

THE MISSING LINK 2.0

The Ultimate Extended BASIC Upgrade

by Harry Wilhelm

The Missing Link 2.0

TABLE OF CONTENTS

INTRODUCTION	-	-	-	-	-	-	3
Equipment required	-	-	-	-	-	-	3
Differences from Extended BASIC	-	-	-	-	-	-	3
Loading The Missing Link	-	-	-	-	-	-	5
USING THE SUBROUTINES	-	-	-	-	-	-	5
FULL SCREEN OPERATIONS	-	-	-	-	-	-	6
Clear screen (CLEAR)	-	-	-	-	-	-	6
Pixel colors (COLOR)	-	-	-	-	-	-	6
Screen color (SCREEN) from XB	-	-	-	-	-	-	6
PEN COMMANDS	-	-	-	-	-	-	6
Pen color (PENHUE)	-	-	-	-	-	-	6
Pen down (PD)	-	-	-	-	-	-	7
Pen reverse (PR)	-	-	-	-	-	-	7
Pen erase (PE)	-	-	-	-	-	-	7
Pen up (PU)	-	-	-	-	-	-	7
THE WINDOW	-	-	-	-	-	-	7
Window size (WINDOW)	-	-	-	-	-	-	7
Reverse window (REVWIN)	-	-	-	-	-	-	8
Fill window (FILL)	-	-	-	-	-	-	8
TEXT	-	-	-	-	-	-	8
Print on screen (PRINT)	-	-	-	-	-	-	9
Input from screen (INPUT)	-	-	-	-	-	-	9
Redefine character patterns (CHAR)	-	-	-	-	-	-	10
Specify character size (CHSIZE)	-	-	-	-	-	-	10
Format numeric output (FORMAT)	-	-	-	-	-	-	10
Change default printing action	-	-	-	-	-	-	11
CARTESIAN GRAPHICS	-	-	-	-	-	-	11
Place pixel on screen (PIXEL)	-	-	-	-	-	-	11
Draw line (LINE)	-	-	-	-	-	-	11
Draw box (BOX)	-	-	-	-	-	-	11
Draw circle (CIRCLE)	-	-	-	-	-	-	12
TURTLE GRAPHICS	-	-	-	-	-	-	12
Turn the turtle (TURN)	-	-	-	-	-	-	12
Move turtle forward (FWD)	-	-	-	-	-	-	12
Move pen (PUTPEN)	-	-	-	-	-	-	12
Get current pen position (GETPEN)	-	-	-	-	-	-	13
SPRITE GRAPHICS	-	-	-	-	-	-	13
Define sprite pattern (CHAR)	-	-	-	-	-	-	13
Create sprites (SPRITE)	-	-	-	-	-	-	13
Delete sprites (DELSPR)	-	-	-	-	-	-	14
Turn off automatic sprite motion (FREEZE)	-	-	-	-	-	-	14
Turn on automatic sprite motion (THAW)	-	-	-	-	-	-	14
Get sprite position (SPRPOS)	-	-	-	-	-	-	14
Sprite distance (DSTNCE)	-	-	-	-	-	-	14
Checking for sprite coincidences	-	-	-	-	-	-	14
Sprite coincidence (COINC) from XB	-	-	-	-	-	-	15
Sprite size (MAGNIFY) from XB	-	-	-	-	-	-	15

The Missing Link 2.0

PERIPHERAL ACCESS	-	-	-	-	-	15
Load TI Artist picture (LOADP)	-	-	-	-	-	15
Save TI Artist picture (SAVEP)	-	-	-	-	-	15
Screen dump (DUMP)	-	-	-	-	-	16
USING FONTS	-	-	-	-	-	16
Defining a font using CHARDEF	-	-	-	-	-	16
Loading a font	-	-	-	-	-	16
CONFIGURING "THE MISSING LINK"	-	-	-	-	-	17
SAVING STACK SPACE	-	-	-	-	-	18
CONVERTING PROGRAM FILES TO IV254 FILES	-	-	-	-	-	19
UNDOCUMENTED FEATURES OF THE MISSING LINK	-	-	-	-	-	21
Sprite early clock-	-	-	-	-	-	21
Numeric input bug	-	-	-	-	-	21
Printing extra large characters	-	-	-	-	-	21
Changing the default colors	-	-	-	-	-	22
Using "RUN" within a program	-	-	-	-	-	22
Using graphics mode with TML	-	-	-	-	-	22
String validation for TML	-	-	-	-	-	23
Modifying ("CLEAR")	-	-	-	-	-	24
Miscellaneous CALL LOADS	-	-	-	-	-	24
HIGH MEMORY ASSEMBLY CODE	-	-	-	-	-	26
ADDITIONAL TML SUBROUTINES	-	-	-	-	-	28
Get a character pattern from screen (GETPAT)	-	-	-	-	-	28
Get a pixel from screen (GETPIX)-	-	-	-	-	-	29
Define a character with decimal string (DECHEX)	-	-	-	-	-	29
Use graphics mode with TML (GRAFIX)	-	-	-	-	-	29
Return to bit-mapped mode (BITMAP)	-	-	-	-	-	29
INTERFACING WITH THE MISSING LINK	-	-	-	-	-	30

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

The Missing Link 2.0

INTRODUCTION

THE MISSING LINK is the ultimate extension for Extended BASIC. It enhances Extended BASIC to the same degree that Extended BASIC enhances console basic. THE MISSING LINK consists of 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 can be 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 advanced graphics features that were built into the computer, but never before available without delving into the complexities of assembly language.

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 TI-Artist format.

In short, THE MISSING LINK can unleash the power of your computer and make Extended BASIC programming fun again.

EQUIPMENT REQUIRED

THE MISSING LINK requires the TI-99/4A console, the Extended BASIC cartridge, the 32K memory expansion, and a disk drive system. 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.

There are two different color configurations that 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

The Missing Link 2.0

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.

THE MISSING LINK is very sensitive to unclean contacts on the Extended BASIC module. Lockups are invariably caused by dirty contacts, and cleaning them alleviates the problem.

The Missing Link 2.0

LOADING "THE MISSING LINK"

Select Extended BASIC from the TI master title screen. Place the program disk into a disk drive (drive number "n"), type RUN "DSKn.TML" and press <Enter>. If you don't press any key the program defaults to 1 disk file and 16 color mode. TML can be renamed LOAD for autoloading from drive 1.

Hold any key as the program begins to get different options. You will be asked if you are using a Myarc disk controller. Press "Y" or "N".

You are then be given the option of having from one to three disk files that can be opened simultaneously. When using a Myarc disk controller you may only have one or two files. Select a number and then press <Enter>. This performs the equivalent of the CALL FILES(n) operation.

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.

You are then given a choice of using either the "16 color" mode or the "2 color" mode. If full color is not necessary, 3752 bytes of stack space are gained by using the "2 color" mode.

After your selections are made, THE MISSING LINK is configured as instructed, and the computer is returned to you in the immediate mode of Extended BASIC. (Once in a great while, an error message in line 290 is issued. This appears to be a quirk of Extended BASIC and can be ignored.) The screen color will be changed to light green and the cursor will be shaped like Texas. This tells you that THE MISSING LINK 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.

Following is a list of the stack space available with the various configurations:

- 424 bytes. 16 color mode with 3 disk files available.
- 942 bytes. 16 color mode with 2 disk files available.
- 1460 bytes. 16 color mode with 1 disk file available.
- 4176 bytes. 2 color mode with 3 disk files available.
- 4694 bytes. 2 color mode with 2 disk files available.
- 5212 bytes. 2 color mode with 1 disk file available.

USING THE SUBROUTINES

THE MISSING LINK contains 32 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

The Missing Link 2.0

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 because 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

The Missing Link 2.0

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. THE MISSING LINK 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.

The Missing Link 2.0

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. THE MISSING LINK 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.

The Missing Link 2.0

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. The rules are simple:

- 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 THE MISSING LINK 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 THE MISSING LINK 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 THE MISSING LINK 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.

The Missing Link 2.0

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, THE MISSING LINK 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.

The Missing Link 2.0

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(11110,64,72) Blanks the background. This is the default action.

CALL LOAD(11110,16,0) Leaves the background unchanged.

CALL LOAD(11112,224) Sets the pen to "pen down." This is the default action.

CALL LOAD(11112,64) Sets the pen to "pen erase."

CALL LOAD(11112,40) Sets the pen to "pen reverse."

(The next two CALL LOADs are only effective in the 16 color mode.)

CALL LOAD(11080,2,129,24,0,22) 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(11080,16,0,209,92,19) 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])

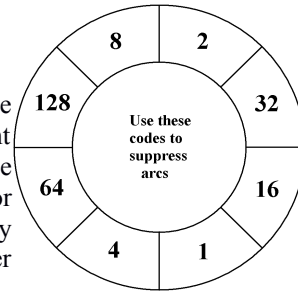
Draws a rectangle between the points specified by row1,column1 and row2,column2. Including the optional color values will simultaneously change the penhue.

The Missing Link 2.0

`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 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.

The Missing Link 2.0

`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.

The Missing Link 2.0

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.

The Missing Link 2.0

```
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.

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, THE MISSING LINK 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 THE MISSING LINK 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 THE MISSING LINK 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 THE MISSING LINK will first save the color data to DSK1 in a file named "PICTURE_C", unless THE MISSING LINK is operating in the 2 color mode. THE MISSING LINK will then save the picture data to DSK1 in a file named "PICTURE_P". If THE MISSING LINK is unable to save the files to disk then an I/O ERROR message will be issued.

The Missing Link 2.0

When operating in the 16 color mode, it is possible to suppress saving the color data. Simply include the optional "I" 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 THE MISSING LINK tries to print the picture. You can either turn on the printer or else press <Fctn 4>.

USING FONTS

DEFINING A FONT USING CHARDEF

An Extended BASIC utility program is included that makes it easy to modify existing fonts or create original fonts. The file name of the program is CHARDEF. THE MISSING LINK must not be loaded when using CHARDEF.

To run the program, select Extended BASIC from the master title screen. Then type RUN "DSKn.CHARDEF" and press <Enter>. CHARDEF will load and run.

The main menu gives you the choice of Editing characters, Saving the font, Loading a font, or Exiting. The on-screen prompts should be self explanatory. The main menu always shows the current font that is loaded, and this will be updated as you edit each character.

FonTS are saved and loaded in the standard INTERNAL, FIXED 241 format.

LOADING A FONT

THE MISSING LINK includes five different fonts that can be loaded by your Extended BASIC program. The file names of the five fonts are: 46FONT, 48FONT, 57FONT, 68FONT, and 88FONT.

These files are in INTERNAL, FIXED 241 format. Each file is eight sectors long and contains seven hexadecimal strings of 240 characters. These strings can be read from disk and then used with the CALL LINK("CHAR") subroutine to change the character definitions.

The following lines of Extended BASIC code demonstrate a subroutine that reads a font from disk and loads it into a program. This subroutine can be incorporated in your Extended BASIC programs if you need to load a font.

The Missing Link 2.0

```
10 A$="DSK1.68FONT"
20 GOSUB 100 ! go to sub tha
t loads font
25 CALL LINK("CHSIZE",6,8) !
redefine character size
30 program continues.....
40 .....
50 .....
100 OPEN #1:A$,VARIABLE 241,
INPUT ,INTERNAL :: FOR C=33
TO 128 STEP 15 :: INPUT #1:A
$ :: CALL LINK("CHAR",C,A$):
: NEXT C :: CLOSE #1 :: RETU
RN
```

The two numbers in the font name specify the smallest character size that can be used with that font. Use of a larger character size will result in the same size characters being printed, but spaced further apart. Using too small a character size will result in partial characters being printed with no spaces between them. You usually will want to change the character size when loading a font, as demonstrated in line 25 above.

CONFIGURING "THE MISSING LINK"

THE MISSING LINK can be configured to match the specifications of your printer. This should be done if it is found that the graphics screen dump routine doesn't perform properly. If your printer is of the daisy wheel type, or if it doesn't support Epson graphics, the screen dump can not be made to work correctly.

Follow this procedure to configure THE MISSING LINK. From the master title screen, select Extended BASIC. Then place THE MISSING LINK disk in disk drive #1 and do the following steps:

```
OLD DSK1.TML<Enter>
MERGE DSK1.CONFIG<Enter>
RUN<Enter>
```

Follow the on screen prompts as they appear. You will be asked for a printer device name. This can be "PIO" or "RS232" but it has to end in "CR". Then you will be asked for some special printer codes. You will have to look in your printer's instruction manual to find out this information. At one point you will be asked for "PRINTER CODES FOR SINGLE DENSITY GRAPHICS." Three codes are provided. If your printer only requires two codes, make sure the first code is a backspace (08), then change just the last two codes to match your printer's requirements. Finally, you will be asked for the code to reset the printer. If your printer doesn't have such a code, just press <Enter>.

After setting the printer specifications, you will be given the option of loading a font, which then becomes the default font. To load the font properly, it must be in the standard INTERNAL, FIXED 241 format. All the fonts on the program disk are in this format. Simply follow the instructions that appear on the screen.

Finally, you will be given instructions for saving your customized version of THE MISSING LINK.

The Missing Link 2.0

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$="FF818181818181FF" :: CALL LINK("CHAR",80,B$) !Uses more
stack space
```

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

The Missing Link 2.0

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
```

Loading a font requires a lot of memory, which can cause a MEMORY FULL condition. If this happens, and if your program only needs to have one font loaded, try loading the font ahead of time. This can be done when you configure THE MISSING LINK, or else by using a short program to load the font that then RUNs the main program.

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

Internal, Variable 254 is the format used by Extended BASIC when saving or loading programs that are larger than the available stack space. Because of this, one of the side effects caused by THE MISSING LINK's reduced stack space is that all except very short Extended BASIC programs must be saved to disk in INTERNAL, VARIABLE 254 format.

When THE MISSING LINK has been loaded and you are writing a program, Extended BASIC will automatically save it in IV254 format when necessary. However, there may be times when you want to load a program that was previously saved in PROGRAM format. If the program is too large, Extended BASIC will issue the error message: I/O ERROR 02.

A utility has been included with THE MISSING LINK package that makes it possible to convert already existing PROGRAM files into IV254 format. To convert from one format to the other:

The Missing Link 2.0

- Select Extended BASIC from the master title screen, then do the following:
- Place THE MISSING LINK disk in a disk drive. (drive number "n")
- CALL INIT :: CALL LOAD("DSKn.CONVERT") <Enter>
- CALL LINK("PROG") <Enter>
- Place the disk containing the program format file in drive #n
- OLD DSKn.PROGRAMNAME <Enter> - This will load the PROGRAM format file.
- CALL LINK("IV254") <Enter>
- SAVE DSKn.PROGRAMNAME <Enter> - This will save the program in IV254 format.

Repeat the process as many times as desired, starting with CALL LINK("PROG"). When finished using the CONVERT utility, always return to the master title screen.

The Missing Link 2.0

UNDOCUMENTED FEATURES OF "THE MISSING LINK"

Most of the following sections describe CALL LOADs used to modify the operation of THE MISSING LINK. If you want the changes to be in effect every time you run TML then add the CALL LOADs to the TML loader between lines 260 and 270. If the modifications are used by a specific XB program then they should be added to that program before the CALL LINKs that they modify. In this case, when exiting the XB program it would be prudent to either restore the default values or else to reload an unmodified TML. It is a good idea to add a REMark after one of these CALL LOADs so that in the future you'll know what it does. Finally, if you make an error when typing in these CALL LOADs it will probably crash the computer, so be sure to SAVE the program you're working on before trying it out for the first time.

SPRITE EARLY CLOCK

The default condition is for the sprite early clock to be off. This causes sprites to fade in or fade out on the right hand side of the screen. The early clock can be turned on. Doing so will shift sprites 32 pixels to the left of the position assigned by the column value. This will make them fade in or fade out on the left hand side of the screen.

To make it possible to use the early clock, first CALL LOAD(13325,144). This alters the limit check for the color so that values up to 144 can be provided. Then when you create the sprite, add 128 to the color value. Use the normal color values to keep the early clock turned off.

```
100 !Sprite early clock demo
110 CALL LINK("CHAR",1,RPT$(
"F",64)): CALL MAGNIFY(4)
120 CALL LOAD(13325,144)!Cha
nge limit checks
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
```

NUMERIC INPUT BUG

This is not exactly a bug, but the routine can be improved. When entering numbers the ability to enter spaces can be useful. CALL LOAD(9782,32) will make this possible.

PRINTING EXTRA LARGE CHARACTERS

The limit check in the CHSIZE subprogram can be changed so that characters can be spaced more than 8 pixels wide. However, any graphics in the area of the screen where the text is being printed must be cleared in advance.

The Missing Link 2.0

The limit checks in CHSIZE can be modified with CALL LOADs to the following addresses:

11540 = minimum width (default 2)
11541 = maximum width (default 8)ca
11542 = minimum height (default 2)
11543 = maximum height (default 12)

```
100 ! demos how to print ch
aracters 16 pixels apart
110 CALL LOAD(11541,16)
120 CALL LINK("CHSIZE",16,12
)
130 CALL LINK("PRINT",1,1,RP
T$("THIS IS A TEST! ",10))
140 GOTO 140
```

CHANGING THE DEFAULT COLORS

The default screen color can be changed from cyan. Change the loader to include CALL LOAD(15001,15+COLOR). To alter the default pen colors change the loader to include CALL LOAD(15680,Foreground*16+ Background-17)

The command mode screen color can be changed from light green. Add CALL LOAD(15497,COLOR-1) to the loader.

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.

USING GRAPHICS MODE WITH TML

There may be times when it is handy to use the standard XB graphics mode within a TML program. One possible example would be to catalog a disk, then go back to your original screen.

The following program shows how to do this: (These CALL LOADs should only be used within a program!)

```
100 !Demo showing how to swi
tch from bit-mapped mode to
graphics mode and back again
110 FOR I=1 TO 100 STEP 4 ::
CALL LINK("CIRCLE",96,I+1,I
):: NEXT I
120 CALL GRAFIX :: PRINT "We
're Now In Graphics Mode": :
:"Press Any Key"
130 CALL KEY(0,K,S):: IF S=0
THEN 130
140 CALL BITMAP
150 GOTO 150
```

The Missing Link 2.0

```
160 SUB GRAFIX :: CALL PEEK(
8192,A,B):: CALL LOAD(15693,
154,"",8192,62,0,"",15872,4,
32,53,216,105,23,4,96,0,106)
170 CALL LOAD(14842,1,0,"",1
4840,0):: CALL LINK("A"):: C
ALL LOAD(8192,A,B,"",15693,1
46,"",-31806,0):: SUBEND

180 SUB BITMAP :: CALL LOAD(
-31806,80,"",14842,1,0,"",14
840,0):: SUBEND
```

STRING VALIDATION FOR TML

There wasn't enough room in memory to include a "validate" option like the one in standard XB. But TML can be tricked into implementing this feature. Following is a demonstration program containing the two subprograms VALID and UNVALID. The first acts like the VALIDATE option by forcing CALL LINK("INPUT") to recognize only those keys that you specify. The second restores normal keyboard activity. These CALL LOADs should only be used within a program.

Line 110 shows how to use the VALID subprogram. Before performing a CALL LINK("INPUT") you must include the following lines of code: CALL PEEK(-31956,A,B):: B\$="A STRING":: CALL VALID,A,B)

The reason for the PEEK is to find out where in the XB program the token code pointer is aimed. A few bytes after that is the validation string. VALID does the math and sets a pointer in TML to point to that string. If you need to use variable names other than A, B, and B\$ it is *vital* that they be the same length. For example: X, Y, and Z\$ would work, but not A1, B1, and B2\$. CALL VALID does not need to be used before every CALL LINK("INPUT"). You only need to use it again if you wish to change the validation string..

```
100 ! Demonstration of string
validation
110 CALL PEEK(-31956,A,B)::
B$="ABCD" :: CALL VALID(A,B)
:: CALL LINK("INPUT",1,1,W$,
20)
120 CALL LINK("INPUT",25,1,A
$,20)!ABCD validation still
in effect
130 CALL UNVALID :: CALL LIN
K("INPUT",50,1,W$,20)

140 SUB VALID(A,B):: AD=A*25
6+B-65530 :: CALL LOAD(9649,
0,"",9652,INT(AD/256),AD)::
SUBEND

150 SUB UNVALID :: CALL LOAD
(9649,6,"",9652,38,40):: SUB
END
```


The Missing Link 2.0

MODIFYING CALL LINK("CLEAR")

CALL LINK("CLEAR") can be modified in several ways. Instead of clearing the screen, it can be changed to fill the screen. (i.e. change all the pixels to the foreground color.) Modify CLEAR with CALL LOAD(15116,7,1). CALL LOAD(15116,4,193) restores the default values so that the routine can clear the screen normally.

CLEAR can be modified to quickly erase or fill only part of the screen. The advantage is that it is much faster than using the FILL operation.

Let's inspect two words contained in RAM. Normally CALL PEEK(15114,A,B) returns values of 32 and 0 and CALL PEEK(15120,C,D) returns values of 24 and 0. The assembly code essentially tells the computer to clear $C*256+D$ bytes starting at $A*256+B$. In other words, clear 6144 bytes starting at 8192. This clears the screen from top to bottom.

If $A*256+B$ is greater than 8192 then some of the top of the screen will remain uncleared. Moreover, $C*256+D$ can be changed to clear less than 6144 bytes. This would cause part of the bottom of the screen to remain uncleared.

It is absolutely vital that $A*256+B$ be equal to or greater than 8192 and less than 14336. Furthermore, $A*256+B+C*256+D$ must be less than or equal to 14336.

This method works well when clearing the top, middle, or bottom third of the screen. Internally the screen is in a non-standard format, so other areas of the screen may not clear as expected. In such cases, experimentation will determine whether this method can work for you!

```
100 ! erase demo
110 CALL LOAD(15116,7,1):: C
ALL LINK("CLEAR")!modify 151
16 so that fill happens, the
n fill the screen
120 CALL LOAD(15116,4,193)!t
he default values - makes er
ase happen
130 CALL LOAD(15114,40,0,"",
15120,8,0):: CALL LINK("CLEA
R")!clear middle third of th
e screen
140 CALL LOAD(15114,32,0,"",
15120,24,0)!restore to defau
lt values so whole screen ge
ts cleared
150 GOTO 150
```

MISCELLANEOUS CALL LOADS

To suppress leading spaces at the beginning of a line:
CALL LOAD(11324,27) (default)

To allow leading spaces at the beginning of a line:
CALL LOAD(11324,16)

The Missing Link 2.0

To enable word wrap:
CALL LOAD(11338,20) (default)

To disable word wrap:
CALL LOAD(11338,16)

To automatically enter text when cursor reaches right margin of screen or window:
CALL LOAD(9794,23)

To automatically enter text when cursor is at the right margin of the bottom line:
CALL LOAD(9794,22)

To disable automatic entry of text:
CALL LOAD(9794,16) (default)

To determine if 16 or 2 color mode was selected:
CALL PEEK(14918,X)
X=24 means 16 color mode
X=9 means 2 color mode

To determine number of disk files:
CALL PEEK(-31888,X)
X=29 means 1 file
X=27 means 2 files
X=25 means 3 files

To change speed of cursor flash:
CALL LOAD(9578,A,B) (1,128 are default - actual number is $1*256+128$)

To change speed of auto repeat:
CALL LOAD(9584,A,B) (0,20 are default - actual number is $0*256+20$)

To disable auto repeat:
CALL LOAD(9624,16)

To enable auto repeat:
CALL LOAD(9624,17) (default)

To make any of these CALL LOADs a permanent part of TML enter the following lines:

- OLD DSKn.TML
- CALL INIT
- CALL LOAD(8282,4,96,226,232)
- CALL LINK("HILOW")

Now TML is in low memory. Make any desired changes with the above CALL LOADs, then:

- CALL LINK("LOWHI")
- SAVE DSKn.TML

Be sure to save to a different disk in case you made a mistake!

Flags – Memory addresses 16126 and 16127 are available for flags.

HIGH MEMORY ASSEMBLY CODE

Ask any knowledgeable 99er how Extended BASIC partitions the 32K memory expansion. They will answer: "The 8K low memory holds any assembly language subroutines, and the 24K of high memory holds the XB program." This statement is understandable, since it agrees with all the available reference material from TI. But it just ain't so - it's an easy task to trick XB's assembly language loader into loading the code into high memory where it can be embedded within an XB program. Then the A/L code can be run directly out of high memory without having to move it to other memory locations.

To understand the advantages and disadvantages of this new method, let's first describe how the other loaders such as SYSTEX and ALSAVE work. These loaders save assembly language subroutines that have been loaded into low memory. They do this by copying an exact byte by byte "snapshot" into high memory where it is "embedded" within a short XB program. It can then be saved to disk in the standard XB manner. This works because Extended BASIC assumes that anything in high memory must be XB code. When loading the routines, this XB program is read from disk into the normal high memory locations. Then the A/L code is almost instantly copied back into the low memory addresses it came from. This type of loader is ideally suited for loading XB extensions such as THE MISSING LINK, EZ-KEYS, or the various 40 column utilities. Once the A/L code has been safely moved into low memory, the entire 24K of high memory is available and you can NEW, LOAD, RUN or write XB programs at will without disturbing the contents of low memory.

A slightly different use for this type of loader occurs when an XB program requires a few specific A/L support routines. Normally CALL LOAD is used to load the routines directly from disk into low memory, but this can be very S L O W ! You can embed your A/L subroutines into the ALSAVE loader and then add an XB program to the loader. When the program RUNs, the A/L subroutines get copied into low memory, then the XB program takes over, using the A/L subroutines as needed. This method works perfectly, but I object to it on general principle. The problem is that there are two copies of the A/L subroutines taking up space in memory, only one of which is functional. The copy in high memory simply sits there, embedded in the XB code but doing nothing. Wasted memory is no small matter on a machine with only 32K. But it works, so this complaint is mainly on aesthetic grounds - or is it?

Suppose you want to add a few assembly routines to a utility such as THE MISSING LINK, which uses the entire low memory for its routines? Suppose your XB program needs more than 8K of support routines? Suppose you want to run large (up to 24222 bytes long) A/L programs out of XB and just use XB to load them and start them up? The answer is simple: YOU CAN'T DO IT WITH THE LOWMEMORY LOADERS!!

To deal with these problems I wrote a high memory loader program called HMLOADER, which RUNs in the standard XB environment. To use it simply type: RUN "DSKx.HMLOADER"

When HMLOADER runs it prompts you for a list of file names for the A/L subprograms you want to load. The list can have any number of file names. After entering the last file name you want loaded, press "ENTER" to start loading. The subprograms will be loaded twice. The first time simply finds out how big they actually are so the right amount of memory can be reserved for them. The second time loads them at the top of high-memory. Then a couple of very important things are done. A separate subprogram look-up routine is poked into high-memory; then a single line (line #3) of XB code is written that will poke the address of this routine into 8192.

If your XB program will not use low memory assembly routines such as TML or EZKEYS then you should add CALL INIT to line #3 before the CALL LOAD. Otherwise when the program runs you will probably get an error message in line #3. Keep in mind that CALL INIT erases any A/L code in low memory, so don't do this if additional assembly routines will be located in low memory. Regardless of whether you

The Missing Link 2.0

added the CALL INIT, you should SAVE DSKx.HMCODENAME before going any further.

If you're using assembly routines in low memory such as THE MISSING LINK or EZKEYS you should load them at this time. Then OLD DSKx.HMCODENAME will reload the high memory code.

At this point the assembly language code has been embedded and you can add lines of XB code from the keyboard or else from disk by using MERGE. The CALL LOAD(8192,NUMBER1,NUMBER2) does not have to be in line #3. If you move it, be sure it executes before any CALL LINKs to the high memory code. Do not type NEW and do not load a program using OLD DSKx.PROGNAME because either of these will overwrite the assembly code that HMLOADER just embedded! SAVE DSKx.PROGNAME will save both your XB code and the embedded A/L routines. After the program is saved you can LOAD DSKx.PROGNAME or RUN "DSKx.PROGNAME" and your A/L files will be loaded along with the XB program. With embedded code you don't need to load the A/L code separately.

RESequencing can lead to problems because the XB interpreter tries to resequence the embedded A/L code! The A/L code should be removed before RESequencing. Follow these steps if you need to RESequence:

SAVE DSKx.PROGNAME,MERGE	\
NEW	Gets rid of A/L code
MERGE DSKx.PROGNAME	/
RES	
SAVE DSKx.PROGNAME,MERGE	
OLD DSKx.HMCODENAME	Previously saved program created by HMLOADER
MERGE DSKx.PROGNAME	

When the XB code has been added you can RUN the program in the usual way. When the XB program encounters a CALL LINK, it first searches high memory for the subroutine. This assumes that line 3 has modified 8192 to point to the high memory look-up routine. If the subroutine cannot be found in high memory then the regular look-up routine will search for it in low memory. This means that if two subprograms have the same name, the one in high memory will be executed. As usual, if the subroutine cannot be found an error message is issued.

The source code should use no AORG's. Also, I've seen (and written) code that uses memory areas after the END statement. There are times when you can get away with this, but not with HMLOADER. If you need a buffer for whatever purpose, be sure to explicitly reserve it with BSS or some other means. Lastly, it's a good idea to CALL LOAD(8192,32,90) when the XB program ENDS. If you don't, and if you then load an XB program that doesn't have high memory A/L code, that program will crash if it does a CALL LINK. That's because the look-up table will be overwritten, but the pointer to it won't be erased!

The next time you want to use A/L subroutines within the XB environment why not give HMLOADER a try - I think you'll like it.

ADDITIONAL TML SUBROUTINES

Five additional assembly language subroutines have been written to supplement THE MISSING LINK. They are contained on the TMLEXTRAS disk in 3 different files:

TMLEXTRASS	Source code for the extra routines.
TMLEXTRASO	Object code for the extra routines.
TMLEXTRAXB	Assembly code embedded in an XB' program.

To avoid frustration, you should read the HMLOADER docs to understand more about how to use high memory assembly code.

The normal way to use the subroutines would be as follows:

OLD DSKn..TMLEXTRAXB

Then type in lines of XB code or else merge them from disk. Remember: don't type NEW or RES, and make sure that line 3 executes before any calls to the high memory routines.

TMLEXTRASO is the object code for the five extra routines. You would only use this file if you have additional assembly language routines you want to include. You would use HMLOADER for this operation.

THE SUBROUTINES

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

This routine reads from the screen an 8x8 pixel block whose 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. To get the full 8x8 pattern, be sure to CALL LINK("CHSIZE",8,8).

```
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
```

The Missing Link 2.0

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", or set to the foreground color. It will return a value of (0) if the pixel is "off", or set to the background color. Include the two optional variables to determine 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.

INTERFACING WITH "THE MISSING LINK"

This section contains information of use to assembly programmers who are writing programs to augment THE MISSING LINK. Because TML uses all of the low memory, programs to augment it must be loaded into high memory. A/L programs in high memory can share some of TML's utilities and workspaces.

GPBUF	EQU	>3DFE	Beginning of a 256 byte long general purpose buffer.
CHDEFS	EQU	>27F2	Beginning of character definitions. ASC 32 to 127. Each character needs 8 bytes.
GPLLNK	EQU	>35D8	
DSRLNK	EQU	>362E	

There are three workspaces available:

BASWS	EQU	>39BC	(used for BASIC subprograms)
BLWPWS	EQU	>39FC	(used by BLWP subroutines)
BLWPW1	EQU	>3A1C	(used by BLWP subroutines)

GETNUM EQU >3BAA Uses BLWPWS)

GETNUM is a BLWP type subroutine to retrieve numbers from XB, do a limit check, and print an error message if it is appropriate. The calling program should be set up this way:

```
BLWP @GETNUM
DATA >0002 How many numbers to retrieve. (In this case get 2)
DATA >010F Limit check for 1st number (Here permits >01 to >0F)
DATA >0000 Limit check for 2nd number (Zero bypasses limit check)
```

GETNUM will put the numbers into the registers of the calling program, starting with R4, and going up from there until all the numbers have been retrieved. One is subtracted from the value of the numbers returned. A value of >0000 for the limit check will bypass the limit check.

To use optional numbers with GETNUM do the following:

```
BLWP @GETNUM
DATA >0031 Retrieve 1 to 3 numbers - 1 number is mandatory; the
      rest optional
DATA >01FF 1st number from 1 to FF
DATA >0001 2nd number from 0 to 1
DATA >0308 3rd number from 3 to 8
```

If optional numbers are not passed a value of >FFFF is returned for them.

The Missing Link 2.0

GETSTR **EQU** >3C12 (Uses BLWPWS)

GETSTR is a BLWP type subroutine to fetch a string and put it into a buffer. The calling program should be set up this way:

```
BLWP    @GETSTR
DATA    >ABCD (A=pos.^in list; B=1/0 for optional/not optional;
        CD=max. length string)
DATA    BUFFER ADDRESS (Usually GPBUF at >3E00)
```

After BLWP @GETSTR R0 of calling workspace will be 0 if the string was successfully passed. An error message will be issued if the string couldn't be passed. If the string was optional and was not passed then R0 will be >FFFF.

RETNUM **EQU** >3532 (Uses BLWPWS)

RETNUM is a BLWP type subroutine to return a number to XB. Set up consists of putting the number into R3 of the calling workspace and the position in the list into R2; then BLWP @@RETNUM.

PRPIX **EQU** >2FBC(Uses BLWPWS)

PRPIX is a BLWP type subroutine to display a pixel on the screen. R4 of the calling program should have the ROW; R5 should have the COL.

PRLINE **EQU** >2F16(Uses BLWP1)

PRLINE is a BLWP type subroutine to print a line on the screen. R4 to R7 of the calling program should have ROW1,COL1,ROW2,COL2 of the line you want to print.

VDP MEMORY & PERIPHERAL ACCESS BLOCKS

VDP memory from >3C80 to >3DEE can be used for temporary storage, PABs and the like. Don't use this area for long term storage, as it may get overwritten. The block from >3DEF to >3FFF is used for disk access – trial and error will tell if it will work for you.