```
                  ########
             ##################
         ######             ######
      #####
    #####  ####  ####      ##      #####   ####  ####  ####  ####  ####
#####
  #####    ##    ##      ####    ##   ##   ##  ###     ##    ####  ##   ##
##
 #####    ########     ##  ##   ##        #####       ##    ## ## ##   ##
#####    ##    ##    ########  ##   ##   ##  ###     ##    ##  ####   ##
##
#####  ####  ####  ####  ####  #####   ####  ####  ####  ####  ####   ######
#####                                                                    ##
 ######            ######        Volume 1, Issue #4
   ##################             October 5, 1992
       ########

--------------------------------------------------------------------------------
-
```

Editor's Notes

```
by Craig Taylor (duck@pembvax1.pembroke.edu)

  My apologies about this issue being posted later than was mentioned in a
  preview post on comp.sys.cbm newsgroup. Due to some problems with coding,
  school and Murphy's law the issue had to be delayed until now.

  I have asked the system admin's at my site concerning a mail-server but
they
  said they did not have enough man-power (go figure) to get somebody to run
it.
  I will be implementing a mail-server system in my account in the near
  future for retrieval of programs and back-issues. I'll post descriptions
of
  how to use it it the next issue of C= Hacking as well as on the newsgroup
  comp.sys.cbm when I finish writing it.

  In this issue of C= Hacking we also start on an ambitious task: Developing
  a game for both the C128 and C64 modes that includes all of the features
  found in commercial games. Take a look in the Learning ML Column for more
  information.

  Also, The article concerning the 1351 mouse has _again_ been delayed due
  to time constraints. Rest assured that it will be in the next issue of
  C= Hacking.

  If you are interested in helping write for C= Hacking please feel free to
  mail duck@pembvax1.pembroke.edu (or duck@handy.pembroke.edu). We're always
```

```
  looking for new authors on almost any subject, software or hardware.

================================================================================

  Also note that this issue and prior ones are available via anonymous ftp
from
  ccosun.caltech.edu under pub/rknop/HACKING.MAG.

================================================================================

  NOTICE: Permission is granted to re-distribute this "net-magazine", in
  whole, freely for non-profit use. However, please contact individual
  authors for permission to publish or re-distribute articles seperately.

     *** AUTHORS LISTED BELOW RETAIN ALL RIGHTS TO THEIR ARTICLES ***

================================================================================
```

In this issue

```
Learning ML - Part 4

  In the next issue we'll embark on a project of making a space invaders
style
game for the C=64/128 from scratch using custom characters, interrupt-driven
music, animation, using the joystick, mouse or keyboard. The C64 and C128
versions will be developed con-currently, each program taking advantage of
the machine's capabilities. This is the first in a series - written by
Craig Taylor.

The Demo Corner: FLI - more color to the screen

All of us have heard complaints about the color constraints on C64.
FLI picture can have all of the 16 colors in one character position.
What then is this FLI and how it is done ? Written by Pasi 'Albert' Ojala.

RS-232 Converter

  This article details plan for a User port TO RS232 connector using just
ONE
IC and 4 capacitors. The circuit is included, and suggestions on alternative
chips and parts are examined.  Written by Warren Tustin

Introduction to the VIC-II
```

```
  This article examines the VIC-II chip in detail and provides an
explanation
of the various registers associated with the chip. Written by Pasi 'Albert'
Ojala.

LITTLE RED READER: MS-DOS file reader for the 128 and 1571/81 drives.

This article presents a program that reads MS-DOS files and the root
directory
of MS-DOS disks.  This program copies files from disk to disk so two disk
drives are required to use it (or a "virtual" drive).  This scheme imposes
no
limit on the maximum size of a file to be transferred.  The user-interface
code is written in BASIC and presents a full-screen file selection menu.
The
grunt-work code is written in assembly language and operates at maximum
velocity.  Complete, explained code listings are included.  By Craig Bruce.

================================================================================
=
```

Learning Machine Language - Part 4

```
by Craig Taylor (duck@pembvax1.pembroke.edu)

                     +---------------------------+
                     | Space Invasion - Part 1   |
                     |                           |
                     | Programming: Craig Taylor |
                     | Graphics   : Pasi Ojala   |
                     | Music/Sound:              |
                     |                           |
                     +---------------------------+

I. Introduction
   ------------

  In this and future Learning Machine Language's we will develop a game
called
  Space Invasion. The game will be similair to Space Indvaders and will run
on
  the Commodore 64 or the Commodore 128 in 80 columns. It will feature all
the
  "features" and "parts" that are found in commercial games with interrupt-
  driven music, custom character definitions, 100% machine language, multi-
level
  game play, and input from the keyboard, joystick or mouse.
```

```
  Note | I am looking for someone to help aid music composition that will
  -----+ be introduced in a later issue. Programming of the 6502 is helpful
  but not a requirement. Please email me at duck@pembvax1.pembroke.edu if
  you are interested.

  Many thanks to Pasi Ojala for his work with the graphics in this program.

  Also please note: This entire program has been assembled sucessfully with
  the Buddy-128 assembler for both the C=128 and C=64 version. Due to the
  length of the source files (over 1,500 lines) I'm not sure if Buddy-64
will
  handle it. Thus if you get errors during assembly, all I can say is:
sorry.
  If this is the case then the next issue will handle dividing the program
and
  data up into segments which can then each be loaded seperatly.


II. Machine Notes
    -------------

  The Commodore 64 and 128 programs for Space Invasion will differ slightly,
  mostly in the following areas:

       - custom character definition
       - memory initialization / setup
       - sound / music

  Because the actual game play and the changes nesscesary between the areas
  listed above, we will use the Buddy Assembler notation for conditional
  assembly to allow the development of only one file containing the source
  code. In addition to conditional assembly most of the routines will be
  written as one with jumps to subroutines containing the C64 or 128 direct
  code as the algorithims are usually the same for each.

  In addation there will be several source files and some miscellaneous
include
  files for graphics and sound. For those of you who are or will be
converting
  the assembly source over to a different assembler the conditional
  assembly directives .if (condition) will only be true if the condition is
  non-zero. Ie: if the symbol computer is defined as 128 then the following
  example illustrates it:

       computer = 128
       .if computer-64         ; non-zero answer so therefore
       ; 128 code goes here
       .else
       ; 64 code goes here
       .ife                    ; end the .if condition.
```

```
  Also note that for much of the program we will _not_ be using the computer
  routines and instead be developing our own.

  In addition the program will show you how to use IRQ interrupts to
simplify
  programming. We will be using them to play music in the background on
three
  voices (sound effects will temporarily pre-empt the third voice from
playing).
  Also animation of characters will be done via the IRQ. A little background
  on interrupts for those of you who are a bit hazy on what they are or have
  never seen them before (Also try taking a look at Rasters: What they are
and
  how to use them in C= Hacking #3 - While this does not necessarily cover
  what we are going to be using interrupts for it does describe them quite
  well.) Basically the computer generates an interrupt every 1/60th a second
  from a timer on the computer (usually from the CIA chip or the screen for
  those of you who are curious). The computer will save all the registers,
jump
  to a subroutine - perform the instructions there (usually updating time,
  scanning the keyboard etc...) and then recall all the registers and return
  to the user program. This is an interrupt. An IRQ interrupt describes an
  interrupt that we can allow to be "turned on" and "turned off" - ie: we
  can temporarily disable it if we have to. A NMI interrupt describes an
  interrupt which we can _not_ temporarily disable -- we will not be using
  NMI interrupts in this program.

III. The Process
     -----------

  Part of what this series of articles is focused at is the development of
being
  able to analyze programming tasks and break them down into smaller
workable
  problems. Once these problems or subroutines are completed your original
  problem is solved.

  Let's take this approach to Space Invasion:

  Problem Statement: Build a Space Invader program called Space Invasion.
  -----------------

  Usually, given a problem you have to re-work the problem statement to
  encompass all of what you want. Let's try again:

  Problem Statement: Develop a Space Invader program called Space Invasion
  -----------------  utilizing the 64 or 128 screen with interrupt driven
                     music / sound, and allowing input from the keyboard,
                     joystick or mouse.

  Hmmm... The problem statement listed above is better but it has no real
```

```
order;
  we have no clear idea of where to start and what we need to do. It does
  however tell us that we have the following sections:

       - 64 / 128 Screen Handling
       - Music / Sound
       - Input Handling
       - Game Driver (implied)

  Let's think a bit more about each of these sections and what each will
  involve:

  128 / 64 Screen Handling:      - Putting characters on screen.
  -------------------------      - Initializing the Screen / Registers.
                                 - Setting up the Custom Characters.
                                 - Handling any Animation.

  Music / Sound:                 - Setting up the Sound Chip Registers.
  -------------                  - Playing a note read from Memory.
                                 - Executing a Sound Effect.

  Input Handling:                - Device Selection (keyboard, mouse,
joystick).
  --------------                 - Keyboard Scanning.
                                 - Mouse Scanning.
                                 - Joystick Scanning.

  Game Driver:                   - Title Screen.
  -----------                    - Initialization of Memory.
                                 - Level Setup.
                                 - Movement of Aliens.
                                 - Movement of Missles.
                                 - Movement of Player.
                                 - Collision Checking.
                                 - Collision Handling.
                                 - End of Level.
                                 - Score Updating.
                                 - End - Game handling.
                                 - High Score Update.

  Shrew! Long list 'eh? - Now you may have thought of some not listed above,
  and we may have possibly overlooked some crucial routines -- that's fine -
-
  the above is just intended as a building block - a place to start coding
from.

  If we think of these as subroutines we can build a skeleton outline of the
  program - yet we need some order in how we call them. Obviously we aren't
  going to move the player until we scan the input and that requires prior
  device selection etc...
```

```
  Hmm... Taking order into account we can re-state the problem as:

  Problem Statement: Develop a game similair to Space Invaders called Space
  ----------------- Invasion by initializing memory, the display device,
                    setting up Custom Characters, setting up the Music
                    Registers and displaying the title screen. From there,
                    select the input device and after that setup the
current
                    level. Next, while playing music in the background and
                    scanning the input device, move the aliens, missles and
                    player checking for collisions and taking appropriate
                    action as required (player dies, score increases etc or
                    what-not). After each level display if the player is
dead,
                    or set-up for the next level and repeat. When the game
has
                    ended update the high score if necessary.

  Try saying that five times real fast! :-) But that problem statement is a
  whole lot better than the one we had at the beginning which simply said to
  develop a game.


IV. Not All At One Time - What We're Doing This Time
    ------------------------------------------------

  Now this program is too complex, (as seen by the problem statement above)
to
  have in one article so this issue we'll concentrate on the basic main loop
  and the initialization of the Custom Characters and the title screen.

  Originally, I was planning on updating and listing the revised code in
each
  issue. However, due to space limitations and the enormity of the program
  currently (1,500+ lines!!) it will be placed for anonymous ftp at
  ccosun.caltech.edu under the directory: pub/rknop/HACKING.MAG.


V. The Main Loop
   -------------

  What is a main loop? Basically it's where everything gets done. It calls
other
  subroutines and keeps repeating until certain criteria are met - usually
when
  the player requests to exit the game. However, inside you'll find inner
loops
  for level play etc.

  Our main loop for this program will be:
```

```
;---------------------------------------------------------------------------
;; * Main Loop - This should be the last section in the source code.
;
; Main Loop
;

main'loop = *
               jsr memory'setup        ; Set-Up memory.
               jsr display'setup       ;  "  "  display.
               jsr char'setup          ;  "  "  custom character display.
               jsr music'setup         ;  "  "  music chip.
               jsr title'screen        ; Display the title screen.
               jsr select'input        ; Select Input Device.

level'loop = *
               jsr play'music          ; Start the music playing.
               jsr setup'level         ; Setup the current level.

             - jsr alien'move          ; Move aliens
               jsr missle'move         ;  "   missles
               jsr player'move         ;  "   player
               jsr check'collision     ; Check for collisions
               ldx collision'flag      ; Check collision flag.
               beq -

               dex                     ; Decrease .X by 1 so if X was 1 then
               beq player'die          ;    it's now 0 so we know player
died.
               dex                     ; Decrease .X again so if X was 2
then
               beq alien'die'sound     ;    it's now 0 so we make alien
death.
               jsr end'level           ; If we got here - than end of level.
               jsr wait'next           ; Wait for next keypress.
               jsr increase'level      ; increase level #.
               sec                     ; And go back....
               bcs level'loop

alien'die'sound = *
               jsr make'alien'sound    ; make alien sound.
               sec                     ; set carry
               bcs -                   ; and jump back.

player'die     jsr show'player'die     ; Show it on-screen.
               lda lives               ; Check # of lives.
               beq end'of'game         ; If 0 the end-of-game.
               bne level'loop          ; go back and re-start level.
               brk                     ; If we get here - than an error.

end'of'game    jsr end'game'screen     ; Show end-of-game screen.
```

```
               jsr high'score'update   ; Update the high score if need-be.
               jsr wait'next           ; Wait for next-game selection.
               lda quit
               beq +
               jsr setup'level'1       ; Set-Up first level.
               sec
               bcs level'loop          ; and start playing it.

             + jmp quit'game
;
; End of Main Loop
;
-------------------------------------------------------------------------
---

  Some of the routines listed above we will later replace with actual code.
It's
  much easier to see:

               inc level

  than to see a

               jsr increase'level

  and try to hunt down the code. I've included them in for now so that we
can
  have a better idea of what is going on.

  In the file: invasion.src most of the statements above are commented out.
  Once we write the routines we'll un-comment them. For now, this serves to
  still remind us of the routines we need to write.

  Also there are a couple of programming tricks that I used in the main loop
  that probably need some clarifying.

  When handling the collisions the .X register is loaded with the result of
the
  collision checking - $00 = no collisions, $01 = player died, $02 = alien
died,
  $03 = end of level. Anytime a load to a register is done the flags are
  automatically set as if you had compared it to 0 - hence we can ldx the
  collision flag and immediately branch if equal to zero for no collisions.
In
  addition to the load anytime the .X or .Y registers are incremented or
  decremented an implicit comparison to zero is performed. So if the .X
register
  is 1 previously, we decrement it then it will be zero and our BEQ
instruction
  will branch. If it's two then it will be one and we can continue like
this.
```

```
    [NOTE: Technically it's not a real comparison to zero but calling it a
    comparison to zero servers our purpose here. The only significant
difference
    would be in the effect of the carry flag which is insignificant in our
    code segment here.]

  Also in several locations are the two instructions:

               sec
               bcs [label]

  What these are doing are simply programming style - they could be
substituted
  with JMP [label] - however they offer advantages over JMP.  They take up
the
  a larger amount of execution time, however they are relocatable so any
mucking
  around / moving sections of code during debugging will be less likely to
  crash. Using other flags are also valid -- the use of which flag (I prefer
  the carry flag) is usually dependent on the programmer. Geos defines a
  similair macro called BRA (branch always) which is equivlent to:

               clv
               bvc [label]
  Note that the above is just programming style, held over from my
programming
  in assembly days. The use of JMP is probably preferable in terms of
  execution and also in being able to branch more than 127 bytes away (the
  branch instructions only have a range of +128/-127).


VI. Custom Characters
    -----------------

  Since we're writing for each of the seperate modes (64 mode, 128 mode) we
have
  to take a look at the differences between the VIC chip (64 mode) and the
8563
  chip in the 128.

  The Vic-Chip
  ------------
  The character sets in the VIC chip are defined as in the example below of
  the character code $00 "@" (all references are to screen "poke" codes -
not
  print codes).

         .byt #%00111100    Try holding the page (or moving away from the
         .byt #%01100110    screen) and taking a look at the patterns the
1's
         .byt #%01101110    and 0's make. Each character is thus defined as
```

```
         .byt #%01101110    eight bytes who's bit patterns define it.
Having a
         .byt #%01100000    total of 256 characters available makes it
         .byt #%01100010    neccesary to set aside a total of 2,048 bytes.
         .byt #%00111100
         .byt #%00000000

  Now, instead of designing all 256 character sets we'll just take advantage
of
  the fact that the letters and numbers we want will already be there --
we'll
  just copy them from the ROM set into RAM, modify some of the other
characters
  to reflect what we want and then tell the VIC chip to look at RAM to get
the
  character set definitions.

  There are some problems with copying the 'system' characters, however. The
  Commodore 64 usually masks out the character set and typically it is only
  available to the VIC chip so that more space can be present for user
programs
  and such.  It also takes up the section of memory that the I/O block in
  $d000-$dfff does so that switching it in while interrupts are enabled is
sure
  to result in a crash.

  We're also going to be doing a few things that you may not expect --
instead
  of copying all 256 characters - we're gonna _just_ copy the first 128.
This
  will give us all of the normal characters as the last 128 are the reverse-
  video counterparts to the first 128 characters. We're doing this to
conserve
  space and because we really don't need that many characters defined.

  Also location $01 contains what $d000-$dfff holds and we will have to
modify
  bit 2 to switch the character ROM in. Hence, the following program code is
  used to copy the character set:

------------------------------------------------------------------------
---
copy'chars = *                  ; must be run w/ interrupts disabled
             lda $01            ; register 1 = the control to switch in the
char.
                                ; rom.
             pha                ; save it as we'll later need to sta' it
back.
             and #%11111011     ; Bit 2 controls it - clear it to switch it
in.
             sta $01            ; and make it so we can read it in.
```

```
               lda #>$3000     ; move chars to $3000
               sta dest+1
               lda #>$d800     ; from $d800 (start of char set) (lower-case)
               sta src+1
               ldy #$00        ; lo-bytes of both src, dest = $00.
               sty src
               sty dest
               ldx #$10        ; copy 2k of data.
             - lda (src),y     ; copy byte.
               sta (dest),y
               iny
               bne -           ; continue until .Y = 0.
               inc src+1       ; increase source & dest by 256
               inc dest+1
               dex             ; decrease .X count.
               bne -           ; if non-zero then continue copying, else
               pla             ; restore value of $01
               sta $01         ; and put back.
               lda $d018       ; set VIC-chip address.
               and #$f1        ; to show char set.
               ora #$0c
               sta $d018       ; and finally tell VIC where the char set
is...
               rts             ; and return.
-------------------------------------------------------------------------
---
  Note that we still need to change the actual characters we're gonna be
using.
  That will be handled in the section after next: Changing the Characters as
  there is a great deal of similarity between the 128 and 64
implementations.

  The 8563 Chip
  -------------
  The 8563 80-Column chip usually has 16k or 64k Ram attatched to the chip
  which the CPU does not have direct control over. It has to direct the 8563
  to store and retrieve values to that memory. What makes control over that
  memory all the more difficult is the fact that the 8563 only has two lines
  or addresses that the CPU can control.

  The 8563 has a character set in much the same way the VIC chip does, save
  one exception - each character set can have up to 16 lines. Normally, the
last
  eight lines are filled with $00 and are not shown. (Provisions can be made
to
  have 8x16 characters but it is not needed for this game and thus, will not
be
  shown - For more information See C= Hacking Issue #2: 8563: An In-Depth
Look.)
  Thus the algorithim is similair to the C=64 but 8 zero-bytes will need to
be
```

```
  written at the end of every eight bytes read.

  However, the 8563 does make things easier for us! - When the computer is
first
  turned on a copy of the Character Set from ROM is copied into the 8563.
The
  8563 has no ROM Character Set associated with it and thus we are able to
just
  simply modify the character set that is in the 8563 memory instead of
copying
  it over.  Because of this no routine will be presented to copy a character
  set into the 8563 memory, rather the discussion of copying individually
  defined characters will take place in the next section. The C=128 also
makes
  life even easier for us at the end when we will exit the program,
  modifying the character set back to the "standard" Commodore character set
  by a routine in the KERNAL that will copy the characters back. We'll take
a
  look at it closer when we write the exit routine.

  Also note that since the 8563 chip supports the 80 column screen we will
  be defining two characters that can be placed side by side for each alien
  so that the playing field will be similair to the C64 version. However,
for
  the title screen we will be switching the 8563 into a "40 column" mode
  to make programming easier, in addition to expanding the character bit-
mapped
  logo.
  Changing the Characters
  -----------------------
  A lot of the times you'll find yourself re-using subroutines and code that
  you have previously created, gradually, over a period of time building up
  a library of routines. When thinking through the purpose and intent of
this
  routine I thought about possibly building it so it would read a table and
  change the character set based on that table. The 64/128 character sets
  would be the same - this routine would automatically generate the eight
  additional bytes needed by the 8563 if need-be and it would call the
  appropriate storage routine - store to either the 8563 or the computer
  memory.

  Now you may be asking why would you want to store to the computer memory
  in 128 mode? Why not just have two seperate versions? - Yes - that could
  be possible but I'm implementing it this way because in the future I may
  see a need to define custom characters in 128 mode for the 40 column
screen.
  This way I can just extract the routine, pop it into my program and I've
got
  that section of the code complete.

  This is what I was thinking of for the data table:
```

```
      .byt 1 = 8563, 0 = comp. memory.
      .word address ; address base of char-set in computer or 8563 memory.
      .byte char #  ; (to start)
      .byte # of chars to define
      .byte # of characters to define
      .byte data,data,....,data8 ; character data.
      .byte data,data,....,data8 ; character data. etc....
      . . .
  Entrance into the routine will consist of .AY holding the location of the
  table.  We will keep the address of the table and keep incrementing it as
  we go along in z-page locations.

------------------------------------------------------------------------
---
       install'char = *
               sta zp1                ; save .ay in table address
               sty zp1+1
               ldy #$00               ; read computer mode.
               jsr get'byte
               sta mode
               jsr get'byte           ; get address base.
               sta adr
               jsr get'byte
               sta adr+1
               jsr get'byte           ; get number of characters to copy.
               sta numb
               jsr get'byte           ; get next character #.
               sta wrk                ; save in temp. location.
               lda #$00
               sta wrk+1
               asl wrk                ; shift left x3 times = *8
               rol wrk+1
               asl wrk
               rol wrk+1
               asl wrk
               rol wrk+1
               lda mode               ; if for 8563 then multiply 1 more
time.
               beq +
               asl wrk
               rol wrk+1
             + lda adr                ; add character address in.
               clc
               adc wrk
               sta wrk
               lda adr+1
               adc wrk+1
               sta wrk+1              ; address now calculated
               jsr setadrs            ; set address in proper chip
  loop'install ldx #$08               ; copy 8 bytes.
             - jsr get'byte
```

```
               jsr writebyte           ; write out byte.
               dex
               bne -
               lda mode                ; if 128 then fill out 8 more $00
bytes.
               beq +
               lda #$00
               ldx #$08
             - jsr writebyte
               dex
               bne -
             + dec numb
               bne loop'install
               rts
-------------------------------------------------------------------------
---
```

What? We have thre subroutines : writebyte, setadrs, and get'byte that we haven't examined yet. These are going to be the routines that are dependant on the computer type. Also, writebyte will require that .XY not be disturbed; setadrs requires that .Y not be disturbed hence the following:

```
-------------------------------------------------------------------------
---
       setadrs tya             ; save .yx
               pha
               txa
               pha
               lda mode        ; check computer type.
               beq +           ; if C=64, then jump ahead.
               ldx #18         ; VDC register - current memory address hi
               lda wrk+1       ; get address hi
               jsr wr'vdc
               ldx #19         ; VDC register - current memory address lo
               lda wrk         ; get address lo
               jsr wr'vdc
             + pla             ; restore .XY
               tax
               pla
               tay
               rts             ; and return.
-------------------------------------------------------------------------
---
```

Note that we really don't need a setadrs for the C=64 -- we can just index off (wrk) in the writebyte routine which follows:

```
-------------------------------------------------------------------------
---
  writebyte    sta temp        ; save as we need it later.
```

```
               txa             ; Save .XY
               pha
               tya
               pha
               lda mode        ; now check computer type.
               beq +           ; if c64 jump ahead
               lda temp        ; recall temp.
               jsr wr'vram
               sec
               bcs ++          ; jump ahead
             + ldy #$00        ; C64 / y-index = $00
               lda temp        ; get value
               sta (wrk),y     ; store
               inc wrk         ; now increase address
               bne +
               inc wrk+1
             + pla             ; now return after recalling .XY
               tay
               pla
               tax
               rts             ; and return.
-------------------------------------------------------------------------
---

  Note that the following routine is fairly short but it is called numerous
  times within the routines that use data tables such as install'char,
  write'txt and write'col.

-------------------------------------------------------------------------
---
  get'byte = *
               lda (zp1),y
               iny
               bne +           ; if zero then increase zp1 hi
               inc zp1+1
             + rts
-------------------------------------------------------------------------
---

  Not bad 'eh?  A quick note: The instructions: PLA, TAY, PLA, TAX, PHA,
etc..
  are routines that Push or Pull (pha,pla) the .A onto the stack. The TAY,
TAX,
  TXA, TYA are instructions that transfer a register to another (ie: the TAY
  transfers the A register to .Y, TXA transfers .X to .A etc...) By using
the
  combination of these with the stack we can save the registers and later
  re-call them so that they are the same when we entered the routine. The
  stack is usually a "mystery" item to new programmers of the 6502 series.
  Basically it's just like any other stacks in the real world - the last
item
```

```
  thrown (I'm non-practicing perfectionist so I throw stuff.. ;-) ) or
pushed
  on the stack will the first item removed or pulled from the stack. For
  example I've got a stack of books sitting near me :

              Mapping the Commodore 128
              128 Internals

  and I'm holding Mapping the Commodore 64 in my hands. If I push (or toss)
  the book onto the stack (and hopefully hit the stack instead of the floor)
  I'll have the following stack:

              Mapping the Commodore 64
              Mapping the Commodore 128
              128 Internals

  and it should be easy to see that if I "pull" the next book off the stack
  that I'll get the Mapping the Commodore 64 book. The next book to be
"pull"ed
  after that would be the Mapping the Commodore 128 book. This idea can be
  applied to the 6502 stack -- It will keep storing values (up to 256) when
you
  "push" them on (via the PHA instruction) and will retrieve the last value
  stored when you "pull" them off (via the PLA instruction). Another PLA
  instruction would return the next value that had been stored.

  The Character Bitmaps
  ---------------------

  Pasi Ojala is to be credited with all the graphics and many thanks go out
  to him.

  The game logo is made up of 120 custom defined characters that will be
  printed in the following manner (on the 128 screen they will be centered).

  (in reverse video)...

           ABCDEFGH . . . [up to 40 characters]
           IJKLMNOP
           QRSTUVWX

  and everything will line up.

  So that it will look like a "mini-bitmap".  We could have used bitmap mode
  and made a very nice looking title screen but that would have involved
  switching and allocating memory for the bitmap, etc . . . On both the
  8563 and the VIC that involves a bit more work and so custom characters
  will be used for the title screen. The regular letter and numeric
characters
  will be available so that we can display credits and game instructions
  below the logo.
```

```
  Now - in the program listing we could list them as binary #'s and that
would
  make editing them very easy but we're gonna use their decimal
representation
  in the program listing.

  The characters are defined similair to the logo except they are treated as
  single characters. In the 128 version due to the 80 column screen we are
  going to use two characters side by side to simulate one alien so that the
  playing field will be similair to the C64 version. In addation, during the
  main loop we will modify the character sets to support animation of the
  aliens. In the data listing there is a reference to "frames" - for each of
  the aliens there are 8 differant frames.

  Oh! - There will be more characters defined in the future. Right now I'm
  mainly interested in getting some base characters down so you can see how
  custom characters are implemented.  When we start setting up different
levels
  and such we'll add more characters then. Currently the custom characters
  are not used - only the characters for the logo. For those of you who are
  curious try installing the characters via install'char and taking a look
  at the aliens.


VII. Title Screen
     ------------

  The title screen is usually a lead-in to the actual game and it's aim is
  to tell the player how to play the game, any available options and p'haps
  present a nice graphic or two to "wow" the user into playing. In addation,
  the main musical theme can be introduced here to unify the game-playing.
  The discussion below does not take into account color but rest-assured we
  will be using varying colors in the title screen. The format for the color
  data will be almost identical to the title screen format except it will
  be structured via the following:

       .word address
       .byte num_of_chars to put color ($00= end of data)
       .byte color_value

  The routine (color'text) can be found in the source listings at the end of
  this article. Because of the similarity between it and write'text it is
  not discussed in this article.

  Title Screen BackGround
  -----------------------
  The title screen I envisoned as a bordered screen (using the normal C=
  character set - ie: C= A,S,Z,X on the keyboard) with our bitmap in the
middle
  and under-neath it a short description of the game and game-play
instructions.
```

```
  Now this is my idea of the screen layout (rough drawing as we're not using
  the actual screen dimensions):

     +-------------------------------------------------------------+
     | -LOGO ------------------------------------------------------ |
     | --------------x 3 lines------------------------------------ |
     | ------------------------------------------------------------ |
     |                                                             |
     |                   Space Invasion C64/128                    |
     |                     Programming : Craig Taylor              |
     |                     Graphics    : Pasi Ojala                |
     |                     Sound       : ???????????               |
     |                                                             |
     | ------------------------------------------------------------ |
     | To Play:                                                    |
     |    Use joystick in port 2, mouse in port 1 or keyboard:     |
     |             A - Left, Z - Right   Space - Fire              |
     |               F1 - Restart                                  |
     |                                                             |
     +-------------------------------------------------------------+

  Title Screen Formatting
  -----------------------
  We come into a problem here -- the screen is some 1000 characters on the
C64,
  and 2000 characters for the C=128.  It would be extremely wasteful to
store
  that many characters in memory just to reproduce a title screen - and most
of
  them consisting of spaces at that!!

  What we'll do is to just specify the address on screen, the # of
characters
  and then list the characters. It will be similair to our custom character
  table driver above but will be different enough that a new routine is
  warrented. We will however use the two subroutines writebyte and setadrs
  that were developed in the previous routine. The data will look like the
  following:

       .word address
       .byte num_of_chars ($00= end of data)
       .ascii "text"
       .byte address .... etc....

  and we'll enter with .AY containing the address of the table.

  So basically we come up with the following:

------------------------------------------------------------------------
---
       write'txt = *
```

```
               sta zp1         ; save .ay in table address
               sty zp1+1
               ldy #$00
   loop'w'text = *
               jsr get'byte   ; set address.
               sta wrk
               jsr get'byte
               sta wrk+1
               jsr get'byte    ; get # of chars to write out.
               cmp #$00
               beq +           ; if zero then exit.
               tax
               jsr setadrs     ; set address to wrk,wrk+1
             - jsr get'byte
               jsr writebyte   ; write out byte.
               dex
               bne -
               sec
               bcs loop'w'text ; this is an absolute jump to loop
             + rts             ; return.

------------------------------------------------------------------------
---

  This is similair to our previous routine, and was in fact copied and
modified
  from the previous routine.


VIII. Debugging
      ---------

  Now, not all programs are perfect, and during the development of this
  portion of the game there were several errors found. Tracing an error in
  Machine/Assembly-Language is like trying to find a grammatical error in a
  language you don't know. ;-) But seriously, there are several ways to
track
  down errors in your code.

  1 - Try tracing it through by hand playing "What if I were the computer"
and
      following what each register does.

  2 - Are you switching the LoHi order of variables? Ie: is it lda #< or
      lda #>??

  3 - Set BRK points and run the program / subroutine within a machine
language
      monitor and make sure the registers / memory locations contain the
values
      that they should. If not, find out why.
```

```
  4 - Try to simplify your code in terms of programming ease - Make the
      assembler do the work for you - it's a lot less likely to make errors
      than you are.
  5 - Think logically!!!
  6 - Change something at random and pray.

  I can't stress numbers 3 and 5 enough. During the writing of the
install'char
  routine there were numerous bugs that were eventually tracked down by
  setting a BRK instruction further along in the code and seeing exactly
what
  the register / memory locations were. Also the use of temporary load and
  store instructions into "safe" regions of memory helped me monitor what
some
  of the values were.

  For example, at one point I had a section of code similair to the
following:

            clc
            lda value
            adc data
            bne +
            inc data+1
          + [.... ]

  And it's purpose was to add value to data. Now I've found simple errors
are
  usually found last, after complex errors. And not until a set a break
point
  like:

            clc
            lda value
            adc data    <-----Missing Instruction after here-------------+
            bne +                                                        |
            inc data+1                                                   |
          + BRK                                                          |
            [.... ]                                                      |
                                                                         |
  did I actually figure out that I was missing the STA DATA instruction --+

  So, when writing, modifying, and trying to debug code try to take your
time
  and isolate every possible problem. Also don't be afraid to stop the code
  mid-stream as in the above with use of the BRK. You can always remove it
  (and probahly should) in the final code and it serves as a very valuable
  debugging tool with the aid of a machine-language monitor.

IX. Memory Map Considerations
    -------------------------
```

```
  Before you start a program it's a good idea to consider where in memory
you
  will have everything. Now we've already started some of the program above
  and just blindly picked numbers at random it seemed like $3000 for the
  character set for the C=64 etc... We didn't - I'm introducing the Memory
  Map Considerations here to show the example of what if we didn't think
  about how memory was going to be organized.

  The C=64 only has 64k of memory of which typically the range $0800-$a000
  is available and $c000-$cfff is also.  If we had blindly picked numbers
  all over the place to store our code then we would have a disorganized
  program that would most likely accidentally use one subroutines storage
  as temporary data for another. It's like shooting randomly in Laser Tag
  not checking to see if there is a target there or not first... The end
  result: Chaos.

  Currently we're not following the rule for "temporary variables" but as
  we gradually fade out of the normal C-64/128 default mode and write our
  own routines / interrupt handlers we'll switch things over. Also, on the
  C=128 instead of using Bank 0 with the I/O block enabled we're currently
  using the BANK 15 configuration as the program doesn't extend past $4000
  yet ($0000-$4000 is common memory in the normal C=128 configuartion).

  64 Considerations
  -----------------
  The 64 will have free memory in the following areas: $0800-$a000, and
  $c000-$cfff. However, if we disable the Basic Rom we can have the whole
  area from $0800-$cfff free for our program. Because we don't need the
  Basic Rom we will do just that (in the listing now we currently won't but
  it will be done in a future issue). Therefore having the character set
  at $3000-$5000, the music data at $5000-$8000, the program will have the
  area free from $8000-$cfff. $0800-$3000 will be available if needed for
  routines who need temporary storage.

  Temporary Storage is going to be defined as follows. Each routine that
needs
  temporary storage will be assigned a "level" number. The lower levels will
  be assigned level 1 on up to level 3. The range $0800-$3000 will be
  broken down into the following sub-ranges.

       Level 1: $0800-$1000
       Level 2: $1000-$1800
       Level 3: $1800-$3000
  This way when writing the sub-routines we can be assured that a section of
  memory is not overwritten by a subroutine we call. When we actually start
  programming we'll decide where in each sub-range the routine will have
  access to.

  128 Considerations
  ------------------
  The 128 has two "banks" of 64k each. Normally for large programs we would
```

```
  think about using both banks - (from the idea: Hey! - We got it, why not
  flaunt it?) but we won't be using both banks.

  Free memory on the C128 typically consists of the range $0400-$09ff
  (where we'll be overwriting the 40 column screen (which we're gonna blank
  anyway) and the Basic run-time stack.) Also the area from $0b00-$0fff
  is free (overwriting the tape area, the rs-232 buffers,l and the sprite
  definition area). Also $1300-$cfff will be free.

  Now, the C=128 has different memory maps it can configure itself to -
  Bank 15 is the standard mode under most basic programs and allows the
  programmer to directly "sys" to calls. The MMU (memory management unit -
  the chip that does everything) sees memory in a slightly differant way
  than from basic. We'll cover it in more detail when we examine the
mem_init
  routine. For now, we're just gonna set up in the program and not in the
  coding segments. The explanation of what we're doing will be "revealed"
  in a future issue.

  We will use Bank 0 of memory and from $1300+ will be the program. The
ranges
  of $0400-$09ff and $0b00-$0fff will be used in a similair mannar as the
C64
  ranges were for Temporary Storage. We will also have the I/O section from
  $d000-$dfff swapped in. This is not a standard "basic BANK #" but when we
  cover the init'memory routine we'll see how we can do this. Music data
  will be from $a000-$d000.

X.  Looking Forward / Back
    ----------------------

  Hopefully through the listing and the discussion of the routines you have
  started to understand the basic concept of programming: breaking down
problems
  into smaller solvable steps. Try looking back over the code asking
yourself
  why that instruction is there. What would happen if you switched the
order?
  Is there an easier, better way to do the same thing? Why? Better yet, how?
  Examine the code, mess with it, muck it up so it doesn't work and then
figure
  out exactly why. The only way to learn is by experimentation. (BTW, muck
up
  a _copy_ of it - not the original ... *grins*)

  Take a look at the different sections of code and analyze them to see how
  they do what they do. Take a look at how the code was organized in terms
  of simplification. Trace through each subroutine so that you're able to
  know what the return values will be. In other words: Study, Study, Study!!
  I'm in school and so I know I just used the dreaded 'S' word but that's
  what you're going to have to do if you're interested in learning 65xx/85xx
```

```
  machine language. The only way to learn it (easily) is to study other
  people's code and try to understand why they did what they did.

  Next time we will take a look at the input routines for the mouse,
joystick
  and keyboard scanning. In addition we will also allow the player to move
  the ship around on the screen to test the input drivers.

  In addation, I am still looking for an individual to help with music and
  sound composition for this program. A knowledge of the SID chip and
  programming is helpful but not required. If you're willing to help then
  please email me at duck@pembvax1.pembroke.edu


XI. Listings
    --------

  Because of the enormity of the program listing (some 1,500+ lines) it will
  not be listed in this article but will instead be available via anonymous
  ftp at ccosun.caltech.edu under pub/rknop/HACKING.MAG as invas1.sfx.

  For those of you on the mailing list who would like to recieve it, a Mail-
  Server will be set up soon to handle requests and information will be
  sent to you concerning information about using it as soon as it's
completed.

  In the invasion1.sfx file there are the following files:

       invasion.src - the main file
       graphics.src - handles all graphics routines
       logo.dat     - custom character logo
       chars64.dat  - alien custom characters for C=64
       chars128.dat - alien custom characters for C=128
       titletxt.dat - text data for title screen
       titlecol.dat - color data for title screen
       invasion-128 - executable version of Space Invasion so far for C=128
       invasion-64  - executable version of Space Invasion so far for C=64

  Note: For the Commodore 128 it's recommended that you do a run/stop-
restore
  and then a "BANK15:SYS7168" to execute the program. For the Commodore 64
it's
  recommended the border be changed via: "POKE53280,0:POKE53281,0:SYS 32768"
to
  run the program.

================================================================================
==
```

The Demo Corner

FLI - more color to the screen

```
by Pasi 'Albert' Ojala (po87553@cs.tut.fi or albert@cc.tut.fi)
                        Written on 16-May-91   Translation 01-Jun-92
(All timings are in PAL, although the principles will apply to NTSC too)

All of us have heard complaints about the color constraints on C64. One 8x8
pixel character position may only carry four different colors.  FLI picture
can have all of the 16 colors in one char position.  What then is this FLI
and how it is done ?

In the normal multicolor mode can one character position (4x8 pixels) have
only four different colors and one of them is the common background color.
Color codes are stored in half bytes (nybbles) to the video matrix memory
(anywhere video matrix pointer points at, normally $0400) and to the color
memory ($D800-$DBFF). In multicolor mode the color of each pixel is
determined by two bits in the graphics memory. Bit pair 11 will refer to
color memory, background color is the color for bit pair 00, and video
matrix will define the colors for bit pairs 01 and 10.


_What happens in the VIC ?_

VIC (Video Interface Controller) fetches color information from memory on
each bad line. This will steal time from processor, because VIC needs to use
processor's bus cycles. Bad line is a curse in the C64 world. Fortunately
VIC's data bus is 12 bits wide and so the color data fetch for each
character
position will take only one bus cycle. Color memory is physically wired to
the VIC databus lines D8-D11.

How does VIC know where to fetch the graphical information ? Some of you
know
the mystical formulas needed to mess with the pixels in the hires screen.
How are these functions obtained ? Are they just magic ? No, there are some
internal counters in VIC. They always point to the right place in graphics
memory and the address is determined like this:

A13  A12 A11 A10 A9  A8  A7  A6  A5  A4  A3  A2  A1  A0
CB13 VC9 VC8 VC7 VC6 VC5 VC4 VC3 VC2 VC1 VC0 RC2 RC1 RC0

Address bits A15 and A14 change according to the selected video bank.
Address bit A13 is CB13, which may be found in VIC register $18. It
selects the right side of the video bank to be the bitmap memory. With
these bits you can set the bitmap to eight different places in memory.
However, some of them are useless because of the character ROM images and
```

zero page/stack. Rest of the bits come from the internal counters.

VC9-VC0 (Video Counter) forms the address bits 12-3. The counter rolls through all 1000 character positions, 0-39 on the first eight lines, 40-79 on the second eight lines and so on. The lowest three bits come from the row counter, RC2-RC0. This is another VIC counter and it counts the scan lines from zero to seven.

A software graphics mode - FLI

VIC will systematically go through every byte in the bitmap memory, but how does it know where and when to get the color information ? This is where the main principle of FLI (Flexible Line Interpretation) lies. Color data is fetched (and this means it is a bad line), when the line counter matches with the vertical scroll register. VC9-VC0 defines where the color data is inside the video matrix and color memory.

If we change the vertical scroll register, we can fool VIC to think that every line is a bad line, so it will fetch the color information on every line too. Because VIC will fetch the colors continuously, we can get independent colors on each scan line. We just have to change colors and VIC will handle the rest. Unfortunately the result is the loss of 40 processor cycles per line (see the Missing Cycles article for more information about VIC stealing cycles).

Doing it in practice

In practice there is no time to change color memory, but in multicolor mode VIC uses video matrix for color information too. We have just enough time to change the video matrix pointer, $D018. Now VIC will see a different video matrix on each scan line, different block of memory. With the four upper bits in the register we select one of the 16 video memories in the video bank. Just remember that the register also selects the position of the graphics memory (bitmap) inside the video bank.

Because we have to keep the bitmap in the same video bank, we only have half of the bank free for video matrices. Fortunately, that's all we need to get individual multicolor colors for each line and character position.
VIC will fetch the color data from the eight video matrices and then it will roll on to the next 40 bytes. After eight lines and matrices we will select the first video matrix again. (See picture 1)

Usually it is not necessary to use the whole screen for a FLI picture, especially if you want to have a scroller or some other effects. You just have to make sure that VIC is foolable in the usual way. The timing is also very important, even one cycle variations in the routine entry are not allowed. There is many ways to do the synchronization. One way is to use a sprite, as in the previous article. (See C= Hacking, Vol. 1, Iss. 3, The Demo Corner: Missing Cycles).

Not much time

Because a bad line will steal 40 cycles, there is only 23 cycles left on each scan line. It is enough for changing the video matrix and background color. There is not a moment to lose, because you must change the vertical scroll register, video matrix pointer and the background color. This is why you can't have sprites in front of a FLI picture.

With FLI we get two selectable colors for each character position and line, each scan line can have it's own background color and each character position
still has its own character color from color memory. In theory each character
position could have 25 different colors, unfortunately VIC only has 16.

A little feature

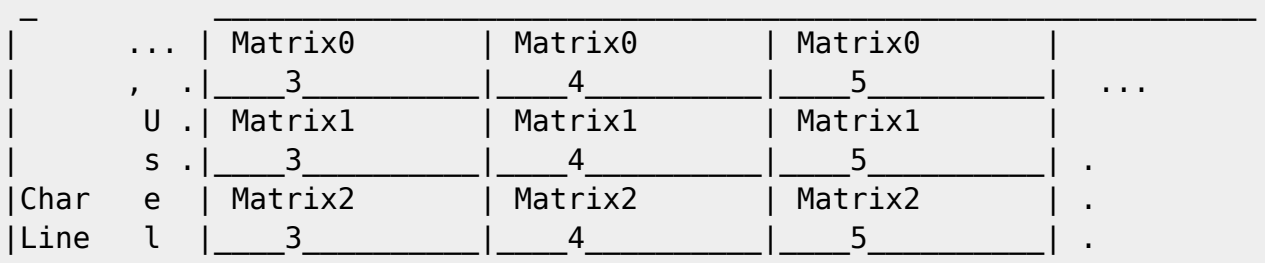
VIC does not like it when we change the vertical scroll register ($D011), and is a bit annoyed. It 'sees' code 255 (light gray) in video matrix and 9 (brown) in the color memory instead of the correct values stored there.
Actually the color value seems to be the lower nybble of the data byte currently on the data bus (accessed by the processor (LDA#=$A9)).
Unfortunately there is no chance to do the register change in the border and thus the three leftmost character columns are a bit useless, because the colors are fixed.

However, this doesn't mean that you can't use those three columns. FLI editors may not support the fixed colors though, so it may be hard to use them.

What to do with FLI ?

Because FLI will eat up all the available processor time (no Copper :-), it is not suitable for any action-games. Each FLI picture takes about 17 kB of memory: not so many pictures fit on one floppy. So, the only place for FLI
is demos, intros, board-type games and maybe a GIF viewer..

Picture 1: From which matrix VIC fetches the multicolor values



```
|Zero   e  | Matrix3       | Matrix3       | Matrix3       |
|       s  |____3__________|____4__________|____5__________|
|       s, | Matrix4       | Matrix4       | Matrix4       |
|          |____3__________|____4__________|____5__________|
|       c  | Matrix5       | Matrix5       | Matrix5       |
|       o  |____3__________|____4__________|____5__________|
|       l  | Matrix6       | Matrix6       | Matrix6       |
|       u  |____3__________|____4__________|____5__________|
|       m  | Matrix7       | Matrix7       | Matrix7       |
|_      n  |____3__________|____4__________|____5__________|
          s  | Matrix0       | Matrix0       | Matrix0       |
             |___43__________|___44__________|___45__________|
             | Matrix1       | Matrix1       | Matrix1       |
             |___43__________|___44__________|___45__________|
             |
             |   ...
             | .
             | .
             | .
```

Additional reading

If you have an Amiga you might want to get your hands into my conversion programs in C64GFX1.lha. The packet also includes FLI viewer for PAL C64's and some documentation about the FLI file format. It also has the same utilities for Koala format pictures.

Available from:
cwaves.stfx.ca
nic.funet.fi:/pub/amiga/graphics/applications/convert

C64GFX.doc

```
                        C64Gfx1.0
           A C64 graphics format conversion package
           )1991,1992  Pasi 'Albert' Ojala

           E-mail: po87553@cs.tut.fi
                   albert@cc.tut.fi
```

This package contains programs which are used to convert portable pixmap (ppm) files to C64 graphics formats (FLI and koala) under AmigaOS. The package includes C source codes for the programs, so it is possible to port the programs to another environment. C64GFX1.1 includes Unix-compilable sources.

In addition to this package you need e.g. PBMPlus to convert Amiga ilbm files to ppm first. And of course some way to transfer files between the machines.

```
================================================================================
=
```