



EMBEDDED SYSTEMS
PROCESSOR ARCHITECTURE LABORATORY

**MSP432 Lab1 - ADC to control a servo-motor
using PWM**

Teacher :

Beuchat René

Student :

Lefebure Nicolas

21 octobre 2019

Table des matières

1	Introduction	2
2	Programming of the system	3
2.1	Periodic sampling of the analog signal send by the joystick	3
2.1.1	Clock system and Timer setting	3
2.1.2	Analog to digital conversion	4
2.2	Pulse Width Modulation generation	5
2.2.1	Clock system and Timer setting	5
2.2.2	Pulse Width Modulation management	6

1 Introduction

The main purpose of this laboratory was to understand the programming of several interfaces available on a micro-controller, in particular the MSP-EXP432P401R LaunchPadTM. In particular, we convert an analog signal (send by a joystick) to a digital one using the Analog to Digital (A/D) converter implemented in the LaunchPad. Based on the signal obtained, the micro-controller outputted a Pulse Width Modulated (PWM) signal with a width that was determined by the provided analog input, in order to control a Servo-motor. A diagram of the assembly system is visible in the figure 1.

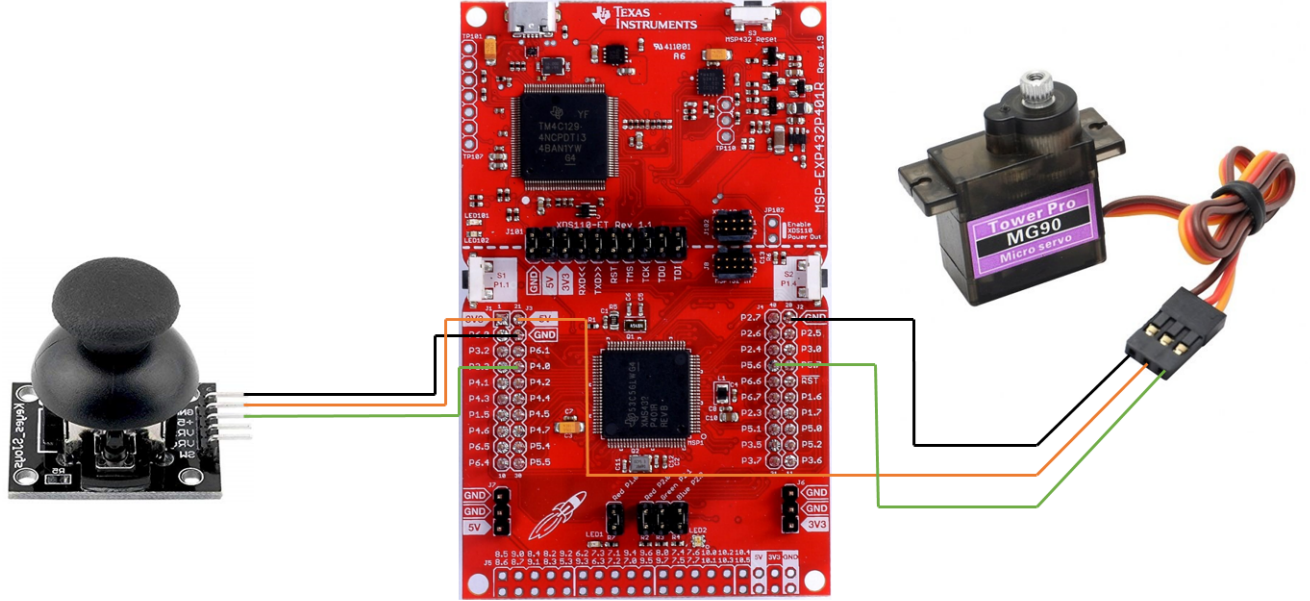


FIGURE 1: Diagram of the system assembly

The system is composed of three elements : the Keyes SJoys Joystick at the left, the MSP-EXP432P401R LaunchPadTM at the center and the MG90S servo motor at the right. We can see that the signal send by the joystick is collected by the Pin 4.0 and the signal collected by the servo motor is send to the Pin 5.6.

2 Programming of the system

The objective is to sample the analog signal release by the joystick and generate a Pulse width Modulation based on the sampling to conduct a servo motor. The implementation was done in two stages described in the section 2. In the section 2.1, we explain how we use a timer interrupt to periodically enable the ADC converter in the software to start a conversion of the joystick value. In the section 2.2, we demonstrate how we use another timer interrupt and a interrupt from the ADC14 module to catch the sampled value and use it to adjust the duty cycle of the pulses.

2.1 Periodic sampling of the analog signal send by the joystick

2.1.1 Clock system and Timer setting

The clock used for the periodic sampling was ACLK, which has a frequency of 32.768 [kHz]. No pre-division was selected for the clock. The Timer used to generate the interrupts was A0. It was configured in UP mode and in Compare mode with an output mode in Set/reset. As we set out the value of compare register (TA0CCR0) to 1638, the timer enables to start a new conversion in the ADC every 50 ms ($f = 32768 \text{ Hz}$, $50\text{ms} : 32768 \cdot 0.05 = 1638$), thanks to the Timer A0 interrupt function : TA0_0_IRQHandler. The signal aspect generated is visible in the figure 2. The following code describes the setting of the Timer A0 and its interrupt function.

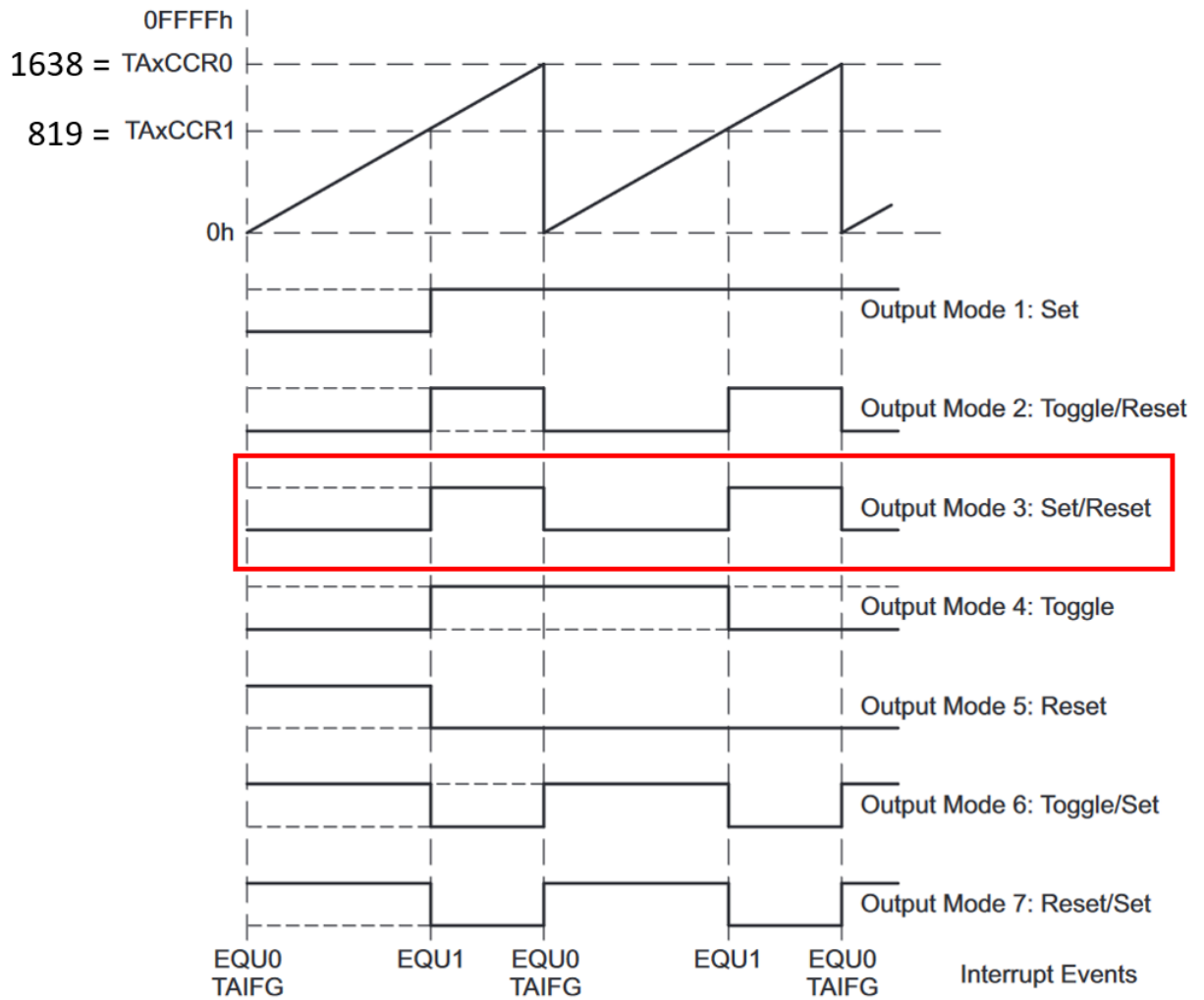


FIGURE 2: Display of the Timer A0 with specified settings

```

1 void TimerA0(uint16_t period, uint16_t pulse_width) {
2
3     TIMER_A0->CTL |= TIMER_A_CTL_TASSEL_1 | // Choice of ACLK for the clock
4                     TIMER_A_CTL_MC_UP | // Picking the UP mode
5                     TIMER_A_CTL_ID_0 | // There is no pre-division of the clock
6                     TIMER_A_CTL_IE; // Enabling the TimerA0 interrupt
7
8     TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CAP; // Disabling capture mode
9     TIMER_A0->CCTL[0] |= TIMER_A_CCTLN_CCIE; // Capture/compare interrupt enabling
10    TIMER_A0->CCTL[1] |= TIMER_A_CCTLN_OUTMOD_3; // Output mode Set/reset choice
11
12    TIMER_A0->CCR[0] |= period; // Value of compare register set to 1638 to have 50ms
13    // interrupts
14    TIMER_A0->CCR[1] |= pulse_width; // Half of the value of compare register as a
15    // start
16
17    NVIC_EnableIRQ(ADC14_IRQn); // Enable of the ADC14 Interrupt
18    NVIC_EnableIRQ(TA2_0_IRQn); // Enable of the Timer A2 Interrupt
19    NVIC_EnableIRQ(TA0_0_IRQn); // Enable of the Timer A0 Interrupt
20
21 }
22
23 void TA0_0_IRQHandler(void) {
24
25     TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG; // Resets the flag to zero
26
27     while (ADC14->CTL0 & ADC14_CTL0_BUSY) {}; // Waiting for the Busy to be at zero
28
29     ADC14->CTL0 |= ADC14_CTL0_SC; // Starting a conversion
30
31 }

```

2.1.2 Analog to digital conversion

To be able to convert the analog input signal given by the joystick, we used the channel 13 of the ADC (and had the Pin 4.0). A conversion was launched every 50ms thanks to the Timer A0 interrupt routine. When the ADC did achieve the whole conversion, the interrupt (ADC14_IRQHandler) was called, allowing the value to be stored in the register MEM0. Based on that value, the PWM of the servo motor was then settled. The following code describes the setting of the ADC and its interrupt routine.

```

1 void ADC14_Fct() {
2
3     P4->SEL0 |= (1<<0); // Analog output set to pin P4.0
4     P4->SEL1 |= (1<<0);
5
6     ADC14->CTL0 &= ~ADC14_CTL0_ENC; // Enabling modifications to ADC settings
7
8     while (ADC14->CTL0 & ADC14_CTL0_BUSY) {}; // Waiting for the Busy to be at zero:
9     // indicates an active sample or conversion operation
10
11    ADC14->CTL0 |= ADC14_CTL0_PDIV_0 | // No pre-division
12                  ADC14_CTL0_SHS_1 | // TA0.1 signal
13                  ADC14_CTL0_SHP | // Signal is sourced from the sampling timer.
14                  ADC14_CTL0_DIV_0 | // No pre-division for the clock
15                  ADC14_CTL0_SSEL__ACLK | // Choice of ACLK for the clock
16                  ADC14_CTL0_CONSEQ_2 | // Single-channel, single-conversion
17                  ADC14_CTL0_ON; // Power on
18
19    ADC14->CTL1 |= ADC14_CTL1_RES__14BIT;
20
21    ADC14->MCTL[0] &= ~ADC14_MCTLN_INCH_MASK;
22    ADC14->MCTL[0] |= ADC14_MCTLN_INCH_13; // Channel 13. P4.0

```

```

23     ADC14->IER0 |= ADC14_IER0_IE0; // Enabling ADC14IFG0 for interrupt
24
25     ADC14->CTL0 |= ADC14_CTL0_ENC; // Enable, stop modifications
26
27 }
28
29 void ADC14_IRQHandler(void) {
30
31     ADCvalue = ADC14->MEM[0] & 0xffff; // ADCvalue is put into MEM0
32
33     if(ADCvalue < 5000){
34         P2->OUT |= (1<<0);
35         P2->OUT &= ~(1<<1); // Red LED on
36         TIMER_A2->CCR[1] = Pulse_width_2_90; // Angle for servo = 90
37     }
38     else if(ADCvalue > 12000) {
39         P2->OUT |= (1<<1);
40         P2->OUT &= ~(1<<0); // Green LED on
41         TIMER_A2->CCR[1] = Pulse_width_2_m90; // Angle for servo = -90
42     }
43     else {
44         P2->OUT |= (1<<1);
45         P2->OUT |= (1<<0); // Yellow LED on
46         TIMER_A2->CCR[1] = Pulse_width_2_0; // Angle for servo = 0
47     }
48 }
49
50 }

```

2.2 Pulse Width Modulation generation

2.2.1 Clock system and Timer setting

The clock used for the settling of the servo motor PWM was also ACLK, which had a frequency of 32.768 [kHz]. As previously, no pre-division was selected for the clock. This time, the Timer used to generate the interrupts was A2. It was configured in UP mode and in Compare mode with an output mode in Set/reset. As we set out the value of compare register (TA0CCR0) to 656, the timer generated an interruption every 20 ms ($f = 32768 \text{ Hz}$, $20\text{ms} : 32768 \cdot 0.02 = 656$), thanks to the Timer A2 interrupt function : TA2_0_IRQHandler. We setted the value of compare register (TA0CCR1) to half of TA0CCR0, but it was later be changed according to ADC value to control the servo motor. The signal generated is the same as the figure 2. The following code describes the setting of the Timer A2 and its interrupt function.

```

1 void TA0_0_IRQHandler(void) {
2
3     TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG; // Resets the flag to zero
4
5     while (ADC14->CTL0 & ADC14_CTL0_BUSY) {}; // Waiting for the Busy to be at zero
6
7     ADC14->CTL0 |= ADC14_CTL0_SC; // Starting a conversion
8
9 }
10
11 void TimerA2(uint16_t period_2, uint16_t pulse_width_2_0) {
12
13     TIMER_A2->CTL |= TIMER_A_CTL_TASSEL_1 | // Choice of ACLK for the clock
14                  TIMER_A_CTL_MC_UP | // Picking the UP mode
15                  TIMER_A_CTL_ID_0 | // There is no pre-division of the clock
16                  TIMER_A_CTL_IE; // Enabling the TimerA2 interrupt
17
18     TIMER_A2->CCTL[0] &= ~TIMER_A_CCTLN_CAP; // Disabling capture mode
19     TIMER_A2->CCTL[0] |= TIMER_A_CCTLN_CCIE; // Capture/compare interrupt enabling
20     TIMER_A2->CCTL[1] |= TIMER_A_CCTLN_OUTMOD_3; // Output mode Set/reset choice
21

```

```

22     TIMER_A2->CCR[0] = period_2; // Value of compare register set to 656 to have 20ms
        interrupts
23     TIMER_A2->CCR[1] = pulse_width_2_0; // Half of the value of compare register as a
        start
24
25 }
26
27 void TA2_0_IRQHandler(void) {
28
29     TIMER_A2->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG; // Resets the flag to zero
30
31 }

```

2.2.2 Pulse Width Modulation management

The goal of the project was to control the angle of a servo motor. According to the data-sheet of the servo motor, this one's angle would change based on the PWM value it would receive. The different angles correspond to the corresponding PWM :

- Position 0 ° = 1.5 pulse -> PWM duty cycle = $\frac{1.5}{20} = 0.075$
- Position +90 ° = 2 pulse -> PWM duty cycle = $\frac{2}{20} = 0.1$
- Position -90 ° = 1 pulse -> PWM duty cycle = $\frac{1}{20} = 0.05$

To be able to control the servo motor, we processed the output given by the ADC of the joystick. As it would offer values from 0 to (around) 16000, we made the corresponding match :

- Position 0 ° = 5000 < ADC < 12000
- Position +90 ° = ADC < 5000
- Position -90 ° = ADC > 12000

The duty cycle of the PWM generated by the Timer A2 was easily changed with the value of compare register (TA0CCR1). The PWM of Timer A2 was associated with pin 5.6 to be directly sent to the servo. The signal output for the position 0 is showed thanks to the logical Analyser in the figure 3. The code for the servo control is given below.

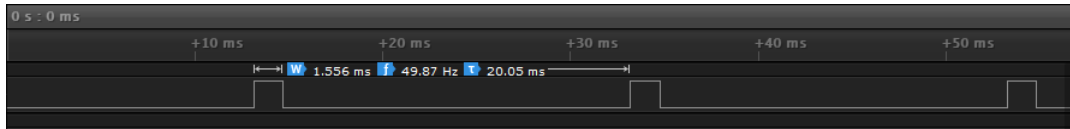


FIGURE 3: PWM output for the position 0

```

1  #define Period_0 1638 // f = 32768 Hz -> 50ms: 32768*0.05 = 1638
2  #define Period_2 656 // f = 32768 Hz -> 20ms: 32768*0.05 = 656
3
4
5  #define Ratio_duty_cycle_0 0.5
6  #define Ratio_duty_cycle_2_0 (1-0.075)
7  #define Ratio_duty_cycle_2_90 (1-0.1)
8  #define Ratio_duty_cycle_2_m90 (1-0.05)
9
10 #define Pulse_width_0 Ratio_duty_cycle_0*Period_0
11 #define Pulse_width_2_0 Ratio_duty_cycle_2_0*Period_2
12 #define Pulse_width_2_90 Ratio_duty_cycle_2_90*Period_2
13 #define Pulse_width_2_m90 Ratio_duty_cycle_2_m90*Period_2
14
15 volatile uint32_t ADCvalue;
16
17 void ADC14_IRQHandler(void) {
18
19     ADCvalue = ADC14->MEM[0] & 0xffff; // ADCvalue is put into MEM0
20
21     if(ADCvalue < 5000){
22         P2->OUT |= (1<<0);
23         P2->OUT &= ~(1<<1); // Red LED on

```

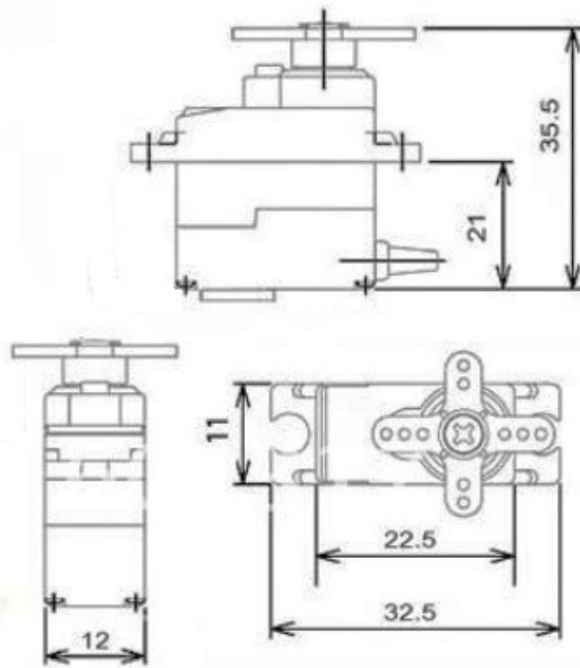
```

24         TIMER_A2->CCR[1] = Pulse_width_2_90; // Angle for servo = 90
25
26     }
27     else if(ADCvalue > 12000) {
28         P2->OUT |= (1<<1);
29         P2->OUT &= ~(1<<0); // Green LED on
30         TIMER_A2->CCR[1] = Pulse_width_2_m90; // Angle for servo = -90
31     }
32     else {
33         P2->OUT |= (1<<1);
34         P2->OUT |= (1<<0); // Yellow LED on
35         TIMER_A2->CCR[1] = Pulse_width_2_0; // Angle for servo = 0
36     }
37
38 }
39
40 void main(void)
41 {
42     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer
43
44     P5->DIR = 0x40; // Taking pin P5.6 as an output
45     P5->SEL0 |= 0x40; // Associating Timer2A function to the pin P5.6
46     P5->SEL1 &= ~0x40;
47
48     P2->SEL0 &= ~(1<<0); // Settings for Red LED
49     P2->SEL1 &= ~(1<<0);
50     P2->DIR |= (1<<0);
51     P2->OUT &= ~(1<<0);
52
53     P2->SEL0 &= ~(1<<1); // Settings for Green LED
54     P2->SEL1 &= ~(1<<1);
55     P2->DIR |= (1<<1);
56     P2->OUT &= ~(1<<1);
57
58     ADC14_Fct();
59     TimerA2(Period_2, Pulse_width_2_0);
60     TimerA0(Period_0, Pulse_width_0);
61     __enable_irq();
62
63     while(1){
64
65     }
66
67 }

```


MG90S

Metal Gear Servo




MG90S servo, Metal gear with one bearing

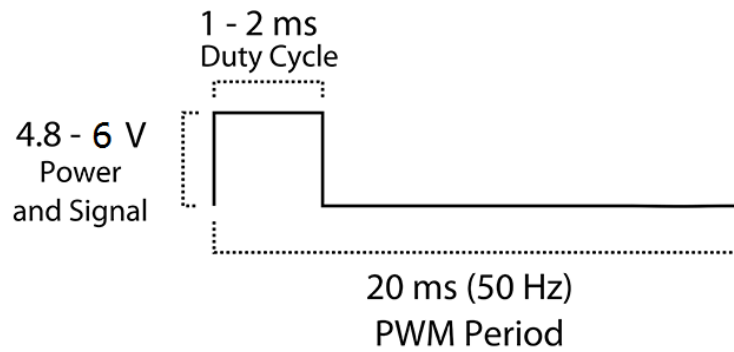
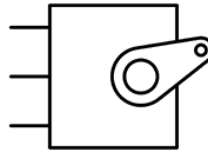
Tiny and lightweight with high output power, this tiny servo is perfect for RC Airplane, Helicopter, Quadcopter or Robot. This servo has *metal gears* for added strength and durability.

Servo can rotate approximately 180 degrees (90 in each direction), and works just like the standard kinds but *smaller*. You can use any servo code, hardware or library to control these servos. Good for beginners who want to make stuff move without building a motor controller with feedback & gear box, especially since it will fit in small places. It comes with a 3 horns (arms) and hardware.

Specifications

- Weight: 13.4 g
- Dimension: 22.5 x 12 x 35.5 mm approx.
- Stall torque: 1.8 kgf·cm (4.8V), 2.2 kgf·cm (6 V)
- Operating speed: 0.1 s/60 degree (4.8 V), 0.08 s/60 degree (6 V)
- Operating voltage: 4.8 V - 6.0 V
- Dead band width: 5 μ s

PWM=Orange ()
Vcc = Red (+)
Ground=Brown (-)



Position "0" (1.5 ms pulse) is middle, "90" (~2 ms pulse) is all the way to the right, "-90" (~1 ms pulse) is all the way to the left.