DTV Documentation Project

Issue #1: RevC
2006-11

By *gmoon*

Contents:

i. Segment Mapper, Part 1
ii. The Register File
iii. Segment Mapper, Part 2
iv. More on the Register File

Appendix A: List of segments

This document wouldn't be possible without many contributions from the DTV community. A certain familiarity with (not expert knowledge of) 65xx assembly language is assumed.

This is a supplement, not a replacement for the ***DTV Programming* Guide**—of which you should print a hardcopy. ☺

Segment Mapper, Part 1

The *Segment Mapper* is used to map memory blocks, including those with addresses greater than 64K, into a location usable by the 65xx processor. The base 64K addressing space ($0000-$FFFF, all that the processor can 'see') is split into four 16K ***banks***, each of which can be mapped separately.

65xx addressing space:	$0000-$FFFF	0-65535	(64K)
DTV total memory:	$000000-$1FFFFF	0-2097151	(2 Megabytes)

Layout of *Segment Banks*:

Bank #	65xx addr	reg #
Bank0	$0000-$3FFF	REG 12
Bank1	$4000-$7FFF	REG 13
Bank2	$8000-$BFFF	REG 14
Bank3	$C000-$FFFF	REG 15

Altering the value of a *Segment Bank* register (Column 3), effectively “remaps” a *different* block of memory into the associated 65xx addressing space (Column 2).

There are **128 possible segment mappings** per register, each 16K, counting from zero.

Important Note:
All memory accesses use the base 65xx $0000-$FFFF 16-bit addressing space. (There are exceptions, such as DMA or blitter. But they don’t use the 65xx addressing modes.)

Each of the *segment banks* can be 'filled' with the image of RAM (or ROM) that lie anywhere within the larger addressing scheme. The processor doesn't go UP to access that memory (it can't) --the segment mapper brings the high memory DOWN, 'mapping' it into the standard addressing space.

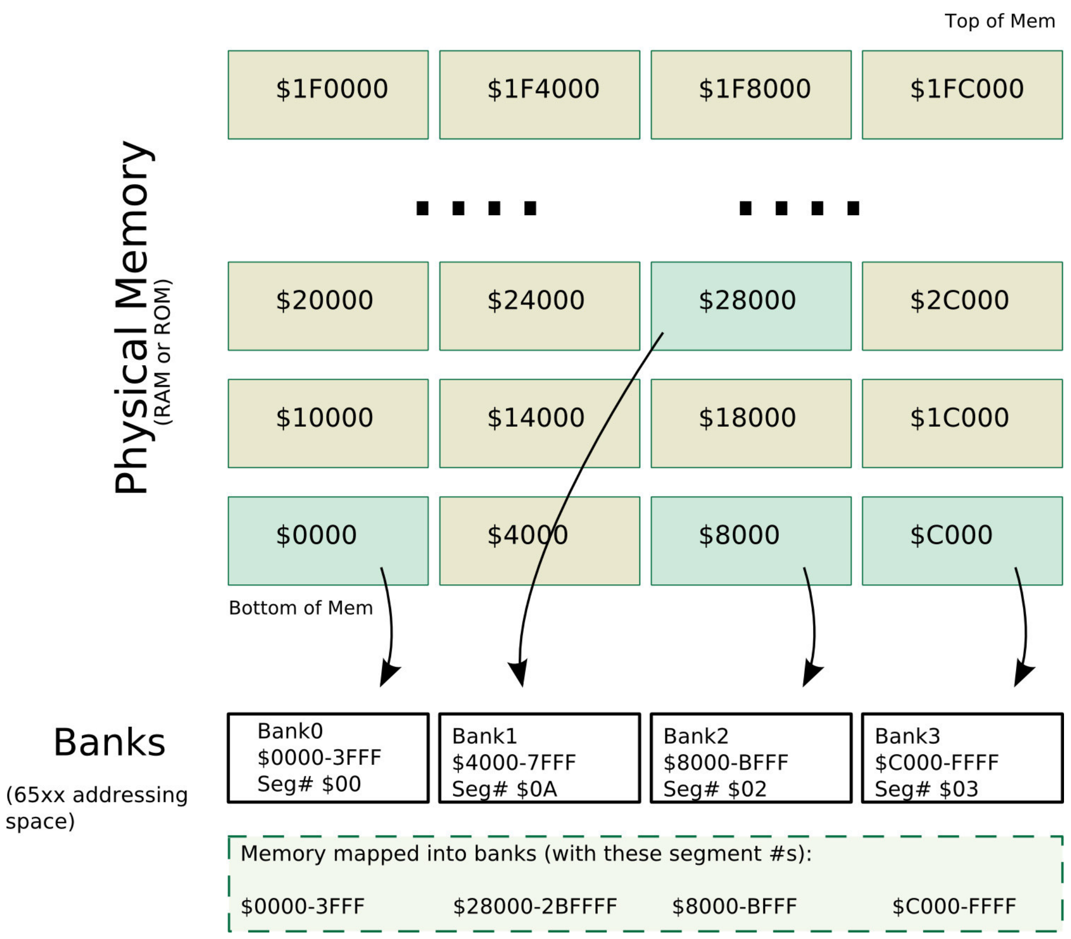
Segment Banks are a separate concept from *segments--banks* are fixed, always in the same location in base 65xx memory. *Segments* (the memory itself) are swapped in-and-out of those banks. Both are 16K in size, as the bank is designed to hold the segment. When the *Segment Mapper* is set correctly, high memory simply ‘appears’ (remapped) in one of the four 16K banks. For the duration of the remap, that lower memory **IS** the high memory.

Think of the *banks* as 'virtual ports', and the *segments* as 'virtual cartridges' of RAM or ROM.

The banks are always set to a segment: the default mapping is $00, $01, $02 & $03-- or Bank0 = $0000, Bank1= $4000, Bank2= $8000, etc.... In theory, you could map the same 16K segment into all four banks. Not that you ever would….

All the standard opcodes for the 65xx will work within the remapped segments. Code can't execute in high memory unless it is remapped low—the addressing modes of the 65xx cannot handle the 24 bit addresses. Without the *Segment Mapper*, the processor essentially couldn't 'touch' that memory.

As noted, the DMA or blitter can access that memory—but they are 'outside' of the functional 65xx processor.



Illus 1.1)

An image of the ***Segment Banks****, with all banks set to default except* Bank1*, which is mapped to segment #$0A, the memory segment @ $28000.*

With this mapping, a PEEK at $4000 would PEEK the memory at $28000. POKEing $4001 would POKE $28001.

The Register File

Before using the *Segment Mapper*, the ***Register File*** concept and operation must first be understood.

The *Register File* is a set of extended registers which are accessed using special DTV-only opcodes. The ‘file’ designation tends to trip people up—it’s just a set of registers, called a ‘file’ because each register must be ‘loaded’ into a destination register. Don’t let the name throw you.

Keeping with the 65xx paradigm, all the registers in the *Register File* are one byte in length (8 bits.)

Register File (source) **registers**:

Name	**Reg#**	**Function**
Reg0	0	general purpose (default mapping-- ACC register)
Reg1	1	general purpose (default mapping-- Y register)
Reg2	2	general purpose (default mapping-- X register)
--	3	*reserved*
--	4	*reserved*
--	5	*reserved*
--	6	*reserved*
--	7	*reserved*
Access mode	8	RAM or ROM for banks
CPU control	9	Special CPU modes (burst, skip cycles)
Base Page	10	Remap Zero Page
Stack Base	11	Remap Stack Page
Bank0	12	Segment Bank @ $0000
Bank1	13	Segment Bank @ $4000
Bank2	14	Segment Bank @ $8000
Bank3	15	Segment Bank @ $C000

DTV 65xx (destination) **registers**:

Name	**Function**
ACC	65xx accumulator
Y	65xx Y reg
X	65xx X reg

Important Note:
All the extended registers of the DTV use the available 65xx resources and modes. The 65DTV02 was designed to emulate a 65xx FIRST, and extended modes are an addition.

The beauty of this approach lies in the fact that the emulated 65xx processor is still operating as a 65xx--once you step away from that, there is no going back. Everything, registers, addressing modes, ALU, etc., etc. would need to be redesigned for extended (24 bit?) addressing.

To access the extra registers, however, some additional opcodes were added. Those codes which control the *Register File* access (and that only):

Mnemonic	Description	Opcode
SAC	Set Accumulator mapping	$32
SIR	Set Index Register mapping	$42

Usage (of SAC):

```
        SAC    #$DD
```

Usage (of SIR):

```
        SIR    #$1D
```

O.K.—how does this work, and what is happening? Let's examine the SAC and SIR instructions.

SAC

First, the SAC instruction: SAC, mnemonic for "**S**et **AC**cumulator mapping," is the opcode for assigning which register is ***mapped into*** the A register (destination or 'target' register, 65xx accumulator.)

To put it simply**: the SAC instruction transforms the ACC register into any of the registers in the *Register File*.**

SAC accepts one operand-- a byte in immediate mode, which represents:

	Hex	0 - F	0 – F
	Bits	[0000	0000]
		destination :	**source**

The high nibble is the destination, the low nibble the source. For general use, set both to the same register.

Example 1.1:

```
1       SAC   #$DD
2       LDA   #$04
3       SAC   #$00
```

- Line1 Sets the accumulator destination and source to the 13th ($D) register (bank1 of the *Segment Mapper.*)
- Line2 loads the number 4 into A register, and since A = REG 13, sets the *Segment Mapper* (Bank1) to 4.
- Line3 resets the A register to REG 0, or back to the 65xx accumulator.

SIR

The SIR instruction, short for "**S**et **I**ndex **R**egister mapping," is the opcode for assigning which registers are ***mapped into*** the 'target' X and Y registers.

The SIR instruction transforms both the X and the Y registers into any of the registers in the *Register File*.

SIR accepts one operand-- a byte in immediate mode, which represents:

```
        Hex     0 - F           0 – F
------------------------------------------------
        Bits   [0000           0000]
               Y register  :  X register
```

The high nibble represents the Y register, the low nibble the X register. Both can be remapped in a single instruction. Or one register at a time can be remapped by retaining the default mapping for the other.

Example 1.2:

```
1       SIR   #$1D
2       LDX   #$04
3       SIR   #$12
```

- Line1 sets the X register to the Segment Mapper (Bank1), REG 13. Y register retains default mapping of Y (REG 1).
- Line2 loads #4 into the X register, which is mapped as REG 13, and the Segment Mapper (Bank1) to 4.
- Line3 resets Y and X registers to normal Y & X mapping (1 & 2).

These two examples (1.1 & 1.2) **do exactly the same thing**. They set the *Segment Mapper* (REG # 13) to remap $010000 into *Bank1* ($4000-$7FFF.)

Once the registers are remapped, the special registers can utilize all the features (ALU, memory address modes, etc.) of the 65xx register they are mapped into. Example 1.1 illustrates a simple immediate load to the ACC, but any of the other modes are usable.

Although computation with the ALU (arithmetic logic unit—add, subtract, bitwise, etc. operations) in the ACC (and Y & X) is possible while remapped, general practice is to set a *Register File* register to a value, quickly restore the register mapping, utilize the behavior (the register retains it's new value, even after register mapping is reset to default) for a time and reset afterwards. However, ALU use with a remapped register could be very handy—accessing the memory slices in a 'bit-plane' style bitmap, for instance.

Important Note:
The *Register File* is always 'in play,' since ACC, Y and X are by default mapped to the first three entries (0, 1 & 2.)

The general-purpose registers (0, 1 & 2) are no different than the other more exotic members. In fact, in general operation, **ACC, Y and X behave as 'standard' 65xx registers only because they are mapped to *Register File* registers 0, 1 & 2, which have no extended function**. The 'destination' registers, ACC, Y and X are essentially *virtual registers*, and must be mapped to the correct entries in the *Register File*, even to function as 'normal' 65xx registers. ACC, Y and X are **always** mapped to entries in the *Register File* (there is no unmapped state for ACC, Y & X, although the *reserved* registers are effectively inactive.)

Also, the behavior of each register depends on the 'destination' register (ACC, Y, X) receiving the mapping, not the source (*register file*.) While this should be apparent from the SAC and SIR codes, take it one step further**: the three general-purpose registers, 0, 1 & 2 can be mapped into ACC, Y, and X *in any order***, and the DTV will operate normally.

However, for the sake of compatibility with existing software, it's best to retain the default order. The value of a *register file* register can be read, but not it's mapping.

If it's not already plain, let's recap:

- SAC will remap one of the special (*Register File*) registers into the ACC (accumulator.)
- SIR will remap special registers into both the Y and X registers.
- SAC is the more powerful instruction (or rather the ACC itself is more powerful,) as it can set the ACC to have separate destination and source registers.

Like most ML instructions, SAC and SIR have a very simple job.

Segment Mapper, Part 2

So clearly the *Register File* is tightly integrated with all DTV functions--but back to the *Segment Mapper*.

As stated, there are four separate *Segment Banks* (0-3) which together constitute the 'target' locations of the *Segment Mapper*. These banks, controlled by *Register File* registers 12, 13, 14 & 15, accept the 'source' memory mapping, in 16K blocks.

So let's look at a *Segment Mapper* register in detail.

The least-confusing *Segment Bank* to modify is ***Bank1*** (REG 13), which controls memory in the $4000-$7FFF space. This bank is easiest because it's not overlaid with any ROM images, like BASIC, Kernal or Character ROM.

The default value for ***Bank1*** is $01, in bit form [00000001]. The default mapping for ***Bank1*** is $4000 (the mapped RAM value is the base address space itself.) So if a value of 1 = the second 16K segment, then a value of 0 = $0000, or the first 16K segment. As expected, **the minimum resolution of the *Segment Mapper* is 16K**.

With 256 possible values of 16K, the highest possible segment is 3FC000 - 3FFFFF. But 3FFFFF indicates 4 Megs of memory, yet the DTV has only 2 Megs.

Why the extra bit of resolution? It could be simply ignored by the mapper, but it isn't. Only 21 lines are needed for a 2 Meg addressing, but the total address bus on DTV is 22 bit, theoretically capable of 4 Megs total. The 22nd line (A21—we count from zero) IS documented on the V2 schematic, and in the ***DTV Programming*** **Guide**. At this writing, no one has reported using it, however.

Without that unused high bit, **the total number of segments is 128**.

Segment Mapper **register layout (any one of four)**:

The entire register represents bits 22-14 of the address bus:

```
                    [00000000]
Address Bits           22-14
```

However, this can also be thought of as the high 6 bits represent bits 22 – 16; I.E., they control '*major*' banking blocks (of 64K), while bits 1 & 0 control '*minor*' banking – 16K blocks.

```
                    [000000      00]
Address Bits          22-16        15-14
```

Of course, if the 6 high bits are set to a value other than 0, the *minor* blocks will lie in memory above $FFFF.

Simple example:
Remapping memory with the *Segment Mapper*

For a simple illustration, let's reexamine example 1.1:

```
1       SAC     #$DD
2       LDA     #$04
3       SAC     #$00
```

Line1-- Remaps A register (dest & src) to REG 13 ($D), or *Segment Mapper*, Bank1.

Line2-- Setting the segment:
#$04 in bit form is [00000100]
Translated into the full 22-bit address (*seg mapper* bits in **bold**):

```
Address byte:       hi              mid              low
                [xx000001]  [00000000000]  [00000000] or $10000
```

$10000 = the very start of high memory (65536).
So a value of 4 remaps the memory at $10000-13FFF ***into*** the $4000-7FFF bank.
(The red bit is unused in a 2-Meg DTV.)

Line3-- Remaps A register (dest & src) back to REG 0, the A register. Generally, we want to do this. Otherwise, anytime we use the accumulator, the remapped segment will change!

When the SAC instruction returns the ACC to the default mapping, the *Segment Mapper* retains the last set value, and the altered memory mapping remains until it's explicitly changed, or the unit is reset.

While mentioned in section (i), it's important to reiterate what 'remapped' means in this context—the high memory blocks are 'moved' into the low block, where they can be acted upon by all the standard 65xx opcodes and addressing modes. ***All memory accesses happen in the base 64K of memory addressing space*** (excluding, as noted, special cases such as DMA, blitter, ROM mapping, etc.)

A more advanced example:
Using one *Segment Bank* to access 64K (four 16K segments)

Each of the 4 *Segment Mapper* registers can remap 16K of RAM. But it's often not practical to remap all 4 banks—messing with zero page and stack memory is usually a bad idea (although both are handled by other Register File registers, and can be remapped themselves.) Plus any executing code needs to remain in 65xx addressing space, although *that* memory can be remapped from higher address RAM, too.

Accessing all the memory in a full-screen ***8bpp*** image requires 64K of memory, the total available addressing space of a 65xx processor. "Eight-bits-per-pixel," means one byte = one pixel (a graphics mode available on the DTV, but not the C64.) A full-screen image in 8bpp totals 320x200 pixels, the same number of pixels as a ***1bpp*** ("one-bit-per-pixel") C64 bitmap. Consequently, an 8bpp screen is 8 times larger than a 1bpp screen (8K.)

But this is an apparent paradox. How can all that graphics memory—enough to fill up all RAM in a C64—be accessed when memory is needed to run the program which creates the screen itself?

Two salient points:

1) All of the 4 available banks can be remapped very, very quickly, and any bank can access any 16K segment of memory. No need to use four separate banks for our four 16K blocks (64K)—just use one or two banks, and change the segments as needed.

2) When memory is remapped with the *Segment Mapper*, the original memory is totally unchanged. It's not difficult to switch code in-and-out, so long as concerns such as interrupt routines are accommodated. After extended memory access is complete, the default memory map is easily restored.

NOTE:
For the sake of clarity, not all the code is present to run this example; functions such as creating the 8bpp screen, multiplication routines, etc. are not shown. The code can be found at the DTVhacking forum at petscii.com.
http://jledger.proboards19.com/index.cgi?board=dtvhacking

Code listing: **The PLOT routine**

```
PLOT
sec1
        LDA line_y       ; y coord * 320: Y in ACC
        JSR MULT320

        CLC              ; add X coord
        LDA MUL_RES
        ADC line_x
        STA MUL_RES
        LDA MUL_RES+1
        ADC line_x+1
        STA MUL_RES+1

sec2
        ROL
        ROL              ; get/set bank 15:14 bits from hibyte of xy math
        ROL
        ORA #$08         ; bank $20000
        AND #$0B         ; mask-out unwanted bits
                         ; bank (ACC) now (reg 13) to $020000 + (1 of 4 segs)
        TAX              ; set the segment mapper

sec3
        LDA MUL_RES+1    ; bitmap location:
                         ; unset bit 15, set bit 14 (bank always=$4000),
        AND #$7F         ;       bank itself will handle those bits
        ORA #$40
        STA MUL_RES+1

sec4
        LDA COL_A        ; load color
        STA (MUL_RES),Y  ; write to scrn. Y already 0 (in MULT320)

        RTS
```

The PLOT routine simply plots a pixel of a color value (COL_A) on an 8bpp screen located at $20000, utilizing the *segment mapper*.

Breakdown

- *Sec1* figures the memory address of the pixel from the X,Y input.
- *Sec2* sets the Segment Mapper
- *Sec3* alters the pixel location so it fits within the single bank
- *Sec4* sets the pixel to COL_A

Basically, this maps the four 16K segments starting at $20000 ($20000-$2FFFF) into ***Bank1*** ($4000-$7FFF). It steals the 2 hi bits (15:14) from the pixel address and uses them to set the current segment.

Once the *segment mapper* is set, bit 15 is cleared and bit 14 is set of the pixel address--so the pixel writes always fall between $4000 and $7FFF. Then the routine can safely store the pixel data in the remapped memory segment.

Breakdown, in detail:

Sec1
This is easy: the pixel address = Y * 320 + X
The result is in MUL_RES (two byte result), and the MSB of that address is in the accumulator.

Sec2 (red)

Our pixel address spans 64K, but we want to use one 16K bank. So we grab the two high bits, which represent 16K of memory addressing (if our banks were 32K in size, we would need only one.)

To set the current segment, we want bits [15:14] of the pixel address, and we want them where they are useful--in the lowest two bits of the *segment mapper* register (see Pt 2, the two lowest bits are bits [15:14] of memory mapping.) The quickest way to achieve this is simply to ROLL the high bits into the low bits (remember, the high byte of the pixel address is already in the ACC):

```
ROL
ROL
ROL
```

The ROL opcode rolls the contents of the ACC left, though the Carry Bit (hence three ROLs.) What was in the [15:14] bits, is now in the [1:0] bits. These will be the low bits [xxxxxx11] of the segment mapping.

That takes care of the *'minor'* part of the address (the 16K offset), but not the location of the screen itself. The *'major'* portion (high 6 bits) of the address of an 8bpp screen at $20000 is [000010xx], or $08. The 4th bit is set by:

```
ORA #$08                ;  bank $20000
```

To remove any remains from the original pixel address, any unwanted bits are cleared with the mask [00001011] used to discard the rest:

```
AND #$0B                ;  mask-out unwanted bits
```

Now, to actually set the *Segment Mapper*:

```
TAX                     ;  set the segment mapper
```

Huhhh? What good does that do?

Well, in order to speedup multiple plots, before PLOT is called the *segment mapper* for ***Bank1*** is mapped into the X register (this code is not included in PLOT; it's called separately):

```
        SEI             ; stop irqs
        SIR #$1D        ; set seg mapper reg 13 to X reg
```

For the duration of the PLOTs, the X register is functioning as the *Segment Mapper* Register #13 (reg for *bank1*.) The interrupts are disabled, since any use of the X reg by the IRQ routines would change the mapping for *bank1*.

So *Bank1* is set to a different segment simply by transferring the contents of the ACC into X reg. This means, of course, 'hands off' normal use of the X register for the duration of the PLOTs.

After all the PLOTs are done, the register (and the bank1 segment) is restored. And before resuming other tasks, re-enable the IRQs:

```
        LDX #$01        ; reset seg mapper reg 13 to default val
        SIR #$12        ; reset X (& Y) registers
        CLI             ; restore the interrupts
```

Sec3 (blue)

The segment bank is now set correctly for the pixel address. But the original address is for a pixel located in a 64K block of memory, not 16K. The high bits are not only unnecessary, they are now incorrect. Since the correct 16K segment is now mapped into Bank1, ***the pixel address is reset so all pixels fall within Bank1***. This is important--both to set the correct pixel and to prevent any pixel writes from falling outside the bank, overwriting other memory.

How can this work? Remember, the two high bits have already chosen into which of the 4 segments the pixel will be written. The lesser part of the address hold the correct memory location in any 16K segment. So the MSB of the pixel address is loaded in the ACC:

```
        LDA MUL_RES+1   ; bitmap location
```

For Bank1, the two high bits must represent the start of the bank, or $40 [01xxxxxx] (MSB of $4000.) First the 8th bit is cleared (the AND op), then the 7th bit is set (OR op.)

```
        AND #$7F        ; clear bit
        ORA #$40        ; set bit
```

And the modified address is stored.

```
        STA MUL_RES+1
```

Sec4

Also easy—
Load the current foreground color (COL_A), and store it in the modified pixel address, thereby setting the pixel. Return.

```
        LDA COL_A        ; load color
        STA (MUL_RES),Y  ; write to scrn. Y already 0 (in MULT320)
        RTS
```

That's it.

To wrap things up, think of the two hi bits moved to the *segment mapper* as a destination offset--choosing in which of the (4) 16K segments (@$20000) the pixel is stored.

Once the responsibility of the two highest bits is moved to the *segment mapper*, those bits don't have any significance anymore (as address bits.) As noted above, they must be replaced with bits that ensure the pixels are written within ***Bank1***. The bank itself now handles the high addressing, by holding the correct segment.

More on the Register File

In addition to the *Segment Mapper* registers, the *Register File* also includes other important registers (some beyond the scope of this article.)

One that is significant to segment mapping is the *Bank Access Control* (REG 8.) This register controls if the 'source' memory of a segment is remapped from RAM or ROM (like the original C64, both exist in the same addressing space.)

Bank Access Control register:

```
          [   00    00    00    00    ]
To set:       Bank3 Bank2 Bank1 Bank0
```

For each pair of bits:

00 = ROM , 01 = RAM
('10' & '11' are reserved and undefined.)

The default value is $55, or [01 01 01 01].

As for any *Register File* entry, use SAC or SIR to map in register #8, and then set with normal 65xx opcodes:

To set the *Access Control* to RAM, except for Bank1 (ROM):

```
SAC   #$88              ; set ACC to REG 8
LDA   #%01010001        ; set Access Control
SAC   #$00              ; restore ACC
```

Register File Mapping and Interrupts

Although mentioned in the 'advanced' *segment mapping* example previously, one point should be stressed:

Standard C64 IRQs have no concept of the *Register File*. They will use ACC, Y & X without regard for register remapping. If an interrupt is called while a register is remapped, **the results will likely be bad.**

Even the general remapping *'triad'* technique in examples 1.1 & 1.2 (a quick remap-and-restore) is vulnerable to this problem. Since *Register File* registers10-15 are considered 'dual purpose,' they will likely be restored by the interrupt as it exits. But, for example, if the *segment mapping* changes during the interrupt then any system vectors (IRQ or otherwise) are redirected to RAM and used in the IRQ, a system meltdown is likely. An unforeseen IRQ during remapping of zero page or the stack will result in the same. The IRQ won't get a chance to restore the mapping on exit. The crash will happen within the IRQ. Using different dest:src registers with the ACC is also likely to confuse the IRQ code, since fundamental ALU operations will have 'unexpected' (compared to standard 65xx) results.

In addition, even if the mapping is restored on IRQ exit, the register will jump wildly during use inside the interrupt. For some registers this is unacceptable, even if they are restored correctly.

A typical ***remapping triad***:

```
SIR   #$1D
LDX   #$04          ; Interrupts occurring between the SIR ops
SIR   #$12          ; can cause BAD things to happen.
```

A 'safe' ***remapping triad,*** bracketed with interrupt disable/enable opcodes (no longer a triad ☺):

```
SEI                 ; set irq disable
SIR   #$1D
LDX   #$04
SIR   #$12
CLI                 ; clear irq disable
```

In average use (simple *segment mapping*) the typical 'unsafe' version will succeed often enough, and the programmer can be lulled in to a false sense of security.

Appendix A
List of segments in a DTV V2/V3 (2 Meg)

Segment #			**Memory Address**	
dec	hex	binary	hex	dec
0	$00	00000000	$ 0000	0
1	$01	00000001	$ 4000	16384
2	$02	00000010	$ 8000	32768
3	$03	00000011	$ C000	49152
4	$04	00000100	$ 10000	65536
5	$05	00000101	$ 14000	81920
6	$06	00000110	$ 18000	98304
7	$07	00000111	$ 1C000	114688
8	$08	00001000	$ 20000	131072
9	$09	00001001	$ 24000	147456
10	$0A	00001010	$ 28000	163840
11	$0B	00001011	$ 2C000	180224
12	$0C	00001100	$ 30000	196608
13	$0D	00001101	$ 34000	212992
14	$0E	00001110	$ 38000	229376
15	$0F	00001111	$ 3C000	245760
16	$10	00010000	$ 40000	262144
17	$11	00010001	$ 44000	278528
18	$12	00010010	$ 48000	294912
19	$13	00010011	$ 4C000	311296
20	$14	00010100	$ 50000	327680
21	$15	00010101	$ 54000	344064
22	$16	00010110	$ 58000	360448
23	$17	00010111	$ 5C000	376832
24	$18	00011000	$ 60000	393216
25	$19	00011001	$ 64000	409600
26	$1A	00011010	$ 68000	425984
27	$1B	00011011	$ 6C000	442368
28	$1C	00011100	$ 70000	458752
29	$1D	00011101	$ 74000	475136
30	$1E	00011110	$ 78000	491520
31	$1F	00011111	$ 7C000	507904
32	$20	00100000	$ 80000	524288
33	$21	00100001	$ 84000	540672
34	$22	00100010	$ 88000	557056
35	$23	00100011	$ 8C000	573440
36	$24	00100100	$ 90000	589824
37	$25	00100101	$ 94000	606208
38	$26	00100110	$ 98000	622592
39	$27	00100111	$ 9C000	638976
40	$28	00101000	$ A0000	655360

41	$29	00101001	$ A4000	671744
42	$2A	00101010	$ A8000	688128
43	$2B	00101011	$ AC000	704512
44	$2C	00101100	$ B0000	720896
45	$2D	00101101	$ B4000	737280
46	$2E	00101110	$ B8000	753664
47	$2F	00101111	$ BC000	770048
48	$30	00110000	$ C0000	786432
49	$31	00110001	$ C4000	802816
50	$32	00110010	$ C8000	819200
51	$33	00110011	$ CC000	835584
52	$34	00110100	$ D0000	851968
53	$35	00110101	$ D4000	868352
54	$36	00110110	$ D8000	884736
55	$37	00110111	$ DC000	901120
56	$38	00111000	$ E0000	917504
57	$39	00111001	$ E4000	933888
58	$3A	00111010	$ E8000	950272
59	$3B	00111011	$ EC000	966656
60	$3C	00111100	$ F0000	983040
61	$3D	00111101	$ F4000	999424
62	$3E	00111110	$ F8000	1015808
63	$3F	00111111	$ FC000	1032192
64	$40	01000000	$100000	1048576
65	$41	01000001	$104000	1064960
66	$42	01000010	$108000	1081344
67	$43	01000011	$10C000	1097728
68	$44	01000100	$110000	1114112
69	$45	01000101	$114000	1130496
70	$46	01000110	$118000	1146880
71	$47	01000111	$11C000	1163264
72	$48	01001000	$120000	1179648
73	$49	01001001	$124000	1196032
74	$4A	01001010	$128000	1212416
75	$4B	01001011	$12C000	1228800
76	$4C	01001100	$130000	1245184
77	$4D	01001101	$134000	1261568
78	$4E	01001110	$138000	1277952
79	$4F	01001111	$13C000	1294336
80	$50	01010000	$140000	1310720
81	$51	01010001	$144000	1327104
82	$52	01010010	$148000	1343488
83	$53	01010011	$14C000	1359872
84	$54	01010100	$150000	1376256
85	$55	01010101	$154000	1392640
86	$56	01010110	$158000	1409024
87	$57	01010111	$15C000	1425408

88	$58	01011000	$160000	1441792
89	$59	01011001	$164000	1458176
90	$5A	01011010	$168000	1474560
91	$5B	01011011	$16C000	1490944
92	$5C	01011100	$170000	1507328
93	$5D	01011101	$174000	1523712
94	$5E	01011110	$178000	1540096
95	$5F	01011111	$17C000	1556480
96	$60	01100000	$180000	1572864
97	$61	01100001	$184000	1589248
98	$62	01100010	$188000	1605632
99	$63	01100011	$18C000	1622016
100	$64	01100100	$190000	1638400
101	$65	01100101	$194000	1654784
102	$66	01100110	$198000	1671168
103	$67	01100111	$19C000	1687552
104	$68	01101000	$1A0000	1703936
105	$69	01101001	$1A4000	1720320
106	$6A	01101010	$1A8000	1736704
107	$6B	01101011	$1AC000	1753088
108	$6C	01101100	$1B0000	1769472
109	$6D	01101101	$1B4000	1785856
110	$6E	01101110	$1B8000	1802240
111	$6F	01101111	$1BC000	1818624
112	$70	01110000	$1C0000	1835008
113	$71	01110001	$1C4000	1851392
114	$72	01110010	$1C8000	1867776
115	$73	01110011	$1CC000	1884160
116	$74	01110100	$1D0000	1900544
117	$75	01110101	$1D4000	1916928
118	$76	01110110	$1D8000	1933312
119	$77	01110111	$1DC000	1949696
120	$78	01111000	$1E0000	1966080
121	$79	01111001	$1E4000	1982464
122	$7A	01111010	$1E8000	1998848
123	$7B	01111011	$1EC000	2015232
124	$7C	01111100	$1F0000	2031616
125	$7D	01111101	$1F4000	2048000
126	$7E	01111110	$1F8000	2064384
127	$7F	01111111	$1FC000	2080768

(Segment numbers >= 128 would access memory in the 3-4 Meg block. That memory currently doesn’t exist. But the ability to map it does.)