TI-83+ Z80 ASM for the Absolute Beginner

APPENDIX C:

- *Working Directly With the Keypad*

WORKING DIRECTLY WITH THE KEYPAD

This is it! The third method for detecting key presses, the method that teased your brain from lesson _____. You desperately wanted to know how to detect multiple keys at once, as well as how to get an action to continue as long as the key is held down. And now you can read about how to do it!

But be aware, you don't use any B_CALLs for this process. You will instead be working directly with the calculator's hardware. Hard? No. But it does require some explanation about **ports**.

The Z80 processor (which your Ti-83+ uses) is designed to work with all sorts of hardware. Your Ti-83+ is not the only device to use the processor. Other calculators use it, Nintendo Game Boys use it, and even some old computers use it!

And as you can imagine, the Z80 processor can't possibly have instructions to work with every single device it's used for. It does not have instructions specifically to print a picture from a printer, or to turn a computer off, or to detect whether a mouse is moving right or left. But, the processor IS able to **send and receive data** to and from such hardware.

Think of it as using the link port to connect two calculators together with a cable. You cannot use your calculator to turn off the other player's calculator, nor can you draw a picture on the other person's screen using your own calculator. But you can send data to your friend's calculator, and receive data from his calculator.

Each hardware device has some kind of **port** that the Z80 processor accesses to send data to the device and receive data from the device. Two things can happen with the port.

One, the Z80 processor sends data through the port to the hardware, and the hardware uses that data to decide what to do. For example, when you draw a picture on the screen and use B_CALL _GrBufCpy, the processor sends picture data through the screen's port to the screen itself. The screen, as hardware, will read that picture data, and then say to itself "Oh, here's some picture data I need to display. Since the processor doesn't know how to do it, I will do it myself."

The other thing that happens is you can tell the processor to READ data from a particular port. Then you can tell the calculator what to do depending on what data the processor reads. In this case, we are going to read data from the keyboard port, and the data will tell us which keys are being pressed.

Every port has a number, and the keyboard port on the Ti-83+ is port 1. So to read data from the Ti-83+ keyboard, we read from port 1. To send data, we send it through port 1.

There are several instructions for sending and receiving data via ports, but we will only concern ourselves with 2 of them: IN and OUT. As you might expect, IN reads data through a port, and OUT sends data through a port to a hardware device.

IN A, (**One-Byte Value**)

One-Byte Value is the port number you want to read data from. This instruction will read a byte of data from the port number, and place it inside of register A.

Examples: IN A, (1)

T-States: 11 Byte Storage: 2

IN **One-Byte Register**, (C)

Register C must contain the port number you want to read data from. This instruction will read a byte of data from the port number, and place it inside of a One-Byte Register. In this case, One-Byte Register CANNOT be (HL).

Examples: LD C, 1

IN E, (C)

T-States: 12 Byte Storage: 2

OUT (**One-Byte Value**), A

One-Byte Value is the port number you want to send data through. This instruction will send a byte of data from register A through the port.

Examples: OUT (1), A

T-States: 11 Byte Storage: 2

OUT (C), **One-Byte Register**

Register C must contain the port number you want to send data through. This instruction will send a byte of data from the register of your choice through the port pointed to by Register C. In this case, One-Byte Register CANNOT be (HL).

Examples: LD C, 1

OUT (C), H

T-States: 12 Byte Storage: 2

So now that you understand that the Z80 uses port 1 to send and read data concerning the keyboard, I'll teach you how to work with this. Remember, when the Z80 sends data through the port, the keyboard hardware of the Ti-83+ takes this and decides what to do based on the data it received. When the Z80 **receives** data from the keyboard hardware, you can tell the Z80 what to do depending on the data it reads.

On the Ti-83+, the keys are divided into several different groups. When we want to see if certain keys have been pressed, we tell the Ti-83+ to check the group (or groups) to see if a key is being held down. Here are the different key groups:

Group1 = $0fe
KDown
KLeft
KRight
KUp

Group2 = $0fd
KEnter
KPlus
KMinus
KMul
KDiv
KPower
KClear

Group3 = $0fb
kMinus2
kThree
kSix
kNine
kRbracket
kTan
kVars

Group4 = $0f7
KPoint
KTwo
KFive
KEight
KLbracket
KCos
KPrgm
KStat

Group5 = $0ef
KZero
KOne
KFour
KSeven
KComma
KSin
KApps
KGraphvar

Group6 = $0df
KSto
KLn
KLog
kX^2
kX^{-1}
kMath
kAlpha

Group7 .= $0bf
KGraph
KTrace
KZoom
KWindow
KY=
k2nd
kMode
kDel

So to tell the Ti-83+ to check a certain group for keys being pressed, store into A the hexadecimal value of that group. (Remember that the dollar signs signify a hexadecimal number.) Then send it via OUT. For example, suppose we want the calculator to see which of the arrow keys have been pressed.

```
LD A, $FE       ; The group of keys pertaining to the arrow keys
OUT (1), A      ; Keyboard Port is Port #1
```

However, before you ask the Ti-83+ to check a key group, you MUST tell the calculator that it's time to check a key group. To do this, send the value $FF through the port. This needs to be done every time you want to check a key group.

```
LD A, $FF
OUT (1), A
```

After this, you need to give the port time to rest itself. NOP is a Z80 instruction that tells the calculator to do absolutely nothing for 4 T-States. You need two NOPs every time you tell the calculator to read a key group.

So on the next page, I've listed what we have so far in terms of reading arrow keys.

```
LD A, $FF
OUT (1), A
LD A, $FE
OUT (1), A
NOP
NOP
```

By now, the Ti-83+ has scanned the arrow-keys group to see which ones have been pressed. Every key uses 1 bit of data. The bit for a key will equal 0 if the key is being pressed, and it will equal 1 if the key is not being pressed. By using the instruction IN A, (1), register A will hold the value of all the bits put together: 1 bit for each key.

Now here's what's important: As you know, there's 8 bits in every byte. If you take a byte and convert it to binary (which gives you 8 bits), the 1 or 0 on the far right will be **Bit 0**. The 1 or 0 on the far left will be **Bit 7**.

Bit #	7,	6,	5,	4,	3,	2,	1,	0
%	1	1	0	1	1	1	0	0

On the table on page 6, in every key group, the keys are listed in order. The first key in the list for every group pertains to BIT 0. The second key in the list for every group pertains to BIT 1.

So let's say you used IN A, (1). Register A will tell you which keys have been pressed—in this case, the keys from Group 1, the arrow

keys. If only the down key is being pressed, A will equal %11111110. If all four arrow keys are being pressed, A will equal %11110000. If the left and up keys are being pressed, A will equal %11110101. If NO arrow keys are being pressed, A will equal %11111111.

The instruction BIT _____, A will set the Z flag if the bit equals 0, and reset the Z flag if the bit equals 1. (In the blank, place a number from 0 to 7, where 7 is bit 7 and 0 is bit 0.)

```
        ld  a, $FF
        out (1), a
        ld a, $FE
        out (1), a
        nop
        nop

        in a, (1)

        BIT 0, A
        CALL z, Key_Down_Has_Been_Pressed

; The reason CALL is used is, in this case, we want to see if both down and left have been
; pressed.  By CALLing Key_Down_Has_Been_Pressed, the code will eventually return
; here, and we can test for the left key.


        BIT 1, A
        jr z, Key_Left_Has_Been_Pressed

        BIT 2, A
        jr nz, Key_Right_Has_NOT_Been_Pressed
```

Now, remember, you need the top seven lines of the red box for every key group you want to read. Group $FE only applies to the arrow keys, so you need to use different values for different groups. Refer to page 6.

And now, you can check multiple key presses, as well as have something happen as long as the key is held down (no pauses)! The next example program, courtesy of Sean McLaughlin, will demonstrate this. The program test for two or more arrow keys being pressed, and the text will always, always move when you hold a key down, until the text reaches the edge of the screen. This example makes use of using Register C for ports, and using BIT for register B instead of register A. The program takes three pages of this lesson, by the way.

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

   B_CALL _RunIndicOff
   LD     HL, $1C23
   LD     (x_pos), HL

DispText:
   B_CALL _ClrLCDFull
   LD     HL, (x_pos)
   LD     (penCol), HL
   LD     HL, string
   B_CALL _VPutS
   LD     C, 1      ; Port 1 is the keyboard port

InKey:

   LD A, $FF
   OUT (C), A
   LD     A, $BF
   OUT    (C), A
   NOP
   NOP
   IN     A, (C)

   BIT    7, A
   JR     NZ, Check_Arrow_Keys
   LD     A, $FF          ; Reset key port
   OUT    (C), A
   RET
```

```
Check_Arrow_Keys:

   LD     A, $FF          ; Reset key port
  OUT     (C), A

   LD     A, $0FE
   OUT    (C), A
   NOP
   NOP
   IN     B, (C)

   BIT    0, B
   JP     Z, Down
   BIT    1, B
   JP     Z, Left
   BIT    2, B
   JP     Z, Right
   BIT    3, B
   JP     Z, Up

   JP     InKey

Down:
   CALL   MoveDown
   BIT    1, B
   CALL   Z, MoveLeft
   BIT    2, B
   CALL   Z, MoveRight
   JP    DispText

Left:
;There is no need to check for Down key anymore.
   CALL   MoveLeft
   BIT    3, B
   CALL   Z, MoveUp
   JP     DispText

Right:
   CALL   MoveRight
   BIT    3, B
   CALL   Z, MoveUp
   JP     DispText

Up:
   CALL   MoveUp
   JP     DispText

MoveDown:
   LD     A, (y_pos)       ; Check if at bottom edge of screen
   CP     57
   RET    Z
   INC    A                ; Down one pixel
   LD     (y_pos), A
   RET
```

```
MoveUp:
   LD     A, (y_pos)        ; Check if at top edge of screen
   OR     A
   RET    Z
   DEC    A                 ; Up one pixel
   LD     (y_pos), A
   RET

MoveLeft:
   LD     A, (x_pos)        ; Check if at left edge of screen
   OR     A
   RET    Z

   DEC    A                 ; Left one pixel
   LD     (x_pos), A
   RET

MoveRight:
   LD     A, (x_pos)        ; Check if at right edge of screen
   CP     96-28             ; 96 - number of pixels the string takes up
   RET    Z

   INC    A                 ; Right one pixel
   LD     (x_pos), A
   RET

x_pos:     .DB    0
y_pos:     .DB    0
string:    .DB    "Let\'s Go!", 0
```