

TI-83+ Z80 ASM for the Absolute Beginner

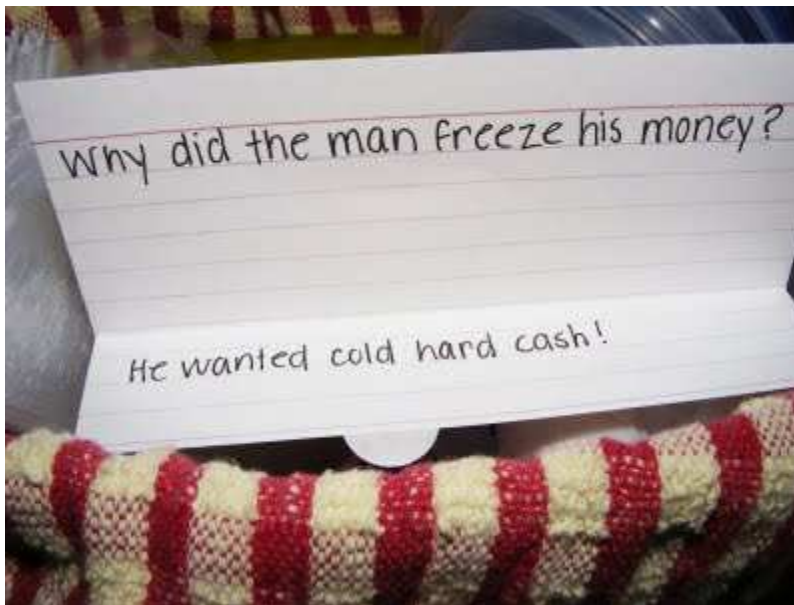
LESSON SEVENTEEN:

- *Introduction to Registers IX and IY*
 - *Sprites!*

INTRODUCTION TO REGISTERS IX AND IY

The Ti-83+ processor provides a couple of two-byte registers for special use. These registers are called index registers, and for a reason.

So, what do you think of when you hear the word “index?” Do you think of your index finger? Do you think about the index at the back of a school book? Or, how about indexing as in index cards?



Think about it. Let's say you are studying in a zoology class, and you need to prepare for a test that quizzes you on terms. You would probably write some terms on index cards. Then you keep these index cards together as a group, and you can access your terms to prepare for the test simply by using your index cards.

Now, if you're a smart person, you'd probably keep them in alphabetical order. Like so:

Alligator, Bear, Chicken, Dragon, Elephant, Fish, Gorilla, Hippo, Iguana, Jackel...

So now, you have all these cards in alphabetical order. So what are you going to do if you only need to look up "Gorilla?" Are you going to flip each card in search of Gorilla? I certainly hope not! Because now that all the cards are in alphabetical order, you can find your card "Gorilla" almost instantly!

Let's apply this to programming. Suppose that you have some data that you need to keep organized. Here's a list of heights, in inches, for people in a classroom:

Heights:

.db 45, 45, 45, 45, 45, 46, 47, 47, 48, 48, 48, 48, 49, 49, 50, 50, 50, 50, 50, 50, 51, 51, 51, 52, 52, 53, 53, 54, 55, 58, 60

Let's say that a girl named Suzie is always the sixth person on the list whenever there is a list of people in the classroom. She is the sixth person in the attendance list, and she is the sixth person on the list of who's doing chores. This is because the teacher likes to keep her lists well-organized and easy to work with.

Furthermore, Bob is the 9th person in the list, Sarah is number 13, Chewy is number 18, and Jason is number 23.

So let's say that the teacher is using her calculator to find the height of these good little students because she forgot their heights. But she's using a Ti-83+ calculator with the Ti-Basic section

corrupted, so she can only access this data by writing and running an ASM program. (Guys, let's just pretend, okay?)

What is she going to do? We did something like this once before when we looked at the characteristics of cars and trucks. Our solution was to LD HL, Label + Offset. For instance, Suzie is at Height + 5, and Chewy is at Height + 17.

However, at that time, we knew exactly where all the data was, because the location of the data was marked by a label. What if the location of this data of heights cannot be marked by a label? For example, let's say that the teacher's data is stored inside of an application variable, and this application variable is stored in RAM. As a programmer in Ti-Basic, I assume that you know that this application variable can be in different locations in RAM at any moment, depending on how much data/programs/etc. are added to RAM or deleted from RAM.

Thus, the location of the data is not consistent. Because of this, we can't say, for instance, LD HL, Height + 5 or LD HL, Height + 11. (Remember, Height, Height + 5 and Height + 11 are just representations of specific RAM locations, and just a reminder that this data in the application variable is not always in the same RAM location) We can still store the beginning of the data in HL, but we can't use labels to do so.

Let's say that the teacher found the location in HL. So now she could use the following code to access the heights of Suzie, Bob, Sarah, Chewy and Jason.

```

ld de, 5
add hl, de      ; HL points to the first person in the list.
                ; Since Suzie is number six in the list, we add 5 to HL
                ; to get to the sixth person in the list.

ld a, (hl)      ; Register A will contain the value of Suzie's height.

ld de, 3
add hl, de      ; Data location + 8
ld b, (hl)

ld de, 4
add hl, de      ; Data location + 12
ld c, (hl)

ld de, 5
add hl, de      ; Data location + 17
ld d, (hl)

; BE CAREFUL! Register D contains a value, so we need to save it!

push de
ld de, 5
add hl, de      ; Data location + 22
pop de
ld e, (hl)

```

Well, this works, but it's long and complicated. And there's another thing to consider: What if we need to access this data again very quickly? HL is used quite frequently in a program. Can you imagine how much pushing, popping, and saving of values is required when we use HL to access heights very, very frequently?

Let's use IX to recover values from our data. This is where you'll see the use of IX and IY. Suppose that IX points to the beginning of wherever the teacher's data for heights is, just like HL did.

```
ld a, ( IX + 5 )      ;Suzie's height
ld b, ( IX + 8 )
ld c, ( IX + 12 )
ld d, ( IX + 17 )
ld e, ( IX + 22 )
```

Could it really be this simple? Yes, yes, yes! This is what is so special about IX and IY. You can use it to easily access data found in lists. You cannot do this with HL, at least not this way.

But there's more! (What? There's more?) IX and IY can do almost anything that HL can do. You can do math with IX and IY. You can point to RAM addresses using IX and IY. And, you can push and pop IX and IY. These registers are perfect to use if HL is tied up and you need to do some math.

There must be a catch though, right? Right. In fact, there's several catches.

1. First and foremost, IX and IY are twice as slow as HL, and require more bytes for instructions than HL does. (For instance, ADD HL, DE requires only one byte, but ADD IX, DE requires two bytes.)
2. IX and IY cannot exchange their values with DE.
3. With HL, you are able to use H and L as individual bytes. You **can** do this with IX and IY, but then your

program **will not** run on a Ti-Nspire. (See the Appendixes for more information.)

4. Use of IY is not recommended for beginners, and these lessons will not encourage changing its value. The Ti-83+ uses IY extensively, so if you try to mess with it without knowing how, you can crash your calculator. You will learn later what to do about IY, but don't try to show off by purposely and needlessly incorporating it into your program.
5. Finally, when you select an **offset** for IX/IY (such as IX + **16**, where 16 is the offset), your offset can only be a number from -127 to 128.

So, you should stick with HL when your processor needs beefy, intensive work done. IX (remember, be very, very cautious with IY) should be used for data access / math only when HL is tied up or when you need to access several areas in the same space of data. But just remember, if you frequently need to access an area with a whole bunch of data, IX will save you a lot of time and processing power, and keep you from going insane. As you program more and more, you will understand times when it is important to choose index registers over HL.

SPRITES!

If you don't know what a sprite is, a sprite is a 2D image you can (and usually will) place anywhere on the screen at any given moment. Unlike pictures, sprites are constantly changing their position on the screen. Some examples of sprites include Mario, Kirby, and tiles used to make a map. (Did you ever see a screen of a Nintendo game that scrolls, and notice that the map contains a lot of small square pieces—like a puzzle—put together? Those pieces are called tiles.)

As you probably have guessed, displaying a sprite is not as easy as putting a picture on the screen. First of all, a sprite can be anywhere, and have any width and height. Thus, you have to have variables and/or registers to tell the calculator where the sprite is to be drawn, and how big the sprite is.

Secondly, you can't simply use LDIR to place a sprite on the screen. Why is that? Look at this diagram, where the red sprite is not 96 pixels wide.



What do you think would happen if we used LDIR? Well, this red sprite is 24 pixels wide. The screen is 96 pixels wide. So if we use LDIR, the calculator will THINK that we want a sprite 96 pixels wide, and I'm not good at picking the words to explain why. This is what you would come out with by using LDIR:



A third reason that sprites are harder to display than a picture—and the most important reason, **remember it**— is the method used to put a sprite on the screen. Here is a diagram showing a sprite that is drawn at the upper-left hand corner of the calculator screen, and thus the X coordinate of the sprite is $X = 0$. The sprite is 24 pixels wide and 3 pixels high. Like pictures, the sprite goes into plotscreen, and we use $hl = \text{plotscreen}$ to draw the sprite to plotscreen. For instance, if we put data in the first byte of plotscreen, we put it in (hl). If we put data in the third byte of plotscreen, we put in $(hl + 2)$ by adding 2 to HL. Don't forget, there are eight pixels for every byte in plotscreen.

<i>These 8 pixels go into (HL).</i>	<i>These 8 pixels go into (HL + 1).</i>	<i>These 8 pixels go into (HL + 2).</i>
<i>These 8 pixels go into (HL + 12). (Why 12?)</i>	<i>These 8 pixels go into (HL + 13).</i>	<i>These 8 pixels go into (HL + 14).</i>
<i>These 8 pixels go into (HL + 24).</i>	<i>These 8 pixels go into (HL + 25).</i>	<i>These 8 pixels go into (HL + 26).</i>

So here's what the first part of plotscreen will look like, with the sprite added to it:

```
%11111111, %00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%11111111, %00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
```

Seems okay. But what happens if we want to put the sprite in another X coordinate, say $X = 16$ instead of $X = 0$, and the Y coordinate never changes? Well, $16 / 8$ bits is 2 bytes, so that means that the picture starts on the **THIRD**, not the first byte, of plotscreen. So we could do $(HL + 2)$, $(HL + 3)$, $(HL + 4)$, $(HL + 14)$, etc.

```
%00000000, %00000000, %11111111, %00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%00000000, %00000000, %00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%00000000, %00000000, %11111111, %00000000, %11111111, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
```

But...what about if we start the picture at the coordinate $X = 1$? Well, $1/8 = \dots$ Uh oh, we've got a problem. We have to start the picture at the next $1/8^{\text{th}}$ of a byte in plotscreen (meaning bit 6, not bit 7, of HL). (When we can't start a sprite at the beginning of a byte, we say that the sprite is **not aligned**.) But we can't say $(HL + 1/8)$, $(HL + 1 \frac{1}{8})$, $(HL + 2 \frac{1}{8})$, $(HL + 12 \frac{1}{8})$, etc. How do we get the picture into the right coordinates?

Well, notice that the first byte of picture data is `%11111111`. These pixels must be displayed at $X = 1$, $X = 2$, $X = 3$, $X = 4$, $X = 5$, $X = 6$, $X = 7$ and $X = 8$. Now, remember that there are 8 pixels per byte in plotscreen, and remember that the very first bit of the very first byte of plotscreen pertains to $X = 0$ (and $Y = 0$). Since the picture does not start on the first bit of plotscreen, we don't mess with that first bit. We

start on the second bit, which is bit 6 of the first byte of `plotsscreen`. However, this in turn leads to the conclusion that we only have 7 bits, not 8 bits, available to us in that byte. So we can only put the first seven 1s of our picture data into that byte.

What do we do with the remaining 1 in that first byte of picture data? We have to move it to the NEXT BYTE of plotscreen.

Of course, we have to make these adjustments to the other bytes of picture data as well.

%01111111, %10000000, %01111111, %10000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%00000000, %01111111, %10000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000
%01111111, %10000000, %01111111, %10000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000, %00000000

(Does that look like we shifted the sprite data one pixel to the right? Well, that's exactly what we do in that kind of situation!)

And finally, our fourth problem with displaying sprites as opposed to pictures is what happens when our sprite is displayed too far to the left, too far to the right, or too far up or down. Plotsscreen is just a long line of RAM, so if you display a sprite too far up or too far down, you will accidentally write data beyond the boundaries of plotsscreen and will most likely corrupt some of your calculator's RAM. As for too far to the left or the right, all the data for the rows of a screen are strung right next to each other in plotsscreen. So if you display data that goes beyond the 96th pixel or 0th pixel of a row, the data will simply extend to other rows. Here's what the first part of plotsscreen will look like if we display our 24 x 3 sprite at $Y = 0$ and $X = 88$.

[illegible]

Well, don't fret eager beavers, because there is something TI provided that fixes all four of these problems! This is the easiest and most convenient method for displaying a sprite.

B_CALL _DisplayImage

Draws a Sprite on the screen. HL points to your image data, which is picture data EXCEPT that you need two bytes at the beginning for the height of your sprite and the width of your sprite. Register D contains the Y coordinate of where you want to draw the sprite, and Register E contains the X coordinate.

Examples: LD HL, Small Sprite

LD DE, 2 * 256 + 3

;THIS IS THE SAME THING AS SAYING LD D, 2 and LD E, 3

B_CALL _DisplayImage

SmallSprite:

.db 2, 16, %11111111, %00000000, %00000000, %11111111

Here's a simple example program. It will use `_DisplayImage` to draw a tree, wait for a keypress, and then use `_DisplayImage` to put a happy little bird on top of the tree.

```
#include "ti83plus.inc"

.org $9D93

.db  t2ByteTok, tAsmCmp

    B_CALL _ClrLCDFull

    ld hl, Tree

    ld d, 63-43 ;We want to display the tree so that the roots
                ;touch the bottom of the calculator screen.

                ;We take the height of the tree, and subtract
                ;this value from how many pixels long the screen is.

    ld e, 95 - 24

    B_CALL _DisplayImage

    B_CALL _getKey

    ld hl, Bird

    ld de, 15 * 256 + 75 ;Same as

                ;LD D, 25

                ;LD E, 75

    B_CALL _DisplayImage

    B_CALL _getKey

    B_CALL _ClrLCDFull

    ret
```

;CONTINUED ON NEXT PAGE

Bird:

```
; Width:18 Height:16 (48 bytes)
.db %00000011,%11000000,%00000000
.db %00000100,%00100000,%00000000
.db %00001000,%00010000,%00000000
.db %11110011,%00010000,%00000000
.db %10000011,%00010000,%00000000
.db %01000000,%00001000,%00000000
.db %00100000,%00001000,%00000000
.db %00010000,%00000100,%00000000
.db %00001000,%00000100,%00000000
.db %00001000,%00000100,%00000000
.db %00001000,%00000100,%00000000
.db %00001000,%00000100,%00000000
.db %00001000,%00000100,%00000000
.db %00000100,%00000101,%10000000
.db %00000010,%00011010,%00000000
.db %00000001,%11100101,%00000000
.db %00000000,%00000100,%00000000
```

Tree:

```
; Width:24 Height:43 (129 bytes)
.db 43, 24
.db %00000000,%01111100,%00000000
.db %00000001,%11111111,%00000000
.db %00000111,%11111111,%11000000
.db %00001110,%01111111,%11100000
.db %00011100,%00111111,%11110000
.db %00011100,%00111111,%11110000
.db %00111110,%01111111,%00111000
.db %00111111,%11111110,%00011000
.db %01110011,%11111110,%00011100
.db %01100001,%11111111,%00111100
.db %01100001,%11111111,%11111100
.db %01110011,%11111111,%11111100
.db %01111111,%11111111,%11111100
.db %00111111,%11001111,%11111000
.db %00111111,%10000111,%11111000
.db %00011111,%10000111,%11110000
.db %00011111,%11001111,%11110000
.db %00001111,%11111111,%11100000
.db %00000111,%11111111,%11000000
.db %00000001,%11111111,%00000000
.db %00000000,%01111100,%00000000
.db %00000000,%01000010,%00000000
.db %00000000,%01000010,%00000000
.db %00000000,%01000010,%00000000
.db %00000000,%10010010,%00000000
.db %00000000,%10100010,%00000000
.db %00000000,%10100001,%00000000
.db %00000000,%10010001,%00000000
.db %00000000,%10000001,%00000000
.db %00000000,%10000001,%00000000
.db %00000000,%10001001,%00000000
.db %00000000,%10000101,%00000000
.db %00000000,%01000001,%00000000
.db %00000000,%01000000,%10000000
.db %00000000,%01000000,%10000000
.db %00000000,%01000100,%10000000
.db %00000000,%01001000,%10000000
.db %00000001,%00000000,%01000000
.db %00000110,%00100000,%01000000
.db %00001000,%01011000,%00100000
.db %00010000,%01000100,%00010000
```

Pretty cool, huh? Except that we have two problems. First of all, as you probably noticed, there's a lot of white space on top of the tree, the white space surrounding the bird. This is because of all those "0s" in the sprite data of the bird. Remember that sprites are drawn similar to pictures, so 1s mean black pixels and 0s mean white pixels.

The second problem is, as usual, B_CALL routines are slow. While I don't recommend B_CALL routines for processor intensive games, I certainly want to point out that they are TERRIBLE for graphics routines in said games.

But, I dare say that the biggest problem is, how do we draw a picture that doesn't mess up the background like our bird did? Easy! We just tell the calculator that all 0s are **TRANSPARENT!** However, we have to make our own sprite routine to do this, and so we will do that for the remainder of this chapter.

Most programming tutorials in ASM will teach you how to make 8x8 sprites, and will not give you the "art" of drawing any-sized sprites. I agree that there are many, many games that have only 8x8 sprites (which are drawn faster than any-sized sprites), but there are exceptions out there. The famous Super Mario for the Ti-83+ uses 8x8 sprites. Wolfenstein for the Ti-83+ uses any-sized sprites. My game Seek and Destroy uses an any-sized sprite routine, AND an 8x8 sprite routine.

Now, making an any-sized sprite routine is not too hard. I repeat, not too hard. And, once you learn how to do it, you can easily make a fast 8x8 routine if you need to. This any-sized sprite routine will not be the fastest one you can get your hands on, but it is fast for most games, and it is most definitely one of the easiest to learn.

For this routine, any 1s in the sprite data will be drawn as black pixels on the screen. Any 0s will be transparent. We will use Register A to hold the X position of our sprite, and we will use Register L to hold the Y position. For the height of the sprite, we use Register B. We use an area of RAM called `Sprite_Width` to hold the width of the sprite. This width **MUST** be the number of **bytes** long, not pixels long. Finally, we will use Register IX to tell the calculator where the sprite actually is.

It is very important for this lesson that you remember that plotsscreen consists of 12 bytes for every row of the screen, because the screen is 96 pixels long and 1 byte takes care of 8 pixels. Plotsscreen has 64 rows for the 64 rows on the calculator screen.

With this information of plotsscreen in mind, here's the basic process for our sprite routine, before we go into details: We will first find the place in plotsscreen to put our sprite data. Since plotsscreen contains exactly what data will be displayed on the calculator screen, we need to find the location in plotsscreen to copy our sprite data so that the sprite will appear in the right place on the screen. Then, we go through our sprite and place the data into plotsscreen: Byte by byte, row by row. We put one row of sprite data into plotsscreen, 8 pixels at a time. (This is why the width of our sprite must be in bytes, that is, multiples of 8.) When that row is finished, we move onto the next row, until we have finished placing the sprite into plotsscreen.

Now, onto our routine and all the details explaining it. It might not hurt for you to take notes ☺ You should assume for this lesson that plotsscreen already holds the background we want drawn, just like in the previous example program when we drew the bird on top of the tree.

The first thing we need to do is find out where to place our sprite in plotsscreen. We start with the Y coordinate. Since the width of the

screen on the Ti-83+ is 96 pixels, the width of the screen 12 bytes long (remember, 1 byte for every 8 pixels). So we multiply the Y coordinate by 12 to get to the correct ROW location in plotscreen.

Now, granted, we haven't learned how to multiply by twelve. You'll learn the full details in the Appendixes—remember that this is a beginner's tutorial, so most of our multiplication in the actual lessons will use B_CALLs. However, I will say that for this sprite routine, we will use the following formula to multiply Y by 12:

$$((Y * 2) + Y) * 2 * 2 = (2Y + Y) * 2 * 2 = 3Y * 2 * 2 = 6Y * 2 = 12Y$$

Put_Sprite:

; A = x coordinate

; L = y coordinate

; B = number of rows

; Variable Sprite_Width = Width of Sprite,

; in **bytes**

; IX = address of sprite

LD H, 0 ;L contains our Y coordinate

LD D, H

LD E, L

;HL and DE now both equal the Y coordinate for the sprite.

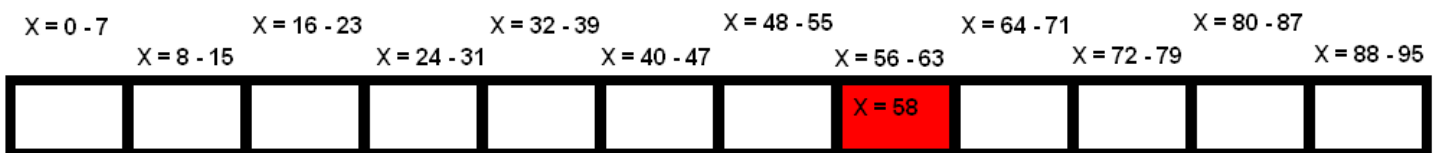
```

ADD    HL, HL      ;(Y multiplied by 2 = 2Y)
ADD    HL, DE      ;2Y + Y
ADD    HL, HL      ;3Y * 2
ADD    HL, HL      ;6Y * 2

```

So this gives the correct position for our sprite in plotscreen, in terms of the Y coordinate. Now we need to find the correct position in terms of the X coordinate. However, since there are twelve bytes for every row of the screen, with 8 pixels for every byte, we have to move our sprite to a particular **byte** of plotscreen. So we divide our X position by 8 and remove the remainder to get to the correct byte.

For example, suppose that the X coordinate for our sprite is 58. 58 divided by 8 is 7 remainder 2, so we go to byte 7 (The first byte of the row is, of course, byte 0.)



```

LD      E, A ;We don't want to lose the
           ;value we have stored inside
           ;of A by messing around with A.

```

;Divide the X Coordinate by 8 to get to the correct X coordinate for plotsscreen

```
SRL    E
```

```
SRL    E
```

```
SRL    E
```

;Now, since D already equals "0", add DE to HL, and we will now have the correct X AND Y coordinates in plotsscreen.

```
ADD HL, DE
```

;Now that we know which byte to start placing picture data in, we add this value to plotsscreen, so the calculator will place picture data in the correct location of the screen data.

```
LD DE, plotsscreen
```

```
ADD HL, DE
```

Now, we need to concern ourselves with what to do if the sprite starts in a position that is not at the beginning of a byte in plotscreen. Review pages 9 – 11 if you need to. As you can see from pages 9-11, if the X coordinate of the sprite is a multiple of eight, the sprite starts at the beginning of a **byte**, rather than an unusual **bit**. We will start by telling the calculator what to do if the sprite is at an X coordinate that is a multiple of 8.

```
AND      7
JR       Z, Sprite_Is_Aligned
```

To see if our sprite is at an X coordinate that is a multiple of 8, we start by clearing the top 5 bits of register A, which holds (or did hold) our X Coordinate. After AND 7, if the lower three bits of our new value of register A all equal zero, the X coordinate is a multiple of 8. This is simply the nature of binary numbers...if a number is a multiple of eight, then the number will have zeros in the last three bits when you view the number as a binary number.

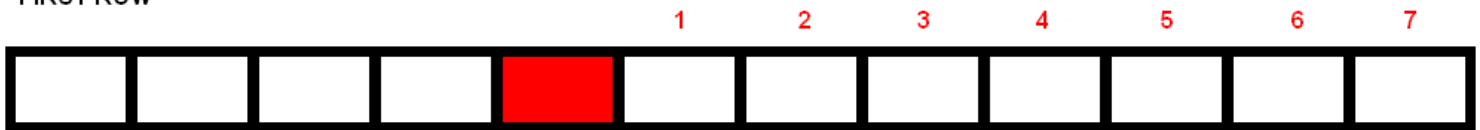
Just remember that by ANDing A with 7, we made the top 5 bits of Register A equal to zero. So if the last 3 bits of Register A are **also** equal to zero, the entire number equals zero and the zero flag (Z Flag) is therefore set. Thus, if AND 7 sets the zero flag, the X coordinate is a multiple of 8, and the sprite is aligned.

Let's go to the next step. As was aforementioned, we take the route of drawing a sprite (in this case, aligned) by drawing one row at a time. Remember that a row of the Ti-83+ screen takes 12 bytes. Since the width of our sprite is in bytes, it will take a certain number of those 12 bytes to put one row of the spryte into plotscreen. At a risk of

stating the obvious, a sprite 3 bytes long will take 3 of the twelve bytes in each row. A sprite 7 bytes long will take 7 of the twelve bytes in each row.

Now, under normal circumstances, since the rows of the Ti-83+ screen each consist of 12 bytes, we can jump from the X position in one row to the same X position in the next row simply by adding 12.

FIRST ROW



SECOND ROW



THIRD ROW



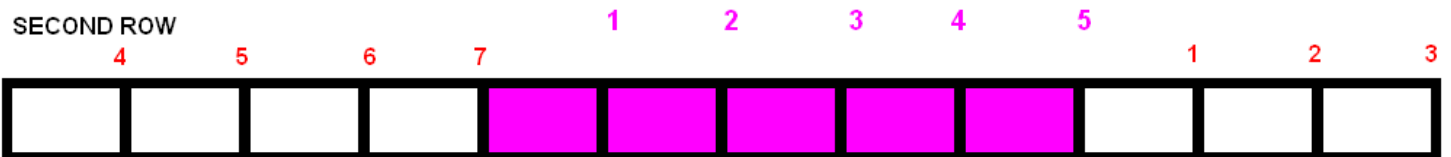
However, when we draw a single row of sprite data (which takes a certain amount of the 12 bytes), we are that much closer to the X position in our next row. So we don't need to add 12. Rather, we add (12 – the width of the sprite in bytes). For instance, suppose that our sprite is 5 bytes long, at X = Byte 4 of the row. (Byte 0 is the first byte of the row.) Since we have advanced 5 bytes in the row, we only need to add 7 to get to X = Byte 4 in the next row we need to draw.

FIRST ROW

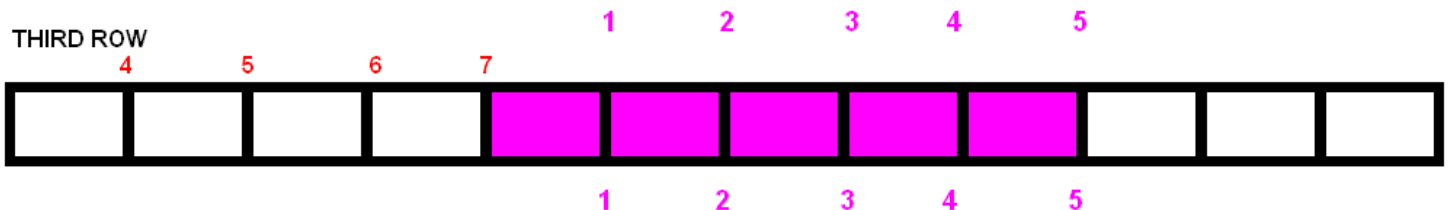
SPRITE 5 BYTES LONG



SECOND ROW



THIRD ROW



We are going to use register DE to hold the number of bytes we need to add to get to the next row of plotsscreen, and of course, we go to the next row of the sprite when we are done drawing the previous row of the sprite.

Sprite_Is_Aligned:

```
PUSH HL      ;We need to use HL temporarily,
              ;but HL holds the position in
              ;plotsscreen to draw our sprite.
              ;Save the value.

LD A, 12      ;Twelve bytes per row
LD HL, Sprite_Width
SUB (HL)      ;12 minus the width of the sprite
LD D, 0
LD E, A       ;Now, DE contains the number
              ;we need to add to get to the correct
              ;X position in each row after the
              ;previous row has finished being drawn
POP HL        ;HL holds where the sprite is placed
              ;in plotsscreen.
```

So, why did we put the number of rows in the sprite into register B? Well, since we're drawing one row of the sprite at a time, we're going to use DJNZ as a For...Loop. (For each row of the sprite, loop.) However, as you can figure out, we can also use DJNZ for the number of bytes long the sprite is. (For each byte long the sprite is, loop). This is because, as was aforementioned, we place a row of sprite data byte by byte. So, we will be performing two DJNZ loops, one inside of the other. I'm quite sure you've done a loop-inside-of-a-loop in Ti-Basic ☺ By pushing BC, we can save the number of times left to loop for drawing rows of the sprite. That leaves register B free to use for the inside loop, the number of bytes left to place in the row of plotscreen.

Put_Loop_Sprite_Aligned:

```
PUSH BC ;Save the number of rows left
        ;to draw

LD A, (Sprite_Width)

LD B, A ;Register B now contains the
        ;number of bytes long our
        ;sprite is.
```

Now, remember that IX holds the location of our sprite. IX points to the first byte of our sprite data, and as you might recall, we're putting the data in plotscreen one byte at a time. Since HL points to the location in plotscreen that the sprite data should go, we simply take the data found in IX and copy it to the location in HL.

However, we want the 0s to be transparent. So how can we do that? By using OR. Remember from lesson 16 that when you OR a number with register A, only the 1s in the number will affect the bits in register A. If your number has bits equal to zero, those respective bits in register A will be left alone. And if you think about it, transparency means “leave this part of the graphics alone.”

Of course, when we draw the sprite to plotsscreen, we already have our background image stored in plotsscreen. So when we take a byte of sprite data and OR it with a byte of background image data already in plotsscreen, all zeros will leave those parts of the background unaffected. Only the 1s in the byte of sprite data will be drawn to plotsscreen, because after all, those are un-transparent parts we want to have drawn in.

Put_Loop_Sprite_Aligned_Two:

```
LD      A, (IX) ;Gets a byte of our sprite data
OR (HL)      ;Takes care of transparency
LD      (HL), A      ;Stores the final result
                        ;into plotsscreen
INC     IX           ;Goes to the next byte of the
                        ;sprite data
INC     HL           ;Goes to the next byte in
                        ;plotsscreen so that we can
```



```
        ;place the next byte of sprite data  
        ;in the correct screen location  
;Remember, register B contains the width, in  
;bytes, of our sprite
```

```
DJNZ Put_Loop_Sprite_Aligned_Two
```

When we have finished placing our sprite byte-by-byte, we move onto the next row of sprite data, and therefore the next row of plotsscreen.

```
Put_Loop_Sprite_Aligned_Next_Row:
```

```
POP BC ;Register B now contains the number  
        ;of rows left. This value will be  
        ;decreased, and if the result is  
        ;zero, our sprite is finished being  
        ;drawn entirely, since there are  
        ;no more rows left to draw.
```

```

ADD HL, DE      ;Move to the correct X
                 ;Coordinate of the next
                 ;row in plotsscreen
DJNZ Put_Loop_Sprite_Aligned

;Our sprite has been drawn completely.

RET

```

So, if the X coordinate for our sprite is a multiple of 8, that's all there is to it! But of course, we want to be able to draw our sprite at **any** X coordinate. If you didn't review pages 9-11, I recommend you do so right now.

Go back to the following 2 lines:

```

AND      7
JR       Z, Sprite_Is_Aligned

```

We will now type up the rest of the routine after these two lines. The rest of the code will take care of sprites that are not aligned.

So, did you notice that when X equaled one, we needed to put TWO bytes into HL for one byte of sprite data? This is because it took two bytes to hold two “halves” of the sprite with the pixels shifted.

We need **only** Register A to put sprite data into plotscreen when our sprite was at an X position that was a multiple of 8—that is, when the sprite was aligned. For a sprite that is not aligned, we need an extra register for the extra second byte. We will use Register C to hold the first half of the sprite data that gets shifted.

In this next section of code, the only differences are the label names and the one extra line, where Register C is made to equal to register A. You’ll see why that is in a moment, but in the meantime, there’s no need for me to explain everything else a second time.

```
LD      C, A

PUSH HL      ;We need to use HL temporarily,
              ;since HL holds the position in
              ;plotsscreen to draw our sprite.
              ;Save the value.

LD A, 12      ;Twelve bytes per row
LD HL, Sprite_Width
SUB (HL)      ;12 - the width of the sprite
LD D, 0
LD E, A      ;Now, DE contains the number
```

```

;we need to add to get to the correct
;X position in each row.

```

```

POP HL      ;HL holds where the sprite is placed
            ;in plotsscreen.

```

```

Put_Unaligned_Sprite_Row:

```

```

PUSH BC

```

```

;Stores width into B, which will be decreased 1
by 1 for each byte placed into plotsscreen until
one row is complete

```

```

LD A, (Sprite_Width)

```

```

LD B, A

```

Now, remember from page 11 that when the sprite was placed at the X coordinate $X = 1$, the data was shifted to the right 1 pixel. Thus, it's logical that at $X = 2$, the data is shifted 2 pixels. If the sprite is placed at $X = 5$, it's shifted 5 pixels. If the sprite is placed at $X = 8$, the sprite is aligned, so we don't worry about shifting. If the sprite is placed at $X = 9$, the data is shifted one pixel. If the sprite is placed at $X = 10$,

the data is shifted two pixels. If the sprite is placed at $X = 17$, the data is shifted 1 pixel. Do the math: The number of pixels to shift is the remainder of $X/8$.

However, did you notice something? AND 7 is the same thing as dividing by 8 and taking the remainder. Thus, when you use AND 7, you get more than just whether the sprite is aligned or not. Register A will also contain the number of pixels the sprite data needs to be shifted to the right! Sweet, huh? It's just another one of those "nature of binary numbers" kinds of things.

We stored the value of Register A—the number of pixels to shift—into Register C because this value is used often, and needs to be saved. We now push BC to save that value.

```
Put_Unaligned_Sprite_Start_Byte_By_Byte:
```

```
    PUSH BC
```

Now, to shift the pixels in the sprite, we're going to use another DJNZ. (For each time we need to shift the pixels, loop.) We store the value of Register C (which holds the number of pixels to shift) into register B so that we can use DJNZ.

Then, we get our sprite data from IX so that we can put it into plotscreen. However, remember that we need two bytes, using Registers C and A, to put into plotscreen, since the sprite does not fall on an X position that is a multiple of 8. As a final review of page 11, two bytes are needed because all the bits that can't be placed in the first byte get put into the second byte. As was aforementioned Register C will be our first byte for the first half, and register A will be our second.

Register C is going to start with the byte of sprite data fully intact. We shift the data one pixel at a time. We use SRL C, because we want the bit shifted off the edge to be stored into the carry flag. (Besides, SRL will place zeros at the front of Register C, which will in turn cause transparency at the front, which is a good thing.) Then, this value of the carry flag can be safely transferred to register A.

```
LD      B, C      ;Stores the number of pixels to
                  ;shift into register B.
```

```
LD      C, (IX) ;Retrives our byte of sprite
                  ;data.
```

```
XOR     A         ;Sets A to equal 0. We want
                  ;nothing but "transparency"
                  ;inside of A so that when we
                  ;shift bits into it, we
                  ;won't have any unnecessary
                  ;1s.
```

```
Shift_Pixels_Loop:
```

```
SRL     C         ;Shift pixels once
```

```
RRA                     ;Puts pixels into A that are not
                        ;allowed to be in C
```

```
DJNZ    Shift_Pixels_Loop
```

Now we have two bytes of sprite data, each of which we'll place inside of `plotsscreen`.

```
;Remember, register A holds our SECOND byte for  
;plotsscreen, not our first. In order to  
;avoid losing our value in A, we increase HL so  
;that A can be placed into the correct location  
;of plotsscreen.
```

```
INC    HL  
OR     (HL)  
LD     (HL), A
```

```
;Now to get to the correct location for register  
;C.
```

```
DEC    HL  
LD     A, C  
OR     (HL)  
LD     (HL), A
```

Now, remember that just like for aligned sprites, this is only for one byte of sprite data. The sprite might be only one byte in length, or it might be more than that. However, the value in Register B has been

destroyed, so it currently does not hold how many bytes are left to draw for the current row. We can recover that value by popping BC, since the value for register B was saved a while back. At the same time, the value for the number of pixels to shift for each byte of sprite data is recovered/restored into register C.

```
POP BC
```

```
;Now, move to the next byte in plotsscreen, and
```

```
;move to the next byte of sprite data
```

```
INC HL
```

```
INC IX
```

```
DJNZ Put_Unaligned_Sprite_Start_Byte_By_Byte
```

```
;Move to the next row, to draw the next row of
```

```
;the sprite.
```

```
ADD HL, DE
```

```
;The value for register B needs to be, in this
```

```
;case, the number of rows left to draw.
```

```
POP BC
```

```
DJNZ Put_Unaligned_Sprite_Row
```

```
RET ;The sprite is finished being drawn.
```


Phew! That was a lot of information. Most tutorials don't give that much info for a sprite routine. But I wanted to make sure you know everything you possibly can. Because if you don't understand this—for instance, if you decided to read quickly through this chapter—you are not going to survive the rest of the tutorials. I highly recommend you go back and read pages 9-32 if you skimmed the chapter.

Okay, a few more things. First of all, on the next 2 pages is another example program to draw a blackbird with transparent pixels on the same tree. (Don't forget to include your sprite routine in the program!) Secondly, after that example, I have taken the time to display the whole sprite routine on the remaining pages of the lesson so that you can make sure you are not missing anything.

Finally, be ready for lesson 18. We didn't cover everything important about sprites, so we will finish up in that lesson.

Great job on getting this far, and good luck!

```

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

    B_CALL _ClrLCDFull

    ld ix, Tree
    ld a, 3
    ld (Sprite_Width), a
    ld b, 43
    ld l, 20
    ld a, 71

    call Put_Sprite
    B_CALL _GrBufCpy

    B_CALL _getKey
    ld ix, Bird

    ld a, 2
    ld (Sprite_Width), a
    ld b, 16

    ld l, 10
    ld a, 75

    call Put_Sprite
    B_CALL _GrBufCpy
    B_CALL _getKey
    B_CALL _ClrLCDFull

    ret

```

Sprite_Width:

```
.db 0
```

;CONTINUED ON NEXT PAGE

Bird:

```
.db %00000011,%11000000
.db %00000111,%11100000
.db %00001111,%11110000
.db %11111111,%11110000
.db %11111111,%11110000
.db %01111111,%11111000
.db %00111111,%11111000
.db %00011111,%11111100
.db %00001111,%11111100
.db %00001111,%11111100
.db %00001111,%11111100
.db %00001111,%11111100
.db %00000111,%11111111
.db %00000011,%11111110
.db %00000001,%11111111
.db %00000000,%00011100
```

Tree:

```
.db %00000000,%01111100,%00000000
.db %00000001,%11111111,%00000000
.db %00000111,%11111111,%11000000
.db %00001110,%01111111,%11100000
.db %00011100,%00111111,%11110000
.db %00011100,%00111111,%11110000
.db %00111110,%01111111,%00111000
.db %00111111,%11111110,%00011000
.db %01110011,%11111110,%00011100
.db %01100001,%11111111,%00111100
.db %01100001,%11111111,%11111100
.db %01110011,%11111111,%11111100
.db %01111111,%11111111,%11111100
.db %00111111,%11001111,%11111000
.db %00111111,%10000111,%11111000
.db %00011111,%10000111,%11110000
.db %00011111,%11001111,%11110000
.db %00001111,%11111111,%11100000
.db %00000111,%11111111,%11000000
.db %00000001,%11111111,%00000000
.db %00000000,%01111100,%00000000
.db %00000000,%01000010,%00000000
.db %00000000,%01000010,%00000000
.db %00000000,%10010010,%00000000
.db %00000000,%10100010,%00000000
.db %00000000,%10100001,%00000000
.db %00000000,%10010001,%00000000
.db %00000000,%10000001,%00000000
.db %00000000,%10000001,%00000000
.db %00000000,%10001001,%00000000
.db %00000000,%10000101,%00000000
.db %00000000,%10000101,%00000000
.db %00000000,%01000001,%00000000
.db %00000000,%01000000,%10000000
.db %00000000,%01000000,%10000000
.db %00000000,%01000100,%10000000
.db %00000000,%01001000,%10000000
.db %00000000,%10001000,%10000000
.db %00000001,%00000000,%01000000
.db %00000110,%00100000,%01000000
.db %00001000,%01011000,%00100000
.db %00010000,%01000100,%00010000
```

```

Put_Sprite:
; A = x coordinate
; L = y coordinate
; B = number of rows
; Variable Sprite_Width = Width of Sprite,
; in bytes
; IX = address of sprite

    LD     H, 0        ;L contains our Y coordinate
    LD     D, H
    LD     E, L

;HL and DE now both equal the Y coordinate for the sprite.

    ADD    HL, HL      ;(Y multiplied by 2 = 2Y)
    ADD    HL, DE      ;2Y + Y
    ADD    HL, HL      ;3Y * 2
    ADD    HL, HL      ;6Y * 2

    LD     E, A        ;We don't want to lose the
                        ;value we have stored inside
                        ;of A by messing around with A.

;Divide the X Coordinate by 8 to get to the correct X coordinate
for plotsscreen

    SRL    E
    SRL    E
    SRL    E

;Now, add this to HL, and we will have the correct X AND Y
coordinate in plotsscreen.

    ADD    HL, DE

;Now that we know which byte to start placing picture data in,
we add this value to plotsscreen, so the calculator will place
picture data in the correct location of the screen data.

    LD     DE, plotsscreen
    ADD    HL, DE

    AND    7

```

```

JR      Z, Sprite_Is_Aligned

LD      C, A
PUSH HL      ;We need to use HL temporarily,
              ;since HL holds the position in
              ;plotsscreen to draw our sprite.
              ;Save the value.

LD A, 12      ;Twelve bytes per row
LD HL, Sprite_Width
SUB (HL)      ;12 - the width of the sprite
LD D, 0
LD E, A      ;Now, DE contains the number
              ;we need to add to get to the correct
              ;X position in each row.

POP HL      ;HL holds where the sprite is placed
              ;in plotsscreen.

Put_Unaligned_Sprite_Row:

PUSH BC

;Stores width into B, which will be decreased 1 by 1 for each
byte placed into plotsscreen until one row is complete

LD A, (Sprite_Width)
LD B, A

Put_Unaligned_Sprite_Start_Byte_By_Byte:
PUSH BC

LD      B, C      ;Stores the number of pixels to
                  ;shift into register B.
LD      C, (IX)    ;Retrives our byte of sprite
                  ;data.
XOR     A          ;Sets A to equal 0. We want
                  ;nothing but "transparency"
                  ;inside of A so that when we
                  ;shift bits into it, we
                  ;won't have any unnecessary
                  ;1s.

Shift_Pixels_Loop:

SRL     C          ;Shift pixels once
RRA     ;Puts pixels into A that are not

```

```

                                ;allowed to be in C
    DJNZ    Shift_Pixels_Loop

;Remember, register A holds our SECOND byte for
;plotsscreen, not our first. We increase HL so
;that A can be placed into the correct location
;of plotsscreen.
    INC     HL
    OR      (HL)
    LD      (HL), A

;Now to get to the correct location for register
;C.
    DEC     HL
    LD      A, C
    OR      (HL)
    LD      (HL), A

    POP BC

;Now, move to the next byte in plotsscreen, and
;move to the next byte of sprite data

    INC HL
    INC IX

    DJNZ Put_Unaligned_Sprite_Start_Byte_By_Byte

;Move to the next row, to draw the next row of
;the sprite.
    ADD     HL, DE

;The value for register B needs to be, in this
;case, the number of rows left to draw.
    POP     BC
    DJNZ    Put_Unaligned_Sprite_Row
    RET     ;The sprite is finished being drawn.

```

Sprite_Is_Aligned:

```

    PUSH HL                ;We need to use HL temporarily,
                           ;since HL holds the position in
                           ;plotsscreen to draw our sprite.
                           ;Save the value.

    LD A, 12               ;Twelve bytes per row
    LD HL, Sprite_Width

```

```

SUB (HL)    ;12 - the width of the sprite
LD D,0
LD E, A      ;Now, DE contains the number
              ;we need to add to get to the correct
              ;X position in each row.

POP HL      ;HL holds where the sprite is placed
              ;in plotsscreen.

```

Put_Loop_Sprite_Aligned:

```

PUSH BC
LD A, (Sprite_Width)
LD B, A      ;Register B now contains the
              ;number of bytes long our
              ;sprite is.

```

Put_Loop_Sprite_Aligned_Two:

```

LD A, (IX)    ;Gets a byte of our sprite data
OR (HL)       ;Takes care of transparency
LD (HL), A    ;Stores the final result
              ;into plotsscreen
INC IX        ;Goes to the next byte of the
              ;sprite data
INC HL        ;Goes to the next byte in
              ;plotsscreen so that we can
              ;place the next byte of
              ;sprite data
;Remember, register B contains the width, in
;bytes, of our sprite

```

```

DJNZ Put_Loop_Sprite_Aligned_Two

```

Put_Loop_Sprite_Aligned_Next_Row:

```

POP BC      ;Register B now contains the number
              ;of rows left. This value will be
              ;decreased, and if the result is
              ;zero, our sprite is finished being
              ;drawn entirely, since there are
              ;no more rows left to draw.

```

```
ADD HL, DE           ;Move to the correct X
                     ;Coordinate of the next
                     ;row in plotsscreen
DJNZ Put_Loop_Sprite_Aligned

;Our sprite has been drawn completely.

RET
```