
* RXB MANUAL *

* (C) 1 JAN 2012 *
* by *
* Richard Lynn Gilbertson *

IMPORTANT NOTICE REGARDING PROGRAMS AND MANUAL MATERIALS

The following should be read and understood before purchasing and/or using RXB. Richard Lynn Gilbertson does not warrant that the program and accompanying book materials will meet the specific requirements of the consumer, or that the programs and manual will be free from error. The consumer assumes complete responsibility for any decision made or actions taken based on information obtained using these programs and manual materials. Any statements made concerning the utility of Richard Lynn Gilbertson's programs or manual materials are not to be construed as express or implied warranties.

RICHARD LYNN GILBERTSON MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

REGARDING THESE PROGRAMS OR MANUAL MATERIALS OR ANY PROGRAMS DERIVED THEREFROM AND MAKES SUCH MATERIALS AVAILIABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL RICHARD LYNN GILBERTSON BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGE IN CONNECTION WITH OR ARRISING OUT OF THE PURCHASE OR USE OF THESE PROGRAMS OR MANUAL MATERIALS, AND THE SOLE AND EXCLUSIVE LIABILITY OF RICHARD LYNN GILBERTSON, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE RXB PRODUCT. MOREOVER, RICHARD LYNN GILBERTSON SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER AGAINST THE USER OF THESE PROGRAMS OR MANUAL MATERIALS BY ANY OTHER PARTY.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

INIT.....	SUBPROGRAM.....	ASSEMBLY.....	I1
INVERSE.....	SUBPROGRAM.....	SCREEN.....	I2
IO.....	SUBPROGRAM.....	TSM9901 CHIP...	I3
ISROFF.....	SUBPROGRAM.....	INTERUPTS.....	I12
ISRON.....	SUBPROGRAM.....	INTERUPTS.....	I13
JOYST.....	SUBPROGRAM.....	JOYSTICKS.....	J1
KEY.....	SUBPROGRAM.....	KEYBOARD.....	K1
LIST.....	COMMAND.....	DISK.....	L1
LOAD.....	SUBPROGRAM.....	DISK/ASSEMBLY..	L2
MKDIR.....	SUBPROGRAM.....	DISK.....	M1
MOTION.....	SUBPROGRAM.....	SPRITE.....	M2
MOVE.....	COMMAND.....	LINES.....	M3
MOVES.....	SUBPROGRAM.....	MEMORY. (ALL)...	M4
NEW.....	SUBPROGRAM or COMMAND...	MEMORY. (XB)...	N1
ONKEY.....	SUBPROGRAM.....	KEYBOARD.....	O1
PEEKG.....	SUBPROGRAM.....	GROM.....	P1
PEEKV.....	SUBPROGRAM.....	VDP.....	P2
POKEG.....	SUBPROGRAM.....	GRAM.....	P3
POKER.....	SUBPROGRAM.....	VDP REGISTERS..	P4
POKEV.....	SUBPROGRAM.....	VDP.....	P5
PROTECT.....	SUBPROGRAM.....	DISK.....	P6
QUITOFF.....	SUBPROGRAM.....	KEYBOARD.....	Q1
QUITON.....	SUBPROGRAM.....	KEYBOARD.....	Q2
RENAME.....	SUBPROGRAM.....	DISK.....	R1
RES.....	COMMAND.....	LINES.....	R2
RMDIR.....	SUBPROGRAM.....	DISK.....	R3
RMOTION.....	SUBPROGRAM.....	SPRITE.....	R4
SCSI.....	SUBPROGRAM.....	SCSI CONTROLLER	S1
SECTOR.....	SUBPROGRAM.....	DISK.....	S2
SIZE.....	SUBPROGRAM or COMMAND...	MEMORY (ALL)...	S3
SWAPCHAR.....	SUBPROGRAM.....	SCREEN.....	S4
SWAPCOLOR.....	SUBPROGRAM.....	SPRITE.....	S5
USER.....	SUBPROGRAM.....	DISK.....	U1
VCHAR.....	SUBPROGRAM.....	SCREEN.....	V1
VERSION.....	SUBPROGRAM.....	GROM RXB.....	V2
VGET.....	SUBPROGRAM.....	SCREEN.....	V3
VPUT.....	SUBPROGRAM.....	SCREEN.....	V4
XB.....	SUBPROGRAM.....	DEVICE.....	X1
XBPGM.....	SUBPROGRAM.....	DISK.....	X2
REA DOCS.....			REA1
AMS DOCS.....			AMS1

This is a copy of the RXB title screen:

FIRMWARE CONTROL

```
*****
* VERSION = 2012 *
*****
*      R X B      *
*                  *
*      creator    *
*                  *
* Rich Gilbertson *
*****
```

>> press ===== result <<

ANY KEY = DSK#.LOAD

ENTER = DSK#.UTIL1

(COMMA) , = DSK#.BATCH

SPACE BAR = RXB COMMAND MODE

(PERIOD) . = EDITOR ASSEMBLER

NOTE: 0 (ZERO) defaults to WDS1.LOAD or after pressing

ENTER defaults to WDS1.UTIL1

This is a explanation of the keys of the title screen:

ANY KEY = DSK#.LOAD

This asks for which drive number the LOAD program is on. Any key means just what it says. The flashing # symbol indicates this option is waiting for a response. If after 5 seconds no key is pressed a search for LOAD begins. Drives 1 to 9 are searched for the XB LOAD program. If the LOAD program cannot be found, like old XB it goes to the XB command mode eventually after many searches. Pressing 0 (Zero) will default to WDS1.LOAD

ENTER = DSK#.UTIL1

After pressing ENTER the number symbol flashes indicating this option has been selected. Any key will then select the Drive to load the UTIL1 program from. If after 5 seconds no key is pressed a search for UTIL1 begins. Drives 1 to 9 are searched for the UTIL1 program. This will only load a EA5 program image file named UTIL1. Pressing 0 (Zero) will default to WDS1.UTIL1

(COMMA) , = DSK#.BATCH

After pressing the , (COMMA) key the number symbol flashes indicating this option has been selected. Any key will then select the Drive to load the BATCH control file from. If after 5 seconds no key is pressed a search for DSK1.BATCH begins. If the control file BATCH cannot be found on Drive 1 it defaults to XB command mode. See CALL USER for more information on BATCH CONTROL FILES.

SPACE BAR = RXB COMMAND MODE

Pressing the SPACE BAR results in XB command mode.

(PERIOD) . = EDITOR ASSEMBLER

Pressing the . (PERIOD) key will switch to EDITOR ASSEMBLER menu. Pressing the . (PERIOD) key while in EDITOR ASSEMBLER will switch back to RXB.

(ZERO) 0 = WSD1.LOAD or WSD1.UTIL1

Pressing the 0 (ZERO) key will start a WSD1.LOAD to execute from hard drive 1. If the root directory has a LOAD program. If a SCSI drive exists at the lowest CRU address it will load else a Myarc HFDC is used.

DISK AND HARD DRIVE ACCESS

CALL CAT catalog disk or hard drives.
 CALL DIR catalog disk or hard drives. (Requested duplicate of CAT)
 CALL FILES same as disk controller version but executes new after.
 CALL FCOPY copy's a file from device to device.
 CALL PROTECT protects or unprotects a file.
 CALL RENAME renames a file or directory.
 CALL MKDIR makes a directory on hard drives or names disks.
 CALL RMDIR deletes a directory on hard drive.
 CALL CUTDIR deletes a directory and all sub-directories.
 CALL SECTOR reads or writes disk or hard drive sectors.
 CALL XBPGM not only runs XB programs but does a CALL FILES first.
 CALL SCSI retrieves the scuzzy device ID codes. The info will contain the device company name, version number, revision number and size. Example XB program provided in SCSI documents.

BATCH FILE SYSTEM:

CALL USER overrides the normal edit mode by allowing a DV80 file to take control. This allows conversions from DV80 to XB program or DV80 to XB MERGE format or loading files, resequencing them, and saving or merging or adding lines through another DV80 file. All variables used through CALL USER are not affected so from a running program more lines or variables can be added to the size of the program without losing anything. Of course the RUN command will as always clear all variables before the program is run, this feature can be turned off with a CALL LOAD. (PRESCAN OFF)

As the USER subprogram can override the Editor many features can be bypassed. Example:

NEW	cr
OLD DSK1.XBPROGRAM	cr
RES 11,3	cr
MERGE "DSK1.MERGEPM"	cr
SAVE "DSK1.NEWPROGRAM"	cr
RUN	cr

The above is a good example of a DV80 Batch file for RXB. Note that there must be a CHR\$(13) or Carriage Return after every input line. If not then RXB assumes the it is the same line. But even that is not much of a problem as RXB allows 21 lines of input per program line. You can make them even longer if you want.

INPUT/OUTPUT ACCESS:

CALL IO controls the 9901 CRU chip. Sound lists can be played independently of current status. (i.e. type in a program while playing music from VDP/GROM.) Control Register Unit can turn on/off single bits of CRU address bus. (i.e. cards/chips)
 Cassette direct bus control. (i.e. no menu input/output, verify)

The REDO (FCTN 8) no longer exists in RXB. USER needed a buffer that would not be molested or modified by CALL LINK, CALL LOAD or routines that need a buffer and usually use the same area that USER previously used. So to update and eliminate questions of compatibility the USER buffer was installed in place of the Edit recall buffer (REDO). The REDO key was not considered to be of much use anyway as the Crunch Buffer is 163 tokens long and in non-tokenized form the Edit recall buffer is only 152 bytes long. That is why when REDO is pressed only part of the line last typed in was recalled to screen. Additionally COPY lines, and MOVE lines commands can do the same thing as REDO could, so not much of anything is lost because it is assumed a TEXT EDITOR will be used to create programs in RXB then use CALL USER.

PROGRAM DEVICE NAMES ACCESS:

New access names established as devices are now available. By using any TRUE DSR (Device Service Routine) you may now access the Editor Assembler main menu by typing 'EA' within Basic or RXB. Example: RUN "EA" or OLD EA or DELETE "EA"

You may also access RXB from Editor Assembler or Basic or even another cartridge. Example: OLD XB or DELETE "XB" from Basic. At any Editor Assembler device prompt type 'XB' then enter.

FOR ASSEMBLY LANGUAGE PROGRAMMERS:

CALL MOVES is a new command that is a GPL command converted and added to RXB to give total control over every type of memory with in the TI-99/4A. MOVES works with address or strings to copy, over-write or move blocks of memory of any type of memory. RAM, VDP, GROM, GRAM, and ROM can be accessed by CALL MOVES.

RXB TO ASSEMBLY DIRECT ACCESS BY ADDRESS:

EXECUTE is much faster than the traditional LINK routine built into XB. The main problem with LINK is it checks everything and pushes everything onto the VDP stack. After getting to Assembly it pops everything off the stack for use or pushes what is to be passed to XB onto the stack. EXECUTE on the other hand just passes a address to a 12 byte Assembly program in Fast RAM and RTWP ends the users program. A LINK will use up 6 bytes for the name, 2 bytes for the address and wastes time checking things. The advantage to EXECUTE is you use LOAD or MOVE or MOVES to place the values needed directly into the registers then do it. EXECUTE uses less space, is faster, and is easy to debug.

AMS SUPPORT ROUTINES:

The AMS now has support routines built into RXB. CALL AMSMAP will turn the AMS mapper on. CALL AMSPASS will turn the AMS mapper back to pass mode. CALL AMSON will turn on the read/write lines of the mapper. CALL AMSOFF will turn off the read/write lines. With these commands pages of memory can be written with a CALL LOAD or read with a CALL PEEK. Also little known by users is the fact that if a RXB program is smaller than 10K it can be run from VDP not the upper 24K. So paging the upper or lower memory of the AMS is possible from RXB programs. RXB AMS SUPPORT USES NO ASSEMBLY OR CALL LINKs. That means up to 1meg of lower 8K pages or upper 24K pages from RXB. That is impossible to do from XB as you have to load your normal support somewhere.

GPL is where all the support routines are stored in RXB so not one byte is wasted on assembly support. That also means not one byte of AMS memory is wasted on control routines.

Speaking of control CALL AMSINIT initializes the mappers and switches the AMS to map mode. CALL AMSBANK switches 4K pages in the lower half and upper half of the lower 8K. AMSBANK is a built in AMS memory bank switcher for the lower 8K.

INTERUPT SERVICE ROUTINE CONTROL:

ISR (Interrupt Service Routine) like MOUSE or Screen dumps or any special program like XB Packer use a ISR. The problem with these programs is unless they are written to work with new devices, a lock-up occurs. EXAMPLE: running a mouse routine and XB Packer. They were never made to work together. RXB now has a handle on this. CALL ISROFF turns off the interrupt and saves the address for turning it back on. CALL ISRON restarts the interrupt. As several pages of the AMS can be used with interrupts a whole new world of programming is now possible. NO ASSEMBLY IS USED OR CALL LINKs. Absolute compatibility.

LOWER 8K PROGRAM IMAGE FILE LOADER AND SAVER

Hidden loaders were created to overcome the slow loading speed of CALL LOAD. The disadvantage of a hidden loader is it can only load one assembly support program at a time. BLOAD loads program image files of the lower 8K, and BLOAD can load as many times as needed within one RXB program. BSAVE is the opposite and creates the program image files of the lower 8K support routines. Lastly loading 200K into the AMS card is easy with BLOAD and AMSBANK. A simple loop can load each AMS 8K bank with AMSBANK and BLOAD loads 8K at a time into the lower 8K.

Format CALL AMSBANK(page-number,page-number)

 CALL AMSBANK(numeric-variable,numeric-variable)

Description

The AMSBANK command will only work with a AMS memory card. AMSBANK turns on the read/write lines of AMS mapper registers stores the first value into the mapper register that sets low half of lower 8K, then stores the second value into the mapper register that sets high half of lower 8K. If the page-number is less then 0 or larger than 239 a BAD VALUE error results. If page-number is larger than the AMS card size allows * AMS BANK NUMBER ERROR * results. Neither error will affect previously loaded pages. No lock-up will result.

AMSBANK breaks the lower 8K into two halves of 4K. The lower 4K is Hex >2000 to >2FFF and the upper is Hex >3000 to >3FFF so switching lower 8K requires two 4K pages to be used. This is the only disadvantage to AMSBANK, but flexibility does result. Less wasted pages results in more memory available. The formula is: AMS SIZE/4-16=pages. EXAMPLE: 256/4-16 is 48 pages.

In other words 0 to 15 pages would be 16 pages for use. The odd ball numbering scheme of AMSBANK results from pages 0 to 15 not being used in MAP mode. AMSBANK creates it's own numbers of pages 0 to 239 by starting actually at page 16 of the AMS. That would be page 0 of AMSBANK. This lay out leaves open 8 4K pages for PASS mode, and 8 4K pages for future use. See docs MANUAL-AMS for additional information.

Programs

```

This sets up & starts map mode| >100 CALL AMSINIT
This turns on the AMS mapper. | >110 CALL AMSON
This reads low half 8K page.  | >120 CALL PEEK(16388,L)
This reads high half 8K page. | >130 CALL PEEK(16390,H)
This shows pages used.        | >140 PRINT "LOW";L;"HIGH";H
This loads a assembly program.| >150 CALL LOAD("DSK1.CHAR")
This changes low/high 8K pages| >160 CALL AMSBANK(16,17)
This loads a assembly program.| >170 CALL LOAD("DSK1.DUMP")
This changes low/high back.   | >180 CALL AMSBANK(L,H)
This uses a routine in CHAR.   | >190 CALL LINK("CHAR")
This changes low/high again.  | >200 CALL AMSBANK(16,17)
This uses a routine in DUMP.   | >210 CALL LINK("DUMP")
                                |

```

The above example program shows one RXB program using two assembly programs with links for both. Thus only 16K of the AMS was used. 1024K would be 120 assembly support programs Compatibility of most software assured in RXB AMS support.

Format CALL AMSINIT

Description

The AMSINIT command will only work with a AMS memory card. AMSINIT turns on the read/write lines of AMS mapper registers stores pages 0 to 15 into mapper registers, then turns on MAP mode. Essentially PASS mode and MAP mode are the same in that the mapper registers are exactly the same in both modes. That means in both modes the same memory is used. This would make the AMS of little use so LOAD is used to change mapper registers and switch pages. But if upper memory is switched from a XB program the page that just did this is switched out and a lock-up occurs. On the other hand switching pages in the lower 8K presents very few problems. So see AMSBANK. AMSINIT can also be used like AMSPASS but stays in MAP mode. See docs MANUAL-AMS for examples of memory maps.

Programs

```

This sets up & starts map mode| >100 CALL AMSINIT
This turn on the AMS mapper.  | >110 CALL AMSON
This reads low half 8K page.  | >120 CALL PEEK(16388,L)
This reads high half 8K page. | >130 CALL PEEK(16390,H)
This shows pages used.        | >140 PRINT "LOW";L;"HIGH";H
This changes low half 8K page.| >150 CALL LOAD(16388,16)
This changes high half 8K page| >160 CALL LOAD(16390,17)

This sets up & starts map mode| >200 CALL AMSINT
This turns on the AMS mapper. | >210 CALL AMSON
This reads low half 8K page.  | >210 CALL PEEK(16388,L)
This reads high half 8K page. | >220 CALL PEEK(16390,H)
This shows pages used.        | >230 PRINT "LOW";L;"HIGH";H
This changes low/high 8K pages| >240 CALL AMSBANK(16,17)
|

```

In the above program 100 to 160 and 200 to 220 do the same thing. Using CALL AMSBANK is more effective than using a CALL LOAD to change mapper registers as a CALL AMSON is needed to turn on the mappers to be written or read. When using the AMS with RXB it is suggested to always use CALL AMSBANK to change pages as it is less complicated and faster with less wasted program lines. EXAMPLE:

```

This line sets up AMS memory. | >100 CALL AMSINIT
This line loads AMS pages 0,1. | >110 CALL AMSBANK(0,1)
This line loads a program.     | >120 CALL LOAD("DSK1.TEST")
This line loads AMS pages 2,3  | >130 CALL AMSBANK(2,3)
This line loads a program.     | >140 CALL LOAD("DSK1.JUNK")

```

Format CALL AMSMAP

Description

The AMSMAP command will only work with a AMS memory card. MAP MODE on the AMS card means the mapper registers are turned on so they can be changed. But even with the mapper on unless the read/write lines are on no mappers will appear to be at the DSR address. AMSON turns on read/write mapper registers. Then a LOAD or AMSBANK can change the memory pages. See docs MANUAL-AMS for examples of memory maps. Also run AMS-TEST or AMS-SAVE or AMS-LOAD programs.

Programs

This turns on map mode.	>100 CALL AMSMAP
This turns on read/write.	>110 CALL AMSON
This fetches map register 2.	>120 CALL PEEK(16388,BYTE)
This turns off read/write.	>130 CALL AMSOFF
This turns on pass mode.	>140 CALL AMSPASS
This prints the page from map	>150 PRINT "Register 2 PAGE#"
mode in register 2.	;BYTE

The above program will print out whatever AMS page is presently stored in AMS map register 2.

It is recommended that CALL AMSMAP only be used to check AMS pages with CALL PEEK. CALL AMSBANK is much more easy to use to manage AMS memory.

Format

CALL AMSOFF

Description

The AMSOFF command will only work with a AMS memory card. The read/write lines to the AMS mapper registers are turned off so they will not be changed. Any PEEK or LOAD to the DSR space will be zero after the AMSOFF command. They can't be read/written to. See docs MANUAL-AMS for examples of memory maps. Also run AMS-TEST or AMS-SAVE or AMS-LOAD programs.

Programs

```
This sets up & starts map mode| >100 CALL AMSINIT
This turns on read/write.      | >110 CALL AMSON
This fetches map register 2.   | >120 CALL PEEK(16388,BYTE)
This turns off read/write.     | >130 CALL AMSOFF
This turns on pass mode.       | >140 CALL AMSPASS
This prints the page from map  | >150 PRINT "Register 2 PAGE#"
mode in register 2.           | ;BYTE
                               |
```

The above program will print out initialized AMS page 2 in register 2.

It is recommended that CALL AMSOFF only be used to protect the AMS mapper registers from being molested by programs that could access the AMS. CALL AMSBANK is more easy to use to manage AMS memory as AMSBANK always turns off the AMS read/write registers like AMSOFF does. Instead use AMSBANK.

Format

CALL AMSON

Description

The AMSON command will only work with a AMS memory card. The read/write lines to the AMS mapper registers are turned on so they can be changed. Any PEEK or LOAD to the DSR space can then be used to change the mapper registers or read them. See docs MANUAL-AMS for examples of memory maps. Also run AMS-TEST or AMS-SAVE or AMS-LOAD programs.

Programs

This sets up & starts map mode		>100 CALL AMSINIT
This turns on read/write.		>110 CALL AMSON
This loads 9 in map register 2		>120 CALL LOAD(16388,9)
This turns off read/write.		>130 CALL AMSOFF
This loads values in lower 8K.		>140 CALL LOAD(8192,1,2,3,4)
This turns on pass mode.		>150 CALL AMSPASS
This peeks values in lower 8K.		>160 CALL PEEK(8192,A,B,C,D)
This prints values.		>170 PRINT A;B;C;D
This turns on map mode.		>180 CALL AMSMAP
This turns on read/write.		>190 CALL AMSON
This loads 2 in map register 2		>200 CALL LOAD(16388,2)
This turns off read/write.		>210 CALL AMSOFF
This peeks values in low page.		>220 CALL PEEK(8192,A,B,C,D)
This prints values.		>230 PRINT A;B;C;D

It is recommended to use CALL AMSON only for when a CALL PEEK is used to check a mapper register value. CALL AMSBANK manages AMS mapping much better.

Format CALL AMSPASS

Description

The AMSPASS command will only work with a AMS memory card. PASS MODE on the AMS card means the mapper registers are not on. This is the normal mode of the AMS. No extra memory is available or used. This renders the AMS like a normal 32K card. See docs MANUAL-AMS for examples of memory maps. Also run AMS-TEST or AMS-SAVE or AMS-LOAD programs.

Programs

This sets up & starts map mode	>100 CALL AMSINIT
This turns on read/write.	>110 CALL AMSON
Load 22 into map register 2.	>120 CALL LOAD(16388,22)
This turns off read/write.	>130 CALL AMSOFF
This turns on pass mode.	>140 CALL AMSPASS

AMSPASS is mainly used to turn off AMS or protect the AMS pages from being used or to behave like a normal TI99/4A when the AMS is not being used.

Please note that AMSBANK does not use the AMSPASS memory so this area stays protected. This is by design in RXB.

```

Format                      RUN "BASIC"

                            DELETE "BASIC"

                            CALL XBPGM("BASIC")

                            CALL CAT("BASIC")

                            OLD BASIC

                            CALL BASIC
  
```

Description

The BASIC DSR (Device Service Routine) allows access to the TI BASIC . The access will work only if the DSR is the GPLDSR or LINK DSR. In other words, a DSR that acknowledges any type of DSR in RAM, ROM, GROM, GRAM, or VDP. Most DSR's only accept DSK or PIO. Others like the SAVE or LIST commands will only work with a program in the memory first. Still others like CALL LOAD("EA") must have the CALL INIT command used first.

Keep in mind that if it does not work, the problem is the DSR your using. Almost all DSR's today only acknowledge the ROM or RAM DSR's. As the BASIC DSR is in GROM/GRAM it seems a bit short sighted on the part of most programmers to use cut down versions of a DSR. Please discourage this practice as it is a disservice to us all.

Programs

The program at the right will go to the BASIC prompt.	>100 CALL XBPGM("BASIC")
This line asks for a string.	>100 INPUT A\$
This line uses the string and if you type BASIC then enter will switch to BASIC.	>110 DELETE A\$
This line will switch to the BASIC.	>CALL BASIC
This line shows even lower case works also.	>CALL EAPGM("basic")

Format

CALL BEEP

Description

The BEEP command produces the same sound as the ACCEPT or INPUT, or BEEP as in DISPLAY options.

See EXTENDED BASIC MANUAL pages 47, 48, 49, 77, 78.

Programs

```
The program to the right will | >100 CALL BEEP
will produce a beep sound.    |
Show request.                  | >110 PRINT "YNyn ?"
Key press request.             | >120 CALL KEY("YNyn",0,K,S)
```

The above program will BEEP then wait for a key and only accept Y N y n from CALL KEY into K.

Format CALL BIAS(numeric-variable,string-variable
 [,...])

Description

The BIAS command adds 96 to all characters in the string or subtracts 96 from all characters in the string. If numeric variable is 0 then it subtracts the XB screen bias of 96 from the characters, if the numeric variable is not 0 then it adds the XB screen bias of 96 to all the characters in the string. ONLY A STRING VARIABLE IS ALLOWED.

The XB screen bias only affects characters read or written to the screen. See PEEKV, and POKEV.

Programs

The program to the right will		>100 CALL MOVES("V\$",255,511
load X\$ with 255 characters		,X\$)
off the screen. But will not		
be readable due to a bias.		
The bias is now subtracted		>110 CALL BIAS(0,X\$)
from the string printed.		>120 PRINT X\$

The above program copies 255 bytes from screen address 511 (511=15 rows plus 31 columns) into string X\$. Then BIAS removes 96 from each byte in string X\$. Finally X\$ is shown on screen by PRINT X\$

Format CALL BLOAD("access-name")

 CALL BLOAD(string-variable)

Description

The BLOAD subprogram loads ONLY program image files created by BSAVE. BLOAD is the opposite of BSAVE. BLOAD is a faster version of CALL LOAD. BLOAD has the speed of a hidden loader and is much easier to use. BLOAD only loads into lower 8K.

Unlike CALL LOAD the BLOAD and BSAVE subprogram will work without CALL INIT being used first. Remember to turn on the interrupts if the program has them. Or the program support will not work. See ISROFF and ISRON.

NOTE: 8K of VDP memory MUST be free for BLOAD to function or a memory full error will result. Always place the BLOAD command at the top of the RXB program.

Programs

This line loads a previously saved program image file.		>100 CALL BLOAD("DSK2.MOUSE")
This line turns on the mouse (program would continue here)		>110 CALL LINK("MSON")
This line load a previously saved program image file.		>100 CALL BLOAD("DSK1.DUMP")
This line turns on interrupt within program.		>110 CALL ISRON(16384)
This line links to support.		>120 CALL LINK("DUMP")

Options

AMS users will find this a easy way to load RXB AMS support.
EXAMPLE:

```

>100 CALL AMSINIT
>110 FOR L=0 TO 15 STEP 2
>120 CALL AMSBANK(L,L+1)
>130 CALL BLOAD("DSK1.BANK"&STR$(L/2))
>140 NEXT L
>150 CALL XBPGM("DSK1.MAINPROGRAM",1)

```

The above program would load RXB AMSBANK banks 0 to 15 into AMS memory from files named BANK0 to BANK7 on disk 1. Then would set CALL FILES 1 and RUN "DSK1.MAINPROGRAM" with 64K of Assembly support for XB.

See AMSINIT, AMSBANK, ISROFF, ISRON, EXECUTE, and MOVES.

Format

CALL BSAVE("access-name")

CALL BSAVE(string-variable)

Description

The BSAVE subprogram saves ONLY program image files to be used for BLOAD. BSAVE is the opposite of BLOAD. BSAVE has the speed of a hidden loader without the hassle. BSAVE saves ONLY lower 8K program image files for ONLY BLOAD to use.

Unlike CALL LOAD the BLOAD and BSAVE subprogram will work without CALL INIT being used first.

To save a program with hidden loaders just break program after loading is complete and type CALL BSAVE("DSK#.NAME") Remember to check for interrupts or the program will not work after loading with BLOAD. See ISRON and ISROFF.

NOTE: 8K of VDP memory MUST be free for BSAVE to function or a memory full error will result. Always place the BSAVE command at the top of the RXB program.

Programs

Initialize lower 8K.	>100 CALL INIT
Load the assembly support.	>110 CALL LOAD("DSK1.MSETUPO")
Load the assembly support.	>120 CALL LOAD("DSK1.HDSR")
Turn on the mouse setup.	>130 CALL LINK("MSETUP")
BSAVE the whole thing.	>140 CALL BSAVE("DSK2.MOUSE")
Procedure for hidden loaders.	
Loads program on disk 1	>CALL XBPGM("DSK1.LOAD")
Break program.	PRESS FCTN 4 to break program.
Get address of interrupts.	>CALL ISROFF(I)
See if they are on.	>PRINT I
Save the program to disk.	>CALL BSAVE("DSK2.EXAMPLE")

Options

AMS users will find this a easy way to save RXB AMS support.

EXAMPLE:

```

>100 CALL AMSINIT
>110 FOR L=0 TO 15 STEP 2
>120 CALL AMSBANK(L,L+1)
>130 CALL BSAVE("DSK1.BANK"&STR$(L/2))
>140 NEXT L

```

The above program would save RXB AMSBANK banks 0 to 15 into 8 program image files named BANK0 to BANK7 on disk 1.

See AMSINIT, AMSBANK, ISROFF, ISRON, EXECUTE, and MOVES.

Format	BYE
	CALL BYE

Description

The BYE command is the same as the BYE command in the EXTENDED BASIC MANUAL page 54. The BYE command ends the program and returns the system to a reset. BYE will close all open files before exiting to a reset.

Command

May only be used from command | >BYE
mode.

Programs

May only be used in program mode.	>100 CALL BYE
The INPUT asks for a Y to go on, if not the loop forever.	>110 INPUT "END PROGRAM":A\$
Must be a Y so reset system.	>120 IF A\$<>"Y" THEN 110
	>130 CALL BYE

CALL BYE

CALL CLSALL

CALL CHAR(ALL,pattern-identifier[,...])

CALL CHARSETALL

CALL COINC(#sprite,#sprite,tolerance,numeric-variable[,...])

CALL COLOR(ALL,foreground,background[,...])

CALL DISTANCE(#sprite,#sprite,numeric-variable[,...])

CALL FILES(number)

CALL GCHAR(row,column,numeric-variable[,...])

CALL HCHAR(row,column,character-code,repitition[,...])

CALL JOYST(key-unit,x-return,y-return[,...])

CALL KEY(key-unit,return-variable,status-variable[,...])

CALL KEY(string,key-unit,return-variable,status-variable[,...])

CALL MOTION(ALL,row-velocity,column-velocity[,...])

CALL NEW

CALL ONKEY(string,key-unit,return-variable,status-variable)
GOTO line-number[,...]

CALL SIZE

CALL VCHAR(row,column,character-code,repitition[,...])

CALL VERSION(numeric-variable)

CALL XBPGM(path-filename,file-number)

```

Format      CALL CAT("#"[,...])

             CALL CAT("DSK#."[,...])

             CALL CAT("DSK.DISKNAME."[,...])

             CALL CAT(string-variable[,...])

             CALL CAT(number[,...])

             CALL CAT(numeric-variable[,...])

             CALL CAT(ASC II value[,...])

```

Description

The CAT command catalogs the disk drive indicated by the # which can be 1 to z or by path name. The path name may be up to 30 characters long. A numeric variable or number can be used for drives 1 to 9 or if higher then it is assumed that the numeric-variable or number is a ASCII value between 30 to 255. This allows a catalog of a RAM-DISK designated by letters or control characters. RXB CAT can be used from program mode or command mode.

Also CAT can catalog up to 32 drives in one command.

The SPACE BAR will pause the catalog routine, then when the pressed again continues the catalog listing.

ANY OTHER KEY WILL ABORT THE CATALOG.

Programs

This line has pathname in A\$		>100 A\$="DSK.ADISKNAME"
This line uses A\$ for the name		>110 CALL CAT(A\$)
of the device to catalog.		
This line will catalog drive 4		>100 CALL CAT(N)
if N=4		
This line will catalog drive C		>100 CALL CAT(X)
if X=67 (ASCII 67 is C)		
This line is path name.		>10 V\$="WDS1.VOLUME.SUB-DIR."
This line will catalog device		>20 CALL CAT(V\$)
WDS1 for directory VOLUME and		
catalog SUB-DIR		
This line catalogs drives 1		>100 CALL CAT(1,2,3,"WDS1.")
then 2 then 3 then WDS11		

Format CALL CHAR(character-code,pattern-identifier
 [,...])

 CALL CHAR(ALL,pattern-identifier[,...])

Description

See EXTENDED BASIC MANUAL page 56 for more data. Addition of the ALL command allows all the characters from 32 to 127 to be redefined all at once. Also CHAR now allows characters 30 (CURSOR) and 31 (EDGE CHARACTER) to be redefined or 144 to 156 may be redefined.

Programs

This line will define all the characters as a empty string.		>100 CALL CHAR(ALL,"")
FOR NEXT loop 30 to 127		>110 FOR X=30 to 127
This line prints a character.		>120 PRINT CHR\$(X);
NEXT to continue loop.		>130 NEXT X
Reset characters 32 to 127		>140 CALL CHARSETALL
This line repeats the program.		>150 GOTO 100
This line sets variable A\$ up.		>100 A\$="FF818181818181FF"
This line will define all the characters as a box.		>110 CALL CHAR(ALL,A\$)
This line defines the cursor.		>100 CALL CHAR(30,"FF81FF")
This line defines the edge character.		>110 CALL CHAR(31,"55")

Options

Sprites may not be used if characters 144 to 156 are being redefined for use.

Format CALL CHARSETALL

Description

The CHARSETALL command is just like the CHARSET command, but it resets characters from 32 to 127 thus resetting characters 95 to 127 unlike CHARSET.

Exactly like CHARSET it also resets colors to original mode.

Programs

This resets all characters and		>100 CALL CHARSETALL
colors to original.		
Set all characters the same.		>100 CALL CHAR(ALL,"4")
Set all colors the same.		>110 CALL COLOR(ALL,14,10)
Reset characters and colors		>120 CALL CHARSETALL
from character 32 to 127		
Go start over.		>130 GOTO 100

Format

CALL CLSALL

Description

The CLSALL command will find and close all open files.
This allows programmers to save time and program space.

Programs

The program to the right will		>100 CALL CLSALL
CLOSE all open files.		
This opens the printer.		>100 OPEN #9:"PIO",OUTPUT
This opens a disk file JUNK.		>110 OPEN #2:"DSK1.JUNK",INPUT
This closes both files.		>120 CALL CLSALL

Format

CALL COINC(#sprite-number,#sprite-number,
tolerance,numeric-variable[,...])

CALL COINC(#sprite-number,dot-row,dot-column,
tolerance,numeric-variable[,...])

CALL COINC(ALL,numeric-variable[,...])

Description

See EXTENDED BASIC MANUAL PAGE 64 for more data. The only difference is the use the comma has been added for auto-repeat. Previously a COINC only allowed one sprite comparison per program line.

Programs

* See EXTENDED BASIC MANUAL page 64

Clear screen set and X to 190	>100 CALL CLEAR :: X=190
Set up 3 sprites to be on	>110 CALL SPRITE(#1,65,2,9,X,
the same vertical plane.	20,0,#2,66,2,9,X,30,0,#3,67,
	2,9,X,-20,0)
COINC scans ALL sprites for a	>120 CALL COINC(ALL,A,#1,#2,1
collision then #1,#2,#3 also.	2,B,#1,#3,12,C,#2,#3,12,D)
Print results on screen.	>130 PRINT A;B;C;D
Loop forever to line 120	>140 GOTO 120

The above program in RXB will put a -1 in A,B,C,D variables unlike normal XB that would never detect all 4 collisions.

Options

While characters 144 to 159 are being used, you cannot use sprites. Notice the ALL option MUST ALWAYS BE FIRST as it was given highest priority to increase the detection rate. Though the ALL option does not improve much, the normal COINC detections are slightly faster as the interpreter is not looking to find the next COINC command on the next line number. Instead the comma and the next sprite is checked.

Format CALL COLOR(#sprite-number,foreground-color[,...])

 CALL COLOR(character-set,foreground-color,
 background-color[,...])

 CALL COLOR(ALL,foreground-color,background-color
 [,...])

Description

See EXTENDED BASIC MANUAL page 66 for more data. Presently modifications to the COLOR subprogram is ALL that will change all character sets from 0 to 14 to the same foreground and background colors and sets 0 to 16 may be individually set.

SET NUMBER	CHARACTER CODES	
0	30-31	
1	32-39	
2	40-47	
3	48-55	
4	56-63	
5	64-71	
6	72-79	
7	80-87	
8	88-95	
9	96-103	
10	104-111	
11	112-119	
12	120-127	
13	128-135	
14	136-143	
15	144-151	(RXB addition)
16	152-159	(RXB addition)

Programs

This line sets all character sets to foreground 2 and the background 11		>100 CALL COLOR(ALL,2,11)
Sets all to transparent, then all to black on transparent.		>100 CALL COLOR(ALL,1,2,ALL,2 1)
Go start over.		>110 GOTO 100

Options

While characters 144 to 159 are being used, you cannot use sprites.

Format COPY start line-end line,new start line,increment

Description

The COPY command is used to copy a program line or block of program lines to any other location in the program. The COPY does not affect the original lines and leaves them intact.

The block to be copied is defined by start line and end line. If either of these numbers are omitted, the defaults are the first program line and the last program line. However, at least one number and a dash must be entered (both can't be omitted), and there must be at least one valid program line between start line and end line. To copy one line enter it as both the start line and end line number. If any of the above conditions are not met, a Bad Line Number Error will result.

The new start line number defines the new line number of the first line in the block to be copied. This number must be entered. There is no default. The increment defines the line number spacing of the copied lines and may be omitted. The default is 10. There must be sufficient space in the program for the copied segment to fit between new start line number and the next program line following the location where the block will be moved. If not, a Bad Line Number Error message is reported. This problem can be corrected by using a smaller increment, or by using RES to open up space for the segment. A Bad Line Number Error also results if the copying process would result in a line number higher than 32767.

The COPY routine does not change any program references to the copied lines. It is an exact copy of the source lines with new line numbers. A check for sufficient memory space is made before each line is copied. If space is not available the copying process is halted and a Memory Full Error is reported.

Before the first line is copied, any open files are closed and all variables are lost.

Description Addendum PLEASE NOTE:

The COPY command copies the lines in reverse order. If the copying process is halted due to insufficient memory space, any uncopied lines will be at the beginning of the block.

Commands

```

Lines 100 to 150 are copied to| >COPY 100-150,9000,5
line 9000 and incremented by 5|
|
Line 10 is copied to line 25 | >COPY 10-10,25
|
Line 5 to last line are copied| >COPY 5-,99
to 99 and incremented by 10 |
(Default). |

```

Format

```
CALL CUTDIR(pathname,directory-name[,...])
```

```
CALL CUTDIR(string-varialbe,string-variable  
[,...])
```

Description

The CUTDIR subprogram removes directories and subdirectories on hard drives. The pathname determines the device used and the pathname can be up to 255 characters in length. The pathname must end with a period and the directory may only be 10 characters in length. Only a SCSI controller supports this command. CUTDIR will remove a directory and all its sub-directories at once. BE CAREFUL WITH THIS COMMAND!

Programs

```
This line removes a directory| >CALL CUTDIR("WDS1.", "TEST")  
named TEST on hard drive 1.  |
```

```
This line removes directory | >100 CALL CUTDIR("WDS1.", "ONE  
ONE and all sub-directories | ")  
that are within it.         |
```

```
This line would remove every | >100 CALL CUTDIR("WDS1.", "WOW  
thing off WDS1 if WOW had    | ")  
all main directories in it.  |
```

Options

This command requires a updated SCSI EPROM. The original SCSI EPROM did not include CUT DIRECTORY SUPPORT.

Format DEL start line-end line

Description

The DEL command is used to delete a line or block of lines from a program. Start line number and end line number define the lines to be deleted. If start line number is omitted, line deletion will begin at the first line of the program. In this case, end line number must be preceded by a dash. If end line number is omitted, line deletion will end at the last line of the program. If start line number and end line number are omitted, then the first line number of the program to the last line number of the program is deleted. At least one valid program line must exist between start line number and end line number or a Bad Line Number Error will be reported. If only one line number is given without a dash, then that one line will be deleted.

After the DEL command has executed any open files are closed and all variables are lost.

Commands

Lines 100 to 150 are deleted.		>DEL 100-150
Line 10 is deleted.		>DEL 10
Line 5 to last line are		>DEL 5-
deleted.		
First line to 80 are deleted.		>DEL -80

```

Format      CALL DIR("#"[,...])

              CALL DIR("DSK#."[,...])

              CALL DIR("DSK.DISKNAME."[,...])

              CALL DIR(string-variable[,...])

              CALL DIR(number[,...])

              CALL DIR(numeric-variable[,...])

              CALL DIR(ASC II value[,...])

```

Description

The DIR command catalogs the disk drive indicated by the # which can be 1 to z or by path name. The path name may be up to 30 characters long. A numeric variable or number can be used for drives 1 to 9 or if higher then it is assumed that the numeric-variable or number is a ASCII value between 30 to 255. This allows a catalog of a RAM-DISK designated by letters or control characters.

RXB DIR can be used from program mode or command mode. Also DIR can catalog up to 32 drives in one command.

The SPACE BAR will pause the catalog routine, then when the pressed again continues the catalog listing.

ANY OTHER KEY WILL ABORT THE CATALOG. See CAT for more info.

Programs

```

This line puts the pathname in| >100 A$="DSK.ADISKNAME"
the string A$                  |
This line uses A$ for the name| >110 CALL DIR(A$)
of the device to catalog.     |
                               |
This line will catalog drive 4| >100 CALL DIR(N)
if N=4                         |
                               |
This line will catalog drive C| >100 CALL DIR(X)
if X=67 (ASCII 67 is C)       |
                               |
This line is path name.       | >10 V$="WDS1.VOLUME.SUB-DIR."
This line will catalog device | >20 CALL DIR(V$)
WDS1 for directory VOLUME and|
catalog SUB-DIR               |
                               |
This line catalogs drives 1   | >100 CALL DIR(1,2,3,"WDS1.")
then 2 then 3 then WDS1      |

```

Format CALL DISTANCE(#sprite-number,#sprite-number,
 numeric-variable[,...])

 CALL DISTANCE(#sprite-number,dot-row,
 dot-column,numeric-variable[,...])

Description

The only thing added by RXB to DISTANCE is the auto repeat.
See EXTENDED BASIC MANUAL page 80 for more data.

Program

The program at the right will		>100 CALL CLEAR
set up 3 sprites on screen and		>110 CALL SPRITE(#1,65,7,99,9
start them moving.		9,0,10,#2,66,4,99,99,10,0,#3
		,67,2,1,2,-50,-50)
Scans three sprites locations		>120 CALL DISTANCE(#1,#2,D,#1
and returns the distance from		,#3,E,#2,#3,F)
each other squared.		>130 DISPLAY AT(1,1):"#1/#2";
		D:"#1/#3";E:"#2/#3";F)
Restart loop		>140 GOTO 120

Options

While characters 144 to 159 are being used, you cannot use sprites. The DISTANCE subprogram does get more accurate if you have more than one to check at a time, but is slightly faster than normal XB as DISTANCE in RXB does not require a search for another line number to CALL DISTANCE and find a value. The RXB version just goes to the comma and finds the next value of DISTANCE, so is much faster and saves program memory.

Format CALL DUPCHAR(character-code,character-code
 [,...])

Description

The DUPCHAR subprogram allows you to duplicate character definitions. You can duplicate character-codes from 30 to 159. The first character-code specifies the character-code to be duplicated, and the second character-code specifies the destination character-code. By duplicating a character definition using sprites would result in duplicate sprites.

Program

The program at the right will | >100 CALL DUPCHAR(65,66)
duplicate the character |
definition of character-code |
65 into character definition |
of character-code 66. |
The program at the right will | >100 FOR C=32 TO 158
blank out all character-code | >110 PRINT CHR\$(C);
definitions from 33 to 159. | >120 CALL DUPCHAR(C,C+1)
Line 110 is just to show what | >130 NEXT C
the character was before the |
DUPCHAR subprogram blanked it. |

Format

CALL DUPCOLOR(character-set,character-set
[,...])

CALL DUPCOLOR(#sprite-number,#sprite-number,
[,...])

Description

The DUPCOLOR subprogram duplicates foreground and background colors of the first set into the second set. Or the first sprite-number color into the second sprite-number color. The character-set numbers are given below:

set-number	character-codes
~~~~~	~~~~~
0 -----	30 to 31
1 -----	32 to 39
2 -----	40 to 47
3 -----	48 to 55
4 -----	56 to 63
5 -----	64 to 71
6 -----	72 to 79
7 -----	80 to 87
8 -----	88 to 95
9 -----	96 to 103
10 -----	104 to 111
11 -----	112 to 119
12 -----	120 to 127
13 -----	128 to 135
14 -----	136 to 143
(also sprite table) 15 -----	144 to 151
(also sprite table) 16 -----	152 to 159

### Programs

The program to the right will  
will duplicate set 6 colors in  
to set 9.

|>100 CALL DUPCOLOR(6,9)  
|  
|  
|>100 CALL CLEAR  
|>110 CALL SPRITE(#1,65,2,99,9  
| 9,#2,66,16,88,88)  
|>120 FOR X=1 TO 1E3 :: NEXT X  
|>130 CALL DUPCOLOR(#1,#2)  
|  
|>140 GOTO 140  
|

Clear screen  
Line 110 sets up two sprites  
on screen.  
Delay loop.  
Line 130 duplicates the color  
from sprite-number 1 into  
sprite-number 2.  
Loops forever.

```

Format      RUN "EA"

             DELETE "EA"

             CALL XBPGM("EA")

             CALL CAT("EA")

             OLD EA

             SAVE "EA"      -(Must have a program within
                             - memory to work at all)

             CALL EA
  
```

#### Description

The EA DSR (Device Service Routine) allows access to the Editor Assembler section of RXB. The access will work only if the DSR is the GPLDSR or LINK DSR. In other words, a DSR that acknowledges any type of DSR in RAM, ROM, GROM, GRAM, or VDP. Most DSR's only accept DSK or PIO. Others like the SAVE or LIST commands will only work with a program in the memory first. Still others like CALL LOAD("EA") must have the CALL INIT command used first.

Keep in mind that if it does not work, the problem is the DSR your using. Almost all DSR's today only acknowledge the ROM or RAM DSR's. As the EA DSR is in GROM/GRAM it seems a bit short sighted on the part of most programmers to use cut down versions of a DSR. Please discourage this practice as it is a disservice to us all.

#### Programs

```

Go to the Editor Assembler.  | >100 CALL XBPGM("EA")
                               |
This line asks for a string.  | >100 INPUT A$
This line uses the string and | >110 DELETE A$
if you type EA then enter will|
switch to the Editor Assembler|
                               |
This line will switch to the  | >CALL CAT("EA")
Editor Assembler.            |
                               |
This line shows lower case can| >call ea
be used.                     |
                               |
This line will have a strange | >CALL EAPGM("EA")
looping effect.              |
  
```

#### Options

BASIC and XB are also available. CALL EA

Format                   CALL EALR("access-name")

#### Description

The EALR subprogram is used to switch to the Editor Assembler Load and Run menu screen prompt. EALR will only load and run Editor Assembler DISPLAY FIXED OBJECT FILES created by the Editor Assembler for the Editor Assembler environment, not the EXTENDED BASIC DISPLAY FIXED OBJECT FILES. They've never been compatible, hence one of RXB's reasons for existing. The access-name is moved into the Editor Assembler and the name is loaded onto the screen so you can see it. This gives you a chance to change the disk if needed, or to see what is wrong if it does not load. After the DISPLAY FIXED OBJECT FILE is loaded, you will receive the normal 'Program Name?' prompt. This name would be the same as the link name from Editor Assembler BASIC. You can ABORT the loader by holding the FCTN BACK (9) key while the name is being placed onto the screen. If an error occurs the code will be returned onto screen and you must press ENTER to restart the loader.

#### Description Addendum

EALR only works from EXTENDED BASIC, not BASIC.

#### Programs

The program at the right will | >100 CALL EALR("DSK3.SAVE")  
load a Display/Fixed 80 file |  
named SAVE from disk drive 3. |

This program loads a Display/ | >100 CALL EALR("DSK.DNAME.FNA  
Fixed Object file named FNAME | ME")  
after searching all disk |  
drives and RAMDISKS for the |  
disk named DNAME. |

Format           CALL EAPGM("access-name")

#### Description

The EAPGM subprogram is used to switch to the Editor Assembler 'Run Program file?' screen prompt. It will not run EXTENDED XB programs or BASIC programs for that see XBPGM. The access-name is moved into the Editor Assembler and the name is loaded onto screen so you can see it. This gives you a chance to change the disk if needed, or to see what is wrong if it does not load. After the Program Image file is loaded, it executes the program normally. You can ABORT the loader by holding down the FCTN BACK (9) key while the name is being placed onto the screen. If an error occurs the error code will be returned onto the screen and you must press ENTER to restart the loader.

#### Description Addendum

EAPGM only works from EXTENDED BASIC, not BASIC.

#### Programs

The program at the right will load a Program Image file named FW from disk drive 2.	>100 CALL EAPGM("DSK2.FW")
This program loads a Program Image file named MG after searching all disk drives and and RAMDISKS for a disk named FW.	>100 CALL EAPGM("DSK.FW.MG")
This program causes a search for MAXPRO in directory ART on SCS1.	>100 T\$="SCS1.ART.MAXPRO"
	>110 CALL EAPGM(T\$)

Format                      CALL EXECUTE(cpu-address[,...])

                            CALL EXECUTE(numeric-variable[,...])

#### Description

The EXECUTE subprogram directly goes to the cpu-address and expects to find 4 bytes to be present. The bytes are 1 and 2 define the workspace register address. Bytes 3 and 4 define the address to start execution at in cpu memory. Programmers can see this is a BLWP at a cpu-address. The programmer is responsible for keeping track of the workspace and program space he is using. Also for any registers while doing a BL or another context switch. A RTWP will end either a BL or a BLWP as long as registers set are not changed. By using CALL LOAD or CALL MOVES the programmer can set up a BLWP routine in the lower 8K by filling the registers with values first, then using CALL EXECUTE to directly complete these commands. This is faster then CALL LINK as no interpretation of the access or values are checked.

EXECUTE runs a XML link from GPL by moving 12 bytes from the Fast RAM at HEX 8300 to VDP at HEX 03C0 then moving the value in FAC passed from XB to HEX 8304 and does a GPL XML >F0 After a RTWP by the Assembly program, it returns VDP HEX 03C0 to Fast RAM HEX 8300 so the 12 bytes are restored. Thus this allows programmers use of FAC and ARG areas in Fast RAM.

Here is the program loaded into Fast RAM by EXECUTE:

	AORG	>8300	
CPUPGM	DATA	>8302	First address.
	BLWP	@>834A	Switch context
			with FAC as dummy.
	CLR	@>837C	Clear for GPL return.
	RT		Return to GPL.
	END		

If a programmer absolutely must use Fast RAM for his program I suggest he set up a buffer for saving HEX 8300 to HEX 83FF if only so it will not mess up any GPL pointers and don't go and mess up the 12 bytes at VDP HEX >03C0. Then the only thing to worry about is messing up something else.

---

### Programs

```

Line 100 initializes lower 8k | >100 CALL INIT
Line 110 loads the assembly  | >110 CALL LOAD(9838,47,0,38,1
program shown below. VMBR   | 14,4,32,32,44,3,128)
Line 120 loads registers with | >120 CALL LOAD(12032,0,0,48,0
VDP address, Buffer, Length. | ,2,255)
Line 130 runs line 110 program | >130 CALL EXECUTE(9838)
Line 140 loads the assembly  | >140 CALL LOAD(9838,47,0,38,1
program shown below. VMBW   | 14,4,32,32,36,3,128)
Line 150 loads registers with | >150 CALL LOAD(12032,0,0,48,0
VDP address, Buffer, Length. | ,2,255)
Line 160 runs line 140 program | >160 CALL EXECUTE(9838)
Line 170 put a command in here | >170 CALL VCHAR(1,1,32,768)
Line 180 loops to line 160    | >180 GOTO 160

```

HEX ADDRESS	HEX VALUE	ASSEMBLY COMMAND EQUIVILENT
>266E	>2F00	DATA >2F00 (workspace area address)
>2670	>2672	DATA >2672 (start execution address)
>2672	>0420	BLWP (first executed command)
>2674	>202C	@VMBR (or >2024 VMBW)
>2676	>0380	RTWP

---

>2F00	>0000	REGISTER 0 (VDP address)
>2F02	>3000	REGISTER 1 (RAM buffer address)
>2F04	>02FF	REGISTER 2 (length of text)

### Normal XB using LINK.

```

Initialize for Assembly. | >100 CALL INIT
Load support routine.    | >110 CALL LOAD("DSK1.TEST")
LINK to program.         | >120 CALL LINK("GO")
                          |

```

### RXB using EXECUTE.

```

Initialize for Assembly. | >100 CALL INIT
Load support routine.    | >110 CALL LOAD("DSK1.TEST")
EXECUTE program address. | >120 CALL EXECUTE(13842)

```

The difference is EXECUTE does no checking so the address must be correct. The LINK method finds the name and uses the 2 byte address after the name to run the Assembly. EXECUTE just runs the address without looking for a name. Execute using a LOAD can be up to 5 times faster than LINK.

### Options.

Dependent on Programmers use and skill.





Format           CALL FILES(number)

                  CALL FILES(numeric-variable)

#### Description

The FILES subprogram differs from the Disk Controller FILES on the CorComp, TI, Myarc or Parcom versions. All of these require a NEW after CALL FILES. NEW is executed after the FILES subprogram in RXB, so there is no need to use NEW. Also RXB FILES accepts values from 1 to 15 unlike the other FILES routines that can only accept 1 to 9. Each open file reduces VDP by 534 bytes, plus each file opened will use 518 bytes more.

#### Programs

FILES opens usual buffers.	>CALL FILES(3)
FILES ends the program and executes NEW.	>100 CALL FILES(1)
Only possible in RXB	>100 CALL FILES(15)
	>NEW
	>SIZE

Will display 5624 Bytes of Stack free and 24488 Bytes of Program space free. At this point up to 15 files may be open at the same time. Not recommended but possible now.

#### Options

See XBPGM for even more powerful applications made easy. For example CALL XBPGM("DSK1.LOAD",2) will CALL FILES(2) then NEW then RUN "DSK1.LOAD"

Format                      CALL GCHAR(row,column,numeric-variable[,...])

#### Description

See EXTENDED BASIC MANUAL page 88 for more data. The only change to GCHAR is the auto-repeat function.

#### Programs

This line stores the character		>100 CALL GCHAR(4,5,A,4,6,B)
at row 4 column 5 in A, then		
stores character at row 4		
column 6 in B.		
Gets row 9 column 3 in Q and		>100 CALL GCHAR(9,3,Q,9,4,R)
row 9 column 4 in R.		
Put R at row 9 column 3 and		>110 CALL HCHAR(9,3,R,1,9,4,Q
Q at row 9 column 4		,1)
Continue loop.		>120 GOTO 100

#### Options

CALL GCHAR in RXB is much faster than normal XB now.

---

Format           CALL GMOTION(#sprite-number,row-velocity,  
                  column-velocity[,...])

### Description

The GMOTION subprogram returns the row-velocity and column-velocity as numbers from -128 to 127. If the sprite is not defined, row-velocity and column-velocity is set to zero. The sprite continues to move after its motion is returned, so this must be allowed for. See EXTENDED BASIC MANUAL MOTION subprogram for more data.

### Program

GMOTION returns the row-velocity into X and the column-velocity into Y.	>100 CALL GMOTION(#1,X,Y)
Set up screen and up,down ,left,right variables A(0) and A(1)	>100 A(0)=-1::A(1)=1::CALL CL
Loop for 28 sprites.	EAR::CALL MAGNIFY(2)::CALL S
Set up 28 random sprites with random colors and motion.	CREEN(15)
	>110 FOR S=1 TO 28
	>120 CALL SPRITE(#S,64+S,INT(
	RND*16)+1,20+S,50+S,INT(A(RN
	D*1))*INT(RND*127),INT(A(RND
	*1))*INT(RND*127))
Loop counter.	>130 NEXT S
Random sprite selector, get that sprites motion, put the values on screen.	>140 S=INT(RND*28)+1::CALL GM
	OTION(#S,X,Y)::CALL HPUT(24,
	3,"CALL GMOTION("#&STR\$(S)&"
	,"&STR\$(X)&","&STR\$(Y)&")")
Delay loop.	>150 FOR L=1 TO 1E3::NEXT L
Clear screen and Z+1.	>160 CALL CLEAR::Z=Z+1::IF
Loop till Z>8	Z<8 THEN 140

### Options

While characters 144 to 159 are being used, you cannot use sprites.

Format           CALL HCHAR(row,column,character-code)

                  CALL HCHAR(row,column,character-code,  
                  repetition[,...])

#### Description

See EXTENDED BASIC MANUAL page 92 for more data. The only change to HCHAR is the auto-repeat function. Notice the new auto-repeat must have the repetitions used or it gets row confused with repetitions.

#### Programs

This line puts character 38 at	>100 CALL HCHAR(1,1,38,99,9,1
row 1 column 1 99 times, then	,87)
puts character code 87 at	
row 9 column 1	
Fills screen with characters.	>100 CALL HCHAR(1,1,32,768,1,
	1,65,768,1,1,97,768,1,1,30,7
	68) :: GOTO 100

#### Options

CALL HCHAR in RXB is faster than normal XB as separate line numbers are needed to continue placing characters on screen.

```

Format      CALL HEX(string-variable,numeric-variable[,...
              ])

              CALL HEX(numeric-variable,string-variable[,...
              ])

```

#### Description

The HEX subprogram converts Decimal to Hexadecimal or from Hexadecimal to Decimal. If a number or numeric-variable is first, HEX will convert the Decimal floating point value (Rounded off) to a four character sting and puts the string into the string-variable. If a string or string-variable is first, HEX will convert the String into a Decimal integer and put it into the numeric-variable. A numeric-variable or number ranges from -32768 to 32767 or the Hexadecimal equivalent of >8000 to >7FFF. The > is not used in HEX.

When a string or string-variable is null (length of zero) the numeric-variable will contain 0. The opposite is if a number or numeric-variable is 0 then the string-variable will contain a length of four and a value of >0000. Any time a string-variable is second it will be cleared before being assigned a new string value. All strings in HEX must be right justified or are returned as right justified, thus each string will be padded with zeros.

HEX will only use the first four characters of a string to convert the value, it will ignore the rest of the string.

Errors will result if a string contains characters other than 0-9 and A-F or a-f. Errors will result if a number is less than -32768 or larger than 32767.

CALL HEX can be used for RXB CALL SECTOR but requires that two strings be combined for Hard Drive access. A CALL HEX of -1 equals >FFFF and -32768 equals >8000 thus 1 equals >0001 and 32767 equals >7FFF so:

```

FOR SECTOR=0 to 32767 would be 0 to >7FFF
FOR SECTOR=-32768 to -1 would be >8000 to >FFFF
Only good for a 65535 sector hard drive.

```

As SECTOR was intended for Hard Drive access using a normal DEC to HEX routine a preferred method over RXB CALL HEX simply for ease of use is required. (SORRY)

>FFFFFFFF would require a 4294967295 sector hard drive.

CALL HEX is designed for memory access not hard drives.  
(again sorry but this was intended)

---

 Programs

From command mode.		
Upper case		>CALL HEX("F",V)
or lower case		>CALL HEX("f",V)
will both return same result.		>PRINT V
V=15		
Line 100 sets address counter.		>100 FOR D=-32768 TO 32767
Line 110 converts it to HEX.		>110 CALL HEX(D,H\$)
Line 120 shows DEC to HEX.		>120 PRINT D,H\$
Line 130 continues loop count.		>130 NEXT D
Line 100 asks for HEX number.		>100 INPUT "HEX=":H\$
Line 110 converts HEX to DEC.		>110 CALL HEX(H\$,D)
Line 120 shows DEC equivalent.		>120 PRINT D: :
Line 130 starts program over.		>130 GOTO 100
Line 100 list of numbers.		>100 DATA 200,124,97,249,140,
It takes 8 bytes to store any		77,81,173,254,78,93,12,38,65
number in XB.		,55,6,0
Line 110 read list into N.		>110 READ N
Line 120 convert to HEX.		>120 CALL HEX(N,N\$)
Line 130 Save into a string as		>130 S\$=S\$&SEG\$(N\$,2,2)
it takes 4 bytes per number.		
Line 140 check for end of list		>140 IF N<>0 THEN 110
Line 150 show number of bytes		>150 PRINT "NORMAL:";8*16
used to store numbers.		
Line 160 show number of bytes		>160 PRINT "USED: ";LEN(S\$)+
it would have used.		1
Line 170 show number of bytes		>170 PRINT "SAVED ";(8*16)-(
it saved using string instead.		LEN(S\$)+1);"BYTES"

---

Format           CALL HGET(row,column,length,string-variable  
                 [,...])

#### Description

The HGET subprogram returns into a string-variable from the screen at row and column. Length determines how many characters to put into the string-variable. Row numbers from 1 to 24 and column numbers from 1 to 32. Length may number from 1 to 255. If HGET comes to the edge of the screen then it wraps to the other side.

#### Programs

The program to the right will		>100 CALL HGET(5,9,11,E\$)
put into string-variable E\$		
the 11 characters at row 5 and		
column 9.		
		>100 CALL HGET(1,3,5,M\$,9,3,1
The program to the right will		,Q\$,24,1,32,N\$)
put into string-variable M\$		
the 5 characters at row 1 and		
column 3, then put into		
string-variable Q\$ the 1		
character at row 9 and column		
3, then put into		
string-variable N\$ the 32		
characters at row 24 and		
column 1.		

## Format

CALL HONK

## Description

The HONK command produces the same sound as the ACCEPT or in INPUT or if a error occurs.

## Programs

The program to the right will		>100 PRINT "YN ?"
will produce a honk sound.		
Key request for YN.		>110 CALL KEY("YN",0,K,S)
Indicate N was pressed.		>120 IF K=78 THEN CALL HONK
Continue on with program.		>130 GOTO 1000



Format

CALL HPUT(row,column,string[,...])  
  
CALL HPUT(row,column,string-variable[,...])  
  
CALL HPUT(row,column,number[,...])  
  
CALL HPUT(row,column,numeric-variable[,...])

#### Description

The HPUT subprogram puts a string, string-variable, number, or numeric-variable onto the screen at row and column. The row numbers from 1 to 24 and column numbers for 1 to 32. If the string, string-variable, number, or numeric-variable being put onto screen goes to an edge it wraps to the other side. Unlike the EXTENDED BASIC DISPLAY AT the HPUT subprogram will not scroll the screen.

#### Programs

Line 100 puts string "THIS" on  
the screen at row 10 and  
column 4.

>100 CALL HPUT(10,4,"THIS")

Line 110 sets string-variable  
A\$ equal to string "HPUT"

>110 A\$="HPUT"

Line 120 puts string "is" at  
row 12 and column 5, then puts  
string-variable A\$ at row 14  
and column 4.

>120 CALL HPUT(12,5,"is",14,4  
          ,A\$)

Line 100 puts string A\$ at row  
16 and column 5.

>100 CALL HPUT(16,5,A\$)

Puts 456 at row 10 col 15

>100 CALL HPUT(10,15,456)

Format                      CALL INIT

#### Description

The INIT command is the same as the EXTENDED BASIC MANUAL page 101. Originally INIT loaded more data than actually existed, this has been fixed. The other correction is that you no longer have to use INIT before LINK, or LOAD. They will function if INIT has been called first or not. Unless loading a program that needs the INIT.

#### Programs

The program to the right will | >100 CALL INIT  
initialize the lower 8K by     |  
loading support routines for   |  
assembly.                       |  
                                 |

---

```

Format          CALL INVERSE(character-code[,...])

                CALL INVERSE(ALL[,...])

```

### Description

The INVERSE subprogram finds the character definition of the character-code and inverts all the bytes in the character definition. That means it just reverses the foreground and background. The ALL feature inverts characters 30 to 143 thus not affecting characters 144 to 159 as this would destroy sprites.

### Programs

```

The program to the right will | >100 CALL INVERSE(65)
INVERSE all character-code (A)|
in the character definition   |
table in memory.             |
                              |
The program to the right will | >100 CALL INVERSE(ALL)
INVERSE all character-codes   |
from 30 to 143.              |
                              |
Line 100 will ask for a string| >100 INPUT A$
of characters terminated by   |
the ENTER key.               |
Line 110 is a loop to counter.| >110 FOR L=1 TO LEN(A$)
Line 120 singles each one of | >120 C=ASC(SEG$(A$,L,1))
the characters in A$.         |
Line 130 INVERSEs each one.   | >130 CALL INVERSE(C)
Line 140 completes the loop.  | >140 NEXT L
Line 150 restarts the program.| >150 GOTO 100
(Be sure and not enter any blank characters in this program)

```

Format CALL IO(type,address[,...])

CALL IO(type,bits,cru-base,variable,variable  
[,...])

CALL IO(type,length,vdp-address[,...])

#### Description

The IO subprogram allows access to and control of any chip in the console or peripheral cards. The type refers to different access methods like playing sound from GROM or VDP memory automatically. The type can also specify reading or writing directly to a Control Register Unit (CRU) address. Thereby allowing direct chip control, or direct chip bypass if the user wishes. The IO subprogram is a Graphics Programming Language (GPL) command. So the function is exactly like GPL despite being run from the XB environment. As most of XB is written in GPL the user gains greater GPL like control. After all the Operating System is written in GPL for a good reason.*Note these docs are from GPL Manuals.

type	address specifications
~~~~~	~~~~~
0 -----	GROM sound list address.
1 -----	VDP sound list address.
2 -----	CRU input.
3 -----	CRU output.
4 -----	VDP address of Cassette write list.
5 -----	VDP address of Cassette read list.
6 -----	VDP address of Cassette verify list.

The length specifies the number of bytes. The length can be from -32768 to 32767 depending on the amount of VDP memory that is available. Of course a value of -32768 is HEX >8000 and 32767 is HEX >7FFF and VDP normally in a TI is only 16384 or HEX >4000 of VDP. So be careful or lock-up will result. The cru-base is the CRU address divided by 2 in decimal form as the command automatically doubles the value input. The CRU -base ranges from 0 to 8191 or HEX >0000 to >1FFF with a EVEN address for 8 bits or more being scanned. That means that a value of 8191 will lock-up the system as it is looking for a bit in 8192 that does not exist.

The variable can input or output values ranging from 0 to 255 as that is equivalent to a single byte value. As there are two variables 16 bits can be represented in the two 8 bit variables. If CRU input reads less than 8 bits, the unused bits in the byte are reset to zero. If CRU input reads less than 16 but more than 8 bits, the unused bits in the word will be reset to zero. The bits range from 1 to 16 for input or output.

AUTO-SOUND INSTRUCTION GROM/GRAM/VDP

Format CALL IO(type,address[,...])

Control of the Sound Generator Chip (SGC) in the system console is through a pre-defined table in GROM/GRAM or VDP memory. Sound output is controlled by the table and the VDP Interrupt Service Routine (ISR). A control byte at the end of the table can cause control to loop back up in the table to continue, or end sound output. The format of the table is the same regardless of where it resides. The table consists of a series of blocks, each of which contains a series of bytes which are directly output to the SGC.

Since the VDP generates 60 interrupts per second, the interrupt count is expressed in units of one-sixtieth of a second.

When the IO command is called, upon the next occurring VDP interrupt, the first block of bytes is output to the SGC. The interpreter (Operating System) waits the requested number of interrupts (for example, if interrupt counts are 1, every interrupt causes the next block to be output). Remember that interpretation of XB continues normally while the SGC control is enabled.

The sound control can be terminated by using an interrupt count of 0 in the last block of the table. Alternatively, a primitive looping control is provided by using a block whose first byte is 0, and the next 2 bytes indicate an address in the same memory space of the next sound block to use. (That means one block points to another block only in the same type of memory).

If the first byte is hex FF or decimal 255, the next two bytes indicate an address in the other memory space. (That means one block points to another block but in another type of memory.) These allow switching sound lists from GROM/GRAM to VDP or VDP to GRAM/GROM. By making this the beginning of the entire table, the sound sequence can be made to repeat indefinitely.

The type 0 indicates sound lists in GROM or GRAM and type 1 indicates sound lists in VDP.

Executing a sound list while table-driven sound control is already in progress (from a previous sound list) causes the old sound control to be totally supplanted by the new sound instruction. (That means any sound chip command will override old sound chip commands).

The SGC has 3 tone (square wave) generators - 0, 1, and 2 all of which can be working simultaneously or in combination. The frequency (pitch) and attenuation (volume) of each generator can be independently controlled. In addition, there is a noise generator which can output white or periodic noise. For more information on controlling the SGC, see the TSM9919 SGC specification.

ATTENUATION CONTROL (for generators 0, 1, 2 or 3)

One byte must be transmitted to the SGC:

Binary 1-REG#-1-Attenuation

REG# = register number (0,1,2,3)

Attenuation = Attenuation/2

(e.g. A=0000 0 db = highest volume;

A=1000 16 db = medium volume;

A=1111 30 db = off.)

EXAMPLE: 1 10 1 0000 : turn on gen. #2 highest volume.

1 01 1 0100 : turn on gen. #1 medium high volume.

1 11 1 1111 | turn off gen. #3 (noise generator).

FREQUENCY CONTROL (for generators 0, 1, 2)

Two bytes must be transmitted to the SGC for a given register and to compute the number of counts from the frequency F
use: $N = 111860 / F$

Binary 1-REG#-N(1s 4 bits)-00-N(ms 6 bits)

Note that N must be split up into its least significant 4 bits and most significant 6 bits (10 bits total).

The lowest frequency possible is 110 Hz and the highest is 55938 Hz.

NOISE CONTROL

One byte must be transmitted to the SGC:

Binary 1-1-1-0-0-T-S

T = 0 for white noise, 1 for periodic noise;

S = Shift rate (0,1,2,3) = frequency center of noise.

S=3=frequency dependent on the frequency of tone generator #3.

 Programs

```

Line 100 clears screen.      | >100 CALL CLEAR ! Chimes
Line 110 to ...              | >110 DATA 5,159,191,223,255,2
                              | 27,1,9,142,1,164,2,197,1,144
                              | ,182,211,6,3,145,183,212,5,3
                              | ,146,184,213,4
                              | >120 DATA 5,167,4,147,176,214
                              | ,5,3,148,177,215,6,3,149,178
                              | ,216,7
                              | >130 DATA 5,202,2,150,179,208
                              | ,6,3,151,180,209,5,3,152,181
                              | ,210,4
                              | >140 DATA 5,133,3,144,182,211
                              | ,5,3,145,183,212,6,3,146,184
                              | ,213,7
                              | >150 DATA 5,164,2,147,176,214
                              | ,6,3,148,177,215,5,3,149,178
                              | ,216,4
Line 160 ends sound list.    | >160 DATA 5,197,1,150,179,208
                              | ,5,3,151,180,209,6,3,152,181
                              | ,210,7,3,159,191,223,0
Line 170 reads list into B and| >170 A=A+1 :: READ B :: CALL
A is counter                  | POKEV(A,B)
Line 180 checks end of list? | >180 IF B=0 THEN 190 ELSE 170
Line 190 shows how to access. | >190 PRINT "TYPE:" : "CALL IO(
                              | 1,8192)"
                              | >200 CALL IO(1,8192)
                              |
Line 310 continues AD loop.   | >310 NEXT AD
Line 320 executes sound list. | >320 CALL IO(1,4096)
Line 330 prints out suggestion| >330 PRINT "CRASH": "TYPE":
on how to test IO.           | "CALL IO(1,4096)"

```

Programs

```
Line 100 clears the screen. | >100 CALL CLEAR ! CRASH
Line 110 to ...             | >110 DATA 2,228,242,5
                             | >120 DATA 2,228,240,18
                             | >130 DATA 2,228,241,16
                             | >140 DATA 2,228,242,14
                             | >150 DATA 2,228,243,12
                             | >160 DATA 2,228,244,10
                             | >170 DATA 2,229,245,9
                             | >180 DATA 2,229,246,8
                             | >190 DATA 2,229,247,7
                             | >200 DATA 2,229,248,6
                             | >210 DATA 2,229,249,5
                             | >220 DATA 2,230,250,4
                             | >230 DATA 2,230,251,3
                             | >240 DATA 2,230,252,2
                             | >250 DATA 2,230,253,1
                             | >260 DATA 2,230,254,1
Line 270 ends sound list.   | >270 DATA 1,255,0,0
Line 280 AD is VDP address to | >280 FOR AD=4096 TO 4160 STE
start with and ends with.   | P 4
Line 290 reads list.        | >290 READ V1,V2,V3,V4
Line 300 moves them into VDP. | >300 CALL POKEV(AD,V1,V2,V3,V
                             | 4)
Line 310 continues AD loop.  | >310 NEXT AD
Line 320 executes sound list. | >320 CALL IO(1,4096)
Line 330 prints out suggestion | >330 PRINT "CRASH": : "TYPE:":
on how to test IO.          | "CALL IO(1,4096)"
```

All data values must converted to Binary in order to see what is going on. You now have all the data that I have as to this phase of IO types 0 and 1. See Editor Assembler Manual also for more data on sound lists and sound chip.

Sound table creator for conversion of sound data.

```
100 CALL CLEAR
110 PRINT "*SOUND DATA TABLE CREATOR*"
120 Q$="0123456789ABCDEF"
130 INPUT "GENERATOR # ?":GN
140 INPUT "DURATION ?":DUR
150 INPUT "FREQUENCY ?":FREQ
160 INPUT "VOLUME ?":VOL :: PRINT : : :
170 IF DUR>17 THEN 190
180 DUR=17
190 REM DURATION
200 DUR=INT((DUR*255)/4250) :: CONV=DUR
210 GOSUB 430
220 DUR$=SEG$(HX$,3,2) :: IF FREQ>-1 THEN 310
230 REM NOISE FREQUENCY
240 FR=ABS(FREQ)-1 :: FR$="E"&STR$(FR)
250 REM NOISE VOLUME
260 VOL=INT(VOL/2) :: CONV=VOL
270 GOSUB 430 :: VOL$="F"&SEG$(HX$,4,1)
280 PRINT "DATA>02";FR$;";,>";VOL$;DUR$: : :
290 GOTO 390
300 REM TONE FREQUENCY
310 FR=INT((111860.8/FREQ)+.5)
320 CONV=FR :: GOSUB 430
330 FR$=SEG$(Q$,GN*2+7,1)&SEG$(HX$,4,1)&SEG$(HX$,2,2)
340 REM TONE VOLUME
350 VOL=INT(VOL/2) :: CONV=VOL
360 GOSUB 430
370 VOL$=SEG$(Q$,GN*2+8,1)&SEG$(HX$,4,1)
380 PRINT "DATA>03";SEG$(FR$,1,1)&SEG$(FR$,2,1);";,>";
SEG$(FR$,3,2);VOL$;";,>";DUR$;"00": : :
390 PRINT: : "ANOTHER SOUND (Y/N)?"
400 CALL ONKEY("YN",3,K,S) GOTO 100,420
410 GOTO 400
420 CALL CLEAR :: END
430 REM DECIMAL TO HEX
440 AY=INT(CONV)/16 :: BY=INT(AY)/16
450 CY=INT(BY)/16 :: DY=INT(CY)/16
460 AP=(AY-INT(AY))*16 :: BP=(BY-INT(BY))*16
470 CP=(CY-INT(CY))*16 :: DP=(DY-INT(DY))*16
480 HX$=SEG$(Q$,DP+1,1)&SEG$(Q$,CP+1,1)&
SEG$(Q$,BP+1,1)&SEG$(Q$,AP+1,1)
490 RETURN
```

Use this program to create Hex strings that can use
CALL MOVES to move strings into VDP to be played from
a CALL IO(1,VDP-address)

 CRU ACCESS INSTRUCTION

Format CALL IO(type,bits,cru-base,variable,variable
 [,...])

The IO types 2 and 3 can be used to control a variety of input-output devices including Speech, or CRU. IO always must be the CRU address divided by 2 as any value above 8192 will be out of range. The cru-base must be divided by 2 as the 9901 chip ignores the least significant bits of the base register it uses. See Editor Assembler Manual page 61.

The CRU data to be written should be right justified in the byte or word. The least significant bit will output to or input from the CRU address specified by the CRU base address. Subsequent bits will come from or go to sequentially higher CRU addresses. If the CRU input reads less than 8 bits, the unused bits in the byte are reset to zero. If the CRU input reads less than 16 bits but more than 8 bits, the unused bits in the full two 8 bit bytes will be reset to zero.

Programs

Line 100 display what it does for you.	>100 DISPLAY AT(1,1)ERASE ALL : "THIS PROGRAM CHECKS FOR UNUSUAL KEYS BEING PRESSED , EVEN IF MORE THEN FOUR KEY ARE BEING PRESSED AT ONCE"
Line 110 scans CRU at >0006 and reports keys pressed.	>110 CALL IO(2,16,3,A,B):: IF A=18 AND B=255 THEN 110 ELS E CALL HPUT(24,3,RPT\$(" ",30),24,24,STR\$(A)&" "&STR\$(B))
Line 120 more reports.	>120 IF A=146 THEN CALL HPUT(24,3,"FUNCTION KEY")ELSE IF B=191 THEN CALL HPUT(24,3,"C ONTROL KEY")ELSE IF B=223 TH EN CALL HPUT(24,3,"SHIFT KEY ")
Line 130 still more reports.	>130 IF B=251 THEN CALL HPUT(24,3,"ENTER KEY")ELSE IF B=2 53 THEN CALL HPUT(24,3,"SPAC E BAR")ELSE IF B=254 THEN CA LL HPUT(24,3,"PLUS/EQUAL KEY ")
Line start over scan of keys.	>140 GOTO 110

The above program scans CRU input/output lines for keys. Please note that EMULATION of the TI99/4A is never complete, so many CRU address in a Emulator are not fully functional.

Programs

Line 100 clears screen.	>100 CALL CLEAR
Line 110 explains program.	>110 CALL HPUT(4,7,"This is a demo of the",6,7,"CALL IO(3 ,8,2176,B)",8,7,"3 = TYPE(CR U output)",10,7,"8 = NUMBER OF BITS",12,7,"2176=address/ 2")
Line 120 turn off card, show the present byte value being sent.	>120 CALL IO(3,8,2176,0):: FO R B=0 TO 255 :: CALL HPUT(14 ,7,"B=byte (value "&STR\$(B)& ")")
Line 130 display block to get attention.	>130 CALL HPUT(18,5,"***** *****",19,5,"WA TCH THE DRIVE LIGHTS",20,5," *****")
Line 140 send byte to card and when done with loop, clear for starting over program.	>140 CALL IO(3,8,2176,B):: NE XT B :: CALL HCHAR(14,24,32, 7):: GOTO 110

The above program will only work on a REAL TI not EMULATORS.

Line 100 explains program.	>100 ! TURNS OFF/ON/OFF EACH CARD FROM >1000 TO >1F00 BUT WILL LOCKUP WITH CERTAIN CARDS.
Line 110 cru address from >1000 to >1F00	>110 FOR CRU=2048 TO 3968 STE P 128
Line 120 turn off card, turn on card, delay for 2 seconds, turn off card.	>120 CALL IO(3,8,CRU,0,3,8,CR U,255)::FOR A=1 TO 200::NEXT A::CALL IO(3,8,CRU,0)
Line 140 loop till done.	>140 NEXT CRU

Options

Some CRU address are used by the Operating System or XB and any attempt to redefine them will create problems. Also some of the address areas will return incorrect values as they have changed since IO has accessed them. These problems will never become completely apparent at first, so take care. Additionally some cards have the same problem, if the card has a program that has a interrupt or CRU links turned on as you access it, a complete lock up will result as a fight for control ensues. So with that happy thought, a alternate way is to use EXECUTE or LINK instead.

CASSETTE INPUT/OUTPUT/VERIFY INSTUCTION

Format CALL IO(type,length,vdp-address[,...])

The three different cassette I/O instuctions use the same format. The write and read instructions physically perform Input/Output to the cassette. The verify instruction will read a tape and compare it, byte by byte, against what is in the specified vdp area. All will report an I/O error if one is detected.

No prompts are present with these three formats. These three types control the cassette directly so no prompt will tell the user to turn on or off the cassette record/play buttons. The programmer must inform the user with his own prompt.

Programs

Presently I have no cassette to write programs with.

AUDIO GATE

CRU bit 24 is the audio gate which allows data being read to be heard. If the bit is set to 1, the data being read is heard, and if the bit is set to 0, the data is not heard. Setting the bit to a 0 or 1 is done with an IO instruction, or a Assembly instruction.

MOTOR CONTROL

There are two CRU bits (22 and 23) used to control cassettes 1 and 2, respectively. When there is no Cassette IO being done, both motors remain on. When Cassette IO is specified, the DSR (Device Service Routine) will control the data being read. If there are two motor units connected, the data will be read simultaneously, or you may have the option of reading data from one motor unit and playing the recorded voice from another motor unit through the TV (Monitor) speaker.

*NOTE:

Compatibility with or without 32K or other devices is not a concern as IO needs no RAM to work with. Therefore from just a console all IO commands will work fine. If you only have a Cassette and RXB you can quickly load/save/verify without menus, or just make up your own.

Format CALL ISROFF(numeric-variable)

Description

The Interrupt Service Routine (ISR) is a routine that executes during timed intervals. The operating system of the TI is set up for these. Mouse or Screen dumps or Hot Key programs bring to mind the common uses of a ISR hook. The ISROFF routine in RXB does as it suggests and turns the ISR hook off. But the numeric-variable is used to store the address of where this ISR hook came from. Of course ISRON is the opposite and will turn it back on. Extreme care must be used when turning on or off the ISR. A PEEK at hex >83C4 (decimal -31804 and -31805) will be 0 when there is no ISR. Otherwise any other value will mean that a ISR is being used.

Programs

This line checks ISR hook.		>100 CALL ISROFF(J)
This shows if ISR is in use.		>110 IF J THEN PRINT "ISROFF"
This line loads another file.		>120 CALL LOAD("DSK1.HOT")
This starts another ISR.		>130 CALL LINK("START")
This line checks ISR hook.		>140 CALL ISROFF(K)
This shows if ISR is in use.		>150 IF K THEN PRINT "ISROFF"
This turns first ISR back on.		>160 CALL ISRON(J)
This turns second ISR back on.		>170 CALL ISRON(K)
The program continues...		
Safer way to check ISRHOOK		>100 CALL PEEK(-31804,I,J)
Check if zero then no ISR ON		>110 IF I+J THEN CALL ISROFF
if I+J<>0 then turn off ISR		(N)
and put into variable N		

The above program has loaded N with the ISR HOOK Address.

Format CALL ISRON(numeric-variable)

Description

The Interrupt Service Routine (ISR) is a routine that executes during timed intervals. The operating system of the TI is set up for these. Mouse or Screen dumps or Hot Key programs bring to mind the common uses of a ISR hook. The ISRON routine in RXB does as it suggests and turns the ISR hook on. But the numeric-variable is used to load the address of where this ISR hook came from. Of course ISROFF is the opposite and will turn it back off. Extreme care must be used when turning on or off the ISR. A PEEK at hex >83C4 (decimal -31804 and -31805) will be 0 when there is no ISR. Otherwise any other value will mean that a ISR is being used.

Programs

This line peeks ISR hook.	>100 CALL PEEK(-31804,I,J)
This checks if ISR is in use,	>110 IF I+J THEN CALL ISROFF(
and if not 0 turn off ISR.	ADDRESS1)
This line loads another file.	>120 CALL LOAD("DSK1.HOT")
This starts another ISR.	>130 CALL LINK("START")
This turns off ISR.	>140 CALL ISROFF(ADDRESS2)
This checks if old ISR is ok,	>150 IF I+J THEN CALL ISRON(A
if yes turn it on.	DDRESS1)
The program continues...	
Safer way to check ISRHOOK	>100 CALL PEEK(-31804,I,J)
Check if zero then no ISR ON	>110 IF I+J THEN CALL ISRON(N)
if I+J<>0 then turn off ISR	
and put into variable N	

The above program has ISR HOOK Address loaded from N.

Format CALL JOYST(key-unit,x-return,y-return[,...])

Description

See EXTENDED BASIC MANUAL page 108

Except for adding auto repeat there is no changes to JOYST

Programs

The program on the right will		>100 CALL CLEAR
illustrate a use of JOYST		>110 CALL SPRITE(#1,33,5,96,1
subprogram. It creates two		28,#2,42,2,96,128)
sprites and then moves them		>120 CALL JOYST(1,X1,Y1,2,X2,
around according to the input		Y2)
from the joysticks.		>130 CALL MOTION(#1,-Y1,X1,#2
Two players with the same		-Y2,X2)
input speed and motion.		>140 GOTO 120

Format

```
CALL KEY(key-unit,return-variable,
status-variable[,...])

CALL KEY(string,key-unit,return-variable,
status-variable[,...])

CALL KEY(string-variable,key-unit,return-
variable,status-variable[,...])
```

Description

See EXTENDED BASIC MANUAL page 109

RXB has added auto repeat features.

Strings or string variables can now be added to KEY to lock out any other keys. The strings can be empty or up to 255 in length. The string function halts program execution unlike a normal key routine similar to ACCEPT or INPUT do.

Programs

This line scans both joysticks	>100 CALL JOYST(1,X,Y,2,XX,YY)
This line scans both of the	>110 CALL KEY(1,F,S,2,FF,SS)
fire buttons & split keyboard.	
Try this for fun.	>CALL KEY(CHR\$(2),0,K,S)
(HINT: FCTN 4)	
Add this line to programs.	>100 CALL KEY("YNyn",0,K,S)
Suspends program until key is	>100 CALL KEY("",0,K,S)
pressed. (any key)	
Suspends program until ENTER	>100 CALL KEY(CHR\$(13),0,K,S)
is pressed.	
Suspends program until the	>100 A\$="123"
key from string A\$ is used.	>110 CALL KEY(A\$,0,KV,STATUS)
Suspends program until YES is	>100 CALL KEY("Y",0,K1,S1,"E"
typed in.	,0,K2,S2,"S",0,K3,S3)

Format	LIST
	LIST "device name"
	LIST "device name":line length:line number- line number

Description

The LIST command is the same as per Extended Basic Manual page 114. The LIST routine has been modified to allow the line length to be output to a device. The line length can only be used if a device is specified. A colon (:) must follow the line length. If not included in the LIST command, the line length is set to the default of the specified output device. The line length can range from 1 to 255. If the length specified is outside this range, a Bad Line Number Error is reported.

Command

This line outputs to a device.	>LIST "PIO":80:100-120
This a dummy line.	>100 ! TEST OF LIST
Another dummy line.	>110 ! TEST OF LIST

```
Format      CALL LOAD("access-name"[,address,byte][,...]
            [,file-field,...])
```

```
CALL LOAD(address,byte[,...])
```

Description

See EXTENDED BASIC MANUAL page 115 for more data. The only change is to allow a CALL LOAD to an address without having to use CALL INIT first.

Program

This line will load address 8192 with 128	>100 CALL LOAD(8192,128)
Loads a 56 at address 8192 then skips loading then loads 78 and 91 at 8196, and 8197	>100 CALL LOAD(8192,56,"",8196,78,91)

Format

CALL MKDIR(pathname,directory-name[,...])

CALL MKDIR(string-varialbe,string-variable
[,...])

CALL MKDIR(number,disk-volume-name[,...])

Description

The MKDIR subprogram MaKes DIRectorys on hard drives or will name a disk. The pathname determines the device used and the pathname can be up to 255 characters in length. The Myarc HFDC can only support 29 characters pathnames plus the filename of 10, so that would add up to 39 characters total. The pathname must end with a period and the filename may only be 10 characters in length. MKDIR can create up to 24 directories in 24 different drives in one command. MKDIR can also create directories then sub-directories in the same command.

Programs

This line names disk 1 NONE

| >CALL MKDIR("DSK1.", "NONE")

|

This line creates a directory

| >CALL MKDIR("WDS1.", "TEST")

named TEST on hard drive 1.

|

This line creates a directory

| >100 CALL MKDIR("WDS1.", "ONE"

on hard drive 1 named ONE

| , "WDS1.ONE.", "TWO", "WDS1.ONE"

then creates a sub-directory

| .TWO.", "THREE")

named TWO of directory ONE

|

then creates a sub-directory

|

named THREE of directory ONE

|

of sub-directory TWO

|

The above program creates a directory then sub-directory then

a sub-directory of that sub-directory.

```

Format      CALL MOTION(#sprite-number,row-velocity,
                  column-velocity[,...])

            CALL MOTION(ALL,row-velocity,column-velocity
                  [,...])

            CALL MOTION(STOP[,...])

            CALL MOTION(GO[,...])

```

Description

See EXTENDED BASIC MANUAL PAGE 125 for more data. A added feature to MOTION is STOP (disable sprite movement) and GO (enable sprite movement). Also ALL that affects all sprites.

Programs

* See EXTENDED BASIC MANUAL.

The program to the right will		>100 CALL CLEAR :: X=190
will set up 3 sprites to be on		>110 CALL SPRITE(#1,65,2,9,X,
the same vertical plane, and		20,0,#2,66,2,9,X,30,0,#3,67,
MOTION will stop all sprites.		2,9,X,-20,0)
GO turns on sprite motion.		>120 CALL MOTION(GO)
This is a delay loop.		>140 FOR D=1 TO 2000::NEXT D
STOP turns off sprite motion.		>150 CALL MOTION(STOP)
This is a delay loop.		>160 FOR D=1 TO 2000::NEXT D
Change #3 motion direction, GO.		>170 CALL MOTION(#3,10,10,GO)
This is a delay loop		>180 FOR D=1 TO 2000::NEXT D
Continue program.		>190 GOTO 120
Clear screen and set up the		>100 CALL CLEAR::A(0)=-127 ::
variables A(0) and A(1)		A(1)=127
Loop to create sprites.		>110 FOR L=1 TO 28::CALL SPRI
		TE(#L,L+65,2,L,L,-L,L) ::
		NEXT L
Use MOTION ALL to change the		>120 CALL MOTION(ALL,A(RND)*R
sprite velocities.		ND,A(RND)*RND)::GOTO 120

Options

While characters 144 to 159 are being used, you cannot use sprites. Notice that GO or STOP can be followed by other motion commands i.e. CALL MOTION(STOP,#1,44,-87) is valid.

Format MOVE start line-end line,new start line,increment

Description

The MOVE command is used to move a program line or block of program lines to another location in the program. The block of lines to be moved is defined by start line number and end line number. If either of these numbers are omitted, the defaults are the first program line and the last program line. However, at least one number and a dash must be entered (both cannot be omitted), and there must be at least one valid program line between start line number and end line number. To move one both the start line number and end line number are the same. If any of the above conditions are not met, a Bad Line Number Error will be reported.

The new start line number defines the new line number of the first line in the moved segment. When the block is moved it will be moved. If not, a Bad Line Number Error message is reported.

This problem can be corrected by using a smaller increment, or by using RES to open up space for the segment. A Bad Line Number Error also results if the renumbering process would result in a line number higher than 32767.

Although moving lines within the program does not increase the size of the program, this command does require 4 bytes of the program space for line moved. This memory use is temporary, but it must be available in order to move the block. If sufficient memory is not available a Memory Full Error results and no lines are moved. This problem can usually be worked around by moving the block a few lines at a time.

Before the block of lines is moved any open files are closed and any variables are lost.

Commands

This line moves lines 100 thru 180 to line 1000, and then increment by 5.	>MOVE 100-180,1000,5
---	----------------------

This line moves lines 40 thru last line to line 120, and increment by 10. (Default)	>MOVE 40-,120,
---	----------------

This line moves line 150 to line 1110	>MOVE 150-150,1110
---------------------------------------	--------------------

This line moves first program line thru line 800 to line 32220, and increment by 2.	>MOVE -800,32220,2
---	--------------------

Format CALL MOVES (type\$,bytes,string-variable,string-variable[,...])

 CALL MOVES (type\$,bytes,from-address,to-address[,...])

 CALL MOVES (type\$,bytes,from-address,string-variable[,...])

 CALL MOVES (type\$,bytes,string-variable,to-address[,...])

 CALL MOVES (string-variable,number,string-variable,string-variable[,...])

Description

The MOVES subprogram moves (copies) from-string to-string the amount of bytes specified using the memory type string. MOVES does not physically move memory but copies it. MOVES can RIPPLE a byte thru memory by the from-address being one byte less than the to address.

The type\$ below specifies what type of memory is being moved and to what other type of memory it is moved into. The bytes are 255 maximum if being moved into a string-variable.

MOVES address range is from -32768 to 0 to 32767.

As MOVES mostly works with string-variables please see the the Extended Basic Manual page 41. MOVES will error out with *BAD VALUE IN ###* with-in a program if the string variable length exceeds 255, or if the number of bytes exceeds 255.

type\$		TYPE OF MEMORY
~~~~~		~~~~~
\$	-----	STRING-VARIABLE
V	-----	VDP ADDRESS
R	-----	RAM ADDRESS
G	-----	GRAM ADDRESS

*NOTE: upper case only for type as lower case are ignored.

RAM may be moved but not into RAM but not ROM, and that you may move memory into GRAM but not GROM. You can copy or move memory from ROM or GROM. Also note that any devices that use phony GRAM will not work with MOVES as these devices don't use the real GRAM/GROM addressing.

VDP address are from 0 to 16384 (>0 to >3FFF)

-----  
Programs

```

Line 100 has the type$ string. | >100 X$="VV"
Line 110 thus uses type$ 0 VDP | >110 CALL MOVES(X$,767,1,0)
to VDP. 767 bytes are moved. A |
VDP from-address of 1 and a |
VDP to-address of 0. Will use |
a ripple effect of moving all |
screen bytes over one address. |
|
Line 100 copies entire screen | >100 CALL MOVES("VR",768,0,81
into lower 8K. | 92)
|
Line 110 clears the screen. | >110 CALL CLEAR
Line 120 copies entire screen | >120 CALL MOVES("VR",768,0,90
into lower 8K. | 00)
Line 130 copies from lower 8K | >130 CALL MOVES("RV",768,8192
to screen, then again. GOTO | ,0,"RV",768,9000,0) :: GOTO
makes it an endless loop. | 130
|
Line 100 sets up loop. Counts | >100 FOR G=-32768 TO 32767
from -32768 to 0 to 32767 or |
(HEX >8000 to >0000 to >7FFF) |
Line 110 move GRAM/GROM to | >110 CALL MOVES("GV",8,G,1024)
VDP. 8 bytes to be moved. GA |
is counter. 1024 is decimal |
address of space character in |
VDP pattern table. |
Line 120 completes loop. | >120 NEXT G
|
|
Loop address VDP | >100 FOR V=0 TO 16384
Load that 8 bytes into space | >110 CALL MOVES("VV",8,V,1024)
Loop back | >120 NEXT V
|
|
Loop address RAM | >100 FOR R=_32768 to 32767
Load that 8 bytes into space | >110 CALL MOVES("RV",8,R,1024)
Loop back | >120 NEXT R
|

```

---

### Programs

Line 100 sets string-variable.		>100 I\$=RPT\$("I",255)
Line 120 type\$ specifies I\$		>110 CALL MOVES("\$V",55,I\$,0)
to VDP. 55 bytes are moved.		
Line 120 copies string J\$ to		>120 CALL MOVES("\$R",255,J\$,8
into lower 8K, then string I\$		192,"\$R",255,I\$,8492)
into lower 8K.		
Line 130 copies string I\$ to		>130 J\$=I\$ :: PRINT J\$ : : I\$
into J\$. Eliminates old J\$.		
Then prints them.		
Line 150 copies from lower 8K		>140 CALL MOVES("R",255,8192
to J\$, then from lower 8K at		,J\$,"R",255,8492,I\$) :: PRI
8492 into I\$ thus restoring		NT J\$: :I\$
both strings and printing them		
thus a way to save stings.		
Line 100 sets up loop. Counts		>100 FOR GA=-32768 TO 32767
from -32768 to 0 to 32767 or		
(HEX >8000 to >0000 to >7FFF)		
Line 110 moves type\$ GRAM/GROM		>110 CALL MOVES("G",8,GA,H\$)
to VDP. 8 bytes to be moved.		
GA is counter. H\$ is string		
for storing data found.		
Line 120 prints H\$ on screen.		>120 PRINT H\$
Line 130 next loop		>130 NEXT GA

### Options

Dependent on Assembly Language programmers and the RXB programs that are developed. MOVES is good for replacing those CALL LOAD loops. It also provides a means to rewrite XB while running XB instead of rewriting MERGE files then loading them. Future devices benefit from MOVES as it can copy or move different types of memory directly from or to them.



Format	NEW
	CALL NEW

#### Description

The NEW command is the same as the EXTENDED BASIC MANUAL page 126. NEW can only be used from edit mode. But now CALL NEW can be called from program mode. As expected all values are reset and all defined characters become undefined. Any open files are closed. Characters 32 to 95 are reset to their standard definitions. The TRACE and BREAK commands are canceled. The program is erased from memory.

#### Command

The line to the right will	>NEW
reset memory for XB.	

#### Programs

The program to the right will	>100 CALL NEW
reset memory for XB.	

```
CALL ONKEY(string-variable, key-unit,  
return-variable, status-variable)  
GOTO line-number[, ...]
```

## Description

ONKEY compares a string or string-variable characters one at a time to the key return-variable until a match is found. The string length may be longer than the number of GOTO line-number list. But a error results if that key is pressed as no line-number corresponds with the position of the key. If the string length is less than the number of GOTO line-numbers then the extra GOTO line-numbers are not used. The position of the characters in the string correspond to the GOTO line-number in the list. i.e. string "12345" GOTO 1,2,3,4,5 in the example:

```
CALL ONKEY("12345",0,K,S) GOTO 10,20,30,40,50
```

The key pressed like say 3 means line 30 will be used.

Another example:

```
10 CALL ONKEY("Crap",0,K,S) GOTO 22,29,34,41
```

If C is pressed then 22 is used.

If r is pressed then 29 is used.

If a is pressed then 34 is used.

If p is pressed then 41 is used.

## Programs

This line accepts a key>	>100 CALL ONKEY("123",0,K,S)
	GOTO 120,130,140
Keep scanning the key.	>110 GOTO 100
First line.	>120 PRINT "ONE":GOTO 100
Second line.	>130 PRINT "TWO":GOTO 100
Third line.	>140 PRINT "THREE":GOTO 100
Using GOSUB	>100 GOSUB 110::GOTO 100
Key scan.	>110 CALL ONKEY("YN",3,K,S)
	GOTO 120,130
First line.	>120 PRINT "YES":RETURN
Second line.	>130 PRINT "NO":RETURN

The above program both act like ON GOTO with the key selecting in the string the position and line number.

Format

CALL PEEKG(address,numeric-variable-list[,...])

Description

The PEEKG command reads data from GROM into the variable(s) specified. It functions identical to the regular EXTENDED BASIC PEEK command page 143. Except it reads from GROM/GRAM. GROM or GRAM address above 32767 must be converted to a negative number by subtracting 65536 from the desired address. Use CALL HEX to do this.

Programs

The program to the right will read a byte from GROM.

Address loop counter

PEEK Grom address value.

Convert to HEX

Show address and value.

Loop.

>100 CALL PEEKG(767,B)

|

|

>100 FOR D=-32768 TO 32767

>110 CALL PEEK(D,X)

>120 CALL HEX(A,H\$,X,B\$)

>130 PRINT "Address:";H\$,

D:"VALUE:";B\$,X

>140 NEXT D

|

---

Format           CALL PEEKV(address,numeric-variable-list[,...])

### Description

The PEEKV command reads data from VDP into the variable(s) specified. It functions identical to the regular EXTENDED BASIC PEEK command page 143. Except it reads from VDP.

The VDP address should not exceed 16384 in a TI with a 9918 VDP chip, 9938 or 9958 VDP chips can go the full 32767.

VDP addresses above 32767 must be converted to a negative number by subtracting 65536 from the desired address. Also whenever a value is peeked or poked to the screen a screen offset is present. 96 must be subtracted from or added to the value to correct it.

### Programs

The program to the right will		>100 CALL PEEKV(767,B)
read a byte from VDP and put		
it into variable B.		
This line will print it.		>110 PRINT B-96
Address loop counter		>100 FOR D=0 TO 16383
PEEK Grom address value.		>110 CALL PEEV(D,X)
Convert to HEX		>120 CALL HEX(A,H\$,X,B\$)
Show address and value.		>130 PRINT "Address: ";H\$,
		D:"VALUE: ";B\$,X
Loop.		>140 NEXT D

Format                      CALL POKEG(address,numeric-variable-list[,...])

#### Description

The POKEG command writes the data in the numeric variable list to GRAM at the specified address. It functions identical to the EXTENDED BASIC command LOAD page 115. Except that it writes to GRAM. GROM or GRAM addresses above 32767 must be converted to a negative number by subtracting 65536 from the desired address. CALL HEX is recommended for this.

#### Programs

The program to the right will | >100 CALL POKEG(1001,128)  
write 128 to GRAM address 1001|  
                                  |

Format                      CALL POKER(vdp-number,numeric-variable[,...])

                            CALL POKER(numeric-variable,number[,...])

#### Description

The POKER command writes to vdp register a byte value. Only registers 0 to 7 are valid. The byte value ranges 0 to 255.

#### Programs

This sets text mode.	>100 CALL POKER(7,244,1,240)
This is a delay loop.	>110 FOR L=1 TO 500 :: NEXT L
This sets multi color mode	>120 CALL POKER(1,232)
This is a delay loop.	>130 FOR L=1 TO 500 :: NEXT L
This sets bit map mode.	>140 CALL POKER(0,2,1,2)
This is a delay loop.	>150 FOR L=1 TO 500 :: NEXT L
This sets normal XB mode.	>160 CALL POKER(0,0)

-----  
Format           CALL POKEV(address,numeric-variable-list[,...])

#### Description

The POKEV command writes data to VDP into the address specified. It functions identical to the regular EXTENDED BASIC PEEK command page 143. Except it reads from VDP. The VDP address should not exceed 16384 in a TI with a 9918 VDP chip, 9938 or 9958 VDP chips can go the full 32767.

VDP addresses above 32767 must be converted to a negative number by subtracting 65536 from the desired address.

CALL HEX is recommended for this.

Also whenever a value is poked or peeked to the screen a screen offset is present. 96 must be subtracted from or added to the value to correct it.

#### Programs

The program to the right will | >100 CALL POKEV(767,65+96)  
write A at address 767.       |  
                                  |

Format

```
CALL PROTECT(pathname,filename,number[,...])

CALL PROTECT(string-variable,string-variable,
numeric-variable[,...])

CALL PROTECT(number,filename,number[,...])
```

#### Description

The PROTECT subprogram protects programs or files. Pathname may be up to 255 characters in length. Pathname must end in a period. The Myarc HFDC can only support a 29 character pathname plus a 10 character filename, so that would add up to 39 characters total. The pathname must end with a period and filenames must only be 10 characters in length. The number may be 0 to 255, any number other than 0 (zero) will protect a file. 0 unprotects. Up to 14 files on 14 different drives may be accessed in one command.

File error will be returned if the device is not accessed or the file or program doesn't exist. File error will be ignored when protecting a already protected file or program. File error will be reported if the disk notch is covered.

#### Programs

This line unprotects a file named A-FILENAME on RAMDISK B.		>CALL PROTECT("DSKB.", "A-FILE NAME,0)
Line 100 protects a file named JUNK on disk drive 3		>100 CALL PROTECT(3,"JUNK,255 )
Line 100 A\$ is drive 2 String-variable O\$ and N\$ are loaded with filenames.		>100 A\$="DSK2." :: O\$="LOAD1" :: N\$="filename"
Line 110 protects files.		>110 CALL PROTECT(A\$,O\$,1,A\$, N\$,1)
Line 110 unprotects file DIET on hard drive 1 in directory named FAT		>110 CALL PROTECT("WDS1.FAT." ,"DIET",0)



Format                      CALL QUITOFF

#### Description

The QUITOFF command disables the QUIT KEY. The QUIT KEY is already disabled upon entering RXB. See QUITON for more data.

#### Programs

The program to the right will | >100 CALL QUITOFF  
turn off the QUIT KEY.        |  
                                 |

Format                      CALL QUITON

#### Description

The QUITON command enables the QUIT KEY. The QUIT KEY is already disabled upon entering RXB. QUITON makes the QUIT once again functional. You may need to use this command before running certain programs that use the QUIT key.

#### Programs

The program to the right will | >100 CALL QUITON  
turn on the QUIT KEY.                |  
                                             |

Format	CALL RENAME(pathname,old-filename,new-filename [,...])
	CALL RENAME(pathname,old-directory-name,new-di rectory-name[,...])
	CALL RENAME(string-variable,string-variable, string-variable[,...])
	CALL RENAME(number,old-filename,new-filename [,...])

### Description

The RENAME subprogram renames directories or files. Pathname may be up to 255 characters in length. Pathname must end in a period. The Myarc HFDC can only support a 29 character pathname plus a 10 character filename, so that would add up to 39 characters total. The pathname must end with a period and filenames must only be 10 characters in length. RENAME can rename up to 15 files or directories on 15 different drives all in one command.

File error will be returned if the device is not accessed or the file or directory doesn't exist. File error will also be reported if renaming a protected file or directory. File error will be reported if the disk notch is covered. File error is also reported if the new-filename is already being used and is protected.

### Programs

This line renames a file named A-FILENAME on RAMDISK C to the new filename NEWNAME-1A	>CALL RENAME("DSKC.", "A-FILEN AME", "NEWNAME-1A")
Line 100 renames a file named JUNK to JUNK2 on disk drive 2	>100 CALL RENAME(2, "JUNK", "JU NK2")
Line 100 pathname A\$ String-variable O\$ and N\$ are loaded with filenames.	>100 A\$="DSK.TRASH.":: O\$="LO AD1" :: N\$="load1"
Line 110 rename LOAD1 to load1 on disk named TRASH	>110 CALL RENAME(A\$,O\$,N\$)
Line 110 access disk TRASH and first renames load1 to LOAD1 and renames LOAD1 to load1	>120 CALL RENAME(A\$,N\$,O\$,A\$, O\$,N\$)

Format	RES	(Uses default values)
--------	-----	-----------------------

RES initial line,increment

RES initial line,increment,start line-end line

#### Description

The RES command is the same as per Extended Basic Manual page 155. The RESEQUENCE command is deleted. The abbreviation RES is the only access name. The RES command now allows a portion of the program to be resequenced. This RES DOES NOT REPLACE any undefined line numbers with 32767. Any undefined line numbers in the program are left as is. This makes it easier to fix if a problem is present.

RES cannot be used to move lines from one location to another inside a program. If the new line numbers generated by the RES command would result in a line being moved, a Bad Line Number Error is generated. A Bad Line Number Error is also reported if there are no valid program lines between start line and end line.

#### Command

Lines 10 to 50 are renumbered.		>RES 20,1,10-50
Line 10 becomes 20, increment		
is 1.		
Lines 700-800 are renumbered.		>RES ,5,700-800
Line 700 becomes 100,		
increment is 5.		
Lines 50-80 are renumbered.		>RES ,,50-80
Line 50 becomes 100,		
increment is 10. (Default)		
Lines 1000 to last line are		>RES 1000,,750-
renumbered. Line 750 becomes		
1000, increment is 10.		
Lines to 400 are renumbered.		>RES ,20,-400
First Line becomes 100		
(Default), increment is 20.		
Line 40 is renumbered 20.		>RES 20,,40

## Format

```
CALL RMDIR(pathname,directory-name[,...])
```

```
CALL RMDIR(string-varialbe,string-variable
[,...])
```

## Description

The RMDIR subprogram ReMoves DIRectorys on hard drives. The pathname determines the device used and the pathname can be up to 255 characters in length. The Myarc HFDC can only support 29 characters pathnames plus the filename of 10, so that would add up to 39 characters total. The pathname must end with a period and the directory may only be 10 characters in length. RMDIR can remove up to 24 directories in 24 different drives in one command. RMDIR can also remove sub-directories then directories in the same command. Before any directory may be removed it must be empty of all files, or a error will be reported.

## Programs

```
This line removes a directory| >CALL RMDIR("WDS1.", "TEST")
named TEST on hard drive 1. |
```

```
This line removes | >100 CALL RMDIR("WDS1.ONE.TWO
sub-directory THREE of the | .", "THREE", "WDS1.ONE.", "TWO"
sub-directory TWO in main | , "WDS1.", "ONE")
directory ONE then removes |
sub-directory TWO of main |
directory ONE then finally |
removes directory ONE |
```

The above line will not work if the directory has files within a directory. CALL CUTDIR does not care though.

## Options

HARD DRIVE ACCESS ONLY!

Format

CALL RMOTION(#sprite-number[,...])

CALL RMOTION(ALL[,...])

### Description

The RMOTION subprogram reverses the row-velocity and column-velocity as numbers from -127 to 127. This means that RMOTION simply reverses the direction of the sprite specified so it goes in the opposite direction it was going in. This also means RMOTION ignores 0 and -128, so you can use those to bypass RMOTION if you do not want RMOTION to change the sprite. The fastest and slowest sprite speeds are never affected by RMOTION. This feature adds more power to RMOTION. The ALL feature also allows all sprites on the screen to reverse all at once. ALL may also be called as many times as wanted in a single program line.

See EXTENDED BASIC MANUAL MOTION PAGE 125, SPRITE PAGE 173, DELSPRITE PAGE 75 for more data.

### Program

RMOTION reverses the row-velocity and the column-velocity in sprite-number 1.		>100 CALL RMOTION(#1)
This line reverses the motion of all sprites.		>100 CALL RMOTION(ALL)
Line 100 sets up a sprite.		>100 CALL SPRITE(#1,33,2,96,18,99,84)
Line 110 waits for a number higher than .8 randomly.		>110 IF RND<.8 THEN 110
Line 120 reverses the motion of the sprite.		>120 CALL RMOTION(#1)
Continues the program.		>130 GOTO 110

### Options

While characters 144 to 159 are being used, you cannot use sprites.

Format            CALL SCSI (pathname, string-variable, ...]

## Description

The SCSI subprogram fetches a 44 byte package from the SCSI card and puts it into a string variable. This 44 byte package consists of 8 bytes internal, 8 bytes vendor ID, 16 bytes of product ID, 4 bytes revision value, 32 bit number of sectors, and 32 bit sector size. The pathname must end with a period.

## Programs

This line gets the 44 byte SCSI packet string from SCSI1.	>CALL SCSI("SCS1.",A\$)
This line gets the 44 byte SCSI packet string from SCSI1, SCS3, and SCS4.	>100 CALL ("SCS1.",X\$,"SCS3."  ,Y\$,"SCS4.",Z\$)

## Options

SCSI will only access a SCSI controller. See RXB Disk Manager program for use of SCSI subprogram.

```

Format      CALL SECTOR(pathname,read/write-flag,#sectors,
               sector-string,[,...])

              CALL SECTOR(number,number,number,string
               [,....])

              CALL SECTOR(string-variable,numeric-variable,
               numeric-variable,string-variable[,....])

```

#### Description

The SECTOR subprogram reads or writes sectors on disk or hard drives. The pathname determines the device used and the pathname can be up to 255 characters in length. The Myarc HFDC can only support 29 characters pathnames plus the filename of 10, so that would add up to 39 characters total. The pathname must end with a period and the directory may only be 10 characters in length. The read/write-flag may be any number to read sectors and 0 will write sectors. The #sectors ranges from 1 to 32 sectors being read/written at one time. The sector-string is a Hexadecimal string of the sector to read or write. Sector-string may be a "0" or up to "FFFFFFFF" or in other words in decimal form ranges from 0 to 4294967295 sectors. (2 Terabyte Hard Drive)

NOTE: The lower 8K for assembly support is used as a buffer for SECTOR so anything in the lower 8K will be corrupted. That means two things.

1. AMS support can store the sectors for duplication.
2. SECTOR is totally compatible with CORCOMP, MYARC, PARCOM, RAMDISKS, and SCSI drive controllers.

#### Programs

```

This line writes 1 sector 0 to| >CALL SECTOR("DSK1.",0,1,"0")
drive 1 from lower 8K.      . |
                               |
This line reads sector 0 and | >100 CALL SECTOR(2,1,2,"0")
1 from drive 2 to lower 8K.  |
This line puts the 2 sectors | >110 CALL MOVE("RV",512,8192,
onto the screen from the lower| 0)
8K. (See MOVES for info)     |
                               |
This line reads sector 1048575| >100 CALL SECTOR("SCS1.",9,
putting 32 sectors into lower | 32,"FFFFFF")
8K (32*256=8192)             |

```

#### Options

Only works when 32K available and destroys lower 8K data.



Format	SIZE
	CALL SIZE

#### Description

See EXTENDED BASIC MANUAL PAGE 169 for more data.

#### Command

May only be used from command mode.	>SIZE

#### Programs

May only be used from program mode.	>100 CALL SIZE
Shows memory used.	>100 CALL SIZE
Set up for Assembly support.	>110 CALL INIT
Shows memory used including Assembly space free.	>120 CALL SIZE
Set up for AMS switching.	>130 CALL AMSINIT
Shows memory used including AMS amount of K and RXB banks.	>140 CALL SIZE

#### Options

Unless you have a 32K installed Assembly support will not work. Also unless a AMS card is installed CALL AMSINIT will not work.

Format           CALL SWAPCHAR(character-code,character-code  
                 [,...])

#### Description

The SWAPCHAR subprogram switches the first character-code character definition with the second character-code character definition. That means they swap definitions. The characters range from 30 to 159.

#### Programs

Line 100 swaps character-code 65 with character-code 97.		>100 CALL SWAPCHAR(65,97)
Line 100 defines character-code 128 and character-code 159.		>100 CALL CHAR(128,"F0F0F0F0F0F0F0F0",159,"0F0F0F0F0F0F0F0F0")
Line 110 swaps them, then will swap space with character 128		>110 CALL SWAPCHAR(128,159,32,128)
Line 120 continues program.		>120 GOTO 110
Try this one on for weird.		>100 CALL SWAPCHAR(31,32,31,32)
		>110 CALL INVERSE(31)
		>120 GOTO 100

Format

CALL SWAPCOLOR(character-set,character-set  
[,...])

CALL SWAPCOLOR(#sprite-number,#sprite-number  
[,...])

### Description

The SWAPCOLOR subprogram swaps foreground and background colors of the first set with the second set. Or swaps the first sprite-number color with the second sprite-number color. The character-set numbers are given below:

set-number	character-codes
~~~~~	~~~~~
0 -----	30 to 31
1 -----	32 to 39
2 -----	40 to 47
3 -----	48 to 55
4 -----	56 to 63
5 -----	64 to 71
6 -----	72 to 79
7 -----	80 to 87
8 -----	88 to 95
9 -----	96 to 103
10 -----	104 to 111
11 -----	112 to 119
12 -----	120 to 127
13 -----	128 to 135
14 -----	136 to 143
(also sprite table) 15 -----	144 to 151
(also sprite table) 16 -----	152 to 159

Programs

The program to the right will swap foreground and background colors of set 15 with set 5.

Line 100 sets up two sprites on screen.

Line 110 swaps sprite #1 color with sprite #2 color.

Continue program.

>100 CALL SWAPCOLOR(15,5)

>100 CALL SPRITE(#1,65,2,99,9,9,9,#2,66,16,88,88,22,33)

>110 CALL SWAPCOLOR(#1,#2)

>120 GOTO 110

Format	CALL USER(quoted-string)
	CALL USER(string-variable)

Description

The USER subprogram overrides the normal editor of edit mode of XB and reads a DV80 file into the key scan routine as if the user was keying it in.

That means Batch Processing is creating XB programs from DV80 files, Editing XB programs, MERGING, Saving, and RUNNING XB programs. Also RESequencing, adding lines, or deleting lines, and re-writing lines from the DV80 file.

Every line to be input from the DV80 file MUST END WITH A CARRIAGE RETURN! A line of input may be up to 588 characters in length. The editor will error out if the crunch buffer is full, reporting a *Line Too Long* error. (Over 163 tokens)

Other errors will be reported but will not stop the process of USER continuing to input lines. To find errors in the DV80 file the input lines are shown on screen as they are input into the editor, and errors will be reported. So you must observe the screen for errors to test the DV80 file.

USER will stop after reaching the end of the file. But USER can have its operation suspended CALL POKEV(2242,0) will halt USER and CALL POKEV(2242,9) will resume USER.

INPUT and ACCEPT will try to read from USER if it is not turned off. On the other hand DV80 files can go directly into a INPUT or ACCEPT prompts. Turn off USER to be safe though.

USER will only report errors upon opening, thus if incorrect device or filename then USER reports * USER ERROR * and just closes the USER file, thus ending operation of USER.

Example files are included with RXB to show and explain the use of USER. The batch processing USER subprogram opens a new world to the RXB programmer. Possibilities are almost endless!

Programs

This line starts USER to use		>CALL USER("DSK1.FILENAME")
Batch processing on a file		
called FILENAME		
Line 100 is same as above.		>100 CALL USER("DSK1.FILE")
but within a program.		
Line 100 variable A\$ equals a		>100 A\$="DSK.VOLUME.FILE"
String-variable path name.		
Line 110 starts USER to use		>110 CALL USER(A\$)
Batch processing on A\$		

Programs

Save this program as LOAD. | >100 CALL USER("DSK1.BATCH")
 |

Here is an example DV80 file you save with the name BATCH.

```
! BATCH file for using  
NEW and CALL FILES and RUN. cr  
cr  
CALL XBPGM("DSK1.A-PROGRAM",#) cr
```

The above DV80 file uses cr to mean Carriage Return. And # is for the number of files you wish open. A-PROGRAM is the name of the XB program that needs a certain number of files open.

Options

To many to list out. See BATCH for demo.

Format CALL VCHAR(row,column,character-code)

 CALL VCHAR(row,column,character-code,
 repetition[,...])

Description

See EXTENDED BASIC MANUAL page 188 for more data. The only change to VCHAR is the auto-repeat function. Notice the new auto-repeat must have the repetitions used or it gets row confused with repetitions.

Programs

This line puts character 38 at| >100 CALL VCHAR(1,1,38,99,9,1
row 1 column 1 99 times, then| ,87)
puts character code 87 at |
row 9 column 1 |
 |
Fills screen with characters. | >100 CALL VCHAR(1,1,32,768,1,
 | 1,65,768,1,1,97,768,1,1,30,7
 | 68) :: GOTO 100
 |

Format CALL VERSION(numeric-variable)

Description

See EXTENDED BASIC MANUAL PAGE 190 for more data. Also see

Programs

This line will ask for version		>CALL VERSION(X)
and return current to numeric-		
variable X.		
Line 100 asks for version num.		>100 CALL VERSION(V)
Line 110 checks for version		>110 IF V>2001 THEN INPUT "DSK
to be larger than 240 and if		NAME":D\$:: INPUT "FILENAME"
it is will ask for input to		:F\$:: CALL XBPGM("DSK."&D\$&
use a new routine CALL XBPGM.		F\$) ELSE END

Options

Will always return current version of RXB.

Format CALL VGET(row,column,length,string-variable
 [,...])

Description

The VGET subprogram returns into a string-variable from the screen at row and column. Length determines how many characters to put into the string-variable. Row numbers from 1 to 24 and column numbers from 1 to 32. Length may number from 1 to 255. If VGET comes to the edge of the screen then it wraps to the other side.

Programs

The program to the right will | >100 CALL VGET(5,9,11,E\$)
put into string-variable E\$ |
the 11 characters at row 5 and |
column 9. |

The program to the right will | >100 CALL VGET(1,3,5,M\$,9,3,1
put into string-variable M\$ | ,Q\$,24,1,32,N\$)
the 5 characters at row 1 and |
column 3, then put into |
string-variable Q\$ the 1 |
character at row 9 and column |
3, then put into |
string-variable N\$ the 32 |
characters at row 24 and |
column 1. |

Format

CALL VPUT(row,column,string[,...])

CALL VPUT(row,column,string-variable[,...])

Description

The VPUT subprogram puts a string or string-variable onto the screen at row and column. The row numbers from 1 to 24 and column numbers from 1 to 32. If the string or string-variable being put onto screen goes to an edge it wraps to the other side. Unlike the EXTENDED BASIC DISPLAY AT the VPUT subprogram will not scroll the screen.

Programs

Line 100 puts string "THIS" on the screen at row 10 and column 4.

Line 110 sets string-variable A\$ equal to string "VPUT"

Line 120 puts string "is" at row 11 and column 5, then puts string-variable A\$ at row 10 and column 6.

Puts 456 at row 10 col 15

>100 CALL VPUT(10,4,"THIS")

>110 A\$="VPUT"

>120 CALL VPUT(11,5,"is",10,6,A\$)

>100 CALL VPUT(10,15,456)

```

Format      RUN "XB"

             DELETE "XB"

             CALL CAT("XB")

             OLD XB

             SAVE XB      -(Must have a program within
                           -memory to work at all)

             CALL XB
  
```

Description

The XB DSR (Device Service Routine) allows access to the RXB title screen. The access will work only if the DSR is in the GPLDSR or LINK DSR. In other words, a DSR that acknowledges any type of DSR in RAM, ROM, GROM, GRAM, or VDP. Most DSR's only accept DSK or PIO. Others like the SAVE or LIST commands will only work with a program in the memory first. Still others like CALL LOAD("XB") must have the CALL INIT command used first.

From EA option 5 you may type XB then enter, or from EA option 3 type XB then enter, then enter again. If the EA option 1 (edit), then 4 (print) type XB. From TI BASIC use OLD XB or DELETE "XB".

Keep in mind that if it does not work, the problem is the DSR your using. Almost all DSR's today only acknowledge the ROM or RAM DSR's. As the XB DSR is in GROM/GRAM it seems a bit short sighted on the part of most programmers to use cut down versions of a DSR. Please discourage this as it is a disservice to us all.

Programs

The program at the right will | >100 CALL EAPGM("XB")
 turn on the AUTO SELECTOR and |
 wait 4 second before switching |
 to the AUTO LOAD. |

This line asks for a string. | >100 INPUT A\$
 This line uses the string and | >110 DELETE A\$
 if you type XB then enter will |
 switch to the RXB. |

This line shows the CALL XB | >CALL XB

Options

BASIC and EA are also available.

Format CALL XBPGM("access-name")

 CALL XBPGM(string-variable)

 CALL XBPGM("access-name",file-number)

 CALL XBPGM(string-variable,numeric-variable)

Description

The XBPGM subprogram is like RUN in XB. (XB manual page 161)
The RUN subprogram can't run strings so special XB loader
programs were written and required. Using RUN A\$ results in
a error report of * syntax error * in normal XB.

XBPGM uses quotes like RUN or strings unlike RUN. So XBPGM
will run XB or BASIC programs from quoted or variables.
The file-number or numeric-variable denote the number of
files to be open before the XB program is loaded and run.
XBPGM first sets the number of files open, uses a NEW and
then runs the access string. See FILES for more info.

If a CALL XBPGM can't find the program or disk it will close
all files, clear all XB memory (Assembly lower 8K unaffected)
and leave you in XB command mode. You will know this by the
* Ready * and the cursor flashing below. This allows you to
try again with either RUN or CALL XBPGM again.

If an empty string is used XBPGM defaults to restart the
RXB title screen. See XB for more info.

Programs

The program at the right will load a XB Program named HOT from disk drive 2 then run it.		>100 CALL XBPGM("DSK2.HOT")
This line loads string GZ\$.		>100 GZ\$="DSK.XBGAMES.FROG"
This line uses the string path		>110 CALL XBPGM(GZ\$)
name to search all drives and		
RAMDISKS for a disk named		
XBGAMES and load a program		
named FROG then run that		
program.		
Line 100 should be added to		>100 CALL QUITON
most RXB program to allow the		>110 CALL XBPGM("DSKR.LOAD")
QUIT key to work for aborting		
XBPGM loader.		
CALL FILES(1) and run DSK1.TML		>100 CALL XBPGM("DSK1.TML",1)

```
*****
*                               RXB  Editor Assembler Version 2011      *
*****
REA is a new completely re-written Editor Assembler module. Any
code not needed was removed, and this left room for many new
features. First of these is the FCTN BACK key which is used as
an ESCAPE key, it will in respond as soon as it is pressed.
```

This is a copy of the REA title screen:

FIRMWARE CONTROL

RXB Editor & Assembler V=2011

- 0 SCREEN COLORS
- 1 EDITOR
- 2 ASSEMBLER
- 3 LOAD & RUN
- 4 RUN
- 5 RUN PROGRAM FILE
- 6 RXB LOADER
- 7 DIRECTORY
- . RXB

0 SCREEN COLORS

F is the foreground colors and B is the background color.

1 EDITOR

Has a arrow to indicate which option has been selected, thus the user will no longer make a mistake of saving a blank file over the original that he actually ment to load or save. Also a prompt is presented asking for the disk number or letter used to load the Editor (EDIT1 file). The ENTER key allows the loading of Editor by a pathname.

i.e. DSK.VOLUMENAME.EDIT1 or WDS1.DIRECTORY.SUBDIRECTORY.EDIT1

The directory will load the selected file if this option is used. See Directory for features.

2 ASSEMBLER

A prompt is presented asking for the disk number or letter used to load the Assembler (ASSM1 file). The ENTER key allows the loading of Assembler by a pathname.

i.e. DSK.VOLUMENAME.EDIT1 or WDS1.DIRECTORY.SUBDIRECTORY.ASSM1

The directory will load the selected file if this option is used. See Directory for features.

3 LOAD and RUN

The directory will load the selected file if this option is used. See Directory for features.

4 RUN

After loading a file all link names will be displayed including all support routines.

5 RUN PROGRAM FILE

By pressing a single key then enter, DSK#.UTIL1 is displayed and executed. # indicates the key pressed.

The directory will load the selected file if this option is used. The lower 8K support routines normally only loaded by the EA3 option are now loaded by this option too. So users can load FORTH, FORTRAM, and C programs from the EA5 prompt.

6 R X B LOADER

New feature that prompts for a XB program file to run. If the file or device errors out, then a return to RXB command mode is done. The * Ready * and a flashing cursor indicates the XB command mode. By pressing a single key then enter, DSK#.LOAD is displayed and executed. # indicates the key pressed.

The directory will load the selected file if this option is used. See Directory for new features

7 DIRECTORY

A new feature that prompts for a device name. EXAMPLE: DSK1. The period MUST be included if the full device name is used. Or type in a path name EXAMPLE: WDS1.DIRECTORY. The quicker way is to just type a number or letter then enter. Thus DSK#. is used and the key pressed represents the # used. While the catalog is being scrolled on screen, ANY key will end the display and reading of the disk, an arrow will appear next to the file read and the ARROW KEYS will move the arrow up or down. (FCNT/CTRL optional). To page forward or backward a screen at a time press left and right arrow keys. The file the arrow was last pointing to will stay at the top or bottom of the screen display. This is much better than other paging methods like DM1000 or Funnel Web Disk review.

ONLY the SPACE BAR will pause the catalog until pressed again.

2011 added new keys to Directory:

V = View file.
A = Assembler file.
G = GPL Assembler file.

Use ENTER to select the filename so it will be placed into a buffer, the cataloger will automatically load Dis/Fix 80 files into the EA3 menu, Programs will be EA5, and only Dis/Var 254 is considered to be XB programs. To load XB programs use the SPACE BAR to buffer the filename, loading is automatic from there. For DIS/VAR 80 or DIS/FIX 80 files to be edited or assembled use ENTER or SPACE BAR, then select the Edit or Assembler from the main menu. Loading is automatic from there.

Directory will automatically assume you wish to catalog a sub-directory if a Directory was selected. To buffer anything else you must use the SPACE BAR, to select a filename to be placed into a buffer, then auto return to REA main menu. Now select the option to be used from this buffer.

If you select 7 DIRECTORY again, the buffer will be used and the last device accessed will be used again. If you wish to clear the buffer just use FCTN BACK to the REA main menu.

. R X B

A previous feature that was optional since version 1000 but had no menu option on screen indicating it was a option. (Period) . will return to RXB menu screen.

SYSTEM SUPPORT

The modified version of the Editor/Assembler no longer supports the 99/4 computer. A 99/4A is required. All TI BASIC support routines (CALL INIT, CALL LINK, CALL LOAD, CALL PEEK, CALL PEEKV, CALL POKEV, and CALL CHARPAT) have been removed from the Editor/Assembler. If you have a program that must be run from TI BASIC and requires these routines, you must plug an Editor/Assembler module into the cartridge connector. There are some assembly language programs that access data internal to the Editor/Assembler cartridge. These programs will not run correctly due to the re-structuring of the data in the Editor/Assembler module. For these programs you must use your Editor/Assembler cartridge. On the other hand like FunnelWeb REA loads the support routines before EA3 or EA5 loaders to engage, so C, FORTRAM, and FORTH will load from the EA5 prompt.

NO 32K NEEDED TO WHAT?

REA has been totally re-written so the user can now use some of the features of REA without that nasty *NO MEMORY EXPANSION* error turning up. The error routine only disallows the user from accessing those aspects of REA that absolutely needs 32K to work.

The user may now use the REA EDITOR PRINT FILE menu, or use the 6 RXB file loader menu, or use the 7 DIRECTORY menu.

That means with RXB and REA the user can now print files, view files, load any BASIC or XB program and catalog from REA with or without a 32K memory.

AMS MAPPER

The AMS card has tons of documents as to its function and use.
So to re-explain these docs would be pointless. Read the docs or
find some, sorry but the RXB package is already huge.

In PASS mode the mapper register setup is equivalent to:

mapper address		mapper	page num		address range
HEX	Dec		HEX	Dec	memory area
>4004	= 16388	is MR02	= >02	= 02	points to >2000 - >2FFF range
>4006	= 16390	is MR03	= >03	= 03	points to >3000 - >3FFF range
>4014	= 16404	is MR10	= >0A	= 10	points to >A000 - >AFFF range
>4016	= 16406	is MR11	= >0B	= 11	points to >B000 - >BFFF range
>4018	= 16408	is MR12	= >0C	= 12	points to >C000 - >CFFF range
>401A	= 16410	is MR13	= >0D	= 13	points to >D000 - >DFFF range
>401C	= 16412	is MR14	= >0E	= 14	points to >E000 - >EFFF range
>401E	= 16414	is MR15	= >0F	= 15	points to >F000 - >FFFF range

(MR=Mapper Register)

In MAP mode the mapper register setup is equivalent to: EXAMPLE1

mapper address		mapper	page num		address range
HEX	Dec		HEX	Dec	memory area
>4004	= 16388	is MR02	= >10	= 16	points to >2000 - >2FFF range
>4006	= 16390	is MR03	= >11	= 17	points to >3000 - >3FFF range
>4014	= 16404	is MR10	= >12	= 18	points to >A000 - >AFFF range
>4016	= 16406	is MR11	= >13	= 19	points to >B000 - >BFFF range
>4018	= 16408	is MR12	= >14	= 20	points to >C000 - >CFFF range
>401A	= 16410	is MR13	= >15	= 21	points to >D000 - >DFFF range
>401C	= 16412	is MR14	= >16	= 22	points to >E000 - >EFFF range
>401E	= 16414	is MR15	= >17	= 23	points to >F000 - >FFFF range

(MR=Mapper Register)

In MAP mode the mapper register setup is equivalent to:
EXAMPLE2

mapper address		mapper	page num		address range
HEX	Dec		HEX	Dec	memory area
---	---		---	---	-----
>4004	= 16388	is MR02	= >19	= 31	points to >2000 - >2FFF range
>4006	= 16390	is MR03	= >01	= 01	points to >3000 - >3FFF range
>4014	= 16404	is MR10	= >09	= 09	points to >A000 - >AFFF range
>4016	= 16406	is MR11	= >00	= 00	points to >B000 - >BFFF range
>4018	= 16408	is MR12	= >07	= 07	points to >C000 - >CFFF range
>401A	= 16410	is MR13	= >18	= 30	points to >D000 - >DFFF range
>401C	= 16412	is MR14	= >05	= 05	points to >E000 - >EFFF range
>401E	= 16414	is MR15	= >04	= 04	points to >F000 - >FFFF range

(MR=Mapper Register)

As you can see you can switch pages around all you want.

Each 4K page of the AMS memory can be engaged even into every 4K section of the 32K at once. Thus giving multiple copies of the same 4K in every 4K chunk. Doing this has few uses but it demonstrates the flexibility of the AMS mapper.

RXB limits the ability of the AMS slightly. All the memory of the AMS is used like this:

1. 32K is used for pass mode. Think of it like a normal TI. 24K for XB programs and variables, and lower 8K assembly support.
2. 32K is used for future use. Stashed away for RXB future use. Pages 0,1,4,5,6,7,8,9 are for future use.
3. 4K pages of the rest of the AMS are for AMSBANK, a RXB routine that needs two (2) 4K pages of the AMS for the lower 8K of Assembly support. Pages 16 to 255 AMSBANK.
4. If a 256K AMS is used by RXB then 32K PASS mode, 32K future use, and the rest for AMSBANK. So $32+32=64$ and $256-64=192$ then 192K of AMS 4K pages would be $192/4=48$ or 48 pages of AMSBANK. Now the lower 8K needs two 4K pages because of course $8/4=2$ so $48/2=24$ and that means 24 8K assembly support routines from one XB program.

5. BSAVE and BLOAD in RXB are subroutines to save or load the lower 8K as Program Image files. 33 sectors in length they are only used by BSAVE or BLOAD. The reasons for BSAVE is to save the lower 8K in one easy to load chunk. ALSAVE that creates hidden loaders has one huge disadvantage. ALSAVE can only save one assembly support program at a time. Also it can't load more than one support routine and XB program to run it. BLOAD can load many support routines into the AMS and control them from one XB program. BLOAD and BSAVE can do what ALSAVE can't. Also up to 1Meg can be loaded into the AMS from one XB program and the same program run the assembly support routines.
6. ISR (Interrupt Service Routine) hooks are used by many XB programs so CALL ISROFF and CALL ISRON are for RXB to control the problems created by them. Also compatibility between software packages is ensured.
7. AMSBANK is the primary way RXB manages memory.
pages 16 to 255 are numbers in AMSBANK as 0 to 240 so
pages 2,3,10,11,12,13,14,15 are for pass mode and
pages 0,1,4,5,6,7,8,9 are unused for future use.
Thus $240 \times 4 = 960K$ of AMS is in AMSBANK.