

TI-83+ Z80 ASM for the Absolute Beginner

LESSON TEN:

- *Working with Key Presses*
- *Saving Register Values*

WORKING WITH KEY PRESSES



There was a time when computer programs could only run hard-coded numbers and instructions. You could not input values, and you could not change what a program could do with key presses. Nowadays,

writing a calculator program without user key presses will easily put it at the bottom of the list at ticalc.org.

There are three methods that will allow you to detect key presses in an ASM program. One method, which you have been using as `_getKey`, will wait for a keypress. The second method will only wait for a minuscule fraction of a second for a keypress, and then the program will keep running.

These two methods, however, are not flexible. First of all, it is difficult, if not impossible, to use these methods to allow continuous key detection. What happens when you press the down key—and hold it—in the catalog on the calculator? Assuming you are not running *Omnicalc* or a similar program, there is a brief pause before the menu keeps scrolling. This is definitely an annoyance if you are writing a game, such as a Mario game, where you need actions to happen constantly as the respective key is held.

Secondly, you can't detect multiply keys at once using these two methods. If you want to use the keys to move a ship and the second key to fire, you wouldn't be able fire and move at the same time.

The third method takes care of both these problems. Read Appendix C to learn about this. It requires an advanced knowledge of ASM to understand, so it will not be covered in the lesson itself. Rest assured, at the end of all the lessons, you will have the capability of understanding Appendix C.

Going back to `_getKey`, we know it waits for a key press. However, it will also tell you what key was pressed, by storing it into register A. Key presses, like most everything else, are numbers. During `_getKey`, if you press the Up key, represented as the number 3, register A will hold the number 3.

But like everything else, we can use constants to make values easier to remember. In this case, `kUp` is a constant for the number three.

Key codes for `_getKey` are represented as hexadecimal numbers. At the end of this lesson, you will find a table containing all the values for key presses, in hexadecimal. The dollar sign in front of the number means that the number is hexadecimal. When you are writing a program requiring hexadecimal numbers, you always put a \$ in front of a number that you want to be hexadecimal.

On the next page is a program that will demonstrate the use of `_getKey` to either increase register B or decrease register B. In other words, we won't simply use `_getKey` just to wait for a key press.

```

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

        B_CALL _ClrLCDFull
        ld b, 127
        jr  Display ; Display initial value of B.
KeyLoop:
        B_CALL _getKey
        cp  kUp    ; If the up arrow key was pressed.
        jr  Z, Increase
        cp  kDown  ; If the down arrow key was pressed.
        jr  Z, Decrease
        cp  kClear ; If the CLEAR key was pressed, quit the program
        ret  Z
        jr  KeyLoop ; If any other key was pressed, redo _GetKey.
Increase:
        ld a, b
        cp  255 ; Don't increment B if it's at its maximum value. That way it doesn't reset to 0
        jr  Z, KeyLoop
        inc b
        jr  Display ; Display new value of B.
Decrease:
        ld a, b
        cp  0    ; Don't decrement B if it's at its minimum value, so it doesn't reset to 255.
        jr  Z, KeyLoop
        dec b ; Since the next part of code run is display, we don't need to jr there.
Display:
        ld a, 0    ; Reset cursor to top of screen.
        ld (curRow), a
        ld a, 0
        ld (curCol), a
        ld h, 0
        ld l, b
        b_call _DispHL
        jr  KeyLoop ; Get another key.

```

The advantage to `_getKey` is that you can use it to detect when a key was pressed in conjunction with 2^{nd} or Alpha. For instance, you can tell the calculator what to do when Alpha-1 is pressed, when 2^{nd} -Math is pressed, etc. The only issue is that you cannot work with 2^{nd} -Up, Alpha-Up, Alpha-Down and 2^{nd} -Down with `_getKey`.

Replace `cp kUp` with `cp kCapA`, and replace `cp kDown` with `cp kCapB`. See what happens when you run the program. You can't change the value of register B with the regular up/down keys, but you can change the value either by pressing Alpha-Math or Alpha-APPS.

However, `_getKey` CANNOT be used to work with the 2^{nd} key by itself. You cannot use it to work with the Alpha key by itself either. The 2^{nd} and Alpha keys must be used in conjunction with other keys.

While `_getKey` waits for the user to press a key, `_GetCSC` does not. The user gets one shot at pressing a key, after which the program will continue. The advantage to `_GetCSC` is that a program can constantly be running without having to wait for a keypress. If the user does press a key, register A will contain the key pressed. If register A is equal to zero, no key was pressed.

There are some other differences between `_GetCSC` and `_getKey`. First of all, `_GetCSC` cannot directly access 2^{nd} -Key and Alpha-Key functions, though it does allow you to use 2^{nd} and Alpha keys by themselves. Also, `_GetCSC` uses different constants to represent key values. `kUp` and `kDown` will not work efficiently with `_GetCSC`.

The next page contains a program similar to the previous one. It uses `_getCSC`, meaning we have to make adjustments since `_GetCSC` waits only a fraction of a second for a keypress, and uses different constants. Pay special attention to the lines in blue.

```

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

        B_CALL _ClrLCDFull
        ld b, 127
        jr  Display ; Display initial value of B.
KeyLoop:
        B_CALL _GetCSC
        cp  skUp    ; If the up arrow key was pressed.
        jr  Z, Increase
        cp  skDown  ; If the down arrow key was pressed.
        jr  Z, Decrease
        cp  skClear ; If the CLEAR key was pressed, quit the program
        ret  Z
        jr  KeyLoop ; If any other key was pressed, or if no key was pressed, redo _GetCSC.
Increase:
        ld a, b
        cp  255 ; Don't increment B if it's at its maximum value. That way it doesn't reset to 0
        jr  Z, KeyLoop
        inc b
        jr  Display ; Display new value of B.
Decrease:
        ld a, b
        cp  0    ; Don't decrement B if it's at its minimum value, so it doesn't reset to 255.
        jr  Z, KeyLoop
        dec b ; Since the next part of code run is display, we don't need to jr there.
Display:
        ld a, 0    ; Reset cursor to top of screen.
        ld (curRow), a
        ld a, 0
        ld (curCol), a
        ld h, 0
        ld l, b
        b_call _DispHL
        jr  KeyLoop ; Get another key.

```

SAVING REGISTER VALUES

So far, we've learned about saving registers by means of saving the needed values to other registers.

© 1999 Randy Glasbergen. www.glasbergen.com



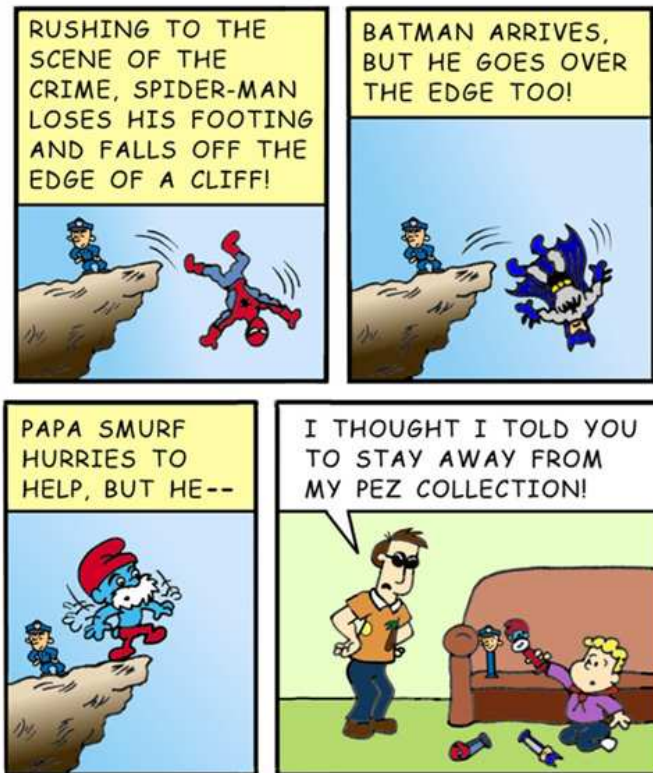
**"It's the latest innovation in office safety.
When your computer crashes, an air bag is activated
so you won't bang your head in frustration."**

However, as you might expect, this can be a problem if other registers you can save to are tied up with values you need. Furthermore, did you see any instructions that allow you to, for example, “LD HL, DE?” Nope, they don't exist. So while you can save the value of two byte registers by, say, “LD H, D LD L, E”, this can be an inconvenient, and in some cases, impossible. (It would have been awesome if Zilog could have made 150 registers to solve the problem of saving registers like this, but that would have been expensive at the time the processor was invented.)

What about saving the values of registers to RAM? Not a bad idea, but then register A would have to be free so you could store the value of the register in A and save it to RAM. This method is also not very optimized, and in a processor-intensive game, every second counts.

Did I just say register A would have to be free? Well, I'm mistaken, but nobody's perfect. You can save the values of registers to RAM in one simple instruction, PUSH. Then you recall it by POP.

Think of this process as a PEZ dispenser. I'm hoping you know what PEZ in. It's a very tasty, but tiny, square candy that will have little ones eating out of your hand in mere moments. What's more fun is the way you get that candy. You have a PEZ dispenser with someone's head at the top—Charlie Brown, Pluto the Dog, Chuck Norris, etc. To get candy out of the dispenser, you move the head back, and out comes the candy! The greatest thing since sliced bread.



New Strips Every Monday, Wednesday & Friday!



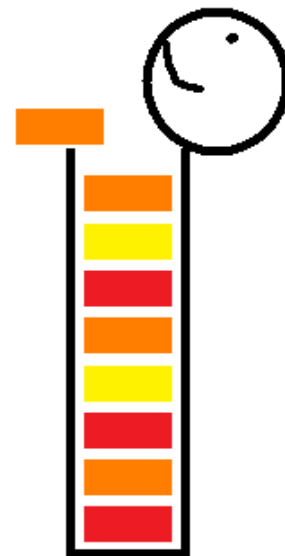
Now, does that candy come magically? Of course not, even though as a three-year-old you might have thought that. You have to put candy inside of the dispenser to fill it, and then you take candy out to eat it.

You refill the PEZ dispenser by pushing candy in. To stick a piece of candy in, it must be "pushed" on top of the other pieces of candy.

The reason you have to push the candy is because all the candy you stored in the dispenser stays toward the top. It won't go down unless you force it down, that is, unless you push it down.



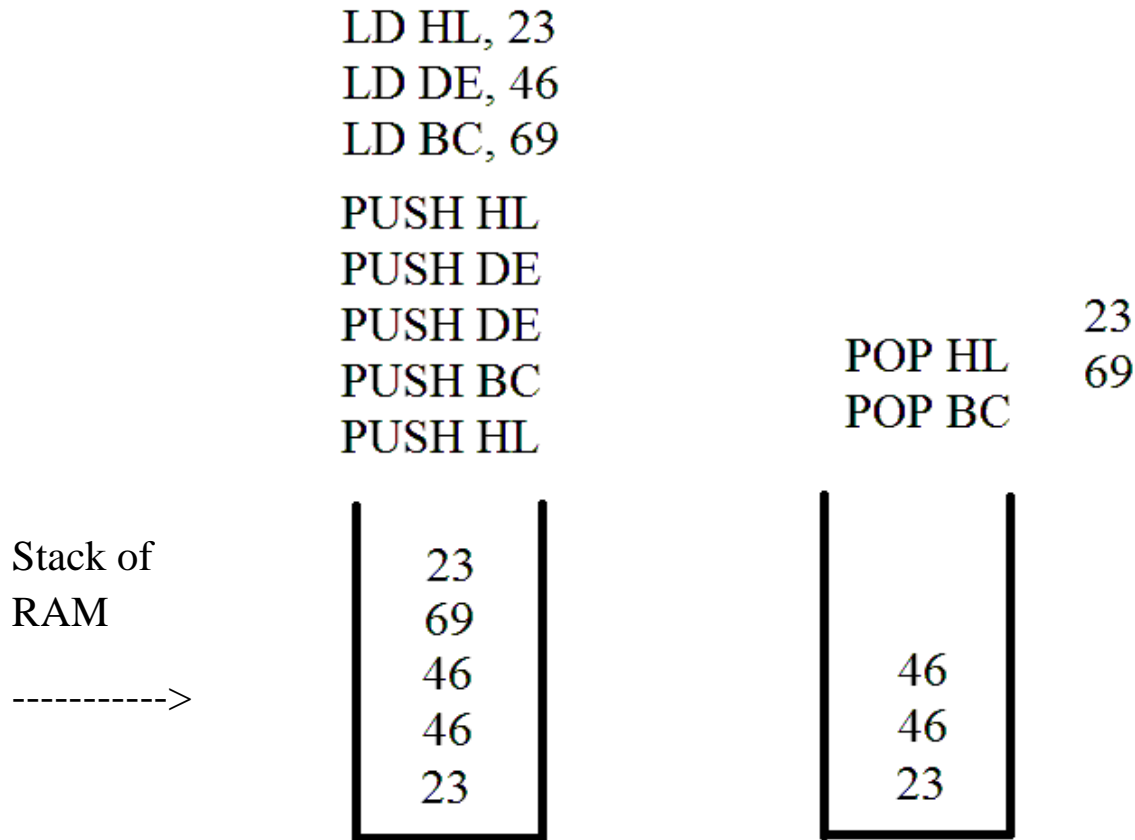
To get a piece of candy from your PEZ dispenser to eat it, you "pop" a piece of candy from the dispenser. By pushing the head of the dispenser back, a piece of candy "pops" out.



I think you can tell that when you put candy inside the dispenser, it always comes out in the opposite order you put it in. If you put in red, yellow, orange, orange, and yellow, you will eat yellow, orange, orange, yellow and red.

So when you use PUSH for saving the value of a register, you save it in RAM. When you POP it, the last value you pushed is POPed. It always goes in order, just like PEZ candy. If you push $HL = 10$, and then push $HL = 11$, the first value that is popped will be 11. By the way,

our “RAM Pez Dispenser” for pushing and popping values is called a **stack**. Think of it as a stack of candy, or a stack of values.



You can only push and pop 2 Byte registers. Therefore, you can push and pop HL, DE, BC and AF. You cannot push A, B, C, D, E, F, H, or L by themselves.

HOWEVER, if you push HL, you don't have to pop HL. If you push in a piece of PEZ candy that belongs to you, do you have to eat it when you pop out that piece of candy? NO! You can eat it, or you can give it to a friend or your little sister (now, let's not be selfish here). Let's say you need to save the value of HL = 10. So you push it. If DE needs that value later, you can use POP DE, and DE will equal 10. Be aware that once you pop a value, it disappears from the stack. A piece

of PEZ candy popped from the dispenser can only appear again if you put it back inside the dispenser.

Pushing and popping is a very effective method for saving the values of registers. But be sure that your stack of values is empty before you exit your program. (In other words, each and every “push” should have a “pop” somewhere in the program.) Otherwise, you will have an error, possibly a crash.

One more thing to be careful about, and I ask you to once again think of CALLs as Ti-Basic subprograms. If you have a PUSH in your main program, **YOU MUST POP IT** in the main program. If you pushed a value in a subprogram, you must pop it in that subprogram. Your calculator will have errors and probably crash if you pop a value outside of where you pushed it.

Key Codes (For `_getKey`)

These are the codes returned from the `GetKey` routine. They are grouped into four categories:

- [Primary-Function Keys](#) (press the key)
- [2nd-Function Keys](#) (press [2nd])
- [Alpha-Function Keys](#) (press [ALPHA])
- [Alpha-Alpha-Function Keys](#) (press [ALPHA] twice)

There are no codes for 2nd+Up or 2nd+Down, they always change the contrast.

Alpha-Alpha keys must be enabled with

```
SET      LwrCaseActive, (IY+AppLwrCaseFlag)
```

`kExtendEcho2 ($FC)` is always returned in A. (`KeyExtend`) holds the keycode.

Primary-Function Keys

Key	Equate	Value	Key	Equate	Value	Key	Equate	Value
[Y=]	kYEqu	\$49	[MODE]	kMode	\$45	[X,T,θ,n]	kVarX	\$B4
[WINDOW]	kWindow	\$48	[DEL]	kDel	\$0A	[STAT]	kStat	\$31
[ZOOM]	kZoom	\$2E	<	kLeft	\$02	V	kDown	\$04
[TRACE]	kTrace	\$5A	Λ	kUp	\$03			
[GRAPH]	kGraph	\$44	>	kRight	\$01			
[MATH]	kMath	\$32	[x ⁻¹]	kInv	\$86	[x ²]	kSquare	\$BD
[APPS]	kAppsMenu	\$2C	[SIN]	kSin	\$B7	[,]	kComma	\$8B
[PRGM]	kPrgm	\$2D	[COS]	kCos	\$B9	[(]	kLParen	\$85
[VARS]	kVars	\$35	[TAN]	kTan	\$BB	[)]	kRParen	\$86
[CLEAR]	kClear	\$09	[^]	kExpon	84	[÷]	kDiv	\$83

[LOG]	kLog	\$C1	[LN]	kLn	\$BF	[STO=>]	kStore	\$8A
[7]	k7	\$95	[4]	k4	\$92	[1]	k1	\$8F
[8]	k8	\$96	[5]	k5	\$93	[2]	k2	\$90
[9]	k9	\$97	[6]	k6	\$94	[3]	k3	\$91
[×]	kMul	\$82	[-]	kSub	\$81	[+]	kAdd	\$80
[0]	k0	\$8E						
[.]	kDecPnt	\$8D						
[(-)]	kChs	\$8C						
[ENTER]	kEnter	\$05						

Second-Function Keys

Key	Equate	Value	Key	Equate	Value	Key	Equate	Value
[STAT PLOT]	kStatEd	\$43	[QUIT]	kQuit	\$40	[LINK]	kLinkIO	\$41
[TBLSET]	kTblSet	\$4B	[INS]	kIns	\$0B	[LIST]	kList	\$3A
[FORMAT]	kFormat	\$57	[2nd] + <	kBOL	\$0E			
[CALC]	kCalc	\$3B	[2nd] + >	kEOL	\$0F			
[TABLE]	kTable	\$4A						
[TEST]	kTest	\$33	[MATRX]	kMatrix	\$37	[√]	kSqrt	\$BE
[ANGLE]	kAngle	\$39	[SIN ⁻¹]	kASin	\$B8	[EE]	kEE	\$98
[DRAW]	kDraw	\$2F	[COS ⁻¹]	kACos	\$BA	[{]	kLBrace	\$EC
[DISTR]	kDist	\$38	[TAN ⁻¹]	kATan	\$BC	[}]	kRBrace	\$ED
			[π]	kPi	\$B5	[e]	kCONSTeA	\$EF

[10 ^x]	kALog	\$C2	[e ^x]	kExp	\$C0	[RCL]	kRecall	\$0C
[u]	kUnA	\$F9	[L4]	kL4A	\$F6	[L1]	kL1A	\$F3
[v]	kVnA	\$FA	[L5]	kL5A	\$F7	[L2]	kL2A	\$F4
[w]	kWnA	\$FB	[L6]	kL6A	\$F8	[L3]	kL3A	\$F5
[[]]	kLBrack	\$87	[)]	kRBrack	\$88	[MEM]	kMem	\$36
[OFF]	kOff	\$3F						
[CATALOG]	kCatalog	\$3E						
[i]	kI	\$EE						
[ANS]	kAns	\$C5						
[ENTRY]	kLastEnt	\$0D						

Alpha-Function Keys

Key	Equate	Value	Key	Equate	Value	Key	Equate	Value
Page Up	kAlphaUp	\$07	[A]	kCapA	\$9A	[D]	kCapD	\$9D
Page Down	kAlphaDown	\$08	[B]	kCapB	\$9B	[E]	kCapE	\$9E
			[C]	kCapC	\$9C	[F]	kCapF	\$9F
						[G]	kCapG	\$A0
						[H]	kCapH	\$A1
[I]	kCapI	\$A2	[N]	kCapN	\$A7	[S]	kCapS	\$AC
[J]	kCapJ	\$A3	[O]	kCapO	\$A8	[T]	kCapT	\$AD
[K]	kCapK	\$A4	[P]	kCapP	\$A9	[U]	kCapU	\$AE
[L]	kCapL	\$A5	[Q]	kCapQ	\$AA	[V]	kCapV	\$AF

[M]	kCapM	\$A6	[R]	kCapR	\$AB	[W]	kCapW	\$B0
[X]	kCapX	\$B1						
[Y]	kCapY	\$B2	[_]	kSpace	\$99			
[Z]	kCapZ	\$B3	[.]	kColon	\$C6			
[θ]	kThetA	\$CC	[?]	kQuest	\$CA			
["]	kQuoteE	\$CB	[SOLVE]	kAlphaEnter	\$06			

Alpha-Alpha-Function Keys

Key Equate Value Key Equate Value Key Equate Value

[a]	kLa	\$E2	[d]	kLd	\$E5	[i]	kLi	\$EA
[b]	kLb	\$E3	[e]	kLe	\$E6	[j]	kLj	\$EB
[c]	kLc	\$E4	[f]	kLf	\$E7	[k]	kLk	\$EC
			[g]	kLg	\$E8	[l]	kLl	\$ED
			[h]	kLh	\$E9	[m]	kLm	\$EE
[n]	kLSmalln	\$EF	[s]	kLs	\$F4	[x]	kLx	\$F9
[o]	kLo	\$F0	[t]	kLt	\$F5	[y]	kLy	\$FA
[p]	kLp	\$F1	[u]	kLu	\$F6	[z]	kLz	\$FB
[q]	kLq	\$F2	[v]	kLv	\$F7			
[r]	kLSmallr	\$F3	[w]	kLw	\$F8			

*This is part of Learn TI-83 Plus Assembly In 28 Days
 Copyright (c) 2002, 2003, 2004 Sean McLaughlin
 See the file gfdl.html for copying conditions*

Scan Codes (For _GetCSC)

These are the codes returned from the `GetCSC` routine.

The [APPS] key is equated to `skMatrix` for portability to the TI-83. You may want to re-equate it in your programs if it's confusing.

Key	Equate	Value	Key	Equate	Value	Key	Equate	Value
[Y=]	skYEqu	\$35	[2nd]	sk2nd	\$36	[ALPHA]	skAlpha	\$30
[WINDOW]	skWindow	\$34	[MODE]	skMode	\$37	[X,T,θ,n]	skGraphVar	\$28
[ZOOM]	skZoom	\$33	[DEL]	skDel	\$38	[STAT]	skStat	\$20
[TRACE]	skTrace	\$32	<	skLeft	\$02	V	skDown	\$01
[GRAPH]	skGraph	\$31	Λ	skUp	\$04	>	skRight	\$03
[MATH]	skMath	\$2F	[x ⁻¹]	skRecip	\$2E	[x ²]	skSquare	\$2D
[APPS]	skMatrix	\$27	[SIN]	skSin	\$26	[,]	skComma	\$25
[PRGM]	skPrgm	\$1F	[COS]	skCos	\$1E	[(]	skLParen	\$1D
[VARS]	skVars	\$17	[TAN]	skTan	\$16	[)]	skRParen	\$15
[CLEAR]	skClear	\$0F	[^]	skPower	\$0E	[÷]	skDiv	\$0D
[LOG]	skLog	\$2C	[LN]	skLn	\$2B	[STO=>]	skStore	\$2A
[7]	sk7	\$24	[4]	sk4	\$23	[1]	sk1	\$22
[8]	sk8	\$1C	[5]	sk5	\$1B	[2]	sk2	\$1A
[9]	sk9	\$14	[6]	sk6	\$13	[3]	sk3	\$12
[×]	skMul	\$0C	[-]	skSub	\$0B	[+]	skAdd	\$0A
[0]	sk0	\$21						
[.]	skDecPnt	\$19						
[(-)]	skChs	\$11						
[ENTER]	skEnter	\$09						

*This is part of Learn TI-83 Plus Assembly In 28 Days
 Copyright (c) 2002, 2003, 2004 Sean McLaughlin
 See the file gfdl.html for copying conditions*