NEW YORK CITY COLLEGE OF TECHNOLOGY

THE CITY UNIVERSITY OF NEW YORK

Department of Computer Engineering Technology

Microcomputers Systems Technology

CET 3510 Section: E360

1K Hackaday Group Report

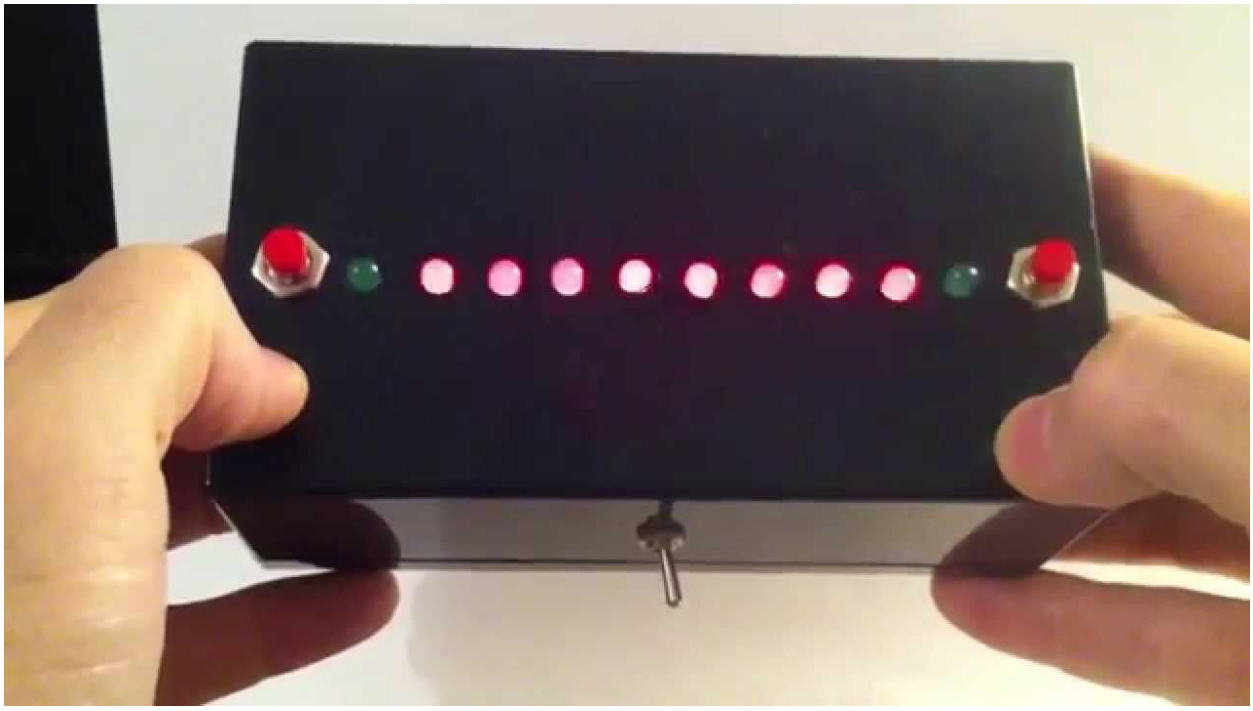


Table of contents

Introduction

2-dimensional game of Pong, which will allow us to apply the skills we have learned coding in assembly in different environments such as windows and Linux.The 2-dimensional pong is a classic game that requires the player to click one of the two buttons at the right time while the LEDs that are placed in series are used as a timer. The hardware of this game includes a casing which houses 11 LEDs and two buttons all arranged in series with the two buttons on the ends, along with a power button. The players scores a point if and when the player clicks the button when the adjacent LEDs is lit; once the last LED is lit in the row the LEDs are then lit in the reverse order that they were initially lit; creating a “bouncing effect”. If the player manages to click the button in time the “ball” will start going back in the opponent’s direction and the cycle continues.

To make this game challenging the speed at which the “ball” moves increases. If a player is unable to press the button in time, or presses it to early, their opponent is awarded a point. This is indicated by the LEDs adjacent to their button. After each point, their light will blink to signal the number of points they have. The game progresses until a player reaches five points, which is signaled by the winning player’s LED staying continuously lit. The game can be restarted by pressing off button for the power change. To complete the build, the components will be housed in a box.

Research and Development

Contributors:

- Eric
- John
- Justin

Research and Development Tasks:

Research current submissions and see their level of creativity and implementations given constraints on coding size. Research possible projects that are achievable with the limited amount of time. Research ways of developing a working project which meet the 1K coding size. Researching which hardware to use in terms of microcontroller or microprocessor. Determining whether to make the project more of a novelty or more of a productive one. Proposed ideas and gave usefulness and possible methods for execution and integration

The 1K project started on 11/17/16 with the task of R&D team to find an idea that is possible to produce with the remaining weeks in the semester. The class agreed to focus the idea on the Amazon Dash button device. A document was posted on 11/23/16 from one of the members with the specifications on the Amazon Dash.

Amazon dash Specifications

- Broadcom Wifi Module (BCM943362)
    - Single-Chip 802.11 b/g/n
    - MAC/Baseband/Radio
- Processor STmicroelectronics (STM32F205RG6)
    - 32-Bit, 120MHz
    - 1Mb of flash memory

    - 128K of RAM
    - ARM cortex M3 Micro
    - Side note: Since Cortex M3 is ARM architecture, the project can use the original Amazon Dash.
- Microphone Analog Device (ADMP441)
    - 24-Bit I2S
    - Digital Microphone

The R&D team had to present their ideas on 12/1/16 on a calibrated Power Point with additional ideas outside the Amazon Dash. The other ideas included a RPG game, Rock Paper Scissors and 2D Pong game. The majority of the class chose 2D Pong as the project because of it simplicity and examples on the internet. Since the point of the project is 1K bytes, a language had to be efficient enough to complete this task, so AVR assembly was chosen because of its easy integration to Atmel microcontrollers. The Arduino Uno was used because most of the team members had one to spare, and easy to use with AVR; technical specification are included below.

Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Quality Assurance

Contributors:

- Samuel Alvarez
- Eric Moran
- Luis Gutierrez
- Himu Ganguli

For the QA team we worked with coders initially viewing their hardware and codes, and gave our input along side them stating what they could have done better to run the code. There were a lot of errors initially, but we helped minimize a lot of them. One error that was occurring a lot was that initially all the LED'S were lighting up for 2 seconds after running the program then after the LED'S would light up going down a row every ½ a second up to half-way. Meaning that the code wasn't fully working because it had to light up every ½ sec all the way back and forth. QA team made a replica of what the coders were using and running it on a Linux operating system to use commands such as AVRDUDE etc.

First we used a Raspberry Pi to install Linux. Once installed we had to use different commands to install the software and copy and paste the code into Linux. Some of the commands we used include "sudo apt-get update" which updates the repository, "Sudo apt-get install avra" which installs the assembler, "Sudo apt-get install AVRDUDE" which installed drivers for uploading. The last command mainly used was "Sudo apt-get install git" which was vital for downloading the codes off GitHub. Then we wrote "// clone repo (cd into Desktop first)" and "git clone [the - URL] and we just paste the GitHub URL to get the codes. The Coder member, Justin, also used a charlieplexing for the project which he chose to go 8 to 56 as he thought it would be easier to set up. We tested out the different codes and it seemed to run a lot better than before. The updated code from Github added .DEF , which are the names of registers so that it would be easier to convert them in the code on linux. The .ORG directive was added to the code which set up the interrupts for the setup and for the clock. The interrupts were still iffy last we checked but for the most part, 85 - 90% of it was good. In the code, there

was also a command that we tested that went pretty well which was rjmp. There was an rjmp SETUP and a rjmp OVERFLOW. rjmp SETUP was for the setup itself in the code and rjmp OVERFLOW was for the clock which like we mentioned previously was set ½ a second approximately. We tested and it ran fine.

An emulator for AVR called Atmel Studio was available for the project, but the problem was all the pieces of the code were not put together at the time and found the program to late for the team to use. QA continued use the physical hardware to test the code from the schematic on github, and test it. We also worked with the coders step by step to review their codes after building it. After testing it we found out some facts about it after building it like the maximum current that goes through each LED would be 120mW, and the power going through each LED is 62.5mW at a duty cycle of 25%. In the Charlieplexing schematic, we found out that to light up an LED you would have to ground the the horizontal row that it's in and input 5V into the resistor that's found in its vertical row for it to light up and again if you want to light up more LED’s you would only be able to light up the ones that are in the same horizontal row as long as you find the resistor that is in it’s vertical row and give it 5V of power to be able to light up. Charlieplexing the LEDs.

Hardware Team

Contributors:

- Justin Wong
- Darius Freeman
- Aldrick Castro
- Eric Moran
- Patrick Gonzalez

The project is a 2D pong game which consisted a certain number of LEDs and two input components such as buttons for two players. One issue occurred is the number of LEDs used surpasses the amount of pins the Arduino has, so the concept of charlieplexing was introduced to solve the problem. The main idea is creating a matrix design where a certain number of LEDs are using the same pin ports on microcontrollers. Out of a set of pins, one is used a row selector, while the others are used to power the LED's in the selected row.

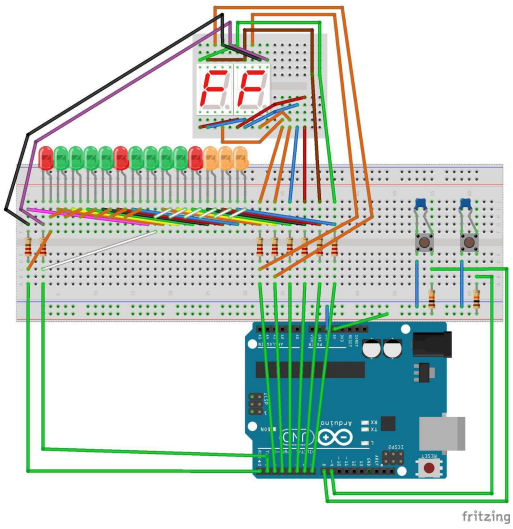
Two formulas were used to decide the number of pin used; The first formula decides the number of pins used with the known number of LEDs used, and the second formula gives you the number of LEDs you can use with the know pins used.

$$\frac{1+\sqrt{1+4L}}{2}, L = LEDs \quad / \quad n^2 - n = \#\ of\ LEDs, n = P ins$$

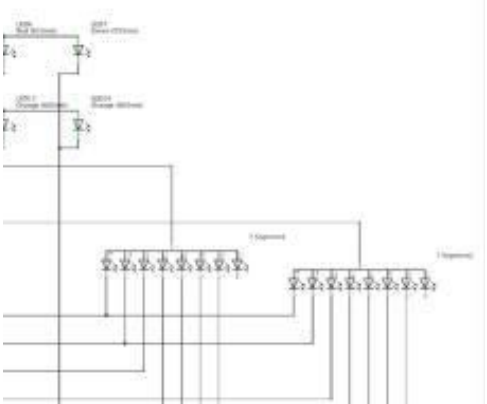
Since the number of LEDs was still unknown during the beginning of the project, we used the second formula to give us the possible LEDs used, and the team decided to use 8 pins strictly for LEDs. 8 pins were selected to controlling the 7 segment displays easier. The formula with 8 pins gave us $8^2 - 8 = 56$ possible LED we can use with charlieplexing. The first 11 LEDs used for charlieplexing were the 2D pong horizontal line which indicated the pong ball moving left and right. Three extra LED's were used as status indicators. Two 7-segment display were also used for the project and will be the next 14 LEDs, since each 7-segment display has 7 internal LEDs. These displays will be the points system for the 2D pong game for both players. Eight 220 resistors are connected to the eight pins used for the 28 LEDs. Two buttons have a 4.7nF

capacitor in parallel to solve the issue of debouncing, and a 10K ohm resistor in series with the positive end of each button.
Materials:

- Arduino Uno
- Wires
- 14 LEDs
    - 8 Green
    - 3 Red
    - 3 Orange
- Two 7-Segment
    - 14 internal LEDs
- Two buttons
- Two 4.7nF capacitors
- Eight 220 Ohm resistors
- Two 10K ohm resistors



Schematic:



Coding Team

Contributors:

- Justin
- Aldrick
- Darius
- Carlos

Coding Tasks:

As the team tasked with the coding aspect of our 1-D Pong project, we had quite a few responsibilities. After deciding on the direction to take our project in design-wise as a class. The first of which being, we had to test our components by coming up with sample code and posting the code to github. One of the main challenges we have face is keeping the code under 1 kilobyte as per contest rules outlined by the HackADay challenge. To mitigate this we used AVR to code in Assembly as suggested by Justin.

This greatly reduced our footprint in term of the overall size of the project. Justin was in charge of the Led Cycling instructions, Darius was in charge of the Seven Segment Display Instructions, Carlos was in charge of _____ and Aldrick was in charge of the Button Instructions. As the time passes we being to realize that the Arduino might be lacking pins for what we want to accomplish so Justin comes up with the solution of making a matrix or an array with the Leds in such a way where you could power multiple Leds with the use of less pins, this is called Charlieplexing. A few days later Justin presents a working LED cycle using the Charlieplexing Technique to the class.

Before switching to the AVR Assembly Instructions, Darius and Aldrick were able to get their share of the coding to work but using the Arduino IDE. But as stated before we noticed that that was going to take up too much memory so we decided to give AVR Assembly a try as Justin suggested. Aldrick and Darius had a difficult time understanding how this language went so Justin hosted a Hackathon at his home and offered to give us a basic rundown of AVR works and how to use it on our different machines. Darius had his windows computer and his raspberry pie so Justin thought Darius how to translate his code to AVR, compile, upload and put into the Github. Since Aldrick had a Mac he was able to do everything for his terminal.

While working together Darius and Justin realized that this technique could also be implemented to the seven segment displays to further reduce the amount of pins needed. After

a few days of working on the project they were able to successfully use Charlieplexing for the seven segment display. It was so successful that they even added a second one. After working with AVR Assembly for a while their code was looking like this:

Aldricks Code --with Justin's help-- :

```
.nolist
.include "./m328Pdef.inc"
.list

.def temp = r16
.def countOF = r17
.def BUTT = r26


.org 0x0000
rjmp SETUP

.org 0x0020
rjmp OVERFLOW



SETUP:
        ldi temp,0b00000000
        out DDRC,temp
        ldi temp,0b00000001
        out PortC,temp

        ldi temp,0b00000010   ;Enable pin change interrupts A0 - A5
        sts PCICR,temp          ;Start from Pages 73

        ldi temp,0b00000001
        sts PCMSK1,temp

        ldi temp, 0b00000101  ;Set clock scaling to 1/1024
        out TCCR0B, r16              ;See Datasheet 107-108 for more info
        ;; Since clock speed 20 MHz,
        ;; counter will increment at about 19.53 kHz when scaled
        ;; Therefore, overflow counter increments at about 76.29 Hz

        ldi temp, 0b00000001  ;Enable timer overflow interrupts
        sts TIMSK0, temp      ;See Datasheet 109 for more info

        sei                   ;Enable global interrupts

        ldi temp, 0b00000000  ;Reset counter to zero
        out TCNT0, temp

        ldi r18,0b00000001
        out DDRB,r18                        ;Setting pin 8 as an Output
```

```
        out PortB,r18                       ;Setting every pin to low except for pin 8

BUTTSETUP:
        BUTT1:
                ldi temp, 0b00000001  ; Toggle light
                eor r26, temp
                out PortC,r26

                ret     ;return for interrupt

        BUTT2:
                ldi temp, 0b00000010  ; Toggle light
                eor r26, temp
                out PortC,r26

                ret     ;return for interrupt

BUTT1LOGIC:
        SBRC PortC
        rjmp BUTT1
        nop

BUTT2LOGIC:
        SBRC PortC
        rjmp BUTT2
        nop

loop:
        rjmp loop

DELAY:                          ;Wait about r16/60 seconds
        ;; WARNING: halts program until delay is done
        clr countOF             ;Reset overflow counter
        cp  countOF, r16        ;Compare counter with r16
        brlt PC-1               ;Loop until counter > r16
        ret                     ;Return

OVERFLOW:
        inc countOF
        reti                    ;Return after interrupt
```

Darius' Code :

```
JUMP_SCORE1
ldi ZL, low(DISPLAY1_0)
ldi ZH, high(DISPLAY1_0)

mov  temp, score
lsl temp, 1
add ZL, temp
ijmp

DISPLAY1_N:
        ldi disp1, 0b00000100
        ret
```

```
DISPLAY1_0:
       ldi disp1, 0b11111110
       ret

DISPLAY1_1:
       ldi disp1, 0b01100100
       ret

DISPLAY1_2:
       ldi disp1, 0b11011101
       ret

DISPLAY1_3:
       ldi disp1, 0b11110101
       ret

DISPLAY1_4:
       ldi disp1, 0b01100111
       ret

DISPLAY1_5:
       ldi disp1, 0b10110111
       ret

DISPLAY1_6:
       ldi disp1, 0b10111111
       ret

DISPLAY1_7:
       ldi disp1, 0b11100100
       ret

DISPLAY1_8:
       ldi disp1, 0b11111111
       ret

DISPLAY1_9:
       ldi disp1, 0b11100111
       ret


JUMP_SCORE2
ldi ZL, low(DISPLAY2_0)
ldi ZH, high(DISPLAY2_0)

mov  temp, score2
lsl temp, 1
add ZL, temp
ijmp

DISPLAY2_N:
       ldi disp1, 0b00000100
       ret

DISPLAY2_0:
       ldi disp1, 0b11111110
       ret

DISPLAY2_1:
       ldi disp1, 0b01100100
```

```
        ret

DISPLAY2_2:
        ldi disp1, 0b11011101
        ret

DISPLAY2_3:
        ldi disp1, 0b11110101
        ret

DISPLAY2_4:
        ldi disp1, 0b01100111
        ret

DISPLAY2_5:
        ldi disp1, 0b10110111
        ret

DISPLAY2_6:
        ldi disp1, 0b10111111
        ret

DISPLAY2_7:
        ldi disp1, 0b11100100
        ret

DISPLAY2_8:
        ldi disp1, 0b11111111
        ret

DISPLAY2_9:
        ldi disp1, 0b11100111
        ret
```

Justin's Code :

Indirect relative jumping were found to be a very useful structures for 7 segment displays and the LED cycle. This allowed for switching by adding a number to the program counter, rather than using a series of comparisons. Therefore, jump tables were both faster and memory efficient. For the score displays, the score was added to the address of the function to display the number zero, then multiplied by 2 since each case was 2 bytes long. For the ball, the position was represented with a number from 0 to

Documentation Team

Contributors:

- Elmer
- Aayush
- Nil

Documentation Tasks:

To properly and thoroughly document all of our subdivided teams, such as coding, research and development, quality assurance and hardware. Also, to provide other individuals who are looking to recreate the project a clear and concise way to rebuild it. This can also help eliminate problems that would otherwise repeat. As such we have implemented the use of screenshots and code snippets to help make the documentation clearer and easier to follow. We have also added a table of contents for the same reason.

We decided to share the document on a platform in which all of us can access to view the document, and for the us documentors to edit and update it (Google Docs). A problem we faced was code snippets

Our group will post and update the final documentation, which will not only provide an all-around view of what every subteam has done but as time goes on we will update the official document so it will also be presentable for the 1K Hackaday Challenge submission.