Final Project Report

Peter Mansour (phm2122)
Shabhari Saravanan(ss5913)
Pratyush Agrawal(pa2562)
Aaron Pickard(ajp2235)

Summary:

Moon Patrol is an implementation of the Moon Patrol eight-bit arcade game from the 1980s. The basic concept of the game as the group has implemented it is that a player controls a vehicle moving across a surface. The vehicle encounters obstacles, which must be jumped over to avoid crashing into them. Jumping over the obstacle increases a player's score. The player is also able to shoot at a target above him. The bullets move vertically, and if they impact the target, the player's score also increases. The game ends when the player's vehicle collides with an obstacle, or their score reaches 25 points.

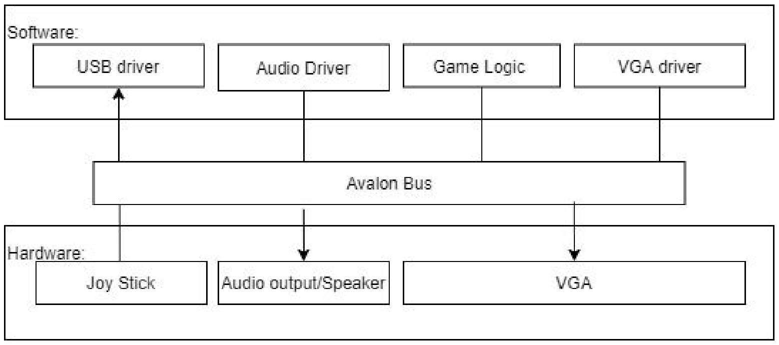


Fig 1. Software & Hardware Interface

The project leverages the Avalon Bus as the software/hardware interface. It manages the relationship between the game controller and the game logic, between the game logic and VGA driver and the visual output, and between the audio driver and audio output. Additional hardware for the project includes a VGA-compatible monitor, an xbox game controller, and an audio output device. Software written in c controls the game logic, and python scripts generate the sprites that move across the monitor. The audio driver is written in system verilog.

GRAPHICS:

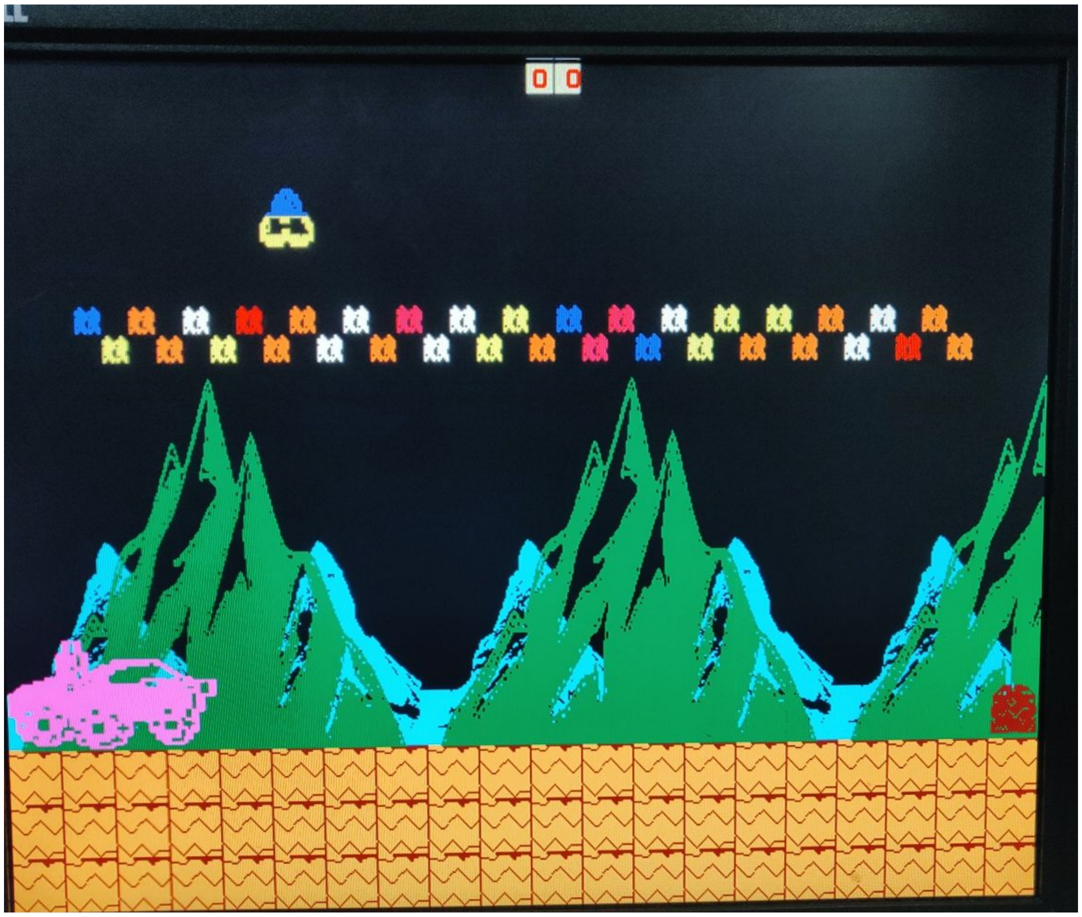


Fig 2. Graphics Display of the game

Background:

We have implemented a stationary tile based background with 2 mountains overlapping. We have implemented the stars in the background as well.

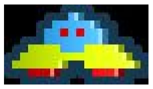
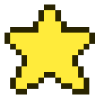
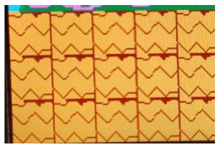
Sprites:

The moon buggy, the obstacle, ground, spaceship and bullets are sprite based and their movement can be controlled with some control registers that can be written by a software through device driver.

The sprites are stored in a ROM, which is a pattern of 1s and 0s. We have used the HCOUNT and VCOUNT to index through these stored patterns. So everytime 1 is encountered a particular colour is displayed at that pixel and if 0 is encountered then another colour is displayed at that pixel. Basically one bit {1 or 0} can be used to display different colours at one pixel. To save on memory we didn't store the information of the colour in the memory, instead we use the vcount and hcount to display different colours. For example, when the car jumps its vcount_reg changes to 4 and when the car is on ground then vcount_reg is 5, thus we can use this as a select of a

multiplexer that outputs different values of colour. Similarly, different colours in the stars were shown with help of hcount.
Hardware logic is written keeping in mind the priority order of different objects. For example the Car has the highest priority of being displayed, so first the pixels of cars are displayed and then the rest of the background is displayed like mountains etc .

Table1: Sprites Used in Game

Player	
Obstacles	
Spaceship	
Stars	
Mountains	
Ground	

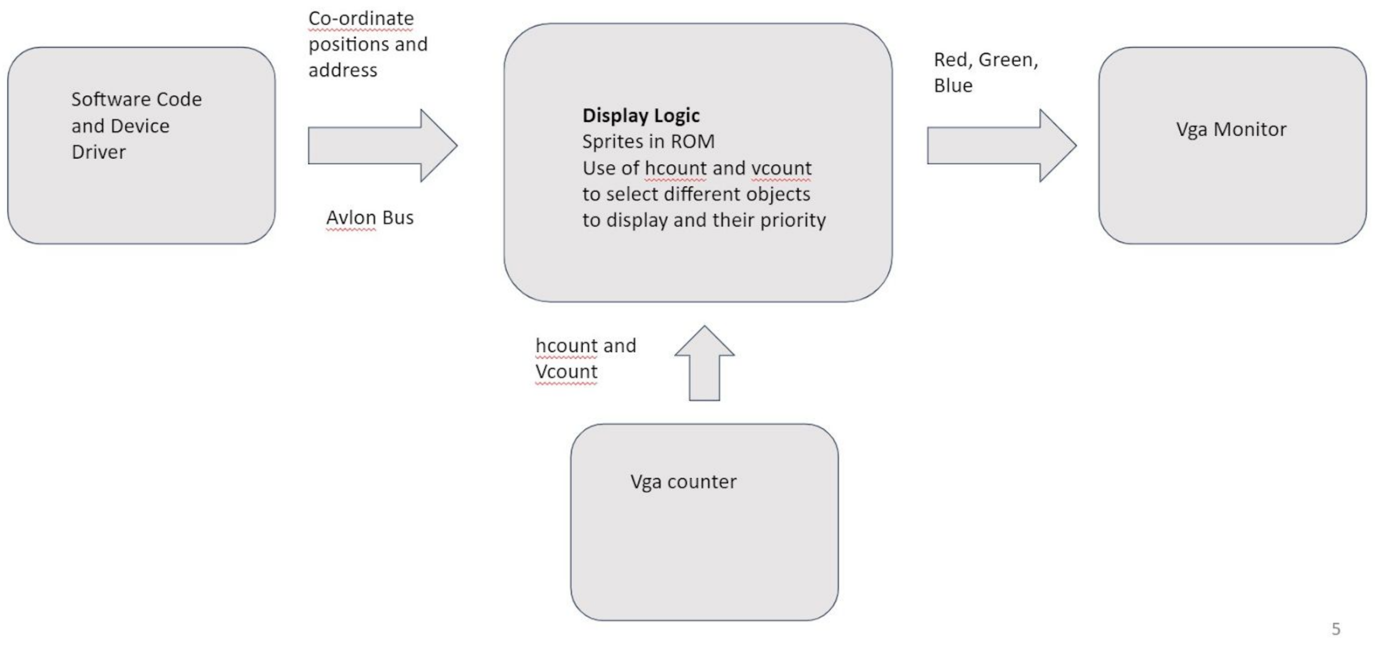
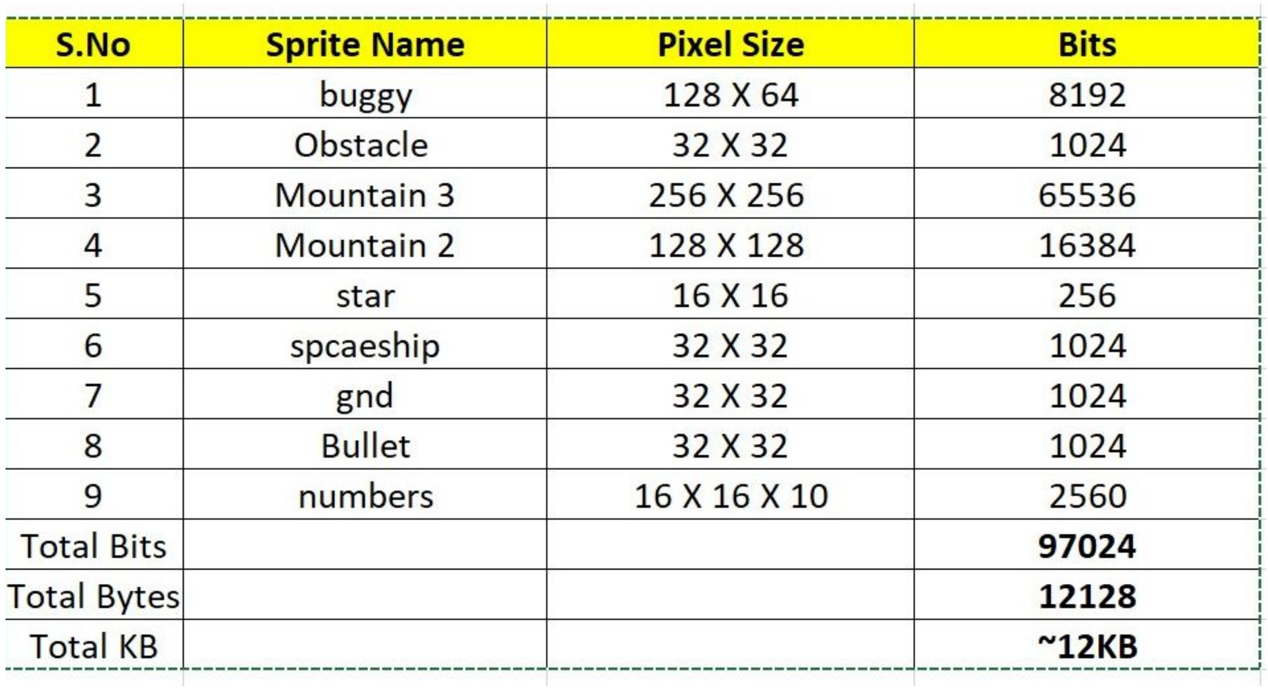

Software Code and Device Driver
Co-ordinate positions and address
Avlon Bus
Display Logic
Sprites in ROM
Use of hcount and vcount to select different objects to display and their priority
Red, Green, Blue
Vga Monitor
hcount and Vcount
Vga counter
5

Fig 3. Architecture of SW/HW interface

Memory Budget:

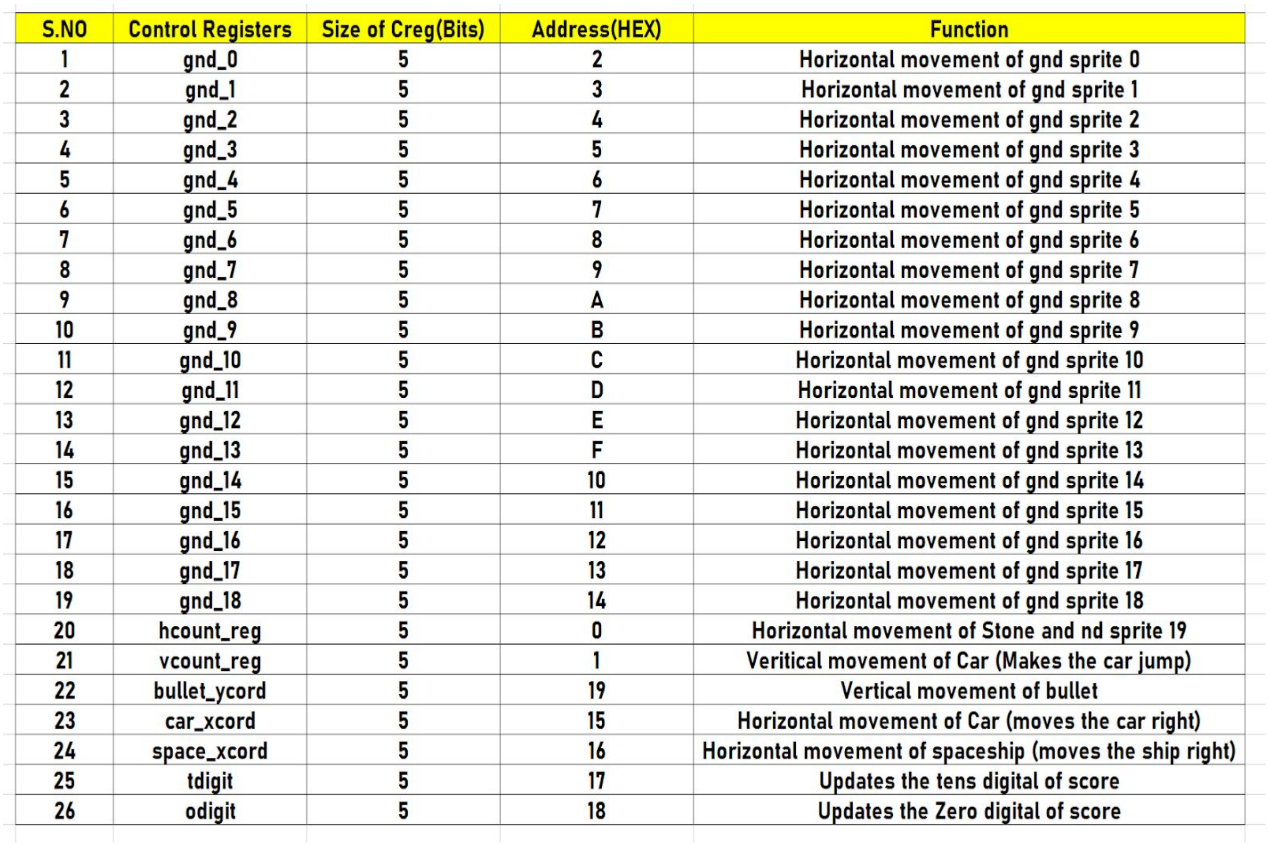
Table2: Sprites and their memory



S.No	Sprite Name	Pixel Size	Bits
1	buggy	128 X 64	8192
2	Obstacle	32 X 32	1024
3	Mountain 3	256 X 256	65536
4	Mountain 2	128 X 128	16384
5	star	16 X 16	256
6	spcaeship	32 X 32	1024
7	gnd	32 X 32	1024
8	Bullet	32 X 32	1024
9	numbers	16 X 16 X 10	2560
Total Bits			**97024**
Total Bytes			**12128**
Total KB			**~12KB**

Control Registers:

Table3: List of Control Registers



S.NO	Control Registers	Size of Creg(Bits)	Address(HEX)	Function
1	gnd_0	5	2	Horizontal movement of gnd sprite 0
2	gnd_1	5	3	Horizontal movement of gnd sprite 1
3	gnd_2	5	4	Horizontal movement of gnd sprite 2
4	gnd_3	5	5	Horizontal movement of gnd sprite 3
5	gnd_4	5	6	Horizontal movement of gnd sprite 4
6	gnd_5	5	7	Horizontal movement of gnd sprite 5
7	gnd_6	5	8	Horizontal movement of gnd sprite 6
8	gnd_7	5	9	Horizontal movement of gnd sprite 7
9	gnd_8	5	A	Horizontal movement of gnd sprite 8
10	gnd_9	5	B	Horizontal movement of gnd sprite 9
11	gnd_10	5	C	Horizontal movement of gnd sprite 10
12	gnd_11	5	D	Horizontal movement of gnd sprite 11
13	gnd_12	5	E	Horizontal movement of gnd sprite 12
14	gnd_13	5	F	Horizontal movement of gnd sprite 13
15	gnd_14	5	10	Horizontal movement of gnd sprite 14
16	gnd_15	5	11	Horizontal movement of gnd sprite 15
17	gnd_16	5	12	Horizontal movement of gnd sprite 16
18	gnd_17	5	13	Horizontal movement of gnd sprite 17
19	gnd_18	5	14	Horizontal movement of gnd sprite 18
20	hcount_reg	5	0	Horizontal movement of Stone and nd sprite 19
21	vcount_reg	5	1	Veritical movement of Car (Makes the car jump)
22	bullet_ycord	5	19	Vertical movement of bullet
23	car_xcord	5	15	Horizontal movement of Car (moves the car right)
24	space_xcord	5	16	Horizontal movement of spaceship (moves the ship right)
25	tdigit	5	17	Updates the tens digital of score
26	odigit	5	18	Updates the Zero digital of score

Sprite generation:
Using python's pillow module, we loaded the images into 3d numpy arrays and resized them to match the dimensions of the verilog 2d register. The rows and columns of the numpy matrix represent the height and width of the image, respectively. Each cell represents a pixel and holds 3 values: R, G, and B. We used the following equation to transform the 3 values in one, which is the luminance factor Y [1].

$$Y = 0.2126R + 0.7152G + 0.0722B \qquad (1)$$

We then used a threshold to divide the pixels into two bins, dark and light, based on their luminance factor, constructed a 2d array in verilog syntax of 1's and 0's, and wrote it to a .sv file with a block comment that shows the size of the image as the total number of pixels as well as the number of dark and light pixels. One the hardware side, 1's and 0's are set to the appropriate RGB colors.

SOFTWARE:

A c program is used to interface with the hardware and update the positions of the obstacle, buggy, bullet, spaceship, and ground throughout the lifetime of the game. The obstacle was allowed to move in the horizontal direction only; we decremented its position by 1 every 2 seconds from 19 to 0, right to left. The Buggy, representing the position of the player, should be updated by the controller. For simulation purposes, it was updated every 2 or 3 seconds. The buggy sprite is advanced from the left edge of the screen to the middle during the initial 2 seconds of the game and is held in place for the rest of the game's lifetime. The player is only allowed to jump up and down and thus only the vertical position of the buggy needs to be updated. If the player collides with the obstacle, the game stops. If the player successfully jumps as the obstacle passes through the middle of the road, the player earns a point. The score is displayed as 2 digits and the maximum score that could be earned before the game ends is 25. To give the illusion that the car is constantly moving from left to right, we moved the ground sprites from right to left at a constant speed. We divided the ground into different sections and decremented their horizontal position every 0.5 seconds. The bullet moves vertically only and its position is updated every 1 second. Enemy spaceship sprite's location is updated every 1 second and is allowed to move in the horizontal direction only. When the spaceship and bullet collide, the player's score is updated by one, otherwise the player's score stays the same. Since hardware registers retain their values when the game is over, all register values are reset at the start of each game.

AUDIO:

The audio.sv file defines a basic tone that is constantly playing. When events in the game occur, particularly jumping and shooting, different tones are triggered. The standard tone is played constantly in an always_ff() loop, unless a condition is triggered. If the flag for a jump is triggered, then the jump tone plays. If the flag for a shot fired tone is triggered, then that plays. The intention was to implement jump tone by monitoring the the vcount_reg register for changes, and the shot fired tone by looking at the bullet_ycord register. Because the loop of the hardware is significantly less than the audio time, it would be possible for the special tone to trigger and then release without the audio being translated to the hardware. This is handled by timer1 and timer2. Timer1 increments every clock cycle. Timer2 only increments on the standard tone clock cycles, and not when the jump or shoot tones would trigger. When one of these special tones triggers, the tone should continue to trigger until the difference between timer1 and timer2 exceeds 2^15 clock cycles, which since the clock triggers 50 million times per second, should happen infrequently enough that the signal is noticeably changed.

The initial plan was to convert audio files to .v files, configure the sounds using an I2C multiplexer, and then use CODEX WM8731 to output the sounds to a speaker. This was attempted, and unsuccessful. It was determined that the BrickBreaker project from Spring 2019 had developed an implementation for audio in a similar context, so that became the inspiration for the audio deliverable that accompanies this final report.

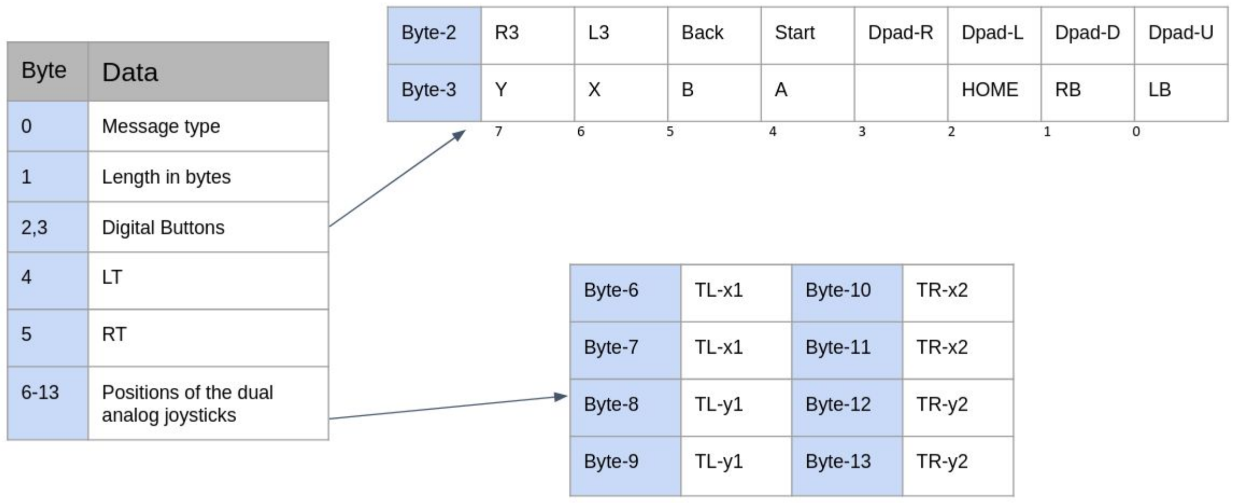
CONTROLLER:

XBOXDRV* is a userspace linux driver that runs on top of libsub-1.0 that supports Xbox360 USB gamepads. Remapping buttons and axes can be done using this driver. It also allows us to simulate keyboard and mouse events.

Dependencies needed to installed

- g++ GNU C++ Compiler
- Libusb1.0
- Pkgconfig
- Libudev
- Boost
- Scons
- X11
- Libdbus
- Glib

HID Descriptors:
Every update from the controller is a 20 byte packet. The descriptor table is given below, final 6 bytes are unused.



Byte	Data
0	Message type
1	Length in bytes
2,3	Digital Buttons
4	LT
5	RT
6-13	Positions of the dual analog joysticks

Byte-2	R3	L3	Back	Start	Dpad-R	Dpad-L	Dpad-D	Dpad-U
Byte-3	Y	X	B	A		HOME	RB	LB
	7	6	5	4	3	2	1	0

Byte-6	TL-x1	Byte-10	TR-x2
Byte-7	TL-x1	Byte-11	TR-x2
Byte-8	TL-y1	Byte-12	TR-y2
Byte-9	TL-y1	Byte-13	TR-y2

Once all the packages are installed, the scons command is used to compile in the source folder. “$ ./xboxdrv --quiet --nouinput” receives information about the HID descriptor. The command line output is read on the host side program using the popen function in C.The quiet flag is applied so that a shorted condensed message is printed on the command rather than a more detailed verbose message.

*https://github.com/xboxdrv/xboxdrv

All project files could be found at https://github.com/peter97mansour/MoonPatrol.git

References:

(1) *Circular Statistics Applied to Colour Images - ResearchGate*. www.researchgate.net/profile/Allan_Hanbury/publication/2905149_Circular_Statistics_Applied_to_Colour_Images/links/53d2515c0cf2a7fbb2e9943b/Circular-Statistics-Applied-to-Colour-Images.pdf.