Model_Train_DC.ino

Setup Instructions

Benfpv 2024

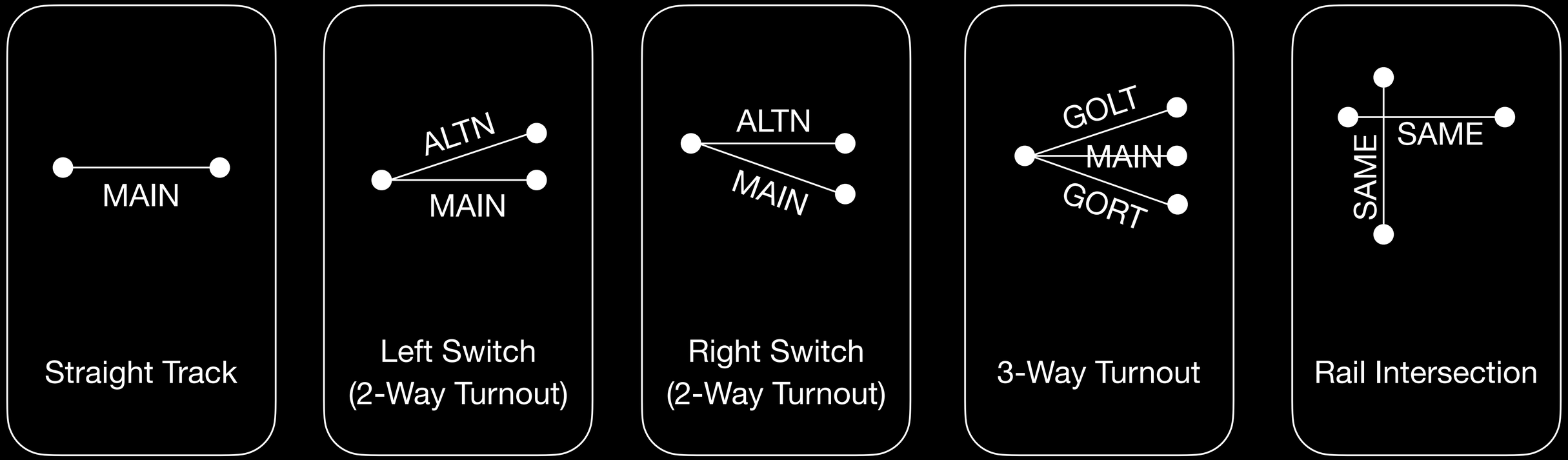
Setup Rail Configuration In-Real-Life (IRL)
Rail Sectioning

- Setup the railway to your preference, while adhering to these rules:
  - The entire railway must be sectioned accordingly:
    - Each rail section is defined by one train-sensor (e.g., proximity sensor) at the beginning of that rail section (rail-section-begin sensor), and one at the end of that rail section (rail-section-end sensor).
    - At least 2 rail sections must exist - rail sections should not overlap.
    - Each rail section must be generously longer than the longest train that will travel through that rail section.
    - Each rail section that will house a rail station (where trains can stop) should be long enough to enable the longest train that will travel through that rail section to stop head-first at the rail station at the rail station (i.e., forward-most train car will stop at the rail-station sensor) without blocking a rail-section-begin or rail-section-end sensor while stopped.
    - Each rail section must be given a unique 4-letter name (capitals-agnostic).
    - For each switch/turnout track, one rail section must exist before each entry, and after each exit.

Setup Rail Configuration In-Real-Life (IRL)

Rail Sectioning

Essential & Example Rail Sections



Rail

Train Sensor Placement

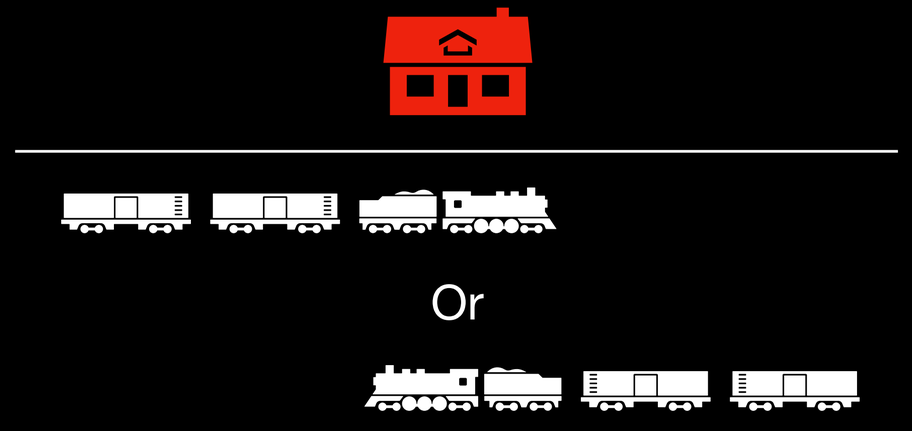
Setup Rail Configuration In-Real-Life (IRL)

Rail Station Placement

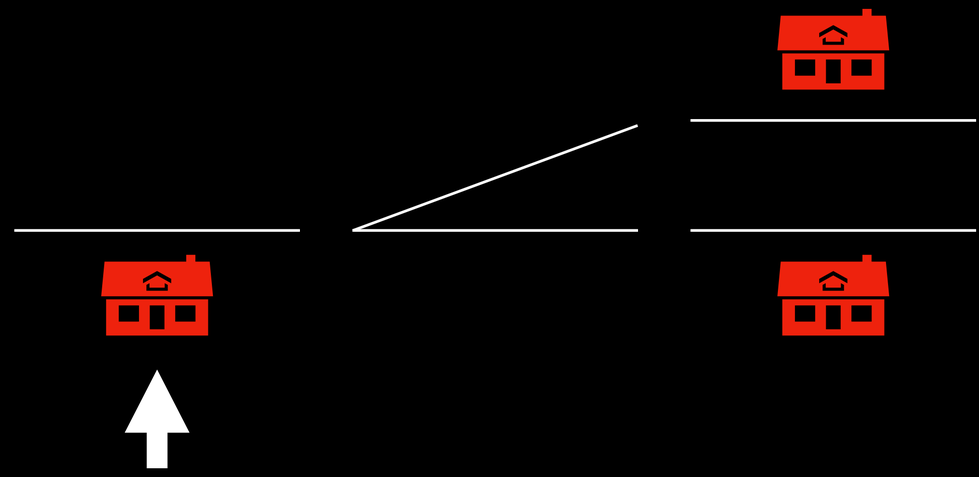
- Setup rail stations to your preference, while adhering to these rules:
  - Each rail station is defined by one train-sensor (e.g., proximity sensor) in between the rail-section-begin and rail-section-end sensors of a rail section. Trains will stop at this station head-first.
  - At least 2 rail stations must exist - each rail station must occupy only one rail section.
  - Each rail station should be placed such that the longest train that will stop at this rail station will be able to stop head-first at the rail station (i.e., forward-most train car will stop at the rail-station sensor) without blocking a rail-section-begin or rail-section-end sensor while stopped.
  - Each rail station must be named identically to the name of the rail section it occupies.
  - Trains can not change direction more than once per trip (including any direction change preceding departure). Thus, rail stations must be placed wherever a train must change direction to navigate from one rail station to another.
  - Each intersection must be considered one rail section.

Setup Rail Configuration In-Real-Life (IRL)

Rail Station Placement



How trains stop at rail stations



Rail stations placed at each direction change

Create a Folder-Tree which Reflects the IRL Rail Configuration

Folder-Tree Creation

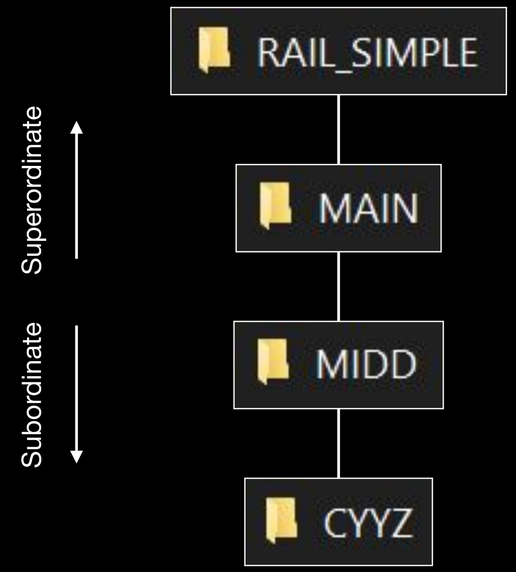
- The highest-level directory should the name of your railway - Spaces should be replaced with underscores (“_”).
- Within the highest-level directory, create a folder tree such that the name of each folder reflects the name of each rail section, each subordinate folder is named according to the rail section(s) existing in the forward direction, and each superordinate folder is named according to the rail section(s) existing in the backwards direction.
- Note, to create the folder tree, I recommend visualizing actually being in the rail section to understand what folders should be located where. Imagine you are situated within a/the very first rail section (e.g., MAIN), looking onward to the rail sections ahead.
    - You start within the 'MAIN' rail section, and thus within the '<your railway name>/MAIN' directory. As you travel and enter into the next rail section (e.g., 'TWOO'), you are analogously entering that corresponding directory ('<your railway name>/MAIN/TWOO'), and so forth.
    - Now let's imagine you have a switching rail that enables going from the 'TWOO' rail section to either the 'THRE' rail section, or the 'ALTV' rail section. In this case, you would add two folders within the '<your railway name>/MAIN/TWOO' directory; one for 'THRE' and one for 'ALTV'. Any name is fine and nevermind which rail should come first or how they correspond to which switch position - you will handle these details later.
    - Now let's imagine you decide to build a railway from elsewhere, which leads into the currently existing 'ALTV' rail section, from a similar direction as from 'MAIN'. First, create a folder for the start of that new rail section arbitrarily named 'NEWR' ('<your railway name>/NEWR'). Then imagine you are situated within this rail section, and you see the rail section lead into the 'ALTV' rail section via another switching rail. Thus, 'ALTV' connects to either 'TWOO' or to 'NEWR' via a switch, but 'NEWR' and 'TWOO' don't directly connect to eachother. In this case, simply create the folder named 'ALTV' within '<your railway name>/NEWR' to indicate that 'NEWR' leads to 'ALTV', and 'TWOO' leads to 'ALTV'. This automatically infers to the script that 'ALTV' connects to both 'TWOO' and 'NEWR' via a switch.

Create a Folder-Tree which Reflects the IRL Rail Configuration

Folder-Tree Creation

Visualization of Railway named “RAIL_SIMPLE”

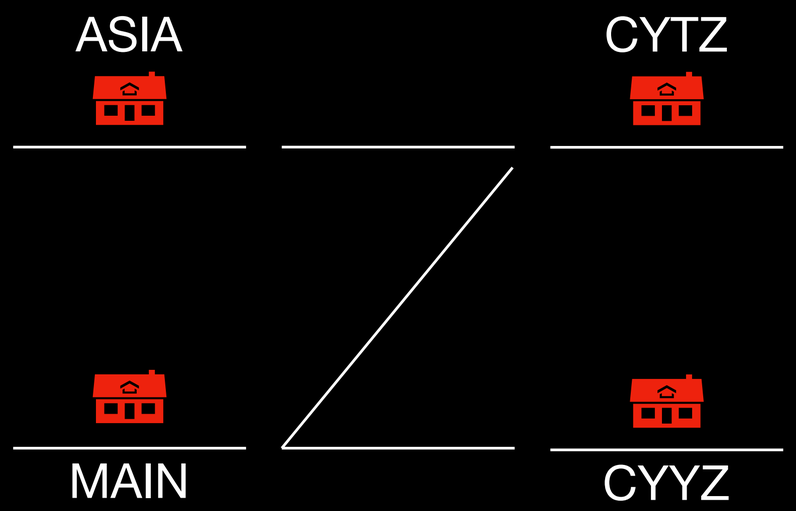
Example Folder Tree of “RAIL_SIMPLE”



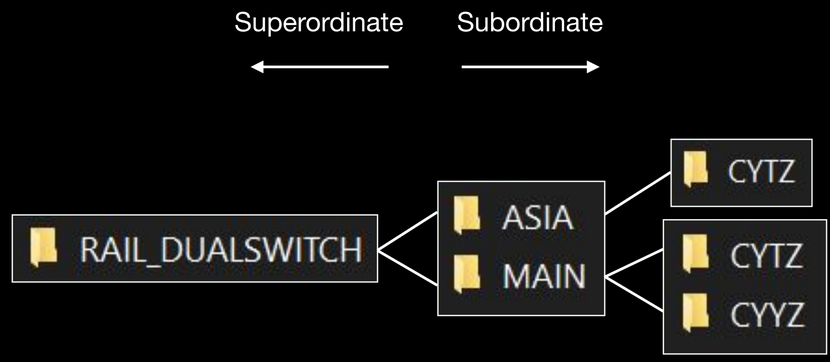
Create a Folder-Tree which Reflects the IRL Rail Configuration

Folder-Tree Creation

Visualization of Railway named “RAIL_DUALSWITCH”

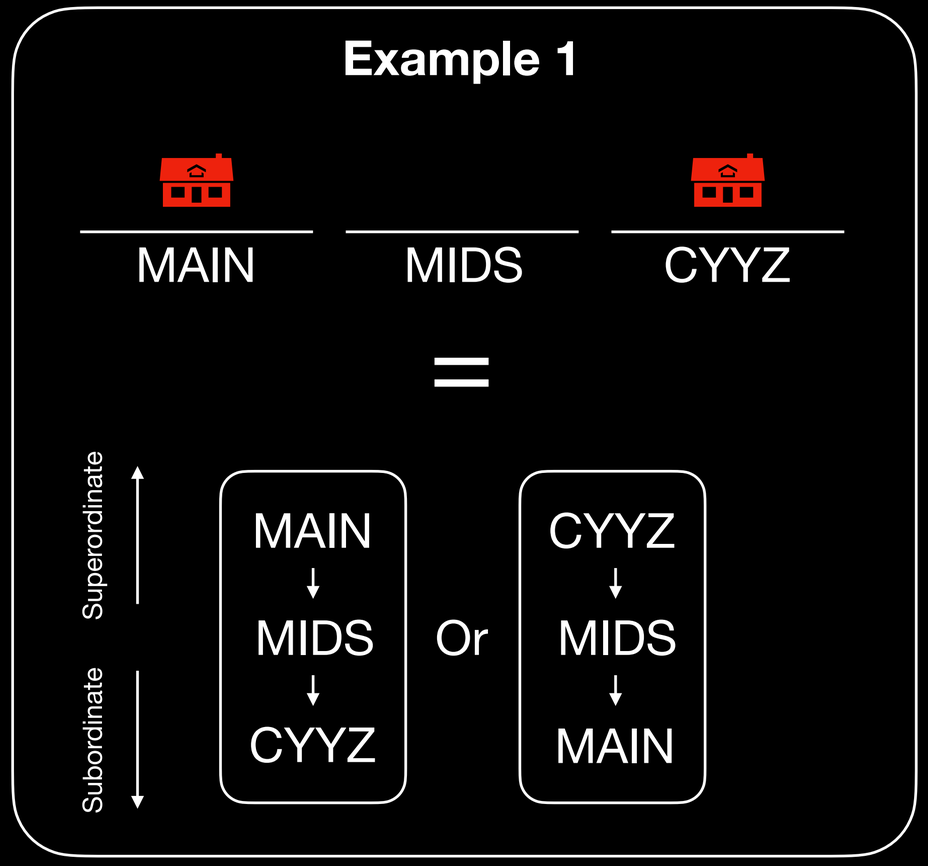


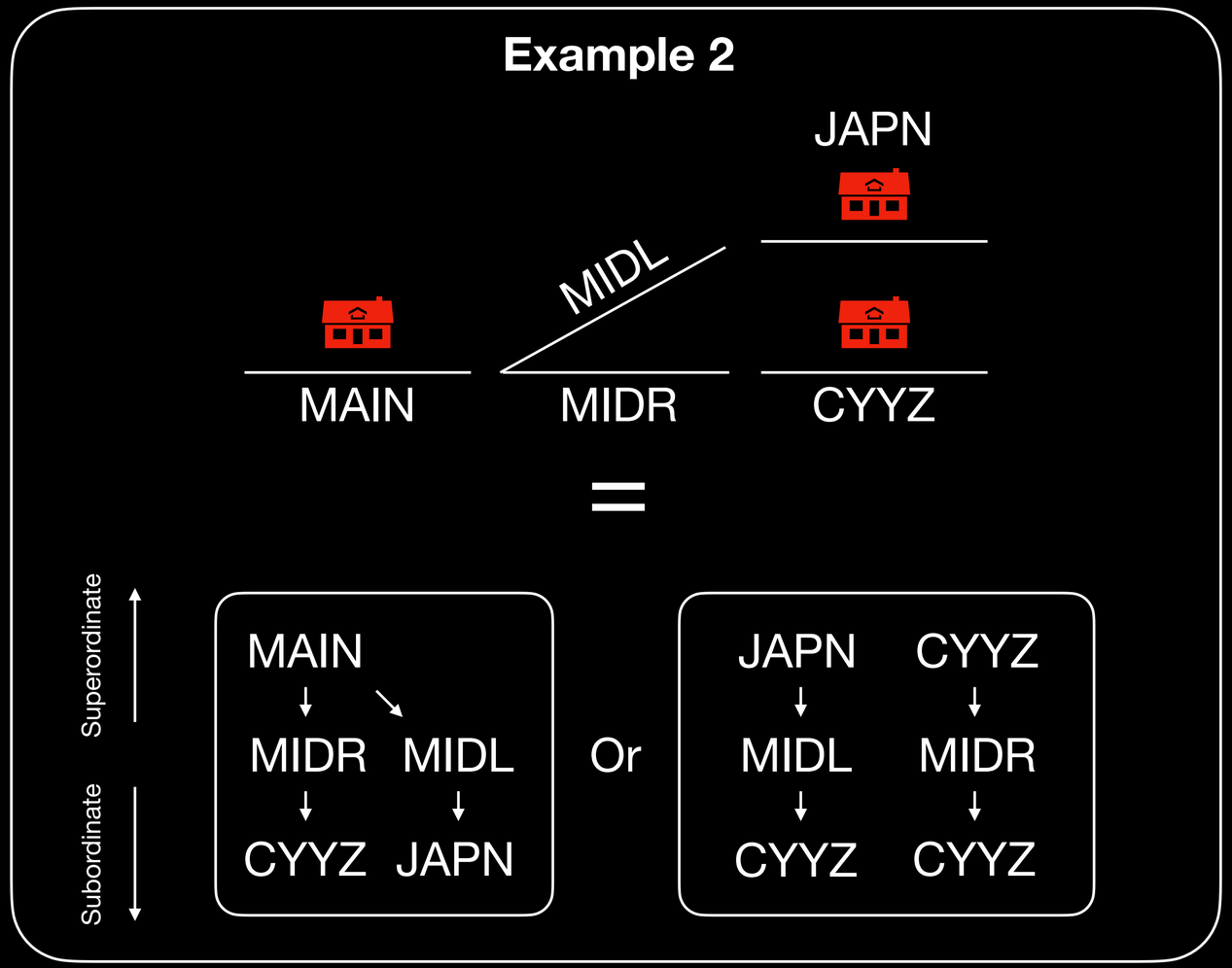
Example Folder Tree of “RAIL_DUALSWITCH”



Create a Folder-Tree which Reflects the IRL Rail Configuration

Folder-Tree Creation





Edit and Run “Rail_Parser.py”

Edit and Run the Python script “Rail_Parser.py”

- Edit the included “Rail_Parser.py” located in “/model_train_dc/*” with a source code editor/ integrated development environment (IDE) (e.g., Notepad++, Visual Studio Code, etc.).
- Within “Rail_Parser.py”, change the value of 'master_folder_name' to equal to the folder name according your railway (e.g., master_folder_name = “Hello_Railway”).
- Run “Rail_Parser.py" with python 3+ (e.g., in Windows commant prompt, type "py Rail_Parser.py" and press enter). This script should then automatically generate the appropriate vectors to the specific to the configuration of your railway. Once generated, you will be asked to confirm whether the information given looks correct. You should see vectors for the directory paths of all rail sections, all rail section names, the names of any switching rails, etc. If all looks good, press 'Y' and enter to confirm, and the script should then replace the relevant lines in the 'Model_Train_DC.ino' arduino script file in the current directory.

Edit and Run “Rail_Parser.py”

Edit and Run the Python script “Rail_Parser.py”

```python
import os
import sys
import time

# User Parameters - Required
master_folder_name = "RAIL_SIMPLE" # Folder Name that contains folder tree of rail section names.

# User Parameters - Advanced
ino_file_name = "model_train_dc.ino" # Name of the main arduino file to run.
ino_params_begin_keyword = "// Rail_Parser Output Parameters Begin"
ino_params_end_keyword = "// Rail_Parser Output Parameters End"
dirs_txt_name = "dirs.txt"
pages_txt_name = "pages.txt"

# Script
class Main():
    def __init__(self):
        print("------- BEGIN -------")
        dirs, pages = Main.parse_folder(master_folder_name)
```

Change this to your railway folder name -

e.g., “RAIL_SIMPLE”

Edit and Run “Rail_Parser.py”

Edit and Run the Python script “Rail_Parser.py”

Once the script is run, outputs to console (e.g., cmd) should look something like this:

Beginning of the output will look like this:

```
------- BEGIN -------
- allSubordinateNames:
['MIDD', 'NONE', 'NONE']
['CYYZ', 'NONE', 'NONE']
['NONE', 'NONE', 'NONE']
- allSuperordinateNames:
['NONE', 'NONE', 'NONE']
['MAIN', 'NONE', 'NONE']
['MIDD', 'NONE', 'NONE']
- allSubordinateSwitchesPositions:
- allSuperordinateSwitchesPositions:
- allSubordinateSwitchesNames:
- allSuperordinateSwitchesNames:
```

A “demo” of what the output to “model_train_dc.ino” will look like:

```
String masterDirName = "RAIL_SIMPLE";

String allRailSectionsDirs[3][10] =
{
{"MAIN"},
{"MAIN", "MIDD"}, {"MAIN", "MIDD", "CYYZ"}
};

String allRailSectionsNames[] =
{
"MAIN", "MIDD", "CYYZ"
};
```

Note the “allRailSectionsNames[]” output - make sure the names of each rail section are correctly shown

Edit and Run “Rail_Parser.py”

Edit and Run the Python script “Rail_Parser.py”

Once the script is run, outputs to console
(e.g., cmd) should look something like this:

At the end, it must be confirmed that the information about the rail configuration is accurate, before outputting this info to ‘model_train_dc.ino’.

- Input “Y” and enter, to proceed.
- Input “N” and enter, to cancel.

```
Completed parsing the folder name: RAIL_SIMPLE.
Please confirm - do the parse outputs correctly reflect your intended layout?
Y = Yes, N = No:
```

Edit “Model_Train_DC.ino” - Rail Parameters

Edit the “Model_Train_DC.ino” Arduino script

- Edit the arduino script 'Model_Train_DC.ino' with a source code editor/IDE (such as Arduino IDE). Edit the following lines accordingly. From top-to-bottom:
- Under "// Demo", set/confirm the 'int demoMode' variable to 1 (i.e., int demoMode = 1;).
- Note, the lines between "Rail_Parser Output Parameters Begin" and "Rail_Parser Output Parameters End" were the lines replaced by 'Rail_Parser.py'. Pay special attention to the vector 'String allRailSectionsNames[]' - you will be setting up the pin inputs to Arduino according to the order in which the name of each rail section is presented.
- Under "// User Parameters - Rail Section Begin/End Detectors", input the respective Arduino pin input numbers that correspond to the rail section begin sensors into the 'int beginProxPins_IN[]' vector, in the same order that the rail section names were presented in the 'String allRailSectionNames[]' vector (e.g., if the first rail section name in 'allRailSectionNames[]' is "MAIN", then the begin sensor pin input number for that rail section should go first in 'int beginProxPins_IN[]'. If the second rail section name in 'allRailSectionNames[]' is "TWOO", then the begin sensor pin input number for that rail section should go second in 'int beginProxPins_IN', and so forth). Note, usually I like to fill in the pin input numbers first, and then wire them up afterwards to avoid confusion/mistakes.
- Do the same for the rail section end sensor pin inputs in the vector 'int endProxPins_IN[]’.
- Under "// User Parameters - Rail Fullstop Detectors", declare any fullstop detector pins at your discretion. Whenever sensors connected to these pins are triggered, the train should emergency stop (i.e., stop at high deceleration rate). This is handy to use at the end points of your railway configuration, where the trains run a risk of driving off the rail if they go further. You can leave this empty otherwise.
- Under "// User Parameters - Rail Stations", in the vector 'String allRailStationsNames[]', declare the names of rail sections that have a rail station within them. Trains will stop at these rail stations. You can omit certain rail stations, and the trains will not use them for navigation. However, you can not declare that a station exists when it does not IRL.
- Under "// User Parameters - Rail Station Detectors", in the vector 'int stationProxPins_IN[]', declare the respective Arduino pin input numbers that correspond to the rail station sensors, in the order that the rail station names were declared in 'String allRailStationsNames[]’.
- If your rail configuration has any switches connected via motor controllers (I used L298N), under "// User Parameters - Subordinate Rail Switches", for all subordinate switches presented in 'int allSubordinateSwitchesNames[]', and in the same corresponding order, declare their EN, IN1 and IN2 pins. The EN pin controls the motor speed, while the IN1 and IN2 pins control the polarity of the motor.
- Do the same for the superordinate rail switches.

Edit “Model_Train_DC.ino” - Train Parameters

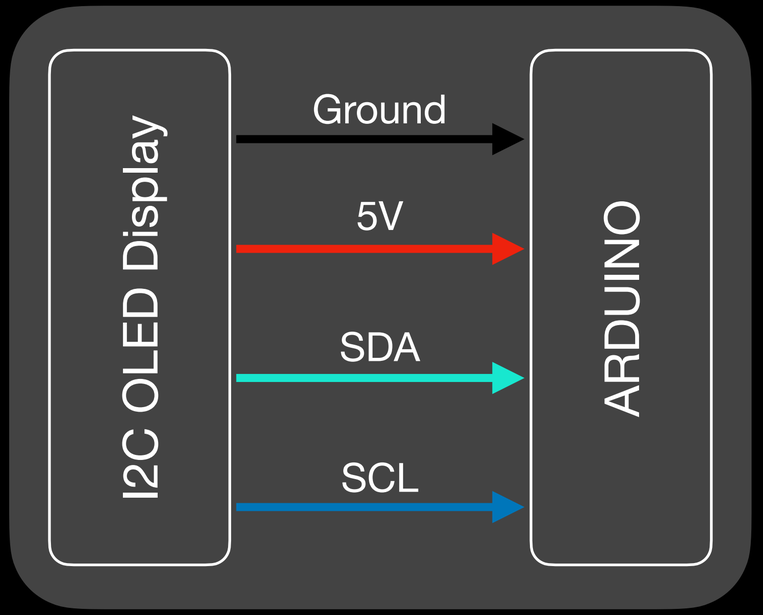
Edit the “Model_Train_DC.ino” Arduino script

- Edit the arduino script 'Model_Train_DC.ino' with a source code editor/IDE (such as Arduino IDE). Edit the following lines accordingly. From top-to-bottom:
- Under "// User Parameters - Trains", for each powered train in your railway (at the time of writing, may only supports one train, but should be easy to adapt the script to run multiple trains), declare the following variables by index - each index corresponds to each train:
- In the vector 'String trainNames[]', input a unique 4-letter name in all-capitals for each train.
- In the vector 'int trainNavigationTypes[]', input a navigation style for each train (0 is randomly navigate to any of the stations declared in 'String trainTargetRailSectionsNamesQueue[]', 1 is sequentially navigate the stations declared in 'String trainTargetRailSectionsNamesQueue[]’).
- In the vectors 'int trainPins_EN[]', 'int trainPins_IN1[]', and 'int trainPins_IN2[]', declare the EN, IN1, and IN2 pins for the motor controllers feeding power to the rails (and thus used to control train motors). Polarity is automatically determined later through practice.
- In the vector 'int trainTargetIdleTimes[]', declare the amount of time (in milliseconds) that each train should stay idle at a station upon arriving, before proceeding to the next one.
- In the vector 'int trainMotorSpeedSlows[]', declare the speed limit in motorspeed (0-255) of each train when travelling in a target rail section (about to arrive at a station). I found 30 to be a good value for a Kato EF81 on my setup.
- In the vector 'int trainMotorSpeedMaxs[]', declare the speed limit in motorspeed (0-255) of each train when travelling in a non-target rail section (enroute to a station). I found 34 to be a good value for a Kato EF81 on my setup.
- In the vector 'int trainMotorAccelerationStoppings[]', declare the acceleration limit (per loop) for each train when stopping normally.
- In the vector 'int trainMotorAccelerationMaxs[]', declare the acceleration limit (per loop) for each train when emergency stopping (i.e., when a fullstop rail section sensor is triggered).
- In the vector 'String trainStartingRailSectionNames[]', declare the starting rail station from which each train will begin at the start of the script.
- In the vector 'int trainTargetRailSectionsInQueue[]', declare the queue of rail stations for each train to either randomly sample from as the destination station, or set as destination station in sequential order, depending on the train navigation type.
- In the vector 'int trainNumberOfTargetRailStationsInQueue[]', for each train, simply input the number of rail stations in that train's queue declared in the 'int trainTargetRailSectionsInQueue[]' vector.

Connect the I2C OLED Display (recommended)

Connect the I2C OLED Display (recommended)

- Connect the I2C OLED Display to the Arduino.
  - I2C Ground to Arduino Ground.
  - Arduino Power (5V) Out to I2C Power In.
  - I2C SDA to Arduino SDA.
  - I2C SCL to Arduino SCL.

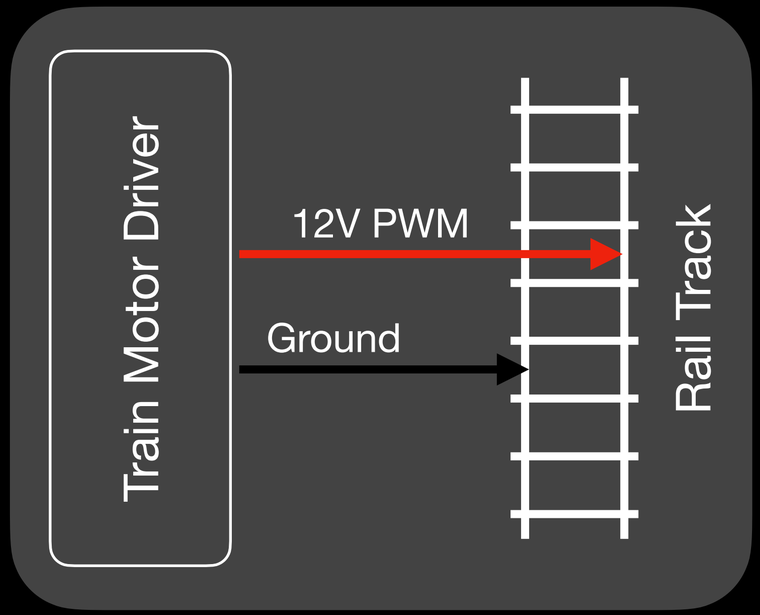


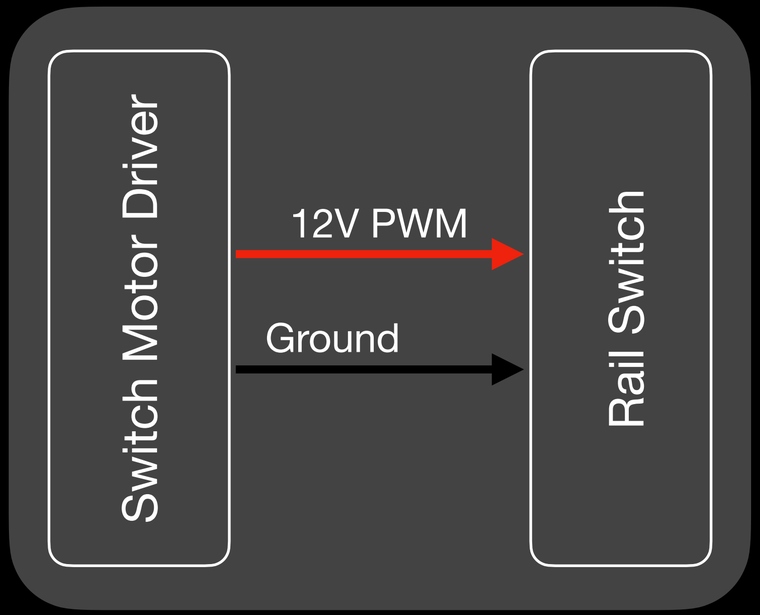
- Within “model_train_dc.ino”, lines such as the “// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)” section may need to change based on your display characteristics. The display I used was a 128x64 I2C OLED display.

Connect Motor Drivers to Trains & Switches

Connect Motor Drivers Outputs to Trains & Switches

- Connect Motor Driver (for the trains) Out (12V PWM) to Rail Track(s).
- Connect Motor Driver (for the switches) Out (12V PWM) to Switches.

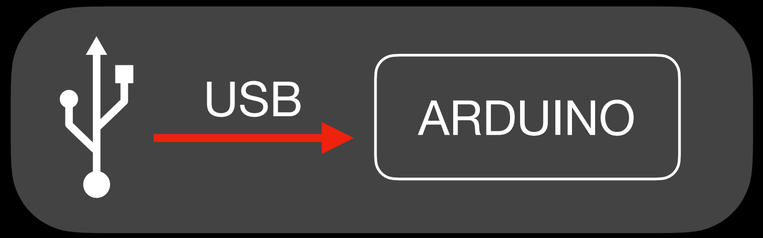


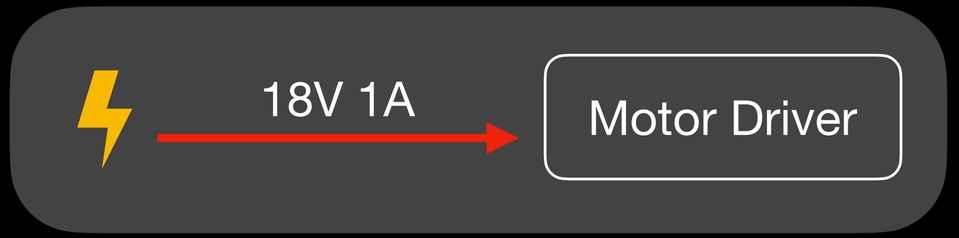


Connect Power to Arduino & Motor Drivers

Connect Power to the Arduino, & Motor Drivers

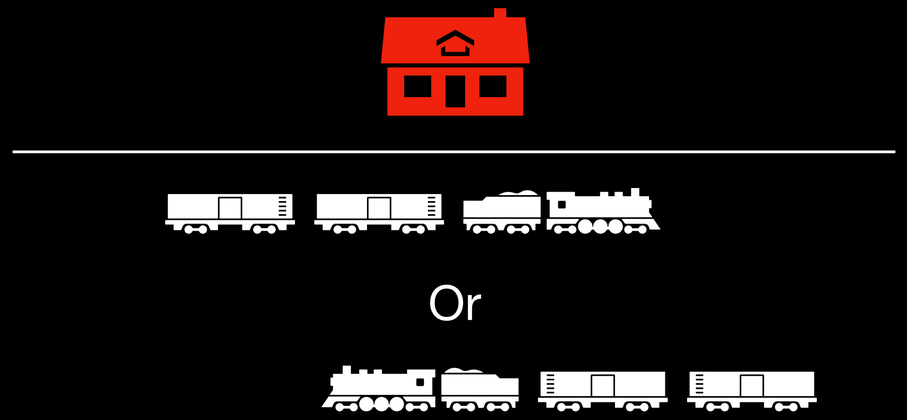
- Connect power to the Arduino - USB, or 12V 1A Into Arduino.
- Connect power to the Motor Drivers (for the Trains) - 18V 1A Into each Motor Driver.
- Connect power to the Motor Drivers (for the Switches) - 18V 1A Into each Motor Driver.



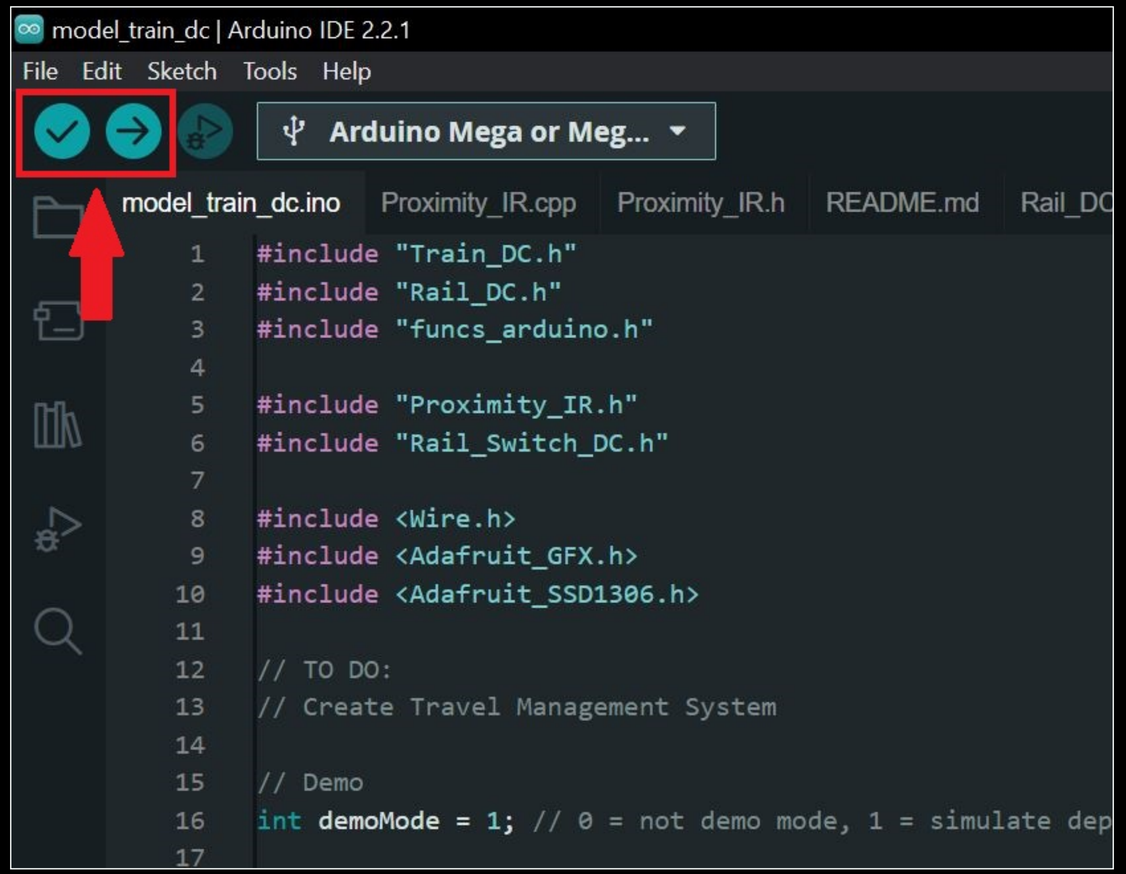


Place the Train(s) at their Starting Stations

Simply make sure each train is placed at their starting station sensor



Upload “Model_Train_DC.ino” to the Arduino Board



“Verify” and “Upload” buttons in Arduino IDE.