#(@)top:

Issue #11
Version 1.0
December 1995

#(@)contents: Table of Contents

#(@)legal: Commodore Hacking Legal Notice

registered trademarks of ESCOM GmbH. Commodore hacking is in no way affiliated with ESCOM GmbH, owners of said trademarks. Commodore Hacking is published 4 times yearly by:

Brain Innovations Inc.
602 N. Lemen
Fenton MI 48430

The magazine is published on on-line networks free of charge, and a nominal fee is charged for alternate mediums of transmission.

#(@)info: Commodore Hacking Information

Commodore Hacking is published via the Internet 4 times yearly, and is presented in both ISO-8859-1 and HTML versions. This and previous issues can be found at the Commodore Hacking Home Page (http://www.msen.com/~brain/chacking/), as well as via FTP (ftp://ccnga.uwaterloo.ca/pub/cbm/hacking.mag/)

In addition, the Commodore Hacking mail server can be used to retrieve each issue. To request a copy of an issue, please send the following electronic mail message:

To: brain@mail.msen.com
Subject: MAILSERV
Body of Message:

```
help
catalog
send c=hacking11.txt
quit
```

To subscribe to the Commodore Hacking and receive new issues as they are published, add the following command to you MAILSERV message prior to the quit command:

```
subscribe c=hacking Firstname Lastname msglen
```

(msglen is largest size of file in kilobytes you can receive in an email message. When in doubt, choose 64)

example:

```
subscribe c=hacking Jim Brain 100
```

Although no fee is charged for this magazine, donations are gladly accepted from corporate and individual concerns. All monies will be used to defray any administrative costs, subscribe to publications for review, and compensate the individual authors contributing to this issue.

Any persons wishing to author articles for inclusion in Commodore Hacking are encouraged to view the submission guidelines on the WWW (http://www.msen.com/~brain/pub/c-hacking-submit.txt) or via the MAILSERV server (send c-hacking-submit.txt).

==========

#(@)rch: Reading C=Hacking

Starting with Issue 11 of Commodore Hacking, the new QuickFind indexing system is utilized to aid readers of the text version in navigating the

magazine. At the top of each article or other important place in the magazine, a word prefixed with a special string is present. (See the title of this article for an example. Throughout the magazine, if an article is mentioned, it will be followed by a reference string. For example, if we mentioned this article, we would add (Reference: rch) after the name. By using your favorite editor's search function and searching for the string after the word "Reference:", prefixed by the magic prefix string, will move you directly to the article of choice. To merely skip to the next article in the magazine, search only for the magic prefix string.

Some handy indexing strings possibly not referenced anywhere are:

```
top      top of issue
bottom   bottom of issue
contents table of contents
legal    legal notice
```

For those with access to a UNIX system, the command "what" can be run on the issue, which will result in all the article titles being printed.

A slightly different magic prefix string "#(A)" is used to delimit sub-topics or main heading in articles. The text after the magic string differs depending on article content. For the Input/Output column (Reference: io), the text after the magic prefix will either be "c" for comment, or "r" for response. In features and columns, a number after the prefix indicates the ordinal of that heading or sub-topic in the article. If a specific sub-topic is referenced elsewhere in the article, a sub-topic reference will be indicated. A reference to "#(A)r" would be written as "(SubRef: r)".

As time goes on, the role of this indexing system will be expanded and changed to ease navigation of the text version, but minimize the clutter added by these extra items.

==========

#(@)editor: The Hacking Editor
by Jim Brain (brain@mail.msen.com)

Two new faces appear in this month's Commodore Hacking. One is its new editor, while the other is its new look. I hope neither causes anyone to worry about the content of the magazine. It's all still here. C=Hacking will continue to provide leading edge technical information about the Commodore computers we all know and love. The magazine will continue to cater to the Commodore computer programmer, whether it be in the areas of sound, graphics, algorithms, disk media access, or communications.

However, the role of the magazine continues to expand. It has been shown that many people other than CBM programmers read the magazine, and programmers have requested other information besides technical content be included in the magazine. To this end, Issue 11 contains many new features, including:

- "Hacking the Mags" (Reference: mags), which will summarize the other Commodore magazines in the market place. Not everyone can read or subscribe to all the quality CBM publications out there, so this column will alert readers to specific issues that may be of interest.

- "Newsfront" (Reference: news), which will bring the Commodore programmer and user up to date on developments in the Commodore community. The Commodore world doesn't stand still, and every programmer should be aware of the newest technologies affecting the CBM line.

- "The Error Channel" (Reference: error), which will formalize the process of fixing errors in earlier issues. Hopefully, this will be unnecessary in most issues, but it will be here just in case.

- "Input/Output" (Reference: io), which will allow C=Hacking readers space for comments and concerns. Many readers have sent me suggestions and comments, some C=Hacking can implement, and some C=Hacking cannot. This spot will detail which is which and why.

- Article separators. As you can see above, each article or column in the magazine is delimited by the special key, followed by a short name of the article. See "Reading C=Hacking" (Reference: rch) in this issue.

- Smaller size. The last issue was over 400kB in size, which generated many complaints. There is no need to create such a long issue, when more issues can be published. This issue should comfortably fit on two sides of a 1541 disk, a 1571 disk, or a 1581 disk.

o Stable publication dates. Circumstances (college, job hunt), made it
   hard for the previous editor to maintain a schedule, so no blame is laid,
   but the magazine does need some stability. Although possibly unrealistic,
   I am striving to publish C=Hacking quarterly, with the following schedule:

      Publication Date   Submission Deadline
      March, 1996   February 10, 1996
      June, 1996   May 10, 1996
      September, 1996   Auguest 10, 1996
      December 1996   November 10, 1996

   If article submissions keep up, a switch to bi-monthly publication might
   be warranted, but I won't get too far ahead.

o Fully HTML-ized version of the magazine. Issue 11 contains many
   improvements designed to make the publication of an World Wide Web
   readable version of the magazine easier. Look for the HTML version of
   this and older issue at URL: http://www.msen.com/~brain/chacking/.

Many people have compared Commodore Hacking to the defunct _Transactor_
magazine, which is encouraging. The new format will hopefully add to the
appeal of Commodore Hacking.

Although many of you know me or of me through previous Commodore work, this
is my first editorship, so please comment on the changes I have made and what
your opinions on each are. As the magazine is for you, the reader, it is
always important to keep the reader happy.

Sadly, some things, like the WWW browser for C=Hacking, did not get done,
but there is always next time.

Enjoy YOUR magazine,

Jim Brain (brain@mail.msen.com)
editor

==========

#(@)io: Input/Ouput

Obviously, Commodore Hacking depends on the comments and article submissions
from the Commodore community to flourish. Everyone sees the articles, but
let's not forget those comments. They are very helpful, and every attempt
is made to address concerns in them. Address any comments, concerns, or
suggestions to:

Commodore Hacking
602 N. Lemen
Fenton, MI 48430
brain@mail.msen.com (Internet)

#(A)c: Need Samples of Samples

From: "Clifford \"Paska\" Anderson" <andersoc@saturn.uaamath.alaska.edu>

Dear C=Hacking,
Hey. Just writing to mention something I'd like to see in C=Hacking, if
you can find someone to write about it. I am interested in knowing more
about how samples work on the 64, how to play, etc.

Sī valēs, valeō

#(A)r:
Your wish is granted. Check out this issue's Hi Tech Trickery
(Reference: trick) by George Taylor for some insight into playing
samples.

#(A)c: You Index, I Index, We all Index

From: coyote@wakko.gil.net

Dear C=Hacking,
I would like to offer an idea for the Chacking mag. Every now and then
I'll come across a reference to an article in this or that Chacking Issue.
I run Zed and load that issue in (Thanks Mr Bruce) and start hunting for
the start of the article.

This process would be made a lot easier if Chacking used a method of
indexing I have seen used in several publications.

It involves the search function of most text editors. A 2/3/4 ? letter code (with a delimiter char to prevent unintentional matches) that the reader uses to find the beginning of the article.

(Outline of suggestion deleted)

I would like to add a personal thanks for all your efforts on behalf of the C= community.

Al Anger
13841 SW 139 Ct.
Miami Fl. 33186
(305) 233-4689

#(A)r:
Fire up that search function in your favorite editor. Issue 11 now contains the QuickFind indexing system that should fit your needs. See "Reading C=Hacking" (Reference: rch) for information on how to utilize the indexing system. We would like to add that C=Hacking appreciates your personal thanks.

#(A)c: Are We Talking the Same Language?
From: Jack Vander White <ceejack@crl.com>

Dear C=Hacking,
Noticed something that may be a potential problem and thought I would let you know.

Way back when hacking mag started I didn't have Internet access. the first couple of issues were sent to me by fellows who had downloaded them and in downloading had set their terms to translate them to PETSCII. This of course changed the coding in the uuencoding parts of the magazine and made them decode improperly.

Since then I have my own access and have re-downloaded them and posted them on my BBS in the original straight ASCII so that those who download them can uudecode the relevant parts and then translate the text for reading.

Since different Terminal Programs are using different Translation tables I can see all kinds of problems in this for the average user.

Any comment????

Jack VW

#(A)r:
The HTML version of Commodore Hacking utilizes the ISO-8859-1 text encoding standard, while the text version utilizes its 7-bit subset, commonly called ASCII. Normally, the encoding standard poses little problem, as text uses but a nominal set of characters which can be translated between PETSCII and ASCII. However, as you point out, the uucode format uses more characters, which may or may not be translated correctly. To circumvent this problem, which only occurs in the text version, the files embedded in a Commodore Hacking issue should be uudecoded prior to converting the file to anoy alternate format.

==========

#(@)news: Newsfront

* Although not new news, Some still may not know that Creative Micro Designs, Inc., is currently designing the Super64CPU accelerator. This external 3" tall by 6" wide by 2" deep cartridge will allow Commodore computers to execute programs at either 10MHz or 20MHz (actually, 9 and 18 MHz, realistically). The unit uses the Western Design Center's 65C816S CPU, which is object code compatible with the 6502/6510/8502. The CPU, used in the Super Nintendo Entertainment System as well as other products, can be switched between 6502 emulation mode and "native" mode, which allows the following:

  - o access to 16 MB of RAM without bank switching.
  - o 64kB stack.
  - o 64kB zero page (now called "direct access").
  - o 16 bit registers.
  - o Support for virtual memory systems.

  The unit is scheduled for production in February, 1996, and will cost ~US$149.00 for the 10MHz unit and US$199.00 for the 20MHz unit.

* The following information was relayed to the USENET newsgroup comp.sys.cbm by Jack Vanderhite, editor and publisher of COMMODORE CEE disk magazine:

> Rather than reply to all the messages asking about DIEHARD I will tell all what has been happening over the last few days.
>
> Brian Crosthwaite, Publisher of Diehard, contacted CMD, Loadstar, and Commodore CEE this week with the following form letter faxed to each of us:
> ----------
>
> Diehard, the Flyer for commodore 8bitters is planning to cease publication and we are looking to transfer our subscription fulfillment. Our number of outstanding subscribers is approximately 8,400 and I would be willing to throw in the balance of the list, totaling approximately 12,000.
>
> Please call me at (xxx)xxx-xxxx if you are interested in acquiring these readers and names.
>
> Sincerely,
>
> Brian L. Crosthwaite
> Publisher
>
> ----------
>
> Each of us did contact Brian for further details. They are bleak. The total number of paper issues due to subscribers is approximately 64,000. This does not count the approximately 1,200 spinner subscribers which would make approximately 10,000 disks due.
>
> The cost of publishing alone would amount to approximately $100,000 for printing, layout, disks, ,mail cost, etc. Not taking into account the cost of articles, etc.
>
> when asked about money Brian's only comment was "There is none. It's gone."
>
> a further complication is that Tom Netsel told me last week that General Media says that Brian has assumed the obligation to deliver the balance of the Gazette subscriptions. I questioned Brian about this. Brian says that general media faxed him the terms of transference of the obligation and that he faxed back an acceptance of the terms. While I have not seen the actual faxes involved it does sound like offer and acceptance of a binding contract from here.
>
> Obviously, all of us have rejected this offer. I have been told that there is an issue of Diehard at the printers, probably printed. However, the printing bill alone is over $8,000 plus the cost of mailing. Since there is no money it sits there.
>
> If anyone were willing to assume the total obligations they would have to assume a liability of well over $100,000 over the next year before any returns from renewals would even make a dent in this huge obligation.
>
> Please Note: I am putting this out as a public message. This is ALL I know.
>
> Please do not come back at me asking questions. I have nothing more I can add to this.
>
> Jack VW

So, if you have outstanding issues of dieHard due you, as the editor does, the fears have been confirmed. However, for those who purchased the dieHard "Spinner" disk, read on for the encouraging news.

* The LOADSTAR disk magazine has been recently purchased from Softdisk Publishing by LOADSTAR principles Fender Tucker and Julie Mangham. Now owned by J and F Publishing, a corporation founded by Mr. Tucker and Mrs. Mangham, provide the magazine with even more flexibility Tucker states that now LOADSTAR is "more solvent then ever before". Existing subscribers will see no difference with this change, as Softdisk and LOADSTAR will continue to maintain a close relationship, with Softdisk continuing to handle subscriptions, in addition to other tasks.

In related news, J and F Publishing has agreed to fulfill the remainder of the outstading dieHard "Spinner" subscriptions. Although unfortunate that dieHard left its subscribers out in the cold, it

is commendable that these subscriptions will be fulfilled with LOADSTAR issues. The agreement will provide one issue of LOADSTAR for every two issues of the Spinner, as the Spinner was a single disk, whereas LOADSTAR is double that. No word has been heard yet on the fate of dieHard paper subscriptions. All 1200 Spinner subscribers should be receiving information about the subscription fulfillment soon.

* For those people wishing to use the Internet with their Commodore 64, only to find out that the local Internet Service Provider (ISP) only provides Serial Line Internet Protocol (SLIP) service with no shell account service, help is coming. A prototype Transmissions Control Protocol/Internet Protocol (TCP/IP) protocol stack with SLIP support has been developed by Daniel Dallmann (Daniel.Dallmann@studbox.rus.uni-stuttgart.de) of Germany. Available now via the Internet (ftp://131.188.190.131:/pub/c64), the package is by no means complete, but does include the basic TCP/IP stack, the SLIP driver, and a rudimentary Telnet application.

* Another Commodore hardware/software supplier has announced its online presence: Performance Peripherals Incorporated. Maker of the RAMDrive and BB units (BBGRAM, BBRTC, and BBGRam), PPI published an online catalog that users can retrieve via the C=Hacking WWW Site (http://www.msen.com/~brain/pub/PPI_catalog.11.95.txt) and the C=Hacking MAILSERV server. (send PPI_catalog.11.95.txt). In addition to importing FLASH8 (the 8MHz accelerator cartridge from Germany), PPI manufactures CommPort, which is a 6551 UART cartridge (ala Swiftlink) which has the basic 6551 functionality with the addition of switch selectable NMI or IRQ interrupt triggering and switch-selectable $de00/$df00 addressing.

* PPI has one more trick up its sleeve. PPI will be carrying Novaterm 9.6, the newest version of Nick Rossi's oft-used terminal program for the C64. The blurb follows:

> Novaterm 9.6 is a complete terminal emulation program on cartridge for the C64. Novaterm has several features such as 80 column ANSI on a stock C64, and compatibility with CommPort, RAMDrive, BBGRam, and many other hardware devices. Just connect a BBGRam, and Novaterm can use it as a "buffer" for storing text or as a "virtual disk" for quickly and easily downloading files. Definately the perfect setup for Internet usage. And since Novaterm is in cartridge form, the program loads in seconds, not minutes. Novaterm 9.6 is the latest version programmed by NICK ROSSI. Includes autoboot switch.

==========

#(@)trick: Hi Tech Trickery: Sample Dither
by George Taylor (yurik@io.org)

#(A): Introduction

You may know of dithering in graphics. It is when a limited number of colors are used to simulate more. This is done by randomly arranging the pixels so they blend at a distance, creating an average of the shades. Here, screen space is being averaged, and more shades are being produced. In playing samples, time is being averaged, and more bits are being produced.

#(A): Dithering Sound

Let's say we do the following:

```
lda #8
sta $d418       This code toggles the low bit of the output.
lda #9
sta $d418
```

Over an average of time, this is the same as:

```
lda #8.5        But we can't really do this.
sta $d418
```

This idea can be used to easily do 5 bit sound. Basically, we take a 5 bit sample, shift right, then add 0. If bit 0 was high, it will increment the 4 bit number. Then as this adding takes place, toggling bit 0, it will average out to give half a bit.

#(A): Is There a Catch?

There is one drawback though. This toggling can be heard as the high frequency square wave it resembles. You must use a high enough sample

rate so this can't be heard. Also it takes two bit toggles to create the 5th bit, so you must double the sample rate. In order to play 8.5, for example, you must play 8 and then 9, so the 8 and 9 must take half the normal time, or your sample will play too slow.
One other problem is that there is the possibility of overflow. In this case you can use hard clipping on the signal. In other words, the 5 bit sample 31 will be played at 16, so instead play 15.

This is actually called pulse width modulation. It is a good example for illustrating sample dithering. For example, you can play TRUE 16 bit sound, even with one bit. To do this, take the 16 bit sample, add a 12 bit random number, then play the high 4 bits of this result. Also remember the clipping problem as mentioned above.

@(A): How Is This Like Pulse Width Modulation?

The random number range is proportional to the 16 bit sample. If the 16 bit number is high, then it is very likely the 0 bit (toggle bit) is high. It is the random number which allows the toggle bit to change. So now we have 16 bit sound with 16db signal to noise ratio.

There are some more advanced technical issues to this. The kind of random number you choose affects the results. You need a triangle density function for perfect linearity (ie., for no distortion). This is the relationship of random numbers in the sequence, and does not affect the probability distribution, which should be equal. The choice of density function is a tradeoff between added noise and linearity. I used pulse density function in my demo, which is non-filtered random numbers, and it's ok but I can still hear some noise pumping.

#(A): Conclusion

Enjoy the ditherdigi!

#(A)5bit: Listing One: 5 bit play routine

Memory map:
3:     start page of sample
4:     end page of sample
5:     sample period (remmber to play twice normal speed)
fb,fc: pointer to sample

```
start lda 3
      sta $fc
      lda #0
      sta $fb           ; initialize sample pointer
      lda #$b
      sta $d011         ; blank screen for better timing
      sei               ; disable interrupts for better timing
play  lda ($fb),y
      lsr
      sta $d418         ; push sample
      ldx 5
d     dex
      bne d
      pha
      nop
      adc #0
      cmp #$10
      beq s1
      sta $d418
      bpl s
s1    nop
      nop
      nop
s     pla
      ldx 5
d1    dex
      bne d1
      iny
      bne play
      inc fc
      lda fc
      cmp 4
      bne play
      lda #$1b
      sta $d011
      cli
end   rts
```

#(A): References

Consult the proceedings of the ACM for further info on digi dithering.

==========

#(@)mags: Hacking the Mags

Not everything good and/or technical comes from Commodore Hacking, which is as it should be. (I still think we have the most, though...) Thus, let's spotlight some good and/or technical reading from the other Commodore publications.

If you know of a magazine that you would like to see summarized here, let C=Hacking know about it. These summaries are only limited by Commodore Hacking's inability to purchase subscriptions to all the Commodore publications available. We are very grateful to those publications that send complimentary copies of their publications for review.

#(A): COMMODORE CEE

Volume 1, Issues 1 and 2 came all packaged as one "mega-issue". This particular double issue should be renamed the memory map issue, with I/O and/or memory maps for the VIC, 64, 128, and PET computers. Information on 6522 bugs and on the 6526 CIA chips that was cut from the final compilation of the Commodore 64 Prorammer's Reference Guide is of interest to Commodore Haking readers. Some of the information is culled from the Internet: the 64 memory maps, the info on the 6522, and a list of all the CSG produced IC numbers with descriptions. Of course, these files are also available on the Internet, if you have access. Howver, for those who don't know where to look or for those without access, the information is welcome. Issue 3 has a PCX to GEOPaint converter, much like _LOADSTAR_, and Issue 4 will begin a column on PAL to NTSC program conversions. One thing we'd like to see at Commodore Hacking is a better menu program, as the current one is somewhat hard to navigate.

#(A): Commodore World

Issue 10 just arrived at the computer room, with a snazzy front cover. Slick paper aside, the picture of Al Anger's Tower 128 was a masterpiece. Editor Doug Cotton spews about the hype of Windows 95, and the first ads for the Super 64 CPU accelerator are present. If you're into hardware mods, you can't miss page 4, which shows some other Al Anger hacked Commodore creations. Jim Butterfield's 4 page 65XX ML reference is useful for the newer programmers, and Doug Cotton's Assembly Line topic of serial routines will help those disk I/O challenged in the crowd. This issue details the high level routines, while #11 will tackle the low level disk I/O. Maurice Randall goes over event handling in GEOS, while Al Anger details how to disable the internal 1571D in the C128D. Gaelyne Moranec touches on the Internet nitty-gritty of learning UNIX commands and includes a table of UNIX-like commands found in ACE and LUnix. At the end, though, C=Hacking's burning question is: What hangup does Doug have with those abstract graphics sprinkled throughout the mag? There's nothing wrong with them, but some look like those psycho-analyst inkblot test cards.

#(A): Driven

Driven 9 contains a rundown on USENET (written by Jim Brain), which will help those Internet "newbies". For those doing cross development, the review of the PC<->C64/C128 networking system called 64NET by Paul Gardner-Stephen might help some get object code from the PC to the 64/128. Eddie Bourdon has some info on GEnie, including what Commodore support is available.

Driven 10 presents some useful WWW addresses, while XMikeX and Pegasus tackle the issues of apathy and pessimism in the Commodore community. Both make for good reading, but the best (in our opinion) was the pessimism piece. How many times have YOU been laughed out of CompUSA for mentioning that modem or SCSI drive was for a Commodore?

#(A): LOADSTAR

Issue 138 just finished loading on the 1581 disk drive, and the disk is packed with information. Fender Tucker goes into much detail on the recent changes at LOADSTAR and its new Publishing company, J and F Publishing. Of interest to programmers is the PCX to GEOPaint converter program, written by Fender Tucker and Doreen Horne. Some details on Al Angers machines that are shown in Commodore World are related. Jeff Jones presents a simple program pause routine, which fiddles with the NMI interrupt, and gives out source code as well. The Internet 101 series takes a month off from the LOADSTAR letter in #28, but is expected back next month. Lastly, Dave Moorman presents his fractal generator called FRACTAL MOUNTAINS. C=Hacking couldn't get it to work, but we think it's user error.

#(A): LOADSTAR 128

In Issue 29, Fender apologizes for not paying enough attention to the 800 LOADSTAR 128 subscribers. Of interest to programmers is the program listing pause program on the issue, but the rest is pretty light stuff, not to knock LOADSTAR. Different audiences need different material.

#(A): Vision

In Issue 7, Rick Mosdell has an article on graphics formats, updated and reproduced in this issue (Reference: gfx). There is some information from USENET reproduced, and a list of FTP sites as posted to USENET is also presented. Not much technical content in here, but C=Hacking was impressed with the graphics, music, and stories in the mag. Besides, everyone needs some time to enjoy the machine.

Other magazines not covered in this rundown include _The Underground_, _Gatekeeper_, _Commodore Network_, _64'er_, _Atta Bitar_ (_8 bitter_), as well as those C=Hacking is simply not aware of. As soon as we can snag a copy of any of these, or get the foreign language ones in English :-), we will give you the scoop on them.

==========

#(@)dbldma: Speed up RAMLink transfers with the Double-DMA Technique
by Doug Cotton (cmd-doug@genie.com) and Mark Fellows

#(A): Introduction

When CMD designed the RAMLink, we tried to make the system as fast as possible, but costs and complexity prohibited us from duplicating the operation of the DMA operation found in the Commodore RAM Expansion Unit (REU), The 8726 DMA controller found in the REU is a very complex item that allows the REU to transfer one byte per 1 MHz CPU clock cycle (1 microsecond). On the other hand, the RAMLink uses the 6510/8502 CPU load and store operations to transfer memory from the RAMLink memory to main memory. For the user who uses RL-DOS and RAMDOS, the difference is not noticeable, because although the RAMLink transfer is slower, RAMDOS continually pages its code in and out of main memory, effectively slowing its effective transfer speed down significantly.

But, what if the programmer isn't using RAMDOS? Then, the speed of the RAMLink becomes an issue. The RAMLink takes about 8 cycles to perform a transfer of a byte, while the REU does it in 1. This is significant. However, if a user owns both a RAMLink and an REU, there is a way to boost the transfer rate of the RAMLink via software. The method is called Double-DMA.

#(A): Double-DMA Description

Basically, the process is quite simple. Since the REU has the ability to transfe memory at 1 byte/microsecond, you can use the REU DMA to transfer memory from the RAMLink to main memory. To understand how we can do this, remember that the normal RL-DOS transfer routines use the CPU to perform the memory transfer. Well, to do that, at least some of the RAMLink RAM must be mapped into main memory. To be exact, 256 bytes is mapped in. So, to utilize the Double-DMA technique, the programmer simply makes the appropriate 256 bytes of RAMLink memory to be transferred visible in the main memory map, uses the REU to transfer that 256 bytes to the REU, and then uses the REU to transfer the 256 bytes in the REU to its destination in the main memory map. Thus, the Double-DMA technique will allow the RAMLink to transfer data at rouyghly 1/2 the speed of the REU, or 3-4 times faster than using the CPU to perform transfers.

#(A): The RAMLink memory map

To achieve this transfer speed gain, the programmer must forego RL-DOS usage and write specialized transfer routines. To do that, we need to discuss how the RAMLink maps itself into main memory and detail the various RAMLink registers needed to make this feat possible:

Address	Description
$de00	256 bytes of data (See $dfc0-$dfc3 for more information)
$df7e	write to this location to activate the RAMLink hardware
$df7f	write to this location to deactivate the RAMLink hardware.
$dfa0	lo byte of requested RAMCard memory page
$dfa1	hi byte of requested RAMCard memory page
$dfc0	write to this location to show RL variable RAM at $de00 (default)
$dfc1	write to this location to show RAMCard memory at $de00
$dfc2	write to this location to show the RAM Port device $de00 page at $de00
$dfc0	write to this location to show Pass-Thru Port dev. $de00 page at $de00

For all locations that have the description "write to this address...", the

program can safely write any byte to those locations, as the RAMLink hardware simply waits for an access, not any particular byte to be written.

#(A): Order of Operations

Although the Double-DMA technique relies on use of the REU, it is beyond the scope of this article to detail how to access the REU RAM under programmatic control. For more information on transferring data from the Commodore 128/64 and the 17XX REU, refer to the back of a REU owner's manual.

The following steps will realize the Double-DMA method:

```
Notes:  P = PAGE in RAMCard RAM to be transferred to/from
        A = PAGE of RAM in main memory to be transferred to/from
        X = single page of memory in REU used as temp RAM


 1)  if computer = 128, set up correct RAM bank
 2)  make I/O visible in main memory
 3)  sei
 4)  sta $df7e - activate RAMLink
 5)  lda #<P
 6)  sta $dfa0
 7)  lda #>P
 8)  sta $dfa1
 9)  sta $dfc1 - make $de00 show PAGE of RAM on RAMCard
```

Now, with the RAMLink hardware enabled in this way, the REU registers are also visible, so one can do a double DMA transfer at this point. There are two choices:

```
Transfer A->P:

10)  set up REU for A->X transfer
11)  initiate REU DMA transfer
12)  set up REU for X->$de00 transfer
13)  initiate REU DMA transfer

Transfer P->A

10)  set up REU for X->$de00 transfer
11)  initiate REU DMA transfer
12)  set up REU for A->X transfer
13)  initiate REU DMA transfer

Now, to go on:

14)  If more byte need transferrring, A=A+1, P=P+1, goto 5
15)  sta $dfc1 - restore contents of $de00
15)  sta $df7f - deactivate RAMLink hardware
16)  if computer = 128, restore bank
17)  restore I/O visibility if needed
18)  cli
```

#(A): Address Translation

To effectively use the Double-DMA technique, a programmer will want to set up a DACC partition in the RAMLink for use as external RAM. The programmer will need to determine the start address of the partition with the RL-DOS G-P command (or its sister command, G-[shift]P) This command will return the address of the DACC partition, or will it?

The answer is: Maybe. If a user has inserted an REU into the RAMLink RAM port and has the Normal/Direct swittch set to Normal, RL-DOS uses REU memory as the lowest RAM in the RAMLink memory map. However, when directly accessing the RAMLink and bypassing RL-DOS, the REU is not mapped into the RAMLink memory map. So, for such a condition, the code that determines the start of the DACC partition must SUBTRACT the size of the REU from the address returned by the G-P command. It's non-utopian, but the program need only do this once. However, for such an REU configuration, one must take care to ensure that at least 256 bytes of REU RAM is available and not already in use before utilizing the Double-DMA technique.

#(A): Performance

Craig Bruce, who has implemented this technique in his ACE operating system, provides the following performance figures for different access techniques:

Type	Bandwidth (bytes/sec)	Latency (~usec)	Notes

```
REU             1,007,641      65.8 REU in Direct mode
REU thru RL     1,007,641      77.8 REU in RAM Port in Normal mode
RAMLink           105,792     199.2 Regular RAMLink access
RL with REU       372,827     319.8 Double-DMA
Internal RAM0     120,181      44.2 Zero-page
Internal RAM1      80,283      56.3 All main memory except zero-page
```

So, using this technique in ACE results in a 3.7x increase in transfer speed. For some applications, that is well worth the trouble.

#(A): Conclusion

Obviously, CMD recommends that the RL-DOS be used for most operations, but we realize that some programmers simply need faster transfer rates. The Double-DMA technique should provide the speed needed from the RAMLink. Obviously, since this technique bypasses RL-DOS, code using it can potentially corrupt RAMLink memory if errors occur or if the technique is improperly used. When using the technique, we recommend extensive testing using various DACC partitions and different REU configurations to ensure proper operation.

#(A)ddcode: Double-DMA Code

Following is a set of functions that will perform transfers using Double-DMA. They are copied from the routines used in Craig Bruce's ACE operating system, Release 14, which incorporates the Double-DMA method. We thank Craig for the code below:

```
; Name:         Double-DMA memory transfer
; Author:       Craig Bruce
; Date:         1995-12-4
; Description: The following routines use the Double-DMA technique to transfer
;               memory to/from main RAM and the RAMLink.  If no RL is present,
;               normal CPU transfer methods are utilized.
;
; Variables:    [mp] holds the address of RAMCard memory to transfer
;               ramlinkNearPtr hold the address of main memory to transfer
;               ramlinkLength is length of data to transfer
;               ramlinkOpcode = $90: main memory -> RL
;                             = $91: RL -> main memory

reu = $df00
rlActivate   = $df7e
rlDeactivate = $df7f
rlSram       = $dfc0
rlPageSelect = $dfa0
rlPageActivate = $dfc1
rlPageData   = $de00

ramlinkOpcode .buf 1
ramlinkLength .buf 2
ramlinkNearPtr .buf 2
ramlinkMpSave .buf 3
ramlinkZpSave .buf 2

ramlinkOp = *  ;( [mp]=farPtr, ramlinkNearPtr, ramlinkLength, ramlinkOpcode )
   lda mp+0
   ldy mp+1
   ldx mp+2
   sta ramlinkMpSave+0
   sty ramlinkMpSave+1
   stx ramlinkMpSave+2
   lda zp+0
   ldy zp+1
   sta ramlinkZpSave+0
   sty ramlinkZpSave+1
   lda ramlinkNearPtr+0
   ldy ramlinkNearPtr+1
   sta zp+0
   sty zp+1
   clc
   lda mp+1
   adc aceRamlinkStart+0
   sta mp+1
   lda mp+2
   adc aceRamlinkStart+1
   sta mp+2
-  lda ramlinkLength+0
   ora ramlinkLength+1
   beq +
   jsr rlTransferChunk
```

```
    jmp -
+   lda ramlinkMpSave+0
    ldy ramlinkMpSave+1
    ldx ramlinkMpSave+2
    sta mp+0
    sty mp+1
    stx mp+2
    lda ramlinkZpSave+0
    ldy ramlinkZpSave+1
    sta zp+0
    sty zp+1
    clc
    rts

    rlTrSize .buf 1

    rlTransferChunk = *  ;( [mp]=rlmem, (zp)=nearmem, rlLength, rlOpcode )
    ;** figure maximum page operation
    lda ramlinkLength+1
    beq +
    lda #0
    ldx mp+0
    beq rlTrDo
    sec
    sbc mp+0
    jmp rlTrDo
+   lda mp+0
    beq +
    lda #0
    sec
    sbc mp+0
    cmp ramlinkLength+0
    bcc rlTrDo
+   lda ramlinkLength+0

    ;** do the transfer
    rlTrDo = *
    tay
    sty rlTrSize
    jsr rlPageOp

    ;** update the pointers and remaining length
    clc
    lda rlTrSize
    bne +
    inc mp+1
    inc zp+1
    dec ramlinkLength+1
    rts
+   adc mp+0
    sta mp+0
    bcc +
    inc mp+1
+   clc
    lda zp+0
    adc rlTrSize
    sta zp+0
    bcc +
    inc zp+1
+   sec
    lda ramlinkLength+0
    sbc rlTrSize
    sta ramlinkLength+0
    bcs +
    dec ramlinkLength+1
+   rts

    rlPageOp = *  ;( [mp]=rlmem, (zp)=nearmem, .Y=bytes, ramlinkOpcode )
    php
    sei
    sta rlActivate
    lda mp+1
    sta rlPageSelect+0
    lda mp+2
    sta rlPageSelect+1
    sta rlPageActivate
    lda aceReuRlSpeedPage+3
    bne rlPageOpReu  ;xxx dependency on aceMemNull==0
    rlPageOpNonReu = *
    tya
    clc
```

```
   adc mp+0
   tax

   lda ramlinkOpcode
   cmp #$91
   bne rlPageOpWrite
   dex
   dey
   beq +
-  lda rlPageData,x
   sta (zp),y
   dex
   dey
   bne -
+  lda rlPageData,x
   sta (zp),y
   jmp rlPageOpContinue

   rlPageOpWrite = *
   dex
   dey
   beq +
-  lda (zp),y
   sta rlPageData,x
   dex
   dey
   bne -
+  lda (zp),y
   sta rlPageData,x

   rlPageOpContinue = *
   sta rlSram
   sta rlDeactivate
   plp
   rts

   rlPageOpReu = * ;( [mp]=rlmem, (zp)=nearmem, .Y=bytes, ramlinkOpcode )
   ;** ramlink hardware already switched in
   ldx #1
   tya
   beq +
   ldx #0
   cmp #0  ;xx cut-off value
   bcc rlPageOpNonReu
+  ldy ramlinkOpcode
   cpy #$90
   beq +
   ldy #$90             ;rl->reu->intern
   jsr rlPageOpReuRl
   ldy #$91
   jsr rlPageOpReuIntern
   jmp ++
+  ldy #$90             ;intern->reu->rl
   jsr rlPageOpReuIntern
   ldy #$91
   jsr rlPageOpReuRl
+  sta rlSram
   sta rlDeactivate
   plp
   rts

   rlPageOpReuIntern = *  ;( .AX=bytes, .Y=op )
   sta reu+7  ;len
   stx reu+8
   sty temp1
   pha
   lda zp+0
   ldy zp+1
   sta reu+2
   sty reu+3
   lda aceReuRlSpeedPage+0
   ldy aceReuRlSpeedPage+1
   sta reu+4
   sty reu+5
   lda aceReuRlSpeedPage+2
   sta reu+6
.if computer-64
   ldy vic+$30
   lda #0
   sta vic+$30
.ife
```

```
   lda temp1
   sta reu+1
.if computer-64
   sty vic+$30
.ife
   pla
   rts

   rlPageOpReuRl = *  ;( .AX=bytes, .Y=op )
   sta reu+7  ;len
   stx reu+8
   sty temp1
   pha
   lda mp+0
   ldy #>rlPageData
   sta reu+2
   sty reu+3
   lda aceReuRlSpeedPage+0
   ldy aceReuRlSpeedPage+1
   sta reu+4
   sty reu+5
   lda aceReuRlSpeedPage+2
   sta reu+6
.if computer-64
   ldy vic+$30
   lda #0
   sta vic+$30
.ife
   lda temp1
   sta reu+1
.if computer-64
   sty vic+$30
.ife
   pla
   rts
```

==========

#(@)usenet: UseNuggets

COMP.SYS.CBM: The breeding ground of programmers and users alike. Let's see what topics are showing up this month:

#(A): We Want More Power!

CMD's announcement of the Super64 CPU accelerator got things stirred up in the newsgroup. When it was announced that the initial product would run on a C64 or on a C128 in 64 mode only, some angry C128 128 mode users vented all over the place. Everything from people wondering aloud what extra work the 128 version would require to threats of non-purchase of the unit ensued. Then, just as the first wave of fighting subsided, the next wave started, programmers worried about RAM transfer speed bottlenecks questioned CMD's decision not to include a DMA device on the unit to speed data transfers. CMD's response:

> From: Doug Cotton <cmd-doug@genie.geis.com>
> Newsgroups: comp.sys.cbm
> Subject: Re: Power Users!
> Date: 28 Nov 1995 00:59:26 GMT
> Organization: Creative Micro Designs, Inc.
>
> There were some earlier questions about how fast memory transfers could be accomplished with the accelerator, and at least one individual emailed me over the lack of a DMA controller. I obtained some figures from Mark concerning this. Presently, the DMA transfers using an REU transfers a byte in 1 microsecond. The accelerator can achieve this same speed when transferring data from either on-board static RAM, or from expansion memory (slower DRAM) to the host computer RAM. Transfers internally (from static RAM to static RAM) will take .35 microseconds per byte (350 nanoseconds). Transfers from RAMLink RAMCard RAM (direct style) to the host computer RAM will take about 2 microseconds per byte. The only figures I don't have yet are for transfers between on-board static RAM and expansion DRAM, but this will be governed by the speed of the DRAM itself, and the number of wait-states required. It definately will be faster than 1 byte per microsecond though. So the only thing slower than a current DMA operation is transferring to and from RAMLink RAMCard memory, which is still pretty impressive at half the speed of present DMA transfers.
>
> Given these speeds, the cost of high-speed DMA controllers ($$$$), and a real lack of anywhere to put one on the main board, I think

going without a DMA controller is reasonable. If you really want one, though, there's always the high-speed expansion port, and a do-it-yourself project.

Doug Cotton

Notice the tiny "high speed expansion port" mention at the end. Reports indicate that such a port or ports will definitely appear on the unit, but it is still undetermined whether a single connector or a small expansion bus will be utilized. Commodore Hacking recommends the latter, as more options for hardware mods are available.

#(A): Let's all design the Commodore 64 Laptop!

Yes, the dreamers are at it once again. Starting in late October, the net was abuzz with thoughts on what should be included on a Commodore Laptop. The designs were flying fast and furious, with many different features discussed. It was agreed that the laptop would need to be a power sipper and have an LCD screen and a keyboard. However, that was where agreement ended. Some of following items were bantered about:

CPU:

- "really fast" 6510
- 65C816S

Disk:

- FLASH RAM cards.
- built in hard drive
- low power 1581 or CMD FD2000/4000

RAM

- definitely more than 64kB, but disagreement as to how much more.

Video

- VIC-II compatibility with more modes.
- VIC-III as found in Commodore 65

Sound

- Built in stereo SIDs
- Quad SIDs

So, on and on it went. Some got down to the nitty gritty of planning designs for chips. Some wanted to put the SIDs into one chip, while others wanted a SID/VIC/CPU single chip solution.

It's December, and the thread is still going strong, but a few great things have surfaced, which is why you can't just discount this type of dreaming:

- Someone posted the procedure for modifying the C64 to run on battery power.

- A few people started looking into how much money such designing would require.

- Most people who thought disk media should be included agreed that the CMD FD drive could/should be used.

- Everyone woke up and noticed that the NMOS CPU process used for the fabbibng of the CBM chips was power hungry and ill-suited to battery operation.

C=Hacking encourages users to answer the quetion: My dream Commodore laptop computer would include.... Send you entries to Commodore Hacking (brain@mail.msen.com) with the subject "LAPTOP". We'll print the best entries next issue.

Everyone seems to think that CMD is going to have one in development before long. Dunno. Commodore Hacking has heard rumors of what is going on at CMD, but we haven't heard about the laptop project. Of course, we're not SPECIAL or anything.... :-)

#(A): The Tower of Power

It seems Al Anger's (coyote@gil.net) Tower 128 picture on Commodore World's Issue 10 cover got everyone excited. A couple of people were sending Al

email about it, Commodore Hacking asked some questions, and some USENETters were deciding how to do it themselves. Al states that $2000 would just about cover it, which turned a few enquiring minds away, we're sure. Still, the reasons given for wanting a tower were solid. Commodore users are getting tired of all the clutter and mess cables, power cords, expansion extenders, Swiftlink cartridges, etc. make in the computer room. C=Hacking notes that at least one manufacturer produces tower 64 systems, but the cost is evidently more than what most folks are willing to fork over (~US$300 - US$550). So, everyone is waiting for the cost to come down....

#(A): Dave Letterman, Eat Your Heart Out!

The latest thread is the top ten list of games. Everyone is submitting their 10 most favorite games for the CBM machines. (Is anyone compiling these?) Anyway, it turns out this thread has a nice side effect. People are reminiscing about the old games, and the Commodore users are noting that the new games "just aren't as good". Here, here!

So, that wraps up the USENET this time. We try to keep an eye out for stuff of interest, but drop us a line if you think we might miss an IMPORTANT topic...

==========

#(@)toolbox: The Graphics Toolbox: Ellipses
by Stephen L. Judd (sjudd@nwu.edu)

#(A): Introduction

After a much needed break from Commodore 64 programming, I thought it would be nice to construct another algorithm for the 2D graphics toolbox. Since we did circles last time, a natural successor would be an algorithm to draw eclipses. We will first review the circle algorithm, and then build upon it to draw eclipses. You may recall that the algorithm had problems with small-radius circles. There is a very easy way to fix this, so we will cover that issue as well.

#(A): Circles

Recall that the equation for a circle is

$$x^2 + y^2 = r^2$$

After taking differentials of both sides, we find that

$$dy = -x/y\ dx$$

That is, if we take a step of size dx in the x-direction, we in principle want to take a step of size dy in the y-direction.

Next we start at the top of the circle, so that y=r and x=0. We start increasing x in step sizes of one. We only care about step sizes of one, since our basic unit is now a pixel. The y-coordinate is going to start piling up these dy's, and at some point the integer part of y will increase, and we get a new y-coordinate for the pixel. The idea, then, is to keep adding the dy's together, and once their sum is greater than one, we decrease y (remember that y starts at the top of the circle).

The sneaky way to do this is to treat y as an integer "constant". Then it is very easy to add the dy's together, since they have a common denominator equal to y. So really all we need to do is start adding x-coordinates together, and once their sum is larger than y, we decrease y and hang on to the remaining fractional part of dy. The algorithm then looks like:

```
      y=r
      x=0
      a=r
loop: x=x+1
      a=a-r
      if a<=0 then a=a+y:y=y-1
      plot (x,y)
      if x<y then loop:
```

Now, Chris McBride pointed something out to me. As you may recall, the algorithm breaks down for small r. Chris said that if a is initially set to r/2 instead of r, the algorithm works perfectly. Why is that? Recall that we add dy to itself until it is greater than one. Wouldn't it make more sense to add dy to itself until it is greater than 0.5? That would have the effect of rounding things up. Thus, starting at r/2

is like adding 0.5 to the fractional part of y -- it is the difference between INT(y) and INT(y+0.5).

Thus, the above line

```
a=r
```

should be changed to

```
a=r/2
```

for a perfect circle every time. Thus, this corresponds to adding an LSR to the machine code. Incidentally, this fix appeared in an earlier C=Hacking, but it was placed in such a crazy place that you probably never saw it.

#(A): Ellipses, HO!

Now we can move on to eclipses. Since ellipses are simply a squashed circle, it seems reasonable that we could modify the above circle algorithm. So, let's get to it!

Everyone knows the equation of an eclipse:

```
x^2/a^2 + y^2/b^2 = 1
```

Upon taking differentials of both sides we have,

```
2*x*dx/a^2 + 2*y*dy/b^2 = 0
```

or, equivalently,

```
dy = -b^2/a^2 * x/y * dx
```

As you can see, life becomes suddenly becomes more complicated by a factor of b^2/a^2. Furthermore, with an eclipse we only have reflection symmetries through the x- and y-axis. In the circle algorithm we could get away with just drawing an eighth of the circle, but now we have to draw a full quarter of the eclipse.

We will start drawing the eclipse at x=0, y=b, so that initially x will increase by one at each step, and y will wait a few steps to increase. At some point, though, we will want y to increase by one at each step, and x to wait a few steps before increasing; in the circle algorithm we just quit once we reached this point, but now we are going to need an equation for dx:

```
dx = -a^2/b^2 * y/x * dy
```

In the circle algorithm, we used a single variable to count up and tell us when it was time to increase y. Perhaps your intuition suggests that we can do an eclipse with _two_ variables; mine said the same thing, so that is exactly what we will do.

First, let us assume we have a way of calculating b^2/a^2:

```
E = b^2/a^2
```

I will suggest a way to perform this calculation later. Let's write out the first few terms in the dy summation, starting at x=0, y=b:

```
dy1 + dy2 + ... = -E * (x0 + x1 + x2 + x3 + ...)/y
                = -E * (0 + 1 + 2 + 3 + ...)/b
                = - (0 + E + 2E + 3E + ...)/b
```

So, the basic structure of the algorithm is: add up 0, E, 2E, etc. until the sum is larger than y. At that point, reset the counter, keeping the remainder, and decrease y. This is where the two variables come in:

```
X=X+1
T2=T2+E
T1=T1+T2
IF T1>=Y THEN T1=T1-Y:Y=Y-1
```

Do you see how it works? T2 simply takes on the values 0, E, 2E, 3E, etc., and T1 is the counter. Furthermore, you can see that once T2 is larger than Y, dy will be larger than one at each step. We need a new algorithm to continue the calculation, and it turns out to be quite simple.

Look at the expression for dx above. We could calculate a^2/b^2, but somehow that goes against the spirit of the calculation so far. Let's instead rewrite dx slightly:

```
   dx = - y/(E*x) * dy
```

Here we have simply written a^2/b^2 as 1/(b^2/a^2) = 1/E. But E*x is exactly the variable T2 above, so we can continue the calcuation without even stopping for breath:

```
   Y=Y-1
   T1=T1+Y
   IF T1>=T2 THEN T1=T1-T2:X=X+1:T2=T2+E
```

(remember that T1 keeps track of the fractional part of y). So, we now have a complete algorithm for drawing an eclipse:

```
   0 REM ELLIPSE ATTEMPT #N SLJ 11/3/95
   10 A=150:B=16:E=B*B/(A*A)
   20 X=0:Y=B:T1=0:T2=0.5
   30 GRAPHIC1,1:SLOW:X0=160:Y0=100:DRAW1,X0+A,Y0:DRAW1,X0,Y0-B
   40 X=X+1:T2=T2+E
   50 T1=T1+T2
   60 IF T1>=Y THEN T1=T1-Y:Y=Y-1
   70 DRAW1,X0+X,Y0-Y
   80 IF T2<Y THEN 40
   90 Y=Y-1
   100 T1=T1+Y
   110 IF T1>=T2 THEN T1=T1-T2:X=X+1:T2=T2+E
   120 DRAW1,X0+X,Y0-Y
   130 IF Y>0 THEN 90
```

Lines 40-80 are the top part of the eclipse, and lines 90-130 handle the bottom part. Note that T2 starts at 0.5, to round off the calculation in the same spirit as we did in the circle algorithm.

Naturally, this algorithm has a few limitations. In line 30 the start and end points are plotted, so you can see how close the algorithm really is. In my experiments it occasionally missed the endpoint by a pixel or two. As usual, I was a little too lazy to investigate possible ways to get around this. If you require a perfect eclipse, you need to start the calculation at x=0, y=b and run it forwards (e.g. lines 40-80 above), and then do another, similar calcuation, starting at x=a, y=0, and running backwards. That is, for the second calculation, calculate E2=a^2/b^2, and then run the algorithm just like lines 40-80, interchanging X and Y.

Now we need to translate this algorithm into assembly. I am going to make a few assumptions: first, that everything fits in a byte. In particular, I require that b^2/a < 256. This insures that b^2/a^2 < 256, and also insures that T2 will not overflow (note that when x=a, T2=E*a, e.g. T2=b^2/a). What this means is that eclipses can't be too squashed.

Next, we need to deal with the fraction E=b^2/a^2. Any number like this consists of two parts, an integer part plus a fractional part (e.g. a number and a decimal). So, let's split E into two parts, EL and EH, where EL represents the decimal part and EH the integer. Now our addition consists of adding together the fractional parts, and if there is an overflow, increasing the integer part. For example, if E=1.62, then EH=1 and EL=0.62. We add EL to our number, and if it is greater than one, we carry the one to when we add EH to our number.

The best thing to do is to represent EL as a fractional part of 256. That is, our EL above should really be 0.62*256. This way, carries and overflows will be handled automatically (this will become clear in a moment).

Let me give some pseudo-assembly code and we'll push off the explanation until later:

```
   35 GOTO 200
   190 REM **********************
   200 XM=0:YM=B:X=128:Y=0:EH%=INT(E):EL%=INT((E-EH%)*256+0.5)
   210 XM=XM+1
   220 C=0:A=X:A=A+EL%:IF A>255 THEN A=A-256:C=1
   230 X=A:A=Y:A=A+EH%+C:Y=A
   235 A=A+T1
   240 IF A>=YM THEN A=A-YM:YM=YM-1
   250 T1=A:DRAW1, X0+XM, Y0-YM
   260 IF Y<=YM THEN 210
   265 T2=Y:A=T1
   270 YM=YM-1
   280 A=A+YM:IF A<T2 THEN 300
   290 A=A-T2:T1=A:XM=XM+1:A=X:C=0:A=A+EL%:IF A>255 THEN A=A-256:C=1
   295 X=A:A=T2:A=A+EH%+C:T2=A:A=T1
   300 DRAW1, X0+XM, Y0-YM
   310 YM=YM-1:IF YM>=0 THEN 280
```

XM and YM are the x and y coordinates of the point to be plotted. Note that in line 200 X starts at 128, and this again is to round up all our calculations; compare to line 20, where we started T2 at 0.5. In the above code I store T2 in the X and Y registers for the first part of the code. Note that in lines 220 and 290 there is some extraneous code to simulate things that in assembly are taken care of by the 6502. Note also that the comparison in line 260 has been changed from < to <=. This makes the branch easier, and I'm not sure how it affects the calculation (I didn't notice any difference in the few runs I tried it on).

Moving through the code, we increase x, and then add the decimal part of E to the counter. Then we add the integer part of E to the counter, along with any carries. If the integer part of the counter is greater than y, it is time to decrease y and reset the counter.

Moving to the second part of the code, we do a little rearranging in line 265. Really a better thing to do would be to let A=T1-T2, so that the compare in line 280 becomes simpler. Anyways, note that the Y register becomes freed up at this point. From here on, it is pretty much the same thing as before.

The full assembly code is then:

```
   ;Ellipse SLJ 11/3/95 Assumptions:
   ;0->XM B->YM, x- and y-coordinates
   ;0->T1
   ;EL and EH contain remainder and integer parts of E, resp.

   LDX #128
   LDY #00
   CLC
L1 INC XM
   TXA
   ADC EL
   TAX
   TYA
   ADC EH
   TAY
   ADC T1
   CMP YM
   BCC :CONT1
   SBC YM
   DEC YM
:CONT1   STA T1
   JSR PLOT
   CPY YM
   BCC L1

   STY T2
   LDA T1
   SBC T2
   DEC YM
L2 ADC YM
   BCC :CONT2
   SBC T2
   STA T1
   INC XM
   TXA
   ADC EL
   TAX
   LDA T2
   ADC EH
   STA T2
   LDA T1
:CONT2  JSR PLOT
   DEC YM
   BPL L2      ;Assuming y<128
```

#(A): Logarithms

Finally, we need a way of calculating b^2/a^2. I suggest using logarithms for this. I do believe I discussed this concept in an earlier issue of C=Hacking. Nevertheless, the idea is that if

$$x = b^2/a^2$$

then

$$\log(x) = 2\log(b) - 2\log(a)$$

so that

```
x = exp(2*(log(b) - log(a))
```

Thus, three tables need to be created: one for log(x), and one each for the integer and remainder parts of e^(2*x). Now, to improve accuracy, the first table might be a table of f(x)=222/log(128) * log(x/2). This constant is chosen so that f(255) is roughly 255. 222 was chosen because the inversion (i.e. the e^x part) works best at that value. This pretty much assumes that x is not zero or one, either. You can of course use more tables for somewhat better accuracy.

One really nice thing about this is that you don't have to worry about squaring things, since that part can be taken care of automatically in logarithm space. On the downside, we are restricted even further by the types of numbers we can divide (e.g. log(a)-log(b) can't be larger than 127 or so).

Division then consists of a table lookup, subtraction of another table lookup, and two more table lookups. Here is a short program to demonstrate the use of logs in this sort of division, and a very rough feel for the type of accuracy to expect -- note that it doesn't compare decimal parts, or convert the decimal parts into fractions of 256, etc.:

```
1 FAST:PRINT"[CLR]"
10 DIM L(256),EI(256),ER(256):FC=222/LOG(128)
20 FOR I=1 TO 256
25 PRINT "[HOME]"I
30 L(I)= INT(FC*LOG(I/2)+0.5):IF I=1 THEN L(I)=0
40 S=I:IF I>127 THEN S=I-256
50 EX=EXP(2*S/FC):IF EX>256 THEN PRINT"WHOOPS! EX="EX"I="I
60 EI(I)=INT(EX+0.5)
70 ER(I)=EX-EI(I)
80 NEXT I
90 EI(0)=1:ER(0)=0
100 FOR A=2 TO 250
110 FOR B=2 TO 250
120 X=L(B)-L(A)
123 IF X>127 THEN PRINT"OOPS:A="A"B="B"X="X
126 IF X<0 THEN X=X+256
130 A1=EI(X)+ER(X):A2=B*B/(A*A):IF A2>255 THEN B=600
135 BL=INT(A2+0.5)-INT(A1+0.5)
140 PRINT A;B,A1;A2,"ERR="INT(A2+0.5)-INT(A1+0.5)
150 NEXT:NEXT
```

#(A): Conclusion

Sorry, no 3D graphics this time around. Watch for a full-screen, hires bitmapped solid 3D virtual world sometime in the not too distant future. Otherwise, may your ellipses never be square :).

==========

#(@)surf: Hack Surfing

For those who can access that great expanse of area called the World Wide Web, here is some new places to visit that are of interest to the Commodore community. In early 1994, when the US Commodore WWW Site started, the number of sites online that catered to Commodore numbered in the 10's. Now, the number is in the 100's. What a change.

If you know of a site that is not listed here, please feel free to send it to the magazine. The following links have been gleaned from those recently changed or added to the US Commodore WWW Site Links page (http://www.msen.com/~brain/cbmlinks.html).

To encourage these sites to strive to continually enhance their creations, and because we like to gripe :-), we'll point out an improvements that could be made at each site.

#(A): Companies

o http://www.escom.nl
  ESCOM Interactive, Inc. The new home of Commodore has links to many of its offices and some general information. The pages are still under construction, but you should probably save this address. C=Hacking gripe: No Commodore 8-bit inforation yet.

o http://www.msen.com/~brain/guest/cmd/

Creative Micro Designs. Stay tuned to this site for information on the accelerator, and keep up to date on the latests prices on CMD peripherals and software. C=Hacking gripe: For a comapny wanting having just announced the Super64CPU, no mention of it is to found anywhere on the WWW site. Bummer.

#(A): Publications

o http://www.softdisk.com/about/c64.html
LOADSTAR and LOADSTAR 128. If you are interested in LOADSTAR, check 'em out here. Some Commodore links are included, and the and a few magazine teasers are present. In addition, details on how to ordr LOADSTAR or any of its related software titles is provided. C=Hacking gripe: the background color. Yellow is hard on our eyes... Oh well.

o http://www.mds.mdh.se/~dat95pkn/8bitar/
Atta Bitar (8 Bitter) magazine. Full indexes for the past 3 years, as well as information on how to subscribe. We'd tell you more, but none of us read German (At least we THINK it's German), and the English transmation page isn't done yet. Anyway, if you would like to subscribe or need to search the index of the magazine, here's the place to go.
C=H gripe: Yes, we know this is English-centric, but we just wish we could actually read all the great info on this site.

#(A): User's Groups

o http://www.cucug.org/
Champaign-Urbana Commodore User's Group. Home of the Universtity of Illinois (the editor's alma mater!) Meeting dates and time, along with newsletters and a user group listing are presented. C=H gripe: No mention of what local CBM 8-bit users are doing. This site recently changed addresses, so change all your links...

o http://www.psc.edu/~eberger/pcg/
Pittsburgh Commodore Group. Local news, meeting dates and time, and some newsletters are present. This site has also recently relocated to this new address. C=H gripe: Same as for CUCUG. We want to know what the CBM 8-bitters are doing in Pittsburgh.

o http://www.slonet.org/~rtrissel/
The Central Coast Commodore User's Group. Those in the Santa Maria CA area will be glad to know that CCCUG is there for them. Past newsletter are available, and some links to other information of interest is present. C=H gripe: Meeting dates and times need to be present in some easy place. C=H plug: It sounds like this club might need a little help, as it is down on members. If you are in the Santa Maria area, consider joining...

#(A): Miscellaneous

o http://www.byte.com/art/9408/sec14/art1.htm
Byte Magazine's Commodore obituary, by Tom Halfhill. Tom spells out many of the things that Commodore DID do right in its lifetime, and reflects on the blunders CBM made. The article makes for very good reading, but will depress some. C=H gripe: The pictures in the real article aren't reproduced in the WWW page.

o http://stud1.tuwien.ac.at/~e9426444/geoswarp/index.html
GEOS Warp 1.0. For the Mac user who needs or wants to run GEOS, this program, run on a Macintosh, will allow GEOS programs to operate on the Mac. The system looks very impressive, to the point of us not asking, Why? C=H gripe: Not really with the page, but the writer laments that progress is slow owing to no agreement with GEOWorks. Such things may doom the project to failure.

o http://vanbc.wimsey.com/~danf/cbm/
Dan Fandrich's WWW Site. For those who develop on alternate platforms or use multiple programming languages with the C64/128, bookmark this page. Very current info, and lots of it is presented. Some defunct Commodore mags are indexed, and pointers are provided to many of the current crop of magazines, including this one. C=H gripe: the page needs a little bit more organization to make it easier to get at juicy info.

o http://www.aloha.net/~scatt/commodore.htm
Scatt's WWW Site. For those just moving into assembly language programming from BASIC or something else, this page has a beginner's tutorial you might find useful. C=H gripe: A little low on content, but we are glad what there is is available.

o http://www.cs.wm.edu/~pbgonz/progc64.html
Pete Gonzalez's WWW Site. Small page, but worth viewing. Pete shows some

screen shots of a new game he is developing, and offers copies of his in progress cross assembler for the PC. C=H gripe: When's the game coming out again? :-)

o http://www.ts.umu.se/~yak/cccc/
The Commodore Computer Cult Corner. Some people play games, and then some people PLAY games. Jonas Hulten has pictures of his game design, implementation, and programming heoes. You can read about each one, and even light a "candle" for them. This site has a CBM links page, which anyone can add their link to automatically. C=H gripe: We can add our home page automatically, but not our hero.

o http://www.slonet.org/~jwilbur/
John Wilbur's WWW Site. Basically, just a links page right now, but we'll check back. C=H gripe: We'd like to see a little more about John as it relates to Commodore.

o http://www.student.informatik.th-darmstadt.de/~supermjk/
Marc-Jano Knopp's WWW Site. Mainly a large links page for Commodore information, this site does give a glimpse of the never produced Commodore LCD laptop computer. C=H gripe: As above, we love to see a little more about Marc-Jano as it relates to Commodore.

==========

#(@)zedace: Design and Implementation of an Advanced Text Editor
by Craig Bruce (csbruce@ccnga.uwaterloo.ca)

Note: Due to the size of the article, no executable or source code is included, but both will be included in ACE Release #15 (ftp://ccnga.uwaterloo.ca/pub/cbm/os/ace/).

#(A)1: 1. INTRODUCTION

This article discusses the design and implementation of the ZED text editor for the ACE operating system (Release #15 and higher). The program and full source code will be freely available when they are ready. ZED is written entirely in assembly language (ACEassembler) and makes heavy use of the full-screen control capabilities of ACE. However, part of the genius of the design of the ACE interface is that its facilities could be replicated into a standalone environment and a new ZED could be made into a one-part program (for greater convenience, with less flexibility).

There was a previous version of ZED, which WAS a standalone program. It was written entirely in _machine_ language (as opposed to assembly language; y'know, hexadecimal and stuff, with a machine-language monitor). Needless to say, upgrading and maintaining the program was a real problem, even though it was only 17K in size. The program also had a couple of limitations, the most serious of which being that all lines were limited to a maximum of 80 characters (plus a carriage return), or they would be split into two physical lines internally. It would also work only on the 80-column C128.

Still, the standalone version had a number of outstanding capabilities, including the ability to edit EXTREMELY large files with an REU and dynamic data structures, the ability to use "burst mode" on devices that supported it, TAB and character-set translation on loading and saving, global search and replace, range delete and recall, and a paragraph "juggling" feature. It is truly an outstanding program (if I do say so myself), and it is my pleasure to use it every day. (I also use Unix's "vi" every day, and I am getting tired of its clumsy user interface... I may just have to port ZED to the Unix environment one of these days).

The ACE version has/will have all of these features and then some. The ACE version supports even larger files by using the ACE dynamic memory management system (see C= Hacking #7 or a newer ACE Programmer's Reference Guide) in addition to its own byte-oriented memory management (see C= Hacking #2) with internal memory (up to 512K on a C128), REUs up to 16 Megs, and RAMLink DACC memory up to 16 Megs. Burst-mode support isn't currently available in ACE (see C= Hacking #3), nor does the ACE version of ZED currently implement the other outstanding editing features of the original ZED mentioned above. (For another C= Hacking reference, ACE does support a three-key rollover for typing convenience (see C= Hacking #6)).

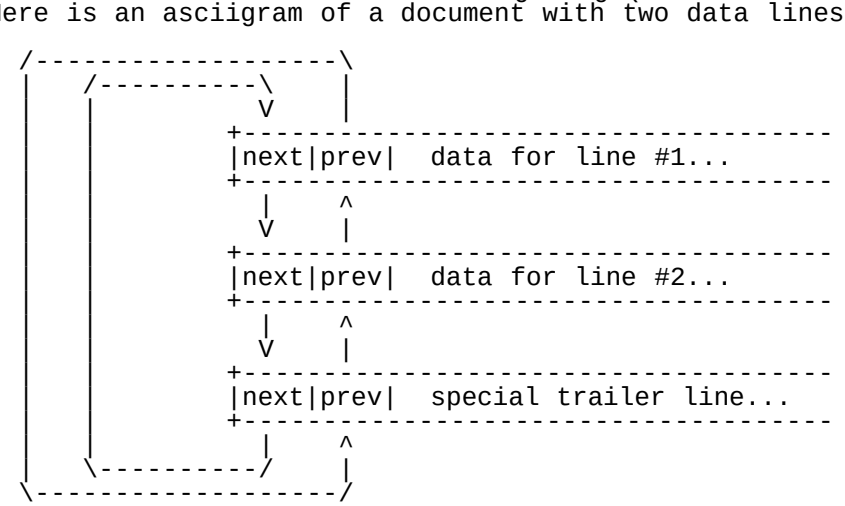
However, the ACE version supports extremely long physical lines (paragraphs) by providing automatic word wrapping and "soft returns" with automatic "text sloshing" (a dynamic form of "juggling"), and it therefore has/will have the functionality of a word processor. The new version also works in all video modes that ACE supports on both the C128 and C64.

#(A)2: 2. DATA STRUCTURES, FILE LOADING

Now we start talking about the implementation of ZED. What a text editor is, basically, is a program that holds a text document in memory that allows you to use certain commands to alter tiny pieces or large-scale chunks of the document and then to save the changed document permanently back to disk. The way in which the document is held in memory is the content of this section.

#(A)2.1: 2.1. DOCUMENT DATA STRUCTURE

ZED uses a bi-directionally linked list to hold the text document in memory. A special "trailer" line is used (which is displayed with a little "house" character on it) to make modifications to the linked list and the whole list is in the form of a big ring (the links run around in circles). Here is an asciigram of a document with two data lines:



I should mention that all pointers are 32-bit values, so that they can point to anywhere in ACE's "far" memory. (In fact, many of the control variables for ZED are 32 bits in size, to avoid all arbitrary restrictions on the magnitudes of various things). And, despite where the arrows point in the diagram, the value that is stored for the pointer is the address of the starting byte of the record that is being pointed to.

I should also mention that lines are stored as they are displayed. If one physical line (terminated by a carriage return) has to be split (using "soft returns") over multiple display lines, then each _display_ line takes up one linked-record position in the document data structure.

Using a bi-directionally (doubly) linked list (instead of a uni-directionaly (singly) linked list is a practical necessity in this environment for two reasons. First, the linking represents the natural way that the user accesses the document: he normally moves cursor up and down, page up and down. It would take a lot of time to move the cursor up using a singly linked list. Second, using a double linked list makes it easier to insert or delete a line from the document. You need to modify the previous record to point beyond the record being deleted, but the previous record is difficult to locate in a singly linked list; you must keep and manage a pointer to it (and if the user moves the cursor up, you lose what you've got for it). I prefer doubly liked lists for any job anyway, even if the ability to go backwards isn't needed, because they are easier to work with (and to construct reusable library routines for).

Using a large block of memory for the data isn't really an option either. Anyone who has used SpeedScript knows what happens when you try to insert text near the beginning of a long document: it takes a lot of time to insert one space in the document memory, and the delay is annoying. Imagine this played out for a document that is N megabytes in size. (Some auxiliary data structure would be needed in this case anyway, to break the 64K barrier). I'm guessing that most other word processors/text editors for Commodore computers use this data structure (a "large" block of memory).

The decision to store each display line in a single record is really one of convenience and efficiency. Most of the time, the use will be positioning the cursor and whizzing between display pages, so we don't want to be wasting any time uselessly re-formatting the text while he (sic) is doing this. We only want to re-format the text when an actual modification is made to the document. This organization does have the ugly implication that physical and logical line numbers may not always match up, but if the "target" line length (the maximum length that a single display line can be) is set to be longer than the maximum physical-line length of a file (often 80 characters), then the two will match up.

Accessing the document is fairly simple. All that we need to locate the entire document is the address of the special trailer line. With this, we know directly where the bottom line is, so we can instantly go to the bottom of the document (Commodore-DOWN) and then follow the links backward to access preceeding lines. Finding the top of the document is also quite easy since the document is in a ring. We just locate the trailer line and then follow the "next" link, and we arrive at the first line (Commodore-UP). It should be no surprise that a pointer is kept to the trailer line in ZED in order to locate a document. The design allows for many documents to be held in memory at the same time, including the "kill buffer" (which is logically a complete and independent document). To make management even simpler, it should be noted that this trailer line never changes (therefore, we never have to update the pointer to the trailer line of a document).

#(A)2.2: 2.2. LINE DATA STRUCTURE

The format of each individual display line within a document held in the linked-list structure described above is as follows:

OFF	SIZ	DESC
0	4	pointer to the next line
4	4	pointer to the previous line
8	1	flags for the line, including $80=hard-return, $40=trailer line
9	1	number of characters on the line
10	n	the displayable characters of the line
n+10	-	SIZE

The two 32-bit pointers have already been mentioned. The "flags" field tells whether the line ends in a "hard return" or a "soft return". A hard return (indicated by the $80 bit being set) is recorded for every place in the text file where a carriage-return character is present. A soft return (indicated by the $80 bit being clear) is formatted into the document every place where a line must be broken in order to avoid it exceeding the target line length. If you modify the document, then words can be wrapped and pulled back around a soft return in order to insure that display lines are as full as possible (whereas they cannot be wrapped around a hard return).

When a file is being written back to disk, lines that have a hard return flag will be written with a trailing carriage-return character, whereas lines ending with a soft return will be written with only the characters displayed on the line (no CR), and, as such, they will be logically concatenated together into the same physical line (again) in the output file. The "trailer line" flag indicates whether the current line is the special trailer line of the document or not. We need a convenient way to check for running into the trailer line, and we cannot use null pointers since the document is in a ring (note that there won't be any null pointers even if the document contains zero data lines; the trailer line will point to itself).

The lower six bits of the flags field are currently unused, but they could be used, for example, to record the number of leading spaces that a line has before the first non-blank character. This would allow us to hold a file with a lot of indentation (a program, for example) using less memory per line. This feature is not currently implemented.

The next field tells the number of displayable characters that are on the line and then the next field stores the actual characters in a simple string. If the line ends in a hard return, then the carriage-return character is NOT stored in the line data, since its presence is already indicated in the line header. When records are allocated in the dynamic-memory space, only the number of bytes that are actually needed are allocated. This is the number of bytes on the line plus ten bytes for the line header. Actually, the number of bytes reserved for an allocation is the number of bytes requested rounded up to the nearest multiple of eight bytes, for technical reasons discussed in C= Hacking #2. A single display line can contain up to 240 characters (not counting the CR).

Every time that a line needs to be accessed, it must be fetched from far memory into a buffer in the program space. There is a slight efficiency problem in accessing a line that there isn't when allocating and storing a line. When going to read the line, you don't know how big it is, since you know nothing about it other than its far location. The conservative thing to do would be to read the first ten bytes of the line record (the header information) and then use the value in the line-length field to figure out how many more bytes you need to fetch. I kind of took a wild stab and made it so that I read the first twenty bytes of a line and then see if the line has ten or fewer displayable characters on it. If so, then I have successfully fetched the whole line and I am done in one access. (Note that there is no problem with fetching far memory beyond one record's allocation

(although there certainly would be a problem with stashing)). If not, then I fetch the remaining characters and I am done in two fetches (actually, I fetch all of the characters for simplicity).

However, often times I don't actually have to fetch the data of a line at all and I only need to access its header information (to follow or change its linkage, for example). In this case, I only have to access the header of the line and I only need one access to get it since I already know how long the header is (ten bytes). I can also write back a modified header in place since it is of fixed length. If I were to, for example, add characters to a line, then I would have to re-allocate a larger line record for it, free the old line-record storage, and link the new line record in with the rest of the document (by updating the previous record's "next" pointer and updating the next record's "prev" pointer and by updating any necessary global variables... quite a bit or work).

#(A)2.3: 2.3. GLOBAL VARIABLES

This section describes all of the global variables that ZED keeps in order to edit a document. First, I have three separate temporary-storage work areas:

```
work1 = $02 ;(16)  ;used by malloc
work2 = $12 ;(16)  ;used by file-load
work3 = $22 ;(14)
```

Each work area is used by successively higher levels of software in order to avoid conflicts between layers. For example, "work1" is used by the dynamic-memory routines and "work2" gets modified when loading a file. The process of loading a file involves a lot of memory-allocation work, so it is good that they use separate working storage and don't clobber each other.

The following variables are used for managing the screen and the current cursor column:

```
scrTopAddr      = $30 ;(2)  ;screen address of the top line on the display
scrRow          = $34 ;(1)  ;row number of the current line on the display
scrCol          = $35 ;(1)  ;virtual screen column number of cursor position
scrRows         = $36 ;(1)  ;number of rows on the display
scrCols         = $37 ;(1)  ;number of columns on the display
scrStartRow:    .buf 1      ;starting row on the display
scrStartCol:    .buf 1      ;starting column on the display
scrRowInc       = $38 ;(1)  ;row increment for the display
scrLeftMargin   = $39 ;(1)  ;left margin for displaying lines
statusMargin:   .buf 1      ;left margin of the status line on the display
conColor:       .buf 8      ;color palette
```

Most of these fields are used for interfacing with ACE's direct-access full-screen-control calls. The ones that are used most often are allocated to zero-page locations (to reduce code size and to increase performance) and the others are allocated to absolute memory. ACE allows application programs to use zero-page locations $02 to $7F for their own purposes.

The current displayed cursor location is stored in "scrRow" and "scrCol". "scrRow" is the current physical display row of the current document line (where display rows start at 2 since the control and separator lines take up rows 0 and 1), and "scrCol" tells the current position on the current line, from 0 up to the length of the line. Since this version of ZED features horizontal scrolling to handle really long display lines, "scrLeftMargin" is also maintained to tell what the column number of the left margin of the display is. When we refresh the screen, we will display all lines starting from this character position. Note that internally, column numbers start from 0 whereas they are numbered starting from 1 in all dialogue with the ape at the keyboard. Every time that the cursor could possibly move off the right or left edge of the screen, a check is made and if this happens, then the "scrLeftMargin" is adjusted and the screen is re-painted (effectively giving us horizontal scrolling).

The following variables are used to keep track of various parameters:

```
targetLen       = $3a ;(1)  ;length to display lines
wrapFlag        = $3b ;(1)  ;$80=wrap,$40=showCR
modified        = $3c ;(1)  ;$00=no, $ff=modified
modeFlags       = $3d ;(1)  ;$80=insert, $40=indent
statusUpdate    = $3e ;(1) ;128=line,64=col,32=mod,16=ins,8=byt,4=fre,2=nm,1=msg
markedLinePtr:  .buf 4      ;line that is marked, NULL of none
markedLineNum:  .buf 4      ;line number that is marked
markedCol:      .buf 1      ;column of marked logical line
```

"targetLen" is the length that ZED tries to keep wrappable lines as close to without exceeding. By default, it will be set to the physical display width

of the screen, but it can be set to 240 characters by the "-l" option and it will eventually be manually settable to any value you want (10<=l<=240). The "wrapFlag" tells, first, whether word wrapping should be used ($80 set) or whether lines should just be broken at the target length regardless of whether it gets broken in the middle of a word or not ($80 clear), and second, tells whether carriage-return characters should be displayed to the user ($40 set) or not ($40 clear). (Actually, a suitable character to represent the carriage return is displayed, taken from the graphical palette of the current character set). You will normally want carriage returns displayed when you are using ZED as a word processor, and you will normally want them not displayed when using ZED as a text editor.

The "modeFlags" tell, first, whether auto-insert ($80 bit set) or over-type ($80 clear) mode is in effect, and second, whether auto-indent ($40 set) or no-indent ($40 clear) mode is in effect. Insert/overtype is currently supported and auto-indent is not. Auto-indent is intended to eventually make programming easier by not requiring you to type a bunch of spaces to indent a new line of a source file that should be at the same nesting level as the previous line.

The "statusUpdate" variable is used to reduce the amount of work the needs to be done in order to keep the status line at the top of the screen up to date. If any of the variables that control the values displayed on the status line change, then the corresponding bit in this variable should be set. After processing a keystroke but before waiting for the next keystroke, ZED will update all of the fields that have a '1' bit in this variable. The "msg" bit is special, since it tells whether a dialogue message is currently being displayed on the separator line (between the status line and the first document line), and if there is, then the message should be erased (overwritten by the separator characters) _after_ the user presses the next key (since it wouldn't be much fun if the user had only a couple of milliseconds to read the message).

The "marked*" fields tell where the "mark" is currently set for range commands (like delete). Like in the stand-alone ZED, I will be making this version clear the mark after every modification to the file. The main reason for this is that it would be a pain in the but to keep track of the mark in some cases like when the line that is marked gets deleted or updated, and that having set-mark/do-operation clear the mark prevents the ape at the keyboard from accidentally hitting a range-destroy key and wiping out his document (he will get a "range not set" error message instead).

The following variables are used to keep track of the current position in the document:

```
linePtr         = $40 ;(4)  ;pointer to current line
lineNum         = $44 ;(4)  ;number of current physical line
headLinePtr     = $4c ;(4)  ;pointer to the special header/trailer line
lineCount       = $50 ;(4)  ;number of display lines in buffer
byteCount       = $54 ;(4)  ;number of bytes in buffer
```

"linePtr" always points to the line record that the cursor is logically on. This is probably the most important global variable. This variable is needed so that we know what line to modify/etc. if the user enteres a letter/etc. "lineNum" gives the line number of the "linePtr" line, where line numbers start from 1. This needs to be maintained in order to tell the ape at the keyboard where in the document he is. These two fields are sequentially updated as the user moves from line to line in the document.

"headLinePtr" is a bit of a misnomer since it actually points to the special trailer line. As explained above, it is used to find the top and the bottom of the document. "lineCount" keeps track of the total number of display lines in the current document, and "byteCount", the total number of bytes. Each carriage-return character counts as one byte. Keeping track of line and byte counts is for convenience rather than necessity.

The following variables manage the "kill buffer", where text goes after it's been deleted but before it's completely discarded (a sort of purgatory):

```
killBufHeadPtr: .buf 4      ;pointer to special header/trailer line of kill buf
killLineCount:  .buf 4      ;number of lines in kill buffer
killByteCount:  .buf 4      ;number of bytes in the kill buffer
```

The kill buffer is maintained in exactly the same structure that the main document is: a ring of doubly linked line records. The three fields shown store the pointer to the special trailer line, the number of data lines in the kill buffer, and the number of data bytes in the kill buffer, respectively.

In addition to the kill buffer, there is also a "rub buffer":

```
RUB_BUFFER_SIZE = 50
rubBufPtr:  .buf 1
rubBufSize: .buf 1
rubBuffer:  .bss RUB_BUFFER_SIZE
```

It is used to hold the fifty single characters that have most recently been deleted by using either the DEL or Commodore-DEL (Rub). I found that when using the old version of ZED, I would sometimes unintentionally delete a single character and then want it back, and I had to expend mental effort to figure out what it was. This mechanism takes the effort out of that job by maintaining a LIFO (Last In First Out) (circular) buffer controlled by the variables given above (note that "RUB_BUFFER_SIZE" is a constant which can be easily changed up to 255 in the source code). The "rubBuffer" is actually contained in the uninitialized-storage section of the program ("bss" in Unix terminology). (The ".bss" directive is not currently implemented in the ACEassembler, but I used it here as a shorthand for the equates that replace it).

The Shift-Ctrl-R (rub recall) keystroke is used to recall the previous character (as if you had typed it in) and has the effect of resetting the "rubBufPtr". Then, each additional time that you type Shift-Ctrl-R in a row, the "rubBufPtr" is advanced backward to the character previous to the one just recalled. Pressing anything other than Shift-Ctrl-R resets the "rubBufPtr", so that you could recall the characters again, if you want. You can recall as few characters as you wish.

Interpreting command keys is done with the following global variables:

```
keychar       = $58 ;(1)
keyshift      = $59 ;(1)
sameKeyCount: .buf 1
sameKeyChar:  .byte $00
sameKeyShift: .byte $ff
```

ACE returns both a shift pattern and a key character code, so both are stored. The shift pattern allows us to take different actions for commands like Ctrl-R and Shift-Ctrl-R. The "same*" fields store the previous keystroke and the number of times that the exact keystroke has been made, so that slightly different actions can be taken when the same keystroke is made multiple times, such as with Shift-Ctrl-R (or maybe HOME).

The following global variables are also maintained for various purposes:

```
exitFlag:     .buf 1
arg:          .buf 2
temp:         .buf 4
stringbuf:    .bss 256
filebuf:      .bss 256
tpaFreemap:   .bss 256
linebuf:      .bss 256
line          = linebuf+headLength ;(241)
headBuffer    = $70 ;(10) ;buffer for holding the head of the current line
headNext      = $70 ;(4)  ;pointer to the next line in a document
headPrev      = $74 ;(4)  ;pointer to the prev line in a document
headLineLe    = $78 ;(1)  ;length of the text line
headFlags     = $79 ;(1)  ;$80=CR-end, $40=headerLine, &$3F=indent
headLength    = 10        ;length of the line header
documentBuf:  .bss 256
 docbufNext   = documentBuf+0   ;(4)
 docbufPrev   = documentBuf+4   ;(4)
 docbufInfo   = documentBuf+8   ;(23)
 docbufFilenameLen = documentBuf+31 ;(1)
 docbufFilename= documentBuf+32  ;(224)
```

"exitFlag" is set to tell the main loop to bail out and exit back to the calling program. "arg" is used for scanning the command-line arguments. "temp" is used miscellaneously. "stringbuf" is used for miscellaneous string processing, and "filebuf" is used for miscellaneous file/string processing. "tpaFreemap" is used by the dynamic-memory-management code as a free-memory-page map for making allocations out of the application program area (or TPA, Transient Program Area). The ACE kernel doesn't dynamically allocate pages in the application space (since this cannot normally be done reliably), so a mechanism is needed inside of ZED to make use of this memory.

"linebuf" is the place where the current line is fetched to/stashed from when it is being accessed or modified. "line" is the sub-field of "linebuf" where the actual line data is stored. The "head*" variables are allocated in zeropage and the record-header information from "linebuf" is copied to these variables whenever a line is fetched and copied from these variables when a line is stashed to far memory. Zero page is used for these variables

since they are manipulated all of the time.

Finally, "documentBuf" stores all of the information about the current main document. There is currently support for only one main document implemented, but the design includes the concept of the user being able to switch between an arbitrary number of documents held in memory at any time.

2.4. LOADING A FILE

When ZED is first started up, its usual first job is to load in the document that was named on the command line (plus you can load a file at any time with Ctrl-L). ZED uses the standard ACE "open", "read", and "close" system calls to do this, although there is a bit of business that has to happen to get the data into the internal form that ZED uses. The job is split into a number of routines to make it easier to program.

The main routine opens the file for reading and initializes that variables that the load routine uses. Among other things, the load routine counts up the number of display lines and physical bytes in the file and must wrap physical lines into display lines while reading. Later, this routine will perform on-the-fly translation from other file formats to PETSCII and will perform TAB expansion if requested.

The main routine repeatedly calls subroutines to read a line into the line buffer, wrap it, and store it to memory. For the purpose of the following discussion, we will assume that the "target" line length is 80 characters, although it can be set to anything that you want. The Read routine copies characters from the "filebuf" to the line buffer, with the filebuf being re-filled with file-data characters as necessary in 254-byte chunks (the natural size of Commodore data sectors, for efficiency). Data bytes are copied until either a carriage-return (CR) character is encountered or we reach the 81st character (target+1). We have to check the 81st character because it may be a CR, and if we are in the normal text-editor mode, we can store a full 80 data characters on a display line, even if it ends in a CR (some editors, quite annoyingly, cannot do this). However, if the user selects the mode where CRs are visibly displayed on the screen, then we stop scanning the current display line at a maximum of 80 characters if a CR isn't encountered.

After the characters of the display line have been put into the line buffer in the above step, it may be the case that the word at the end of the line has been abruptly broken in the middle. If the line ended in a CR, then this doesn't need to be checked. If the line didn't end in a CR and if the "wrap" mode is currently on, then an abruptly cut word will have to be wrapped to the next line. To do this, we scan from the end of the line back until we encounter the last space character on the line. Then, we cut the line immediately after that space, and remember where we cut it so that we can later process the overflown characters. If there is no space on the line (i.e., the first and only word on the line is longer than the target length), then we keep it as-is and end up breaking the word abruptly. Note that we break the line with a "soft return", so there is no damage done to the data by word wrapping.

After the fat (if any) has been trimmed off the current display line, we want to store it into far memory, into the doubly linked ring structure discussed above. To do this, we first, set the pointer to the previous line to the address of the previous line record (we keep track of this) and set the pointer to the next line to Null (for now). Then we we allocate memory for the current line and Stash it out. But we're not done yet; we need to set the "next" pointer on the previous line record to point to our newly allocated line. This can be done by simply writing the 32-bit pointer value to the start address of the previous line (there is no need to re-fetch the previous-line contents or header).

And after stashing out the line, we recall where we wrapped it (if we did) and copy the characters that were cut off to the beginning of the line buffer and we pretend that we have fetched these from the file as if in the Load Line step above. Then we go back to the Load Line step and continue.

When loading is finished (after we hit End-Of-File (EOF) on the file being read), we flush the last incomplete line segment, if there was one (a file is not required to end in a CR) and then we cap things off with the special trailer line. This trailer line is allocated before we start loading the file (although I didn't mention it above), and now we set up its links so that our entire file is the nice doubly linked ring that we like so much. Now we are finished, and we return the trailer-line pointer, the number of display lines read in, and the number of physical bytes read in to whomever called us (it could be the command-line parser, the Ctrl-L (Load) command, or the Ctrl-I (Insert) command).

There are three reasons why I like having this special trailer line around.

(1) It allows us to not have to worry about Null pointers. For example, you will notice that while stashing a loaded line, I allocated the trailer line first so that we would always have a "previous" line to link with. (2) It allows the user to move the cursor beyond the physical end of the document to do operations like recall (Ctrl-R) a block of text. This was a problem with the original ZED; you had to go the the end of the file, press RETURN to open up a blank line, recall the text, and then delete the bogus blank line. (3) It allows a document to have zero characters in it in a consistent fashion.

There is also a subtle but complicated issue dealing with dynamic memory allocation that I haven't discussed yet: what if it fails? (I.e., what if we run out of memory?). In this case, we must handle the failure gracefully, maintain the integrity of the document as best we can, and inform the user. In some cases, maintaining the integrity will involve un-doing committed changes to the document, which is a real pain in the butt, but which is still very important. It would be kind of annoying if you just made the last keystroke on five hours of editing work and the program aborted on an "out of memory" error, sending all of your work to the great bit bucket in the sky.

#(A)3: 3. SCREEN DISPLAY, MOVING AROUND

This section discusses the operations that have to do with maintaining the current secion of the document on the display and moving the cursor around within the document.

#(A)3.1: 3.1. PAINTING THE SCREEN

This is the essential operation that keeps the screen up to date with the docment in memory. ZED has a single function that does this operation, and it is called with the arguments: a pointer to the starting line to display, the starting screen line number to start painting at, and the ending screen line number to end painting at, plus one (lots of endings are 'plus one' since this allows me to use the BCC instruction). Some implied arguments include general information about the screen for use with the "aceWin*" kernel functions and the left margin for displaying lines.

This subroutine simply goes through the display lines one by one, displays the line contents, and advances the line pointer. To display a single line, the line is fetched into the line buffer from far memory and the number of displayable characters is calculated according to the line length and the left-hand display margin. The displayable characters (possibly zero) are written to the screen memory and if the line isn't completely full, then the remainder of the line is filled in with space characters. The displaying is completely performed by the "aceWinPut" system call. One oddity that needs to be handled is running out of document lines before filling the screen range. In this case, each remaining screen line is cleared.

Attributes (colors) are not used for this operation, since they are not needed and using them would only slow us down. They display-area color cells are initialized when ZED starts (as are the status-line and separator-line color cells) and don't change during operation. The colors come from the ACE palette (which the user can configure to his own liking).

This subroutine is used to both repaint the entire screen (when necessary) and to repaint only one line or a few lines (when I can get away with this). Repainting takes time, so we want to repaint only what has changed. However, repainting isn't too slow, especially when compared to serial-line speeds, since screen updates are written directly to screen memory, although the C64's soft-80 screen is significantly slower than the other screen types since so much more processor work needs to be done just to make a single character appear.

In the future, it may be useful to allow this function to abort in the middle of this operation if the user presses a key before the repainting is finished, in order to allow the user to work faster. For example, if you hold down the Page-Down keystroke, the speed that you go forward in the document is limited by how fast the screen can be repainted. If the repaint operation were abortable, then you could always go forward as fast as the key repeates, and when you get to where you are going and release the Page-Down key, the screen would repaint one final time and everything would be consistent. A flag would need to be kept to tell whether the screen is consistent or not, in order to make this work.

There is also a function that displays a message in the separator line on the screen. It also needs to store the displayed message in order to allow the user to scroll through it if the screen is not wide enough to display it in its entirety. When the message is no longer needed, it is erased by overwriting it with separator characters. And, there is also a function that updates all of the fields that have changed on the status line.

#(A)3.2: 3.2. MAIN CONTROL

Before I start talking about the implementations of the individual commands, I should say something about the main control for the program. After ZED initializes and loads the initial document (even if you don't specify one, ZED will default to the name "noname" and try to load it), control is passed to a small main loop of simply displaying the cursor, waiting for a keystroke, undisplaying the cursor, calling the subroutine associated with the keystroke, and repeating.

#(A)3.3: 3.3. END UP, DOWN, LEFT, RIGHT

Moving the current line to the top and bottom of the document is straight forward, because of the organization that was discussed in the data-structure secion above. To go to the top of the document, copy the trailer-line pointer to the current line pointer and then fetch the next pointer from the trailer line's header; this will give a pointer to the top line. Then we set the current line number to one and a couple of other variables and call the subroutine discussed above to repaint the screen.

Going to the bottom of the document would be just as easy as going to the top, except that we want the last line (the trailer line) to be displayed on the bottom of the screen in order to present as much useful document content to the user as possible. If there are fewer lines in the file than will fill a screen, then we cannot, of course, display an entire screen.

To make this business easier, a subroutine is provided that, given a starting line pointer and a count, will scan until it either hits the count-th line previous to the given one or it hits the top line of the document. It returns the number of lines that were actually scanned upwards (possibly zero), the pointer to the line that it stopped scanning at, and a flag indicating whether it stopped because it hit the top of the document or not. This subroutine is quite generally useful. There is a similar subroutine that scans downward.

So, after locating the bottom line of the document and setting the line number, the scan-upwards subroutine is called to scan upwards the number of displayable lines on the screen. The screen is then repainted in its entirety from the line that was scanned up to, and the new cursor-display location is computed from the count of the lines that were scanned over. This works equally well for a long document, a document shorter than the height of the screen, and an empty document.

The End-Left and End-Right commands are very simple in that they don't even have to change the current line pointer, but they do have to check if the cursor has moved off either the left or right edge (margin) of the screen. The visible columns go from the column number of the left margin, up to that plus the width of the screen. If the cursor goes off an edge of the screen, then the new left margin will have to be computed and the entire display will need to be repainted.

This repainting effectively achieves horizontal scrolling. Lines in ZED, of course, can be up to 240 characters across (241 if you count the carriage-return character), but the widest screen that ACE supports is 80 columns. Arguably, the horizontal scrolling could be done more efficiently by moving the contents of the display left or right by the requred number of columns and then filling in the opened spaces with the data from the correct columns of the display lines in memory. However, the additional complexity is non-trivial and the speedup may not be all that great except for the soft-80 screen of the C64. A better approach might be to go with the interruptable-repainting idea that I spoke of earlier, if the current line were updated first (so that you can see what you're doing) and the rest of the lines afterwards.

#(A)3.4: 3.4. PAGE UP, DOWN, CURSOR UP, DOWN, LEFT, RIGHT, WORD LEFT & RIGHT

All of these functions follow quite naturally from what is above. For Page-Up and Down, the scan-up or scan-down subroutines already described are called to find the new current line for the cursor and then the entire screen is repainted, effectively paging up or down.

For cursor up and down, we just go up or down to the next line in the document and adjust adjust the cursor location on the screen. If the cursor goes off the top or bottom of the screen, then we scroll the screen up or down as appropriate (ACE can scroll the screen up or down and will eventually be able to scroll it left and right (although this will be a bit painful for the soft-80 screen since it may mean scrolling left and right nybbles)). Then, we display the current line at either the top or bottom of the screen to fill in the blank line that we just opened up. Because we use scrolling and painting only a single line, we can scroll the screen fairly

quickly, easily keeping up with the cursor repeat rate, except on the soft-80 screen.

For cursor left and right, we advance the cursor one position on the line and see if it has gone over the edge. If not, then we are done; nothing needs to be redisplayed. If we have gone off the edge, then we call the cursor-up or cursor-down routines to go to the previous/next line and we position the cursor to either the end or start of the new line.

The word left/right functions are similar to the cursor left/right functions, except that we keep scanning until we run into the start of the next word. For word left, this is defined as running into a non-whitespace character that is preceeded by a whitespace character. A whitespace character is defined as either a space, a TAB, a hard return, or the beginning or ending of the document. For word right, the start of the next word is defined as a non-whitespace character that is preceeded by a whitespace character, where we start searching from one position to the right of the current cursor position. If we run into the beginning or end of the document, then we stop there.

BTW, all of these moving-around functions check the cursor position against the display bounds and "scroll" the display left or right if the cursor has gone off the screen. Well, actually, there is one exception to this rule. If the current line is the target-length number of characters long, and the target length equals the screen width, and carriage returns are selected not to be displayed, and the left margin of the display is column one (external), and the cursor is in the target-length-plus-one position of the line, then the screen is NOT scrolled right. Instead, the cursor is displayed on the last position of the line and is made to blink fast (an ACE feature). This is done to avoid the annoyance of having the screen scroll right when you are editing a text file on, say, an 80-column screen that has up to 80-character lines in it. The standalone ACE does this too, when you logically hit the 81st column.

#(A)4: 4. TEXT INPUT AND "SLOSHING"

So far, we can load up a document and whiz around inside of it, but we can't actually change anything. This section describes the single-character modification operations of character input, rub, and delete, and the text "sloshing" algorithm that is needed to make sure that lines are always as full as they can be without going over the target length ("Come on down!").

#(A)4.1: 4.1. TEXT INPUT, DELETION

Adding a single character to a document isn't really very difficult. There are two modes for single-character inputting: insert and overtype. Insert mode is generally more useful and more often used, but overtyping can be very useful when dealing with tabular or specially formatted text. Therefore, we must support both modes. In some other text editors, overtype mode is the natural mode because the cost of inserting a character can be so high, but not here.

Actually, there isn't a whole lot of difference in the implementations of the two modes. For overtype mode, you just fetch the current line, take the inputted character and store it into the line buffer at the current position, stash the line back into memory, and repaint the line. For insert mode, we do the same thing, except that we copy the line from the cursor to the end to one position beyond the cursor (backwards) and bump its length up by one before storing the new character on the line.

Rubbing out a character (Commodore-DEL) is done in quite the same way, except that we copy the rest of the line back one space and decrement the length. Oh, and when the length of the new line is different from the length of the old line, we have to deallocate the old line and allocate new memory for the new line, and surgically link it in with the rest of the document. I have a subroutine that does this.

The DEL key is handled as if you had typed Cursor Left then RUB, except when you press DEL on the first column of a line and the previous line ends with a hard return, the hard return of the previous line is removed instead. I decided to make the RUB (Co-DEL) key return an error instead of joining lines together like DEL, because sometimes it is convenient to just lean on the RUB key to delete to the end of a line.

#(A)4.2: 4.2. TEXT SLOSHING

But, we're not done with text modifications yet. If we just left the modifications as described in the previous sections, we would be end up with lines that are longer than the target length, with ragged lines, and with lines that don't join together like they should after pressing DEL in the first column.

After each of the modifications, the text-sloshing routine is called to straighten everything up and figure out how to redisplay the screen. Often, only one line needs to change, but sometimes, many lines or even the whole screen will have to be updated, as sloshing text can continue for many lines as line overflows and underflows cascade forward from the line that has just been modified. In fact, the text-sloshing routine is enormously complicated and has many, many special cases. (Although, for all of its complexity, it is only 400 lines long, although it still needs a few more features).

There are many more cases that could cause sloshing than you might think. There are the obvious inserting/deleting one-too-many characters in the current line, but there is also the case that an insertion or deletion causes a space character to move to the right position on a line to allow the line to be sloshed backward, or maybe you remove a space from the end of a line that creates a word that would be too long to be contained on one line and therefore needs to be sloshed forward.

The sloshing algorithm doesn't introduce or take away any characters from the body of the document; it just reorganizes the existing characters. All of the spaces are retained at the ends of wrapped lines. We don't want to delete spaces, since it is difficult to reconstruct them, since two spaces are normally between two sentences in text, but it is difficult for a computer to figure out where a sentence ends. In fact, keeping these spaces can sometimes cause an anomaly: if all of the spaces won't fit on the end of one line, then they will be displayed at the beginning of the next line.

The variables that are maintained by the algorithm are as follows:

```
sloshLinesAltered  = work2+0 ;(1) ;a simple count
sloshRedisplayAll  = work2+1 ;(1) ;$80=redisplay to bottom, $ff=force all
sloshRedisplayPrev = work2+2 ;(1) ;whether previous line needs to be repainted
sloshMaxChars      = work2+3 ;(1) ;number of chars that can be sloshed
sloshTailChar      = work2+4 ;(1) ;the last char of prev line
sloshTheCr         = work2+5 ;(1) ;whether a CR should be sloshed
sloshLinesInserted = work2+6 ;(1) ;number of new line records created
sloshLinesDeleted  = work2+7 ;(1) ;number of existing line records deleted
sloshCurAltered    = work2+8 ;(1) ;whether the current line has been altered
sloshTerminate     = work2+9 ;(1) ;flag to terminate (hit a hard return)
sloshCurTailChar   = work2+10 ;(1);last char of current line
```

The algorithm has a main loop that is repeated for each line that can be sloshed. The loop exits when we either run into a hard return or we run into a line that does not need to be sloshed. We start scanning from the cursor line of the main document.

We first look at the previous line and see how many more characters it can accommodate before being full. Then, we scan the current line from this point (the number of characters that could potentially be sloshed backwards) back to the start of the line searching for spaces. If there are no spaces, then the current line cannot be sloshed back onto the previous line. If we do run into a space, then we stop searching and know that we can (and must) slosh back the current line. So, we remove the characters from the current line and write it back (maintaining links as appropriate) and then go back to the previous line and append these characters to it.

If the cursor happened to be on the line in a position that got sloshed back, then we must adjust the cursor position and move it back to the previous line. If the previous line is before the start of the screen, we must set the flag to redisplay the entire screen later. If the current line is the first line of the slosh area but the cursor didn't get moved back to the previous line, then we must set the flag to indicate that we must redisplay the previous line too when we repaint the screen.

If it turns out that we have sloshed back ALL of the characters on the current line, then we must remove the empty line record of the current line from the document and adjust the global line count. We also have to worry about sloshing back a hard return, and if we do, then we bail out of the algorithm since we are done. Oh, and we have to keep in mind whether we are displaying carriage returns or not in calculating line lengths.

After shoshing backward, we check if the current line needs to be sloshed forward. It needs to be sloshed forward if it is either longer than the target length or it ends in a non-space character and the first word on the next line cannot be sloshed back. The latter is a special case and needs to be checked for specially, even thought the functionality for doing so is redundant.

If we do need to slosh forward, we start scanning the current line at the target-line-length point and scan backwards until we hit the first space. If there is no space, then the current word is longer than the target length

and must remain abruptly broken over two (or more) lines. We wrap it at the target length. If we do find a space earlier on the line, then we wrap the line right after that space. Oh, I forgot to mention: we need to do something special for lines that end with non-spaces for backward sloshing too. If the previous line ends in a non-space (presumably because it contains a single very long word), then we don't find any spaces to slosh back to the end of the word, then we slosh back as many characters as will fit, since the word is broken anyway, and we want it to have as many characters as possible on a single line.

To wrap the line, we set the length of the current line to the new length and replace the old version of the line in memory. Then, we create a new line record and store the characters that were wrapped in it and link this line record in with the rest of the document. And this is all that we do here; we don't actually insert the wrapped characters into the next line, since that would be more complicated, and since it might cause that line to overflow. If we just leave the wrapped characters, they will be joined with the next line (if necessary) on the next iteration of the main sloshing loop in a slosh-back operation. We must adjust the cursor location, like before, if the cursor was on the part of the line that got wrapped around.

At the end of a loop, we check to see if we have passed a hard return in the document, in which case, we exit. Otherwise, if either the current line has been modified or if the current line is the first line to be sloshed, then we go on to the next line and repeat the above procedure.

On our way out, we do a little fine tuning of the "sloshLinesAltered", "sloshRedisplayAll", and "sloshRedisplayPrev" variables, which were described earlier. These variables will be used to repaint the changed portions of the screen display. Part of the fine adjustment includes comparing the number of lines that have been inserted and deleted from the document. If these two numbers match, then the bottom portion of the screen doesn't have to be repainted, only the altered lines themselves; otherwise, we need to repaint from the current line all the way to the bottom of the screen.

The sloshing algorithm currently does not handle non-wrap mode; lines will always be word wrapped. Later, all lines will be broken at the N-th column if you are not in word-wrap mode. Also, the algorithm can produce an anomalous wrapping in one case involving lines that end with non-spaces that I can't seem to remember that this algorithm will fail in (but, the document will still be interally consistent). And finally, if you change the target line length while editing a document (which you can't currently do), then the algorithm may not be able to give optimal word wrapping in all cases (though, again, the document will always be interally consistent).

#(A)5: 5. OTHER FEATURES

This secion discusses the other features of the editor that have not been described yet. In general, the operation of this version follows closely from the standalone version, so you can read its documentation for more details. First, I will give a summary of all of the implemented and planned commands, and then I will discuss the operation of a few selected commands.

#(A)5.1: 5.1. COMMAND SUMMARY

Here is a command summary. The "I" column in this list tells whether the feature is currently implemented or not. A blank means "no", and an asterisk means "yes". Note that "currently" means "by the time that you read this" (which will be a couple of weeks after I have written this). The "CODE" column tells the internal ACE-PETSCII code for the key. A plus symbol following it means that the shift status of the key is checked to distinguish this key. The "KEY" column tells what keystroke you must make ("CT-" means Ctrl, "CO-" means Commodore, "SH-" means Shift, and "AL-" means Alt). The "ACTION" column tells you what happens.

I	CODE	KEY	ACTION
	$e0	CT-@	Exchange cursor position with mark position
*	$e1	CT-A	Alter case of letter under cursor
	$e2	CT-B	Go on to next document buffer
	$e2+	SH-CT-B	Go on to previous document buffer
	$e3	CT-C	Copy range
*	$e4	CT-D	Delete range
*	$e5	CT-E	Exit with save
*	$e6	CT-F	Find next occurrence of hunt string
	$e6+	SH-CT-F	Find previous occurrence of hunt string
	$e7	CT-G	Go to given line number
	$e7+	SH-CT-G	Go to given _physical_ line number
*	$e8	CT-H	Set Hunt string

```
* $e9   CT-I     Insert new file into current one
  $ea   CT-J     Juggle the lines of paragraphs, keep separate
  $eb   CT-K     Kill current line
* $ec   CT-L     Load file
  $ed   CT-M     Set Mark for range operations
* $ee   CT-N     Set Name of current file
  $ef   CT-O     Set Options: input/output translation/tab-expansion, etc.
  $f0   CT-P     Print current file
* $f1   CT-Q     Quit without save
* $f2   CT-R     Recall text from the Kill buffer
* $f2+  SH-CT-R  Recall text from the Rub buffer
* $f3   CT-S     Save file
  $f4   CT-T     Tie together multiple lines into one big line (paragraph)
  $f5   CT-U     Undo the last change made to the document
  $f6   CT-V     Verify file
  $f7   CT-W     Write range with new name
  $f8   CT-X     Extract the individual lines from a paragraph
  $f9   CT-Y     Replace (all the other letters were taken!)
* $fa   CT-Z     Goto bottom of screen
* $fb   CT-[     Toggle insert mode
* $fc   CT-\     Toggle modified flag
* $fd   CT-]     Toggle indent mode
  $fe   CT-^     Change the current working directory
  $ff   CT-_     Compose ISO-8859-1 character

I CODE  KEY          ACTION
- ----  ---          ------
* $91   UP           Cursor up
* $11   DOWN         Cursor down
* $9d   LEFT         Cursor left
* $1d   RIGHT        Cursor right
* $06   SH-LEFT      Word left
* $0b   SH-RIGHT     Word right
* $16   CT-UP        Page up
* $17   CT-DOWN      Page down
* $19   CT-LEFT      Page left
* $1a   CT-RIGHT     Page right
* $0c   CO-UP        Goto top of document
* $0f   CO-DOWN      Goto bottom of document
* $10   CO-LEFT      Goto beginning of line
* $15   CO-RIGHT     Goto end of line

* $0d   RETURN       Split current line (indent not yet implemented)
  $8d   SH-RETURN    Go to next paragraph
  $01   CT-RETURN    Go up one paragraph
  $09   TAB          Tab
  $02   SH-TAB       Backtab
  $18   CT-TAB       Insert to next tab stop
* $14   DEL          Delete character
* $08   CO-DEL       Rubout
* $94   INST         Insert one space
* $13   HOME         <nothing>
* $93   CLR          Cursor home
  $04   HELP         Bring up help window
  $84   SH-HELP      Display help screen
* $0a   LINEFEED     <nothing>
* $07   SH-LINEFEED  <nothing>
* $1b   ESCAPE       Redisplay screen
* $0e   SH-ESCAPE    <nothing>
* $03   STOP         <stop some operations>
* $83   RUN          <nothing>

  $90   CT-1         Clear document
  $05   CT-2         Clear buffer
  $1c   CT-3         Enter hexadecimal PETSCII character code
  $9f   CT-4         Display directory
  $9c   CT-5         Destroy current document buffer
  $1e   CT-6         Create new document buffer
  $1f   CT-7         Display PETSCII code of current character
* $9e   CT-8         Scroll left margin of status line
* $12   CT-9         Reverse screen on
* $92   CT-0         Screen reverse off

  $81   CO-1         Set display to show single buffer
  $95   CO-2         Set display to show two buffers
  $96   CO-3         Set display to show three buffers
  $97   CO-4         Set display to 40 columns, default rows
  $98   CO-5         Set display to take full screen
  $99   CO-6         Set display to default number of rows
  $9a   CO-7         Set display to maximum number of rows
  $9b   CO-8         Set display to 80 columns, default rows
```

```
  $85   F1          Function key 1  : user-defined string
  $89   SH-F1       Function key 2  : user-defined string
  $86   F3          Function key 3  : user-defined string
  $8a   SH-F3       Function key 4  : user-defined string
  $87   F5          Function key 5  : user-defined string
  $8b   SH-F5       Function key 6  : user-defined string
  $88   F7          Function key 7  : user-defined string
  $8c   SH-F7       Function key 8  : user-defined string
  $80   CT-F1       Function key 9  : user-defined string
  $82   CT-F3       Function key 10 : user-defined string
  $8e   CT-F5       Function key 11 : user-defined string
  $8f   CT-F7       Function key 12 : user-defined string
```

#(A)5.2: 5.2. TEXT SAVE

This function is, of course, implemented, since the text-modification functions of the editor would be useless without it. It is really quite simple, because of the data structure of the document. First, we try to open the file for writing. If not successful and we get a "file exists" error, then we scratch the old file. Then, we re-open for writing.

To save the file contents, we start at the top line, and fetch each line in turn until we hit the trailer line of the document, at which point we are finished. After fetching the line, we check if it ends with a hard return, and if so, we append the line buffer with a carriage return character and bump up the line length. We then call the ACE "write" primitive with the line buffer as the argument to write out the line. Writing in this size of chunk rather than in single bytes gives ACE the opportunity to carry out this operation as efficiently as it can.

We then close the file and we are done. We display status information to the user during all phases of this operation, and we certainly tell him if anything goes wrong.

#(A)5.3: 5.3. RANGE DELETION & RECALL

Range delete and recall are implemented, since they are very useful for general editing. What will normally happen is that the user will set the mark with Ctrl-M (mark) to one end of the range to be deleted, and then move the cursor to the other end of the range and press Ctrl-D (delete). The text then disappears into the kill buffer and can be recalled any number of times at any point in the document using the Ctrl-R (recall).

One difference between this ZED and the operations mentioned here are "character oriented" rather than "line oriented". So, you can now delete only portions of lines rather than entire lines. You just have to keep in mind that the cursor is logically located "between" the previous character and the character that the cursor is currently over. For example, if the cursor was on the "y" in "xyz", then the mark would be set to between the "x" and "y" if you pressed Ctrl-M at that point. This also means that if you wanted to delete an entire line (that ended with a hard return), then you would move the cursor to the first character of the line and press Ctrl-M and then move the cursor to the first character of the NEXT line and press Ctrl-D. (The Hard Return itself won't be included in the delete operation if you move the cursor to the end of the line to be deleted--this is one of the reasons for having a displayable trailer line).

To implement the delete operation, all of the lines in the operation are unlinked from the main document and are linked into the kill buffer. If there already was something in the kill buffer, then it is deallocated and the kill buffer is cleared. A trailer line is permanently allocated to the kill buffer, to make it work consistently with the main document. Partial lines (potentially, the first and last lines of the range) are a bit of a pain and have to be split into two lines at the point of the mark/cursor, where one of the broken lines stays with the document and the other goes into the kill buffer. After extracting the range, the lines around the extracted region are sewn back together ("sponge, nurse!") and text is "sloshed" about the stitch point (if necessary). The number of bytes and lines involved are counted up and are subtracted from the global counts for the main document.

Ctrl-C (copy) is very similar to the delete operation, except that the data to be deleted is actually copied to the kill buffer and the document is left unmodified. Range copy is not currently implemented, since its operation can be emulated with a Ctrl-D followed immediately by a Ctrl-R.

To implement the recall operation, the kill buffer is replicated into a temporary document and the current line of the main document is broken into two lines at the recall point (if necessary). Then, the temporary document is linked into the main document at the recall point and the text is

"sloshed" about the two stitch points. The line and byte counts are adjusted, and we are done.

#(A)5.4: 5.4. TEXT SEARCHING

Forward text searching is implemented, since it is very useful for both finding things and for moving quickly around in a document. (Reverse search and Range search and replace are not currently implemented, since they are less useful).

The implementation is quite straightforward. The user will first use Ctrl-H (hunt-string) to set the string to search for. The user will input this on the top line of the screen, and we don't have to do much work for inputting the string, since ACE already provides a console-input routine complete with left/right cursor movement and a scroll-back buffer (although it is a bit hard to use if the input line is longer than the input window on the screen).

After the search string is set, the user will press Ctrl-F (find) to find the next occurrence of the string. So, we just search for that string, starting at the cursor position to the right of the current position. A simple algorithm of keeping a pointer to the current scan position in the hunt string and pointers to both the current position in the document and to the position in the document corresponding to the start of the string is used.

If the current document character matches the current hunt-string character, then both the document and hunt-string pointers are advanced. If the hunt string is exhausted by this, then we have found a match and can stop searching. We move the cursor to the saved document position of the start of the hunt string and exit. If the characters don't match, then we move the current document pointer back to the postition corresponding to the start of the hunt string, advance it by one, save it, and start searching again. Our algorithm needs to be able to wrap around soft returns in the main document.

#(A)6: 6. CONCLUSION

So, here we finally have the basic ACE version of the ZED text editor that I have been promising for a very long time. The new version doesn't contain all of the features of the standalone version, but I am working on it. The new version does, however, include a few features that the old version does not, like long lines, horizontal scrolling, text "sloshing", the ability to use additional memory types for storage, the ability to work on the C64, integration with a command-line environment, and full assembler-code availability (Real Soon Now(TM)).

In order to make ZED operational as a word processor, some means of giving embedded commands and for formatting and printing these commands must be provided. I was originally thinking that an EasyScript or SpeedScript kind of embedded-command structure, and then I was thinking abouta LaTeX kind of structure (the LaTeX structure is superior), but I am now thinking that an HTML type of format-command structure might be rather apropos. Why then I would need to create a print formatter and previewer that might be usable for other purposes, too.

==========

#(@)trivia: Commodore Trivia
by Jim Brain (brain@mail.msen.com)

#(A): Introduction

Well, the cold has moved in on us in Michigan, but the Commodore information coming into the house is keeping us warm. Some orphan computers have showed up, including a Commodore 65 and C116, as well as a couple of Commodore B-128 computers with all the fixin's. So, armed with the hardware, I have come up with some brain ticklers for the Commodore know-it-all.

As some may know, these questions are part of a contest held each month on the Internet, in which the winner receives a donated prize. I encourage those who can received the newest editions of trivia to enter the contest.

This article contains the questions and answers for trivia editions #19-22, with questions for the current contest, #23.

If you wish, you can subscribe to the trivia mailing list and receive the newest editions of the trivia via Internet email. To add your name to the list, please mail a message:

To: brain@mail.msen.com

Subject: MAILSERV
Body:
subscribe trivia Firstname Lastname
help
quit

#(A): Trivia Questions

Q $120) What is the model number of the assembler/monitor for the KIM-1?

A $120) The KIM-5 was the model number of the editor/assembler product.

Q $121) How many LEDs are on the KIM-1?

A $121) The basic unit contains 6 7-segment LED displays, or 42 LEDs if you count each LED in a segment.

Q $122) What is the model number of the REC chip used in the REU?

A $122) MOS 8726.

Q $123) At least two versions of the above chip exist. What is the main physical difference between the versions?

A $123) The eraly versio of the chip (8726-R1) exists in DIP form, while the 8726-r4-r8 exists as a "J-lead" square surface mount unit.

Q $124) Why couldn't regular Atari(tm) style joystcks be used with the Commodore Plus/4 series?

A $124) Instead of using the de-facto 9 pin D-subminuture connector for the joysticks, the Plus/4 series used small mini-DIN connectors. Some sources claim the older connectors were leaking a fair bit of radio interference and were preventing the units from attaining FCC approval, so the connectors were changed to the better-shielded mini-DIN types.

Q $125) What was the first joystick model Commodore produced that would function with the Plus/4 computer line?

A $125) The Commodore T-1341 Joystick, which had the special mini-DIN connector

Q $126) How many computer models are included in the Plus/4 line?

A $126) At last count, 3 models in the Plus/4 series were produced:

The Commodore Plus/4
The Commodore 16
The Commodore 116

Some Commodore 264 models are known to exist, but are not counted, since the 264 was the prototype model of the Plus/4. Also, a V364 model was planned, but only one unit is known to exist.

Q $127) In a normal Commodore disk drive Directory Entry, what relative offset denotes the start of the program name?

A $127) The filename starts at the 4th byte in the directory entry.

Q $128) How many tracks in a 1541 or 4040 are normally available for use as storage?

A $128) 35 tracks.

Q $129) How many bytes comprise a single disk drive directory entry?

A $129) 30 bytes.

Q $12A) What is the model number of the Commodore dual drive with a total capacity per unit of 2.12MB?

A $12A) The Commodore 8250 or 8250LP dual disk drive.

Q $12B) On the drive denoted in $12A, how large could a single sequential file be?

A $12B) 1.025 megabytes.

Q $12C) At least two version of the Commodore 64C keyboard exist. What is the difference between them? Extra Credit: Why?

A $12C) One one keyboard style, the Commodore graphics are printed on the front of the keys, while they appear above the letters on the keys in the second type of keyboard. I can't answer the extra credit part except to say that Commodore was always seeking the best deal. aybe a new keyboard manufacturer got the bid and changed the layout.

Q $12D) On the Commodore 64, what area of memory is swapped out when using an REU with RamDos?

A $12D) $6000 - $7fff is swapped out when a RAMDOS command is executing.

Q $12E) Commodore manufactured two different versions of the 1541-II drive. What is the difference between them?

A $12E) The drive mechanisms differ in the two drives. You can tell which you have by the appearance of the front of the drive. If the lever hits a rest in the release position, you have the direct drive model. If the lever has no such rest visible, the drive cotains the belt drive mechanism.

Q $12F) How many colors could the Commodore 1520 plotter plot in?

A $12F) 4. red, black, blue, and green.

Q $130) The Commodore Plus/4 was referred to as the "_____ Machine".

A $130) Productivity.

Q $131) Although the Commodore 16 and 116 were functionally equivalent, what two physical characteristics distinguished one from another?

A $131) Case style and keyboard. The C16 is enclosed in a VIC20/C64 style case with keyboard, while the C116 sports a scaled down Plus/4 style case and "chicklet" keyboard.

Q $132) How many pins are there on the Commodore plus/4 expansion port connector?

A $132) 50 pins.

Q $133) On which side of the Commodore 65 (as it is facing you) did Commodore place the power switch on?

A $133) The left side. Since the disk drive fills the entire right side, the left side is an obvious choice, as the swith would require cabling if installed on the right side.

Q $134) How many keys are on a standard Commodore 128 keyboard?

A $134) 92 keys.

Q $135) What color are the drive LEDs on the SX64 drive?

A $135) There is only one LED, a red in-use LED.

Q $136) True or False? The Commodore 64 and VIC-20 keyboards are interchangeable.

A $136) True.

Q $137) On a 1526/MPS 802 printer, how many redefinable characters were available for use per line of text?

A $137) 1. True fact: In order to print a line of graphics, one must print a GFX char, do a returb without linefeed (resets the graphic character, evidently), the tab over and repeat the cycle until 80 characters were printed. I had one, and it took me 7 hours to print 21 pages of GEOWrite text!

Q $138) To set up a redefinable character on the MPS 802/1526 printer, what secondary address must be opened?

A $138) Secondary address #5.

Q $139) How many pins are in each Euro-DIN plug used on the Plus/4-C16 joysticks?

A $139) 8 pins.

Q $13A) How many pins are on a regular Commodore VIC-20/C64 joystick

connector?

A $13A) 9 pins.

Q $13B) What BASIC command is used to change from C128 mode to C64 mode on a C128?

A $13B) go 64. It will ask for confirmation.

Q $13C) What were the four integrated programs included in the infamous "3+1" software in the Plus/4?

A $13C) A word processor, spreadsheet, graphics software, and a data management program.

Q $13D) Which Commodore serial printer(s) had a small switch that allowed it to be addressed as either device 4 or device 5?

A $13D) The 1525, MPS 801, and MPS 803 had such a switch. Although I cannot confirm this, I believe the 1515, the precursor to the 1525, also had the 4/5 switch.

Q $13E) How many addressable registers does the Commodore VIC-II IC have?

A $13E) There are 47 control registers in the Commodore VIC-II chip.

Q $13F) On a Commodore PET machine, what output appears on the screen after typing in SAVE "",2?

A $13F) PRESS PLAY AND RECORD ON TAPE #2

Q $140) What was the model number of the microprocessor used in the first of the Commodore 264 Series?

A $140) The early Plus/4 units contained a 7501 microprocessor, and the later units featured a 8501 microprocessor. The only differences between the two units is the manufacturing process and die size.

Q $141) How fast could the microprocessor in the Commodore 264 Series theoretically run at?

A $141) 1.76 MHz.

Q $142) How many colors can a Commodore Plus/4 display at once?

A $142) 8 shades each of 16 colors, but the 8 shades of black are still still black, so a total of 121 colors are possible.

Q $143) What anomaly exists in the numbering of the BASIC interpreter in the Plus/4 as 3.5?

A $143) This version contained almost all of the commands in Version 4.0, plus some new commands for graphics and sound.

Q $144) After the very first 1581 disk drives were introduced, Commodore found that the WD1770 disk controller chip in the drive could corrupt the disk in some situations. So, Commodore offered a replacement IC to fix the problem. What was the number of the replacement IC?

A $144) The Western Digital WD1772 IC.

Q $145) On some very early CBM 1541 drives, what would happen if the serial bus CLOCK and DATA lines were high upon startup?

A $145) On the very first 1541 drives (I suspect the feature was also on the 1540 as well), On power-up, the drive would jump to a subroutine at $E780 after performing the reset routine. The code there would check for the high state of CLOCK and DATA. If found, the code would wait until both go low and then store '*' into the filename buffer, sets the filename length to 1, and then jumps to the & command, which loads a USR file and executes it.

Since the Commodore computer never used this feature, and some machines would boot with these lines randomly high, Commodore removed the feature.

Q $146) In question $0F8, we learned that one must DIMension an array in BASIC if it will have more than 11 elements. Which Commodore produced reference book ncorrectly claims the need to DIMension arrays for more than 10 elements.

A $146) The Commodore 128 Programmer's Reference Guide. Page 17.

Q $147) Why should serial device number 31 not be used?

A $147) While it is specified as a valid serial bus address, when "or"ed with certain commands, it results in a bad command, hanging the bus and the serial drivers.

Q $148) On most VIC game cartridges from VIC-1910 up, toggling interlaced screen display can be done with a keypress. Which key?

A $148) Press the F7 function key.

Q $149) Which cartidge fitting the criteria in $148 does not toggle interlace display with the same keypress as the others? How is it toggled on this cartridge?

A $149) Gorf, VIC-1923. Pushing the joystick up toggles interlace mode.

Q $14A) The Commodore 64 KERNAL and BASIC code use every opcode in the 6510 CPU except three. Which three?

A $14A) BRK, CLV, and SED.

Q $14B) For what purpose does the BASIC interpreter in a Commodore 64 require the Complex Interface Adaptor (CIA) IC?

A $14B) In order to calculate random values for the BASIC function RND(0), the first 4 registers of the CIA whose address is provided by the IOBASE KERNAL routine are read.

Q $14C) On the Commodore 128, the 80 column output is output by the VDC chip. What does VDC stand for?

A $14C) Video Display Controller.

Q $14D) By now, most people know about the ill-fated Commodore 65. What were the specifications on the original Commodore 65 idea?

A $14D) A Commodore C64C with a built-in 1581.

Q $14E) When referring to the Commodore 4032, one usually states that one has a "thin 40" or a "fat 40". What does "thin" and "fat" signify?

A $14E) A "thin 40" had a 9" screen and could not be upgraded. The "fat 40" had a 12" screen, and could be upgraded to a 8000 series machine with some upgrade chips.

Q $14F) If you own a Commodore 4032, how can you tell which kind (thin or fat) you have?

A $14F) If you hold down the cursor key and it repeats, you have a "fat 40". (Of course, inspection could also be used, as the "thin" unit had a smaller screen)

Q $150) How nmany keys are on a standard Commodore B-128 keyboard?

A $150) 94 keys.

Q $151) How many revisions of the 1541 printed circuit board are known to exist?

A $151) For the 1541:

```
PCB# 1540001    The "long board", as used in the 1540.
PCB# 1540008-01 Minor revisions to the 1540001 board.
PCB# 1540048    The "short board".
PCB# 1540050    Minor revisions to the 1540048 board.
                -01   ALPS mechanism
                -03   Newtronics mechanism
PCB# 250442-01  A revision of the short board. 1541 A board
PCB# 250446-01  Minor revisions to the #250442 board, 1541 A-2 board
PCB# 250446-03  Cost reduced 250442-03 board.  the 1541A C/R.

For the 1541C:

PCB# 250448-01  Contains the track 1 sensor logic.  the 1541B board.

For the 1541-II:
```

PCB# 340503 Cost reduced board. Termed the 1541-II board.

There might be others, but these we can confirm. There are 9 if you count the 1541-II board as a 1541 board, 8 if not.

Q $152) The Commodore 6510 CPU has two internal I/O registers. Where in the Commodore 64 are these two registers located at?

A $152) Location $0000 and $0001

Q $153) The Commodore 64 cotains 64kB of memory. How many bytes is in 64kB?

A $153) 65536 bytes

Q $154) What is the name of the Commodore employee responsible for much of the Commodore 128 and 65 software development, among other accomplishments? (hint: initials are FB)

A $154) Fred Bowen.

Q $155) In question $13F, we found out the message that was displayed after typing SAVE "",2. Why did Commodore change that message on the VIC-20?

A $155) The original message, as detailed in Q $13F was:

PRESS PLAY AND RECORD ON TAPE #2

Commodore found that people were pressing the play buttopn BEFORE the record button, which would prevent the record button from functioning in some cases. So, Commodore changed the message to:

PRESS RECORD AND PLAY ON TAPE

To circumvent the problem. Note that the VIC did not have 2 tape interfaces, so no cassette number was needed.

Q $156) What was the number of Commodore 64 machines sold, within 4 million?

A $156) 17 million (This information came from Dave Haynie)

Q $157) What was the number of Commodore 128 machines sold, within 1 million?

A $157) 4.5 million (This information came from Dave Haynie)

Q $158) In 1985, Commodore previewed the Commodore LCD Laptop computer at the January CES show. How many software packages were to be built-in?

A $158) 8:

Word Processor
File Manager
Spreadsheet
Address Book
Scheduler
Calculator
Memo Pad
Telecommunications Package

Q $159) In the Commodore LCD unit, what were the text screen dimensions?

A $159) 80 coumns by 16 rows. 1200 characters on screen.

Q $15A) What is the version number of the only known "bug-free" VIC-II IC?

A $15A) 6569-R5. What's funny is that this chip was manufactured after the Commodore 128 was introduced, so they used the 6569-R3 for the development of the Vic-IIe chip (8563 series), which is buggy. So, the newest PAL 64s have a better VIC than the C128.

Q $15B) Machine language programmer typically use the .X register to index into small arrays. What is the largest byte-array size that can be handled in this way?

A $15B) 256 bytes.

Q $15C) In the mid-1980's, Commodore started manufacturing IBM clone PCs. One of the models had a name which was a type of horse. Name the term.

A $15C) The Commodore "Colt" PC.

Q $15D) What is the model number of the first mouse introduced for the Commodore 64?

A $15D) The 1350.

Q $15E) What was the problem with the mouse in question $15D?

A $15E) As Commodore was either still developing the (now more popular) 1351 mouse or the 1350 was designed as a lower cost alternative, this mouse could only emulate a joystick. When you rolled it up, the joystick "UP" pin was triggered. Likewise for the other directions.

Q $15F) If you hold down the cursor key on the CBM 4000 series machine and it does not repeat, what fact about the machine do you now know? (other than the key doesn't repeat)

A $15F) It is a thin 40XX machine, meaning it could not be upgraded to an 80XX machine via chip swaps.

------- A publication describing BASIC on the Commodore makes the claim that BASIC variables are limited to 5 characters, with the first two being significant. The example to prove this point in the book is given as:

```
ABCDE=5   works, while
ABCDEF=6  does not.
```

The following questions refer to this claim:

Q $160) What is wrong with the above statement?

Q $161) What causes the variable ABCDEF to fail?

Q $162) How long can variable names really be?

Extra Credit: Who was the book publisher?

------- The Commodore LCD Computer system, much like the Commodore 65, was a product that never reached the market. Do you remember this pint-size CBM machine?

Q $163) How many keys were on the CLCD keyboard?

Q $164) What does LCD in the Commodore LCD stand for?

Q $165) Was an internal modem to be includes?

Q $166) Like the Plus/4 the CLCD unit had integrated software. What programs were included?

Q $167) How many batteries of what type did the CLCD use for power?

Q $168) Approximately how much did the CLCD unit weigh?

Q $169) What version of BASIC was to be included with the CLCD computer?

Q $16A) The CLCD unit contained a port that could be used with a Hewlett-Packard device. What did the device do?

Q $16B) What microprocessor did the CLCD unit utilize?

Q $16C) In addition to the usual inclusion of standard Commodore ports, what two industry standard ports were included on the CLCD?

Q $16D) How much RAM did the CLCD computer include?

Q $16E) How many pixels are on the LCD screen on the CLCD machine?

Q $16F) How much ROM did the CLCD computer contain?

==========

#(@)gfx: Hacking Graphics: Let's Get Graphical
by Rick Mosdell (rick.mosdell@canrem.com)

#(A): Introduction

High resolution graphics on the C64 is not all that complicated. How to set up the VIC-chip to see your bitmap and colors IS, so this will not become a discussion on which bits to flip or where to put your blocks of data inside a computer already renowned for its lack of space. Instead, I am interested in the internal data formats of the only two graphics formats.

#(A): Definitions

nybble	the first (lowest) or last(highest) group of 4 bits found in a:
byte	the fundamental 8 bit unit of our C64: an 8-bit computer.
word	two bytes side by side and related, ie. 16 bits. c64 pointers into RAM are lo/hi byte style.
bitmap	a contiguous block of data where a shape is defined when some bits or bit-pairs take a fore-ground color and others take on a background color.
color-ram	a block of data which defines the colors for the bitmap, thus completing the picture.
color nybble	since the C64 has a maximum of nybble 16 colors, to conserve space 2 colors fit into 1 byte.
LORES	important! This would refer to pictures created by using the normal 256-byte character set. Extensive use of the graphics characters in lowercase plus RVSon/RVSoff here. These are NOT graphic files and are text files stored in SEQ format.
MEDRES	this refers to Koala Paint files and related formats.
HIRES	this refers to Doodle files and their derivatives.

#(A): Doodle!

This graphics format is the simplest of all! Here the screen is divided into 64000 pixels of light (320x200) and is truly HIRES. Neither the text color-ram at $D800 nor the 4 background colors at $D021 apply at all. The downside is that only 2 colors can be displayed at one time in an 8x8 pixel block. This is the format geoPaint uses (but allows for a whole page displayed a screenful at a time). These files are prefixed with "dd..." that when compressed become "jj...". They are PRG files that load at $5C00. Internally, the 1K of color (1024 bytes) is first followed by 8K of bitmap (8192 bytes). The "dd..." files are invariably 37 blocks long, which makes sense since they are 9K long (36+ disk sectors each 254 bytes). The "1" bit of the bitmap uses the low nybble value of the color-ram while the "0" bit displays the high nybble color. Simple, straight-forward and direct! Some artists might find this color restriction too hazardous to their health. Since our screen is 320x200 pixels (64000 total remember?) dividing by 8 will give us only 8000 bytes needed for the bitmap and only 1000 bytes necessary for the color-ram. So this format actually wastes 216 bytes! Careful placement of colors can result in spectacular HIRES pictures though: look for the "Lobster" pic in geoPaint and the Doodle files "Middle Earth" and "Pagoda". Other related formats are (refer to the program "Autograph+" by Fuzzy Fox):

* OCP Art Studio
* RUN Paint HIRES
* SID/PIC HIRES
* (geoPaint)

#(A): Koala Paint

Koala is certainly the most colorful and interesting format. Here the screen is MEDRES, resulting in dots double pixel width for a resolution of 32000 elements (160x200). The loss of resol- ution is compensated by the ability to display 4 colors at once in each 8x8 pixel block. Here you have 2 blocks of color-ram, one wherever you put it AND the normal color-ram at $D800. Your picture is also effected screen-wide by the background color at $D021. These files, prefixed by "[cbm-1]pic...", are PRG files that load at $6000. The same files, when compressed, are prefixed by "gg...". That first character in the filename of the raw format means that you cannot delete these files normally. It is made by pressing CBM-1 (orange) and is hex $81 (decimal 129). The only way I know to scratch these files is by typing:

```
OPEN15,DV,15,"s0:[cbm-1]pic...":close15
```

Where DV is your current device. Here the creators of Koala were smart

and wasted NO space. Internally the file contains 8000 bytes for the bitmap(not 8192 bytes), 1000 bytes (not 1024) for the movable color-ram, 1000 bytes for the color-ram at $D800, and (this IS important) ONE more byte for the background color. The length ends up at 41 blocks which is ok considering it is about 10K long. These double width dots are called bit-pairs and they draw their colors from various sources:

%00 from the background color at $D021
%01 from the high nybble of the movable color-ram
%10 from the low nybble of the movable color-ram
%11 from the low nybble of the normal color-ram at $D800.

Interesting eh? There's your 4 colors and where they come from! Related formats would be:

* Advanced OCP Art Studio
* Artist64
* Blazing Paddles
* RUN Paint MEDRES
* SID/PIC multi-color

#(A): Conclusion

Although I am not an expert on graphics screens (not am I claiming to be), I think programmers should be aware of these two poular formats for storing and displaying graphics on the Commodore 64/128.

==========

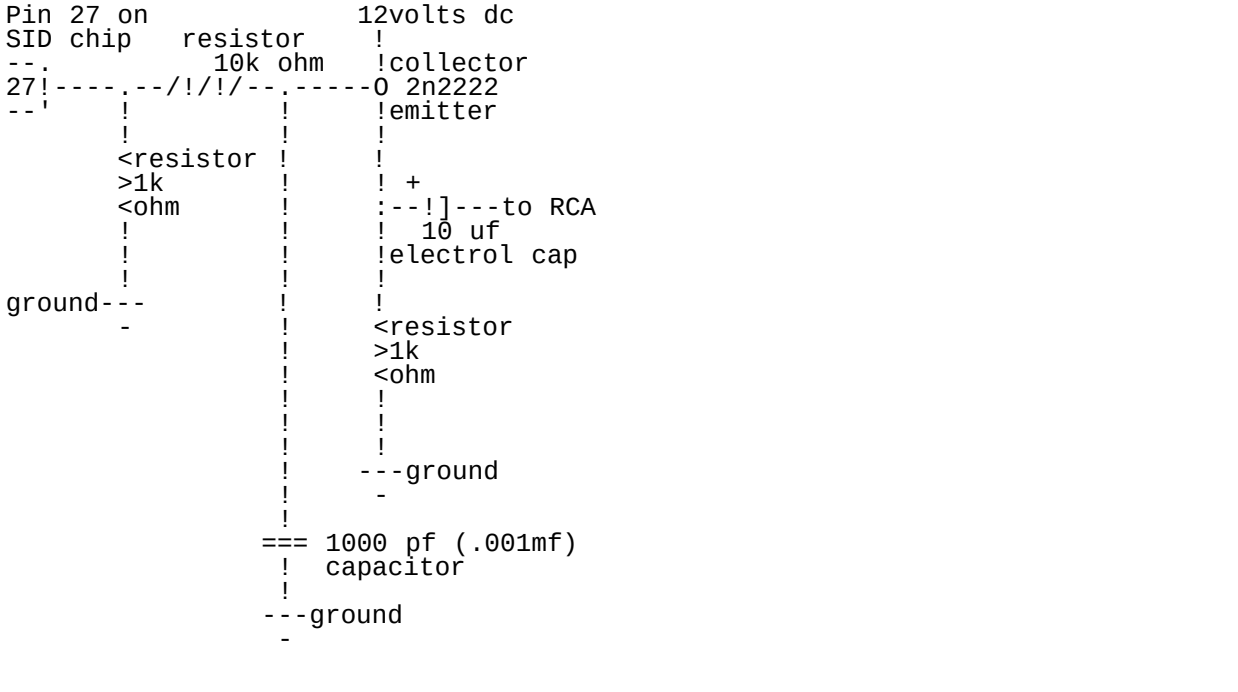
#(@)error: ? DS, DS$: rem The Error Channel

In Commodore Hacking Issue #10, the end of the "Second SID Chip Installation" article was omitted. The omitted section is included below. (SubRef: sid)

Certain early revisions of C=Hacking #10 are missing the end of the article by Alan Jones entitled "Solving Large Systems of Linear Equations on a C64 Without Memory". The omitted text is included below. (SubRef: linear)

#(A)sid: Second SID Chip Installation (Line 137 on)
         (c) 1987 Mark A. Dickenson

Here comes the difficult part to explain. This is the coupling circuit for the audio output. Here is a rough schematic.



You can get the 12 volts you need for the transistor directly from pin #28 of the SID chip.

If you need any help on constructing this circuit check out any of the many books that have schematics on the C-64. This is similar to the one already inside the C-64.

The ground wire from the RCA plug can be soldered to the main grounding strip between the serial and video ports. The center wire will be connected to the negative side of the 10uf electrolitic capacitor.

I still think you should have someone familier with electronics install this circuit for you.

If you have a problem with some cartridges, you will have to install a switch between pin #25 of BOTH SID chips. This will CUT the power to the extra SID chip, effectivly turning it off. I would suggest that you turn OFF the computer before you turn the extra SID chip ON or OFF with this switch.

A good place to mount the switch and RCA plug is on the back of the computer and above the monitor jack on the 64. I still haven't found a GOOD place on the 128. A suggestion was made that if you are not going to use the RF output on the computer, you can cut the wire going to that RCA plug. Then connect your audio output wire to the center connector of the plug. This does work but BE CAREFUL!

Good luck on the construction.

Mark A. Dickenson

#(A)linear: SOLVING LARGE SYSTEMS OF LINEAR EQUATIONS ON A C64
by Alan Jones (alan.jones@qcs.org) WITHOUT MEMORY (Line 239 on)

```
PROC slv(n#,nr#,i#,REF a(),REF c(),REF b(,),sdb#,REF sw#(),REF fail#) CL
OSED
  // This routine solves a system of equations using the quartersolve
  // algorithm with partial pivoting.
  // It is called a "line at a time" and uses only
  // 0.25*nn memory locations which enables larger problems to be solved
```

Slv calls the swap'real and swap'integer proocedures from the strings package. The strings package is a ROMMED package on the Super Chip ROM.

It does exactly what it says, e.g. swap'real(a,b) is the same as: t:=a; a:=b; b:=t.

Slv calls the sdot, isamax#, sswap, sscal, saxpy, and scopy routines from the blas package. The blas package is LINKed to the program, but it could, and should, be placed on EPROM.

Basic Linear Algebra Subroutines, BLAS

The BLAS were originally written for the Fortran language to speed execution and streamline code used for solving linear algebra and other matrix problems. The LINPACK routines, Ref. 3, use the BLAS and are perhaps the best known. The idea is that the BLAS routines will be highly optimized for a particular computer, coded in ML or a High Order Language. Some operating systems even include BLAS like routines. Writing fast efficient programs is then a simple matter of selecting the best solution algorithm and coding it in a manner that makes best use of the blas routines. There are blas routines for single precision, double precision, and complex numbers. The level 1 BLAS perform operations on rows or columns of an array and typicaly do n scalar operations replacing the inner most loop of code. There are also level 2 BLAS that perform n*n operations and Level 3 BLAS that perform n*n*n operations. Nicholas Higham has coded most of the single precision level 1 blas routines and put them in a Comal 2.0 package. The Comal blas package is included on the Packages Library Volume 2 disk. I am not aware of ML blas routines coded for any other C64/128 languages although this is certainly possible and recommended.

The Comal blas routines behave exactly the same way that the Fortran blas routines do except that Fortran can pass the starting address of an array with just "a", while Comal requires "a(1)". The Comal blas will allow you pass an array, by reference, of single or multiple dimensions and start from any position in the array. If you code the blas routines as ordinary Comal routines you have to pass additional parameters and have separate routines for single dimensioned arrays and two dimensional arrays. Note also that Fortran stores two dimensional arrays by columns, and Comal (like many other languages) stores two dimensional arrays by rows. If you translate code between Fortran and Comal using blas routines you will have to change the increment variables.

```
          Fortran                          Comal
dimension c(n), a(ilda,isda)     DIM c(n#), a(lda#,sda#)
```

```
    scopy(n,c,1,a(i,1),ilda)         scopy(n#,c(1),1,a(i#,1),1)
    scopy(n,c,1,a(1,j),1)            scopy(n#,c(1),1,a(1,j#),sda#)
```

The first scopy copies array c into the ith row of array a. The second scopy copies array c into the jth column of array a.

This is what scopy does in Fortran:

```
    subroutine scopy(n,sx,incx,sy,incy)
    real sx(1),sy(1)
    ix=1
    iy=1
    do 10 i = 1,n
      sy(iy) = sx(ix)
      ix = ix + incx
      iy = iy + incy
 10 continue
    return
    end
```

The Comal BLAS does exactly the same thing. If coded entirely in COMAL rather than as a package it would have to be different. The call would change.

scopy(n#,c(1),1,a(1,j#),sda#) would have to become,
scopy(n#,c(),1,1,a(,),1,j#,sda#,sda#) and the Comal procedure might be:

```
PROC scopy(n#, REF x(), ix#, incx#, REF y(,), iy#, jy#, sdy#, incy#) CLOSED
  iyinc#:=incy# DIV sdy#  //assuming y is dimensioned y(?,sdy#)
  jyinc#:=incy# MOD sdy#
  FOR i#=1 TO n# DO
    y(iy#,jy#):=x(ix#)
    ix#:+incx#; iy#:+iyinc#; jy#:+jyinc#
  ENDFOR
ENDPROC scopy
```

Note that more information has to be passed to the procedure and used that the ML blas picks up automatically. Also we would need separate procedures to handle every combination of single and multi dimensional arrays. The Comal ML blas are indeed wonderful. For speed considerations this should also be left as an open procedure or better yet just use in line code.

Here is a very simplified description of what each of the routines in the Comal BLAS package does.

```
sum:=sasum(n#,x(1),1)  Returns sum of absolute values in x().
  sum:=0
  FOR i#:=1 TO n# DO sum:+ABS(x(i#))

saxpy(n#,sa,x(1),1,y(1),1)  Add a multiple of x() to y().
  FOR i#:=1 TO n# DO y(i#):+sa*x(i#)

prod:=sdot(n#,x(1),1,y(1),1)  Returns dot product of x() and y().
  prod:=0
  FOR i#:=1 TO n# DO prod:+x(i#)*y(i#)

sswap(n#,x(1),1,y(1),1)  Swaps x() and y().
  FOR i#:=1 TO n# DO t:=x(i#); x(i#):=y(i#); y(i#):=t

scopy(n#,x(1),1,y(1),1)  Copy x() to y().
  For i#:=1 TO n# DO y(i#):=x(i#)

max#:=isamax#(n,x(1),1)  Returns index of the element of x() with the
                          largest absolute value.
  t:=0; max#:=1
  FOR i#:=1 TO n#
    IF ABS(x(i#))>t THEN t:=ABS(x(i#)); max#:=i#
  ENDFOR i#

sscal(n#,sa,x(1),1)  Scale x() by a constant sa.
  FOR i#:=1 TO n# DO x(i#):=sa*x(i#)

snrm2(n#,x(1),1)  Returns the 2 norm of x().
  norm2:=0
  FOR i#:=1 TO n# DO norm2:+x(i#)*x(i#)
  norm2:=SQR(norm2)

srot(n#,x(1),1,y(1),1,c,s)  Apply Givens rotation.
  FOR i#:=1 TO n# DO
    t:=c*x(i#) + s*y(i#)
```

```
    y(i#):=s*x(i#) + c*y(i#)
    x(i#):=t
  ENDFOR i#
```

Bear in mind that each of these simple examples can be more complex as was given for scopy. You now have enough information to write your own BLAS routines in ML or the programming language of your choice, or to expand the BLAS routine calls in slv to ordinary in line code.

You can also apply the BLAS routines in creative ways besides just operating on rows or columns. For example you could create the identity matrix with:

```
  DIM a(n#,n#)
  a(1,1):=1; a(1,2):=0
  scopy(n#*n#-2,a(1,2),0,a(1,3),1) // zero the rest of the matrix
  scopy(n#-1,a(1,1),0,a(2,2),n#+1) // copy ones to the diagonal.
```

References

1. Zambardino, R. A., "Solutions of Systems of Linear Equations with Partial Pivoting and Reduced Storage Requirements", The Computer Journal Vol. 17, No. 4, 1974, pp. 377-378.

2. Orden A., "Matrix Inversion and Related Topics by Direct Methods", in Mathematical Methods for Digital Computers, Vol. 1, Edited by A. Ralston and H. Wilf, John Wiley and Sons Inc., 1960.

3. Dongarra, J. J., Moeler, C. B., Bunch, J. R., Stewart, G. W., Linpack Users' Guide, SIAM Press, Philadelphia, 1979.

==========

#(@)next: The Next Hack

"But wait, there's more!" Actually, there is, but you'll have to wait until next issue for it. Here's a little appetizer of what's ahead in Commodore Hacking issue #12

- All you cross-development folks, listen up! Issue #12 will enter the realm of cross-compilers with Craig Bruce as he details the construction of a high level language cross compiler. It'll be written in C and will compile a a simplified but structured custom language including concepts from BASIC, C, and Pascal. Concepts like declaration handling, expression interpretion and optimization, and structure control definitions will be discussed

- Gearing up for the 65C816S. C=Hacking will detail the new opcodes available to programmers, show how to detect CPU clock speed on any C64, accelerated or not, discuss pitfalls in code migration, and possibly give a rundown on a prototype accelerator unit from CMD.

- SLIP, Sliding away.... C=Hacking will take an in-depth look at Daniel Dallmann's SLIP-DEMO program and go over the SLIP and TCP/IP protocols in detail as they relate to the Commodore.

- Here Boy, here Boy! Good Dog. The "FIDO's Nuggets" column will bring readers up to date on the discussions in the FIDO CBM echo.

- The RumorMonger. The best rumors we've heard so far. Your mileage may vary...

- All that and C=Hacking's regular columns.

So, set aside a place on that disk drive for the next issue now, because you won't want to miss it...

==========

#(@)bottom