

RetroChallenge 2021/10 competition entry: Using the Commodore 64 to control a remote control car.



Description:

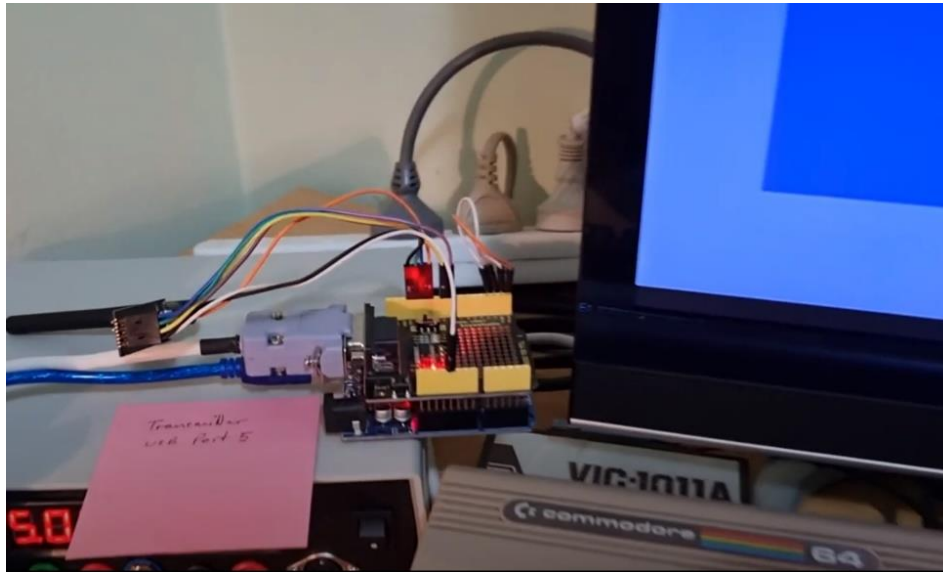
The Commodore 64 was released in 1982 and quickly became the best selling computer of all time. Many users treated the system as a games controller, but it was a real computer and could also be used as a microcontroller, using its ports like the User Port.

In this project I wanted to use this retro computer to control a remote control (RC) car. Using an inexpensive RC car and re-fitting it with an Arduino to allow connection to a radio transceiver (NRF24L01). On the Commodore 64 side I decided to use RS232, which on the C64 needs an adapter to bring it up to the voltages needed. The adapter released by Commodore was called the VIC-1011A.

Control of the RC car can be via a joystick and recorded (to a floppy disk) for later autonomous 'play back' by the C64. The C64 will use RS232 to a microcontroller with a wireless transceiver that can connect to the RC car. Commodore 64 control on an RC car By Steve Smit Started in September 2021

The code on the Arduinos used in this project is written using the IDE in the standard C++. The code on the Commodore 64 will initially be in BASIC, but will likely be re-written in either Forth (DurexForth) or assembler.

22<sup>nd</sup> September 2021 : First piece of progress, get C64 to send bytes representing the directions of the joystick by RS232 to a transceiver to a receiver to confirm 'commands' can be sent. Code on C64 written in DurexForth: <https://youtu.be/1RyrSI5RLjI>



I purchased 1/14 scale RC 'Monster Truck' off-road 2WD car for just under AUD\$40.

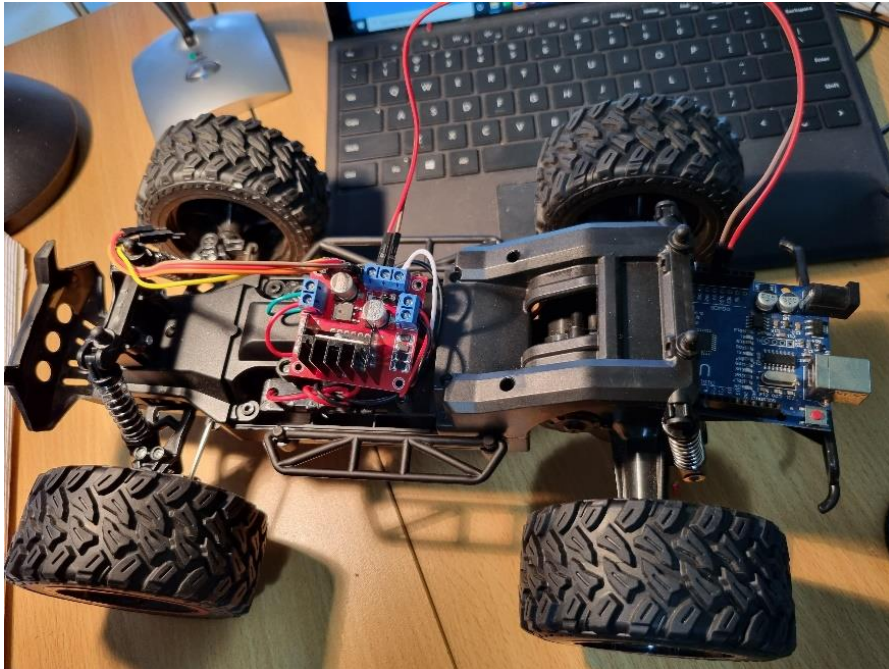


I first tried to 'hack' the controller to see if perhaps this might mean I could leave the electronics in the car unmodified. This initially worked!:

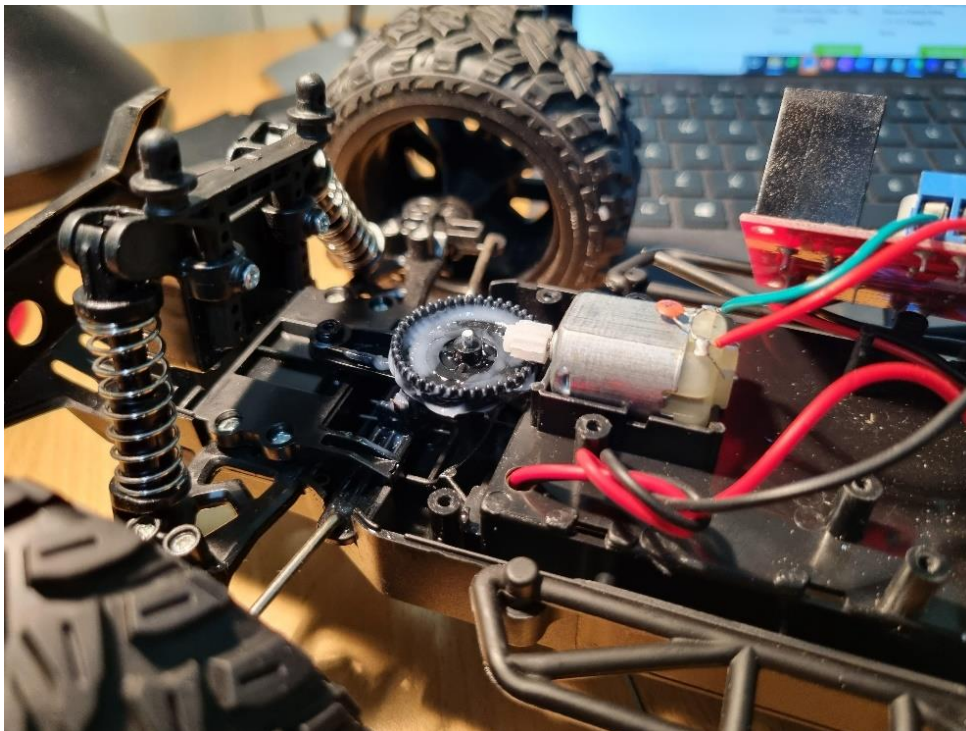
<https://youtu.be/TNTKXIKmA1Y>

But, for no apparent reason, the 'steering' motor, while making a straining noise, now doesn't seem to have the torque to turn the car anymore. Hacking the controller wasn't ideal in any case, as I wanted to have more control via the Commodore 64. I decided to open the car up, firstly to see if I could address the steering motor issue, but also to replace the motor controller with a L298N dual h-bridge motor controller module and hook up an Arduino to do some testing.





Even directly connecting to the 'steering' motor didn't fix the lack of movement issue. I have ordered a few replacement DC motors as an option to fix the issue. An alternative is to replace the steering mechanism with a servo motor. Here is how the current DC motor controls the steering:



While I wait for replacement items, I better get onto the code on the C64. I decided that the opening screen should offer the user 4 choices:

F1 = Operate RC car without recording

F3 = Operate RC car with recording

F5 = Show saved sequences

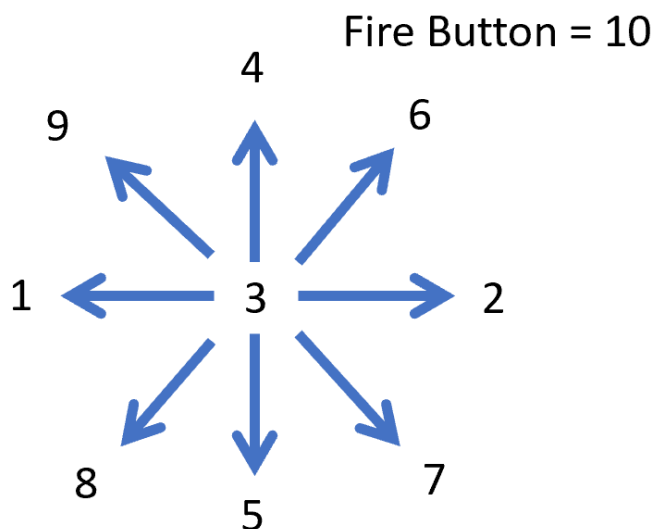
F7 = "Play" current sequence in memory , or something like this

I realized that I didn't know what byte is returned if someone presses the Function keys (F1, F3, etc.) so I wrote a quick basic program to get the values:

```
READ KEYBOARD & DISPLAY BYTE
PRESS ANY KEY TO SEE VALUE
PRESS RUN/STOP TO EXIT
A
B
65
66
133
134
135
136
BREAK IN 40
READY.
LIST
10 PRINT" READ KEYBOARD & DISPLAY BYTE"
20 PRINT" PRESS ANY KEY TO SEE VALUE"
30 PRINT" PRESS RUN/STOP TO EXIT"
40 GET AS:IF AS="" THEN GOTO 40
50 PRINTAS,ASC(A$) THEN GOTO 40
60 GOTO 40
READY.
```

From this I see that F1 = 133, F3 = 134, F5 = 136 and F7 = 137.

For the joystick movements, I decided that each direction and the fire button be given a value that would be sent as a command to the RC car.



On a Commodore 64 the joystick in port 1 can be read by reading the contents of 56321 (\$DC01). In BASIC this would be PEEK(56321).

I have chosen to write my code in assembler, so this is LDA \$DC01

The joystick default is all 1s, i.e. 255 or \$FF

Buttons (fire or a direction) pull bits low in the byte. Fire on it's own is then = 239 or \$EF

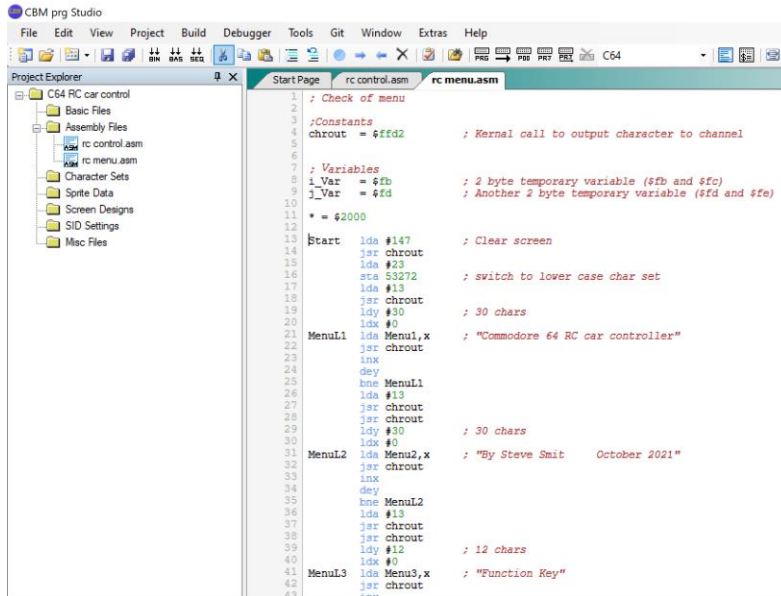
Up = 254, Down 253, Left 251, Right 247,

Up&Left 250, Up&Right 246, Down&Left 249,

and Down&Right 245. I then convert these to the commands in the diagram to the left.

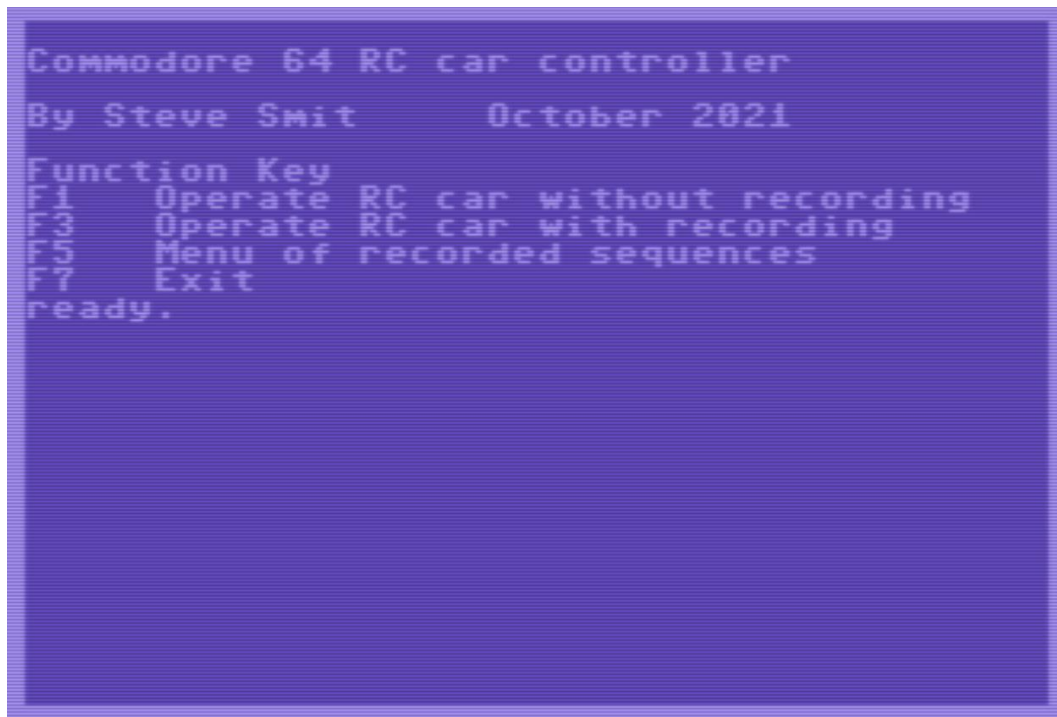
27 September 2021 – Assembly programming

I'm still not sure what language to use on the Commodore 64, but I know that Assembly has the maximum level of control and speed, so I will make a start and see how I go. I'm using an IDE called CBM Prg Studio to enter the assembly that compiles the machine code:



The screenshot shows the CBM Prg Studio IDE with the 'rc menu.asm' file open. The code is written in assembly language and includes comments in red. The code defines constants, variables, and three menu routines (MenuL1, MenuL2, MenuL3). The Project Explorer on the left shows the file structure of the project.

```
1 ; Check of menu
2
3 ; Constants
4 chROUT = $FFD2 ; Kernel call to output character to channel
5
6 ; Variables
7 i_Var = $FB ; 2 byte temporary variable ($fb and $fc)
8 j_Var = $FD ; Another 2 byte temporary variable ($fd and $fe)
9
10 * = $2000
11
12
13 Start lda #147 ; Clear screen
14 jsr chROUT
15 lda #23
16 sta 53272 ; switch to lower case char set
17 lda #13
18 jsr chROUT
19 ldy #30 ; 30 chars
20 ldx #0
21 MenuL1 lda Menu1,x ; "Commodore 64 RC car controller"
22 jsr chROUT
23 ldx #0
24 dey
25 bne MenuL1
26 lda #13
27 jsr chROUT
28 jsr chROUT
29 ldy #30 ; 30 chars
30 ldx #0
31 MenuL2 lda Menu2,x ; "By Steve Smit October 2021"
32 jsr chROUT
33 ldx #0
34 dey
35 bne MenuL2
36 lda #13
37 jsr chROUT
38 jsr chROUT
39 ldy #12 ; 12 chars
40 ldx #0
41 MenuL3 lda Menu3,x ; "Function Key"
42 jsr chROUT
43 ldx #0
```

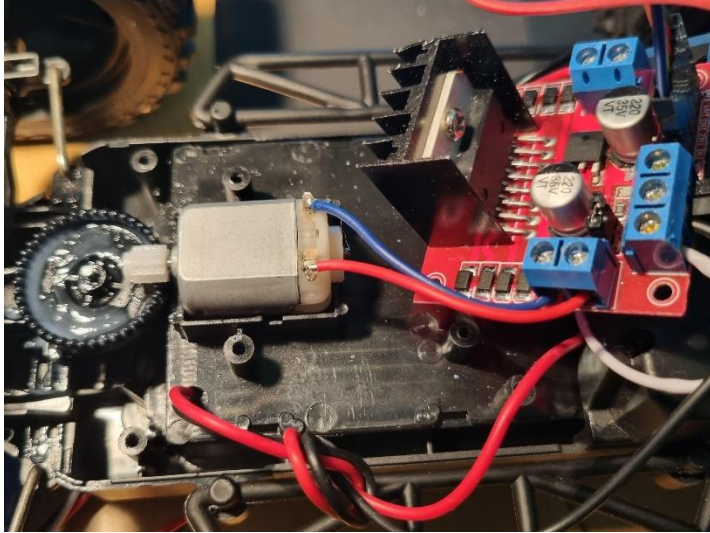


Well, not the flashiest of menus, but it's a start.



28 September 2021 – car steering motor repair

The replacement motor I ordered arrived and it fits perfectly and works! The Arduino sketch I used to test the car just spins the rear motor forward then backward, then confirms the front steering can be done. <https://youtu.be/1il9svvIH1M>



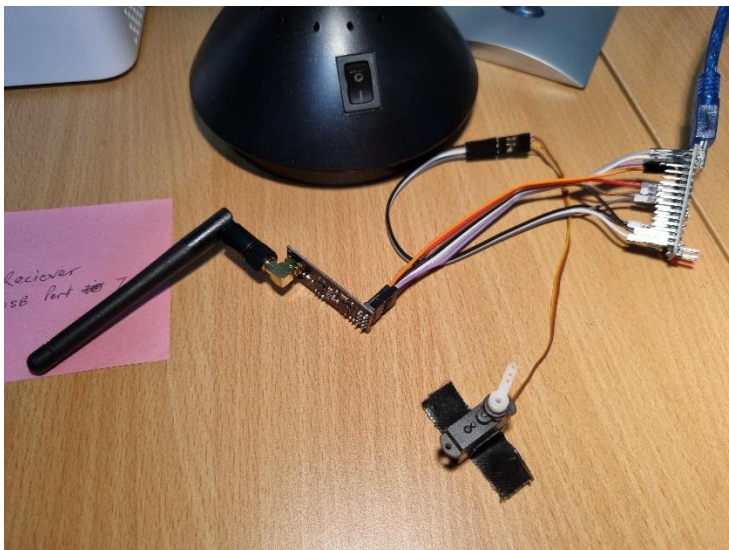
I still have a lot of coding to do on the Commodore 64, which I am persisting with Assembly language.

29 September 2021 – More machine coding

Now that I have a reasonable menu displayed, I need read the keyboard and act if one of the Function Keys is pressed.

1 October 2021

As the car has limited space, I decided to see if I could use an Arduino Nano in the car instead of the UNO. My first test was to see if I can send the commands for left, centre and right, including using a small servo to show this in action, and it worked! <https://youtu.be/7N0zTxdJpyA>

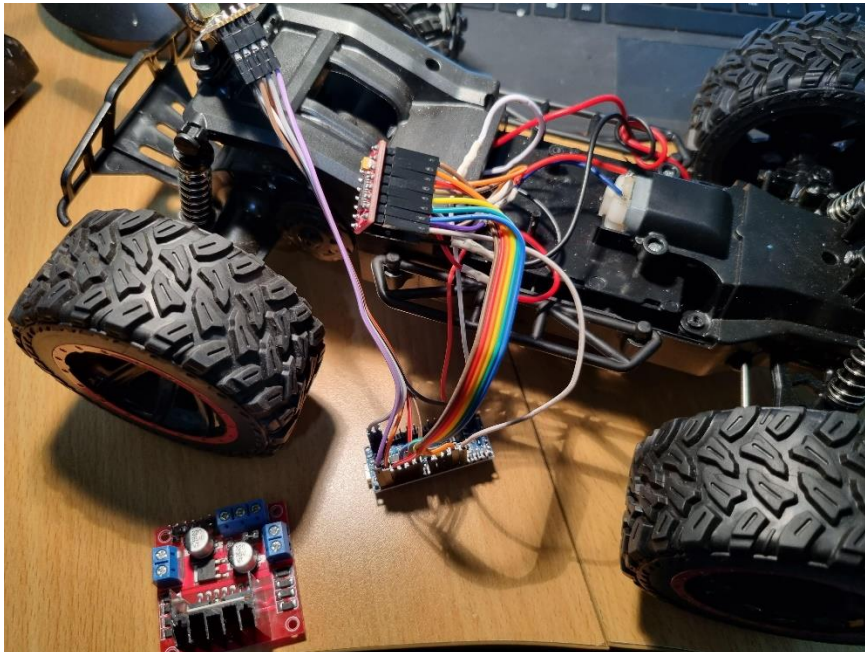


2<sup>nd</sup> October 2021 – Testing Nano in car with radio transceiver connected

After changing the Arduino UNO to an Arduino Nano, I did another test to confirm this works in the RC car. [https://youtu.be/uOMSnjM\\_ak8](https://youtu.be/uOMSnjM_ak8) . I'm pleased to see that it worked fine.

3<sup>rd</sup> October 2021 – Test change from L298N to TB6612FNG

I also decided that the L298N is a rather bulky device and loses around 1.4v from the battery, that I should see if I can use the more efficient and smaller TB6612FNG H-Bridge instead.



Of course, this also needed to be tested <https://youtu.be/IEmlqmNZZGI>

I then updated both the Arduino sketch and C64 programs to test also the forward and reverse control. I'm pleased to see that this works too! [https://youtu.be/RIUS3Q\\_FON8](https://youtu.be/RIUS3Q_FON8)

4<sup>th</sup> October 2021 – Code on Nano (in car) updated to support all directions

While my initial test for forward, reverse, left and right worked independently of each other, I also need to be able to support both moving forward while turning, as per the diagonal directions of joystick. I also wanted a way to avoid full power going to the front steering motor when it's reached the end of travel. This got more complicated than I anticipated, as I need to use millisecond timers and keep track of duration of time during turning as well as when already in fully locked turned position where I wanted to reduce power (down from 250 to 150 for instance).

I used a flag I called TTFlag (which stood for Turning Time Flag) which would be false when the steering motor was in the centre position. If the flag wasn't false, it meant that the steering motor was already turning under full power, so I needed to also read a variable that was measuring time in milliseconds to

know when to drop power to the steering motor from full to hold (i.e. 250 down to 150). I am pleased to report that I succeeded in getting this sorted out ( <https://youtu.be/m04mfECTdz8> ). Now to re-charge the battery, secure the electronics and aerial and replace the top of the RC car, and give it a test on the floor.

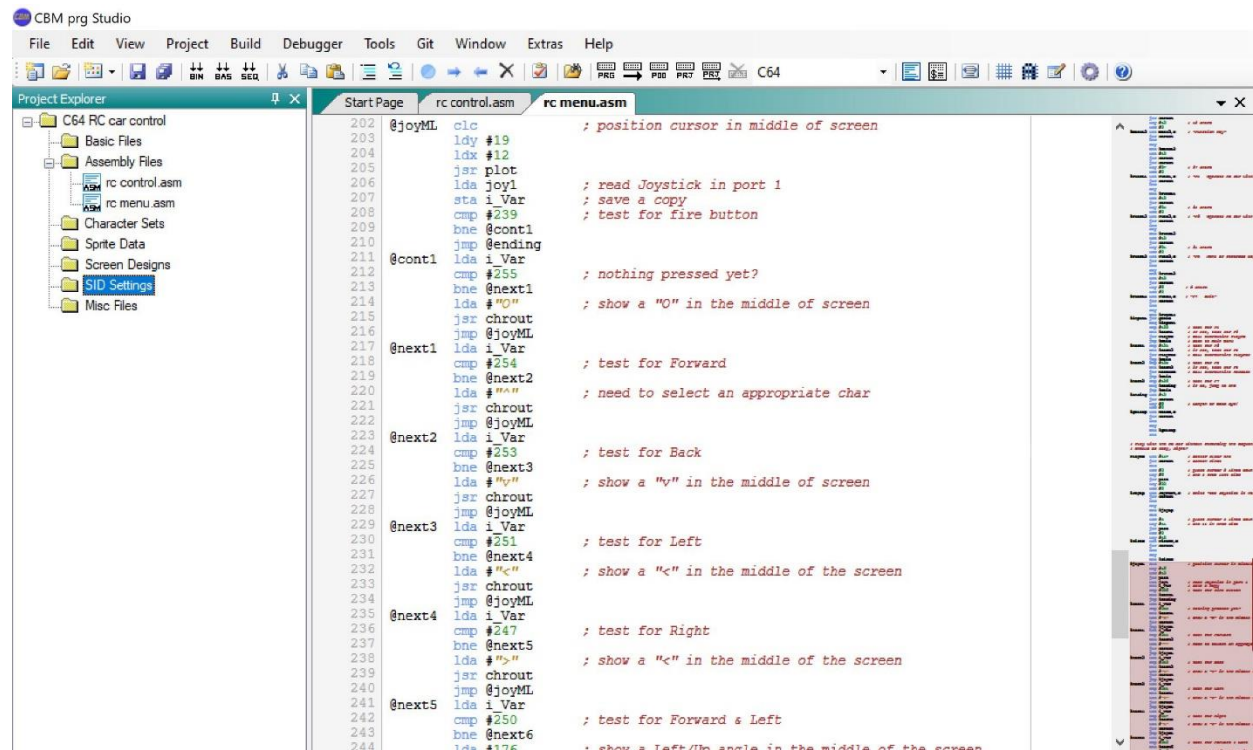
5<sup>th</sup> October 2021 – First floor test (without securing the electronics or adding the top of the RC car)

OK, the RC car is working (sort of). All the controls are being properly interpreted, but there appears to be an excess of current for the motor controller. Fortunately, the motor controller has over-current/heating protection but this is resulting in jerky movement after less than 30 seconds of use. <https://youtu.be/hHUsoKOsomM> . What I can be happy about is that all the other elements outside of the Commodore 64 seem to be working, so it's just the code on the Commodore 64 that needs to be done now.

6<sup>th</sup> October 2021 – Coding on the Commodore 64

I decided to write my code in assembler. The Commodore 64 uses a MOS 6510 processor, which is essentially the same as the famous 6502 used in other systems like the Apple II, Atari 2600, BBC Micro and Nintendo Entertainment System, just to name a few. Assembler is more challenging, but the reward is that the code is small and fast. I used an IDE (Integrated Development Environment) called CBM prg Studio (<https://www.ajordison.co.uk/>), written by Arthur Jordison.

The program so far sets up the RS232 port at 1200 baud, displays the main menu and F1 (Operate RX car without recording) is functional <https://youtu.be/LmQ5uK1qb5w>



```
202 @joyML clc ; position cursor in middle of screen
203 ldy #19
204 ldx #12
205 jsr plot
206 ldx joy1
207 sta i_Var ; read Joystick in port 1
208 cmp #239 ; save a copy
209 bne @cont1 ; test for fire button
210 jmp @ending
211 @cont1 ldx i_Var
212 cmp #255 ; nothing pressed yet?
213 bne @next1
214 ldx #0 ; show a "0" in the middle of screen
215 jsr chrout
216 jmp @joyML
217 @next1 ldx i_Var
218 cmp #254 ; test for Forward
219 bne @next2
220 ldx #0 ; need to select an appropriate char
221 jsr chrout
222 jmp @joyML
223 @next2 ldx i_Var
224 cmp #253 ; test for Back
225 bne @next3
226 ldx #0 ; show a "v" in the middle of the screen
227 jsr chrout
228 jmp @joyML
229 @next3 ldx i_Var
230 cmp #251 ; test for Left
231 bne @next4
232 ldx #0 ; show a "c" in the middle of the screen
233 jsr chrout
234 jmp @joyML
235 @next4 ldx i_Var
236 cmp #247 ; test for Right
237 bne @next5
238 ldx #0 ; show a "c" in the middle of the screen
239 jsr chrout
240 jmp @joyML
241 @next5 ldx i_Var
242 cmp #250 ; test for Forward & Left
243 bne @next6
244 ldx #176 ; show a Left/In angle in the middle of the screen
```



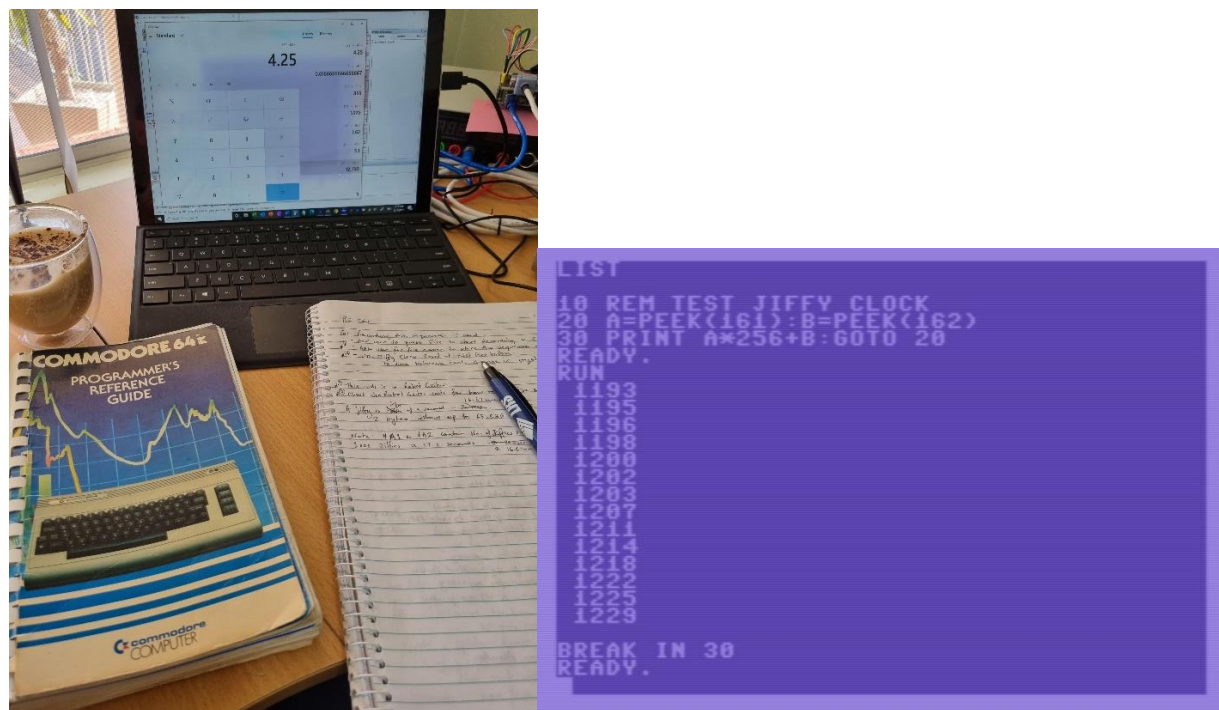
October 7<sup>th</sup> 2021 – Testing assembler to control RC car without recording

I worked on and completed a new assembler code routine called “RC CAR.ASM”. The code now waits until there is activity on the joystick before sending the appropriate command out via RS232. On the screen is also shown characters representing the directions the user is moving the joystick. For this function, exit is by pressing the fire button. <https://youtu.be/aGeefbiyoig>

Now I need to move on to writing code that will store the sequence, including the timing between each movement of the joystick. The code will be very similar to above, except that I will allocate the fire button as start and stop of the sequence recording. To exit I will use something like the space bar. This next coding will be the most challenging, and I still have the ‘play-back’ function code to write after that (which shouldn’t be as difficult, I hope).

October 8<sup>th</sup> 2021 – Starting to write the ‘function’ that will allow a sequence to be recorded

The Commodore 64 has a jiffy clock that ‘ticks’ every 16.67 milliseconds (60 per minute). After referring to the trusty Commodore 64 programmers reference guide, I see that the Kernal (think operating system) of the C64 after reset, increments from 0 the number of jiffies with the current count stored in most significant byte (of a 3 byte sequence) at \$A0, \$A1 and \$A2. I wrote a small basic program just to confirm this indeed can be read to check the number of jiffies after starting.



```
LIST
10 REM TEST JIFFY CLOCK
20 A=PEEK(161):B=PEEK(162)
30 PRINT A*256+B:GOTO 20
READY.
RUN
1193
1195
1196
1198
1200
1202
1203
1207
1211
1214
1218
1222
1225
1229
BREAK IN 30
READY.
```

The Jiffy clock can be reset by calling SETTIM (\$FFDB) with the Accumulator, X and Y registers set to 0. Then when a change occurs (i.e. joystick movement), the values in the Zero Page address \$A1 and \$A2 contains the number of jiffies since last reset. Two bytes of jiffies will be sufficient for this project, as this equates to over 18 minutes maximum gap between changes to the RC car’s last ‘command’, which is way more than needed. I was tempted to just use one byte, but that only allows for 4.25 seconds which won’t be realistic for some driving sequences (i.e. someone could want to have the car drive forward for 10 seconds before stopping).

October 10<sup>th</sup> 2021 – Getting heavy into 6510 machine code programming (Assembler)

I have now written routines for starting and stopping recording of RC car sequences. A new routine that not only sends the command byte to the RC car (in real time) but also saves the time since the last command. I have also commenced writing code that will prompt for a filename to save the current sequence (which is triggered when the user hits the fire button for the 2<sup>nd</sup> time. The first time is to start recording [note to self, I may need to watch out for 'button bounce']).

```
; Send Byte X without recording
; Sub-routine to send a byte via RS232

SndByt  ldx #$02          ; Set device
        jsr chkout        ; as output
        lda X_Var         ; Command to be sent to RC car
        jsr chrout        ; send the byte
        ldx #3            ; The screen
        jsr chkout
        rts

; Send Byte X with recording
; Sub-routine to send a byte via RS232

SndBytR  lda SeqPos       ; What position are we at
        ldx #3            ; this needs to be multiplied by 3
        jsr multi         ; Result acc = Hi Byte, x = Lo byte
        stx RCarray       ; store in Lo byte
        clc
        adc #$40
        sta RCarray+1     ; Hi byte order
        ldy #0            ; initial offset from where RCarray is pointing
        lda $a1           ; High Byte of Jiffies since last Reset
        sta (RCarray),y   ; Store in the RC array ($4000 + SeqPos*3)
        iny
        lda $a2           ; Low Byte of Jiffies
        sta (RCarray),y
        iny
        lda X_Var         ; load the command byte to be sent and stored
        sta (RCarray),y
        ldx #$02          ; Set device
        jsr chkout        ; as output
        lda X_Var         ; Command to be sent to RC car
        jsr chrout
        ldx #3            ; The screen
        jsr chkout
        inc SeqPos
        jsr Reset         ; reset the jiffy clock timer
        rts
```

October 11<sup>th</sup> 2021 – Added ability to prompt for a filename to be given to save a sequence file

```
; Get file name
; stores filename text at filetxt = $3f6c up to 18 characters
; will need to support backspace to delete back to fix file name error
; stores filename length in filnaml = $65

getflnm  lda #0           ; Use filnaml to store filename length
        sta filnaml       ; starting at 0
        sta BLNSW         ; make sure cursor is flashing
@loop    jsr chrin        ; read keyboard buffer
        beq @loop         ; loop if no character entered
```

```

        cmp #13           ; Has a CR been pressed
        beq @end          ; If so go end
        cmp #148          ; check if a backspace (del) has been pressed
        bne @cont         ; if not then jmp ahead to add character to filename
        lda filnaml
        beq @loop         ; if filename length is 0 then we can't backspace
        dec filnaml
        jmp @loop
@cont   ldx filnaml        ; load x as offset for storing filename
        sta filetxt,x     ; Store each charater at filetxt
        inc filnaml       ; Increment filnaml for each character stored
        jmp @loop         ; return back to get more char or a CR
@end    lda #1            ; Flag to turn the cursor blink back off
        sta BLNSW         ; Turn off the flashing cursor
        rts

```

## October 13<sup>th</sup> 2021 – Progress on saving a sequence file

I found I had used a variable called SeqLgth that was supposed to be the number of times the joystick had been moved by the user, but I was actually using another variable called SeqPos to measure this. So after some frustrations why I couldn't save the full sequence, all I had to do was use SeqPos times 3 for the total bytes to save to a file. Now I need to test being able to load a SEQ file and 're-play' this. Note: some of the code below is from Codebase64 "Writing to a file byte-by-byte".

```

; Save a recorded sequence
; This routine prompts the user for a filename (no error checking yet)
; Then saves the sequence just recorded

SaveRC   clc              ; Clear Carry flag = Set cursor position
        ldy #1            ; Cursor X position
        ldx #23           ; Cursor Y position
        jsr plot          ; Move cursor to initial 'Filename?:' position
        ldy #10
        ldx #0
@loop2   lda filen,x       ; Print "Filename?:" at line 23 and 1 in from side
        jsr chrout
        inx
        dey
        bne @loop2
        jsr getflnm       ; Get filename
        lda filnaml       ; check if any filename actually provided
        bne @cont1        ; if we got a filename let's continue
        jmp SaveRC        ; let's re-prompt for a filename
@cont1   ldy #0
        ldx filnaml       ; Read filnaml which will have the filename length
@loop3   lda fnseqx,y      ; Add the text ",s,w" to end of filnaml
        sta filetxt,x
        inx
        iny
        cpy #4
        bne @loop3        ; Have we added all the characters yet?
        clc
        lda filnaml
        adc #4            ; Accumulator should now contain filnaml+4
        ldx #<filetxt     ; Address in memory holding filename text
        ldy #>filetxt
        jsr setnam        ; Kernal call to set filename
        lda SeqPos
        ldx #3
        jsr multi

```



```

    stx i_Var
    sta i_Var+1
    lda #$01          ; Now to add $4001
    sta j_Var         ; I need to add $4001 because the routine below
    lda #$40          ; For saving does an inc after saving each byte
    sta j_Var+1       ; and therefore stops one byte short otherwise
    jsr Addit         ; Add $4000 to Song Length * 4, result in k_Var
    lda #$05          ; file number 5
    ldx #$08          ; default to device 8
    ldy #$02          ; secondary address 2
    jsr setlfs        ; call SETLFS
    clc
    jsr open          ; call OPEN
    bcs error2        ; if carry set, the file could not be opened
    ldx #$05          ; filename 5
    jsr chkout        ; call CHKOUT (file 5 now used as output)
    lda #$00          ; set j_Var to start address of where in mem file
    sta j_Var         ; file data starts from (being 2 bytes after $4000)
    lda #$40          ; so ignore the 2 bytes that are the load address
    sta j_var+1
    ldy #$00
@loop4  jsr readst     ; call READST (read status byte)
        bne werror    ; write error
        lda (j_Var),Y ; get byte from memory
        jsr chrout    ; call CHROUT (write byte to file)
        inc j_var
        bne @skip4
@skip4  inc j_Var+1
        lda j_Var
        cmp k_Var
        lda j_Var+1
        sbc k_Var+1
        bcc @loop4    ; next byte
close3  lda #$05       ; filename 5
        jsr close     ; call CLOSE
        jsr clrchn    ; call CLRCHN
        lda #0        ; since song is now saved
        sta RecFlg    ; reset the Modded flag
        sta DirFlag   ; Ensure the flag to read the directory is reset
endis   rts

error1  ; Akkumulator contains BASIC error code
        ; most likely errors:
        ; A = $05 (DEVICE NOT PRESENT)
        ; ... error handling for open errors ...
        jmp close3    ; even if OPEN failed, the file has to be closed

rdererror ; for further information, the drive error channel has to be read
        ; ... error handling for read errors ...
        jmp close3

error2  ; Akkumulator contains BASIC error code
        ; most likely errors:
        ; A = $05 (DEVICE NOT PRESENT)
        ; ... error handling for open errors ...
        jmp close3    ; even if OPEN failed, the file has to be closed

werror  ; for further information, the drive error channel has to be read
        ; ... error handling for write errors ...
        jmp close3

```

18<sup>th</sup> October 2021 – Coding for replaying

I now have code that on the emulator (Vice64) is working to allow the user to replay any sequence before deciding if they want to save it or not (when using F3 from the main menu). However, for some reason, on a real C64 the fire button isn't initiating the recording mode, but rather going straight to the option to replay sequence. Even the keyboard selection for yes or no isn't working on the C64 but is in the emulator.

I'm still very pleased with the progress and I suspect the issue lies in handling the bounce in the joystick fire button. I will tidy my code to see if I can exclusively filter for the fire button being released when pressed before moving to the next stage in the code.

<https://www.youtube.com/watch?v=l0moorJZzT8>

21<sup>st</sup> of October – Code for selecting previously saved sequences on the floppy disk

The way Commodore disk drives worked to get a directory is to use the Load "\$",8 command. I had to write code that collected the text that returned from this and create an array of only the SEQ (sequential) files. This is cool as the RC car commands and the delays between them I am calling a sequence, so SEQ is very appropriate.

[https://www.youtube.com/watch?v=XvLg5ZHbv\\_w](https://www.youtube.com/watch?v=XvLg5ZHbv_w)

I then wrote code to be able to list these sequences in a box. I decided to only show up to 10 in the box. If the number of sequences on the disk is more than 10, I also wrote code to allow the list to scroll up to see the extra files. Once the user has moved to the file they wish to load, pressing enter loads that sequence into memory and I wrote a new screen where the loaded sequence can then be 'played'. Even re-played if wanted again after playing.

[https://www.youtube.com/watch?v=XvLg5ZHbv\\_w](https://www.youtube.com/watch?v=XvLg5ZHbv_w)

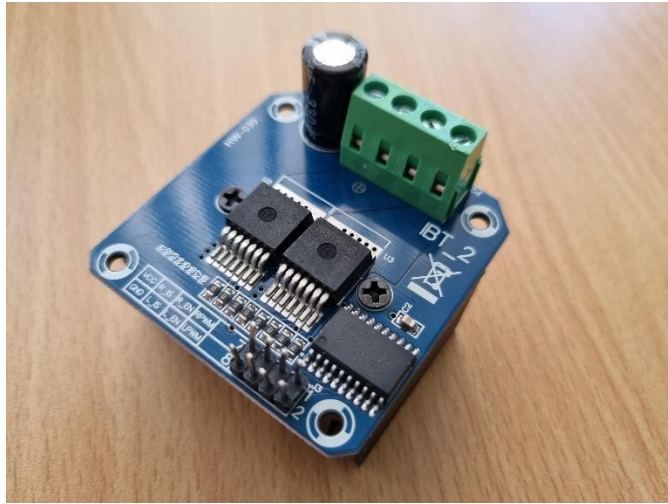
24<sup>th</sup> of October – Code complete!

In an attempt to improve the bounce handling when detecting the fire button of the joystick, I was getting rather frustrated as instead of fixing this one issue, I was introducing others. I'm still not completely satisfied with the reliability of using the fire button to toggle recording on and off, but at least all the code is now 100% otherwise working well.

<https://youtu.be/HLzApmj3KTE>

With the RC car there is a decision to make regarding the motor controller. The smaller TB6612FNG H-Bridge motor controller is more efficient, but when programmed to deliver full power to the motors for driving and steering, it's over current (or over temperature) protection soon kicks in and the car movements become jerky. I could leave this controller in and simply adjust the maximum power (by adjusting the PWM values for power to the motors) in the Arduino Nano code, or replace the motor controller (again) with one that can handle the full power.

I decided in advance to order a Double BTS7960 High Power Motor Driver Module which has just arrived:

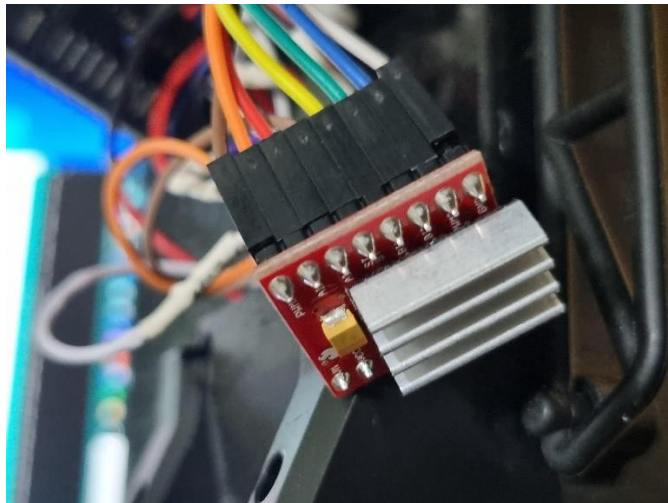


Perhaps, if time permits, I will try both options.

Still to do is to make sure the RC car works well on the floor or outdoors, and get the software loaded onto a real floppy disk to show the code works even loading from a Commodore 1541 disk drive. To complete the retro look, I may also setup the system with a Commodore 1901 monitor as the display (we shall see).

27<sup>th</sup> of October – Added heat sink to TB6612FNG

Before trying the BTS7960 I decided to see if the smaller TB6612FNG motor controller would work for longer with the addition of a heat sink.



This appears to add more time before the over-heat protection kicks in. I also tried tweaking the power delivered to the motors and this also gave me more time. It's still not perfect so, depending on available time, I may have to change to the higher current capable BTS7960.



30<sup>th</sup> of October – Transferred Commodore code onto a 1541 floppy disk.

I used a relatively modern utility called DRACOPY that works with the DS2IEC and a real floppy drive to copy a whole virtual disk to a real one. The copy worked, although did report a couple of errors so I'll have to see if this has affected the use of the disk.



31<sup>st</sup> October 2021 – Running the whole setup outside to demonstrate

I assembled all my original Commodore gear (well as much as I needed for this demonstration). This included using an original 1541 floppy disk drive to load the program from an actual floppy disk. I used a Commodore 1804 monitor as the display and recorded some video.





<https://youtu.be/gv8PXJQiT2U>