So You Want To Crack and Train Commodore 64 Games Like The Pros?

Note: This article does not cover any 'illegal' topics. It is merely a tutorial on how to capture memory, save it to disk and manipulate it. I will not be discussing bypassing copy protection. This article is meant as an entry level tutorial as to how one can capture single file disk or tape games, and save them to disk. If any copy protection is in the game, this tutorial will not be of any assistance. I will also cover how to train games for infinite lives, etc.

It's not an advanced tutorial such as setting break points because I'd like readers to understand the basics first. There are more advanced methods as I'm sure crack groups can attest to.

Table of Contents

Tutorial written June 2015 by Mr. NOP (Canada)
Creator of Oil's Well and Ladybug for Commodore PET
nopsoftware@hotmail.com

PDF by Sauhund/F64
21/06/15

1.0 Introduction

This article is by no means meant to be a complete comprehensive guide to capturing games and saving the data. You can't put years of experience into a single document and expect a reader to be able to become the next Eagle Soft, Hokuto Force or Ikari.

I've attempted to use layman terms and analogies that some advanced programmers might find amusing, but we've all been at the beginning stage of trying to learn how to program or crack 6502 code – and layman explanations would have been helpful back then. Explaining things to you in simple terms may help you to understand the concepts better than if I dived into full technical jargon.

You may skip to section 1.2 if you're easily confused (or very lazy) but I recommend that you attempt to learn all sections. In fact you can skip all of the sections that you don't understand and probably still be able to have some success in capturing game memory and training. I'm going to try to write this article so that you don't have to know everything from the previous chapters.

Finally, we're not going to cover actual 'cracking' methods so perhaps the article title is misleading. Before you can learn how to crack, you need to know how to capture the 64's memory to disk without freezing. If you freeze a game, that doesn't allow you to place a trainer in the game or a high score saver, etc. Saving the memory to disk however does allow you to add extra features such as cheats and high score savers, etc.

The Commodore 64 contains 64K of memory. One K or (kilobyte) is 1024 bytes. A byte is simply one location of the C64's memory. For example if you typed "HELLO" on the C64's screen, that would be 5 bytes. Type the word "AWESOME" in the top left of the screen and you've used up 7 BYTES.

So one character put into the C64's memory uses up one BYTE. And you can place one character (or one value) in one memory location. You can place a letter A in the top corner of the screen but you can't place two characters there at the same time. One memory location, one value placed into that location. The screen consists of actual memory locations that are part of what we call SCREEN RAM, but all you really need to know is that one memory location can hold one value. If it's a character on the screen, that's one byte – if it's one character that's part of your name in a high score table, that's also one byte. If it's one note played through the SID chip, that's one byte. If it's one letter in your word processor document, it's one byte.

The Commodore 64 contains memory locations numbered from 0 up to 65535.

Proof of this is if you should type into the Commodore 64 the following:

```
POKE 65535,1
```

it would work whereas

```
POKE 65536,1
```

results in an "illegal quantity error" because there is no 65536th location on the C64.

And it's no coincidence that 65535 divided by 1024 equals 64. In other words, the Commodore 64 has 64 kilobytes of memory – or 64 blocks of 1024 byte memory locations. All you really need to know is that there are 65535 locations available to you the cracker, or to you the programmer. You don't need to understand the Kilobyte part of it.

So when it comes to learning to crack games, creating trainers or dumping tape games to disk, we need to keep this in mind – no matter what protection scheme is used, no matter what we're trying to cheat in a game (lives, sprite collision, level skipping) it will always just be a matter of changing one or more of these memory locations.

A single file game or a 10 disk game, will still work the same way – data will be loaded into the memory and it will be executed. Change the code and you change the way it subtracts a life. Change the code and it decides not to crash if it doesn't find the copy protection.

We know that the C64 has RAM and ROM memory. The RAM is memory you can change and you lose that data when you turn off the C64. The ROM is memory you cannot change and it remains intact when you reset the C64.

For example, the programming that allows you to type

```
LOAD "*",8
```

have to come from somewhere. And those load routines are found in Kernel ROM. The SID chip is ROM otherwise you'd have to load in some form of sound software every time you turned on the C64. You wouldn't even be able to see a character set on the screen were it not for a character set ROM which defines what the letters look like. How does the C64 know what pixels to light up to create the letter "A" compared to a letter "B"? All of this information is contained in ROM (character ROM to be precise).

The layout of the C64's memory is what we refer to as the 'memory map' and it looks like this:

Locations 0 to 40959 are RAM. You can change these locations contents to any value you want. Locations 40960 to 49151 are BASIC ROM. The Basic language resides here, which understands what you're trying to do when you type `PRINT "HELLO"` Locations 49152 to 53247 are RAM. You can change these locations to anything you'd like. Locations 53248 to 57343 are Character ROM. The actual character set you get when you look at the c64's screen. It also holds the colours for what is displayed on the screen among other graphic purposes. The sound chip also uses this memory. Locations 57344 to 65535 are the Kernal ROM. These are the built in routines that allow disk loads, tape saves, flashes the cursor, etc. It is the heart of the C64 so to speak.

Now that I've confused you to hell and back, allow me to make sense of things:

The C64's memory isn't just a single chip. It's a series of chips such as the Kernel, the SID chip (music), the Video chip and the RAM chips. These all serve to offer you 65535 locations in which you can place your information.

My first analogy – I'll be using a lot of analogies just to forewarn you.

Think of the C64's memory as a block of houses on your street. Your first house is Mr. RAM's house, the next house is Mr. BASIC's house, followed by Mr. Char Rom's house, followed by Mr. Video's house and finally Mr. Kernel's house.

Together these houses make up your entire 64K of memory. When you want to write a program in BASIC, your program gets put into Mr. RAM's house. It gets cleared when you turn off the C64. Mr. BASIC supplies the interpreter that allows the C64 to know what PRINT means, and what INPUT means, etc. Mr. Video allows you to change the colour of the screen in your BASIC program when you type `POKE53280,0`

Mr. Kernel's house provides the routines that flash the cursor, or load and save data from devices. All of these neighbours of yours are used by the C64 when you write in BASIC. You don't need to know how they work, just that they do work. They allow the C64 to translate your BASIC program into a working program, to allow disk loads, to play sound effects, to display sprites, etc.

The characters on the screen never change in physical appearance because Mr. Char Rom's house defines what the characters look like (Character ROM). Mr. SID (who is a tenant in Mr. Video's house) plays his music loud and annoys his neighbours but he's the sound chip guy. If your BASIC game uses sound effects then you're using some of his stuff.

Mr. Kernel allows us to be able to use LOAD or SAVE commands in our BASIC program and the C64 knows how to do that – because Mr. Kernel's house tells the C64 how to be able to load and save. If Mr. Kernel wasn't there, using the LOAD command would result in a "Syntax Error". (The Kernel is a series of routines to allow for input and output from devices such as the keyboard, tape and disk)

So when you create a BASIC game, you can only use Mr. RAM's house. You can of course call upon the ROM neighbours to do things as I've mentioned (move sprites, read the joystick port, print text to the screen, etc.). As your program grows in size though, eventually you're going to reach the point that it hits the brick wall at location 40960 – where BASIC ROM is held.

And since that is ROM, it can't be used to store anything. So for this reason you can only use up to location 40959 for BASIC. Anything beyond that is off limits…. Even if you could write to the memory at location 40960 and onwards, you'd be overwriting BASIC itself which would render your BASIC program a bunch of meaningless words to the C64. PRINT? What does that mean? What is a POKE command?

The television advertisements told us that the C64 was a 64K computer so we'd better be able to put 64k worth of information into it. But the reality is that the Kernel, Video chip, Sound (SID) chips are all ROM and you can't change what's in ROM. So did Commodore lie to us? Not exactly.

You can "page out" or "switch out" this ROM for RAM. That is, you can actually have a full 64k of RAM by telling Mr. Video, Mr. BASIC ROM, Mr. Kernel to temporarily leave their houses so you can store data in them. But you can't do this with BASIC programs because as your program grows in size, the first house it would try to fill up with your program data would be Mr. BASIC's.

If however you decide to write a game entirely in machine language (6502), then you don't need Mr. BASIC to be in his home, you can evict him for the day and use his house for storage. You can also tell Mr. Char to get lost for the day and use his memory. You can also do the same for Mr. SID and Mr. Kernel provided you don't plan to play music or use Kernel routines.

Summary

If you only want to write a program in BASIC you can use up to location 40959. Anything from 40960 up to 65535 is considered part of the ROM and you can't change it. If you want to write entirely in machine language then you can tell the ROM houses to leave for the day through a process known as 'switching out' the ROM. This makes all the memory of the C64 available to you – at the price of not being able to use the ROM functions because they are now empty RAM awaiting your needs. You've evicted the home owners for the day, and are using their houses for RAM storage.

6502 Machine language is the native language of the C64. It doesn't need BASIC to interpret it. We can turn off (or switch out) the ROM memory if we want, if machine language programs want to use it.

Some games you'll be looking at may only use RAM memory (and never go beyond location 40959). Other larger games might use memory where ordinarily ROM would be. This is because the game designer chose to use these ROM houses for storage – they decided to switch out the ROM to make RAM.

In the end it doesn't really matter if you understand this concept – but it would certainly be beneficial. Some games that were released by crack groups never worked properly (you can still find some bugged ones on Gamebase) because the cracker only saved the true RAM memory up to location 40959. They neglected to save the additional RAM memory that was underneath the ROM because the game programmer told those "houses" to leave for the day so he could use their houses to store his game.

It's possible to switch out some ROM while leaving other ROM intact. For example you can use the BASIC ROM at 40960 as RAM, and leave the Kernel ROM intact. We'll cover this later on.

For the purposes of dumping game memory and training, just understand that you don't just want to save BASIC RAM, you also want to get what might be 'hidden' or switched out under the ROM.

1.1 Binary

Memory on the C64 ranges from 0 to 65535 however when programming in machine language we use what's known as hexadecimal (or 'hex').

This can be a bit of a learning process to understand but it works like this: When programming in machine language, numbers in the C64 range from 0-9 and then changes to numbers:

```
10=A
11=B
12=C
13=D
14=E
15=F
```

There is no hex value higher than 15.

You can only put numbers from 0-255 into a memory location. If you doubt me, try typing `POKE1024, 256` and see if that works. It won't because 256 is too large of a number. This is an 8 bit machine and 8 bits = numbers of 255 at the most.

To convert decimal numbers into hexadecimal we break numbers into a two letter value. The first letter is a unit of 16 and the second digit is the remainder.

For example the decimal number 00 in hex is 00.
The decimal number 8 in hex is 08.
The decimal number 16 in hex is 10.
The decimal number 42 in hex is 2A.
The decimal number 255 in hex is FF (which is why we don't need a 'G', 'H', etc.)

Take the number 42 for example… how does 2A become a value of 42? Well 2 times 16 = 32 and add ten to it (A=10) and it becomes 42.

We'd arrive at FF for the decimal number 255 because 15 (F) times 16 = 240 and the remainder is 15 (F). 240 + 15 = 255

You can use your Windows calculator to convert between decimal and hex. It will take a while to memorize the more common values. For example most programmers know that 2A is the asterix (decimal value 42) in machine language. The space character is 20 (2 *16 = 32).

We're getting into hexadecimal values because when we examine machine language, the values are all displayed in hex. There's no decimal being used.

We use this process when determining what number to put into a memory location on the C64.

Now as far as the actual memory locations in the C64, because they are significantly larger than 255 numbers, the two digit hex formula doesn't work here. We need a four digit formula but it works much the same.

A hexadecimal value representing the C64's memory looks like this: `C000`

MUST KNOW: The first digit is multiplied by 4096, the second by 256, the third by 16 and the last one is the remainder. It's an extension of the two digit hex formula.

So location 49152 in decimal is really C000 in machine language because we know C = 12, and 12 * 4096 = 49152.

1234 in hexadecimal is 4660 in decimal because 1*4096, 2*256, 3*16 and adding 4 = 4660.

I'm not going to get involved into how to convert decimal to hex and vice versa – if you're having difficulties use the calculator built into Windows.

Wrap Up Summary

The C64 has 65535 locations in which you can place information. Not all of these are available for BASIC programs because the C64 needs some memory to serve as the "BASIC interpretor", to provide sound, video (screen colours, sprites, etc.) and Input/Output functions. These functions are stored in ROM chips in the computer.

We can turn off/switch out/page out (basically all the same meaning) the ROM functions to create temporary RAM provided we're not going to be doing our programming in BASIC. BASIC needs the BASIC ROM memory to function. If you're using your own custom character set you can switch out the Character ROM. If you aren't using Kernel routines (disk saving, etc.) you can switch out the Kernel ROM.

Of the C64's memory locations we can put numbers from 0-255 in them. In BASIC this is done by typing:

```
POKE (location), (value)
```

example:

`POKE 1024, 42` places a star in the top left of the screen
`POKE 53280,1` places a value of 1 in the VIC-chip memory (the border colour)

In machine language we use a formula known as hexadecimal when reading and writing to the memory of the C64. It's the equivalent of the BASIC POKE command.

Values from 0-255 in hexadecimal translate into 00-FF Memory locations translate into 0000 up to FFFF (0-65535 in decimal)

We use a maths formula whereby we multiply:
For two digit numbers (what you'd be placing in memory locations) it's: XX where the first X is * 16, and you carry (add) the second X value.
For four digit numbers (memory locations) 4096, 256, 16 and carry the remainder. So 49152 is really C000 (c=12, and 12*4096=49152)

If you're confused as to why some are two digit and others are four (I know I was when I first tried learning 6502), here's a simpler explanation: The C64 only needs 2 digit numbers when placing information into memory locations because we're never using numbers higher than 255. You can't use values such as 256 or 1000 as in `POKE 1024, 256` or `POKE 53280, 1000`.

Because memory locations can, and do, go beyond 256 though, we need a four digit number to represent the memory location in hexadecimal.

The memory map of the C64 in hex looks like this (the same as the one I gave you earlier but represented in hex this time):

```
0000-A000 RAM MEMORY
A000-C000 BASIC ROM (*)
C000-D000 RAM
D000-E000 CHARACTER SET ROM (*)
E000-FFFF KERNAL ROM (*)
```

(*) means that this can be used as temporary RAM if you're not planning on using BASIC.

Now you might have seen some games written in BASIC that also contained portions of machine language. Perhaps the machine language was for playing music or displaying a scrolling background. The most common area for this machine code to go was 49152 (or $C000 in hex). This is because there's a little window in the A000 to FFFF ROM that is still usable as RAM. Everything else above A000 is considered unusable for placing BASIC code. The window of C000 to CFFF is usable though. I'm sure we've all seen a `SYS49152` command in a BASIC game. This is why, it's away from BASIC and in no danger of being overwritten by your BASIC program.

1.2 The CCS64 Emulator

For the purposes of dumping memory and training, I'll be looking at using CCS64 emulator. This is available online at http://www.ccs64.com

I'm not going to be using Vice because I find the machine language monitor built into it to be not very friendly. If you're using a REAL C64, you will absolutely need a freeze cartridge such as the Action Replay or Final Cartridge. It's essential that you have a cartridge that has a built in machine language monitor that you can 'jump into' at any point. Without one, you'll have to do all your cracking and training through an emulator.

A machine language monitor is a tool to examine the C64's memory and to write to that memory. It's no different than the POKE command in BASIC. In fact some game cheats of the past were published as POKE's that you entered in before RUN'ing a game.

A machine language monitor allows you to halt a program, enter the C64's memory and change the memory, then resume the program. For example, you can break into memory while playing Pac Man and decide to turn off all the bad guy sprites so that you have the maze to yourself. Then you can resume the game. You could break into a machine language monitor and fill the screen with blank characters so that once you exited the monitor, you're joystick controlled man (or ship) can move anywhere – now that there are no walls or barriers on the screen.

2.0 Learning to use a MLM

MLM = machine language monitor, the tool that allows you to view and change the C64's memory as a program is running. I'll be using the term MLM from here on in.

For the purposes of this article we're only going to focus on tapes and disks that are one file (or multiple files that load just once before the game begins). When we get into multiple disk games, multiple file games or multiple file tape games, there are other considerations that are just too extensive to cover at this time.

Essentially we're just going to be looking at 'capturing' a game in memory so that we can save it and play it. If it's a single file tape, the game should play fine. It's not really a crack if it's not doing any copy protection checks, and single tape files rarely do. There are too many scenarios to cover, to cover all disk copy protection methods so again we're only going to capturing the game and saving it.

Once you've learned how to 'capture' a game, and save the memory to disk, you can advance to the 'Training' section and learn how to train games (cheat). We're not going to be 'freezing' games because freezing is not cracking, and it's a practice generally frowned upon.

So… we know that there are 65535 locations of memory that could be potentially loaded with data. How do we crack or capture a game? How do we know what area of memory is actually being used? We'll discuss this later.

The first 256 bytes of memory in the C64 is referred to as "zero page" memory because it's the first page, the first section, the beginning, of the C64's memory. These locations are most often the area used by games to hold temporary data and we'll discuss that in the 'Training' section. It's not common for games to actually load themselves into this area, but they frequently use this area while you're a playing the game.

TAKE NOTE: Hexadecimal is represented by a dollar sign ($) to differentiate it from decimal numbers. So if I write "20" I mean 20. If I were to write $20 I'd mean a hex number (in this case decimal value of 32 because 2*16 = 32).

Therefore zero page in decimal is 0-255, and in hex it's $0000-$0100 (because 0*4096, 1*256, 0*16, plus 0)=256

So if you typed FOR I = 0 TO 255: POKE I, 0 : NEXT you'd wipe out all of zero page (and cause a crash) in machine language this would be the same as filling in $0000 to $0100 with $00's.

I know this is probably overwhelming – I once struggled with the concept of how to print just a single character to the screen in machine language. This is why I'll repeat myself and use analogies.

There are several memory maps online that you can Google, that will explain the C64's memory layout to you. The main areas to know (print this out and tape to your wall):

```
$0000-$0100 zero page
$0400-$07E7 screen RAM (what you type on the screen is held in here)
$0800       start of BASIC (all basic programs load from this location upward)
$9FFF       end of BASIC (all basic programs stop at this point)
$A000-$C000 BASIC ROM (this is BASIC, the actual language)
$C000-$D000 RAM (free to use for any purpose)
$D000-$E000 CHARACTER SET ROM (what defines your characters)
$E000-$FFFF KERNAL ROM (Special functions for I/O devices, etc.)
```

In most (but not all) cases, once a game has loaded, and just a split second before it starts to run, you can get away with saving memory from $0400-$FFFF (which would be 1024 decimal to 65535 decimal) and you'd have all of the game code you need. Most games don't touch below $0400 for purposes of loading, but they may use this memory once you're playing them.

So…. Let's begin shall we.

2.1 Switching out ROM

First, I'm assuming you've installed CCS64 by now or you have a real C64 with a cartridge.

I also assume that you know that the memory from $A000-$FFFF is mostly ROM but if needed, it can be called upon to serve as RAM if you want to turn off or "switch out" the ROM.

1) Start CCS64 (or turn on your c64 and enter your freeze cartridge and find the Machine Language Monitor option)
2) In CCS64 press ALT-M for the machine language monitor.
3) In both cases you should now see a prompt and should be able to type commands. Most commands are universal as it pertains to machine language monitors:

```
L - load
S - save
F - fill
H - hunt
M - view memory
I - visual display of memory
X - exit
```

Whether you're using Vice, CCS64, or a cartridge, try typing in:

```
M 0400
```

Remember, we're talking hex now – the decimal world has been left behind. So $0400 is 1024 decimal which we know is screen RAM. That is, if you look at $0400 you should see whatever was on your screen before you started the monitor. If you had typed your name into the upper left corner of the screen and then entered the machine language monitor (heron referred to as MLM) you'd see your name at $0400.

Let's try it…

Press 'X' to exit the monitor and type your name in the top left of the screen then re-enter the monitor (ALT-M on CCS64, pressing the cartridge button on a real 64) then type M 0400 0420
and you should see your name in the memory dump

The MLM is displaying the RAM memory to you…. But this is also a two way street. You can also write to memory from within the MLM. Remember, a machine language monitor allows you to not just inspect memory but also write to memory.

So on a new line type in "`F 0400 07E7 2A`" (without quotes) this is the Fill command and fills in $0400 to $07E7 with the value of $2A. The format of the Fill command is : F (start address) (end address) (value)

Going back to the section on hexadecimal, $2A (2*16 plus A [where A=10] is 32+10 or 42) and $0400 is (4 * 256 = 1024) and 1024 is the top left of the screen (POKE 1024, 42 puts a star in the top left of the screen).

Type X and press return to exit the MLM. Now look at the result… the screen SHOULD be full of stars now because we've just FILLed in the entire screen memory with $2A's.

Let's try again….

Enter the MLM and type

F D020 D021 04

This will fill $D020 to $D021 with a value of $04 (4 decimal).

$D020 and $D021 happen to be 53280 and 53281, the border and screen colours. So when you press X to exit the monitor, the screen's colours should be changed.

And this is the way you crack and train games… by entering the MLM and modifying some of the 64's 65535 various locations. You can inspect memory but you can also change memory and when you exit the MLM, those changes take effect be it screen colour, screen RAM, sprite data, etc.

It's entirely possible to fill random areas of memory with zeroes and see what effect it has on a game. Maybe a ship disappears, maybe half of an alien disappears, maybe you overwrite the music, etc.

For fun, enter the MLM (ALT-M on CCS, or press the button on your cartridge and find the built in monitor)

Now type:

```
F 0000 FFFF 00
```

This will FILL $0000 to $FFFF with 00's (fill the entire 64's memory with zeros)

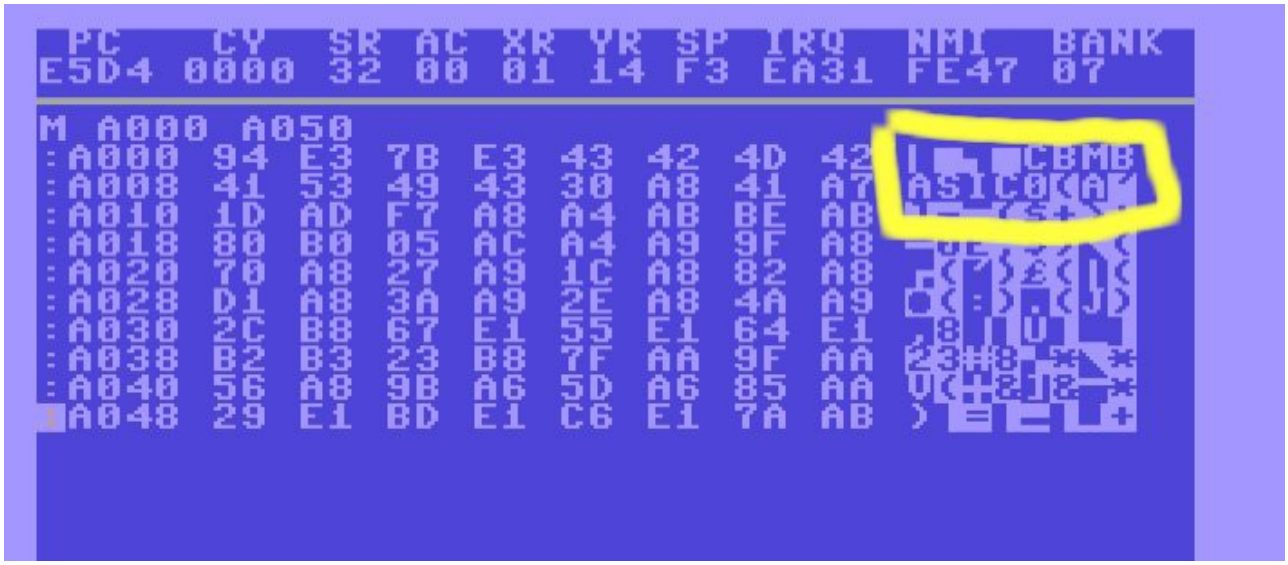
If your computer hasn't crashed yet, press X to exit the MLM and the 64 should crash. You went and filled in the video memory with zero's as well as the zero page memory ($0000 to $0100) and the C64 doesn't like that at all.

Restart your C64.

Now I want you enter the MLM and type

```
M A000 A050
```

To view memory from $A000 to $A050. This is Bill Gate's BASIC you're looking at. You can actually see the words "CBMBASIC" in the result.



Let's tell these people living in the ROM houses to take a trip for the day, so we can use their houses for RAM. In CCS64 you will type:

BANK 0

In other monitors you may have to change the value of location 1 to a $34. The most universal way I can think of doing this is to type

```
F 0001 0001 34
```

We're using the Fill command to fill location 1 to location 1 with the value of $34

IMPORTANT: Location 1 is very important to us. It tells the C64 which houses we want to keep using as ROM and which ones to page out and use as RAM. If we change the value of 1 to something different, we can use the entire C64 memory for our own purposes.

The hex value of $34 placed in location 1 turns off ALL the ROM chips. Since you read my section on hex, you know that $34 is 52 in decimal because 16*3 = 48 and adding the remainder of 4 = 52. So the BASIC equivalent of putting $34 in location 1 is going to be `POKE 1, 52`

Try it sometime … turn on your c64 and type `POKE 1, 52` and it will surely crash. Why? Because you entered a BASIC command that told the C64 to turn off BASIC (you told Mr. BASIC to leave his house and use his house for RAM). Naturally without a BASIC language, you can't proceed so it locks up.

So… back to the MLM. You've now typed in:

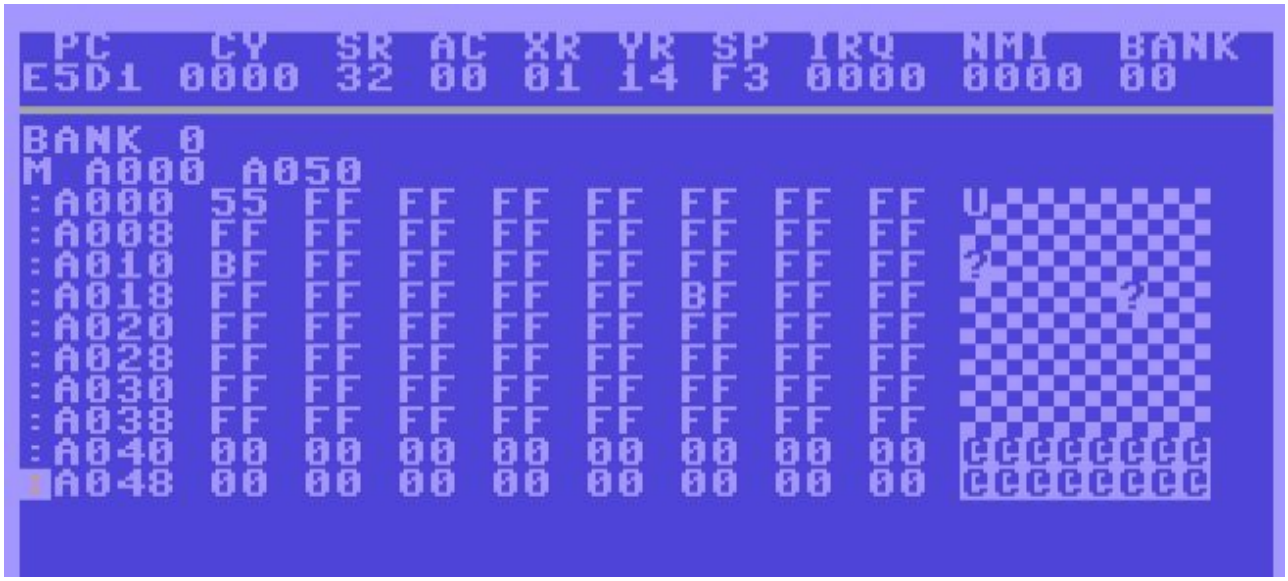
```
F 0001 0001 34
```

which is a simple way to change the value of memory location 1 to $34 hex. This effectively turns off all ROM and allows you full 64K of memory. If you're using CCS64 you would have typed BANK0 instead.

Now type in:

M A000 A050

And look at the results…



No longer should you see anything… no random characters, no "CBMBASIC" word. In fact as you scroll through memory, you see nothing at all because we've switched out the ROM for RAM. BASIC is now gone.

Try typing

```
M E000
```

It's also blank…. There's simply nobody home on the street in the $E000 house. All the ROM guys have left for the day.

The only reason I'm teaching you about turning off ROM is because when it comes to capturing C64 games in memory, often times you'll need to save the hidden RAM that the game might be using where ROM normally is (the houses on the street).

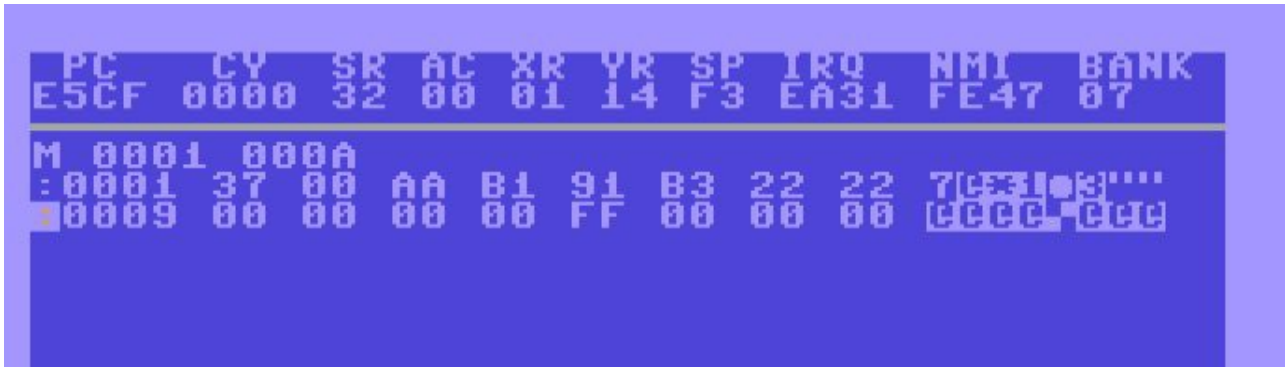
To turn off all ROM and make it RAM, we change location 1 to $34.

There's another way to change the value of 1 other than using the fill command. If you type:

`M 0001` (press return)

You should see something like:

```
0001: 37 00 AA B1 91 B3 22 22
0009: 00 00 00 00 FF 00 00 00
```



This would mean location 1 is $37, location 2 is $00, location 3 is $AA, location 4 is $B1, location 5 is $B3. You read it left to right. Each new line shows you eight memory locations. The very first number is the starting memory location for that line.

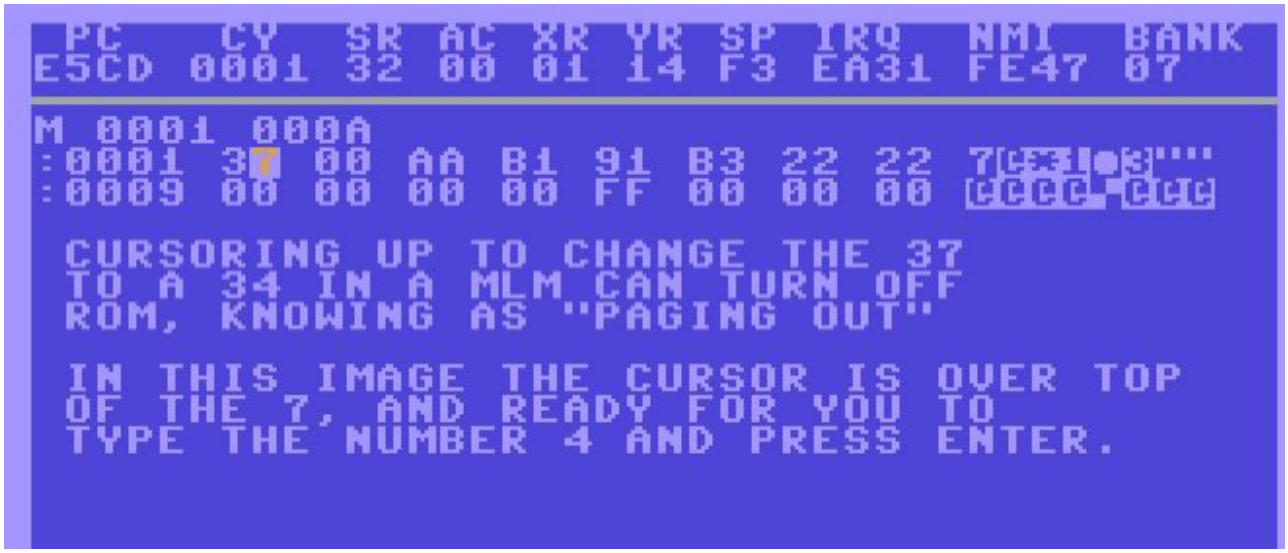
Line two would mean that location 9 has a $00 in it, location 0A has a $00. Each line is 8 bytes of memory.

In the event you type `M 0001` and the whole screen begins to scroll data up the screen too quick to read, it's because you need to set a limit on what memory you want to view. I suggest:

`M 0001 000A`

instead. This will say "I only want to see memory from location 1 up to location 10 ($0A in hex).

You SHOULD be able to scroll the cursor up and type over top of the 37 and change it to a 34. This should work in Action Replay and other cartridges. If you exit your cartridge and return back to the READY prompt, it might crash though. This is because you've told Mr. BASIC to leave for the day – and without BASIC, your computer is a dumb terminal that doesn't know what to do. So always remember to change the value of location 1 back to $37 before you exit to BASIC. CCS64 users don't have to do this, the BANK command works great for this because you can exit without changing the value of 1 back to $37.



Summarizing so far

The C64 contains RAM and ROM memory.

Basic programs that we type in, begin growing in the C64's memory at $0800 (looking at the memory map I told you to print out). As we add more and more lines of code to our BASIC program, that memory gets used up until it hits the brick wall at $9FFF. It's a brick wall because $A000 is the BASIC language itself stored on a ROM chip. Even if we could overwrite that memory, we'd be overwriting BASIC which would leave our program a series of commands that meant nothing.

In machine code though, we aren't running any BASIC program and so we can use memory from $A000 right on through to $FFFF. To do this, we change the value of location 1 in the C64. We know that if we change the location 1 to a value of 54 decimal ($34 hex) that we do away with ALL ROM. Naturally we cannot change the value of 1 to a $34 and still be able to type things like PRINT or LOAD on the screen though because that's the job of BASIC to interpret what we typed. So if we do change the value of 1, we need to set it back to it's normal value of $37 when we're done. That will allow us to exit our MLM and still be able to see the READY prompt on the screen.

It's the equivalent of telling Mr. BASIC to come back home to his house before we exit the MLM. Otherwise the C64 won't know what to do. Fortunately in CCS64 you can exit the MLM regardless because the BANK 0 command only unlocks our RAM until we exit the MLM.

TEST YOURSELF: Do you know how you'd enter your monitor, examine all the memory to see if anything was hidden under the ROM, and exit again without crashing if you had an Action Replay or machine language monitor cartridge? Try to figure it on your own before I tell you how.

Answer: First you change the value of location 1 to a $34 to open up the houses of these ROM guys by telling them to leave for the day ("Hey I want use you for RAM, see you later")

How? Type:

```
F 0001 0001 34
```

Then you can use the M command to view memory.

Type:

`M A000` (press return)

And you can scroll down until you reach $FFFF the very last memory location in the C64. Notice that most of that memory will appear as 00's and FF's which is the default blank values put there. Perhaps you'll see all 00's. This is unused area, it's empty.

Before you exit, put the value of 1 back to $37 by typing

```
F 0001 0001 37
```

$37 is the normal value of location 1 and it tells the C64 to bring all of the ROM guys back home into their houses. If you try exiting the MLM without changing the value of 1 back, it might crash the computer because Mr. BASIC is not home – and the C64 needs him to interpret whatever you'd be typing once you exited the MLM. Needless to say without Mr. Kernel either, there's no cursor flashing.

And again, typing 'x' and pressing return is how you exit the MLM.

We also know that in friendly MLM's like CCS, we can just scroll the cursor up and type in new numbers over top of the old ones. For example we can change that $37 to a $34 without using the F (fill) command. I use the F command because it should work in most variants of machine language monitors.

Quick Tip

If location 1 = $37 then all is normal. BASIC is working.
If location 1 = $34 then BASIC is turned off, you have 100% RAM, but can't type things like PRINT, etc. because BASIC is turned off.

Always set location 1 back to $37 if you plan to exit back to BASIC (except if you use CCS64 and the BANK 0 command)

2.2 The basics of crunching and linking

From here on I'm going to assume that you know how to:

- enter the machine language monitor
- understand that memory ranges from $0000 to $FFFF (or 0000 to 65535 decimal)
- vaguely understand hexadecimal
- Know that ROM can be turned off temporarily to allow us full 64K of memory
- Know that some games may or may not use ROM memory ($A000-$FFFF)
- Know that location 1 when it is set to $37 is normal, and $34 means that all memory is RAM, no ROM.

Most single file games are compressed. Compression makes games smaller in size. For example let's say that a game had all of it's game code at memory $0800 to $4000 (that's 2048 to 16384 in decimal). Now let's also say that the game had music at $C000-$CFFF. In order to save all of this game memory to disk, using our machine language monitor we'd have to save from: $0800 through to $CFFF right? We need both the game code and the music but they're at different 'ends' of the C64's memory. One is down low, the other is up high in memory.

We need to capture both $0800 to $4000 for the game code and then get the music at $C000 to $CFFF. But what about that large gap from $4000 through to $BFFF?? That's a lot of empty space… wasted space (128 blocks of disk space). Many utilities exist that will allow you compress that large amount of wasted space into a tighter smaller file. It does so by recognizing the repetitive characters in that wasted space. For example if $4000 to $BFFF is blank memory filled with zeroes, the compression program can reduce that 128 blocks of disk space down to less than 1 block. It does this by crunching the repetitive data and writing a little 'decrunch' routine that unpacks everything again. It says "I know there are 128 blocks of empty space here, I need you to place that many zeroes right after the game at memory location $4000. Then put the music at $C000".

Another analogy…. You have 10 houses on a street. Each of those houses is RAM in the C64 in chronological order. You fill the first house with code for your game. The code flies a ship around the screen. You then use the last house for the game's music. In between the first and last house are 8 empty houses with nothing in them. When you go to save the memory to disk, you can't just save the first house. You also need the last house too. So you have to save the whole block of houses to disk… that's a lot of empty space in between.

Then you have a large file on disk that's over 120 blocks in size when you've only needed 72 blocks of actual memory. You decide to download a 'cruncher' that will make your game smaller. It works by realizing 8 of your 10 houses are empty (blank memory) and strips all that empty space out. It's like putting those 10 houses into a vice and turning the handle to squeeze them all into one small little house.

When your cruncher is done, you have a final product. A small file you can load and run. When you run it, using our vice/house analogy, it unsqueezes the small file back into the large 10 houses including the large gap of empty memory.

Crunchers work using a variety of methods – just know that it's capable of squeezing out all the repetitive characters in memory and putting them back in when you run the final product. Without a cruncher (or a packer or a squeezer) we might see large 200+ block games instead of 50 block games.

Now what if a game only uses memory from $0800-$4000 and that's it? There's no music at $C000. Well you wouldn't have to worry about saving any memory beyond $4000 or worry about empty gaps. Regardless, music, sprite data, character set data all contain repetitive characters that can be crunched. So even if there are no 'gaps' of unused memory in your program, there's usually something that a cruncher can do to make your game smaller.

Crunchers (also known as packers, squeezers) are used very frequently. Sometimes even store bought commercial games have been crunched with public domain crunchers.

(Side note: There are programs such as ECA Linker that will link multiple files into a single file. A better way than saving all the memory from $0800 to $CFFF would be to just save two files to disk. File "1" would be your game code from $0800 to $4000 and file "2" would be the music at $C000 to $CFFF. ECA Linker will not worry about the memory in between, it crunches only the files that you give it. However this is not covered in this article.)

Let's download the game Blue Max 2001
Here's the download link: http://tapes.c64.no/tapes/BlueMax2001.zip

Unzip the file and place the tape file into your CCS directory (or another folder). If you use a real C64, you'll have to transfer the tape image to your C64. This is a commercial game so be sure to delete the file after you're done with this tutorial.

If you have a real C64 then I would suggest for now you use the CCS64 emulator on your Windows machine or try to copy the tape file to a real disk drive.

Now lets look at the game blue Max:

Load it but don't run it (in CCS you'll have to attach a tape image, the same as you'd do under Vice emulator). You do this by pressing ALT & 1 and then selecting the tape image.

Type LIST and you'll see:

```
0 SYS2061
```

That's a BASIC program. It loads into the C64's memory at the start of BASIC which is $0800 ($0801 to be exact because $0800 is always a value of zero). You run it and you have a game to play….. this game has been crunched using a cruncher.

But what if the game programmer decided to put game data at $0800? How can the cruncher unpack the data and place it at $0800 without overwriting the uncrunch program that's currently already at $0800?

Another analogy… you have a program written in BASIC.

```
10 PRINT "HELLO"
20 END
```

You crunch it with a cruncher (why, I don't know why) and the final result is a file that reads:

```
10 SYS 2061
```

You run the crunched program and voilà!, there's your BASIC program again. You type `LIST` and your two line program is there but the `10 SYS 2061` crunched program is long gone.

Well what happened to the old program?

!!! YOU SHOULD KNOW THIS !!!

This is something you're going to have to understand…. Unlike some of the previous content, you need to understand how this works. Crunchers usually work in the same fashion. First, the final product is a program you can `LOAD` and `LIST` and `RUN`. It is a basic program with one line of code: an `SYS` command. `SYS` of course calls a machine language routine that's in memory. The purpose of the program being LISTable is that you can load it and run it like any other BASIC program. The code that does the de-crunching however is all machine code.

What happens is that when someone takes a large file from disk and crunches it, the output is a smaller file that you can then RUN. But no matter how much we crunch and squeeze, we still need all the original information back. We still need the cruncher to put everything back the way it was before we crunched it… squeezing out repetitive characters isn't a one way process. You don't know if all of that empty space is needed or not… maybe you have a very large graphic picture that's nothing more than a KoalaPad picture of the word "Hi" in the middle of the screen.

You could crunch that Koala picture, and all the empty blank space around the word "Hi" would be crunch-able because it's likely just zeroes in memory. But when it comes time to run that crunched Koala picture, unless you filled back all that empty space with zeroes, you'd have "garbage" or "static" hires all around the word "Hi".

Crunching isn't ever one way removing of redundant data, ever! Crunching is reducing while also putting EVERTHING back as it was after de-crunching. Think of your Windows computer and WinZIP or WinRAR. Do they make your programs smaller? Yes, but they also expand them back to their original sizes after you unzip them.

Still not following? Think of it like this: You have a word processor document with a story in it. You crunch the document. Pretend that the first line of your document reads as follows:
"**.....**..My Story**.....**."

The cruncher might crunch that down to "$eMy Story$e." but when you decrunch it, you want those repetitive periods back. This is what I mean when I say that crunching always puts back what it has crunched. It doesn't just condense, it also expands. You don't lose the original uncrunched data.

So in answer to how does a cruncher put data back as it was without writing over itself, this is the answer….

If you turn on your C64 and type:

```
10 PRINT "HI"
```

And go into your MLM (machine language monitor) and type:

```
M 0801
```

You'll see your BASIC program. It resides at $0801 which is the Start of BASIC as I've told you. ($0800 is the start of basic but $0800 will always be a zero, so technically your program loads at $0801)

A cruncher will provide you with a final product that you can LOAD and RUN because it has a line number -in this case of the game Blue Max it's:

```
0 SYS 2061
```

When you run the program, the cruncher copies a little bit of code down into lower memory (most often $0100 or the $0300 area). This memory isn't often used by games. Once it's copied itself to that area, it then proceeds to go through everything it once crunched and puts it back into place.

This is why you can crunch data at $0800 and still be able to unpack it even though the crunched program loads in at $0800. By copying itself to lower memory, it provides itself a safe area in which to unpack the data, out of way.

Having difficulty understanding this? If the answer is no, continue to Section 2.4. If the answer is yes, read on….

Take a piece of paper and draw 5 boxes to represent the C64's memory.

```
[1] [2] [3] [4] [5]
```

The first box is the C64's memory from $0000 to $0800. No matter what program you load from disk, that area is not being used at this time because we know that BASIC programs always load in at $0801. Box 2 is memory starting at $0800.

Even if it's a single line program that calls a machine language game, it's still considered BASIC if you can LIST it.

So the second box (memory beginning at $0800) will always contain our program…. And depending on the size of the program, it might also use up the 3rd and 4th box of RAM memory. The final box, the fifth one, is the ROM memory. Our BASIC program doesn't ever touch that area. In fact we can draw a large X through the last box.

Now say that you have a very large BASIC program that uses the 2nd, 3rd and fourth boxes of memory. You want to crunch that program with a cruncher. So you crunch it and end up with a final product that's almost ONE THIRD the size of your original program.

Now… you load in your crunched program and type RUN. Your BASIC program starts right away. When you loaded in your crunched program, where did it load into? It loaded into the second box of course. All BASIC programs load into $0801. In fact the new file is so small that the third and fourth boxes aren't even needed. Your crunched file is so small that it now only occupies one 'box' instead of the original three, the rest is free memory.

Still…. With our crunched program in the second box, it needs to uncrunch into something that is your full blown BASIC program. A BASIC program that was so large that it used to fill the 2nd, 3rd and fourth boxes. Remember, crunching isn't a one way process, we need our original program back – the very large program that uses up three boxes of memory.

But how can we have our crunched program in box 2 AND overwrite it with our uncompressed program that also belongs in box 2 (and boxes 3 and 4)? Won't the crunched program and the uncrunched program overwrite one another in box #2? Yes.

The solution

When you run a crunched game, the cruncher copies part of itself into the box #1, the area BELOW where BASIC programs load. That little piece of code handles all of the de-crunching so that it can take your data and de-crunch it without overwriting itself. I'm sure you've seen programs that, when you ran them, the border changes colours, maybe you heard a bunch of static noise from the speaker, maybe some characters on the screen change, the entire screen might flash colours… this is the program 'uncrunching'. And it's uncrunching process is happening BELOW where BASIC is, so that it doesn't overwrite itself.

2.3 Stopping the de-crunch routine

When trying to crack a single file game, or training (cheating) a game or just trying to capture a game from disk or tape, it's not recommended to do so once the game has started. The game might overwrite part of itself once the code has been executed. Some memory locations might have been changed. You might not be able to find out what the 'start address' is.

We always want to save a game to disk AFTER it's de-crunched but BEFORE it begins to play. A crunched game is of no use to us, we can't examine the true data nor could we make a cheat for it, or add a high score saver, etc. It's all squeezed into a small file. We want the original game de-crunched. (uncrunched and de-crunched mean the same)

So how does one accomplish this? It can be impossible to enter your machine language monitor in the split second between the end of the de-crunching and the game starting. We need something more practical.

To accomplish this… we need to create a 'loop'. A loop that will cause the computer to go into an endless circle, doing nothing. We want the equivalent of typing:

```
10 GOTO 10
RUN
```

The process will usually be the same.

1) Load in the single file game.
2) RUN
3) Right away enter the machine language monitor (ALT-M in CCS64 emulator, or pressing the appropriate button on your cartridge)
4) Once you enter your MLM, there's something you need to pay attention to. There is immediately a series of numbers and letters on the screen:

It looks something like:

```
 PC   CY  SR AC XR YR SP IRQ  NMI  BANK
0231 0001 36 00 00 21 F4 0000 0000 00
```

These are very important. Much of the information shown is beyond the scope of the article but the IRQ and NMI tells us what 'interrupts' are being used. The AC/XR/YR shows us what's in the Accumulator, X and Y registers. But the one we want to look at is the PC.

PC means Program Counter. You know that when you run a BASIC program it starts with the first line number and works its way down. It might branch off for a `GOSUB` but it will come back on a `RETURN`.

```
10 PRINT “I AM ON LINE 10”
20 GOSUB 100
30 END
100 PRINT “I AM NOW ON LINE 100”
110 RETURN
```

Machine language works the very same way. It has a starting address (in BASIC that would be line 10), it can branch off like a `GOSUB` (line 100) and it can return (to line 30) like the `RETURN` command. By looking at the PC number, we can tell what memory location the C64 is currently processing. It’s like pressing RUN STOP and seeing `BREAK IN LINE 100` and knowing the C64 is processing Line 100 and then type `CONT` to continue the program.

PC is like telling us the line number, only it’s a memory location. It’s not all that different from BASIC line numbers.

I’m going to give you a trick, it doesn’t always work but it will work in about 50% of the games you try. In fact this is the trick I use all the time for my own cracks. I recommend that you use this trick all the time. You don’t even need to look at what the PC number is because this trick works so well. It will work with tape loaders, it may also work with full disk games where there’s only a single auto-run file.

The Trick

By now we know that some games will use that ‘hidden’ RAM that’s under the ROM chips. We know that if you set the value of location 1 to $37 or $34 that we can turn on and off this hidden RAM.

In any de-crunch routine, it’s going to turn off the ROM and turn on the RAM (it’s going to switch out the ROM). It has to do this because as it de-crunches, we know that we can’t just write on top of ROM. In our analogy speak – a decrunch routine is going to always tell the ROM guys to get out of their houses while we depack just in case our game needs to place code in memory originally used by ROM. We can’t write to ROM memory so we change location 1 to allow us to use the ‘houses’ as RAM.

So what if we look for the part where the decruncher turns ROM back on when its done uncrunching? That is… what if we look for the part where it says, “Okay guys I’m done uncrunching the game, I’m ready to start the game. I’m going to bring the ROM guys back to their houses and away we go.”

And for this, we look for the following 6502 commands:

```
LDA #$37
STA $01
JMP $xxxx
```

Usually the line JMP $xxxx is the start address of the game we want. It’s not always going to be that easy, but sometimes it is.

In English the above code means LOAD A with a value of #$37
STORE A in location $01 (turn our ROM back on)
JUMP to location xxxx (where xxxx can be anything)

This code is essentially is the same as typing `POKE 1, 55` and it's the final stage before a decruncher starts a game. Why am I so sure this trick will work? Well I can't say it will work 100% of the time but 90% of the time it's the very last line of code the decruncher does before starting the game – or it's not too far off from the code that does start the game.

The reason that a decruncher will set the value of location 1 back to $37 is that it doesn't know if the game wants to use the ROM or not. It just wants to be safe… if the game programmer wants to use the memory under ROM, he'll change the value of 1 to $34 on his own.

Every command in machine language has a two digit 'op code' associated with it. You can see these op-codes when you disassemble the code in CCS.

Do you understand what you're looking at when you disassemble a program? Are you comfortable with using the D and M commands in a MLM? If you're comfortable with using a MLM, proceed. Otherwise please read Section A – Additional Help on Op-codes

Putting it all together

- We know we need to intercept the game before it starts and after it's de-crunched
- We know that the location of 1 will probably be set back to $37
- We know that op-codes are the numbers found in memory that also represent actual machine language commands.

I'd like you to now load the game Blue Max which can be found here:
http://tapes.c64.no/tapes/BlueMax2001.zip

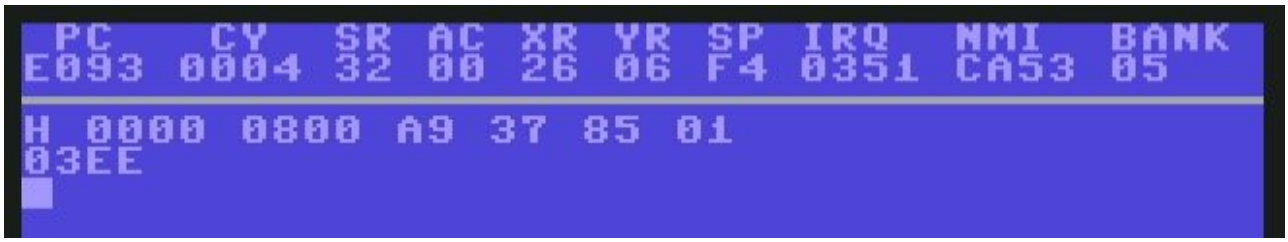
RUN the game Blue Max and WAIT until the scroller and loading music starts. This tape loader played a little tune to keep you entertained during the old days of long tape loads. Once you hear the loading music but the actual game hasn't begun yet, enter your MLM by pressing ALT-M in CCS64.

So we want to search for `LDA #$37` and `STA $01` and how we do this is through opcodes:

Type in:

```
H 0000 0800 A9 37 85 01
```

This says "hunt from memory location $0000 to $0800 for the following: `$A9 $37 $85 $01`"



IMPORTANT TIP: Remember, no matter what the game, no matter what the decruncher you should always start with this line:

```
H 0000 0800 A9 37 85 01
```

Now I chose $0000 and $0800 as a 'safe zone' to search. I don't know exactly how much of the C64's memory is being used for the decrunch routine. I just know that most decrunch routines are below BASIC (that is, less than $0800). You normally don't put decrunch code higher than $0800 because that's memory we want to keep free for our game as it loads and then decrunches. A decruncher doesn't know how much of the C64's memory your game will use, it might only need a little bit or it might need the full RAM.

Just to be safe a decruncher will usually do its magic BELOW BASIC (that is, in memory less than $0800).

By typing

```
H 0000 0800 A9 37 85 01
```

I'm searching from $0000 to $0800 which should be 99% of any and all decrunch routines out there (that's quite a claim but I think it's true).

TIP

As for what we're searching for, I can break down the op-codes:

A9 means `LDA` ("Load A with a value"), 37 is just the value to load into "A". Put together this means `LDA #$37` or "Load A with a value of $37"

85 means `STA` and 01 means, well location 1. Put together it means "Store A into location 1"

It's just like saying in BASIC

```
A = 37
POKE 1, A
```

(except that 37 is hex so it would actually be A=55). However aside from that, this is exactly what those two lines of machine code accomplish. We just want to find where the decruncher is setting 1 back to it's normal value because the next line of code will probably be the command to start the game.

And voilà!!! Location $3EE has what we're looking for.

If we type `D 3EE` to see what's going on here… it shows:

```
:03EE a9 37    lda #$37
:03F0 85 01    sta $01
:03F2 20 84 ff jsr $ff84
:03F5 4c 00 40 jmp $4000
```

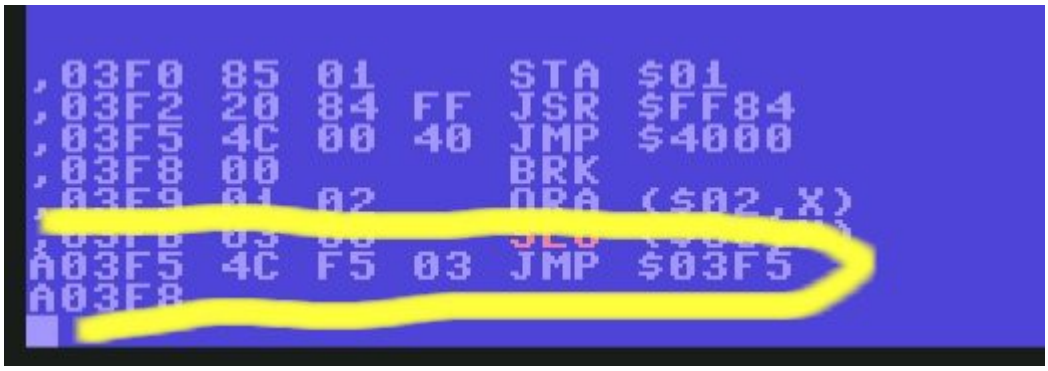
This says, load "a" with $37, store "a" into $01 (turning ROM back on) and `JSR` (jump to $4000).

I suspect that $4000 is going to be the start of our game. Why not $FF84 as seen at memory location $03F2? Because $F000 bank is ROM, and we see location 1 is set to $37 so we know ROM is turned on. There's no way this is game code.

Since location $03F5 is a `JMP $4000` let's change that to make an indefinite loop. We're going to use a new command, the ASSEMBLE command. The A command is used to assemble and it allows us to create 6502 code.

So on a new line type

```
A 03F5 JMP $03F5
```



We're saying, assemble code into location $03F5. And that code is telling the 64 to JUMP to location $03F5 (jump back to yourself). It's like typing `10 GOTO 10`.

Now exit the MLM and wait…. (if the game starts, you didn't do it right. You have to enter the loop AFTER the loader scroller and music begin to play)

Note: While testing this, I found that around the last two "blocks to load", the loader puts back the code to "JMP $4000". So even if you change it, it puts it back. I recommend waiting until the Blocks to Load reaches the 01 mark and then quickly break into the MLM and change $03F5 to `JMP $03F5`.

And once the counter reaches zero, we press ALT-M to check the PC value which should show $3F5. so yes, we're in the endless loop now because PC tells us what line number (err location) is currently being executed.

Now what?

We want to save memory to disk. To do this we use the "S" command in our MLM. First we want to avoid mistakes like those other crackers used to make – we want to turn off ROM. Type "BANK 0" in your CCS64 MLM, or using your cartridge monitor change the value of location 1 to a $34. Remember you can do this using the F command to fill, or you can scroll up to the 37 and change it to a 34.

The format for Save is:

```
S "filename" (device) (start address) (end address)
```

So for cartridge users it would be

```
S "bluemax" 08 0800 FFFF
```

In CCS64 you're going to type:

```
SP "bluemax" 0800 FFFF
```

instead of S, and you also don't need to supply a device.

I'm not sure how other monitors save, but you want to capture $0800 to $FFFF. You'll have to read up on how you cartridge loads and saves.

And just to test it… completely reset the CCS emulator by pressing Alt-Shift-R to do a hard reset (or exit and reload CCS). And we're going to load in our Blue Max 2001 game as raw data, no compression and no tape loader. In your MLM first, turn off ROM because we saved what was under the ROM and when we load it back in, it will try to load into ROM. `Type BANK 0` (or change location 1 to a $34)

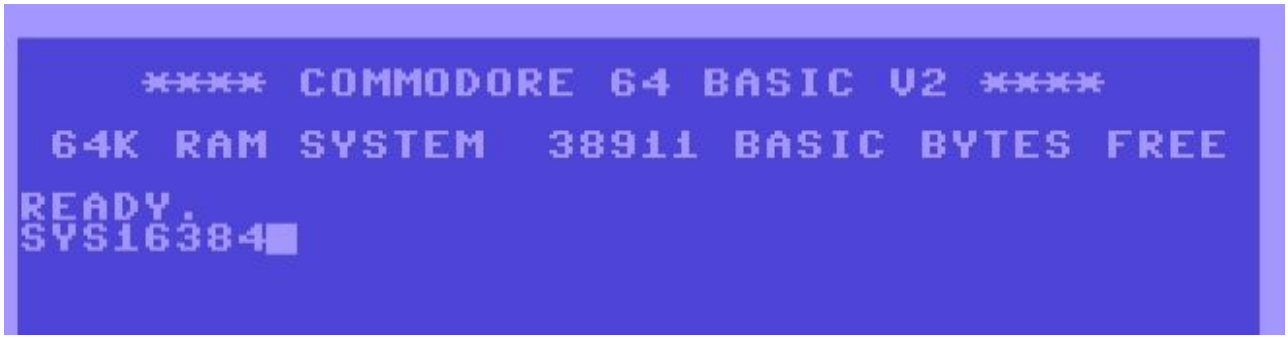
In CCS64 type:

```
LP "bluemax"
```

Or with your cartridge, something like

```
L "bluemax" 08
```

Now exit the monitor (be sure to set $01 back to $37 if not using CCS64)



And remember that JMP $4000? We know that 4*4096 = 16384 so type SYS16384
And tell me how you think you did If you see the game running, you did just fine!

2.4 Review

Games that are single file LOAD and RUN programs, will always load in at $0801 (I say $0800 but $0800 always holds a value of zero).

De-crunch routines will always be found in memory above $0800 *before* you run the game because, well… where else do programs load when you type LOAD? They load in at $0801 and proceed from there. These de-crunch routines will copy themselves to below $0800 once you type in RUN.

We know that we have to intercept the de-crunch after we've typed RUN and BEFORE the game starts,

We know that once we type RUN we can immediately so a search for these bytes:
A9 37 85 01 which translates to:

```
LDA #$37
STA $01
```

We know that if there's a JMP command immediately after the above two lines of code, it's going to be the start of the game (or a trainer).

2.5 Other Examples

You're not ready to form your own training group just yet cowboy.

Sometimes you may encounter a program that doesn't work with the steps I've shown to you. It could be the game is frozen with a cartridge, it could be that there are several 'layers' of crunched programs (e.g. Someone used ECA Linker first and then crunched that file).

((((((Pinball Power case study)))))))

Download 3d Pinball from Mastertronic: http://tapes.c64.no/tapes/3DPinball.zip

Now run the program until you see the flashing colours to indicate the game is loading.

Enter your MLM and do the search for where it sets location 1 to $37

```
H 0000 0800 a9 37 85 01
```

It finds it at $02ee

At $02ee we see:

```
:2ee a9 37     lda #$37
:2f0 85 01     sta $01
:2f2 85 c0     sta $c0
:2f4 a9 01     lda #$81
:02f6 8d 0d dc sta $dc0d
:02f9 ee fd 02 inc $02fd
```

So the loader is setting memory location 1 back to $37, good. It's also putting $37 into location $c0. It then puts $81 into location $dc0d and then does what we call an INC which is INCREMENT (increases the value of whatever is in $02FD by one).

I don't see a JMP to start the game…

But just for the heck of it, lets change the last line to JMP (jump) to itself.

In your MLM type:

```
A 02F9 jmp $02F9
```

This changes that INC command to now loop to itself. Exit the MLM and wait… once the colours stop flashing as part of the tape loader, we know it's in the endless loop.

Type ALT-M to enter the monitor (or use your cartridge)
Same process…

1) Turn off the ROM by typing `BANK 0` (or change location 1 to a $34 if using a cartridge)
2) Save the data using SP "pinball" 0800 ffff (Or with cartridge something like S "pinball" 08 0800 ffff)
3) Reset the computer
4) Load in the raw data. In CCS you go into the MLM and type `LP "pinball"`. In a cartridge you'll probably use `L "pinball" 08` (and if it crashes it's because you keep forgetting to turn ROM off by setting location 1 to $34 first)
5) Now we don't know a start address though because there wasn't one given… but type `LIST`. And we see it is indeed a program we can run. And there it is… our pinball game.

Now I can tell you that just by looking at the program, and seeing the screen go blank and noticing the large delay, that this is a crunched program. So we've managed to save the game that we ripped from tape – but it's still crunched. ARGH!!!

Worry not my friend… what did I teach you?

Run the program and BEFORE the game begins but while the screen is blank, go into your MLM, search for the magic bytes

```
H 0000 0800 a9 37 85 01
```

It finds those exact bytes at location $01CF:

Disassemble $01cf and you'll see :

```
$01e1 JSR 080D
$01e4 jsr $a68e
$01e7 jmp $a7ae
```

That first JSR is the one we want. How do I know this? Because it's above $0800 so we know it's game code, but also those two locations after it are in the $A000 range so we know it's not game code. That's BASIC ROM according to the memory map you have printed out and have taped to your wall like I told you to.

Now if you want to get technical – yes, $A68E could be game code as could $A7AE but because the code at $01CF changes the value of location 1 to $37 there is NO WAY that this can be game code. The value of location 1 says "I want the ROM guys to be in their houses, I'm not using them for RAM houses.". Those two lines of code at $01e4 and $01e7 by the way, perform a simple "RUN" command in machine code.

Soooo again we assemble some code:

```
A 01E1 jmp $01E1
```

To create an endless loop, we now exit the MLM and wait… after a few moments we enter the MLM. We check that the PC is at $01e1 telling us it's currently executing the endless loop.

Again we're going to type in "`BANK 0`" (to turn on all RAM) and again we dump the memory to disk with:

```
SP "pinball2" 0800 ffff
```

We reset the C64, we go into the MLM, put in the BANK 0 command again and then

```
LP "pinball2"
```

Type '`x`' to exit.

But how do we start the game?

Well every time we create an endless loop it's because we removed the original JUMP (`JMP`) that started the game. So to play the game we have to again call that jump which we do from basic with the `SYS` command. It used to be `JMP $080d` did it not?

And we know that 080D is 0*4096, 8*256 and D which is 13. that gives us 2061. So a `SYS2061` will let you know if you succeeded. I can tell you I'm looking at the game right now, playing… so it works under CCS64.

The file named "pinball2" is the full, raw, unpacked version of the game. It's ready for training if you so desire.

2.6 Tips and Questions

First let's clear up some things that may be confusing you.

We use "L" and "S" to load and save using a cartridge based machine code monitor. In CCS64 we use "LP". Under CCS64 emulator we always type in BANK0 before saving and before loading in our final de-crunched game. Or if we're using a cartridge, we always set location 1 to $34 before saving or loading the final de-crunched game.

Why?

Well if we save a game without typing Bank0 or setting location 1 to $34, we will save what's in the ROM memory at $A000-$FFF because we didn't tell the tenants of those ROM houses to leave for the afternoon while we borrow them for RAM purposes. And when it comes to loading in a game, any time you load something that is going into $A0000 or higher, if you haven't turned on RAM (through the means I keep mentioning to you) then it causes a miserable crash. Want proof? Just start up your C64 and go into the MLM and fill $D000-$Dfff with this command:

```
F D000 DFFF 22
```

Without setting location 1 to $34, you write to the ROM and it causes a crash.

So always remember to set your ROM banks to RAM when loading or saving. Maybe you can recall the old days when you tried to load in a 220+ block game that was clearly too large. Eventually the screen turned to garbage and it crashed – this was because the game loaded into $D000 and beyond. It was too large for the C64.

Another question you might have…. Why do I have to run the program first and then look for the magic bytes you keep telling me to look for? I tried searching for them before I ran the game and it finds them

```
H 0800 A000 A9 37 85 01
```

Yes, because the game has loaded into memory beginning at location $0801 but remember that a de-cruncher will copy itself to lower memory under $0800. So even if you did change a JMP to loop to itself

example:

`0840 JMP $1000` is changed to `0840 JMP $0840`

It would CRASH because when it gets copied to under $0800 it's going to look something like 0133 `JMP $0840` instead of being "`0133 JMP $0133`" as it should be

2.7 Tetris “Hello Hacker” loader

If you feel confident in your skills, download Tetris: http://tapes.c64.no/tapes/Tetris.zip

Now by looking at the game, it has a loading picture so we’re not even going to bother looking for anything until we run the game and see the loading picture appear. We’d only be dumping the picture to disk if we proceeded at this point.

When the picture appears we’re going to enter the MLM and hunt for the bytes that set location 1 to $37

```
H 0000 0800 a9 37 85 01
```

Reveals nothing….

So let’s try a backup plan of searching for just the `STA $01` (store “A” into location 1) portion of the code – which is the two op-codes : 85 01

TIP: A9 is the op-code for LOAD A with, and 85 is the op-code for STORE whatever is in A into a memory location. The numbers immediately after each of those is what we’re loading and storing with. So when you see a9 37, those go together in pairs. It says “LOAD A with $37” and the 85 01 says “Store A into location 1”.

So this isn’t good… we have three places where it found a STORE A into location 1.

$018D, $019C and $02AA

Let’s look at $018D with the disassembly of the code:

Type `D 018D`

Scroll down until you come to the end of the code and you see a JMP $02EB at location $01CD. $2EB isn’t likely to be the game as it’s below BASIC. Maybe it’s part of the loader.

```
01CD JMP $02EB
```

Well let’s disassemble that code at $02EB and see what it is…

Type D 02EB to see what we’re jumping to…

Okay at location $02EB we see a series of `NOP`’s. `NOP` is like a `REM` statement in BASIC, it means No Operation – do nothing. But as we follow those `NOP`’s we see that there’s no code after it, it’s all just `BRK`’s. It’s not so much the `BRK` I’m paying attention to as it is that the memory is all zeros. In other words, by seeing a series of zeros this code doesn’t make sense. What’s going to happen is that the game is going to JUMP to $02EB, it’s going to rip through those `NOP`’s in a hurry because they mean “do nothing” and it’s going to crash into those 00’s (`BRK`’s). It’s like a train heading down the track with a brick wall at the end…. Clearly those `NOP`’s are there for a reason and I know what it is. At some point the loader is going to try to trick us and put some code where those `NOP`’s are. It’s meant to trick us… but we know better because those `NOP`’s lead to a crash.

In fact if you scroll down beyond those zero bytes to $033C it looks like we have some text here… switch to the graphic view and see what it says

Type

```
I 033C 03A0
```

“Wild Save © Interceptor Micros 1987 Written by Andrew Challis January 1987 Hello Hacker I Hope you have fun with this loader”

And back in the days before cartridges, I’m sure this game would have been a challenge. But using our CCS64 emulator we can fix this.

So we don’t know what’s going to happen with the code at $02EB we just know that it will probably change at the last minute in an attempt to trick us.

So let’s change the line that jumps to $02EB.

Type in: `A 01CD JMP $01CD`

Which will tell the loader to loop instead of going to that code that looks suspicious.

Wait until the loader stops…. And disassemble 02EB

AHA! It did change the code, those sneaky programmers…

The code is simple

```
LDA #$37
STA $01
STA $C0
LDA #$00
STA $D020
LDA $DC0D
JMP $02A7
```

So it does set 1 back to $37 (turn ROM on) and changes the border to black (load A with zero, store A into $D020 the border colour). It then Jumps to $02A7

So we disassemble $02A7. I’ll spare you the code, but it basically turns off interrupts and then we see:

```
JSR $A659
JSR $A533
JMP $A7AE
```

This brings us to the next “tip” on how to dump games. Any time you see `JSR $A659` and `JMP $A7AE` this is like saying “RUN” in BASIC. In other words, this is going to be a BASIC program.

Just to confirm type

```
I 0800
```

And we see "`2076 AUSTRO-COMP E1`" at $0806 so this is going to be a program that has a line number

```
10 SYS 2076 AUSTRO-COMP E1
```

The tip I have for you is, when looking at loader code, also look at $0800 visually with the I command (I 0800) and see what's there. MANY times it's going to look like "(2064)" or "2061" or some other series of numbers. This could be an indication that this is a program that you just need to RUN. No need to worry about what the `JMP` is to start the game.

So we're not going to bother with trying to figure out what the start address is, we're going to treat this game like any other BASIC program and just save the memory to disk.

Switch out your ROM and save $0800-$ffff

Load it back in and type `LIST` – there's a program there. Type `RUN` and it works! There's a few glitches on the screen but for a beginner you did pretty well against this loader that was supposed to defeat a 'hacker'

2.8 Vixen Loader

This time we're going to try dumping the game Vixen to disk.

http://tapes.c64.no/tapes/Vixen.zip

We know there's a loading picture so we're not going to bother looking for any code until after the loading photo appears.

We do our search

```
H 0000 0800 a9 37 85 01
```

No results

Let's try our backup search:

```
H 0000 0800 85 01
```

Three results: `$02B8 $03E7 $0431 $04A3`

A little overwhelming I'm sure… but if we look at the line $042f we see that

```
042f LDA #$35
0431 STA $01
0433 JMP $0811
```

That `JMP $0811` looks pretty good to me… let's loop it.

```
A 0433 JMP $0433
```

We wait for the loader to stop, dump $0800 to $FFFF, restart the 64, turn off ROM, load the data back in and `SYS 2065` (which is $0811, the address that was at location $0433 in the loader)

You don't need my help now. You're a pro at dumping data. Incidentally the reason it put a #$35 into location 1 instead of a #$37 is most likely because the game uses memory under ROM ($A000).

2.9 Bosconian

Now this is a rather cool idea. In the tape era of the C64, games often took a long time to load. To make the long process easier, companies introduced music that played while the game loaded. This loader allowed you to play Space Invaders while the game loaded.

Download Bosconian from this link
http://tapes.c64.no/tapes/Bosconian87.zip

Now run the game… when the border flashes, enter your MLM and search for these bytes:

```
H 0000 0800 a9 37 85 01
```

It doesn't work… so there's no code that says:

```
LDA #$37
STA $01
```

Let's do the backup search…

```
H 0000 0800 85 01
```

and there's three results: $0350 $036F and $03E8. If we scroll down nothing really jumps out as the game JMP (jump to start the game) however at $0428 we see the `JMP $A7AE` which as I mentioned in the game Tetris, is indicative of the game doing a "RUN" command.

Soooo…. Type

```
A $0428 jmp $0428
```

and exit the MLM to allow the loader to finish.

When the music stops, the flashing border stops and we enter the MLM, we see that the PC shows $0428 which is good (it's stuck in our loop). Dump the game to disk, I'm not even going to tell you how at this point. You should know how.

Now turn off the ROM, and load the game back in. Knowing that it's likely doing a RUN instead of a `JMP`, type `LIST`. And there you are…

```
1987 SYS2068 FOR ALL
```

it is indeed a `RUN`'able program.

But I suspect this is also going to be a crunched game…. So run it and then immediately enter the MLM and check for the usual bytes (`H 0000 0800 a9 37 85 01`)

`A9 37 85 01` aren't found but at $0368 we find the bytes 85 01 because your backup plan was to type : `H 0000 0800 85 01`

Disassemble $0368 and look at $0370, that has to be the game JMP.

```
0368 STA $01
036A LDA #$1B
036C STA $D011
036F CLI
0370 JMP $0810
```

Type in: `A 0370 JMP $0370`

This will prevent the execution of code at $0810 and force an endless loop when it reaches $0370 of the de-crunch routine.

Now you can dump this to disk. Now you might think that the game is good to go, but it isn't because $0810 isn't the start of the game, This game has been linked using ECA Linker. How do I know? Just from looking at the code – and from experience. You might not always be able to know when you've peeled the last layer of the de-crunching onion away until you get experience in looking through 6502 code.

But regardless, type RUN and immediately enter your MLM.

Search for `a9 37 85 01` which are found at $0199

At $019E we see a `JSR $7274`

So in the MLM type: `A 019E JMP $019E`

Then dump the memory to disk again (Section A).

Now load the data back in… and since we know $7274 is 29300 in decimal we're going to type

```
SYS 29300
```

And there you are….

Now you might wonder, how did I know that the game still had one final hurdle to go with one more crunch? Well when you see

```
1998 SYS (2064)
```

It's more than likely ECA Linker that was used because of the line number being 1998. That's your typical ECA Linker result.

2.10 Intros

Now there may come a time you want to dump a game to disk that has an intro before it. The intro might be from a crack group, but they didn't train the game. You just want to get the raw game data so you can train the game.

Intros work like this: They load into memory at around $0800, in the beginning of the C64's RAM. The game itself is loaded just AFTER the intro, piggybacked on top of it. For instance an intro might be placed at $0800 to $2000 and the game will be loaded into memory at $2000. When you press Space, the intro executes a small piece of code that copies the game from $2000 down to $0800 and runs it. Since this transfer has to occur in a way that the transfer routine isn't overwritten, it usually occurs on the screen RAM which is why you often see characters incrementing (changing) on the screen after you exit an intro. It might also occur around the $033C area of memory which is the cassette buffer.

Think of it this way: You have a pipe that runs vertically and it's 5 feet tall. The bottom one foot of the pipe is empty and capped at the end. There's a turn valve just above the one foot mark. The top four feet of the pipe is filled with water. In this analogy the empty bottom foot of the pipe is the C64 memory at $0800 with the intro. The top four feet of pipe with water is the game.

Now if you turn the valve, the water pours down into the bottom of the pipe. So too does the data pour down when you press space. But if you have the transfer routine anywhere in that 'pipe', it's going to also be transferred down with the flow and it will crash. Copying the transfer routine to a safe area below BASIC makes it safe. What happens to the intro? All that data is overwritten which is why we can't place a transfer routine in the intro.

Now trying to find the transfer routine can be tricky because there are hundreds of intros out there that all use different methods.

One way is to search for the routine that checks for when you press the spacebar and try to figure out where the transfer routine is. If you know machine code you'll want to look for: `JSR $FFE4` or `LDA $DC01` which are two different methods used to check for a space bar press.

I'm going to recommend you look for the code where the transfer actually occurs.

Download Defender from Gamebase64.com

http://gamebase64.hardabasht.com/games/d1/DEFENDR1_02079_02.zip

Run the game and watch the intro from Avatar. Now enter the MLM. We want to find the code where the intro transfer routine copies the data from where the game is (in memory above the intro memory) down to $0801. So type

```
H 0800 5000 01 08
```

The $5000 is just a guess as to where the intro would end in memory. The 01 and 08 is representing $0801 (we write the bytes high byte first, low byte second) and I'm searching for $0801 because that's most likely where the program will be copied to. Some intros copy down to $0800 but this is unnecessary because $0800 is always set to zero. If it's other than zero, your program will not run.

We find two results at $0C06 and 2D7F.

If we type D 2D60 and scroll down, we can see there's no machine code there. This is a false positive. It could be data for something else.

If we type `D 0C00` and scroll down we see:

```
0C00 LDX #$00
0C02 LDA $5700,x
0C05 STA $0801,x
0C08 inx
0C09 BNE $0c02
0C0B inc $07C4
0C0E inc $07c7
```

(I chose to disassemble $0C00 instead of $0C06 because it's helpful to see what code is just before the search result you're looking for. As evident in this example.)

What this code does is it sets X to zero (like in BASIC `x=0`)

It then LOADS $5700,x and stores what it finds at $5700 into $0801,x

then we increment X by one

if branch not equal (that is, if X hasn't rolled back to zero) go to $0c02

This is like a BASIC loop:

```
x = 0
for I = 0 to 255
a = peek($5700) + x
poke ($0801) + x, a
next
```

Do you follow?

The game itself is found at $5700 and copies the first byte found at $5700 into $0801

X goes from a 0 to a 1, and then it loads $5701 and stores that location into $0802, then it copies $5702 into $0803 and so on.

the ",X" is like saying "+x"

Finally when x rolls around to zero (it will go up to 255 and reset to zero) then the $57 in $5700 will become a $58 and the $08 in `STA $0801,x` will become a $09, and so on.

If you don't follow, just scroll down to $0C1B and that's your jump to the game.

So we know what we're looking for, but we also need to wait for the routine to copy itself to under BASIC (the safe zone). Exit the MLM and press Space. When you see the credits for the intro, then you'll want to enter the MLM again. Note the changing characters on the bottom of the screen – that's the two increments we saw at $0C0b and $0C0E incrementing the locations being copied in groups of 255 bytes at a time.

So to fix this, press SPACE and when the intro exits and you see characters changing on the bottom of the screen QUICKLY enter your MLM. The transfer routine is found on the screen (screen RAM) so your results won't be below $0800 this time.

Type:

```
H 0400 0800 a9 37 85 01
```

Which will result in the code having been moved to $07D6 and you'll want to loop the

```
07DB JMP $07DB
```

3.0 Training Concepts

There are different types of trainers including infinite lives, sprite collision, level skipping, infinite energy, etc.

- Lives – involves preventing the decrementing of a number where the number is your number of lives.
- Sprite collision – a register in the C64's memory that indicates whether you've collided with another sprite or character on the screen. Stop that register from revealing a collision and you're invincible.
- Level skipping – hacking the game code to set a value, or jump to a location, that causes the game to skip a level.
- Energy – The same concept as lives however energy may be a bar that gradually shrinks on the screen rather than an actual number (e.g. XXXXXXXXXX where the X's slowly get removed instead of "Energy: 50")

The easiest way of training is to use an Action Replay Cartridge that has a trainer built into it. It works by asking you the number of lives that you currently have (e.g. 3). You then exit the cartridge, lose a life and enter the cartridge again. The cartridge will then check all memory locations that used to contain a 3 but now contain a 2. If you have such a cartridge, you should have little difficulty finding the code that removes a life.

In a game, you will always start with a certain number of lives, on a certain level and with a certain amount of energy. This is done by loading the number necessary and storing it into a memory location. You could theoretically write a game and save it with a memory location already set to the number 3 (for 3 lives) instead of loading a 3 and storing it into a location, but once you died and tried to play the game a second time, the memory location would still hold a 0 (zero). For this reason the game needs to initialize the memory locations before the game begins – each time and every time.

For example if the makers of the arcade game Galaga didn't initialize the memory location that held your lives, with a "3" each new game, you'd have to turn off the machine and turn it back on so it would load up with that "3" in the right memory location. So memory locations will always need to be initialized instead of just being loaded from memory (or disk) with the number of ships already in the designated memory location.

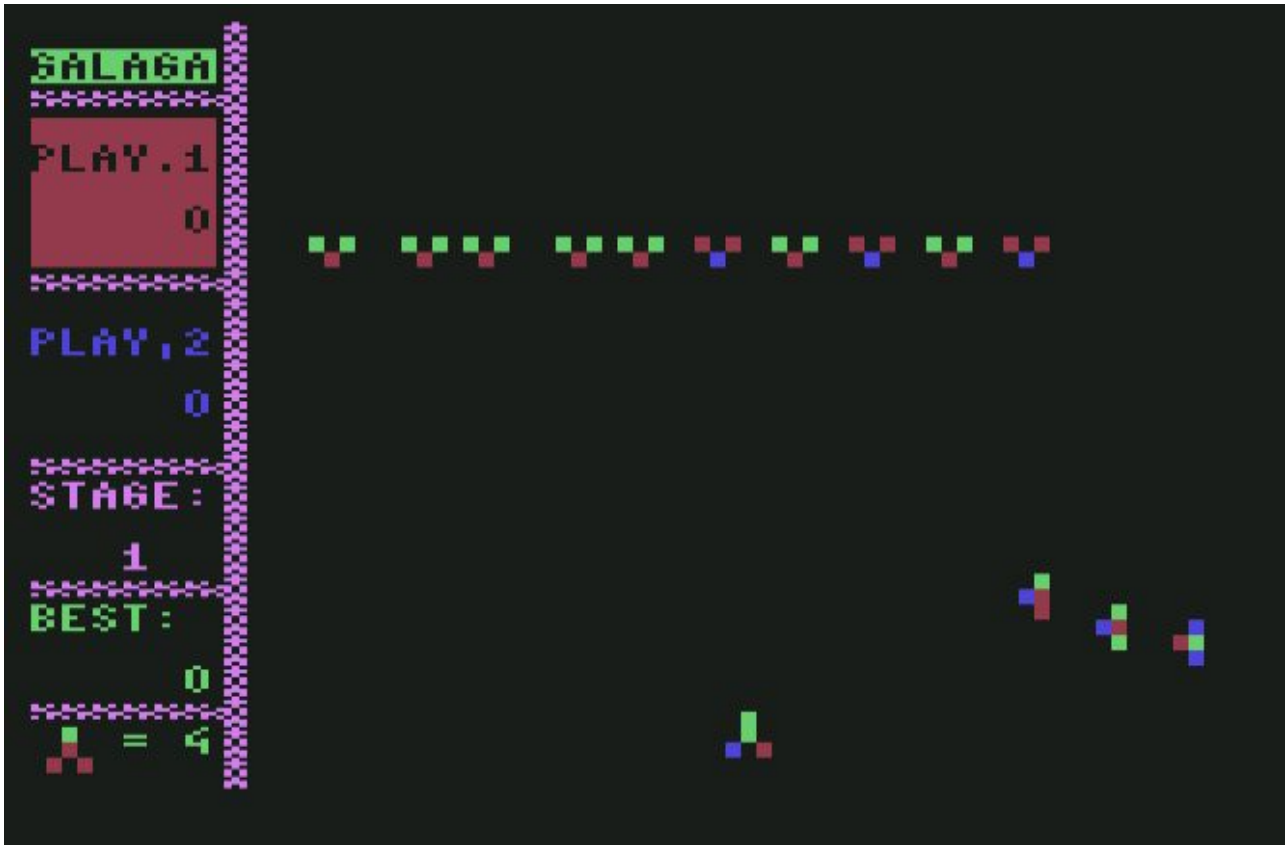
The concept behind this is that you know how many lives you begin with and you look for code in the game where it loads that number and stores it into a location. You then check those locations after losing a life to see which of them has decremented in number. The con of this approach is that you could have dozens of memory locations that are loaded with your number of lives, meaning you have to check them all.

3.1 Method 1 – The Screen RAM Method

In this method we choose not to look for the code where your lives are set and decremented. Instead we look at screen memory to see what location on the screen displays your number of ships.

PRO: Easy to find usually and if the game reads the number of lives from the screen memory, it's easy to change.
CON: The location might just reflect the value of a secondary location that is your number of ships. Example: Location 2 might hold your number of ships, but the game copies what's in location 2 and puts it on the screen. The actual screen location has no bearing on the actual number of ships. Location 2 does.

Download Galaga from Gamebase64.com at the following link:
http://gb64.com/game.php?id=3021&d=18&h=0



Type `I 0800` and scroll down and down until you find what looks like screen memory. I'll give you a hint, it's at $8000 instead of $0400.

If you look at around $8360 you can see that your number of lives shows as a '4' in location $839D. You can change that 4 to a 9 if you want, and you have 9 lives. It works! How about a more permanent method though? We know that location $839D is showing our number of ships.

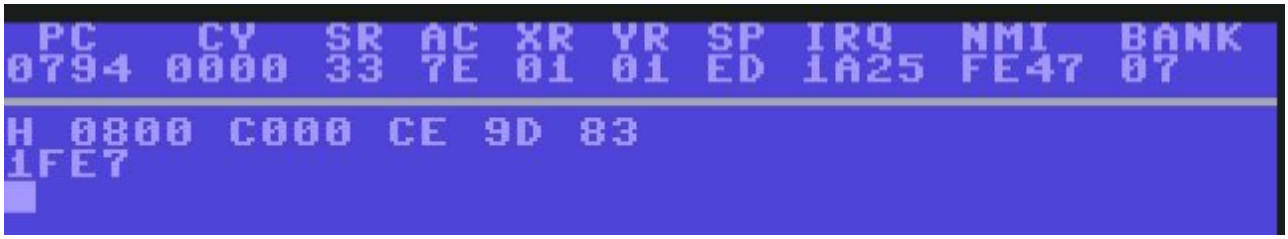
We'll search the game for the command:

```
DEC $829D
```

This means decrement whatever is in location $839D. The command is much the same as we've been doing:

```
H 0800 C000 CE 9D 83
```

CE means Decrement, 9d and 83 is $839D where the high byte goes first (the first two digits of the memory location are typed last, the last two digits of the location are typed first)



And with our command, we find code where this is decremented at $1FE7.

If you look at line $1FE7 it says `DEC $839D`. So if we change this to do something other than reduce the value of our ships from the screen memory, we'll have cheated the game. How about that NOP command? We know NOP does nothing at all (it means "No Operation")

On a new line type:

`A 1FE7 NOP` and press return

Then enter two more NOP's, the MLM should automatically advance to the next memory location so you don't have to type the "A"ssessemble command each time. When done, press enter.

Exit the MLM and continue to play the game…. And it worked! You've just trained your first game.

3.2 Galaxia – Screen RAM Method

Let's now download Galaxia from Gamebase64.com at the following link:
http://gb64.com/game.php?id=24566&d=18&h=0

This time we see we don't have a number representing our ships, but we see actual ships at the top right of the screen. Again we're going to enter the MLM and search for the screen RAM. I'll give you a hint: It's at normal screen RAM - $0400.

Look at $041F by typing `I 041F`

In bytes we'd see: `30 31 20 4F 20 4F`

In text mode we'll see: `01 O O`

The 30, 31 is equal to the numbers "0" and "1" and the two "O"'s are the ships shown on the screen.

Now by counting the number of memory locations over we see that $0422 is where our first ship is shown and $0424 is where the second ship is shown. Now it's not going to decrement a memory location because this isn't a number value. It's two characters on the screen. What I mean is, we're not looking for a "3" lives than becomes a "2" then a "1" and then Game Over. We're looking for a routine that places ships on the screen – tiny ships that indicate your number of lives.

I suggest searching for $0422 and $0424 to see if we find any hits.

```
H 0800 C000 22 04
```

0A3D and $0A6C are found!

Disassemble at $0A38 to see what it's doing:

```
0A38 LDX #$00
0A3A LDA #$20
0A3C STA $0422,X
0A3F INX
0A40 CPX #$06
0A42 BNE $0A3C
```

This isn't what we want – but it's close. It loads X with zero, loads A with a value of $20 which is 32 in decimal (32 is also ASCII for space). It stores the space in $0422,X and increments X until it becomes 6. So this basically erases the ship count from the top of the screen.

Now the second result is of interest to us. Great interest!

We'll disassemble a little bit before our result…

```
D 0A63

0A63 LDX #$00
0A65 CPX $50
0A67 BEQ $0A70
0A69 LDA #$4F
0A6B STA $0422,X
0A6E INX
0A6F INX
0A70 CPX $50
0A72 BNE $0A65
```

What this does is as follows. It loads X with zero (in BASIC it would be x = 0)

It then compares X to $50. Now there's something to take note of and that's that there is no number sign before the $50. When you are loading or comparing numbers there's two ways to do it. You can load the actual number or you can load a memory location.

For example:

```
LDA #$00
STA $0400
```

Is not the same as

```
LDA $00
STA $0400
```

The first example will load A with the actual number 0 and store it at $0400. The second example will load whatever is in the actual memory location $00. When we use the #$ prefix we mean "I want you to use this actual number". So when we do a CPX #$50 we're saying, "compare X to see if it's $50 in hex" whereas if we did a CPX $50 we'd be saying, "compare X to whatever is in LOCATION 50"

Getting back to our example.

- LOAD X with zero
- compare X to whatever is in location $50
- if X = whatever is in location $50 (branch equal) to $0A70
- otherwise load A with #$4F and store A at $0422,X
- increase X once
- increase X again
- compare X to location $50, if not equal go back to $0A65

What this is doing is it is loading A with #$4F and placing it on the screen, it is then incrementing X by 2 (inx and inx) which effectively places another ship on the screen double spaced from the last ship. We also know by viewing the screen RAM that #$4F is what's being used to show a picture of a ship on the top of the screen.

So say you have two ships remaining – it will load X with zero, it will compare X with location $50 which holds a value of 4. Since X is not equal (0 Is not equal to 4) then it goes to line $0A69 to place ONE ship on the screen. It then executes the two INX commands at $0A6E which increases X to a value of 2 (because X was set to zero on line $0A63). Essentially for every ship that you have, location $50 will hold a value of 2. So if you had 3 ships, location $50 would be equal to 6. I know this only because I checked the value of location $50 when the game first began and it was a value of 4 yet I only had 2 lives.

This simple loop routine draws a ship for each life you have. I deduced that location $50 was responsible for holding the number of lives because it is the only comparing being done in this loop to indicate when we should stop displaying ships.

To cheat the game we're going to look for where it DECREMENTS location $50. Now since we know $50 holds DOUBLE your actual lives (e.g. One ship means location $50 is equal to a 2) we're probably going to be looking for TWO decrements.

The command to search for is

```
H 0800 C000 C6 50
```

This is because C6 means 'decrement' and of course the 50 is the memory location we'd want to see being decremented.

TIP

To confirm that these are the bytes though, try this test. Assemble the command "decrement 50" or "`DEC $50`" into location $0100 by typing:

`A 0100 DEC $50` (and press return)

You'll notice that it now says "C6 50" which are the "op-codes" or operational codes that we want to find.

And we do find results at locations $1b97 and $1b99. Coincidentally 1b97 and 1b99 and right after one another, so I was right about the 'double decrement'.

Type `D 1B97` and you'll see:

```
1B97 DEC $50
1B99 DEC $50
1B9B LDA $50
1B9D CMP #$FE
```

So this routine is the one that decrements our lives. Let's change those two decrements to our "NOP" (no operation, do nothing at all). Type:

`A 1B97 NOP` (press return)

And continue to type in three more NOP's. It's important not to put in too many or too few NOP's. Exit and test the game… and it works! Infinite lives!

TIP

Why did the game decrement location $50 and then compare location $50 to a value of #$FE as seen in line $1B9D?

Ships: 3 – value of $50 = 4
Ships: 2 – value of $50 = 2
Ships: 1 – value of $50 = 0

When on your final ship it would decrement (subtract) 2 from location $50 which would be equal to a zero causing it to wrap around. 0 minus one=$FF. $FF minus one = $FE. Memory locations wrap when you add or subtract beyond 0.

3.3 Method 2 – The Load and Store Method

There are times where the screen memory method won't work.

Download Kong Strikes Back from this link:
http://tapes.c64.no/tapes/KongStrikesBack.zip

Now this game can actually be cheated using the screen RAM method if you look at $4400 and onward, but I'd like to see you advance your skills to do it using a more advanced approach.

Play the game… immediately you'll notice that you begin the game with 4 Bombs and 4 Lives.

So enter the MLM and let's try to find where those memory locations are initialized through the `LDA #$04` and `STA $xxxx` commands (where $xxxx could be anything).

Type:

```
H 0000 A000 A9 04
```

We find 12 results

The first one is at $223D

So let's see what it does with that value of 4.

```
D 223D

223D LDA #$04
223F STA $23DC
```

Now let's look at location 23DC using the M command:

```
M 23DC
```

In my game, location $23DC is showing a zero value and I've still got 4 lives. So this isn't the right location. The second result at $2267 also stores the number 4 into location $23DC so this is wrong too.

The third result at $260E stores the number 4 into $D839 which is screen colour RAM. This is also wrong.

The fourth result at $2B69 looks like this:

```
2B69 LDA #$04
2B6B STA $2517
```

And currently location 2517 holds a 4… so I'll lose a life. No, that didn't work… but I noticed that when I used a bomb in the game that the value in location $2517 went down to a 03.

So let's find where it decrements location $2517.

TIP

Can't remember the op-codes you're looking for? Just assesmble the actual code you're looking for, into a location such as $0100.

Type:

```
A 0100 DEC $2517
```

And press return. You'll now see your op-codes are: C6 17 and 25.

```
H 0800 A000 CE 17 25
```

location $2FDE is the result

```
2FDE DEC $2517
2FE1 LDA $2517
2FE4 CLC
2FE5 ADC #$30
2FE7 STA $47D1
```

If you decided to use the screen RAM method I taught you, and you tried searching for location $47D1 which also held the number of bombs in screen RAM, you should also have found this routine. The routine decrements a life, it then loads whatever is in $2517 and then it adds #$30 to it and stores it in $47D1. What this does is it takes the number of lives (1 to 4) and adds an ASCII value of "0" (#$30) to it so that it can be displayed on the screen as a number. Values of 1-4 are not ASCII numbers so the addition is necessary. (e.g. #$31 = ASCII "1", #$32 = ASCII "2")

Let's make infinite bombs… type:

`A 2FDE NOP` (press return and type in two more NOP commands)
`2FDF NOP`
`2FE0 NOP`

and we've just created infinite bombs. Now let's find the lives….

For the lives I'm going to use the screen RAM method which would be $47DB. I'm choosing this because I don't readily see where it is putting the lives. So let's search for anything decreases the location $47DB

`H 0800 A000 CE DB 47` (remember high byte first)

results: $2DB3 and $3AEF

Overwrite the $2DB3 DEC $47DB by typing:

`A 2DB3 NOP` (and two more NOP's)

We put three NOP's because the command DEC $47DB uses three op-codes (CE DB and 47). And there we are… infinite lives! I might suggest doing the same for the other decrement found at $3AEF just to be safe.

3.4 Infinite Time

Now we also see that there's a countdown timer for Kong Strikes Back. Let's see if we can find that too. Type "`I 4400`" and scroll down until you find the timer numbers in the screen RAM (which ordinarily would start at $0400 but not in this case)

And we see the timer numbers are found at location $47E3 through to $47E6.

Let's try to find where it's decrementing those numbers…. Going back to the screen RAM method.

I am typing:

```
H 0000 A000 e3 47
```

nothing

```
H 0000 A000 e6 47
```

nothing

I tried searching for anything to do with the first and also the last numbers of the timer ($47e3 and $47e6) because it subtracts from the last number, right?

Now here's a tip…. Maybe the game allows for even MORE time than we can see on the screen. Let's just try searching for $47E2 which is one character to the LEFT of the four digit time on the screen memory.

```
H 0000 A000 E2 47
```

Results: $29F8 and $2A06

Now we're getting into some code that I'm just not able to teach you – you need to have some understanding of machine language to be a successful game trainer. But if you look at line $29EE it loads A with $29EA. At line $29F4 and $29F5 it adds a "0" to the number and stores the number at line $29F7 into $47E2, X. So this is definitely our timer. What it's doing is converting whatever is in location $29EA into a four digit number.

I suspect $29EA will be our timer. We'll search for a decrement of that location:

```
H 0000 A000 CE EA 29
```

Nope, there is no DEC $29EA to be found.

Since we can't find a DEC-rement command let's see if we can find anything to do with location $29EA.

```
H 0000 A000 EA 29
```

Results:

$2087, $29EF, $2A1D, $2A23, $2A39 and $2BB0.

```
$2086 LDA $29EA
$2089 STA $266B
```

This does nothing more than copies the location $29ea and puts it into $266b. Move to the next result….

Beginning around $29EE we can see this just puts the numbers onto the screen – we already know this. Next result…

```
2A1C LDA $29EA
2A1F SEC
2A20 SBC #$01
2A22 STA $29EA
```

Can you figure out what this is doing? It loads $29EA into A, it then subtracts using the SBC command where the #$01 means "subtract 1" and then it stores it back into the same location. This is exactly what we want. It didn't use the DEC command at all, and this is another thing you'll need to know – how to search for the SBC command.

How can we make it infinite time? There are a few methods…

Method A: Change the 1 to a 0 so it reads "SBC #$00" and subtracts nothing at all.
Method B: NOP out the subtraction code
Method C: NOP out the final STA $29EA so that it subtracts but doesn't store it back.

To do this:

Method A:

```
A 2A20 SBC #$00
```

Method B:

```
A 2A20 NOP

NOP
```

Method C:

```
A 2A22 NOP
NOP
NOP
```

Unfortunately methods A and B won't work because at line $227F the game it replaces the "1" back in at the "`SBC #$01`" command. If we were to put it to "`SBC #$00`" then it would revert back. If we changed it to `NOP`'s, one `NOP` would be changed to the `ORA` command when that "1" was put back.

To solve this problem let's just go to line $2A22 and change the `STA $29EA` to `LDA $29EA`. So after it loads $29EA and subtracts 1 through the `SBC` command, instead of putting the reduced value back into $29EA it will just re-load $29EA.

Type:

```
A 2A22 LDA $29EA
```

now this doesn't quite work yet because there is a second location that holds our time… go to line $2A31 and change the `STA $29EB` to a `LDA $29EB`. Now it works 100%.

In basic this would be like saying:

```
10 X = 3 : REM Number of lives
20 X = X - 1 : REM reduce life
30 X = 3
```

3.5 Dukes of Hazzard – Sprite Collision

In this case study we're going to introduce you to sprite collision. When two or more sprites touch on the screen, a register in the C64 changes. The register is $D01E.

There are eight allowable sprites on the screen at any one time. Location $D01E will read the sprites that have collided – It does this through the use of bits. Since we have eight sprites and only one register to read, it's somehow necessary to break down the result into a way of determining which of the sprites have collided. We do this through assigning the sprites bit values.

The values are 1,2,4,8,16,32,64 and 128. Sprite 1 would hold a value of 1, sprite 2 would hold a value of 2, sprite 3 would hold a value of 4 and so on. So if we have the very FIRST sprite and the very LAST sprite touching, the value of $D01E would be 129. It would be 129 because it adds up the bits of the sprites that are touching (1 and 128). If all 8 sprites were touching one another, we'd have a value of 255 in $D01E.

So if we want to create" invincibility" in a game, we can often do so by changing the line of code that loads the register $D01E and changing it to load a zero instead. Zero to indicate no sprites are touching.

Download and run Dukes of Hazzard from this link.
http://gb64.com/game.php?id=2385&d=18&h=0

Play the game and notice that you lose a life if you collide with other cars, motorcycles, etc.

If the game uses a straightforward `LDA $D01E`, this can be one of the easiest cheats to make. To accomplish this we look for the bytes: `AD 1E D0` which means `LDA $D01E`

We find two results. The first one is at $203E

```
203E LDA $D01E
2041 LDA $D01F
```

Why did the game load the register but not do anything with it? It loads $D01F right afterward. This would be like typing in BASIC:

```
A = peek(53278)
A = peek(53279)
```

The reason it, and many other games, does this is because the register $D01E holds the value of a collision until you read it in. So it's necessary at the start of a game to read the registers to "clear them out" and reset them to zero. This isn't what we're looking for but I can almost guarantee you the second result is.

The second result is found at $5C3E

```
5C3E LDA $D01E
5C41 STA $3B03
```

So location $3B03 will hold the result of any sprite collisions. To gain invincibility we want to change line $5C3E. Now this line holds three op-codes because it uses up three bytes: AD for the LOAD (LDA$) command, and $1E and $D0 for the $D01E. So we'll need to replace it with THREE op-codes. Type:

```
A 5C3E LDA #$00
5C40 NOP
```

We need that `NOP` in there because `LDA #$00` uses only two op-codes. So if we didn't include the `NOP`, the game will crash.

You see the original bytes were:

`53ce – AD 1E D0` (or `LDA $D01E`)

we changed them to

`5c3e – a9 00 ea` (or `LDA #$00`, `NOP`)

had we not put the NOP in, the three bytes would have been:

`53ce – a9 00 d0`

And after the game processed the `LDA #$00`, that $d0 command stands for `BNE` (branch not equal) and we'd be sending the program to execute some area of memory that it's not supposed to. Remember op-codes are commands and the 64 will execute whatever comes after the `LDA #$00` command – so if you change a three byte command to a two byte command, that third byte is going to be treated as a command when it's not supposed to be one. In this case the $d0 is a left-over from the $d01e in the old `LDA $D01E` command.

Anyway I'll stop blabbering… the game now allows you to be invincible to other vehicles.

Run the game and play it. You see that we start with 3 cars. So let's try to find the lives. I first start by typing:

```
H 0800 A000 A9 03
```

this is the `LOAD #$03` command in 6502. A9 is LOAD A and 3 of course is the value to load.

TIP

> If you don't know the specific op-codes to search for, again, try assembling the code into memory at around $0100 to confirm.
>
> ```
> A 0100 LDA #$03
> ```
>
> To which when you press return it will place the op-codes in automatically (A9 and 03).

Now I've checked the results (there are 6 of them) and none appear to store a value of 5 into any particular location. Some of the results are for screen colour.

3.6 Dukes of Hazzard – Character Collision

Just as $D01E is to sprites colliding with one another, $D01F is a register to indicate when a sprite has touched a character on the screen. Sprites and characters can overlap and often to.

After you've created invincibility for yourself you'll notice that the helicopter dropping ammunition on your car will still cause you to lose a life. This is because what is being dropped from the helicopter isn't a sprite, it's a character.

Much like steps taken for finding sprite collision, we search for:

```
H 0000 A000 AD 1F D0
```

Those three op-codes mean `LDA $D01F`

Two results: $2041 and $5C44. The first result doesn't appear to do much, I suspect it's clearing out the register because like $D01E, it holds its values until you read them and then clears them out.

The second result at $5c44:

```
5c44 LDA $D01F
5c47 STA $3B04
```

Looks good… it's storing the result of any collision into a location that it can check later. So let's replace that line of code. Remember it's three bytes (ad 1f d0) so it needs to be replaced with three bytes. Type:

```
A 5C44 LDA #$00

5C46 NOP
```

This will place a value of zero into the memory location $3b04 that is being used to check for any collisions with the helicopter's dropping things on you.

Exit and resume the game… and you've now got complete invincibility!

Just for fun, download The Machine which is a racing car game. It can be found at this link: (*link missing in original article)*

Enter the MLM and search for LDA $D01E using the command

```
H 0800 A000 ad 1e d0
```

There are many results but most of them load the register and do nothing with the result. The one at line $8400 however does some AND'ing which is a way of isolating which sprite has collided. AND is often used to isolate which bits are turned on. So now you'll type

```
A 8400 LDA #$00
```

And add the extra NOP at $8402 and now you can drive right through other vehicles.

Sometimes you want sprite collision to a degree, especially if you're trying to destroy other enemies. So you'll have to learn how to fine tune your skills and learn how games will use the AND command.

3.7 Lightforce – Infinite Lives

Download Lightforce (the game with the cool music) from this link:
http://tapes.c64.no/tapes/Lightforce.zip

Notice that you begin with 5 lives. So naturally we want to look for the command:

```
LDA #$05
```

in this case there are only two results: You already should know the command to search for this, I think at this point I don't need to tell you any more but I will one last time -> A9 05 which represents `LDA #$05`

One result is found at:

```
1A3E LDA #$05
1A40 STA $01
```

Right away we know this isn't what we're looking for because location 1 is the location we use to turn on and off the ROM when we want to use those 'houses' for RAM.

The second result is found at:

```
2D1B LDA #$05
2D1D STA $081C
```

And by playing the game and losing a life, so we're down to four ships, we enter the MLM and verify that location $081C holds a value of 4.

```
M 081C 081C
```

YEP!!!

And how will we lose a life? Most likely through a `DEC` (decrement) of that location. So we search for the bytes: CE 1C 08 (dec $081c)

And voila, location $379B is the line we want to change. So we'll change it to three `NOP` commands. You know how by this point:

`A 379B NOP` (press return and type two more NOP's)

There you are, infinite lives!

3.8 Toy Bizarre

Download Toy Bizarre from this link: http://tapes.c64.no/tapes/ToyBizarre.zip

You begin with four lives so we search for:

```
A9 04
```

which returns many results. Most of the time it's storing the value of '4' in zero page memory ($0000-$0100) and when I check the values of the locations it stored the 4 in, they are no longer holding a value of 4. So these aren't the right ones.

However when we get to the $2F0B:

```
$2F0B LDA #$04
$2F0D STA $17
```

We find that location $17 changes to a 3 when we lose a life and now have 3 guys.

We want to search for the DECREMENT $17 command. It's not the same as earlier examples though.

> Op-Code quick tip:
>
> `DEC $C000` = op-codes `CE 00 C0`
>
> `DEC $C0` = op-codes `C6 C0`
>
> Notice that $CE is used for decrementing four digit memory locations (that is, above zero page) while $C6 is used for decrementing zero page locations (below $0100).

And a result for `H 0800 A000 c6 17` is found at $3033. Now we're NOT going to NOP out that because the next line of code reads:

3035 BPL $3062

When you decrement or increment a location, you can then execute a BPL command to branch off if the value is anything but zero. So in this case once it decremented location $17 it would call the BPL to go to location $3062 is location $17 was anything but zero (you still had lives left).

What this did originally was branch off to $3062 if you still had lives left. Knowing the line before was LDA $17 if we change it to two NOP's it will look like this:

```
NOP
NOP
BPL $3062
```

Which means… do nothing, but branch to $3062 if the result is above zero. But as we haven't loaded anything into A, this could be catastrophic. It means that we're relying on whatever is in the "A" register instead of location $17. Who knows what that could be…

Best course of action just change the `DEC $17` to a `LDA $17`. Instead of decrementing, it will just re-load your number of lives.

```
A 3033 LDA $17
```

and you've got infinite lives!

3.9 Ghosts and Goblins 1994

Download Ghosts and Goblins 1994 from this link:
http://gb64.com/game.php?id=18820&d=18&h=0

Now you see four lives at the bottom but you actually have five (including the life you begin with). So let's look for the code `LDA #$05`. You should know how by now and you'll find a result at $087E

```
087E LDA #$05
0880 STA $359A
```

And it so happens by viewing memory for location $359A that it equals the number of lives we have even after we lose a life. So for infinite lives we search for:

`CE 9A 35` (which translates to `DEC $359A`)

We find a result at $0936 so we'll change just one byte. At $0936 we see a $CE which means Decrement. Type:

```
M 0936 0936
```

And cursor up and just change the CE to an AD. This changes the `DEC`rement to a LOAD instead. Much easier than typing :

```
A 0936 LDA $359A
```

There's infinite lives for you!

Once you become comfortable with knowing op-codes you can just type over top of the old op-codes rather than assembling new code over top of the old code.

Now as for infinite time… we know that you lose a life at around line $0936 right? And we know that the time is reset when you lose a life.

Look at line $0936

```
0936 DEC $359A
0939 BEQ $093E
093B JMP $08B0
093E LDA #$01
```

This says:

decrement a life, is it EQUAL (to zero) then jump to $093E otherwise jump to $08b0. Now when you `DEC`rement a location you can perform a `BEQ` (branch if equal). And if the result of the location you just decremented is equal to zero, it will branch off. So we know that if you lose a life and it's your last life, you go to line $093e. Otherwise you JUMP to $08B0. For clarification on `BEQ` and `BNE` see Section B for reference.

Let's look at $08B0 to see if we can find the time…

Look closely at $08BF

```
08BF LDA #$02
08C1 STA $359B
08C4 LDA #$59
08C6 STA $359C
```

Doesn't loading a 2 and loading a 59 seem awfully coincidental with our beginning time of 2:59 minutes? Hell yes.

So we're going to look for `CE 9C 35` (`DEC $359C`)

```
H 0800 A000 CE 9C 35
```

Three results: 0D94, 0ECD and 0EE6. However if we set all three of these to `LDA $359C` instead of leaving them as DEC(rement), the timer still counts down. Is this not the countdown timer location?

Let's try another approach before we give up. We know that you can also use the SBC command to Subtract, which would work for a countdown timer. So let's look for that code. But first we need to know what op-codes to look for.

So let's type in the code at a 'safe' location which will be $0400 (screen memory).

The commands will be

```
SBC #$01
STA $359C
```

This code subtracts 1 and stores the result in $359C

So we type

`A 0400 SBC #$01` (press return)

```
0402 STA $359C
```

And it shows us the bytes to look for are: `E9 01 8D 9C 35`

`e9 01` = `sbc #$01`

`8d 9c 35` = `sta $359c`

and now with high hopes we type:

```
H 0800 A000 e9 01 8d 9c 35
```

which it finds at $0F3C

so we change line 0F3C to `SBC #$00` which we can do two ways:

```
A 0F3C SBC #$00
```

Or type in `M 0F3C` and move the cursor over the E9 01 and change the E9 01 to E9 00.

Exit and resume the game… did it work? Yes it did!

TIP: When working with countdown timers it's important to keep in mind that some games will subtract your time when you complete a level, and award you points. If you've set the game for infinite time, this COULD cause an endless loop as the time never counts down to zero.

TIP 2: In time you'll come to learn the op-codes by memory. You'll know that e9 01 8d can by used with the H command in your MLM if you add onto them the high and low bytes to those numbers, of the location you're interested in. In this case it was $359C.

4.0 Whittling Method

Unfortunately due to personal issues in my life, I'm unable to dedicate more time to complete this guide from this point on. However the guide is 99% completed.

The last method I want to show you is the 'whittle' method. While not an advanced technique, it can certainly work for the novice person.

The whittle method works like this…. You load and run your game, making note of how many lives you have (or energy, or weapon, or time, etc.)

Step 1: Save the memory into two or more large files.

```
SP "A" 0000 0100
SP "B" 0100 0800
SP "C" 0800 A000
SP "D" A000 FFFF
```

We have the zero page saved as file A, we have $0100 to $0800 as file B, and then two large chunks of memory in files C and D.

You could also save the files into two files only:

```
SP "A" 0000 5000
SP "B" 5000 FFFF
```

It doesn't matter – just save all the memory into large chunks. It doesn't matter how many files you choose or what start and ending addresses you use. Just save ALL the memory to disk.

Step 2: Press F12 to freeze the game as it is.

Step 3: Lose a life, or change weapon, or lose energy or complete the level, etc. You want to decrease or alter the characteristic you're trying to train.

Step 4: Load back in the files, one at a time and see if your energy/lives/etc has been restored.

For example, you'd load R-Type, Gryzor, Pac Man, Blue Max, etc.

As the game begins, you'll save the memory into large chunks as per Step 1. Let's say you have four files: A, B, C and D.

You'll lose a life or drain some energy or lose some time. You'll then press F12 to save the game in this state.

You'll then enter the MLM and type:

```
LP "A"
```

Type X to exit the MLM

Step 5: Examine the energy or lives to see if they've gone back to their original values.

If not, repeat step 4 but load in file B this time… and repeat for files C, D, etc.

Step 6: Once you've loaded in a file and your lives go back to what were before (that is, your energy is restored or your lives went back to 3 instead of 2, etc.) you know which file contains the memory location you want.

Remember to only load one file at a time and then exit the MLM to check if that files worked. Don't load them all in at once.

Step 7: Once you've found the right file, you 'whittle' the memory into even more sections. You'll then press F11 to restore the game back to the original number of lives/energy/weapon/time/etc and repeat the process.

Putting it into practice:

Just to demonstrate what I mean, in case you don't follow. We're going to use Blue Max as a demonstration. I've loaded and started Blue Max and am now going to save the memory into two files.

```
SP "A" 0800 5000
SP "B" 5000 FFFF (making sure ROM is turned off)
```

I press F12 to capture the game with my fuel remaining set to 199 (full time remaining).

I play the game and watch the fuel drop down to 195 (or whatever).

Now I enter the MLM and begin to load in the files, one by one to see which one resets my fuel back to 199.

```
LP "A"
```

and it works… the time has been reset back to 199. So somewhere between $0800 and $5000 is my time. We didn't need to load file 'B'.

Now I press F11 to restore my freeze (putting the game back as it was when I saved the memory). Now I whittle down the memory into even more files:

```
SP "A" 0800 1000
SP "B" 1000 2000
SP "C" 2000 3000
SP "D" 3000 4000
SP "E" 4000 5000
```

And I again load in the files one by one…. And make note of the fuel remaining in Blue Max. As it turns out, file E works. When I load in file E, the Fuel jumps back up to it's initial value.

Now we whittle down that memory into even more files. Press F11 to restore the game back to how it was when you started and whittle down that bank of memory even more.

```
SP “A” 4000 4800
SP “B” 4800 5000
```

It turns out when you exit the MLM, drain a bit of fuel and load in file A, the Fuel goes back up again.

So we’ve narrowed down the memory to $0800-$5000, then narrowed it down to $4000-5000 and then to $4000-4800.

Now let’s whittle it down even more…

```
SP “A” 4000 4100
SP “B” 4100 4200
SP “C” 4200 4300
SP “E” 4400 4500
SP “F” 4500 4600
SP “G” 4600 4700
SP “H” 4700 4800
```

As it happens, the first file works… no need to load any more files to see which one works! So we know that somewhere between $4000 and $4100 is the fuel.

Now I break down the files even more…

```
SP “A” 4000 4080
SP “B” 4080 4100
```

I press F11 to load the frozen game, and load in file A… it works.

So now I want to narrow down $4000 to $4080.

```
SP “A” 4000 4040
SP “B” 4040 4080
```

Now I load in file A and the fuel hasn’t changed. So I load file “B” and notice the fuel jumps back up in value.

So somewhere between 4040 and 4080 is the fuel!!!

```
SP “A” 4040 4050
SP “B” 4050 4060
SP “C” 4060 4070
SP “D” 4070 4080
```

I load file A, nothing happens… I load file B, nothing. However file C changes the fuel back to it’s original value.

$4060-$4070 holds the fuel.

Now you can do a few things here.

1) You can save the memory to disk and compare it to what's in $4060-$4070.

You'd do this by typing:

```
SP "A" 4060 4070
```

then you'd load in file "A" into memory, loading it into memory $c000

```
LP "A" C000
```

then you'd use the C (compare) command to compare what values are different

```
C 4060 4070 c000
```

This compares the original memory from $4060 to $4070 with the file you loaded into memory at $c000 ($c000 may be used by the game, so you'll want to choose an area that looks relatively unused). The results are: 4061, 4062, 4065, 4066, 4067 so we know those locations are changing as we are expending fuel.

I can tell you that $4060 holds $1e which is the bombs (30 bombs)
I can tell you that $4061 holds the hex value of the fuel remaining.

I found this by changing the values of $4060 and $4061 to $ff just to see if the time would change – and it did (as did the bombs). Setting $4060 to $ff changes your bomb value. Setting $4061 to $ff sets your fuel to 255.

So to create infinite time we'll look for `DEC $4061`

```
S 0800 FFFF CE 61 40
```

And there you have it…. Location 4605 is where the infinite fuel can be found.

What we did was broke down the memory into large chunks, then depending on which file contained the memory that held our item being drained (lives, energy, etc.) we broke down that memory into even more files. Eventually we 'whittle' the memory down into small enough areas that we can narrow down the location.

This can also work when trying to find a location that 'completes' a level (for level skipping). Note that when you use this method, you're liable to change things in the game such as background graphics, your location on the screen, etc. and it's not always pretty. This is why you have the F12/F11 to load and freeze the game back to normal.

Thanks for taking the time to read this file!

Section A – Additional Help on Op-Codes

Let's go back to the game 1942 that you loaded but didn't run it yet. Go into your MLM and type:

```
D 080D
```

This means "Hey I want you to disassemble the code at location $080D." Disassemble means show me in words that I can understand, in English, what you're doing.

You will see:

```
080D A9 93 LDA #$93
080F 20 D2 FF JSR $FFD2
```

To make sense of this, the first column is the memory location. In this case $080D is the first line, $080F is the memory location on the second line. The next two columns are the op-codes. The op-codes are the actual values in hex of what's in memory.

Basically if you were to go to a new line and type:

```
M 080D 080F
```

You will see

```
A9 93 20 D2 FF A9 8E
```

Notice that the numbers are the same and the ones when we typed D 080D. (A9, 93, 20, D2, FF, etc) To explain it so that you'll understand it… the M command shows us only the values of what's in memory – the actual numbers. If you type POKE 1024, 42 to put a star in the top left of the screen and you then went into your MLM and typed

```
M 0400 0400
```

It would show something like:

```
:0400 2A 20 20 20 20 20 20 20
```

Which means the first number ($2A) is what's at location $0400, the second ($20) is what's in memory at location $0401, $0402 and so on. If you cursor up and replace that 2A with a 2C and exit the MLM, that asterix has now become a comma. Why? Because we overwrote what was in memory at $0400.

TIP:

> In a machine language monitor the M command displays the beginning memory location in the first column follow by what is in those next 8 memory locations. This goes on for every column that you see – which is why the values in the first row increment by 8. You're seeing the memory in rows of 8 bytes.
>
> The D command will show us a disassembly of what those numbers mean. The op-codes are the actual numbers of what's in those locations. The far right is the actual 6502 commands.

So then…. We've typed in

```
D 080D
```

After the game 1942 was loaded, but not run yet.

We see:

```
080D A9 93    LDA #$93
080F 20 D2 FF JSR $FFD2
0812 A9 8E    LDA #$8E
0814 20 D2 FF JSR $FFD2
```

What we can make of this is that location $080D has an $A9 in it (169 decimal), location $080E has a $93 in it, location $080F has a $20 in it. We know if we type

`M 080D 0812` that we see those numbers in those memory locations. Here in the Disassembly (typing D 080D in your MLM) we not only see those same numbers but their actual meaning to the C64's 6502 processor.

At location $080D we're LOADing a $93

At location $080F we're `JSR`'ing to FFD2 (in Basic this would be `GOSUB`)

At location $0812 once the computer returns from the `GOSUB` to FFD2 at the line above it again `LOAD`s a value of $8E and again JSR's (Jump subroutine) to $FFD2.

Just for fun try typing…

```
M 080D 0810
```

You see

```
:080D A9 93 20 D2 FF A9 8E 20
:0815 D2 FF A9 00 8D 20 D0 A9
```

Scroll up and change the 93 in the first row to a 42 and press return. Get to a new line with nothing on it and type again

```
D 080D
```

And we see

```
080D A9 42 LDA #$42
```

So the original code was

```
080D A9 93 LDA #$93
```

Has now become

```
080D A9 42 LDA #$42
```

Because we changed one value. And here we see the impact of changing that value. It Loads a $42 instead of a $93. The D and M commands in a MLM are showing you the same data except the D is to show you the actual 6502 commands and the M is just a memory dump without any explanation as to what the commands are.

Section B – BNE/BEQ

BNE means Branch if Not Equal. BEQ means Branch if Equal. It works like this:

```
LDA $0400
CMP #$2A
BNE $xxxx
RTS
xxxx
```

In BASIC:

```
10 a = peek(1024)
20 if a <> 42 then 100
30 end
100 print “a does not equal a *”
```

The above code loads $0400 into A (the Accumulator as it’s called). The next line says COMPARE what’s in A to #$2A. The next line says BRANCH if NOT EQUAL to xxxx. So if you put an asterix in the top corner of the screen (location $0400, and the ASCII code for an asterix is #$2a) it will fall through to the RTS (return from subroutine). Basically is there is an asterix in the top left of the screen it will just return to the Ready prompt if you called this actual code from memory.

If there ISN’T an asterix in the corner of the screen it will branch off to xxxx because it’s a “branch NOT equal”. That is, what you had loaded into A and compared to a value of #$2A was not equal to one another.

In comparison….

```
LDA $0400
CMP #$20
BEQ xxxxx
RTS
xxxx
```

In BASIC:

```
10 a = peek(1024)
20 if a = 42 then 100
30 end
100 print “a equals a *”
```

The above code loads whatever is located in the top left corner of the screen (location $0400 also known as location 1024 decimal). It compares what it just loaded with #$20 (which is a value of 32 – or – an ASCII space). If there’s a blank space in the top left of the screen it is a BRANCH EQUAL because the results are equal. What you loaded into A (location $0400) is equal to what you compared.

In machine code that would be a BRANCH EQUAL because it branched off with the result being equal to what you compared it to.

Section C – Final Comments

- If your game shows a character in the bottom right of the screen that changes colour during de-crunching, this is likely Exomizer Cruncher. Replace your search bytes `a9 37 85 01` with `C6 01` which translates to "`DEC $01`". This should reveal the final JMP$ you're looking for.
- Often times you can change the final `JMP` to a `JMP $A474` instead of creating a loop. This will jump to the READY. Prompt where you can save the raw data (or type LIST to see if it's a BASIC program underneath the program being de-crunched. In the case of Defender for example you can change the `JMP $3000` to `JMP $A474`.
- In CCS64 you can type ?$xxxx where xxx is an address, and it will show you the decimal value (e.g. ?$C000 will result in 49152)
- You can convert decimal to hex by typing: ?#xx where xx is a number (either 2 or four digit hex)
- If you're trying to isolate part of a game where you think you're losing a life, I like to change a line of code where there's three bytes (example: `LDA $4000`) to `INC $D020`. Then I resume the game – if the border colour changes colour because of the INCREMENT BORDER COLOUR command I just put in, I know I'm in the right place.
- Sometimes you'll notice that I tell you to replace the `DEC` command with three `NOPs` and other times I told you to replace it with a `LDA` command. Why is this? Well sometimes you can get away with using `NOP` commands if the code that comes after your initial `DEC`rement code has nothing to with the decrement.

  This would be a safe place to use `NOP`'s:

  ```
  DEC $45
  LDA $50
  STA $66
  ```

  Replacing the `DEC $45` will have no effect on the code after it.

  However in a case like this:

  ```
  DEC $45
  BPL $C050
  LDA $50
  STA $66
  ```

  That `BPL` command relies on the decrement $45 to make a judgement call on whether to branch off or not. If you put in `NOP`'s then the `BPL` command would rely on whatever code came BEFORE your `NOP`'s.

  Lastly, you should know that it's not always going to be the "A" that is used for loading and storing. There are in fact three registers: A, X and Y. These are much like variables in BASIC.

  A=20
  X=30
  Y = 44

A is used primarily for maths (this is why it's called the Accumulator) whereas X and Y are used secondarily. For training purposes though you shouldn't rely solely upon looking for `LDA` and `STA`. There are also `LDX`, `STX` and `LDY`, `STY`. The examples I've provided you with should be enough to get started on training. Once you're comfortable with this tutorial, it's just a matter of also searching for A2 and A0 instead of A9 (A0 = `LDY` and A2 = `LDX`)