VOLUME 2 ISSUE 2 $1.75 JULY 1986

"Serving 99'ers Since 1984"

THE SMART PROGRAMMER

Many thanks go to everyone who commented on last month's first issue from Bytemaster. Apparently we hit on many of the subjects of interest. There were also a lot of requests for other topics, so this month we offer an issue that includes several Assembly articles. We'll continue to diversify our coverage each month.

For those of you who like to stay informed of the possibilities of the future (and are suffering from a streak of boredom), how would you like to have an 8ØØ megabyte drive? What's that? You can't add on to your house? No need! Several firms, including Richo and Toshiba, have developed WORM (Write Once Read Mostly) optical drives that use 5 1/4" removable disks! Before you rush out to buy one, the Richo drive without controller is $2,94Ø and the Toshiba system with controller is about $88ØØ. But, an 8ØØ megabyte disk pack is scheduled to go for only $88! Prices will likely drop some and hopefully such products will eventually be priced for the home market. For now, let's get back to the realities of our homes!

Q&A

Can I use GRAM Kracker™ to eliminate the foreign language menu options of TI-Writer?

Yes. Use the GK Editor to change address g6ØØ6 from 6Ø 1Ø to 6Ø CB. That will re-direct the application pointer, thereby avoiding display of the foreign menu options.

While on the topic of GK and TI-Writer, for early versions of the TIWGRAMDSK, the .IF function did not work. In setting up, the program does the equivalent of CALL FILES(1) and should do a CALL FILES(4). To fix this, use the GK Editor to change g6323 from Ø1 to Ø4. To correct the MILK disk, edit sector 224, byte 79, with the change being the same, Ø1 to Ø4.

Which version of FORTH supports the FORTH Recursive Decompiler found in the June 1986 issue?

As it was published, the program runs under TI-FORTH. For Wycove users, add the following to the end of the definition of CK: (but before the semi-colon, of course), which we listed as being on Screen 1Ø3:

```
OVER DUP ' R/W =
OVER ' R/W-CLOSE =
OR SWAP ' SAVEBLK =
OR Ø= AND
```

By the way, our FORTH section, Mariusz Stanczak's 5th 1- =FORTH, will be returning soon with more useful programs and tips.

Mini Memory ROM >6000->6FFF

>6000	>AA00	Space for Standard Rom/Grom Header - All >0000
>6010	>605A	Start address of NAME LINK routine (i.e. Start Name)
>6012	>62CA	Start address of Tagged Object Code Loader from GPL
>6014	>618C	Start address of CIF (Convert Integer to Floating Point)
>6016	>0000	not used
>6018	UTILITY VECTOR TABLE (ie: BLWP @KSCAN)	
>6018	>7092	GPLLNK Utility workspace pointer
>601A	>60F6	Start address for BLWP @GPLLNK
>601C	>7092	XMLLNK Utility workspace pointer
>601E	>60C8	Start address for BLWP @XMLLNK
>6020	>7092	KSCAN Utility workspace pointer
>6022	>6110	Start address for BLWP @KSCAN
>6024	>7092	VSBW Utility workspace pointer
>6026	>6126	Start address for BLWP @VSBW
>6028	>7092	VMBW Utility workspace pointer
>602A	>6132	Start address for BLWP @VMBW
>602C	>7092	VSBR Utility workspace pointer
>602E	>6140	Start address for BLWP @VSBR
>6030	>7092	VMBR Utility workspace pointer
>6032	>614C	Start address for BLWP @VMBR
>6034	>7092	VWTR Utility workspace pointer
>6036	>615A	Start address for BLWP @VWTR
>6038	>7098	DSRLNK Utility workspace pointer
>603A	>61E4	Start address for BLWP @DSRLNK
>603C	>70D8	LOADER Utility workspace pointer
>603E	>62EC	Start address for BLWP @LOADER
>6040	>70F8	NUMASG Utility workspace pointer
>6042	>660E	Start address for BLWP @NUMASG
>6044	>70F8	NUMREF Utility workspace pointer
>6046	>66FE	Start address for BLWP @NUMREF
>6048	>70F8	STRASG Utility workspace pointer
>604A	>6768	Start address for BLWP @STRASG
>604C	>70F8	STRREF Utility workspace pointer
>604E	>6888	Start address for BLWP @STRREF
>6050	>70F8	ERR Utility workspace pointer
>6052	>6966	Start address for BLWP @ERR
>6054	>0064	Data 100
>6056	>2000	Data >2000 (H20 and H2000)
>6058	>2E	Byte Decimal Point '.'
>6059	>00	Byte >00
>605A	Start of Name Link routine. (Finds Start Name in REF/DEF Table)	
>60BC	Routine to Return to Assembly Language from GPLLNK	
>60C8	Start of XMLLNK Routine. (Link to system Utilities)	
>60F6	Start of GPLLNK Routine. (Link to GPL Routines)	
>6110	Start of KSCAN Routine. (Keyboard Scan)	
>6126	Start of VSBW Routine. (VDP single byte write)	
>6132	Start of VMBW Routine. (VDP multiple byte write)	
>6140	Start of VSBR Routine. (VDP single byte read)	
>614C	Start of VMBR Routine. (VDP multiple byte read)	
>615A	Start of VWTR Routine. (Write to VDP register)	
>618C	Start of CIF Routine. (Convert Integer to Floating Point)	
>61E4	Start of DSRLNK Routine. (Link to DSR routines)	
>62CA	Start of Tagged Object Code Loader when coming from GPL	
>62EC	Start of LOADER Routine. (Loads Tagged Object Code - DIS/FIX 80)	
>660E	Start of NUMASG Routine. (Basic Numeric Variable Assignment)	

Mini Memory ROM >6000->6FFF Continued

>66FE		Start of NUMREF Routine. (Basic Numeric Variable Reference)
>6768		Start of STRASG Routine. (Basic String Variable Assignment)
>6888		Start of STRREF Routine. (Basic String Variable Reference)
>6966		Start of ERR Routine. (Basic Error Message Routine)
>697E		Thru >6F0C Not Used All >0000 - Can Be used in the Gram Kracker
>6F0E		Start of Default Mini Memory REF Table
>6F0E	UTLTAB >7020	Pointer to Utility Table in MM RAM
>6F16	PAD >8300	Start address of Scratch Pad Ram
>6F1E	GPLWS >83E0	GPL Workspace pointer
>6F26	SOUND >8400	Location of the Sound Chip
>6F2E	VDPRD >8800	Address for VDP Read Byte port
>6F36	VDPSTA >8802	Address for VDP Read Status port
>6F3E	VDPWD >8C00	Address for VDP Write Byte port
>6F46	VDPWA >8C02	Address for VDP Write (set) Address port
>6F4E	SPCHRD >9000	Address for Speech Read port
>6F56	SPCHWT >9400	Address for Speech Write port
>6F5E	GRMRD >9800	Address for Grom/Gram Read Byte port
>6F66	GRMRA >9802	Address for Grom/Gram Read Address port
>6F6E	GRMWD >9C00	Address for Grom/Gram Write Byte port
>6F76	GRMWA >9C02	Address for Grom/Gram Write (set) Address port
>6F7E	SCAN >000E	BL address for key scan routine in Console ROM
>6F86	XMLLNK >601C	BLWP address for XMLLNK Routine
>6F8E	KSCAN >6020	BLWP address for Keyboard Scan
>6F96	VSBW >6024	BLWP address for VDP Single Byte Write
>6F9E	VMBW >6028	BLWP address for VDP Multiple Byte Write
>6FA6	VSBR >602C	BLWP address for VDP Single Byte Read
>6FAE	VMBR >6030	BLWP address for VDP Multiple Byte Read
>6FB6	VWTR >6034	BLWP address for VDP Write To VDP Registers
>6FBE	DSRLNK >6038	BLWP address for DSRLNK Routine
>6FC6	LOADER >603C	BLWP address for Tagged Object Code Loader
>6FCE	GPLLNK >6018	BLWP address for GPLLNK Routine
>6FD6	NUMASG >6040	BLWP address for Numeric Assignment Routine
>6FDE	NUMREF >6044	BLWP address for Numeric Reference Routine
>6FE6	STRASG >6048	BLWP address for String Assignment Routine
>6FEE	STRREF >604C	BLWP address for String Reference Routine
>6FF6	ERR >6050	BLWP address for Error Message Routine

Mini Memory RAM >7000->7FFF

>7000	>A55A	Constant that indicates that INIT MINI MEM has been done
>7002	>0000	Start of Identifiers for Arguments passed by CALL LINKs
>701C	>7118	First Free address in Mini Memory Ram
>701E	>7FFF	Last Free address in Mini Memory Ram
		also pointer to user's REFs and DEFs thru >7FFF
	UTLTAB	
>7020	>0000	Default START address for program just loaded
>7022	>A000	First Free address in High Memory
>7024	>FFE0	Last Free address in High Memory
>7026	>2000	First Free address in Low Memory
>7028	>3FFF	Last Free address in Low Memory

Mini Memory RAM >7000->7FFF Continued

>702A	>0000	Saved Checksum
>702C	>0000	Saved Pointer to FLAG byte in PAB (in VDP)
>702E	>0000	Saved GPL return address
>7030	>0000	Saved CRU base of Peripheral (>1100 for Disk Controller)
>7032	>0000	Saved Entry address of DSR
>7034	>0000	Saved Device Name Length
>7036	>0000	Saved Pointer to Device Name in PAB (in VDP)
>7038	>0000	Saved Version Number of DSR (i.e. >0001)
>703A		Start of 80 BYTE RECORD Buffer for LOADER
>708A		Start of DEVICE NAME Buffer
>7092		Start of UTILITY Workspace Registers
>7098		Start of DSRLNK Workspace Registers
>70B8		Start of USER Workspace Registers
>70D8		Start of LOADER Workspace Registers
>70F8		Start of Variable Storage area (temp data)
>7118		First Free Address in Mini Memory (Pointed to by >701C)
>7FFF		Last Free Address in Mini Memory (Pointed to by >701E) Also Start of User REF/DEF Table (Grows toward >7118)

Mini Memory HIGH MEMORY EXPANSION >A000->FFFF

>A000	>000x	File type and record length for EXPMEM2 RAM file (see MM manual for x values to CALL LOAD)

NOTE: The Mini Memory Tagged Object Code Loader always loads RORG files starting at >A000. If you want to load a small file into the Mini Mem RAM it must be AORG'd to >7118.

If the file contains any AORG (Absolute Origin) Code the loader will load it where the programmer specified (i.e. AORG >2000). Also, since the Mini Mem Loader resides entirely in cartridge Rom and uses the Mini Mem Ram for temporary storage, it can load DIS/FIX 80 Tagged Object Code anywhere in Low or High Mem according to the RORG and AORG directives in the file.

Cassette to Disk and Back

Article by Richard M. Mitchell
Programs by Thomas S. Freeman, MD
and Richard M. Mitchell

While many of you may have vowed to stay away from the slow load of cassette forever, there are times when cassette is a welcome advantage, as can be evidenced by use of the programs listed below.

Tom Freeman's program was developed to allow Tom's son to carry a somewhat minimal system on vacation and still be able to run disk-origin Assembly games. Editor/Assembler and 32K are still required, but if you have a 32K that is not

in the PEB (in the console, in the Speech Synthesizer or in a "freight train" model), you're ready to travel light (without a forklift for the PEB). Of course, the program is not limited to use for games and can be used on many Program Image files.

The program I wrote was developed in response to a number of requests from readers who wanted to use the Assembly programs they had stored on cassette prior to their purchase of a disk system. It is written for Mini Memory, 32K and disk.

Though the two programs are somewhat similar, there are some interesting contrasts. From the E/A, there was an advantage to pokeing (CALL LOAD'ing, in TI parlance) the filename, as the E/A's BASIC support must be loaded from disk. The program logic is dependent on the size of the files, so that adding the BSCSUP file would have complicated matters without gaining substantial speed in running the program. From Mini Memory, the BASIC support routines are resident and all disk files for the program are the same size (17 sectors), so I chose to pass the filename as a variable.

One very interesting feature of the E/A program is that if you have more than one disk file, such as UTIL1 and UTIL2, you simply save all files to tape. When loading the program, you may be as impressed as I was to watch the second cassette load prompt appear!

With the Mini Memory program, my intent is for the program to be somewhat flexible. The program saves a memory image of the entire Mini Mem RAM (>7ØØØ to >7FFF), so that Assembly code with a pre-existing DEF Table is not a requirement -- you could even save data for peeks and pokes.

Another reason for including these programs in this issue is that a lot of readers are getting started in Assembly and wanted some more examples, especially for using DSRLNK and linking to BASIC. I hope these programs inspire you to write Assembly code.

```
*****************************************************************
* MINI MEMORY SAVE AND LOAD (DISK)                              *
* COPYRIGHT RICHARD M. MITCHELL                                 *
* JUNE, 1986                                                    *
* FOR JULY 1986 ISSUE OF THE SMART PROGRAMMER                   *
* TO SAVE MINI MEMORY RAM:                                      *
*      1) INITIALIZE MINI MEMORY                                *
*      2) LOAD MINI MEMORY RAM WITH CODE OR DATA                *
*      3) FROM BASIC, CALL THIS A/L PROGRAM WITH A PROGRAM SUCH AS:  *
*         100 CALL LOAD("DSK1.MM/O")                            *
*         110 CALL LINK("MMSAVE","DSK1.TEST/MM")                *
*         120 END                                               *
* TO LOAD MINI MEMORY RAM:                                      *
*      1) FROM BASIC, CALL THIS A/L PROGRAM WITH A PROGRAM SUCH AS:  *
*         100 CALL INIT                                         *
*         110 CALL LOAD("DSK1.MM/O")                            *
*         120 CALL LINK("MMLOAD","DSK1.TEST/MM")                *
*         130 END                                               *
*      2) ACCESS YOUR CODE OR DATA NORMALLY                     *
* OR, USE PROGRAM LISTED HEREIN THAT PROMPTS FOR FILENAME       *
*****************************************************************
* CAN BE USED TO LOAD THE MINI MEMORY DISPLAY ROUTINE, LINES, OR WHATEVER  *
* NOTE: DEF TABLE ENTRY WILL OVER-WRITE PORTION OF CONTENTS OF MINI MEMORY *
*       RAM -- MAKE ALLOWANCES!                                 *
*****************************************************************
```

```
*****************************************************************************
* CORRECTIONS TO MM A/L DISPLAY AT (MM MANUAL)                               *
* (FROM NOVEMBER 1984 SUPER 99 MONTHLY)                                     *
*                                                                           *
* LOCATION  TYPOGRAPHICAL ERROR  CORRECTION                                 *
* --------  -------------------  ----------                                 *
* >7E60     MOVE                 MOV                                        *
* >7E90     A1                   AI                                         *
* >7E94     L1                   LI                                         *
* >7EC4     A1                   AI                                         *
* >7EC8     A1                   AI                                         *
*****************************************************************************
* MINI MEM SAVE AND LOAD
       DEF  MMSAVE,MMLOAD      PROGRAM ENTRY POINTS
STRREF EQU  >604C              STRING REFERENCE UTILITY BLWP ADDRESS
VMBW   EQU  >6028              VIDEO MULTIPLE BYTE WRITE BLWP ADDRESS
VMBR   EQU  >6030              VIDEO MULTIPLE BYTE READ BLWP ADDRESS
DSRLNK EQU  >6038              DEVICE SERVICE ROUTINE BLWP ADDRESS
PAB    EQU  >0F80              VDP ADDRESS OF PERIPHERAL ACCESS BUFFER
PABBUF EQU  >1000              SIZE OF PAB BUFFER
MM     EQU  >7000              ADDRESS OF MINI MEMORY RAM
STATUS EQU  >837C              GPL STATUS BYTE ADDRESS
PNTR   EQU  >8356              POINTER TO 1ST CHARACTER AFTER PAB
WS     EQU  >8300              OUR WORKSPACE
BUF    EQU  >2001              POINTER TO BUFFER FOR STRING FROM BASIC BUFFER
BUFEND EQU  >2010              POINTER TO END OF STRING FROM BASIC BUFFER
FNAME  EQU  >201C              POINTER TO FILENAME PORTION OF PDATA (TO PAB)
       AORG >2000              BEGIN AT HEX 2000, BEGINNING OF LOW MEMORY
BUFFER BYTE >0F                LENGTH OF STRING FROM BASIC (MAX DECIMAL 15)
       BSS  >0F                BUFFER FOR STRING FROM BASIC
CONST  BYTE >0F                CONSTANT TO RESTORE MAX BUFFER LENGTH
PDATA  DATA >0600,PABBUF,>0000,>1000,>000F   PAB INFORMATION (SEE A/L MANUAL)
       TEXT '                '                PAB INFO (FILENAME)
SV     BYTE >06                FOR MOVING TO FIRST BYTE OF PDATA (SAVE)
LD     BYTE >05                FOR MOVING TO FIRST BYTE OF PDATA (LOAD)
SAVRTN DATA 0                  WHERE WE SAVE RETURN TO BASIC
MMSAVE MOV  R11,@SAVRTN        CONTENTS OF REGISTER 11 IS RETURN TO BASIC (KEEP)
       LWPI WS                 LOAD WORKSPACE POINTER IN SCRATCH PAD RAM (>8300)
       BL   @STRING            GET STRING FROM BASIC
       MOVB @SV,@PDATA         PUT SAVE REPRESENTATION IN PDATA (GOES TO PAB)
       BL   @BEGPAB            BEGIN THE SET-UP OF PERIPHERAL ACCESS BUFFER (PAB)
       BL   @SET               SET MEMORY AREA (MINI MEMORY RAM)
       BLWP @VMBW              WRITE THE MEMORY AREA IN VDP
       BLWP @DSRLNK            BRANCH TO DEVICE SERVICE ROUTINE (DISK ACCESS)
       DATA 8                  DATA FOR DSRLNK
       B    @EXIT              BRANCH TO EXIT
MMLOAD MOV  R11,@SAVRTN        -
       LWPI WS                   |
       BL   @STRING              |
       MOVB @LD,@PDATA           | BASICALLY DOING ABOUT THE SAME AS
       BL   @BEGPAB              | MMSAVE EXCEPT LOADING FROM DISK TO MEMORY
       BLWP @DSRLNK              | INSTEAD OF SAVING MEMORY TO DISK
       DATA 8                    |
       BL   @SET                 |
       BLWP @VMBR                |
       B    @EXIT              -
STRING CLR  R0                 -
       LI   R1,1                 | GET A STRING FROM BASIC
       LI   R2,BUFFER            | AND PUT IT IN BUFFER
       BLWP @STRREF           -
```

```
       MOVB @BUFFER,@PDATA+9  MOVE FILENAME LENGTH TO PROPER AREA OF PDATA
       MOVB @CONST,@BUFFER    RESTORE MAX. VALUE FOR LENGTH OF STRING FROM BASIC
       LI   R0,BUF            -
       LI   R1,FNAME            |
S1     MOVB *R0+,*R1+           | MOVE BASIC STRING BUFFER TO PDATA FILENAME
       CI   R0,BUFEND           |
       JNE  S1                -
       RT                     RETURN TO CALLING ROUTINE (MMSAVE OR MMLOAD)
BEGPAB LI   R0,PAB            -
       LI   R1,PDATA            |
       LI   R2,>25              | BEGIN SETTING UP PAB
       BLWP @VMBW               |
       LI   R6,PAB+9            |
       MOV  R6,@PNTR          -
       RT
SET    LI   R0,PABBUF         -
       LI   R1,MM               | AREA IN MEMORY ACCESSED (MINI MEMORY RAM)
       LI   R2,>1000          -
       RT
EXIT   CLR  R0                -
       MOVB R0,@STATUS          | RETURN TO BASIC
       MOV  @SAVRTN,11          |
       RT                     -
       END
```

```
> 100 CALL CLEAR
> 110 PRINT "FOR MINIMEM":"  1
  .  SAVE":"  2.  LOAD":"  3.
   EXIT"
> 120 CALL KEY(5,K,S)
> 130 IF (S<1)+(K<49)+(K>51)TH
  EN 120
> 140 ON K-48 GOTO 1000,2000,9
  99
> 150 PRINT "AGAIN (Y OR N)?"
> 160 CALL KEY(5,K,S)
> 170 IF K<91 THEN 180
> 175 K=K-32
> 180 IF (S<1)*(K<>78)*(K<>89)
  THEN 160
> 185 KS=K
> 190 IF K=89 THEN 100
> 999 END
> 1000 PRINT "SAVE"
> 1010 GOSUB 3000
> 1020 CALL LINK("MMSAVE",F$)
> 1030 GOTO 150
> 2000 PRINT "LOAD"
> 2010 GOSUB 3000
> 2020 CALL LINK("MMLOAD",F$)
> 2030 GOTO 150
> 3000 IF K<>50 THEN 3030
> 3010 IF KS=89 THEN 3040
> 3020 CALL INIT
> 3030 CALL LOAD("DSK1.MM/O")
> 3040 PRINT "ENTER FILENAME"
> 3050 INPUT F$
> 3060 RETURN
```

```
**************************************************************************
* DISK TO TAPE AND TAPE TO DISK CONVERSION PROGRAM
* TOM FREEMAN
* 515 ALMA REAL DR.
* PACIFIC PALISADES, CA  90272
* FOR USE WITH PROGRAMS MEANT TO BE LOADED BY THE RUN
* PROGRAM FILE OPTION (#5) OF E/A.  IT MAY BE USED FOR
* OTHER, NON-STANDARD, FILES, BUT IN THAT CASE THE TWO
* INSTANCES OF BL @ CHANGE SHOULD BE DELETED, AND THE
* 4TH WORD OF EACH PAB SHOULD BE REPLACED BY >XX00,
* WHERE >XX IS THE HEX EQUIVALENT OF THE NUMBER OF
* SECTORS TAKEN UP BY THE PROGRAM (PER DISK CATALOG)
* MINUS 1.  IF THE ORIGINAL FILE IS ON TAPE AND THIS
* NUMBER IS NOT KNOWN, USE >2F, THEN CHECK THE DISK
* FILE WITH A SECTOR EDITOR TO SEE WHERE 00'S BEGIN.
* THE PROGRAM CAN THEN BE RERUN WITH THE PROPER NUMBER.
* NOTE:  BECAUSE OF THE REF'S TO GPLLNK AND DSRLNK, THE
* PUBLISHED BY PERMISSION
* OF TOM FREEMAN
* THANKS, TOM!
```

```
* PROGRAM WILL ONLY WORK WITH E/A.  IT IS CALLED FROM
* BASIC.
* DISK TO TAPE AND TAPE TO DISK CONVERSION PROGRAM
       DEF  DISTAP,TAPDIS
       REF  DSRLNK,GPLLNK,VMBW,VMBR
STATUS EQU  >837C
FAC    EQU  >834A
PAB    EQU  >0F80
PNTR   EQU  >8356
WS     EQU  >8300
       AORG >3000
*
* THE FOLLOWING IS THE DISK FILE
* AND HAS BEEN PREPARED FROM BASIC
*
PABDSK DATA >0500,>1000,0,>2000
       BYTE 0
       BYTE 0              LENGTH BYTE
       BSS  15             FILE NAME
*
* THE FOLLOWING IS THE CASSETTE FILE
* NOTE:  IF USING CS1 FOR INPUT IN "RUN PROGRAM FILE" IN E/A
* USE CS1.X AS DEVICE NAME, NOT CS1
*
PABCS  DATA >0600,>1000,0,>2000,>6003  LAST WORD IS SCR OFFSET & LEN BYTE
CS1    TEXT 'CS1'
SAVE   BYTE >06
LOAD   BYTE >05
SAVRTN DATA 0
DISK   LI   0,PAB
       LI   1,PABDSK       LOAD PAB FOR DISK FILE
       LI   2,25
       BLWP @VMBW
       LI   6,PAB+9
       MOV  6,@PNTR
       BLWP @DSRLNK
       DATA 8              MOVE FILE TO VDP AT >1000
       RT
CHANGE LI   0,>1002        2ND WORD CONTAINS # BYTES IN FILE
       LI   2,2            AND BELONGS IN 4TH WORD OF PAB (R1)
       BLWP @VMBR
       RT
TAPE   LI   0,PAB
       LI   1,PABCS
       LI   2,13
       BLWP @VMBW          SET UP CASSETTE PAB TO SAVE
       LI   1,PAB+13       1ST CHAR AFTER PAB MUST BE AT PNTR
       MOV  1,@PNTR
       LI   1,>0800
       MOVB 1,@>836D       >836D MUST CONTAIN 8 (DSR CALL)
       LI   0,PAB+10
       LI   1,FAC
       LI   2,3
       MOV  2,@PNTR-2          >8345 MUST CONTAIN NAME LEN (3)
       BLWP @VMBR              FAC MUST CONTAIN DEVICE NAME
       CLR  @>83D0             >83D0 MUST CONTAIN 0
       MOVB @>83D0,@STATUS     CLEAR STATUS BYTE
       BLWP @GPLLNK            BRANCH TO THE DSR
       DATA >3D
       RT
```

```
DISTAP MOV  11,@SAVRTN
       LWPI WS
       MOVB @LOAD,@PABDSK      PREPARE DISKFILE FOR LOAD
       MOVB @SAVE,@PABCS       PREPARE TAPEFILE FOR SAVE
       BL   @DISK
       LI   1,PABCS+6
       BL   @CHANGE
       BL   @TAPE
       JMP  RETURN
TAPDIS MOV  11,@SAVRTN
       LWPI WS
       MOVB @LOAD,@PABCS       PREPARE TAPEFILE FOR LOAD
       MOVB @SAVE,@PABDSK      PREPARE DISKFILE FOR SAVE
       BL   @TAPE
       LI   1,PABDSK+6
       BL   @CHANGE
       BL   @DISK
RETURN CLR  Ø
       MOVB Ø,@STATUS
       MOV  @SAVRTN,11
       RT                      RETURN
       END
```

```
> 100 DNAME=4096*3+9
> 110 CALL INIT
> 120 CALL LOAD("DSK1.DISKTAPE
  /O")
> 130 INPUT "DISKFILE TO SAVE/
  LOAD        ":NAME$
> 140 LE=LEN(NAME$)
> 150 CALL LOAD(DNAME,LE)
> 160 FOR X=1 TO LE
> 170 CALL LOAD(DNAME+X,ASC(SE
  G$(NAME$,X,1)))
> 180 NEXT X
> 190 PRINT :"PRESS  D. DISK T
  O TAPE":" OR    T. TAPE TO D
  ISK"
```

```
> 200 CALL KEY(Ø,K,S)
> 210 IF S=Ø THEN 200
> 220 IF K=68 THEN 260
> 230 IF K<>84 THEN 200
> 240 CALL LINK("TAPDIS")
> 250 GOTO 270
> 260 CALL LINK("DISTAP")
> 270 PRINT :"DO ANOTHER? Y/N"
  :::
> 280 CALL KEY(5,K,S)
> 290 IF S=Ø THEN 280
> 300 IF K=89 THEN 130
> 310 IF K<>78 THEN 280
> 320 STOP
```

Universal GPLLNK and DSRLNK

code by Craig Miller and D.C. Warren
article by Richard M. Mitchell

Listed below are versions of Assembly Language GPLLNK and DSRLNK subroutines that will work from virtually any 99/4A (not 99/4) environment (the addresses used are common to all versions of 99/4A's)! The subroutines will work with any module loader or disk controller loader, with DIS/FIX 8Ø Auto Start or Non-Auto Start programs, as well as Program Image type files! Module GROM addresses are not used for returning to the caller. GROM Ø address >176C is used for XML RTN's.

The DSRLNK uses GROM Ø's DSR LINK, so it works exactly the same as BASIC's or Extended BASIC's. It will recognize any valid DSR name, including CS1 and CS2! Readers are cautioned that unbridled access to cassette can have negative effects, as the cassette messages are generated for Graphics mode, so that if a program is not in standard Graphics mode, the prompts would not appear properly

on the screen. Additionally, for some programs, one may not want cassette access. In such cases, a routine to exclude CS1 and CS2 as valid parameters would be necessary.

Best of all, the subroutines are very compact -- only 186 bytes for both subroutines! The GPL access in the subroutines is slower than straight Assembly code, but will likely be satisfactory for most applications.

Enjoy!

```
*---------------------------------------------------------------*
* GPLLNK- A Universal GPLLNK - 6/21/85 - MG                     *
* This routine will work with any GROM library slot since it is *
* indexed off of R13 in the GPLWS.  (It does require Mem Expansion)  *
* This GPLLNK does NOT require a module to be plugged into the  *
* GROM port so it will work with the Editor/Assembler,          *
* Mini Memory (with Mem Expansion), Extended Basic, the Myarc   *
* CALL LR("DSKx.xxx") or the CorComp Disk Manager Loaders.      *
* It saves and restores the current GROM Address in case you want    *
* to return back to GROM for Basic or Extended Basic CALL LINKs      *
* or to return to the loading module.                           *
*                                                               *
*    ENTER: The same way as the E/A GPLLNK   ie: BLWP @GPLLNK   *
*                                                DATA >34       *
*                                                               *
*    NOTES: Do Not REF GPLLNK when using this routine in your code   *
*                                                               *
* 70 Bytes - including the GPLLNK Workspace                     *
*---------------------------------------------------------------*

GPLWS   EQU  >83E0          GPL workspace
GR4     EQU  GPLWS+8        GPL workspace R4
GR6     EQU  GPLWS+12       GPL workspace R6
STKPNT  EQU  >8373          GPL Stack pointer
LDGADD  EQU  >60            Load & Execute GROM address entry point
XTAB27  EQU  >200E          Low Mem XML table location 27
GETSTK  EQU  >166C

GPLLNK  DATA GLNKWS  R7     Set up BLWP Vectors
        DATA GLINK1  R8

RTNAD   DATA XMLRTN  R9     Address where GPL XML returns to us

GXMLAD  DATA >176C   R10    GROM Address for GPL XML (0F 27 Opcode)
        DATA >50     R11    Initialized to >50 where PUTSTK address resides

GLNKWS  EQU  $->18                  GPLLNK's workspace of which only
        BSS  >08     R12-R15        registers R7 through R15 are used

GLINK1  MOV  *R11,@GR4              Put PUTSTK Address into R4 of GPL WS
        MOV  *R14+,@GR6             Put GPL Routine Address in R6 of GPL WS
        MOV  @XTAB27,R12            Save the value at >200E
        MOV  R9,@XTAB27             Put XMLRTN Address into >200E
        LWPI GPLWS                  Load GPL WS
        BL   *R4                    Save current Grom Addres on stack
        MOV  @GXMLAD,@>8302(R4)     Push GPL XML Add on stack for GPL RTurn
        INCT @STKPNT                Adjust the stack pointer
        B    @LDGADD                Execute our GPL Routine
```

```
XMLRTN MOV  @GETSTK,R4          Get GETSTK pointer
       BL   *R4                 Restore GROM address off the stack
       LWPI GLNKWS              Load our WS
       MOV  R12,@XTAB27         Restore >200E
       RTWP                     All Done - Return to Caller


*--------------------------------------------------------------*
* DSRLNK - A Universal Device Service Routine Link - MG        *
*      (uses console GROM 0's DSRLNK routine)                  *
*      (do not REF DSRLNK or GPLLNK when using these routines) *
*      (this DSRLNK will also handle Subprograms and CS1, CS2) *
*                                                              *
*      ENTER: The same way as the E/A DSRLNK ie:  BLWP @DSRLNK *
*                                                 DATA 8       *
*                                                              *
*      NOTES: Must be used with a GPLLNK routine               *
*             Returns ERRORs the same as the E/A DSRLNK        *
*             EQ bit set on return if error                    *
*             ERROR CODE in callers MSB of Register 0 on return*
*                                                              *
* 186 Bytes total - including GPLLNK, DSRLNK and both Workspaces *
*--------------------------------------------------------------*

PUTSTK EQU  >50          Push Grom Add to stack pointer
TYPE   EQU  >836D        DSRLNK Type byte for GPL DSLLNK
NAMLEN EQU  >8356        Device name length pointer in VDP PAB
VWA    EQU  >8C02        VDP Write Address location
VRD    EQU  >8800        VDP Read Data byte location
GR4LB  EQU  >83E9        GPL Workspace R4 Lower byte
GSTAT  EQU  >837C        GPL Status byte location

DSRLNK DATA DSRWS,DLINK1           Set BLWP Vectors

DSRWS  EQU  $                      Start of DSRLNK workspace
DR3LB  EQU  $+7                    R3 lower byte of DSRLNK workspace
DLINK1 MOV  R12,R12       R0       Have we already looked up the LINK address?
       JNE  DLINK3        R1       YES! Skip look up routine

*<<----------------------------------------------------------->>*
* This section of code is only executed once to find the GROM address *
* for the GPL DSRLNK - which is placed at DSRADD and R12 is set to >2000 *
* to indicate that the address is found and to be used as a mask for EQ & CND *
*--------------------------------------------------------------*
       LWPI GPLWS          R2,R3    Else load GPL workspace
       MOV  @PUTSTK,R4     R4,R5    Store current GROM address on the stack
       BL   *R4            R6
       LI   R4,>11         R7,R8    Load R4 with address of LINK routine vector
       MOVB R4,@>402(R13)  R9,R10   Set up GROM with address for vector
       JMP  DLINK2         R11      Jump around R12-R15
       DATA 0              R12      contains >2000 flag when set
       DATA 0,0,0          R13-R15  contains WS, PC & ST for RTWP
DLINK2 MOVB @GR4LB,@>402(R13)       Finish setting up GROM address
       MOV  @GETSTK,R5              Take some time & set up GETSTK pointer
       MOVB *R13,@DSRAD1            Get the GPL DSR LINK vector
       INCT @DSRADD                 Adjust it to get past GPL FETCH instruction
       BL   *R5                     Restore the GROM Address off the stack
       LWPI DSRWS                   Reload DSRLNK workspace
       LI   R12,>2000               Set flag to signify DSRLNK address is set
*<<----------------------------------------------------------->>*
```

```
DLINK3 INC  R14                    Adjust R14 to point to Callers DSR Type byte
       MOVB *R14+,@TYPE            Move it into >836D for GPL DSRLNK
       MOV  @NAMLEN,R3             Save VDP address of Name Length
       AI   R3,-8                  Adjust it to point to PAB Flag byte
       BLWP @GPLLNK                Execute DSR LINK
DSRADD BYTE >Ø3                    High byte of GPL DSRLNK address
DSRAD1 BYTE >ØØ                    Lower byte of GPL DSRLNK address

*------Error Check & Report to Callers RØ and EQU bit ---------------------
       MOVB @DR3LB,@VWA            Set up LSB of VDP Add for Error Flag
       MOVB R3,@VWA                Set up MSB of VDP Add for Error Flag
       SZCB R12,R15                Clear EQ bit for Error Report
       MOVB @VRD,R3                Get PAB Error Flag
       SRL  R3,5                   Adjust it to Ø-7 error code
       MOVB R3,*R13                Put it into Callers RØ (msb)
       JNE  SETEQ                  If its not zero set EQ bit
       COC  @GSTAT,R12             Else test CND bit for Link Error (ØØ)
       JNE  DSREND                 No Error Just return
SETEQ  SOCB R12,R15                Error so set Callers EQ bit
DSREND RTWP                        All Done - Return to Caller
```

TMS9995 Performance: An Introduction for 99/4A Owners

By D.C. Warren

The heart of our 99/4A Home Computer is the TMS99ØØ microprocessor (uP). The 99ØØ was introduced by TI a few years back and represents one of their first generation uPs. Since its introduction, a couple of "next generation" devices have been designed and produced by TI with one of them being the TMS9995 microcomputer (uC). There has been some interest regarding the 9995, so it might be informative to compare its performance to the 99ØØ uP. To do this, let's imagine that we could put a 9995 into a 99/4A. We'll call it the 4B for convenience and use it for performance comparisons with the 4A.

One of the features most commonly desired in going to a new processor is to run it at a higher clock rate than the old processor. If we can run the 4B at a higher clock speed than the 4A, then our software might run faster. The 99ØØ can handle about a 3MHz (4-phase) clock speed internally with the development of these clock signals coming from an external IC (since the 99ØØ is not capable of the chore by itself). The 9995, on the other hand, can handle a 12MHz crystal directly at its clock inputs. At first it appears favorable since the input clock frequency of the 9995 is FOUR times that of the 99ØØ. Unfortunately, the 9995 divides this input frequency by four and runs internally at 3MHz. So, it turns out that both the 99ØØ and 9995 run at the same internal clock frequency, the difference being that the 9995 has the convenience of built-in clock circuitry.

The data bus width of the two processors also differs, with the 99ØØ having a 16-bit bus and the 9995 having an 8-bit bus. The data bus width can be important in the performance of a system because it determines how much information can be passed to and from the processor at any one time (i.e. the wider the bus the faster a system can be). Well, that means that the 4B must make two memory fetches in order to grab a word out of memory whereas the 4A can do the same in one memory fetch. It at first appears that we have taken a

step backwards with the 4B. However, a couple of other factors come into play to balance out performance of the two processors. The 9995 is capable of making a byte memory fetch (no wait states) in just one 3MHz clock cycle. The 99ØØ must take two cycles (no wait states) to make a word (two bytes) memory fetch. Two times one equals one times two and they're even.

Another thing to consider is that the bus in the Peripheral Expansion Box (PEB) is only 8-bits in width. The 99ØØ couldn't grab a complete word from memory expansion in one memory fetch if it wanted to! In fact, one wait state minimum (some devices insert more) is inserted with each byte fetch. That means the 4A has to expend six clock cycles instead of two just to get a word from expansion memory. The trouble spills over to the 4B since it must be (should be) compatible with any card residing in the Expansion Box. Two wait states per byte fetch must be inserted to match the six clock cycles per word fetch of the 4A. One could reduce the number of wait states inserted by the 4B but at the risk of not being compatible with some of the cards in the PEB. So, it appears that the present 4A equipment is part of our performance bottle neck.

If the 9995 runs at the same internal clock frequency as the 99ØØ and both processors take about the same amount of time to fetch a word from memory then what have we gained with the 4B? Well, there are factors other than internal clock frequencies and memory cycle times that are often overlooked which can be important to performance. Three of these factors are processor efficiency, the 9995 instruction pre-fetch and the fact that the 99ØØ has to do a "read before write" memory sequence not required by the 9995.

What does it mean when we say that the 9995 is more efficient than the 99ØØ? For every machine instruction, LI or BLWP for example, known to the processor, there is a microprogram (composed of microcode) inside the processor which executes the instruction. So, after a machine instruction is fetched, the processor decides what instruction it has and executes its own microprogram to accomplish the task defined by the machine instruction. The 9995 can do this on the average using fewer clock cycles per machine instruction than the 99ØØ.

The instruction pre-fetch of the 9995 also helps increase performance by "pre-fetching" the next instruction to be executed by the processor. While the processor is executing one of its microprograms, the pre-fetch is busy going back out to memory and grabbing the next machine instruction. When the processor finishes with the current machine instruction, the next one is already decoded and waiting for it. This saves on some time-consuming memory cycles.

The last area mentioned is the "read before write" performed by the 99ØØ on every write memory cycle. Because of the way the 99ØØ was designed, it is necessary for the processor to do a read memory cycle before it can perform a write memory cycle. We won't go into the why's here but those interested can investigate further by reading page 22 of the TI-99/4A CONSOLE and PERIPHERAL EXPANSION SYSTEM TECHNICAL DATA book. The 9995 does not have to go through the same "read before write" process with every write memory cycle and can, therefore, execute a write memory cycle in less time than the 99ØØ.

Now that we've touched upon some of the major performance differences between the 99ØØ and 9995, let's see what kind of difference it might make in the speed of the 4B. We'll take a piece of code from the key scan routine in the 4A and calculate the speed at which the 99ØØ and the 9995 execute the code. When the console is executing the key scan routine its workspace is in the GPL workspace area and the routine itself is in console ROM. The method of calculating processor instruction speed can be found in the data manual of each

respective processor.

INSTRUCTION	ADDR MODE	9900 CYCLES INSTR	9900 CYCLES ADDR	9995 CYCLES INSTR	9995 CYCLES ADDR
CLR R3	WR	*10,3	0,0	5,4	0,0
SETO R4	WR	10,3	0,0	5,4	0,0
STCR R4,5	WR	42,4	0,0	28,8	0,0
SRL R4,9	WR	30,3	0,0	17,6	0,0
JOC JSCAN	--	8,1	0,0	4,2	0,0
MOVB @H1D(R5),@R3LB	INDX	14,4	8,2	4,4	5,4
	SYMB		8,1		2,2
SLA R4,1	WR	14,3	0,0	9,6	0,0
AI R4,BBJOY	--	14,4	0,0	8,8	0,0
	TOTALS	142,25	16,3	80,42	7,6
GRAND TOTALS		158,28		87,48	

* First operand=clock cycles
and second operand=memory accesses
if there are wait states.

We find that the 9900 must expend 158 cycles with 28 additional memory cycles per wait state to execute the above code segment while the 9995 expends 87 cycles and 48 cycles respectively on the same segment. Now we want to translate our results into total time and do a comparison between the two processors. The conversion formulas are:

```
==========================================================================
 9900: T=tc(C+(W*M))        T-->Total time in microseconds
                           tc-->Clock period in microseconds
                            C-->Clock cycles
#1)                         W-->Number of wait states inserted
                            M-->Memory accesses

                              NOTE: No wait states in SRAM and console ROM!

      T=0.333(158+(0*28))=52.614uS
 ==========================================================================
 9995: T=tc[C1+C2+W*(XM1+XM2)]   T-->Total time in microseconds
                                tc2-->CLKOUT clock period in microseconds
                                 C1-->Clock cycles
                                 C2-->Clock cycles for operand address derivation
 2)                               W-->Wait states
                                XM1-->Off chip memory cycles
                                XM2-->Off chip memory cycles for operand address
                                      derivation

      T=0.333[80+7+0*(42+6)]=28.971uS
==========================================================================

     Taking a ratio:  52.614-28.971
#3)                   -------------*100=81.61%
                          28.971

==========================================================================
```

So, the 4B can execute the above code from the key scan routine about 80% faster than the 4A. On the average, this is representative of most of the code

executed from console ROM (GPL interpreter, key scan, interrupt routine, etc.).

Overall, we see that the 4B can run programs at an increased speed compared to the 4A. There are other things to consider, however, before running out and trying to put a 9995 into the 99/4A. The internal memory and CRU structure of the 9995 is different than that of the 4A. The 9995 has an internal timer and on-chip memory which may affect some existing 4A software. It also reserves CRU bits for its own use, so that in evaluating possible configurations, one might consider the possibility of a system override of a particular card, as I have not investigated the consequences of the CRU structure.

Major changes to the 4A console would also be in order. The fact that the two processors have different data bus widths implies a major modification. There are also timing signal changes, etc. which would have to be dealt with.

If one is serious about such a project, there is another alternative which may be even more attractive. The next set of processors introduced by TI after the 9995 is the 99ØØØ family. Technologically, the 99ØØØ's are an improvement over the 9995 in just about every way (The significantly higher price of the 99ØØØ's is one possible explanation of why 99ØØØ's have apparently not yet been considered a commercially viable option in today's price-conscious marketplace.). The 99ØØØ's do not have any special memory mapping or CRU reservations and, like the 99ØØ, are on a 16-bit bus. The performance increase over the 4A could be more than double that of the 4B we discussed (maybe more). Someone may wish to investigate that possibility some more.

In this article, I've tried to give a brief performance overview of the TMS9995 uC chip produced by TI. I hope that comparing its operation to the more well-known TMS99ØØ-based 99/4A was interesting as well as informative.

Editor: The above article discusses only one of many possible implementations of a TMS9995. Other configurations might present advantages or problematic ramifications not covered herein. As I have stated before, applications dictate hardware. Whether you have an interest in using a TMS9995 is your decision. I simply hope this article will allow readers the basis for a portion of the insight required to make such an evaluation.

Richard Mitchell

New Software

Public Domain

Program: MAX/RLE
Author: Travis Watford
Availablility: Communications networks and user groups
Significance: Uses RLE (Run Length Encoded) standard format to provide break-through interface with graphics from other computers.

Fairware

Program: RAG Macro Assembler
Author: R.A. Green, 1Ø32 Chantenay Drive, Gloucester, Ontario, Canada K1C 2K9.
Availability: Author, user groups. 2 disks. $15 suggested to author.
Significance: Many useful features plus user-extensible macro facility. Source code for several programs included in package.

Publication

Program: XXB
Author: Barry Traver, et. al.
Availability: *GENIAL TRAVelER*, 835 Green Valley Drive, Philadelphia, PA 19128 (by subscription, a "diskazine")
Significance: First major simplified Assembly interface with Extended BASIC.

Commercial

Program: DISkASSEMBLER™
Author: Tom Freeman
Availability: Millers Graphics, 1475 W. Cypress Ave., San Dimas, CA 91773. $19.95 plus shipping and handling.
Significance: Very quick and accurate disassembly from disk or memory. Revise, de-bug or move programs. Easy to use. Most programs re-assemble with little or no further work.

BYTEMASTER ORDER FORM

The Smart Programmer

- |_| $18.00 U.S. AND CANADA FIRST CLASS
- |_| $15.00 U.S. THIRD CLASS (no back issues)
- |_| $20.00 FOREIGN SURFACE (no back issues)
- |_| $32.00 FOREIGN AIRMAIL
- |_| $ 1.75 U.S. JUNE 1986 ISSUE
- |_| $ 2.75 FOREIGN JUNE 1986 ISSUE

Super 99 Monthly

- |_| $18.00 Complete set of back issues
- |_| $ 1.00 Back issues - ea. (U.S. Third Class)
- |_| $ 1.50 Back issues - ea. (Canada and U.S. First Class)
- |_| $ 2.50 Back issues - ea. (Foreign Air Mail)
- |_| $12.00 Programs on disk (non-FORTH)
- |_| $15.00 Super 99 Handicapper (req. XB, 32K, Disk, Printer)

Name_____

Address_____

City_____

State_____

Zip Code_____

Country_____

Payments accepted by check or money order in U.S. Funds, coded for processing through the U.S. Federal Reserve Banking System. No billings or credit sales. Dealer inquiries invited. Discounts available on quantity orders.

The Smart Programmer 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 Editor of *The Smart Programmer*.

Editor Richard M. Mitchell

Staff
Craig Miller
Charles M. Robertson
Mariusz Stanczak
Steven J. Szymkiewicz, MD
Barry A. Traver
D.C. Warren

Bytemaster Computer Services
171 Mustang Street
Sulphur, LA 70663

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