

THE ABSTRACT ASSEMBLY DEVELOPMENT KIT



AN ASSEMBLY LANGUAGE PROGRAMMING FRAMEWORK
FOR THE
ATARI 2600 VIDEO COMPUTER SYSTEM

CONTENTS:

OVERVIEW AND CONCEPT

INTRODUCTION	3
--------------------	---

USING THE ASDK

HOW TO USE THE ASDK	4
ASDK GRAPHICS AND SOUND	5
COMPILING YOUR ASSEMBLY WITH DASM	6

EXPLORING THE ASDK

LEARNING 6502 ASSEMBLY EASY AND FAST	7
REVIEWING THE SCROLLDRAW.ASM EXAMPLE	8
DISCUSSING THE ABSTRACT CALLS IN THE EXAMPLE	10
ABSTRACT PLAYER AND MISSILE SPRITES	11
SPRITE LIBRARY AND ANIMATION EXAMPLES	12
GRAPHICS MEMORY MAP	13
POSSIBLE USES OF bB WITH THE ASDK	14
ADVANCED ASSEMBLY PROGRAMMING	15

NEW ENHANCEMENTS TO THE ASDK FOR 2013

ADDITIONAL VARIABLES	16
ASDK MEMORY MAP (ROM)	16
NEW GRAPHICS FUNCTIONS	16
VIDEO GAME EXAMPLE	17
FLICKER FREE MODE WITHOUT THE ARM CHIP	18

WHAT'S NEXT FOR THE ASDK

FUTURE ENHANCEMENTS	18
GAMES ON TAPE SUPERCHARGER EDITION	19
SUPERCHARGER MEMORY MAP	21

ADDITIONAL PROGRAMMING RESOURCES

LEARN ASSEMBLY WITH bB AND THE ASDK	22
---	----

Introduction to the ASDK

The ASDK is an *Assembly Language Programming Framework* for the Atari 2600 that provides the equivalent of Hardware based horizontal scrolling and a screen buffer! It also abstracts and eliminates Racing The Beam (scanlines) and timing/cycle counting and flipping the portions of the screen backwards and forwards (large WYSIWYG bitmaps are supported) so the developer can focus just on the logic for their Assembly games and leverage high level calls to the phantom hardware.

This Framework is an excellent resource for advanced Batari BASIC developers to rapidly gain a working knowledge of Assembly language programming; you just have to start writing code to learn Assembly but it's daunting and near impossible to learn the chipset and concepts such as programming without a screen buffer let alone horizontal scrolling without hardware support (when I learned Assembly in the 80's I had these things all available in hardware, the 2600 is quite a unique beast!)

Anyone may use the kit however they wish (no strings attached), please enjoy!



- Mr SQL,
- RelationalFramework.com

How to use the ASDK

Using the ASDK is simple; your Assembly language game (or demo) loop just goes between here:

```
;-----  
; ----- Your Abstract Assembly code goes here:  
;-----  
GameLoop  
And here:  
;--- end GameLoop  
; ---- Resume Framework
```

Just like the game loop you would have in bB, your game loop will repeat every frame allowing you to create very fast games and demo's – Assembly is lightning fast!

There is also an initialization section just as with bB where you can add any initialization code you want to run before entering your game loop:

```
;-----  
; init section:      --  
;-----
```

ASDK Graphics and Sound:

The table MyAbstractExtendedPlayfield, holds a large panoramic WYSIWYG bitmap, very similar to a bB playfield bitmap. Just find that section and edit it to create your large scrollable virtual world:

```
MyAbstractExtendedPlayfield ; 20x10 grid (3 bytes) is read from a larger play area 4x as wide (12 bytes)
;
; 1 0 4 2 12 3 20 4 20 5 36 6 44 7 52 8 60 9 68 72* 10 76 11 84 12
;
.byte <00001001, <11111111, <11111111, <00001111, <11011111, <11011111, <00001111, <11111011, <01000101, <01000111, <01000100, <01100111
.byte <00001100, <00000000, <01111011, <11111111, <11001111, <10101111, <00001111, <11111011, <01011011, <01010111, <01110111, <01010111
.byte <00001000, <11101111, <00000001, <11001111, <10000000, <00110111, <00001111, <11111000, <01000101, <01010111, <01000100, <11000011
.byte <00001111, <00000001, <11111011, <00001111, <11001110, <11111011, <00001111, <10011011, <01011011, <01010111, <01011111, <01110111
.byte <00001000, <10111101, <00000001, <11000111, <11111101, <11111101, <00001111, <01111011, <01000101, <01000111, <01000100, <01110111
.byte <00001000, <10111101, <01111011, <00001011, <11111011, <11111110, <10001110, <11111111, <11111111, <11111111, <11111111, <11111111
.byte <00001011, <11000001, <00111011, <11001101, <11110111, <11111111, <01001101, <01001111, <10111011, <11011111, <11111111, <11110001
.byte <00001011, <11011100, <01011011, <11001110, <11110111, <11111111, <10101011, <01001101, <10111010, <11011011, <01101110, <00111011
.byte <00001000, <00000001, <00000011, <00001111, <01011111, <11111111, <01010111, <01001101, <00010010, <10011011, <01100110, <00111011
.byte <00001111, <11111111, <11111111, <00001111, <10111111, <11111111, <00101111, <01001001, <00010010, <10011001, <01000010, <00111111
```

The large virtual world is 92x20 pixels, and the screen display is 20x10 phat retro pixels; you can pan anywhere in the virtual world with simple calls to the phantom hardware and also change any of the pixels (it's in RAM); this will be illustrated in the next section, *Exploring the SCROLLDRAW.ASM example*. This first virtual world is loaded on startup, but additional virtual worlds may be created and they may be loaded anytime - some templates have been left in from the SCROLLOUT.ASM example, which uses three different virtual worlds.

As you will see in the ensuing chapters, sprite data is also stored in intuitive WYSIWYG format like the large virtual worlds.

To add a musical score to your game or demo, find and edit the MusicData table:

```
-----
; Music Data -----
;
;-----
; volume0,wave,freq,volume1,wave,freq,framesduration <0 duration loops it>
;-----
MusicData
.byte 14,6,8,14,6,11,10
.byte 14,6,29,14,6,20,80
.byte 7,6,7,14,6,20,10
.byte 3,6,29,3,6,20,15
```

Compiling your Assembly with DASM

To compile your Assembly output with DASM save the file in your sample directory or wherever you usually compile your bB programs from, you already have all the tools you need:

```
C:\ATARI2~1\bB\samples>dasm scrolldraw.asm -f3 -o$scrolldraw.bin
DASM V2.20.07, Macro Assembler (C)1988-2003
Complete.
C:\ATARI2~1\bB\samples>
```

That's it; you should be able to compile the scrolldraw.asm sample contained within the ASDK source code release and any programs you create with the ASDK the same way - obviously you would change the target to *myprogram.asm* and the *-o* output flag to *-omyprogram.bin* or whatever you've named your saved program and wish to compile it as.

Learning 6502 Assembly Easy and Fast

Update: The best resource for rapidly learning Assembly Language Programming for the Atari VCS is the book or online tutorial *Learn Assembly in 8 hours on the 2600 with bB and the ASDK*, see page 17 for details.

Machine Language for Beginners is also an excellent resource for rapidly learning Assembly Language programming with the ASDK; because the ASDK provides phantom hardware you can address with high level calls, you do not have to do the equivalent of “learning BASIC or C by starting with three dimensional Array’s” – very few programmers would ever learn BASIC or C if that was a requirement!

Consider that some or all of the phantom hardware provided by the ASDK is actually present on the systems that most of the developers programming ASM on the 2600 actually learned on!

You *learn Assembly* by actually writing code, so now that you can focus on conditional logic, loops and conditions you will grasp the Assembly Language concepts presented in these books very quickly indeed:

<http://atariage.com/forums/topic/210550-learn-assembly-in-8-hours-with-bb-and-the-asdk/>
<http://www.atariarchives.org/mlb/>

Important concepts to learn about are the Stack and the Register objects (note that the Accumulator is also a register), particularly the condition code register – follow the book, for now you need only concern yourself with a few of the cc flags and the allusions to similar constructs in BASIC are excellent – you’ll be surprised, it’s not that tough because you already know it!

I also recommend learning how to translate from binary to decimal and optionally, from binary to hexadecimal with a pencil and paper; hexadecimal is largely irrelevant but understanding binary is crucial because it will give you a good foundation for understanding bitwise operations.

Reviewing the SCROLLDRAW.ASM example:

The SCROLLDRAW.ASM example is contained inside the ASDK as illustrated under *How to Use the ASDK*, we need only examine the Assembly code for that section to see how it works, the high level calls to the phantom hardware are all commented:

```
;-----
;----- Your Abstract Assembly code goes here:
;-----
;-----SCROLL Draw:
;-----Simple Demo for the Abstract Assembly Development Kit
; Hold the Button Down to Draw or Erase
;
;-----

; moved joystick? Then Move the Draw Cursor

lda #$10 ; 16 or 00001111
bit SWCHA ; like AND but we drop the result in the bit bucket behind the CPU
bne skipup
lda bity
beq skipup ; don't decrement bitx further if it is at zero
dec bity
skipup

lda #$20 ; 26 or 00011010
bit SWCHA
bne skipdown
lda bity
cmp #19
beq skipdown ; don't increment it past 19
inc bity
skipdown

bit SWCHA
BVS skipleft
lda bitx
beq skipleft ; don't decrement the x position value past zero
dec bitx
skipleft

bit SWCHA
BMI skipright
lda bitx
cmp #91
beq skipright ; virtual world is 92 pixels wide
inc bitx
skipright
;-----End Drawing Routine

;-----
;Follow the Cursor with the Camera:

; Camera X Axis: -----
lda bitx
cmp #10
bcs followXaxis ; >=10
lda #0
sta BITIndex ; (X position for camera is 0 - nothing to scroll to yet)
jmp donefollowXaxis
followXaxis
sec ; set carry for subtracts, clear it for additions
sbc #10
```



```

cmp #72
bcc dontadjustxaxis ; <72
lda #72 ; adjust x axis so code to the right of the virtual world is not shown on screen
dontadjustxaxis

sta BITIndex

donefollowXaxis

; Camera Y Axis: -----

lda bity
cmp #6
bcs followYaxis ; if >=6
lda #0 ; keep Y pos for virtual world at the top for the top 6 pixels
sta YIndex ; superfluous
jmp offsetdone

followYaxis
lda bity
sec ;set carry for subtracts, clear for additions
sbc #6
sta YIndex
donefollowYaxis

cmp #10
bcc dontadjust ; <10
lda #10
sta YIndex ; don't show the code below the virtual world; this sets the bottom border
dontadjust

; Now turn Y Index into a Row Offset for the bitmap (it's 12 bytes across)
lda #0
ldy YIndex
offsetcalc sec ; set carry before additions
clc
adc #12
dey
beq offsetdone
jmp offsetcalc
offsetdone
sta BYTERowoffset

;-----
;---- Done Following the draw cursor with the camera

;--- Turn on the target bit in the virtual world:

lda #1 ; argument to turn pixel on always
jsr getbitstatus ; call subroutine, passing argument
;---- Done Turning on the target bit

;-----Erase the target bit under the cursor?
;--- if button is pressed, erase it:
bit INPT4
bmi skiperase
lda #0 ; argument to invert bit, since it was just set it will be turned off
jsr getbitstatus ; call subroutine, passing argument
skiperase
; note: The getbitstatus subroutine can get, set, invert or clear a bit
;-----

;-- Now Call the Twin Engines that emulate hardware level
;-- Horizontal Scrolling and Scaling
;-----slide view window along bitmapped panorama currently loaded into CBS RAM:
jsr pushabstractextendedplayfield
;----- expand and flip 30 bytes of system RAM buffer into 60 Bytes for display:
jsr AbstractPlayfieldBuilder

```

```
;-----  
;-----End Scrolling Draw Demo  
  
; --- Resume Framework
```

Discussing the Abstract Calls in the Example:

getbitstatus can set, unset, flip or return the status of a bit anywhere in your large virtual world depending upon the argument passed or returned by the function/sub (the A register/accumulator is used to pass and return arguments).

Note that the calls to *pushabstractextendedplayfield* and *AbstractPlayfieldBuilder* are the equivalent of DrawScreen in bB and address the phantom hardware for screen buffering, display and hardware scrolling, and are called just once, usually at the end of your gameloop; you can change the graphics anywhere in your large virtual world with *getbitstatus* after calling them, but the changes won't be visible until the next frame (sometimes this is useful).

Abstract Player and Missile Sprites

The ASDK provides abstract support for the player and missile sprite objects much like Batari BASIC:

player0y, player0x, player1y, player1x

and

missile0y, missile0x, missile1y, missile1x

To instantiate a sprite, just set it's x and y variables and it will appear; setting them both back to zero will collapse the sprite.

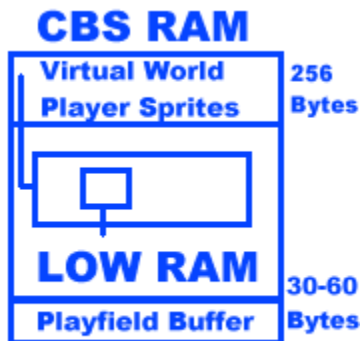
Sprite Library and Animation Example

The ASDK allows the Assembly developer to write an inline sprite library in WYSIWYG format, just like the WYSIWYG format used for the large virtual worlds!

Like the large virtual worlds, sprite definitions are loaded from ROM into high RAM to render them malleable. The first two sprite images are automatically assigned, but you can reassign the image with a simple call to the sprite loaders, or by modifying the RAM directly for more interesting effects as illustrated in the demo.

Note: Keep in mind the sprite definitions are loaded *upside down* into high RAM if you intend to work with them there.

GRAPHICS MEMORY ALLOCATION:



Large Virtual World and Sprite bitmaps loaded into high RAM:

Large panoramic WYSIWYG bitmaps are loaded from ROM into 240 bytes of CBS RAM; the remaining 16 bytes of high RAM is used to hold the current sprite definitions.

Playfield Double Buffering in low RAM:

The panned playfield (anywhere within the panoramic) is a 30 byte segment (20x10 phat retro pixels) loaded into low RAM still in WYSIWYG format by the primary graphics engine, the call to the secondary engine upsizes those 30 bytes into 60 bytes and flips them; it is possible to modify them after and between passes. This can be useful if you want the changes to be temporary (just for that frame or until you scroll) or permanent in the underlying virtual world.

The complex sprite animation demo that scrolls the scrolling playfield through the sprite runs between calls to the primary and secondary graphics engines in order to access the data in low RAM.

Possible uses of bB with the ASDK

It should be possible to use Batari BASIC with the ASDK instead of Assembly if you wish; you could create your game or demo loop logic entirely with bB, using the variables a-q (they are supported by the ASDK, though a through e are temp vars that will be overwritten if you do not push them into the stack). Then compiling it, and pasting the Assembly output for that section into the ASDK as specified.

Of course you would have to manually add the calls to the phantom hardware but they are abstract – you would need to know how to code in bB and how to at least make minor changes in asm for this approach.

See the *future enhancements* section for more about bB integration.

Update:

It is possible to use the batari BASIC compiler to generate Assembly code for use with the ASDK if preferred; please see the tutorial or the Book referenced on page 17 for details and examples.

Advanced Assembly Language Programming

Advanced Assembly Language developers may be interested in exploring the Framework – there's a lot of interesting bitwise operations going on; some are optimised and some not so much as the approach used is geared to provide plenty of cycle space for the developer.

I favour heavy use of the stack but there are sections where I pulled out stack usage and switched in variables during debugging and never revised back ;)

You will also see I'm a big fan of traditional 80's Assembly and like really long naming conventions; I don't care much for macro's or other neat enhancements that have been added on to editor/assemblers since then, I agree they are quite valuable it's just a matter of preference.

My tool of choice in the 80's was EDTASM+ for the 6809 and I think the 6502/7 is tremendous fun for it's comparative RISC architecture; I still find myself reaching for the plethora of 6809 branch instructions and their long branch equivalents, BRN and LBRN my favourites ☺

I look forward to your comments and ideas about this Framework!

New Enhancements to the ASDK for 2013

ADDITIONAL VARIABLES

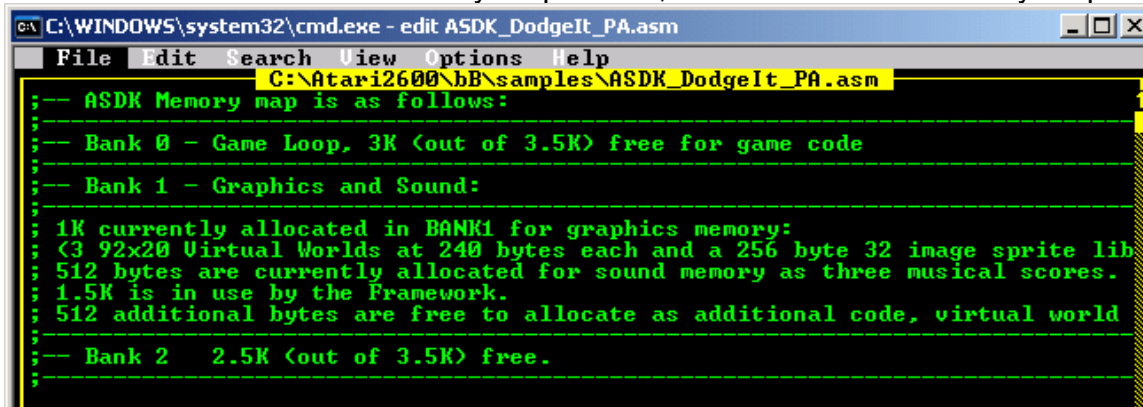
The ASDK now supports the following variables:

a-z (a-e are temp vars with e used to surface sound effects in the music engine).
Additional variables: var1, var2, bx, by, px, py.

20 additional 4-bit variables (use function `set4bitvar` to typecast).

ASDK MEMORY MAP (ROM)

We've looked at the RAM memory map earlier, here is the ROM memory map:

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - edit ASDK_DodgeIt_PA.asm". The window shows the contents of the file "C:\Atari2600\bB\samples\ASDK_DodgeIt_PA.asm". The text in the window is as follows:

```
-- ASDK Memory map is as follows:
-- Bank 0 - Game Loop, 3K (out of 3.5K) free for game code
-- Bank 1 - Graphics and Sound:
; 1K currently allocated in BANK1 for graphics memory:
; (3 92x20 Virtual Worlds at 240 bytes each and a 256 byte 32 image sprite lib
; 512 bytes are currently allocated for sound memory as three musical scores.
; 1.5K is in use by the Framework.
; 512 additional bytes are free to allocate as additional code, virtual world
-- Bank 2 2.5K (out of 3.5K) free.
```

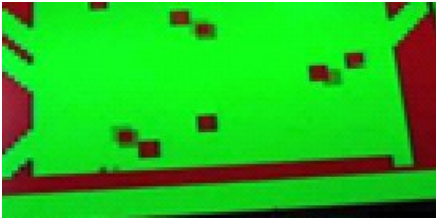
Note that out of the total 10.5k of ROM (3.5k per bank) up to 7.5K can be made available for game code, graphics and musical scores.

NEW GRAPHICS FUNCTIONS

New Graphics functions have been added, see the example video game in the next section for a hands on illustration of how to use these and other ASDK functions.

VIDEO GAME EXAMPLE

Dodgelt Panoramic Adventure



Dodgelt Panoramic Adventure is an Assembly Game written using the 11/2013 release of the ASDK. You can download the source code along with the latest version of the ASDK [here](http://atariage.com/forums/topic/203463-abstract-assembly-development-kit/page-2#entry2858080):

<http://atariage.com/forums/topic/203463-abstract-assembly-development-kit/page-2#entry2858080>

The Beta is currently 2K of Game code (all in the Game Loop) and illustrates using the additional functions:

The Pac-Man character is calculated using virtual world coordinates (92x20) just like the virtual world block characters and the sprite is mapped to the screen using the *findspriteXYfromvirtualworld* function and then smoothly animated from one coarse position to the next based on target direction. The Vertical sprite flipping function is used as well as Horizontal flipping.

The effect is that sprite characters can also roam around off screen and interact with the virtual world!

The release version of *Dodge It Panoramic Adventure* illustrates the interesting effects of several characters interacting both on the screen and off in the virtual world:



You can download the game binary to play on your Atari [here](#).

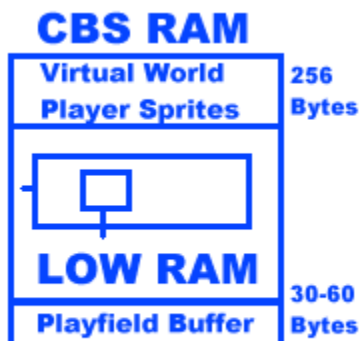
FLICKER FREE MODE WITHOUT THE ARM CHIP

Flicker Free mode is now available without the ARM chip via the ASDK's unique dual kernel design:

For details on utilising flicker-free mode, please refer to the subchapter *disengaging the Primary Kernel for a flicker free display* in Lesson 7, *Getting The most out of the ASDK* in either the online tutorial or the book *Learn Assembly in 8 hours on the 2600* (see page 20).

Future Enhancements

Possibly integrating the scrolling engines for the large virtual worlds into the bB DPC+ kernel; these engines are processor intense but they could run flicker free on the ARM chip with no contention for the bB framework overhead. Even better, the power of the ARM and the additional RAM could support a much higher res playfield and virtual world than the 6507 and double super chip:



SUPERCARGER EDITION

Make Atari games on Tape!



The StarPath SuperCharger Edition of the ASDK has been condensed and feature packed to fit into the spare memory bank of the 6K SuperCharger! This leaves nearly 4K available which is quite a lot of room for game development.

SuperCharger RAM

**1st 2K of address space
is always present**

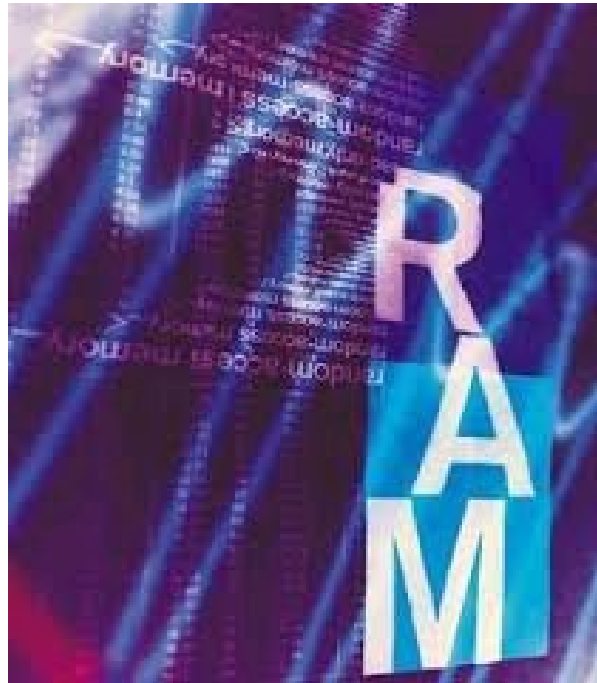


About SuperCharger Static RAM

SuperCharger RAM is *static RAM* which has some unique space saving properties – you can define it inline in your code. This allows for pre-loaded variables: the 20 4-bit vars and the optional extra block of variables are all static RAM.

The Virtual World is also in static RAM to maximise memory space. The vertical flip function for sprites also conserves additional space.

Because the SuperCharger has the option of addressing more than the 256 bytes of extra RAM present in the CBS RAM version, additional static RAM variables may be allocated.



SUPERCHARGER MEMORY MAP

```
-----  
;--- SuperCharger Memory Map:  
-----  
;--- The ASDK has been condensed to fit in the SuperChargers spare 2k bank  
-----  
;--- About 4K of memory is available for game code, graphics and sound:  
;--- bank 1 is entirely free for the game loop and bank 3 has 1310 bytes free  
;--- bank 2 has 165 bytes free.  
;--- This adds to 3.5K so you're wondering, where is the other 1/2k?  
;--- There are 128 bytes used in bank 2 for the inline sprite library;  
;--- this can be expanded or reduced in size, you can also move it to bank 3.  
;--- And, you have 384 bytes of RAM  
-----  
;---  
;--- Need more RAM? ;>  
-----  
;--- If you want more than 50 variables uncomment the superchargervar routines  
-----  
F1=Help | Line:34 Col:11
```

The SuperCharger Edition makes 4K of free memory feel roomy, more like 16k or 32k because of the high level Framework objects you are marshalling and the space saving features of SuperCharger static RAM!

Reload that game!

Because SuperCharger RAM can be initialised inline the Virtual World comes preloaded in RAM. The three predefined virtual world images in ROM are gone; you can add additional virtual worlds to load but they take up 240 bytes each.

If you don't want to allocate space for at least one 240 byte ROM image to reload the virtual world (provided it has changed during the game and needs to be reloaded) one option is to require the player to reload the game should you need to use every last bit of memory to build a more awesome game.



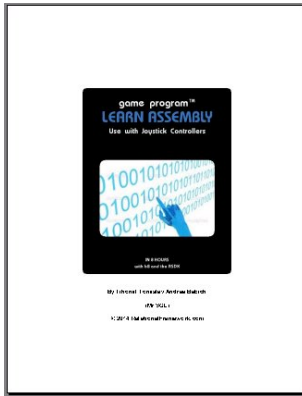
It's fun pressing play on tape & throwing switches on the console! ☺

Learn Assembly Language in 8-Hours with bB and the ASDK

The online tutorial is available at AtariAge:

<http://atariage.com/forums/topic/210550-learn-assembly-in-8-hours-with-bb-and-the-asdk/>

The Book contains an enhanced version of the online tutorial with clearer text and images and exclusive bonus chapters for retro gamers with an inside perspective on writing Video Games in the 80's:



- Strategies for selling your Video Games to magazines and software companies in the 80's.
- Bootstrapping your own Video Game company in the 80's.
- Colourful full page magazine ads for the authors 80's Video Games.
- Video Game Marketing models from the 80's.
- Insider stories and pictures from the 80's.
- Colourful video game artwork for the authors 80's Video Games inspired by Atari!

If you want to learn 6502 Assembly in 8 hours or just like to read about retro, don't miss out on this awesome fun book! ☺

The price is \$10 for the eBook, please email Support@RelationalFramework.com to order. Also included is a special publishers deal for anyone who wishes a printed version of the book for an additional \$10.