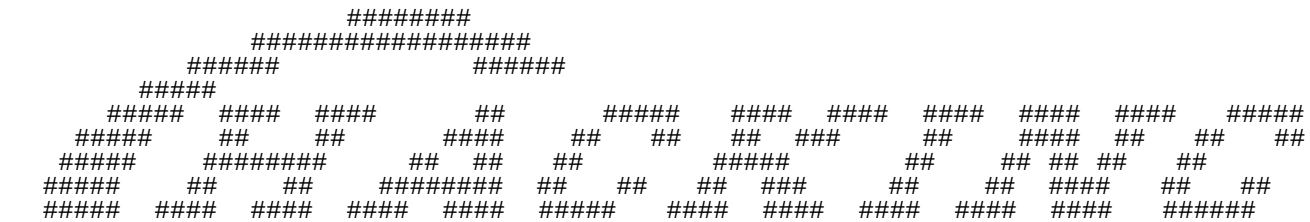


Issue #19
May 29, 2000
(Memorial Day)

.....

Seek, and ye shall find.

Ask, and it shall be given.

.....

BSOUT

C'mon, it's only, what, 9 months late?

Many of you, I am sure, have been wondering, "Is C=Hacking still alive? Has he lost interest?" The respective answers are yes, and no.

- BUT -

Although I have not lost interest in the 64, I have lost a lot of free time I once had, and I am now able to pursue a lot of other interests! So the total time allocated to the 64, and hence to C=Hacking, has decreased considerably. Work on this issue actually began last summer, around August or September. But work on jpx began about the same time, followed by work on Sirius, and I devoted my C64 time to them instead of C=Hacking. Then work intensified at work, and work began on a garage, and a plane, and... well, you get the idea. Poor issue #19 just got worked on in little dribbles every few weeks.

The main reason I share this sad tale is that, the way I see it, C=Hacking could use a little help, if it is to come out more frequently. If nobody volunteers it will still come out, but in exactly the way it does right now -- a little less frequently than it ought to. Some of the more time-consuming tasks are: finding articles, reviewing (actually refereeing) articles, and collecting the latest news and tips. Finding articles means finding people who are doing some nifty Commodore project, or talking someone into doing some nifty Commodore project. Refereeing an article means reading the article carefully, making sure everything is technically correct, making suggestions for improvement, and so on. And collecting news means being plugged into the system.

I have a few people I rely on for some of these things, but I could use more, and if you'd like to help out (especially finding new articles, or keeping up to date on the latest C64 news) please drop me an email.

With that out of the way, brother Judd would like to preach on a malaise that afflicts the C64 world and which has been getting worse: Not Finishing The Job. I just think about all the promising projects I've heard about over the last few years -- off the top of my head I remember a SCPU game, a SCPU monitor, several demos, multiple utilities, a VDC code library, several OSes... -- which were Almost Done. And where are they now? Presumably, still Almost Done. So if you have a project which is Almost Done, but has been sitting around for the last few months/years... please, please finish up that last 10% and release it.

We, the technical community, are a community. We draw strength from each other, we get ideas and motivation from each other, and we push each other to do great things. It's a big feedback loop, where activity stimulates more activity, and decreased activity begets yet less activity. I suppose C=Hacking serves as a prime example of this.

I'm not saying we're on the verge of a big programming renassaince, but I am concerned that we are drying up. Maybe if people finish up those programs lying around it will reverse the trend. (I mean, hey, doesn't this finally finished-up issue want to make you go out and do cool stuff?)

In other news, The Wave seems to be testing out wonderfully and is totally cool. In case you've been under a rock these past few months The Wave is an integrated TCP/IP suite for Wheels -- telnet, graphical web browser, PPP, the works. Lots of people have been beta-testing it for several months now and it is solid. Outstanding.

I was asked lo these many months ago to put in a plug for

http://www.6502.org

which is run by Mike Naberezny (mnaberez@nyx.net). He is looking for comments, suggestions, and maybe even contributions, so drop him a line and tell him what you think.

The ever-resourceful Pasi Ojala has several new thingies on his web site. This is probably ancient history by now but it's in my "latest news" file, sooo...

1) a voice-only copy of the Amiga Expo 1988 presentation by R.J.Mical about the early years of Amiga is available in four parts as .mp3 from http://www.cs.tut.fi/~albert/Dev/
(24kbit/s, 16kHz, mono, ~20MB total, over 100 minutes)
Includes facts and fiction and funny stories about the making of the Amiga. The files may change location in the future but you will find links to them from my page. Enjoy!

2) Some VIC20 graphics are also available at
http://www.cs.tut.fi/~albert/Dev/VicPic/
There is one picture which can be viewed with unexpanded VIC20 (with 154x/7x or 1581 drive) and others for 8k-expanded machine. Both PAL and NTSC versions are available.
There are also gif version of the pictures on the page.

Myke Carter (mykec@delphi.com) has developed a filter program that allows C=Hacking to be converted to geoWrite format. Thus, if you'd like a geoWrite version of C=Hacking, send him some email!

Finally, this is memorial day here in the States, and I'd just like to suggest folks take a little time to think about the purpose of this holiday and why we have it.

Okay then, enough with the jabber, and on to hacking excellence.

....
....
..
.

:::::::::::::::::::::::::::::::: Contents ::::::::::::::::::::::::::::::::

and Adrian Gonzalez <adrianglz@globalpc.net>. Actually it's two articles:

"Decoding JPEGs". This article covers the basics and details of JPEG encoding and decoding, with special attention to the IDCT, and some related C64 issues.

"Bringing 'true color' images to the 64". This article discusses Floyd-Steinberg dithering, and how the IFLI graphics in jpz are rendered.

..... Credits .....

Editor, The Big Kahuna, The Car'a'carn..... Stephen L. Judd
C=Hacking logo by..... Mark Lawrence

Special thanks to the folks who have helped out with reviewing and such, and to the article authors for being patient!

Legal disclaimer:
1) If you screw it up it's your own fault!
2) If you use someone's stuff without permission you're a dork!

About the authors:

Jolse Maginnis is a 20 year old programmer and web page designer, currently taking a break from CS studies. He first came into contact with the C64 at just five or six years of age, when his parents brought home their "work" computer. He started out playing games, then moved on to BASIC, and then on to ML. He always wanted to be a demo coder, and in 1994 met up with a coder at a user's group meeting, and has since worked on a variety of projects from NTSC fixing to writing demo pages and intros and even a music collection. JOS is taking up all his C64 time and he is otherwise playing/watching sports, out with his girlfriend, or at a movie or concert somewhere. He'd just like to say that "everyone MUST buy a SuperCPU, it's the way of the future" and that if he can afford one, anyone can!

Richard Cini is a 31 year old vice president of Congress Financial Corporation, and first became involved with Commodore 8-bits in 1981, when his parents bought him a VIC-20 as a birthday present. Mostly he used it for general BASIC programming, with some ML later on, for projects such as controlling the lawn sprinkler system, and for a text-to-speech synthesyzer. All his CBM stuff is packed up right now, along with his other "classic" computers, including a PDP11/34 and a KIM-1. In addition to collecting old computers Richard enjoys gardening, golf, and recently has gotten interested in robotics. As to the C= community, he feels that it is unique in being fiercely loyal without being evangelical, unlike some other communities, while being extremely creative in making the best use out of the 64.

Adrian Gonzalez is a 26 year old system/network administrator for an ISP serving Laredo, TX and Neuvo Laredo, Mexico. He and his brother convinced their parents to buy them a C64 in 1984, and whereas his brother moved on to PCs he stuck with the 64 and later bought an Amiga. He learned BASIC programming in sixth grade and wrote a few BASIC programs for the family business; since then Adrian has put several demos and utilities under his belt. In addition to fancy graphics and music, Adrian has an interest in copy protection schemes (and playing the occasional game, of course). When he's not coding, he's either playing basketball, playing piano, editing videos, or going out to movies/parties. You can visit his web page at http://starbase.globalpc.net/c64/main.html for more info.

For information on the mailing list, ftp and web sites, send some email to chacking-info@jbrain.com.

While http://www.ffd2.com/fridge/chacking is the main C=Hacking homepage, C=Hacking is available many other places including

http://www.funet.fi/pub/cbm/magazines/c=hacking/
http://metalab.unc.edu/pub/micro/commodore/magazines/c=hacking/

..... Jiffies .....

$FFC6

I actually have a little Jiffy that I 'discovered' recently. It's one of those things that is so obvious and simple that it took me several tries before I stumbled onto it. It also highlights a rather powerful feature

of the lowly C64 kernal.

Not long ago, I was asked to write a slideshow program for jpz. Ideally, a slideshow program should be a "plug-in" for the regular viewer, which can load pictures from some list in a file. But I didn't see a decent way to do this, especially for jpz which has maybe 200 bytes free total. Then the thunderclap finally occured.

Everyone has used CMD4 to redirect a file to the printer. But just as the kernal can redirect _output_ to different devices, it can redirect the _input_ to be from different devices, using CHKIN. So all the slideshow program has to do is open a list of filenames, redirect input to that file, and execute the normal jpz. jpz just uses JSR CHRIN to get data -- normally that data comes from the keyboard, but with CHKIN it comes from the file instead, akin to "a.out < input" in unix. Since jpz doesn't close the file, calling jpz repetitively will keep reading from the input file.

The result is a simple and effective slideshow program, and a trick which ought to be useful in other situations. Here is the entire slideshow code, located at $02ae to be autobooting. The main loop is seven lines long:

```
*
* Simple slideshow -- slj 4/2000
*

          org $02ae

name      txt 'ssw.files'

start
          lda #start-name
          ldx #<name
          ldy #>name
          jsr $ffbd
          lda #3
          tay
          ldx $ba
          jsr $ffba
          jsr $ffc0

          ldx #<main              ;Modify JPZ to jump to main instead
          ldy #>main              ;of exiting
          lda $10fb               ;Check if jpy or jpz is in memory
          cmp #$4c
          bne :jpy
          stx $10fc
          sty $10fd
          beq main
:jpy      stx $10ed
          sty $10ee

main
          ldx #3
          jsr $ffc6
          jsr $ffe4
          lda $90                 ;loop until EOF reached
          and #$40
          bne :done
          jmp $1000               ;call jpz
:done
          lda #3
          jsr $ffc3
          jsr $ffcc
          jmp $a474

          da start
          da start
```


Burst Fastloader for C64 by Pasi Ojala, albert@cs.tut.fi

Commodore disk drives 1570/71 and 1581 implemented a new fast serial protocol to be used with the C128 computer. This synchronous serial protocol speeds up data transfer between the computer and the drive ten-fold. The amazing thing is that this kind of serial protocol was supposed to be used in VIC-20 and the 1540 drive until it was discovered that a hardware bug in the 6522 VIA (versatile interface adapter) chip prevented the use of the chip's synchronous serial interface.

The synchronous serial port would've allowed whole bytes to be sent in both directions without processor intervention with the maximum speed of one bit per two clock cycles. Without a bug-free synchronous serial port the transfer had to be slowed down considerably so that the receiver has a chance to detect all changes in the serial bus lines. This became the dead slow software-driven Commodore serial protocol.

Syncronous Serial

The complex interface adapter (6526 CIA) chips used in Commodore 64 and later in Commodore 128 have bug-free synchronous serial interfaces: serial data and serial clock inputs/outputs. In input mode, each time a rising edge is detected in the serial clock pin (CNT), the state of the serial data (SP) is shifted into a register. When 8 bits are received the accumulated bits are moved into the serial data register and a bit is set in the interrupt status register to reflect this. If the corresponding interrupt is enabled, an interrupt is generated.

In output mode the serial clock line is controlled by Timer A. The serial clock is derived from the timer underflow pulses. When a byte is written to the serial data register, the value is clocked out through the serial data pin (SP) and the corresponding clock signal appears on the serial clock pin (CNT). After all 8 bits are sent, the serial interrupt bit is set in the interrupt status register.

Synchronous serial bus is used in C128/157x/1581 fast serial protocol. An obsolete signal in the peripheral serial bus (SRQ) was taken into service as the new fast (synchronous) serial clock line. The old serial data line doubles as slow and fast serial data line. And the old serial clock line doubles as slow serial clock line and fast serial (byte) acknowledge line.

The fast serial protocol is basically very simple. The side sending data configures its synchronous serial port into output mode, the other side uses input mode. The old peripheral serial bus clock line is controlled by the receiving side and is used as an acknowledge: when the receiver is ready for data, it toggles the state of the clock line. The actual data is transferred using the synchronous serial ports. The sender writes the data to be sent into the serial data register and waits for the transfer to complete. The receiver waits for a byte to arrive into its serial data register. The actual transfer is automatically handled by the hardware.

Both the drive and the computer must detect whether the other side can handle fast serial transfers. This is accomplished by sending a byte using the synchronous serial port while doing handshaking. The drive sends a fast serial byte when the computer sends a secondary address (SECOND, which is called by e.g. CHKOUT), the computer can in practice send the fast serial byte anytime after the drive is reset and before the drive would send fast serial bytes.

Modification to c64

To use burst fastloader with C64 we need to connect the CIA synchronous serial port to the synchronous serial lines of the Commodore peripheral serial bus. Two wires are needed: one to connect the serial bus data line to the syncronous serial port data line and one to connect the serial bus SRQ (the obsolete line for service request, now fast serial clock) to the synchronous serial port clock line. Select the right connections depending on whether you want to use CIA1 or CIA2.

	1570/1,1581		C64	
Pin1	SRQ	Fast serial bus clk	CNT1/2	User port 4/6
Pin5	DATA	Data - slow&fast bus	SP1/2	User port 5/7

Top view - old c64, CIA2

Solder the wires either to the resistor pack or directly to the user port connector, but remember to leave the outer half of the connector free so that you can still plug in your user port devices.

Then solder the other ends to the serial connector. Those left- and rightmost pins are 1 and 5, respectively, so it is fairly easy to do the soldering. You can also build a cable which connects those lines externally.

Software for C64

Of course the C64 only uses the standard slow serial routines and we need a seperate fastloader routine to take advantage of the fast serial connection we just soldered into our machine. The following load routine is located in the unused area $2a7-$2ff and in the cassette buffer $334-$3ff. Just load and run the "burster" program. It installs the loader and replaces the default load routine by our routine. The old load routine is used if

* a verify operation is requested
* a directory load operation is requested (filename starts with '$')
* the filename starts with a colon (':')

So, it is possible to use the old load routine by prepending a colon (':') to the filename. This is needed if you need to use both fast and slow serial devices at the same time. Unfortunately detecting fast-serial-capable devices is not feasible, because a lot of ROM code would have to be duplicated and then the loader would become too large. Because of this it becomes the responsibility of the user to prepend the colon (':') if a slow serial device is accessed.

A fastloader version is available for both CIA1 (asm, exe) and CIA2 (asm, exe) versions, uuencoded versions are attached to this article. Only the CIA1 version is discussed here.

```
; DASM V2.12.04 source
;
; Burst loader routine, minimal version to allow loading of programs upto 63k
; in length ($400-$ffff). Directory is loaded with the normal load routine.
;
; (c)1987-98 Pasi Ojala, Use where you want, but please give me some credit
;
; This program needs SRQ to be connected to CNT1 and DATA to SP1 (CIA1).
; Cassette drive won't work with those wires connected if the disk drive
; is turned on. (SRQ is connected to cassette read line.)
;
; SRQ = Bidirectional fast clock line for fast serial bus
; DATA= Slow/Fast serial data (software clocked in slow mode)
;
; In C128D (64-mode) you should use CIA2, because it has special hardware
; which inhibits the use of CIA1 (or so I'm told).
;
; A short description of the burst protocol and commands can be found
; from the "1581 Disk Drive User's Guide".

        processor 6502

        ORG $0801
        DC.B $b,8,$ef,0 ; '239 SYS2061'
        DC.B $9e,$32,$30,$36,$31
        DC.B 0,0,0

install:
        ; copy first block to $2a7..$2ff
        ldx #block1_end-block1-1        ; Max $58
0$      lda block1,x
        sta _block1,x
        dex
        bpl 0$
        ; copy second block to $334..$3ff
        ldx #block2_end-block2          ; Max $cc
1$      lda block2-1,x
```

```
        sta _block2-1,x
        dex
        bne 1$

        lda $0330       ; load vector
        ldx $0331
        cmp #MyLoad
        beq 3$
2$      sta OldVrfy+1   ; chain the old load vector
        stx OldVrfy+2
        lda #MyLoad
        sta $0331
3$      rts

block1
#rorg $02a7
_block1
OldLoad lda #0
OldVrfy jmp $f4a5       ; The 'normal' load.

MyLoad: ;sta $93
        cmp #0          ; Is it a prg-load-operation ?
        bne OldVrfy     ; If not, use the normal routine
        stx $ae         ; Store the load address
        sty $af
        tay             ; ldy #0
        lda ($bb),y     ; Get the first char from filename
        ldy $af
        cmp #$24        ; Do we want a directory ($) ?
        beq OldLoad     ; Use the old routine if directory
        cmp #58         ; ':'
        beq OldLoad

        ; Activate Burst, the drive then knows we can handle it
        sei             ; We are polling the serial reg. intr. bit
        ldy #1          ; Set the clock rate to the fastest possible
        sty $dc04
        dey             ; = ldy #0
        sty $dc05
        lda #$c1
        sta $dc0e       ; Start TimerA, Serial Out, TOD 50Hz
        bit $dc0d       ; Clear interrupt register
        lda #8          ; Data to be sent, and interrupt mask
        sta $dc0c       ; (actually we just wake up the other end,
0$      bit $dc0d       ;  so that it believes that we can do
                        ;  burst transfers, data can be anything)
        beq 0$          ; Then we poll the serial (data sent)
        ; Clears the interrupt status

        ; This program assumes you don't try to use it on a 1541
        ; If you try anyway, your machine will probably lock up..

        lda #$25        ; Set the normal (PAL) frequence to TimerA
        sta $dc04       ; Change if you want to preserve NTSC-rate
        lda #$40
        sta $dc05
        lda #$81
        jmp LoadFile

GetByte lda #8          ; Interrupt mask for Serial Port
0$      bit $dc0d       ; Wait for a byte
        beq 0$          ;  (Serial port int. bit changes, hopefully)
        ;ldy $dc0c      ; Get the byte from Serial Port Register
ToggleClk:
        lda $dd00       ; Toggle the old serial clock (=send Ack)
        eor #$10        ;  so that the disk  drive will start
        sta $dd00       ;  sending the next byte immediately
        ;tya            ; return the value in Accumulator, update flags
        lda $dc0c       ; Get the byte from Serial Port Register
        rts
#rend
block1_end


block2
#rorg $0334
_block2

LoadFile:
        sta $dc0e       ; Start TimerA, Serial IN, TOD 50Hz (PAL)
        ;cli
```

```
        jsr $f5af       ; searching for ..

        lda $b7         ; Preserve the filename length
        pha
        lda $b9         ; Do the same with secondary address
        sta $a5         ; We store it to cassette sync countdown..
                        ;  No cassette routines are used anyway, as
        lda #0          ;  this prg is in cassette buffer..
        sta $b7         ; No filename for command channel
        lda #15
        sta $b9         ; Secondary address 15 == command channel
        lda #239
        sta $b8         ; Logical file number (15 might be in use?)
        jsr $ffc0       ; OPEN
        sta ErrNo+1
        pla
        sta $b7         ; Restore filename length
        bcs ErrNo       ; "device not present",
                        ; "too many open files" or "file already open"
        ; Send Burst command for Fastload
        ldx #239
        jsr $ffc9       ; CHKOUT Set command channel as output
        sta ErrNo+1
        bcs NoDev       ; "device not present" or other errors

        ; Bummer, the interrupt status register bit indicating fast serial
        ; will be cleared when we get here..

        ldy #3
3$      lda BCMD-1,y    ; Burst Fastload command
        jsr $ffd2
        dey
        bne 3$
        ; ldy #0
1$      lda ($bb),y
        jsr $ffd2       ; Send the filename byte by byte
        iny
        cpy $b7         ; Length of filename
        bne 1$
        jsr $ffcc       ; Clear channels

        sei
        jsr $ee85       ; Set serial clock on == clk line low
        bit $dc0d       ; Clear intr. register
        jsr ToggleClk   ; Toggle clk

        jsr HandleStat  ; Get Initial status
        pha             ; Store the Status

        ;jsr $f5d2      ; loading/verifying
        ; (uses CHROUT, which does CLI, so we can't use it)

; We could add a check here..
; if we don't have at least two bytes, we cannot read load address..

; It seems that for files shorter than 252 bytes the 1581 does not count
; the loading address into the block size.

        jsr GetByte     ; Get the load address (low) - We assume
                        ; that every file is at least 2 bytes long
        tax
        jsr GetByte     ; Get the load address (high)
        tay             ; already in Y
        lda $a5         ; The secondary address - do we use load
                        ;  address in the file or the one given to
        bne Our         ;  us by the caller ?
        stx $ae         ; We use file's load addr. -> store it.
        sty $af
Our     ldx #252        ; We have 252 bytes left in this block
        pla             ; Restore the Status
        bne Last        ; If not OK, it has to be bytes left
Loop    jsr GetAndStore ; Get X bytes and save them
        jsr HandleStat  ; Handle status byte
        beq Loop        ; If all was OK, loop..
Last    tax             ; Otherwise it is bytes left. Do the last..
        jsr GetAndStore ; Get X number of bytes and save them
        jsr $ee85       ; Serial clock on (the normal value)
        lda #239
        jsr $ffc3       ; Close the command channel
        clc             ; carry clear -> no error indicator
```

```
        bcc End

FileNotFound:
        pla             ; Pop the return address
        pla
        jsr $ee85       ; Serial clock on (the normal value)
        lda #4          ; File not found
        sta ErrNo+1
NoDev   lda #239
        jsr $ffc3       ; Close the command channel
ErrNo   lda #5          ; Device not present
        sec             ; carry set -> error indicator
End     ldx $ae         ; Loader returns the end address,
        ldy $af         ;  so get it into regs..
        cli
        rts             ; Return from the loader

HandleStat:
        jsr GetByte     ; Get a byte (and toggle clk to start the
                        ;  transfer for next byte)
        cmp #$1f        ; EOI ?
        bne 0$
        jmp GetByte     ; Get the number of bytes to follow and RTS
0$      cmp #2          ; File Not Found ?
        bcs FileNotFound        ; file not found or read error
        ; code 0 or 1 -> OK
        ldx #254        ; So, the whole block is coming
        lda #0          ; No error -> Z set
        rts

GetAndStore:
        jsr GetByte     ; Get a byte & toggle clk
        ;sta $d020
        ldy #$34
        sty 1           ; ROMs/IO off (hopefully no NMI:s occur..)
        ldy #0
        sta ($ae),y     ; Store the byte
        ldy #$37
        sty 1           ; Restore ROMs/IO (Should preserve the
                        ;  state, but here it doesn't..)
        inc $ae         ; Increase the address
        bne 0$
        inc $af
0$      dex             ; X= number of bytes to receive
        bne GetAndStore
        rts

BCMD:   dc.b $1f, $30, $55      ; 'U0',$1F == Burst Fastload command
                                ; If $9F, Doesn't have to be a prg-file
#rend
block2_end
```

   Now that was it. Now I just hold back and wait until someone
   implements this for VIC-20's buggy 6522 chips so that I don't have
   to.. :-)

```
begin 644 burster-cia1
M`0@+".\`GC(P-C$```"B5[U"")VG`LH0]Z+'O9D(G3,#RM#WK3`#KC$#R:S0[
M!.`"\!"-J@*.JP*IK(TP`ZD"C3$#8*D`3*7TR0#0^8:NA*^HL;NDK\DD\.K)Y
M.O#F>*`!C`3QNR#2_\C$M]#V(,S_R
M>""%[BP-W"#S`B#,`T@@[`*J(.P"J*6ET`2&KH2OHOQHT`@@WP,@S`/P^*H@@
MWP,@A>ZI[R##_QB0$FAH((7NJ02-Q`.I[R##_ZD%.*:NI*]88"#L`LD?T`-,A
G[`+)`K#:HOZI`&`@[`*@-(0!H`"1KJ`WA`'FKM`"YJ_*T.A@'S!5/
`
end
size 354

begin 644 burster-cia2
M`0@+".\`GC(P-C$```"B2[U"")VG`LH0]Z+)O8T(G3,#RM#WK3`#KC$#R:S0E
M!.`"\!"-J@*.JP*IK(TP`ZD"C3$#8*D`3*7TR0#0^8:NA*^HL;NDK\DD\.K)Y
M.O#F>*`!C`3=B(P%W:G!C0[=+`W=J0B-#-TL#=WP^TPT`ZD(+`W=\/NM`-U)T
M$(T`W:T,W6"I@(T.W2"O]:6W2*6YA:6I`(6WJ0^%N:GOA;@@P/^-Q@-HA;>PZ
M:Z+O(,G_CZI!(W&`ZGO(,/_J04XIJZDKUA@(.`"R1_0`TS@`LD"L-JB_JD`H
=8"#@`J`TA`&@`)&NH#>$`>:NT`+FK\K0Z&`?,%4"Y
``
end
size 344
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

8000's USER PORT & CENTRONICS PRINTERS

by Ken Ross
petlibrary@bigfoot.com
http://members.tripod.com/~petlibrary

A recent query had me digging out an old item dealing with the user port on the CBM/PETs. The main use I've put it to in the past has been to drive a parallel printer with just the addition of a home brew cable (a Panasonic Daisy Wheel printer salvaged before bin men got it!). The user port is the edge connection tween the IEEE edge and the cassette#1. The top side is mostly diagnostic, the underside is the easy to use area. It's an I/O (Input/ Output) system that you can control with a few PEEKs and POKEs. Reading from left to right (as you look at the back of the beastie):

A _ ground
B _ input to 6522 VIA, CA1
C D E F G H J K L _ are I/O lines (8 of them) , PA0-7 [data lines]
M _ CB2 line from VIA can be I/O
N _ ground

A text file to be printed out can be read a character at a time with MID$(etc) for this PRG to deal with and quite high speeds can be reached even without having to compile it .

(This is actually a section of listing just printed out from my 8096 - hence untidy numbers)

```
3010 POKE 59459, 255:REM make PA0-7 into outputs
3020 POKE 59467,PEEK(59467) AND 277 :REM disable shift register
3022 RETURN :REM finished with this sub
     [this enables the user port for this purpose]
3023 REM this sub puts the data into output
3024 if DATA <32 then goto 3080 :REM line does biz for LF & CR
3026 if DATA =>65 and DATA<= 90 then DATA=DATA +32 : goto 3029
     [petscii lower case is chr$(65-90) but ascii uses 97-122]
3027 if DATA =>193 and DATA<= 218 then DATA=DATA -128 :goto 3029
     [petscii upper case is chr$(193-218) which has to be shifted to
      ascii 65-90]
     [ascii uses up to 127 but petscii uses up to 255 for chars]
3029 REM line below sets strobe low to inform printer new data character on
way
3031 POKE 59468, PEEK(59468) AND 31 OR 192
3035 REM below sets strobe high as data arrives
3045 POKE 59468,PEEK(59468) AND 31 OR 224
3050 POKE 59471, DATA:REM at last data is POKE'd !!!
     [the data numbers from above]
3060 POKE 59468,PEEK(59468) AND 31 OR 224 :REM strobe high still
3065 REM handshake sub
3066  POKE 59467, PEEK(59467) OR 1
3067 WAIT 59469,2
3068 K=PEEK(59457)
3069 REM end of handshake sub
     [well it works for me!!]
3070 RETURN :REM back to main area for next data
3080 REM bit for LF & CR sub & return
     [this depends on the printer and the same procedure for paper eject
      if needed]
```

The cable connections are

```
CBM     - CENTRONICS
CB2     - DATA STROBE   #1
PA0~7   - DATA1-8       #2-9
CA1     - ACKNOWLEDGE   #10 ( or BUSY #11 depending on printer ! )
GND     - grounds #14, 16, 24, 33, chassis gnd 17
```

More modern printers will also need additional commands to enable things. The commands needed for Epson printers (with the exception list of Epsons that don't use them !) are on my website at :

http://members.tripod.com/~petlibrary/printesc.htm

If any more info turns up it'll be there in time .

....
....
..
.

::

::

Sex, lies and microkernel based 65816 native OSes. - Part 1

By Jolse Maginnis

Some readers may have read my article in GO64 issue 8/1999, which was a bit of an introduction to JOS and some Operating System concepts, but it wasn't very technical, and didn't really get into the nitty gritty. Getting down and dirty with the bits and bytes is what C-Hacking is all about, so that's what this series of articles will try to do wherever possible.

I'll try to go into detail about modern OS designs, paying particular detail to what is relevant to the C64/SuperCPU and what we can do without. I'll also try and make comparisons to the kind of coding most of us are used to, e.g. just using the kernel to access hardware, or just skipping the kernel altogether. Most of the article will be in reference to the SuperCPU, specifically it's 65816 CPU, and the OS I'm making for it, called JOS. If you haven't got a SuperCPU yet, hopefully you'll want one by the end! (Remember it won't stop you running stock programs!)

OK, So what do you plan to do.. And why bother?

When I first heard about the SuperCPU, I got pretty excited. "20Mhz! That's 20 times faster! 16Mbs! That's 256 times more RAM! I can only imagine what it's capable of!", well I didn't actually say those things, but I at least thought them! At the time I had already started making an OS for the C64, and at the time I didn't know much at all about making an OS, all I knew about was multitasking, and how to do it on C64. After that day, I decided I'd wait until I managed to get myself a SuperCPU and make an OS on that, and to my surprise, at that time, there didn't seem to be anyone else developing an OS for the SuperCPU.

Only when the SCPU arrived and I had started coding for it, did I realise how powerful it was. Yeah it's 20 times faster in clock speed, but it's also a 16 bit processor, which might not seem like a great step up, but once you start coding in 16 bits, it's hard to see how you did without it!

The 65816 has some great advantages over the 6502:
It's stack pointer is not limited to 256 bytes.
The Zero Page isn't stuck in the zero page! (It's now called the Direct Page).
There are a few more ways to put values on the stack.
Long addressing allowing upto 16mb directly accessible memory.
Plenty more..

The top three things in particular, together with the 16 bit wide registers means it's very suited to programming in a high level language like C, particularly when compared to code that has to be produced for 6502. Higher level languages can actually use the real CPU stack rather than having to simulate it, as with 6502. Also by moving the Direct Page register, local variables can be accessed like zero page variables, so performance isn't hurt too much.

All this would be good even at a lower speed like 1 or 2Mhzs, but it's at 20! The SuperCPU adds some real power to your old C64, but it's all hidden away because we're running a ~20 year old "OS". It's just crying out for a new one!

The C64 has many limitations, most of which are provided by the kernel and the CBM serial bus. Here's a list of the main limits:

Single Tasking - Running two seperate programs at the same time impossible.

Some devices aren't catered for - Some devices don't have a chance at running with old programs that were designed before their time.

Old sequential filesystem - It's not designed for random access files, although random access is possible, it's just slower. All C64 programs have to written, so that files are read from the beginning to the end, which is a little bit limiting. Also it's the drives that dictate the filesystem, so we aren't just stuck with the kernel's limits, we're stuck with the drives' as well. Having several files open on many drives, while reading and writing to all of them just isn't a possibility. Why would you want to do that? If you we're multitasking several programs, that's just might be what happens!

It became pretty clear that the C64's kernel was of no use to JOS, since it had too many limitations. So everything had to be re-written from scratch, with the limits removed.

Along with re-doing the filesystem and adding multitasking, I had some other plans for JOS:

Networking - Everything is internet, internet, internet these days, and why not, the internet is great! So TCP/IP and SLIP/PPP were high on the list of TODO's.

GUI - The SuperCPU is ideal for a nice, flexible, easy to program GUI.

Console - I wanted the console to be as close as possible to one of the standard terminals (vt100,ansi etc..) thus making it easy to get by without needing a terminal emulation program.

Shared libraries & shared code, relocatable binary format - Sharing as much code as possible really saves memory and loading time. The binary format means that you don't have to worry about where in memory your program will be.

Modular and scalable - It's nice to be able to choose exactly what your OS needs, rather than getting lumped with it all. E.g. Do you really need tcp/ip loaded if your not going to use the internet? If i'm running a webserver, do I really need the console driver loaded?

Device independence - Application should not have to worry at all about what devices they are using, which means that they'll be compatible any device including new ones. This is particularly useful when it comes to disk drives and filesystems.

Porting and writing C programs - Wouldn't it be great if our C64's could take advantage of the Open Source movement that's sweeping the world, and compile some of these open source programs?

OK, so why am I bothering? At first I just wanted to see what I could do with it, but now that it's come so far, it's not only of interest to me, as it's become a very powerful OS.

Bloat: My layers theory

Unless you've been living on a remote desert island for the last 5 years, you'll know about the terrible trend in personal computing these days; buy a new PC now and in 6 months or less it's outdated. As CBM users, we successfully avoid all this. Sure, CMD have tonnes of upgrades available, but they're all "once in a lifetime" upgrades, I'm pretty sure I wont be upgrading my SuperCPU!

Have you ever thought about why PC's become outdated so quickly? It's very popular to blame Microsoft (and I will!), since they are the main proponent of bloat with their ever expanding OSes and applications, but it's just generally accepted now that it's ok to leave things unoptimized, and just add more and more "layers". I run Linux on my 486 PC, with 10mb of RAM, and it's unbelievable how much time is spent "chunking" or "thrashing", due to programs and their components taking up so much RAM. For me, it's all about layers. It's what separates C64's from the bloated world of the PC. Here's my comparisons...

CPU Type

PC - 32 bit processors
C64/SuperCPU - 8/16 bit Processor

This is quite arguable, but when most of your code doesn't deal with numbers over 32768, 32 bit's can be a bit wasteful, but of course if you need to do 32 bit arithmetic on an 8 or 16 bit processor, that too is wasteful. For me a 16 bit processor is the ideal size, particularly after doing lots of 8 bit coding.

Language used

PC - Mainly C, C++
C64/SuperCPU - Just about everything in Assembler

C can be a thin layer or a thick layer, depending on the processor. On 6502 it's quite a thick layer, which is why most things for C64 were written in ASM. On 65816, that layer isn't so thick, so it's a much more viable alternative. Although, when you write in a higher level language, you tend to forget about the actual code it produces, and don't bother optimizing it. C++ adds another layer onto C, not only because of the code it produces, but the style of program. Good object oriented programming practice adds extra bloat, because there is more emphasis on doing function calls, to do things that ordinarily are done by directly accessing the data. The real bloat of Object Orientation isn't actually the code that you write yourself, you can still write optomized code in an OO language, but the bloat is in the libraries of objects that you use when

writing your application, take a look at JAVA's huge object libraries for example.

OS type

PC   - Multitasking OS
C64/SuperCPU - Kernel, or no OS at all.

A multitasking OS adds some layers by default, since it has to switch between processes. The OS isn't just the task switcher however, it's everything that's needed to run applications, such as device drivers and shared libraries. In my opinion, absolutely none or as little as possible of the OS should be written in a high level language, since it's going to be used by every application, and you want frequently used things to be as optimized as possible. Most definitely the most useful task an OS can provide is doing all the Disk I/O. Unfortunately for us, the C64's kernel and CBM's serial bus are no where near fast enough, so coders made their own DOS routines.

User Interface

PC   - Windows, X Windows
C64   - BASIC, GEOS

Windows and X are the most popular GUI's going around. X doesn't impose any standards on applications, they are free to use whatever widget toolkits they want, and usually do! When you have a few different applications running, each with it's own GUI toolkit, you soon run out of memory, particularly if they're big bloated C++ toolkits. Windows isn't quite the same, you at least have a consistent look and feel, which also adds up to less memory wastage because most apps use the same code. GEOS is nice looking but isn't very flexible at all, but this does mean that it's a very thin layer. My hope is to achieve a balance between the two.

So why'd I bother with all that? Well I just want to hilight that JOS will be taking all those things into account, and I want to minimize the amount and size of layers being added to our beloved C64's.

```
 ------------------------------------------------
| Monolithic or Micro? How do we want our kernel? |
 ------------------------------------------------
```

There are two main styles of OSes doing the rounds at the moment, both with their own good and bad points.

Monolithic kernels

These, as the name suggests, are one large monolith of code, which usually contain driver code for all devices. You would definitely consider the C64's kernel as a monolithic kernel. Multitasking kernels sometimes allow modularization, which is basically very similar to what a microkernel does, by allowing parts of the OS to be dynamically loaded. Linux is a very popular example of this. It's a monolithic kernel which allows kernel modules to be loaded dynamically. Last time I checked Lunix Next Generation worked along these lines.

Good   - Generally a little faster than Microkernels, particularly if the time
          taken to switch processes is slow.

Bad   - Not as scalable as a Microkernel. You get everything in a big chunk,
          whether you need it or not.

How   - Generally applications need to make calls to a jump table, which
          usually will point to routines for Opening, Closing, Reading and
          Writing devices.

          e.g.
                lda #'a'
                jsr $ffd2
          Prints 'a' character to the current file/device.

Microkernel

Microkernels truly are micro in size, if they're done correctly. Rather than lump all the device driver and API code in together, Microkernels only provide very simple services for setting up processes and allowing them to communicate

with each other. All the device drivers and file-systems are then supplied by optional programs that are loaded dynamically at run time. This allows maximum scalability, as you simply don't have to load parts of the OS that you don't need. The best example other than JOS would be QNX (http://www.qnx.com), a UNIX based Microkernel OS, which is extremely scalable and very small in code size. On 6502/C64, OS/A65 is another Microkernel OS.

Microkernel OSes rely heavily on fast Inter Process Communication (IPC). Luckily this is quite easy to achieve on 65816, and is basically a matter of passing pointers between processes.

Good - Extremely scalable. Nicely split up into easy managle parts. Easier to debug. I chose a Microkernel in JOS for these reasons.

Bad - Can be slower if too much time is spent switching between processes.

How - A jump table is still used, but to actually do any I/O you need to communicate with the server process via IPC.

To do this in JOS it involves setting up a message somewhere in memory and then calling the S_send system call, to send to the server process. Usually the message will be put on the stack and then popped off when returned, much like a C function call.

e.g. to open the file "hello.txt" for reading

```
      pea O_READ         ; flags
      pea ^hellostr      ; high byte
      pea !hellostr      ; low word
      pea IO_OPEN        ; Message code
      tsc
      inc
      tax                ; Low word of Message = Stack+1
      ldy #0             ; Stack is in Bank 0
      lda #Channel       ; Channel where "hello.txt" is.
      jsr @S_send
      tsc
      clc
      adc #8
      tcs

hellostr .asc "hello.txt",0
```

note: These are 65816 instructions, so if you don't know what they do you better look them up! The '@' symbol is used to force long addressing, '^' is used for the high 8 bits of a 24bit address, and '!' is used as the bottom 16 bits. Note that pea is a 16-bit instruction, so pea ^hellostr will add an extra 00 byte.

The first 4 pea's prepare an 8 byte filesystem message, containing:
Message code for an Open: IO_OPEN
24 bit Pointer to Filename: hellostr
Open flags for reading: O_READ

This message is passed to the filesystem using one of JOS's Inter Process Communication (IPC) system calls, S_send. This call takes the 24 bit address of the message in X/Y, and the IPC channel for which to send the message to, in the A register. Every system call in JOS assumes 16 bit A/X/Y registers, as there really isn't anything to be gained by switching to 8 bits for things that only need 8 bits. Adding 8 to the stack pointer at the end "pops" the message back off the stack.

This all looks a bit complicated doesn't it? Which is where shared libraries help out. The standard C library for JOS allows you to do I/O and such without actually worrying about the system calls. Yes it is a "layer", but it's a very thin one, since the library is written in ASM.

```
      pea O_READ       ; same as the c code: open("hello.txt",O_READ);
      pea ^hellostr
      pea !hellostr
      jsr @_open
      pla
      pla
      pla
```

Much simpler right?

Compare that with the C64 kernel equivalent of:

```
      lda #namelen
```

```
                ldx #<hellostr
                ldy #>hellostr
                jsr $ffbd        ; SETNAM
                lda #1
                ldx #8
                ldy #1
                jsr $ffba        ; SETLFS
                jsr $ffc0        ; OPEN
```

Notice that the JOS version doesn't worry about device numbers or anything.. I'll get to that later...

```
 ---------------------------------------------
|       C isn't just the letter after B       |
 ---------------------------------------------
```

Before I get into juicy OS details, I should explain about C and the standard C library, as I'll be mentioning it quite a bit.

C is a very powerful language that was created by the same people who created UNIX, so the two really go hand in hand. The majority of applications written for UNIX type OSes are written in C; in fact, rather than give you executable files, they are normally distrubuted as C source code, that you have to compile yourself. Why is it used so much? Well if the only high level language you've seen is BASIC, then you'd wonder how any high level language could be used for good quality programs. C is different because it's just about as close as you can get to programming in assembly without actually doing it, particularly on newer processors. It isn't quite so pretty on 6502, but it's quite good on the 65816.

In BASIC you're used to having "built in" commands that will print to the screen, and commands for opening files and reading input, and any other I/O you can think of. But C on the other hand, has nothing "built in", it doesn't even have much of a notion of strings! Strings are just pointers to null terminated arrays of characters in C. So how do you actually get C to do anything useful? i.e. do some I/O?

This is where the C standard library comes in. This library contains functions that deal with the underlying OS, and in particular opening/closing & reading/writing files. It also has code for dealing with strings, allocating memory, reading directories and various other useful functions. The standard library also contains more UNIX orientated functions, for dealing with OS features such as IPC and process control (more on processes later).

JOS implements a large section of the standard C library, in particular the section that most command line applications will use. It does implement some of the UNIX specific functions, but not in a compatible way, and programs that use these functions are likely to be system applications that aren't useful for any other system anyway.

Although it's called the standard 'C' library, that doesn't mean it can't be used in assembly language, in fact it's quite a bit easier to call the C functions than to deal directly with the OS, and there is no speed penalty in using the C library because it's been hand coded in assembly language anyway.

Would you like to see what it's like to code using the standard C library? I've been talking about functions, and if you're familiar with C64 BASIC's functions, it's quite similar to that, except that you can pass more than one value to the function. It's basically the same as writing subroutines in assembly, where we usually pass values using the A,X & Y registers or a ZP value etc.. The only difference is that ALL values are passed using the CPU stack, which is easily accesible with the 65816. Ok let's take a look at the previous open file example:

C code:

```
file = open("hello.txt",O_READ);
```

65816 assembly (16 bit regs):

```
                pea O_READ
                pea ^hellostr
                pea !hellostr
                jsr @_open       ; C functions get "_" prepended to their names
                pla              ; so you don't get them mixed with assembly ones
                pla
                pla
                stx file         ; store the result in file
                sty file+2
```

Notice that the values are placed onto the stack in reverse order, so they come out in the correct order when the function accessing them. They are also long jsr's because they aren't likely to be in the same bank as the calling program.

You might think that having to pop the values back off the stack is cumbersome, and you're right. Why can't _open pop them off? Well it could, it'd need to do some messing around with the stack at the end but it'd make things look nicer. The reason it can't is because C functions don't always know how much data will be on the stack, so they might pop the wrong amount off. It may look ugly, but you get used to it.

Now I'll give you a bigger example of what C code looks like after it's been compiled to prove that the 65816 is capable of producing half decent code. This will probably only make sense if you've done C programming before, so if you're not interested in this kind of thing skip this section..

Here's a minimal version of the standard unix util 'cat', which concatenates files together and sends them to the screen or whatever the stdout file is, as it can be redirected in UNIX.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        FILE *fp;
        int ch=0;
        int upto=1;

        if (argc<2) {
                fprintf(stderr,"Usage: cat FILE ...\n");
                exit(1);
        }
        argc--;
        while(argc--) {
                fp = fopen(argv[upto++],"r");
                if (!fp) {
                        perror("cat");
                        exit(1);
                }
                while((ch = fgetc(fp)) != EOF)
                        if (putchar(ch) == EOF) {
                                perror("cat");
                                exit(1);
                        }
                fclose(fp);
        }
}
```

and here's the (unoptomized) compiled version:

```
#define _AS sep #$20:.as
#define _AL rep #$20:.al
#define _XS sep #$10:.xs
#define _XL rep #$10:.xl
#define _AXL rep #$30:.al:.xl
#define _AXS sep #$30:.as:.xs

        .xl             ; make sure it's 16 bit code
        .al

        .(

mreg    = 1
mreg2   = 5

        .text

+_main
-_main:

        .(

RZ      = 8             ; RZ = register size: Two psuedo 32 bit registers
LZ      = 26            ; LZ = Local size: size of the local variables for this
                        ; function

        phd
        tsc             /* make space for local variables */
        sec
        sbc #LZ
        tcs
        tcd             /* set up the DP register as the frame pointer */

        stz RZ+1        /* ch = 0; */
```

```
        lda #1          /* upto = 1; */
        sta RZ+7

        lda LZ+6        /* if (argc < 2)  NOTE: could be just      */
        .(              /* cmp #2 : bpl L2                         */
        cmp #2          /* but the compiler doesn't know how far   */
        bmi skip        /* away L2 is.                             */
        brl L2
skip    .)

        pea ^L4         /* fprintf(stderr,"Usage: cat FILE ...\n"); */
        pea !L4
        pea ^___stderr
        pea !___stderr
        jsr @_fprintf
        tsc
        clc
        adc #8
        tcs

        pea 1           /* exit(1) */
        jsr @_exit
        pla
L2:
        lda LZ+6        /* argc-- NOTE: dec LZ+6 would be better! */
        dec
        sta LZ+6
        brl L6
L5:
        pea ^L8         /* This rather large bit of code is all for */
        pea !L8         /* fopen(argv[upto++],"r");                 */

        lda RZ+7        /* arrays don't translate so well! */
        sta RZ+9
        lda RZ+9
        inc
        sta RZ+7
        ldx RZ+9
        lda #0
        .(
        stx mreg2
        ldy #2
        beq skip
blah    asl mreg2
        rol
        dey
        bne blah
skip    ldx mreg2
        .)
        clc
        tay
        txa
        adc LZ+8
        tax
        tya
        adc LZ+8+2
        sta mreg2+2
        stx mreg2
        lda [mreg2]
        tax
        ldy #2
        lda [mreg2],y
        pha
        phx
        jsr @_fopen
        tsc
        clc
        adc #8
        tcs
        stx RZ+11
        sty RZ+11+2

        ldx RZ+11       /* assign it to fp */
        lda RZ+11+2
        sta RZ+3+2
        stx RZ+3

        .(              /* if (!fp)
        lda RZ+3
        cmp #!0
```

```
        bne made
        lda RZ+3+2
        cmp #^0
        beq skip
made    brl L13
skip    .)

        pea ^L11        /* perror("cat"); */
        pea !L11
        jsr @_perror
        pla
        pla

        pea 1           /* exit(1) */
        jsr @_exit
        pla

        brl L13
L12:
        pei (RZ+1)      /* putchar(ch); */
        jsr @_putchar
        pla
        stx RZ+15

        lda RZ+15       /* if (putchar(ch) == EOF)
        .(
        cmp #-1
        beq skip
        brl L15
skip    .)

        pea ^L11        /* perror("cat"); */
        pea !L11
        jsr @_perror
        pla
        pla

        pea 1           /* exit(1)
        jsr @_exit
        pla
L15:
L13:
        pei (RZ+3+2)    /* fgetc(fp); */
        pei (RZ+3)
        jsr @_fgetc
        pla
        pla
        stx RZ+17       /* ch = fgetc(fp); */
        lda RZ+17
        sta RZ+1

        lda RZ+17       /* while ((ch = fgetc(fp)) != EOF) */
        .(
        cmp #-1
        beq skip
        brl L12
skip    .)

        pei (RZ+3+2)    /* fclose(fp); */
        pei (RZ+3)
        jsr @_fclose
        pla
        pla
L6:
        lda LZ+6        /* while(argc--) */
        sta RZ+9
        lda RZ+9
        dec
        sta LZ+6
        lda RZ+9
        .(
        cmp #0
        beq skip
        brl L5
skip    .)

        ldx #0          /* return from main() */
L1:
        tsc
        clc
        adc #LZ
```

```
        tcs
        pld
        rtl
        .)

        .text

-L11    .asc "cat",0
-L8     .asc "r",0
-L4     .asc "Usage: cat FILE ...",10,0
        .)
```

As you can see, there's still quite a bit to be optomized as far as the compiler is concerned, but the code is still quite good.

Having a C compiler and a standard C library that contains the most used standard functions, is going a long way towards being able to port UNIX's and other similar environments' applications. So what i've done is create a 65816 backend for a free ANSI C compiler called LCC.

I'm no longer talking theory here either, since a little while ago I decided to give my standard C library and the compiler a test on portability, with some great results. I've managed to do extremely simple porting jobs on: Pasi's C versions of his gunzip and puzip, Andre Fachat's XA 6502/65816 cross compiler, Marco Baye's ACME cross assembler. All of which, besides ACME, so far seem to be working exactly how they should. There'd be thousands of open source programs that could easily be ported to JOS, many of which wouldn't be of much use to anyone, but still!

```
 ---------------------------------------------
| Multitasking - Seeming to do it all at once.|
 ---------------------------------------------
```

We've all had experience with multitasking so I won't bore you too much. For our purposes, it means being able to do several things at once.

But what actually is a "thing"? They're usually called "processes" or "tasks". I usually call them processes, so that's what I'll refer to them as.

There are two main types of multitasking, pre-emptive and co-operative. The latter is as you would expect, processes need to co-operate together in order to work, processes can't "do their own thing". Pre-emptive multitasking is the more flexible approach, because processes don't need to explicitly hand over the processor to another process, they just have it taken away from them if they use it for too long. So it was a pretty easy choice for which kind of multitasking JOS would have, pre-emptive of course!

You might think that the C64 already does multitasking because programs normally set up interrupt routines to go off during the processing of the program, so it can do more than one thing, but that's a very special case of what I'm referring to here. I'm referring to the ability to run seperate unrelated programs at the same time, like reading your email, and typing in a text editor. We'd all like to be able to do that wouldn't we? Particularly if we've got the processing power and RAM to do it, and the SuperCPU certainly does.

Each process "owns" resources. The resources I'm talking about are simply parts of the computer and OS like RAM, interrupts, kernel IPC objects, and some other things.

Along with the resources it owns, each process has a number of attributes. First of all it needs a unique identifier, so anything that wants to talk to it knows how to address it. In Unix-like systems, this is called a Process IDentification (PID). In JOS a PID is just a positive integer, simple.

Along with other processes being able to address it, the PID is used so that the OS can keep track of which resources the process actually owns, and when it exits (or is explicity killed) the OS can free up those things and let other processes use them.

Processes can start other processes, so everything except the first process keeps track of who its parent was in its Parent PID (PPID). You may wonder what use it is to keep track of the parent? It's always been used in UNIX to set up IPC, but it really isn't needed in JOS, apart from cosmetic purposes, since JOS has better IPC mechanisms. That's the first example of "Just because it's in UNIX doesn't mean it's needed", and there are plenty of others.

In JOS, a process can own multiple "threads" of execution. Threads are what most people's idea of what a process is: some code running.

Consider starting a C64 game, which has several different interrupt routines running concurrently. We certainly wouldn't consider each interrupt routine to

be a seperate program, and that's generally the idea behind threads, except threads are at the mercy of the pre-emptive scheduler. Almost the same result can be achieved by creating multiple processes, but why go to the hassle of loading and executing two tightly related processes with 1 thread each, when you can do the same thing with 1 process that has 2 threads? A good example of this is JOS's very own web server, which creates new threads whenever a new connection has been established by a client.

Some new technologies are particularly keen on the use of threads, namely JAVA and the BeOS. A good example of using multiple threads is given by BeOS, which starts a seperate thread for every window displayed on the screen, so it can update its on-screen appearance and remain responsive to the user, while also doing other processing.

Unix programs have generally just started other processes if they wanted to do two of their own things at once. Threads are much cleaner and nicer. Threads themselves have their own attributes, such as priority (the higher the priority the more processor time it's likely to get), state (whether they are running or waiting for something), stack and zero page space, and some other things.

I know i've mentioned that JOS uses pre-emptive multitasking, but that doesn't mean that doing:

```
                jmp *
```

is a good idea! Programs should still try and co-operate.

A typical menu program on C64 using the kernel has a structure something like this:

1. Setup variables and interrupts
2. Set up menu
3. Check for input
4. If no input go back to 3
5. Process input

If you were to run this program on a multitasking system, it would chew up a lot of processing time and slow everything else down. Polling for input on a multitasking system is generally a bad thing, but blocking and waiting for input is a good thing. So instead it would be best to do:

1. Setup variables and interrupts
2. Set up menu
3. Wait for input
4. Process input

Now this is the correct way to do it, as it only uses up cpu time when it's actually received some input. But what happens if every process is waiting? What runs then? Well there is a special process that runs when no other processes are, it's called the Idle process, and does what it's name suggests, just sits there and idles. Here is the thread code that runs in my idle process:

```
nully           jmp nully
```

For some reason I started calling it the Null process, and it's called that all throughout JOS...

I have introduced you to a couple of the main ideas behind multitasking, but wouldn't you like to know how it's done? Well here's how JOS does it..

For starters, since it's pre-emptive multitasking, JOS needs some way of interrupting the currently running process after it's consumed its alloted time. The C64 has 4 CIA timers capable of producing IRQ's and NMI's, and in JOS's case i've decided to let it use CIA 1 Timer A, which produces an IRQ. This of course means that a process could stop itself from being interrupted by doing an SEI, but if they behave well that won't happen!

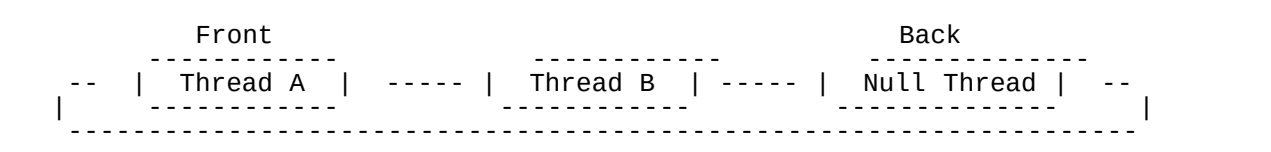
Rather than set this timer to the amount of time before a process should be pre-empted (called a "timeslice"), I double up the use of TIMER A as the system counter, which is used for timing another kind of process resource: timers. Timers can either count upwards, or downwards and give off an alarm. They really need a higher precision than a timeslice, so they set the timer to 20 milliseconds (about 1 PAL screen). The timeslice is then calcualted as 3 counts of this timer i.e. 60 milliseconds. Why don't I use TIMER B for the system timer? Well, because I want to leave as many resources open for application and device driver processes.

I mentioned that processes and threads each have their own attributes, these attributes are stored in Process Control Blocks (PCB's) and Thread Control Blocks (TCB's).

Every process has a PCB, and every process has at least one thread, which has it's own TCB. There is one process which is always loaded, and that's the Null

process. Each process's PCB and TCB's are contained in everyone's favourite data structure, the circular (or double) linked lists. The Null PCB is always at the head of the PCB list, and PCB's will only ever be on this one list, since they are either alive (in the list), or dead (no PCB exists!).

Threads on the other hand can be in various states, but in particular they can be ready for the CPU, or waiting for something (blocked). When a thread is ready, it's just waiting for its turn at the CPU, and it goes on the Ready list, which is a queue. The Null thread is ALWAYS at the back of this queue, so it only gets to run if nothing else can. The ordering of this queue is up to a part of the kernel called the Scheduler.



Some OSes have complex schedulers which take in many parameters, like priority and various CPU time measurements. On multi-user OSes like UNIX, this is important because it wants to be "fair" to all processes. But for our purposes and many other OSes, it's usually a whole lot simpler than that, it's just a simple matter of which ever process/thread has the highest priority can run. If two threads have the same priority, it normally comes down to "round robin" scheduling, where they just take it turns. JOS doesn't even implement priorities properly yet, because they actually don't make much difference to the normal processing, at the moment it's just a simple round robin scheduler that doesn't care about priorities.

What if a thread is blocked? It'll go onto a wait queue, and will return to the ready queue only when it's ready to run. At this stage of JOS, the only thing a thread will need to block for is IPC.

You may be wondering about the issue of relocatable code, as we all know the 6502 nor the 65816 is designed for running relocatable code. Sure, branches are relative to the PC, but nothing else is. So everything needs to be physically relocated before executing, and to do this properly without needing to code in a specific way, a relocatable binary format is needed. Fortunately for me, Andre Fachat had already designed such a format for OS/A65, and it fits JOS nicely because it includes 65816 extensions. Of course you need a special assembler to output this file format, which is where XA comes in. XA now even compiles for JOS, so self hosted development is now possible. The binary format will be talked about in greater detail in a future article.

Well, it's all very fine having a bunch of processes running, but that's no operating system.. Who's looking after the devices? Who's managing the memory? And how do we ask the drivers to do something for us? It's all IPC...

Inter Process Communication - Let's get talking!

Before I get into the specifics of IPC, I should give an idea of what typically happens when JOS boots. Because JOS has a very scalable microkernel design, it can load as many different device drivers and applications at boot time as it wants and infact they can loaded and removed anytime at all. So there is no one bootup procedure in JOS. There are certain things that happen every time, however.

For starters, JOS has 2 system processes, which are always started at bootup. They aren't actually loaded off disk because they are part of the microkernel code. One is the memory manager and the other is the process manager.

The memory manager as you would expect manages all the memory, but it doesn't manage the Process space memory (Bank 0), that's the job of the Microkernel. Process space memory (or kernel memory) is where all the PCB's, TCB's, Stack space and Direct Page space is located. The Memory manager, manages all the other RAM, e.g. Ram in Bank 1 and above, although, if there is no SuperRAM, it allocates 00e000-010000 as system Ram instead of using it as kernel space RAM, since it's more likely that you will run out of System Ram rather than kernel RAM.

I won't go into the specifics of the Memory Manager just yet, I'll just tell you that it performs the following requests:

Allocate any size block of RAM.
Free RAM.
Allocate any size block of Bank Aligned RAM. (Needed in some cases).
Reallocate RAM.
See how much RAM is left.

See what the largest block is.

All these things are requested via IPC, but there are Shared library routines (such as malloc, free, realloc etc) for preparing the right IPC messages to send.

The process manager's main functions are loading new processes + shared libraries, and looking up device drivers & file-systems. Whenever you open a file, you first must send a message to the process manager asking it where to send the open message.

The very first process to start however is called the "init" process (it's actually built into the microkernel, "init" isn't a filename), which starts the 2 system processes, then it starts a simple Ramdisk process and loads another process from the ramdisk called "initp".

The "initp" process should then load a proper filesystem and disk device driver also from the ramdisk, and "mount" this filesystem and executes another file this time called "init".

Note that "mounting" is preparing a filesystem for use, and all filesystems should actually be "unmounted" before switching off, because all changes may not be actually written to disk yet, even though the applications think they are. I'm guessing this is why Macintoshes refuse to let you take a disk out without the OSes permission!

The "init" file will usually be a shell script, and is responsible for starting up most of the drivers. A shell script, if you've never heard of it, is a file that has lists of commands to be run by the system, or more specifically the shell program. If you've ever seen MS-DOS .bat files, you'll know what I mean.

A typical init script has to load a user interface, unless of course you're using your machine as some kind of server, in which case you wouldn't need one and could save yourself a bit of memory!

The text based interface would require the console driver (con.drv), and the shell (sh). The console driver is capable of 4 virtual consoles, which you can switch between by pressing CBM and 1-4. This lets you exploit multitasking, as you could be running 4 different text apps on each of the screens. The shell is a pretty basic shell at the moment (like DOS's command.com), but it's enough to let you load and run any program. It also has support for pipes, but now I'm off topic..

The init script could instead load the GUI, which I'm sure most people would prefer to a text based interface!

The script also should load other drivers like: tcp/ip, ppp, digi sound driver, other filesystems, modem drivers etc... Everything is of course optional, which is where Microkernels really excell over their monolithic counterparts.

Well that's what happens at boot time, but how do the drivers and the applications communicate? I've been mentioning "messages", and that's all that JOS's IPC is: message passing. Message passing is a fast and effective way to do IPC, and for a microkernel this is essential. I chose message passing because it's the most flexible method, and you can actually implement other types of IPC by using message passing.

You can think of message passing as an extended subroutine call, but rather than being a call to a subroutine, it's a call to another process. A process, or in particular a thread, can "send" a message to another thread, the other thread "receieves" it, and then after it has processed it, "replies" to it.

You can't just send a message and expect it to be receieved straight away, the receiver has to be ready to receive it, which may not be straight away. If the receiver isn't ready, the thread that sent the message will block and wait until it's ready. Once the receiver has received it, it processes the message, and will issue a reply, which then unblocks the sender, which can then continue processing. This type of message passing is called "synchronous" message passing, as it requires synchronization between the two threads. It may help to think of "sending" as doing a JSR, "receiving" as the Program Counter being transferred to the routine, and "replying" as executing an RTS. It's a little more complicated than that, but essentially that's what it's like.

There is a great description of this kind of IPC at http://www.qnx.com/ in their technical section, with diagrams and all -- highly recommended!

Normally, OSes have to copy messages between processes, because each process gets its own address space, and can't view the memory of other processes, but as we know, the 65816 doesn't have an MMU so all memory is shared, which means that messages don't need to be copied, which gives it a significant speed increase over message passing in OSes with MMU's. Of course it does mean that

processes can accidently screw up another process's memory, but who cares! :)

All messages in JOS is directed at Channels. Channels are a resource that allow threads to receive message from other threads. Generally device drivers register a channel and use it to receive requests from applications. Channels are referred to by number, the only channels that have fixed numbers are the memory manager (0) and the process manager (1). All other channels are looked up by sending a message to the process manager's channel, e.g. Channel 1.

What exactly is a message? All the JOS system calls for IPC just deal with 24 bit pointers to messages, and the actual message data itself can be anything! However the first byte of the message should be the message code, and always is in JOS system messages. You could of course make your own protocol for your own IPC, but it's probably not a good idea.

Each different kind of driver has its own set of message codes..

```
#define PROCMSG $80
#define MEMMSG  $40

#define MMSG_Alloc      0+MEMMSG
#define MMSG_AllocBA    1+MEMMSG
#define MMSG_Free       2+MEMMSG
#define MMSG_Left       3+MEMMSG
#define MMSG_Large      4+MEMMSG
#define MMSG_LeftK      5+MEMMSG
#define MMSG_LargeK     6+MEMMSG
#define MMSG_KillMem    7+MEMMSG
#define MMSG_Realloc    8+MEMMSG

#define PMSG_Spawn      PROCMSG+0
#define PMSG_AddName    PROCMSG+1
#define PMSG_ParseFind  PROCMSG+2
#define PMSG_FindName   PROCMSG+3
#define PMSG_QueryName  PROCMSG+4
#define PMSG_Alarm      PROCMSG+5
#define PMSG_KillChan   PROCMSG+6
#define PMSG_WaitPID    PROCMSG+7
```

Those are the messages defined for the Process manager and Memory manager. Each message code defines its own structure, for example the MMSG_Alloc message has the structure:

```
        .word MMSG_Alloc
        .word !Size
        .byte ^Size,0
```

The message codes $e0-$ff are left for processes that want their threads to communicate with each other.

Anything that wants to receive messages needs to have some code like this:

```
                jsr @S_makeChan         ; make a channel System call
                sta Chan                ; save it

loop            lda Chan                ;
                jsr @S_recv             ; receieve a message from channel
                stx MsgP                ; Save X/Y in MsgP
                sty MsgP+2              ; MsgP is a zero page variable
                sta RcvID               ; Save RcvID - for replying
                lda [MsgP]
                and #$ff                ; 8 bit message code
                cmp #MSGCODE            ; check which type
                beq processMes          ; and process it
                cmp #MSGCODE2
                beq processMes2
                ...
                ldx #-1                 ; replying with $ffff in X and Y
                txy                     ; means "message not understood"
                lda RcvID
                jsr @S_reply            ; reply and loop back for more messages
                bra loop
```

All device drivers have a message loop like that. Which forces them to be modular, and thus easier to code.

Ok now let's see what sending a message would look like:

```
                lda #PROC_CHAN
                ldx #!Message
```

```
                ldy #^Message
                jsr @S_sendChan         ; Send the message
                ...

Message         .word PMSG_WaitPID,2    ; Wait for PID 2 to finish.
```

*note: it's generally a good idea to put messages on the stack, rather than use global variables, since using the stack is thread safe. No other thread will accidentally wipe over the message because they each have their own stack.

Just about everything that you consider an OS to be is done in JOS via IPC. This includes file operations, such as opening and closing, reading and writing files. How does the filesystem driver know which file you want to access after you've opened it? It could include a connection number in the IO_READ and IO_WRITE messages (you guessed it, the message codes for reading and writing!). That's a little cumbersome, though. There is a better solution: connections.

What is a connection? It's a kernel object which keeps a track of the destination channel of the messages directed at it. It also has an ID associated with it, so server processes can tell which file, for example, it refers to. Each process has a so called "file descriptor list" associated with it. People who know much about UNIX programming will know about this. In JOS, this table is really just a connection table. This table is just an array of connection numbers, which the process can access. Each element in the array can point to any connection number, which means that two file descriptors can actually point to the same file, and in the case of the first three it usually does. The first three are STDIN, STDOUT & STDERR, and they usually point to the screen, but not always!

An example File Descriptor list: (0 = no connection)

```
   0      1      2     3     4     5     6     7     8     9 .... 32
 ------------------------------------------------------------------
|  1  |   1   |  1  |  2  |  3  |  0  |  0  |  0  |  0  |  0  .... |
 ------------------------------------------------------------------
```

E.G.
Connection 1 is connected to the /dev/con/1 device (the screen). Thus STDIN, STDOUT and STDERR all point to this.
Connection 2 is connected to a file "/blah.txt" which is on the 1541 filesystem.
Connection 3 is a tcpip connection to altavista.com.

Connections are global objects, and whenever a process is loaded, it inheritis its file descriptor table from the parent, which is how it receives its STDIN, STDOUT and STDERR. File descriptors can also be explicitly redirected to other connections, or just not inherited at all. This is how JOS performs shell redirection.

I've discussed JOS's synchronous message passing, but what happens if you don't want to block and wait for a reply? You might just want to notify a server that an event has occurred, and don't need to know if it received it, nor what it thinks about it.

In this case you can send a pulse. A pulse is a tiny message (just 4 bytes), which doesn't require a reply. Probably the best property of pulses are that they can be sent during an interrupt. A good example of doing this is the console driver, which implements virtual consoles. The console driver starts an interrupt routine which scans the keyboard and checks for CBM key plus 1-4 and then sends a pulse message to its channel telling it to switch consoles.

By now you might be thinking "Microkernels must be real slow with all that process switching", but the switching code is pretty fast, particularly at 20mhz. There isn't as much switching as you would expect either, considering that IO_READ and IO_WRITE messages deal with buffers as large as 64k, so it's not as if ever single character requires a switch.

```
            ------------------------------------------------
           |     Device Independence - Everything's a file!   |
            ------------------------------------------------
```

One of the major things that people who are learning UNIX have to learn, is that practically everything is a file. Devices such as the keyboard and screen (the console) are accessed using a file. Why you may ask! Well there isn't one compelling reason, but it just makes it handy if you can access the console as a file, especially for debugging. Take for example, the ability to redirect screen output to files, a program doesn't have to be explicity designed for doing that if everything is a file, including the console, it's just a simple matter of changing the output file.

Not only are devices files, but filesystems can be "mounted" on any directory, which gets rid of the need for devices numbers. Navigating through different

filesystems is just a simple matter of changing directories. It also means that applications don't concern themselves with what the actual filesystem and device is, just that it's there. So applications will work with any devices that have drivers.

Ok so now you know some of the reasons behind the "everythings a file", so how is it done in JOS? I mentioned that the process manager is in charge of "looking up" channels, but how does it perform this lookup?

The process manager contains a table with entries for file-systems, devices and special processes. File-systems are names that end in a '/', device files usually start with '/dev/' and special processes start with '*'. So the table may look something like this:

```
Name            Channel         Unit
/               2               1               ; file-system mounted at /
/usr/           2               2               ; file-system mounted at /usr/
*digi           3               0               ; digi driver
*tcpip          4               0               ; tcp/ip
/net/           4               1               ; tcp connections
*cbmfsys        5               0               ; the cbm file-system
*packet         6               0               ; the packet driver (ppp/slip)
/dev/null       1               0               ; the process manager handles
                                                ; this
```

The name and channel fields are self explainitory but the Unit field allows a channel to determine which of its names was used.

Whenever the process manager receives a request to look something up, depending on what type of request it is (special process requests don't), it will prepend the processes Current Working Directory to the filename (unless the name starts with a '/'), and then parse the name for '.' and '..' directories, which alter the string.

So for example you ask for the file "./hello/./../afile.txt" and your CWD was "/usr/files/" it would be parsed as:

"/usr/files/afile.txt"

This string is then compared to the table, and finds the longest full match, in this case it would find "/usr/" and return channel 2, unit 2, plus the string "files/afile.txt", which is what is left over after subtracting.

The great thing about this whole "pathname space" approach is that processes don't necessarily need to know what they're dealing with, and pieces of the OS can be loaded and unloaded at will for the ultimate in scalability and modularity.

You might think that setting up the request and dealing with the responses, every time you want to open a file is a bit tiresome, but it's all handled for you with the "open" library call.

```
                pea O_READ
                pea ^devcon1
                pea !devcon1
                jsr @_open      ; returns file number in x or -1 on failure
                pla
                pla
                pla

                ...

devcon1         .asc "/dev/con/1",0
```

That's all for now. In the next article, i'll be writing about process loading + shared libraries, networking, terminal IO (console + modems) + some other things...

Hopefully you will have learned something from this article, and can see the power that a real multitasking OS, such as JOS, can bring to the SuperCPU.

Any feedback goes to jmaginni@postoffice.utas.edu.au , i'm particularly on the lookout for people who can help with hardware; docs, code etc...
Also, check the JOS homepage at http://www.jolz64.cjb.net/ and join the JOS mailing list if you're interested in updates.
....
....
..
.

::

VIC KERNAL Disassembly Project - Part III
Richard Cini
September 1, 1999

Introduction

In the last installment of this series, we examined the two remaining hard-coded processor interrupt vectors, the IRQ and NMI vectors. Although we took a complete look at the routines, we did not examine some of the subroutines that IRQ and NMI call. We'll examine these routines first.

Having completed the main processor vectors, we'll continue this series by examining other Kernal routines.

Remaining Subroutines

The NMI and IRQ routines together call 11 subroutines, five of which we previously examined in Part I of this series, and two call the NMI vectors in the BASIC ROM and A0 Option ROM. So, let's examine the four remaining subroutines.

UDTIM/IUDTIM

The IRQ vector calls the update time function UDTIM through the jump table at the end of the Kernal ROM, while the NMI function skips the intermediate call through the jump table and directly calls the time function.

```
UDTIM:
FFEA 4C 34 F7    JMP IUDTIM                    ;$F734

F734   ;=============================================================
F734   ; IUDTIM - Update Jiffy Clock (internal)
F734   ;         Called by IRQ; no params; no return
F734   ;
F734           IUDTIM
F734 A2 00               LDX #$00
F736 E6 A2               INC CTIMR2      ;bump timer tick
F738 D0 06               BNE UDTIM1      ;not 0, move on (no roll)
F73A E6 A1               INC CTIMR1      ;rolled-over, INC next reg
F73C D0 02               BNE UDTIM1      ;not 0, move on (no roll)
F73E E6 A0               INC CTIMR0      ;rolled-over, INC next reg
F740
F740           UDTIM1                    ;done updating registers,
F740                                     ; check for 24hr roll
F740                                     ; A0-A2 hold max of 4F1A00
F740 38                  SEC             ;set carry
F741 A5 A2               LDA CTIMR2      ; get LSB
F743 E9 01               SBC #$01        ; minus 1
F745 A5 A1               LDA CTIMR1      ;
F747 E9 1A               SBC #$1A        ; minus 1Ah
F749 A5 A0               LDA CTIMR0      ;
F74B E9 4F               SBC #$4F        ; minus 4Fh
F74D 90 06               BCC UDTIM2      ; ok
F74F
F74F 86 A0               STX CTIMR0      ;24-hr roll-over, so reset
F751 86 A1               STX CTIMR1      ; registers to zero
F753 86 A2               STX CTIMR2
F755
F755               UDTIM2                ;no 24-hr rollover-continue
F755 AD 2F 91            LDA D2ORAH      ;check for STOP key
F758 CD 2F 91            CMP D2ORAH
F75B D0 F8               BNE UDTIM2      ;not same, check again
F75D
F75D 85 91               STA STKEY       ;same, save status and exit
F75F 60                  RTS
```

UDTIM is called every 1/60th of a second by the IRQ routine, and begins execution by incrementing each of the time-keeping registers in the Zero Page locations $A0 to $A2. As each is incremented, it is checked for roll-over (i.e., for the count exceeding the maximum allowed for the register). Taken together, the three consecutive memory locations make-up the "jiffy clock" (as the VIC's RTC is sometimes referred; a "jiffy" being 1/60 of one second).

At the label UDTIM1, the code checks for a 24hr roll-over. The three byte-sized registers (no pun intended) can store the 24-hour jiffy count of 5,184,000 decimal, or 4F1A00 hex. If the count exceeds this value, the registers are reset to zero.

The BASIC TI function accesses the jiffy clock, representing the count as a decimal number. Similarly, the TI$ function represents the jiffy clock as a 24-hour HH:MM:SS clock instead of a jiffy count.

UDTIM is also responsible for processing the STOP key on behalf of the IRQ and NMI routines, so if a user program handles either of these interrupts, the programmer must remember to call UDTIM in order to maintain the time clock and STOP key functionality.

CCOLRAM

This short routine is responsible for determining the location of the color ram. In the VIC, the screen and color memory locations change based on the amount of RAM installed, as follows:

Function	Unexpanded	Expanded
User BASIC	$1000 00010000	$1200 00010010
Screen Memory	$1E00 00011110	$1000 00010000
Color RAM	$9600 10010110	$9400 10010100

The two least significant bits of the most-significant byte of each of the screen memory and color RAM pointer registers defines the resulting location. If the bit pattern of the screen memory is "10", the code sets the color RAM base to page $96. If the bit pattern is "00", the code sets the color RAM base to page $94.

The two other possible bit patterns result from screen memory beginning at $1100 or $1F00, and produce color RAM locations of $9500 and $9700, respectively. The $1100 starting location will actually work, but result in 256 bytes of wasted user RAM. The $1F00 starting location will not work since the color RAM locations overlap the I/O Block 2 addresses, which have no RAM associated with them.

```
EAB2   ;=======================================================
EAB2   ; CCOLRAM - Calculate pointer to color RAM
EAB2   ;
EAB2        CCOLRAM
EAB2 A5 D1              LDA LINPTR      ;get ptr to screen RAM LSB
EAB4 85 F3              STA COLRPT      ;save it as color LSB
EAB6 A5 D2              LDA LINPTR+1    ;get screen RAM MSB
EAB8 29 03              AND #%00000011  ;mask bits 0-1
EABA 09 94              ORA #%10010100  ;OR with $94 to get color
EABA                                    ; RAM pointer
EABC 85 F4              STA COLRPT+1    ;save as color ptr MSB
EABE 60                 RTS             ;exit
```

ISCNKY

This is the low-level keyboard scan function which is called 60 times per second by the IRQ routine. ISCNKY scans the keyboard matrix to retrieve a keypress, maps the key number to its ASCII equivalent, and places the ASCII value at the end of the keyboard buffer. If IRQs are disabled, the keyboard scanning is suspended. ISCNKY is accessible to user programs through the Kernal jump table, although calling it with interrupts enabled is not recommended.

To retrieve a character from the keyboard, a user program would typically call GETIN ($FFE4), the buffered keyboard input routine. GETIN returns the ASCII value of the character at the head of the keyboard buffer, or zero if no character is available.

VIA2 is directly connected to the keyboard. Port B is used as the column strobe and Port A is used as the row input. To read the keyboard matrix, the code brings all column strobe lines to 0 and reads the row inputs, in order, until a key is found (or not found). The code also begins decoding the ASCII using the "unshifted" decoding table. Three other decoding tables are for shifted, C= (Commodore) keys, and shift+C= keys.

```
EB1E   ;=======================================================
EB1E   ; ISCNKY - Scan keyboard
EB1E   ;        Scans keyboard for character. Called by IRQ routine.
EB1E   ;  ASCII value placed in keyboard buffer.
EB1E            ISCNKY
EB1E A9 00              LDA #$00        ; set shft/ctrl flag to 0
EB20 8D 8D 02           STA SHFTFL
```

```
EB23 A0 40              LDY #$40        ; assume no keys pressed
EB25 84 CB              STY KEYDN       ;  ($40=no keys)

EB27 8D 20 91           STA D2ORB       ; bring all column bits low
EB2A AE 21 91           LDX D2ORA       ; read row inputs
EB2D E0 FF              CPX #$FF        ; any character keys pressed?
EB2F F0 5E              BEQ PROCK1A     ; no, exit

EB31 A9 FE              LDA #%11111110  ; begin testing at COL 0
EB33 8D 20 91           STA D2ORB       ; output bit pattern

EB36 A0 00              LDY #$00        ; zero character count reg

                                        ; set default translation
                                        ; table to Table 1
EB38 A9 EA              LDA #$EA        ;FIXUP2+2;#$5E
EB3A 85 F5              STA KEYTAB
EB3C A9 EA              LDA #$EA        ;FIXUP2+3;#$EC
EB3E 85 F6              STA KEYTAB+1
EB40
EB40            ISCKLP1                 ; begin testing loop
EB40 A2 08              LDX #$08        ; 8 rows to test in column
EB42 AD 21 91           LDA D2ORA       ; get column
EB45 CD 21 91           CMP D2ORA       ; test again - debounce
EB48 D0 F6              BNE ISCKLP1     ; not equal, retry
EB4A
EB4A            ISCKLP2                 ; got bit pattern
EB4A 4A                 LSR A           ; shift through carry flag
EB4B B0 16              BCS ISCNK1+3    ; CY=1 for key not pressed
EB4D
EB4D 48                 PHA             ; save column bit pattern
EB4E B1 F5              LDA (KEYTAB),Y  ; .Y is index into ASCII
EB4E                                    ;  translation table
EB50 C9 05              CMP #$05        ; ASCII > 5, move on
EB52 B0 0C              BCS ISCNK1      ;  (<5=shft, C=, STOP, CTRL)
EB54
EB54 C9 03              CMP #$03        ; ASCII=3 STOP key
EB56 F0 08              BEQ ISCNK1      ; got STOP so skip flag updt
EB58
EB58 0D 8D 02           ORA SHFTFL      ; save SHFT, CTRL, C= flag
EB5B 8D 8D 02           STA SHFTFL
EB5E 10 02              BPL ISCNK1+2    ; move on to next row in col
EB60
EB60            ISCNK1
EB60 84 CB              STY KEYDN       ; save key#
EB62 68                 PLA             ; restore col bit pattern
EB63 C8                 INY             ; increment key count
EB64 C0 41              CPY #$41        ; 64 keys scanned?
EB66 B0 09              BCS ISCNEXIT    ; yes, return ASCII value
EB68
EB68 CA                 DEX             ; go on to next row in col
EB69 D0 DF              BNE ISCKLP2     ;  {loop}
EB6B
EB6B 38                 SEC             ; done with first column, so
EB6C 2E 20 91           ROL D2ORB       ;   move on to next column
EB6F D0 CF              BNE ISCKLP1     ;  {loop}
EB71
EB71            ISCNEXIT                ; function evaluation vector
EB71 6C 8F 02           JMP (FCEVAL)    ; CINT1A points this to SHEVAL
EB71                                    ; the shift evaluation code
EB74            ;
EB74            ; Process key image
EB74            ;
EB74            PROCKY
EB74 A4 CB              LDY KEYDN       ; get key number (as index)
EB76 B1 F5              LDA (KEYTAB),Y  ; covert key# to ASCII code
EB78 AA                 TAX             ; copy ASCII code to .X
EB79 C4 C5              CPY CURKEY      ; is it the same as the
                                        ;  current character?
EB7B F0 07              BEQ PROCK1      ; yes, do repeat eval
EB7D
EB7D A0 10              LDY #$10        ; set repeat delay
EB7F 8C 8C 02           STY KRPTDL
EB82 D0 36              BNE PROCK4      ; not same key, so exit
EB84
EB84            PROCK1
EB84 29 7F              AND #%01111111  ; test for {REVERSE}
EB86 2C 8A 02           BIT KEYRPT      ; do test
EB89 30 16              BMI PROCK2      ;  BIT7 set? reverse only
EB8B 70 49              BVS PROCK5      ;  BIT6 set? alpha or reverse
EB8D
```

```
EB8D C9 7F              CMP #$7F        ; last non-revs'd character
EB8F
EB8F            PROCK1A
EB8F F0 29              BEQ PROCK4
EB91
EB91 C9 14              CMP #$14        ; {DEL}?
EB93 F0 0C              BEQ PROCK2      ;  process {DELETE}/INS
EB95
EB95 C9 20              CMP #$20        ; {SPACE}?
EB97 F0 08              BEQ PROCK2      ;  process {SPACE}
EB99
EB99 C9 1D              CMP #$1D        ; {<-}?
EB9B F0 04              BEQ PROCK2      ;  process cursor right/L
EB9D
EB9D C9 11              CMP #$11        ; {CRS DN}?
EB9F D0 35              BNE PROCK5      ;  process cursor down/U
EBA1
EBA1            PROCK2
EBA1 AC 8C 02           LDY KRPTDL      ; get repeat delay
EBA4 F0 05              BEQ PROCK3      ;  if 0, check repeat speed
EBA6
EBA6 CE 8C 02           DEC KRPTDL      ; not done delaying, so exit
EBA9 D0 2B              BNE PROCK5      ;  {exit}
EBAB
EBAB            PROCK3
EBAB CE 8B 02           DEC KRPTSP      ; decrement repeat speed cnt
EBAE D0 26              BNE PROCK5      ; not done delaying, so exit
EBB0
EBB0 A0 04              LDY #$04        ; delay speed cnt reached 0,
                                        ;  so reset speed count
EBB2 8C 8B 02           STY KRPTSP      ; save it
EBB5 A4 C6              LDY KEYCNT      ; get count of keys in kbd
                                        ;  buffer
EBB7 88                 DEY             ; at least one, so exit
EBB8 10 1C              BPL PROCK5      ;  {exit}
EBBA
EBBA            PROCK4
EBBA A4 CB              LDY KEYDN       ; get current key number
EBBC 84 C5              STY CURKEY      ; re-save as current
EBBE AC 8D 02           LDY SHFTFL      ; get current shift pattern
EBC1 8C 8E 02           STY LSSHFT      ; save as last shft pattern
EBC4 E0 FF              CPX #$FF        ; re-check for any keys down
EBC6 F0 0E              BEQ PROCK5      ; none, so exit
EBC8
EBC8 8A                 TXA             ; restore ASCII code to .A
EBC9 A6 C6              LDX KEYCNT      ; get count of keys in buffer
EBCB EC 89 02           CPX KBMAXL      ; more than maximum allowed?
EBCE B0 06              BCS PROCK5      ; yes, drop current key press
EBD0
EBD0 9D 77 02           STA KBUFFR,X    ; save ASCII code in buffer
EBD3 E8                 INX             ; increment buffer count and
EBD4 86 C6              STX KEYCNT      ;   save it
EBD6
EBD6            PROCK5
EBD6 A9 F7              LDA #$F7        ; clear bit for COL3 (STOP key
EBD8 8D 20 91           STA D2ORB       ; is in COL3); save it to VIA
EBDB 60                 RTS             ; exit routine
```

Part of the keyboard scanning includes evaluating whether or not key modifier keys are pressed. Modifier keys include the SHIFT, Commodore, and CTRL keys. The ASCII decoding table is changed based on whether or not one of these keys is pressed. It also looks like the following code went through several revisions considering the multiple patch areas (filled with NOPs). Alternatively, these areas could support alternate decoding schemes for different languages.

```
EBDC            ;
EBDC            ; Evaluate for shift/CTRL/Commodore keys
EBDC            ;
EBDC            SHEVAL
EBDC AD 8D 02           LDA SHFTFL      ; 1=SHFT; 2=C> 4=CTRL
EBDF C9 03              CMP #$03        ; C> + shft?
EBE1 D0 2C              BNE PROCK6A     ; no, select proper decode
EBE3                                    ;  table
EBE3 CD 8E 02           CMP LSSHFT      ; is the pattern the same as
EBE6 F0 EE              BEQ PROCK5      ; last one? Yes, exit.
EBE8
EBE8 AD 91 02           LDA SHMODE      ; different pattern
EBEB 30 56              BMI PROCKEX     ;  {exit}
EBED
```

```
EBED EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EBF3 EAEA
EBF5 EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EBFB EAEA
EBFD EA EA EA           .db $ea, $ea, $ea
EC00
EC00 AD 05 90           LDA VRSTRT      ; get char ROM address
EC03 49 02              EOR #%00000010  ; flip between L/C and U/C
EC05 8D 05 90           STA VRSTRT      ;  ROMs
EC08
EC08 EA EA EA EA        .db $ea, $ea, $ea, $ea
EC0C
EC0C             PROCK6                 ; proper ROM is set, so go
EC0C 4C 43 EC           JMP PROCKEX     ;  on with key image process
EC0F
EC0F             PROCK6A                ; define correct decode table
EC0F 0A                 ASL A           ; multiply index by 2
EC10 C9 08              CMP #$08        ; >= 8 (5 entries)?
EC12 90 04              BCC $+6         ; no, continue
EC14
EC14 A9 06              LDA #$06        ; yes, assume CTRL table
EC16
EC16 EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EC1C EAEA
EC1E EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EC24 EAEA
EC26 EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EC2C EAEA
EC2E EAEAEAEAEAEA       .db $ea, $ea, $ea, $ea, $ea, $ea, $ea, $ea
EC34 EAEA
EC36 EA EA              .db $ea, $ea
EC38
EC38 AA                 TAX             ; reset pointer to point
EC39 BD 46 EC           LDA KDECOD,X    ;  at right decoding table
EC3C 85 F5              STA KEYTAB      ;  .A is table index
EC3E BD 47 EC           LDA KDECOD+1,X
EC41 85 F6              STA KEYTAB+1
EC43
EC43             PROCKEX
EC43 4C 74 EB           JMP PROCKY      ; continue processing image
EC46

EC46            ;=======================================================
EC46            ; KDECOD - Pointers to keyboard decode tables
EC46            ;
EC46             KDECOD
EC46 5E EC              .dw KDECD1              ;$EC5E Unshifted
EC48 9F EC              .dw KDECD2              ;$EC9F Shifted
EC4A E0 EC              .dw KDECD3              ;$ECE0 Commodore
EC4C A3 ED              .dw KDECD5              ;$EDA3 Control
EC4E 5E EC              .dw KDECD1              ;$EC5E Unshifted
EC50 9F EC              .dw KDECD2              ;$EC9F Shifted
EC52 69 ED              .dw KDECD4              ;$ED69 Decode
EC54 A3 ED              .dw KDECD5              ;$EDA3 Control
EC56 21 ED              .dw GRTXTF              ;$ED21 Graphics/text control
EC58 69 ED              .dw KDECD4              ;$ED69 Decode
EC5A 69 ED              .dw KDECD4              ;$ED69 Decode
EC5C A3 ED              .dw KDECD5              ;$EDA3 Control
```

Now, let's look at a few very simple routines just so that we can check them off of the list:

IIOBASE
=======

IIOBASE is the internal label behind the Kernal IOBASE function. Calling IOBASE results in code execution being transferred to IIOBASE:

```
IOBASE:
FFF3 4C 00 E5           JMP IIOBASE             ;$E500 IOBASE
```

IOBASE returns the address of the beginning of the I/O region of the VIC memory map in the .X and .Y registers. Locations $9110 to $912F are the addresses reserved for the VIC's two 6522 VIAs. This is the first routine in the Kernal ROM.

The value of this function in the VIC is questionable since there is no way to change the address at which the VIAs appear, and interestingly, the Kernal code does not call IOBASE at all. The Kernal instead relies on hard-coded addresses.

However, one could conclude that the actual location of the VIAs in the VIC's address space changed during the Kernal development process, so IOBASE was somehow used to normalize the address. This also enabled code portability between the VIC and the C64.

The BASIC ROM appears to call IOBASE in the RND function. The existence of other calls is unknown at this time since the BASIC ROM has yet to be disassembled.

```
E500    ;============================================================
E500    ; IIOBASE - Return I/O base address
E500    ;       Returns the IO Base address in .X(LSB) and .Y(MSB)
E500            IIOBASE
E500 A2 10              LDX #$10        ;return $9110 as IO Base
E502 A0 91              LDY #$91
E504 60                 RTS
```

ISCREN

```
FFED 4C 05 E5           JMP ISCREN      ;$E505 SCREEN

E505 ;============================================================
E505 ; ISCREN - Return screen organization
E505 ;  Returns the screen organization .X(columns) and .Y(rows)
E505 ;
E505            ISCREN
E505 A2 16              LDX #$16       ;return 22 cols x 23 rows
E507 A0 17              LDY #$17
E509 60                 RTS
```

This code returns the row and column organization of the screen in the .X and .Y registers. It doesn't appear that the Kernal calls this function to determine the screen size, instead relying on hard-coded values under the assumption that the screen is 22x23. So, this function's utility appears to be purely for the benefit of user code.

IPLOT

```
FFF0 4C 0A E5           JMP IPLOT               ;$E50A

E50A    ;============================================================
E50A    ; IPLOT - Read/set cursor position
E50A    ; On entry:  SEC to read cursor position to .X(row) and .Y(col)
E50A    ;            CLC to save cursor position from .X(row) and .Y(col)
E50A    ;
E50A            IPLOT
E50A B0 07              BCS READPL      ;carry set? yes, read position
E50C 86 D6              STX CURROW      ;save row...
E50E 84 D3              STY CSRIDX      ;...and column
E510 20 87 E5           JSR SCNPTR      ;update position
E513
E513            READPL
E513 A6 D6              LDX CURROW      ;return row...
E515 A4 D3              LDY CSRIDX      ;...and column
E517 60                 RTS
```

The Kernal again does not call this function, instead managing cursor movement by changing the values of the current row and current cursor index (i.e., the cursor's position in the row). Upon storing the new cursor location, the code commits the changes by jumping to an internal routine in CINT1 which is responsible for moving the cursor block in screen memory.

Conclusion

In this installment, we examined several routines, two of which are integral to the operation of the VIC. The Jiffy clock routine also scans the STOP key, which is important to overall usability and the ability

to halt a program. The second routine, SCNKEY, is responsible for scanning the keyboard matrix. That's pretty important, too.

Next time, we'll examine more routines in the VIC's KERNAL, including I/O routines.

....
....
..
.

::

JPEG: Decoding and Rendering on a C64

---------- Stephen Judd
<sjudd@ffd2.com>
Adrian Gonzalez
<adrianglz@globalpc.net>

In the C64 world there are a disturbing number of cases where people have said, "It can't be done on a C64." This goes on for a while until someone actually takes a look at the task and its requirements, and says "Not only can it be done, but it can be done easily." JPEG is one such case.

This article is divided into two parts. In part 1, I discuss JPEGs and the decoding process. The primary focus is on several important issues not covered well, if at all, in existing documentation, especially the IDCT; the article also covers the principles of decoding JPEGs and JFIF files.

In part 2, Adrian discusses Floyd-Steinberg dithering, and how it can be applied to various C64 graphics modes (and how it can be used to display jpegs!). In both articles the actual C64 code and algorithms will of course be discussed, and the source code is available at

http://www.ffd2.com/fridge/jpeg

for both the decoder and the renderer.

The decoder is about 4k of code, the renderer is around 2k, and there are about 9k of tables. With the grayscale versions, there is ample memory left over. With the color IFLI versions, memory is extremely tight -- there are 32k of graphics, six 24-bit image buffers. The Huffman trees are stored in the screen RAM area. The renderer crams all the data into the graphics area, which is why you see garbage while the image is rendering. There are a few tens of bytes free in page 0, probably 100-200 bytes free in page 1, and a few tens of bytes free in page 2, and that's it! Everything else just kind-of barely/exactly fits, and then only for 'typical' jpegs.

Finally, Errol Smith deserves a special mention as the guy who first tracked down some decent JPEG documentation. Errol pointed me in the right direction and within a few weeks we had JPEGs on a 64.

Part I: Decoding jpegs

Decoding jpegs is a fairly straightforward process, and in recent years some free documentation has become available. This article is meant to complement that documentation, by filling in some of the gaps and detailing some of the broader issues, not to mention some specific implementation issues. The first part of this article covers general jpeg issues: encoding/decoding, Huffman tree storage, Fourier transforms, JFIF files, and so on. The second part covers implementation issues more specific to the C64.

There are several sources of JPEG documentation online and in the library. Out of all of them, I found three that were particularly useful:

Cryx's jpeg writeup at http://www.wotsit.org

ftp://ftp.uu.net/graphics/jpeg/wallace.ps.gz, an updated article from one which appeared in the April 1991 "Communications of the ACM" (v34 no.4).

"JPEG Still Image Data Compression Standard" by William B. Pennebaker and Joan L. Mitchell, published by Van Nostrand Reinhold, 1993, ISBN 0-442-01272-1.

The first, Cryx's writeup, is a programmer's description of JPEG files, so it has good, detailed descriptions of the encoding/decoding process and the file structure/organization, including a list of all the JFIF segments

and markers. The second reference is also excellent, and explains most of the basic principles of JPEGs, the how's and why's of the standard, and has some helpful examples. The third reference (the book) is very comprehensive, but is written in a way which I feel tends to obscure the important points. Nevertheless, it has an entire chapter on the discrete cosine transform and several fast DCT algorithms, which is invaluable. As an additional source of information, some people might find the IJG's cjpeg/djpeg source code helpful.

JPEG Encoding/Decoding

It's really simple, folks.

Start with a grayscale image and divide it up into 8x8 pixel blocks (just like a C64 bitmap). The first block is the upper-left corner of the image; the second block is to the right of the first block, and so on until the end of the row is reached, at which point the next row begins.

The next step is to take the two dimensional discrete cosine transform of each 8x8 component, and filter out the small-amplitude frequencies. This will be explained in detail later, but the net result is that you are left with a lot of zeros in the 64-byte data block, and a few nonzero elements from which you can reconstruct the main features of the image. This filtering process is called the "quantization" step.

The next step is to RLE-encode the resulting 8x8 block (since most of the components are zero), and finally to Huffman-encode the RLE-encoded data. And that's it. Done. Finished. Repeat Until Done.

Color pictures are similar, but now each pixel has an 8-bit R, G, and B value, so there will be three 8x8 blocks, for a total of 24 bits (not quite like a C64 bitmap...). The RGB values are converted to luminance/chrominance values (RGB -> YCrCb), but what's important is that for each 8x8 section of a color image there are three 64-byte blocks of data, and each block is encoded as above.

So to summarize: transform the data, filter ("quantize") the transformed data, and RLE-encode and Huffman-encode the result. Do this for each component, and then move on to the next 8x8 block. Therefore, to decode the image data:

read in the bits,
find the Huffman code,
unpack the RLE,
de-quantize the data,
and perform the inverse transform,

for each 8x8 block of image data to be plotted to the screen. Repeat until done.

It turns out that there are other methods of JPEG compression in the standard, such as arithmetic compression, but this is rarely supported due to legal reasons (lame software patent owned by IBM, AT&T, and Mitsubishi), and it doesn't seem to offer substantial compression gains. There are also different types of jpegs, most importantly "baseline" or sequential jpegs, and "progressive" jpegs. In a progressive jpeg the image is stored in a series of "scans" which go from lower to higher resolution. I'll be focusing on baseline jpegs (which are more common).

Finally, it turns out that an 8x8 block of image data doesn't have to correspond to an 8x8 block of pixels. For example, each byte of data might represent an average of a 2x2 block of pixels, so an 8x8 block of data might expand to a 16x16 block of pixels. In a JPEG the "sampling factor" determines how to expand an 8x8 block of data. You can see that this can offer substantial compression gains, but will coarsen the data; on the other hand, if the data is already coarse, it's a way of getting a whole lot for nothing. Most color jpegs use one-to-one pixel mapping for the luminance, and one-to-four (each data byte = 2x2 pixel block) mapping for the two chrominance components. From an implementation standpoint, this means that a decoder typically decodes 16 scanlines at a time (16x16 pixel chunks). For more details, see Cryx's document.

Before a JPEG can be decoded, though, the decoder needs a fair amount of information, such as the Huffman trees used, the quantization tables used, information about the image such as its size, whether it's a color or a grayscale image, and so on. In a JPEG file, all information is stored in "segments".

Segments

A JPEG segment looks like the following:

[header] Two bytes, starting with $FF

```
        [length]        Two bytes, in hi/lo order (not usual 6502 lo/hi)
        [data]          Segment data
```

A list of JPEG (and JFIF) headers can be found in Cryx's document.

Let's have a look at a hex dump of a jpeg file (from unix, use "od -tx1 file.jpg | more"):

```
0000000  ff d8 ff e0 00 10 4a 46 49 46 00 01 01 01 00 48
0000020  00 48 00 00 ff fe 00 17 43 72 65 61 74 65 64 20
0000040  77 69 74 68 20 54 68 65 20 47 49 4d 50 ff db 00
```

The first two bytes are $ff $d8 -- these two bytes identify the file as a jpeg. All jpegs start with ff d8.

Next we encounter the header ff e0. ff e0 is a special header which identifies this file as a JFIF file. It turns out that in the original JPEG standard a specific file format is not given; this in turn led to different companies using their own formats, to try and establish the "standard". The JFIF format was put forwards to remedy this problem, and is the de-facto standard -- but more on this later.

In a JFIF file, the JFIF segment always follows the JPEG ID byte. You can see that it is length 16, and that that length includes the two length bytes. Immediately following the length byte are the four letters J F I F and the number 0; following that are some bytes for revision numbers, the x/y densities, and some thumbnail info.

The next segment starts with the header ff fe. This is the "comment" header; the length is $17 bytes. Following the length bytes are the ascii codes for "Created with The GIMP", a popular image processing program. The next header is ff db, which is the "Define Quantization Table" header. And on it goes, until the actual image data -- a stream of Huffman-encoded bits -- is reached.

Huffman Decoding

If you don't know anything about Huffman decoding, then I suggest you read Pasi's nice article in C=Hacking #16, which has a nice example. Briefly, a Huffman tree is a binary tree whose left and right branches correspond to bits 0 and 1 respectively; starting from the top of the tree, you read bits and move left or right accordingly until a leaf is reached, containing the Huffman code value. Then you start over again at the top of the tree and decode the next Huffman code.

In a JPEG, Huffman trees are stored in "Define Huffman Tree" segments (header = ff c4):

```
0000300                                 ff c4 00 1c 00 00
0000320  01 05 01 01 01 00 00 00 00 00 00 00 00 00 03
0000340  01 02 04 05 06 00 07 08
```

The first byte in the DHT segment (00) is an ID byte -- JPEGs can have up to eight Huffman trees. This is then followed by 16 bytes, where each byte represents the number of Huffman codes of lengths 1, 2, 3, ..., up to length 16, followed by the Huffman code values. In the above example, there are 0 codes of length 1, 1 code of length 2, 5 codes of length 3, and so on. Following these 16 bytes are the Huffman values: 3, 1, 2, 4, ..., 8. But what are the Huffman codes corresponding to those values?

It turns out that these trees are so-called "canonical Huffman trees", and work as follows: to get the next code, add 1 to the current code. When the length increases, add 1 and shift everything left. The exception is that you don't increment until the first code is defined, so the first code is always zeroes.

For example, to decode the above DHT segment, start with Huffman code = 0. There are no codes of length 1, so we shift it left to get code = 00 (and don't add 1 because the first code hasn't been defined yet). There is one code of length 2, so we read the first Huffman value and assign it to the current code

```
        Code    Value
        00        3
```

That's the only code of length two, so now we move to length 3 by incrementing and shifting: code = 010. There are five values of length 3, and the next five Huffman values are 1, 2, 4, 5, 6, so the Huffman tree is now

```
        Code    Value
        00        3
        010       1
        011       2
        100       4
        101       5
```

```
        110       6
```

and the rest of the Huffman tree is given by

```
        1110      0
        11110     7
        111110    8
```

What's the best way to implement a Huffman tree?

The most obvious way is to use five bytes per "node", i.e.

```
        left pointer    (2 bytes)
        right pointer   (2 bytes)
        value           (1 byte)
```

where the left and right pointers are just offsets to be added to the current pointer, and if left = right = $FFxx then this is a leaf. If you fetch a bit that says "go left", and the left pointer = $FFxx (but right pointer is valid) then you've hit an invalid Huffman code -- i.e. decoding error. This five-byte method is used in jpx (grayscale decoder).

But there is another rather cool method, first described to me by Errol Smith, which uses only two bytes per node. Now, the five-byte method works fine in jpx, but in the full-color IFLI jpz code -- well, suddenly memory becomes extremely tight, and without this routine jpz probably wouldn't have happened on a stock machine. The routine is also very efficient, especially if implemented using 16-bit 65816 code.

The trick is simply to organize the tree such that if the current node is at location NODE, then the left node is at NODE+2 and the right node is at NODE+(NODE). Leaf nodes can be indicated by e.g. setting the high bit. So the decoding process is:

```
        get next bit
        if 0 then pointer = pointer + 2
        if 1 then pointer = pointer + node value
        if high byte of node value < $80 then loop
```

For example, the first part of the earlier Huffman tree

```
        00        3
        010       1
        011       2
        100       4
```

would be encoded as

```
0d 00 04 00 03 80 04 00 01 80 02 80 00 00 00 00 04 80
-----|-----|-----|-----|-----|-----|-----|-----|-----|
```

Try decoding the Huffman values, using the above algorithm.

Astute readers may ask the question: won't you decode incorrectly if there is no left node? Even more astute readers can answer it: in a canonical Huffman tree, the only nodes without left-node pointers are leafs.

To see this, consider a counterexample: a tree that looks like

This corresponds to Huffman code 01 -- one move left, one move right. In a canonical Huffman tree, the only way to generate the code 01 is to increment the code 00; since code 00 has already occured, there must be a left-node. In a canonical Huffman tree, you always create a left-node before creating a right-node. So error checking this kind of tree amounts to checking the right-pointer; the only nodes without left-pointers are leafs. Moreover, since left-nodes are always created first, you can add nodes in the order they are created -- you never have to insert nodes between existing nodes.

Pretty nifty, eh?

Restart Markers

The image data in a jpeg is a stream of Huffman-encoded bits. The jpeg standard allows for "restart markers" to be perodically inserted into the stream. Thus a decoder needs to keep count of how far it is in the data stream, and periodically re-synchronize the bitstream. So

far so good -- this is explained in detail in Cryx's document.

What _isn't_ explained is that the restart markers do not merely re-synchronize the data stream, but when a restart marker is hit the DC coefficients need to be reset to zero. That is, it really does "restart" the decoder.

What's a DC coefficient, you may ask? It's the very first element in the 8x8 array, and instead of encoding the actual value a jpeg encodes the _offset_ from the previous value. That is, the decoded DC element is added to the current DC value to get the new value. That value needs to be reset to zero when a restart marker is hit.

Most jpegs do not use restart markers, but unless you reset the coefficient you're going to spend a few months wondering why Photoshop images don't decode correctly.

Why is it called the DC coefficient? You'll have to read the section on Fourier transforms for the answer.

Note also that when the byte $FF is encountered in the data stream it must be skipped; the exception is if it is immediately followed by a 00, in which case $FF00 represents the value $FF. Why do I bring this up? Because Cryx's document could be interpreted by naive people like myself as saying this is true throughout a jpeg file, and it's only true within the image data -- that in other segments, $FF is a perfectly valid byte.

Unpacking the RLE

Once a Huffman code is retrieved and decoded, the resulting byte represents RLE-compressed data to be uncompressed. This procedure is described quite well in Cryx's document, so I'll just refer you to it. This is repeated until you are left with a 64-byte chunk of data which needs to be re-ordered and dequantized. This process is again described in Cryx's document; briefly, during the encoding process, the original 8x8 data is re-ordered into a 64-byte vector as follows:

```
0  1  5  6  ...
2  4  7  13 ...
3  8  12 17 ...
9  11 18 24 ...
10 19 23 ...
20 22 ...
...
```

That is, the first element in the vector is the (0,0) component of the 8x8 array, the next element is the (1,0) component, the next element is the (0,1) component, and so on. The reason for this "zig-zag" ordering is to enhance the RLE-compression, since it concentrates the lower frequencies at the beginning of the vector and the higher frequencies -- most of which are typically zero-amplitude -- at the end of the vector (more on this later). The decoder thus needs to "un-zigzag" the vector back into an 8x8 array. All de-quantization amounts to is multiplying each element by a corresponding element in a quantization table:

```
data[i,j] = data[i,j]*quant[i,j]
```

The final step is to take the resulting 64-byte chunk and apply the inverse discrete cosine transform (IDCT).

Fourier Transforms and the (I)DCT

Let's begin with the definition you'll see in any document on JPEGS (hear that? That's the sound of one thousand eyes simultaneously glazing over).

OK, let's back up a moment. In computers, grasping new ideas is usually straightforward: you read about it, play around with it a little, and ah, it makes sense. Mathematics isn't like that. These are ideas that took people decades and centuries to figure out. College students spend multiple months, working hundreds of problems, to gain just a basic working knowledge of a subject. There's simply a constant learning process. Fourier transforms represent a fundamentally different way of thinking, and the timescale for enlightenment in the subject is years, not minutes. So don't worry if you don't understand everything immediately; the purpose of this part isn't to make you an instant expert in Fourier transforms, but rather to give you a toehold into the subject that you can expand on over time.

So, let's begin with a definition that you'll see in any document on JPEGS. The one-dimensional discrete cosine transform (DCT) of a function f(x) with eight points (x=0..7) may be written as

$$F(u) = c(u)/2 * \sum_{x=0}^{7} f(x) * \cos(\frac{2*x+1}{16}*u*PI), \quad u = 0..7$$

where c(0) = 1/sqrt(2) and c=1 otherwise. This may look very mysterious to you, and it should, because it is rather mysterious-looking. For now, think of it as some sort of grinder: you insert f(x) into the grinder, turn the crank, and out pops a new function, F(u). In other words, the original function f(x) has been _transformed_ into a new function F(u).

Notice that we need to perform a separate sum for each value of u:

```
F(0) = 1/(2*sqrt(2)) * sum f(x)
F(1) = 1/2           * sum f(x)*cos((2*x+1)*PI/16)
F(2) = 1/2           * sum f(x)*cos((2*x+1)*2*PI/16)
```

and so on. So there are a total of eight summations, each of which involves eight summands, for a total of 64 operations to perform.

One of the important properties about this transform is that it is _invertible_. That is, you can take a transformed function F(u), put it into the other end of the grinder, turn the crank backwards, and out pops the original function f(x). Moreover it is _uniquely_ invertible -- for every function f(x), there is one and only one transform F(u), and vice-versa (the functions f(x) and F(u) are often called a transform pair). In this case, the inverse DCT (IDCT) is given by

$$f(x) = 1/2 * \sum_{u=0}^{7} c(u)*F(u) * \cos(\frac{2*x+1}{16}*u*PI), \quad x = 0..7$$

You'll notice that it is very similar to the forward transform, except now the sum is over u, and c(u) is inside of the summation; as before, there are 64 sums total to perform. Expanding the sum gives

```
f(x) = 1/2 * ( 1/sqrt(2) F(0) + F(1) * cos((2*x+1)*PI/16) +
                                F(2) * cos((2*x+1)*2*PI/16) + ...)
```

For now, just note that the original function f(x) is given by a sum of the transformed function F(u) times different cosine components.

The transform of a two-dimensional function f(x,y) is done by first taking the transform in one direction (e.g. the x-direction) followed by the transform in the other direction (e.g. the y-direction). Thus the two-dimensional 8x8 discrete cosine transform of a function f(x,y) may be written as

$$F(u,v) = \frac{c(u)c(v)}{4} * \sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * \cos(\frac{2*x+1}{16}*u*PI) * \cos(\frac{2*y+1}{16}*v*PI)$$

$u,v = 0,1,...,7$

where, as before, c(0) = 1/sqrt(2) and c=1 otherwise. The IDCT is then given by

$$f(x,y) = \frac{1}{4} * \sum_{u=0}^{7}\sum_{v=0}^{7} c(u)c(v)*F(u,v) * \cos(\frac{2*x+1}{16}*u*PI) * \cos(\frac{2*y+1}{16}*v*PI)$$

$x,y = 0,1...7$

Note that some documentation (e.g. Cryx's document) incorrectly gives c(u) and c(v) as c(u,v) = 1/2 for u=v=0 and c(u,v) = 1 otherwise.

This is an _extremely_ expensive computation to do, requiring 64 multiplies of cosines (and computations of the arguments of the cosines) to calculate the value at a _single_ point (x,y), and there are 64 points in each 8x8 block, so, even discounting the argument computation (i.e. u*pi*(2*x+1)/16) we're looking at 64*64 = 4096 multiplications for _every_ 8x8 block of pixels (where these are 16-bit multiplications). On a C64, in such a case, the decoding time could be measured in hours if not days.

But if this were the only way to compute a DCT, then JPEGs would never have been DCT-based. There are much faster methods of computing Fourier transforms, that take advantage of the symmetries of the transform. You may have heard of the Fast Fourier Transform, which is used in almost all spectral computing applications; well, there are also fast DCT algorithms. The one I used is actually an adaptation of the FFT.

So the first task is: where do we find a fast DCT algorithm? One place to look is existing source code, like cjpeg/djpeg. Unfortunately I found it pretty incomprehensible, and hence tough to translate to 65816; it is also pretty large. And it's basically impossible to debug a routine that isn't understood (if something goes wrong, then where's the error?).

The next place to look is the literature -- many papers have

been written on fast DCT routines. Unfortunately, the ones I found were quite dense, very general (we only need an 8x8 routine, not an NxN routine), and again, fairly complicated.

What is needed is a _simple_, but fast, IDCT algorithm. Salvation came in the book by Pennebaker and Mitchell, mentioned at the beginning of the article and available in the library. This book has several 8x8 DCT routines in it, with detailed discussions of the algorithms, both one-dimensional and two-dimensional. The 2D one is again fairly lengthy, but the 1D ones are pretty fast and straightforward -- something like 29 adds and 13 multiplies to compute 8 components of a 1D DCT. Moreover the 13 "multiplies" are multiplies by constants, which means table lookups, not full multiplications. Compare with at least 1024 full multiplies and adds using the DCT definition, and you can see that the fast routine is *hundreds* -- and possibly thousands -- of times faster. To put this in perspective, it's the difference between taking 30 seconds to decode a picture and taking 1-2 hours -- maybe even 10 hours or more -- to decode the same picture!

As mentioned earlier, we can do a 2D IDCT by doing a 1D transform of the rows of some 2D array followed by a 1D transform the columns (or vice-versa). Thus a 1D routine is all that is needed. Although there are specialized 2D routines, they are quite large and significantly more complicated than a 1D routine. Small and straightforwards Good; large and complicated Bad. And cjpeg/djpeg makes the observation that they don't seem to give much speed gain in practice.

There's just one problem -- the book chapter discusses lots of _forwards_ DCT routines, but devotes just one paragraph to _inverse_ DCT routines! "Just reverse the flowgraph" is the advice given, with a few hints on reversing flowgraphs.

To make a long story short, IF you reverse the flowgraph correctly, AND you overcome the errors/misleading notation in the book, AND you prepare the coefficients correctly before performing the transformation, then yes, by golly, it works! And working code is awfully sweet after days of intense frustration! I have included an easy-to-read Java version of the 1D IDCT routine at the end of this article.

At this point, the more experienced programmers are asking, how do you _know_ it works? With so many possible 8x8 arrays, how do you test and debug such a routine? To answer these questions, it is important to understand a few things about Fourier transforms. In the process, we shall also see why JPEG is based on the DCT, and why it is so effective at compressing images.

Fourier Transforms for dummies

There are several ways of thinking about a Fourier transform. One way to think about it is that you can expand any function in a series of sines and cosines:

```
f(x) = a0 + a1*cos(wx) + a2*cos(2wx) + a3*cos(3wx) + ...
          + b1*sin(wx) + b2*sin(2wx) + b3*sin(3wx) + ...
```

where the a0 a1 etc. are constant coefficients (amplitudes) and w 2w etc. are the frequencies. In the discrete cosine transform, the function is expanded solely in terms of cosines:

```
f(x) = a0 + a1*cos(wx) + a2*cos(2wx) + a3*cos(3wx) + ...
```

"Taking the transform" amounts to computing the coefficients a0, a1, a2, etc. Once you know them, you can reconstruct the original function by adding up the cosines.

Now, let's forget about computing the coefficients, and stand back for a moment and look at that expression. Each coefficient tells "how much" of f(x) is in each cosine component -- for example, the value of a2 says "how much" of f(x) is in the cos(2wx) component. Conversely, each coefficient tells us how much of each "frequency" there is in f(x) -- a2 says how much frequency=2w there is, a0 says how much frequency=0 there is, and so on.

So another way of thinking about a Fourier transform is that it transforms a function from the space (or time) domain into the _frequency_ domain -- instead of thinking about how much the function varies with x (how it varies in space), we can see how it varies with _w_, the frequency; instead of looking at "how much f" is at a given point in space or time, we can look at "how much f" is at a given frequency.

So, imagine measuring something simple, like the voltage coming out of a wall socket. A plot of the signal will be a sinusoidal function -- this is a graph of how the signal varies with time. The Fourier transform of this signal, however, will have a large spike at 60Hz (or 50Hz if you're in Europe or .au). Small amplitudes of other frequencies will probably be

seen, too, indicating noise in the signal. So a graph of how the signal varies with _frequency_ might look something like this:

That is, lots of zero or very small amplitude frequencies, and a large frequency amplitude at around 60/50Hz.

If you've ever seen an equalizer display on a stereo, you've seen a Fourier transform -- the lights measure how much of the audio signal there is in a given frequency range. When the bass is heavy, the lower frequencies will have large amplitudes. When there's some high instrument playing (or lots of distortion), then the high frequencies will have large amplitudes.

Now we can take this a few steps further. The frequencies convey a lot of information. For example, cos(wx) wiggles very slowly if w is small, and wiggles very rapidly if w is large (and it doesn't wiggle at all if w=0). (If you don't see this, just think of x as an angle which goes around a circle: if x goes around the circle once, then 7x goes around the circle seven times). Therefore, a function which changes slowly will have a lot of low-frequencies in the transform; a function which changes rapidly will have large high-frequency components (rapid wiggles give rapid changes).

The zero frequency is special. A constant function will have only the zero-frequency component (since cos(0x) is a constant). Moreover, the zero-frequency represents the average value of the function over a period of cosine -- this is easy to see because the average value of cos(x), cos(2x), etc. is 0 over a full period: it is above zero half of the time, and below zero the other half of the time, and the two halves cancel.

Now consider an image. A typical photograph changes fairly smoothly -- there aren't many sudden sharp changes from black to white. This means that the transform of some small area of the picture will have fairly large-amplitude low-frequencies, but not much in the way of high frequencies. If those small-amplitude high-frequencies are simply thrown away, then the image won't change much at all -- the high frequencies represent super-fine details of the picture. And that's why JPEG is a "lossy" algorithm, and why it gets such high compressions -- the idea is to throw away the fine details and the unnecessary components, and keep just the major features of the picture. It's also why JPEG isn't so great for things like line-art, where the image can change rapidly -- you may have noticed that things like slanted lines tend to get jagged in a jpeg.

The important point to remember is that high frequencies correspond to rapid changes in the image, low frequencies correspond to smooth changes, and the zero frequency is the "average" value. Because there were obviously electrical engineers on the JPEG comittee, the zero frequency is referred to as the "DC component" of the transform, and the nonzero frequencies are referred to as the "AC components" (for Direct Current and Alternating Current).

Finally, for completeness, note that there is a difference between a discrete Fourier transform and a continuous Fourier transform, namely that one gives the transform in terms of discrete frequencies (w, 2w, 3w, etc.) and the other gives the transform as a continuous function of frequency. When dealing with discrete data -- like an 8x8 set of values -- we necessarily use a discrete transform.

Now, how can you test a Fourier transform routine?

Fourier Transforms for smarties

The basic question is: how do we know if the IDCT is working correctly? Quite simply, by feeding it a problem we already know the answer to.

Remember that we are working with transformed data; each element represents the amplitude of a specific frequency. Imagine a transformed vector with a single nonzero element, for example, let a3=10 and all the other coeffs equal zero. What will the inverse transform of this vector be? Since a3 is the amplitude of cos(3x), the transform will simply be... a3*cos(3x)! Similarly, if a1 is the only nonzero coefficient, the transform will be a1*cos(x).

The above explanation actually isn't _quite_ right, because of the form of the IDCT used:

$$\frac{1}{2} \sum_{x=0}^{7} \cdots \cos\left(\frac{(2x+1)}{\cdots}\right)$$

$$f(x) = \frac{1}{2} * \sum_{u=0} c(u)F(u) * \cos\left(\frac{u\pi}{16}\right)$$

Now it should be easy to see that if, say F(3)=10, and all the other F's are zero, then the result of the transform -- whatever transform algorithm is used -- must be

$$f(x) = 5\cos(3\pi(2x+1)/16)$$

So, for a one-dimensional IDCT, it is easy to test each component separately and compare the result with the actual answer. But what about a 2D IDCT that has many nonzero components?

There are two important properties of Fourier transforms which come into play here. The first is that Fourier transforms are _linear_; a linear operator L satisfies

$$L(c*f1) = c*L(f1), \text{ where } c = \text{constant}$$
$$L(f1 + f2) = L(f1) + L(f2)$$

That is, constants factor out of the operator, and operating on the sum of two functions is the same as operating on each function separately and adding them together. As a simple example, consider the operators L1(x) = x and L2(x) = x^2. The first one satisfies the conditions above; the second one does not. Some other linear transforms you may be familiar with are rotations, and taking the derivative. You can test for yourself that the Fourier transform satisfies the above conditions; you can also look at the fast DCT algorithm and see that it only involves additions and multiplications by constants, which are all linear operations.

This property is enormously important here. It first says that we can multiply the transformed data by a constant, and the constant will just multiply the final answer; said another way, if F(3)=10 and all other F's are nonzero, then we know that F(3)=const*10 will work too, no matter what the constant is! So in testing one component at a time, you can pretty confidently say "F(3) works" (as opposed to "F(3)=10 works, and F(3)=11 works"). The _only_ thing that can cause problems is overflows and other _computer_ issues; the basic algorithm _cannot_.

Even more importantly, however, is that the transform of the sum of two functions is equal to the sum of the transforms. If we know that

$$F1 = (0,0,10,0,0,0,0,0)$$

works, and we know that the transform of

$$F2 = (0,0,0,10,0,0,0,0)$$

works, then we _know_ that the transform of

$$F1 + F2 = (0,0,10,10,0,0,0,0)$$

works! Moreover, since we can multiply each function by arbitrary constants, we know that the transform of

$$(0,0,a,b,0,0,0,0)$$

works, _no matter what a and b are_. So we can _completely_ test a 1D DCT simply by testing each component _separately_. The _only_ things that can cause problems are things like overflow, erronius multiplications, etc.

Now, what about a 2D IDCT? The way a 2D IDCT is computed is by first transforming in one direction (e.g. the x-direction), then transforming in the other direction (e.g. the y-direction). Therefore, we can compute the 2D IDCT by first transforming each row, then transforming each column (or vice versa).

Therefore, once the 1D IDCT works, so does the 2D IDCT.

So, to summarize: to test the routine completely we simply need to test each component of a 1D IDCT separately, and compare the result with the known answer.

And if you really want to test it on a 2D set of data, there is an example DCT array given in the Wallace paper (and the result of the inverse transform).

Quantization revisited

The quantization step filters out all the small-amplitude frequencies. A JPEG can have up to four quantization tables; each table is a 64-byte (8x8) set of integers. When encoding a JPEG, taking the DCT of an 8x8 block of data leaves an 8x8 block of amplitudes. Each amplitude is divided by the corresponding entry in the quantization table, thus filtering out the small amplitudes in a weighted fashion. The quantized amplitudes are then

re-ordered into a 64-byte vector which concentrates the lower frequences (the ones more likely to be nonzero) at the beginning of the vector, and the higher frequencies (more likely to be zero) at the end of the vector. This last step (zig-zag reordering) clearly increases the efficiency of the RLE encoding of the amplitude vector.

The decoder just reverses these steps -- it dequantizes the data (i.e. multiplies by the quantization coefficients) and re-orders the data, before performing the IDCT. Now, you may have noticed that the IDCT routine has to prepare the coefficients by multiplying (dividing) by a set of constants:

```
    for (int i=0; i<8; i++)
        F[i] = S[i]/(2.0*Math.cos(i*ang/2));
```

(This is done because the algorithm is actually an adapted FFT routine). In principle, this step can be incorporated into the de-quantization step, since dequantization is also just multiplying by constants. In a 1D transform this is very straightforward, but I see no way to extend it to the 2D transform. That is, it is possible to incorporate the above into the quantization such that, say, the row transforms will not need preparation, but the column transforms will still need the preparation. I did not feel that this was a very useful "optimization", and simply mention it here for completeness.

Note also that a wise programmer would replace the Math.cos calls above with constants, if the code were to be actually used in a decoder.

Miscellaneous

You may recall that all JPEGs begin with FFD8, and JFIF files immediately follow this with the FFE0 JFIF segment. Although most jpegs have the JFIF segment, some don't! For example, some digital cameras do not include a JFIF header. But the files decode just fine if you don't worry about it.

Moreover, be sure to skip unknown segments using the segment length byte -- as opposed to, say, moving forwards in the file until another valid segment header is found.

When reading some of the other jpeg documentation, you'll read that the byte $FF is a special byte, to be skipped (unless followed by $00). Just to be clear, this only applies to the image data -- $FF is a normal data byte within other segments. Similarly, restart markers only appear within the image data.

C64 Implementation

As you probably understand by now, and as we shall see below, jpegs on a C64 are far from being an impossible task. So to wrap up, this section will cover the main issues in implementing a jpeg decoder on a C64, and examine some of the comments regarding jpegs being "impossible" on a C64.

One frequently-heard comment was that a C64 doesn't have enough memory to decode a jpeg, so let's look at the numbers. From the preceding discussion, jpegs require memory for

1 - Quantization tables
2 - Huffman trees
3 - Image data

The quantization tables are 64 bytes each, and there are a maximum of four -- so, no big deal. Using the two-byte storage method, the Huffman trees typically take up around 1.5k, and using the 5-byte method they take on the order of 4k. The image data is stored in a jpeg on a row-by-row basis, where each row is some multiple of 8 lines large. The normal C64 display is 320 pixels wide, so that means an image buffer size of 320x8 = $0A00 bytes per 8 scanlines.

So, a few K for the Huffman tables, and a few K for the image buffers. I think you'll agree that these are hardly massive amounts of memory.

Now, as you may recall, the data decoded from a JPEG file is luma/chroma data -- Y (intensity) CrCb (chroma). For a grayscale picture, all that is needed is the intensity -- there's no need to convert to RGB. You may also recall that, because of sampling factors, a jpeg might decode to 16x16 blocks of data (or more), which means several 320x8 image buffers need to be available -- at $0A00 bytes/buffer, there's plenty of buffer space available.

For a full-color picture, however, all three components need to be kept, which means three buffers for each 320x8 row of data, which means $1E00 bytes per row. So there's still plenty of room for multiple buffers.

Until, that is, you throw IFLI into the mix -- but more on this later.
The bottom line is that jpegs really don't require much memory.

Another common comment was that the C64 was far too slow to do the necessary calculations, especially the discrete cosine transforms. As was stated earlier, the IDCT routine used in this program needs some 29 adds and 13 multiplies to do a 1D transform. More importantly, the "multiplies" are always multiplies by a constant -- which means they can be implemented using tables. So, we're talking 29 16-bit adds and 13 16-bit table-lookups for the IDCT, which is really pretty trivial.

Another important calculation is the dequantization, which means doing 64 integer multiplications per 8x8 data block. Each integer is 8-bits large (and the result can be 16-bits), and the multiplications are done using the usual fast multiply routine (let f(x)=x^2/4, then a*b = f(a+b)-f(a-b)), as described in all the C=Hacking 3D articles. Again, not a big deal.

So, in summary, the mathematical calculations are well within the grasp of the 64.

In fact, all the routines are quite straightforward -- only the IDCT routine is special.

One important issue is grayscale versus color. The first program released, jpx, is grayscale, and for several very good reasons. Grayscale is much faster to compute, since no RGB conversion needs to be done (the intensity Y is exactly the grayscale levels). It is more memory-efficient, since the color components may be thrown away, and the bitmap requirements are modest. And it is easier and faster to render.

With some pretty solid fundamental routines and a reasonable grasp of the important issues, color was a reasonably straightforward addition to the code, with just one problem: memory. IFLI requires 32k for the bitmaps. The IDCT routine uses some 6k of tables. At least two image buffers are needed, for almost 16k. The RGB conversion code uses table lookups. The renderer needs memory for image buffers and tables. The decoder needs memory for Huffman and quantization tables. When we added it all up, there just wasn't room.

With a little more thought and planning, though, a few things became clear: first, IFLI doesn't use the first three columns, which means the image buffers only need to be 296x8 x 3 components = $1BC0 bytes (instead of 320x8 x 3). Typical jpegs use a maximum sampling factor of 2, so using just two buffers requires $3780 bytes -- a savings of almost $0600 bytes over a 320-pixel wide bitmap. Moreover, the needs of the renderer came out to almost exactly 16K per bitmap, which means that all the data can be squished into the two IFLI bitmaps and sorted out later. So by scrimping here and saving there, and economizing on tables and rearranging memory, we were able to cram everything into 64k, with just a few hundred bytes to spare -- pretty neat.

And that, I think, sums up JPEG decoding on a C64.

```
/*
 * idct.java -- Attempts to implement the IDCT by reversing the flowgraph
 * as given in Pennebaker & Mitchell, page 52.
 *
 * Almost there!
 *
 * SLJ 9/15/99
 */

import java.lang.Math.*;
import java.io.*;
import java.util.*;

public class idct2d {

    // a1=cos(2u), a2=cos(u)-cos(3u), a3=cos(2u), a4=cos(u)+cos(3u), a5=cos(3u)
    // where u = pi/8

    static double ang = Math.PI/8;

//    static double a1=0.7071, a2= 0.541, a3=0.7071, a4=1.307, a5=0.383;
    static double       a1 = Math.cos(2.0*ang),
                        a2 = Math.cos(ang)-Math.cos(3.0*ang),
                        a3 = Math.cos(2.0*ang),
                        a4 = Math.cos(ang)+Math.cos(3.0*ang),
                        a5 = Math.cos(3.0*ang);

//    static double[] f = {31, 41, 52, 65, 83, 15, 34, 117},
    static double[] f = {10, 9.24, 7.07, 3.826, 0, -3.826, -7.07, -9.24},
             F = {0, 0, 0, 0, 0, 0, 0, 256},
             S = {0, 0, 0, 0, 0, 0, 0, 256};
```

```
  static double[][] trans = new double[8][8];

  void idct2d() {}

void calcIdct() {
  double t1, t2, t3, t4;

  // Stage 1

  for (int i=0; i<8; i++)
      F[i] = S[i]/(2.0*Math.cos(i*ang/2));

  F[0] = F[0]*2/Math.sqrt(2.0);

  t1 = F[5] - F[3];
  t2 = F[1] + F[7];
  t3 = F[1] - F[7];
  t4 = F[5] + F[3];
  F[5] = t1;
  F[1] = t2;
  F[7] = t3;
  F[3] = t4;

  //printF();

  // Stage 2

  t1 = F[2] - F[6];
  t2 = F[2] + F[6];
  F[2] = t1;
  F[6] = t2;

  t1 = F[1] - F[3];
  t2 = F[1] + F[3];
  F[1] = t1;
  F[3] = t2;

  //printF();

  // Stage 3

  F[2] = a1*F[2];

  t1 = -a5*(F[5] + F[7]);
  F[5] = -a2*F[5] + t1;
  F[1] = a3*F[1];
  F[7] = a4*F[7] + t1;

  //printF();

  // Stage 4

  t1 = F[0] + F[4];
  t2 = F[0] - F[4];
  F[0] = t1;
  F[4] = t2;

  F[6] = F[2] + F[6];

  //printF();

  // Stage 5

  t1 = F[0] + F[6];
  t2 = F[2] + F[4];
  t3 = F[4] - F[2];
  t4 = F[0] - F[6];
  F[0] = t1;
  F[4] = t2;
  F[2] = t3;
  F[6] = t4;

  F[3] = F[3] + F[7];
  F[7] = F[7] + F[1];
  F[1] = F[1] - F[5];
  F[5] = -F[5];

  //printF();

  // Final stage
```

```
    f[0] = (F[0] + F[3]);
    f[1] = (F[4] + F[7]);
    f[2] = (F[2] + F[1]);
    f[3] = (F[6] + F[5]);

    f[4] = (F[6] - F[5]);
    f[5] = (F[2] - F[1]);
    f[6] = (F[4] - F[7]);
    f[7] = (F[0] - F[3]);
  }


    static public void main(String s[]) {

        idct2d test = new idct2d();
        int i,j;

        // Init to test transform in Wallace paper
        for (i=0; i<8; i++)
          for (j=0; j<8; j++) trans[i][j]=0;
        trans[0][0] = 240;
        trans[0][2] = -10;
        trans[1][0] = -24;
        trans[1][1] = -12;
        trans[2][0] = -14;
        trans[2][1] = -13;

        //First the row transforms
        for (i=0; i<8; i++) {
          for (j=0; j<8; j++) S[j] = trans[i][j];
          test.calcIdct();
          for (j=0; j<8; j++) trans[i][j] = f[j];
        }

        for (i=0; i<8; i++) {
          System.out.println();
          for (j=0; j<8; j++) System.out.print((int) trans[i][j]+" ");
        }
        System.out.println();

        System.out.println("Columns:");

        //Now the column transforms
        for (i=0; i<8; i++) {
          for (j=0; j<8; j++) S[j] = trans[j][i];
          test.calcIdct();
          for (j=0; j<8; j++) trans[j][i] = f[j]/4 + 128;
        }

        //Print it out!
        for (i=0; i<8; i++) {
          System.out.println();
          for (j=0; j<8; j++) System.out.print((int) (trans[i][j]+0.5)+" ");
        }
        System.out.println();
    }
}
.......
....
..
.
```


::

Part II: Bringing "True Color" images to the 64
----

by Adrian Gonzalez <adrianglz@globalpc.net>

The Commodore 64 has a somewhat limited resolution, 16 predefined colors, and some very peculiar restrictions as to the number of different colors that can be placed next to each other. These restrictions make drawing colorful pictures on the 64 a difficult task, and displaying full color photographic images almost impossible.

I've been fascinated with bringing full color images to the c64 for a long time now. My first image conversion project was a C program that could convert 16 color IFF pictures to koalapaint format. I started work on this project somewhere back in 1992 or so. It ran on the Amiga, and it was one of my first 'serious' C projects, so I was basically refining my C skills while doing it. After some time I rewrote the converter completely and

added support for Doodle, charsets and a few other things.

Shortly after and with the help of a few friends on the net, I learned about a "magical" graphic mode called FLI. Before I could do a FLI converter, however, somebody on irc #c-64 pointed me to a couple of 'amazing' images available on an ftp site that were supposedly in some new, colorful vic mode. I was reluctant because I thought I had seen the best graphics a c64 could do. Boy was I wrong. I was absolutely amazed by this 'new' VIC mode called IFLI. Shortly thereafter the thought of doing an IFLI converter grew stronger and stronger in my head and the idea of a FLI converter practically vanished. After several weeks of hard work I came up with my first attempt at IFLI conversion. Several years passed until there was a reason to port this converter/renderer to the c64. The reason, of course, was Steve Judd's JPEG decoder.

My involvement with the JPEG project kind of started before Steve even started to work on it. About two years ago, Nate Dannenberg asked me to do a renderer for his QuickCam interface. I first came up with a 160x100 renderer in 4 grays. After that I came up with the 2 gray 320x200 hires renderer that was used first for Nate's Quick cam, and later modified to work with the first version of Steve's JPEG decoder. This same renderer was later hacked into rendering drazlace grayscale images.

The big challenge, of course, was porting the full color IFLI renderer to the c64. I don't think I would've ever bothered if it wasn't for jpx. We faced the obvious restriction of the c64's limited RAM (The IFLI image itself takes up half the c64's memory!). Things were tight, but it the end, it worked out just fine. But how exactly does the renderer do it's magic? What's all that garbage on the screen while it's rendering? Well, I'd like to start off by giving a quick explanation of what dithering is, and how the renderer uses a particular kind called Floyd-Steinberg dithering.

Floyd-Steinberg Dithering

Dithering is the process of using patterns of two or more colors to trick the eye into seing a different color. Let's say that you want to display 3 shades of gray with just two colors, you could have dither patterns such as:

```
. . . .   *  .  *  .   *  *  *  *
. . . .   .  *  .  *   *  *  *  *
. . . .   *  .  *  .   *  *  *  *
. . . .   .  *  .  *   *  *  *  *
. . . .   *  .  *  .   *  *  *  *
. . . .   .  *  .  *   *  *  *  *
```

Where the dots (.) are black pixels and the asterisks (*) are white pixels. If the pixels are small enough, the eye will see the middle pattern as a shade of gray. This is the basic concept behind dithering.

Floyd-Steinberg dithering is an 'error diffusion' dither algorithm. Basically that means that when drawing an image, if a color in the source image can't be matched with the available colors we have to use the closest available color. After that we have to figure out the difference between the color we wanted to use (source image color) and the closest one we had available. That difference, or error, has to be distributed (diffused) amongst adjacent pixels.

For example, imagine we have a video chip that can only display black and white pixels. Black pixels would be 0% brightness and white pixels 100% brightness. Let's say we want to use this chip to display an image with 100 shades of gray. We can store the image as an array of numbers from 0 to 99, where 0 represents 0% brightness and 99 represents 100% brightness. A small part of our test image could look something like this (5 x 2 pixel chunk of the image):

```
00  25  45  75  99
30  50  80  30  10
```

Without dithering, the best we could do is pick the color closest to the one we want to display, so we'd end up with something like:

```
00  00  00  99  99
00  99  99  00  00
```

Where 00 is black and 99 is white. Basically, any pixels with brightness greater or equal to 50 were converted to white (99) and the rest were converted to black (00), since those are the only two colors our hypotetical video chip can display.

With Floyd-Steinberg error diffusion dithering we also plot the closest color we have, but instead of just moving on to the next pixel we calculate by how much we were off (error) and diffuse that amount among adjacent pixels. Going back to our test image, the first pixel is completely black so we can display it right away without incurring any error, because we matched the color exactly. The second pixel (25) is dark gray so we plot it with the closest color we can, in this case, black (00). We then proceed to compute the error, which is equal to the color we wanted (25) minus the color we had available (00), so for this pixel, the error is +25. We then diffuse the error (+25) to the adjacent pixels. F-S dithering uses the following distribution:

```
       C.Pix  7E/16

1E/16  5E/16  3E/16
```

Where C.Pix is the current pixel, and E is the error. Basically that means, add seven sixteenths of the error to the pixel to the right of the current pixel, five sixteenths of the error to the pixel below the current pixel, etc.

So in our example, we wanted to plot a dark gray pixel (25) but we only had black available (00), so the error is +25. So then we have (rounded off)

```
(7/16)E = 11
(5/16)E = 8
(3/16)E = 5
(1/16)E = 2
```

When we add this to the original image buffer, we get:

```
(Original)
00  CP >45< 75 100
25  50  80  30  10

(Diffused)
00  CP >56< 75 100
27  58  85  30  10
```

Again, CP stands for 'Current pixel'. After doing these calculations, we're ready to move on to the next pixel. You'll notice that the third pixel (originally 45) would have been plotted as black but now, because of the error diffusion, the new value is 56 so we'll plot it as white, and the error will be 56-99 = -43. We then repeat the procedure:

```
(7/16)E = -19
(5/16)E = -13
etc
```

And adjust the buffer accordingly. Repeat this procedure for each pixel, processing each scanline from left to right and scanlines from top to bottom and the result is a nice looking dithered image. Note that errors can be positive or negative, so we should prepare for a case such as this:

```
55 00 00
00 00 00
```

Get the 55, plot it as white, and we have an error of -44, so that means that our buffer needs to be able to handle negative values as well. After difusing, the buffer would look like:

```
 CP -20  00
-14  -8  00
```

Note also that the 1E/16 was discarded because we're at the left edge of the screen. The same overflow condition applies to the opposite case:

```
 44  99  99
 99  99  99
```

The error +44 will make the values of adjacent pixels greater than 99, which is the maximum that can be displayed. The buffer needs to be able to hold values large enough to accomodate for this.

Now let's assume our hypothetical video chip manufacturer came up with a new video chip that can display 4 grays: black (0), dark gray (33), light gray (66), and white (99). If we want to plot an image with 100 shades of gray we will still always plot the closest color we can, i.e. 0-16 will be plotted as 0 (black), 17-49 as 33 (dark gray), etc. The error will be

positive or negative depending on whether we're under or over the color we wanted to plot. For example, the color 15 would be plotted as 0 (black), with an error of +15, while the color 20 would be plotted as 33 (dark gray) with an error of -13. And I think I've managed to confuse everybody including myself, but if you read this paragraph over, it should make at least some sense. Always remember the error is computed as the color we want minus the color we have.

As if things weren't fun enough, we can also apply this to a full color (RGB) display where we have 3 buffers, one for each primary color (red green and blue). Each buffer contains the corresponding levels of each primary color for a given pixel. Everything works exactly the same, except now colors are specified as triplets, for example:

```
   R   G   B
(  0,  0,  0) black
( 99,  0,  0) bright red
( 99, 99,  0) bright yellow
( 99, 99, 99) white
```

When we plot a color we now have to compute three errors, one for each primary color component. Each component is used to figure out the error for its corresponding buffer. For example, let's say we want to draw a red pixel (80, 0, 0) but our video chip can only display bright red (99, 20, 0). The error would still be computed as the color we want minus the color we can display:

We want:
r1=80, g1= 0, b1=0

We have:
r2=99, g2=20, b2=0

The error would be: (r1-r2, g1-g2, b1-b2) = (-19, -20, 0). After computing the error we proceed to distrubute it in the same fashion as before, except that we now have three image buffers, each with its own error to be distributed among its adjacent pixels. The best way to visualize this is to imagine you're displaying 3 independent images, each with it's own error. In the previous example, we would diffuse the -19 in the red buffer, the -20 in the green buffer and the 0 in the blue buffer.

With grayscale images, finding which shade of gray was the closest to the one we wanted to display was pretty straightforward. With full color images, the way to figure out the closest color changes a little bit. In order to find which of our available colors is the closest match for the color we want to display, we need to calculate the 'distance' from the color we want to each of the colors we have available and use the one with the shortest distance. To do this you can imagine the RGB color space as a cube, with the R, G, and B as each of the 3 axis. The origin (0,0,0) is black, and the corner opposite to the origin (99,99,99) is white, so figuring out the distance between two colors is as simple as figuring out the distance between two points in space:

```
color1 = (r1, g1, b1)
color2 = (r2, g2, b2)
d = sqrt( (r1-r2)^2 + (g1-g2)^2 + (b1-b2)^2 )
```

Let's say that our video chip can display 5 colors: black, red, green, blue and white. The RGB triplets for these colors would be:

```
( 0, 0, 0): Black
(99, 0, 0): Red
( 0,99, 0): Green
( 0, 0,99): Blue
(99,99,99): White
```

Let's also say we want to find out which of these is the closest match for the color (50,80,10). We have to compute the distance between this color and all of our 5 available colors and see which one is the closest. The calculations would be as follows:

Black:
sqrt((0-50)^2 + (0-80)^2 + (0-10)^2) = 94.87

Red:
sqrt((99-50)^2 + (0-80)^2 + (0-10)^2) = 94.35

Green:
sqrt((0-50)^2 + (99-80)^2 + (0-10)^2) = 54.42

Blue:

```
sqrt( ( 0-50)^2 + ( 0-80)^2 + (99-10)^2 ) = 129.70

White:
sqrt( (99-50)^2 + (99-80)^2 + (99-10)^2 ) = 103.36
```

In this case, the color with the shortest distance is Green (54.42). Note that we're not interested in knowing the exact distance, just knowing which color has the smallest distance, so it's safe to toss out the square root in order to things faster. If we don't calculate the square root we end up with the following squared distances:

```
Black:  9000
Red:    8901
Green:  2961
Blue:  16821
White: 10683
```

Of course, Green still has the smallest distance^2, and we're saved from performing a potentially troublesome (and slow) calculation.

Based on the previous explanation, we're ready to move on to implementing Floyd-Steinberg dithering on the C64. We will need to have the RGB values for each C64 color handy in order to be able to compute the error and the closest colors for each pixel we want to draw.

This article would probably end at this point if the C64 would let us choose any of the 16 colors for any pixel on the screen, but we're not quite that lucky.

Multicolor Bitmap Mode

The VIC-II video chip on the C64 has somewhat strict color limitations. In multicolor bitmap mode, the screen has a resolution of 160x200 and it's divided into 4x8 pixel 'cells'. Each of these cells can have up to 3 different colors out of the C64's 16 colors plus one background color common to all cells on the screen. If we wanted to display a 4x8 cell like this:

```
4  4  4  3
4  4  3  3
4  3  3  3
3  3  3  0
3  3  3  0
1  3  3  3
1  1  3  3
1  1  1  3
```

We could choose color 3 as the background color common to all cells, and the colors 0, 1 and 4 as the colors available to this particular cell (called foreground, multicolor 0, and multicolor 1). We can't display any additional colors on this cell. This makes multicolor bitmap mode a very tough choice for displaying true color images.

FLI Mode

Flexible Line Interpretation (FLI) mode is a software graphics mode in which the video chip is tricked by software in order to achieve higher color placement freedom. It is basically the same as multicolor bitmap mode, except that each 4x8 cell is further divided into eight 4x1 cells. Each 4x1 cell can have 2 completely independent colors, 1 color common to the entire 4x8 cell and one background color common to the entire image (some implementations of FLI change the background color on every scanline as well). One small downside of FLI mode is that the leftmost 3 columns of cells are lost due to the trickery used to get the video chip to fetch color data on every scanline. This means that the effective display area is reduced from 160x200 to 148x200.

IFLI Mode

IFLI mode or "Interlaced" FLI mode is basically two FLI images alternating rapidly. The C64 has a fixed vertical refresh rate of 60 frames per second for NTSC models and 50 frames per second for PAL models. This means that the screen is redrawn 60 times per second on NTSC units and 50 times per second on PAL units. IFLI alternates between two FLI images, displaying each for 1/60th of a second (1/50th for PAL), giving the illusion of a single blended image with more than 16 colors. One of the biggest

advantages of IFLI mode is that one of the FLI images is shifted one hires pixel (1/2 of a multicolor pixel) to the right to give a pseudo 320x200 hires effect.

For example, let's say a little part of the images looks like this:
(11 = one multicolor white pixel, 33 = one multicolor cyan pixel, etc)

```
Image1
11335577

Image2
 22446688
```

Alternating these two would give an effect that looks like:
12345678

Except that the colors would also mix and blur slightly, giving the illusion of more colors than the VIC-II can actually display. Of course, some color combinations work better than others. Don't expect to mix black and white and get a nice looking shade of gray (you'll get a very flickery shade of gray because of the alternation).

The renderer in jpz doesn't attempt to mix colors, mainly because I was never happy with the results I got by doing that. Instead, it treats the IFLI display as a 'true' 296x200 display capable of displaying any single one of the c64's 16 colors in any position. Note that the 3 column 'bug' also applies to IFLI, so the resolution is 296x200 instead of 320x200.

The color restrictions are somewhat more complex in IFLI mode. The renderer in jpz treats the display as if it was made up of 8x8 cells, with each cell divided into eight 8x1 cells, and each of those divided into two 4x1 cells (fun, huh?). To illustrate this better, look at the following 8x8 cell sample:

```
A I A I A I A I
B J B J B J B J
C K C K C K C K
D L D L D L D L
E M E M E M E M
F N F N F N F N
G O G O G O G O
H P H P H P H P
```

The odd columns belong to a 4x8 cell in the first FLI image and the even columns belong to a 4x8 cell in the second FLI image like this:

```
Image 1    Image 2
AAAA       IIII
BBBB       JJJJ
CCCC       KKKK
DDDD       LLLL
EEEE       MMMM
FFFF       NNNN
GGGG       OOOO
HHHH       PPPP
```

Remember the two images are offset by half a multicolor pixel to give the pseudo-hires effect. As for the color restrictions, each 4x1 cell on each image has 2 completely independent colors, but each 8x8 cell (the combination of the 4x8 cells from the two images) shares one color, and the entire image shares one background color.

The renderer in jpz is divided into two parts. The first part takes the source RGB image and remaps it to the c64's colors, using Floyd-Steinberg dithering as described in the first part of this article. This part outputs an array of numbers, each number corresponds to a c64 color. The second part of the renderer takes this array of c64 colors and displays it in IFLI mode as best as it can, taking into consideration the color placement limitations mentioned above.

The second part of the renderer works with blocks of 8x8 pixels and follows these steps:

1) Choose one color as common to the entire 8x8 cell
2) Choose two colors for each 4x1 cell
3) Render the 8x8 block (as two 4x8 cells, one on each FLI image)

In step one the renderer has to determine which one of the C64's 16 color would be the most helpful when chosen as common to the 8x8 block. This means that the common block color should be chosen to aid in 4x1 cells with more than 2 different colors (remember that 4x1 cells only have 2 completely

independent colors for them). If we wanted to display a 4x1 cell like

```
 1 15 12 12
```

We have two independent colors for the cell, which could be chosen as 1 and 15. We need either the common 8x8 block color or the background color to be 12 so we can correctly display this 4x1 cell. So how do we decide? We create a histogram!

A histogram is nothing more than a count of how many pixels of each color we have in a particular area (in this case an 8x8 block). Note that we only want to count the cases in which the common block color would actually be helpful for displaying a particular 4x1 cell. This is easier to explain with an example 8x8 block:

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
2 1 3 1 3 1 4 1
```

If we count all the colors in this block we would find 60 ones, one 2, two 3's, and one 4, and we would decide that 1 is the best choice as a common color for the 8x8 block because it's the most 'popular' color. A closer look reveals that this block will be rendered as the following 4x8 blocks:

```
Image1  Image2
1111    1111
1111    1111
1111    1111
1111    1111
1111    1111
1111    1111
1111    1111
2334    1111
```

Note that in the last 4x1 cell of image 1 we have 3 different colors. We have the ability to choose only two individual colors for this 4x1 cell, so if we choose 2 and 3, we won't be able to display 4 and our common 8x8 block color can't help us either. The best solution in this case is to _not_ count 4x1 cells with 2 or fewer different colors. This means that the only cell we would count in our histogram is the last 4x1 cell in image 1. So the new histogram would be one 2, two 3's, and one 4. We would proceed to choose 3 as the common 8x8 block color and this allows us to render the entire 8x8 block without a single problem!

In theory, the same should be done for the background color, in order to choose the best background color for the picture we're rendering, but that would mean that we have to do a histogram for the entire image before starting to render it. In practice, we don't have enough memory on the C64 to do this while reserving enough memory for an IFLI display (and decoding a JPEG), so we choose black as the default background color.

The second step in the process is to choose two colors for each 4x1 cell. This is done with the same histogram technique described earlier, except we have to take into consideration the color we picked as common to the entire 8x8 block so we don't repeat any colors and have the best chances of representing the original image as closely as possible. Basically, a histogram is made for each 4x1 cell, and the top two most popular colors are picked, assuming they're not the same as the background color (black) or the common 8x8 block color. For example, let's say the common 8x8 color is white (1) and we have a 4x1 cell that looks like this:

```
1223
```

The histogram would be: two pixels of color 2 (red), one pixel of color 1 (white) and one pixel of color 3 (cyan). In this case, since white is already our common 8x8 block color, we skip it and pick colors 2 and 3 as our 4x1 cell colors. The same skipping is done with black pixels because black is already available as the background color.

The third and last step is to render the actual image with the correct bitpairs. As you may know, multicolor images sacrifice half the horizontal resolution in favor of more colors. Basically, bits are paired up to have 4 possible combinations:

```
00: Background color (black in our case)
01: Upper nybble of screen memory   (4x1 cell color #1)
```

10: Lower nybble of screen memory   (4x1 cell color #2)
11: Video matrix color nybble (Common 8x8 block color)

All that's left to do is to output the corresponding bit pairs in each 4x1 cell to match the colors in the source (remapped) image as close as possible.

Depending on the complexity of the source image, there can be a few or a lot of 4x1 cells where we can't match all the colors. Remember we only have 2 completely independent colors for each 4x1 cell, and a cell can potentially have each pixel be a different color. When this happens, the best we can do is approximate the colors we can't match with the ones we have available. The renderer does this with a color closeness lookup table to avoid having to compute the color distances in realtime.

The table is basically a list of what colors are most similar to any particular c64 color, ordered from the most similar to the least. Let's say we want to plot the color white (1) but none of our bitpairs for the current cell can represent it. We have to look up white in our table and get the first color closest to it. If that color isn't available either, we will fetch the next closest color from the table and try again untill we find a match.

It is worth mentioning that due to the memory limitations of the C64 the bitmaps are stored in memory in 'packed' form while rendering. If you go back to the brief description of FLI mode, you'll remember that the leftmost 3 char columns were lost due to VIC chip limitations. When rendering, the bitmaps are stored contiguously in memory, without these 3 char block gaps in order to have enough room to render the entire image. After the entire image is rendered, it is 'unwound' by a small routine and then finally displayed in its full IFLI glory. In the stock version of the renderer you can see this 'unwinding' take place right before the image is displayed. Also, the colorful blocks on the screen while the image is being rendered are the actual buffers where the floyd-steinberg dithering is taking place (note that all of this is invisible in the SCPU version due to the memory mirroring optimizations provided by the hardware).

Well, that basically wraps up this article. I hope that it will give the reader an idea of the enormous amount of calculations that have to take place in order to be able to convert the images to a format suitable for viewing on our beloved C64. I also hope it explains the basic principles behind the rendering of these images, and why it takes so long for a stock system to display them.
....
....
..
.   - fin -