XB256

Expanded Graphics for TI Extended BASIC

by Harry Wilhelm

06/04/18

TABLE OF CONTENTS

INTRODUCTION

XB256 is part of the Extended BASIC Game Developer's package. It 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, so 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 default screen. This is the screen normally used by Extended BASIC and it is accessed using the usual XB graphics statements. 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 preserve the graphics on each screen. When using Screen2 there are assembly equivalents that replace CHAR, CHARPAT, COLOR, and CHARSET. Except for these subroutines, all 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. You can specify a window area for scrolling and leave the rest of the screen unchanged. Other routines let you scroll smoothly one pixel at a time to the left, right, up or down. Yet another lets you do a text crawl like the title scene of “Star Wars”

There are miscellaneous subroutines that let you hilight text, set the sprite early clock, print on the screen using all 32 columns, read from or write to the VDP RAM, write compressed strings to VDP, move sound tables into VDP, play a sound list, freeze and thaw sprite movement, catalog a disk, and more.

A utility (COMPRESS) is included that lets you save selected areas of VDP memory into compressed strings that can be merged with your program. You can save character definitions, sound tables, screen images, etc. in a more compact form that can be loaded much more rapidly.

There are two utilities, SLCOMPILER and SLCONVERT, that can convert the CALL SOUND statements in an XB program into sound lists. This way music and sound effects can be saved in a compressed form that can be loaded directly into VDP memory. This is much more compact, plus your program is free to do other tasks while a sound list plays

EQUIPMENT REQUIRED

XB256 has been tested with a real TI-99/4A and the emulators 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. It will work with the CF7 expansion.

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 preserved and the new screen takes its place. When you break a program by pressing <Fctn 4> the display will automatically revert to Screen1. After breaking 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 will not be 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 8056 bytes (with the default 3 disk files). The stack is primarily used to contain string data and subprogram names. There should be enough room for most programming needs. If your program is very long or uses a lot of strings, you may have to adjust your programming style in order to conserve stack space There is a section in the appendices that describes how to save stack space.

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

Disk files are allocated in the usual manner using CALL FILES(n)

LOADING “XB256”

Set up DSK1 as described in ***Using XBGDP*** so it contains the latest version. (ENCRUZADO). Select Extended BASIC from the TI master title screen. Use the space bar or up/down arrows to select XB256 and press <Enter>. XB256 loads into low memory and NEW is performed. The colors in the immediate mode are changed to black on gray to show that XB256 is loaded and active. In the file XB256, line 10 can be changed to reserve a block of VDP memory starting at 6176 to be used for sound tables. (See page 4)

Or, you may want to set up a disk that autoloads XB256 and then runs a program under XB256. Copy XB256 to a disk, change the name to LOAD and add “RUN “DSKn.PROGNAME to line 10 of the loader. (See page 26 for more information)

USING THE SUBROUTINES

XB256 contains 24 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, OFF, SAVEIV and CAT, all 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("XB256”[,n bytes VDP memory to reserve])
CALL LINK("XB256A”[,n bytes VDP memory to reserve])
Turns the XB256 interrupt routine on and optionally reserves a buffer in the VDP ram for sound lists. *This should only be used in the immediate mode or in the XB256 loader; **never** within an XB program*. You can add a value to reserve 1 to 6000 bytes of VDP ram starting at 6176. Default is to omit the value, reserving no additional memory and maximizing the stack space. This routine can be used to reactivate XB256 if you turned it off with CALL LINK(“OFF”). XB256A is identical but also turns on autocomplete. (Immediate mode)

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. XB256 is still loaded can can be reactivated with CALL LINK(“XB256”) (Immediate mode)

CALL LINK(“SAVEIV”,”DSKn.FILENAME”)
This will save an XB program in IV254 format. See page 26 for more information.

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 standard character set is loaded and the colors are set to white on transparent, with a dark blue screen.

CALL LINK(“SCREEN”,color-code) (both screen1 and screen2)
This can be used to change the background color in the current screen. This is identical to CALL SCREEN except the background colors of the screens are saved/restored when you switch screens. (This is the default action for the compiler, even with CALL SCREEN).

COLOR AND CHARACTER PATTERNS IN SCREEN 2

There are six 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 cause XB256 to save Screen2 and restore Screen1 because it thinks the program is breaking from <Fctn 4>. Sprite patterns use screen1 character definitions, so be sure to use CALL CHAR to define these.

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 with 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.

CALL LINK("CHSETD")
This is another variation of CALL LINK("CHSET2"). It is identical to CHSET2 except that the 0 is slashed and the lower case characters are true lower case with descenders, rather than the small capital letters in the standard character set.

SUBROUTINES FOR SCROLLING

CALL LINK("WINDOW",row1,col1,row2,col2)
Lets you set the boundaries of a rectangular area on the screen where scrolling will take place, and where CALL LINK("DISPLY") will print text if you use a direction of 0. *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. This is the same as CALL LINK("WINDOW",1,1,24,32)

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("CRAWL",string) (Screen2 only)
Lets your program do a "text crawl" similar to the *Star Wars* movies. Before calling CRAWL 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 CRAWL 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 CRAWL must be followed 15 times with CALL LINK("CRAWL") without the string. Then provide the next string to display with CALL LINK("CRAWL",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 CRAWL you should CALL LINK("CHSET2") to restore the characters to their default patterns.

SCROLLING BY SINGLE PIXELS

The next four subprograms allow you to scroll horizontal rows of characters or vertical columns of characters 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 just the character definitions of the specified ASCII characters; 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[,...]) (Screen2 only)
Shifts the character definitions in a horizontal sequence of characters to the right by the specified number of pixels. 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 SCPXRT.

CALL LINK("SCPXLF",ascii code,length,#pixels[,...]) (Screen2 only)
Shifts the character definitions in a horizontal sequence of characters to the left by the specified number of pixels. Setup is the same as SCPXRT.

CALL LINK("SCPXUP",ascii code,length,#pixels[,...]) (Screen2 only)
Shifts the character definitions in a vertical sequence of characters up by the specified number of pixels. Setup is the same as SCPXRT.

CALL LINK("SCPXDN",ascii code,length,#pixels[,...]) (Screen2 only)
Shifts the character definitions in a vertical sequence of characters down by the specified number of pixels. Setup is the same as SCPXRT.

MISCELLANEOUS SUBROUTINES

There are seventeen additional assembly language subroutines that further extend XB256.

CALL LINK(“IRND”,limit,variable[,.....])
Returns a random number as an integer from 0 to limit-1. The results are the same as `variable=INT(limit*RND)` but IRND is *much* faster than the slow XB random number generator. Up to 8 random numbers can be created with one call to IRND.

CALL LINK(“CAT”)
Catalogs a disk drive. This is most useful on a real TI99, but will work fine in emulation. Enter the drive number at the prompt. You can press any key to pause or resume the listing.

CALL LINK("HILITE”,row,column,length) (Screen2 only)
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 as many as 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 set to inverse video when a program runs under XB256 or when CHSET2, CHSETL, or CHSETD is called, the hiliting takes place automatically. If you want to restore text back to normal, do a second CALL LINK(“HILITE”,row,column,length). (Screen2 only)

CALL LINK(“EARLYC”,number[,...])
This routine sets the sprite early clock. When the early clock is 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(“DELAY”,duration)
This routine will pause execution of the program for a period of time from .001 to 30 seconds. Duration is the time to delay in thousandths of a second and must be a number from 1 to 30000. Sprite motions and sounds work normally when using DELAY..

CALL LINK(“SYNC”)
This is useful if you always want a loop to take the same amount of time. Set it up with CALL LOAD(-1,N) where N/60 is the time you want the loop to take. Add CALL LINK(“SYNC”) at the end of the loop just before the NEXT or GOTO that restarts the loop.

```
100 CALL LOAD(-1,60) !want each loop to take exactly 1 second (60/60 seconds)
110 The main loop. When done put ::CALL LINK(“SYNC”)::GOTO 110
```

CALL LINK(“DISPLY”,row,col,string[,direction,repeat])
DISPLY lets you print a string on the screen without the 28 column limitation imposed by XB. It starts printing at screen location *row* and *col*. If it reaches the right hand edge of the screen it drops down one row, goes to the left side of the screen and continues printing. If it reaches the lower right corner of the screen it continues 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 for direction. A 1 prints characters one space to the right; a 2 prints characters 2 spaces to the right. A 32 prints text vertically down, 33 prints text diagonally down and to the right. A -32 prints text vertically upwards, and so on. *If the direction is zero then DISPLY prints from left to right within whatever window boundaries have been defined.* 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. Remember that with DISPLY column 1 is the first column, unlike XB DISPLAY AT where 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 of up to 255 bytes. Up to 5 strings can read with one call to VREAD.

CALL LINK (“VWRITE”,memory address, string[, . . .])
VWRITE writes a string to the VDP ram starting at the specified memory address. Up to 8 strings can be written 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. The most important VDP addresses are listed in the memory map on page 11 below.

CALL LINK (“CWRITE”,compressed string[, . . .])
CWRITE writes a compressed string to VDP ram starting at the memory address embedded in the beginning of the string. Compressed strings are created with the COMPRESS utility described on page 21. Up to 16 compressed strings can be written with one call to CWRITE. If the compressed string is written to the sprite motion table, then the number of sprites in motion is automatically updated so the sprites will move.

CALL LINK(“PLAY”,vdpaddress)
PLAY will start playing the sound list located at the address in VDP RAM. CALL LINK(“PLAY”,0) will stop any sounds that are being played. The sound list must first be moved to VDP RAM before it can be played. Typically, it would be saved in data statements using the COMPRESS utility described below, merged into your XB program, then written to VDP with CWRITE. If a sound list is playing and you want to use a CALL SOUND statement, you can either turn off the sound list with CALL LINK(“PLAY”,0) or interrupt the sound list using a negative duration for the first CALL SOUND.

CALL LINK(“ST2VDP”,1 or 2)
This subroutine is normally used by COMPRESS or in the immediate mode. An assembly language sound table must first have been created. The easiest way to do this is to write it in XB, then use SLCOMPILER or SLCONVERT to convert it to assembly language source code. This should then be assembled into object code. Then the object code is loaded into cpu ram with CALL LOAD(“DSKn.SOUNDLISTO”). After it is loaded, CALL LINK (“ST2VDP”,1) will copy the sound list into a 640 byte long buffer in VDP ram starting at 2432 (>0980). If the sound table is longer than 640 bytes then an error message is issued. CALL LINK (“ST2VDP”,2) will copy the sound table into a buffer you have reserved starting at 6176 (>1820). If you have not reserved enough memory, you will be told to CALL LINK(“XB256”,n) where “n” is the number bytes needed needed to contain the sound table. When first loading and testing a sound table it is best to be in the immediate mode. In the immediate mode the first and last addresses of the sound table are shown, as well as any labels that were added to the REF table along with their addresses. You can then use CALL LINK(“PLAY”,address) to check that the sound lists play correctly. See the section below on page 12 entitled *Sound Lists and Sound Tables* for more information.

CALL LINK(“FREEZE”)
FREEZE will turn off automatic sprite motion for all sprites..

CALL LINK(“THAW”)
THAW will turn on automatic sprite motion for all sprites. You can freeze the sprites, put as many as you like on the screen, then thaw them to set them in motion simultaneously.

CALL LINK(“RUN”,A$)
RUN is used to run an XB256 program from a running program. It does the same thing as RUN “DSK1.PROGRAM” but you can use a string variable which is not permitted in XB. This could be useful in a menu program or when chaining programs together.

CALL LINK(“RUNL1”,A$)
RUNL1 is used to run an XB256 program from a running program. It is similar to CALL LINK(“RUN”) but there is a major difference. After it loads the program it runs line #1, *but without doing a prescan or resetting any of the variables of the calling program.* By chaining programs together with RUNL1 you can create a very long XB256 program, with instant start up of the chained programs. See page 26 for more information.

CALL LINK(“SAVEIV”,”DSKn.PROGNAME”).
Used to save an already loaded program to disk in IV254 format. This is the format used by XB when saving long programs to disk. CALL LINK ("SAVEIV","DSK2.TEST3") will save the current XB program as TEST3 onto DSK2. If the program is very small, line 1 is created to pad the program with a DATA statement, so you can save even the shortest XB program as IV254. (Immediate mode)

More information on using RUN, RUNL1 and SAVEIV can be found in the section starting on page 26 entitled AUTOLOADING AND CHAINING PROGRAMS.

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 >0000->02FF	Screen Image Table	VDP address is given by (Row-1)*32+Column-1
768 – 879 >0300->036F	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 >03F0->077F	Pattern Descriptor Table for Screen1	8 bytes per character – char 30 starts at 1008
1920 – 2031 >0780->07EF	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 >0800->081F	Color Table for Screen1	1 byte per character set – char set 0 = 2063 (foreground-1)*16+background-1
2432 – 3071 >0980 - >0BFF	Free space - sound tables	640 bytes for sound tables, etc.
3072 – 4095 >0C00 ->0FFF	Buffer used to store the screen that is not being displayed.	
4096 – 6143 >1000 - >17FF	Pattern Descriptor Table for Screen2	8 bytes per character – char 160 starts at 4096 char 0 at 4864, and char 32 at 5120
6144 - 6175 >1800 - >181F	Color Table for Screen2	1 byte per character set – char set 17 =6144
6176 - 14295 >1820 - >37D7	Value stack	Used by XB for strings, etc. CALL LINK(XB256,n) will reserve N bytes starting at 6176
14296 – 16383 >37D8 - >3FFF	Disk Buffering Area for CALL FILES(3)	

SOUND LISTS AND SOUND TABLES

For the purposes of this manual, a sound list is a sequence of bytes to be sent to the sound generator that can play a theme song, make an explosion sound, a chime sound, etc. A sound table is a collection of sound lists containing various sound effects to be used by your program. After a sound table is loaded into VDP RAM you can play any sound list in it using CALL LINK("PLAY",address) Once the sound list has been started it will continue playing automatically until it reaches the end, while the XB program is free to do other things. Additionally, XB256 has a second player capable of playing a different sound list simultaneously with the first. This way you can have background music playing and add sound effects without interrupting the background music. The second sound list player can play alongside either the first soundlist player or standard CALL SOUNDs.

There are two memory locations in VDP that can contain a sound table:
-At 2432 (>0980) is a 640 byte long buffer that is always available.
-At 6176 (>1820) is a user defined buffer. Use CALL LINK("XB256,N) from the command line or from the XB256 loader to reserve N number of bytes starting at 6176.

CREATING A SOUND TABLE FROM EXTENDED BASIC

There are two utilities, SLCOMPILER and SLCONVERT, that can be used in Extended BASIC to create sound tables. Either of these will automatically convert an XB program containing CALL SOUND statements into an assembly language source code file that can then be assembled and loaded into XB256.

SLCOMPILER

To use SLCOMPILER you must save the XB program containing the CALL SOUNDs in merge format. SLCOMPILER is an XB program that reads the saved merge format program, converts any CALL SOUND it finds into the proper data, and adds the data to a sound table. The XB program with the CALL SOUNDs must follow these guidelines:

Multiple statement lines can be used. Any legal XB statement can be used, but only those statements listed below will be recognized. Anything else will be ignored.
CALL SOUND statements can only use numeric constants. Numeric variables will not compile correctly. Negative durations will be compiled as positive durations.
GOTO is implemented, but you cannot use GO TO
FOR/NEXT loops are implemented. The values must be numeric constants, must start at 1, and cannot loop more than 9 times.
END or STOP should be placed at the end of each sound list to tell the compiler to turn off all the sound generators that were used.
The following comments are used to send instructions to the compiler
!PLAYER2 tells the compiler to play the sound list using the second player.
!LGENON tells the compiler to not turn off generators. Pages 19 and 20 have more details.
!LABEL - Use a remark to name and provide an entry point for each sound list. This should be a separate line just before the entry point. A sound list can have multiple entry points by adding a new label for each entry point. All REM remarks and ! remarks more than 6 bytes long will be ignored.

The second sound list player has some minor limitations. The sound list must play straight through without FOR/NEXT loops or GOTO's. It should not use the same sound generators used by the first sound list. For example, if a theme song uses generators 1 and 2, then a sound list played by player2 should only use generators 3 and 4 (the noise generator).

SLCOMPILER TUTORIAL

To get you started, here is a step by step tutorial that shows the steps necessary to create and load a simple sound table using SLCOMPILER. After this tutorial there is a section describing SLCONVERT,; then the fine points of using SLCOMPILER or SLCONVERT will be described.

Select XB256 from the main menu or RUN “DSK1.XB256”
Create an XB program following the guidelines above. For your first project, try keying in the program below, or copy and paste into Classic99.

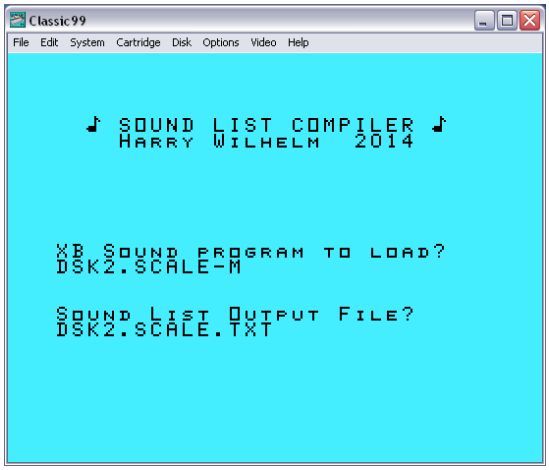
```
100 !CHORD          !entry point at start of first sound list
110 FOR I=1 TO 2
120 CALL SOUND(600,233,0):: CALL SOUND(600,233,0,294,0):: CALL
SOUND(600,233,0,294,0,349,0):: CALL SOUND(250,233,30)
130 NEXT I          !When done looping continue to line 140
140 !SCALE          !alternate entry point. Line 210 loops back to here.
150 CALL SOUND(600,233,0):: CALL SOUND(600,262,0):: CALL
SOUND(600,294,0):: CALL SOUND(600,311,0):: CALL SOUND(600,349,0)
160 CALL SOUND(600,392,0):: CALL SOUND(600,440,0)
180 CALL SOUND(600,466,0):: CALL SOUND(600,440,0):: CALL SOUND(600,392,0)
190 CALL SOUND(600,349,0):: CALL SOUND(600,311,0):: CALL
SOUND(600,294,0):: CALL SOUND(600,262,0):: CALL SOUND(1200,233,0)
200 CALL SOUND(600,233,30)
210 GOTO 140        !endless loop to line 140
220 STOP            !turn off all sound generators
230 !CHARGE         !entry point at start of second sound list
240 !PLAYER2        !tells the compiler to use player2
250 CALL SOUND(100,466,0):: CALL SOUND(100,587,0):: CALL
SOUND(100,698,0):: CALL SOUND(225,932,0):: CALL SOUND(75,784,0):: CALL
SOUND(600,932,0)
260 STOP            !turn off all sound generators
```

Lines 100 to 130 (CHORD) play a chord that builds up from 1 to 3 notes. This loops twice. Lines 140 to 210 play a scale, then go back to 140 to keep playing it. Lines 230 to 260 use the second sound list player to play the familiar “Charge” song heard at baseball games.

You can test the lists with RUN 100, RUN 140, and RUN 230. When you are happy with the sounds, save the program in MERGE format:

SAVE DSK2.SCALE-M,MERGE (*be sure it is in merge format)*
RUN “DSK1.SLCOMPILER” *SLCOMPILER should be run under XB256*

After entering the file names the screen should look something like this:

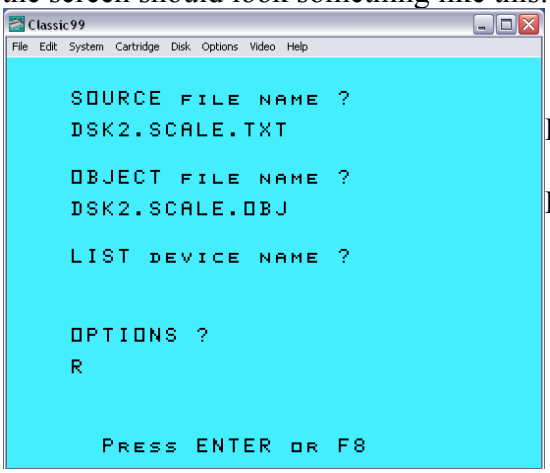


Press Enter and the XB program is converted to an assembly language source code file.

Now the source code file must be assembled into an object code file.

Do not use Asm994a. It is an outstanding program, but has a bug that can cause an error when assembling a sound table. This happens when a byte that falls on an even address is followed by two or more labels before the next byte in the sound list. The next byte should be at an odd address but Asm994a forces it to be at the next even address.

For best results you should use the TI Assembler contained in the XB Game Developer's Package. Type “BYE” and press 2 for Extended BASIC. The assembler is the fourth option in the menu. After entering the file names, (there is no autocomplete here for the file names) the screen should look something like this:



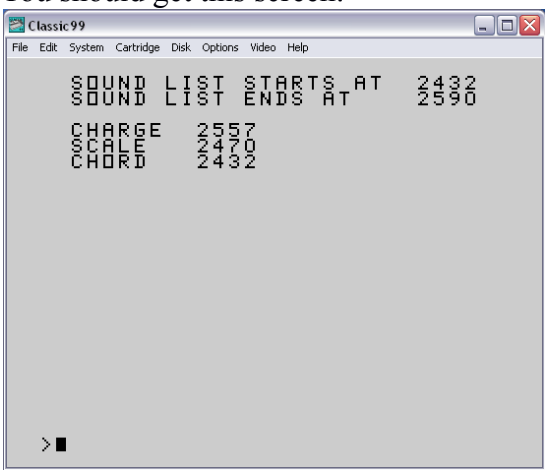
DSK2.SCALE-S on real TI99 or Win994a

DSK2.SCALE-O on real TI99 or Win994a

Press Enter to create the object code file or F8 to redo the file names..
Press ENTER when the assembler is finished.

Now it's time to test out how the sound table works:
Select XB256 from the main menu.
Press F3 to kill the autocomplete routine.
CALL LOAD(“DSK2.SCALE.OBJ”) *loads the object code*
CALL LINK(“ST2VDP”,1) *moves sound table into VDP buffer1*

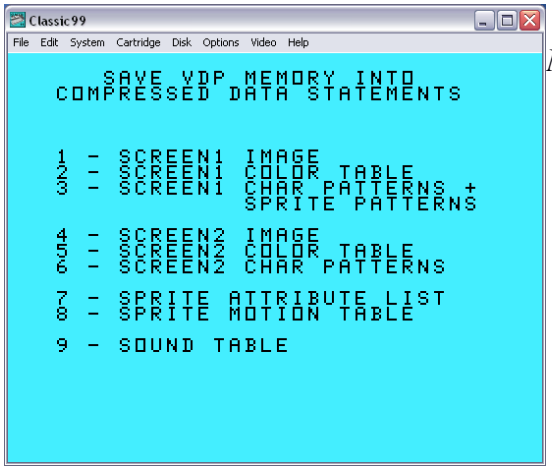
You should get this screen:



Now let's see how it sounds. Type:
CALL LINK(“PLAY”,2432) - will play the chord twice and then go into the scale. Because of the GOTO the scale will not stop. If you want to stop the player, enter any letter to cause an error. The beep will stop the sound list
CALL LINK(“PLAY”,2470) - will play the scale.
CALL LINK(“PLAY”,2557) - will play “charge”.

If the sound table works as expected, it's time to incorporate it in a program. With XB256 running, type:
NEW
MERGE DSK1.COMPRESS *part of the Game Developers Package in Disk 1*
100 CALL COMPRESS(1) *run COMPRESS in screen 1*
110 END
RUN

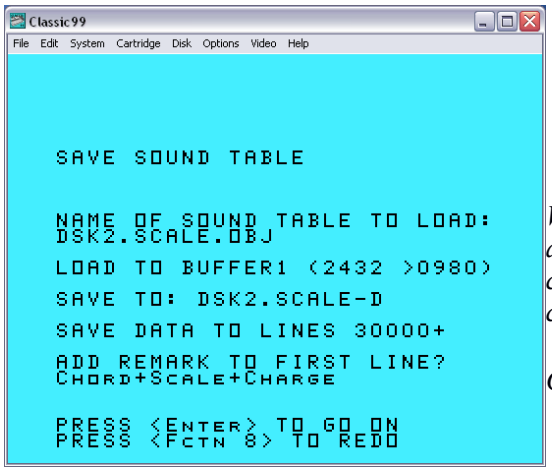
You will see this screen:



NE

Press “9”.to save a sound table as compressed DATA statements.

After entering the file names and other requested information the screen should look something like this:

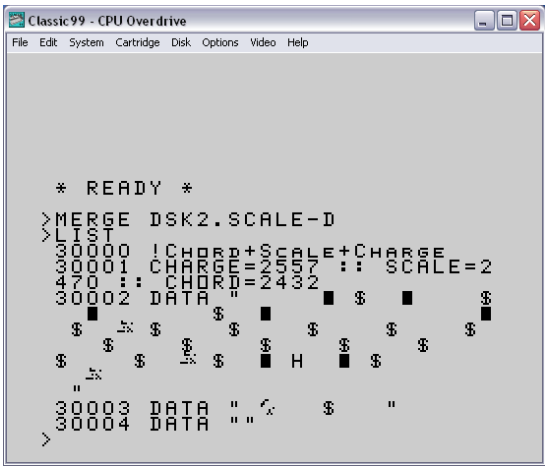


When using COMPRESS to save a sound table you can append a "-D" to show that it is DATA and to avoid confusing it with the Merge format program you just compiled.

Chord+Scale+Charge will become a comment

Press <Enter>. When COMPRESS returns to the main screen the sound table will have been saved in merge format.

Now let's add it to a program. Press <Fctn 4> to break COMPRESS, then type:
NEW
MERGE DSK2.SCALE-D
LIST *and you will see this screen:*



The addresses of CHARGE, SCALE and CHORD are in line 30001 and the sound table is contained in the strange looking DATA statements in line 3002 and 3003. Change line 30001 to line 20, remove line 30001 and add lines 30 to 60 to get:

```
20 CHARGE=2557 :: SCALE=2470::CHORD=2432
30 READ A$::IF A$="" THEN 40::CALL LINK("CWRITE",A$)::GOTO 30
40 CALL LINK("PLAY",CHORD)
50 CALL KEY(0,K,S):: IF S<1 THEN 50
60 CALL LINK("PLAY",CHARGE):: GOTO 50
30000 !Chord+Scale+Charge
30002 DATA "strange looking sound table"
30003 DATA "strange looking sound table"
30004 DATA ""
```

RUN the program and the scale will keep playing until you break the program. Press any key to play "Charge" in addition to the scale. This XB program can be saved normally.

This may seem like a lot of steps, but using SLCOMPILER and COMPRESS as described above makes it possible to have the sound tables contained as an integral part of the XB program. This way sound tables can be rapidly loaded without the need for disk access and can play without using CALL SOUND statements. Besides playing smoothly and automatically, sound lists can save a great deal of memory. The XB program on page 13 above uses 661 bytes of memory. The sound table created takes only 194 bytes.

SLCONVERT

When using SLCOMPILER the values in the CALL SOUND statements have to be numeric constants. This can be quite limiting; it is very convenient to use numeric variables or numeric expressions with CALL SOUND. In fact, most of the existing XB music programs use variables to compute the values.

SLCONVERT was written to to deal with this limitation. It is an adaptation of SLCOMPILER but it uses a different technique for creating a sound table.

As described above, when using SLCOMPILER comments are used to send instructions to the compiler and it can recognize GOTO, FOR/NEXT loops and END/STOP.

But SLCONVERT is *very* different. With SLCONVERT, you first load or write a XB music program. SLCONVERT is then merged with the XB music program and the resulting program is run.. The program runs normally, but whenever XB performs a CALL SOUND it is intercepted by SLCONVERT. The values that were passed to CALL SOUND are used to create the sound list. This way *any* numeric expression that works with CALL SOUND can be used to create a sound list. If CALL SOUND can play it, SLCONVERT can convert it.

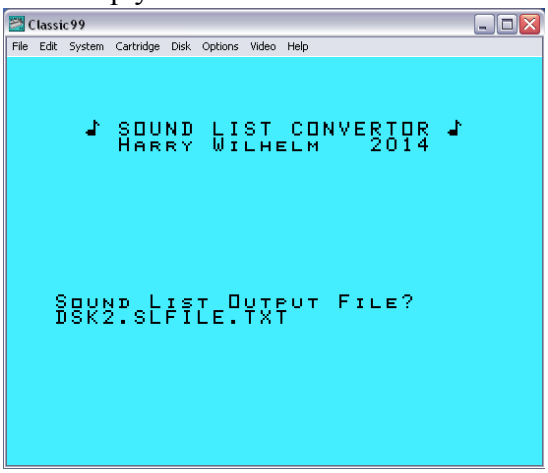
The down side of this method is that it knows nothing about the XB program that performs the CALL SOUND statements. You have to be able to communicate with SLCONVERT. This is done by adding subprograms to the program that act as instructions.

CALL LABEL(A$) is used to add a label to the sound list. This can be a pointer to the beginning of the sound list, or it can be a label to jump or loop to. A$ must be no more than 6 bytes long
CALL GOTO(A$) tells the player to jump to a label created with the CALL LABEL.
CALL LOOP(A$,N) is used to tell the player to loop to the label A$. It will loop N times, then the sound list goes on to the next CALL SOUND.
CALL STOP is used at the end of each sound list. It tells the sound list creator to turn off all the sound generators, then it will reset SLCONVERT to the default values, which are to use Player1 and to turn off the sound generators.
CALL PLAYER2 tells the sound list processor to use player 2. This player is useful for simple sound effects such as explosions, beeps, etc.
CALL LGENON tells the sound list creator to leave the sound generators on as described in pages 18-19 of this manual.
CALL LASTLINE is used when finished making the sound table. It tells the sound list converter to finish up the sound lists and close the file.

As usual, you should test the program thoroughly to make sure it works properly. Be sure you are using XB256.The XB music program can use FOR/NEXT loops, GOTO, STOP or END when testing, but they must be removed when you run SLCONVERT. Once the program works as desired save a copy, then MERGE DSK1.SLCONVERT.

Now for the tricky part. You need to add instructions that tell SLCONVERT what to do:
-Put CALL LABEL("NAME") at the entry point of each sound list you want to play.
-FOR should be deleted or deactivated with a REM or !
-NEXT should be replaced with CALL LOOP("LABEL,N)-this loops N times to LABEL.
-GOTO should be replaced with CALL GOTO(LABEL)
-Add CALL STOP at the places where you want the sound list to stop.
-If you want to use player2, add CALL PLAYER2 at the start of that sound list.
-Use CALL LGENON to leave the sound generators on as described in pages 18-19.
-Put CALL LASTLINE after the last CALL SOUND.

When you are finished you should have modified the XB program so it runs from beginning to end, doing every CALL SOUND just one time. Save a backup copy just in case, then Run the program.
At startup you will see this:



Enter the file name of the sound list source code you want to create and press enter. The XB program will run and when CALL SOUND statements are processed, SLCONVERT will convert them to source code for the sound table. The program stops when finished. You will not hear any sounds when it is running because SLCONVERT uses an assembly language subroutine that modifies every CALL SOUND to be SOUN3, SOUN5, SOUN7 or SOUN9. You can see this if you list the program after running it.

Once SLCONVERT has created the source code file, go back to page 14 and follow the steps there to assemble and load the sound table.

Default settings at startup are to turn off unused sound generators (standard behavior in XB) and to use player1, the main sound list player that can handle jumps and loops. Line number 1 or line numbers greater than 32000 cannot be used; they are needed by SLCONVERT.

Below is the demo sound program from page 13 showing the modifications needed to work with SLCONVERT. The old instructions are kept as comments. To familiarize yourself with SLCONVERT, start by studying the changes that were made and then follow these steps.

First key in this program or copy and paste into CLASSIC99.
MERGE DSK1.SLCONVERT
RUN - The sound list will be created. Follow the steps on page 14 to assemble and load.

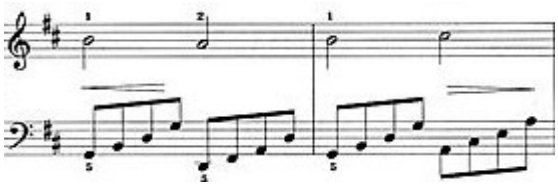
```
100 CALL LABEL("CHORD")      !CHORD      entry point at start of first sound list
110                          !FOR I=1 TO 2 not used – see 130
120 CALL SOUND(600,233,0):: CALL SOUND(600,233,0,294,0):: CALL
SOUND(600,233,0,294,0,349,0):: CALL SOUND(250,233,30)
130 CALL LOOP("CHORD",2)     !NEXT I loop to CHORD two times, then 140
140 CALL LABEL("SCALE")      !SCALE      alternate entry point. Line 210 branches here
150 CALL SOUND(600,233,0):: CALL SOUND(600,262,0):: CALL
SOUND(600,294,0):: CALL SOUND(600,311,0):: CALL SOUND(600,349,0)
160 CALL SOUND(600,392,0):: CALL SOUND(600,440,0)
180 CALL SOUND(600,466,0):: CALL SOUND(600,440,0):: CALL
SOUND(600,392,0)
190 CALL SOUND(600,349,0):: CALL SOUND(600,311,0):: CALL
SOUND(600,294,0):: CALL SOUND(600,262,0):: CALL SOUND(1200,233,0)
200 CALL SOUND(600,233,30)
210 CALL GOTO("SCALE")       !GOTO 140   Branches back to line 140 (SCALE)
220 CALL STOP                !STOP       Turn off all sound generators
230 CALL LABEL("CHARGE")     !CHARGE     entry point at start of second sound list
240 CALL PLAYER2             !PLAYER2    tells the converter to use player2
250 CALL SOUND(100,466,0):: CALL SOUND(100,587,0):: CALL
SOUND(100,698,0):: CALL SOUND(225,932,0):: CALL SOUND(75,784,0):: CALL
SOUND(600,932,0)
260 CALL STOP                !STOP       turn off all sound generators
270 CALL LASTLINE            !STOP       all done, finish up and close file
```

More information about SLCOMPILER and SLCONVERT

The TMS9919 sound chip has three sound generators and one noise generator. When using the main sound player the compiler starts with sound generator #1 and works up as needed. When PLAYER2 is selected the compiler starts with generator #3 and works down as needed. For example, CALL SOUND(500,220,0,330,0) needs two generators. With the normal player, generator 1 will play 220 hz and generator 2 will play 330 hz. If using PLAYER2, generator 3 will play 220 hz and generator 2 will play 330 hz. If you want to use the two players simultaneously, it is possible that your sound lists might overlap. If this happens not all the notes would play, but it would not cause any other problems.

SLCOMPILER and SLCONVERT give you an option to leave the sound generators on. Consider this line: 10 CALL SOUND(1000,220,0,330,0,440,0)::CALL SOUND (1000,660,0) Normally when the second CALL SOUND is processed generators 2 and 3 are turned off, which is how XB behaves. You can use LGENON in your program if you want to leave those generators on. Then they will keep playing until you specify a volume of 30 or when the program comes to STOP or END. In the line above, you can silence generators 2 and 3 with CALL SOUND (1000,660,0,330,30, 440,30).

Why would you want to leave the sound generators on? Consider these measures from Pachelbel's *Canon in D:*



Normally in XB you would have to treat the half notes as 4 eighth notes because the half notes are turned off every time a new eighth note is played. But if you don't turn off the generators, then you could write it this way: (CALL SOUND(eighth note,half note), CALL SOUND(eighth note), CALL SOUND(eighth note), CALL SOUND(eighth note) and the half note keeps playing until you come to the next measure, CALL SOUND(eighth note,half note) which starts playing a new half note. You will have to be careful in testing this, because the compiled sound list will behave differently than it did when testing in XB

There is one condition that can cause unexpected behavior. If the CALL SOUND statement just before a NEXT or GOTO contains more tones than the CALL SOUND executed after NEXT or GOTO, then the extra sound generators are *not* turned off. by the sound list player. (This will not show up when testing in XB because XB *does* turn them off.) If this is a problem, you can turn them off by adding one or more frequencies with a volume of 30 to the CALL SOUND just before NEXT or GOTO.

Creating a Sound Table in Assembly Language

Although SLCOMPILE/SLCONVERT will create XB256 compatible sound tables automatically, more advanced programmers may wish to create their own sound tables in assembly language as described in pages 312 to 324 of the Editor/Assembler manual. Or you may have two or more sound tables you wish to combine into a single large sound table.

The sample code below shows the format expected by XB256. (Since the point of this code is to show the differences from a normal assembly language sound list, most of the sound bytes have been deleted for the sake of clarity.) The lines hilited in yellow must be at the beginning and end of the sound table so that ST2VDP knows where to load the program and how long it is. The DEFs point to where you want each individual sound to start playing. You can have as many DEFs as you want.
The lines hilited in cyan tell the sound list player to do something special. BYTE 0 followed by a label tells the player to jump to that label. BYTE >FE tells the player to loop. The next word is the address to loop to, followed by the number of times to loop (this number is needed twice) and then a zero. Be sure that every value in the sound table is a BYTE. The only exceptions are the DATA statements shown hilited below. The format of the special codes 0 and >FE for jumping and looping are compatible with Matt Haggarty's assembly language sound list player; the >FD code for the second player is not supported by that player.

```
        DEF HONEYC
        DEF SOUND2
        DEF SOUND3
        DEF CHARGE

        AORG >A000
        DATA SLEND,0,0

HONEYC                                  Shows how to GOTO
        BYTE 9,133,42,144,166,8,176,204,31,223,18
        BYTE 0                          0 tells player to GOTO next address
        DATA HONEYC                     GOTO HONEYC

SOUND2                                  Shows how to loop
        BYTE >09,>8C,>1F,>91,>A2,>15,>B1,>C9,>0A,>D0,>19
        BYTE >FE                        >FE tells player to loop
        DATA SOUND2,>0404,>0000         Loop to SOUND2, loop 4 times
        BYTE >04,>9F,>BF,>DF,>FF,>00  bytes to turn off all generators
SOUND3                                  Shows nested loops
        BYTE 9,133,42,144,166,8,176,204,31,223,18
S3LP1   BYTE 3,166,8,176,18
        BYTE 3,167,9,176,9
S3LP2   BYTE 6,164,28,176,201,10,208,18
        BYTE 6,172,31,191,204,31,223,9
        BYTE >FE                        >FE tells player to loop
        DATA S3LP2,>0303,0              Loop to S3LP2, loop 3 times
        BYTE 6,140,31,144,175,7,176,18
        BYTE >FE                        >FE tells player to loop
        DATA S3LP1,>0202,0              Loop to S3LP1, loop 2 times
        BYTE >04,>9F,>BF,>DF,>FF,>00   Turn off 4 generators + stop player
CHARGE
        BYTE >FD                        >FD selects 2nd sound list player
        BYTE 3,>D0,>C0,>0F,4   Bb       Using generator #3
        BYTE 1,>DF,0                    Turn off generator #3 and stop player
SLEND   BYTE 0
        END
```

Once created, the sound table should be assembled and loaded as described above in the SLCOMPILER tutorial.

You may have noticed that DATA can follow BYTE without using an EVEN directive. When the sound list player comes to an >FE or to a >00 it will advance the sound list counter to the next even address to find where to jump to.

COMPRESS UTILITY

COMPRESS is a utility written for XB256 that creates MERGE format disk files with DATA statements that contain images of selected portions of VDP memory. These can be merged into an XB program and then written to VDP memory using CWRITE. The idea is that you first use an XB program to write to the screen, create sprites, define characters, etc. Once the graphics are created, COMPRESS is used to save the desired portion of VDP memory. If you have made a sound table, COMPRESS will load it into memory and then save an image of it. A sound table can be put into the disk buffers (default) or into a different area of VDP memory. Once these are saved, you delete the program lines that created the graphics, merge the files created by COMPRESS and add a simple one line routine that writes them to VDP using much less memory and in a fraction of the time..

COMPRESS is a subprogram that should be merged into your program. A line must be added to the program that does CALL COMPRESS(number). The number must be 1 or 2 and it tells COMPRESS which screen to use when it opens. You want to open COMPRESS in the screen that your program is *not* using. XB256 must be loaded and running when using COMPRESS.

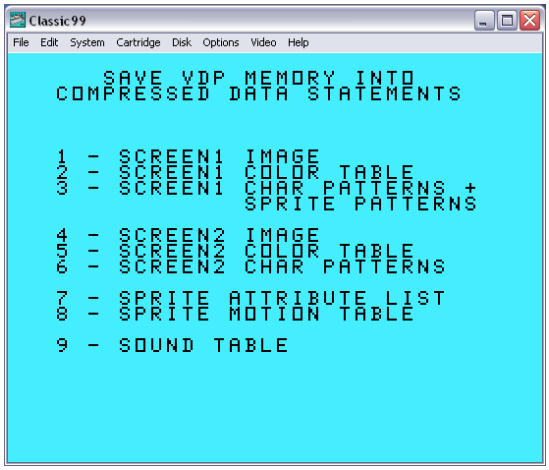
Following is a short example of how to use COMPRESS:

Let's say you are writing a game in XB256 that uses SCRN2:
Lines 100 to 300 draw a maze on the screen.
Lines 300 to 370 define characters 128 to 159
Lines 380 to 400 define character colors.
Lines 410+ are the engine that drives the game.

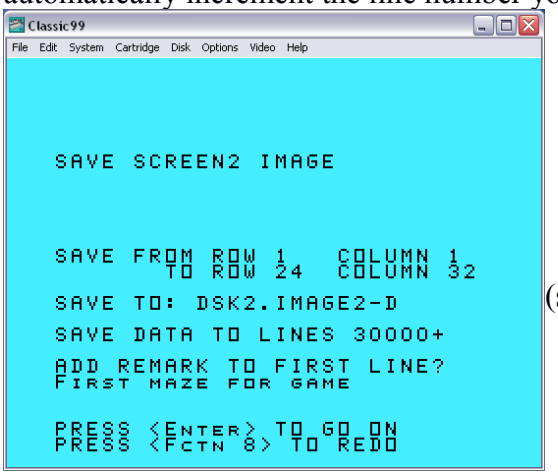
With the program loaded but not running, type:

MERGE DSK1.COMPRESS, then add this line:
405 CALL COMPRESS(1)

Run the program. When it gets to line 405 the graphics colors and characters will have been defined and CALL COMPRESS(1) is executed in line 405, the following screen will appear: (You are using SCRN2 for the game, so you opened COMPRESS in SCRN1.)



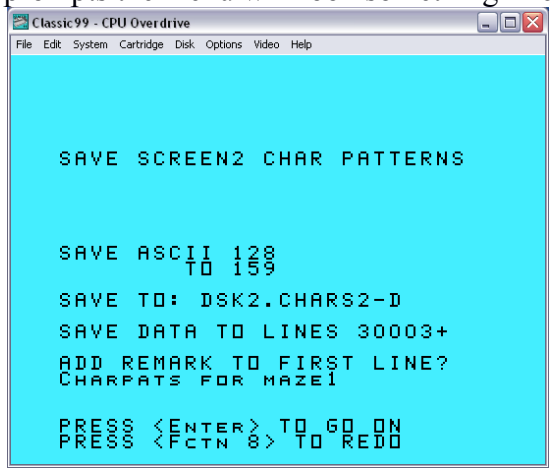
Press 4 to save SCREEN2 image. After answering the prompts the menu will look something like this: (COMPRESS suggests 30000 as the starting line number and will automatically increment the line number you are saving to.)



(should be -D to identify as DATA)

Press <Enter>

When the main menu appears press 6 to save the character patterns. After answering the prompts the menu will look something like this::



Press <Enter>

Similarly, press 5 and follow the prompts to save the color table. When finished press <Fctn 4> to exit the utility.

Option 9 lets you load a sound table created by SLCOMPILER. Just follow the prompts to load and save the sound table. Later, when you merge the sound table, the first or second line (depending on whether you made a comment line) contains the labels and their addresses like so: 30001 CHARGE=2479 :: SCALE=2392 and so on. Change the line number of this to a low number so your program comes to it before playing any sounds. Then CALL LINK("PLAY",SCALE) will play the soundlist called SCALE.

Now you can reload the XB program (without COMPRESS). Merge the 3 files you just created. Delete lines 100 to 400 which are no longer needed and replace them with:
100 RESTORE 30000 (*If compiling then RESTORE must be to a DATA statement!)*
110 READ A$::IF A$="" THEN 410::CALL LINK("CWRITE",A$)::GOTO 110

Each MERGE format file created by COMPRESS ends with DATA "" which line 110 uses at the flag to exit. Because you merged three files there are three lines with DATA "". The next step is to find and delete the first two of these lines so there is only one DATA "" at the end of the DATA statements. Using a null string as the flag means your program doesn't need to know the number of DATA statements to be written. *Important! If you intend to compile the program then RESTORE line number must be to a DATA statement, not the comment before the DATA statement!*

The DATA statements that are created may contain ASCII characters from 0 to 255. They can be LISTed but XB will not let you edit them because they contain characters not recognized by the line editor. It is OK to resequence a program containing these data statements.

The following is for those interested in the makeup of the strings.

The first two bytes are the starting address in VDP ram. The next byte is a length byte. If this is from 1 to 127 CWRITE prints that number of consecutive bytes to VDP. If the length byte is from 129 to 255 CWRITE will subtract 128 from the length and repeat the next byte the specified number of times.

If "HELLOxxxxxxxxxxHOW ARE YOUyyyyyyyyyyyy" is on the screen at the bottom at 736 (Row 24, Col 1), COMPRESS would make this string:
(2)(224)(5)HELLO(139)x(11)HOW ARE YOU(140)y

If you save the sprite motion table, the number of sprites in motion at the time of the save is placed immediately after the two address bytes. CWRITE will restore that number when it writes the string to VDP memory

SAVING STACK SPACE

Normally Extended BASIC has 11840 bytes of stack space in the VDP ram. The additional graphics capabilities of XB256 require some of that VDP memory. If 1 disk file has been allocated the stack space is 9092 bytes; if 2 disk files then it will be 8574 bytes, and if the default 3 files have been allocated it will be 8056 bytes. Also, if VDP ram has been reserved with CALL LINK("XB256,n) then the stack space will be further reduced. 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.

The simplest way to gain stack space is to CALL FILES(1) or CALL FILES(2). Try the following if that still gives an insufficient amount of stack space:

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. (Named subprograms cannot be used in a compiled program.)

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 for 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$="THIS IS A TEST"::DISPLAY AT(1,1):A$ ! Uses more
   stack space

10 DISPLAY AT(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$="FFFFFFFFFFFFFFFF" :: CALL LINK("CHAR2",40,A$)
   12 B$="FF818181818181FF" :: CALL LINK("CHAR2",80,B$) !Uses
more  stack space

   10 A$="FFFFFFFFFFFFFFFF" :: CALL LINK("CHAR2",40,A$)
   12 A$="FF818181818181FF" :: CALL LINK("CHAR2",80,A$) !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$(I) :: NEXT I
20 DISPLAY AT(1,1):A$(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 DISPLAY AT(1,1):A$) ! Print String7 on the screen
100 DATA
  String1,String2,String3,String4,String5,String6,String7,
  String8,String9,String10
```

AUTOLOADING AND CHAINING PROGRAMS

You may develop a program where you want to have XB256 load automatically and then load and run your program or even a chain of programs. The first part is easy. First type OLD DSK1.XB256. If you want to automatically run a program, change line 10 to:

```
5 !XB256  H. Wilhelm 2017 Free distribution only
10 CALL INIT :: CALL LOAD(8192,255,152):: CALL LINK("X")::
CALL LINK("XB256")):: RUN "DSKn.PROGNAME"
```

(not XB256A - you don't want to use autoprompt)

If your program uses the second sound table buffer at 6176 (>1820), when you merged the sound table file created by COMPRESS there was a line 1 which is a remark telling how many bytes must be reserved for the sound list. Using that information, change the CALL LINK(“XB256”) to CALL LINK(“XB256”,number-of-bytes). If you don't use that buffer then no change is needed.

Then save the XB256 loader you just modified under the filename LOAD. (Don't save to the XBGDP disk or you will overwrite the menu program!) It loads and runs automatically if it is in disk 1 when you select XB from the master title screen. If the program you are running is self contained nothing else needs to be done.

Now let's say you have written a complex program that needs to be in two or more parts. You have put all the character definitions, loading of sound lists, etc. into the first program and then want that program to run a second program while preserving the characters you just defined. Simple enough, just add RUN “DSKn.PARTTWO to the first program and it will load and run PARTTWO, or you can use CALL LINK(“RUN”,A$) if the disk name is a string variable. *But if you want to preserve sound tables or the screen2 character definitions and colors then PARTTWO* ***must*** *be in IV254 format.*

Internal, Variable 254 is the format used by Extended BASIC when saving or loading large programs that are longer than the available stack space. If the program is large enough it will be saved in this format automatically. If you don't know what format the program has been saved in, you can CALL LINK(“CAT”) from XB256 to catalog the disk and display the file types.

CALL LINK(“SAVEIV”,”DSKn.PROGNAME”)
XB256 has a subprogram that is be used to force XB to save a program of any length in IV254 format. In the immediate mode, with XB256 running and your program loaded, simply type:CALL LINK(“SAVEIV”,”DSKn.PROGNAME”) The program will be saved in IV254 format. If the program is very short a line 1 will automatically be added. This has a DATA statement that does nothing other than adding enough bytes to permit saving even the shortest programs in IV254 format. This is not necessary if the program is larger than 256 bytes.
This utility should be compatible with a CF7 device, but has not been tested.

CALL LINK(“RUN”,A$)

RUN is used to run an XB256 program from a running program. The normal way to do this in XB is with RUN “DSK1.PROGRAM”, which normally works fine. The only reason for using CALL LINK(“RUN”) would be to run a file name contained in a string variable:
10 D$=”DSK1.PROGRAM”
20 CALL LINK(“RUN”,A$) (which XB cannot do)

As long as there is no break in the programs you can chain programs together this way and when the new program runs the screen, colors, character definitions, sprites, etc. will be preserved. But variables are *not* preserved. When the new program runs, prescan will reset all numeric variables to zero and string variables are to a null string. If you want to pass variables from one program to the next you have to store them with CALL LOAD and retrieve them with CALL PEEK, or save them to disk and then read them with the new program. Wouldn't it be nice if there was a way to preserve the variables and automatically pass them along to the new program? Well, there is.

CALL LINK(“RUNL1”,A$)

RUNL1 is used to run an XB256 program from a running program. It is similar to CALL LINK(“RUN”) but with a major difference. After it loads the program it runs line #1, *but without doing a prescan or resetting any of the variables of the calling program.* By chaining programs together with RUNL1 you can create a very long XB256 program, and the neat thing is that bypassing the prescan means super fast start up time and that all the variables are automatically carried over into the newly loaded program. You don't have to save them in one program and retrieve them in the next.

RUNL1 plays some tricks on the XB interpreter and to avoid problems, there are a couple of things that are important to observe.

1 – The initial program can be saved in either program format or IV254 format.
2 – A program being loaded with RUNL1 must be in IV254 format
3 – The chained programs must start with line 1
4 – A program being loaded with RUNL1 cannot be longer than the initial XB program at the beginning of the chain. A "Memory Full" error message will be issued if the chained program is longer than the original program. If it is in IV254 format you can use RUNL1 to load and return to the initial program.
5 – Because prescan is not performed, the chained programs cannot introduce new variables. They must all be contained in the first program. If the second program uses HCHAR, and X,Y,Z and A$ and they are not used in the first, the first program should include them this way:

```
10 GOTO 100::X,Y,Z,A$::CALL HCHAR
```

This causes XB to reserve space for the four variables and HCHAR. This line doesn't give a syntax error because XB never gets to the second part of the line.
6 – If the chained programs use DATA statements, they need to include RESTORE or RESTORE line-number so the program can find the DATA.
7 – DEF, SUB and SUBEND (named subprograms) can only be used in the initial program. GOSUB can be used in any of the programs.

Here is a short demonstration showing how to chain programs in XB256. This assumes you are running the game developers package from drive #1. First rename LOAD to LOADGDP so you don't overwrite it. Then type OLD DSK1.XB256 and change line 10 of the program:

```
10 CALL INIT :: CALL LOAD(8192,255,152):: CALL LINK("X"):: CALL LINK ("XB256"):: RUN "DSK1.DEMO1" :: END
```

Save this program as LOAD. This will autoload when you select Extended BASIC, load and activate XB256, then run the program DSK1.DEMO1

Now enter this program and save as DSK1.DEMO1:

```
100 ! 1st part of demo
110 DIM SR(100)
120 CALL CLEAR :: CALL SCREEN(4)
130CALL CHAR(96,"0103070B0A1B2A2B2A2B4A4B4A7B0E0A0080C0A0A0B0
A8A8A8A8A4A4A4BCE0A0")
140 CALL MAGNIFY(4):: CALL SPRITE(#1,96,5,100,120,-7,0)
150 FOR I=1 TO 100 :: SR(I)=SQR(I):: NEXT I
160 INPUT "ENTER A STRING: ":A$
170 RUN "DSK1.DEMO2"
180 CALL LINK("RUNL1","DSK1.DEMO2")
```

Save as DSK1.DEMO1, then enter this program:

```
100 ! 2nd part of demo
110 CALL CLEAR :: CALL SCREEN(12)
120 PRINT A$
130 FOR I=1 TO 100 :: PRINT SR(I):: NEXT I
140 GOTO 140
```

Save this with CALL LINK(“SAVEIV”,”DSK1.RUNL1DEMO2”)
If you want you can use SIZE to verify that DEMO1 is larger than DEMO2

Now quit and select Extended BASIC. A sprite appears, then some time elapses while the program generates 100 square roots and puts them in the array SR(). Then you are prompted to enter a string. When you press enter, line 170 loads and runs DEMO2. The sprite remains on the screen, but then the program breaks with BAD SUBSCRIPT IN 130. You can see that none of the variables were passed. The prescan set the string to “”, the variables to 0 and the default array size to 10.

Let's try changing DEMO1. Type OLD DSK1.DEMO1 then delete line 170 and then SAVE DSK1.DEMO1. Now quit and select XB. When the program comes to line 180, it will use RUNL1 to load DEMO2 and go to line 1. But XB has been duped; it has no way to know that another program has been loaded, so no prescan is done. This time A$ and all 100 square roots were retained in memory and passed to DEMO2 where they could be printed. All this happens automatically and with no delay for prescan or to retrieve the variables.

When you are done experimenting, don't forget to delete LOAD and change LOADGDP back to LOAD.

PACKAGING XB256 WITH AN XB PROGRAM

For development work, you generally would want to load XB256 into low memory as described above. Having the XB program and XB256 in distinct segments is the easiest way to develop programs. When the program is complete, if speed is not an issue you may want to run the XB program without compiling it. It would be nice to combine the program with XB256, making a nice, neat, one program package. XB256HM is a high memory version of XB256 that makes this easy.

Using XB256, develop your program as usual. (But do not use lines 1-3). When it is complete, follow these steps, which assume the Game Developer's Package is in drive 1:

1 – SAVE DSK1.PROGRAM
2 – SAVE DSK1.PROGRAM-M,MERGE (Or you can LIST “CLIP” with Classic99)
3 – CALL LINK(“OFF”) (Or you can quit and restart XB)
4 – OLD DSK1.XB256HM
5 – MERGE DSK1.PROGRAM-M (Or you can PASTE XB with Classic99)
6 – SAVE DSK1.PROGRAMHM

That's all there is to it. When it starts, the program turns on XB256, then runs the XB code.

Now for some details:

Because XB256 is combined with the XB program, the available program space is slightly reduced. There are 17314 bytes available instead of the usual 24488 bytes.

When XB256HM is active there is an interrupt routine running in high memory. If you load another XB program when this interrupt is active the computer will crash. To load another program, first turn off XB256 with CALL LINK(“OFF”) or else quit and restart.

The three lines of XB256HM XB code are:

```
1 !XB256 high memory version   by H. Wilhelm
2 CALL INIT :: CALL LOAD(8192,255,178):: CALL LINK("XB256"):: RUN 3
3 !@P-              (this has to be the third line!)
```

Line 3 turns off prescan which lets the program start up quickly without a time consuming prescan of the XB program. When CALL LINK(“XB256”) is performed, it turns on the XB256 interrupt routine and changes the third line of the program to enable prescan:

```
3 !@P+
```

Then RUN 3 starts the program again which initializes the stack and variables so that XB256 can work.

When XB256HM is activated, you can safely make changes to the XB program. But do not resequence it, as that can modify the embedded XB256 code. Be sure to change line 3 back to !@P- before saving. If you are making major changes it is best to return to the standard XB256 and make them there.