

Commodore 64: A Look at a BASIC language

By Javier Rodriguez and Amar Jagrup

Abstract: This project focuses on the Commodore 64 and the BASIC programming languages.

We have selected six topics to explore with. We looked at hardware, BASIC commands, input/output, graphics, sound, and compared BASIC to some modern languages like Java.

Section 1: Introduction and Hardware

The Commodore 64 (C64) at the time it was developed was designed around the philosophy that the sound and graphics of the system were to be state of the art for the world's next big video game (Matthews). The purpose of this paper is to explore the now almost forgotten language used in the C64 on the hardware it was built on. First let's look at the hardware in detail. The C64 uses MOS Technologies 6510 CPU with a clock speed of 1Mhz, a VIC-II 6567 video chip which produces 16 colors and the ability to display a resolution of 320x200, a SID 6581 sound chip which at the time was truly state of the art, it could produce a recognizable human voice without additional hardware. While the Commodore 64 does have 64K of ram available only 38,911 bytes are actually free because the rest of the memory goes towards the internal function of the language Commodore BASIC 2.0 (Matthews). Being developed with video games in mind the C64 has several parts of its hardware devoted to it. It has two game parts for controllers on the side panel. On the rear of the system it has a cartridge slot specifically for game cartridges, There is also a cassette interface which connects to an external cassette reader which is used to load programs from cassettes (Commodore Business Machines). This can be seen in Figure 1.1 (For more information on other aspects of the hardware not covered here see the (C64 reference manual).

At the time these specifications were extraordinary. Now they pale in comparison to the weakest of smartphones. However, it wasn't just its power that made the system popular it was the also the language it used. In order to create software and in order to use software on the C64, the

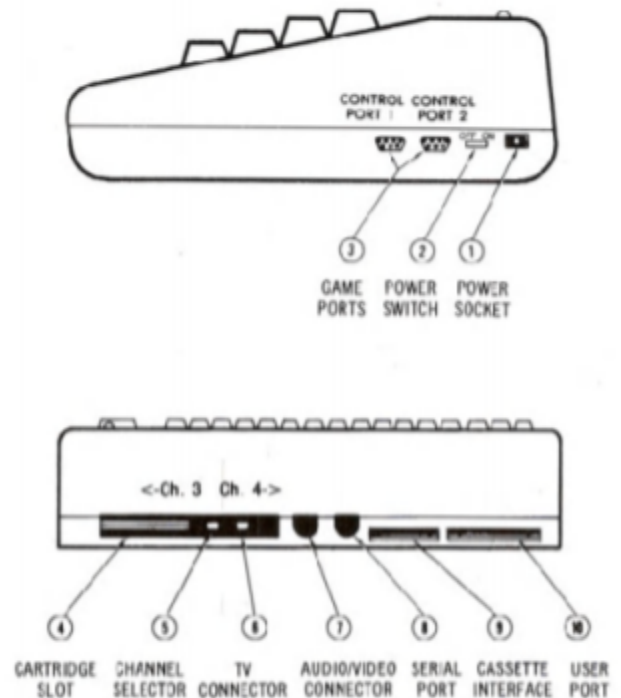


Figure 1: From the C64 User's Guide

system used a language called BASIC. The language gave the user essentially full access to the machine's now laughable hardware capabilities and allowed for games and programs that at the time were hard to replicate elsewhere. BASIC, while versatile is now almost an alien language to the modern programmers.

We will be examining BASIC on the C64 and describe its functions in four topics Graphics, Sound, Input/Output and comparing BASIC to the functionality provided by modern programming languages and each of these sections will discuss the hardware in a bit more detail where appropriate.. The following section **BASIC Commands and Programming Today** will cover the basic knowledge of how C64 BASIC V2 (From here on will just be referred to as C64 BASIC) keywords work as well as how to code for the C64 in the modern day.

Section 2: BASIC Commands and Programming Today

Before we discuss the the keywords used by the language it's imperative to understand what the BASIC Interpreter is and what it does. The BASIC interpreter is what analyzes the statement syntax the programmer types in and performs the required calculations and data manipulations. There are a total of 65 keywords that the interpreter recognized. There are also several characters that contain special meaning to the interpreter (Commodore Business Machines) . This can be seen in Figure 1.2.

There are two modes of operation in BASIC: direct mode and programming mode. Direct mode doesn't use line numbers and the statement is executed whenever the "Return" Key is pressed. Programming mode is the one used for actually used for making programs. Each line has a number associated with it.

Multiple statements can be used one one line but the system has an 80 character limit for each line. So, if that's reached than the next statement has to be placed on the next line (Commodore Business Machines). With all of that we can now discuss several key words that BASIC uses. One key word is

CHARACTER	NAME and DESCRIPTION
	BLANK—separates keywords and variable names
;	SEMI-COLON—used in variable lists to format output
=	EQUAL SIGN—value assignment and relationship testing
+	PLUS SIGN—arithmetic addition or string concatenation (concatenation: linking together in a chain)
-	MINUS SIGN—arithmetic subtraction, unary minus (-1)
*	ASTERISK—arithmetic multiplication
/	SLASH—arithmetic division
↑	UP ARROW—arithmetic exponentiation
(LEFT PARENTHESIS—expression evaluation and functions
)	RIGHT PARENTHESIS—expression evaluation and functions
%	PERCENT—declares variable name as an integer
#	NUMBER—comes before logical file number in input/output statements
\$	DOLLAR SIGN—declares variable name as a string
,	COMMA—used in variable lists to format output; also separates command parameters
.	PERIOD—decimal point in floating point constants
"	QUOTATION MARK—encloses string constants
:	COLON—separates multiple BASIC statements in a line
?	QUESTION MARK—abbreviation for the keyword PRINT
<	LESS THAN—used in relationship tests
>	GREATER THAN—used in relationship tests
π	PI—the numeric constant 3.141592654

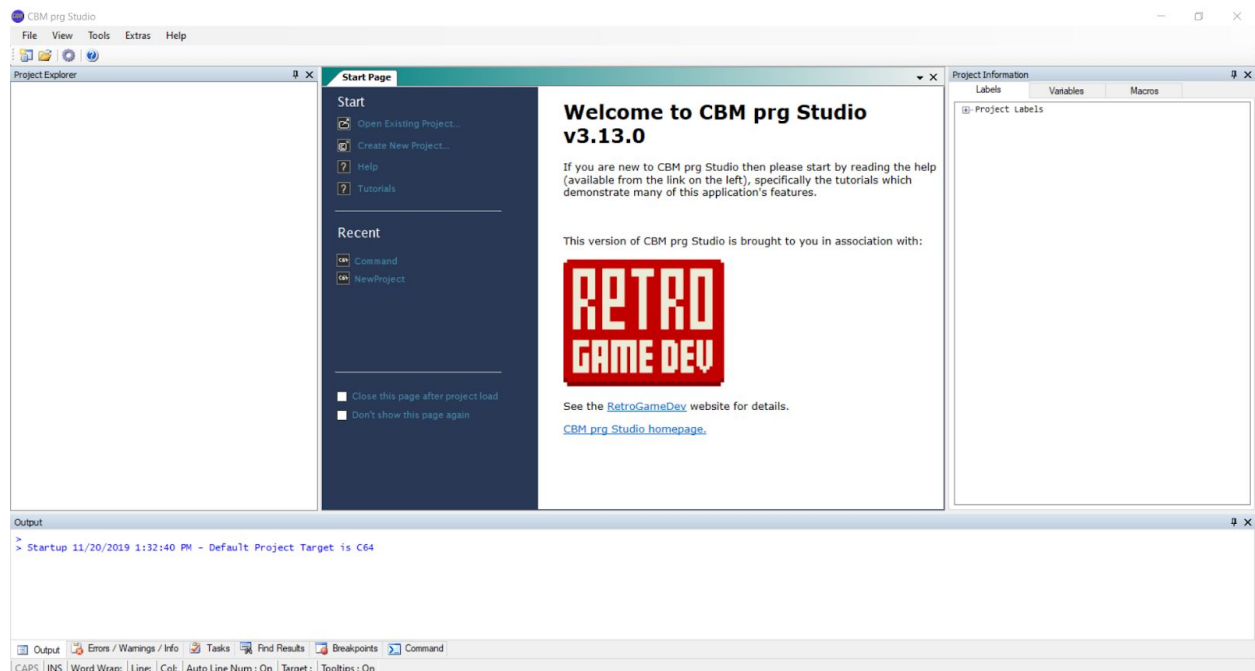
Figure 2: from the C64 Programmers Reference Manual

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
PRINT "HELLO WORLD"
HELLO WORLD
READY.
```

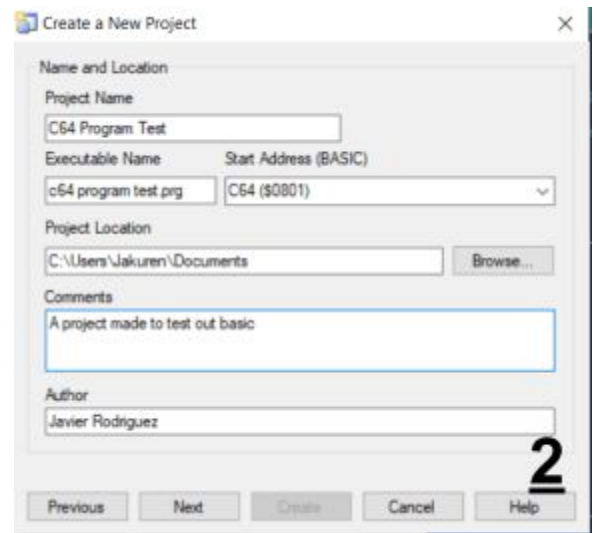
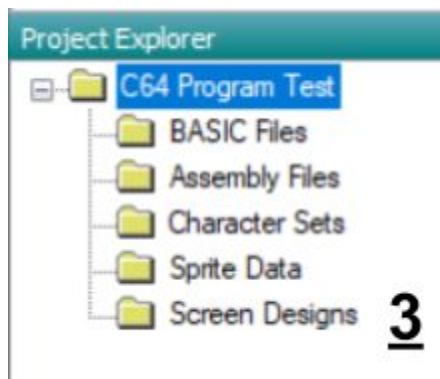
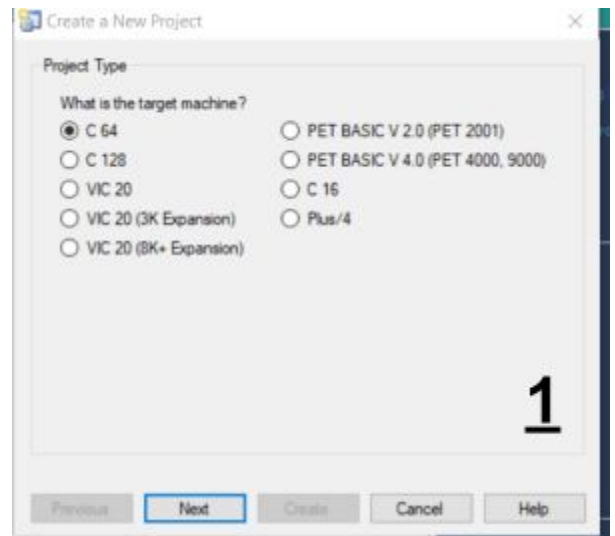
Print which is used to output strings, numbers and the value of variables to the screen. It can be used in Direct mode but is more meant for programming mode where it can be used to print the result of a program.

Now while an emulator is useful for getting nice screenshots of the Commodore 64 it doesn't add any quality of life features that would make programming or inputting keywords easier. So while there will be cases in which a keyword can be used in direct mode, many keywords really have their use in programming mode and programming in C64 to be blunt is a bit of a nightmare in the modern day. In order to program for the C64 today it's best to use a programs like "CBM prg Studio" alongside an Emulator like VICE (Links to these software are in the links section). Let's look at setting this up as this is what we'll be using to write and test our code.

As an aside, it's possible to get code written in this software to run on actual hardware as we don't have actual hardware to test that process that won't be covered here. One you've downloaded the previously mentioned software, start "CBM prg Studio"

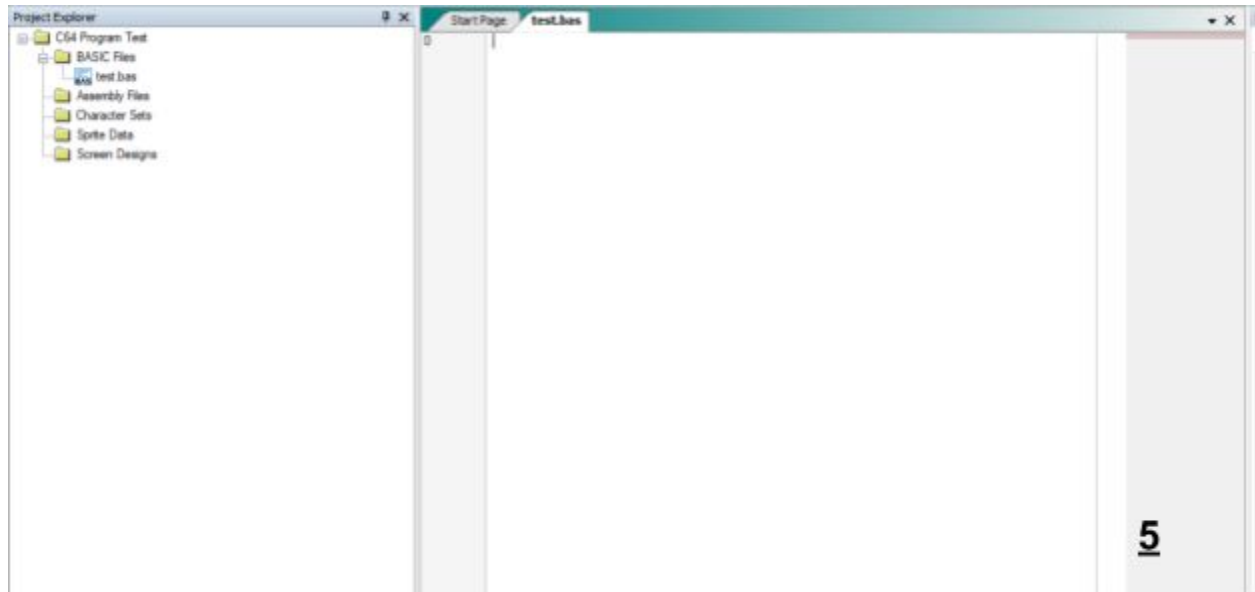
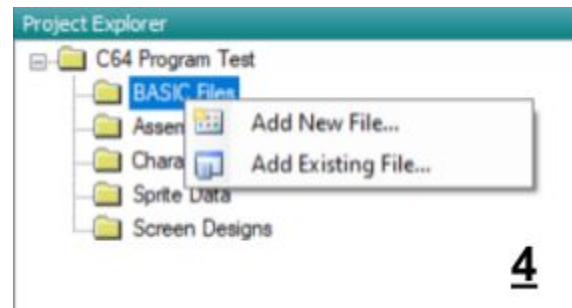


You'll be greeted with a screen like the one above. After this now we need to set up a project specifically for use with the C64. Click on "File" at the upper left then on "New Project. Make sure "C 64" is selected then click "Next" (1) and fill out the information to your liking on this second screen (2). Afterwards click "Next" and then on "Create" Once that's done you'll see a file tree in Project explorer (3).



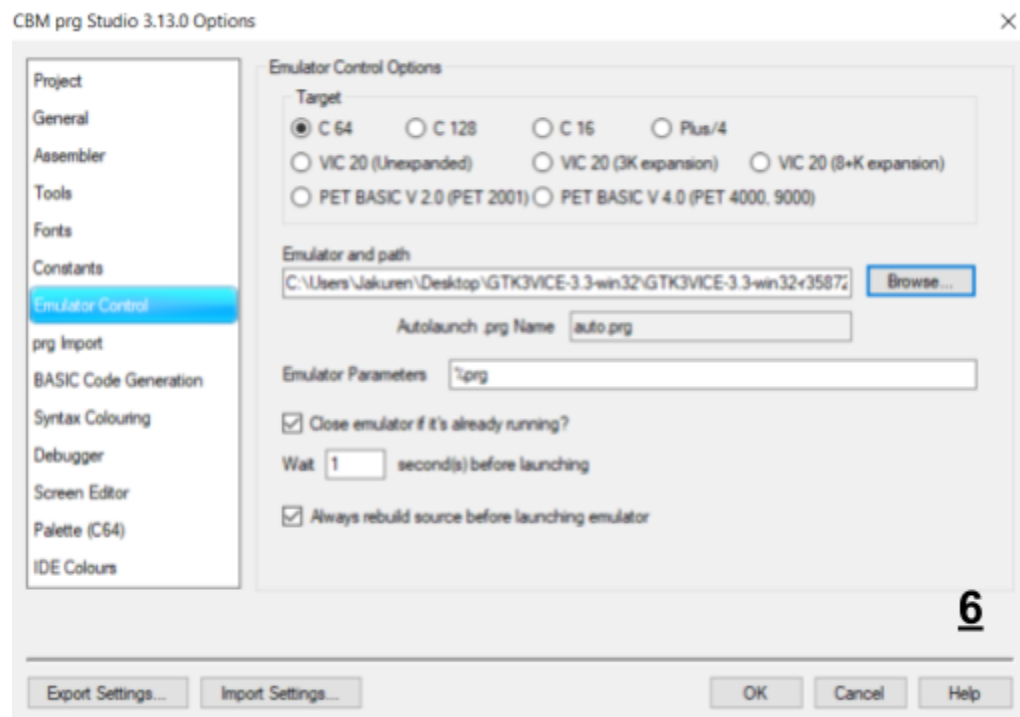
Now we right click on “BASIC Files.” (4) Then you just input a name for our new basic file and now we have a BASIC file we can edit and export to the emulator once some code has been written.

(5)

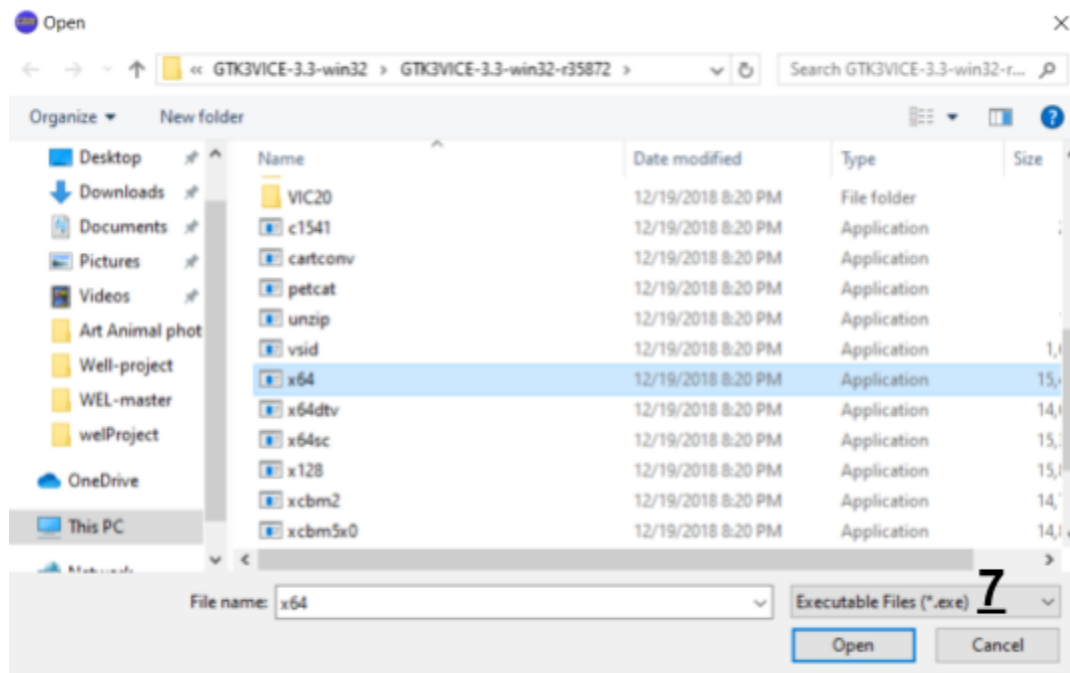


But before we can do that we need to connect the emulator. Now click on the “Tools” tab at the top of the program and then click on “Options”. On the screen that pops up click on “Emulation Control”.

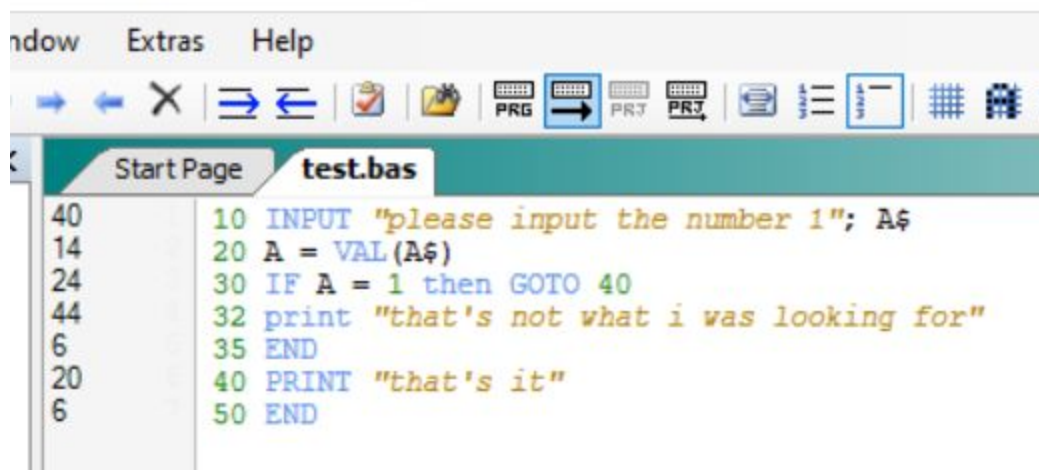
That screen should be set up like this (6). At this point you will need to have the file containing the emulator extracted somewhere. Click



on browse and navigate to the folder containing the VICE emulators. Ensure that the “x64” executable is selected as this is the C64 emulator (7). With this programming with BASIC for the C64 became a lot practical as with his program we can more easily edit the code we write. The image below shows us an



example of what we can create. Clicking the highlighted button will run the created program in the emulator. In this development environment, keywords can be upper or lower case but strings have to be lower case.



This program looks gets user input with “INPUT” and then stores it in A\$. The “\$” after a variable name denotes that variable as a string variable. A variable without any suffix is known as a Real Variable in C64 BASIC and can hold any number from $\pm 2.93873588 \times 10^{-38}$ thru $\pm 1.70141183 \times 10^{38}$. There are also integer variables which are denoted with a “%” suffix they can hold 16-bit signed binary integers in the range from -32768 thru 32767. “VAL()” looks in a

string for a numerical value so by using that we store the user input in A. Then I use “IF” to check if A is equal to 1. If it is then we go to line 40 and print the message then hit “END.” Which just ends the program, if we don’t find a 1 then we continue to the next line and print a different message and then end the program. The line numbers in the above program don’t just exist for show, they are a part of the program itself. Unless instructed by a “ GOTO” statement which takes the program to a specific line number, the C64 just reads each line in order until it has no more lines to read or hits an END keyword. C64 BASIC is a procedural programming language. A lot different than languages like Java or C++ and it preceded C. C64 BASIC is closer to Assembly language than modern languages in fact the C64 even supports the writing in the Assembly language of the CPU. However, that’s difficult not only to learn but to read and write. It’s worth noting that some functions of the C64 are exclusive to its version of Assembly, as an example, screen scrolling (think of something like scrolling up and down on a webpage) is a feature that can only be accessed with Assembly (C64-wiki).

Section 3: Graphics

Before coding graphics on the C64 it's imperative to understand how exactly the VIC-II chip (the GPU) deals with graphics. There are 1000 possible locations on the C64 screen and each has a location in memory associated with it. This starts at location 1024 and goes to location 2023. Each location has 8-bits associated with it, so it can store an integer between 0 to 255. After this we have Color Memory (also known as Color Ram). Color memory starts at location 55296 to 56295. Each of these locations only hold 4 bits of memory so they can only store a number from 0 to 15. Which is perfect since the C64 can only display 16 colors. There are also graphic modes that are available by selecting one of the 47 registers on the VIC-II chip. These can be accessed by simply using the POKE function on any of these registers which can be found from locations 53248 to 53294. Now the VIC-II chip only can see up to 16K of memory at a time but the C64 has 64K of memory. The VIC-II can see all of this through the use of sections or Banks. So the VIC-II can switch what bank it's looking at with a Bank select bit located on the CIA chip #2 at location 56576. The idea is that each bank can hold different character sets or sprites, so if you need to switch what's being displayed on screen with something else store that data in a different bank and then swap banks. A bit of a quirk is that the keyboard won't work if the bank being switched too doesn't have the character set for the keyboard characters. Those have to be manually copied over in order to use the keyboard or if you want to display those characters with a character set you have created. That's the gist of the graphics capabilities of the C64 (Commodore Business Machines). Now as the C64 was built with gaming in mind, it's important to understand what a sprite is.

To put it simply a sprite is a pattern or design used to represent something graphics wise and sprites are usually used to represent visual assets of a game like the player character or an enemy. For example this image on the right is a simple sprite of a face. Now on a C64 a sprite must be 24 bits in width and 21 bits in height. A sprite would be stored in a bank taking up 63 bytes of space. These sprites must be placed in a memory location that is divisible by 64. Now sprites can be made in one of two modes for the C64. High Resolution or Multicolor.



There is a lot to these modes but the basic difference is the width of the pixels. Multicolor has pixels that are twice the width of those of High Resolution pixels. Since Multicolor pixels are defined by 2 bits of data instead of one. You can assign more colors to a Multicolor sprite (Commodore Business Machines). A High Resolution sprite is best used for something you want to add a lot of detail in at the expense of color. You would use Multicolor if you want more colors on you sprite at the expense of detail. Sprites are created pixel by pixel so it's highly recommended that you use some kind of software to create one. CBM prg Studio has a sprite editor under the tools tab that can be used to create sprites and it even has an export that will give you, "DATA" lines describing each pixel of your sprite that can be read with the "READ" command. The C64 wiki has a great example of how graphics work.

```
1 PRINT CHR$(147): V=53248: POKE V+33,0: REM clear screen
2 FOR X=12800 TO 12927: POKE X,0: NEXT X: REM Clear RAM for sprites

10 FOR X=12800 TO 12881: READ Y: POKE X,Y: NEXT X: REM sprite generation
11 POKE 2040,200: POKE 2041,201: POKE 2042,201: POKE V+21,7
12 POKE V+28,6: POKE V+37,15: POKE V+38,2: REM multicolor for sprite 1&2
13 POKE V+39,7: POKE V+40,8: POKE V+41,6: REM sprite color sprite 0&1&2
15 POKE V+23,7: POKE V+29,7: POKE V+16,1: REM sprite properties height, width, x
16 POKE V+1,133: POKE V+2,170: POKE V+5,115: REM x/y positions

19 REM moving and changing colors
20 FOR X=200 TO 1 STEP -1: POKE V,X: Z=Z+0.61: POKE V+3,Z
21 POKE V+4,(201-X)/2: NEXT X
22 POKE V+16,0: POKE V,255: M=PEEK(V+4)
23 FOR X=255 TO 170 STEP -1: POKE V,X: Z=Z+0.66: POKE V+3,Z
24 POKE V+4,M+(256-X)/1.2: NEXT X
25 FOR X=0 TO 5: FOR Y=1 TO 255: POKE V+37+X,Y: NEXT Y,X
26 POKE V+38,2: POKE V+39,7: POKE V+41,6
27 FOR Y=1 TO 65: POKE V+40,Y: POKE V+37,Y+10
28 FOR Z=0 TO 15: POKE V+39,Y: NEXT Z, Y

29 REM waiting, deleting sprite 0 and fade off
30 FOR X=0 TO 3000: NEXT X
31 FOR X=0 TO 32: POKE 12832+X,0: POKE 12832-X,0
32 FOR Y=0 TO 100: NEXT Y,X: POKE V+21,0

39 REM SPRITE C64-WIKI.DE (unicolor; sprite 0)
40 DATA 239,81,85,139,81,20,137,81,28,137,81,89,137,213,89,142,85,93,138
41 DATA 95,85,138,91,85,238,91,85,0,0,0,0,0,0,0,0,0
42 DATA 0,199,0,0,231,0,0,164,0,0,180,0,0,151,0,0,180,0,0,164,0,0,231,0,0,199,0

44 REM multicolor sprite lines (sprite 1&2)
45 DATA 0,255,255,255,170,85,170,170,85,170,85,170,85,170,85,170,85,255,255,255
```

“REM” stands for remarks and is used for comments in basic. The above code makes an animation on screen where two pixel blocks and some text will slide into the center and then the pixel blocks will change colors. After a while the text will fade and then you’ll be taken back to the command line. The text is a High Resolution sprite (sprite 0) and as such can only be one color. The blocks are multicolor sprites (sprites 1 and 2) and can have multiple colors. Lines 39 to 45 have the sprite data and using POKE and PEEK the sprites are moved pixel by pixel and the colors are changed pixel by pixel in order to create the animation with FOR loops. Before the text fades Figure 3.1 will be seen on screen.



Figure 1

Now we attempted to display the face sprite Javier created but we couldn’t get it to work. It may be because the RAM wasn’t cleared before attempting to display a sprite but even then while we understand the basics of Graphics on the C64, the way to do program Graphics here is different from anything we had to do before and requires in depth knowledge of which memory locations your messing with and to make sure your not using POKE on the wrong locations. A programmer really needs to know the VIC-II chip in its entirety before they attempt any sort of animation.

Section 4: Sound

The Commodore 64 has the ability to play sounds and using BASIC you can make complex sounds and songs. Able to play music because the commodore has a SID chip. It comes complete with three voices. To program sounds on the C64 you will use the POKE statement. POKE sets the indicted memory location (MEM) equal to a specific value (NUM) (C64 reference guide). There are specific memory locations for making music on the Commodore and they are 54272 to 54296. The numbers you use in the POKE statement must be between 0 and 255. There is also a peek function, which is equal to the current value in the indicated memory location. The commodore has parameters called attack/decay/sustain/release, which is related to how the musical tone changes. When a note is first struck, it rises from zero volume to its peak volume. The rate at which this happens is called the ATTACK. Then, it falls from the peak to some middle-ranged volume. The rate at which the

Line(s)	Description
5	Set S to start of sound chip.
10	Clear all sound chip registers.
20	Set Attack/Decay for voice 1 (A=0,D=9). Set Sustain/Release for voice 1 (S=0,R=0).
30	Set volume at maximum.
40	Read high frequency, low frequency, duration of note.
50	When high frequency less than zero, song is over.
60	Poke high and low frequency of voice 1.
70	Gate sawtooth waveform for voice 1.
80	Timing loop for duration of note.
90	Release sawtooth waveform for voice 1.
100	Return for next note.
110-180	Data for song: high frequency, low frequency, duration (number of counts) for each note.
190	Last note of song and negative 1s signaling end of song.

Figure 1:Example code from c64 reference manual that plays some music shown on the left and a line by line explanation of the code shown on the right)

fall of the note occurs is called the DECAY. The mid-ranged volume itself is called the SUSTAIN level. And finally, when the note stops playing, it falls from the SUSTAIN level to zero volume. The rate at which it falls is called the RELEASE (c64 reference guide). To get an idea of how this works the reference manual gives an example. Using the same code from Figure 4.1 if we change line 20 to poke s+5,88:pokes+6,195. At first it did not sound that different, but

playing it back a few times it does sound different that the sound generated by figure 4.1. You can see this code in Figure 2.

```
5 s=54272
10 for l=s to s+24:poke l,0:next: rem clear sound chip
20 poke s+5,9:poke s+6,0
30 poke s+24,15: rem set maximum volume level
40 read hf,lf,dr
50 if hf<0 then end
60 poke s+1,hf:poke s,lf
70 poke s+4,33
80 for t=1 to dr:next
90 poke s+4,32:for t=1 to 50:next
100 goto 40
110 data 25,177,250,28,214,250
120 data 25,177,250,25,177,250
130 data 25,177,125,28,214,125
140 data 32,94,750,25,177,250
150 data 28,214,250,19,63,250
160 data 19,63,250,19,63,250
170 data 21,154,63,24,63,63
180 data 25,177,250,24,63,125
190 data 19,63,250,-1,-1,-1
```

Figure 2 (c64 reference guide)

Section 5: Input and Output

The simplest form of output in BASIC is the print statement. For output to the tv the print statement is used. There is also output to different devices such as a cassette deck, printer, disk drive or modem. The open statement in BASIC creates a “channel” to talk to one of these devices(C64 reference manual).

Example of writing to a printer 100 open 4,4:print# 4, "Writing on printer"

FORMAT: OPEN file#, device#, number, string

DEVICE	DEVICE#	NUMBER	STRING
CASSETTE	1	0 = Input 1 = Output 2 = Output with EOT	File Name
MODEM	2	0	Control Registers
SCREEN	3	0,1	
PRINTER	4 or 5	0 = Upper/Graphics 7 = Upper/Lower Case	Text Is PRINTed
DISK	8 to 11	2-14 = Data Channel 15 = Command Channel	Drive #, File Name, File Type, Read/Write Command

(Figure 1 c64 ref manual)

Each device has a different open statement, but they all follow a similar format. The output of the printer is similar to the output of the tv except when printing you are concerned about the format. The format should be easy to read. The Commodore 64 has two 9 pin game ports which allow the use of joysticks, paddles or a light pen. One thing to note about paddles is that they are not reliable when read from BASIC alone. There is a machine language routine on page 365 on the reference manual on how to use the paddles in either BASIC or machine code. The commodore has a built-in RS-232 interface for connections to any RS-232 modem, printer or other device. To connect a device to the Commodore 64, all you need is a cable and a little bit of programming. BASIC has its own subroutine to read the joystick(C64 reference manual). That subroutine is shown in figure 5.2 along with the corresponding jv values.


```

start tok64 page344.prg
  10 fork=0to10:rem set up direction string
  20 readdr$(k):next
  30 data"", "n", "s", "", "w", "nw"
  40 data"sw", "", "e", "ne", "se"
  50 print"going...";
  60 gosub100:rem read the joystick
  65 ifdr$(jv)=""then80:rem check if a direction was chosen
  70 printdr$(jv);" ";rem output which direction
  80 iffr=16then60:rem check if fire button was pushed
  90 print"-----f-----i-----r-----e-----!!!":goto60
  100 jv=peek(56320):rem get joystick value
  110 fr=jvand16:rem form fire button status
  120 jv=15-(jvand15):rem form direction value
  130 return
stop tok64

```

The values for JV correspond to these directions:

JV EQUAL TO	DIRECTION
0	NONE
1	UP
2	DOWN
3	-
4	LEFT
5	UP & LEFT
6	DOWN & LEFT
7	-
8	RIGHT
9	UP & RIGHT
10	DOWN & RIGHT

(Figure 2 c64 reference manual)

Section 6: Modern Languages

Modern Languages like Java and Python have a lot of functionality that can be or can't be replicated by the C64 BASIC. C64 BASIC does have the ability to use comments as we seen in the Graphics section with the REM command. Any text after REM will be a comment. It also has an IF keyword. With the IF command BASIC can use most most logical operators that every language has although the syntax may be different. An IF keyword has to be followed by a THEN or GOTO statement. While there isn't an ELSE keyword like in Java or Python you do have that functionality. If a condition is not met BASIC ignores that line and goes to the next one so just include what you would have included in the else statement in the line after. If you wanted to have a nested ifs in BASIC the

best way to do that is to do an IF...GOTO.

Where after the condition passes you GOTO a Line that has the code you want to perform, then the line after that has an IF statement. Look at Figure 6.1 as an example of this.

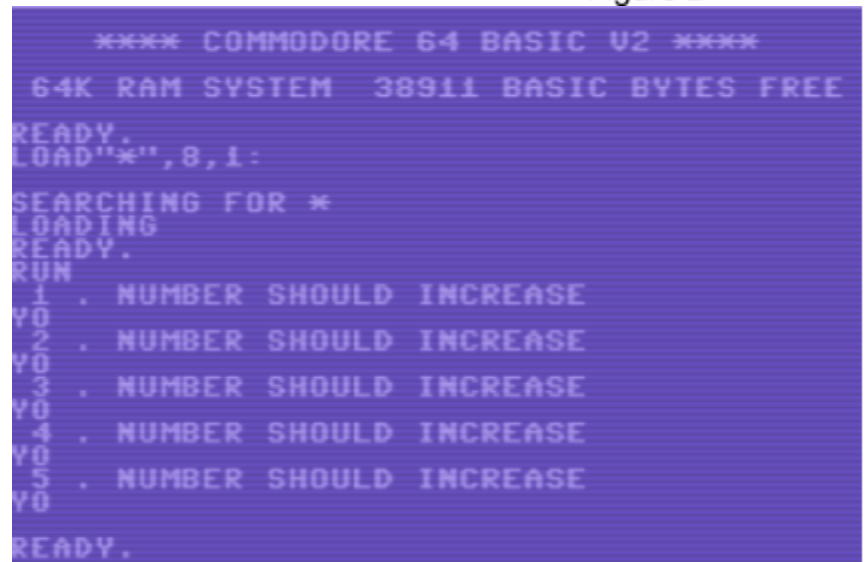
The FOR construct in C64 BASIC is analogous to most of its counterparts in other languages. The difference though is its structure. It has an ending Keyword with NEXT and it has a STEP keyword as a part of its structure. STEP is optional and simply determines how much to increment the loop variable when the loop reaches NEXT. BASICally what happens is everything between FOR and NEXT is repeated until the limit of what the loop increment variable is set

```
10 INPUT "please input a number"; A$
20 A = VAL(A$)
30 IF A > 1 GOTO 40
32 print "that's not what i was looking for"
35 END
40 PRINT "that's it"
45 IF A > 100 then PRINT "thats pretty big"
50 END
```

Figure 1

```
10 REM For Loop Example
20 FOR X=1 TO 5 STEP 1
30 PRINT X". number should increase"
35 PRINT "yo"
40 NEXT X
```

Figure 2



```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
LOAD"*.8,1:
SEARCHING FOR *
LOADING
READY.
RUN
1 . NUMBER SHOULD INCREASE
V0
2 . NUMBER SHOULD INCREASE
V0
3 . NUMBER SHOULD INCREASE
V0
4 . NUMBER SHOULD INCREASE
V0
5 . NUMBER SHOULD INCREASE
V0
READY.
```

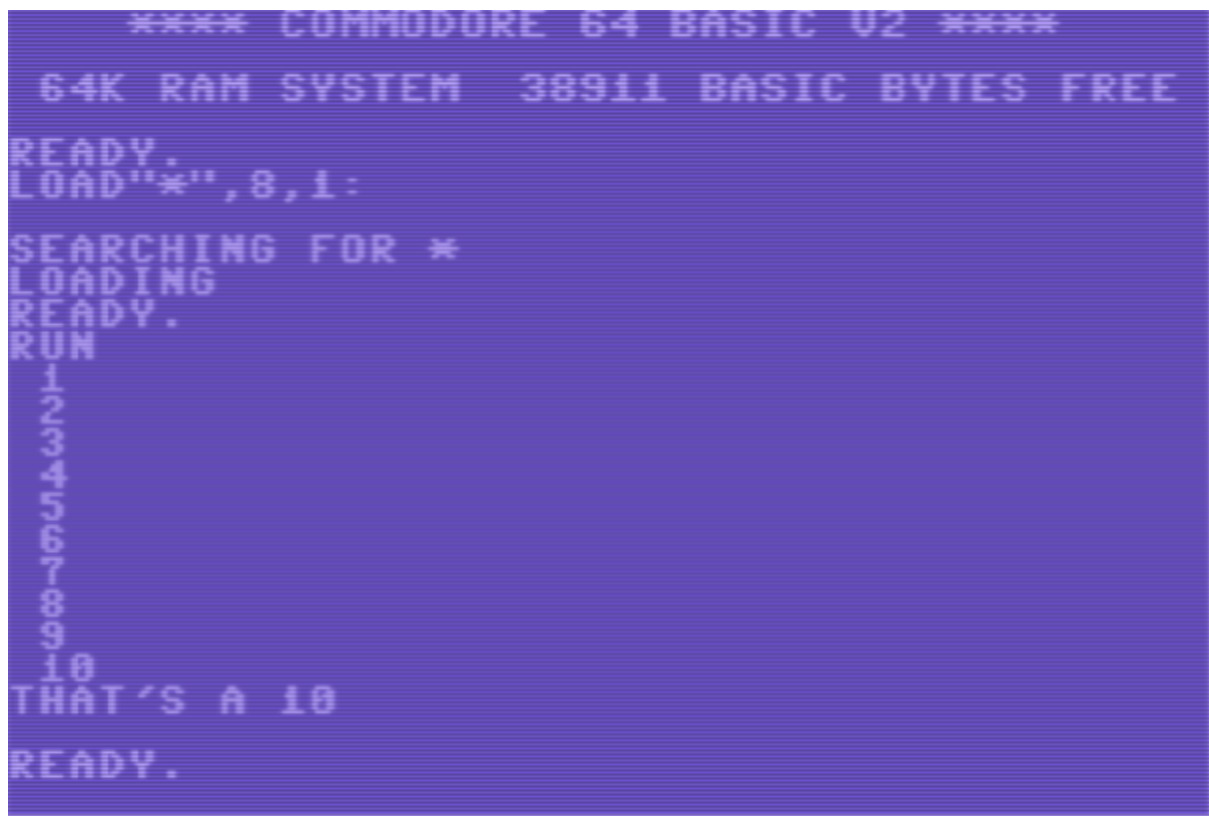
Figure 3

to is reached, Look at Figure 2 for a better idea on how this is structured and Figure 6.3 for the code execution of Figure 6.2. Nested for loops are possible, The FOR and NEXT structure just has to be repeated within the outer FOR structure.

What BASIC doesn't have is a While loop construct but that functionality can be obtained, simply by having an IF statement that fails on the first go, have the code you want to execute, Finally at the end have a GOTO back to the IF statement where when the condition is met you go to a line past the GOTO that was part of our "while" loop. Figure 6.4 is an example of this "while" loop. What this code does is print the numbers 1-10 while A !=10. Once A is equal to 10 the loop ends and we're done. You can see the result of this code in Figure 6.5.

```
10 REM While loop
20 A = 1
25 PRINT A
30 IF A = 10 GOTO 40
32 A = A+1
34 PRINT A
35 GOTO 30
40 PRINT "that's a 10"
50 END
```

Figure 4



```
**** COMMODORE 64 BASIC 02 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD"*",8,1:
SEARCHING FOR *
LOADING
READY.
RUN
1
2
3
4
5
6
7
8
9
10
THAT'S A 10
READY.
```

Figure 5

The GOTO statement can be considered the strongest part of C64 BASIC. It allows you to jump from one line to another. It allows the programmer much greater control over their code, but

BASIC doesn't watch out for you. If you jump to a line that doesn't have anything to do with what you're currently doing but the code is still executable than you won't get an error. This leads to what C64 doesn't have over modern languages and that's error messages. Modern languages also have hard to understand Error messages with their compilers but BASIC more so. The most you'll get is a line number and an error that the syntax is incorrect otherwise if that isn't an issue and your still having problems then you'll have to figure that one out on your own.

Interestingly enough BASIC does have the ability to define functions with "DEF FN" The best way to understand how this works is to look at an

```
10 DEF FN FTE(X) = X*3
20 INPUT "type in a number:"; A
30 PRINT FN FTE(A) : PRINT
```

example as seen in Figure 6. 6. In it I define a

Figure 6

function FTE(X) that multiplies the parameter

passed by 3. You can even use define a function using other functions (C64 Wiki). This great to use if a programmer knows they have a mathematical operation of some kind that is going to be used a lot in their program.

I have mentioned already that BASIC does use line numbers as a part of the program and as any modern programmer may have noticed, line numbers aren't really used in this way in modern programming languages and any Language that does have a GOTO nowadays usually uses a label instead of a number for a GOTO statement. In many of the programs, that has been shown so far the line numbers have been in multiples of 10. Some of the programs have lines in between these multiple of 10. That signifies that these lines were included after the lines that have a multiple of 10. When writing code for languages like BASIC it's considered good practice to write line numbers in multiples of 10 in case you have to add more code in between those lines later. Why? The reason is because if you don't then you can't add more lines later. You can't add a line in between lines 1 and 2. BASIC doesn't increment lines for you that's all manual. If you mess up and didn't account for how many lines you would need then you have rewrite every single line number and probably every GOTO statement as well. CBM prg Studio automatically increments in multiples of 10 for you but even then if you use up lines 11-19 and you need to add another it's required for the user to rewrite those line numbers for every line after to account for that extra line you needed to insert. A variant of the GOTO statement is the

GOSUB Command. These work functionally the same as a GOTO command. The difference is that it's meant for subroutines. Basically, the way

you use this is you specify a far of line number one that you're sure that your code won't reach like line number 1000. Then after writing the code here for however many lines, than the RETURN keyword is used. The RETURN

keyword brings you back to the line right after

```
10 REM For Loop Example with subroutine
20 FOR X=1 TO 5 STEP 1
30 PRINT X". number should increase"
35 GOSUB 1000
40 NEXT X
50 END
1000 PRINT "this is a subroutine"
1010 RETURN
```

Figure 7

GOSUB was called, look at Figure 6.7 for a better idea of how this looks. Subroutines created with GOSUB are put on the stack and while you can call more than one subroutine, they will be done in the order they were called and you risk running out of memory if you call too many at once. C64 BASIC functions while useful doesn't serve the same functionality as functions in a language like Java. In BASIC, subroutines with GOSUB are analogous to functions in Java. Since there aren't names though you have to remember what line the subroutine starts on whenever you need to use it. If you have a lot of subroutines it can become difficult to tell which one is the one you need to use. We recommend using the REM command in the first line of your subroutine that states its purpose. That way it becomes much easier at a glance to differentiate between subroutines.

As we've discussed throughout this paper many of BASICs functions are associated with specific locations in memory and in some cases you need to free up certain locations in order to accomplish a task, like what we did in our graphics example. In regards to memory management BASIC a bit unique and we can

see that with memory

management with arrays. Figure 8 gives an idea on how arrays are

made. Arrays are created like this in BASIC "DIM S\$(N)" (N is the

```
100 INPUT "HOW MANY STUDENTS";N:DIM S$(N)
110 FOR I=1 TO N
120 PRINT "STUDENT";I;:INPUT S$(I)
130 NEXT I
```

Figure 8: From the C64-wiki Array Page

size of the array) DIM is unnecessary for arrays with 10 or fewer variables as all arrays in C64 BASIC are created with a size of 10 by default. Any array bigger than a size of 10 requires the

DIM command so it can set aside its memory and dimensions. The C64 supports array dimensions up to 255 though you are likely to run out of memory if one was to create an array with that many dimensions. Memory for arrays and variables are stored in locations starting at 2049 and ends at 40960. While BASIC will handle allocating memory for variables and arrays on its own. You can't free up memory on your own or if there is a way it isn't obvious to do at least for arrays and variables. Like Java, BASIC has a garbage collector, though it's only for strings as they are dynamically allocated. BASIC starts up this process on its own usually when string memory is about to run out. Though for programs with large amounts of strings the C64 may appear to be frozen as it can take up to 20 minutes to free up space (C64-wiki).

C64 BASIC requires a lot of planning beforehand before you start coding as it's a big timewaster if you miscalculated how many line numbers you needed. Which is why combining programs together really isn't possible in BASIC. Modern Languages you can extend or inherit another program to gain some of its functionality or one could even copy and paste the code. BASIC can't do this at all really. With CBM prg Studio copy and pasting a small bit of code is possible but you have to rewrite line numbers, and even then it's not like you can call a function that it used before it was created so you'll have to move to the start of the program. If the code was just pasted in them there is no way for it to interact with your code besides a few GOTOs but at that point it's better to just use the two programs you wanted to combine as a reference for a new program with the functionality of both because then you can account for the amount of lines you'll need and it'll just lead neater code in general.

Conclusion

Based on the research conducted we gained a better understanding of the BASIC programming language and the Commodore 64. We learned about different aspects of BASIC by exploring the six topics we choose. We learned about the limitations and the benefits. For future investigations getting the ability to transfer code from CBM prg studio to actual hardware would be nice as that was an aspect that we wanted to cover but weren't able to due to costs associated with that. Another avenue for research is Assembly, we didn't look at it in depth but as we discovered it has some exclusive features so having Assembly and Basic work side by side would be an interesting research project. Working with a language from the early 80s showed us not only how languages developed and the hardware that run these languages evolved over time. It also showed us that the waterfall model made sense for the time as BASIC and similar languages required thorough planning to build programs. While hardware is still important today, C64 BASIC requires the programmer to know the ins and outs of both the language and the hardware in order to create a complex program and so it created a fun research environment for the entire group. While it's a language we wouldn't want to work with today, BASIC made us appreciate the design philosophies of modern languages and how they support the programmer.

Links

Here are the links to the software we used in this paper:

<https://www.ajordison.co.uk/download.html> - This link takes you to the download page of CBM prg Studio.

<http://vice-emu.sourceforge.net/index.html#download> - This is the Commodore 64 Emulator we used. We use the latest binary distribution.

<https://www.7-zip.org/> - You may need a program to extract some of the files. A free alternative is 7zip which can be found at this link.

<https://www.c64.com/> - Not covered in this paper but contains many free games and animations for the C64 that can be used with the emulator. This is a great site to get lost in as it shows you how versatile the C64 actually is with the programs here. I recommend the Redefinition Demo by Offence and Farlight.

Works Cited

Commodore Machines Inc. Commodore 64 Programmer's Reference Guide. First ed., Commodore Machines Inc., 1982, [http://www.classiccmp.org/cini/pdf/Commodore/C64Programmer's Reference Guide.pdf](http://www.classiccmp.org/cini/pdf/Commodore/C64Programmer'sReferenceGuide.pdf).

This is the programmer's reference guide to the Commodore 64 and it looks like it contains most of the information we need for this project. It covers all of our topics to some extent so this source will be the main one we will use.

Zimmers. "The Commodore 64." Zimmer.net, <http://www.zimmers.net/cbmpics/c64s.html>.

This source just contains the links to the C64 User's guide and reference manual. As well as other official material.

The Commodore 64 Users Guide. Commodore Business Machines (Uk), 1984,

This source is for the C64 users guide which we obtained from the Zimmer site. It includes some information not found on the programmers reference guide. So it should be useful for information on the C64 in general.

Mathhews, Ian. "Commodore 64 – The Best Selling Computer In History." Commodore Computers, Running Technologies Inc, 29 Nov. 2018,

<https://www.commodore.ca/commodore-products/commodore-64-the-best-selling-computer-in-history/>.

This source contains a summary of several other sources on several aspects of the Commodore 64. We'll be using this for mainly to describe the hardware in a bit more detail than what the reference manual and user's guide describes it as.

“Main Page.” C64, 2019, <https://www.c64-wiki.com/>.

This site has a lot of info despite being a wiki. Sp, we'll use this when other sources aren't available