Timers and PWM

PWM generation is one of the main methods of producing *analog* output. We will examine them in detail and try to produce PWM signals using **timers**.

2.1 PWM signals

In previous lessons, we introduced how to produce output on LED lights. This signal is limited to two levels only: high (5V) and low (0V). In practice, we often need to produce output *somewhere in the middle*.

PWM (pulse width modulation) is a method of producing intermediate output. Imagine if we program the following:

1. Turn on LED for 1.2 seconds
2. Turn off LED for 0.8 seconds
3. Repeat 1 and 2

To our eyes, the LED would be blinking at a certain pattern. Since turning on the LED gives an output of 5V, on average the output of the mainboard pin would be:

$$\frac{1.2}{1.2+0.8} \times 5 = 3 \text{ V}$$

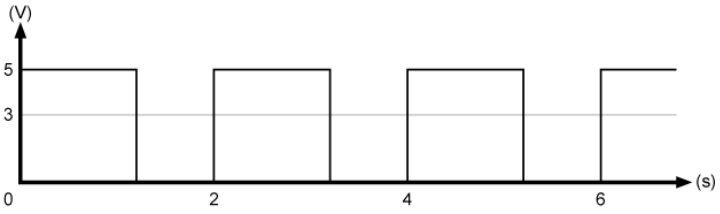

Figure 1: An on-off output pattern

Suppose we then change the on and off time to **12 ms** and **8 ms** respectively. The average output is still 3V, as the ratio of high to low time is unchanged. However, the on and off action is now so fast, the LED would appear to be turned on but at a dimmer level. We have now achieved an output in between 5V (bright light) and 0V (no light).

Pulse width modulation (PWM) is the act of alternating between high and low output at a specified high frequency, in order to simulate an intermediate level of output.

A PWM signal is usually given by its **frequency** and **duty cycle**. The frequency refers to the no. of **periods** in one second, where one period refer to the duration of both on time and off time. For example, the above signal has a frequency of

$$\frac{1}{0.012 + 0.008} = 50 \text{ Hz}$$

whereas the duty cycle refers to the ratio of on time in each period, often expressed as a percentage:

$$\frac{1.2}{1.2 + 0.8} \times 100\% = 60\%$$

The duty cycle is essential as it determines the actual level of output. In comparison, the frequency of a PWM signal is theoretically unimportant as long as it is high enough.

However, for certain applications, the frequency needs to be at a certain value. This is the case for **servos**, a motor with a rotary arm whose position is determined by the duty cycle of PWM signal. Servos will be used in the upcoming Robot Design Contest for car steering. We will be experimenting with servos to demonstrate the effect of PWM signals.

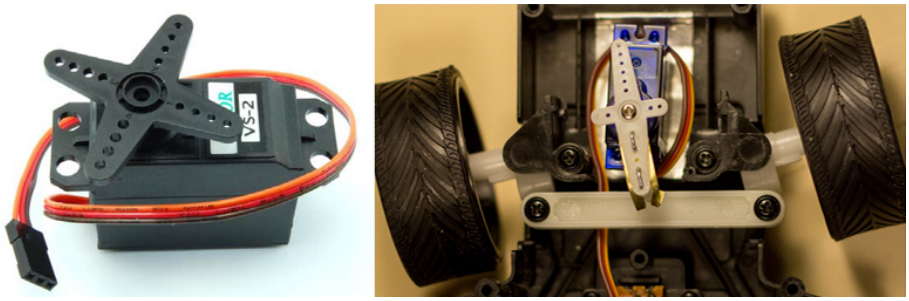

Figure 2: Servos used in car steering

2.2 Timers in STM32

In order to produce PWM signals, we use **timers**. Together with GPIO, timers are one of the many **peripherals** included in the STM32 microcontroller. A timer is essentially a counter that counts up from 0.

A timer is denoted by its id `TIMn`, where `n` is the timer ID. The mainboard has a *medium-density* STM32F103RB MCU, thus has 4 timers labelled `TIM1` to `TIM4`. In actual competitions, we also use *high-density* MCUs with more functionalities, which has 8 timers (up to `TIM8`) instead.

The counting ability of timers comes from the **system clock** (SYSCLK). Like all processing units, the STM32 MCU has a clock - a high-low oscillating signal - so that the MCU can perform calculations based on the regular pulses of the clock. The actions of different peripherals, including each count of a timer, are coordinated by the clock signal.

Timer setup

The F103RB MCU, together with all processors of the STM32F1 family, has a *clock rate* of 72 MHz. This corresponds to 72 million pulses per second.

By default, the timer will count 72 million times per second as well. This is certainly too fast for most cases. We use a **prescaler value** to slow down the rate of counting. The equation for calculating the prescaler is as follows:

$$p = \frac{c}{f_t} - 1$$

Where $c = 7.2 \times 10^7$ is the clock rate; f_t is the slower frequency of timer counts; and p is the prescaler value. For example, if we would like the timer counter to increase 200,000 times every second, our prescaler value would be:

$$\begin{aligned} p &= \frac{7.2 \times 10^7}{2.0 \times 10^5} - 1 \\ &= 359 \end{aligned}$$

The default value for prescaler is 0. By substituting 0 into p above, we can see that a prescaler of zero gives a prescaled frequency of 72 MHz, i.e. no scaling down is performed.

Now, as the counter increases, there is bound to be a upper limit so that the counter starts from zero again ("auto-reload") after reaching the max value. This limit is known as the **auto-reload value**. For PWM generation, the timer should auto-reload at the end of every period.

In other words, the auto-reload value should be the *no. of counts per PWM period*. The formula for auto-reload value is

$$a = \frac{f_t}{f_p}$$

Where f_t is, again, the prescaled timer frequency (no. of counts per second), f_p is the desired PWM frequency (no. of periods per second), and a is the auto-reload value (no. of counts per period). For example, to achieve a 40 Hz PWM signal using the prescaler above, the correct auto-reload value is

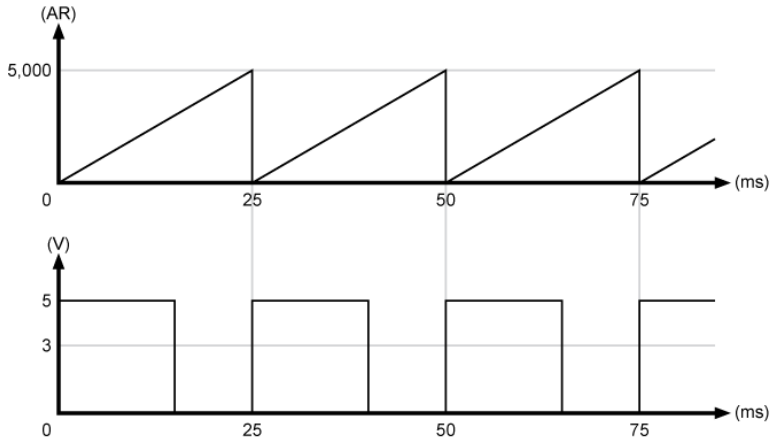

Figure 3: Setting the auto-reload value

$$\begin{aligned} a &= \frac{2.0 \times 10^5}{40} \\ &= 5000 \end{aligned}$$

The timer auto-reloads before reaching the auto-reload value. In the above example, the highest value reached by the timer is 4999, and in the next count it is reset to 0. A total of 5000 numbers, from 0 to 4999, can be represented by the timer counter.

Task 1 (20 pts)

Suppose we are going to set up a timer for controlling a servo. All PWM signals feeding into the servo should have a frequency of 50 Hz.

a) We would like to let the timer count at a frequency of 500 KHz. What is the correct prescaler? (5 pts)

b) What is the correct auto-reload value? (5 pts)

c) What is the slowest timer frequency that can be generated by a prescaler? (Hint: the prescaler value is a `u16` value.) (10 pts)

2.3 Timer PWM generation

Right now, the timer counts according to the PWM frequency, but there is no actual high / low output. The **PWM mode** is one of many operation modes of the STM32 timer, which generates PWM using a **output compare** (OC) value, in the following fashion:

1. A GPIO pin is set to output, just as the same way as the LED pins.
2. When the timer counter is smaller than the OC value, the pin will be in high state (5V);
3. When the timer counter is greater or equal than the OC value, the pin will be in low state (0V).

In this manner, a PWM signal can be generated, by comparing with the OC value.

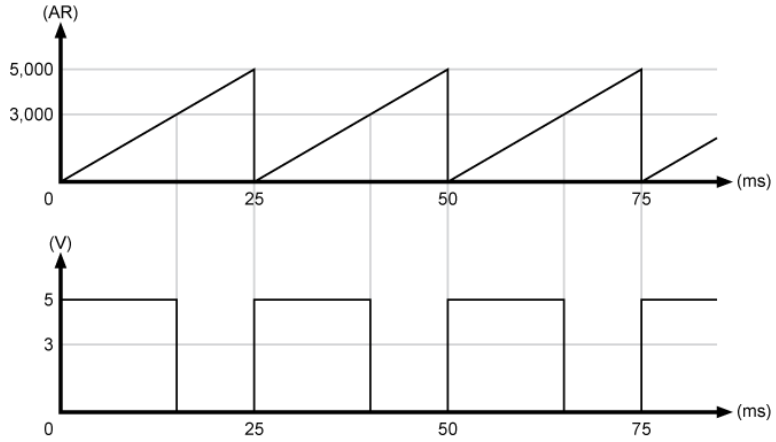

Figure 4: Determining duty cycle by OC value

The duty cycle of PWM signal is adjusted by changing the OC value. When OC value is 0, there will be no output at all; when the OC value is equal (or greater) than the auto-reload value, the counter will always be smaller and there will be full output.

To help you get started, our mainboard library provides the `servo.c` library for interfacing with servo PWM, using the `TIM3` timer. Each timer has 4 OC **channels**, named `OC1` to `OC4`. Our mainboard has two servo ports, `SERVO1` and `SERVO2`, which corresponds to the `OC1` and `OC4` channels of `TIM3` respectively. Since each OC channel has a separate OC value, the two servos can be controlled independently.

The `servo_init()` function initializes the timer and both servo OC channels, with the following parameters:

- `presc` - Prescaler
- `count` - Auto-reload value
- `init_pos` - Initial OC value

```
void servo_init(u16 presc, u16 count, u16 init_pos);
```

On the other hand, the `servo_control()` function controls the duty cycle of the PWM signal by changing the OC value:

- `id` - Servo to be changed (`SERVO1` or `SERVO2`)
- `pos` - new OC value (i.e. new servo position)

```
void servo_control(SERVO_ID id, u16 pos);
```

There are two servo ports on the mainboard. Each servo port have 3 pins, shown below in the *schematic diagram*. The *leftmost* pin is connected to the timer GPIO output pin, hence provides the PWM signal; the *middle* and *rightmost* pins connects the VCC (5V) and ground (0V) from the mainboard to the servo, so that the servo is powered and can move. Follow the diagram below to correctly hook up the servo and mainboard.

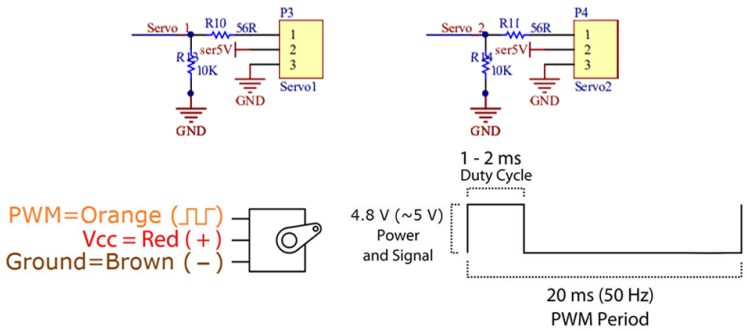

Figure 5: Servo PWM and schematic of servo ports

For the servo we are using:

- The leftmost servo position corresponds to an on-time of **0.9 ms**
- The rightmost servo position corresponds to an on-time of **2.1 ms**

Notice that this is only a small range out of the total period, which should be *20 ms* (50 Hz). Please do **NOT** program the mainboard to output any PWM signal outside this range, as it may damage the servo.

Task 2 (15 pts)

Initialize the servo using `servo_init()`, using the prescaler and auto-reload value you have calculated in Task 1. Set the servo so that it points at the middle position, i.e. set an initial OC value corresponding to an on-time of **1.5 ms**.

Tutors will first verify correctness of program output using an **oscilloscope**. If your code correctly displays the PWM signal on the oscilloscope, you may test it with a servo motor. You can try to twist the arm to some other position first, and see if it will move to the middle position upon pressing the `RESET` button.

Task 3 (30 pts)

By using prior knowledge in Tutorial 1 and the `servo_control()` function, create a program that:

- Initializes the servo to the leftmost position
- Change the servo position every 50 ms, such that the servo arm moves to rightmost position in *exactly 1 second*
- Similarly, move the arm back to leftmost position within 1 second
- Repeat the above two steps

2.4 Motor control

A major application of PWM control is to drive motors, which provide the main driving force for the robot. Controlling motors, however, is slightly more complex than rotating servos.

Motor driver

Most servos, including the ones used in class, can operate under a power source of 5V. Motors, on the other hand, often needs to be powered by higher voltages, such as from Li-Po (lithium polymer) batteries at 12V or 24V.

However, our mainboard runs on 5V only. A simple way of controlling the motor using a 5V PWM signal is by using a **transistor**. A transistor acts as an electric "switch" across its three terminals; when a lower voltage is provided to the gate terminal, the transistor allows current through the other two terminals, and vice versa.

We can feed in a 5V PWM signal to control whether current will flow from 12V battery through the motor. For example, if we feed a 75% duty cycle PWM signal, the average voltage over the motor will also be 75% of the power source (i.e. 9V). Hence, the motor turns at 75% of its fastest speed. This PWM signal is now controls the *speed* of the motor, as opposed to controlling the arm position of

a servo. Unlike servo PWM signals, which is restricted to a small range (e.g. 0.9 - 2.1 ms), the duty cycle of motor PWM signals can go from 0% to 100%.

The fastest speed is termed **full speed** and is the speed obtained when 12V is constantly running through the motor (100% duty cycle). However, the actual value of full speed is not constant; it determines on how much force is being exerted on the motor. For example, compared with a motor spinning in air without any wheel attached to it, the full speed of a motor on a moving robot vehicle would be drastically lower. This is because the *normal force* of the vehicle, dependent on the robot weight, is acting on the motor axis.

The PWM signal allows the robot to moves at some speed, although it cannot change the direction of motor rotation. To change the direction, we would need to let current flow through the motor in opposite direction. This is achievable through an **H-bridge circuit**, consisting of 4 transistors:

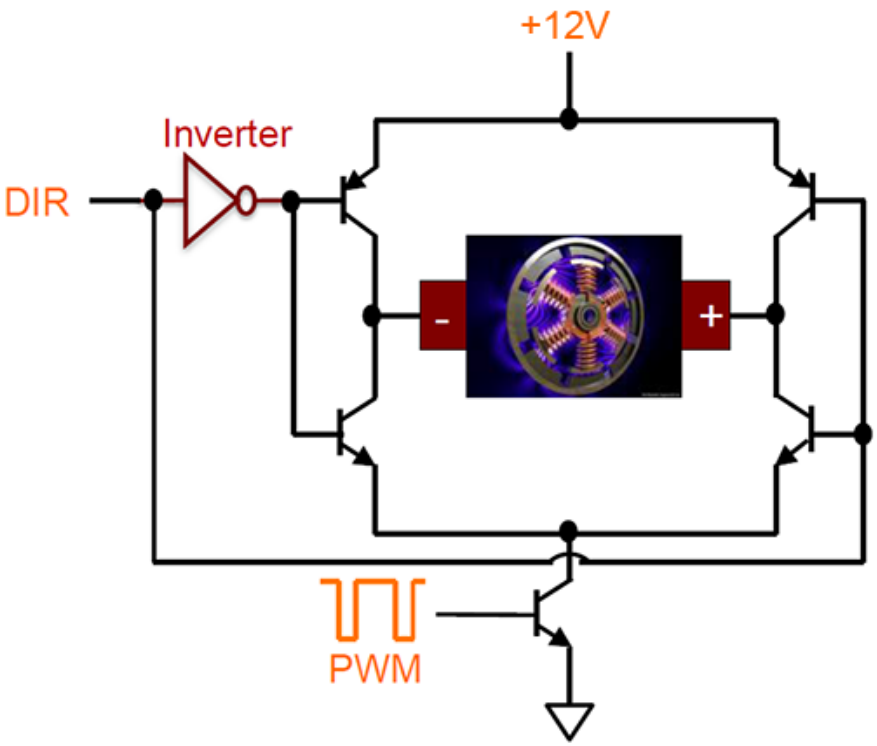

Figure 6: Motor driver circuit

Notice a second input termed `DIR` - the **direction pin**, which controls the H-bridge transistors. A 5V output to this pin turns the motor in one direction, whereas a 0V output turns it in the other direction. The whole PWM motor

driving circuit is known as a **motor driver**. In the Robot Design Contest, you will be controlling the motor with drivers soldered by teammates from the hardware tutorial classes.

Motor PWM control

The mainboard library provides another file, `motor.c`, for controlling 3 motor ports, using the `TIM1` timer. The library is more or less similar to `servo.c`. Each motor ports have 4 pins, the middle two being `DIR` and `PWM` output; whereas the VCC and ground pins give power to the motor driver circuit. The order of pins, once again, can be found in the mainboard schematic (partially shown below).

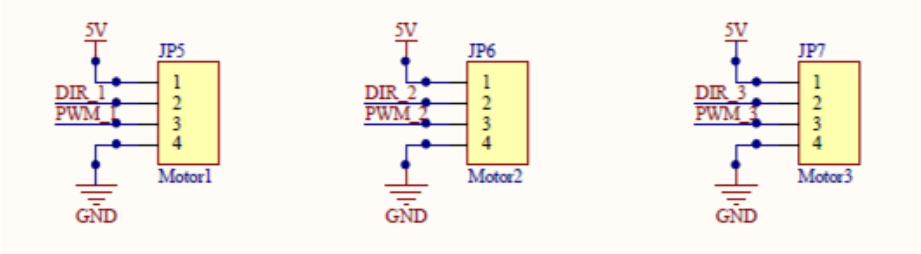

Figure 7: Schematic of motor ports

The `motor_init()` function initializes the timer, in a similar manner to `servo_init()`:

```
void motor_init(u16 presc, u16 count, u16 init_val);
```

On the other hand, the `motor_control()` function controls motor direction and speed at the same time.

- `id` - ID of motor port (`MOTOR1`, `MOTOR2` or `MOTOR3`)
- `dir` - Motor direction, either 0 or 1
- `magnitude` - OC value, specifying the motor speed

```
void motor_control(MOTOR_ID motor_id, u8 dir, u16 magnitude);
```

Which direction (clockwise or anticlockwise) does 0 or 1 represent is entirely dependent on how the motor driver is designed. The simplest way to find out is by trial and error. Since neither motors nor drivers will be provided during tutorial sessions, the actual direction of rotation is not important for classwork.

Task 4 (35 pts)

Create a program that:

- Sets the prescaled timer frequency of `TIM1` to 1 MHz

- Sets `TIM1` such that the output PWM has a frequency of 5 KHz
- Initially, there should be no PWM signal (duty cycle = 0%)
- Each time `BUTTON1` is pressed, increase duty cycle by 25%
- If full speed is reached, press `BUTTON1` once more to reset to 0%
- Change direction each time `BUTTON3` is pressed

Homework

Homework will be distributed via email to individual classes.

Please visit the Robotics Team lab (CYT3007B, Lifts 35-37 3/F) to show your work to tutors. Mainboards for testing and oscilloscope will be provided there during daytime *except when there is another software tutorial ongoing* (2-5PM Thu / Sat, 7-10PM Tue / Wed / Fri).

HKUST Robotics Team
Software Tutorial
Notes by Hong Wing PANG (hwpang@connect.ust.hk)