TI-83+ Z80 ASM for the Absolute Beginner

LESSON EIGHT:

- *Math Applied to Constants*
- *The Special Purpose of HL*
- *Displaying Text*

MATH APPLIED TO CONSTANTS

If you remember, .db is used to enter raw data into the calculator, data which you access by the ram addresses in which the data is contained.

Let’s say, hypothetically, that you want to program a four-color grayscale game with black, white, light grey and dark grey. This game—we can pretend, can’t we—will be a four-player, split-screen multiplayer game, that allows each player to build a car. There are five values of data in particular: Color of car, color of wheels, type of car, size of car, and speed of car. (Don’t compile this)

```
#include “ti83plus.inc”
.org  40339
.db    t2ByteTok, tAsmCmp

Truck .equ 0              ; A whole bunch of constants
Car .equ 1
Motorcycle .equ 2
Slow_Speed .equ 0
Medium_Speed .equ 1
Fast_Speed .equ 2
Small_Size .equ 0
Medium_Size .equ 1
Big_Size .equ 2
White .equ 0
Light_Grey .equ 1
Dark_Grey .equ 2
Black .equ 3


          ; Continued on next page
```

```
Player1:
        .db White, Light_Grey, Truck, Small_Size, Slow_Speed
Player2:
        .db Black, Dark_Grey, Car, Medium_Size, Fast_Speed
Player3:
        .db Light_Grey, Dark_Grey, Motorcycle, Large_Size, Slow_Speed
Player4:
        .db Black, White, Motorcycle, Small_Size, Medium_Speed
```

So then, you remember how to access data in a variable, right? For example, ld a, (Player1). However, register A will only contain the color White, meaning the number zero! Want to know why?

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include "ti83plus.inc"		
.org 40339		
ld a, (Player1)	40339	58, 40342
Player1:		
.db White,	40342	0,
Light_Grey,	40343	1,
Truck, Small_Size, Slow_Speed	40344	0, 0, 0

As you can see, register A accesses only what is inside RAM address 40342, and nothing after it! That is, Player1 pertains to RAM address 40342, which holds the first value of data, and that is all that is stored inside register A. Register A does not retrieve values for any addresses before and after 40342, in this case. So what if you want to get the other four values and store them in registers B, C, D and E? These four values are stored in addresses 40343, 40344, 30345 and 30346, so how do you get them? You could create a label for each value and therefore turn each value into a variable, but that can get quite tedious and messy.

However, if you think of the label as just a constant, you can do some math with the constant. In this case, Player1 is equal to 40342, which means that SPASM puts in the number 40342 whenever it comes across Player1. So what if you do a little math, and add 1 to the constant? For instance, Player1 + 1? Then SPASM will put in 40343, which is 40342 + 1!

So, you can access the first value in Player1 by ld a, (Player1), the second value using ld a, (Player1 + 1), the third value by using ld a, (Player1 + 2), etc.

You can also apply a minus sign to a constant. And even multiplication and division! (Be careful with division. The calculator is very picky when it comes to decimal numbers, and you will usually only get integers.) For instance, Player1 – 5 and Player1 * 15, and Player1 / 15.

So why can you do this and not use multiplication, addition or subtraction directly on registers? Why can't you say "ld a, e + 5?" This is because the processor, directly, can't handle this. Whenever there is a value that has to change, the processor can only do so much to change it.

But the value for constants is set, and remember that constants are handled by SPASM, not by the calculator. When you apply addition, subtraction, and multiplication to a constant before compiling it, the value is fixed, it will never change…a constant multiplied by a constant is still a constant. In other words, if you tell your program to ld a, (Player1 + 7), the program doesn't translate into "find out what Player1 + 7 is equal to, and then put into register A whatever is inside of Player1 + 7." Instead, the program translates into "put into register A whatever is inside of RAM at address 40350."

For this reason, you can use addition, subtraction, multiplication and limited division on constants. For instance, ld a, 3+ 3 will translate into ld a, 6. Similarily, ld a, 4 * 23 wil translate into ld a , 92. Finally, ld a, NumberSeven + NumberFour is the same as ld a, 11.

THE SPECIAL PURPOSE OF HL

Now that you understand how you can access subsequent areas of ram, let's look at an example of how to do this. We will use our previous data, and store the color of the car in register b, the color of the wheels in register c, the type of car in register d, and the size of the car in register e. Register A will contain the speed of the car. The values we load will depend on the player.

```
; Register A will contain the player, whether 1, 2, 3 or 4.  We use the value
; in register A to decide what data to load.

        cp 1
        jr z, Load_Player1_Data
        cp 2
        jr z, Load_Player2_Data
        cp 3
        jr z, Load_Player3_Data
        cp 4
        jr z, Load_Player4_Data

Load_Player1_Data:
        ld a, (Player1)
        ld b, a
        ld a, (Player1+1)
        ld c, a
        ld a, (Player1+2)
        ld d, a
        ld a, (Player1+3)
        ld e, a
        ld a, (Player1+4)
        ret
```

```
Load_Player2_Data:
        ld a, (Player2)
        ld b, a
        ld a, (Player2+1)
        ld c, a
        ld a, (Player2+2)
        ld d, a
        ld a, (Player2+3)
        ld e, a
        ld a, (Player2+4)
        ret
Load_Player3_Data:
        ld a, (Player3)
        ld b, a
        ld a, (Player3+1)
        ld c, a
        ld a, (Player3+2)
        ld d, a
        ld a, (Player3+3)
        ld e, a
        ld a, (Player3+4)
        ret
```

```
Load_Player4_Data:
        ld a, (Player4)
        ld b, a
        ld a, (Player4+1)
        ld c, a
        ld a, (Player4+2)
        ld d, a
        ld a, (Player4+3)
        ld e, a
        ld a, (Player4+4)
        ret

Player1:
        .db White, Light_Grey, Truck, Small_Size, Slow_Speed
Player2:
        .db Black, Dark_Grey, Car, Medium_Size, Fast_Speed
Player3:
        .db Light_Grey, Dark_Grey, Motorcycle, Large_Size, Slow_Speed
Player4:
        .db Black, White, Motorcycle, Small_Size, Medium_Speed
```

Okay, so this works, but you notice how much space this takes? You also notice that if you were to write a similar program in Ti-Basic, you wouldn't have to make a 4x copy of your simple code. The problem is, you have to pick a different constant, a different ram address, each time. This is because of the function ld a, (Value), which calls for a strict value, a strict ram address. What we need is to load register A with an address that can change.

I WAS THINKING, WE'VE BEEN GOING OUT FOR A WHILE NOW...
AND DON'T GET ME WRONG, I LOVE YOU JUST THE WAY YOU ARE...
JUST THINK OF THEM AS LI'L IMPROVEMENTS, TWEAK OF HAIR HERE, CLOTHES...
AND VOILA! YOU'RE PERFECT!
SIGH...*

In a perfect world, we could load a ram address into a register depending on what player we want to load. In a perfect world, we could use this value to access the car color for the player, and then increase the value in the register to access the next value of data (in RAM) for the car, aka the color of the wheels. On the next page is our perfect world program, using the register H to hold the ram address.

```
; Register A will contain the player, whether 1, 2, 3 or 4.  We use the value
; in register A to decide what data to load.

        cp 1
        jr z, Load_Player1_Data
        cp 2
        jr z, Load_Player2_Data
        cp 3
        jr z, Load_Player3_Data
        cp 4
        jr z, Load_Player4_Data

Load_Player1_Data:
        ld  h, Player1
        call  Load_Player_Data
        ret
Load_Player2_Data:
        ld  h, Player2
        call  Load_Player_Data
        ret
Load_Player3_Data:
        ld  h, Player3
        call  Load_Player_Data
        ret
Load_Player4_Data:
        ld  h, Player4
        call  Load_Player_Data
        ret

; Continued on next page
```

```
Load_Player_Data:
        ld a, (h)  ; In a perfect world, h contains the ram address to load from
        ld b, a
        inc h
        ld a, (h)
        ld c, a
        inc h
        ld a, (h)
        ld d, a
        inc h
        ld a, (h)
        ld e, a
        inc h
        ld a, (h)
        ret

Player1:
        .db White, Light_Grey, Truck, Small_Size, Slow_Speed
Player2:
        .db Black, Dark_Grey, Car, Medium_Size, Fast_Speed
Player3:
        .db Light_Grey, Dark_Grey, Motorcycle, Large_Size, Slow_Speed
Player4:
        .db Black, White, Motorcycle, Small_Size, Medium_Speed
```

But let's face the facts: it's not a perfect world. Remember that H cannot be bigger than 255? Most RAM addresses are much bigger than 255!

Oh, but this was so much smaller (and would be faster) than using constants! Oh, if only we could do something like this in a Z80 ASM program.

Oh, but we can! It's just we can't use a one-byte register, as it only goes up to 255. H and L are both one-byte registers. But, what happens when you put one byte and one byte together? You get TWO bytes! Simple math! ☺ Oh, and 2 bytes can go up to 65535, which, as you can tell, is high enough to store RAM addresses!

So you use HL (H and L put together) to store RAM addresses. Then, as you might expect, you use (HL) to access whatever is inside of the RAM pointed to by HL. Use ld hl, 2-byte value to store a value into hl, such as LD HL, 12482.

By the way, if you use a fixed number/constant to store a RAM address and retrieve whatever is inside of it, you can only use register A to do so. BUT when you use HL, you can immediately store it to any one-byte register! **Whenever you see a function, such as ld, that has a parameter of a one-byte register, you can also use (HL) inside that parameter.**

<u>For example, instead of</u>

ld a, (Player1)

ld e, a

<u>You can use</u>

ld hl, Player1

ld e, (hl).

This will give us VAST improvements to our data-accessing program. You can use INC and DEC functions on HL by the way.

```
; Register A will contain the player, whether 1, 2, 3 or 4.  We use the value
; in register A to decide what data to load.

        cp 1
        call z, Load_Player1_Data
        cp 2
        call z, Load_Player2_Data
        cp 3
        call z, Load_Player3_Data
        cp 4
        call z, Load_Player4_Data
        call  Load_Data_Into_Registers

Load_Player1_Data:
        ld  hl, Player1
        ret
Load_Player2_Data:
        ld  hl, Player2
        ret
Load_Player3_Data:
        ld  hl, Player3
        ret
Load_Player4_Data:
        ld  hl, Player4
        ret

; Continued on next page
```

```
Load_Player_Data:
        ld b, (hl)
        inc hl
        ld c, (hl)
        inc hl
        ld d, (hl)
        inc hl
        ld e, (hl)
        inc hl
        ld a, (hl)
        ret

Player1:
        .db White, Light_Grey, Truck, Small_Size, Slow_Speed
Player2:
        .db Black, Dark_Grey, Car, Medium_Size, Fast_Speed
Player3:
        .db Light_Grey, Dark_Grey, Motorcycle, Large_Size, Slow_Speed
Player4:
        .db Black, White, Motorcycle, Small_Size, Medium_Speed
```

You can also use (hl) to store values into variables.

<u>For example, instead of</u>

ld a,1

ld (Player1), a

<u>You can use</u>

ld hl, Player 1

ld (hl), 1

DISPLAYING TEXT

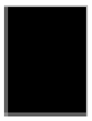
Now that you understand on how HL is used to store **changing, non-constant** RAM addresses for data access, we can start playing with functions that require HL as a parameter. After reading the title, you might wonder, “why do we need HL to display text?” Because text is stored as pure, unaltered data with .db, meaning ***you need a label!*** And since labels pertain to RAM addresses, HL is used to access this label.

Since .db is used to enter raw data, you can use it to store strings. However, there is a catch, nothing that you need to be concerned about, and actually something that gives you an advantage.

Remember that a translated ASM program, in the end, consists of a whole bunch of numbers. When your calculator stores a string, it’s stored as a bunch of numbers. When your calculator needs to display the string, these numbers are, when the program is running, translated into their respective characters so you see them on the screen as letters.

For example, the letter “A” is the number 65, and the letter “C” is the number 67. So if you need your calculator to display “ACA,” your program should contain the numbers 65 67 65, in other words .db 65, 67, 65. When ordered to display this data as a string, your calculator will read these numbers and translate them onto the screen as ACA.

The advantage is that you can display characters you normally couldn’t display in Ti-Basic! For instance, what if you wanted to display the “blinking block” you see on the text screen all the time?



The number for this character is 224. So by putting .db 224 in your program and telling your calculator to display it as a string, the calculator will see the number 224 and display this block character.

However, let's face it, .db Number, Number, Number for a bunch of letters can be tedious. Looking up the numerical translations for a bunch of characters is very time-consuming. Thus, you can enter strings directly as well with .db.

For example,

```
Text:
        .db "Hello World", "This is my string", 224, 65, 67, 67, 65, etc.
```

Of course, you can combine string data and numbers, such as with the above example.

It's just important to understand that the string is stored as numbers, just in case you want to use letters and symbols that your computer doesn't supply, such as the arrows you see in the calculator for scrolling menus.

The built-in function PutS will take the numerical data, and translate them into letters that are placed on the screen as text you can see.

B_CALL _PutS

Displays a string specified by the label stored in HL. The string must end in a zero, or an error occurs. The zero tells the calculator where the string ends.

Examples:

```
        LD HL, String
        B_CALL  _PutS
String:
        .db "Hello World", 0
```

As a reminder, I don’t recommend you copy this program directly, in case “copy-and-paste” bugs occur. You should retype it yourself.

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

         B_CALL _ClrLCDFull

         ld hl, HelloWorldString
         B_CALL _PutS
         B_CALL _getKey
         B_CALL _ClrLCDFull
         ret

HelloWorldString:

         .db “Hello World”, 0
```

Wouldn’t it be if we could put the string anywhere we want?

If you read chapter three, you remember that the calculator has special areas of RAM reserved for its uses, each with its own addresses. For displaying text on the main screen, the calculator keeps track of the location of the cursor using curCol and curRow—variables if you will. curCol is a constant, and so is curRow. curRow is a constant for RAM address 33867. The RAM at 33867 is used for storing what row—the first row, row 0; the second row, row 1; the third row, row 2; etc.—the cursor is on the home screen. curCol is right after curRow in terms of location/address, and is used to store what column—0 to 15—the cursor is located. By manually specifying a row and a column, we can tell the calculator where we want our text. Add the lines in blue on the next page.

```
#include “ti83plus.inc”
.org  40339
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

        ld a, 3
        ld (curRow), a
        ld a, 2
        ld (curCol), a

        ld hl, HelloWorldString
        B_CALL _PutS
        B_CALL _getKey
        B_CALL _ClrLCDFull
        ret

HelloWorldString:

        .db “Hello World”, 0
```

Remember, the first row is row 0, and the first column is column 0.

Some exercises are on the next page. Next lesson we’ll learn about other two byte registers, by putting D and E together, and by putting B and C together.

Exercises: Write programs to display strings at the specified positions on the screen.

1. The string “Get a grip” at anywhere in particular
2. The string “This is fun!!” on the first row, the first column
3. The string “Ti-83+ Z80 ASM” in the second column and the 4th row
4. The string “A” in the 16th column, the 6th row
5. The string “Hello World” preceded by an up arrow and proceeded by a down arrow. The numerical value for an up arrow is 30, and the numerical value for a down arrow is 31. Display it in the 3rd Column and the 4th row.

ANSWERS:

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

        ld hl, String
        B_CALL _PutS
        B_CALL _getKey
        B_CALL _ClrLCDFull
        ret

String:

        .db “Get a grip”, 0
```

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

        ld a, 0
        ld (curRow), a
        ld a, 0
        ld (curCol), a

        ld hl, String
        B_CALL _PutS
        B_CALL _getKey
        B_CALL _ClrLCDFull
        ret

String:

        .db “This is fun!!”, 0
```

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

          B_CALL _ClrLCDFull

          ld a, 3
          ld (curRow), a
          ld a, 1
          ld (curCol), a

          ld hl, String
          B_CALL _PutS
          B_CALL _getKey
          B_CALL _ClrLCDFull
          ret

String:

          .db “Ti-83+ Z80 ASM”, 0
```

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

          B_CALL _ClrLCDFull

          ld a, 5
          ld (curRow), a
          ld a, 15
          ld (curCol), a

          ld hl, String
          B_CALL _PutS
          B_CALL _getKey
          B_CALL _ClrLCDFull
          ret

String:

          .db “A”, 0
```

```
#include “ti83plus.inc”
.org  $9D93
.db    t2ByteTok, tAsmCmp

          B_CALL _ClrLCDFull

          ld a, 3
          ld (curRow), a
          ld a, 2
          ld (curCol), a

          ld hl, HelloWorldString
          B_CALL _PutS
          B_CALL _getKey
          B_CALL _ClrLCDFull
          ret

HelloWorldString:

          .db 30, “Hello World”, 31, 0
```