CS452 Assignment 0

[redacted]

[redacted]

May 11, 2012

Program Operation

Path to Executable

/u/cs452/tftp/ARM/[redacted].elf

Program Display

On startup, the program displays the following information: the uptime of the process, the current position of the switches on the track, and a list of recently triggered sensors.

The process uptime is shown in the upper-right hand corner, as “Time: HH:MM:SS.S”.

The table of switch positions is in the upper left. Each switch is labelled “<switch_number> <switch_position>”, where the switch number is printed in decimal and the switch position is denoted with “S” for straight or “C” for curved.

Below the table of switch positions is a list of recently triggered sensors. When a sensor is triggered it is written to the bottom of this region. As new sensors are written the regions scrolls up. The most recent sensor is always at the bottom. Sensors are denoted “S<switch_number>”, where the switch number is printed in decimal.

Usage

Resetting the train controller, resetting the ts7200, and clearing the terminal screen (CTRL-L) are recommended to avoid interference from any previously run programs.

When the program starts, it begins by initializing all switches on the track to the curved position. This ensures that (1) switch position is known at startup time, and (2) usually causes the train to get stuck in a loop rather than derail.

While the switches are being reset “Resetting switches” is printed at the bottom of the screen with a progress indicator. Once the switches are fully reset the program becomes interactive.

Commands available to the user are:

```
// Excerpt from command_parser.h:

/*
 * Commands:
 *

 Move train:
 tr train_number:[1,80] speed:[1,14]
```

```
Reverse train:
rv train_number:[1,80]

Throw switch:
sw switch_number:[1,256] direction:['S','C']

Quit:
q

*/
```

When the user sends a newline signal to the terminal, the command they entered will be reprinted on the line above the terminal and the command will be executed. If the command entered was invalid, an error message is displayed instead.

Backspace works in the conventional way when entering commands. Other special characters (arrow keys, etc.) are not supported.

Program Structure

Overview

The core of the program is a polling loop found in main.c. The loop contains the following key elements:

- Writing commands to the train controller.
- Reading from the train controller and handling sensor information.
- Writing updates to the terminal.
- Reading commands from the terminal.

The core polling loop also performs some updates on a fixed schedule as part of the polling loop:

- The uptime count in the display is updated every tenth of a second.
- Whenever the train controller finishes sending track sensor data it is immediately queried again for new data. If the train controller failed to send complete sensor data, it is queried again after 100ms have elapsed.

Data Structures

Ring-buffers are used extensively to coordinate between the producers and consumers of the data being passed over the UARTS. All input is read into a buffer, and operations that read the input consume it from the buffer. Similarly, all operations that write data (either to the terminal or the train controller) write to a buffer, and a separate operation checks the buffer and writes to the UART if the device is ready.

No other non-trivial data structures were used. State used inside the polling loop (primarily timers and flags to coordinate different actions) was stored on the main method’s stack.

Terminal Display

Updating the console IO is done via a family of functions that write different data types (strings, ints, characters, …) to the COM2 output ring-buffer. 'Printf' style formatting functions were abandoned because they were complex and slow, and introduced undesirable performance overhead. The smaller IO operations are faster and more easily split up.

Command Parsing

My initial implementation of command parsing accumulated user input into a buffer, and when an end-of-line character was received parsed the buffer as a single operation. This parsing operation was quite expensive, and had a negative impact on the worst-case runtime of the polling loop.

My current implementation of command parsing uses a primitive but fast deterministic finite automaton. Each time a character is read from the terminal the state transition function is applied to the current command parse state. The transition function is quite fast, and the state machine approach allows the cost of parsing the command to be amortized over many iterations of the polling loop.

One problem with the state machine was that I did not initially implement it to support backspace, which makes the interface significantly more user friendly. Rather than rewrite the already complicated transition function, I added a stack (implemented as a ring-buffer) of previous command parse states. When a backspace character is received the buffer is used to walk back to the previous DFA state. This approach could be optimized but was found to have acceptable performance.

Sensor Display

As mentioned in the polling loop overview, the train controller is re-polled for sensor data as soon as it finishes returning data for the last query.

The current sensor state is saved in a buffer, and when the sensor data indicates that the sensor state has changed from low to high the triggered sensor is printed to the screen.

User Commands

Setting the train speed causes the appropriate train controller commands to be queued immediately and sent as fast as possible. (Sensor polling is rate-limited to avoid saturating the buffer, so train commands are always sent nearly instantaneously.)

Similarly, switch commands result in the appropriate commands being sent immediately.

The quit command causes the program to quit to redboot immediately. Buffer input and output are discarded.

When the reverse command is executed, train speed is set to zero. A timer is set for 5s in the future, which is a conservative estimate of the deceleration time of a train running at speed 14. When the timer expires the train is reversed and re-accelerated to its previous speed. If a second reverse command is sent before the train has reversed, it re-accelerates in its original direction. If the train speed is changed while it reverses the new speed will be honoured when the train re-accelerates after stopping and reversing.

If the user command exceeds 78 characters in length it is considered invalid and the command line is reset.

Source Code

Files and md5sums

```
linux016:~/cs452/a0>find /u9/[redacted]/cs452/a0 -type f | xargs md5sum

9af226f127c1fd759530cd45236c37b8  /u9/[redacted]/cs452/a0/include/ts7200.h
39e75d9b2a7c5438b4e7c56315273c86  /u9/[redacted]/cs452/a0/src/bwio.h
bece2707a33edeeb534784e6218c0d50  /u9/[redacted]/cs452/a0/src/bwio.c
5c6c3ba58cdb537ce9b1d610c3aba9ae  /u9/[redacted]/cs452/a0/src/Makefile
8200dca9d6284c9f0969e5f2d67d674f  /u9/[redacted]/cs452/a0/src/main.c
68b329da9893e34099c7d8ad5cb9c940  /u9/[redacted]/cs452/a0/src/main.h
49a5c582a40ccdf819744992ed67b2c0  /u9/[redacted]/cs452/a0/src/bprintf_test.cc
34506cfb1bd599f4b8ec59eea8cb36db  /u9/[redacted]/cs452/a0/src/clock.h
3d6905de6ba0f79f5b43e457d664e988  /u9/[redacted]/cs452/a0/src/orex.ld
969d8339502a5ef8367bef58d0188080  /u9/[redacted]/cs452/a0/src/bwioMake
c76d27e8c208f94d9c30f2cd7032d511  /u9/[redacted]/cs452/a0/src/circlebuf.c
721679c2e7ceb3279c8e596eb48153a4  /u9/[redacted]/cs452/a0/src/circlebuf.h
d07530b870eea68895df2a120f63b5a9  /u9/[redacted]/cs452/a0/src/command_parser.c
0f870fa326b8219a77a790469639d467  /u9/[redacted]/cs452/a0/src/circlebuf_test.cc
fb5de654ea0587884339b433c53bfd9a  /u9/[redacted]/cs452/a0/src/iotest.map
ec714e8419da15868bb9c3386929da43  /u9/[redacted]/cs452/a0/src/iotest.elf
ac40b75d9effa79c8e8cf134f83dcb71  /u9/[redacted]/cs452/a0/src/test/Makefile
7d10df0e75d88bbd6a479e59925a41ae  /u9/[redacted]/cs452/a0/src/bprintf.h
e64cfec4b78ca298e6a114d126ad9ec8  /u9/[redacted]/cs452/a0/src/bprintf.c
329e37641c820e1a4be7df042c36866b  /u9/[redacted]/cs452/a0/src/screen.h
b01dace33a93a34216dff98ee223cbca  /u9/[redacted]/cs452/a0/src/screen.c
ac838b58eca0e0216c82b13ac08f8ccd  /u9/[redacted]/cs452/a0/src/command_parser.h
3f7227aec469f5e207b702f51ab20032  /u9/[redacted]/cs452/a0/src/stddef.h
c3c382fdcafb396f64e79c749399fab7  /u9/[redacted]/cs452/a0/src/command_parser_test.cc
9352ff5ccda0f77fcd7541947f605623  /u9/[redacted]/cs452/a0/src/sensor_table.c
0351a393bef94b1bed7fd33a057ec19b  /u9/[redacted]/cs452/a0/src/sensor_table.h
```

Build Instructions

Run `make’ from /u9/[redacted]/cs452/a0/src.

Questions

i.

During development, the worst case runtimes of the elements in the polling loop were recorded for analysis. The worst case total running time of the polling loop was found to be 535 ticks of the 508kHz timer, or about 1.05 ms. The average-case polling time was less than 100 ticks.

Every iteration of the polling loop the uptime is compared to the displayed uptime. When the current decisecond increments the output to update the clock is buffered. The clock update requires 16 characters, including cursor repositioning. The COM2 UART connected to the terminal display runs at 115200 baud, so with an 8 bit message length and 1 stop bit we can expect to write bytes to COM2 every (508468.9 / (115200 / 10)) = 42 ticks of the 508kHz timer. COM2 output is written once per iteration of the polling loop, so we are limited by polling time rather than the UART speed.

In the worst case, updating the clock takes 16 iterations at 535 ticks per iteration, or 8560 ticks of the clock (16.8 milliseconds).

Since the on screen clock is specified as being accurate to 0.1s, and it is updated in worst-case time of 16.8ms, the polling loop is at least 6 times as fast as necessary to ensure we never lose ticks.

To ensure the clock doesn't drift, the value of the timer is read directly from the timer register and converted to seconds using the exact frequency specified in Table 5-3 of the EP93xx User's Guide (508.4689kHz) for every update. The uptime displayed on screen will not drift beyond the base +/- 40 seconds / month guarantee of the EP93xx clock.

ii.

A stripped-down polling loop was used to query the train controller for all sensor data, and then record the how long it took after writing the query to the COM1 UART to receive each of the ten bytes of sensor data.

Byte Offset	Time From Query (Ticks, 508kHz)	Time From Query (ms)	Time from last byte (ms)
0	8954	17.61	
1	11424	22.47	4.86
2	13893	27.32	4.85
3	16363	32.18	4.86
4	18852	37.08	4.9
5	21312	41.91	4.83
6	23801	46.81	4.9
7	26261	51.65	4.84
8	28751	56.54	4.89
9	31211	61.38	4.84

Since the COM1 UART operates at 2400 baud with 2 stop bits, it is possible to send / receive about (2400 / 11) = 218 bytes per second. So just sending the sensor query and transmitting the first byte back takes a minimum of 9.2ms. Transmitting all 11 bytes (1 query, 10 bytes of sensor data) takes a minimum of 50.4ms. So the overhead above the expected transport delay between querying and getting the first byte back suggests that the train controller takes about 8.4ms to query the sensors, after which it sends all of the results back roughly as fast as the UART allows. The delay between all the subsequent bytes of data is very close to the theoretical maximum of 4.6ms.