TI-83+ Z80 ASM for the Absolute Beginner

APPENDIX B:

- *Sprites*

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, and since they can be moved, they are sprites.) Sprites allow you to draw ships that can fly across the screen, balls that can bounce up and down, and even monsters that come from a corner and wreck havoc.

Creating your own sprite routine is an advanced topic, and so I will instead give you a sprite routine to use freely. Special thanks goes to Chris Shappell (Buckeye Dude) and, especially, Jim E for this sprite routine. Look in the folder "Appendix B Files."

To create a sprite, you can do everything you did in creating a full-sized picture, with one exception: Before every sprite label, bm_min_w should come before it and be equal the number of pixels wide your sprite is. For instance, a sprite 32 pixels wide: bm_min_w = 32

However, there are some things you need to be aware about with sprites. Just like pictures, sprite data consists of 1s and 0s. However, if we let sprites be drawn like a picture, 1s for black pixels and 0s for white pixels, what happens to transparency? There is none. So drawing a sprite will mess up the background with the black and white pixels that are drawn.

Now, you can take care of this problem, but you have to decide how you want to draw your sprite with transparency. You can XOR the

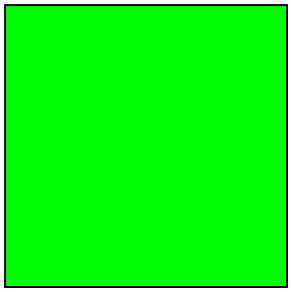
sprite, OR it, OVERWRITE it, AND it, or MASK it. Most of the time, you will use OR, XOR, or MASK, so the routine I have given you will allow you to draw a sprite using these three methods.

If you draw a sprite using OR, any 1s will be drawn as black pixels, and 0s will be transparent. If you draw a sprite using XOR, 0s will still be transparent. But with XOR, 1s will invert any pixels underneath them. If your sprite is a square full of 1s, and you draw it on a black background, you will see a white square on a black screen.

If you OVERWRITE a sprite, 1s will be black and 0s will be white. This, of course, messes up the background, but there are times that you might want to do that.

If you AND a sprite, 0s will be drawn as white, and 1s will be transparent. This is very useful if you need a white sprite on a black background.

What if you want a sprite that has black pixels, white pixels, AND transparency? This is where you draw a sprite using the MASK sprite routine. To do this, you must create a picture that is NOT monochrome. Use black ink for black pixels, and white ink for white pixels, as usual. But then, color in green all the parts that you want transparent. If you are looking at the RGB value, the green should be at full blast (FF or 255), with absolutely no red or blue.



Save this picture as a 24-bit bitmap.

Masked sprites take twice as much data as regular sprites. Don't use them lightly.

Now to introduce you to the routine. This sprite routine allows you to draw sprites up to 64 x 64 in size, either XOR, OR or Masked. You can place the sprite anywhere on the screen, and even "off" of the screen if you need to. (This technique is called clipping. If you ever need to draw your sprite so that it is partially off of the screen, you need clipping.)

Be aware, however, that this sprite routine is not for processor-intensive games if you need a bunch of little sprites. Most games for the Ti-83+ use 8x8 sprites, and there are plenty of those on ticalc.org. This routine is meant to get you started as a general-purpose routine, but if you are serious about fast games such as Mario and Kirby, you won't get very far using just this routine, especially since it's HUGE (almost 600 bytes!)

With that said, it takes some work to incorporate this routine into your ASM program or application. Also, if you are creating an application, you will not be able to compute statistics in your program with this routine in place.

To use this sprite routine, make sure that spriteroutines.asm is in the same folder as all your programming stuff, including spasm and the ASM program itself. Then include inside of your program all the sprites you need, each with a label and an #include "bitmap.bmp" statement, where bitmap is the name of your bitmap file. Remember, before each sprite, you should include

```
.option bm_min_w =
```

with the number of pixels wide the sprite is, UNLESS you have a whole bunch of sprites, one after the other, that are the same size. If, for example, you have a bunch of sprites 8 pixels wide by 8 pixels high, you only need to type .option bm_min_w = 8 once, until you reach a sprite of a different width.

You need a label before the start of your program. Let's call it Routine. After .db t2ByteTok, tAsmCmp, you need the following code:

```
jp Routine
#include "spriteroutines.asm"
Routine:
```

Also, type the following after Routine if you are writing an application:

```
LD HL, RamCodeStart
LD DE, statvars
LD BC, EndRamCode-RamCodeStart
LDIR
```

Now, to display your sprite, IX must be equal to the label pertaining to your sprite. Register B must hold the value of the height of your sprite. For register C, take the width of your sprite, divide it by eight, and round UP to the nearest whole number. (This converts your width in pixels to width in bytes.) Finally, register D contains the X

coordinate of where you want to draw the sprite, and register E contains the Y coordinate of where you want to draw the sprite.

Use CALL LargeClippedSpriteOr to draw a sprite using OR. Use CALL LargeClippedSpriteXor to draw a sprite using XOR.

LargeClippedMaskedSprite will draw a Masked Sprite, but you need to use IY to hold the Mask. Remember in Lesson 13 when I said that you have to be careful when using IY? Before you use LD IY, Mask, you need to type in CALL Use_IY_Safely. Then use CALL Return_IY_To_Normal after you draw your sprite.

One more thing about LargeClippedMaskedSprite: you can't use a label for IY in this case. Take the value you gave register B, and multiply it by the value you gave register C. (DO NOT use B * C in your program, because that will not work. Instead, use the values you placed in the registers.) This value is, essentially, the width in bytes of the sprite times the height. Then add this value to the label you assigned to IX.

For example, let's say you have a sprite 32 x 32. So if you are doing everything right, register B contains the value 32, and register C contains the value 4. Suppose your sprite is at label Sprite.

```
LD IY, 32 * 4 + Sprite
```

On the next page is an example, which demonstrates drawing a Sprite as OR, as well as drawing a masked sprite. I have included the two images you need: "Bird.bmp" and "Tree.bmp."

```
#include "ti83plus.inc"
#include “fastcopy.asm”  ;This file has a routine much faster than B_CALL _GrBufCpy.  I recommend
                                ; you use it instead of _GrBufCpy.
.org $9D93
.db    t2ByteTok, tAsmCmp

            jp Routine

#include "spriteroutines.asm"

Routine:
            B_CALL _ClrLCDFull

            ld ix, Tree
            ld c, 3

            ld b, 43
            ld e, 20
            ld d, 71

            call LargeClippedSpriteOr
            call fastcpy
            B_CALL _getKey

            CALL Use_IY_Safely
            ld ix, Bird
            ld iy, Bird + 128
            ld c, 4
            ld b, 32
            ld e, 1
            ld d, 64

            call LargeClippedMaskedSprite
            CALL Return_IY_To_Normal

            B_CALL _GrBufCpy
            B_CALL _getKey
            B_CALL _ClrLCDFull
            ret

.option BM_SHD = 2
.option bm_min_w = 24
Tree:
#include "Tree.bmp"

.option bm_min_w = 32
.option bm_msk = TRUE
.option BM_MSK_RGB = $00FF00
Bird:
#include "Bird.bmp"
```