THE MISSING LINK
XB v2.8 G.E.M.

Bit mapped graphics for Extended BASIC

by Harry Wilhelm

08/27/20

The Missing Link 2.8

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to express my thanks to the following people, all of whom made significant contributions to *The Missing Link:*
Harry Brashear, John Wilforth, and Barry Traver all offered good suggestions for improving *The Missing Link.*
Ollie Hebert contributed vast amounts of time and energy. He tested the program, edited the manual, and wrote numerous demonstration programs.
My wife Donna was incredibly understanding about the many hours that were put in writing, debugging, and documenting the program.

If you like *The Missing Link,* it is in part due to the extra polishing made possible by these people.

Harry Wilhelm

INTRODUCTION

The Missing Link is a collection of assembly language subroutines that give the Extended BASIC programmer complete access to the bit-mapped graphics mode built into the TI-99/4A computer. These subroutines replace the usual methods of accessing the screen.

No knowledge of assembly language is required to fully utilize *The Missing Link.* Programs are written completely in Extended BASIC. This means that they are both easy to write and easy to understand. The average TI user can now take advantage of the bit mapped mode that is built into the TMS9918a video processor, but not available in XB.

The Missing Link includes text operations that can input information or display it on the screen. There is automatic word wrap when displaying text. Text can even be displayed vertically. The character size can be changed, permitting up to 32 rows by 60 columns on the screen. Different sized text can be displayed simultaneously on the same screen.

Cartesian graphics are available for plotting points, lines, circles, and boxes. Turtle graphics can be used with none of the ink and color restrictions found in LOGO. Sprite graphics can place up to 32 moving sprites on the screen. Best of all, there are no limits when combining graphics and text on the screen. A graphics screen dump is always available, and pictures can be loaded or saved in the standard TIArtist format.

The bank switched roms in XB 2.8 G.E.M. provide extra memory, making it possible to add 8 new assembly subroutines. Also, the code has been modified so that graphics are displayed noticeably faster.

EQUIPMENT REQUIRED

The Missing Link requires the TI-99/4A console, the Extended BASIC 2.8 G.E.M. cartridge, and the 32K memory expansion. A disk drive system is recommended. An Epson compatible printer is needed to use the screen dump feature.

DIFFERENCES FROM EXTENDED BASIC

The Missing Link will set the screen display to the standard graphics mode whenever an Extended BASIC program is not running. It will change to a bit-mapped screen whenever a program is running. This happens automatically, and requires no intervention by the user.

The bit-mapped screen is made up of pixels, which are the smallest dots that can be placed on the screen. There are 192 pixel rows and 240 pixel columns when using *The Missing Link*. Screen location 1,1 is at the upper left hand corner, and location 192,240 is at the lower right hand corner. The screen is less than 256 pixels wide because an 8 pixel wide strip on each side of the screen is used to hide the XB crunch buffer which cannot be relocated.

Two different color configurations can be used by *The Missing Link*:

The 16 color mode gives you access to all 16 colors that can be produced by the computer. The color data for the screen is in the form of 1 pixel high by 8 pixel wide strips. Each strip can have a foreground color and a background color. The eight pixels contained in each strip can be individually "turned on" or "turned off". In other words, they can be set to the foreground color or the background color for that strip. Different strips can have different foreground and background colors, but each strip can contain only two colors. Having the color data in the form of 1x8 strips is somewhat limiting; nevertheless, spectacular full color displays can be produced. This limitation is built into the TMS 9918A video chip. Unexpected colors in the display are almost always the result of this limitation.
The 2 color mode provides one foreground and one background color for the entire screen. The 2 color mode is the easiest to work with because there is more stack space and no problems with the 8 pixel long strips of colors.

The Bit-Mapped mode requires that the video memory be configured in a drastically different way than is normally used by Extended BASIC. This has several important consequences.

None of the usual methods of putting characters or graphics on the screen will work properly. PRINT, DISPLAY AT, INPUT, ACCEPT AT, HCHAR, VCHAR, GCHAR, SPRITE, and others can be used without an error resulting. However, nothing will appear except some garbled color patches in the upper third of the screen. *The Missing Link* provides assembly language subroutines to accomplish these same operations using bit-mapped graphics.

Except for those concerned with screen and sprite access, all other Extended BASIC program statements and subprograms will function normally.

The bit-mapped screen consumes a great deal of video memory. This memory has to come from somewhere. In this case it is obtained at the expense of the stack space. Fortunately, this is not as drastic as it first appears, because there are still the usual 24488 bytes of memory available for your program.

The stack space is primarily used to contain string data and subprogram names. You may have to adjust your programming style in order to conserve the limited stack space, especially when operating in the 16 color mode. Refer to page 18 of this manual for more information on how to conserve stack space. In the 16 color mode, the use of named subprograms with names more than eight characters long will cause spurious blocks of color to appear on the screen.

In the 16 color mode, INPUTing from a DISPLAY format disk file will cause spurious blocks of color to appear on the screen. You can avoid this by using LINPUT instead. (The use of INTERNAL format files will present no problem.)

Using TRACE to help with debugging will cause spurious blocks of color to appear on the screen. However, the line numbers become visible when you "break" the program with <Fctn 4>. After you "break" a running program by pressing <Fctn 4>, you can type CON to continue. The screen will reappear just as it was when the program was interrupted, but the colors will be set to black on cyan, and any sprites will be shown in the unmagnified size.

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

On vintage TI99 equipment, be sure the contacts on the XB module are clean. This will help avoid lockups.

LOADING *The Missing Link*

Press any key at the TI master title screen, then push 7 for *The Missing Link.*

The following appears on the screen:

```
CONFIGURE TML
COLORS? (1=16,2=2) 1
```

Normally you would want to use 16 colors, but if full color is not necessary, 3752 bytes of stack space are gained by using the "2 color" mode.

You are then have an option to set the number of disk files

```
DISK FILES (0-3) 1
```

This performs the equivalent of the CALL FILES(n) operation but the VDP buffers are configured differently.

Stack space is directly related to the number of disk files that are opened. Each disk file costs 518 bytes of stack space. Therefore, it is important to choose the smallest number of disk files that will still permit your particular application to run.

The message * Extended Basic v2.8 G.E.M. appears, followed by TML. This tells you that *The Missing Link* is loaded and active. The screen color and fonts will be either the default ones or your custom settings. The TML message is displayed at startup, when a program ends, or when it breaks because of Fctn 4 or an error. (The original TML would change the screen color to light green and the cursor to a Texas shape to tellyou that TML is loaded and active.)

Once *The Missing Link* is activated, the only way to change the number of disk files or the color mode is to type "BYE" to leave Extended BASIC and return to the master title screen. You can then reload the program and choose a different configuration.

Below is the stack space available for the 8 possible configurations:

	16 color mode	2 color mode
3 disk files	424 bytes	4176 bytes
2 disk files	942 bytes	4694 bytes
1 disk file	1460 bytes	5212 bytes
0 disk files	1978 bytes	5730 bytes

USING THE SUBROUTINES

The Missing Link with XB 2.8 G.E.M. contains 40 assembly language subroutines, which can be grouped in the following categories:

- FULL SCREEN OPERATIONS
- PEN COMMANDS
- WINDOWS
- TEXT
- CARTESIAN GRAPHICS
- TURTLE GRAPHICS
- SPRITE GRAPHICS
- PERIPHERAL ACCESS

All of these subroutines are intended to be called from within a running Extended BASIC program. No error message results when the subroutines are called from the immediate mode, but no action will occur on the screen.

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.

Unless explicitly stated otherwise, numbers and strings can be constants, variables, or elements of an array. Numeric values do not have to be integers. *The Missing Link* will automatically round them up or down.

FULL SCREEN OPERATIONS

CALL LINK("CLEAR")

This subroutine clears the entire screen by setting all pixels to the background color.

CALL LINK("COLOR",foreground-color,background-color)

This subroutine changes all the pixels on the screen to the specified color combination. Also, the PENHUE (described in the next section) is changed to the same color combination. The two color codes must be numbers from 1 to 16. Refer to page 99 in the Extended BASIC manual for a list of the colors. This subroutine has no effect on the screen color seen at the four edges of the display.

CALL SCREEN(color-code)

The screen color can be changed by using Extended BASIC's CALL SCREEN subprogram. Additionally, any pixel in the display that is set to transparent (color code 1) will appear in the screen color. Refer to page 165 in the Extended BASIC manual for more information.

PEN COMMANDS

The pen commands are used to control the status of the pen. By controlling the pen status, the programmer determines exactly what occurs when the pen touches any pixel while performing all subsequent graphics and text operations. (Text operations always assume that the pen is down, regardless of the actual pen condition.)

CALL LINK("PENHUE",foreground-color,background-color)

This routine changes the color of the pen to the specified color combination. The color codes must be numeric values from 1 to 16. The penhue then determines the foreground and background colors of the 8 pixel wide strip containing any pixel that is subsequently touched by the pen. A special case occurs when the penhue is set to transparent on transparent with CALL LINK("PENHUE",1,1). This color combination causes *The Missing Link* to not change the colors of any pixel touched by the pen. This is useful in circumstances where a previous operation has already set the colors of the pixels.

PENHUE is only functional in the 16 color mode.

CALL LINK("PD")

This routine sets the status of the pen to pen down. Any pixel subsequently contacted by the pen will be "turned on" - it will be set to the foreground color. The pen hue determines the foreground and background colors of the 8 pixel wide strip containing any pixel that is touched by the pen.

CALL LINK("PR")

This routine sets the status of the pen to pen reverse. Any pixel subsequently contacted by the pen will be inverted. Pixels that are "on" will be turned "off", and pixels that are "off" will be turned "on". The penhue determines the foreground and background colors of the 8 pixel wide strip containing any pixel that is touched by the pen.

CALL LINK("PE")

This routine sets the status of the pen to pen erase. Any pixel subsequently contacted by the pen will be "turned off" - it will be set to the background color. The penhue determines the foreground and background colors of the 8 pixel wide strip containing any pixel that is touched by the pen.

CALL LINK("PU")

This routine sets the status of the pen to pen up. The pen will "pass over" pixels without changing them. However, the foreground and background colors of each 8 pixel wide strip will still be determined by the penhue as if the pen had actually touched the pixel.

THE WINDOW

The "window" is the rectangular area on the screen that determines the boundaries where text and graphics can be displayed. Text will always be displayed within the window. Graphics may be displayed either inside or outside of the window. The window's boundaries have no influence on sprites. There is only one window and it is always active. However, since its location and size can be changed at will by your program, it is possible to place multiple windows on the same screen.

CALL LINK("WINDOW"[,row1,column1,row2,column2,1])

This routine is used to modify the location and size of the window. To specify that the window is to be the entire screen, simply omit all the optional values. TML automatically defaults to the full screen window when an XB program begins to run.

Row1 and column1 specify the location of the upper left hand corner of the window. Row2 and column2 specify the location of the lower right hand corner. The rows must be numeric values from 0 to 193 and the columns must be numeric values from 0 to 241. To place a frame around the window, include the optional trailing "1". To draw the frame correctly, the pen should have previously been set to pen down.

CALL LINK("REVWIN")

Normally, graphics can only be displayed within the boundaries of the window. Calling this routine reverses the window so that graphics will only be displayed outside the boundaries of the window. Calling this routine a second time restores the normal operation of the window. This routine has no effect when text is being displayed.

CALL LINK("FILL"[,row1,column1,row2,column2])

FILL is a very versatile routine. It causes the pen to touch each pixel within the specified rectangular area. By modifying the pen condition and the penhue, FILL can be used to erase selected areas, set text to inverse video, change colors, and so on.

If the optional values are omitted, this routine will fill the entire window area.

The optional row and column values can be used to specify the size of a smaller rectangle within the window. The row values must be from 1 to 192 and the column values must be from 1 to 240. These values are relative to the upper left hand corner of the window. For example, CALL LINK("FILL",2,3,10,11) will fill a rectangle inside the window starting at row 2, column 3 and ending at row 10, column 11. Row1 and column1 must fall within the window area or an error message will be issued. Row2 or column2 can fall outside of the window. In this case, the fill operation will only proceed as far as the window boundaries. No error message will be issued.

TEXT

A program can input numbers and strings up to 255 characters long. Also, numbers and strings can be displayed on the screen. TML does not restrict the display to 24 rows by 28 columns. Instead, there is pixel by pixel accuracy in placing characters, and there is complete control over the size of the characters. There is even a font size that lets text be printed with 32 rows by 60 columns. By using characters of this size you can place the same amount of text on the screen as is contained in a 24 by 80 column display. When using very small characters, the color combination of dark green on light green (13,4) gives the most legibility.

Only characters having ASCII codes from 32 to 127 can be printed or input. When operating in the 16 color mode, the current penhue determines the color of the characters. The pen condition is always ignored. When characters are displayed, the character is shown in the foreground color atop a rectangle in the background color.

Each character is placed on the screen by erasing a small rectangular block of pixels and then printing the character inside that block. The programmer can modify the width and height of this block of pixels. This is the character size and it directly determines the number of rows and columns that will fit on the screen. Different size characters can be displayed on the screen at the same time.

Characters are displayed sequentially from left to right and from top to bottom. When *The Missing Link* no longer has room for an entire character on the current line, it drops down a row, goes to the left hand boundary of the window, and continues.

Text and numbers are always printed or input inside a window, which gives the programmer total control over the screen appearance. By using a tall, narrow window it is possible to print text vertically.

When numbers are being displayed there is precise control over the numeric format used.

CALL LINK("PRINT",row,column,string-or-number[,string_variable])

Used to print a string or a number to the screen. A string can be up to 255 characters long. The row and the column are the pixel row and the pixel column relative to the upper left hand corner of the window. Numeric values are printed as specified by the FORMAT routine described below.

Word wrap is always on when using PRINT. There are three simple rules:
* Leading spaces are deleted so that the first character on a line will not be a space
* If a word will not fit in the remaining space on the line, then TML will fill the rest of the line with spaces, drop down a line, go to column 1 of the window, and continue printing. A word will only be broken if it is too long to fit totally within the left and right columns of the window.
* Trailing spaces at the end of a string are not deleted unless TML has to start a new line.

Sometimes a string or a number has too many characters to completely fit inside the window. In that case, it will be truncated upon reaching the lower right corner of the window. Including the optional string variable will retrieve the portion of the string that would not fit within the window. Your program can then deal with the string fragment as desired.

When printing a succession of strings to the screen, it sometimes is helpful to have a pointer to where TML left off printing. A text adventure is an example of an application where this would be useful. If the row and column are numeric variables, they will automatically be updated so that they point to the next available character position in the window. If you don't want to have these variables updated, simply enclose either of them in parentheses. No update can occur if the row or column are numeric constants.

CALL LINK("INPUT",row,column,string-or-numeric-variable[,length,prompt-string])

Used to input a number or a string up to 255 characters long from the screen. The row and the column are the pixel row and column relative to the upper left hand corner of the window. The routine assumes that

255 characters are to be input unless you specify an optional length ranging from 1 to 255 characters.

An optional prompt can be specified that will appear on the screen as a suggested response. The prompt must be a string, but it can be used when inputting either a string or a number. If the response is appropriate, the user can simply press <Enter>. Otherwise, the response can either be erased or modified as desired.

The routine first clears a space on the screen long enough to allow the specified number of characters to be input. If the window is too small to permit all the characters to be input, then the window boundaries determine the maximum number of characters. Then the optional prompt, if any, will be displayed, and finally the cursor will appear flashing atop the first character or space.

The keyboard functions are virtually identical to those used by Extended BASIC. <Fctn S> and <Fctn D> move the cursor left and right. <Fctn 1> deletes a character. <Fctn 2> is used to insert characters. <Fctn 3> erases from the cursor position to the end of the line. The keys will repeat if held down. Press <Enter> to input the string or number.

There is no equivalent to Extended BASIC's VALIDATE option. However, when inputting a numeric value, only the ten number keys and the "E + - ." keys are active. If no numbers are typed before pressing Enter, then the numeric variable will equal zero.

CALL LINK("CHAR",ASCII-code,hexadecimal-string)

Used to redefine character patterns. With a few differences, this is the equivalent of the CALL CHAR subprogram in Extended BASIC.

Only ASCII characters from 33 to 127 can be redefined. The space and the cursor cannot be redefined. The hexadecimal string that defines the character pattern can be up to 240 characters long. This means that up to 15 consecutive ASCII characters can be redefined each time this subroutine is called. When defining a sequence of character patterns, trying to define ASCII characters higher than 127 by using a long hexadecimal string will result in the excess string being ignored. No error message will be issued. If necessary, the computer will add zeros to the string so that its length is an even multiple of sixteen. Refer to pages 56-58 of the Extended BASIC manual for a more detailed description of how to redefine characters.

CALL LINK("CHSIZE",width,height)

Used to specify the size of each character in pixels. The width must be a numeric value from 1 to 8. The height must be a numeric value from 1 to 12. The character size will determine the number of rows and columns that can fit on the screen. Once the character size is set, TML automatically uses characters of that size when displaying or inputting data.

Character patterns are used starting at the upper left hand corner. If the character size is less than 8 x 8 the extra pixels at the bottom or right hand side of the pattern will be ignored. If the character size is greater than 8 pixels high then the extra pixels at the bottom will be blank.

CALL LINK("FORMAT"[,format-code,number1,number2])

Used to determine the format used when displaying a number on the screen.

If the format code is "0" or if no numbers are passed to the subroutine then numbers will subsequently be displayed in standard BASIC format.

If the format code is not zero then results will be similar to those obtained when using the IMAGE statement in Extended BASIC. The number being displayed will always occupy a predetermined amount of space on the screen. Number1 determines the number of characters to the left of the decimal point. Number2 specifies the number of characters to the right of the decimal point &plus the decimal point. Thus, adding number1 to number2 will determine how many columns are required to print a number.

When using scientific notation, add four characters if using a two digit exponent, or five characters if using a three digit exponent. If the number is too large, asterisks will be printed to identify the overflow condition. No error message will be issued.

The following format codes can be used:

0 - Standard Extended BASIC format.
1 - Positive numbers will have a space displayed instead of a plus sign. If the number is long enough, an extra digit can be displayed instead of the plus sign.
2 - Positive numbers will have a space displayed instead of a plus sign.
4 - Both positive and negative numbers will have their signs displayed.
8 - Scientific notation with a two digit exponent. Positive numbers will have a space displayed instead of a plus sign.
12 - Scientific notation with a two digit exponent. Both positive and negative numbers will have their signs displayed.
24 - Scientific notation with a three digit exponent. Positive numbers will have a space displayed instead of a plus sign.
28 - Scientific notation with a three digit exponent. Both positive and negative numbers will have their signs displayed.

CHANGING THE DEFAULT PRINTING ACTION

Several optional changes can be made to the default printing action by using the CALL LOAD subprogram. The following values may be useful:

CALL LOAD(10618,64,72) Blanks the background. This is the default action.
CALL LOAD(10618,16,0) Leaves the background unchanged.
CALL LOAD(10620,224) Sets the pen to "pen down." This is the default action.
CALL LOAD(10620,64) Sets the pen to "pen erase."
CALL LOAD(10620,40) Sets the pen to "pen reverse."
(The next two CALL LOADs are only effective in the 16 color mode.)
CALL LOAD(10560,2,129,24,0,22) The color of both the foreground and background pixels of the character pattern will be changed to the current penhue. This is the default action.
CALL LOAD(10560,16,0,209,92,19) Only the foreground pixels of the character pattern will be changed to the current penhue. The background pixels will remain unchanged.

CARTESIAN GRAPHICS

These routines let the programmer plot points, lines, circles, and boxes on the screen. If the program is operating in the 16 color mode, they can also be used to change the pen color. If the window size is smaller than the full screen, then the graphics will only be displayed inside the window. The REVWIN subroutine can be used to specify that the graphics will only be displayed outside the window. No problems will result from drawing either partly or totally off the edges of the screen.

In all cases, both of the optional penhue values must be included to have any effect. The pen position used when generating turtle graphics has no effect when plotting Cartesian graphics.

CALL LINK("PIXEL",row,column[,foreground-color,background-color])

This routine places a pixel on the screen. Including the optional color values will simultaneously change the penhue.

CALL LINK("LINE",row1,column1,row2,column2[,foreground-color,background-color])

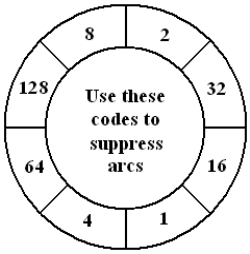
Draws a line between the points specified by row1,column1 and row2,column2. Including the optional color values will simultaneously change the penhue.

CALL LINK("BOX",row1,column1,row2,column2[,foreground-color,background-color])

Draws a rectangle between the points specified by row1,column1 and row2,column2. Including the optional color values will simultaneously change the penhue.

CALL LINK("CIRCLE",row,col,radius[,suppression-code,foreground-color,background-color])

Draws a circle of any radius with the center at the point specified by the row and column.



The circle is made up of eight arcs. Certain graphics applications may require that only part of the circle be displayed. Any combination of these eight segments can be blanked with the optional suppression code. This should be a number between 0 and 255. If the optional suppression code isn't passed, or if it is zero, then the entire circle will be displayed. Otherwise you can simply add up the numbers of the arcs you want to suppress and supply that number to the CIRCLE routine.

Including the optional color values will simultaneously change the penhue. Be sure to supply a zero for the suppression code if you are changing the penhue and want to display the entire circle.

TURTLE GRAPHICS

The Missing Link gives the programmer many of the turtle graphics commands available in the LOGO language with some improvements. Turtle graphics can now be in different colors, you can never run out of "ink", and the turtle has to travel 3 screen widths before it it will "wrap around" the screen.

When *The Missing Link* is loaded, the turtle is placed at row 96, column 120. The heading is 0 degrees, which is straight up. The turtle is always hidden from view.

CALL LINK("TURN",angle)

This routine is used to turn the turtle. The angle is a number specifying the degrees the turtle is to turn. If the angle is negative, the turtle is turned to the left (counterclockwise); if positive it is turned to the right (clockwise). Angles larger than +180 degrees or less than -180 degrees will "wrap around": i.e. +380 degrees will become +20 degrees.

CALL LINK("FWD",distance[,angle,1])

Used to move the turtle forward the specified distance in pixels. If the distance is negative the turtle will be moved backwards. The distance can be a number from -32767 to +32768. You can specify an optional angle, which causes the turtle to make a turn after completing its forward motion. Also, including a "1" for the optional third parameter will cause the turtle to return to the point it started from.

CALL LINK("PUTPEN",row,column[,angle])

Used to "pick up" the pen and move it to a specified location. The row should be a number from 0 to 192; the column can be from 0 to 240. If a zero is passed for either (or both) the row or column, the current value will not be changed.

Including the optional angle will cause the turtle heading to be set to the specified direction.

CALL LINK("GETPEN",row,column,angle)

Returns the current position of the turtle in the three numeric variables specified.

SPRITE GRAPHICS

The Missing Link can place up to 32 moving sprites on the screen at a time. Thirty-two different ASCII codes are available for sprite patterns. These ASCII codes are independent of the normal ASCII codes used when printing text. Operations can be performed simultaneously on consecutive numbered sprites, changing their locations, patterns, colors or motions at the same time.

CALL LINK("CHAR",ASCII-code,hexadecimal-string)

Used to define sprite patterns. With a few differences, this is the equivalent of the CALL CHAR subprogram in Extended BASIC. There are no default sprite patterns. Your program has to define the patterns of any sprites that are used.

Only the ASCII characters from 1 to 32 can be used as sprite patterns. Double size sprites use four successive patterns and must start at ASCII 1,5,9,13,17,21,25, or 29.

The hexadecimal string that defines the sprite pattern can be up to 240 characters long. This means that up to 15 consecutive ASCII characters can be redefined each time this subprogram is called. When defining a series of sprite patterns, trying to define ASCII characters higher than 32 with a long hexadecimal string will result in the excess string being ignored. This means you cannot define both sprite patterns and character patterns in the same CALL LINK("CHAR") operation.

CALL LINK("SPRITE",sprite-#,ASCII[,color,row,col,row-velocity,col-velocity])

Used to create sprites, set them in motion, or modify any of their attributes. Notice that the parameters used by this routine should be provided in the same order used by the CALL SPRITE subprogram normally used in Extended BASIC. However, the number sign (#) should not be placed before the sprite number.

If the sprite number is between 1 and 32 then only a single sprite will be created or modified. If the sprite number has three digits or more, then successive sprites will be created or modified simultaneously. For example, number 804 will simultaneously effect 8 sprites, starting with number 4. Number 2102 will effect 21 sprites starting with number 2. (Note that number 212 effects 2 sprites starting with number 12, not 21 sprites starting with 2.)

The ASCII, color, row, column, row velocity, and column velocity all operate as they do in Extended BASIC's SPRITE subprogram. Although most of the parameters are optional, the ASCII, color, row, and column must be provided when first creating a sprite for it to be visible.

Once a sprite has been created, any of its attributes can be modified independently of the others. If the list of sprite attributes contains less than six values, the attributes that were omitted from the list will not be changed. Also, with the exception of the velocities, providing a zero or a negative number for any attribute will result in no change to that attribute.

For example, CALL LINK("SPRITE",1,0,0,96,120) will move sprite #1 to the center of the screen, but will not change the pattern, color, or velocities. CALL LINK("SPRITE", 804,9,5) will effect 8 successive sprites, starting with #4. The eight sprites will now use ASCII 9 for a pattern, and will be a dark blue color. CALL LINK("SPRITE",10,0,0,0,0,10) will give sprite #10 a row velocity of 10 without effecting any of the other current attributes.

Thus, it will be seen that the CALL LINK("SPRITE") subroutine can perform all the operations provided by Extended BASIC's SPRITE, LOCATE, PATTERN, COLOR, and MOTION subprograms.

CALL LINK("DELSPR",sprite_number)

Used to delete either individual or consecutive sprites from the screen. The sprite number operates the same way as it does when creating sprites. See the description of CALL LINK("SPRITE") for more details on this. If the sprite number is zero then all the sprites will be deleted.

CALL LINK("FREEZE")

Used to "turn off" the automatic sprite motion for all the sprites. Their motion will stop even if motions have been assigned using the CALL LINK("SPRITE") subroutine.

CALL LINK("THAW")

Used to "turn on" the automatic sprite motion for all the sprites.

CALL LINK("SPRPOS",sprite-#,row,column)

Used to retrieve the location of a sprite. The sprite number must be a number from 1 to 32. The location of the upper left hand corner of the sprite will be returned in the numeric variables used for row and column. These must be numeric variables or an error message will be issued.

CALL LINK("DSTNCE",sprite-#,sprite-#,numeric-variable)
CALL LINK("DSTNCE",sprite-#,row,col,numeric-variable)

This functions identically to the CALL DISTANCE subprogram normally used in Extended BASIC, except that the number sign (#) should not precede the sprite numbers. Refer to page 80 of the Extended BASIC manual for more details.

CHECKING FOR SPRITE COINCIDENCES

The CALL COINC subprogram is the usual method for determining coincidences in Extended BASIC. For example, in the program line below, X will be -1 if sprite #1 is within 10 pixels of sprite #2. Otherwise, X will be 0.

```
10 CALL COINC(#1,#2,10,X):: IF X=-1 THEN 100 ! COINCIDENCE HAS
OCCURRED
```

But *The Missing Link* requires that the CALL LINK("DSTNCE") subroutine be used to determine coincidences.

In the program line below, X will be 100 or less if sprite #1 is within 10 pixels of sprite #2. Otherwise, X will be greater than 100.

```
10 CALL LINK("DSTNCE",1,2,X)::IF X<101 THEN 100 ! COINCIDENCE
HAS OCCURRED
```

CALL COINC(ALL,numeric-variable)

Extended BASIC's CALL COINC subprogram can be used in the above manner to determine if any two sprites are in actual contact with each other. This is the only way *The Missing Link* can use the CALL COINC subprogram. Refer to page 64 of the Extended BASIC manual for more details.

CALL MAGNIFY(magnification-factor)

Extended BASIC's CALL MAGNIFY subprogram works in the normal manner.. Refer to page 118 of the Extended BASIC manual for more details.

SPRITE EARLY CLOCK

Normally the sprite early clock is off, which causes sprites to fade in or fade out on the right hand side of the screen. When the early clock is on, sprites are shifted 32 pixels to the left of the position assigned by the column value. This makes them fade in or fade out on the left hand side of the screen. When creating a sprite you can turn on the early clock by adding 128 to the color value. To make this possible, the limit checks for sprite color have been modified so that values from 1 to 144 can be used. (Be careful not to use colors from 17 to 127.)

```
100 !Sprite early clock demo
110 CALL LINK("CHAR",1,RPT$(
"F",64)):: CALL MAGNIFY(4)
130 CALL LINK("FREEZE")
140 CALL LINK("SPRITE",1,1,1
6,92,120,0,6)!Sprite #1,ASCI
I 1,white with early clock o
ff,row 92,col 120
150 CALL LINK("SPRITE",2,1,1
4+128,124,120,0,6)!Sprite #2
,ASCII 1,magenta with early
clock on,row 124,col 120
160 CALL LINK("THAW")
170 GOTO 170
```

PERIPHERAL ACCESS

The Missing Link can load and save pictures in standard TI-Artist format. A single density screen dump can be obtained at any time by calling a subroutine from within a program, or else by simply pressing the <FCTN> and <CTRL> keys at the same time.

When saving or loading pictures, there must be at least one disk file available that has not been opened by your Extended BASIC program.

CALL LINK("LOADP",device-name[,1])

Used to load a screen from disk. The device name should be a string that specifies the disk number and the file name. An example of a valid device name is "DSK1.PICTURE".

If the file name is "DSK1.PICTURE", then *The Missing Link* will first search DSK1 for a file named "PICTURE_C". If that file is successfully found it will be loaded as the color data for the screen. Whether or not the color file is found, TML will then look for a file named "PICTURE_P". If that file is found it will be loaded as the picture data for the screen.

If TML is operating in the 2 color mode, then there will be no search for the color data, and the screen colors will not be changed. If TML is operating in the 16 color mode but fails to find the color data, the screen colors will be set to black on cyan. In either case, an I/O ERROR message will be issued if the picture data cannot be found.

When operating in the 16 color mode it is possible to suppress the search for the color data. Simply include the optional "1" after the device name to do so. The screen colors will remain unchanged when loading a picture in this manner.

CALL LINK("SAVEP",device_name[,1])

Used to save a screen to disk. The device name should be a string that specifies the disk number and the file name. An example of a valid device name is "DSK1.PICTURE".

If the file name is "DSK1.PICTURE", then TML will first save the color data to DSK1 in a file named "PICTURE_C", unless TML is operating in the 2 color mode. TML will then save the picture data to DSK1 in a file named "PICTURE_P". If TML is unable to save the files to disk then an I/O ERROR message will be issued. When operating in the 16 color mode, it is possible to suppress saving the color data. Simply include the optional "1" after the device name to do so.

Sprite data cannot be saved to disk.

CALL LINK("DUMP")

This subroutine is used to produce a single density graphics screen dump on Epson compatible printers. Pixels that are set to the foreground color will be black, while pixels set to the background color will be white. Sprites are not included in the screen dump. You can press <Fctn 4> to break out of the screen dump. However, that will also cause your Extended BASIC program to halt unless you have included the ON BREAK NEXT statement. See page 22 for information on how to configure *The Missing Link* so the screen dump codes match your particular printers requirements.

Another way of obtaining a screen dump is to press the <Fctn> and the <Ctrl> keys simultaneously. IMPORTANT: If you accidentally hit the <Fctn> and the <Ctrl> keys together and the printer is off, then your program will freeze as TML tries to print the picture. You can either turn on the printer or else press <Fctn 4>.

ADDITIONAL SUBROUTINES

Eight additional subroutines are available in XB 2.8 G.E.M.

The three routines to load/save fonts into TML for use by the bit mapped mode end in "A" to show they are used by the assembly routines and not XB 2.8. Earlier disk based versions of TML have a section showing how to use CHARDEF to define and save a font. These routines make that unnecessary.

CALL LINK("FONTA",fontnumber)

FONTA is used to load a font for the bit-mapped mode. Any of the 60 fonts contained in the XB 2.8 G.E.M. roms can be loaded. Patterns are loaded for ASCII 32 to 127. The number must be from 1 to 60 and can be a constant or a variable.

CALL LINK("LFONTA",Filename)

LFONTA is used to load a font from disk to be used by the bit mapped mode. Patterns are loaded for ASCII 32 to 127.

CALL LINK("SFONTA",Filename)

SFONT is used to save a font to disk that is used by the bit mapped mode. Characters from 32 to 127 are saved in memory image (binary) format.

The next five routines are part of the original TML, but if you wanted to use them it was necessary to embed them in your XB program in high memory due to memory limitations. That is not necessary with XB 2.8 G.E.M. because they are included in the cartridge.

CALL LINK("GETPAT",row,column,dec$)

Reads an 8x8 pixel block from the screen.from the screen. The upper left corner is at row, column. It is converted into a decimal string in TI ARTIST instance format and returned to XB for use by your program.

Additionally, the character definition for ASC 127 is redefined the same as the 8x8 pixel block at row, column. You can then print CHR$(127) in any desired screen location. You have to CALL LINK("CHSIZE",8,8) to get the full 8x8 pattern.

```
100 !GETPAT demo
110 CALL LOAD(8192,253,218)!
points to lookup routine. HM
LOADER provides this value.
120 CALL LINK("CIRCLE",4,4,3
)!put graphics on screen
```

```
130 CALL LINK("GETPAT",1,1,A
$)!get 8x8 pattern at row 1,
col 1
140 CALL LINK("PRINT",184,1,
A$)!print A$ which is in INS
TANCE format
150 CALL LINK("CHSIZE",8,8):
: CALL LINK("PRINT",100,180,
CHR$(127))!shows that ASC 12
7 was modified
160 GOTO 160
```

CALL LINK("GETPIX",row,column,x[,foreground,background])

This routine is used to determine the condition of the pixel at row, column. The third variable (x) will return a value of 1 if the pixel is "on" (set to the foreground color). It will return a value of (0) if the pixel is "off " (set to the background color.) Include the two optional variables if you want to retrieve the foreground and background colors of the specified pixel

CALL LINK("DECHEX",dec$[,hex$])

This routine is used when you have a decimal string in TI ARTIST instance format that you want to print on the screen. Your XB program must provide DEC$, (an instance format string) which redefines the pattern for ASC 127. Then your XB program can print CHR$(127) in any desired screen location. Including the optional second string variable will retrieve the equivalent character definition in hexadecimal format.

```
100 !DECHEX demo
110 CALL LOAD(8192,253,218)!
Pointer to lookup table. HM
LOADER provides this value.
120 DEC$="255,129,129,129,12
9,129,129,255" !INSTANCE for
mat decimal string
130 CALL LINK("DECHEX",DEC$,
HEX$)!takes DEC$ and modifie
s ASC 127; returns a hex str
ing
140 CALL LINK("CHSIZE",8,8):
: CALL LINK("PRINT",100,100,
CHR$(127))! shows that ASC 1
27 was modified by DEC$
150 GOTO 150
```

CALL LINK("GRAFIX")

Used to restore the screen to the normal XB graphics mode. Then all the usual XB program statements used for screen access will function normally. This routine is useful for a disk cataloger or for help screens because it leaves the bit mapped screen hidden but unchanged. N.B. The available stack space is not modified when you use this subroutine.

CALL LINK ("BITMAP")

Used to return the screen to the bit-mapped mode. You are returned to the same screen that was displayed when CALL LINK("GRAFIX") was used. Using GRAFIX garbles the bit mapped screen colors, so when returning to bit-mapped mode they are set to black on cyan, the normal start up colors for TML.

SAVING STACK SPACE

Bit-mapped graphics require a lot of video memory. This memory is obtained at the expense of stack space, which is especially limited when using the 16 color mode.

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 inside the 32K memory expansion. As before, there are 24488 bytes of free space available for a program. The only reduction is in stack space.

Extended BASIC uses the stack for a number of purposes. After the prescan, it contains a list of all the variable names that are 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$="FF8181818181FF" :: CALL LINK("CHAR",80,B$) !Uses more
stack space

10 A$="FFFFFFFFFFFFFFFF" :: CALL LINK("CHAR",40,A$)
12 A$="FF8181818181FF" :: CALL LINK("CHAR",80,A$) !Uses less
stack space
```

If possible, avoid string arrays, as even a small array is likely to cause a MEMORY FULL condition. Instead, keep 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
```

Despite your best efforts, if you write a very long program, it may be impossible to conserve enough string space when using the 16 color mode. If that happens, you will have little choice but to go to the 2 color mode.

There is an experimental method that enlarges the stack to 1984 bytes. After TML starts up enter CALL LOAD(-31888,31,255). This seems to be functional in classic 99. This works on the real TI but affects the sprites. There may be other side effects, so be sure to save any program before trying out this method.

CONVERTING PROGRAM FILES TO IV254 FILES

CALL SAVEIV(filename)

This is built into the XB 2.8 G.E.M. cartridge and is used to save a loaded program to disk in Internal, Variable 254 format. This format is automatically used by Extended BASIC when saving or loading programs that are larger than the available stack space. Because of *The Missing Link's* reduced stack space, all except very short Extended BASIC programs will be saved to disk in IV254 format. SAVEIV normally is not needed, but there may be circumstances where you might need to convert to IV254. For example, a program developed in the 2 color mode might not load in the 16 color mode without being converted.

USING "RUN" WITHIN A PROGRAM

You can RUN one XB program directly from another XB program. The problem is that the screen information used by the first program is carried over into the second program along with some screen garbage created by the RUN statement. Include CALL LOAD(14840,0) in the first line of the program to clear the screen and restore everything to the normal start up values.

TML FONTS/G.E.M. FONTS

Originally TML came with 5 fonts. Below are these fonts and the equivalents in XB 2.8 G.E.M.

TML	XB 2.8 G.E.M.
88FONT	FONT4
68FONT	FONT1
57FONT	FONT8
48FONT	FONT9
46FONT	FONT10

CONFIGURING THE SCREEN DUMP

One other change when starting TML would be to printer codes for the bit mapped screen dump, although this is only of interest ot users of a real TI99. (If your printer is the daisy wheel type, or doesn't support Epson graphics, then the screen dump will not work properly.) Because TML is stored in the cartridge ROMs there is no way to permanently change it, so the changes have to be done with CALL LOADs in an XB program.

You can adapt this short XB program and use it to modify TML to work with your printer.

```
1 ! Modify bit mapped screen dump in TML28
10 DN$="PIO.CR" :: DNAD=13791 :: L=LEN(DN$):: CALL LOAD(DNAD,L):: FOR
I=1 TO L :: CALL LOAD(DNAD+I,ASC(SEG$(DN$,I,1))):: NEXT I
20 CALL LOAD(DNAD+23,10,8,27,51,24,27,75,0,1,27,64,10)
```

In line 10, DN$ is the printer device name. This should be no more than 23 bytes long. Be sure to include .CR at the end.

In line 20, the first 9 bytes in the CALL LOAD are:

10,8 – linefeed, backspace.

27,51,24 – Escape, line spacing,24 (line spacing should be 24/216” or 8/72”)
For the the Epson FX80 this could be: 27,65,8 for 8/72” line spacing.

27,75,0,1 – Escape, single density graphics,256 bytes. (0+256*1)

27,64,10 – The last 3 bytes are used at the end of the dump routine to reset the printer and do a line feed: