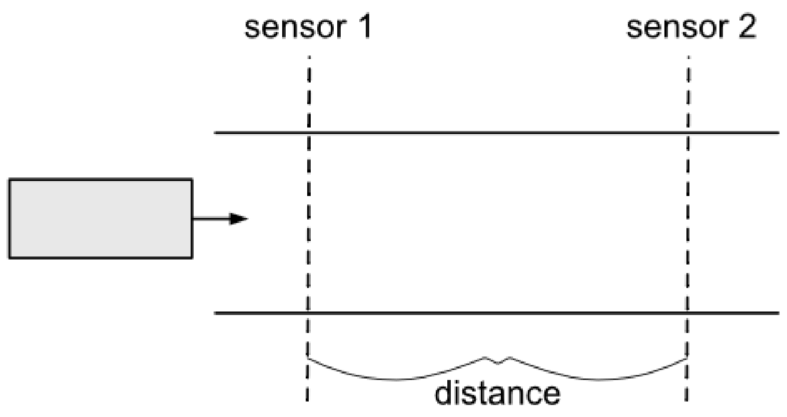
Photogate Speedometer

Video overview: https://youtu.be/JHX56pLNx40

Code: https://github.com/aams86/Speedometer

Application Description:



This device consists of two sets of ir transmitters and receivers, that are a set distance apart. Each transmitter is pointed towards the receiver it is paired with. When the beam between the transmitter and receiver is blocked, the device begins tracking time, and when the second beam is blocked, the device uses the set distance and the measured time to calculate the speed of the object that had traveled past the sensors. The speed is then displayed on an OLED display. The pcbs would be designed in such a way that it could be placed in enclosures of different shapes, so you could attach it to a hot wheels track to measure the speed of a car, and have the flexibility to place it in other places where you might want to measure the speed of something.

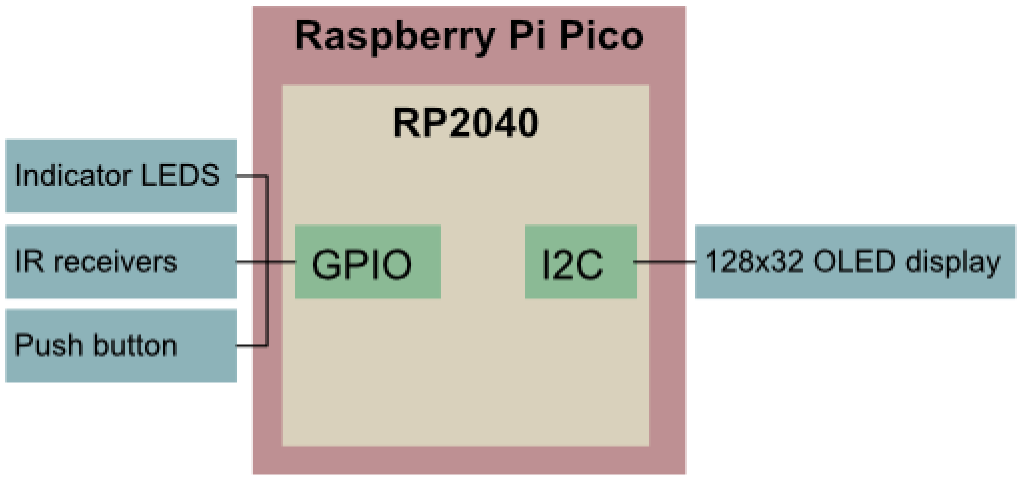
Hardware Description:

This system is built around the RP2040 microcontroller, and is currently built using the raspberry pi pico development board. The schematic can be accessed here: schematic. The cad files can be accessed here: CAD files. The full list of components is provided below:

- Raspberry Pi Pico
- 128x32 I2C OLED Display
- 2x3mm LEDs
- 2x3mm IR LEDs
- 3mm IR receivers
- 2x330Ω resistors
- 2x1M resistors

- Push button

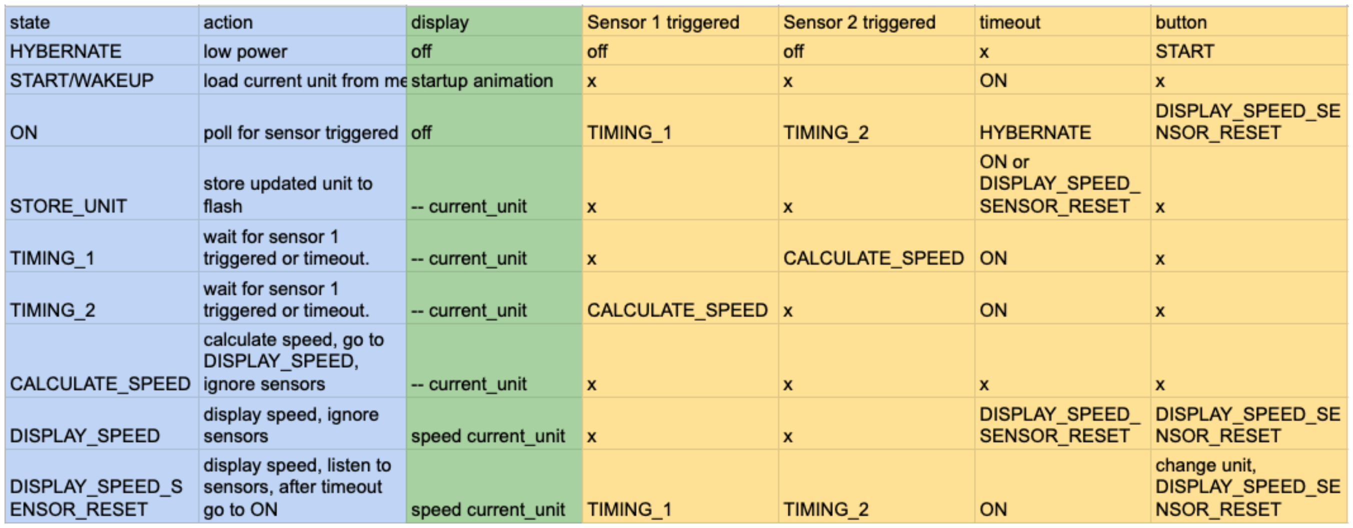
Hardware Block Diagram:



Software Description:

The starts up, initializing peripherals, and then transitions to an on state, where the display is cleared and there interrupts are set for each of the sensors. When one sensor is triggered, the speedometer transitions to one of the two timing states where it waits for the other sensor to trigger. During the transition to this state, a timeout alarm is set in the case of a false trigger, and the second sensor is reset (I was having an issue where both sensor flags were triggering, causing a 6us delay to be measured as the time between sensors). If the timeout is triggered, the device returns to the on state, or if the sensor is triggered, the device goes to the display speed state. Speed is now calculated and displayed on the OLED display. At this point new timers are set, and after a short timeout, the sensors are reenabled, and after a longer timeout the display turns off. Each state transition is handled by a function, to ensure the state gets initialized correctly regardless of the state it is transitioning from. If the button is pressed in the on state, the device will transition to the display on mode, and if the display is already on, it will change the displayed unit and reset the display timeout alarm. Currently the hibernate state is not implemented, however, after an additional timeout in the on state the plan is to have the device transition to the hibernate mode.

State Machine Diagram:



state	action	display	Sensor 1 triggered	Sensor 2 triggered	timeout	button
HYBERNATE	low power	off	off	off	x	START
START/WAKEUP	load current unit from me	startup animation	x	x	ON	x
ON	poll for sensor triggered	off	TIMING_1	TIMING_2	HYBERNATE	DISPLAY_SPEED_SENSOR_RESET
STORE_UNIT	store updated unit to flash	-- current_unit	x	x	ON or DISPLAY_SPEED_SENSOR_RESET	x
TIMING_1	wait for sensor 1 triggered or timeout.	-- current_unit	x	CALCULATE_SPEED	ON	x
TIMING_2	wait for sensor 1 triggered or timeout.	-- current_unit	CALCULATE_SPEED	x	ON	x
CALCULATE_SPEED	calculate speed, go to DISPLAY_SPEED, ignore sensors	-- current_unit	x	x	x	x
DISPLAY_SPEED	display speed, ignore sensors	speed current_unit	x	x	DISPLAY_SPEED_SENSOR_RESET	DISPLAY_SPEED_SENSOR_RESET
DISPLAY_SPEED_SENSOR_RESET	display speed, listen to sensors, after timeout go to ON	speed current_unit	TIMING_1	TIMING_2	ON	change unit, DISPLAY_SPEED_SENSOR_RESET

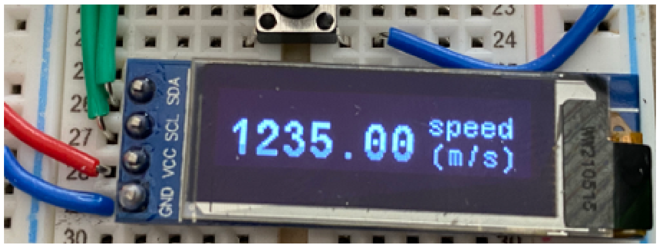
Written by me:

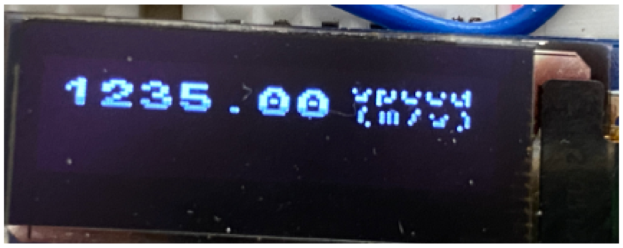
- Speedometer.c: This file contains the main function, calls the initialization functions, and calls the manage_states() function.
- speedometerStateHandlers.h, speedometerStateHandlers.c: The majority of the code written for this project is contained in these files. They contain the function calls to initialize the peripherals, and the functions to handle the states and the state transitions. The callback functions for the gpio and timer interrupts are defined here as well. The manage_states() function calls handler functions for each state. The handler functions mainly poll to see if flags have been triggered and call the corresponding state transition function if one has. There is an enum for the speedometer states, and the gpio pins and timeout lengths are defined within these files.
- speedometerAlarms.h, speedometerAlarms.c: All timeouts in the states are handled using the timer alarms provided by the rp2040 c/c++ sdk. The alarms are managed by functions written in the files referenced here. There is a struct for containing alarm ids and alarm flags. The functions here are for setting alarms, initializing the alarm callback, and for disabling alarms.
- speedometerDisplay.h, speedometerDisplay.c: Functions for controlling the display are contained in these files. There is an enum for selecting the current unit, although it is not really used. The functions here clear the display, and display the speed passed to the function along with a unit ID. The display_speed() function also does some calculations to update the precision of the display depending on the magnitude of the speed it is currently displaying.

- speedometerMemory.h, speedometerMemory.c: These files contain incomplete code. They represent the beginnings of what will be used to store/load the current setting for the unit if the unit has been changed.

Modified by me:

- ssd1306.h, ssd1306.c, fonts.h, fonts.c: These documents were originally created by: https://github.com/MaJerle/ and are licensed by the GNU General Public License. I modified the ssd1306.c to get it to work with the RP2040 i2c. The library was only written for 128x64 displays, and so I ran into issues when I attempted to use the smaller display. The buffer is organized in vertical pages, ordered horizontally (kind of a zigzag pattern). The first 8 rows of pixels are contained in the first set of pages, then the next 8 in the next row. For the 32 bit display, instead of using the first 4 rows, it used all 8 rows, but only reading the odd numbered bit from each page. In retrospect, there is likely a setting for the driver to easily solve this problem, but I learned a whole lot about it figuring this all out. See pg 25 of this document for more information about the layout of the displays: ssd1306 datasheet.



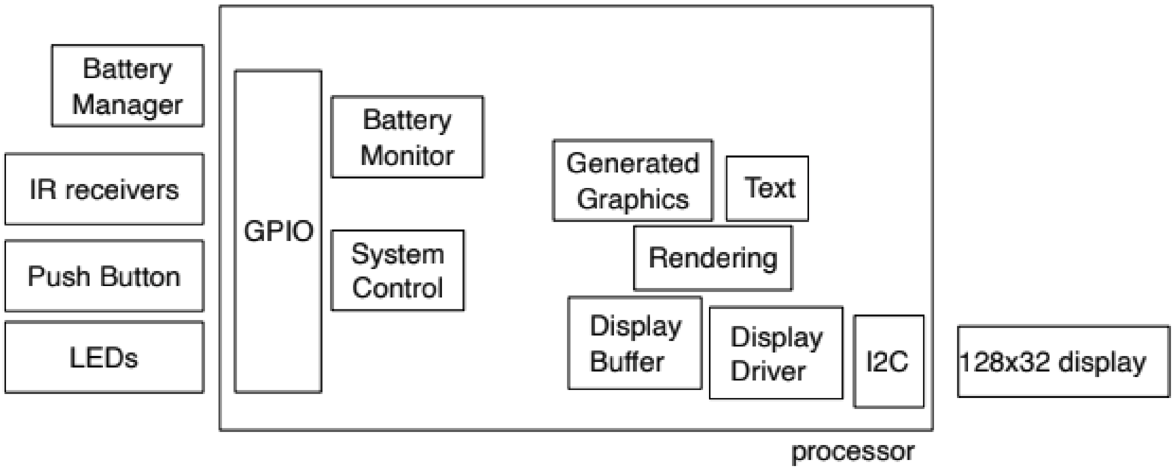


From other sources:

- https://raspberrypi.github.io/pico-sdk-doxygen/ The code utilizes the RP2040 SDK to perform most of the operations. This code has the BSD-3-Clause License. No modifications were made to the pico sdk for this project. It relies mainly on the following tools: gpio configuration, setting and reading gpio, gpio interrupts, alarm interrupts, current clock value, and i2c.

Diagram(s) of the Architecture

Software Block Diagram:



Build Instructions:

- The system was built using cmake and the arm-none-eabi-gcc compiler from vscode. Further build instructions can be found here: https://raspberrypi.github.io/pico-sdk-doxygen/index.html
- To debug the speedometer with the raspberry pi pico, a second pico is needed. Set up instructions are located here: getting-started-with-pico.pdf, on page 57. The "picoprobe" uses a modified version of openOCD. The set up was a little difficult at first, but once completed, was easy to use.

Future:

- Here are the ways I would like to modify the speedometer in the future:
  - Button debouncing: The current method for button debouncing is very limited, and needs some attention. This is the first item to fix when time is available to revisit this project.
  - Long term flash storage to remember the setting: This is not implemented, but would be a useful feature to have. When you set the unit to the one you want, it will remember it for the next time the device is powered on.
  - Hibernate mode: Currently there is no low power mode. The device should power down to a low power mode when not in use for an extended period of time. It will wake up from the button being pressed.
  - Battery management: I have to decide how I want to power it. This is a device that will be used for short intervals in bursts, randomly a few times a year. I will likely need to add a power switch to conserve power when not in use. Especially if I stick with the rp2040, as the sleep state is fairly low, but not low enough that it will last for more than a year or two on alkaline batteries.
  - The enclosure should be modified to work with modular connectors, like hot wheels tracks, maybe wooden train tracks, other toys?.
- Here is what needs to be completed before this device is ready for production:
  - The rp2040 was fun to work with, but likely uses too much power for this purpose. I would also need something I could hand solder depending on the production size. I am thinking I will look at using the ATTINY824. It has sufficient GPIO, and enough program memory to hold the font and the sprintf function. It also has EEPROM, which is a plus.
  - The display is cheap and doesn't use too much power, but it may be worth exploring other options to see what else is out there.
  - I will need to do testing on the output for the speed calculation, as well as the timer measurements. It is possible I will need to add an external crystal oscillator to improve accuracy, although I think I may be able to get away with the timers on the chips. I should also explore how much additional effort would be necessary to do some timer calibration and if it significantly impacts the measurements. The temperatures this device will be used in should generally be pretty close to room temp, so I do not foresee needing to do any temperature adjustments.
  - The production version will likely run on lower voltage. The hardware needs to be tested to see that the interrupts still trigger as expected when operating at the lower voltage. Likely different resistors are necessary for the LEDs as well.

    - Depending on what is decided for the power solution, the IR transmitters should be modified so they are powered by GPIO pins. This way they can be turned off when the device goes into its low power mode.
- Other features:
    - Potentially I could add bluetooth capability to this device, maybe to offload data. It doesn't seem like it makes sense as a feature for this particular device, though.
    - I could add a second button for more ability to configure, and for extra mode. Maybe a lap mode or something like that. Could also store in RAM recent results, so pressing the second button could pull results from the memory and display them.

Grading:

Self assessment:

Criteria	Score	Rationale
Project meets minimum project goals	1	My project does not meet the minimum requirement for peripherals
Completeness of deliverables	3	I think I am between a 2 and a 3 here, but I am going to round up. I may have left a few things out, and the video is not my best work.
Clear intentions and working code	2.5	The code functions, although there are some discrepancies between the software block diagram and the actual implementation. The hardware block diagram leaves out some components.
Reusing code	2.5	No versioning information included, and licensing information for this product is not included.
Originality and scope of goals	2	Not sure on this one. I think it is pretty awesome, but could certainly see why others wouldn't. I am not sure how many points I lose here due to the missing requirements.