C64 Assembler Tutorial

By Cruzer/CML

Updated 2001-12-29

Contents

Intro

This tutorial is intented for newbies who wanna code c64 assembler for the first time, aswell as oldies who want to get it freshened up after years of lameness... I will try to write it as softcore/easy as possible, so everyone should have a chance to try out the magical world of C64 machine code/assembler.

If you get hooked and want to learn more I can recommend the chapter "Basic to Machine Language" from the good old C64 Programmers Reference Guide, which can be found online right here on C64.CH

First of all, to give the newbies an idea of what machine code is, let me quote the mentioned C64PRG. (You can skip it if you think you know what it's about)...

The Bible

What is Machine Language?

At the heart of every microcomputer, is a central microprocessor. It's a very special microchip which is the "brain" of the computer. The Commodore 64 is no exception. Every microprocessor understands its own language of instructions. These instructions are called machine language instructions. To put it more precisely, machine language is the ONLY programming language that your Commodore 64 understands. It is the NATIVE language of the machine.

If machine language is the only language that the Commodore 64 understands, then how does it understand the CBM BASIC programming language? CBM BASIC is NOT the machine language of the Commodore 64. What, then, makes the Commodore 64 understand CBM BASIC instructions like PRINT and GOTO? To answer this question, you must first see what happens inside your Commodore 64. Apart from the microprocessor which is the brain of the Commodore 64, there is a machine language program which is stored in a special type of memory so that it can't be changed. And, more importantly, it does not disappear when the Commodore 64 is turned off, unlike a program that you may have written. This machine language program is called the OPERATING SYSTEM of the Commodore 64. Your Commodore 64 knows what to do when it's turned on because its OPERATING SYSTEM (program) is automatically "RUN."

The OPERATING SYSTEM is in charge of "organizing" all the memory in your machine for various tasks. It also looks at what characters you type on the keyboard and puts them onto the screen, plus a whole number of other functions. The OPERATING SYSTEM can be thought of as the "intelligence and personality" of the Commodore 64 (or any computer for that matter). So when you turn on your Commodore 64, the OPERATING SYSTEM takes control of your machine, and after it has done its housework, it then says:

READY.

The OPERATING SYSTEM of the Commodore 64 then allows you to type on the keyboard, and use the built-in SCREEN EDITOR on the Commodore 64. The SCREEN EDITOR allows you to move the cursor, DELete, INSert, etc., and is, in fact, only one part of the operating system that is built in for your convenience. All of the commands that are available in CBM BASIC are simply recognized by another huge machine language program built into your Commodore 64. This huge program "RUNS" the appropriate piece of machine language depending on which CBM BASIC command is being executed. This program is called the BASIC INTERPRETER, because it interprets each command, one by one, unless it encounters a command it does not understand, and then the familiar message appears:

?SYNTAX ERROR

READY.

Okay, so now you know that... But how do we type in the code? There's basicly these alternatives: A **machine code monitor**, an **assembler** and a **cross-assembler**. A machine code monitor is a little program built into cartridges like The Final Cardridge and Action Replay which lets you write machine code and examine code from other programs, like demos. An assembler is like a machine code monitor, just with some extra features which makes it much easier to write big programs. The disadvantage is that it eats up quite a few valuable bytes in the computer (typically about half the mem) so it's not ideal for memory intesive effects. A solution for this problem is to use a cross-assembler which is placed in another computer (a PC for example) which is connected to the c64 via a special cable to transfer the raw machine code.

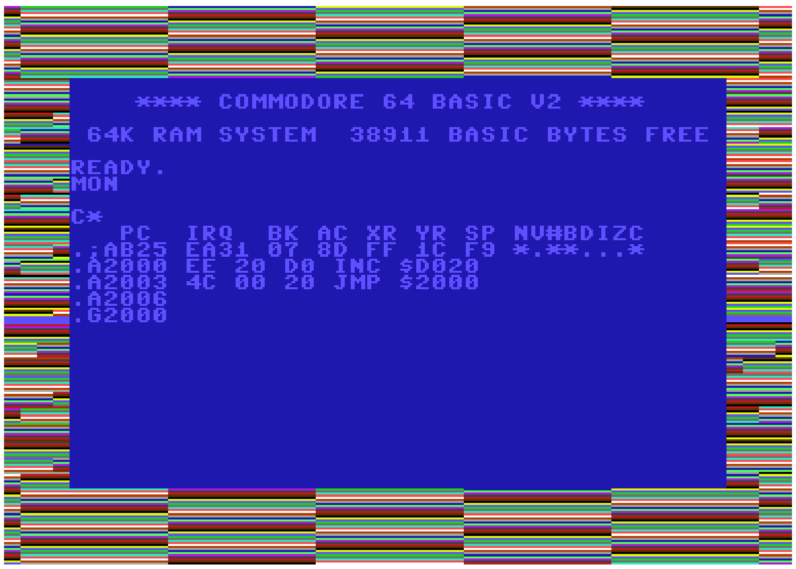
Machine Code Monitors

This is IMHO the best way to learn machine code, since you can easily see what's going on in the machine. The first thing you need to get started is a cartridge with an MC-mon (I use The Final Cartridge 3.) If you're using an emulator all you need to do is download a cardridge file, and attach it to the emu. For example in Vice select **file -> attach cartridge image -> CRT file**, and select the file... You can download The Final Cartridge 3 here. Remember to reset the emu to make it work (alt-r in Vice).

Now, let's begin the fun by typing **MON**. Then some strange stuff appears, and you're ready to go. Okay, time for our 1st piece of code (finally!) Type this:

```
.A2000 INC $D020
.A2003 JMP $2000
```

"A2000" means "Assemble from address 2000 (hex)". All addresses and numbers are hex numbers, but more about that in a little while. The address could have been any other from $0000 to $FFFF, but be careful, cuz some of them are reserved for other purposes, so until further notice we better stay within $1000-$9FFF. After you press return on each line some weird numbers and letters appear on the line, and the code you entered is moved to the right. Don't worry, that's the way it s'posed to be! ".A2003" should also appear automaticly, so there's no need to type that yourself. After the last line ".A2006" appears, just press return here to exit the assembly mode, and type the magic command **G2000** (means "goto address 2000") to start... If everything went well you have now made your first machine code program: **flickering border color** ... Woah!...



```
    **** COMMODORE 64 BASIC V2 ****

 64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
MON

C*
   PC  IRQ  BK AC XR YR SP NV#BDIZC
.;AB25 EA31 07 8D FF 1C F9 *.**...*
.A2000 EE 20 D0 INC $D020
.A2003 4C 00 20 JMP $2000
.A2006
.G2000
```

Whenever you feel you have enjoyed it enuff, you can stop it by pressing **Run/Stop+Restore** or if you're on emulator it's **Esc+Page Up** (atleast on Vice.) Alternatively you can also reset the computer. Remember to start the mon again.

Hex Numbers

Okie dokie, time to explain a bit of what's going on. The weird numbers in the MC-mon are called hex numbers, and the advantage of 'em is that it's easy to translate between binary numbers (0's and 1's) and hex numbers, since a hex digit equals 4 bits. While normal numbers (aka decimal numbers) are based on 10 digits (0-9), hex numbers have 16 (**0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f**). This means that the numbers from **a-f** have the values from 10-15, and 10 in hex means 16. Confused? Then you can play around with the numbers in the mon by typing any 4-digit hex number like **$C74F** or **$0003** and get the corresponding decimal number, and vice verca. Remember to preceed hex numbers by a $-sign and decimal numbers by a #. You can also play with 'em in the windows calculator, in scientific mode. Here you don't need $'s and #'s, you can just press **hex** and **dec**. Also note that you don't need $-signs before hex numbers in mon-commands like **A2000**, **G2000**, etc.

Here's a little table that might clear things up a bit...

Decimal	**Hex**	**Binary**
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000

9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111
16	10	00010000
17	11	00010001
18	12	00010010
32	20	00100000
64	40	01000000
128	80	10000000
255	FF	11111111

Move Commands

Time for some commands, which gives you the power to command the commodore to do anything you want it to. The first ones we're going to look at are in the category "move commands", since they can move data. On other platforms you often have a command called move or mov, but on a c64 they're called other things like **LDA**, **STA**, etc. The principle of move commands on c64 is that either you move something into a processor register, or you move from a register to a memory address. The registers are called **A** (accumulator), **X** and **Y**. They can each hold a byte, which is a number from **0** to **255**, or in hex **00** to **FF**, or in binary **00000000** to **11111111**. Let's have a little example, which you can type in and test like the previous piece of code... (Remember to type .A2000 before the 1st line and all that!)

```
LDA #$07
STA $D021
BRK
```

Again, start it with **G2000**, and the amazing result should be a yellow screen.

Explaination:

LDA #$07 means *"load accumulator with the value 7"* and moves the number 7 (the color code for yellow) into the accumulator (aka the A register.)

STA $D021 means *"store the value of the accumulator into the memory address $D021"*, which is where the background color is located.

BRK breaks out of the program so you return to the mon. This is necessary because the code doesn't loop forever like in the flicker example.

Instead of **LDA #$07** we could also write **LDA $07**, which would mean that it was the content of the address $0007 we were moving into the accumulator. That's ofcuz a whole another story, so please notice that there's a big difference between **LDA value** and **LDA address**. Always remember the number sign (**#**) when it's a value you wanna LDA! If you try this...

```
LDA $D021
STA $D020
BRK
```

... it moves the background color to the border. Or maybe "move" is the wrong word since it doesn't change the source address, it just copies it. As you might have guessed $D020 and $D021 controls the border/background color. Everything that has to do with graphics is controlled by the addresses that start with $D0, but more about that later.

Time to see what we can use the **X and Y registers** for. Every time we write LDA or STA we could just aswell have used **LDX**/**STX** or **LDY**/**STY**. That would have resulted in the same effect, the only difference is that it's the X and Y registers that are used instead of A. But these registers can also be used for some more purposes. Let's try this...

```
LDX #$21
LDA #$0B
STA $D000,X
BRK
```

This is an example of socalled **relative** addressing. The first line loads X with the value $21, in the next line A gets the value $00, and in the 3rd line the value of A is moved to the address **$D000+X**, which means $D021 in this case. (The background color again... I'm really original, huh?)... Y can also be used for realive addressing (**STA $1234,Y**), and you can also LDA relaive (**LDA $1234,X**)... The relative addressing modes are very useful in loops, but more about that in the Compare/Branch section.

Another type of move commands is called *transfer commands*, and they move data within the registers. An example is **TXA** which means *transfer X to A*. So if X held the value $DF, A is now $DF. As in other move commands the source (X in this case) is not affected, so they both holds the value $DF now. Here's some more transfer commands is: **TXA**, **TYA**, **TAX** and **TAY**. They all transfer the 1st register to the 2nd **-** eg. TAY means A -> Y.

Well, we haven't really covered all of the move commands, but I guess it's time to move on, so it doesn't get too boring...

Math and Logical Commands

Now, let's see how we can alter the bit 'n' bytes instead of just moving 'em around. The math/arithmetic commands lets you calculate and manipulate the bytes, either directly in the memory, or in the registers. For instance, in the first example we had a command called **INC $D020** which made the border color flicker. This command means "increment $D020" and as you might have guessed, it adds 1 to the given address. If the content of the address has reached maximum ($FF/255) it just restarts from 0. A related command is **DEC**, which works the same way, just other way around, which means it decrements the value by 1, and restarts from $FF if the value was allready 0. Let's have another color flicker example...

```
INC $D020
DEC $D020
JMP $2000
```

This time it just flickers between two colors, since it first adds 1 and then subtracts 1.

You can also inc/dec the X and Y registers (but not A for some strange reason.) This is done with these commands: **INX, DEX, INY,** and **DEY**. So we could also make color flicker this way...

```
LDX $D020
INX
STX $D020
JMP $2000
```

This just looks a bit different since it's slower first to move the color value into the X register, then increment it, and then move it back to the memory address.

Ok, say you want to increment a byte by 27. Then you could ofcuz just write an INC command 27 times. But a smarter way would be to use the command **ADC** (*add with carry*) which can add any given number to the accumulator. The following piece of code adds 27 to the address $0400, which by the way is the char in the upper left corner of the screen, so you should see some kind of change there...

```
LDA $0400
CLC
ADC #$1B
STA $0400
BRK
```

As you might have guessed **1B** is the hex value for 27. What you might not have guessed is that **CLC** means **clear carry flag**. We will get into flags later, but intil now you just have to remember to **clear carry before adding**, and **set carry before subtracting**. Here's an example of the latter...

```
LDA $0400
SEC
SBC #$1B
STA $0400
BRK
```

SEC means **set carry** and SBC means **subtract with carry**. Like most other commands ADC and SBC has lots of different addressing modes. For example **ADC $1234** which means *add the value found in address $1234 to A*. So it does not affect the address, only A. To make it affect the address you must STA it afterwards. Remember the relative addressing mode? It also works for ADC/SBC - eg. **ADC $1234,X**

If we can add and subtract, can we then multiply and divide too? Not quite, I'm afraid. It's only a C64, you know! But we can push all the bits in a byte left or right, which is alsmost the same as multiplying or dividing by 2. For instance `LSR $0400` shifts the bits in address $0400 to the right, so if you had the value 6 there it will now be 3. However, if you LSR it again it will now be 1, because the rightmost bit is thrown away. `ASL $0400` will shift them left which is the same as multiplying by 2. So the original value was 100 it will now be 200. This trick also has its limitations since a byte can only be 255 at max, so if you try to ASL a value greater than 127 (hex 7F) you will not get a correct result. You can ASL/LSR as many times as you want, and thereby multiply/divide by 2,4,8,16...

If you wanna multiply/divide by other values it's a bit trickier. The following code uses a combination of ASL and ADC to multiply by 3...

```
LDA $5000
ASL
CLC
ADC $5000
STA $5000
BRK
```

As you can see we can also just type **ASL** without any address. This means that it's A that's shifted left. I think I'll explain the previous example a bit more so it's clear what happens for everyone. Let's say the value of $5000 is 4, and then let's see what happens with the accumulator...

```
LDA $5000 A=4
ASL       A=8
CLC
ADC $5000 A=12 (or $0C in hex)
STA $5000
BRK
```

If you want to play around with this some more you can use the mon command **M** which lets you view the content of the memory. For example **M 5000** lists the 8 bytes from $5000 to $5007. Then you can change the values, and after pressing return it will take effect. For example you can change the value of $5000 to 04, then run the previous piece of code, and then type **M 5000** again to see if it has changed to 0C.