XB256

Expanded Graphics for TI Extended BASIC

by Harry Wilhelm

TABLE OF CONTENTS

INTRODUCTION - - - - - - - 2
Equipment required - - - - - - 2
Differences from Extended BASIC - - - 2
Loading XB256 - - - - - - - 3
USING THE SUBROUTINES - - - - - - 3
COMMANDS- - - - - - - - - 4
Turn off XB256 (OFF) - - - - - - 4
Turn on XB256 (XB256) - - - - - - 4
SELECTING THE SCREEN - - - - - - - 4
Screen1 (SCRN1) - - - - - - - 4
Screen2 (SCRN2) - - - - - - - 4
COLOR AND CHARACTER PATTERNS IN SCREEN 2 - - 4
Define colors (COLOR2) - - - - - - 4
Define characters (CHAR2) - - - - - - 5
Get character pattern (CHPAT2) - - - - 5
Load character set (CHSET2) - - - - 5
Large character set (CHSETL) - - - - 5
SUBROUTINES FOR SCROLLING - - - - - 5
Define window boundaries (WINDOW) - - - 5
Scroll up (SCRLUP) - - - - - - - 6
Scroll down (SCRLDN) - - - - - - 6
Scroll right (SCRLRT) - - - - - - 6
Scroll left (SCRLLF) - - - - - - - 6
Scroll pixel up (SCPXUP) - - - - - 6
Scroll pixel right (SCPXRT) - - - - - 7
Scroll pixel left (SCPXLF) - - - - - 7
MISCELLANEOUS SUBROUTINES - - - - - 8
Highlight text (HILITE) - - - - - - 8
Set early clock (EARLY) - - - - - - 8
Display in 32 columns (DISPLY) - - - - 8
Read from VDP RAM (VREAD) - - - - 9
Write to VDP RAM (VWRITE) - - - - 9
APPENDICES - - - - - - - - 10
Character Sets Used By CALL LINK(“COLOR2”) - - 10
116K VDP RAM MEMORY MAP WITH XB256 - - 10
Saving Stack Space - - - - - - - 11

INTRODUCTION

XB256 is a collection of assembly language subroutines that give the Extended BASIC programmer much more access to the graphics capabilities built into the TMS 9918A video display processor. No knowledge of assembly language is required to use XB256. Programs are written completely in Extended BASIC, meaning they are both easy to write and easy to understand. The average TI user can now take advantage of graphics capabilities that were built into the computer, but never before available without having to program in assembly language.

XB256 lets you select from two independent screens. Screen1 is the screen normally used by Extended BASIC and is accessed exactly as described in the XB manual. Screen2 lets you define 256 characters, compared to the 112 normally available to XB. Additionally, you can use up to 28 double sized sprites using the patterns available to Screen1. You can toggle between the two screens as desired and the graphics on each screen will be preserved. When using Screen2 there are assembly equivalents that replace CHAR, CHARPAT, COLOR, and CHARSET. Except for these subroutines, screen access in Screen2 is by the usual Extended BASIC statements such as PRINT, SPRITE, ACCEPT, etc.

There are scrolling routines that allow you to scroll screen characters left, right, up, or down. Other routines let you scroll smoothly one pixel at a time to the left, right or vertically. You can specify a window area for scrolling and leave the rest of the screen unchanged.

There are some miscellaneous subroutines that let you hilight text, set the sprite early clock, print on the screen using all 32 columns, and read from or write to the VDP RAM.

EQUIPMENT REQUIRED

XB256 has been tested with Classic 99 and Win994a. If you use vintage hardware, it requires the TI-99/4A console, the Extended BASIC cartridge, the 32K memory expansion, and a disk drive system.

DIFFERENCES FROM EXTENDED BASIC

When a program starts running in XB256 it defaults to Screen1. The program can then select Screen1 or Screen2 as desired. The old screen is saved and the new screen takes its place. When you break a program by pressing <Fctn 4> the display will automatically revert to Screen1. After you break a program, you can type CON to continue. If you were using Screen 2, XB256 will restore the screen just as it was when the program was interrupted, although sprites are not restored.

The memory available for programming is 24488 bytes which is the normal size for XB. The stack space is reduced from 11840 bytes to 8808 bytes. The stack space is primarily used to contain string data and subprogram names. There should be enough room for most programming needs. You may have to adjust your programming style in order to conserve stack space if your program is very long or uses a lot of strings. There is a section in the appendices that describes how to save stack space.

CALL FILES(2) is automatically performed when XB256 starts. Do not use CALL FILES because nothing useful will happen and it will reset pointers needed by XB256.

The "quit" key has been disabled. Instead, type "BYE" to return to the master title screen.

LOADING “XB256”

Select Extended BASIC from the TI master title screen. Place the program disk into a disk drive (drive number "n"), type RUN "DSKn.XB256" and press <Enter>. XB256 loads into low memory and then the loader is erased. XB256 can be renamed LOAD for autoloading from drive 1. If you want to load XB256 and then RUN a program from the loader, add “RUN “DSKn.PROGNAME to line 10 of the loader.

USING THE SUBROUTINES

XB256 contains 22 assembly language subroutines, which can be grouped in the following categories:

- COMMANDS
- CHANGING THE SCREEN
- CHANGING COLORS AND CHARACTER DEFININTIONS IN SCREEN 2
- SUBROUTINES FOR SCROLLING
- MISCELLANEOUS SUBROUTINES

Except for the commands XB256 and OFF, all of these subroutines should be called from within a running Extended BASIC program. No error message results when the subroutines are called from the immediate mode, but nothing useful will result. Except as noted, they can be used in either Screen1 or Screen2.

The subroutines are described in the next sections. The first line of each description shows the correct syntax to use when calling the subroutine. Most of the subroutines require that additional information be included after the name of the subroutine. This information is supplied in the form of a parameter list. Be careful to include these parameters in the order described, and not to mix strings and numbers. Sometimes there are optional parameters. These optional parameters are shown enclosed in brackets. The purpose of each of the parameters in the list is fully described. Numbers and strings can be constants, variables, or elements of an array.

COMMANDS

CALL LINK("OFF")

Turns off the interrupt routine for XB256. This turns off the interrupt routine without having to return to the master title screen. (Immediate mode)

CALL LINK("XB256")

Turns the interrupt routine back on so XB256 is active. The loader does this when loading XB256, so you would only use this if you turned off XB256 with CALL LINK("OFF") as described above. (Immediate mode)

SELECTING THE SCREEN

CALL LINK("SCRN1")

Selects Screen1, which is the normal XB screen. This has no effect if you are already in Screen1.

CALL LINK("SCRN2")

Selects Screen2, which is the enhanced graphics screen. This has no effect if you are already in Screen2. When a program first RUNs, Screen2 is cleared, the character set for Screen2 is loaded and the colors are set to white on dark blue.

COLOR AND CHARACTER PATTERNS IN SCREEN 2

There are five assembly language subroutines used to modify the colors and patterns defining the characters in Screen 2. These only effect graphics in Screen2, but can be called from Screen1 if you want to predefine colors, characters, etc. before changing to Screen2. Likewise, when in Screen2 you can call the XB subprograms COLOR, CHAR, etc. if you want to predefine these before changing to Screen1. One exception is that you should not CALL CHARSET while in Screen2. This will confuse XB256 into saving Screen2 and restoring Screen1 because it thinks the program is breaking.

CALL LINK("COLOR2",character-set,foreground-color,background-color [,...])

This is the equivalent of CALL COLOR in XB. This will change the color of the characters used by Screen2. *Foreground-color* and *background-color* must be from 1 to

16. *Character-set* is a number from 0 to 31. If *character-set* is less then 0 or greater than 31 then 32 is added or subtracted as needed to make it 0 to 31. If *character set* is 81 (think 9*9) then all 32 character sets are changed to the specified colors. Up to five character sets can be changed with one call to COLOR2.

CALL LINK("CHAR2",character-code,pattern-identifier[,...])

This is the equivalent of CALL CHAR in XB. It is used to change the patterns of the characters used by Screen2. *Character-code* specifies the ASCII code of the character you wish to define. It must be a value from 0 to 255. *Pattern-identifier* can be a string of any length up to 255 bytes. (This is much longer than XB allows.) 16 bytes are used to define each character. If the length of *pattern identifier* is not a multiple of 16, then zeros are used for the remainder of the character being defined. CHAR2 will wrap around from ASCII 255 and continue starting at ASCII 0 if the pattern identifier string is long enough. Up to 8 character-codes and their pattern-identifier strings can be included in one call to CHAR2

CALL LINK("CHPAT2",character-code,string-variable[,...])

This is the equivalent of CALL CHARPAT in XB. It is used to return in s*tring-variable* the 16 character pattern identifier that specifies the pattern of *character-code. Character-code* must be from 0 to 255. Up to 8 patterns can be read in one call to CHPAT2

CALL LINK("CHSET2")

This is the equivalent of CALL CHARSET in XB. This restores the character patterns for characters 32 to 127 to the default patterns, which are the standard characters used in XB. Then it copies those patterns into characters 160 to 255 but with the bits reversed so they will appear in inverse video.

CALL LINK("CHSETL")

This is a variation of CALL LINK("CHSET2"). It is identical to CHSET2 except that the capital letters are the larger capital letters seen in the TI title screen.

SUBROUTINES FOR SCROLLING

CALL LINK("WINDOW",row1,col1,row2,col2)

Lets you specify a rectangular area on the screen where you would like scrolling to take place. *Row1* and *Row2* are numbers from 1 to 24. *Col1* and *col2* are from 1 to 32. *Row2*

and *Col2* cannot be smaller than *row1* or *col1*. CALL LINK("WINDOW") with no parameters sets the full screen as the window. The full screen is the default window when a program is run under XB256.

CALL LINK("SCRLUP"[,number])

Scrolls the characters contained in the window up one row. The vacated row at the bottom will be filled with spaces. If any number is included then it will be a circular scroll with the characters leaving the window row at the top reappearing in the row at the bottom.

CALL LINK("SCRLDN"[,number])

Scrolls the characters contained in the window down one row. The vacated row at the top will be filled with spaces. If any number is included then it will be a circular scroll with the characters leaving the window row at the bottom reappearing in the row at the top.

CALL LINK("SCRLRT"[,number])

Scrolls the characters contained in the window one column to the right. The vacated column at the left will be filled with spaces. If any number is included then it will be a circular scroll with the characters leaving the window column at the right reappearing in the column at the left.

CALL LINK("SCRLLF"[,number])

Scrolls the characters contained in the window one column to the left. The vacated column at the right will be filled with spaces. If any number is included then it will be a circular scroll with the characters leaving the window column at the left reappearing in the column at the right.

CALL LINK("SCPXUP",string) (Screen2 only)

Scrolls the window up one pixel per call to SCPXUP. Before calling SCPXUP the window should be cleared by filling it with spaces. Use CALL CLEAR if the window is the entire screen or CALL HCHAR if it is smaller than the screen. The first time SCPXUP is called and every 16th time afterward, you must provide a string to display. The string will be truncated if it is longer than the window width. If it is shorter than the window width it will be padded with spaces. The first call to SCPXUP must be followed 15 times with CALL LINK("SCPXUP") without the string. Then provide the next string to display with CALL LINK("SCPXUP",string), and so on. This routine uses the ASCII characters 32 to 127 and 160 to 255. The blank lines between each line of text are needed by the scrolling routine and cannot be avoided. When finished with SCPXUP you should CALL LINK("CHSET2") to restore the characters to their default patterns.

Horizontal scrolling by single pixels.

The next two subprograms, SCPXRT and SCPXLF, allow you to scroll rows of characters horizontally one or more pixel at a time. These are a little different from the scroll routines you may be used to. They work by shifting the character patterns of the specified ASCII characters to the right or left, but the actual characters are not moved. Before calling the routines you set the screen by displaying the sequence of characters that you want to scroll.

For example, print ABCDABCDABCDABCDABCDABCDABCDABCD on the screen. Four characters (ABCD) are used in this row. Then CALL LINK("SCPXRT",65,4,1) would move the character patterns for the 4 characters starting at ASCII 65 (ABCD) one pixel to the right. These are circular scrolls, meaning the bits shifted out of the last character definition(in this example a D) are put into the first character definition. (in this example an A). For clarity, letters were used in this example, but you would normally use different ASCII codes. In a game these would probably be defined with CHAR2 to look like mountains, trees, etc.

If you want to avoid circular scrolling and have a more complex landscape such as in PARSEC you should make the length 33 and only print 32 characters on the screen. After every eight calls to SCPXRT or SCPXLF you should redefine the hidden character using CHAR2 so a new character pattern is ready to be scrolled onto the screen.

These routines permit "parallax scrolling" - you can scroll a row in the foreground faster than the row behind it to give a 3 dimensional effect. The window boundaries have no effect on these routines.

CALL LINK("SCPXRT",ascii code,length,#pixels[,...])

Shifts the character definitions in the selected sequence of characters a specified number of pixels to the left. The sequence of characters starts at *ascii-code* and is *length* characters long. The number of pixels to be shifted is set by *#pixels*. *Ascii-code* can be from 0 to 255, *length* can be from 1 to 33, and *#bits* can be a number from 1 to 7. The sequence given by *ascii-code* and *length* cannot go from 159 to 160. If it does an error message will be issued. Up to 5 sequences of character patterns can be modified per call to SCPXLF. (Screen2 only)

CALL LINK("SCPXLF",ascii code,length,#pixels[,...]) (Screen2 only)

Shifts the character definitions in the selected sequence of characters a specified number of pixels to the left. The sequence of characters starts at *ascii-code* and is *length* characters long. The number of pixels to be shifted is set by *#pixels*. *Ascii-code* can be from 0 to 255, *length* can be from 1 to 33, and *#bits* can be a number from 1 to 7. The sequence given by *ascii-code* and *length* cannot go from 159 to 160. If it does an error message will be issued. Up to 5 sequences of character patterns can be modified per call to SCPXLF. (Screen2 only)

MISCELLANEOUS SUBROUTINES

There are five additional assembly language subroutines that address various limitations in XB.

CALL LINK("HILITE",row,column,length)

HILITE is used to toggle text to inverse video or back to normal. *Row* and *column* determine the first character you want to hilite and *length* is the number of characters to hilite. *Row* is from 1 to 24. *Column* is from 1 to 32. *Length* is from 1 to 256. Usually you would hilite a single line, but the length can be up to 256 characters. HILITE will stop at the lower right of the screen even if the combination of *row, column* and *length* should go off the bottom of the screen. A character in the area being hilited with an ASCII of less than 128 will have 128 will be added. A character in the area being hilited with an ASCII greater than 127 will have 128 subtracted. Because characters 160 to 255 are defined in inverse video when a program runs under XB256 or when CHSET2 is called, the hiliting takes place automatically. A second CALL LINK("HILITE",row,column,length) will restore text to back to normal. (Screen2 only)

CALL LINK("EARLYC",number[,...])

This routine sets the sprite early clock. With the early clock set, the sprite's location is shifted 32 pixels to the left, allowing it to fade in and out on the left edge of the screen. *Number* is the sprite number for which you want to set the early clock. Up to 16 sprites can included in one call to EARLYC. Changing the color of a sprite will turn off the early clock.

CALL LINK("DISPLY",row,col,string[,direction,repeat])

One of the limitations of Extended BASIC is that you are restricted to only 28 columns on the screen. The two columns on both sides of the screen cannot be printed to directly. You have to use HCHAR or VCHAR and even then it's not simple.. DISPLY addresses this limitation. It will print a string starting at screen location row and col, and can use all 32 columns when it prints. By default it will print to the right. If it reaches the right hand edge of the screen it will drop down one row, go to the left side of the screen and continue printing. If it reaches the lower right corner of the screen it will continue printing at the upper left corner of the screen. If you want DISPLY to print in a different direction you can include the optional fourth parameter. A 1 will print each character one space to the right; a 2 will print each character 2 spaces to the right. A 32 will print text vertically down, 33 will print text diagonally down and to the right. A -32 will print text vertically upwards, and so on. If you want DISPLY to print the string repeatedly (somewhat like HCHAR or VCHAR) include the optional fifth parameter to tell DISPLY how many times you want the text to be repeated. If you want to use the repeat function, you must include the direction. Keep in mind that with DISPLY column 1 is the first column, while with DISPLAY AT column 1 is the third column on the screen.

CALL LINK("VREAD", memory address, # bytes, string[, . . .])

VREAD reads the specified number of bytes from the VDP ram into a string variable. Up to 5 strings can read with one call to VREAD.

CALL LINK ("VWRITE",memory address, string[, . . .])

VWRITE writes the string to the VDP ram starting at the specified memory address. Up to 8 strings can be read with one call to VWRITE.

VREAD and VWRITE are extremely versatile subroutines. The programmer can use them to read a row from the screen, erase the row, input text using ACCEPT AT, then restore the screen to its original state. You can open a small window, input text in the window, and scroll just the window. If you want to dig more deeply into the TI than XB normally allows, you can prepare strings ahead of time with a short utility that saves them in MERGE format, then include them in your program to write character definitions, to load sprite definitions and motions, to load a sound list to be played automatically, etc. It is beyond the scope of this manual to go into detail on these more advanced uses although the most important VDP addresses are listed below. The editor/assembler manual has information on memory locations in the VDP and how to set up sprites and sound lists.

Character Sets Used By CALL LINK("COLOR2")

Set	ASCII Codes	Set	ASCII Codes	Set	ASCII Codes
-3(29)	0-7	8	88-95	19	176-183
-2(30)	8-15	9	96-103	20	184-191
-1(31)	16-23	10	104-111	21	192-199
0	24-31	11	112-119	22	200-207
1	32-39	12	120-127	23	208-215
2	40-47	13	128-135	24	216-223
3	48-55	14	136-143	25	224-231
4	56-63	15	144-151	26	232-239
5	64-71	16	152-159	27	240-247
6	72-79	17	160-167	28	248-255
7	80-87	18	168-175		

16K VDP RAM MEMORY MAP WITH XB256

0 – 767	Screen Image Table	VDP address is given by (Row-1)*32+Column-1
768 – 879	Sprite Attribute Table	Room for 28 sprites – each sprite needs 4 bytes vertical position-1,horizontal position, char #+96,color-1(+128 for early clock)
1008 – 1919	Pattern Descriptor Table for Screen1	8 bytes per character – char 30 starts at 1008
1920 – 2047	Sprite Motion Table	4 bytes per sprite. Vertical velocity, horizontal velocity; sys use; sys use (you can use 0 for the sys use value)
2048 – 2079	Color Table for Screen1	1 byte per character set – char set 0 = 2063 (foreground-1)*16+background-1
2392 – 11263	Value stack	Used by XB for strings, etc.
11264 – 12287	Buffer used to store the screen that is not being displayed.	
12288 – 14335	Pattern Descriptor Table for Screen2	8 bytes per character – char 160 starts at 12288 char 0 at 13056, and char 32 at 13312
14336 – 14367	Color Table for Screen2	1 byte per character set – char set 17 =14336 char set 0 = 14352
14368 – 14813	available VDP RAM	can safely be used for sound lists, etc.
14814 – 16106	Disk buffering area for Files(2)	Can be used for sound lists, etc. if you don't use disk access.

SAVING STACK SPACE

Normally Extended BASIC has 11840 bytes of stack space. The additional graphics capabilities of XB256 require extra VDP memory which reduces the stack space to 8808 bytes. It is important to realize that the reduced stack space does not decrease the maximum size that an Extended BASIC program can be. Both the program and all numeric values generated by the program are contained in the 32K memory expansion. Just like in XB, there are 24488 bytes of space available for a program. The only reduction is in stack space. Most programs will run out of program space before running out of stack space, but if you find that stack space is a problem the following may help.

Extended BASIC uses the stack for a number of purposes. After the prescan, it contains a list of all the variable names used by the program. Both string variable names and numeric variable names are in this list, as well as any array names. The stack space needed for each entry in the list is eight bytes plus the number of characters in the name. The prescan also generates a list of all the named subprograms such as JOYST, KEY, LINK and so on. User defined subprograms are also contained in this list. The stack space needed for each entry in this list is eight bytes plus the number of characters in the name. Finally, every string used by the program is also stored in the stack.

One way to conserve stack space is to limit your use of named subprograms. The stack space will not be significantly reduced if you use just a few named subprograms. However, if you are accustomed to writing a lot of your own subprograms, you should convert them to GOSUBs and ON GOSUBs wherever possible.

Stack space can be conserved by using as few numeric variables as possible, and by keeping their names as short as possible. Use numeric constants when possible, since constants require no entry in the list of variable names. Because the actual numeric values are stored in the 32K expansion, numeric arrays require only one entry in the list per array, so very little stack space is used.

Because the actual string is also stored in the stack, strings are the worst offender as far as using up stack space.

Instead of using string variables, use string constants whenever possible. Following are two examples that display text on the screen. The first example uses 28 bytes more stack space than the second:

```
10 A$="THIS IS A TEST"::CALL LINK("PRINT",1,1,A$) ! Uses more
   stack space

10 CALL LINK("PRINT",1,1,"THIS IS A TEST") ! Uses less stack
   space
```

If you must use string variables, reuse the same variable name as many times as possible.

Keep the number of string variables to a minimum. Following are two examples that redefine characters. Because the second example reuses A$, it uses 30 bytes less stack space than the first.

```
10 A$="FFFFFFFFFFFFFFFF" :: CALL LINK("CHAR",40,A$)
12 B$="FF818181818181FF" :: CALL LINK("CHAR",80,B$) !Uses more
   stack space

10 A$="FFFFFFFFFFFFFFFF" :: CALL LINK("CHAR",40,A$)
12 A$="FF818181818181FF" :: CALL LINK("CHAR",80,A$) !Uses less
   stack space
```

String arrays use a lot of stack space. If you are using string arrays and find that you are running short of stack space try keeping the strings in DATA statements and have your program READ them as needed. Following are two examples. The first reads strings from a DATA statement into a string array, where they are ready for use. The second uses 122 bytes less stack space by leaving the strings in the DATA statement until they are needed. The latter method does have the disadvantage of being slightly slower.

```
10 FOR I=1 TO 10 :: READ A$(I) :: NEXT I
20 CALL LINK("PRINT",1,1,A$(7)) ! Print String7 on the screen
100 DATA
   String1,String2,String3,String4,String5,String6,String7,
   String8,String9,String10


10 FOR I=1 TO 7 :: READ A$ :: NEXT I
20 CALL LINK("PRINT",1,1,A$) ! Print String7 on the screen
100 DATA
   String1,String2,String3,String4,String5,String6,String7,
   String8,String9,String10
```