# Servo Motor Lab Report

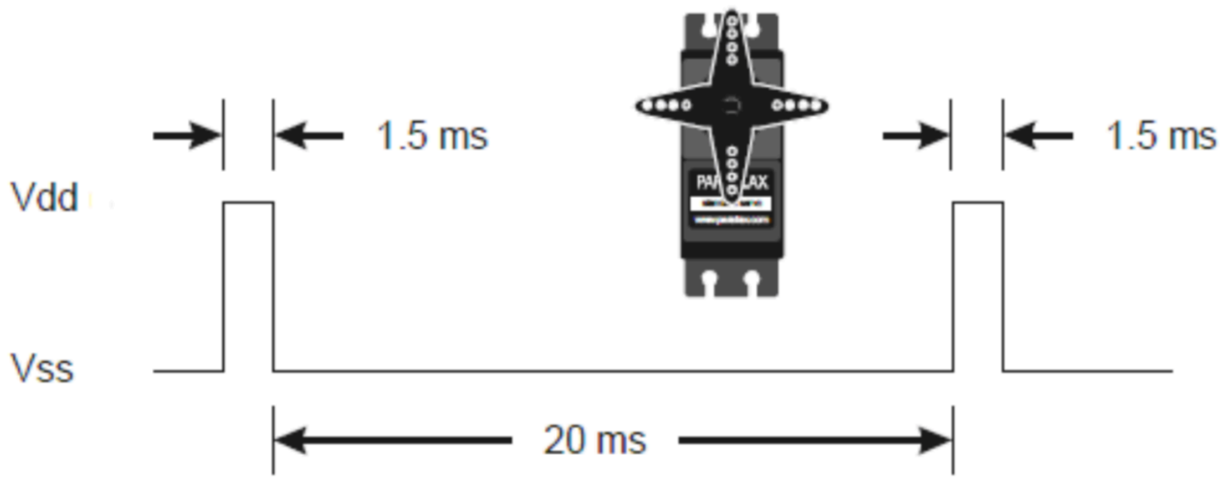
## Servo Motor Lab

In this lab, we will make two kinds of servo motors move with Pulse Width Modulation(PWM) signals. We will find out about the signals we will use and the built in clock and timers in the embedded processor to generate specific signals that can make a standard servo and a continuous servo motor work. To go beyond just making the servo's simply turn, we will also use some code to control the servo motors from the LCD screen.

### Servo specifications

To use the servo, we have to understand it first. The servo needs 3 connections, a ground, a power, and a signal line. The servos need 5 volts for the power, and the specification sheet recommends 5V for the signal too.

The continuious servo is governed by PWM signals, and the duration of the on pulses are the key to actioning the servos.



The continuous servo has the the time range of 1.3ms to 1.7ms for the duration of the pulses for continuously turning left and right.

| BASIC Stamp Model | 1.3 ms (Full speed clockwise) | 1.5 ms (Center, no rotation) | 1.7 ms (Full speed counterclockwise) |
|---|---|---|---|

The continuous servo also has an adjustment screw on it, where we can reset the zero to the right place on the motor.

The standard servo only turns to a certain degree, and doesn't do full rotations. It has no adjustment options. The PWM signals are similar to the continuous servo, but the time range is different.

| BASIC Stamp Module | 0.75 ms | 1.5 ms (center) | 2.25 ms |
|---|---|---|---|

It is 0.75ms for the farthest left position, and 2.25ms for the farthest right position, and any angle in between corrosponding to within the 0.75ms ~ 2.25ms time range for the PWM pulses.

## Timers

There are quite a few timers on our STM32 Discovery board, and many of them have advanced features. We should choose a timer that has the minimum amount of features we need, and from that, see if it has the board support for the small set of functions we need. The timers are shown below:

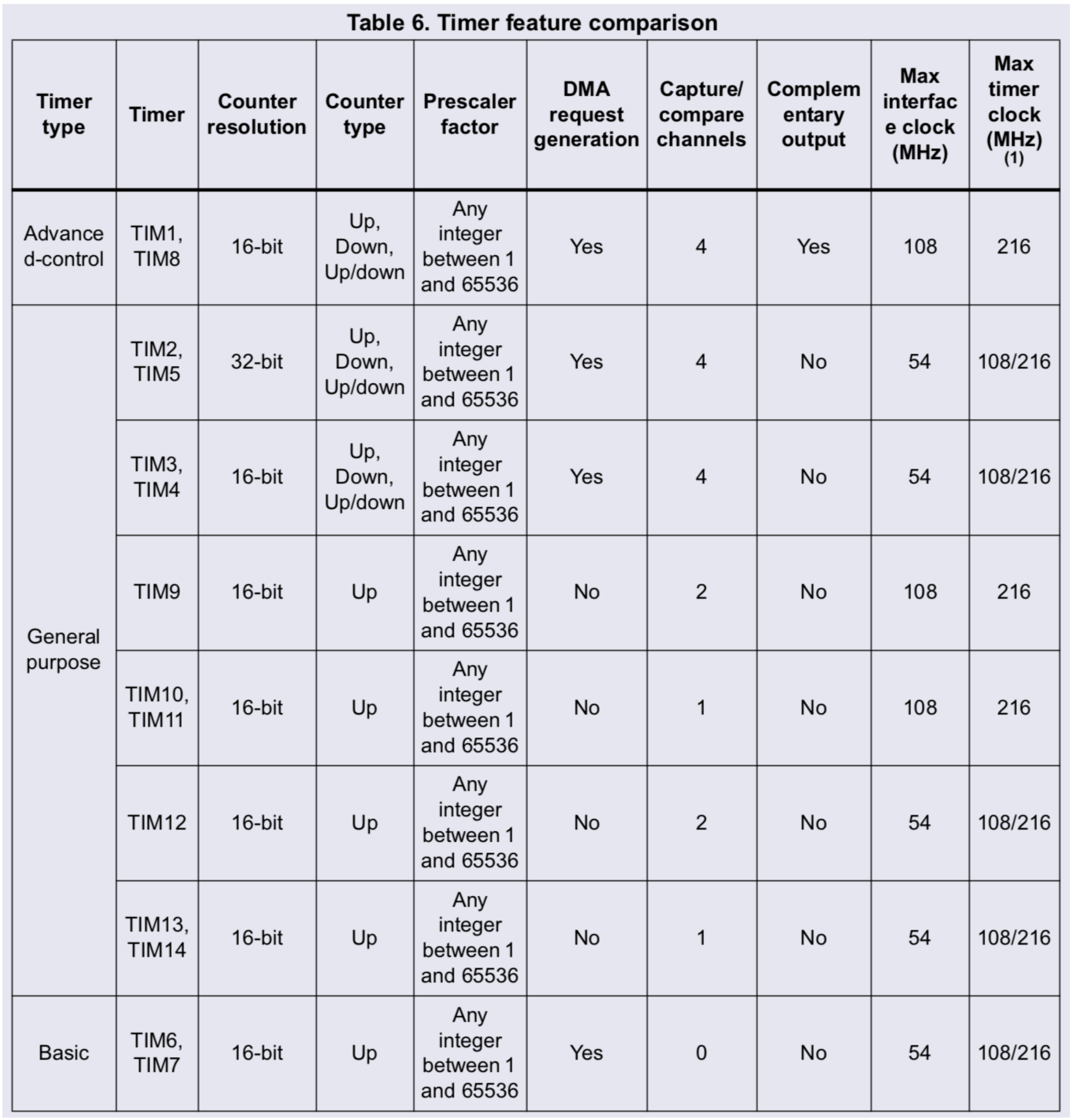
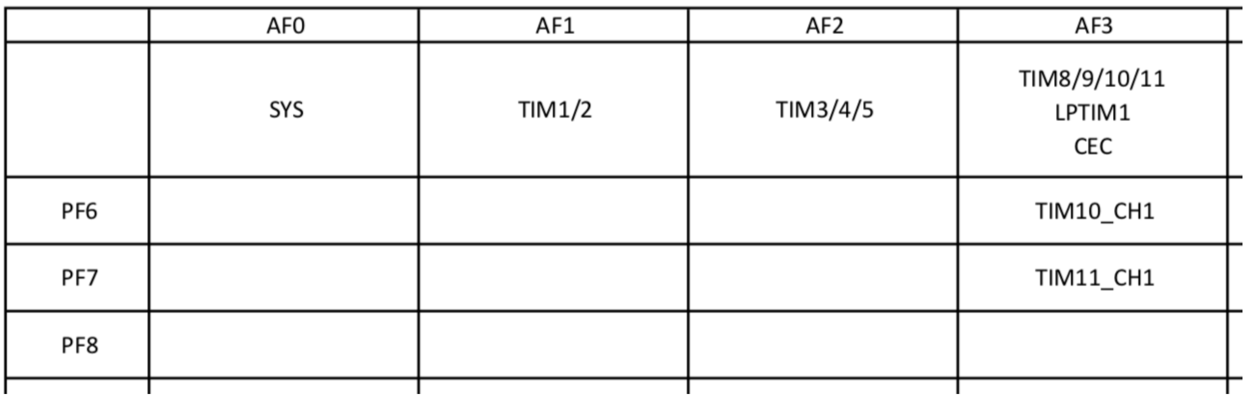


**Table 6. Timer feature comparison**

| Timer type | Timer | Counter resolution | Counter type | Prescaler factor | DMA request generation | Capture/ compare channels | Complementary output | Max interface clock (MHz) | Max timer clock (MHz) (1) |
|---|---|---|---|---|---|---|---|---|---|
| Advanced-control | TIM1, TIM8 | 16-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | Yes | 108 | 216 |
| General purpose | TIM2, TIM5 | 32-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | No | 54 | 108/216 |
|  | TIM3, TIM4 | 16-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | No | 54 | 108/216 |
|  | TIM9 | 16-bit | Up | Any integer between 1 and 65536 | No | 2 | No | 108 | 216 |
|  | TIM10, TIM11 | 16-bit | Up | Any integer between 1 and 65536 | No | 1 | No | 108 | 216 |
|  | TIM12 | 16-bit | Up | Any integer between 1 and 65536 | No | 2 | No | 54 | 108/216 |
|  | TIM13, TIM14 | 16-bit | Up | Any integer between 1 and 65536 | No | 1 | No | 54 | 108/216 |
| Basic | TIM6, TIM7 | 16-bit | Up | Any integer between 1 and 65536 | Yes | 0 | No | 54 | 108/216 |

Since we are working two different servo motors, we need two timers because of the different timing we need for each, but we only need one channel from each timer. The best timers that fit the description is Timer 10 and 11. Timer 10 and 11 are also perfect for our needs, because according to the alternate function chart the timers are on the same AF3, the pins are on the same GPIO port, they are right next to each other, and they are also physically in sequence.



| | AF0 | AF1 | AF2 | AF3 |
|---|---|---|---|---|
| | SYS | TIM1/2 | TIM3/4/5 | TIM8/9/10/11 LPTIM1 CEC |
| PF6 | | | | TIM10_CH1 |
| PF7 | | | | TIM11_CH1 |
| PF8 | | | | |

The timers have a prescaler(PSC) and an automatic reload register (ARR) that determines how much the counter counts and outputs a signal. Using the PWM modes on the timer, the capture compare register(CCR) determines how long the pulse should be on or off when the counter is counting up to ARR.

## Using Pulse Width Modulation

To ensure the correct timing, we need know the existing clock freequency for our timer. From the timer feature chart, we can see that the freequency for our timers 10 and 11 is 216Mhz. Normally the period for square waves has a factor of two in it

$$f_{square} = \frac{f_{clk}}{2(PSC+1)(ARR+1)}$$

But we are using the PWM mode that allows us to use CCR as the count to determine where "half" of the period is, and thus our equation looses a factor of 2:

$$f_{PWM} = \frac{f_{clk}}{(PSC+1)(ARR+1)}$$

## Equations to action servo

When we want to reproduce a specifically timed length of pulses, we can solve the equation equation for the PWM version of the equation by setting ARR+1 to the maximum value. Our motor needs 20ms in between each pulse, and the pulses themselves are 1.3ms.

$$\frac{1}{(20+1.3)ms} = \frac{216Mhz}{(PSC+1)\cdot 65535}$$

$PSC+1$ comes out to be $\approx 70.2$, which we round up. Then we place the value back in to find ARR+1.

$$\frac{1}{(20+1.3)ms} = \frac{216Mhz}{71\cdot(ARR+1)}$$

$ARR+1$ comes out to exactally $64800$.
The reasoning behind setting PSC to the smallest value is that we can create the most accurate rounding of timeing into the nanosecond range with a small PSC.

We can find CCR by solving

$$CCR = 64800\,\frac{1.3}{21.3}$$

Because we know CCR is the amount of time where the PWM is outputting a 1.
We get $CCR \approx 3954.93$ which we round to the closest integer.

Putting all of this together, we can make the continuous servo turn left.

```c
#include "stm32f7xx.h"                  // Device header

int main() {
    RCC->AHB1ENR |= 1 << 5; // enable GPIO F
    RCC->APB2ENR |= 1 << 18; // enable TIM11

    GPIOF->MODER |= 2 << 14; // AF mode on PF7

    GPIOF->AFR[0] |= 3 << 28; // select AFR7 to AF3(TIM11_CH1)

    TIM11->CCMR1 |= 6 << 4; // set OC1M to PWM

    TIM11->CCER |= 1 << 0; // enable CH1

    TIM11->CR1 |= 1 << 0; // start timer

    TIM11->PSC = 70;
    TIM11->ARR = 64799;
    TIM11->CCR1 = 3955;
}
```

This really does bring up a few questions though, our timing for the PWM is accurate enough, but it isn't exact, and if we want to change the time of the on pulse, we have to change both ARR and CCR every single time!

```c
    // for 1.5ms of on pulse
    TIM11->ARR = 65407;
    TIM11->CCR1 = 4563;
```

Rounding the numbers, finding the ratio, set the CCR, there is too many moving parts that we have to manage. Is there a better way of changing the pulse lenghs where it involves less registers and ratios? Can we have more exact numbers? Yes we can.

## Using PWM Mode 2

Instead of composing the denominator with the largest ARR value, we should set a PSC that is exactally multiples of $\mu$ seconds. Lets do the calculations on the continuous motor (time range 20.75ms to 22.25). Lets set $PSC + 1 = 72$

$$\frac{1}{(20 + .75)ms} = \frac{216Mhz}{72 \cdot (ARR + 1)}$$

We get exactally $ARR + 1 = 63900$, no rounding done.

And instead of calculating the CCR with ratios, we will use PWM mode 2, which simply flips what the CCR describes. CCR in PWM2 will be the amount of time that the signal will be a 0, meaning we are specifying the off period. This has great effects on simplifying our maths! We know that between each pulse, there is a definite

20ms of off periods. We can calculate that directly without any ratios!

$$\frac{1}{20ms} = \frac{216Mhz}{72 \cdot CCR}$$

$CCR$ comes out to be exactally 60000.

This also has the upside of only needing to change the ARR, and not changing the CCR at all!

Making the servos adjustable is now simply finding the range of the ARRs for the timers, and we are all set to start programming.

| Timing Range | Minimum | Maximum |
|---|---|---|
| Cont. servo ARR | 41499 | 44499 |
| Stan. servo ARR | 63899 | 65099 |

## The code

The only numbers that are unexplained are the multiplications for changing ARRs. They are simply a ratio to match the LCD screen pixel range to the 1200 range, and the 3000 range of our ARRs.

```c
#include "stm32f7xx_hal.h"              // Keil::Device:STM32Cube HAL:Common
#include "Board_GLCD.h"                 // ::Board Support:Graphic LCD
#include "GLCD_Config.h"                // Keil.STM32F746G-Discovery::Board Support:Graphic LCD
#include "Board_Touch.h"                // ::Board Support:Touchscreen
#include <stdlib.h>


static void SystemClock_Config(void);
extern GLCD_FONT GLCD_Font_16x24;

int main(void)
{
    short x, y;
    TOUCH_STATE state;

    SystemClock_Config();

    RCC->AHB1ENR |= 1 << 5; // enable GPIO F
    RCC->APB2ENR |= 1 << 17 | 1 << 18; // enable TIM10 TIM11

    GPIOF->MODER |= 2 << 12 | 2 << 14; // AF mode on PF6 PF7

    GPIOF->AFR[0] |= 3 << 24 | 3 << 28; // select AFR6 to AF3(TIM10_CH1), AFR7 to AF3(TIM11_CH1)

    TIM10->CCMR1 |= 7 << 4; // set OC1M to PWM2
    TIM11->CCMR1 |= 7 << 4; // set OC1M to PWM2

    TIM10->CCER |= 1 << 0; // enable CH1
    TIM11->CCER |= 1 << 0; // enable CH1

    TIM10->CR1 |= 1 << 0; // start timer
    TIM11->CR1 |= 1 << 0; // start timer

    TIM10->PSC = 71;
    TIM10->ARR = 63899 + 600; // min + half = middle (stationary)
    TIM10->CCR1 = 60000;

    TIM11->PSC = 107;
    TIM11->ARR = 41499 + 1500; // min + half of full range = middle
    TIM11->CCR1 = 40000;

    GLCD_Initialize();
    GLCD_SetBackgroundColor(GLCD_COLOR_BLUE);
    GLCD_SetFont(&GLCD_Font_16x24);
    GLCD_ClearScreen();

    GLCD_SetForegroundColor(GLCD_COLOR_WHITE);
    GLCD_DrawHLine(0, GLCD_SIZE_Y / 2, GLCD_SIZE_X);
    GLCD_DrawVLine(GLCD_SIZE_X / 2, 0, GLCD_SIZE_Y);
    GLCD_SetForegroundColor(GLCD_COLOR_YELLOW);

    Touch_Initialize();

    x = GLCD_SIZE_X / 2;
    y = GLCD_SIZE_Y / 2;

    GLCD_DrawPixel(x, y);
    while(1)
    {
        Touch_GetState(&state);

        if (state.pressed == 1)
        {
            GLCD_SetForegroundColor(GLCD_COLOR_BLUE);
            GLCD_DrawPixel(x, y);

            x = state.x;
            y = state.y;
            GLCD_SetForegroundColor(GLCD_COLOR_YELLOW);
            GLCD_DrawPixel(x, y);

            TIM10->ARR = 63899 + 600 + (136 - y) * 5;  // y has a maximum of 272, 5 > 1200/262
            TIM11->ARR = 41499 + 1500 + (240 - x) * 7; // x has a maximum of 480, 7 > 3000/480
        }
    }
}

void SysTick_Handler (void)
{
    HAL_IncTick();
}

static void SystemClock_Config(void)
{
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_OscInitTypeDef RCC_OscInitStruct;
  HAL_StatusTypeDef ret = HAL_OK;

  /* Enable HSE Oscillator and activate PLL with HSE as source */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 25;
  RCC_OscInitStruct.PLL.PLLN = 432;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 9;

  ret = HAL_RCC_OscConfig(&RCC_OscInitStruct);
  if(ret != HAL_OK)
  {
    while(1) { ; }
  }

  /* Activate the OverDrive to reach the 216 MHz Frequency */
  ret = HAL_PWREx_EnableOverDrive();
  if(ret != HAL_OK)
  {
    while(1) { ; }
  }

  /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2 clocks dividers */
  RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  ret = HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7);
  if(ret != HAL_OK)
  {
    while(1) { ; }
  }
}

```