RMIT University

School of Electrical and Computer Engineering
EEET2256- Embedded Systems

FINAL REPORT
Sun Tracker: Sun Tracking System

Student Name: Ishan Jagaty **Student Number**: s3489519
Student Name: Dale Giancono **Student Number**: s3422566
Student Name: Vijayaselvam Panneerselvam **Student Number**: s3486412
Tutor: Lincy Jim

Group: Wednesday and Friday 9:30pm-11:30pm
Submission Due Date: Friday 9th October 2015.

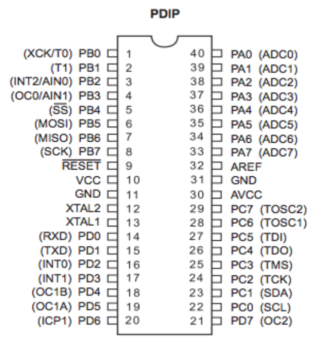
Introduction

The extraction of electricity from the sun was possible due to the discovery of the photoelectric effect. The Sun Tracker is a simple device that orients or aligns various payloads towards the sun. The payloads can be photovoltaic panels, reflector, collectors, lenses or other optical devices. The system focuses on the optimizing the electric energy by tracking the light emitted from the sun. The sun changes it's angle from 0-90 degrees rising and a 90-180 degree decline from dawn to dusk. Due to the fluctuations factors described previously, a stationary sun tracker will not provide higher efficiency. Whereas a solar tracker will keep its orientation towards the sun at the optimum possible angle. A single axis tracking system will improve efficiency by 30% and a dual axis will increase the efficiency by another additional 6%. The Sun Tracker is a simple device that requires little to no user interface. X and Y axis motors and will automatically adjust its panel facing direction to face the sun or other light sources. This process will be entirely automated, with no user intervention required besides powering on the unit. It will compare values generated from partially obscured LDR's to determine where a light source is coming from. Once it has determined where a light source is originating from, it will automatically adjust the panel facing position to face the light source. An Atmel AMTEGA32 microcontroller will output TTL values in to a SN754410 H-bridge motor driver, which will then driver the x and y axis motors. The microcontroller is programmed to orient the panel at optimum position against the sun by comparing the inputs. Once the unit is powered on, software will compute differences in the left and right side LDR values. It will then compute the differences in the up and down side LDR values. Once these differences have been computed we can ascertain whether or not the panel is facing the correct direction. If it is not, x and y axis motor will move for a fraction of a second before this entire process is repeated. This will ensure a high degree of accuracy.

Components Required

For the project, the following components will be required.

Figure 1 ATMEGA 32

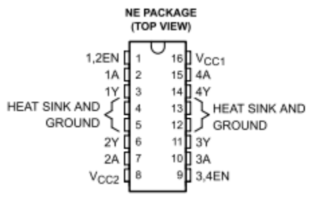


ATMEGA32 Processor

The ATMEGA32 (Figure 1) is a high performance, low power 8-bit microcontroller that is designed and manufactured by Atmel. It has a very advanced RISC architecture that can execute 131 powerful instructions in a single clock cycle. It has 32 x 8 general purpose working registers. It has high endurance and non volatile memory segments that provide us with 32Kbytes of in-system self programmable flash memory. IT has 32 programmable I/O lines and it operates at around 4.5-5.5V. The VCC pin on the ATMEGA provides the chip with a digital supply voltage and the GND pin connects it to ground. Port A (PA0…PA7) serves as the analogue inputs from the LDR circuit. Port C (PC3…PC7) is an 8-bit directional port with internal pull up resistors that are connected to the H Bridge Motor Driver. These pins provide inputs and outputs to the motor driver chip that drives the motor.

SN754410 H-Bridge Motor Driver

Figure 2 H-Bridge Motor Driver



The SN754410 is a quadruple high-current half-H driver designed to provide bidirectional drive currents up to 1A at voltages from 4.5-36V.This device is designed to drive inductive loads such as DC Motor Drivers, Stepper Motor Drivers and Latching Relay Drivers. It is designed for Positive-supply applications and has 3-state outputs. When an enable input is high, the associated drivers are enabled, their respective outputs become active and become in phase with their inputs. When the enable input is low, these drivers are disabled and their outputs are off and move to a high-impedance state. With the correct data inputs each pair of drivers form a Full-H reversible drive suitable for motor applications. The pins 1,2EN and 3,4EN enable the driver channels from the ATMEGA32 and act as active high inputs. Pins <1:4>A act as non inverting driver inputs and the pins <1:4>Y act as driver outputs. The pin Vcc_1 supplies a regulated 5V for the internal logic translation and the pin Vcc_2 supplies power to the motor drivers.

LDR Circuit

Figure 3 LDR

An LDR (Figure 3) is a light controlled variable resistor that exhibits photoconductivity. It's resistance increases with decreasing light intensity and it can be applied in a light sensitive detector circuit. The resistance of the LDR's used in our circuit vary from 2.8-8.4kΩ when illuminated and in a dark environment they can be up to 500kΩ.These LDR's are soldered into a chip that detect the light and provide inputs to the Port A pins of the ATMEGA32.

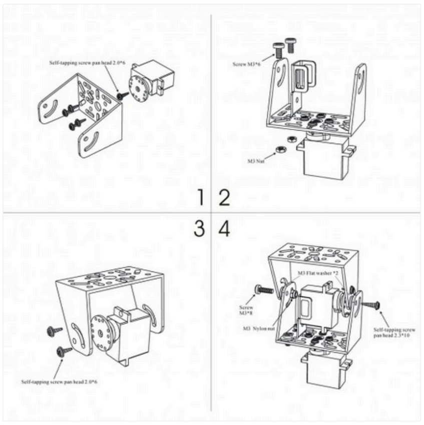


Figure 4 - Dagu Mini Pan with Tilt Kit

Dagu Pan and Tilt Kit with Mini Servos

The Dagu Mini Pan and the Tilt Kit (Figure 4) is the tilt and roll system that will be used in the project to move the solar panels and will be connected to the motor and the ATMEGA32 microprocessor. It includes brackets and miniature servos with nuts and bolts to hold the motor in one place and adjust the panels accordingly.

VM-Lab and EagleCAD

VM-Lab is a microcontroller design tool that is used to develop the software for the project. It is one of the best AVR simulator with multiple integrated tools and in line error reports. EagleCAD is a PCB design tool that was used to simulate PCB circuits.

Veroboard

A Veroboard is a brand of strip board with copper stripes on an insulating board. It is used for numerous types of prototyping. In our project it will be used to develop the circuit for the motor and connect the microcontrollers.

Open USB-IO

The open USB-IO board (figure 5) is a powerful microcontroller that includes a built in ATMEGA32 microprocessor. It can program the ATMEGA32 microprocessor using a USB boot loader and can communicate with a PC very easily. The ATMEGA32 can be programmed using the user's own code or a standard interface code. It has multiple input switched and LED's. The USB port of the open USB-IO provides us with a +5V power supply to power the motor drivers and the LDR's.

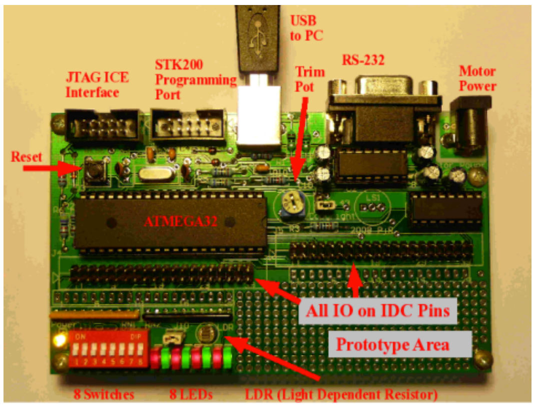


Figure 5- Open USB-IO

Hardware Block Diagram and Schematics

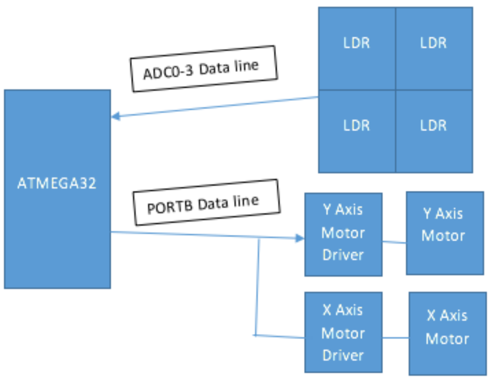


Figure 6 - Hardware Schematics Drawings

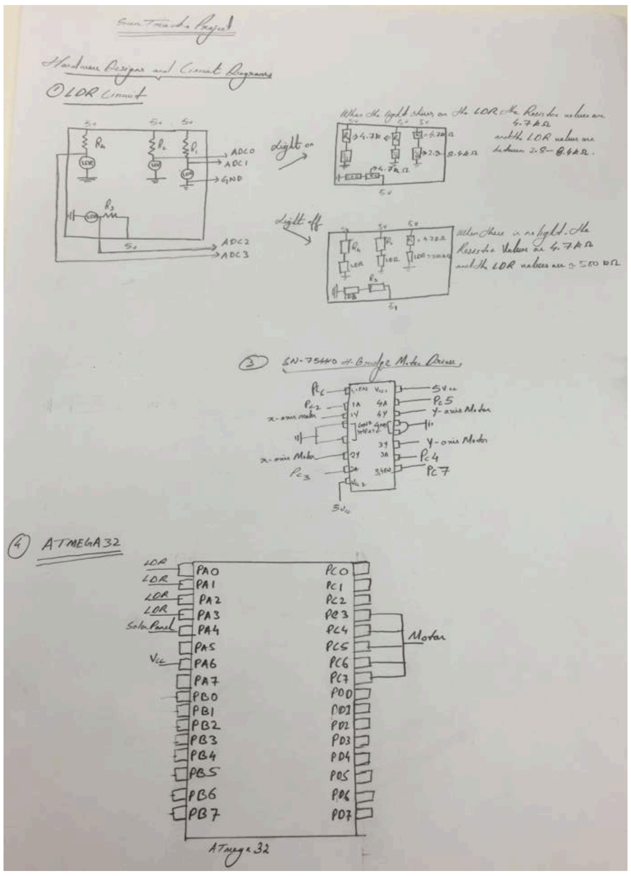
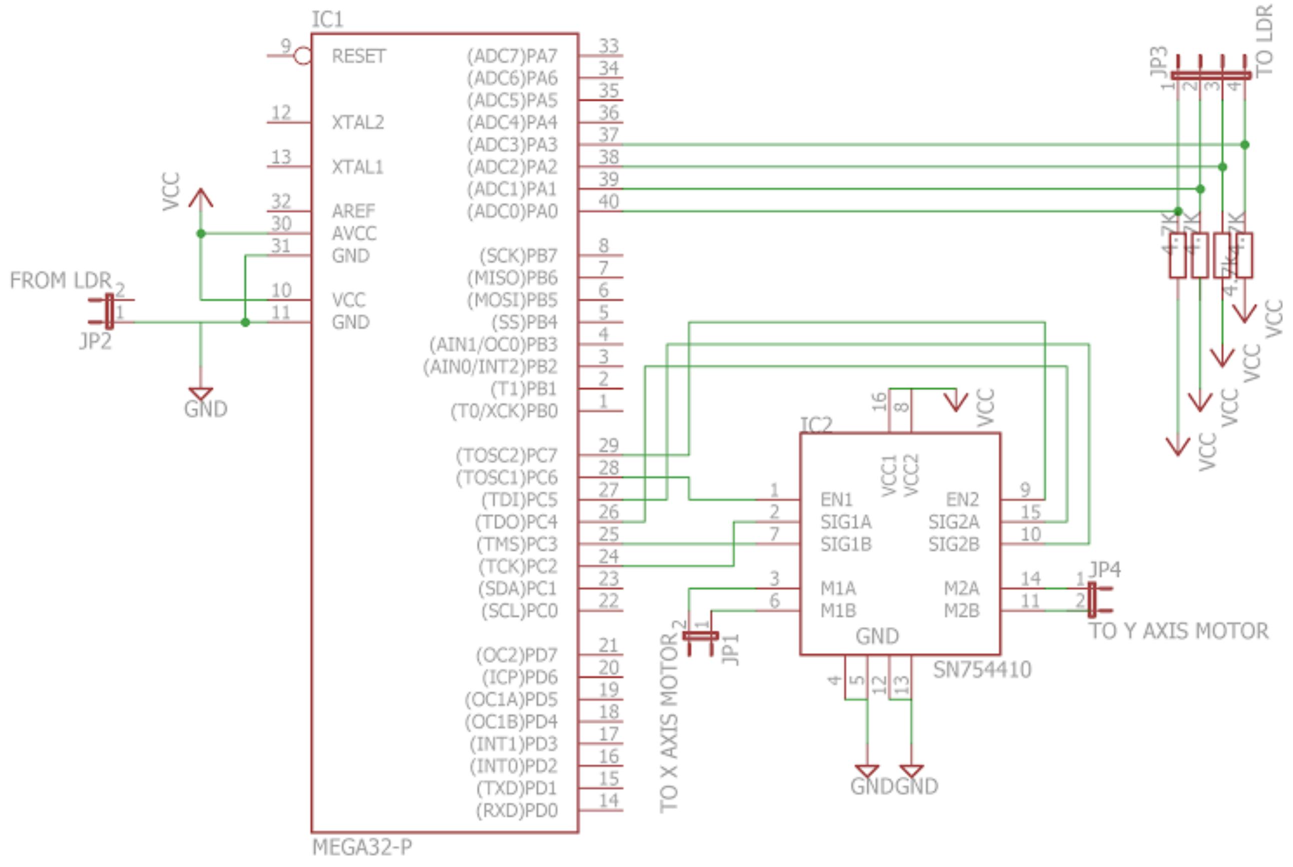
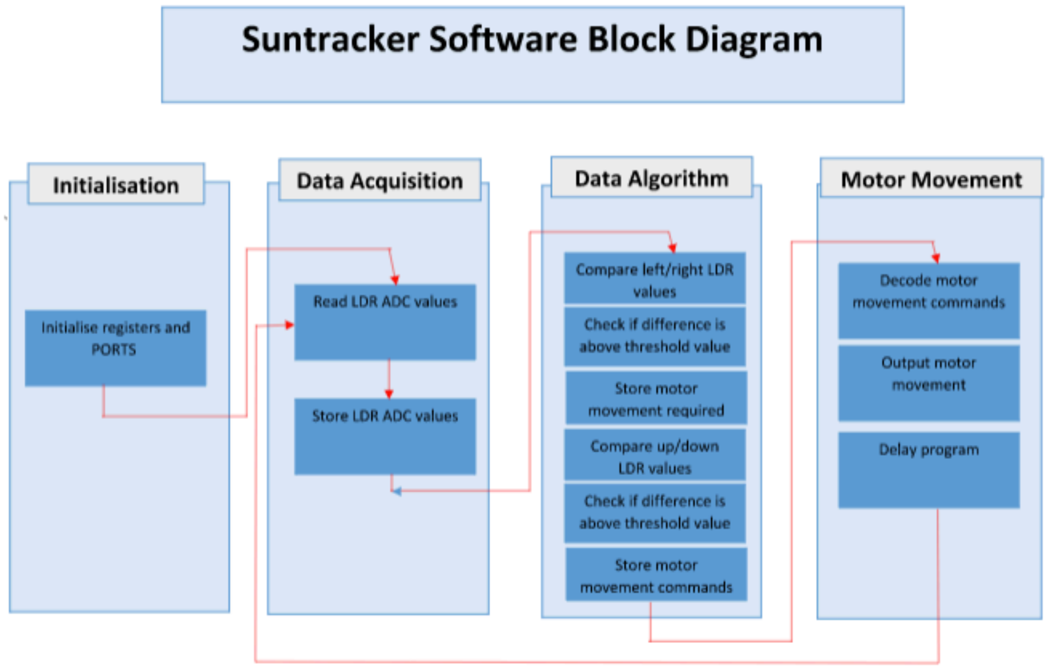


Figure 7- EagleCAD Simulation



Software Block Diagram


Suntracker Software Block Diagram
Initialisation
Initialise registers and PORTS
Data Acquisition
Read LDR ADC values
Store LDR ADC values
Data Algorithm
Compare left/right LDR values
Check if difference is above threshold value
Store motor movement required
Compare up/down LDR values
Check if difference is above threshold value
Store motor movement commands
Motor Movement
Decode motor movement commands
Output motor movement
Delay program

Software Code Structure Description

The software component for the Sun tracker is arguably the most intricate and important component of the project. It can be separated in to four components. Initialization, Data Acquisition, Data Algorithm and Motor Output. Above is a block diagram for the software component of the sun tracker project.

Initialisation

This phase is only called once at the start up of the device and is not included in the main program loop. The following label entitled "Initialise" contains all the assembler commands used for the initialisation of the device.

```
Initialise:
            LDI temp, $FF
            OUT DDRC, temp          ; Sets all PORTC pins as outputs.
            LDI temp, $00
            OUT DDRA, temp                          ; Sets all PORTA pins as inputs.
            OUT PORTC, temp         ; Clears all PORTC pins, turning them low.
```

The above assembler code sets the DDRx registers for the project and ensure that all PORT pins start low. It sets PORTC pins as outputs. These pins will be used to issue commands the hardware motor driver. PORTA pins are configured as inputs as we will be using these pins as ADC's

Data Acquisition:

After the initialisation is complete the main program loop begins. This loop always starts with the Data Acquisition phase. In this phase, Voltage values are converted to digital numbers with the ATMEGA32s multiplexed input ADC. We convert and store these LDR output voltage values one by one before moving on to the data algorithm phase. To achieve this, we use three separate labels in the Data Acquisition phase which are essentially repeated throughout the phase.

data_acquire:

```
        LDI temp, $60           ; Move ADMUX adc0 value to temp register.
        CALL adc_start          ; Starts the ADC conversion proccess.
        CALL convert_check      ; Checks if conversion is complete.
        CALL adc0_store         ; Stores conversion result in adc0 register.
```

The above assembler code shifts a value in to the temp register before calling the three labels that make up the data acquisition phase. Adc_start, convert_check and adc0_store are these three labels. I will now describe the operation of these three labels.

adc_start:

```
    OUT ADMUX, temp     ; Selects top reference voltage as AVCC, left adjusted result, and ADC channel.
    LDI temp, $90
    OUT ADCSR, temp     ; Enables ADC, disables auto trigger, disables interupt enable, factor 2 prescaler.
    SBI ADCSR, ADSC     ; Sets start conversion bit. Starts the ADC conversion.
    RJMP exit           ; Returns to data_ acquire label.
```

The adc_start label updates the ADMUX register to use AVCC as the reference voltage. It also left adjusts the result as we only intend to use 8 bits out of the 10 bit result. In this case we are also choosing ADC0 as the input we wish to use. Once we do this, we move $90 in to the ADSCR register as to enable the ADC, disable all auto triggers and interrupt enables and uses a prescaler division factor of two. Once that is completed, we then use the SBI instruction to start the ADC conversion. We then return to the data_acquire label and move on to the convert_check label.

convert_check:

```
    SBIC ADCSR, ADSC    ; Skips the next instruction with the ADSC bit in ADCSR is still set.
                        ; This bit clears once conversion is complete.
    RJMP convert_check  ; If conversion is still going, repeat convert_check.
    RJMP exit           ; If conversion is over, exit to data_acquire label.
```

Before we store the ADC value, we must first wait for the conversion to be complete. This takes quite a few clock cycles. So in order to check that if it is complete, we wait for the ADSC bit in the ADCSR register to automatically clear itself. We achieve this by using the SBIC instruction. The SBIC instruction will check if the start conversion bit is set. If it is, the next instruction will be executed. If it is cleared, the next instruction will be skipped. So in this case, If the start conversion bit is high, the convert_check able will repeat itself, if it is low, the program will exit from the convert check label and return to the data_acquire label before moving on to the ADC storing label.

adc0_store:

```
    IN adc0, ADCH       ; Moves ADC0 data to adc0 register.
    RJMP exit           ; Returns to data_acquire label.
```

The adc0_store label simply reads the ADCH value and stores it in a dedicated register.
The process which consists of these three labels repeats for each ADC we require to use. Once each ADC value is read and stored, we can move on to the Data Algorithm stage.

Data Algorithm:

Once we have acquired all of our digitised LDR output values, we can now process the data and come up with a direction we need our motors to move, if they need to move at all. To do this, we need to compare the left and right LDR values, as well as the top and bottom LDR values. We can from this information ascertain whether or not the tracker is facing the sun, and which direction the tracker needs to move in order to face the sun. We also have to incorporate a cut off threshold in to the software. If this is not in place minute changes in LDR outputs that do not necessary have to do with sun movement will result in the movement of the tracker. This could cause “hunting” or oscillations, where the tracker is constantly moving in order to compensate for these tiny variations in output. The below assembler code represents the Data Algorithm stage.

data_algorithm:

```
    CLR x_count
    CLR y_count
    CALL x_axis         ; Calls x_axis data algorithm label.
    CALL y_axis         ; Calls y_axis data algorithm label.
    RJMP exit           ; Returns to main label.
```

x_axis:

```
    CP adc0, adc1       ; Compares left and right LDR values.
    BRSH left_threshold ; If left value is greater than right value, the x axis must move in the left direction.
                        ; Branch to left_threshold label.

    CALL right_threshold ; Else if rigt value is greater than left value, the x axis must move in the right direction.
                        ; Branch to right_threshold label.
    RJMP exit           ; Returns to data_algorithm labl.
```

```
y_axis:
        CP adc2, adc3           ; Compares top and bottom LDR values.
        BRSH up_threshold       ; If top value is greater than bottom value, the y axis must move in the up direction.
                                ; Branch to up_threshold label.
        CALL down_threshold     ; If bottom value is greater than top value, the y axis must move in the down direction.
                                ; Branch to down_threshold label.
        RJMP exit               ; Returns to data_algorithm labl.

left_threshold:
        SUB adc0, adc1          ; If left value was greater than right value, the x axis must move in the left direction.
                                ; But first we must make sure that the difference between the two LDRs is greater than our threshold.
                                ; To do this we first subtract the adc0 value from adc1.
        CPI adc0, threshold     ; We then compare the result with the defined threshold number as specified at the top of the program.
        BRLO exit               ; If the result was less than the threshold number, the x axis is not required to move.
                                ; So we branch to the exit label, and in return, return to the data_algorithm label.
        LDI x_count, $1         ; If the result was higher than the threshold number, the x axis is required to move left.
                                ; To tell the motor this is the case, we make the x_count register equal $1. We will decode this value later.
        RJMP exit               ; Returns to x_axis label.

right_threshold:
        SUB adc1, adc0          ; If right value was greater than left value, the x axis must move in the right direction.
                                ; But first we must make sure that the difference between the two LDRs is greater than our threshold.
                                ; To do this we first subtract the adc1 value from adc0.
        CPI adc1, threshold     ; We then compare the result with the defined threshold number as specified at the top of the program.
        BRLO exit               ; If the result was less than the threshold number, the x axis is not required to move.
                                ; So we branch to the exit label, and in return, return to the data_algorithm label.
        LDI x_count, $2         ; If the result was higher than the threshold number, the x axis is required to move right.
                                ; To tell the motor this is the case, we make the x_count register equal $2. We will decode this value later.

        RJMP exit               ; Returns to x_axis label.
up_threshold:
        SUB adc2, adc3          ; If top value was greater than right value, the x axis must move in the left direction.
                                ; But first we must make sure that the difference between the two LDRs is greater than our threshold.
                                ; To do this we first subtract the adc2 value from adc3.
        CPI adc2, threshold     ; We then compare the result with the defined threshold number as specified at the top of the program.
        BRLO exit               ; If the result was less than the threshold number, the y axis is not required to move.
                                ; So we branch to the exit label, and in return, return to the data_algorithm label.
        LDI y_count, $10        ; To tell the motor this is the case, we make the y_count register equal $1. We will decode this value later.
        RJMP exit               ; Exits to y_axis label.

down_threshold:
        SUB adc3, adc2          ; If bottom value was greater than the top value, the x axis must move in the bottom direction.
                                ; But first we must make sure that the difference between the two LDRs is greater than our threshold.
                                ; To do this we first subtract the adc3 value from adc2.
        CPI adc3, threshold     ; We then compare the result with the defined threshold number as specified at the top of the program.
        BRLO exit               ; If the result was less than the threshold number, the y axis is not required to move.
                                ; So we branch to the exit label, and in return, return to the data_algorithm label.
        LDI y_count, $20        ; If the result was higher than the threshold number, the y axis is required to move down.
                                ; To tell the motor this is the case, we make the y_count register equal $20. We will decode this value later.

        RJMP exit               ; Exits to y_axis label.

exit:       ; Returns to main label.
        ret
```

To start off the Data Algorithm stage, we compare the ADC0 and ADC1 values. These values represent the left and right LDR outputs respectively. If ADC0 is a lesser value than ADC1, we know the motor needs to move left. If ADC1 is greater than ADC0 then the motor needs to move right. The left_threshold and right_threshold labels test if the difference between the two ADC values is great than a values stored in the threshold register. If they are greater, a register called the x_count register has a number moved in to it. In this case $0 represent no motor movement. $1 represents movement to the left. $2 represents movement to the right. The same process is completed for ADC3 and ADC4 excepted the end values are stored in the y_count register. $00 represent no motor movement. $10 represents movement upwards and $20 represents movement downward. Once all these processes are completed we exit out of the data_algorithm label and return to the main program loop where we will then enter in to the motor movement phase.

Motor Movement:

The final phase of the program is motor movement. Before we send commands to the motor driver we must first decode the motor movement values stored in the Data Algorithm phase.

```
motor_movement:


        CALL decode                     ; Calls decode label.
        CALL output                     ; Calls output label.
  RJMP exit                     ; Returns to main label.
```

```
decode:

            LDI ZH, high(motor<<1)  ; Set the base pointer to the table.
            LDI ZL, low(motor<<1)
            OR x_count, y_count     ; ORs x_count and y_count. y_count will be stored in the 4 MSBs, x_count in the 4 LSBs.
            ADD ZL, x_count                     ; Offsets address with temp value
            LPM temp, Z                         ; Moves the decoded data value to the temp register. This value will be outputed to the motors.
            RJMP exit                           ; Return to motor_movement label.

output:
            OUT PORTC, temp                     ; Output temp value retrieved from decode label to PORTB
            CALL delay_on                       ; Call the delay_on label. This delay will specify how long the on
                                                ; cycle of the PWM motor signal will be.
            LDI temp, $C0
            OUT PORTC, temp                     ; Output temp value $0. This will turn all motors off
            CALL delay_off                      ; Call the delay_off label. This delay will specify how long the off
                                                ; cycle of the PWM motor signal will be.
            RJMP exit                           ; Returns to motor_movement label.

delay_on:

            LDI temp, 0xFF          ; Sets couter inner loop value at 255.
            LDI delay_count, motor_speed
            RJMP delay_loop         ; Jumps to loop count down.

delay_off:

            LDI temp, 0xFF          ; Sets couter loop value at 255.
            LDI delay_count, $8
            RJMP delay_loop                     ; Jumps to loop count down.

delay:
            LDI temp, 0xFF          ; Sets couter loop value at 255.
            LDI delay_count, $FF
            RJMP delay_loop                     ; Jumps to loop count down.

delay_loop:

            DEC temp                            ; Decrements couter value by one.
            CPI temp, $00           ; Checks if counter value is 0.
            BREQ delay_count                    ; If it is we exit the delay loop.
            RJMP delay_loop                     ; If it isn't we repeat the process.

delay_count:

            DEC delay_count                     ; Decrements couter value by one.
            CPI delay_count, $00                ; Checks if counter value is 0.
            BREQ exit                           ; If it is we exit the delay loop.
            RJMP delay_loop                     ; If it isn't we repeat the process.


;PORTB OUTPUT FOR MOTOR DRIVER

; 0b11000000 ($0) = no movement
; 0b11000100 ($1)($C4)= move left
; 0b11001000 ($2)($C8)= move right
; 0b11010000 ($4)($D0)= move up
; 0b11100000 ($8)($E0)= move down
; 0b11010100 ($5)($D4)= move up and left
; 0b11011000 ($6)($D8)= move up and right
; 0b11100100 ($9)($E4)= move down and left
; 0b11101000 ($A)($E8)= move down and right

motor:
            ;************** x_count **************************************************
            ;x0  x1  x2  x3  x4  x5  x6  x7  x8  x9  xA  xB  xC  xD  xE  xF              *
.db $C0, $C4, $C8, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0 ;0x         y_count
.db $D0, $D4, $D8, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0 ;1x                  *
.db $E0, $E4, $E8, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0, $C0 ;2x                  *
```

To decode the motor movement values, we must first OR x_count and y_count. Doing this will give us 9 possible combinations of values. Each value represents a type of movement. This includes no movement ($00), move left ($01), move right ($02), move up ($10), move down ($20), move up and left ($11), move up and right ($12), move down and left ($21) and move down and right ($22). To turn these values in to something we can control a motor driver with we must make a database with out desired values and outputs. The above code shows this database and each corresponding PORT output. You may also note that the two most significant bits always remain high. This is because they are connected to enable inputs on the motor driver. The two least significant bits and the pins they represent are never connected to anything and therefore always low.

Once we have decoded our value, we generate a type of pseudo pulse width modulation. To do this we create to delays of differing lengths. One delay will be our “on” delay. This will specify how long the on time of our motor output will be. The other delay is our “off” delay. This delay will specify how long the off time of our motor will be. The on delay time can be adjusted by altering the value of the “motor_speed” register. Altering this value will affect the speed in which the motor moves.

Once one cycle of this PWM has been completed, the whole main program will begin to loop from the beginning. Such small bursts of motor movement enable the device to make fine adjustments. It also ensures that the device does not have much inertia while moving. If the inertia generated by movement is too great, the device may begin to “hunt” or oscillate between altering positions. For a copy of the complete code, please refer to the appendix.

Complications

The project was completed fairly smoothly however along the way there were some complications that prevented the system from working optimally. Initially when prototyping and testing the servo a breadboard, y-axis motor would respond with strange movements, later this was found out to be due to an inbuilt driver chip already in the motors which interfered with the SN754410 and after removing worked perfectly.

The initial range of movement the x and y axis motors were limited to 180 degrees and 90 degrees respectively to improve this the blocks preventing the full movement were removed. After the physical prototype was built the motors would barely move in response to light. This was due to the duty cycle being far too small. After increasing the duty cycle the motors were moving far too quickly to get accurate readings and would never find the optimal light source and after much testing a balanced cycle allowed the motors to move at an acceptable rate. After more testing the motor movement was found to be quite jittery, this was due to a delay introduced into the software that allowed the motors to “settle” before taking another reading but this period of this delay was too large and reducing the delay smoothed the movement.

Another issue the motors were having was getting caught in a cycle where they would simply oscillate between two spots and unable to face the optimal area. This was due to the time it took for the LDRs to send back the data being too small and increasing the time minimized the chance of this problem from occurring again.

Testing Process

For the testing process, we broke each software module down and tested them separately before bringing them all together. First, we tested the motor output section of hour software. To do this, we soldered the SN75440 motor driver IC to some spare holes on the OUSB board and wired as some header connectors to the IC. This enabled us to easily interface the driver chip with the ATMEGA32 via jumper leads. Once wired up as per the schematic we ran the following simple test code which moved the motor in every possible direction. Left, right, up, down, up and left, up and right, down and left and down and right.

Once working, we tested the data acquisition stage. We did this by wiring up our LED board to the ADC pins on the ATMEGA32, then directly outputting the result through PORTB pins on the OUSB board. These pins are already hardwired to the 8 LEDs also included on the board which worked almost immediately.

We then wrote the Data algorithm module and added all the code together as this was the only way to truly see if the data algorithm module was working correctly.

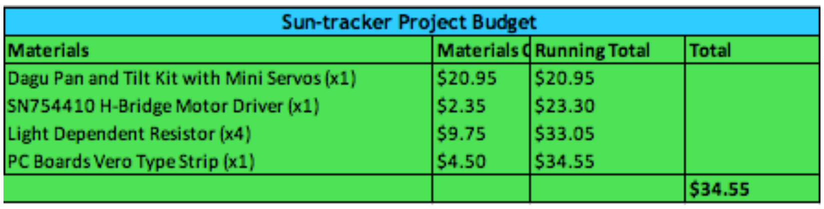
Finally, we had to calibrate the device. To calibrate the device, we need to adjust two parameters which are stored in separate registers in the ATMEGA32 assembler code. These parameters control the threshold of difference and the motor speed of movement. By altering the threshold parameter, we adjust the value of tolerable difference in LDR output Values before the Sun Tracker moves to correct the difference.

In the end, we found a value of 8 ensures the Sun Tracker is not always “hunting” for the perfect position, but is still sensitive enough react to small changes in sunlight. The motor speed parameter is also important as if it is too high, oscillations can occur.

This is due to the fact that the motor can move before far before acquiring new values from the LDRs, and therefor move past the optimum sun facing position. It will then try and correct itself and the process will repeat causing a kind of feedback loop. However, if we make the motor speed too low the motors will struggle to move the weight placed on them. We found the optimum value for the motor speed register to be $7F.

After the calibration was complete the testing phase finished as we had a working project.

Budget Allocation



Sun-tracker Project Budget			
Materials	Materials C	Running Total	Total
Dagu Pan and Tilt Kit with Mini Servos (x1)	$20.95	$20.95	
SN754410 H-Bridge Motor Driver (x1)	$2.35	$23.30	
Light Dependent Resistor (x4)	$9.75	$33.05	
PC Boards Vero Type Strip (x1)	$4.50	$34.55	
			$34.55

Conclusion

In this report a sun tracking system has been presented. The key feature of the system is its ability to move 360 degrees on the x-plane (horizontally) and 180 degrees on the y-plane (vertically). This feature can be used to greatly improve the efficiency of standard stationary solar panels. Several advantages of the system include, virtually an automated process with little human interaction, efficient energy usage as movement is only carried out in required situations (mostly during sunrise and sunset), eliminating unnecessary energy use. The removal of unnecessary movements, at very small light intensity differences by implementing a threshold and maximising output energy that can be produced by optimal positioning of the system frequently adjusting to find the greatest light source. When applying this model on a large scale some changes would need to be made. For the movement of the solar panels an actuator would be a much better option as actuators can be programmed to have finer movements with more accuracy and they are virtually silent as opposed to the servo, which makes the actuator ideal for household use. The solar sun tracking system is an essential addition to stationary solar panels. Though this system greatly improves solar panel efficiency there are still a few disadvantages that cannot be mitigated such as, very little use during winter where most days are overcast or raining where very little light is absorbed and the fact that solar panels alone are still quite expensive and implementing actuator movement further increases that price making it not as viable for consumers, however businesses may still have the funds to introduce this system.

References

[1] Pj Radcliffe, 'Open-USB-IO', 2009. [Online]. Available: https://pjradcliffe.wordpress.com/open-usb-io/. [Accessed: 24- Sep- 2015].
[2] Technologystudent.com, 'Light Dependent Resistors', 2015. [Online]. Available: http://www.technologystudent.com/elec1/ldr1.htm. [Accessed: 28- Sep- 2015].
[3]'Sn754410 Datasheet', 2015. [Online]. Available: http://www.ti.com/lit/ds/symlink/sn754410.pdf. [Accessed: 26- Sep- 2015].
[4]'ATMEGA32 Datasheet', 2015. [Online]. Available: http://www.atmel.com/Images/2503s.pdf. [Accessed: 27- Sep- 2015].
[5] Rpc.com.au, 'Solar Tracking in Australia | Pros and Cons', 2015. [Online]. Available: http://www.rpc.com.au/information/faq/solar-power/trackers.html. [Accessed: 24- Sep- 2015].
[6] Solarchoice.net.au, 'Solar tracker performance and economics in Australia - Solar Choice', 2015. [Online]. Available: http://www.solarchoice.net.au/blog/solar-trackers/. [Accessed: 23- Sep- 2015].
[7]Y. Rambhowan and V. Oree, 'Improving the dual-axis solar tracking system efficiency via drive power consumption optimization', Appl. Sol. Energy, vol. 50, no. 2, pp. 74-80, 2014.