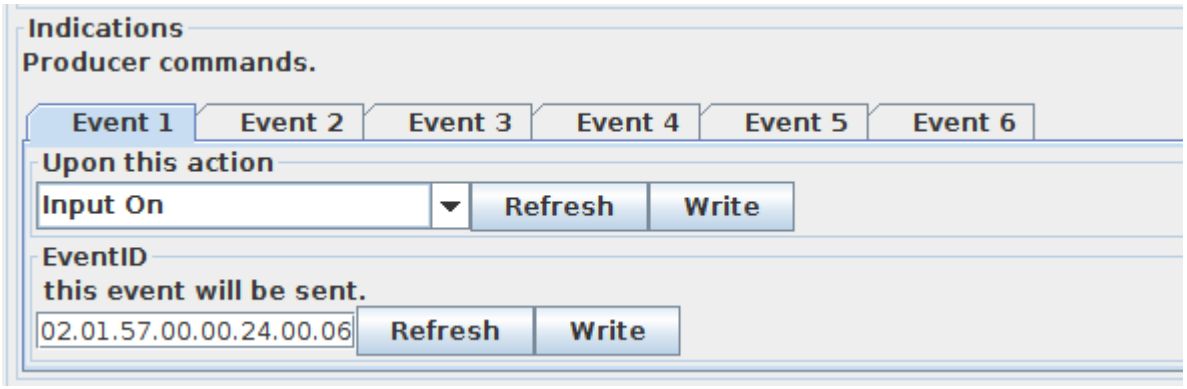
- **Gated On (Non Veto Output)** – Sets the output state to ‘true’ if veto is ‘false’
- **Gated Off (Non Veto Output)** – Sets the output state to ‘false’ if veto is ‘false’
- **Gated Change (Non Veto Output)** – Inverts the output state if veto is ‘false’

The last three items probably need some clarification. An event set to ‘*On*’ will always set the output state to ‘true’. However an event set to ‘*Gated On*’ will only set the output to ‘true’ if the veto state is ‘off’. If the veto state is ‘on’ then the three ‘Gated’ events are ignored, and the line does not change state.

For example when a CTC operator controls a turnout he would send the events configured as ‘On’ and ‘Off’. However a local operator button would send the (different) events configured as ‘Gated On’ and ‘Gated Off’. The CTC operator could then send a ‘Veto On’ event that would block the local operator from controlling the turnout, but still allow normal operation from his own panel. Then the CTC operator could send a ‘Veto Off’ event that would re-enable the local operator’s control over the turnout.

5.3.5 Indications

Indications are producer events that are sent to the bus. Each line has 6 producer events associated with it.



Each Producer event is associated with one action. For example event 1 could be used to indicate that the input line is changed to ‘active’, and event 2 could be used to indicate that the input line is changed to ‘inactive’. The consumer events are:

- **None** – This event is not created in response to anything.
- **Input On** – Responds to an input level change to ‘true’
- **Input Off** – Responds to an input level change to ‘false’
- **Gated On (Not Veto Input)** – Gated response to an input level change to ‘true’
- **Gated Off (Not Veto Input)** – Gated response to an input level change to ‘false’
- **Cascade command** – Responds to any output state change

- **Output State On command** – Responds to an output state change to ‘true’
- **Output State Off command** – Responds to an output state change to ‘false’
- **Output On (Function hi)** – Responds to an output level change to ‘high’
- **Output Off (Function lo)** – Responds to an output level change to ‘low’

5.3.6 Tower LCC+Q Secondary Messages

In these examples there are no second or third messages being sent in the sense of our previous LocoNet products. However additional messages may be sent or responded to if desired by utilizing unused events. For example to activate a 'Next' turnout whenever the 'first' turnout event is sent, in order to sequence a yard ladder, you could use the ‘Cascade command’ to send an event to the next turnout in the ladder. These additional messages, if enabled, are sent whenever the primary event occurs. Another simple example would be to allow two or even three different messages to be sent when ever a single button is pressed. For example a single event could be the master reset for many turnouts simply by adding it as an additional ‘On’ or ‘Off’ event to each turnout in the group.

6 Tower LCC+Q STL Logic Overview

The logic contained in the Tower LCC+Q is a major departure from our previously available logic. In the past we included some simple conditional logic that could be configured by filling in boxes or selecting options from a list. This was useful, and powerful enough to do basic signaling. However it was not capable of creating generic logic statements, nor even complex signal logic without needless repetition.

To get around these restriction we have completely eliminated the previous conditional based logic and replaced it with a simplified version of STL (Statement List) logic.

For those folks already familiar with STL, we have removed all temporal functions. In other words we can not count, do math, nor do any analog operations. We can preset timers to fixed values, but we can not change timer values based on the result of logic operations. (other than by choosing between different timers)

We have also intentionally restricted any program branching (Jumps, Go-to, etc.) to only be allowed in the forward direction. This prevents the creation of any loops, both intentional or accidental. The logic itself steps through its statement list every 20mS recalculating each logic statement found in the current path. If a recalculation results in a change from a previous state, then that change is used to create one or more EventIDs representing the new logic state.

6.1 Statement List (STL) for the Tower LCC+Q

The STL for the Tower LCC+Q is a subset of the Siemens S7-x language.

6.1.0 Preface

In addition to its 16 I/O lines, and virtual track code lines, the Tower LCC+Q also includes 32 logic operator groups. This logic is state driven, not event driven. To interface the logic with LCC EventIDs we provide two tables to convert Inputs and Outputs into their corresponding events. The STL logic may be used to create signaling logic or other animations such as control of grade crossings. It may also be used to create NX (eNtry Exit) routing.

6.1.1 Purpose

This section of the manual is your guide to creating user programs in the Statement List programming language STL.

The manual also includes a reference section that describes the syntax and functions of the language elements of STL.

6.1.2 Basic Knowledge Required

The manual is intended for Tower LCC+Q programmers, operators, and maintenance/service personnel.

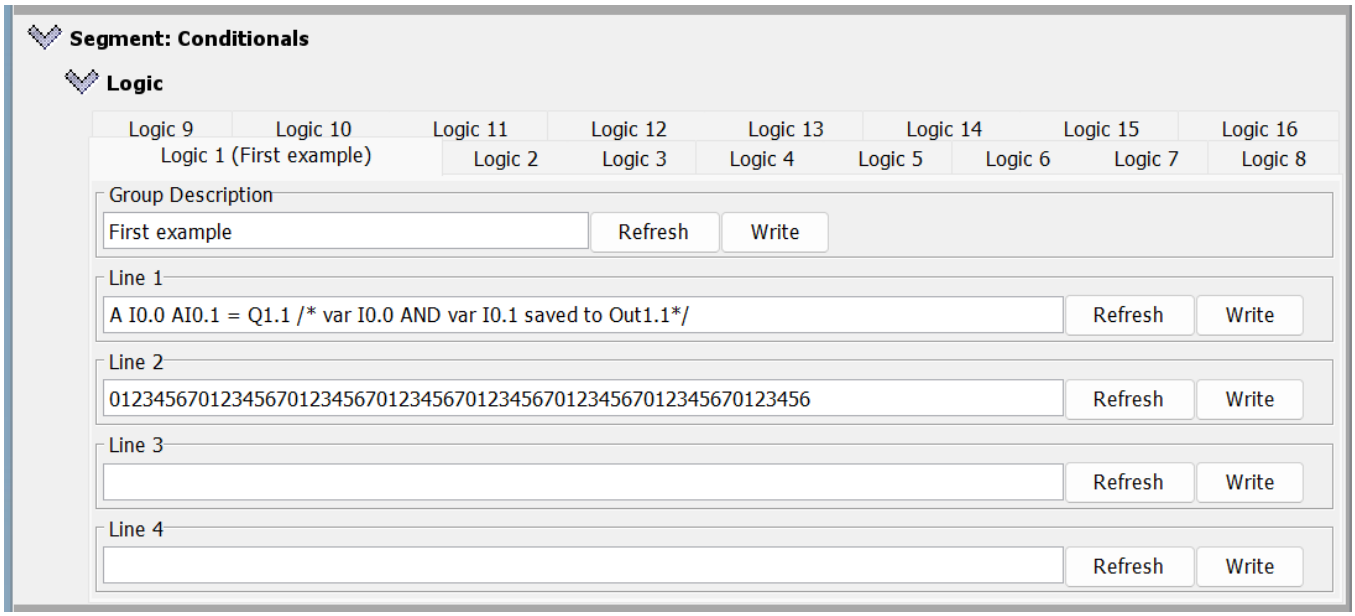
In order to understand this manual, general knowledge of automation technology is required.

In addition to, general computer literacy and the knowledge of other working equipment similar to the PC and the operating systems MS Windows, Apple macOS, or Linux, a copy of JMRI 5.6 or later is required to configure this system.

This consumer grade hardware and software is not intended to be used in any part of a safety-critical system.

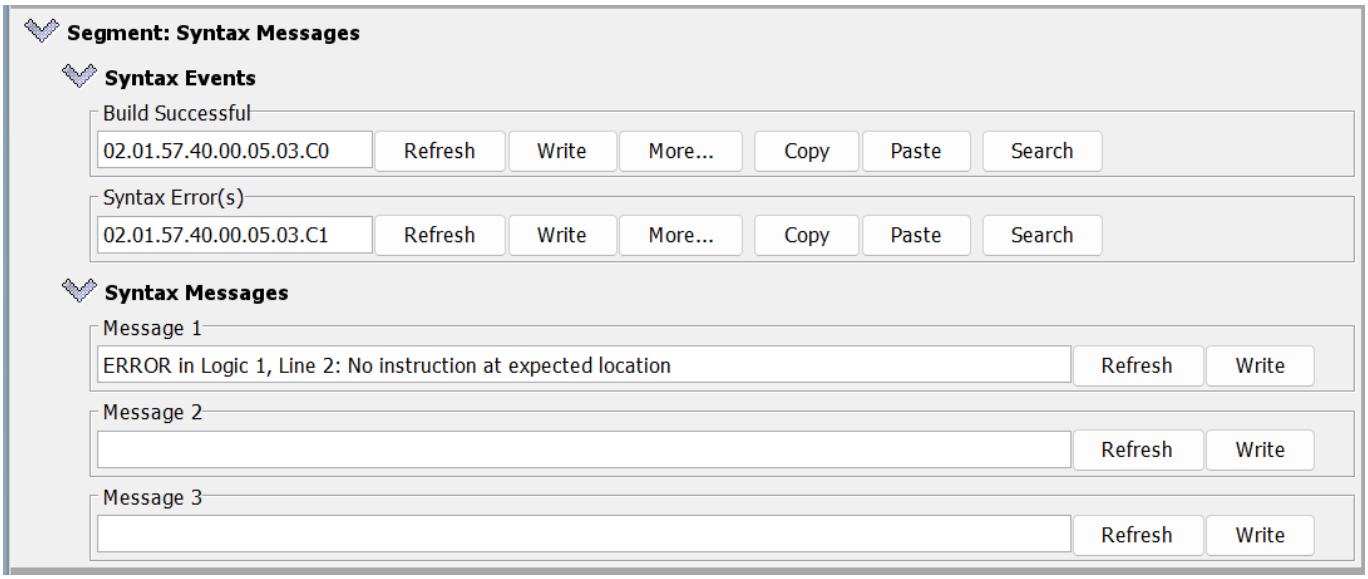
6.1.3 Segment: Conditionals

The various logic operators are entered into the segment **Conditionals**. The logic is broken up into 16 logic groups. Each group contains 4 lines. Each line has room for 63 characters. This is a total of 4,032 characters of text based logic. The only syntax related to the line endings is that the comment operator '//' is automatically closed at the end of each line. Comment operators '/*' and '*/' may continue across lines and groups.



Entering text into a statement line.

In the example above we see a single conditional with a comment in Line 1. In line 2. there is a string of numbers that makes no sense, other than to show the maximum number of characters allowed per line. Once you have completed the compilation cycle (see section 6.2.3) you will see the following error code in the syntax messages.



Example error message

These error messages only check for proper syntax, with no compilation errors. You are responsible for entering the correct logic.

6.2 The Language

The Tower LCC+Q programming language is based on a subset of the IEC 1131 IL (Instruction List) standard as used by Siemens in their STL language. This subset does not include any of the counters, variables (other than Boolean) nor any math functions, (that require variables other than true/false) that are normally included by Siemens in their version of STL.

Like any other PLC (Programmable Logic Controller) programming language, IL languages have both benefits and drawbacks. One of the clearest benefits is program execution speed. (efficiency) As with assembly language in general, instruction lists are a low overhead language and execute faster than graphical languages. Another plus is that ILs also tend to take up less memory. These are both clear benefits, especially in a device that is tight on memory space like an LCC node, and is the primary reason that we have chosen it. The Siemens STL is even more compact than IL in general.

On the downside, STL is not that common of a language mainly because so many people tend to prefer visual programming languages and environments. As a result it is becoming less common for industrial PLC equipment.

6.2.1 The Operator Statement

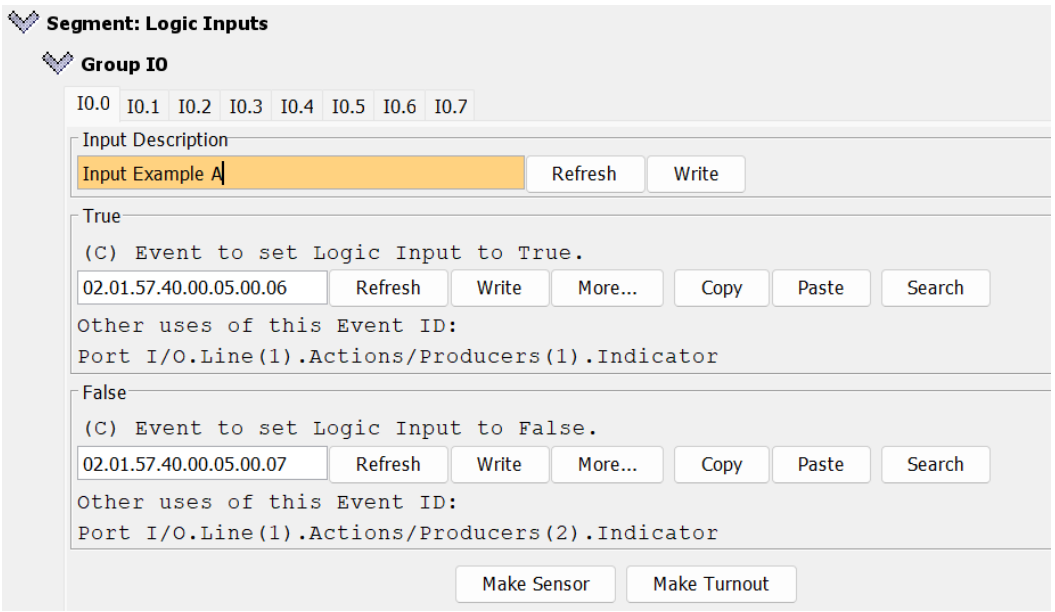
Like many other programming languages, STL logic has a very limited set of allowed operations. These are typically short hand for the logic operations that you need to perform. Some examples are ‘O’ for ‘OR’, ‘A’ for ‘AND’, ‘N’ for ‘Not’, etc.

Each logic statement is started off directly with a logic operation. (A, O, or X) This may seem a bit strange to those familiar with other languages, but this shorthand is done to save the extra ‘Load’ command that would be required if the ‘Load’ was not implied. Also with this format, the statement may be a continuation from a previous statement using the previous RLO. (Result of Logic Operation) In order to accomplish this, the previous RLO must be have been saved in the BR register by using the ‘SAVE’ command before it is terminated.

Each statement is terminated with an ‘=’ (Assign), ‘R’ (Reset), or ‘S’ (Set) command. Unless a ‘SAVE’ command has been done, the logic value (RLO) will have been cleared and the next statement will begin anew.

6.2.2 The Variables

To simplify the logic processing the various logic variables are saved in arrays, or images, in memory. Each variable only requires one bit of storage space in these images. Of course the LCC EventIDs are much larger than this. This compression is possible because each logic entry only has access to the values stored in this node. Any outside references require assigning an EventID pair to each item (bit) in the image. Inputs are controlled by Consumers, and each Output change will Produce matching EventIDs.

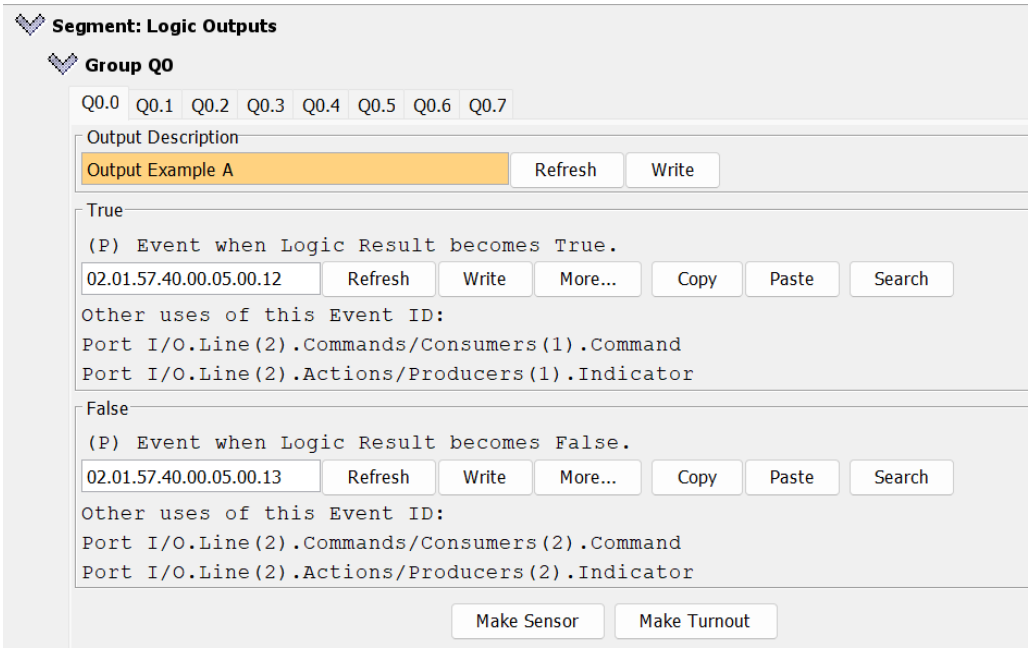


Selecting the EventIDs for Input Variable I0.0

You must assign all required input and output variables to matching locations in the input and output variable tables prior to their use. Unused table entries do not cause any issues. The ‘M’ (Memory) variables are used for temporary (intermediate) storage of results, and have no associated LCC EventIDs, nor create any network traffic.

There are 128 possible input (I) variables and 128 possible output (Q) variables available on the Tower LCC+Q. These are grouped together 8 at a time simply for naming convenience.

On PLC equipment the numbering corresponds to actual input and output lines, but that would be very limiting in our event driven environment where events can be to/from anyplace in the network, not just physical I/O lines on a single node.



Selecting the EventIDs for Output Variable Q0.0

In addition to the I and Q variables, we have also added ‘Y’ and ‘Z’ variables in order to transmit groups of speed information variables between signal masts. (16 input groups, (Y) and 16 output groups. (Z)

6.2.3 The *Compilation cycle*

Before the logic statements you have entered can be used in the logic engine they need to be pre-processed into a form that can be used efficiently. For example you may have entered a jump label that is called “SAVE:”. The instruction “SAVE” is the one that saves the logic value from one statement for use by the next statement. We must be able to distinguish between these. The compile process creates a table of locations in the final logic to act as targets for any jump instructions, then removes their names from the list. This prevents them from being confused with one another.

Another job of the compiler process is to remove all white space from the list, and remove all comments.

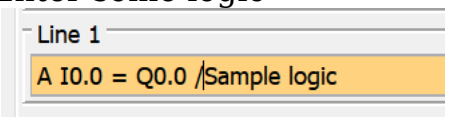
Yet another job is to interpret the timer set commands and change them into actual timing values for use by the timers.

Probably the most useful thing that the Compilation cycle does is to check that you have used the proper syntax for your logic. Obviously it can not tell you if your logic is correct, but it will check that you have entered variables for your operations, that you have closing parens that match your opening ones, etc.

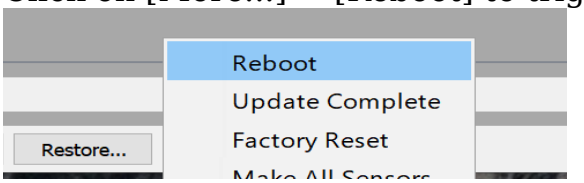
The compile process is run automatically each time the node is booted. The first few errors are written to messages in the Segment: ‘Syntax Messages’. Two EventIDs are also created, one for ‘Build Successful’ and the other for ‘Syntax Error/s’ To read the resulting messages you must first click on [Refresh] for each message.

For example:

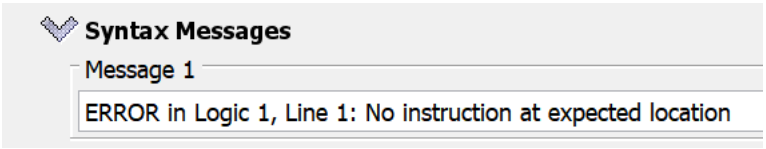
1. Enter some logic



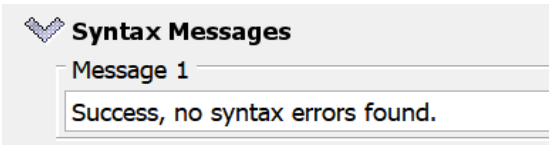
2. Click on [More...] → [Reboot] to trigger the compile process.



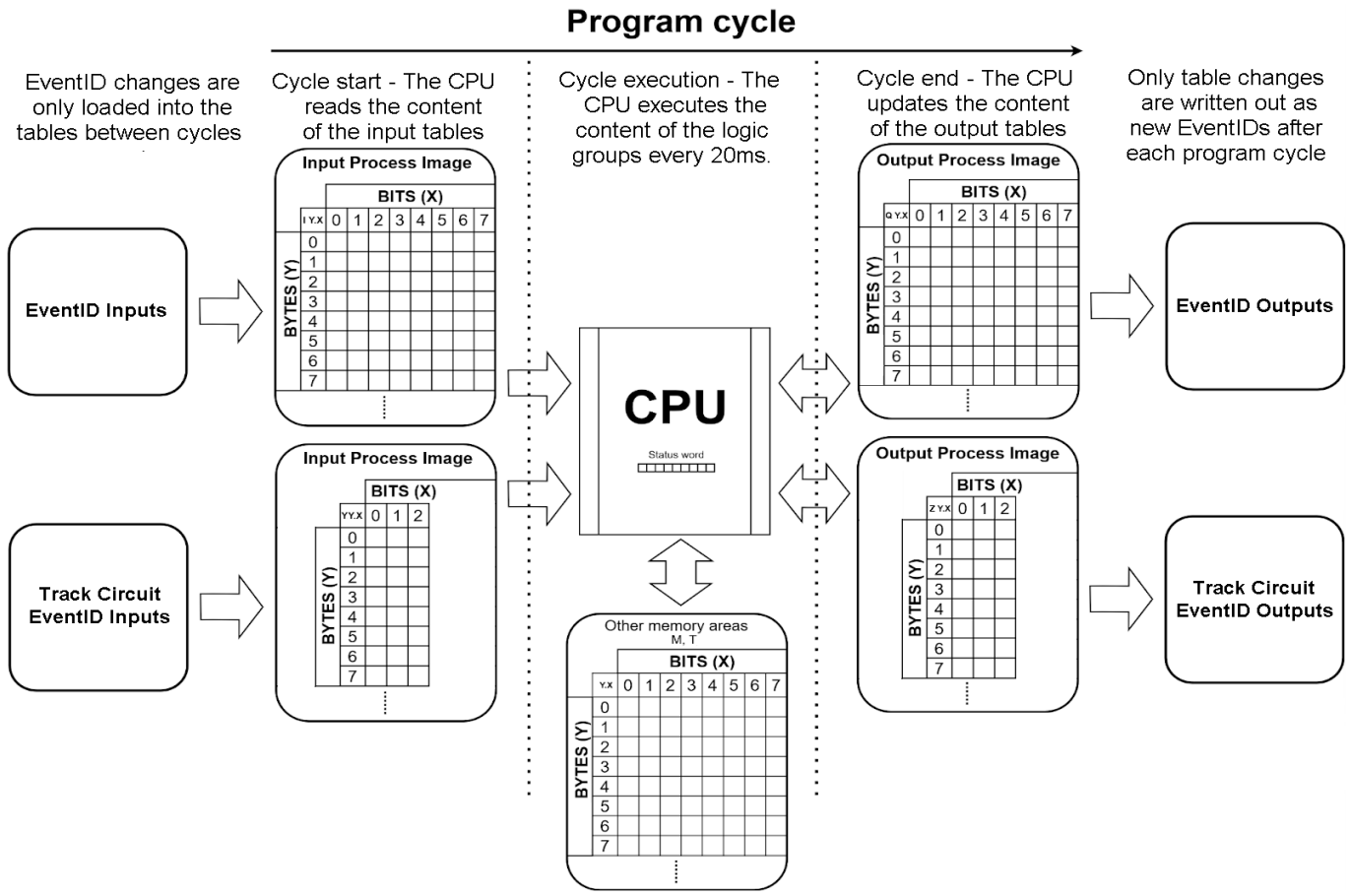
3. Click [Refresh] for Message 1 to see the result.



4. The error is someplace in Logic 1, Line 1. An investigation reveals that the 'Comment' operator needs to be '//' not just a single '/'. Correct this and redo steps 2. - 4. to see the results.



6.2.4 The program cycle



6.2.5 The Status Word

The Status Word is an internal register used to keep track of the state of the instructions as they are being processed. In order to use STL more effectively it is important to understand the Status Word and its functions. However for most basic logic operations you do not need to worry about the status word. It will hide behind the curtain and take care of itself.

Each bit in the Status Word has a specific function to keep track of bit logic (RLO, STA), and whether the logic should continue, be nested or start anew (/FC, OR, BR).

The Most Important Status Word Bits

/FC – Not First Check

If the /FC bit is a 0 then the instruction is considered to be the first instruction being processed in a statement. If the /FC is a 1 then the instruction being scanned will use the logic value from the previous instruction. Certain instructions like =, S and R will set the /FC bit to 0 thus starting a new logic statement after it. Other instructions like A or O will set the /FC bit to 1 signaling to combine the current logic status with the next instruction.

RLO - Result of Logic Operation

The **RLO** bit stores the running logic state of the currently processing instructions.

Certain bit logic and comparison instruction will turn the RLO to a 1 when the condition is TRUE and write a 0 when the condition is FALSE. Other instructions read the RLO (=, S, R) to determine how they are to execute.

STA - Status

The **STA** bit reflects the state of the current Boolean address being processed.

Aditional Status Word Bits

OR

The **OR** bit is used for combining AND functions before OR functions.

BR - Binary Result

The Binary Result transfers the result of the operations (RLO) on to the next instruction for reference. When the **BR** bit is 1 it enables the output of the block to be TRUE and thus allow other blocks after it to be processed. The SAVE, JCB and JNB instructions set the BR bit. This allows you to jump to another location in the logic and continue the logic evaluation from there.

6.3 Bit Logic Instructions

6.3.1 Overview of Bit Logic Instructions

Description

Bit logic instructions work with two digits, 1 and 0. These two digits form the base of a number system called the binary system. The two digits 1 and 0 are called binary digits or bits. In the world of contacts and coils, a 1 indicates activated or energized, and a 0 indicates not activated or not energized. Typical Inputs could be block detectors or fascia contacts. Typical Outputs could be turnout positions or signal aspects.

The bit logic instructions interpret these signal states of 1 and 0 and combine them according to Boolean logic. These combinations produce a result of 1 or 0 that is called the "result of logic operation" (**RLO**), and is usually assigned to an output or stored in a memory location once the logic statement is complete.

These signal states are stored in arrays or process map for easy access by the logic engine. LCC consumers (Inputs) store their resulting state in the Input (I) map area any time they are seen. LCC producers watch the Output (Q) map and send the appropriate EventID any time the output map value changes.

6.3.2 Boolean bit logic operators:

In these descriptions ‘<Bit>’ represents the bit variable in the process map being tested or set by the logic operator. ***‘RLO’*** stands for ***R***esult of ***L***ogic ***O***perator. In other words it is the result of any previous operations, not an operator in the example. `A` (red highlight) shows the operator position in the example. ‘/*’ and ‘*/’ are the open and close comment markers respectively.

A - And

- Format
  A <Bit>

- Description
  A checks whether the state of the addressed bit is "1", and ANDs the test result with the RLO.

- Example
```
RLO /* Previous */ A I1.1 /* AND with Input var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

AN - And Not

- Format
  AN <Bit>

- Description
  AN checks whether the state of the addressed bit is "0", and ANDs the test result with the RLO.

- Example
```
RLO /* Previous */ AN I1.1 /* AND with NOT Input var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

O - Or

- Format
  O <Bit>

- Description
  O checks whether the state of the addressed bit is "1", and ORs the test result with the RLO.

- Example
```
RLO /* Previous */ O I1.1 /* OR with Input var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

ON - Or Not

- Format
  ON <Bit>

- Description
  ON checks whether the state of the addressed bit is "0", and ORs the test result with the RLO.

- Example
```
RLO /* Previous */ ON I1.1 /* OR with Input NOT var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

X - Exclusive Or

- Format
  X <Bit>

- Description
  X checks whether the state of the addressed bit is "1", and XORs the test result with the RLO.

You can also use the Exclusive OR function several times in succession. The final result of the logic operation is "1" if an odd number of checked addresses are "1".

- Example

```
RLO /* Previous */ X I1.1 /* XOR with Input var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

XN - Exclusive Or Not

- Format

  XN <Bit>

- Description

  XN checks whether the state of the addressed bit is "0", and XORs the test result with the RLO.

- Example

```
RLO /* Previous */ XN I1.1 /* XOR with Input NOT var1.1 */ = Q4.0 /* Assign the result to Output var4.0 */
```

6.3.3 Nesting expressions:

A(- And with Nesting Open

- Format

  A(

- Description

  A((AND nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..

- Example

```
RLO /* Previous */ A( O I1.2 O M0.3) /* AND with Group (O Input 1.2 O Memory 0.3) */ = Q4.0 /*Assign the RLO to Output 4.0 */
```

AN(- And with Nesting Open

- Format

  AN(

- Description

  AN((AND NOT nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..

- Example

```
RLO /* Previous */ AN( O I1.2 O M0.3) /* AND with NOT Group (O Input 1.2 O Memory 0.3) */ = Q4.0 /*Assign the RLO to Output 4.0 */
```

O(- Or with Nesting Open

- Format

  O(

- Description

  O((OR nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..

- Example

```
RLO /* Previous */ O( O I1.2 O M0.3) /* OR with Group (O Input 1.2 O Memory 0.3)
*/ = Q4.0 /*Assign the RLO to Output 4.0 */
```

ON(- Or Not with Nesting Open

- Format
  ON(
- Description
  ON((Or nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..
- Example

```
RLO /* Previous */ ON( O I1.2 O M0.3) /* OR with NOT Group (O Input 1.2 O
Memory 0.3) */ = Q4.0 /*Assign the RLO to Output 4.0 */
```

X(- Exclusive Or with Nesting Open

- Format
  X(
- Description
  X((XOR nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..
- Example

```
RLO /* Previous */ X( O I1.2 O M0.3) /* XOR with Group (O Input 1.2 O Memory
0.3) */ = Q4.0 /*Assign the RLO to Output 4.0 */
```

XN(- Exclusive Or Not with Nesting Open

- Format
  XN(
- Description
  XN((XOR NOT nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible..
- Example

```
RLO /* Previous */ XN( O I1.2 O M0.3) /* X OR with Not Group (Input 1.2 Or
Memory 0.3) */ = Q4.0 /* Assign the RLO to Output 4.0 */
```

) - Nesting Closed

- Format
)

Description
) (nesting closed) removes an entry from the nesting stack, restores the OR bit, interconnects the RLO that is contained in the stack entry with the current RLO according to the function code, and assigns the result to the RLO. The OR bit is also included if the function code is "AND" or "AND NOT".
Statements which open parentheses groups:

- **A(** And with Nesting Open
- **AN(** And Not with Nesting Open
- O(Or with Nesting Open

- **ON(** Or Not with Nesting Open
- **X(** Exclusive Or with Nesting Open
- **XN(** Exclusive Or Not with Nesting Open

- Example
```
RLO /* Previous */ ) A M1.1 /* Close Group, AND with Memory 1.1 */ = Q4.0 /*
Assign the RLO to Output 4.0 */
```

6.3.4 String termination:

= - Assign

- Format
  = <Bit>

- Description
  = <Bit> writes the content of the RLO into the addressed bit. You may make multiple assignments of the RLO in the same statement.

- Example
```
RLO /* Previous */ = Q4.0 = Q4.1 = M1.0 */ Assign the RLO to Output 4.0, Output
4.1, and Memory 1.0 */
```

R - Reset

- Format
  R <Bit>

- Description
  R (reset bit) places a "0" in the addressed bit if the RLO = 1.

- Example
```
R M1.1 /* Reset Memory 1.1 to 0. (False) */
```

S - Set

- Format
  S <Bit>

- Description
  S (set bit) places a "1" in the addressed bit if the RLO = 1.

- Example
```
S M1.1 /* Set Memory 1.1 to 1. (True) */
```

6.3.5 Change the Result of Logic Operation (RLO):

NOT - Negate RLO

- Format
  NOT

- Description
  NOT negates the RLO.

- Example
  ```
  NOT /* invert the RLO */ = M1.1 /* Assign RLO to Memory 1.1 */
  ```

SET - Set RLO (=1)

- Format
  SET
- Description
  SET sets the RLO to signal state "1".

Example
```
SET /* set the RLO to "1" (True) */ = M1.1 /* Assign RLO to Memory 1.1 */
```

CLR - Clear RLO (=0)

- Format
  CLR
- Description
  CLR sets the RLO to signal state "0".
- Example
  ```
  CLR /* clear the RLO to "0" (False) */ = M1.1 /* Assign RLO to Memory 1.1 */
  ```

SAVE - Save RLO in BR Register

- Format
  SAVE
- Description
  SAVE saves the RLO into the BR bit. The first check bit /FC is not reset. For this reason, the status of the BR bit is included in the AND logic operation in the next network.

  The use of SAVE and a subsequent query of the BR bit in the same block or in secondary blocks is not recommended because the BR bit can be changed by numerous instructions between the two. It makes sense to use the SAVE instruction before exiting a block because this sets the ENO output (= BR bit) to the value of the RLO bit and you can then add error handling of the block to this.
- Example
  ```
  SAVE /* save the RLO for use in the next statement */
  ```

6.3.6 Edge transition:

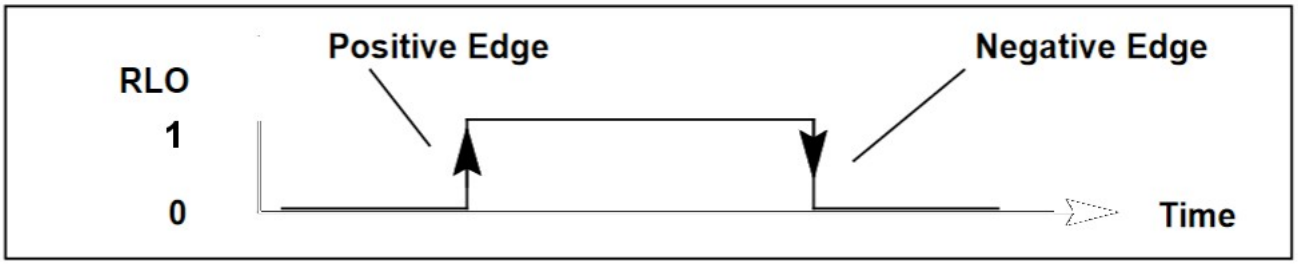
FN - Edge Negative

- Format
  FN <bit>
- Description
  FN <Bit> (Negative RLO edge) detects a falling edge when the RLO

transitions from "1" to "0", and indicates this by RLO = 1.

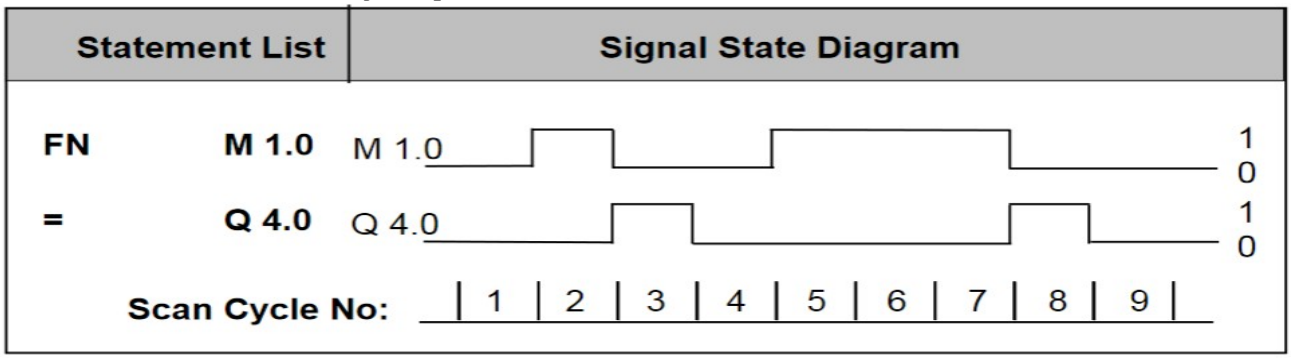
During each program scan cycle, the signal state of the RLO bit is compared with that obtained in the previous cycle to see if there has been a state change. The previous RLO state must be stored in the edge flag address (<Bit>) to make the comparison. Usually a Memory variable is used for this purpose, like M 1.0 in the example below. If there is a difference between current and previous RLO "1" state (detection of falling edge), the RLO bit will be "1" after this instruction

- Definition



- Example
  If the programmable logic controller detects a negative edge at input M 1.0, it energizes the output at Q 4.0 for one program scan cycle. (20mS)

```
FN  M 1.0  /* When a Negative edge is detected at Memory 1.0 */  = Q4.0  /* Output 4.0 is set to true for one logic cycle */
```



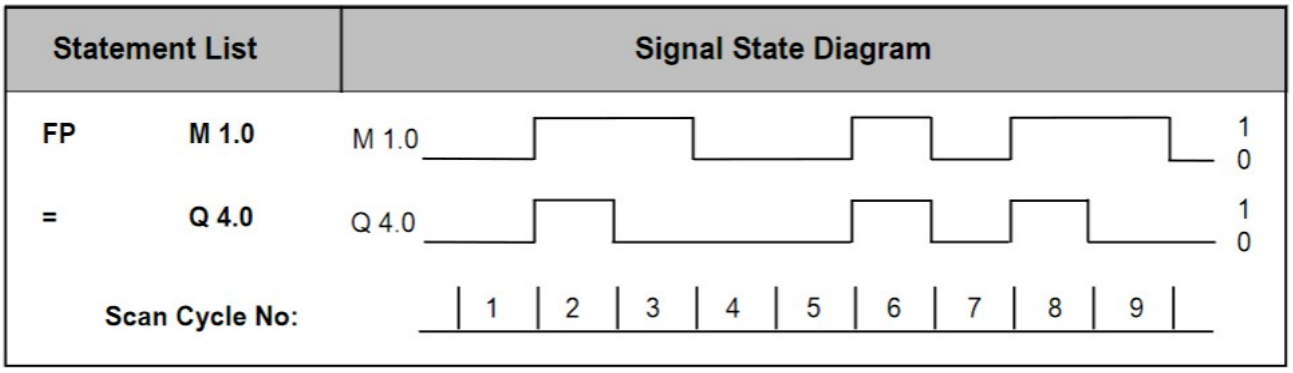
FP - Edge Positive

- Format
  FP <bit>

- Description
  FP <Bit> (Positive RLO edge) detects a rising edge when the RLO transitions from "0" to "1", and indicates this by RLO = 1.

- Example
  If the programmable logic controller detects a positive edge at input I 1.0, it energizes the output at Q 4.0 for one program scan cycle. (20mS)

```
FP  M 1.0   /* When a Positive edge is detected at Memory 1.0 */  = Q4.0  /* Output 4.0 is set to true for one logic cycle */= Q4.0
```




```
FP M1.1 A I 0.1 S Q1.1

// The RLO will go true if Input 0.1 is positive when the positive edge of Memory 1.1
happens. If so, then Output Q 1.1 will be set to true;

// If Input 0.1 is negative, when the positive edge of Memory 1.1 happens, then the RLO will
remain negative. and nothing happens.
```

6.4 Logic Control

6.4.1 Overview of Logic Control Instructions

Description:

You can use the Jump instructions to control the flow of logic, enabling your program to interrupt its linear flow to resume scanning at a different point.

The address of a Jump instruction is a **label**. A jump label may be as many as four characters, and the first character must be a letter. Jump labels are followed with a mandatory colon "**:**" and must precede the program statement in the line being jumped to. (the jump label must not be somewhere in the middle of a logic string) We intentionally restrict any jump instructions to be in a forward direction in the logic to prevent accidental endless looping.

Note:

Please note that the jump destination always forms the **beginning** of a Boolean logic string in the case of jump instructions. The jump destination must not be placed in the middle of a logic string.

6.4.2 The Jump Instructions

You can use the following jump instructions to interrupt the normal flow of your program unconditionally:

JU - Jump Unconditional

- Format
  JU <jump label>

- Description
  JU <jump label> interrupts the linear program scan and jumps to a jump destination, regardless of the status word contents. The linear program scan resumes at the jump destination. The jump destination is specified by a jump label. Only forward jumps are possible by design to prevent looping.

- Example
  ```
  JU EXIT /* Jump Unconditional to EXIT. */ APPR: SET Q 1.1 /* The Intermediate Operations are ignored */ EXIT: /* Operations continue at EXIT: */
  ```

The following jump instructions interrupt the flow of logic in your program based on the result of a logic operation (RLO) produced by the previous instruction statement:

JC - Jump if RLO=1

- Format
  JC <jump label>

- Description
  If the result of logic operation is 1, **JC <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is a specified jump label. Only forward jumps are possible.

  If the result of logic operation is 0, the jump is not executed. The RLO is set to 1, and the program scan continues with the next statement.

- Example
  ```
  RLO /* Previous True */ JC STOP /* If RLO is true Jump Conditional to STOP: */ AN I 1.2 /* If RLO was false continue at */ JC APPR /* Jump Conditional to APPR: */ STOP: SET Q 1.0 JU EXIT /* Set aspect to Stop then Exit */ APPR: SET Q 1.1 EXIT: /* Set aspect to Approach then Exit */
  ```

JCN - Jump if RLO=0

- Format
  JCN <jump label>

- Description
  If the result of logic operation is 0, **JCN <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is specified a jump label. Only forward jumps are possible.

  If the result of logic operation is 1, the jump is not executed. The program scan continues with the next statement.

  Example
  ```
  RLO /* Previous NOT True */ JCN APPR /* Jump Conditional to APPR: */ STOP: /* If previous was true */ SET Q 1.0 /* Set aspect to Stop then Exit */ JU EXIT APPR: SET Q 1.1 /* Set aspect to Approach then continue to Exit */ EXIT:
  ```

Note the change from the previous example where the conditional had to be inverted before checking in order to check for the inverted sense.

JCB - Jump if RLO=1 with BR

- Format
  JCB <jump label>
- Description
  If the result of logic operation is 1, **JCB <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is specified by a jump label. Only forward jumps are possible.

  If the result of logic operation is 0, the jump is not executed. The RLO is set to 1, and the program scan continues with the next statement.

  Independent of the RLO, the RLO is copied into the BR for the **JCB <jump label>** instruction.
- Example
  A I 1.0 A I 1.2 JCB STOP AN I 1.2 JCB APPR STOP: SET Q 1.0 JU EXIT APPR: SET Q 1.1 EXIT:
  AND I 1.0 AND I 1.2 Jump Conditional with BR to STOP:. AND NOT I 1.2 Jump Conditional with BR to APPR:. STOP: Set Q 1.0 Jump Unconditional to EXIT. APPR: Set Q1.1. EXIT:

JNB - Jump if RLO=0 with BR

- Format
  JNB <jump label>
- Description
  If the result of logic operation is 0, **JNB <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is specified a jump label. Only forward jumps are possible.

  If the result of logic operation is 1, the jump is not executed. The RLO is set to 1 and the program scan continues with the next statement.

  Independent of the RLO, the RLO is copied into the BR for the **JNB <jump label>** instruction.
- Example
  A I 1.0 A I 1.2 JC STOP A I 1.2 JNB APPR STOP: SET Q 1.0 JU EXIT APPR: SET Q 1.1 EXIT:
  AND I 1.0 AND I 1.2 Jump Conditional to STOP:. AND I 1.2 Jump Conditional Not with BR to APPR:. STOP: Set Q 1.0 Jump Unconditional to EXIT. APPR: Set Q1.1. EXIT:

  Note the change from the previous example where the conditional had to be inverted before checking in order to check for the inverted sense.

The following jump instructions interrupt the flow of logic in your program based on the signal state of a bit in the status word:

JBI - Jump if BR=1

- Format
  JBI <jump label>

- Description
  If status bit BR is 1, **JBI <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is specified by a jump label. Only forward jumps are possible.

  If the status bit is 0, the jump is not executed. The RLO is set to 1 and the program scan continues with the next statement.

- Example

```
A I 1.0 A I 1.2 JBI OVER AN I 1.2 SET Q 1.0 JU OVER:
```

AND I 1.0 AND I 1.2 Jump if BR =1 to OVER:. AND Not I 1.2 Set Q 1.0 Jump Unconditional to OVER.

JNBI - Jump if BR=0

- Format
  JNBI <jump label>

- Description
  If status bit BR is 0, **JNBI <jump label>** interrupts the linear program scan and jumps to a jump destination. The linear program scan resumes at the jump destination. The jump destination is specified by a jump label. Only forward jumps are possible.

  If the status bit is 1, the jump is not executed. The RLO is set to 1 and the program scan continues with the next statement.

Example

```
A I 1.0 A I 1.2 JNBI OVER AN I 1.2 SET Q 1.0 JU OVER:
```

AND I 1.0 AND I 1.2 Jump if BR =0 to OVER:. AND Not I 1.2 Set Q 1.0 Jump Unconditional to OVER.

6.5 Logic Variables

Variables as we use the term here refer to individual locations in memory that can store one of two values, true or false. (1/0, on/off, normal/reverse, etc.) In computer science this is called Boolean logic. With the exception of Timers (T) we never give variables any value other than zero and one.

6.5.1 Overview of different STL variable types

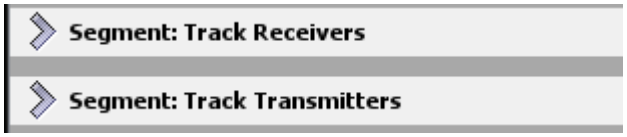
- I – Input (Consumer)
- Q – Output (Producer)
- Y – Track Receiver
- Z – Track Transmitter
- M – Local Memory

- T – Timer

External Input (I) and Output (Q) variables are linked to LCC EventIDs by using two segments of the CDI.

and

In a similar manner Track Circuit Receivers (Y) and Track Circuit Transmitters (Z) are linked to external Event ID groups by using two segments of the CDI. Each link connects together a group of 8 EventIDs, typically variables from one mast sent to the logic of another adjacent mast.



Local (internal) Memory (M) variables may be used as required in your logic. Because there is no link from memories to external objects there is no need to define them in terms of EventIDs in the CDI. If a ‘memory’ item is needed in another node, then use ‘I’ or ‘Q’ which do link to other locations using EventIDs over the network, or else assign them to to a ‘Q’ variable as well.

In like manner Timers (T) are directly created in the logic by defining and using them, but are not visible outside of the logic engine. To use them on the network you must assign them to to a ‘Q’ variable as well to make them visible. Remember that the ‘Network’ includes I/O lines on this node as well as items on other nodes.

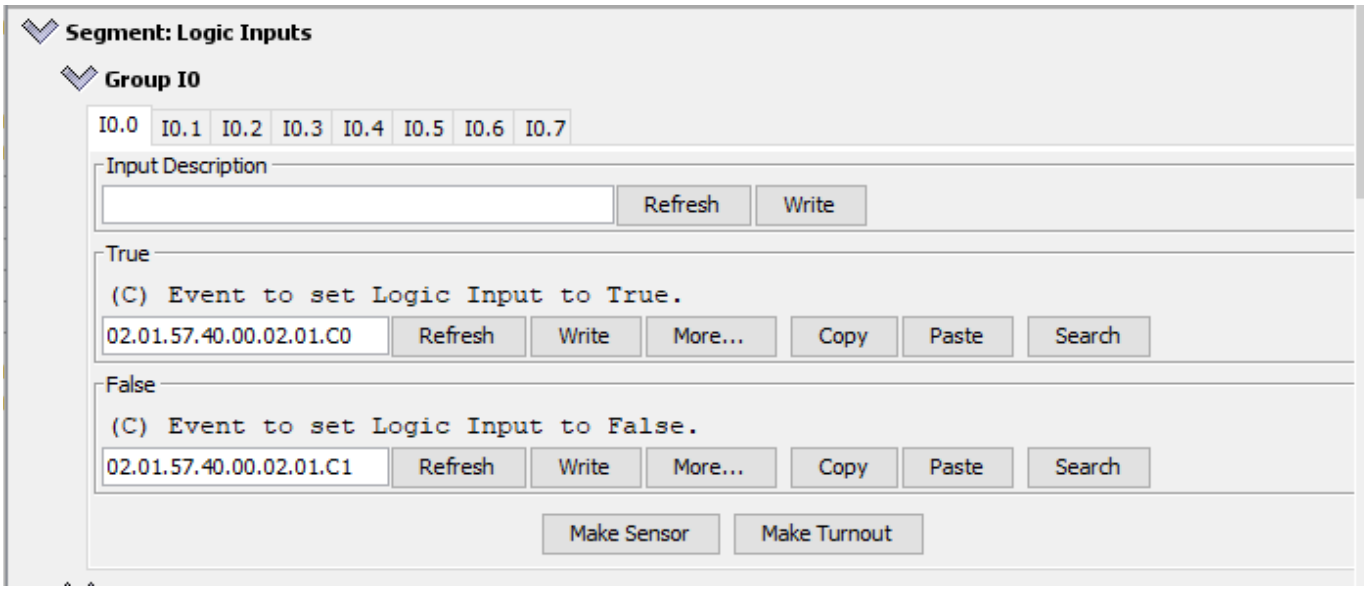
6.5.2 Logic Operation

The logic table itself is re-calculated every 20ms on a continuously repeating basis. However Logic Outputs (producers) only result in EventIDs being sent out if they change from their previous state. This logic re-calculation is not dependent upon, nor is it synchronized with any EventIDs changing. This is an important difference between this logic and the previous logic as used in older versions of our RR-CirKits nodes.

One effect of this is that you are not guaranteed to be able to notice brief changes in external events. If an event changes, then is restored in less than 20ms, then the change may be missed entirely by the logic. In a similar manner any change will not be noticed by the logic for as long as 20ms after it appears on the network. Normally in the scale of time used on our layouts this is not an issue, but it is something to be aware of. For example, if you are calculating train direction by using sensor pairs, then the sensors need to be spaced far enough apart to give longer than 20ms between activation at the highest train speeds allowed.

6.5.3 Segment: Logic Inputs

The logic inputs (LCC Consumers) take the shorthand form of “I0.0” where each input item is identified by a group number followed by an item number. This shorthand form is what is used in the logic statements when calculating logic functions. Block occupancy sensors, push buttons, and turnout position contacts are some examples of inputs. “I” (input) variables are typically used for external inputs, but not necessarily limited to these. Normally you would not assign (=) values to inputs because they are Consumers and will not be written back to the network.



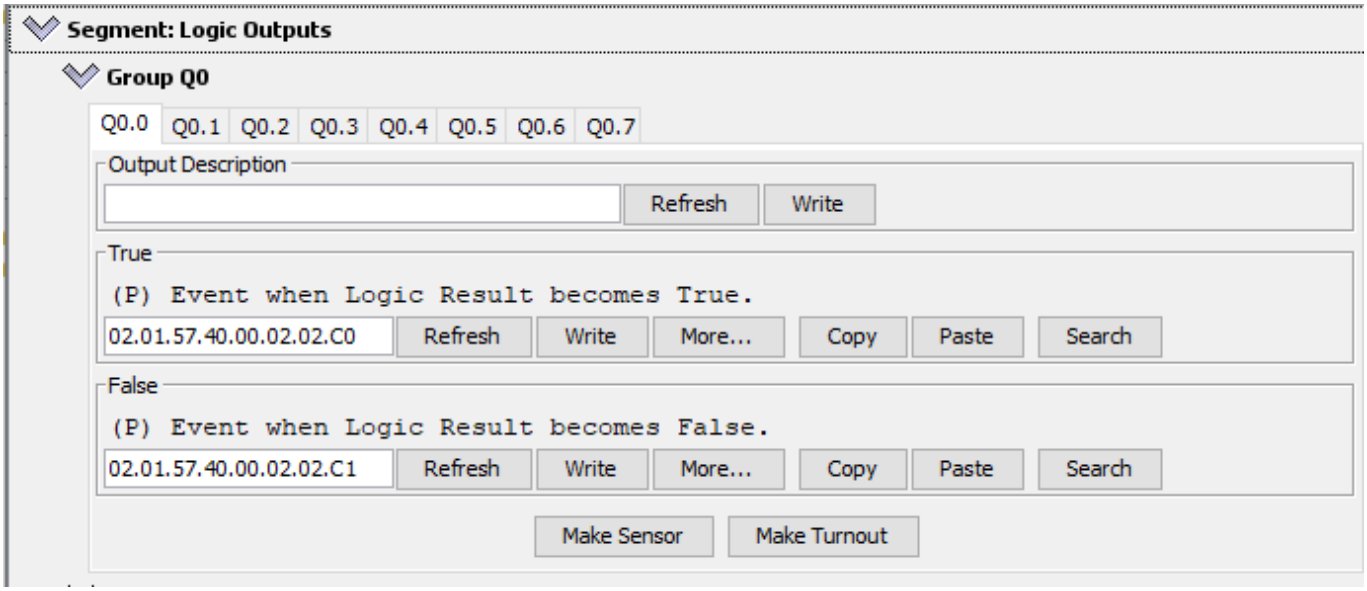
EventIDs into Logic Inputs

Each logic input has two EventIDs associated with it, one to set it ‘true’ and the other to set it ‘false’. It also has the option of assigning a user name, and using the [Make Sensor] or [Make Turnout] to add the item into the JMRI tables.

External actions as seen on the LCC bus can change the states of these items. The resulting values are stored in the node for use by the logic.

6.5.4 Segment: Logic Outputs

The logic outputs (LCC Producers) take the shorthand form of “Q0.0” where each output is identified by a group number followed by an item number. This shorthand form is what is used in the logic statements as outputs from calculating logic functions. Signal aspects, turnout positions, and relay controls are typical outputs that you may have. “Q” (output) variables are typically used to control external devices, but are not necessarily limited to these.



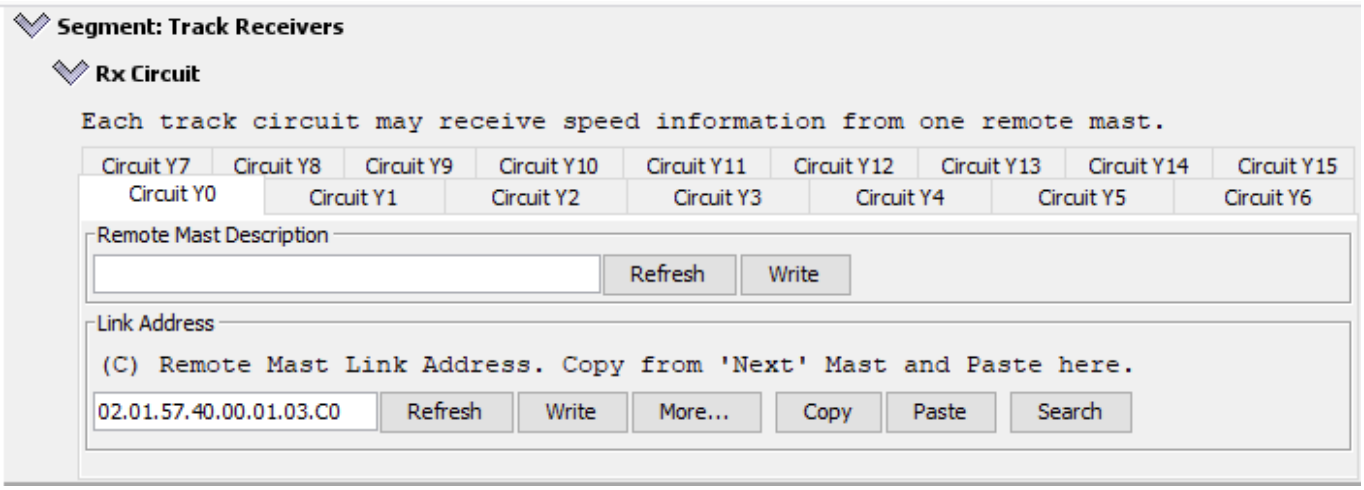
Logic Outputs into EventIDs

Each logic output has two EventIDs associated with it, one is produced when it is set to ‘true’ and the other when it is set to ‘false’. It also has the option of assigning a user name, and using the [Make Sensor] or [Make Turnout] to add the item into the JMRI tables.

When ever these items change state as a result of logic operations, then the changes are sent to the LCC bus in the form of EventIDs that may change the state of other items on the layout. These resulting values are also stored in the node and may be used by the logic for further calculations. For example if some logic calculates a turnout position (Q0.0) it will be sent out to the network to move the turnout. That same item (Q0.0) may also be checked by another logic section in the same (or other) node that is calculating a signal aspect.

6.5.5 Segment: Track Receiver

The Track Receiver (LCC Consumers) take the shorthand form of “Y0.0” where each Track Circuit takes the form of an 8 item group connecting one signal mast to another. The first digit is the track “Circuit” receiver number and the second digit is the associated track speed number. When using these values within a logic statement you manually add the “.0”, etc. to indicate the desired speed. The shorthand example above “Y0.0” means track circuit “Receiver Zero is Stop”. (see below for suggested speed values)



Track Circuit Input Link EventID

TABLE 5-1 CODE RATE AND ASPECT

CODE RATE	ASPECT
7	Clear
4	Advance Approach
3	Approach Limited
8	Approach Medium
2	Approach
9	Approach Slow
6	Accelerated Tumble Down
5	Non-Vital code indicating track occupancy, or a hand-throw switch in the block out of normal correspondence
M	Non-Vital code indicating power off in the block, or a lamp out condition in the block. Power Off will indicate from the east end CP, lamp out from the west end CP

To the left we see the prototype speed associations. For simplicity and to fit into our available numbering system we can use:

0. Stop (*not in the prototype set)
1. Approach Medium
2. Approach
3. Approach Limited
4. Advance Approach
5. Approach Slow
6. Accelerated Tumble Down
7. Clear

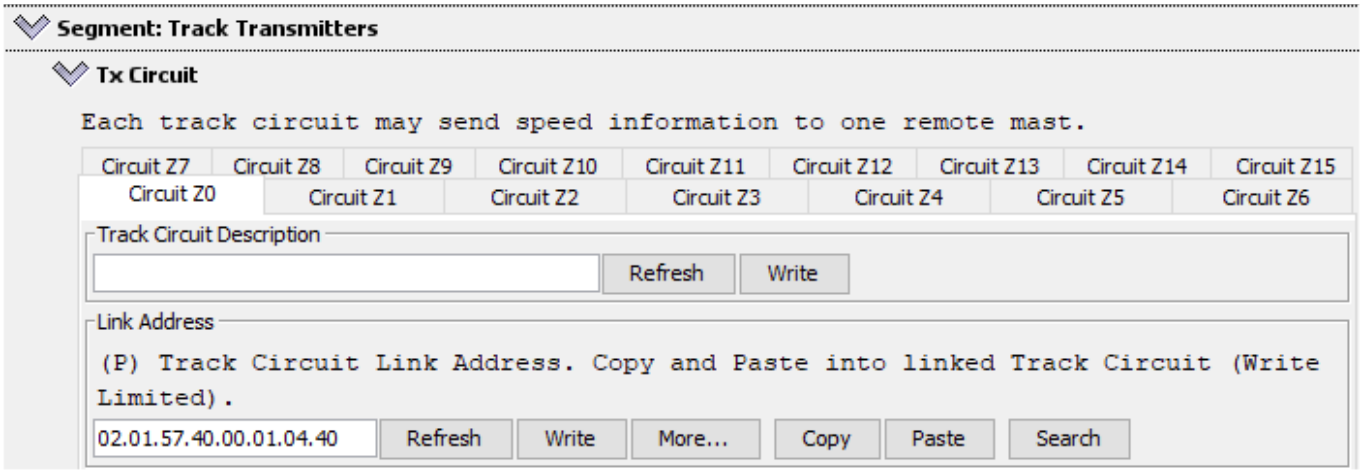
*Because the rails are shorted by the train and no messages can be received.

This suggested numbering and their associated speed values are arbitrary, and you may use any order and values that you need. (for any purpose)

On the prototype, ‘track circuits’ are a way to send speed information from one signal mast to another over the rails themselves. For our purposes track circuits are a shorthand way to link speed data from one mast to another by setting a single EventID value. Because track circuits primarily send speed values they operate as a group, with only one value being true at a time. Setting any speed to true using its Track circuit Transmitter number will automatically set all other speed values for the same track circuit to false. The prototype has no “Stop” speed value because if the block is occupied (Stop) then the circuit is shorted out and thereby disabled. We needed a value that corresponds to this and have added “Stop” to the set. The actual speed values for each aspect name are a function of your railroad’s rule book.

6.5.6 Segment: Track Transmitter

The Track Transmitter (LCC Producers) take the shorthand form of “Z0.0” where each Track Circuit takes the form of an 8 item group connecting one signal mast to another. The first digit is the track circuit transmitter number and the second digit is the associated track speed number to be sent. When using these values within a logic statement you simply add the “.0”, etc. to indicate the desired speed being sent. The shorthand example above “Z0.0” means track circuit “Transmitter Zero is Stop”.



Track Circuit Output Link EventID

The “Link Address” value shown for each track circuit transmitter is a read only value. Use [Copy] to grab this value and then [Paste] to place the value into the track circuit receiver that is being used by the previous mast. If you assign user names to each circuit and use the “Sensor/Turnout creation” tool to save them in JMRI, then you can use the search tool when creating these links.

6.5.7 Memory Variables

The STL logic includes local memory variables to allow information to be saved and/or exchanged between different logic items. The only values saved are ‘True’ or ‘False’. These local memory variables take the shorthand form of “M0.0” where each variable is simply entered as required. You will need to keep track of these by yourself. We support 128 memory variables labeled from “M0.0” to “M15.7” By the term “Local Variable” it means that any information being stored is only accessible to the logic engine itself. This information is not visible outside of the node. It can only be assigned and/or evaluated by the logic within a single node. (Actually any shorthand entry is only visible to the logic within a single node.) In order for a shorthand entry to be seen outside of the logic engine itself, you must create a “Logic Output” (Q) entry to create matching EventID entries in order to place the information onto the LCC Bus. This is even true when referencing the I/O ports on the Tower LCC+Q node itself. All of the LCC EventIDs are outside of the logic engine itself.

6.5.8 Timer Variables

Our version of STL logic includes 64 timers to create logic delays for various purposes. Timer labels take the form of “T0” to T63”. Note that flashing outputs or pulses are created by the I/O lines themselves and do not rely on these logic timers for their operation.

Load value. You must pre-load a value for use by a timer using the **L** instruction immediately prior to starting it. ***W#t#xyz*** is the format of the value where ‘W’ is a 16 bit Word, formed of '#t' = the time base (that is, the time interval or resolution) and '#xyz' = the time value in decimal format. For example, ***L W#0#10*** = 100MS. The ‘L<value>’ always precedes a ‘Start Timer’ instruction. (SD, SE, SF, SP, SS) The following timer instructions relate to timers:

Overview of Timer Instructions

Memory and components of a Timer.

The following timer instructions are available:

- **FR** Enable Timer (Free)
- **R** Reset Timer
- **SD** On-Delay Timer
- **SE** Extended Pulse Timer
- **SF** Off-Delay Timer
- **SP** Pulse Timer
- **SS** Retentive On-Delay Time

FR Enable Timer (Free)

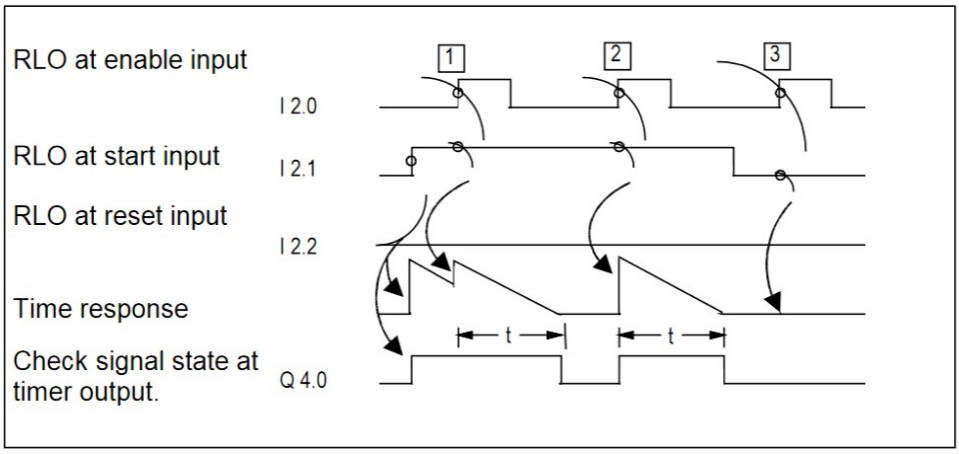
- Format
  FR <timer>
- Description
  When the RLO transitions from "0" to "1", **FR <timer>** clears the edge-detecting flag that is used for starting the addressed timer. A change in the RLO bit from 0 to 1 in front of an enable instruction (FR) enables a timer.

  Note: FR (Timer enable) is not required to start a timer, nor is it required for normal timer instruction. An enable is used only to re-trigger a running timer, that is, to restart a timer. The restarting is possible only when the start instruction continues to be processed with RLO = 1.

Example

```
A I 2.0 FR T1 A I 2.1 L W#2#10 SP T1 A I 2.2 R T1 A T1 = Q 4.0
```

AND I 2.0 Enable T1, And I 2.1 Load 10 seconds Start Pulse T1 AND I 2.2 Reset Load timer into Q 4.0



Timer Enable trigger description ***t*** *= programmed time interval*

(1) A change in the RLO from 0 to 1 at the enable input while the timer is running completely restarts the timer. The programmed time is used as the current time for the restart. A change in the RLO from 1 to 0 at the enable input has no effect.

(2) If the RLO changes from 0 to 1 at the enable input while the timer is not running and there is still an RLO of 1 at the start input, the timer will also be started as a pulse with the time programmed.

(3) A change in the RLO from 0 to 1 at the enable input while there is still an RLO of '0' at the start input has no effect on the timer.

R Reset Timer

- Format
  R <timer>

- Description
  R <timer> stops the current timing function and clears the timer value and the time base of the addressed timer word if the RLO transitions from 0 to 1.

Example
```
RLO  /* Previous RLO True */ R T1
```
Check the signal state of previous. If RLO transitioned from 0 to 1, then Reset timer T1.

SP Pulse Timer

- Format
  SP <timer>

- Description
  SP <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time elapses as long as RLO = 1. The timer is stopped if RLO transitions to "0" before the programmed time interval has expired.

  This timer start command expects the time value and the time base to be loaded prior to starting the timer.

  Example
  ```
  RLO  /* Previous RLO True */ L W#2#10 SP T1 A T1 = Q 4.0
  ```
  Load 10 seconds then Start Pulse T1 AND T1 Load timer status into Q 4.0

SE Extended Pulse Timer

- Format
  SE <timer>

- Description
  SE <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time interval elapses, even if the RLO transitions to "0" in the meantime. The programmed time interval is started again if RLO transitions from "0" to "1" before the programmed time has expired.

  This timer start command expects the time value and the time base to be loaded prior to starting the timer.

  Example
  RLO `/* Previous RLO True */` **`W#2#10 SE T1 A T1 = Q 4.0`**
  Load 10 seconds then Start Extended Pulse T1 AND T1 Load timer status into Q 4.0

SD On-Delay Timer

- Format
  SD <timer>

- Description
  SD <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time interval elapses as long as RLO = 1. The time is stopped if RLO transitions to "0" before the programmed time interval has expired.

  This timer start command expects the time value and the time base to be loaded prior to starting the timer.

  Example
  RLO `/* Previous RLO True */` **`L W#2#10 SD T1 A T1 = Q 4.0`**
  Load 10 seconds then Start On-Delay T1 AND T1 Load timer status into Q 4.0

SS Retentive On-Delay Timer

- Format
  SS <timer>

- Description
  SS <timer> (start timer as a retentive ON timer) starts the addressed timer when the RLO transitions from "0" to "1". The full programmed time interval elapses, even if the RLO transitions to "0" in the meantime. The programmed time interval is re-triggered (started again) if RLO transitions from "0" to "1" before the programmed time has expired.

  This timer start command expects the time value and the time base to be loaded prior to starting the timer.

  Example
  RLO `/* Previous RLO True */` **`L W#2#10 SS T1 A T1 = Q 4.0`**
  Load 10 seconds then Start Retriggerable On-Delay T1 AND T1 Load timer status into Q 4.0

SF Off-Delay Timer

- Format

SF <timer>

- Description
  SF <timer> starts the addressed timer when the RLO transitions from "1" to "0". The programmed time elapses as long as RLO = 0. The time is stopped if RLO transitions to "1" before the programmed time interval has expired.

  This timer start command expects the time value and the time base to be loaded prior to starting the timer.

  Example
  ```
  RLO  /* Previous RLO True */  L W#2#10 SF T1 A T1 = Q 4.0
  ```
  Load 10 seconds then Start Off-Delay T1 AND T1 Load timer status into Q 4.0

6.5.9 Timer details.

Format of the **L** <value> instruction **`L W #t #xyz`**

L W (**L**oad **W**ord) = Loads the following values into a 16bit register.

#t = The time base defines the interval at which the time value is decremented by one unit. Only one of the following 4 single digit values is allowed.

Time Base #t numeric values for intervals

Time base	Interval	Resulting Timer Range
0	10 MS	10MS to 9S 990MS
1	100 MS	100MS to 1M 39S 900MS
2	1 S	1S to 16M 39S
3	10 S	10S to 2H 46M 30S

#xyz = The multiplier determines the desired number of intervals during the time setting. Each of the x, y, and z entries represents a single digit. Leading zeros are ignored. The minimum value is 1, and maximum value is 999.

Example time value settings:

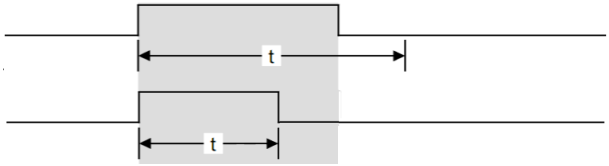
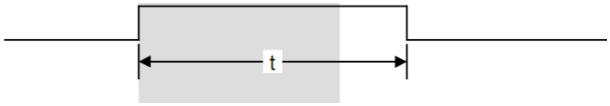
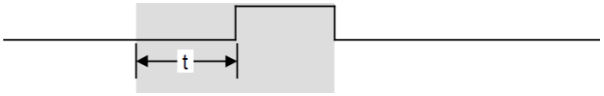
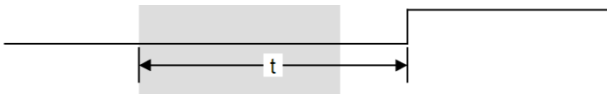
`L W#0#500` = 5 seconds (10MS x 500)
`L W#1#50` = 5 seconds (100MS x 50)
`L W#2#5` = 5 seconds (1 second x 5)
`L W#3#5` = 50 seconds (10 seconds x 5)

For even seconds use 1S or 10S intervals because they take up less statement space. For fractional seconds use a smaller interval such as 100MS. E.g. **`LW#1#45`** will give you a 4.5 second timer. For longer timers convert the desired time into 1 second or 10 second intervals. Example: 8 minutes equals 480 one second intervals, or 48 ten second intervals. The time entry for 8 minutes could therefore be written as either **`LW#2#480`** or **`LW#3#48`**. It may not be written as **`LW#1#4800`** nor as **`LW#0#48000`** because #xyz may not exceed 999 intervals.

Choosing the right Timer
Following the loading of the desired timing value you must immediately **S**tart the

type of timer you need.

This overview is intended to help you choose the right timer for your timing job. The available Start types are:

Pulse Type	Visual Description	Description
Trigger value I 1.0		Duration ***t*** of trigger signal I 1.0 is shown with gray background.
SP Pulse Timer		The time that the output signal remains at 1 is the programmed time value ***t***, unless the trigger signal returns to 0 before the end of time ***t***. In that case the output signal immediately returns to 0.
SE Extended pulse timer		The output signal remains at 1 for the programmed length of time ***t***, regardless of how long the input trigger signal stays at 1.
SD On-delay timer		The output signal changes to 1 only when the programmed time ***t*** has elapsed and the input trigger signal is still 1. If the input signal is shorter than time ***t***, then no pulse will occur.
SS Retentive on-delay timer		The output signal changes from 0 to 1 only after the programmed time ***t*** has elapsed, regardless of how long the input signal trigger stays at 1.
SF Off-delay timer		The output signal changes to 1 when the input trigger signal changes to 1 or while the timer is running. The time ***t*** is started when the input trigger signal returns from 1 to 0.

The start format is **SD T3** where the timer type is followed by the timer number. The Start timer instruction loads the previously loaded <value> into the timer and starts it running.

Example statement for starting an 8 minute elapse timer using Timer 3 with the trigger of **I1.0**:

```
A I1.0 L W#2#480 SE T3
```

Example statement for checking the timer status and placing it in Q4.0:

```
A T3 = Q4.0
```

Adding the **FR** 'Timer Reset' instruction e.g. `A I1.5 FR T3` anywhere in the logic string will reload timer T3 with its preset value of W#2#480 when Input 1.5 transitions from "0" to "1", presuming that timer T3 is still running, and Input 1.0 is still at "1", otherwise it will have no effect.

6.6 STL Logic Operators

6.6.1 Supported instruction Mnemonics

Mnemonic	Element type	Description
L: La: Lab: Labl: L1: etc.	Jumps	The address of a Jump instruction is a label. A jump label may be as many as four characters, is case sensitive, and the first character must be a letter. Jumps labels are followed with a mandatory colon ":" and must precede the program statement in a line. Note: the jump destination always forms the **beginning** of a Boolean logic string in the case of jump instructions (the jump label must not be somewhere in the middle of a logic string).
=	Bit logic Instruction	Assign
//	Format	Open Line Comment - The '**//**' comment is automatically closed at the end of its statement line.
/*	Format	Open Comment - The '**/***' comment is not automatically closed at the end of its statement line. It must always be closed by using the '***/**'.
*/	Format	Close Comment - The '***/**' comment may be used with '**/***' to hide a single operator or multiple statement lines.
)	Bit logic Instruction	Nesting Closed
A	Bit logic Instruction	And
A(	Bit logic Instruction	And with Nesting Open
AN	Bit logic Instruction	And Not
AN(	Bit logic Instruction	And Not with Nesting Open
CLR	Bit logic Instruction	Clear RLO (=0)
FN	Bit logic Instruction	Edge Negative
FP	Bit logic Instruction	Edge Positive
FR	Timers	Enable Timer (Free)
JBI	Jumps	Jump if BR = 1
JC	Jumps	Jump if RLO = 1
JCB	Jumps	Jump if RLO = 1 with BR
JCN	Jumps	Jump if RLO = 0
JNB	Jumps	Jump if RLO = 0 with BR
JNBI	Jumps	Jump if BR = 0
JU	Jumps	Jump Unconditional
L <value>	Value	Load value. You can pre-load a value for use by a timer using the **L** instruction. **W#t#xyz** is the format of the value where 'W' is a 16 bit Word, formed of '#t' = the time base (that is, the time interval or resolution) and '#xyz' = the time value in decimal format. For example, L W#0#10 = 100MS. The 'L<value>' always precedes a Start Timer instruction.
NOT	Bit logic Instruction	Negate RLO
O	Bit logic Instruction	Or
O(	Bit logic Instruction	Or with Nesting Open
ON	Bit logic Instruction	Or Not
ON(	Bit logic Instruction	Or Not with Nesting Open
R	Bit logic / Timer	Reset
S	Bit logic Instruction	Set
SAVE	Bit logic Instruction	Save RLO in BR Register
SD	Start Timer	On-**D**elay Timer
SE	Start Timer	**E**xtended Pulse Timer
SET	Bit logic Instruction	Set
SF	Start Timer	O**ff**-Delay Timer
SP	Start Timer	**P**ulse Timer
SS	Start Timer	Retentive On-Delay Timer

X	Bit logic Instruction	Exclusive Or
X(	Bit logic Instruction	Exclusive Or with Nesting Open
XN	Bit logic Instruction	Exclusive Or Not
XN(	Bit logic Instruction	Exclusive Or Not with Nesting Open

7 Track Circuits

The prototype railroads have the need to send signal indication information from one signal to the next in order to calculate the proper aspects to display. These calculations are done locally, not at the dispatcher's office, nor by some central computer. This indication information can be sent over local wires from one signal to the next, but that means lots of infrastructure to maintain. Fortunately there is always one pair of conductors that already need power on them for detection circuits and that automatically go to the right place. That pair of conductors is the rails themselves. Even from the earliest semaphore days some basic indication information was passed over the rails simply by switching the polarity of the battery feeding the DC track circuit being fed from one end of a block to the other.

As signaling became more complex and speed signaling was introduced, there was a need to pass more information over the same two rails. Genrakode® and Electrocode® are two brands of equipment that do this by sending a series of pulses rather than just the DC polarity from one end of the block to the other. These circuits are normally bi-directional, so the transmit and receive circuits switch from one direction to the other every few seconds.

We are not actually using the rails, so we do not have that limitation, and can send information immediately in both directions.

7.1 Simulating a Code Line with Events

To send this indication information from signal to signal over the usual EventIDs is difficult because each link needs to be specifically made using different EventID numbers for each indication. That makes it difficult or impossible to determine ahead of time what all the EventID numbers will be required, especially for a modular setup.

One solution to this dilemma is to use events in a way similar to those used in coded track circuits and then allow nodes to learn these groups of internally generated events simply by using the CDI tools to change a few entries.

These internal event groups are then translated into normal user programmable EventIDs for use in Logic and connections with other nodes and panels. To link up these connections at a modular meet the operators would simply link the signal nodes at each side of a module boundary to create these virtual groups of links automatically.

7.2 Linking Virtual Code Lines

To link these virtual code circuits use the CDI to copy the key from one track circuit to another. The EventID for the TX code set will always come from the transmitting node and be entered into the receiving node to avoid accidental reuse of EventID numbers from show to show.

Each node can setup one or more virtual code lines to any other node. For simplicity these virtual links are named 'Track 1', ‘Track 2’, etc. It is incumbent upon the user to keep track of which 'Coded' virtual links are created between nodes. Be sure to record which block (1-8) is used for each side of the virtual links if you have not standardized these connections. There is no need to use the same 'block' number on both sides of any virtual coded track circuits, and in fact they will not normally be matching. Normally these virtual links will follow along with the rails, but there is no actual requirement that they do so.

7.3 Prototype Code Line

In our solution to this problem we follow the prototype with a few changes. When each prototype code is sent onto the rails there are four data items simultaneously sent in each packet. They are the ‘Start’ bit, (1) the 'Speed' or 'Indication' code, (2, 3, 4, 6, 7, 8, or 9), the '5' code, (5) and the 'M' code. (M) In order to make this solution more 'EventID' friendly we represent each code with two events, one to turn it on and the other to turn it off. This allows us to ignore the ‘Start’ bit and send any changes immediately.

TABLE 5-1 CODE RATE AND ASPECT

CODE RATE	ASPECT
7	Clear
4	Advance Approach
3	Approach Limited
8	Approach Medium
2	Approach
9	Approach Slow
6	Accelerated Tumble Down
5	Non-Vital code indicating track occupancy, or a hand-throw switch in the block out of normal correspondence
M	Non-Vital code indicating power off in the block, or a lamp out condition in the block. Power Off will indicate from the east end CP, lamp out from the west end CP

Example Prototype Code Line Values

Note: the prototype can only send indications via an unoccupied block, because the train itself is shorting the rails going toward the signal ahead when it is occupied. On the model we do not have that restriction.

We can send either ‘Stop’ (5) or ‘Tumble down’ (6) as an indication. This simplifies the coding of APB signals.

To limit the number of indication EventIDs to 16 we have omitted the ‘M’ code as being unnecessary.

8 ABS and APB Signal plus other examples

For examples of using logic to control signals see the Signal LCC manual. The Tower LCC+Q logic may be used to expand a Signal LCC logic table if more statements are required. To be determined.

9 Tower LCC+Q compatible Input/Output Cards

The RR-CirKits Tower LCC+Q and its compatible I/O modules are designed to be clipped into Tyco 3-1/4" Snap-Track® mounted to the bench work. (Snap-Track® is a plastic channel designed to mount PC cards to a chassis, not something to run trains on.)

A single Tower LCC+Q or compatible I/O module fits into the 3TK2-1 (single) mounting track. Other widths are available for compact installations using multiple boards.

Each I/O module is equipped with two connectors to facilitate these I/O board connections. Use IDC connectors and ribbon cables to connect the Tower LCC+Q to the I/O cards.

9.1 BOD-4 (DCC Block Occupancy Detector - 4 block plus 4 I/O)

This board operates as a DCC block occupancy detector for 4 blocks using remote CT coils. It outputs logic levels, and has a RR-CirKits standard ribbon connector interface. The "Power-Lok" feature optionally monitors the DCC bus power. A power failure latches the detection status of each block until power is restored and re-stabilized. There are also 4 general purpose I/O connections fed through to the driver board.

9.2 BOD4-CP (DCC BOD 4 block, 4 Inputs, plus 2 turnout drivers)

This board operates as a DCC occupancy detector for 4 blocks using remote CT coils. It outputs logic levels, and has a RR-CirKits standard ribbon connector interface. The "Power-Lok" feature optionally monitors the DCC bus power. A power failure latches the detection status of each block until power is restored and re-stabilized. The CP version also includes dual turnout drivers. When used with the Tower LCC+Q or Signal LCC boards there are also 4 general purpose I/O connections available using the Sample options.

9.3 BOD-8 (DCC Block Occupancy Detector - 8 block)

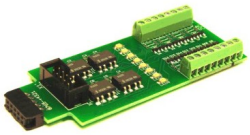
The BOD-8 does not expect you to re-wire your layout to bring track feeders to the detector cards. The small CT (Current Transformer) detection coils are placed directly on the track feeders where they belong. Simple lengths of Cat-5 cable are the usual way to run the signals back to the detector boards. Use of CT coils means that there are no track voltage losses associated with the detectors. Normal detection levels are 1mA. but may be adjusted to higher levels with on board pots.

During a DCC bus power failure the Power-Lok input on the BOD-8 instantly locks the current state of each block detector. I.e. the state of the layout does NOT change during a DCC power outage, neither to all occupied, nor to all vacant. It just suspends sending any occupancy changes until after power is restored and things have stabilized again. If you do not want the feature there is a jumper to disable it.

The BOD-8 outputs are low during detection so the Tower LCC+Q should be configured accordingly.

It is planned to build a ‘Detector LCC’ board that will combine the LCC interface and a detector card.

9.4 OIB-8 (Opto Isolator Board - 8 input)



This 8 input board is used when a non-isolated source of voltage needs to be monitored and input to the Tower LCC. One example would be to monitor the DCC voltage on a set of points to determine the position of a turnout without using auxiliary contacts.

This board may be configured to monitor the absence or presence of an AC or DC signal. This board requires 10mA. for reliable operation and includes built in current limiters.

9.5 SCSD-8 (Single Coil Solenoid Driver)

The SCSD-8 Output Module is designed to drive individual solenoid coils or other high voltage high power devices. Normally the input voltage should not exceed 27VDC. The SCSD-8 board is optically isolated from the driving circuitry to protect the Tower LCC+Q or other control device from the high power outputs. When driving single coils or high power loads configure the line as a steady output.

By using the proper options on the Tower LCC+Q the SCSD-8 may also be used to control dual coil momentary switch machines. In 'Dual Coil' mode the output lines must be paired such that the pair of lines requires just single address pair. However reverse the two EventIDs. This action will normally require a 0.1 second pulse when driving solenoids.

The lines are paired and only the primary event of the first line of each pair will be used to trigger a pulse.

Dual coil operation should not be attempted if the switch machine power supply is not of the capacitive discharge type that will limit the long term current to a low value in case of hardware or configuration errors.

Failure to observe this precaution may result in destruction of equipment and be a fire hazard!

9.6 SMD-8 (Stall Motor Driver – 8 line)

The SMD-8 board contains 8 individual, optically isolated, H-Bridge drivers. This allows the board to be powered from any supply between 9 Volts and 27 Volts. It is primarily designed to drive stall motor turnout machines such as those found in Tortoise® and Switchcraft® machines . Do not exceed 20VAC or 27VDC at the power input.

This board includes an adjustable buck switching regulator to allow you to control the speed of your switch machine motors. This regulator can not boost the drive voltage above the supply voltage.

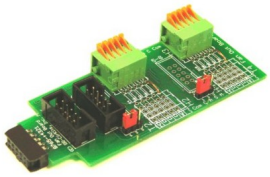
9.7 RB-4 (Relay Board - 4 x SPDT)

Relay Board - 4 is a Quad 10A SPDT relay board with logic level drivers. It is suitable for use with Tower LCC+Q or other logic level output devices. It requires 12V auxiliary power to drive the relay coils. Auxiliary power is optically isolated from the logic inputs for double isolation. LED indicators for each relay make it easy to monitor activity.

Includes dual ribbon connectors with offset lines to allow easy connection as output 1-4, or output 5-8, of the Tower LCC, or other driver.

The RB-4 input lines are active low so all lines on this Tower LCC+Q port should be configured appropriately. This inverted input mode matches most types of driver outputs, and the drive polarity may be easily switched either in the Tower LCC+Q configuration or by reversing the RB-4 output contacts.

9.8 RB-2 (Dual DPDT Relay Board)



This board is a convenient way to reverse track polarity by using logic in place of Auto Reverse devices. In many cases track polarity may be calculated. In these cases using a DPDT relay is more cost effective and easier on the equipment. Some examples include the tail track of a simple “Y” and the turnout feeding a reverse loop. In both of these cases the required polarity change follows the turnout position.

9.9 BOB-S (Break Out Board – Screw Terminal)

This board is a convenient way to convert from 10 pin ribbon cable to screw terminals. It may be used for inputs or outputs.

Do not exceed 5V on any input or output or the Tower LCC+Q will be damaged.

The BOB-S may be mounted to a panel or stringer using #4 or smaller screws and spacers.

10 Trouble shooting

10.1 Sanity Test

To perform a very basic Tower LCC+Q sanity test perform the following steps:

- Power up the Tower LCC+Q by plugging it into a powered network.
- The green power LED should come on.
- Once the node powers up it will briefly flash its Blue, Gold, and Red LEDs as it reports its status to the network.
- The previous output states should be automatically be restored.

If the green power LED does not light, be sure that a power supply is connected to the LCC network segment, and provides at least 7.5V to the Tower LCC. The green power LED itself will initially light at much lower voltages, so it is not a reliable indicator of suitable power.

10.2 Activity Test

The Tower LCC's input circuit and code sends data directly to the unit's processor, so if you send any command to the unit it should immediately be seen on the actuator (ACT) LED. This test uses the free software available from the JMRI project to watch the test commands. (www.jmri.org)

Steps:

- Open the JMRI LCC® Monitor window. Using the JMRI turnout control send a command to any output line on this Tower LCC. The command should appear in the LCC® monitor window and the Tower LCC+Q command (Y) LED should blink.
- The connected output should respond.

If there is activity at the LCC Terminator blue LED, but no activity light at the Tower LCC+Q when events are sent, check the LCC wiring. If the command is seen in the LCC® monitor, but not in the command light, be sure that the command you are sending is configured to respond on this Tower LCC. If there is no activity shown in the LCC® monitor window, check that you have the correct interface selected in the JMRI preferences, and that you have the correct COM port selected.

The Tower LCC+Q is initially configured as simple input lines. You may use a RR-CirKits I/O test board to send events simply by connecting it to either input port and pressing the test buttons. An unprogrammed node should respond with the default EventIDs to any input changes.

11 Boot Loader

11.1 Boot Loader Upgrade

If you should ever need to upgrade the Boot Loader itself for some reason follow these steps. Skip this section for normal firmware upgrades.

!! IMPORTANT !!: If you have JMRI open, please close it.

1) Restart JMRI V5.6 or later and do not open the 'OpenLCB > Configure Nodes' menu under any circumstance.
2) Select OpenLCB > 'Firmware Update'.
3) From the 'Target Node ID' drop down box, select the TowerLCC node to be updated.
4) Click Select to pick a firmware file.
5) From the file menu, select this file: 'xxx'.
6) Click "Open' and leave 'Address Space' at '239'; do not check 'Lock Node'.
7) Now click 'Load' to download the new boot loader Vxx.

During the download the Gold led will blink to show that the node is in 'Boot Loader State'. (10% flash)

The progress bar on screen will now fill up to 100%, on the node the Blue and Red led show bus activity.

After some time the message 'Download completed successfully' should appear.

The Gold led will continue to blink, because the node will remain in 'Boot Loader State'.

Do not close the 'Firmware Downloader' window at this time, leave it open.

From the OpenLCB menu, click 'Configure Nodes' and select the (partially) updated node.

It should now show 'Mod: Tower-LCC Bootloader' and 'Software: Vxx'.

Do not close the 'OpenLCB Network Tree' window at this time, leave it open and proceed to 10.2 (step 3).

11.2 Firmware Upgrade

Note: Version C and later CDI files store the Virtual Track Circuit information in a different format than version B. You will need to re configure this section after an upgrade. Do NOT restore a configuration saved from an earlier firmware version. It will destroy your virtual track circuit information.

If an update to your Tower LCC+Q firmware is needed, a program such as "Firmware Update" in JMRI version 5.4 or later is required.

To enter Firmware upgrade mode:

1) Start JMRI and select "OpenLCB".

2) Select ‘Firmware Update’ from the OpenLCB drop down list.
3) Select your 'Target Node ID'. Note: If you have just completed the boot loader upgrade it should still be selected.
4) Click ‘Select’ to pick a firmware file.
5) From the file menu, select: 'Tower-LCC-Q-V104-UPDATE.hex' or the latest upgrade available.
6) Optionally you may check the ‘Lock Node’ check box to take it off line during the upgrade.
7) Click the 'Load' button to initiate the upgrade to Tower-LCC+Q revision v1.04. (or the latest version)
8) Wait until 'updating device firmware..' is complete.
9) Switch back to the OpenLCB Network Tree window.
10) It should now show 'Model: Tower-LCC+Q' and 'Software Version: v1.04'.
11) Any errors will be shown in the lower window ticker tape display.

If the node does not automatically enter boot mode and start the upgrade it may be forced into boot mode by un-powering it, then holding down the ‘Gold’ button as you power it up again. The gold LED should start flashing to indicate that it is in forced boot mode. This will likely be required after a failed upgrade attempt.

12 Grounding and Isolation

Unlike the LCC Buffer-USB, the Tower LCC+Q is not optically isolated from the LCC bus. This allows for possible ground loop problems between the LCC® and your layout accessory power supplies, so be sure to keep the ground connection to the Power-Point either isolated, or else in common with your layout power source.

Normally all Tower LCC+Q connections will originate or reference to the Tower LCC+Q board itself, so there is no danger of ground loops with these connections. RR-CirKits High power output boards are optically isolated from the Tower LCC+Q ports and use their own power sources.

If you are building your own I/O boards or using third party units be sure to observe the common/isolated ground rules, and never exceed 5V on any I/O pin.

Properly ground your boosters, your power supplies, and your desktop computer through a 3 wire cable if they are not double isolated, and isolate them from each other via isolated equipment where necessary.

13 Warranty Information

We offer a one year warranty on the Tower LCC+Q. This device contains no user serviceable parts.

If a defect occurs, please contact RR-CirKits at: service@rr-cirkits.com for a replacement.

14 FCC Information

This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions:

1. This device may not cause harmful interference, and

2. this device must accept any interference received, including interference that may cause undesired operation.

Note: This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:
--Reorient or relocate the receiving antenna.
--Increase the separation between the equipment and receiver.
--Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
--Consult the dealer or an experienced radio/TV technician for help.

Any modifications to this device voids the user's authority to operate under and be in compliance with these regulations. The actual measured radiation from the Tower LCC+Q is much lower than the maximum that is permitted by the FCC Rules, so it is unlikely that this device will cause any RFI problems.

RR-CirKits, Inc.
7918 Royal Ct.
Waxhaw, NC USA 28173

http://www.rr-cirkits.com
sales@rr-cirkits.com
service@rr-cirkits.com
1-704-843-3769
Fax: 1-704-243-4310