Text, Fonts, and Keyboard Input

At one point or another, almost every application will need to place text directly on the screen or get keyboard input from the user.

GEOS text output facilities support disk-loaded fonts, multiple point sizes, and additive style attributes. The application can use GEOS text routines to print individual characters, one at a time, or entire strings, including strings with embedded style changes and special cursor positioning codes. GEOS will automatically restrict character printing to margins allowing text to be confined within screen or window edges. GEOS even contains a routine for formatting and printing decimal integers.

GEOS keyboard input facilities the translation of keyboard input to text output by mapping most keypress so that they correspond to the printable characters within the GEOS ASCII character set. GEOS will buffer keypresses and use them to trigger **MainLoop** events, giving the application full control of keypresses as they arrive. And if desired, GEOS can also automate the process of character input, prompting the user for a complete line of text.

Text Basics

Fonts and Point Sizes

Fonts come in various shapes and sizes and usually bear monikers like *BSW 9*, *Humbolt 12*, and *Boalt 10*. A *font* is a complete set of characters of a particular size and typeface. In typesetting, the height of a character is measured in *points* (approximately 1/72 inch), so Humbolt 12 would be a 12 point (1/6 inch) Humbolt font. A text point in GEOS is similar to a typesetter's point: when printed to the screen, each GEOS point corresponds to one screen pixel. GEOS printer drivers map screen pixels to 1/80 inch dots on the paper to work best with 80 dot-per-inch printers.A GEOS 1/80 inch point is, therefore, very close to a typsetter's 1/72 inch point.

GEOS has one resident font, BSW 9 (Berkeley Softworks 9 point). The application can load as many additional fonts as memory will allow. Fonts require approximately one to three kilobytes of memory.

Proportional Fonts

Computer text fonts are typically *monospaced fonts*. The characters of a monospaced font are all the same width, compromising the appearance of the thinnest and widest characters. GEOS fonts are *proportional fonts*, fonts whose characters are of variable widths. Proportional fonts tend to look better than monospaced fonts because thinner characters occupy less space than wider characters; a lower-case "i," for example, is often less than 1/5th the width of an upper-case "W."

Character Width and Height

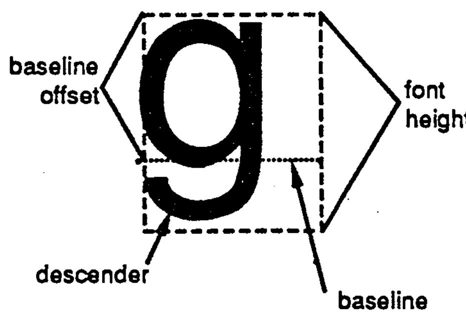
Although some characters are taller than others, all characters in a given font are treated as if they are the same height. This height is the font's point size. A 10 point font has a height of ten pixels. If a character's image is smaller than 10 pixels, it is because its definition includes white pixels at the top or bottom. The height of the current font is stored in the GEOS variable **curHeight**. Although fonts taller than 28 points are rare (some megafonts are as tall as 48 points), a font could theoretically be as tall as 255 points.

Because GEOS uses proportional fonts, the width of each character is determined by its pixel definition — the thinner characters occupy fewer pixels horizontally than the wider characters. Most character definitions include a few columns of white pixels on the right side so that the next character will print an appropriate distance to the right. If this space didn't exist, adjacent characters would appear crowded. The width of any single character cannot exceed 57 pixels after adding any style attributes, which means that the plaintext version of the character can be no wider than 54 pixels.

The Baseline

Each font has a *baseline*, an imaginary line that intersects the bottom half of its character images. The baseline is used to align the characters vertically and can be thought of as the line upon which characters rest. The baseline is specified by a relative pixel offset from the top of the characters (the *baseline offset*). Any portion of a character that falls below the baseline is called a *descender*. For example, an 18 point font might have a baseline offset of 15, which means that the 15th pixel row of the character would rest on the baseline. Any pixels in the 16th, 17th, or 18th row of the character's definition form part of a descender. The baseline offset for the current font is stored in the GEOS variable **baselineOffset**. The application may increment or decrement the value in this variable to print subscript or superscript characters

The following diagram illustrates the relationship between the baseline and the font height:



The y-position passed to GEOS printing routines usually refers to the position of the baseline, not the top of the character. Most of the character will appear above that position, with any descender appearing below. If it is necessary to print text relative to the top of the characters, a simple transformation can be used:

charYPos = graphicsYPos + baselineOffset

Where **graphicsYPos** is the true pixel position of the top of the characters, **charYPos** is the transformed position to pass to text routines, and **baselineOffset** is the value in the global variable of that name.

Styles

The basic character style of a font is called *plaintext*. Applying additional *style attributes* to the plaintext modifies the appearance of the characters. There are five available style attirbutes: reverse, italic, bold, outline, and underline. These styles may be mixed and matched in any combination, resulting in hybrids such as ***<u>bold italic underline</u>***. The current style attributes are stored in the variable **currentMode**. Whenever GEOS outputs a character, it first alters the image (in an internal buffer) based on the flags in this variable:

```
;
;       Action:         DecW2 decrements a word with inline code. No
;                       flags are set on reaching $0000. Destroys a.
;
;******************************************************************
.macro DecW2  addr
       lda     addr            ;get low byte
       bne     noOvrflw        ;if low_byte != $00, then skip high byte dec
       dec     addr+1          ;decrement hi-byte
noOvrflw:
       dec     addr            ;decrement low-byte
.endm
```

Most applications will use **IncW** and **DecW** to take advantage of the flags which are set when the values reach zero. However, **DecW2** can be useful when a word needs to be decremented quickly and the zero flag is not needed.

Unsigned Arithmetic

GEOS provides the following routines for arithmetic with unsigned numbers:

• **BBMult**	Byte-by-byte multiply: multiplies two unsigned byte operands to produce an unsigned word result.
• **BMult**	Word-by-byte multiply: multiplies an unsigned word and an unsigned byte to produce an unsigned word result.
• **DMult**	Word-by-word (double-precision) multiply: multiplies two unsigned words to produce an unsigned word result.
• **Ddiv**	Word-by-word (double-precision) division: divides one unsigned word by another to produce an unsigned word result.

Example:

```
;*****************************************************************************
; ConvToUnits
;
;       This routine converts a pixel measurement to inches or, optionally,
;       centimeters, at the rate of 80 pixels per inch or 31.5 pixels per
;       centimeter.
;
;       pass:
;               r0 - number to convert (in pixels)
;       return:
;               r0 - inches / centimeters
;               r1L - tenths of an inch / millimeters
;       affects:
;               nothing
;       destroys:
;               a, x, y, r0-r1, r8-r9
;
;*****************************************************************************

.if     AMERICAN                ;decide whether inches or centimeters is
                                ;appropriate
        INCHES = TRUE
.else   ;!AMERICAN
        INCHES = FALSE
.endif


ConvUnits:
```

```
                                        ; First, convert r0 to length in 1/20 of
                                        ;standard units

.if     INCHES                          ;*** START INCHES SPECIFIC CODE ***
                                        ; For ENGLISH, need to multiply by
                                        ;      20              1
                                        ;---------------- =  ---
                                        ; 80 dots/inch         4
                                        ;
                                        ; which amounts to a divide by four
        ldx     #r0                     ; ( /4 = two right shifts)
        ldy     #2                      ;
        jsr     DShiftRight             ; r0 = r0>>2   (r0 = r0/4)

.else   ;CENTIMETERS                    ;*** START OF CENTIMETER SPECIFIC CODE ***

                                        ; For centimeters, need to do multiply by
                                        ;      20                40
                                        ;  -------------- =  ------
                                        ;   31.5 dots/cm         63
                                        ;
;--- Following lines changed to save bytes
;       LoadW   r1,#40
;       ldx     #r0
;       ldy     #r1
;       jsr     DMult
        LoadB   r1,#40                  ; First multiply by 40
        ldx     #r0                     ; (word value)
        ldy     #r1                     ; (byte value)
        jsr     BMult                   ; r0 = r0*40 (byte by word multiply)
        LoadW   r1,#63                  ; then divide by 63
        ldx     #r0                     ;
        ldy     #r1                     ;
        jsr     Ddiv                    ; r0 = r0/63
.endif

                                        ;*** START OF COMMON CODE ***

                                        ; r0 = result in 1/20ths
        IncW    r0                      ; add in one more 1/20th, for rounding
        LoadW   r1,#20                  ; now divide by 20 (to move decimal over one)
        ldx     #r0                     ; dividend
        ldy     #r1                     ; divisor
        jsr     Ddiv                    ; r0 = r0/20 (r0 = result in proper unit)
        MoveB   r8L,r1L                 ; r1L = 1/20ths
        lsr     r1L                     ; and convert to 1/10ths (rounded)
        rts                             ; exit
```

Signed Arithmetic

GEOS provides the following routines for arithmetic with signed numbers:

• **Dabs**	Computes the absolute value of a two's-complement signed word.
• **Dnegate**	Negates a signed word by doing a two's complement sign-switch.
• **DSdiv**	Signed word-by-word (double-precision) division: divides one two's complement word by another to produce an signed word result.

There is no signed double-precision multiply routine in the GEOS Kernal. The following subroutine can be used to multiply two signed words together.

```
;*******************************************************************
```

```
;DSmult        double-precision signed multiply.
;
;pass: x - zpage address of multiplicand
;              y - zpage address of multiplier
;
;returns:      signed result in address pointed to by x
;              word pointed to by y is absolute-value of the
;                     multiplier passed
;              x, y unchanged
;
;Strategy:
;       Establish the sign of the result: if the signs of the
;       multiplicand and the multiplier are different, then the result
;       is negative; otherwise, the result is positive. Make both the
;       multiplicand and the multiplier positive, do unsigned
;       multiplication on those, then adjust the sign of the result
;       to reflect the signs of the original numbers.
;
;destroys:     a, r6 - r8                                  (mgl)
;*******************************************************************
DSmult:

       lda     zpage+1,x              ;get sign of multiplicand (hi-byte)
       eor     zpage+1,y              ;and compare with sign of multiplier
       php                            ;save the result for when we come back
       jsr     Dabs                   ;multiplicand = abs(multiplicand)
       stx     r6L                    ;save multiplicand index
       tya                            ;put multiplier index into x
       tax                            ;for call to Dabs
       jsr     Dabs                   ;multiplier = abs(multiplier)
       ldx     r6L                    ;restore multiplier index
       jsr     Dmult                  ;do multiplication as if unsigned
       plp                            ;get back sign of result
       bpl     90$                    ;ignore sign-change if result positive
       jsr     Dnegate                ;otherwise, make the result negative
90$:
       rts
```

Dividing by Zero

Division by zero is an undefined mathematical operation. The two GEOS division routines (**Ddiv** and **DSdiv**) *do not* check for a zero divisor and will end up returning incorrect results. It is easy to divide-by-zero error checking to these two routines:

Example:

```
;******************************************************************************************
;NewDdiv       -- Ddiv with divide-by-zero error checking
;NewDSdiv      -- DSdiv with divide-by-zero error checking
;
;Pass:         x      zp address of dividend
;              y      zp address of divisor
;
;Returns       x,y    unchanged
;              zp,x   result
;              r8     remainder
;              a      $00     -- no error
;                     %ff     -- divide by zero error
;              st     set to reflect error code in accumulator
;
;Destroys      r9
```

```
;
;*******************************************************************************************
DIVIDE_BY_ZERO        = $ff

NewDdiv:
        lda     zpage,y             ;get low byte of divisor
        ora     zpage,y             ;and high byte of divisor
        bne     10$                 ;if either is non-zero, go divide
        lda     #DIVIDE_BY_ZERO     ;return error
        rts                         ;exit
10$:                                ;
        jsr     Ddiv                ;divide
        lda     #$00                ;and return no error
        rts

NewDSdiv:
        lda     zpage,y             ;get low byte of divisor
        ora     zpage,y             ;and high byte of divisor
        bne     10$                 ;if either is non-zero, go divide
        lda     #DIVIDE_BY_ZERO     ;return error
        rts                         ;exit
10$:                                ;
        .if     Apple               ;save x-register because Apple destroys
                stx     Xsave       ;
        .endif                      ;
        jsr     DSdiv               ;divide
        .if     Apple               ;restore x-register because that ageos destroyed
                ldx     Xsave       ;
        .endif                      ;
        lda     #$00                ;and return no error
        rts                         ;

        .if     Apple               ;temp x register save variable for ageos
                .ramsect            ;
                Xsave   .block 1    ;
                .psect              ;
        .endif                      ;
```

Text, Fonts, and Keyboard Input

At one point or another, almost every application will need to place text directly on the screen or get keyboard input from the user.

GEOS text output facilities support disk-loaded fonts, multiple point sizes, and additive style attributes. The application can use GEOS text routines to print individual characters, one at a time, or entire strings, including strings with embedded style changes and special cursor positioning codes. GEOS will automatically restrict character printing to margins allowing text to be confined within screen or window edges. GEOS even contains a routine for formatting and printing decimal integers.

GEOS keyboard input facilities the translation of keyboard input to text output by mapping most keypress so that they correspond to the printable characters within the GEOS ASCII character set. GEOS will buffer keypresses and use them to trigger **MainLoop** events, giving the application full control of keypresses as they arrive. And if desired, GEOS can also automate the process of character input, prompting the user for a complete line of text.

Text Basics

Fonts and Point Sizes

Fonts come in various shapes and sizes and usually bear monikers like *BSW 9*, *Humbolt 12*, and *Boalt 10*. A *font* is a complete set of characters of a particular size and typeface. In typesetting, the height of a character is measured in *points* (approximately 1/72 inch), so Humbolt 12 would be a 12 point (1/6 inch) Humbolt font. A text point in GEOS is similar to a typesetter's point: when printed to the screen, each GEOS point corresponds to one screen pixel. GEOS printer drivers map screen pixels to 1/80 inch dots on the paper to work best with 80 dot-per-inch printers.A GEOS 1/80 inch point is, therefore, very close to a typsetter's 1/72 inch point.

GEOS has one resident font, BSW 9 (Berkeley Softworks 9 point). The application can load as many additional fonts as memory will allow. Fonts require approximately one to three kilobytes of memory.

Proportional Fonts

Computer text fonts are typically *monospaced fonts*. The characters of a monospaced font are all the same width, compromising the appearance of the thinnest and widest characters. GEOS fonts are *proportional fonts*, fonts whose characters are of variable widths. Proportional fonts tend to look better than monospaced fonts because thinner characters occupy less space than wider characters; a lower-case "i," for example, is often less than 1/5th the width of an upper-case "W."

Character Width and Height

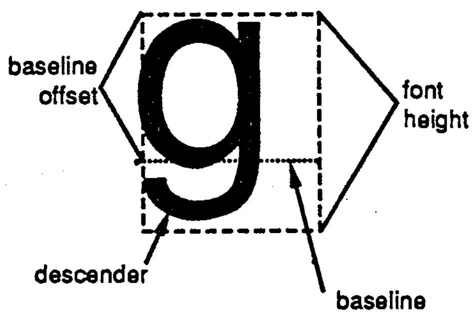
Although some characters are taller than others, all characters in a given font are treated as if they are the same height. This height is the font's point size. A 10 point font has a height of ten pixels. If a character's image is smaller than 10 pixels, it is because its definition includes white pixels at the top or bottom. The height of the current font is stored in the GEOS variable **curHeight**. Although fonts taller than 28 points are rare (some megafonts are as tall as 48 points), a font could theoretically be as tall as 255 points.

Because GEOS uses proportional fonts, the width of each character is determined by its pixel definition — the thinner characters occupy fewer pixels horizontally than the wider characters. Most character definitions include a few columns of white pixels on the right side so that the next character will print an appropriate distance to the right. If this space didn't exist, adjacent characters would appear crowded. The width of any single character cannot exceed 57 pixels after adding any style attributes, which means that the plaintext version of the character can be no wider than 54 pixels.

The Baseline

Each font has a *baseline*, an imaginary line that intersects the bottom half of its character images. The baseline is used to align the characters vertically and can be thought of as the line upon which characters rest. The baseline is specified by a relative pixel offset from the top of the characters (the *baseline offset*). Any portion of a character that falls below the baseline is called a *descender*. For example, an 18 point font might have a baseline offset of 15, which means that the 15th pixel row of the character would rest on the baseline. Any pixels in the 16th, 17th, or 18th row of the character's definition form part of a descender. The baseline offset for the current font is stored in the GEOS variable **baselineOffset**. The application may increment or decrement the value in this variable to print subscript or superscript characters

The following diagram illustrates the relationship between the baseline and the font height:



The y-position passed to GEOS printing routines usually refers to the position of the baseline, not the top of the character. Most of the character will appear above that position, with any descender appearing below. If it is necessary to print text relative to the top of the characters, a simple transformation can be used:

charYPos = graphicsYPos + baselineOffset

Where **graphicsYPos** is the true pixel position of the top of the characters, **charYPos** is the transformed position to pass to text routines, and **baselineOffset** is the value in the global variable of that name.

Styles

The basic character style of a font is called *plaintext*. Applying additional *style attributes* to the plaintext modifies the appearance of the characters. There are five available style attirbutes: reverse, italic, bold, outline, and underline. These styles may be mixed and matched in any combination, resulting in hybrids such as ***bold italic underline***. The current style attributes are stored in the variable **currentMode**. Whenever GEOS outputs a character, it first alters the image (in an internal buffer) based on the flags in this variable:

TITLE HERE.

7	6	5	4	3	2	1	0
b7	b6	b5	b4	b3	b2	b1	0

b7	underline: 1 = on; 0 = off.
b6	boldface: 1 = on; 0 = off.
b5	reverse: 1 = on; 0 = off.
b4	italic: 1 = on; 0 = off.
b3	outline: 1 = on; 0 = off.
b2†	superscript: 1 = on; 0 = off.
b1†	subscript: 1 = on; 0 = off.
b0	*unused.* FAST BLAST ON APPLE.

†Superscript and subscript characters are not supported by the standard text routines. However, geoWrite uses these bits in its ruler escapes. An application can print superscript and subscript by characters by changing the value in **baselineOffset** before printing: subtracting a constant will superscript the following characters and adding a constant will subscript the following characters. Additionally, some Apple GEOS printer drivers support these two bits when **SetMode** is used to format ASCII output.

Normally it is not necessary to modify the bits of **currentMode** directly. Special style codes can be embedded directly in text strings.

Style attributes temporarily modify the plaintext definition of the character and, in some cases, change the size and ultimate shape of the character:

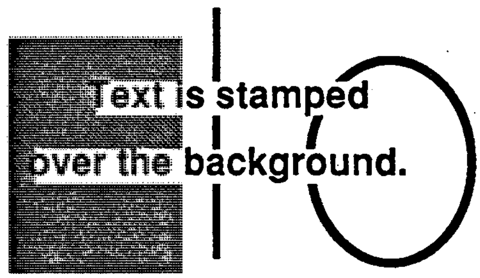
Underline	Inverts the pixels of the line *below* the baseline. The size of the character does not change.
Boldface	The character image is shifted onto itself by one pixel. The width of the character increases by one.
Outline	Transforms the character into an outline style. This transformation occurs after boldfacing and underlining. HEIGHT & WIDTH INCREASE BY 2.
Italic	Pairs of lines above the baseline are shifted right and pairs of lines below the baseline are shifted left. Thus the baseline is not changed, the two lines above it are shifted to the right one pixel, the next two are shifted four pixels from their original position, and so forth. The effect of this is to take the character rectangle and lean it into a parallelogram. The width is not actually changed. The same number of italicized characters will fit on a line as non-italicized characters, and because the shifting is consistent from character to character, adjacent italic characters will appear next to each other correctly. However, if a non-italic character immediately follows an italic character, the non-italic character will overwrite right side of the shifted italic character. This can be avoided by inserting an italicized space character.
Reverse	Reverses the pixel image of the character. This is the last transformation to take place. THE SIZE DOES NOT CHANGE.

10 pt

Note: Although, at this time, style attributes affect the printed size of a character in a predictable fashion, the application should not perform these calculations itself but use the GEOS **GetRealSize** routine to ensure compatibility with future versions of the operating system. For more information, refer to "Calculating ~~Character Widths~~" in this chapter.

How GEOS Prints Characters

When a character is printed, a rectangular area the width of the character and the height of the current font is stamped onto the background, leaving cleared pixels surrounding the character. When writing to a clear background, the cleared pixels around the character will mesh with the cleared background, leaving no trace. But when writing to a patterned background, the background will be overwritten:



There is no simple way to print to a non-cleared background without getting clear pixels surrounding the characters. Solutions usually involve accessing screen memory directly.

Text and dispBufferOn

Like graphics routines, most text routines use the special bits in **dispBufferOn** to direct printing to the foreground screen or the background buffer as necessary. For more information on using **dispBufferOn**, refer to "Display Buffering" in Chapter @GR@.

GEOS 128 Character X-position Doubling

GEOS 128 text routines pass character x-coordinates through **NormalizeX**, allowing automatic x-position doubling. (The character *width* is never doubled, only the x-position). Character x-position doubling is very much like graphic x-positions doubling and is explained in "GEOS 128 X-position and Bitmap Doubling" in Chapter @GR@. There is one notable difference: because **smallPutChar** will accept negative x-positions (allowing characters to be clipped at the left screen screen edge), the **DOUBLE_W** and **ADD1_W** constants should be bitwise exclusive-or'ed into the x-positions as opposed to merely bitwise or'ed. This will maintain the correct sign information with negative numbers.

Character Codes

Each character in GEOS is referenced by a single-byte code called a *character code*. GEOS character codes are based upon the ASCII character set, offering 128 possible characters (numbered 0-127). GEOS reserves the first 32 codes (0–31) as *escape codes*. Escape codes are non-printing characters that provide special functions, such as boldface enabling and text-cursor positioning. Character codes 32 through 126 represent the 95 basic ASCII characters, consisting of upper- and lower-case letters, numbers, and punctuation symbols. The 127th character is a special *deletion character*: a blank space as wide as the widest character, used internally for deleting and backspacing.

Most GEOS fonts do not offer characters for codes above 127 except in one special instance: the standard system character set (BSW 9) includes a 128th character that is a visual representation of the shortcut key (a Commodore symbol on Commodore computers and a filled Apple logo on

Apple computers). There is no inherent limitation in the text routines that would prevent an application from printing characters corresponding to codes 129 through 159, assuming the current font has image definitions for these character codes. The printing routines cannot handle character codes beyond 159, however. The text routines do no range-checking on character codes; do not try to print a character that does not exist in the current font.

A complete list of GEOS character codes appears in Appendix @TBL@.

Printing Single Characters

GEOS will print text at the string level or at the character level. The high-level string routines, where many characters are printed at once, will often provide all the text facilities an application evers need outside the environment of a dialog box. However, in return for generality, string-level routines sacrifice some of the flexibility offered by character level routines.Character level routines, where text is printed a character at a time, require the application to do some of the work: deciding which character to print next and where to place it. Because of this overhead, t is tempting to dispense with text at the character level, relying entirely on the string level routines instead. But the character level routines are the basic text output building blocks and the string level routines depend upon them greatly. For this reason, it helps to understand character output even when dealing entirely with string-level output.

GEOS provides two character-level routines that are available in all configurations of GEOS:

• **PutChar**	Process a single character code. Processes escape codes and only prints the character if it lies entirely within the left and right margins (**leftMargin, rightMargin**).
• **SmallPutChar**	Draw a single character. Does not check margins for proper placement. Does not handle escape codes. Prints partial characters, clipping at margin edges.

And one routine that only exists in Apple GEOS:

• **EraseCharacter**	Erase a character from the screen, accounting for the current font and style attributes.

PutChar is the basic character handling routine. It will attempt to print any character within the range 32 through 256 ($20 through $ff) as well as process any escape codes (character codes less than 32), such as style escapes. It will also check to make sure that the character image will fit entirely within the left and right margins. **SmallPutChar**, on the other hand, carries none of the overhead necessary for processing escape codes and checking margins; it is smaller (hence, the name) and faster but requires that the application send it appropriate data. Do not send escape codes to **SmallPutChar**.

Typically an application will call **PutChar** in a loop, using **SmallPutChar** to print a portion of a character that crosses a margin boundary. **SmallPutChar** can also be used by an application that does its own range-checking, thereby avoiding any redundancy. Be sure to only send **SmallPutChar** character codes for printable characters.

PutChar and Margin Faults

Prior to printing a character, **PutChar** checks two system variables, **leftMargin** and **rightMargin**. When an application is first run, these two margin variables default to the screen edges (0 and **SC_PIX_WIDTH**-1, respectively). If any part of the current character will fall outside one of these two margins, the character is not printed. Instead, GEOS **jsr**'s through **stringFaultVec** with the following parameters:

- **r11** Character x-position. If the character exceeded the right margin, then this is the position GEOS tried to place the offending character. If the character fell outside of the left margin, then the width of the offending character was added to the x-position, making this the position for the *next* character.
- **r1H** Character y-position.

Note:	When Apple GEOS vectors through **StringFaultVec**, the current values of **r11** and **r1H** are stored on the alternate zero-page. Do a **sta ALTZP_ON** before accessing them and a **sta ALTZP_OFF** after accessing them. When the string fault routine returns, **PutChar** will automatically copy these working registers over to the main zero-page.

stringFaultVec defaults to $0000. Because GEOS uses the conditional **jsr** mechanism, **CallRoutine**, a $0000 will cause character faults to be ignored.

There are many ways to handle a margin faults (including ignoring them entirely), . Faults on the left margin are usually ignored or not even bothered with because printing will usually begin predictably *at* the left margin, thereby precluding that type of fault. But faults on the right margin, (which are less predictable) will often get special handling, such as using **SmallPutChar** to output the fractional portion of the character that lies to the left of **rightMargin**.

There is one unfortunate problem with faults through **PutChar**: the fault routine has no direct way of knowing which character should be printed and so will lose some of its generality by needing access to data that should be local to the routine that calls **PutChar**. One simple way around this problem is to use a global variable — call it something like **lastChar** — to hold the character code of the character being printed, or perhaps, make it a pointer into memory (**PutString** does just that with **r0**). This way the fault routine will know which character caused the fault.

Example:

```
;****************************************************************
;macro: PutChar char          (char = character code)
;
;       Macro to replace jsr PutChar in your code so that lastChar
;       holds the value of the last character printed
;
;****************************************************************

.macro PutChar        character
       sta     lastChar              ;character is already in A-reg
       jsr     PutChar
.endm
```

Calculating the Size of a Character

Text formatting techniques such as right justification require the application to know the size of a character before it is printed. GEOS offers two routines for calculating the size of a character:

• **GetCharWidth**	Calculates the pixel width of a character as it exists in the font (in its plaintext form). Ignores any current style attributes.
• **GetRealSize**	Calculates the pixel height, width, and baseline offset for a character, accounting for any style attributes.

These routines can be used in succession to calculate the printed size of any character combination, whether groups of random characters, individual words, or complete sentences.

Partial Character Clipping

Confining text output to a window on the screen is called *clipping*. Characters that will appear outside the window's margins are not printed; they are "clipped," so to speak. Sometimes, however, it is desirable to print the portion of the offending character that lies within the margin and only clip the portion that lies outside the window area. This sort of clipping is called *partial character clipping*.

Top and Bottom Character Clipping

Both **PutChar** and **SmallPutChar** handle top and bottom partial character clipping. Any portion of a character that lies outside of the vertical range specified by **windowTop** and **windowBottom** will not be printed. **windowTop** and **windowBottom** default to the full screen dimensions (0 and **SC_PIX_HEIGHT–1**, respectively). They may be changed by the application before printing text.

Left and Right Character Clipping with SmallPutChar

Whenever a character crosses the left or right margin boundary, **PutChar** vectors through **StringFaultVec** without printing the character. **SmallPutChar**, unlike **PutChar**, will not generate string faults. If a character crosses a margin boundary, **SmallPutChar** will print the portion of the character that lies within the margin.

SmallPutChar will also accept small negative values as the character x-position, allowing characters to be clipped at the left screen edge by placing **leftMargin** at 0.

Note:	Clipping at the left margin, including negative x-position clipping, is not supported by early versions of GEOS 64 (earlier than version 1.4) — the entire character is clipped instead. Left margin clipping is supported on all other version of GEOS: GEOS 64 v1.4 and above, GEOS 128 (in both 64 and 128 mode), and Apple GEOS. Early versios of Apple GEOS (versions earlier than 2.0.3) did not properly clip at the left-margin.

Manual Character Clipping

Once of the criticisms of GEOS is the iconsistent and sometimes capricious character clipping capabilities — not all versions of GEOS fully support partial character clipping and the versions that do have inherent ideosyncracies. A carefully designed program can usually work around these limitations. Some applications, however, will need a reliable method to perform partial character clipping. The following **ClipChar** subroutine will properly clip and print a character that partially exceeds one of the left or right margins. Be aware that **ClipChar** does quite a bit of caculation and should only be used in special cases where controlled character clipping is needed.

Example:

```
.if (0)
*****************************************************************************
ClipChar        -- print a character, clipping to window margins.

Description:
        Draw a character, clipping it EXACTLY to leftMargin, rightMargin,
        windowTop and windowBottom

        Operates by temporarily modifying the font definition (making the
        character thinner, so as to fit in the margin).

Pass:
        a - character to print
        r11 - x position
        r1H - y position

Return:
        r11 - x position for next char
        r1H - y position for next char

Destroyed:
        a, x, y, r2-r10L
*****************************************************************************
.endif
ClipChar:
        sta     r1L                     ;store character
        ldx     currentMode             ;get width of character
        jsr     GetRealSize             ;
        dey                             ;use width - 1 to calc last position
        tya                             ;
        add     r11L                    ;r2 = last pixel that char covers
        sta     r2L                     ;
        lda     #0                      ;
        adc     r11H                    ;
        sta     r2H                     ;

        CmpW    r2,leftMargin           ;check for char entirely off window
        blt     3$                      ;if so then exit
        CmpW    rightMargin,r11         ;
        bge     5$                      ;
3$:
        AddWVW  r2,1,r11                ;r11 = one pixel beyond where char would have gone
        rts                             ;exit
5$:
        lda     r1L                     ;push old width table values
        sub     #32                     ;get card #
        sta     r3L                     ;
        asl     a                       ;
        tay                             ;
        ldx     #0                      ;
10$:
        lda     (curIndexTable),y       ;store this char's index values
        sta     savedWidths,x           ;
        iny                             ;
        inx                             ;
        cpx     #4                      ;
        bne     10$                     ;loop to copy values

        CmpW    leftMargin,r11          ;
        blt     30$                     ;
        lda     r3L                     ;
        asl     a                       ;
        tay                             ;
        lda     leftMargin              ;check for clipping on left
```

```
        sub     r11L                ;
        clc                         ;
        adc     (curIndexTable),y   ;
        sta     (curIndexTable),y   ;
        iny                         ;
        lda     #0                  ;
        adc     (curIndexTable),y   ;
        sta     (curIndexTable),y   ;
        MoveW   leftMargin,r11      ;
30$:
        CmpW    r2,rightMargin      ;
        blt     50$                 ;
        lda     r2L                 ;check for clipping on right
        sub     rightMargin         ;
        sta     r3H                 ;save amount to subtract
        lda     r3L                 ;
        asl     a                   ;
        tay                         ;
        iny                         ;
        iny                         ;
        lda     (curIndexTable),y   ;
        sub     r3H                 ;
        sta     (curIndexTable),y   ;
        iny                         ;
        lda     (curIndexTable),y   ;
        sbc     #0                  ;
        sta     (curIndexTable),y   ;
50$:
        lda     r1L                 ;draw the character !!
        pha                         ;save it for later
        jsr     SmallPutChar        ;
        pla                         ;
        sub     #32                 ;
        asl     a                   ;recover old widths
        tay                         ;
        ldx     #0                  ;
60$:
        lda     savedWidths,x       ;
        sta     (curIndexTable),y   ;
        iny                         ;
        inx                         ;
        cpx     #4                  ;
        bne     60$                 ;
        rts                         ;

        .ramsect
savedWidths:                        ;values from index tabel stored here
        .block  4
        .psect
```

Printing Decimal Integers (PutDecimal)

One of the unfortunate side-effects of binary math is the conversion necessary to print numbers in decimal. Fortunately, GEOS offers a routine to remove this drudgery from the application:

• **PutDecimal**	Format and print a 16-bit, positive integer

PutDecimal is like a combination of character and string level routines. The application passes it a single 16-bit, positive integer, some formatting codes (e.g., right justify, left justify, suppress leading zeros), and a printing position. **PutDecimal** converts the binary number into a series of one to five numeric characters and calls **PutChar** to output each one.

String Level Routines

Many applications will never need complex text output and can rely on GEOS's string-level routines for simple text output and input. GEOS provides two string-level text routines, one for printing strings to the screen and one for getting strings through the keyboard.

• PutString	Print a string to the screen.
• GetString	Get a string from the keyboard using a cursor prompt and echoing characters to the screen as they are typed.

GEOS Strings

A *GEOS string* is a null-terminated group of character codes. (*Null-terminated* means the end of the string is marked by a **NULL** character ($00).) These strings can contain alphanumeric characters as well as special escape codes for changing the style attributes or changing the printing position.

There is no basic limit to the possible length of a string; GEOS processes the string one character at a time until it encounters the **NULL**, which it interprets as the end of the string. If the string is not terminated, GEOS will have way of knowing where the end of the string is and will continue printing until it encounters a $00 in memory.

A simple string of ASCII characters might look like this:

```
String1:
        .byte   "This is a simple string.",NULL
```

The above string, including the **NULL**, is 25 characters long (and therefore 25 bytes long also). Escape codes may be embedded within the string to effect changes while printing. An individual word, for example, may be underlined by embedding an **ULINEON** escape code before the word and an **ULINEOFF** after it as in:

```
String2:
        .byte   "This word is "
        .byte   ULINEON, "underlined", ULINEOFF,".", NULL
```

The embedded escape codes change the style attribute bits in **currentMode** mid-string, resulting in something like:

This word is <u>underlined</u>.

PutString

PutString offers a simple way to handle text output. It is really does nothing more than call **PutChar** in a loop, so issues that apply to **PutChar**, such as top and bottom character clipping, also apply to **PutString**. **PutString** directly supports a feature that **PutChar** doesn't, though: multibyte escape codes, such as **GOTOXY**,which require **r0** to contain a pointer to the auxiliary bytes in a multibyte sequence (**PutString** maintains **r0** automatically, allowing the extra parameters to be embedded directly in the string). Printing a string to the screen with **PutString** involves specifying a position to begin printing and passing a pointer to a null-terminated string:

Example:

```
;*****************************************************************
;       Example use of PutString. Places a test string onto the
;       screen. Assumes that leftMargin, rightMargin, windowTop and
;       windowBottom contain their default, startup values (full
;       screen dimensions).
;*****************************************************************

        STR_X   = 40                    ;x-position of first character
        STR_Y   = 100                   ;y-position of character baseline

Print:
        LoadB   dispBuffOn,#(ST_WR_FORE | ST_WR_BACK)     ;both buffers!
        LoadW   r11,#STR_X              ;string x-postion
        LoadB   r1H,#STR_Y              ;string y-position
        LoadW   r0,#String              ;address of text string
        jsr     PutString               ;print the string
        rts                             ;exit

String:
        .byte   "This is a test.", NULL       ;null-terminated string
```

String Faults (Left or Right Margin Exceeded)

Because **PutString** calls **PutChar**, if any part of the current character will fall outside of **leftMargin** or **rightMargin**, the character is not printed. Instead, GEOS **jsr**'s through **stringFaultVec** with the following parameters:

r11 Character x-position. If the character exceeded the right margin, then this is the position GEOS tried to place the offending character. If the character fell outside of the left margin, then the width of the offending character was added to the x-position, making this the position for the *next* character.

r1H Character y-position.

r0 Pointer to the offending character in the string. *Only valid with PutString, unused by* ***Putchar***.

Note:	When Apple GEOS vectors through **StringFaultVec**, the current values of **r11**, and **r1H**, and **r0** are stored on the alternate zero-page. Do a **sta ALTZP_ON** before accessing them and a **sta ALTZP_OFF** after accessing them. When the string fault routine returns, **PutString** will automatically copy these working registers over to the main zero-page.

GEOS 64 and GEOS 128 do nothing special to handle these string faults. If the application has not installed its own string fault routine, **stringFaultVec** it should contain a default value of $0000, which will cause the string fault to be ignored. If this is the case, the following will happen:

- If part of the character was outside of the left margin, the width of the offending character was added to the x-position in **r11** before the fault. **PutString** moves on to the next character in the string and attempts to print it at this new position.

- If part of the character was inside the left margin but outside the right margin, **PutString** leaves the x-position unchanged and moves on to the next character in the string.

The strategy behind this system is to only print the portion of the string that lies entirely within the left and right margins. Unfortunately, this strategy is flawed. Whenever the right margin is encountered, **PutString** should stop completely. But it doesn't. It continues searching through the string, looking for a character that *will* fit. This can be a problem when a thin character follows a wide character. For example, trying to print the word "working" with only a few pixels of space before the right margin, **PrintString** would try to print the "w," but since it doesn't fit, would move on and try its luck with the following "o." But the "o" won't fit either, so it moves on until it encounters the "i," which just happens to fit in the available space. **PutString** proudly prints the "i," thinking it has done a good thing, entirely unaware that the proper sequence of characters has been lost.

The Apple GEOS version of **PutString** offers a partial solution to this problem. If **stringFaultVec** contains $0000, it installs a temporary string fault routine (**PutStringFault**). **PutStringFault** immediately terminates string printing on any fault (left or right margin) by moving **r0** forward to point to the null. To disable the Apple **PutStringFault** so that Apple GEOS **PutString** is identical to GEOS 64 and GEOS 128 **PutString**, point **StringFaultVec** to an **rts** prior to calling **PutString**. **PutStringFault** can be implemented on GEOS 64 and GEOS 128 by placing the following routine into **StringFaultVec** prior to calling **PutString**:

```
;*****************************************************************
;PutStrFault   (for GEOS 64/128 only)
;
;String fault routine for duplicating the Apple GEOS PutString
;fault handling on GEOS 64 and GEOS 128. Immediately terminates
;string printing when any fault (left or right margin) is
;generated by setting r0 to point at the end of the string.
;*****************************************************************

PutStrFault:

;Go through the string looking for the null
       ldy          #1              ;load index to look at next character
       bne          20$             ;always branch -- don't inc on 1st pass

20$:
       IncW         r0              ;check next character
10$:   lda          (r0),y ;get character
       bne          20$             ;loop until we find null

;Return to PutString pointing at a null
       rts
```

The above technique, however, has two flaws: if a character lies outside the left margin, printing is aborted, and, with either type of fault, the application has no way of knowing which character in the string caused the fault. The following routine, **SmartPutString**, will solve both these problems. If a character lies outside the left margin, it is skipped, and if it lies outside the right margin, **SmartPutString** returns with **r0** pointing to the character in the string that caused it to terminate. If **r0** points to a **NULL**, then **SmartPutString** was able to print the whole string and terminated normally.

```
.if 0
;*****************************************************************
;SmartPutString
```

```
;
;       New front-end to PutString that handles right-edge string
;       faults by exiting immediately rather than moving through
;       the string until it finds a character that fits. It operates
;       by replacing the current string fault service routine with
;       its own routine that tricks PutString into thinking it
;       encountered a null on a right-margin fault.
;
;       Pass:  same as PutString. The string must not be located
;                  .  in zero-page ($00-$ff).
;
;       Returns:       same as PutString, except that if the string
;                      faulted, then r15 points to the offending character
;                      rather than the null at the end of the string. If
;                      r15 = $0000, then the string printed without a fault
;
;       Destroys:      same as PutString.
;
;       Note:  No inline support.
;*******************************************************************
.endif

SmartPutString:

;Insert our own string fault routine into stringFaultVec,
;saving the old one so that we can restore it when we leave.
       PushW  stringFaultVec              ;save old
       LoadW  stringFaultVec,#FaultFix    ;install new

;Clear the flag that alerts us to a right-edge fault. If the high
;byte of r15 is zero then PutString returns immediately because
;our string never encountered the right edge.
       LoadW  r15,#0                      ;clear r15 to $0000

;Call PutString with our string fault routine in place
       jsr    PutString

90$:
;Restore the old string fault routine
       PopW   StringFaultVec

;Return
       rts

;*******************************************************************
;FaultFix
;
;Our own string fault routine that sets up the string pointer
;so that PutString is tricked into thinking it encountered a
;NULL when the right margin is exceeded.
;*******************************************************************

FaultFix:

;Check to see if we exceeded the right margin or if we just
;haven't reached the left margin yet. If the right margin was
;not exceeded, return early -- the text routine handles this
;case appropriately.If the first character in the string will not
;fit within either margin, we treat it as if the right margin was
;exceeded. Don't need to normalize the coordinates under GEOS 128
;because the character output routine has already taken care of
;this for us.

       .if    (APPLE)              ;Apple GEOS hides registers
```

```
        sta     ALTZP_ON                ;on aux. zpage.
        .endif
        CmpW    rightMargin,r11         ;check x with right edge
        ble     90$                     ;exit if right not exceeded;
                                        ;the character was outside the
                                        ;left edge.

;Save the pointer to the offending character in r15 (which is left
;untouched by the normal PutString)
        lda     r0L
        ldx     r0H
        .if     (APPLE)                 ;need to change r15 on main zp if apple
        sta     ALTZP_OFF
        .endif
        stx     r15H
        sta     r15L
        .if     (APPLE)                 ;return Apple to
        sta     ALTZP_ON                ;aux. zpage.
        .endif

;Change the string pointer so that PutString thinks the next
;character is a null.
        LoadW   r0, #(FakeNull-1)       ;one less to compensate for
                                        ;increment that PutString will
                                        ;do before it checks.

90$:
;Return to let PutString do its stuff
        .if (APPLE)
        sta     ALTZP_OFF
        .endif
        rts                             ;return to to StringFault caller

FakeNull:       .byte   NULL            ;null for FaultFix
```

Embedding Style Changes Within a String

A string may contain embedded escape codes for changing the style attributes mid-string. For example, if while printing a string GEOS encounters a **BOLDON** (24) escape code, then **PutString** will temporarily *escape* from normal processing to set the boldface bit in **currentMode**. Any characters thereafter will be printed in boldface.

Style changes are typically cumulative. If a **OUTLINEON** code is sent, for example, then the outline style attribute will be added to current set of attirbutes. If boldface was already set, then subsequent characters will be both outlined and boldfaced. The **PLAINTEXT** escape code returns text to its normal, unaltered state.

When **PutString** is first called, it begins printing in the styles specified by the value in **currentMode** and when it returns, **currentMode** retains the most recent value, reflecting any style-change escapes. The next call to **PutString** (or any other GEOS printing routine) will continue printing in that style. To guarantee printing in a particular style without inheriting any style attributes from previous strings, the first character in the string should be a **PLAINTEXT** escape code. Any specific style escape codes can then follow.

Position Escapes (Moving the Printing Position Mid-string)

GEOS provides escape codes for changing the current printing position. Like other escape codes, these can be embedded within the string. Some of them are simple, such as **LF** and **UPLINE**, which move the current printing position down one line or up one line, respectively, based on the height of the current font. Others, such as **GOTOX**, **GOTOY**, and **GOTOXY**, require byte or word pixel coordinates to be embedded within the string immediately after the escape code.

Example:

```
String:
       .byte   HOME            ;start in the upper-left corner
       .byte   LF              ;move down one line.so we have room
       .byte   "This ",LF,"is ",LF,"stepping ",LF
       .byte   "Down",LF"ward",CR
       .byte   LF, "HELLO"
       .byte   GOTOXY
       .word   40                      ;x-position
       .byte   15                      ;y-position of baseline
       .byte   "Look! I moved."
       .byte   NULL
```

Escaping to a Graphics String

GEOS provides a special escape code (**ESC_GRAPHICS**) that takes the remainder of a string and treats it as input to the **GraphicsString** routine. This allows graphics command to be embedded within a text string, which is useful for creating complex displays, especially those that require graphics to be drawn *over* text. The current pen positions for the graphics are unitialized so the first graphics string command should be a **MOVEPENTO**.

Example:

```
TextGraphics:                  ;string with both text and graphics
       .byte   GOTOXY
       .word   20
       .byte   20
       .byte   "BOX:   "
       .byte   ESC_GRAPHICS
       .byte   MOVEPENTO
       .word   10
       .byte   10
       .byte   RECTANGLETO
       .word   50
       .byte   30
       .byte   NULL
```

Note: When **GraphicsString** encounters the **NULL** marking the end of a string, control is returned to the application as if **PutString** had terminated normally. The **NULL** *does not* resume **PutString** processing.

If it is necessary to print additional text after graphics, the **ESC_PUTSTRING** command may be used to escape from **GraphicsString**. A subsequent **NULL** will still mark the end of the string. Be aware that each context-switch between these two routines allocates additional 6502 stack space that is not released until the **NULL** terminator is encountered.

GetString

GetString provides a convenient way for an application to get text input from the user without using a dialog box. **GetString** takes care of intercepting keypresses and echoing the characters to the screen. The beauty of **GetString** is that it builds the string concurrently with the rest of

MainLoop, allowing menus, icons, and processes to remain functional while the user is typing in the string.

When you call **GetString**, you place the address you want GEOS to call when the user presses [Return] into **keyVector**. GEOS saves this address, prints out an optional default string, and inserts its own routine (**SystemStringService**) into **keyVector**, assuming control of future keypresses. GEOS then returns back to the application with an **rts**, which is left to return to **MainLoop** in its normal course of events. As **MainLoop** encounters keypresses, it vectors through **keyVector**, calling **SystemStringService**. **SystemStringService** masks out invalid keypresses and prints valid characters, backspacing as necessary when the backspace key is pressed. When the [Return] key is pressed, GEOS clears **keyVector** and calls the event routine specified in **keyVector** when **GetString** was called. The null-terminated string is passed in a buffer.

GetString has a variety of options and flags that are described compeletely in the **GetString** reference section. These include specifying a maximum length for the entered string, providing a default string, and enabling an option to give application control of string faults. But **GetString** is of limited usefullness, and applications that rely on a lot of this type of keyboard and text interaction might warrant a customized string/keyboard routine.

GetString and dispBufferOn

GetString uses the **Putchar** routine to print text to the screen, and **Putchar** depends on the value in **dispBufferOn** to decide where to direct its output. Because **SystemStringService** runs concurrently with other **MainLoop** events — events that might alter the state of **dispBufferOn** — it needs a way to override the current value of **dispBufferOn**, otherwise text will print based on the current value of **dispBufferOn**, which, depending on the events running off of **MainLoop**, may contain different values on every keypress, sending characters to different screen buffers at different times.

NOT ENTIRELY CLEAR; MAKE SURE PEOPLE KNOW THAT THIS IS NOT REALLY OF THAT MUCH IMPORTANCE...

Some early versions of GEOS used bit 5 of **dispBufferOn** as a flag to limit **GetString**'s character printing to the foreground screen. This bit, however,is no longer guaranteed to have this effect and should always be zero.

One solution to controlling where **GetString** sends its characters, demonstrated below, involves patching into **keyVector** and updating **dispBufferOn** before **SystemStringService** gets control.

```
.if 0
;**************************************************************
;NewGetString
;
;       New front-end to GetString that wedges into keyVector before
;       SystemStringService gets control. This routine uses StringPatch to
;       adjust dispBufferOn so that it holds the value that it
;       contained when NewGetString was first called, making every
;       character print consistently. It otherwise acts just like
;       GetString.
;
;       Pass:   same as GetString.
;
;       Returns:        same as GetString.
;
;       Destroys:       same as GetString.
;
;**************************************************************
.endif
```

```
NewGetString:

;Save the current value of dispBufferOn to stuff back each time
;SystemStringService gets control.
        MoveB   dispBufferOn,tempDisp

;Call GetString as normal
        jsr     GetString

;Now that GetString has put SystemStringService into keyVector, we
;need to pre-empt that. We save off the address in keyVector and
;place our StringPatch routine in its place.
        MoveW   keyVector,sysKeySave          ;save old
        LoadW   keyVector,#StringPatch        ;install ours

;Exit
        rts

;*******************************************************************
;StringPatch:
;
;When a key is pressed during a GetString, control comes here.
;We load up the correct value of dispBufferOn, link through to
;the correct SystemStringService, and restore dispBufferOn when
;control comes back. When the string is terminated with [Return],
;SystemStringService will take care of removing us.
;*******************************************************************

StringPatch:

;Save the current value of dispBufferOn
        PushB   dispBufferOn

;Load up the correct value for dispBufferOn that NewGetString
;saved away for us.
        MoveB   tempDisp,dispBufferOn

;Continue through SystemStringService
        lda     sysKeySave
        ldx     sysKeySave+1
        jsr     CallRoutine

;We will eventually get control again. Restore the old value
;of dispBufferOn before going back to MainLoop
        PopB    dispBufferOn

;Exit
        rts

        .ramsect
tempDisp:      .block 1               ;temporary hold for dispBufferOn
sysKeySave:    .block 2               ;holds address of system key routine

        .psect
```

Note: When **GetString** returns, **keyVector** will always be set to $0000. If the application was using **keyVector**, it will need to reload it after the string has ended.

Forcing End of String Input

Because **GetString** accepts input concurrently with **MainLoop**, there might be some user action other than pressing [Return] that the application may want to recognize as the end of input marker.

Unfortunately, there is no direct way to terminate **GetString** before the user presses [Return]. The trick of choice in this situation is to simulate a press of the return key by loading **keyData** with a **CR** and vectoring through **keyVector** as in:

```
;Simulate a CR to end GetString
        LoadB   keyData,#CR          ;load up a CR [Return] key
        lda     keyVector            ;and vector through keyVector
        ldx     keyVector+1          ;so SystemStringService will
        jsr     CallRoutine          ;think it was pressed now
```

This same technique can be used to terminate a **DBGETSTRING** when an icon is pressed to leave a dialog box.

Note:	The Apple GEOS version of **GetString** always keeps the partial string null-terminated while it is building it in the buffer. An application can peek at the status of the string by looking in this buffer. GEOS 64 and GEOS 128 (through v1.3) do not null-terminate the string until [Return] is pressed (or simulated).

Fonts

In GEOS a font is a complete set of characters of a particular size and typeface. On disk, fonts are organized by style, where a single font file holds all the available point sizes for a given style. Each point size occupies its own VLIR record in the font file. The record number corresponds to the point size. For example, a font file called MyFont might use three VLIR records, one for each available font size: the MyFont 10 would occupy record 10, MyFont 12 would occupy record 12, and MyFont 24 would occupy record 24.

It is the job of the application to decide which fonts to keep in memory at any one time, reading in the appropriate records from the VLIR font file. Once a font is in memory (usually as the result of a call to **ReadRecord**), the application must inform GEOS to begin using the new font with the following routine:

• **LoadCharSet**	Instruct GEOS to begin using a new font. (Font is already in memory.)

Because Apple GEOS allows font data to be stored in auxiliary memory, an additional routine is provided for doing the equivalent of **LoadCharSet** when the font is in auxiliary memory (**LoadCharSet** is still used when the font is in main memory):

• **LoadAuxCharSet**	Instruct GEOS to begin using a new font. (Font is already in auxiliary memory.)

Although the word "Load" in **LoadCharSet** and **LoadAuxCharSet** is misleading in that it implies they automatically load the character set from disk into memory, the application must read the font data into memory prior to calling these routines. **LoadCharSet** and **LoadAuxCharSet** expect an address pointer to the beginning of the font in memory. It will then build out a variable table for the text routines, providing information such as the baseline offset and font point height. The application may keep as many fonts resident as free memory will allow, switching them at will with calls to **LoadCharSet** and **LoadAuxCharSet** . Some sophisticated GEOS applications use a font-cacheing system where fonts are kept in memory based on their frequency of use.

GEOS provides an additional routine for returning to the always-resident BSW 9 system font:

• **UseSystemFont**	Instruct GEOS to begin using the default BSW 9 font.

UseSystemFont passes the address of the the system BSW 9 font to **LoadCharSet**.

The Structure of a Font File

Fonts are stored in VLIR files of GEOS type **FONT**. A single font file contains all the available point sizes for a particular style (up to a maximum of 16). Each point size occupies one complete VLIR record. The record number corresponds to the point size (i.e., record 9 would contain the data for the nine point character set). If a VLIR record in a font file is empty, then the corresponding point size is not available (the record will exist, but will marked as empty in the index table). The data in each of these records is what GEOS considers a character set, and its structure is described later in "Character Set Data Structure." Unless the application is creating or modifying fonts, this data structure is unimportant.

The font files on a given disk can be found using the **FindFTypes** routine. Once the font files are known, the application can use ~~**GetHdrInfo**~~ GetFHdrInfo to access the header block for each font file. The font file header block contains information pertinent to the particular font file, such as the font style ID, the available point sizes, and the amount of memory required for each point size. These values can be accessed in the header block by using the following offsets:

HEADER? OFFSETS INTO FONT FILE HEADER

Offset	Field size	Description
O_GHFONTID	1 word	Font style ID.
O_GHPTSIZES	16 words	Character set ID's for those available in this file. Arranged from smallest to largest point size. Table is padded with zeros.
O_GHSETLEN	16 words	Size (in bytes) of each character set from smallest to largest point size. (These numbers have a one-to-one coorespondence with the **O_GHPTSIZES** table. Table is padded with zeros.

SC
19
f

Every font style has a unique 10-bit ID number. This number is stored in the word-length field **O_GHFONTID**. The next field, **O_GHPTSIZES**, has room for 16 character set ID numbers.A character set ID number is a 16-bit combination of the style ID and a point size identifier. The style ID is stored in the upper 10 bits and the point size is stored in the lower 6 bits:

Character Set ID Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
10-bit style ID (b6–b15)										point size (b0-b5)					

b6–b15	Style ID.
b0–b5	Point Size.

This combination of style ID number and point size gives each character set (font) a unique word-length identifier. This allows any style/point-size combination to be referenced with a two-byte number. For example, the Durrant style has a style ID of 15, so the Durrant 10 font would have a character ID of

(15 << 6) | 10 or **$03ca**

Berkeley Softworks' applications use the **NEWCARDSET** escape followed by the character set ID word to flag font changes within a text document.

> **Note:** A complete list of font ID's for registered fonts appears in Chapter XX. Developers wishing to register their own fonts should contact Berkeley Softworks in writing for available ID numbers:
>
> Berkeley Softworks
> Attn: Font Registration
> 2150 Shattuck Avenue
> Berkeley, CA 94704
>
> Each request must include a disk with the font files along with the desired registration names.

Character Set Data Structure

A character set is stored — both in memory and in its VLIR record — as a contiguous data structure consisting of an eight-byte header, followed by an index table and the actual character image data. The image data for the characters are stored in a *bitstream* format, pixel row by pixel row. Imagine laying every printable character side by side, in character code order, starting with character number 32 (the space character). If the top row of pixels from every character were then stored together as a contiguous stream of bits, this would be the proper bitstream format. In GEOS, for every pixel of height in a character set, there is a corresponding *bitstream row*. Starting with the top row, each bitstream row is padded with zeros to make it end on a byte boundary. The next row (if there is one) is appended at the next byte. The number of bytes in each bitstream row is called the *set width.*

Because each character in a GEOS font can be of a different pixel width, GEOS needs some way of indexing into the bitstream data to find the beginning of each character. For each character there is a ~~pixel~~ index ~~word,~~ that indicates where the character begins in the bitstream. For example, if the first pixel for the "A" character begins at pixel 148 in the bitstream, then the index value for character code 65 (uppercase "A") would be 148.

Character Set Data Structure

Offset	Field size	Description IN
+0	byte	Baseline offset (pixels from top of character).
+1	word	Bytes in one bitstream row (set width).
+3	byte	Font height. (NUMBER OF BITSTREAM ROWS)
+4	word	Pointer to beginning of index table (relative to beginning of data structure). Usually $0008 because the index table follows immediately after the ~~header~~. NEXT WORD.
+6	1 word	Pointer to beginning of character bitstream data (relative to beginning of data structure). Bitstream data typically follows the index table.
+?	? words	Index table: one word entry for each printable character (the first word corresponds to character code 32). Each index word is pixel position of the character in each bitstream row. Total number of words = number of printable characters in the set.
+?	? bytes	Bitstream rows: one row of bitstream data for each pixel of height in the character set. Each bitstream row is padded with zeros out to the next byte. Total bytes = number of printable characters in the set times the set width.

CHAR SET DEFAULT

Saving and Restoring the Font Variables

In both GEOS 64 and GEOS 128, all the information GEOS needs for using a font is stored in the variable table beginning at **fontTable** and stretching for **FONTLEN** bytes. Whenever GEOS needs to switch fonts internally (while drawing the BSW 9 text into menus, for example), these bytes are saved off to **saveFontTab**, which is also **FONTLEN** bytes long. If a Commodore GEOS application needs to temporarily change fonts, it can simply duplicate this technique, saving and restoring between **fontTable** and **saveFontTab** as needed.

Under Apple GEOS, however, not all the font information is accessible to applications. Apple GEOS, therefore, includes two routines for saving and restoring all the necessary font table information between its own variables and **saveFontTab**:

• **SaveFontData**	Save internal font data to **saveFontTab**.
• **RestoreFontData**	Restore internal font data from **saveFontTab**.

An application should never return to **MainLoop** with valid data in the **saveFontTab** area because **MainLoop** may use the **saveFontTab** area for its own purposes, thereby destroying any font information that may be saved there. Of course, the information in **saveFontTab** can always be copied to another buffer before giving **MainLoop** control.

When GEOS runs a desk accessory, it saves off all the current font variables to a special area of memory. However, the temporary **saveFontTab** area is *not* saved. If a desk accessory uses menus, the menu routines will use the area at **saveFontTab**, thereby overwriting any saved data. Since the **saveFontTab** area is not saved in the context switch between the application and the desk accessory, it will come back with incorrect data. It is, therefore, the desk accessory's job to save and restore the data at **saveFontTab** if necessary.

Keyboard Input

Many keyboard input needs can be accomodated through normal processing with **GetString** and through dialog boxes with **DBGETSTRING**, but many specialized functions require servicing keypresses directly. The application might want to implement shortcut keys — special key combinations that allow quick access to menu items or other functions — or an application, such as a word processor, might need to do dynamic text formatting as characters are typed.

Key-scan Conversion

The internal code that the computer hardware returns for each keypress usually reflects the position of the key on the keyboard, not the actual character on the keycap. GEOS pre-processes all keypresses, ignoring some and translating others. For most keys, the keypress is translated into the GEOS ASCII character code equivalent: [a] translates to 97, [SHIFT] + [a] translates to 65, and [RETURN] translates to **CR**. These keys can go directly to GEOS text routines wihtout any further work. However, there are some key combinations that get translated outside of the printable character range (codes between 0 and 32), and the application will need to filter these out.

Note:	Apple GEOS input drivers and aux-drivers both have the opportunity to preprocess or translate keypresses before they undergo the standard GEOS translation. For more information, refer to **KeyFilter** and **AuxDKeyFilter** in the Routine Reference Section.

If the shortcut key (designated by the Commodore logo on CBM computers and the filled Apple logo on Apple computers) is pressed in combination with another key, the high-bit (bit 7) of the keypress byte will be set. This means, for example, that [SHORTCUT] + [a] is equivalent to

```
.byte  (SHORTCUT | 'a')
```

How GEOS Handles Keypresses

At interrupt level, GEOS scans the keyboard looking for a key presses and releases. If a new key has been pressed or an old key has been held down long enough to begin auto-repeating, GEOS places the corresponding character code for the key at the end of the keyboard queue. The keyboard queue is a circular FIFO (first-in, first-out) buffer that holds keypresses. A queue is used because many typists can, at times, type keys faster than the application can process them. If there was no key buffer, keypresses would be lost. As long as there are characters in in the keyboard queue, the **KEYPRESS_BIT** of **pressFlag** is set.

On each pass through **MainLoop**, GEOS checks the **KEYPRESS_BIT** of **pressFlag**. If the bit is set, GEOS removes the oldest keypress from the queue, places it in the global variable **keyData**, and attempts to vector through **keyVector**. **keyVector** usually contains a $0000, which causes GEOS to ignore the vector and, hence, ignore the keypress. As long as **keyVector** is $0000, keypresses will continue to accumulate in the queue at interrupt level and be ignored, one at a time, at **MainLoop** level.

By placing the address of a key-handling routine in **keyVector**, the application can be called off of **MainLoop** to process keypresses as they become available. When the application's key handler gets called, it merely picks up the key code from **keyData**, does any necessary processing, and returns to **MainLoop** with an **rts** when done.

With this technique, though, the application can only process one keypress on each pass through **MainLoop**, even though the keyboard queue may have more than one character in it. This is typically not a problem because the overhead most applications need to handle a character is minimal. But take geoWrite, for example. If only one character could be processed at a time, it might need to print, word-wrap, and scroll for each character. Even a medium speed typist could get far ahead of the screen updating. If there was a way to get at all the keypresses in the queue at once, then all the calculating and screen manipulations could be done for more than one character on each pass through **MainLoop**. GEOS offers a routine to do just this:

• GetNextChar	Retrieve the next character from the keyboard queue.

GetNextChar gets the keycode of the next available character from the keyboard queue and returns it in the accumulator. If there are no more characters available, **GetNextChar** returns a **NULL**. To retrieve all the queued keypresses, an application can call **GetNextChar** in a loop, transferring all queued characters to its own buffer. This buffer must be at least **KEY_QUEUE** bytes long so that it won't be overflowed.

Example:

```
;*******************************************************************
;KeyHandler
;
;       Sample key handler. Stuff address of this routine into
;       keyVector. Unloads the keyboard queue into an internal
;       buffer but does nothing with the characters.
;
;*******************************************************************

KeyHandler:
        ldx     #0                      ;start at beginning of internal buffer
        lda     keyData                 ;get first keypress
        sta     newKeys,x               ;store it in my buffer

        php                             ;lock out interrupts for a moement
        sei                             ;so we don't get any new keypresses
10$:
        inx                             ;point to next position in buffer
        jsr     GetNextChar             ;get another character
        sta     newKeys,x               ;put it in our buffer
        cmp     #NULL                   ;was that the last
        bne     10$                     ;loop back to get more

        plp                             ;restore interrupt disable status

;All new keys are now in our buffer. Our buffer is conveniently
;null-terminated because the last character we set down was a
;NULL. Neat, huh?
        jsr     DoNewKeys               ;go process the keys we picked up

99$:
        rts                             ;return to MainLoop

        .ramsect
newKeys:        .block KEY_QUEUE+1      ;max queue size + NULL
        .psect

;*******************************************************************
;DoNewKeys
;
;       A do-nothing routine that just pretends to empty our own
```

```
;       keyboard buffer.
;*******************************************************************

DoNewKeys:
        ldx     #$00                    ;start at beginning of buffer
10$:
        lda     newKeys,x               ;get a key
        beq     20$                     ;exit loop if it's the null
        nop                             ;do nothing with this keypress
        inx                             ;point to next position
        bne     10$                     ;always branch (X should never go to 0)
20$:
;We've encountered the NULL and therefore gone through the entire
;string. Clear the buffer by storing the null in the first
;position of the string.
        sta     newKeys+0

99$:
        rts                             ;exit
```

Ignoring Keys While Menus are Down

Becuase **MainLoop** is still running full-speed when menus are down, **keyVector** will still ber vectored through on a regular basis. The application may want to postpone any text output or keypress interpretation when menus are down. Checking for this case is simple:

```
        lda             menuNumber      ;check current menu level
        bne             99$                     ;leave if any menus are down
```

Implementing Shortcuts

Shortcut keys are a common user-interface facility found in GEOS applications. Briefly, a shortcut key is a key combination that allows the quick selection of a menu item or function in the application. Typically shortcuts are distinguished from other keypresses by pressing the shortcut key (the Commodore logo or the filled Apple logo) while typing another key. Key combinations that include the shortcut key will have the high-bit set, which makes them easy to recognize. Even if an application is not using shortcuts, it will most likely want to at least filter out all shortcut keys.

To process shortcut keys, the normal key handler (the one the application installs into **keyVector**) should first check the high-bit of the keypress and branch to the shortcut key handler if the bit is set:

```
KeyHandle:
        lda             menuNumber      ;check current menu level
        bne             99$             ;ignore keys while menus down
        lda             keyData         ;get the keypress
        bmi             10$             ;was it a shortcut?
        jsr             NormalKey       ;no, process normally
        bra             99$             ;exit
10$:
        jsr             ShortKey        ;yes, process as a shortcut
99$:
        rts                             ;exit
```

The shortcut key handler will need to decide what to do based on the key that was pressed. Usually the shortcut bit (bit 7) will be removed, the character will then be converted to uppercase, and the resulting character code will be used to search through a table of valid shortcut keys. If the particular shortcut key is not supported, the handler just returns, ignoring the keypress. If the key is implemented, the handler needs to call an appropriate subroutine to process the shortcut key:

```
;*****************************************************************
;Shortcut key handler. Call with keycode in A-register
;*****************************************************************
ShortKey:
;Do some minor conversion on the keycode
       and           #~SHORTCUT            ;lop off shortcut bit
       cmp           #'a'                  ;check if lowercase
       blt           10$                   ;branch if less than "a"
       cmp           #'z'+1        ;or greater than "z"
       bge           10$                   ;
       sec                                 ;it's lowercase: convert to upper
       sbc           #('a'-'A')    ;by subtracting the ASCII difference
                                           ;between a lowercase 'a' and an
                                           ;uppercase 'A'

10$:
;Now that we have a shortcut key, we go searching through
;a table of valid shortcut keys, looking for a match. Use Y-reg
;to index so we can use X-reg later for CallRoutine.
       ldy           #NUM_SHORTCUTS        ;start at top of table
20$:
       cmp           shortCuts,y   ;check for a keycode match
       beq           30$                   ;branch if found
       dey                                 ;else, try next
       bpl           20$                   ;loop until done. NOTE: must
                                           ;not have more than 127 shortcuts
                                           ;or this branch will fail!
       bmi           99$                   ;no match, ignore this key
30$:
;We've found a match. Get the corresponding routine address from
;the jump table and call the routine
       ldx           h_shortCutTbl,y       ;get high address of routine
       lda           l_shortCutTbl,y       ;and low address
       jsr           CallRoutine           ;call the routine

99$:
       rts                                 ;exit

;*****************************************************************
;Table of shortcut keys and their corresponding routines
;*****************************************************************

;Valid shortcut keys
shortCuts:
       .byte  'O'           ;1  undo
       .byte  'T'           ;2  text
       .byte  'P'           ;3  print
       .byte  'Q'           ;4  quit
       .byte  'N'           ;5  new document
       .byte  'G'           ;6  goto page
       .byte  'B'           ;7  boldface toggle
       .byte  'O'           ;8  outline toggle
       .byte  'I'           ;9  italic toggle
       .byte  'U'           ;10 underline toggle
       .byte  'D'           ;11 delete
       .byte  'C'           ;12 copy
       .byte  'S'           ;13 scroll
       .byte  'L'           ;14 load document

NUM_SHORTCUTS == (* - shortCuts - 1)       ;number of shortcuts
.if    (NUM_SHORTCUTS > 127)
       .echo  WARNING: too many shortcuts
.endif
```

```
;Table of low bytes of shortcut routine
l_shortCutTbl:
        .byte   [DoUndo             ;1
        .byte   [DoText             ;2
        .byte   [DoPrint            ;3
        .byte   [DoQuit             ;4
        .byte   [DoNew              ;5
        .byte   [DoGoto             ;6
        .byte   [DoBoldface         ;7
        .byte   [DoOutline          ;8
        .byte   [DoItalic           ;9
        .byte   [DoUnderline        ;10
        .byte   [DoDelete           ;11
        .byte   [DoCopy             ;12
        .byte   [DoScroll           ;13
        .byte   [DoLoad             ;14

;Table of high bytes of shortcut routine
h_shortCutTbl:
        .byte   ]DoUndo             ;1
        .byte   ]DoText             ;2
        .byte   ]DoPrint            ;3
        .byte   ]DoQuit             ;4
        .byte   ]DoNew              ;5
        .byte   ]DoGoto             ;6
        .byte   ]DoBoldface         ;7
        .byte   ]DoOutline          ;8
        .byte   ]DoItalic           ;9
        .byte   ]DoUnderline        ;10
        .byte   ]DoDelete           ;11
        .byte   ]DoCopy             ;12
        .byte   ]DoScroll           ;13
        .byte   ]DoLoad             ;14
```

The Text Entry Prompt

Whenever an application will be accepting text input, it is a good idea to offer a prompt, or cursor, to mark the point at which text will appear. GEOS offers three routines for automatically configuring sprite #1 to act as a text entry prompt:

• InitTextPrompt	Initialize sprite #1 for use as a text prompt.
• PromptOn	Turn on the prompt (show the text cursor on the screen).
• PromptOff	Turn off the prompt (remove the text cursor from the screen).

The prompt automatically flashes on the screen without disrupting the display and can be resized to reflect the point size of a particular font.

Important: Interrupts should always be disabled and **alphaFlag** should be cleared when **PromptOff** is called. The following subroutine illustrates the proper use of **PromptOff**:

```
KillPrompt:
        php                                     ;save i status
        sei                                     ;disable interrupts
        jsr             PromptOff               ;prompt = off
        LoadB   alphaFlag,#0  ;clear alpha flag
        plp                                     ;restore i status
        rts                                     ;exit
```

Sample Keyboard Entry Routine

As an example, we will use some of the concepts covered in this chapter in real-world code. The following routine will patch into **keyVector** and output text as keys are pressed:

```
;*** CONSTANTS ***

TXT_LEFT        == 10                           ;text left margin
TXT_RIGHT       == (SC_PIX_WIDTH - TXT_LEFT)    ;text right margin
TXT_TOP         == 20                           ;text top margin
TXT_BOT         == (SC_PIX_HEIGHT - TXT_TOP)    ;text bottom margin

;text (x,y) starting position
TXT_X   == 20
TXT_Y   == 50

;size of the text buffer
TXTBUFSIZE      == $200                 ;1/2K is far more than enough for
                                                ;now. To accept multiple lines,
                                                ;the buffer will need to grow

;Characters to accept before buffer overflow fault
MAX_CHARS               == 30


.if 0
;*****************************************************************
;StartText:
;
;       Initializes the text input process by loading the proper
;       vectors, setting flags, etc. Wedges KeyIn into keyVector to
;       intercept keypresses and output them to a single line.
;
;       Pass:   nothing
;
;       Returns:        text input routine in keyVector
;
;       Destroys:       ?
;
;*****************************************************************
.endif

StartText:

;Send our text output to both screens
        LoadB   textDispBufOn,#(ST_WR_FORE | ST_WR_BACK)
```

```
;Install our character handler
        LoadW   keyVector,#KeyIn            ;keypresses vector thru here
        LoadW   stringFaultVec,#TextFault   ;and string faults here

;Install the system font and clear all text attributes
        jsr     UseSystemFont
        lda     #PLAINTEXT
        jsr     PutChar

;Set the left and right margins
        LoadW   leftMargin,#TXT_LEFT
        LoadW   rightMargin,#TXT_RIGHT

;Set the top and bottom margins
        LoadW   windowTop,#TXT_TOP
        LoadW   windowBottom,TXT_BOT

;Set the text starting position
        LoadW   stringX,#TXT_X
        LoadB   stringY,#TXT_Y

;Initialize the prompt
        lda     currentHeight
        jsr     InitTextPrompt
        jsr     PromptOn

;Point at the start of the line buffer
        LoadW   txtBuf,#bigTextBuffer       ;where to start
        LoadB   txtBufIndex,#0              ;index from start

;Max number of characters to accept
        LoadB   txtInMax,#MAX_CHARS

;And where control goes if we go over...
        LoadW   bufFaultVec,#BufOverflow

;Turn text on
        LoadB   textOn,#TRUE

;Exit
        rts

        .ramsect

;Buffer that will hold all the text we enter. We let the key input
;routine build it up a line at a time by passing

bigTextBuffer:          .block TXTBUFSIZE
textDispBufOn:          .block 1            ;holds dispBufferOn value for text
txtInMax:               .block 1            ;number of characters that will
                                            ;generate buffer overflow fault
textOn:                 .block 1            ;text is ON flag. (TRUE = ON)

.if ((* & $ff) == $ff)          ;if indirect jump vector straddles a page
        .block 1                ;boundary, fix it to compensate for a bug
.endif                          ;in the 6502 architecture
bufFaultVec     .block 2

        .psect

;*******************************************************************
;KeyIn:
;
;When a key is pressed, control comes here off of MainLoop
```

```
;
;****************************************************************

KeyIn:
        lda     menuNumber              ;check current menu level
        bne     99$                     ;ignore keys while menus down
        lda     keyData                 ;get the keypress
        bmi     10$                     ;was it a shortcut?
        jsr     NormalKey               ;no, process normally
        bra     99$                     ;exit
10$:
        jsr     ShortKey                ;yes, process as a shortcut
99$:
        rts                             ;exit

;****************************************************************
;ShortKey:
;
;Control comes here when shortcut keys are pressed
;
;****************************************************************

ShortKey:
        rts                             ;no shortcut key handler now. just ignore keypress.


;****************************************************************
;NormalKey:
;
;Control comes here when non-shortcut keys are pressed
;
;****************************************************************

SPACE   = 32                            ;first printable character code

NormalKey:
;Return immediately if text is off
        lda     textOn
        bne     5$                      ;branch if text on
        rts                             ;

5$:
        jsr     KillPrompt              ;turn the prompt off

;Save the current value of dispBufferOn and load up the correct
;value for text output.
        PushB   dispBufferOn
        MoveB   textDispBufOn,dispBufferOn

;Load the current cursor position into the PutChar position
;registers, just in case we need to use them later.
        MoveW   stringX,r11             ;x printing position
        lda     stringY                 ;convert y cursor position to
        clc                             ;baseline position
        adc     baselineOffset          ;
        sta     r1H                     ;y printing position

;Process the character
        lda     keyData                 ;get the keypress again
        cmp     #SPACE                  ;cmp with first printable char
        bge     40$                     ;branch if printable

;Check the control character against a table of special action
;keys. Use Y-reg to index so we can use X-reg later for
```

```
;CallRoutine.
        ldy     #NUM_CTRL               ;start at top of table
20$:                                    ;
        cmp     ctrlKeys,y              ;check for a keycode match
        beq     30$                     ;branch if key matches table entry
        dey                             ;else, try next
        bpl     20$                     ;loop until done. NOTE: must not
                                        ;have more than 127 special keys
                                        ;or this branch will fail!
        bmi     88$             ;no match was found, ignore this key

30$:
;We've found a match on a control character. Get the corresponding
;routine address from the jump table and call the routine
        ldx     h_CtrlTbl,y             ;get high address of routine
        lda     l_CtrlTbl,y             ;and low address
        jsr     CallRoutine             ;call the routine
        bra     88$                     ;go clean up and exit

40$:
;It's a normal alphanumeric character. Output it to the
;screen and save it in the text buffer
        pha                             ;save the character code
        ldy     txtBufIndex             ;pointer into current text buffer
        sta     (txtBuf),y              ;place the character into the buffer
        iny                             ;point to next position in buffer
        lda     #NULL                   ;and null-terminate the string
        sta     (txtBuf),y              ;
        sty     txtBufIndex             ;set down the new index value
        pla                             ;get the character code back. (Note:
                                        ;  we could have pulled it off of
                                        ;  keyData, but future versions may
                                        ;  pre-process or translate the char
                                        ;  code in the A-reg before passing)
        jsr     PutChar                 ;print it on the screen
        MoveW   r11,stringX             ;update the prompt X-position
        lda     txtBufIndex             ;was that the last character we
        cmp     txtInMax                ;can accept?
        blt     88$                     ;OK if under max.
        lda     bufFaultVec             ;otherwise, call buffer overflow
        ldx     bufFaultVec+1           ;routine
        jsr     CallRoutine             ;

88$:
;Clean up
        lda     textOn                  ;only re-enable the prompt if text
        beq     99$                     ;is still on (might have changed!)
        jsr     PromptOn                ;turn the prompt back on

99$:
        PopB    dispBufferOn            ;restore dispBufferOn
        rts                             ;Exit


;*******************************************************************
;Table of control keys and their corresponding routines
;*******************************************************************

;Valid control keys
ctrlKeys:
        .byte   CR                      ;1 Carriage return
        .byte   BACKSPACE               ;2 backspace
        .byte   KEY_DELETE              ;3 ditto
        .byte   KEY_INSERT              ;4 ditto
```

```
        .byte   KEY_RIGHT               ;5 ditto

NUM_CTRL        == (* - ctrlKeys - 1)           ;number of control keys
.if     (NUM_CTRL > 127)
        .echo   WARNING: too many control keys
.endif

;Table of low bytes of control key routine addresses
l_CtrlTbl:
        .byte   [DoReturn               ;1
        .byte   [DoBackSpace            ;2
        .byte   [DoBackSpace            ;3
        .byte   [DoBackSpace            ;4
        .byte   [DoBackSpace            ;5

;Table of high bytes of control key routine addresses
h_CtrlTbl:
        .byte   ]DoReturn               ;1
        .byte   ]DoBackSpace            ;2
        .byte   ]DoBackSpace            ;3
        .byte   ]DoBackSpace            ;4
        .byte   ]DoBackSpace            ;5

;Exit
        rts

        .ramsect
tempDisp:       .block 1                ;temporary hold for dispBufferOn
sysKeySave:     .block 2                ;holds address of system key routine

        .psect

;*****************************************************************
;KillPrompt:
;
;Proper way to use PromptOff. Disable interrupts and
;clears alphaFlag.
;
;*****************************************************************
KillPrompt:
        php                             ;save i status
        sei                             ;disable interrupts
        jsr     PromptOff               ;prompt = off
        LoadB   alphaFlag,#0            ;clear alpha flag
        plp                             ;restore i status
        rts                             ;exit

;*****************************************************************
;DoReturn:
;
;Process a carriage return
;
;*****************************************************************

DoReturn:
;No real carriage return handler, yet. Just shut text off
        LoadB   textOn,#FALSE           ;
        rts                             ;


;*****************************************************************
;DoBackspace:
;
;Process a backspace
```

```
;
;*****************************************************************

DoDoBackspace:
       ldy     txtBufIndex          ;get ptr into current text buffer
       beq     99$                  ;if no characters in buffer, exit
       dey                          ;back up a character
       sty     txtBufIndex          ;and make the new index permanent
       lda     (txtBuf),y           ;get the character we want to delete
       jsr     EraseCharacter       ;and remove it from the screen
       ldy     txtBufIndex          ;get the index to the character we
       lda     #NULL                ;we just deleted and make it the
       sta     (txtBuf),y           ;null-terminator
       MoveW   r11,stringX          ;update the cursor's x-position
99$:                                ;
       rts                          ;exit

;*****************************************************************
;EraseCharacter:
;
;Physically remove a character from the screen
;*****************************************************************
.if    (C64 || C128) ;This routine is in the Apple GEOS jump table
EraseCharacter:
       MoveW   r11,r4               ;current X is rectangle's right edge
       ldx     currentMode          ;get the mode we're in
       jsr     GetRealSize          ;go calc the size of the character
       sta     r3L                  ;set down baseline offset
       lda     r1H                  ;calc top of character by subtracting
       sec                          ;baseline offset from y-position
       sbc     r3L                  ;
       sta     r2L                  ;and making top edge of rectangle
       txa                          ;add char height to top edge
       clc                          ;to calc bottom edge
       adc     r2L
       sta     r2H                  ;and make bottom of rectangle
       sty     r3L                  ;set down width so we can subtract it
       sec                          ;from the current x-position to
       sbc     r11L                 ;find the character's starting
       sta     r3L                  ;position
       ldy     r11H
       bcs     10$                  ;subtract one from hi if borrow
       dey
10$:
       sty     r3H                  ;make left edge of rectangle
       jsr     Rectangle            ;erase in current pattern
       rts                          ;exit
.endif

;*****************************************************************
;BufOverflow:
;
;What to do if the buffer hits its maximum.
;
;*****************************************************************

BufOverflow:
;No real overflow handler, yet. Just shut text off
       LoadB   textOn,#FALSE        ;
       rts                                 ;

;*****************************************************************
;TextFault:
```

```
;
;String faults come here.
;
;*****************************************************************

TextFault:
;No real text fault handler, yet. Just shut text off
       LoadB  textOn,#FALSE        ;
       rts                                ;
```