Contents

T1 - Train Control 1 1

Train Control Application 1

Operating Instruction . . . 1
UI . . . 3
Initialization . . . 3
`TrackOracle` . . . 4
Interface . . . 4
Key data structures . . . 5
Implementation . . . 6
`CmdTask` . . . 7
`route` command . . . 8

Train Measurement - Collection 9

Initialization . . . 9
Logging and Exfiltration . . . 9
Tests . . . 10
Steady-State Velocity . . . 10
Stopping Distance . . . 10

Train Measurement - Analysis and Integration 11

Future Work 11

T1 - Train Control 1

or: How we Learned to Stop Worrying and Love the Trains

Train Control Application

Operating Instruction

Upon starting the application, the user is asked to enter which track the application is running on (either A or B).

After a brief initialization procedure, the display should look something similar to the following (albeit with a little more color).

```
CPU Usage (03%) ____________________________________________________________________________

[ 6797] train=58 spd=14 vel= 891mm/s pos=C14 ofst=0mm next=(A04 at t= 6862) error=-0.05s/-4
--------------------------------------------------------------------------------------------
>
--------------------------------------------------------------------------------------------
```

```
Enter the track you'll be using (A or B)
Spawned TrackOracleTask!
Initializing Track...
Stopping all trains...
Setting switch positions...
Track has been initialized!
Ready to accept commands!
t1 commands:
  addtr <train>                     - register a train with the track
  route <train> <sensor> <offset> - route train to given sensor + offset
  tr <train> <speed>              - set a train to a certain speed
  mkloop                          - reset track to have a loop
  q                               - quit

debug commands:
  help                   - prints this help message
  sw <branch> <c|s>      - set a branch to be (c)urved or (s)traight
  rv <train>             - reverse a train (MUST BE AT SPEED ZERO!)
  n <sensor> <offset>    - print the position, normalized
  path <sensor> <sensor> - calculate route between two sensors

max tick delay updated to 16 ticks
```

The display consists of the following components:

- A constantly updating readout of the system's CPU utilization, with a time-series graph showing past values using Unicode block chars
- A status readout for each of the trains currently registered with the application
- A command prompt
- A scrollable region to display program output and debug logs

The system supports the following commands:

T1 Assignment Specific Commands:

Command	Description
`addtr <train>`	Registers a train with the track
`route <train> <sensor> <offset>`	Routes a train to the specified sensor + offset
`mkloop`	Resets the track's branches to have a loop
`q`	Quits the program and return back to RedBoot
`tr <train> <train>`	Sets the speed of a given train (from 1 - 14, with 0 to stop)

Additional debug commands:

Command	Description
`help`	Prints a brief description of available commands
`sw <branch> <c|s>`	Sets a given track branch to be (s)traight or (c)urved
`rv <train>`	Reverses the direction of a given train
`n <sensor> <offset>`	"Normalizes" a sensor + offset on the track
`path <sensor> <sensor>`	Displays a route between two sensors on the track

UI

Instead of using raw calls to `Uart::{Putc,Putstr,Printf}`, we've modularized our UI calls into a dedicated UI module, which provides methods and macros for outputting data to the screen in a structured and orderly fashion. Some of the methods and macros the UI module exposes include:

- `log_line`, `log_success`, `log_warning`, `log_error`, etc…
  - `printf`-like macros which display appropriately colored output to the logging area
- `prompt_user(int uart, char* buf, size_t len)`
  - Responsible for drawing the command prompt, and waiting for user input.
- `render_train_descriptor(int uart, const train_descriptor_t& td)`
  - Renders a readout of a particular train's state
  - Includes information about: [6797] train=58 spd=14 vel= 891mm/s pos=C14 ofst=0mm next=(A04 at t= 6862) error=-0.05s/-44mm
    - Last observed timestamp
    - Speed
    - Assumed velocity
    - Last known position
    - Expected next sensor + time to reach said sensor
    - Last recorded position error
- `void render_initial_screen(int uart, const TermSize& term_size);`
  - Renders any static UI elements which do not change throughout the program's lifetime
    - e.g: the lines separating the command prompt for the logging area / train descriptors
  - Uses terminal dimensions to lay out UI elements
    - Dimensions are determined using a VT100 cursor position query

Initialization

The `FirstUserTask` begins by executing various mundane, program-agnostic setup routines: - Spawning the Uart and Clock servers - Clearing the terminal, and querying it's dimensions - Spawning the CPU Idle Time task

Once the basic system is up-and-running, FirstUserTask calls the `t1_main` method, which encapsulates any and all T1-specific initialization logic: - `render_initial_screen` is called, painting the T1 UI - Prompts the user to enter which track the program will be running on

At this point, the T1 application is ready to begin in earnest. `FirstUserTask` goes ahead and spawns the two central tasks which implement T1's functionality: The `TrackOracle` and `CmdTask`.

`TrackOracle`

The `TrackOracle` is the heart-and-soul of the train control program. It is responsible for maintaining an accurate model of the physical train track, including the state of each of the switches, and the position and velocity of each train on the track. It does so by constantly polling the track for sensor data, while simultaneously trying to interpolate and approximate the position of each train on the track. To ensure consistency with the physical realm, all interactions with the track (such as setting train speeds and switching branch directions) are mediated through the `TrackOracle`.

It's important to note the `TrackOracle` does *not* implement any high-level train control logic (such as routing)! Instead, then `TrackOracle` exposes a set of primitives which other tasks in the application can use to implement complex train control logic on top of.

Interface

As expected, the `TrackOracle` provides a set of methods for performing direct manipulation on the track, such as setting train speeds/direction, and branch direction. It also exposes a query interface, whereby other tasks can query the current state of a particular branch, or the calculated current position of a train on the track.

The `TrackOracle` also exposes a way to register physical trains with the internal train model: `calibrate_train`. When called, the `TrackOracle` will accelerate the specified train, and assume that the next sensor that is triggered was a result of the specified train. Once a train has been registered with the `TrackOracle`, subsequent sensor updates will be intelligently attributed to the registered trains (see the "Implementation" section for more details). If this method is *not* called, the `TrackOracle` has no way of attributing sensor activations to a particular train!

Lastly, and arguably the most interesting and "important" part of the `TrackOracle` interface is the `wake_at_pos(uint8_t train, train_pos_t pos)` primitive. Given a train ID and a position on the track, the `wake_at_pos` method blocks the calling task until the train hits the specified position on the track. If the track state changes such that the train will never reach the desired destination, the task is woken up and notified of the failure.

This powerful primitive serves as the foundation for the "smart" train control algorithms implemented in the rest of our application. The "Implementation" section below discusses how this primitive is implemented, but for a proper example of how it can be used to implement higher-level train control programs, see the "`route` command" section below.

Key data structures

The `TrackOracle` maintains an up-to-date view of the track, with the following fields:

`TrackGraph track` is a static graph of the track, which does not include the state of the branches. The `TrackGraph` class implements a bunch of useful methods used by the oracle: `next_sensor` computes the next sensor that a train would hit after a given sensor, depending on the state of the branches. `prev_sensor` is also implemented. `distance_between(sensor1, sensor2, branches)` calculates the distance between two sensors by walking along the path determined by the branches. `shortest_path(sensor1, sensor2)` runs Dijkstra's algorithm to find the shortest possible path from `sensor1` to `sensor2`. `normalize(pos)` takes a `track_pos_t` (a sensor plus an offset in millimeters), and turns it into a another position with the smallest possibly positive offset. For example, `C13+1000mm` turns into `E7+125mm`.

`Marklin::BranchState branches[BRANCHES_LEN];` is the (mutable) state of each branch on the track. As branches are switched, this array is updated, affecting future calls to certain methods on the `track_graph`.

`train_descriptor_t trains[MAX_TRAINS];` contains the state for every calibrated trains. `train_descriptor_t` is a large type, containing fixed parameters (such as a train's number and speed level), and other observed measurements such as velocity and position. Here is the struct in its current form:

```
struct train_descriptor_t {
    // Fixed params
    uint8_t id;
    uint8_t speed;
    bool reversed;
    bool lights;

    // Computed params
    int velocity;
    Marklin::track_pos_t pos;
    int pos_observed_at;
    int speed_changed_at;
    // when the train changes speed levels, accelerating is set to true and its
    // old speed level and velocity are remembered.
    bool accelerating;
    uint8_t old_speed;
    int old_velocity;

    bool has_next_sensor;
    Marklin::sensor_t next_sensor;
    int next_sensor_time;

    bool has_error;
    int time_error;
    int distance_error;
};
```

The next sensor and most recent error calculations are also stored on the train descriptor.

`std::optional<wakeup_t> blocked_task;` contains the task that is waiting to be woken up when a train reaches a certain position on the track. In the future we will likely extend this to a list of blocked tasks. `wakeup_t` is very simple:

```
struct wakeup_t {
    int tid;
    uint8_t train;
    Marklin::track_pos_t pos;
};
```

`int max_tick_delay;` measures the maximum time observed between successive calls to `tick()` this is important for real-time capabilities. `tick()` needs to know if it might not be called again before a deadline (such as waiting to wake up a blocked task). If it won't it needs to block on a `Clock::Delay` until the deadline, and then respond in time.

Implementation

Under the hood, `TrackOracle` spawns a task and exposes methods that `Send()` to that task. As such, an actual `TrackOracle` object is very small - containing only the tid of the underlying task. The task runs the usual `Receive()` loop, which calls functions on an instance of a private class called `TrackOracleImpl`, which contains the actual state of the track oracle. Almost every function `Reply()`s immediately, the major excepting being `wake_at_pos`, which is expected to reply only once a train reaches a certain position on the track.

The `TrackOracle` has a method called `calibrate_train`, which is used to get an idea where a train is on the track. `calibrate_train` interacts with the UI, prompting the user to place the train on the track and press `[ENTER]` once done. The oracle then stops the train, and then runs it a a slow speed

(speed level 8) until is hits a sensor. Once it does, it stops the train, and creates its `train_descriptor` based on the sensor it hit. With an up-to-date `train_descriptor`, the oracle can start reliably tracking the train's speed and position.

Two background tasks are running continuously - one to call `tick()` on the track oracle at every clock tick, and one to call `update_sensors()` in a loop.

`update_sensors()` runs a sensor query, which notably blocks the oracle for at least 70ms. If any sensors are returned, they are attributed to a train (based on the sensors a train is expected to hit next, given the track graph and positions of the switches), and the train's model is updated. Velocity is interpolated using an exponentially-weighted moving average that smooths out noise when cruising an a constant speed level. Here we also predict the next sensor that the train will hit, and the time at which we think we will hit it. If there was already a prediction for the sensor we just hit, we calculate the error (in terms of time and distance), and update the UI.

`tick()` runs computations that should update the model every time the clock ticks. There are two important routines: `interpolate_acceleration` and `check_scheduled_wakeups`. `interpolate_acceleration` continuously updates a train's velocity while it is changing speed levels, by interpolating it's starting velocity, expected final velocity, and the time that the train has been accelerating. `check_scheduled_wakeups` extrapolates the position of each train to check if a task waiting for the train to reach a position needs to be woken up.

This all comes together in the `wake_at_pos()` method, which puts a task to sleep until a certain train reaches a certain position. Through a combination of continuous calls to `update_sensors` and `tick`, we wake the calling task almost exactly when it expects to be woken up.

`CmdTask`

The `CmdTask` is the task responsible for reading, parsing, and executing user commands.

Specifically, it runs a very simple loop:

1. call the `prompt_user` UI method
2. parse the provided input as a `Command` structure
3. execute the requested command

The `Command` parser was ported over from K4, and has been extended with new commands.

Executing commands is done in-task, either inline, or by calling a separate method which implements the command's logic.

Most of the commands have fairly trivial implementations, whereby the command's arguments are marshalled to the corresponding `TrackOracle` method.

e.g: the `tr` command simply calls the `TrackOracle::set_train_speed` method, etc…

Instead of discussing each commands implementation, this documentation will only discuss the implementation details on non-trivial commands, namely: the `route` command.

`route` command

The `route` command implements the main deliverable of T1: the ability to route a train to a specific point on the track.

At a high level, this command is implemented as follows:

1. Query the train's current position on the track via the `TrackOracle`
2. Run Dijkstra's algorithm to find the shortest path from the train's current position to the specified position on the track
    - *NOTE:* if no path could be found, *and the train is currently stopped*, attempt to reverse the train, and re-run pathfinding.
3. Assuming a path was found, we traverse the path (given as a series of `train_node` structures) and determine which direction each of the branches along the path should have.
4. Switch all the branches to the correct position
    - *See the "Future Work" section below for some discussion on how we plan to transition to a "Just in Time" model.*
5. (optionally) If the path is quite long, set the train speed to 14, and schedule a `wake_at_pos` call to slow the train down as it approaches it's destination (to speed 8)
6. Schedule a `wake_at_pos` call to send a stop command once it's "stopping distance" away from the destination
    - The specific stopping distance is determined using our calibration data, and varies by train

In our current implementation, we treat two sensors at the same position on the track as two separate nodes, each with a different route to them. Namely, when routing to a sensor, the train will attempt to stop at the sensor "head-on", which varies by the sensor's direction.

Known Issues

- At the moment, the route command doesn't support "short" paths, namely, those where the distance to the destination node is shorter than a train's stopping distance. This is because we currently do not have a very detailed model of train acceleration and deceleration curved, which would be required to accurately "inch" a train forward towards it's destination.

Future work In our architecture, it should be fairly straightfoward to flip the switches "Just in Time" by using the `wake_at_pos` primitive, whereby the task is woken up just before it hits a branch. This is complicated by the fact that these `wake_at_pos` calls need to be scheduled with respect to the `wake_at_pos` calls which slow/stop the train. Interleaving these `wake_at_pos` calls requires knowing the distance each node is along the path, which is information that is not currently returned by our pathfinding method.

Train Measurement - Collection

To perform our measurements, we created a new target executable called `measure`. While this executable does in-fact run on top of our operating system, we never spawn more than one task, and only ever invoke the "Exit" and "Shutdown" syscalls. All I/O is done via polling and busy-waiting, as using the UART servers for I/O would add additional overhead when reading / writing data, interfering with the precise timing measurements.

The `measure` target encapsulates all the various tests we run (such as stopping distance, steady-state velocity measurements, etc. . .), and provides some basic test-agnostic initialization code:

Initialization

Upon launching `measure`, the program will ask the user which train they will be testing, and a "start speed" to perform automated tests from. This is useful, as the automated tests can occasionally fail mid-way through the test (e.g: lower speeds might stall the train), and need to be restarted from a set speed.

Additionally, the program asks the user if the track should be reset (i.e: stop all trains, and make a large outer loop), as back-to-back tests might not require resetting the track. The `measure` program does *not* use the CTS flag, and resetting the track can be quite slow.

`measure` uses Timer 3 running at 508khz for measurements. It does *not* need to manually reset the timer, as our kernel is responsible for setting up timer 3.

Logging and Exfiltration

Our tests dump measurements to the screen as JSON, with debug lines prepended with a `//` for easy filtering.

We opted to use gtkterm's built-in "save-to-log" functionality to save UART2 output to a file. While this did result in some stray redboot output being saved to our logs, it isn't too difficult to filter out these lines manually.

To get the logs off the train PCs, we simply SCP'd the raw log files to our home directories on the UW CS servers. After a bit of manual cleanup, we were left with some pristine JSON files that contained all the juicy measurement data that we collected.

Tests

We developed two different test routines to collect information about the trains: a steady-state velocity test, and a stopping distance test.

By the T1 submission deadline, we were able to collect the following data sets: - Steady-State Velocity for **speeds 5 through 14** for **all** trains - Stopping Distance measurements for **speeds 8 and 14** for **all** trains

Steady-State Velocity

This test measures the average steady-state velocity of each train at the given speed settings.

The test methodology is as follows:

1. Set the train going at the desired speed
2. Busy-poll for sensor updates, logging a line with the sensor + timestamp when a sensor is triggered
    - Line has the format of `R"({"event":"sensor","speed":%hhu,"sensor":"%c%hhu","time":%lu},`
3. After ~13 seconds (a fairly arbitrary amount), wait until one-last sensor is triggered
4. Bring the train up to speed 14, and wait for it to hit a particular sensor on the outer loop
5. Send the stop command to halt the train

Steps 4 and 5 are useful to automatically bring the train back to a specific known location on the train set. In fact, the first run of the steady-state velocity test doesn't actually record any data, and is solely performed to "calibrate" the train to an expected state!

Stopping Distance

This test measures the distance a train covers when it is stopped. The test gets the train up to a steady state-state speed, and then waits for a specific sensor to be hit. As soon as the sensor is observed, a `speed=0` command is sent, and the train comes to a stop. At this point the train's stopping distance can be measured.

An additional mechanism allows "stopping time" to be measured. The program records the time when the stop command is sent, and then waits for any key to be pressed by the user. When the user presses a key, the time is recorded and added to the outputted measurements. This way, is the user is attentive, they can press a key as soon as they see the train reach a stop, giving a decent measurement for stopping time.

Train Measurement - Analysis and Integration

We wrote a python script (the aptly named `process.py`) to process the raw measurement data and emit calibration data. The script parses the two measurement files for each train, and turns the raw data into expected velocities and stopping distances.

For velocity, the distances between each successive sensor is combined with the time difference to produce a bunch of average velocities. We found that aside from the first few data points (when a train is accelerating), we observed pretty consistent measurements. So for each train, at each speed level, the expected velocity is the median velocity observed during the measurements.

Stopping distance uses the mean of the measurements for each train/speed combination. We found stopping distances at speed 14 to be highly variable, but at speed 8 to be quite reliable. Both are included in the generated calibration code, but we only rely on the speed 8 stopping distance when trying to accurately stop a train.

The python scripts emits `src/raw_calibration/calibration.c` and an associated header file. This is then loaded by `src/assignments/t1/calibration.cc` where it the raw data is exposed through a friendly interface:

```
namespace Calibration {
int /* mm/s */ expected_velocity(uint8_t train, uint8_t speed);
int /* mm */ stopping_distance(uint8_t train, uint8_t speed);
int /* ticks */ stopping_time(uint8_t train, uint8_t speed);
int /* ticks */ acceleration_time(uint8_t train,
                                  int current_velocity,
                                  uint8_t target_speed);
}  // namespace Calibration
```

Future Work

The `TrackOracle`/`wake_at_pos` abstraction seems to be working well for us - our main limiting factor is time! Hopefully we can get just-in-time branch switching working quickly, and this will be essential for routing multiple trains at the same time.

If we want to collect more measurements, the pipeline of turning raw data into calibration code is already there, so we should be able to get new data into the program will relatively little effort. The hard part is the actual measurement.

We'd definitely like to clean up some of the code. `t1.cc` is a bit of a beast, and it's hard to discern what belongs in `t1.cc` vs the track oracle. For example, both tasks have a `TrackGraph` instance, because `t1.cc` does the actual pathfinding, while the `TrackOracle` implements `wake_at_pos`. We should figure out the real abstraction boundary, and move towards it.

Our UI could also use some work. We could probably turn a bunch of the noisy logs into fixed elements in the UI, so the log doesn't get spammed with data while a train is, for example, routing.