TI-83+ Z80 ASM for the Absolute Beginner

LESSON ONE:

- *Introduction to the Method Used in this Course*
- *The Calculator's Version of Russian or Spanish*

INTRODUCTION TO THE METHOD USED IN THIS COURSE

I find it necessary to start this course with a short introduction to myself concerning Z80 ASM. I could program in Ti-Basic, but I found that none of the games I wanted to program could be done fast enough in TI-Basic. Even Celtic III and Xlib were not much help. Sadly, there are several very good ASM tutorials on the internet, but I found myself incapable of understanding them for a while, no matter how much I worked on them. My difficulty was not my unwillingness to work through them; rather, it was the fact that the system simply did not work for me.

But I am happy to say that once one understands ASM, it can become an easy language. The trick is, it's not hard to program in ASM…the difficulty arises in understanding it. Z80 ASM is now the one programming language I remember the most out of all the languages I have learned, and the RTS game I am programming, S.A.D., http://www.omnimaga.org/index.php?board=60.0, shows how much I am beginning to develop a foundation in this language.

My point? Once I understood the language it was easy, but it took me three years to understand it. In essence, I had difficulty with the methods the tutorials used to teach me ASM. And I have a feeling that this applies to a lot of people: I feel that if the method used were easy to work with, a lot more people would be programming in Z80 ASM, and we'd see as many ASM games each month as Ti-Basic games out there. It is my goal to provide such a simple-to-use method.

Interested? I only recommend knowledge of Ti-Basic, and I'm assuming that if you want to learn Z80 ASM, you know some Ti-Basic.

The approach I'm going to take to this language is teach less in more time. A tutorial like "Learn Z80 ASM in 28 days" attempts to teach you absolutely everything you should introduce yourself to in a month. ***But there's a lot of unnecessary information in that tutorial***. And once you read a lesson in that tutorial, you have to read it two times, maybe three times, and still might have difficulty. My goal is to provide only the most important information, meaning that there will be very little of the hardware of a calculator. I am making sure you can understand a lesson in at most two attempts. I will provide diagrams, numerous examples, and sample problems (which I expect you to do) for each lesson.

Know this: You will not leave this course ready to program an RPG or a side scrolling game. However, you will leave ready to be able to read "Learn Z80 ASM in 28 days" and learn/understand even more about Z80 ASM. This tutorial will also provide enough information so that when I can't teach you how to do something, you can find routines and asm programs that do it for you. For instance, I'm not going to teach you how to draw lines in ASM, but there's routines people wrote that allow you to draw lines, and I'll make sure you have the knowledge to be able to use them in your program by yourself. Similarly, I'm not going to teach you how to write a program to fully black out a screen, but people have written routines to black a screen that they will allow you to use in your program.

If you're ready to start, your first lesson begins on the next page.

THE CALCULATOR'S VERSION OF RUSSIAN OR SPANISH

We're not going to do any programming today. But did you ever wonder how to talk to the Ti-83+ calculator? Let's start this lesson with a fun exercise. Below is a list of words and their equivalent code. I'll give you a bunch of these numbers, and you can translate them into English.

1 = Throw	11 = Cat	21 = Into The	31 = Tree
2 = Eat	12 = Spaghetti	22 = Outside The	32 = Bar
3 = Walk	13 = Dog	23 = Under The	33 = Bridge
4 = Smash	14 = Computer	24 = Against The	34 = Wall
5 = Move	15 = Lamp	25 = Above The	35 = Skyscraper
6 = Play	16 = Billiards	26 = On The	36 = Water
7 = Cross	17 = Street	27 = In The	37 = Morning
8 = Fight	18 = Sister	28 = For The	38= Front Seat

So, 1 16 27 32 is Throw Billiards In The Bar,

and 2 15 23 31 is Eat Lamp Under The Tree.

Go ahead, try the next few numbers, and then make some up!

1. 7 11 25 36
2. 8 12 26 38
3. 5 13 22 32
4. 6 14 24 34
5. 3 18 27 17
6. 4 11 24 35

Believe it or not, this is **exactly** how the calculator speaks and is told what to do: a series of numbers. When you write a Ti-Basic program for the Ti-83+, the words such as "Line", "Text" and "For" are translated into a series of numbers that the calculator reads. Rather than words, however, the calculator reads these as instructions. For an unrealistic/hypothetical example,

021 244 231 192 096

might tell the calculator "3 → A". A much longer sequence of numbers, such as

045 012 195 095 001 084 201 065 196

might tell the calculator to display the character "A" on the home screen. The idea behind ASM is, like Ti-Basic, using words to represent these numbers that tell the calculator what to do. However, we'll worry about that later.

The calculator does not read numbers quite the same way we do. Here's a way to look at it:

In our language, that is a cat. In Spanish, this is a ***gato***. In Russian, this furry critter is called a ***koshka***. They all mean the same thing, but different words. So it is with the calculator: when we say 136, the calculator might say "10001000." They mean the same number, but different ways of saying the same number.

To explain the calculator's different language, let's take a look at our numbering system. Start with 0, and count by one number at a time.

00 01 02 03 04 05 06 07 08 09 **10**

Notice what happened? After the ones place reached 9, the biggest it can be, it reset itself to zero, the smallest it could be. The tens place increased by one. This is something we don't often think of, but it's very important.

10 11 12 13 14 15 16 17 18 19 20

Now, once again, the ones place reset itself to 0, and once again, the tens place has increased by one.

To save time, let's continue by counting by 10s.

20 30 40 50 60 70 80 90…96 97 98 99 **100**

Now what happened? In the number 99, the ones place was at its maximum, 9, so it reset back to zero. The tens place was supposed to increase by one, but it also was at 9, its largest. Therefore it also reset to 0. Thus, the **hundreds** place reset itself to one.

Now suppose that we didn't have 9 as a maximum: what if we had only 1 as a maximum? That means we don't have 2, 3, 4, 5, 6, 7, 8 or 9. We only have 0 and 1. This is exactly the way it is with the calculator. This is called counting in **binary numbers**. Let's start counting in binary on the next page.

Once again, start with 0, and let's count by ones, but in binary.

0 1 10 Since we have a maximum of 1, the number at the right resets itself to zero, and the number at the left resets itself to one. (We don't use the words ones, tens, hundreds etc. place in binary numbers)

11 100 101 110 111 1000 …

So why is 1 a maximum? The calculator, which contains a whole bunch of electronic components, can only tell if an electronic flow is on or off. If an electronic flow is on, the calculator reads a "1". If an electronic flow is off, the calculator reads a "0". Thus, a calculator cannot read 2, 3, 4, 5, 6, 7, 8 or 9 in a digit. In other words a calculator reads a whole bunch of these "ons" and "offs."

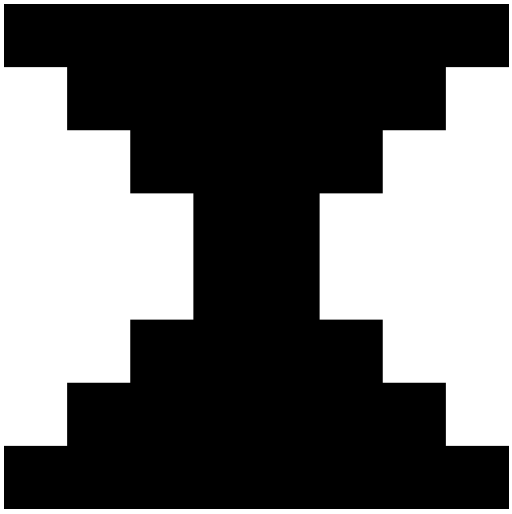
A single one or a zero is called a **bit**. However, most numbers the calculator reads—and translates into instructions—are composed of 8 bits or 16 bits. Thus, there are some instructions a calculator receives given by numbers from 0 to 255, where 11111111 (eight bits at their maximum, 1) equals 255. Other instructions a calculator receives, with 16 bits, are given by numbers from 0 to 65535. A number of 8 bits is called a **byte**, and a number with 16 bits is called a **word**.

Just remember that Decimal numbers and Binary numbers are different "languages," meaning two different ways of representing the same number.

Our Numbers (Called Decimal)	Calculator (Called Binary)
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100

Since the goal of this series of lessons is not to tell you the unnecessary, I'm going to avoid telling you how to translate back and forth. You can read about translating back in forth in Appendix A of these tutorials. However, I almost always use a calculator when I need to translate from Decimal to Binary and from Binary to Decimal. Windows provides such a calculator, and you can also find these online. http://acc6.its.brooklyn.cuny.edu/~gurwitz/core5/nav2tool.html is one such online site.

When you program in ASM and you need to write numbers instead of words (such as values for variables), you can always write in Decimal Numbers, such as 15 and 16, and when you compile this, the numbers will be translated into binary numbers. However, it is important that you understand how binary numbers work, because there are times that you want to use binary numbers directly rather than decimal numbers. For example, suppose I wanted to put this picture into the calculator:



When you program in ASM, you have to create this picture as a series of numbers. Now, there's two ways of designing this picture as a series of numbers. One way is to use our numbers:

255 126 60 24 24 60 126 255

Recalling that a calculator reads numbers, this is EXACTLY the numbers in our language that tell the calculator what this picture is. However, what would it look like in binary numbers? Look on the next page!

```
11111111
01111110
00111100
00011000
00011000
00111100
01111110
11111111
```

In binary numbers, each "1" means a black dot, and each "0" means a white dot. Now, which is easier to understand, and which is easier to edit?

I'll give one more example of the importance of understand binary numbers. Suppose you wanted to write a Mario game in Ti-Basic.

Let's say you have a bunch of yes and no values: Is Mario alive? Is he swimming? Is he flying? Does he have a fire flower? Is he invincible? Is he at the end of the level? In Ti-Basic, you might use 6 variables: A, B, C, D, E and F. Then each variable might be equal to 1 if the answer is yes, or zero if the answer is no. Let's say Mario is alive, has a fire flower, and is swimming.

Mario is:	1 = Yes or 0 = No
A = Alive	1
B = Swimming	1
C = Flying	0
D = Has Fire Flower	1
E = Invincible	0
F = At the End of the Level	0

Using six values like this is hard to work with, and uses a lot of RAM. However, did you notice that there's only two values for each variable? In ASM, you could put these into a single variable:

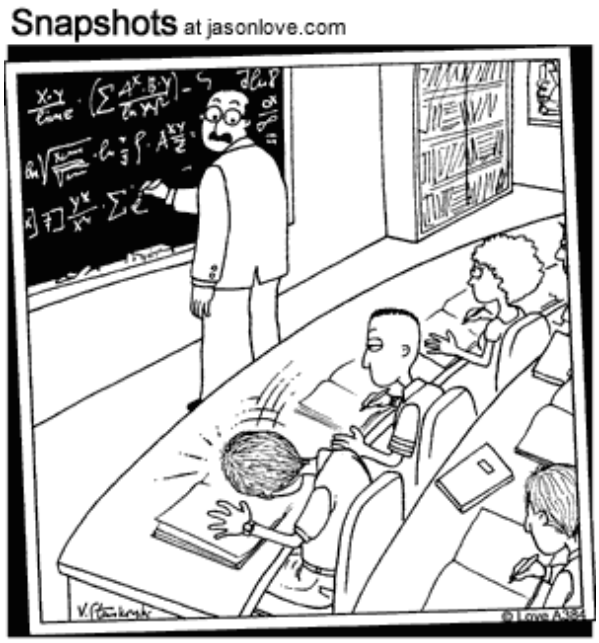
1 1 0 1 0 0

Alive, Swimming, Flying, Has Fire Flower, Invincible, End of Level

As a Decimal Number, this is 53. In binary, this is 110100. Which is easier to visualize?

Now, before you start throwing sticks and stones at me, let me explain why I wrote a whole section on counting and numbers. You might say to me, "But Hot Dog, you just told me that ASM uses words that translate to numbers. Why do we give a hoot about numbers if we can just type in words?" Because when you write in ASM code, you are only a hairline away from writing in numbers, aka the language of the calculator. Counting happens A LOT in ASM. You will understand in future lessons how much counting actually comes up. Here's an example:

In Ti-Basic, you can easily solve the problem 9 * 6. When you put 9 * 6 into a program, or on the home screen, the calculator translates this problem into numbers—instructions—that it can understand. However, in ASM, you have to tell the calculator how to solve this problem; it does not know how to do it! I'm serious, the Ti-83+ calculator does not know how to do something as simple as 9 * 6, you have to tell it how to do so. You have to give these instructions to the calculator manually. There's several ways to do this, for example, "Solve 9 + 9 + 9 + 9 + 9 + 9." In other words, you're counting 9 six times in ASM.



Professor Herman stopped when he heard that unmistakable thud -- another brain had imploded.

Next week, we'll look at how the calculator is similar to our brain, in order to make a very important part of ASM easier to understand.