RoboTrike Functional Specification

Author

Sushant Sundaresh

UID 1917422

EE/CS 51, Caltech, Fall 2014

Last Modified: January 1, 2015

Revision History

10.13.2014. Chose high-level trike features, UI board keypad functions, and debug-mode interactivity. Defined display resolution, modes & data formats.

1.1.2014 After finishing HW9 and testing final code, revised UI-level functional spec to match actual functionality. One spec is given combining both boards.

User Interface Board Description

The user interacts with the UI board through a keypad to direct the RoboTrike via the commands defined in *UI Inputs*, below. There is no center-of-mass rotation. Linear movement is the target core function.

The user controls the reference orientation of the Trike (its forward) with 5 degree resolution, its direction of movement in 90 degree increments relative to this reference, and its speed. When asked to move the Trike will not stop until commanded. Thus to move in a circle, the user need only change the reference continuously while the Trike is moving, and the forward vector will rotate, though the Trike will not, about its center anyways. The user can also specify whether the Trike should halt (reset) or fire its laser.

The user receives only error messages and generic command strings via the display. Errors freeze UI, requiring user reset to restore function. Only one ‘catastrophic error,’ buffer overflow, is blocking. By design this last is never expected to happen.

The user is never told the actual speed or angle of the Trike because we do not know and it would not help. We focus on visual feedback, and so user inputs are sampled at 10Hz, repeating, to make input fast. Angle and speed changes are sufficiently small so as to appear smooth.

Global Variables None

Inputs

The UI board has a 16-key keypad. The keypad layout is as follows, with *Full Reset* on to top left. Empty keys have no function and are ignored. Key presses are sampled at 1kHz and debounced for 100ms, repeating, for a maximum keypress rate of 10Hz.

	Move forward		*Full Reset*
Move left		Move right	Fire Laser
	Move backward		Laser off
Increase Speed	Decrease Speed	Turn left (5º)	Turn right (5º)

Changes in angle are wrapping, modulo 360. Changes in speed truncate at 0 and 100%, which will be visually obvious to the user.

The UI board has a serial input from the Motor board, through which status updates and Motor-side error messages are received. Serial IO is done with an odd parity bit, single stop bit, and 8 data bits per transmission, and no handshaking, at 2400 baud.

Status updates include only the true reference angle, which allows us to verify on the UI board that our commands and perception of the Motor state are accurate.

Outputs

The UI board outputs sampled commands to its serial output cable, connected to the Motor control board. Commands are transmitted with arguments. These arguments will have leading zeros in all cases, and sign, and are delimited by ASCII carriage returns. Their grammar will be of the form "\CR (VDFOR) ± ##### \CR" matching that of Motor board status updates, which are of the form "\CR (DR) +##### \CR" where D,R have a different meaning. Invalid command characters (VDFOR) or (DR) are ignored, as is case. Numerically, arguments only need to be within a signed word though with repeated "increase speeds" one can certainly push the Trike speed to its hardcoded maximum, an unsigned word. These numbers are irrelevant here, as are the specific commands implementing the keypad functionality described.

The UI board maintains an 8-digit 7-segment display capable of roughly displaying a restricted set of ASCII characters. This display normally shows the last command key pressed, or an error message, scrolling at 0.5Hz. Error messages can originate from the UI board, in which case they are identified as such, or from the Motor board. In both cases errors can be due to serial errors, parsing errors (possibly caused by serial errors), or buffer-overflow style data loss (treated as a non-recoverable error, here).

Errors passed up via the control board are displayed with a 'cAr' identifier. UI board errors are preceded by the 'Ui' identifier.

The motor board drives the Trike motors and laser using parallel outputs; motors are driven using PWM control with 0.6% duty cycle resolution (*Algorithms)* with a PWM period of 32 ms (~30Hz). The laser is steadily on or off.

User Interface

Movement key presses and laser on/off need only be pressed once for the movement to persist. There is no time-out. Reset is simple enough; to reset is to halt the device and synchronize both board states back to a reference angle of 0 degrees.

Physically, tapping keys seems to be enough to meet the debounce period. Holding them is quite satisfying – speeds and angles ramp quickly and smoothly. High speeds result in motor whine, likely because the PWM frequency is ~30Hz, which is quite low.

Pressing multiple keys at once on the same row will guarantee that your input is ignored. Pressing invalid keys will not result in an error, but won't do anything else, either. So far as I can tell you can hold any button indefinitely without issue, though this seems very dependent on the memory map used (too low level for this writeup, but suffice it to say that not being careful about where the stack is placed results in a very tricky stack overflow that I've yet to pin down).

Full Reset halts the Trike and resets its reference angle to 0 on the UI board. It also tells the UI board to await a Trike status update confirming the reference angle change. Full reset is the only key press recorded after an error, until the system state has been reset. It displays the string 'reset,' and does not affect the laser status.

Rotate CCW or CW rotates the RoboTrike reference angle about its vertical axis more or less continuously, in 5 degree increments. If the Trike is moving while these are pressed (non-zero speed) the result will be a circular motion, sort of like a planet that has no spin orbiting a star. The results of this button press are verified, and discrepancies between the expected reference angle and the Trike status update after this command result in an error on the UI side.

Increase/Decrease Speed change the Trike's speed by an arbitrary fixed parameter related to the resolution of PWM and motor control (±500 units). This state is not really checked as the Motor board was not designed to transmit speed, mainly because the user is expected to have a lead foot and mash these two buttons regardless of what the status says.

Move X represents a signal to move along the current reference x'-y' axes of the Trike. You may move left, right, back, or forward relative to the reference angle. The results of these commands are checked – the status of the Trike is its current direction, and we ensure that this matches what we intended, else an error halts the UI board (to let the user know to reset).

Error Handling

Communication errors are both critical and detectable. A serial handshake is the only feedback we have!

On the motor board, a serial reception error results in the Trike halting, outputting a serial-specific error message instead of a status update, and then waiting for a full-reset command, parsing future input, but ignoring parse errors (since they could have been caused by the original serial error). Parsing is called on a non-blocking thread, and all error handling is

done in this thread, so errors retain their time-ordering. Command parse errors yield the same result as serial errors, and event buffer overflows (tracking errors, serial IO) halt the Trike permanently and repeatedly output a unique error status to the UI board. The requirement that a full-reset be pressed constitutes a partial handshake, for the UI board does look for our status update post-reset to verify that we received the reset, but if there is an error on the UI-side, the UI-board halts till the user resets, but the Motor board hears nothing of it, and keeps moving. It is assumed the user will respond fairly quickly; allowing user-independent communication faster than debounce times to starts to get troublesome without handshaking (see *limitations)*.

All noted errors are reported; practically this means the user sees the last one in a key-press interval.

Algorithms Holonomic drive controls how PWM duty cycles are calculated. Effectively, ignoring COM rotation, we pretend that torques scale linearly with traction forces and that the Trike is axially symmetric. Then we pretend the timescale of steady state motion is fast enough that velocity is proportional to torque, then we simply assume that torque is proportional to motor drive signal pulse-width if this pulse cycle is sampled fast enough. Treating the motors as couples acting at three roots of unity around the Trike, we can solve the problem in polar coordinates to find that knowing the required COM direction in 2D space is enough to know how each of the motors should move. The speed then just becomes a proportionality constant where we arbitrarily set a full unsigned word to mean a 100% duty cycle.

The motor and UI parsers, and the keypad, are implemented using finite-state-machine equivalents, but they were not done with code tables, but rather in if-then-else form.

Data Structures Circular queues are used for serial buffering and event buffering.

UI Board Limitations Boards must be loaded by the debugger with the code to run and power-off means the code must be re-installed.

Without feedback from sensors on movement direction or speed, the control board will not be able to correct for dragging wheels or obstacle impact, or respond gracefully in the case of drive failure.

IO and UI timing matter a lot. Motor PWM control timing matters as well. For example, on the Motor board, the PWM resolution of ~0.2ms/32ms ~ 0.6% duty cycle is set by the fact that the core code I've written consistently takes between 90-100us to run, and there has to be time for other things to run in between! However, we can actually stick quite close to this (within 0.1%) by allowing PWM timers to interrupt serial IO. Then, however, we need to make sure serial IO is not lost because we've prioritized PWM, which would result in error loops. This slows down serial IO to 2400 baud, or 300 char/s, forcing us to limit the status update length and keypad functionality so that a keypress and its status update happen before the next keypress, so that we can easily keep a running state estimate on both boards synched. On the UI board, this forces serial IO to have priority over display mux/ keypad debouncing, whose timing now gets jittery and less precise (though it doesn't matter to the user).

All of this, mind you, was without serial handshaking, which would make error-handling way more robust (you'd never lose characters again, or need to reset after every serial error), and there would be no cause for non-resolvable blocking errors, which only occur in my design because when they occur, it is not possible to know without handshaking what was lost (or if an error even occurred).

Still, with my timing, commands cannot overlap even in the worst case, and so we will always know if our state estimate is valid before we try to issue a new command to the Motor control board, and most errors are very quickly resolved (catastrophic errors haven't ever been seen).

UI Board Known Bugs This is a resolved bug, but it still bothers me. When I located my code using NOIC AD(SM(CODE 1000H, DATA 400H, STACK 7000H)) but then addressed SS and DS via the same DGROUP, I was seeing something that looked incredibly like a stack overflow to me. I was not able to pinpoint the error despite considerable effort and ended up realizing that if I just forced STACK to actually point to the value I located it to, 7000H, then there'd be no way I'd overflow into anything important (if that was what was happening). Running the code that way resulted in the main issue (couldn't hold keys for longer than 44 repeats during serial IO) going away instantly and for all intents and purposes now, there is no problem.