```
     CCCCCC       HH    HH    AAAA    CCCCC KK  KK IIIIII NN   NN  GGGG
    CC   ====     HH    HH  AA    AA CC     KK KK    II   NNN  NN GG
    CC   ===      HH    HH  AA    AA CC     KKKK     II   NN NNNN GG
    CC           HHHHHHHH  AA    AA CC     KKKK     II   NN NNNN GG GGG
    CC   ===      HH    HH  AAAAAAAA CC     KK KK    II   NN  NNN GG   GG
    CC   ====     HH    HH  AA    AA CC     KK KK    II   NN   NN GG   GG
     CCCCCC       HH    HH  AA    AA  CCCCC KK  KK IIIIII NN   NN  GGGGG
```

Volume 1 - Issue 2 - April 22, 1992

==========

Editor's Notes:
by Craig Taylor (duck@pembvax1.pembroke.edu)

Eeegh! - When I first started this I never realized how much work it'd be. I'm glad of the reception it's gotten from the Commodore community at large. I'd like to thank each of the author's in this issue and last for their work they've put into it as well as their time.

Please note that all files, documentation etcetera associated with C= Hacking and whatnot contained within are also now available at tybalt.caltech.edu via anonymous ftp under the directory /pub/rknop/hacking.mag. Any updates to files contained within or corrections will be posted there as well as mentioned here. Currently it has the correct 1st issue and (soon to be) 2nd issue. Also Robert Knop's file bmover.sfx is there for the Banking Geos article in this issue.

It seems as if we're averaging about 2 months for each issue and hopefully we'll keep that rate during the summer but due to an internship (I'll hopefully get) I may not have net access during the summer. In that case it'll be delayed until after I get back to school in the fall.

Also, if you've got any ideas for articles or have written a program that is unique that you'd be interested in documenting and p'haps letting other people see some of the tricks of the trade then please send any queries to duck@pembvax1.pembroke.edu.

********** WARNINGS, UPDATES, BUG REPORTS, ETC... **********

Please note that in the last issue when the undocumented opcodes were discussed that they are _VERY NON-STANDARD_. And any future accelerator boards for the C=128 or C=64, in all likelehood, _will not work_. Zip's board [when are they ever gonna finish it?] for the C=128 will be based on a similair processor to the 8502 and is practically guarenteed not to work with the undocumented op-codes. If you plan to release any ML programs for general use PLEASE be aware that they may be in-compatible with future systems.

==========

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

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

==========

In this edition we've got the following articles:

Learning ML - Part 2

Yes, the infamous learning machine langauge tutors by Craig Taylor (duck@pembvax1.pembroke.edu). In this session we examine indexed addressing and it's usefulness in printing strings of characters.

8563 : An In-Depth Look

This article documents and details the 8563 in-depth. It covers all available registers, documents their functions and suggested methods of getting certain effects. Also covers how to read and write to 8563 registers as well as read the 16k or 64k of memory that contains the VDC char-set, screen memory etc. Written by Craig Taylor (duck@pembvax1.pembroke.edu).

The Poor Man's Way to Getting Files from MS-Dos Diskettes

Now there's a way to transfer files of any length from MS-Dos diskettes using a public domain program that will only read files of 43k or less and a IBM program to split the files up. There are better ways, but if you don't want to pay for Big-Blue Reader this is one method to go. Written by Mark Lawrence (9152427D@Levels.UniSa.Edu.Au).

Banking on Geos

GEOS 128, being an extended and expanded version of GEOS 64, provides a contiguous block of application space in a single RAM bank. The "standard" programming documentation makes few references to the use of other banks in GEOS. This article describes accessing other RAM banks (including RAM banks 2 and 3 for 256K expanded 128's) under GEOS128, using both the GEOS Kernal routines and more direct methods. By Robert Knop (rknop@tybalt.caltech.edu).

Dynamic Memory Allocation

Written by Craig Bruce (csbruce@ccnga.uwaterloo.ca) this article examines how to implement and use dynamically allocated memory that will allow your programs to better utilize all of the available memory on your C=128, including expansion memory. These routines are extracted from the Zed-128 program which is a text editor that can edit 600KByte files on a 512K expanded 128.

==========

Beginning ML #2
by Craig Taylor (duck@pembvax1.pembroke.edu)

Last time we introduced the definition of what exactly Machine Language / Assembly Language is along with an example of clearing the screen in Machine Language.

Now, in this issue let's print my name (and later your name). Looking at the code from last time the following assembly source jumps to mind:

---
print_1.asm:

```
          lda #147         ; clr/screen code
          jsr $ffd2        ; print
          lda #'C'         ; code for ascii "C"
          jsr $ffd2        ; print
          lda #'r'
          jsr $ffd2
          lda #'a'
          jsr $ffd2
          lda #'i'
          jsr $ffd2
          lda #'g'
          jsr $ffd2
          lda #32          ; code for space
          jsr $ffd2
          lda #'T'         ; print my last name....
          jsr $ffd2
```

```
            .
              .            (ad naseum...)
                .
          rts
----------
```

  Now, for short strings like "HI!" that might be fine but if your name is something like "Seymour Johnson the third" it can get a little bit ridiculous in terms of the amount of memory and the amount of typing (eegh! - typing!) involved. There's an easier way.

  It's called indexed addressing. What is this you say? Let's first take a look at the above program using indexed addressing and then explain it.

```
------------
print_2.asm

          ldy #$00
   loop   lda string,y
          jsr $ffd2
          iny
          cpy #numchars
          bne loop
          rts

 string   .byte 147
          .ascii "Craig Taylor"

 numchars = *-string

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

  Hmm, looks a little bit confusing 'eh?

  What we're doing is using the register y to act as a pointer to grab the y'th character in what is at memory location STRING. If y is 0 then we'll get string+0 which happens to be a 147.

  The .byte and .ascii directives are not real instructions the computer understands. What happens is that your assembler sees that you want data put at those locations so it convert 147 and "Craig Taylor" to numbers and puts them at the proper locations, relieving you the burden of doing it.

   The numchars = *-string looks confusing at first... obviously, numchars stands for the number of characters we need to print out but how is it being figured? Most assemblers keep the current location in memory it's assembling to in something called a program counter of PC. Most assemblers also will let you have the value at assembly time by referencing it using the "*" symbol. "string" is already a symbol that has been set an address in memory and after assembling the .byte and .ascii instruction "*" will be equal to the next address that the assembler would put any instructions at, had we had any. Now, *-string basically is saying to the compiler, look, take the current program counter (where it's assembling to) and subtract it from where the symbol string starts at (which it just assembled a while back). This should be then, our number of characters we have.

WALK-THROUGH:

  Register Y is initially set to zero in the first instruction, as we want to begin with the first character. The first character is at string+0, not string+1. This may seem a little bit odd at first, but try thinking of it this way:

     Take, for example, 3 diskettes. Put them in a row and call the one on the left "string" (or some other name). Then point at "string+1", "string+2"..

Notice there's no "string+3" even tho' there's 3 diskettes? This may seem a little bit strange at first, but after thinking about it a while you'll begin to understand. In machine / assembly language, you typically count starting from zero, in the real world, typically from one.

The lda string,y instruction is telling the computer to reference string as if it was an array, get the byte at location string + y, or for you basic programmers thinking of string as an array of bytes (with no bounds checking) string[y]. Thus, the accumulator is equal to the yth byte starting from the location string is defined to be.

We then call the routine Commodore so graciously provided for us that prints out the contents of the accumulator. Now, some routines, as we'll see in other editions, are not so nice and may change the value of the accumulator, the x and y registers. However, Commodore was extra nice and the routine at $ffd2 is guaranteed not to change any of the registers.

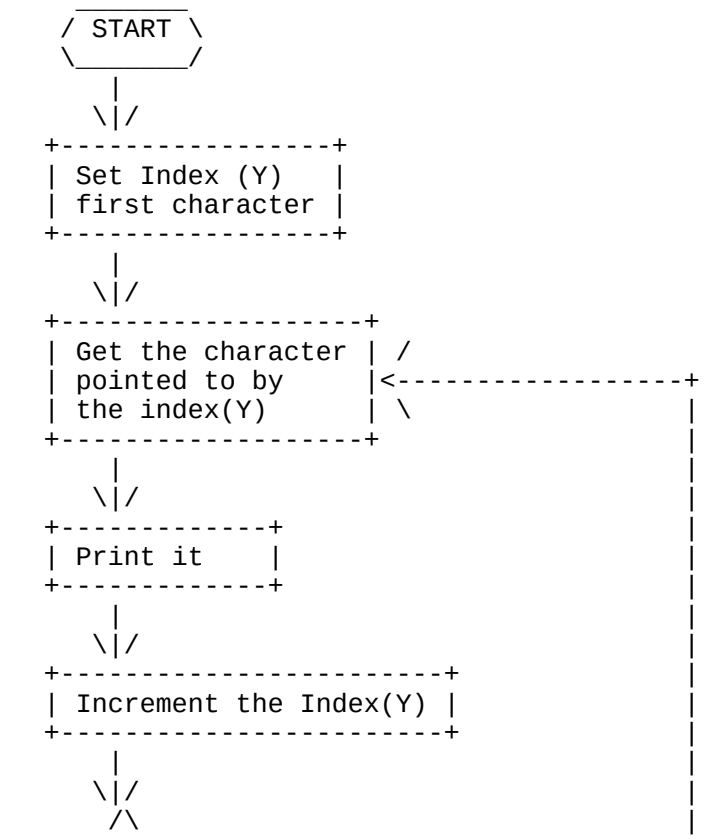
The routine then "iny". What is this? It "INcrements the Y register". INX will "INcrement the X register". The X and Y register can not have any math performed on them other than increment and decrement operations (ie: adding one and subtracting one). The only register that allows addition or subtraction is the accumulator. However, in this case we just want y to point to the next character, the next byte so "INY" serves us fine.

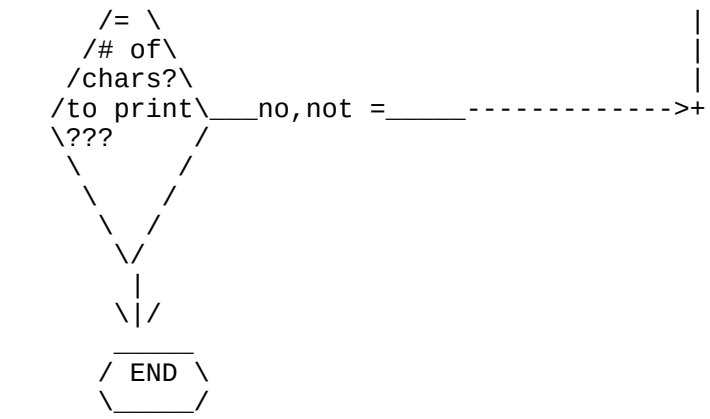
We then "ComPare Y" register to the number of characters in the string. Notice the # sign. If we hadn't have had that there, it would've tried to look at whatever memory location numchars was defined for. Numchars was set up to hold the number of characters in the string, not be a pointer for a memory location.

Now that we've compared it, we "Branch if the last comparison was Not Equal" back to where loop is at (in this case, where we load a with character again).

If it was equal we fall through to the RTS where we return from our little program.

Basically, we've got something like the following flowchart:





Indexed addressing is used *very* often in assembly language. Try playing with the second program and experiment with it until you understand fully what is going on. Next time we'll look at how to access some of the diskette routines and to display a file on disk.

==========

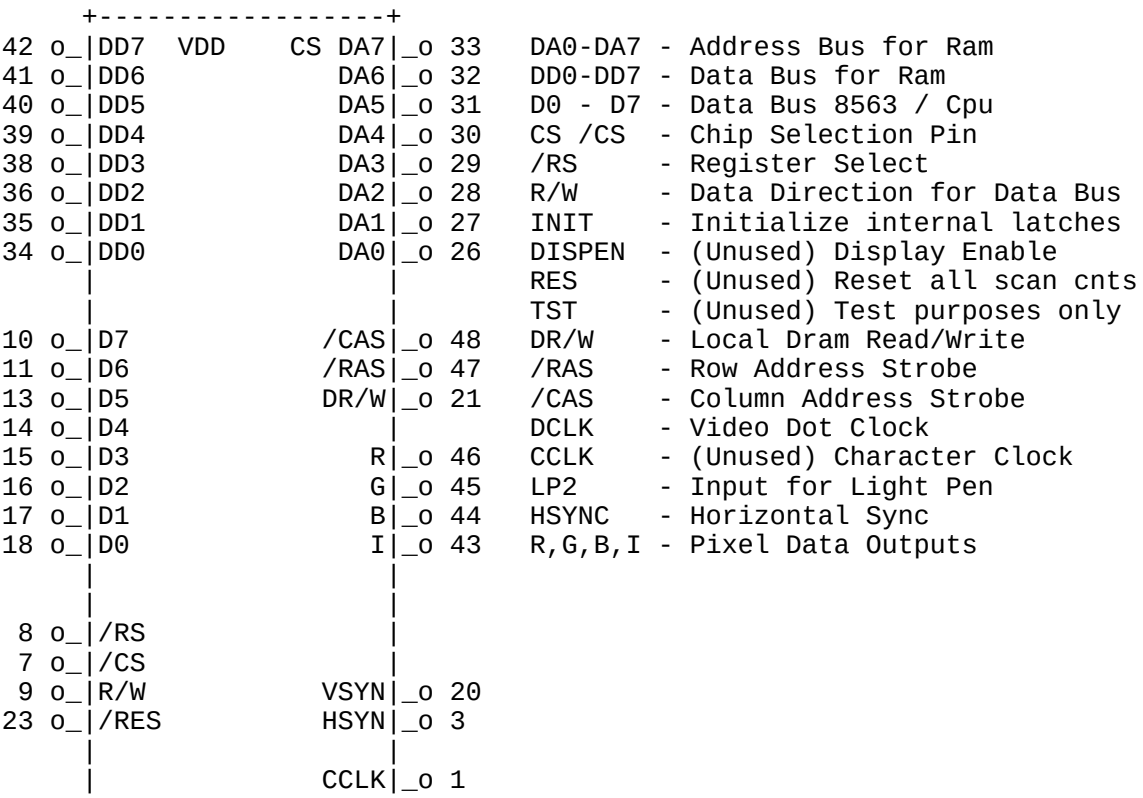
An In-Depth Look at the 8563 Video Chip on the C= 128

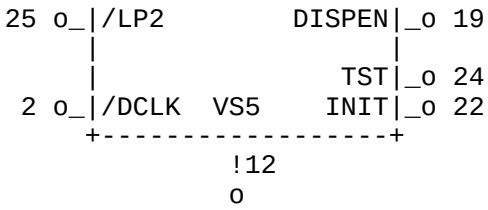
by Craig Taylor (duck@pembvax1.pembroke.edu)

Due to the article in the last issue by Craig Bruce (csbruce@ccnga.uwaterloo.ca) and some letters from people asking about how the 8563 Video Chip works and more technical information this article will attempt to present as much detail as possible on the 8563 chip & it's various capibilities.

! Hardware Aspects: !

The following is a physical layout of the 8563 and the available pin outs:

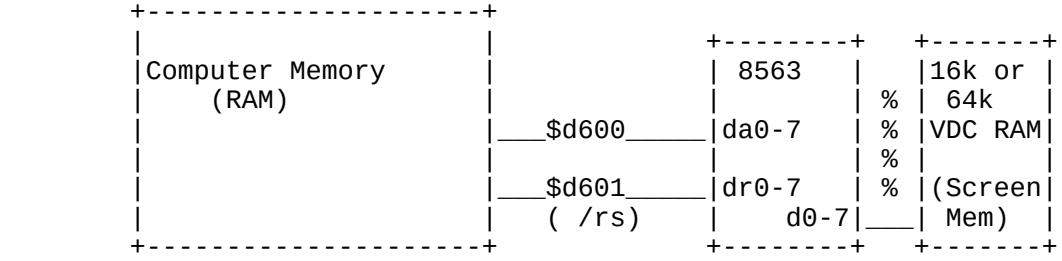




Taken from Pg. 596-8 C=128 Programmer's Reference Manual Publ. Feb 1986 Bantem Books

How Commodore Hooked It Up!

Now, the 8563 is hooked up to the computer via the following method:



Confusing 'eh? (The %'s represent control signals that also are used).. What basically happens is that every time the computer wants to access the 8563 to program or change one of it's numerous registers it has to store the register number to $d600, then loop until the 7th bit of $d600 changes to a 1. Once this is done, you can then read or write a value to/from $d601.

Commodore also employed the MMU (Memory Management Unit) to manipulate pages of memory and thus, the 8563 only shows up in the I/O page (usually referenced as Bank 15 or a value of $00 in the MMU Register at $ff00) or in pages that the I/O section of memory is enabled.

The register at $d600 in the I/O space of the C=128 is laid out as follows:

Bit Position:

7	6	5	4	3	2	1	0
Status	LightPen	VBlank	-----Unused----			------Version #--------	

When a value is placed in $d600 instead of putting the value in Status, LightPen bits etc, the value reflects which register # is requested. Bit 7 of this register (Status) will then return a binary 1 when $d601 reflects the actual value of the register just poked to $d600. (See the ML routines for storing and reading values to/from registers at the end of this article). When a value is first place in this register, $d600 bit 7 is equal to a zero.

Bit 6, is used to indicate when new values have been latched into the lightpen registers (16-17). Bit 5, VBlank refers to when the 8563 is in the period known as "Vertical Blanking Period". Usually, however this bit is seldom referred to as updating the 8563 is usally too slow to make use of this for any special effects.

Bits 0-2 return a version # of which %000 and %001 are the known versions out. Early 128's will contain the value of $0 while later 128's will contain the value of $1. Note that there are slight differences between the 8563's, in that register 25 (horizontal smooth scoll register) requires different settings.

The register at $d601 returns the value of register # that has been written into $d601 (when bit 7 of $d600 = %1). Note that storing a value here will also do a write into the register # selected. (Refer to the ML routines for storing and reading values to/from registers at the end of this article for an example).

Register Definitions

Reg#	7	6	5	4	3	2	1	0	Description	Notes
0	HzT7	HzT6	HzT5	HzT4	HzT3	HzT2	HzT1	HzT0	Horizontal Total	^1
1	HzD7	HzD6	HzD5	HzD4	HzD3	HzD2	HzD1	HzD0	Horizontal Displayed	^1
2	HzS7	HzS6	HzS5	HzS4	HzS3	HzS2	HzS2	HzS0	Horizontal Sync Position	^1
3	VSW3	VSW2	VSW1	VSW0	HSW3	HSW2	HSW1	HSW0	Vert/Horiz. Sync Width	^2
4	VeT7	VeT6	VeT5	VeT4	VeT3	VeT2	VeT1	VeT0	Vertical Total	^3
5	....	....	....	VeA4	VeA3	VeA2	VeA1	VeA0	Vertical Total Fine Adju	^3
6	VeD7	VeD6	VeD5	VeD4	VeD3	VeD2	VeD1	VeD0	Vertical Displayed	^3
7	VeS7	VeS6	VeS5	VeS4	VeS3	VeS2	VeS1	VeS0	Vertical Sync Position	^2
8	....	....	....	....	....	....	Ilc1	Ilc0	Interlace Mode	^4
9	....	....	....	CTV4	CTV3	CTV2	CTV1	CTV0	Character Total Vertical	^5
10	....	CrM1	CrM0	Css4	Css3	Css2	Css1	Css0	Cursor Mode/ Start Scan	^6
11	....	....	....	Ces4	Ces3	Ces2	Ces1	Ces0	Cursor End Scan	^6
12	Ds15	Ds14	Ds13	Ds12	Ds11	Ds10	Ds09	Ds08	Display Start Adrs (Hi)	^7
13	Ds07	Ds06	Ds05	Ds04	Ds03	Ds02	Ds01	Ds00	Display Start Adrs (Lo)	^7
14	Cp15	Cp14	Cp13	Cp12	Cp11	Cp10	Cp09	Cp08	Cursor Position (Hi)	^7
15	Cp07	Cp06	Cp05	Cp04	Cp03	Cp02	Cp01	Cp00	Cursor Position (Lo)	^7
16	LpV7	LpV6	LpV5	LpV4	LpV3	LpV2	LpV1	LpV0	Light Pen Veritcal	^8
17	LpH7	LpH6	LpH5	LpH4	LpH3	LpH2	LpH1	LpH0	Light Pen Horizontal	^8
18	Ua15	Ua14	Ua13	Ua12	Ua11	Ua10	Ua09	Ua08	Update Address (Hi)	^9
19	Ua07	Ua06	Ua05	Ua04	Ua03	Ua02	Ua01	Ua00	Update Address (Lo)	^9
20	At15	At14	At13	At12	At11	At10	At09	At08	Attribute Start Adrs (Hi)	^7
21	At07	At06	At05	At04	At03	At02	At01	At00	Attribute Start Adrs (Lo)	^7
22	HcP3	HcP2	HcP1	HcP0	IcS3	IcS2	IcS1	IcS0	Hz Chr Pxl Ttl/IChar Spc	^A
23	....	....	....	VcP4	VcP3	VcP2	VcP1	VcP0	Vert. Character Pxl Spc	^5
24	BlkM	RvsS	Vss5	Vss4	Vss3	Vss2	Vss1	Vss0	Block/Rvs Scr/V. Scroll	^9^B^C
25	Text	Atri	Semi	Dble	Hss3	Hss2	Hss1	Hss0	Diff. Mode Sw/H. Scroll	^D,^E
26	Fgd3	Fgd2	Fgd1	Fgd0	Bgd3	Bgd2	Bgd1	Bgd0	ForeGround/BackGround Col	^F
27	Rin7	Rin6	Rin5	Rin4	Rin3	Rin2	Rin1	Rin0	Row/Adrs. Increment	^G
28	CSa2	CSa1	CSa0	RamT	....	....	....	....	Character Set Addrs/Ram	^H,^I
29	....	....	....	UdL4	UdL3	UdL2	UdL1	UdL0	Underline Scan Line	^6
30	WdC7	WdC6	WdC5	WdC4	WdC3	WdC2	WdC1	WdC0	Word Count (-1)	^9
31	Dta7	Dta6	Dta5	Dta4	Dta3	Dta2	Dta1	Dta0	Data	^9
32	BlkF	BlkE	BlkD	BlkC	BlkB	BlkA	Blk9	Blk8	Block Copy Source (hi)	^9
33	Blk7	Blk6	Blk5	Blk4	Blk3	Blk2	Blk1	Blk0	Block Copy Source (lo)	^9
34	DeB7	DeB6	DeB5	DeB4	DeB3	DeB2	DeB1	DeB0	Display Enable Begin	^J
35	DeE7	DeE6	DeE5	DeE4	DeE3	DeE2	DeE1	DeE0	Display Enable End	^J
36	....	....	....	....	Drm3	Drm2	Drm1	Drm0	DRAM Refresh Rate	^K

Register Usage:

^1 : Register #0: Horizontal Total
--- Register #1: Horizontal Displayed
Register #2: Horizontal Sync Pulse

These two register function to define the display width of the screen. Register 0 will contain the number of characters minus 1 between sucessive horizontal sync pulses, the horizontal border and the interval between horizontal sync pulses. The normal value for this is usually set to 126. Register 1 specifies how many of the positions as specified in register 0 can actually be used to display characters. The default value for this is 80. The VDC can take values less than 80 and thus, will only display that many

characters. A useful effect can be a sweep from the right by incrementing the value here from 1 to 80. Register #2 specifies the starting character position at which the vertical sync pulse begins. Thus, it also determines where on the active screen characters appear. A default value of 102, increasing the value moves the screen to the left, decreasing it moves it to the right.

^2 : Register #3: Vertical / Horizontal Sync Position.
---- Register #7: Vertical Sync Position

In Register 3, Bits 0-3 of this register specifies the horizontal sync width and should be equal to 1 + the number of pixels per character. Thus, the value here is normally 1+8 or 9. Bits 4-7 of register 3 specify the vertical sync width and normally contains a value of 4. For interlace sync and video mode, use a value that is twice the number of scan lines desired. Register #7 allows for adjustment of where the vertical sync will be generated allowing shifting of the actual display up and down. Normally, a value of 4, decreasing the value will move the screen down, increasing it will appear to move it upwards.

^3 : Register #4: Vertical Total
---- Register #5: Vertical Total Fine Adjust
Register #6: Vertical Displayed

Register #4 of this register determines the total number of screen rows, including the rows for the active display, and the top and bottom borders in addition to that of the vertical sync width. The value held here is normally a value of 32 for NTSC systems (US Standard) or 39 for PAL(European) systems. Register #5 holds in bits 0-4 a "fine-adjust" where any extra scan lines that are necessary to make up the display can be specified here. The value here is normally a 0 in both the NTSC and PAL initializations by the kernal and bits 5-7 are unused, always returning a binary 1111. Register #6 specifies the total number of the vertical character positions (as set in Register 4) that can be used for actual display of characters. Thus, this register usually holds a value of 25 for a standard 25-row display.

^4 : Register #8: Interlace Mode Control

Register 8 allows control of various display modes the 8563 can generate. Bits 0 and 1 are the only bits used in this register, the rest always reading a binary 1. Bits 0 and 1 are configured as follows:

Binary %00, %10 - NonInterlaced Mode
%01 - Interlaced Sync
%11 - Interlaced Sync and Video

Note that the default value is $00 which is standard, non-interlaced. Interlaced sync draws each horizontal scan line twice but appears to suffer from an annoying jitter due to how it is drawn. Interlaced Sync and Video draws twice as many lines, thus doubling the resolution. However, it also suffers from jitter and that is why most monitors suffer horribly when using programs that support more than 30 rows. Note that for interlaced sync and video, the following registers will need to be changed: #'s: 0,4,6,7,8.

^5 : Register #9: Total Scan Lines Per Character

Bits 0-4 of this register are the only relevant ones, the rest returning a binary 1. Bits 0-4 determine the character height in scan-lines of displayed characters and allow up to scan-line heights of 32 scan lines. The VDC normally sets aside 16 bytes for each character (normally, each byte is equivlent to 1 scan line) so the value here could be increased to 16-1 and a double-height character set could be loaded in. Note, however that values less than 16 will tell the VDC to use a 8,192 byte character set (normal) while specifying values greater than 16 will make it use 32 bytes per character even if some of the

bytes are not used.

```
^6 : Register #10:    Cursor Mode / Start Scan Line
---- Register #11:    Cursor End Scan Line.
     Register #29:    UnderLine Scan Line Control.
```

These registers allow the user to specify the cursor blink mode, as well as the starting and ending scan lines for the cursor (allowing a full solid, an underline, Bits 0-4 of regiseter #10 determines the scan line within each position for the top of the cursor. Normally, this value holds a value of 0 for the block cursor, or a value of 7 for the underline cursor. Bits 5-6 of Register 10 specify the blink rate for the cursor. A value of %00 specifies no blink, ie: a solid cursor. A value of %01 specifies no cursor, a value of %10 specifies a flash rate of 1/16 the screen refresh rate, while a value of %11 specifies a flash rate of 1/32 the screen refresh rate. Note that bit 7 of Register 10 is unused and normally returns a binary 1. Register 11 specifies the bottom scan lines in bits 0-4, the other unused bits returning a binary 1. The value held in these bits usually is 7 for the block and underline cursor modes in the normal 128 editor. Register #29 is used to indicate where the scan line is "set" in the character. The "underline" is only 1 pixel tall and thus, this location just indicates the start and end location in pixels, similair to registers #10 and #11 being the same value. Note that bits 5-7 of this register is unused and normally return a binary 1.

```
^7 : Register #12:    Display Start Address (Hi)
---- Register #13:    Display Start Address (Lo)
     Register #14:    Cursor Position       (Hi)
     Register #15:    Cursor Position       (Lo)
     Register #20:    Attribute Start Addrs (Hi)
     Register #21:    Attribute Start Addrs (Lo)
```

Note first, that all of these registers are grouped in Hi byte, Lo byte order which is usually different from the 6502 convention of low byte, hi byte (ie: in normal 6502 ml, $c000 is stored as $00 $c0, however in the 8563 it would be stored as $c0 $00). Registers 12 and 13 determine, where in VDC memory the 8563 is the start of the screen. Incrementing this value by 80 (the number of characters per line) and with a little additional work can provide a very effecient way of having a screen that "seems" to be larger than just 80x25. The cursor position in registers 14 and 15 reflect the actual character in memory that the cursor currently lies over. If it's not on the display screen, then it is not displayed. Registers 20 and 21 reflect where in the 8563 memory attribute memory is held. Attribute memory refers to the character attributes such as flash, inverse video, color etc that can be set for each character.

```
^8 : Register #16:    Light Pen Vertical
---- Register #17:    Light Pen Horizontal
```

These registers return the light pen position and refer to the actual character positions on screen (ie: values ranging from 1..25 for vertical). The horizontal reading will not corrospond exactly to character positions, but will range from values of 27-29 to 120 depending on the edge of the screen. It's recommended that the horizontal character position is given more tolerance than the vertical light pen position for this reason.

```
^9 : Register #18:    Update Address (Hi)
---- Register #19:    Update Address (Lo)
     Register #24:7   Copy / Fill Bit
     Register #30:    Word Count(-1)
     Register #31:    Data
     Register #32:    Block Copy Src (Hi)
     Register #33:    Block Copy Src (Lo)
```

These registers allow control and manipulation of the 16k or 64k block within the 8563 memory. Registers 18 and 19 point to where in VDC memory the next read or write will take place from. Register 30 specifies the number of bytes

- 1 to copy or fill depending on bit # 7 of register #24. Normally, the 8563 will automatically perform the designated operation (of what bit 7 of register #24 says) when register #31 (the data byte) is written to. Registers 18 and 19 automatically update upon read or write, so that is why register #30 specifies a value 1 less than what is actually needed. Register #31, as already mentioned is the byte to write for register #30 times (or copy from Register#32 / #33). If register #24, bit 7 is specified as a binary 1 then the memory is copied from the address in VDC memory pointed to by registers #32 and #33.

^A : Register #22: Character Horizontal Size Control

Bits 0-3 of this register determines how many horizontal pixels are used for each displayed character. Values greater than 8 here result in apparent gaps in the display. Inter-character spacing can be achieved by setting this value greater than that of bits 4-7. Bits 4-7 determine the width of each character position in pixels. Thus, while bits 0-3 allocate n-pixels, bits 4-7 specify how many of those pixels are used for character display.

^B : Register #24:5 Reverse Screen Bit
---- Register #24:6 Blink Rate for Characters.

Bit #6 specifies for the VDC for all pixels normally unset on the VDC screen to be set, and all set pixels to be unset. Bit #5 specifies the blink rate for all characters with the blink attribute. Setting this to a binary 1 specifies a blink rate of 1/32 the refresh rate, while a binary 0 is equivlant to a blink rate 1/16th of the refresh rate.

^C : Register #24:0-4 Vertical Smooth Scroll

The 8563 provides for a smooth scroll, allowing bits 0-4 to function as an indicator of the number of bits to scroll the screen vertically upward.

^D : Register #25:7 Text or Graphics Mode Indicator Bit
---- Register #25:6 Monochrome Mode Bit
     Register #25:5 Semi-Graphics Mode
     Register #25:4 Double-Pixel Mode

The 8563 allows the implementation of a graphics mode, in where all of the 16k of the screen may be bit-mapped sequentially resulting in a resolution of 640x200 (see Craig Bruce's 8563 Line-Plotting routine in the first issue for a more detailed explanation of this feature). Setting this bit to 1 specifies graphics mode, binary 0 indicates text mode. Bit 6 indicates to the 8563 where to obtain its color information etc, about the characters. Bit 6 when it is a binary 0 results in the 8563 taking it's color information from bits 4-7 of register 26. When this bit is a binary 1, the attribute memory is used to obtain color, flash, reverse information. Also note than when this bit is a binary 1 that only the first of the two character sets is available. Bit #5 indicates a semi-graphics mode that allows the rightmost pixel of any characters to be repeated through-out the intercharacter spacing gap. Activating it on the normal display will result in what appears to be a "digital" character font. The 8563 with bit #4 allows a pixel-double feature which results in all displayed horizontal pixels having twice their usual size. Thus, a 40 column screen is easily obtainable although the values in registers #00-#02 must be halved.

^E : Register #25: Horizontal Smooth Control

This register is analogous to register #24 Vertical Smooth Control and functions similairly. Increasing this bits moves the screen one pixel to the right, while decreasing them moves the screen one pixel to the left.

^F : Register #26: ForeGround / BackGround Color Register

This register, in bits 0-3 specifies the background color of the display while bits 4-7 specify the foreground character colors when attributes are disabled (via bit 6 of register #25). Note, these are not the usual C= colors but are instead organized as follows:

Bit Value	Decimal Value	Color
%0000	0 / $00	Black
%0001	1 / $01	Dark Gray
%0010	2 / $02	Dark Blue
%0011	3 / $03	Light Blue
%0100	4 / $04	Dark Green
%0101	5 / $05	Light Green
%0110	6 / $06	Dark Cyan
%0111	7 / $07	Light Cyan
%1000	8 / $08	Dark Red
%1001	9 / $09	Light Red
%1010	10 / $0A	Dark Purple
%1011	11 / $0B	Light Purple
%1100	12 / $0C	Dark Yellow
%1101	13 / $0D	Light Yellow
%1110	14 / $0E	Light Gray (Dark White)
%1111	15 / $0F	White

```
+------------------------------+
|  Note: Bit 0 = Intensity     |
|        Bit 1 = Blue          |
| RGBI   Bit 2 = Green         |
|        Bit 3 = Red           |
|                              |
+------------------------------+
```

^G : Register #27: Row Address Display Increment

This register specifies the number of bytes to skip, when displaying characters on the 8563 screen. Normally, this byte holds a value of $00 indicating no bytes to skip; however typically programs that "scroll" the screen do so by setting this to 80 or 160 allowing the program to then alter the Screen Start (Registers #12 and #13) and appear to "scroll". Note the normal C= 128 Kernal Screen Editor does not support this function.

^H : Register #28:7-5 Character Set Address

These bits indicate the address of screen memory * 8k. Thus the values in these bits may be multiplied by 8192 to obtain the starting character set position (normall these bits hold a value of $01 indicating the character set begins at 8192). Note that the character set is not in ROM, but is usually copied to 8192 when the computer is first turned on and the 8563 is initialized. (Examine the INIT80 routine at $CE0C in bank 15).

^I : Register #28:4 Ram Chip Type

This bit specifies whether 16k or 64k of RAM has been installed. Note, however that this value may not reflect future upgrades from 16k to 64k. It is best, if a program is dependant on 64k to write to an address > 16k and see if it is mirrored at any other location in another section of memory. This bit has a binary value of 0 if 16k or 1 if 64k RAM.

^J : Register #34: Display Enable Begin
---- Register #35: Display Enable End

The 8563 can extend it's horizontal blanking interval to blank a portion of the displayed screen. The value in register #34 determines the rightmost blanked column, and register #35 determines the leftmost blanked column. Note that a value of 6 usually corresponds to the leftmost column of the screen, while a value of 85 corresponds to the rightmost column. This feature is useful for "inside-out" wraps in which both the right and left margin can close-in on text, the text can be cleared, these values reset etc...

^K : Register #36: Refresh Cycles per Scan Line

  This register in bits 0-3 allows the user (if he had any reason) to specify the number of refresh cycles for memory for the ram. Setting this value too low may cause the RAM to not remember all the information. Changing this value gives some advantage, in terms of display speed increases but is not advised. The value normally held here is $05, for five refresh cycles per scan line.

8563 Memory Organization

  Normally, the extra memory of the C=128's equipped with 64k goes unused (48k worth) unless programs like Basic-8 etc, take advantage of it. There are various mod files describing the upgrade from 16k to 64k and it is _strongly_ advised (although the author has not yet done so) and be aware that ***OPENING YOUR COMPUTER JUST TO LOOK, YOU MAY MESS IT UP*** and it is _strongly_ advised that you contact a person experienced with electronics to perform the upgrade for you. Note also that some mail order companies are offering an "up-grade board" which plugs into the 8563 slot and does not involve desoldering the RAM chips.

  Now, the 8563 uses the 16k of memory (it ignores the extra 48k of memory when it's got 64k, thus the following applies also to the 8563's equipped with 64k of memory) and normally, has the following memory map:

```
  $0000 - $07ff - Screen Memory
  $0800 - $0fff - Attribute Memory
  $1000 - $1fff - Unused
  $2000 - $2fff - UpperCase / Graphic Character Set   (Char Set #1)
  $3000 - $3fff - LowerCase / UpperCase Character Set (Char Set #2)
```

Writing to 8563 Registers

  Now how do we write to these registers we've learned so much about? There's several ways depending on how lazy you are. The pure-ml version:

WRITING TO A REGISTER:

```
   writereg = * ; this routine writes .a to register # .x, Asssumes I/O block in
         stx $d600
       - ldx $d600
         bpl -
         sta $d601
         rts
```

  also, in bank 15 there is a similair routine at $cdcc. Calling it at $cdca loads .x with a value of 31 indicating the data register which is often useful.

  From basic, just use a SYS 52684, value, register#

 READING FROM A REGISTER:

```
   readreg = * ; this routine returns the contents of register # .x in .a
               ; Assumes I/O block switched in
        stx $d600
      - ldx $d600
        bpl -
        lda $d601
```

   or use the routine in bank 15 at $cdda. From basic, a SYS 52698,,register# and then a RREG A returns the value in variable A.

Further 8563 Notes

Many C=128 owners are still using their monitors they had when they had their C=64's and are able to use the 80 column screen through a "converter-cable" (basically taking pin 7 of the RGBI port and feeding it as raw video). There is also a text file out explaining how to take the R,G,B,I pins on that port to display shades of gray on a monochrome monitor (basically tying resistors with diodes across each color pin and then joining them). There is relief!! :-)

The 8563 is a chip full of cabibilities waiting to be found and developed. I'd be interested in seeing any code / techniques that readers of this net-mag have found. Given that enough are submitted, a possible listing of some of the better tricks and techniques might be possible in the future.

FILE SPLITTER - Mark Lawrence 9152427d@levels.unisa.edu.au

This program stemmed from the inability of XLINK to transfer CS-DOS from my pc to my 128. XLINK transfers about 43K (I think), whereas CS-DOS was about 48K.

Rather than do the whole thing at once, why not cut the job up into more sizeable pieces, transfer the program piece by piece, and then reassemble the pieces at the other end?

And so eventuated the birth of SPLIT :-)

SPLIT, written entirely in Turbo Pascal, allows you to split DOS files into smaller pieces - you can either tell it a size to split the files into, or tell it a number of files to create. You then give SPLIT the base filename for the new files WITH NO EXTENSION - SPLIT will give the new files their own extensions, and SPLIT will then create these files to your liking.

Just transfer the following program to Turbo, compile it, and away you go!!!

Hopefully, the program is commented enough to give you a fair idea of what's going on - although it isn't at all complicated to understand.

At some points I have comments that seem the least important - END { CASE } - they are to help me when I program... I find it easy to lose track of which END is for what, stuff up my indentation, lost bits and pieces, delete the wrong parts, etc, etc.

I found it helped me, so it may help others.

If you need any further explanation, just let me know :-)

Another interesting thing I discovered about XLINK. It doesn't transfer the files to the correct size. I think (haven't had time to sit down and check it out yet) it transfers to the nearest 256, 512 or 1024 byte boundary. If your file doesn't reach the boundary, it will pad the rest out with zeroes I think. So, when you go to reassemble the file, it's got all this garbage in places where it shouldn't be, and the thing won't work.

So, when SPLITting a file, specify the size to a multiple of one of these boundaries.

Then, using a m/c monitor, load all the parts in together.

I'll try to set aside a little time in the not too distant future to write a m/c program to join the parts for you, since it can get confusing reassembling the parts by hand, and the built in dos copy that commodore so kindly graced us with is so darned fast <cough> <cough> :-)

[Ed. Note: While the dos copy command is slow.... for those of you who are
 impatient try using somethine like the following to join files togather making
 sure that there's enough space on the disk:
     open15,8,15,"c0:name=name1,name2...": close15]

So, good luck and enjoy!

```
Program Split (input,output);

Uses Dos;
  { uses specific file handling routines }
Var
   InFile,Outfile                : File of Byte;
   Count,Number,Size,NewSize,
   Last,Counter                  : Longint;
   InFileName,Newfile,OutFileName: String;
   S,P                           : PathStr;
   D                             : DirStr;
   N                             : NameStr;
   E                             : ExtStr;
   SplitType,Check               : Char;
   Data                          : Byte;
   Extension                     : String[3];

Begin
   For count := 1 to 25 do
      Writeln;
      { Dumb way to clear the screen :-)  }

   Writeln ('*********************************************************');
   Writeln ('*  FILE SPLITTING UTILITY V0.01 (C) 1992 MARK LAWRENCE  *');
   Writeln ('*               (-: MADE IN AUSTRALIA :-)               *');
   Writeln ('*********************************************************');
   Writeln;
   Write   ('Enter Filename (including drive and path)    ');
   ReadLn  (InFileName);
   Writeln;

   For count := 1 to length (InFileName) do
      InFileName[count] := UpCase ( InFileName[count] );
      { change filename to all uppercase characters }

   FSplit(InFileName,d,n,e);
      { split filename into it's respective parts:
              d - Directory
              n - Name
              e - Extension }

   S := FSearch(InFileName,GetEnv(D));
      { search for file FILENAME in directory D }

   if S = '' then
      writeln ('*ERROR*     File "',InFileName,'" not found.')
      { S equals '' (nothing) if FILENAME doesn't exist }

   Else
   Begin
      Assign  (Infile,InFileName);
      Reset   (Infile);
      { Open the Input File }

      Size := FileSize (InFile);
```

```
{ Get file size }

Writeln ('FileName:     ',InFileName);
Writeln ('FileSize:     ',Size,' Bytes.');
Writeln;
{ Show file info }

Writeln ('In which way would you like the file split?');
Writeln ('     (a)  Number of Bytes.');
Writeln ('     (b)  Number of Files.');
Repeat
   Write ('Enter your selection    ');
   Readln (SplitType);
   SplitType := UpCase(SplitType);
Until (SplitType >= 'A') and (SplitType <= 'B');
{ let user choose which way to split file }

Writeln;
Case SplitType of
   'A': Begin
      { split by number of bytes }
      Write ('Enter byte size of new files    ');
      Readln (NewSize);
      Writeln;
      If (NewSize > Size) then
         Writeln ('Hey - Even I can''t do that!!!')
      Else
      begin
         Number  := Size div NewSize;
         Last    := Size - Number * NewSize;
         Number  := Number + 1;
         Write ('Enter Base Filename (including drive and path)    ');
         Readln (NewFile);
         Writeln;
         Writeln ('Creating ',Number,' new files...');
      End;
   End; { A }

   'B': Begin
      { Split by file size }
      Write ('Enter number of new files: ');
      Readln(Number);
      Writeln;
      NewSize := Size div Number + 1;
      Last    := Size - (Number - 1) * Newsize;
      Number  := Number;
      Write ('Enter Base Filename (including drive and path)    ');
      Readln (NewFile);
      Writeln;
      Writeln ('Creating ',Number,' new files...');
   End; { B }
End;  { Case }

   Writeln;

   For Count := 1 to Number do
   { NUMBER new files will be created }

   Begin
      If Count = Number then
         NewSize := Last;
      { More often than not, the files won't divide evenly from the
        original.  So, the last file will be smaller than the rest.
        Because of this, I previously calculated the size of the final
        file, and here check if we're up to the last part yet - and if
```

```
              we are, I set the size of the last file accordingly }

            Str(Count,Extension);
            { Make EXTENSION a string representation of COUNT, to be added to
              the OutFileName to make things a tad easier }

            OutFileName := Concat(NewFile,'.',Copy('00',1,3-Length(Extension)),E
            { Create filename based on which part we're up to }

            Assign  (OutFile,OutFileName);
            Rewrite (Outfile);
            { Open each Output File }

            Write   ('Creating ',OutFileName,'... ');

            For Counter := 1 to NewSize do
            { Write to each Output File }

            Begin
               Read  (Infile,Data);
               Write (OutFile,Data);
               { Transfer data from input file to output file }
            End;

            Close   (Outfile);
            { Close each Output File }
            Writeln ('Done!');

         End;

      Writeln;
      Writeln ('Job Completed :-)');

   end;

   For Counter := 1 to 3 do
      Writeln;
   { Make a bit of space when finished :-) }
end.
```

==========

BANKING ON GEOS

by Robert A. Knop Jr.

I. Introduction

GEOS was originally written for the Commodore 64. When Berkeley Softworks came out with GEOS128 (and, for a time, it wasn't clear that they would; then, it looked like they would release a GEOS128 that wouldn't support the 80 column screen; finally, the release of GEOS128 did turn out to be a full 128 program), it was largely compatible with GEOS64. Applications could share documents (a geoPaint file is a geoPaint file), and even many GEOS64 appliations run on the 128 in 40 columns. This heritage is also evident to the GEOS programmer.

As we all know, the C-128 has two 64K RAM banks; the C-64 only has one 64K RAM "bank." Thus, of course, all of GEOS64 goes into that one 64K RAM space. This includes the Kernal as well as the space available to applications. Once the Kernal, graphics screens, and so forth, have claimed their RAM, the GEOS programmer is left with 23.75K of memory from $0400 to $5fff.

To a cursory "glance," the GEOS128 programming environment looks very much like the GEOS64 programming enviroment. You still have $0400 to $5fff available for applications; graphics screens and variables are in the same

place; the Kernal jump table is the same (with some 128 specific additions). What happened to the other 64K that the 128 has available?

As it turns out, the core of GEOS128- including the application program space, the 40 column foreground and background screen, and the Kernal jump table- are all in the 128's RAM block 1, what GEOS calls FrontRAM. To us 128 programmers used to RAM block 0 being the "main" RAM block, this may sound odd. However, it actually makes sense. First of all, since GEOS is an operating system in and of itself, and applications almost never need to call the C128's Kernal routines, the application no longer needs access to Bank 15. Second, it allows GEOS128 to keep much of its memory map the same as GEOS64; it can use the memory range from $200-$3ff in RAM 1 without worrying about disturbing key system routins like STAFAR which are in the same memory range in RAM 0.

II. Yeah, Yeah, But What Happened to RAM 0 Anyway?

It's still there. Some of RAM 0 is used by GEOS128 to improve the system performance and to take advantage of the 128's unique features. (For instance, the code for the "software sprites" seen on the 128's 80 column screen is found beneath $2000 in RAM 0.) Fortunately, some space does remain available for an application to use. In RAM 0, the 32K memory space between $2000 and $9fff is not normally used by GEOS, and is ALMOST available for application use [1].

Why do I say "almost"? The problem is desk accessories. When GEOS 64 loads a desk accessory (DA), it must load it into the same application space as the application loading the DA. The memory that the DA will used is first saved to disk in a swap file. Under GEOS128, the routine LdDeskAcc, instead of saving a swap file to disk, copies the memory to be overwritten by the DA to RAM0 between $2000 and $9fff. So, if your application uses DA's (and it is highly recommended that major applications support DA's), you have to be careful using the space between $2000 and $9fff. You can use it as temporary swap space within routines- but you cannot assume that it will remain intact whenever your routine returns to the GEOS MainLoop with your application in a state that will allow the loading of DA's.

Nowadays, RAM 0 is not the be-all and end-all. GEOS128 was written for the C=128, not the C=256. Consequently, if you have expanded your 128 to 256K or 512K as described in the articles by Richard Curcio in Twin Cities 128 Issues #30 and #31 [2,3], you have free use of RAM 2-3 (256K) or RAM 2-7 (512K). (Note that you should not touch RAM 4-7 on a 512K 128 if you want to be compatible with task switching as described in TC128 #31. Also, although GEOS right now does not run in the 2nd 256K, applications should not assume they are in the 1st 256K, and thus should be careful with the 512K mode bits (4-5) in the MMU Ram Configuration Register (RCR), $d506.) While the number of people with 256K and 512K 128's is now small, you can be sure that it will increase when the promised ZIP accelerator board for the 128 comes out; the current specs for the ZIP board include provisions for memory expansion on the board.

RAM 2-3 provide almost another complete 128K available for your application to use. So how do you go about accessing this?

III. Storing Data In Other RAM Blocks

The most obvious use for RAM blocks other than FrontRAM (which is the only block where GEOS Kernal routines are available) is as data storage. For instance, one could visualize a geoPaint previewing utility which loads and decompacts an entire geoPaint document at once to RAM 2. (The full decompacted geoPaint document would reqire 56.25K.) One could then quickly scroll through the document by just copying the relevant portions of the bitmap from RAM 2 to the foreground screen. Or, if one were really bold, one could just redirect the VIC screen memory to the relevant range in RAM2 using

the proper MMU and VIC registers. (This would actully require use of both RAM 2 and 3, since VIC screen locations are quantized to 8K; you lose the use of the highest 8K, since you don't want to overwrite the MMU registers at $ff00-$ff05; additional practical considerations make use of the lowest 8K difficult.)

GEOS128 provides a few routines for easily moving data between FrontRAM and what it calls BackRAM (but we know it just means RAM 0). Happily, these routines work quite admirably with RAM 2 and 3. (To access RAM 4-7, fiddle bits 4 and 5 of the MMU RCR to make the desired RAM blocks appear to the system as virtual RAM 2 and RAM 3, then call these routines.) The core routine is DoBOp, which is summarized below [4]:

```
*********************************************************************
DoBOp=$c2ec:  Copy/verify memory between RAM blocks on the C-128.

Pass:
   r0   : ADDR1 - address of first ("source") memory range
   r1   : ADDR2 - address of second ("destination") memory range
   r2   : COUNT - number of bytes to operate on
   r3L  : A1BANK - bank of ADDR1 (e.g. 1=FrontRAM, 0=BackRAM)
   r3H  : A2BANK - bank of ADDR2
   y    : MODE - operation to perform


Returns: r0-r3 unchanged

  when verifying: x=$00 if two ranges match, x=$ff if they don't match


Destroys: a,x,y


The operation mode is passed in y as follows:

        bit0  bit1   Description
        ----  ----   -----------
         0     0      Move from memory at ADDR1 to memory at ADDR2
         0     1      Move from memory at ADDR2 to memory at ADDR1
         1     0      Swap memory at ADDR1 and ADDR2
         1     1      Verify (compare) memory at ADDR1 and ADDR2
*********************************************************************
```

(r0, r1, etc. are all the standard BSW symbols defined in the Official GEOS Programmer's Reference Guide [5], and that come in the file geosSym with geoProgrammer.)

There are a number of additional routines which are also provided for programmer convenience which automatically set the MODE in the y register for you. In all of these routines, r0-r3 have the same meaning as they do in DoBOp.

Routine	Address	MODE	Description
MoveBData	$c2e3	00	Copy data at ADDR1 to ADDR2
SwapBData	$c2e6	10	Swap data between ADDR1 and ADDR2
VerifyBData	$c2e9	11	Compare data at ADDR1 and ADDR2

I have written a short demonstration program which shows the use of MoveBData and VerifyBData. The full source to this program, BMover, is available through anonymous ftp at tybalt.caltech.edu (in the /pub/rknop/hacking.mag directory) as well as elsewhere. If you can't find it, contact me (addresses are below). The source is geoProgrammer code, in geoWrite 2.1 format. All of the files you need (except geosSym and geosMac, which come with geoProgrammer) are in the bmover.sfx archive.

The first function of BMover repeatedly copies a single block on RAM 1 to successive parts in memory in any other specified bank. The destination bank, destination addresses, size of the block to move, and number of times to copy it are all set in constants found at the beginning of the source file BMovAsm. Once the moves (which use MoveBData) have all been performed, BMover uses VerifyBData to make sure that all of the blocks were copied succesfully.

For informational purposes, BMover reports the amount of time (in tenths of seconds) it took to perform all of the moves. (For this, I use the CIA #1 TOD clock, saving its value at the beginning and end of the move, and subtracting to get the difference.) I ran a trial where I copied an 8K block of memory to RAM 2 7 times (thus filling 56K of RAM 2). These moves together took 1 second at 2 MHz, and 2.2 seconds at 1 Mhz. 56K/second may be no DMA, but it's faster than a burst load!

IV. Executing Routines In Other Banks

So, you've written an object oriented drawing program that stores its list of objects (32 byte records) in RAM 2. Or, you have a database that has records in RAM 0. You want to delete one record at the beginning of the list, which means moving all of the subsequent records down over the memory freed up by the deletion. There are a few things you can do. One, you can use Craig Bruce's dynamic memory allocation routines (highly recommended). Two, you can repeated do MoveBData to move memory from RAM 2 (or 0) to a buffer in FrontRAM and back. Or, you can write a short mover routine in the RAM bank where all the moving is going to happen.

This is just an example. One can visualize other reasons for calling routines in other RAM banks (what I call "extrabankal routines"). There exist no GEOS Kernal routines for calling extrabankal routines. Additionally, since your main application memory is in RAM 1, you are inable to use the 128 Kernal's JSRFAR (which returns you to Bank 15). So, we are left with implementing our own JSRFAR.

GEOS128 normally operates with NO common memory enabled. Thanks to one of the less well-known features of the MMU, there is no need to enable common memory. The MMU zero page registers ($d507 and $d508) allow you to locate the zero page that the processor sees anywhere in RAM 0 or RAM 1. What this means is, no matter what your memory configuration is, the processor sees zero page in the RAM block specified in $d508. (Unless you have common memory enable, in which case it is not a good idea to put ZP in RAM blocks other than RAM 0 [6,7].) So, zero page is effectively common memory!

This provides for the possiblity of copying to zero page a short "switchboard" routine, basically a reimplementation of JSRFAR, which configures the system for the destination bank, jsr's to a routine, reconfigures the system for the calling bank, and rts's.

I also demonstrate this technique in BMover. The second function of BMover first uses MoveBData to copy a routine to $2000 in DESTBANK (which is set right now in the source code to RAM 0). It then copies the routine ZPJSR to $02, which stores DESTCFG in $ff00 and jsr's to $2000. The routine at $2000 moves some data around in DESTBANK. Once ZPJSR has returned the program flow to FrontRAM, BMover calls VerifyBData to make sure everything worked.

While messing around in different banks, to be safe I dissable IRQ interrupts. On a related note, geoDebugger 2.0 seems to have problems with programs messing around with different banks. It is not surprising that the BackRAM debugger (which locates itself in RAM 0) would have trouble with programs that tried to use RAM 0, but it also has trouble with programs that try to use RAM 2 and 3. This is true even when one uses the system routine MoveBData. (I found that I was sometimes able to make it past a call to MoveBData while in the debugger, but that more often the system would hang. This is all probably

an interrupt-related issue.)

If one is to be really classy, one doesn't actually have to copy the ZPJSR routine to zero page. One could assemble the application such that ZPJSR fell to a known offset from a page boundry; then, use the MMU to point zero page to the page containg ZPJSR. Unfortunately, this technique did not work on my 512K expanded 128. The one incompatility I have found is that with the 512K modification enabled (I do have a switch to disable it, don't worry), the MMU fails to correctly see zero page in RAM 1 when requested to. Richard Curcio experimented with it, and it seems that when you try to relocate zero page to a page in RAM 1, it is actually seen in RAM 3. It is not yet clear whether this is a problem with the 256K/512K modification, or if the MMU in a stock 128 just relocates ZP to RAM 3 figuring that RAM 3 = RAM 1 (which is true on a stock 128, but not on a 256K expanded 128!)

Anyone who wants to get ahold of the BMover source, or who has other questions/comments/flames can contact me, Robert Knop, at the following addresses:

InterNet: rknop@tybalt.caltech.edu
GEnie: R.KNOP1
U.S. Mail: Robert Knop
123 S. Chester #3
Pasadena, CA 91106

V. References

[1] William Coleman, 1989: "Inside GEOS 128" _The_Transactor_ 9(4), p. 29.

[2] Richard Curcio, 1991: "Expanding the 128 Part One: 256K" _Twin_Cities_128_ #30, p. 7.

[3] Richard Curcio, 1992: "Expanding the 128 Part Two: 4 Mode 512K" _Twin_Cities_128_ #31, p. 5.

[4] Berkeley Softworks, 1988: _The_Hitchhiker's_Guide_To_Geos_.

[5] Michael Farr, 1987: _The_Offical_GEOS_Programmer's_Reference_Guide_. Bantam Books, New York/Toronto.

[6] Larry Greenly et. al, 1986: _Commodore_128_Programmer's_Reference_Guide_. Bantam Books, New York/Toronto.

[7] Ottis R. Cowper, 1986: _Mapping_the_Commodore_128_. Compute! Publications, Greensboro, NC.

==========

DYNAMIC MEMORY ALLOCATION FOR THE 128: Breaking the 64K Barrier

by Craig Bruce (csbruce@ccnga.uwaterloo.ca)

Although this article would be best described as extremely technical, I think that it has something for everyone. It could also be described as being extremely long.

Below I have written a program that will read in the lines of a file, sort them, and write then back out to another file. Because of the nature of the problem, the each line of the entire file must reside in the memory of the computer. I implement and use dynamic memory allocation such that the file to be sorted can be larger than 64K, and I use a dynamic data structure such that the memory is used very efficiently. The memory routines were extracted from a text editor called "Zed-128" which also breaks the 64K barrier and can edit

some humongous files (and very efficiently too). Although implemented for the C-128, the dynamic memory scheme could also be fairly easily (ie. in a single lifetime) ported to the C-64.

---

1. INTRODUCTION

How many of us are sick and tired of the "64K limit" that a lot of programs for the 128 and 64 seem to have? Many terminal programs, text editors, and even file copiers seem to be afflicted with this problem. Another problem is that programs often reserve large sections of memory for specific purposes (such as the kill buffer of a text editor) and cannot reconfigure themselves (very easily) for different demands. Still another problem is that many programs do not make use of a Ram Expansion Unit (if you are fortunate enough to have one) to store your volumnous user data.

The way to overcome the limitations of the 64K architecture of the C128 and C64 is to use dynamically allocated memory. What this means is that initially, all of the memory of the computer is free and when a user program requires some memory to store user data, it calls a special subroutine that allocates a given number of bytes of memory to the program to store the user data. And when the program is finished using that chunk of memory, it calls a special subroutine to free the memory chunk and make it available for future allocation requests.

One complication of this memory usage scheme is that a program has to keep track of which chunks of memory it uses for what. This is where dynamic data structures come in. The most important concept here is a pointer. A pointer is simply a variable that stores the address of some data structure (ie. some chunk of memory). If we know the address of a data structure, then we can read it and modify it.

To overcome the problem of not knowing how many records will need to be stored, records are often stored in lists, where every record contains a pointer to the next record in the list, except for the last one, which contains a special value that could not be mistaken for an ordinary pointer (it is called the Null (or Nil for you Pascalers) pointer). Thus, if we know the address of the first record (by using a "head pointer"), then we have sequential access to all of the records in the list. If we want to add or delete records from the list, then we must modify the other pointers such that the consistency of the list is maintained. Organizations other than simple lists are also possible.

The implementation here is able to allocate RAM0 memory for storing user data records, as well as RAM1 memory and even REU memory. As long as the application program keeps track of the pointers to its records, large volumes of user data can be stored since it will be distributed among all of the memory that is available from both the internal memory banks and the external memory banks, thus breaking the 64K barrier.

---

2. FOR THE NOVICE HACKER

You get a sorting utility program. This program implements the insertion sort algorithm, so don't expect to break any speed records. Also, the way that dynamic memory is implemented here is more suited for large data structures that will only be accessed slowly and infrequently (such as the current document in a text editor); however, I wanted to come up with a useful utility and I have never heard of a general file sorter for the 128 or 64. The insertion sort does, however, lend itself well to being used with dynamic data structures in general, since you don't actually have to move anything; you just change a couple of pointers in order to insert a line (record) between two other lines. Also, it turns out the the insertion sort is quite efficient

if your input file is already mostly or partially sorted.

The sort utility itself is completely machine language but assumes that the input and output files are already opened, so a BASIC driver program is required to set things up to and allow the user to easily change the sorting parameters. Such a program is listed here:

```
 1 i$="inputfile.txt" : id=8 : sf=1
 2 o$="outputfile.txt" : od=8
 3 :
 100 print"loading sort.bin..."
 110 bank 15
 120 bload"sort.bin",u(id)
 130 print"scratching old file..."
 140 scratch(o$),u(od)
 150 print"sorting..."
 160 open1,id,2,"0:"+i$
 170 open2,od,3,"0:"+o$+",s,w"
 180 sys dec("1300"),sf
 190 close2
 200 close1
 210 print"finished!"
```

Lines 1 and 2 set up the sorting parameters: the input and output filenames, the input and output file device numbers, and the sorting field position. Change the "sf" value to the position of the first character of the key field. The first position on the line is 1 (not 0). (This corresponds to what Zed uses for columns). Starting from that position, the rest of the line is used for the comparison that determines the order of the lines. If a line is encountered that is shorter than the position of the sorting field, the key value is taken to be the Null String (which comes before any other string).

The program continues to load in the machine language (which fits into the $1300 slot) and scratch the output file if it already exists. Then the files are opened, machine language is called, and the files are closed and the program exits. While reading the file, the program will split any lines that are longer than 242 characters and treat them as multiple lines.

For testing the sort utility, I used a file that contains 1058 lines of the following form:

```
ROXETTE                     MUST HAVE BEEN LOVE                        A01-1-01
ADAMS, BRYAN                SUMMER OF '69                              A05-1-10
JOEL, BILLY                 PRESSURE                                   M11-1-07
EAGLES                      NEW KID IN TOWN                            R06-2-04
ELECTRIC LIGHT ORCHESTRA    CALLING AMERICA                            R11-1-05
COCKBURN, BRUCE             WONDERING WHERE THE LIONS ARE              R14-1-03
```

As you may guess, it is a tape library. The file is 83K in length. I sorted it on both my 1581 (with JiffyDOS) and my RamLink and then I sorted again the file that I sorted in the first place. The resulting execution times are as follows:

WITH EXPANSION MEMORY		WITHOUT EXPANSION MEMORY	
Ramlink regular	= 110 seconds	Ramlink regular	= 376 seconds
Ramlink sorted	= 20 seconds	Ramlink sorted	= 24 seconds
1581 regular	= 120 seconds	1581 regular	= 397 seconds
1581 sorted	= 33 seconds	1581 sorted	= 55 seconds

You'll note that having expansion memory makes sort operate faster. This is because the REU Controller can transfer data around faster than the CPU can. The effect is even more pronounced when using records longer than 78-character lines of text. This is why it is sensible to use expansion memory for general data storage and accessing. The reason why the execution times are so long is

that approximately 1058*529/2 = 280,000 "far memory" line-length fetches have to take place, along with that number of "zpload"s and string comparisons, and 1058 "malloc"s and "free"s. Also, we all know that the Commodore file reading and writing mechanisms are not severely swift.

---

3. FOR THE INTERMEDIATE HACKER

You get a dynamic memory allocation and usage package that can be incorporated into your own programs.

3.1. MEMORY PACKAGE CALLS

The package includes eight system calls:

```
 startup ()
 shutdown()
 zpload  ( [zp1]=FarPointer, .X=ZpAddr, .Y=Length )
 zpstore ( [zp1]=FarPointer, .X=ZpAddr, .Y=Length )
 fetch   ( [zp1]=FarPointer, (zw1)=Ram0pointer, .AY=Length )
 stash   ( [zp1]=FarPointer, (zw1)=Ram0pointer, .AY=Length )
 malloc  ( .AY=Length ) : [zp1]=FarPointer, .CS=error
 free    ( [zp1]=FarPointer, .AY=Length ) : .CS=error
```

The "(...)" means input parameters and ":" preceeds output parameters. ".X" and ".Y" refer to the processor registers and ".AY" means the 16-bit value with the .A register holding the low byte and the .Y register holding the high byte. With "(zw1)" I am refering to the indirect pointer in the zero page locations "zw1" (low byte) and "zw1+1" (high byte) ("zw" means zero page word and it is assigned to locations $FE to $FF) and with "[zp1]" I am refering to the three byte pointer value in locations "zp1" (address low byte), "zp1+1" (address high byte), and "zp1+2" (bank number byte) ("zp" means zero page pointer and it is assigned to addresses $FA to $FC). This three-byte pointer is refered to as a "Far Pointer". The ".CS=error" means that if the routine returns with the carry flag set, an error has occured. The only possible error return in this package is if malloc cannot find enough contiguous free memory to satisfy your request.

You do not actually have to know what the bank numbers mean since they are generated and used by the package as an opaque data type (parlez-vous Modula-2?), but here is what they mean anyway. A value of $3F means internal bank RAM0 and a value of $7F means internal bank RAM1. This works out conveniently in the implementation, since these are the MMU configuration register values for those two banks. A value from $80 to $FE refers to an expansion (REU) memory bank. $80 means expansion bank0, $81 bank1, etc. This means that the package can support up to 8 Megs (minus 64K) of expansion memory (and it does). These values are convenient to use since after loading the bank number into a register, the Negative flag of the processor will be set (useful if handling expansion memory is a special case), and this value can be put directly into the REU Controller's bank register. I don't think you have to worry about having the high bit be a "1" since it is done consistently and I have never heard of an REU larger than 2 Megs. A bank value of $FF is used to represent the Null pointer.

The "startup" routine installs the common code, determines the size of your REU, and initializes the dynamic memory allocation mechanism. In order for the package to access internal memory bank RAM1, it has to call a routine that is in memory below address $0400. Since the package starts at $1300, it has to copy a few "common code" subroutines into low memory such that it can call them later. The common code is installed at address $0200, the BASIC input buffer. Don't overwrite this area while the package is in use. The "sniff" routine is called to determine the number of banks that your REU has. Zero banks means that you have no REU. While sniffing, the package overwrites the first four bytes of every existing expansion bank (unless you limit the number

of expansion banks that the package is allowed to use). To initialize the dynamic memory allocation, the "free" routine is called for RAM0, RAM1, and each expansion bank. RAM0 from $4000 to the top of BASIC memory ($FEFF) is freed, RAM1 from $0400 to $FEFF is freed, and all expansion banks are freed between addresses $0000 to $FFF7. Thus, if you have no expansion memory, you get about 110K free and if you have a 512K expander, you get about 620K free.

The "shutdown" routine doesn't actually have very much to do. Basically, it just zeros out the common code. I did this so if you called the sort routine from BASIC direct input mode, you would not get a "syntax error" from BASIC trying to interpret the garbage left behind. Now, when BASIC encounters a zero, it stops interpreting.

The "zpload" routine will load the given number of bytes into zero page starting at the given zero page address, from any far pointer address. It doesn't matter whether the far address is in internal or expansion memory; the operation is the same. This is the level of software that makes accessing the different types of memory transparent to the user. To load from RAM0, true RAM0 is switched into context (did I mention that the package is meant to execute with MMU configuration $0E in context - this configuration gives RAM0 from $0000 to $BFFF, the kernel ROM from $C000 to $FFFF and the I/O space on top of the kernel ROM from $D000 to $DFFF - I call this the SYS or SYS0 bank) and the transfer is done with a loop. For a zpload from RAM1, a common code routine is called that switches RAM1 into context, copies in a loop, and then switches back to SYS0. For an expansion memory pointer, the REU Controller registers are set up and the transfer is performed. The package will work with whatever Zero Page is in context (with MMU register $D507), since it is convenient to use your own zero page in your programs. For transfers of less than about 16 bytes, internal memory is faster, and for longer transfers, expansion memory turns out to be faster. For really long transfers (say, 80 bytes), using the expansion memory is MUCH faster (a marginal cost of one microsecond per byte as opposed to nine). The "[zp1]" parameter is unaltered by this call, but the register values are quite changed. The "(zw1)" parameter area is also left untouched.

The "zpstore" routine works the same as "zpload" except it stores to the far memory from zero page.

The "fetch" routine fetches the given number of bytes from a far address into the RAM0 bank (not SYS0) at the given address. Unlike the zero page load routine, you can transfer up to 64K of memory with this routine. Again, the type of memory to be fetched is transparent to the user. For an internal memory fetch, the transfers are performed in 256 byte chunks. This makes the implementation easier. For each byte transferred from RAM1, RAM1 is switched in and then RAM0 is switched in, so the transfer is not extremely efficient. For the expansion memory, the REU Controller is set up and then the entire transfer (up to 64K) is performed at a rate of 1 Meg/second. This is considerably faster than internal memory fetching. This routine handles a transfer length of 0 bytes properly. The "zp1" and "zw1" parameters are returned unaltered, but again, the registers are smashed.

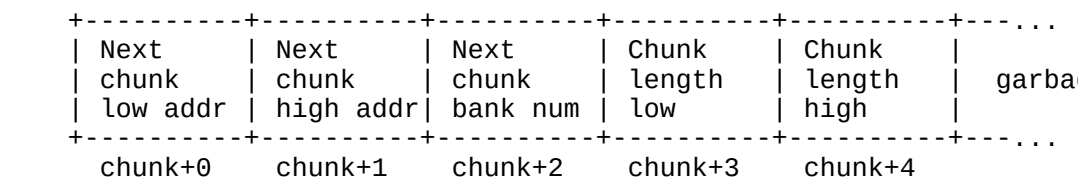
The "stash" routine operates the same as fetch, except that the data is transferred from the near ("zw1") address to the far ("zp1") address.

The "malloc" routine attempts to find a chunk of contiguous memory of the given length to allocate to you. If it can find one, it returns the far pointer to it in the "[zp1]" parameter. If it cannot find one, it returns with the carry flag set. This routine clobbers the registers.

The "free" routine returns to the pool of free memory the chunk of memory specified by the far pointer and length parameters. This routine clobbers the "[zp1]" parameter and the registers. The carry flag is always cleared upon return, since the routine does not (currently) check for any errors.

3.2. MEMORY ALLOCATE AND FREE

The malloc and free routines maintain a linked list of free memory chunks. A free memory chunk is described by a five byte structure that is at the beginning of the chunk. The first three bytes are a far pointer to the next free memory chunk and the following two bytes give the total length of the chunk. The structure is thus:



All of the free (and allocated) memory chunks are always aligned on an eight byte boundary. This guarantees that no matter what happens, there will always be at least eight bytes available in each free memory chunk to hold the free chunk descriptor information. Thus, if you were to make a request for three bytes, the system would give you eight, and when you request to free those three bytes, the system would automatically free eight. This can lead to some some wasted space when using small structures.

The memory chunks are kept in order of "increasing" address. I say "increasing" because while the chunks within a bank are in increasing address order, the system considers bank number $87 (expansion bank 7) to be lower than bank number $3F (RAM0). This anomaly makes the system allocate its external memory before allocating internal memory. This is good since external memory generally works faster than internal memory.

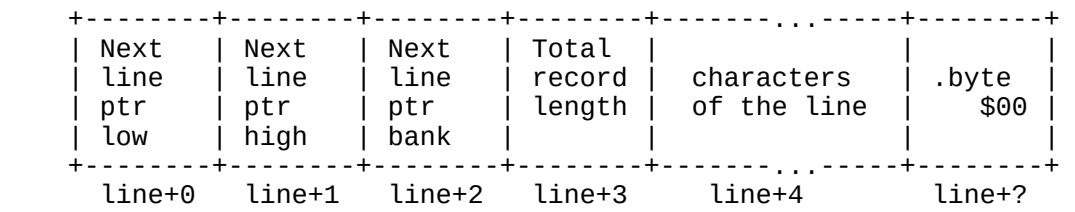
This memory is allocated first since the malloc routine uses a first-find algorithm for searching for a sufficient free memory chunk. It stops searching when it finds a free memory chunk large enough to satisfy the user's request. If the free chunk is exactly the same size as the request, the free chunk is unlinked from the free chunk list and the pointer is returned. If the free chunk is larger than the requested size, it is split up. A pointer to the top N bytes of the chunk is retured to the user and the size of the free chunk is reduced by N. The memory is allocated from the top of the chunk to make it so no linking and unlinking has to take place in this case.

The free routine is more complicated than the allocate routine since free has to deal with more cases. Free has to search through the linked list of free memory chunks to find the two chunks that straddle the chunk to be freed. Free attempts to coalesce (merge) the new chunk with the previous chunk and with the next chunk in order to end up with the largest free chunk that it can under the circumstances. Large free chunks are good since they can be used for larger requests. Two chunks can be coalesced if they are side-by-side in memory (zero bytes apart) and on the same bank. Note that chunks on different banks cannot be coalesced together, so the largest possible free chunk is 64K in length. To coalesce them, the size of the first one is increased by the size of the second one and the pointer to the second one is forgotten.

Note that this scheme works differently from the dynamic allocation scheme that BASIC uses for its strings. BASIC does not attempt to coalesce together (or even re-use) freed chunks; it relies upon garbage collecting to get rid of the free chunks. The scheme implemented here is more static (interesting word to choose) in that once you are allocated a chunk, that chunk is pinned to that address and will never move. This static organization can lead to the problem of memory fragmentation, where lots of memory can be free but is in un-coalescable chunks that are too small to be useful. Oh well. I don't think that it is really a problem for storing lines of text as individual records, and it is no problem at all for a program that always uses fixed size records.

3.3. THE SORT UTILITY

The sort utility makes full use of the capabilites of the package. First it reads in the input file one line at a time and stores the lines in a linked list as individual records of the form:



Note that these are variable length records; each record is only as long as it has to be. The total record length is stored at the front of the record. In order to read a line into a processing buffer, a "zpload" is done that reads the first four bytes of the record in order to get the length of the record. Then the entire record can be fetched since its length is known at that time. Each record ends with a $00 byte to simplify the string comparison subroutine.

The line list is maintained in alphabetical order (actually, reverse alphabetical order; below). When a new line is read in from the input file, the line list is searched for the two other lines whose values straddle the value of the new line. The line is then linked in at that position in the list. No other lines have to be moved around since pointers are used to maintain the order of the list. In order for a line already in the list to be compared with the new line, the old line has to be fetched from far memory (using the zpload + fetch scheme above) into a work buffer in the SYS0 bank. On average, half of the existing list will have to be searched in this way in order to find the correct spot to insert the new line.

After the position for the new line is found, space for the line is allocated by calling "malloc" and then the data is stored from the work buffer it was read into to far memory. The zpload and zpstore routines are used to modify the pointers to link in the new line. A number of pointer manipulations are also required on the zero page varialbles.

If the line list was generated in forward alphabetic order, then the utility would achieve its WORST performance when the input file was already mostly or partially sorted. This is because when each line is read, if it comes after most or all of the other lines, the most or all of the line list would have to be searched to find the final resting position for the new line. This would be unacceptable and extremely wasteful. A better scheme is to generate the line list in reverse alphabetic order. Then, when a "higher valued" line is read in, its correct position would be at or near the top of the list, so it would only have to be compared against a few of the lines already on the list. In the case of an input file that is already in pretty much random order, it makes no difference whether the list is in forward or reverse order.

Since the list is generated in reverse order, it must be reveresed again before writing it to the output file, since the user would want it to be in forward order (and since this is the order that can be most easily sorted again later). A clever little subroutine is called that reverses the order of the list. It only has to make use of zpload and zpstore to read/change the first few bytes of each record, since it is not concerned with the data contents of each record.

Although this is not strictly necessary, all of the records in the line list are freed before the sort utilitiy exits. This is a good practice, and would be necessary if the program were to continue to do useful work after writing the sorted file to output. A pointer is stepped through the list (starting from the head pointer) and the space for each line is deallocated by calling free, after determining the size of the record by reading the first few bytes

of it. Since the list will be in (pretty much) random order (of addresses), the deallocation mechanism does not achieve its best performance.

A convenient jump table is set up at the start of the code to make it easier for you to link your own programs to the package. Make sure that MMU configuration value $0E is in effect before calling any of the routines. You may have to muck with the code a little bit to get it to work for you.

---

4. FOR THE EXPERT HACKER

You get to see the code that actually implements the memory package and the sort utility. I have it here in a special form; each code line is preceeded by a few special characters and the line number. The line number is there to allow me to refer to specific lines, and the special characters are there to allow you to easily extract the assembler code from the rest of this magazine (and all of my ugly comments). On a Unix system, all you have to do is execute the following command line (substitute filenames as appropriate):

```
grep '^\.%....\!' Hack2 | sed 's/^.%....\!..//' | sed 's/.%....\!//' >sort.asm
```

Dontcha just love those Unix commands! Here is the assembler code:

```
.%0001!  ;Sort utility using dynamic memory allocation with expanded memory
.%0002!  ;written 92/04/22 by Craig Bruce for C= Hacking Net Magazine
.%0003!  ;-------------------------------------------------------------------
```

This program is written for the Buddy assembler. Like most assemblers, it needs a few directives to start off, so here they are. Note that my comments come BEFORE the section of code that I am commenting on.

```
.%0004!  .mem
.%0005!  .bank 15
.%0006!  .org $1300
.%0007!
.%0008!  ;*** global declarations
.%0009!
```

Here are the zero page locations that the package uses for its own purposes. I stuck the sysWork variable over the BASIC graphics command parameters since it seems like a good place. It requires 16 bytes and is used by most of the routines for temporary storage. "temp1" is used for "very" temporary storage.

```
.%0010!  zp1 = $fa
.%0011!  temp1 = $fd
.%0012!  zw1 = $fe
.%0013!  sysWork = $80    ;16-byte block
.%0014!
```

These are the non-zero page storage locations. The common code buffer pretty much has to be at $200 since that is (about) the only free section of memory below address $0400 (in the common memory range).

```
.%0015!  comCodeBuffer = $200
.%0016!  workBuffer = $b00
.%0017!
```

These are the MMU configuration register values and some important I/O addresses.

```
.%0018!  bkSys = $0e
.%0019!  bkKernel = $00
.%0020!  bkSelect = $ff00
```

```
.%0021!  bkSelectRam0 = $ff01
.%0022!  bkSelectRam1 = $ff02
.%0023!  bkRam0 = $3f
.%0024!  bkRam1 = $7f
.%0025!  bkExp0 = $80
.%0026!  bkNull = $ff
.%0027!  zpSelect = $d507
.%0028!  reu = $df00
.%0029!  vic = $d000
.%0030!
.%0031!  errInsufficientMemory = 1
.%0032!
.%0033!  ;*** jump to main routine
.%0034!
.%0035!     jmp main
.%0036!
.%0037!  ;*** jump table
.%0038!
```

Here's that jump table.

```
.%0039!  startup   jmp internStartup
.%0040!  shutdown  jmp internShutdown
.%0041!  zpload    jmp internZpLoad
.%0042!  zpstore   jmp internZpStore
.%0043!  fetch     jmp internRam0Fetch
.%0044!  stash     jmp internRam0Stash
.%0045!  malloc    jmp internAlloc
.%0046!  free      jmp internFree
.%0047!
.%0048!  ;*** storage
.%0049!
```

Here are some useful storage locations. "errno" contains the code for the error encountered in a routine if the routine exits with the carry flag set (and it is supposed to be cleared for OK). "nExpBanks" gives the number of expansion memory banks, and "freeMemory" gives the number of bytes currently free in the system. Both of these are useful status values and can be read directly.

```
.%0050!  errno      .buf 1
.%0051!  nExpBanks  .buf 1
.%0052!  mallocHead .buf 3
.%0053!  freeMemory .buf 3
.%0054!
.%0055!  ;***startup
.%0056!
```

This routine gets the ball rolling. It clears the status register in case you start up the system with the decimal mode flag set or interrupts disabled.

```
.%0057!  internStartup = *
.%0058!     lda #0
.%0059!     pha
.%0060!     plp
.%0061!     lda #bkSys
.%0062!     sta bkSelect
.%0063!     jsr installCommonCode
.%0064!     jsr sniffREU
.%0065!     jsr initDynamicMemory
.%0066!     rts
.%0067!
```

And this routine stops the ball from rolling. I fill the BASIC command line buffer with zeros to stop that syntax error thing.

```
.%0068!  internShutdown = *
.%0069!     ldx #0
.%0070!     lda #0
.%0071!  -  sta $200,x
.%0072!     inx
.%0073!     cpx #comCodeEnd-comCodeStart
.%0074!     bne -
.%0075!     lda #bkKernel
.%0076!     sta bkSelect
.%0077!     rts
.%0078!
.%0079!  ;***install common code
.%0080!
```

This routine copies the common code subroutines into the common code buffer (at $0200).

```
.%0081!  installCommonCode = *
.%0082!     ldx #0
.%0083!  -  lda comCodeStart,x
.%0084!     sta comCodeBuffer,x
.%0085!     inx
.%0086!     cpx #comCodeEnd-comCodeStart
.%0087!     bcc -
.%0088!     rts
.%0089!
.%0090!  ;-----------------------------------------------------------------------
.%0091!  ;***common code
.%0092!
```

And this is the common code. It contains four subroutines for accessing RAM1 (and the zero page routines are used for RAM0 as well).

```
.%0093!  comCodeStart = *
.%0094!
```

Selects the MMU configuration according to the bank number and copies the number of bytes required for a zpload. It exits by restoring the SYS bank. This is used only for internal memory zploads.

```
.%0095!  comZpLoad = *
.%0096!     lda zp1+2
.%0097!     sta bkSelect
.%0098!     sty temp1
.%0099!     ldy #0
.%0100!  -  lda (zp1),y
.%0101!     sta 0,x
.%0102!     inx
.%0103!     iny
.%0104!     cpy temp1
.%0105!     bcc -
.%0106!     lda #bkSys
.%0107!     sta bkSelect
.%0108!     rts
.%0109!
```

Pretty much the same as zpload.

```
.%0110!  comZpStore = *
.%0111!     lda zp1+2
.%0112!     sta bkSelect
.%0113!     sty temp1
.%0114!     ldy #0
.%0115!  -  lda 0,x
```

```
.%0116!     sta (zp1),y
.%0117!     inx
.%0118!     iny
.%0119!     cpy temp1
.%0120!     bcc -
.%0121!     lda #bkSys
.%0122!     sta bkSelect
.%0123!     rts
.%0124!
```

As the name suggests, this copies from RAM1 to RAM0. Only .Y number of bytes are copied, and if .Y=0, 256 bytes are copied. You'll notice that the MMU configurations are switched between for every byte copied. This is not the most efficient scheme, but it suffices. The MMU preconfiguration registers are used and the value that BASIC put in them are assumed to still be there.

```
.%0125!  comCopyRam1ToRam0 = *
.%0126!     dey
.%0127!     beq +
.%0128!  -  sta bkSelectRam1
.%0129!     lda (zp1),y
.%0130!     sta bkSelectRam0
.%0131!     sta (zw1),y
.%0132!     dey
.%0133!     bne -
.%0134!  +  sta bkSelectRam1
.%0135!     lda (zp1),y
.%0136!     sta bkSelectRam0
.%0137!     sta (zw1),y
.%0138!     lda #bkSys
.%0139!     sta bkSelect
.%0140!     rts
.%0141!
```

The opposite direction.

```
.%0142!  comCopyRam0ToRam1 = *
.%0143!     dey
.%0144!     beq +
.%0145!  -  sta bkSelectRam0
.%0146!     lda (zw1),y
.%0147!     sta bkSelectRam1
.%0148!     sta (zp1),y
.%0149!     dey
.%0150!     bne -
.%0151!  +  sta bkSelectRam0
.%0152!     lda (zw1),y
.%0153!     sta bkSelectRam1
.%0154!     sta (zp1),y
.%0155!     lda #bkSys
.%0156!     sta bkSelect
.%0157!     rts
.%0158!
```

The end of the common code. The length of the common code is determined by subtracting the end address from the start address.

```
.%0159!  comCodeEnd = *
.%0160!
.%0161!  ;---------------------------------------------------------------------
.%0162!  ;*** zpload( [zp1]=Source, .X=ZpDest, .Y=Length )
.%0163!
```

The actual zpload routine. It dispatches to the common code routine if internal memory is specified by the far pointer, or falls through to REU code

if expansion memory is specified.

```
.%0164!  internZpLoad = *
.%0165!     lda zp1+2
.%0166!     bmi +
.%0167!     jmp comZpLoad-comCodeStart+comCodeBuffer
.%0168!  +  sty reu+7
.%0169!     ldy #$91
.%0170!
```

Sets up the REU Controller registers for the parameters of the transfer. Note that the value of the zero page address is not assumed to be absolute $0000 but is taken from the zero page selection register of the MMU. The REU Controller does not use the MMU for decoding zero page and stack page addresses; it accesses the absolute memory directly.

```
.%0171!  zeroPageReuOp = *
.%0172!     sta reu+6
.%0173!     stx reu+2
.%0174!     lda zpSelect
.%0175!     sta reu+3
.%0176!     lda zp1
.%0177!     sta reu+4
.%0178!     lda zp1+1
.%0179!     sta reu+5
.%0180!     lda #0
.%0181!     sta reu+8
```

Here the system clock speed is put into Slow mode while the transfer occurs and is then restored. This is necessary.

```
.%0182!     lda vic+$30
.%0183!     ldx #$00
.%0184!     stx vic+$30
.%0185!     sty reu+1
.%0186!     sta vic+$30
.%0187!     rts
.%0188!
.%0189!  ;*** zpstore( .X=ZpSource, [zp1]=Dest, .Y=Length )
.%0190!
```

Pretty much the same as the zpload routine, except that a command code for the REU Controller is different (specifying an internal to expansion memory transfer). The REU code in the zpload routine is called.

```
.%0191!  internZpStore = *
.%0192!     lda zp1+2
.%0193!     bmi +
.%0194!     jmp comZpStore-comCodeStart+comCodeBuffer
.%0195!  +  sty reu+7
.%0196!     ldy #$90
.%0197!     jmp zeroPageReuOp
.%0198!
.%0199!  ;---------------------------------------------------------------------
.%0200!  ;*** fetch( [zp1]=FarSource, (zw1)=Ram0Dest, .AY=Length )
.%0201!
```

Some working storage locations are necessary for this routine, since it is designed to copy data a page at a time. The source (zp1+1) and destination (zw1+1) page addresses are saved and later restored because this routine alters them while copying. If the far address is in expansion memory, this routine dispatches to the REU fetch/stash code.

```
.%0202!  fetchLength = sysWork
.%0203!  fetchSaveSource = sysWork+2
```

```
.%0204!  fetchSaveDest = sysWork+3
.%0205!
.%0206!  internRam0Fetch = *
.%0207!     ldx zp1+2
.%0208!     bpl +
.%0209!     ldx #$91
.%0210!     jmp doReu
```

If the transfer is less than one page long, it can be done by calling the fetchPage code directly. Otherwise, the long fetch code has to be called.

```
.%0211!  +  cpy #0
.%0212!     bne fetchLong
.%0213!     tay
.%0214!     bne fetchPage
.%0215!     rts
.%0216!
```

If the (internal) page to be fetched is on RAM1, the common code routine is called; otherwise, the copy is done here by switching RAM0 into context. We can copy between RAM0 locations without switching contexts for every byte.

```
.%0217!     fetchPage = *
.%0218!     cpx #bkRam0
.%0219!     beq +
.%0220!     jmp comCopyRam1ToRam0-comCodeStart+comCodeBuffer
.%0221!  +  stx bkSelect
.%0222!     dey
.%0223!     beq +
.%0224!  -  lda (zp1),y
.%0225!     sta (zw1),y
.%0226!     dey
.%0227!     bne -
.%0228!  +  lda (zp1),y
.%0229!     sta (zw1),y
.%0230!     lda #bkSys
.%0231!     sta bkSelect
.%0232!     rts
.%0233!
```

This is called for long (>=256 byte) (internal) fetches. It calls the fetchPage code repeatedly, after incrementing the source and destination page numbers. The transfer length is decremented until it is less than 256 bytes.

```
.%0234!     fetchLong = *
.%0235!     sta fetchLength
.%0236!     sty fetchLength+1
.%0237!     lda zp1+1
.%0238!     sta fetchSaveSource
.%0239!     lda zw1+1
.%0240!     sta fetchSaveDest
.%0241!     lda fetchLength+1
.%0242!     beq fetchLongExit
.%0243!  -  ldx zp1+2
.%0244!     ldy #0
.%0245!     jsr fetchPage
.%0246!     inc zp1+1
.%0247!     inc zw1+1
.%0248!     dec fetchLength+1
.%0249!     bne -
.%0250!
.%0251!     fetchLongExit = *
```

This fetches the last chunk of less than 256 bytes and then restores the zp1 and zw1 parameters to what they were before this routine was called.

```
.%0252!     ldy fetchLength
.%0253!     beq +
.%0254!     ldx zp1+2
.%0255!     jsr fetchPage
.%0256!  +  lda fetchSaveSource
.%0257!     sta zp1+1
.%0258!     lda fetchSaveDest
.%0259!     sta zw1+1
.%0260!     rts
.%0261!
.%0262!  ;*** stash( (zw1)=Ram0Source, [zp1]=FarDest, .AY=length )
.%0263!
```

Stash has exactly the same structure as fetch.

```
.%0264!  stashLength = sysWork
.%0265!  stashSaveSource = sysWork+2
.%0266!  stashSaveDest = sysWork+3
.%0267!
.%0268!  internRam0Stash = *
.%0269!     ldx zp1+2
.%0270!     bpl +
.%0271!     ldx #$90
.%0272!     jmp doReu
.%0273!  +  cpy #0
.%0274!     bne stashLong
.%0275!     tay
.%0276!     bne stashPage
.%0277!     rts
.%0278!
.%0279!     stashPage = *
.%0280!     cpx #bkRam0
.%0281!     beq +
.%0282!     jmp comCopyRam0ToRam1-comCodeStart+comCodeBuffer
.%0283!  +  stx bkSelect
.%0284!     dey
.%0285!     beq +
.%0286!  -  lda (zw1),y
.%0287!     sta (zp1),y
.%0288!     dey
.%0289!     bne -
.%0290!  +  lda (zw1),y
.%0291!     sta (zp1),y
.%0292!     lda #bkSys
.%0293!     sta bkSelect
.%0294!     rts
.%0295!
.%0296!     stashLong = *
.%0297!     sta stashLength
.%0298!     sty stashLength+1
.%0299!     lda zw1+1
.%0300!     sta stashSaveSource
.%0301!     lda zp1+1
.%0302!     sta stashSaveDest
.%0303!     lda stashLength+1
.%0304!     beq stashLongExit
.%0305!  -  ldx zp1+2
.%0306!     ldy #0
.%0307!     jsr stashPage
.%0308!     inc zp1+1
.%0309!     inc zw1+1
.%0310!     dec stashLength+1
.%0311!     bne -
.%0312!
```

```
.%0313!     stashLongExit = *
.%0314!     ldy stashLength
.%0315!     beq +
.%0316!     ldx zp1+2
.%0317!     jsr stashPage
.%0318!  +  lda stashSaveSource
.%0319!     sta zw1+1
.%0320!     lda stashSaveDest
.%0321!     sta zp1+1
.%0322!     rts
.%0323!
.%0324!  ;*** ram0 load/store(.X) expn memory [zp1] <- -> (zw1) for .AY bytes
.%0325!
```

This is the code that does the fetching and stashing from/to expansion memory. The only difference between a fetch and a stash is the REU Controller command code, so that is an input parameter. The REU Controller registers are set up, the clock is slowed, the transfer happens, and then the clock speed is restored. The bulk transfer is done entirely by the REU Controller. Interestingly, it would have been faster to transfer the internal memory to expansion memory and then fetch it back again in order to achieve an internal memory transfer (if you have an REU), but I didn't bother with that.

```
.%0326!  doReu = *
.%0327!     sta reu+7
.%0328!     sty reu+8
.%0329!     lda zw1
.%0330!     ldy zw1+1
.%0331!     sta reu+2
.%0332!     sty reu+3
.%0333!     lda zp1
.%0334!     ldy zp1+1
.%0335!     sta reu+4
.%0336!     sty reu+5
.%0337!     lda zp1+2
.%0338!     sta reu+6
.%0339!     ldy vic+$30
.%0340!     lda #0
.%0341!     sta vic+$30
.%0342!     stx reu+1
.%0343!     sty vic+$30
.%0344!     rts
.%0345!
.%0346!  ;*** sniffREU - determine number of banks of expansion memory
.%0347!
```

The work locations are used to store a string to the first four addresses of each expansion memory bank and then fetch them back again in order to determine whether the bank exists or not. Expansion bank #0 is also checked after each bank to see if a bank number wrap-around occured. The "reuSizeLimit" will force this routine to stop searching after that number of banks have been sniffed. The maximum value is 127, since only bank numbers $80 to $FE are available. By changing this value, you can stop this package from using expansion memory reserved by another program. Note that this program uses expansion banks 0 up to but not including "reuSizeLimit".

```
.%0348!  sniffWork1 = sysWork
.%0349!  sniffWork2 = sysWork+4
.%0350!  reuSizeLimit .byte 127
.%0351!
.%0352!  sniffREU = *
```

Here I save the data in the memory "beneath" the REU Controller registers. If there isn't a REU installed, this memory would otherwise be corrupted by I/O addresses bleeding through to the underlying RAM.

```
.%0353!     lda #bkRam0
.%0354!     sta bkSelect
.%0355!     ldx #$a
.%0356!  -  lda reu,x
.%0357!     sta workBuffer,x
.%0358!     dex
.%0359!     bpl -
.%0360!     lda #bkSys
.%0361!     sta bkSelect
```

Here I initialize the configuration REU Controller registers. They are set only once by this package.

```
.%0362!     lda #$00
.%0363!     sta reu+$9
.%0364!     sta reu+$a
.%0365!     lda reu+$0
```

The three-byte identifier string is copied into the source tag. The fourth byte will be filled in by the bank number.

```
.%0366!     ldx #2
.%0367!  -  lda expRamId,x
.%0368!     sta sniffWork1,x
.%0369!     dex
.%0370!     bpl -
```

Initialization continues.

```
.%0371!     lda #0
.%0372!     sta nExpBanks
.%0373!     lda #$00
.%0374!     ldx #bkExp0
.%0375!     sta zp1
.%0376!     sta zp1+1
.%0377!     stx zp1+2
.%0378!
```

This is the main loop. It tests the current expansion bank and then goes on to the next one if ok. Otherwise, it stops at the number of okay banks.

```
.%0379!  -  jsr testExpBank
.%0380!     bcs +
.%0381!     inc nExpBanks
.%0382!     inc zp1+2
.%0383!     bne -
.%0384!  +  lda nExpBanks
.%0385!     bne +
```

Restore the underlying RAM contents and exit.

```
.%0386!     lda #bkRam0
.%0387!     sta bkSelect
.%0388!     ldx #$a
.%0389!  -  lda workBuffer,x
.%0390!     sta reu,x
.%0391!     dex
.%0392!     bpl -
.%0393!     lda #bkSys
.%0394!     sta bkSelect
.%0395!  +  rts
.%0396!
.%0397!  ;*** test expansion bank( [zp1]=BankPtr ) : .CC=ok
.%0398!
```

First checks that the maximum number of allowed expansion banks has not been exceeded. Stores the test string through the bank pointer and then tests to see that the string has been stored correctly and that the string on expansion bank 0 is still ok (it wouldn't be ok if a wrap-around occured).

```
.%0399!  testExpBank = *
.%0400!     lda nExpBanks
.%0401!     cmp reuSizeLimit
.%0402!     bcc +
.%0403!     rts
.%0404!  +  lda zp1+2
.%0405!     sta sniffWork1+3
.%0406!     ldx #sniffWork1
.%0407!     ldy #4
.%0408!     jsr zpstore
.%0409!     jsr testExpBankInternal  ;test current bank
.%0410!     bcs +
.%0411!     lda zp1+2
.%0412!     pha
.%0413!     lda #bkExp0
.%0414!     sta zp1+2
.%0415!     sta sniffWork1+3
.%0416!     jsr testExpBankInternal  ;test expansion bank 0
.%0417!     pla
.%0418!     sta zp1+2
.%0419!  +  rts
.%0420!
```

This routine reads the bytes at address [zp1] and makes sure they are the same as the previous routine put there. On return, the carry flag is set if the string found is not the same as what was previously put out.

```
.%0421!  testExpBankInternal = *
.%0422!     lda #$00
.%0423!     sta sniffWork2
.%0424!     sta sniffWork2+3
.%0425!     ldx #sniffWork2
.%0426!     ldy #4
.%0427!     jsr zpload
.%0428!     ldx #3
.%0429!  -  lda sniffWork2,x
.%0430!     cmp sniffWork1,x
.%0431!     bne +
.%0432!     dex
.%0433!     bpl -
.%0434!     clc
.%0435!     rts
.%0436!  +  sec
.%0437!     rts
.%0438!
```

This is the three-byte string put into the expansion banks. The value means "RAM identifier".

```
.%0439!  expRamId   .byte "r"
.%0440!             .byte "I"
.%0441!             .byte "d"
.%0442!
.%0443!  ;---------------------------------------------------------------------
.%0444!  ;*** initialize dynamically allocated memory() : nExpBanks
.%0445!
```

This routine calls "free" to initialize the free memory on each existing bank. RAM0 is set to be free from $4000 to the top of BASIC memory, so you'll

have to change the "ram0FreeStartPage" parameter if you want to have a program that occupies memory higher than this address. RAM1 is declared to be free from $0400 to $FEFF

```
.%0446!  ram0FreeStartPage .byte $40
.%0447!  ram1FreeStartPage .byte $04
.%0448!  ram1FreeLength    .byte 256-1-$04
.%0449!
.%0450!  currentExpBank = sysWork+$f
.%0451!
.%0452!  initDynamicMemory = *
```

Set the memory allocation first free chunk pointer to Null and set the number of bytes of free memory to 0.

```
.%0453!     ldx #2
.%0454!  -  lda #$00
.%0455!     sta freeMemory,x
.%0456!     lda #$ff
.%0457!     sta mallocHead,x
.%0458!     dex
.%0459!     bpl -
```

Determine the length of free memory on RAM0 and free the memory.

```
.%0460!     sec
.%0461!     lda $1212   ;top of BASIC program Low
.%0462!     beq +
.%0463!     clc
.%0464!  +  lda $1213   ;top of BASIC program High
.%0465!     sbc ram0FreeStartPage
.%0466!     tay
.%0467!     lda ram0FreeStartPage
.%0468!     ldx #bkRam0
.%0469!     jsr initInternalBankMalloc
```

Free the memory of RAM1

```
.%0470!     lda ram1FreeStartPage
.%0471!     ldy ram1FreeLength
.%0472!     ldx #bkRam1
.%0473!     jsr initInternalBankMalloc
.%0474!
```

For each existing expansion bank, free it from addresses $0000 to $FFF7. You cannot free all 65536 bytes since this would cause the length of the free chunk to be set to $0000 which would cause problems later on. $FFF8 bytes are set to free since then length has to be a multiple of eight bytes.

```
.%0475!     lda #0
.%0476!     sta currentExpBank
.%0477!  -  lda currentExpBank
.%0478!     cmp nExpBanks
.%0479!     bcs +
.%0480!     ora #bkExp0
.%0481!     sta zp1+2
.%0482!     lda #$00
.%0483!     sta zp1
.%0484!     sta zp1+1
.%0485!     lda #$f8
.%0486!     ldy #$ff
.%0487!     jsr free
.%0488!     inc currentExpBank
.%0489!     bne -
.%0490!  +  rts
```

```
.%0491!

This routine is called for freeing banks RAM0 and RAM1.  It does nothing other
than set parameters and is put in for convenience.

.%0492!  initInternalBankMalloc = *
.%0493!     sta zp1+1
.%0494!     stx zp1+2
.%0495!     lda #0
.%0496!     sta zp1
.%0497!     jmp free
.%0498!
.%0499!  ;---------------------------------------------------------------------
.%0500!  ;*** malloc( .AY=Bytes ) : [zp1]=FarPointer
.%0501!

One of the biggies.  The "MemNextPtr" and "MemLength" variables are used to
store the information at the start of the current free memory chunk.  "Length"
is used to hold the length input parameter and "Q" is the pointer to the
previous free memory chunk whereas "zp1" is used to point to the current free
chunk.  I prefix these variables with "malloc" to avoid naming collisions with
other routines.  The concept of local variables might be a nice thing for
future assemblers to have.

.%0502!  mallocMemNextPtr = sysWork
.%0503!  mallocMemLength  = sysWork+3
.%0504!  mallocLength     = sysWork+5
.%0505!  mallocQ          = sysWork+7
.%0506!
.%0507!  internAlloc = *

Align the number of bytes requested to an even multiple of eight.

.%0508!     clc
.%0509!     adc #7
.%0510!     bcc +
.%0511!     iny
.%0512!  +  and #$f8
.%0513!     sta mallocLength
.%0514!     sty mallocLength+1

Set the current free chunk pointer to the first free chunk and set Q to Null.

.%0515!     ldx #2
.%0516!  -  lda mallocHead,x
.%0517!     sta zp1,x
.%0518!     lda #$ff
.%0519!     sta mallocQ,x
.%0520!     dex
.%0521!     bpl -
.%0522!

Search for a free chunk that is long enough to satisfy the request.

.%0523!     mallocLook = *

If the current free chunk pointer is Null, then we are S.O.L. (Out of Luck)
since that means we have exhausted the list of free chunks and have to report
that insufficient free memory could be found.

.%0524!     lda zp1+2
.%0525!     cmp #$ff
.%0526!     bne +
.%0527!
.%0528!     mallocErrorExit = *
```

```
.%0529!     lda #$ff   ;return a Null pointer
.%0530!     sta zp1
.%0531!     sta zp1+1
.%0532!     sta zp1+2
.%0533!     lda #errInsufficientMemory
.%0534!     sta errno
.%0535!     sec
.%0536!     rts
.%0537!
```

Fetch the header information of the current free chunk and check the length. If the current free chunk is not large enough, then we set the Q pointer to the current pointer, and take the new value for the current pointer from the header of the current free chunk (mallocMemNextPtr) and then continue searching.

```
.%0538!  +  ldx #mallocMemNextPtr
.%0539!     ldy #5
.%0540!     jsr zpload
.%0541!     lda mallocMemLength
.%0542!     cmp mallocLength
.%0543!     lda mallocMemLength+1
.%0544!     sbc mallocLength+1
.%0545!     bcs mallocGotBlock
.%0546!     ldx #2
.%0547!  -  lda zp1,x
.%0548!     sta mallocQ,x
.%0549!     lda mallocMemNextPtr,x
.%0550!     sta zp1,x
.%0551!     dex
.%0552!     bpl -
.%0553!     jmp mallocLook
.%0554!
```

Now, we've found a block that is large enough.

```
.%0555!     mallocGotBlock = *
.%0556!     sec
```

Subtract the number of bytes requested from the total number of bytes free.

```
.%0557!     lda freeMemory
.%0558!     sbc mallocLength
.%0559!     sta freeMemory
.%0560!     lda freeMemory+1
.%0561!     sbc mallocLength+1
.%0562!     sta freeMemory+1
.%0563!     bcs +
.%0564!     dec freeMemory+2
```

If the size of the current free chunk is exactly the same as the number of bytes requested, then branch ahead.

```
.%0565!  +  lda mallocMemLength
.%0566!     cmp mallocLength
.%0567!     bne +
.%0568!     lda mallocMemLength+1
.%0569!     sbc mallocLength+1
.%0570!     beq mallocTakeWholeBlock
```

Subtract the number of bytes requested from the length of the current free chunk and then write the updated header back to the current free chunk.

```
.%0571!  +  sec
.%0572!     lda mallocMemLength
```

```
.%0573!     sbc mallocLength
.%0574!     sta mallocMemLength
.%0575!     lda mallocMemLength+1
.%0576!     sbc mallocLength+1
.%0577!     sta mallocMemLength+1
.%0578!     ldx #mallocMemNextPtr
.%0579!     ldy #5
.%0580!     jsr zpstore
```

Add the length of the free chunk to the pointer to the start of the free chunk to determine the address of the memory that has just been allocated. Then exit, returning this address.

```
.%0581!     clc
.%0582!     lda zp1
.%0583!     adc mallocMemLength
.%0584!     sta zp1
.%0585!     lda zp1+1
.%0586!     adc mallocMemLength+1
.%0587!     sta zp1+1
.%0588!     clc
.%0589!     rts
.%0590!
```

Here, the size of the free chunk is exactly the same size as the request, so the entire block has to be allocated and thus removed from the free chunk list. This is why the Q pointer has been maintained.

```
.%0591!     mallocTakeWholeBlock = *
```

If there is no previous block (Q == Null) then set the free chunk list head pointer to the next free chunk after the current one. Then exit with the current chunk as the return pointer.

```
.%0592!     lda mallocQ+2
.%0593!     cmp #bkNull
.%0594!     bne +
.%0595!     ldx #2
.%0596!  -  lda mallocMemNextPtr,x
.%0597!     sta mallocHead,x
.%0598!     dex
.%0599!     bpl -
.%0600!     clc
.%0601!     rts
```

If there is an actual previous chunk, then we have to set it to point to the next chunk from the current chunk. This will unlink the current free chunk from the free chunk list, thereby allocating it.

First, we swap the Q and current pointers, since we can only access memory through the "zp1" pointer.

```
.%0602!  +  ldx #2
.%0603!  -  lda zp1,x
.%0604!     ldy mallocQ,x
.%0605!     sta mallocQ,x
.%0606!     sty zp1,x
.%0607!     dex
.%0608!     bpl -
```

Then we set the the NextPointer of the previous free chunk to point to the next free chunk after the current chunk.

```
.%0609!     ldx #mallocMemNextPtr
.%0610!     ldy #3
```

```
.%0611!     jsr zpstore
```

And then we restore the current chunk pointer and return it to the user.

```
.%0612!     ldx #2
.%0613!  -  lda mallocQ,x
.%0614!     sta zp1,x
.%0615!     dex
.%0616!     bpl -
.%0617!     clc
.%0618!     rts
.%0619!
.%0620!  ;*** free( [zp1]=FarPointer, .AY=Length )  {alters [zp1]}
.%0621!
```

And here is the real biggie, since Free is more complicated than Malloc. The variables are the same as for free, except that "NewPtr" is required to remember the input parameter to new chunk to be freed.

```
.%0622!  freeMemNextPtr = sysWork
.%0623!  freeMemLength  = sysWork+3
.%0624!  freeLength     = sysWork+5
.%0625!  freeNewPtr     = sysWork+7
.%0626!  freeQ          = sysWork+10
.%0627!
.%0628!  internFree = *
```

Again, align the length of the chunk. The pointer to the start of the new chunk is assumed to be aligned (since malloc only returns aligned chunks). If the chunk pointer is not aligned, all hell can break loose.

```
.%0629!     clc
.%0630!     adc #7
.%0631!     bcc +
.%0632!     iny
.%0633!  +  and #$f8
.%0634!     sta freeLength
.%0635!     sty freeLength+1
```

Save the new chunk input parameter and set "zp1" for searching the free chunk list. Also set Q to Null since Q will be used to remember the previous block to "zp1".

```
.%0636!     ldx #2
.%0637!  -  lda zp1,x
.%0638!     sta freeNewPtr,x
.%0639!     lda mallocHead,x
.%0640!     sta zp1,x
.%0641!     lda #$ff
.%0642!     sta freeQ,x
.%0643!     dex
.%0644!     bpl -
.%0645!
```

Search for the two free chunks whose addresses straddle the new free chunk.

```
.%0646!     freeSearchLoop = *
```

If the current free chunk pointer is Null or if the current free chunk's bank number is less than the new chunk's bank number, then we can stop searching; we have found a free chunk that is "higher" than the new chunk, so Q and zp1 must straddle the address of the new chunk. Note that by using a "bcc" on line 652, external memory free chunks will be allocated first. If I had used a "bcs" there, the internal memory starting from RAM0 would be allocated first.

```
.%0647!     lda zp1+2
.%0648!     cmp #$ff
.%0649!     beq freeCoalesceQandNew
.%0650!     lda zp1+2
.%0651!     cmp freeNewPtr+2
.%0652!     bcc freeCoalesceQandNew  ;** determines bank order
```

Here we know that the bank number is not "higher", so if the bank numbers are not equal, then we continue searching. If the bank numbers are equal, we must check the addresses within the bank to see if zp1 is higher than the new chunk. If so, we stop searching.

```
.%0653!     bne +
.%0654!     lda zp1
.%0655!     cmp freeNewPtr
.%0656!     lda zp1+1
.%0657!     sbc freeNewPtr+1
.%0658!     bcs freeCoalesceQandNew
```

Here we continue searching. We stick the current free chunk pointer into Q and get the next free chunk pointer from the current chunk in memory. Then we go back to the top of the search.

```
.%0659!  +  ldx #freeMemNextPtr
.%0660!     ldy #3
.%0661!     jsr zpload
.%0662!     ldx #2
.%0663!  -  lda zp1,x
.%0664!     sta freeQ,x
.%0665!     lda freeMemNextPtr,x
.%0666!     sta zp1,x
.%0667!     dex
.%0668!     bpl -
.%0669!     bmi freeSearchLoop
.%0670!
```

Here we know that Q and zp1 straddle the new chunk, and we try to coalesce the new chunk to the Q chunk.

```
.%0671!     freeCoalesceQandNew = *
.%0672!     ldx #2
.%0673!  -  lda freeQ,x
.%0674!     sta zp1,x
.%0675!     dex
.%0676!     bpl -
```

If the Q pointer is Null, then there is no Q chunk to coalesce with, so the free chunk head pointer is set to point to the new chunk and the new chunk header is set to the size of the new chunk. Then next pointer for the new chunk is set to what was previously the head pointer.

```
.%0677!     lda zp1+2
.%0678!     cmp #$ff
.%0679!     bne +
.%0680!     ldx #2
.%0681!  -  lda mallocHead,x
.%0682!     sta freeMemNextPtr,x
.%0683!     lda freeNewPtr,x
.%0684!     sta mallocHead,x
.%0685!     dex
.%0686!     bpl -
.%0687!     lda freeLength
.%0688!     ldy freeLength+1
.%0689!     sta freeMemLength
```

```
.%0690!     sty freeMemLength+1
.%0691!     jmp freeCoalesceNewAndP
.%0692!
```

Here there actually is a previous (Q) chunk, so its header is fetched. If it is not on the same bank as the new chunk, then the new chunk cannot be coalesced with it. Also, if the address of the new chunk does not exactly follow the Q chunk, then they cannot be coalesced.

```
.%0693!  +  ldx #freeMemNextPtr
.%0694!     ldy #5
.%0695!     jsr zpload
.%0696!     lda zp1+2
.%0697!     cmp freeNewPtr+2
.%0698!     bne +
.%0699!     clc
.%0700!     lda zp1
.%0701!     adc freeMemLength
.%0702!     tax
.%0703!     lda zp1+1
.%0704!     adc freeMemLength+1
.%0705!     cmp freeNewPtr+1
.%0706!     bne +
.%0707!     cpx freeNewPtr
.%0708!     bne +
```

Here, we know that the previous chunk and the new chunk can be coalesced. We add the length of the new chunk to the length of the previous chunk and change the new chunk pointer to point to the previous chunk.

```
.%0709!     clc
.%0710!     lda freeMemLength
.%0711!     adc freeLength
.%0712!     sta freeMemLength
.%0713!     lda freeMemLength+1
.%0714!     adc freeLength+1
.%0715!     sta freeMemLength+1
.%0716!     ldx #2
.%0717!  -  lda freeQ,x
.%0718!     sta freeNewPtr,x
.%0719!     dex
.%0720!     bpl -
.%0721!     bmi freeCoalesceNewAndP
.%0722!
```

Here, we know that the previous and new chunks cannot be coalesced. We change the actual header of the pervious chunk to point to the new chunk and change the new chunk header length to the free request length. The pointer to the next chunk is already in the new chunk header from before. Note that now we are using "memNextPtr" and "memLength" to construct the new free chunk header. Line 729 caused Mr. Bruce some problems because he forgot to stick the "+1" there after extracting the code from Zed.

```
.%0723!  +  ldx #freeNewPtr
.%0724!     ldy #3
.%0725!     jsr zpstore
.%0726!     lda freeLength
.%0727!     ldy freeLength+1
.%0728!     sta freeMemLength
.%0729!     sty freeMemLength+1
.%0730!
```

At this point, we are finished trying to coalesce the new chunk with the previous chunk, so we will attempt to coalesce the new chunk with the next higher address free chunk. The "memNextPtr" and "memLength" variables hold

the header information for the new chunk (the "memNextPtr" also points to the next free chunk), and "NewPtr" points to the new chunk. We check to see if the new chunk immediately preceeds the next chunk in the same way as before. Note that the case of a Null next chunk pointer is handled here implicitly, since the bank numbers won't match.

```
.%0731!     freeCoalesceNewAndP = *
.%0732!     lda freeNewPtr+2
.%0733!     cmp freeMemNextPtr+2
.%0734!     bne +
.%0735!     clc
.%0736!     lda freeNewPtr
.%0737!     adc freeMemLength
.%0738!     tax
.%0739!     lda freeNewPtr+1
.%0740!     adc freeMemLength+1
.%0741!     cmp freeMemNextPtr+1
.%0742!     bne +
.%0743!     cpx freeMemNextPtr
.%0744!     bne +
.%0745!
```

Here, we know that the new chunk can be coalesced with the next chunk. We have to fetch the header of the next chunk to know the length and the pointer to the free chunk after the next chunk. We then add the length of the next chunk to the length of the new chunk and keep the pointer to the chunk after the next chunk for the new chunk header. Effectively, the next free chunk is unlinked (since nothing is left to point to it) and the new chunk grows to swallow it up.

```
.%0746!     ldx #2
.%0747!  -  lda freeMemNextPtr,x
.%0748!     sta zp1,x
.%0749!     dex
.%0750!     bpl -
.%0751!     lda freeMemLength+1
.%0752!     pha
.%0753!     lda freeMemLength
.%0754!     pha
.%0755!     ldx #freeMemNextPtr
.%0756!     ldy #5
.%0757!     jsr zpload
.%0758!     clc
.%0759!     pla
.%0760!     adc freeMemLength
.%0761!     sta freeMemLength
.%0762!     pla
.%0763!     adc freeMemLength+1
.%0764!     sta freeMemLength+1
.%0765!
```

Here, we wrap things up. We have the header for the new free chunk all prepared and we have tried to coalesce the two neighboring chunks to the new chunk. All we do now is write the new chunk header out to main memory and increase the number of bytes free variable by the length of the (original) free request.

```
.%0766!  +  ldx #2
.%0767!  -  lda freeNewPtr,x
.%0768!     sta zp1,x
.%0769!     dex
.%0770!     bpl -
.%0771!     ldx #freeMemNextPtr
.%0772!     ldy #5
.%0773!     jsr zpstore
```

```
.%0774!     clc
.%0775!     lda freeMemory
.%0776!     adc freeLength
.%0777!     sta freeMemory
.%0778!     lda freeMemory+1
.%0779!     adc freeLength+1
.%0780!     sta freeMemory+1
.%0781!     bcc +
.%0782!     inc freeMemory+2
```

We always return with carry cleared, since we don't check for any errors.

```
.%0783!  +  clc
.%0784!     rts
.%0785!
.%0786!  ;---------------------------------------------------------------------
.%0787!  ;*** sort - the application: reads from file #1, writes to file #2
.%0788!
```

This is where the actual application code starts. If you want to write your own program that uses the dynamic memory allocation package, then you can follow the structure of this application.

We start off by declaring the storage areas for the current line being processed and for the line being compared to the current line. The addresses reflect the structure of the record for the input line that was discussed earlier. The sorting field starting column number parameter is can be put at $8FF since the input line can only be 242 characters long.

```
.%0789!  sortbuf    = $b00
.%0790!  sortbuflen = $b03
.%0791!  sortline   = $b04
.%0792!  cmpbuf     = $800
.%0793!  cmpbuflen  = $803
.%0794!  cmpline    = $804
.%0795!  sortColumn = $8ff
.%0796!
```

These are the zero page locations that sort uses.

```
.%0797!  eofstat  = $02  ;deferred ST variable ($90)
.%0798!  sorthead = $03  ;pointer to first line in line list
.%0799!  sortP    = $06  ;current line for list searching
.%0800!  sortQ    = $09  ;previous line for list searching
.%0801!  header   = $0c  ;4 bytes - holds the current line record's header
.%0802!
```

And these are the kernel routines that are called.

```
.%0803!  kernelChkin  = $ffc6
.%0804!  kernelChkout = $ffc9
.%0805!  kernelClrchn = $ffcc
.%0806!  kernelChrin  = $ffcf
.%0807!  kernelChrout = $ffd2
```

"echoStatus" can be changed to point to an RTS if you do not want sort to print status information out while it is working.

```
.%0808!  echoStatus   = kernelChrout
.%0809!
.%0810!  ;*** getline( sortline ) : .CS=eof
.%0811!
```

This routine reads a new line in from the current input channel and puts it into the processing buffer. It returns with carry set if there are no more

lines to read or if a read error occurs.

```
.%0812!  getline = *
.%0813!     ldy #0
```

The "eofstat" is checked first to see if the previous character read before the new call was the last of the file. This overcomes the kernel's awkward way of setting EOI for the last character rather than when for when you go beyond the last character.

```
.%0814!  -  bit eofstat
.%0815!     bvs getlineEof
.%0816!     jsr kernelChrin
.%0817!     bcs getlineEof
.%0818!     sta sortline,y
.%0819!     iny
.%0820!     ldx $90
.%0821!     stx eofstat
```

It exits when the maximum line length is exceeded or when a carriage return is encountered.

```
.%0822!     cpy #242
.%0823!     bcs getlineExit
.%0824!     cmp #13
.%0825!     bne -
.%0826!     dey
.%0827!
```

A trailing '\0' is appended to the string for easier processing later, and the length of the input line record is recorded. The length of the entire record rather than the length of just the text is more convenient to know when working with the memory package.

```
.%0828!     getlineExit = *
.%0829!     lda #0
.%0830!     sta sortline,y
.%0831!     clc
.%0832!     tya
.%0833!     adc #5
.%0834!     sta sortbuflen
.%0835!     clc
.%0836!     rts
.%0837!
```

On end of file, we exit with carry set. If, however, we have read characters before the EOF was encountered, they are returned as belonging to the last line of the file. True EOF will be returned on the next call.

```
.%0838!     getlineEof = *
.%0839!     lda #$40
.%0840!     sta eofstat
.%0841!     cpy #0
.%0842!     bne getlineExit
.%0843!     sec
.%0844!     rts
.%0845!
.%0846!  ;*** putline( appline )
.%0847!
```

This routine simply writes out the current line ('\0' terminated) and writes an additional carriage return, since the getline routine strips off the CR.

```
.%0848!  putline = *
.%0849!     ldy #0
```

```
.%0850!  -  lda sortline,y
.%0851!     beq +
.%0852!     jsr kernelChrout
.%0853!     iny
.%0854!     bne -
.%0855!  +  lda #13
.%0856!     jmp kernelChrout
.%0857!
.%0858!  ;*** fetchline( sortP=LinePtr, .AY=Ram0buf )
.%0859!
```

This routine fetches the line at the pointer sortP into RAM0 at the given address. It has to zpload the line header first to determine the record size to fetch.

```
.%0860!  fetchline = *
.%0861!     sta zw1
.%0862!     sty zw1+1
.%0863!     ldx #2
.%0864!  -  lda sortP,x
.%0865!     sta zp1,x
.%0866!     dex
.%0867!     bpl -
.%0868!     ldx #header
.%0869!     ldy #4
.%0870!     jsr zpload
.%0871!     lda header+3
.%0872!     ldy #0
.%0873!     jmp fetch
.%0874!
.%0875!  ;*** sortGTcmp( sortline, cmpline ) : .CS={sortline >= cmpline}
.%0876!
```

This routine compares the lines stored in the "sortline" and "cmpline" buffers and returns with carry set if the "sortline" is larger (alphabetically). It also takes into account the starting comparison positions and handles the case of either or both lines not being as long as the start position of the string comparison.

```
.%0877!  sortGTcmp = *
```

This section of code makes bit0 of .X a "1" if sortline is not long enough to be compared, and makes bit1 a "1" if cmpline is too short.

```
.%0878!     ldx #0
.%0879!     clc
.%0880!     lda sortColumn
.%0881!     adc #5
.%0882!     cmp sortbuflen
.%0883!     bcc +
.%0884!     inx
.%0885!  +  cmp cmpbuflen
.%0886!     bcc +
.%0887!     inx
.%0888!     inx
```

And here is where it takes action depending whether the lines are large enough or not. The cases are:

```
. .X=%00000000 - strings are long enough to be compared, so continue
. .X=%00000001 - sortline is too short, cmpline ok, so return with carry clear
. .X=%00000010 - cmpline is too short, sortline ok, so return with carry set
. .X=%00000011 - both sortline and cmpline are too short; carry set
```

```
.%0889!  +  txa
```

```
.%0890!     beq doCompare
.%0891!     cmp #2
.%0892!     rts
.%0893!
.%0894!     doCompare = *
```

This section does the compare if both lines are long enough.

```
.%0895!     ldy sortColumn
.%0896!  -  lda sortline,y
.%0897!     cmp cmpline,y
.%0898!     bne +
.%0899!     cmp #0
.%0900!     beq +
.%0901!     iny
.%0902!     bne -
.%0903!  +  rts
.%0904!
.%0905!  ;*** positionLine( sortline ) : sortQ=prev, sortP=next
.%0906!
```

This routine searches for the correct position in the line list to insert the new line, and returns sortQ and sortP to straddle the new line position. Note that this routine causes the list to be in reverse order as discussed earlier.

```
.%0907!  positionLine = *
```

Set P to head and Q to Null.

```
.%0908!     ldx #2
.%0909!  -  lda #bkNull
.%0910!     sta sortQ,x
.%0911!     lda sorthead,x
.%0912!     sta sortP,x
.%0913!     dex
.%0914!     bpl -
.%0915!
.%0916!     positionSearch = *
```

This routine breaks out if the current line pointer is Null. Otherwise, it fetches the current line pointer (sortP) into the cmpline buffer and calls the string compare routine. If the new line read in from the file is greater than or equal to the current line already in the list, the search kicks out. The "bcs" on line 924 controls the order of the sort. Otherwise, the P and Q pointers are updated in the usual way and the search continues.

```
.%0917!     lda sortP+2
.%0918!     cmp #bkNull
.%0919!     beq positionExit
.%0920!     lda #<cmpbuf
.%0921!     ldy #>cmpbuf
.%0922!     jsr fetchline
.%0923!     jsr sortGTcmp
.%0924!     bcs positionExit    ;** controls sort order
.%0925!     ldx #2
.%0926!  -  lda sortP,x
.%0927!     sta sortQ,x
.%0928!     lda cmpbuf,x
.%0929!     sta sortP,x
.%0930!     dex
.%0931!     bpl -
.%0932!     bmi positionSearch
.%0933!
.%0934!     positionExit = *
```

At this point, sortP and sortQ straddle the position to put the new line, so we return.

```
.%0935!     rts
.%0936!
.%0937!  ;*** storeline( sortline )    {between sortQ and sortP}
.%0938!
```

This routine actually stores the new line read in between the sortQ and sortP lines.

```
.%0939!  storeline = *
```

First, space for the new line is allocated.

```
.%0940!     lda sortbuflen
.%0941!     ldy #0
.%0942!     jsr malloc
.%0943!     bcc +
.%0944!     rts
```

And the new line's next pointer is set to point to sortP.

```
.%0945!  +  ldx #2
.%0946!  -  lda sortP,x
.%0947!     sta sortbuf,x
.%0948!     dex
.%0949!     bpl -
```

And the new line is stashed out to main memory.

```
.%0950!     lda #<sortbuf
.%0951!     ldy #>sortbuf
.%0952!     sta zw1
.%0953!     sty zw1+1
.%0954!     lda sortbuflen
.%0955!     ldy #0
.%0956!     jsr stash
```

Now all that is left to is make the previous line record (sortQ) point to the new line record.

```
.%0957!     lda sortQ+2
.%0958!     cmp #bkNull
.%0959!     beq storelineFirst
```

If there is an actual previous line, the new line pointer is written out over the next line pointer in its header.

```
.%0960!     ldx #2
.%0961!  -  lda zp1,x
.%0962!     ldy sortQ,x
.%0963!     sta sortQ,x
.%0964!     sty zp1,x
.%0965!     dex
.%0966!     bpl -
.%0967!     ldx #sortQ
.%0968!     ldy #3
.%0969!     jsr zpstore
.%0970!     clc
.%0971!     rts
.%0972!
```

If there is no actual previous line, then the line list head pointer is set to

point to the new line (which is now the first line on the list).

```
.%0973!     storelineFirst = *
.%0974!     ldx #2
.%0975!  -  lda zp1,x
.%0976!     sta sorthead,x
.%0977!     dex
.%0978!     bpl -
.%0979!     clc
.%0980!     rts
.%0981!
.%0982!  ;*** readfile()
.%0983!
```

This routine reads in the file and puts the lines into their correct sorted positions as it is reading.

```
.%0984!  readfile = *
```

Clear the line list by setting the head pointer to Null.

```
.%0985!     ldx #2
.%0986!     lda #bkNull
.%0987!  -  sta sorthead,x
.%0988!     dex
.%0989!     bpl -
```

Set the EOF flag to 0 and set the current input channel to logical file #1 which is assumed to be opened before the sort utility is invoked.

```
.%0990!     lda #0
.%0991!     sta eofstat
.%0992!     ldx #1
.%0993!     jsr kernelChkin
.%0994!     bcs readExit
```

Until EOF, read the new line, find the position in the line list, store it, print out a "." to indicate to the user that another line has been processed, and repeat. Exit on EOF.

```
.%0995!  -  jsr getline
.%0996!     bcs readExit
.%0997!     jsr positionLine
.%0998!     jsr storeline
.%0999!     bcs readExit
.%1000!     lda #"."
.%1001!     jsr echoStatus
.%1002!     jmp -
.%1003!
.%1004!     readExit = *
.%1005!     rts
.%1006!
.%1007!  ;*** writefile()
.%1008!
```

This routine writes the line list out to logical file number 2 which is assumed to be opened before the sort utility is invoked. This routine follows the standard structure for processing a linked list.

```
.%1009!  writefile = *
.%1010!     ldx #2
.%1011!  -  lda sorthead,x
.%1012!     sta sortP,x
.%1013!     dex
.%1014!     bpl -
```

```
.%1015!     ldx #2
.%1016!     jsr kernelChkout
.%1017!
.%1018!     writeLine = *
.%1019!     lda sortP+2
.%1020!     cmp #bkNull
.%1021!     beq writeExit
.%1022!     lda #<sortbuf
.%1023!     ldy #>sortbuf
.%1024!     jsr fetchline
.%1025!     jsr putline
.%1026!     ldx #2
.%1027!  -  lda sortbuf,x
.%1028!     sta sortP,x
.%1029!     dex
.%1030!     bpl -
.%1031!     jmp writeLine
.%1032!
.%1033!     writeExit = *
.%1034!     jsr kernelClrchn
.%1035!     rts
.%1036!
.%1037!  ;*** reverseList()
.%1038!
```

This routine will reverse the order of the line list. Starting from the head line, each line is extracted and is made to point to the previous line extracted. No data actually has to be moved around; only the headers of the line records have to be changed.

```
.%1039!  reverseFile = *
.%1040!     ldx #2
.%1041!  -  lda sorthead,x
.%1042!     sta zp1,x
.%1043!     lda #bkNull
.%1044!     sta sorthead,x
.%1045!     dex
.%1046!     bpl -
.%1047!
.%1048!     reverseLine = *
.%1049!     lda zp1+2
.%1050!     cmp #bkNull
.%1051!     beq reverseExit
```

Fetch the pointer from the current line into sortP and then replace it with the value at sorthead (the previous line altered).

```
.%1052!     ldx #sortP
.%1053!     ldy #3
.%1054!     jsr zpload
.%1055!     ldx #sorthead
.%1056!     ldy #3
.%1057!     jsr zpstore
```

Make sorthead point to the current line, and then go to the next line whose pointer was extracted from the current line (before the current line was changed).

```
.%1058!     ldx #2
.%1059!  -  lda zp1,x
.%1060!     sta sorthead,x
.%1061!     lda sortP,x
.%1062!     sta zp1,x
.%1063!     dex
.%1064!     bpl -
```

```
.%1065!     bmi reverseLine
.%1066!
.%1067!     reverseExit = *
.%1068!     rts
.%1069!
.%1070!  ;*** freefile()
.%1071!
```

This routine scans through the lines in the line list and deallocates each line record.

```
.%1072!  freefile = *
.%1073!     ldx #2
.%1074!  -  lda sorthead,x
.%1075!     sta zp1,x
.%1076!     dex
.%1077!     bpl -
.%1078!
.%1079!     freeLine = *
.%1080!     lda zp1+2
.%1081!     cmp #bkNull
.%1082!     bne +
.%1083!     rts
.%1084!  +  ldx #header
.%1085!     ldy #4
.%1086!     jsr zpload
.%1087!     lda header+3
.%1088!     ldy #0
.%1089!     jsr free
.%1090!     ldx #2
.%1091!  -  lda header,x
.%1092!     sta zp1,x
.%1093!     dex
.%1094!     bpl -
.%1095!     jmp freeLine
.%1096!
.%1097!  ;*** main()
.%1098!
```

Finally! The main routine sets the sort key column and calls each of the subroutines for the different phases of the sort and prints out a letter indicating what the program is currently doing.

```
.%1099!  main = *
.%1100!     cmp #1
.%1101!     bcc +
.%1102!     sbc #1
.%1103!  +  sta sortColumn
.%1104!     lda #"s"
.%1105!     jsr echoStatus
.%1106!     jsr startup
.%1107!     lda #"r"
.%1108!     jsr echoStatus
.%1109!     jsr readfile
.%1110!     lda #"v"
.%1111!     jsr echoStatus
.%1112!     jsr reverseFile
.%1113!     lda #"w"
.%1114!     jsr echoStatus
.%1115!     jsr writefile
.%1116!     lda #"f"
.%1117!     jsr echoStatus
.%1118!     jsr freefile
.%1119!     lda #"x"
.%1120!     jsr echoStatus
```

```
.%1121!     jsr shutdown
.%1122!     lda #13
.%1123!     jsr echoStatus
```

It returns with .A set to zero in case the user calls sort again and forgets to specify a value for the sorting column using the BASIC SYS statement.

```
.%1124!     lda #0
.%1125!     rts
```

---

5. FUTURE ENHANCEMENTS

This dynamic memory allocation package does not support expanded internal memory (as specified in Twin Cities-128 Magazine) or RamLink memory. I am planning to modify the memory allocation in the Zed-128 program to support both of these kinds of memory. The extra internal memory banks would be accessed in a similar manner as RAM1 is, except that I will need to have some special bank numbers for them, since they cannot be handled in exactly the same way as RAM0 and RAM1. I will also have to modify that other MMU register in order to select which real banks show up in the RAM2 and RAM3 positons.

The memory inside a RamLink can be accessed in a similar way to how memory is accessed in an REU. One big difference is that the layout of the storage in a RamLink is actually organized. A RamLink (and a RamDrive I assume) can have up to 31 partitons of various types. I am thinking that to sniff a RamLink, the package will check to see if you have a RamLink and will then check to see if you have partiton number 31 set up as a "foreign" mode partition with the name "swap". If so, the package will ask the RL-DOS for the start address and length of the partition and will then use the RamLink memory instead of an REU. This makes sense since an REU can be made to be part of the RamLink and since you can get a lot more memory in a RamLink than I have ever heard of in an expanded REU. I personally have an 8 Meg RamLink and I have set aside a 1 Meg partition for the swap space. Now I just have to write the software to use it.

These additional types of memory can be seemlessly implemented into this package and the usage will be compeletely transparent to the user and to the higher level routines.

Also, although I have not attempted to do this, the code presented here could be ported to the Commodore 64. The common code routines would be removed since the 64 has only one internal bank, and instead of using the MMU to select RAM0, you would store into the processor I/O port to select the bare internal RAM. (You would also have to worry about interrupts happening while you are accessing this memory). All of the higher level code above the zpload, zpstore, fetch and stash routines would (probably) stay (pretty much) the same, since they call the lower level routines to do the actual machine-specific grunt work.

If you have any questions or comments about this article, feel free to drop me a line.

---

6. UUENCODED BINARIES

Here are the BASIC program and the machine language subroutines for the sorting utility. They are in uuencoded form and you will probably have to extract them into separate files before you uudecode them. Enjoy!

```
begin 640 sort
M`1PF'`$`222R(DE.4%541DE,12Y46%0B(#H@242R."`Z(%-&LC$`11P"`$\D
MLB)/5510551&24Q%+E185"(@.B!/1+(X`$L<`P`Z`&8<9`"9(DQ/041)3D<@
```

```
M4T]25"Y"24XN+BXB``'`<;@#^`B`Q-0"'''@`_A$B4T]25"Y"24XB+%4H240I
M`*4<@@"9(E-#4D%40TA)3D<@3TQ$($9)3$4N+BXB`+4<C`#R*$\D*2Q5*$]$
M*0#'')8`F2)33U)424Y'+BXN(@#;'*``GS$L240L,BPB,#HBJDDD`/8<J@"?
M,BQ/1"PS+"(P.B*J3R2J(BQ3+%<B``D=M`">(-$H(C$S,#`B*2Q31@`0';X`
=H#(`%QW(`*`Q`"@=T@"9(D9)3DE32$5$(2(````H
`
end

begin 640 sort.bin
M`!-,I!E,(Q-,-A-,R!-,_A-,#11,;11,`Q9,R18``````````*D`2"BI#HT`
M_R!($R#\%""G%6"B`*D`G0`"Z.!RT/BI`(T`_V"B`+U6$YT``NC@<I#U8*7\
MC0#_A/V@`+'ZE0#HR,3]D/:I#HT`_V"E_(T`_X3]H`"U`)'ZZ,C$_9#VJ0Z-
M`/]@B/`-C0+_L?J-`?^1_HC0\XT"_['ZC0'_D?ZI#HT`_V"(\`V-`?^Q_HT"
M_Y'ZB-#SC0'_L?Z-`O^1^JD.C0#_8*7\,`-,``*,!]^@D8T&WXX"WZT'U8T#
MWZ7ZC03?I?N-!=^I`(T(WZTPT*(`CC#0C`'?C3#08*7\,`-,&0*,!]^@D$S4
M$Z;\$`6BD4S-%,``T"*HT`%@X#_P`TPR`HX`_XCP!['ZD?Z(T/FQ^I'^J0Z-
M`/]@A8"$@:7[A8*E_X6#I8'P#Z;\H``@'A3F^^;_QH'0\:2`\`6F_"`>%*6"
MA?NE@X7_8*;\$`6BD$S-%,``T"*HT`%@X#_P`TQ2`HX`_XCP!['^D?J(T/FQ
M_I'ZJ0Z-`/]@A8"$@:7_A8*E^X6#I8'P#Z;\H``@?A3F^^;_QH'0\:2`\`6F
M_"!^%*6"A?^E@X7[8(T'WXP(WZ7^I/^-`M^,`]^E^J3[C03?C`7?I?R-!M^L
M,-"I`(TPT(X!WXPPT&!_J3^-`/^B"KT`WYT`"\H0]ZD.C0#_J0"-"=^-"M^M
M`-^B`KVA%96`RA#XJ0"-'!.I`**`A?J%^X;\(%P5L`?N'!/F_-#TK1P3T!6I
M/XT`_Z(*O0`+G0#?RA#WJ0Z-`/]@K1P3S?L4D`%@I?R%@Z*`H`0@#!,@A16P
M#Z7\2*F`A?R%@R"%%6B%_&"I`(6$A8>BA*`$(`D3H@.UA-6`T`7*$/<88#A@
M4LE$0`3[H@*I`)T@$ZG_G1T3RA#S.*T2$O`!&*T3$NVD%:BMI!6B/R#X%:VE
M%:RF%:)_(/@5J0"%CZ6/S1P3L!4)@(7\J0"%^H7[J?B@_R`8$^:/T.1@A?N&
M_*D`A?I,&!,8:0>0`<@I^(6%A(:B`KT=$Y7ZJ?^5A\H0]*7\R?_0#ZG_A?J%
M^X7\J0&-&Q,X8**`H`4@"1.E@\6%I83EAK`0H@*U^I6'M8"5^LH0]4P=%CBM
M(!/EA8T@$ZTA$^6&C2$3L`/.(A.E@\6%T`:EA.6&\",XI8/EA86#I83EAH6$
MHH"@!2`,$QBE^F6#A?JE^V6$A?L88*6)R?_0#*("M8"='1/*$/@88*("M?JT
MAY6'E/K*$/6B@*`#(`P3H@*UAY7ZRA#Y&&`8:0>0`<@I^(6%A(:B`K7ZE8>]
M'1.5^JG_E8K*$/"E_,G_\"BE_,6)D"+0"J7ZQ8>E^^6(L!:B@*`#(`D3H@*U
M^I6*M8"5^LH0]3#2H@*UBI7ZRA#YI?S)_]`:H@*]'1.5@+6'G1T3RA#SI86D
MAH6#A(1,A!>B@*`%(`D3I?S%B=`J&*7Z98.JI?MEA,6(T!SDA]`8&*6#986%
M@Z6$98:%A*("M8J5A\H0^3`/HH>@`R`,$Z6%I(:%@X2$I8G%@M`S&*6'98.J
MI8AEA,6!T"7D@-`AH@*U@)7ZRA#YI81(I8-(HH"@!2`)$QAH98.%@VAEA(6$
MH@*UAY7ZRA#YHH"@!2`,$QBM(!-EA8T@$ZTA$V6&C2$3D`/N(A,88*``)`)P
M)"#/_[`?F00+R*:0A@+`\+`%R0W0YXBI`)D$"QB8:06-`PL88*E`A0+``-#J
M.&"@`+D$"_`&(-+_R-#UJ0U,TO^%_H3_H@*U!I7ZRA#YH@R@!"`)$Z4/H`!,
M#Q.B`!BM_PAI!<T#"Y`!Z,T#")`"Z.B*\`/)`F"L_PBY!`O9!`C0!\D`\`/(
MT/%@H@*I_Y4)M0.5!LH0]:4(R?_P'*D`H`@@*Q@@1ABP$*("M0:5";T`")4&
MRA#T,-Y@K0,+H``@%1.0`6"B`K4&G0`+RA#XJ0"@"X7^A/^M`PN@`"`2$Z4+
MR?_P%J("M?JT"94)E/K*$/6B":`#(`P3&&"B`K7ZE0/*$/D88*("J?^5`\H0
M^ZD`A0*B`2#&_[`5(.47L!`@=!@@I!BP"*DN(-+_3`098*("M0.5!LH0^:("
M(,G_I0C)__`7J0"@"R`K&"`9&*("O0`+E0;*$/A,*!D@S/]@H@*U`Y7ZJ?^5
M`\H0]:7\R?_P':(&H`,@"1.B`Z`#(`P3H@*U^I4#M0:5^LH0]3#=8*("M0.5
M^LH0^:7\R?_0`6"B#*`$(`D3I0^@`"`8$Z("M0R5^LH0^4R#&<D!D`+I`8W_
M"*E3(-+_(`,3J5(@TO\@\!BI5B#2_R!)&:E7(-+_(!H9J48@TO\@>AFI6"#2
,_R`&$ZD-(-+_J0!@
`
end
```

==========

Next Issue: (Hopefully!!! :-])

The 1351 Mouse Demystified

An indepth look at how the 1351 mouse operates and how to access it within your ML programs, in addition to a BASIC driver for the 80 column screen.

ML Tutor - Part 3

In this edition we take a look at reading and writing commands to the disk drive, including reading the disk directory. This article will also parallel the discussion of the C=128 and C=64 KERNAL jump tables of available routines.

KERNAL 64/128

The C=128 and C=64 jump table that points to many valuable system routines is listed and discussed with example applications and how to use them.

Bursting your 128

This article will examine the routines and mysteries about how to use Burst commands on the 1571 and 1581, including the Fastload utility and the block Read and Write calls.

==========

END of C= Hacking Issue 2.

==========