TI-83+ Z80 ASM for the Absolute Beginner

LESSON ELEVEN:

- *ASM Gorillas, Part III: The Menus*

ASM GORILLAS, PART III: THE MENUS

Now that you have learned how to display text, as well as how to work with some two-byte registers, let's work on the menu system for ASM Gorillas.

The approach we will take is to have one routine to display text for the menus. Depending on the data received, the menu routine will display particular text for a particular menu. By having a flexible routine of this kind, we can display as many menus as we want, where the only cost in storage space is the number of menus we have. In other words, we don't have to use storage space by putting menu code in for each menu we want to display.

We are not going to display menus on the main text screen. Rather, we are going to use the graph screen, meaning we will use small text. Believe it or not, while there is definitely a difference between "Big Text" and "Small Text", you can almost always transition from one set of functions to the other by remembering only two simple rules of thumb:

1. To use a "Big Text" function to display small text, always place a "V" before the respective function. For instance, you can transition from _PutS for big text to displaying small text via the B_CALL function "_VPutS."
2. Instead of curCol and curRow, use penCol and penRow as text coordinates. In this case, since the Ti-83+ graph screen consists of 96 columns and 64 rows, penCol can be a value from 0 to 95, and penRow can be a value from 0 to 63.

Before we continue, however, be aware that there are some changes we will need to make to our previous code as we progress. The truth is, as I was working on this program, I found that I had to add some lines of code, change some lines, and even remove some lines. But rather than make these changes in the 6th lesson, I decided to let you make the changes during the course of the programming session, as I had to do. It's just a good chance to practice.

To display menus, we're going to use a different approach than VPutS. VPutS, like PutS, calls for a string that ends in a zero so the calculator knows how big the string is. We're going to use a different function, called VPutSN.

```
B_CALL  _VPutSN
```

Displays a string specified by HL. The string does not end in a zero. Instead, the number of characters to display is stored in register B.

```
 Examples:          LD HL, String
                    LD B, 11
                    B_CALL  PutS


String:
        .db "Hello World"
```

The reason we're using VPutSN is for a flexible menu routine. Rather than having to calculate the menu length, we can have it specified manually.

Let's start by typing in the data we'll need for the menus, in term of text data. For the Main Menu, we will have four options: New Game, Load Game, Options and Quit. The Settings Menu will have options for Gravity, Wind Speed, # of Rounds (this is new), and Back. The Players Menu, formally called the Names Menu, will have options for Player One, Player Two, and Back. The AI/Human Menu, formally called the Players Menu, will have options for Player vs. AI, Two Players, or Back. Finally, we will have a new menu, AI Difficulty: Easy, Medium, Hard, Crazy, and Back.

In front of each menu option, we'll have a number, pertaining to the number of characters the line of text for the menu option has. Go to ASMGorillasMain.asm and type in #include "ASMGorillasData.asm" beneath your other include lines. In addition, create a new file, ASMGorillasData.asm. In this new file, type in the data on the next page.

```
Main_Menu_Text:

.db 8, "New Game"
.db 9, "Load Game"
.db 7, "Options"
.db 4, "Quit"

Settings_Menu_Text:

.db 7, "Gravity"
.db 10, "Wind Speed"
.db 11, "# of Rounds"
.db 4, "Back"

Players_Menu_Text:

.db 10, "Player One"
.db 10, "Player Two"
.db 4, "Back"

AI_Or_Human_Menu_Text:

.db 13, "Player vs. AI"
.db 11, "Two Players"
.db 4, "Back"

AI_Difficulty_Menu_Text:

.db 7, "Easy AI"
.db 9, "Medium AI"
.db 7, "Hard AI"
.db 8,"Crazy AI"
.db 4, "Back"
```

You will notice that this is the text for the five menus we will have. However, if you look at the data in ASMGorillasConstants.asm, we have some inconsistencies. We will need to change these to make this program work. The final file will be on the next page so you can make sure you did everything right, but make whatever changes you can understand so that you can see for yourself what's going on.

First of all, we have to change some menu names. Change Players_Menu to AI_Or_Human_Menu, change Names_Menu to Players_Menu, and add AI_Difficulty_Menu .equ 4. For consistency of order, let Players_Menu equal 3, and let AI_Or_Human_Menu equal 4. Also, change these names in the respective "Items" constants…for instance, change Names_Menu_Items to Players_Menu_Items. Finally, add the following line: "AI_Difficulty_Menu_Items .equ 5"

Something else I hoped you noticed. For each of the constants pertaining to the number of items (with the word Items at the end of its name), the number after .equ is supposed to be the number of items the menu has. Some of these are incorrect. For instance, you will notice that the Settings Menu has 4 items now, instead of 3. Change that. Now, let Player_Menu_Items be equal to 3 (don't use a constant for this like we did in the previous text file), and let AI_Or_Human_Menu be equal to 3.

Finally, let's change where our menu is displayed. After running the program, I found that when I displayed the text at X = 11, the text was too far to the right. So change MainMenuItem1X to equal 8.

```
MainMenuItem1X .equ 8

MainMenuItem1Y .equ 14


Main_Menu .equ 0

Settings_Menu .equ1

AI_Or_Human_Menu .equ 2

Players_Menu .equ 3

AI_Difficulty_Menu .equ 4


Main_Menu_Items .equ 4

Settings_Menu_Items .equ 4

Players_Menu_Items .equ 3

AI_Or_Human_Menu_Items .equ 3

AI_Difficulty_Menu_Items .equ 5


Player1NameX .equ 0

Player1NameY .equ 0

Player2NameX .equ 51

Player2NameY .equ 0

Player1ScoreX .equ 0

Player1ScoreY .equ 58

Player2ScoreX .equ 83

Player2ScoreY .equ 58




LoadingTextX .equ 3

LoadingTextY .equ 3
```

Let's work on the routine to display the menu. I'll give you the code line by line, and explain it on the way. Create a new text file called "ASMGorillasStartProgram.asm", and include this in "ASMGorillasMain.asm."

```
;hl is the location of the menu text
;b is the number of items the menu has
;d contains the X position, the column to
display the text at.
;e contains the Y position, the row to display
the text at.
```

These are four comments that describe what happens in the subroutine. It is very important to include comments in your program so that you, and the reader of your code, can understand (and more importantly, REMEMBER) what in the world is happening in your code.

These comments tell you, and remind you, what registers are needed to display a menu. Remember how you used hl to use the line B_CALL _PutS? This is similar; to use the menu routine, you need registers hl, b, and de.

Remember that d and e can be put together. If you put d and e together, de contains both the X position and the Y position. This is not important for this menu routine, but it is important to remember.

```
Display_Text_Menu:
```

The reason that we have this label here is that we loop several times in this menu routine to display text. Every time we display one line of text, we need to loop back here. The label tells us where the beginning of the loop is.

```
ld a, e
ld (penRow), a
```

Since register e contains the Y position, the Row to display our text out, we need to store it in penRow. Just a reminder that this is one of the ways to store values in RAM.

```
ld a, d
ld (penCol), a
```

Stores d, which contains the X position, into penCol.

```
ld c, b     ;Save the number of menu items
            ;left to display
```

As the comment states, we need to save the number of menu items left to display. We need register B to use B_CALL _VPutSN, so we don't want to lose our value in register B that we had stored there before.

```
ld b, (hl)
```

From our parameters in the first four comments of our program, we understand that hl contains the location of our menu text. If you look back at our "ASMGorillasData.asm" text file, you will see that the first value is a number, the number of characters in the line of text you want to display. Since register B needs to hold this value for _VPutSN, we store this value.

As an example, suppose that hl was equal to Main_Menu_Text. Then register B will contain the value 8, and as you can see, "New Game" contains 8 characters.

```
inc hl
```

Our first value in hl, as previously stated, pertained to the number of characters in the line of text. Now we want hl to point to the line of text. By increasing hl by one, hl now equals the line of text. In our previous example, for instance, (hl) was at first equal to 8. After inc hl, (hl) is equal to "N". Thus hl now points where our text starts. (In other words, HL is now equal to Main_Menu_Text + 1, where our string starts) Recall that B_CALL _VPutSN requires hl to be the location of the string.

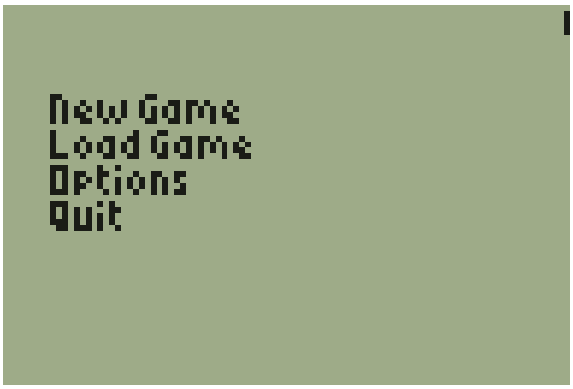
```
B_CALL _VPutSN
```

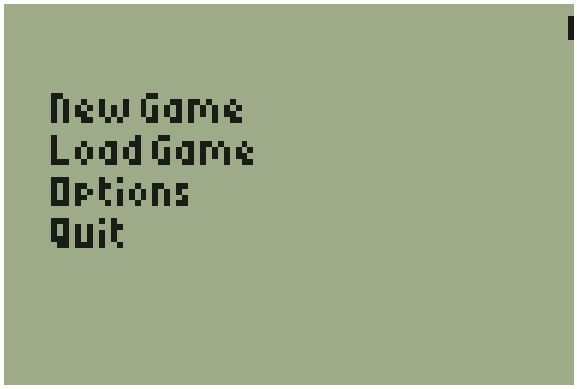
Displays our string in small text, at the X and Y position we specified. By the way, after we run the routine _VPutSN, hl will point to the number of characters that the next line of text contains.

```
ld a, e   ; Move the line of text to the next
          ;row
add a, 7
ld e, a
```

We now need to store a new value for Y so that we can display the next line of text correctly. If we don't specify a new row, we'll end up displaying text on the same row we did before, and we'll have a collision. This means that our menu will display incorrectly. Here's a screenshot with an example of what happens when all the text is accidentally displayed at the same Y coordinate:

As a good rule of thumb, any line of small text with capital letters has a maximum height, in pixels, of six. Therefore, to avoid squeezing lines of text on top of each other as in the previous screenshot, you need at least six pixels for each line of text. In the case of our menu, we also want one pixel of blank space to provide additional room so that people can read the menu easily. Compare the two screen shots. The first is when we advance the row six pixels, and the second is when we advance the row seven pixels.





Afterwords, register E contains our new Y position.

```
ld b,c
djnz Display_Text_Menu
ret
```

Now that we no longer need register B to hold the number of characters in our string, we put back the number of loops left, the number of lines left to display. Since this value had temporarily been stored in register C, we put it back in register B.

After that, we decrease B by one. If it equals zero, there are no more lines of text left to display. In that case, we exit the subprogram.

Now, add the following code to your program. This code should come before "Display_Text_Menu." Comments will explain everything. Don't run the program yet.

```
B_CALL _ClrLCDFull

        ld d, MainMenuItem1X        ; Stores the X position for the menu
        ld e, MainMenuItem1Y        ; Stores the starting Y position for the menu


        ld a, Main_Menu         ; For right now, use register A to select a menu to test.
                                ; By letting register A be the value of a different menu,
                                ; you can test them.  For example, try ld a, Settings_Menu

        cp Main_Menu
        jr z, Display_Main_Menu
        cp Settings_Menu
        jr z, Display_Settings_Menu
        cp AI_Or_Human_Menu
        jr z, Display_AI_Or_Human_Menu
        cp Players_Menu
        jr z, Display_Players_Menu
        cp AI_Difficulty_Menu
        jr z, Display_AI_Difficulty_Menu

Display_Main_Menu:

        ld hl, Main_Menu_Text       ; The values we need to run the routine
                                    ; Display_Text_Menu
        ld b, Main_Menu_Items

        jr Continue_To_Display_Menu   ; Continued on Next Page
```

```
Display_Settings_Menu:

        ld hl, Settings_Menu_Text
        ld b, Settings_Menu_Items

        jr Continue_To_Display_Menu

Display_AI_Or_Human_Menu:

        ld hl, AI_Or_Human_Menu_Text
        ld b, AI_Or_Human_Menu_Items

        jr Continue_To_Display_Menu

Display_Players_Menu:

        ld hl, Players_Menu_Text
        ld b, Players_Menu_Items

        jr Continue_To_Display_Menu

Display_AI_Difficulty_Menu:

        ld hl, AI_Difficulty_Menu_Text
        ld b, AI_Difficulty_Menu_Items

        jr Continue_To_Display_Menu

        ; Code continued on next page
```

```
Continue_To_Display_Menu:


        call Display_Text_Menu


        B_CALL _getKey

        B_CALL _ClrLCDFull

        ret
```

DON'T run the program yet. There's an important thing you need to know about #include files. SPASM will compile your include files in the order you specify them. So if you have the following:

#include "ASMGorillasData.asm"

#include "ASMGorillasStartProgram.asm"

Your program will compile the data first, and then Start Program, **IN THAT ORDER**. That means the program will run **IN THAT ORDER.** So when you run the program, it will start by trying to run data. Does that sound ridiculous? Running data? It is ridiculous. A program can't run "data." So then your program tries to run ridiculous nonsense, and the calculator will crash.

Therefore, it is important that you compile included files so that the file you want run first will be compiled first. However, ti83plus.inc MUST be the first include file.

Open "ASMGorillasMain.asm" and make sure that the first few lines are equal to the following on the next page.

```
.org 40339

.db t2ByteTok, tAsmCmp


#include "ti83plus.inc"

#include "ASMGorillasStartProgram.asm"

#include "ASMGorillasConstants.asm"

#include "ASMGorillasData.asm"
```

Now you can run your program safely. Be sure to try different values for register A to try out the different menus.

Notice after using the program that the menus can only display. You can't select anything from them! Annoying? Sure! But we'll start taking care of that next lesson.