UNIVERSITÉ DE LIÈGE

INFO2055-1 Embedded Systems Project

FINAL REPORT
SUNFLOWER SOLAR PANEL

HOORELBEKE Jordi - MEUNIER Loïc - NAA Marco

Academic year 2021-2022

Contents

1 Introduction

To improve a payload's[1] efficiency, one uses a *solar tracker*, which is a device that orients it 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 circuit's 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 highest.

2.1 Rotate function

The first concept is horizontal and vertical rotation. This is achieved by using servomotors that are 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, photoresistors, also called LDRs, are used. An LDR has a internal resistance that depends on the light intensity on the component. By using two LDRs, we can compare the two resistances and adjust accordingly. As we need to rotate on the two axis, two LDRs per axis will be used and arranged to form a quad-sensor.

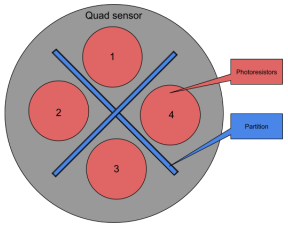

Figure 1: Quadsensor sketch.

[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.

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 Mr. 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.

2.4 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 together with a 9V battery.

3 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.

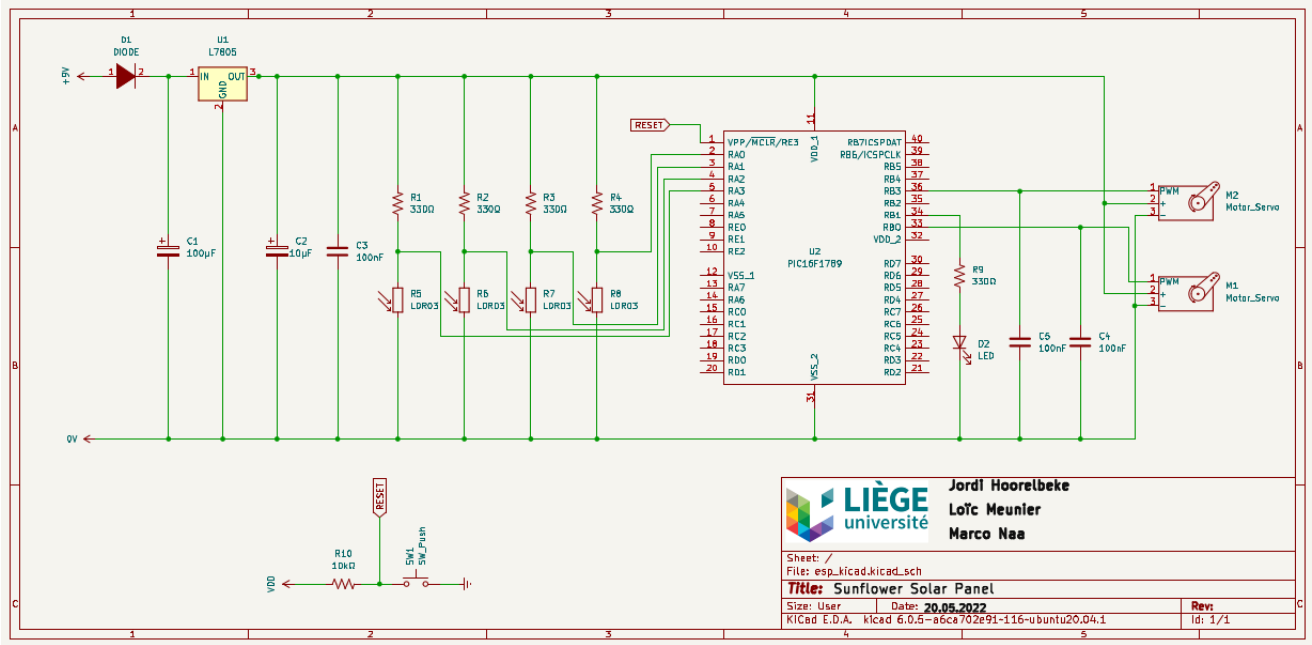

Figure 3: Schematic made on KiCad.

[3]The .plt patterns of these components were designed by Mr. Fernando Bueno and can be found at https://www.thingiverse.com/thing:708819

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

- The system is powered by a $9V$ battery and uses a pic16f1789 as microcontroller.
- $D2$ is a LED simply to inform that the system is still alive.
- The microcontroller requires $5V$ to function correctly, hence, a linear regulator $U1$ is used to put down the initial $9V$ to $5V$.
- The diode $D1$ prevents one to destroy the components of the circuit by connecting the battery in the wrong polarity.
- $C3$ 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.
- All four LDRs are connected to a resistor and to the $RA < 0:3 >$ pins of the microcontroller.
- Both servomotors are connected to the microcontroller through pin $RB0$ and $RB3$ as well as to capacitors $C4$ and $C5$ in order to regulate the current inside of them.
- Finally, $MCLR$ is connected to a pull-up resistor and a switch in order to be able to reset manually the system.

4 In depth review: Software

Now that we have a good understanding on the system's hardware, we must still program the pic16f1789 microcontroller. Therefore, we will now be having a look at the code written in assembly language[4].

4.1 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
```

[4]The code is present in a single .asm file but has been separated in the report into smaller, in order, chunks for pedagogic convenience.

```
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   PLLEN = ON              ; 4x PLL enabled
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. Two directives worth mentioning for the rest of the code are :

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

$PLLEN = ON$, which enables the oscillator module 4xPPL.

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

4.2 Data memory

```
PSECT udata_bank0
ready:               ; semaphore used to know if interrupt has
    occured
        DS      1
left_ldr_value:      ; value of left ldr
        DS      1
right_ldr_value:     ; value of right ldr
        DS      1
lower_ldr_value:     ; value of lower ldr
        DS      1
upper_ldr_value:     ; value of upper ldr
        DS      1
counter_l:           ; last 8 bits of 24-bit counter
        DS      1
counter_h:           ; middle 8 bits of 24-bit counter
        DS      1
counter_hh:          ; first 8 bits of 24-bit counter
        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. As the pic16f1789 microcontroller's RAM is partitioned in 32 banks and as few data will be used, data will be stored in the default bank 0. 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 describe well what the variables in data memory will be used for.

4.3 Reset vector

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

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.
Which leads us to the next section: The interrupt vector.

4.4 Interrupt vector

```
PSECT isr_vec, class = CODE, delta = 2
isr:
        bcf     INTCON, 7 ; disable all interrupts inside isr (GIE)
        bsf     INTCON, 1 ; polling of the interrupt flag bit (INTF
           )
        incf    ready
        bcf     INTCON, 1 ; clear interrupt flag bit (INTF)
        bsf     INTCON, 7 ; enable all interrupts on exit (GIE)
        retfie
```

Upon entering the ISR, the GIE bit of the $INTCON$ register is cleared, disabling all interrupts, and the $INTF$ interrupt flag bit is set. Next, the $ready$ variable is set in order to signal the main program that an interrupt has occurred. Eventually, before resuming the main program, the interrupt flag is cleared to avoid repeated interrupts and all interrupts are enabled again.

4.5 Initialization and event loop

```
PSECT code

start:  banksel OSCCON
        movlw   0xf8    ; PLL enable, 32MHz HF, FOSC bits in config
        movwf   OSCCON
        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 PORTB   ; PORTB initialization
        clrf    PORTB
        banksel LATB
        clrf    LATB
        banksel TRISB
        movlw   0xff
        movwf   TRISB   ; set RB<0;1;3> to output
```

```
        banksel ANSELB
        clrf    ANSELB   ; set RB<0;1;3> to digital I/O
        movlw   0x00     ; clear ready
        movwf   ready
        banksel OPTION_REG
        movlw   0x87
        movwf   OPTION_REG ; set prescale of 1:256 for TIMR0
        banksel INTCON
        movlw   0xA0     ; select Timer0 Overflow Interrupt
        movwf   INTCON   ; enable interrupts
        clrf    BSR
```

Before entering the event loop, some last settings must be done:

1. Defining the oscillator's clock rate by configurating the *OSCCON* register. The *SPLLEN* bit is set to enable 4xPLL, the *IRCF* bits are set to 1111 to use 16 MHz HFINTOSC and the *SCS* bits are cleared in order to use the clock previously determined by the *FOSC* configuration bits.

2. *RAx* and *RBx* pins are initialized as analog inputs and digital outputs respectively.

3. The *ready* variable is cleared.

4. The Timer0 overflow interrupt is selected and enabled with a prescaler of 1 : 256. With this configuration, we get the following formula :

$$T = cycle\ time \times prescaler \times Timer0\ size \Leftrightarrow T = \frac{4}{32^6} \times 256 \times 255$$

   Hence, on average, the ISR will be generated every $8.16ms$, which is more than enough to let time to the microcontroller computing what has to be computed after every interrupt.

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

The programs loop indefinitely while checking the *ready* variable. When it is set, in other words, when the interrupt has happened, the program branches to the *computation* function.

4.6 LDR value computation and servo motor rotation

```
computation:
        clrf    ready
        call    led_blink
        banksel ADCON0
        movlw   0x01     ; select channel AN0
        call    conversion ; left_ldr
        movlw   0x05     ; select channel AN1
        call    conversion ; right_ldr
        movlw   0x09     ; select channel AN2
        call    conversion ; lower_ldr
        movlw   0x0d     ; select channel AN3
```

```
        call    conversion ; upper_ldr
        clrf    ADCON0  ; turn of ADC module
        goto    rotate360
        goto    rotate180
        goto    loop
```

- First, the ready variable is cleared so that when entering the loop again, the next computation happens after the next interrupt generation.
- In addition, the led $D2$ blinks, which will give the user the visual information that the program is still running.
- The ADC module is sequentially used for each LDR by changing to the right channel after each analog-to-digital conversion.
- Once the conversions done, both servomotors change position according to the new LDR values stored in RAM and the program counter returns back to the loop.

```
conversion:
        movwf   ADCON0  ; turn on ADC module
        call    delay   ; Acquisition delay
        bsf     ADCON0, 1 ; start conversion (ADGO)
        btfsc   ADCON0, 1 ; is conversion done? (ADGO)
        goto    $-1     ; no, test again
        ; <"Store result from ADRESH and ADRESL">
        return
```

- First, the ADC module is enabled on the right channel with the value present in the working directory.
- For the ADC to meet its specified accuracy, the charge holding capacitor (CHOLD) must be fully charged to the input channel voltage level. Therefore, a delay function is used for the program to wait the appropriate amount of time, which is $5\mu s$ according to the datatsheet (p.182).
- Ones the ADC module ready to operate, the conversion starts and the program loops until it finishes.
- Eventually, the appropriate variable is set to the result stored in $ADRESH$ and $ADRESL$ registers. ⚠**Unfortunately, the implementation is still in progress.** ⚠

```
rotate360:
    ; <"rotate the 360 servo motor">
        goto    loop

rotate180:
    ; <"rotate the 180 servo motor">
        goto    loop
```

⚠**Unfortunately, the implementation is still in progress.** ⚠

```
led_blink:
        banksel PORTB
        btfsc   PORTB, 0
        goto    turn_led_off ; if RB0 == 1
        goto    turn_led_on ; elsif RB0 == 0
led_blink_end:
        return
turn_led_off:
        bcf     PORTB, 0 ; clear RB0
        goto    led_blink_end
turn_led_on:
        bsf     PORTB, 0 ; set RB0
        goto    led_blink_end
```

If $RB0$ is set, unset it; if $RB0$ is clear, set it.

```
delay:  movlw   0xff
        movwf   counter_hh
        movlw   0xcb
        movwf   counter_h
        clrf    counter_l
delay_loop:
        incfsz  counter_l, f
        goto    delay_loop
        incfsz  counter_h, f
        goto    delay_loop
        incfsz  counter_hh, f
        goto    delay_loop
        return
```

The delay function uses a 24 bit variable (3×8 bits) to count up until overflow before coming back to normal program flow. The idea is as follows :
With a $32MHz$ oscillator, the microprocessor computes an average of $\frac{32^6}{4} = 8^6$ 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:

$$\frac{V \times 3}{8^6} = T \Leftrightarrow V = \frac{T \times 8^6}{3}$$

where V is the result aka 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 $T = 5\mu s$ $(0.005s)$ $\Leftrightarrow V \approx 13333$. Therefore, $counter_{hh}$ is $0xff$ (unused), $counter_l$ is $0x00$ and $counter_h$ is $0d256 - 0d53 = 0xCB$ because $256 \times 53 = 13568$ which is the closest integer possible above 13333.

4.7 End of source file

```
        end         reset_vec
```

5 Summary

Hardware wise, the electronic circuit is complete, while, of course, open for some potential changes until the project presentation if necessary.
Software wise, however, we have not managed to convert the voltages passing through the LDRs and use the results for orienting the servomotors yet.
The objective until the project presentation is to have a well working system together with a complete code.