

# PLAYGROUND

Run assembly language on an unexpanded TI-99/4A

by Harry Wilhelm

November 2013 – January 2014

Recently Atariage printed a program by James Abbatiello called *Escape the BASIC sandbox*. The program was a simple loop that printed “Hello, World!” shifted by one position every loop so that the text scrolled down the screen. The intriguing thing about the program is that it did this using assembly language on an unexpanded TI-99/4a. Abbatiello found a way to exploit quirks in the BASIC interpreter to load a file name that contained an assembly language program and then run the program. TI did their very best to make it impossible to run assembly language in TI BASIC, but Abbatiello proved that it actually could be done. His idea took a just a few tentative steps out of the sandbox but once that happened there was no looking back. PLAYGROUND builds on his idea and makes it possible to run complicated assembly language programs using nothing but an unexpanded TI-99/4a and a cassette player. Now that we have left the sandbox behind we can join the big kids in the playground.

A program written for playground can be run on any TI system directly from TI BASIC; it can also be run from Extended BASIC; as an E/A5 program; loaded into a supercart; and can even be made into an actual cartridge. One motivation for writing programs for playground is to see just how far you can push the unexpanded TI-99. It is fun to see what can be done with only 256 bytes of RAM; even more so to do something that TI tried so hard to make impossible. It would not be practical to write programs for playground on an unexpanded TI-99, so you will need either the Editor/Assembler or the MiniMemory cartridge plus 32K for development work. Once created, the program can be loaded from cassette and run on an unexpanded TI. This manual assumes the reader has a working knowledge of TMS9900 assembly language.

## HOW PLAYGROUND USES MEMORY

The unexpanded TI-99 has 16K of VDP memory but only 256 bytes of RAM (the scratchpad) can be directly accessed by the 9900 microprocessor. An assembly language program can be stored in the VDP ram but cannot be run directly from there. To run a program of any complexity, some sort of paging is required to load the various segments of the program into the scratchpad. Naturally, the software loading the page must be separate from the page being loaded. Additionally, certain built in console routines are needed to scan the keyboard, move sprites automatically, and play a soundlist. These routines use the interrupt workspace at >8C00 and the GPL workspace at >8E00 plus bytes from >8372 to >837D. Also, the assembly program being run by playground requires its own workspace. With these factors in mind, this memory map of the scratchpad as used by playground will make sense.

## SCRATCHPAD MEMORY MAP

>8300 - >8371	114 bytes available for a program page. (KSCAN and GPLLNK use memory from >836A to >8371)
>8372 - >837D	System use for keyscan, sprite motion, etc. >8373 is GPL substack pointer; >8378 is the random number seed
>837E - >83A1	36 bytes - Page loader
>83A2 - >83C1	Workspace (R15 overlaps into R0 of interrupt workspace)
>83C0 - >83DF	Interrupt workspace. (R0 not available) R8 and R9 (>83D0 and >83D2) are used to store return address for a subroutine.
>83E0 - >83FF	GPL workspace

## PLAYGROUND WORKSPACE AT >83A2

Your program can use R0 to R10 as needed

R10 = >8C00 VDPWD (VDP write data register). You can change this if desired.

R11 and R12 can be used within a page but are changed when loading a new page.

R13 – R15 are used by the page loader and must not be changed:

R13 = offset to be added to a program address to find the address in VDP

R14 = >8800 VDPRD (VDP read data register)

R15 = >8C02 VDPWA (VDP write address register)

## LOADING A PAGE

A program consists of pages of code that are loaded as needed into the scratchpad memory starting at >8300. A page must be no longer than 114 bytes. After loading a page, playground will branch to >8300 which is the first line of code. (At start-up playground will load the first page of code in the program and branch to the first line.) Each page of code is preceded by one word containing the length of the page. A new page is loaded with the following code:

BL @PAGE

DATA PGNAME(The name of the page)

As an example, below is a commented page of code from the DEMO program. The page loads 16 bytes of data to the sprite motion table at >0780 in the VDP ram, then loads a page named RSLOOP.

```
UFOMOT DATA UFOMOE-UFOMOS      Length - Pointer to end minus pointer to start
UFOMOS LI R0,>4780                >0780 with 2nd bit set
      MOV @WKSP+1,*R15            move lsb to VDPWA
      MOV R0,*R15                move msb to VDPWA
      LI R1,UFOMDT-UFOMOS+>8300  make R1 point to UFOMDT
*Be sure you understand this way to find data in your program!
      LI R0,16                    16 bytes to write
UFOMLP MOV *R1+,*R10              a standard loop in assembly - R10 contains VDPWD
      DEC R0
      JNE UFOMLP                 loop till done
      LI R0,>0400                 4 sprites in motion
      MOV R0,@>837A              and move to >837A
      BL @PAGE                   load and run next page
      DATA RSLOOP               which is RSLOOP (reset the loop)
```

```
UFOMDT DATA >FDF2,>0000,>FFE0,>0000,>FE08,>0000,>0118,>0000  UFO motion data
UFOMOE      label for end of program-I like to append "S" at
*           start and "E" at the end.
```

## SUBROUTINES

In general subroutines cannot be used via the usual 9900 methods. Because all code must run from >8300 the BLWP instruction is not useful except perhaps to a console routine. The BL instruction can be used, but only within a page of code. You cannot BL outside of the page of code you are running. The following method has been developed that lets you use one level of subroutines:

BL @SUB

DATA SBNAME (The name of the subroutine)

As an example, below is a commented subroutine that clears the screen. Notice that it uses its own workspace and when the page is loaded registers R0 to R3 are preloaded with data. Because this uses its own workspace it does not effect the workspace of the calling program. Another way to avoid changing the registers of the calling program is to use only R11 and R12, which the pageloder uses when loading a new page.

```

CLEAR  DATA  CLEARE-CLEARs          length of subroutine
CLEARs  LWPI  CLRWS-CLEARs+>8300      this shows how to find the workspace
        MOVb  @WKSP+1, *R1            sets the address to write to
        MOVb  R0, *R1                which is 0
        NOP
CLRLP   MOVb  R2, @>8C00              \VDPWD
        DEC  R3                      a normal loop in A/L
        JNE  CLRLP                    /
        LWPI >83A2                    back to the calling workspace
        BL   @SUBPAG
        DATA RETURN                  load and run RETURN
CLRWS   DATA >4000  R0 = >0000 with bit 2 set
        DATA >8C02  R1 = vdpwa
        DATA >8000  R2 = space with offset
        DATA 768    R3 = counter
CLEARE

```

A large subroutine may need to branch to another page of code. In this case you do not want to change the return addresses that were previously stored. The following code will do this:

```

BL @SUBPAG
DATA SPNAME (The name of the subroutine page)

```

By using its own workspace, the example above is as close as you can get to a BLWP type of subroutine. One advantage is that registers can be preloaded and will always have those values when you call the subroutine. Be sure to restore the normal workspace (>83A2) before returning from the subroutine.

## RETURNING FROM A SUBROUTINE

The page below is used when you want to return from a subroutine. It sets up pointers so the page loader can restore the calling page and return to the code at the right address. This must called with SUBPAG to avoid changing the return addresses. It is used at the end of the CLEAR subroutine above.

```

RETURN DATA RETURE-RETURS
RETURS  MOV  @>83D0, R11              Get page to return to
        A   R13, R11                  Add VDP offset
        MOVb @>83B9, *R15             lsb of R11 to VDPWA
        MOVb R11, *R15               msb of R11 to VDPWA
        MOV  @>83D2, R11              position in page when sub was called
        INCT R11                      need to go to the next instruction
        LI   R12, >8300
        JMP  RETURS+>0094             jump to >8394
RETURE

```

## THE DIFFERENCE BETWEEN BL @PAGE or BL @SUB or BL @SUBPAG

Each of these loads the page the same way. The difference is in how the return addresses are saved. If your page will not use a subroutine then BL @SUBPAG will work fine and save two instructions.

## SOURCE CODE FOR THE PAGELoader

The source code for the pagelader is below. This may be helpful in understanding how it works.

```
*Pagelader code begins at >837E)
PAGE    MOV *R11,@>83D0      stores address of page being loaded
SUB     MOV R11,@>83D2      stores address within page calling sub routine
SUBPAG  A R13,*R11          adjust to point to code in VDP
        MOV @1(R11),*R15    set address to read from
        MOV *R11,*R15      set address to read from
        LI R12,>8300        load code starting at >8300
        MOV R12,R11        entry point of page into R11
*(The RETURN subroutine skips all the above and jumps to the line below.)
        MOV *R14,@>839D    puts length byte into LSB 2 lines below
PGLLOOP MOV *R14,*R12+      start to copy page out of VDP
        CI R12,>8300        length byte of page was put in LSB
        JNE PGLLOOP        loop till done
        B *R11             branch to address contained in R11
```

## CREATING A BASIC LOADER CONTAINING YOUR ASSEMBLY CODE

After writing the assembly language program you will need to create a BASIC program that contains the embedded assembly program. First save the assembly source code, then assemble it to create object code. Once the object code is created, with the E/A or minimemory cartridge in the slot and 32K of memory expansion, type these lines: (Here the object code is DEMO-O)

```
CALL INIT
CALL LOAD("DSKn.DEMO-O") this will load the DEMO program starting at >A000
CALL LOAD("DSKn.MAKEBX-O") this loads the code that will embed DEMO in a BASIC
program (this code is AORG'd at >F000)
CALL LOAD(-31804,240) this runs MAKEBX and creates the BASIC program.
```

Then SAVE or RUN as you desire. If your program is long you may need to CALL FILES(1), then NEW before loading to free up additional room. You can paste the above into BASIC using Classic99 if you want to avoid having to key it in. Use Paste, not Paste XB.

The BASIC program consists of two lines. Line 10 is a REM statement. This can be modified as you desire. Other REM statements and even program statements can be added before line 20, but do not try to edit line 20

## RUNNING A PLAYGROUND PROGRAM IN EXTENDED BASIC

The BASIC program created above can be run from Extended BASIC. If there is no memory expansion you can RUN it like any other XB program. If there is a memory expansion, it must be turned off before running the program. The following line of code shows how to do this:

```
CALL INIT :: CALL LOAD(-31868,0,0):: RUN "DSK2.PROGRAM"
```

This can be part of a menu driven loader program or entered while in the immediate mode.

## CONVERTING TO E/A5

Programs written for playground can be saved and run as a standard E/A5 program. Load 3 files in this order using the Load and Run option in the Editor/Assembler:

```
DSKn.MAKEEA1-0  
DSKn.YOURPROG-0  
DSKn.MAKEEA2-0
```

Then save the EA5 file in the usual way.

## LOADING A PLAYGROUND PROGRAM INTO A SUPERCART

Programs written for playground can be loaded into a supercart with ram at >6000. To do this you first make an E/A5 program similar to what was done above. You need to change the name and length of the program in the code in MAKE6K1-S then save the file and reassemble the code. Then load 3 files in this order using the Load and Run option in the Editor Assembler:

```
DSKn.MAKE6K1-0  
DSKn.YOURPROG-0  
DSKn.MAKE6K2-0
```

Save the E/A5 file in the usual way for future use. When you run the E/A5 program the playground program is loaded to the 6K ram of the supercart and the computer returns to the master title screen. Press a key and your program will show up on the menu. This code can be burned into a rom if you want to make a physical cartridge.

## TIPS ON CODING FOR PLAYGROUND

The source code DEMO-S.TXT, LIFE-S.TXT and SCOLORS-S.TXT are included in this package. You can study these programs for examples of how to work within the confines of playground.

Your program should begin with the following equates:

```
PAGE    EQU >837E  
SUB      EQU >8382  
SUBPAG   EQU >8386  
WKSP     EQU >83A2
```

None of the E/A utilities such as VSBW, VMBW, VSBR, VMBR, VWTR etc. are available. You can write subroutines that perform the same functions as needed. Perhaps if enough people write programs for playground a library of subroutines can be developed.

When you load assembly object code with CALL LOAD it is loaded into CPU memory starting at >A000 and then is transferred from there to the VDP ram inside the BASIC loader program. R13 is used when you need to find the address of data in the VDP ram. R13 points to the LSB of the length byte which simplifies the page handler. In the demo program, a sound list is located at HONEYC. To find this in VDP ram:

```
LI R0,HONEYC-1    subtract one so it points to the msb  
A R13,R0          and now R0 points to the start of the sound list in VDP
```

The 114 bytes of program space can contain a buffer as needed. It should be put at the end of the page for example, a 32 byte buffer could be at >8352. You don't need to use BSS, but it is important to count the bytes of code to make sure it does not overwrite the buffer. If it is short enough, you can load another page without overwriting the buffer.

You will become proficient at counting the words in a page to see how many bytes it contains and to be sure it will fit.

The sprite attribute table begins at >0300, which is also where the color table is. If your program uses more than 3 sprites you should move the color table to >0800 so the sprite data does not conflict with the color table.

With playground there is 16K of VDP ram that is available. This has to contain the screen image, the color tables, character definitions, sprite data, etc. as well as your program. TI's decision to put the character definitions where they did and use a screen offset of >60 makes sense if you want to maximize the amount of program space that is available. When a playground program starts up it calls the GROM routine at G27E3. This clears the screen (ASCII 32 with screen offset of >60), loads the character sets and color tables, and sets the VDP registers. These are the same as the values used by TI BASIC in the immediate mode.

## **SPEED OF EXECUTION**

Less time spent loading pages means more time to execute the code. Loading a page takes 10 assembly instructions plus 3 more for every byte loaded, so there is a slight speed advantage in making a page as long as possible. This changes when you are using subroutines. The subroutine has to be loaded and run, then the loader has to reload the calling page. If that page is short then less time has to be spent loading it. If you need more room for your code, the workspaces from >83C2 to >83FF can be used. You would first store their contents in VDP ram, then load your code in these areas. Be sure to restore these workspaces before enabling interrupts or using keyscan.

## **DEBUGGING**

Because all the code is loaded to and run from >8300, it can be difficult to know where to set breakpoints. MAKEBX-O can help you know where to set breakpoints and the sizes of pages. In your code, add page names to the DEF table with DEF PAGENAME. Then, when creating the embedded BASIC program using MAKEBX-O you will get a list of up to 23 pages in the program that were added to the DEF table. After the name of each page a breakpoint is shown. The breakpoint is the address of the last byte of the page in VDP memory. To use this in Classic99 set a breakpoint with <Vxxxx and a break will occur when that last byte is read from VDP and moved to the scratchpad. Then single step 3 times and you will be at the beginning of the page.

After the address for the breakpoint, the last address needed by the page is displayed. An asterisk is appended to warn you if a page will be at >8372 or greater. Don't forget to account for the memory used by buffers. Also, if a page scans the keyboard remember that you can only use up to >836E.

## PLAYING A SOUND LIST

Playing a soundlist is easier in playground than in normal assembly code because the data is already in the VDP ram. The code below will play the soundlist HONEYC.

```
HONEYS LI R0,HONEYC-1      pointer to soundlist - be sure to subtract 1
      A R13,R0              R0 now points to soundlist in VDP
      MOV R0,@>83CC

      LI R12,>0100
      MOVB R12,@>83CE
      SOCB R12,@>83FD
```

```
HONEYC
      BYTE 9,133,42,144,166,8,176,204,31,223,18
      and the rest of the sound list follows...
```

## RANDOM NUMBERS

On the TI-99 the random number seed is normally located at >83C0. Playground overwrites that address when it loads and also uses it as R15 of the workspace. The random number seed has been moved to >8378. LIM1 will increment >8379 60 times a second which does the same thing as RANDOMIZE in BASIC.

Below is a subroutine used for generating a random number. This code can be used as a subroutine or placed within a page if speed is important. This code has been adapted from the random number routine in the console.

```
*****
*RANDOM is a subroutine to generate a random number.      *
*If you want a RND from 0 to 5, R0 should be 6           *
*The random number is returned in R1                      *
*****
RANDOM DATA RANDOE-RANDOS
RANDOS LI R11,>6FE5
      MPY @>8378,R11
      AI R12,>7AB9
      MOV R12,@>8378
      CLR R11
      SWPB R12
RANDO2 DIV R0,R11
      MOV R12,R1
      BL @SUBPAG
      DATA RETURN
RANDOE
```

## KEYSCAN

Keyscan and GPL routines use the GPL substack pointer at >8373. When initialized, playground sets this pointer to >68. This gives just enough room on the stack to use the GPL routines described below, but it requires the memory locations >836A to >8371 for the stack. If you only want to use keyscan, you can set the pointer to >6C which only requires >836E to >8371.

The following code is the equivalent of KSCAN in normal assembly code.

```
LWPI >83E0
MOV R11,@>836E  store the return address
BL @>000E      (keyscan uses both intws and gplws)
MOV @>836E,R11  restore the return address
LWPI WKSP
```

## GPL ACCESS

A GPLLNK subroutine has been written that can be used to give access to some of the useful GPL routines built into the console. Put the address of the GPL routine in R0, then:

```
BL @SUB
DATA GPLLNK
```

The GPL routine must be set up as described below.

### USEFUL GPL ROUTINES:

No memory setup is needed for the four routines below.

- >0034 Accept tone
- >0036 Bad response tone
- >4D00 scroll screen one line; fill bottom line with 2 edge characters, 28 spaces, 2 edge characters.
- >27EA Load characters, colors, registers, etc. to same values as TI BASIC. (Does not clear screen)

To load character sets, set >834A to point to the address in VDP where the characters will be loaded.

- >0016 Load character set - large capital letters
- >0018 Load character set - normal size capital letters
- >004A Load character set - lower case letters

## EDITOR

The GPLLNK subroutine has been modified to let you access the line editor used by BASIC. To use this, place the starting address in VDP into R0, place the ending address into R1, then:

```
BL @SUB
DATA EDITOR
```

If the ending address is off the bottom of the screen then the line editor will scroll the screen as needed. If you want to use the scrolling feature you should use the edge character (>1F or 31) as in TI BASIC. If you are not using the scrolling feature the edge characters are optional. The line editor uses the normal BASIC character offset (>60 or 96). You will find this subroutine in the section below.



## USEFUL SUBROUTINES

Copy and paste these into your program as needed.

This is part of the playground package – filename is USEFULSUBS.TXT

```
*****
*MOVVDP a subroutine to move bytes from one location to *
*another in the VDP ram. *
*Put address to move from in R0 *
*Put address to move to in R1 *
*Put number of bytes in R2 *
*uses r11 & r12 does not modify R2 calling; R0 & R1 inc'd by r2 count *
*****

MOVVDP DATA MOVVDE-MOVVDS
MOVVDS ORI R1,>4000          set bit in R1 so we can write
      MOV R2,R12
MOVVD1 MOVB @WKSP+1,*R15      start reading bytes of text
      MOVB R0,*R15
      INC R0
      MOVB *R14,R11          read a byte from VDP
      MOVB @WKSP+3,*R15      lsb R1
      MOVB R1,*R15          msb R1
      INC R1
      MOVB R11,*R10          move byte into VDP
      DEC R12
      JNE MOVVD1
      BL @SUBPAG
      DATA RETURN

MOVVDE
*****
*GPLLNK is adapted from millers graphics universal GPL & DSR link *
*Put GPL routine into R0 and call as a subroutine *
*****

GPLLNK DATA GPLLNE-GPLLNS
GPLLNS LWPI >83E0          change to GPL workspace
      MOV @>0050,R4          put >0864 into GPLWS4
      MOV @>83A2,R6          GPL routine is in R0. Move into GPLWS6
      BL *R4                push grom address to substack (routine @ >0864)
      MOV @>8300+GXMLAD-GPLLNS,@>8302(R4)      put >00E8 onto stack
      INCT @>8373            and INC stack pointer
      B @>0060              to GPL interpreter
* >831C below
      DATA XMLRTN-GPLLNS+>8300      pointer to return from GPLLNK
*
      this MUST be at >831C !!!
GXMLAD DATA >1675          grom at >1675 contains >0FFE in reg TI and V2.2
XMLRTN MOVB @>166C,R4      puts >0842 into R4
      BL *R4
      LWPI WKSP
      LI R11,>C81B          some gpl routines trash >837E
      MOV R11,@>837E
      BL @SUBPAG
      DATA RETURN

GPLLNE
```

```

*****
*RETURN - a SUBPAG to return from a subroutine. Must have to use subs. *
*****
RETURN DATA RETURE-RETURS

RETURS  MOV @>83D0,R11      Get page to return to
        A R13,R11          Add VDP offset
        MOVB @>83B9,*R15 lsb of R11
        MOVB R11,*R15      msb of R11

        MOV @>83D2,R11      position in page when sub was called
        INCT R11           next instruction
        LI R12,>8300
        JMP RETURS+>0094    jump to >8394
RETURE

*****
*PRTEXT is a subroutine to print text on the screen *
*Pointer to text is in R0 - uses r11 and r12 *
*set up with LI R0,TEXT-1 then A R13,R0 - then call the sub *
*TEXT BYTE 10,13 row and column to start printing *
* TEXT '3 COLORS' *
* BYTE 11,13 row and column *
* TEXT '4 COLORS' *
* BYTE 0 flag for end of text *
* EVEN *
*****
PRTEXT DATA PRTEXE-PRTEXS
PRTEXS  MOVB @WKSP+1,*R15 will read from VDP from address in R0
        MOVB R0,*R15
        INC R0 point to next byte
        CLR R11
        MOVB *R14,R11 read the byte
        JEQ PRTEXX if EQ then at end of text so return

PRTEX2  CI R11,>2000 if LT >20 then a row, need to set up pointer
        JL GTPNTR

PRTEX3  AI R11,>6000 screen offset
        MOVB @WKSP+25,*R15 lsb R12
        MOVB R12,*R15 msb R12
        INC R12
        MOVB R11,*R10 print character to screen
        JMP PRTEXS

GTPNTR  MOV R11,R12
        SRL R12,3
        AI R12,>3FDF add >4000 and -33=SCREEN LOC
        MOVB *R14,R11
        SRL R11,8
        A R11,R12
        INC R0
        JMP PRTEXS

PRTEXX  BL @SUBPAG
        DATA RETURN
PRTEXE

```

```

*****
*RANDOM is a subroutine to generate a random number *
*if you want a RND from 0 to 5, R0 should be 6      *
*the random number is returned in R1                *
*****

```

RANDOM DATA RANDOE-RANDOS

```

RANDOS  LI R11,>6FE5
        MPY @>8378,R11
        AI R12,>7AB9
        MOV R12,@>8378
        CLR R11
        SWPB R12
RANDO2  DIV R0,R11
        MOV R12,R1
        BL @SUBPAG
        DATA RETURN

```

RANDOE

```

*****
*CLEAR is a sub to fill screen with spaces          *
*****

```

CLEAR DATA CLEARE-CLEARs

```

CLEARs  LI R11,>4000          1st screen position with 2nd bit set
        MOVB @WKSP+23,*R15
        MOVB R11,*R15
        LI R11,>8000          a space with bias of >60
        LI R12,768
CLEAR1  MOVB R11,*R10
        DEC R12
        JNE CLEAR1
        BL @SUBPAG
        DATA RETURN

```

CLEARE

```

*****
*PRNUM is subroutine to print a number on the screen *
*R0= address on screen(prints right to left)         *
*R1=number to print                                  *
*****

```

PRNUM DATA PRNUM-PRNUMs

```

PRNUMs  ORI R0,>4000          set bit 2
        MOV R1,R11
        LI R1,10
PRNUM1  MOV R11,R12
        CLR R11
        DIV R1,R11
        JEQ PRNUMX
        AI R12,>0090          to get ASCII plus offset
        MOVB @WKSP+1,*R15
        MOVB R0,*R15
        DEC R0
        MOVB @WKSP+25,*R10
        JMP PRNUM1
PRNUMX  BL @SUBPAG
        DATA RETURN

```

PRNUME

```

*****
*EDITOR is an adaptation of MG universal GPL & DSR link *
*put start screen position into R0 *
*put last screen position into R1-if off bottom will scroll as needed *
*Returns with R1 set to highest position actually used *
*uses BASIC character offset of >60 *
*****
EDITOR DATA EDITOE-EDITOS
EDITOS MOV R0,@>8320      put starting position into >8320
      MOV R1,@>835E      ending position goes into >835E
      LWPI >83E0          GPLWS
      MOV @>0050,R4      put >0864 into GPLWS4
*>8310
      LI R6,>2A46        address of line editor to GPLWS6
      BL *R4             push grom address to substack (routine @ >0864)
      INCT @>8373        INC stack pointer
      JMP H8322

*>831C below
DATA XMLRTN-EDITOS+>8300  pointer to return from GPLLNK MUST be at >831C
GXMLAD DATA >1675      grom at >1675 contains >0FFE in reg TI and V2.2
DATA 0                  starting position goes here (>8320)
H8322 MOV @>8300+GXMLAD-EDITOS,@>8302(R4)  put >1675 onto stack
      B @>0060          to GPL interpreter
XMLRT1 DATA >C81B      >>>>>can remove if not scrolling<<<<<
XMLRTN MOV @>166C,R4    puts >0842 into R4
      BL *R4
      LWPI WKSP
      MOV @>832A,R1
      MOV @XMLRT1-EDITOS+>8300,@>837E >>>>>can remove if not scrolling<<<<<

      BL @SUBPAG
DATA RETURN
EDITOE

*****
*HCHAR (similar to BASIC subprogram) *
*R0=Screen address - ROW*32-1+COL-1 *
*R1=byte to print in MSB - can add offset if desired *
*R2=number of bytes to print *
*****
HCHAR DATA HCHARE-HCHARS
HCHARS ORI R0,>4000      set bit so we can write
*      AI R1,>6000      offset of >60 if desired
HCHAR1 DEC R2            one less byte to write
      JLT HCHAR2
      MOVB @>83A3,*R15 tell VDP what address to write to
      MOVB R0,*R15      "" ""
      INC R0            one position to right
      MOVB R1,*R10      write the byte
      CI R0,>4300        past bottom of screen?
      JLT HCHAR1
      LI R0,>4000        reset to upper left of screen
      JMP HCHAR1
HCHAR2 BL @SUBPAG
DATA RETURN
HCHARE

```

```

*****
*VCHAR (similar to BASIC subprogram) *
*R0=Screen address - ROW*32-1+COL-1 *
*R1=byte to print in MSB - can add offset if desired*
*R2=number of bytes to print *
*****

```

```

VCHAR DATA VCHARE-VCHARS
VCHARS ORI R0,>4000          set bit so we can write
*      AI R1,>6000          offset of >60 if desired
VCHAR1 DEC R2
      JLT VCHAR2
      MOVB @>83A3,*R15 tell VDP what address to write to
      MOVB R0,*R15        "" ""
      AI R0,>0020          down one row
      MOVB R1,*R10        write the byte
      CI R0,>4300
      JLT VCHAR1
      AI R0,>FD01          up to top row and over one
      CI R0,>4020
      JLT VCHAR1
      LI R0,>4000
      JMP VCHAR1
VCHAR2 BL @SUBPAG
DATA RETURN
VCHARE

```

```

*****
*BIT REVERSAL ROUTINE - SIMILAR TO ROUTINE AT GROM >003B *
*PUT ADDRESS IN R0 AND # BYTES IN R1, THEN CALL THE SUB *
*USES R1,R2,R3 *
*****

```

```

BITREV DATA BRRE-BRRS

```

```

BRRS  MOVB @WKSP+1,*R15      lsb of R0
      MOVB R0,*R15          msb of R0
      ORI R0,>4000          will write back to this address
      MOVB *R14,R3          read the byte
      LI R11,8
BRR2  SRL R12,1              shift puts a zero in left hand bit
      SLA R3,1
      JNC BRR3              if a bit was not shifted out skip, else
      AI R12,>8000          set left hand bit
BRR3  DEC R11
      JNE BRR2              loop till 8 bits are done
      MOVB @WKSP+1,*R15
      MOVB R0,*R15
      ANDI R0,>3FFF          next operation will be a read
      MOVB R12,*R10          write the byte
      INC R0                 next byte in vdp
      DEC R1
      JNE BRRS
      BL @SUBPAG
DATA RETURN
BRRE

```

```

*****
*SCROLL will scroll the screen and fill the bottom row with spaces.      *
*Does not change user registers. This is faster than the GPLLNK scroll    *
*Code is from >8300 to >8349; 32 byte buffer is at >8352                *
*Be careful to keep buffer and code separate when using a buffer this way.*
*****

```

# SCROLL DATA SCROLE-SCROLS

```

SCROLS  LI R11,>0020          start with second line on screen

SCROL1  MOVB @WKSP+23,*R15
        MOVB R11,*R15        now are set up to read from VDP
        LI R12,>8352          will read to buffer starting at >8352
SCROL2  MOVB *R14,*R12+ \
        CI R12,>8372          read 32 bytes to buffer
        JNE SCROL2          /

        AI R11,>3FE0          up a line and with 2nd bit set
        MOVB @WKSP+23,*R15
        MOVB R11,*R15        now are set up to write to VDP
        LI R12,>8352          buffer is here
SCROL3  MOVB *R12+,*R10 \
        CI R12,>8372          write 32 byte buffer to vdp
        JNE SCROL3          /

        AI R11,>C040          reset 2nd bit and down 2 lines
        CI R11,>0300          off bottom?
        JLT SCROL1          no, so keep scrolling

        LI R11,>8000          space with offset of >60
        LI R12,32
SCROL4  MOVB R11,*R10 \
        DEC R12              write 32 spaces to screen
        JNE SCROL4          /

        BL @SUBPAG
        DATA RETURN          end of code is at >8349; buffer starts at >8352
SCROLE

```