# REAL-TIME BRUSHLESS DC MOTOR CONTOL

Embedded Systems Coursework II Report

Meng Kiang Seah (mks211)
Martin Opatovsky (mo1013)
David Salmon (ds2713)

# Table of Contents

## Directions for use (Readme)

The motor fields sometimes develop too little torque to start spinning. In such instances, a little push is needed to start the motor.

In the singing mode, optimal performance is achieved when the motor is spinning at the top speed. If the motor cannot start spinning, a little push is required as above.

# 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 an instruction is detected. If an instruction is detected, the program terminates all threads (i.e. stops all executing tasks) and parses the input.

Once this happens, the program records each character into a character 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 index of the last valid character is recorded for later use.

While there are 4 possible types of instructions possible (R, RV, 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 character.

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 counter-clockwise 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 by ignoring the "-".

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. If a note is a flat(^), it is half a note lower, and if is it sharp(#), it 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.

*Table 1: Playable notes correspondence indexing.*

| | | | |
|---|---|---|---|
| 0 | C^ | 7 | F# or G^ |
| 1 | C | 8 | G |
| 2 | C# or D^ | 9 | G# or A^ |
| 3 | D | 10 | A |
| 4 | D# or E^ | 11 | A# or B^ |
| 5 | E or F^ | 12 | B |
| 6 | F or E# | 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 an integer, as shown above (1, 3, 5, 6, 8, 10, 12), which is 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 that is the variable *lead*, the sign 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, the speed timer is initialized, and the flag denoting mode of operation (speed control) is set. At the end, the speed control PID is started in a thread.

To spin the motor, the interrupt function *state_interrupt* is triggered at each rising and falling edge of the photointerrupters, except at the rising edge of the I1 photointerrupter (explained below). In the function, the rotor state is read from the photointerrupters and the next state is outputted 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 off all transistors (to avoid shoot-through) 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, the PID controller from the mbed PID library is used. It is created, set up, and started in a thread during the 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, the current speed of the rotor is calculated and saved into the appropriate variable. Afterwards, based on the mode-of-operation flags, the 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, the next motor state is set as explained above.

Thus, overall, the speed control mode consists of 2 parts. One is the PID controller running in a thread calculating appropriate the PWM duty cycle to achieve desired speed. The other part is driven by

interrupts from the photointerrupters. It spins the motor field around with the 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.

The final performance of the speed control was observed and the results are shown in the figure below. There is some oscillation apparent, which would have been resolved by further PID tuning. This was unfortunately not possible due to the tight deadline at the time of code development, and the need for a manual approach.

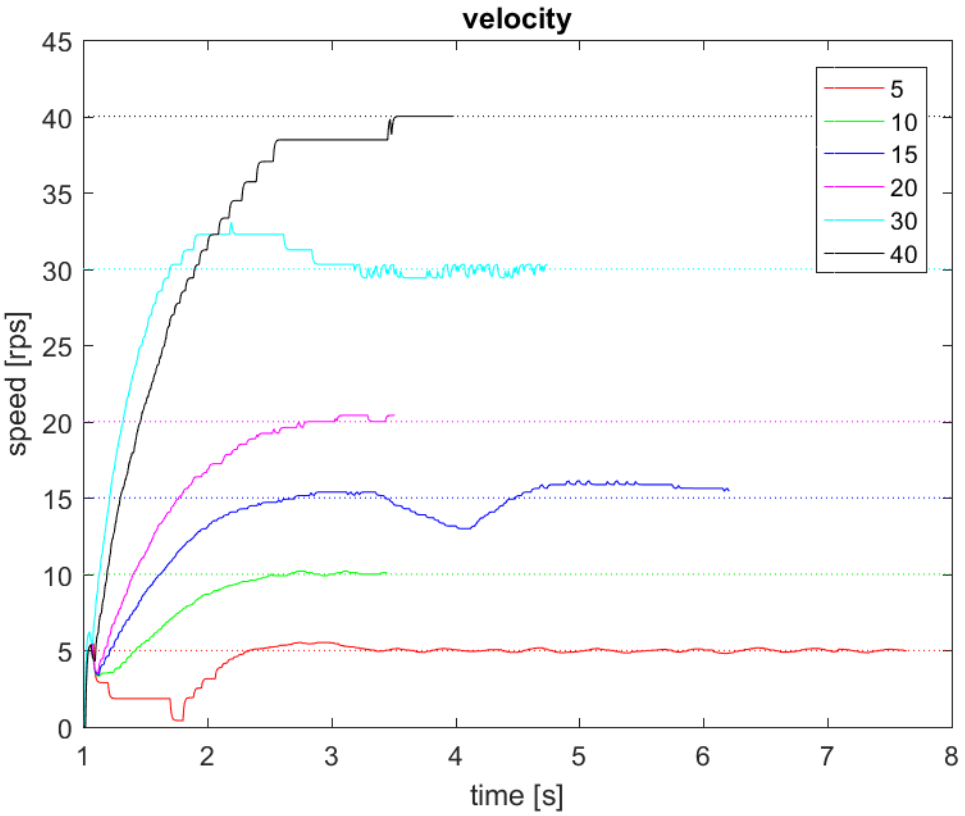


*Figure 1:Vvelocity control performance at different set speeds*

## 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, the interrupts are enabled, the timer started, the counter for number of revolutions initialised, and the flag for R mode set to 1. Finally, the position control PID controller is started 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 the process variable the number of revolutions already completed.

The controller outputs PWM duty cycle, which is applied to the rotor fields in the same way as in the 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)

In RV mode, speed of spinning should be controlled up to a desired number of revolutions. In the code, functionalities from both V and R mode are used. First, user-requested values are extracted and clockwise/anticlockwise flags set. The integral part of speed PID is then suppressed, so that the desired speed is not overshot, since that would cause significant problems to achieve desired number of turns. The rotation of the motor is driven by the interrupts as in previous cases.

In the execution, motor is speed controlled up until certain number of revolutions, when rotation control is switched on. This number is calculated based on the requested speed and revolutions, and number of rotations needed to stop the motor (this was obtained through testing). Based on these the number of revolutions at which the rotation control should be switched on is calculated. If this number is lower than a defined threshold, rotation control is switched on directly. This is due to the fact that there are minimum number of rotations the controller needs to reliably stop the motor.

Once these calculations are performed, speed and position PID controllers are started in separate threads. The motor is spun using the photointerrupter interrupt functions. Once every revolution, the interrupt function calculates current speed and number of revolutions completed, and selects the PWM duty cycle (coming either from speed PID or position PID) to be used. This function checks whether maximum number of revolutions to be completed with speed control has been achieved. If this is true, the PWM to be used comes from position PID, otherwise it is the speed PID.

### 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].

# Real time operation analysis

## Thread description and functional deadlines analysis

To ensure the 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. The deadline for the PWM thread is thus soft.

[1] *source: http://people.virginia.edu/~pdr4h/pitch-freq.html*

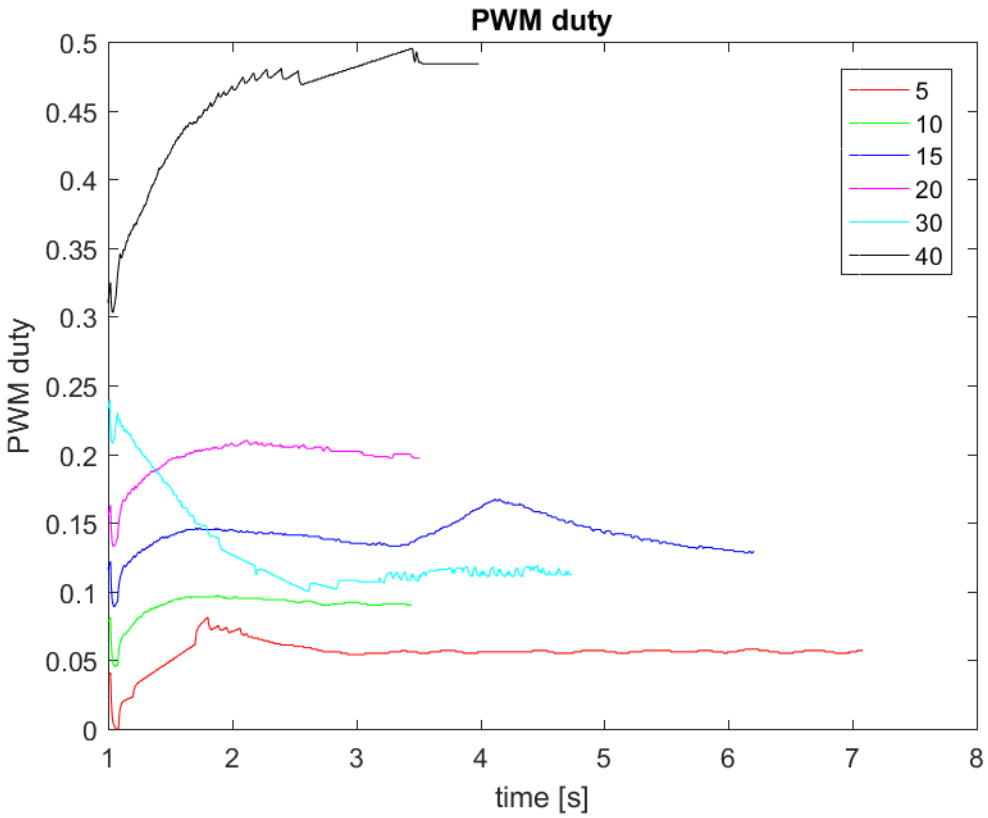


*Figure 2: Evolution of PWM duty cycle for speed control at various user-requested speeds*

In the RV mode, two threads for 2 PID controllers (speed and position) are running concurrently. Their deadlines are soft as explained above and therefore there are no concerns for performance. The advantage of running both is that at the time of switching from velocity to position control, the PWM duty values from position PID are already well initialised and stabilised and hence more effective in control.

In the singing mode of operation, there are two functions running is separate threads with no interrupts. *FixedSpeedRevolutions* spins the motor, while *playNotes* controls the actual singing. The former has a short execution period to spin the motor as fast as it can. The *playNotes* thread executes in a period specified by the user – how long the note should be played. Both timings are soft, since missing a deadline in the first causes slower spinning, which does not affect the functionality. Missing a deadline in *playNotes* will result in slightly longer note period (execution time of *fixedSpeedRevolutions* measured as 74 µs), which the user cannot distinguish and hence there is no loss of functionality.

## Thread deadline details

More general impact of thread and task deadlines on performance of the code was analysed above. In this section, more detailed analysis of thread deadlines is presented. The basic requirement of a thread is that the code in it executes in time shorter that the thread period. Moreover, threads can be overridden by tasks of higher priority, and this should be considered in analysing the deadlines.

The results are summarised in the following table. In the code, each thread has only one function assigned. In R and V mode, there is only one thread with the respective PID controller. In RV mode, there are still these two threads, running concurrently. In T mode, the two constituent functions also run in separate threads. In R, V and RV mode, the threads can be theoretically overridden by the interrupt function (even though this was not observed in the tests), which will prolong their execution. This is accounted for in third column.

To measure the task execution time, a digital output in was toggled at the start and end of the function. The trace was then observed with the oscilloscope, the longest pulse was found, and its width measured. Note that toggling of the output pin would have added to the execution time of the function and thus all data are a worst case estimations and should not take longer in any circumstances.

*Table 2: thread execution times and their deadlines*

| Thread function name | Thread task execution time [µs] | Thread execution time with interrupt [µs] | Thread deadline [µs] |
|---|---|---|---|
| **PPID (position control)** | 4.5 | 63.5 | 20,000 |
| **VPID (speed control)** | 10.4 | 69.4 | 20,000 |
| **playNotes (singing)** | 38.8 | - | 1,000,000* |
| **fixedSpeedRevolutions (singing)** | 74 | - | 1000 |

**The period of playNotes thread depends on user input i.e. how long should the note sound for. The deadline in table assumes 1s long note, which is a reasonable estimate of shortest user-requested time.*

As can be seen in the table, all threads execute in much shorter time than their deadlines. There is therefore no concern of a missed thread deadline.

## Other significant system latencies

The most significant latency in the system is the propagation of the control signal to the motor field transistors. This has to be as small as possible to ensure effective motor control. Measurements were taken to investigate this latency. First, a pin was toggled at the start of the interrupt function and at

the end of it. This yielded a 45µs execution time for an ordinary interrupt to switch motor state and 59µs for an I1 interrupt, which in addition to switching states calculates current speed and revolutions completed, and assigns the appropriate duty cycle into a variable used by the state switching function. These execution times are small enough to support quick and effective motor control and do not interfere with any of the system deadlines (i.e. if the interrupt function interrupts a thread function, the latter will still complete in time – as shown above).

Secondly, the latency from photointerrupter pulse edge (rising or falling) to motor transistor switching was measured. This is closely related with the above function latency; however, it also reflects system signal propagation times and transistor switching times. The longest latency was found to be 151µs from the I2 rising edge to switching of the L3L transistor. Again, this time is small enough so that it does not affect any system functionality.

## Thread dependencies analysis

The threads were designed so that they share as few resources and variables as possible to prevent any possibility of deadlock. Therefore, no resource locking methods (e.g. *mutex*) were used. The two PID controllers are completely independent and use separate sets of variables. They depend on measurements of speed and revolutions from I1 interrupt, however, they will not wait for a new value, but use the value available. This value is possibly old, but as explained above this does not affect performance. Similarly, the I1 interrupt uses the values of PWM duty cycle produced by the PID controllers and will possibly reuse old values with no negative consequences. In the singing mode, the two functions are completely independent of each other.

## Thread priorities analysis

The thread priorities affect execution of the T and RV modes, when two threads are running at the same time. All threads in the program were given equal priority (*osPriorityNormal*) for theoretical reasons outlined below. Additionally, in testing it was found that setting threads to higher priority levels made the program more susceptible to unexplained errors and malfunctions.

In T mode, the spinning function (*fixedSpeedRevolutions*) is functionally less important than setting PWM to change notes (*playNotes*). However, the spinning function has a shorter deadline and thus overriding it with the singing function could potentially affect its performance. Equally, the singing function has much longer deadline than it might possibly need. Equal priority thus works well in this situation.

In RV mode, a case could be made for prioritising speed PID during speed control and position PID during position control. However, this would cause overhead when reassigning the priorities at the

point of switching control, which could potentially affect performance. Moreover, as shown in the Thread deadline details section, both threads execute extremely fast with respect to their deadlines and hence no priority control is necessary.

## CPU free time

The quality of the real-time program can be quantified by how much CPU time is available after all tasks were performed. This was measured in hardware using the following method. The *main()* loop (background task waiting for user input) was configured to set a digital output pin high on each loop of its execution. Each task running in a thread or interrupt performing operations connected to any of the modes of operation was setting the same pin low. The output trace of the pin was observed with digital oscilloscope, which calculated the duty cycle of the pin – this is shown in figure below. The duty cycle is the time the CPU spent in the main loop, idly waiting for user input and hence is the CPU idle time, potentially available for other functions.

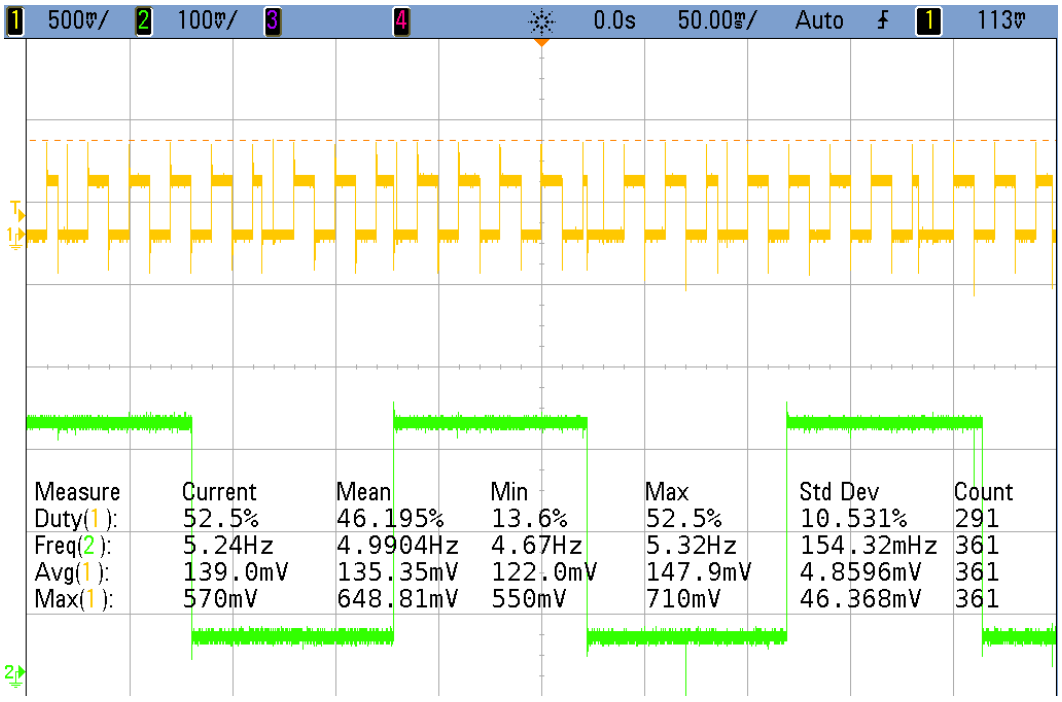


*Figure 3: Measurement of CPU idle time with user command "V-5". Measurement pin in yellow, I1 output for speed checking in green. CPU idle time given by Duty(1) mean value.*

The idle time obviously depends on the task being executed, since some require more frequent operation than others. For instance, when the motor is spinning faster, there will be more time spent in interrupts. Similarly, in an RV mode, we would expect more calculation time spent in PID threads. The results of the measurements are shown in the table below.

*Table 3: CPU idle time measurements for various operating conditions*

| mode | argument | duty (CPU idle) |
|---|---|---|
| **V** | 20 | 29.5 |
| **V** | 50 | 21 |
| **V** | -10 | 40 |
| **V** | -5 | 46 |
| **R** | 40 | 40 |
| **R** | 100 | 40 |
| **R** | 999 | 29 |
| **RV** | R50V10 | 35 |
| **RV** | R20V10 | 50 |
| **RV** | R500V40 | 37 |
| **T** | A1D1E2 | 15 |
| **T** | C1D1E1F1G1A1B1 | 15 |

Several observations can be made on this data. First, the singing mode takes the largest amount of system time. This is caused by the way the motor is spun, which is not driven by interrupts as in the other modes, but rather executed in a thread with short period. This thread takes more resources than an interrupt driven spinning and could potentially be changed (longer period or swapped for interrupts) if more resources are needed.

The RV mode maybe surprisingly takes less time than R or V mode. This can be explained by the fact that is it the interrupts spinning the motor field which take the most of the CPU time. The calculation of the PID controllers is a quick task. This is supported by the fact that R mode takes less time than V mode, since the motor is usually spun much slower in R mode. Later, this hypothesis was confirmed by measurement of the function execution time, where it was found that the interrupts with the motor state switching function take 45µs (ordinary interrupt) and 59µs (I1 interrupt measuring speed, rotations, and setting PWM duty), while PID functions only took 10µs (VPID) and 4.5µs (PPID).