CS 452
Assignment 0

Mark Rada (`marada`)

Overview

The Märklin train set can be controlled by user input to a terminal mediated by an intermediate computer running my program. Commands issued into the terminal can manipulate trains and switches, and monitor sensors. Trains can start, stop, toggle lights, adjust speed, and in the case of train 51 the horn function can be activated. Switches can be toggled or explicitly set to the straight or curved state. The terminal displays information about current state of the trains and switches, as well as a short history of the most recent sensor activations. However, sensor activity cannot be controlled by the user in any way.

Operation Instructions

A pre-compiled kernel exists at `/u/cs452/tftp/ARM/marada/kernel.elf`, which can be loaded with default RedBoot command given in the course tutorial document:

```
load -b 0x00218000 -h 10.15.167.4 "ARM/marada/kernel.elf''; go
```

The source code for my program exists in `/u3/marada/cs452/`, which has the following list of files:

File	MD5 Hash
Makefile	`64a51ce2166817e27001e195440fab0e`
Rakefile	`a8a1dfca069d7461af991134d6e0c37e`
include/circular_buffer.h	`8415aa7d838e512ea9df93fff23fe1a4`
include/clock.h	`4985d550dcaa1aec022319c21fd67652`
include/input.h	`707c28f9c46f9b9e924da60ba784d352`
include/io.h	`fbb536f6b293075cc7fd8202e0fce6a4`
include/memory.h	`79a504175e6a2d1054dd64f6eef5fa68`
include/stdarg.h	`2107acfd362fb7033d0e1d03ad1955cd`
include/trains.h	`6ce082512ed5ef02f98cbe4690573434`
include/ts7200.h	`051c699e70f9b70a3f9f90d4ba2857f3`
include/unistd.h	`7df7748267eb108873fe9fbad929aebd`
orex.ld	`20b96d367138e77359a7914b582ca16a`
src/circular_buffer.c	`40df1cf8b4df34ffcb9041b1940438f1`
src/clock.c	`5d42f56f514ec626903e80a9c964044b`
src/input.c	`8d74ac14b0b671b06e289338cae95a74`
src/io.c	`65ec72cd20a849decbf9640b04016eb3`
src/main.c	`514f173b157bb485bad9e2b669063f7d`
src/memory.c	`edf9a7263a88ff505053874dfd527679`
src/trains.c	`be360422e28fc457c9c7c3aa29c03533`

The program can be compiled with the following command:

```
cd /u3/marada/cs452 && source /u3/marada/.cs452 && make clean && make
```

Which will produce a `kernel.elf` file in the same directory as the make file.

Program Operation

When the program begins it will initialize state on the track and draw the initial UI state.

The digital clock is located in top the left corner, and should begin counting immediately when the program begins. Underneath the clock is the table of switch states, followed by the list of recently activated sensors, and underneath that is a table of train states, and finally a circular table of debug and status messages at the bottom of the screen. The command prompt is on the right side of the sensor history list, and begins where the `>` character is located.

The train state table only shows information for the seven trains which were found in the lab at the time of writing: 43, 45, 47, 48, 49, 50, and 51. Only the trains in the table can be controlled and command parameters are validated to ensure that only those trains can be controlled. The train table shows the train speed as well as the current state of the train lights. A speed of `--` indicates that the train has been stopped.

The sensor list will scroll recently activated sensors, with the most recently activated sensor at the top of the list. Prolonged sensor activations will only appear in the table once, a sensor must return to the initial state and be activated again in order to add a new entry into the history list.

The debug messages area is a circular list, but is prefixed with a message number so that users can keep track of message order. The area is primarily used for debugging, but also contains feedback for some commands.

Startup will take a few seconds to send all initialization commands to the train controller, approximately 5 seconds based on testing done. However, commands can be entered and queued up during initialization and they will be executed as soon as possible.

Commands

The command prompt supports entering any alphanumeric character, as well as the space, backspace, and return keys. Behaviour of the command prompt after entering anything else is undefined. Once a command has been entered it will be echoed back in the debug console, parsed, and executed.

Commands will validate arguments if needed, and a train with a queued command cannot receive new commands until the queued command has been executed. Extra space between command arguments is tolerated, however the command buffer is not much longer than the maximum size of a command without extra spaces.

`tr <train> <speed>`

> Set the train speed as specified by the assignment. Valid train identifiers are any train listed in the train state table. Valid train speeds are `0-14`. Train speeds will be queued immediately and sent to the train controller as soon as possible.

`sw <switch> <state>`

> Explicitly change a switch state as specified by the assignment. Valid switch identifiers are `0-18` and `153-156`. Valid switch states are `S` and `C`, but they are not case-sensitive. The state change will be queued immediately and sent to the train controller as soon as possible.

`rv <train>`

> Reverse the train as specified by the assignment. Train reversals are queued commands which have a delay. Any other commands to the train while it is being reversed will be ignored until the reverse is complete. Messages will be printed to the debug console at each step of the reverse. As with the `tr` command, valid train identifiers are those listed in the train state table.
>
> A reverse will wait approximately 4 seconds after stopping the train to change direction and begin moving again at the previous speed. A delay of 4 seconds was chosen by experimentation, by find the maximum time a train would take to come to a full stop from maximum speed using the dial control.

`all stop`

Turn off the train console. This command may be useful if trains have collided, or if the horn on train 51 fails to turn off. This operation is queued and sent out as soon as possible.

`all start`

Turn on the train console. This command may be useful if trains have previously collided and short circuited the track. This command can restart the train controller without restarting the program. This operation is queued and sent as soon as possible.

`halt`

Set all train speeds to 0. This is equivalent to using the `tr` command for each train and setting the speed to 0. It results in a batch of command being queued and sent as soon as possible.

`go`

Set all train speeds to `6`. This is the compliment of the `halt` command and is hardcoded to set train speeds to 6.

`flip`

Toggle all switch states. It results in a large batch of commands being queued and sent as soon as possible.

`lt <train> <state>`

Set the train lights to the given state. As with the `tr` command, valid train identifiers are those listed in the train state table. Valid states are `on` and `off`, which are case-sensitive arguments. This command is queued immediately and sent out as soon as possible.

`horn`

Turn on the horn for train 51. This command takes no parameters and only works with train 51 as it was the only train that responded to functions during testing. The horn will turn on for approximately 10 seconds, but the response time of train 51 may delay starting or stopping of the horn command. The horn command is a queued command and will block all other commands to train 51 until the horn stops. Messages will be output to the debug console when the horn command is issued and cancelled.

`q`

Quit the program and return to RedBoot.

Program Operation

The polling loop of the program is preceded by initialization of the UARTs, clock timers, and buffers. Output buffers, to the train and terminal, are initialized with the initialization commands to set the track into a known state and UI to display the state respectively. The UART FIFO's are disabled and the baud rate divisors are correctly set. Additionally, any data left over in the receive buffer for the train UART, presumably from a previous program running, is flushed out.

Each iteration of the loop will first check the clock timers and update clock state as needed. Clock state is stored as both a single running timer and also as components to make the clock UI easier to display. The clock is based on Timer3 values set to 2kHz in free running mode, and I count deciseconds at every 200 timer tick milestone. Additionally, I also a have a counter that increments at every 50 timer ticks which is used by the train output logic to add a delay between train commands. Without the delay some commands were getting trashed, and using the regular clock ticks was too slow for practical use.

After the clock update, the receive buffer for the terminal is checked once. If a byte is available it is added to the input buffer for the terminal. All I/O buffers are implemented as circular buffers which perform no size checking, so it is possible for new data to overwrite old data which has not yet been written.

If there is a byte ready for output to the terminal in the write buffer, and the UART transmit buffer is empty with the clear-to-send bit not set, then the byte is written out to the wire.

Similarly, read and write checks are done for the train buffers. However, for the train output buffer, the busy bit is also checked before writing, as output was far too fast for the train controller otherwise.

After I/O is handled, the train operations are handled, and then user input is handled.

Train Processing

Train processing starts by checking the state of sensor polling. If the previous sensor poll has been completed then another sensor poll is immediately scheduled on the train controller output buffer.

If a sensor poll is already in progress then the next byte is taken from the train controller input buffer if it is available. The byte is then parsed. If a sensor has been tripped it will be added to the activation history list and output to the screen. The activated sensor will also be tracked until it resets to zero so that the same sensor activation is not entered into the history multiple times in cases where the sensor is held down.

The activation history is stored as a circular buffer, and the entire output table is printed for each update.

Finally, the delayed job queue for train operations is checked. However, the job queue is only checked once per second, and tracked against the running clock. The queue is a statically sized array of structures which store the relevant train identifier, operation, the remaining time until execution, and one integer of data specific to the operation. In the case of train reverse operations, the extra integer stores the original speed of the train being reversed. The queue is currently set to a size of 4, but the size is controlled by a C preprocessor macro at compile time.

The delay time in each struct is checked. Jobs with a delay of 0 are considered to be completed and are ignored, which avoids the need to clear out the other data in the structure. Jobs with a delay of 1 are ready to execute, and will have their relevant commands queued and sent out as soon as possible. All other delay values are decremented by 1.

User Input Processing

User input is copied from the input buffer as it comes in, and is placed into a line buffer. The line buffer is a simple character array sized at 16 bytes. Unlike the circular buffers, it will stop accepting input once it fills up instead of overwriting other buffered data.

Special characters and escape sequences are not handled, but the backspace key can be used to correct command typos in the standard fashion. The return key will confirm a command, which will then be echoed back in the debug console.

After a command is confirmed by a user it will be parsed and validated. Only commands that take parameters will have the parameters validated. For example, `tr` command for setting train speed will validate that the train number corresponds to a known train and that the speed falls in the valid range for speeds.

After the user input checks, the loop iteration is completed and begins again.

Questions

Profiling was performed was done using Timer4 hooked into the clock tick logic. Since Timer4 runs at a much higher frequency than the other timers, it allowed for reasonably precise measurements of the time taken for certain operations.

Loop Iteration

A full loop iteration was measured by saving the previous value for Timer4 and subtracting it from new value for Timer4 at the start of the loop. The differences were averaged and displayed on screen. A maximum value was also stored and displayed to show the worst case.

Testing was done on all 4 hardware setups in the lab running for at least 3 minutes each. In each test there was a phase of no user activity and a phase of high user activity, including flipping sensors on the track (when using computers attached to a track).

The worst case time for one loop was approximately 1 millisecond and occurred during startup initialization when the buffers were all busy trying to do I/O. After running for a couple of minutes the average time stabilized at approximately 0.2 milliseconds.

Clock Drift

Clock drift was measured against the stopwatch on my phone and compared against the digital clock display of my program.

My initial implementation showed that the clock would fall behind by approximately 1 second every 5 minutes. This drift seemed to be the approximately the same for all the machines in the lab.

My solution to this problem was to add a leap decisecond for every 30 seconds of measured time. Using this strategy the clock falls behind by less than 1 second in 5 minutes, though it is sill not perfect by any means.

Given that my measurement for the polling loop was on the scale of milliseconds, there is no chance that the clock logic was missing timer milestones for updating the clock and so the leap decisecond approach should be sufficient for short running times.

Sensor Response Time

The sensor response time was measured by storing the Timer4 value when the sensor poll command was queued and checking against the Timer4 value when the first byte is read from the input buffer.

As with loop iteration measurements, the value was added to a running total and the average was displayed on screen. After running for a couple of minutes with periods of high activity and no activity, the average stabilized at approximately 19 milliseconds. However, the initial values for the average were quite high, so it is possible that the average would have decreased to 18 milliseconds if the program was left running for a few more minutes. This value also will include at least two loop iterations of delay, so the true value is most likely somewhere between 17 and 18 milliseconds.

Bugs

1. Commands that take number parameters can take non-numeric values and interpret them as `0`.
2. Commands that take number parameters can overflow while parsing the text into an unsigned integer. The value will simply wrap around and may end up as a valid value.
3. The train delay queue is small and more than 4 operations will overwrite existing operations. While the size of the queue is configurable, I chose a small size to cut down on the amount of work done per polling loop iteration, and it is not realistic to manually control more than two or three trains.
4. Some sensors are stuck and will start off as showing that they were activated. I can think of no reasonable solution to handling broken sensors, so this quirk exists.
5. The `horn` command will sometimes be extremely delayed, despite the command going over the wire almost immediately. In some cases the command will fail all together. I was unable to confirm if this was the result of poor contact on parts of the track causing the signal to be missed or by some programming error on my part. Given that other commands seem to work correctly with train 51, I suspect there are some quirks with train functions which I did not discover in time. Though I did find that during manual control the function keys sometimes had to be held down in order to activate the functions. One solution would have been to send the commands multiple times, but I could not think of a sane way to implement this in time. Another solution might have been to bring in some cotton swabs to clean the track using the rubbing alcohol in the lab.
6. Reverse commands occasionally fail. This is rare, but seems to happen if the command is sent when the train is on part of the track with poor contact. There were instances when reversing using the speed dial on the train controller would have the same outcome, so I am led to believe that this is an issue with the hardware, though it could be the timing of my commands. Unlike the horn function, sending the command multiple could never work sanely since reverse is a toggling command.
7. Lights sometimes will turn on and turn off again right away. This is also very rare and seems to only happen if the command is sent while the train is over a portion of the track with poor contacts. I was able to use the lights to determine where the train would have poor contact by turning on the lights successfully and seeing when the lights flickered as the train went around the track.
8. It is possible to overflow the train output buffer by repeated use of the `flip` command by a fast typist. One solution would have been to make the queue larger, but this is a contrived case and the user deserves whatever bad things happen if they try to overflow the output buffer in this way.