Learn Multi platform 6502 Assembly Programming... For Monsters!

Super Simple Series

In this series we'll take a look at a variety of basic tasks that you may need to look at when you're starting out with programming these classic systems...

In each case the source-code will be a single file with no Includes to make things easy if you're using a different assembler (though some may include a binary.)

Lesson S1 - Bitmap Drawing on the BBC

Let's look at the basics, Lets learn how to draw a simple bitmap onto the screen... effectively a software sprite.

We'll be able to use this to make a simple game!

BBC_Bitmap.asm

Starting our program

When our program starts, the BBC may be making a noise!... we need to turn off the sound chip - we need to send some commands to $FE40-3 to do this.

```
lda 255     ;Set all bits to write
sta $FE43 ; Data direction port

;     1CCOVVVV = CC=channel O=operation (1=volume) V=Value (Volume 15=off)
lda #%10011111  ;Turn off channel 0
sta $FE41

    ; ----BAAA   =A=address (0=sound chip, 3=Keyboard) B=new setting for address AAA
lda #%00001000      ;Send data to Sound Chip
sta $FE40
lda #%00000000      ;Stop sending data to sound chip
sta $FE40
```

Next we're going to move our running code... Our code Origin is $0200 - but it actually loads at $3000

$3000 is good for basic - but bad for our graphics screen, so we're going to copy our program to $0200 then execute it.

it's important there are no JSR or JMP commands before this routine completes - we do have a BCS, but this is a relative command which 'relocates fine' - don't worry if you don't know what this means yet!

```
    lda #$30    ;Source H $3000
    sta z_h

    lda #>(BBCLastByte-BBCFirstByte+256)
    sta z_b     ;Byte count H

    lda #>RunLocation
    sta z_d     ;Destination H $0200

    ldy #0      ;Low byte of address
    sty z_l
    sty z_e

BBCLDIR:
    lda (z_HL),Y
    sta (z_DE),Y
    iny
    BNE BBCLDIR_SkipInc1
    INC z_H ;Inc Ybytes of address
    INC z_D
    DEC z_B
    BEQ BBCLDIR_Done
BBCLDIR_SkipInc1:
    sec ;Relative jump (JR)
    bcs BBCLDIR ;this program code is relocated
BBCLDIR_Done:
;Jump to the new address in copied code ($0200)
    jmp start
start:
```

We're going to need to initialize the screen

We need to configure the CRTC which defines the screen shape and memory address.

We also need to configure the ULA which configures the way the colors are defined.

We have some arrays of data which we'll use to feed the correct settings for the screen layout we want.

Our screen is finally set up... Phew!

```
CRTCConfig:
    db $7F      ;0 - Horizontal total
    db $50      ;1 - Horizontal displayed characters
    db $62      ;2 - Horizontal sync position
    db $28      ;3 - Horizontal sync width/Vertical sync time
    db $26      ;4 - Vertical total
    db $00      ;5 - Vertical total adjust
    db $18      ;6 - Vertical displayed characters (25)
    db $22      ;7 - Vertical sync position
    db $01      ;8 - Interlace/Display delay/Cursor delay
    db $07      ;9 - Scan lines per character
    db %00110000;10 - Cursor start line and blink type
    db $0       ;11 - Cursor end line
    db $08      ;12 - Screen start address H (Address /8)
    db $30      ;13 - Screen start address L ($4130/8=$0830)
    db 0
    db 0

ULAConfig:
Palette0:   ;Colours
;       SC  SC      -   S=Screen C=Color
    db $07,$17  ;0
    db $47,$57  ;0
Palette1:
    db $22,$32      ;1
    db $62,$72      ;1
Palette2:
    db $81,$91          ;2
    db $C1,$D1          ;2
Palette3:
    db $A0,$B0              ;3
    db $E0,$F0              ;3
```

```
CRTCConfig:
    db $7F      ;0 - Horizontal total
    db $50      ;1 - Horizontal displayed characters
    db $62      ;2 - Horizontal sync position
    db $28      ;3 - Horizontal sync width/Vertical sync time
    db $26      ;4 - Vertical total
    db $00      ;5 - Vertical total adjust
    db $18      ;6 - Vertical displayed characters (25)
    db $22      ;7 - Vertical sync position
    db $01      ;8 - Interlace/Display delay/Cursor delay
    db $07      ;9 - Scan lines per character
    db %00110000;10 - Cursor start line and blink type
    db $0       ;11 - Cursor end line
    db $08      ;12 - Screen start address H (Address /8)
    db $30      ;13 - Screen start address L ($4130/8=$0830)
    db 0
    db 0

ULAConfig:
Palette0:   ;Colours
;       SC  SC      -   S=Screen C=Color
    db $07,$17  ;0
    db $47,$57  ;0
Palette1:
    db $22,$32      ;1
    db $62,$72      ;1
Palette2:
    db $81,$91          ;2
    db $C1,$D1          ;2
Palette3:
    db $A0,$B0              ;3
    db $E0,$F0              ;3
```

Drawing an 8x8 sprite

We're going to show a 8x8 smiley face to the screen... we've set up the screen in 4 color mode - meaning each pixel uses 2 bits...

The screen layout of the BBC is a bit annoying!... the first 8 bytes go down the screen... then the 9th bit jumps back up to the top! - this is represented in the pixel data of our bitmap.

```
Bitmap:
        ;   11110000
;Vline 1 (left)
        DB %00000011      ;  0
        DB %00000111      ;  1
        DB %00101111      ;  2
        DB %00001111      ;  3
        DB %00001111      ;  4
        DB %00101101      ;  5
        DB %00010110      ;  6
        DB %00000011      ;  7
;Vline 2 (Right)
        DB %00001100      ;  8
        DB %00001110      ;  9
        DB %01001111      ; 10
        DB %00001111      ; 11
        DB %00001111      ; 12
        DB %01001011      ; 13
        DB %10000110      ; 14
        DB %00001100      ; 15
BitmapEnd:
```

When we want to show the bitmap to the screen, we need to calculate the destination memory address for the screen...

We use GetScreenPos... which takes an XY position, and calculates a destination memory address in zero page entries z_de (a 16 bit pair)

We then copy the 16 bytes of the 8x8 smiley to the screen memory.

```
    lda #<Bitmap     ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

    ldx #10          ;Xpos
    ldy #10          ;Ypos
    jsr GetScreenPos;Get screen pos from XY into Z_DE

    ldY #0           ;Offset for bytes in this strip
BitmapNextByte:
    lda (z_hl),Y     ;Load in a byte from source - offset with Y
    sta (z_de),Y     ;Store it in screen ram - offset with Y

    inY              ;INC the offset
    cpY #8*2         ;We draw 8 lines * bitmap width
    bne BitmapNextByte

    jmp *
```

The GetScreenPos calculation is pretty tricky!

Our screen address starts at $4180...

Because of the screen layout (8 Y lines are consecutive in ram), we're only going to look at the top 5 bits of the Y position
Each line is 320 pixels, and there are 4 pixels per byte... this means our screen is 80 bytes wide, and because the 8 lines of each horizontal strip are combined, the top 5 bits must be multiplied by 640 ($280) (80*8=640)

Since the 8 Y lines are consecutive in memory, between each horizontal byte, we multiply the Xpos by 8

Our formula is:
$4180 + ({Top 5 bits of Y} * $280) + Xpos * 8

We achieve the multiplication by bitshifting ops.

```
GetScreenPos:
    lda z_b
    pha
    lda z_c
    pha
        lda #0
        sta z_d

        txa         ;Xpos
        asl
        rol z_d     ;2
        asl
        rol z_d     ;4
        asl
        rol z_d     ;8      ;8 bytes per X line
        sta z_e

        ;We have to work in 8 pixel tall strips on the BBC
        tya         ;Ypos
        and #%11111000
        lsr         ;$04 00
        lsr         ;$02 00
        sta z_b     ;Multiply Y strip num by $02
        clc
        adc z_d     ;Add to D
        sta z_d
        lda #0
        ror z_b     ;$01 00
        ror
        ror z_b     ;$00 80
        ror
        adc z_e     ;Add to E
        sta z_e
        lda z_b     ;Add to D
        adc z_d
        sta z_d

        lda z_e
        adc #$80    ;Screen Offset $4180
        sta z_e
        lda z_d
        adc #$41    ;Screen Offset $4180
        sta z_d
    pla
    sta z_c
    pla
    sta z_b
    rts
```

The result can be seen here!

The example code above only works in 8x8 strips, you can make it draw to XY position 4,8 or 8,16... but it can't do 8,4... the Y-pos must be divisible by 8 if it's not, it will be rounded down.

Making a version which does not have this limitation would be more tricky.

Drawing a larger bitmap

Drawing a small bitmap is useful, but what if we want to use a bigger sprite... well we'll have to cope with the strange layout of the screen, but it's still pretty easy.
We're going to include a bitmap file to show

```
Bitmap:
    incbin "\ResALL\Sprites\RawBBC.RAW"
BitmapEnd:
```

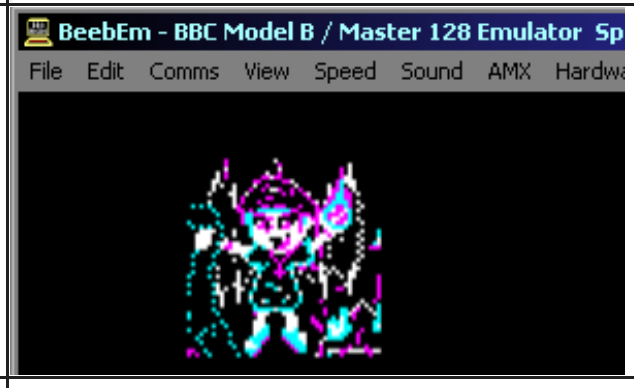
We now need an extra loop for the number of '8 pixel Y strips'...

After we've done each strip, we reset our start position, and add $280 (the bytes used by one 8x320 pixel Y-strip)

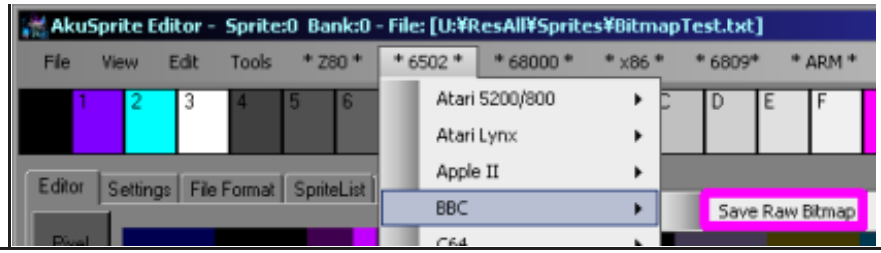
We repeat until the image has been drawn.

```
    lda #<Bitmap          ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

    ldx #10
    ldy #10
    jsr GetScreenPos      ;Get screen pos from XY into Z_DE

    ldx #6                ;Height in strips
NexBitmapNextStrip:
    lda z_d
    pha
    lda z_e
    pha
BitmapNextLine:
        ldY #0            ;Offset for bytes in this strip
BitmapNextByte:
        lda (z_hl),Y      ;Load in a byte from source - offset with Y
        sta (z_de),Y      ;Store it in screen ram - offset with Y

        inY               ;INC the offset
        cpY #12 *8        ;We draw 8 lines * bitmap width (12 bytes)
        bne BitmapNextByte
        clc
        tya               ;Add Y to HL
        adc z_l
        sta z_l
        lda z_h
        adc #0
        sta z_h
    pla ;z_e
    clc
    adc #$80     ;Move down one strip (+$0280)
    sta z_e
    pla ;z_d
    adc #$02
    sta z_d
    dex          ;NO of Y-strips in Bitmap 8 rows per strip
    bne NexBitmapNextStrip
```

Our Chibiko Character will be shown on the screen.



My AkuSprite Editor can export files in the correct format for this tutorial - it's free and open source.



Lesson S2 - Bitmap Drawing on the C64

Lets take a look at the C64 this time - and get either a 2 or 4 color image on the screen!

We'll look at a simple 8x8 smiley, and something bigger!

C64_Bitmap.asm

Starting our program

The start of our program needs an INIT routine for the PRG file,

Next we need to initialize the screen, we need to set the base address, and color depth

We also need to define the background and border color...

In these tutorials, we'll use an optional symbol 'FourColor' to enable 4 color mode - otherwise the example will be 2 colors

```
;Init Routine
*=$0801
    db $0E,$08,$0A,$00,$9E,$20,$28,$32,$30,$36,$34,$29,$00,$00,$00
*=$0810 ;Start at $0810

;     LXMSHVVV - L=Cur Line X=extended BG M=mode
                ;(Txt/Bmp) S=screen on H=height V=Vert scroll
    lda #%00111011  ;turn on graphics mode
    sta $D011

;     ---MWHHH - M=Multicolor W=scr width H=horiz scroll
    ifdef FourColor
        lda #%11011000  ;1=Multicolor 4 coor
    else
        lda #%11001000  ;0=standard 2 color
    endif
    sta $D016

;     SSSSTTT- - T=Text/Bmp screen address S=Screen (color) address
    lda #%00011000  ;T=1 Screen at $2000
    sta $D018           ;(Other bits have no function in bitmap mode)

;     ----CCCC
    lda #%00000000
    sta $D021       ;Background color (only bits #0-#3).
    sta $D020       ;Border
```

Drawing an 8x8 sprite

We're going to draw an 8x8 sprite to screen in 2 or 4 color.

Actually it'll be half the width on the 4 color screen... but who's counting!

```
Bitmap:
    ifdef FourColor
        DB %00010100      ;  0
        DB %00010101      ;  1
        DB %01110101      ;  2
        DB %01010101      ;  3
        DB %01010101      ;  4
        DB %01100101      ;  5
        DB %00011001      ;  6
        DB %00010100      ;  7
    else
        DB %00111100      ;  0
        DB %01111110      ;  1
        DB %11011011      ;  2
        DB %11111111      ;  3
        DB %11111111      ;  4
        DB %11011011      ;  5
        DB %01100110      ;  6
        DB %00111100      ;  7
    endif
BitmapEnd:
```

We're going to draw our image.

We use 'GetScreenPos' to calculate a memory address of a screen position, it takes XY, and returns a zeropage pair z_de for the destination.
We then need to copy the byte data.

We also need to set the color attributes... we use GetColMemPos...
In the case of the two color screen, we write one color byte to z_de... for four color screens we need two bytes (3 nibbles) to z_bc and z_de... the background color is constant and cannot be set per tile.

```
;Draw Bitmap Data
    lda #<Bitmap        ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

    ldx #4              ;Xpos
    ldy #8              ;Ypos
    jsr GetScreenPos    ;Get screen pos from XY into Z_DE
BitmapNextLine:
    ldY #0              ;Offset for bytes in this strip
BitmapNextByte:
    lda (z_hl),Y        ;Load in a byte from source - offset with Y
    sta (z_de),Y        ;Store it in screen ram - offset with Y
    inY                 ;INC the offset
    cpY #8              ;We draw 8 lines * bitmap width
    bne BitmapNextByte

;Fill Color Data
    ldx #4              ;Xpos
    ldy #8              ;Ypos
    jsr GetColMemPos    ;Get color pos from XY into Z_DE

    ldy #0
    ifdef FourColor
        lda #$43        ;Color
        sta (z_de),Y    ;%22221111 Color 1,2
        lda #01
        sta (z_bc),Y    ;%----3333 Color 3 (White)
    else
        lda #$40        ;Color
        sta (z_de),Y    ;%11110000 Color 1,2
    endif
```

The GetScreenPos calculation is pretty tricky!

Our screen address starts at $2000...

Because of the screen layout (8 Y lines are consecutive in ram), we're only going to look at the top 5 bits of the Y position
Each line is 320 pixels, and there are 8 pixels per byte (160 pix and 4 px per byte in 4 color mode)... this means our screen is 40 bytes wide, and because the 8 lines of each horizontal strip are combined, the top 5 bits must be multiplied by 640 ($140) (40*8=320)

Since the 8 Y lines are consecutive in memory, between each horizontal byte, we multiply the Xpos by 8

Our formula is:
$2000 + (Y * 40) + Xpos * 8

We achieve the multiplication by bitshifting ops.

```
;Address= (X * 8) + (Top5BitsOfY * 40) + $2000
GetScreenPos:
    lda #0
    sta z_b
    sta z_d
    txa                 ;Multiple X by 8
    asl                 ;-------- XXXXXXXX
    rol z_d
    asl
    rol z_d
    asl
    rol z_d             ;-----XXX XXXXX---
    sta z_e

;40 bytes per Yline =00000000 00101000
    tya
    and #%11111000      ;00000000 YYYYYyyy
    asl
    rol z_b
    asl
    rol z_b
    asl                 ;00000000 00101000
    rol z_b             ;00000YYY YYyyy000
    tax
        adc z_e         ;Add part to total L
        sta z_e
        lda z_b         ;Add part to total H
        adc z_d
        sta z_d
    txa
    asl
    rol z_b
    asl                 ;00000000 00101000
    rol z_b             ;000YYYYY yyy00000

    adc z_e             ;Add part to total L
    sta z_e
    lda z_b             ;Add part to total H
    adc z_d
    adc #$20+0          ;Screen Base $2000
    sta z_d
    rts
```

Each 8x8 square has it's own color attributes... these are stored at two memory addresses $D800+ and $0400+

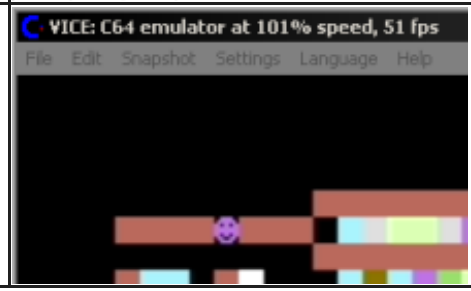
There is one byte per 8x8 block at each of these addresses... we calculate these using formulas below:

Address1= $0400 + (Ypos * 40) + Xpos
Address2= $D800 + (Ypos * 40) + Xpos

We calculate these in pretty much the same way as before.

```
GetColMemPos:
    lda #0
    sta z_d
    txa
    sta z_e          ;Xpos
;40 bytes per Yline =00000000 00101000
    tya              ;Need to multiply by 40 (%00101000)
    and #%11111000   ;One color per 8x8 square
    tay
        clc
        adc z_e      ;Add Ypos part to Xpos
        sta z_e      ;Save %00-01000 part
    tya
    asl
    rol z_d
    asl              ;00000000 00101000
    rol z_d          ;000YYYYY yyy00000
    clc
    adc z_e          ;Add Ypos part to total
    sta z_e
    sta z_c

    lda z_d
    adc #$04+0       ;Color Offset $0400
    sta z_d

    adc #$D8-4       ;Color Offset $D800
    sta z_b
    rts              ;z_DE = $0400-07FF byte
                   ;z_BC = $D800-DBFF byte (for 4 color)
```

Our Smiley will be show onscreen



The C64 screen is rather a pain! 8 Consecutive bytes go down the screen, and the 9th jumps back up!... this 'zigzag' layout probably makes font drawing easier, but it's not very friendly for graphics! The example above will only work on an 8x8 grid.

Drawing a larger bitmap

We're going to include a bitmap file, this will be shown to the screen.

```
Bitmap:
    ifdef FourColor
        incbin "\ResALL\Sprites\RawC64-4col.RAW"
    else
        incbin "\ResALL\Sprites\RawC64-2col.RAW"
    endif
BitmapEnd:
```

We now need an extra loop for the number of '8 pixel Y strips'...

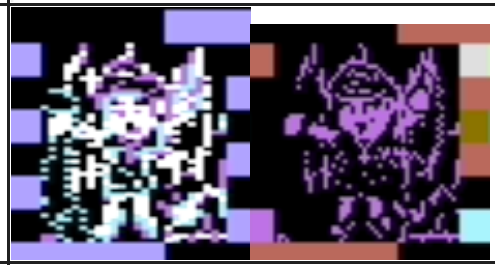
After we've done each strip, we reset our start position, and add $140 (the bytes used by one 8x320 pixel Y-strip)

We repeat until the image has been drawn. we then need to do the same for the color data.

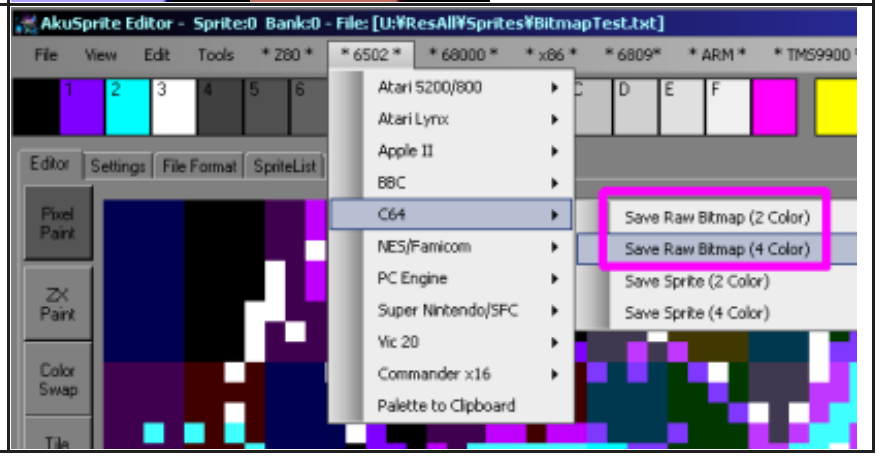
```
;Draw Bitmap Data
    lda #<Bitmap                    ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

bmpwidth equ 6
BmpHeight equ 6
xpos equ 4
ypos equ 8

    ldx #xpos
    ldy #ypos
    jsr GetScreenPos    ;Get screen pos from XY into Z_DE

    ldx #bmpwidth
NexBitmapNextStrip:
    lda z_d
    pha
    lda z_e
    pha
BitmapNextLine:
        ldY #0          ;Offset for bytes in this strip
BitmapNextByte:
        lda (z_hl),Y    ;Load in a byte from source - offset with Y
        sta (z_de),Y    ;Store it in screen ram - offset with Y

        inY             ;INC the offset
        cpY #bmpwidth *8        ;We draw 8 lines * bitmap width (12 bytes)
        bne BitmapNextByte
        clc
        tya             ;Add Y to HL
        adc z_l
        sta z_l
        lda z_h
        adc #0
        sta z_h
    pla ;z_e
    clc
    adc #$40    ;Move down one strip (+$0140)
    sta z_e
    pla ;z_d
    adc #$01
    sta z_d
    dex         ;NO of Y-strips in Bitmap 8 rows per strip
    bne NexBitmapNextStrip

;Fill Color Data
    ldx #xpos
    ldy #ypos
    jsr GetColMemPos

    ldx #BmpHeight
```

```
FillColMemAgainY:
    lda z_d
    pha
    lda z_e
    pha
        ldy #0
FillColMemAgainX:
        ifdef FourColor
            lda #$43                ;Color
            sta (z_de),Y            ;%22221111 Color 1,2
            lda #01
            sta (z_bc),Y            ;%----3333 Color 3 (White)
        else
            lda #$40                ;Color
            sta (z_de),Y            ;%11110000 Color 1,2
        endif
        iny
        cpy #bmpwidth
        bne FillColMemAgainX
    pla ;z_e
    clc
    adc #$28    ;Move down one strip (+40)
    sta z_e
    sta z_c
    pla ;z_d
    adc #0      ;Color offset $0400
    sta z_d
    adc #$D8-4  ;Color Offset $D800
    sta z_b
    dex         ;NO of Y-strips in Bitmap 8 rows per strip
    bne FillColMemAgainY
```

The sprite will be shown onscreen



You can use my AkuSprite Editor to export data in the correct format for this tutorial, it's free and open source (and included in the sources.7z



Lesson S3 - Bitmap Drawing on the VIC-20

The Vic 20 can't do 'bitmap graphics' it's more like a Tilemap... we can define custom characters, then show those onto the screen to draw our Graphics.

Lets give it a go!

VIC_Bitmap.asm

Starting our program

To start our program we need a PRG header first...

We then need to define the graphics screen settings, we do this by writing a bank of settings to address $9000+

This will set our screen up, we can now start our program!

```
* = $1001
        ; BASIC program to boot the machine language code
        db $0b, $10, $0a, $00, $9e, $34, $31, $30, $39, $00, $00, $00
;start of code $100A

;Screen Init
    ldx #16                 ;We're going to copy 16 registers
ScreenInitAgain:
    dex
    lda VicScreenSettings,x ;Get A parameter
    sta $9000,X             ;Store to the video registers at $9000
    txa
    bne ScreenInitAgain


VicScreenSettings:
    db $0C      ;$9000 - horizontal centering
    db $26      ;$9001 - vertical centering
    db $96      ;$9002 - set # of columns /
                    ;Bit7 = screen base bit ($16 for screen at $1000)
    db $AE      ;$9003 - set # of rows
    db $7A      ;$9004 - TV raster beam line
    db $FF      ;$9005 - bits 0-3 start of character memory /
                    ;bits 4-7 is rest of video address
                    ;$(CF for screen at $1000)
    db $57      ;$9006 - horizontal position of light pen
    db $EA      ;$9007 - vertical position of light pen
    db $FF      ;$9008 - Digitized value of paddle X
    db $FF      ;$9009 - Digitized value of paddle Y
    db $00      ;$900A - Frequency for oscillator 1 (low)
    db $00      ;$900B - Frequency for oscillator 2 (medium)
    db $00      ;$900C - Frequency for oscillator 3 (high)
    db $00      ;$900D - Frequency of noise source
    db $00      ;$900E - bit 0-3 sets volume of all sound /
                    ;bits 4-7 are auxiliary color information
    db $00+8    ;$900F - Screen and border color register
```

Drawing an 8x8 sprite

We're storing our bitmap data in memory... it's one bit per pixel, and the characters are 8x8

```
Bitmap:
        DB %00111100      ;  0
        DB %01111110      ;  1
        DB %11011011      ;  2
        DB %11111111      ;  3
        DB %11111111      ;  4
        DB %11011011      ;  5
        DB %01100110      ;  6
        DB %00111100      ;  7
BitmapEnd:
```

Our first task will be to copy the Bitmap data into the definable characters, we use a 'define tiles' function to do this

We're using Zero page pairs z_HL as a source, z_DE as a destination and z_BC as a byte count

Our destination is memory address $1C00 - Character 0 in the configurable characters.

```
  lda #<Bitmap                    ;Source Bitmap Data
  sta z_L
  lda #>Bitmap
  sta z_H

  lda #<(BitmapEnd-Bitmap);Source Bitmap Data Length
  sta z_C
  lda #>(BitmapEnd-Bitmap)
  sta z_B

  lda #<$1C00     ;Tile 0 in VIC Custom Characters
  sta z_E
  lda #>$1C00
  sta z_D

  jsr DefineTiles     ;Define the tile patterns
```

the 'Define Tiles' routine is essentially a 'LDIR' copy routine, it will copy from the source address to a destination.

```
DefineTiles:
        ldy #0
DefineTiles2:
        lda (z_HL),Y          ;Copy From z_HL
        sta (z_DE),Y          ;To z_ DE
        iny                   ;Inc Byte
        BNE DefineTiles_SkipInc1
        INC z_H               ;Inc H bytes
        INC z_D
DefineTiles_SkipInc1:
        DEC z_C               ;Dec Counter
        BNE DefineTiles2
        LDA z_B
        BEQ DefineTiles_Done
        DEC z_B
        jmp DefineTiles2
DefineTiles_Done:
        rts
```

When it comes to showing the character, we need to calculate the memory address of the XY pos, and then det that character (0) onscreen.

Once we've set the character, we also want to color it... the address of the color for this tile is a +$7800 offset from the character number, so we just add $78

```
;Draw Bitmap
    lda #3              ;Start SX
    sta z_b
    lda #3              ;Start SY
    sta z_c
    jsr GetVDPScreenPos

    lda #0              ;Tile 0 (smiley)
    sta (z_hl),y        ;Transfer Tile to ram
    lda z_h
    clc
    adc #$78            ;add Offset to Color Ram ($9600)
    sta z_h
    lda #4              ;Color
    sta (z_hl),y        ;Set Tile Color

    jmp *
```

When we want to calculate the memory address our formula is:
Address= $1E00 + (Ypos * 22) + Xpos

As multiplying by 22 isn't so easy (and our screen is pretty short) we'll use a loop and addition to effect the multiply

```
GetVDPScreenPos:    ; BC=XYpos
    lda #$1e            ;Screen base is $1E00
    sta z_h                 ;Colors at $9600 (add $7800 offset)

    lda z_b             ;Xpos
    sta z_l

    ldy z_c             ;Ypos
    beq GetVDPScreenPos_YZero
GetVDPScreenPos_Addagain:   ;Repeatedly add screen width (22) Y times
    clc
    lda z_l
    adc #22         ;22 bytes per line
    sta z_l
    lda z_h
    adc #0          ;Add Carry
    sta z_h

    dey
    bne GetVDPScreenPos_Addagain
GetVDPScreenPos_YZero:
    rts
```

Using the redefinable characters will 'offset' the proper letters...

If you want to show an 'A' in ths mode, you'll want to add 64... so use LDA #'A'+64

Drawing a larger bitmap

If we want to draw a larger bitmap, we'll have to split it into characters... We can load those characters in with the same Define Tiles function.

Our test character is 6x6 characters (48 x 48 pixels)

```
Bitmap:
    incbin "\ResAll\Sprites\RawVIC.raw"
BitmapEnd:
```

When it comes time to get the graphic on the screen, we'll need to draw the characters into the area to fill in order - the 'Fill Area with Tiles' function will do this for us!

```
lda #4
sta z_d             ;Color

lda #3              ;Start SX
sta z_b

lda #3              ;Start SY
sta z_c

ldx #6              ;Width in tiles
ldy #6              ;Height in tiles
lda #0              ;TileStart
jsr FillAreaWithTiles   ;Draw the tiles to screen
```

The FillAreaWithTiles function is long, but the concept is pretty simple...

We load in tile numbers from z_E... and set the screen characters to those tile numbers, incrementing z_E each time.

When we get to the end of a line, we move back to the start of the line, and add #22, to move down the screen a line.

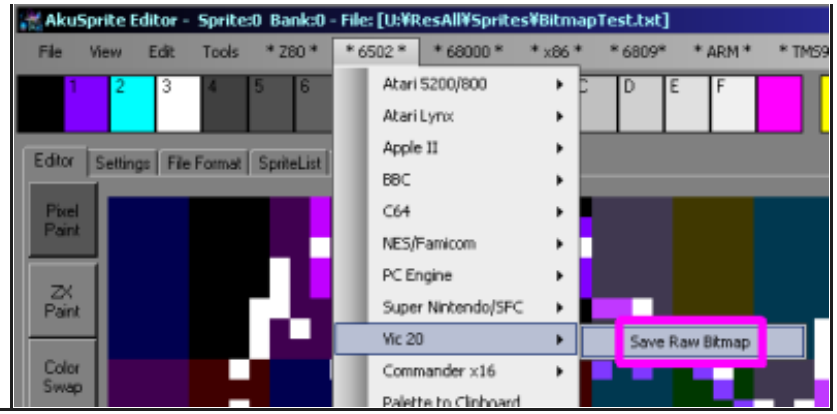
We repeat this procedure until our image is finished.

```
FillAreaWithTiles:
    sta z_e          ;Backup Tile number
    tya
    pha
                     ;Calculate screen ram location of tile
    jsr GetVDPScreenPos
    pla
    tay
FillAreaWithTiles_Yagain:
    tya
    pha
    txa
    pha
        lda z_e          ;Tilenum
        ldy #0
FillAreaWithTiles_Xagain:
        sta (z_hl),y     ;Transfer Tile to ram
        pha
            lda z_h      ;Back up z_H
            pha
                clc
                adc #$78     ;Offset to Color Ram
                sta z_h
                lda z_d
                sta (z_hl),y ;Set Tile Color
            pla
            sta z_h      ;Restore z_H
            iny
        pla
        clc
        adc #1           ;Increase tile number
        dex              ;Decrease X counter
        bne FillAreaWithTiles_Xagain
        sta z_e          ;Back up Tilenum for next loop
        inc z_c
    pla
    tax
    pla
    tay
    lda z_l
    adc #22          ;Move Down (22 bytes per line)
    sta z_l
    lda z_h
    adc #0           ;Add Carry
    sta z_h
    dey              ;Decrease Y counter
    bne FillAreaWithTiles_Yagain
    rts
```

And here's our Chibiko character!

To convert a bitmap to the correct format, you'll need to make it black and white, and split it up into 8x8 chunks...

You can export a bitmap in the correct format for todays tutorial with my AkuSprite editor... it's free and open source, and included in the sources file.



Lesson S4 - Bitmap Drawing on the Atari 800 / 5200

To Draw our bitmap to the screen on the Atari, we need a 'Display list'... this defines the line of the screen, we can use this to specify a memory address as the visible screen, then use that screen to display our bitmap... Lets make it happen!

A52_Bitmap.asm

Starting our program

At the start of our program, we'll first start by resetting the Zeropage and GTIA (Graphics hardware)

Once we're ready, we load the address of the Display List - this defines the screen settings, and turn on the display DMA

```
ProgramStart:
    sei             ;Disable interrupts

    ldx #$00        ;Zero GTIA area and Zero Page
    txa
ClearLoop
    sta $00,x       ;Clear zero page
    sta GTIA,x      ;Clear GTIA
    dex
    bne ClearLoop

    lda #<DisplayList
    sta $D402       ;DLISTL - Display list lo
    lda #>DisplayList
    sta $D403       ;DLISTH - Display list hi

    lda #%00100010
    sta $D400       ;DMACTL - DMA Control (screen on)
```

The Display list defines the screen layout... we need one byte per line for the bitmap modes, and because of limitations of the hardware we have to manually define an offset when the VRAM reaches address

$3000

The end of the list contains a loop back to the start ... this defines the screen...

if we set Smode to $0E we will have a 4 color screen... if we se Smode to $0F we will have a 2 color high res screen.

```
DisplayList:                ;Display list data
    db $70,$70,$70;$70 7= 8 blank lines 0= blank lines

      db $40+Smode,$60,$20    ;Strange start ($2060) to safely step over the boundary

      db            Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode

      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode

      db $40+Smode,$00,$30    ;Have to manually step over the 4k boundary ($3000)
      db            Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode

      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode
      db      Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode,Smode

    db $41                  ;Loop
    dw DisplayList
```

We now want to define some colors... on the 2 color machine the foreground and background have to be the same color (but different brightness')

We're now ready to start our program.

```
ifdef FourColor
    lda #$98        ;Set color PF1 (foreground) (CYAN)
    sta GTIA+ $17   ;COLPF1 equ

    lda #$0F        ;Set color PF2 (background) (White)
    sta GTIA+ $18   ;COLPF2

    lda #$68        ;Set color PF0 (Purple)
    sta GTIA+ $16

    lda #$00       ;Set color PF0 (Black)
    sta GTIA+ $1A
else
    lda #$0F
    sta GTIA+ $17   ;COLPF1 equ

    lda #$00        ;2 color mode only uses the brightness of color1
    sta  GTIA+ $16
    sta  GTIA+ $18
    sta  GTIA+ $1A
endif
```

The Displaylist we've defined here is pretty boring··· all the lines are the same mode! We can define the lines to be different screen modes··· having part of the screen graphics, and part text··· if this is what we need we can save some memory!

Drawing an 8x8 sprite

We're going to draw an 8x8 smiley onto the screen.

Depending on our Colordepth we'll want a 1bpp (1 bit per pixel) or 2bpp image

```
Bitmap:
    ifdef FourColor
        DB %00010100      ; 0
        DB %01010101      ; 1
        DB %01110111      ; 2
        DB %01010101      ; 3
        DB %01010101      ; 4
        DB %01100110      ; 5
        DB %01011001      ; 6
        DB %00010100      ; 7
    else
        DB %00111100      ; 0
        DB %01111110      ; 1
        DB %11011011      ; 2
        DB %11111111      ; 3
        DB %11111111      ; 4
        DB %11011011      ; 5
        DB %01100110      ; 6
        DB %00111100      ; 7
    endif
BitmapEnd:
```

When we want to draw lines of our smiley we first load in the address of the byte data into zero page entries z_hl

We need to calculate the destination address... we do this with the function Get ScreenPos... this loads the address of the screen pos XY into z_de

we need to copy one byte per line from z_hl to z_de

When we need to move down a line, we add 40 ($28) to z_de to do so, and repeat until all the lines are done.

```
    lda #<Bitmap     ;Bitmap source
    sta z_L
    lda #>Bitmap
    sta z_H

    ldx #6           ;Xpos
    ldy #6           ;Ypos

    jsr GetScreenPos     ;Calculate Memory address of XY

    ldY #0           ;Line Count
BitmapNextLine:
    ldx #0
    lda (z_hl),Y     ;Copy a byte from the source
    sta (z_de,x)     ;to the destination

    clc
    lda z_e
    adc #$28         ;Move down a line (40 Bytes / $0028)
    sta z_e
    lda z_d
    adc #$00
    sta z_d

    iny
    cpy #8           ;Height (8 bytes)
    bne BitmapNextLine
```

GetScreenPos calculates the screen address from X,Y... our screen base is $2060 - and each line is 40 bytes wide... so our calculation is:

Address= $2060 + (Ypos * 40) + Xpos

Because the 6502 has no multiplication, we do bitshifts to double the value... it's easiest to split (Ypos * 40) into (Ypos * 32) + (Ypos * 8)

We also add the Xpos and $2060 - this calculates the final address in memory...

```
GetScreenPos:
    txa
    sta z_e          ;Store X pos in E

    tya              ;Get Ypos - store in B
    sta z_b          ;%YYYYYYYY 00000000

    lsr z_b          ;Shift 3 Bits
    ror              ;%0YYYYYYY Y0000000
    lsr z_b
    ror              ;%00YYYYYY YY000000
    lsr z_b
    ror              ;%000YYYYY YYY00000 = Y*32
    tax
        clc
        adc z_e      ;Update Low Byte
        sta z_e

        lda #0       ;Update High Byte
        adc z_b
        sta z_d
    txa
    lsr z_b          ;Shift 2 bits
    ror              ;%0000YYYY YYYY0000
    lsr z_b
    ror              ;%00000YYY YYYYY000 = Y*8

    adc z_e          ;Add to Update Low Byte
    sta z_e

    lda z_b          ;Add to Update High Byte
    adc z_d
    sta z_d

    clc
    lda z_e
    adc #$60
    sta z_e
    lda z_d          ;Add Screen Base ($2060)
    adc #$20
    sta z_d

    rts
```

And here's the result... in 4 color or 2 color!

Drawing a larger bitmap

This time we're going to draw a 48x48 sprite... we'll include it from a binary file.

```
Bitmap:
    ifdef FourColor
        incbin "\ResALL\Sprites\RawA52.RAW"
    else
        incbin "\ResALL\Sprites\RawZX.RAW"
    endif
BitmapEnd:
```

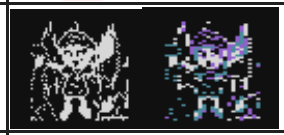
We need to alter our sprite drawing routine with a second inner loop that will draw all the bytes of a line to screen.

Once we've done a line we update the source address by adding Y, and the destination address by adding 40 (one line)

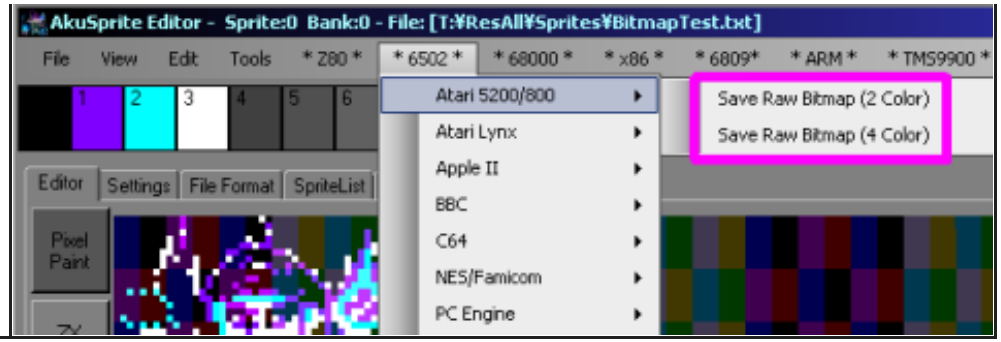
We repeat until the sprite is drawn

```
    ldx #48          ;Height
BitmapNextLine:
    ldY #0           ;Line Byte Count
BitmapNextByte:
    lda (z_hl),Y     ;Copy a byte from the source
    sta (z_de),Y     ;to the destination

    inY
    cpY #6           ;WIDTH in bytes
    bne BitmapNextByte

    clc
    tya              ;Add Y to HL
    adc z_l
    sta z_l
    lda z_h
    adc #0
    sta z_h

    lda z_e
    adc #$28     ;Move down a line (40 Bytes / $0028)
    sta z_e
    lda z_d
    adc #$00
    sta z_d

    dex          ;Next Y Line
    bne BitmapNextLine
```

And here's the result!



You can out put bitmap files in the correct format for this tutorial with AkuSprite editor, it's free and open source (included in the sources.7z)



Lesson S5 - Bitmap Drawing on the Apple II

The Apple II screen is kind of Black and white - and Kind of color!... depending on the combination... lets learn what it all means, and get a smiley onscreen!

AP2_Bitmap.asm

Starting our program

Ok, let's start our program!

We're going to start our program at $0C00 as before...

We now need to do some reads from some memory addresses - it's weird, but reading these addresses turns on these functions... effectively we're turning on the graphics mode

```
ORG $0C00    ;Program Start
sei      ;Disable interrupts

lda $C050 ; TXTCLR:   Display Graphics
lda $C052 ; MIXCLR:   Display Full Screen
lda $c057 ; HIRES:    Display HiRes Graphics
lda $C055 ; TXTPAGE2: If 80STORE Off: Display Page 2
```

Drawing an 8x8 sprite

We're going to draw a smiley face on screen.... the format of screen bytes on the Apple II is weird... the leftmost bit is the 'color palette' and the other 7 are visible pixels.

We'll see what that means onscreen in a moment!

```
Bitmap:
        DB %00011100      ; 0
        DB %00111110      ; 1
        DB %01101011      ; 2
        DB %01111111      ; 3
        DB %01111111      ; 4
        DB %01101011      ; 5
        DB %00110110      ; 6
        DB %00011100      ; 7
BitmapEnd:
```

When we want to get our byte wide bitmap onscreen, we first load it's address into zero-

page pair z_hl.

Next we calculate the memory address (in z_de)... GetScreenPos will do this, using XY as an X,Y pos.

We load in one byte from our source (z_hl) and write it to the screen (z_de)

We do this 8 times - using Y as an offset for the source data... to move down a line we add $0400 to the z_de screen address

```
    lda #<Bitmap     ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

    ldx #6           ;Xpos
    ldy #8           ;Ypos
    jsr GetScreenPos

    ldY #0
BitmapNextLine:
    ldx #0
    lda (z_hl),Y     ;Read byte from source
    sta (z_de,X)     ;Write to screen

    lda z_d          ;move mempos down a line
    clc
    adc #$04         ;add $0400 to the line number
    sta z_d          ;  (only works within an  8 line block)

    iny
    cpy #8
    bne BitmapNextLine  ;Some systems need a recalc every 8 lines
```

Calculating our screen address is rather annoying on the Apple II - the screen is split into 3rds... and each line in 8 is separate... the result is a rather annoying formula:

Yline... AABBBCCC - AA*$0028 BBB*$0080 CCC*$0400
+Xpos
+$4000 (screen base address)

We test the top 2 bits and branch out into sub-code that will do additions to calculate the final address...

```
GetScreenPos:
    lda #0
    sta z_e
    tya                 ;--BBB---   ;Multiply by $0080
    and #%00111000
    lsr
    lsr
    lsr
    lsr                 ;Shift 1 bit right
    ror z_e
    adc #$40            ;Screen base
    sta z_d
    tya                 ;AA------       ;multiply by $0028
    rol                 ;Get 1st A from AA------
    bcc GetScreenPos_SecondThird
GetScreenPos_ThirdThird:
    lda z_e
    clc
    adc #$50            ;3/3 = Add $0050 to address
    jmp GetScreenPos_ThirdDone
GetScreenPos_SecondThird:
    rol                 ;Get 2nd A from AA------
    bcc GetScreenPos_FirstThird
    lda z_e
    clc
    adc #$28            ;3/2 = Add $0028 to address
GetScreenPos_ThirdDone:
    sta z_e
GetScreenPos_FirstThird:;1/3 = Add nothing to addreess
    tya             ;-----CCC   ;Multiply by 4
    and #%00000111
    asl
    asl
    adc z_d
    sta z_d

    txa             ;Process X
    clc
    adc z_e         ;Add X to calculated address
    sta z_e
    rts
```

Here's our smiley!... notice the slight color distortion around the face

Drawing a larger bitmap

If we want to draw a larger bitmap, we need to use a more complex routine...

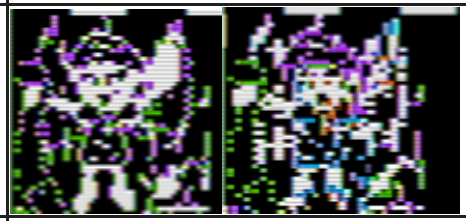
Because the screen is split into so many sections we'll need to use GetScreenPos to recalculate whenever we get outside an 8x8 block...

We'll copy all the bytes for each line... then add $0400 to move down a screen line - this will only work for within an 8x8 block..

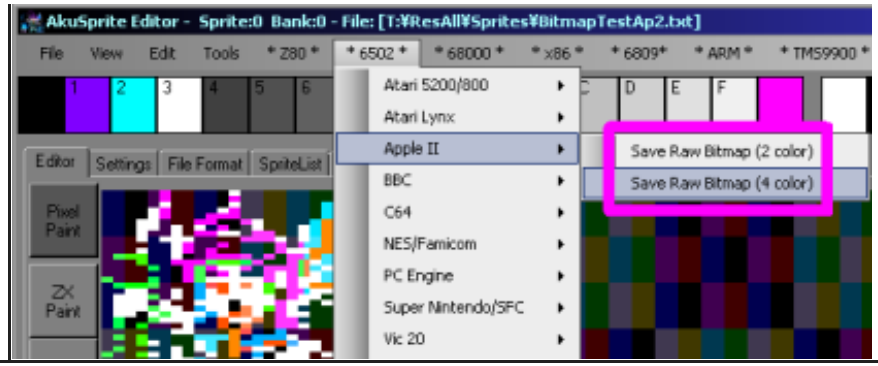
When we're outside the 8x8 block we need to call GetScreenPos to recalculate things properly again.

```
    ldx #6          ;Xpos
    ldy #8          ;Ypos

bmpwidth equ 8
    lda #6          ;HEIGHT 6 Strips of 8 lines = 48 pixels tall
NexBitmapNextStrip:
    pha
    phx
        phy
            jsr GetScreenPos
            ldx #8          ;8 lines per square
BitmapNextLine:
            phy
                ldY #0
BitmapNextByte:
                lda (z_hl),Y ;Read byte from source
                sta (z_de),Y ;Write to screen
                inY
                cpY #bmpwidth
                bne BitmapNextByte

                tya
                clc
                adc z_l     ;ADD Y to Z_HL to move source
                sta z_l
                lda z_h
                adc #0
                sta z_h

                lda z_d     ;move mempos down a line
                clc
                adc #$04    ;add $0400 to the line number
                sta z_d     ;  (only works within an  8 line block)
            ply
            dex
            bne BitmapNextLine  ;Some systems need a recalc every 8 lines
        pla
        clc
        adc #8              ;Move Y down 8 lines
        tay
    plx
    pla
    sec
    Sbc #1
    bne NexBitmapNextStrip
```

Here is the result... We've got two versions... a 2 color version, and a 6 color one... note the code is identical... it's the combination of on and of neighboring pixels that make the 'colors'



You can export basic color sprites with Akusprite Editor - it will attempt to export pixels in the correct layout for the pixels to appear colored onscreen.



The Apple II colors are pretty weird.. but thats the way things worked! The colors are an 'artifact' of the screen display - so even the modern emulators have to 'simulate' the color distortion of the old screen.

Lesson S6 - Bitmap Drawing on the Atari Lynx

With a 16-color standard 'bitmap' screen - the Lynx is easy!... and because the Lynx has a fast CPU and small screen the screen is really fast...

Lets learn how to create a nice smiley onscreen.

LNX_Bitmap.asm

Starting a Lynx Cartridge

Our cartridge needs a header
This will start our program in ram at address $0200

We need to initialize the screen - setting up the base address of our screen ($C000)

We also need to define some colors so we can see the result onscreen.

```
    org $200-10      ;Our program starts at $0200
    db $80,$08,$02,$00,$40,$0A,$42,$53,$39,$33

;ScreenInit -   SUZY chip needs low byte setting first
                ;OR IT WILL WIPE THE HIGH BYTE!

    ;Set screen ram pointer to $C000
    stz $FD94        ;DISPADR    Display Address L (Visible)
    lda #$C0
    sta $FD95        ;DISPADR    Display Address H (Visible)

    ;Do the palette
    stz $FDA0        ;Palette Color 0 ----GGGG (Black)
    stz $FDB0        ;Palette Color 0 BBBBRRRR

    ;lda #%00000000 ;Palette Color 1 ----GGGG (Purple)
    stz $FDA1
    lda #%01110111   ;Palette Color 1 BBBBRRRR
    sta $FDB1

    lda #%00001111   ;Palette Color 2 ----GGGG (Cyan)
    sta $FDA2
    lda #%11110000   ;Palette Color 2 BBBBRRRR
    sta $FDB2

    lda #%00001111   ;Palette Color 3 ----GGGG (White)
    sta $FDA3
    lda #%11111111   ;Palette Color 3 BBBBRRRR
    sta $FDB3
```

Drawing an 8x8 bitmap

We're going to show a bitmap of a smiley...

As each pixel is 16 color - each pixel will be defined by a single nibble.

```
Bitmap:
    DB $00,$11,$11,$00      ;  0
    DB $01,$11,$11,$10      ;  1
    DB $11,$31,$13,$11      ;  2
    DB $11,$11,$11,$11      ;  3
    DB $11,$11,$11,$11      ;  4
    DB $11,$21,$12,$11      ;  5
    DB $01,$12,$21,$10      ;  6
    DB $00,$11,$11,$00      ;  7
BitmapEnd:
```

We're going to use a function called "GetScreenPos"... this will calculate the memory address of the pixel we want to write from an X,Y co-ordinate:

Each line is 80 bytes wide ($50) and our screen starts at $C000 so our formula is:

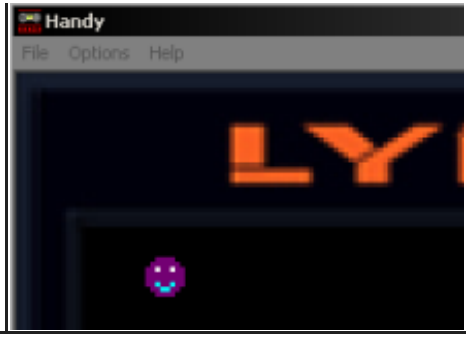
Address = $C000 + (Ypos * $50) + Xpos

We effect a multiply by bitshits... $50 in binary is %01010000 ...

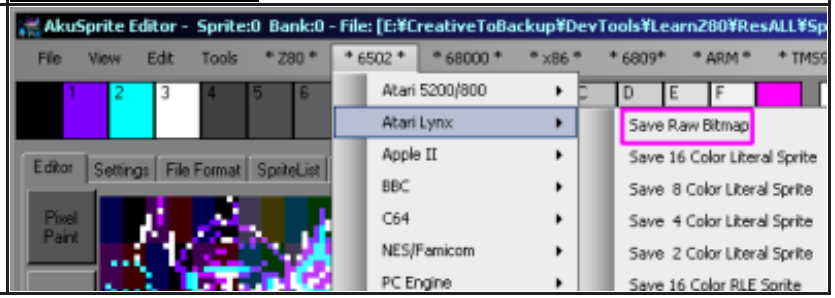
We will shift the Y bits into each '1' position then add to a running total... then add the base and the Xpos

```
GetScreenPos:
    lda #$00        ;Reset z_C
    sta z_c

    tya             ;Move Y into top byte   = YYYYYYYY 00000000
    lsr
    ror z_c
    lsr
    ror z_c         ;Shift Right Twice      = 00YYYYYY YY000000

    sta z_d         ;Store High byte in total
    lda z_c
    sta z_e         ;Store Low byte in total

    lda z_d         ;Shift Right Twice      = 0000YYYY YYYY0000
    lsr
    ror z_c
    lsr
    ror z_c

    clc             ;Add High byte to total
    adc z_d
    adc #$C0        ;Screen base at &C0000
    sta z_d


    lda z_c         ;Add Low byte to total
    adc z_e
    sta z_e

    lda z_d         ;Add any carry to the high byte
    adc #0
    sta z_d

    clc             ;Add the X pos
    txa
    adc z_e
    sta z_e

    lda z_d         ;Add any carry to the high byte
    adc #0
    sta z_d
    rts
```

Our Smiley will be shown on the screen.



Drawing a larger bitmap

The smiley is fine... but lets try something a bit more substantial... lets draw our Chibiko character onscreen. We'll include the bitmap data from a file.	`Bitmap:` `  incbin "\ResALL\Sprites\RawMSX.RAW"` `BitmapEnd:`
The routine we wrote before will do the job... we just need to change the parameters... Each byte holds two pixels - so our 48 pixel image is 24 bytes wide.	`bmpwidth equ 24  ;Width in bytes` `bmpheight equ 48  ;Height in lines`
Our larger sprite will be shown onscreen	
You can export bitmap data for this example with my Akusprite editor - it's free and open source, and included in the Sources.7z	

Lesson S7 - Bitmap Drawing on the Nes / Famicom

The Famicom graphics hardware is a little tricky... to get an image on the screen we'll have to define a tile, but we'll also need to set up the screen.

Lets get a 8x8 Smiley onto the NES Screen!

NES_Bitmap.asm

Starting a Nes or Famicom Cartridge

We need a header for our cartridge - the settings shown will work for a simple program.

```
org $BFF0

db "NES",$1a          ;ID
db $01                ;Rom pages (16k each)
db $0                 ;CHR-ROM pages
db %01000010          ;nnnnFTBM        nnnn = mapper no bottom 4 bits ,
db %00000000          ;nnnn--PV        mapper (top 4 bits...  Pc10 arcade,
db 0                  ;Ram pages
db 0,0,0,0,0,0,0
                      ;We selected Mapper 4 - it has 8k VRAM , 8K Sram and 128k rom
```

We also need a footer... this has definitions pointing to the start of the program and interrupt handlers

```
;Cartridge Footer
    org $FFFA
    dw nmihandler           ;FFFA - Interrupt handler
    dw ProgramStart         ;FFFC - Entry point
    dw irqhandler           ;FFFE - IRQ Handler
```

We're going to need a few bytes in the zero page to store data, we also need an IRQ handler of some kind (a return in this case)

Vblank (The point when the screen is not being drawn) is important, this is the only time we can write to VRAM... so we can detect when this is possible we use zero page entry $7F as a marker.. and alter this when vblank occurs

Our program will start by disabling interrupts and turning on the stack.

```
vblanked    equ $7F     ;Zero page address of Vblank count

nmihandler:     ;This procuedure runs after each frame (See footer.asm)
    php
    inc vblanked        ;Alter Vblank Zero page entry
    plp
irqhandler: ;Do nothing
    rti

ProgramStart:
    sei                 ;Interrupts off
    cld                 ;Clear Decimal flag

    ldx #$ff            ;Set up stack
    txs
```

We need to set up our screen, we'll define the palette and turn on the screen.

To define a palette entry we need to select a VRAM address $3F00+ ... we select the palette entry by writing the address $2006 (in big endian format)

We then write the bytes which select our colors.

We're ready to start our program.

```
;Palette
    lda #$3F        ;Select Palette ram &3F00
    sta $2006       ;PPUADDR H
    lda #0
    sta $2006       ;PPUADDR L

    ldx #4
PaletteAgain
    lda Palette-1,x
    sta $2007       ;PPUDATA
    dex
    bne PaletteAgain

;Turn ON the screen
;(Sprite enable/back enable/Sprite leftstrip / backleftstrip)
    lda #%00011110
    sta $2001       ;PPUMASK

    lda #$80        ;NMI enable (Vblank)
    sta $2000       ;PPUCTRL - VPHB SINN


Palette:
;   Color   3   2   1   0
        db $30,$21,$14,$0D
```

Functions to help with the NES video hardware.

We need some functions to help us with the Nes graphics... First is we need to wait for Vblank before we write to VRAM...

Vblank is when the screen is currently not being drawn (at the very top or bottom of the screen)

To do this we write 0 to zeropage entry 'Vblanked' (defined by a symbol)... then we wait for it to change... when vblank occurs, the value will be nonzero

```
waitframe:
    pha
        lda #$00
        sta vblanked        ;Zero Vblanked
waitloop:
        lda vblanked        ;Wait for the interrupt to change it
        beq waitloop
    pla
    rts
```

When we're defining tiles, we can only do so during Vblank, or when the screen is off...

We'll define some functions to turn the screen OFF or ON... this will help if we're defining tiles.

```
NesDisableScreen:           ;Turn OFF the screen
;(Sprite enable/back enable/Sprite leftstrip / backleftstrip)
    lda #%00000000
    sta $2001               ;PPUMASK
    ;lda #$00               ;NMI disable (Vblank)
    sta $2000               ;PPUCTRL - VPHB SINN
    rts

NesEnableScreen:            ;Turn ON the screen
;(Sprite enable/back enable/Sprite leftstrip / backleftstrip)
    lda #%00011110
    sta $2001               ;PPUMASK
    lda #$80                ;NMI enable (Vblank)
    sta $2000               ;PPUCTRL - VPHB SINN
    rts
```

We'll use a zero page pair z_de to define a vram destination address... we'll use a 'PrepareVram' function to select the destination address.

```
prepareVram:                ;Select a destination address
    lda z_d                 ;MSB - DEST ADDR
    sta $2006               ;PPUADDR
    lda z_e                 ;LSB - Dest ADDR
    sta $2006               ;PPUADDR
    rts
```

Whenever we write to vram, the tilemap scroll will change - messing up our layout (Grr!)

We'll create a function to fix the tilemap position.

```
ResetScroll:
    lda #0                  ;Scroll X
    sta $2005               ;PPUSCROLL
    lda #0-8                ;Scroll y
    sta $2005               ;PPUSCROLL
    rts
```

When it comes to getting our tile data we'll create a 'Define Tiles' Function.... this will transfer a sequence of bytes from our cartridge ROM to VRAM.

The routine will assume it's being run at the start of the program (not in game) so rather than waiting for VBLANK before writes, it will imply turn off the screen

```
    ;BC=Bytes
    ;DE=Destination Ram
    ;HL=Source Bytes
DefineTiles:                ;Send Data to tile definitions
    ldx z_C                 ;B=High byte of count - X=Low byte
    ldy #0
    jsr NesDisableScreen
    jsr prepareVram         ;Calculate destination address

DefineTilesAgain
    lda (z_HL),Y
    sta $2007               ;PPUDATA - Write data to data-port
    iny
    bne DefineTilesAgainYok

    inc z_h                 ;INC High byte Y=low byte
DefineTilesAgainYok:
    txa                     ;Is Low Byte Zero
    bne DefineTilesDecBC_C
    lda z_B                 ;Is High Byte Zero (Are We done?)
    beq DefineTilesAgainDone
    DEC z_B                 ;DEC Count High byte (X is low byte)
DefineTilesDecBC_C:
    DEx                     ;Decrease Count Low Byte
    jmp DefineTilesAgain
DefineTilesAgainDone:
    jmp NesEnableScreen
```

Drawing an 8x8 bitmap

We're going to define an 8x8 smiley tile!

Each pixel on the NES tilemap is 4 colors... so is defined by 2 bits...

The NES tilemap is defined in bitplanes, the 8x8 Bit0's are all grouped in 8 consecutive bytes... then all the bit 1's

```
Bitmap:
;        00000000 - Bitplane 0
    DB %00111100      ;  0
    DB %01111110      ;  1
    DB %11111111      ;  2
    DB %11111111      ;  3
    DB %11111111      ;  4
    DB %11011011      ;  5
    DB %01100110      ;  6
    DB %00111100      ;  7
;        11111111 - Bitplane 1
    DB %00000000      ;  1
    DB %00000000      ;  2
    DB %00100100      ;  3
    DB %00000000      ;  4
    DB %00000000      ;  5
    DB %00100100      ;  6
    DB %00011000      ;  7
    DB %00000000      ;  8
BitmapEnd:
```

We need to calculate the VRAM address of the next tile we want to change...
The Tilemap starts at VRAM address $2000, and the tilemap is 32 tiles wide, and each tile is 1 byte in memory, so our formula is:

Address= $2000 + (Ypos*32) + Xpos

We need to multiply the CursorY by 32... we do this by repeated bitshifts.

Before writing we have to wait for Vblank... We write the address bytes to $2006

```
GetVDPScreenPos:              ;BC=XYpos
        lda z_c
        ;and #%00000111       ;Ypos * 32 tiles per line
        asl
        asl
        asl
        asl
        asl
        ora z_b               ;Add Xpos
        sta z_l               ;Store in L byte
        lda z_c
        and #%11111000        ;Other bits of Ypos for H byte
        lsr
        lsr
        lsr
        clc
        adc #$20              ;$2000 ofset for base of tilemap
        jsr waitframe         ;Wait for Vblank
        sta $2006             ;PPUADDR
        lda z_L
        sta $2006             ;PPUADDR
    rts
```

First we'll need to transfer our tile into VRAM using the DefineTiles function... this transfers z_bc bytes from ROM address z_hl to VRAM address z_de

Next we set the VRAM location with GetVDPScreenPos - this uses XY position z_b,z_c

Finally we actually write our tilenumber into VRAM, we do this by writing 128 to $2007

```
lda #<Bitmap            ;Source Bitmap Data
sta z_L
lda #>Bitmap
sta z_H

lda #<(BitmapEnd-Bitmap);Source Bitmap Data Length
sta z_C
lda #>(BitmapEnd-Bitmap)
sta z_B

lda #<$0800             ;Tile 128 (16 bytes per tile)
sta z_E
lda #>$0800
sta z_D

jsr DefineTiles         ;Define the tile patterns

lda #3                  ;Start SX
sta z_b
lda #3                  ;Start SY
sta z_c
jsr GetVDPScreenPos     ;Calculate Tilemap mempos

lda #128                ;Tile Num
sta $2007               ;PPUDATA - Save Tile selection to Vram

jmp resetscroll         ;Need to reset scroll after writing to VRAM
```

The 8x8 smiley will be shown to screen.

Drawing a larger sprite

If we want to draw a bigger sprite, we'll need to split it up into 8x8 tiles...

We'll import a bitmap into our rom with INCBIN

```
Bitmap:
    incbin "\ResALL\Sprites\RawNES.RAW"
BitmapEnd:
```

We'll need a new command called 'FillAreaWithTiles' This will draw consecutive tile numbers to a grid, and rebuild our sprite.

```
lda #3                  ;Start SX
sta z_b
lda #3                  ;Start SY
sta z_c

ldx #6                  ;Width in tiles
ldy #6                  ;Height in tiles

lda #128                ;TileStart
jsr FillAreaWithTiles   ;Draw the tiles to screen
```

The "Fill Area with Tiles" will do the tile drawing for us -

It recalculates the start of a line with GetVDPScreenPos, then writes a line of tiles...

This is repeated for each Y line

```
DefineTiles:                ;Send Data to tile definitions
    ldx z_C                 ;B=High byte of count - X=Low byte
    ldy #0
    jsr NesDisableScreen
    jsr prepareVram         ;Calculate destination address

DefineTilesAgain
    lda (z_HL),Y
    sta $2007               ;PPUDATA - Write data to data-port
    iny
    bne DefineTilesAgainYok

    inc z_h                 ;INC High byte Y=low byte
DefineTilesAgainYok:
    txa                     ;Is Low Byte Zero
    bne DefineTilesDecBC_C
    lda z_B                 ;Is High Byte Zero (Are We done?)
    beq DefineTilesAgainDone
    DEC z_B                 ;DEC Count High byte (X is low byte)
DefineTilesDecBC_C:
    DEx                     ;Decrease Count Low Byte
    jmp DefineTilesAgain
DefineTilesAgainDone:
    jmp NesEnableScreen
```

Here is the result!

We've only used the tilemap in this example... if we want to do flexible Sprites - we would want to use the Nes' hardware sprites ...

These can move by pixels, and allow for fast smooth moving objects - but there's a limit to how many can be onscreen.

Lesson S8 - Bitmap Drawing on the SNES / Super Famicom

We learned last time how to get a bitmap onto the screen on the NES - this time we're going to draw an 8x8 smiley on the SNES

Lets learn how!

SNS_Bitmap.asm

Starting a SNES/SFC Cartridge

Our cartridge needs to start at address $8000, When our program starts, we'll disable interrupts	`org $8000 ;Start of ROM` `SEI ;Stop interrupts`
Our cartridge also needs a footer - the one here will work for our purposes.	

```
    org $FFC0
    ; "123456789012345678901"
    db "www.ChibiAkumas.com  "  ; Program title (21 byte Ascii string)

    db $20      ;Rom mode/speed (bits 7-4 = speed, bits 3-0 = map mode)
    db $00      ;Rom type (bits 7-4 = co-processor, bits 3-0 = type)
    db $01      ;Rom size in banks (1bank=32k)
    db $00      ;Ram size (0=none)
    db $00      ;Country/video refresh (ntsc 60hz, pal 50hz) (0=j 1=us/eu)
    db $00      ;Developer id code
    db $00      ;Rom version number
    db "cc"     ;Complement check
    db "cs"     ;Checksum

;65816 mode vectors
    dw $0000    ;Reserved
    dw $0000    ;Reserved
    dw $0000    ;Cop vector   (cop opcode)
    dw $0000    ;Brk vector   (brk opcode)
    dw $0000    ;Abort vector (unused)
    dw $0000    ;Vblank interrupt handler
    dw $0000    ;Reset vector (unused)
    dw $0000    ;Irq vector   (h/v-timer/external interrupt)

;6502 mode vectors
    dw $0000    ;Reserved
    dw $0000    ;Reserved
    dw $0000    ;Cop vector   (cop opcode)
    dw $0000    ;Brk vector   (unused)
    dw $0000    ;Abort vector (unused)
    dw $0000    ;Vblank interrupt handler
    dw $8000    ;Reset vector (cpu is always in 6502 mode on reset)
    dw $0000    ;Irq/brk vector
```

We're going to need to set up our screen...

First we need to initialize the tilemap, we need to set the base address in VRAM ($0000) and the tilemap size (32x32)

We can only write to VRAM during Vblank while the screen is on, so we turn the screen off during our initialization

```
    ;aaaabbbb -aaa=base addr for BG2 bbb=base addr for BG1
lda #%00010001
sta $210B       ;BG1 & BG2 VRAM location register [BG12NBA]

;     xxxxxxss  - xxx=address… ss=SC size  00=32x32
stz $2107       ;BG1SC - BG1 Tilemap VRAM location

; abcdefff - abcd=tile sizes e=pri fff=mode def
lda #%00001001
sta $2105       ;BGMODE - Screen mode register

;     x000bbbb - x=screen disable (1=disable)
lda #%10000000  ;Screen off
sta $2100       ;INIDISP - Screen display register
```

We need to set the palette next... We're going to set colors 0-3

We select a color by writing to $2121, and RGB bytes to $2122

	`` ;Background (Color 0) stz $2121 ;CGADD - Colour selection (0=Back) ;gggrrrrr stz $2122 ;CGDATA - Colour data register ;?bbbbbgg stz $2122 ;CGDATA ;Color 1 lda #1 ;Color 1 sta $2121 ;CGADD - Colour selection (15=Font) ;gggrrrrr lda #%00001111 sta $2122 ;CGDATA - Colour data register ;?bbbbbgg lda #%00111100 sta $2122 ;CGDATA ;Color 2 lda #2 ;Color 2 sta $2121 ;CGADD - Colour selection (15=Font) ;gggrrrrr lda #%11100000 sta $2122 ;CGDATA - Colour data register ;?bbbbbgg lda #%01111111 sta $2122 ;CGDATA ;Color 3 lda #3 ;Color 3 sta $2121 ;CGADD - Colour selection (15=Font) ;gggrrrrr lda #%11111111 sta $2122 ;CGDATA - Colour data register ;?bbbbbgg lda #%01111111 sta $2122 ;CGDATA ```
We need to configure the $2118/9 ports ... we write zero to $2115... we're setting the Vram address to AutoInc on a write to $2118	``` ;TileDefs ; i000abcd - I 0=inc on $2118 or $2139 1=$2119 or $213A.. abcd=move size stz $2115 ;VMAIN - Video port control (Inc on write to $2118) ```
Right! Our font is ready, but we now need to initialize the Tilemap... we need to reset the scroll position with $210D/E We also need to clear the tilemap... we do this by writing zeros to all the tiles in the tilemap The Tilemap starts at $0000 - and there are 1024 pairs of bytes to zero (32x32 tiles)	

```
;Set Scroll position
    stz $210D       ;BG1HOFS BG1 horizontal scroll
    stz $210D       ;BG1HOFS

    lda #-1
    sta $210E       ;BG1VOFS BG1 vertical scroll
    stz $210E       ;BG1VOFS

;Clear Screen
    stz $2116       ;MemL -Video port address [VMADDL/VMADDH]
    stz $2117       ;MemH

    ldy #4          ;Tilemap Size: 32*32 = 1024
    ldx #0
ClearTilemap:
    stz $2119       ;Zero all Tiles in Tilemap
    stz $2118
    dex
    bne ClearTilemap
    dey
    bne ClearTilemap
```

We're finally done... We now just need to actually turn on the screen!

Phew! that was hard work!

```
;Turn on the screen
        ; ---S4321 - S=sprites 4-1=enable Bgx
    lda #%00000001  ;Turn on BG1
    sta $212C       ;Main screen designation [TM]

    ;     x000bbbb - x=screen disable (1=disable) bbbb=brightness (15=max)
    lda #%00001111  ;Screen on
    sta $2100       ;INIDISP - Screen display register
```

We're not defining a font this time... We're going to define some tile patterns in a moment though - and we'll use those to draw our character.

Drawing our Smiley

We're going to draw a smiley to the screen!... we'll define it as tile patterns in VRAM...

The tile is 8x8 and because it's 16 color each pixel is defined by 4 bits..

The image is split into bitplanes, but two bitplanes (0+1) are sent line by line first, then the remaining two (2+3)

```
Bitmap:
    ;   11111111 00000000  - Bitplane 0/1
    DB %00000000,%00111100      ;  0
    DB %00000000,%01111110      ;  1
    DB %00100100,%11111111      ;  2
    DB %00000000,%11111111      ;  3
    DB %00000000,%11111111      ;  4
    DB %00100100,%11011011      ;  5
    DB %00011000,%01100110      ;  6
    DB %00000000,%00111100      ;  7
    ;   33333333  22222222  - Bitplane 0/1
    DB %00000000,%00000000      ;  0
    DB %00000000,%00000000      ;  1
    DB %00000000,%00000000      ;  2
    DB %00000000,%00000000      ;  3
    DB %00000000,%00000000      ;  4
    DB %00000000,%00000000      ;  5
    DB %00000000,%00000000      ;  6
    DB %00000000,%00000000      ;  7
BitmapEnd:
```

Now that the screen is enabled, we need to wait for Vblank before writing to the screen...

Vblank is the time during redraw when the screen has finished drawing, and the next frame hasn't started.

we can check if the screen is in Vblank by reading $4212

```
WaitVblank:
    lda $4212       ;HVBJOY - Status
    ; xy00000a  - x=vblank state y=hblank state a=joypad ready
    and #%10000000
    beq WaitVblank  ;Wait until we get nonzero - this means we're in VBLANK
    rts
```

When we want to select a destination address for VRAM writes, we do so using ports $2116/7

We're defining z_de as zero page entries for our use.

```
prepareVram:
    jsr WaitVblank
    lda z_e
    sta $2116       ;VMADDL - Destination address in VRAM L
    lda z_d
    sta $2117       ;VMADDH - Destination address in VRAM H
    rts
```

We're going to need a function to transfer the bytes from ROM to VRAM....

Because of the way the SNES hardware works,we can only write to VRAM during Vblank

we're using zero page entries z_hl as a source, z_bc as a bytecount and z_de as a VRAM destination.

```
DefineTiles:
    jsr prepareVram ;Get VRAM address

    ldx z_C         ;B=High byte of count - X=Low byte
    ldy #0
DefineTilesAgain
    jsr WaitVblank
    lda (z_HL),Y
    sta $2119       ;VMDATAH - Write first byte to VRAM
    iny

    jsr WaitVblank
    lda (z_HL),Y
    sta $2118       ;VMDATAL - were set to Autoinc address
    iny
    bne DefineTilesAgainYok

    inc z_h         ;INC High byte Y=low byte
DefineTilesAgainYok:
    txa             ;Is Low Byte Zero
    bne DefineTilesDecBC_C
    lda z_B         ;Are We done
    beq DefineTilesAgainDone
    DEC z_B         ;DEC high byte (X is low byte)
DefineTilesDecBC_C:
    DEx             ;Subtract 2
    DEX             ;Since we did 2 bytes
    jmp DefineTilesAgain
DefineTilesAgainDone:
    rts
```

We'll use a function called GetVDPScreenPos to calculate tile positions from a X,Y co-ordinate

Our tilemap is 32 tiles wide and starts from memory address $0000 , so our formula is:
Address= (Ypos*32) + Xpos

We achieve the multiplication by bitshifting.

```
GetVDPScreenPos:    ; BC=XYpos
        lda z_c
        sta z_h     ;32 tiles per Y line

        lda #0
        lsr z_h
        ror
        lsr z_h
        ror
        lsr z_h
        ror
        adc z_b     ;Add X line
        sta z_l
        jsr WaitVblank

        lda z_l
        sta $2116   ;MemL -Video port address [VMADDL/VMADDH]
        lda z_h
        sta $2117   ;VMDATAL - We're writing bytes in PAIRS!
    rts
```

To draw a tile on screen, first we need to transfer the bitmap data to VRAM with the 'DefineTiles' Function

Once we've done that, we use the Get VDPScreenPos function to select the destination XY address, then we write the tile number (and other settings) to the write port $2119/8

```
lda #<Bitmap          ;Source Bitmap Data
sta z_L
lda #>Bitmap
sta z_H

lda #<(BitmapEnd-Bitmap) ;Source Bitmap Data Length
sta z_C
lda #>(BitmapEnd-Bitmap)
sta z_B

lda #<$1800           ;Snes patterns start at $1000
sta z_E               ;each adddress holds 1 word...
lda #>$1800           ;so each 32 byte tile takes 16 addreses,
sta z_D               ;and tile 128 is at $1800
jsr DefineTiles       ;Define the tile patterns

lda #3                ;Start SX
sta z_b
lda #3                ;Start SY
sta z_c
jsr GetVDPScreenPos
jsr WaitVblank

;lda #$00             ;;vhoppptt
stz $2119             ;VMDATAH - Write first byte to VRAM

lda #128              ;tttttttt - Tilenumber
sta $2118             ;VMDATAL - were set to Autoinc address
                          ; on 2118 write
```

Our smiley will be drawn to the screen!

The tile has been drawn to the screen, but there's a lot of corruption - this is because tile 0 has not been defined - and the Mesen emulator fills the ram with garbage on bootup.

If we wanted to set the background to black - we need to set tile 0 to all zeros - Loading our font would have done this.

Drawing a larger bitmap

This time we're going to draw a 48x48 image of our 'Chibiko' mascot
We'll import a bitmap image from file.

```
Bitmap:
    incbin "\ResALL\Sprites\RawSNS.RAW"
BitmapEnd:
```

If we want to draw a bigger image, we'll need to split our image into multiple tiles, we'll then need to set all the positions of the area that makes up the bitmap to the correct patterns.

we'll specify a start position XY in (z_b,z_c) - and a Width / Height in in X / Y - finally a tile number to start with in z_D

The routine will calculate the start of a line... and write consecutive tile numbers from z_D

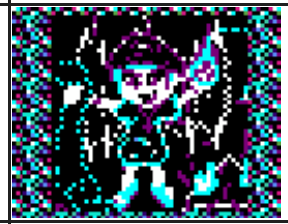
After each horizontal line,we need to recalculate the memory position

```
FillAreaWithTiles:
    sta z_d
FillAreaWithTiles_Yagain:
    jsr GetVDPScreenPos
    phx
FillAreaWithTiles_Xagain:
        jsr WaitVblank
        ;lda #$00    ;vhopppttt
        stz $2119    ;VMDATAH - Write first byte to VRAM

        lda z_d      ;ttttttttt
        sta $2118    ;VMDATAL - were set to Autoinc address
                     ;   on 2118 write
        inc z_d
        dex
        bne FillAreaWithTiles_Xagain
        inc z_c
    plx
    dey
    bne FillAreaWithTiles_Yagain
    rts
```

We use the FillAreaWithTiles function to fill the area of the screen with our character.

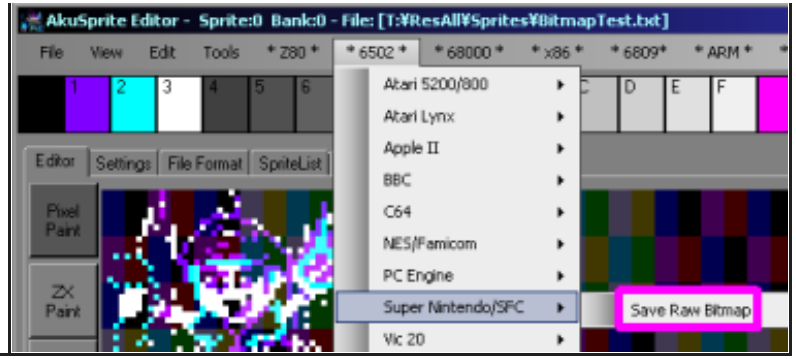
```
  lda #3              ;Start SX
  sta z_b
  lda #3              ;Start SY
  sta z_c

  ldx #6              ;Width in tiles
  ldy #6              ;Height in tiles

  lda #128            ;TileStart

  jsr FillAreaWithTiles ;Draw the tiles to screen
```

Our character will be drawn to the screen!



To use this code, you'll need to split a bitmap into tiles - and those tiles will have to be in the correct format.

You can use my AkuSprite editor to do this - it's free and open source, and included in the Sources.7z



Lesson S9 - Bitmap Drawing on the on the PC Engine/TurboGrafx-16 Card

The PC Engine uses a tilemap for its background graphics... to show Hello World we'll need to define our font as tiles, then use those tiles to show the letters of our message

SpriteTest.asm

Starting a PC Engine/TurboGrafx-16 Card

The header is pretty simple... Our program will start at $E000

```
    org $e000       ;bank $0
ProgramStart:
```

We also need a footer, it's just a word pointing to the start of our program

```
  org $fffe
  dw ProgramStart         ;Reset Vector
```

When our program starts, we need to set a lot of things up!

First we turn off interrupts, set highspeed mode, and clear the decimal flag.

Next we need to 'Page in the RAM and IO banks - this configures parts of the addressable memory, pointing them to underlying hardware.... we do this with a special 6280 command called TAM

We also set up the stack pointer... finally we turn the interrupots off with port $1402

```
ProgramStart:
    sei             ;Disable interrupts
    csh             ;Highspeed Mode
    cld             ;Clear Decimal mode

    lda #$f8        ;map in RAM
    tam #%00000010  ;TAM1 (2000-3FFF)

    lda #$ff        ;map in I/O (#$ff)
    tam #%00000001  ;TAM0 (0000-1FFF)
    tax
    txs             ;Init stack pointer

        ;       T12 - TIQ, IRQ1, IRQ2
    lda #%00000111
    sta $1402       ;IRQ mask... 1=Off
```

We need to set up the Tilemap... we need to select the video registers with the special command ST0... then set values for those registers with ST1 and ST2

First we turn the tilemap on... next we set the tilemap size - we set it to 32x32, Finally we reset the position - so that the first tile in the tilemap is the top left corner of the screen.

```
;   ScreenInit
    st0 #5              ;RegSelect 5
        ;BSXXIIII  Backgroundon Spriteon eXtendedsync Interruptenable
    st1 #%10000000      ;Background ON, Sprites On
    st2 #0

    st0 #9
        ; 0BBB0000
    st1 #%00000000      ;BACKGROUND Tilemap size (32x32)
    st2 #0

;Reset Background scroll registers
    st0 #7              ;Background X-scroll (------XX XXXXXXXX)
    st1 #0
    st2 #0

    st0 #8              ;Background Y-scroll (-------Y YYYYYYYY)
    st1 #248            ;Move Byte pos 0 to top left of screen
    st2 #0
```

Next we're going to set up the palette... we select a color to change with registers $0402/3 and set the new RGB value with registers $0404/5

```
;Background Color
    stz $0402           ;Palette address L
    stz $0403           ;Palette address H
    stz $0404           ;GGRRRBBB
    stz $0405           ;-------G
;Color 1
    lda #1
    sta $0402           ;Palette address L
    stz $0403           ;Palette address H
    lda #%00011011
    sta $0404           ;GGRRRBBB
    lda #%00000000
    sta $0405           ;-------G
;Color 2
    lda #2
    sta $0402           ;Palette address L
    stz $0403           ;Palette address H
    lda #%11000111
    sta $0404           ;GGRRRBBB
    lda #%00000001
    sta $0405           ;-------G
;Color 3
    lda #3
    sta $0402           ;Palette address L
    stz $0403           ;Palette address H
    lda #%11111111
    sta $0404           ;GGRRRBBB
    lda #%00000001
    sta $0405           ;-------G
```

We're going to clear the screen - we'll do this by setting all the tiles to tile number 256

We select select the 'Address select' register (0) with ST0... then write the VRAM

address to write to $0000 with ST1/2

We then select the 'data port' register (2) with ST0... then write tilenumber 256 with ST1/2

```
    st0 #0          ;VDP reg 0 (address)
    st1 #$00        ;L - Start of tilemap $0000
    st2 #$00        ;H

    st0 #2          ;Select VDP Reg2 (data)

    ldx #4
    ldy #0          ;1024 tiles total (32x32)
ClsAgain:
    st1 #0          ;Fill the entire area with our "Space tile"
    st2 #%00000001      ;(tile 256)
    dey
    bne ClsAgain
    dex
    bne ClsAgain
```

The PC Engine has lots of special commands... most important for us here are ST0 , ST1 and ST2... these save fixed values to the graphics hardware, and are equivalent to STA $0100, STA $0102 and STA $0103

ST0 Selects a register, and ST1/2 save values to that register... Register 0 is the 'address select' register... Register 2 is Data write - sending data to the address selected with Register 0
It may sound confusing, but don't worry too much if you don't understand it yet - just copy the code here for now.

Drawing an 8x8 Smiley to the screen.

We're going to draw a smiley to the screen!... we'll define it as tile patterns in VRAM...

The tile is 8x8 and because it's 16 color each pixel is defined by 4 bits..

The image is split into bitplanes, but two bitplanes (0+1) are sent line by line first, then the remaining two (2+3)

```
Bitmap:
    ;   00000000  11111111       - Bitplane 0/1
    DB %00111100,%00000000      ;  0
    DB %01111110,%00000000      ;  1
    DB %11111111,%00100100      ;  2
    DB %11111111,%00000000      ;  3
    DB %11111111,%00000000      ;  4
    DB %11011011,%00100100      ;  5
    DB %01100110,%00011000      ;  6
    DB %00111100,%00000000      ;  7
    ;   22222222  33333333       - Bitplane 2/3
    DB %00000000,%00000000      ;  0
    DB %00000000,%00000000      ;  1
    DB %00000000,%00000000      ;  2
    DB %00000000,%00000000      ;  3
    DB %00000000,%00000000      ;  4
    DB %00000000,%00000000      ;  5
    DB %00000000,%00000000      ;  6
    DB %00000000,%00000000      ;  7
BitmapEnd:
```

We're going to define a command called PrepareVram - which will select a memory address

to write to, we'll need this for our define tiles function... we write the Low address byte to $0102, and the High address to $0103 when register 0 is selected with ST0

We'll use zero page entries z_de to store the address we want to use.

```
prepareVram:        ;z_HL=VRAM address to select

    st0 #0              ;Select Memory Write Reg
    lda z_e
    sta $0102           ;st1 - L address
    lda z_d
    sta $0103           ;st2 - H Address
    rts
```

When we want to define tiles -we'll store a source address in VRAM z_de... a source address in ROM in z_hl, and a byte count in z_bc

We use the PrepareVram function to select an address, then write new bytes to $0102/3 after selecting Reg 0 with ST0

```
DefineTiles:
    jsr prepareVram         ;Select Ram address
    st0 #2                  ;Select Data reg
    ldx z_C                 ;B=High byte of count - X=Low byte
    ldy #0
DefineTilesAgain:
        lda (z_HL),Y        ;Load a byte
        sta $0102           ;Store Low byte
        iny
        lda (z_HL),Y        ;Load a byte
        sta $0103           ;Store High Byte
        iny
        bne DefineTilesAgainYok
        inc z_h             ;INC High byte Y=low byte
DefineTilesAgainYok:
        txa                 ;Is Low Byte Zero?
        bne DefineTilesDecBC_C
        lda z_B             ;Are We done
        beq DefineTilesAgainDone
        DEC z_B             ;DEC high byte (X is low byte)
DefineTilesDecBC_C:
        DEx                 ;Subtract 2
        DEX                 ;Since we did 2 bytes
        jmp DefineTilesAgain
DefineTilesAgainDone:
    rts
```

We're going to define a function called GetVDPScreenPos to select an X,Y Tilemap position in Vram memory

As the tilemap is at VRAM address $0000 and each line is 32 tiles wide, our formula is:

Address=(Ypos *32) + X

We multiply Y by 32 by bishifts, and select the calculated address...

```
GetVDPScreenPos:     ; BC=XYpos
        st0 #0          ;Select Vram Write
        lda z_c
        ;and #%00000111 ;Multiply Ypos by 32 - low byte
        asl
        asl
        asl
        asl
        asl
        clc
        adc z_b         ;Add Xpos
        sta $0102       ;Send to Data-L

        lda z_c
        and #%11111000  ;Multiply Ypos by 32 - low byte
        lsr
        lsr
        lsr
        sta $0103       ;Send to Data-H
    rts
```

We need to transfer the tile pattern data into vram with the define tiles function using the DefineTiles function we wrote.

```
lda #<Bitmap        ;Source Bitmap Data
sta z_L
lda #>Bitmap
sta z_H

lda #<(BitmapEnd-Bitmap);Source Bitmap Data Length
sta z_C
lda #>(BitmapEnd-Bitmap)
sta z_B

lda #<$1800         ;Tile 384 (256+128 - 32 bytes per tile)
sta z_E
lda #>$1800
sta z_D
jsr DefineTiles     ;Define the tile patterns
```

Next we want to actually set the tilemap to show our tile... we use GetVDPScreenpos to select a screen tile vram address..

Next we use ST0 to select Register 2 (Data Write) and write 256+128 (384 - the tile of our smiley) using $0102/3 or ST1/2

```
lda #3              ;Start SX
sta z_b
lda #3              ;Start SY
sta z_c

jsr GetVDPScreenPos ;Recalculate memory position
st0 #2              ;Set Write Register

                    ;Save the TileNum to Vram
lda #128            ;Tile 128 (Smileu)
sta $0102           ;L Byte
st2 #1              ;H Byte - Tile 256+
```

Our Smiley tile will be shown to screen..

Phew! that was a lot of work!

Drawing a larger bitmap

This time we'll try something a bit more substantial... we'll draw our website mascot 'Chibiko' to the screen...

This image is 48x48 pixels, so we'll need to split it into multiple tiles... we'll load the data in with an IncBin command.

```
Bitmap:
    incbin "\ResALL\Sprites\RawPCE.RAW"
BitmapEnd:
```

If we want to draw a bigger image, we'll need to split our image into multiple tiles, we'll then need to set all the positions of the area that makes up the bitmap to the correct patterns.

we'll specify a start position XY in (z_b,z_c) - and a Width / Height in in X / Y - finally a tile number to start with in z_D

The routine will calculate the start of a line... and write consecutive tile numbers from z_D

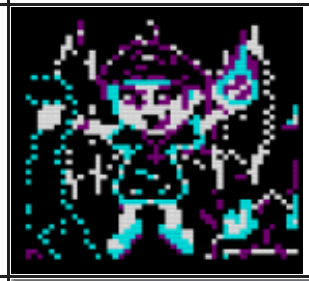
After each horizontal line,we need to recalculate the memory position

```
FillAreaWithTiles:  ; z_b = SX... z_c = SY... X=Width...
                    ; Y= Height... A=start tile
    sta z_d
FillAreaWithTiles_Yagain:
    phx
        jsr GetVDPScreenPos ;Recalculate memory position
        st0 #2              ;Set Write Register
        lda z_d
FillAreaWithTiles_Xagain:   ;Save the TileNum to Vram
        sta $0102           ;L Byte
        st2 #1              ;H Byte - Tile 256+
        clc
        adc #1              ;Increase Tile Number
        dex
        bne FillAreaWithTiles_Xagain
        sta z_d
        inc z_c             ;Inc Ypos
    plx
    dey                     ;Decrease Y count
    bne FillAreaWithTiles_Yagain
    rts
```

We use the FillAreaWithTiles function to fill the area of the screen with our character.

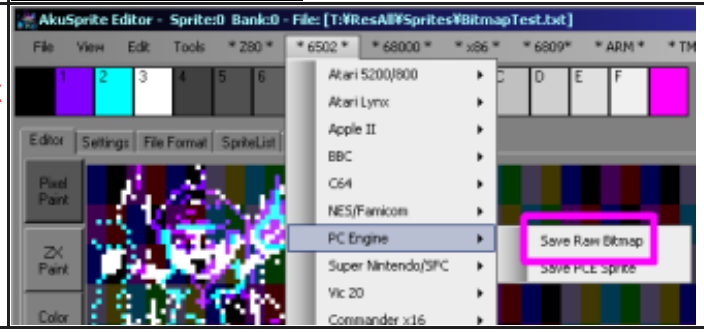
```
lda #3              ;Start SX
sta z_b
lda #3              ;Start SY
sta z_c

ldx #6              ;Width in tiles
ldy #6              ;Height in tiles

lda #128            ;TileStart

jsr FillAreaWithTiles ;Draw the tiles to screen
```

Our Chibiko character will be drawn on the screen.



To use this code, you'll need to split a bitmap into tiles - and those tiles will have to be in the correct format.

You can use my AkuSprite editor to do this - it's free and open source, and included in the Sources.7z



Lesson S10 - Joystick Reading on the BBC

We're going to extend the previous simple bitmap example, and add joystick reading to allow a player sprite to be moved around the screen.
Lets learn how to add some control to our game!

BBC_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position

We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the drawing routine from last time

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
BlankPlayer:
    lda #<BitmapBlank       ;Source Bitmap Data
    sta z_L
    lda #>BitmapBlank
    sta z_H
    jmp DrawSprite
DrawPlayer:
    lda #<Bitmap        ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

DrawSprite:
    stx z_b
    sty z_c
    jsr GetScreenPos;Get screen pos from XY into Z_DE

    ldY #0          ;Offset for bytes in this strip
BitmapNextByte:
    lda (z_hl),Y    ;Load in a byte from source - offset with Y
    sta (z_de),Y    ;Store it in screen ram - offset with Y

    inY             ;INC the offset
    cpY #8*2        ;We draw 8 lines * bitmap width
    bne BitmapNextByte
    rts

Bitmap:
        DB %00000011      ;  0   ;Vline 1 (left)
        DB %00000111      ;  1
        DB %00101111      ;  2
        DB %00001111      ;  3
        DB %00001111      ;  4
        DB %00101101      ;  5
        DB %00010110      ;  6
        DB %00000011      ;  7
        DB %00001100      ;  8   ;Vline 2 (Right)
        DB %00001110      ;  9
        DB %01001111      ; 10
        DB %00001111      ; 11
        DB %00001111      ; 12
        DB %01001011      ; 13
        DB %10000110      ; 14
        DB %00001100      ; 15
BitmapBlank:
        ds 16,0
```

We're going to move our sprite 8 Y lines at a time - this gets around the problem of the weird screen layout.

If you want to work in 1 pixel Y movements, you'll have to write some better sprite routines!

Reading the Joystick

We're going to read in from the joystick - the BBC joystick is analog - which is a bit of a pain.

We need to select an axis by writing to $FEC0, then read in from $FEC1

We need to see if the result is outside of the 'dead zone' and set the bits in z_h depending on the result.

```
Player_ReadControlsDual:;----LRUD
    lda #0                  ;Set port to read (For fire button)
    STA $FE43               ;SN76489 - Data Direction
    sta z_h

    ;lda #%00000000         ;Get Channel 0 - Joy 1 LR
    jsr Player_ReadControlsGetData
    lda #%00000001          ;Get Channel 1 - Joy 1 UD
    jsr Player_ReadControlsGetData
    rts

    ;See page 429 of the 'BBC Microcomputer Advanced user Guide'
Player_ReadControlsGetData: ;We need to convert analog to digital
    sta $FEC0                       ;Select channel
Player_ReadControlsDualWait:
    lda $FEC0                       ;Get Data
    and #%10000000
    bne Player_ReadControlsDualWait ;0= data ready

    lda $FEC1                       ;8 bit analog data
    cmp #255-32
    bcs Player_ReadControlsDualHigh
    cmp #32
    bcc Player_ReadControlsDualLow  ;Centered
    clc
    bcc Player_ReadControlsDualB    ;efective branch always

Player_ReadControlsDualLow:     ;R/D
    sec
Player_ReadControlsDualB:
    rol z_h
    clc
    rol z_h
    rts
Player_ReadControlsDualHigh:    ;U/L
    clc
    rol z_h
    sec
    rol z_h
    rts
```

Our test program

First we're going to clear our screen... we'll define an address to clear in z_hl ($4180 - the start of

the bitmap screen)

we'll define the number of bytes to clear in XY ($3F00)

We then use a loop to clear the screen.

```
    lda #$41    ;Clear $4180-$8080
    sta z_h
    lda #$80
    sta z_l

    ldx #$3F    ;Clear $3F00 bytes
    ldy #$00
    lda #0
FillZeros:
    sta (z_hl),y
    dey
    bne FillZeros
    inc z_h
    dex
    bne FillZeros
```

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
  lda #4          ;Start SX
  sta PlayerX
  lda #80         ;Start SY
  sta PlayerY

  lda #0          ;Fake No Keys on first run
  sta z_h
  jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    beq infloop     ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX     ;Back up X
        stx PlayerX2

        ldy PlayerY     ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX     ;Back up X
        ldy PlayerY     ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    pla
    sta z_h
    and #%00000001  ;---FLRUD
    beq JoyNotUp    ;Jump if UP not presesd
    tya
    sec             ;Move Y Up the screen
    sbc #8
    tay
JoyNotUp:
    lda z_h
    and #%00000010  ;---FRLDU
    beq JoyNotDown  ;Jump if DOWN not presesd
    tya
    clc             ;Move Y Down the screen
    adc #8
    tay
JoyNotDown:
    lda z_h
    and #%00000100  ;---FLRUD
    beq JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00001000  ;---FLRUD
    beq JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #80-1
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #200-8
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer   ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY      ;Wait a bit!

    jmp infloop


PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

We've got a movable sprite onscreen! You could use this as a basic game.

If you want to read two players, or fire buttons you'll need to use better joystick code - see the platform specific series for more details!

Lesson S11 - Joystick reading on the C64

Lets take our previous Bitmap Smiley example, and extend it out into a joystick controlled movable object - we could use this as the starting point for a simple game.

C64_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the sprite drawing routine from last time:

```
DrawSprite:
    txa
    pha
    tya
    pha
```

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
        stx z_b
        sty z_c
        jsr GetScreenPos ;Get screen pos from XY into Z_DE

BitmapNextLine:
        ldY #0           ;Offset for bytes in this strip
BitmapNextByte:
        lda (z_hl),Y     ;Load in a byte from source offset with Y
        sta (z_de),Y     ;Store it in screen ram - offset with Y
        inY              ;INC the offset
        cpY #8           ;We draw 8 lines * bitmap width
        bne BitmapNextByte

    pla
    tay
    pla
    tax

;Fill Color Data
    stx z_b
    sty z_c
    jsr GetColMemPos     ;Get color pos from XY into Z_DE

    ldy #0
    ifdef FourColor
        lda #$43         ;Color
        sta (z_de),Y     ;%22221111 Color 1,2
        lda #01
        sta (z_bc),Y     ;%----3333 Color 3 (White)
    else
        lda #$40         ;Color
        sta (z_de),Y     ;%11110000 Color 1,2
    endif

    rts


Bitmap:
    ifdef FourColor
        DB %00010100     ;  0
        DB %00010101     ;  1
        DB %01110101     ;  2
        DB %01010101     ;  3
        DB %01010101     ;  4
        DB %01100101     ;  5
        DB %00011001     ;  6
        DB %00010100     ;  7
    else
        DB %00111100     ;  0
        DB %01111110     ;  1
        DB %11011011     ;  2
        DB %11111111     ;  3
        DB %11111111     ;  4
        DB %11011011     ;  5
        DB %01100110     ;  6
        DB %00111100     ;  7
    endif

BitmapBlank:
    ds 8,0               ;Empty Sprite
```

Reading the Joystick

The C64 makes joystick reading super easy! all we need to do is read in from $DC00 / $DC01 for port 1 / 2

```
Player_ReadControlsDual:
    ;lda $DC00          ;Read in Joystick 1
    lda $DC01           ;Read in Joystick 2
    sta z_h         ;---FRLDU
    rts
```

Our test program

First we're going to clear our screen... we'll define an address to clear in z_hl... We'll define the number of bytes to clear in XY

We then use a loop to clear the screen.

We need to fill two screen areas:

The bitmap area $2000-$4000
The color attributes $0400-$0800

```
  lda #$20          ;Clear $4000-$6000
  sta z_h
  ldx #$20          ;Clear $2000 bytes
  jsr FillZeros

  lda #$04          ;Clear $0400-$0800
  sta z_h
  ldx #$04          ;Clear $400 bytes
  jsr FillZeros


FillZeros:
    lda #$00      ;Byte to fill
    sta z_l       ;Bottom Byte of Dest Address
    tay           ;Bottom Byte of Byte Count

FillZerosB:
    sta (z_hl),y
    dey
    bne FillZerosB
    inc z_h
    dex
    bne FillZerosB
    rts
```

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
  lda #4            ;Start SX
  sta PlayerX
  lda #80           ;Start SY
  sta PlayerY

  lda #0            ;Fake No Keys on first run
  sta z_h
  jmp StartDraw     ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    cmp #%11111111
    beq infloop      ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX      ;Back up X
        stx PlayerX2

        ldy PlayerY      ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX      ;Back up X
        ldy PlayerY      ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 1 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    pla
    sta z_h
    and #%00000001  ;---FRLDU
    bne JoyNotUp    ;Jump if UP not presesd
    tya
    sec             ;Move Y Up the screen
    sbc #8
    tay
JoyNotUp:
    lda z_h
    and #%00000010  ;---FRLDU
    bne JoyNotDown  ;Jump if DOWN not presesd
    tya
    clc             ;Move Y Down the screen
    adc #8
    tay
JoyNotDown:
    lda z_h
    and #%00000100  ;---FRLDU
    bne JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00001000  ;---FRLDU
    bne JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #40
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #200
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer  ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY     ;Wait a bit!

    jmp infloop


PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

We've only used the tilemap in this example... if we want to do flexible Sprites - we would want to use the Nes' hardware sprites ...

These can move by pixels, and allow for fast smooth moving objects - but there's a limit to how many can be onscreen.

Lesson S12 - Joystick Reading on the VIC-20

The VIC-20 uses a digital joystick, and we can read it's directions in from a pair of ports...

Lets enhance our previous bitmap example, and make it move!

VIC_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're defining two character tiles this time - one's a smiley, the other's just an empty sprite.

We're going to use a slightly modified version of the tile setting routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
BlankPlayer:
    lda #1      ;Tile Num (blank sprite)
    jmp DrawSprite
DrawPlayer:
    lda #0      ;Tile Num (Smiley)
DrawSprite:
    pha
        stx z_b
        sty z_c
        jsr GetVDPScreenPos ;Calculate Tilemap mempos
    pla

    sta (z_hl),y        ;Transfer Tile to ram
    lda z_h
    clc
    adc #$78            ;add Offset to Color Ram ($9600)
    sta z_h
    lda #4              ;Color
    sta (z_hl),y        ;Set Tile Color
    rts

Bitmap: ;Smiley
        DB %00111100    ;  0
        DB %01111110    ;  1
        DB %11011011    ;  2
        DB %11111111    ;  3
        DB %11111111    ;  4
        DB %11011011    ;  5
        DB %01100110    ;  6
        DB %00111100    ;  7

        DS 8,0          ;Blank Sprite
BitmapEnd:
```

Reading the Joystick

On the Vic 20, we ned to read in from ports $9120 and $911F to get UDLR and Fire...

We'll need to ensure that port B is set to READ by writing to $9122

```
Player_ReadControlsDual:
    lda #%01111111
    sta $9122    ;Set Data Direction of port B to READ (0=read)

;   lda #%11000011
;   sta $9113    ;Set Data Direction of port A to READ (0=read)

    lda $9120    ;Port B (R------- Switch)
    sta z_as

    lda $911F    ;Port A (--FLDU-- Switches)
    rol z_as
    rol
    sta z_h      ;-FLDU--R

;   lda #255
;   sta $9122    ;Reset port B (for Keyb col scan)
    rts
```

Note: we're cutting corners to save memory ... we don't need to set port A ($9113) to read, and we're not switching port B back!

It works ok on our emulator, but you may prefer to do this too.

Our test program

First we're going to clear our screen... We'll use a loop to clear the characters onscreen.

We do this by writing Tile 1 (the blank tile) to areas $1E00-$2000

```
    lda #$1E     ;Clear $1E00-$2000
    sta z_h
    lda #$00
    sta z_l

    ldx #$02     ;Clear $200 bytes
    tay          ;#$00
    lda #1       ;Tile 1=Blank
FillZerosB:
    sta (z_hl),y
    dey
    bne FillZerosB
    inc z_h
    dex
    bne FillZerosB
```

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
    lda #3          ;Start SX
    sta PlayerX
    lda #3          ;Start SY
    sta PlayerY

    lda #0          ;Fake No Keys on first run
    sta z_h
    jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    cmp #%11111111
    beq infloop     ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX     ;Back up X
        stx PlayerX2

        ldy PlayerY     ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX     ;Back up X
        ldy PlayerY     ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 1 then the button isn't down - and we skip over the routine to move the character's X or Y position

```
    pla
    sta z_h
    and #%00001000  ;-FLDU--R
    bne JoyNotUp    ;Jump if UP not presesd
    dey             ;Move Y Up the screen
JoyNotUp:
    lda z_h
    and #%00010000  ;-FLDU--R
    bne JoyNotDown  ;Jump if DOWN not presesd
    iny             ;Move Y Down the screen
JoyNotDown:
    lda z_h
    and #%00100000  ;-FLDU--R
    bne JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00000001  ;-FLDU--R
    bne JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #22
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #23
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer  ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY     ;Wait a bit!

    jmp infloop


PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Lesson S13 - Joystick Reading on the Atari 800 / 5200

Joystick reading on the Atari is easy... or a pain!... it just depends if you're programing for the Atari 800 (which has a digital input via the PIA) or the 5200 (which has no PIA, so only has analog via the POKEY!)

Either way, We'll figure it out... Here we go!

A52_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position

We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We need to define the address of the PIA on the Atari 800 - we'll use it to get the digital joystick

There's no PIA on the 5200 - so we'll use the pokey instead

```
    ifdef BuildA8O      ;Atari 800 settings
GTIA equ $D000          ;GTIA address
PIA  equ $D300          ;PIA address
    org $A000           ;Start of cartridge area

    else                ;Atari 5200 settings
GTIA  equ $C000         ;GTIA address
POKEY equ $E800         ;POKEY address
    org $4000           ;Start of cartridge area
    endif
```

We're going to use a slightly modified version of the bitmap drawing routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
BlankPlayer:
    lda #<BitmapBlank       ;Source Bitmap Data
    sta z_L
    lda #>BitmapBlank
    sta z_H
    jmp DrawSprite

DrawPlayer:
    lda #<Bitmap        ;Source Bitmap Data
    sta z_L
    lda #>Bitmap
    sta z_H

DrawSprite:
    jsr GetScreenPos    ;Calculate Memory address of XY
    ldY #0          ;Line Count
BitmapNextLine:
    ldx #0
    lda (z_hl),Y    ;Copy a byte from the source
    sta (z_de,x)    ;to the destination

    clc
    lda z_e
    adc #$28        ;Move down a line (40 Bytes / $0028)
    sta z_e
    lda z_d
    adc #$00
    sta z_d

    iny
    cpy #8          ;Height (8 bytes)
    bne BitmapNextLine
    rts

Bitmap:                 ;Smiley
    ifdef FourColor
        DB %00010100    ;  0
        DB %01010101    ;  1
        DB %01110111    ;  2
        DB %01010101    ;  3
        DB %01010101    ;  4
        DB %01100110    ;  5
        DB %01011001    ;  6
        DB %00010100    ;  7
    else
        DB %00111100    ;  0
        DB %01111110    ;  1
        DB %11011011    ;  2
        DB %11111111    ;  3
        DB %11111111    ;  4
        DB %11011011    ;  5
        DB %01100110    ;  6
        DB %00111100    ;  7
    endif

BitmapBlank:
        DS 8,0          ;Blank Sprite
```

Reading the Joystick

On the Atari 800 , we can read in the UDLR controls of the first joystick from the PIA

The top nibble is joystick 2 - so we ignore this for todays example

```
;   Atari 800 Joystick Routine

    ifdef BuildA80
Player_ReadControlsDual:
        lda PIA+$0  ;22221111 - RLDU in player controls
        and #%00001111  ;Bottom Nibble is Player 1 Joystick
        sta z_h
    rts
```

On the 5200 we'll have to read from the analog hardware

We read in each axis from the POKEY ($E800/1)

We use a deadzone of 128 - and if the axis is outside of that deadzone we set a bit of the resulting joystick value - this converts analog to digital.

```
;   Atari 5200 - Atari 5200 doesn't have PIA

Player_ReadControlsDual:
    lda pokey+0         ;$E800 - POT0 - game paddle 0
    jsr Player_ReadControlsProcessAnalog

    lda pokey+1         ;$E801 - POT1 - game paddle 1
    jsr Player_ReadControlsProcessAnalog

    lda z_h
    and #%00001111  ;Bottom Nibble is Player 1 Joystick
    sta z_h
    rts

    ;Convert Analog to Digital
Player_ReadControlsProcessAnalog:
    cmp #255-64
    bcs Player_ReadControlsProcessHigh
    cmp #64
    bcc Player_ReadControlsProcessLow
    sec
    bcs Player_ReadControlsProcessB

Player_ReadControlsProcessHigh:     ;B/R
    clc
Player_ReadControlsProcessB:
    rol z_h
    sec
    rol z_h
    rts
Player_ReadControlsProcessLow:      ;T/L
    sec
    rol z_h
    clc
    rol z_h
    rts
    endif
```

*We're only loading in UDLR directions here of Joystick 1 - if you want to use Fire, or joystick 2 - see the **Platform Specific series!***

Our test program

First we're going to clear our screen... We'll use a loop to clear the characters onscreen.

We do this by writing Tile 1 (the blank tile) to areas $1E00-$2000

```
    lda #$1E    ;Clear $1E00-$2000
    sta z_h
    lda #$00
    sta z_l

    ldx #$02    ;Clear $200 bytes
    tay         ;#$00
    lda #1      ;Tile 1=Blank
FillZerosB:
    sta (z_hl),y
    dey
    bne FillZerosB
    inc z_h
    dex
    bne FillZerosB
```

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
  lda #3          ;Start SX
  sta PlayerX
  lda #3          ;Start SY
  sta PlayerY

  lda #0          ;Fake No Keys on first run
  sta z_h
  jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    beq infloop     ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX     ;Back up X
        stx PlayerX2

        ldy PlayerY     ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX     ;Back up X
        ldy PlayerY     ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 1 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    pla
    sta z_h
    and #%00000001  ;---FRLDU
    bne JoyNotUp    ;Jump if UP not presesd
    tya
    sec
    sbc #8
    tay
JoyNotUp:
    lda z_h
    and #%00000010  ;---FRLDU
    bne JoyNotDown  ;Jump if DOWN not presesd
    tya
    clc
    adc #8
    tay
JoyNotDown:
    lda z_h
    and #%00000100  ;---FRLDU
    bne JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00001000  ;---FRLDU
    bne JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #40
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #200-8
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer   ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY      ;Wait a bit!

    jmp infloop


PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Lesson S14 - Joystick Reading on the Apple II

Reading the Apple II Joystick is a serious pain! We have to read a single bit from the ports - and wait until it changes to get the analog values, then convert them to digital.

Lets figure it out!

AP2_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the bitmap drawing routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
BlankPlayer:
    lda #<BitmapBlank   ;Source Bitmap Data
    sta z_L
    lda #>BitmapBlank
    sta z_H
    jmp DrawSprite

DrawPlayer:
    lda #<BitmapSmiley  ;Source Bitmap Data
    sta z_L
    lda #>BitmapSmiley
    sta z_H

DrawSprite:
    jsr GetScreenPos

    ldY #0
BitmapNextLine:
    ldx #0
    lda (z_hl),Y    ;Read byte from source
    sta (z_de,X)    ;Write to screen

    lda z_d         ;move mempos down a line
    clc
    adc #$04        ;add $0400 to the line number
    sta z_d         ;  (only works within an  8 line block)

    iny
    cpy #8
    bne BitmapNextLine;need a recalc every 8 lines
    rts

BitmapSmiley:
    ;   C0123456
    DB %00011100    ;  0
    DB %00111110    ;  1
    DB %01101011    ;  2
    DB %01111111    ;  3
    DB %01111111    ;  4
    DB %01101011    ;  5
    DB %00110110    ;  6
    DB %00011100    ;  7

BitmapBlank:
    ds 8,0
```

Reading the Joystick

Apple Joysticks are annoying!
they are analog... we have to strobe the port then read from the X and Y ports, and count up until the top bit changes...

this is a 'timer'...using just 1 bit (the top one) it effectively returns an 'analog' value from about 0-100

We're reading in both the X and Y axis at the same time, skipping out of part of the loop when the X or Y part is complete.

```
Player_ReadControlsDual:    ;---FRLDU
    lda $C061           ;Fire 1
    and %00000001       ;Move in the fire button
    sta z_h

    lda $C070           ;Strobe Joypads
    ldy #0
    ldx #0
Joy_ReadAgain:
    pha
    pla                 ;delay
Joy_gotPDL1:            ;Jump backhere when we get X
Joy_ChkPDl0:
    lda $C064           ;<--SM ***   Y
JoySelfModAA_Plus2:
    bpl Joy_gotPDL0     ;Have we got Y?
    nop
    iny
    lda $C065           ;<--SM ***   X
JoySelfModB_Plus2:
    bmi Joy_nogots      ;Have we got X?
    bpl Joy_gotPDL1
Joy_nogots:
    inx
    jmp Joy_ChkPdl0
Joy_gotPDL0:            ;We've Got Tpos - just waiting for X
    lda  $C065          ;<--SM ***   X
JoySelfModBB_Plus2:
    bmi Joy_Nogots
```

Once we've read in the analog values, we'll convert them from analog (0-255) to digital (just a single bit 1/0 for Off/On)

```
    tya
    jsr JoyConvertAnalog;Convert Y
    txa                 ;Convert X
JoyConvertAnalog:   ;covert analog from 0-100 into L/R or U/D
    cmp #$66
    bcs Joy_Rbit
    cmp #$33
    bcc Joy_Lbit
    clc
    bcc Joy_Cbit
Joy_Rbit:
    sec
Joy_Cbit:
    rol z_h
    clc
    rol z_h
    rts
Joy_Lbit:
    clc
    rol z_h
    sec
    rol z_h
    rts
```

Reading the Apple II joystick is a Helluva pain, but the code here does manage to convert it into something vaguely manageable!

Also note, while we typically use an 8x8 smiley - due to the Apple II weirdness, it's 7x8 on this system!··· grr!

Our test program

After our screen is set up (See the bitmap tutorial) we're going to clear our screen... We'll use a loop to clear the screen.

We do this by writing Tile zeros to the screen at addresses $4000-$6000

```
    lda #$40    ;Clear $4000-$6000
    sta z_h
    lda #$00
    sta z_l

    ldx #$20    ;Clear $2000 bytes
    ldy #$00

FillZeros:
    sta (z_hl),y
    dey
    bne FillZeros
    inc z_h
    dex
    bne FillZeros
```

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to 255 - as we need to run the main loop but don't want to cause any keypresses.

```
lda #4          ;Start SX
sta PlayerX
lda #80         ;Start SY
sta PlayerY

lda #0          ;Fake No Keys on first run
sta z_h
jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h

    beq infloop      ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX      ;Back up X
        stx PlayerX2

        ldy PlayerY      ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX      ;Back up X
        ldy PlayerY      ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    pla
    sta z_h
    and #%00000001  ;---FRLDU
    beq JoyNotUp    ;Jump if UP not presesd
    tya
    sec             ;Move Y Up the screen
    sbc #8
    tay
JoyNotUp:
    lda z_h
    and #%00000010  ;---FRLDU
    beq JoyNotDown  ;Jump if DOWN not presesd
    tya
    clc             ;Move Y Down the screen
    adc #8
    tay
JoyNotDown:
    lda z_h
    and #%00000100  ;---FRLDU
    beq JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00001000  ;---FRLDU
    beq JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #40
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #192
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer  ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY     ;Wait a bit!

    jmp infloop


PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Lesson S15 - Joypad Reading on the Atari Lynx

The Lynx makes joypad reading easy for us... we can just read the keypresses in from port $FCB0

Lets make it happen!

LNX_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.

We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the bitmap routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

```
bmpwidth equ 4      ;Width in bytes
bmpheight equ 8     ;Height in lines

BlankPlayer:
    lda #<BitmapBlank   ;Bitmap source
    sta z_L
    lda #>BitmapBlank
    sta z_H
    jmp DrawSprite
DrawPlayer:
    lda #<BitmapSmiley  ;Bitmap source
    sta z_L
    lda #>BitmapSmiley
    sta z_H
DrawSprite:
    jsr GetScreenPos
    ldx #0
BitmapNextLine:
    phy
        ldY #0
BitmapNextByte:
        lda (z_hl),Y        ;Copy a byte from the source
        sta (z_de),Y        ;to the destination

        inY
        cpY #bmpwidth       ;Repeat for next byte of line
        bne BitmapNextByte

        clc
        tya
        adc z_l             ;ADD Y to Z_HL to move source
        sta z_l
        lda z_h
        adc #0
        sta z_h

        clc
        lda z_e
        adc #$50            ;ADD 50 to Z_DE to move Destination
        sta z_e
        lda z_d
        adc #0
        sta z_d
    ply
    inx
    cpx #bmpheight          ;Check if we've done all the lines
```

```
    bne BitmapNextLine      ;Repeat until we have
    rts

BitmapSmiley:
    DB $00,$11,$11,$00      ;  0
    DB $01,$11,$11,$10      ;  1
    DB $11,$31,$13,$11      ;  2
    DB $11,$11,$11,$11      ;  3
    DB $11,$11,$11,$11      ;  4
    DB $11,$21,$12,$11      ;  5
    DB $01,$12,$21,$10      ;  6
    DB $00,$11,$11,$00      ;  7

BitmapBlank:
    ds 32,0
```

Reading the Joystick

Reading in the Lynx joypad is super easy! (Yay!) we just read in from port $FCB0... this returns a byte with each button UDLR12IO represented by a bit... where 1=ButtonDown and 0=ButtonUp

```
Player_ReadControlsDual:
    lda $FCB0       ;JOYSTICK   Read Joystick and Switches
    sta z_h         ;UDLR12IO
    rts
```

Our test program

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
 lda #3          ;Start SX
 sta PlayerX
 lda #3          ;Start SY
 sta PlayerY

 lda #0          ;Fake No Keys on first run
 sta z_h
 jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    beq infloop     ;See if no keys are pressed
    pha
StartDraw:
        ldx PlayerX     ;Back up X
        stx PlayerX2

        ldy PlayerY     ;Back up Y
        sty PlayerY2

        jsr BlankPlayer ;Remove old player sprite

        ldx PlayerX     ;Back up X
        ldy PlayerY     ;Back up Y
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    pla
    sta z_h
    and #%10000000  ;UDLR12IO
    beq JoyNotUp    ;Jump if UP not presesd
    dey             ;Move Y Up the screen
JoyNotUp:
    lda z_h
    and #%01000000  ;UDLR12IO
    beq JoyNotDown  ;Jump if DOWN not presesd
    iny             ;Move Y Down the screen
JoyNotDown:
    lda z_h
    and #%00100000  ;UDLR12IO
    beq JoyNotLeft  ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00010000  ;UDLR12IO
    beq JoyNotRight ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX     ;Update X
    sty PlayerY     ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #80-3
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #102-7
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of boundsS

PlayerReset:
    ldx PlayerX2    ;Reset Xpos
    stx PlayerX

    ldy PlayerY2    ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer  ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY     ;Wait a bit!
    jmp infloop

PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Hardware sprites on the Lynx are a bit of a pain... but software sprites are pretty easy! And thanks to the Lynx Small screen and speedy CPU, we can make a great game without worrying about the hardware sprites!

Lesson S16 - Joypad Reading on the Nes / Famicom

The Famicom/NES have digital joysticks... we can read the buttons in, but we have to do this a single key at a time...

Let get it working!

NES_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the bitmap routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

We're defining tile 128 as our smiley... 129 is 16 zero bytes.

```
BlankPlayer:
    lda #129          ;Tile Num (blank)
    jmp DrawSprite
DrawPlayer:
    lda #128          ;Tile Num (Smiley)
DrawSprite:
    pha
        stx z_b
        sty z_c
        jsr GetVDPScreenPos ;Calculate Tilemap mempos
    pla
    sta $2007         ;PPUDATA - Save Tile selection to Vram
    jmp resetscroll ;Need to reset scroll after writing to VRAM

Bitmap: ;Smiley Sprite
;       00000000 - Bitplane 0
    DB %00111100      ; 0
    DB %01111110      ; 1
    DB %11111111      ; 2
    DB %11111111      ; 3
    DB %11111111      ; 4
    DB %11011011      ; 5
    DB %01100110      ; 6
    DB %00111100      ; 7
;       11111111 - Bitplane 1
    DB %00000000      ; 1
    DB %00000000      ; 2
    DB %00100100      ; 3
    DB %00000000      ; 4
    DB %00000000      ; 5
    DB %00100100      ; 6
    DB %00011000      ; 7
    DB %00000000      ; 8
;Clear Sprite
    ds 16,0
BitmapEnd:
```

Reading the Joystick

We need to read in a sequence of 8 bits from port $4016 to get each direction key... we also need to strobe the port by writing 1 then 0 to the same port...

The 8 reads from the port will return the directions:
Read 1 - A
Read 2 - B
Read 3 - Select
Read 4 - Start
Read 5 - Up
Read 6 - Down

Read 7 - Left
Read 8 - Right

```
Player_ReadControlsDual:
    txa
    pha
        ;Strobe joysticks to reset them
        ldx #$01        ;Send a 1 to joysticks (strobe reset)
        stx $4016       ;JOYPAD1 port

        dex             ;Send a 0 to joysticks (read data)
        stx $4016       ;JOYPAD1 port

        ldx #8          ;Read in 8 bits from each joystick
Player_ReadControlsDualloop:
        lda $4016       ;JOYPAD1
        lsr             ; bit0 -> Carry
        ror z_h         ;Add carry to Joy1 data
        dex
        bne Player_ReadControlsDualloop
    pla
    tax
    rts
```

Reading the NES joystick is a bit of a pain, but this code will sort it out... if you want both players pads, check our the full tutorial here

Our test program

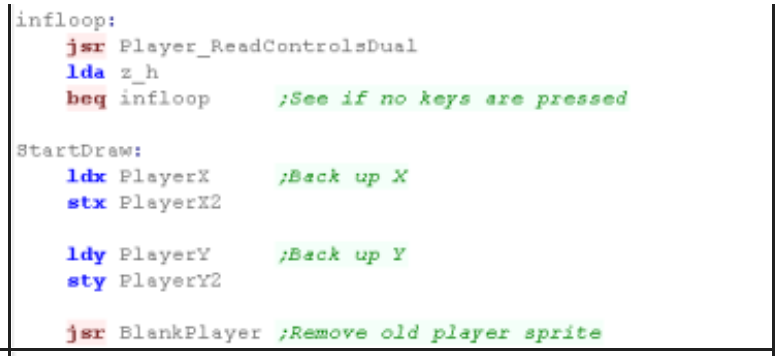
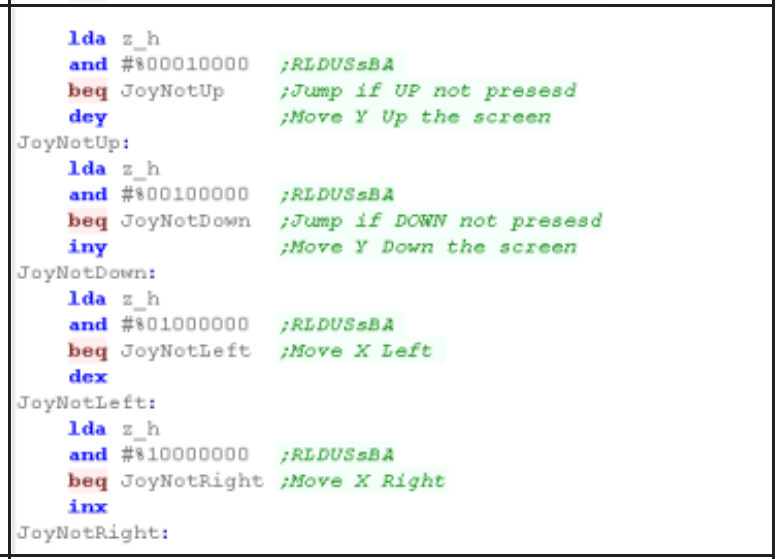
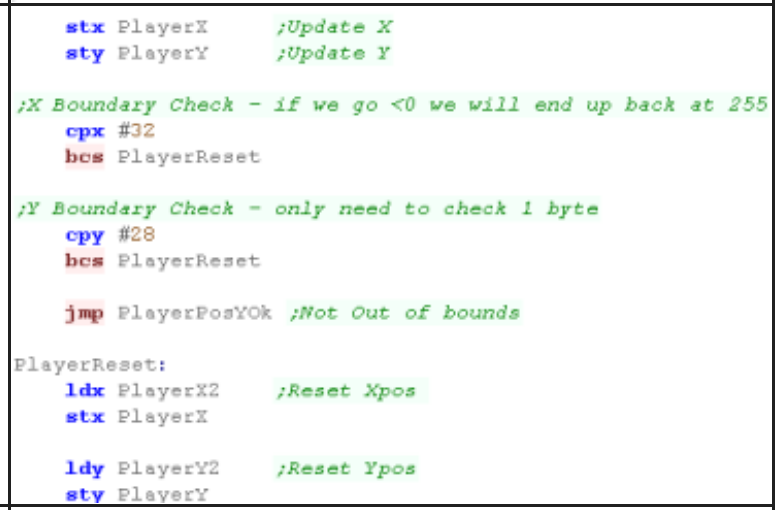
We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
lda #3          ;Start SX
sta PlayerX
lda #3          ;Start SY
sta PlayerY

lda #0          ;Fake No Keys on first run
sta z_h
jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

	
We test each of the bits in z_h (which was returned from the joystick routine) In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity	
We save the new XY position. Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255) Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup	
We're finished, so we draw the player to the screen, and wait a while! Finally we loop back, and start the procedure again!	

```
PlayerPosYOk:
    jsr DrawPlayer   ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY      ;Wait a bit!
    jmp infloop

PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Lesson S17 - Joypad Reading on the SNES / Super Famicom

Probably as a result of the originally planned NES compatibility, the SNES joypad can be read like the NES one

Lets give it a go!

SNS_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the bitmap routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

We're defining tile 128 as our smiley... 129 is 32 zero bytes.

```
BlankPlayer:
    lda #129          ;Tile Num
    jmp DrawSprite
DrawPlayer:
    lda #128          ;Tile Num
DrawSprite:
    pha
        stx z_b
        sty z_c
        lda z_h
        pha
            jsr GetVDPScreenPos ;Calculate Tilemap mempos
        pla
        sta z_h
        ;vhoppptt
        stz $2119 ;VMDATAH - Write first byte to VRAM
    pla
    sta $2118       ;VMDATAL - were set to Autoinc address
    rts

Bitmap:
;Smiley Sprite
    ;   11111111 000000000  - Bitplane 0/1
    DB %00000000,%00111100      ;  0
    DB %00000000,%01111110      ;  1
    DB %00100100,%11111111      ;  2
    DB %00000000,%11111111      ;  3
    DB %00000000,%11111111      ;  4
    DB %00100100,%11011011      ;  5
    DB %00011000,%01100110      ;  6
    DB %00000000,%00111100      ;  7
    ;   33333333  22222222  - Bitplane 0/1
    DB %00000000,%00000000      ;  0
    DB %00000000,%00000000      ;  1
    DB %00000000,%00000000      ;  2
    DB %00000000,%00000000      ;  3
    DB %00000000,%00000000      ;  4
    DB %00000000,%00000000      ;  5
    DB %00000000,%00000000      ;  6
    DB %00000000,%00000000      ;  7

;Empty Sprite
    ds 32,0
BitmapEnd:
```

Reading the Joystick

We need to read in a sequence of 8 bits from port $4016 to get each direction key... we also need to strobe the port by writing 1 then 0 to the same port...

The 8 reads from the port will return the directions:
Read 1 - A
Read 2 - B

Read 3 - Select
Read 4 - Start
Read 5 - Up
Read 6 - Down
Read 7 - Left
Read 8 - Right

```
Player_ReadControlsDual:
    txa
    pha
        ;Strobe joysticks to reset them
        ldx #$01        ;Send a 1 to joysticks (strobe reset)
        stx $4016       ;JOYPAD1 port

        dex             ;Send a 0 to joysticks (read data)
        stx $4016       ;JOYPAD1 port

        ldx #8          ;Read in 8 bits from each joystick
Player_ReadControlsDualloop:
        lda $4016       ;JOYPAD1
        lsr             ; bit0 -> Carry
        ror z_h         ;Add carry to Joy1 data
        dex
        bne Player_ReadControlsDualloop
    pla
    tax
    rts
```

This routine only loads in the classic NES buttons of Joypad 1, if you want the full buttons, or the second Joypad, see the details ***here***

Our test program

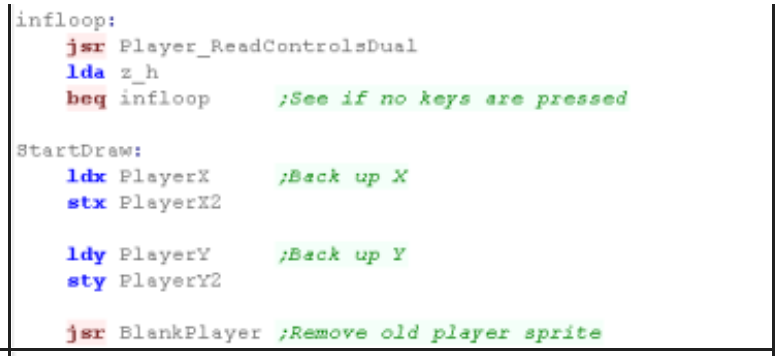
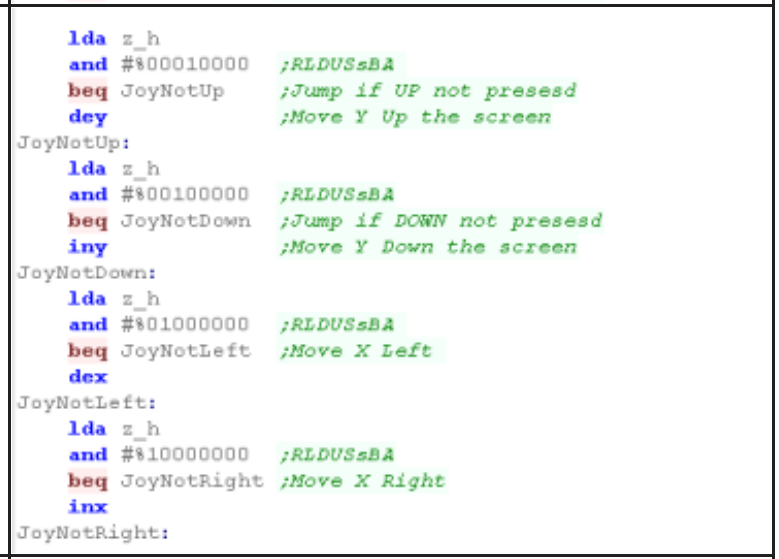
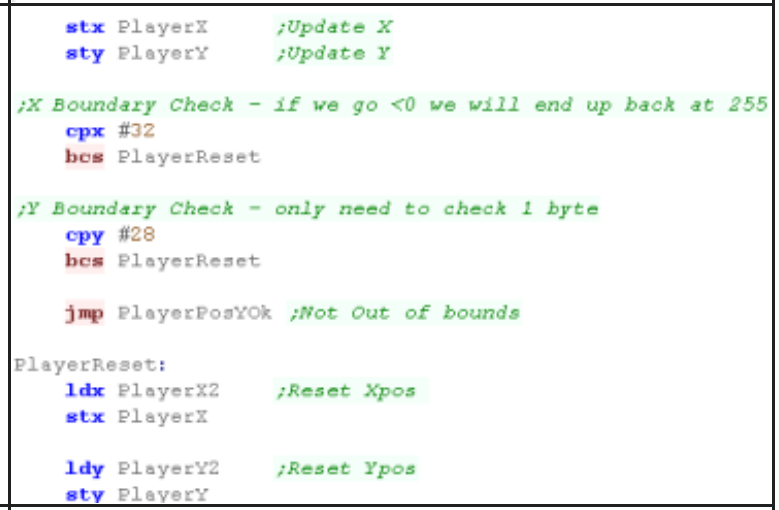
We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
lda #3          ;Start SX
sta PlayerX
lda #3          ;Start SY
sta PlayerY

lda #0          ;Fake No Keys on first run
sta z_h
jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

	
We test each of the bits in z_h (which was returned from the joystick routine) In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity	
We save the new XY position. Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255) Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup	
We're finished, so we draw the player to the screen, and wait a while! Finally we loop back, and start the procedure again!	

```
PlayerPosYOk:
    jsr DrawPlayer   ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY      ;Wait a bit!
    jmp infloop

PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```

Lesson S18 - Joypad Reading on the PC Engine/TurboGrafx-16 Card

The PC-Engine uses a single port with 4 bits to send the joypad buttons - it also supports up to 5 joypads via a multitap.

We'll need to reset the multitap, and read in Joypad 1 - Lets get started.

PCE_Joystick.asm

Defining Parameters, and drawing sprites

First we're going to need some zero page bytes to define an X,Y position.
We'll need one pair for the current X,Y position
We'll need a second pair for the last position - We'll use them to keep the player onscreen if the player goes out of bounds

```
;Current player pos
PlayerX     equ $60
PlayerY     equ PlayerX+1

;Last player pos (For resetting sprite)
PlayerX2    equ PlayerX+2
PlayerY2    equ PlayerX+3
```

We're going to use a slightly modified version of the bitmap routine from last time:

This time there are two versions - one will draw the smiley, the other will draw a blank sprite to clear the last position of the player.

We're defining tile 128 as our smiley... 129 is 32 zero bytes.

```
BlankPlayer:
    lda #129          ;Tile Num
    jmp DrawSprite
DrawPlayer:
    lda #128          ;Tile Num
DrawSprite:
    pha
        stx z_b
        sty z_c
        jsr GetVDPScreenPos ;Calculate Tilemap mempos
    pla
    st0 #2                ;Set Write Register
                          ;Save the TileNum to Vram
    sta $0102             ;L Byte
    st2 #1                ;H Byte - Tile 256+
    rts

Bitmap:
;Smiley
    ;   00000000  11111111      - Bitplane 0/1
    DB %00111100,%00000000      ; 0
    DB %01111110,%00000000      ; 1
    DB %11111111,%00100100      ; 2
    DB %11111111,%00000000      ; 3
    DB %11111111,%00000000      ; 4
    DB %11011011,%00100100      ; 5
    DB %01100110,%00011000      ; 6
    DB %00111100,%00000000      ; 7
    ;   22222222  33333333      - Bitplane 2/3
    DB %00000000,%00000000      ; 0
    DB %00000000,%00000000      ; 1
    DB %00000000,%00000000      ; 2
    DB %00000000,%00000000      ; 3
    DB %00000000,%00000000      ; 4
    DB %00000000,%00000000      ; 5
    DB %00000000,%00000000      ; 6
    DB %00000000,%00000000      ; 7

;Empty Sprite
    ds 32,0
BitmapEnd:
```

*We're not going to look at the 'screen init' routines, as we've already covered them in the simple series... check that out **here**.*

Reading the Joystick

We need to reset the 'multitap' hardware by sending 1,3 to port $1000

We then send a 1 - read in 4 bits... and a 0 and read in 4 bits

This returns a byte in the format:
Run / Start / B / A / Left / Down / Right / Up

```
Player_ReadControlsDual:
    ldx #%00000001          ;Reset Multitap 1
    jsr JoypadSendCommand
    ldx #%00000011          ;Reset Multitap 2
    jsr JoypadSendCommand

    ldx #%00000001
    jsr JoypadSendCommand   ;----LDRU (Left/Down/Right/Up)
    jsr JoypadShiftFourBits
    dex
    jsr JoypadSendCommand   ;---RSBA (Run/Start/B/A)
    jsr JoypadShiftFourBits
    rts

JoypadShiftFourBits:        ;Shift RSBA in to z_as
    ldy #4
JoypadShiftFourBitsB:
    ror
    ror z_h
    dey
    bne JoypadShiftFourBitsB
    rts

JoypadSendCommand:
    stx $1000           ;Set option from X
    PHA                 ;Delay
    PLA
    NOP
    NOP
    lda $1000           ;Load result
    rts
```

Our test program

We're going to define the starting position of our character - z_h will store our joystick buttons - we initialize this to zero - as we need to run the main loop but don't want to cause any keypresses.

```
lda #3          ;Start SX
sta PlayerX
lda #3          ;Start SY
sta PlayerY

lda #0          ;Fake No Keys on first run
sta z_h
jmp StartDraw   ;Force Draw of character first run
```

Next comes the joystick reading routine - we skip this the first run

We read in from the joystick into z_H - until a key is pressed we'll wait in an infinite loop... when it is pressed we start our drawing routine...

First we back up the last position of the player and clear the old player sprite from the screen.

```
infloop:
    jsr Player_ReadControlsDual
    lda z_h
    beq infloop      ;See if no keys are pressed

StartDraw:
    ldx PlayerX      ;Back up X
    stx PlayerX2

    ldy PlayerY      ;Back up Y
    sty PlayerY2

    jsr BlankPlayer ;Remove old player sprite
```

We test each of the bits in z_h (which was returned from the joystick routine)

In each case, if the bit is 0 then the button isn't down - and we skip over the routine to move the character's X or Y position

Because of the screen layout we're going to move vertically in 8 pixel blocks for simplicity

```
    lda z_h
    and #%00000001   ;RSBALDRU
    bne JoyNotUp     ;Jump if UP not presesd
    dey              ;Move Y Up the screen
JoyNotUp:
    lda z_h
    and #%00000100   ;RSBALDRU
    bne JoyNotDown   ;Jump if DOWN not presesd
    iny              ;Move Y Down the screen
JoyNotDown:
    lda z_h
    and #%00001000   ;RSBALDRU
    bne JoyNotLeft   ;Move X Left
    dex
JoyNotLeft:
    lda z_h
    and #%00000010   ;RSBALDRU
    bne JoyNotRight  ;Move X Right
    inx
JoyNotRight:
```

We save the new XY position.

Next we check if the player has gone off the screen... as the top left of the screen is 0,0 if the player goes off the left hand side (<0) they will return at the far right (255)

Therefore we only need to check the top boundaries!... we check if the player has gone over the boundaries, if they have we reset the player position from the backup

```
    stx PlayerX      ;Update X
    sty PlayerY      ;Update Y

;X Boundary Check - if we go <0 we will end up back at 255
    cpx #32
    bcs PlayerReset

;Y Boundary Check - only need to check 1 byte
    cpy #28
    bcs PlayerReset

    jmp PlayerPosYOk ;Not Out of bounds

PlayerReset:
    ldx PlayerX2     ;Reset Xpos
    stx PlayerX

    ldy PlayerY2     ;Reset Ypos
    sty PlayerY
```

We're finished, so we draw the player to the screen, and wait a while!

Finally we loop back, and start the procedure again!

```
PlayerPosYOk:
    jsr DrawPlayer   ;Draw Player Sprite

    ldx #255
    ldy #100
    jsr PauseXY      ;Wait a bit!
    jmp infloop

PauseXY:
    dex
    bne PauseXY
    dey
    bne PauseXY
    rts
```