Exercise 3

In exercise 2, we got familiar with the SPI LCD, the built in flash memory, the built in ADC and we implemented and calibrated a simple voltmeter.

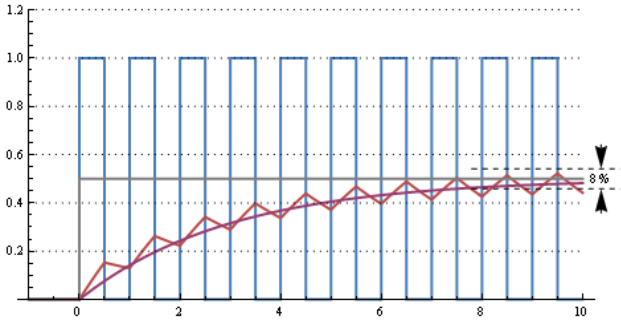
To become familiar with the pulse width modulation (PWM) functionality, you will develop a switch-mode power supply where the output voltage is monitored by the ADC. A feedback loop is to be implemented in order to keep the output voltage constant with a varying load.

Literature

1. STM32F302x6/x8 Datasheet (Table 15)
2. STM32F302x6/x8 Reference manual (Reference manual RM0365)
    a. PWM Mode (21.3.9-21.3.11 for TIM2/3/4, 22.4.10-22.4.11 for TIM15/16/17)
3. STM32 General purpose timer cookbook (Application note AN4776)
4. IR4426_27_28.pdf

Introduction to Pulse-Width Modulation (PWM)

The PWM signal is a pulse train with a variable pulse width and a fixed time period. The advantage of using PWM signals, as opposed to switching GPIOs directly, is that once the PWM/timer hardware is configured, it will continue independently of the microprocessor's other tasks.



PWM can be used to control the amount of power delivered to a load, without the losses that would result from linear power delivery by resistive means. This method is often used when dimming LEDs but it requires a sufficiently high PWM frequency to ensure that flicker is not perceived by the user.

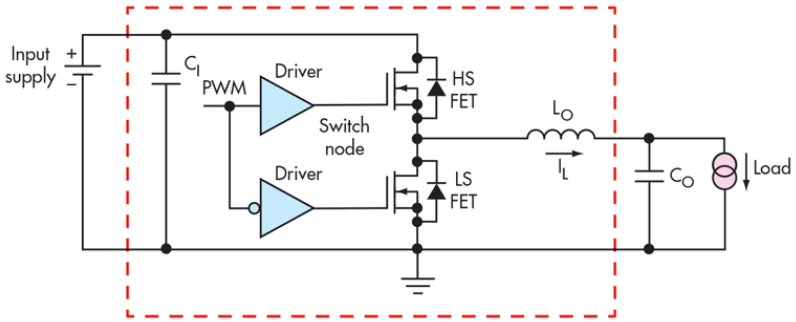
Power flow from the supply is not constant, however, and requires energy storage (eg. a capacitor) on the output side in most cases. Therefore the pulsed output is often smoothed with a low-pass filter, thus converting the pulsed signal into an analog voltage as seen in figure 1. By switching the output with the appropriate duty cycle, the analog voltage will settle at the desired level but with some ripple, depending on the low-pass filter. In this way, the PWM signal can be used as a simple digital-analog converter (DAC).

PWM signals are also used to regulate the output of switched-mode regulators. By switching voltage to the load with the appropriate duty cycle, the supply output will settle at a specific level. By measuring the output voltage and adjusting the duty cycle accordingly, it is possible to regulate the supply output.

Introduction to the assignment

In this exercise, you will build a simple switch-mode voltage regulator using the STM32 microprocessor, a driver circuit (IR4428), an inductor and a capacitor. The output voltage is measured with the STM32 ADC. Figure 2 illustrates how a synchronous buck regulator can be constructed. The two switches inside the IR4428 are switched with opposite polarity and will therefore never be closed at the same time:

1. When switch 1 (HS) is closed and switch 2 (LS) is open, current flows from the input supply through the coil. The current increases almost linearly with time.
2. After some time, the switches change state and switch 1 (HS) is now open while switch 2 (LS) is closed. The current through the coil will now run through switch 2 from ground through the coil. Now the current decays almost linearly with time.

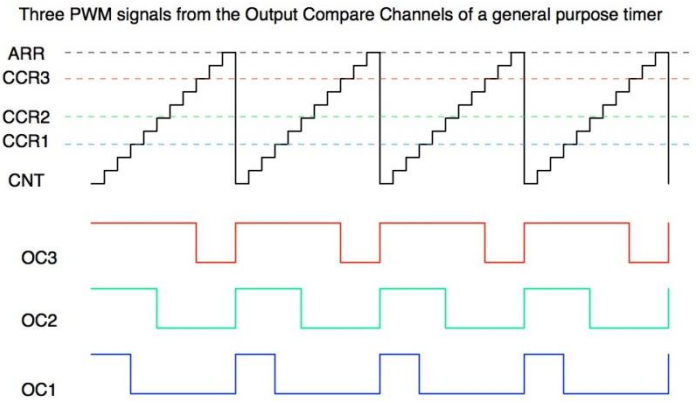


If the switching frequency is sufficiently high, and the capacitor is sufficiently large, the output voltage across the load will settle at a constant level. The average of this signal will be superimposed with a slight triangular signal with the same frequency as the switching frequency. The output voltage will drop if it is loaded with a resistor. To hold the output voltage constant; independent of the load, it is necessary to regulate the switching time, i.e. the duty cycle of the PWM signal applied at the driver input.

Exercise 3.1 – Generating a PWM signal

The STM32F302 microcontroller used in this course has a number of general-purpose timers that can be used to generate PWM signals. Up to 18 PWM channels are available. In this exercise, we will use TIM2 (4 channels), TIM15 (2 channels), TIM16 or TIM17 (both are 1 channel). First we must select a suitable GPIO pin and timer to use for the PWM output:

- Use the Alternate function tables (Table 14-18) in [DS] to find a suitable GPIO pin and timer channel to use for the PWM output. The following list shows the GPIO pins that can be mapped to a timer channel using the Alternate functions:
  TIM2; *PA2*, *PA3*, PB3, PB10, PB11 can be used
  TIM15; *PA2*, *PA3*, PB14, PB15 can be used
  TIM16; PA6, PA12, PB4, PB8 can be used
  TIM17; PB5, PB9 can be used
  Note: Some of these pins (indicated by underline) are connected to hardware on the mbed expansion board (see the table in the appendix) and some (indicated by *italic*) are connected to the UART. **Do not select those pins**
- Take note of the channel and the Alternate Function (AF) identifier for your selected timer (Use Table 14-18 [DS]). Example: For PA6 and TIM16, AF1 will correspond to TIM16_CH1.



The general-purpose timers have several modes that are used just for generating PWM signals. Each timer has at least one PWM channel and each channel can have a different duty cycle although they must use the same basic

frequency. Basic PWM mode is similar to the output compare toggle mode except that the PWM output pin is cleared whenever there is a match between the CCRx and the CNT register and then set again when the counter reloads. The default behavior is illustrated in the above figure.

PWM Frequency

The PWM frequency is an important parameter of the setup. When changing the brightness of an LED, any frequency above a few 10 of Hz will not be seen by the eye. Below this threshold and the light will seem to flicker. For motors and switch-mode power supplies, it is preferred to select a PWM frequency well above the range of normal hearing (>~22kHz). Otherwise, your circuit will make unpleasant noises!

PWM Resolution

Another important factor is the minimum resolution that you will need from the PWM system. That is, how many different steps should your duty cycle be divided in. Often 256 steps are used, corresponding to the range of values in an 8 bit integer.

The overall steps to setup the MC to produce a PWM signal is laid out here:

To use the Timer in Output Compare mode, the following steps are mandatory:

1. Enable TIM clock using *RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE)* function
2. Configure the TIM pins by configuring the corresponding GPIO pins
    a. Must be set in AF mode
3. Configure the Time base unit
4. Fill the TIM_OCInitStruct with the desired parameters including:
    a. The TIM Output Compare mode: TIM_OCMode
    b. TIM Output State: TIM_OutputState
5. Call *TIM_OCxInit(TIMx, &TIM_OCInitStruct)* to configure the desired channel with the corresponding configuration
6. In PWM mode, this function call is mandatory: *TIM_OCxPreloadConfig(TIMx, TIM_OCPreload_ENABLE);*
7. And to enable the PWM signal at last call these to functions: *TIM_CtrlPWMOutputs(TIMx, ENABLE);* and *TIM_Cmd(TIMx,ENABLE).*

Now the different parts will be discussed further.

Configuring the timer

When configuring the timer, there are a number of constraints to consider:

Timer frequency:	The input clock to the timer hardware
PWM Steps:	Number of steps used for the duty cycle
PWM Frequency:	The repetition rate for the PWM signal

These parameters define the values that must be configured in the TIMx_Period (reload value) and the TIMx_prescaler. First the counter frequency must be determined. We will use a PWM frequency of 10kHz and 256 steps for the duty cycle:

$$f_{CK} = f_{PWM} \cdot PWM\_STEPS = 10kHz \cdot 256 = 2.56\,MHz$$

Note that we need to multiply the PWM frequency by the number of steps, as it takes 256 cycles for the timer to wrap around. Now a prescaler must be selected to obtain f_{CK} from the base timer clock. Make sure that you use a prescaler value within the valid limits for the timer that you have selected.

$$\text{prescaler} = (f_{Timer}/f_{CK}) - 1$$

The TIMx_Period register will get a value corresponding to the number of PWM steps

$$\text{Period} = PWM\_STEPS - 1 = 254$$

- Write a function that initializes the timer 16 clock according to the above specifications (see Exercise 2.4). Timer interrupts are not needed here. You can call it `timerX_clock_init(..)`. Remember to enable peripheral clocks

Configuring PWM

For the next step, we will configure the counter compare registers of the timer. This is done by choosing mode, pulse, high/low polarity and so on.

The pulse length can also be set using this function *TIM_SetCompare1(TIMx,Pulse);* Pulse is the PWM_Steps.

For some timers (TIM15, TIM16 and TIM17), the TIMx_BDTR register must be configured as well to enable the CCx output. Here, the MOE bit must be set to enable "Main output". This is NOT required for TIM2!

This is done by calling *TIM_CtrlPWMOutputs(TIMx, ENABLE);*

Configuring GPIO

Finally, we must configure the selected GPIO pin in Alternate mode as it was done in exercise 2.6.

```
GPIO_StructInit(&GPIO_InitStructAll);
// Then set things that are not default.
GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructAll.GPIO_PuPd = GPIO_PuPd_DOWN;
GPIO_InitStructAll.GPIO_Speed = GPIO_Speed_50MHz;

GPIO_Init(GPIOx, &GPIO_InitStructAll);
GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex,GPIO_AF_x);
```

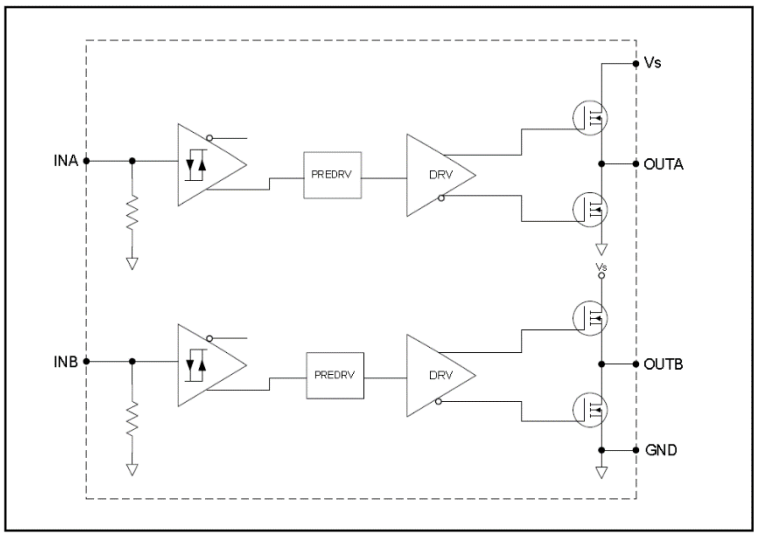
Hint: find which "GPIO_AF_x" you are looking for in [DS] table 14-18.

- Write a function that configures the counter compare registers according to the above example (PWM init) and a function that configures the selected GPIO pin in alternate mode. You can call them `timerX_pwm_init(..)` and `GPIO_set_AFx_PAy(..)`.
- Use the oscilloscope to verify the PWM signal frequency and duty cycle.
- Experiment with different values in the call to *TIM_SetCompare1(TIMx,Pulse);* and observe how the duty cycle changes.

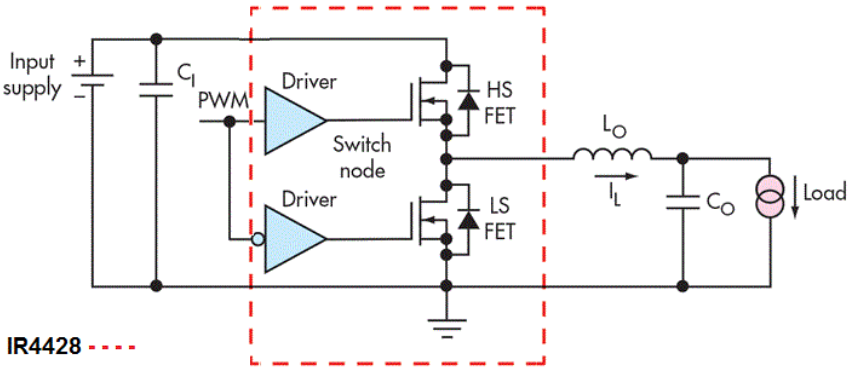
Exercise 3.2 - Implementing the circuit

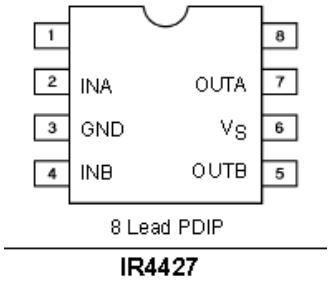
The driver IC IR4427, an inductor and a capacitor is used to build the synchronous buck converter. Only one half of the IR4427 is used. (If you were using the IR4428, you would only use INB, i.e. the non-inverting input). The adjustment of the output voltage is done using one of the PWM channels on the STM32. The output voltage of the buck converter will be measured by the built in ADC, as you did in the previous exercise. At this point, you must have a working PWM signal on either PA6 or PA12.

Functional Block Diagram IR4427



The picture (left) below illustrates how a buck converter can be constructed, using the before mentioned parts. The IR4427 is here illustrated by the dotted line. On the right the pin description of the IC.

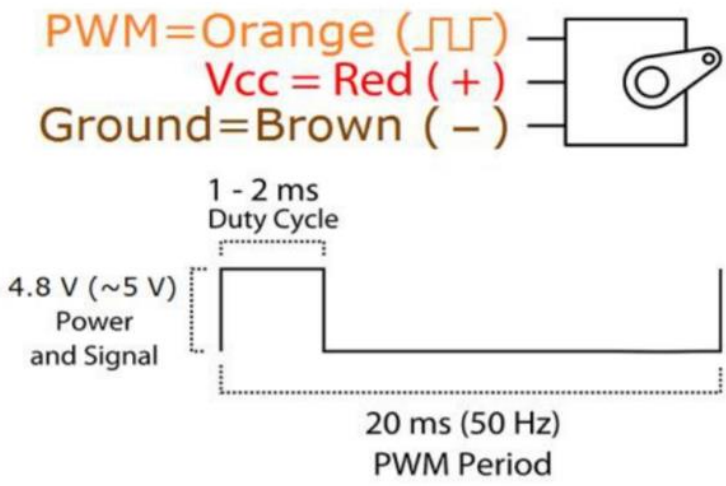




1. Construct the circuit on your breadboard. Connect the output side to an ADC of your choice (eg. PA0 or PA1)
2. Set the duty cycle to 50% and measure the resulting output voltage with the built in ADC. Show the voltage on the LCD display, as you did in the previous exercise. Repeat the test with a loaded output (fx. 100 ohm)
3. Add a software feedback to your program to ensure a stable output of 1.0 V. Use the ADC to determine the output voltage and adjust the duty cycle accordingly. Ensure that the output voltage is maintained at 1.0 V, even if the load is varied. As a load you can use a multi-turn potentiometer. Measure the output current with a multimeter. Examine how the duty cycle varies with the load and determine the maximum load current where an output voltage of 1.0 V can still be achieved. Write the duty cycle (%) on the LCD display and show the signals on the oscilloscope. Make a screenshot on the oscilloscope (do NOT take a picture with your smartphone!) and use it to document your results in the report.
   TIP: You can display the frequency, duty cycle and period on the oscilloscope using the “Measure->Add Measurement” menu.

Exercise 3.3 - Servo Motor

Now that you are familiar with PWM signals you are able to control servo motors using the microprocessor. Servo motors have three wires: Red = Vcc (5V), Brown (GND) and Orange = signal. A servo motor will move to a given position when receiving a PWM signal. The position is based on the duty cycle (see figure).



Write a program that takes the position of the two potentiometers on the Mbed expansion board and moves the two servos accordingly.

Exercise 3.4 - Stepper Motor

The microprocessor must be programmed to control a stepper motor. In order to be able to draw sufficient current a driver IC is inserted between the microprocessor and the stepper motor. The stepper motor must be controlled with commands from the keyboard that appears on the debugger monitor screen.

Literature

1. Stepper Motor Theory.pdf: General description of Stepper Motors.
2. UNL2003A.pdf: Datasheet for the UNL2003A driver IC.
3. Motor Technologies.pdf side A4-A12: General description of Stepper Motors.

Serial communication via the debug port

It is often convenient to be able to control the microprocessor from a keyboard. Communication are easy with the functions like uart_getc () (see App2).

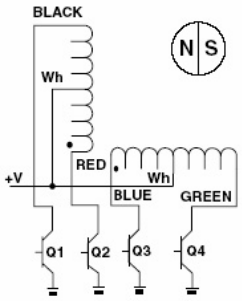
Write a simple program that lets the microprocessor receive a character from the PC's keyboard and return an answer to the debugger to monitor. If possible use switch() to distinguish between different characters from the keyboard, which is shown in this program fragment:

```
while( true )
{
  switch( uart_getc() )
  {
    case '1': putc( '1' ); break;
    .....
  }
}
```

The program will stop on the line switch(getc ()), while waiting for a character from the keyboard.

Introduction to stepper motors

A stepping motor consists of a rotor with permanent magnet and a stator with typically two or four electromagnets. When current flows through one or more of the electromagnets the rotor will rotate into a particular position and remain there. By alternating the transmit power through some of the electromagnets and disconnecting power to the other it is possible to make the rotor stepwise rotate in one or the other direction. Figure 1 shows the principle of a unipolar stepper motor. The stator consists of the two sets of coils each with a central tap (Wh), which is connected to a positive supply voltage (+V). The four coil ends can be connected to ground via switches (Q1-Q4). Either Q1 or Q2 are switched to ground, and either Q3 and Q4 are switched to ground. The switches Q1, Q3 and Q2, Q4 are 90 degrees out of phase (in quadrature).



CW Rotation ↓

Unipolar Step	Q1	Q2	Q3	Q4
1	ON	OFF	ON	OFF
2	OFF	ON	ON	OFF
3	OFF	ON	OFF	ON
4	ON	OFF	OFF	ON
1	ON	OFF	ON	OFF

↑ CCW Rotation

Figure 1: Unipolar stepper motor. The colors of the wiring on the specific stepper motor may be different from the figure.

Figure 2 shows how the rotor rotates 90 degrees after the power is changed in the one pair of coils and kept unchanged in the other pair.

In the exercise the four switches Q1-Q4 are implemented with the component ULN2003A, see Figure 3. This circuit includes seven grounding switches, each of which may draw a current of up to 500 mA. A high value on an input (pin 1-7) will short the output to ground. Each of the four coil ends are connected to an output circuit (pins 10-16). The middle outlets of the coils (Wh) and pin 9 are connected to the engine voltage supply. This power supply can be the 5V from ICD-U40 or a separate voltage supply. When the current through one of the coils is interrupted, a counter electromotive force is generated this is limited by the diode. In order to reduce the transients in the 5V supply, which could destroy or stop the microprocessor, it is important to connect the two 100nF capacitors across the V_{DD} and V_{SS} as in the previous exercises. Pin 8 is connected to GND.

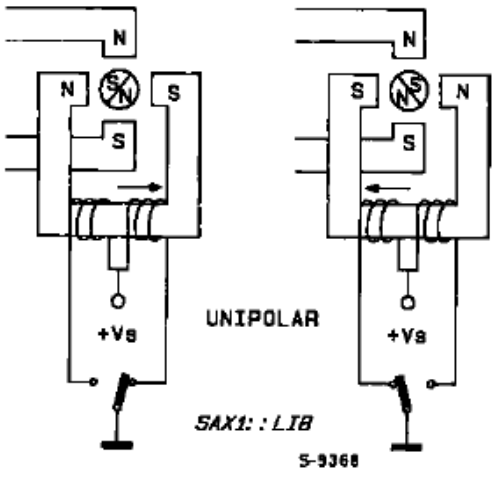


Figure 2: Unipolar stepper motor.

Exercise 3.5 – Implementing the Stepper Motor

Build a stepper motor driver with the µP and a ULN2003A. Figure 3a shows how one of the four coils must be connected. Remember to connect pin 9 to Vcc. Use one of the ports to control ULN2003A.

For each step/position, two coils may be turned on. The four positions can be specified in an array with the following: POSITIONS[4] = { 0b0101, 0b1001, 0b1010, 0b0110 };

The motor is set in rotation by alternately sending these bytes to port C. This can be done in a loop this way: POSITIONS[i&3];

Index for POSITIONS is maintained within the range [0, 3] by &(AND) 3, where the variable i may be of any size. It is necessary to set the four bits the port as output.

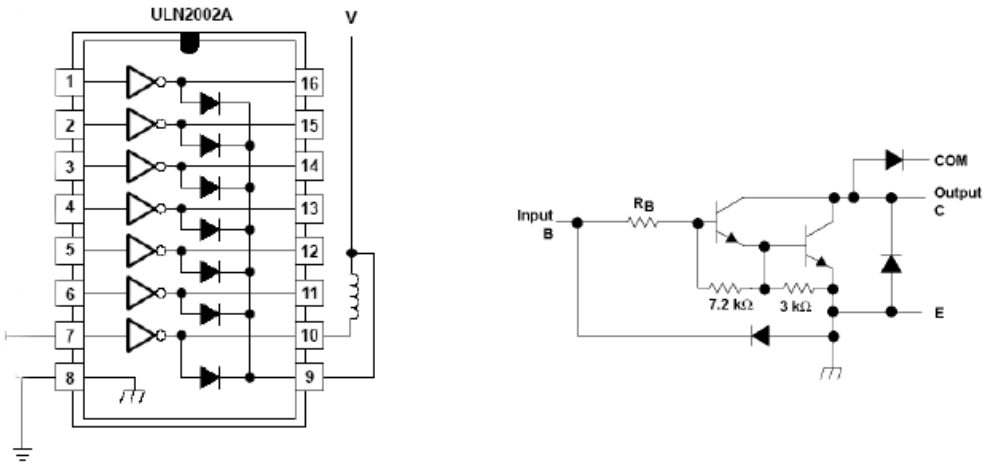


Figure 3: Darlington transistor Array.

In the exercise we use a 12V unipolar hybrid (multi-pole) stepper motor with typically 200 steps/revolution. (See Fig 4 for a picture of the stepper motor function) If the motor used in the exercise is supplied by 5V, it can only run with reduced power.
Identify the two central taps (Wh in Figure1) with an ohmmeter. It will likely be necessary to swap the other wires around to make the motor run.

The USB port cannot provide sufficient , therefore, a separate voltage supply for driving the stepping motor may be used. The voltage can be set to 12V. With a command (a single character) from the keyboard, the stepper motor should start and make one revolution typically 200 steps. With two other commands the delay between two steps could be adjusted up and down in 0.1 millisecond interval. With a delay function a delay can be defined with a microsecond resolution. Initialize the delay to 5ms. Print the current delay on the monitor screen. Determine the least delay that still causes the motor to run a full one revolution (no lost step).

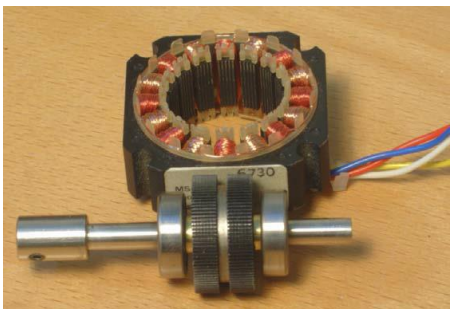


Figure 4.- Hybrid stepper motor with 200 steps revolution. The rotor includes a magnetic disk and two multi- toothed coils, each with 100 teeth. The teeth on the two poles are 180 ° out of phase

Report:

You will write a report for this exercise. The results should be demonstrated for the teacher.

Appendix 1 – mbed Application Shield Pinout

mbed Application Shield Pinout		Arduino Pin	STM32 Pin
128x32 Graphics LCD, SPI Interface	Newhaven C12332A1Z		
	MOSI	D11	PB_15
	nRESET	D12	PB_14
	SCK	D13	PB_13
	A0	D7	PA_8
	nCS	D10	PB_6
3 Axis +/1 1.5g Accelerometer,I2C Interface	Freescale MMA7660, Address:0x98		
	SCL	SCL	PB_8
	SDA	SDA	PB_9
Temperature sensor	LM75B, Address:0x90		
	SCL	SCL	PB_8
	SDA	SDA	PB_9
5 way Joystick	ALPS SKRHADE010 (pulldown)		
	Down	A3	PB_0
	Left:	A4	PC_1
	Centre:	D4	PB_5
	Up:	A2	PA_4
	Right:	A5	PC_0
2 x Potentiometers	Iskra PNZ10ZA, 10k		
	Pot 1 (left)	A0	PA_0
	Pot 2 (right):	A1	PA_1
RGB LED, PWM connected	Cree Inc CLV1A-FKB		
	Red:	D5	PB_4
	Green:	D9	PC_7
	Blue:	D8	PA_9
Speaker, PWM Connected	MULTICOMP MCSMT-8030B-3717		
	D6	D6	PB_10
Xbee socket (unused)	Rx:	D1	PA_2
	Tx:	D0	PA_3
	nReset:	D3	PB_3
	Status:	D2	PA_10

Appendix 2 – Serial Read

Controlling the stopwatch using the joystick is pretty cool, but not very intuitive. The next step is therefore to control it from the terminal!

To read from the terminal you can use the command `uart_getc()`. This function waits for an input byte from the UART and then returns it. Note that it will wait forever if nothing is written to the UART!

- Create a function that uses `uart_getc()` to read a specified number of characters into a byte array and then returns the array. The last element of the array should always equal 0x00, regardless of what is written to the UART.

If you run this function, you should see characters appear in PuTTY when you strike keys on the keyboard. After the specified number of characters have been written things should return to normal. If you try printing the array using `printf("%s",array)` you should see the text you wrote being re-written to the terminal. This works because a byte array ending with a zero is treated as a string by C.

To avoid being limited by a fixed number of characters we should update the function.

- Update the function to return if the UART receives a carriage return character (0x0D), i.e., when the 'enter'-key is pressed. Note: you should also set the corresponding element in the array to 0x00.

Now the function will return early if the user presses enter.

At this stage we have the ability to send text to and from the PC. Next, we need to interpret the text so it can be used for user input. An easy way to do this is by using `strcmp()`. This function takes two strings and returns 0 if and only if the two strings are identical (ignoring case differences).

- Create a function that returns a different number depending on whether the user inputs "start", "stop", "split1", "split2", "reset", or "help". You may add other keywords if you like.
- Use the function that you've just created to control your stopwatch. Note: Because the `uart_getc()` function waits indefinitely for an input the screen will not update except when the user presses enter.

Now you have your very own keyboard controlled stopwatch! However, if one of your friends who haven't taken this course wants to use it, they'll have no idea how it works (unless you tell them, of course).

- Create a function that prints out a simple user guide to PuTTY.
- You should call this function at the start of your program and whenever the user inputs "help".
- Create another function that prints out an abbreviated list of commands and call it every time the user inputs an un-recognized command.

There you go: a terminal based stopwatch that anybody will be able to use without introduction!

Report

Write a report of the exercise. The report should be about 5 pages long including a description, a chart, a table of used ports, documentation of results (eg. screendumps) and a well commended featured source code including header file.

Participants on a team draw up a single report with a cover, which show participants names, student number and group number. It should not be specified in the text, who wrote what. The report is uploaded preferably as PDF file in the course Assignment. The team will have an annotated report back via e -mail.

The report should typically include most of these standard points:

Title

Abstract Subject , methods, results.

Introduction general description, no details.

Problem definition What is it about.

problem limitation What you cannot with this method or this hardware.

Theory How do the various components and methods.

Hardware Diagram and table of used ports.

Software Function Description (no source code here).

Test method/software What worked, what did not and why.

Conclusion Must correspond to the problem formulation.

Appendix Source code, diagrams, possibly. Print drawings,

Data Sheets (only important pages), references and links.