# Directions for use

Sometimes the motor needs a kick

Motor needs to be spinning to sing nicely

# Algorithms description

## Reading input from user

To read the instructions from the user, the serial output was declared in the code. While the *main*() function runs, it is checking if the input from the serial port is readable. It is in this state that it waits until it an instruction is sent.

Once this happens, the program records each character into a char array. Based on the specifications, the maximum length of an instruction is 49 characters, which is the size of the array. This continues until it detects an ENTER key. The end of the last valid character is recorded for later use.

While there are 4 possible types of instructions possible (R, R & V, V, and T), there are 3 possible characters in the first position (R, V, and T). There needs to be a way to differentiate between R and RV. Thus, the program checks if a V exists other than at the first position.

If the first character is a T, the remaining instruction characters are sent to the *charToNotes* function. Then the singing actions are performed. If it is a V, the remaining characters must be a float, and are sent to the *charsToFloat* function. The return value is assigned to the *desiredSpeedValue*, and execution happens.

If the first character is a R, and no V is detected, the remaining characters are sent to be converted to a float. This is assigned to *desiredRevolutions* and then the instruction is executed. If there are both, characters between R and V are converted into the float for *desiredRevolutions* and the ones between V and the end are converted into *desiredSpeedValue*.

For the case of a *desiredSpeedValue* or a *desiredRevolutions*, the values are then checked for either *counterclockwise* or clockwise operation. This sets the *spinCW* boolean, before all the values are set to their absolute value.

### charToFloat

This function takes in the buffer, and the start and end points for it to calculate the float. The first step is to check for negative values with the first character. If this is the case, *isPositive* is set to -1, and the start index increased, to make processing easier.

The next step is the decimal portion, if it exists. The decimal place in a float is either in the second last position, the third last position, or does not exist. Thus, the program checks for this. In the first two cases, this makes it easy to calculate the decimal part after converting the characters from ASCII by subtracting the value of '0'. Those values are multiplied by .1 and .01 if needed.

The location of the decimal point is recorded to help isolate the whole number portion. In the third case, where there is no decimal point, the decimal is assumed to be after the last character.

To calculate the whole number part, the start character's location is compared with the decimal point location to determine how many digits there are. Depending on the number (1, 2, or 3), they are converted from ASCII before being multiplied by 100 or 10 as needed.

The whole number part and the decimal part are added up, before being multiplied by *isPositive*. *isPositive* is either 1 if there was no negative, or -1 if there was. The entire value is then returned.

### charToNotes

Looking at the regex expression, the notes range from A to G, with the option of flats and sharps for each one. Applying some musical knowledge, an octave goes in the order of C-D-E-F-G-A-B. A flat(^) is half a note lower, and sharp(#) is half a note higher. However, to make things confusing, some notes are separated by half notes, such as E-F and B-C. This also means that a sharp of one note can be the flat of another note. Putting this altogether means that there are in fact only 14 distinct notes playable. They are shown below.

|  |  |
| --- | --- |
| 0 | C^ |
| 1 | C |
| 2 | C# or D^ |
| 3 | D |
| 4 | D# or E^ |
| 5 | E or F^ |
| 6 | F or E# |
| 7 | F# or G^ |
| 8 | G |
| 9 | G# or A^ |
| 10 | A |
| 11 | A# or B^ |
| 12 | B |
| 13 | B# or the next C. |

Each note can either be two characters long (a letter and a number), or three (letter, sharp/flat, and number). Also, there can only be 8 possible notes. To deal with this, a counter first points to the first character. The next character is checked for a # or a ^. If this is not found, the second character must be the time, and the first character the note. The note is stored as a integer, as shown above (1, 3, 5, 6, 8, 10, 12), which selected through a case statement. An array of integers (timeArray) stores the times, and another (noteArray) stores the respective note integer. The counter is incremented by two.

If a # or a ^ is found, the third character must be the time, and the first the note. The entire process of storing the time and the note integer is repeated as before. However, if there is a #, the note integer is increased by one, and if a ^ is found, it is decreased by one. Because of the indexing in the array, this means that it will correspond to the correct sharp/flat. The counter is incremented by 3 this time.

This checking process happens until the end of the buffer is reached.

## Spinning at defined speed (V mode)

At the beginning, several initialisations are performed. The speed value read from user is converted to float variable. Based on a comparison with 0, flag for direction of spinning (clockwise or anticlockwise) is set and with is variable *lead*, polarity of which depends on the desired rotation direction. Next, the speed PID controller is set up with the correct constants for its P, I and D parts. Then, interrupts are enabled, speed timer is initialized and flag denoting mode od operation (speed control) is set. At the end speed control PID is started in a thread.

To spin the motor interrupt function *state\_interrupt* is triggered at each rising and falling edge of the photointerrupters, except the rising edge of the I1 photointerrupter (explained below). In the function, the rotor state is read from the photointerrupters and next state is output into the rotor fields. The state is set in another function, which finds the setting of the field transistors corresponding to the requested state, turns of all transistors (to avoid shoot-trough) and then sets them according to the state setting found before. The transistors are excited with PWM, calculated in the speed controller.

To calculate the PWM duty cycle PID controller from the mbed PID library is used. It is created, set up and started in a thread during initialisation of the program. The constants for the P, I and D parts were tuned during testing. The controller takes the input of the current speed (measurement of this is explained below) and the reference is the value requested by user. The control signal produced is saved into a distinct variable.

To measure the speed, interrupt function *state\_interrupt\_speed* is triggered on each rising edge of I1. It operates with a timer which is read and then restarted. Based on the timer output, current speed of the rotor is calculated and saved into appropriate variable. Afterwards, based on the mode-of-operation flags, appropriate duty cycle (in this case the one from speed controller) is assigned to a global variable. This is used in the function switching rotor field states as described above. At the end, next motor state is set as explained above.

Overall thus, the speed control mode consists of 2 parts. One is the PID controller running in a thread calculating appropriate PWM duty cycle to achieve desired speed. The other part is driven by interrupts from the photointerrupters. It spins the motor field around with application of the PWM pulse, and at each revolution calculates the speed of the rotor and updates the duty cycle of the PWM to be used.

From optimisation point of view, it was considered to set of the PWM duty cycle and calculate the speed in each of the interrupts (6 times a cycle). This way higher accuracy of the control could have potentially been achieved. However, this approach would also potentially pose problems at high speeds, where the new state might not get output to the transistors before the next interrupt is triggered. Moreover, the PWM calculation in the controller would have to be performed more often, thus taking more CPU time.

## Spinning for defined number of revolutions (R mode)

The mode for spinning for defined number of rotations is very similar in its operation to the V mode. It uses the same interrupts triggered by the photointerrupters and a PID controller calculating the required PWM duty cycle. At the beginning, desired number of revolutions from the user is read, and flags and variables for direction of spinning are set. Then interrupts are enabled, timer started, counter for number of revolutions initialised and flag for R mode set to 1. In the end position control PID controller is starter in a thread.

The position control PID is also an object created using the mbed PID library. The setpoint is the user-defined number of revolutions and measured variable number of revolutions already completed. The controller outputs PWM duty cycle, which is applied to the rotor fields in the same way as in V mode. The number of revolutions completed is tracked by a variable, which is incremented in the *state\_interrupt\_speed* function, i.e. once a revolution.

## Spinning for defined number of revolutions at defined speed (RV mode)

## Singing (T mode)

To make the motor sing while spinning, the method used was to apply a square wave to the transistors during their normal operation. To change the frequency of the note played, the frequency of the square wave must be changed. This can be easily achieved by using the PWM output of the pins.

Once the notes have been deciphered, as previously described, the notes are printed out for the user's convenience, before the motor spinning thread runs at a set speed. The PWM needs only to be applied to one set of pins. The L1L, L2L, and L3L pins were chosen for simplicity. All three must be used because this ensures that at least one is operational at any point during the motor spinning.

Then the singing thread starts. This threads checks the first value in *noteArray*. The desired period is stored in another constant float array known as *frequencyPeriodTable*. The value in *noteArray* corresponds to the correct index in *frequencyPeriodTable* to give the correct period in microseconds. This is applied to the PWM pins. Then the thread waits according to the corresponding wait time in *timeArray*. This runs itself until the end of the notes, before the modulo function on the *notePointer* means the thread goes back to the first note. This continues until another command is entered.

However, this alone will not cause the pins to sing. The motor spinning thread assigns either 1 or 0 to the pins. This works normally even with the PWM pins before 1 means a duty cycle of 100%, which is a constantly high pin. A 0 means a 0% duty cycle, so a pin that is off. However, for singing to happen, there must be a square wave. This is why *isSinging* is set to true when T is detected. This conditional means that in assigning values to the motor pins, the L1L, L2L, and L3L pins have their value (usually either 0 or 1) divided by 2. This means that they are either off when there is a 0, or set to 0.5 when there is 1. This 0.5 value is a 50% duty cycle, meaning that they output a square wave of the desired frequency.

The values of *frequencyPeriodTable* were selected because of their frequency range. To ensure that the PWM does not interfere with the motor's operation, the frequency must be higher than the maximum switching frequency. The octave selected was C8 (C4 is ``middle C''). Thus, the notes range from B7 to C9 (3951Hz to 8372Hz)[[1]](#footnote-1).

# High level program flow description

To ensure most efficient use of limited system resources, the program makes use of threads and interrupts in its operation. Generally, the main loop continually polls for another user input, while current task execution depends on functions executed in an active thread or triggered by interrupts.

In speed and position control, the spinning of the motor is driven by interrupts triggered by photointerrupters. This ensures the motor can spin up to its physical limits, since interrupts take priority in the execution. The PWM duty is calculated in a thread with period of 20ms. Since the PWM duty for the transistors is changed once every cycle (in interrupt triggered on rising edge of I1), this theoretically allows for speed control up to 50Hz. If a new duty cycle is not ready every revolution of motor, the old value of duty cycle will be used. At speeds above 50Hz this will not have any detectable effect, since the consequent values of duty cycle will always vary by small values. This was confirmed in testing and can be seen in graph below.

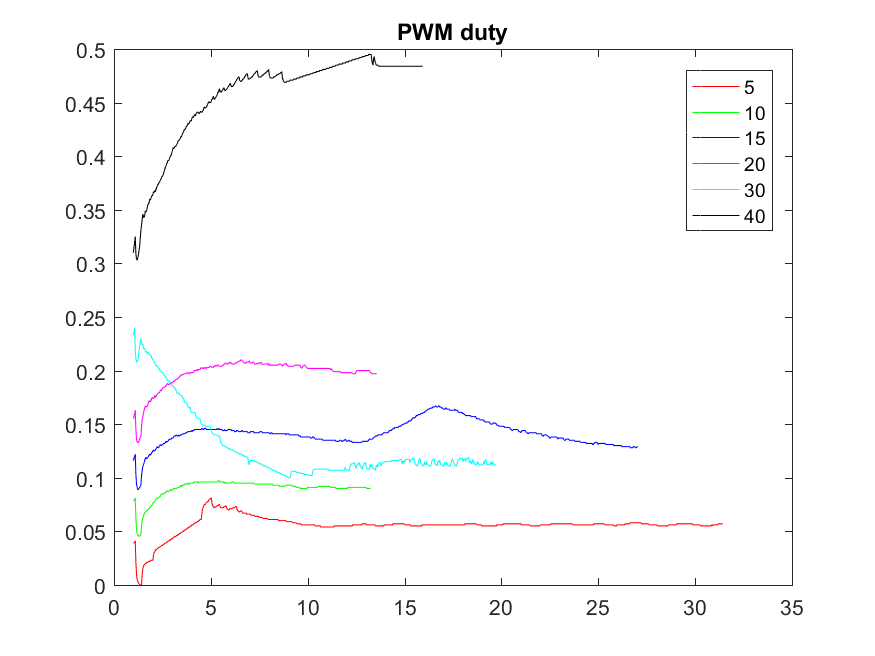


Figure : evolution of PWM duty cycle for speed control at various user-requested speeds

In the singing mode of operation, there are two functions running is separate threads. *FixedSpeedRevolutions* spins the motor, while *playNotes* controls the actual singing.

Threads

Deadlock analysis

CPU free time

# Tasks and threads analysis

## Execution time for threads

Measure latency from photointerrupt to motor

## Deadlines

### Photointerrupter to motor output

### PID output ready

## Thread priorities (scheduling)

1. *source: http://people.virginia.edu/~pdr4h/pitch-freq.html* [↑](#footnote-ref-1)