

mailserver which documentation can be obtained by sending mail to "duck@pembvax1.pembroke.edu" with a subject line of "mailserver" and the lines of "help" and "catalog" in the body of the message.

=====
Legal Mumbo-Jumbo

Permission is granted to re-distribut3e this "net-magazine", in whole, freely for non-profit use. However, please contact individual authors for permission to publish or re-distribute articles seperately. A charge of no greater than 5 US dollars or equivlent may be charged for library service / diskettte costs for this "net-magazine".

=====

In This Issue

Commodore Trivia Corner

This edition of Commodore Trivia Corner contains the answers to the July edition of trivia (\$070 - \$07F), the questions and answers for August (\$080 - \$08F), September (\$090 - \$09F), October (\$0A0 - \$0AF), November (\$0B0 - \$0BF), and the questions for the December edition (\$0C0 - \$0CF). Enjoy them!

A Different Perspective, part II

This month George and Steve continue their series on 3D graphics on the C-64 with a look at hidden faces and filled faces. In addition to adding these features into last month's program some other improvements to the old program will be discussed, such as fast multiplication (around 24 cycles) and various bug fixes -- for instance, the program now works on older C-64's which initialize color RAM to the background color when the screen is cleared (sorry about that ;-). A very primitive form of texture mapping is also included. As usual, full source and executables are included. The native C64 files are in a Lynx archive, so you will obviously need Lynx to get at them -- check your favorite BBS or FTP site.

2D Graphics Toolbox: Circles

To augment three-dimensional algorithms this series will focus on two-dimensional drawing algortihms. Circles are the subject this

time around (heh -- get it?), and a very fast algorithm for drawing them on your C64 is presented, with examples in assembly and BASIC7.0. How fast is fast? How does 11 cycles per pixel without the use of tables grab ya?

AFLI=specs v1.0

In AFLI we can get 120 colors in theory (counted like this $16!/(2!*14!)=120$). When we put red and blue hires pixels close to each other we get a vision of purple - thanks the television. This article details what AFLI is, how it's used and done.

Coding Tricks

Included are a series of postings to comp.sys.cbm about neat coding tricks (in machine language) that are interesting and useful.

C.S.Bruce Interview

An interview with the author of Zed, the ACE os and many other numerous utilities for the Commodore 64/128.

Aligning 1541 Drives

A discussion regarding Commodore 1541 disk drive alignment procedures, with suggestions.

=====

Commodore Trivia Corner

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

Well, it is a new year, and I am sending up a new collection of the Commodore trivia for all to enjoy. If you haven't seen this already, the following is a collection of trivia questions that I post to various networks every month. I have collected Trivia Edition #8-13 in this article. As you may know, these questions form part of a contest in which the monthly winner gets a prize (Thanks to my various prize donators). The whole thing is mainly just for fun, so please enjoy.

As the new year rolls in, I am happy to report the following:

- 1) As I have gained access to FIDONet, the trivia is now posted to both the USENET newsgroup COMP.SYS.CBM on the Internet AND the FIDONet echo CBM every month.
- 2) A number of publications have started publishing the trivia, including

Commodore World, and a variety of club newsletters.

3) I have moved into my new house (See new address at bottom). While this may not seem important, the extra room I now have means I can now bring all of my old CBM machine to the new house and work with them. Working with them gives me fodder for more trivia.

As always, I welcome any questions (with answers), and encourage people to enter their responses to the trivia, now at #13. Be sure you get the responses to me by January 12th at noon.

Jim.

The following article contains the answers to the July edition of trivia (\$070 - \$07F), the questions and answers for August (\$080 - \$08F), September (\$090 - \$09F), October (\$0A0 - \$0AF), November (\$0B0 - \$0BF), and the questions for the December edition (\$0C0 - \$0CF). Enjoy them!

Here are the answers to Commodore Trivia Edition #8 for July, 1994

Q \$070) On a PET series computer, what visual power-on indication will tell the user whether the computer has Revision 2 or Revision 3 ROMs?

A \$070) Revision Level 2 ROMS (the ones with more bugs) power up with: *** COMMODORE BASIC ***, with '*' in place of the more familiar '#' character.

Q \$071) The IEEE-488 interface is sometimes called the GPIB interface. What does GPIB stand for?

A \$071) General Purpose Interface Bus. Another name is Hewlett Packard Interface Bus (HPIB), since HP developed this standard for its instrumentation device networking.

Q \$072) Commodore manufactured at least two hard drives with IEEE-488 interfaces. Can you name them?

A \$072) The Commodore D9060 and D9090. From the cbmmodel.txt file:

- * CBM D9060 5 MB Hard Drive, DOS3.0, Off-White, IEEE-488. GP
- * CBM D9090 7.5 MB Hard Drive, DOS3.0, Off-White, IEEE-488. GP

The following model has been said to be in existence, though no one has one on hand to prove it:

- * CBM D9065 7.5 MB Hard Drive

And this model may never have made it past the prototype stage:

CBM D9062 Dual D9065.

Q \$073) Why didn't buyers like the original PET-64?

A \$073) It looked just like a old-style C-64. It had a "home" computer look that the schools didn't care for. They liked the "business" look of the PET series, so Commodore put refurbished and new 64 motherboards in PET cases and sold them as PET 64s. The repackaging suited the schools.

Q \$074) On a PET Revision 2 ROM, what was the largest single array size that BASIC could handle?

A \$074) An array can have a cumulative total of 256 elements. For single dimension arrays, that means D(0) to D(255), but a 2D array can only go from DD(0,0) to DD(1,127) etc. All types of arrays had this limitation.

Q \$075) On the stock 1541, data is transmitted one bit at a time. How many bits are transferred at a time on the Commodore 1551 disk drive?

A \$075) 3 bits were transmitted at a time. I assume that each byte had a parity bit tacked on for error detection, so it would have taken 3 transfers to transmit a byte of information from the drives.

Q \$076) On all Commodore floppy disk drives, how fast does the disk spin?

A \$076) 300 RPM.

Q \$077) Upon first reading the Commodore 1541 Error channel after turning on the disk drive, what error number and text is returned?

A \$077) 73, CBM DOS V2.6 1541, 0, 0

Q \$078) What error number and text is returned on a 1551?

A \$078) 73, CBM DOS V2.6TDISK, 0, 0 Notice that the new text JUST fits!

Q \$079) Commodore printers are normally assigned to device #4, but they can be also used as device #?

A \$079) #5. The Commodore 1525 has a switch to do this, but not all printers have such a switch.

Q \$07A) What microprocessor is used in the Commodore 1551 disk drive?

A \$07A) the 6510T. It is a slight variant on the 6510 microprocessor used on the C64. Some say it runs at 2 MHz, but the specs drives spec sheet doesn't say.

Q \$07B) When the VIC-20 was designed, the serial port throughput was roughly equivalent to the throughput of the IEEE-488 bus? Why isn't it very fast in production VICs?

A \$07B) Let's go back to question \$04F:

<begin insert>

Q \$04F) What was the primary reason Commodore went to a serial bus with the introduction of the VIC-20?

A \$04F) Jim Butterfield supplied me with this one:

As you know, the first Commodore computers used the IEEE bus to connect to peripherals such as disk and printer. I understand that these were available only from one source: Belden cables. A couple of years into Commodore's computer career, Belden went out of stock on such cables (military contract? who knows?). In any case, Commodore were in quite a fix: they made computers and disk drives, but couldn't hook 'em together! So Tramiel issued the order: "On our

next

computer, get off that bus. Make it a cable anyone can manufacture". And so, starting with the VIC-20 the serial bus was born. It was intended to be just as fast as the IEEE-488 it replaced.

<end insert>

And here is what Jim Butterfield followed up with:

"Technically, the idea was sound: the 6522 VIA chip has a "shift register" circuit that, if tickled with the right signals (data and clock) will cheerfully collect 8 bits of data without any help from the CPU. At that time, it would signal that it had a byte to be collected, and the processor would do so, using an automatic handshake built into the 6522 to trigger the next incoming byte. Things worked in a similar way outgoing from the computer, too. We early PET/CBM freaks knew, from playing music, that there was something wrong with the 6522's shift register: it interfered with other functions. The rule was: turn off the music before you start the tape! (The shift register was a popular sound generator). But the Commodore engineers, who only made the chip, didn't know this. Until they got into final checkout of the VIC-20.

By this time, the VIC-20 board was in manufacture. A new chip could be designed in a few months (yes, the silicon guys had application notes about the problem, long since), but it was TOO LATE!

A major software rewrite had to take place that changed the VIC-20 into a "bit-catcher" rather than a "character-catcher". It called

for

eight times as much work on the part of the CPU; and unlike the

shift

register plan, there was no timing/handshake slack time. The whole thing slowed down by a factor of approximately 5 to 6.

When the 64 came out, the problem VIA 6522 chip had been replaced by the CIA 6526. This did not have the shift register problem which had caused trouble on the VIC-20, and at that time it would have been possible to restore plan 1, a fast serial bus. Note that this would have called for a redesign of the 1540 disk drive, which also used a VIA. As best I can estimate - and an article in the IEEE Spectrum magazine supports this - the matter was discussed within Commodore, and it was decided that VIC-20 compatibility was more important than disk speed. Perhaps the prospect of a 1541 redesign was an important part of the decision, since current inventories needed to be taken into account. But to keep the Commodore 64 as a "bit-banger", a new problem arose.

The higher-resolution screen of the 64 (as compared to the VIC-20) could not be supported without stopping the CPU every once in a while.

To be exact: Every 8 screen raster lines (each line of text), the CPU

had to be put into a WAIT condition for 42 microseconds, so as to allow the next line of screen text and color nybbles to be swept

into

the chip. (More time would be needed if sprites were being used). But the bits were coming in on the serial bus faster than that: a bit would come in about every 20 microseconds! So the poor CPU, frozen for longer than that, would miss some serial bits completely! Commodore's solution was to slow down the serial bus even more. That's why the VIC-20 has a faster serial bus than the 64, even

though

the 64 was capable, technically, of running many times faster.

Fast disk finally came into its own with the Commodore 128."

--Jim

Q \$07C) On Commodore computers, how much RAM is set aside as a tape buffer?

A \$07C) 192 bytes is used as a tape buffer. Blocks of data on tape are 192 bytes long.

Q \$07D) On Commodore computers, most every peripheral has a device number. What is the device number of the screen?

A \$07D) #3

Q \$07E) What is the device number of the keyboard?

A \$07E) #0

Q \$07F) Commodore computers use 2's-complement notation to represent integers.

What is the 2's-complement hex representation of the single byte -1?

A \$07F) (This was not a Commodore specific question) Commodore computers use this notation to represent integer quantities. In 2's complement

notation, a -1 looks like 11111111(binary) or \$FF(hex).

Here are the answers to Commodore Trivia Edition #9 for August, 1994

Q \$080) During the days of the Commodore 64 and the VIC-20, Commodore produced at least two Commodore magazines. What were their names?

A \$080) The magazines were originally called "Commodore Microcomputers" and "Power/Play: Commodore Home Computing". They never did seem to nail down the name of the latter as I see "Power/Play" and "Commodore: Power/Play" used as the original names as well. Anyway, Commodore Microcomputers started its life in 1979, whereas "Power/Play" started in 1981. Both magazines were published until around 1987, when they were merged to form "Commodore Magazine". Then, around 1990, the magazine was sold to IDG Communications and was merged into RUN. RUN was continued for a while, but was finally pulled out of circulation. Creative Micro Designs purchased the rights to the magazine, and now Commodore World is being produced by CMD. I am not sure how strong (if any) a link there is between RUN and CW, but some of the same authors write for the new publication. Just for added info, here are the ISSN numbers:

Commodore Microcomputers (Commodore Magazine)	0744-8724
Power/Play:Commodore Home Computing	0739-8018
RUN (Commodore/RUN)	0741-4285

"The Transactor" is also a correct answer, and info on it is below.

Q \$081) Back in the PET heyday, another magazine was produced by Commodore Canada. This magazine was later sold and showed up as a hardware journal. Name the magazine.

A \$081) The infamous "Tarnsactor". One of the noted C64 hardware-hacking magazines, it was originally published by Commodore Canada, before being sold to an individual named Mr. Hilden. Its ISSN number is 0838-0163. As far as I can tell, this magazine, died many deaths, but ceased to exist in 1989-90. Its first issue is dated April 30, 1978.

Q \$082) The Commodore 128 has a VIC-II compatible chip inside it. Can this chips be switched for a VIC-II from a Commodore 64?

A \$082) No! The newer 128 compatible chip (VIC-IIe) has 8 extra pins to

perform timing functions specific for the 128. In addition, some of the registers have extra functions. However, a suitable card to make it compatible can be made.

Q \$083) What does the video encoding standard PAL expand to?

A \$083) Phase Alternating Line is the answer I was looking for, which describes the video encoding used in Europe, but Programmable Array Logic is also correct, which describes the family of chips used as "glue" logic for the C64 I/O and processing chips.

Q \$084) How many buttons were present on the earliest of Commodore tape decks?

A \$084) 5: Play, Rewind, Fast-Forward, Record, and Stop/Eject. Later models separated the stop and eject functions into two buttons.

Q \$085) Earlier SID chips had a distinctive "clicking" sound that some demo coders used to an advantage. Commodore subsequently removed the click, and then later reintroduced it. When does the telltale click occur?

A \$085) When you change the volume of a voice. The voice need not be outputting anything.

Q \$086) What does CP/M stand for?

A \$086) Take your pick:

Control Program/Monitor
Control Program for Microprocessors
Control Program for Microcomputers.

The last one is considered by many to be most correct.

Q \$087) What is the highest line number allowed for a program line in Commodore BASIC V2?

A \$087) Normally, the user cannot enter a line number higher than 63999. If you want to be tricky, however, the numbers can be made to go up to 65535.

Q \$088) What symbol, clearly printed on the front of a key on the Commodore VIC, 64, and 128 keyboard, is not available when the lower case character set is switched in?

A \$088) The PI symbol. It is [SHFT-UPARROW] in uppercase mode, but becomes a checkerboard-like character when in lower-case mode. Unlike the graphics characters printed on the fronts of the keys, this one is positioned in the middle of the keycap, and should probably be accessible in both character sets.

Q \$089) How do you get the "checkmark" character ?

A \$089) In lowercase mode, type a shift-@

Q \$08A) On the PET computers, what memory location holds the Kernal ROM version?

A \$08A) It is different from the 64/128. It is 50003. 0 here indicates old ROMs, while 1 indicates new ROMs.

Q \$08B) The Commodore computers have 2 interrupts, called IRQ and NMI. What does IRQ stand for?

A \$08B) Interrupt ReQuest. This interrupt is used for things that should usually be allowed to interrupt the processor. This interrupt can be masked off by the SEI instruction.

Q \$08C) What does NMI stand for?

A \$08C) Non-Maskable Interrupt. Unlike the IRQ, this interrupt cannot be masked by an instruction. However, some tricks can be used to mask it.

Q \$08D) The 6502 line of microprocessors has a number of flags that can be used to test for certain conditions. One of them is the N flag. What does it stand for?

A \$08D) 'N' stands for Negative. On instructions that change this flag, it is set to be equal to bit 7 of the result of the instruction.

Q \$08E) How about the D flag?

A \$08E) It stands for decimal mode. This mode causes certain instructions to treat a byte as 2 4 bit BCD-coded nybbles.

Q \$08F) The shorthand for the BASIC keyword PRINT is '?'. What is the shorthand equivalent for PRINT#?

A \$08F) pR is the way to abbreviate PRINT#. Note that ?# will fail.

Here are the answers to Commodore Trivia Edition #10 for September, 1994

Q \$090) The 6502 has a rich history. It is modeled after another 8-bit microprocessor. Name the processor.

A \$090) The 65XX series of processors was modeled after the Motorola 6800. Motorola hampered the design groups' efforts to pursue product developments using the 6800. A core group of 8 designers left Motorola and went to MOS Technologies, which was the largest producer of

calculator chips at the time. MOS decided it was time to go into the CPU business.

Q \$091) The 6502 has a older brother that was never produced. Name its number designation and why it was not produced.

A \$091) The older brother to the 6502 was the 6501. The 6501 was pin-compatible with the 6800, which prompted a suit by Motorola. Eventually, MOS reached an agreement where they scrapped the 6501 marketing, but were free to market the 6502.

Q \$092) How many different opcodes are considered valid and "legal" on the MOS NMOS 6502 line?

A \$092) 151 opcodes are documented in the NMOS 6502 data book. The remaining 105 opcodes were not implemented, and exist as "don't care" states in the opcode matrix. That means that some seemingly invalid opcodes will actually perform pieces of two or more valid opcodes. Newer CPU systems trap all non-implemented opcode usages, but not the 6502.

Q \$093) Every instruction takes at least __ cycles to complete. Fill in the missing number.

A \$093) 2. The architecture assumes that each opcode has two bytes in it and one byte can be fetched per cycle. For instructions that use only 1 byte, the extra fetched byte (actually the next opcode), is thrown away.

Q \$094) Which instructions take more time than necessary as a result of the answer to Q \$093?

A \$094) Although this is a subjective answer, One could nominate NOP on the basis that NOP is generally believed to waste one execution cycle on a particular processor, namely one cycle on the 65XX line. However, one can argue that NOP simply means no operation, and has no ties to length of execution. You be the judge.

All other instructions must take at least two cycles: one for opcode fetch, one for operation.

Q \$095) What did MOS Technologies manufacture before introducing the 650X line of microprocessors?

A \$095) As stated above, it was calculator chips.

Q \$096) Three companies manufactured the 6502 under a cross-licensing agreement. Name them.

A \$096) Rockwell, MOS Technologies, and Synertek.

Q \$097) In NTSC-land, how fast does the 1MHz 6510 in the C64 actually run?

A \$097) 1.022727143 MHz. It is derived by taking the main clock frequency (14.31818MHz) and diving it by 14.

Q \$098) What about in PAL-land?

A \$098) 985.248449 kHz. It is derived by taking the main clock frequency (17.734472MHz) and dividing it by 18. Thus the PAL 64 actually runs slower than the NTSC one.

Q \$099) Data is latched into the 650X microprocessor on the (rising/falling) edge?

A \$099) Data is latched in to the 65XX on the falling edge of Phi0 (Phi1). The timing diagram in some books (64 PRG is one) is incorrect.

Q \$09A) Through the years, the 650X line has changed family numbers, yet the part has not been changed. (A family number is the upper 2 digits in this case) Name the other family numbers used by MOS to denote the 650X line.

A \$09A) the 75XX line used in the 264 series (Plus/4 and C16), and the 85XX series used in the C64C and C128 series.

Q \$09B) Consider the following code:

```
ldx #10
lda $ff,x
```

what location does the accumulator get loaded with?

A \$09B) The answer is location \$ff+10 mod 256 = \$09. The answer involves explaining a (mis)features of the NMOS 65XX CPU line. The above code instructs the 65XX CPU to use zero-page addressing mode to load the accumulator. In zero-page addressing, the

address need only be one byte wide (\$ff in this case), because the high byte is considered to be \$00. Now, as humans, we would expect the CPU would add 10 to 255 (\$ff), giving 265 (\$109) as the address to load the accumulator from. However, the CPU designers decided that zero-page addressing means that the high byte will be \$00 all

the time, no exceptions. If a situation like the above occurs, the low byte of the addition will be used as the low byte of the address (9 in this case), but the high-byte will be ZERO. All zero page addressing modes work this way. Note that the CMOS versions of the 6502 do perform the high byte "fix-up", so this behavior is only

seen

on the NMOS parts.

Q \$09C) What about the following?

```
ldx #10
lda ($ff),x
```

A \$09C) This was a trick. The code is trying to use INDIRECT INDEXED indexing

mode using the x register, but that addressing mode can only be used with the y register. If the code is changed to the following, legal code:

```
ldx #10
lda ($ff),y
```

Then, the above discussion for zero-page addressing holds true here as well. The effective address would have been (hi:lo) \$100:\$0ff, but is instead (hi:lo) \$000:\$0ff. The simple rule is: zero page means exactly that. There is no way to address outside of zero-page with zero-page addressing.

Q \$09D) How many CPU clock signal lines does the 650X require to run?

A \$09D) 1. The 6501 used two, as the 6800 used two, but the 6502 and successors only required Phi0 (Phi1). Phi2 was generated on the CPU.

Q \$09E) Where does the 650X line fetch its first byte from after reset?

A \$09E) \$fffc. The address formed by reading \$fffd and \$fffc is stuffed into the IP, and the code is read starting there. \$fffc is read first, since the 65XX line stores addresses in low byte, high byte format.

Q \$09F) One of the original designers on the NMOS 6502 CPU now heads up Western Design Center in Arizona, and makes the 65C02 and 65C816 CPU chips. Name him. Hint: it is not Chuck Peddle!

A \$09F) Bill Mensch. He hand-designed these newer parts in the 65XX line in the same manner he and Chuck Peddle and others hand-designed the 6501 and 6502.

Here are the answers to Commodore Trivia Edition #11 for October, 1994

Q \$0A0) In the mid 1980's, Commodore introduced RAM Expansion Units for the Commodore 64, 64C, 128, and 128D. There were three of them. Give their model numbers, and what was different among them.

A \$0A0) The 1700 (128kB), the 1764 (256kB), and the 1750 (512kB). The 1700 and the 1750 were marketed for the 128, while the 1764 was marketed from the 64 line.

Q \$0A1) Some of the CIA integrated circuits used on the C64 and C128 computers have a hardware defect. What is the result of this defect, and when does it occur? (May be more than one, but I need only one)

A \$0A1) The only one I have documented in front of me is the timer B interrupt bug, which is explained in the "Toward 2400" article by George Hug in Transactor 9.3. (1) However, I had many people relate other bugs (2 and 3), which I haven't been able to test, so I add them as possibilities. (I encourage readers to confirm/deny the latter 2.)

1) If timer B of the 6526 CIA times out at about the same time as a read of the interrupt register, the timer B flag may not be set at all, and no interrupt will occur if timer B interrupts were turned on.

2) When the hour on the TOD clock is 12, the AM/PM must be reversed from its normal setting to set/reset the AM/PM flag.

3) The TOD clock sometimes generates double interrupts for alarm trigger.

Q \$0A2) Name the Commodore machine(s) on which a Intel 8088 was an OPTIONAL coprocessor. (Hint, not the IBM clones)

A \$0A2) I was looking for the B series computers, which contains the B computers (B128, B256), as well as the 600 series and the 700 series. These computers could be fitted with an optional 8088 processor on a separate card. However, another correct answer is the Amiga, which can have a 8088 attached via an expansion card or a SideCar(tm) unit.

Q \$0A3) On Commodore computers beside the Plus/4 series, there are three frequencies used to record the data on the tape. Name the frequencies used.

A \$0A3) 1953.125Hz, 2840.909Hz, and 1488.095Hz. These correspond to waveforms with periods: 512us, 352us, and 672us, respectively.

Q \$0A4) Commodore Plus/4 series computers can not read any cassettes recorded on other Commodore computers. Why? (Hint: It has nothing to do with the nonstandard connecotr on the Plus/4)

A \$0A4) The tones recorded on the Plus/4-C16 are exactly one-half the

frequencies shown above. This suggests to many that the Plus/4 and C16 were supposed to run at twice its present frequency, but were downgraded at the last-minute, and the code to generate the tones was not updated to reflect the change. This is just heresy, so you decide for yourself.

Q \$0A5) During power-up, the Commodore 64 checks to see if it running in PAL-land or NTSC-land. How does it determine its location?

A \$0A5) It sets the raster compare interrupt to go off at scan line 311. If the interrupt occurs, we are on a PAL system, since NTSC will never get to line 311 (NTSC only has 262.5 lines per frame, every other frame shifted down a bit to create 525 lines).

Q \$0A6) What is the 65XX ML opcode for BRK?

A \$0A6) \$00, or 00

Q \$0A7) On the 65XX CPU, what gets pushed onto the stack when an interrupt occurs?

A \$0A7) The program counter gets saved high byte first, then the processor status flags get saved.

Q \$0A8) Speaking of the stack, where is the stack located in the 65XX address map?

A \$0A8) \$0100 to \$01FF

Q \$0A9) On the 65XX CPU line, it is possible to set and clear a number of processor status flags. Examples include SEC and CLC to set and clear the carry flag. What flag has a clear opcode, but no set opcode?

A \$0A9) The overflow flag: V. However, the V flag can be set via an external

pin on some members of the 65XX line. The 1541 uses this as an ingenious synchronization tool.

Q \$0AA) When saving a text file to tape, the computer records 192 bytes of data, an inter-record gap, and then the same 192 bytes of data again. How wide is this inter-record gap, and why is it there?

A \$0AA) Some terminology: "inter" means "between". Most everyone knows that a tape block is recorded twice on the tape, but Commodore considers the two copies and the gap between them a single "record". Thus, this question is referring to the gap in between two dissimilar records. With that in mind, the interrecord gap is nominally 2 seconds long, (or 223.2 byte lengths, although the gap contains no data). It is there to allow

the tape motors to get up to speed before the next data comes under the read/write head. The tape motors may need to stop between records if the program is not requesting any more data from the tape data file at this time. If the program subsequently asks for data from the tape, the drive must get up to speed before the read can occur. Note: on the first version of PET BASIC, the gap was too small, so programmers had problems retrieving data files.

For completeness, the "intra-record" gap (The one between the two copies of the data) consists of 50+ short pulses, each of which is 352us in length, giving a timing of .0176s+. This time was used to copy important data to safe locations, reset pointers, and do error logging. The entire "record" is recorded in 5.7 seconds.

Q \$0AB) On an unexpanded VIC-20, where does the screen memory start?

A \$0AB) \$1e00, or 7680

Q \$0AC) In Commodore BASIC, what is the abbreviated form of the "Load" command?

A \$0AC) lO (L SHIFT-0)

Q \$0AD) In Commodore BASIC, what is the abbreviated form of the "List" command?

A \$0AD) lI (L SHIFT-I)

Q \$0AE) On the Commodore 64, there is section of 4 kilobytes of RAM that cannot be used for BASIC programs. It is the favorite hiding places for many ML programs, however. What is its address in memory?

A \$0AE) \$c000, or 49152

Q \$0AF) What is stored at locations \$A004-\$A00B, and why is it strange?

A \$0AF) The text "CBMBASIC" is stored there. It is strange because this text is not referenced by any routine. It can also be called strange because the code is Microsoft's. Doesn't it make you wonder?

Here are the answers to Commodore Trivia Edition #12 for November, 1994

Q \$0B0) What will happen if you type ?""+-0 into the CBM BASIC interpreter on the PET series, the 64 series, or the 128 series?

A \$0B0) The BASIC interpreter has a bug in it that shows up while interpreting the above statement. The interpreter leaves two bytes on the CPU

stack prior to returning from a subroutines call. At least on the C64, the two bytes are both zeros. Since subroutines put the return address on the stack, the return retrieves the two bytes left on the stack and attempts to se that as the return address. So, depending

on

what code it executes after the return, it can do a number of things.

Most of the time after the bug occurs, the interpreter limps along for a while until it hits a BRK instruction, \$00. Then, that instruction causes the system to execute an interrupt. On the C64, the system vectors through \$316-\$317 (BRK vector) and does a warm start. On the C128 and PETs with Monitors, the system dumps into

the

internal machine language monitor. If the machine under use did not do something with the BRK vector, the machine will hang.

Now, note that the above is not the only result. Since the interpreter is executing code from the wrong location, any result from no effect to hung machine is possible.

Note that this is NOT normal behavior. The system should report an error while interpreting the above statement.

Q \$0B1) In the first CBM 64 units, what color was the screen color RAM changed to when you cleared the screen?

A \$0B1) The screen color RAM was changed to value 1 when the screen was cleared. Thus, when a byte was poked into screen RAM, the resulting character was white on the screen. The white contrasted nicely with the normal blue background.

Q \$0B2) Why was it changed in later versions of the 64?

A \$0B2) Commodore found that this practice sometimes caused "light flashes" during screen scrolls. I was going to leave this for another time, but ... The change was to make the color RAM equal to background color register #0. Well, this got rid of the "light flashes", but then poking values to screen RAM caused invisible characters, since the foreground color of the character was the same as the background color of the screen.

Well, this broke a number of older programs that did not properly initialize the color RAM. Also, Commodore fixed the problem with the VIC-II that had caused these "light flashes" So, Commodore changed the KERNAL a third time. Since the above change caused invisible characters, Commodore made a third revision that changed the color RAM to the value in location 646 (the current cursor foreground color).

Q \$0B3) What is "special" about the text that displays the "illegal quantity

error" in CBM BASIC?

A \$0B3) The text is actually "?ILLEGAL QUANTITY ERROR". Notice the two spaces between "QUANTITY" and "ERROR". John West supplies the explanation:

"The vector at \$0300 points to a routine at \$A43A, which is the general error message printing routine. Load .X with the number of the error, and it prints it. It looks up the address of the error text from a table, then prints the text, which does not have any trailing spaces. It then prints ' ERROR', with *2* spaces. It does this for all errors."

Historically, this effect is caused by the VIC-20, which only had 22 columns. When the VIC-20 BASIC was being ported from the PET BASIC code, someone noticed that some of the error strings would span two VIC-20 lines. So, the BASIC error messages were changed a little, so that they all printed neatly on two lines: The PET error string:

?ILLEGAL QUANTITY ERROR (one space) became:
?ILLEGAL QUANTITY
ERROR (carriage return plus one space).

When the C64 BASIC was being ported from the VIC-20, the carriage return was replaced with a space character.

I admit this caught me by surprise. I have used Commodore computers for years, and never noticed that "?SYNTAX ERROR" had 2 spaces in it.

Q \$0B4) On what Commodore machine was the operating system OS/9 available?

A \$0B4) Since OS/9 was a real-time operating system for the 6809 microprocessor, it was available on only one Commodore machine, which had two different names: The Commodore SuperPET. The machine was sold as the "MMF (Micro MainFrame) 9000 in Germany, and its model number was SP9000.

Q \$0B5) Which Commodore machine(s) does not have a user port?

A \$0B5) There were a number of answers to this question, and there may be more:

The Commodore C16. Commodore decided to cut out telecommunications, and thus designed the user port out of the computer, as the modem is the only use Commodore ever made of the user port. This also includes the C116, a version of the C16 with a chicklet keyboard.

The Commodore Ultimax/MAX machine. This was the ill-fated game console produced in the early 80s. It was basically a stripped down Commodore 64.

The 64 GS (Game System). This machine was another flop produced in the late 80s.

Q \$0B6) How many pins are there in a Commodore Serial Connector?

A \$0B6) 6.

Q \$0B7) There are 13 addressing modes available on the 6502. Name them.

A \$0B7)	No#	Name	Description
	----	-----	-----
	01)	accumulator	asl a
	02)	immediate	lda #\$00
	03)	zero page	lda \$00
	04)	zero page,X	lda \$00,X
	05)	zero page,Y	lda \$00,Y
	06)	absolute	lda \$1000
	07)	absolute,X	lda \$1000,X
	08)	absolute,Y	lda \$1000,Y
	09)	implied	clc
	10)	relative	bne
	11)	(indirect,X)	lda (\$00,X)
	12)	(indirect),Y	lda (\$00),Y
	13)	(absolute indirect)	jmp (\$1000)

Q \$0B8) If you were to put one large sequential file onto an 8050 disk drive, how big could that file be?

A \$0B8) According to the 8050 User Manual, a sequential file could be 521208 bytes in size.

Q \$0B9) How many characters can be present in a standard Commodore DOS filename?

A \$0B9) 16 characters.

Q \$0BA) How many pins does a 6502 IC have on it?

A \$0BA) 40 pins.

Q \$0BB) How many pins does the standard IEEE-488 connector have on it?

A \$0BB) 24 pins.

Q \$0BC) On the IEEE-488 bus, what does the acronym for pin 7, NRFD, stand for?

A \$0BC) Not Ready For Data.

Q \$0BD) On the NMOS 6502, what is the ML opcode for SED, and what does this

opcode do?

A \$0BD) \$f8, SEt Decimal mode. Sets the D flag in the status flags byte. Although used rarely, this opcode switches on Binary Coded Decimal mode. In BCD mode, the byte \$10 is treated as 10, not 16. The add and subtract instructions are the only legal ones affected by this mode, although some undocumented/illegal opcodes are also affected. For example, in this mode, adding the byte \$15 (21) to the byte \$25 (37) yields \$40 (64) not \$3a (58). emember that, in this mode, \$40 = 40, not 64.

Q \$0BE) Assuming a PET computer and a non-PET computer have access to a common disk drive or tape drive, there are two ways to load a PET BASIC program on the non PET CBM computer. Name them.

A \$0BE) Most differing series of Commodore computers had different places for the start of BASIC programs. For instance, on the C64, \$0801 (2049) is the start of BASIC memory, but most PET computers start BASIC memory at \$0401 (1025). This wouldn't matter, except that BASIC programs are stored on tape and disk with the start address, and the line links in a BASIC program have absolute addresses in them.

To fix these problems, the Commodore VIC-20 and newer computers came out with a "relocatable load". So, here are the two choices:

1) Save the program on the PET like so: save "name",X (X is device).

Then, you could load the program into the non-PET machine by using a relocatable load: load "name",X. This would load the program in at start of BASIC memory and refigure the line links.

2) Redefine start of BASIC memory on non-PET machine. A couple of pokes to relevant BASIC pointers, and the start of BASIC was moved. Then, load the program non-relocatable.

Now, from the above discussion, it looks like option 1 is the simplest route. Well, it would be, exopt for one small detail: Earlier PET computers saved the BASIC program from \$0400, not \$0401 as is expected. The effect: loading relocatable on a non-PET would have a zero byte as the first byte of the program. The quick fix: change BASIC pointer to itself-1, load PET program, reset BASIC pointer. Commodore didn't make it easy!

Q \$0BF) Only one of the ways detailed in \$0BE works the other way around. Which one?

A \$0BF) Since the earlier PET computers did not have a "relocatable load", the only way to load a program from, say, a C64 into an 2001 was to use option #2 above and move the start of BASIC memory to \$0801 (2049).

Commodore Trivia Edition #13

- Q \$0C0) The early 1541 drives used a mechanism developed by _____. Name the company.
- Q \$0C1) On later models, Commodore subsequently changed manufacturers for the 1541 drive mechanism. Name the new manufacturer.
- Q \$0C2) What is the most obvious difference(s). (Only one difference is necessary)
- Q \$0C3) On Commodore BASIC V2.0, what answer does the following give:
PRINT (SQR(9)=3)
- Q \$0C4) In Commodore BASIC (Any version) what does B equal after the following
runs: C=0:B=C=0
- Q \$0C5) The first PET cassette decks were actually _____ brand cassette players, modified for the PET computers. Name the company.
- Q \$0C6) In Commodore BASIC (Any version), what happens if the following program is run:
- ```
10 J=0
20 IF J=0 GO TO 40
30 PRINT "J<>0"
40 PRINT "J=0"
```
- Q \$0C7) In question \$068, we learned how Jack Tramiel first happened upon the name "COMMODORE". According to the story, though, in what country was he in when he first saw it?
- Q \$0C8) On the Commodore user port connector, how many edge contacts are there?
- Q \$0C9) On most Commodore computers, a logical BASIC line can contain up to 80 characters. On what Commodore computer(s) is this not true?
- Q \$0CA) If a file is saved to a Commodore Disk Drive with the following characters: chr\$(65);chr\$(160);chr\$(66), what will the directory entry look like?
- Q \$0CB) What is the maximum length (in characters) of a CBM datasette filename?
- Q \$0CC) How many keys are on a stock Commodore 64 keyboard?
- Q \$0CD) Commodore BASIC uses keyword "tokens" to save program space. Token 129 becomes "FOR". What two tokens expand to include a left

parenthesis as well as a BASIC keyword?

Q \$0CE) There are 6 wires in the Commodore serial bus. Name the 6 wires.

Q \$0CF) On the Commodore datasette connector, how many logical connections are there?

Some are easy, some are hard, try your hand at:

Commodore Trivia Edition #13!

Jim Brain  
brain@mail.msen.com  
602 North Lemen (New address)  
Fenton, MI 48430  
(810) 737-7300 x8528

=====  
==

## A Different Perspective, part II

by George Taylor (aa601@cfn.cs.dal.ca) and Stephen Judd (sjudd@nwu.edu).

We... are... VR Troopers! Okay Troopers, once again we need to make an excursion out of the three dimensional world and into our own little virtual world inside the C64. So sit back in your virtual chair, put on your virtual thinking helmet, maybe grab a virtual beer, and prepare for a virtually useful experience with another virtually humongous article.

Last time we laid down the foundations of 3D graphics: rotations and projections. In this article we will build upon this foundation with a look at hidden surfaces as well as filled surfaces. In addition we will snaz up the old program so that it is a little more efficient by for instance introducing a much faster multiplication routine and moving all variables into zero-page.

To get us in the mood let's review from last time. We are in a three-dimensional space; in particular, a right-handed three-dimensional space, so that the x-axis comes towards you, the y-axis increases to the right, and the z-axis increases "up". Now we have some object, centered at the origin.

To rotate the object we derived a 3x3 matrix for each axis which describes a rotation about that axis. After rotating we translate the object along the z-axis and then project it through the origin onto a plane z=constant.

As you recall the projection of a point is done by drawing a line from the point through the origin, and then figuring out where this line intersects our plane  $z=\text{constant}$ . You can think of this line as a ray of light bouncing off the object and through our little pinhole camera lens.

Clearly for any solid object some parts of the object will remain hidden, though, i.e. when you look at your monitor you can't see the back of it, and you probably can't see the sides. How do we capture this behavior mathematically?

## Hidden Surfaces

-----

Imagine our object with some light shining on it -- when will a part of the object be hidden? Clearly it is hidden when the light reflected off of it never reaches our eyes, which happens whenever a part of the object is "turned away" from us. How do we express this mathematically? Consider a flat plate, like your hand (you also might think of a cube). Now imagine a rod sticking out of the plate, exactly perpendicular to the plate (take your index finger from your other hand, and touch it to your palm at a ninety-degree angle). Now rotate the plate around, and imagine the light bouncing off and heading towards your eyes.

No matter where you place your hand in space, the very last point at which it is visible is when it is exactly parallel to the light rays coming from it to your eyes; or, to put it another way, when the light rays are exactly perpendicular to a normal vector to the surface (in the above case this vector is either a rod or your finger). If the angle between the normal and a light ray is less than ninety degrees, then the surface is visible. If greater, then the surface is invisible.

At this point you may be wondering how to figure out the angle between two vectors. It turns out we really don't have to calculate it at all: instead we use a very important tool in our mathematical toolbox, the dot product.

If we have two vectors  $v_1=(x_1,y_1,z_1)$  and  $v_2=(x_2,y_2,z_2)$  then the dot product is defined to be

$$v_1 \text{ dot } v_2 = x_1*y_1 + x_2*y_2 + x_3*y_3$$

note that this is a `_scalar_` (i.e. a number), and not a vector. You can also show that

$$v_1 \text{ dot } v_2 = |v_1|*|v_2|*\cos(\text{theta})$$

where  $| |$  denotes length and  $\text{theta}$  is the angle between the two vectors. Since  $\cos(\text{theta})$  is positive or negative depending on whether or not  $\text{theta}$

is

less than or greater than ninety degrees, all we have to do is take the dot product and look at the sign.

But we need to understand something about the dot-product.  $\theta$  is the angle between two vectors joined at their base; mathematically the way we are going to draw the light ray is to draw a line FROM the origin TO a point on the surface. In our model above, we are going to draw a line from your eyes to the palm of your hand and then slide the normal vector down this line until the base of the normal vector touches your eye.

The whole point of this is that when we look at the dot product we need to keep in mind that if the dot product is negative, the face is visible.

All that remains is to write down an equation: let's say that we've rotated the surface and know a point  $P=(x,y,z)$  on the rotated surface, and we have a normal vector to the surface  $v_n=(v_x,v_y,v_z)$ . First we need to translate down the z-axis so that  $P \rightarrow (x,y,z-z_0) = P - (0,0,z_0)$ . If we then take the dot product we find that

$$P' \cdot v_n = (P \cdot v_n) - z_0 \cdot v_z$$

But  $(P \cdot v_n)$  is simply a constant: because these are rigid rotations the length of  $P$  never changes, presumably the length of  $v_n$  never changes, and the angle between the two never changes. So introduce a constant  $K$  where

$$K = (P \cdot v_n) / z_0$$

so that all we need to do is subtract the z-component of the normal vector from  $K$  and check if it is positive or negative: if negative, the face is visible. Note that if we translate by an amount  $P + (0,0,z_0)$  (instead of  $-(0,0,z_0)$ ) we simply add the two together.

We seem to have left something out here: how do we calculate the normal vector  $v_n$ ? One way to do it is by using another vector operator, the cross-product. The dot product of two vectors is just a scalar, but the cross product of two vectors is another vector, perpendicular to the first two.

The most common way to visualize the cross-product is by using your right hand: imagine two vectors in space, and place your right hand along one of them, with your thumb sticking out. Now curl your fingers towards the other vector. Your thumb points in the direction of the vector formed from the cross-product of the first two. You can easily convince yourself then that  $(A \times B) = -(B \times A)$ , that is, if you reverse the order of the cross product, you get a vector pointing in the opposite direction.

Therefore, if we take any two vectors in the face (in particular, we know the



edge of the face), and then take their cross-product, we have a normal vector.

But because we are dealing with a cube, we have an even easier method! We can use the fact that the faces on a cube are perpendicular to each other: if

we take two points and subtract them we get a vector going between the two points. On a cube, this will give us a normal vector if we use two "opposite" points. Therefore all we need to do is rotate the cube, subtract two z-coordinates, add to K, and check if it is positive or negative.

This is how the program does it, and the specifics will be explained later. Right now I want to show you a second method of hidden surface detection. Instead of using the three-dimensional rotate vectors, what if we use the two-dimensional `_projected_` vectors? If we take the cross-product of two of these vectors we get a vector which either points into the screen or out of it, which corresponds to a positive or a negative result.

The cross-product is usually done by taking the determinant of a matrix. I am not going to explain that here -- you can look in any decent calculus book

for the full cross-product. All we really care about is the z-coordinate of the vector, and the z-coordinate of  $v_1 \times v_2$  is:

$$v_1x*v_2y - v_1y*v_2x$$

Whether or not the face is visible depends on how you define  $v_1$  and  $v_2$ ! Always remember that  $(v_1 \times v_2) = -(v_2 \times v_1)$ .

What is this quantity anyways? Consider a parallelogram made up of our two vectors  $v_1$  and  $v_2$ . The magnitude of the cross-product just happens to be

$$|v_1|*|v_2|*\sin(\theta)$$

which you can easily see is the area of a parallelogram with sides  $v_1$  and  $v_2$ .

For this reason the second method apparently goes by the name SAM -- Signed Area Method. (Now you need to think about the interpretation of the dot product in a similar way).

Note that the second method is quite general, while the first method only works for objects which have perpendicular surfaces (at least, in its current form presented here). On the other hand, the first method is significantly faster.

Now that we've hidden the faces, it's time to fill them:

Filled Faces  
-----

Q: How do you make a statue of an elephant?

A: Start with a block of granite and carve away everything that doesn't look like elephant!

The first method of filling faces is very simple in concept. Let's say we want a cube with white faces and black edges. Before, the program would make the buffer black and then draw in the white edges. The idea here is to make the entire buffer white, draw the edges in black, and then make everything outside of the edges black. Quite simply, we start with a solid block and then take away everything that doesn't look like a cube! You can also think of it like a cookie cutter: we press our cube-shaped cutter down and remove all the dough outside of the cutter.

This simplistic method actually has some advantages. If the object is very large, we spend very little time doing the actual un-filling. We don't care about how complicated the object is, because we just trace out the edge. Finally, this gives us an extremely easy way of implementing a rudimentary texture-mapping in multicolor mode. For instance, instead of coloring the block white, what if we used a changing pattern of colors? As long as the edge is a specific color, the pattern can be any combination of the other three colors. An example program which does just this is included -- note that the initialization program needs to be changed slightly to run this program.

In other words, we roll the dough, draw a pattern into it, press our cutter down and remove the outside dough. We are left with a cube with patterns all over it.

On the downside it's not quite so easy to do things like have each face a separate color (but who wants wimpy separate colors when you can have evolving texture patterns, eh? :).

The program makes a few refinements to this technique. For instance, instead of coloring the entire buffer white, it calculates ahead of time the minimum and maximum values for y, and only colors that part of the drawing area white.

For the sake of completeness, here is another method of filling: exclusive-or. A clever way of filling faces is to use some EOR magic. Let's say we want to fill everything between two points A and B. We want to start filling at point A and stop at point B, and since EOR is a bit flipper this gives us a means of filling. Consider the following situation in memory:

```
00010000 <-- Point A
00000000
00000000
00010000 <-- Point B
```

Now consider the following little piece of code:

```
LDA #00
EOR A
STA A
EOR A+1
STA A+1
EOR A+2
STA A+2
EOR A+3 ;point B
```

The result is:

```
00010000
00010000
00010000
00010000
```

This is the conceptual idea behind an EOR-buffer. Pretty neat, eh? But we can't just implement this as-is. In fact we have a whole slew of things to worry about now. Try EORing a vertical line. What about when two lines share a single pixel at their intersection? What happens in color?

Ah reckon y'all will just have to wait until next time tuh see :).

Da Program

-----

Let's review the code. We check to see if we should increase or decrease the rotation rate (or quit), and then update the angles. Next we calculate the rotation matrix using a table of sines and cosines. Then we rotate and project the points by using a table of  $d/(z/64-z_0)$  values. Somewhere in there we clear a working buffer, draw all of the lines, swap the buffers, pass Go, collect \$200 (actually, considering where the buffers are located we either collect \$300 or \$380 :), and go around the loop again.

First, some bugs. There were two places in the line drawing routine where an SBC was performed with the carry clear when it should have been set, so we need to add some SECs in there. Somewhere there is a strange bug related to y-rotations, but I didn't track it down.

Although not a bug, there is something to think about. On the computer, x increases to the right, and y-increases downwards, with z coming out of the screen. But this is a left-handed coordinate system, and all our calculations were performed in a right-handed coordinate system. What this means is that one of our coordinates is actually a mirror-image of what it should be, while the other coordinate is where it is supposed to be.

Remember that a projection generates a negative mirror-image of the object -  
 -  
 the computer coordinate system mirrors a single axis of the image again!

Because of the symmetry of a cube, this makes no difference. A smart way to fix this is to translate the object in `_front_` of the projection plane, i.e. to use the translation  $z=z+c$  instead of the currently used  $z=z-c$ , but still project through the origin and into the plane  $z=1$ . Since I am not particularly smart though, not to mention lazy and unmotivated, I didn't bother to fix this.

Before we start adding the new stuff like hidden surfaces into the code, why don't we think about doing some simple optimizations to the old code? One really easy thing to fix is in the projection routine. You will recall that the earlier program rotated  $z$  and then added 128 to it to use as an index. Why bother to add 128 at all? I dunno -- sometimes things seem like a good idea at the time. So that's something to fix. It's not that it's a big waste of time, it's just one of those annoying things that there's no reason for.

How about the variables? They're all just sitting at the end of the program -- why not move them all into zero page? Sounds good to me! We just need to make sure we don't use any sensitive locations in zero page that will hose the whole computer. So now that's fixed.

On the C64 an interrupt is performed every 60th of a second which scans the keyboard and things like that -- why in the world do we want that running in the middle of all our calculations? I dunno -- let's turn it off (but turn it back on before checking to see if F1 etc. was pressed!).

A footnote observation: when the rotation matrix is calculated, two macros are used (MUL2 and DIV2) which multiply and divide a signed number by two. It never ceases to amaze me what happens when you simply sit down and think about something, and in this case these two macros can be made much simpler:

```
MUL2 ASL ;That's all, folks

DIV2 CLC
 BPL :POS
 SEC
 :POS ROR
```

These two routines will multiply/divide a signed 2's complement number by two (note that the source included with this article uses the old method).

There's the easy stuff to fix. What about the calculations themselves? The rotation is pretty straightforward -- nah, skip that. The line drawing routine takes up an awful lot of time -- maybe we can speed that up? That's for a future article :). Clearing the buffer takes a lot of time, but now that we're going to have filled faces there isn't too much we can do about that. In fact, so much more time is spent in those two areas than is spent

in other parts of the code that any other optimizations we make really aren't going to make a very big difference... BUT...

How about multiplications?

Fast Signed Multiply

-----

Ah, now here is something we can fix. Consider the following function:

$$f(x) = x*x/4$$

Now notice that

$$f(a+b) - f(a-b) = a*b$$

Wowsers! All we need to do is have a table of squares, and we can do a multiplication in no time!

Whoa there, wait a minute, all of our calculations are using signed numbers. Won't that make a difference? Well, the above calculation is completely general -- I never said what the sign of a and b are. The fact that we are using two's complement notation makes this even simpler!

Recall that our multiplication is

$$x \rightarrow x * [d/(z_0 - z/64)]$$

where x is a signed floating point number multiplied by 64 (that is, instead of going from -1 to 1 x would go from -64 to 64). Previously we made d large, so that the table of d/(z-z<sub>0</sub>) would be more accurate. Then we multiplied the numbers together and divided by 64, a procedure which took between 150 and 180 cycles, leaving us with a signed, 8-bit result.

Well, that's easy to duplicate. From our first equation above, we see that

$$\begin{aligned} (a*b)/64 &= [ f(a+b) - f(a-b) ]/64 \\ &= g(a+b) - g(a-b) \end{aligned}$$

where

$$g(x) = x*x/256.$$

In other words, if we modify our table slightly, we get exactly the result we want. So here is the code to multiply two numbers together:

\* A\*Y -> A Signed, 8-bit result

```
STA ZP1 ;ZP1 -- zero page pointer to table of g(x)
```

```

EOR #$FF
CLC
ADC #$01
STA ZP2 ;ZP2 also points to g(x)
LDA (ZP1),Y ;g(Y+A)
SEC
SBC (ZP2),Y ;g(Y-A)

```

And that's it -- we're done. The above takes 24-26 cycles to execute -- not bad at all! Yes, with another table we could make it even faster, but this is good enough for us.

At the moment we don't do very many multiplications, but in the future, when we write a generalized routine to rotate and project an arbitrary object, this will give us a humongous time savings.

Astute readers may be thinking ahead here: in the program, for each projection we have two multiplications,  $x=x*c$  and  $y=y*c$ , where  $c$  is the same in both cases. So if we store  $c$  in  $ZP1$  and  $ZP2$ , we can make the multiplication even more efficient, right? The answer is yes, but only by being extremely careful, for reasons that will be detailed in exactly two paragraphs.

BUT WAIT! We have to think about a few things here. What happens when we multiply, say,  $-1 * -1$ . In two's complement notation  $-1$  is equal to 255. So

our above algorithm adds 255 to 255 to get an index into the table and gets... oops! Our table needs to be larger than 256 bytes! In fact this is very easy to fix, because of the way two's complement works. All we need to do is put an exact copy of the 256 byte table on top of itself, and the table

will work fine. (If you look at the initialization program you will notice that the statement is: `q%=s*s:poke bm+j,q%:poke bm+j+256,q%`).

BUT WAIT!!! What kinds of numbers are we multiplying together here? Our vertices start out at the points  $(+/-1,+/-1,+/-1)$ . Our rotations correspond to moving these points around on a sphere, so it is easy to see that the largest rotated value we can have is  $\text{sqr}(3)$ , the radius of this sphere. Since we are multiplying these numbers by 64, the largest value we can have for these numbers is  $64*\text{sqr}(3) = 111$ . Okay, no big whoop, what are the values for  $d/(z_0-z/64)$  anyways? Well, for  $z_0=5$  and  $d=150$  say we get values like 28...

ACK! When we go to multiply we are going to add 111 to 28 and get 137, but in two's complement notation this is equal to -119.

Example:

`a=28`

`b=111`

`f(b+a) = f(137) = f(-119)` in two's-complement notation

`f(b-a) = f(83)`

In our table lookup we really want  $137*137$  but we are going to get  $-119*-119!$

One option is to never choose  $d$  very large so that we don't have this problem, but the solution turns out to be much simpler, again due to the way two's complementing works.

We can see that we can get numbers larger than 127 when we add the two multiplicands together. What is the `_smallest_` number we will come up with? Certainly the smallest  $x$  is going to get is  $-111$ . Ahhh...  $d/(z_0-z/64)$  is always positive, so when we add them together we will get something larger than  $-111$ , which in two's complement notation is 145. This means that we can treat the table entries between 127 and at least 145 as positive numbers instead of two's complement negative numbers, and everything will work out great.

Incidentally, fudging this table provides an easy way to add pretty cool special effects. The initialization program sets up the math table using the following line:

```
[for j=0 to 255]
290 S=J:IF S>150 THEN S=256-S
 [poke bm+j,S*S]
```

Try changing the inequality from  $S>150$  to  $S>120$  (or  $S>127$ , where it would be for a two's complement table), and see what happens!

And this is why we can't store  $d/(z_0-z)$  in the pointers ZP1 and ZP2 -- if we did, then for a given multiplication we could get numbers larger than 127 and smaller than  $-128$ , and our clever table would no longer work right. We can still get around this -- all we need is two clever tables, one for the positive  $d/(z_0-z)$  and one for negative  $d/(z_0-z)$ . For the first table, we have the situation outlined above: no numbers smaller than  $-90$  or so, and numbers possible larger than 127. For the second table we have the reverse situation: no numbers larger than 90 or so, but possible numbers less than  $-128$ . Since we are using two pointers anyways (ZP1 and ZP2), this is not difficult to implement.

The end result is that you can do the entire projection in around 36 cycles if you so desire. 36 cycles? Note that for the second table the code does something like `EOR #$FF; CLC; ADC #$01`. Well, if we set up the second table as  $f(x)=(x+1)^2/4$  then we have included the `CLC` and `ADC #$01` into the table, so the instructions can be removed. The entire projection routine is then:

```
... (rotate z)
STA ZP1
EOR #$FF
STA ZP2
... (rotate x)
```

```

TAY
LDA (ZP1),Y
SEC
SBC (ZP2),Y ;Now A contains projected X
... (rotate y)
TAY
LDA (ZP1),Y
SEC
SBC (ZP2),Y

```

Looks like 36-40 cycles to me! The program doesn't implement this -- it only uses a single table, and repeats the STA ZP1 stuff at each step. A few cycles wasted won't kill us (there are plenty of cycles wasted in the code), and it is probably tricky enough to follow as it is.

You might be asking, what is the true minimum value for a given  $z_0$  and  $d$ ? Well, I tried writing down a set of equations and minimizing according to some constraints, and I ended up with a sixth-order polynomial which I had to write little newton-iteration solver for. In other words, I don't think you can write down a simple function of  $d$  and  $z_0$  to give the table boundaries. I found 150 to be a perfectly reasonable number to use.

Incidentally, this is why the projection was not changed to  $z=z+c$  -- I didn't want to go fiddling around with it again. Maybe next time :).

ONE MORE THING!!! This is very important. The math table MUST be on an even boundary for the above algorithm to work correctly. This one is easy to get bit by.

### Logarithmic Multiplication

-----

As long as we're talking about fast multiplication here, it is worthwhile to mention another method for multiplying two numbers together. To understand it you need to understand two important properties of logarithms:

$$\log(x*y) = \log(x) + \log(y)$$

$$\log_b(x) = y \iff b^y = x$$

These are a reflection of the fact that logarithms are inverses of exponentiation. So you can see that another way to multiply two numbers together is to take their logs, add, and then exponentiate the result. So you could have a table of  $\log_2$  (base 2 logarithms) and another table of  $2^x$ , and do a multiplication very quickly. (Actually, you'd want a table of  $32*\log_2(x)$ , since  $\log_2(256)=8$ ). Why wasn't this method used?



First, dealing with signed numbers is much trickier -- the logarithm of a negative number is a complex (i.e. real+imaginary) number, complete with branch cuts. You can get around this by setting up the tables in a special way (for instance by letting  $\log(-x)=-\log(x)$ ) and putting in some special handling, but it isn't as efficient as the algorithm used in the program.

Second, accuracy decreases significantly as x and y get large, so that for an eight-bit table of logarithms you will often get an answer that is off by one or more. You can in fact get around this problem by using some sneaky manipulation -- if you are interested in seeing this, contact us!

But it is worthwhile to keep this method in mind if you need a really fast multiplication and you aren't too worried about accuracy.

Christopher Jam (phillips@ee.uwa.edu.au) has come up with an interesting variation on this method. It calculates  $64+64*x/z$  and uses a slightly different structure for the signed numbers, and runs almost as fast as the method used by the program -- contact him for more information if you're interested.

### Hidden Surfaces

-----

The remainder of this follows right from the discussion section. In the program the cube vertices are labeled as

- P1 = 1,1,1
- P2 = 1,-1,1
- P3 = -1,-1,1
- P4 = -1,1,1
- P5 = 1,1,-1
- P6 = 1,-1,-1
- P7 = -1,-1,-1
- P8 = -1,1,-1

and the faces are chosen to be

- Face 1: P1 P2 P3 P4
- 6: P5 P6 P7 P8
- Face 2: P1 P2 P5 P6
- 5: P3 P4 P7 P8
- Face 3: P1 P4 P8 P5
- 4: P2 P3 P6 P7

(think of it as a six-sided dice, with six opposite of one, etc.). The normal vectors are then

Face 1: P1-P5  
Face 2: P1-P4  
Face 3: P1-P2

This means that we need to store the z-coordinates for points 1,2,4, and 5. Note that the opposite faces have exactly opposite normal vectors, so that for instance the normal vector for face 6 is P5-P1, the negative of face 1.

Here is something to consider: when one face is visible, the opposite face cannot be visible! Because of the way projections work, though, the converse is not true; it is entirely possible to have two opposite faces invisible. To prove this to yourself just look at your favorite box, like your monitor, straight-on, and notice that you can't see the sides!

All that the program does is subtract z-coordinates and add them to the constant K, and check the sign. Unfortunately we can have a positive overflow while adding stuff together (since these are signed numbers), and if we don't catch the positive overflow we will calculate a negative result when the result is actually positive! This will of course wreck the hidden surface removal.

#### Filled Faces

-----

The program currently uses the first algorithm to fill faces, i.e. the cookie-cutter elephant-carving method. During the projections the program checks each value of y to find the minimum and maximum values for this plot, ymin and ymax. The program then clears the buffer up to and including ymin, fills the buffer from ymin+1 to ymax-1, and then clears the rest of the buffer. Why does it clear ymin and ymax? Because the only thing that can happen on those lines is an edge -- there is no point in filling these lines and then clearing them, since they will always be clear. By only filling the buffer between ymin and ymax, we save some time in removing the junk from the edges of the cube.

Next, the cube is drawn. The background is black and the faces are white, i.e. our fill color is white. Clearly then we want to draw our lines in black. I could have reversed background and foreground colors and left the line routine as-is, but of course being the lazy programmer I am I decided instead to change the table BITP. You may recall that the earlier table had entries like %10000000 %01000000 etc. Now it has entries like %01111111 %10111111 etc., and instead of ORAing the values into the buffer, they are ANDed into the buffer. This then draws lines of zeroes into our buffer which is solid ones.

Finally, to un-fill the outside of the cube the program simply goes through the buffer from ymin to ymax, coloring everything black until it hits a zero, i.e. an edge. At this point it calculates the appropriate pattern to clear up to the edge, and then does the same thing starting from the right hand side of the buffer. In other words it runs along a specific y-value coloring everything black until it hits the edge of the cube, and does this for all the relevant y-values.

## Texture Mapping

-----

More of a fill-pattern really. The program cube3d2.1.o does all of the above but in multicolor mode. Now instead of using a solid color to fill the buffer the program uses a series of colored lines -- really a very simple pattern. A much neater thing would be to have a pattern drawn out in a pattern buffer, and to copy that into the drawing buffer. Other things to try are colored squares which shift around. cube3d2.1.o is just a really quick hack, but at least it demonstrates the concept.

MAKE SURE that you change the value of D from 170 to 85 if you try this program! Pixels are doubled now, so that resolution is cut in half. This is located at line 240 in INIT3D2.0

## Memory Map

-----

The main program is located at \$8000=32768 and is 3200 bytes long.

|               |                                         |
|---------------|-----------------------------------------|
| \$8000-\$8C00 | - Program                               |
| \$8C00-\$8C80 | - Bit position table                    |
| \$8C80-\$8D00 | - Table of sines                        |
| \$8D00-\$8D80 | - Table of cosines.                     |
| \$8D80-\$8E80 | - Table of $d/(z_0-z/64)$               |
| \$8F00-\$9100 | - Two 256-byte tables of $g(x)=x*x/256$ |
| \$3000        | - First drawing buffer                  |
| \$3800        | - Second drawing buffer                 |

INIT3D is a simple basic program to set up the tables. For INIT3D2.x the important setup routines are:

|               |                                         |
|---------------|-----------------------------------------|
| lines 100-150 | - Set up the trigonometric tables       |
| lines 233-310 | - Set up the projection and mult tables |
| 240           | - Location of constants D and Z0        |
| 290           | - Set table boundary for multiplication |

That's all -- until next time...

Steve Judd George Taylor 12/2/95

This document is Copyright 1995 by Stephen Judd and George Taylor. Much like the previous one. It is also freely distributable.

And here is the source code:

```

*
* Stephen Judd
* George Taylor
* Started: 7/11/94
* Finished: 7/19/94
* v2.0 Completed: 12/17/94
*
* Well, if all goes well this
* program will rotate a cube.
*
* v2.0 + New and Improved!
* Now with faster routines,
* hidden surfaces, filled
* faces, and extra top secret
* text messages!
*
* This program is intended to
* accompany the article in
* C=Hacking, Jan. 95 issue.
* For details on this program,
* read the article!
*
* Write to us!
*
* Myself when young did
* eagerly frequent
* Doctor and Saint, and heard
* great Argument
* About it and about: but
* evermore
* Came out by the same Door
* as in I went.
* - Rubaiyat
*
* Though I speak with the
* tongues of men and of angles
* and have not love, I am
* become as sounding brass, or
* a tinkling cymbal.
* - 1 Corinthians 13

```



```

P2X = $94
P2Y = $95 ;They are here so that we
P3X = $96 ;don't have to recalculate them.
P3Y = $AE
P4X = $AF ;They make life easy.
P4Y = $B0
P5X = $B1 ;Why are you looking at me like that?
P5Y = $B2 ;Don't you trust me?
P6X = $B3
P6Y = $B4 ;Having another child wasn't my idea.
P7X = $71
P7Y = $50
P8X = $51
P8Y = $52
P1Z = $57 ;These are z-coordinates
P2Z = $58 ;We only need these four to check
P4Z = $59 ;for hidden faces
P5Z = $60
DSX = $61 ;DSX is the increment for
 ;rotating around x
 ;Similar for DSY, DSZ
DSY = $62
DSZ = $63
SX = $64 ;These are the actual angles in x y and z
SY = $65
SZ = $66
T1 = $67 ;These are used in the rotation
T2 = $68
T3 = $69 ;See the article for more details
T4 = $6A
T5 = $6B
T6 = $6C
T7 = $6D
T8 = $6E
T9 = $6F
T10 = $70
A11 = $A5 ;These are the elements of the rotation matrix
B12 = $A6 ;XYZ
C13 = $A7
D21 = $A8 ;The number denotes (row,column)
E22 = $A9
F23 = $AA
G31 = $AB
H32 = $AC
I33 = $AD

```

\*\*\* Macros

```

MOVE MAC
 LDA]1
 STA]2

```

```

 <<<

GETKEY MAC ;Wait for a keypress
WAIT JSR GETIN
 CMP #00
 BEQ WAIT
 <<<

*-----

 LDA #$00
 STA BKGND
 STA BORDER
 LDA VMCSB
 AND #%00001111 ;Screen memory to 1024
 ORA #%00010000
 STA VMCSB

 LDY #00
 LDA #<TTEXT
 STA TEMP1
 LDA #>TTEXT
 STA TEMP2
 JMP TITLE
TTEXT HEX 9305111111 ;clear screen, white, crsr dn
 TXT ' cube3d v2.0',0d,0d
 TXT ' by',0d
 HEX 9F ;cyan
 TXT ' stephen judd'
 HEX 99
 TXT ' george taylor',0d,0d
 HEX 9B
 TXT ' check out the jan. 95 issue of',0d
 HEX 96
 TXT ' c=hacking'
 HEX 9B
 TXT ' for more details!',0d
 HEX 0D1D1D9E12
 TXT 'f1/f2',92
 TXT ' - inc/dec x-rotation',0d
 HEX 1D1D12
 TXT 'f3/f4',92
 TXT ' - inc/dec y-rotation',0d
 HEX 1D1D12
 TXT 'f5/f6',92
 TXT ' - inc/dec z-rotation',0d
 HEX 1D1D12
 TXT 'f7',92
 TXT ' resets',0d
 TXT ' press q to quit',0d
 HEX 0D05

```

```

 TXT ' press any key to begin',0d
 HEX 00
TITLE LDA (TEMP1),Y
 BEQ :CONT
 JSR CHROUT
 INY
 BNE TITLE
 INC TEMP2
 JMP TITLE
 TXT 'This is a secret text message!'
:CONT >>> GETKEY

**** Set up tables(?)

* Tables are currently set up in BASIC
* and by the assembler.

TABLES LDA #>TMATH
 STA Z1+1
 STA Z2+1

**** Clear screen and set up "bitmap"
SETUP LDA #$01 ;White
 STA $D021 ;This is done so that older
 LDA #147 ;machines will set up
 JSR CHROUT
 LDA #$00 ;correctly
 STA $D021
 LDA #<SSTART
 ADC #12 ;The goal is to center the graphics
 STA TEMP1 ;Column 12
 LDA #>SSTART ;Row 9
 STA TEMP1+1 ;SSTART points to row 9
 LDA #00
 LDY #00
 LDX #00 ;x will count 16 rows for us
 CLC

:LOOP STA (TEMP1),Y
 INY
 ADC #16
 BCC :LOOP
 CLC
 LDA TEMP1
 ADC #40 ;Need to add 40 to the base pointer
 STA TEMP1 ;To jump to the next row
 LDA TEMP1+1
 ADC #00 ;Take care of carries
 STA TEMP1+1
 LDY #00
 INX

```



```
TXA ;X is also an index into the character number
CPX #16
BNE :LOOP ;Need to do it 16 times

**** Set up buffers

LDA #<BUFF1
STA BUFFER
LDA #>BUFF1
STA BUFFER+1
STA ZTEMP ;ztemp will make life simple for us
LDA VMCSB
AND #%11110001 ;Start here so that swap buffers will work right
ORA #%00001110
STA VMCSB

**** Set up initial values

INIT LDA #00
 STA DSX
 STA DSY
 STA DSZ
 STA SX
 STA SY
 STA SZ

*-----
* Main loop

**** Get keypress

MAIN
KPRESS CLI
 JSR GETIN
 CMP #133 ;F1?
 BNE :F2
 LDA DSX
 CMP #ANGMAX/2 ;No more than pi
 BEQ :CONT
 INC DSX ;otherwise increase x-rotation
 JMP :CONT
:F2 CMP #137 ;F2?
 BNE :F3
 LDA DSX
 BEQ :CONT
 DEC DSX
 JMP :CONT
:F3 CMP #134
 BNE :F4
 LDA DSY
 CMP #ANGMAX/2
```

```

 BEQ :CONT
 INC DSY ;Increase y-rotation
:F4 JMP :CONT
 CMP #138
 BNE :F5
 LDA DSY
 BEQ :CONT
 DEC DSY
:F5 JMP :CONT
 CMP #135
 BNE :F6
 LDA DSZ
 CMP #ANGMAX/2
 BEQ :CONT
 INC DSZ ;z-rotation
:F6 JMP :CONT
 CMP #139
 BNE :F7
 LDA DSZ
 BEQ :CONT
 DEC DSZ
:F7 JMP :CONT
 CMP #136
 BNE :Q
 JMP INIT
:Q CMP #'q' ;q quits
 BNE :CONT
 JMP CLEANUP

:CONT SEI ;Speed things up a bit

**** Update angles

UPDATE CLC
 LDA SX
 ADC DSX
 CMP #ANGMAX ;Are we >= maximum angle?
 BCC :CONT1
 SBC #ANGMAX ;If so, reset
:CONT1 STA SX
 CLC
 LDA SY
 ADC DSY
 CMP #ANGMAX
 BCC :CONT2
 SBC #ANGMAX ;Same deal
:CONT2 STA SY
 CLC
 LDA SZ
 ADC DSZ
 CMP #ANGMAX

```

```

 BCC :CONT3
 SBC #ANGMAX
:CONT3 STA SZ

**** Rotate coordinates

ROTATE

*** First, calculate t1,t2,...,t10

** Two macros to simplify our life
ADDA MAC ;Add two angles together
 CLC
 LDA J1
 ADC J2
* Use two trig tables to remove the below CMP etc. code
 CMP #ANGMAX ;Is the sum > 2*pi?
 BCC DONE
 SBC #ANGMAX ;If so, subtract 2*pi
DONE <<<

SUBA MAC ;Subtract two angles
 SEC
 LDA J1
 SBC J2
 BCS DONE
 ADC #ANGMAX ;Oops, we need to add 2*pi
DONE <<<

** Now calculate t1,t2,etc.

>>> SUBA,SX ;SZ
STA T1 ;t1=sy-sz
>>> ADDA,SX ;SZ
STA T2 ;t2=sy+sz
>>> ADDA,SX ;SZ
STA T3 ;t3=sx+sz
>>> SUBA,SX ;SZ
STA T4 ;t4=sx-sz
>>> ADDA,SX ;T2
STA T5 ;t5=sx+t2
>>> SUBA,SX ;T1
STA T6 ;t6=sx-t1
>>> ADDA,SX ;T1
STA T7 ;t7=sx+t1
>>> SUBA,T2 ;SX
STA T8 ;t8=t2-sx
>>> SUBA,SX ;SX
STA T9 ;t9=sy-sx
>>> ADDA,SX ;SY
STA T10 ;t10=sx+sy

```

```

* Et voila!

*** Next, calculate A,B,C,...,I

** Another useful little macro
DIV2 MAC ;Divide a signed number by 2
 ;It is assumed that the number
 BPL POS ;is in the accumulator
 CLC
 EOR #$FF ;We need to un-negative the number
 ADC #01 ;by taking it's complement
 LSR ;divide by two
 CLC
 EOR #$FF
 ADC #01 ;Make it negative again
 JMP DONEDIV
POS LSR ;Number is positive
DONEDIV <<<

MUL2 MAC ;Multiply a signed number by 2
 BPL POSM
 CLC
 EOR #$FF
 ADC #$01
 ASL
 CLC
 EOR #$FF
 ADC #$01
 JMP DONEMUL
POSM ASL
DONEMUL <<<

** Note that we are currently making a minor leap
** of faith that no overflows will occur.

:CALCA CLC
 LDX T1
 LDA COS,X
 LDX T2
 ADC COS,X
 STA A11 ;A=(cos(t1)+cos(t2))/2
:CALCB LDX T1
 LDA SIN,X
 SEC
 LDX T2
 SBC SIN,X
 STA B12 ;B=(sin(t1)-sin(t2))/2
:CALCC LDX SY
 LDA SIN,X
 >>> MUL2
 STA C13 ;C=sin(sy)

```

```

:CALCD SEC
 LDX T8
 LDA COS,X
 LDX T7
 SBC COS,X
 SEC
 LDX T5
 SBC COS,X
 CLC
 LDX T6
 ADC COS,X ;Di=(cos(t8)-cos(t7)+cos(t6)-cos(t5))/2
 >>> DIV2
 CLC
 LDX T3
 ADC SIN,X
 SEC
 LDX T4
 SBC SIN,X
 STA D21 ;D=(sin(t3)-sin(t4)+Di)/2
:CALCE SEC
 LDX T5
 LDA SIN,X
 LDX T6
 SBC SIN,X
 SEC
 LDX T7
 SBC SIN,X
 SEC
 LDX T8
 SBC SIN,X ;Ei=(sin(t5)-sin(t6)-sin(t7)-sin(t8))/2
 >>> DIV2
 CLC
 LDX T3
 ADC COS,X
 CLC
 LDX T4
 ADC COS,X
 STA E22 ;E=(cos(t3)+cos(t4)+Ei)/2
:CALCF LDX T9
 LDA SIN,X
 SEC
 LDX T10
 SBC SIN,X
 STA F23 ;F=(sin(t9)-sin(t10))/2
:CALCG LDX T6
 LDA SIN,X
 SEC
 LDX T8
 SBC SIN,X
 SEC
 LDX T7

```

```

 SBC SIN,X
 SEC
 LDX T5
 SBC SIN,X ;Gi=(sin(t6)-sin(t8)-sin(t7)-sin(t5))/2
 >>> DIV2
 CLC
 LDX T4
 ADC COS,X
 SEC
 LDX T3
 SBC COS,X
 STA G31 ;G=(cos(t4)-cos(t3)+Gi)/2
:CALCH CLC
 LDX T6
 LDA COS,X
 LDX T7
 ADC COS,X
 SEC
 LDX T5
 SBC COS,X
 SEC
 LDX T8
 SBC COS,X ;Hi=(cos(t6)+cos(t7)-cos(t5)-cos(t8))/2
 >>> DIV2
 CLC
 LDX T3
 ADC SIN,X
 CLC
 LDX T4
 ADC SIN,X
 STA H32 ;H=(sin(t3)+sin(t4)+Hi)/2
:WHEW CLC
 LDX T9
 LDA COS,X
 LDX T10
 ADC COS,X
 STA I32 ;I=(cos(t9)+cos(t10))/2

** It's all downhill from here.
 JMP DOWNHILL
 TXT 'Gee Brain, what do you want to do '
 TXT 'tonight?'

** Rotate, project, and store the points
DOWNHILL

* A neat macro
NEG MAC ;Change the sign of a two's complement
 CLC
 LDA]1 ;number.
 EOR #$FF

```

```

 ADC #$01
 <<<

*-----
* These macros replace the previous projection
* subroutine.

SMULT MAC ;Multiply two signed 8-bit
 ;numbers: A*Y/64 -> A

 STA Z1
 CLC
 EOR #$FF
 ADC #$01
 STA Z2
 LDA (Z1),Y
 SEC
 SBC (Z2),Y
 <<< ;All done :)

ADDSUB MAC ;Add or subtract two numbers
 ;depending on first input
 IF -=]1 ;If subtract
 SEC ;then use this code
 SBC]2
 ELSE ;otherwise use this code
 CLC
 ADC]2
 FIN
 <<<

PROJECT MAC ;The actual projection routine
 ;two inputs are used (x,y)
 ;corresponding to (+/-1,+/-1)
 ;The third input is used to
 ;determine if the rotated
 ;z-coordinate should be
 ;stored, and if so where.
 ;The calling routine handles
 ;changing the sign of z.

 LDA I33 ;Calculate rotated z:
 >>> ADDSUB,]1 ;G31 ;Add or subtract x
 >>> ADDSUB,]2 ;H32 ;Add or subtract y
 IF P,]3 ;Do we need to store the point?
 STA]3 ;Then do so!
 FIN

* EOR #128 ;We are going to take 128+z
 TAX ;Now it is ready for indexing
 LDA ZDIV,X ;Table of d/(z+z0)

```

```

TAY ;Y now contains projection

LDA C13 ;Now calculate rotated x
>>> ADDSUB,]1 ;A11
>>> ADDSUB,]2 ;B12
>>> SMULT ;Signed multiply A*Y/64->A
CLC
ADC #64 ;Offset the coordinate
TAX ;Now X is rotated x!

LDA F23 ;Now it's y's turn
>>> ADDSUB,]1 ;D21
>>> ADDSUB,]2 ;E22
>>> SMULT
CLC
ADC #64 ;Offset
CMP YMIN ;Figure out if it is a
BCS NOTMIN ;min or max value for y
STA YMIN
BCC NOTMAX ;This is used in calculating
NOTMIN ;the filled faces
CMP YMAX
BCC NOTMAX
STA YMAX
NOTMAX ;Not really necessary

<<<< ;All done

LDA #64 ;Reset Ymin and Ymax
STA YMIN
STA YMAX

* P1=[1 1 1]
>>> PROJECT,1;1;P1Z ;Rotated z stored in P1Z
STX P1X
STY P1Y

* P2=[1 -1 1]
>>> PROJECT,1 ;-1;P2Z
STX P2X
STY P2Y

* P3=[-1 -1 1]
>>> PROJECT,-1;-1;NOPE ;Don't store z-value
STX P3X
STY P3Y

* P4=[-1 1 1]
>>> PROJECT,-1;1;P4Z
STX P4X
STY P4Y

* P8=[-1 1 -1]
>>> NEG,C13
STA C13

```



```

 >>> NEG,F23
 STA F23
 >>> NEG,I33
 STA I33
 >>> PROJECT,-1;1;NOPE
 STX P8X
 STY P8Y
* P7=[-1 -1 -1]
 >>> PROJECT,-1;-1;NOPE
 STX P7X
 STY P7Y
* P6=[1 -1 -1]
 >>> PROJECT,1;-1;NOPE
 STX P6X
 STY P6Y
* P5=[1 1 -1]
 >>> PROJECT,1;1;P5Z
 STX P5X
 STY P5Y

* A little macro

SETBUF MAC ;Put buffers where they can be hurt
 LDA #00
 STA BUFFER
 LDA ZTEMP ;ztemp contains the high byte here
 STA BUFFER+1
 <<<

**** Clear buffer

* >>> SETBUF
*CLRBUF LDA #$00 ;Pretty straightforward,
* LDX #$08 ;I think
* LDY #$00
*:LOOP STA (BUFFER),Y
* INY
* BNE :LOOP
* INC BUFFER+1
* DEX
* BNE :LOOP

* This is the new and improved buffer clear
* routine for filled faces

 >>> SETBUF
 STA TEMP1+1 ;buffer2 will point to
 LDA #$80 ;buffer+128
 STA TEMP1 ;Makes life faster for us
FILCLR LDA #00
 LDX #$08 ;We'll do it two at a time

```

```

 LDY #$00
:LOOP1 STA (BUFFER),Y
 STA (TEMP1),Y
 INY
 CPY YMIN
 BNE :LOOP1
 LDA #$FF ;Now load with fills
:LOOP2 STA (BUFFER),Y
 STA (TEMP1),Y
 INY
 CPY YMAX
 BCC :LOOP2
 LDA #$00 ;Black out the rest
:LOOP3 STA (BUFFER),Y
 STA (TEMP1),Y
 INY
 BPL :LOOP3 ;Until Y=128
 LDY #00
 INC BUFFER+1
 INC TEMP1+1
 DEX
 BNE :LOOP1 ;Go all the way around

**** Now draw the lines.
**** But first check for hidden faces!
**** Remember: P1=[1 1 1] P2=[1 -1 1] P3=[-1 -1 1]
**** P4=[-1 1 1] P5=[1 1 -1] P6=[1 -1 -1] P7=[-1 -1 -1]
**** P8=[-1 1 -1]

LINES LDA #00
 STA FACES ;Hidden face counter
:FACE1 LDA K
 SEC
 SBC P1Z
 BVS :FACE6 ;Overflow already?
 CLC
 ADC P5Z ;Is k-v1z < 0?
 ;If not, face is invisible
 BVC :DRAW1 ;But we might have overflow
 LDA P5Z ;Was overflow pos or neg?
:DRAW1 BPL :FACE6 ;If pos then k-v1z > 0

 LDA #$01 ;Otherwise, draw the
 STA FACES ;face!

 LDA P1X
 STA TX1
 LDA P1Y
 STA TY1
 LDA P2X
 STA TX2

```

```

LDA P2Y
STA TY2
JSR DRAW ;P1-P2

LDA P3X
STA TX1
LDA P3Y
STA TY1
JSR DRAW ;P2-P3

LDA P4X
STA TX2
LDA P4Y
STA TY2
JSR DRAW ;P3-P4

LDA P1X
STA TX1
LDA P1Y
STA TY1
JSR DRAW ;P4-P1 Face 1 done.
JMP :FACE2 ;If one is visible, the other
 ;isn't.
:FACE6 LDA K
SEC
SBC P5Z
BVS :FACE2
CLC
ADC P1Z ;Now check if K-v6z < 0
BVC :DRAW6 ;Love that overflow
LDA P1Z
:DRAW6 BPL :FACE2 ;If not, go on

LDA #$20
STA FACES ;otherwise, draw it

LDA P5X
STA TX2
LDA P5Y
STA TY2
LDA P6X
STA TX1
LDA P6Y
STA TY1
JSR DRAW ;P5-P6

LDA P7X
STA TX2
LDA P7Y
STA TY2
JSR DRAW ;P6-P7

```

```

LDA P8X
STA TX1
LDA P8Y
STA TY1
JSR DRAW ;P7-P8

LDA P5X
STA TX2
LDA P5Y
STA TY2
JSR DRAW ;P8-P5

:FACE2 LDA K
SEC
SBC P1Z
BVS :FACE5
CLC
ADC P4Z ;K-v2z < 0?
BVC :DRAW2
LDA P4Z
:DRAW2 BPL :FACE5
LDA #$02 ;If so, draw it!
ORA FACES
STA FACES

LDX P1X ;We're doing this this way
STX TX1 ;to save a few cycles
LDX P1Y
STX TY1

AND #$01 ;Shares an edge with face 1
BNE :F2S2 ;Skip to next edge if present

LDA P2X
STA TX2
LDA P2Y
STA TY2
JSR DRAW ;P1-P2

:F2S2 LDX P5X
STX TX2
LDX P5Y
STX TY2
JSR DRAW ;P1-P5

LDX P6X
STX TX1
LDX P6Y
STX TY1

LDA FACES

```

```

 AND #$20 ;Also shares an edge with 6
 BNE :F2S4

 JSR DRAW ;P5-P6

:F2S4 LDA P2X
 STA TX2
 LDA P2Y
 STA TY2 ;Such is face 2
 JSR DRAW ;P6-P2
 JMP :FACE3 ;Skip 5

:FACE5 LDA K
 SEC
 SBC P4Z
 BVS :FACE3
 CLC
 ADC P1Z ;Same thing again...
 BVC :DRAW5
 LDA P1Z
:F5S2 BPL :FACE3
 LDA #$10
 ORA FACES
 STA FACES

 LDX P3X
 STX TX1
 LDX P3Y
 STX TY1

 AND #$01 ;Shares with 1
 BNE :F5S2

 LDA P4X
 STA TX2
 LDA P4Y
 STA TY2
 JSR DRAW ;P3-P4

:F5S2 LDA P7X
 STA TX2
 LDA P7Y
 STA TY2
 JSR DRAW ;P3-P7

 LDA P8X
 STA TX1
 LDA P8Y
 STA TY1

 LDA FACES

```

```

 AND #$20 ;Shares with 6
 BNE :F5S4

:F5S4 JSR DRAW ;P7-P8
 LDA P4X
 STA TX2
 LDA P4Y
 STA TY2 ;P8-P4
 JSR DRAW ;Two more to go!

:FACE3 LDA K
 SEC
 SBC P1Z
 BVS :FACE4
 CLC
 ADC P2Z
 BVC :DRAW3
 LDA P2Z
:F3S3 BPL :FACE4 ;Ah reckon it's a'hidden, yup
 LDA #$04
 ORA FACES
 STA FACES

 LDX P1X
 STX TX1
 LDX P1Y
 STX TY1

 AND #$01 ;Shares with 1
 BNE :F3S2

 LDA P4X
 STA TX2
 LDA P4Y
 STA TY2
 JSR DRAW ;P1-P4

:F3S2 LDX P5X
 STX TX2
 LDX P5Y
 STX TY2

 LDA FACES
 AND #$02 ;Shares with 2
 BNE :F3S3

:F3S3 JSR DRAW ;P1-P5
 LDX P8X
 STX TX1
 LDX P8Y
 STX TY1

```

```
LDA FACES
AND #$20 ;Shares with 6
BNE :F3S4

:F3S4 JSR DRAW ;P5-P8
LDX P4X
STX TX2
LDX P4Y
STX TY2

LDA FACES
AND #$10 ;Shares with 5
BNE FACEDONE

JSR DRAW ;P8-P4
JMP FACEDONE

:FACE4 LDA K
SEC
SBC P2Z
BVS FACEDONE
CLC
ADC P1Z
BVC :DRAW4
LDA P1Z
:DRAW4 BPL FACEDONE

LDA P2X
STA TX1
LDA P2Y
STA TY1

LDA FACES
AND #$01 ;Shares with 1
BNE :F4S2

LDA P3X
STA TX2
LDA P3Y
STA TY2
JSR DRAW ;P2-P3

:F4S2 LDA P6X
STA TX2
LDA P6Y
STA TY2

LDA FACES
AND #$02 ;Shares with 2
BNE :F4S3
```

```

: F4S3 JSR DRAW ;P2-P6
 LDA P7X
 STA TX1
 LDA P7Y
 STA TY1

 LDA FACES
 AND #$20 ;Shares with 6
 BNE :F4S4

: F4S4 JSR DRAW ;P6-P7
 LDA P3X
 STA TX2
 LDA P3Y
 STA TY2

 LDA FACES
 AND #$10 ;Shares with 5
 BNE FACEDONE

FACEDONE JSR DRAW ;P7-P3
 ;Whew! Time for a beer.

**** Now we need to unfill the outside from the faces
UNFILL LDY YMIN
: LOOP >>> SETBUF
 LDX #08
: L1 LDA (BUFFER),Y
 EOR #$FF ;Go till we find a plotted
 BNE :GOTCHA ;point (i.e. A <> $FF)
* LDA #00 ;Unfilling as we go...
 STA (BUFFER),Y
 LDA #$80
 STA BUFFER
 LDA (BUFFER),Y
 EOR #$FF
 BNE :GOTCHA
* LDA #00
 STA (BUFFER),Y
 STA BUFFER
 INC BUFFER+1
 DEX ;This is our safety valve
 BNE :L1 ;Really shouldn't need it
 JSR CHOKE
 JMP SWAPBUF

: GOTCHA ;A contains the EOR plot value
 STA TEMP1 ;Now find the high bit
 LDA #00
: L2 SEC
 ROL

```



```

 LSR TEMP1 ;Should really use a table
 BNE :L2 ;for this!
 AND (BUFFER),Y
 STA (BUFFER),Y

 LDA ZTEMP ;Now go to the end
 ;Carry is clear
 ;Actually we add 7
 ;16 columns of 128 bytes
 ADC #$06
 STA BUFFER+1
 LDA #$80
 STA BUFFER
:L00P2 LDA (BUFFER),Y ;And work backwards!
 EOR #$FF
 BNE :GOTCHA2
 STA (BUFFER),Y
 STA BUFFER ;Stick a zero into buffer
 LDA (BUFFER),Y
 EOR #$FF
 BNE :GOTCHA2
 STA (BUFFER),Y
 LDA #$80
 STA BUFFER
 DEC BUFFER+1
 BNE :L00P2

:L00TCH2 STA TEMP1 ;Again find the high bit
 LDA #00
:L3 SEC
 ROR
 ASL TEMP1
 BNE :L3
 AND (BUFFER),Y
 STA (BUFFER),Y

 INY ;Now keep going
 CPY YMAX
 BCC :L00P ;Until we hit ymax!
 BEQ :L00P ;We need the last one too.

**** Swap buffers

SWAPBUF LDA VMCSB
 EOR #$02 ;Pretty tricky, eh?
 STA VMCSB
 LDA #$08
 EOR ZTEMP ;ztemp=high byte just flips
 STA ZTEMP ;between $30 and $38

 JMP MAIN ;Around and around we go...

```

```

 TXT 'Same thing we do every night, Pinky: '
 TXT 'try to take over the world!'

```

```
*-----
```

```
* General questionable-value error procedure
```

```

CHOKE LDX #00
:LOOP LDA :CTEXT,X
 BEQ :DONE
 JSR CHROUT
 INX
 JMP :LOOP
:DONE RTS
:CTEXT HEX 0D ;CR
 TXT 'something choked :('
 HEX 0D00

 TXT 'Narf!'

```

```
*-----
```

```
* Drawin' a line. A fahn lahn.
```

```
*** Some useful macros
```

```

PLOTPX MAC ;plot a point in x
 PHA ;Use this one every time
 LDA BITP,X ;X is increased
 BMI C1
 LDA #$80 ;Table has been rearranged
 EOR BUFFER ;for filling faces
 STA BUFFER
 BMI C2
 INC BUFFER+1
C2 LDA #%01111111 ;Note that this is changed
C1 AND (BUFFER),Y ;for plotting filled faces
 STA (BUFFER),Y
 PLA ;Need to save A!
 <<<<

PLOTPTY MAC ;Plot a point in y: simpler and necessary!
 PHA ;Use this one when you just increase Y
 LDA BITP,X ;but X doesn't change
 AND (BUFFER),Y
 STA (BUFFER),Y
 PLA
 <<<<

CINIT MAC ;Macro to initialize the counter
 LDA]1 ;dx or dy
 LSR

```

```

 EOR #$FF ;(Not really two's complement)
 ADC #$01 ;A = 256-dx/2 or 256-dy/2
 <<< ;The dx/2 makes a nicer looking line

XSTEP MAC ;Macro to take a step in X
XLOOP INX
 ADC DY
 BCC L1
* Do we use INY or DEY here?
 IF I,]1 ;If the first character is an 'I'
 INY
 ELSE
 DEY
 FIN
L1 >>> PLOTPX ;Always take a step in X
 CPX X2
 BNE XLOOP
 <<<

YSTEP MAC ;Same thing, but for Y
YLOOP IF I,]1
 INY
 ELSE
 DEY
 CLC ;Very important!
 FIN
 ADC DX
 BCC L2
 INX ;Always increase X
 SBC DY
 >>> PLOTPX
 JMP L3
L2 >>> PLOTPY ;We only increased Y
L3 CPY Y2
 BNE YLOOP
 <<<

**** Initial line setup

DRAW >>> MOVE,TX1 ;X1 ;Move stuff into zero page
 >>> MOVE,TX2 ;X2 ;Where it can be modified
 >>> MOVE,TY1 ;Y1
 >>> MOVE,TY2 ;Y2
 >>> SETBUF ;Now we can clobber the buffer

 SEC ;Make sure x1<x2
 LDA X2
 SBC X1
 BCS :CONT
 LDA Y2 ;If not, swap P1 and P2

```

```

LDY Y1
STA Y1
STY Y2
LDA X1
LDY X2
STY X1
STA X2

SEC
SBC X1 ;Now A=dx
:CONT STA DX
LDX X1 ;Put x1 into X, now we can trash X1

COLUMN LDA X1 ;Find the first column for X
LSR ;(This can be made much faster!)
LSR ;There are x1/8 128 byte blocks
LSR ;Which means x1/16 256 byte blocks
LSR
BCC :EVEN ;With a possible extra 128 byte block
LDY #$80 ;if so, set the high bit
STY BUFFER
CLC
:EVEN ADC BUFFER+1 ;Add in the number of 256 byte blocks
STA BUFFER+1 ;And store it!

SEC
LDA Y2 ;Calculate dy
SBC Y1
BCS :CONT2 ;Is y2>y1?
EOR #$FF ;Otherwise dy=y1-y2
ADC #$01
:CONT2 STA DY
CMP DX ;Who's bigger: dy or dx?
BCS STEPINY ;If dy, we need to take big steps in y

STEPINX LDY Y1 ;X is already set to x1
LDA BITP,X ;Plot the first point
AND (BUFFER),Y
STA (BUFFER),Y
>>> CINIT,DX ;Initialize the counter
CPY Y2
BCS XDECY ;Do we step forwards or backwards in Y?

XINCY >>> XSTEP,INY
RTS

STEPINY LDY Y1 ;Well, a little repetition never hurt anyone
LDA BITP,X
AND (BUFFER),Y
STA (BUFFER),Y
>>> CINIT,DY

```

```

 CPY Y2
 BCS YDECY

YINCY >>> YSTEP,INY
 RTS

XDECY >>> XSTEP,DEY ;This is put here so that
 RTS ;Branches are legal

YDECY >>> YSTEP,DEY
 RTS

*-----
* Clean up

CLEANUP LDA VMCSB ;Switch char rom back in
 AND #%11110101 ;default
 STA VMCSB

 RTS ;bye!

 TXT 'Happy Holidays! '
 TXT 'slj 12/94'

*-----
* Set up bit table

 DS ^ ;Clear to end of page
 ;So that tables start on a page boundary
BITP LUP 16 ;128 Entries for X
 DFB %01111111
 DFB %10111111
 DFB %11011111
 DFB %11101111
 DFB %11110111
 DFB %11111011
 DFB %11111101
 DFB %11111110
 --^

SIN EQU SIN+128 ;Table of sines, 120 bytes
COS EQU SIN+128 ;Table of cosines
 ;Both of these trig tables are
 ;currently set up from BASIC

ZDIV EQU COS+128 ;Division table
TMATH EQU ZDIV+384 ;Math table of f(x)=x*x/256

```

And here are the native C64 files:









M4J!D<WDLH&1S>@UD<WH@/2`D-C, -<W@/@/2`D-C0@.W1(15-%H\$%21:!42\$6@M04-454%,H\$%.1TQ%4Z!)3J!8H%F@04Y\$H%H-<WD@/2`D-C4-<WH@/2`D-C8-M=#\$@/2`D-C@.W1(15-%H\$%21:!54T5\$H\$E.H%1(1:!23U1!5\$E/3@UT,B`]M("0V.`UT,R`]("0V.2`[<T5%H%1(1:!!4E1)0TQ%H\$9/4J!-3U)%H\$1%5\$%)M3%, -=#0@/2`D-F\$-=#4@/2`D-F(-=#8@/2`D-F, -=#<@/2`D-F0-=#@@/2`DM-F4-=#D@/2`D-F8-=#\$P(#T@)#<P#6\$Q,2`]("1A-2`[=\$A%4T6@05)%H%1(M1:!%3\$5-14Y44Z!/1J!42\$6@4D]4051)3TZ@34%44DE8#6(Q,B`]("1A-B`[M>'EZ#6,Q,R`]("1A-PUD,C\$@/2`D83@@.W1(1:!.54U"15\*@1\$5.3U1%4Z`HM4D]7+\$/3%5-3BD-93(R(#T@)&\$Y#68R,R`]("1A80UG,S\$@/2`D86(-: #,RM(#T@)&%C#6DS,R`]("1A9`T-#2HJ\*J!M04-23U,-#6U0=F4@;6%C#2!L9&\$@M73\$-( '-T82! =, @T@/#P\#0UG971K97D@;6%C("` [=T%)5\*!&3U\*@0: !+15E0M4D534PUW86ET(&IS<B!G971I;@T@8VUP(",P,`T@8F5Q('=A:70-(#P\`T-M9&5B=6<@;6%C("` [<%)3E2@0: !#2\$%204-415(-H"!D;Z`P("` [9\$.)U2@M05-314U"3\$4-#2!L9&\$@(UTQ#2`J<W(@8VAR;W5T#2`^/CX@9V5T:V5Y(#MAM3D2@5T%)5\*!43Z!#3TY424Y510T@8VUP(",G4R<@.VU9H%-0U)%0U2@4U=)M5\$-(H\$M%60T@8FYE(&PQ#2!J<W(@8VQE86YU<`T@:FUP(&10;F4-;#\$@8VUPM(",G6"<@.VU9H%-0U)%5\*!!0D]25\*!+15D-(&)N92!D;VYE#2!J;7`@8VQEM86YU<`T@9FEN#610;F4@/#P\#0UD96)U9V\$@;6%C#2!D;Z`P#2!L9&\$@73\$-M(' -T82`Q,#(T#2!F:6X-9&]N96\$@/#P\#0TJ+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+0T-(&QD82`C)#`P#2!S=&\$@8FMG;F0-( '-T82!B;W)DM97(-(&QD82!V;6-S8@T@86YD(",E,#`P,#\$Q,3\$@.W-#4D5%3J!-14U/4EF@M5\$^@,3`R-`T@;W)A(",E,#`P,3`P,#`-( '-T82!V;6-S8@T-(&QD>2`C,#`-M(&QD82`C/'1T97AT#2!S=&\$@=&5M<#\$( -(&QD82`C/G1T97AT#2!S=&\$@=&5MM<#(-(&IM<"!T:71L90UT=&5X="!H97@@.3,P-3\$Q,3\$Q,2`[0TQ%05\*@4T-2M145.+\*!72\$E412R@0U)34J!\$3@T@='AT(">@H\*"@H\*"@H\*"@H\*"@0U5"13-\$MH%8R+C`G+#!\$+#!\$#2!T>'0@)Z"@H\*"@H\*"@H\*"@H\*"@H\*"@H\$)9)RPP1`T@M:&5X(#EF(#M#64%.#2!T>'0@)Z"@H\*!35\$502\$5.H\$I51\$0G#2!H97@@.3D-M('1X="`GH\*"@H\$=%3U)'1:!405E,3U(G+#!\$+#!\$#2!H97@@.6(-('1X="`GMH\*!#2\$5#2Z!/552@5\$A%H\$I!3BZ@.36@25-3546@3T8G+#!\$#2!H97@@.38-M('1X="`GH\*!#/4A!0TM)3D<G#2!H97@@.6(-('1X="`GH\$9/4J!-3U)%H\$1%M5\$%)3%,A)RPP1`T@:&5X(#!D,60Q9#EE,3(-('1X="`G1C\$01C(G+#DR#2!TM>'0@)Z`MH\$E.0R]\$14.@6"U23U1!5\$E/3B<L,\$0-(&AE>"`Q9#%D,3(-('1XM="`G1C,01C0G+#DR#2!T>'0@)Z`MH\$E.0R]\$14.@62U23U1!5\$E/3B<L,\$0-M(&AE>"`Q9#%D,3(-('1X="`G1C401C8G+#DR#2!T>'0@)Z`MH\$E.0R]\$14.@M6BU23U1!5\$E/3B<L,\$0-(&AE>"`Q9#%D,3(-('1X="`G1C<G+#DR#2!T>'0@M)Z!215-%5%,G+#!\$#2!T>'0@)Z"@4%)%4U.@4: !43Z!154E4)RPP1`T@:&5XM(#!D,#4-('1X="`GH\*"@H\*"@4%)%4U.@04Y9H\$M%6: !43Z!"14=)3B<L,\$0-M(&AE>"`P,`UT:71L92!L9&\$@\*`1E;7`Q\*2QY#2!B97\$@.F-0;G0-(&IS<B!CM: ')0=70-(&EN>0T@8FYE('1I=&QE#2!I;F,@=&5M<#(-(&IM<"!T:71L90T@M='AT("=T2\$E3H\$E3H\$&@4T5#4D54H%1%6%2@34534T%'12\$G#3IC;VYT(#X^M/B!G971K97D-#2HJ\*BJ@<T54H%50H%1!0DQ%4R@\_\*0T-\*J!T04),15.@05)%MH\$-54E)%3E1,6: !3152@55"@24Z@8F%S:6,-\*J!!3D2@0EF@5\$A%H\$%34T5-M0DQ%4BX-#71A8FQE<R!L9&\$@(SYT;6%T:`T@<W1A('HQ\*\$S-(' -T82!Z,BLQM#0TJ\*BHJH&- ,14%2H%-#4D5%3J!!3D2@4T54H%50H")"251-05`B#7-E='5PM(&QD82`C)#`Q(#MW2\$E410T@<W1A("1D,#(Q(#MT2\$E3H\$E3H\$1/3D6@4T^@M5\$A!5\*!/3\$1%4@T@;&1A(",Q-#<@.TU!0TA)3D53H%=)3\$R@4T54H%50#2!JM<W(@8VAR;W5T#2!L9&\$@(R0P,"`[0T]24D5#5\$Q9#2!S=&\$@)&0P,C\$-(&QDM82`C/'-S=&%R=`T@861C(",Q,B`[=\$A%H\$=/04R@25.@5\$^@0T5.5\$52H%1(M1:!'4D%02\$E#4PT@<W1A('1E;7`Q(#MC3TQ534Z@,3(-(&QD82`C/G-S=&%RM="` [<D]7H#D-( '-T82!T96UP,2LQ(#MS<W1A<G2@4\$)3E13H%1/H%)/5Z`YM#2!L9&\$@(S`P#2!L9`D@(S`P#2!L9`@@(S`P(#M8H%=)3\$R@0T]53E2@,3:@

M4D]74Z!&3U\*055, - (&-L8PT- . FQ0;W`@<W1A ("AT96UP, 2DL>0T@: 6YY#2!A
M9&, @(\$V#2!B8V, @. FQ0;W` - (&-L8PT@;&1A('1E;7`Q#2!A9&, @ (S0P (#MN
M145\$H%1/H\$%\$1\*`T, \*!43Z!42\$6@0D%31: !03TE.5\$52#2!S=&\$@=&5M<#\$\$@
M.W1/H\$I535"@5\$^@5\$A%H\$Y%6%2@4D]7#2!L9&\$@=&5M<#\$\$K, 0T@861C(" , P
M, "` [= \$%+1: !#05)%H\$]&H\$- !4E) )15, - ('-T82!T96UP, 2LQ#2!L9`D@ (S`P
M#2!I;G@- ('1X82`@.WB@25.@04Q33Z!!3J!)3D1%6\*!)3E1/H%1(1: !#2\$%2
M04-415\*@3E5-0D52#2!C<'@@(\$V#2!B;F4@.FQ0;W`@.VY%142@5\$^@1\$^@
M252@, 3:@5\$E-15, -#2`^/CX@9&5B=6<L)S(G#2HJ\*BJ@<T54H%50H\$)51D9%
M4E, -#2!L9&\$@ (SQB=69F, 0T@<W1A (&)U9F9E<@T@;&1A(" , ^8G5F9C\$- ('-T
M82!B=69F97(K, 0T@<W1A('IT96UP(#M:5\$5-4\*!724Q, H\$U!2T6@3\$E&1: !3
M24U03\$6@1D]2H%53#2!L9&\$@=FUC<V(- (&%N9"`C)3\$Q, 3\$P, #`Q(#MS5\$%2
M5\*!(15)%H%-/H%1(052@4U=!4\*!"549&15)3H%=)3\$R@5T]22Z!224=(5`T@
M;W)A(" , E, #`P, #\$\$Q, 3`- ('-T82!V;6-S8@T-\*BHJ\*J!S152@55"@24Y)5\$E!
M3\*!604Q515, -#6EN:70@;&1A(" , P, `T@<W1A(&1S>`T@<W1A(&1S>0T@<W1A
M(&1S>@T@<W1A('X#2!S=&\$@<WD- ('-T82!S>@T- (#X^/B!D96)U9RPG- "<-
M#2HM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM#2J@;4%)3J!, 3T]0
M#0TJ\*BHJH&=%5\*!+15E04D534PT- ;6%I;@T@8VQI#6MP<F5S<R!J<W(@9V5T
M:6X- (&-M<"`C, 3, S(#MF, 3\ - (&)N92`Z9C(- (&QD82!D<W@- (&-M<"`C86YG
M;6%X+S(@.VY/H\$U/4D6@5\$A!3J!020T@8F5Q(#IC;VYT#2!I;F,@9'-X(#M/
M5\$A%4E=)4T6@24Y#4D5!4T6@6"U23U1!5\$E/3@T@:FUP(#IC;VYT#3IF,B!C
M;7`@(S\$S-R`[9C(\_#2!B;F4@.F8S#2!L9&\$@9'-X#2!B97\$@.F-0;G0- (&1E
M8R!D<W@- (&IM<"`Z8V]N=`TZ9C,@8VUP(" , Q, S0- (&)N92`Z9C0- (&QD82!D
M<WD- (&-M<"`C86YG;6%X+S(- (&)E<2`Z8V]N=`T@:6YC(&1S>2`[:4Y#4D5!
M4T6@62U23U1!5\$E/3@T@:FUP(#IC;VYT#3IF-"!C;7`@(S\$S.`T@8FYE(#IF
M-0T@;&1A(&1S>0T@8F5Q(#IC;VYT#2!D96,@9'-Y#2!J;7`@.F-0;G0-.F8U
M(&-M<"`C, 3, U#2!B;F4@.F8V#2!L9&\$@9'-Z#2!C;7`@(V%N9VUA>`\R#2!B
M97\$@.F-0;G0- (&EN8R!D<WH@.UHM4D]4051)3TX- (&IM<"`Z8V]N=`TZ9C8@
M8VUP(" , Q, SD- (&)N92`Z9C<- (&QD82!D<WH- (&)E<2`Z8V]N=`T@9&5C(&1S
M>@T@:FUP(#IC;VYT#3IF-R!C;7`@(S\$S-@T@8FYE(#IQ#2!J;7`@:6YI=`TZ
M<2!C;7`@(R=1)R`[4: !154E44PT@8FYE(#IC;VYT#2!J;7`@8VQE86YU<`T-
M.F-0;G0@<V5I("`[<U!%142@5\$A)3D=3H%50H\$&@0DE4#2J@/CX^H&1E8G5G
M+ "<U)PT-\*BHJ\*J!U4\$1!5\$6@04Y'3\$53#0UU<&1A=&4@8VQC#2!L9&\$@<W@-
M(&%D8R!D<W@- (&-M<"`C86YG;6%X(#MA4D6@5T6@/CV@34%824U53: !!3D=,
M13\ - (&)C8R`Z8V]N=#\$- ('-B8R`C86YG;6%X(#II1B!33RP@4D53150-.F-0
M;G0Q(' -T82!S>`T@8VQC#2!L9&\$@<WD- (&%D8R!D<WD- (&-M<"`C86YG;6%X
M#2!B8V,@.F-0;G0R#2!S8F,@(V%N9VUA> "`[<T%-1: ! \$14%, #3IC;VYT, B!S
M=&\$@<WD- (&-L8PT@;&1A(' -Z#2!A9&, @9' -Z#2!C;7`@(V%N9VUA>`T@8F-C
M(#IC;VYT, PT@<V)C(" -A;F=M87@-.F-0;G0S(' -T82!S>@T-\*BHJ\*J!R3U1!
M5\$6@0T]/4D1)3D%415, -#7)0=&%T90T-\*BHJH&9)4E-4+\*!#04Q#54Q!5\$6@
M5#\$L5#(L+BXN+%0Q, `T-\*BJ@= %= /H\$U!0U)/4Z!43Z!324U03\$E&6: !/55\*@
M3\$E&10UA9&1A(&UA8R`@.V%\$1\*!45T^@04Y'3\$53H%1/1T542\$52#2!C;&, -
M(&QD82! =, 0T@861C(%TR#2!C;7`@(V%N9VUA> "`[:5.@5\$A%H%-53: `^H#(J
M4\$D\_#2!B8V,@9&]N90T@<V)C(" -A;F=M87@@.VE&H%- /+\*!354)44D%#5\*`R
M\*E!)#610;F4@/#P\#0US=6)A(&UA8R`@.W-50E1204-4H%173Z!!3D=, 15, -
M(' -E8PT@;&1A(%TQ#2!S8F,@73(- (&)C<R!D;VYE#2!A9&, @(V%N9VUA> "`[
M;T]04RR@5T6@3D5%1\*!43Z!!1\$2@, BI020UD;VYE(#P\ /`T-\*BJ@;D]7H\$- !
M3\$-53\$%41: !4, 2Q4, BQ%5\$, N#0T@/CX^(' -U8F\$L<WD[<WH- (' -T82!T, 2`[
M5#\$]4UDM4UH- (#X^/B!A9&1A+' -Y.W-Z#2!S=&\$@=#(@.U0R/5-9\*U- :#2`^
M/CX@861D82QS>#MS>@T@<W1A('0S(#M4, SU36"M36@T@/CX^(' -U8F\$L<W@[
M<WH- (' -T82!T- "`[5#0]4U@M4UH- (#X^/B!A9&1A+' -X.W0R#2!S=&\$@=#4@
M.U0U/5-8\*U0R#2`^/CX@<W5B82QS>#MT, 0T@<W1A('0V(#M4-CU36"U4, 0T@

M/CX^(&%D9&\$L<W@[=#\$-(' -T82!T-R`[5#<]4U@K5#\$-(#X^/B!S=6)A+'0R  
M.W-X#2!S=&\$@=#@@.U0X/50R+5-8#2`^/CX@<W5B82QS>3MS>`T@<W1A('0Y  
M(#M4.3U362U36`T@/CX^(&%D9&\$L<W@[<WD-(' -T82!T,3`@.U0Q,#U36"M3  
M60T-\*J!E5\*!63TE,02\$-#2HJ\*J!N15A4+\*!#04Q#54Q!5\$6@82QB+&,L+BXN  
M+&D-#2HJH&% .3U1(15\*@55-%1E5,H\$Q)5%1,1:!!-04-23PUD:78R(&UA8R`@  
M.V1)5DE\$1:!!H%-)1TY%1\*!.54U"15\*@0EF@,@T[:52@25.@05-354U%1\*!4  
M2\$%4H%1(1:!.54U"15(-(&)P;"!P;W,@.TE3H\$E.H%1(1:!!0T-5355,051/  
M4@T@8VQC#2!E;W(@R1F9B`[=T6@3D5%1\*!43Z!53BU.14=!5\$E61:!!42\$6@  
M3E5-0D52#2!A9&,@(S`Q(#M"6:!!404M)3D>@250G4Z!#3TU03\$5-14Y4#2!L  
M<W(@(#M\$259)1\$6@0EF@5%=#/2!C;&,-(&50<B`C)&9F#2!A9&,@(S`Q(#MM  
M04M%H\$E4H\$Y%1T%4259%H\$%'04E.#2!J;7`@9&]N961I=@UP;W,@;' -R("`[  
M;E5-0D52H\$E3H%!/4TE4259%#610;F5D:78@/#P\#0UM=6PR(&UA8R`@.VU5  
M3%1)4\$Q9H\$&@4TE'3D5\$H\$Y534)%4J!"6:`R#2!B<&P@<&]S;0T@8VQC#2!E  
M;W(@R1F9@T@861C(",D,#\$-(&S;`T@8VQC#2!E;W(@R1F9@T@861C(",D  
M,#\$-(&IM<"!D;VYE;75L#7!0<VT@87-L#610;F5M=6P@/#P\#0TJ\*J!N3U1%  
MH%1(052@5T6@05)%H\$-54E)%3E1,6:!!-04M)3D>@0:!!-24Y/4J!,14%0#2HJ  
MH\$]&H\$9!251(H%1(052@3D^@3U9%4D9,3U=3H%=)3\$R@3T-#55(N#0TZ8V%L  
M8V\$@8VQC#2!L9'@@=#\$-(&QD82!C;W,L>`T@;&1X('0R#2!A9&,@8V]S+'@-  
M(' -T82!A,3\$@.V\$]\*\$-/4RA4,2DK0T]3\*%0R\*2D0,@TZ8V%L8V(@;&1X('0Q  
M#2!L9&\$@<VEN+'@-(' -E8PT@;&1X('0R#2!S8F,@<VEN+'@-(' -T82!B,3(@  
M.V(]\*%-)3BA4,2DM4TE.\*%0R\*2D0,@TZ8V%L8V,@;&1X(' -Y#2!L9&\$@<VEN  
M+'@- (#X^/B!M=6PR#2!S=&\$@8S\$S(#MC/5-)3BA362D-.F-A;&-D(' -E8PT@  
M;&1X('0X#2!L9&\$@8V]S+'@-(&QD>"!T-PT@<V)C(&-0<RQX#2!S96,-(&QD  
M>"!T-0T@<V)C(&-0<RQX#2!C;&,-(&QD>"!T-@T@861C(&-0<RQX(#MD23TH  
M0T]3\*%0X\*2U#3U,H5#<I\*T-/4RA4-BDM0T]3\*%0U\*2D0,@T@/CX^(&1I=C(-  
M(&-L8PT@;&1X('0S#2!A9&,@<VEN+'@-(' -E8PT@;&1X('0T#2!S8F,@<VEN  
M+'@-(' -T82!D,C\$@.V0]\*%-)3BA4,RDM4TE.\*%0T\*2MD22D0,@TZ8V%L8V4@  
M<V5C#2!L9'@@=#4-(&QD82!S:6XL>`T@;&1X('0V#2!S8F,@<VEN+'@-(' -E  
M8PT@;&1X('0W#2!S8F,@<VEN+'@-(' -E8PT@;&1X('0X#2!S8F,@<VEN+'@@  
M.V5)/2A324XH5#4I+5-)3BA4-BDM4TE.\*%0W\*2U324XH5#@I\*2\R#2`^/CX@  
M9&EV,@T@8VQC#2!L9'@@=#,-(&%D8R!C;W,L>`T@8VQC#2!L9'@@=#0-(&%D  
M8R!C;W,L>`T@<W1A(&4R,B`[93TH0T]3\*%0S\*2M#3U,H5#0I\*V5)\*2\R#3IC  
M86QC9B!L9'@@=#D-(&QD82!S:6XL>`T@<V5C#2!L9'@@=#\$P#2!S8F,@<VEN  
M+'@-(' -T82!F,C,@.V8]\*%-)3BA4.2DM4TE.\*%0Q,"DI+S(-.F-A;&-G(&QD  
M>"!T-@T@;&1A(' -I;BQX#2!S96,-(&QD>"!T.`T@<V)C(' -I;BQX#2!S96,-  
M(&QD>"!T-PT@<V)C(' -I;BQX#2!S96,-(&QD>"!T-0T@<V)C(' -I;BQX(#MG  
M23TH4TE.\*%0V\*2U324XH5#@I+5-)3BA4-RDM4TE.\*%0U\*2D0,@T@/CX^(&1I  
M=C(-(&-L8PT@;&1X('0T#2!A9&,@8V]S+'@-(' -E8PT@;&1X('0S#2!S8F,@  
M8V]S+'@-(' -T82!G,S\$@.V<]\*\$-/4RA4-"DM0T]3\*%0S\*2MG22D0,@T@/CX^  
M(&1E8G5G82QG,S\$-(#X^/B!D96)U9RPG1R<-.F-A;&-H(&-L8PT@;&1X('0V  
M#2!L9&\$@8V]S+'@-(&QD>"!T-PT@861C(&-0<RQX#2!S96,-(&QD>"!T-0T@  
M<V)C(&-0<RQX#2!S96,-(&QD>"!T.`T@<V)C(&-0<RQX(#MH23TH0T]3\*%0V  
M\*2M#3U,H5#<I+4-/4RA4-2DM0T]3\*%0X\*2D0,@T@/CX^(&1I=C(-(&-L8PT@  
M;&1X('0S#2!A9&,@<VEN+'@-(&-L8PT@;&1X('0T#2!A9&,@<VEN+'@-(' -T  
M82!H,S(@.V@]\*%-)3BA4,RDK4TE.\*%0T\*2MH22D0,@TZ=VAE=R!C;&,-(&QD  
M>"!T.0T@;&1A(&-0<RQX#2!L9'@@=#\$P#2!A9&,@8V]S+'@-(' -T82!I,S,@  
M.VD]\*\$-/4RA4.2DK0T]3\*%0Q,"DI+S(-#2HJH&E4)U.@04Q,H\$1/5TY(24Q,  
MH\$923TV@2\$5212X-(&IM<"!D;W=N:&EL;`T@='AT("=G146@8E)!24XLH%=(  
M052@1\$^@64]5H%=!3E2@5\$^@1\$^@)PT@='AT("=43TY)1TA4/R<-#2HJH')/  
M5\$%412R@4%)/2D5#5"R@04Y\$H%-43U)%H%1(1:!!03TE.5%,-9&]W;FAI;&P-  
M#2J@8:!.14%4H\$U!0U)/#6YE9R!M86,@(#MC2\$.1T6@5\$A%H%-)1TZ@3T:@

M0: !45T\G4Z!#3TU03\$5-14Y4#2!C;&, - (&QD82!=, 2` [3E5-0D52+@T@96]R  
M(", D9F8- (&%D8R`C)#`Q#2`\/#P-#2HM+2TM+2TM+2TM+2TM+2TM+2TM+2TM  
M+2TM+2TM+2TM#2J@=#A%4T6@34%#4D]3H%)%4\$Q!0T6@5\$A%H%!2159)3U53  
MH%!23TI%0U1)3TX-\*J!354)23U5424Y%+@T-<VUU; '0@; 6%C (#MM54Q425! ,  
M6: !45T^@4TE'3D5\$H#@M0DE4#2`@(#M.54U"15)3.J!A\*GD0-C2@+3Z@80T@  
M<W1A('HQ#2!C;&, - (&50<B`C)&9F#2!A9&, @ (R0P, 0T@<W1A('HR#2!L9&\$@  
M\*'HQ\*2QY#2!S96, - ('-B8R`H>C(I+'D-(#P\/'`@.V%, 3\*!\$3TY%H#HI#0T-  
M861D<W5B(&UA8R`@.V%\$1\*!/4J!354)44D%#5\*!45T^@3E5-0D524PT@('` [   
M1\$5014Y\$24Y'H\$].H\$9)4E-4H\$E.4%54#2!I9B`M/5TQ(#MI1J!354)44D%#  
M5`T@<V5C('` [5\$A%3J!54T6@5\$A)4Z!#3T1%#2!S8F, @73(- (&5L<V4@(#M/  
M5\$A%4E=)4T6@55-%H%1(25.@0T)\$10T@8VQC#2!A9&, @73(- (&9I;@T@/#P\  
M#0T-<' )0:F5C="!M86, @(#MT2\$6@04-454%, H%!23TI%0U1)3TZ@4D]55\$E.  
M10T@('` [5%=/H\$E.4%544Z!!4D6@55-%1\*`H6"Q9\*0T@('` [0T]24D534\$).  
M1\$E.1Z!43Z`H\*R\M, 2PK+RTQ\*0T@('` [= \$A%H%1(25)\$H\$E.4%54H\$E3H%53  
M142@5\$ \- ('`@.T1%5\$234E.1:!)1J!42\$6@4D]4051%1`T@('` [6BU#3T]2  
M1\$E.051%H%- (3U5, 1\*!"10T@('` [4U1/4D5\$+\*!!3D2@24:@4T^@5TA%4D4N  
M#2`@(#MT2\$6@0T%, 3\$E.1Z!23U5424Y%H\$A!3D1, 15, - ('`@.T- (04Y'24Y'  
MH%1(1: !324=.H\$]&H%HN#0T@;&1A(&DS, R` [8T%, 0U5, 051%H%)/5\$%4142@  
M6CH- (#X^/B!A9&1S=6(L73\$ [9S, Q(#MA1\$2@3U\* @4U5"5%)!0U2@6`T@/CX^  
M(&%D9'-U8BQ=, CMH, S(@.V%\$1\*!/4J!354)44D%#5\*!9#2!I9B!P+%TS(#MD  
M3Z!71:!.145\$H%1/H%-43U)%H%1(1: !03TE.5#\ - ('-T82! =, R` [= \$A%3J!\$  
M3Z!33R\$- (&9I;@TJH&50<J`C, 3(XH#MW1:!!4D6@1T])3D>@5\$^@5\$%+1: `Q  
M, C@K6@T@=&%X('` [ ;D]7H\$E4H\$E3H%)%0419H\$9/4J!)3D1%6\$E.1PT@;&1A  
M('ID:78L>"` [= \$%"3\$6@3T:@1`\H6BM: , "D- ('1A>2`@.WF@3D]7H\$-/3E1!  
M24Y3H%!23TI%0U1)3TX-#2!L9&\$@8S\$S(#MN3U>@0T%, 0U5, 051%H%)/5\$%4  
M142@6`T@/CX^(&%D9'-U8BQ=, 3MA, 3\$- (#X^/B!A9&1S=6(L73([8C\$R#2`^  
M/CX@<VUU; '0@.W-)1TY%1\*!-54Q425!, 6: !A\*GD0-C0M/F\$- (&-L8PT@861C  
M(", V-" [ ;T9&4T54H%1(1: !#3T]21\$E.051%#2!T87@@(#MN3U>@>\*!)4Z!2  
M3U1!5\$5\$H%#@A#0T@;&1A(&8R, R` [ ;D]7H\$E4)U.@62=3H%154DX- (#X^/B!A  
M9&1S=6(L73\$ [9#(Q#2`^/CX@861D<W5B+%TR.V4R, @T@/CX^(' -M=6QT#2!C  
M; &, - (&%D8R`C-C0@.V]&1E-%5`T@8VUP('EM:6X@.V9)1U521:!/552@24:@  
M252@25.@00T@8F-S(&Y0=&UI; B` [34E.H\$]2H\$U!6\*!604Q51: !&3U\* @60T@  
M<W1A('EM:6X- (&)C8R!N; W1M87@@.W1(25.@25.@55-%1\*!)3J!#04Q#54Q!  
M5\$E.1PUN; W1M:6X@8VUP('EM87@@.U1(1: !&24Q, 142@1D%#15, - (&)C8R!N  
M; W1M87@- ('-T82!Y; 6%X#6Y0=&UA>"!T87D@(#MN3U2@4D5!3\$Q9H\$Y%0T53  
M4T%260T- (#P\/'`@.V%, 3\*!\$3TY%#0T- (&QD82`C-C0@.W)%4T54H'E-24Z@  
M04Y\$H'E-05@- ('-T82!Y; 6EN#2!S=&\$@>6UA>`T-\*J!P, 3U; , : `QH#%=#2`^  
M/CX@<' )0:F5C="PQ.S\$ [<#%Z(#MR3U1!5\$5\$H%J@4U1/4D5\$H\$E.H`Q>@T@  
M<W1X('`Q>`T@<W1Y('`Q>0TJH`R/5LQH" TQH#%=#2`^/CX@<' )0:F5C="PQ  
M.RTQ.W`R>@T@<W1X('`R>`T@<W1Y('`R>0TJH`S/5LM, : `M, : `Q70T@/CX^  
M('!R; VIE8W0L+3\$ [+3\$ [ ;F]P92` [9\$].)U2@4U1/4D6@6BU604Q510T@<W1X  
M('`S>`T@<W1Y('`S>0TJH`T/5LM, : `QH#%=#2`^/CX@<' )0:F5C="PM, 3LQ  
M.W`T>@T@<W1X('`T>`T@<W1Y('`T>0TJH`X/5LM, : `QH" TQ70T@/CX^ (&YE  
M9RQC, 3, - ('-T82!C, 3, - (#X^/B!N96<L9C(S#2!S=&\$@9C(S#2`^/CX@; F5G  
M+&DS, PT@<W1A(&DS, PT@/CX^('!R; VIE8W0L+3\$ [ , 3MN; W!E#2!S='@@<#AX  
M#2!S='D@<#AY#2J@<#<]6RTQH" TQH" TQ70T@/CX^('!R; VIE8W0L+3\$ [+3\$ [   
M; F]P90T@<W1X('`W>`T@<W1Y('`W>0TJH`V/5LQH" TQH" TQ70T@/CX^('!R  
M; VIE8W0L, 3LM, 3MN; W!E#2!S='@@<#9X#2!S='D@<#9Y#2J@<#4]6S&@, : `M  
M, 5T- (#X^/B!P<F]J96-T+#\$ [ , 3MP-7H- ('-T>"!P-7@- ('-T>2!P-7D-#2J@  
M8: !, 25143\$6@34%#4D\ -#7-E=&)U9B!M86, @(#MP552@0E5&1D524Z!72\$52  
M1: !42\$59H\$- !3J!"1: ! (55)4#2!L9&\$@ (S`P#2!S=&\$@8G5F9F5R#2!L9&\$@

M>G1E;7`@.UI414U0H\$-/3E1!24Y3H%1(1:!(24=(H\$)95\$6@2\$5210T@<W1A  
M(&)U9F9E<BLQ#2`\/#P-#2HJ\*BJ@8TQ%05\*@0E5&1D52#0TJH#X^/J!S971B  
M=68-\*F-L<F)U9J!L9&&@(R0P,\*`[<%)%5%19H%-44D%)1TA41D]25T%21"P-  
M\*J!L9'B@(R0P.\*`[::42\$E.2PTJH&QD>: `C)#`P#2HZ;&]0<\*!S=&&@\*&)U  
M9F9E<BDL>0TJH&EN>0TJH&)N9: `Z;&]0<`TJH&EN8Z!B=69F97(K,0TJH&1E  
M>`TJH&)N9: `Z;&]0<`T-\*J!T2\$E3H\$E3H%1(1:!.15>@04Y\$H\$E-4%)/5D5\$  
MH\$)51D9%4J!#3\$5!4@TJH%)/551)3D6@1D]2H\$9)3\$Q%1\*!&04-%4PT-(#X^  
M/B!S971B=68-( '-T82!T96UP,2LQ(#M"549&15(RH%=)3\$R@4\$)3E2@5\$\-  
M(&QD82`C)#@P(#M"549&15(K,3(X#2!S=&\$@=&5M<#\$@.VU!2T53H\$Q)1D6@  
M1D%35\$52H\$9/4J!54PUF:6QC;'(@;&1A(",P,`T@;&1X(",D,#@@.W=%)TQ,  
MH\$1/H\$E4H%173Z!!5\*!!H%1)344-(&QD>2`C)#`P#3IL;V]P,2!S=&\$@\*&)U  
M9F9E<BDL>0T@<W1A("AT96UP,2DL>0T@:6YY#2!C<'D@>6UI;@T@8FYE(#IL  
M;V]P,0T@;&1A(",D9F8@.VY/5Z!,3T%\$H%=)5\$B@1DE,3%,-.FQ0;W`R(' -T  
M82`H8G5F9F5R\*2QY#2!S=&\$@\*'1E;7`Q\*2QY#2!I;GD-(&-P>2!Y;6%X#2!B  
M8V,@.FQ0;W`R#2!L9&\$@(R0P,"`[8DQ!0TN@3U54H%1(1:1215-4#3IL;V]P  
M,R!S=&\$@\*&)U9F9E<BDL>0T@<W1A("AT96UP,2DL>0T@:6YY#2!B<&P@.FQ0  
M;W`S(#MU3E1)3\*!Y/3\$R.`T@;&1Y(",P,`T@:6YC(&)U9F9E<BLQ#2!I;F,@  
M=&5M<#\$K,0T@9&5X#2!B;F4@.FQ0;W`Q(#MG3Z!!3\$R@5\$A%H%=!6:!!4D]5  
M3D0-#2HJ\*BJ@;D]7H\$1205>@5\$A%H\$Q)3D53+@TJ\*BHJH&)55\*!&25)35\*!#  
M2\$5#2Z!&3U\*2\$E\$1\$5.H\$9!0T53(0TJ\*BHJH')%345-0D52.J!P,3U;;:`Q  
MH#%=H`R/5LQH" TQH#%=H`S/5LM,:`M,:`Q70TJ\*BHJH`T/5LM,:`QH#%=  
MH`U/5LQH#&@+3%=H`V/5LQH" TQH" TQ7: !P-SU;+3&@+3&@+3%=#2HJ\*BJ@  
M<#@]6RTQH#&@+3%=#0UL:6YE<R!L9&\$@(S`P#2!S=&\$@9F%C97,@.VA)1\$1%  
M3J!&04-%H\$-/54Y415(-.F9A8V4Q(&QD82!K#2!S96,-(' -B8R!P,7H-(&)V  
M<R`Z9F%C938@.V]615)&3\$]7H%,4D5!1%D\_#2!C;&,-(&%D8R!P-7H@.VE3  
MH\$LM5C%:H#R@,#\-(#MI1J!.3U0LH\$9!0T6@25.@24Y625-)0DQ%#2!B=F,@  
M.F1R87<Q(#MB552@5T6@34E'2%2@2\$%61:!/5D521DQ/5PT@;&1A('`U>B`[  
M=T%3H\$]615)&3\$]7H%!/4Z!/4J!.14<\_#3ID<F%W,2!B<&P@.F9A8V4V(#MI  
M1J!03U.@5\$A%3J!++58Q6J`^H#`-#2!L9&\$@(R0P,2`[;U1(15)725-%+\*!\$  
M4D%7H%1(10T@<W1A(&9A8V5S(#M&04-%(0T-(&QD82!P,7@-(' -T82!T>#-\$-  
M(&QD82!P,7D-(' -T82!T>3\$-(&QD82!P,G@-(' -T82!T>#(-(&QD82!P,GD-  
M(' -T82!T>3(-(&IS<B!D<F%W(#MP,2UP,@T-(&QD82!P,W@-(' -T82!T>#-\$-  
M(&QD82!P,WD-(' -T82!T>3\$-(&IS<B!D<F%W(#MP,BUP,PT-(&QD82!P-'@-  
M(' -T82!T>#(-(&QD82!P-'D-(' -T82!T>3(-(&IS<B!D<F%W(#MP,RUP-`T-  
M(&QD82!P,7@-(' -T82!T>#-\$-(&QD82!P,7D-(' -T82!T>3\$-(&IS<B!D<F%W  
M(#MP-"UP,: "@9D%#1: `QH\$1/3D4N#2!J;7`@.F9A8V4R(#MI1J!/3D6@25.@  
M5DE324),12R@5\$A%H\$]42\$52#2`@(#M)4TXG5"X-.F9A8V4V(&QD82!K#2!S  
M96,-(' -B8R!P-7H-(&)V<R`Z9F%C93(-(&-L8PT@861C('`Q>B`[;D]7H\$-(  
M14-+H\$E&H&LM5C9:H#R@,`T@8G9C(#ID<F%W-B`[;\$]61:42\$%4H\$]615)&  
M3\$]7#2!L9&\$@<#%Z#3ID<F%W-B!B<&P@.F9A8V4R(#MI1J!.3U0LH\$=/H\$].  
M#0T@;&1A(",D,C`-(' -T82!F86-E<R`[;U1(15)725-%+\*!\$4D%7H\$E4#0T@  
M;&1A('`U>`T@<W1A('1X,@T@;&1A('`U>0T@<W1A('1Y,@T@;&1A('`V>`T@  
M<W1A('1X,0T@;&1A('`V>0T@<W1A('1Y,0T@:G-R(&1R87<@.W`U+7`V#0T@  
M;&1A('`W>`T@<W1A('1X,@T@;&1A('`W>0T@<W1A('1Y,@T@:G-R(&1R87<@  
M.W`V+7`W#0T@;&1A('`X>`T@<W1A('1X,0T@;&1A('`X>0T@<W1A('1Y,0T@  
M:G-R(&1R87<@.W`W+7`X#0T@;&1A('`U>`T@<W1A('1X,@T@;&1A('`U>0T@  
M<W1A('1Y,@T@:G-R(&1R87<@.W`X+7`U#0TZ9F%C93(@;&1A(&L-(' -E8PT@  
M<V)C('`Q>@T@8G9S(#IF86-E-0T@8VQC#2!A9&,@<#1Z(#MK+58R6J` \H#`\_  
M#2!B=F,@.F1R87<R#2!L9&\$@<#1Z#3ID<F%W,B!B<&P@.F9A8V4U#2!L9&\$@  
M(R0P,B`[:4:@4T\LH\$1205>@250A#2!0<F\$@9F%C97,-(' -T82!F86-E<PT-  
M(&QD>"!P,7@@.W=%)U)%H\$1/24Y'H%1(25.@5\$A)4Z!705D-(' -T>"!T>#@\$

M.U1/H%- !5D6@0: !&15>@0UE#3\$53#2!L9 '@@<#%Y#2!S='@@='DQ#0T@86YD  
M(",D,#\$@.W-(05)%4Z!!3J!%1\$=%H%=)5\$B@1D%#1: `Q#2!B;F4@.F8R<S(@  
M.W-+25"@5\$^@3D585\*!%1\$=%H%E&H%!215-%3E0-#2!L9&\$@<#)X#2!S=&\$@  
M='@R#2!L9&\$@<#)Y#2!S=&\$@='DR#2!J<W(@9')A=R`[<#M<#(-#3IF,G,R  
M(&QD>"!P-7@-(' -T>"!T>#(-(&QD>"!P-7D-(' -T>"!T>3(-(&IS<B!D<F%W  
M(#MP,2UP-0T-(&QD>"!P-G@-(' -T>"!T>#(-(&QD>"!P-GD-(' -T>"!T>3\$-  
M#2!L9&\$@9F%C97,-(&%N9" `C)#(P(#MA3%-/H%-(05)%4Z!!3J!%1\$=%H%=)  
M5\$B@-@T@8FYE(#IF,G,T#0T@:G-R(&1R87<@.W`U+7`V#0TZ9C)S-"!L9&\$@  
M<#)X#2!S=&\$@='@R#2!L9&\$@<#)Y#2!S=&\$@='DR(#MS54-(H\$E3H\$9!0T6@  
M,@T@:G-R(&1R87<@.W`V+7`R#2!J;7`@.F9A8V4S(#MS2TE0H#4-#3IF86-E  
M-2!L9&\$@:PT@<V5C#2!S8F,@<#1Z#2!B=G,@.F9A8V4S#2!C;&,-(&%D8R!P  
M,7H@.W-!346@5\$A)3D>@04=!24XN+BX-(&)V8R`Z9')A=S4-(&QD82!P,7H-  
M.F1R87<U(&)P;"`Z9F%C93,-(&QD82`C)#\$P#2!0<F\$@9F%C97,-(' -T82!F  
M86-E<PT-(&QD>"!P,W@-(' -T>"!T>#(-(&QD>"!P,WD-(' -T>"!T>3\$-#2!A  
M;F0@(R0P,2`[<TA!4D53H%=)5\$B@,0T@8FYE(#IF-7,R#0T@;&1A('`T>`T@  
M<W1A('1X,@T@;&1A('`T>0T@<W1A('1Y,@T@:G-R(&1R87<@.W`S+7`T#0TZ  
M9C5S,B!L9&\$@<#)=X#2!S=&\$@='@R#2!L9&\$@<#)=Y#2!S=&\$@='DR#2!J<W(@  
M9')A=R`[<#M<#<-#2!L9&\$@<#AX#2!S=&\$@='@Q#2!L9&\$@<#AY#2!S=&\$@  
M='DQ#0T@;&1A(&9A8V5S#2!A;F0@(R0R,"`[<TA!4D53H%=)5\$B@-@T@8FYE  
M(#IF-7,T#0T@:G-R(&1R87<@.W`W+7`X#3IF-7,T(&QD82!P-'@-(' -T82!T  
M>#(-(&QD82!P-'D-(' -T82!T>3(@.W`X+7`T#2!J<W(@9')A=R`[=%/H\$U/  
M4D6@5\$^@1T\A#0TZ9F%C93,@;&1A(&L-(' -E8PT@<V)C('`Q>@T@8G9S(#IF  
M86-E-`T@8VQC#2!A9&,@<#)Z#2!B=F,@.F1R87<S#2!L9&\$@<#)Z#3ID<F%W  
M,R!B<&P@.F9A8V4T(#MA2\*!214-+3TZ@250G4Z!!)TA)1\$1%3BR@6550#2!L  
M9&\$@<#)A(@9A8V5S#2!S=&\$@9F%C97,-#2!L9 '@@<#%X#2!S='@@  
M='@Q#2!L9 '@@<#%Y#2!S='@@='DQ#0T@86YD(",D,#\$@.W-(05)%4Z!7251(  
MH#\$-(&)N92`Z9C-S,@T-(&QD82!P-'@-(' -T82!T>#(-(&QD82!P-'D-(' -T  
M82!T>3(-(&IS<B!D<F%W(#MP,2UP-`T-.F8S<S(@;&1X('`U>`T@<W1X('1X  
M,@T@;&1X('`U>0T@<W1X('1Y,@T-(&QD82!F86-E<PT@86YD(",D,#(@.W-(  
M05)%4Z!7251(H#(-(&)N92`Z9C-S,PT-(&IS<B!D<F%W(#MP,2UP-0TZ9C-S  
M,R!L9 '@@<#AX#2!S='@@='@Q#2!L9 '@@<#AY#2!S='@@='DQ#0T@;&1A(&9A  
M8V5S#2!A;F0@(R0R,"`[<TA!4D53H%=)5\$B@-@T@8FYE(#IF,W,T#0T@:G-R  
M(&1R87<@.W`U+7`X#3IF,W,T(&QD>"!P-'@-(' -T>"!T>#(-(&QD>"!P-'D-  
M(' -T>"!T>3(-#2!L9&\$@9F%C97,-(&%N9" `C)#\$P(#MS2\$%215.@5TE42\*`U  
M#2!B;F4@9F%C9610;F4-#2!J<W(@9')A=R`[<#M<#0-(&IM<"!F86-E9&]N  
M90T-.F9A8V4T(&QD82!K#2!S96,-(' -B8R!P,GH-(&)V<R!F86-E9&]N90T@  
M8VQC#2!A9&,@<#%Z#2!B=F,@.F1R87<T#2!L9&\$@<#%Z#3ID<F%W-"!B<&P@  
M9F%C9610;F4-#2!L9&\$@<#)X#2!S=&\$@='@Q#2!L9&\$@<#)Y#2!S=&\$@='DQ  
M#0T@;&1A(&9A8V5S#2!A;F0@(R0P,2`[<TA!4D53H%=)5\$B@,0T@8FYE(#IF  
M-' ,R#0T@;&1A('`S>`T@<W1A('1X,@T@;&1A('`S>0T@<W1A('1Y,@T@:G-R  
M(&1R87<@.W`R+7`S#0TZ9C1S,B!L9&\$@<#9X#2!S=&\$@='@R#2!L9&\$@<#9Y  
M#2!S=&\$@='DR#0T@;&1A(&9A8V5S#2!A;F0@(R0P,B`[<TA!4D53H%=)5\$B@  
M,@T@8FYE(#IF-' ,S#0T@:G-R(&1R87<@.W`R+7`V#3IF-' ,S(&QD82!P-W@-  
M(' -T82!T>#(-(&QD82!P-WD-(' -T82!T>3\$-#2!L9&\$@9F%C97,-(&%N9" `C  
M)#(P(#MS2\$%215.@5TE42\*`V#2!B;F4@.F8T<S0-#2!J<W(@9')A=R`[<#8M  
M<#<- .F8T<S0@;&1A('`S>`T@<W1A('1X,@T@;&1A('`S>0T@<W1A('1Y,@T-  
M(&QD82!F86-E<PT@86YD(",D,3`@.W-(05)%4Z!7251(H#4-(&)N92!F86-E  
M9&]N90T-(&IS<B!D<F%W(#MP-RUP,PUF86-E9&]N92`@(#MW2\$57(:"@=\$E-  
M1:!&3U\*0@:!"1452+@T-\*BHJ\*J!N3U>@5T6@3D5%1\*!43Z!53D9)3\$R@5\$A%  
MH\$]55%-)1\$6@1E)/3:!42\$6@1D%#15,-=6YF:6QL(&QD>2!Y;6EN#3IL;V]P  
M(#X^/B!S971B=68-(&QD>"`C,#@-.FPQ(&QD82`H8G5F9F5R\*2QY#2!E;W(@

M(R1F9B` [9T^@5\$E, 3\*!71: !&24Y\$H\$&@4\$Q/5%1%1`T@8FYE (#IG;W1C:&\$@  
M.U!/24Y4H"A)+D4NH&&@/#Z@)&9F\*0TJH&QD8: `C,#"@.W5.1DE,3\$E.1Z!!  
M4Z!71:!'3RXN+@T@<W1A("AB=69F97(I+'D-(&QD82`C)#@P#2!S=&\$@8G5F  
M9F5R#2!L9&\$@\*&)U9F9E<BDL>0T@96]R(",D9F8-(&)N92`Z9V]T8VAA#2J@  
M;&1AH",P,`T@<W1A("AB=69F97(I+'D-( '-T82!B=69F97(-(&EN8R!B=69F  
M97(K,0T@9&5X("` [= \$A)4Z!)4Z!/55\*@4T%&1519H%9!3%9%#2!B;F4@.FPQ  
M(#MR14%,3%F@4TA/54Q\$3B=4H\$Y%142@250-(&IS<B!C:&]K90T@:FUP(' -W  
M87!B=68-#3IG;W1C:&\$@.V&@0T].5\$%)3E.@5\$A%H&50<J!03\$]4H%9!3%5%  
M#2!S=&\$@=&5M<#\$@.VY/5Z!&24Y\$H%1(1:!(24=(H\$))5`T@;&1A(",P,`TZ  
M;#(@<V5C#2!R;VP-(&QS<B!T96UP,2`[<TA/54Q\$H%)%04Q,6: !54T6@0: !4  
M04),10T@8FYE(#IL,B`[1D]2H%1(25,A#2!A;F0@\*&)U9F9E<BDL>0T@<W1A  
M("AB=69F97(I+'D-#2!L9&\$@>G1E;7`@.VY/5Z!'3Z!43Z!42\$6@14Y\$#2`@  
M(#MC05)26:!)4Z!#3\$5!4@T@("`[84-454%,3%F@5T6@041\$H#<-(&D8R`C  
M)#`V(#LQ-J!#3TQ534Y3H\$]&H#\$R.\*!"651%4PT@<W1A(&)U9F9E<BLQ#2!L  
M9&\$@ (R0X,`T@<W1A(&)U9F9E<@TZ;&]0<#(@;&1A("AB=69F97(I+'D@.V%.  
M1\*!73U)+H\$)!0TM705)\$4R\$-(&50<B`C)&9F#2!B;F4@.F=0=&-H83(-(' -T  
M82`H8G5F9F5R\*2QY#2!S=&\$@8G5F9F5R(#MS5\$E#2Z!!H%I%4D^@24Y43Z!"  
M549&15(-(&QD82`H8G5F9F5R\*2QY#2!E;W(@R1F9@T@8FYE(#IG;W1C:&\$R  
M#2!S=&\$@\*&)U9F9E<BDL>0T@;&1A(",D.#`-( '-T82!B=69F97(-(&1E8R!B  
M=69F97(K,0T@8FYE(#IL;V]P,@T-.F=0=&-H83(@<W1A('1E;7`Q(#MA1T%)  
M3J!&24Y\$H%1(1:!(24=(H\$))5`T@;&1A(",P,`TZ;#,@<V5C#2!R;W(-(&S  
M;"!T96UP,0T@8FYE(#IL,PT@86YD("AB=69F97(I+'D-( '-T82`H8G5F9F5R  
M\*2QY#0T@:6YY("`[;D]7H\$M%15"@1T])3D<-(&-P>2!Y;6%X#2!B8V,@.FQ0  
M;W`@.W5.5\$E,H%=%H\$A)5\*!934%8(0T@8F5Q(#IL;V]P(#MW1:!.145\$H%1(  
M1:!,05-4H\$].1: !43T\N#0TJ\*BHJH'-705"@0E5&1D524PT-<W=A&)U9B!L  
M9&\$@=FUC<V(-(&50<B`C)#`R(#MP4D545%F@5%)0TM9+\*!%2#\ - (' -T82!V  
M;6-S8@T@;&1A(",D,#@-(&50<B!Z=&5M<`[6E1%35`]2\$E'2\*!"651%H\$I5  
M4U2@1DQ)4%, - (' -T82!Z=&5M<`[0D545T5%3J`D,S"@04Y\$H"0S.`T-(&IM  
M<!"M86EN(#MA4D]53D2@04Y\$H\$%23U5.1\*!71:!'3RXN+@T-('1X="`G<T%-  
M1: !42\$E.1Z!71: !3Z!%5D526:!.24=(5`R@<\$E.2UDZH"<-( '1X="`G5%)9  
MH%1/H%1!2T6@3U9%4J!42\$6@5T]23\$0A)PT-#2HM+2TM+2TM+2TM+2TM+2TM  
M+2TM+2TM+2TM+2TM+2TM#2J@9T5.15)!3\*!154535\$E/3D%"3\$4M5D%,546@  
M15)23U\*@(4%)/0T5\$55)%#0UC:&]K92!L9'@@(S`P#3IL;V]P(&QD82`Z8W1E  
M>'0L>`T@8F5Q(#ID;VYE#2!J<W(@8VAR;W5T#2!I;G@-(&IM<`Z;&]0<`TZ  
M9&]N92!R=',-.F-T97AT(&AE>"`P9"[8W(-('1X="`G4T]-151(24Y'H\$-(  
M3TM%1\*`Z\*`<-(&AE>"`P9#`P#0T@='AT("=N05)&(2<-#2HM+2TM+2TM+2TM  
M+2TM+2TM+2TM+2TM+2TM+2TM#2J@9%)!5TE.)Z!!H\$Q)3D4NH\*!AH\$9!  
M2\$Z@3\$(3BX-#2HJ\*J!S3TU%H%5314953\*!-04-23U,-#7!L;W1P>!"M86,@  
M(#M03\$]4H\$&@4\$])3E2@24Z@6`T@<&AA("`[-5-%H%1(25.@3TY%H\$5615)9  
MH%1)344-(&QD82!B:71P+'@@.WB@25.@24Y#4D5!4T5\$#2!B;6D@8S\$-(&QD  
M82`C)#@P(#MT04),1:!(05.@0D5%3J!214%24D%.1T5\$#2!E;W(@8G5F9F5R  
M(#M&3U\*@1DE,3\$E.1Z!&04-%4PT@<W1A(&)U9F9E<@T@8FUI(&,R#2!I;F,@  
M8G5F9F5R\*\$S-8S(@;&1A(",E,#\$Q,3\$Q,3\$@.VY/5\$6@5\$A!5\*!42\$E3H\$E3  
MH\$-(04Y'140-8S\$@86YD("AB=69F97(I+'D@.T9/4J!03\$]45\$E.1Z!&24Q,  
M142@1D%#15,-(' -T82`H8G5F9F5R\*2QY#2!P;&\$@(#MN145\$H%1/H%-!5D6@  
M82\$-(#P\/'T-<&Q0='!Y(&UA8R`@.W!,3U2@0: !03TE.5\*!)3J!9.J!324U0  
M3\$52H\$.1\*!.14-%4U-!4EDA#2!P:&\$@(#MU4T6@5\$A)4Z!/3D6@5TA%3J!9  
M3U6@2E535\*!)3D-214%31: !Y#2!L9&\$@8FET<"QX(#M"552@>\*!3T533B=4  
MH\$-(04Y'10T@86YD("AB=69F97(I+'D-( '-T82`H8G5F9F5R\*2QY#2!P;&\$-  
M(#P\/'T-8VEN:70@;6%C("`[;4%#4D^@5\$^@24Y)5\$E!3\$E:1: !42\$6@0T]5  
M3E1%4@T@;&1A(%TQ(#M\$6\*/4J!\$60T@;'-R#2!E;W(@R1F9B`[\*&Y/5\*!2

M14%,3%F@5%=/)U.@0T]-4\$Q%345.5"D-(%D8R`C)#`Q(#MAH#V@,C4V+418  
M+S\*3U\*@,C4V+419+S(-(#P\"@.W1(1: !\$6\"RH\$U!2T53H\$&@3DE#15\*  
M3\$]/2TE.1Z!,24Y%#0UX<W1E<\"!M86,@(#MM04-23Z!43Z!404M%H\$&@4U1%  
M4\*!)3J!X#7AL;V]P(&EN>`T@861C(&1Y#2!B8V,@;#-\$\*J!D3Z!71: !54T6@  
M:6YYH\$]2H&1E>:!(15)%/PT@:68@:2Q=,2`[:4:@5\$A%H\$9)4E-4H\$-(05)!  
M0U1%4J!)4Z!!3J`G:2<-(&EN>0T@96Q590T@9&5Y#2!F:6X-( '-B8R!D>`UL  
M,2`^/CX@<&Q0='!X(#MA3%=!65.@5\$%+1:!!H%-415"@24Z@>`T@8W!X('@R  
M#2!B;F4@>&Q0;W`-(#P\"@T->7-T97`@;6%C("`[<T%-1: !42\$E.1RR@0E54  
MH\$9/4J!Y#7EL;V]P(&EF(&DL73\$-(&EN>0T@96Q590T@9&5Y#2!C;&,@(#MV  
M15)9H\$E-4\$]25\$%.5"\$-(&9I;@T@861C(&1X#2!B8V,@;#(-(&EN>`"@.V%,  
M5T%94Z!)3D-214%31: !X#2!S8F,@9'D-(#X^/B!P;&]T<'@-(&IM<\"!L,PUL  
M,B`^/CX@<&Q0='!Y(#MW1:!/3DQ9H\$E.0U)%05-%1\*!Y#6PS(&-P>2!Y,@T@  
M8FYE('EL;V]P#2`\"/#P-#2HJ\*BJ@:4Y)5\$E!3\*!,24Y%H%-5%50#0UD<F%W  
M(#X^/B!M;W9E+'1X,3MX,2`@.VU/5D6@4U151D:@24Y43Z!:15)/H%!!1T4-  
M(#X^/B!M;W9E+'1X,CMX,B`@.W=(15)%H\$E4H\$-!3J!\"1: !-3T1)1DE%1`T@  
M/CX^(&U0=F4L='DQ.WDQ#2`^/CX@;6]V92QT>3([>3(-(#X^/B!S971B=68@  
M.VY/5Z!71: !#04Z@0TQ/0D)%4J!42\$6@0E5&1D52#0T@<V5C("`[:4%+1: !3  
M55)%H%Q/%@R#2!L9&\$@>#(-(' -B8R!X,0T@8F-S(#IC;VYT#2!L9&\$@>3(@  
M.VE&H\$Y/5\"R@4U=!4\*!P,: !3D2@<#(-(&QD>2!Y,0T@<W1A('DQ#2!S='D@  
M>3(-(&QD82!X,0T@;&1Y('@R#2!S='D@>#-\$(-T82!X,@T-(' -E8PT@<V)C  
M('Q@(#MN3U>@83U\$6`TZ8V]N=\"!S=&\$@9'@-(&QD>\"!X,2`[<%54H%QHQH\$E.  
M5\$^@>\"R@3D]7H%=%H\$-!3J!44D%32\*!X,0T-8V]L=6UN(&QD82!X,2`[9DE.  
M1\*!42\$6@1DE24U2@0T],54U.H\$9/4J!X#2!L<W(@(#LH=\$A)4Z!#04Z@0D6@  
M34%\$1: !-54-(H\$9!4U1%4B\$I#2!L<W(@(#MT2\$521: !!4D6@6#\$0.\*`Q,CB@  
M0EE41: !\"3\$]#2U,-(&QS<B`@.W=(24-(H\$U%04Y3H%Q+S\$VH#(U-J!\"651%  
MH\$),3T-+4PT@;' -R#2!B8V,@.F5V96X@.W=)5\$B@0: !03U-324),1: !%6%12  
M0:`Q,CB@0EE41: !\"3\$]#2PT@;&1Y(\" ,D.#`@.TE&H%-/+\*!3152@5\$A%H\$A)  
M1TB@0DE4#2!S='D@8G5F9F5R#2!C;&,-.F5V96X@861C(&)U9F9E<BLQ(#MA  
M1\$2@24Z@5\$A%H\$Y534)%4J!/1J`R-3:@0EE41: !\"3\$]#2U,-(' -T82!B=69F  
M97(K,2`[84Y\$H%-43U)%H\$E4(0T-(' -E8PT@;&1A('DR(#MC04Q#54Q!5\$6@  
M1%D-(' -B8R!Y,0T@8F-S(#IC;VYT,B`[:5.@63(^63\$\_#2!E;W(@R1F9B`[  
M;U1(15)725-%H\$19/5DQ+5DR#2!A9&,@(R0P,0TZ8V]N=#(@<W1A(&1Y#2!C  
M;7`@9'@@.W=(3R=3H\$))1T=%4CJ@1%F@3U\*@1%@\_#2!B8W,@<W1E<&EN>2`[  
M:4:@1%DLH%=%H\$Y%142@5\$^@5\$%+1: !\"24>@4U1%4%.@24Z@60T-<W1E<&EN  
M>\"!L9'D@>3\$@.WB@25.@04Q214%\$6: !3152@5\$^@6#\$-(&QD82!B:71P+'@@  
M.W!,3U2@5\$A%H\$9)4E-4H%!/24Y4#2J@96]RH\",D9F8-(%N9\"`H8G5F9F5R  
M\*2QY#2!S=&\$@\*&)U9F9E<BDL>0T@/CX^(&-I;FET+&1X(#MI3DE424%,25I%  
MH%1(1: !#3U5.5\$52#2!C<'D@>3(-(&)C<R!X9&5C>2`[9\$^@5T6@4U1%4\*!&  
M3U)705)\$4Z!/4J!\"04-+5T%21%.@24Z@>3\ -#7AI;F-Y(#X^/B!X<W1E<\"QI  
M;GD-(')T<PT-<W1E<&EN>2!L9'D@>3\$@.W=%3\$PLH\$&@3\$E45\$Q%H%)%4\$54  
M251)3TZ@3D5615\*@2%525\*!!3EE/3D4-(&QD82!B:71P+'@-\*J!E;W\*@(R1F  
M9@T@86YD(\"AB=69F97(I+'D-(' -T82`H8G5F9F5R\*2QY#2`^/CX@8VEN:70L  
M9'D-(&-P>2!Y,@T@8F-S('ED96-Y#0UY:6YC>2`^/CX@>7-T97`L:6YY#2!R  
M=',-#7AD96-Y(#X^/B!X<W1E<\"QD97D@.W1(25.@25.@4%54H\$A%4D6@4T^@  
M5\$A!5`T@<G1S(\"`[8E)!3D-(15.@05)%H\$Q%1T%,#0UY9&5C>2`^/CX@>7-T  
M97`L9&5Y#2!R=',-#0TJ+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM  
M+0TJH&- ,14%.H%50#0UC;&5A;G5P(&QD82!V;6-S8B`[<U=)5\$-(H\$-(05\*  
M4D]-H\$)!0TN@24X-(%N9\"`C)3\$Q,3\$P,3`Q(#M\$149!54Q4#2!S=&\$@=FUC  
M<V(-#2!R=',@(#M\"644A#0T@='AT(\"=H05!06: !H3TQ)1\$%94R&@)PT@='AT  
M(\"=33\$J@,3(0.30G#0TJ+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM  
M+0TJH'-%5\*!54\*!\"252@5\$%\"3\$4-#2!D<R!>(#MC3\$5!4J!43Z!%3D2@3T:@



M4\$%'10T@(" [ <T^@5\$A!5\*!404), 15.@4U1!4E2@3TZ@0: !004=%H\$)/54Y\$  
M05)9#6)I='`@;'5P(#\$V(#LQ,CB@94Y44DE%4Z!&3U\*@>`T@9&9B("4P,3\$Q  
M,3\$Q,0T@9&9B("4Q,#\$Q,3\$Q,0T@9&9B("4Q,3`Q,3\$Q,0T@9&9B("4Q,3\$P  
M,3\$Q,0T@9&9B("4Q,3\$Q,#\$Q,0T@9&9B("4Q,3\$Q,3`Q,0T@9&9B("4Q,3\$Q  
M,3\$P,0T@9&9B("4Q,3\$Q,3\$Q,`T@+2U>#0US:6X@.W1!0DQ%H\$]&H%-)3D53  
M+\*`Q,C"@0EE415,-8V]S(&5Q=2!S:6XK,3(X(#MT04),1:!/1J!#3U-)3D53  
M#2`@(#MB3U1(H\$]&H%1(15-%H%1224>@5\$%"3\$53H\$%210T@(" [0U524D5.  
M5\$Q9H%-5\*!54\*!&4D]-H&)A<VEC#7ID:78@97%U(&-0<RLQ,C@@.V1)5DE3  
M24].H%1!0DQ%#71M871H(&5Q=2!Z9&EV\*S,X-" [;4%42\*!404),1:!/1J!&  
M\*%@I/5@J6"\R-38-="`G4TQ\*H#\$R+SDT)PT-\*BTM+2TM+2TM+2TM+2TM+2TM  
M+2TM+0" `J0"-(-"-("M&-`I#PD0C1C0H`"I'X7[J8"%\_ \$Q:@9\*3!1\$1\$2`@  
M("`@("`@("`@(" !#54)%,T0@5C(N,`T-("`@("`@("`@("`@("`@ED-  
MGR`@(" !35\$502\$5.(\$I51\$29("`@(\$=%3U)'12!405E,3U(-#9L@(\$-(14-+  
M(\$)55"!42\$4@2D%.+B`Y-2!)4U-512!/1@V6("!#/4A!0TM)3D>;(\$9/4B!-  
M3U)%( \$1%5%)3%,A#0T='9X21C\$01C\*2("T@24Y#+T1%0R!8+5)/5\$%424].  
M#1T=\$D8S+T8TDB`M(\$E.0R]\$14,@62U23U1!5\$E/3@T='1)&-2]&-I(@+2!)  
M3D,01\$5#(%HM4D]4051)3TX-'1T21C>2(%)%4T544PT@(%!215-3(%\$@5\$\  
M455)5`T-!2`@("`@(%!215-3(\$%.62!+15D@5\$\  
M@0D5'24X-`+' [\"D@T0\_  
MT;/F\_ \$Q:@71(25,@25,@02!314-2150@5\$585"!-15-304=%(2#D\_\  
M\D\FM  
M%M`)\$(T6T\*F/A2.%) :D!C2'0J9,@T0^I` (TAT\*E` :0R%^ZD%A?RI`\*`H@`8  
MD?0(:1`0^1BE^VDHA?NE\_&D`A?R@`.B\*X!#0Y\*E`&&D,A?NIV87\H@&@`\*D)  
MD?0(P!#0^:7[&&DHA?NE\_&D`A?R@`.B\*"0@I#^`3\`K)"-#<Z`D!3.F!J56%  
MQ\*D`A:.I,(6DA0\*M&-`I\0D.C1C0J0"%885BA6.%9(5EA698(.3\_R870"Z5A  
MR3SP6.9A3)^"R8G0":5A\ \$0&84R?@LF&T`NE8LD\ \#SF8DR?@LF\*T`FE80`0  
MQF),GX+)A]`+I6//`@YF-,GX+)B])I6/P\$\  
M9C3)^"R8C0`TPK@LE1T`-,  
MSXMX&\*5D96')>)`Z7B%9!BE965BR7B0`NEXA648I69E8\EXD`+I>(5F.\*5E  
MY6:P`FEXA6<8I65E9LEXD`+I>(5H&\*5D96;)>)`Z7B%:3BE9.5FL`)I>(5J  
M&\*5D96C)>)`Z7B%:SBE9.5GL`)I>(5L&\*5D96?)>)`Z7B%:3BE:.5DL`)I  
M>(5N.\*5EY62P`FEXA6\8I61E9<EXD`+I>(5P&\*9G00"-IFA]`(V%I:9G08",  
M.\*9H\_8",A::F9;V`C!`.&\$G\_:0\$\*&\$G\_:0%,;X,\*A:<XIFZ]`(VF;?T`C3BF  
M:\_T`C1BF;'T`C1`.&\$G\_:0%\*&\$G\_:0%,F8-\*&\*9I?8",.\*9J\_8",A:@XIFN]  
M@(RF;/V`C#BF;?V`C#BF;0V`C!`.&\$G\_:0%\*&\$G\_:0%,SX-\*&\*9I?0"-&\*9J  
M?0"-A:FF;[V`C#BF</V`C(6JIFR]@(PXIF[ ]@(PXIFW]@(PXIF0]@(P0#AA)  
M\_VD!2AA)\_VD!3!\*\$2ABF:GT`C3BF: ?T`C86K&\*9L00"-IFU]`(TXIF0]`(TX  
MIF[ ]`(T0#AA)\_VD!2AA)\_VD!3\$B\$2ABF:7V`C!BF:GV`C(6L&\*9000"-IG!]  
M`(V%K4R0A&=%12!B4D%)3BP@5TA!5"!\$3R!93U4@5T%.5"!43R!\$3R!43TY)  
M1TA4/ZE`A?>%^\*6M&&6K&&6LA5>J08"-J\*6G&&6E&&6FA2(82?]I`84DL2(X  
M\208:2"JI:H89:@89:F%(AA)\_VD!A22Q(CCQ)!AI0,7WL`2%]Y`&Q?B0`H7X  
MJ(:2A).EK1AEJSCEK(58JKV`C:BEIQAEI3CEIH4B&\$G\_:0&%)+\$B./\$D&&D@  
MJJ6J&&6H..6IA2(82?]I`84DL2(X\208:4#%][`\$A?>0!L7XD`\*%`\*B&E(25  
MI:TXY:LXY:RJ08"-J\*6G..6E..6FA2(82?]I`84DL2(X\208:2"JI:HXY:@X  
MY:F%(AA)\_VD!A22Q(CCQ)!AI0,7WL`2%]Y`&Q?B0`H7XJ(:6A\*ZEK3CEJQAE  
MK(59JKV`C:BEISCEI1AEIH4B&\$G\_:0&%)+\$B./\$D&&D@JJ6J..6H&&6IA2(8  
M2?]I`84DL2(X\208:4#%][`\$A?>0!L7XD`\*%`\*B&KX2P&\*6G2?]I`86G&\*6J  
M2?]I`86J&\*6M2?]I`86MI:TXY:L89:RJ08"-J\*6G..6E&&6FA2(82?]I`84D  
ML2(X\208:2"JI:HXY:@89:F%(AA)\_VD!A22Q(CCQ)!AI0,7WL`2%]Y`&Q?B0  
M`H7XJ(91A%\*EK3CEJSCEK\*J]@(VHI:<XY:4XY::%(AA)\_VD!A22Q(CCQ)!AI  
M(\*JEJCCEJ#CEJ84B&\$G\_:0&%)+\$B./\$D&&E`Q?>P!(7WD`;%^)`"A?BHAG&\$  
M4\*6M&&6K..6LJKV`C:BEIQAEI3CEIH4B&\$G\_:0&%)+\$B./\$D&&D@JJ6J&&6H  
M..6IA2(82?]I`84DL2(X\208:4#%][`\$A?>0!L7XD`\*%`\*B&LX2TI:T89:L8  
M9:R%8\*J]@(VHI:<89:489::%(AA)\_VD!A22Q(CCQ)!AI(\*JEJAAEJ!AEJ84B







```
x=r*cos(theta)
y=r*sin(theta)
```

where theta runs from zero to  $2\pi$ . If we have a table of sines and cosines, and use the fast multiplication algorithm given in this months 3D article, we have a routine which gives a good circle in around 50 cycles or so per pixel, sans plot.

But let's step back and consider: the above equations all give a "perfect" circle. Is there any way we can draw an "approximate" circle, with a large speed increase? The answer is of course yes, since this wouldn't be a very interesting article otherwise!

Let's say we are standing at a point on the circle, and want to take a step towards the next point. In what direction do we take this step? Consider a line tangent to the circle, touching the point where we are standing. If we take a little step in that direction, we ought to get close to the next point on the circle. The way to calculate the tangent line is to use the derivative, which will give us the slope of the tangent. Taking differentials of equation (1) above we have

$$2*x \, dx + 2*y \, dy = 0$$

or

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

Now, if I am at a point  $(x,y)$ , I want to take a little step in  $x$  and a little step in  $y$ :

$$x = x+dx$$

$$y = y+dy$$

but from the first equation we know that  $dx = -y/x * dy$  so that at each step our iteration is

$$y = y + dy$$

$$x = x - dy*(y/x)$$

This is the basis of our algorithm. You could also of course use

$$x = x + dx$$

$$y = y - dx*(x/y)$$

(this is called Euler's method for solving a differential equation, and these concepts are fundamental to most algorithms for solving differential equations numerically).

Let's start at the point  $x=r, y=0$  i.e. the right-endpoint of the circle. Since we are on a computer we want to take a step of length one in some direction (i.e. step one pixel), and at this point on the circle  $y$  is clearly

increasing much faster than  $x$ . You may remember from the line drawing algorithm that we viewed the process as "keep taking steps in  $x$  until it is time to take a step in  $y$ ". We are going to apply that same philosophy here: keep taking steps in  $y$  until it is time to take a step (backwards) in  $x$ . So our iteration is now:

$$\begin{aligned} y_{\{n+1\}} &= y_n + 1 \\ x_{\{n+1\}} &= x_n - y_n/x_n \end{aligned}$$

Hmmm... we have this little problem now of  $y/x$ . I certainly don't want to deal with floating point. How do we get around this? The trick is to think of  $x$  as a `_discrete_ variable` -- as far as a pixel is concerned  $x$  is just an integer, and has no floating point part. So we are going to treat  $x$  as a constant, until it is time to decrease  $x$  by one! When does this happen? Let's expand the  $x$ -iteration:

$$\begin{aligned} x_{\{n+1\}} &= x_n - y_n/x_n \\ &= (x_{\{n-1\}} - y_{\{n-1\}}/x_{\{n-1\}}) - y_n/x_n \end{aligned}$$

This is where the magic of using a discrete  $x$  comes in. If we make this assumption of constant  $x$ , then  $x_{\{n-1\}} = x_n$  and the above iteration becomes

$$x_{\{n+1\}} = x_{\{n-1\}} - (y_{\{n-1\}} + y_n)/x_{\{n-1\}}$$

If we continue this process over an interval where  $x$  is constant we get

$$x_{\{n+1\}} = x_0 - (y_n + y_{\{n-1\}} + y_{\{n-2\}} + \dots)/x_0$$

When is it time to decrease  $x$ ? When the sum of the  $y$  values at each iteration exceeds the current  $x$ -value the fraction above will be greater than

one, and we will then decrease  $x$ . Like I said, we keep  $x$  constant, until it is time to decrease it! :)

How long do we do this for? In the same way that at  $x=r$   $y$  increases much faster than  $x$  does, when  $y=r$   $x$  increases much faster than  $y$  does. Somewhere in-between they have to be increasing at the same rate, which means the slope

of the tangent line is equal to  $\pm 1$ , i.e. at the point  $x=y$ . At this point we have drawn one-eighth of the circle. We can either draw all eight segments of the circle independently, or else we can use the symmetry of a circle and do an 8-way plot.

TO SUMMARIZE: Here is the basic algorithm

$$\begin{aligned} x &= r \\ y &= 0 \end{aligned}$$

```

a=0
:loop
y=y+1
a=a+y
if a>x then x=x-1:a=a-x
plot8(x,y)
:until x<=y

```

Of course you can refine this in several ways, which will speed things up in assembly. For instance, if instead of `a=0` we start with `a=x`, then the logic becomes

```

a=a-y
if a<0 then x=x-1:a=a+x

```

To do the `a=a-x` you could use a table where  $f(x)=x$ , or you might try something else; here is some code in assembly:

```

LDX R
LDY #00
STY Y
TXA
:loop JSR PLOT8 ;Eight-way symmetric plot
SEC ;Might not need this depending on PLOT8
INC Y ;Y is in zero page
SBC Y
BCS :loop
DEX
STX X ;X is also in zero-page
ADC X ;Carry is already clear
CPX Y
BCS :loop
JSR PLOT8 ;Catch the last point

```

Starting at `:loop` we have  $2+3+3+3=11$  cycles in the best case and  $2+3+3+2+2+3+3+3+3=24$  cycles in the worst case, excluding `PLOT8`. You can, of course, change the program logic around; if `PLOT8` returned with the carry always clear it would be much smarter to start `A` as `A=256-A` and use `A=A+Y` at each step instead of `A=A-Y`. Christopher Jam ([phillips@ee.uwa.edu.au](mailto:phillips@ee.uwa.edu.au)) suggested starting at  $X=Y(=R/\sqrt{2})$  so that the `CPX Y` instruction could be removed (I don't know how this affects accuracy, though).

"Yeah, but how well does it work?" Quite well, as a matter of fact. It will draw a perfect circle for circles with a radius greater than twelve or so. For circles with a smaller radius the sides start to flatten out, and the circle becomes squareish. Interestingly, the "discrete x" approximation improves the result over the straight floating-point calculation significantly!

So this is the best algorithm I was able to come up with for drawing a

circle. If you have any suggestions for improvements or other ideas, please feel free to share them :). As always I must thank George Taylor and Christopher Jam for their suggestions and for helping me work out some ideas.

If you have any particular 2D algortihms/calculations that you would like to see, please feel free to suggest future topics for the 2D graphics toolbox.

Finally, here is a BASIC7.0 program which demonstrates the algorithm:

```
0 REM FAST CIRCLE -- SLJ 9/94
10 GRAPHIC 1,1
15 REM X=Radius
20 X=40:Y=0:TX=X:X0=160:Y0=100
30 DRAW1,X+X0,Y+Y0:DRAW1,Y+X0,X+Y0
40 DRAW1,X0-X,Y0+Y:DRAW1,X0-Y,Y0+X
50 DRAW1,X0-X,Y0-Y:DRAW1,X0-Y,Y0-X
60 DRAW1,X0+X,Y0-Y:DRAW1,X0+Y,Y0-X
70 IF X<=Y THEN 100
80 Y=Y+1:TX=TX-Y
90 IF TX<0 THEN X=X-1:TX=TX+X
95 GOTO 30
100 END
```

=====

## AFLI-specs v1.0

by written by D'Arc/Topaz for Chief/Padua on 28.6.1994

Advanged FLI is name I came up with during the time I coded the first version of AFLI editor. I have never claimed to be the one who discovered this new graphics mode for 64. I myself give the credit for COLORFUL/ORIGO but I am not sure if anyone did it before him (splits have been done but in my eyes they don't count).

In AFLI we can get 120 colors in theory (counted like this  $16!/(2!*14!)=120$ ). When we put red and blue hires pixels close to each other we get a vision of purple - thanks the television.

AFLI is just like FLI with \$08-\$0f (hires value) in \$d016 and a couple of sprites over the first three marks. With \$d018 we change the start of screen memory. And the good old \$d011 for the main work.

AFLI is the same as FLI but we don't use the \$d800-\$dc00 area for the third color. Actually we can't. In normal hires pictures the colors on the picture is ordered in a normal screen (normal



text screen is on \$0400+). The upper 4 bits is the color for bit 0 in picture bitmap and the lower 4 bits is the color for bit 1 in picture bitmap (or the other way...but let us think that was the right way).

For example: a normal hires picture char (8x8 bits)

|   |         |                        |   |          |
|---|---------|------------------------|---|----------|
| 0 | 1234567 | in hires picture where | 0 | 1234567  |
| 0 | *****   | the first spot of the  | 0 | bgggggbb |
| 1 | *** **  | screen has a value of  | 1 | gggbgggb |
| 2 | *** **  | \$68 (blue&green) the  | 2 | gggbgggb |
| 3 | *****   | hires picture looks    | 3 | gggggggb |
| 4 | *** **  | like this ---->        | 4 | gggbgggb |
| 5 | *** **  | b=blue, g=green        | 5 | gggbgggb |
| 6 | *** **  |                        | 6 | gggbgggb |
| 7 |         |                        | 7 | bbbbbbbb |

The bitmap is built just as in a hires picture bit 1 means the pixel is on and 0 that the pixel is off.

In FLI we have built the screen to have badlines on every scanline of the screen. This gives us the possibility to change the screenmemory the picture uses on everyline. Now... when AFLI (and FLI) uses screen memory for colors and we change the screenmemory start on everyline, we can have new colors on everyline.

The screens are usually ordered like this.

```

screen memory used
 0 $4000-$43ff
 1 $4400-$47ff
 2 $4800-$4bff
 3 $4c00-$4fff
 4 $5000-$53ff
 5 $5400-$57ff
 6 $5800-$5bff
 7 $5c00-$5fff
 $6000-$7fff BITMAP (the actual picture data)

```

The number of the screen is considered as the number of the line in 8x8 pixel area.

An example... Here we have cut from the memory showing the first bytes in every screen.

```

screen/rownumber
 00 01 02 03 04 05 06 07 08 09 10 11 12 13 ... 39
$4000 ff ff ff 56
$4400 ff ff ff 67
$4800 ff ff ff 91 ..
$4c00 ff ff ff b3

```

```

$5000 ff ff ff 54
$5400 ff ff ff 8f
$5800 ff ff ff 54
$5c00 ff ff ff 10

```

Actually the \$ff won't have to be there. It will come to the screen anyway. We have the same 'A' on the screen on the fourth mark (\$6018-\$601f).

| BITMAP   |             | AFLI PICTURE (number is the color number) |                                        |
|----------|-------------|-------------------------------------------|----------------------------------------|
| 01234567 | screenvalue | 01234567                                  |                                        |
| 0        | *****       | \$56                                      | 0 56666655 1=white, 0=black, 2=red ... |
| 1        | *** **      | \$67                                      | 1 77767776                             |
| 2        | *** **      | \$91                                      | 2 11191119                             |
| 3        | *****       | \$b3                                      | 3 33333333                             |
| 4        | *** **      | \$54                                      | 4 44454445                             |
| 5        | *** **      | \$8f                                      | 5 fff8fff8                             |
| 6        | *** **      | \$54                                      | 6 44454445                             |
| 7        |             | \$10                                      | 7 11111111                             |

Now the 'A' surely has a lot of colors.

When we code a FLI routine we know that we have succeeded when we get a 3 marks wide area filled with value \$ff on the screen. In FLI the thing is easily taken away; we just fill the three first bytes of a line with empty bytes (\$00). In AFLI the value \$ff is a color. If we try to clear the three first marks, we still have the gray area. WHY? The \$ff value comes to the screen.. so... the \$ff is a color and the upper four bits of the byte is the color for empty pixels. We can not clear the first three marks to wipe the thing off. We have a new lovely problem: we have to put black (or whatever) sprites over that area. This is just timing.

This may look very complicated and I think you will still be asking many questions from me - the text I have written surely ain't the best novel ever written. I'm great in jumping from a thing to another.

--  
 Chief/Padua +44 (0) 757 706791

-----  
 My opinions are not my employers

=====

# Coding Tricks

The following are messages posted to comp.sys.cbm that contain little "coding tricks" that I thought were useful. If you've got any of your own that you want to post feel free to email them to me and I'll post them to comp.sys.cbm - duck@pembvax1.pembroke.edu.

-----  
 From: paulvl@python.es.ele.tue.nl (Paul van Loon)  
 Date: 11 Oct 1994 14:28:49 GMT

Hello everyone,

I really would like to start a thread on your favorite sequences of 6502 code. I hope much people will react so we can build up a library of the most beautiful 6502 code ever seen. I would suggest little code fragments, probably not longer than 10 lines, doing some tiny function.

I propose the following standard:

```

<xxx> <yyyy> "text"
 <aaa> ; comment
 <bbb> ; comment
```

```
"description"

```

where <xxx> is the symbolic name (3 letters?) you give to your code sequence, <yyyy> describes your arguments "text" gives a full name to pronounce for the symbolic name and <aaa> and on are the 6502 instructions you use. "description" is a description of the functionality of your code. The lines with "----" are seperators.

I will hereby start with my all-time favourite code, I discovered it only recently and I will also show some examples of how to use it!

```

B7C "Bit 7 to Carry"
 cmp #$80
```

```
This instruction copies bit 7 of A to C

```

This is really a beauty! It works because `cmp` clears the carry if A is below the immediate value, and sets it if A is higher or same. All values lower than \$80 have bit 7 equal 0, `cmp` will clear C for all values below \$80, and thus will 'copy' bit 7 into carry. All values equal or above \$80 will have bit 7 equal 1, `cmp` will set C for all values above \$80 and thus for this case it will also 'copy' bit 7 into carry!

```

ASR "Arithmetic Shift Right"
 B7C
 ror
```

This instruction does a signed divide by 2

-----

Again a beauty in my eyes! I have puzzled many times who to write the on other CPUs well-known ASR instruction, but I never seemed to get it implemented without an extra register to use like:

```
tax
asl
txa
ror
```

or even without a branch (figure that out yourself!).

With these two beauties I want to give you an idea of what I mean, and please follow me up!.

BONUS. A little routine to clear the screen without erasing the sprite pointers.

```

CLS "Clear the screen"
 ldx #250
 lda #$20
 clp sta $0400-1,x ; 0-249
 sta $0400+249,x ;250-499
 sta $0400+499,x ;500-749
 sta $0400+749,x ;750-999
 dex
 bne clp
```

Clear only the screen

-----

```

From: paulvl@python.es.ele.tue.nl (Paul van Loon)
Subject: Re: Post your favourite little code too!
Date: 20 Oct 1994 12:17:40 GMT

```

```

RSL "Rotate Straight Left"
 B7C
 rol

```

This instruction does a rotation left through 8 bits (rol does a rotation left through 9 bits, 8 bits of A and 1 bit C)

```

Another useful instruction, for a wraparound rol, e.g. for rotating bytes in characters to simulate parallax scrolling.

```

```

A reliable way to create a Straight IRQ?

```

Well, this is the way I've always used, because it's reliable, flexible doesn't mess up, or set restrictions on the display, unlike routines that depend on D011 etc...

This one only observes the VIC chip:

It's some time since I actually coded a routine of this kind, and I'm writing this off memory, so there might be some errors in the following code.

```

<irq is initiated in the usual way, I almost always use FFFE/FFFF instead of 0314/0315 for interrupt vectors, but the routine should work with 0314/315, but delay timing will be different because of the time being up by the kernal>

```

```

<Set up irq vector to label 'irq1' and program desired rasterline with d012 & $ d011 in the usual way>

```

```

<other initiation of program>

```

```

...

```

```

 jmp mainPrg

```

```

;----- Standard routines for all sharp irqs -----

```

```

sharp inc $d012 ;Set interrupt to the next line.
 inc $d019 ;Ready for new interrupt

```

```

;same as lda #1 : sta $d019
 sta storeA
 lda #<sharpirq
 sta $fffe ;Alter interrupt vector
 lda #>sharpirq
 sta $ffff
 cli ;Allow new interrupt even if we
 ;still are in the previos irq call
noploop
 nop
 nop ; We add nop's here so that the next
interrupt
 ; will interrupt while we are executing nop
... ; commands. Since nop commands use two
cycles
... ; the interrupt will at most be delayed 1
... ; cycle.
... ; I think we needed about 11 nop's to be
sure
 ; that execution is interrupted while they
are
 ; run. (I'm not sure about this number)
will
 jmp noploop ; Although we are sure that the interrupt
 ; interrupt before we reach this point, we
 ; add this loop to be on the safe side.
frozen
 ; Imagine if the program happened to be
 ; by a carterigde and later restarted while
 ; executing the nop's.
sharpirq
 pla ; Delete data put to stack by the last
 pla ; interrupt, we don't intend to return from
 pla ; it.
 stx storeX
 sty storeY
 nop ; Now we only have an uncertainty of 1 cycle
 nop ; as to where the interrupt is.
 ; By waiting until the edge of the current
interrupt
 ; rasterline we can determine if the
... ; was delayed by one cycle or not.
... ; (How many nop's that is required to reach
the
... ; edge of the rasterline i don't remember.
 ; You'll just have to find it out yourself)
 ; These nop's may of course be exchanged by
 ; equivelent time consuming instructions.

```

```

 lda $d012 ; get current rasterline
 cmp $d012 ; still on same line = was delayed;
 bne addCycle ; add 1 cycle if not delayed.
addCycle ; doing a branch jump takes 1 cycle more
than ; not doing one.

 ; the rastertiming is now 'straight'/sharp
 rts ; return to routine that cal led sharp

endIrq ; restore a x y
storeA = * + 1
 lda #0 ; For those who don't like self modifying
code
storeX = * + 1 ; this can be changed easy.
 ldx #0
storeY = * + 1
 ldy #0
 rti ; Return from interrupt.

nextIrq
 stx $fffe
 sty $ffff
 sta $d012
 inc $d019 ;or lda #1 : sta $d019 if you prefere
 rts

;----- The actual interrupt -----
irq1
 jsr sharp ;Make interrupt sharp and store A X Y regs.

 <here you put your sharp interrupt dependant code>

 <other code that you need executed this interrupt>

 ldx #<irq?> ;<?> is the next interrupt that is due
 ldy #>irq?> ;1 if irq1 is the only interrupt.
 lda #<rasterline for interrupt>
 jsr nextIrq

 jmp endIrq

;----- Main program -----
mainPrg
 cli ;allow interrupt

 <wait for space or something>

```

```
sei
```

```
<shut everything down and restore what needs to be restored>
```

```
rts ; Or jmp exitCode
```

```

```

By adding this routine:

```
saver sta storeA
 stx storeX
 sty storeY
 rts
```

...you can at any time turn on/off the sharpening by exchanging 'jsr sharp' and 'jsr saver'.

I hope this is what you were looking for. Sorry for not supplying a complete source, but as I said this is all from memory.

PS. My native language is not english and I typed this in a hurry, so sorry for the lousy language / source.

Furhermore, ways of getting an rnd.

\* lda \$d012, only in large complex programs with lots of variable execution times where rnd is only needed once in a while.  
(Never use in raster interrupts of obvious reasons)

```
* lda $d800 ; The 4 upper bits of mem at area $d800 - $dc00 are
completly
 lsr ; unpredictable...
 lda $d801
 lsr
 lda $d802
 ...
```

\* Read the values from the white noise generator to the SID.  
3rd voice set to whitenoise, and read the result.  
(sorry don't remember address to read, think it is around \$d416-\$d41c)

These give you `_true_` random numbers. It's also possible to create 'random' routines that create 'random' numbers based on a seed value.

Anything unclear? Mail me at [s514@ii.uib.no](mailto:s514@ii.uib.no)

Bye...

- Rolf Wilhelm Rasmussen



Equal of Eternity

=====

## C.S. Bruce Interview

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

The following is an interview of Craig Bruce, author of numerous programs such as ZED, ACE etc for the Commodore 64/128.

> What computer experience did you have before the Commodore computers?

Very little. I was 14 at the time, in grade nine, and it was December 1982. My Jr. High school had a few CBM 8032s, but I never actually got to touch one. I took a "mini course" on computers and learned a tiny little bit about what computers were like and about BASIC. To give you an idea of how little I learned, I was entirely incapable at the time of figuring out how to increment a variable ( $X=X+1$ ).

> What was your first Commodore computer and why?

My first computer was a VIC-20. I had the choice narrowed down to a VIC, a TI99-4A, or a Timex/Sinclair 1000(?). (I don't think I had heard of the Ataris or the Apple). I chose the VIC partly because it was related to the computers at school but mostly because it had the most impressive brochure and I had the most information about it. It was theoretically a Christmas present, but it didn't stay in the box very long. I paid half of the \$400.00 price tag and my parents paid the other half.

> How did you learn programming on the Commodore? Did experience from other PC's help?

Ahhh, those were the days. I learned programming from a few sources. The user's guide that came with the VIC was quite helpful, and I read the magazines of the day, mostly Compute!s. I also had a friend who went to high school and used the computers there who knew a thing or two, and two other friends who got VIC-20s soon after me, so we learned from each other.

I also took a relatively informal night course that was offered for programming VIC-20s. By the time I took it, though, I had already learned just about everything that the course taught: BASIC. Then, in the last class, the instructor talked just a little bit about machine language, just enough for me to understand what was in the VIC-20 Programmer's Reference

Guide. I learned 6502 machine language shortly after that.

Experience from other PCs didn't really factor into things, since I had no experience with any other PC. However, when the time came to learn about other computers and other programming languages (in high school and bachelor university), I had an enormous advantage over the other students, since I understood so thoroughly how computers worked because of my VIC-20 experience.

> What other interests, besides hacking on the Commodore, do you have?

Not many. I don't get out much and I have only a handful of friends. I spend most of my time sleeping, watching TV, net surfing, doing school work, and/or hacking on Commodores. Hacking on Commodores is in my blood. While I'm doing these other things, I'm usually thinking about hacking on Commodores. You might say that I'm a sterotypical total computer geek. I do like biking, though. I bike to school every day. And music.

> What is your feelings on the demise of Commodore?

Losing Commodore was a little sad, but as someone said on the newsgroup when Commodore went under, "What has Commodore done for you lately?". We 8-biters lost all support from Commodore long before its demise. I certainly don't blame them; 8-bit computers are a thing of the past and there wasn't a big enough market to support a company the size of Commodore.

> Do you see anything in the future that signals anything that will extend the useful lifetime of the Commodore 8-bits?

IMHO, there are only two things that 8-bit computers need to survive in the hands of hobbieists indefinitely: serious system software and modern hardware peripherals. Some serious system software has begun to surface recently :-), and Creative Micro Designs has been providing modern peripherals for us to use. New application programs are needed too, but I'm assuming that hobbieists + serious\_system\_software --> serious\_new\_application\_programs. (Perhaps this is a bit self-serving).

Eight-bit computers have two big advantages over bigger PCs: a much lower price and a much higher understandability quotient. Both of these are very important to hobbieists.

> Do you feel that with the addition of newer periphials that are gradually superceeding the CPU's job (REU, RamLink, SwiftLink etc...) that the Commodore 8-bit standard machine is no longer standard??

Indeed, there are lots of options. But I think that this is a good thing. The original Commodores are quite limited, and this modern hardware is

needed

to allow the Commodores to remain useful in the networked world. An important feature of all of these new products is that you can flip a switch or pull out a cartridge and you're back to your little old standard Commodore.

Of course, who really wants to do this.

The reason that I have stayed with my little Commodore for all of these years

is that I believe that it still has quite enough power (using modern peripherals and expanded memory) to do what I require of it. For example, it's quite possible to have a nice little 17K text editor that can edit huge files and be very useful. You don't need a multi-megabyte program with all kinds of snazzy features that requires a monstrous machine to run on to do this. These multi-megabyte programs are simply bloated and poorly designed.

> Will there ever be an update to Zed? (a question asked on a lot of the commercial providers)

Yes. I've been promising this for a long time, but the right time to do this is finally near.

> What is the process that you use for writing your programs?

I'll start this answer with a Unix-fortune quotation:

"Real programmers don't draw flowcharts. Flowcharts are, after all, the illiterate's form of documentation. Cavemen drew flowcharts; look how much good it did them."

For complicated algorithms, I'll sit down write some pseudo-code, and I always plan and write out complicated data structures. But other than this, I usually just sit down and write code, after kicking ideas around in my head

long enough for me to know what I have to do.

> It's been noticed that you have a "fanaticism" about speed in your programs.

Can you elaborate on this?

Guilty as charged. As I said above, I despise bloated software that needs a mega-machine to run on fast enough. I like software that is sleek and mean, and I have an axe to grind that little 8-bit Commodore computers offer quite enough computing power for most applications that most people would use them for. The exceptions are number-crunching, huge-data processing, and heavy computation. However, for most interactive programs, an 8-bit processor is quite enough. So, I grind my axe by producing fast programs. Arguably, that

effort is sometimes misspent (like in the printing to the screen in ACE -- I still have a few more tricks up my sleeve though...), but I like to go a

little too far sometimes to make people go, "Wow! I didn't know this little machine could do that so fast!!" I like to upset the notion that you need a huge machine to get adequate performance. (In fact, sometimes the opposite is true, since programmers assume that they can be extra sloppy when programming for huge machines).

As a user, I like crisp responsiveness. This is a feature of personal computers that can sometimes be absent on big multi-user virtual-memory machines.

I also have a big thing against backwards compatibility ("hysterical raisins"). This is a significant cause of software bloatedness. This is one reason that ACE, for example, was designed from scratch rather than with the pre-set limitation that all BASIC-compatible programs should run with it (a la CS-DOS).

> Is there anything that you find particularly useful / handy about the Commodore's architecture?

Yeah, it's simple.

> And the corollary: Is there anything particularly annoying?

Yeah, it's limited.

> What is currently in the works / planned??

My Commodore job queue looks like the following:

1. Update my Unix VBM file filter. Make it produce a new format of VBM files with run-length encoding compression. Investigate LZW(?) compression.
2. Work on ACE release #13: Internal cleanup. Reorganize the internal memory usage, add features to the command shell, clean up memory and device management inside the kernel, update the VBM program, add a SwiftLink device driver, make a simple glass-tty terminal program.
3. Develop a new portable archiver format and write a C program for Unix, ".car" format.
4. Write my next article for C= Hacking, which will be about the detailed design of a distributed multitasking microkernel operating system for the C128. This article will also include a minimal multitasking implementation.
5. Work on ACE release #14: Port Zed to ACE. Get the basic editing features going.
6. Work on ACE release #15: Finish the ACE assembler. Add the file-

inclusion,

conditional and macro assembly features. Make it accept more dyadic operators in expressions, fix the label typing, make it generate relocatable executable code modules, and make it handle modular

compilation

("o" files).

7. Work on ACE release #16: Archiving. Update the "bcode" and "unbcode" programs to support uucode, nucode and hexcode formats. Toss the old "uuencode" and "uudecode" programs. Implement "car" and "uncar" programs.

Look into "zip" format.

8. Start on B0S, the distributed multitasking microkernel operating system for the 128.

(Actually, some of these things will be done by the time that C= Hacking comes out).

Keep on Hackin'!

---

## Aligning 1541 Drives

by Ward Shrake (taken from comp.sys.cbm)

A discussion regarding Commodore 1541 disk drive alignment procedures, with suggestions.

Background information.

The best way I've ever seen to consistently and reliably get a 1541 disk drive aligned perfectly, was caused by copy protection. It is sort of appropo that copy protection, which usually causes the "head knock" problem that puts drives out of alignment in the first place, should also be able to solve the problem it created.

An older version of a disk utility program, ("Disector" v3.0, as I remember it), had copy protection that would not let you load the disk up unless your disk alignment was perfect. While initially loading itself, it would search and search, never quitting, until it found what it was looking for, exactly where it was looking for it. It would stay in an endless loop, searching forever, never making it to so far as the first screen. This essentially "locked up" the computer, if the program thought the disk it was on was an illegal copy.

This quickly became the most hassle-free, no-worry alignment program I've ever seen. I have seen and used most of the others; this method beat them all, no contest, in my opinion.

The other programs, the ones made for aligning your drive, never consistently worked acceptably well, in my experience. Other technical users apparently feel the same way about them, as the "General FAQ, v2.1" on Commodores points out. They would work OK part of the time, or on part of the drives you tried, but not all, I found. Or they would say you now had a perfectly-aligned drive, but some difficult copy protection schemes would still not load and run on the newly tuned-up drive friend. A friend of mine, now deceased, once had a drive no alignment program could fix. We tried everything we could find. After aligning it with a given method or program, some programs would load that would not load before, but others would now no longer work, that used to work before. All in all, it was very frustrating, and the general feeling was that there has to be a better, easier, more reliable way to do this.

All an alignment program has to do, is to make sure that when the disk drive says it is precisely at a given track's physical location, that it is really there, centered on that track.

There are other Commodore adjustments, but alignment seems to be, by far, the most common problem. Disk drive rotational speed can be adjusted, but it usually is not the problem. In fact, I've seen more than one drive, that when adjusted to read a program-reported "perfect" 300 rpm rotation speed, they quit reading disks; requiring speed to be set at a reported 310 rpm, to work again. The end stop gap can also be adjusted, but I've never seen it be the real culprit with a non-working disk drive. Your experience may vary, of course, but I've always found that it is best to concentrate on alignment first, then fool around with the other adjustments ONLY after alignment is truly corrected, and only if it still refuses to work properly.

Once alignment is corrected, there are methods available to insure that it stays that way. For instance, you can have the stepper motor's pulley mechanically pinned to its shaft, instead of merely relying on the factory's interference fit to hold it. Commodore 1541 drives were made to be self-aligning, apparently, which would be fine if "head knocking" protection schemes were not around. Since they are, the pulley should, ideally, not be allowed to turn on its shaft, which is what causes misalignment problems.

How I used to align 1541 disk drives....

To precisely align a given 1541 disk drive, I used the old, unbroken copy I had of Disector (v3.0, I think), and followed these steps. With power to the drive off and disconnected, you first took off the upper and lower halves of the outer plastic casing of the drive. This exposed the electronics inside.

You then found and loosened (but not removed!) the two stepper motor mounting screws, which are on the underside of the disk drive's internal mechanisms. After that, you hooked the power cable back up, and hooked the drive to the computer like it normally is.

Once you've done this, you set the drive up on one side, so that you can (carefully!) reach into the mechanism, to physically rotate the stepper motor, which would normally be on the bottom of the drive. You type in the program's loading instructions on the computer, and you then wait until the screen went black (copy protection searching for certain info on the diskette). This is where the program "locks up," with the unaligned drive.

Once the program is loading, but stuck and unable to find what it wants, you reach into the mechanism, very slowly and carefully, turning the stepper motor a slight bit in either direction, and stopping. Tiny adjustments are a lot; don't overdo it. Be patient; don't go too fast, or move it too much! You watch the screen carefully, and listen to the drive's sounds.

When you have rotated the stepper motor to the proper place, the sounds and the screen will act a little different, perhaps only slightly so. Wait a second, not moving the stepper motor at all. When you are right on, alignment-wise, the program will find what it is looking for, and the program's main menu will appear.

Once the main menu has come onto the screen, you have a perfectly aligned drive. Then you have to retighten the stepper mounting screws, being very careful not to accidentally move the motor in the process. Hold the motor firmly while retightening both screws in small steps, alternating back and forth between them until they are both tight. The rotational force of the screws turning, forces the motor to move some, so watch for it.

With this method, using a specially-prepared disk, I always got perfect results; everything would load, every time, from then on. (Assuming that the disk was formatted with a good drive to begin with; any disks you made recently, on your badly-aligned drive, may not load after the alignment procedure. Transfer the info on these disks, to a second, known-good drive, before you do this procedure. This is normal, however, no matter what method you use to align a bad drive.)

Here's the problem with this method...

This procedure only works with a special disk, one that is no longer available. With the special disk, alignment is quick, hassle-free, and it always gave excellent, reliable results the first time around. Without the "perfect" disk, this procedure is worthless. This is obviously a problem, since the method relies on a disk that is no longer available to the public. You can't make your own, because you don't know if the disk drives you are using, are truly perfect to start with! Disks made by users, on Commodore equipment, never worked; they just matched your drive's alignment to that of

someone else's equipment, which may be borderline bad to start with.

Here's what I suggest to solve this...

Your mission, should some hot programmer out there choose to accept it, is to create a program that will create a "special" disk, and a Commodore-compatible program to try to read that special disk.

Ideally, the Commodore-compatible reading program would be short and simple enough to fit inside 8k of memory, so it could fit on a cartridge. This would allow it to work, even if a user's disk drive would not load programs anymore. It could still be stored on a diskette, too, with a little planning.

Theoretically, once you had the specially-formatted diskette, and the program on cartridge, you would only need a screwdriver to take the drive apart, and a Commodore microcomputer to run the program on. No other special tools would be needed, and very little technical knowledge would be required; just some general safety tips, because you are working around sensitive electronic parts, with wall current coming into the drive itself, at least on older 1541's.

Why should a programmer go to all the trouble?

I'm sure there are a lot of people out there who could use this, if some hot programmer should decide to write it, and make it available to the rest of us. There are always programmers out there, somewhere, eager to show off their computer skills, and their creativity. It is one of the things that makes the computer community so great in the first place. (If people out there can write IBM-to-Commodore disk file readers, this should be a breeze!)

Techies should appreciate it as a great, reliable and cheap way to align troublesome disk drives, and those people with a C64 in a closet would sure appreciate their technical buddies getting their dead systems going again!

Description of what I have in mind, as to how it should work.

You need a Commodore computer (the 64 is most popular), one 1541 disk drive that needs alignment (or that you want to check), a screwdriver to open the drive up, a specially-prepared disk used only for alignment purposes, and a computer program that would run on the Commodore that would look at and analyze the information that is on the specially-prepared diskette, as the computer program tries to read it.

OK. Here's where it gets cute.



The problem with most disk alignment methods, as I see it, is that it relies totally on technology that the Commodore has available; trying to create a special disk on a 1541, I just don't see as being realistic, or the best way to do it. The 1541 has many limitations, compared to some other disk drives which operate on other computer platforms. Don't get me wrong; I love Commodore computers, and have for years. But, realistically, the 5.25 inch drives found in say an IBM machine, are just plain better in many ways than the 1541. They are made to hold much more information, and to do that, they have to be much more precise in doing so than the 1541 was ever designed to be.

If a person were to do this, I would suggest that they write an IBM program that would use a high density, 1.2 megabyte capacity, 5.25-inch type of disk drive to create the special diskettes, which the 1541 would later read.

Doing this would allow the creation of very thin tracks on the diskette's surface, spaced closely together. This would, within the limitations of the 1541's read head, allow the Commodore to "see" precisely where it currently was, to one side or the other of some "centered" position. The advantage of thin tracks, widthwise, is that the read head won't see them at all, reliably, unless you are exactly, perfectly right on top of them. Another advantage to this, again within the limitations of the 1541's read head (whatever that may be), is that left or right of center, the head would likely pick up the next track over, letting you know you were off by a certain amount automatically.

I hope I'm making myself clear, in my explanation of this. If I am not, Email

me with your questions, and I'll try to answer them better, and/or update this file, to entice someone else to work on this. I really would like to see

it done. (Current Email address, as of Sep 94: wardshrake@aol.com on the Internet, or just WardShrake on AOL. Will soon have a Compuserve Email address, too: I'll be user 75207,1005 there, or 75207.1005@compuserve.com on the Internet.)

Anyway, let's continue. With the IBM creating a specially-made disk just for this one purpose, you would not even have to worry about following any standard formatting procedures. No user-stored data would ever be written to the diskette, so standard sectoring could be safely ignored. You could create

any signal or sectoring scheme you like, as long as the IBM could create it, and the Commodore could read it; and you'd be writing both programs, anyway, making this easier to insure, right?

I can hear some die-hard Commodore users saying, "I hate IBM's" or "I don't even have an IBM" or some such. Fine. Not a problem. If all the IBM-compatible program did was to create a special floppy disk, once, then quit, you would not even need to OWN an IBM, you'd just need to be able to USE one for a few minutes.

Even if you don't have access to one at work, and don't know of anyone who has one to lend you, I will stick with this suggestion, because I know that some businesses that make photocopies often also rent IBM's and Mac's on an hourly basis, for very little money. My local Kinko's copy center rents them both at \$10.00 an hour. You would only need it for a few minutes or so.

The diskette-creation program would only need a few minutes to run, to make up a special disk, so you'd only be paying for a good quality, blank high density floppy, and ten or fifteen minutes of rental time, tops. The copy center person may even be able to start up the floppy-based IBM program for you, if you don't know how to do it yourself. That should come to \$5.00 or less, even if you don't own or normally have access to an IBM compatible computer! You can't beat that, for a utility to align equipment!

OK. In overview, you'd need to use an IBM-compatible computer, just long enough to load an IBM-compatible program which would create one special, 5.25" diskette, perhaps on a high density floppy. You would then open up your Commodore drive's case, and start up a special program on your Commodore 64, to read the created diskette. (Again, an 8k Commodore program would fit very easily on a cartridge, for easiest loading and running.)

While the computer and drive were running, you would (very carefully, and observing safety precautions) loosen the stepper motor's screws, and slowly turn the motor clockwise and counter-clockwise, until the Commodore program's screen info told you that you were exactly where you should be, right over the proper track. Not to the left or right of it, but in perfect alignment.

Because the Commodore disk-reading program would be "on" constantly, and reporting any small changes to you via information on your screen, you would only have to take a few minutes of fiddling, doing a simple, non-technical turning of the stepper motor, to get the drive aligned. The two computer programs that would make up this package would be doing most of the work.

I imagine a drive could be perfectly aligned, and back in running order, in fifteen minutes or less. Five, if you paid attention to the process, and had some practice before. Remember, this is based on an alignment procedure I really used to do, using a heavily-protected diskette, so I am extrapolating from my personal experiences, even though I'm talking about a theory here.

I don't see where there would be any easier, simpler method of doing a disk alignment. The user wouldn't even have to know a thing about tracks and sectors; they would just loosen two screws, following some instructions, and turn the motor. What could be any easier?

The program could, if it was really creative and well-done, tell them to rotate the motor clockwise or counter-clockwise (as they face it), to dial the motor precisely in. Tracks to either side of an arbitrary (track 18?) center position would say to go one way, tracks on the other would say the reverse. When you turn it too far one way, it would reverse its instructions

to you; you would know you were very close then. When you were "right on," the program would tell you so. You'd lock the screws down, carefully, and as long as you hadn't jiggled the motor when you tightened it back down, you would be all done!

How much easier could it be, right? (On the final user, that is!)

If anyone is interested in doing this, or goes out and does it, please let me know via Email. I'd like to hear about it. Again, it would be something possible, useful, and a really neat trick. I know there are people out there that program on both the IBM and the Commodore; the various cross-reading programs attest to that, well enough!

Ward Shrake  
Covina, California

=====  
-----END-----  
===

From:  
<http://codebase64.org/> - **Codebase 64 wiki**

Permanent link:  
<http://codebase64.org/doku.php?id=magazines:chacking9>

Last update: **2015-04-17 04:34**

