Reaction Tester Mini Project

KNN - Embedded Systems

By Kian Bay

Intro

Reaction Tester Mini-Project.

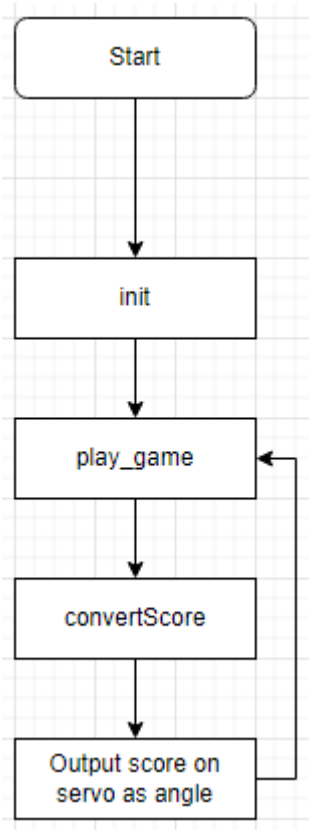
Design and implement a reaction test game in ATMEGA1284

The purpose is to test the users reaction time and display this on an mechanical display (Meter)

To test the reaction a switch is used to measure the time between a visual start signal(ex: LED) and when the user hits the switch.

Design

High Level Flow



Flow description

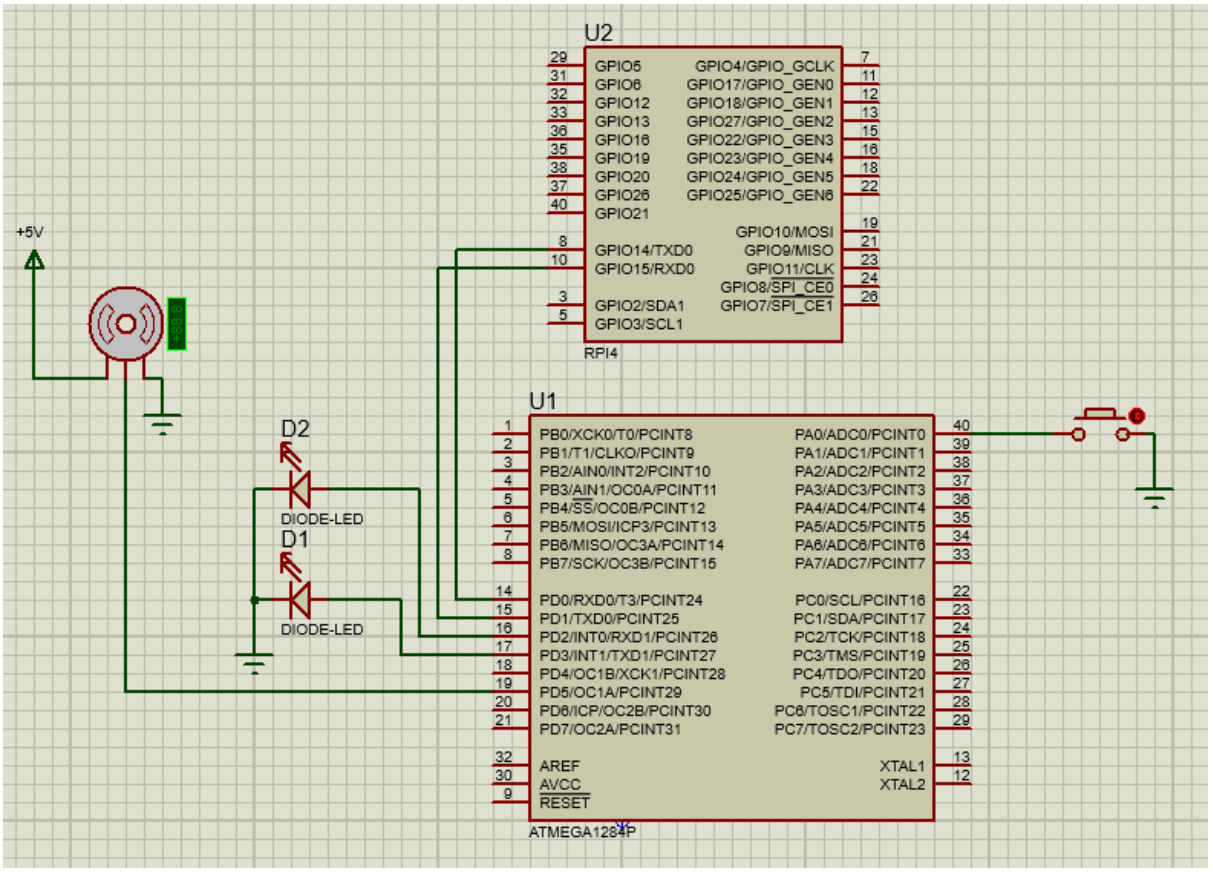
When the device is started, the init function runs all the initialization routines for pin and timer setup. Once done, the program enters the main loop of playing the game, converting the score to a usable PWM value for the servo to display, and then use the servo to display the score.

Since the score is the time in milliseconds it has taken the player to react, a low score means faster reaction, which is what the player is aiming for in this game. The servo will start

at the position of the lowest score, and when the game is over the servo has to move a larger distance if the score is low, increasing the dramatic effect of scoring a fast reaction.

Schematic

Only few parts are needed for this project: a solderino (depicted as 1284p), servo, some LEDs to indicate certain things to the user and for debugging, a button to play the game.



Finally a raspberry pi 4 will be used to power the solderino (unfortunately not seen in schematic), and I have hooked up the RX/TX lines as well in case things should get really fun in the future.

Interfacing with and testing servo

Since every servo apparently differs slightly from each other, and no real standard being in place, it seemed like an obvious first step to properly figure out how to control and test the servo before moving forward with the business logic.

For servo control it seems very convenient to use timer1 in Phase Correct PWM mode. We will clear timer on compare and set a top value that is the whole PWM period, and then manipulate the OCRnA value to toggle the OCnA pin on the board, allowing for very precise hardware PWM generation.
Something really smart about using this mode on a 16MHz oscillator is how the 8 prescaler lines up the values we need.

First off, to get a 50Hz signal from our prescaled timer:

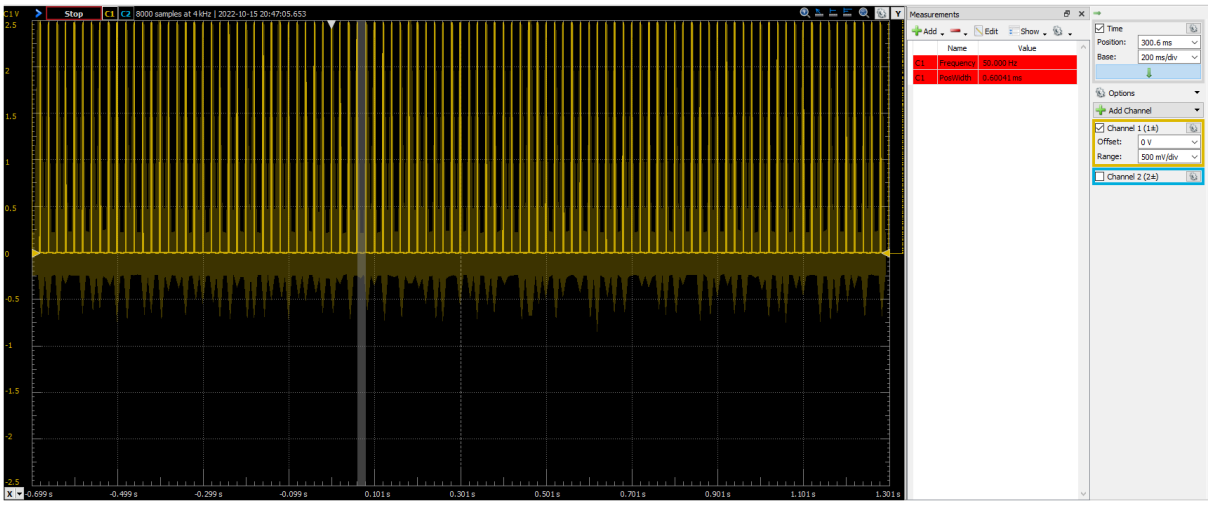
$$Ticks = \frac{BaseClk/Prescaler}{Freq} - 1 = \frac{16MHz/8}{50} - 1 = 39999$$

In this mode though, we count to TOP and back to BOTTOM, so our TOP value should be

$$Top = \frac{\left(\frac{BaseClk/Prescaler}{Freq}\right)}{2} - 1 = \frac{\left(\frac{16MHz/8}{50}\right)}{2} - 1 = 19999$$

Interestingly, a 50Hz signal also means there is 20ms between every pulse, and looking above we can see it's actually something we can work with as a count in µs!
So, given a servo generally works with on a 50Hz frequency with a pulse width of 1-2ms, It means we can directly use µs as the OCRnA value. Now we just need to find the min and max positions of the servo and define these as a macro for the starting point of the real program.
The found values were 460µs = 0.46ms and 2010µs = 2.01ms.
Since the servo did move as expected, we want to totally verify this by using a scope with the probe reading the pulses we would otherwise send to the servo

Using the value OCR1A = 597:



We can see from the measurements tab that the frequency is very nice at 50.000Hz and the positive pulse width is 0.600ms, which is a mere 3µs from the target. This is probably precise enough for our needs.

Designing game loop

The general flow of the game is:

- An LED lights up to indicate the user should be ready
- Some (currently very very random and unpredictable…) elapses, 1-3 seconds
- When LED turns off, a score timer begin and the player should press the button as quickly as possible
- Until the button is pressed, the score timer keeps tick tocking…

- On button press, the timer stops and is returned as a score. Score = ms taken to react
- The score is then mapped to a valid servo degree ([0-180])
- The score is then mapped to a valid PWM value ([min, max] from above)
- OCnA is set to this PWM value and will move to show an angular representation of the score
- A small delay is introduced before looping back to the top and starting a new game

Defining score

Since the score is one of the most important things in a game like this, it's important enough thought is put into defining the range and usage of it. According to internet sources, reaction times around the 200ms range are the fastest, so minimum close to this would seem ideal. On the other extreme, if your reaction time is around half a second, then maybe you were not ready to play or the game is not for you. We'll define the minimum, ie. the best score as 150(ms), and the maximum, ie. the worst score, as 500(ms).

Mapping the values

In order to convert the score, we need to move it from one domain into another, in this case a degree. The theory is that we can create a linear relationship between these two ranges by using the min and max values of each, and then use score as an input for the calculation.

Initially we create an abstract function map()

```
int map(unsigned long x, int minIN, int maxIN, int minOUT, int maxOUT) {
    // Mapping input from one domain to an output of another

    return (((x - minIN) * (maxOUT - minOUT)) / (maxIN - minIN)) +minOUT;
}
```

And then just utilise this inside a more targeted function to make the higher level functions more readable

```
int convertScoreToDeg(unsigned long score) {
    return map(score, SCORE_TIME_MIN, SCORE_TIME_MAX, SERVO_ANGLE_MIN, SERVO_ANGLE_MAX);
}
```

Let's say we got a score of 237, and we know the min and max scores possible are 150 and 500 respectively. We also know that min and max degrees are 0 and 180, so our call to the second function could be convertScoreToDeg(237) and it would simply grab the inside variables from the defined macro.

(In a later moment of brilliance, it was realised that the easiest way to do this is not by mapping twice, but simply mapping directly from score->PWM. Also, interesting stuff on the unsigned long later on!)

Creating a method to keep track of time

Since there is no millis() readily available, we need to create this ourselves. This functionality should be a counter incrementing in the background at an interval of 1ms. We can then call a function to return the current millis_value and store this in a variable, allowing us to keep track of time.
To create this we will use timer0 and compare match to trigger an interrupt every ms and increment the counter.
The required OCRnA value to trigger this interrupt is found by

$$OCRx = \frac{TF_{CPU}}{N} - 1$$

Where T is required time in seconds, F_CPU is base clock, N is prescaler.
Any prescaler 64 or higher allows us to keep this in 8bit

$$OCRx = \frac{0.001s * 16MHz}{64} - 1 = 249$$

```
ISR(TIMER0_COMPA_vect) {
    millis_value++;
    // PORTD ^= (1 << PD2 ); Used to verify interval using scope
}
```

Now this global value is updated every time this interrupt executes

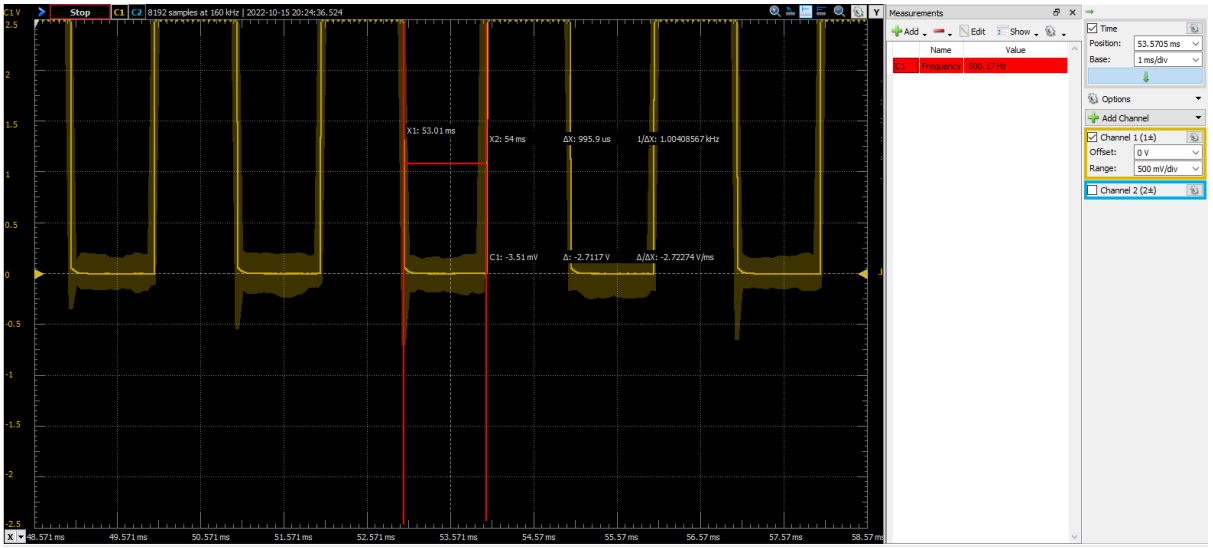
```
unsigned long millis(void) {
    unsigned long m;

    //This runs cli() before the statement, then sei() after, (allegedly) allowing us to safely read the variable manipulated by the interrupt
    ATOMIC_BLOCK(ATOMIC_FORCEON) {
        m = millis_value;
    }

    return m;
}
```

And we can query the current value using this function. We also make sure to not allow interrupts here when reading the variable.

Testing by probing with the scope on PD2 while this is running

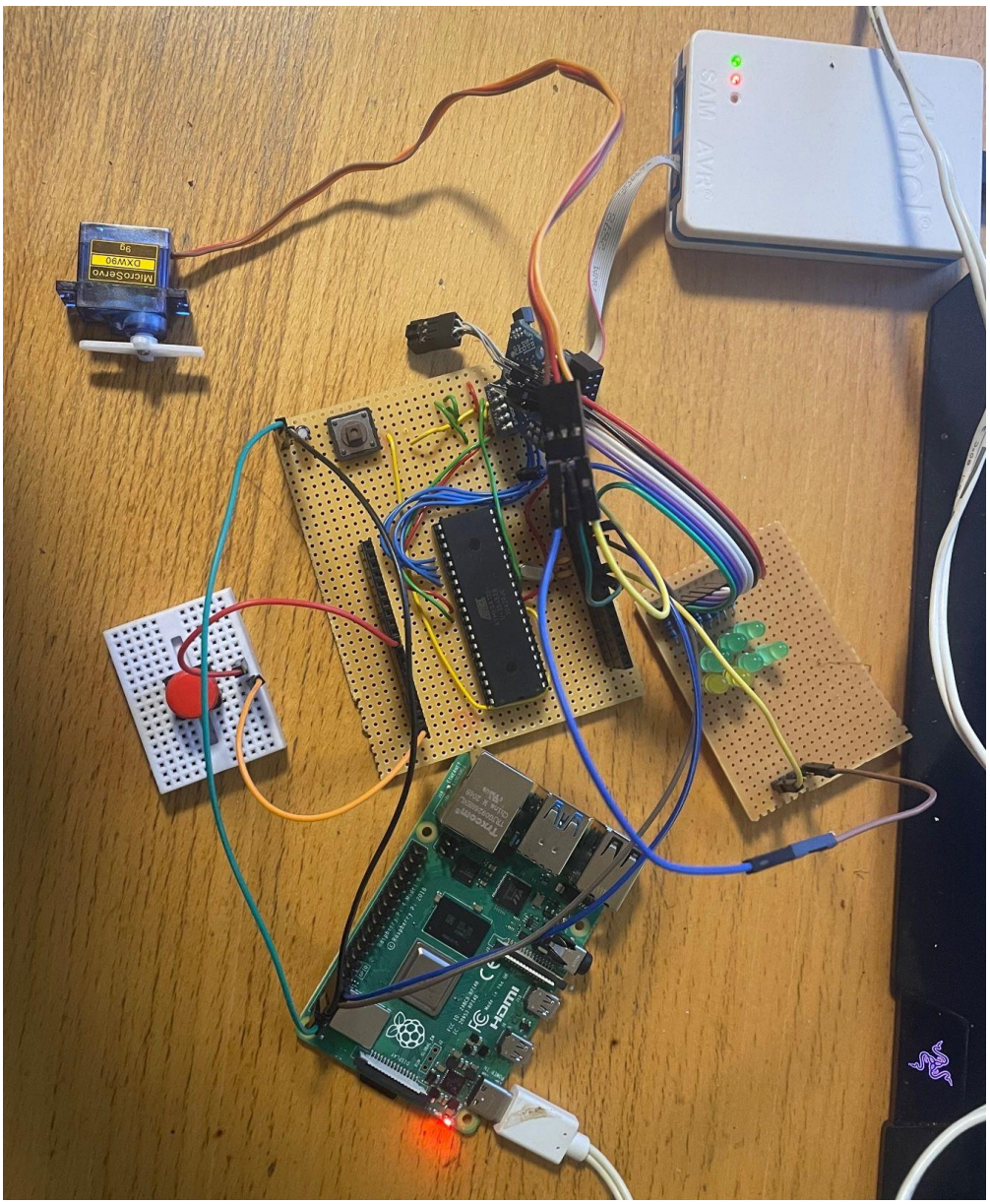


Since t_on and t_off should both be 1ms, the measurements should show a frequency of 500Hz since

$t_{wave} = t_{on} + t_{off} = 2ms$

Test looks good with ~995.9µs width and measurement showing 500.17Hz

Implementation



Demo

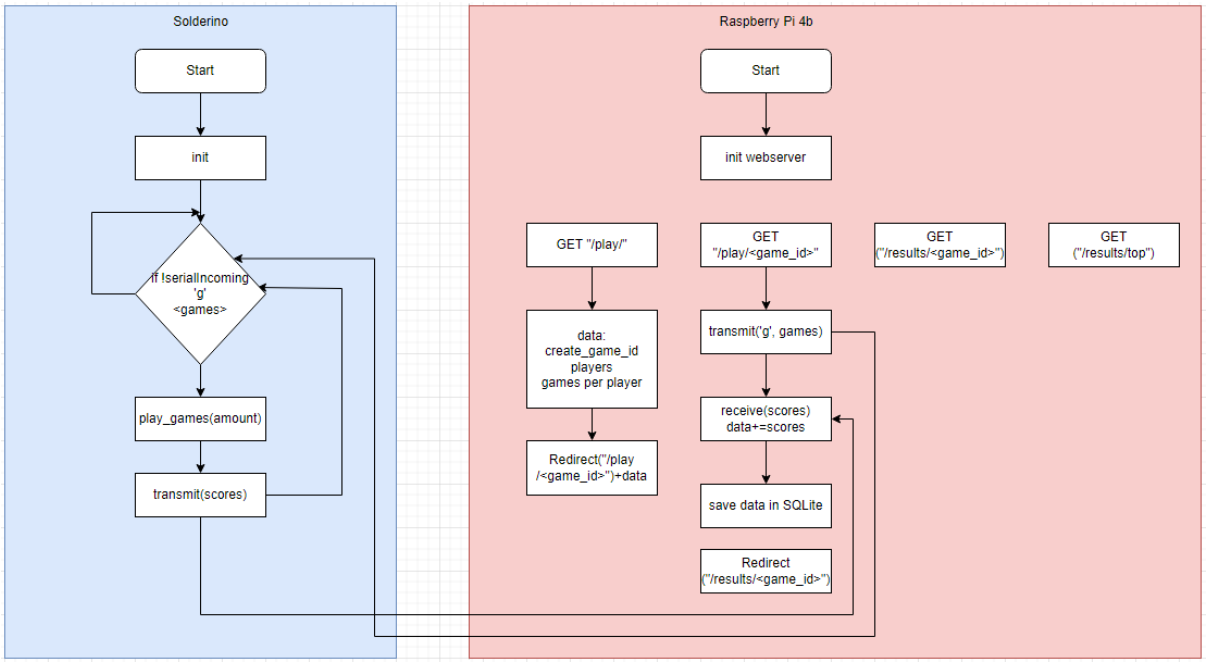
https://www.youtube.com/watch?v=1cTCH2hyYeQ

Demo of core functionality with a game length of 4.
As seen, the servo starts from the correct position for every attempt, and slower reaction scores mean there's not much movement. A really fast score of 150ms would make the servo move the full 180 degrees.

But what if…?

Right now the solderino runs a very simple loop without much functionality aside from the actual game. An idea that came to mind was, what if we gave some persistence to some data, namely the scores, and what if we allowed a way to keep track of good scores, or maybe even allow playing versus other players too?

A very powerful way of creating some kind of user interface is through a browser, so we will create a webserver on the Pi. We also need some way to transmit and receive data between the two devices, so we need to implement UART on the solderino in c and use some library in python on the Pi.
With this in place it should be possible to create a simple API to control the whole system, which can be extended with a frontend for a better user experience. I will only be covering the solderino/USART part here



UART

Luckily the data sheet has us covered, and all the needed stuff is there as code examples for setting registry, receiving and transmitting data in c.
Though for serial communication to work well, we need to design some kind of data structure or protocol that reflects the values we work with.

To start a game, the raspberry pi will send a character (‘g’) to signal start, and then a number of games to be played (1-9 games). This breaks the loop on solderino waiting for a serial character, then runs the play_games_and_transmit function passing the amount of games as parameter.

Since the score is actually an unsigned long, 32 bit long, but we designed the maximum value to be max 500, we can shift the lower 2 bytes of this into an array of unsigned characters (the USART input params), and send each of the bytes: ‘D’ + scoreH + scoreL

```
USART_Transmit('D');
data[0] = (score[i] >> 8) & 0xFF;
data[1] = (score[i] >> 0) & 0xFF;
USART_Transmit(data[0]);
USART_Transmit(data[1]);
```

Then on the server side we listen for a b’D’, and the next two bytes we decode into an int, add a timestamp for the game and push all the data to the database.

Funky Issues

The Unsigned Long of Despair

After learning to control the servo, I created the map function and tested these by using mock scores and degrees as inputs, ensuring the servo would still move correctly to the expected position.

However,when integrating the different high level functions, I realised when testing the play_game function, the servo would sort of just jitter and move at max ~20 degrees from the baseline. This was very odd because when running the test routine, all the servo movements were done correctly and in order.
It would also happen when using the map functions, even though I knew they had been functional and untouched since then.

After thoroughly checking the map functions, recreating the same logic in python too, I still had no idea why the integration was unsuccessful.
What I didn’t realise was, when calculating the score in play_game, it was done by finding

the difference between two readings of millis().
The issue, however, is that the millis() returns an unsigned long, a behemoth of 32 bits. This variable was being passed through convertScoreToDeg, which at the time accepted only an int. Obviously this was a huge error in the logic and after changing the type everything worked perfectly again.

Full Source Code

https://github.com/KianBay/reaction-tester-mini-project