

```

*****
*                               TABLE OF CONTENTS                               *
*****
NAME-----TYPE-----ACCESS-----PAGE---#
ALL              OPTION SUBPROGRAM              A1
ALPHALOCK        SUBPROGRAM          JOYSTICK/KEYBOARD A2
BASIC            DSR or SUBPROGRAM    DEVICE          B1
BEEP             SUBPROGRAM          SOUND            B2
BIAS             SUBPROGRAM          CONVERSION       B3
BYE              SUBPROGRAM/COMMAND  EXIT RXB         B4
CAT              SUBPROGRAM          DISK or HARD DRIVE C1
CHAR             SUBPROGRAM          SCREEN           C2
CHARSET          SUBPROGRAM          SCREEN           C3
CLEARPRINT       SUBPROGRAM          SCREEN           C4
CLSALL           SUBPROGRAM          DISK             C5
COINC            SUBPROGRAM          SPRITE           C6
COLLIDE          SUBPROGRAM          SPITE            C7
COLOR            SUBPROGRAM          SPRITE           C8
COPY             COMMAND              EDITOR LINES     C9
DEL              COMMAND              EDITOR LINES     D1
DIR              SUBPROGRAM          DISK or HARD DRIVE D2
DISTANCE         SUBPROGRAM          SPRITE           D4
EA               DSR                  DEVICE           E1
EXE              SUBPROGRAM          ASSEMBLY SUPPORT E3
EXECUTE          SUBPROGRAM          ASSEMBLY SUPPORT E4
FILES            SUBPROGRAM          DISK             F1
GCHAR            SUBPROGRAM          SCREEN           G1
GO               OPTION SUBPROGRAM  SPRITE           G2
GMOTION          SUBPROGRAM          SPRITE           G3
HCHAR            SUBPROGRAM          SCREEN           H1
HEX              SUBPROGRAM          CONVERSION       H2
HGET             SUBPROGRAM          SCREEN           H4
HONK             SUBPROGRAM          SOUND            H5
HPUT             SUBPROGRAM          SCREEN           H6
INIT             SUBPROGRAM          ASSEMBLY SUPPORT I1
INVERSE          SUBPROGRAM          CHARACTER        I2
IO               SUBPROGRAM          TSM9901 CONTROL I3
ISROFF           SUBPROGRAM          INTERRUPTS       I14
ISRON            SUBPROGRAM          INTERRUPTS       I15
IV254            OPTION SUBPROGRAM  DISK             I16

```

```

*****
*                                     *
*               TABLE OF CONTENTS   *
*                                     *
*****
NAME-----TYPE-----ACCESS-----PAGE---#
JOYST          SUBPROGRAM          JOYSTICKS          J1
JOYLOCATE      SUBPROGRAM          JOYSTICK & SPRITES J2
JOYMAP         SUBPROGRAM          JOYSTICK & SPRITES J3
JOYMOTION      SUBPROGRAM          JOYSTICK & SPRITES J4
KEY            SUBPROGRAM          KEYBOARD          K1
LIST           COMMAND             SCREEN/DSK/PRINTER L1
LOAD           SUBPROGRAM          DISK/ASSEMBLY      L2
MAGNIFY        SUBPROGRAM          SPRITE            M1
MAP (SAMS)     OPTION SUBPROGRAM   SAMS MEMORY        M2
MERGE          COMMAND             DISK/FILES          M3
MOD            SUBPROGRAM          CONVERSION          M4
MOTION         SUBPROGRAM          SPRITE            M5
MOVE           COMMAND             EDIT LINES          M6
MOVES          SUBPROGRAM          MEMORY (ALL TYPES) M7
NEW            SUBPROGRAM or COMMAND MEMORY (XB)        N1
OFF (SAMS)     OPTION SUBPROGRAM   SAMS MEMORY        01
ON (SAMS)      OPTION SUBPROGRAM   SAMS MEMORY        02
ONKEY          SUBPROGRAM          KEYBOARD          03
PASS (SAMS)    OPTION SUBPROGRAM   SAMS MEMORY        P1
PATTERN        SUBPROGRAM          SPRITE/CHARACTER   P2
PEEKG          SUBPROGRAM          GROM               P3
PEEKV          SUBPROGRAM          VDP                P4
PLOAD          SUBPROGRAM          DISK or HARD DRIVE P5
POKEG          SUBPROGRAM          GRAM               P7
POKER          SUBPROGRAM          VDP REGISTERS      P8
POKEV          SUBPROGRAM          VDP                P9
PRAM           SUBPROGRAM          RAM MEMORY         P10
PSAVE          SUBPROGRAM          DISK or HARD DRIVE P11
QUITOFF        SUBPROGRAM          KEYBOARD           Q1
QUITON         SUBPROGRAM          KEYBOARD           Q2

```

*****			
*	TABLE OF CONTENTS		*
*****			
NAME-----	-----TYPE-----	ACCESS-----	PAGE--##
RANDOMIZE	SUBPROGRAM	INITIALIZE SEED	R1
RES	COMMAND	EDIT LINES	R2
RMOTION	SUBPROGRAM	SPRITE	R3
RND	SUBPROGRAM	CONVERSION	R4
ROLLDOWN	SUBPROGRAM	SCREEN	R5
ROLLLEFT	SUBPROGRAM	SCREEN	R6
ROLLRIGHT	SUBPROGRAM	SCREEN	R7
ROLLUP	SUBPROGRAM	SCREEN	R8
SAMS	SAMS CONTROL	SAMS MEMORY	S1
SAVE	COMMAND	DISK or HARD DRIVE	S5
SCREEN	SUBPROGRAM	SCREEN	S6
SCROLLEDOWN	SUBPROGRAM	SCREEN	S7
SCROLLLEFT	SUBPROGRAM	SCREEN	S8
SCROLLRIGHT	SUBPROGRAM	SCREEN	S9
SCROLLUP	SUBPROGRAM	SCREEN	S10
SIZE	SUBPROGRAM/COMMAND	MEMORY (ALL)	S11
STOP	OPTION SUBPROGRAM	SPRITES	S14
SWAPCHAR	SUBPROGRAM	CHAR	S15
SWAPCOLOR	SUBPROGRAM	COLOR	S16
USER	SUBPROGRAM	CONTROL/DISK DOS	U1
VCHAR	SUBPROGRAM	SCREEN	V1
VDPSTACK	SUBPROGRAM	VDP STACK LOCATION	V2
VERSION	SUBPROGRAM	GROM RXB	V3
VGET	SUBPROGRAM	SCREEN	V4
VPUT	SUBPROGRAM	SCREEN	V5
XB	DSR or SUBPROGRAM	DEVICE	X1



Format	CALL CHAR(ALL[,...])  CALL CHARSET(ALL)  CALL COINC(ALL,numeric-variable[,...])  CALL COLOR(ALL,foreground-color,background-color [,...])  CALL DELSPRITE(ALL[,...])  CALL INVERSE(ALL[,...])  CALL MOTION(ALL,row-velocity,column-velocity [,...])  CALL RMOTION(ALL,[,...])
--------	---

#### Description

The ALL command is used as a option in many subprograms. Each option by ALL is slightly different so find the above subprogram to find that use of the ALL option.

#### Programs

See each subprogram for examples of use of ALL.

-----

Format            CALL ALPHALOCK(numeric-variable)

### Description

The ALPHALOCK detects if the key ALPHALOCK key is on or off. If the ALPHALOCK key is off the numeric-variable will be 0. But if the ALPHALOCK key is on numeric-variable will be non zero i.e. -26294 just for giggles that is hex >994A ALPHALOC runs from ROM.

### Programs

Check ALPHALOCK key on/off	>100 CALL ALPHALOCK(N)
Show the value of N	>110 PRINT N
If N not zero then HONK	>120 IF N THEN CALL HONK
Loop forever	>130 GOTO 100

Format	RUN "BASIC"
	DELETE "BASIC"
	CALL XB("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

Will go to BASIC prompt	>100 CALL XB("BASIC")
This line asks for a string.	>100 INPUT A\$
If string A\$="BASIC" will go	>110 DELETE A\$
will switch to BASIC.	
Will switch to BASIC.	>CALL BASIC
Lower case also works!	>CALL EA("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 FOR BIAS.

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

### Programs

The program to the right will load X\$ with 255 characters off the screen. But will not be readable due to a bias.	>100 CALL MOVES("V\$",255,511,X\$)
The bias is now subtracted from the string printed.	>110 CALL BIAS(0,X\$)
	>120 PRINT X\$
Set up a string to use	>100 Y\$="This is a test!"
Remove add BIAS to string	>110 CALL BIAS(1,Y\$)
Put the string onto screen	>120 CALL MOVES("\$V",15,Y\$,96)

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	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. Must be a Y so reset system.	>110 INPUT "END PROGRAM":A\$   >120 IF A\$<>"Y" THEN 110   >130 CALL BYE

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

---

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

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

### Description

See EXTENDED BASIC MANUAL page 56 for more data. Addition characters 30 to 159 by redefined, but this affects sprites. Now 30 (CURSOR) and 31 (EDGE CHARACTER) to be redefined. Also 144 to 159 may be redefined if sprites are not used. Pattern-identifier increased from 64 to 240 string. Thus up to 15 characters may be defined in single command, 4 was old limit in XB allowed to be defined in XB manual page 56 CHAR runs from ROM.

### 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 CHARSET(ALL)
This line repeats the program.		>150 GOTO 100
Sets variable A\$ up.		>100 A\$="FF818181818181FF"
Define all the characters same		>110 CALL CHAR(ALL,A\$)
This line defines the cursor.		>120 CALL CHAR(30,"FF81FF")
This line defines the edge character.		>130 CALL CHAR(31,"55")

### Options

Sprites may not be used if characters 144 to 159 are being redefined for use. 15 characters now defined up from 4 in XB.

## Format

CALL CHARSET

CALL CHARSET(ALL)

## Description

The CHARSET command is just like the CHARSET command, but it resets characters from 30 to 127. CHARSET thus resets 32 to characters to 95 only. Exactly like CHARSET it also resets colors to original mode. CALL CHARSET(ALL) resets all the characters from 30 to 159 and all colors to original. This is same as you would expect from TI Basic or RXB when not using SPRITES, or just the first few SPRITES. CHARSET runs from ROM.

## Programs

This resets all characters and colors to original.	>100 CALL CHARSET(ALL)
Set up a loop.	>100 FOR X=30 to 127
Show characters on screen.	>110 PRINT CHR\$(X);
Set all colors the same.	>120 CALL COLOR(ALL,14,10)
Set each character definition.	>130 CALL CHAR(X,"FF00FF00FF")
Continue loop.	>140 NEXT X :: CALL BEEP
Press any key.	>150 CALL KEY("",5,K,S)
Reset all characters.	>160 CALL CHARSET(ALL)
Restart it.	>170 GOTO 100

---

Format            CALL CLEARPRINT

#### Description

The CLEARPRINT command is just like the CLEAR command, but it clears columns 3 to 28 that is the PRINT or DISPLAY area leaving columns 1 and 2 along with columns 31 and 32 as are. Use CLEARPRINT to take the place of CALL HCHAR loops. CLEARPRINT runs from ROM.

#### Programs

Shows what CLEARPRINT does	CALL CLEARPRINT
Set loop chars 30 to 159	>100 FOR X=30 to 159
Show characters on screen.	>110 CALL HCHAR(1,1,X,768)
Clear columns 3 to 28	>120 CALL CLEARPRINT
Delay loop	>130 FOR Y=1 TO 200::NEXT Y
Next character	>140 NEXT X
Loop forever	>150 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 CLOSE all open files.	>100 CALL CLSALL
This opens the printer.	>100 OPEN #9:"PIO",OUTPUT
This opens a disk file JUNK.	>110 OPEN #2:"DSK1.JUNK",INPUT
This opens a RS232 port.	>120 OPEN #4:"RS232",OUTPUT
This opens a disk file CRAP.	>130 OPEN #7:"DSK2.CRAP",OUTPUT
This closes all files.	>140 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. COINC runs from ROM.

## 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 COLLIDE(#sprite-number,#sprite-number,  
                  tolerance,dot-row,dot-column[,...])

                  CALL COLLIDE(#sprite-number,dot-row,  
                  dot-column,tolerance,dot-row,  
                  dot-column[,...])

### Description

See EXTENDED BASIC MANUAL PAGE 64 has COINC. The XB COINC never tells you the location of a sprite and absolutely limits the types of way sprites could be used. If sprites COLLIDE where did this happen? COLLIDE tells you exactly where they did collide and the location of how close to the hit box you wanted be informed. Tolerance could be up to 256 pixels which could always be a collide result or 0 for exactly on pixel of top left corner of the sprite. I recommend a setting of 6 for best results. COLLIDE runs from ROM.

### Programs

Clear screen	>100 CALL CLEAR ! SPRITES
Set up 3 sprites to be on screen	>110 CALL SPRITE(#1,65,2,9,99, ,20,22,#2,66,2,64,99,X,30,25, ,#3,67,2,9,99,-20,-35)
COLLIDE scans 3 sprites for sprite hits on #1,#2,#3 sprite	>120 CALL COLLIDE(#1,#2,8,R1, C1,#1,#3,8,R2,C2,#2,#3,8,R3, C3)
Check for non zero?	>130 IF R1+C1+R2+C2+R3+C3
If zero loop forever	THEN 140 ELSE 120
Show hits or non hits	>140 PRINT "#1";R1;C1;"#2";R2; ;C2;"#3";R3;C3
Zero out variables	>150 R1,C1,R2,C2,R3,C3=0
Loop forever	>160 GOTO 120
Clear screen	>100 CALL CLEAR ! ROW: COLUMN
Set up 3 sprites to be on screen	>110 CALL SPRITE(#1,65,2,9,99, 20,22,#2,66,2,64,99,30,25,#3, 67,2,9,99,-20,-20)
COLLIDE for DOT ROW DOT COLUMN at row 99 and column 99 for sprites #1,#2,#3 hit?	>120 COLLIDE(#1,99,99,8,R1,C1, #2,99,99,8,R2,C2,#3,99,99,8, R3,C3)
Check for non zero?	>130 IF R1+C1+R2+C2+R3+C3
If zero loop forever	THEN 140 ELSE 120
Zero out variables	>150 R1,C1,R2,C2,R3,C3=0
Loop forever	>160 GOTO 120

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, presently modifications to the COLOR subprogram is ALL will change character sets 0 to 14 to the same foreground and background colors.

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

All characters set foreground		>100 CALL COLOR(ALL,1,2,ALL,
transparent and background 1		2,1) :: GOTO 100
Swap characters set colors		>100 CALL COLOR(S,3,5)

### Options

Characters 144 to 159 cannot be used with 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 than Memory Full Error 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 unoccupied lines will be at the  
beginning of the block.

## Commands

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

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 deleted.		>DEL 5-
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 the string A\$	>100 A\$="DSK.ADISKNAME"
This line uses A\$ for the name of the device to catalog.	>110 CALL DIR(A\$)
This line will catalog drive 4 if N=4	>100 CALL DIR(N)
This line will catalog drive C if X=67 (ASCII 67 is C)	>100 CALL DIR(X)
This line is path name.	>10 V\$="WDS1.VOLUME.SUB-DIR."
This line will catalog device WDS1 for directory VOLUME and catalog SUB-DIR	>20 CALL DIR(V\$)
This line catalogs drives 1 then 2 then 3 then WDS1	>100 CALL DIR(1,2,3,"WDS1.")

Format           CALL DELSPRITE(#sprite-number[,...])

CALL DELSPRITE(ALL)

### Description

The only thing added by RXB to DELSPRITE is the auto repeat.

See EXTENDED BASIC MANUAL page 80 for more data.

DELSprite runs from ROM to delete sprites.

### Program

The program at the right will set up 3 sprites on screen and start them moving.	>100 CALL CLEAR >110 CALL SPRITE(#1,65,7,99,9 9,0,10,#2,66,5,99,99,10,0,#3 ,67,2,1,2,-50,-50)
Deletes all sprites on screen Restart loop	>120 CALL DELSPRITE(ALL) >130 GOTO 110
The program at the right will set up 3 sprites on screen and start them moving	>100 CALL CLEAR >110 CALL SPRITE(#1,65,7,99,9 9,0,10,#2,66,4,99,99,10,0,#3 ,67,2,1,2,-50,-50)
Loop delay	>130 FOR L=1 TO 1000::NEXT L
Randomly delete sprite of the 3 sprites on screen	>120 CALL DELSPRITE(#INT(RND* 3)+1)
Loop delay	>140 FOR L=1 TO 1000::NEXT L
Loop program	>150 GOTO 110

### Options

While characters 144 to 159 are being used, you cannot use  
sprites. The DELSPRITE routine deletes all sprites or a  
chosen sprite.



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.  
DISTANCE runs from ROM.

### 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	RUN "EA"	
	DELETE "EA"	
	CALL XB("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. Almost all DSR's today only acknowledge the ROM or RAM DSR's.

### Programs

Go to the Editor Assembler.	>100 CALL XB("EA")
This line asks for a string. Type EA will go to EA module	>100 INPUT A\$ :: DELETE A\$
Switch to Editor Assembler	>CALL CAT("EA")
Lower case can also be used.	>call ea
Strange looping effect.	>CALL EA("EA")

### Options

BASIC and XB are also available.

Format                CALL EA

#### Description

The EA subprogram is used to switch to the Editor Assembler

#### Description Addendum

EA only works from EXTENDED BASIC, not BASIC.

#### Programs

The program at the right will		>100 CALL EA("DSK2.FW")
switch to Editor Assembler		

---

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

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

#### Description

The EXE subprogram directly goes to the cpu-address >8300 using the GPL XML >F0 and expects to work like an assembly BL @address so EXE(address) put that address at >8300 thus to return you need an assembly RT to end. Programmers can see this is a BL 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 RT will end the BL as long as registers R11, R13, R14, R15 are not changed. By using CALL LOAD or CALL MOVES the programmer can set up a BL routine in the lower 8K by filling the registers with values first, then using CALL EXE to directly complete these commands. This is faster then CALL LINK as no interpretation of the access or values are checked.

Here is the example of using EXE doing VDP garbage collect the VDP memory not needing Memory Expansion but using the XB ROM 1 COMPCT routine.

CALL EXE(29656) ! does VDP COMPCT garbage collection  
or execute the SCROLL routing in XB ROM 1  
CALL EXE(31450) ! does SCROLL screen routine

FOR L=1 TO 4 :: CALL EXE(31450) :: NEXT L  
This would scroll the screen 4 times like PRINT does.

---

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

```

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 thus faster.

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 0 to 15 unlike the other FILES routines that can only accept 0 to 9. Each open file reduces VDP by 534 bytes, plus each file opened will use 518 bytes more. Only RXB has a valid CALL FILES(0) or a CALL FILES(15) that works.

## 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)
	>SIZE
Only possible in RXB	>CALL FILES(0)
	>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. Also 0 files now is possible in RXB.

## Options

See XB for even more powerful applications made easy.

For example CALL XB("DSK1.LOAD",1) will do a CALL FILES(1) then NEW then RUN "DSK1.LOAD" AUTOMATICALLY

---

Format           CALL FILES(number)

                  CALL FILES(numeric-variable)

If you use CALL FILES and SIZE:

CALL FILES(15)	5624	Stack	Bytes	Free
CALL FILES(14)	6142	Stack	Bytes	Free
CALL FILES(13)	6660	Stack	Bytes	Free
CALL FILES(12)	7178	Stack	Bytes	Free
CALL FILES(11)	7696	Stack	Bytes	Free
CALL FILES(10)	8214	Stack	Bytes	Free
CALL FILES(9)	8732	Stack	Bytes	Free
CALL FILES(8)	9250	Stack	Bytes	Free
CALL FILES(7)	9768	Stack	Bytes	Free
CALL FILES(6)	10286	Stack	Bytes	Free
CALL FILES(5)	10804	Stack	Bytes	Free
CALL FILES(4)	11322	Stack	Bytes	Free
CALL FILES(3)	11840	Stack	Bytes	Free
CALL FILES(2)	12358	Stack	Bytes	Free
CALL FILES(1)	12876	Stack	Bytes	Free
CALL FILES(0)	13394	Stack	Bytes	Free

This shows the free VDP memory for XB programs and how it is calculated is first free address in high VDP memory minus VDP Stack Address plus 64 bytes in lower VDP memory.

You should notice each CALL FILES is 518 bytes in size.

Lastly RXB created CALL FILES(0) and is not the same as any other CALL FILES(0) by others as RXB follows TI standards to be consistant and predictable.



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

#### Options

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

-----

Format           CALL MOTION(GO[,...])

### Description

The GO command is a option in the MOTION subprogram.  
GO does exactly what you would expect starts all sprite  
motion by making them use previous sprite motion table.  
MOTION runs from ROM.

### Programs

See MOTION subprogram for examples of use of GO.

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. GMOTION runs from ROM.

### Program

GMOTION returns the row-velocity into X and the column-velocity into Y.

Set up screen and up,down, left, right variables A(0) and A(1)  
Loop for 28 sprites.  
Set up 28 random sprites with random colors and motion.

Loop counter.  
Random sprite selector, get that sprites motion, put the values on screen.

Delay loop.  
Clear screen and Z+1.  
Loop till Z>8

```
>100 CALL GMOTION(#1,X,Y)

>100 A(0)=-1::A(1)=1::CALL CLEAR::CALL MAGNIFY(2)::CALL SCREEN(15)
>110 FOR S=1 TO 28
>120 CALL SPRITE(#S,64+S,INT(RND*16)+1,20+S,50+S,INT(A(RND*1))*INT(RND*127),INT(A(RND*1))*INT(RND*127))
>130 NEXT S
>140 S=INT(RND*28)+1::CALL GMOTION(#S,X,Y)::CALL HPUT(24,3,"CALL GMOTION("#&STR$(S)&"&STR$(X)&","&STR$(Y)&"))
>150 FOR L=1 TO 1E3::NEXT L
>160 CALL CLEAR::Z=Z+1::IF 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 188 for more data. The only syntax 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. Also RXB HCHAR is now in ROM.

#### Programs

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

#### Options

CALL HCHAR is now written in Assembly so much faster is faster than normal XB, also as separate line numbers are needed to continue placing characters on screen. See VCHAR, HPUT, VPUT, HGET and VGET.

---

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. HEX runs from ROM.

HEX subroutine in RXB is for Assembly mostly but is usefull for new RXB routines like VDPSTACK or PRAM or EXECUTE. HEX runs from ROM.

---

### 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"

### Options:

See LOAD and EXECUTE for better utilities for Assembly or GPL access. Also useful as better then a calulator.

---

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. HGET runs from ROM.

#### Programs

The program to the right will  
put into string-variable E\$  
the 11 characters at row 5 and  
column 9.

>100 CALL HGET(5,9,11,E\$)

The program to the right will  
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.

>100 CALL HGET(1,3,5,M\$,9,3,1  
                 ,Q\$,24,1,32,N\$)

#### Options:

See HPUT, VPUT, and VGET.

## 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 100



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. HPUT runs from ROM.

### 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)

### Options:

CALL HPUT is now written in Assembly so much faster is faster than normal then XB DISPLAY AT(row,column)  
See HCHAR, VCHAR, HGET and VGET.

Format

CALL INIT

## Description

The INIT command is the same as the EXTENDED BASIC MANUAL page 101. Originally INIT loaded more data then 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 first.

## \* NOTE \*

RXB only loads up to >24F4 first open byte. Reasons unknown XB loads useless junk from >24EA to >25FF that seems to be a programming error loading 277 useless bytes. Thus normal XB over writes these 277 bytes.

## 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 INVERSE all character-code (A) in the character definition table in memory.	>100 CALL INVERSE(65)
The program to the right will INVERSE all character-codes from 30 to 143.	>100 CALL INVERSE(ALL)
Line 100 will ask for a string of characters terminated by the ENTER key.	>100 INPUT A\$
Line 110 is a loop to counter.	>110 FOR L=1 TO LEN(A\$)
Line 120 singles each one of the characters in A\$.	>120 C=ASC(SEG\$(A\$,L,1))
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 start with and ends with.	>280 FOR AD=4096 TO 4160 STE
Line 290 reads list.	P 4
Line 300 moves them into VDP.	>290 READ V1,V2,V3,V4
	>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 on how to test IO.	>330 PRINT "CRASH": : "TYPE:":
	"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 :: PRINT "*SOUND DATA TABLE CREATOR*"
110 Q$="0123456789ABCDEF"
120 INPUT "GENERATOR # ?":GN
130 INPUT "DURATION ?":DUR
140 INPUT "FREQUENCY ?":FREQ
150 INPUT "VOLUME ?":VOL :: PRINT : : :
160 IF DUR>17 THEN 180
170 DUR=17
180 REM DURATION
190 DUR=INT((DUR*255)/4250) :: CONV=DUR :: GOSUB 400
200 DUR$=SEG$(HX$,3,2) :: IF FREQ>-1 THEN 290
210 REM NOISE FREQUENCY
220 FR=ABS(FREQ)-1 :: FR$="E"&STR$(FR)
230 REM NOISE VOLUME
240 VOL=INT(VOL/2) :: CONV=VOL
250 GOSUB 430 :: VOL$="F"&SEG$(HX$,4,1)
260 PRINT "DATA>02";FR$;";,>";VOL$;DUR$: : :
270 GOTO 360
280 REM TONE FREQUENCY
290 FR=INT((111860.8/FREQ)+.5)
300 CONV=FR :: GOSUB 400
310 FR$=SEG$(Q$,GN*2+7,1)&SEG$(HX$,4,1)&SEG$(HX$,2,2)
320 REM TONE VOLUME
330 VOL=INT(VOL/2) :: CONV=VOL :: GOSUB 400
340 VOL$=SEG$(Q$,GN*2+8,1)&SEG$(HX$,4,1)
350 PRINT "DATA>03";SEG$(FR$,1,1)&SEG$(FR$,2,1);";,>";
SEG$(FR$,3,2);VOL$;";,>";DUR$;"00": : :
360 PRINT: : "ANOTHER SOUND (Y/N)?"
370 CALL ONKEY("YN",3,K,S) GOTO 100,390
380 GOTO 370
390 CALL CLEAR :: END
400 REM DECIMAL TO HEX
410 AY=INT(CONV)/16 :: BY=INT(AY)/16
420 CY=INT(BY)/16 :: DY=INT(CY)/16
430 AP=(AY-INT(AY))*16 :: BP=(BY-INT(BY))*16
440 CP=(CY-INT(CY))*16 :: DP=(DY-INT(DY))*16
450 HX$=SEG$(Q$,DP+1,1)&SEG$(Q$,CP+1,1)&
SEG$(Q$,BP+1,1)&SEG$(Q$,AP+1,1) :: 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 devices. 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

---

### 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
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, turn off card, turn on card, delay for 2 seconds, turn off card, turn off card. Loop end.	>110 FOR CRU=2048 TO 3968 STE P 128::CALL IO(3,8,CRU,0,3,8 >,CRU,255)::FOR A=1 TO 200::N EXT A::CALL IO(3,8,CRU,0)::N EXT 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, 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.

---

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 INVERSE all character-code (A) in the character definition table in memory.		>100 CALL INVERSE(65)
The program to the right will INVERSE all character-codes from 30 to 143.		>100 CALL INVERSE(ALL)
Line 100 will ask for a string of characters terminated by the ENTER key.		>100 INPUT A\$
Line 110 is a loop to counter.		>110 FOR L=1 TO LEN(A\$)
Line 120 singles each one of the characters in A\$.		>120 C=ASC(SEG\$(A\$,L,1))
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).



---

Format           SAVE DSK2.PRGM,IV254

### Description

The IV254 command functions normally to save XB programs in Internal Variable format of 254 size per record. An additional feature is IV254 may be specified after the SAVE command to convert to Internal Variable 254 format. The IV254 format makes it much more easy to tell an XB program from EA programs when cataloging a disk. Internal Variable files do take up one sector more than XB program format. It should be noted that XB programs smaller than 3 sectors can not be saved in IV254 format.

### Command

Saves to DISK 2 in XB program image format TEST	>SAVE DSK2.TEST
Saves to disk 3 in XB program Internal Variable 254 named STUFF	>sAVE DSK3.STUFF,IV254
Saves to WDS1 in directory EXB XB program Internal Variable 254 named RB	>SAVE WDS1.EXB.RB,IV254

### Options

Allows better cataloging options for saving XB files.

-----  
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  
Some of JOYST runs from ROM.

#### Programs

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

#### Options:

See JOYMOTION, JOYLOCATE, KEY or ONKEY making it much more easy to use then normal XB routines as it combines several commands into a single command to use, also much faster response and more variables are used to control routines for a user.

## Format

```
CALL JOYLOCATE(key-unit,x-return,y-return,
row-index,column-index,#sprite,dot-row,
dot-column)
```

```
CALL JOYLOCATE(key-unit,x-return,y-return,
row-index,column-index,#sprite,dot-row,
dot-column),key-return-variable)
```

```
CALL JOYLOCATE(key-unit,x-return,y-return,
row-index,column-index,#sprite,dot-row,
dot-column),key-return-variable)
GOTO line-number
```

## Description

JOYLOCATE combines commands JOYST, KEY, LOCATE and a built in IF fire-button GOTO line-number. Keyboard key or fire button is in key-return-variable, but only joystick fire or key Q is used for GOTO line-number. As seen above line number option can be left out or further key-return-variable can be left out too. Index is number of dots for row and column.

## Programs

Clear screen.	>100 CALL CLEAR
Set character for use.	>110 CALL CHAR(143,"FFFFFFFFF
	FFFFFF")
Set up a sprite to use.	>120 CALL SPRITE(#1,143,2,9,19
	0)
Look for joystick movement	>130 CALL JOYLOCATE(1,X,Y,8,8,
and move it or ignore.	#1,R,C,K) GOTO 160
Show variables on screen.	>140 PRINT X;Y;K;R;C
Loop forever	>150 GOTO 130
Show variables on screen.	>160 PRINT X;Y;K;R;C;"FIRE"
Loop forever	170 GOTO 130

## Options:

See JOYMOTION or ONKEY or KEY for more XB changes created by RXB to speed up the programs and make them easier to read and write.

---

Format           CALL JOYMAP(key-unit,x-return,y-return,#sprite,  
row-index,column-index,dot-row,dot-col)

                  CALL JOYMAP(key-unit,x-return,y-return,#sprite,  
row-index,column-index,dot-row,dot-col,  
key-return-variable)

                  CALL JOYMAP(key-unit,x-return,y-return,#sprite,  
row-index,column-index,dot-row,dot-col,  
key-return-variable) GOTO line-number

### Description

JOYMAP combines commands JOYST, MOTION, POSITION, KEY and a built in IF fire-button GOTO line-number. Keyboard key Q or fire button is in key-return-variable, but only joystick fire or key Q is used for GOTO line-number. As seen above GOTO line number option can be left out or further the key-return-variable can be left out too. Index is number of dots for row and column. Dot-row and Dot-col are same LOCATE or POSITION.

### Programs

Clear screen.	>100 CALL CLEAR
Set character for use.	>110 CALL CHAR(143,"FFFFFFFFF FFFFFF")
Set up a sprite to use.	>120 CALL SPRITE(#1,143,2,9,19 0,20,0)
Look for joystick movement	>130 CALL JOYMOTION(1,X,Y,#1,9
and move it or ignore.	,9,K) GOTO 160
Show variables on screen.	>140 PRINT X;Y;K
Loop forever	>150 GOTO 130
Show variables on screen.	>160 PRINT X;Y;K;"FIRE"
Loop forever	170 GOTO 130

### Options:

See JOYMOTION or ONKEY or KEY for more XB changes created by RXB to speed up the programs and make them easier to read and write.

## Format

```
CALL JOYMOTION(key-unit,x-return,y-return,
#sprite,row-index,column-index)
```

```
CALL JOYMOTION(key-unit,x-return,y-return,
#sprite,row-index,column-index,
key-return-variable)
```

```
CALL JOYMOTION(key-unit,x-return,y-return,
#sprite,row-index,column-index,
key-return-variable)
GOTO line-number
```

## Description

JOYMOTION combines commands JOYST, KEY, MOTION and a built in IF fire-button GOTO line-number. Keyboard key or fire button is in key-return-variable, but only joystick fire or key Q is used for GOTO line-number. As seen above line number option can be left out or further key-return-variable can be left out too. Index is number of dots for row and column.

## Programs

Clear screen.	>100 CALL CLEAR
Set character for use.	>110 CALL CHAR(143,"FFFFFFFFFF
	FFFFFF")
Set up a sprite to use.	>120 CALL SPRITE(#1,143,2,9,19
	0,20,0)
Look for joystick movement	>130 CALL JOYMOTION(1,X,Y,#1,9
	,9,K) GOTO 160
and move it or ignore.	
Show variables on screen.	>140 PRINT X;Y;K
Loop forever	>150 GOTO 130
Show variables on screen.	>160 PRINT X;Y;K;"FIRE"
Loop forever	170 GOTO 130

## Options:

See JOYMOTION or ONKEY or KEY for more XB changes created by RXB to speed up the programs and make them easier to read and write.

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

## Format

LIST

LIST "device name"

LIST "device name":line length:start line-  
end line

## 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 line outputs to a device.	>LIST "RS232.BA=1200":132:
--------------------------------	----------------------------

This a dummy line.	>100 ! TEST OF LIST
--------------------	---------------------

Another dummy line.	>110 ! TEST OF LIST
---------------------	---------------------

Format           CALL LOAD(address,value[,...])

                  CALL LOAD("access-name"[,...])

#### Description

The LOAD subprogram is used along with INIT, LINK, and PEEK, to access assembly language subprograms. The LOAD subprogram loads an assembly language object file or direct data into the Memory Expansion for later execution using the LINK statement.

The LOAD subprogram can specify one or more files from which to load object data or lists of direct load data, which consists of an address followed by data bytes. The address and data bytes are separated by commas. Direct load data must be separated by file-field, which is a string expression specifying a file from which to load assembly language object code. File-field may be a null string when it is used merely to separate direct load data fields. Use of LOAD subprogram with incorrect values can cause the computer to cease to function and require turning it off and back on.

Assembly language subprogram names (see LINK) are included in the file.

RXB does not check for Memory Expansion if address, values are loaded. EXAMPLE: CALL LOAD(-32000,15) {-32000 = >8300 hex}  
 This was a oversight by original XB teams. This change allows a poke into memory with or without Memory Expansion. If Object Code File is loaded a CALL INIT is still checked.



Format            CALL MAGNIFY(magnification-factor[,...])

### Description

See EXTENDED BASIC MANUAL PAGE 118 for more data. A added feature to MAGNIFY is using a comma more switching of the sprite can be done, like instantly enlarge and reduce a sprite for a shadow effect in XB.

### 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.		20,0,#2,66,2,9,X,30,0,#3,67,
		2,9,X,-20,0)
MAGNIFY enlage and reduce it.		>120 CALL MAGNIFY(1,2,1)
This is a delay loop.		>140 FOR D=1 TO 2000::NEXT D
STOP turns off sprite motion.		>150 GOTO 120
Clear screen and set up the		>100 CALL CLEAR
Loop to create sprites.		>110 FOR L=1 TO 28::CALL SPRI
		TE(#L,L+65,2,L,L,-L,L) ::
		NEXT L
Use MAGNIFY for effects.		>120 CALL MAGNIFY(3,4,3,4)::
		GOTO 120

### Options

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

---

Format            CALL SAMS("MAP"[,...])

### Description

The SAMS MAP 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. SAMS ON turns on read/write mapper registers.

Then a LOAD or SAMS can change the memory pages.

See docs MANUAL-SAMS for examples of memory maps. Also run SAMS-TEST or SAMS-SAVE or SAMS-LOAD programs.

### Programs

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

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

It is recommended that CALL SAMS("MAP") only be used to check SAMS pages with CALL PEEK. CALL SAMS is much more easy to use to manage AMS memory.

-----

FormatMERGE "device.filename"

Description

See EXTENDED BASIC MANUAL PAGE 122 for more data. The only reason for this page in RXB is a problem with SIZE and the MERGE command breaks SIZE from working as they both use the same address to record XB RAM END ADDRESS. This problem will only happen if you use PRAM to change program normal start and end locations of XB RAM. Please never use the merge command if you have changed XB RAM with PRAM command.

Command

Change locations to start XB		>CALL PRAM(-12288,-16384)
to >C000 and end to >D000		
This will load a program.		>OLD DSK1.TEST
This will merge both programs.		>MERGE DSK1.TEST2
SIZE will report wrong program		>SIZE
space incorrectly		

---

Format            CALL MOD(number,divisor,quotiant,remainder  
                  [,...])

### Description

The MOD command will make a MODULO FACTOR of a number and divisor to produce a quotient and remainder. MOD command will only factor numbers from -32678 to 32767 larger values will be clipped by the internal integer format. Also if the number is 0 or divisor is 0 a error of bad value will result as you can not divide 0 by anything or anything by 0.

### Programs

Number=10 and Divisor=3		>100 N=10 :: D=3
Do MOD on values with results		>110 CALL MOD(N,D,Q,R)
Print Q and R values on screen		>120 PRINT Q,R
N=number,D=divisor,Q=Quotiant		
and R=remainder		
Divide 32767/3		>100 CALL MOD(32767,3,Q,R)
Show results		>110 PRINT Q,R
Q=10922 and R=1		
Divide -32768/3		>100 CALL MOD(-32768,3,Q,R)
Show results		>110 PRINT Q,R
Q=10922 and R=2		

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. MOTION runs from ROM.

### 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 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

Move lines 100 thru 180 to line 1000, increment by 5.	>MOVE 100-180,1000,5
Moves lines 40 thru last line to line 120, increment by 10.	>MOVE 40-,120,
Moves line 150 to line 110	>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 TO 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 see the Extended Basic Manual page 41. MOVES will error out with \* BAD VALUE IN ####\* 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.

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

RAM may be moved but not into 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

### Programs

Line 100 has the type\$ string.	>100 X\$="VV"
Line 110 thus uses type\$ 0 VDP 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.	>110 CALL MOVES(X\$,767,1,0)
Line 100 copies entire screen into lower 8K.	>100 CALL MOVES("VR",768,0,8192)
Line 110 clears the screen.	>110 CALL CLEAR
Line 120 copies entire screen into lower 8K.	>120 CALL MOVES("VR",768,0,9000)
Line 130 copies from lower 8K to screen, then again. GOTO makes it an endless loop.	>130 CALL MOVES("RV",768,8192,0,"RV",768,9000,0) :: GOTO 130
Line 100 sets up loop. Counts from -32768 to 0 to 32767 or (HEX >8000 to >0000 to >7FFF)	>100 FOR G=-32768 TO 32767
Line 110 move GRAM/GROM to VDP. 8 bytes to be moved. GA is counter. 1024 is decimal address of space character in VDP pattern table.	>110 CALL MOVES("GV",8,G,1024)
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



---

### Programs

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

---

Format            CALL SAMS("OFF")

### Description

SAMS("OFF") command will only work with a SAMS 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 SAMS("OFF") command. They can't be read/written to. See docs MANUAL-AMS for examples of memory maps.

Also run SAMS-TEST or SAMS-SAVE or SAMS-LOAD programs.

### Programs

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

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

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

---

Format            CALL SAMS("ON")

### Description

SAMS("ON") command will only work with a SAMS memory card. The read/write lines to the SAMS 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-SAMS for examples of memory maps. Also run SAMS-TEST or SAMS-SAVE or SAMS-LOAD programs.

### Programs

This turns on read/write.	>100 CALL SAMS("ON")
This loads 9 in map register 2	>110 CALL LOAD(16388,9)
This turns off read/write.	>120 CALL SAMS("OFF")
This loads values in lower 8K.	>130 CALL LOAD(8192,1,2,3,4)
This turns on pass mode.	>140 CALL SAMS("PASS")
This peeks values in lower 8K.	>150 CALL PEEK(8192,A,B,C,D)
This prints values.	>160 PRINT A;B;C;D
This turns ON read/write & MAP	>170 CALL SAMS("ON","MAP")
This loads 2 in map register 2	>180 CALL LOAD(16388,2)
Turns off read/write & PASS	>190 CALL SAMS("OFF","PASS")
This peeks values in low page.	>200 CALL PEEK(8192,A,B,C,D)
This prints values.	>210 PRINT A;B;C;D

It is recommended to use CALL SAMS("ON") only for when a CALL PEEK is used to check a mapper register value. CALL SAMS manages AMS mapping much better.

---

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

                     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 an 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("Test",0,K,S) GOTO 22,29,34,41 :: GOTO 10
```

If T is pressed then 22 is used.

If e is pressed then 29 is used.

If s is pressed then 34 is used.

If t is pressed then 41 is used.

If no key pressed GOTO 10 to repeat line.

---

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 SAMS("PASS")

### Description

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

### Programs

This turns on read/write.		>110 CALL SAMS("ON")
Load 37 into map register 2.		>120 CALL LOAD(16388,37)
This turns off read/write.		>130 CALL SAMS("OFF")
This turns on pass mode.		>140 CALL SAMS("PASS")
This shows multiple commands		>100 CALL SAMS("ON","PASS")

SAMS("PASS") is mainly used to turn off SAMS or protect the SAMS pages from being used or to behaves like a normal 32K when the SAMS is not being used.

---

Format           CALL PATTERN(#sprite-number,character-value  
                 [,...])

#### Description

See EXTENDED BASIC MANUAL page 142 for more data.  
Now 30 (CURSOR) and 31 (EDGE CHARACTER) and 144 to 159 may  
used if only the top highest sprite numbers are used. For  
example you can not use sprite #1 if you are using characters  
143 to 146 to define a sprite pattern, but you could use  
sprite #28 instead with no issues. Thus some care must be  
taken to use all characters from 144 to 159 when using sprites.  
But the advantage is now you can use 30 to 159 in RXB.  
PATTERN runs from ROM.

CALL PATTERN just allows Sprite patterns not characters.

#### Options

Sprites may not be used if characters 144 to 159 are being  
redefined for use.



-----

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.	>100 CALL PEEKG(767,B)
Address loop counter	>100 FOR D=-32768 TO 32767
PEEK Grom address value.	>110 CALL PEEK(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 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 read a byte from VDP and put it into variable B.	>100 CALL PEEKV(767,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 PLOAD(memory-boundry,"access-name")

CALL PLOAD(contant,string-variable)

### Description

The PLOAD subprogram loads ONLY program image files created by PSAVE. PLOAD is the opposite of PSAVE. PLOAD is a faster version of CALL LOAD. PLOAD has the speed of a hidden loader and is much easier to use. PLOAD loads any 4K boundry in 32K.

Memory boundries are 2, 3, A, B, C, D, E, F (upper case). i.e. 2 is >2000 or 3 is >3000 or A is >A000 up to F is >F000 Removing the zeros made more sense then adding 3 zeros.

Unlike CALL LOAD the PLOAD and PSAVE 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: 4K of VDP memory MUST be free for PLOAD to function or a memory full error will result. Always place the PLOAD command at the top of the RXB program.

### Programs

This line loads a previously saved programs image files.	>100 CALL PLOAD(2,"DSK2.MOUSE",3,"DSK2.MOUSE2")
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 PLOAD(B,"DSK1.DUMP")
This line turns on interrupt within program.	>110 CALL ISRON(16384)
This line links to support address DUMPIT routine.	>120 CALL LINK("DUMPIT") ! link to Program Support

-----

PLOAD is faster then CALL LOAD as it loads Program Image vs LOAD which is stuck with slow uncompressed DF 80 files.

#### Options

SAMS users will find this a easy way to load RXB AMS support into lower 8K.

#### EXAMPLE:

```
>100 Z$="DSK1.PAGE"  
>110 FOR L=0 TO 15 STEP 2  
>120 CALL SAMS(2,L,3,L+1)  
>130 CALL PLOAD(2,Z$&STR$(L),3,Z$&STR(L+1))  
>140 NEXT L  
>150 CALL XB("DSK1.MAINPROGRAM",1)
```

The above program would load RXB SAMS pages 0 to 15 into SAMS memory from files named PAGE0 to PAGE15 on disk 1. Then would set CALL FILES 1 and RUN "DSK1.MAINPROGRAM" with 64K of Assembly support for RXB. (16x4K=64K)

See SAMS, ISROFF, ISRON, EXECUTE, and MOVES.

-----

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 63 are valid. The byte value ranges 0 to 255. The number of Registers were increased to 63 VDP Registers for use with F18 register set.

### 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)
This resets memory.	>170 CALL NEW

-----

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 PRAM(start-RAM-address,end-RAM-address)

### Description

The PRAM command changes the location of the Start and End of XB RAM program space. Normally XB RAM is start address is >FFE7 and end address is >A040 in hex so the PRAM command allows changing this location to as low as 1 byte of XB RAM PROGRAM SPACE.

Any location from >A000 to >FFFF is a valid change in PRAM.

This command has no effect on Lower 8K Assembly RAM.

Use of PRAM is for control of XB RAM space and XB programs can reside anywhere in the upper 24K RAM locations. Combined with PSAVE and PLOAD assembly can be utilized in upper 24K.

### Programs

This line is comment.	>100 ! CALL PRAM(start-address
	s,end-address) 12K size
Clear screen.	>110 CALL CLEAR
Show size, delay, clear screen	>120 SIZE::CALL KEY("",5,K,S)
Display it.	>130 PRINT "CALL PRAM(-25,-24
	576)": ">E000->B000 =12K RAM"
Change locations to start XB	>140 CALL KEY("",5,K,S)::CALL
to >E000 and end XB to >B000	PRAM(-8192,-20480)
This defaults to what ever the	>CALL PRAM(0,0)
previous values were same as	>SIZE
nothing was called	
Change locations to start XB	>CALL PRAM(-12288,-16384)
to >C000 and end to >D000	>SIZE
Change locations to start XB	>CALL PRAM(-8192,-12288)
to >E000 and end XB to >E000	>SIZE



-----

Format           CALL PSAVE(memory-boundry,"access-name")

                  CALL PSAVE(constant,string-variable)

### Description

The PSAVE subprogram saves ONLY program image files to be used for PLOAD. PSAVE is the opposite of PLOAD. PSAVE has the speed of a hidden loader without the hassle.

PLOAD saves any 4K boundry from 32K.

Memory boundries are 2, 3, A, B, C, D, E, F (upper case). i.e. 2 is >2000 or 3 is >3000 or A is >A000 up to F is >F000 Removing the zeros made more sense then adding 3 zeros.

Unlike CALL LOAD the PLOAD and PSAVE 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 PSAVE(2,"DSK#.NAME1",3,"DSK#.NAME2") ! 2 4K of lower 8K Remember to check for interrupts or the program will not work after loading with PLOAD. See ISRON and ISROFF.

NOTE: 4K of VDP memory MUST be free for PSAVE to function or a memory full error will result. Always place the PSAVE command at the top of the RXB program.

### Programs

Initialize lower 8K.	>100 CALL INIT
Load the assembly support.	>110 CALL LOAD("DSK1.MSETUP0")
Load the assembly support.	>120 CALL LOAD("DSK1.HDSR")
Turn on the mouse setup.	>130 CALL LINK("MSETUP")
BSAVE 2 of 4K sections of lower 8K.	>140 CALL PSAVE(2,"DSK2.MOUSE1",3,"DSK2.MOUSE2")

---

Procedure for hidden loaders.	
Loads program on disk 1	>CALL XB("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 programs to disk.	>CALL PSAVE(2,"DSK2.EXAMPLE1",
	3,"DSK2.EXAMPLE2")

#### Options

SAMS users will find this a easy way to save RXB SAMS support  
EXAMPLE:

```
>100 Z$="DSK1.PAGE"  
>110 FOR L=15 TO 32 STEP 2  
>120 CALL SAMS(2,L,3,L+1)  
>130 CALL PSAVE(2,Z$&STR$(L),3,Z$&STR$(L+1))  
>140 NEXT L
```

The above program would save RXB SAMS pages 16 to 33 into  
8 program image files named PAGE15 to PAGE33 on disk 1.

See SAMS, ISROFF, ISRON, EXECUTE, and MOVES.

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                RANDOMIZE

                      RANDOMIZE SEED

### Description

The RANDOMIZE command can be found on XB manual page 151 to help explain it's use. RXB unlike any other XB produced has a feature that makes RND and RANDOMZE different and better. RANDOMIZE SEED in RXB is same as TI BASIC randomize seed. Thus in RXB do not expect the same random numbers as you would get with any other XB made. RXB is way more random due to this change different then any other Extended Basic.

### Program

Will put hex >3567 into seed		>100 RANDOMIZE
RND example to prove speed		>110 DIM N(100)
Counter in a FOR loop		>120 FOR X=1 TO 100
Load Array with random numbers		>130 N(X)=RND
Show that number		>140 PRINT N(X)
Repeat loop till done		>150 NEXT X

Run this above example in TI BASIC, XB and RXB 2020 to show game type results of program results with new RND

### Options

Random Music programs will sound very very fast due to the speed increase in RXB RND is much faster.

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

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. RMOTION runs from ROM.

### 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

RND

## Description

The RND subprogram in RXB has been replaced with a TI BASIC version as the normal XB RND subprogram is hindered with so much Floating Point as to make it 3 times slower than the TI BASIC version of RND. Extensive testing proves that the new RXB RND is many times faster than the previous version.

There will actually be some programs expecting a particular RND pattern of random numbers that will no longer work the same as a result of this change. But games will appear more random than normal Extended Basic.

The RANDOMIZE seed still works but the results of the that pattern of random numbers will be different than normal XB, thus unless absolutely required will be a bigger benefit than the cost of this XB previous feature.

## Program

RND example to prove speed		>100 DIM N(100)
Counter in a FOR loop		>110 FOR X=1 TO 100
Load Array with random numbers		>120 N(X)=RND
Show that number		>130 PRINT N(X)
Repeat loop till done		>140 NEXT X

Run this above example in TI BASIC, XB and RXB 2015 to show game type results of program results with new RND

## Options

Random Music programs will sound very very fast.



Format

CALL ROLLDOWN

CALL ROLLDOWN(repetition)

### Description

ROLLDOWN scrolls screen to the down so repetition will repeat the scroll number of times to down.  
 Repetition can be 1 to 65535 max. ROLLDOWN runs from ROM.

### Programs

Roll screen down 2 times	>CALL ROLDOWN(2)
Prints line	>100 PRINT "SCREEN PRINT"
Roll screen down	>110 CALL ROLLDOWN
Repeat the program	>100 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL DOWN"
Print X\$	>110 PRINT X\$
Roll down 9 times use X\$	>120 CALL ROLLDOWN(9)
Repeat the program	>130 GOTO 100

### Options

New features allow for some special that can take the place of some routines that are slower in XB.

Format

CALL ROLLLEFT

CALL ROLLLEFT(repetition)

### Description

ROLLLEFT scrolls screen to the left so repetition will repeat the scroll number of times to left.  
 Repetition can be 1 to 65535 max. ROLLLEFT runs from ROM.

### Programs

Roll screen left 2 times	>CALL ROLLLEFT(2)
Prints line	>100 PRINT "SCREEN PRINT"
Roll screen left	>110 CALL ROLLLEFT
Repeat the program	>120 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL LEFT"
Print X\$	>110 PRINT X\$
Roll left 9 times use X\$	>120 CALL ROLLLEFT(9)
Repeat the program	>130 GOTO 100

### Options

New features allow for some special that can take the place of some routines that are slower in XB.

-----

Format            CALL ROLLRIGHT

                  CALL ROLLRIGHT(repetition)

### Description

ROLLRIGHT scrolls screen to the right so repetition will repeat the scroll number of times to right.  
Repetition can be 1 to 65535 max. ROLLRIGHT runs from ROM.

### Programs

Roll screen right 2 times	>CALL ROLLRIGHT(2)
Prints line	>100 PRINT "SCREEN PRINT"
Roll screen right	>110 CALL ROLLRIGHT
Repeat the program	>120 GOTO 110
Load X\$ string variable	>100 X\$=" ROLL RIGHT"
Print X\$	>110 PRINT X\$
Scroll right 9 times use X\$	>120 CALL ROLLRIGHT(9)
Repeat the program	>130 GOTO 100

### Options

New features allow for some special that can take the place of some routines that are slower in XB.

Format

CALL ROLLUP

CALL ROLLUP(repetition)

### Description

ROLLUP scrolls screen to the up so repetition will repeat the scroll number of times to up.  
 Repetition can be 1 to 65535 max. ROLLUP runs from ROM.

### Programs

Roll screen up 2 times	>CALL ROLLUP(2)
Prints line	>100 PRINT "SCREEN PRINT"
Roll screen UP	>110 CALL ROLLUP
Repeat the program	>120 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL UP"
Print X\$	>110 PRINT X\$
Roll up 9 times use X\$	>120 CALL ROLLUP(9)
Repeat the program	>130 GOTO 100

### Options

New features allow for some special that can take the place of some routines that are slower in XB.

Format            CALL SAMS(address-boundry,page-number[,...])

                  CALL SAMS(address-boundry,numeric-variable  
                  [,...])

                  CALL SAMS(command [,...])

## Description

The SAMS command will only work with a SAMS memory card.

The address-boundry is a value in Hexadecimal denoted by

2 is >2000 or 3 is >3000 or A is >A000 or D is >D000

EXAMPLE: CALL SAMS(3,page-number[,...])

This 3 stands for >3000 hexidecimal address boundry.

CALL SAMS uses boundry symbols upper case only.

i.e. 2 = >2000, 3 = >3000, A = >A000, B = >B000, C = >C000,

D = >D000, E = >E000 and F = >F000

SAMS turns on the read/write lines of SAMS mapper registers  
stores the value into the mapper register chosen. Less wasted  
pages results in more memory available. Page numbers can be  
from 0 to 16383 so it is hard to explain this easy.

See 16383 would be >FFFF hexidecimal 64 Meg SAMS. Pages 0 to  
255 would be a 1 Meg SAMS, Pages 256 to 511 would be a 2 Meg  
SAMS, so on up to page 7935 to 8191 would be 32 Meg SAMS.

Pages 8192 to 16383 would be above 32K Meg SAMS so RXB 2020  
handles 64 Meg SAMS, but not tested above 32 Meg yet.

(\*Note: 16384 to 32767 would be for above 32 Meg to 64 Meg.)

A additional new feature in 2020 RXB SAMS is use of upper 24K  
memory can now be switched, but of course care must be taken  
or will crash XB by removing the program running SAMS from  
upper 24K. Imagine 8 Meg XB program swapping lines.

The order of changing 4K pages does not matter thus a  
CALL SAMS(A,55,3,34) example is put 4K page 55 SAMS Memory  
at >A000 and 4K page 34 at >3000

Original SAMS commands like ON, OFF, MAP or PASS still work.

"ON" turns on Mapper Registers.

"OFF" turns off Mapper Registers.

"MAP" turns on Map Mode so pages can be changed.

"PASS" default mode making the SAMS just like a normal 32K.

-----

Example is mixing commands:

```
100 CALL SAMS("ON","MAP",2,237,"OFF")
```

This turns on SAMS read/write Registers, turns on MAP mode, sets 4K page with page 237 than turns off SAMS read/write Registers.

### Programs

This turns on the SAMS mapper.		>110 CALL SAMS("ON")
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 4K pages		>160 CALL SAMS(2,16,3,17)
This loads a assembly program.		>170 CALL LOAD("DSK1.DUMP")
This changes low/high back.		>180 CALL SAMS(2,L,3,H)
This uses a routine in CHAR.		>190 CALL LINK("CHAR")
This changes low/high again.		>200 CALL SAMS(2,16,3,17)
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 SAMS was used. 1024K would be 120 assembly support programs. Compatibility of most software assured in RXB AMS support.

### Options:

See ON, OFF, MAP and PASS pages in RXB Documents for more information on SAMS.

## SAMS MAPPER

\*\*\*\*\*

The SAMS 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)

## SAMS MAPPER

\*\*\*\*\*

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	= >62	= 98	points to >2000 - >2FFF range
>4006	= 16390	is MR03	= >63	= 99	points to >3000 - >3FFF range
>4014	= 16404	is MR10	= >64	= 100	points to >A000 - >AFFF range
>4016	= 16406	is MR11	= >65	= 101	points to >B000 - >BFFF range
>4018	= 16408	is MR12	= >66	= 102	points to >C000 - >CFFF range
>401A	= 16410	is MR13	= >67	= 103	points to >D000 - >DFFF range
>401C	= 16412	is MR14	= >68	= 104	points to >E000 - >EFFF range
>401E	= 16414	is MR15	= >69	= 105	points to >F000 - >FFFF range

(MR=Mapper Register)

In MAP mode the mapper register setup is equivalent to: EXAMPLE3

mapper address		mapper	page num		address range
HEX	Dec		HEX	Dec	memory area
>4004	=16388	is MR02	=>1FF9	= 8185	points to >2000 - >2FFF range
>4006	=16390	is MR03	=>1FFA	= 8186	points to >3000 - >3FFF range
>4014	=16404	is MR10	=>1FFB	= 8187	points to >A000 - >AFFF range
>4016	=16406	is MR11	=>1FFC	= 8188	points to >B000 - >BFFF range
>4018	=16408	is MR12	=>1FFD	= 8189	points to >C000 - >CFFF range
>401A	=16410	is MR13	=>1FFE	= 8190	points to >D000 - >DFFF range
>401C	=16412	is MR14	=>1FFF	= 8191	points to >E000 - >EFFF range
>401E	=16414	is MR15	=>2000	= 8192	points to >F000 - >FFFF range

(MR=Mapper Register)



-----

Format

SAVE DSK3.PRGM

SAVE DSK2.PRGM,IV254

### Description

The SAVE command functions normally to save XB programs. An additional freature is IV254 may be specified after the SAVE command to convert to Internal Variable 254 format. The IV254 format makes it much more easy to tell an XB program from EA programs when cataloging a disk. Internal Variable files do take up one sector more then XB program format. It should be noted that XB programs smaller then 3 sectors can not be saves in IV254 format.

### Command

Saves to DISK 2 in XB program image format TEST	>SAVE DSK2.TEST
Saves to disk 3 in XB program Internal Variable 254 named STUFF	>sAVE DSK3.STUFF,IV254
Saves to WDS1 in dirctory EXB XB program Internal Variable 254 named RB	>SAVE WDS1.EXB.RB,IV254

### Options

Allows better cataloging options for saving XB files.

Format

```
CALL SCREEN(color-code[,...])

CALL SCREEN("OFF"[,...])

CALL SCREEN("ON"[,...])
```

### Description

See EXTENDED BASIC MANUAL PAGE 165 for more data.  
 RXB has added features of OFF and ON to the SCREEN command. OFF turns off the screen display while the ON turn the screen back on. Use of OFF command allows for writing to screen happens but not visible to user.

### Programs

Turn screen to white	>100 CALL SCREEN(16)
Turn off the screen display	>100 CALL SCREEN("OFF")
Prints line but screen off	>110 PRINT "THE SCREEN IS OFF"
Waits for any key	>120 CALL KEY("",5,K,S)
This opens a RS232 port.	>130 CALL SCREEN("ON")
Prints line but screen on	>140 PRINT "NOW SCREEN ON"
Waits for any key	>150 CALL KEY("",5,K,S)
Special effect use of SCREEN	>160 CALL SCREEN(0,2,0,2,0,2)

### Options

New features allow for some special effects like draw screen while screen is off and then pop it to user. Or use of the comma to switch colors making some special effects.

---

Format            CALL SCROLLDOWN

                  CALL SCROLLDOWN(repetition)

                  CALL SCROLLDOWN(repetition,string)

                  CALL SCROLLDOWN(repetition,string,tab)

### Description

SCROLLDOWN scrolls screen to the down so repetition will repeat the scroll number of times down, the string will only display horizontally 32 characters of the string on right side of screen. SCROLLDOWN puts the string on screen and wraps to bottom if string is too long.

If the string is empty (null) it will just scroll the screen. Like PRINT TAB will go to next line right of left side of screen each line till end of string. Numbers or variables can be used instead of a string. SCROLLDOWN runs from ROM.

Repetition can be 1 to 65535 max.

### Programs

Scroll down 2 times print PI	>CALL SCROLLDOWN(2,PI)
Clear screen for demo	>100 CALL CLEAR
Prints line	>110 PRINT "SCREEN PRINT"
Scroll screen down	>120 CALL SCROLLDOWN
Repeat the program	>130 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL DOWN"
Print X\$	>110 PRINT X\$
Scroll down 9 times use X\$	>120 CALL SCROLLDOWN(9,X\$)
Repeat the program	>130 GOTO 100

Format

CALL SCROLLLEFT

CALL SCROLLLEFT(repetition)

CALL SCROLLLEFT(repetition,string)

CALL SCROLLLEFT(repetition,string,tab)

### Description

SCROLLLEFT scrolls screen to the left so repetition will repeat the scroll number of times left, the string will only display vertically 24 characters of the string on left side of screen. SCROLLLEFT unlike SCROLLRIGHT puts the string on screen and wraps to other side if string is to long. If the string is empty (null) it will just scroll the screen each line till end of string. Numbers or variables can be used instead of a string. SCROLLLEFT runs from ROM. Repetition can be 1 to 65535 max.

### Programs

Scroll left 2 times print PI	>CALL SCROLLLEFT(2,PI)
Clear screen for demo	>100 CALL CLEAR
Prints line	>110 PRINT "SCREEN PRINT"
Scroll screen left	>120 CALL SCROLLLEFT
Repeat the program	>130 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL LEFT"
Print X\$	>110 PRINT X\$
Scroll left 9 spaces use X\$	>120 CALL SCROLLLEFT(9,X\$)
Repeat the program	>130 GOTO 100

---

Format            CALL SCROLLRIGHT

                  CALL SCROLLRIGHT(repetition)

                  CALLL SCROLLRIGHT(repetition,string)

                  CALL SCROLLRIGHT(repetition,string,tab)

### Description

SCROLLRIGHT scrolls screen to the right so repetition will repeat the scroll number of times right, the string will only display vertically 24 characters of the string. SCROLLRIGHT unlike SCROLLLEFT puts the string on screen and does not wrap to other side if string is too long. If the string is empty (null) it will just scroll the screen each line till end of string. Numbers or variables can be used instead of a string. SCROLLRIGHT runs from ROM. Repetition can be 1 to 65535 max.

### Programs

Scrollright 2 times print PI	>CALL SCROLLRIGHT(2,PI)
Clear screen for demo	>100 CALL CLEAR
Prints line but screen off	>110 PRINT "SCREEN PRINT"
Scroll screen right	>120 CALL SCROLLRIGHT
Repeat the program	>130 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL RIGHT"
Print X\$	>110 PRINT X\$
Scroll right 9 spaces use X\$	>120 CALL SCROLLRIGHT(9,X\$)
Repeat the program	>130 GOTO 100

---

Format

CALL SCROLLUP

CALL SCROLLUP(repetition)

CALL SCROLLUP(repetition,string)

CALL SCROLLUP(repetition,string,tab)

### Description

SCROLLUP scrolls screen up so repetition will repeat the scroll number of times to up, the string will only display horizontally 32 characters of the string. SCROLLUP puts the string on screen and wraps to top if string is too long. If the string is empty (null) it will just scroll the screen each line till end of string. Numbers or variables can be used instead of a string. SCROLLUP runs from ROM. Repetition can be 1 to 65535 max.

### Programs

Scroll up 2 times print PI	>CALL SCROLLUP(2,PI)
Clear screen for demo	>100 CALL CLEAR
Prints line but screen off	>110 PRINT "SCREEN PRINT"
Scroll screen UP	>120 CALL SCROLLUP
Repeat the program	>130 GOTO 110
Load X\$ string variable	>100 X\$=" SCROLL UP"
Print X\$	>110 PRINT X\$
Scroll up 9 spaces use X\$	>120 CALL SCROLLUP(9,X\$)
Repeat the program	>130 GOTO 100

Format	SIZE
	CALL SIZE

### Description

See EXTENDED BASIC MANUAL PAGE 169 for more data.  
 RXB has added many more features to SIZE. RXB shows the size and memory address of VDP, RAM and SAMS. Very useful for XB or Assembly programmers. EXAMPLE:

```
>SIZE
11840 Bytes of Stack Free
24488 Bytes of Program
8192 Bytes of Assembly
* PAGE NUMBER = LOCATION *
2    Page = >2000 - >2FFF
3    Page = >3000 - >3FFF
10   Page = >A000 - >AFFF

11   Page = >B000 - >BFFF
12   Page = >C000 - >CFFF
13   Page = >D000 - >DFFF
14   Page = >E000 - >EFFF
15   Page = >F000 - >FFFF
* MEMORY UNUSED and FREE *
>37D7 VDP Free Address
>0958 VDP STACK Address
>FFE7 Program Free Address
>A040 Program End Address
>2000 RAM Free Address
>4000 RAM End Address
```

This shows normal XB values but also includes more useful things like Assembly free and SAMS pages used and where these pages are. Lastly it shows VDP STACK location, First free VDP address, XB RAM First free address and End address. Lastly first free Assembly address and End address used. SAMS size is not reported just like Foppy size or hard drive is'nt!

-----

Format

SIZE

CALL SIZE

Command

May only be used from command mode.

>SIZE

Programs

May only be used from program mode.

>100 CALL SIZE

Delay for keypress.

>110 CALL KEY("",0,K,S)

Set up for Assembly support.

>120 CALL INIT

Shows memory used including Assembly space free.

>130 CALL SIZE

Set VDP STACK to >1820 hex.

>140 CALL VDPSTACK(6176)

Show VDP STACK location.

>150 CALL SIZE

Delay for keypress.

>150 CALL KEY("",0,S,S)

Set XB RAM to >A000 hex.

>160 CALL PRAM(-24576)

Shows 64 more bytes of XB RAM for use in XB.

>170 CALL SIZE