UNIVERSITÉ DE LIÈGE

INFO2055-1 Embedded Systems Project

FINAL REPORT
SUNFLOWER SOLAR PANEL

HOORELBEKE Jordi - MEUNIER Loïc - NAA Marco

Academic year 2022-2023

Contents

1 Introduction

To improve a payload's[1] efficiency, one can make use of a *solar tracker*, i.e. a device that orients itself towards the sun[2]. The idea behind this project is to mimic a simplified and miniaturized version of such a device.
We will start by explaining the general idea behind the system: its behavior and how it interacts with its environment.
In the next section, one can find technicalities concerning the hardware and the software such as the electronic circuits as well as the code's architecture, what time constraints the system has to fulfill, etc.
Eventually, we will summarize by describing how the system turned out and what improvements could have been done.

2 System behavior

This section presents the overall idea of the system. More in depth explanations will be given later in the report.
The system's behaviour is pretty simple: when turned on, the system should orient the platform -containing a solar panel for instance- towards the direction where the light intensity is maximum.

2.1 Rotate function

The first concept is horizontal and vertical rotation. This is achieved by using servomotors, i.e. components that convert electrical energy into mechanical energy and are used for precise control. The vertical rotation is managed by a closed loop type servomotor (180°) and the horizontal rotation by an open loop type (360°).

2.2 Light detection function

To detect light, we use photoresistors, also called LDRs. An LDR has a internal resistance that depends on the light intensity received by the component. By using two LDRs, we can compare the two resistances and adjust the positions of the servomotors accordingly. As we need to rotate on the two axis, two LDRs per axis will be used and arranged to form a quad-sensor. Each LDR is separated from the others by a partition in order to make sure that the only way the position is optimal is when the solar tracker faces the brightest source of light.

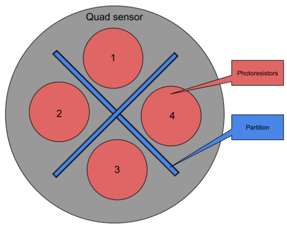


Figure 1: Quadsensor sketch.

2.3 Structure

Figure 2: 3D printed structure.

The structure holding everything together is separated into three parts[3], which have been 3D printed at the Montefiore institute thanks to Professor Boigelot. The two articulations incorporate the two servomotors: the 360° servomotor faces the pedestal part and takes care of the rotation on the horizontal axis, while the 180° one is attached perpendicularly so that the system can rotate on the vertical axis. The system is incorporated on a shoe-box so the electronic circuit can be stored inside.

All the pictures of the quadsensor and the final model can be seen in the appendix (see 6)

[1] Payloads are usually solar panels, parabolic troughs, fresnel reflectors, lenses or the mirrors of a heliostat.
[2] According to the Wikipedia page of Solar Trackers.
[3] The .plt patterns of these components were designed by Mr. Fernando Bueno and can be found at https://www.thingiverse.com/thing:708819

3 Sensors and actuators validation

This section will explain how we managed to determine how the different components work, and the methods we used in the first place to test their individual mechanisms, as well as if they have no malfunction.
We can divide this section into two main parts: LDR value acquisition and servomotor rotation.

3.1 LDR value acquisition test

For this part, we only used 2 of the 4 LDRs we had. To test our understanding, as well as the LDR working, our test setup was to turn on a LED if one given LDR receives more light than the other, and to turn it off otherwise.
This gave us the following code:

```
loop:    btfsc    ready, 0
         call     computation
         goto     loop

computation:
         clrf     ready
         call     ldr0
         call     ldr1
         call     difference_h
         return

ldr0:
         banksel  ADCON0
         movlw    0x81
         movwf    ADCON0   ; ADC enabled on channel AN0 with 10-bit result
         call     delay
         bsf      ADCON0, 1
         btfsc    ADCON0, 1
         goto     $-1
         banksel  ADRESH
         movf     ADRESH, 0
         movwf    ldr0h
         movf     ADRESL, 0
         movwf    ldr0l
         return

ldr1:
         banksel  ADCON0
         movlw    0x85
         movwf    ADCON0   ; ADC enabled on channel AN1 with 10-bit result
         call     delay
         bsf      ADCON0, 1
         btfsc    ADCON0, 1
         goto     $-1
         banksel  ADRESH
         movf     ADRESH, 0
         movwf    ldr1h
         movf     ADRESL, 0
         movwf    ldr1l
         return

difference_h:
         movf     ldr0h, 0
         subwf    ldr1h
         btfsc    STATUS, 2        ; enter if Z = 1 (equal)
         goto     difference_l
         btfsc    STATUS, 0        ; enter if ldr0 > ldr1
         goto     led_off
         goto     led_on

difference_l:
         movf     ldr0l, 0
         subwf    ldr1l
         btfsc    STATUS, 2        ; enter if Z = 1 (equal)
         goto     led_off
         btfsc    STATUS, 0        ; enter if ldr0 > ldr1
         goto     led_off
         goto     led_on

led_on:
         banksel  PORTB
         movlw    0xff
         movwf    PORTB
```

```
        return

led_off:
        banksel PORTB
        movlw   0x00
        movwf   PORTB
        return
```

This part worked fine from the beginning, and we could not report any malfunction in the LDR. The only trouble we have noticed is their high sensibility and therefore their vulnerability to noise.

3.2 Servomotor rotation test

This section was the most critical in our tests, due to the fact that the PWM management is probably the most technical part of this project, with a microcontroller such as PIC16F789. Indeed, these are not really well suited for the pulse width modulator, especially for servomotors. The reason is that the period of 20ms we had to set (see datasheet) is not possible to achieve with the 32MHz clock rate. Therefore, we had to slow down the whole PIC to 2MHz. In fact, this was not a critical issue as, for our application, we do not need such speed, but for some other projects this could have been an issue and we would have to create the signal by ourselves using the GPIO ports.
In order to test the servomotors and the PWM, we wanted to compute the 2 extreme values of it and feed them to the motor at initialization. For instance, this should make the vertical servomotor go to -90° or 90°at the start. Thanks to the equation 25-1[4]:

$$PWM\_period = (PR2+1) \times 4 \times Tosc \times TMR2\_prescaler \tag{1}$$

$$\Leftrightarrow PR2 = \frac{0.2}{\frac{4}{Fosc} \times TMR2\_prescaler} - 1 \tag{2}$$

With the $TMR2\_prescaler = 64$ and the $Fosc = 2Mhz$, we get that $PR2 = 0x9B \approx 155$. Then we had to compute the value of the PWM signal, for this, we have first to determine which duty cycle ratio we want to achieve. Looking at the servomotor datasheet, we see that we need it to be 5% for -90° and 10% for 90°. We can thus replace these values in the equation 25-3:

$$Duty\_Cycle\_Ratio = \frac{CCPRxL : CCPxCON < 5:4 >}{4(PR2+1)} \tag{3}$$

We finally get the two theoretical values for the extremes:

- 90°: $CCPRxL = 0x08$ and $CCPxCON < 5:4 > = 0$ from which we can derive, on 1 byte the value $servov = 0x08$.
- -90°: $CCPRxL = 0x0F$ and $CCPxCON < 5:4 > = 2$ from which we can derive, on 1 byte the value $servov = 0x3F$

Once implemented, it did not work. To determine the reason, we went to the R100 to test with an oscillator if our calculations were correct. Luckily, they were. The problem was coming from the battery, which was running low, resulting in a servomotor not moving. Once we swapped batteries, we were able to observe that the motors were working well. Here is the code corresponding:

```
init_pwm:
        banksel PORTC
        clrf    PORTC
        banksel LATC
        clrf    LATC
        banksel TRISC
        movlw   0xff
        movf    TRISC
        banksel PR2
        movlw   0x9b
        movwf   PR2
        banksel CCP1CON
        bsf     CCP1CON, 2
        bsf     CCP1CON, 3
        movlw   0x08
        movwf   CCPR1L          ; or 0x3F
        bcf     CCP1CON, 4
        bcf     CCP1CON, 5      ; or bsf
        banksel PIR1
        bcf     PIR1, 1
        bsf     T2CON, 0
        bsf     T2CON, 1
        bsf     T2CON, 2
        banksel TRISC
        clrf    TRISC
        return
```

[4]page 228 of the datasheet

4 In depth review: Hardware

We just had a brief look at how the system is assembled and how it should interact with its environment. Let us now have a look at the electronic circuit attached, for convenience, to a Breadboard.

4.1 Electronic circuit

In order for the whole structure to operate as intended, the microcontroller PIC16F1789 from Microchip is used. All components are connected to a Breadboard, and powered by a 9V battery.

4.2 Components

Below is an exhaustive list of the required components to build this project.

Quantity	Name	Reference
1	180° servo motor	MP-708-0001
1	360° servo motor	MP-708-0002
4	Photoresistor	N5AC501085
1	$9V$ GP Lithium battery	/
4	100Ω resistor	/
1	330Ω resistor	/
1	$2200\mu F/25v$ capacitor	/
1	$100\mu F/50v$ capacitor	/
1	$10\mu F/50v$ capacitor	/
1	$2.2\mu F/25v$ capacitor	/
1	linear regulator 7805	/
1	Standard recovery diode	1N4001-T
1	Light-emitting diode	/
1	PIC16(L)F1789	/

These components have been assembled in the following manner in order to compose the circuit.

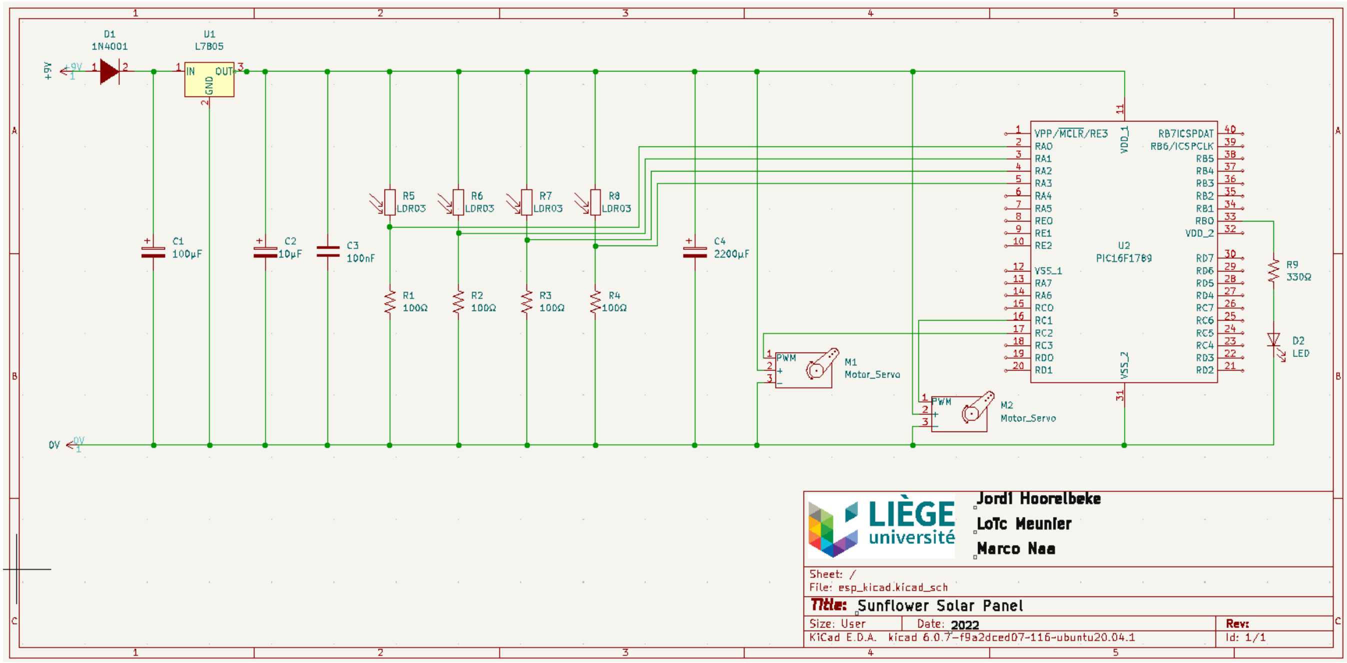


Figure 3: Schematic made on KiCad.

To understand this circuit, lets review the crucial parts of it:

- $D1$ is used to prevent the potential harm that could be caused by the current flowing in the wrong direction.
- The system is powered by a $9V$ battery and uses a PIC16F1789 as microcontroller.
- The microcontroller requires $5V$ to function correctly, hence, a linear regulator $U1$ is used to put down the initial $9V$ to $5V$.

- $C3$ is a decoupling capacitor, preventing brown-out by powering the digital components during voltage dips caused by the simultaneous switching of the transistors inside the microcontroller.
- Since the pic cannot directly interpret resistance, we need to use a voltage divider to use our photoresistors. Hence, each LDR has one side connected to 5 $volts$, the other side to an analog input pin ($RA < 0 : 3 >$), and a resistor connected between the pin and GND. This voltage divider will output a high voltage when it is getting a lot of light and a low voltage when little or no light is present.
- Both servomotors are connected to the microcontroller through pin $RC1$ and $RC2$.
- The capacitor $C4$ are used to regulate the current inside the circuit.
- $D2$ is a LED that is used for debugging.
- The whole circuit relies on two distinct breadboards connected to each other through two wires. The smaller breadboard has a 9V passing through, while the big one has 5V. This is not mandatory but very convenient in order to clean up the circuit (i.e. use less wires and ease debugging).

5 In depth review: Software

Now that we have a good understanding of the system's hardware, we must still program the PIC16F1789 microcontroller. Therefore, we will now take a look at the code written in assembly language[5].

5.1 Beforehand configuration bits

```
PROCESSOR 16F1789 ; used processor definition

#include <xc.inc> ; header naming a bunch of register

CONFIG  FOSC = INTOSC           ; INTOSC oscillator
CONFIG  WDTE = OFF              ; Watchdog Timer disabled
CONFIG  PWRTE = ON              ; Power-up Timer enabled
CONFIG  MCLRE = ON              ; MCLR/VPP pin function is MCLR
CONFIG  CP = OFF                ; Flash Program Memory Code Protection off
CONFIG  CPD = OFF               ; Data Memory Code Protection off
CONFIG  BOREN = ON              ; Brown-out Reset enabled
CONFIG  CLKOUTEN = OFF          ; Clock Out disabled
CONFIG  IESO = ON               ; Internal/External Switchover enabled
CONFIG  FCMEN = ON              ; Fail-Safe Clock Monitor enabled
CONFIG  WRT = OFF               ; Flash Memory Self-Write Protection off
CONFIG  VCAPEN = OFF            ; Voltage Regulator Capacitor disabled
CONFIG  STVREN = ON             ; Stack Overflow/Underflow Reset enabled
CONFIG  BORV = LO               ; Brown-out Reset Voltage trip point low
CONFIG  LPBOR = OFF             ; Low Power Brown-Out Reset disabled
CONFIG  LVP = OFF               ; Low-Voltage Programming disabled
```

The CONFIG directive is used to set some options of the microcontroller. A directive worth mentioning for the rest of the code is:

$FOSC = INTOSC$, which selects the internal oscillator clock source.

These configuration bits will be important later when setting the oscillator clock rate.

5.2 Data memory

```
PSECT udata_shr
ready:                   ; boolean used to know if the timer interrupt has occured
        DS      1
delayc:                  ; The acquisition delay needed after enabling the adc on a channel
        DS      1
ldr0h:                   ; left ldr
        DS      1
ldr0l:
        DS      1
ldr1h:                   ; right ldr
        DS      1
ldr1l:
        DS      1
ldr2h:                   ; lower ldr
        DS      1
```

[5]The code is present in a single .asm file.

```
ldr2l:
        DS      1
ldr3h:                          ; upper ldr
        DS      1
ldr3l:
        DS      1
servoh:                         ; horizontal servo
        DS      1
servov:                         ; vertical servo
        DS      1
temp:                           ; temporary register
        DS      1
counter_l:                      ; interrupt counter (LSB) th slow the system a bit down
        DS      1
counter_h:                      ; interrupt counter (MSB) th slow the system a bit down
        DS      1
```

First, it is worth mentioning the *PSECT* directive as it will appear a few times later in the code. This directive structures the program and data memory.
In the above code, memory is being reserved in RAM. Since only a few variables will be allocated in RAM, which is shared across all RAM banks, we will be able to access those variables from everywhere in the code without using the BANKSEL instruction. The PIC Assembler's DS directive advances the location counter, allowing memory to be allocated to a label defined before the directive, providing a mechanism to reserve memory for variables. The number coming after *DS* is the number of bytes to be allocated for the variable. Comments describing the usage of the variables are mostly self-explanatory.

5.3 Reset vector

```
PSECT reset_vec, class = CODE, delta = 2
reset_vec:
        goto    start
```

The reset_vec situated at address $0000h$ is the start of the code, in other words, where the device goes to when it is reset. As the interrupt vector is situated at address $0004h$, we cannot start our code here, which is why the reset vector is only used to jump over the interrupt vector, straight to the main code.

5.4 Interrupt vector

```
PSECT isr_vec, class = CODE, delta = 2
isr_vec:
        goto    isr

; Interrupt service routine
isr:
        btfss   INTCON, 2
        retfie
        incfsz  counter_l, f
        retfie
        incfsz  counter_h, f
        retfie
        clrf    counter_l
        movlw   0xfd
        movwf   counter_h
        movlw   0x01
        movwf   ready
        bcf     INTCON, 2
        retfie
```

The interrupt vector sends us to the interrupt routine service function *isr*. This ISR is the piece of code that will be executed each time an interrupt occurs, i.e. when timer 0 overflows.
The following instructions will be executed once this function has been invoked:

- Check the cause of the interrupt: the second bit of the INTCON register should be raised if timer 0 has overflown. This is the only case of interrupt we handle in this routine.
- A counter is incremented. The reason for this is that timer 0 overflows too quickly, hence, we artificially lower its overflow rate by executing the sequel of this function only when the two $counters$ overflow.
- The ready flag is raised. This flag is used to trigger the `computation` function (see 5.7) in the event loop.
- Finally, the second bit of the INTCON register is cleared.

5.5 Initialization

```
PSECT code
```

```
start:  call    init_clock              ; 2MHz oscillator initialization
        call    init_adc                ; ADC initialization for LDRs
        call    init_portb              ; PORTB initialization for led
        call    init_data               ; Data initialization
        call    init_timer_interrupt    ; Timer 0 initialization
        call    init_pwm                ; PWM initialization
        clrf    BSR
        goto    loop                    ; enter event loop
```

The *start* label is where the code starts and will initialize the appropriate registers before entering the event loop.

5.5.1 Clock initialization

```
init_clock:
        banksel OSCCON
        movlw   0x60            ; 2MHz HF, FOSC bits in config
        movwf   OSCCON
        return
```

The clock rate has been set to $2MHz$. This value is not random. In fact, this choice comes from two constraints:

1. We need to have a responsive system, so we want to take the highest frequency possible, which is $32MHz$ in the case of the $PIC16F1789$. However, the following constraint requires us to lower this frequency.

2. The servo motors need to have a certain period specified in their data sheet, which is $20ms$. Hence, The highest possible frequency we can set the clock to is $2MHz$. The calculations to arrive to this conclusion will be described later in the report (see 4).

5.5.2 ADC initialization

```
init_adc:
        banksel PORTA   ; PORTA initialization
        clrf    PORTA
        banksel LATA
        clrf    LATA
        banksel TRISA
        clrf    TRISA   ; set RA<0:3> to input
        banksel ANSELA
        movlw   0xff
        movwf   ANSELA  ; set RA<0:3> to analog
        banksel WPUA
        movlw   0x00
        movwf   WPUE    ; weak pull-ups disabled
        banksel ADCON2
        movlw   0x0f
        movwf   ADCON2  ; CHSN: single-ended signal
        movlw   0xf0
        movwf   ADCON1  ; Reference setting + FRC clock (see p.171)
        return
```

After initializing the PORTA and LATA registers, we clear the TRISA register, making the RA<0:3> pins as input, as we want to read the LDR's value from them. We then set these inputs as analog and disable the weak pull-up.
Putting $0x0F$ in the ADCON2 register will make the PIC know the ADC is a single ended ADC converter.
Finally we configure the ADCON1 register to make the VDD be the positive reference and the VSS to be the negative reference. Finally we take the FRC as a clock.

5.5.3 PORTB initialization

```
init_portb:
        banksel PORTB   ; PORTB initialization
        clrf    PORTB
        banksel LATB
        clrf    LATB
        banksel ANSELB
        clrf    ANSELB  ; set RB0 to digital
        banksel TRISB
        clrf    TRISB   ; set RB0 to output
        return
```

Simple PORTB initialization for the debug led.

5.5.4 Data initialization

```
init_data:
        clrf    ready   ; clear ready flag
        movlw   0x2f
        movwf   servoh  ; initialize 360 servo to a centered position
        movlw   0x25
        movwf   servov  ; initialize 180 servo to a centered position
        clrf    counter_l
        movlw   0xfd
        movwf   counter_h       ; initialise interrupt counter
        return
```

This function will put the ready flag to zero, and put the 2 servomotors to a default position. The counter is also initialized.

5.5.5 Interrupt initialization

```
init_timer_interrupt:
        bsf     OPTION_REG, 1
        bsf     OPTION_REG, 2
        bcf     OPTION_REG, 0
        bcf     OPTION_REG, 5
        bsf     INTCON, 5
        bsf     INTCON, 7
        return
```

We set bits 1 and 2 and clear bits 0 and 5 of OPTION_REG. These values configure the timer to be driven by the processor clock, at the rate of one increment every 4 clock cycles, as well as to use a 1:128 prescaler. We also set the interrupt enable bit TMR0IE for this timer, as well as the global interrupt enable bit GIE.

5.5.6 PWM initialization

```
init_pwm:
        banksel PORTC
        clrf    PORTC
        banksel LATC
        clrf    LATC
        banksel TRISC
        movlw   0xff
        movf    TRISC   ; set RC<1:2> to output
        banksel PR2
        movlw   0x9b
        movwf   PR2     ; pwm period (p.228) of 20ms
        ; 360 servo
        ; Put the value of servoh as PWM
        banksel CCP1CON
        bsf     CCP1CON, 2      ; pwm mode
        bsf     CCP1CON, 3
        movf    servoh, 0
        movwf   temp
        lsrf    temp, 1
        lsrf    temp, 0
        movwf   CCPR1L
        bcf     CCP1CON, 4
        bcf     CCP1CON, 5
        movlw   0x03
        andwf   servoh, 0
        movwf   temp
        lslf    temp, 1
        lslf    temp, 1
        lslf    temp, 1
        lslf    temp, 0
        banksel CCP1CON
        iorwf   CCP1CON, 1
        ; 180 servo
        ; Put the value of servov as PWM
        bsf     CCP2CON, 2      ; pwm mode
        bsf     CCP2CON, 3
        movf    servov, 0
        movwf   temp
        lsrf    temp, 1
        lsrf    temp, 0
        movwf   CCPR2L
        bcf     CCP2CON, 4
```

```
        bcf      CCP2CON, 5
        movlw    0x03
        andwf    servov, 0
        movwf    temp
        lslf     temp, 1
        lslf     temp, 1
        lslf     temp, 1
        lslf     temp, 0
        banksel  CCP2CON
        iorwf    CCP2CON, 1
        ; configure and start time 2
        banksel  PIR1      ; timer 2 init and start
        bcf      PIR1, 1
        bsf      T2CON, 0          ; prescaler de 1:64
        bsf      T2CON, 1
        bsf      T2CON, 2          ; Timer 2 on
        banksel  TRISC
        clrf     TRISC
        return
```

This function will initialize the PWM module of the microcontroller, which implies:

- The PORTC pins are initialized: they are then set as output, as we want them to output the value of the PWM to the motors.
- The period it set to an adequate value. This value is computed thanks to the equation 25-1 (page 228 of the datasheet):

$$PWM\_period = (PR2+1) \times 4 \times Tosc \times TMR2\_prescaler \tag{4}$$

$$\Leftrightarrow PR2 = \frac{0.2}{\frac{4}{Fosc} \times TMR2\_prescaler} - 1 \tag{5}$$

  With the $TMR2\_prescaler = 64$ and the $Fosc = 2Mhz$, we get that $PR2 = 0x9B \approx 155$

- Setting the PWM mode by initializing the CCP1CON register.
- Putting the value contained in the servov register as PWM.
- Initializing the timer 2 and start it.
- The same development is also done for the horizontal (360) servomotor (with servoh register).

5.6 Event loop

```
loop:
        btfsc    ready, 0          ; Check ready. If set -> enter, skip instruction otherwise
        call     computation
        goto     loop
```

The programs loop indefinitely while checking the *ready* variable. When it is set (see 5.4), i.e. when the interrupt has happened, the program branches to the `computation` function (5.7).

5.7 LDR value acquisition and servo motor rotation

```
computation:
        clrf     ready
        ; Horizontal (360) servomotor
        call     ldr0      ; Get LDR value (down)
        call     ldr1      ; Get LDR value (up)
        call     differenceH_360 ; Get vertical servomotor's direction
        ; Vertical (180) servomotor
        call     ldr2      ; Get LDR value (left)
        call     ldr3      ; Get LDR value (right)
        call     differenceH_180 ; Get horizontal servomotor's direction
        call     pwm
        return
```

- First, the ready variable is cleared so that when entering the loop again, the next computation happens after the next interrupt generation.
- We get the values from two antagonist LDRs (either left and right, or up and down) using one of the `ldr<0:3>`[6] functions and call the adequate difference function (see 5.8).

[6]This notation has been used in the report code as well, in order for it to be shorter and more readable.

- Once the differences are computed, the servov and servoh registers are modified and pwm function is called, yielding the modification of position of both servomotors.

LDR value acquisition

```
; Compute ldr values
ldr<0:3>:
        banksel ADCON0
        movlw   <0x81, 0x85, 0x89, 0x8d> ; the 4 different channels
        movwf   ADCON0  ; ADC enabled on channel AN0 with 10-bit result
        call    delay
        bsf     ADCON0, 1
        btfsc   ADCON0, 1
        goto    $-1
        banksel ADRESH
        movf    ADRESH, 0
        movwf   ldr<0:3>h
        movf    ADRESL, 0
        movwf   ldr<0:3>l
        bcf     ldr<0:3>l, 0 ; lower the precision in order to avoid noise
        bcf     ldr<0:3>l, 1
        return
```

- First we select the right channel for the reading of the LDRs.
- We call `delay` function to be sure to acquire the value.
- We take the value of the LDR and put it in some variables designed for it.
- Finally we clear the last two bits of the value recorded to lower the precision of the LDRs, in an attempt to smooth out the values and lower the noise.

Acquisition delay

```
delay:
        movlw   0xe9
        movwf   delayc

delay_loop:
        incfsz  delayc, f
        goto    delay_loop
        return
```

The delay function uses a 8 bit variable to count up until overflow, before coming back to normal program flow. The idea is the following:
With a $2MHz$ oscillator, the microprocessor computes an average of $\frac{2^6}{4} = 500000$ cycles per second. A regular operation costs 1 cycle, whereas a branching operation costs 2 cycles, hence, in this delay function, incrementing by 1 the variable costs 3 cycles on average. With this knowledge, we can compute a delay using the following formula:

$$\frac{V \times 3}{500000} = T \Leftrightarrow V = \frac{T \times 500000}{3}$$

where V is the result i.e., the number of times the variable should be incremented and T is the amount of delay we want to insert in our program. e.g. with $T = 5\mu s$ $(0.000005s) \Leftrightarrow V < 1$. Therefore, the delay variable should be incremented only once. We have set the $delayc$ variable to $0xE9$ so that it increments 23 times in order to have some error margin.

5.8 Servo motor rotation

This section covers the PWM creation, as well as the rotation of the servomotors.

```
differenceH_<180, 360>:
        movf    ldr<0, 2>h, 0
        subwf   ldr<1, 3>h
        btfsc   STATUS, 2       ; enter if Z = 1 (equal)
        goto    differenceL_<180, 360> ; Call differenceL_<180, 360> if equal
        btfsc   STATUS, 0       ; enter if ldr<0, 2> > ldr<1, 3>
        goto    turn_left_<180, 360>
        goto    turn_right_<180, 360>
        return

differenceL_<180, 360>:
        movf    ldr<0, 2>l, 0
        subwf   ldr<1, 3>l
        btfsc   STATUS, 2       ; enter if Z = 1 (equal)
        goto    stop
```

```
        btfsc   STATUS, 0        ; enter if ldr0<0, 2> > ldr<1, 3>
        goto    turn_left_<180, 360>
        goto    turn_right_<180, 360>
        return

; Compute horizontal servo new position
turn_left_360:
        banksel PORTB
        movlw   0x00
        movwf   PORTB    ; set led off
        movlw   0x30
        movwf   servoh
        return

turn_right_360:
        banksel PORTB
        movlw   0xff
        movwf   PORTB    ; set led on
        movlw   0x2d
        movwf   servoh
        return

stop:
        banksel PORTB
        movlw   0x00
        movwf   PORTB    ; set led off
        movlw   0x2f
        movwf   servoh
        return

; compute vertical servo new position
turn_left_180:
        movlw   0x30     ; up
        subwf   servov, 0
        btfsc   STATUS, 0
        return
        incf    servov, 1
        return

turn_right_180:
        movlw   0x20     ; down
        movwf   temp
        movf    servov, 0
        subwf   temp, 0
        btfsc   STATUS, 0
        return
        decf    servov, 1
        return

; Move both servos according to new positions calculated
pwm:
        ; 360 servo
        ; Put the value of servoh as PWM
        movf    servoh, 0
        movwf   temp
        lsrf    temp, 1
        lsrf    temp, 0
        banksel CCP1CON
        movwf   CCPR1L
        bcf     CCP1CON, 4
        bcf     CCP1CON, 5
        movlw   0x03
        andwf   servoh, 0
        movwf   temp
        lslf    temp, 1
        lslf    temp, 1
        lslf    temp, 1
        lslf    temp, 0
        iorwf   CCP1CON, 1
        ; 180 servo
        ; Put the value of servov as PWM
        movf    servov, 0
        movwf   temp
        lsrf    temp, 1
        lsrf    temp, 0
        banksel CCP2CON
        movwf   CCPR2L
        bcf     CCP2CON, 4
```

```
        bcf      CCP2CON , 5
        movlw    0x03
        andwf    servov , 0
        movwf    temp
        lslf     temp , 1
        lslf     temp , 1
        lslf     temp , 1
        lslf     temp , 0
        iorwf    CCP2CON , 1
        return
```

The previous code is structured as follows:

- The `difference` functions: these functions will simply make a difference between the 2 adequate registers (horizontal and vertical). As the reading of the LDRs gives us 10 bits of information, we had to divide the task in 2 functions for each axis, one for the upper register, one for the lower. Therefore, we always call `difference_H<180, 360>` which, in turn calls `difference_L<180, 360>` only if the most significant bits don't suffice to determine which LDR gets the most light.
- Based on the result, we call `turn_right<180, 360>`, `turn_left<180, 360>`, or `stop`. These functions will put the suitable value in servoh and servov registers.
- The `pwm` function will create the signal by setting in the `CCPR1L` and the `CCP1CON<4:5>` registers the value computed before contained in the two control registers servoh and servov.

Remark

The two servomotors do not work in the same way. The vertical (180 degrees) one will get a PWM signal that will correspond to a certain angle. Once this angle reached, the motor stops. The horizontal one has a "stoppage value" which is $0x2F$ (see `stop` function). It won't stop turning if the PWM signal it gets is not $0x2F$, meaning that, if the signal is above this value, the motor will turn right, otherwise it will turn left. The farthest the value will be from $0x2F$, the quicker the motor will rotate.

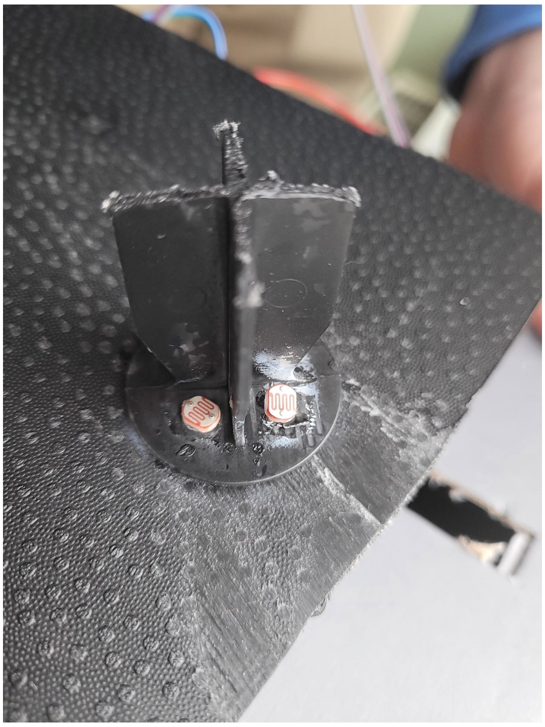
5.9 End of source file

```
        end          reset_vec
```

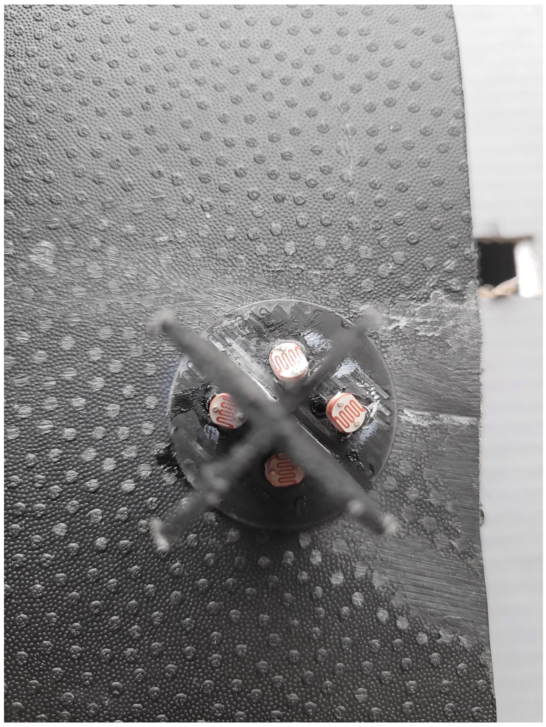
6 Conclusion

This project has been challenging for both the software and the hardware. It took us some time to grasp all the details of the PIC16F1789 but once done, the project could be done in a few man days. The hardware works fine but the different 3D-printed components do not fit together very well, which slowed the mounting process of the system. Also, memory management was quite a big issue, and we had to be well-organized to make this project work fine. Lastly, the different components' behaviours were sometimes difficult to understand. Some were not as precise as we expected, others were too susceptible to noise, etc.
To go further, we could still make this solar tracker more robust and stable, and the cable management could also be done more efficiently. Due to the rotation, the system can sometimes get stuck and some parts can even end being ripped off. Also, the whole circuitry could be printed on a circuit board, which would greatly reduce the clutter generated by the breadboards. Eventually, one could mount a solar panel to the model, which could in turn charge the battery for the system to be fully autonomous. This, however, falls outside the scope of this project.

Appendices



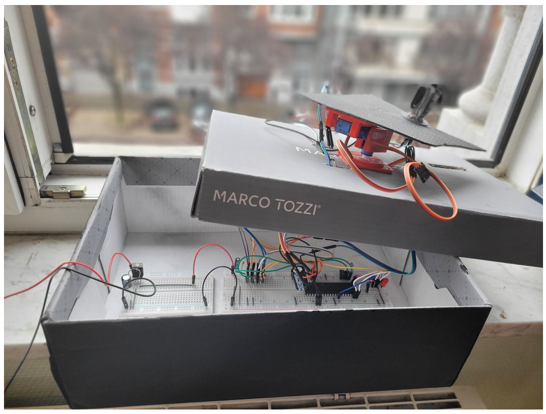
(a) from the side



(b) from above

Figure 4: Quadsensor

(a) Outside



(b) Inside

Figure 5: Full model