Volume 2 Issue 4 $1.50 December 1985

SUPER 99 MONTHLY

GRAM KRACKER™

GRAM Kracker™:
Our First Report

STANDARD: 1A 9B 15A

Millers Graphics' GRAM Kracker™ (hereinafter referred to as "GK"), a new hardware/software package, was delivered to users as promised shortly before Christmas. Our initial reaction is that GK is the most exciting new product to come along since the 99/4A itself.

GK completely opens the architecture of the 99/4A. For the first time, 99/4A owners can modify resident council GROM (operating system and console BASIC). While making program changes to the council is certainly a project for experienced programmers, information on such modifications can easily be conveyed in an easy to implement format to almost anyone who has at least minimal computer keyboard experience. (Note: Changes to the council and certain other operations require optional RAM at a cost of $13.50 U.S. and may also require memory expansion and/or disk).

GK allows the user to back up any module (except the Milton-Bradley MBX series, which have a much more complex architecture than other modules) to disk, cassette, RAM disk or hard disk. Modules can also be modified, including changes such as changing the default colors, changing printer defaults (for example RS-232 to PIO), changing the default baud rate of TEII from 110/300 to 300/1200 and many other changes made directly to the modules' programming (no load interrupt required). With the device, approximately 15 modules can be saved on one DS/DD disk (the exact number depends on the amount of memory present in the particular modules, with Extended BASIC requiring 170 disk sectors, while TI-Writer can be saved in only 34 sectors). Users can also write additional code for a module or their own module by using TI's proprietary GPL language (user "modules" could make use of BASIC and Assembly as well). GK comes with 56K of battery backed up cartridge RAM and GRAM, expandable to 80K, and includes its own module port for saving of modules plus 8K of pre-programming for on-line saving, loading and editing of modules and room for optional console GRAM's. The device allows a menu with more than one module on it, plugs into the module port and is easy to install and operate.

Our first project with GK was, of course, to save all of our modules to disk, thereby eliminating worries of module failures. Additionally, we have noted a complete elimination of the frequent system lockups we were previously experiencing while using modules. The gold-plated GROM port connector on the GK is a much more reliable connection than is available on modules and the connector is easier to access for occassional cleaning.

In order to meet their promised shipping schedule, Millers Graphics departed from their typical procedure and released the device with a temporary manual and utility disk. Though the material furnished does get the user started quite nicely, Millers Graphics offers assurances that the final manual and disk will be much more thorough. So, for now, we'll try to help everyone who already has a GK get started. GK was designed for system and module exploration and a lot of information on modules has already come to light! Though GK is easy to use, modules come in a variety of "flavors", so, for those who have or soon will have a GK, we'll take a look at some of the peculiarities of how the modules work and how to handle the variety of available modules from the GK environment.

TEII

The GK manual describes changes to the TEII module from 110/300 to 300/1200 baud. While the instructions worked fine for our TEII, not all of the TE II's were the same! (Note: Some modules saved some files as D/F 80, while the more common modules save all text files as D/V 80. Some modules were labeled TEII, while others were labeled TE2. Also, some differences have been noted in the way different versions output to printer.). If you are unable to find the memory contents as described in the manual, try a search beginning about 20 bytes before the address suggested in the manual (Howie Rosenberg reports finding the values described as being at g6F40 at g6F38, for example. However, the number of bytes off from the manual example floats, so you must search each address individually.). When doing a search, be sure to follow the manual's instructions carefully, placing your cursor on the last character in the search string (not one character beyond!). Also, in order to access speech emanating from a TIBBS™ BBS, the Loader must be Off while using the TEII module (We have found no way to access speech emanating from a TIBBS™ at 1200 baud, so the lack of speech at 1200 is not

-->

related to 6K. The SPEAK command, <CTRL> <1>, will work with the Loader On or Off.). The Loader Off is required to receive TIBBS℠ speech due to the speech routine using a BASIC PAB (see the BASIC modules section below).

Auto Execute Modules

Modules that auto execute, such as Plato℠, do not allow access to a power-up menu from which 6K could be selected (except, if you have a CorComp Disk Controller, that controller always takes control on power-up and does therefore provide a proper menu). Millers Graphics, upon being questioned about this matter, created a fix within hours and will provide the information in the final manual. The fix is a 3 byte patch to GRAM 0 (a "console" GRAM).

RAM Modules

To use Mini Memory, the module should be saved and loaded as normal, except that, after loading, the Write/Protect switch on 6K should be switched to Bank 1 to allow use of RAM, as RAM is available from the module. The "W/P" environment is the normal environment for nearly all modules, with the exceptions being Mini Memory and mimicking a "Super Cart" (by loading E/A and enabling Bank 1).

Bank-swapped Modules

Bank-swapped modules, such as Extended BASIC and some of the Atari modules (Pole Position℠ is one), must be operated in strict accordance with 6K instructions. That is, the Write/Protect switch must be on "W/P". The bank swapping is performed by "writing" to ROM (you can't actually write to ROM), so if you enable Bank 1 or Bank 2, an actual write will occur, no swap will occur and you won't get very far!

BASIC Modules

A few modules are written in BASIC or both GPL and BASIC and therefore require that the Loader switch be in the Loader Off position, unlike the Loader On that is recommended for other modules. The BASIC modules include Personal Record Keeping, Securities Analysis, Statistics and Personal Report Generator. There may be other modules in this category.

Early LOGO Learning Fun

The Early LOGO Learning Fun module is not compatible with the CorComp Disk Controller. The problem is that the module has an Application Program name length of >00. When the CorComp DSR goes through the module looking for Application names for the menu it starts moving them out and then it decrements the name length counter. >00 decremented is >FF or 255 bytes, which causes a mess on the title screen (names can't be that long). To correct this, SAVE the module to disk using cassette with the P.E.B. off or use a TI or MYARC disk controller and then LOAD it back into the 6K. Select the 6K's Memory Editor and change the byte at g6047 to >01 and resave the module.

$$$$

The Utilities disk instructions mention using the file CHARA1, but that file exists in its original form from TI, an update from TI and several implementations from various programmers. Because only one of the versions was intended to work and so many versions exist, Millers Graphics will be releasing a completely new character set on the final Utilities disk.

For the file XBCALLS, the CALL CLSALL (to close all open files) was omitted from the temporary disk (a few disks included a non-functioning version) and will appear on the final disk. Other CALL's will also be available, including a CALL CAT to catalog any disk (floppy, RAM or hard disks) from imperative mode. You may want to make note that the CALL's load into address gD800. CALL NEW, CALL BYE, CALL CLOCK and CALL CLKOFF work from either imperative (immediate) mode or from within a program, but you must also do a CALL LINK("SETCLK","000000") to actuate the interrupt driven CALL CLOCK routine. "000000" can be set to the current time or used as an elapsed timer. The clock appears at the upper right corner of the screen (not the left corner as stated in the instructions). As the routines are not in Low Memory, CALL INIT does not affect the clock.

We do not recommend loading modules through the MYARC MEXP-1 RAM Disk. Doing so sometimes yields a module that does not function properly. Speculation is that RESET from either 6K or a Navarone Cartridge Expander℠ affects random bytes on the RAM Disk, but no definite information is available as to why the files do not load properly, only that they don't. One certainty is that this problem has been noted outside of the 6K environment.

Another note on using the MEXP-1 card from MYARC is that when in the 6K Editor, spooling print goes to the MEXP-1 buffer, but does not print until the 6K Editor has been exited. Any time interrupts are disabled, as in the 6K Editor, the spooler pauses.

Well, we've tried to answer most of the questions that have surfaced so far on CIS, BBS's and through Millers Graphics. Modifications for some of the module peculiarities will likely be found on the TI FORUM on CIS before release of the final disk. Users have been so eager to learn about the powers of 6K that the TI FORUM has titled Section 4 as Kracker Hackers, for the exlusive use of 6K owners. If you have questions, Craig and Sue Miller at Millers Graphics are always eager to offer full support for their products, so be sure to contact them so you'll be sure to be happy with your 6K!

6ROM Modifications: Tips from Millers Graphics

STANDARD: 1A 9B 15A

Thanks to Craig Miller of Millers Graphics for granting us permission to print the following modifications, which were uploaded to CIS by Millers Graphics.

<><><><><><>

** Extended BASIC CALL INIT correction

Presently, the CALL INIT loads >600 bytes starting at >2000 in Low Expansion Memory, but only >4F3 bytes need to be moved. Because of this, some routines that were loaded into Low Expansion Memory get overwritten. The patch corrects this situation.

1) With Extended BASIC loaded into 6K, select 5 (Memory Editor) from the 6K Menu.

2) Press FCTN = HEX, FCTN 1 for GRAM Memory Window and the FCTN 5 for SEARCH *(ed.: SEARCH is a really powerful tool!)*

-->

3) Type in C200 for the START address and C300 for the FINISH address. Press FCTN 9 to put the cursor in the Search String Input field and type in 31 06 00. Press FCTN S (left arrow) to place the cursor on top of the last byte to search for and press enter.

4) Turn off Write Protection, press FCTN 5 to leave SEARCH and press FCTN 9 to put the cursor in the Memory Window. Now replace 31 06 00 with 31 04 F3.

5) Restore Write Protect, return to GK menu and resave the module.

6) CALL INIT will now work a little quicker and it will not move unneccessary bytes out to Low Memory Expansion.

<><><><><><>

SLASHING THE ZERO IN AN ALTERNATE OPERATING SYSTEM

This modification requires an 80K GK with GROM 0, the TI Operating System, copied into GRAM 0 so it can be modified.

1) With an Alternate Operating System loaded into the Optional GRAM 0 (80K GK), turn on GRAM 0 and select GK from the menu.

2) Select 5 (Memory Editor) from the GK Menu. Press FCTN = for hex, FCTN 1 for GRAM Memory and then press FCTN 5 (SEARCH).

3) Type in 0000 for START address and 1000 for the FINISH address. Press FCTN 9 to place the cursor in the Search String Input area and type in 00 38 44 44 44 44 44 38 and then press FCTN S (left arrow) to put the cursor on the last byte to search for and press enter.

4) For most Operating Systems the hex string will be found at >0723. Now press FCTN 5 to leave SEARCH and then press FCTN 9 to put the cursor in the memory window. Turn off Write Protection and type in the following bytes to change the zero: 00 38 44 4C 54 64 44 38.

5) Now turn on Write Protection, press CTRL = to leave the Memory Editor and then resave the modified Operating System (GRAM 0).

<><><><><><>
Extended BASIC Auto-Boot (DSK1.LOAD) Bypass Patch

1) Load Extended BASIC into the GK.

2) From the GK Menu, select 5 (Memory Editor). Then press FCTN = for hex, FCTN 1 for the GRAM Memory Window and then press FCTN 5 for SEARCH.

3) Type in >6300 for the START address and >6400 for the FINISH address. Press FCTN 9 to put the cursor in the Search String Input area and type in 86 A3 71 and then press FCTN S (left arrow) to put the cursor on the last byte to search for. Next press ENTER to start the Search.

4) For most Extended BASIC modules the hex string will be found at >63CD. We'll call that "address A". Now press FCTN 5 to leave SEARCH and then press FCTN 9 to put the cursor in the Memory Window. Turn off the Write Protect (turn it to Bank 1). Now change the first two bytes (86 A3) to 5B 00. This is a BRANCH ON RESET to >7800 instruction.

5) Press FCTN 9 and change the Memory Window to g7800. You will see garbage here (unless you have previously put something in this space!!). The GROM's are only 6K in length, so the bytes in the last 2K are "garbage wrap around" read by the GK Save routine. So, it's a good area for adding routines to your module.

6) Press FCTN 9 to put the cursor in the Memory Window and at the g7800 memory location, put in the following code:

```
86 A3 71   CLR V@>371  Clear Auto  Load
                       needed flag
03         SCAN        Scan the keyboard
D6 75 20   CEQ >20,@>8375
                       Is the Space Bar
                       pressed?
```

[Now take your "address A" and add 6 to it]
[>63CD + 6 = >63D3]

```
63D3       BS          "address A" plus
                       6 bytes.  Yes!
                       (Branch on Set)
```

[Take your "address A", add 3 to it and replace the first digit with 4]
[>63CD + 3 = >63D0 .....change it to 43D0]

```
43D0       BR          "address A" plus
                       3 bytes.  No!
                       (Branch on Reset)
```

7) For a module with a >63CD "address A" your Memory Window should now look like this:

```
g7800
================================
B6 A3 71 03 D6 75 20 63 D3 43 D0 xx
xx xx xx xx xx xx xx xx xx xx xx xx
```

xx = don't care

8) Now restore the Write Protect, return to GK Menu and resave your module.

9) Now when you select Extended BASIC, you can bypass the auto-load command by holding down the space bar! (No more DSK1.LOAD search)

<><><><><><>

The above modifications by D.C. Warren (Millers Graphics programmer) represent only the "tip of the iceberg"! An entirely new world of programming has now been opened to 99/4A owners. While we feel that the 99/4A council and modules have served us well for a long time, much of the code was written over six years ago! Many great ideas have come down the pike during these years, but now, for the first time, we actually have the means to change the original code. With GK, 1986 will surely be the most exciting year ever for 99/4A owners!

EXTENDED BASIC

Linking Assembly to Extended BASIC -- A Tutorial for Beginners

STANDARD: 1A 2XB EA 4B 5A 6B 7B 9A

A couple of months ago one of our I Wish I Had requests was for a way to simply build a screen that would display quickly. Our suggestion was to use a program called MENGEN, available on the CIS TI FORUM. While MENGEN is a very useful tool, its use assumes having a screen already displayed and the data stored includes more than just text (colors, etc.). Another excellent screen builder program we've seen is Barry Traver's DISPLAY/AT program. DISPLAY/AT offers the advantage of using TI-Writer to do full screen editing of text, generating a program with DISPLAY AT's (hence the name) in MERGE format. We decided to combine the basic ideas of MENGEN and DISPLAY/AT and detail what we are doing in our Assembly Language

-->

program for the benefit of those who are new to Assembly and/or to linking Extended BASIC to Assembly.

We've heard from many of you that you have no interest in learning Assembly. No problem! Our program requires almost no knowledge of Assembly Language! For those who do want to understand what we are doing, the program is fully annotated in hopes you will completely understand the process.

The program that follows will allow you to move your cursor around freely on a simulated Extended BASIC "screen" (the area within the single quote marks, 32 characters wide by 24 lines long) from the Editor/Assembler Editor. The only other information that you must understand is that the DEF that we have called START and the label START must be changed to whatever you would like (up to six characters long) to enable you to load several screens into Low Memory at once (labels and DEF's must match for each screen used). As this program is intended primarily for beginners, it uses a fairly traditional approach to Assembly and is offered for its simplicity of use. It is certainly not the only method of coding and we don't claim it to be the best method for advanced programmers.

```
        DEF  START          CHANGE 'START' HERE AND BELOW AS DESIRED---
*                           DEF IS STORED IN THE DEF TABLE AND IS     |
*                           REQUIRED TO LINK FROM EXTENDED BASIC      |
VSBW    EQU  >2020          POINTER TO UTILITY WORKSPACE FOR VIDEO    |
*                           SINGLE BYTE WRITE.  NOW VSBW EQUATES (IS  |
*                           ANOTHER NAME FOR) >2020                   |
GPLWS   EQU  >83E0          THE WORKSPACE FOR GPL IS ALWAYS AT >83E0  |
*                           NOW GPLWS EQUATES TO >83E0                |
STATUS  EQU  >837C          THE GPL STATUS BYTE IS ALWAYS AT >837C    |
*                           NOW STATUS EQUATES TO >837C               |
SAV11   BSS  2              THE BLOCK STARTING SPACE RESERVES 2 BYTES |
*                           FOR OUR SAVING OF REGISTER 11, WHICH      |
*                           TELLS BASIC WHERE TO RETURN TO            |
MYWS    BSS  >20            OUR WORKSPACE WILL BE 32 BYTES LONG (2    |
*                           BYTES PER REGISTER X 16 REGISTERS)        |
*                           (32=>20).  REGISTERS ARE MEMORY WORDS     |
*                           THAT SERVE A SPECIFIC PURPOSE.            |
*                           GENERALLY, IN ORDER TO ACT UPON A VALUE   |
*                           (SUCH AS TO ADD), THE VALUE MUST BE IN OR |
*                           PASSED TO OR THROUGH A REGISTER.          |
*                           REGISTERS ARE ALSO FREQUENTLY USED TO     |
*                           REFERENCE ADDRESSES, EITHER DIRECTLY OR   |
*                           INDIRECTLY.                               |
START   MOV  R11,@SAV11     CHANGE 'START' AS DESIRED-----------------
*                           STORE THE VALUE IN REGISTER 11 SO WE CAN
*                           RETURN TO BASIC PROPERLY
        LWPI MYWS           LOAD OUR WORKSPACE POINTER AT MYWS
        LI   R0,0           OUR BEGINNING SCREEN POSITION WILL BE 0,
*                           THE UPPER LEFT CORNER OF THE SCREEN
        LI   R2,L1          WE'LL WRITE WHAT'S AT LABEL L1 ------------
        LI   R3,768         WE'LL WRITE 768 BYTES (ENTIRE SCREEN)      |
        BL   @WRITE         BRANCH AND LINK TO ROUTINE 'WRITE' -----   |
RETURN  LWPI GPLWS          (NOTE:  LABEL 'RETURN' NOT REQUIRED)    |  |
*                           PREVIOUS REGISTER VALUES WILL NOT BE    |  |
*                           IN USE, WE'RE MOVING TO A NEW           |  |
*                           WORKSPACE AFTER RETURNING FROM 'WRITE'  |  |
        MOV  @SAV11,R11     RETRIEVE THE RETURN TO BASIC POINTER    |  |
        CLR  @STATUS        CLEAR THE STATUS BYTE TO                |  |
*                           CLEAR ERRORS UPON RETURN TO BASIC       |  |
        RT                  RETURN TO BASIC                         |  |
WRITE   MOVB *R2+,R1        MOVE THE CONTENTS OF THE HIGH  ---------  |
*                           BYTE OF REGISTER 2 (ON THE 1ST            |
*                           PASS IT IS THE FIRST CHARACTER AT         |
*                           'L1') TO REGISTER 1 AND                   |
```

-->

```
*                       INCREMENT R2 BY ONE (BYTE
*                       INSTRUCTIONS INCREMENT BY 1,
*                       WORD INSTRUCTIONS BY 2)
        AI    R1,>6000  ADD >6000 TO THE VALUE IN R1.
*                       WE'RE WORKING WITH BYTES, SO WE ARE
*                       FUNCTIONALLY ADDING >60 TO THE HIGH BYTE
*                       OF R1.  IN LINKING TO BASIC AN OFFSET
*                       OF >60 (DECIMAL 96) BETWEEN MEMORY AND
*                       VDP MEMORY IS REQUIRED.
        BLWP @VSBW      BRANCH AND LOAD THE WORKSPACE
*                       POINTER AT THE VIDEO SINGLE
*                       BYTE WRITE ROUTINE
*                       VSBW USES REGISTER 0 AS THE ADDRESS IN
*                       VDP AND THE MOST SIGNIFICANT BYTE OF
*                       REGISTER 1 FROM MYWS AS THE VALUE TO BE
*                       WRITTEN
        INC   R0        INCREMENT REGISTER 0 TO THE
*                       NEXT SCREEN POSITION
        DEC   R3        DECREMENT THE BYTE COUNTER
*                       IN REGISTER 3 SO WE'LL KNOW
*                       WHEN 768 BYTES HAVE BEEN WRITTEN
        JNE   WRITE     IF ALL 768 BYTES HAVE NOT BEEN
*                       WRITTEN, JUMP TO 'WRITE'
        RT              RETURN TO THE INSTRUCTION
*                       FOLLOWING WHERE WE BRANCHED FROM
L1      TEXT '                                ' ----------------------
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '     <-----
        TEXT '                                '     YOUR EXTENDED
        TEXT '                                '     BASIC SCREEN
        TEXT '                                '     WILL DISPLAY
        TEXT '                                '     THE CHARACTERS
        TEXT '                                '     YOU PUT WITHIN
        TEXT '                                '     THE SINGLE
        TEXT '                                '     QUOTE MARKS
        TEXT '                                '     <-----
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        TEXT '                                '
        END             TELLS THE ASSEMBLER WHERE OUR CODE ENDS
```

Now, the above routine must be entered in the Editor/Assembler Editor. Our comments (remarks on the right and lines that begin with an asterisk) are not required. The above routine is Source code. It must be saved to disk (ending your file name with "/S" will remind you that the file is a Source file). Use the Assembler to Assemble the file to Object code (end the file name with "/O"). The Assembler will check for Syntax-type errors. If no such errors have been keyed in, the Assembler will report 0 errors. If you have keyed the program in wrong, errors may be reported, but the absence of reported errors only means that the Assembler found your program syntactically correct. For instance, keying ">8000" instead of ">6000" will not produce code that works, but will also not generate an Assembler error!

Once the program is Assembled, it is ready to be linked to Extended BASIC.
-->

The following line will set up Low Memory for code (CALL INIT), load the Object file (CALL LOAD) and link to the code (CALL LINK):

```
100 CALL INIT :: CALL LOAD("
DSK1.SCREEN/S"):: CALL LINK(
"START")
```

In some situations, you may find it advantageous to place the INIT and LOAD in a separate program and RUN the main program from the LOAD program, so that if you break the main program you will not need to LOAD the object file again to RUN the program again. The code will remain in memory until you power off the memory card (for TI equipment, powering off the expansion box clears the memory card). Or, your program will no longer recognize your routines and memory will be available for loading more routines if you do another CALL INIT. If you want more than one program in memory at one time, use CALL INIT only before LOAD'ing the first program. At any point in your program that you want to display the screen, simply use the CALL LINK. Remember, the LINK is to your DEF and the entry point, so be sure those three match!

Now, why do all of the above? The displays are extremely fast. However, there is a trade-off. The LOAD time for the routines is very long unless you combine your A/L code with your Extended BASIC program and save it as one file (refer to Barry Traver's diskazine for a way to do that -- it's beyond the scope of this article). So, unless timing is critical (such as for games) or you will be using the same screens repeatedly, Extended BASIC's slow Loader will not be a good trade-off for the time you save in displaying the screens (unless you are in the habit of eating a sandwich while a program loads). However, there are some other advantages to using the program.

In using Assembly Language to display screens, the code for the displays loads into Low Memory, while your Extended BASIC program can only reside in High Memory. You are left with far more room for your Extended BASIC program! You'll cut your code from numerous DISPLAY AT's to only one CALL LINK! Not only will you have more room for other Extended BASIC code, but your Extended BASIC program will also appear less cluttered! It's great for program instructions! Finally, by using the Assembly program above, you will be able to create your screen using a full screen editor (just move the cursor around freely with the arrow keys)!

We hope you'll find this introduction to linking Assembly to Extended BASIC useful. We'll try to answer questions you might have and then assume you followed this article well enough to move on to more advanced projects.

Chaining Programs: Some Suggestions

STANDARD: 1A 2XB 4B 5A 6B 7B (o) 9A

One of the elementary rules of programming is that, upon leaving a program environment, the computer should be in the same state as prior to the execution of the program. In practice, perhaps no other convention is so widely disregarded. In Extended BASIC, the most common break from this fundamental approach is in leaving the character patterns and color sets in the state that happens to occur at the end of a program, thereby providing "garbage" screens for any program chained to the program (via a program line that uses RUN DSKx.y, where x is the drive number and y is the filename of the program to follow, in place of an END).

Of course, debugging another programmer's work is often a nightmare situation. An effort to go through hundreds of programs on dozens of disks to restore the color sets and character patterns would almost certainly be a cumbersome, if not vain, project. But, there is a rather simple solution.

-->

Rather than chain directly to another program, chaining to a loader program provides the opportunity to return VDP to its normal state outside the previous program environment. The code to do so would be very simple. A CALL CHARSET will restore the patterns for characters 32 to 95. A loop from 96 to 143 could easily restore the remainder of the characters. Of course, to obtain the normal character patterns, a CALL CHARPAT upon power up will reveal the hex codes that initially exist and a CALL CHAR within the 96 to 143 loop, interacting with DATA statements containing the hex codes, will restore the patterns. A loop of I=1 to 12 with CALL COLOR(I,2,1) will restore the color sets. And, a CALL SCREEN(8) will restore the screen color.

For those of you who do not have the time to key in the code for the characters from 96 to 143, Tigercub Software's Nuts and Bolts No. 1 has such a routine. The address is Tigercub Software, 156 Collingwood Ave., Columbus, OH 43213. If you write and enclose $1, you'll receive the latest catalog. Just in case someone doesn't know (where have you been?), Tigercub is run by Jim Peterson, a truly dedicated and honest 99'er. Each month, Jim sends out tips free to over 100 user groups! While on the topic of Tigercub and Nuts and Bolts, Tigercub has now released Nuts and Bolts No. 2, which is even better than the first disk of Extended BASIC subprograms!

And, of course, if you are into the speed of Assembly, writing the code to restore Extended BASIC's color and character information is a snap for experienced A/L programmers.

While many methods of restoring VDP information and tracking the next program to be RUN could be used, the important factor is that you do it!

FORTH

An Array of Strings

by Warren Agee

Last time we met, I covered how to handle the basic string in FORTH. I also stressed the importance of the count byte and how to move it along with the string. Now, we have graduated to the realm of string arrays, which is an entirely new mess with which to work.

Think of a string array as a super-long string. Since the character (or bytes) of a string sit sequentially in memory, it stands to reason that the elements in a string array do also. But the physical structure of an array must be forced by the programmer; maintaining an array is not automatically done. The structure is what we will discuss first.

Here is a possible string array:

```
-----------------------------------------
!3!C!A!T! !3!D!O!G! ! ! !4!B!I!R!D!5!P!O!O!C!H!
-----------------------------------------
```

This array has 4 elements: CAT, DOG, BIRD, and POOCH. Fine, right? No way! This is a mess! Each element in this array has a different length. Element #1 has 5 bytes, #2 has 7, #3 has 5, and #4 has 6. How in the world are you going to keep track of all this? You cannot! Elements in a string array must have a constant length. A much better way to construct the above array is like this:

```
-----------------------------------------
#3!C!A!T! ! #3!D!O!G! ! #4!B!I!R!D! #5!P!O!O!C!H!
-----------------------------------------
```

Note: From now on, the boundaries between elements will be pound (#) signs. Now each element is exactly 6 bytes long. Remember, the actual strings in an array can have variable length, but each element has to have the same maximum

-->

length. If the string is shorter than the maximum, then blanks will fill the excess space.

So much for structure and theory. How do we go about achieving this neat and tidy array? Well, start out with good ol' VARIABLE. Remember, arrays (string OR numeric) are just stretched-out variables. Think of a good name, let's say PETS. Now, decide how many elements this array is going to contain. Let's say 20. Now decide the maximum length of the elements. Let's keep it at 6. Remember to allow enough room for the count byte for each element! This sequence will then create our array:

```
0 VARIABLE PETS
60 ALLOT              (10 elements X 6 bytes each)
```

That's it! Easy, eh? Actually, you can think of the 60 ALLOT as a DIM statement in BASIC. It reserves memory for the array. The hard part is accessing the individual elements. Also notice that I totally ignored the initial two bytes which VARIABLE automatically reserves; when dealing with large arrays, the first two bytes are insignificant and may be ignored. This makes for much better readability when going over your program listings.

Now refer back to my diagram of the PETS array. The first box of the array is the address provided by PETS. Since the first element has a count byte, simply typing in PETS COUNT TYPE will print out "CAT". But how do you get at the rest of the array? You have to calculate the address of the element, using this simple formula:

```
base addr + (element # * length of each element)
```

The base addr is PETS. Now, as with most of FORTH, element numbers start at zero. Let's say you want the first element using this formula. Plug in the values: base addr=TEST, element #=0, length=6. 6 * 0 = 0, so you are adding 0 to the base addr to find the first element. That makes sense! Similarly, to get to the second element, the sequence to type in is (TEST 1 6 * +). What you are actually doing is adding an offset to the base address. Once you have the address of the element, a simple COUNT TYPE will print the contents, providing you stored the count byte! If you want to view all the elements in PETS, type in:

```
: GO 10 0 DO CR PETS I 6 * + COUNT TYPE LOOP ;
```

Since element #s start at zero, we want to print out elements 0-9. However, you must always add 1 to the upper limit whenever using DO LOOPs in FORTH.

As you can imagine, if you have a lot of string arrays, you will need to make these calculations often. To make it more readible (and more convenient), we can easily turn that into a definition, as follows:

```
: PETS() PETS SWAP 6 * + ;

: GO 10 0 DO CR I PETS() COUNT TYPE ;
```

This is MUCH easier to read than before. As a naming convention, I use the () symbol to indicate that PETS is an "indexing" word; all it requires on the stack is the index, or element #. A word of warning: When you are using DO LOOPS, the word "I" must used in the - same - definition as the loop itself. You cannot put the "I" in the definition of PETS(); it MUST appear in the same definition as the DO LOOP. This problem is actually a blessing in disguise. Since we removed the "I" from PETS(), we are free to use the index word outside of the loop. In other words, if all I needed was the last element of the array, I could just type in 9 PETS() COUNT TYPE. No loop is needed!

Up till now, all you have done is sit back with your arms folded and watch me babble on about accessing an array. Here's your chance to follow along with me as I show you how to store things in your array. First we will use ACCEPT
-->

and input the strings directly into the dictionary, then we will modify our routine so we first input into PAD. First of all, we have to modify our array a bit. In the above example, POOCH barely fit into the space allot for it -- 6 characters. If we are to use ACCEPT (which was defined in the previous tutorial) and input directly into the array, we need to tack on 2 more bytes for each element. You see, ACCEPT (and EXPECT) always glue 2 nulls onto the end of each string. So if you input a string exactly 6 characters long directly into PETS, ACCEPT will over-write the next element with nulls! With this in mind, here is the complete routine:

```
0 VARIABLE PETS
80 ALLOT                (10 items * (6+2) bytes each)
PETS 80 BLANKS
: PETS() PETS SWAP 8 * + ;
: INPUT-IT
  10 0 DO I PETS()   ( addr of each element)
  6 ACCEPT ( max. len for each string=6)
  LOOP ;
: PRINT-IT
  10 0 DO I PETS() COUNT TYPE
       LOOP ;
```

If you have been following since the first installment in this series, the mechanics of this loop are self-explanatory.

This is fine, but remember what I said about avoiding inputting directly into the array? To avoid those darn blanks from creeping in, Input the string into PAD first, then move them into the array. Here is the new routine (remember to FORGET PETS first):

```
0 VARIABLE PETS 60 ALLOT    (10 items + 6 bytes)
PETS 60 BLANKS
: PETS() PETS SWAP 8 * + ;
: INPUT-IT
  10 0 DO PAD 6 ACCEPT
  PAD COUNT 1+ SWAP 1- SWAP
  I PETS()                          (Get addr of element #I)
  SWAP                              ( source addr,dest. addr, cnt)
  CMOVE$ LOOP                       ( CMOVE$ was defined in the previous$)
                                    ( tutorial )
: PRINT-IT
  10 0 DO CR I PETS() COUNT TYPE LOOP ;
```

The PAD COUNT 1+ ... sequence seems confusing, but if you read my last tutorial, you should remember it. We want to move not only the string, but the count byte as well. But PAD COUNT returns the address of the string itself, along with its length. Subtracting 1 backs up the addr to the count byte; meanwhile, add 1 to the cnt on the stack so CMOVE$ will move the entire string+cnt. Also remember that I PETS() just returns the proper address of the element in the array; a similiar sequence in BASIC would be FOR I=1 TO 10 :: INPUT PET$(I) :: NEXT I.

Well, I've run out of room for this issue. Next time I will introduce some string array utility words which will allow you to do some heavy-duty string processing! Bye for now!

NAVARONE DBM

Switching Between Single and Dual Disk Drive Systems

STANDARD: 1A 2DB 3B (o) 4B 5A 6B 7B 9B 10B (o)

In many ways, we've found the Navarone Database Management package to be really great. It is fast and uses a minimum of keystrokes (though the "manual" is not very good). But, there are some serious problems that one can encounter and we'll take a look at a fix for one in this article.

DBM files are stored with a reference to the disk drive number! In other words, the Setup file chains to the Data file and the file chain will always -->

reference the drive from which the Data file was originally established, even though the Setup file can be loaded from either drive. This can spell disaster if you lose a disk drive or you hoped to demonstrate the program on a single-drive system.

The solution is to sector edit the disk, but before you load up your sector editor, the edit that is required is not a straight-forward task! In the Setup file, the first 15 bytes are for the drive reference and filename. Sounds simple, huh? Well, you'll discover that there is an unusual offset! As an example, let's assume that your Data filename is "DSK2.FILENAME". All of the values you find in the sector editor are decimal 128 above the values you might expect. Our example would look like this:

C4 D3 CB B2 AE C6 etc.

Reducing by decimal 128 and converting to ASCII, it looks like this:

D S K 2 . F etc.

If you have difficulties converting hex to decimal or adding and subtracting hex values, you might want to get a programmer's calculator or use the conversion tables available in many printer manuals. Anyway, the B2 in the example can be changed to B1 to change the drive number reference! Ah, it's really simple once you know how! Simply change the least significant nibble (second digit) of the hex value of byte 3 (the fourth byte) of the Setup file to correspond directly to the drive number.

TI-ARTIST II

Getting the Most From a Graphics Program

STANDARD: 1A 2XB or EA or MM or TW
3B (o) 4B 5A 6B 7B 9B
10B (o)

Though many graphics programs exist for the 99/4A, our pick as the leader is TI-Artist II. The program emerges at the top of the pack due not only to its own merits, but also from the availability of compatible software.

First of all, TI-Artist II will load files created by other graphics packages, such as Draw-a-Bit, Draw 'n Plot™, and Graphx. This is often quite convenient. For instance, we find that Draw 'n Plot™ is not as good as some of the other packages for drawing, but it does offer easy to use plotting routines that are quite powerful. Enhancing Draw 'n Plot plots from TI-Artist II is ideal for many applications. And, Graphx also enjoys support from products such as Graphx Companion (Asgard) and, again, the ability to pass such drawings into TI-Artist II is a big plus.

The most impressive support of all is the recent introduction of TI-Artist Companion #1, a 5 disk package of 25 character fonts, 30 pictures and 160 5 x 5 graphics that were originally developed for use in Character Sets and Graphics Design. Developed by Dave Rose, TI-Artist Companion #1 is a giant leap forward in 99/4A graphics! Using the Slides Menu of TI-Artist II, the characters and graphics can be easily loaded to any area of the screen, then mixed with one's own artwork if desired. The package works with Okidata 92-93, IBM, Epson and compatibles, Axiom GP-100 and C-Itoh Prowriter 8510 and compatibles (quite an impressive range of printers). The package is available for only $17.95 from Dave Rose, 2781 Resor Road, Fairfield, OH 45014-5053. The "#1" label on the name seems to imply there will be a "#2", which would be more good news.

TI-Artist II is a program that anyone interested in creating graphics designs on the 99/4A will surely want to have. The program is $19.95 plus $1.50 shipping and handling (MD add 5% sales tax) from Inscebot, Inc., P.O. Box 260, Arnold, MD 21012.

-->

99 POTPOURRI

News, Corrections, Updates, Editorials, Kudos and Come-what-may

A Note From the Editor

In order to provide maximum coverage without raising our prices or diminishing our quality, we have introduced new print formats this month. Your comments would be appreciated.

Also, this issue is much later in going out than anticipated due to family obligations (we're all back in good health now). Work is already under way on our next issue.

Richard Mitchell, Editor

I WISH I HAD:

F3: For F.J. Bubenik, Jr., Hicksville, NY, the program you need for dumping graphics and text to your Prowriter 8510 printer is "Screen Image Dump", by Dave Rose, 2781 Resor Road, Fairfield, OH 45014-5053. The program runs from either BASIC with the Editor/Assembler or Mini Memory or from Extended BASIC (instructions require E/A, XB or TI-W, send $1 if all you have is MM). A screen can be dumped at any point in a program via a CTRL/FCTN key sequence (a keyscan is done at all times). The program is in Assembly Language for speed. Other printers supported are TI-Impact, Epson FX-RX-MX, Panasonic KXP 1090-1091, Star Gemini 10-10X-SG10, BMC BX80, Tally Spirit 80 and NEC 8023. The program is $16.95 (First Class Postage and Handling included). When ordering from Dave Rose, always include your full printer information (brand, serial or parallel, port 1 or 2, baud rate for serial).

M4: I wish I had either a Fortran or a Pascal compiler (or both). James N. Stricherz, College Station, TX.

F4: James, TI did produce a P-Code card and software. The card goes in the TI Peripheral Expansion Box. Not many of the cards were sold, so demand far exceeds supply for used cards. We have heard that some users have implemented a P-Code version of Fortran written for PC's, but we have not yet determined whether that is a legal implementation or exactly how it was accomplished. The P-Code card has its own power switch, but the card's addressing does not permit use with the MYARC MEXP-1 RAM disk (unless some talented hardware hacker builds a bypass). Other cards may also be incompatible. Also, note that the P-Code card runs warmer than other cards. Do not remove the clamshell case, as that is what provides the heatsink for the card (the P.E.B. fan does little to cool cards)! If anyone has further information for James or has a P-Code card for sale at a reasonable price, please let us know.

There will be several 99/4A shows this spring. Word from QB-99'ers User Group is that the group will host the TI Metropolitan Area Regional Conference of User Groups (TIMARC) on April 12, 1986 at Queensborough Community College, 56th Ave. and Springfield Blvd., Bayside, NY 11364. Groups from New York and New Jersey will participate. The conference will start at 1 PM and continue until 6 PM. Tutorials will be from 2 to 5 PM, with products being on view before and after the tutorial session. Admission is free upon presentation of recognized TI User Group ID ($1 to all others). Further information can be obtained by sending a SASE to Mr. Frank Cotty at the college address.

The Texas Instruments Computer Owners Fun Festival, T.I.C.O.F.F., will be March 15, 1986, at Roselle High School, Roselle, NJ. A flea market and a number of prominent guest speakers are on the agenda. Vendor inquiries are invited. Anyone with questions should contact Steve Citron at (201) 686-5616.

News from the 99' Fest-West '86 show in Los Angeles is that a major new hardware product will be on display. The LA show will be March 1 and 2 at the Shrine Auditorium in Los Angeles.

Pilgrims' Pride, a major 99/4A retailer, has moved to 5 Williams La., Hatboro, PA 19040. Phone (215) 441-4262.

Please! Notify us if your address changes! U.S. Third Class Mail cannot be forwarded! Avoid long delays in getting your issues by notifying us promptly of your new address!

Asgard Software has become the first 99/4A firm to offer products on the Electronic Mall section of CIS. To place on-line orders, GO TDS.

The popular terminal emulator program 4A/Talk is now available through Triton, according to the firm's Fall '85 catalog. Triton's address is P.O. Box 8123, San Francisco, CA 94128.

Navarone Industries has changed ownership and moved to Dallas, TX. Chuck Humphries, a long-time key employee, remains with the firm. The new phone number is (214) 941-1816.

Well, we've finally received reports from persons outside of MYARC who have seen Extended BASIC II. The package was unveiled at the CES in Las Vegas, but will likely not be in the hands of the buying public for at least another month. The package is said to be impressive.

A new communications network in Canada is gaining the support of many individuals and Canadian User Groups. TIMELINE is available at $30 Canadian or $25 U.S. sign-up fee. Connect time rates seem to be fairly reasonable. The system supports up to 24 users who can teleconference. A U.S. toll-free number is available (the surcharge to cover this service is far below what the long distance charge would be). For further information, contact T.U.G., 680 - 100th Ave., Laval, Quebec, Canada H7W-3Z6 or call (514) 681-2280. This information was provided by Raynald Makinen, who asks that you mention his user ID, RAYNALD.A1464 when signing up for the service.

-->

SUBSCRIPTIONS (PER YEAR):

U.S. AND POSSESSIONS	
FIRST CLASS	$16.00
THIRD CLASS	$12.00
OTHER COUNTRIES	
AIR MAIL	$26.50
SURFACE MAIL	$16.00
INDIVIDUAL COPIES:	
U.S. SUBSCRIBERS	
FIRST CLASS	$ 1.35
THIRD CLASS	$ 1.00
CANADA SUBSCRIBERS	$ 1.35
OTHER	$ 1.50

Check or Money Order in U.S. funds, coded for processing through the U.S. Federal Reserve Bank System.
No billings or credit sales.
(all issues available at press time)

SUPER 99 MONTHLY ORDER FORM

NAME_____

ADDRESS_____

CITY_____STATE_____

ZIP_____COUNTRY_____

For back issues, specify which:

_____

READER FEEDBACK: (Attach comments)

SUPER 99 MONTHLY is published monthly by Bytemaster Computer Services, 171 Mustang Street, Sulphur, LA 70663.
All correspondence received will be considered unconditionally assigned for publication and copyright and subject to editing and comments by the editors of *SUPER 99 MONTHLY*.

STANDARD KEY

1	Computer	A	TI-99/4A
2	Module	XB	Extended BASIC
		TW	TI-Writer
		EA	Editor/Assembler
		MM	Mini Memory
		DB	Navarone DBM
3	RS-232	B	TI
4	Disk Drive	B	TEAC 55-B
5	Expansion Box	A	TI
6	Disk Controller	B	CorComp
7	Memory Card	B	MYARC MEXP-1 (128K)
9	Monitor or TV	B	TI Color Monitor
10	Printer	B	Gemini 15-X
15	GRAM device	A	GRAM Kracker (tm)

EDITOR
Richard M. Mitchell (CIS 70337,1011)

CORRESPONDING STAFF WRITERS
Barry A. Traver
Charles M. Robertson
Steven J. Szymkiewicz, MD

Bytemaster Computer Services
171 Mustang Street
Sulphur, LA 70663

Bulk Rate
U.S. Postage
PAID
Sulphur, LA 70663
Permit No. 141

POSTMASTER: ADDRESS CORRECTION REQUESTED.
RUSH -- TIME DATED MATERIAL.