Garfield-Clarendon Model Railroad Club

4501 N Clarendon Avenue
Chicago, IL 60640

Layout Control System

LCS Overview

The Layout Control System or LCS is a micro-controller based system used to control model railroad accessory devices such as turnout motors, signals, control panels, etc. The system is based around the ESP-12F ESP8266 Wi-Fi module. This low cost, Wi-Fi-microcontroller module has the perfect combination of power and features which makes it ideally suited for a project of this type. For the purpose of this overview, the Layout Control System is referred to as "LCS" throughout this document.

Layout Control System

LCS Overview

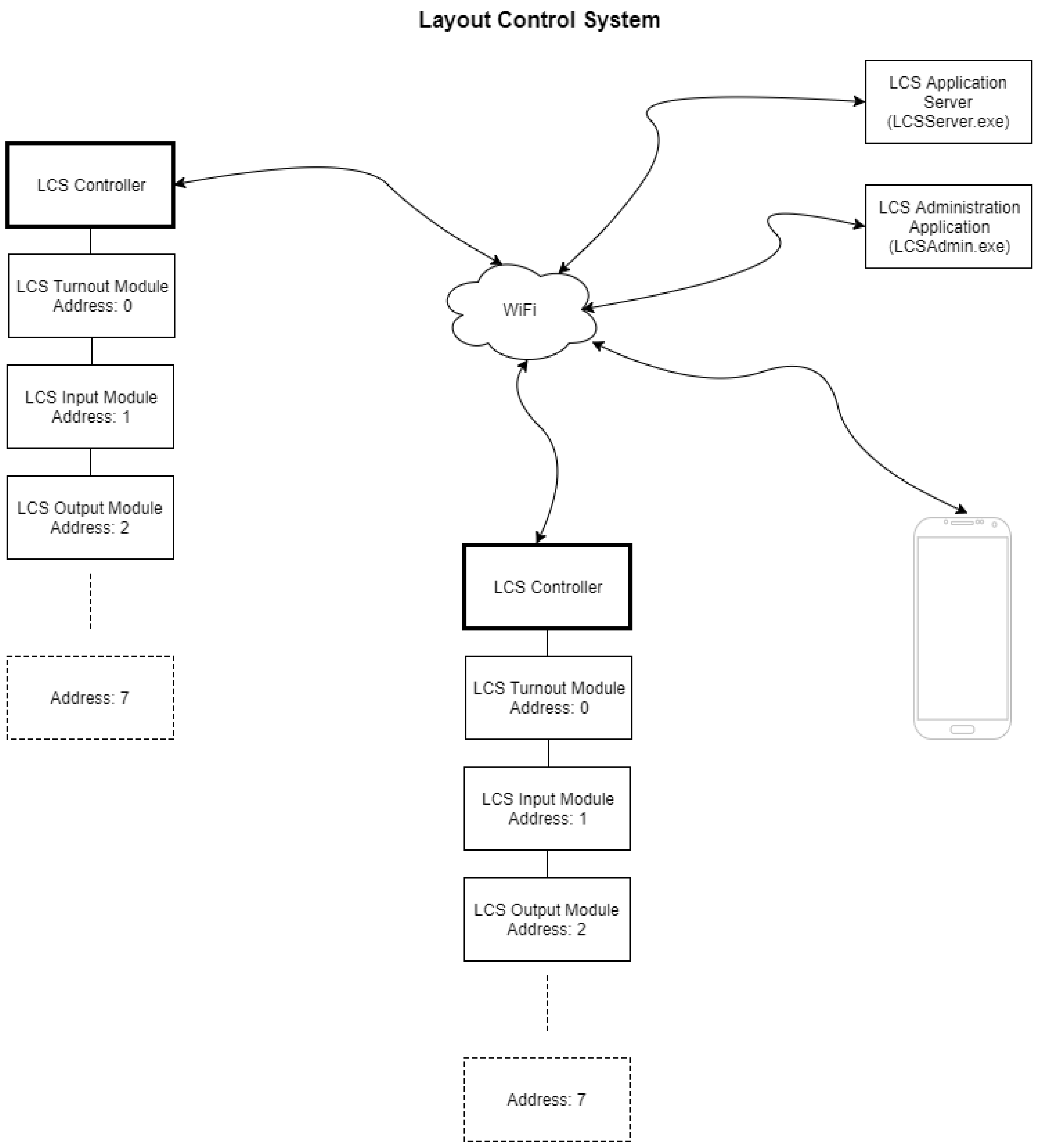

Layout Control System
LCS Application Server (LCSServer.exe)
LCS Administration Application (LCSAdmin.exe)
LCS Controller
LCS Turnout Module Address: 0
LCS Input Module Address: 1
LCS Output Module Address: 2
Address: 7
WiFi
LCS Controller
LCS Turnout Module Address: 0
LCS Input Module Address: 1
LCS Output Module Address: 2
Address: 7

Figure 1 LCS Overview

Layout Control System

LCS Overview

As pictured, LCS is broken down into the following components:

Controller	The ESP8266 based micro-controller board that controls the attached accessories. Each controller board can communicate with up to 8 module boards over the I2C serial bus.
Turnout Module	Controls two turnout motors and receives feedback from the turnout motor as to its current position (normal or thrown)
Input Module	16 input pins are available to be used by control panel buttons and/or block detection units
Output Module	16 output pins are available to be used by control panel LED's and/or signals
Application Server	The application that manages the database which stores the configuration data that drives the system. The application server also acts as the interface between outside systems such as the NCE Command Station, smart phones, dispatcher panels, etc.
Administration Application	A user-friendly application for managing the system configuration data.

Table 1 LCS Components

LCS Controller

The controller provides processing and communication. There are three types of controllers:

Class	Controller Type	Description
7	Multi-Controller	The most common controller type. This controller manages multiple module boards connected via an I2C serial bus cable.
1	Turnout Controller	A controller and turnout module combined into one board which can control two turnouts.
5	Semaphore Controller	Semaphore Signal Controller which is a controller and turnout module combined into one board. The turnout module is used to run the motor that moves the signal arm.

Table 2 Controller Types

Most of the controllers on the layout will be the Multi-Controller as it can handle many more devices. Since we have only one Semaphore signal on the layout, there's only one Semaphore Controller. Although there are three controller types, there are only 2 actual controller printed circuit boards. The Semaphore Controller uses the same board as the Turnout Controller. Internally, the three controllers use the same firmware. The firmware code knows what controller it's actually running on based on the Controller Class setting in the controller's configuration file (described shortly).

LCS Modules

The LCS Controller does not directly control devices. Rather, each controller may have up to 8 module boards connected which, in turn, control the attached devices. There are 4 types of module boards (see Table 5). Module boards are connected using a communication cable (figure 2).

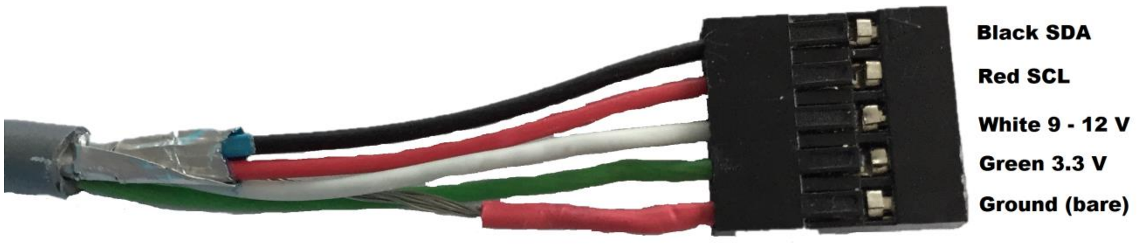


Figure 2 Communication Cable

The black and red (SDA, SCL) wires are the actual communication wires used by the I2C bus. The white wire acts as a pass-through power source which can be used by modules to power devices that require higher voltages (like turnout motors). Although labeled as 9 – 12 V, the voltage is actually the same as the power supply plugged into the controller board. If the power supply is a 9 volt power supply, then the voltage between the white wire and the ground (red) wire will be 9 volts. The green wire is the 3.3 V output of the voltage regulator on the controller board. The regulator will adjust the input voltage to keep it as close to 3.3 volts as possible. This supplies the regulated voltage needed by the IC chips on the module boards.

Each module has a corresponding entry in the database which defines the module's type, which controller it's connected to and the module's address. Module boards must have a unique address of 0 to 7. To achieve this, a three-position dip switch is used to set the module's address:

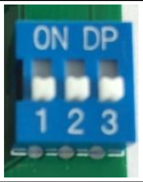
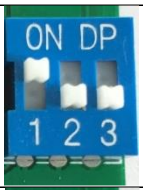
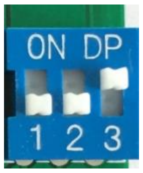
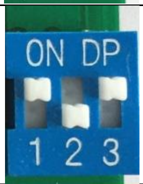
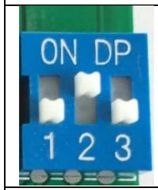
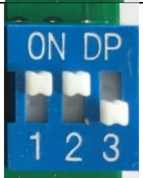
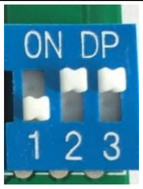
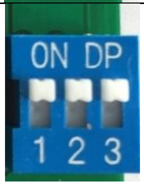
	Address 0		Address 4
	Address 1		Address 5
	Address 2		Address 6
	Address 3		Address 7

Table 3 Address DIP switch settings

NOTE: The address set with the dip switches must match the address entered in the module's entry in the database. Module boards may be connected to the controller in any order.

LCS Device

LCS treats everything it interacts with as a device. A device represents an item on the layout that the system either controls or monitors. LCS supports the following types of devices:

Turnout	A slow-motion motor Which controls a turnout on the layout.
Panel Input	A momentary push-button or switch on a layout control panel. These buttons, when pressed, activate a route which aligns one or more turnouts to a desired route
Panel Output	A panel LED indicator light. The LED indicates the current state of a turnout , route or block occupancy.
Signal	A three-light (LED) signal head.
Block Occupancy Detector	Monitors an external block detection circuit.

Table 4 Device Types

Like modules, each device has a corresponding database entry which defines the device type, assigned module and the port (pin) on the module the device is connected to.

LCS Configuration Files

JSON formatted configuration files are stored in the controller's permanent flash memory. When the controller starts up, the first file it loads is the controller's configuration file:

```
{
    "controllerClass": "7",
    "controllerID": "34",
    "controllersToNotify": [{
            "controllerID": "28"
        }
    ],
    "modules": [{
            "address": "0",
            "class": "8"
        }, {
            "address": "1",
            "class": "9"
        }, {
            "address": "2",
            "class": "9"
        }
    ]
}
```

This is the configuration file for the CA Auxiliary control panel. Here's what the entries mean:

"controllerClass" contains a number that represents the type of controller board (see table 2) “controllerID” contains the unique ID of the controller from the controller table in the database. Each controller is identified by a controllerID and a serialNumber. The controllerID entry is used to cross-reference the controller with the modules connected to it. The serial number is a unique id assigned by the manufacturer to each ESP8266 Wi-Fi module. The reason for the two identifiers is so that if a controller needs to be replaced, the only field that needs to be updated in the database is the serial number field.

“controllersToNotify” is a list of controller ID’s that need to be notified whenever the state of a connected device changes. Since this configuration is from the CA Auxiliary panel, the device that will trigger a notification message is a Panel Input device (panel button). When the user presses a button on the panel, the Panel Input Device will generate a message which will be sent to each controller listed in this section. In this example just one controller will be notified which is controller 28 – “CA West End Multi 1” controller.

“modules” is a list of modules connected to this controller via the I2C serial bus . Each module entry contains the address of the module and the module class:

Class	Module Type	Software Module Class
1	Turnout Module	TurnoutModule
5	Semaphore Module	TurnoutModule
8	Input Module	InputModule
9	Output Module	OutputModule

Table 5 Module Types

In this example, three modules are connected; an input module (class 8) and two output modules (class 9) with the addresses of 0, 1 and 2 respectively.

Next, the LCS Controller creates software modules that manage the interaction between the physical module board and the devices connected to the module. The LCS Controller communicates with each module via the I2C serial bus communication cable (Figure 2).

Each module’s configuration file is the next file to be loaded:

```
 {
    "devices": [{
            "c": "2",
            "id": "44",
            "p": "0"
        }, {
            "c": "2",
            "id": "43",
            "p": "1"
        .
        .
        .

        }, {
            "c": "2",
            "id": "52",
            "p": "15"
        }
    ],
}
```

Each JSON file contains a list of devices attached to the controller module. In this example, the devices are all the same; device class 2 – Panel Input, as defined by the "c": "2" entry. Each device entry in this example file represents a button on the CA Auxiliary Panel. The other two properties in the device configuration represent the device ID "id" and the pin assignment "p". Putting it all together, the first entry is a Panel Input device (2) with a device ID of 44 (id) and is assigned to pin 0 (p). For each entry in the file, a software class that manages the device is created. This software class has all the logic needed to manage the device. Once an instance of the device class is created, the next step is to load the configuration data for that device. As you probably guessed, this involves loading a JSON file for each device attached to the controller module:

```
{
    "ROUTEID": "19",
    "deviceClass": "2"
}
```

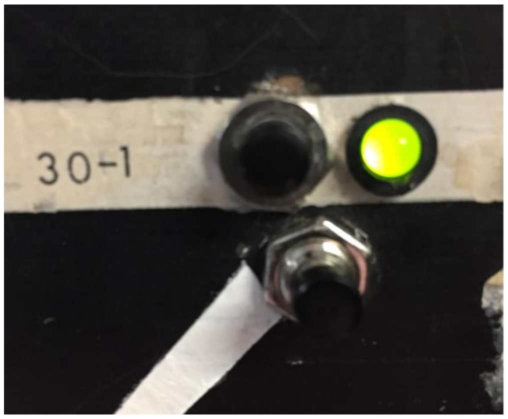
Continuing with the previous example, this is the configuration for the first device entry, device 44. The important line in this file is the ROUTEID entry. This represents the route that gets activated when the button connected to pin 0 is pressed on the CA Auxiliary Panel.

The reality is, the configuration files are not actually stored as JSON files. The reason is these files must be loaded and then parsed by the micro-controller and converted to a binary structure it can actually use. This process is pretty slow, so instead, the binary structure is what is actually stored in the file. The JSON format is only used when the controller needs to download the configuration files from the

server. Once downloaded, the JSON information is parsed and converted to the binary structure and saved to a file.

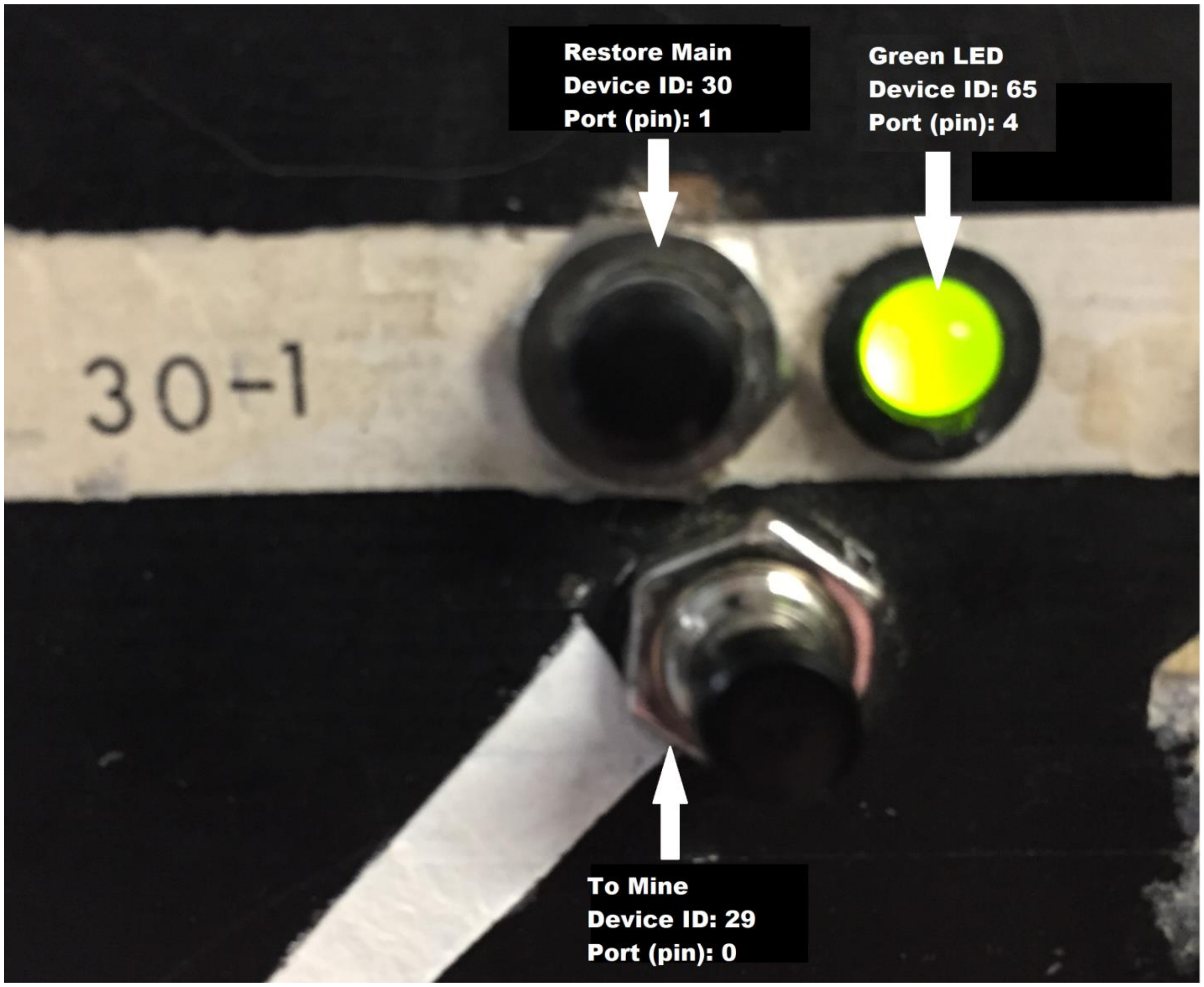
LCS in Action

The best way to understand how a system works is by following an example. Let's do a simple run to the Gaskill Mine.

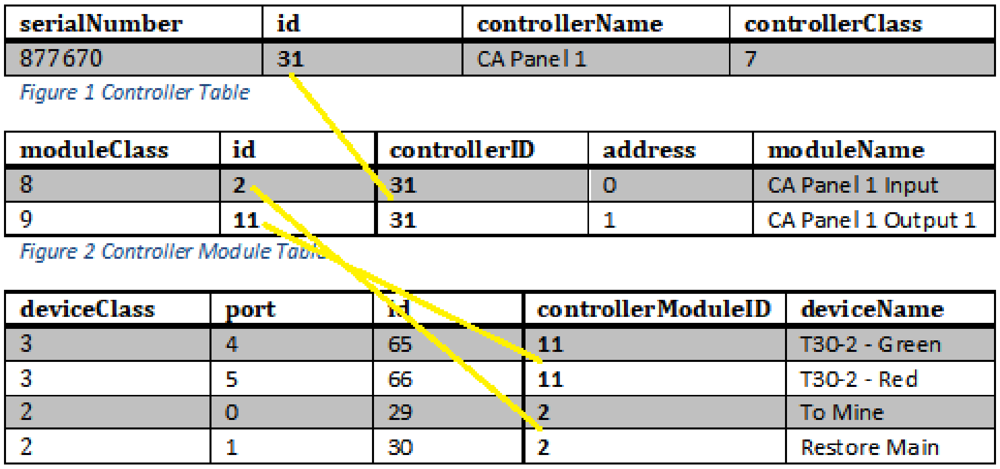


Milwaukee Road 5506 has two empties to drop off at the Gaskill Mine and awaits the turnout alignment to proceed into the mine branch. Currently, the mine's turnout, T30-2, is aligned for the main, its normal route, as indicated by the green LED indicator on the CA panel.

In order to route the train to the mine branch, we need to press the bottom button. But what is really happening when the button is pressed?



LCS treats each item it interacts with as a device. In the picture above, we have three devices listed; two buttons and one LED. The reality is, as we'll see in a moment, there are really 4 devices as the LED is really two LEDs in one package; a red and a green. Notice that each device has a name, a unique ID and a port/pin assignment. In addition each device is assigned to a module that actually controls the device. In this example there are two modules; an Input Module and an Output Module. Let's look at the database entries that make up this example:



serialNumber	id	controllerName	controllerClass
877670	31	CA Panel 1	7

Figure 1 Controller Table

moduleClass	id	controllerID	address	moduleName
8	2	31	0	CA Panel 1 Input
9	11	31	1	CA Panel 1 Output 1

Figure 2 Controller Module Table

deviceClass	port	id	controllerModuleID	deviceName
3	4	65	11	T30-2 - Green
3	5	66	11	T30-2 - Red
2	0	29	2	To Mine
2	1	30	2	Restore Main

Figure 3 Device Table Entries

Note the additional entry in the device table (figure 3) for the "T30-2 – Red" LED device. Also note how the entries in each table are crossed reference with the table above it: The controller entry of id 31 is cross referenced in each entry in the module table via the controllerID field. Each device is cross referenced with its assigned module via the controllerModuleID field. Using this information, the controller has almost everything it needs to manage the buttons and LEDs. It knows *how* to interact with each device but not *when* each device LED should be on or off. For that information, we need to dig a little deeper into each device entry.

Devices have properties that are unique to each device type. These properties are stored in another database table called "deviceProperty". Let's look at the first button we're going to press, the "To Mine" button, and examine its device property entry:

id	deviceID	key	value
7	29	ROUTEID	7

Figure 3 Device Property Table

This type of table is referred to in programming terms as a "key/value pair" table. In this case, "ROUTEID" is the key and "7" is the value. This data is used by the PanelInputDevice class to know what route to activate when a button is pressed; in this case, route 7.

As you probably guessed by now, each route is stored in its own database table called "route" and has a corresponding table, "routeEntry", which contains an entry for each turnout the route needs to set when the route is activated. Let's take a look at route 7:

id	routeName	routeDescription
7	CA 1	To Mine

Figure 4 Route Table

id	routeID	deviceID	turnoutState
11	7	7	3

Figure 5 Route Entry Table

Route 7 has one Route Entry item; Turnout device ID 7 (T30-2). When route 7 is activated, the Turnout Device will set the turnout with ID 7 to the turnout state of 3 (Diverging).

Let's put it all together. When we push the bottom button, the InputModule will detect that the pin that button is connected to, pin 0, has changed. This information is passed on to the PanelInputDevice which activates route 7. How is route 7 actually activated? By sending a message!

UDPMessage and the Wi-Fi Network

The way one controller talks to another controller and the Application Server is via the Wi-Fi network. Because the ESP8266 is a very basic micro-controller with limited speed and memory, we need an efficient, low-overhead solution for sending messages between devices. The User Datagram Protocol, UDP, is the ideal solution for this problem. UDP is very fast and efficient. The tradeoff is reliability. UDP by itself is an unreliable protocol, meaning there is no guarantee the message will actually reach its intended destination. The LCS Controller compensates for this by retrying each message until it reaches its target controller.

In our example, the PanelInputDevice creates a UDPMessage containing the route ID of 7 which is sent to the controllers in the Notification List from the controller 31's configuration file. In this file is the `controllersToNotify` section:

```
"controllersToNotify": [ { "controllerID": "28" } ]
```

In this case, the message is sent to controller "28 - CA West End Multi 1" which is the controller that the T30-2 turnout is connected to.

On the receiving end, controller 28 forwards the message to the TurnoutDevice that is managing T30-2. Let's take a look at its configuration file:

```
{
    "deviceClass": "1",
    "routes": [{
            "routeID": "7",
            "turnoutState": "3"
        }, {
            "routeID": "8",
            "turnoutState": "1"
        }
    ]
}
```

As you can see, each turnout stores the routes its' assigned to and the state they need to be set to when the route is activated. In our example, when the message arrives, TurnoutDevice T30-2 will lookup route 7 and set its state to "3 – Diverging". It then immediately sends out a new message indicating the turnout is now changing by setting its current state to "2 – To Diverging" indicating the turnout is in the process of throwing to the diverging route. The TurnoutDevice's status message is sent to controller 28's notification list:

```
"controllersToNotify": [{
        "controllerID": "31"
    }, {
        "controllerID": "34"
    }, {
        "controllerID": "35"
    }, {
        "controllerID": "30"
    }
]
```

Notice there are many more controllers that need to be notified when something connected to controller 28 changes! The first controller in the list is our example controller,"31 - CA Panel 1". It also sends messages to:

- 34 - CA Aux Panel
- 35 - Overhead Panel
- 30 - Semaphore1

Controller 31 forwards this message to the PanelOutputDevices that manage device ID "65 – T30-2 Green" and device ID "66 – T30-2 Red". The device properties for device 65 are:

id	deviceID	key	value
89	65	ITEMID	7
90	65	ONVALUE	1
91	65	FLASHINGVALUE	4

ITEMID is set to the device the PanelOutputDevice is monitoring; in this case device ID “7 – T30-2” and sets its output pin ON when T30-2 is “1 – Normal” and flashing when T30-2 is “4 – To Normal”. In our example, the current state of T30-2 is “2 – To Diverging”. Since this doesn’t match either of the two entries, the PanelOutputDevice sets the output to LOW turning the green LED off.

Device 66 has the following entries:

id	deviceID	key	value
89	66	ITEMID	7
90	66	ONVALUE	3
91	66	FLASHINGVALUE	2

Which means T30-2’s current state of “2 – To Diverging” matches its FLASHINGVALUE entry, so the PanelOutputDevice sets the output to a blinking state; turning the red LED ON and then OFF every ¾ of a second.

The result is T30-2 goes from its normal route:

To its diverging route:

Now that the turnout is fully thrown, the TurnoutDevice sends a final status message, this time with the status of "3 – Diverging" as T30-2's current state. The new state of 3 matches the PanelOutputDevice 66's ONVALUE configuration entry so it sets the output to ON turning on the red LED (the LED stops flashing).