THE SUPERCPU CAN FLI!

Dragon & George, a multi-color FLI picture by Andreas W. Andersen of SHAPE demo group.

Two Articles by Todd S. Elliott

editorial Currents

Welcome to the premiere issue of ***Commodore Currents***! This endeavor is intended to be a semi-regular printed publication someday. Currently, due to the high cost of printers that can print 11x17 paper, this publication will be published as a PDF file on the Internet. Naturally, the PDF file will be free to all who can download it and view/print it out. Hopefully, in the not too distant future, I can publish ***Commodore Currents*** in-house with a 11x17 printer and distribute it via a subscription method on a semi-regular basis.

Just what is ***Commodore Currents*** all about? Initially, I wanted a publication that could reprint all of my articles I've written for various **Commodore** oriented publications. I fully intend to publish this endeavor via **Wheels**, **geoPublish**, **geoWrite** and **PostPrint** II/III and use the power of **PostScript** to get my articles in print. However, I will be using a mainstream computer to do some graphics, such as the ***Commodore Currents*** logo, grabbing screenshots of actual Commodore 64/128 screens, **GhostScript** for the PDF conversion, **GhostView** for the layout proofing, and more! *This is* ***Commodore GEOS*** *printing at its finest!*

For the premiere issue here, I've selected two articles I've originally submitted to two different publications, all on the same subject, ***The SuperCPU can do Full-Screen FLI Pictures***. The first article was originally published in the **Commodore Digest**, one of the premier Commodore GEOS publishing efforts edited by **K. Dale Sidebottom** and **Rolf L. Miller** of the **UCUGA** users group. This article is intended for beginners, who are not familar with demo coding tricks used in achieving the FLI mode of the C64/128 computers. The second article was originally published in ultra-technical **C=Hacking** online magazine published by **Steve Judd**. That article is fully intended for experienced programmers knowledgeable about the FLI mode of the C64/128 computers, and SuperCPU coding experience is not a pre-requistite. I've included some screenshots of FLI pictures for some 'eye candy' throughout the two articles.

A little sidebar here-- The FLI picture mode was first used in the **milestone** demo, *Dutch Breeze*, by the **Blackmail** demo group. I give grateful acknowledgement to these pioneering coders who simply **dared** to push further the boundaries of the VIC-II chip to depths never imagined by Commodore itself.

About the future of this natty publication from the esteemed editor: ***Commodore Currents***, for the next few issues, will contain reprints from my Commodore article repository. While I am not as prolific as other notable Commodore scribes, I hope to organize my content over several issues on a semi-regular basis. Once I have a 11x17 printer, I will seriously examine publishing possibilities for a newsletter or magazine giving the usual breadth and depth of leading Commodore concepts out there. Such coverage, not necessarily limited to, would emcompass GEOS, demos, programming, with an eye towards using the machines as a main hobby instead of waxing a nostalgic look like those 'retro' magazines usually do. Of course, since this would be published in-house, there's no printer delays, etc. Still, this publication would be a hobbyist endeavor and the promise of a prompt publishing schedule is one that cannot be kept, hence I hope to undertake it on a semi-regular basis.

Without further ado, enjoy reading about how the SuperCPU can achieve a full-screen FLI picture, opening up the possibilities for further explorations of demo programming revolving around the **VIC-II** chip and the SuperCPU's DMA capabilities.

This publication, while not perfect, strives to publish material that recognizes all relevant intellectual properties as belonging to their respective authors. To the extent practical, I have secured some permissions from the author(s) to reprint the relevant works here. If you, the reader, possess an intellectual property interest in any material presented in this publication, please let me know of any objection(s) you may have.

I have included a short bibliography here to give due credit(s) to various authors and other program(s) that have contributed to the successful creation of this publication.

Wheels OS
SuperCPU
Concept+
All by **Maurice Randall** at:
www.cmdrkey.com

C64 Gallery
Maintained by **Roman Cheblec** (**CreaMD**). A must see for C64 graphics enthusiasts!
www.studiostyle.sk/dmagic/gallery/

GhostScript/GhostView
www.cs.wisc.edu/~ghost/

UCUGA/Laser Lovers
Promoted by **K. Dale Sidebottom** and **Rolf L. Miller**. If you want the finest in *Commodore GEOS printing*, check it out!
www.luckyclub.net

C=Hacking
A technical resource for hardcore CBM 8-bit programming, now edited by **Stephen L. Judd**.
www.ffd2.com/fridge/chacking/

Photos
www.royaltyfreeclipart.com

ConGo
I used this program to convert the photos into FLI pictures. It is created and maintained by **Matthias Matting**.
www.editorix.org/congo/

VICE
The ultimate C128 emulator. Nothing else comes close.
viceteam.bei.t-online.de

Star Commander
Maintained by **Joe Forster**. A cool interface for maintaining CBM/GEOS files on modern platforms.
sta.c64.org

geoZIP
Maintained by **Todd Elliott**, **Werner Weicht**, **Pasi Ojala** and **Maurice Randall**. It helps to keep all elements of the publication together outside the GEOS environment.
www.cs.tut.fi/~albert/Dev/gunzip-geos/

Introducing Full Screen FLI's for the SuperCPU
Copyright (C) 2002 By Todd S. Elliott

Ever since the VIC-II chip's (referred to as VIC in this article) introduction in January 1982 in the **UltiMax** computer marketed in Japan, inquisitive programmers have continually explored its varied registers in every way imaginable in creating truly stunning, spectacular and uniquely creative demo effects. For twenty years running, the venerable VIC has helped propel many demo programmers, graphicians (pixel artists) and demo crews to elite status within the Commodore 8-bit demo scene for their continued brilliance, cutting-edge innovation and unbridled creativity. I now introduce full-screen FLI's to this demo legend and lore.

But, what is this 'FLI' mode? This is a well-known expansion of the VIC chip and is standard demo fare. FLI is shorthand for **Flexible Line Interpretation**. Before I delve into this FLI mode, a brief explanation as to the inner nature of the VIC chip is warranted. This chip is primarily responsible for redrawing the screen 60 times a second on our North American systems. However, It has a lot of tasks to do other than just displaying stuff onscreen. For example, the VIC has to refresh the local RAM memory (referred to as RAM in this article), keeping such contents alive.

How can the VIC display stuff onscreen? It is really a microcontroller of sorts, similar to the 6510/8510 CPU (referred to CPU in this article) which is the true brains of the Commodore 64/128 system. What it means is that most of the time, the CPU will continually fetch stuff from memory locations, act on data, set conditions, stash stuff onto memory locations, etc. The VIC does similar things as it will refresh memory, get data from memory locations, act on such data, stash such stuff onto the screen.

There are 200 visible scanlines on our Commodore 64/128 screens. The computer can't read (buffer) in 64,000 different bits of information and an untold number of additional color information all at once. Now, you've got the general idea as to the sheer volume of information that the computer has to manage in just creating a single frame on the screen and the computer has to do it *60 times a second*! The computer has to somehow break this daunting task down to much smaller and manageable chunks.

This job is left to the VIC chip. The VIC chip has its own internal counters, addresses, registers, just like what the CPU has to offer for programmers. The internal workings of the VIC chip can be accessed via its registers and can be manipulated by the CPU through a user program like GEOS, for instance. The VIC chip follows an internal sequence of instructions; It will start outputting its display at scanline 0 and go all the way down to scanline 262 on NTSC screens, and only 200 of those scanlines will be visible by the Commodore user and the rest of the display is taken up by the borders.

The VIC video chip and the CPU brains are continually at odds in your Commodore 64/128. What makes these two chips coexist peacefully? One word: DMA.

Depending on how the VIC is configured via the registers that were set up by the CPU, it will look up video data from RAM that is resident in the computer. It will then calculate this video data, add in color information and output it to the screen. This is called housekeeping and the VIC chip will temporarily stop the system on every eight scanlines to do so. During this time, It has to update its internal row counters, registers, change its memory location lookups, and read in new video data from RAM. This way, the daunting task of updating the entire video screen is broken up into a very manageable chore of only eight scanlines for the VIC chip to follow and this 'eight scanline' routine is repeated forty times to fill the entire screen with a picture.

Now, if the VIC chip tries to read new graphics data from RAM, and the CPU tries to do the same, what happens? The computer would get locked up or messed up as two chips are vying for control of the databus, the critical pathways in the computer where data flows within. The CPU is always reading and writing memory locations in the computer at all times. Both the VIC and the CPU shares the databus for each and every machine cycle.

At certain times, the VIC needs more data from RAM and how can it coexist peacefully with the CPU? Enter a new concept called DMA, shorthand for **Direct Memory Access** and is what *marries* the VIC to the CPU into one happy union.

The VIC has the capability to do DMA. When the VIC chip needs to access the databus, it effectively disables the CPU chip. Once that happens, the VIC chip has now free reign to the databus, and fetches video data from RAM so that it can render such data onto the screen. When the VIC chip is done, it relinquishes control of the databus and frees up the CPU for it to resume control of the system.

This is called a **Badline** condition and the VIC will issue a badline on every eight scanlines of the video screen just so that it can update itself with new video data for rendering the next eight scanlines in its normal housekeeping duties.

A high resolution FLI logo drawn by Daniel 'DeeKay' Kottmair of Crest . The logo was originally drawn with a thick black border and it was removed for this publication. GEOS is able to use this mode with additional work.

Why is it called a 'Badline' condition? Because, the CPU is *disabled* while the VIC is busy doing its own thing; nothing else can happen within the computer. Disk accesses do not happen. Arithmetic computations do not happen. Many computer tasks do not happen.

Leave it to the demo coders to turn a bad thing into a very good thing, indeed! Badlines in the VIC are *essential* for many demo coders in achieving what seems to be impossible video tricks and color combinations. Sometime in late 1980's or early 1990's, enterprising demo programmers discovered badlines and how to use them to their advantage in creating different screen modes. The discovery was that while the VIC chip's normal behavior was to issue a badline on every eight scanlines, it could be forced to issue one on *every scanline*, all 200 of them onscreen. The programmer would insert VIC DMA retrigger code to force a badline at any scanline onscreen.

Think of this; the VIC chip is forced to do its internal housekeeping on all 200 scanlines instead of just every eight scanlines. That means video data can be fetched on every scanline. The FLI mode was born. It consists of a simple bitmap with eight video data matrixes. Each video matrix corresponds with one of the eight scanlines and are located in RAM. In FLI mode, the video matrixes are switched for each scanline, where matrix one is used for the first scanline, matrix two is used for the second scanline, etc. This pattern would repeat for every eight scanlines.

Think of it as an expansion of the standard multicolor bitmap mode, where there is only one video matrix for the entire bitmap. Now, in FLI mode, we have eight video matrixes and the resulting color possibilities makes for far more superior pictures. Hence, this mode is christened as Flexible Line Interpretation (FLI) video mode, *where maximum color flexibility is given for each scanline*.

I realize I have greatly oversimplified FLI mode and VIC chip's badlines. There is a groundbreaking study of the VIC chip as outlined in the **VIC Article** by **Christian Bauer** and it is well over 30+ pages in length and is only intended for hardcore VIC chip programmers. Interested persons should check out this website containing the HTML version of the VIC Article at::

http://www.minet.uni-jena.de/~andreasg/c64/vic_artikel/vic_article_1.htm

William Shakespeare once said, a rose by any other name is still a rose. I certainly agree and a rose is beautifully rendered here in multicolor FLI mode. The artist is unknown and probably is a graphics conversion.

The two warring factions in your Commodore 64/128 are the VIC & CPU chips, vying for databus supremacy. This kind of war leaves orphans and these lost children causes the FLI Bug!

But, the forced DMA of the VIC chip in FLI mode comes with a cost. The VIC chip has to stun the CPU and that action alone has to take up some time, where nothing virtually happens on the computer and is called a *transition period*. The internal registers of the VIC chip is still closed until control of the databus is transferred to the VIC chip. Once the CPU is fully disabled, the VIC chip now assumes control and opens its internal registers and starts reading video data from the databus. This time lag results in the *loss of video capability* for the first three columns on the screen and is called the **FLI Bug**.

This is why most FLI graphics screens always have their first three columns cut off, devoid of any video data. However, the VIC chip will display video data in the first three columns of the screen in FLI mode. But, what is displayed there? Normally, the VIC chip, on a badline condition, will get video data from the video matrix and put it on screen. But, in the first three columns, the whole system is still essentially shut down from the transition period. Yet, the VIC chip will display video data onscreen! Where is this data coming from?!?

From the databus, of course! :) Specifically, the last value still left in the databus. In the normal course of events, the CPU will fetch its next instruction from the databus and intend to execute this instruction. However, it would get stunned by the VIC and this instruction is left hanging on the databus, almost orphaned. When the VIC badline condition is over, the CPU gets the orphan instruction back and acts on it accordingly. *This orphan instruction is what creates the video data in the first three columns of a FLI screen*. This instruction is supposed to immediately follow the VIC DMA retrigger code in the FLI display loop.

Some creative democoders discovered this and managed to put in specific 6502 opcodes (instructions) in their FLI display loops after the VIC DMA retrigger code, to influence color information for the first three lines of the FLI screen. This creative approach doesn't really solve the first three column problem inherent in FLI screens. A programmer just can't pick any opcode to influence color information because they would be executed by the CPU once the badline is over. Color information comes in any combination up to 256 values and the CPU instruction opcodes only number up to 151 variations. The opcode approach to

The marriage between the VIC and the CPU chips forms the uneasy union of the Commodore 64 and 128 computers. This is a high-resolution FLI picture and is converted by GoDot.

putting color information in the first three columns of the FLI screen is very limited.

Another creative solution for the FLI Bug was to put sprites over the first three columns of the FLI picture. This worked, but required more exact timing in the FLI display loop code and took away some scanlines at the top or bottom of the graphics screen as they were used for sprite data themselves. The sprites did not offer a lot of color flexibility as they could only be switched every 21 scanlines as opposed to video matrixes which are switched at every scanline. Not a lot of demos used sprites to cover up this FLI bug with graphics data, but with some creativity, full-screen FLI displays can be done with this method.

A solution is at hand for overcoming the FLI Bug; allow me to introduce the **SuperCPU**! The SuperCPU is a separate computer and is *not affected* by the VIC's DMA (badlines). Even if the VIC chip takes control of the databus, the SuperCPU is not affected and it runs normally, doing computational tasks at 20MHz. Basically, when a Commodore 64/128 system is powered up with a SuperCPU accelerator, the SuperCPU issues *its own DMA* and stuns the CPU that is in the host Commodore computer. The CPU is effectively **tristated** (disabled) for the life of the SuperCPU currently running. The CPU never even wakes up again and remains forever at peace. Of course, once the SuperCPU is turned off or disabled, the CPU springs back to life again.

The VIC chip isn't disabled and remains active when the SuperCPU tristates the CPU. The VIC chip now has free reign of the databus for its activities and really doesn't have to worry about the CPU. But, the VIC was designed back in 1982 and

The SuperCPU upsets the delicate balance between the VIC & CPU chips' databus struggles, creating a power vacuum. The SuperCPU fills this vacuum & forcefeeds the VIC a full-screen FLI mode!

wasn't designed for a possible future accelerator upgrade in the future, which turns out to be the SuperCPU. The VIC still remains a creature of habit, popping up on the databus on every eight scanlines, doing its housekeeping, displaying video data, and stuns the CPU. The CPU is already tristated by the SuperCPU, so the VIC's attempts to disable it results in no action.

The SuperCPU, while is a computer in its own right with its own 65816 CPU unit, lacks some features. For example, it doesn't have customized chips needed to read the keyboard, to communicate with disk drives, and not the least of all, to display video data onscreen. The SuperCPU has to access the host computer's databus and it does so by DMA. When the SuperCPU tries to access the host computer's databus, it checks VIC activity; If the VIC is currently active on the databus, the SuperCPU has to wait until the badline is over, so that the SuperCPU can then access the databus. If the VIC isn't active, the SuperCPU goes ahead and accesses the databus.

Now, let's go back to the FLI video display mode of the VIC chip. Once a forced DMA (badline) occurs, the databus is seized by the VIC chip during the transition period. As explained earlier, the last instruction opcode still on the databus is passed onto video memory and displayed onscreen in the first three columns of the video screen. But, in this case, the CPU is tristated by the SuperCPU! There's *no instructions* waiting on the databus for execution by the CPU. There exists a **vacuum** on the databus when a badline condition occurs in the VIC chip and it isn't affected by this anomaly.

Now the magic happens with the SuperCPU! The SuperCPU can do its DMA and stash values on the host computer's databus, *filling this vacuum*. It is fast enough to stash a value onto the databus after the DMA retrigger code is done in the FLI display loop. In fact, it is fast enough to stash **three values** onto the host computer's databus, and the VIC chip, no wiser, fetches video data from the databus in this transition period to fill the first three columns of the FLI screen before adjusting to its normal housekeeping chores and fetching video data values from RAM.

Viola! A full-screen FLI mode is now readily achievable on the VIC chip and we have the SuperCPU to thank for allowing us to have such a capability. The SuperCPU truly expands the horizons of the VIC chip because the SuperCPU can stash values on the databus so quick that the VIC chip responds immediately. Thanks goes to **Per Olofsson** (MagerValp) for pointing me in the right direction regarding the databus.

Just what a full-screen FLI mode on the VIC can do for our SuperCPU equipped Commodore setups? For starters, having a full-screen mode opens up avenues for more color possibilities for our graphics onscreen. Let's take the standard multicolor mode. You know this screen mode as it is very common in games and drawing programs. The screen resolution is 160x200 pixels (dots) and is broken up into 1,000 blocks of 8x8 pixels. In each block, up to three colors can be independently selected.

However, one color is common in every block of the multicolor screen and is referred to as the background color. Theoretically, a program (like **GoDot**) or a graphician could optimize this background value, carefully choosing the most common color for this background. This way, designating the most common color for the entire bitmap frees up color possibilities in all blocks of the screen.

In multicolor FLI mode, the screen resolution is still the same, but is broken up to 8,000 blocks of 8x1 pixels. The same color restrictions still apply, but for only a block that is 8x1 pixels in size. So, color possibilities increase by a factor of eight as 8,000 blocks can be manipulated. However, there is still one color that is common to all 8,000 blocks on the screen and is still the background color. Graphics programs and artists still has to optimize for this mode, carefully choosing a single color that is most often used in the screen for maximum color flexibility for those 8,000 blocks of graphics data.

But, this background color can be changed on each and every scanline, all 200 of them onscreen. Why restrict your graphics creation to a single optimized value that applies to the entire screen when *you can pick up to 200 values*, and each one of those optimized values would only apply for their respective scanlines? Many demo programs and some graphics drawing programs do just that. Color flexibility is *best maximized* when the background value can be changed at every scanline of the screen, giving more color freedom for those graphical building blocks of the multicolor screen.

This feature of changing the background value for each scanline does not work too well in FLI mode, unfortunately. The background color value will *bleed* through the first three columns of the FLI Bug, appearing as colored lines, compromising the aesthetic beauty of the graphics screen. This 'bleed-thru' can be covered up with sprites, though, and some demos and graphics programs do that. But, we now have a full-screen FLI mode and the background color value will show through in the first three columns normally as it would do for the rest of the screen!

Having a full-screen FLI mode for our SuperCPU's truly enables us to have fully optimized graphical masterpieces in which the entire 160x200 screen area is fully used and full color optimization is used for each and every one of the 8,000 graphical building blocks that makes up such screens. Add interlace mode and you've got fully optimized color IFLI screens at 320x200 resolution. Imagine viewing JPEG files using **Steve Judd's JPEG viewer** for the SuperCPU, graphics which occupy the entire screen and are fully optimized for superior color fidelity. I'm not saying that Steve Judd and/or **Adrian Gonzalez** will improvise their SuperCPU JPEG viewer along those lines, but to point out truly creative and artistic possibilities that now exist for our SuperCPU's.

The graphics potential of the full-screen FLI mode for the SuperCPU is promising. Imagine browsing the Internet in color via the WAVE browser, or viewing a JPEG in full color fidelity.

A picture conversion of the talkative parrot. What secrets will he spill next of the SuperCPU? This image is in multicolor FLI mode.

Another possibility that full-screen FLI's offer for our SuperCPU's is the ability to have a high-resolution screen with better color depth than is possible for a regular high resolution bitmap. The screen resolution for a regular high resolution bitmap is 320x200 pixels and is broken up into 1,000 blocks of 8x8 pixels. In each block, up to two colors can be independently selected. This is what **Wheels**' (GEOS) 40 column screen uses for its display.

But with full-screen FLI mode, the entire 320x200 screen is now broken up into 8,000 blocks of 8x1 pixels! In each of those 8,000 blocks, up to two colors can be independently selected, multiplying color possibilities by a factor of eight. Imagine browsing the Internet using the **WAVE** in such a full-screen high-resolution FLI mode. The 80 column screen is capable of similar color freedom, but at the expense of flickering. (Think **I*Paint 128** and you'll know what I'm talking about.) **Wheels SC** could implement a special mode for the 40 column screen, so that Wheels can display in either regular high resolution mode or FLI high resolution mode in 40 columns. Again, **Maurice Randall** can choose to implement this mode or not. I'm just showing what possibilities that a full-screen high resolution FLI mode can offer for GEOS usage.

I will work on integrating this full screen FLI mode for my **dotView** program. This program is a *true switch-hitter*, running for Wheels OS, in either 64 or 128 versions and in either 40 or 80 column modes. My goals for this program is to display a single 4Bit graphics file into any screen mode that the VIC chip is capable of supporting, all from within the comforts of the Wheels GUI.

The possibilities that a full-screen FLI mode, either in high resolution or multicolor format, are numerous indeed for our SuperCPU's and its graphical future with the VIC chip has never shone so brighter.

C=Hacking Full-Screen FLI Article

The '**FLI Bug**', where the first three columns of a **FLI** screen are essentially unusable, can be squashed with the help of a **SuperCPU**. I won't go into great detail on FLI, as it has been well-documented elsewhere, but I'll begin with a short summary to get us all up to speed. I refer you to **Albert 'Pasi' Ojala's** excellent coverage of the FLI mode in **C=Hacking #4**. Pasi also proofread this article.

A Three-Minute Summary of the FLI mode

The **VIC-II** chip asserts a badline when it needs to access the databus and fetch character data or videomatrix data. It was discovered that the VIC-II chip can be manipulated by its vertical scroll register at $d011 (**SCROLY**) to induce a **badline** at any given rasterline. By having a badline at every visible rasterline, the program can manipulate $d018 (**VMCSB**) to point at the right **videomatrix** to achieve the maximum flexibility of colors given to a multi-color screen.

Unfortunately, when a program forces a badline via SCROLY, the **BA** (*Bus Available*) line in the computer goes high, and for three cycles the 6510/8510 processor has to finish its write operations or halt its read operations before the BA line is released to the VIC-II chip. The maximum number of successive write operations is three, hence the 3-cycle delay. *It is in those three cycles that the VIC-II does not fetch video matrix data to fill in the first three columns and causes the 'FLI Bug'*.

I wish to stress that in those first three cycles, when the BA line is high, the 6510/8510 processor is **still active** and can complete write operations. It isn't fully shut down. After the badline retrigger at STA SCROLY, the code following it is fetched on the databus and is ready to be executed by the 6510/8510 processor. When BA is high, the VIC-II will reference the value on the databus as videomatrix data and display it in the first three columns of the screen. The actual instructions that follow the STA SCROLY in the FLI loop *constitutes the video matrix data* for the first three columns of the screen.

Enter the SuperCPU!

Normally, a VIC-II chip access is only possible every 4 cycles. The SuperCPU can access the VIC-II chip in **1 cycle (1MHz) intervals**, making *cycle to cycle changes possible within the VIC-II chip*. More importantly, the SuperCPU **tristates** the 6510/8510 processor inside the host Commodore computer (which is a fancy way of saying that you can disconnect the processor from the system without physically removing it).

When a forced badline retrigger occurs with a STA SCROLY in a FLI loop under the SuperCPU, the BA signal inside the host Commodore computer goes high. But, the SuperCPU runs **asynchronously** and really doesn't have to pay attention to the host Commodore as it runs code after the STA SCROLY. In fact, *the SuperCPU will execute code even if the VIC-II badline is in full swing inside the host Commodore computer*.

It is in those three cycles that the VIC-II does not fetch video matrix data to fill in the first three columns and causes the 'FLI Bug'.

I knew that the instruction opcodes left on the databus after the STA SCROLY made up the video matrix data for the VIC-II chip for those first three columns of the screen. But I wondered how this was possible in a SuperCPU configuration because there would be no instruction opcodes left hanging on the databus inside the host Commodore computer. After some discussions with **Per Olofsson** ("**MagerValp**"), he suggested that *writes/reads to the i/o area will force a value to be put on the databus*.

This is where the magic begins, when the FLI loop forces the SuperCPU to write to the i/o area of the host Commodore after the forced badline retrigger at STA SCROLY. The SuperCPU will note that the BA signal is still high, so it can still access the databus and stash values there via **DMA**. This BA high signal will last for 3 cycles, enough for the SuperCPU to stash three values onto the databus.

The 6510/8510 is still tristated by the SuperCPU, and there's nothing on the databus after the forced badline retrigger at STA SCROLY. Normally, the 6510/8510 CPU shares the databus with the VIC-II for each machine cycle. With the 6510/8510 CPU out of the equation, the SuperCPU can stash a value onto this shared bus on the CPU half of this machine cycle and the VIC-II chip *will see it in its other half of the machine cycle*.

However, the databus is only eight bits wide. The VIC-II chip fetches video matrix data and color ram data **12 bits at a time**. The SuperCPU can force values onto the databus during the first three cycles after the forced badline retrigger, but on each

This multi color FLI logo was drawn by Pawel 'Valsary' Petkowicz of the Elysium demo group.

cycle the *last four bits belonging to Color RAM would not be fed to the VIC-II chip*. Only pixel values of %10 and %01 can be individually selected in multicolor FLI mode, while %11 pixel values cannot be individually set for those first three columns of the screen. The high resolution FLI mode does not suffer from this problem because it does not use color RAM for color attribute information.

Full-Screen FLI in practice

Let's get down to the nitty gritty. The **Write I/O Approach** requires three 200-byte tables, corresponding to each column. Each value on those tables correspond to each visible rasterline. For example, the first byte of each table corresponds to rasterline 50, the second byte of each table corresponds to rasterline 51, etc. The first table contains values needed for the first column of the screen, the second table contains values needed for the second column of the screen, etc.

In the FLI display loop prior to the STA SCROLY command, the current rasterline is used as an index to all three tables. The values are then fetched from the tables and inserted into the code that follows the STA SCROLY command using self-modifying code techniques. When the STA SCROLY happens, the code that immediately follows it *starts writing the values onto the databus*, all three in a row to complete the first three columns of the screen.

There is a disadvantage with this approach. It requires that three 200-byte tables be specially constructed and stored somewhere in memory that is not mirrored by the SuperCPU. A routine would have to read in a FLI graphics file, extract information from the first three columns and store it into their respective 200-byte tables.

Pasi Ojala came up with a graph depicting the SuperCPU interacting with the VIC-II in action, showing what happens after the **forced DMA retrigger** at STA SCROLY. The 'LDA #$xx' would have been modified earlier in the FLI routine (before the STA SCROLY) using self-modifying code. Here is the relevant source code which takes up 4 machine cycles inside the host Commodore computer.

```
sta  scroly   ; abcd
              ; d = write Y to SCROLY on 1MHz
              ; bus CPU half - Mach. Cycle #1
lda  #$00     ; ef
sta  $d022    ; ghij
              ; j = write 1 to $D022 on 1MHz bus
              ; CPU half - Mach. Cycle #2
lda  #$00     ; kl
sta  $d022    ; mnop
              ; p = write 2 to $D022 on 1MHz bus
              ; CPU half - Mach. Cycle #3
lda  #$00     ; qr
sta  $d022    ; stuv
              ; v = write 3 to $D022 on 1MHz bus
              ; CPU half - Mach. Cycle #4
```

There are the two shared halves consisting of a machine cycle inside the host Commodore bus, and by stashing values onto the databus, this value is *carried over* to the VIC-II half and *is read as videomatrix data* during the first three columns of the FLI screen.

Values on the databus which is carried over onto the VIC-II half of the databus:

DMA : DMA condition detected by VIC-II
col0 : colors for column 0 read, gets the
: value 1 put into the bus by SCPU
col1 : colors for column 1 read, gets the
: value 2 put into the bus by SCPU
col2 : colors for column 2 read, gets the
: value 3 put into the bus by SCPU

An alternative approach bites the dust

The SuperCPU can also fetch values onto the databus by reading from the I/O region. If a coder were so inclined to use a '**Read I/O Approach**', where is a program going to find 600 free bytes in the i/o region at $d000-$dfff? The idea is to force the SuperCPU to do a read on the databus via DMA and this can't be done with mirrored locations similar to the ones used in those VIC optimization modes. When a SuperCPU reads a value from mirrored memory, *it does so from its local RAM* and not the RAM that is inside the host Commodore computer. However, if the SuperCPU reads from the I/O block at $d000-$dfff, *it will read a value from inside the host Commodore computer using DMA*.

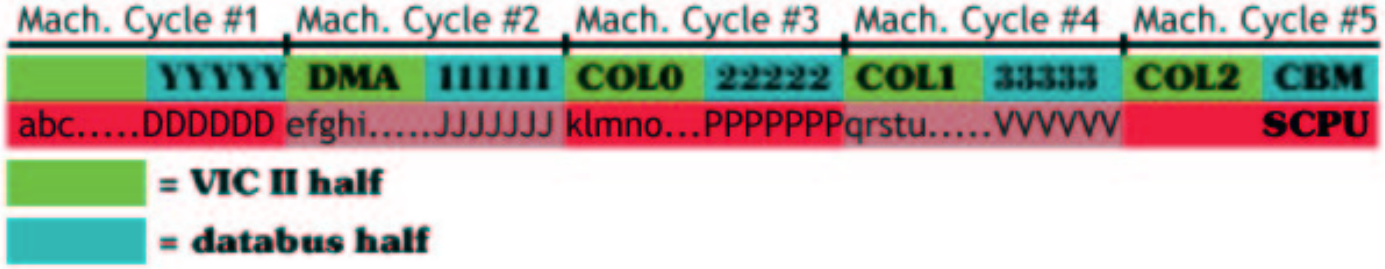
Unfortunately, this approach did **not work** when the BA line went high inside the host Commodore computer and is unworkable for a full-screen FLI mode. The *SuperCPU stops for reads if BA is high*, just like its 1MHz 6510/8510 counterpart.

Other Considerations.

There were some interesting observations while debugging the full-screen FLI routines. The full-screen FLI routines were originally inspired by **Robin Harbron's** *IRQ-based* FLI routines. Because they are driven by an IRQ, the CPU is *still available for normal computational tasks*.

When all three videomatrix values are written to after the STA SCROLY in the **line-interruptible FLI routine**, the IRQ must then exit quickly with the restoration of the registers. It's a good idea to **avoid writing** to any mirrored location or read/write to any I/O region ($D000-$DFFF), since the SuperCPU will have to **wait** for VIC to finish with the data bus.

Using a raster IRQ will naturally lead to trouble, since *cycle-exact timing is needed*, so a **CIA** timer is used. The timer may be set to synchronize a **PAL** or **NTSC** machine. Then in the FLI routine the timer



can be checked and indexed into a table of preset timing values so that the forced badline retrigger at STA SCROLY will always happen at the right time on the screen, no matter what the SuperCPU is doing when the VIC-II interrupted it with an raster IRQ. Thanks goes to **Ninja/The-Dreams** (aka **Wolfram Sang**) for tips on how to create a stable line-interruptible FLI routine using timers.

Source code

Without further ado, here is the source code. This source code was used in the **Santa Claus FLI Demo** for **Wheels OS**. This code will run in either Commodore 64 or 128 computers and in either PAL or NTSC systems. It did take a lot of tweaking at **Points #1, #2, #3, #4, & #5** as I tried to perfect the routines as closely as possible. The full source code for the Santa Claus FLI demo can be supplied via email upon request. It is in **Concept+** (**geoProgrammer**) format.

The Black Mail demo group coined the FLI mode and used it in their ground breaking demo, 'Dutch Breeze'.

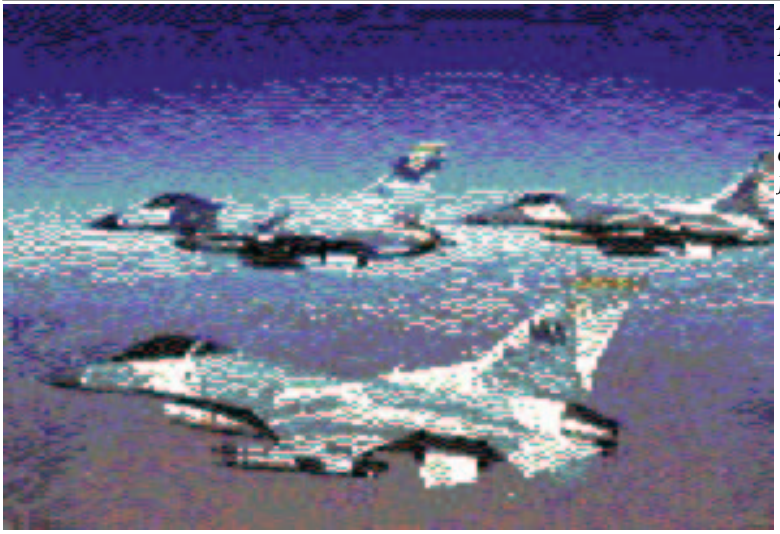

```
; Wedges the full-screen FLI interrupt
; handler in Wheels systems.
; Thanks goes to Robin Harbron for the
; idea of a line-interruptible FLI routine.
InstallFLI:
            ; Installs the FLI routine
jsr         ClearMouseMode
            ; Turn off the mouse.
sei
lda         CPU_DATA
            ; Save 6510/8510 Location #$01.
sta         r6510
lda         screenMode
            ; Check computer.
bne         2$
            ; Take branch in 128 mode.
lda         #IO_IN
.byte       $2c
2$: lda     #%00110111; for 128 mode only
sta         CPU_DATA
lda         vmcsb
            ; Save original video bank info
            ; for Wheels.
sta         oVMBmp
lda         scroly      ; save screen Y axis
sta         yaxis
MoveW       $fffe, oldVector
            ; Saves the old Wheels IRQ
            ; vector.
lda         #<fli       ; Sets up fli raster.
sta         $fffe
            ; At the IRQ interrupt vector.
lda         #>fli
sta         $ffff
```

When the STA SCROLY happens, the code ... starts writing the values onto the databus, all 3 in a row to complete the first 3 columns of the screen.

```
; Point #1
lda         #$31
            ; Trigger the IRQ request
sta         raster      ; At rasterline 49.
lda         scroly
and         #$7f
            ; Clear bit 7 of raster register.
sta         scroly
lda         #1
sta         vicirq      ; Ack raster ints.
cli
rts

RemoveFLI:
            ; Removes the FLI routine
sei
MoveW       oldVector, $fffe
            ; Restores the old Wheels IRQ
            ; vector.
lda         #$fb
            ; Trigger the IRQ request
sta         raster      ; At rasterline 251.
lda         yaxis
            ; restore screen Y axis
and         #$7f
            ; Clear bit 7 of raster register.
sta         scroly
lda         #1
sta         vicirq      ; Ack raster ints.
lda         oVMBmp      ; Restore original
sta         vmcsb       ; video bank info for
lda         r6510       ; Wheels.
sta         CPU_DATA
            ; Restores 6510 Port #$01
cli
jmp         StartMouseMode
            ; Start the mouse on.

fli:        ; The actual FLI interrupt routine
            ; lies here.
pha
.byte       $da         ; phx
.byte       $5a         ; phy
php                     ; Save CPU flags.
; Point #2
ldx         #$03
            ; #$0f for PAL SuperCPU
            ; systems.
3$:         dex
bpl         3$
lda         raster
tax
ldy         colOneClrs,x
            ; Get colors for the first three
            ; columns.
sty         mark4+1
ldy         colTwoClrs,x
sty         mark5+1
ldy         colThreeClrs,x
sty         mark6+1
inx
; Point #3
cpx         #$f9        ; Have we reached
bne         1$          ; scanline 249?
```



A bunch of F-16's fly the skies. This is a multi-color FLI picture converted from a photo.

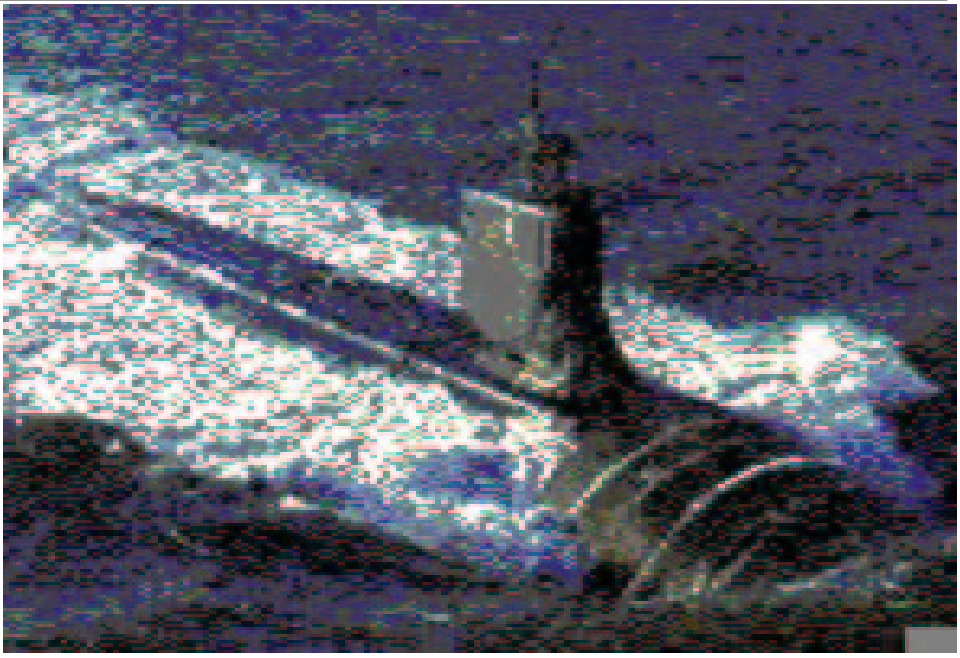
```
; Point #1
ldx     #$31
        ; Restart IRQ at rasterline 49.
1$: stx raster
; By this time, the raster interrupt register
; is incremented by one, and will re-trigger
; the same fli routine. This way, it is fully
; line-interruptible & frees up SCPU time.
ldy     #$01
sty     vicirq      ; Ack raster ints.
and     #$07
tax     ; Mask out lower three bits.
ldy     tabd018,x
        ; Use preset values for vmcsb.
lda     d011tab,x
        ; Use preset values for scroly.
sty     vmcsb       ; Select video matrix.
sta     scroly      ; Forces the badline.
mark4:  lda #$00    ; Stores a video
sta     $d022       ; matrix value onto
                    ; the first column.
mark5:  lda #$00    ; Stores a video
sta     $d022       ; matrix value onto
                    ; the second column.
mark6:  lda #$00    ; Stores a video
sta     $d022       ; matrix value onto
                    ; the third column.
plp                 ; Restore CPU Flags.
.byte   $7a         ; ply
.byte   $fa         ; plx
pla
rti
; Do NOT use any memory accesses to the
; host CBM databus in this part because it
; will be blocked by the VIC-II badline.
; Point #4
tabd018:            ; Preset video matrix
                    ; values.
.byte $78,$08,$18,$28,$38,$48,$58,$68
                    ; NTSC systems
;.byte $08,$18,$28,$38,$48,$58,$68,$78
                    ; PAL systems
d011tab:            ; Preset VIC DMA
                    ; retrigger values.
.byte $38,$39,$3a,$3b,$3c,$3d,$3e,$3f

ChkAbortKey:
                    ; Checks the RUN/STOP key
LoadB   $dc00, #$7f
                    ; checks STOP key
3$: asl $dc01
; Check for the RUN/STOP key. This
; also synchronizes the line-interruptible
; FLI routine.
bcs     3$          ; Branch if it isn't
rts                 ; pressed.

prep3Cols:
; Prepares the first three columns of the
; FLI screen. Ideally, a FLI file would be
; loaded in and this routine would then be
; called to set up the three 200-byte tables
; corresponding to each column, covering
; the first three columns of the screen.
lda     #$40
sta     mark1+2     ; Prepare the marks.
sta     mark2+2
sta     mark3+2
ldy     #$00
sty     mark1+1
iny
sty     mark2+1
iny
sty     mark3+1
php
sei
lda     screenMode
beq     1$          ; take branch in 64
lda     $ff00       ; mode.
pha                 ; save 128 config.
lda     #%01111110; select RAM at
sta     $ff00       ; $4000
1$: ldy #$00
ldx     #$07
; Use self-modifying code to create three
; 200-byte tables for each column of the FLI
; screen and each value is indexed by the
; scanline in the FLI routine.
; Point #5
mark1: lda $4000
sta     colOneClrs+49,y
mark2: lda $4001
sta     colTwoClrs+49,y
mark3: lda $4002
sta     colThreeClrs+49,y
clc                 ; use +48 for the
lda     mark1+2     ; column offset in
adc     #$04        ; PAL systems.
sta     mark1+2
sta     mark2+2
sta     mark3+2
iny
dex
bpl     mark1
sec
lda     mark1+2
sbc     #$20
sta     mark1+2
clc
lda     mark1+1
adc     #$28
sta     mark1+1
tax
inx
stx     mark2+1
inx
stx     mark3+1
lda     mark1+2
adc     #$00
sta     mark1+2
sta     mark2+2
sta     mark3+2
cpy     #200
bne     mark1-2
lda     screenMode
beq     2$          ; take branch in 64
pla                 ; mode.
sta     $ff00       ; restore 128 config.
2$: plp
rts
```

```
.ramsect $1000
; All column colors are referenced by
; scanline.
colOneClrs:
; Column one colors of the FLI screen.
.block256
colTwoClrs:
; Column two colors of the FLI screen.
.block256
colThreeClrs:
; Column three colors of the FLI screen.
.block256
```

Hopefully the full-screen FLI possibilities that the SuperCPU can now unlock will bring forth cool software for our SuperCPU's and tons of 'eye candy'.



A seawolf submarine begins its descent into the deep waters; A multi-color FLI picture converted from a photograph.

This is a multi-color FLI picture drawn by Pawel 'Valsary' Petkowicz of the Elysium demo group, saying so long to the Commodore Currents issue with a bang!