Interrupt driven Arduino Stepper Motor Driver.

Note: this document uses a common notation to have a leading '/' to indicate the signal is inverted.

Preface

There are several Arduino bipolar stepper motor drivers , but I couldn't find any that was non-blocking. When one uses a stepper motor, it's - at least to my consideration - the minimal requirement the motor steps are applied in the background. Reason is that this preserves the non-used core cycles to perform other tasks. Since the AVR is a single core processor, a blocking driver will only be able to provide motor steps from trajectory A to B, but while that trajectory is controlled, the core cannot do anything else.

Pro's
- This motor driver is non-blocking. It interrupts other tasks running on the Arduino to generate the required motor steps.
- A virtual 32bit timer is used to control the motor step frequency. This allows a huge range of step intervals, going from 1 step every 4 seconds to approx. 25000 steps/second.
- In order to support automatic speed increase / decrease, some additional parameters are foreseen for this kind of automation. If unused, fixed motor speeds are obtained.
- The motor end position can be defined as "none", "xxxx steps" or "xxxx milliseconds".
- On Arduino Mega, any 16bit timer may be selected: (TIMER1/TIMER3/TIMER4/TIMER5, hereafter referred to as TIMER1/3/4/5.

Con's
- TIMER1 is the only 16bit timer on an Arduino Uno and is used for the stepper pulse interval. One may consider to rewrite to Timer 2, but loss of timer bits will affect RPM accuracy.
- Since AVR has limited computing power, units are steps/s (not RPM). However, there are some provisions to allow for linear speed increase / decrease.
- Although different timers can be assigned to the motor shield on Arduino Mega, it's at present not possible to modify a Motor Shield and drive 2 shields with different timers.
  (however, this is a relative minor change one can implement him/herself).

So it's obvious, no other library should use the active timer in combination with this driver, unless modifications are implemented accordingly.
Taken into account TIMER1 is also used by the Servo driver, it's expected TIMER1 use is not problematic for stepper motor control.
Only for the MEGA, there is option to #define MOTOR_TIMER3, MOTOR_TIMER4 or MOTOR_TIMER5 to select another hardware timer.

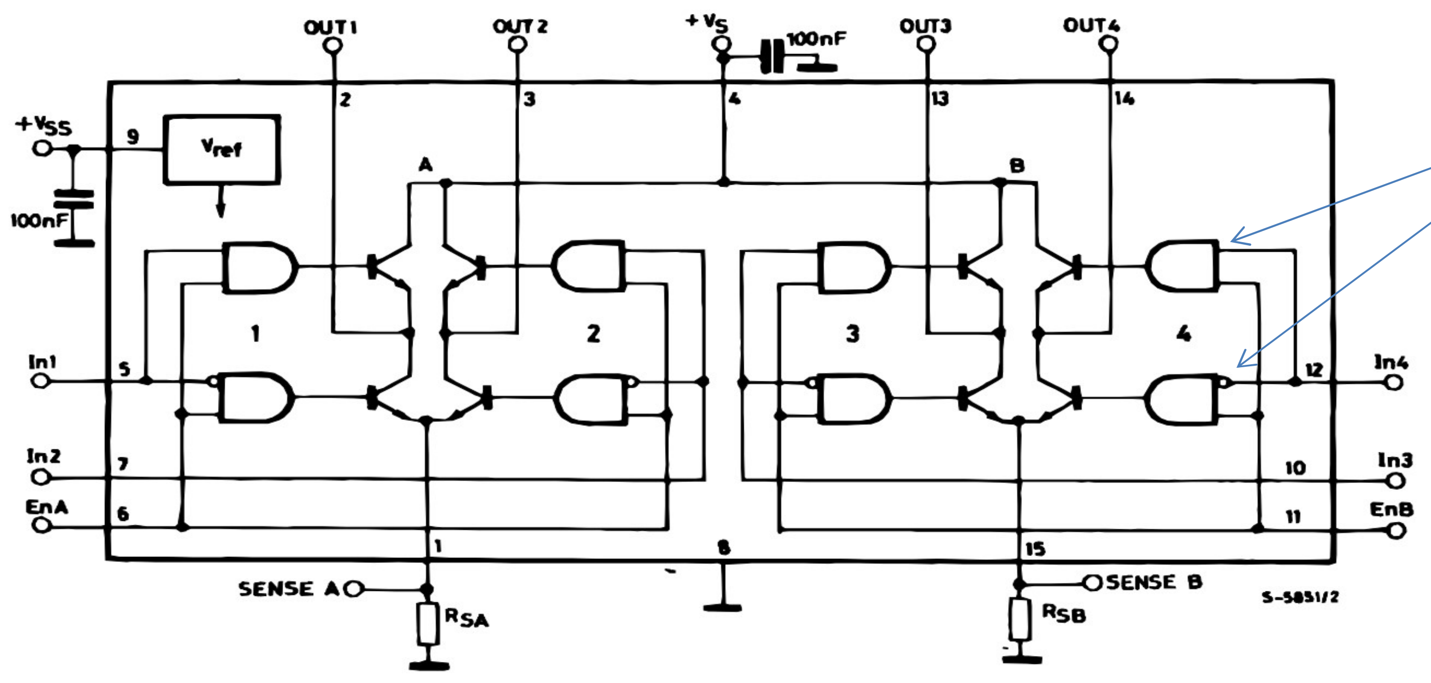
1. The bipolar stepper motor and a H-Bridge (ex. the L298 Driver).

The bipolar stepper motor has 2 coils. How it works is explained on Wikipedia: https://en.wikipedia.org/wiki/Stepper_motor
To drive it by a micro controller, one needs to apply a current trough each coil at any time. The current direction for each coil is:

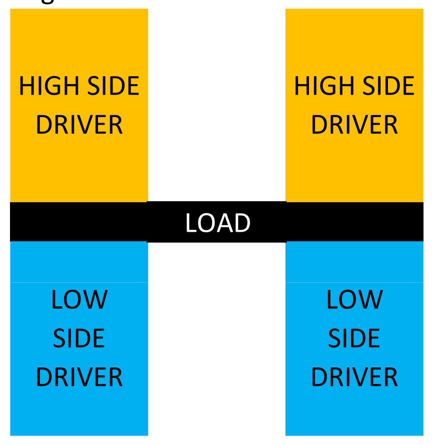
Step	Coil A	Coil B
0	-	-
1	+	-
2	+	+
3	-	+

The above is repeated in a loop (from step 3 back to step 0) till the desired position is reached.
To revert the motor direction, revert the step sequence from bottom to top.
The rotation speed depends on the amount of steps provided every second. A stepper motors' RPM is typically much lower vs. linear motors, but are better in terms of open-loop positioning.
Common used stepper motor have 1.8 degree rotation per step. That means 1 revolution is made by applying 200 steps.

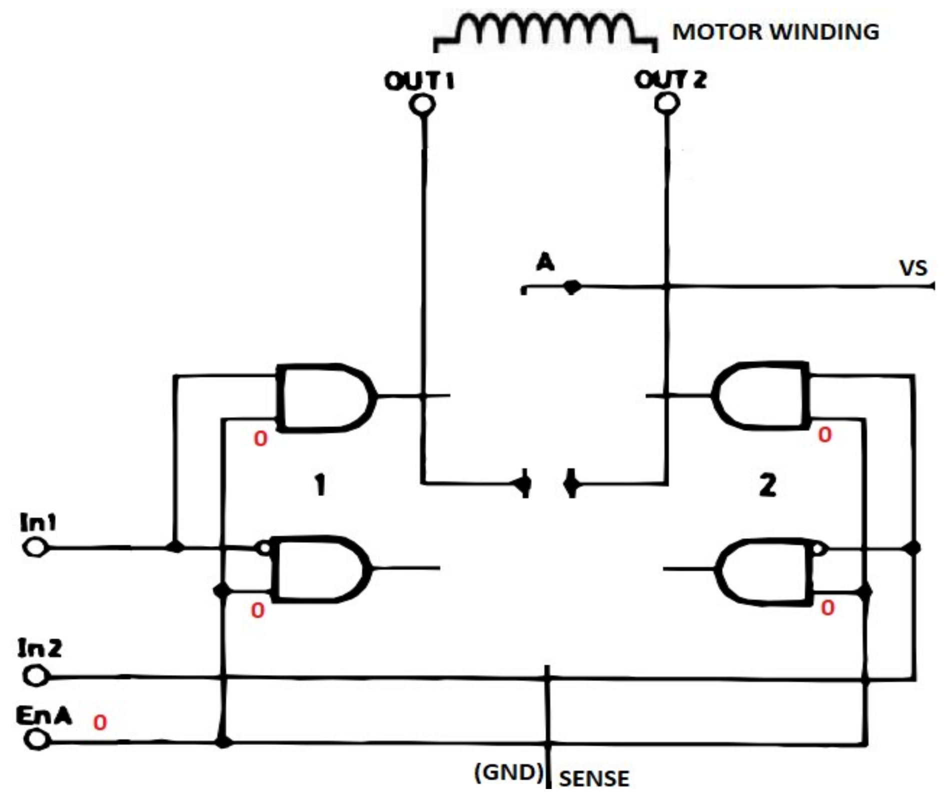
The L298 looks like this. Clearly visible is there are 2 identical blocks. One around In1, In2, EnA, Out1, Out2; the 2nd around In3, In4 and EnB, Out3, Out4



Called a H-Bridge:
Such bridge can invert the voltage polarity on the load:
To avoid shorts, High side and Low side driver of the same side are never active together.



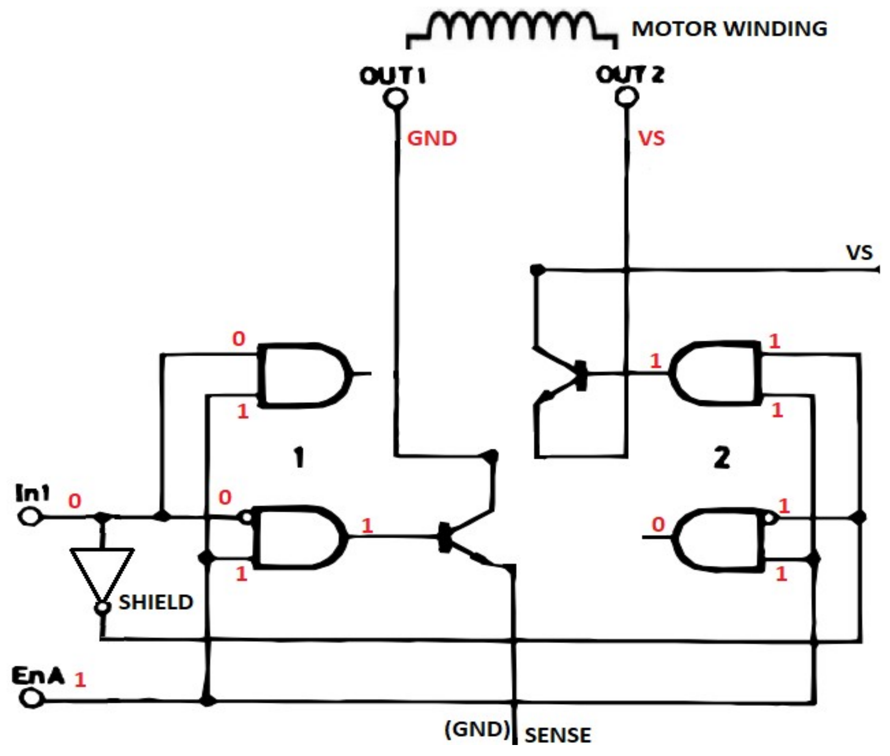
For bipolar stepper motor applications, the Arduino motor shield (see further) allows us to use the L298 driver in 3 different states, depending on the Motor Shield inputs.
Option #1: the Ena Pin is Low: the driver is disabled, the coil acts as if it's not connected to anything (inactive driver outputs are removed from the drawing):



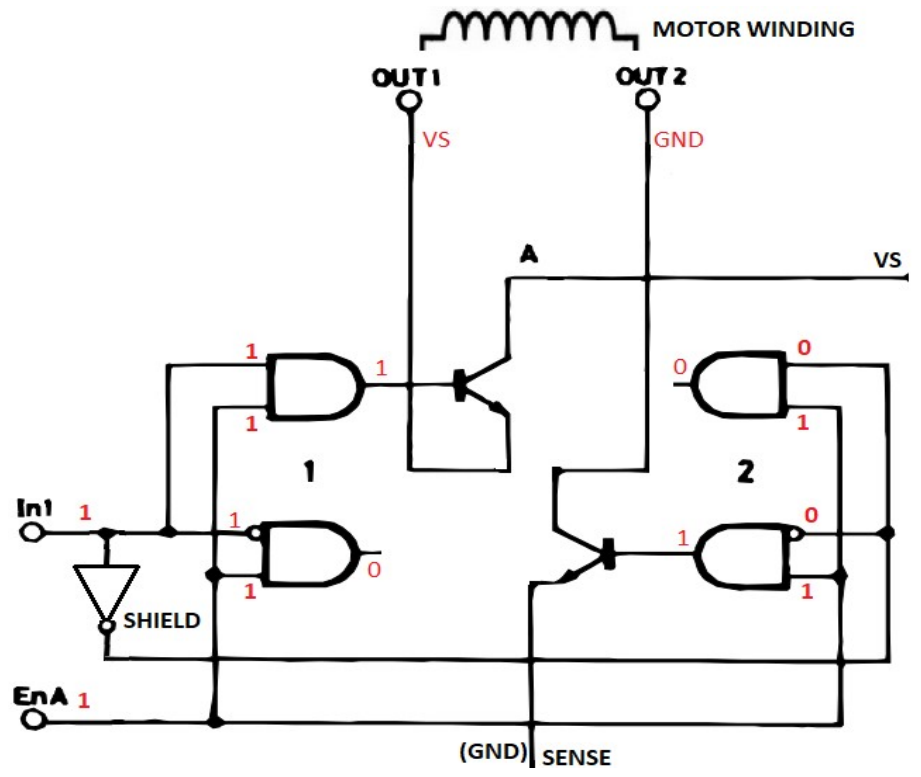
Note:
The sense pin is connected by a small resistor to Ground.
To simplify things explained as GND (in reality there is a small voltage vs. the series resistor & the load current)

Since the coil is not powered at all, no current flows through the windings.

Since for Bipolar Stepper Motor application, the Arduino Motor Shield always has the even input inverted vs. the odd numbered input, the interter is now place in the schematic:
Option #2. In1 is low.
Option #3. In1 is high



In conventional current flow, the coil has now a current from right to left.

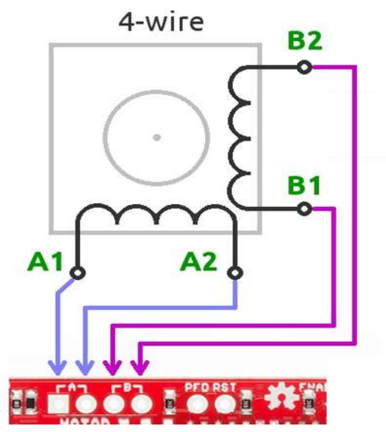


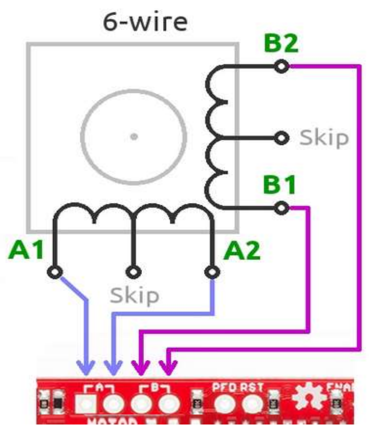
In conventional current flow, the coil has now a current from left to right.

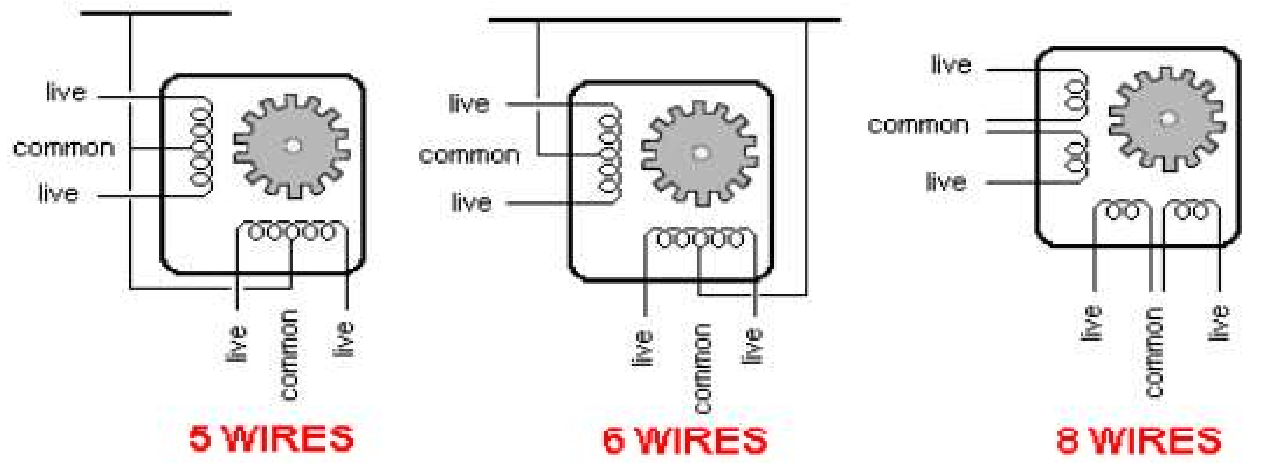
Since a bipolair stepper motor has 2 coils, we need 2 such drivers as shown above. As a result, the L298 can drive only a single, bipolar stepper motor.

2. Unipolar Stepper Motor and Darlington Driver.

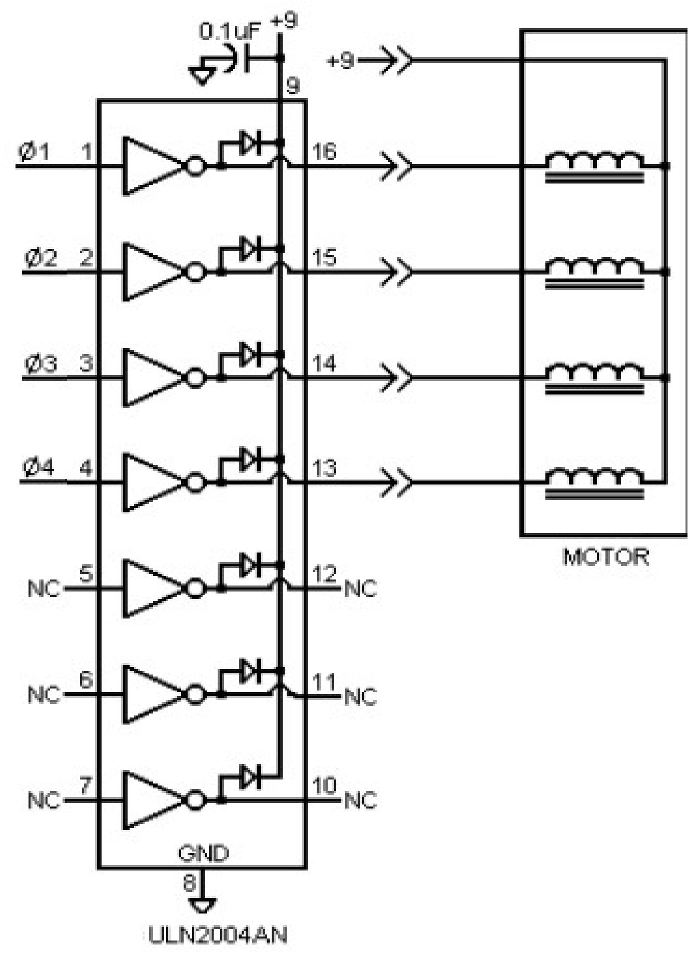
Some stepper motors have 6 wire connection. In such case, the motor can still be considered a bipolar motor with an unused center connection.







Some stepper motors have 8 wires: that's similar to 6-wire, but all coils have individual connections. Those motors are the most flexible to use, but more complex to determine the coil sequence.
Other stepper motors may only have a 5 wire connection. In such case, the center taps of both coils are connected together. These motors cannot be driven by the present software for the Arduino motor shield. However, unipolar stepper motors can be driven by a cheaper darlington or similar driver, like the ULN2003.
The major advantage of this kind of motor is that a ULN2003 or similar driver is cheaper than L298. The major disadvantage is these motors are heavier vs. the provided torque, since only half a coil is active per step. This is how a 5-wire motor can be connected to an ULN2003 / ULN2004 / ULN2803 etc (the pinout of the driver changes vs. the type, ULN2004 is shown):



Φ1-Φ4 are sometimes also referred to as Phase A-D.

PhaseA	PhaseB	PhaseC	PhaseD
1	0	0	1
1	1	0	0
0	1	1	0
0	0	1	1

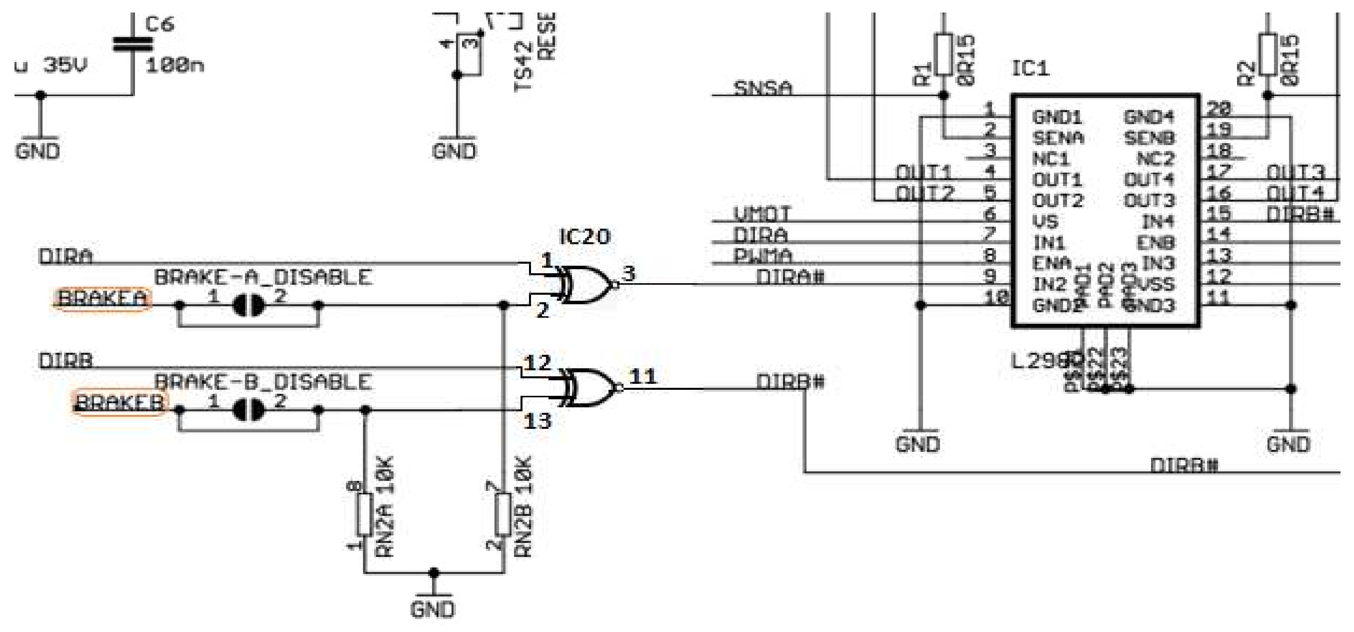
('1' means the driver is in non-conducting state).

3. The Arduino Motor shield with H-Driver bridge (L298)

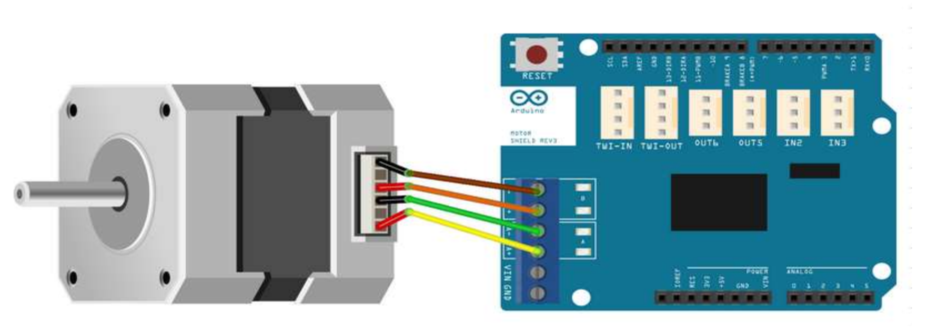
As mentioned above, In2 is always the inverted state of In1 (when the motor shield BREAKA is low), and In4 is always the inverted state of In3 (when BREAKB is low)
If either BREAK signal is kept high, the resulting input is always low (since this is a NOR gate).
Connectivity:

Motor Shield	L298	Uno	Mega	Arduino
DIRA	In1	PB4	PB6	D12
/DIRA	In2	/PB4	/PB6	/D12
DIRB	In3	PB5	PB7	D13
/DIRB	In4	/PB5	/PB7	/D13
PWMA	En1-2	PD3	PE5	D3
PWMB	En3-4	PB3	PB5	D11
BREAKA				D9
BREAKB				D8

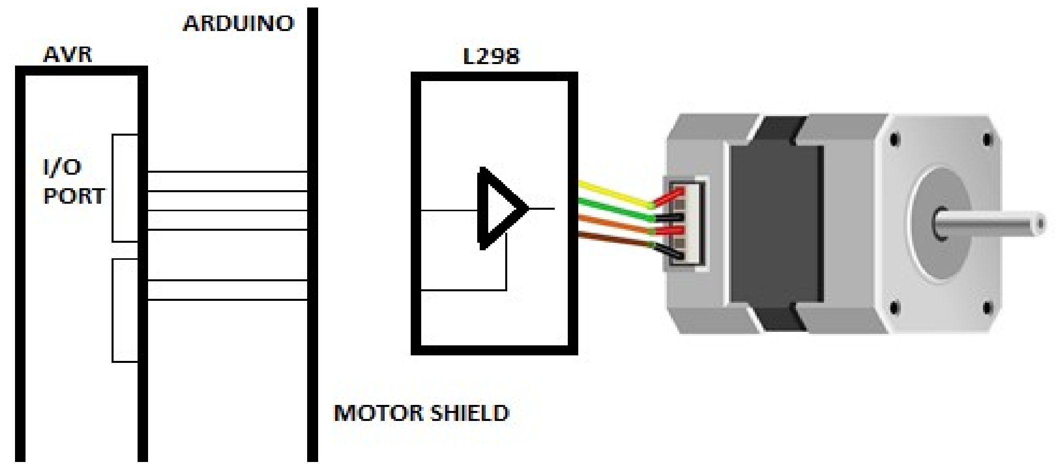
Step	D12	D13
0	0	0
1	1	0
2	1	1
3	0	1



The above information already made it clear the L298 can only drive a single Bipolar Stepper Motor.
The connectiviy is as shown below (motor winding colors may be different):



Looking it from a "Low Level" perspective, we have this:



- The Arduino is built around an AVR core (the Uno uses ATMega328P; the Mega uses ATMega2560).
- The AVR core has I/O ports. The Uno and Mega may use different I/O ports for the same digital I/O pin.
  If one sticks to Arduino digital numbering, the Arduino software takes care of those different I/Os.
  When using the ports directly, the user has to take care of the different port allocation.
- The Arduino motor shield connects to a number of I/O pins and contains the L298 Driver.
  The L298 has 4 digital output stages. Each stage can drive to either supply or ground level.
  Two stages share one enable pin. If that pin is driven low, the two drivers are disabled.
  An amplifier is typically drawn as a triangle; the enable input is in this case at the bottom.
  Only one amplifier is drawn for simplicity, in reality there are 4, each pair charing 1 enable input.

Arduino did not provide an option to drive the stepper motor in half step mode. The BREAK signals are not used by this driver (according my interpretation there is error in the schematic and the labels should be moved).
As a result, BREAK pins must remain LOW. If BREAK is high, IC20 no longer functions as an inverter required for stepping motor (I didn't check if playing with BREAK might still enable half-step motor steps).
The PWM named pins (D3 and D11) are in fact the L298 driver enable pins. These must remain HIGH to enable the driver for bipolar stepper motor. When LOW, the motor is not driven at all.
Binary pins D12 and D13 must be driven by the above shown pattern to rotate the motor. If one looks to the Arduino Uno & Mega schematics, different AVR I/O pins are used for the Uno & the Mega.
Considerations:

1 Since the BREAK lines are not used with bipolar stepper motor, consider to cut BRAKE-A_DISABLE and BRAKE-B_DISABLE. This frees up D9 and D8.
This change would allow to drive a 2nd motor shield where pins D12 and D13 are cut and connected to - for instance D9 and D8 (on both PCB the BRAKE-x_DISABLE wires must be cut to disconnect D9 and D8 from BRAKE-x).

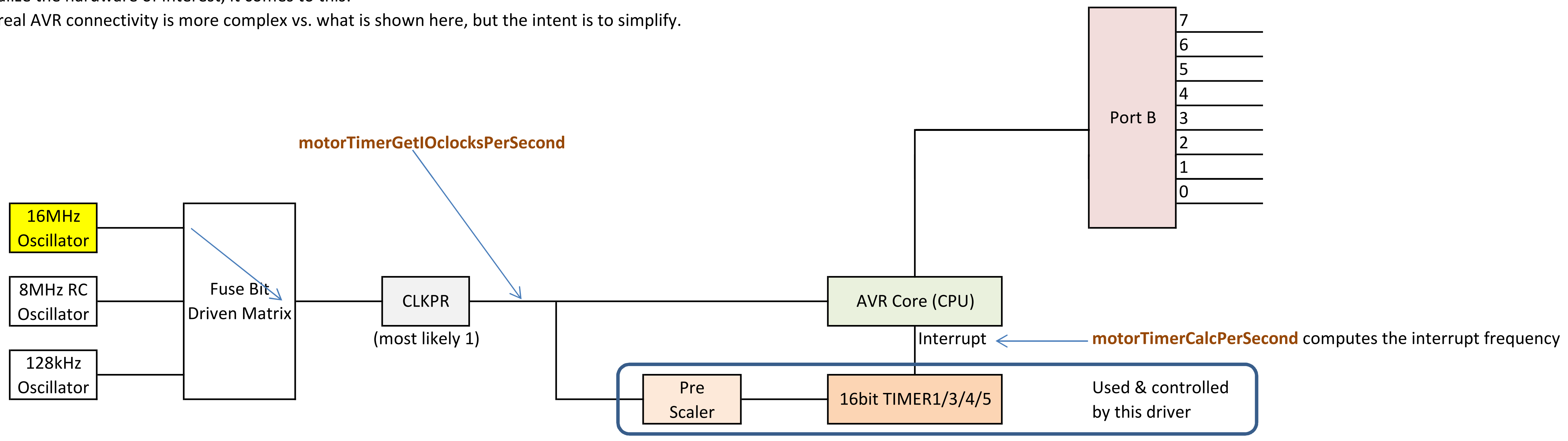
2 To allow half step (requires software modification), consider to remove IC20 and bring pins 3 and 11 (In2 and In4) to other Arduino pins (preferably port B to keep software changes simple.

4. The AVR core, used in the Arduino Uno & Mega.

This documentation only provides the minimal AVR core knowledge to understand the driver software.
For additional AVR core details, consult the AVR datasheet on Microchip website (ATMega 328P for Uno, ATMega2560 for Mega) and the respective Arduino schematics.
All Arduino I/O pins are pins made available by the AVR designers in so called Ports. Since AVR is an 8bit chip, these ports are also 8bit in size. Ports are named A, B, C and so on. The Mega has more ports vs. the Uno.
A digital I/O pin is basically a set of RAM cells where the state (0 or 1) of one RAM cell chooses between input or output mode. In case of output mode, the 2nd RAM cell drives the output level high or low.
In order to operate, the processor has a clock. The clock is typically generated from an oscillator. Transitions of the clock signal are used to slice time into individual, sequential operations.
Those operations may be instruction execution, counting clocks (to increase the time span), performing serial communications etc. So there's a core clock, I/O clock etc.
So the I/O clock is generated from the oscillator output. On Arduino that oscillator is 16MHz (crystal) although the AVR allows to select also an internal 8MHz or 128kHz oscillator.
The function **motorTimerGetIOclocksPerSecond** returns the amount of expected IO clocks per second. Most users will not use the CLKPR option nor AVR fuses, so this is expected to be 16000000.
Since the AVR core is also driven by the oscillator and most instructions require a single clock transition to execute, the core is nearly capable of 16MIPs (Million Instructions Per Second), where each instruction is 8bit. The AVR is an integer processor. These processors are much slower using real numbers vs. integers. For this reason, the motor driver does not use RPM, but pulses per second.
For example, if one uses a motor that requires 200 pulses per rotation, 100 pulses per second equals half a rotation per second, or 30RPM.

The first connection between the AVR and the motor shield is via the digital I/Os. The motor shield is just an amplifier, because the digital I/O is unable to drive the bipolar stepper motor directly.
Accessing AVR ports via Arduino commands is much slower vs. driving the ports directly. A direct drive has the disadvantage the code is no longer universal / portable. So we make a tradeoff:
Where possible, Arduino I/O is used, for motor steps, the ports are directly addressed.
As a result: if another Arduino board is targetted, one has to check if the I/O ports are covered by existing code or modify it accordingly.

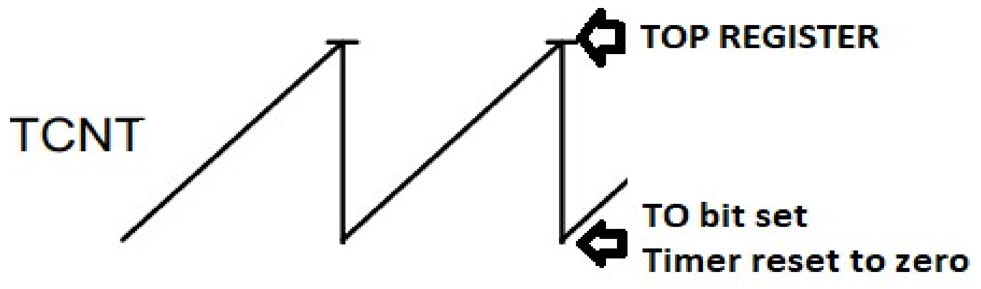
If we visualize the hardware of interest, it comes to this:
Note the real AVR connectivity is more complex vs. what is shown here, but the intent is to simplify.



The 3 possible oscillators are selected by AVR fuse bits. On Arduino, it's expected the 16MHz is always selected, but the existing code checks the appropriate fuses to allow any of the 3 options.
The CLKPR register allows to divide the oscillator by 1, 2, 4, 8, 16, 32, 64 or 256. On Arduino, this divider is expected to be 1 at any time. Any value other than 1 makes the AVR slower, but it also reduces power consumption (that's the only reason why one may select a divider other then 1).
The **motorTimerGetIOclocsPerSecond** function typically will return 16 million since most users are not expected to use the CLKPR feature.
Suppose we want to rotate the motor by 2 revolutions / s and it requires 200 steps per revolution. In that case, the motor stepping pattern (chapter 1) must be changed every
0.0025 seconds or every 2.5 ms
What we don't want is that the Arduino is "delay"-ing 2.5ms and applies another step. The driver aims to interrupt the AVR core every step, let it change the I/O pins (spin the motor 1 step) and return to what it was doing before. In other words: the driver should be non-blocking. Remind the AVR core can only perform 1 task at any time.
The solution: interrupt. Just like a phone call interrupts a person from the work one was doing, interrupts stop the AVR core from what it was doing, store it's operating state, perform the interrupt (like human example: the phone call) and when ready, return to the work one was doing.
The AVR has timers on board to achieve this. The ATMega328 is a scaled down ATMega2560, so common available hardware is used. TIMER1/3/4/5 is a 16bit timer that has a pre-scaler.
The timer has 15 possible configurations, but we use only one: the FAST PWM mode. What happens is:

1. The IO clock is divided by TIMER1/3/4/5 pre-scaler. This divider can be 1, 8, 64, 256 or 1024. So the timer input is the CLKPR output additionally divided by the timer prescaler.
2. The counter counts up at the rate of the prescaler output. When it reaches the "TOP" register, the following events are done by AVR hardware:
   - The timer overflow interrupt (TOIF) flag bit is set in a register called TIFR1 (TO = Timer Overflow, IF = Interrup Flag).
   - The timer is cleared but counting continues by AVR hardware. So no matter how long the core needs to react, the next motor step occurs at exact the same interval.
   - If TOIE was set (TOIE: IE = Interrupt Enable) in the TIMR1 (MR = Mask) register and the AVR core allows interrupts, the function performed by the AVR is interrupted just as a human phone call. The core stores it's present operational data to be able to continue that work later on and jumps to a function called ISR (TIMER1/3/4/5_OVF_vect). This driver uses that function to advance the motor stepping pattern.
     If TOIE was not set, or TOIF is not set because the counter did not time out yet, the AVR core performs the usual code (the one in your Arduino loop function).

In a graphic representation, the timer counts as shown here:

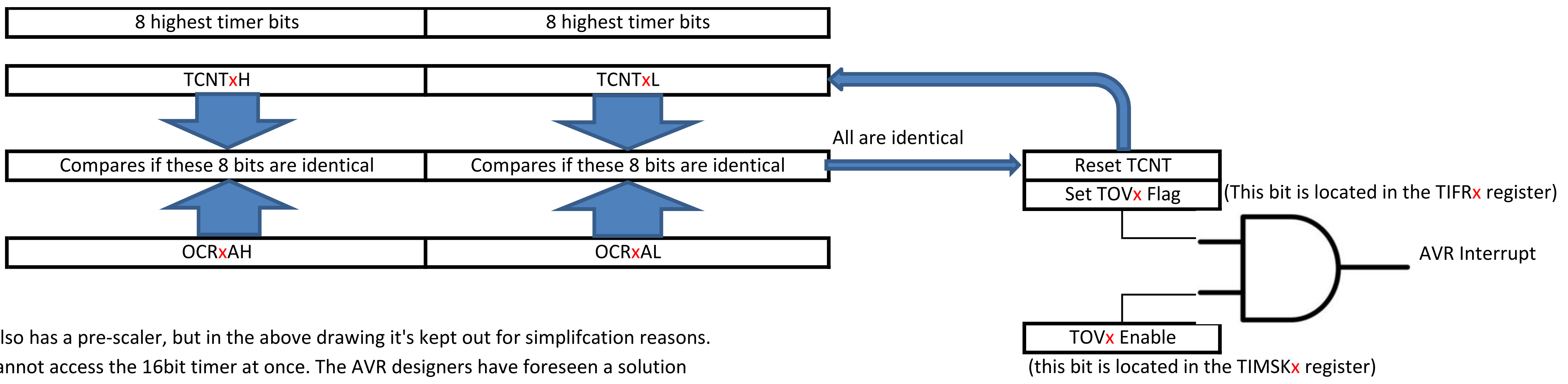


Notes
- Since the TIMER1/3/4/5_OVF_vect takes some time to execute, the minimal timer top value is defined as 640 (this allows up to 25000 steps/s at 16MHz, most likely out of reach for most stepper motors.
- This driver uses often uint8_t, uint16_t and uint32_t. This stands for "unsigned integer", the 8, 16 or 32 is the amount of bits. Since unsigned, it means the value can be 0 up to (2^bitsize) - 1.
  For example, uint16_t has range 0 … 65535.
- In a few cases, the leading 'u' is missing. These are signed values. Since the highest bit is reserved for the sign, the range is from negative (-2^(bitsize - 1) up to positive ((2^(bitsize - 1) - 1).
- A few additions where made to the software: the motor driver can pay attention to so called "update events". This is either an elapsed time (in ms) or amount of moving steps sent to the motor.
  When an update event occurs, the present virtual 32bit TOP value is increased or decreased by another value. An increase makes the time between steps larger, leading to slower rotation. A decrease speeds up the motor rotations.
  When a time-out event occurs, the motor is stopped. This can be used when for instance the speed was gradually decreased.

Specific details about how the software uses the 16bit timer.

Note: as mentioned before, the UNO only one 16bit timer (1). The Mega has four such timers: 1, 3, 4 and 5
(x = 1, 3, 4 or 5 - depending on what's available)
The timer is used in Fast PWM mode. That means the timer counter (TNCT, split in a high and low byte) just counts at a pace controlled by the timer pre-scaler.
The 16bit OCR register (output compare) defines the TOP value at wich 2 operations are performed automatically by the AVR hardware:

1 TCNT (the whole 16bit timer) is reset. So the timer re-starts counting from zero. That ensures the timer interval will re-occur at the same rate (unless a speed change has changed OCRx)
2 The TOVx flag (this is just a 1bit memory cell) is set in a register called TIFR (Timer Interrupt Flag Register). This flags the system the timer had reached the end point.
3 Since we use interrupts, the corresponding TIMR bit is also set once the motor stepping is initialized. The TIMR is the Timer Interrupt Mask register: it enables or prevents that the flag generates a real interrupt to the AVR.
4 The AVR hardware performs a logic AND function between TOVx (in the TIFR register) and TOVx (in the TIMSK register). Since both bits are set at the same time, the logic AND output is high.
5 If the AVR core has interrupts enabled (the default case on Arduino, since it uses interrupts already for USB transmission and it's own millis timer), the **TIMERx_OVF_vect** function gets called in the software.


8 highest timer bits
8 highest timer bits
TCNTxH
TCNTxL
Compares if these 8 bits are identical
Compares if these 8 bits are identical
OCRxAH
OCRxAL
All are identical
Reset TCNT
Set TOVx Flag
(This bit is located in the TIFRx register)
AVR Interrupt
TOVx Enable
(this bit is located in the TIMSKx register)

As mentioned above, the Timer also has a pre-scaler, but in the above drawing it's kept out for simplifcation reasons.
Since the AVR is an 8bit core, it cannot access the 16bit timer at once. The AVR designers have foreseen a solution for this: in reality, the highest 8-bit must be accessed first when written and last when read. Why:

- Ones the software writes to the high byte, it does not write to that part of the timer. These 8bits are stored in a latch. This is just an 8bit memory that preserves the information for now.
- When the software writes the low byte, the AVR hardware writes the 8bit latch and the lower 8bit in one cycle. So the whole 16bit are written in one operation.
- For read operations, a similar operation occurs: if the 8 lowest bits are read, at the same time the 8 upper bits are copied to the latch. This ensures the read operation has the entire 16bit at once. After that, the high byte can be read. It will not return the real upper 8bit, but the latch instead.

This has a reason. Suppose the timer is counting up and the prescaler is set to 1 (it counts every clock cycle of the 16MHz oscillator). The present timer value is 0x12FE. Without the latch, the following can happen:

1 - The software reads the lower byte. This is 0xFE.
2 - Suppose the software writes this result to RAM. That may require 2 clock cycles. So when it has finished the RAM write, the timer has also continued to count up. At this time, timer is 0x1301.
3 - The software reads the upper byte. This is now 0x13 and writes to RAM.

Conclusion: the software would incorrectly assume the 16bit value from the timer is 0x13FE. Now we repeat these steps with the latch:

1 - The software reads the lower byte. This is 0xFE. At the same time, the AVR hardware design copies the timer upper byte to the latch. So the latch contains 0x12.
2 - The software stores again the result in RAM. After it has finished, the real Timer TCNT value is still 0x1301.
3 - The software reads the upper byte. But since it reads the latch, it reads 0x12, not the real value of the timer upper byte (that is already 0x13).

This way, the AVR design prevents software from making the wrong assumption the timer was 0x13FE. Thanks to the upper timer latch, the correct value is read by software.

If the AVR interrupt becomes active, basically the following actions happen:

1 - The processor finishes the instruction it was doing.
2 - The processor status is saved on a special memory, called the stack. This allows the processor to continue the next instruction after the interrupt is processed.
3 - The processor jumps to a software function - in this case **TIMERx_OVF_vect** that has to process the reason of interrupt. In this case, apply another step pattern to the motor.
4 - Typical for AVR core, the bit that indicated a timer compare match occurred (how the timer is used: this is called the Timer Overflow Flag (TOVx) is cleared without software intervention.
5 - When the interrupt handling is finished, the processor state is restored.
6 - The processor continues what it was doing in step 1 as if it was never interrupted.

Some considerations.

1 - If the processor was doing something that is time-sensitive (for instance, waiting for an I/O line to become low or high), that work may still be affected by the interrupt.
If a specific job is time sensitive, that job should temporary disable interrupts and re-enable as soon as possible when the timing critical function has finished. Note Arduino ms timer uses Timer0. If one disables interrupts overall, the ms timer registration will also be stopped (Arduino timer will not be accurate anymore). Also motor pulses will be stopped.
2 An interrupt function must be as short as possible and it should never block operations (for instance, start waiting for something else to occur). The AVR only allows one interrupt to be processed at any time (or one really has to foresee that software function it can be interrupted once more). So while a motor step function is performed by TIMERx_OVF_vect function, no other interrupt will be accepted. For instance: the Arduino millis timer will not be serviced while a motor step is executed but delayed till that motor step function has finished.
3 Since the OCRx register (as a 16bit value) can be modified at any time, it may occur the present TCNT register was already higher than the updated OCRx value. This may happen when the motor RPM is supposed to accelerate. In such cases, it will take a - to computers considerable - amount of time before the TOV event occurs. This may result the motor - when accelerating makes one slower step and than speeds up again to the normal step rate. However, this behavior is not often seen in real life but occurs most if the complete timing is handled by the interrupt routine. The only way to improve this would be to:

1 - Always call the function **motorTimerUpdateCheck** from the main function.
2 - Change that function so that it does not write directly to OCRxA, but instead write the two bytes to two RAM parameters (except the 1st time when the timer is initialized).
3 - OCRxA is always updated in the **TIMERx_OVF_vect** function from these 2 RAM parameters.

This change would make the **TIMERx_OVF_vect** function very short, with low probability any updates to OCRxA register in sync with the TCNT reset (so low probability it counted much up from zero).

All the used timer registers listed (only used bits are explained). Bits not explained are written zero.

Register Name	D7	D6	D5	D4	D3	D2	D1	D0	
TCCRxA	COMxA1	COMxA0	COMxB1	COMxB0	COMxC1	COMxC0	WGMx1	WGMx0	**Timer Counter Control Register A**

These 2 bits must be '1' to select the FAST PWM mode used by this software.

TCCRxB	ICNCx	ICNSx	-	WGMx3	WGMx2	CSx2	CSx1	CSx0	**Timer Counter Control Register B**

These 2 bits must be '1' to select the FAST PWM mode used by this software.

TCCRxC	FOCxA	FOCxB	FOCxC	-	-	-	-	-	**Timer Counter Control Register C**
TCNTxH	D15	D14	D13	D12	D11	D10	D9	D8	Upper byte of Timer Counter.
TCNTxL	D7	D6	D5	D4	D3	D2	D1	D0	Lower byte of Timer Counter.
OCRxAH	D15	D14	D13	D12	D11	D10	D9	D8	Upper byte of Output Compare A
OCRxAL	D7	D6	D5	D4	D3	D2	D1	D0	Lower byte of Output Compare B

These 2 registers are programmed with the step motor interval.

There are OCR registers B and C as well, but we don't use. Also, ICR (input capture register) is not used.

TIMSKx	-	-	ICIEx	-	OCIExC	OCIExB	OCIExA	TOIEx	Timer Interrupt Mask Register

When this bit is '1', TOV interrupt is enabled (used to advance the stepper motor by 1 step).

TIFRx	-	-	ICFx	-	OCFxC	OCFxB	OCFxA	TOVx

This bit is set by the AVR hardware, when TCNTx matches OCRxA.

The WGM bits select the pre-scaler output. Possible options are:

WGMx3	WGMx2	WGMx1	WGMx0	
1	1	1	1	Fast PWM Mode

CSx2	CSx1	CSx0	
0	0	0	No Timer Clock Source (does not count)
0	0	1	IO_CLK / 1
0	1	0	IO_CLK / 8
0	1	1	IO_CLK / 64
1	0	0	IO_CLK / 256
1	0	1	IO_CLK / 1024
1	1	0	External clock - cannot be used
1	1	1	External clock - cannot be used

CS selects the step motor interval.

Example how we fit a 32bit virtual timer in the 16bit hardware timer.
For example, we select a step interval of 0x12345.

AVR clock:	16000000 Hz
Virtual 32bit	0x000E2345
Decimal =	926533
Steps/revolution	200
Steps per second:	17
Expected RPM:	5.1

This is part of the **motorTimerUpdateCheck** function

```
unsigned char cs = (1 << CS10);
if ((timTop32 & 0xFFFF0000UL) != 0)
{
  timTop32 >>= 3;
  cs = (1 << CS11);
}
if ((timTop32 & 0xFFFF0000UL) != 0)
{
  timTop32 >>= 3;
  cs = (1 << CS11) | (1 << CS10);
}
if ((timTop32 & 0xFFFF0000UL) != 0)
{
  timTop32 >>= 2;
  cs = (1 << CS12);
}
if ((timTop32 & 0xFFFF0000UL) != 0)
{
  timTop32 >>= 2;
  cs = (1 << CS12) | (1 << CS10);
}
if ((timTop32 & 0xFFFF0000UL) != 0)
{…
}
unsigned char lo = (unsigned char) (timTop32);
TCCR1B = (1 << WGM13) | (1 << WGM12) | cs;
timTop32 >>= 8;
cs = (unsigned char) (timTop32);
OCR1AH = cs;
OCR1AL = lo;
```

timTop32 Value		Result of the "if" test:	cs value
			1 << CS10 (prescaler = 1)
926533	000E2345	000E	

Since the result is not zero, this step is executed.

115816	0001C468		
			1 << CS11 (prescaler = 8)
115816	0001C468	0001	

Since the result is not zero, this step is executed.

14477	0000388D	
		1 << CS11 + 1 << CS10 (prescaler = 64)

Result is now zero, step is not executed. Value written to the timer.

So the wanted interval was	000E2345	926533
But in reality, it will be	000E2340	926528
		99.999% accurate

Since the result is now zero, this step is not executed.

This is only there to detect a virtual timer value exceeds 1024 * 65536.
Since this is not the case, this code is skipped.

The low byte of the 16bit timer value is stored in lo.
WGMx3 + WGMx2 are set to '1' (half the init for FAST PWM mode) / prescaler is set.
timTop is now only the upper byte of the remaining 16bit value
This 8bit top value is preserved in cs.
Here we write the timer output compare register. The high byte is written first.
Followed by the low byte.

5 How to use with a Unipolar Motor Driver.

As shown in section 2, a unipolar stepper motor driver has 4 individual outputs (as is the case with Arduino motor shield). For that reason the software setup is different.
In this case, any I/O can be assigned to the motor driver, **all motor driver pins must belong to the same AVR I/O Port** (see also section 4). These I/O Ports are available:
For instance, one can consider to use D4, D39, D40 and D41, since all belong to port G.
Not all pins are available on each AVR processor. The amount of available ports on the Arduino Mega 2560 is considerable higher vs. the Arduino UNO.

AVR Port	Pin	Uno	Mega
A	0	-	D22
	1	-	D23
	2	-	D24
	3	-	D25
	4	-	D26
	5	-	D27
	6	-	D28
	7	-	D29
B	0	D8	D53
	1	D9	D52
	2	D10	D51
	3	D11	D50
	4	D12	D10
	5	D13	D11
	6	-	D12
	7	-	D13
C	0	A0	D37
	1	A1	D36
	2	A2	D35
	3	A3	D34
	4	A4	D33
	5	A5	D32
	6	RESET	D31
	7	-	D30

AVR Port	Pin	Uno	Mega
D	0	D0	D21
	1	D1	D20
	2	D2	D19
	3	D3	D18
	4	D4	-
	5	D5	-
	6	D6	-
	7	D7	D38
E	0	-	D0
	1	-	D1
	2	-	-
	3	-	D5
	4	-	D2
	5	-	D3
	6	-	-
	7	-	-
F	0	-	A0
	1	-	A1
	2	-	A2
	3	-	A3
	4	-	A4
	5	-	A5
	6	-	A6
	7	-	A7

AVR Port	Pin	Uno	Mega
G	0	-	D41
	1	-	D40
	2	-	D39
	3	-	-
	4	-	-
	5	-	D4
H	0	-	D17
	1	-	D16
	2	-	-
	3	-	D6
	4	-	D7
	5	-	D8
	6	-	D9
	7	-	
J	0	-	D15
	1	-	D14
	2	-	-
	3	-	-
	4	-	-
	5	-	-
	6	-	-
	7	-	-

AVR Port	Pin	Uno	Mega
K	0	-	A8
	1	-	A9
	2	-	A10
	3	-	A11
	4	-	A12
	5	-	A13
	6	-	A14
	7	-	A15
L	0	-	D49
	1	-	D48
	2	-	D47
	3	-	D46
	4	-	D45
	5	-	D44
	6	-	D43
	7	-	D42

To disable the Arduino Motor Shield, comment out the **#define ARDUINO_MOTOR_SHIELD** statement in the header file.
Further, one has to define the I/O Port and the Data Direction Register (to make sure the driver sets the correct pins as output):

MOTOR_IOPORT_?	This can be **MOTOR_IOPORT_A** … **MOTOR_IOPORT_L** (any of the above ports that has sufficient pins available to drive the motor).
MOTOR_PATTERN0	Contains the motor pattern (the 7th bit on the left, the 0th bit on the right). For instance **B00110000** ->In this case, bit 5 and 4 are driven high (when set in DDR)
MOTOR_PATTERN1	
MOTOR_PATTERN2	
MOTOR_PATTERN3	
MOTOR_OUTPUTS	Each bit output to a driver pin has to be set in this pattern. For example (**B11110000**) sets the upper 4 bits in the port as output.

6. Software to compensate AVR hardware restrictions.

From the above, we know the hardware timer is 16bit and two divider stages affect the speed it counts (CLKPR and TIMER1/3/4/5 prescaler).

What I didn't want is to hardcode the prescaler for maximal flexibility. For example, at 16MHz, using a fixed prescaler of 1 would result:

highest step rate	25000	steps/s
lowest step rate	244.140625	steps/s

This would result a 200 step/revolution motor would spin at a minimal of 1.5 rotations/s (~90RPM) at lowest possible speed.

We could also use the prescaler of 8 hardcoded, but that would still lead to 12RPM as minimal speed. Bottom line: using a fixed prescaler is not flexible.

I did not want to limit the steps this way by software, because that would make the software too much depending on the hardware. Instead, I defined a virtual 32bit TOP register.

Using a 32bit virtual TOP register, the maximal step rate is still the same as above. The lowest step rate however becomes (under the same conditions):

lowest step rate	0.238418579	steps/s

The result comes from the hardware timer prescaler (1024) and the 16bit range of the timer. So in theory, about every step rate between 25000 and 0.24 steps/s is possible from software point of view.

If we consider the 2nd timer (Timer2): this is only an 8bit hardware timer. Without additional software to deal with this restriction, the minimal step rate would be:

IO Clock	16000000	PrescaleMax	1024	Tim2HW	255	61.27451	steps/s	On a	200	step/revolution motor this is minimal of	18.38235 RPM

A 200 step/revolution motor would need 838.8608 seconds for one rotation!

A function called **timerUpdate** is provided for this. The function:

- Set TIMER1/3/4/5 prescaler to 1
- If the virtual 32bit timer fits in 16bit, use as-is.
- else divide the virtual 32bit timer by 8 and select the hardware timer prescaler = 1/8th.
- if the remaining virtual 32bit timer result still doesn't fit in 16bit, perform additional division by 8 (so original is now divided by 64)
- select timer prescaler 1/64th.
- if the remaining virtual 32bit timer result still doesn't fit in 16bit, perform additional division by 4 (so original is now divided by 256)
- select timer prescaler 1/256th.
- if the remaining virtual 32bit timer result still doesn't fit in 16bit, perform additional division by 4 (so original is now divided by 1024)
- select timer prescaler 1/1024th.
- if the remaining virtual 32bit timer result still doesn't fit in 16bit, set timer error detected.

This way, the software selects the best possible pre-scaler vs. the actual virtual 32bit timer value. The timer hardware 16bit TOP register is loaded vs. the best fitting virtual 32bit timer TOP.

Examples can be computed filling in the yellow fields. The Excel formula act as the real software implementation; the 1st allows calculation based on step rate, the 2nd based on RPM.

Wanted step rate	180	steps/s
Oscillator	16000000	Hz *(This is most likely fixed on your Arduino)*
Steps/revolution	200	*(Motor specific)*
CLKPR	1	*(use only 1, 2, 4, 8, 16, 32, 64 or 256)*
IO Clock	16000000	Hz
32bit Virtual TOP	88888	*(when the result no longer fits in 16bit hardware, the foreground color changes to red; this means the prescaler kicks in).*
16bit Timer TOP	11111	*(0 means the timer top cannot be computed)*
16bit Timer Prescale	8	*(0 means the timer divider cannot be computed)*
Real steps	180.0018	steps/s
Deviation Real-Wante	0.00100%	
RPM:	54.00054001	
Rotations/s	0.900	

Wanted RPM:	10	
Oscillator	16000000	Hz *(This is most likely fixed on your Arduino)*
Steps/revolution	200	*(Motor specific)*
CLKPR	1	*(use only 1, 2, 4, 8, 16, 32, 64 or 256)*
Needed step rate	33	steps/s
IO Clock	16000000	Hz
32bit Virtual TOP	484848	*(when the result no longer fits in 16bit hardware, the foreground color changes to red; this means the prescaler kicks in).*
16bit Timer TOP	60606	*(0 means the timer top cannot be computed)*
16bit Timer Prescale	8	*(0 means the timer divider cannot be computed)*
Real steps	33.000033	steps/s

Deviation Real-Wante	0.00010%
RPM:	9.9000099
RPM Deviation	-0.999901%
Rotations/s	0.165

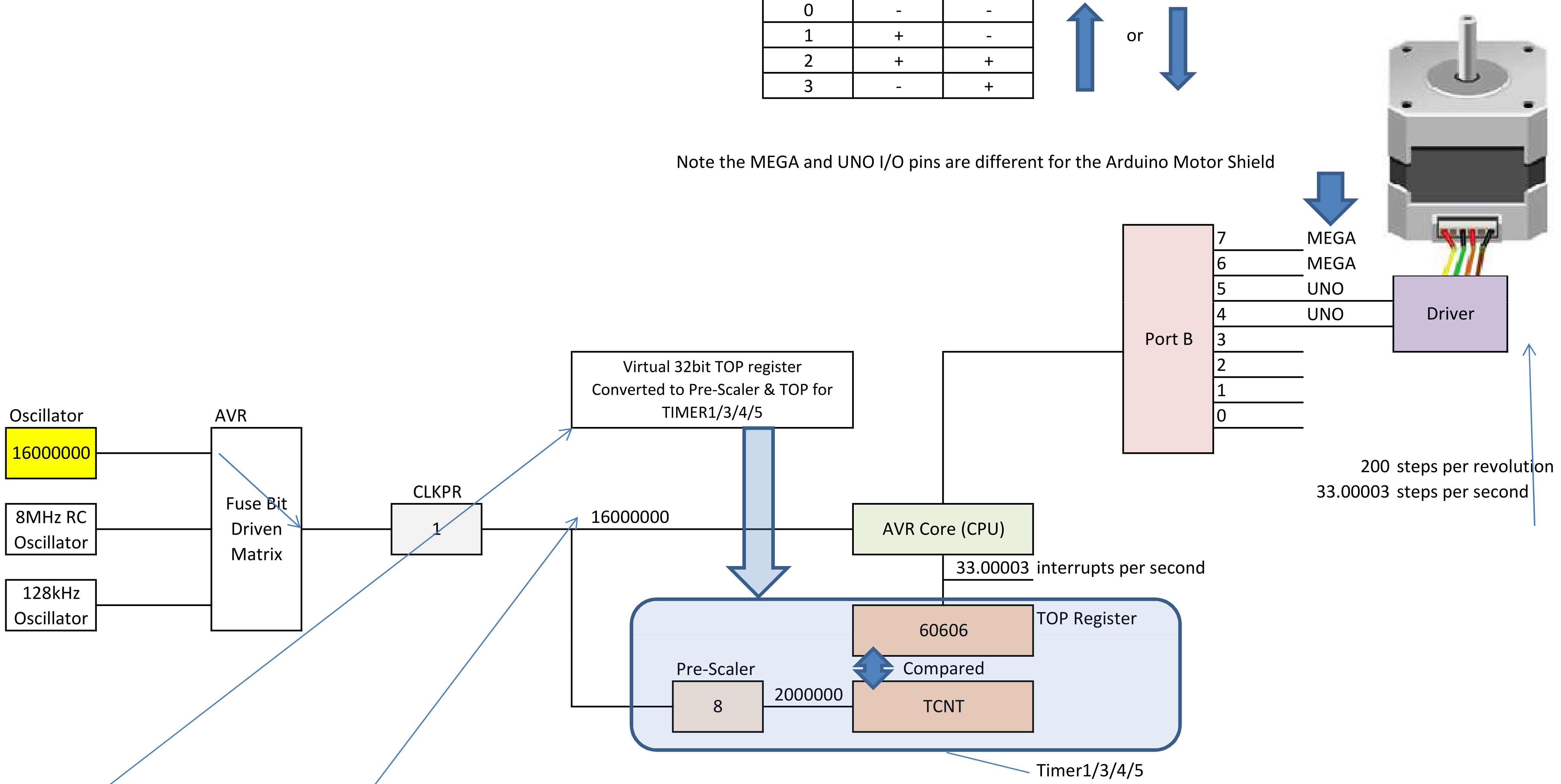
If we put the RPM input back in the above diagram:

Step	Coil A	Coil B
0	-	-
1	+	-
2	+	+
3	-	+

Direction depends on sequencer:

10 RPM

Note the MEGA and UNO I/O pins are different for the Arduino Motor Shield



7. Software functions

motorTimerCalcPerSecond

Calculates the 32bit virtual TOP timer, required to apply motorPulses per second

Argument:	uint16_t motorPulses	The amount of pulses one wants to apply every second. The amount of pulses per revolution is motor dependent.
Return value	uint32_t	

The motor speed is expressed in pulses per second. As shown in the calculations above, if one knows the amount of steps the motor requires for one revolution, the RPM is ((steps_per_second) / (steps_per_revolution) * 60).

Example 200 steps/Revolution 100 steps/s = 30 RPM

motorShieldInit

Requires no arguments, returns nothing. Must be called once to bring the motor outputs in a known state; preferably in the Arduino setup function.

motorTimerGetIOclocksPerSecond

Returns the amount of I/O clocks per second. This is the clock after being optionally divided by the CLKPR settling.

Return value uint32_t

motorTimerUpdateCheck

This function is only required if the motor speed has to be altered during stepping. It must either be called by the user about every 2-50ms (the faster the motor turns, interval becomes shorter)
No arguments are required, no arguments are returned. If needed, it modifies the 32bit virtual TOP register and updates TIMER1/3/4/5 settling to vary the rotation speed of the motor.

motorReadStatus

The return value is unsigned and combines one or more from these bits:

Return value	uint8_t	
	MOTOR_STAT_FAULT	A faulty condition is discovered (ex. TIMER1/3/4/5 TOP value out-of-bounds.
	MOTOR_STAT_FAULT_CALLED	Indicates the error callback function was executed (it will be executed only once per **motorSetSpeed** function call).
	MOTOR_STAT_TIMER_DIV_ERROR	Virtual timer does not fit in TIMER1/3/4/5 hardware.
	MOTOR_STAT_TIMER_DIV_CALLED	Error function has been called for TIMER1/3/4/5 hardware limit (prevents it's called again till new event occurs).
	MOTOR_STAT_REACHED	The motor end position has been reached / is nog longer spinning.
	MOTOR_STAT_REACH_CALLED	The user defined callback function for motor position reached has been called once (prevents called again till new event occurs).
	MOTOR_STAT_SPIN_FIXED	Motor spins at a fixed RPM.

One can perform the following to check if the motor has reached the end position:

```
if (motorReadStatus () & MOTOR_STAT_REACHED) { /* the motor has reached the end position */ }
```

Another example may be if the motor RPM exceeds the lowest possible value (the virtual 32bit TOP register is equal or above 1024 * 65536).

motorSetSpeed

The major function that initiates all the motor stepping related work. Therefore, it has may arguments. Some are optional.

Return value	int8_t	Returns an error code (negative), warning (positive value) or zero if no errors nor warnings are detected.	
Arguments:	uint8_t control	Control information for the driver.	
		MOTOR_CTRL_DRV_EN	Sets the L298 PWM pins of the motor driver shield HIGH, otherwise LOW (and no motor drive is performed). *If not included in the control byte, all parameters below have no effect.*
		MOTOR_CTRL_DIRECTION	When used, the motor direction is inverted by playing the port bit sequence in the reversed order.
		MOTOR_CTRL_ENDPOINT_MILLIS	limit is ms relative from the start of the motorSetSpeed function (default is motor steps), including eventual motor start-up.
		MOTOR_CTRL_SPEED_VARIABLE	When used, the motor speed is changed by the driver on the fly. This bit can be combined with other bits: *If not included in the control byte, all parameters below have no effect.*
		MOTOR_CTRL_FASTER	Every update event, **timerModification** is deducted from **timerTop32virtual** If not set combined with **MOTOR_CTRL_SPEED_VARIABLE,** every update event **timerModification** is added to **timerTop32virtual**, increasing the timer interval period.
		MOTOR_CTRL_SLOWER	Dummy declaration to make code more read-able (if **MOTOR_CTRL_FASTER** is not set, SLOWER is default).
		MOTOR_CTRL_VARIABLE_PWR	Modifies timerModification in such way that motor RPM changes become linear.
		MOTOR_CTRL_IRQ_TIMING	If not set, the user has to call **motorTimerUpdateCheck** at least every 20ms.
		Note: multiple arguments can be given separated by pipe (OR). For instance: **MOTOR_CTRL_DIRECTION | MOTOR_CTRL_DRV_EN**.	
	uint32_t timerTop32virtual	The initial 32bit timer interval register. On a 16MHz AVR, one step is applied every 16000000 divided by this value (in seconds).	
	uint32_t limit	if **MOTOR_CTRL_ENDPOINT_MILLIS** is not used, the driver stop sending step pulses when the total amount of steps equals or exceeds this value (0 = never stop). if **MOTOR_CTRL_ENDPOINT_MILLIS** is set, the driver stops sending motor pulses this amount of ms after this function was called. Set this value to 0 if one does not want the motor pulses to stop (continuous run).	
	uint32_t timerModification	Only required if above mentioned **MOTOR_CTRL_SPEED_VARIABLE** is included in the control settling.	
	uint16_t timerUpd_ms	Sets the amount of ms (accuracy depends on calling the **motorTimerUpdateCheck**) interval to compute the virtual 32bit Timer TOP register.	

This parameter is only used if **MOTOR_CTRL_SPEED_VARIABLE** is used in the control settling.

uint16_t modCounts — After this amount of **timerTop32virtual** update has been counted, the update event is suppressed (*)
This parameter is only used if **MOTOR_CTRL_SPEED_VARIABLE** is used in the control settling.
(*) whoever comes first suppresses update events.

Note: red parameters are optional in the function call. Default values are zero. When a fixed motor speed is required, one can skip these parameters in the function call.

Examples:

motorSetSpeed (**MOTOR_CTRL_DRV_EN**, 32000UL); //On a 16MHz AVR runs the motor at a fixed interval of 500 steps/s (on a 200 step/revolution motor this is 150RPM).
motorSetSpeed (**MOTOR_CTRL_DRV_EN**, 64000UL, 20000UL); //On a 16MHz AVR runs the motor at a fixed interval of 250 steps/s (on a 200 step/revolution motor this is 75RPM) and stops after 100 revolutions (=20000 steps).

How to calculate things:

The arduino runs at	16000000	Hz							
We have motor requiring	200	steps per revolution. We want to start at	2	RPM and increase	4	time per second, up to	120	RPM
		Start with motor =	6	steps/s	29	updates	800	steps/s
We need one update event every	250	ms Real:	1.8	RPM			240	RPM
The initial virtual 32bit timer TOP register	2666666							
The final virtual 32bit timer TOP register	20000	Modify: 91264						

```
motorSetSpeed (   MOTOR_CTRL_SPEED_VARIABLE             //uint8_t  control     - The speed is not constant
                | MOTOR_CTRL_IRQ_TIMING                 //                     - We do not call the timer adjust function, so it's done in the TIMER1/3/4/5 ISR, blocking the core a bit
                                                        //                       longer for other interrupts.
                | MOTOR_CTRL_FASTER                     //                     - The speed has to increase.
                | MOTOR_CTRL_DRV_EN                     //This is a security that must be set to enable the motor driver, without the motor is stopped.
                , 2666666UL                             //uint32_t timerTop32virtual. Add "UL" to direct parameters because this is unsigned long value.
                                                        //One may also obtain this value as uint32_t initial32bitTop = motorTimerCalcPerSecond (200 / 60) //start at 1 RPM, but this
                                                        //will create additional rounding errors.
                , 0UL                                   //uint32_t limit not used: the motor keeps spinning when reaching the final RPM (there is no "endpoint" in steps.
                , 91264UL                               //uint32_t timerModification = 0,          //Virtual 32bit correction on timerTop32Virtual.
                , 250                                   //Every xx ms, the correction is applied (this must be 50ms or more).
                , 29                                    //uint16_t modCounts: after this amount of virtual32bit timer updates, further updates are disabled.
                  );
```

Note: this shows how control bits can be combined (this is an OR function in C code).

Below declarations define the Arduino digital pin numbers for various motor shield pins. However, one has to take into account MOTOR_DIR is not taken from this table and hardcoded for Uno & Mega only.

MOTOR_ENABLE_A	3	
MOTOR_BREAK_B	8	
MOTOR_BREAK_A	9	
MOTOR_ENABLE_B	11	
MOTOR_DIR_A	12	Note: used by the driver for the initial motor driver configuration, but not during interrupts.
MOTOR_DIR_B	13	Note: used by the driver for the initial motor driver configuration, but not during interrupts.

These functions are at present not guaranteed and are not guaranteed:

motorCallBackPositionReached

Arguments: (void (*userFunction) ()) — The user can provide here an own defined function that gets called when the motor reaches the end position. This will only be called if the user calls the **motorTimerUpdateCheck** on a timed basis. If **MOTOR_CTRL_IRQ_TIMING** is used, this has no meaning. Reason is that we have no control over this user function and an interrupt should never be a blocking function - what cannot be guaranteed. Therefor, the user function is not called when **MOTOR_CTRL_IRQ_TIMING** is used.

motorCallBackFault

Arguments: (void (*userFunction) ()) — Similar to **motorCallbCackPositionReached**. This function gets called on detecting an error (only if **MOTOR_CTRL_IRQ_TIMING** is not used).

8. RPM vs. Timer

In order to have a linear RPM behavior, an option is built in to modify the timerModification parameter. However, this limits the amount of modCounts to maximal ~30, most likely even much less. Below table shows how the RPM behaves vs. timerTop32virtual. By shifting the timerModification value left (when slowing down the rotation speed) or right (when increasing the rotation speed), the motor RPM changes by a factor of 2 as can be seen in the 2nd table. Since timerModification is a 32bit parameter, shifting more than this certainly makes the result 0, stopping the motor speed modification. When using this option, one has to calculate the initial RPM, the required modification and based on those two parameters, one can determine the maximal amounts of possible shifts. This option allows to speed up and down in a more user-friendly manner at the expense of somewhat more calculation work by the calling function.

IO Clock		16000000
Steps/Rotation		200
Timer	Steps/s	RPM
1000	16000	4800
2000	8000	2400
3000	5333.333	1600
4000	4000	1200
5000	3200	960
6000	2666.667	800
7000	2285.714	685.7143
8000	2000	600
9000	1777.778	533.3333
10000	1600	480
11000	1454.545	436.3636
12000	1333.333	400
13000	1230.769	369.2308
14000	1142.857	342.8571
15000	1066.667	320
16000	1000	300
17000	941.1765	282.3529
18000	888.8889	266.6667
19000	842.1053	252.6316
20000	800	240
21000	761.9048	228.5714
22000	727.2727	218.1818
23000	695.6522	208.6957
24000	666.6667	200
25000	640	192
26000	615.3846	184.6154
27000	592.5926	177.7778
28000	571.4286	171.4286
29000	551.7241	165.5172
30000	533.3333	160
31000	516.129	154.8387
32000	500	150
33000	484.8485	145.4545
34000	470.5882	141.1765
35000	457.1429	137.1429
36000	444.4444	133.3333
37000	432.4324	129.7297
38000	421.0526	126.3158
39000	410.2564	123.0769
40000	400	120

Set **MOTOR_CTRL_VARIABLE_PWR**

Each Timer modification will be half or twice the previous modification value (for as far it fits in 32bit integer)

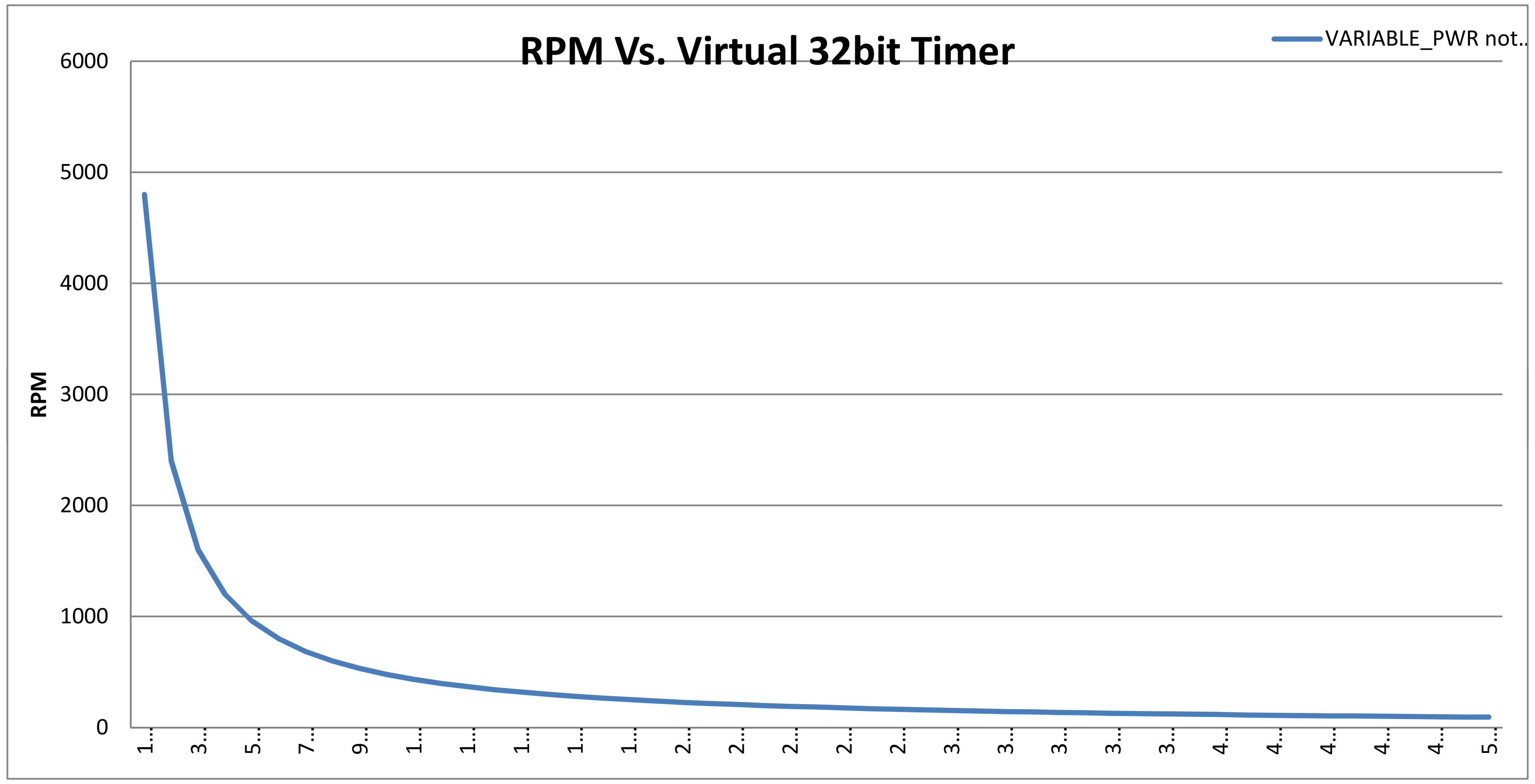
Increment	Timer1	Steps/s	**RPM**	ModCount
0	1000	16000	4800	0
1000	2000	8000	2400	1
2000	4000	4000	1200	2
4000	8000	2000	600	3
8000	16000	1000	300	4
16000	32000	500	150	5
32000	64000	250	75	6
64000	128000	125	37.5	7
128000	256000	62.5	18.75	8
256000	512000	31.25	9.375	9
512000	1024000	15.625	4.6875	10
1024000	2048000	7.8125	2.34375	11
2048000	4096000	3.90625	1.171875	12
4096000	8192000	1.953125	0.585938	13
8192000	16384000	0.976563	0.292969	14
16384000	32768000	0.488281	0.146484	15
32768000	65536000	0.244141	0.073242	16

MOTOR_CTRL_SLOWER

MOTOR_CTRL_FASTER

Start	3.333333 steps/s		1 RPM	
Target	500 steps/s		150 RPM	
Mods <=	10			
			Tim1Mod	
Tim1start	4800000	4800000		
Tim1end	32000	2415000		2385000
Tim1decr	4768000	1222500		1192500
		626250		596250
		328125		298125
		179063		149062
		104532		74531
		67267		37265
		48635		18632
		39319		9316
		34661		4658
		32332		2329

The curve shows the RPM when we change the timer divider on a linear basis.

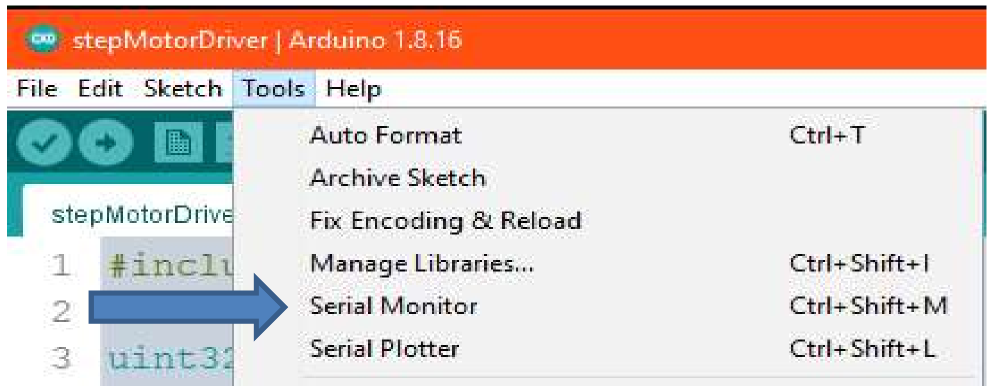


41000	390.2439	117.0732
42000	380.9524	114.2857
43000	372.093	111.6279
44000	363.6364	109.0909
45000	355.5556	106.6667
46000	347.8261	104.3478
47000	340.4255	102.1277
48000	333.3333	100
49000	326.5306	97.95918
50000	320	96
51000	313.7255	94.11765

9.Conclusions.

Is there still some math to be done: if one wants to use RPM, calculate initial and final RPM: there is surely some extra requirement.

But if one uses steps, just wants linear behavior of the speed increases / decreases, this code is a good starting point. Since RPM calculations can be done in the loop function, nearly un-affected by present motor rotations, it becomes more realistic to assume slower floats can be used without affecting the application too much.


stepMotorDriver | Arduino 1.8.16
File Edit Sketch Tools Help
stepMotorDrive
#inclu
uint32
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L

10.Examples / validation

Below some code examples used during the validation. The code also shows the use of the callback function.

Case 1: Decrease speed while spinning

```
18:01:58.018 -> IRQ Timer 1 Enabled
18:01:59.020 -> >>>motorSetSpeed
18:01:59.020 -> Arduino Mega
18:01:59.020 -> motor_ctrl_irq: 0xF0
18:01:59.067 -> motor_timer_top32virt_irq: 15000
18:01:59.067 -> motor_timer_modify_irq: 64
18:01:59.067 -> motor_step_count_limit_irq: 8844
18:01:59.067 -> motor_time_irq: 1009
18:01:59.067 -> motor_stat_irq: 0x00
18:01:59.121 -> motor_update_time_irq: 1000
18:01:59.121 -> motor_millis_endtime_irq: 0
18:01:59.121 -> motor_modCounts_irq: 19
18:01:59.121 -> motor_pattern_irq: 3
18:01:59.121 -> Exit Code: 00                      Update
18:01:59.168 -> motor_ctrl_irq: 0xF0
18:01:59.168 -> motor_timer_top32virt_irq: 15000        64
18:01:59.168 -> motor_timer_modify_irq: 64
18:01:59.168 -> motor_step_count_limit_irq: 8722
18:01:59.221 -> motor_time_irq: 1009
18:01:59.221 -> motor_stat_irq: 0x00
18:01:59.221 -> motor_update_time_irq: 1000
18:01:59.221 -> motor_millis_endtime_irq: 0
18:01:59.221 -> motor_modCounts_irq: 19
18:01:59.268 -> motor_pattern_irq: 1
18:02:00.180 -> motor_ctrl_irq: 0xF0
18:02:00.180 -> motor_timer_top32virt_irq: 15064       128
18:02:00.180 -> motor_timer_modify_irq: 128
18:02:00.180 -> motor_step_count_limit_irq: 7656
18:02:00.180 -> motor_time_irq: 2009
18:02:00.224 -> motor_stat_irq: 0x00
18:02:00.224 -> motor_update_time_irq: 1000
18:02:00.224 -> motor_millis_endtime_irq: 0
18:02:00.224 -> motor_modCounts_irq: 18
18:02:00.271 -> motor_pattern_irq: 2
18:02:01.173 -> motor_ctrl_irq: 0xF0
18:02:01.173 -> motor_timer_top32virt_irq: 15192       256
18:02:01.173 -> motor_timer_modify_irq: 256
18:02:01.226 -> motor_step_count_limit_irq: 6594
18:02:01.226 -> motor_time_irq: 3009
18:02:01.226 -> motor_stat_irq: 0x00
```

```
//Remove the comment #define DEBUG_BAUDRATE (115200) and open a serial monitor to the port the Arduino is connnected to.
void
loop ()
{
  static uint8_t once = 1;
  if (once == 1)
  {
    Serial.begin (115200);
    int8_t ec = motorSetSpeed (  MOTOR_CTRL_SPEED_VARIABLE
                                | MOTOR_CTRL_IRQ_TIMING
                                | MOTOR_CTRL_VARIABLE_PWR
                              //| MOTOR_CTRL_DIRECTION
                                | MOTOR_CTRL_SLOWER
                                | MOTOR_CTRL_DRV_EN,
                               15000UL,        //uint32_t: Virtual 32bit timer TOP
                               8900UL,         //uint32_t: amount of steps
                               64UL,           //uint32_t: correction on timerTop
                               UPDATE_INTERVAL,//uint16_t: Correction interval ms
                               19);            //uint16_t: Correction Limitter.

    if (ec < 0) { }                            //Optional error handler.
    once = 0;
  }
  #ifdef DEBUG_BAUDRATE
    uint32_t now = millis ();
    static uint32_t lastUpdate = 0;
    if ((now - lastUpdate) > UPDATE_INTERVAL)
    {
      if (TIMSK1 & (1 << TOIE0)) debugDump ();
      else if ((once & 2) == 0)
      {
        motorSetSpeed ();                      //This stops the motor.
        once |= 2;
      }
      lastUpdate = now;
    }
  #endif
}
```

```
18:02:01.226 -> motor_update_time_irq: 1000
18:02:01.226 -> motor_millis_endtime_irq: 0
18:02:01.273 -> motor_modCounts_irq: 17
18:02:01.273 -> motor_pattern_irq: 0
18:02:02.175 -> motor_ctrl_irq: 0xF0
18:02:02.175 -> motor_timer_top32virt_irq: 15448                512
18:02:02.175 -> motor_timer_modify_irq: 512
18:02:02.229 -> motor_step_count_limit_irq: 5543
18:02:02.229 -> motor_time_irq: 4009
18:02:02.229 -> motor_stat_irq: 0x00
18:02:02.229 -> motor_update_time_irq: 1000
18:02:02.229 -> motor_millis_endtime_irq: 0
18:02:02.275 -> motor_modCounts_irq: 16
18:02:02.275 -> motor_pattern_irq: 1
18:02:03.180 -> motor_ctrl_irq: 0xF0
18:02:03.180 -> motor_timer_top32virt_irq: 15960               1024
18:02:03.231 -> motor_timer_modify_irq: 1024
18:02:03.231 -> motor_step_count_limit_irq: 4512
18:02:03.231 -> motor_time_irq: 5009
18:02:03.231 -> motor_stat_irq: 0x00
18:02:03.231 -> motor_update_time_irq: 1000
18:02:03.278 -> motor_millis_endtime_irq: 0
18:02:03.278 -> motor_modCounts_irq: 15
18:02:03.278 -> motor_pattern_irq: 3
18:02:04.180 -> motor_ctrl_irq: 0xF0
18:02:04.180 -> motor_timer_top32virt_irq: 16984               2048
18:02:04.233 -> motor_timer_modify_irq: 2048
18:02:04.233 -> motor_step_count_limit_irq: 3519
18:02:04.233 -> motor_time_irq: 6009
18:02:04.233 -> motor_stat_irq: 0x00
18:02:04.233 -> motor_update_time_irq: 1000
18:02:04.280 -> motor_millis_endtime_irq: 0
18:02:04.280 -> motor_modCounts_irq: 14
18:02:04.280 -> motor_pattern_irq: 0
18:02:05.182 -> motor_ctrl_irq: 0xF0
18:02:05.236 -> motor_timer_top32virt_irq: 19032
18:02:05.236 -> motor_timer_modify_irq: 4096
18:02:05.236 -> motor_step_count_limit_irq: 2593
18:02:05.236 -> motor_time_irq: 7009
18:02:05.236 -> motor_stat_irq: 0x00
18:02:05.283 -> motor_update_time_irq: 1000
18:02:05.283 -> motor_millis_endtime_irq: 0
18:02:05.283 -> motor_modCounts_irq: 13
18:02:05.283 -> motor_pattern_irq: 0
18:02:06.238 -> motor_ctrl_irq: 0xF0
18:02:06.238 -> motor_timer_top32virt_irq: 23128
18:02:06.238 -> motor_timer_modify_irq: 8192
18:02:06.238 -> motor_step_count_limit_irq: 1777
18:02:06.238 -> motor_time_irq: 8009
18:02:06.238 -> motor_stat_irq: 0x00
```

```
18:02:06.285 -> motor_update_time_irq: 1000
18:02:06.285 -> motor_millis_endtime_irq: 0
18:02:06.285 -> motor_modCounts_irq: 12
18:02:06.285 -> motor_pattern_irq: 2
18:02:07.241 -> motor_ctrl_irq: 0xF0
18:02:07.241 -> motor_timer_top32virt_irq: 31320
18:02:07.241 -> motor_timer_modify_irq: 16384
18:02:07.241 -> motor_step_count_limit_irq: 1116
18:02:07.241 -> motor_time_irq: 9009
18:02:07.287 -> motor_stat_irq: 0x00
18:02:07.287 -> motor_update_time_irq: 1000
18:02:07.287 -> motor_millis_endtime_irq: 0
18:02:07.287 -> motor_modCounts_irq: 11
18:02:07.341 -> motor_pattern_irq: 0
18:02:08.243 -> motor_ctrl_irq: 0xF0
18:02:08.243 -> motor_timer_top32virt_irq: 47704
18:02:08.243 -> motor_timer_modify_irq: 32768
18:02:08.243 -> motor_step_count_limit_irq: 634
18:02:08.290 -> motor_time_irq: 10009
18:02:08.290 -> motor_stat_irq: 0x00
18:02:08.290 -> motor_update_time_irq: 1000
18:02:08.290 -> motor_millis_endtime_irq: 0
18:02:08.290 -> motor_modCounts_irq: 10
18:02:08.343 -> motor_pattern_irq: 3
18:02:09.227 -> motor_ctrl_irq: 0xF0
18:02:09.227 -> motor_timer_top32virt_irq: 80472
18:02:09.261 -> motor_timer_modify_irq: 65536
18:02:09.261 -> motor_step_count_limit_irq: 326
18:02:09.261 -> motor_time_irq: 11009
18:02:09.308 -> motor_stat_irq: 0x00
18:02:09.308 -> motor_update_time_irq: 1000
18:02:09.308 -> motor_millis_endtime_irq: 0
18:02:09.308 -> motor_modCounts_irq: 9
18:02:09.308 -> motor_pattern_irq: 2
18:02:10.249 -> motor_ctrl_irq: 0xF0
18:02:10.249 -> motor_timer_top32virt_irq: 146008        131072
18:02:10.249 -> motor_timer_modify_irq: 131072  <-
18:02:10.296 -> motor_step_count_limit_irq: 143
18:02:10.296 -> motor_time_irq: 12009
18:02:10.296 -> motor_stat_irq: 0x00
18:02:10.296 -> motor_update_time_irq: 1000
18:02:10.296 -> motor_millis_endtime_irq: 0
18:02:10.349 -> motor_modCounts_irq: 8
18:02:10.349 -> motor_pattern_irq: 0
18:02:11.252 -> motor_ctrl_irq: 0xF0
18:02:11.252 -> motor_timer_top32virt_irq: 277080
18:02:11.252 -> motor_timer_modify_irq: 262144
18:02:11.299 -> motor_step_count_limit_irq: 42
18:02:11.299 -> motor_time_irq: 13009
18:02:11.299 -> motor_stat_irq: 0x00
```

```
18:02:11.299 -> motor_update_time_irq: 1000
18:02:11.352 -> motor_millis_endtime_irq: 0
18:02:11.352 -> motor_modCounts_irq: 7
18:02:11.352 -> motor_pattern_irq: 2
18:02:12.001 -> Motor halted
18:02:12.255 -> >>>motorSetSpeed
18:02:12.255 -> Arduino Mega
18:02:12.302 -> motor_ctrl_irq: 0x00
18:02:12.302 -> motor_timer_top32virt_irq: 0
18:02:12.302 -> motor_timer_modify_irq: 0
18:02:12.302 -> motor_step_count_limit_irq: 0
18:02:12.302 -> motor_time_irq: 13009
18:02:12.355 -> motor_stat_irq: 0x00
18:02:12.355 -> motor_update_time_irq: 0
18:02:12.355 -> motor_millis_endtime_irq: 0
18:02:12.355 -> motor_modCounts_irq: 0
18:02:12.355 -> motor_pattern_irq: 0
18:02:12.402 -> Exit Code: 00
```

19 updates where not possible, the step limit has been reached.
The motor was stopped at very low speed.

Case 2: increase speed

```
12:54:08.076 -> IRQ Timer 1 Enabled
12:54:09.132 -> >>>motorSetSpeed
12:54:09.132 -> Arduino Mega
12:54:09.132 -> motor_ctrl_irq: 0xF2
12:54:09.132 -> motor_timer_top32virt_irq: 4800000
12:54:09.132 -> motor_timer_modify_irq: 2385000
12:54:09.179 -> motor_step_count_limit_irq: 0
12:54:09.179 -> motor_time_irq: 1009
12:54:09.179 -> motor_stat_irq: 0x00
12:54:09.179 -> motor_update_time_irq: 1000
12:54:09.232 -> motor_millis_endtime_irq: 0
12:54:09.232 -> motor_modCounts_irq: 11
12:54:09.232 -> motor_pattern_irq: 1
12:54:09.232 -> Exit Code: 00
12:54:09.232 -> motor_ctrl_irq: 0xF2
12:54:09.232 -> motor_timer_top32virt_irq: 4800000
12:54:09.279 -> motor_timer_modify_irq: 2385000
12:54:09.279 -> motor_step_count_limit_irq: 0
12:54:09.279 -> motor_time_irq: 1009
12:54:09.279 -> motor_stat_irq: 0x00
12:54:09.332 -> motor_update_time_irq: 1000
12:54:09.332 -> motor_millis_endtime_irq: 0
12:54:09.332 -> motor_modCounts_irq: 11
12:54:09.332 -> motor_pattern_irq: 1
12:54:10.250 -> motor_ctrl_irq: 0xF2
12:54:10.250 -> motor_timer_top32virt_irq: 4800000
12:54:10.250 -> motor_timer_modify_irq: 2385000
12:54:10.297 -> motor_step_count_limit_irq: 0
12:54:10.297 -> motor_time_irq: 1009
12:54:10.297 -> motor_stat_irq: 0x00
```

Motor:	200	steps/rotation
Ioclk	16000000	Hz
Start conditions:		
	13.08793	steps/s
	3.92638	RPM

```
void
loop ()
{
  static uint8_t once = 1;
  if (once == 1)
  {
    Serial.begin (115200);
    int8_t ec = motorSetSpeed (  MOTOR_CTRL_SPEED_VARIABLE
                               | MOTOR_CTRL_IRQ_TIMING
                               | MOTOR_CTRL_VARIABLE_PWR
                             //| MOTOR_CTRL_DIRECTION
                               | MOTOR_CTRL_FASTER
                               | MOTOR_CTRL_DRV_EN,
                              4800000UL,
                              0UL,
                              2385000UL,
                              UPDATE_INTERVAL,
                              11);

    if (ec < 0) { }
    once = 0;
  //delay (40000);
  //ec = motorSetSpeed (MOTOR_CTRL_DRV_EN,
  //                      24615UL);
  }
  #ifdef DEBUG_BAUDRATE
    uint32_t now = millis ();
    static uint32_t lastUpdate = 0;
    if ((now - lastUpdate) > UPDATE_INTERVAL)
    {
```

```
12:54:10.297 -> motor_update_time_irq: 1000
12:54:10.335 -> motor_millis_endtime_irq: 0
12:54:10.335 -> motor_modCounts_irq: 11
12:54:10.335 -> motor_pattern_irq: 0
12:54:11.284 -> motor_ctrl_irq: 0xF2                      1st update
12:54:11.284 -> motor_timer_top32virt_irq: 1222500          13.08793 steps/s
12:54:11.284 -> motor_timer_modify_irq: 596250               3.92638 RPM
12:54:11.284 -> motor_step_count_limit_irq: 0
12:54:11.284 -> motor_time_irq: 3009
12:54:11.337 -> motor_stat_irq: 0x00
12:54:11.337 -> motor_update_time_irq: 1000
12:54:11.337 -> motor_millis_endtime_irq: 0
12:54:11.337 -> motor_modCounts_irq: 9
12:54:11.337 -> motor_pattern_irq: 0
12:54:12.286 -> motor_ctrl_irq: 0xF2                      2nd update
12:54:12.286 -> motor_timer_top32virt_irq: 626250            25.5489 steps/s
12:54:12.286 -> motor_timer_modify_irq: 298125              7.664671 RPM
12:54:12.286 -> motor_step_count_limit_irq: 0
12:54:12.340 -> motor_time_irq: 4009
12:54:12.340 -> motor_stat_irq: 0x00
12:54:12.340 -> motor_update_time_irq: 1000
12:54:12.340 -> motor_millis_endtime_irq: 0
12:54:12.340 -> motor_modCounts_irq: 8
12:54:12.387 -> motor_pattern_irq: 2
12:54:13.289 -> motor_ctrl_irq: 0xF2                      3rd update
12:54:13.289 -> motor_timer_top32virt_irq: 328125            48.7619 steps/s
12:54:13.289 -> motor_timer_modify_irq: 149062              14.62857 RPM
12:54:13.289 -> motor_step_count_limit_irq: 0
12:54:13.343 -> motor_time_irq: 5009
12:54:13.343 -> motor_stat_irq: 0x00
12:54:13.343 -> motor_update_time_irq: 1000
12:54:13.343 -> motor_millis_endtime_irq: 0
12:54:13.343 -> motor_modCounts_irq: 7
12:54:13.389 -> motor_pattern_irq: 1
12:54:14.292 -> motor_ctrl_irq: 0xF2
12:54:14.292 -> motor_timer_top32virt_irq: 179063
12:54:14.292 -> motor_timer_modify_irq: 74531
12:54:14.345 -> motor_step_count_limit_irq: 0
12:54:14.345 -> motor_time_irq: 6009
12:54:14.345 -> motor_stat_irq: 0x00
12:54:14.345 -> motor_update_time_irq: 1000
12:54:14.345 -> motor_millis_endtime_irq: 0
12:54:14.392 -> motor_modCounts_irq: 6
12:54:14.392 -> motor_pattern_irq: 2
12:54:15.279 -> motor_ctrl_irq: 0xF2
12:54:15.326 -> motor_timer_top32virt_irq: 104532
12:54:15.326 -> motor_timer_modify_irq: 37265
12:54:15.326 -> motor_step_count_limit_irq: 0
12:54:15.326 -> motor_time_irq: 7009
12:54:15.326 -> motor_stat_irq: 0x00
```

```
          if (TIMSK1 & (1 << TOIE0))
          {
            debugDump (ONLY_IF_VARIABLE_SPEED);
          }
          else if ((once & 2) == 0)
          {
            motorSetSpeed ();
            once |= 2;
          }
          lastUpdate = now;
        }
    #endif
  }
```

```
12:54:15.363 -> motor_update_time_irq: 1000
12:54:15.363 -> motor_millis_endtime_irq: 0
12:54:15.363 -> motor_modCounts_irq: 5
12:54:15.410 -> motor_pattern_irq: 1
12:54:16.297 -> motor_ctrl_irq: 0xF2
12:54:16.297 -> motor_timer_top32virt_irq: 67267
12:54:16.350 -> motor_timer_modify_irq: 18632
12:54:16.350 -> motor_step_count_limit_irq: 0
12:54:16.350 -> motor_time_irq: 8009
12:54:16.350 -> motor_stat_irq: 0x00
12:54:16.350 -> motor_update_time_irq: 1000
12:54:16.397 -> motor_millis_endtime_irq: 0
12:54:16.397 -> motor_modCounts_irq: 4
12:54:16.397 -> motor_pattern_irq: 1
12:54:17.299 -> motor_ctrl_irq: 0xF2
12:54:17.299 -> motor_timer_top32virt_irq: 48635
12:54:17.353 -> motor_timer_modify_irq: 9316
12:54:17.353 -> motor_step_count_limit_irq: 0
12:54:17.353 -> motor_time_irq: 9009
12:54:17.353 -> motor_stat_irq: 0x00
12:54:17.353 -> motor_update_time_irq: 1000
12:54:17.400 -> motor_millis_endtime_irq: 0
12:54:17.400 -> motor_modCounts_irq: 3
12:54:17.400 -> motor_pattern_irq: 2
12:54:18.301 -> motor_ctrl_irq: 0xF2
12:54:18.355 -> motor_timer_top32virt_irq: 39319
12:54:18.355 -> motor_timer_modify_irq: 4658
12:54:18.355 -> motor_step_count_limit_irq: 0
12:54:18.355 -> motor_time_irq: 10009
12:54:18.355 -> motor_stat_irq: 0x00
12:54:18.402 -> motor_update_time_irq: 1000
12:54:18.402 -> motor_millis_endtime_irq: 0
12:54:18.402 -> motor_modCounts_irq: 2
12:54:18.402 -> motor_pattern_irq: 2
12:54:19.357 -> motor_ctrl_irq: 0xF2              <- final update
12:54:19.357 -> motor_timer_top32virt_irq: 34661      461.6139  steps/s
12:54:19.357 -> motor_timer_modify_irq: 2329          138.4842  RPM
12:54:19.357 -> motor_step_count_limit_irq: 0
12:54:19.357 -> motor_time_irq: 11009
12:54:19.404 -> motor_stat_irq: 0x00
12:54:19.404 -> motor_update_time_irq: 1000
12:54:19.404 -> motor_millis_endtime_irq: 0
12:54:19.404 -> motor_modCounts_irq: 1
12:54:19.404 -> motor_pattern_irq: 2
```

Case 3: calling the timer update function from main and use callback on position reached
Declare a function that must be executed when the motor stops:
Notify the callback function to the motor driver as shown below.
Further changes are:

- Do NOT include MOTOR_CTRL_IRQ_TIMING in the control byte;
- Define an endpoint (or the motor will not stop spinning - no callback is performed.
- Call the timer update function from your own main loop.

Here the callback function was executed, when the motor reached the endpoint restriction.

```
void
endPosReadched ()
{
  Serial.println ("Motor position reached / rotation stopped");
}

void
setup ()
{
  pinMode (     MOTOR_ENABLE_A, OUTPUT);
  pinMode (      MOTOR_BREAK_B, OUTPUT);
  pinMode (      MOTOR_BREAK_A, OUTPUT);
  pinMode (     MOTOR_ENABLE_B, OUTPUT);
  pinMode (        MOTOR_DIR_A, OUTPUT);
  pinMode (        MOTOR_DIR_B, OUTPUT);
  digitalWrite (MOTOR_ENABLE_A,     LOW);
  digitalWrite ( MOTOR_BREAK_B,     LOW);
  digitalWrite ( MOTOR_BREAK_A,     LOW);
  digitalWrite (MOTOR_ENABLE_B,     LOW);
  digitalWrite (   MOTOR_DIR_A,     LOW);
  digitalWrite (   MOTOR_DIR_B,     LOW);
  Serial.begin (115200);
  motorCallBackPositionReached (endPosReadched);
}

void
loop ()
{
  static uint8_t once = 1;
  if (once == 1)
  {
    int8_t ec = motorSetSpeed (  MOTOR_CTRL_SPEED_VARIABLE
                              //| MOTOR_CTRL_IRQ_TIMING          //Do not use IRQ_TIMING when setting the motor speed (must be called by user instead).
                                | MOTOR_CTRL_VARIABLE_PWR
                              //| MOTOR_CTRL_DIRECTION
                                | MOTOR_CTRL_FASTER             //Increase speed: TOP register is subtracted with decrease value.
                                | MOTOR_CTRL_DRV_EN,
                               4800000UL,                       //uint32_t: Virtual 32bit timer TOP to apply motor steps; 0 = STOP motor.
                               10000UL,                         //uint32_t: Limits amount of steps to execute (0 = never stop) or amount of millis to execute.
                               2385000UL,                       //uint32_t: Virtual 32bit correction on timerTop32Virtual.
                               UPDATE_INTERVAL,                 //uint16_t: Every xx ms, the correction is applied.
```

```
14:05:38.197 -> From now you see increasing counter from main loop
14:05:39.199 -> 127269
14:05:40.202 -> 254599
14:05:41.251 -> 381783
14:05:42.253 -> 509076
14:05:43.256 -> 636195
14:05:44.258 -> 763373
14:05:45.260 -> 890306
14:05:46.263 -> 1017213
14:05:47.265 -> 1143824
14:05:48.267 -> 1270424
14:05:49.323 -> 1396796
14:05:50.325 -> 1523260
14:05:51.328 -> 1649723
14:05:52.330 -> 1776185
14:05:53.333 -> 1902649
14:05:54.335 -> 2028983
14:05:55.337 -> 2155445
14:05:56.369 -> 2281779
14:05:57.365 -> 2408243
14:05:58.365 -> 2534575
14:05:59.385 -> 2661039
14:06:00.375 -> 2787373
14:06:01.393 -> 2913835
14:06:02.417 -> 3040169
14:06:03.418 -> 3166633
14:06:04.434 -> 3292967
14:06:05.455 -> 3419429
14:06:06.055 -> Motor position reached / rotation stopped
14:06:06.457 -> 3546160
14:06:07.462 -> 3673381
```

```
                             11);                                      //uint16_t: After this amount of mods has been reached, mod stops.

      if (ec < 0) { }                                                  //Optional error handler.
      once = 0;
      Serial.println ("From now you see increasing counter from main loop");
    }
    static uint32_t counts = 0;
    static uint32_t previous;
    uint32_t now = millis ();
    static uint32_t motorUpdateTime = 0;                               //This is example how to call the motor timer update not too frequent from main loop.
    if ((now - motorUpdateTime) >= 50UL)                               //If one does not want to take care of this, remove this code and uncomment above
    {                                                                  // | MOTOR_CTRL_IRQ_TIMING (so it gets included in the build).
      motorTimerUpdateCheck ();                                        //This shows how motor timer update check is now called by the user application.
      motorUpdateTime = now;                                           //When the motor position is reached, this function will result the endPosReached function
    }                                                                  //gets called (a subroutine only called once after the motor has been stopped).
    ++counts;
    if ((now - previous) > 1000UL)
    {
      previous = now;
      Serial.println (counts);
    }
  }
```