CS473: MSP432 laboratory
Controlling servomotor with analog joystick

Olivér Facklam

October 19, 2020

Contents

1 Problem statement

The goal of this lab is to create a system in which a servomotor's position is controlled by a joystick. We want to link the servomotor and the joystick together through some logic implemented on an MSP432 microcontroller. The servomotor's angle should follow the joystick's position.

More precisely the requirements are the following. The analog value provided by the joystick should be sampled and converted every 50 ms. The servomotor should be controlled by a pulse-width modulated signal (PWM), with a period of 20 ms and a width between 1 ms and 2 ms. The joystick value should be mapped to a PWM duty-cycle and used to update the servomotor's angle.

I decided to target a precision of 1%.

2 High-level system description

The following building blocks are required to achieve the desired operation. An overview of this setup is given in a block-diagram form in figure 1.

- The clock system is set up to provide a sufficiently precise and high-frequency signal to drive the processor and all the subsystems. The DCO (digitally controlled oscillator) is routed to the MCLK (master clock) and SMCLK (low-speed subsystem master clock) signals.
- Two timers are used: TIMER_A0 is dedicated to generating the PWM signal for the servomotor control, while TIMER_A3 is used to periodically initiate the sampling and conversion of the analog value.
- TIMER_A0 uses the SMCLK signal and operates in UP mode, with a period corresponding to the required PWM period, which is 20 ms. The capture-compare block CCR1 is used to set the PWM duty cycle. It operates in a RESET/SET output mode, generating the required PWM signal which is routed to pin 2.4.
- TIMER_A3 is set to UP mode and is used to trigger a processor interrupt at the predefined rate (every 50 ms).
- The analog-to-digital converter (ADC) uses the SMCLK signal and pulse sample mode for the conversions. These are triggered by the processor every 50 ms. The analog channel A13 (corresponding to port 4.0) is sampled, and the value is stored in memory slot 0. A processor interrupt is triggered at the end of the conversion, allowing it to read the sampled value.
- The processor operates with two interrupts. The interrupt from TIMER_A3 is used to trigger the conversion process in the ADC. The interrupt from the ADC is used to read the sampled value, do some calculations and finally update the duty-cycle of the PWM in TIMER_A0's CCR1.

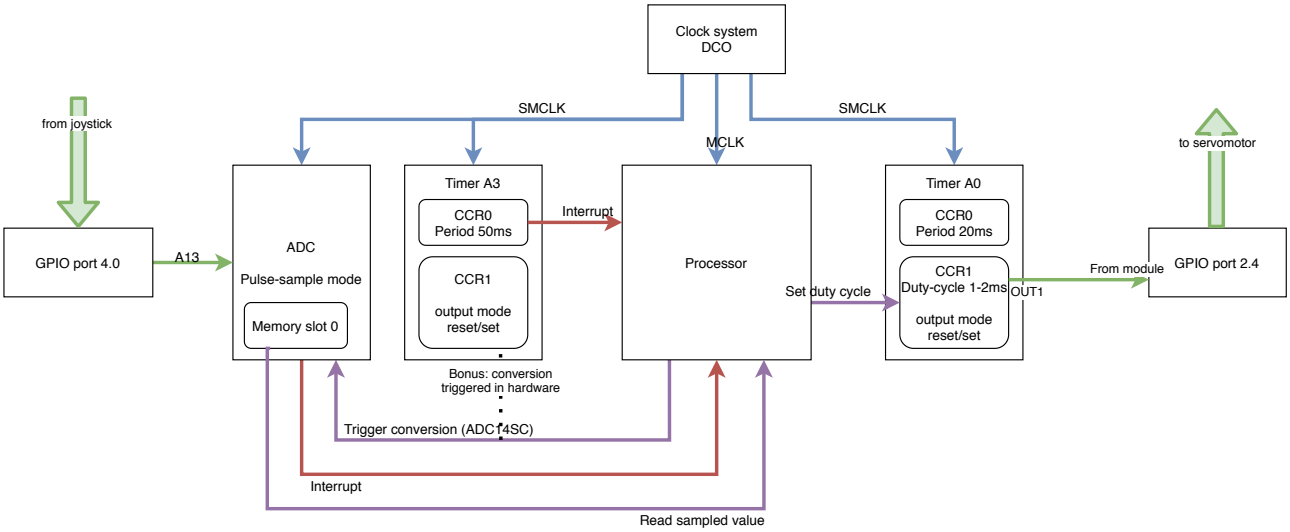


Figure 1: Block diagram of system operation

3 Detailed operation description

3.1 General setup

The general setup is controlled by `main.c`. The function `manip_7` implements the targeted system, with a software-triggered ADC conversion. The function `manip_7bis` uses a timer-triggered ADC conversion.

First, in both cases the clock system is set up by configuring and linking the DCO. Then the ADC is set to the correct mode with input A13 mapped to memory slot 0. Interrupts are enabled for the ADC and the function `update_pwm` is set as the interrupt handler. Finally PWM is set up on pin 2.4 for the servomotor.

The two versions differ in their way of triggering conversions:

- `manip_7` configures a periodic interrupt which triggers the ADC conversion by software every 50 ms.
- `manip_7bis` creates a PWM of period 50 ms on TIMER_A3 which directly triggers the ADC conversion.

```
/**
 * Manipulation 7 -- periodic interrupt to trigger ADC conversion to update PWM duty cycle for servo
 */
void update_pwm(int value) {
        // Calculate new PWM duty cycle (between 0.05 and 0.1)
        float duty = 0.05 + 0.05 * value / (BIT(14) - 1);
        pwm_hw_fixed_update(duty);
}

void manip_7() {
        // ADC inputs
        adc_setup(ADC14_CTL0_SSEL__SMCLK, ADC14_CTL0_SHS_0, ADC14_CTL1_RES__14BIT, ADC14_CTL0_CONSEQ_0); // software trigger, single conversion
        adc_memory_setup(0, 13, 5); // A13 to memory idx 0
        port_to_adc(0, &P4->SEL0, &P4->SEL1); // P4.0 is A13

        // Interrupts for conversion finish
        adc_irq_handler(update_pwm);
        adc_interrupts_enable();

        // Set up initial PWM for servo
        pwm_hw_fixed_init(0.05);

        // Set up periodic conversion
        periodic_interrupt(50, trigger_conversion);
}

void manip_7bis() {
        // ADC inputs
        adc_setup(ADC14_CTL0_SSEL__SMCLK, ADC14_CTL0_SHS_7, ADC14_CTL1_RES__14BIT, ADC14_CTL0_CONSEQ_2); // timer A3 C1 trigger, repeat single channel
        adc_memory_setup(0, 13, 5); // A13 to memory idx 0
        port_to_adc(0, &P4->SEL0, &P4->SEL1); // P4.0 is A13

```

```
33          // Interrupts for conversion finish
34          adc_irq_handler(update_pwm);
35          adc_interrupts_enable();
36
37          // Set up initial PWM for servo
38          pwm_hw_fixed_init(0.05);
39
40          // Set up periodic conversion with timer A3 C1 triggering the conversion
41          adc_conversion_enable();
42          timer_pwm_setup(TIMER_A3, 1, 50, 0.5); // don't care about duty cycle
43  }
44
45  void main(void) {
46          WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;  // stop watchdog timer
47
48          dco_setup(DCO_FREQ);
49          dco_to_master(M_DIV, SM_DIV);
50
51          manip_7();
52          while(1);
53  }
```

3.2 Clock system

We need a sufficiently precise and high-frequency clock signal to target a 1% precision on the following timing constraints:

- 50 ms for the sampling period
- 20 ms for the PWM period
- 1 ms for the PWM active time

A 1% precision on the 1 ms active time requires a timer granularity of 10 µs. Thus I decided to configure the clock signal in the MHz range.

The digitally-controlled oscillator (DCO) operates in the MHz range. According to the device-specific datasheet, it also has an acceptable precision of 1.2% when in external resistor mode.

So I decided to use the DCO at a frequency of 4 MHz. It is connected to MCLK with a divider of 1, and to SMCLK with a divider of 4. This way, the processor runs at 4 MHz and the subsystems at 1 MHz.

The code for configuring the DCO in a generic way can be found in `time.c`.

```
1   /**
2    * Set up the DCO clock
3    */
4   void dco_setup(int khz) {
5           if(khz < 1000 || khz >= 64000)
6                   return;
7
8           // calculate correct frequency mode
9           uint8_t mode = log2(khz/1000);
10          uint32_t rsel = (mode << CS_CTL0_DCORSEL_OFS) & CS_CTL0_DCORSEL_MASK;
11          int center_frq = ((1 << mode) + (1 << (mode+1))) * 500; // in khz
12
```

```
        // get calibration values
        uint32_t fcal = mode < 5 ? TLV->DCOER_FCAL_RSEL04 : TLV->DCOER_FCAL_RSEL5;
        float k = *((float*) (mode < 5 ? &TLV->DCOER_CONSTK_RSEL04 : &TLV->
            ↪ DCOER_CONSTK_RSEL5));

        // calculate tuning with formula
        uint16_t n_tune = (khz-center_frq) * (1 + k * (768-fcal)) / (khz * k);
        uint32_t tune = n_tune & CS_CTL0_DCOTUNE_MASK;

        // write-enable the clock system
        CS->KEY = CS_KEY_VAL;

        // switch to external resistor
        CS->CTL0 &= ~CS_CTL0_DCORSEL_MASK;
        CS->CTL0 |= CS_CTL0_DCORSEL_1;
        CS->CTL0 |= CS_CTL0_DCORES;

        // write DCO tuning info
        CS->CTL0 &= ~(CS_CTL0_DCORSEL_MASK | CS_CTL0_DCOTUNE_MASK);
        CS->CTL0 |= rsel | tune;

        // lock
        CS->KEY = 0;
}
```

3.3 Timers

3.3.1 Timer block configuration

Both our timers need to be configured in UP mode, to allow a periodic operation (with periods of 20 ms and 50 ms). This setup is done in the following two functions `timer_block_setup_mode_up` and `timer_block_setup`.

```
/**
 * Setup timer block
 */
void timer_block_setup(Timer_A_Type* T, uint16_t mode, uint16_t source, int div_id,
    ↪ int div_idex) {
        uint16_t ctl_id = (div_id << TIMER_A_CTL_ID_OFS) & TIMER_A_CTL_ID_MASK;
        uint16_t ex0_idex = (div_idex - 1) & TIMER_A_EX0_IDEX_MASK;

        // set input, mode & divider ID
        T->CTL &= ~(TIMER_A_CTL_MC_MASK | TIMER_A_CTL_ID_MASK |
            ↪ TIMER_A_CTL_SSEL_MASK);
        T->CTL |= mode | source | ctl_id;

        // set second divider
        T->EX0 = ex0_idex;
}


```

```
/**
 * Set the timer block to UP mode, with the specified period
 */
int timer_block_setup_mode_up(Timer_A_Type* T, int period) {
        // get cycle count for period
        int div_id, div_idex;
        int num_cycles = calculate_cycles(period, &div_id, &div_idex);
        if(num_cycles < 0)
                return -1;

        // setup input
        timer_block_setup(T, TIMER_A_CTL_MC__UP, TIMER_A_CTL_SSEL__SMCLK, div_id,
            ↪ div_idex);
        // clear interrupt flag & set compare mode
        T->CCTL[0] &= ~(TIMER_A_CCTLN_CCIFG | TIMER_A_CCTLN_CAP);
        // EQU0 for max value
        T->CCR[0] = num_cycles;

        return 0;
}
```

The function `calculate_cycles` determines the number of clock cycles which correspond to the targeted period, knowing the clock frequency. It also adjusts and returns the required divider values. This dynamic scaling allows us to use period values from 1 ms up to 4 s.

3.3.2 PWM generation

Once the timer block is set up, generating a PWM is just a question of setting the correct duty cycle and output mode in one of the capture/compare registers of the timer. `timer_pwm_setup` initializes such a register to output mode RESET/SET, and `timer_pwm_duty` allows to modify the duty cycle.

```
/**
 * Create PWM on timer output
 */
void timer_pwm_setup(Timer_A_Type* T, int idx, int width, float duty_cycle) {
        // idx between CCR1 and CCR4
        if(idx < 1 || idx > 4)
                return;

        // reset timer
        T->CTL |= TIMER_A_CTL_CLR;

        // set timer block to up mode, with specified width
        if(timer_block_setup_mode_up(T, width) < 0)
                return;

        // EQUn for active value
        timer_pwm_duty(T, idx, duty_cycle);

        // outmod -> reset/set
        T->CCTL[idx] &= ~TIMER_A_CCTLN_OUTMOD_MASK;
        T->CCTL[idx] |= TIMER_A_CCTLN_OUTMOD_7;
}
```

```
23
24
25 /**
26  * Edit PWM duty cycle
27  */
28 void timer_pwm_duty(Timer_A_Type* T, int idx, float duty_cycle) {
29         // idx between CCR1 and CCR4
30         if(idx < 1 || idx > 4)
31                 return;
32
33         // duty between 0 and 1
34         if(duty_cycle < 0 || duty_cycle > 1)
35                 return;
36
37         T->CCR[idx] = duty_cycle * T->CCR[0];
38 }
```

Note: functions `pwm_hw_fixed_init` and `pwm_hw_fixed_update`, which are called in `main.c`, are simply wrappers around `timer_pwm_setup` and `timer_pwm_duty` respectively. They additionally configure the correct GPIO port to accept the timer module output.

3.3.3 Timer interrupts

Processor interrupts are configured for TIMER_A3 by the function `periodic_interrupt`. It sets the timer to UP mode, and enables the interrupt triggers and their handling. A pointer to a callback function is passed as an argument: this function will be called by the IRQ handler at every interrupt.

```
1  /**
2   * Global variable to store timer callback function pointer
3   */
4  TimerHandler timer_handler = NULL;
5
6
7  /**
8   * Timer A3 interrupt handler -> calls callback
9   */
10 void TA3_0_IRQHandler(void) {
11         // Clear interrupt flag before doing anything
12         TIMER_A3->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;
13
14         // Call callback if one is set
15         if(timer_handler)
16                 timer_handler();
17 }
18
19
20 /**
21  * Set a new timer callback
22  */
23 void timer_irq_handler(TimerHandler fun) {
24         timer_handler = fun;
25 }
26
```

```
27
28 /**
29  * Periodic interrupt, calling function fun
30  * Uses timer A3
31  */
32 void periodic_interrupt(int period, TimerHandler fun) {
33         Timer_A_Type* T = TIMER_A3;
34         IRQn_Type irq = TA3_0_IRQn;
35
36         if(timer_block_setup_mode_up(T, period) < 0)
37                 return;
38
39         // enable CCR interrupts triggers
40         T->CCTL[0] |= TIMER_A_CCTLN_CCIE;
41
42         // set handler
43         timer_irq_handler(fun);
44
45         // enable interrupt handling
46         NVIC_EnableIRQ(irq);
47         NVIC_SetPriority(irq, 4);
48 }
```

3.4 Precision ADC

3.4.1 Conversion setup

The ADC configuration is done using two functions. The first one, called `adc_setup`, initializes the ADC to pulse mode sampling, and then sets conversion mode, clock input, trigger signal and resolution.

```
1  /**
2   * Set up ADC inputs (pulse mode)
3   */
4  void adc_setup(uint32_t clk_src, uint32_t trigger, uint32_t resolution_mode,
       ↪ uint32_t conversion_mode) {
5          // Turn ADC on, clear conversion
6          ADC14->CTL0 |= ADC14_CTL0_ON;
7          adc_conversion_disable();
8
9          // Select pulse mode
10         ADC14->CTL0 |= ADC14_CTL0_SHP;
11
12         // Conversion mode
13         ADC14->CTL0 &= ~ADC14_CTL0_CONSEQ_MASK;
14         ADC14->CTL0 |= (conversion_mode & ADC14_CTL0_CONSEQ_MASK);
15
16         // Select clock
17         ADC14->CTL0 &= ~ADC14_CTL0_SSEL_MASK;
18         ADC14->CTL0 |= (clk_src & ADC14_CTL0_SSEL_MASK);
19
20         // Select trigger signal
21         ADC14->CTL0 &= ~ADC14_CTL0_SHS_MASK;
22         ADC14->CTL0 |= (trigger & ADC14_CTL0_SHS_MASK);
```

```
23
24          // Select resolution
25          ADC14->CTL1 &= ~ADC14_CTL1_RES_MASK;
26          ADC14->CTL1 |= (resolution_mode & ADC14_CTL1_RES_MASK);
27  }
```

The second one, `adc_memory_setup`, configures a certain memory control, links it to an input channel and enables interrupts for it.

```
1   /**
2    * Set up ADC memory control, link to input channel, enable interrupts for this
         ↪ memory
3    */
4   void adc_memory_setup(int idx, int channel, int sample_time_us) {
5           // Single-ended mode, with ref (Vcc, Vss)
6           ADC14->MCTL[idx] &= ~ADC14_MCTLN_DIF;
7           ADC14->MCTL[idx] &= ~ADC14_MCTLN_VRSEL_MASK;
8
9           // Select timer mode
10          int mode = calculate_adc_timer_mode(sample_time_us);
11          if(mode < 0 || mode > 7)
12                  return;
13
14          int mask, ofs;
15          if(idx <= 7 || idx >= 24) {
16                  mask = ADC14_CTL0_SHT0_MASK;
17                  ofs = ADC14_CTL0_SHT0_OFS;
18          } else {
19                  mask = ADC14_CTL0_SHT1_MASK;
20                  ofs = ADC14_CTL0_SHT1_OFS;
21          }
22          ADC14->CTL0 &= ~mask;
23          ADC14->CTL0 |= (mode << ofs) & mask;
24
25          // Select channel
26          ADC14->MCTL[idx] &= ~ADC14_MCTLN_INCH_MASK;
27          ADC14->MCTL[idx] |= channel & ADC14_MCTLN_INCH_MASK;
28
29          // Set conversion address to this memory
30          ADC14->CTL1 &= ~ADC14_CTL1_CSTARTADD_MASK;
31          ADC14->CTL1 |= (idx << ADC14_CTL1_CSTARTADD_OFS) & ADC14_CTL1_CSTARTADD_MASK
                ↪ ;
32
33          // Enable interrupt for this memory
34          ADC14->CLRIFGR0 |= 1 << idx;
35          ADC14->IER0 |= 1 << idx;
36  }
```

3.4.2 ADC interrupts

The management of ADC interrupts is similar to that of timer interrupts. A function pointer to a callback can be stored to be used as the interrupt handler. At each ADC14 interrupt, the interrupt vector is analyzed to detect the interrupt reason. If it corresponds to a finished

conversion, the sampled value is read from the ADC's memory control and is handed as an argument to the callback. This logic is implemented by the following snippet of code.

```
/**
 * Global variable to store ADC callback function pointer
 */
AdcHandler adc_handler = NULL;


/**
 * ADC interrupt handler
 */
void ADC14_IRQHandler(void) {
        // Read interrupt vector & get memory index
        int int_nb = ADC14->IV;
        int idx = int_nb/2 - 6;

        // if not a conversion finish, simply clear flags
        if(idx < 0 || idx > 31) {
                ADC14->IV = 0;
                return;
        }

        // Reading the value resets the interrupt
        int value = ADC14->MEM[idx];

        // Call handler with value
        if(adc_handler)
                adc_handler(value);
}


/**
 * Set a new ADC callback
 */
void adc_irq_handler(AdcHandler fun) {
        adc_handler = fun;
}
```

4 Operation & results

4.1 Observed results

Table 1 shows the inputs and corresponding reactions in three simple cases: the two extreme positions, and the neutral position. The corresponding PWM signals on the logic analyzer are displayed in figure 2.

4.2 Precision considerations

During this whole lab, I decided to target a precision of 1%.

We can immediately note that the period of the PWM is 19.91 ms for an expected 20 ms, meaning an error of 0.5%. Thus the clock system and timer block respect the targeted precision.

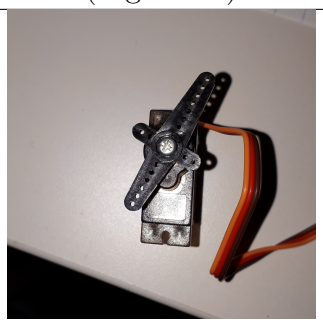
Situation	minimum	neutral	maximum
Input			
Sampled value	0 – 9	8428 – 8433	16369 – 16373
PWM	Width: 0.9955 ms Period: 19.91 ms Duty cycle: 5.000% (Figure 2a)	Width: 1.507 ms Period: 19.91 ms Duty cycle: 7.569% (Figure 2b)	Width: 1.990 ms Period: 19.91 ms Duty cycle: 9.995% (Figure 2c)
Result			

Table 1: Inputs & results in three simple cases

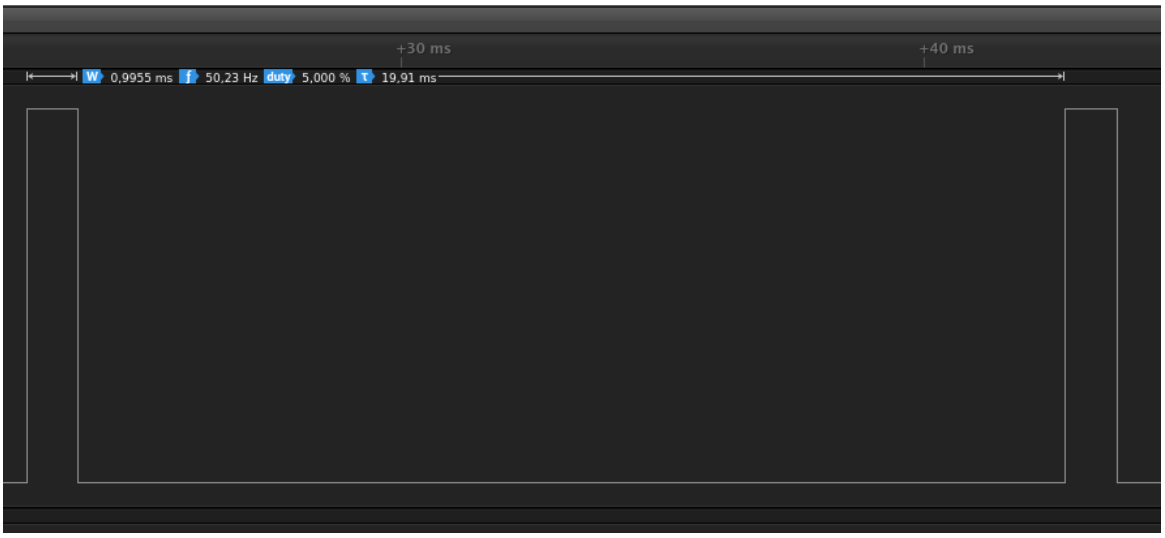
The sampled values cover almost the whole range of the 14-bit ADC; however the maximum value doesn't quite reach $2^{14} - 1 = 16383$. I believe this is simply due to the potentiometer's range, but I haven't been able to verify its output voltage. Also, the neutral position doesn't seem to correspond exactly to $V_{cc}/2$, and this is reflected on the PWM width.

The PWM width ranges from 0.9955 ms to 1.990 ms, which corresponds to the expected 1 ms to 2 ms range with a precision of 0.5%.

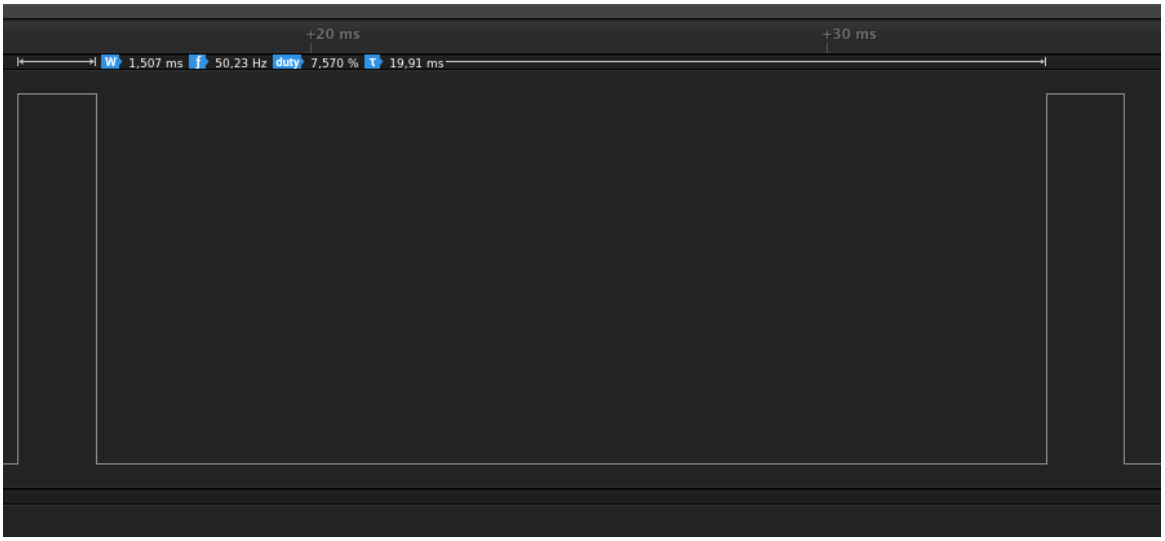
5 Conclusion

The analog voltage of the joystick is sampled every 50 ms by the 14-bit ADC and is used to determine the width of the output PWM. This PWM respects the constraints (period of 20 ms with a width of 1–2 ms) with a precision better than 1%.

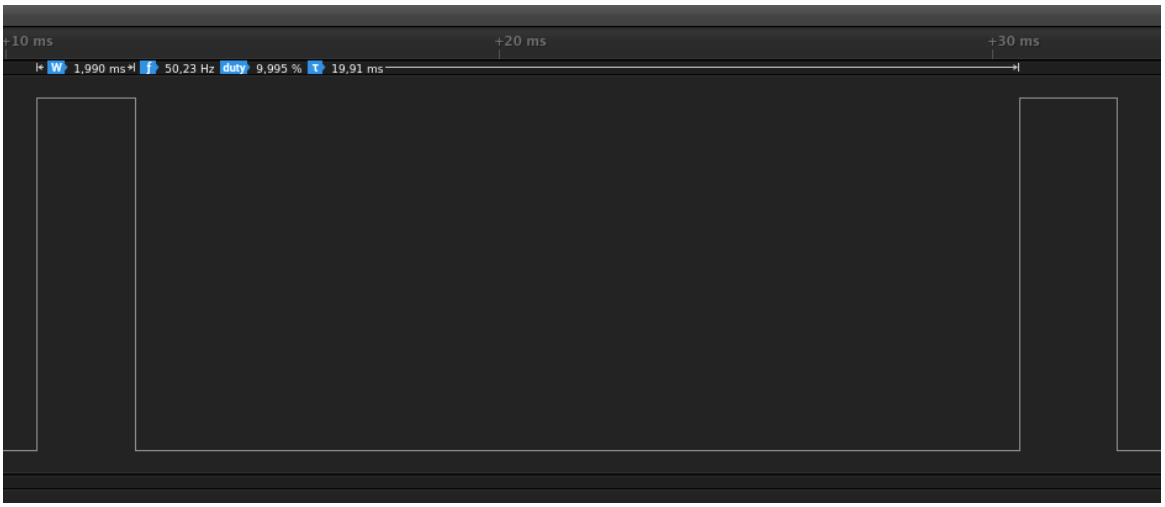
The servomotor is correctly controlled by the joystick. It reacts quickly to changes in joystick position, and the resolution of its angle also seems satisfactory. You can see the system working in the demo video.



(a) PWM for minimum input



(b) PWM for neutral input



(c) PWM for maximum input

Figure 2: PWM on pin 2.4 for three different inputs