CS 452 Milestone 1

Benjamin Zhao, Kyle Verhoog

Mar 08, 2018

Contents

Operation

The elf file is located at `/u/cs452/tftp/ARM/yl4zhao/tc1.elf` and can be initiated using the typical `load` command.

After `load`, wait a few seconds for tasks to initialize.

You should be presented with our interface featuring *windows*.

You can type commands described in the next section.

Commands

- `tr <tr#> <speed>` sets the given train to the given speed.
- `rv <tr#>` reverses the train.
- `rt <tr> <speed>` sets the track to be in a loop and starts the train moving.
- `go <start node> <end node>` tells the train to go to `<end node>`, `<start node>` is not currently used for anything.
- `tk <tr> <starting node>` initializes the train with the system.
- `cls` clears the terminal.
- `cr <offx> <offy> <width> <height>` creates a dummy task with writes
- `sc <train> <speed> <pivot>` stopping calibration command, uses pivot as the node to pivot around.

Design

Train Track

Railway Manager

The Railway Manager acts mostly as an initializor for the track. It is responsible for spawning all track-related tasks and sharing common data like the track data structure. It also is responsible for getting all the managers able to talk to each other.

Sensor Manager

The Sensor Manager is in charge of fetching sensor data from the train controller over UART1. It provides a subscription service in which tasks can subscribe to sensor events.

Sensor Manager puts to use `BLPutC` which is a blocking `PutC` to UART1 which permits it to know to a much better degree when the actual sensor poll byte has been transmitted.

Switch Manager

The Switch Manager manages all switch related actions on the track. This includes sending and maintaining the switch states of the track.

Other tasks can query it to gain information about the switches or to switch the states.

Train Manager

The Train Manager facilitates the basic train commands sent from the shell. It also is responsible for getting trains initialized with the Prediction Manager.

Prediction Manager

The Prediction Manager attempts to maintain knowledge about trains on the track. It attempts to track trains by keeping track of where they were last seen and are expected to be seen next. It can handle a single sensor miss, but otherwise is quite fragile.

The Prediction Manager subscribes to the Sensor Manager for updates. When the sensor data changes, the Prediction Manager is alerted and updates its model of the train position and speed.

For TC 1, it also manages calculating pathing data and sending required switch changes to the Switch Manager.

Display Manager

We figured that it would be worth it to have a structured way to present data to the screen both for presentation as well as for debugging.

Terminal Manager

Terminal Manager manages a set of windows. Tasks can request a window from the Terminal Manager and output from the task can be redirected to either its own window or a common-to-all-tasks logging window.

There is currently support (not enabled currently) to route input to windows other than the shell, depending on the cursor focus.

The Terminal Manager attempts to smartly render the screen as to limit the amount of cursor movement byte-sequences needed.

Shell

Shell is just the first task which registers to the terminal manager which is configured to accept input. It is currently a monstrosity which handles the parsing and executing of all commands.

IOServer

Blocking PutC

We added a blocking version of the `PutC` function which is very useful in applications like the Sensor Manager for when it polls to the train controller.

TC 1

Stopping

After calibrating, we manually store the stopping distances. The stopping logic is relatively straightforward. We step backwards from the last node until we find the first sensor that is more than the stopping distance away from the node. Then after passing this sensor we can set a delay to stop at the correct position.

Calibrating

Method 1

Our intial method to calibrate train stopping distances was to have the train use a starting sensor as a measuring stick to try to land on a targer sensor further on down the track. We make an intial guess and check whether or not we overshoot the target. We then subtract or add to our guess and bring the train around to try again.

Obviously this is not very efficient and took quite a while to run, taking up to 5 or 6 iterations to achieve accurate results.

The plus side to this technique was that the results we get from the test are quite accurate.

Method 2

Our more efficient method was an inching strategy. Again, we start out with a guess. But this time if we overshoot the sensor we inch at a slow speed that we know to the next sensor, measuring how much time it takes. By using a slower speed we make the assumption of very little acceleration and thus we can calculate the stopping distance using the time and the speed.

This method is much more efficient and only takes a couple of iterations to get good results.

A problem, not with the method, is that moving at a slow speed means a greater chance of getting stuck, which ruins calibration results.

Pathing

Pathing is done using Dijstra's algorithm using our own heap implementation. There is nothing particularly special about the algorithm or its implementation.

It is thoroughly unit tested and we are fairly confident in its correctness.

When given a node to stop at, we calculate the path starting two sensors ahead of the train to the destination node. Then we check the switches in the path and set them in order starting from the destination node.

Resetting

We have a simple reset task which configures the switches to form a loop which we use to set ourselves up for the pathing and stopping.

Files/Hashes