The Hitchhiker's Guide to GEOS™

A Potpourri of Technical Programming Notes

(provided "as is" without support)

April 1988

Introduction

In 1986, Berkeley Softworks pioneered GEOS — the Graphic Environment Operating System — for the Commodore 64. GEOS offered the power of an icon/windowing operating system, once thought possible on the likes of Apple's Macintosh, to one of the world's lowest priced microcomputers. The computing community quickly recognized this innovation as significant: the Software Publisher's Association (SPA) gave GEOS a Technical Achievement Award and Commodore Business Machines endorsed it as the official operating system for the Commodore 64. Some industry critics even said it brought the Commodore 64 out of obsolescence. Since that time, GEOS has been ported to the Commodore 128 and, most recently, to the Apple II family of computers. Boasting an installed base approaching one-million units, GEOS not only promises to be around for some time, but to grow into the operating system for low-end computers

Why Develop GEOS Applications

GEOS provides an environment for programmers and software companies to quickly and efficiently develop sophisticated applications. GEOS insulates the programmer from the frustrating details and dirty work usually associated with application development. By using the GEOS facilities for disk file handling, screen graphics, menus, icons, dialog boxes, printer and input device support, the application can concentrate on doing what it does best, applying itself to the task at hand, using the GEOS system resources, routines, and user-interface facilities to both speed program development and build better programs.

Consistent User-interface

A very large portion of GEOS is devoted to supporting the user-interface. The GEOS interface has proven popular with thousands of users, and an application that takes advantage of this will likely be well received because the users will already be familiar with the basic program operation. Once a user has learned to operate geoWrite, for example, it is a smooth transition to another application such as geoCalc.

Large Installed Base and Portability

GEOS is currently available for three machines: the Commodore 64, the Commodore 128, and the Apple II. There are hundreds of thousands of owners who use GEOS on these machines and there is a correspondingly large demand for follow-on products. With careful programming, an application can be developed to run under all available system configurations with only minor changes. Berkeley Softworks plans to port GEOS to other 6502-based microcomputers, thereby further increasing the user base. As the popularity of GEOS grows, so does the market for your product.

Application Integration

GEOS offers a flexible cut and paste facility for text and graphic images. These photo scraps and text scraps allow applications to share data: a word processor can use graphics from a paint program and a graph and charting application can use data from a spreadsheet. The scrap format is standard and allows applications from different manufacturers to exchange data. Berkeley Softworks is currently developing a second-generation scrap facility for object-oriented graphics such as those used in desktop publishing and CAD programs.

Input and Output Technology

GEOS supports the concept of a device driver. A device driver is a small program which co-resides with the GEOS Kernal and communicates with I/O devices. Device drivers translates data and parameters from a generalized format that GEOS understands into a format relevant to the specific device. GEOS has *input drivers* for mice, joysticks, light pens, and other input devices, *printer drivers* for text and graphic output devices (including laser printers), and *disk drivers* for storage devices such as floppy disk drives, hard disks, and RAM expansion units (RAMdisks). As new devices become available, it is merely necessary to write a driver to support it.

What Exactly is GEOS?

First and foremost, GEOS is an operating system: a unified means for an application to interact with peripherals and system resources. GEOS is also an environment — specifically, a graphics-based user-interface environment offering a standard library of routines and visual-based controls, such as menus and icons. And finally, GEOS is a programmer's toolbox, providing routines for double-precision integer math, random-number generation, and memory manipulation..

NOTE:	*GEOS* as a general term can represent full range of concepts — an operating system, a user environment, the deskTop, a group of integrated applications — but in this book it usually refers specifically to the *GEOS Kernal*, the resident portion of the operating system with which the application deals with.

GEOS As an Operating System

College textbook writers are forever coming up with splendid new metaphors to describe operating systems. But as the coach of a baseball team or the governor of California, an operating system has the same basic function: it is the manager of a computer, providing facilities for controlling the system while isolating the application from the underlying hardware. An operating system allows the application to function in higher-level abstract terms such as "load a file into memory" rather than "let a bit rotate into the serial I/O shift register and send an acknowledge signal." The operating system will handle the laborious tasks of reading disk files, moving the mouse pointer, and printing to the printer.

GEOS provides the following basic operating system functions:

- Complete management of system initialization, multiple RAM banks, interrupt processing, keyboard/joystick/mouse input, as well as an application environment that supports dynamic overlays for programs larger than available memory, desk accessories, and the ability to launch other applications.
- A sophisticated disk file system that supports multiple drives, fast disk I/O, and RAM disks.
- Time-based processes. allowing a limited form of multitasking within an application.
- Printer output support, offering a unified way to deal with a wide variety of printers.

GEOS As a Graphic and User-Interface Environment

Interactive graphic interfaces have become the norm for modern day productivity. GEOS provides a services for placing lines, rectangles, and images on the screen, as well as handling menus, icons, and dialog boxes. Using the GEOS graphic elements make applications look better and easier to use.

GEOS provides the following graphic and user-interface functions:

- Multi-level dynamic menus which can be placed anywhere on the screen. GEOS automatically handles the user's interaction with the menus without permanently disrupting the display.
- Icons — graphic pictures the user can click on to perform some function.
- Complete dialog box library offering a standard set of dialog boxes (such as the file-selector) ready for use. The application may also define its own custom dialog boxes.
- A library of graphic primitives for drawing points, lines, patterned rectangles, and pasting photo scraps from programs like geoPaint.
- Sprite support. (Sprites are small graphic images which overlay the display screen and can be moved easily. The mouse pointer, for example, is a sprite.)
- A secondary screen buffer for undo operations.

GEOS As a Programmer's Toolbox

GEOS also contains a large library of general support routines for math operations, string manipulations, and other functions. This relieves the application programmer of the task of writing and debugging common routines ("re-inventing the wheel" as it were).

GEOS provides the following support routines:

- Double-precision (two-byte) math: shifting, signed and unsigned multiplication and division, random number generation, etc.
- Copy and compare string operations.
- Memory functions for initializing, filling, clearing, and moving.
- Miscellaneous routines for performing cyclic redundancy checks (CRC), initialization, error handling, and machine-specific functions.

Development System Recommendations

There are many ways to develop GEOS applications. Berkeley Softworks, for example, uses a UNIX™ based 6502 cross assembler and proprietary in-circuit emulators to design, test, and debug GEOS applications. Most developer's, however, will find this method too costly or impractical and will opt to develop directly on the target machines. Anticipating this, Berkeley

Softworks has developed geoProgrammer, an assembler, linker, debugger package designed specifically for building GEOS applications.

geoProgrammer

geoProgrammer is a sophisticated set of assembly language development tools designed specifically for building GEOS applications. geoProgrammer is a scaled-down version of the UNIX™ based development environment Berkeley Softworks actually uses to develop GEOS programs. In fact, nearly all the functionality of our microPORT™ system has been preserved in the conversion to the GEOS environment. All sample source code, equates, and examples in this book are designed for uses with geoProgrammer.

The geoProgrammer development system consists of three major components:

geoAssembler, the workhorse of the system, takes 6502 assembly language source code and creates linkable object files.

- Reads source text from geoWrite documents; automatically converts graphic and icon images into binary data.
- Recognizes standard MOS Technology 6502 assembly language mnemonics and addressing modes.
- Allows over 1,000 symbol, label, and equate definitions, each up to 20 characters long.
- Full 16-bit expression evaluator allows any combination of arithmetic and logical operations.
- Supports local labels as targets for branch instructions.
- Extensive macro facility with nested invocation and multiple arguments.
- Conditional assembly, memory segmentation, and space allocation directives.
- Generates relocatable object files with external definitions, encouraging modular programming.

geoLinker takes object files created with geoAssembler and links them together, resolving all cross-references and generating a runable GEOS application file.

- Accepts a link command file created with geoWrite.
- Creates all GEOS applications types (sequential, desk accessory, and VLIR), allowing a customized header block and file icon. geoLinker will also create standard Commodore applications which do not require GEOS to run.
- Resolves external definitions and cross-references; supports complex expression evaluation at link-time.
- Allows over 1,700 unique, externally referenced symbols.
- Supports VLIR overlay modules.

geoDebugger allows you to interactively track-down and eliminate bugs and errors in your GEOS applications.

- Resides with your application and maintains two independent displays: a graphics screen for your application and a text screen for debugging.
- Automatically takes advantage of a RAM-expansion unit, allowing you to debug applications which use all of available program space.
- Complete set of memory examination and modification commands, including memory dump, fill, move, compare, and find.
- Symbolic assembly and disassembly.
- Supports up to eight conditional breakpoints.
- Single-step, subroutine step, loop, next, and execute commands.
- RESTORE key stops program execution and enters the debugger at any time.
- Contains a full-featured macro programming language to automate multiple keystrokes and customize the debugger command set.

Commodore 64

GEOS was first implemented on the Commodore 64, and currently there are more GEOS applications for this system than the Apple II or the Commodore 128. The following is recommended for developing under this environment:

- Commodore 64 or 64c computer.
- Commodore 1351 mouse.
- At least one 1541 or 1571 disk drive.
- Commodore 1764 or 1751 RAM-expansion unit.
- GEOS supported printer.
- The basic GEOS operating system (GEOS 64), version 1.3 or later which includes geoWrite and geoPaint.
- geoProgrammer for the Commodore 64.

Commodore 128

The Commodore 128 may be the ideal environment for prototyping and developing GEOS applications because it can be used to create programs which run under GEOS 64 (in 64 emulation mode) and GEOS 128. The 128 sports a larger memory capacity, and geoProgrammer takes advantage of this extra space for symbol and macro tables. The following is recommended for developing under this environment:

- Commodore 128 computer.
- Commodore 1351 mouse.
- At least one 1541 or 1571 disk drive.
- Commodore 1764 or 1751 RAM-expansion unit.
- GEOS supported printer.
- The basic GEOS operating system (GEOS 64), version 1.3 or later which includes geoWrite and geoPaint.
- The basic GEOS 128 operating system, version 1.3 or later which includes geoWrite 128 and geoPaint 128.
- geoProgrammer for the Commodore 128.

Apple II

The Apple II is the latest addition to the GEOS family. The following is recommended for developing in this environment:

- Apple IIe, IIc, or IIgs computer.
- Apple Mouse card.
- At least one floppy disk drive.
- RAM card or hard disk.
- GEOS supported printer.
- The basic GEOS operating system (Apple GEOS), which includes geoWrite and geoPaint.
- geoProgrammer for the Apple II.

Other Useful GEOS Applications

In addition to those applications listed above, you may find the following useful:

- GEOS Icon Editor for customizing you icons.
- Photo Manager and Text Manager for cutting and pasting text and graphics to and from your geoProgrammer source code modules.
- Text Grabber for converting any 6502 source code you may have already written to geoWrite.

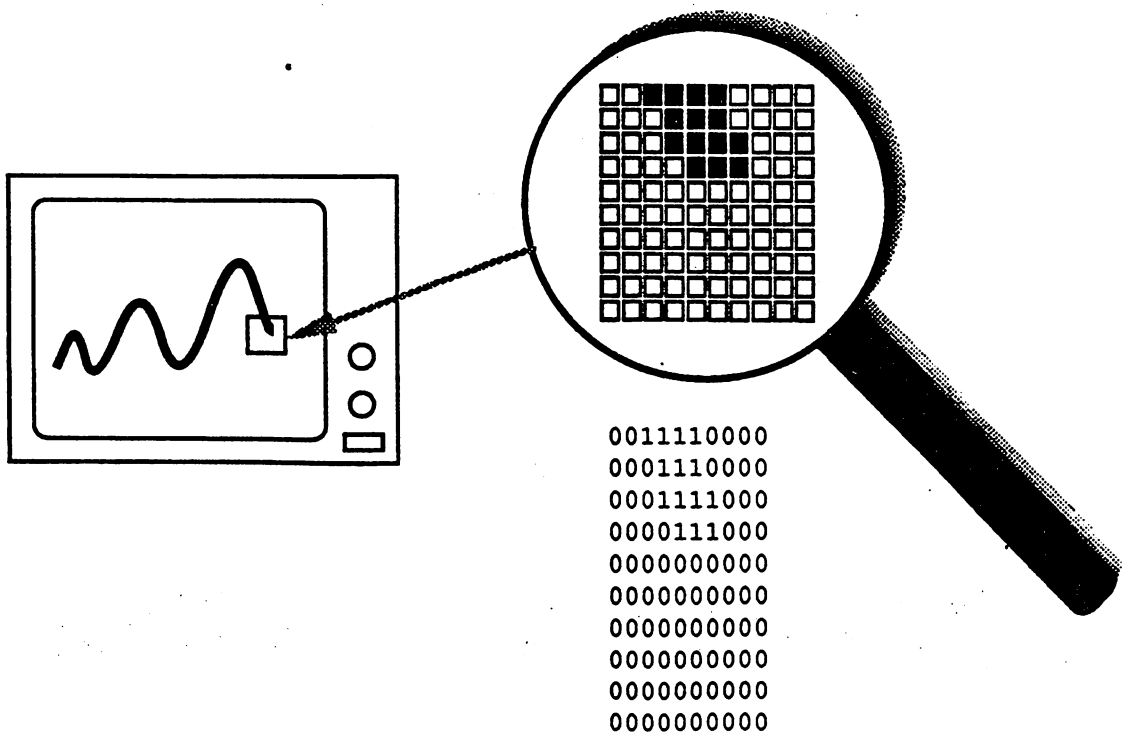
Graphic Routines

As the name GEOS (Graphics Environment Operating System) implies, screen graphics are central to both the operating system and its applications. GEOS provides a number of *graphic primitives* ("primitive" because they are the basis of more complex objects) for drawing points, lines, rectangles, and other objects, as well as displaying bitmap images such as those cut from geoPaint. GEOS also provides graphic support routines for undoing regions, inverting areas, scrolling,.and directly accessing the screen memory.

Drawing with the built-in GEOS routines increases program portability by making much of the internal, machine-dependent screen architecture transparent to the application. When you draw a line, for example, you merely supply the two endpoints. GEOS takes care of calculating the proper pixel locations and modifying the screen memory. This allows an application to use the same code to draw lines on machines with very different graphics hardware and spares the programmer from dealing directly with screen memory.

Introduction to GEOS Graphics

If you look closely at a monitor or television screen, you will notice that the image is made up of many small dots. These small dots, called *pixels*, can be either on or off and are represented in memory by 1's and 0's, respectively. A pixel with a value of one is considered *set* and a pixel of value zero is considered *clear*. This binary, or bitwise, representation of images is refered to as *bitmapped graphics*, and a *bitmap* is a picture or image created in this way.



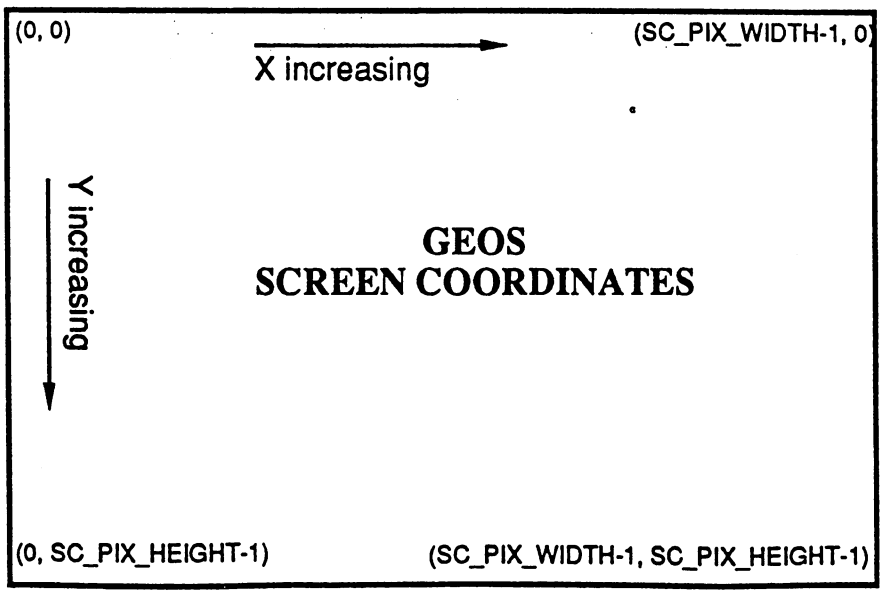
Color

Although some hardware configurations support color graphics, GEOS assumes that the screen is a monochromatic device; that is, GEOS only deals with one drawing color and one background color. Typically, the drawing color is black, like ink and the background color is white, like a piece of paper. Depending on the monitor being used and the Preference Manager settings, the actual displayed colors may be different. We will refer to the color displayed by a zero-pixel as the *background color* and the color displayed with a one-pixel as the *drawing color*. Applications that support multiple drawing colors, such as the Commodore 64 version of geoPaint, must do so on their own, bypassing GEOS (at the expense of portability) to provide multiple colors on the screen.

The GEOS Virtual Screen

The GEOS screen is often referred to as *virtual screen*, one whose layout and internal storage characteristics exist independent of any underlying graphics hardware. For this reason, the GEOS screen is fundamentally identical under all versions of the operating system.

The GEOS screen is a rectangular array of pixels arranged like a sheet of graph paper. Each pixel on the screen has a corresponding (x,y) coordinate. The x-axis begins with zero and runs horizontally (left to right) across the screen, and the y-axis begins with zero and runs vertically (top to bottom) down the screen. The maximum x- and y-positions, because they differ from machine to machine, are calculated by subtracting one from the GEOS constants **SC_PIX_WIDTH** and **SC_PIX_HEIGHT**.



Important:	GEOS does no clipping or range-checking on coordinates passed to it. If you pass it invalid data or coordinates, the results are unpredictable and will often crash the application.

GEOS 128 40/80-Column Support

Because applications that run under GEOS 128 may want to take advantage of both the 40- and 80-column screen modes, the following conventions have been adopted for the screen width and height constants:

- *The following constants can be used to access the dimensions of the 40- or 80-column screen specifically:*

SC_40_WIDTH	Pixel width of 40-column screen.
SC_40_HEIGHT	Pixel height of 40-column screen.
SC_80_WIDTH	Pixel width of 80-column screen.
SC_80_HEIGHT	Pixel height of 80-column screen.

- *If the application is designed to run under GEOS 128 only and not run under GEOS 64 (the **C64** constant is set to $00 and the **C128** constant is set to $01), then the standard **SC_PIX_WIDTH** and **SC_PIX_HEIGHT** constants take on the following values:*

SC_PIX_WIDTH	Pixel width of 80-column screen.
SC_PIX_HEIGHT	Pixel height of 80-column screen.

- *If the application is designed to run under GEOS 64 and GEOS 128 (both the **C64** constant and the **C128** constant set to $01), then the standard **SC_PIX_WIDTH** and **SC_PIX_HEIGHT** constants take on the following values:*

SC_PIX_WIDTH	Pixel width of 40-column screen.
SC_PIX_HEIGHT	Pixel height of 40-column screen.

This is because the application (typically) will be written with the 40-column screen in mind. At runtime, the application can check to see which version of GEOS it is running under and add doubling bits to the appropriate coordinate values so that the 40-column coordinates will be normalized automatically when GEOS 128 is in 80-column mode.

An application can use the following subroutine to determine whether it is running under GEOS 128 or GEOS 64:

```
.if (0)
*******************************************************************************
Check128:
       Check for GEOS 128.

Pass:
       nothing

Returns:
       st      minus flag set if running under GEOS 128.

Example usage:

       jsr     Check128
       bpl     10$             ;ignore if under GEOS 64
       jsr     DoDoubling      ;else, patch x-coordinates with doubling bits
10$:
```

```
;*******************************************************************************
.endif

Check128:
       lda     #$12            ; c128Flag not guaranteed to be valid in version 1.2 and lower
       cmp     version         ; first see if version <= 1.2
       bpl     10$             ; if so, branch and say C64. Note this is a signed comparison.
                               ; (it WILL NOT work if GEOS goes beyond version $7f!)
       lda     c128Flag        ; else set minus based on high bit c128Flag
10$:
       rts
```

When running under GEOS 128, the **graphMode** variable may be checked to determine whether GEOS is in 40- or 80-column mode:

```
       bit     graphMode               ; check 40/80 mode bits
       bpl     C64Mode                 ; branch if in 40-column mode
                                       ; else, handle as 80-column...
```

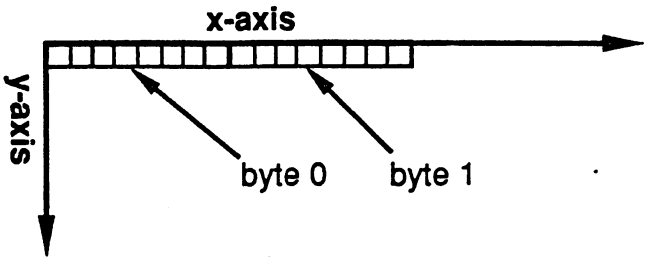
For more information, refer to "GEOS 128 X-poisition and Bitmap Doubling" in this chapter. Also see **NormalizeX** in the Routine Reference Section.

Inclusive Dimensions

All dimensions and GEOS coordinates are inclusive: a line contains the endpoints which define it, and a rectangle includes the lines that make up its sides. For example, a rectangle defined by an upper-left corner of (10,10) and a lower-right corner of (20,20) would include the lines around its perimeter defined by the points (10,10), (10,20), (20,10), and (20,20).

Linear Bitmap

For the purpose of bitmap compaction and patterns, the GEOS screen is treated as a *linear bitmap*, a contiguous block of bytes with each bit controlling an individual pixel. The bytes are lined up end-to-end for each screen line. The high-order bit (bit 7) of each byte controls the leftmost pixel and the low-order bit (bit 0) controls the rightmost pixel.



GEOS Virtual Screen

Keep in mind that this is a conceptual organization of the screen; the actual in-memory storage of the screen and bitmap data may be very different.

Dividing the Screen Into Cards

Many GEOS routines subdivide the GEOS virtual screen into 8x8-pixel blocks called *cards*. A card is a two-dimensional unit of measurement eight pixels on each side. The first card begins in the

upper-left corner of the screen (0, 0) and extends to (7, 7). The next card is just to the right of the first and extends from (8, 0) to (15, 7).

Cards are always aligned to eight pixel boundaries called *card boundaries* (pixel positions 0, 8, 16, 24, etc.). Aligning an object to a card boundary is called *card alignment*, and the position of an object expressed in cards is called its *card position*. Pixel position (32, 72), for example, would correspond to card position (4, 9) because 32/8 = 4 and 72/8 = 9). The *card width* of an object is its width in cards, and the *card height* is its height in cards. An entire row of cards is called a *cardrow*.

The card is a convenient unit of measurement because its dimensions, 8x8, which is a power of 2, lend themselves to simple binary arithmetic. For example, converting a pixel position to a card position is merely a matter shifting right three times.

Example:

```
*******************************************************************************
;MseToCardPos:
;converts current mouse positions to card position
;
;Pass:          nothing
;
;Uses:          MouseXPos, MouseYPos
;
;Returns:       r0L     mouse card x-position (byte)
;               r0H     mouse card y-position (byte)
;
;Destroys:      a,x,y
;                                                                        (mgl)
*******************************************************************************
MseToCardPos:
       php                          ; save current interrupt disable status
       sei                          ; disable interrupts so mouseXPos doesn't change
       MoveW   MouseXPos,r0         ; copy mouse x-position to zp work reg (ro)
       plp                          ; reset interrupt status asap.
       ldx     #r0                  ; divide x-position (r0) by 8
       ldy     #3                   ; (shift right 3 times)
       jsr     DShiftRight          ; this gives us the card x-position in r0L
       lda     MouseYPos            ; get mouse y-position
       lsr     a                    ; and shift right 3 times
       lsr     a                    ; which is a divide by 8
       lsr     a                    ; and gives us the card y-position in a
       sta     r0H                  ; set down card y-position
       rts                          ; exit
```

Cards are also convenient because they map directly to the internal storage format of the Commodore 40-column graphics screen. (Converting to other formats, such as the Commodore 128 80-column screen or the Apple II double hi-res screen, requires additional translation. This translation is handled automatically by the GEOS graphics routines.)

Display Buffering

Normally the application has control of the screen. But when an item such as a dialog box or a menu is displayed, GEOS overwrites the screen. When the dialog box is removed or the menu is retracted, GEOS needs to restore the portion of the screen it destroyed. For this purpose, GEOS

maintains a *background screen buffer*. Most of the time, the background buffer contains an exact copy of the *foreground screen* (the screen that is displayed) because GEOS normally sends graphics data to both screen buffers. When a temporary object is displayed, however, it is only drawn to the foreground screen. Removing the object, or *recovering* the original area of the screen, is then simply a matter of copying pixels from the background buffer to the foreground screen. The GEOS dialog box and menu routines handle this sort of recovery automatically.

dispBufferOn

Usually the application will want to draw to both buffers so that GEOS can properly recover the foreground screen after menus and dialog boxes. If graphics are only drawn to one buffer and a menu is brought down or a dialog box is displayed, the subsequent recover may restore the wrong data.

However, sometimes an application may want to limit drawing to only the foreground or background screen buffer. GEOS graphics and text routines use the global variable **dispBufferOn** to determine whether to draw to the foreground screen, the background buffer, or both simultaneously. Bits 6 and 7 of **dispBufferOn** determine the writing and reading mode:

bit 7:	1	— use foreground screen.
	0	— do not use foreground screen.
bit 6:	1	— use background buffer.
	0	— do not use background buffer.
bits 5-0:	*reserved for future use* — should always be 0.	

There are some constants which allow you to gain access to these bits:

ST_WR_FORE	use foreground.
ST_WR_BACK	use background.

and they can be used in following manner:

```
;Use both foreground screen and background buffer (normal).
       LoadB dispBufferOn, #(ST_WR_FORE | ST_WR_BACK)

;Use foreground screen only.
       LoadB dispBufferOn, #ST_WR_FORE

;Use background buffer only.
       LoadB dispBufferOn, #ST_WR_BACK
```

Note:	If bits 6 and 7 of **dispBufferOn** are both zero, GEOS considers this an undefined state and will not produce useful results. In most cases, the internal address calculations will force your graphic objects to appear in the center of the drawing area where they can do little harm. If the center line on the screen becomes garbled, **dispBufferOn** probably contains a bad value.

Using dispBufferOn

Typically applications leave **dispBufferOn** set to draw to both screens, whereas most desk accessories will only draw to the foreground screen. In some situations, an application may want to limit drawing to the foreground screen so that it may recover from the background buffer at a later time. Internally this is what GEOS does when it opens a menu or dialog box: the object is

only drawn to the foreground screen, and when it needs to be erased, the original data is recovered from the background buffer. **dispBufferOn** can also be used to pre-draw complex objects in the background buffer (**ST_WR_BACK**) and make them instantly appear on the foreground screen by doing a recover.

An application must take special precautions.when using **dispBufferOn** to draw selectively to one buffer or the other. For example, when GEOS automatically recovers from a menu or a dialog box, it recovers the data from the background buffer. If the background buffer has not been updated (the application has been drawing with the **ST_WR_BACK** bit cleared, for example), then the menu or dialog may recover the wrong data.

Since dialog boxes are only displayed when the the application calls **DoDlgBox** and menus are only opened while GEOS is in **MainLoop**, the application has some control over GEOS's automatic recovering. The application can postpone displaying dialog boxes and returning to **MainLoop** until the foreground screen and background buffer contain the same data. If an application *must* return to **MainLoop** while the buffers contain different data (to let processes run, for example), it can always disable menus by clearing the **MENUON_BIT** bit of **mouseOn**. The menus may be reenabled again by restoring the **MENUON_BIT** bit of **mouseON**:

Example:

— HEADER ?

```
StopMenus:
        MoveB   mouseOn,oldMouseOn      ; save current enable status for later
        rmbf    MENUON_BIT,mouseOn      ; disable menus temporarily
        rts                             ; exit.

RestartMenus:
        lda     oldMouseOn              ; get old menu enable status
        and     #(%1 <<MENUON_BIT)      ; ignore all but menu bit
        ora     mouseOn                 ; restore old menu bit
        sta     mouseOn                 ; in current mouseOn byte
        rts                             ; exit

oldMouseOn:     .byte   $00             ; temp save area for mouseOn variable
```

Using the Background Buffer as Extra Memory

Some applications are so starved for memory that they opt to use the background buffer for program code or data. To do this, they must always keep the **ST_WR_BACK** bit of **dispBufferOn** clear so that the background buffer is not corrupted with graphic data.

If you disable the background buffer, GEOS cannot automatically recover after menus and dialog boxes. The application must provide its own routine for restoring the foreground screen. There is a GEOS vector called **RecoverVector**, which normally points to the **RecoverRectangle** routine. Whenever GEOS needs to recover from a menu, dialog box, or desk accessory, it sets up parameters as if it were going to call **RecoverRectangle** and **jsr**'s indirectly through the address in **RecoverVector**. If the application is using the background buffer, it must place the address of its own screen recover routine in **RecoverVector**. When GEOS needs to recover a portion of the screen, it will **jsr** to the application's recover routine with the following register values describing the rectangular area to recover:

r3	X1 — x-coordinate of upper-left (word).
r2L	Y1 — y-coordinate of upper-left (byte).
r4	X2 — x-coordinate of lower-right (word).
r2H	Y2 — y-coordinate of lower-right (byte).

where (X1,Y1) is the upper-left corner and (X2,Y2) is the lower-right corner of the rectangular area to recover. The rectangle's coordinates are inclusive. The application must then use these values to restore the portion of the screen that lies within the rectangle's boundaries and return with an **rts**. This recovery can be as simple as filling with a halftoned pattern or as involved as redrawing graphic and text objects that fall within the rectangular recover area.

Most of the larger Berkeley Softworks GEOS applications use a technique called *saveFG/recoverFG* (short for "save foreground" and "recover background") to save and recover the foreground screen when displaying menus and dialog boxes. Basically, saveFG will save a rectangular subregion of the foreground screen to a special buffer just before GEOS displays a menu or a dialog box. When GEOS tries to recover from the background buffer, recoverFG restores the data from the special buffer. Although the size of the buffer varies from application to application, it will seldom be larger than 5.5K (just large enough to hold the largest standard dialog box).

Transferring data to and from the buffer is fairly straightforward. With the Commdore 40-column screen, it is mostly a matter of calculating the proper address offsets and copying bytes. With the GEOS 128 80-column screen, the process is complicated a bit because the bytes must be read from the VDC chip's RAM. With Apple GEOS, the process is simplicity itself because there are two routines for saving and restoring automatically: **saveFG** and **recoverFG**.

The real trick is knowing how to intercept the normal GEOS menu and dialog box drawing and recovering mechanisms. Dialog boxes are the easiest because they are always called by the application. The program only needs to save the foreground screen area prior to calling **DoDlgBox**. The size of the dialog box can be caculated from its table (be sure to account for any shadow) and the foreground data can be copied into the saveFG buffer. When the dialog box is finished, GEOS will jsr through **RecoverVector**. The application installs its own recoverFG routine into **RecoverVector** and restores the foreground area from the saveFG buffer. The GEOS dialog box recovery does have one quirk that concerns shadowed dialog boxes. GEOS shadowed dialog boxes consists of two overlapping rectangular areas: the actual dialog box and the slightly offset shadow rectangle. GEOS first calls through **RecoverVector** once for the region bounded by the shadow box, then again for the region bounded by the dialog box. When saving the foreground area, the entire dialog box region (the area bounded by the union of all eight corner points) should be saved and a special flag should be set so that the area is only recovered once. Under Apple GEOS, the recovery of dialog box shadows can be suppressed by setting **recoverOnce** to a non-zero value. When **recoverOnce** is non-zero, GEOS only vectors through **RecoverVector** once with the bounding rectangle of the dialog box. The application's recover routine will need to compensate for the shadow box. For more information on dialog boxes, refer to Chapter @DLG@.

Saving the foreground area before a menu is displayed is a bit tougher because GEOS displays menus at **MainLoop**, the application has little notice that a submenu is opening up. Fortunately, there is a workaround: GEOS supports a special type of sub-menu called a *dynamic sub-menu.* Just before a dynamic sub-menu opens, GEOS calls a subroutine whose address is stored in the menu data structure. This opporunity can be used to save the foreground screen area before GEOS draws the menu by calculating the bounding rectangle from the menu structure. When GEOS recovers a menu, it calls through **RecoverVector** as it does with dialog boxes. With multiple sub-menus, the menus are always recovered in the reverse order they were drawn. For more information on menus, refer to Chapter @ICNMENU@

Manual Imprinting and Recovering

Within an application, data can be moved between the foreground screen and background buffer with GEOS routines that copy data to and from the two areas. Copying data from the foreground screen to the background buffer is called *imprinting*, and copying data from the background buffer to the foreground screen is called *recovering*. There are GEOS routines for imprinting and recovering points, lines, and rectangular regions.

Some Possible dispBufferOn Complications

When drawing with both buffers enabled (with both foreground and background bits set in **dispBufferOn**), GEOS requires that the foreground screen and the background buffer contain exactly the same data. If they are different, the results of graphic operations may be unpredictable. If you need to draw to the foreground screen and the background buffer when they contain different data, you must perform the graphic operation once by writing only to the foreground screen, and then a second time, writing only to the background buffer — you cannot write to both of screen areas simultaneously if they contain different data.

Machine Dependencies

The GEOS graphics routines hide much of the underlying hardware from the application. This allows the same code to run under a variety of different environments with very few changes. However, it is sometimes necessary to optimize graphic routines for a specific machine. This can be as simple as taking advantage of color display capabilities or as complex as direct screen memory manipulation. Either way, an application should only resort to such tactics when the desired effect cannot be achieved through the standard graphics routines. Be aware that circumventing the GEOS Kernal will very likely increase your development time and that there is no guarantee that the techniques will be compatible with future versions of GEOS.

Commodore 64

The Commore 64 version of GEOS uses the standard high-resolution bitmap mode (not multi-color bitmap mode), which is 320 pixels wide by 200 pixels high. Memory is mapped to the screen in eight-byte stacks called *cards*: byte 0 controls pixels (0,0) through (7,0), with bit 7 on the left and bit 0 on the right, and byte 1 controls the same pixels on the line below, which is pixels (0,1) through (7,1). This stacking continues through byte 7, which controls pixels (0,7) through (7,7) and completes the 8x8-pixel card. Byte 8 begins the next card, controlling pixels (8,0) through (15,0). The screen memory begins at **SCREEN_BASE** and occupies 8,000 bytes, extending to **SCREEN_BASE+7999**. The background buffer begins at **BACK_SCR_BASE** and extends to **BACK_SCR_BASE+7999**.

GEOS does not directly support the foreground and background color options of the standard high-resolution bitmap mode. The color matrix, located from **COLOR_MATRIX** to **COLOR_MATRIX+999**, is set to a constant foreground and background color as determined by the Preference Manager. If an application wants to support color (like geoPaint), it must manage the color matrix itself. Each byte in the color matrix sets the foreground and background colors of a card (8x8 pixel block): color byte 0 sets the colors for card 0 (bitmap bytes 0-7) and color byte 1 sets the colors for card 1 (bitmap bytes 8-15). Before the application exits, it must restore the original color matrix. This best done by saving the first byte and then filling the color matrix before calling **EnterDeskTop**, as the following code fragments illustrate:

Example:

```
;On entry, save off the first byte of the color matrix
       MoveB  COLOR_MATRIX,saveColor
       .
       .
       .
;On exit, fill the color matrix with the saved value
       LoadW  r0,#1000              ;color matrix is 1000 bytes
       LoadW  r1,#COLOR_MATRIX
       MoveB  saveColor,r2L ;fill with original color
       jsr           FillRam
```

Commodore 128

In 40-column mode, GEOS 128 screen memory is identical to the Commodore 64. In 80-column mode, GEOS 128 uses the high-resolution 640x200 mode supported by the 8563 VDC (Video Display Controller) chip.. The foreground screen memory is not stored in the normal Commodore memory but on the VDC chip instead. The VDC RAM is accessed indirectly through the VDC control registers. The screen occupies 16,000 bytes, and each byte is accessed one at time by its address within the VDC display RAM(the first screen byte is at 0, the last at 15999). Bits are mapped sequentially from memory to the screen pixels: bits 7 through 0 of byte 0 (in that order) control the first seven pixels, (0,0) through (7,0). The following byte controls the next seven pixels, (8,0) through (15,0). And so on for the remainder of the screen. The following two subroutines will access bytes in the VDC screen RAM when GEOS 128 is in 80-column mode:

Example:

```
;*******************************************************************************
;
;Sta80Fore     -- stores byte to 128 80-column foreground screen
;Lda80Fore     -- loads byte from 128 80-column foreground screen
;
;Pass:
;       r5      = address in foreground memory
;       A       = data value (for Sta80Fore)
;
;Returns:
;       A       = data value (for Lda80Fore)
;
;Destroyed:
;       x
;
;Note: Call TempHideMouse to disable software sprites before accessing
;       foreground screen directly.
;                                                                        (mgl)
;*******************************************************************************
;Constants for VDC internal registers
VDC_HI_UPDATE          = 18          ;update hi-byte of VDC pointer
VDC_LO_UPDATE          = 19          ;update lo-byte of VDC pointer
VDC_DATA               = 31          ;data byte at current VDC pointer

Sta80Fore:
; Send data byte to the VDC chip
       jsr     NewVDCAddress         ; Update VDC address with fg screen pointer (r5)
       ldx     #VDC_DATA             ; request VDC data register
       stx     VDC                   ;
30$:   bit     VDC                   ; test VDC status
       bpl     30$                   ; loop till VDC ready for data byte
       sta     VDC+1                 ; store data byte
       rts                           ; exit
```

```
Lda80Fore:
; Get data byte from the VDC chip
       jsr    NewVDCAddress          ; Update VDC address with fg screen pointer (r5)
       ldx    #VDC_DATA              ; request VDC data register
       stx    VDC                    ;
30$:   bit    VDC                    ; test VDC status
       bpl    30$                    ; loop till data byte ready
       lda    VDC+1                  ; get data byte
       rts                           ; exit

NewVDCAddress:
; Transfer value in r5 to VDC internal hi/lo address register.
; Destroys: x
       ldx    #VDC_HI_UPDATE         ; ask VDC for high byte
       stx    VDC                    ;
10$:   bit    VDC                    ; check VDC status
       bpl    10$                    ; and loop till VDC ready
       ldx    r5H                    ; store hi-byte of address
       stx    VDC+1                  ; to VDC chip
       ldx    #VDC_LO_UPDATE         ; ask VDC for low-byte
       stx    VDC                    ;
20$:   bit    VDC                    ; check VDC status
       bpl    20$                    ; and loop till VDC ready
       ldx    r5L                    ; store lo-byte of address
       stx    VDC+1                  ; to VDC chip
       rts                           ; exit
```

For more information on controlling the 8563 VDC chip, refer to the *Commodore 128 Programmer's Reference Guide.*

Before writing directly to the 80-column foreground screen, be sure to call **TempHideMouse** to temporarily disable the virtual sprites (for more information, refer to **TempHideMouse** in Chapter XX).

Because the 80-column screen requires a 16,000-byte background buffer, GEOS 128 (when in 80-column mode) uses the 8,000-byte 40-column screen foreground buffer (**SCREEN_BASE** to **SCREEN_BASE+7999**) for store the first 100 scanlines of background buffer data and the 8,000-byte foreground screen buffer (**BACK_SCR_BASE** to **BACK_SCR_BASE+7999**) to store the last 100 scanlines of background buffer data. Because these data areas are not contiguous, an application that directly accesses the background screen must compensate for this break.

Apple II

Apple GEOS uses the double hi-res screen, which is 560 pixels wide by 192 pixels high. The seven lower bits (0-6) of each graphic byte are displayed in bit 0 to bit 6 order on the screen, and bit 7 is ignored (not displayed). That is, Pixel (0,0) is controlled by bit 0 of byte 0, pixel (1,0) is controlled by bit 1 of byte 0, and pixel (7,0) is controlled by bit 0 of byte 1. The graphic screen is located in memory at **SCREEN_BASE** ($2000) to **SCREEN_BASE+$1fff** ($3fff) in both the main and auxiliary memory banks. The bytes in main memory are mapped to odd byte positions on the screen (bytes 1,3,5...), and the bytes in auxiliary memory are mapped to even byte positions on the screen (byte 0,2,4...). This means that adjacent bytes on the screen are in separate banks of memory. For example, byte 0 is located at $2000 in the auxiliary bank and byte 1 is located at $2000 in the main memory bank. (For more information on accessing the Apple foreground screen across memory banks, refer to **GetScanLine** in the Routine Reference Section.)

Apple GEOS uses over 7K of tables to efficiently map pixel positions to screen memory bytes, thereby avoiding time-consuming shift and convert algorithms. If an application writes directly to screen memory, keep in mind that it will need to handle this pixel conversion manually. Also, before writing directly to the Apple screen, be sure to call **TempHideMouse** to temporarily disable the virtual sprites (for more information, refer to **TempHideMouse** in Routine Reference Section).

Porting Considerations and Techniques

Outside of the normal considerations for porting a GEOS application from one machine to another, there are a few additional elements which pertain specifically to graphics.

Apple GEOS and GEOS 128 Virtual Sprites

Apple GEOS and GEOS 128 (in 80-column mode) render sprites entirely in software by modifying the actual bitmap screen. (GEOS 64 and GEOS 128 in 40-column mode, use the hardware sprite capabilities of the VIC chip.) In order to properly treat these *virtual sprites* as if they were apart from the bitmap screen, they must be erased before any graphic operation, whether drawing, testing, imprinting, or recovering, is done. To do this, Apple GEOS and GEOS 128 provide the **TempHideMouse** routine to temporarily remove all sprites. The sprites are not redrawn until the application returns to **MainLoop**. Normal GEOS graphics and text routines will automatically call **TempHideMouse**; only applications that are directly accessing the foreground screen area need call **TempHideMouse**. For more information, refer to **TempHideMouse** in the Routine Reference Section "Software Sprites" in Chapter @SPRITE@.

GEOS 128 X-position and Bitmap Doubling

Because the GEOS 128 80-column bitmap screen has a horizontal resolution exactly twice that of GEOS 64 (640 vs. 320), GEOS 128 supports the ability to automatically double the x-coordinate(s) of graphic and text objects, and the width of bitmap objects, by setting special bits in the x-position and width calling parameter(s). This allows the visual elements of a GEOS 64 application to run in 80-column mode under GEOS 128 with a minimum of effort. The special bits can also be added at run-time to dynamically configure a program to run correctly under both GEOS 64 and GEOS 128. X-position and bitmap doubling is supported by nearly every GEOS 128 routine that writes to the screen (including text, dialog box, and icon routines).

The following constants may be bitwise or'ed into GEOS 128 x-coordinates and bitmap widths to take advantage of the automatic 80-column doubling features:

DOUBLE_W	For doubling word-length values. Normal x-coordinates, such as those passed to **Rectangle** and **DrawPoint**.
DOUBLE_B	For doubling byte-length values. A byte-length value is either a card x-position or a card width, both of which apply almost exclusively to bitmap routines, such as **BitmapUp** and **BitmapClip**.
ADD1_W	Used in conjunction with **DOUBLE_W**; adds one to a doubled word-length value. This allows addressing odd-coordinates, as when drawing a one-pixel frame around a filled rectangle.
ADD1_B	Used in conjunction with **DOUBLE_B**; adds one to a doubled byte length value.

These doubling bits have no effect when GEOS 128 is in 40-column mode but come to life when GEOS 128 is in 80-column mode. For example, the following code fragment will frame a filled rectangle. It will appear similarly in both 40- and 80-column modes.

Example:

```
X1      = 35                    ;left edge
X2      = 301                   ;right edge
Y1      = 40                    ;top edge
Y2      = 100                   ;bottom edge

;Draw a filled rectangle using the current pattern
        jsr     i_Rectangle                     ;inline call
        .byte   Y1                              ;y1
        .byte   Y2                              ;y2
        .word   (X1|DOUBLE_W|ADD1_W)            ;x1 with doubled width + space on left for frame
        .word   (X2|DOUBLE_W)                   ;x2 with doubled width
        jsr     i_FrameRectangle                ;inline call
        .byte   Y1                              ;y1
        .byte   Y2                              ;y2
        .word   (X1|DOUBLE_W)                   ;x1 with doubled width
        .word   (X2|DOUBLE_W|ADD1_W)            ;x2 with doubled width + offset for frame
        .byte   $ff                             ;solid line pattern
        rts                                     ;exit
```

NOTE: GEOS 128 filters all word-length x-coordinates (but not widths or byte-length x-coordinates) through the routine **NormalizeX** to process the doubling. For more detailed information on how this routine works, refer to its documentation in this chapter. **NormalizeX** will also double signed x-coordinates. If the x-coordinate is a signed number (like you might pass to **SmallPutChar**), then the double bits must be exclusive-or'ed into the x-coordinate parameters rather than simply or'ed.

The graphic elements of existing GEOS 64 applications can be ported to run under GEOS 128 with a minimum of effort by taking advantage of the GEOS 128 doubling bits. However, once the doubling bits have been installed, the application will no longer run under GEOS 64. The simplest approach to this problem is to have two entirely different applications. One designed to run under GEOS 64 and the other designed to run under GEOS 128. The doubling bits may be controlled at assembly-time with conditional assembly, as the following example illustrates.

Example:

```
DblDemo1:
;Will assemble differently depending on the status of the C64 and C128 assembly
;constants. If assembling for GEOS 64, doubling constants will be set to zero so
;that they will not affect the x-positions. If assembling for GEOS 64, doubling
;constants will be set according to geosConstants file so that graphic operations
;will double automatically in 128 mode.

.if     (C128 ^^ C64)               ;C64/C128 flags must be mutually exclusive!

        .if     !C128               ;if not assembling for GEOS 128, force doubling
                                    ;constants to harmless values so GEOS 64 graphics
                                    ;routines don't get confused.
                DOUBLE_B        = $00
                ADD1_B          = $00
                DOUBLE_W        = $0000
                ADD1_W          = $0000
        .endif

BM_XPOS         = (32/8)            ;byte x-position of bitmap (40-col)
BM_YPOS         = 20                ;y-position of bitmap

Bitmap:
```

MOVED TO CONSTANTS FILE

```
BM_WIDTH        = PicW          ;byte bitmap width (40-col)
BM_HEGHT        = PicH          ;bitmap height

FPATTERN        = %11111111     ;pattern for surrounding frame

DoBmap:

;Place the bitmap on the screen, loading the registers with
;inline data (note double-width settings).
        jsr     i_BitmapUp                  ;inline call
        .word   Bitmap                      ;bitmap address
        .byte   (BM_XPOS|DOUBLE_B)          ;xpos
        .byte   BM_YPOS                     ;ypos
        .byte   (BM_WIDTH|DOUBLE_B)         ;width
        .byte   BM_HEIGHT                   ;height

90$:    rts             ;exit

.else   ;(both C128 & C64 constants were both true or both false)
        .echo "DblDemo routine designed to assemble for both GEOS 64 and GEOS 128!"
.endif
```

Designing an application so that it runs well under both GEOS 64 and GEOS 128 is a more difficult task. It usually involves using self-modifying code: part of the initialization code for each module can check the version of GEOS it is running under (use the **Check128** subroutine illustrated in "GEOS 128 40/80-Column Support" in this chapter) and add the proper doubling-bits to all relevant x-coordinates.

Apple Bitmap Doubling and Aux-memory Bitmaps

Apple GEOS supports the ability to automatically double the width of bitmap objects by setting special bits in the x-width calling parameter(s). This allows GEOS 64 bitmaps to be converted to Apple GEOS with a minimum of effort. By doubling the width, a similar appearance can be maintained. Apple GEOS bitmap routines can also specify whether the bitmap data is in main memory or auxiliary memory by setting special bits in the x-position parameters. Bitmap doubling and aux-memory specification applies to the following routines and any other higher-level routines which depend on these for placing bitmaps on screen (such as **DoIcons**):

- **BitmapUp**
- **NewBitUp**
- **BitmapClip**
- **NewBitClip**
- **BitOhterClip**
- **NewBitOtherClip**

Because Apple GEOS allows widths specified by byte values (as in **BitmapUp**) and widths specified by word values (as in **NewBitUp**), there are different bits and constants to use for doubling the width, depending on the number of bytes (one or two) in the parameter. To double the width of a bitmap, bitwise-or one of the following constants into the width parameter:

DOUBLE_W	For doubling word-length values.
DOUBLE_B	For doubling byte-length values.

To force Apple GEOS to grab the bitmap data from auxiliary memory, bitwise-or the bitmap x-position with one of the following values, depending on whether the x-position is a byte-length or word-length parameter:

INAUX_B	for byte-length x-position parameters.
INAUX_W	for word-length x-position parameters.

For more information on the bits to set for bitmap doubling and auxiliary memory specificatiion, refer to the documentation of the specific routines in Routine Reference Section.

Example:

```
;Put a bitmap up, using an address in auxiliary memory and doubling its width

        LoadW   r0,#MyAuxBitmap             ;aux address of bitmap
        LoadW   r3,#(MY_XPOS|INAUX_W)       ;x-position + in-aux flag
        LoadB   r1H,#MY_YPOS                ;y-coordinate
        LoadB   r2,#(MY_CWIDTH*8)|DOUBLE|W  ;width = card width * 8 + doubling bit
        LoadB   r1L,#MY_HEIGHT              ;height
        jsr     NewBitUp                    ;put bitmap on screen
```

Points and Lines

Points

The simplest graphic operation involves setting, clearing, or testing the state of an individual pixel, or point, on the screen. GEOS provides two routines for working with points:

• **DrawPoint**	Set or clear a single point.
• **TestPoint**	Test a single point: is it set or clear?

Horizontal and Vertical Lines

Due to the rectangular nature of bitmapped graphics, horizontal and vertical lines are inherently fast and easy to create and manipulate. GEOS provides five routines for working with horizontal and vertical lines:

• **HorizontalLine**	Draw a horizontal line with a repeating bit pattern.
• **VerticalLine**	Draw a vertical line with a repeating bit pattern.
• **InvertLine**	Invert the pixels in a horizontal line.
• **ImprintLine**	Imprint a horizontal line to the background buffer.
• **RecoverLine**	Recover a horizontal line from the background buffer.

Line Patterns.

Both **HorizontalLine** and **VerticalLine** use a byte-sized bit pattern when creating the line. Each bit in the pattern byte represents a pixel in the line: wherever a one appears in the pattern byte, the corresponding pixel will be set, and wherever a zero appears , the corresponding pixel will be cleared. This allows lines which vary from solid (all 1's) to dashed (a mixture of 1's and 0's) to clear (all 0's). Note: this concept of a line-pattern is different from the 8x8 GEOS fill patterns used for rectangles.

Bits in the pattern byte are used left-to-right for horizontal lines and top-to-bottom in vertical lines, where bit 7 is at the left and the top, respectively. A bit pattern of %11110000 would create a horizontal line like:

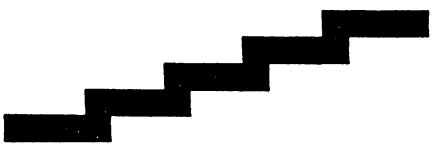
and a vertical line like:

The pattern byte is always drawn as if aligned to an eight-pixel boundary. If the endpoints of a line do not coincide with eight-pixel boundaries, then bits are masked off the appropriate ends. The effect of this is that a pattern is always aligned to specific pixels, regardless of the endpoints and that adjacent lines drawn in the same pattern will line up. That is, positions 0, 8, 16, 24, etc. will always depend on pattern bit 7, and positions 1,9, 17, 25, etc. will always depend on pattern bit 6.

NOTE:	Because of the internal memory layout of screen memory, horizontal lines will often draw up to eight times faster than vertical lines.

Diagonal Lines

For the same reason that bitmap displays are well-suited for displaying horizontal and vertical lines, they are ill-suited for displaying diagonal lines. A smooth, even-density line cannot be drawn diagonally between two points (except at 45-degree angles) — the points on the line must be approximated in a stairstep fashion:



GEOS provides one routine for drawing and recovering a line between two arbitrary points:

• DrawLine	Draw or recover a line between any two points.

DrawLine does not utilize a pattern byte; it will either set or clear all pixels between the two endpoints.

NOTE:	**DrawLine** is the most general-purpose drawing routine. It can be used to draw single points (both endpoints the same), horizontal and vertical lines, or lines at arbitrary angles. However, it is burdened by this flexibility, making it appreciably slower than the other plotting routines.

Patterns and Rectangles

Fill Patterns } KEEP W/ NEXT ¶

GEOS uses two types of patterns: line patterns and fill patterns. A line pattern is a one-byte repeating pixel pattern used by routines like **HorizontalLine** and **VerticalLine**, and a fill pattern is an 8x8 pixel block represented by eight bytes in memory and used by routines like **Rectangle**. Line patterns are discussed in "Points and Lines" earlier in this chapter. Fill patterns are discussed here.

A 50% fill pattern might be defined by the following:

```
.byte   %10101010
.byte   %01010101
.byte   %10101010
.byte   %01010101
.byte   %10101010
.byte   %01010101
.byte   %10101010
.byte   %01010101
```

The pattern has alternating set and clear pixels. Drawing a filled rectangle in this pattern would produce a medium-dark block.

All versions of the GEOS Kernal contains the following predefined patterns:

Apple GEOS contains an additional, user-defined pattern which is left for the application to modify.

Fills occur in the current pattern. The current pattern can be changed with the following routine:

• **SetPattern**	Set the current pattern.

To use one of the system patterns, the application would first call **SetPattern** with the appropriate pattern number. **SetPattern** calculates the proper pattern address, the address of the eight-byte

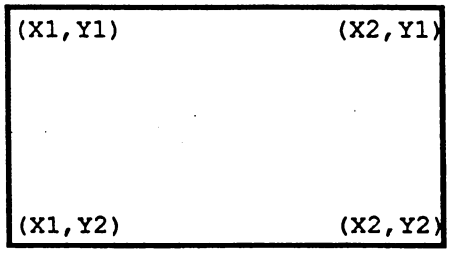
block, and places it in the GEOS variable **curPattern** (formerly **currentPattern**). Any subsequent call to a routine which uses a system pattern will index off of the address in **curPattern** to access the 8x8 block. Some applications, finding the need to define their own patterns, modify either the address in **curPattern** to point to their own eight-byte pattern or use the address in **curPattern** (after a valid call to **SetPattern**) to modify the GEOS system patterns directly. This technique will work on both GEOS 64 and GEOS 128, but will not on Apple GEOS because the patterns are stored in a fairly inaccessible portion of memory. Apple GEOS provides two routines for accessing and redefing patterns:

• **GetPattern**	Download a GEOS pattern to an eight-byte buffer.
• **SetUserPattern**	Upload an eight-byte buffer to a GEOS pattern.

NOTE: GEOS does not restore the system patterns when an application exits. If an application modifies the patterns, it should restore them when it exits unless it is desirable for the next application to inherit the redefined patterns (as with the GEOS Pattern Editor).

Rectangles

Rectangles in GEOS are defined by their upper-left and lower-right corners. The upper-left is usually referred to as (X1,Y1) and the lower-right as (X2,Y2), where X1, X2, Y1, and Y2 are valid x and y screen positions. From these two coordinates, the rectangle routines can determine the coordinates of the other two corners:



GEOS provides five routines for dealing with rectangular regions:

• **Rectangle**	Draw a solid rectangle using the current fill pattern.
• **FrameRectangle**	Draw an unfilled rectangle (bounding frame).
• **InvertRectangle**	Invert the pixels in a rectangular area.
• **ImprintRectangle**	Imprint a rectangular area to the background buffer.
• **RecoverRectangle**	Recover a rectangular area from the background buffer.

Bit-mapped Images

All graphic picture objects, such as icons and Photo Scrap images cut from geoPaint, are stored internally in GEOS Compacted Bitmap Format to save space. When you paste an image or icon into a geoProgrammer source file, it is in compacted bitmap format, and when you read a geoPaint image, it too is in compacted bitmap format. If a compacted image were to be copied directly to the screen, it would very likely be unrecognizable. GEOS bitmap routines first decompact the image and then transfer it to the screen area.

Standard Bitmap Routines

All versions of GEOS support the following bitmap routines:

• **BitmapUp**	Place a full compacted bitmap on the screen.
• **BitmapClip**	Place a rectangular subset of a compacted bitmap on the screen.
• **BitOtherClip**	Special version of **BitMapClip** which uses an application-defined routine to collect the compacted bitmap data a byte at a time, allowing the image to come from disk or other I/O device.

GEOS bitmaps are compacted from the GEOS virtual screen format rather than the internal machine format. Because the standard bitmap routines deal with byte-sized chunks (eight-pixels at a time), the following apply:

• Horizontally, the bitmap occupies pixels up to the nearest eight-pixel (byte) boundary. That is: a bitmap of five pixels is extended to eight and a bitmap of 30 pixels is extended to 32 pixels. Bitmaps which are not evenly divisable by eight (in the horizontal direction) are usually padded with zero bits.

• Bitmaps can only be placed at eight-pixel intervals on the x-axis (0, 8, 16...). This limitation does not apply to the y-axis.

Apple GEOS, however, provides two extended bitmap routines for overcoming these limitations. These routines might eventually be incorporated into GEOS 64 and GEOS 128.

Apple GEOS Extended Bitmap Routines

The Apple GEOS extended bitmap routines allow bitmaps of any pixel width and do not force the bitmap to be drawn on eight-pixel x-axis boundaries. These bitmaps are still compacted in byte-sized chunks, but by specifying a pixel width, any extra bits at the end of the last byte are ignored (masked out) when the bitmap is displayed. Apple GEOS offers three new bitmap routines:

• **NewBitUp**	Pixel width and positioning version of **BitmapUp**.
• **NewBitClip**	Pixel width and positioning version of **BitmapClip**.
• **NewBitOtherClip**	Pixel width and positioning version of **BitOtherClip**.

GEOS Compacted Bitmap Format

The GEOS compacted bitmap format relies on the observation that pixel patterns in bitmap images are frequently repetitive. If you were to examine a rectangular area of the screen (in GEOS linear bitmap format) it would often be the case that adjacent bytes would be identical. The compacted bitmap format encodes this redundancy into groups of bytes called *packets*. Each packet can decompress to a large number of bytes in the actual bitmap.

Packet Format

Each packet in a GEOS compacted bitmap follows a specific format. The first byte of each packet is called the **count** byte and is part of the *packet header*. Depending on its value, it has the following significance:

COUNT	(HEX)	SIGNIFICANCE
0	($00)	*reserved for future use.*
1 – 127	($00 – $7f)	***repeat:*** repeat the following byte **count** times. The total length of this packet is two bytes and decompresses to **count** bytes in the actual bitmap.
128	($80)	*reserved for future use.*
129 – 219	($81 – $db)	***unique:*** use the next **count**–128 bytes literally. The total length of this packet is (**count**–128)+1 or **count**–127 bytes and decompresses to **count**–128 bytes.
220	($dc)	*reserved for future use.*
221 – 255	($dd – $ff)	***bigcount:*** the next byte is a **bigcount** value in the range 2 through 255. The following **count**–220 bytes comprise data in ***repeat*** and ***unique*** format that should be repeated **bigcount** times. The total length of this packet depends on the decompacted size of the.***repeat*** and ***unique*** packets. A ***bigcount*** cannot containt another ***bigcount.***

Decompaction Walkthrough

Given the following compacted data:

```
.byte    25, 0, 133, 240, 220, 10, 0, 7, 224, 4, 3, 10, 5, 3
```

The decompaction routine would interpret it like this:

25, 0

repeat: the decompaction routine encounters the **count** value 25. Since it is in the range 1–127, the following byte (0), is repeated 25 times:

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

133, 240, 220, 10, 0, 7

unique: the next packet begins with a **count** of 133,which is in the range 129–219. The next 133–128 = 4 bytes are used once each:

240, 220, 10, 0, 7

224, 4, 3, 10, 5, 3

bigcount: the final packet begins with a **count** of 24 which is in the range 221–255. This signals a two byte header and the following byte, the **bigcount**, is 4. These two bytes are interpreted to mean repeat the next 224–220 = 4 bytes four times. The next four bytes, however, are expected to be in the ***unique*** and ***repeat*** compacted formats. In this case, its 3,10 (***repeat***: 10 three times) and 5,3 (***repeat***: 3 five times), which in turn are repeated four times:

10, 10, 10 ,3, 3, 3, 3, 3, 10, 10, 10 ,3, 3, 3, 3, 3, 10, 10, 10 ,3, 3, 3, 3, 3, 10, 10, 10 ,3, 3, 3, 3, 3

Compacting Strategy

The easiest way to compact a bitmap image is to let geoPaint do it for you by cutting the image out as a photo scrap and pasting it directly into your geoProgrammer source code. Sometimes this method is impractical and you will want to compress images directly from within an application. The following subroutine can be used to compact bitmap data:

```
.if (0)
*********************************************************************************************
BitCompact
*********************************************************************************************

DESCRIPTION:
        Converts linear bitmap data into compacted bitmap format, suitable
        for passing to routines such as BitmapUp.

        When compacting bitmaps directly from screen memory, the data must
        must first be converted from the internal screen format to linear
        bitmap format. The left edge of the source bitmap must start on a
        card boundary and the right edge must extend to the end of another
        card boundary. (Under Apple GEOS, strictly speaking, the ~~right~~ ~~edge~~ need
        not fully extend to a card boundary because the new bitmap routines
        (NewBitUp, NewBitClip, etc.) can mask bits at the right edge.)

        This bitmap data must then be converted to a linear format, where
        the first byte represents the first eight pixels of the upper-left
        corner of the bitmap, the next byte represents the next eight pixels
        and so on to the right edge of the bitmap. The byte following the
        last byte in a single line of a bitmap is the first byte of the next
        line. (The actual dimensions of the bitmap will be reconstructed from
        the WIDTH and HEIGHT parameters passed to the bitmap display routine.

        To convert from internal screen format to linear bitmap format:

                C64:    Set dispBufferOn appropriately (to reflect which
                        screen buffer to grap data from) and...

                  Cnvrt40:
                        ldx     yPos            ; get y coord of top of bitmap
                        jsr     GetScanLine     ; use it to calc screen ptrs
                        lda     xPos            ; get x pixel coord (lo byte)
                        and     #%11111000      ; strip off 3 bits for card x-position
                        clc                     ; Add card offset to
                        adc     r5L             ; base pointer (lo byte first)
                        sta     r5L
                        lda     xPos+1          ; (hi byte also)
                        adc     r5H
                        sta     r5H
                        ;At this point, (r5) points to the first byte in
                        ;the bitmap (upper-left corner).

                        Now step through each byte in this scanline by adding
                        8 to the pointer in r5 (compensating for the card architecture)
                        to get to the next byte, and repeat this process for each line
                        in the bitmap (incrementing yPos appropriately for each scanline).

                C128:   (40-column, same as C64; 80-column, read on...)
                        Conveniently, the 80-column data is already in linear bitmap
```

```
                format. The data will probably be coming from the
                background buffer because the foreground screen is
                entirely contained on the VDC chip's internal RAM and
                is difficult to access...

            Cnvrt80:
                bit     graphicsMode            ; make sure in 80-col mode
                bpl     Cnvrt40                 ; handle 40 like C64
                PushB   dispBufferOn            ; save current dispBuffer
                LoadB   dispBufferOn,#ST_WR_BACK ;force use of back buffer
                ldx     yPos                    ; get y coordinate
                jsr     GetScanLine             ; use it to calc screen ptrs
                Movew   xPos,r0                 ; copy x-position to zp work reg
                ldx     #r0                     ; divide r0 by 8
                ldy     #3                      ; (shift right 3 times)
                jsr     DShiftRight             ; this gives us the card offset
                AddW    r0,r6                   ; add card (byte) offset to scanline addr.
                ;At this point (r6) points to the first byte of the
                ;bitmap.

                Now step each byte in this scanline by adding
                1 to the pointer in r6 to get to the next byte,
                and repeat this process for each line in the
                bitmap (incrementing yPos appropriately).

        Apple:  Use the Apple GEOS Kernal routines ReadScanLine
                and ReadBackLine to convert the internal Apple
                screen format into linear bitmap format. Nice and simple.

CALLED BY:

PASSED:
        r0      Pointer to destination buffer to store compacted data
                (this buffer must be at least 1 and 1/64 of size of the
                uncompacted data because it is possible, but unlikely, that
                the compacted data will actually be larger than the ucompacted
                data).
        r1      Pointer to linear bitmap data to compact.
        r2      # of bytes to compact.

RETURNS:
        r0      Points to byte following last byte in compacted data.

DESTROYED:
        a,x,y,r1-r6

PSEUDO CODE / STRATEGY:
        Starts with the first source byte and counts the number of identical bytes
        following it to determine whether to generate a UNIQUE or REPEAT packet. If
        there are three or less identical bytes in a row, a UNIQUE packet is generated,
        four or more generates a REPEAT packet. The packet is placed in the destination
        buffer and this process is then repeated until all bytes in the source buffer
        have been compressed.

KNOWN BUGS / SIDE EFFECTS / IDEAS:
        Only uses the UNIQUE and REPEAT compaction types. The BIGCOUNT compaction type is
        such that it is difficult to determine the compaction payoff point. BIGCOUNT could
        be used to compress adjacent scanlines that are identical because this type of check
        would be trivial. The basic scanline could be compressed with UNIQUE and REPEAT, then
        duplicated by placing it inside a BIGCOUNT.

        This routine is not limited to compressing bitmap data. In fact, it works quite
```

well on any data where strings of identical bytes are common (e.g., fonts). It does not, for example, compress text very efficiently. A Huffman-based algorithm yields better results.

(mgl)

```
**********************************************************************************************
.endif

MAX_REPEAT      =       127             ; maximum repeat COUNT value
MAX_UNIQUE      =       91              ; maximum unique COUNT value
UNIQ_THRESH     =       3               ; byte count threshold, beyond which a REPEAT type
                                        ; should be used instead of UNIQUE.

BitCompact:
10$:
                                        ; r1 = current addr in source buffer
                                        ; r0 = current addr in destination buffer
                                        ; r2 = # bytes left in source
        jsr     CountRepeat             ; count the # of identical bytes here
        cmp     #UNIQ_THRESH            ; Enough repeats to justify REPEAT type?
        ble     20$                     ; No, go use UNIQUE
                                        ; yes, use REPEAT (A = # to repeat)
        sta     r5L                     ; store repeat # for later
        ldy     #0                      ; init. index into buffers
        sta     (r0),y                  ; store repeat # to destination
        lda     (r1),y                  ; get repeat value
        iny                             ; point to next byte in dest buffer
        sta     (r0),y                  ; store to destination buffer
        AddVW   #2,r0                   ; move up dest. pointer
        bra     100$                    ; exit.
20$:                                    ;
                                        ; use UNIQUE
        jsr     GetUnique               ; Calc # of unique bytes to use
                                        ; (A = number of unique)
        ldy     #0                      ; init. index into buffers.
        ora     #$80                    ; convert unique count to packet count value
        sta     (r0),y                  ; store to dest.buffer
30$:                                    ;
        lda     (r1),y                  ; get first unique value
        iny                             ; increment pointer pointer
        sta     (r0),y                  ; store to destination buffer
        cpy     r5L                     ; done yet? (r5L = repeat #)
        bne     30$                     ; loop till done copying
        inc     r5L                     ; convert to # to add to dest pointer
        AddBW   r5L,r0                  ; move up destination pointer
        dec     r5L                     ; correct back to # done
                                        ; fall through to exit
100$:                                   ;
        AddBW   r5L,r1                  ; move up source pointer
        SubBW   r5L,r2                  ; subtract off # left in source buffer
        lda     r2L                     ; check for zero bytes left
        ora     r2H                     ; more to do?
        bne     10$                     ; if so, loop
        rts                             ; else, exit.

CountRepeat:
                                        ; r1 = current pointer into source buffer
                                        ; r0 = current pointer into destination buffer
                                        ; r2 = number of bytes left in source
        ldy     #0                      ; initialize relative buffer index
        ldx     #0                      ; initalize current repeat count
                                        ;
```

```
        lda     (r1),y                  ; get first byte
        sta     r6L                     ; keep in r6L. This is the byte we're trying
                                        ; to match.
10$:                                    ;
        lda     r2H                     ; more than 255 bytes left in source?
        bne     20$                     ; if so, ignore # check
        cpx     r2L                     ; else, are we at the last byte?
        beq     90$                     ; if so, exit
20$:                                    ;
        cpx     #MAX_REPEAT             ; check repeat count with max # of repeats
        beq     90$                     ; if at maximum, branch to exit.
        lda     (r1),y                  ; does it actually match?
        cmp     r6L                     ; check against 1st byte
        bne     90$                     ; if no match, exit
        inx                             ; else, we found a match. increment repeat count
        iny                             ; move to next byte in source
;NOTE -- following branch changed to save a byte. y is never incremented to $00.
;       bra     10$                     ; and loop to check it
        bne     10$                     ; branch always... iny above will always clear z flag
90$:                                    ;
        txa                             ; return repeat count in A
        rts                             ; exit

GetUnique:
        PushW   r1                      ; Save orig pointer
        LoadB   r5L,#0                  ; start none unique
10$:                                    ;
        inc     r5L                     ; do one more unique
        ldx     r5L                     ; get # unique so far
        lda     r2H                     ; lots left?
        bne     20$                     ; if so, skip end check
        cpx     r2L                     ; all of them?
        beq     90$                     ; if yes, then that many
20$:                                    ;
        cpx     #MAX_UNIQUE             ; max # unique
        beq     90$                     ; if full, do them
        AddVW   #1,r1                   ; move up a byte
        jsr     CountRepeat             ; how many of the following bytes are repeats?
        cmp     #UNIQ_THRESH            ; Enough to warrant a REPEAT packet?
        ble     10$                     ; No, go stuff them in this UNIQUE packet
                                        ; Yes, close this UNIQUE packet.
90$:                                    ;
        PopW    r1                      ; retrieve start pointer
        lda     r5L                     ; get # to do unique
        rts
```

Direct Screen Access and Block Copying

Direct Screen Access

One purpose of an operating system such as GEOS is to insulate the application from the peculiarities of the machine it is running on, allowing the programmer to worry more abot how the application will function than how it will interact with the hardware. However, because of the complexity of GEOS graphics routines, it is sometimes necessary, for performance reasons, to bypass the operating system and manipulate the screen memory directly. Although this practice is not recommended — it increases portability problems, defeating much of the purpose of a GEOS

— it is a reality. And with that in mind, Berkeley Softworks built routines into GEOS to facilitate direct screen access. The following routine exists in all versions of the Kernal:

• **GetScanLine**	Calculate the address of the first byte of a particular screen line.

And these two additional routines exist in Apple GEOS:

• **GetScreenLine**	Copy a horizontal line (in internal format) from the screen to an application's buffer.
• **PutScreenLine**	Copy a horizontal line (in internal format) from an application's buffer to the screen.

GetScreenLine and **PutScreenLine** are intended to let the application directly access the Apple double hi-res screen without worrying about bank switching on alternate bytes. The screen is treated as a contiguous block, as if the alternating bytes in the two memory banks were actually adjacent in memory. These routines are not well-suited for scrolling large regions (they are too slow), but are sufficient for drawing with small brushes. For scrolling, use the block copy routines.

Linear Bitmap Conversion (Apple GEOS)

Because the Apple's pixel memory mapping scheme is so convoluted, Apple GEOS provides routines to convert a line of data (in Apple's internal format) to linear bitmap data, where pixels occupy contiguous bits in contiguous bytes:

• **ReadScanline**	Translates a screen line from the foregorund screen from Apple internal format to linear bitmap format.
• **ReacBackLine**	Translates a screen line from the background buffer from Apple internal format to linear bitmap format.

Although it would seem that routines which translate in the other direction — from linear bitmap format to Apple format — are necessary, **BitmapUp** can be used for this purpose. Merely prepend a 70 to the front of the linear bitmap and call **BitmapUp** as if the line were a bitmap 70 bytes wide and one pixel high.

Block Copy and Scrolling (Apple GEOS)

Apple GEOS also extends the direct screen access facilities by offering three block copy routines which are useful for scrolling and moving rectangular areas of the screen:

• **CopyLine**	Copies a horizontal line from one area of the foreground screen to another.
• **CopyScreenBlock**	Copies a rectangular area from one part of the screen to another.
• **CopyFullScreen**	Copies a rectangle of the full width of the screen (but of variable height) to another position vertically.

Although these routines deal directly with the screen, the screen architecture is actually transparent to the application. Therefore, future versions of GEOS may implement these functions.

Special Graphics Related Routines

GEOS provides a few graphics-related routines which don't fit nicely into any other category:

• **GraphicsString**	Execute a string of graphics commands.
• **DivideBySeven**	Quickly divide a screen coordinate by seven for direct screen access (Apple GEOS only).
• **NormalizeX**	Adjust an x-coordinate (under GEOS 128 only) to compensate for the higher-resolution 80-column mode.
• **SetNewMode**	Change GEOS 128 graphics mode (40/80-column).

Icons, Menus, and Other Mouse Presses

When the user clicks the mouse button, GEOS determines whether the mouse pointer was positioned over an icon, a menu item, or some other region of the screen. GEOS has a unique method of handling a mouse press for each of these cases. If the user pressed on an icon, GEOS calls the appropriate icon event routine. If the user pressed on a menu, GEOS opens up a sub-menu or calls the appropriate menu event routine, whichever is applicable. And if the user pressed somewhere else, GEOS calls through **otherPressVector**, letting the application handle (or ignore) these "other" mouse presses.

Icons

When you open a disk by clicking on its picture, delete a file by dragging it to the trash can, or click on the CANCEL button in a dialog box, you are dealing with *icons*, small pictorial representations of program functions. A GEOS icon is a bitmapped image, whether the picture of a disk or a button-shaped rectangle, that allows the user to interact with the application. When the application enables icons, GEOS draws them to the screen and then keeps track of their positions. When the user clicks on an icon, an *icon event* is generated, and the application is given control with information concerning which icon was selected.

Icon Table Structure

The information for all active screen icons is stored in a data structure called the *icon table*. GEOS only deals with one icon table at a time. The icon table consists of an *icon table header* and a number of *icon entries*. The whole table is stored sequentially in memory with the header first, followed by the individual icon entries.

Icon Table Header

The icon table header is a four byte structure which tells GEOS how many icons to expect in the structure and where to position the mouse when the icons are enabled. It is in the following format:

Icon Table Header:

Index	Constant	Size	Description
+0	OFF_I_NUM	byte	Total number of icons in this table.
+1	OFF_I_MX	word	Initial mouse x-position. If $0000, mouse positiion will not be altered.
+3	OFF_I_MY	byte	Initial mouse y-position.

This first byte reflects the number of icon entries in the icon table (and, hence, the number of icons that can be displayed). The table can specify up to **MAX_ICONS** icons.

The next word (bytes 1 and 2) is an absolute screen x-coordinate and the following byte (byte 3) is an absolute screen y-coordinate. The mouse will be positioned to this coordinate when the icons are first displayed. If you do not want the mouse positioned, set the x-coordinate word to $0000, which will signal **DoIcons** to leave the mouse positions alone.

Icon Entries

Following the icon table header are the icon entries, one for each specified in the **OFF_I_NUM** byte in the icon table header. Each icon entry is a seven-byte structure in the following format:

Icon Entries:

Index	Constant	Size	Description
+0	OFF_I_PIC	word	Pointer to compacted bitmap picture data for this icon. If set to $0000, icon is disabled.
+2	OFF_I_X	byte	Card x-position for icon bitmap.
+3	OFF_I_Y	byte	Y-position of icon bitmap.
+4	OFF_I_WIDTH	byte	Card width of icon bitmap.
+5	OFF_I_HEIGHT	byte	Pixel height of icon bitmap.
+6	OFF_I_EVENT	word	Pointer to icon event routine to call if this icon is selected.

The first word (**OFF_I_PIC**) is a pointer to the compacted bitmap data for the icon. The icon can be of any size (up to the full size of the screen). If this word is set to **NULL** ($0000), the icon is disabled.

The third byte (**OFF_I_X**) is the x byte-position of the icon. The x byte-position is the x-position in bytes — icons are placed on the screen by **BitmapUp** and so must appear on an eight-pixel boundary. The byte-position can be calculated by dividing the pixel-position by eight (x_byte_position = x_pixel_position/8).

The fourth byte (**OFF_I_Y**) is the pixel position of the top of the icon. The icon will be placed at (x_byte_position*8 , y_pixel_position).

The next two bytes (**OFF_I_WIDTH** and **OFF_I_HEIGHT**) are the width in bytes and height in pixels, respectively. These values correspond to the geoProgrammer internal variables **PicW** and **PicH** when they are assigned immediately after a pasted icon image.

The final word (**OFF_I_EVENT**) is the address of the icon event handler associated with this icon.

Sample Icon Table

The following data block defines three icons which are placed near the middle of the screen. The mouse is positioned over the first icon:

```
;*****************************************************************************************
;SAMPLE ICON TABLE
;*****************************************************************************************

;Icon positions and bitmap data

I_SPACE         = 1             ;space between our icons (in cards)

PaintIcon:




PAINTW = PicW
```

```
PAINTH = PicH
PAINTX = 16/8
PAINTY = 80

WriteIcon:




WRITEW = PicW
WRITEH = PicH
WRITEX = PAINTX + PAINTW + I_SPACE
WRITEY = PAINTY

PublishIcon:




PUBLISHW        = PicW
PUBLISHH        = PicH
PUBLISHX        = WRITEX + WRITEW + I_SPACE
PUBLISHY        = WRITEY

;The actual icon data structure to pass to DoIcons follows
IconTable:

I_header:
        .byte   NUMOFICONS                      ;number of icon entries
        .word   (PAINTX*8) + (PAINTW*8/2)       ;position mouse over paint icon
        .byte   PAINTY + PAINTH/2               ;

I_entries:
PaintIStruct:
        .word   PaintIcon                       ;pointer to bitmap
        .byte   PAINTX, PAINTY                  ;icon position
        .byte   PAINTW, PAINTH                  ;icon width, height
        .word   PaintEvent                      ;event handler

WriteIStruct:
        .word   WriteIcon                       ;pointer to bitmap
        .byte   WRITEX, WRITEY                  ;icon position
        .byte   WRITEW, WRITEH                  ;icon width, height
        .word   WriteEvent                      ;event handler

PublishIStruct:
        .word   PublishIcon                     ;pointer to bitmap
        .byte   PUBLISHX, PUBLISHY              ;icon position
        .byte   PUBLISHW, PUBLISHH              ;icon width, height
        .word   PublishEvent                    ;event handler

NUMOFICONS      = (*-I_entries)/IESIZE          ;number of icons in table

;Dummy icon event routines which do nothing but return
PaintEvent:
WriteEvent:
```

```
PublishEvent:
        rts
```

Installing Icons

When an application is first loaded, GEOS will not have an active icon structure. GEOS must be given the address of the applications icon table before **MainLoop** can display and track the user's interaction with them. GEOS provides one routine for installing icons

• DoIcons	Display and activate an icon table.

DoIcons draws the enabled icons and instructs **MainLoop** to begin watching for a single- or double-click on one. The icon table stays activated and enabled until the **ICONS_ON_BIT** of **mouseOn** is cleared or another icon table is installed by calling **DoIcons** with the address of a different icon structure. In either case, the old icons are not erased from the screen by GEOS.

DoIcons will draw to the foreground screen and background buffer depending on the value of **dispBufferOn**. Icons are usually permanent structures in a display and so often warrant being drawn to both screens. If icons are only drawn to the foreground screen, they will not be recovered after a menu or dialog box.

Example:

```
;*******************************************************************************************
;IconsUp        Draw and intialize our icons
;
;*******************************************************************************************
;
IconsUp:
        LoadB   dispBufferOn,#(ST_WR_FORE | ST_WR_BACK)     ;draw to both buffers
        LoadW   r0,#IconTable                               ;point to icon table
        jsr     DoIcons                                     ;install icons
        rts                                                 ;exit
```

Important:	Due to a limitation in the icon-scanning code, the application must *always* install an icon table with at least one icon. If the application is not using icons, create a dummy icon table with one icon (see below).

```
;*************************************************************************
;NoIcons        Install a dummy icon table. For use in applications that
;               aren't using icons. Call early in the initialization of the
;               application, before returning to MainLoop.
;
;*************************************************************************
;
NoIcons:
        LoadW   r0,#DummyIconTable      ;point to dummy icon table
        jmp     DoIcons                 ;install. Let DoIcons rts

DummyIconTable:
        .byte   1               ;one icon
        .word   $0000           ;dummy mouse x (don't reposition)
        .byte   $00             ;dummy mouse y
        .word   $0000           ;bitmap pointer to $0000 (disabled)
        .byte   $00             ;dummy x-pos
```

```
.byte   $00             ;dummy y-pos
.byte   1,1             ;dummy width and height
.word   $0000           ;dummy event handler
```

MainLoop and Icon Event Handlers

When the user clicks the mouse button on an active icon, GEOS **MainLoop** will recognize this as an icon event and call the icon event handler associated with the particular icon. The icon event handler is given control with the number of the icon in **r0L** (the icon number is based on the icon's position in the table: the first icon is icon 0). Before the event handler is called, though, **MainLoop** might flash or invert the icon depending on which of the following values is in **iconSelFlag**:

Constants for iconSelFlag:

ST_NOTHING	The icon event handler is immediately called; the icon image is untouched.
ST_FLASH	The icon is inverted for **selectionFlash** vblanks and then reverted to its normal state before the event handler is called.
ST_INVERT	The icon is inverted (foreground screen image only) before the event handler is called. The event handler will usually want to revert the image before returning to **MainLoop** by calculating the bounding rectangle of the icon, loading **dispBufferOn** with **ST_WR_FORE**, and calling **InvertRectangle**.

Detecting Single- and Double-clicks on Icons

When the user first clicks on an icon, GEOS loads the global variable **dblClickCount** with the GEOS constant **CLICK_COUNT**. GEOS then calls the icon event handler with **r0H** set to **FALSE**, indicating a single-click. **dblClickCount** is decremented at interrupt level every vblank. If the icon event handler returns to **MainLoop** and the the user again clicks on the icon before **dblClickCount** reaches zero, GEOS calls the icon event handler a second time with **r0H** set to **TRUE** to indicate a double-click.

Checking for a double-click or a single-click (but not both) on a particular icon is trivial: merely check **r0H**. If **r0H** is **TRUE** when you're looking for a single-click or its **FALSE** when you're looking for a double-click, then return to **MainLoop** immediately. Otherwise, process the click appropriately. This way, if the user single-clicks on an icon which requires double-clicking or double-clicks on an icon which requires single-clicking, the event will be ignored.

However, checking for both a double- or a single-click on the same icon (and performing different actions) is a bit more complicated because of the way double-clicks are processed: during the brief interval between the first and second clicks of a double-click, the icon event handler will be called with **r0H** set to **FALSE**, which will appear as a single-click; when the second press happens before **dblClickCount** hits zero, the icon event handler is called a second time with **r0H** set to **TRUE**, which will appear as a double-click. There is no simple way (using the GEOS double-click facility) to distinguish a single-click which is part of a double-click from a single-click which stands alone

There are two reliable ways to handle single- and double-click actions on icons: the *additive function method* and the *polled mouse method*. The additive function method relies on a simple single-click event which toggles some state in the application and a double-click event (usually more complicated) which happens *in addition to* the single-click event. The GEOS deskTop uses the additive function method for selecting (inverting) file icons on a single-click and selecting and opening them on a double-click. The icon event handler first checks the state of **r0H**. If it is

FALSE (single-click) then the icon (and an associated selection flag) is inverted. If it is **TRUE** (double-click) then the file is opened. If the user single-clicks, the icon is merely inverted. If the user double-clicks, the icon is inverted (on the first click) and then processed as if opened (on the second click).

Example:

```
;*******************************************************************************************
;       Icon double-click handler
;       additive function method
;
;*******************************************************************************************
IconEvent1:
        lda     r0H                     ;check double-click flag
        bne     10$                     ;branch if second click of a double-click
                                        ;else, this is a single-click or the
                                        ;first push of a double-click,
        jsr     InvertIcon      ;       ;so just invert the selection
        bra     90$                     ;and exit.
10$:
        jsr     OpenIcon                ;double-click detected, go process it
                                        ;fall through to exit
90$:
        rts                             ;return to MainLoop
```

The polled-mouse method can be used when the single-click and double-click functions are mutually exclusive. When a single-click is detected the icon event handler, rather than returning to **MainLoop** and letting GEOS manage the double-click, handles it manually by loading **dblClickCount** with a delay and watching **mouseData** for a release followed by a second click.

Example:

```
;*******************************************************************************************
;       Icon double-click handler
;       polled mouse method
;
;*******************************************************************************************
;
IconEvent2:
;User pressed mouse once, start double-click counter going
        LoadB   dblClickCount,#CLICK_COUNT   ;start delay

;Loop until double-click counter times-out or button is released
10$:
        lda     dblClickCount           ;check double-click timer
        beq     30$                     ;If timed-out, no double-click
        lda     mouseData               ;Else, check for release
        bpl     10$                     ;loop until released

;mouse was released,loop until double-click counter times-out or
;button is pressed a second time.
20$:
        lda     dblClickCount ;check double-click timer
        beq     30$                     ;If timed-out, no double-click
        lda     mouseData               ;Else, check for second press
        bmi     20$                     ;loop until pressed

;Double-click detected (no single-click)
30$:    jsr     DoDoubleClick           ;do double-click stuff
        bra     90$                     ;exit

;Single-click detected (no double-click)
```

```
        jsr     DoSingleClick           ;do single-click stuff
                                        ;and fall through to exit

;Exit
90$:    rts                             ;return to MainLoop
```

Note:	These techniques for handling single- and double-clicks are described here as they pertain to icons; they are not directly applicable to applications that detect mouse clicks through **otherPressVector**. When control vectors through **otherPressVector**, the value in **r0H** is meaningless. For more information on **otherPressVector**, refer to "Other Mouse Presses" in this chapter.

Other Things to Know About Icons

Icon Releases and otherPressVector

When the user clicks on an active icon, **MainLoop** will call the proper icon event routine rather than vectoring through **otherPressVector**. However, the routine pointed to by **otherPressVector** will get called when the mouse is released. Applications that aren't using **otherPressVector** can disable this vectoring by storing a $0000 into **otherPressVector** ($0000 is actually its default value). Applications that depend on **otherPressVector**, however, can check **mouseData** and ignore all releases.

Example:

```
;OtherPressVector routine that ignores releases (high bit of mouseData is set on releases)
;
MyOtherPress:                           ;control comes here from otherPressVector
        lda     mouseData               ;check state of the mouse button
        bmi     90$                     ;ignore it if it's a release
        jsr     PressDown               ;otherwise process the press
90$:
        rts                             ;return to MainLoop
```

For more information on **otherPressVector**, refer to "Other Mouse Presses" in this chapter.

Icon Precedence

GEOS draws icons sequentially. Therefore, if icons overlap, the ones which are drawn later will be drawn on top. When the user clicks somewhere on the screen, GEOS scans the icon table in this same order, looking for an icon whose rectangular boundaries enclose the coordinates of the mouse pointer. If more than one icon occupies the coordinate position, the icon that is defined first in the icon table (and therefore drawn on bottom) will be given the icon event. If an active menu and an icon overlap, the menu will *always* be given precedence.

Disabling Icons

An application can disable an icon in the current icon structure by clearing the **OFF_PIC_ICON** word of the icon (setting it to $0000). If an icon is disabled prior to a call to **DoIcons**, the icon will not be drawn. If an icon is disabled after the call to **DoIcons**, the icon will remain on the screen but will be ignored during the icon scan. The application can reenable the icon by restoring the **OFF_PIC_ICON** word to its original value. (Actually, any non-zero value will do because reenabling an icon does not redraw it, it only restores the coordinates to **MainLoop**'s active search list.)

GEOS 128 Icon Doubling

as with bitmaps, special flags in the icon data structure can be set to automatically double the x-position and/or icon width when GEOS 128 is running in 80-column mode. To have an an icon's x-position automatically doubled in 80-column mode, bitwise-or the **OFF_I_X** parameter with **DOUBLE_B**. To double an icon's width in 80-column mode, bitwise-or the **OFF_I_WIDTH** parameter with **DOUBLE_B**. These bits will be ignormed when GEOS 128 is running in 40-column mode. Do not, however, use these doubling bits when running under GEOS 64. GEOS 64 will try to treat the doubling bit as part of the coordinate or width value rather than a special-case flag. For more information, refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter @GR@.

Example:

```
;*******************************************************************************************
;SAMPLE GEOS 128 ICON TABLE THAT USES AUTOMATIC DOUBLING FEATURE
;*******************************************************************************************

.if !C128
        .echo  Error: cannot assemble GEOS 128 specific code without C128 flag set
.else

PaintIcon:




PAINTW = PicW
PAINTH = PicH
PAINTX = 16/8
PAINTY = 80

;The actual icon data structure to pass to DoIcons follows
IconTable:

I_header:
        .byte  NUMOFICONS                                   ;number of icon entries
        .word  ((PAINTX*8) + (PAINTW*8/2)) | DOUBLE_W)      ;position mouse over paint icon
        .byte  PAINTY + PAINTH/2                            ;

I_entries:
PaintIStruct:
        .word  PaintIcon                    ;pointer to bitmap
        .byte  (PAINTX | DOUBLE_B)          ;x card position (dbl in 80-column mode)
        .byte  PAINTY                       ;y-position
        .byte  (PAINTW | DOUBLE_B)          ;icon width (dbl in 80-column mode)
        .byte  PAINTH                       ;icon height
        .word  PaintEvent                   ;event handler

NUMOFICONS      = (*-I_entries)/IESIZE      ;number of icons in table

;Dummy icon event routines which do nothing but return
PaintEvent:
        rts

.endif
```

Apple GEOS Double-width and Aux-memory Icons

As with Apple GEOS bitmaps, special flags can be set in the icon data structure to double an icon's width and/or look for the icon image data in auxiliary memory. To double an icon's width, bitwise-or the **OFF_I_WIDTH** parameter with **DOUBLE_B**. To mark the **OFF_I_PIC** word as an address in auxiliary memory, bitwise-or the **OFF_I_X** parameter with **INAUX_B**. For more information, refer to "Apple Bitmap Doubling and Aux-memory Bitmaps" in Chapter @GR@.

Menus

Menus, one of the most common and powerful user-interface facilities provided by GEOS, allow the application to offer lists of items and options to the user. The familiar menus of the GEOS deskTop, for example, provide options for selecting desk accessories, manipulating files, copying disks, and opening applications. Virtually every GEOS-based program will take advantage of these capabilities, providing a consistent interface across applications.

GEOS menus come in two flavors: horizontal and vertical. The *main menu*, the menu which is always displayed, is usually of the horizontal type and is typically placed at the top of the screen. Each selection in the main menu usually has a corresponding vertical *sub-menu* that opens up when an item in the main menu is chosen. These sub-menus can contain items that trigger the application to perform some action. They can also lead to further levels of sub-menus. For example, a horizontal main menu item can open up to a vertical menu, which can have items which then open up other horizontal sub-menus, which can then lead to other vertical menus, and so on.

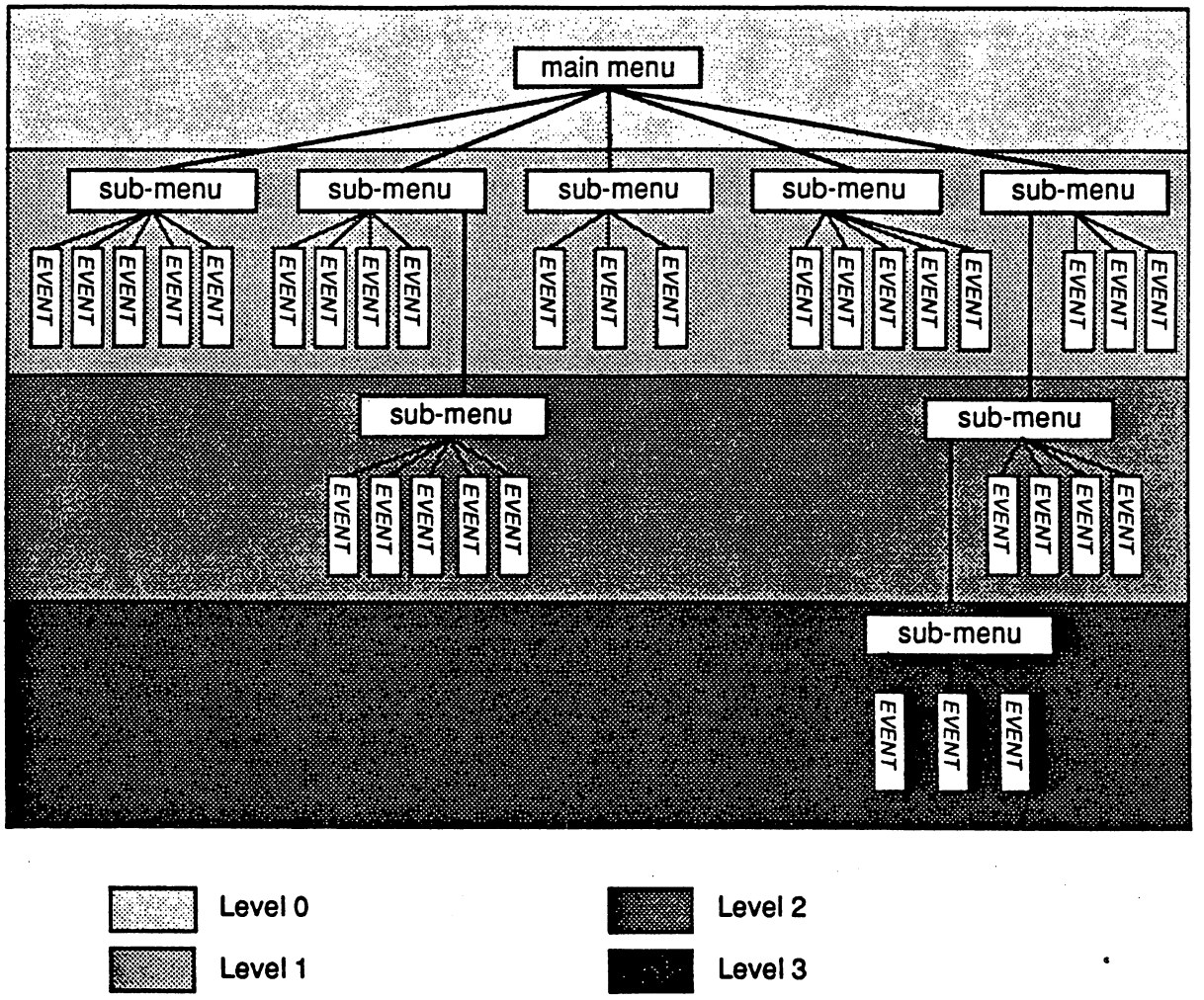
Division of Labor with Menus

GEOS divides the labor of handling menus between itself and the application. The GEOS Kernal handles all of the user's interaction with the menus. This includes drawing the menu items, opening up necessary sub-menus, and restoring the screen area from the background buffer when the menus are retracted. **MainLoop** manages the menus, keeping track of which items the user selects. If the user moves off of the menu area without making a selection, GEOS automatically retracts the menus without alerting the application.

If the user selects a menu item which generates a menu event, the application's *menu event handler* is called with the menus left open. Leaving the menus open allows the application to choose when and how to retract them: all the way back to the main menu, up one or more levels (for multiple sub-menus), or up no levels (keeping the current menu open). This lets the application choose the menu level which is given control upon return, thereby allowing multiple selections from a sub-menu without forcing the user to repeatedly traverse the full menu tree for each option.

Menu Data Structure

The main menu, all its sub-menus, their individual selectable items, and various attributes associated with each menu and each item are all stored in a hierarchical data structure called the *menu tree*. Conceptually, a menu tree with multiple sub-menus might have the following layout:

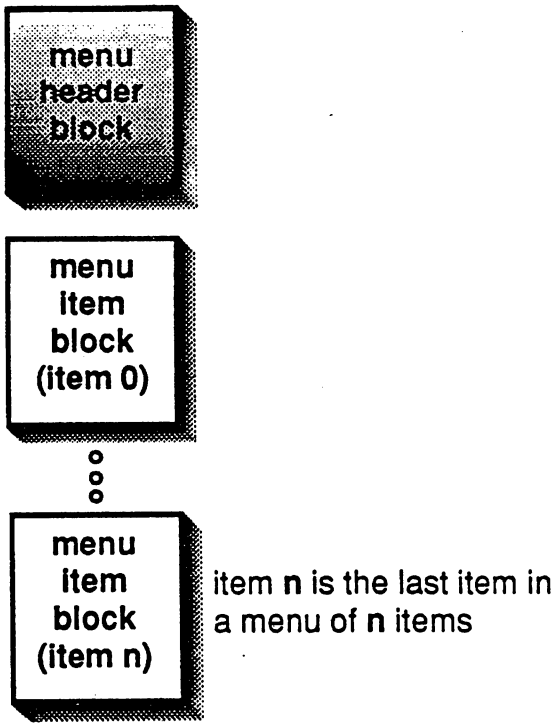


Sample Menu Tree

The main menu (or level 0) is the first element in the tree; it is the menu that is always displayed while menus are enabled. Each item in a main menu will usually point to a secondary menu or submenu. Items in these submenus can point to events (alerts to the application that an item was selected) or they can point to additional submenus. Menus are linked together by address pointers.

Sub-menus are sometimes referred to as *child* menus, and the menu which spawned the sub-menu as its *parent*. Sub-menus can be nested to a depth determined by the GEOS constant MAX_M_NESTING, which reflects the internal variable space allocated to menus. The depth or level of the current menu can be determined by the GEOS variable **menuNumber**, which can range from 0 to (MAX_M_NESTING-1).

In memory, all menus, whether the main menu or its children, are stored in the same basic *menu structure* format. Each menu is comprised of a single *menu header block* followed by a number of *menu item blocks* (one for each selectable item in the menu):



Menu/Sub-menu structure

Menu/Sub-menu Header

The menu header is a seven-byte structure that specifies the size and location of the menu (How big is the rectangle that surrounds the menu and where should the menu be drawn?), any attributes that affect the entire menu (Is it a vertical or horizontal menu?), and the number of selectable items in the menu. The header is in the following format:

Menu/Sub-menu Table Header:

Index	Constant	Size	Description
+0	OFF_M_Y_TOP	byte	Top edge of menu rectangle (y1 pixel position).
+1	OFF_M_Y_BOT	byte	Bottom edge of menu rectangle (y2 pixel position).
+2	OFF_M_X_LEFT	word	Left edge of menu rectangle (x1 pixel position).
+4	OFF_M_X_RIGHT	word	Right edge of menu rectangle (x2 pixel position).
+6	OFF_NUM_M_ITEMS	byte	Menu type bitwise-or'ed with number of items in this menu/sub-menu.

The first six bytes specify the screen location and size of the menu with the positions of the bounding rectangle in pixel positions. The x-positions are word (two-byte) values and the y-positions are byte values These values are absolute screen pixel positions. The size of the bounding rectangle depends on the number of menu items and the size of text strings within the menu. The height of the rectangle can be calculated with the constant **M_HEIGHT**: a horizontal menu is always a height of **M_HEIGHT**, and a vertical menu is a height of the number of menu items multiplied by **M_HEIGHT**. For example, the height of a vertical menu with seven items would be **7*M_HEIGHT**. The width of a menu is more difficult to calculate because it depends on the

length of the individual text strings. It is best to use a large number for this dimension and adjust it to a smaller size if necessary.

Important:	GEOS 64 and GEOS 128 before version 2.0 do not correctly handle menus that extend beyond an x-position of 255. This problem does not exist in Apple GEOS.

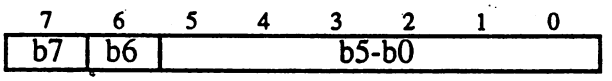
All menus and sub-menus are positioned independently. This means that the main menu need not be at the top of the screen (it can be inside a window, for example), and sub-menus need not be adjacent to their parent menus (although that is where you will usually want them). You can experiment with the flexibility of menu positioning to customize your applications.

The seventh byte is the attribute byte. It is the number of selectable items in the menu bitwise-or'ed with any menu type flags. A menu can have as many as MAX_M_ITEMS selectable menu items.

Menu/Sub-menu Types (use in attribute byte):

Constant	Description
HORIZONTAL	Arrange menu items in this menu/sub-menu horizontally.
VERTICAL	Arrange menu items in this menu/sub-menu vertically.
CONSTRAINED	Constrain the mouse to the menu/sub-menu. If the menu is a sub-menu, the mouse can still be moved off to the parent menu (off the top of a vertical sub-menu or off the left of a horizontal menu).
UN_CONSTRAINED	Do not constrain the mouse to the menu/sub-menu. If the user moves off of the menu, GEOS will retract it.

Bitwise Breakdown of the Attribute Byte:



b7 orientation: 1= vertical; 0 = horizontal.
b6 constrained: 1 = yes; 0 = no.
b5–b0 number of items in menu/sub-menu (up to MAX_M_ITEMS).

Some of the menu types are obviously mutually exclusive: you can't, for example, make a menu both vertical and horizontal, nor simultaneously constrained and unconstrained.

A vertical, unconstrained menu with seven selectable items would have an attribute byte of:

```
.byte   (7 | VERTICAL | UN_CONSTRAINED)
```

A horizontal, constrained menu with 11 selectable items would have an attribute byte of:

```
.byte   (11 | HORIZONTAL | CONSTRAINED)
```

Most sub-menus are unconstrained: if the user moves the pointer off the sub-menu, all opened menus are retracted as if **GotoFirstMenu** had been called. A constrained menu, on the other hand, restricts the pointer from moving off the menu area from all but one side. A constrained menu will only allow the pointer to move off the side leading back to where it expects the parent menu to be: off the top for a vertical sub-menu and off the left for a horizontal sub-menu. If the

user moves off of a constrained menu (in the only available direction), the current sub-menu is retracted and the parent menu becomes active as if **DoPreviousMenu** had been called.

NOTE:	The constrain option is only applicable to sub-menus — if the **CONSTRAINED** flag is set in the main menu (level 0), the option will have no effect.

Menu Item Structure

For each selectable item in a menu (the number items is specified in the header) there is a five-byte item structure. These item structures follow the menu header in memory. The first item represents the first menu selection (top- or leftmost), the second, the second, and so on. Each item structure specifies the text that will appear in the menu, what happens when the item is selected (Will it generate an event or a sub-menu?), and the appropriate event routine or sub-menu. Each menu item is in the following format:

Menu Item:

Index	Constant	Size	Description
+0	OFF__TEXT_ITEM	word	Pointer to null-terminated text string for this menu item.
+2	OFF_TYPE_ITEM	byte	Selection type (sub-menu, event, dynamic sub-menu).
+3	OFF_POINTER_ITEM	word	Pointer to sub-menu data structure, event routine, or dynamic sub-menu routine, depending on selection type.

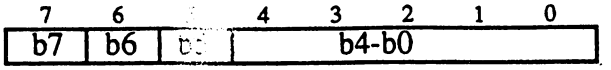
The first word of the item is a pointer to the text that will be placed in the menu. The text is expected to be null-terminated (the last byte should be $00 or **NULL**). If the menu rectangle specified in the header is not wide enough to contain the entire text string, the text will be clipped at the right edge when the menu is drawn.

The byte following the text pointer (the third byte) is an item type indicator. Each selectable item can either be an an *action*, a *sub-menu*, or a *dynamic sub-menu* selection. An action-type item generates a menu event from **MainLoop**. A sub-menu type item automatically opens up a sub-menu structure. And a dynamic sub-menu type selection opens up a sub-menu, but before it does, it calls an application's routine. Dynamic sub-menus are useful for modifying a menu structure on the fly. For example, a point size sub-menu, such as those used in geoWrite, can be changed dynamically when a new font is selected. When the user chooses the font item, the dynamic sub-menu routine checks the list of available point sizes and builds out the point size sub-menu based on its findings. The following table summarizes the three menu item types:

Types of Menu Items (for use in item type byte):

Constant	Description
SUB_MENU	This menu item leads to a sub-menu. The **OFF_POINTER_ITEM** is a pointer to the sub-menu data structure (points the first byte of a menu/sub-menu header).
DYN_SUB_MENU	This menu item is a dynamic sub-menu. The **OFF_POINTER_ITEM** is a pointer to a dynamic sub-menu routine that is called *before* the menu is actually drawn. The dynamic sub-menu routine can do any necessary pre-processing and return with **r0** containing a pointer to a sub-menu data structure or $0000 to ignore the selection.
MENU_ACTION	This menu item generates an event. The **OFF_POINTER_ITEM** is a pointer to the event routine that will to call.

Bitwise Breakdown of the Item Type Byte:



b7	sub-menu flag.
b6	dynamic sub-menu flag.
b4–b0	*reserved for future use.*

Example Menu

The following code fragment defines an unconstrained horizontal menu of three items, suitable for use as the main menu. Each item in the menu points to a sub-menu that is not shown (**GeosMenu**, **FileMenu**, and **EditMenu**).

Example:

```
;*******************************************************************************************
;SAMPLE MENU TABLE
;*******************************************************************************************

;*** MENU CONSTANTS ***
;bounding rectangle
MAINX1          = 0                             ;left edge
MAINY1          = 0                             ;top edge
MAINX2          = 72                            ;right edge
MAINY2          = MAINY1 + M_HEIGHT             ;bottom edge

;************************
;*** MENU DEFINITION ***
;************************

;       HEADER
MainMenu:
        .byte   MAINY1                          ;top
        .byte   MAINY2                          ;bottom
        .word   MAINX1                          ;left
        .word   MAINX2                          ;right
        .byte   ( HORIZONTAL | UN_CONSTRAINED | M_ITEMS)

;*** ITEMS ***
MainItems:
;Geos
        .word   GeosText                        ;pointer to null-terminated text
        .byte   SUB_MENU                        ;generates sub-menu
        .word   GeosMenu                        ;pointer to sub-menu structure

;File
        .word   FileText                        ;pointer to null-terminated text
        .byte   SUB_MENU                        ;generates sub-menu
        .word   FileMenu                        ;pointer to sub-menu structure

;Edit
        .word   EditText                        ;pointer to null-terminated text
        .byte   SUB_MENU                        ;generates sub-menu
        .word   EditMenu                        ;pointer to sub-menu structure

M_ITEMS         = (* - MainItems)/MISIZE        ;number of items in the menu

;text string for Geos selection
```

```
GeosText:
        .byte   "Geos", NULL                ;null-terminated item string

;text string for File selection
FileText:
        .byte   "File", NULL                ;null-terminated item string

;text string for Edit selection
EditText:
        .byte   "Edit", NULL                ;null-terminated item string
```

Installing Menus

When an application is first loaded, GEOS will not have an active menu structure. GEOS must be given the address of the application's menu structure before **MainLoop** can display and track the user's interaction with it. GEOS provides one routine for installing menus:

• DoMenu	Display and activate a menu structure.

DoMenu draws the main menu on the foreground screen and instructs **MainLoop** to begin taking care of all menu processing. The menu stays activated and enabled until the **MENU_ON_BIT** or the **MOUSE_ON_BIT** of **mouseOn** is cleared or another menu is installed by calling **DoMenu** with the address of a different menu structure. In either case, the old menu is not erased from the foreground screen by GEOS. The application must recover the area from the background buffer itself.

MainLoop and Menu Events

When the user clicks the mouse button on a menu item, GEOS **MainLoop** will invert the selection and examine the item data block, processing the selection according to its type.

SUB_MENU

If the menu item is of the **SUB_MENU** type, then **menuNumber** is incremented, the appropriate sub-menu is drawn, and **MainLoop** begins tracking the user's interaction with the sub-menu, making it the current menu. If the user moves off of a sub-menu back onto its parent menu, **MainLoop** will retract the sub-menu, decrement **menuNumber**, and make the parent menu the current menu. If the user moves off of the menus entirely (assuming this is possible — the menu might be constrained), then **MainLoop** retracts all sub-menus back up to the the main menu and sets **menuNumber** to zero.

DYNAMIC_SUB_MENU

If the menu item is of the **DYNAMIC_SUB_MENU** type, **MainLoop** calls the routine whose address is in the item structure. This routine is called before the sub-menu is drawn and before **menuNumber** is incremented. The accumulator will contain the item number selected (item numbers start with zero). When the routine returns with the address of the appropriate sub-menu in **r0**, **MainLoop** continues processing as if it was handling a **SUB_MENU** type menu. If the dynamic sub-menu routine returns $0000 in **r0**, then the sub-menu is not opened and the current menu remains active.

MENU_ACTION

If the menu item is of the **MENU_ACTION** type, GEOS flashes the menu inverted for **selectionFlash** vblanks. **selectionFlash** is a GEOS variable which is initialized with the constant **SELECTION_DELAY**, but may be adjusted by the application. **MainLoop** will then call the menu event routine whose address is in the item structure, passing the number of the selected item in the accumulator (item numbers start with zero). One of the first things a menu event routine must do, among its own duties, is specify which menu level **MainLoop** should return to when it gets control. This is done by calling one of the GEOS routines designed for this purpose:

• **ReDoMenu**	Reactivate the menu at the current level.
• **DoPreviousMenu**	Retract the current sub-menu and reactivate the menu at the previous level.
• **GotoFirstMenu**	Retract all sub-menus and reactivate the menu at the main menu level.

These routines retract menus as necessary (recovering from the background buffer) and sets special flags which tell **MainLoop** what has happened; **MainLoop** is not given control at this time — that is the job of the menu event handler's **rts**. If an application's menu event handler does not call one of these routines before it returns to **MainLoop**, the menu will remain open but inactive.

NOTE: A menu remains on the foreground screen until **DoPreviousMenu** or **GotoFirstMenu** is called to retract it. If graphics need to be drawn in the area obscured by a menu, but menus cannot be retracted, then limit drawing to the background buffer by setting the proper bits in **dispBufferOn**.

Specialized Menu Recover Routines

GEOS provides two very low-level menu routines which recover areas obscured by menus from the background buffer. Usually these routines are only called internally by the higher-level menu routines such as **DoPreviousMenu**. They are of little use in most applications and are included in the jump table mainly for historical reasons. There are two routines:

• **RecoverMenu**	Recovers the current menu from the background buffer to the foreground screen.
• **RecoverAllMenus**	Recovers all extant menus and sub-menus from the background buffer to the foreground screen.

Advanced Menu Ideas

Menu routines can be as clever as desired. One common technique involves dynamically modifying the text strings associated with menu items. This can be used, for example, to add asterisks next to currently active options as they are selected.

Menus and Mouse-Fault Interaction

How GEOS uses Mouse Faults

In general, the following is true:

- When a menu is down, the system interrupt-level mouse-processing routine is checking for two types of mouse faults: the mouse moving outside of the rectangle defined by **mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight** and the mouse moving off of the menu. It sets bits in **mouseFault** accordingly.

- If the menu is unconstrained, **mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight** are set to full-screen dimensions, thereby ruling out this type of mouse fault.

- If the menu is constrained, **mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight** are set to the dimensions of the current menu's rectangle. This will keep the mouse from moving off of the menu area (and will also generate a mouse fault when an edge is encountered).

- The system mouse fault routine (called through **mouseFaultVec**) checks the **mouseFault** variable. If the mouse faulted by moving off of the menu (only possible if the menu is unconstrained), **DoPreviousMenu** is called. If the user moved off of the sub-menu without moving onto another menu, mouse menu faults will continue to retract menus until only the main menu is displayed. If the mouse faulted by attempting to move beyond the **mouseTop** on a vertical sub-menu or **mouseLeft** on a horizontal sub-menu (only possible on a constrained menu) then **DoPreviousMenu** is called.

Application's Use of Mouse Faults

When the user is interacting with menus, the system uses the mouse fault variables (**mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight**) and expects its own fault service routine to be called through **mouseFaultVec**. If an application needs use mouse faults for its own purposes, should first disable menus by clearing the **MENUON_BIT** of **mouseOn** Before reenabling menus, it should set the fault variables to the full screen dimensions and call **StartMouseMode** to restore the system's fault service routine:

```
;*******************************************************
;Routine to restore the mouse service routines to an
;operational state after an application's use of
;mouse faults through mouseFaultVec. Should be called
;before menus are reenabled.
;
;*******************************************************
ResetMouse:

;--- Following lines changed to save bytes
;       LoadW   mouseLeft, #0                   ;reset mouse left to left screen edge
;       LoadB   mouseTop, #0                    ;and mouse top to top screen edge
        lda     #0                              ;
        sta     mouseLeft                       ;
        sta     mouseLeft+1                     ;
        sta     mouseTop                        ;
.if     (C128)
        LoadW   r0,#(SC_40_WIDTH-1 | DOUBLE_W | ADD1_W)     ;put in zp reg to normalize
        ldx     #r0                             ;point to register
        jsr     NormalizeX                      ;double if in 80-column
        MoveW   r0,mouseRight                   ;mouse right to right screen edge
.else   ;(APPLE || C64)
```

```
        LoadW   mouseRight,#SC_PIX_WIDTH-1          ;mouse right to right screen edge
.endif
        LoadB   mouseBottom,#SC_PIX_HEIGHT-1        ;mouse bottom to bottom screen edge
        clc                                         ;don't reposition mouse...
        jsr     StartMouseMode                      ;
        rts                                         ;exit
```

Sample Menu

Other Mouse Presses

When the user clicks the mouse somewhere on the screen where there is no active menu or icon, GEOS consideres this an "other" press and checks **otherPressVector** for an application-provided subroutine. If **otherPressVector** is $0000, then the press is ignored. if **otherPressVector** contains anything but $0000, GEOS treats the value as an absolute address and simulates an indirect **jsr** to that address. **otherPressVector** defaults to $0000 at application startup.

otherPressVector gets called on all *presses* that are not on an active icon or menu and on all *releases*, whether on a menu, icon, or anywhere else. In most cases, the application will want to ignore the releases. This is done simply by checking **mouseData** for the current state of the mouse button, as in:

```
        lda     mouseData           ;check state of the mouse button
        bpl     10$                 ;branch to handle presses
        rts                         ;but return immediately to ignore releases
10$:
```

Because **otherPressVector** gets called on each press (and release), any double-click detection must be performed manually by the other-press routine. Handling double-clicks through

otherPressVector is similar to the polled mouse method used with icons, the major difference being a check for releases on entry.

Example:

```
;Ignore releases on entry
       lda     mouseData              ;check state of the mouse button
       bpl     5$                     ;branch to handle presses
       rts                            ;but return immediately to ignore releases
5$:
;User pressed mouse once, start double-click counter going
       LoadB   dblClickCount,#CLICK_COUNT   ;start delay

;Loop until double-click counter times-out or button is released
10$:
       lda     dblClickCount          ;check double-click timer
       beq     30$                    ;If timed-out, no double-click
       lda     mouseData              ;Else, check for release
       bpl     10$                    ;loop until released

;mouse was released,loop until double-click counter times-out or
;button is pressed a second time.
20$:
       lda     dblClickCount          ;check double-click timer
       beq     30$                    ;If timed-out, no double-click
       lda     mouseData              ;Else, check for second press
       bmi     20$                    ;loop until pressed

;Double-click detected (no single-click)
30$:   jsr     DoDoubleClick          ;do double-click stuff
       bra     90$                    ;exit

;Single-click detected (no double-click)
       jsr     DoSingleClick          ;do single-click stuff
                                      ;and fall through to exit

;Exit
90$:   rts                            ;return to MainLoop
```

Process Library

A *process* is a is an event that is triggered on a regular basis by a timer. This allows GEOS to generate an event at specific time intervals, such as 20 times per second, once every minute, or five times each hour. Processes allow a limited form of multitasking, where many short routines can appear to run concurrently with **MainLoop**. Thus an application could update an alarm clock and scroll the work area while calculating a cell in a spreadsheet. Applications can also use processes to monitor the mouse. geoPaint, for example, uses a process to monitor the mouse's position when using the line tool; when the mouse moves, the process prints the new line length in the status window. geoPublish operates in a similar manner, using a process to update the values in the coordinate boxes as the user moves the across the preview page.

Note:	Processes do not provide true multitasking. There is no interrupt-driven context switching, nor any concurrence (where two routines run simultaneously). Processes are best thought of as events triggered off of **MainLoop** just like any other event. When one process is running, the next process in line won't get executed until the first finishes and returns to **MainLoop**.

Process Nomenclature

There are a number of terms associated with processes. Each process has a *countdown timer*. When the countdown timer reaches zero or *times-out*, the process becomes *runable*. If a process is *frozen*, its timer is not being decremented. The timer will continue when the process is *unfrozen*. If a process is *blocked*, a *process event* will not be generated until the process becomes *unblocked*.

Process Data Structure

The application must initialize the GEOS process handler with a process data structure. The process data structure contains the necessary information for all the desired processes. The table can specify up to **MAX_PROCESSES** (formerly **MAXIMUM_PROCESSES**) processes. Each process in the table is in the following format:

Index	Constant	Size	Description
+0	OFF_P_EVENT	word	Pointer to event routine that is called when this process times-out.
+2	OFF_P_TIMER	word	Timer initialization value: number of vblanks to wait between one event trigger and the next.

The first word is the address of the process event handler. The process event handler is much like any other event handler: it is called by **MainLoop** when process becomes runable (as opposed to, say, when the user clicks on an icon or selects a menu item) and is expected to return with an **rts**.

The second word is the number of vblanks to wait between one event trigger and the next. If the the **OFF_P_TIMER** word of a process is set to 20, for example, then the process event handler will be called every 20 vblanks (about 3 times per second on NTSC machines and 2.5 times per second on PAL machines).

Sample Process Table

The following data block defines three processes, each with a different process event handler. The first process will execute once every 10 vblanks, the second will execute once every second, and the third will execute once every five minutes. Notice the use of the **FRAME_RATE** constant to calculate the correct vblank delay for PAL and NTSC machines and the automatic assignment of process constants with **(* – PrTable)/PSIZE.**

```
;*******************************************************
;Sample process data structure
;*******************************************************

PrTable:
;*** MOUSE CHECK PROCESS***
;       Check mouse position and change pointer form as
;       necessary.
MOUSECHECK      = (*-PrTable)/PSIZE             ;process number
        .word   CheckMouse                      ;process event routine
        .word   10                              ;check every 10 vblanks

;*** REAL-TIME CLOCK PROCESS ***
;       Increment a real-time clock counter every second
RTCLOCK         = (*-PrTable)/PSIZE             ;process number
        .word   Tick                            ;process event routine
        .word   FRAME_RATE                      ;one second worth of vblanks

;*** SCREEN-SAVER PROCESS ***
;       Save the screen by turning off colors after five
;       minutes.
SCRNSAVER       = (*-PrTable)/PSIZE             ;process number
        .word   ScreenSave                      ;process event routine
        .word   5*60*FRAME_RATE                 ;frames in 5 minutes
                                                ;delay = 5 min* 60 sec/min * frames/sec)

NUM_PROC        = (*-ProcTable)/PSIZE           ;number of processes in this table
                                                ;for passing to InitProcesses

.if     (NUM_PROC > MAX_PROCESSES)              ;check for too many processes
        echo    Warning: too many processes
.endif
```

Process Management

Installing Processes

The application must install its processes by telling GEOS the location of the process data structure and the number of processes in the structure. GEOS provides one routine for installing processes:

• **InitProcesses**	Initialize and install processes.

InitProcesses copies the process data structure into an internal area of memory, hidden from the application. GEOS maintains the processes within this internal area, keeping track of the event routine addresses, the timer initialization values (used to reload the timers after they time-out), the current value of the timer, and the state of each process (i.e., frozen, blocked, runable). The application's copy of the process data structure is no longer needed because GEOS remembers this information until a subsequent call to **InitProcesses.**

Example:

```
;*** Initialize process table ***
       LoadW   r0,#ProcTable        ;point at process data structure
       lda     #NUM_PROC            ;pass actual number of processes
       jsr     InitProcesses        ;call GEOS to install processes
                                    ;processes in table are now blocked and frozen
```

Starting and Restarting Processes

When a process table is installed, the processes do not begin executing immediately because all processes are initialized as frozen. GEOS provides a routine to simultaneously unblock and unfreeze a single process while reinitializing its countdown timer:

• **RestartProcess**	Initialize a process's timer value then unblock and unfreeze it.

RestartProcess should *always* be used to start a process for the first time, otherwise the timer will begin in an unknown state.

Example:

```
;*** Start all processes ***
       ldx     #NUM_PROC-1          ;process numbers range from 0 to NUM_PROC-1
10$:
       jsr     RestartProcess       ;reset timer, unblock, and unfreeze process
       dex                          ;next process
       bpl     10$                  ;loop until done
```

RestartProcess can also be used rewind a process to the beginning of its cycle. One application for this is a screen-saver utility which blanks the screen after, say, five minutes of inactivity to prevent phosphor burn-in. A five-minute process is established which, when it triggers an event, blanks the screen. Any routine which detects activity from the user (a mouse movement, button press, keypress, etc.) before the screen is blanked can call **RestartProcess** to reset the screen-saver countdown timer to its initial five minute value.

Freezing and Blocking Processes

When a process is frozen, its timer is no longer decremented every vblank. It will therefore never time-out and generate a process event. When a process is unfrozen, its timer again begins counting from the point where it was frozen. GEOS provides the following routines for freezing and unfreezing a process's timer:

• **FreezeProcess**	Freeze a process's countdown timer at its current value.
• **UnfreezeProcess**	Resume (unfreeze) a process's countdown timer.

Example:

```
;*** Freeze all processes ***
       php                          ;disable interrupts to synchronize freezing
       sei                          ;
       ldx     #NUM_PROC-1          ;process numbers range from 0 to NUM_PROC-1
10$:
       jsr     FreezeProcess        ;freeze process
       dex                          ;next process
       bpl     10$                  ;loop until done
       plp                          ;restore interrupt status
```

A process may also be blocked. Blocking a process temporarily prevents the event service routine from being executed. It does not stop the timer from decrementing, but when the timer reaches zero and the process bcomes runable, the event is not generated. When a process is subsequently unblocked, its events will again be generated. GEOS provides the following routines for blocking and unblocking processes:

• **BlockProcess**	Block a process's events.
• **UnblockProcess**	Allow a process's events to go through.

Example:

```
;*** Block mouse-checking process ***
       ldx     #MOUSECHECK           ;process number of mouse check
       jsr     BlockProcess          ;block it

;*** Unblock Real-time clock process ***
       ldx     #RTCLOCK              ;process number of real-time clock
       jsr     UnblockProcess        ;unblock it
```

When a timer reaches zero (times-out), its process becomes runable. An internal GEOS flag (called the *runable flag*) is set, indicating to **MainLoop** that an event is pending. The timer is then restarted with its initialization value. **MainLoop** will ignore the runable flag as long as the process is blocked. When the process is later unblocked, **MainLoop** will see the runable flag, recognize it as a pending event, and call the appropriate service routine. However, multiple pending events are ignored: if a blocked process's timer reaches zero more than once, only one event will be generated when it is unblocked.

Freezing vs. Blocking

The differences between freezing and blocking are in many cases unimportant to the application. However, a good understanding of their subtleties will prevent problems that may arise if the wrong method is used.

Normally, a process's timer is decremented every vblank. If a process is frozen, however, the GEOS vblank interrupt routine will ignore the associated timer. The timer value will not change and, hence, will never reach zero. The process will never become runable. If you think of a process as a wind-up alarm clock, freezing is equivalent to disconnecting the drive spring — even the second hand stops moving.

Freezing a process only guarantees that the process will not *subsequently become runable*. The process may in fact already be marked as runable and GEOS is only awaiting the next pass through **MainLoop** to generate an event. (A process that is marked as runable but not yet run is said to be a *pending event.*)

If a process is blocked (but not also frozen), GEOS Interrupt Level will continue to decrement the associated timer. If the timer reaches zero, GEOS will reset the timer and make the process runable. But **MainLoop** will ignore the process and not generate an event because the process is blocked. If the process is later unblocked, the event will be generated during the next pass through **MainLoop**. Using the alarm clock analogy, freezing is equivalent to disconnecting the alarm bell — the clock continues to run but the alarm does not sound unless the bell is reconnected.

The only way to absolutely disable a process — both stopping its clock and preventing any pending events to get through — is to freeze and block it.

Example:

```
;*******************************************************************************
;StopProcess            --      freeze a process timer and block any pending events
;UnstopProcess          --      unfreeze and unblock the process
;
;Pass:          x       = process number
;
;Returns:       x       unchanged
;
;Destroys:      a
;
;*******************************************************************************

StopProcess:
        jsr     FreezeProcess           ;not that it really matters, but we'll freeze first
        jmp     BlockProcess            ;then block (let BlockProcess rts)

UnstopProcess:
        jsr     UnblockProcess          ;unblock first
        jmp     UnfreezeProcess         ;then unfreeze (let UnfreezeProcess rts)
```

Forcing a Process Event

Sometimes it is desirable to force a process to run on the next pass through **MainLoop**, independent of its timer value. GEOS provides one routine for this:

• **EnableProcess**	Makes a process runable immediately.

EnableProcess merely sets the runable flag in the hidden process table. When **MainLoop** encounters a process with this flag set, it will attempt to generate an event, just as if the timer had decremented to zero. This means that **EnableProcess** has no priveleged status and cannot override a blocked state. However, because it doesn't depend on (or affect) the current timer value, the process can become runable even with a frozen timer.

The Nitty-gritty of Processes

Processes involve a complex (but hopefully transparent to the application) interaction between multiple levels of GEOS. In advanced uses, it may be necessary to understand this interaction. The following discussing clarifies some of the fine points of processes.

Interrupt Level and MainLoop Level

Processes involve two distinct levels of GEOS: interrupt level and **MainLoop** level. Every vblank an IRQ (Interrupt ReQuest) signal is generated by the computer hardware. Part of the GEOS interrupt service routine manages process timers: if a process exists and it is not frozen, its timer is decremented. When the timer reaches zero, the interrupt level routine sets the associated runable flag and restarts the timer with its initialization value. *The process event routine is not called at this time.*

If for some reason interrupts are disabled (usually by setting the interrupt disable flag with an sei instruction) and a vblank occurs, the interrupt will be ignored and the process timers, therefore, will not be decremented during that vblank. This is usually not a problem because interrupts are normally enabled. However, be aware that some operating system functions (such as disk I/O) disable interrupts.

During a normal pass through **MainLoop**, GEOS will examine the active processes. If a process's runable flag is set and it is not blocked, **MainLoop** clears the runable flag and and calls the process. If a process is blocked, **MainLoop** ignores it.

Because of the way **MainLoop** and the interrupt level interact, there is a certain level of imprecision with processes:

1: If a process has a very low timer initialization value (e.g., less than five) such that it is possible it will time-out more than once during the time it takes for a single pass through **MainLoop**, **MainLoop** may miss some of these time-outs.Each time the timer reaches zero it sets the runable flag, but since there is only one runable flag per process, **MainLoop** has no way of knowing if it should generate more than one event.

2: It is impossible to guarantee any precise relationship (e.g., a timer difference less than five) between two or more timers. Although all processes that time-out during the same interrupt will become runable at that time, the interrupt may occur while **MainLoop** is the midst of handling processes: processes that have already been passed-by may bec runable but not get executed until the next time through **MainLoop**, which could fraction of a second later.

For more Information refer to Chapter @ML&INTS@.

Process Synchronization

It is sometimes desirable to maintain a synchronized relationship between the timer values of two or more processes. This is nontrivial because even if the calls to restart, freeze, or unfreeze these timers are done immediately after each other, there is always a slight chance that the vblank interrupt will occur after the status of some of the timers has changed but before all have been changed. For example: if an application is trying to freeze three timers simultaneously and the interrupt happens after the first timer has been frozen but before the other two, the remaining two timers will still be decremented. To circumvent this problem, bracket the calls by disabling interrupts before freezing, blocking, or restarting, and reenabling afterward. This is best done as in the following example:

```
;*** RESTART CLOCK PROCESSES AT THE SAME TIME ***
;
RstartP:
        php                             ;save interrupt disable flag
        sei                             ;disable interrupts (stopping timers)
        ldx     #RTCLOCK                ;restart clock
        jsr     RestartProcess
        ldx     #SCRNSAVER      ;restart screen-saver
        jsr     RestartProcess
        plp                             ;restore interrupt disable status
```

Disabling Processes While Menus Are Down

Because **MainLoop** is still running when menus are down, process events continue to occur. It is often desirable to disable a process while the user has a sub-menu opened. The easiest way to handle this situation is to check **menuNumber** at the beginning of the process event routine. If **menuNumber** is non-zero, then a menu is down and the event routine can exit early:

```
PrEventRoutine:
        lda     menuNumber      ;check menu level
        bne     90$                     ;and exit immediately if a menu is down
        jsr     DoPrEvent               ;else, process the event normally
90$:
        rts                             ;return to MainLoop
```

Sleeping

Sleeping is a method of stopping execution of a routine for a specified amount of time. That is: a routine can stop itself and "go to sleep," requesting **MainLoop** to wake it up at a later time. GEOS provides one routine for sleeping:

• Sleep	Pause execution for a given time interval.

Sleep does not actually suspend execution of the processor. When the application does a **jsr Sleep**, GEOS sets up a hidden timer, much like a process timer, that is decremented during the vblank interrupt. It removes the return address from the stack (which corresponds to the **jsr Sleep**) and saves it for later use, then perfoms an **rts**. Since the return address on the stack no longer corresponds to the **jsr Sleep**, control is returned to a **jsr** one level lower. In many cases, this will return control directly to **MainLoop**.

When the timer decrements to zero, a wake-up flag is set, and, on the next pass through **MainLoop**, the sleeping routine will be called with a **jsr** to the instruction that immediately follows the **jsr sleep**. When the routine finishes with an **rts** (or another **jsr Sleep**), **MainLoop** will resume processing.

Important:	Any temporary values pushed onto the stack must be pulled off prior to calling **Sleep**. Also, when a routine is awoken, the values in the processor registers and the GEOS pseudoregisters will most certainly contain different values from when it went to sleep. This is because **MainLoop** has been running full-speed, calling events and doing its own internal processing, thereby changing these values. If a routine needs to pass data from before it sleeps to after it awakes, it must do so in its own variable space.

Sleep can be used to set up temporary, run-once processes by placing calls to **Sleep** inside subroutines. For example, an educational program may want to flash items on the screen and make a noise when the student selects a correct answer. The routines that handle these "bells and whistles" can be established using **Sleep** without needlessly complicating the function that deals with correct answers. The following code fragment illustrates this idea:

```
;*****************************************************************
;
;Routine to handle a correct answer. Does some graphics, makes
;some noise, and adjusts the student's score.
;
;*****************************************************************

BELL_DELAY      =       60              ;length of bell
FLASH_DELAY     =       23              ;delay between flashes

Correct:
        IncW    score                   ;score += 1
        jsr     Bell                    ;start the bell going
        jsr     Flash                   ;start the answer flashing
        rts
```

```
;*****************************************************************
;Subroutine:  If sound is enabled (user-determined), start the
;                    bell sound and then go to sleep; Sleep
;                    returns control to the routine that called us
;
;                    When we wake up, we stop the bell sound and return
;                    to MainLoop
;
;                    If sound is disabled, then the rts returns
;                    directly to the routine that called us.
;*****************************************************************
Bell:
       lda     soundFlag              ;check sound flag
       beq     90$                    ;exit if user turned sound off
       jsr     BellOn                 ;else, turn the bell on
       LoadW   r0,BELL_DELAY          ;and delay before turning off
       jsr     Sleep                  ;by going to sleep (think rts)
       jsr     BellOff                ;turn bell off when we awake
90$:
       rts                            ;exit

;*****************************************************************
;Subroutine:  Invert the answer. Go to sleep. Re-invert the
;                    answer when we wake up.
;*****************************************************************
Flash:
       jsr     InvAnswer              ;Graphically invert the answer
       LoadW   r0,FLASH_DELAY         ;and delay before reverting
       jsr     Sleep                  ;by going to sleep (think rts)
       jsr     InvAnswer              ;when we awake, revert the image
       rts                            ;exit
```

Math Routines

One of the major limitations of eight-bit microprocessors such as the 6502 is their math capabilites: they can only operate directly on eight-bit quantites (0-255), and multiplication and division require extensive computational energy. For the sake of the application programmer, GEOS has some of the more popular arithmetic routines built into the Kernal. These include double-precison (two-byte) shifting, as well as multiplication and divison.

Parameter Passing to Math Routines

The math routines use a flexible parameter passing convention: rather than putting values into specific GEOS pseudoregisters, the application can place the values in any zero-page location (almost) and then tell GEOS where to find the values by passing the *address* of the of the parameter. Because the parameters are located on zero-page, their addresses are one-byte quantites that can be passed in the **x** and **y** index registers. For example, a GEOS math routine might require two word values. The application could place these values in pseudoregisters **a0** and **a1**, then call a GEOS math routine, like **Ddiv** (double-precision divide) with the address of **a0** and **a1** in the x and y registers.

Example:

```
ldx     #a0             ;load up address of first parameter
ldy     #a1             ;and address of other parameter
jsr     Ddiv            ;divide the word in a0 by the word in a1
```

Important:	It is easy to get confused and leave off the immediate-mode sign (#) when trying to load the *address* of a zero-page variable, thereby loading the value *contained in* the variable instead.

Double-precision Shifting

The 6502 provides instructions for shifting eight-bit quantites left and right but no instructions for directing these operations on 16-bit (double-precision) numbers. GEOS provides two routines for double-precision shifting:

• **DShiftLeft**	Arithmetically left-shifts a 16-bit word value.
• **DShiftRight**	Arithmetically right-shifts a 16-bit word value.

Double-Precision Arithmetic

Many of the possible double-precision arithmetic operations (such as word+word addition) are provided with GEOS macros. The standard set of GEOS macros, which include the likes of **AddW** and **SubW**, are listed in Appendix XX. Many double-precision operations, however, such as multiplication and division, are complicated enough to warrant an actual subroutine. GEOS provides many of these routines, some of which have signed and unsigned incarnations.

Signed vs. Unsigned Arithmetic

6502 arithmetic operations rely on the two's complement numbering system — an artifact of binary math — to provide both signed and unsigned operations with the same instructions (**adc** and **sbc**). For example, an **adc #$6c** can be seen as either adding 188 to the accumulator (unsigned math: all eight bits represent the positive number; any carry out of bit 7 indicates an overflow) or as adding a -68 to the accumulator (signed math: the high-bit, bit 7, holds the sign and any carry out of bit 6 indicates an overflow). The 6502 has little trouble adding and subtracting these two's-complement signed numbers. Operations such as multiplication and division, however, need to special-case the sign of the numbers.

Incrementing and Decrementing

GEOS has only one routine in the category of incrementing and decrementing:

• **Ddec**	Decrements a word, setting a flag if the value reaches zero.

However, because incrementing and decrementing words are such common operations, Berkeley Softworks has created a set of macros specifically designed for incrementing and decrementing word values:

```
;*****************************************************************
;IncW
;       Increment Word          --      IncW    addr
;
;       Args:   addr            --      address of word to increment
;
;       Action:         IncW increments a word. If the result is zero,
;                       then the zero flag in the status register is set.
;
;*****************************************************************
.macro  IncW    addr
        inc     addr            ;increment low-byte
        bne     done            ;branch if no carry into hi-byte
        inc     addr+1          ;propagate carry into hi-byte
done:                           ;z-flag is set if both hi & low were $00
.endm

;*****************************************************************
;DecW
;       Decrement Word          --      DecW    zpaddr
;
;       Args:   zpaddr          --      zero-page address of word to decrement
;
;       Action:         DecW Decrements a word. If the result is zero,
;                       then the zero flag in the status register is set.
;                       a and x registers are destroyed.
;
;*****************************************************************
.macro  DecW    zpaddr
        ldx     #zpaddr         ;load x with address of word for call
        jsr     Ddec            ;call GEOS routine
                                ;z-flag is set if both hi & low were $00
.endm

;*****************************************************************
;
;       Decrement Word #2       --      DecW2   addr
;       (fast version)
;
;       Args:   addr --         address of word to decrement
```

```
;
;       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

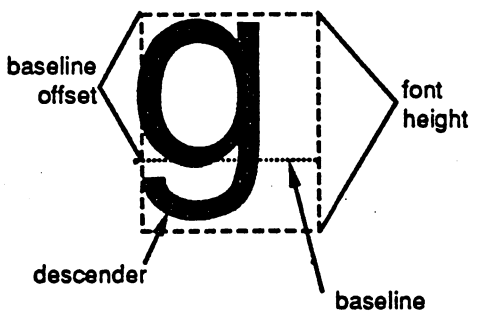
Although some characters are taller than others, all characters in a given font are treated as if they are the same height. This height is the font's point size. A 10 point font has a height of ten pixels. If a character's image is smaller than 10 pixels, it is because its definition includes white pixels 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:

```
;
;       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

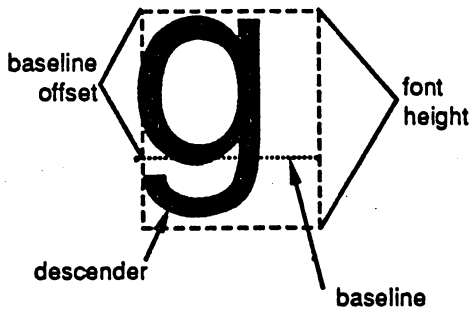
Although some characters are taller than others, all characters in a given font are treated as if they are the same height. This height is the font's point size. A 10 point font has a height of ten pixels. If a character's image is smaller than 10 pixels, it is because its definition includes white pixels 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:

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.*

†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.

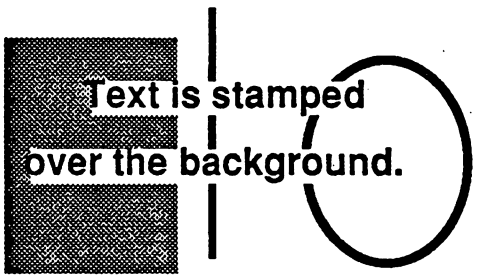
Style attributes temporarily modify the plaintext definition of the character and, in some cases, change the size and ultimate shape of the character:

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.
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.

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 underlined.

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

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

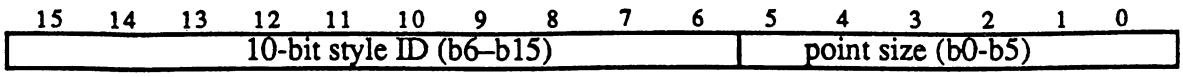
Fonts are stored in VLIR files of GEOS type **FONT.** A single font file contains all the available point sizes for a particular style (up to a maximum of 16). Each point size occupies one complete VLIR record. The record number corresponds to the point size (i.e., record 9 would contain the data for 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** 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:

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.

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:



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
+0	byte	Baseline offset (pixels from top of character).
+1	word	Bytes in one bitstream row (set width).
+3	byte	Font height.
+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.
+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.

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                                    ;
```

MainLoop and Interrupt Level: a Technical Breakdown

The GEOS Kernal operates on two distinct levels: *MainLoop Level* and *Interrupt Level*. MainLoop Level is characterized by the GEOS **MainLoop** — a never-ending loop at the heart of GEOS that routes events to the application. Whenever the application does not have control, **MainLoop** usually does.

But there is also Interrupt Level. Periodically (usually every 1/60th of a second) the computer hardware temporarily interrupts the microprocessor. The processor may be in the middle of **MainLoop**, deep within a GEOS routine, or somewhere in the applcation. Either way, the 6502 immediately suspends whatever it is doing and passes control to the GEOS Interrupt Level. Interrupt Level scans the keyboard circuitry, moves the mouse pointer, flashes the text prompt, decrements timers, and performs other low-level tasks. Interrupt Level operates independently of **MainLoop** and ensures that certain things get done on a regular basis. When the Interrupt Level processing is complete, control returns to the point where the original interrupt occurred.

Whatever GEOS does at Interrupt Level is mostly transparent to the application. Only when an application strays from the beaten path will it need to worry about the specifics of Interrupt Level processing.

This is a technical discussion of **MainLoop** and **Interrupt Level**. For a more general discussion, refer to Chapter @GEOSAPPS@.

MainLoop Level

When GEOS starts an application, it first initializes the operating system and then **jsr**'s to the application's start address. The application is expected to perform its basic startup procedures, such as initializing its menus, icons, and processes, and the return immediately with an **rts**. This **rts** will place GEOS at the beginning of **MainLoop**. **MainLoop** is primarily a small, endless loop of function calls:

MainLoop Service Routines

MainLoop itself is rather short. The meat of its function is hidden in the various service routines that it calls. Because these service routines interact directly with the application, it is useful to understand the specific conditions that affect their operation. The pseudo-code diagrams at the end of this chapter illustrate the operation of the more important service routines.

Patching Into MainLoop

Although most applications can function entirely off of events, some may find the need to install their own service routine directly off of **MainLoop**. GEOS has a single vector for this purpose: **applicationMain**, which usually contains $0000 and is therefore unused. By placing a routine address into this vector, GEOS will call through this vector every pass through **MainLoop**. To remove this call, the application can again store $0000 into the vector.

The Basics of Interrupt Level

Interrupt Level is primarily responsible for maintaining the interactive and time-based apsects of GEOS. Interrupt Level updates the mouse state and the mouse cursor position, watches for double-clicks, decrements process and sleep timers, gets keyboard input, flashes the prompt, and generates a new random number every vblank, among other (more obscure) tasks.

The Vertical Blank Interrupt

The Interrupt Level interrupt is tied directly to the video circuitry. In order to keep the screen phosphors glowing, the image must be redrawn, or *refreshed*, many times per second. Each complete coverage of the picture tube is called a *frame*, and the rate at which frames are drawn is called the *frame rate* or *refresh rate.*

At the end of each frame, the electron beam is switched off and returned to the upper left corner of the picture tube to begin drawing again. This period when the beam is off is called the *vertical blank*, or *vblank*. Every vblank, the IRQ (Interrupt ReQuest) line on the 6502 is pulled low. If the interrupt disable bit in the status register is clear (as it usually should be), an interrupt is generated. This interrupt is often called the *vblank interrupt.* GEOS uses the vblank interrupt as the basis for its Interrupt Level processing.

The vblank interrupt, along with the scanning of the video frame, occurs in a precisely timed sequence: 60 times per second on NTSC monitors (the United States standard) and 50 times per second on PAL monitors (the European standard).The GEOS **FRAME_RATE** constant reflects the number of frames per second (either 50 or 60) depending on the state of the **PAL** and **NTSC** constants.

How to Disable Interrupts

Because the vblank interupt is an IRQ (Interrupt ReQuest), the 6502 has the option of ingoring the request. To disable IRQ interrupts, an application need only set the interrupt disable bit in the 6502's status register using the **sei** (SEt Interrupt disable bit) instruction. Because GEOS depends on Interrupt Level executing on a timely basis, an application should disable interrupts only when absolutely needs to and then only for short periods of time. If an interupt occurs while the interrupt-disable bit is set, the interrupt will not be serviced. If too many interrupts are missed, much of the real-time features of GEOS — the mouse pointer, processes, double click detection, etc. — will become sluggish.

In conventional 6502 programming, it is standard practice to surround blocks of interupt-sensitive code with an **sei-cli** sequence: an initial **sei** to disable interrupts and an ending **cli** to reenable interrupts. This, however, is not a totally safe practice because the **cli** *always* reenables interrupts regardless of their original state. If interrupts were originally disabled, the **cli** may inadvertently reenable them. As applications get large, it becomes easier to embed these interrupt disable/enable sequences deep within subroutines. If one subroutine disables interrupts then calls another subroutine that then performs a **cli** (returning with interrupts enabled when they shouldn't be), the results may be a disasterous bug.

It is good to practice a little defensive coding and get into the habit of saving the interupt status when disabling them around blocks of code. The following sequence works well:

```
php             ;save current interrupt disable status
sei             ;disable interupts
.
.               ;(interrupt-sensitive code goes here)
.
```

```
        plp             ;restore old interrupt status
```

This **php-sei-plp** method will save, set, and then restore the interrupt disable bit. This way interrupts won't be inadvertently reenabled when they're expected to be disabled.

Apple GEOS Interrupts

Unlike the Commodore computers, the Apple IIe does not generate its own vblank IRQ interrupts. This function is usually provided by external hardware plugged into slot 7: the Apple mouse card or the Berkeley Softworks IRQ Management Card. If neither of these devices is present and there is no other interrupt source, Apple GEOS will generate *software interrupts*.

Apple Software Interrupts

GEOS may occasionally be run on systems with no interrupt source. This is an unfortunate situation because GEOS depends heavily on interrupt processing. GEOS will recognize this configuration and generate software interrupts during **MainLoop** by calling **IrqMiddle**. With applications that don't have time-consuming event routines hanging off of **MainLoop**, Interrupt Level processing will occur often enough to make the system usable. If system degradation is too great, an application can simulate its own software interrupts as necessary. For more information, refer to **IrqMiddle** in the Routine Reference Section.

Example:

```
;*****************************************************************************************
;DoSoftInts
;
;Description:  Simulate vblank interrupts under Apple GEOS when no interrupt source is
;              present.
;
;Pass:         nothing
;
;Destroys:     a,x,y
;
;*****************************************************************************************

DoSoftInts:
        bit     intSource               ;check interrupt source
        bne     10$                     ;exit if hardware interrupt source
        php                             ;else, generate soft interrupt
        sei                             ;disable interrupts (just in case)
        jsr     IrqMiddle               ;software interrupt now!
        plp                             ;restore old interrupt status
10$:
```

The Apple GEOS Interrupt Management Card

The Berkeley Softworks ~~IRQ~~ Management card requires reenabling after every interrupt in order to generate next interrupt. Part of the Apple Interrupt Level processing reenables the ~~IRQ Generator~~ card to interrupt on the next vblank. Normally this will keep interrupts triggering on a regular basis.

However, if the ~~IRQ Management~~ card generates an interupt while the 6502 interrupt disable bit is set, the interrupt service routine will not run, and the IRQ Management card will never be reenabled for the next interrupt.

Apple GEOS attempts to keep interrupts running by reenabling the IRQ Management card during **MainLoop** and whenever a call through the jump table switches banks. Normally applications will do this often enough in their normal operation to reenable the IRQ Management card ron a regular basis. It is conceivable, however, that in some very odd cases neither of these circumstances will occur often enough (*very odd cases*, indeed— if an application is both disabling interrupts and not going back to **MainLoop**, there is probably something fundamentally wrong with the structure of the program). An application can reenable the IRQ Management cardwhen necessary with the following sequence:

```
        bit     intSource               ;check the interrupt source
        bvc     5$                      ;ignore if not BSW card IRQ Manager
        sta     IRQ_GEN                 ;otherwise, trigger IRQ Manager
5$:
```

WHEN POLLING THE MOUSE

Important Things to Know About Interrupt Level

The vblank interrupt service routine is one of the most complex aspects of GEOS. Fortunately, most applications will need to know little more about the Interupt Level process than its basic functionioning. However, there are some unavoidable conflicts between Interupt Level and normal, mainstream processing, and these are important to know.

Two-byte Variables

During non-interrupt level processing, it is important to disable interrupts before referencing a word value that might get changed at Interrupt Level or changing a word value that might get referenced at Interupt Level. A two-byte quantity requires two memory accesses, and there is a small chance that an interrupt may occur after the first byte has been accessed but before the second byte has been accessed. This can result in a situation where a word value has the high-byte of one number and the low-byte of another. Take for example the variable **mouseXPos**, which is modified at Interrupt Level. The seemingly innocent code fragment below illustrates the problem:

```
        MoveW   mouseXPos,oldX          ;update our old mouse x-position with current mouse x
```

Which exands to the following at assembly time:

```
        lda     mouseXPos               ;update our old mouse x-position with current mouse
        sta     oldX
        lda     mouseXPos+1
        sta     oldX+1
```

If an interrupt occurs between the **lda mouseXPos** and the subsequent **lda mouseXPos+1**, the result word result stored in **oldX** may be entirely wrong. The solution is to temporarily disable interrupts around the access:

```
        php                             ;disable interrupts around access
        sei                             ;
        MoveW   mouseXPos,oldX          ;update our old mouse x-position with current mouse x
        plp                             ;restore old interrupt status
```

Be aware, though, that the **php-sei-plp** sequence has its own set of ideosyncracies: the **plp** restores the entire status register, not just the interrupt disable bit, thereby overwriting any new condition codes. Therefore, disabling as in

```
        php                             ;disable interrupts around compare
        sei                             ;
```

```
        CmpW    mouseXPos,oldX          ;compare current X with Old X
        plp                             ;restore interrupts
```

would defeat the whole purpose of the **CmpW**. In such cases, the condition codes can, of course, be tested *before* the **plp**. A better solution, however, would disable interrupts, shadow the word value to a temporary variable, restore the interrupt disable status, then do all checking against this temporary value, which won't get changed by Interrupt Level.

Example:

```
;Check if mouse is within the left and right text margin
IsMseInMargins:
        php                             ;disable interrupts around mouseXPos access
        sei                             ;and copy current pos to a working location
        MoveW   mouseXPos,r0            ;
        plp                             ;restore interrupts

        CmpW    r0,leftMargin           ;check left margin
        bge     10$                     ;branch if inside left
        lda     #FALSE                  ;else, flag fault
        beq     99$                     ;branch always to exit
10$:
        CmpW    r0,rightMargin          ;check right margin
        ble     20$:                    ;branch if inside right
        lda     #FALSE                  ;else, flag fault
        beq     99$                     ;branch always to exit
20$:
        lda     #TRUE                   ;no fault (inside text margins)
99$:
        rts                             ;exit
```

AND WITHOUT DISABLING INT'S

Word variables to be careful with include **mouseXPos**, **mouseLeft**, **mouseRight**, **intTopVector**, and **intBotVector**, all of which are either read or written to by Interrupt level.

The Decimal Mode Flag

GEOS adopts the convention that the normal operating state of the computer has decimal mode disabled. Any routine that enables decimal mode must also disable it. Versions of GEOS 64 prior to v1.2 do not disable decimal mode during interrupt level processing. If operating under one of these versions, it is necessary to disable interrupts prior to using the decimal mode flag.

Patching Into Interrupt Level

Very few applications will need access to the system at Interrupt Level. Most tasks that would traditionally require the use of a time-based interrupt can be handled deftly enough with GEOS processes. If an application can drive itself entirely off of **MainLoop** events, it should. The world of Interrupt Level is a delicate one; it is very easy to disrupt the entire system by doing the wrong thing during Interrupt Level. With that said, though, GEOS provides two vectors that allow an application that knows what it's doing to tap directly into Interrupt Level: **intTopVector** and **intBotVector**.

As illustrated in the Interrupt Level pseudo-code at the end of this chapter, control passes through these two vectors at different points in the interrrupt process. **intTopVector** allows the application to patch in *before* most of the Interrupt Level processing has occurred and **intBotVector** allows the application to patch in *after* most of the Interrupt Level processing has occurred.

Important:	The application should always disable interrupts before loading a new address into either **intTopVector** or **intBotVector**. The program will very likely crash if this precaution is not taken.

System Use of intTopVector and intBotVector

GEOS 64 and GEOS 128 use **intTopVector** to point to **InterruptMain**, a vital function of the Commodore GEOS Interrupt Level, whereas Apple GEOS does not use either of these vectors. The application can use either **intTopVector** or **intBotVector** under Apple GEOS without any worry. However, under Commodore GEOS, an application that uses **intTopVector** should call the address that was originally in **intTopVector** when it is done.This will ensure that the Commodore GEOS **InterruptMain** will be executed properly.

Example:

```
;Install our interrupt routine into intTopVector
InstallInt:
       php                                  ;disable interrupts
       sei
       MoveW  intTopVector,oldTopVector     ;save address of current routine
       LoadW  intTopVector,#MyIntRout       ;install our interrupt routine
       plp                                  ;restore interrupts
       rts

;Remove our interrupt routine from intTopVector, replacing it with old.
RemoveInt
       php                                  ;disable interrupts
       sei
       MoveW  oldTopVector,intTopVector     ;restore old routine
       plp                                  ;restore interrupts
       rts

;My interrupt service routine
MyIntRout:
                                            ;nothing to do yet...
       lda            oldTopVector          ;exit by vectoring through
       ldx            oldTopVector+1        ;old interrupt routine
       jmp            CallRoutine           ;let it rts...
```

Guidlines for Interrupt Level Routines

There are a few general guidelines for any routine that patches into Interrupt Level:

- Keep the routines short. Interrupt level is not the place for time-consuming code.
- Stay away from GEOS. Some routines will work correctly at interrupt level and other won't. Even worse, the ones that won't work might only show this trait after your product has been released and in the hands of users for months. (It is O.K., though, to use **CallRoutine**, as many of the examples in this chapter illustrate.)
- Never clear the interrupt disable bit.

Following these guidelines will keep your Interrupt Level routines as innocuous as possible.

Interrupt Level Pseudo-Code

The following pseudo-code diagrams illustrate the general Interrupt Level constructs in each of the three systems (GEOS 64, GEOS 128, Apple GEOS). This information can be crucial when trying to track down a subtle interaction between the various levels of GEOS.

GEOS 64 and GEOS 128 Interrupt Level

```
CBMInterruptLevel:
{
   / * Context Save:
      Save out any information about the system configuration that we might destroy */
   Save6502Regs();          /* save the status of the A, X, Y, and S registers */
   SaveGEOSRegs();          /* save r0-r15 and a few internal variables*/
   SaveCBMState();          /* save state of Commodore memory banks */
   SetIOIn();               /* set RAM 1 and I/O registers in. Much of Kernal
                               is now inaccessible*/

   DblClicks();             /* decrement dblClickCount if non-zero */

   if  (GEOS128)
   {   DoMouse();           /* GEOS 128 updates mouse here */
       DoSetMouse();        /*and also calls SetMouse in mouse driver. SetMouse
                               doesn't exists in GEOS 64 input drivers.*/
   }

   DoKeyboard();            /* scan the keyboard and add a char to the queue if key pressed */
   DoAlarmSnd();            /* update timer for alarm sound duration */

   / * Application can patch into the following two vectors. The application's routine should
      always end by indirectly calling the routine whose address was originally installed in
      the vector. Use CallRoutine in the Kernal (it's bank is in) in case the pointer is $0000.
      * /
   CallIndirect(intTopVector) /* call indirectly through intTopVector. On the C64/128, this
                                 points to InterruptMain. */
   CallIndirect(intBotVector) /* call indirectly through intBotVector. This is usually
                                 $0000, which CallIndirect ignores. */

   / * Context Restore:
RESTORE  ~~Save out any~~ information about the system configuration that we ~~might destroy~~ SAVED */
   RestoreCBMState();       /* put memory banks back as they were */
   RestoreGEOSRegs();       /*  restore r0-r15,  etc.*/
   Restore6502Regs();       /* restore A, X, Y, and S registers */
   ReturnFromIRQ();         /* pick up where we left off */
}
```

GEOS 64 and GEOS 128 InterruptMain

```
/ *
    InteruptMain:
    Called through  intTopVector under GEOS 64/128. This is *VERY* different from
    InterruptMain under Apple GEOS!
* /

InterruptMain:
{
    if  (GEOS64)
    {   DoMouse();              / * GEOS 64 updates mouse here */
    }
    UpdateProcesses();          / * Update the process timers */
    UpdateSleeps();             / * Update the sleep timers */
    UpdatePrompt();             / * Flash/Update the text prompt */
    GetNewRandom();             / * jsr GetRandom in Kernal */
    Return();
}
```

Apple GEOS Interrupt Level

```
AppleInterruptLevel:
{
    / * Context Save:
        Save out any information about the system configuration that we might destroy */
    Save6502Regs();          /* save the status of the A, X, Y, and S registers */
    SaveAppleState();        /* save state of Apple memory banks */

    /* Set memory configuration to normal/default state */
    RamReadOff();
    RamWriteOff();
    Page2Off();
    AltZPOff();

    IrqMiddle();             /* main IRQ processing */

    / * Context Restore:
        Save out any information about the system configuration that we might destroy */
    RestoreAppleState();     /* put memory banks back as they were */
    Restore6502Regs();       /* restore A, X, Y, and S registers */
    ReturnFromIRQ();         /* pick up where we left off */
}
```

Apple GEOS IrqMiddle

```
/*
    IrqMiddle (APPLE VERSION).
    This is *VERY* different from InterruptMain under CBM GEOS!
    This is where software generated interrupts are sent.
*/
IrqMiddle:
InterruptMain:
{
    SaveGEOSRegs();              /* save r0-r15 and a few internal variables*/

    /* Application can patch into the following vector to get control before most of the
       Interrupt Level processing has occurred. */
    CallIndirect(intTopVector)  /* call indirectly through intTopVector. On the Apple this
                                   defaults to $0000, which CallIndirect ignores */

    /* Apple GEOS draws the mouse cursor (soft sprite #0) at interrupt level so that it can
       minimize flicker by avoiding the raster beam. However, because diffent interrupt
       sources (mouse vs. BSW IRQ Generator) pull the IRQ line low at different times in the
       Vblank sequence, the sprite is either drawn earlier or later depending on its position
       on the screen. */
    if  (intSource != BSWIRQ)      /* if the BSW IRQ card generated the interrupt... */
    {   if  (mouseYPos < 40)       /* if mouse is at the top of the screen... */
        {   doMouseLater = TRUE    /* then avoid the beam by drawing it later */
        }
        else
        {   AppleSoftMouseService();      /* else, draw it now.*/
        }
    }

    DblClicks();                 /* decrement dblClickCount if non-zero */
    DoMouse();                   /* update mouse now */

    /* Turn on aux. memory and give the clock driver control */
    RamReadOn();            /* switch auxiliary memory in */
    RamWrtOn();
    ReadClockInt();              /* call to aux. memory jump table for clock driver */
    RamReadOff();           /* put main memory back in */
    RamWrtOff();

    DoKeyboard();           /* scan the keyboard and add a char to the queue if key pressed */
    UpdateProcesses();           /* Update the process timers */
    UpdateSleeps();              /* Update the sleep timers */
    UpdatePrompt();              /* Flash/Update the text prompt */
    GetNewRandom();         /* jsr GetRandom in Kernal */

    /* Call the auxiliary device driver interrupt code*/
    AuxDInt();

    /* Application can patch into the following vector to get control after most of the
       Interrupt Level processing has occurred. */
```

```
    CallIndirect(intBotVector)  /* call indirectly through intBotVector. This is usually
                                   $0000, which CallIndirect ignores. */

    /* if the BSW IRQ card generated the interrupt and we haven't done the mouse yet... */
    if  ((intSource == BSWIRQ) && (doMouseLater == TRUE))
    {   AppleSoftMouseService();      /* then draw the mouse now */
    }

    RestoreGEOSRegs();              /* restore r0-r15, etc. */
    Return();
}
```

UpdateProcesses

```
UpdateProcesses:
{
    if (numProcesses > 0)       / * Only do this if there are processes in the table */
    {
        for (EachProcess)           /* go through each process in the table */
        {
            if (Process != FROZEN)      /* only if unfrozen... */
            {   DecrementTimer();               /* count down one tick */
                if (Timer == 0)                 /* if timer timed-out
                {   Process = RUNABLE;          /* make it runable */
                    ResetTimer();             /* and reset the counter */
                }
            }
        }
    }
    Return();
}
```

UpdateSleeps

```
UpdateSleeps:
{
    if (numSleeping > 0)        / * Only do this if there routines are sleeping */
    {
        for (EachSleeping)      /* go through each sleeping routine */
        {
            if (SleepTimer              /* if counter not zero, then still asleep! */
            {   Decrement       ;       /* so count down one tick */
            }
        }
    }
}
```

UpdatePrompt

```
UpdatePrompt:
{
    if (alphaFlag(BIT7) == 1)          / * prompt enabled if hi-bit of alphaFlag set */
    {
        DecrementAlphaFlagTimer();      /* dec timer in lower 6 bits of alphaFlag */
        if ((alphaFlag&$3f) == 0)       /* if time to change prompt state */
        {
            /* Toggle the state of the prompt */
            if (PromptState == ON)              /* bit 6 of alphaFlag = 1 */
            {   PromptOff();
            }
            else
            {   PromptOn();
            }
    }
    Return();
}
```

DoMouse

```
DoMouse:
{
    UpdateMouse();              / * call input device driver for new positioning */

    if (mouseOn(MOUSEON_BIT) == 1)  /*if mouse is on... */
    {
        FaultCheck();                       /* check for faults */

        /* Commodore machines draw the mouse here, Apples don't */
        if (GEOS64 || GEOS128)                  /* if CBM machine... */
        {
            DrawSprite(mousePicture)     /* copy mouse picture into sprite data table*/
            PosSprite(mouseXpox,mouseYpos) /* position the sprite */
            if (GEOS64)                         / * if GEOS 64... */
            {   EnablSprite(MOUSE)          /* always enable the sprite each time */
            }
        }
    Return();
}
```

AppleSoftMouseService

```
/* Routine to move/draw the mouse (sprite #0) during interrupt level on the Apple.
   GEOS 128 soft-sprite handler does a similar update during MainLoop.*/

AppleSoftMouseService:
{
    /* Only draw the mouse if sprite #0 is enabled and mouse is not temporarily off */
    if ( (mobenble(BIT_0) == 1) &&  (offFlag(BIT_7) == 0)
    {
        /* OK to update the mouse, only erase if not yet erased */
        if ( offFlag(BIT_6) == 0) )  /* if mouse not yet erased... */
        {
            /* Has the mouse moved since last time? Only erase if no movement */
            if (     ( lastYPos ! = (mouseYPos) ) ||
                     ( lastXPos != (mouseXPos) )     )
            {
                EraseSoftMouse(lastXPos,lastYPos);   /* erase if mouse moved */
            }
        }

        offFlag = FALSE;        /* flag: mouse is on and drawn */

        /* make current position the old position for erasure next time around */
        lastYPos = mouseYPos;
        lastXPos = mouseXPos;

        DrawSoftMouse(mouseXPos,mouseYpos);       /* draw at new position */
    }
    Return();
}
```

FaultCheck

```
FaultCheck:
{
    /* Check mouse against left constraint and left screen edge*/
    if ((mouseXPos < mouseLeft) || (mouseXPos < 0))
    {   mouseXPos = mouseLeft;              /* force mouse to constraint*/
        faultData(OFFLEFT_BIT) = 1;         /* show left fault */
    }

    /* Check mouse against right constraint and right screen edge*/
    if ((mouseXPos > mouseRight) || (mouseXPos > SC_PIX_WIDTH-1))
    {   mouseXPos = mouseRight;             /* stop mouse at edge */
        faultData(OFFRIGHT_BIT) = 1;        /* show right fault */
    }

    /* Check mouse against top constraint and top screen edge*/
    if ((mouseYPos < mouseTop) || (mouseYPos < 0))
    {   mouseYPos = mouseTop;               /* stop mouse at edge */
        faultData(OFFTOP_BIT) = 1;          /* show top fault */
    }

    /* Check mouse against bottom constraint and bottom screen edge*/
    if ((mouseYPos > mouseBottom) || (mouseYPos > SC_PIX_HEIGHT-1))
    {   mouseYPos = mouseBottom;            /* stop mouse at edge */
        faultData(OFFBOTTOM_BIT) = 1;       /* show bottom fault */
    }

    if (mouseOn(MENUON_BIT) == 1)           /* if menus on, see if mouse is off current menu */
    {
        if  (   (mouseYPos < menuTop) ||
                (mouseYPos > menuBottom) ||
                (mouseXPos < menuLeft) ||
                (mouseXPos > menuRight)
            )                                   /* if mouse outside any menu edge... */
            {   faultData(OFFMENU_BIT) = 1;     /* show menu fault */
            }
    }
    Return();
}
```

MainLoop Level Pseudo-Code

The following pseudo-code diagrams illustrate the general MainLoop Level constructs in each of the three systems (GEOS 64, GEOS 128, Apple GEOS). This pseudo-code is useful for determining exactly how icons, menus, and other event-generating mechanisms interact with your application.

MainLoop

```
MainLoop:
{
   while (TRUE)          /* This loop is never ending */
   {
      if (APPLE)                    /* Apple specific */
      {   if (offFlag(BIT7) == 1)   /* If mouse cursor was turned off... */
          {   offFlag = $40;        /* Turn it back on now that we're in MainLoop */
          }
      }

      KeyboardService();        /* service keyboard and related MainLoop functions */
      ProcessService();     /* service processes */
      SleepService();           /* service sleeping routines */

      if (APPLE)            /* Apple differs here, too*/
      {
          AuxDMain();               /* let aux driver's MainLoop routine do what it needs to */
          ReadClock();              /* Get clock driver to set the time and date variables */
          AppleTimeService();   /* service the apple time */

          if (intSource == Software) /* if generating software interrupts... */
          {
              InterruptMain();              /* simulate interrupts */
          }
      }
      else
      {
          CBMTimeService();         /* service the Commodore time */
      }

      CallIndirect(applicationMain);  / * Call any application code that NEEDS to be handled
                                          Every MainLoop */
   } /* endwhile */
}
```

KeyboardService

```
KeyboardService:
{
    if (C128 || APPLE)      /* GEOS 128 and Apple GEOS handle sprites here */
    {       SoftSprHandler();
    }

    /* RUN THROUGH THE BITS IN PRESSFLAG AND DISPATCH AS NECESSARY.
       THESE DISPATCHES GO THROUGH VECTORS THAT TYPICALLY DEFAULT TO
       GEOS ROUTINES FOR HANDLING THE VARIOUS USER-INPUTS */

    /* input device changed vector (currently unused by GEOS) */
    if (pressFlag(INPUT_BIT) == 1)     /* if input device changed */
    {
       pressFlag(INPUT_BIT) = 0) /* clear flag */
       CallIndirect(inputVector)       /* and go through vector <<$0000>>*/
    }

    /* state of mouse changed vector (mouse moved; state of button changed)
       mouseVector usually points to an internal GEOS routine SystemMouseService()*/
    if (pressFlag(MOUSE_BIT) == 1)    /* if mouse state changed... */
    {
       pressFlag(MOUSE_BIT) = 0)     /* clear flag */
       CallIndirect(mouseVector)     /* and go through vector <<SystemMouseService>>*/
    }

    /* keyboard character ready
       keyVector defaults to $0000. */
    if (pressFlag(KEYPRESS_BIT) == 1)    /* if key in queue... */
    {
       keyData = GetCharFromQueue();    /* get keypress */

       if (QUEUE_EMPTY)                 /* if no more keys in the queue... */
       {
          pressFlag(KEYPRESS_BIT) = 0);    /* clear flag */
       }
       CallIndirect(keyVector)             /* go through vector <<$0000>>*/
    }

    /* any mouse faults since last time?
       mouseFaultVec usually points to an internal GEOS routine SystemFaultService()*/
    if (faultData != 0)                 /* if any faults... */
    {
       CallIndirect(mouseFaultVec);  /* go through vector <<SystemFaultService>>*/
       faultData = 0;                /* and clear faults afterward */
    }
    Return();
}
```

ProcessService

```
ProcessService:
{
    if (numProcesses > 0)       / * If no processes, ignore */
    {
        for (EachProcess)           /* go through each process in the table.
                                        (start with last in table & work backward) */
        {
            if ((Process == (RUNABLE & ~BLOCKED))   /* only if runable & not blocked */
            {   Process == ~RUNNABLE;   /* clear runable flag */
                ProcessEvent();             /* and generate a process event by calling the
                                                routine in the table. */
            }
        }
    }
    Return();
}
```

SleepService

```
SleepService:
{
    if (numSleeping > 0)        / * If no slee;                ignore */
    {
        for (EachSleeping)          /* go through each process in the table.
                                        (start with last in table & work backward) */
        {
        if (SleepTimer = 0)             /* if counter zero, then time to awake! */
            {
            RemoveSleep();              /* remove this sleeper from the internal list */
            WakeUp();               /* and go wake it up*/
            }
        }
    }
    Return();
}
```

SytemMouseService

```
SleepService:
{
    if ( mouseData(BIT_7) == DOWN )   /* if mouse button down (bit == 0)... */
    {
        if ( mouseOn(MOUSEON_BIT) == 1 ) /* if mouse checking is on... */
        {
            if ( mouseOn(MENUON_BIT) == 1)   /* if menus scanning is on... */
            {
                /* Check if the mouse is within the currently active menu (level 0/main) */
                if  (    (mouseYPos > menuTop) &&
                         (mouseYPos < menuBottom) &&
                         (mouseXPos > menuLeft) &&
                         (mouseXPos < menuRight)        )
                {
                    MenuService();              /* mouse was pressed on menu, go handle it */
                    Return();                 /* Return without checking icons */
                }
            }
            /* Not on a menu, see if press was on an icon */
            if ( mouseOn(ICONSON_BIT) == 1)   /* if icon scanning is on... */
            {
                /* search through the icon table looking for a match */
                for (EachIcon)
                {
                    if (icon(OFF_I_PIC) != $0000)  / * if icon not disabled... */
                    {
                        if (MouseOnIcon() == TRUE)     /* if mouse on top of this icon... */
                        {
                            /* flash or invert icon as necessary */
                            if (iconSelFlag(ST_FLASH_BIT))    /* flash icon? */
                            {   InvertIcon();                          /* invert once */
                                Sleep(selectionFlash);          /* sleep awhile */
                                InvertIcon();                          /* invert back again */
                            }
                            else if (iconSleFlag(ST_INVERT_BIT))       /* invert icon? */
                            {   InvertIcon();                          /* just invert */
                            }

                            /* check for double click */
                            if (DBL_CLICK)      /* if this is the second click of a dbl click...*/
                            {   r0H = TRUE;         /* set double click flag */
                            }
                            else                        / * else, set single click flag *.
                            {   r0H = FALSE;
                            }

                            /* call the icon event routine*/
                            r0L = icon;          /* tell event routine which icon */
                            CallIndirect(icon(OFF_I_EVENT));   /* generate an event */
```

```
                        Return(); /* break out of the for loop (check no more icons!) */
                    }
                }
            }
        }
    }
    /* If we got here, the following is true:
        1) mouse button was released (as opposed to pressed)
           - or -
        2) mouse was pressed, but not on an icon nor on a menu
     */
    CallIndirect(otherPressVec);  /* it's an "other" press.. "other" as in something the
                                     system doesn't really care about */
}
```

SytemFaultService

```
SystemFaultService:
{
    /* only deal with faults if the mouse is on, menu scanning is enabled, and we've got a
       submenu down... */
    if (    (mouseOn(MOUSEON_BIT) == 1) && (mouseOn(MENUON_BIT) == 1) &&
            (menuNumber > 0)                                                    )
    {
        if (menuType == CONSTRAINED)
        {
            /* for constrained menus... */
            /* If mouse faulted off the top of a vertical menu or off the left of a horizontal
               menu, then we go to the previous menu. Otherwise, the fault is ignored because
               the menu is constrained */
            if (    (menuType == VERTICAL && faultData(OFFTOP_BIT) == TRUE) ||
                    (menuType == HORIZONTAL && faultData(OFFLEFT_BIT) == TRUE)  )
            {
                DoPreviousMenu();
            }
        }
        else    /* menuType == UNCONSTRAINED */
        {
            DoPreviousMenu();           / * always try to go to the previous menu. If mouse didn't
                                          move onto the previous menu, then next pass through
                                          mainloop will see this as a fault and try to remove
                                          that menu, and so on until we're back to the main menu
                                        * /
        }
    Return();
}
```

Alphabetical Listing of Routines

AllocateBlock (Apple, C64, C128) mid-level disk

Function: Allocate a disk block, marking it as in use.

Parameters:

Commodore:

r6L	track number of block (byte).
r6H	sector number of block (byte).

Apple:

r6	block number (word).

Uses: **curDrive**

Commodore:

curDirHead	this buffer must contain the current directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error); Commodore only: returns **BAD_BAM** if block already allocated.
r6	unchanged.

Apple:

c	carry flag is set if block is already in use.

Alters:

Commodore:

curDirHead	BAM updated to reflect newly allocated block.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r7, r8H.**

Description: **AllocateBlock** allocates a single block on this disk by setting the appropriate flag in the allocation map (the BAM on Commodore computers and the VBM on Apple computers).

Commodore: If the sector is already allocated then a **BAD_BAM** error is returned. **AllocateBlock** does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

The Commodore 1541 device drivers do not have a jump table entry for **AllocateBlock**. All other device drivers, however, do. The following subroutine will properly allocate a block on any device, including the 1541.

```
;*******************************************************
;     NewAllocateBlock -- allocate specific block in BAM
;     with any CBM GEOS device driver.
;
;     Pass:     r6L,r6H      track,sector to allocate
;
;     Uses:     BAM in curDirHead
;
;     Returns: x     error status ($00 = success, BAD_BAM =
;                    block already in use, etc.)
;
;     Destroys:      a,y,r7,r8H
;*******************************************************
.if   (C64 || C128)
NewAllocateBlock:
      ldy       curDrive            ; get current drive
      lda       driveType-8,y       ; get drive type
      and       #$0f                ; keep only drive format
      cmp       #DRV_1571           ; see if 1571 or above
      bcc       1541$               ; branch if 1541
      jmp       AllocateBlock       ; else, use driver routine

1541$:
      jsr       FindBAMBit          ; get BAM bit info
      beq       110$                ; if zero, then it's not free
                                    ; otherwise, it's free...
      lda       r8H                 ; get bit mask for BAM
      eor       #$ff                ; convert to clearing mask
      and       curDirHead,x        ; and with BAM byte to clear
                                    ;   bit and show as allocated
      sta       curDirHead,x        ; and store back.
      ldx       r7H                 ; get base of track's entry
      dec       curDirHead,x        ; dec # free blocks this track
      ldx       #$00                ; show no error
      rts                           ; exit

110$:
      ldx       #BAD_BAM            ; show error -- already in use
      rts                           ; exit
.endif
```

Apple: Apple GEOS did not include **AllocateBlock** in its jump table until version 2.1. The following patch places the entry into the jump table, thereby allowing applications to call **AllocateBlock** under version 2.0 (which was the first version of Apple GEOS).

```
.if   (APPLE)
LowSwitch           = $fc16       ; direct entries into Kernal; do not use
o_AllocateBlock     = $45c1       ; except for authorized patches

JMPABS              = $4c         ; JMP absolute opcode
VER_2_0             = $20         ; version 2.0

PatchAllocateBlock:
```

```
     lda      version             ; get Kernal version number
     cmp      #VER_2_0            ; check against version 2.0
     bne      99$                 ; if not v2.0, then no patch necessary
     lda      #JMPABS             ; store jmp LowSwitch into main
     sta      AllocateBlock       ; jump table
     lda      #[LowSwitch         ;
     sta      AllocateBlock+1     ;
     lda      #]LowSwitch         ;
     sta      AllocateBlock+2     ;
     sta      RAMWRT_ON           ; switch in aux bank
     lda      #JMPABS             ; store jmp o_AllocateBlock into aux
     sta      AllocateBlock       ; jump table
     lda      #[o_AllocateBlock   ;
     sta      AllocateBlock+1     ;
     lda      #]o_AllocateBlock   ;
     sta      AllocateBlock+2     ;
     sta      RAMWRT_OFF          ; switch back to main bank
99$: rts                          ; exit
.endif ;(APPLE)
```

Example:

```
     MoveW    DiskBlock, r6       ; block to allocate
.if (C64 || C128)
     jsr      NewAllocateBlock    ; (see above)
     cpx      #BAD_BAM            ; BAD_BAM means block in use
     beq      200$                ; branch if block already in use
     txa                          ; check for other error
     beq      150$                ; branch if no error
     jmp      MyDiskError         ; call error handler with error in x
.else ;(APPLE)
     jsr      AllocateBlock       ; Allocate the block
     php                          ; save status of allocation
     txa                          ; get error status
     beq      100$                ; branch if no error
     plp                          ; error: fix stack
     jmp      MyDiskError         ; call error handler with error in x
100$:
     plp                          ; restore status of allocation
     bcs      200$                ; branch if block already in use
.endif

150$:                             ; block was free and is now allocated
     ;--- code to handle new block goes here ---

200$:                             ; block is not free...
     ;--- code to handle block already allocted goes here ---

MyDiskError:                      ; error occurred...
     ;--- code to handle disk errors goes here ---

.ramsect
DiskBlock      .block 2           ; disk block to allocate
.psect
```

See also: **SetNextFree, BlkAlloc, FreeBlock.**

AppendRecord (Apple, C64, C128) VLIR disk

Function: Adds an empty record after the current record in the index table, moving all subsequent records down one slot to make room.

Parameters: none.

Uses:

curDrive	
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM/VBM into memory.
curRecord	current record pointer
fileHeader	VLIR index table stored in this buffer.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	current directory header/BAM.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:

curRecord	new record becomes the current record.
usedRecords	incremented by one.
fileWritten†	set to **TRUE** to indicate the file has been altered since last updated.
fileHeader	new record added to index table.

Commodore:

curDirHead	directory header read in if **fileWritten** is **FALSE** on call.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r0L, r1L, r4.**

Description: **AppendRecord** inserts an empty VLIR record following the current record in the index table of an open VLIR file, moving all subsequent records down in the record list. The new record becomes the current record. A VLIR file can have a up to **MAX_VLIR_RECS** records (127 on the Commodore and 254 on the Apple). If adding a record exceeds this value, then an **OUT_OF_RECORDS** error is returned.

A record added with **AppendRecord** occupies no disk space until data is written to it. The new record is marked as empty in the VLIR index table. (When a VLIR file is first created by **SaveFile**, all records are marked as unused). Some applications call **AppendRecord** repeatedly after creating a new file until an

OUT_OF_RECORDS error is returned. This marks all the records as used and prepares them to accept data with calls to **WriteRecord.**

Note: **AppendRecord** does not write the VLIR index table out to the disk. Call **CloseRecordFile** or **UpdateRecordFile** to save the index table when all modifications are complete.

Apple: An empty record is marked with $ffff in the VLIR index table (stored in the buffer at **fileHeader**). An unused record is marked with $0000. Use **PointRecord** to check the status of a particular record (unused, empty, or filled).

CBM: An empty record is marked with $ff00 in the VLIR index table (stored in the buffer at **fileHeader**). An unused record is marked with $0000. Use **PointRecord** to check the status of a particular record (unused, empty, or filled).

Example:

```
SaveRecord:
     LoadW    r0, #Filename       ; pointer to filename
     jsr      OpenRecordFile      ; open VLIR file
     txa                          ; check open status
     bne      99$                 ; exit on error
     lda      appendPoint         ; get record to append to
     jsr      PointRecord         ; go to that record
     txa                          ; check point status
     bne      99$                 ; exit on error
     jsr      AppendRecord        ; append a record at this point
     LoadW    r7, #BufStart       ; point at data buffer
     LoadW    r2, #BUFLENGTH      ; bytes in buffer (bufEnd-bufStart)
     jsr      WriteRecord         ; write buffer to record
     txa                          ; get write status
     bne      99$                 ; exit on error
     jsr      CloseRecordFile     ; close VLIR file
99$:
     rts                          ; return with any error in x

.ramsect
appendPoint:  .block 1                    ; record to append to
Filename:     .block NAME_LENGTH+1        ; holds null-terminated filename
BufStart      .block 1024                 ; data buffer
BUFLENGTH     = (* - bufStart)+1          ; length of buffer
.psect
```

See also: **InsertRecord, DeleteRecord, PointRecord.**

AuxDExit (Apple) — aux driver

Function: De-install and remove the current aux driver.

Parameters: none.

Returns: **a** bit 7 set if reinitialization required.

Destroys: assume **x**, **y**, **r0–r15**.

Description: The GEOS deskTop calls **AuxDExit** prior to loading a new aux driver. This allows the current aux driver to deinstall itself properly, removing any hooks it may have placed into the operating system. A RAMdisk aux driver, for example, would need to remove itself from the device table before it can be replaced. The GEOS deskTop and the Configure utility, for example, will call **AuxDExit** prior to replacing the current auxiliary driver.

If bit 7 of **a** is set on return, then this indicates that the device table has changed and a reset is in order. The deskTop, for example, will reinitialize itself if bit 7 is set. This is equivalent to choosing **RESET** from the **options** menu.

Example:

```
;*** SAMPLE AuxDExit ROUTINE ***
;*******************************************************************************
;     o_AuxDExit -- Deinstall RamFactor Auxiliary driver
;
;     Synopsis:      Before loading another aux driver, control vectors
;                    through the AuxDExit entry in the aux driver's jump
;                    table. This is the AuxDExit routine for the Ram Factor
;                    RAM disk driver. We leave the drive in the drive list
;                    but restore the original ProDOS vector.
;
;     Called by:     Aux driver jump table AuxDExit entry.
;
;     Pass:          Nothing
;
;     Returns: X     error status (NOT_IMPLEMENTED is returned if unable to
;                    uninstall
;              A     = 0 for no RESET needed
;                    = $80 if reset needed
;
;*******************************************************************************
AUXRESET       = $80                ; flag to force RESET function if we change
                                    ; the drive tables
AUXNORESET     = $00                ; and for not resetting...
                                    ;
RamExit:                            ;
      lda      #$ff                 ;
      sta      DD_Command           ;
      ldx      curDrive             ; Get current drive
      jsr      IsPatched            ; Check current drive against this driver
      beq      110$                 ; Can't uninstall current drive, do error
      ldy      #AUXNORESET          ; assume no reset necessary
      ldx      numDrives            ; We're OK. go through each drive in the
                                    ; drive table and if it's a RAM factor,
                                    ; then restore the old vector
10$:                                ;
      jsr      IsPatched            ; see if this drive is patched
```

```
        bne     20$                 ; if not, then skip it
        ldy     #AUXRESET           ; found a patched entry, enable RESET
        lda     auxDevTabLo,x       ; and put back the real entry point
        sta     devTabLo,x          ;
        lda     auxDevTabHi,x       ;
        sta     devTabHi,x          ;
20$:                                ;
        dex                         ; try next drive
        bpl     10$                 ; and loop until no more
100$:                               ;
        tya                         ; put RESET flag in accumulator
        ldx     #NO_ERROR           ; and return no error
        rts                         ; exit
110$:                               ;
        ldx     #NOT_IMPLEMENTED    ; return error
        lda     #AUXNORESET         ; no need to RESET
        rts                         ;
                                    ;
IsPatched:                          ; Pass: device number in X
                                    ; Returns: result of compare in ST
        lda     devTabHi,x          ; Check address of this aux patch
        cmp     #]PatchRamFactor    ; against vector in device table for
        bne     20$                 ; equality.
        lda     devTabLo,x          ;
        cmp     #[PatchRamFactor    ;
        rts                         ; exit
```

AuxDInt (Apple) — aux driver

Function: Aux driver Interrupt Level routine.

Parameters: none.

Returns: nothing.

Destroys: assume **a, x, y, r0–r15.**

Description: Apple GEOS calls **AuxDInt** during Interrupt Level. This allows an aux driver to have routines that execute at Interrupt Level. With a hardware interrupt source, Interrupt Level will execute approximately every vertical blank (60 times per second).

An aux driver that has no need for Interrupt Level processing should simply perform an **rts.**

See also: **AuxDMain.**

AuxDKeyFilter (Apple) — aux driver

Function: Aux-driver keypress filter; pre-processes keyboard input so aux driver may interpret special keystrokes.

Parameters: **a** KEY — key as scanned from keyboard circuitry.

Returns: **a** translated character or **NULL** if filtering out.

Destroys: **x, y**

Description: An application does not call **AuxDKeyFilter** directly. The Apple GEOS keyboard scanning routine calls **AuxDKeyFilter** at Interrupt Level immediately after calling **KeyFilter**. This allows the aux driver to translate, remap, or filter-out certain keypresses.

A typical **AuxDKeyFilter** routine compares the *KEY* parameter against a list of specific keypresses. If there is no match, then **AuxDKeyFilter** returns without altering the *KEY* code. If there is a match, **AuxDKeyFilter** can perform some action in response and return a different *KEY* value or a *KEY* value of **NULL**. (A *KEY* value of **NULL** effectively filters out the keypress because GEOS ignores the "null-key" value, never placing the key in the keyboard input queue.)

If *KEY* is **NULL** when **AuxDKeyFilter** is called, then a key was pressed but it has already been processed by some other prior filter (most likeley **KeyFilter**) and should be ignored.

Note: The *KEY* parameter comes from the Apple hardware register **KEYBD_DATA**. GEOS then sets high-bit is set to reflect the state of the **Option** or ♦ key (keyboards that have one, don't have the other). If bit 7 is set, then that key is pressed. To detect the state of the ♦ key, check the **OPEN_APPLE** hardware location.

```
;Check for open-apple key
     bit      OPEN_APPLE          ;check open apple key status
     bmi      OpenPressed         ;branch if pressed
```

Example:

```
;*** SAMPLE AuxDKeyFilter ROUTINE ***
;*****************************************************************************
;     o_AuxDKeyFilter -- key filter routine for screen dump aux driver
;
;     Synopsis:      At each keypress, GEOS vectors through the aux driver's
;                    AuxDKeyFilter jump table entry which points here.
;
;     Called by:     AuxDKeyFilter entry in Aux driver's jump table.
;
;     Pass:          key in A
;
;     Returns:       key value unchanged or NULL if it's the key we're looking
;                    for
;
;*****************************************************************************
MAGIC_KEY      = ($80 | '!')       ;close-apple + shift + 1

o_AuxDKeyFilter:
```

```
        cmp     #MAGIC_KEY          ; check keypress against our key
        bne     99$                 ; if no match, then ignore
        jsr     DoDump              ; else, do screen dump
        lda     #NULL               ; and return a null key
99$:                                ;
        rts                         ; exit

DoDump:
        ;--- code to handle screen dump goes here ---
```

See also: **KeyFilter.**

AuxDMain (Apple) — aux driver

Function: Aux driver MainLoop Level processing.

Parameters: none.

Returns: nothing.

Destroys: assume **a, x, y, r0–r15.**

Description: Apple GEOS calls **AuxDMain** on every pass through **MainLoop**. This allows an aux driver to have MainLoop Level functions.

An aux driver that has no need for MainLoop Level processing should simply **rts.**

See also: **AuxDInt.**

BBMult (Apple, C64, C128) math

Function: Unsigned byte-by-byte multiply: multiplies two unsigned byte operands to produce an unsigned word result.

Parameters:

- **x** OPERAND1 — zero-page address of single-byte multiplicand in the low-byte of a word variable (byte pointer to a word variable).
- **y** OPERAND2 — zero-page address of the byte multiplier (byte pointer to a byte variable).

Note: *result= OPERAND1(byte) * OPERAND2(byte).*

Returns: x, y, and byte pointed to by *OPERAND2* unchanged.
word pointed to by *OPERAND1* contains the word result.

Destroys: a, r7L, r8

Description: **BBMult** is an unsigned byte-by-byte multiplication routine that multiplies two bytes to produce a 16-bit word result (low/high order). The byte in *OPERAND1* is multiplied by the byte in *OPERAND2* and the result is stored as a word back in *OPERAND1*. Note that*OPERAND1* starts out as a byte parameter but becomes a word result with the high-byte at *OPERAND 1*+1.

Note: Because **r7** and **r8** are destroyed in the multiplication process, they cannot be used to hold either operand.

No overflow can occur when multiplying two bytes becuase the result always fits in a word ($ff*$ff = $fe01).

Example:

```
; Multiply r1L by r1H and store the word result in r2
     MoveB r1L,r2L              ; r2L <- r1L copy of OPERAND1
     ldx      #r2L              ; point to copy of OPERAND1 (r2L)
     ldx      #r1H              ; point to OPERAND2 (r1H)
     jsr      BBMult            ; r2 <- r2L * r2H  do multiplication
```

SeeAlso: **BMult, DMult, Ddiv, DSdiv.**

Bell (Apple) utility

Function: Makes a brief beeping sound through the Apple's internal speaker.

Parameters: none.

Returns: nothing.

Destroys: x, y

Description: **Bell** sounds a 1000 Hz signal through the Apple's internal speaker. The sound lasts approximately 1/10th of a second.

Note: **Bell** does not return until after the full duration of the sound. Interrupts are disabled during this period. The interrupt-disable status is restored when **Bell** returns.

C64 & C128: The following routine provides a bell sound under Commodore GEOS. It is provided for portability.

```
.if (C64 || C128)          ; only for Commodore versions; use Kernal
                           ; routine on Apple.
;*****************************************************************************
;Bell          -- Make a bell sound on C64/128
;
;Author:       Dan Kaufman (w Chris Hawley)
;                                                              (mgl)
;*****************************************************************************
;
;Pass:         Nothing
;Return:       Nothing
;Destroyed:    a
;
;Synopsis:     This routine allows you to have the c64 beep when the user
;              makes a mistake or does something he shouldn't.
;
;*****************************************************************************

sidBase        = $D400
voice1Regs = sidBase
     freqLo1        = voice1Regs
     freqHi1        = voice1Regs+1
     PWLo1          = voice1Regs+2
     PWHi1          = voice1Regs+3
     controlReg1    = voice1Regs+4
     att_dec1       = voice1Regs+5
     sus_rel1       = voice1Regs+6

     FCLo           = voice1Regs+7+7+$7
     FCHi           = voice1Regs+7+7+$8
     res_filt       = voice1Regs+7+7+$9
     mode_vol       = voice1Regs+7+7+$A

     pulse          = %01000001
     SOUND_ON       = $30

Bell:
     PushB    CPU_DATA                    ;switch to I/O space
     LoadB    CPU_DATA,#IO_IN
```

```
        LoadB   controlReg1,#0          ;twiddle sound chip
        sta     att_dec1
        LoadB   mode_vol,#$18
        LoadB   sus_rel1,#SOUND_ON
        LoadW   PWLo1,#$800
        LoadB   FCLo,#0
        sta     FCHi
        sta     res_filt
        LoadB   att_dec1,#6
        LoadB   sus_rel1,#0
        LoadB   freqLo1,#$DF
        LoadB   freqHi1,#$25
        LoadB   controlReg1,#pulse
        PopB    CPU_DATA                ;return to memory space
        rts
.endif ;(C64 || C128)
```

Example:

```
;*** Beep three times ***
;     Runs off of MainLoop by using Sleep
BELL_INTERVAL = (FRAME_RATE/10)   ;approx. 1/10 second
BeepThrice:
        jsr     Bell              ; sound the bell
        LoadW   r0,#BELL_INTERVAL ; pause a bit
        jsr     Sleep             ;
        jsr     Bell              ; sound the bell again
        LoadW   r0,#BELL_INTERVAL ; pause a bit
        jsr     Sleep             ;
        jmp     Bell              ; sound the bell again and let bell rts .
```

BitmapClip (Apple, C64, C128) graphics

Function: Place a rectangular subset of a compacted bitmap on the screen.

Parameters:

r0	DATA — pointer to the compacted bitmap data (word).
r1L	XPOS — x card position: pixel_position /8 (byte).
r1H	Y — y-coordinate (byte).
r2L	W_WIDTH — clipping window width in cards: pixel_width/8 (byte).
r2H	W_HEIGHT — height in pixels of clipping window (byte).
r11L	DX1 — delta-x1: offset of left edge of clipping window in cards from left edge of full bitmap (byte).
r11H	DX2 — delta-x2: offset of right edge of clipping window in cards from right edge of full bitmap (byte).
r12	DY1 — delta-y1: offset of top edge of clipping window in pixels from top edge of full bitmap (word).

*where the upper-left corner of the clipped bitmap (the window) is placed at (XPOS*8, Y). The lower-right corner is at ([XPOS*8]+[W_WIDTH*8], Y+W_HEIGHT).*

Uses: **dispBufferOn:**

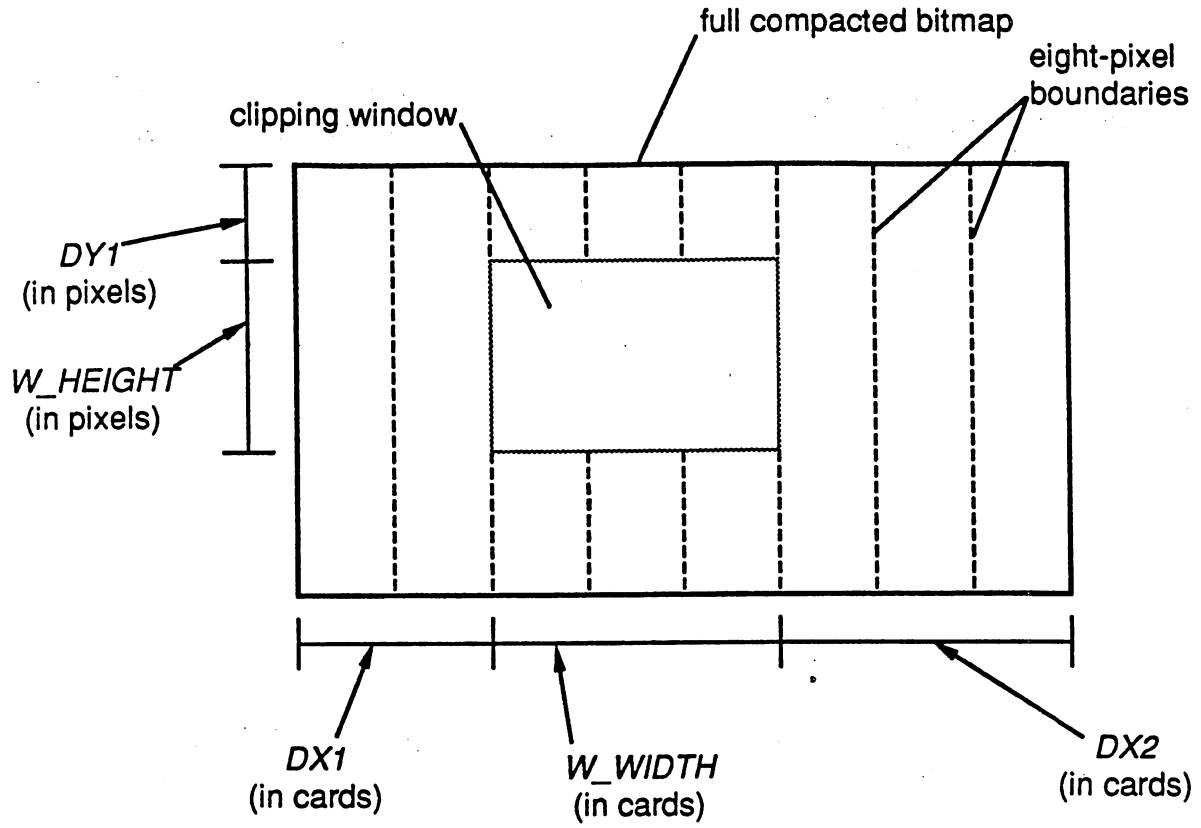
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing.

Destroys: **a, x, y, r0–r12**

Description: **BitmapClip** uncompacts a rectangular area of a full bitmap, clipping (ignoring) any data that exists ouside of the desired area. The rectangular subset is called the *clipping window.*

The following diagram illustrates the eight **BitmapClip** parameters:



No checks are made to determine if the data, dimensions, or positions are valid. Be careful to pass accurate values. Do not pass a value of $00 for either the *W_WIDTH* or *W_HEIGHT* parameters, and pay special attention to the fact that *XPOS*, *W_WIDTH*, *DX1*, and *DX2* are specified in cards (groups of eight pixels horizontally), not in individual pixels.

Note: It may be helpful to think of *DY1* as the number of scanlines in the bitmap to skip initiallly, to think of *W_HEIGHT* as the number of scanlines to display, to think of *DX1* as the number of cards to skip at the beginning of each scanline, to think of *W_WIDTH* as the number of cards to display, and to think of *DX2* as the number of bytes to skip at the end of each scanline.

C128: Under GEOS 128, OR'ing **DOUBLE_B** into the *XPOS* and *W_WIDTH* parameters automatically doubles the x-position and the width of the bitmap (respecitvely) when running in 80-column mode.

BitmapClip in the first release of GEOS 128 does not call **TempHideMouse** to disable the sprites and does not properly double the width when drawing to the 80-column screen. On Kernals where the **release** byte is greater than $01, these problems have been fixed.

Apple: **DOUBLE_B** may be OR'ed into the *W_WIDTH* parameter to double the bitmap width and **INAUX_B** may be OR'ed into the *XPOS* parameter to specify auxiliary memory.

Example:

```
;*****************************************************************************
;     DisplayImage   --      General purpose routine to display a portion of
;                            compacted bitmap image in a window
;
;     Pass:          pixBuf        compacted bitmap image in pseudo-photoscrap
;                                  format. Byte 0 is card width of image.
;                                  Byte 1 and 2 is the pixel height (word).
;                                  The compacted image data starts at byte 3.
;                    xOffset       card index into bitmap to display
;                    yOffset       pixel index into bitmap to display
;
;     Destroys:      a,x,y, r0-r12
;
;*****************************************************************************
WINDOW_X       = 4                 ; card x-position of window
WINDOW_Y       = 30                ; pixel y-position of window
WINDOW_WIDTH   = 5                 ; card width of window
WINDOW_HEIGHT  = 60                ; pixel height of window
                                   ;
DisplayImage:                      ;
                                   ;
;*** Set up initial parameters ***;
     LoadW     r0,#PixImage        ; r0 <- compacted picture data (DATA)
     LoadB     r1L,#WINDOW_X       ; r1L <- left edge of window (XPOS)
     LoadB     r1H,#WINDOW_Y       ; r1H <- top edge of window (Y)
     LoadB     r2L,#WINDOW_WIDTH   ; r2L <- width of window (W_WIDTH)
     LoadB     r2H,#WINDOW_HEIGHT  ; r2H <- height of window (W_HEIGHT)
     MoveB     xOffset,r11L        ; r11L <- x offset into bitmap (DX1)
     MoveW     yOffset,r12         ; r12 <- y offset into bitmap (DY1)
                                   ;
;*** Clip to window ***            ;
     lda       PixWidth            ; get bitmap width
     sec                           ;
     sbc       WINDOW_WIDTH        ; subtract window width
;--- following line changed to    ;
;--- use zero-page copy           ;
;    sbc       xOffset             ; subtract x-index into bitmap
     sbc       r11L                ; now we have the right edge clip distance
     sta       r11H                ; r11H <- right edge clip (DX2)
     bpl       10$                 ; if we're >0, branch to skip x clipping
     adc       #WINDOW_WIDTH       ; add back the window width
     sta       r2L                 ; make that the new clip window
     LoadB     r11H,#0             ; r11H <- $00 (fixes underflow of DX2)
10$:                               ;
     lda       PixHeight           ; subtract window height from bitmap height
     sec                           ; (two byte subtraction)
     sbc       #WINDOW_HEIGHT      ;
     sta       r3L                 ; store intermediateresult in r3
     lda       PixHeight+1         ;
     sbc       #0                  ;
     sta       r3H                 ;
     lda       r3L                 ; now subtract y index into bitmap
     sec                           ;
;--- following line changed to    ;
;--- use zero-page copy           ;
;    sbc       yOffset             ;
     sbc       r12L                ; (r12 = yOffset)
     sta       r3L                 ;
     lda       r3H                 ;
;--- following line changed to    ;
;--- use zero-page copy           ;
;    sbc       yOffset+1           ;
```

```
        sbc     r12H                ; (r12 = yOffset)
;       sta     r3H                 ; value in r3H never used after this
        bpl     20$                 ; branch if no underflow
        lda     r3L                 ;
        adc     #WINDOW_HEIGHT      ; correct for underflow
        sta     r2H                 ;
20$:                                ;
        jsr     BitmapClip          ; display the bitmap with clipping
99$:                                ;
        rts                         ;exit
                                    ;
.ramsect                            ;
  ffset         .block 1            ;   rd x index into bitmap (byte)
    fset        .block 2            ; pixel y index into bitmap (word)
    .if         .block $800         ; 2K picture buffer
.....dth        PixBuf+0            ; width of picture in cards (byte)
Pixheight       PixBuf+1            ; height of picture in pixels (word)
PixImage        PixBuf+3            ; start of bitmap image
.psect                              ;
```

BitmapUp, i_BitmapUp (Apple, C64, C128) graphics

Function: Place a compacted bitmap onto the screen.

Parameters: Normal:

r0	DATA — pointer to the compacted bitmap data (word).
r1L	XPOS — x card coordinate: pixel_position /8 (byte).
r1H	Y — y-coordinate (byte).
r2L	WIDTH — width in cards: pixel_width/8 (byte).
r2H	HEIGHT — height in pixels (byte).

Inline:
data appears immediately after the **jsr i_BitmapUp**

.word	DATA	pointer to the compacted bitmap data.
.byte	XPOS	x card position: pixel_position /8.
.byte	Y	y-coordinate.
.byte	WIDTH	width in cards: pixel_width/8.
.byte	HEIGHT	height in pixels.

*where the upper-left corner of the bitmap is placed at (XPOS*8,Y). The lower-right corner is at (XPOS*8+WIDTH*8,Y+HEIGHT).*

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing.

Destroys: Commodore:
a, x, y, r0–r9L

Apple:
a, x, y, r0–r2

Description: **BitmapUp** uncompacts a GEOS compacted bitmap according to the width and height information and places it at the specified screen position. No checks are made to determine if the data, dimensions, or positions are valid, and bitmaps which exceed the screen edge will not be clipped. Be careful to pass accurate values. Do not pass a $00 for the *WIDTH* or the *HEIGHT* parameter, and pay special attention to the fact that both the x-position and the width are specified in cards (groups of eight pixels horizontally), not in pixels.

128: Under GEOS 128, OR'ing **DOUBLE_B** into the *XPOS* and *WIDTH* parameters will automatically double the x-position and the width (respectively) in 80-column mode. The first release of GEOS 128 did not properly remove the sprites before placing the bitmap on the screen. The easiest way to correct for this is to always precede a call to **BitmapUp** with a call to **TempHideMouse**. The redundant call to **TempHideMouse** when running under later releases is minimal compared to the number of cycles it takes to decompact and draw the bitmap.

```
jsr     TempHideMouse       ;correct for bug in release 1 of GEOS 128
jsr     BitmapUp            ;then put up the bitmap
```

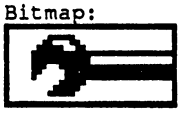
Apple: Under Apple GEOS, **DOUBLE_B** may be OR'ed into the *WIDTH* paramter to double the bitmap width and **INAUX_B** may be OR'ed into the *X1* parameter to specify auxiliary memory.

```
;Put a bitmap up from an address in
;auxiliary memory and doubling the
;picture's width.
.if (APPLE)
     LoadW    r0,#MyAuxBitmap            ; r0 <- aux address of bitmap
     LoadW    r1L,#(MY_CXPOS|INAUX_B)    ; r3 <- card x-pos + in-aux
     LoadB    r1H,#MY_YPOS               ; r1H <-y-coordinate
     LoadB    r2L,#(MY_CWIDTH|DOUBLE|W)  ; r2L <- card width + double bit
     LoadB    r2H,#MY_HEIGHT             ; r2H <- height
     jsr      BitmapUp                   ; put bitmap on screen
.endif
```

Example:

```
;******************
;*** ShowBitmap ***
;******************
; For C64, C128, or Apple
.if   (C128 && C64)
     .echo ShowBitmap cannot assemble for both C64 and C128 at the same time
.endif

BM_XPOS        = (32/8)                  ;card x-position of bitmap
BM_YPOS        = 20                      ;y-position of bitmap
                                         ;
Bitmap:                                  ;
```



```
                                         ;
BM_WIDTH       = PicW                    ; card width of bitmap
BM_HEGHT       = PicH                    ; bitmap height
                                         ;
                                         ;
                                         ; Place the bitmap on the screen,
                                         ; loading the registers with
                                         ; inline data (note double-width
ShowBitmap:                              ; settings).
     lda       #(ST_WR_FORE | ST_WR_BACK) ; use both screens
     sta       dispBufferOn              ;
  .if (C128)                             ; bug fix for 128 release 1
     jsr       TempHideMouse             ;   remove sprites
  .endif                                 ;
     jsr       i_BitmapUp                ; inline bitmap call
     .word     Bitmap                    ;      *bitmap address
     .byte     (BM_XPOS|(DOUBLE_B&(!APPLE*-1))); *xpos (can't dbl on apple)
     .byte     BM_YPOS                   ;      *ypos
     .byte     (BM_WIDTH|DOUBLE_B)       ;      *width
     .byte     BM_HEIGHT                 ;      *height
90$: rts                                 ; exit
```

See also: **BitmapClip, BitOtherClip, NewBitUp.**

BitOtherClip (Apple, C64, C128) graphics

Function: Special version of **BitmapClip** that allows the compacted bitmap data to come from a source other than memory (e.g., from disk).

Parameters:

r0	BUFFER — pointer to a 134-byte buffer area (word).
r1L	XPOS — x-position in bytes: pixel_position /8 (byte).
r1H	Y — y-coordinate (byte).
r2L	W_WIDTH — width in cards of clipping window: pixel_width/8 (byte).
r2H	W_HEIGHT — height in pixels of clipping window (byte).
r11L	DX1 — delta-x1: offset of left edge of clipping window in cards from left edge of full bitmap (byte).
r11H	DX2 — delta-x2: offset of right edge of clipping window in cards from right edge of full bitmap (byte).
r12	DY1 — delta-y1: offset of top edge of clipping window in pixels from top edge of full bitmap (word).
r13	APPINPUT — pointer to application-defined input routine. Called each time a byte from a compacted bitmap is needed; data byte is returned in the a-register.
r14	SYNC — pointer to synchronization routine. Called after each bitmap packet is decompressed. Due to improvements in **BitOtherClip**, this routine need only consist of reloading **r0** with the *BUFFER* address.

*where the upper-left corner of the clipped bitmap (the window) is placed at (XPOS*8, Y). The lower-right corner is at ([XPOS*8]+[W_WIDTH*8], Y+W_HEIGHT).*

Uses: **dispBufferOn:**

bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing.

Destroys: **a, x, y, r0–r12,** and the 134-byte buffer pointed at by **r0.**

Description: **BitOtherClip** allows the application to decompress and display a bitmap without storing the compressed bitmap in memory. Call **BitOtherClip** with the address of an input routine (*APPINPUT*). Each time **BitOtherClip** needs another byte, it calls this routine. The *APPINPUT* routine is expected to read data from the disk or some other device and return a single byte each time it is called.

The basic width, height, position, and clipping window parameters the same as those for **BitmapClip**. Refer to the documentation of that routine for more information.

BitOtherClip calls the user-supplied *APPINPUT* routine until it has enough bytes to form one bitmap packet. *APPINPUT* must preserve **r0-r13** and return the data byte in the accumulator. A typical *APPINPUT* routine saves any registers it might destroy, calls **ReadByte** to get a byte from a disk file, places the byte in the **BitOtherClip** buffer (pointed at by **r0**), then returns, as illustrated in the following example.

```
AppInput:
     PushW    r1                ;save r1, r4, and r5
     PushW    r4                ; (saved for calls to ReadByte routine
     PushW    r5                ;
     MoveW    saveR1,r1         ; r1 <- saveR1
     MoveW    saveR5,r5         ; r5 <- saveR5
     LoadW    r4,#diskBlkBuf    ; r4 <- disk buffer we use
     jsr      MyReadByte        ; get a byte from the file
                                ; (byte is in A)
     stx      error             ; save any error
.if (APPLE)                     ; Apple needs to use buffer pointer
     sta      ALTZP_ON          ; that is on Alt zero-page
.endif                          ;
     ldy      #$00              ; null indirection index
     sta      (r0),y            ; store byte into buffer
.if (APPLE)                     ; Apple needs to
     sta      ALTZP_OFF         ; restore normal zero-page
.endif                          ;
     MoveW    r5,saveR5         ; r5 -> saveR5
     MoveW    r1,saveR1         ; r1 -> saveR1
     PopW     r5                ; restore r1, r4, and r5
     PopW     r4                ;
     PopW     r1                ;
     rts                        ; exit
```

When **BitOtherClip** detects a complete packet, it uncompacts the data from the buffer to the screen. After the bitmap packet has been uncompacted, **BitOtherClip** calls the *SYNC* routine supplied by the caller. The *SYNC* routine prepares the bitmap buffer for the next packet by reloading **r0** with the address of *BUFFER* and performing an **rts.**

```
Sync:
     LoadW    r0,#ClipBuffer    ; reset the pointer
     rts                        ; exit
```

128: Under GEOS 128, OR'ing **DOUBLE_B** into the *XPOS* and *W_WIDTH* parameters will automatically double the x-position and the width (respecitvely) in 80-column mode.

Apple: **DOUBLE_B** can be OR'ed into the *WIDTH* paramter to double the bitmap width and **INAUX_B** can be OR'ed into the *XPOS* parameter to specify auxiliary memory. Also on Apple GEOS, the *APPINPUT* routine must use the alternate zero-page version of **r0** to index into **BitOtherClip**'s compaction buffer.

```
.if (APPLE)                     ; Apple needs to use buffer pointer
     sta      ALTZP_ON          ; that is on Alt zero-page
     ldy      #$00              ; null indirection index
     sta      (r0),y            ; store byte into buffer
     sta      ALTZP_OFF         ; restore normal zero-page
.endif                          ;
```

Note: Do not pass a value of $00 for either the *W_WIDTH* or *W_HEIGHT* parameters.

Example:

```
*** CONSTANTS ***
NO_PICTURE     = -1             ; no picture error. MUST BE NON-ZERO

;window coordinates and dimensions
```

```
WIN_CRDX        = 5                 ; card x-position
WIN_CRDWIDTH    = 12                ; card width
WIN_Y           = 40                ; y-position
WIN_HEIGHT      = 110               ; height

*** VARIABLES ***
.ramsect
saveR1          .block 2            ; temp save for GEOS registers that need to
saveR5          .block 2            ;   be preserved between calls to ReadByte
leftOffset      .block 1            ; scroll x-index into bitmap
topOffset       .block 2            ; scroll y-index into bitmap
picWidth        .block 1            ; bitmap card width
picLength       .block 2            ; bitmap card height
ClipBuffer      .block 135          ; BitOtherClip buffer (+1 for safety)
.psect

;*******************************************************************************
;DrawPhoto
;*******************************************************************************
;DESCRIPTION:  Reads a picture in from a photo album record and draws it
;              clipped to a window. Scroll values allow a specific portion
;              of the bitmap to be shown.
;
;PASSED:       Open VLIR album file with photo scraps in records
;              curPhoto      record to use
;              leftOffset    scroll value on x
;              topOffset     scroll value on y
;
;RETURNS:
;              picWidth      from photo record
;              picLenght     from photo record
;              x             error
;*******************************************************************************
DrawPhoto:
      jsr       ClearWindow         ; clear the drawing window
      jsr       GetPicSize          ; get the size of the picture
      txa                           ; check for error
      bne       99$                 ;
      jsr       SetUpPhoto          ; set up clipping parameters
                                    ; carry comes back set if we can draw
      ldx       #NO_ERROR           ; no errors yet
      bcc       99$                 ; skip drawing if necessary
      jsr       PutUpPhoto          ; draw photo from the record
99$:                                ;
      rts                           ; exit with error in x


;*******************************************************************************
;ClearWindow
;*******************************************************************************
;DESCRIPTION:  Erase the window areas where we plan to put the bitmap
;
;PASS:         Nothing
;
;RETURNS:      curPattern    = 0
;
;DESTROYS:     CBM:          a, x, y, r5-r8, r11
;              APPLE:        a, x, y, r11
;*******************************************************************************
ClearWindow:
      LoadB     curPattern,#0               ;use blank fill pattern
      jsr       i_Rectangle
      .byte     WIN_Y
```

```
        .byte   (WIN_Y+WIN_HEIGHT)
        .word   (WIN_CRDX * 8)
        .word   (WIN_CRDX*8 + WIN_CRDWIDTH*8)
        rts


;*************************************************************************
;SetUpPhoto
;*************************************************************************
;DESCRIPTION: Set up clipping regions and other parameters
;
;PASS:          picWidth     card width of bitmap
;               picLength    height of bitmap
;               leftOffset   card scroll index into bitmap
;               rightOffset  line scroll index into bitmap
;
;RETURNS:       carry        set = OK to draw
;                            clear = don't draw (lies outside of region.)
;               r0    BitOtherClip buffer
;               r1L   Card x-position of window
;               r1H   y-position of window
;               r2L   number of cards of bitmap to display in window
;               r2H   number of lines of bitmap to display in window
;               r12   lines to skip on top
;               r11L  cards to skip on left
;               r11H  cards to skip on right
;
;DESTROYS:      a
;*************************************************************************
SetUpPhoto:
        LoadW   r0, #ClipBuffer     ; r0 <- buffer for BitOtherClip's use
        LoadB   r1L, WIN_CRDX       ; r1L <- window's card x-position
        LoadB   r1H, WIN_Y          ; r1H <- window's y-position
        lda     picWidth            ; A <- (picWidth-leftOffset)
        sec                         ;   (difference between width of
        sbc     leftOffset          ;   picture and offset into picture)
        bcc     20$                 ; If offset exceeds width, then skip
        beq     20$                 ;   over picture draw
        cmp     #WIN_CRDWIDTH               ; If width to display exceeds width
        bcc     8$                  ;   of bitmap, then display as much as
        lda     #WIN_CRDWIDTH               ;   will fit in the window.
8$:                                 ;
        sta     r2L                 ; r2L <- card width to display
                                    ;
        lda     picLength           ; A <- (picLength-topOffset)
        sec                         ;   (difference between height of
        sbc     topOffset           ;   picture and offset into picture)
        bcc     20$                 ; If offset exceeds height, then skip
        beq     20$                 ;   over picture draw
        cmp     #WIN_HEIGHT         ; If height to display exceeds height
        bcc     15$                 ;   of bitmap, then display as much as
        lda     #WIN_HEIGHT         ;   will fit in the window.
15$:                                ;
        sta     r2H                 ; r2H <- pixel height to display
                                    ;
        MoveW   topOffset,r12       ; r12 <- lines to skip on top
        MoveW   leftOffset,r11L     ; r11L <- cards to skip on left
                                    ;
        lda     picWidth            ;
        sec                         ;
        sbc     r2L                 ;
        sbc     leftOffset          ;
        sta     r11H                ; r11H <- cards to skip on right
```

```
                                        ;
     clc                                ; flag as OK to draw
     rts                                ; exit
20$:                                    ;
     sec                                ; flag not to draw
     rts                                ;

;***************************************************************************
;PutUpPhoto
;***************************************************************************
;DESCRIPTION: Draw photo from record.
;
;RETURNS:     x    error
;
;DESTROYS:    a,x,y,r0-r15
;***************************************************************************
PutUpPhoto:
     jsr      GetPicSize            ; reload picture length and width
     txa                            ; check for error or no picture
     bne      99$                   ; leave on error
     jsr      Sync                  ; r0 <- clipBuffer
     LoadW    r13,#AppInput         ; r13 <- AppInput routine
     LoadW    r12,#Sync             ; r12 <- Sync routine
     LoadB    error,#NO_ERROR       ; start out with no error
     jsr      BitOtherclip          ; display photo
     ldx      error                 ; put any error into x
99$:                                ;
     rts                            ; exit


;***************************************************************************
;AppInput
;***************************************************************************
;DESCRIPTION: Bitmap input routine called by BitOtherClip. Returns a single
;             byte of the uncompacted bitmap.
;
;APPLE:       This routine resides in main memory on the Apple, so the
;             ClipBuffer should also be in main memory.
;
;RETURNS:     bitmap byte in BitOtherClip's buffer (off of r0)
;             any error in error
;***************************************************************************
AppInput:
     PushW    r1                    ;save r1, r4, and r5
     PushW    r4                    ;
     PushW    r5                    ;
     MoveW    saveR1,r1             ; r1 <- saveR1
     MoveW    saveR5,r5             ; r5 <- saveR5
     LoadW    r4,#diskBlkBuf        ; r4 <- disk buffer we use
     jsr      MyReadByte            ; get a byte from the file
                                    ; (byte is in A)
     stx      error                 ; save any error
.if (APPLE)                         ; Apple needs to use buffer pointer
     sta      ALTZP_ON              ; that is on Alt zero-page
.endif                              ;
     ldy      #$00                  ; null indirection index
     sta      (r0),y                ; store byte into buffer
.if (APPLE)                         ; Apple needs to
     sta      ALTZP_OFF             ; restore normal zero-page
.endif                              ;
     MoveW    r5,saveR5             ; r5 -> saveR5
     MoveW    r1,saveR1             ; r1 -> saveR1
```

```
	PopW	r5			; restore r1, r4, and r5
	PopW	r4			;
	PopW	r1			;
	rts				; exit


;***************************************************************************
;Sync
;***************************************************************************
;DESCRIPTION: Dumb synchronization routine needed by BitOtherClip
;
;***************************************************************************
Sync:
	LoadW	r0,#ClipBuffer		; reset the pointer
	rts				; exit


;***************************************************************************
;GetPicSize
;***************************************************************************
;DESCRIPTION: Get picture size and other misc. setup for PutUpPhoto
;
;RETURNS:	x	error
;***************************************************************************
GetPicSize:
	PushW	r1			; save r1 and r4
	PushW	r4			;
	lda	curPhoto		; get current photo's record number
	jsr	PointRecord		; point to that record
					;     r1 <- block# of first record
	lda	r1L			; make sure there's something there
.if (APPLE)				;
	ora	r1H			; (check both bytes on Apple)
.endif					;
	bne	10$			; branch if valid record found
	ldx	#NO_PICTURE		; otherwise, flag no picture
;--- Following line changed to save bytes ---
;	bra	40$			; and exit
	bne	40$			; unconditional (NO_PICTURE != 0)
10$:					;
	jsr	SetUpReadByte		; prepare for ReadByte
	txa				; check status
	bne	40$			; exit on error
	jsr	ReadSizeBytes		; read the size bytes out of the record
					; and store them in the photo size
					; variables (error comes back in x)
	MoveW	r1,saveR1		; save off r1 and r5
	MoveW	r5,saveR5		;
;--- Following line removed to let error propagate back ---
;	ldx	#NO_ERROR		; we got this far, no errors found...
40$:					;
	PopW	r4			; restore r4 and r5
	PopW	r5			;
	rts				; exit


;***************************************************************************
;SetUpReadByte
;***************************************************************************
;DESCRIPTION: Set up variables and stuff for ReadByte
;
;STRATEGY:
;	Special case Apple because it requires an extra step. The first block in
```

```
;     an Apple VLIR record is actually an index block with pointers to all the
;     other blocks in the chain. We read this block into the aux memory buffer
;     INDEXBLOCKBUF so ReadByte has something to work with (this requires that
;     ReadByte be called from aux memory using JsrToAux). The general disk
;     buffer diskBlkBuf is used for the intermediate blocks since it is in
;     high aux memory and is always visible (thus eliminating the need to
;     set RWbank).
;
;PASS:        APPLE:       r1            block number of chain's index block
;             CBM:         r1L,r1H       Track/sector of first block in chain
;
;RETURNS:     r1, r4, r5 set up for ReadByte
;             x = error code
;
;DESTROYS:    a, y
;***************************************************************************
SetUpReadByte:
.if (APPLE)                          ;
     LoadW    r4,#INDEXBLOCKBUF      ; r4 <- buffer for index block
     LoadB    RWbank,#AUX            ; set disk read to proper bank
     jsr      GetBlock               ; read in the chain's index block
                                     ; (error now in x)
     LoadB    RWbank,#MAIN           ; restore disk bank to normal
.else ;(CBM)                         ;
     ldx      #NO_ERROR              ; no error on CBM
.endif                               ;
     MoveW    r4,r1                  ; r1 <- track/sector of 1st block (CBM)
                                     ;    <- index block pointer (APPLE)
     LoadW    r4,diskBlkBuf          ; r4 <- disk buffer for ReadByte
     lda      #0                     ; r5 <- $0000 (for ReadByte)
     sta      r5L                    ;
     sta      r5H                    ;
     rts                             ; exit with error in X


;***************************************************************************
;MyReadByte
;***************************************************************************
;DESCRIPTION: My version of ReadByte that deals with all machine weirdness.
;             This routine expects to use the general-purpose disk buffer at
;             diskBlkBuf because, since it is in high-aux on the Apple, it
;             avoids the bank switching complications.
;
;             Call wherever you would normally call ReadByte. Call
;             SetUpReadByte to set the proper parameters prior to calling.
;
;             This would make a good Macro.
;
;PASS:        Same as ReadByte
;
;RETURNS:     Same as ReadByte
;
;DESTROYS:    Same as ReadByte
;***************************************************************************
MyReadByte:

.if (APPLE)                          ; APPLE
     lda      #[ReadByte             ;   Call ReadByte from aux memory
     ldx      #]ReadByte             ;   so that the aux memory
     jsr      JsrToAux               ;   INDEXBLOCKBUF is visible

.else ;(CBM)                         ; CBM
     jsr      ReadByte               ; just use normal ReadByte
```

```
.endif
     rts                          ; exit


;*****************************************************************************
;ReadSizeBytes
;*****************************************************************************
;DESCRIPTION:  Read in the first three bytes from the current record and
;              store them in the size variables.
;
;PASS:         File open and system ready for calls to ReadByte.
;
;RETURNS:      r1, r4, r5 ready for ReadByte calls to load bitmap data
;              x             error
;              picWidth      card width of bitmap
;              picLength     line height of bitmap
;
;DESTROYS:     a, y
;*****************************************************************************
ReadSizeBytes:
     jsr      MyReadByte           ; get photo width
     sta      picWidth             ;
     txa                           ; check for error
     bne      99$                  ;
     jsr      MyReadByte           ; get photo length (low byte)
     sta      picLength            ;
     txa                           ; check for error
     bne      99$                  ;
     jsr      MyReadByte           ; get photo length (high byte)
     sta      picLength+1          ;
99$:                               ;
     rts                           ; exit error in x
```

BldGDirEntry (Apple, C64, C128) mid-level disk

Function: Builds a directory entry in memory for a GEOS file using the information in a file header. Apple GEOS version can also build an entry for a folder (subdirectory).

Parameters:

Commodore:

r2 NUMBLOCKS — number of blocks in file (word).

r6 TSTABLE — pointer to a track/sector list of unused blocks (unused but allocated in the BAM), usually a pointer to **fileTrScTab**; **BlkAlloc** can be used to build such a list (word).

r9 FILEHDR — pointer to GEOS file header (word).

Apple:

r2 NUMBLOCKS — number of blocks in file or $0000 to create a subdirectory (word).

r6 INDXBLK — if creating a file, block number of sequential-file index block or VLIR master index block; if creating a subdirectory, key block number for subdirectory (word).

r7 FILEBYTES — number of bytes in file (word).

r8 HDBLKNUM — block number of the file's header block (word).

r9 FILEHDR — if creating a file, pointer to GEOS file header; if creating a subdirectory, pointer to null-terminated name for the subdirectory (up 15 characters plus a **NULL**); must be in main memory. (word).

Returns:

Commodore:

r6 pointer to first non-reserved block in track/sector table (**BldGDirEntry** reserves one block for the file header and a second block for the index table if the file is a VLIR file).

Alters: **dirEntryBuf** contains newly-built directory entry.

Destroys:

Commodore:
a, x, y, r1H.

Apple:
a, x, y, r3.

Description: Given a GEOS file header, **BldGDirEntry** will build a system specific directory entry suitable for writing to an empty directory slot. **BldGDirEntry** takes care of differences between Apple and Commodore GEOS.

Most applications create new files by calling **SaveFile**. **SaveFile** calls **SetGDirEntry**, which calls **BldGDirEntry** as part of its normal processing.

Apple: The Apple version of **BldGDirEntry** can also be used to build subdirectory entries by passing $0000 in **r2**. The easiest way to create a subdirectory is to call **MakeSubDir**.

Example:

```
;*******************************************************************************
;MySetGDirEntry (My Version of Set Geos Directory Entry)
;
; This routine duplicates the function of the Kernal's SetGDirEntry for
```

```
; demonstration purposes. It shows examples of the following routines:
;
;             BldGDirEntry
;             GetFreeDirBlk
;             UpdateParent
;             PutBlock
;
;PASS:        Same as SetGDirEntry
;DESTROYS:    Same as SetGDirEntry
;***************************************************************************

.if (APPLE)                         ;bytes in directory to copy over
DIRCOPYSIZE    = 39                 ;after BldGDirEntry is called
.else (CBM)                         ;
DIRCOPYSIZE    = 30                 ;
.endif

MySetGDirEntry:
     jsr       BldGDirEntry         ; build directory entry for GEOS file
     jsr       GetFreeDirBlk        ; get block with free directory entry
                                    ; r3 = 1st byte of free entry
                                    ; block number of block in r1
     txa                            ; test for error code
     bne       90$                  ; if error, exit...
.if (APPLE)                         ;
     MoveW     r3,r5                ; make r5 point to directory entry
.else ;(CBM)                        ;
     tya                            ; get offset into diskBlkBuf for dir entry
     clc                            ;
     adc       #[diskBlkBuf         ; and get absolute address in buffer
     sta       r5L                  ;
     lda       #]diskBlkBuf         ;
     adc       #0                   ; (propagate carry)
     sta       r5H                  ;
.endif ;(else CBM)                  ; (r5 = pointer to dir entry in buffer)
     ldy       #DIRCOPYSIZE         ; copy over some bytes
10$:                                ;
     lda       dirEntryBuf,y        ; get byte from directory entry built
     sta       (r5),y               ; store new entry into block buffer
     dey                            ;
     bpl       10$                  ; loop till copied
                                    ;
     jsr       TimeStampEntry       ; stamp the dir entry with time & date
     LoadW     r4,#diskBlkBuf       ; write out the new directory entry
     jsr       PutBlock             ;
     txa                            ; get error status
     bne       90$                  ; if error, exit
                                    ;
.if (APPLE)                         ; Update parent directory on Apple
     PushW     r1                   ; save block number of directory block that
                                    ;      this file entry is stored in
     lda       #(INC_NUM_FILES | MOD_DATE)
     jsr       UpdateParent         ; update parent directory's directory entry
     PopW      r1                   ; restore block number of directory block
                                    ;      that this file entry is saved on
                                    ; (error status now in x)
.endif ;(APPLE)                     ;
90$:                                ;
     rts                            ;


;***************************************************************************
;TimeStampEntry
;     Add time/date stamp to directory entry
```

```
;*****************************************************************************
TimeStampEntry:
.if (C64 || C128)   ;*** COMMODORE ***
TDSIZE        == 5                      ; number of bytes in time/date entry
      ldy     #OFF_YEAR                 ; offset to time/date stamp
10$:                                    ;
      lda     year-OFF_YEAR,y           ; get the yr/mo/day/hr/min
      sta     (r5),y                    ; store in dir entry
      iny                               ;
      cpy     #OFF_YEAR+TDSIZE          ; done yet?
      bne     10$                       ; if not, loop
      rts
.else ;(APPLE)      ;*** APPLE ***
      clc                               ; stamp creation date
      jsr     StampTimeDate             ;
      sec                               ; stamp modification date
;     jsr     StampTimeDate             ; FALL THROUGH to StampTimeDate
;     rts                               ;
StampTimeDate:                          ; (carry indicates stamp to modify)
      lda     minutes                   ; get minutes (0-59)
      ldy     #OFF_FCTIME               ; get offset to creation minutes
      bcs     110$                      ; if stamping creation time, skip
      ldy     #OFF_FMTIME               ; get offset to modification minutes
110$:                                   ;
      sta     (r5),y                    ; save hours in directory entry
      lda     hour                      ; get hours (0-23)
      iny                               ; get offset to hours
      sta     (r5),y                    ; save minutes in directory entry
                                        ;
      ldy     #OFF_FCDATE               ; get offset to mmmddddd
      bcs     120$                      ; if stamping creation time, skip
      ldy     #OFF_FMDATE               ;
120$:                                   ;
      lda     month                     ; get month (1-12)
      asl     a                         ; shift to get mmm00000
      asl     a                         ;
      asl     a                         ;
      asl     a                         ;
      asl     a                         ; carry holds month's msb
      ora     day                       ; combine with day to get mmmddddd
      sta     (r5),y                    ; save mmmddddd
      iny                               ; get offset to yyyyyyym
      lda     year                      ; get year
      rol     a                         ; carry has m's MSB; move it into
                                        ;      LSB to get yyyyyyym
      sta     (r5),y                    ; save yyyyyyym
      rts                               ; exit
.endif ;(.else APPLE)                   ;
```

See also: **SetGDirEntry.**

BlkAlloc (Apple, C64, C128) — mid-level disk

Function: Allocate enough disk blocks to hold a specified number of bytes.

Parameters:

r2 BYTES — number of bytes to allocate space for. Commodore version can allocate up to 32,258 bytes (127 Commodore blocks); Apple version can allocate up to 65,535 ($ffff) bytes (128 ProDOS blocks) (word).

Commodore:

r6 TSTABLE — pointer to buffer for building out track and sector table of allocated blocks, usually points to **fileTrScTab** (word).

Uses:

curDrive currently active disk drive.

Commodore:

curDirHead this buffer must contain the current directory header.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)
interleave† desired physical sector interleave (usually 8); used by **setNextFree**. Applications need not set this explicitly — will be set automatically by internal GEOS routines.

Apple:

VBMBlkno† Apple GEOS **BlkAlloc** starts searching for free blocks beginning with the block that follows the last VBM block.
numVBMBlks† the block following the last VBM block is found by adding the value in **numVBMBlks** to the value in **VBMBlkno**.
curVBlkno† used by VBM cacheing routines.
VBMchanged† used by VBM cacheing routines

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x error ($00 = no error).
r2 number of blocks allocated to hold *BYTES* amount of data.

Commodore:

r3L track of last allocated block.
r3H sector of last allocated block.

Apple:

r3 last block allocated.

Alters:

Commodore:

curDirHead BAM updated to reflect newly allocated blocks.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

Apple:

curVBlkno† used by VBM cacheing routines.
VBMchanged† set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: Commodore:
a, y, r4–r8.

Apple:
a, y, r4, r6, r7, r8H.

Description: **BlkAlloc** allocates enough blocks on the disk for *BYTES* amount of data. The GEOS **SaveFile** and **WriteRecord** routines call **BlkAlloc** to allocate multiple blocks prior to calling **WriteFile**. Most applications do not call **BlkAlloc** directly, but rely, instead, on the higher-level **SaveFile** and **WriteRecord**.

BlkAlloc calculates the number of blocks needed to store *BYTES* amount of data, taking any standard overhead into account (such as the two-byte track/sector link required in each Commodore block), then calls **CalcBlksFree** to ensure that enough free blocks exist on the disk. If there are not enough free blocks to accomodate the data, **BlkAlloc** returns an **INSUFFICIENT_SPACE** error without allocating any blocks. Otherwise, **BlkAlloc** calls **SetNextFree** to allocate the proper number of unused blocks.

C64 & C128: The Commodore version of **BlkAlloc** builds out a track and sector table in the buffer pointed to by *TSTABLE*. The 256 bytes at **fileTrScTab** are usually used for this purpose. When **BlkAlloc** returns, the table contains a two-byte entry for each block that was allocated: the first byte is the track and the second byte is the sector. The last entry in the table has its first byte set to $00, indicating the end of the table. The second byte of the last entry is an index to the last byte in the last block. This track/sector list can be passed directly to **WriteFile** for use in writing data to the blocks.

BlkAlloc does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM. **BlkAlloc** does not allocate blocks on the directory track. Refer to **GetFreeDirBlk** for more information on allocating directory blocks.

Apple: The Apple version of **BlkAlloc** builds out a list of allocated blocks in the internal **INDEXBLOCKBUF** buffer. This index block is a modified 512-byte ProDOS sapling index block. The block is divided into two 256-byte portions. The lower half (bytes 0 through 255) represent the low-bytes of the data blocks and the upper half (bytes 256 through 511) represent the associated high-bytes.Byte 0 and byte 256+0 combine to form the block number of the first data block, byte 1 and byte 256+1 combine to form the number of the second data block, and so on for each allocated block. A block pointer of $0000 ($00 high, $00 low) represents the end of the list.

The last two pointers in the index block—the ones in a standard ProDOS sapling index that point to the 254th and 255th blocks in the chain—comprise a four-byte size value in GEOS. The four-byte value, which represents the number of bytes stored in the allocated blocks, can be accessed using the following offsets into the index block:

DSIZE0	Low byte
DSIZE1	.

DSIZE2
DSIZE3 High byte

BlkAlloc does not patch the size information into **INDEXBLOCKBUF**. The application must, therefore, do this itself before writing the index block to disk. However, because **INDEXBLOCKBUF** is in auxiliary memory, applications cannot patch the index block directly from main memory. **MoveAuxData** can transfer the 512 bytes from auxiliary to a buffer visible from main memory.

```
LoadW    r0, #INDEXBLOCKBUF  ; copy from index blk
LoadW    r1, #diskBlkBuf     ; to temp buffer
LoadW    r2, #BLOCKSIZE      ; move a full block
lda      #AUXtoMAIN          ; copy aux to main
jsr      MoveAuxData         ;
```

The size information can then be patched into the index block, masquerading as the last two block pointers.

```
MoveB    dataSize+0,diskBlkBuf+DSIZE0      ; transfer size bytes
MoveB    dataSize+1,diskBlkBuf+DSIZE1      ; (low to high order)
MoveB    dataSize+2,diskBlkBuf+DSIZE2      ;
MoveB    dataSize+3,diskBlkBuf+DSIZE3      ;
```

The block can then be transferred back to **INDEXBLOCKBUF**.

```
LoadW    r0, #diskBlkBuf     ; copy from buffer
LoadW    r1, #INDEXBLOCKBUF  ; back to index block
LoadW    r2, #BLOCKSIZE      ; move a full block
lda      #MAINtoAUX          ; copy main to aux
jsr      MoveAuxData         ;
```

Use the index block to write out the appropriate data, perhaps by calling **WriteFile**, then use **PutBlock** to write out the index block to a previously allocated block.

```
MoveW    yourIndex,r1        ; where to write block (you suppy)
LoadW    r4, #INDEXBLOCKBUF  ; where data is
LoadB    RWbank, #AUX        ; bank to aux
jsr      PutBlock            ; write out block
LoadB    RWbank, #MAIN       ; back to main
```

BlkAlloc does not flush the VBM cache. See **PutVBM** for more information on flushing the cache.

Note: For more information on the scheme used to allocate successive blocks, refer to **SetNextFree**.

Example:

```
GrabSomeBlocks:
     LoadW    r2,#BUF_SIZE        ; number of bytes to allocate
.if (C64 || C128)
     LoadW    r6,#fileTrSecTab    ; CBM needs buffer to build out table
                                  ; of links. APPLE will automatically use
                                  ; INDEXBLOCKBUF
.endif
     jsr      BlkAlloc            ; allocate the blocks
     txa                          ; check status
     bne      99$                 ; and exit on error
```

```
;--- more code here

99$: rts                            ;exit

.ramsect
K               = 1024              ; one kilobyte
buffer:         .block      5*K     ; 5K buffer
BUF_SIZE        = (*-buffer)+1      ; size of buffer
.psect
```

For a more detailed example, see **WritFile.**

See also: **NxtBlkAlloc, SetNextFree, GetFreeDirBlk, FreeBlock.**

BlockProcess (Apple, C64, C128) process

Function: Block a process's events.

Parameters: **x** PROCESS — process to block (0 to *n*–1, where *n* is the number of processes in the table) (byte).

Returns: **x** unchanged.

Destroys: **a**

Description: **BlockProcess** causes **MainLoop** to ignore the runable flag of a particular process so that if a process timer reaches zero (causing the process to become runable) no process event is generated until the process is subsequently unblocked with a call to **UnblockProcess. BlockProcess** stops the process at the MainLoop level. Refer to **FreezeProcess** to stop the process at the Interrupt Level.

BlockProcess does not stop the countdown timer, which continues to decrement at Interrupt Level (assuming the process is not frozen). When the timer reaches zero, the runable flag is set and the timer is restarted. As long as the process is blocked, though, **MainLoop** ignores this runable flag and, therefore, never generates an event. When a blocked process is later unblocked, **MainLoop** checks the runable flag. If the runable flag was set during the time the process was blocked, **this** pending event generates a call to the appropriate service routine. Only one event is generated when a process is unblocked, even if the timer reached zero more than once.

Note: If a process is already blocked, a redundant call to **BlockProcess** has no effect.

Example:

```
SuspendClock:
     ldx      #CLOCK_PROCESS      ; x <- process number of the clock
     jmp      BlockProcess        ; block that paricular process
```

See also: **UnblockProcess, FreezeProcess.**

BMult (Apple, C64, C128) math

Function: Unsigned word-by-byte multiply: multiplies an unsigned word and an unsigned byte to produce an unsigned word result.

Parameters:

- **x** OPERAND1 — zero-page address of word multiplicand (byte pointer to a word variable).
- **y** OPERAND2 — zero-page address of multiplier (byte pointer to a word variable — use a word variable; only the low-byte is used in the multiplication process, but the high-byte of the word is destroyed).

Note: *result = OPERAND1(word) * OPERAND2(byte).*

Returns: **x, y** unchanged.
word pointed to by *OPERAND2* has its high-byte set to $00, and its low-byte unchanged.
word pointed to by *OPERAND1* contains the word result.

Destroys: **a, r6–r8.**

Description: **BMult** is an unsigned word-by-byte multiplication routine that multiplies the word at one zero-page address by the byte at another to produce a 16-bit word result. **BMult** operates by clearing the the high-byte of *OPERAND2* and calling **DMult**. The result is stored as a word back in *OPERAND1*.

Note: Because **r6, r7** and **r8** are destroyed in the multiplication process, they cannot be used to hold the operands.

Overflow in the result (beyond 16-bits) is ignored.

Example:

```
; Multiply the value in r9 by 87 and store the result back in r9
; (r1 is destroyed)
     ldx      #r9                ; point to OPERAND1 in r9
     LoadB    r1L,#87            ; r1 <- 87 (OPERAND2)
     ldy      #r1                ; point to OPERAND2 in r1
     jsr      BBMult             ; r9 <- r9 * r1L
```

SeeAlso: **BBMult, DMult, Ddiv, DSdiv.**

BootGEOS (C64, C128) internal

Function: Restart GEOS from a non-GEOS application.

Parameters: none.

Returns: Does not return.

Destroys: n/a

Description: **BootGEOS** provides a method for an non-GEOS to run in the GEOS environment—starting up from the deskTop and returning to GEOS when done. The non-GEOS application need only preserve the area of memory between **BootGEOS** ($c000) and **BootGEOS**+$7f ($c07f). The rest of the GEOS Kernal may be overwritten. To reboot GEOS, simply **jmp BootGEOS**, which completely reloads the operating system (either from disk in a "boot" procedure or from a RAM-expansion unit in an "rboot" procedure) and returns to the GEOS **deskTop**.

A program can check to see if it was loaded by GEOS by checking the memory starting at $c006 (**bootName**) for the ASCII (not CBMASCII) string "GEOS BOOT". If loaded by GEOS, the program can check bit 5 of $c012 (**sysFlgCopy**): if this bit is clear, ask the user to insert their GEOS boot disk before continuing, otherwise a boot disk is not needed because GEOS will rboot from the RAM expansion unit. To actually return to GEOS, set **CPU_DATA** to $37 (**KRNL_BAS_IO_IN**) on a Commodore 64 and set **config** to $40 (**CKRNL_BAS_IO_IN**) on a Commodore 128, then jump to **BootGEOS** ($c010).

Example:

```
BYTESTOSAVE     = $80               ; no. of bytes to save at BootGEOS
RBOOT_BIT       = 5                 ; bit in sysFlgCopy to check

OnEntry:
      ldx       #BYTESTOSAVE        ; save bytes GEOS needs so we can use area
10$:                                ; STARTLOOP
      lda       BootGEOS-1,x        ;    copy a byte
      sta       GEOS_save-1,x       ;
      dex                           ;    count--
      bne       10$                 ;    if (count != 0), then loop
      rts                           ; ENDLOOP
.ramsect
GEOS_save       .block BYTESTOSAVE  ; save area for GEOS restart block
.psect

OnExit:
  .if (C64&&C128)
      .echo     Warning: expected both C64 and C128 constants TRUE
  .endif
      lda       version             ; Get version of GEOS
      cmp       #$13                ;
      blt       64$                 ; If version < 1.3, then branch
      lda       c128Flag            ; Else, test for GEOS 128
      bpl       64$                 ; If GEOS64, then branch
128$:                               ;
      lda       CKRNL_BAS_IO_IN     ; load 128 memory mapping
      sta       config              ;
```

```
     bra       200$                ;
64$:                               ;
     lda       #KRNL_BAS_IO_IN     ; load 64 memory mapping
     sta       CPU_DATA            ;
200$:                              ;
     ldx       #BYTESTOSAVE        ; restore bytes GEOS needs to restart
10$:                               ; STARTLOOP
     lda       GEOS_save-1,x       ;    copy a byte
     sta       BootGEOS-1,x        ;
     dex                           ;    count--
     bne       10$                 ;    if (count != 0), then loop
                                   ; ENDLOOP
     lda       #(%1<<RBOOT_BIT)    ;check for Rboot flag
     and       sysFlgCopy          ;
     bne       99$                 ; if flag is clear, branch to rboot
     jsr       AskForBootDisk      ; else, get user to insert boot disk
99$:                               ;
     jmp       BootGEOS            ;
```

See also: **FirstInit, StartAppl, GetFile, EnterDeskTop.**

This page intentionally left blank to maintain right/left (verso/recto) page ordering. Final version will correct this.

CalcBlksFree (Apple, C64, C128) mid-level disk

Function: Calculate total number of free blocks on disk.

Parameters:

Commodore:
r5 DIRHEAD — address of directory header, should always point to **curDirHead** (word).

Apple:
none.

Uses: **curDrive**

Commodore:
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

Apple:
curVBlkno†
VBMchanged†
numVBMBlks†

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **r4** number of free blocks.

Apple:
x error ($00 = no error).

Commodore:
r5 unchanged.
r3 in GEOS v1.3 and later: total number of available blocks on empty disk. This is useful because v1.3 and later support disk devices other than the 1541. GEOS versions earlier than v1.3 leave **r3** unchanged.

Destroys: **a, y.**

Description: **CalcBlksFree** calculates the number of free blocks available on the disk. An application can call **CalcBlksFree**, for example, to tell the user the amount of free space available on a particular disk. GEOS disk routines that allocate multiple blocks at once, such as **BlkAlloc**, call **CalcBlksFree** to ensure enough free space exists on the disk to prevent a surprise **INSUFFICENT_SPACE** error. midway through the allocation. (This is why it is usually not necessary to check for sufficent space before saving a file or a VLIR record—the higher level GEOS disk routines handle this checking automatically.)

C64 & C128: The Commodore version of **CaclBlksFree** looks at the BAM in memory and counts the number of unallocated blocks. The BAM is stored in the directory header and the directory header is stored in the buffer at **curDirHead**. Calling **CalcBlksFree** requires first loading **r5** with the address of **curDirHead**.

```
LoadW    r5, #curDirHead     ; point to the directory header
jsr      CalcBlksFree        ; before calling CalcBlksFree
```

When checking the total number of blocks (both allocated and free) on a particular disk device, call **CalcBlksFree** with **r3** loaded with the number of blocks on a 1541 disk device. On GEOS v1.3 and above, this number is changed to reflect the actual number of blocks in the device. On previous versions of GEOS, **r3** comes back unchanged.

```
N1541_BLOCKS   = 664              ; total number of blocks on 1541 devices
     LoadW     r3, #N1541_BLOCKS  ; assume 1541 block count for v1.2 Kernals
     LoadW     r5, #curDirHead    ; point to the directory header
     jsr       CalcBlksFree       ; r3 comes back with total number of blocks
                                  ; on this device
```

Apple: The Apple version of **CalcBlksFree** looks through the VBM blocks on the disk (larger capacity disks may have more than one) and determines the number of unallocated blocks by counting the allocation bits.

Example:

```
;*******************************************************************************
;CheckDiskSpace
;
;DESCRIPTION:  Ensures that the current disk has a enough space for a
;              minimum number of bytes. Does not take into account any
;              index blocks or other blocks needed to maintain the file
;              structure. Works with GEOS 64, GEOS 128, and Apple GEOS.
;                    •
;PASSED:       r2    number of bytes we need
;
;RETURNS:      x =   error if any. If not enough space, returns an
;                    INSUFFICIENT_SPACE error.
;
;DESTROYED:    a, y, r2, r3, r8, r9
;*******************************************************************************

; Number of bytes that can be stored in each block on the disk. Accounts for
; two-byte track/sector link on Commodore versions of GEOS.
BLOCK_BYTES    = BLOCK_SIZE-((C64||128)*2)

CheckDiskSpace:                   ; r2 = # of BYTES to check for
     lda       r2L                ; check if zero bytes requested
     ora       r2H                ;
     beq       80$                ; if so, exit with no error
     LoadW     r3,#BLOCK_BYTES    ; r3 <- number of bytes per block.
     ldx       #r2                ; divide r2 by r3 to get number of
     ldy       #r3                ; blocks to hold BYTES
     jsr       Ddiv               ; r2 <- r3/r2 ; r8L <- remainder
     lda       r8L                ; Any remainder bytes?
     ora       r8H                ;
     beq       5$                 ; if not, OK
     IncW      r2                 ; otherwise 1 more block needed
5$:                               ; r2 = BLOCKS needed to hold BYTES.
                                  ; get number of free blocks on disk
.if (C64||C128)                   ; CBM GEOS requires extra step
     LoadW     r5,#curDirHead     ;   point to directory header
.endif                            ;
     jsr       CalcBlksFree       ; r4 <- free blocks on disk
.if (APPLE)                       ; APPLE GEOS must error check here
     txa                          ;   check status
     bne       100$               ;   branch to exit with error...
.endif                            ;
     ldx       #INSUFFICIENT_SPACE ; assume not enough space
     CmpW      r2,r4              ; are there enough free blocks?
```

```
        bgt     100$                ; if not, asssump. correct, branch...
80$:                                ;
        ldx     #NO_ERROR           ; otherwise, no error
100$:                               ;
        rts                         ; exit
```

See also: **BlkAlloc, SetNextFree, FreeBlock.**

CallRoutine (Apple, C64, C128) utility

Function: Perform a pseudo-subroutine call, checking first for a null address (which will be ignored).

Parameters:

a	[ADDRESS — low byte of subroutine to call.
x	]ADDRESS — high byte of subroutine to call.

where ADDRESS is the address of a subroutine to call.

Returns: depends on subroutine at *ADDRESS*.

Destroys: depends on subroutine at *ADDRESS*.

Description: **CallRoutine** offers a clean and simple way to perform an indirect **jsr** through a vector or call a subroutine with an address from a jump table. Before simulatin a **jsr** to the address in the **x** and **a** registers, it also checks for a null address ($0000). If the address is $0000 (x=$00 and a=$00), **CallRoutine** performs **rts** without calling any subroutine address. This makes it easy to nullify a vector or an entry in a jump table by using a $0000 value.

GEOS frequently uses **CallRoutine** when calling through vectors. This is why placing a $0000 into **keyVector**, for example, causes GEOS ignore the vector. Other examples of this usage are **intTopVec**, **intBotVec**, and **mouseVector**.

Note: **CallRoutine** modifies the **st** register prior to performing the **jsr**. It, therefore, cannot be used to call routines that expect processor status flags as parameters (flags may be *returned* in the **st** register, however). **CallRoutine** may be called from Interrupt Level (off of routines in **IntTopVec** and **IntBotVec**). Do not use **CallRoutine** to call inline (**i_**) routines, as it will not return properly.

Example:

```
;*******************************************************************************
;HandleCommand
;
;DESCRIPTION:  Given a command number this routine handles dispatching
;              control to the appropriate routine.
;
;PASSED:       y     command number
;
;RETURNS:      depends on command
;
;DESTROYED:    depends on command
;*******************************************************************************

HandleCommand:
      cpy      #TOT_CMDS-1          ; check command # against last cmd num
      bgt      99$                  ; exit if command is invalid
      ldx      CMDtabH,y            ; get high byte routine address
      lda      CMDtabL,y            ; get low byte of routine address
      jsr      CallRoutine          ; call the routine
99$:                                ;
      rts                          ; exit

; The table below is a collection of the the high/low bytes of the routine
; associated with each command number. If a command is not yet implemented,
```

```
; use the UNIMPLEMENTED constant ($0000) to have it ignored.
UNIMPLEMENTED = $0000              ; CallRoutine ignores this

CMDtabH:      ;high bytes
      .byte   ]UNIMPLEMENTED       ; command 0
      .byte   ]Cmd1                ; command 1
      .byte   ]Cmd2                ; command 2
      .byte   ]Cmd3                ; command 3
      .byte   ]Cmd4                ; command 4
      .byte   ]Cmd5                ; command 5

CMDtabL:      ;low bytes
      .byte   [UNIMPLEMENTED       ; command 0
      .byte   [Cmd1                ; command 1
      .byte   [Cmd2                ; command 2
      .byte   [Cmd3                ; command 3
      .byte   [Cmd4                ; command 4
      .byte   [Cmd5                ; command 5

TOT_CMDS      = (CMDtabL-CMDtabH) ;total number of commands
```

See also: **JsrToAux**

CancelPrint (Apple) — printer driver

Function: Cancel printing immediately, clearing any buffers on the printer card or printer.

Parameters: none.

Returns: **x** STATUS — printer error code; $00 = no error.

Destroys: assume **a, y, r0–r4.**

Description: **CancelPrint** instructs the printer driver to cancel the current printing operation, clearing any data already accumulated in a buffer on the printer or the printer card. Its main use is to allow the application to have the printer *really* stop printing if the user decides to cancel.

Note: The **CancelPrint** routine does not work properly in the first and second versions of most Apple printer drivers.

See also: **StopPrint.**

ChangeDiskDevice (C64, C128) — very low-level disk

Function: Instruct a drive to change its serial device number.

Parameters: **a** NEWDEVNUM — new device number to give current drive (byte).

Uses: **curDrive** drive whose device number will change.

Returns: **x** error ($00 = no error).

Alters: **curDrive** *NEWDEVNUM.*
curDevice *NEWDEVNUM.*

Destroys: **a, y.**

Description: **ChangeDiskDevice** requests the turbo software to change the serial device number of the current drive. Most applications have no need to call this routine, as it is in the realm of low-level disk utilities. **ChangeDiskDevice** is used primarily by the deskTop and Configure programs to add, rearrange, and remove drives.

Be aware that changing the device number merely instructs the turbo software in the drive to monitor a different serial bus address. Many internal GEOS variables and disk drivers expect the original device number to remain unchanged.

Note: If **ChangeDiskDevice** is used on a RAMdisk, **curDrive** and **curDevice** both change. However, because of the nature of the RAMdisk driver, the RAMdisk does not respond as this new device.

Apple: Apple GEOS has no **ChangeDiskDevice** equivalent.

See also: **SetDevice.**

ChkDkGEOS (C64, C128) mid-level disk

Function: Check Commodore disk for GEOS format.

Parameters: **r5** DIRHEAD — address of directory header, should always point to **curDirHead** (word).

Returns: nothing

Alters: **isGEOS** set to **TRUE** if disk is a GEOS disk, otherwise set to **FALSE.**
st set according to valu in **isGEOS.**

Destroys: **a, y.**

Description: **ChkDkGEOS** checks the directory header for the version string that flags it as a GEOS disk (at **OFF_GEOS_ID**). The primary difference between a GEOS disk and a standard Commodore disk is the addition of the off-page directory and the possiblity of GEOS files on the disk. GEOS files have an additional file header block that holds the icon image and other information, such as the author name and permanent name string. To convert a non-GEOS disk into a GEOS disk, use **SetGEOSDisk.**

OpenDisk automatically calls **ChkDkGEOS.** As long as **OpenDisk** is used before reading a new disk, applications should have no need to call **ChkDkGEOS.**

Apple: All ProDOS disks are GEOS compatible.

Example:

```
.if (C64||C128)
     jsr      GetDirHead          ; read in the directory header
     txa                          ; check status
     bne      99$                 ; exit on error
     LoadW    r5,#curDirHead      ; point to directory header
     jsr      ChkDkGEOS           ; Check for GEOS disk
     beq      40$                 ; if not a GEOS disk, branch
;--- code here to handle GEOS disk
     bra      99$                 ; jump to exit
50$:
;--- code here to handle non-GEOS disk

99$: rts                          ; exit
```

See also: **SetGEOSDisk.**

ClearCard (Apple) — card driver

Function: Instructs the interface card to stop transmitting data and clear its internal buffer if it has such a feature.

Parameters: none.

Returns: x STATUS — card error code; $00 = no error (byte)

Destroys: a, y.

Description: **ClearCard** instructs the interface card to immediately stop sending data and clear its internal buffer if it has such a feature. **ClearCard** is designed to be called by **CancelPrint** to abort any buffered print data, but, because this feature is rarely supported by interface cards, most printer drivers omit this call to save space.

Note: **ClearCard** must be called after an **OpenCard** and before a **CloseCard**.

See also: **CancelPrint**.

ClearMouseMode (Apple, C64, C128) mouse/sprite

Function: Stop monitoring the input device.

Parameters: nothing.

Returns: nothing.

Alters:

mouseOn	set to $00, totally disabling all mouse tracking
mobenble	sprite #0 bit cleared by **DisablSprite.**

Destroys: **a, x, y, r3L**

Description: **ClearMouseMode** instructs GEOS to totally disable its monitoring of the input device. It clears **mouseOn** to reset mouse tracking to its cleared state and calls **DisablSprite.** Applications will normally not have a need to call this routine. It is the functional opposite of **StartMouseMode.**

See also: **StartMouseMode, MouseOff.**

ClearRam (Apple, C64, C128) memory

Function: Clear a region of memory to $00.

Parameters:
r1 ADDR —address of area to clear (word).
r0 COUNT — number of bytes to clear (0 - 64K) (word).

Returns: nothing.

Destroys: **a, y, r0, r1, r2L**

Description: **ClearRam** clears COUNT bytes starting at ADDR to $00. It useful for initializing ramsect variable and data sections.

Note: Do not use **ClearRam** to initalize **r0** through **r2L**. Also, for when space is at a premium, it actually takes fewer bytes to call **i_FillRam** with a fill value of $00.

Example:

```
; initialize buffers and variables to zero
     LoadW    r0,#varStart          ; clear variable space
     LoadW    r1,#(varEnd-varStart) ;
     jsr      ClearRam              ;
     LoadW    r0,#heapStart         ; clear heap
     LoadW    r1,#(heapEnd-heapStart) ;
     jsr      ClearRam              ;
```

See also: **FillRam, InitRam.**

ClockInt (Apple) — clock driver

Function: Clock driver Interrupt Level routine. Called by GEOS during Interrupt Level.

Parameters: none.

Alters: (usually)
year
month
day
hour
minutes
seconds

Destroys: assume **a, x, y, r0–r15.**

Description: GEOS calls **ClockInt** during Interrupt Level, which allows a clock driver to have an Interrupt Level routine.This facility is used primarily by the software clock, which expects to be called approximately **FRAME_RATE** times per second. Every time **ClockInt** gets called, the software clock decrements a counter that is initialized to **FRAME_RATE**. When the counter reaches zero, approximately one second has elapsed.

Hardware clock drivers usually have a **ClockInt** routine that simply performs an **rts.**

Example:

```
.if (APPLE)    ; clock drivers only exist in Apple GEOS
;*** Sample ClockInt clock driver routine ***
;************************************************************************
;o_ClockInt
;
;Synopsis:     This is called at interrupt level for the software clock.
;
;Author:       Andrew Wilson       January 1988
;Called by:    Interrupt Level through clock driver jump table
;Pass:         nothing
;Returns:      nothing
;Alters:       assume a, x, y, r0-r15
;************************************************************************
o_ReadClockInt:
     lda      hour              ; if nothing set, exit
     ora      minutes           ;
     ora      seconds           ;
     ora      year              ;
     ora      month             ;
     ora      day               ;
     beq      90$               ;
     dec      counter           ; have a second's worth of int's occurred?
     bpl      90$               ; branch if it hasn't happened

; this code happens once a second, assuming interrupts weren't shut off

     ldx      #FRAME_RATE-1     ; counter <- (interrupts/second)-1
     stx      counter           ;
     ldx      #60-1             ; x <- seconds/minute (and min/hr)
                                ; (actually, one less so we can use bge
                                ;  branch, which is smaller)
     inc      seconds           ; add one second to clock
```

```
        cpx     seconds             ; do we add another minute?
        bge     90$                 ; branch if seconds don't roll

; this code happens roughly once a minute, assuming interrupts weren't off

        lda     #0                  ; reset seconds
        sta     seconds             ; save the new seconds
        inc     minutes             ; add one to minutes
        cpx     minutes             ; do we add another hour?
        bge     90$                 ; branch if minutes don't roll

; this code happens once an hour, assuming interrupts weren't off

        sta     minutes             ; reset minutes
        inc     hour                ; add another hour to the software clock
        ldx     #24-1               ; x <- (hours/day)-1
        cpx     hour                ; do we add another day
        bge     90$                 ; branch if hours don't roll

; this code happens once a day, assuming interrupts are off

        sta     hour                ; zero hours
        inc     day                 ; add another day
        ldy     month               ; get the current month
        lda     #28+1               ; febMonth <- days in February+1
        sta     febMonth            ;
        lda     year                ; check for leap year
        and     #%00000011          ; (nice and convenient that 4
        bne     50$                 ;   is a power of 2)
        inc     febMonth            ; if leap year, then one more day in Feb.
50$:
        lda     daysInMonth-1,y     ; see if the current day ends the month
        cmp     day                 ; compare
        bne     90$                 ; branch if month doesn't roll

; this code happens once a month, assuming the computer lasts that long...

        ldx     #1                  ; first day of month
        stx     day                 ; save it
        inc     month               ; add one month
        lda     month               ;
        cmp     #12                 ; see if we're past the last month
        blt     90$                 ;

; this code happens once a year. It rolls over in 1999, and so we'll need
; a new clock driver before then...

        lda     #1                  ; first month of the year
        sta     month               ;
        inc     year                ; new year
90$:                                ;
        rts                         ; exit

; daysInMonth is the number of days in each month. It actually contains
; number of days plus one since we increment days before we check.

        daysInMonth:    .byte   32
        febMonth:       .byte   29      ; February is modified on leap years
                        .byte   32,31,32,31,31,32,31,32,31,32
.endif ;(APPLE)
```

See also: **ReadClock, AuxDInt.**

CloseCard (Apple) — card driver

Function: Close access to interface card.

Parameters: none.

Returns: x STATUS — card error code; $00 = no error (byte)

Destroys: a, y.

Description: **CloseCard** closes access to the interface card. It is the functional opposite of **OpenCard**. The printer driver sends blocks of data to the printer between calls to **OpenCard** and **CloseCard**. The **CloseCard** routine in most card drivers simply returns without an error, but printer drivers should always bracket their access to the card driver between calls to **OpenCard** and **CloseCard**.

Example:

```
.if (APPLE)    ; card drivers only exist on Apple GEOS
; Initialize the card

     jsr      InitCard     ; call card driver init routine
     txa                   ; check status
     bne      90$          ; branch if error

; Open the card so we access the printer

     jsr      OpenCard     ; call card driver open routine
     txa                   ; check status
     bne      90$          ; branch if error

; Initialize printer

     ldy      #PRINITCODE  ; y <- control code to init printer
     jsr      OutputByte   ; send through card to printer
     txa                   ; check status
     bne      90$          ; branch if error

; Close the card and exit

     jsr      CloseCard    ; call card driver close routine
90$:
     rts                   ; exit with error in x
.endif ;(APPLE)
```

See also: **OpenCard.**

CloseRecordFile (Apple, C64, C128) VLIR disk

Function: Close the current VLIR file (updating it in the process) so that another may be opened.

Parameters: none.

Uses:

curDrive	
fileWritten†	if **FALSE**, no updating occurs because file has not been written to.
fileHeader	VLIR index table stored in this buffer.
fileSize	total number of disk blocks used in file (includes index block, GEOS file header, and all records).
dirEntryBuf	directory entry of VLIR file.
year, month, day, hours, minutes	for date-stamping file.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	this buffer must contain the current directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

fileBytes	total number of bytes in file (written to bytes 254, 255, 511, and 512 of the master index block).
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:

fileWritten†	set to **FALSE** to indicate that file hasn't been altered since last updated.
diskBlkBuf	used for temporary storage of the directory block.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4, r5.**

Description: **CloseRecordFile** first calls **UpdateRecordFile** then closes the VLIR file so that another may be opened.

C64 & C128: Because Commodore GEOS stores the BAM in global memory, the application must be careful not to corrupt it before the VLIR file is updated or closed. For more information, refer to **UpdateRecordFile.**

Example: See example at **AppendRecord.**

See also: **OpenRecordFile, UpdateRecordFile.**

CmpFString (Apple, C64, C128) string

Function: Compare two fixed-length strings.

Parameters:

- **x** SOURCE — zero-page address of pointer to source string (byte pointer to a word pointer).
- **y** DEST — zero-page address of pointer to destination string (byte pointer to a word pointer).
- **a** LEN — length of strings (1-255). A LEN value of $00 will cause **CmpFString** to function exactly like **CmpString**, expecting a null-terminated source string.

Returns: **st** status register flags reflect the result of the comparison.

Destroys: **a, x, y**

Description: **CmpFString** compares the fixed-length string pointed to by *SOURCE* to the string of the same length pointed to by *DEST*.

CmpFString with a *LEN* value of $00 causes the routine to act exactly like **CmpString**.

CmpFString compares each character in the strings until there is a non-matching pair. The result of the comparison between the non-matching pair is passed back in the processor status register (**st**). If the strings match, the **z** flag is set. This allows the application to test the result of a string comparison with standard test and branch operations:

bne	branch if strings don't match
beq	branch if strings match
bcs	branch if source string is greater than or equal to dest string
bcc	branch if source string is less than dest string

Note: The strings may contain internal **NULL**'s. These will not terminate the comparison.

Example:

```
REC_SIZE == 16                    ;size of each record

Find:
     LoadW    r2,#NUM_RECS        ; r2 <- total number of records
     LoadW    r0,#Key             ; r0 <- pointer to keyword
     LoadW    r1,#Data            ; r1 <- pointer to start of search list
10$:                              ; DO
     ldx      #r0                 ;   x <- source string = key
     ldy      #r1                 ;   y <- dest string = list
     lda      #REC_SIZE           ;   a <- length of each record
     jsr      CmpFString          ;   compare key with current record
     beq      20$                 ;   if they match, branch
     AddVW    #REC_SIZE,r1        ;   otherwise point to the next record
     DecW     r2                  ;   r2-- (decrement counter)
     bne      10$                 ; WHILE (r2 > 0)
; --- code to handle no matches goes here
     rts
```

```
; --- code to handle a match goes here
     rts

Key: .byte    "alpha           "

Data:
     .byte    "ZETA            "
     .byte    "0123456789ABcDeF"
     .byte    "gAMma over beta "
     .byte    "stewardesses    "
     .byte    "123ABC123abc    "
     .byte    "abcdefghijklmnop"
     .byte    "qrstuvwxyz012345"
     .byte    "stewardess      "
     .byte    "beta            "
     .byte    "alpha           "
     .byte    "delta           "
     .byte    "steward         "
EndData:

NUM_RECS      = ((EndData-Data)/REC_SIZE)
.if   ( (Data + NUM_RECS*REC_SIZE) != *)
     .echo    Something is wrong with sort data
.endif
```

See also: **CmpString, CopyFString.**

CmpString (Apple, C64, C128) string

Function: Compares two null-terminated strings.

Parameters:

x	SOURCE — zero-page address of pointer to source string (byte pointer to a word pointer).
y	DEST — zero-page address of pointer to destination string (byte pointer to a word pointer).

Returns: **st** status register flags reflect the result of the comparison.

Destroys: **a, x, y**

Description: **CmpString** compares the null-terminated source string pointed to by *SOURCE* to the destination string pointed to by *DEST*. The strings are compared a byte at a time until either a mismatch is found or a null is encountered in both strings. **CmpString** expects the strings to be the same length; strings of different lengths are treated as not matching.

CmpString compares each character in the strings until there is a non-matching pair. The result of the comparison between the non-matching pair is passed back in the processor status register (**st**). If the strings match, the **z** flag is set. This allows the application to test the result of string comparison with standard test and branch operations:

bne	branch if strings don't match
beq	branch if strings match
bcs	branch if source string is greater than or equal to dest string
bcc	branch if source string is less than dest string

Note: **CmpString** cannot compare strings longer than 256 bytes (including the null). The compare process is aborted after 256 bytes.

Example:

```
Find2:
     LoadW    r0,#Original        ; r0 <- pointer to original string
     LoadW    r1,#Copy            ; r1 <- pointer to copy
     ldx      #r0                 ; x <- source string = key
     ldy      #r1                 ; y <- dest string = list
     jsr      CmpString           ; compare the strings
     beq      20$                 ; if they match, branch
; --- code to handle no matches goes here
     rts
; --- code to handle a match goes here
     rts

Original:
     .byte    "Mark Charles Heartless",NULL
Copy:
     .byte    "Mark Charlie Heartless",NULL
```

See also: **CmpFString, CopyString.**

CopyFString (Apple, C64, C128) string

Function: Copy a fixed-length string.

Parameters:

- **x** SOURCE — zero-page address of pointer to source string (byte pointer to a word pointer).
- **y** DEST — zero-page address of pointer to destination buffer (byte pointer to a word pointer).
- **a** LEN — length of string (1-255). A *LEN* value of $00 will cause **CopyFString** to function exactly like **CopyString**, expecting a null-terminated source string.

Returns: Buffer pointed to by *DEST* contains copy of source string.

Destroys: **a, x, y**

Description: **CopyFString** copies a fixed-length string pointed to by *SOURCE* to the buffer pointed to by *DEST*. If the source and destination areas overlap, the source must be lower in memory for the copy to work properly.

Because the *LEN* parameter is a one-byte value, **CopyFString** cannot copy a string longer than 255 bytes. A *LEN* value of $00 causes **CopyFString** to act exactly like **CopyString**.

Note: The source string may contain internal NULL's. These will not terminate the copy operation.

Example:

```
        LoadW   r5,#SrcString       ; point to start of source string
        LoadW   r11,#DestString     ; point to start of destination string
        ldx     #r5                 ; x <- source register address
        ldy     #r11                ; y <- dest register address
        lda     #LENSTRING          ; a <- length of string
        jsr     CopyFString         ; DestString <- SrcString (copy)

SrcString:  .byte "This is a string",CR,LF
LENSTRING   = (*-SrcString)
.ramsect
DestString  .block LENSTRING
.psect
```

See also: **CopyString, CmpFString, MoveData.**

CopyFullScreen (Apple) graphics

Function: Copy a rectangle of the full width of the screen to another position vertically. Operates only on the foreground screen.

Parameters:
r0L SRCY1 — top line of source region (byte).
r0H SRCY2 — bottom line of source region (byte).
r3L DESTY — top line of destination (byte).

where (0,SRCY1) defines the upper-left corner of the source rectangle, (SC_PIX_WIDTH-1,SRCY2) defines the lower-right corner of the source rectangle, and (0,DESTY) defines the upper-left corner of the destination.

Destroys: **a, x, y, r0, r3L**

Description: **CopyFullScreen** is optimized to quickly copy a rectangular area of the full width of the screen. It copies the desired lines from the source to the destination (which may overlap if necessary). The copy only occurs on the foreground screen.

Note: No clipping at the screen top or bottom is performed; the values passed are assumed to lie entirely within the screen boundaries.

Example:

```
.if (APPLE)
SCROLL_TOP     = 0                          ; top line of scroll region
SCROLL_BOT     = SC_PIX_HEIGHT-1            ; bottom line of scroll region
SCROLL_INC     = 2                          ; lines to scroll

ScrollUp:
      LoadB    r0L,#SCROLL_TOP+SCROLL_INC ; r0L <- top of source
      LoadB    r0H,#SCROLL_BOT            ; r0H <- bottom of source
      LoadB    r3L,#SCROLL_TOP            ; r3L <- destination
      jmp CopyFullScreen                  ; scroll, let CopyFullScreen rts

ScrollDown:
      LoadB    r0L,#SCROLL_TOP            ; r0L <- top of source
      LoadB    r0H,#SCROLL_BOT-SCROLL_INC ; r0H <- bottom of source
      LoadB    r3L,#SCROLL_TOP+SCROLL_INC ; r3L <- destination
      jmp CopyFullScreen                  ; scroll, let CopyFullScreen rts
.endif ;(APPLE)
```

See also: **CopyLine, CopyScreenBlock**

CopyLine (Apple) graphics

Function: Copy a horizontal line to another pixel location; operates on pixel boundaries.

Parameters:

r1	SRCX1 — x-coordinate of left edge of source line (word).
r2	SRCX2 — x-coordinate of right edge of source line (word).
r0L	SRCY — y-coordinate of source line (byte).
r4	DESTX — x-coordinate of left edge of destination (word).
r3L	DESTY — y-coordinate of destination (byte).

where (SRCX1,SRCY) and (SCRC2,SRCY) define the endpoints of the line to move to (DESTX,DESTY).

Uses: **dispBufferOn:**

bit 7 — copy from foreground screen if set.
bit 6 — copy from background buffer if set.
If both bits are set, copy occurs on both screens.

Destroys: **a, x, y**

Description: **CopyLine** copies pixels from a horizontal source line to a destination line. The source and destination coordinates may lie on any pixel boundary. Pixels will be shifted if necessary. The source and destination may overlap.

Note: No clipping at the screen edge is performed; the values passed are assumed to lie entirely within the screen boundaries. Also, the copying is appreciably faster if the left-edge x-coordinates of the source and destination lie on the same bit-boundary. That is, if (*SRCX1 //7*) == (*DESTX // 7*).

Example:

See also: **CopyFullScreen, CopyScreenBlock**

CopyScreenBlock (Apple) graphics

Function: Copy a rectangular portion of the screen to another area.

Parameters:

r1	SRCX1 — x-coordinate of upper-left of source (word).
r0L	SRCY1 — y-coordinate of upper-left of source (byte)
r2	SRCX2 — x-coordinate of lower-right of source line (word).
r0H	SRCY2 — y-coordinate of lower-right of source (byte).
r4	DESTX — x-coordinate of upper-left of destination (word).
r3L	DESTY — y-coordinate of upper-left of destination (byte).

where (SRCX1,SRCY1) defines the upper-left corner of the source rectangle, (SCRCX2,SRCY2) defines the lower-right corner of the source rectangle and (DESTX,DESTY) defines the upper-left corner of the destination.

Uses: **dispBufferOn:**

bit 7 — copy from foreground screen if set.
bit 6 — copy from background buffer if set.
If both bits are set, copy occurs on both screens.

Destroys: **a, x, y**

Description: **CopyScreenBlock** copies pixels from a source rectangle to a destination rectangle. The source and destination coordinates may lie on any pixel boundary. Pixels will be shifted if necessary. The source and destination rectangles may overlap.

Note: No clipping at the screen edge is performed; the values passed are assumed to lie entirely within the screen boundaries. Also, the copying is appreciably faster if the left-edge x-coordinates of the source and destination lie on the same internal bit-boundary. That is, if (SRCX1%7) == (DESTX%7).

Example:

```
.if (APPLE)
;*** Window Coordinate Constants ***
WIN_X          = 20                      ; pixel x-position (left edge)
WIN_WIDTH      = 85                      ; pixel width
WIN_Y          = 40                      ; y-position
WIN_HEIGHT     = 110                     ; height
WIN_X2         = WIN_X+WIN_WIDTH         ; right edge x-position
WIN_Y2         = WIN_Y+WIN_HIEGHT        ; bottom edge y-position

SCROLL_INC     = 2                       ; lines to scroll

ScrWinUp:      ; scroll window up
     jsr       SetScrXPositions          ; set the x-coordinates for scroll
                                         ;    r1 <- left edge source
                                         ;    r2 <- right edge source
                                         ;    r4 <- left edge dest (== r1)
     LoadB     r0L,#WIN_Y+SCROLL_INC     ; r0L <- top of source
     LoadB     r0H,#WIN_Y2               ; r0H <- bottom of source
     LoadB     r3L,#WIN_Y                ; r3L <- top edge of dest
     jmp       CopyScreenBlock           ; scroll, let CopyScreenBlock rts

ScrWinDown:    ; scroll window down
     jsr       SetScrXPositions          ; set the x-coordinates for scroll
```

```
                                        ;     r1 <- left edge source
                                        ;     r2 <- right edge source
                                        ;     r4 <- left edge dest (== r1)
        LoadB   r0L,#WIN_Y              ; r0L <- top of source
        LoadB   r0H,#WIN_Y2-SCROLL_INC  ; r0H <- bottom of source
        LoadB   r3L,#WIN_Y+SCROLL_INC   ; r3L <- top edge of dest
        jmp     CopyScreenBlock         ; scroll, let CopyScreenBlock rts

SetScrXPositions:   ; Set the common dimensions for up or down scroll
;--- Following lines changed to save bytes ---
;       LoadW   r1,#WIN_X               ; r1 <- left edge of source
;       LoadW   r4,#WIN_X               ; r4 <- left edge of dest
        lda     #[WIN_X                 ;   get low byte of left edge
        sta     r1L                     ;   and store it in low byte of
        sta     r4L                     ;   source and dest left edge
        lda     #]WIN_X                 ;   do the same with the hi byte
        sta     r1H                     ;
        sta     r4H                     ;
        LoadW   r2,#WIN_X2              ; r2 <- right edge of dest
        rts                             ; exit
.endif ;(APPLE)
```

Dabs (Apple, C64, C128) math

Function: Compute absolute value of a two's-complement signed word.

Parameters: **x** OPERAND — zero-page address of word to operate on (byte pointer to a word variable).

Returns: **x,y** unchanged
word pointed to by *OPERAND* contains the absolute value result.

Destroys: **a**

Description: **Dabs** takes a signed word at a zero-page address and returnes its absolute value. The address of the word (*OPERAND*) is passed in **x**. The absolute value of *OPERAND* is returned in *OPERAND*.

The equation involved is: **if (value < 0) then value = –value.**

Example:

SeeAlso: **DNegate.**

Ddec (Apple, C64, C128) math

Function: Decrement a word.

Parameters: **x** OPERAND — zero-page address of word to decrement (byte pointer to a word variable).

Returns: **x,y** unchanged
st z flag is set if resulting word is $0000.
zero page word pointed to by *OPERAND* contains the decremented word.

Destroys: **a**

Description: **Ddec** is a double-precision routine that decrements a 16-bit zero-page word (low/high order). The absolute address of the word is passed in **x**. If the result of the decrement is zero, then the the **z** flag in the status register is set and can be tested with a subsequent **beq** or **bne**. **Ddec** is useful for loops which require a two-byte counter.

Example:

```
COUNT       = $fff0      ;counter value
counter     = a0         ;pseudoreg to count with
      lda   #[COUNT      ;init pseudoregister
      sta   counter      ;with counter value
      lda   #]COUNT
      sta   counter+1
      ldx   #counter     ;tell Ddec which zp reg to use
10$:
      nop                ;do nothing in loop
      jsr   Ddec         ;decrement counter (x = reg addr.)
      bne   10$          ;loop until done.
```

Ddiv (Apple, C64, C128) math

Function: Unsigned word-by-word (double-precision) division: divides one unsigned word by another to produce an unsigned word result.

Parameters:
- **x** OPERAND1 — zero-page address of word dividend (byte pointer to a word variable).
- **y** OPERAND2 — zero-page address of word divisor (byte pointer to a word variable).

Note: *result = OPERAND1(word) / OPERAND2(word).*

Returns: **x**, **y**, and word pointed to by *OPERAND2* unchanged.
word pointed to by *OPERAND1* contains the result.
r8 contains the fractional remainder (word).

Destroys: **a**, **r9**

Description: **Ddiv** is an unsigned word-by-word division routine that divides the word at one zero-page address (the dividend) by the word at another (the divisor) to produce a 16-bit word result and a 16-bit word fractional remainder The word in *OPERAND1* is divided by the word in *OPERAND2* and the result is stored as a word back in *OPERAND1*. The remainder is returned in **r8**.

Note: Because **r8** and **r9** are used in the division process, they cannot be used to hold operands.

If the divisor (*OPERAND2*) is greater than the dividend (*OPERAND1*), then the fractional result will be returned as $0000 and *OPERAND1* will be returned in **r8**.

Although dividing by zero is an undefined mathematical operation, **Ddiv** makes no attempt to flag this as an error condition and will simply return incorrect results. If the divisor might be zero, the application should check for this situation before dividing as in:

```
     lda     zpage,y          ;get low byte of divisor
     ora     zpage+1,y        ;get high byte of divisor
     bne     10$              ;if either non-zero, go divide
     jmp     DivideByZero     ;else, flag error
10$:
     jmp     Ddiv             ;divide (Ddiv will rts)
```

There is no possibility of overflow (a result which cannot fit in 16 bits).

Example:

SeeAlso: **DSdiv**, **DMult**, **BMult**, **BBMult**, **DivideBySeven**.

DeleteDir (Apple) — high-level disk

Function: Delete a subdirectory from the current directory by freeing its directory blocks and removing its directory entry. The subdirectory must not contain any files to be deleted.

Parameters: **r0** DIRNAME — pointer to null-terminated directory name (must be in main memory.) (word).

Uses:

curDrive	
curKBlkno	current directory.
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:

diskBlkBuf	used for temporary block storage.
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **FALSE** by VBM cacheing routines to indicate cached VBM block has already been flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4–r7, r8H.**

Description: Given the null-terminated name of an empty subdirectory, **DeleteDir** will remove the subdirectory from the disk by deleting its directory entry and calling **FreeDir** to free its chain of directory blocks.

DeleteDir first calls **FindFile** to get the directory entry of the subdirectory. If the subdirectory does not exist in the current directory, a **FILE_NOT_FOUND** error is returned. A subdirectory is recognized by a **PRO_DIR** in the GEOS type byte in the directory entry (at **OFF_FTYPE**).

DeleteDir will not delete a subdirectory that has files in it. A **DIR_NOT_EMPTY** error will be returned. (The number of files in the subdirectory is stored in the **OFFB_FLCNT** word of the directory's header. This maintains this value by calling **UpdateParent** when files are added or removed.)

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **DeleteDir, DeleteFile, FreeFile, FreeBlock.**

DeleteFile (Apple, C64, C128) — high-level disk

Function: Delete a GEOS file by deleting the its directory entry and freeing *all* its blocks. Works on both sequential and VLIR files.

Parameters:

r0	FILENAME — pointer to null-terminated name of file to delete (Apple GEOS: must be in main memory.) (word).

Uses: **curDrive**

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).

Alters:

diskBlkBuf	used for temporary block storage.
dirEntryBuf	deleted directory entry.

Commodore:

curDirHead	BAM updated to reflect newly freed blocks.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
fileHeader	temporary storage of index table when deleting a VLIR file.

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **FALSE** by VBM cacheing routines to indicate cached VBM block has already been flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Commodore:
a, y, r0–r9.

Apple:
a, y, r1, r2L, r4–r7, r8H, r9.

Description: Given a null-terminated filename, **DeleteFile** will remove it from the current directory by deleting its directory entry and calling **FreeFile** to free all the blocks in the file.

DeleteFile first calls **FindFile** to get the directory entry and ensure the file does in fact exist. If the file specified with *FILENAME* is not found, a **FILE_NOT_FOUND** error is returned.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk. The directory entry is deleted by setting its **OFF_CFILE_TYPE** byte to $00.

Apple: Will only delete a file from the current directory. To delete a file from another directory, the application must change directories (refer to **GoDirectory** for moving to another directory).

The directory entry is deleted by setting its **OFF_FSTORE** byte to $00. If this directory entry is the last in a particular directory block, the block is removed and the previous directory block's next-block pointer (at **OFFB_NXTBLK**) is given the deleted directory's next-block pointer, thereby maintaining the continuity of the directory block chain.

After deleting the directory entry, **DeleteFile** calls **UpdateParent** to update the status bytes of the parent directory.

DeleteFile cannot delete a subdirectory. To delete a subdirectory, use **DeleteDir**.

Example:

See also: **DeleteDir, FreeDir, FreeFile, FreeBlock.**

DeleteRecord (Apple, C64, C128) VLIR disk

Function: Removes the current VLIR record from the record list, moving all subsequent records upward to fill the slot and freeing all the data blocks associated with the record.

Parameters: none.

Uses:

curDrive	
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM/VBM into memory.
curRecord	current record pointer
fileHeader	VLIR index table stored in this buffer.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	current directory header/BAM.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:

curRecord	only changed if deleting the last record in the table, in which case it becomes the new last record.
fileWritten†	set to **TRUE** to indicate the file has been altered since last updated.
fileHeader	new record added to index table.
fileSize	decremented to reflect any deleted record blocks.

Commodore:

curDirHead	current directory header/BAM modified to free blocks.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

fileBytes†	decremented to reflect any deleted record data.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Apple:
a, y, r1–r2, r4, r6, r7, r8H.

Commodore:
a, y, r0–r9.

Description: **DeleteRecord** removes the current record from the record list by moving all subsequent records upward to fill the current record's slot. Any data blocks associated with the record are freed.

DeleteRecord does not update the BAM/VBM and VLIR file information on the disk. Call **CloseRecordFile** or **UpdateRecordFile** to update the file when done modifying.

Example:

See also: **AppendRecord, InsertRecord.**

DisablSprite (Apple, C64, C128) sprite

Function: Disable a sprite so that it is no longer visible.

Parameters: **r3L** SPRITE — sprite number (byte).

Returns: nothing.

Alters: **mobenble**

Destroys: **a, x**

Description: **DisablSprite** disables a sprite so that it is no longer visible. Although there are eight sprites available, an application should only directly disable sprite #2 through sprite #7 with **DisablSprite**. Sprite #0 (the mouse pointer) is always enabled when GEOS mouse-tracking is enabled (disable mouse-tracking with **mouseOff**), and sprite #1 (the text cursor) should be disabled with **PromptOff**.

Example:

See also: **EnablSprite, MouseOff, PromptOff, DrawSprite, GetSpriteData, PosSprite, InitSprite.**

DivideBySeven (Apple) graphics

Function: Quickly divide a word value (between 0 and 649) by seven for direct screen access graphic operations.

Parameters:
- **a**]DIVIDEND — high byte of word to divide by seven (byte).
- **x** [DIVIDEND — low byte of word to divide by seven (byte).

Returns:
- **a** quotient (byte).
- **y** remainder (byte).
- **x** unchanged.

Destroys: nothing.

Description: **DivideBySeven** is a highly specialized divide routine designed for applications that require direct screen access. The Apple's double high-res screen is based on a seven-bit byte which requires dividing the x-position by seven to calculate the byte-position. The quotient is the byte position in a scanline and the remainder is the bit-position in that byte. **DivideBySeven** uses a series of lookup tables to quickly calculate the quotient and the remainder.

Note: Although the maximum screen x-position on the Apple is 559 (**SC_PIX_WIDTH** – 1), **DivideBySeven** will handle values up to 649 so that graphics can be generated for 640-column printer drivers.

The average computational time for **DivideBySeven** is roughly 21 cycles.

Example:

```
;XorPoint       -- invert a single screen point on Apple.
;
;Pass:          r3    x-coordinate
;               r11L  y-coordinate
;
;Uses:          dispBufferOn
;
;Destroys:      a, x, y, r5, r6.
;
XorPoint:
        jsr     TempHideMouse       ;turn off sprites temporarily
        ldx     r11L                ;get y-coordinate
        jsr     GetScanLine         ;set r5 and r6 with scanline addressses
        lda     r3H                 ;get hi of x-coordinate
        ldx     r3L                 ;get lo of x-coordinate
        jsr     DivideBySeven       ;calculate byte and bit offsets
        pha                         ;save byte offset
        tya                         ;transfer bit offset to x-register
        tax                         ;to pass to SetCorrectPage
        pla                         ;and put byte offset into a-reg
        jsr     SetCorrectPage      ;(see GetScanLine reference)
        lda     GrBitTable,x        ;get pixel bit-mask
        eor     (r6),y              ;xor pixel in first buffer
        sta     (r5),y              ;store into both buffers
        sta     (r6),y              ;if necessary
        rts

GrBitTable:
        .byte %00000001,%00000010,%00000100,%00001000
```

```
.byte %00010000,%00100000,%01000000,%10000000
```

See also: **GetScanLine, NormalizeX, Ddiv.**

DMult (Apple, C64, C128) math

Function: Unsigned word-by-word (double-precision) multiply: multiplies two unsigned words to produce an unsigned word result.

Parameters:
- **x** OPERAND1 — zero-page address of word multiplicand (byte pointer to a word variable).
- **y** OPERAND2 — zero-page address of word multiplier (byte pointer to a word variable).

Note: *result= OPERAND1(word) * OPERAND2(word).*

Returns: **x**, **y**, word pointed to by *OPERAND2* unchanged.
word pointed to by *OPERAND1* contains the word result.

Destroys: **a**, **r6–r8**.

Description: **DMult** is an unsigned word-by-word multiplication routine that multiplies the word at one zero-page address by the word at another to produce a 16-bit word result (all stored in low/high order). The word in *OPERAND1* is multiplied by the word in *OPERAND2* and the result is stored as a word back in *OPERAND1*.

Note: Because **r6**, **r7** and **r8** are destroyed in the multiplication process, they cannot be used to hold the operands.

Overflow in the result (beyond 16-bits) is ignored.

Example:

SeeAlso: **BMult**, **BBMult**, **Ddiv**, **DSdiv**.

This page intentionally left blank to maintain right/left (verso/recto) page ordering. Final version will correct this.

Dnegate (Apple, C64, C128) math

Function: Negate a signed word (two's complement sign-switch).

Parameters: **x** OPERAND — zero-page address of word to operate on (byte pointer to a word variable).

Returns: **x,y** unchanged
word pointed to by *OPERAND* contains the negated word.

Destroys: **a**

Description: **DNegate** negates a zero-page word. The absolute address of the word (*OPERAND*) is passed in **x**. The absolute value of *OPERAND* is returned in *OPERAND*.

The operation of this routine is: **value = (value ^ $ffff) + 1.**

Example:

SeeAlso: **Dabs.**

DoBOp (C128) memory

Function: Back-RAM memory move/swap/verify primitive.

Parameters:

r0	ADDR1 — address of first block in application memory (word).
r1	ADDR2 — address of second block in application memory (word).
r2	COUNT — number of bytes to operate on (word).
r3L	A1BANK — *ADDR1* bank: 0 = front RAM; 1 = back RAM (byte).
r3H	A2BANK — *ADDR2* bank: 0 = front RAM; 1 = back RAM (byte).
y	MODE — operation mode:

b1	b0	Description
0	0	move from memory at *ADDR1* to memory at *ADDR2*.
0	1	move from memory at*ADDR2* to memory at *ADDR1*.
1	0	swap memory at *ADDR1* with memory at *ADDR2*.
1	1	verify (compare) memory at *ADDR1* against memory at *ADDR2*.

Note: the DoBOp MODE parameter closely matches the low nyblle of the DoRAMOp CMD parameter.

Returns: **r0–r3** unchanged.
When verifying:
x $00 if data matches; $ff if mismatch.

Destroys: **a, x, y**

Description: **DoBOp** is a generalized memory primitive for dealing with both memory banks on the Commodore 128. It is used by **MoveBData, SwapBData,** and **VerifyBData.**

Note: **DoBOp** should only be used on designated application areas of memory. When moving memory within the same bank, the destination address must be less than source address. When swapping memory within the same bank, *ADDR1* must be less than *ADDR2*.

Example:

See also: **MoveBData, SwapBData, VerifyBData, DoRAMOp.**

DoDlgBox (Apple, C64, C128) — dialog box

Function: Initializes, displays, and begins interaction with a dialog box.

Parameters: **r0** DIALOG — pointer to dialog box definition (word).
r5–r10 can be used to send parameters to a dialog box.

Returns: **r0L** return code: typically the number of the system icon clicked on to exit.
*Note: returns when dialog box exits through **RstrFrmDialog**.*

Destroys: n/a

Description: **DoDlgBox** saves off the current state of the system, places GEOS in a near-warmstart state, displays the dialog box according to the definition table (whose address is passed in **r0**), and begins tracking the user's interaction with the dialog box. When the dialog box finishes, the original system state is restored, and control is returned to the application.

Simple dialog boxes will typically contain a few lines of text and one or two system icons (such as **OK** and **CANCEL**). When the user clicks on one of these icons, the GEOS system icon routine exits the dialog box with an internal call to **RstrFrmDialog**, passing the number of the system icon selected in **sysDBData**. **RstrFrmDialog** restores the system state and copies **sysDBData** to **r0L**.

More complex dialog boxes will have application-defined icons and routines that get called. These routines, themselves, can choose to load a value into **sysDBData** and call **RstrFrmDialog**.

Dialog boxes cannot be nested. That is, a dialog box should never call **DoDlgBox**.

Apple: **DoDlgBox** automatically calls **InitForDialog** as part of its initialization sequence.

Note: Part of the system context save within **DoDlgBox** saves the current stack pointer. Dialog boxes cannot be nested. **DoDlgBox** is not reentrant. That is, a dialog box should never call **DoDlgBox**.

Example:

See also: **RstrFrmDlg, InitForDialog, RestoreSysRam, SaveFG.**

DoIcons (Apple, C64, C128) icon/menu

Function: Display and activate an icon table.

Parameters: **r0** ICONTABLE — pointer to the icon table to use.

Uses: **dispBufferOn:**
bit 7 — draw icons to foreground screen if set.
bit 6 — draw icons to background screen if set.

Destroys: **r0–r15, a, x, y**

Description: **DoIcons** takes an icon, draws the enabled icons (those whose **OFF_PIC_ICON** word is non-zero) and instructs **MainLoop** to begin tracking the user's interaction with the icons. This routine is the only way to install icons. Every application should install at least one icon, even if only a dummy icon.

If **DoIcons** is called while another icon table is active, the new icons will take precedence. The old icons are not erased from the screen before the new ones are displayed.

DoIcons is a complex routine which affects a lot of system variables and tables. The following is an outline of its major actions:

1: All enabled icons in the table are drawn to the foreground screen and/or the background buffer based on the value in **dispBufferOn**.

2: **StartMouseMode** is called. If the **OFF_IC_XMOUSE** word of the icon table header is non-zero, then **StartMouseMode** loads **mouseXPosition** and **mouseYposition** with the values in the **OFF_IC_XMOUSE** and the **OFF_IC_YMOUSE** parameters of the icon table header (see **StartMouseMode** for more information).

4: **faultData** is cleared to $00, indicating no faults.

5: If the **MOUSEON_BIT** of **mouseOn** is *clear*, then the **MENUON_BIT** is forced to one. This is because GEOS assumes that it is in a power-up state and that mouse tracking should be fully enabled. If the **MOUSEON_BIT** bit is set, GEOS leaves the menu-scan alone, assuming that the current state of the **MENUON_BIT** is valid.

6: The **ICONSON_BIT** and **MOUSEON_BIT** bits of **mouseOn** are set, thereby enabling icon-scanning.

When an icon event handler is given control, **r0L** contains the number of the icon clicked on (beginning with zero) and **r0H** contains **TRUE** if the event is a double-click or **FALSE** if the event is a single click.

Example:

See Also: **DoMenu.**

DoInlineReturn (Apple, C64, C128) utility

Function: Return from an inline subroutine.

Parameters:
- **a** DATABYTES — number of inline data bytes following the **jsr** plus one(byte).
- ***stack*** top byte on stack is the status register to return (execute a **php** just before calling).

Returns: (to the inline **jsr**) **x**, **y** unchanged from the **jmp DoInlineReturn**. **st** register is pulled from top of stack with a **plp**.

Uses: **returnAddress** return address as popped off of stack.

Destroys: **a**

Description: **DoInlineReturn** simulates an **rts** from an inline subroutine call, properly skipping over the inline data. Inline subroutines (such as the GEOS routines which begin with **i_**) expect parameter data to follow the subroutine call in memory. For example, the GEOS routine **i_Rectangle** is called in the following fashion:

```
jsr          i_Rectangle    ;subroutine call
.byte    Y1,Y2              ;inline data
.word    X1,X2
jsr          FrameRectangle        ;returns to here
```

Now if **i_Rectangle** were to execute a normal **rts**, the program counter would be loaded with the address of the inline data following the subroutine call. Obviously, inline subroutines need some means to resume processing at the address following the data. **DoInlineReturn** Provides this facility. The normal return address is placed in the global variable **returnAddress**. This is the return address as it is popped off the stack, which means it points to the third byte of the inline **jsr** (an **rts** increments the address before resuming control). The status registers is pushed onto the stack with a **php**, **DoInlineReturn** is called with the number of inline data bytes plus one in the accumulator, and control is returned at the instruction following the inline data.

Inline subroutines operate in a consistent fashion. The first thing one does is pop the return address off of the stack and store it in **returnAddress**. It can then index off of **returnAddress** as in **lda (returnAddress),y** to access the inline parameters, where the y-register contains $01 to access the first parameter byte, $02 to access the second, and so on (not $00, $01, $02, as might be expected because the address actually points to the third byte of the inline **jsr**). When finished, the inline subroutine loads the accumulator with the number of inline data bytes and executes a **jmp DoInlineReturn**.

Note: **DoInlineReturn** must be called with a **jmp** (not a **jsr**) or an unwanted return address will remain on the stack. The **x** and **y** registers are not modifiend by **DoInlineReturn** and can be used to pass parameters back to the caller. Inline calls cannot be nested without saving the contents of **returnAddress**. An inline routine will not work correctly if not called directly through a **jsr** (e.g., **CallRoutine** cannot be used to call an inline subroutine).

Example:

```
;i_VerticalLine
;
;     Inline version of VerticalLine.
;
;Pass:
;     .word     x1
;     .word     x2
;     .byte     y1
;
i_VerticalLine:
V_BYTES       = 5                        ;number of inline bytes in call

;Save away the inline return address
      PopW      returnAddress

;Load up VerticalLine's parameters
      ldy       #V_BYTES
      lda       (returnAddress),y          ;get y1 parameter first
      sta       r11L
10$:
      dey                                  ;load other params in a loop
      lda       (returnAddress),y          ;They occupy consecutive GEOS
      sta       r3L-1,y                    ;pseudoregisters, so this will.
      cpy       #1                         ;work correctly
      bne       10$

;Now call VerticalLine with registers loaded
      jsr       VerticalLine

;and do an inline return when we come back
      php                                  ;save st reg to return
      lda       #V_BYTES+1                 ;# of bytes + 1
      jmp       DoInlineReturn             ;jump to inline return. Do not jsr!
```

DoMenu (Apple, C64, C128) icon/menu

Function: Displays and activates a menu structure.

Parameters:
- **r0** MENU — pointer to the menu structure to display.
- **a** POINTER_OVER — which menu item (numbered starting with zero) to center the pointer over.

Destroys: **r0–r13, a, x, y**

Description: **DoMenu** draws the main menu (the first menu in the menu structure) and instructs **MainLoop** to begin tracking the user's interaction with the menu. This routine is the only way to install a menu.

If **DoMenu** is called while another menu structure is active, the new menu will take precedence. The old menu is not erased from the screen before the new menu is displayed. If the new menu is smaller (or at a different position) than the old menu, parts of the old menu may be left on the screen. A typical way to avoid this is to erase the old menu with a call to **Rectangle**, passing the positions of the main menu rectangle and drawing in a white pattern. However, a more elegant solution involves calling **RecoverAllMenus**, which will erase any extant menus by recovering from the background buffer.

DoMenu is a complex routine which affects a lot of system variables and tables. The following is an outline of its major actions:

1: Menu level 0 (main menu) is drawn to the foreground screen.

2: **StartMouseMode** is called. **mouseXPosition** and **mouseYposition** are set so that the pointer is centered over the selection number passed in **a**. Under Apple GEOS, if the the POINTER_OVER number in the accumulator has its high-bit set, then the mouse will not be repositioned. Under GEOS 64 and GEOS 128, **DoMenu** always forces the mouse to a new position. If you do not want the mouse moved, surround the call to **DoMenu** with code to save and restore the mouse positions. The following code fragment will install menus without moving the mouse.

```
DoMenu2:
    .if (APPLE)
    lda     #$80            ;set high-bit so mouse is not repositioned
    jsr     DoMenu          ;put up the menu
    .else ;(C64 || C128)
    php                     ;disable interrupts around call
    sei
    PushW   mouseXPos       ;save mouse x
    PushB   mouseYPos       ;save mouse y
    lda     #0              ;dummy menu value
    jsr     DoMenu          ;install menus (mouse will move)
    PopB    mouseYPos       ;restore original mouse y
    PopW    mouseXPos       ;restore original mouse x
    plp                     ;restore interrupts
    .endif
    rts
```

3: **SlowMouse** is called. With a joystick this will kill all accumulated speed in the pointer, requiring the user to reaccelerate. With a proportional mouse, this will have no effect.

4: **faultData** is cleared to $00, indicating no faults.

5: If the **MOUSEON_BIT** of **mouseOn** is *clear*, then the **ICONSON_BIT** is forced to one. This is because GEOS assumes that it is in a power-up state and that mouse tracking should be fully enabled. If the **MOUSEON_BIT** bit is set, GEOS leaves the icon-scan alone, assuming that the **ICONSON_BIT** is valid.

6: The **MENUON_BIT** and **MOUSEON_BIT** bits of **mouseOn** are set, thereby enabling menu-scanning.

7: The mouse fault variables (**mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight**) are set to the full screen dimensions.

Example:

See Also: **DoIcons**, **GotoFirstMenu**, **DoPreviousMenu**, **ReDoMenu**.

DoneWithIO (C64, C128) — very low-level disk

Function: Restore system after I/O across the serial bus.

Parameters: none.

Returns: nothing.

Destroys: a, y.

Description: **DoneWithIO** restores the state of the system after a call to **InitForIO**. It restores the interrupt status, turns sprite DMA back on, returns the 128 to its original clock speed, and switches out the ROM and I/O banks if appropriate (only on C64).

Disk and printer routines access the serial bus between calls to **InitForIO** and **DoneWithIO**.

Apple: Apple GEOS has no **DoneWithIO** equivalent.

Example: See **WriteBlock**.

See also: **InitForIO**.

DoPreviousMenu (Apple, C64, C128) icon/menu

Function: Retracts the current sub-menu and reactivates menus at the previous level.

Parameters: none.

Destroys: assume **r0–r15, a, x, y**

Description: **DoPreviousMenu** is used by a menu event handler to instruct GEOS to back up one level of menus, erasing the current menu from the foreground screen and making the parent menu active when control is returned to **MainLoop**. **menuNumber** is decremented.

When using **DoPreviousMenu**, if the parent menu (the one which will be given control) is of type **UN_CONSTRAINED**, then the mouse *must* be manually repositioned over the parent menu. This can be done by loading **mouseXPosition** and **mouseYPosition** with values calculated from the menu structure. If the parent menu is of type **CONSTRAINED**, then the mouse is automatically positioned over the selection in the parent menu which led to the sub-menu.

Note: **DoPreviousMenu** may be called repeatedly to back up more than one level.

Do not call **DoPreviousMenu** when the menu is at level 0 (**menuNumber** = $00). The effects may be disasterous.

Example:

See Also: **DoMenu, GotoFirstMenu, ReDoMenu, RecoverMenu.**

DoRAMOp (C64 v1.3 & C128) memory

Function: Primitive for communicating with REU (RAM-Expansion Unit) devices.

Parameters:
- **r0** ADDR1 — address in Commodore to start (word).
- **r1** ADDR2 — address in REU bank to start (word).
- **r2** COUNT — number of bytes to operate with (transfer length) (word).
- **r3L** REUBANK — REU bank number to use (byte).
- **y** CMD — command to send to REU (byte).

Returns:
- **r0–r3** unchanged.
- **x** error code: $00 (no error) or **DEV_NOT_FOUND** if bank or REU not available.
- **a** REU status byte and'ed with $60

Destroys: y

Description: **DoRAMOp** is a very low-level routine for communicating with a RAM-expansion unit on a C64 or C128. This routine is a "use at your own risk" GEOS primitive

DoRAMOp operates with the with the RAM-expansion unit directly and handles all the necessary.communication protocols and clock-speed save/restore (if necessary).

The *CMD* parameter is stuffed into the REC Command Register (**EXP_BASE**+$01). Although **DoRAMOp** does no error checking on this parameter, it expects the high-nybble to be %1001 (transfer with current configuration and disable FF00 decoding). The lower nybble can be one of the following:

- %00 Transfer from Commodore to REU.
- %01 Transfer from REU to Commodore.
- %10 Swap.
- %11 Verify.

Note: the low nybble of the DoRAMOp CMD parameter closely matches the DoBOp MODE parameter.

Note: On a Commodore 128, if the VIC chip is mapped to front RAM (with the MMU VIC bank pointer), the REU will read/write using front RAM. Similarly, if the VIC chip is mapped to back RAM, the REU will read/write using back RAM. The REU ignores the stadard bank selection controls on the 8510. GEOS 128 defaults with the VIC mapped to front RAM.

For more information on the Commodore REU devices, refer to the *Commodore 1764 RAM Expansion Module User's Guide* or the *1700/1750 RAM Expansion Module User's Guide.*

Example:

See also: **StashRAM, FetchRAM, SwapRAM, DoBOp.**

DownDirectory (Apple) — mid-level disk

Function: Makes a subdirectory (child directory) in the current directory the new current directory.

Parameters:

r9	DIRENTRY — pointer to directory entry of subdirectory, usually points to **dirEntryBuf** (must be in main memory) (word).

Uses:

curDrive	
curKBlkno	current directory.

Returns:

x	error ($00 = no error).
y	pathname status ($00 = OK; **BFR_OVERFLOW** = pathname longer than **pathnameBuf**).

Alters:

curKBlkno	new current directory.
curDirHead	header of new directory.
pathnameBuf†	system pathname buffer updated to reflect new path.
curDirTabLo†	
curDirTabHi†	

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, r1, r4.**

Description: **DownDirectory** moves down one level in the hierarchical file system, making a specific child directory the current working directory.

The directory entry of the subdirectory is a standard data structure returned by routines such as **FindFile**, **Get1stDirEntry** and **GetNextDirEntry**. A subdirectory has a **PRO_DIR** (ProDOS directory) GEOS file type.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **UpDirectory, GoDirectory.**

DrawLine (Apple, C64, C128) graphics

Function: Draw, clear, or recover a line defined by two arbitrary endpoints.

Parameters:
r3 X1 — x-coordinate of first endpoint (word).
r11L Y1 — y-coordinate of first endpoint (byte).
r4 X2 — x-coordinate of second endpoint (word).
r11H Y2 — y-coordinate of second endpoint (byte).
st MODE:

n	c	Description
1	x	recover pixel from background screen to foreground.
0	1	set pixel using **dispBufferOn**.
0	0	clear pixel using **dispBufferOn**.

where (X1,Y1) and (X2,Y2) are the two endpoints of the line.

Uses: if **n** is set (drawing, not recovering):
dispBufferOn:
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: status register (S) unchanged

Destroys:
Commodore
a, x, y, r3–r13

Apple
a, x, y, r3, r4, r11

Description: **DrawLine** will set, clear, or recover the pixels which comprise the line between two arbitrary endpoints. Setting a pixel sets its bit value to one, clearing a pixel sets its bit value to zero, and recovering a pixel copies the bit value from the background buffer to foreground screen.

DrawLine uses the Bresenham DDA (Digital Differential Analyzer) algorithm to determine the proper points to draw. The line will be drawn correctly regardless of which endpoint is used for (*X1,Y1*) and which is used for (*X2,Y2*). In fact, the line is reversible: the same line will be drawn even if the endpoints are swapped.

The carry (c) flag and sign (n) flag in the processor status register (s) are used to pass information to **DrawLine**. The following tricks can be used to set or clear these flags appropriately:

- Use **sec** and **clc** to set or clear the carry (c) flag.
- Use **lda #[-1** to set the sign (n) flag.
- Use **lda #0** to clear the sign (n) flag.

Note: Calculates each pixel position on the line and calls **DrawPoint** repeatedly.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **HorizontalLine,VerticalLine, InvertLine, ImprintLine, RecoverLine.**

DrawPoint (Apple, C64, C128) graphics

Function: Set, clear, or recover a single screen point (pixel).

Parameters:
r3 X1 — x-coordinate of pixel (word).
r11L Y1 — y-coordinate of pixel (byte).
st MODE:

n	c	Description
1	x	recover pixel from background screen to foreground.
0	1	set pixel using **dispBufferOn**.
0	0	clear pixel using **dispBufferOn**.

where (X1,Y1) is the coordinate of the point.

Uses: when setting or clearing pixels:
dispBufferOn:
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: **r3**, **r11L** unchanged.

Destroys: Commodore
a, **x**, **y**, **r5-r6**

Apple
a, **x**, **y**

Description: **DrawPoint** will set, clear, or recover a single pixel. Setting a pixel sets its bit value to one, clearing a pixel sets its bit value to zero, and recovering a pixel copies the bit value from the background buffer to foreground screen.

The carry (c) flag and sign (n) flag in the processor status register (s) are used to pass information to **DrawPoint**. The following tricks can be used to set or clear these flags appropriately:

- Use **sec** and **clc** to set or clear the carry (c) flag.
- Use **lda #[-1** to set the sign (n) flag.
- Use **lda #0** to clear the sign (n) flag.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **TestPoint**, **DrawLine**.

DRAW SPRITE
~~DrawPoint~~

DrawSprite (Apple, C64, C128) sprite

Function: Copy a 64-byte sprite image to the internal data buffer that is used for drawing the sprites.

Parameters:
r3L SPRITE — sprite number (byte).
r4 DATAPTR — pointer to 64-bytes of sprite image data (word).

Returns: nothing.

Alters: internal sprite image.

Destroys:
Commodore
a, y, r5

Apple
a, y

Description: **DrawSprite** copies 64-bytes of sprite image data to the internal buffer that is used for drawing the sprites. **DrawSprite** does not affect the enabled/disabled status of a sprite, it only changes the image definition.

Although there are eight sprites available, an application should limit itself to sprites #2 through #7 because GEOS reserves sprite #0 for the mouse cursor and sprite #1 for the text prompt.

C64: The 64 bytes are copied to the VIC sprite data area, which is located in memory immediately after the color matrix. The size information byte (byte 64) is unused by GEOS 64 but is copied to the data area, nonetheless. A *SPRITE* value of $00 can be used to change the shape of the mouse cursor.

C128: The data is transferred to the VIC sprite area (regardless of the current graphics mode). This data is used by the VIC chip in 40-column mode and by the soft-sprite handler in 80-column mode. The last byte (byte 64) of the sprite definition is used as the size information byte by the soft-sprite handler. In 80-column mode, the sprite is not visually updated until the next time the soft-sprite handler gets control. To change the mouse cursor, the application can use a *SPRITE* value of $00 in 40-column mode or call **SetMsePic** in 80-column mode (doing both is a simple solution: it will do no harm regardless of the graphics mode).

Apple: The data is transferred to an internal sprite area. The last byte (byte 64) of the sprite definition is used as the size information byte. The sprite is not visually updated until the next time the soft-sprite handler gets control. The soft-sprite handler will draw sprite #1 through sprite #7. In no case should the *SPRITE* parameter be $00; a value of $00 will most likely trample GEOS.

The **INAUX_B** constant may be or'ed into the *SPRITE* parameter to indicate that the *DATAPTR* parameter is an address in auxiliary memory.

Example:

See also: **GetSpriteData, PosSprite, EnablSprite, DisablSprite, InitSprite.**

DSdiv (Apple, C64, C128) math

Function: Signed word-by-word (double-precision) division: divides one two's complement word by another to produce a signed word result.

Parameters:
x OPERAND1 — zero-page address of signed word dividend (byte pointer to a word variable).
y OPERAND2 — zero-page address of signed word divisor (byte pointer to a word variable).

Note: *result = OPERAND1(word) / OPERAND2(word).*

Returns:
y unchanged.
r8 the fractional remainder (word).
word pointed to by *OPERAND2* equals its absolute value.
word pointed to by *OPERAND1* contains the word result.

Commodore:
x unchanged

Destroys: **a, r9**

Apple:
x

Description: DSdiv is a signed, two's complement word-by-word division routine that divides the word in one zero-page pseudoregister (the dividend) by the word in another (the divisor) to produce a 16-bit word signed result and a 16-bit word fractional remainder The word in *OPERAND1* is divided by the word in *OPERAND2* and the result is stored as a word back in *OPERAND1* with the remainder in **r8**.

C64 & C128: The remainder is always positive regardless of the sign of the dividend. This will cause problems with some mathematical operations that expect a signed remainder. The following code fragment will fixe this problem:

```
;NewDSdiv:     call as you would call DSdiv. Returns a
;              signed remainder. Destroys x!
;
;              sign(remainder) = sign (dividend)
NewDSdiv:
     lda       zpage,x              ;save sign of dividend
     php
     jsr       DSdiv                ;divide as normal
     plp                            ;then get back sign of dividend back
     bpl       20$                  ;ignore if positive
     ldx       #r8                  ;else, negate remainder
     jsr       Dnegate              ;(e.g., -10/9 = -1 rem -1)
20$:
     rts
```

Apple: The sign of the remainder is the same as the sign of the dividend.

Note: Because **r8** and **r9** are used in the division process, they cannot be used to hold the operands.

Although dividing by zero is an undefined mathematical operation, **DSdiv** makes no attempt to flag this as an error condition and simply returns incorrect results. If the divisor might be zero, the application should check for this situation before dividing:

```
     lda      zpage,y            ;get low byte of divisor
     ora      zpage+1,y          ;get high byte of divisor
     bne      10$                ;if either non-zero, go divide
     jmp      DivideByZero       ;else, flag error
10$:
     jmp      DSdiv              ;divide (DSdiv will rts)
```

Example:

SeeAlso: **Ddiv, DMult, BMult, BBMult, DivideBySeven.**

DShiftLeft (Apple, C64, C128) math

Function: Arithmetically left-shift a zero-page word.

Parameters:
- **x** OPERAND — address of the zero-page word to shift (byte pointer to a word variable).
- **y** COUNT — number of times to shift the word left (byte).

Returns:
- **a, x** unchanged
- **y** #$ff
- **st** c (carry flag) is set with last bit shifted out of word.

zero page address pointed to by*OPERAND* contains the shifted word.

Destroys: nothing

Description: **DShiftLeft** is a double-precision math routine that arithmetically left-shifts a 16-bit zero-page word (low/high order). The absolute address of the word is passed in **x** and the number of times to shift the word is passed in **y**. Zeros are shifted into the low-order bit.

An arithmetic left-shift is useful for quickly multiplying a value by a power of two. One left-shift will multiply by two, two left-shifts will multiply by four, three left-shifts will multiply by eight, and so on: **value = value * 2^{count}**.

Note: If a *COUNT* of $00 is specified, the the word will not be shifted.

Example:

SeeAlso: **DShiftRight.**

DShiftRight (Apple, C64, C128) math

Function: Arithmetically right-shift a zero-page word.

Parameters:
- **x** OPERAND — zero-page address of word to shift (byte pointer to a word variable).
- **y** COUNT — number of times to shift the word right (byte).

Returns:
- **a, x** unchanged
- **y** #$ff
- **st** c (carry flag) is set with last bit shifted out of word.

register pointed to by *OPERAND* contains the shifted word.

Destroys: nothing

Description: **DShiftRight** is a double-precision routine that arithmetically right-shifts a 16-bit zero-page word (low/high order). The address of the word is passed in **x** and the number of times to shift the word is passed in **y**. Zeros are shifted into the high-order bit.

An arithmetic right-shift is useful for quickly dividing a value by a power of two. One left-shift will divide by two, two left-shifts will divide by four, three left-shifts will divide by eight, and so on: **value = value / 2^{count}**.

Example:

SeeAlso: **DShiftLeft.**

EnableProcess (Apple, C64, C128) process

Function: Makes a process runable immediately.

Parameters: x PROCESS — number of process (0 – *n*–1, where *n* is the number of processes in the table) (byte).

Returns: x unchanged.

Destroys: a

Description: **EnableProcess** forces a process to become runable on the next pass through **MainLoop**, independent of its timer value.

EnableProcess merely sets the runable flag in the hidden process table. When **MainLoop** encounters an unblocked process with this flag set, it will attempt to generate an event just as if the timer had decremented to zero.

EnableProcess has no priveleged status and cannot override a blocked process. However, because it doesn't depend on or affect the current timer value, the process can become runable even with a frozen timer.

EnableProcess is useful for making sure a process runs at least once, regardless of the initialized value of the countdown timer. It is also useful for creatingapplication-defined events which run off of **MainLoop**: a special process can be reserved in the data structure but never started with **RestartProcess**. Any time the desired event-state is detected, a call to **EnableProcess** will generate an event on the next pass through **MainLoop**. **EnableProcess** can be called from Interrupt Level, which allows a condition to be detected at Interrupt Level but processed during **MainLoop**.

Example:

See also: **InitProcesses, RestartProcess, UnfreezeProcess, UnblockProcess.**

EnablSprite (Apple, C64, C128) sprite

Function: Enable a sprite so that it becomes visible.

Parameters: **r3L** SPRITE — sprite number (byte).

Returns: nothing.

Alters: **mobenble**

Destroys: **a, x**

Description: **EnablSprite** enables a sprite so that it becomes visible. Although there are eight sprites available, an application should only directly enable sprites #2 through #7 with **EnablSprite**. Sprite #0 (the mouse pointer) is enabled through **MouseOn** and **StartMouseMode**, and sprite #1 (the text cursor) should be enabled with **PromptOn**.

Example:

See also: **DisablSprite, MouseUp, PromptOn, DrawSprite, GetSpriteData, PosSprite, InitSprite.**

EnterDeskTop (Apple, C64, C128) high-level disk

Function: Standard application exit to GEOS deskTop.

Parameters: none.

Returns: *never returns to application.*

Description: An application calls **EnterDeskTop** when it wants to exit to the GEOS deskTop. **EnterDeskTop** takes no parameters and looks for a copy of the file **DESK TOP** on each drive. Later versions of GEOS are only compatible with the correspondingly later revision of the deskTop and will check the version number in the permanent name string of the **DESK TOP** file to ensure that it is in fact a newer version. If after all drives are searched no valid copy of the deskTop is found, **EnterDeskTop** will prompt the user to insert a disk with a copy of the deskTop on it.

C64 & C128: **EnterDeskTop** will first search a RAMdisk for a copy of the deskTop to ensure the fastest loading time.

Apple: **EnterDeskTop** will first search all non-removable media devices (RAMdisk, hard disk, etc.) for a copy of the deskTop, then other devices (floppy disk devices). These devices are searched in the order that they appear on the deskTop, which corresponds to their order in the expansion slots: a device in slot seven will be searched before a device in slot six.

Example:

See also: **RstrAppl, GetFile.**

EnterTurbo (C64, C128) — very low-level disk

Function: Activate disk drive turbo mode.

Parameters: none.

Uses:

curDrive	currently active disk drive.
curType	v1.3+: checks disk type because not all use turbo software.

Returns: **x** error ($00 = no error).

Destroys: **a, y.**

Calls:

Description: **EnterTurbo** activates the turbo software in the current drive. If the turbo software has not yet been downloaded to the drive, **EnterTurbo** will download it. The turbo software allows GEOS to perform high-speed serial disk access.

EnterTurbo treats different drive types appropriately. A RAMdisk, for example, does not use turbo code so **EnterTurbo** will not attempt to download the turbo software.

The very-low level Commodore GEOS read/write routines, such as **ReadBlock**, **WriteBlock**, **VerWriteBlock**, and **ReadLink**, expect the turbo software to be active. Call **EnterTurbo** before calling one of these routines.

Apple: Apple GEOS has no **EnterTurbo** equivalent.

Example: See **WriteBlock**.

See also: **ExitTurbo, PurgeTurbo.**

ERASE CHARACTER
~~EnterTurbo~~

EraseCharacter (Apple) text

Function: Erase a character from the screen, accounting for the current font and style attributes.

Parameters:
a CHAR — character code of character to erase (byte).
r11 XPOS — x-coordinate of left of character (word).
r1H YPOS — y-coordinate of character baseline (word).

Uses: same as **PutChar.**

Returns: **r11, r1H** unchanged.

Alters: **lastWidth** width of character just erased.

Destroys:
Commodore
r1L, r2–r10, r12, r13, a, x, y.

Apple
r1L, r2, a, x, y.

Description: **EraseCharacter** calculates the width of the character and erases it from the screen using the **USELAST** character and **SmallPutChar.**

Note: Uses **currentMode** to calculate the character width. When deleting multiple characters, the application will need to take this into account when backspacing from one style to another. Also, does not always work with characters that are both bolded and outlined because not all fonts have a wide enough **USELAST** character.

See also: **SmallPutChar, PutChar, PutString.**

Example:

ExitTurbo (C64, C128) — very low-level disk

Function: Deactivate disk drive turbo mode.

Parameters: none.

Uses: **curDrive** currently active disk drive.

Returns: **x** error ($00 = no error).

Destroys: **a, y.**

Description: **ExitTurbo** deactivates the turbo software in the current drive so that the serial bus may access another device. **SetDevice** automatically calls this before changing devices.

Note: If the turbo software has not been downloaded or is already inactive, **ExitTurbo** will do nothing.

Apple: Apple GEOS has no **ExitTurbo** equivalent.

Example: See **WriteBlock.**

See also: **EnterTurbo, PurgeTurbo.**

FastDelFile (C64, C128) mid-level disk

Function: Special Commodore version of **DeleteFile** that quickly deletes a sequential file when the track/sector table is available.

Parameters:
- **r0** FILENAME — pointer to null-terminated file name (word).
- **r3** TSTABLE — pointer to track and sector table of file, usually points to **fileTrScTab** (word).

Uses:
- **curDrive.**
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.
- **curDirHead** BAM updated to reflect newly freed blocks.
- **dir2Head**† (BAM for 1571 and 1581 drives only)
- **dir3Head**† (BAM for 1581 drive only)

Returns: **x** error ($00 = no error).

Destroys: **a, y, r0, r9.**

Description: **FastDelFile** quickly deletes a sequential file by taking advantange of an already existing track/sector table. It first removes the directory entry determined by *FILENAME* and calls **FreeBlock** for each block in a track/sector table at *TSTABLE*. The track/sector table is in the standard format, such as that returned from **ReadFile**, where every two-byte entry constitutes a track and sector. A track number of $00 terminates the table.

FastDelFile is fast because it does not need to follow the chain of sectors to delete the individual blocks. It can do most of the deletion by manipulating the BAM in memory then writing it out with a call to **PutDirHead** when done.

FastDelFile will not properly delete VLIR files without considerable work on the application's part. Because there is no easy way to build a track/sector table that contains all the blocks in all the records of a VLIR file, it is best to use **DeleteFile** or **FreeFile** for deleting VLIR files or **DeleteRecord** for deleting a single record.

FastDelFile calls **GetDirHead** before freeing any blocks. This will overwrite any BAM and directory header in memory.

Note: **FastDelFile** can be used to remove a directory entry without actually freeing any blocks in the file by passing a dummy track/sector table, where the first byte (track number) is $00 signifying the end of the table:

```
;Pass:        r0   pointer to filename
DeleteDirEntry:
     LoadW    r3,#NullTrScTable          ;pass dummy table
     jmp      FastDelFile                ;delete dir entry
;    never get here -- FastDel returns to caller
```

This will also work correctly with a VLIR file.

For freeing (deleting) all the blocks in a file without removing the directory entry, refer to **FreeFile**.

Apple: Apple GEOS has no **FastDelFile** equivalent because the ProDOS blocks are not linked with internal forward pointers. Use **DeleteFile.**

Example:

```
;*******************************************************
;Read sequential file into memory and then delete it
;from disk
;
;Pass:          r6    pointer to filename
;               r7    where to put data
;               r2    size of buffer (max size of file)
;
;Returns:
;     x               error code
;
;Destroys:
;     a, y, r0-r9
;
;Implementation:
;     Call Findfile to get the directory entry of the
;     file to load/delete. We pass the directory entry
;     to GetFHdrInfo to get the GEOS header block. We
;     check the header to ensure we're not trying to
;     read in a VLIR file. After GetFHdrInfo, the
;     parameters are already set up correctly to call
;     ReadFile (fileTrScTab+0,fileTrScTab+1 contains
;     header block and r1 contains first data block).
;     Readfile reads in the file's blocks, building out
;     the remainder of the fileTrScTab, which we pass
;     to FastDelFile to free all blocks in the file
;     (including the file header block, which is the
;     first entry in the table).
;*******************************************************
ReadAndDelete:
      MoveW     r6,r0                ;save pointer for FastDel
      jsr       FindFile             ;find file on disk
      txa                            ;set status flags
      bne       96$                  ;branch on error
      LoadW     r9,dirEntryBuf       ;get directory entry
      jsr       GetFHdrInfo          ;get GEOS file header
      txa                            ;set status flags
      bne       96$                  ;branch on error
      lda       fileHeader+OFF_GSTRUCT_TYPE
      cmp       #VLIR                ;check filetype
      bne       10$                  ;branch if not VLIR
      ldx       #STRUCT_MISMAT       ;can't load VLIR
      bne       96$                  ;branch always for error
10$:
      jsr       ReadFile             ;read in file
      txa                            ;else set status flags
      bne       96$                  ;branch on other error
20$:
      LoadW     r3,#fileTrScTab      ;track/sector tbl
      jsr       FastDelFile          ;file read OK, delete it!
96$
      rts                            ;error in x
```

See also: **FreeFile, DeleteFile.**

FdFTypesInDir
~~FastDelFile~~

FdFTypesInDir (Apple) — high-level disk

Function: Special Apple GEOS version of **FindFTypes** that will build the filename list using a directory other than the current directory.

Parameters:

r6	BUFFER — pointer to buffer for building-out file list; allow 16 bytes for each entry in the list (word).
r10	PERMNAME — pointer to permanent name string to match or $0000 to ignore permanent name string (word).
r7H	MAXFILES — maximum number of filenames to return, usually used to prevent overwriting end of *BUFFER* (must be less than 127). If **INAUX_B** is bitwise-or'ed into this parameter, *BUFFER* will be treated as an address in auxiliary memory.
r7L	FILETYPE — GEOS file type to search for or **WILDCARD**, which will match all types except subdirectories (byte).
r1	DIRBLKNO — block number of directory to search (word).

Uses: **curDrive**

Returns:

x	error ($00 = no error).
r7H	decremented once for each file name (Apple GEOS: high-bit is always cleared).

Alters: **diskBlkBuf** used as temporary buffer for directory blocks.

Destroys: **a, y, r0–r2L, r4–r6.**

Description: **FdFTypesInDir** performs the same action as **FindFTypes**, except that *DIRBLKNO* temporarily becomes the current directory. The current directory is restored before returning.

The data area at *BUFFER*, where the list is built-out, must be large enough to accomodate *MAXFILES* filenames of 16 bytes each.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **FindFTypes, FndFilInDir.**

FetchRAM (C64 v1.3 & C128) memory

Function: Primitive for transferring data from an REU.

Parameters:
- **r0** CBMDST — address in Commodore to put data (word).
- **r1** REUSRC — address in REU bank to start reading (word).
- **r2** COUNT — number of bytes to fetch (word).
- **r3L** REUBANK — REU bank number to fetch from (byte).

Returns:
- **r0–r3** unchanged.
- **x** error code: $00 (no error) or **DEV_NOT_FOUND** if *REUBANK* or REU not available.
- **a** REU status byte and'ed with $60 ($40 = successful fetch).

Destroys: y

Description: **FetchRAM** moves a block of data from an REU bank into Commodore memory. This routine is a "use at your own risk" low-level GEOS primitive

FetchRAM uses the **DoRAMOp** primitive by calling it with a *CMD* parameter of %10010001.

Note: Refer to **DoRAMOp** for notes and warnings.

Example:

See also: **StashRAM, SwapRAM, VerifyRAM, DoRAMOp, MoveBData.**

FillRam, i_FillRam (Apple, C64, C128) memory

Function: Fills a region of memory with a repeating byte value.

Parameters: Normal:

r0	COUNT — number of bytes to clear (0 - 64K) (word).
r1	ADDR —address of area to clear (word).
r2L	FILL — byte value to fill with (byte).

Inline:

.word	COUNT — number of bytes to clear (0 - 64K) (word).
.word	ADDR —address of area to clear (word).
.byte	FILL — byte value to fill with (byte).

Returns: **r2L** unchanged.

Destroys: **a, y, r0, r1**

Description: **FillRam** fills COUNT bytes starting at ADDR with the FILL byte. This routine is useful for initializing a block of memory to some non-zero or variable value (for filling a region with $00, use **ClearRam**).

Note: Do not use **FillRam** to initialize **r0–r2L**.

Example:

See also: **ClearRam, InitRam.**

FindBAMBit (C64, C128) mid-level disk

Function: Get disk block allocation status.

Parameters:
r6L TRACK —track number of block (byte).
r6H SECTOR — sector number of block (byte).

Uses:
curDrive
curDirHead this buffer must contain the current directory header.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:
st z flag reflects allocation status (1 = free ; 0 = allocated).
r6 unchanged.

1541 drives only:
x offset from **curDirHead** for BAM byte.
r8H mask for isolating BAM bit.
a BAM byte masked with **r8H**.
r7H offset from **curDirHead** of byte that holds free blocks on track total.

Destroys:
non-1541 drives:
a, y, r7H, r8H.

1541 drives:
y (**a**, **r7H**, and **r8H** all contain useful values).

Description: **FindBAMBit** accesses the BAM of the current disk (in **curDirHead**) and returns the allocation status of a particular block. If the BAM bit is zero, then the block is in-use; if the BAM bit is one, then the block is free. **FindBAMBit** returns with the **z** flag set to reflect the status of the BAM so that a subsequent **bne** or **beq** branch instructions can test the status of a block after calling **FindBAMBit**:

```
bne     BlockIsFree         ;branch if block is free
      - or -
beq     BlockInUse          ;branch if block is in-use
```

Note: **FindBAMBit** will return the allocation status of a block on any disk device, even those with large or multiple BAMs (such as the 1571 and 1581 disk drives). Only the 1541 driver, however, will return useful information in **a**, **y**, **r7H**, and **r8H**. For an example of using these extra 1541 return values, refer to **AllocateBlock.**

Apple: See **FindVBMBit.**

Example:

```
LoadB   r6L,#TRACK          ; get track and sector number
LoadB   r6H,#SECTOR
jsr     FindBAMBit          ; get allocation status
beq     BlockInUse          ; branch if already in use
```

See also: **FindVBMBit, AllocateBlock, FreeBlock, GetDirHead, PutDirHead.**

FindFile (Apple, C64, C128) high-level disk

Function: Search for a particular file in the current directory.

Parameters:
- **r6** FILENAME — pointer to null-terminated name of file of a maximum of **ENTRY_SIZE** bytes (not counting null terminator). (Apple GEOS: must be in main memory.) (word).

Uses: **curDrive**

Commodore:
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
- **curKBlkno** current directory.

Returns:
- **x** error ($00 = no error).
- **r1** block number (Commodore track/sector) of directory block containing entry.
- **r5** pointer into **diskBlkBuf** to start of directory entry.

Alters:
- **diskBlkBuf** contains directory block where *FILENAME* found.
- **dirEntryBuf** directory entry of file if found.

Destroys: **a, y, r4, r6.**

Description: Given a null-terminated filename, **FindFile** searches through the current directory and returns the directory entry in **dirEntryBuf.** If the file specified with *FILENAME* is not found, a **FILE_NOT_FOUND** error is returned.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk. The directory entry is deleted by setting its **OFF_CFILE_TYPE** byte to $00.

Apple: Only the current directory, the directory specified by **curKBlkno,** is searched. To search a directory other than the current one, use **FndFilInDir.**

Example:

See also: **FndFilInDir, Get1stDirEntry, GetNxtDirEntry, FindFTypes.**

FindFTypes (Apple, C64, C128) — high-level disk

Function: Builds a list of files of a particular GEOS type from the current directory.

Parameters:

r6 BUFFER — pointer to buffer for building-out file list; allow ENTRY_SIZE+1 bytes for each entry in the list (word).

r10 PERMNAME — pointer to permanent name string to match or $0000 to ignore permanent name string (word).

Commodore:

r7H MAXFILES — maximum number of filenames to return, usually used to prevent overwriting buffer.

r7L FILETYPE — GEOS file type to search for (byte).

Apple:

r7H MAXFILES — maximum number of filenames to return, usually used to prevent overwriting end of *BUFFER* (must be less than 127). If **INAUX_B** is bitwise-or'ed into this parameter, *BUFFER* will be treated as an address in auxiliary memory.

r7L FILETYPE — GEOS file type to search for or **WILDCARD**, which will match all types except subdirectories (byte).

Uses: **curDrive**

Commodore:

curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:

curKBlkno current directory.

Returns:

x error ($00 = no error).

r7H decremented once for each file name (Apple GEOS: high-bit is always cleared).

Alters: **diskBlkBuf** used as temporary buffer for directory blocks.

Destroys:

Commodore:
a, y, r0–r2L, r4, r6.

Apple:
a, y, r0–r2L, r4–r6.

Description: **FindFTypes** build a list of files that match a particular GEOS file type and, optionally, a specific permanent name string.

The data area at *BUFFER*, where the list is built-out, must be large enough to accomodate *MAXFILES* filenames of **ENTRY_SIZE**+1 bytes each.

FindFTypes first clears enough of the area at *BUFFER* to hold *MAXFILES* filenames then calls **GetlstDirEntry** and **GetNxtDirEntry** to go through each directory entry in the current directory. When the GEOS file type of a directory

entry matches the *FILETYPE* parameter, **FindFTypes** goes on to check for a matching permanent name string.

If the *PERMNAME* parameter is $0000, then this check is bypassed and the filename is added to the list. If the *PERMNAME* parameter is non-zero, the null-terminated string it points to is checked, character-by-character, against the permanent name string in the file's header block. Although the permanent name string in the GEOS file header is 16 characters long, the comparison only extends to the character before the null-terminator in the string at *PERMNAME*.

Since permanent name strings typically end with V*x.x*, where *x.x* is a version number (e.g., 2.1), a shorter string can be passed so that the specific version number is ignored. For example, a program called geoQuiz version 1.3 might use "geoQuiz V1.3" as the permanent name string it gives its data files. When geoQuiz version 3.0 goes searching for its data files, it can pass a *PERMNAME* string of "geoQuiz V" so data files for all versions of the program will be added to the list.

When a match is found, the filename is copied into the list at *BUFFER*. The filenames are placed in the buffer as they are found (the same order they appear on the pages of the deskTop notepad). With a small buffer, matching files on higher-numbered pages may never get added to the list.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk. The filenames appear in the list null-terminated even though they are padded with $a0 in the directory.

Apple: Only the current directory specified by **curKBIkno** is searched. To search a directory other than the current one, use **FdFTypesInDir**. The filenames in the list are null-terminated.

To search for subdirectories in the current directory, pass **PRO_DIR** as the *FILETYPE*. When searching for directories, the permanent string check is bypassed. To match all file types except subdirectories, pass **WILDCARD** as the *FILETYPE* and $0000 for the *PERMSTRING* parameter.

A filename list created with **FindFTypes** can be sorted alphabetically with a call to **SortAlpha**.

Example:

See also: **FdFTypesInDir, FindFile, Get1stDirEntry, GetNxtDirEntry.**

FindVBMBit (Apple) mid-level disk

Function: Get disk block allocation status.

Parameters: **r6** BLOCK — block number (word).

Uses:

VBMBlkno†	first VBM block starts here; set initially by **OpenDisk**
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).
st	z flag reflects allocation status (1 = free; 0 = allocated).
r8H	mask for isolating VBM bit.
r7	offset to BAM byte in current VBM block.
a	BAM byte masked with **r8H**.
r6	unchanged.

Alters:

curVBlkno†	Block that contains the VBM bit for *BLOCK*.
VBMchanged†	changed to **FALSE** if VBM cache flushed.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **y**

Description: **FindVBMBit** accesses the VBM of the current disk and returns the allocation status of a particular block. If the VBM bit is zero, then the block is in-use; if the VBM bit is one, then the block is free. **FindVBMBit** returns with the **z** flag set to reflect the status of the BAM bit so that a subsequent **bne** or **beq** branch instructions can test the status of a block after calling **FindVBMBit**:

```
bne     BlockIsFree         ;branch if block is free
        - or -
beq     BlockInUse          ;branch if block is in-use
```

C64 &C128: See **FindBAMBit**.

Example:

```
LoadW   r6,#BLOCK           ; get block number
jsr     FindVBMBit          ; get allocation status
cpx     #NO_ERROR           ; check for error
bne     diskError           ; branch on error
tay                         ; check block status
beq     BlockInUse          ; branch if block in use
```

See also: **FindBAMBit, AllocateBlock, FreeBlock, GetDirHead, PutDirHead.**

FirstInit (Apple, C64, C128) internal

Function: Simulates portions of the GEOS coldstart procedure without actually rebooting GEOS or destroying the application in memory.

Parameters: none

Returns: GEOS variables and system hardware in a coldstart state; stack and application space unaffected.

Destroys: **a, x, y, r0–r2**

Description: **FirstInit** is part of the GEOS coldstart procedure. It initializes *nearly all* GEOS variables and data structures (both global and local), including those which are usually only done once, when GEOS is first booted, such as setting the configuration variables to a default, power-up state.

GEOS calls this routine internally. Applications will not find it especially useful.

Note: The GEOS font variables are not reset by **FirstInit**; a call to **UseSystemFont** may be necessary.

See also: **WarmStart, StartAppl.**

FndFilInDir (Apple) — high-level disk

Function: Special Apple GEOS version of **FindFile** that will search for a file in a directory other than the current directory.

Parameters:
- **r6** FILENAME — pointer to null-terminated name of file (word).
- **r1** DIRBLKNO — block number of directory to search (word).

Uses: **curDrive**

Returns:
- **x** error ($00 = no error).
- **r1** block number of directory block containing entry.
- **r5** pointer into **diskBlkBuf** to start of directory entry.

Alters:
- **diskBlkBuf** contains directory block where *FILENAME* found.
- **dirEntryBuf** directory entry of file if found.

Destroys: **a, y, r4–r6.**

Description: **FndFilInDir** performs the same action as **FindFile**, except that *DIRBLKNO* temporarily becomes the current directory. The current directory is restored before returning. **FndFilInDir** returns the directory entry in **dirEntryBuf**. If the file specified with *FILENAME* is not found in the *DIRBLKNO* directory, a **FILE_NOT_FOUND** error is returned.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **FindFile, FdFTypesInDir, GoDirectory.**

FollowChain (C64, C128) mid-level disk

Function: Follow a chain of Commodore disk blocks, building out a track/sector table.

Parameters:
- **r1L** START_TRACK —track number of starting block (byte).
- **r1H** START_SEC — sector number of starting block (byte).
- **r3** TSTABLE — pointer to buffer for building out track and sector table of chain, usually points to **fileTrScTab** (word).

Uses:
- **curDrive.**
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns:
- **x** error ($00 = no error).
- **r3** unchanged.

track/sector built-out in buffer pointed to by *TSTABLE.*

Alters: **diskBlkBuf** used for temporary block storage.

Destroys: **a, y, r1, r4.**

Description: **FollowChain** constructs a track/sector table for a list of chained blocks on the disk. It starts with the block passed in *START_TR* and *START_SC* and follows the links until it encounters the last block in the chain. Each block (including the first block at *START_TR, START_SC*) becomes a part of the track/sector table.

Commodore disk blocks are linked together with track/sector pointers. The first two bytes of each block represent a track/sector pointer to the next block in the chain. Each sequential file and VLIR record on the disk is actually a chained list of blocks. **FollowChain** follows these track/sector links, adding each to the list at *TSTABLE* until it encounters a track pointer of $00, which terminates the chain. **FollowChain** adds this last track pointer ($00) and its corresponding sector pointer (which is actually an index to the last valid byte in the block) to the track/sector table and returns to the caller.

FollowChain builds a standard track/sector table compatible with routines such as **WriteFile** and **FastDelFile.**

Apple: Apple GEOS has no **FollowChain** equivalent because ProDOS links blocks together, not by pointers within each block, but by a list of blocks in an index associated with each sequential file and VLIR record. This index is called the *index block* and, as its name implies, occupies a single block on the disk.

Example:

```
LoadB   r1L,#START_TR       ; start track
LoadB   r1H,#START_SC       ; and sector
LoadW   r3,#fileTrScTab     ;buffer for table
jsr     FollowChain         ; create tr/sc table
txa                         ; set status flags
bne     HandleError         ; branch if error
```

See also:

FRAME RECTANGLE
~~FollowChain~~

FrameRectangle, i_FrameRectangle (Apple, C64, C128) graphics

Function: Draw a rectangular frame (one-pixel thickness).

Parameters: Normal:

a	eight-bit line pattern.
r3	X1 — x-coordinate of upper-left (word).
r2L	Y1 — y-coordinate of upper-left (byte).
r4	X2 — x-coordinate of lower-right (word).
r2H	Y2 — y-coordinate of lower-right (byte).

where (X1,Y1) is the upper-left corner of the frame and (X2,Y2) is the lower-right corner.

Inline:
data appears immediately after the **jsr i_FrameRectangle**

.byte	Y1	y-coordinate of upper-left.
.byte	Y2	y-coordinate of lower-right.
.word	X1	x-coordinate of upper-left.
.word	X2	x-coordinate of lower-right.
.byte	PATTERN	eight-bit line pattern.

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: **r2**, **r3**, and **r4** unchanged.

Destroys: Commodore
a, x, y, r5–r9, r11

Apple
a, x, y, r11

Description: **FrameRectangle** draws a one-pixel rectangular frame on the screen as determined by the coordinates of the upper-left and lower-right corners. The horizontal and vertical lines which comprise the frame are drawn with the specified line pattern.

FrameRectangle operates by calling **HorizontalLine** and **VerticalLine** with the desired line-pattern. As with these two routines, the line pattern is drawn as if aligned on an eight-pixel boundary. The values of the corner pixels will be determined by the vertical sides because they are drawn after the horizontal sides.

Note: Because all GEOS coordinates are inclusive, framing a filled rectangle requires either calling **FrameRectangle** *after* calling **Rectangle** (and thereby overwriting the perimeter of the filled area) or calling **FrameRectangle** with (*X1*-1,*Y1*–1) and (*X2*+1,Y2+1) as the corner points.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will

automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **Rectangle, ImprintRectangle, RecoverRectangle, InvertRectangle.**

FreeBlock
~~FrameRectangle~~

FreeBlock (Apple, C64, C128) — mid-level disk

Function: Free an allocated disk block.

Parameters:

Commodore:
r6L track number of block to free (byte).
r6H sector number of block to free (byte).

Apple:
r6 block to free (word).

Uses: **curDrive**

Commodore:
curDirHead this buffer must contain the current directory header.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

Apple:
curVBlkno† used by VBM cacheing routines.
VBMchanged† used by VBM cacheing routines.
numVBMBlks† used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error); returns **BAD_BAM** if block already free.
r6L, r6H unchanged.

Alters:

Commodore:
curDirHead BAM updated to reflect newly allocated block.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

Apple:
curVBlkno† used by VBM cacheing routines.
VBMchanged† set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r7, r8H.**

Description: **FreeBlock** tries to free (deallocate) the block number passed in **r6**. If the block is already free, then **FreeBlock** returns a **BAD_BAM** error.

C64 & C128: **FreeBlock** was not added to the Commodore GEOS jump table until v1.3, but it can be accessed directly under GEOS v1.2. The following routine will check the GEOS version number and act correctly under GEOS v1.2 and later.

```
;*******************************************************
;
;    MyFreeBlock -- allocate specific block in BAM
;    with any CBM device driver.
```

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

MyFreeBlock:
     lda      version             ;check GEOS version number
     cmp      #$12                ;version 1.2?
     bne      10$                 ;if not, go through jump table

                                  ; r6L = track #
                                  ; r6h = sector #
     jsr      FindBAMbit          ; Get mask for BAM byte in r8H
                                  ;      offset to track in r7h
                                  ;      offset into bam in X
                                  ;      masked value A
     bne      88$                 ; if 1, then not allocated, give error
     lda      r8h                 ; get mask
     eor      curDirHead,x        ; flip BAM bit to show available
     sta      curDirHead,x        ;
     ldx      r7H                 ; one more free block
     inc      curDirHead,x        ;
     ldx      #0                  ; NO ERROR
     rts
88$:
     ldx      #BAD_BAM            ; BAM ERROR
     rts
10$:
     jmp      FreeBlock           ;v1.3+: go thru jump tbl
```

FreeBlock does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

Apple: **FreeBlock** does not automatically flush the VBM cache. See **PutVBM** for more information on flushing the VBM cache.

Example:

See also: **FreeFile, AllocateBlock.**

FreeDir (Apple) — mid-level disk

Function: Free the chained directory blocks associated with a subdirectory. Does not delete the subdirectory's directory entry. The subdirectory must be empty.

Parameters:

r9	DIRENTRY — pointer to directory entry of subdirectory being freed, usually points to **dirEntryBuf** (must be in main memory.) (word).	

Uses:

curDrive	
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).

Alters:

diskBlkBuf	used for temporary block storage.
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **FALSE** by VBM cacheing routines to indicate cached VBM block has already been flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4, r6, r7, r8H.**

Description: Given a valid subdirectory entry, **FreeDir** will free all the directory blocks allocated to the subdirectory. The subdirectory's directory entry, however, is left intact.

The subdirectory entry is a standard data structure returned by routines such as **FindFile, Get1stDirEntry** and **GetNextDirEntry.** A subdirectory is flagged by a **PRO_DIR** in the GEOS type byte in the directory entry (at **OFF_FTYPE**). **FreeDir** is called automatically by **DeleteDir.**

FreeDir will not free the blocks of subdirectory that has files in it. A **DIR_NOT_EMPTY** error will be returned. (The number of files in the subdirectory is stored in the **OFFB_FLCNT** word of the directory's header. This maintains this value by calling **UpdateParent** when files are added or removed.)

FreeDir flushes the VBM cache.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **DeleteDir, DeleteFile, FreeFile, FreeBlock.**

FreeFile (Apple, C64, C128) — mid-level disk

Function: Free *all* the blocks in a GEOS file (sequential or VLIR) without deleting the directory entry. The GEOS file header and any index blocks are also deleted.

Parameters:

r9	DIRENTRY — pointer to directory entry of file being freed, usually points to **dirEntryBuf** (Apple GEOS: must be in main memory.) (word).

Uses: **curDrive**

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).

Alters:

diskBlkBuf	used for temporary block storage.

Commodore:

curDirHead	BAM updated to reflect newly freed blocks.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
fileHeader	temporary storage of the index table when deleting a VLIR file.

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **FALSE** by VBM cacheing routines to indicate cached VBM block has already been flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Commodore:
a, y, r0–r9.

Apple:
a, y, r1, r2L, r4, r6, r7, r8H, r9.

Description: Given a valid directory entry, **FreeFile** will delete (free) all blocks assocatied with the file. The GEOS file header and any index blocks associated with the file are also be freed. The directory entry on the disk, however, is left intact.

The directory entry is a standard GEOS data structure returned by routines such as **FindFile**, **Get1stDirEntry** and **GetNxtDirEntry**. **FreeFile** is called automatically by **DeleteFile**.

C64 & C128: **FreeFile** calls **GetDirHead** to get the current directory header and BAM into memory. It then checks at **OFF_GHDR_PTR** in the directory entry for a GEOS file header block, which it then frees.

If the file is a sequential file, **FreeFile** walks the chain pointed at by the **OFF_DE_TR_SC** track/sector pointer in the directory header and frees all the blocks in the chain. **FreeFile** then calls **PutDirHead** to write out the new BAM.

If the file is a VLIR file, the index table (the block pointed to by **OFF_INDEX_PTR**) is first read into **fileHeader** then marked as free in the BAM. **FreeFile** then goes through each record. If the record has data in it, **FreeFile** walks through the chain, freeing all the blocks in the record. **FreeFile** finishes by calling **PutDirHead** to write out the new BAM.

When using **Get1stDirEntry** and **GetNxtDirEntry**, do not pass **FreeFile** a pointer into **diskBlkBuf**. Copy the full directory entry (**DIRENTRY_SIZE** bytes) from **diskBlkBuf** to another buffer (such as **dirEntryBuf**) and pass **FreeFile** the pointer to that buffer. Otherwise when **FreeFile** uses **diskBlkBuf** it will corrupt the directory entry.

Because **FreeFile** deletes a block at a time as it follows the chains, it is capable of deleting files with chains larger than 127 blocks, which is the standard GEOS limit imposed by the size of **TrScTable**.

Apple: **FreeFile** first copies the entire directory entry (**ENTRY_SIZE** bytes) from *DIRENTRY* (in main memory) to an internal buffer in auxiliary memory.

If the file is a VLIR file, then **FreeFile** reads the VLIR index block (the ProDOS master index block) into memory and frees all the data blocks in each record. The VLIR index block and all the individual record index blocks are then deleted. **FreeFile** finishes by calling **PutVBM** to flush the VBM cache.

If the file is a sequential file, **FreeFile** reads the sequential index block into memory, frees all assoiciated data blocks,.then frees the index block itself. **FreeFile** finishes by calling **PutVBM** to flush the VBM cache.

FreeFile cannot free a subdirectory. To free a subdirectory, use **FreeDir**.

Example:

See also: **DeleteFile, FreeDir, FreeBlock.**

FreezeProcess (Apple, C64, C128) process

Function: Freeze a process's countdown timer at its current value.

Parameters: x PROCESS — number of process (0 – *n*–1, where *n* is the number of processes in the table) (byte).

Returns: x unchanged.

Destroys: a

Description: **FreezeProcess** halts a process's countdown timer so that it is no longer decremented every vblank. Because a frozen timer will never reach zero, the process will not become runable except through a call to **EnableProcess**. When a process is unfrozen with **UnFreezeProcess**, its timer again begins counting from the point where it was frozen.

Note: If a process is already frozen, a redundant call to **FreezeProcess** will have no effect.

Example:

See also: **UnfreezeProcess, BlockProcess.**

Get1stDirEntry (Apple, C64, C128) — mid-level disk

Function: Loads in the first directory block of the current directory and returns a pointer to the first directory entry within this block.

Parameters: none.

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
curKBlkno current directory.

Returns:
x error ($00 = no error).
r5 pointer to first directory entry within **diskBlkBuf**.

Alters: **diskBlkBuf** directory block.

Destroys: **a, y, r1, r4.**

Description: **Get1stDirEntry** reads in the first directory block of the current directory and returns with **r5** pointing to the first directory entry. **Get1stDirEntry** is called by routines like **FindFTypes** and **FindFile.**

To get a pointer to subsequent directory entries, call **GetNxtDirEntry.**

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk.

Get1stDirEntry did not appear in the jump table until version 1.3. An application running under version 1.2 can access **Get1stDirEntry** by calling directly into the Kernal. The following subroutine will work on Commodore GEOS v1.2 and later:

```
;*******************************************************
;
;    MyGet1stDirEntry -- Call instead of Get1stDirEntry
;    to work on GEOS v1.2 and later
;
;*******************************************************

;EQUATE: v1.2 entry point directly into Kernal. Must
;do a version check before calling.
o_Get1stDirEntry    = $c9f7        ;exact entry point

MyGet1stDirEntry:
     lda      version              ;check version number
     cmp      #$13
     bge      10$                  ;branch if v1.3 or later
     jmp      o_Get1stDirEntry     ;direct call

10$:
     jmp      Get1stDirEntry       ;go through jump table
```

Apple: **Get1stDirEntry** did not appear in the jump table until version 2.0, release 3. An application running under an earlier version can add **Get1stDirEntry** into the jump table with the patch provided in Appendix @@.

Example:

See also: **GetNxtDirEntry, FindFTypes.**

GetBlock (Apple, C64, C128) — low-level disk

Function: General purpose routine to get a block from current disk.

Parameters:

r4 BUFFER — address of buffer to place block; must be at least **BLOCKSIZE** bytes (word).

Commodore:
r1L TRACK — valid track number (byte).
r1H SECTOR — valid sector on track (byte).

Apple:
r1 BLOCK — ProDOS block number (word).

Uses:

curDrive currently active disk drive.

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank bank *BUFFER* is in (**MAIN** or **AUX**).

Returns:

x error ($00 = no error).
r1, r4 unchanged

Destroys: **a, y.**

Description: **GetBlock** reads a block from the disk into *BUFFER*. **GetBlock** is useful for implementing disk utility programs and new file structures.

C64 & C128: **GetBlock** is a higher-level version of **ReadBlock**. It calls **InitForIO**, **EnterTurbo**, **ReadBlock**, and **DoneWithIO**. If an application needs to read many blocks at once, **ReadBlock** may offer a faster solution. If the disk is shadowed, **GetBlock** will read from the shadow memory, resulting in a faster transfer.

The Commodore 1581 driver has a bug that causes its **GetBlock** to trash **r1** and **r4**.

Apple: **GetBlock** provides the lowest-level block access to a ProDOS compatible device. It uses the ProDOS device driver **READ** block command directly. Apple GEOS, for this reason, does not have a **ReadBlock** equivalent.

Example:

See also: **PutBlock, ReadBlock.**

GetCharWidth (Apple, C64, C128) text

Function: Calculate the pixel width of a character as it exists in the font (in its plaintext form). Ignores any style attributes.

Parameters: **a** CHAR — character code of character (byte).

Uses: **curIndexTable**

Returns: **a** character width in pixels.

Destroys: **y**

Description: **GetCharWidth** calculates the width of the character before any style attributes are applied. If the character code is less than 32, $00 is returned. Any other character code returns the pixel width as calculated from the font data structure.

Because **GetCharWidth** does not account for style attributes, it is useful for establishing the number of bits a character occupies in the font data structure.

Example:

See also: **GetRealSize.**

GetDimensions (C64, C128) printer driver

Function: Get printer resolution.

Parameters: none.

Returns:

- **a** $00 = printer has graphics and text modes; $ff = printer only has text modes (e.g., daisy wheel printers).
- **x** PGWIDTH — page width in cards: number of 8x8 cards that will fit horizontally on a page (1–80, standard value is 80 but some printers only handle 60, 72, or 75).
- **y** PGHEIGHT — page height in cards: number of 8x8 cards that will fit vertically on a page (1–255, usually 94).

The width and height return values are typically based on an 8.5" x 11" page with a 0.25" margin on all sides, leaving an 8" x 10.5" usable print area.

Destroys: nothing.

Description: **GetDimensions** returns the printable page size in cards. At each call to **PrintBuffer**, the printer driver will expect at least*PGWIDTH* cards of graphic data in the 640-byte print buffer. To print an entire page , the application will need to call **PrintBuffer** *PGHEIGHT* times.

Most dot-matrix printers have a horizontal resolution of 80 dots-per-inch and an eight inch print width. Eight inches at 80 dpi gives 640 addressable dots per printed line, and 640/8 equals 80 cards per line. GEOS assumes an 80 dpi output device.

Drivers for printers with a different horizontal resolution will usually return a *PGWIDTH* value that reflects some even multiple of the dpi. For example, a lower resolution 72 dpi printer can only fit 72*8 = 560 dots per line, and 560 dots reduces to 72 cards. *PGWIDTH* in this case would come back as 72.

A 300 dpi laser printer, however, can accomodate 2,400 dots on an eight inch line. To scale 80 dpi data to 300 dpi, each pixel is expanded to four times its normal width. If the printer driver tried to print the full 640 possible dots at this expanded width, it would lose the last 160 dots because the printer itself can only handle 2,400 dots in an eight inch space and 640*4 = 2,560. To alleviate this problem he printer driver truncates the width at the card boundary nearest to 2,400 dots, which happens to be 75 cards. Hence, in this case, *PGWIDTH* would come back as 75.

The size, *PGHEIGHT*, reflects the number of card rows to send through **PrintBuffer** to fill a full-page. If more rows are sent, then (depending on the printer and the driver) the printing will usually continue onto the next page (printing over the perforation on z-fold paper). The application will usually keep an internal card-row counter and call **StopPrint** to advance to the next page.

Note: It is not necessary to call **GetDimensions** when printing ASCII text. Commodore GEOS printer drivers always assume 80 columns by 66 lines.

Apple: This routine is not supported under Apple GEOS. Apple GEOS offers a more sophisticated printer driver scheme. Refer to **GetMode** for more information.

See also: **StartPrint, StartASCII.**

GetDirHead (Apple, C64, C128) mid-level disk

Function: Read directory header from disk. Commodore GEOS also reads in the BAM.

Parameters: none.

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
curKBlkno current directory key block.

Returns: **x** error ($00 = no error).

Commodore:
r4 pointer to **curDirHead.**

Alters: **curDirHead** contains directory header (39 bytes on Apple).

Commodore:
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: Commodore:
a, y, r1.

Apple:
a, y, r1, r4.

Description: **GetDirHead** reads the directory header into the buffer at **curDirHead.** Because of differences in the Commodore and Apple file systems, this can mean different things. Commodore GEOS places the full directory header block (256 bytes) into **curDirHead.** This block also includes the BAM (block allocation map) for the entire disk. Apple GEOS, on the other hand, only places the 39-byte ProDOS directory header for the current directory into **curDirHead.**

C64 & C128: GEOS disks, like the standard Commodore disks upon which they are based, have one directory header. The directory header occupies one full block on the disk. The Commodore directory header contains information about the disk, such as the location of the directory blocks, the disk name, and the GEOS version string (if a GEOS disk). The Commodore directory header also contains the disk BAM, which flags particular sectors as used or unused.

GetDirHead calls **GetBlock** to read in the directory header block into the buffer at **curDirHead.** The directory header block contains the directory header and the disk BAM (block allocation map). Typically, applications don't call **GetDirHead** because the most up-to-date directory header is almost always in memory (at **curDirHead**), **OpenDisk** calls **GetDirHead** to get it there

initially. Other GEOS routines update it in memory, some calling **PutDirHead** to bring the disk version up to date.

Because Commodore disks store the BAM information in the directory header it is important that the BAM in memory not get overwritten by an outdated BAM on the disk. An application that manipulates the BAM in memory (or calls GEOS routines that do so), must be careful to write the BAM back out (with **PutDirHead**) before calling any other routine that might overwrite the copy in memory. **GetDirHead** is called by routines such as **OpenDisk**, **SetGEOSDisk**, and **OpenRecordFile**, etc.

Apple: Apple GEOS disks, like the ProDOS disks upon which they are based, have a directory header for each directory. The header for the root directory is called a *volume directory header* and the header for a subdirectory is called a *subdirectory header*. These directory headers are 39-byte structures defined by ProDOS. They contain such information as the directory name, the date stamp, and read/write access flags. The directory header *does not* contain the VBM (volume bit map, the Apple equivalent of a BAM).

GetDirHead reads in the key block of the current directory (pointed at by **curKBlkno**) and copies the 39-byte directory header information to **curDirHead**.

Since Apple GEOS does not store the allocation map (VBM) in the directory header like Commodore GEOS, it not necessary to be as careful about rereading the directory header. For more information on reading the Apple VBM, refer to **GetVBM**.

Example:

See also: **PutDirHead, GetVBM, PutVBM.**

GetFHdrInfo (Apple, C64, C128) mid-level disk

Function: Loads the GEOS file header for a particular directory entry.

Parameters:
r9 DIRENTRY — pointer to directory entry of file, usually points to **dirEntryBuf** (Apple GEOS: must be in main memory) (word).

Uses:
curDrive

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns:
x error ($00 = no error).
r7 load address copied from the **O_GHST_ADDR** word of the GEOS file header.

Commodore:
r1 track/sector copied from bytes +1 and +2 of the directory entry (*DIRENTRY*). This is the track/sector of the first data block of a sequential file (**OFF_DE_TR_SC**) or the index table block of a VLIR file (**OFF_INDEX_PTR**).

Apple:
r1 block number of sequential-file index block or VLIR master index block as copied from the **OFF_FINDX** byte of the directory entry.

Alters:
fileHeader contains 256-byte GEOS file header.

Commodore:
fileTrScTab track/sector of header added to first two bytes of this table; a subsequent call to **ReadFile** or similar routine will augment this table beginning with the third byte (**fileTrScTab+2**) so as not to disrupt this value.

Destroys:
a, y, r4.

Description:
Given a valid directory entry, **GetFHdrInfo** will load the GEOS file header into the buffer at **fileHeader.**

The directory entry is a standard GEOS data structure returned by routines such as **FindFile, GetlstDirEntry** and **GetNextDirEntry. GetFHdrInfo** is called by routines such as **LdFile** just prior to calling **ReadFile** (to load in a sequential file or record zero of a VLIR).

GetFHdrInfo gets the block number (Commodore track/sector) of the GEOS file header by looking at the **OFF_GHDR_PTR** word in the directory entry.

Example:

See also:

GetFile (Apple, C64, C128) — high-level disk

Function: General-purpose file routine that can load an application, desk accessory, or data file.

Parameters: **r6** FILENAME — pointer to null-terminated filename (word).

When loading an application:
r0L LOAD_OPT:
- bit 0: 0 load at address specified in file header; application will be started autmatically
  1 load at address in **r7**; application will not be started automatically.
- bit 7: 0 not passing a data file.
  1 **r2** and **r3** contain pointers to disk and data file names.
- bit 6: 0 not printing data file.
  printing data file; application should print file and exit.

r7 LOAD_ADDR — optional load address. only used if bit 0 of *LOAD_OPT* is set (word).
r2 DATA_DISK — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the disk that contains the data file, usually a pointer to one of the **Drx CurDkNm** buffers (word).
r3 DATA_FILE — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the data file (word).

When loading a desk accessory:
r10L RECVR_OPTS — no longer used; set to $00 (see below for explanation (byte).

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank destination bank (**MAIN** or **AUX**); always set to **MAIN** when loading an application or desk accessory.

Returns: When loading an application:
only returns if alternate load address or disk error.
x error ($00 = no error).
r0, r2, r3, and **r7** unchanged.

When loading a desk accessory:
returns when desk accessory exits with a call to ***RstrAppl.***
x error ($00 = no error).

When loading a data file:
x error ($00 = no error).

Passes: When loading an application:
warmstarts GEOS and passes the following to the application:

r0 as originally passed to **GetFile.**
r2 as originally passed to **GetFile** (use **dataDiskName**).
r3 as originally passed to **GetFile.**(use **dataFileName**).
dataDiskName contains name of data disk if bit 7 of **r0** is set.
dataFileName contains name of data file if bit 6 of **r0** is set.

When loading a desk accessory:
warmstarts GEOS and passes the following:
r10L as originally passed to **GetFile.**

When loading a data file:
not applicable.

Alters: When loading an application:
GEOS brought to a warmstart state.

Destroys: **a, x, y, r0–r10** (only applies to loading a data file).

Description: **GetFile** is the preferred method of loading most GEOS files, whether a data file, application, or desk accessory. (The only exception to this is a VLIR file, which is better handled with the VLIR routines such as **OpenRecordFile** and **ReadRecord**). Most applications will use **GetFile** to load and execute desk accessories when the user clicks on an item in the geos menu. Some applications will use **GetFile** to load other applications. The GEOS deskTop, in fact, is just another application like any other. Depending on the user's choice of actions — open an application, open an application's data file, print an applications' data file — the deskTop sets *LOAD_OPT*, *DATA_DISK*, *DATA_FILE* appropriately and calls **GetFile.**

GetFile first calls.**FindFile** to locate the file at *FILENAME*, then checks the GEOS file type in the directory entry. If the file is type **DESK_ACC**, then **GetFile** calls **LdDeskAcc.** If the file is type **APPLICATION** or type **AUTO_EXEC**, **GetFile** calls **LdApplic.** All other file types are loaded with the generic **LdFile.**

The following GEOS constants can be used to set the *LOAD_OPT* parameter when loading an application:

ST_LD_AT_ADDR	Load at address: load application at the address passed in r7 as opposed to the address in the file header.
ST_LD_DATA	Load data file: application is being passed the name of a data file to load.
ST_PR_DATA	Print data file: application is being passed the name of a data file to print.

Note: The *RECVR_OPTS* flag used when loading desk accessories originally carried the following significance:

bit 7: 1 force desk accessory to save foreground screen area and restore it on return to application.
0 not necessary for desk accessory to save foreground.
Commodore only:
bit 6: 1 force desk accessory to save color memory and restore it on return to application.

0 not necessary for desk accessory to save color memory.

However, it was found that the extra code necessary to make desk accessories save the foreground screen and color memory provided no real benefit because this context save can just as easily be accomplished from within the application itself. The *RECVR_OPTS* flag is set to $00 by all Berkeley Softworks applications, and desk accessories can safely assume that this will always be the case. (In fact, future versions of GEOS may force **r10H** to $00 before calling desk accessories just to enforce this standard!)

The application should always set **r10H** to $00 and bear the burden of saving and restoring the foreground screen and the color memory. (Color memory only applicable to GEOS 64 and GEOS 128 in 40-column mode.)

Apple: Applications and desk accessories are designed to load into main memory. Ensure that **RWbank** contains **MAIN** before calling **GetFile** to load anything but a data file.

Example:

See also: **LdFile, LdDeskAcc, LdApplic.**

GetFreeDirBlk (Apple, C64, C128) — mid-level disk

Function: Search the current directory for an empty.slot for a new directory entry. Allocates another directory block if necessary.

Parameters:

r10L DIRPAGE — directory page to begin searching for free slot; each directory page holds eight files and corresponds to one notepad page on the GEOS deskTop. The first page is page one.

Uses:

curDrive

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	this buffer must contain the current directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
interleave†	desired physical sector interleave (usually 8);. applications need not set this explicitly — will be set automatically by internal GEOS routines. Only used when new directory block is allocated

Apple:

curKBlkno	current directory.
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x error ($00 = no error).
r10L page number of empty directory slot.
r1 block (Commodore track/sector) number of directory block in **diskBlkBuf.**

Commodore:

y index to empty directory slot in **diskBlkBuf.**

Apple:

r3 pointer to empty directory slot in **diskBlkBuf.** This is an absolute address, not an index into **diskBlkBuf.**
r10H unused directory entry within block in **diskBlkBuf.** There are 13 directory entries per ProDOS block, numbered 1–13.

Alters:

Commodore:

curDirHead	BAM updated to reflect newly allocated block.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

used internally by GEOS disk routines; applications generally don't use.

Destroys: Commodore:
a, r0, r3, r5, r7–r8.

Apple:
a, y, r4–r5, r7–r8.

Description: **GetFreeDirBlk** searches the current directory looking for an empty slot for a new directory entry. A single directory page has eight directory slots, and these eight slots correspond to the eight possible files that can be displayed on a single GEOS deskTop notepad page.

GetFreeDirBlk starts searching for an empty slot beginning with page number *DIRPAGE*. If **GetFreeDirBlk** reaches the last directory entry without finding an empty slot, it will try to allocate a new directory block. If *DIRPAGE* doesn't yet exist, empty pages are added to the directory structure until the requested page is reached.

$01 will most often be passed as the *DIRPAGE* starting page number, so that all possible directory slots will be searched, starting with the first page. If higher numbers are used, **GetFreeDirBlk** won't find empty directory slots on lower pages and extra directory blocks may be allocated needlessly.

GetFreeDirBlk is called by **SetGDirEntry** before writing out the directory entry for a new GEOS file.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk. A directory page corresponds exactly to a single sector on the directory track. There is a maximum of 18 directory sectors (pages) on a Commodore disk. If this 18th page is exceeded, **GetFreeDirBlk** will return a **FULL_DIRECTORY** error.

GetFreeDirBlk allocates blocks by calling **SetNextFree** to allocate sectors on the directory track. **SetNextFree** will special-case the directory track allocations. Refer to **SetNextFree** for more information.

GetFreeDirBlk does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

Apple: ProDOS directory blocks (and therefore GEOS directory blocks) contain 13 directory slots. This requires Apple GEOS to map GEOS's eight-entry directory pages to these 13-entry blocks, sometimes forcing a page to straddle a block boundary. Most applications need not worry about this straddling as it is made transparent by GEOS routines such as **FindFile**.

If an empty directory slot is not found and the end of the last block of the current directory is reached, **GetFreeDirBlk** calls **SetNextFree** to allocate a new directory block. The number of blocks in a ProDOS directory is limited only by space. If there is no more space on the disk for another directory block, and **INSUFFICIENT_SPACE** error is returned.

GetFreeDirBlk does not automatically flush the VBM cache. See **PutVBM** for more information on flushing the cache.

Example:

See also: **AllocateBlock, FreeBlock, BlkAlloc.**

GetLdVars (Apple) — mid-level disk

Function: Transfers the internal "Ld" variables (those used by **LdAppl, LdDeskAcc,** and **LdFile**) to GEOS pseudoregisters.

Parameters: none.

Returns:

r0L LOAD_OPT:
- bit 0: 0 load at address specified in file header; application will be started autmatically
  - 1 load at address in **r7**; application will not be started automatically.
- bit 7: 0 not passing a data file.
  - 1 **r2** and **r3** contain pointers to disk and data file names.
- bit 6: 0 not printing data file.
  - printing data file; application should print file and exit.

r7 LOAD_ADDR — optional load address. only used if bit 0 of *LOAD_OPT* is set (word).

r2 DATA_DISK — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the disk that contains the data file, usually a pointer to one of the **Drx CurDkNm** buffers (word).

r3 DATA_FILE — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the data file (word).

r10L RECVR_OPTS (byte).

Destroys: **a.**

Description: **GetLdVars** transfers the internal load varaibles to GEOS pseudoregisters. **GetFile, LdApplic, LdDeskAcc,** and **LdFile** call **GetLdVars** as necessary. Most applications will not need to call it directly.

See also: **SetLdVars.**

GetMode (Apple) — printer driver

Function: Get printer resolution and setable attributes.

Parameters: none.

Returns:

a	ASCIIWIDTH — page width for ASCII printing (characters per line), assumes 10 cpi pitch (byte).
x	PGWIDTH — page width in cards: number of 8x8 cards that will fit horizontally on a page (1–80, standard value is 80 but some printers only handle 60, 72, or 75) (byte). If $00, then this is an older driver: assume 80 cards.
y	PGHEIGHT — page height in cards: number of 8x8 cards that will fit vertically on a page (1–255, usually 94) (byte).
r0–r2L	MODE — flags for possible printer modes and capabilities (five bytes).

The width and height values are typically based on an 8.5" x 11" page with a 0.25" margin on all sides, leaving an 8" x 10.5" usable print area.

Destroys: assume **r2H–r4**.

Description: **GetMode** returns information concerning the resolution and capabilites of the currently installed printer.

Graphic Printing:
At each call to **PrintBuffer**, the printer driver will expect at least*PGWIDTH* cards of linear bitmap data in the 640-byte print buffer. To print an entire page , the application will need to call **PrintBuffer** *PGHEIGHT* times.

Most dot-matrix printers have a horizontal resolution of 80 dots-per-inch and an eight inch print width. Eight inches at 80 dpi gives 640 addressable dots per printed line. Since GEOS printer output is designed to map directly from screen graphics, it is sometimes useful to think of it in terms of cards (8x8 pixel blocks). There 80 (640/8) 80 cards per printable line. GEOS always assumes an 80 dpi output device.

Drivers for printers with a different horizontal resolution will usually return a *PGWIDTH* value that reflects some even multiple of the dpi. For example, a lower resolution 72 dpi printer can only fit 72*8 = 560 dots per line, and 560 dots reduces to 72 cards. *PGWIDTH* in this case would come back as 72. The printer driver will only ouput the first 72 cards (560 dots) on each line.

A 300 dpi laser printer, however, can accomodate 2,400 dots on an eight inch line. To scale 80 dpi data to 300 dpi, each pixel is expanded to four times its normal width. But 640*4 = 2,560, which is 160 dots more than the printer can handle. If the printer driver tried to print the full 640 possible dots at this expanded width, it would lose the last 160 dots. The printer driver truncates the width at the card boundary nearest to 2,400 dots, which happens to be 75 cards (75*8*4 = 2400). Hence, in this case, *PGWIDTH* would come back as 75.

The size, *PGHEIGHT*, reflects the number of card rows to send through **PrintBuffer** to fill a full-page. If more rows are sent, then (depending on the

printer and the driver) the printing will usually continue onto the next page (printing over the perforation on z-fold paper). The application will usually keep an internal card-row counter and call **StopPrint** to advance to the next page.

ASCII Printing:
The printer is capable of printing ASCIIWIDTH characters per line and 66 lines per page.

The *MODE* Values:
GetMode returns a number of flags in **r0–r2L.** These flags determine whether a specific feature is offered by the current printer and its printer driver. These bits correspond to features that may be set with **SetMode:**

r0L: (matches **currentMode** text variable)

bit	description
b7	†underline.
b6	†bold.
b5	reverse.
b4	†italic.
b3	outline.
b2	†superscript.
b1	†subscript.
b0	*reserved for future use.*

r0H: (text density)

bit	description
b7	†pica type (10 cpi).
b6	†elite type (12 cpi).
b5	†condensed type (16 cpi).
b4	†proportional type.
b3	†double height.
b2	†half-height.
b1	†eight lines per inch vertical density.
b0	†six lines per inch vertical density.

r1L: (miscellaneous features)

bit	description
b7	print red or magenta.
b6	print yellow.
b5	print blue or cyan.
b4	†expanded (double-width) type.
b3	†NLQ (near letter quality) type.
b2	CR always linefeeds, preventing manual overstrike by application
b1	*reserved for future use.*
b0	*reserved for future use.*

r1H: (internal font availability)

bit	description
b7	font 7.
b6	font 6.
b5	font 5.
b4	font 4.
b3	font 3.
b2	font 2 (Roman?).
b1	font 1 (Helvetica?).
b0	font 0 (Courier?).

r2L: (horizontal graphics density)

bit	description
b7	60 dpi.
b6	†80 dpi.
b5	90 dpi.
b4	120 dpi.
b3	180 dpi.
b2	300 dpi.
b1	360 dpi.
b0	*reserved for future use.*

†This feature implemented in most Berkeley Softworks printer drivers.

C64 & C128: Refer to **GetDimensions.**

See also: **SetMode.**

GetNextChar (Apple, C64, C128) text/keyboard

Function: Retrieve the next character from the keyboard queue.

Parameters: none.

Returns: **a** keyboard character code of characer or **NULL** if no characters available.

Alters: **pressFlag** if the call to **GetNextChar** removes the last character from the queue, then the **KEYPRESS_BIT** is cleared.

Destroys: **x**.

Description: **GetNextChar** checks the keyboard queue for a pending keypress and returns a non-zero value if one is available. This allows more than one character to be processed without returning to **MainLoop**.

Example:

See also: **GetString**.

GetNxtDirEntry (Apple, C64, C128) — mid-level disk

Function: Given a pointer to a directory entry returned by **Get1stDirEntry** or **GetNxtDirEntry**, returns a pointer to the *next* directory entry..

Parameters:
r5 CURDIRENTRY — pointer to current directory entry as returned from **Get1stDirEntry** or **GetNxtDirEntry**; will always be a pointer into **diskBlkBuf** (word).

Uses:
curDrive
diskBlkBuf must be unaltered from previous call to **Get1stDirEntry** or **GetNxtDirEntry**.

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns:
x error ($00 = no error).
r5 pointer to next directory entry within **diskBlkBuf**.

Commodore:
y non-zero if end of directory reached

Alters:
diskBlkBuf directory block.

Destroys:
Commodore:
a, r1, r4.

Commodore:
a, y, r1, r4.

Description: **GetNxtDirEntry** increments **r5** to point to the next directory entry in **diskBlkBuf**. If **diskBlkBuf** is exceeded, the next directory block is read in and **r5** is returned with an index into this new block. Before calling **GetNxtDirEntry** for the first time, call **Get1stDirEntry**.

C64 & C128: **GetNxtDirEntry** did not appear in the jump table until version 1.3. An application running under version 1.2 can access **GetNxtDirEntry** by calling directly into the Kernal. The following subroutine will work on Commodore GEOS v1.2 and later:

```
;*******************************************************
;
;     MyGetNxtDirEntry -- Use instead of GetNxtDirEntry
;     to work on GEOS v1.2 and later
;
;*******************************************************

;EQUATE: v1.2 entry point directly into Kernal. Must
;do a version check before calling.
o_GetNxtDirEntry    = $ca10        ;exact entry point

MyGetNxtDirEntry:
      lda      version             ;check version number
      cmp      #$13
      bge      10                  ;branch if v1.3 or later
```

```
        jmp     o_GetNxtDirEntry    ;direct call

10$:
        jmp     GetNxtDirEntry      ;thru jump table
```

Apple: **GetNxtDirEntry** did not appear in the jump table until version 2.0, release 3. An application running under an earlier version can add **GetNxtDirEntry** into the jump table with the patch provided in Appendix @@.

Example:

See also: **Get1stDirEntry, FindFTypes.**

GetOffPageTrSc (C64, C128) mid-level disk

Function: Get track and sector of off-page directory.

Parameters: none.

Uses:

curDrive	
curType	GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns:

x	error ($00 = no error).
y	$ff if the disk is not a GEOS disk and therefore has no off-page directory block, otherwise $00.
r1L	track of off-page directory.
r1H	sector of off-page directory.
r4	pointer to **curDirHead**.

Alters:

curDirHead	contains directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
isGEOS	set to **TRUE** if disk is a GEOS disk, otherwise set to **FALSE**.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Commodore:
a, y, r5.

Description: Commodore GEOS disks have an extra directory block somewhere on the disk called the off-page directory. The GEOS deskTop uses the off-page directory block to keep track of file icons that have been dragged off of the notepad and onto the border area of the deskTop. The off-page directory holds up to eight directory entries.

GetOffPageTrSc reads the directory header into the buffer at **curDirHead** and calls **ChkDkGEOS** to ensure that the disk is a GEOS disk. If the disk is not a GEOS disk, it returns with $ff in the **y** register. Otherwise, **GetOffPageTrSc** copies the off-page track/sector from the **OFF_OP_TR_SC** word in the directory header to **r1** and returns $00 in **y**.

Apple: Apple GEOS does not use an off-page directory block. The Apple GEOS deskTop manages files that are dragged onto the border area entirely in software.

Example:

```
; Put off-page block into DiskBlkBuf
     jsr      GetOffPageTrSc          ;get off-page directory block
     txa                              ;check for error
     bne      99$                     ;
     tya                              ;check for GEOS disk
     tax                              ;put in x in case error
     bne      99$                     ;
     LoadW    r4,#diskBlkBuf          ;get off-page block
     jsr      GetBlock                ;
99$: rts                              ; return with error in x
```

See also: **PutDirHead, GetVBM, PutVBM.**

GetPathname (Apple) — high-level disk

Function: Builds out an ASCII pathname string to any disk directory.

Parameters:

r1	PATHBUF — pointer to buffer to place pathname (word).
r0L	BUFSIZE — size of pathname buffer (byte).
r2	KEYBLKNO — key block of directory to build path for; pass the value in **curKBlkno** to get the path of the current directory.

Uses: **curDrive**

Returns:

x	error ($00 = no error).
y	pathname status ($00 = OK; **BFR_OVERFLOW** = pathname longer than *BUFSIZE* bytes).

Alters: **diskBlkBuf** used to hold temporary blocks.

Destroys: **a, r0–r2L, r4.**

Description: **GetPathname** works backward from the directory specified by *KEYBLKNO*, building out a pathname in the buffer pointed to by *PATHBUF*. The pathname is built up in ASCII from right to left. If the actual path is larger than *BUFSIZE* bytes, the lowest *BUFSIZE* characters of the path are placed in the buffer and a **BFR_OVERFLOW** error is returned.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **GoDirectory, UpDirectory, DownDirectory.**

GetPattern (Apple) graphics

Function: Copy an eight-byte GEOS pattern definition to an application's buffer.

Parameters:
a GEOS system pattern number.
r0 pointer to eight-byte destination buffer (word).

Returns:
r0 unchanged.
Pattern data in buffer pointed to by **r0**

Destroys: **a, y**

Description: **GetPattern** downloads an eight-byte pattern definition from auxiliary high memory to a buffer defined by the application. This is the only convenient way to gain access to the system pattern definitions under Apple GEOS. To redefine a fill pattern, use **SetUserPattern.**

Example:

See also: **SetPattern, SetUserPattern.**

GetPtrCurDkNm (Apple, C64, C128) high-level disk

Function: Get pointer to the current disk name.

Parameters: **x** PTR — zero-page address to place pointer (byte pointer to a word variable).

Uses: **curDrive** currently active drive.

Returns: **x** unchanged.
zero-page word at **$00,x** (*PTR*) contains a pointer to the current disk name.

Destroys: **a, y.**

Description: **GetPtrCurDkNm** returns an address that points to the name of the current disk. Disk names are stored in the **Dr*x*CurDkNm** variables, where *x* designates the drive (A, B, C, or D). If drive A is the current drive then **GetPtrCurDkNm** would return the address of **DrACurDkNm**. If drive B is the current drive then **GetPtrCurDkNm** would return the address of **DrBCurDkNm**. And so on.

Although the locations of the **Dr*x*CurDkNm** buffers are at fixed memory locations, they are not contiguous in memory. It is easier to call **GetPtrCurDkNm** than hardcode the addresses into the application. This will also ensure upward compatibilty with future versions of GEOS that might support more drives.

C64: Versions of GEOS before v1.3 only suport two disk drives and therefore only have two disk name buffers allocated (**DrACurDkNm** and **DrBCurDkNm**). GEOS v1.3 and later support additional drives C and D. **GetPtrDkNm** will return the proper pointer values in any version of GEOS as long as.**numDrives** does not exceed the number of disk name buffers. Trying to get a pointer to **DrDCurDkNm** under GEOS v1.2 will return an invalid pointer because the buffer does not exist.

C64 & C128: Commoodore disk names are always a fixed-length 16 character string. If the name is less than 16 characters, the string is padded with $a0.

Apple: Apple disk names are null-terminated strings of 16 characters or less (counting the null-terminator).

Example:

See also:

GetRandom (Apple, C64, C128) utility

Function: Creates a 16-bit random number.

Parameters: none.

Uses: **random** seed for next random number.

Alters: **random** contains a new 16-bit random number.

Destroys: **a**

Description: **GetRandom** produces a new pseudorandom (not truly random) number using the following linear congruential formula:

random = (2*(random+1) // 65521)
(remember: // is the modulaus operator)

The new random number is always less than 65221 and has a fairly even distribution between 0 and 65521.

Note: GEOS calls **GetRandom** during Interrupt Level processing to automatically keep the **random** variable updated. If the application needs a random number more often than **random** can be updated by the Kernal, then **GetRandom** must be called manually

Example:

GetRealSize (Apple, C64, C128) text

Function: Calculate the printed size of a character based on any style attributes.

Parameters:
a CHAR — character code of character (byte).
x MODE — style mode (as stored in **currentMode**).

Uses:
curHeight
baselineOffset

Apple:
lastWidth (if character code is **USELAST**)

Returns:
y character width in pixels (with attributes).
x character height in pixels (with attributes).
a character baseline offset (with attributes).

Destroys: nothing.

Calls: **GetCharWidth**

Description: **GetRealSize** calculates the width of the character based any style attributes The character code must be 32 or greater. If the character code is **USELAST**, the value in **lastWidth** is returned. Any other character code returns the pixel width as calculated from the font data structure and the *MODE* parameter.

C64 & C128: **lastWidth** is local to the GEOS Kernal and therefore inaccessible to applications. It contains the actual width of the most recently printed character.

See also: **GetCharWidth.**

Example:

```
; Calculate size of largest character in current font
     lda      #'W'                      ; capital W is a good choice
     ldx      #(SET_BOLD|SET_OUTLINE    ; widest style combo
     jsr      GetRealSize               ; dimensions come back in x,y
```

GetScanLine (Apple, C64, C128) graphics

Function: Calculate the memory address of a particular screen line.

Parameters: **x** Y — y-coordinate of line.

Uses: **dispBufferOn:**
bit 7 — calculate foreground screen address.
bit 6 — calculate background buffer address.

Returns: x unchanged.
addresses in **r5** and **r6** based on **dispBufferOn** status:

bit 7	bit 6	returns
1	1	r5 = foreground; r6 = background
0	1	r5, r6 = background
1	0	r5, r6 = foreground
0	0	*error: r5,r6 = address of screen center*

Destroys: **a**

Description: **GetScanLine** calculates the address of the first byte of a particular screen line. The routine always places addresses in both **r5** and **r6**, depending on the value in **dispBufferOn**. This allows an application to automatically manage both foreground screen and background buffer writes according to the bits set in **dispBufferOn** by merely doing any screen stores twice, indirectly off both **r5** and **r6** as in:

```
; Note: this code is C64 specific (see notes below for 128 and Apple)
     ldy      xpos        ;byte index into current line
     lda      grByte      ;graphics byte to store
     sta      (r5),y      ;store using both indexes
     sta      (r6),y      ;
```

128: When GEOS 128 is operating in 80-column mode, all foreground writes are sent through the VDC chip to its local RAM. In this case, the address of the foreground screen byte is actually an index into VDC RAM for the particular scanline. For background writes, the address of the background screen byte is an absolute address in main memory (be aware, though, that the background screen is broken into two parts and is not a contiguous chunk of memory).

In 40-column mode, **GetScanLine** operates as it does under GEOS 64.

Apple: Because the Apple double hi-res screen is spread across two memory banks, it is sometimes necessary to enable or disable a given bank before accessing the screen. The following subroutine will swap in the correct memory bank:

```
;************************************************
;Routine to set correct memory bank in Apple
;
;Pass:              a - byte index into line (xpos)
;
;Returns:      proper memory bank swapped in
;                   y - corrected index into line
```

```
;
;Destroys:     a
;***********************************************
SetCorrectPage:
      lsr       a             ;xpos/2; low-bit into carry
      tay                     ;return correct index in y
      bcs       10$           ;Is low-bit set?
      sta       PAGE2_ON      ;-->NO: swap in aux. memory
      rts                     ;       return
10$:
      sta       PAGE2_OFF     ;-->YES: swap in main memory
      rts                     ;       return
```

Example:

GetScreenLine (Apple) graphics

Function: Copies a byte-aligned horizontal line from the screen, in internal format, to an application's buffer.

Parameters:
r0 DATA — address of buffer to copy data to (word).
r1H XINDX — byte in line to begin with (seven-bit Apple screen byte) (byte).
r2L XWIDTH — width in bytes of line (seven-bit Apple screen bytes) (byte).
r1L Y — y-coordinate of line (byte).

*where (XINDX*7,Y) and (XINDX*7+XWIDTH,Y) define the endpoints of the line.*

Uses: **dispBufferOn:**
bit 7 — get data from foreground screen if set.
bit 6 — get data from background buffer if set.
If both bits are set, foreground screen is used.

Returns: **r0** address of byte following last byte in line.

Destroys: **a, x, y, r0, r1H**

Description: **GetScreenLine** copies bytes directly from the screen memory to the specified buffer. The screen is treated as a contiguous block of bytes even though, in actuality, alternate bytes lie in different memory banks.

Bytes are copied into the buffer pointed to by **r0** beginning at the byte-index into the line stored in **r1H**.

Note: No clipping at the screen edge is performed; the values passed are assumed to lie entirely on one screen line.

Example:

GetSerialNumber (Apple, C64, C128) internal

Function: Return the 16-bit serial number or pointer to the serial string for the current GEOS kernal.

Parameters: none.

Returns: Commodore:
r0 16-bit serial number.

Apple:
r0 pointer to serial string.

Destroys: a.

Description: **GetSerialNumber** gives an application access to an unencrypted copy of the GEOS serial number or serial string for comparison purposes. You cannot change the actual serial string or number by altering this copy.

GetSpriteData (Apple) sprite

Function: Copies a 64-byte sprite image from the internal data buffer that is used for drawing the sprites to the application's memory space.

Parameters:
r3L SPRITE — sprite number (byte).
r4 BUFPTR — pointer to 64-byte buffer (word).

Returns: sprite data in buffer pointed to by *BUFPTR*.

Destroys: **a, y**

Description: **GetSpriteData** is the functional opposite of **DrawSprite**. It copies 64-bytes of sprite image data from the internal buffer that is used for drawing the sprites to a buffer in the application memory space.

Note: In no case should the *SPRITE* parameter be $00; a value of $00 will return incorrect data.

The **INAUX_B** constant may be or'ed into the *SPRITE* parameter to indicate that the *BUFPTR* parameter is an address in auxiliary memory.

Example:

See also: **DrawSprite, PosSprite, EnablSprite, DisablSprite, InitSprite.**

GetString (Apple, C64, C128) text/keyboard

Function: Get a string from the keyboard using a cursor prompt and echoing characters to the screen as they are typed. Runs concurrently with **MainLoop**.

Parameters:

r0	BUFR — pointer to string buffer. When called this buffer can contain a null-terminated default string (if no default string is used, the first byte of the buffer must be NULL). This buffer must be at least *MAX_CH*+1 bytes long.
r1L	FLAG — $00 = use system fault routine; $80 = use fault routine pointed to by r4 (byte).
r2L	MAX_CH — maximum number of characters to accept (not including the null-terminator).
r11	XPOS — x-coordinate to begin input (word).
r1H	YPOS — y-coordinate of prompt and upper-left of characters. To calculate this value based on baseline printing position, subtract the value in **baselineOffset** from the baseline printing position (byte).
r4	FAULT — optional (see *FLAG*) pointer to fault routine.
keyVector	STRINGDONE — routine to call when the string is terminated by the user typing a carriage return. $0000 = no routine provided.

Uses:

at call to GetString:

curHeight	for size of text prompt.
baselineOffset	for positioning default string relative to prompt.

any variables used by **PutString**.

while accepting characters:

keyVector	vectors off of **MainLoop** through here with characters.
stringX	current prompt x-position.
stringY	current prompt y-position.
string	pointer to start of string buffer.

any variables used by **PutChar**.

Returns:

from call to GetString:

keyVector	address of **SystemStringService**.
stringFaultVec	address of fault routine being used.
stringX	starting prompt x-position.
stringY	starting prompt y-position.
string	*BUFR* (pointer to start of string buffer).

when done accepting characters:

x	length of string; index to null, relative to address in **string**.
string	*BUFR* (pointer to start of string buffer).
keyVector	$0000
stringFaultVec	$0000

Destroys:

at call to GetString:

r0–r13, a, x, y.

Description: **GetString** installs a character handling routine into **keyVector** and returns immediately to the caller. During **MainLoop**, the string is built up a character at a time in a buffer. When the user presses [Return], GEOS calls the *STRINGDONE*

routine with the starting address of the string in **string** and the length of the string in the **x**-register.

The following is a breakdown of what **GetString** does:

1: Variables local to the **GetString** character input routine are initalized. Global string input variables such as **string**, **stringX**, and **stringY** are also initialized.

2: **PutString** is called to output the default input string stored in the character buffer. If no default input string is desired, the first byte of the buffer should be a **NULL**.

3: The *STRINGDONE* parameter in **keyVector** is saved away and the address of the **GetString** character routine (**SystemStringService**) is put into **keyVector**.

4: If the application supplied a fault routine, install it into **StringFaultVec**, otherwise install a default fault routine.

5: The prompt is initalized by calling **InitTextPrompt** with the value in **curHeight**. **PromptOn** is also called.

6: Control is returned to the application.

C64 &C128: String is not null-terminated until the user presses [Return]. To simulate a [Return], use the following code:

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

Note that this will also terminate the **GetString** input.

Apple: String is always kept null-terminated.

Note: This note courtesy of Bill Coleman...Because **GetString** runs off of **MainLoop**, it is a good idea to call **GetString** from the top level of the application code and return to **MainLoop** while characters are being input. That is, while at the top level of your code you can call **GetString** like this:

```
     jsr      GetString          ; Start GetString going
     rts                         ; and return immediately to MainLoop so
                                 ; that string can be input.
```

Since the routine specified by the *STRINGDONE* value stored in **keyVector** is called when the user has finished entering the string, that is where your application should again take control and process the input.

See also: **PutChar**, **PutString**, **GetNextChar**.

Example:

GetVBM (Apple) — mid-level disk

Function: Read the first (or only) VBM block into the internal VBM cache.

Parameters: none.

Uses:

curDrive	
VBMBlkno†	block number of first VBM block.

Returns:

x	error ($00 = no error).

Alters:

curVBlkno†	block loaded into cache.
VBMchanged†	set to **FALSE**; indicates current cache matches disk.
numVBMBlks†	number of VBM blocks.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4.**

Description: **GetVBM** reads the first block of the VBM (volume bit map) into the internal VBM cache. This cache is inaccessible to applications but is used indirectly by routines such as **SetNextFree**. **GetVBM** is called automatically by **OpenDisk** and **ReOpenDisk** to get a valid VBM block into the cache. Specialized applications that switch disks or drives without using **OpenDisk** or **ReOpenDisk**, should call **GetVBM** to erase any VBM bits from a previous disk that might be lurking in the cache.

For more information on Apple GEOS VBM-cacheing, refer to **PutVBM**.

Note: **GetVBM** ignores the **VBMchanged** flag. The block is *always* read in from disk, overwriting whatever is currently in the cache.

Example:

See also: **PutVBM, GetDirHead, PutDirHead.**

GoDirectory (Apple) — mid-level disk

Function: Go to a specific directory on the disk, making it the current directory.

Parameters: **r0** KEYBLKNO — block number of this directory's key block (word).

Uses:
curDrive
curKBlkno current directory.

Returns:
x error ($00 = no error).
y pathname status ($00 = OK; **BFR_OVERFLOW** = pathname longer than **pathnameBuf**).

Alters:
curKBlkno new current directory.
curDirHead header of new directory.
pathnameBuf† system pathname buffer updated to reflect new path.
curDirTabLo† system directory table
curDirTabHi†

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, r0L, r1, r2, r4.**

Description: **GoDirectory** makes the directory at *KEYBLKNO* the current working directory so that all operations happen within it.

GoDirectory first changes the current key block number, then calls **GetPathname** to build-out the full pathname in **pathnameBuf**. The current directory header is read in with a call to **GetDirHead**

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **UpDirectory, DownDirectory.**

GotoFirstMenu (Apple, C64, C128) icon/menu

Function: Retracts all sub-menus and reactivates menus at the main menu level.

Parameters: none.

Destroys: assume **r0–r15, a, x, y**

Description: **GotoFirstMenu** is used by a menu event handler to instruct GEOS to back up to the main menu level, erasing the current menu and any parent menus (except the main menu) from the foreground screen, making the main menu active when control is returned to **MainLoop**. **menuNumber** is set to $00.

GotoFirstMenu can be called from a menu event routine at any menu level, including main menu level. It operates by checking for level zero and calling **DoPreviousMenu** in a loop.

Example:

See Also: **DoMenu, DoPreviousMenu, ReDoMenu, RecoverAllMenus.**

HideOnlyMouse (Apple, C128) mouse/sprite

Function: Temporarily removes the soft-sprite mouse pointer from the graphics screen.

Parameters: nothing.

Returns: nothing.

Uses:

graphMode	(128 only).
offFlag	(Apple only).

Alters:

offFlag	set to **TRUE** (Apple only).

Destroys: **a, x, y, r1–r6**

Description: **HideOnlyMouse** temporarily removes the mouse-pointer soft-sprite. It does not affect any of the other sprites. This can be used as an alternative to **TempHideMouse** when only the mouse pointer need be hidden. The mouse pointer will remain hidden until the next pass through **MainLoop**. Any subsequent calls to **TempHideMouse** before passing through **MainLoop** again will not erase any sprites.

128: In 40-column mode (when bit 7 of **graphMode** is zero), **HideOnlyMouse** exits immediately without affecting the hardware sprites. Also, be aware that any subsequent GEOS graphic operation will hide any visible sprites by calling **TempHideMouse**, so this routine is not especially usefull if using GEOS graphics routines.

Apple: This routine sets **offFlag** to **TRUE** which will stop **TempHideMouse** from erasing any sprites until the soft-sprite handling code in **MainLoop** is encountered. In some cases this causes problems with graphic operations that occur off of **MainLoop** before this code is executed. The easiest solution to this problem is to avoid using **HideOnlyMouse** and use **TempHideMouse** with **noEraseSprites** set to **TRUE**. **offFlag** can, instead, be set to $40 if the application wants the GEOS Interrupt Level to redraw the mouse before the application returns to **MainLoop**.

Example:

See also: **TempHideMouse**.

HorizontalLine (Apple, C64, C128) graphics

Function: Draw a horizontal line with a repeating bit-pattern.

Parameters:

a	PATTERN —eight-bit repeating pattern to use (*not* a GEOS pattern number).
r3	X1 — x-coordinate of leftmost endpoint (word).
r4	X2 — x-coordinate of rightmost endpoint (word).
r11L	Y1 — y-coordinate of line (byte).

where (X1,Y1) and (X2,Y1) define the endpoints of the horizontal line.

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing

Destroys: Commodore
a, x, y, r5–r8, r11H

Apple
a, x, y, r11H

Description: **HorizontalLine** sets and clears pixels on a single horizontal line according to the eight-bit repeating pattern. Wherever a 1-bit occurs in the pattern byte, a pixel is set, and wherever a 0-bit occurs, a pixel is cleared.

Bits in the pattern byte are used left-to-right where bit 7 is at the left .A bit pattern of %11110000 would create a horizontal line like:

The pattern byte is always drawn as if aligned to a card boundary. If the endpoints of a line do not coincide with card boundaries, then bits are masked off the appropriate ends. The effect of this is that a pattern is always aligned to specific pixels, regardless of the endpoints, and that adjacent lines drawn in the same pattern will align.

Note: To draw patterned horizontal lines using the 8x8 GEOS patterns, draw rectangles of one-pixel height by calling the GEOS **Rectangle** routine with identical y-coordinates.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **VerticalLine, InvertLine, ImprintLine, RecoverLine, DrawLine.**

ImprintLine (Apple) graphics

Function: Imprints a horizontal line from the foreground screen to the background buffer.

Parameters:
r3 X1 — x-coordinate of leftmost endpoint (word).
r4 X2 — x-coordinate of rightmost endpoint (word).
r11L Y1 — y-coordinate of line (byte).

where (X1,Y1) and (X2,Y1) define the endpoints of the line to imprint.

Returns: nothing

Destroys: **a, x, y**

Description: **ImprintLine** imprints the pixels which fall on the horizontal line whose coordinates are passed in the GEOS registers. The pixel values are copied from the foreground screen to the background buffer.

Note: The flags in **dispBufferOn** are ignored; the pixels are always copied to the background buffer regardless of the value in this variable.

This routine does not exist in GEOS 64 or GEOS 128. Use **ImprintRectangle** with a height of one.

Example:

See also: **RecoverLine, HorizontalLine, InvertLine,VerticalLine , DrawLine.**

ImprintRectangle, i_ImprintRectangle (Apple, C64, C128) graphics

Function: Imprints the pixels within a rectangular region from the foreground screen to the background buffer.

Parameters: Normal:
r3 X1 — x-coordinate of upper-left (word).
r2L Y1 — y-coordinate of upper-left (byte).
r4 X2 — x-coordinate of lower-right (word).
r2H Y2 — y-coordinate of lower-right (byte).

Inline:
data appears immediately after the **jsr i_ImprintRectangle**
.byte Y1 y-coordinate of upper-left.
.byte Y2 y-coordinate of lower-right.
.word X1 x-coordinate of upper-left.
.word X2 x-coordinate of lower-right.

where (X1,Y1) is the upper-left corner of the rectangular area and (X2,Y2) is the lower-right corner.

Returns: nothing

Destroys: Commodore
a, x, y, r5–r8, r11L

Apple
a, x, y, r11L

Description: **ImprintRectangle** copies the pixels within a rectangular region from the foreground screen to the background buffer by calling **ImprintLine** in a loop. A subsequent call to **RecoverRectangle** with the same parameters will restore the rectangle to the foreground screen.

Note: The flags in **dispBufferOn** are ignored; the pixels are always copied to the background buffer regardless of the value in this variable.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **RecoverRectangle, Rectangle, InvertRectangle.**

InfoCard (Apple) — card driver

Function: Get information about the current printer card.

Parameters: none.

Returns:

x	STATUS — card error code; $00 = no error (byte)	
y	TYPE:	

b7	0: seven-bit data card. 1: eight-bit data card.
b6	0: serial interface card. 1: parallel interface card.
b5	0: card only capable of output. 1: card capable of input and output.
b4	0: does not support **StatusCard** ready-for-output flag. 1: supports **StatusCard** ready-for-output flag.
b3	0: does not support **StatusCard** input-ready flag. 1: supports **StatusCard** input-ready flag.

Destroys: a.

Description: **InfoCard** returns information about the installed card.

Note: This routine may be called at any time.

Example:

See also: StatusCard.

InitCard (Apple) — card driver

Function: Initialize printer card.

Parameters: **r0–r1L** SERIALCONFIG — card configuration information for serial I/O cards; parallel cards do not use this information (three bytes).

Returns: **x** STATUS — card error code; $00 = no error (byte)

Destroys: assume **a, y.**

Description: **InitCard** performs the basic initialization and configuration of a printer card. A printer driver will usually perform this operation from **InitForPrint. InitCard** configures the printer card with a three-byte set of *CONFIG* parameters.

The *CONFIG* Values:

Most Apple printer cards use the standard 65xx series 6551 ACIA (Asynchronous Communications Interface Adapter) to communicate between the Apple bus and the printer. The *CONFIG* bits directly reflect the available configurations for this chip. Bits marked as *unused* are either unused on the chip or forced to a specific state by the card driver. For more information, refer to the *RM 65 Family Asynchronous Communications Interface Adapter (ACIA) Module User's Manual* (Rockwell International).

r0L: control register.

bits	description	
b7	0:	one stop bit.
	1:	1-1/2 stop bits with five data bits and no parity. one stop bit with eight data bits and parity. two stop bits in all other cases.
b6–b5	00:	8 data bits.
	01:	7 data bits.
	10:	6 data bits.
	11:	5 data bits.
b4	*unused.*	
b3–b0	0000:	*undefined.*
	0001:	50 baud.
	0010:	75 baud.
	0011:	109.92 (110) baud.
	0100:	134.56 (135) baud.
	0101:	150 baud.
	0110:	300 baud.
	0111:	600 baud.
	1000:	1200 baud.
	1001:	1800 baud.
	1010:	2400 baud.
	1011:	3600 baud.
	1100:	4800 baud.
	1101:	7200 baud.
	1110:	9600 baud.
	1111:	19200 baud (19.2 Kbaud).

r0H: command register.

bits	description	
b7–b6	00:	odd parity.
	01:	even parity.
	10:	mark parity.
	11:	space parity.
b5	0:	parity disabled (no parity, ignore b7–b6).
	1:	parity enabled.
b4–b0	*undefined.*	

r1L: handshaking.

bits	description	
b7	0:	do not check for XOFF.
	1:	detect XOFF, wait for XON.
b6–b5	00:	ignore hardware handshaking.
	11:	follow hardware handshaking.
b4	must be 1.	
b3–b0	must be 000	

Most serial printers operate with eight data bits, no parity, and one stop bit. Some printers, like the Apple LaserWriter require two stop bits.

Note: **InitCard** must be called before **OpenCard.**

Example:

See also: **OpenCard.**

InitForDialog (Apple) internal

Function: Saves away the state of GEOS as if about to pass control to a dialog box.

Parameters: none.

Returns: system initialized to a near-warmstart state.

Destroys: a,x, y, r0-r4

Description: Prior to displaying a dialog box, **DoDlgBox** saves away the state of the application in an internal buffer and then reinitializes GEOS so that the dialog box runs as if it were an independent application. This allows a dialog box to use nearly all the facilities of GEOS without disrupting the parent application. This facility for temporarily suspending the state of GEOS is potentially useful in an application and can be accessed by calling **InitForDialog**.

InitForDialog saves off the current system state and then places GEOS in a near-warmstart state. To return GEOS to its previous state, call **RecoverSysRam**.

Note: Calls to **InitForDialog** cannot be nested because GEOS can only buffer its state to a depth of one level. A second call to **InitForDialog** without calling **RecoverSysRam** will overwrite the saved state of the system with the current state. Because **DoDlgBox** calls **InitForDialog** as part of its normal operation, it, too, will overwrite the saved state of the system.

See also: **RecoverSysRam.**

InitForIO (C64, C128) — very low-level disk

Function: Prepare for I/O across the serial bus.

Parameters: none.

Returns: nothing.

Destroys: a, y.

Description: **InitForIO** prepares the system to perform I/O across the Commodore serial bus. It disables interrupts, turns sprite DMA off, slows the 128 down to 1Mhz, switches in the ROM and I/O banks if necessary, and performs anything other initialization needed for fast serial transfer.

Call **InitForIO** before directly accessing the serial port (e.g., in a printer driver) or before using **ReadBlock**, **WriteBlock**, **VerWriteBlock**, or **ReadLink**. To restore the system to its previous state, call **DoneWithIO**.

Apple: Apple GEOS has no **InitForIO** equivalent.

Example: See **WriteBlock**.

See also: **DoneWithIO**, **SetDevice**.

InitForPrint (Apple, C64, C128) printer driver

Function: Initialize printer. Perform once per document.

Parameters: none.

Returns: Commodore
nothing.

Apple
x STATUS — printer error code; $00 = no error.

Destroys: Commodore
assume **a, x, y, r0–r15.**

Apple
assume **a, y, r0–r15.**

Description: **InitForPrint** performs any initialization necessary to prepare the printer for a GEOS document. Often this involves resetting the printer to bring it into a default state as well as suppressing automatic margins and perforation skipping. **InitForPrint** does not do any initialization specific to graphic or ASCII printing.

Commodore: **InitForPrint** is also used to set the printer baud rate for serial printers.

Apple: Some printer drivers may try to send a non-printing character to the printer to ensure that it is ready to accept data. If the printer is not ready, **InitForPrint** will return a **PR_TIME_OUT** error. This usually means the printer is switched off or is not online: Many printer drivers, however, will do no such checking and have no facility for detecting a printer time-out.

See also: **StartPrint, StartASCII.**

InitMouse (Apple, C64, C128) — input driver

Function: Initialize the input device.

Parameters: none.

Returns: nothing

Alters:

mouseXPos	initialized (typically 8).
mouseYPos	initialized (typically 8).
mouseData	initialized (typically reflects a released button).
pressFlag	initialized (typically set to $00).

Destroys: assume **a, x, y, r0–r15**

Description: GEOS calls **InitMouse** after first loading an input driver. The input driver is expected to initialize itself and begin tracking the input device. An application should never need to call **InitMouse.**

See also: **SlowMouse, UpdateMouse, SetMouse, KeyFilter, StartMouseMode, MouseUp.**

InitProcesses (Apple, C64, C128) process

Function: Initialize and install a process data structure.

Parameters:
a NUM_PROC — number of processes in table (byte).
r0 PTABLE— pointer to process data structure to use (word).

Returns: **r0** unchanged.

Destroys: **a, x, y, r1**

Description: **InitProcesses** installs and initializes a process data structure. All processes begin as frozen, so their timers are not decremented during vblank. Processes can be started individually with **RestartProcess** after the call to **InitProcesses.**

InitProcesses copies the process data structure into an internal area of memory hidden from the application. GEOS maintains the processes within this internal area, keeping track of the event routine addresses, the timer initialization values (used to reload the timers after they time-out), the current value of the timer, and the state of each process (i.e., frozen, blocked, runable). The application's copy of the process data structure is no longer needed because GEOS remembers this information until a subsequent call to **InitProcesses.**

Note: Although processes are numbered starting with zero, *NUM_PROC* should be the actual number of processes in the table. To initialize a process table with four processes, pass a *NUM_PROC* value of $04. When refering to those processes (i.e., when calling routines such as **UnblockProcess**), use the values $00–$03. Do not call **InitProcesses** with a *NUM_PROC* value of $00 or a *NUM_PROC* value greater than **MAX_PROCESSES** (the maximum number of processes allowable).

To disable process handling, merely freeze all processes or call **InitProcesses** with a dummy process data structure.

Example:

See also: **Sleep, RestartProcess.**

InitRam (Apple, C64, C128) memory

Function: Table driven initialization for variable space and other memory areas.

Parameters: **r0** TABLE —address of initialization table (word).

Returns: nothing.

Destroys: **a, x,y, r0–r2L**

Description: **InitRam** uses a table of data to initialize blocks of memory to preset values. It is useful for setting groups of variables to specific values. It is especially good at initializing a group of noncontiguous variables in a "two bytes here, three bytes there" fashion.

The initialization table that is pointed to by the TABLE parameter is a data structure made up from the following repeating pattern:

```
.word   address                 ;start address of this block
.byte   count                   ;number of bytes to initialize
.byte   byte1, byte2,..byteN    ;count bytes of data

.word   address..               ;start address for next block
```

The table is made of blocks that follow the above pattern. *count* bytes starting at *address* are initialized with the next *count* bytes in the table. (A *count* value of $00 is treated as 256.) To end the table, use

```
.word   $0000                   ;end table
```

where **InitRam** expects the next *address* parameter.

Note: Do not use **InitRam** to initalize **r0–r2L.**

Example:

See also: **FillRam, ClearRam.**

InitSprite (Apple) — sprite

Function: Initializes all sprites.

Parameters: nothing.

Returns: nothing.

Alters: **offFlag** set to **TRUE.**
noEraseSprites set to $00.
all mouse and sprite parameters set to $00.

Destroys: **a, x**

Description: **InitSprite** is called by the GEOS Kernal during the boot and warmstart sequences. It is probably not useful to applications.

See also: **DrawSprite, GetSpriteData, PosSprite, EnablSprite, DisablSprite.**

InitTextPrompt (Apple, C64, C128) — text/keyboard

Function: Initialize sprite #1 for use as a text prompt.

Parameters: **a** HEIGHT — pixel height for the prompt (byte)

Alters: **alphaFlag** %10000011

Destroys: **a, x, y.**

Description: **InitTextPrompt** initializes sprite #1 for use as a text prompt. The sprite image is defined as a one-pixel wide vertical line of *HEIGHT* pixels. If *HEIGHT* is large enough, the double-height sprite flags will be set as necessary. *HEIGHT* is usually taken from **curHeight** so that it reflects the height of the current font.

The text prompt will adopt the color of the mouse pointer.

See also: **PromptOn, PromptOff.**

Example:

InputByte (Apple) — card driver

Function: Get a byte of input from the interface card..

Parameters: none.

Returns:

x	STATUS — card error code; $00 = no error (byte)
y	DATA — single input byte as read from card.

Destroys: a, y.

Description: **InputByte** returns a byte of data from the card, assuming the card supports two-way communication. The input-ready flag may be checked with **StatusCard** prior to calling **InputByte**.

Note: **InputByte** must be called after an **OpenCard** and before a **CloseCard.**

Most card drivers always return **NO_ERROR** from from **InputByte** due to a lack of memory to properly handle complex error checking.

Example:

See also: **OutputByte, StatusCard.**

InsertRecord (Apple, C64, C128) VLIR disk

Function: Adds an empty record before the current record in the index table, moving all subsequent records (including the current record) downward.

Parameters: none.

Uses:

curDrive	
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM/VBM into memory.
curRecord	current record pointer
fileHeader	VLIR index table stored in this buffer.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	current directory header/BAM.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:

curRecord	new record becomes the current record.
fileWritten†	set to **TRUE** to indicate the file has been altered since last updated.
fileHeader	new record added to index table.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r0L.**

Description: **InsertRecord** attempts to insert an empty VLIR record in front of the current record in the index table of an open VLIR file, moving all subsequent records downward in the record list. The new record becomes the current record. A VLIR file can have a maximum of **MAX_VLIR_RECS** records. If adding a record will exceed this value, an **OUT_OF_RECORDS** error is returned. In the index table, the new record is marked as used but empty.

InsertRecord does not update the VLIR file information on disk. Call **CloseRecordFile** or **UpdateRecordFile** to update the file when done modifying.

Example:

See also: **AppendRecord, DeleteRecord.**

InterruptMain (Apple, C64, C128) internal

Function: Main Interrupt Level processing.

Parameters: none.

Returns: nothing.

Destroys: **a, x, y, r0–r15**

Description: **InterruptMain** is the main GEOS Interrupt Level processing loop and that means different things on different systems.

C64 & C128: **InterruptMain** is a subset of the full Interrupt Level process. **InterruptMain** is typically called through the **IntTopVec**. An application could conceivably **jsr InterruptMain** to "catch up" on some system updating if interrupts have been disabled for a considerable period of time. **InterruptMain** is not re-entrant, so it is important that interrupts be disabled around the catch-up calls.

Apple: **InterruptMain** is currently the same as **IrqMiddle**. It is best to use **IrqMiddle** to generate software interrupts because the Apple GEOS **InterruptMain** may change in the future to more closely reflect the operation under the Commodore environment. At this time, avoid this jump table entry.

See also: **IrqMiddle, MainLoop.**

InvertLine (Apple, C64, C128) graphics

Function: Invert the pixels on a horizontal line.

Parameters:
r3 X1 — x-coordinate of leftmost endpoint (word).
r4 X2 — x-coordinate of rightmost endpoint (word).
r11L Y1 — y-coordinate of line (byte).

where (X1,Y1) and (X2,Y1) define the endpoints of the line to invert.

Uses: **dispBufferOn:**
bit 7 — invert foreground screen if set.
bit 6 — invert background screen if set.

Returns: nothing

Destroys:
Commodore
a, x, y, r5–r8

Apple
a, x, y

Description: **InvertLine** inverts the pixel state of all pixels which fall on the horizontal line whose coordinates are passed in the GEOS registers. Set pixels become clear, and clear pixels become set.

Note: If **dispBufferOn** is set to invert on the foreground and the background screen, both the foreground and the background screen will get the inverted foreground pixels. GEOS assumes both screens contain the same image.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **VerticalLine, HorizontalLine, ImprintLine, RecoverLine, DrawLine.**

InvertRectangle (Apple, C64, C128) graphics

Function: Inverts the pixels within a rectangular region.

Parameters:

r3	X1 — x-coordinate of upper-left (word).
r2L	Y1 — y-coordinate of upper-left (byte).
r4	X2 — x-coordinate of lower-right (word).
r2H	Y2 — y-coordinate of lower-right (byte).

where (X1,Y1) is the upper-left corner of the rectangular area and (X2,Y2) is the lower-right corner.

Uses: **dispBufferOn:**
bit 7 — invert on foreground screen if set.
bit 6 — invert on background screen if set.

Returns: nothing

Destroys: Commodore
a, x, y, r5–r8, r11H

Apple
a, x, y, r11H

Description: **InvertRectangle** inverts all the pixels within the rectangular area as determined by the coordinates of the upper-left and lower-right corners. All set pixels become clear and clear clear pixels become set.

InvertRectangle operates by calling **InvertLine** in a loop.

InvertRectangle is handy to use for indicating a selected object (as GEOS does with icons) or for flashing an area by inverting a rectangle twice, first inverting the area and then inverting it back to its original state.

Note: If **dispBufferOn** is set to invert on the foreground and the background screen, both the foreground and the background screen will get the inverted foreground pixels. GEOS assumes both screens contain the same image.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **Rectangle, ImprintRectangle, RecoverRectangle, FrameRectangle.**

IrqMiddle (Apple) internal

Function: Interrupt Level processing.

Parameters: none.

Returns: nothing.

Destroys: a, x, y

Description: **IrqMiddle** is the software-interrupts entry into the Apple GEOS Kernal. When there is no hardware interrupt source, GEOS generates software interrups by calling **IrqMiddle**. In most cases, as long as the application returns to **MainLoop** frequently enough, the system degradation will only be minor. However, applications which do massive amounts of processing before returning to **MainLoop** may need to generate software interrupts periodically.

```
    bit     intSource           ;check interrupt source
    bne     10$                 ;do nothing if hard interrupts
    php                         ;else, disable interrupts
    sei                         ;(just in case) because int
                                ;code is not re-entrant
    jsr     IrqMiddle           ;software interrupt now!
    plp                         ;restore old interrupt status
10$:
```

Example:

See also: **InterruptMain.**

IsMseInRegion (Apple, C64, C128) mouse

Function: Checks to see if the mouse is within a specified rectangular region of the screen.

Parameters:

r3	X1 — x-coordinate of upper-left (word).
r2L	Y1 — y-coordinate of upper-left (byte).
r4	X2 — x-coordinate of lower-right (word).
r2H	Y2 — y-coordinate of lower-right (byte).

where (X1,Y1) is the upper-left corner of the rectangle and (X2,Y2) is the lower-right corner.

Returns:

a	**TRUE** if in region, **FALSE** if not in region.
st	result of loading **TRUE** or **FALSE** into the **a** register.

Destroys: nothing.

Description: **IsMseInRegion** tests the position of the mouse against the boundaries of a rectangular region (passed in the same GEOS registers as the **Rectangle** routine). It returns **TRUE** if the mouse is within the region (inclusive) and **FALSE** if the mouse is outside the region. Because the **st** register reflects the result of loading **TRUE** or **FALSE** into the accumulator, the call can be followed by a branch instruction that tests the result, such as:

```
      beq     InRegion            ;branch if mouse was in region
        - or -
      bne     NotInRegion  ;branch if mouse not in region
```

Note: Interrupts should always be disabled around a call to **IsMseInRegion**.If the **php-sei-plp** method is used, be aware that the **plp** will reset the **st** flags. If this is troublesome, it may warrant creating a new version of **IsMseInRegion** that does its own interrupt disable and leaves the values in the **st** register intact:

```
NewIsMseInRegion:
      lda     mouseYPos           ;get mouse y-position
      cmp     r2L                 ;compare to top edge
      blt     10$                 ;branch if outside
      cmp     r2H                 ;compare to bottom edge
      bgt     10$                 ;branch if outside
      php                         ;disable interrupts around x check
      sei                         ;so it doesn't change while we're looking
      cmpW    mouseXPos,r3        ;compare mousex with left edge
      blt     5$                  ;branch if outside
      cmpW    mouseXPos,r4        ;compare mousex with right edge
      bgt     5$                  ;branch if outside
      plp                         ;(restore interrupts before setting st reg)
      lda     #TRUE               ;return inside region statys
      rts                         ;exit
5$:
      plp                         ;(restore interrupts before setting st reg)
10$:
      lda           #FALSE        ;return outside region status
      rts                         ;exit
```

Example:

```
LoadW    r3,#windowX1         ;get coordinates of window's rectangle
LoadW    r2L,#windowY1
LoadW    r4,#windowX2
LoadW    r2H,#windowY2
php                           ;disable interrupts around region check
sei
jsr      IsMseInRegion        ;check for mouse inside region
plp
tax                           ;set status flags that plp killed
beq      MouseOutsideWindow   ;branch if outside window area
```

JsrToAux (Apple only) utility

Function: jsr's to a subroutine in auxiliary memory. Returns to main memory.

Parameters:
a [AUXADDR — low byte of subroutine to call in aux. memory.
x]AUXADDR — high byte of subroutine to call in aux memory.

where AUXADDR is the address of a subroutine to call in aux. memory

Returns: depends on the subroutine.

Destroys: depends on the subroutine.

Description: **JsrToAux** performs a **jsr** to a subroutine in application auxiliary memory. GEOS will first switch application aux in and application main out, then simulate a **jsr** to the address passed in **x** and **a**. When the aux memory subroutine returns, control will resume in main memory. **JsrToAux** does not check for a null ($0000) address like **CallRoutine**.

Example:

See also: **CallRoutine**

KeyFilter (Apple) — input driver

Function: Pre-processes keyboard input for filtering out special characters, translating others, and updating the mouse variables when positioning keys are pressed.

Parameters: **a** KEY — key as scanned from keyboard circuitry.

Returns: **a** translated character or **NULL** if filtering out.

Alters:

mouseXPos	mouse x-position.
mouseYPos	mouse y-position.
mouseData	state of mouse button.
pressFlag	**MOUSE_BIT** and **INPUT_BIT** set appropriately.
inputData	depends on device.

Uses: **OPEN_APPLE** read-only hardware location; negative if open-apple pressed.

Destroys: assume **x, y**

Description: The Apple GEOS keyboard scanning routine calls **KeyFilter** at Interrupt Level whenever a keypress is detected. This allows the input driver to translate certain keypresses into mouse movements. **KeyFilter** has first dibs on any incoming keypress.

A typical **KeyFilter** routine will compare the *KEY* parameter against a table of specific keypresses. If there is no match, then **KeyFilter** returns without altering the *KEY* code. If there is a match, **KeyFilter** updates the mouse variables appropriately and returns a *KEY* value of **NULL**. Apple GEOS will ignore the null-key, never placing it in the keyboard input queue.

If *KEY* is already **NULL** when **KeyFilter** is called, then the keypress has already been filtered out by some other prior filter and should be ignored.

An application should not call **KeyFilter** directly; Apple GEOS will call it automatically at Interrupt Level.

Note: The *KEY* parameter is not a GEOS key value. It comes almost directly from the Apple hardware register **KEYBD_DATA**. The only difference is that the high-bit is set to represent the state of the closed-apple key. If bit 7 is set, then the closed-apple key is pressed. To detect the state of the open-apple key:

```
bit     OPEN_APPLE          ;check open apple key status
bmi     OpenPressed         ;branch if pressed
```

See also: **AuxDKeyFilter.**

LdApplic (Apple, C64, C128) mid-level disk

Function: Load and (optionally) run a GEOS application, passing it the standard application startup flags as if was launched from the deskTop.

Parameters:

r9 DIRENTRY — pointer to directory entry of file, usually points to **dirEntryBuf** (Apple GEOS: must be in main memory) (word).

r0L LOAD_OPT:
- bit 0: 0 load at address specified in file header; application will be started automatically
  - 1 load at address in **r7**; application will not be started automatically.
- bit 7: 0 not passing a data file.
  - 1 **r2** and **r3** contain pointers to disk and data file names.
- bit 6: 0 not printing data file.
  - printing data file; application should print file and exit.

r7 LOAD_ADDR — optional load address. only used if bit 0 of *LOAD_OPT* is set (word).

r2 DATA_DISK — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the disk that contains the data file, usually a pointer to one of the **Drx CurDkNm** buffers (word).

r3 DATA_FILE — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the data file (word).

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank destination bank (**MAIN** or **AUX**); set to **MAIN** when loading an application.

Returns: *only returns if alternate load address or disk error.*

x error ($00 = no error).

Passes: *usually doesn't return, but warmstarts GEOS and passes the following:*

r0 as originally passed to **LdApplic**.
r2 as originally passed to **LdApplic** (use **dataDiskName**).
r3 as originally passed to **LdApplic**.(use **dataFileName**).

Alters: GEOS brought to a warmstart state.

dataDiskName contains name of data disk if bit 7 of **r0** is set.
dataFileName contains name of data file if bit 6 of **r0** is set.

Destroys: **a, x, y, r0–r15.**

Description: **LdApplic** is a mid-level application loading routine called by the higher level **GetFile**. Given a directory entry of a GEOS application file, **LdApplic** will attempt load it into memory and optionally run it. **LdApplic** calls **LdFile** to load the application into memory: a sequential file is loaded entirely into memory but

only record zero of a VLIR file is loaded. Based on the status of bit 0 of *LOAD_OPT*, optionally runs the application by calling it through **StartAppl**.

Most applications will not call **LdApplic** directly but will go indirectly through **GetFile**.

Note: Only in extremely odd cases will an alternate load address be specified for an application. Loading an application at another location is not particularly useful because it will most likely not run at an address other than its specifiec load address. When **LdApplic** returns to the caller, it does so before calling **StartAppl** to warmstart GEOS.

Apple: Applications are designed to load and run in main memory. Ensure that **RWbank** contains **MAIN** before calling **LdApplic**.

Example:

See also: **GetFile, LdDeskAcc, LdDeskAcc, StartAppl.**

LdDeskAcc (Apple, C64, C128) mid-level disk

Function: Load and run a .GEOS desk accessory.

Parameters:
- **r9** DIRENTRY — pointer to directory entry of file, usually points to **dirEntryBuf** (word).
- **r10L** RECVR_OPTS — no longer used; set to $00 (see below for explanation (byte).

Uses: **curDrive**

Commodore:
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
- **RWbank** destination bank (**MAIN** or **AUX**); set to **MAIN** when loading a desk accessory.

Returns: *returns when desk accessory exits with a call to **RstrAppl**.*
- **x** error ($00 = no error).

Passes: *warmstarts GEOS and passes the following to the desk accessory:*
- **r10L** as originally passed to **LdDeskAcc** (should be $00; see below).

Alters: nothing directly; desk accessory may alter some buffers that are not saved.

Destroys: **a, x, y, r0–r15.**

Description: **LdDeskAcc** is a mid-level desk accessory loading routine called by the higher level **GetFile**. Given a directory entry of a GEOS desk accessory file, **LdDeskAcc** will attempt load it into memory and run it. When the user closes the desk accessory, control returns to the calling application.

LdDeskAcc first loads in the desk accessory's file header to get the start and ending load address. Under GEOS 64 and Apple GEOS, it will then save out the area of memory between these two addresses to a file on the current disk named **SWAP FILE**. The GEOS 128 version saves this area to the 24K desk accessory swap area in back RAM. Desk accessories larger than 24K cannot be used under GEOS 128 (to date, there are none); a **BFR_OVERFLOW** error is returned.

After saving the overlay area, the dialog box and desk accessory save-variables are copied to a special area of memory, the current stack pointer is remembered, and the desk accessory is loaded and executed. When the desk accessory calls **RstrAppl** to return to the application, this whole process is reversed to return the system to a state similar to the one it was in before the desk accessory was called. The **SWAP FILE** file is deleted.

Most applications will not call **LdDeskAcc** directly, but will go indirectly through **GetFile**.

Note: The *RECVR_OPTS* flag originally carried the following significance:

bit 7: 1 force desk accessory to save foreground screen area and restore it on return to application.
0 not necessary for desk accessory to save foreground.
Commodore only:
bit 6: 1 force desk accessory to save color memory and restore it on return to application.
0 not necessary for desk accessory to save color memory.

However, it was found that the extra code necessary to make desk accessories save the foreground screen and color memory provided no real benefit because this context save can just as easily be accomplished from within the application itself. The *RECVR_OPTS* flag is set to $00 by all Berkeley Softworks applications, and desk accessories can safely assume that this will always be the case. (In fact, future versions of GEOS may force **r10H** to $00 before calling desk accessories just to enforce this standard!)

The application should always set **r10H** to $00 and bear the burden of saving and restoring the foreground screen and the color memory. (Color memory only applicable to GEOS 64 and GEOS 128 in 40-column mode.)

C64 : GEOS versions 1.3 and above have a new GEOS file type called **TEMPORARY**. When the deskTop first opens a disk, it deletes all files of this type. The **SWAP FILE** is a **TEMPORARY** file.

Apple: Desk accessories are designed to load into main memory. Ensure that **RWbank** contains **MAIN** before calling **LdApplic**.

Example:

See also: **GetFile, LdApplic, LdFile, RstrAppl, InitForDialog, RstrFrmDialog, WarmStart.**

LdFile (Apple, C64, C128) — mid-level disk

Function: Given a directory entry, loads a sequential file or record zero of a VLIR record.

Parameters:
r9 DIRENTRY — pointer to directory entry of file, usually points to **dirEntryBuf** (Apple GEOS: must be in main memory) (word).

Uses:
curDrive

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank destination bank (**MAIN** or **AUX**).
loadOpt† load option:
bit 0:
0 –load at address specified in file header.
1 –load at address in **loadAddr**.
loadAddr† alternate load address.

†*former local variable; applications will generally not use outside of this context.*

Returns:
x error ($00 = no error).
r7 pointer to last byte read into *BUFFER* plus one.

Alters:
fileHeader contains 256-byte GEOS file header. (This is a 512-byte buffer in Apple GEOS, although only 256 bytes are used in the GEOS file header for compatibility).

Commodore:
fileTrScTab track/sector of header in first two bytes of this table (**fileTrScTab**+0 and **fileTrScTab**+1); As the file is loaded, the track/sector pointer to each block is added to the file track/sector table starting at **fileTrScTab**+2 and **fileTrScTab**+3.

Apple:
INDEXBLOCKBUF index block for file if sequential or record 0 if VLIR (in auxiliary memory; see **ReadFile** for information on accessing this buffer).

Destroys:
Apple:
a, y, r1, r4, r7.

Description:
LdFile is a mid-level file handling routine called by the higher level **GetFile**. Given a directory entry of a sequential file, **LdFile** will load it into memory. Given the directory entry of a VLIR file, **LdFile** will load its record zero into memory.

Most applications will not call **LdFile** directly but will access this routine indirectly through **GetFile**.

C64 &C128: All versions of **LdFile** to date under Commodore GEOS are unusable because the load variables that are global under Apple GEOS (**loadOpt** and **loadAddr**) are local to the Kernal and inaccessible to applications. Fortunately this is not a problem because applications can always go through **GetFile** to achieve the same effect.

Example:

See also: **GetFile, LdApplic, LdDeskAcc, ToBASIC.**

LoadAuxSet (Apple) text

Function: Begin using a new font that resides in ausiliary memory.

Parameters: **r0** FONTPTR — aux. memory address of font header.

Returns: **r0** unchanged.

Alters:

curHeight	height of font.
baselineOffset	number of pixels from top of font to baseline.
cardDataPntr	pointer to current font image data.
curIndexTable	pointer to current font index table.
curSetWidth	pixel width of font bitstream in bytes.

Destroys: **a, x, y.**

Description: **LoadAuxSet** operates exactly like **LoadCharSet** except that it lets the font data reside in auxiliary memory.

Example:

See also: **LoadCharSet, UseSystemFont.**

LoadCharSet (Apple, C64, C128) text

Function: Begin using a new font.

Parameters: **r0** FONTPTR — address of font header. If Apple, address must be in main memory (word).

Returns: **r0** unchanged

Alters:

curHeight	height of font.
baselineOffset	number of pixels from top of font to baseline.
cardDataPntr	pointer to current font image data.
curIndexTable	pointer to current font index table.
curSetWidth	pixel width of font bitstream in bytes.

Destroys: **a, y.**

Description: **LoadCharSet** uses the data in the character set data structure to initalize the font variables for the font pointed at by the *FONTPTR* parameter.

Apple: If the font data is stored in auxiliary memory, **LoadAuxSet** must be used.

Example:

See also: **LoadAuxSet, UseSystemFont.**

MainLoop (Apple, C64, C128) internal

Function: Direct entry into the GEOS **MainLoop**..

Parameters: nothing.

Returns: n/a

Destroys: n/a

Description: Although the term "**MainLoop**" usually refers to GEOS MainLoop Level processing, it also represents an entry in the GEOS jump table. By performing a **jmp MainLoop**, the application would be returning to the top of the MainLoop Level without letting it run through its normal course of events. The application is expected to return to MainLoop Level with an **rts**, not with a call to **MainLoop**. Hence, this jump table entry is not terribly useful to applications and is primarily used internally by GEOS.

The **MainLoop** jump table entry is perhaps useful when debugging. The system could, conceivably, be returned to a "known state" by resetting the stack pointer and executing a **jmp MainLoop**. Of course, there is no guarantee that this will work.

Example:

```
ldx  #$ff          ;reset stack pointer
txs
jmp  Mainloop      ;try to get back to normal...
```

See also: **InterruptMain, FirstInit.**

MakeSubDir (Apple) — high-level disk

Function: Creates a subdirectory within the current directory.

Parameters:

r0	DIRNAME — pointer to null-terminated name for directory, maximum of 16-characters including the null terminator (word).
r10L	DIRPAGE — directory page to begin searching for free slot; each directory page holds eight files and corresponds to one notepad page on the GEOS deskTop. The first page is page one (byte).

Uses:

curDrive	
curKBlkno	current directory.

Returns:

x	error ($00 = no error).
r10L	directory page actually used.
r6	block number of key block of new directory.

Alters:

dirEntryBuf	contains newly-built directory entry.
diskBlkBuf	contains directory header just written.

Destroys: **a, y, r3–r8.**

Description: **MakeSubDir** creates an empty subdirectory in the current directory. Care must be taken not to create two subdirectories with the same name as **MakeSubDir** does not prevent this from happening.

MakeSubDir first calls **SetNextFree** to allocate the subdirectory's key block then calls **SetGDirEntry** to build its directory entry, adding it to the current directory. **MakeSubDir** then builds a directory header in **diskBlkBuf** and writes it out to the newly-allocated key block.

MakeSubDir flushes the VBM cache.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **SetGDirEntry, DeleteDir.**

MouseOff (Apple, C64, C128) mouse/sprite

Function: Temporarily disables the mouse pointer and GEOS mouse tracking.

Parameters: nothing.

Returns: nothing.

Modifies:

mobenble	sprite #0 bit cleared by **DisablSprite.**
mouseOn	clears the **MOUSEON_BIT.**

Destroys: **a**

Description: **MouseOff** temporarily disables the mouse cursor and GEOS mouse tracking by clearing the proper bit in **mouseOn** and calling **DisablSprite.** Applications can call **MouseOff** temporarily disable the mouse. The mouse can be reenabled to its previous state by calling **MouseUp.**

Example:

See also: **MouseOff, ClearMouseMode.**

MouseUp (Apple, C64, C128) mouse/sprite

Function: Reenables the mouse pointer and GEOS mouse tracking.

Parameters: nothing.

Returns: nothing.

Modifies:

mobenble	sets enable bit for sprite #0
mouseOn	sets the **MOUSEON_BIT.**

Destroys: **a**

Description: **MouseUp** reenables the mouse cursor and GEOS mouse tracking after a call to **MouseOff** by setting the proper bits in **mouseOn** and **mobenble.** **StartMouseMode** calls this routine.

Example:

See also: **MouseOff, StartMouseMode.**

MoveAuxData
~~MouseUp~~

MoveAuxData (Apple) memory

Function: Special version of **MoveData** that will move data within either main or auxiliary memory (or from one bank to the other).

Parameters:

r0 SOURCE — address of source block in application memory (word).
r1 DEST — address of destination block in application memory (word).
r2 COUNT — number of bytes to move, 0 – 32K (word).
a BANKS — which banks to use, based on the following bit fields:

b7	b6	Description
0	0	SOURCE and DEST both in main memory.
0	1	SOURCE in main memory, DEST in auxiliary memory.
1	0	SOURCE in auxiliary memory, DEST in main memory.
1	1	SOURCE and DEST both in auxiliary memory.

Returns: **r0, r1, r2** unchanged.

Destroys: **a, x, y**

Description: **MoveAuxData** (formerly Apple **MoveBData**) is a block move routine that allows data to be moved in either main memory, auxiliary memory, or between main and auxiliary. Like **MoveData**, it will move up to 32K (32,768 bytes) from one area of memory to another and the source and destination blocks can overlap. The move is actually a copy in the sense that the source data remains unaltered unless the destination area is in the same bank and overlaps the source.

MoveAuxData is especially useful for copying data from main memory to auxiliary memory or from auxiliary memory to main memory.

Note: **MoveAuxData** should only be used to move data within the application space in main and auxiliary memory. This area lies in the 48K block affected by RAMRD and RAMWRT.

Example:

See also: **MoveData, MoveBData, SwapMainAndAux.**

MoveBData (C128) memory

Function: Special version of **MoveData** that will move data within either front RAM or back RAM (or from one bank to the other).

Parameters:

r0	SOURCE — address of source block in application memory (word).
r1	DEST — address of destination block in application memory (word).
r2	COUNT — number of bytes to move (word).
r3L	SRCBANK — source bank: 0 = back RAM; 1 = front RAM (byte).
r3H	DSTBANK — destination bank: 0 = back RAM; 1 = front RAM (byte).

Returns: **r0–r3** unchanged.

Destroys: **a, x, y**

Description: **MoveBData** is a block move routine that allows data to be moved in either front RAM, back RAM, or between front and back (bank 1, the front bank, is the normal GEOS application area). If the *SOURCE* and *DEST* areas are in the same bank and overlap, *DEST*. must be less than *SOURCE*.

MoveBData is especially useful for copying data from front RAM to back RAM or from back RAM to front RAM.

MoveBData uses the **DoBOp** primitive by calling it with a *MODE* parameter of $00.

Note: **MoveBData** should only be used to move data within the designated application areas of memory. **MoveBData** is signifcantly slower than **MoveData** and should be avoided if the move will occur entirely within front RAM.

Example:

See also: **MoveData, MoveAuxData, SwapBData, VerifyBData, DoBOp.**

MoveData, i_MoveData (Apple, C64, C128) memory

Function: Moves a block data from one area to another

Parameters: Normal:
r0 SOURCE — address of source block (word).
r1 DEST — address of destination block (word).
r2 COUNT — number of bytes to move (word).

Inline:
data appears immediately after the **jsr i_MoveData**
.word SOURCE
.word DEST
.word COUNT

Returns: **r0, r1, r2** unchanged.

Destroys: **a, y**

Description: **MoveData** will move data from one area of memory to another. The source and destination blocks can overlap in either direction, which makes this routine ideal for scrolling, insertion sorts, and other applications that need to move arbitrarily large areas of memory around. The move is actually a copy in the sense that the source data remains unaltered unless the destination area overlaps it.

64 & 128: If the *DMA MoveData* option in the Configure program is enabled (GEOS v1.3 and later), **MoveData** will use part of bank 0 of the installed RAM-expansion unit for an ultrafast move operation. An application that calls **MoveData** in the normal manner will automatically take advantage of this selection. An application that relies upon a slower **MoveData** (for timing or other reasons) can disable the DMA-move by temporarily clearing bit 7 of **sysRAMFlg**. This bit can also be used to read the status of the DMA-move configuration.

64: Due to insufficent error checking in GEOS, do not attempt to move more than 30,976 ($7900) bytes at one time when the DMA-move option is enabled. Break the move up into multiple calls to **MoveData**.

128: Due to insufficent error checking in GEOS, do not attempt to move more than 14,592 ($3900) bytes at one time when the DMA-move option is enabled. Break the move up into multiple calls to **MoveData**. **MoveData** should only be used to move data within the standard front RAM application space. Use **MoveBData** to move memory within back RAM or between front RAM and back RAM. Because the RAM-expansion unit DMA follows the VIC chip bank select, an application that is displaying a 40-column screen from back RAM must either disable DMA-moves or temporarily switch the VIC chip to front RAM before the **MoveData** call.

Apple: **MoveData** should only be used to move data within application main memory. See **MoveAuxData** for moving data within aux main memory or between memory banks.

Note: Do not use **MoveData** on **r0–r6**.

Example:

See also: **MoveAuxData, MoveBData, CopyString.**

NewBitClip (Apple) graphics

Function: A special version of **BitmapClip** which allows bitmaps of pixel-widths and pixel-coordinate x-positioning.

Parameters:

r0	DATA — pointer to the compacted bitmap data (word).
r3	X — x-position in pixels (word).
r1H	Y — y-coordinate (byte).
r2	W_PWIDTH — width in pixels of clipping window (word).
r1L	W_HEIGHT — height in pixels of clipping window (byte).
r11L	DX1 — delta-x1: offset of left edge of clipping window in bytes from left edge of full bitmap (byte).
r11H	DX2 — delta-x2: offset of right edge of clipping window in cards from right edge of full bitmap (byte). Right pixel-edge of clipping window rounded upward to the next card boundary.
r12	DY1 — delta-y1: offset of top edge of clipping window in pixels from top edge of full bitmap (word).

where the upper-left corner of the clipped bitmap (the window) is placed at (X,Y). The lower-right corner is at (X+W_PWIDTH, Y+W_HEIGHT).

Uses: **dispBufferOn**:
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing.

Destroys: **a, x, y, r0–r12**

Description: **NewBitClip** is an enhanced version of **BitmapClip** which allows the left edge of the bitmap to be placed on a pixel boundary and the bitmap to have a pixel-sized width. It is otherwise identical to **BitmapClip**.

No checks are made to determine if the data, dimensions, or positions are valid. Be careful to pass accurate values.

Pay special attention to the fact that *DX1*, and *DX2* are still specified in cards (groups of eight pixels horizontally), not in individual pixels.

Note: Do not pass a value of $00 for either the *W_PWIDTH* or *W_HEIGHT* parameters.

DOUBLE_B can be or'ed into the *W_WIDTH* paramter to double the bitmap width and **INAUX_B** can be or'ed into the *X* parameter to specify auxiliary memory. For more information, refer to "Apple GEOS Bitmap Doubling and Aux-memory Bitmaps" in chapter @gr@.

Example:

See also: **NewBitOtherClip, BitmapClip, BitmapUp, BitOtherClip.**

NewBitOtherClip (Apple) graphics

Function: Special version of **NewBitClip** that allows the compacted bitmap data to come from a source other than memory (e.g., from disk).

Parameters:

- **r0** BUFFER — pointer to a 134-byte buffer area (word).
- **r3** X — x-position in pixels (word).
- **r1H** Y — y-coordinate (byte).
- **r2** W_PWIDTH — width in pixels of clipping window (word).
- **r1L** W_HEIGHT — height in pixels of clipping window (byte).
- **r11L** DX1 — delta-x1: offset of left edge of clipping window in bytes from left edge of full bitmap (byte).
- **r11H** DX2 — delta-x2: offset of right edge of clipping window in cards from right edge of full bitmap (byte). Right pixel-edge of clipping window rounded upward to the next card boundary.
- **r12** DY1 — delta-y1: offset of top edge of clipping window in pixels from top edge of full bitmap (word).
- **r13** APPINPUT — pointer to application-defined input routine. Called each time a byte from a compacted bitmap is needed; data byte is returned in the a-register.
- **r14** SYNC — pointer to synchronization routine. Called after each bitmap packet is decompressed. Due to improvements in **NewBitOtherClip**, this routine need only consist of reloading **r0** with the *BUFFER* address.

where the upper-left corner of the clipped bitmap (the window) is placed at (X,Y). The lower-right corner is at (X+W_PWIDTH, Y+W_HEIGHT).

Uses: **dispBufferOn:**

- bit 7 — write to foreground screen if set.
- bit 6 — write to background screen if set.

Returns: nothing.

Destroys: **a, x, y, r0–r12,** and the 134-byte buffer pased through **r0.**

Description: Sometimes the application will need to display a large bitmap but cannot afford to store the entire bitmapped object in memory. **NewBitOtherClip** allows the application to specify an input routine (*APPINPUT*) that returns the next byte of a compacted bitmap each time it is called.

APPINPUT must preserve **r0-r13,** it must return the data byte in the a-register, and it is expected to exit with an **rts.** A typical *APPINPUT* routine might save **r0-r13,** call **ReadByte** to get a byte from a disk file, restore **r0–r13,** load the data byte into the accumulator, and **rts.** Note that **BitOtherClip** can be used to add another level of compression to the compacted bitmap format. The *APPINPUT* routine can decompress the data into compacted bitmap format and pass those bytes back.

The basic width, height, position, and clipping window parameters are identical to **NewBitClip**. Refer to the documentation of that routine for more information.

NewBitOtherClip calls the *APPINPUT* routine until it has enough bytes to form one bitmap packet. **BitOtherClip** stores the bytes in the buffer pointed by *BUFFER* and then uncompacts the data to the screen. After the bitmap packet has been uncompacted, **BitOtherClip** calls the *SYNC* routine. This extra call is no longer particulary useful and can merely consist of reloading **r0** with the *BUFFER* address and performing an **rts**.

Note: **DOUBLE_B** can be or'ed into the *W_WIDTH* parameter to double the bitmap width.

Before resetting **r0** when, the *SYNC* routine must switch in the alternate zero page:

```
SYNC:
     sta      ALTZP_ON          ;switch in alt zp
     MoveW    syncr0,r0         ;reload r0
     sta      ALTZP_OFF         ;switch alt zp back out
     rts                        ;return from sync
```

Example:

See also: **BitOtherClip, NewBitClip, BitmapClip, BitmapUp, BitOtherClip.**

NewBitUp, i_NewBitUp (Apple) graphics

Function: A special version of **BitmapUp** that allows bitmaps of pixel-widths and pixel-coordinate x-positioning.

Parameters: Normal:

r0	DATA — pointer to the compacted bitmap data (word).
r3	X — x-coordinate (word).
r1H	Y — y-coordinate (byte).
r2	WIDTH — bitmap width in pixels (word).
r1L	HEIGHT — bitmap height in pixels (byte).

Inline:
data appears immediately after the **jsr i_BmapUp**

.word	DATA	pointer to the compacted bitmap data.
.word	X	x-coordinate.
.byte	Y	y-coordinate.
.word	WIDTH	bitmap width in pixels.
.byte	HEIGHT	bitmap height in pixels.

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing.

Destroys: **a, x, y, r0–r4**

Description: **NewBitUp** uncompacts a GEOS compacted bitmap using the width and height information and places it at the specified screen position. No checks are made to determine if the data, dimensions, or positions are valid, and bitmaps which exceed the screen edge will not be clipped. Be careful to pass accurate values.

NewBitUp differs from **BitmapUp** in that it allows bitmaps to be of a pixel-width and to be placed at any pixel postion on the x-axis. The data is still compacted in card-size (eight-pixel) chunks, but when placed on the screen any extra pixels (bits) at the right edge are ignored.

Note: Neither the *WIDTH* nor the *HEIGHT* parameter should be $00.

DOUBLE_B can be or'ed into the *WIDTH* paramter to double the bitmap width and **INAUX_B** can be or'ed into the *X1* parameter to specify auxiliary memory. For more information, refer to "Apple GEOS Bitmap Doubling and Aux-memory Bitmaps" in chapter @gr@.

Example:

See also: **BitmapUp, NewBitClip, NewBitOtherClip.**

NewDisk (C64, C128) — mid-level disk

Function: Tell the turbo software that a new disk has been inserted into the drive.

Parameters: none

Uses:

curDrive	drive that disk is in.
curType	GEOS 64 v1.3 and later: for disk shadowing (shadow memory is cleared).

Returns: **x** error ($00 = no error).

Destroys: **a, y, r0–r3.**

Description: **NewDisk** informs the disk drive turbo software that a new disk has been inserted into the drive. It first calls **EnterTurbo** then sends an initialize command to the turbo code. If the disk is shadowed, the shadow memory is also cleared.

NewDisk gets called automatically when.**OpenDisk** opens a new disk. An application that does not deal with anything but the low-level disk routines might want to call **NewDisk** instead of **OpenDisk** to avoid the unnecessary overhead associated with reading the directory header and initializing internal file-level variables.

Note: **NewDisk** has no effect on a RAMdisk. Also, some early versions of the 1541 turbo code leave the disk in the drive spinning after is first loaded. A call to **NewDisk** during the application's initialization will stop the disk.

Apple: Apple GEOS has no **NewDisk** equivalent.

Example:

See also: **OpenDisk, SetDevice.**

NextRecord (Apple, C64, C128) VLIR disk

Function: Makes the next record the current record.

Parameters: none.

Uses: **fileHeader** index table checked to establish whether record exists.

Returns:

x	error ($00 = no error).
y	if no error, then a value of $00 here means record is empty.
a	new current record number.

Commodore:

r1	track/sector of first data block in record.

Apple:

r1	block number of record's index block.

Alters: **curRecord** new record number.

Destroys: nothing.

Description: **NextRecord** makes the current record plus one the new current record. A subsequent call to **ReadRecord** or **WriteRecord** will operate with this record.

If the record does not exist, then **NextRecord** returns an **INV_RECORD** (invalid record) error.

Example:

See also: **PointRecord, PreviousRecord.**

NormalizeX (C128) graphics

Function: Adjust an x-coordinate to compensate for the higher-resolution 80-column mode.

Parameters: **x** GEOSREG — zero-page address of word-length GEOS register which contains the word-length X-coordinate to adjust..

Returns: **x** unchanged.
register passed as *GEOSREG* parameter contains the adjusted x-coordinate.

Destroys: **a**

Description: **NormalizeX** is used by nearly every GEOS 128 routine that writes to the screen. It adjusts an x-coordinate (two's complement signed word) based on the graphics mode (40- or 80-column) and the status of the special bits in the coordinate. **NormalizeX** allows an application to run in both 40- and 80-column modes with a minimum of programming effort. If the proper bits in a 40-column coordinate is set, **NormalizeX** will automatically double the value when in 80-column mode.

Since GEOS graphics operations automatically call **NormalizeX** to adjust the coordinates, most applications will not need to call it directly.

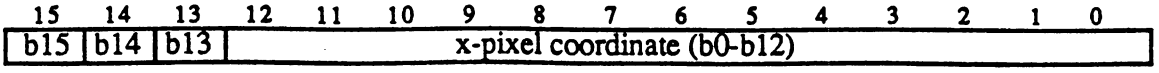
Bit 15 of the coordinate specifies doubling. Bit 13 adds one to a doubled coordinate (allowing odd-pixel addressing). Bit 12 is a pseudo-sign bit. Use the **DOUBLE_W** and **ADD1_W** constants to access these bits.

If the coordinate might be negative, the **DOUBLE_W** and **ADD1_W** constants should be exclusive-or'ed into the x-position so that the sign is preserved. However, if the coordinate is guaranteed to be a positive number, the constants may simply be or'ed in.

The *GEOSREG* parameter is an actual zero-page address. Usually this will be a GEOS register (**r0-r15**) or an application's register (**a0-a15**). If, for example, an application had a value in **r9** which it wanted normalized, it would first exclusive-or in the special bits, then call **NormalizeX** in the following manner:

```
ldx     #r9             ;load x with addr of r9
jsr     NormalizeX      ;normalize the val in r9
```

The following breakdown of the word-length x-coordinate illustrates how the special bits affect the adjustment process.



b0-b12	x-coordinate in pixels (two's comp. number).
b13	add one to doubled x-coordinate (flag).
b14	x-coordinate sign-extension from b12 (pseudo sign-bit).
b15	double x-coordinate (flag).

If in 40-column mode, then the special bits are ignored and the x-coordinate is returned to its original state (the state it was in before any special constants were exclusive-or'ed in).

If in 80-column mode, then the following applies:

b15	b14	b13	Effect
0	0	n	x value unchanged (normal positive).
1	1	n	x value unchanged (normal negative).
0	1	n	x = x*2–n (doubled negative).
1	0	n	x = x*2+n (doubled positive).

Note: For more information, refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter @gr@.

Example:

NxtBlkAlloc (Apple, C64, C128) — mid-level disk

Function: Special version of **BlkAlloc** that begins allocating from a specific block on the disk.

Parameters:

r2 BYTES — number of bytes to allocate space for (word). Commodore version can allocate up to 32,258 bytes (127 Commodore blocks); Apple version can allocate up to 65,536 bytes (128 ProDOS blocks).

Commodore:
r3L START_TR — start allocating from this track (byte).
r3H START_SC — start allocating from this sector (byte).
r6 TSTABLE — pointer to buffer for building out track and sector table of the newly allocated blocks, usually a position within **fileTrScTab** (word).

Apple:
r3 START_BLK— start allocating from this block (word).

Uses: **curDrive**

Commodore:
curDirHead this buffer must contain the current directory header.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)
interleave† desired physical sector interleave (usually 8); used by **setNextFree**. Applications need not set this explicitly — will be set automatically by internal GEOS routines.

Apple:
curVBlkno† used by VBM cacheing routines.
VBMchanged† used by VBM cacheing routines.
numVBMBlks† used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x error ($00 = no error).
r2 number of blocks allocated to hold *BYTES* amount of data.

Commodore:
r3L track of last allocated block.
r3H sector of last allocated block.

Apple:
r3 last block allocated.

Alters:

Commodore:
curDirHead BAM updated to reflect newly allocated blocks.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

Apple:
curVBlkno† used by VBM cacheing routines.
VBMchanged† set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: Commodore:
a, y, r4–r8.

Apple:
a, y, r4, r6, r7, r8H.

Description: **NxtBlkAlloc** begins allocating blocks from a specific block on the disk, allowing a chain of blocks to be appended to a previous chain while still maintaining the sector interleave. **NxtBlkAlloc** is essentially a special version of **BlkAlloc** that starts allocating blocks from an arbitrary block on the disk rather than from a fixed block. **NxtBlkAlloc** is otherwise identical to **BlkAlloc.**

C64 & C128: The Commodore version of **NxtBlkAlloc** for appending more blocks to a list of blocks just allocated with **BlkAlloc,** thus circumventing the 32,258-byte barrier. Point *TSTABLE* at the last entry in a track/sector table (the terminator bytes which we can overwrite), load the *BYTES* parameter with the number of bytes left, and call **NxtBlkAlloc.** The *START_TR* and *START_SC* parameters in **r3L** and **r3H** will contain the correct values on return from **BlkAlloc. NxtBlkAlloc** will allocate enough additional blocks to hold *BYTES* amount of data, appending them in the track/sector table automatically. This combined list of track and sectors can then be passed directly to **WriteFile** to write data to the full chain of blocks.

NxtBlkAlloc does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM. Also, the *START_TR* parameter should not be track number of the directory track. Refer to **GetFreeDirBlk** for more information on allocating blocks on the directory track.

Apple: The Apple version of **NxtBlkAlloc** builds out a list of allocated blocks in the internal **indexBlock** buffer just as if **BlkAlloc** had been called: it starts filling up the table from the beginning, overwriting any block pointers that may already be there. The only difference between **BlkAlloc** and **NxtBlkAlloc** is that **BlkAlloc** forces **SetNextFree** to start searching for free blocks beginning at the block following the VBM, whereas **NxtBlkAlloc** instructs it to start searching from the *START_BLK* block. **NxtBlkAlloc** could be used to append blocks to an existing file or record, but the application would need to manually append the new block pointers (built-out in **indexBlock**) to the end of the original index block. For this reason, the Apple GEOS version of **NxtBlkAlloc** is not especially useful.

NxtBlkAlloc does not automatically flush the VBM cache. See **PutVBM** for more information on flushing the cache.

Note: For more information on the scheme used to allocate successive blocks, refer to **SetNextFree.**

Example:

See also: **BlkAlloc, SetNextFree, AllocateBlock, FreeBlock.**

OpenCard (Apple) — card driver

Function: Open printer card for access.

Parameters: none.

Returns: x STATUS — card error code; $00 = no error (byte)

Destroys: assume **a, y.**

Description: **OpenCard** opens access to the printer card. The printer driver calls this routine to switch in the printer card's ROMs. This operation may be necessary before each block of data because another card's ROMs may be enabled (the disk controller ROMs, for example).

The printer driver sends blocks of data to the printer between calls to **OpenCard** and **CloseCard.**

Note: Do not call **OpenCard** without having called **InitCard.**

Example:

See also: **InitCard, CloseCard.**

OpenDisk (Apple, C64, C128) — high-level disk

Function: Open the disk in the current drive

Parameters: none

Uses:

curDrive	drive that disk is in.

Commodore:

driveType	type of drive to open (for shadowing information).

Returns:

x	error ($00 = no error).
r5	pointer to disk name buffer as returned from **GetPtrCurDkNm**. This is a pointer to one of the **Drx CurDkNm** arrays.

Alters:

DrxCurDkNm	current disk name array contains disk name.
curDirHead	current directory header.

Commodore:

isGEOS	set to **TRUE** if disk is a GEOS disk, otherwise set to **FALSE**.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
driveType	

Apple:

sysDirBlkno	block number of /SYSTEM directory. If system directory not found on disk, then this number is same as **curKBlkno**.
curKBlkno	block number of root directory's key block.
totNumBlks†	total number of blocks in volume.
pathnameBuf†	reset to root directory.
curVBlkno†	block currently cached in **VOLUMEBITMAP**
numVBMBlks†	number of VBM blocks on disk.
VBMchanged†	set to **FALSE**.
curDirTabLo	set to root directory.
curDirTabHi	set to root directory.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Commodore:
a, y, r0–r4.

Apple:
a, y, r1, r4.

Description: **OpenDisk** initiates access to the disk in the current drive. **OpenDisk** is meant to be called after a new disk has been inserted into the disk drive. It prepares the drive and disk variables for dealing with a new disk. An application will usually call **OpenDisk** immediately after calling **SetDevice**.

Note: Because GEOS uses the same allocation and file buffers for each drive, it is important to close all files and update the BAM/VBM if necessary (use **PutDirHead** or **PutVBM**, respectively) before accessing another disk.

C64 & C128: **OpenDisk** first calls **NewDisk** to tell the disk drive a new disk has been inserted (if the disk is shadowed, the shadow memory is also cleared). **GetDirHead** is then called to load the disk's header block and BAM into **curDirHead**. With a valid header block in memory, **ChkDkGEOS** is called to check for the GEOS I.D. string and set the **isGEOS** flag to **TRUE** if the disk is a GEOS disk. Finally, **OpenDisk** copies the disk name string from **curDirHead** to the disk name buffer returned by **GetPtrCurDkNm**.

Apple: Apple GEOS **OpenDisk** resets the path to the root directory, then reads the disk header into the **curDirHead** array. The volume disk name is copied from this buffer into the disk name buffer returned by **GetPtrCurDkNm**. This disk name is also used to build out the initial pathname in **pathnameBuf** (the inital pathname is a slash "/" character followed by the null-terminated disk name.). **OpenDisk** also initializes the VBM cache and its associated variables, computing the total number of VBM blocks and reading the first block into the cache. The flag **VBMchanged** is set to **FALSE** to indicate that the cached VBM block in memory matches its copy on the disk. When all this is done, **OpenDisk** searches the root directory for the /SYSTEM directory (using **FndFilInDir**) and stores the resulting block number into **sysDirBlkno**.

Example:

See also: **ReOpenDisk, SetDevice, NewDisk.**

OpenRecordFile (Apple, C64, C128) VLIR disk

Function: Open an existing VLIR file for access.

Parameters: **r0** FILENAME — pointer to null-terminated name of file (word).

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
curKBlkno current directory.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:
- **x** error ($00 = no error).
- **r1** block number (Commodore track/sector) of directory block containing entry.
- **r5** pointer into **diskBlkBuf** to start of directory entry.

Alters:
- **fileHeader** buffer contains VLIR index table.
- **usedRecords** number of records in file that are currently in use.
- **curRecord** current record set to 1 by default or -1 ($ff) if there are no records in the file.
- **fileWritten**† set to **FALSE** to indicate VLIR file has not been written to.
- **fileSize** total number of disk blocks used in file (includes index block, GEOS file header, and all records).
- **dirEntryBuf** directory entry of VLIR file.

Apple:
- **fileBytes** total number of bytes in file (as picked up from last from bytes 254, 255, 511, and 512 of the master index block.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4–r6.**

Description: Before accessing the data in a VLIR file, an application must call **OpenRecordFile**. **OpenRecordFile** searches the current directory for *FILENAME* and, if it finds it, loads the index table into **fileHeader**. **OpenRecordFile** initalizes the GEOS VLIR variables (both local and global) to allow other VLIR routines such as **WriteRecord** and **ReadRecord** to access the file. Only one VLIR file may be open at a time. A previously opened VLIR file should be closed before opening another.

If an application passes a *FILENAME* of a non-VLIR file, **OpenRecordFile** will return a **STRUCT_MISMATCH** error.

Note: An application can create an empty VLIR file with **SaveFile**.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk.

Apple: Once a VLIR file is opened, the current directory may be changed without affecting access to the file because the index information is kept in **fileHeader**.

Example:

See also: **CloseRecordFile, UpdateRecordFile.**

OutputByte (Apple) — card driver

Function: Put a byte of output data to the interface card.

Parameters: **y** DATA — single ouput byte to send to card (byte)

Returns: **x** STATUS — card error code; $00 = no error (byte)

Destroys: **a, y.**

Description: **OutputByte** sends a byte of data to the card. The ready-for-output flag may be checked with **StatusCard** prior to calling **OutputByte.**

Note: **OutputByte** must be called after an **OpenCard** and before a **CloseCard.**

Most card drivers always return **NO_ERROR** from from **OutputByte** due to a lack of memory to properly handle complex error checking.

Example:

See also: **InputByte, StatusCard.**

Panic (Apple, C64, C128) internal

Function: Display "system error" dialog box.

Parameters: Apple & C64:
top word on stack is the system error address+2.

C128:
top eight bytes on stack are unused, next word on stack is the system error address+2

Returns: Never returns.

Description: **Panic** puts up a system error dialog box. It is usually not called directly by an application. Usually the global GEOS variable **BRKVector** will contain the address of this routine When GEOS encounters a **brk** (opcode: $00) instruction in memory, it jumps indirectly through **BRKVector** with system-specific status values on the stack. This usually results in a system error dialog box. The hex address in the dialog box is the address of the offending **brk** instruction.

An application that patches into **BRKVector** processes **brk** instructions on its own may need to simulate the normal GEOS course of events by performing a **jmp Panic**.

Although this is not a typical use, an application can use **Panic** as a means of communicating fatal error messages. This may be useful in a beta-test version of a software product, for example.

Example:

```
;FatalError:
;use Panic to send a fatal error message to the user
;
;Pass:
;     r0       Error number

FatalError:
      IncW     r0          ;add 2 to error number
      IncW     r0          ;to compensate for Panic

.if   (C64 || APPLE)       ;apple & c64 only expect an address
      PushW    r0          ;push error number onto stack

.else;(C128)               ;128, however, expects all kinds of internal
                           ;machine-state information (10 bytes total) on the
                           ;stack. It ignores all but the bottomost word.
      ldx      #5-1        ;place 5 words (10 bytes) total onto stack
10$:
      PushW    r0          ;push error number onto stack
      dex                  ;(use error number repeatedly as dummy value)
      bne 10$              ;loop until all done.
.endif

      jmp      Panic       ;go put up the panic dialog box
```

See also: **DoDlgBox.**

PointRecord (Apple, C64, C128) VLIR disk

Function: Make a particular record the current record.

Parameters: **a** RECORD — record number to make current (byte).

Uses: **fileHeader** index table checked to establish whether record exists.

Returns:

x		error ($00 = no error).
y		if no error, then a value of $00 here means record is allocated but not in use (has no data blocks).
a		new current record number.
Commodore:		
r1	$0000	record is not allocated
	$ff00	record is allocated but not in use (has no data blocks); this information is already flaged in **y**.
	other	track/sector of first data block in record.
Apple:		
r1	$0000	record is not allocated
	$ffff	record is allocated but not in use (has no data blocks); this information is already flaged in **y**.
	other	block number of the index block that corresponds to the record (Apple).

Alters: **curRecord** new record number.

Destroys: nothing.

Description: **PointRecord** makes *RECORD* the current record so that a subsequent call to **ReadRecord** or **WriteRecord** will operate with *RECORD*. VLIR records are numberd zero through **MAX_VLIR_RECS**-1.

If the record does not exist (you pass a record number that is larger than the number of currently used records), then **PointRecord** returns an **INV_RECORD** (invalid record) error.

Example:

See also: **NextRecord, PreviousRecord.**

PosSprite (Apple, C64, C128) sprite

Function: Positions a sprite at a new GEOS (x,y) coordinate.

Parameters:
r3L SPRITE — sprite number (byte).
r4 XPOS — x-position of sprite (word).
r5L YPOS — y-position of sprite (byte).

Returns: nothing.

Alters:

mobNxpos	(64 and 128 only) sprite x-position (lower 8-bits)
msbNxpos	(64 and 128 only) sprite x-position (bit 9).
reqXposN	(Apple only) sprite x-position.
mobnypos	(all versions)

where N *is the number of the sprite being positioned.*

Destroys: **a, x, y, r6**

Description: **PosSprite** positions a sprite using GEOS coordinates (*not* C64 hardware sprite coordinates). **PosSprite** does not affect the enabled/disabled status of a sprite, it only changes the current position.

Although there are eight sprites available, an application should only directly position sprites #2 through #7 with **PosSprite**. Sprite #0 (the mouse pointer) should not be repositioned (except, maybe through **mouseXPos** and **mouseYPos**), and sprite #1 (the text cursor) should only be repositioned with **stringX** and **stringY**.

C64: The positions are translated to C64 hardware coordinates and then stuffed into the VIC chip's sprite positioning registers. The C64 hardware immediately redraws the sprite at the new position.

C128: The positions are translated to C64 hardware coordinates and then stuffed into the VIC chip's sprite positioning registers. This data is used by the VIC chip in 40-column mode and by the soft-sprite handler in 80-column mode. In 80-column mode, the sprite is not visually updated until the next time the soft-sprite handler gets control.

Apple: The y-position is stuffed, unaltered, into **mobnypos** (simulated hardware register), and the x-position is stuffed, unaltered, into **reqXposn**. The sprite is not visually updated until the next time the soft-sprite handler gets control.

Example:

See also: **DrawSprite, GetSpriteData, EnablSprite, DisablSprite, InitSprite.**

PreviousRecord (Apple, C64, C128) VLIR disk

Function: Makes the previous record the current record.

Parameters: none.

Uses: **fileHeader** index table checked to establish whether record exists.

Returns:
x error ($00 = no error).
y if no error, then a value of $00 here means record is empty.
a new current record number.

Commodore:
r1 track/sector of first data block in record.

Apple:
r1 block number of record's index block.

Alters: **curRecord** new record number.

Destroys: nothing.

Description: **PreviousRecord** makes the current record minus one the new current record. A subsequent call to **ReadRecord** or **WriteRecord** will operate with this record.

If the record does not exist, then **PreviousRecord** returns an INV_RECORD (invalid record) error.

Example:

See also: **PointRecord, NextRecord.**

PrintASCII (Apple, C64, C128) — printer driver

Function: Send ASCII string to the printer.

Parameters:

Commodore

r0 PRINTDATA — pointer to null-terminated ASCII string (word).

r1 WORKBUF — pointer to a 640-byte work buffer for use by the printer driver (word). This is the same buffer that was established in **StartASCII** and must stay intact throughout the entire page.

Apple

r0 PRINTDATA — pointer to null-terminated ASCII string (word).

Uses:

Apple

RWbank the memory bank that the *PRINTDATA* buffer is in. Valid settings are **MAIN** and **AUX**.

Returns:

Commodore

nothing.

Apple

x STATUS — printer error code; $00 = no error.

Destroys:

Commodore

assume **a, x, y, r0–r15.**

Apple

assume **a, y, r1–r4.**

Description: **PrintASCII** sends a null-terminated ASCII string to the printer. The application must call **StartASCII** before sending ASCII data to the printer with **PrintASCII**. It is the job of the application to keep track of the number of possible lines per page and call **StopPrint** to formfeed when necessary (or desired).

In order to begin printing on the next line, the string must contain a **CR** character to signify a carriage return. A **NULL** character marks the end of the string.

C64 & C128: The data passed in *PRINTDATA* is in regular ASCII format (not Commodore ASCII). The text is printed using the printer's standard character set. Some printer drivers allow switching the printer into high-quality print mode with **SetNLQ**. Commodore GEOS printer drivers are set to print 80 characters per line and 66 lines per page.

Apple: The data passed in *PRINTDATA* is in regular ASCII format. The text is printed using the printer's standard character set unless some other option has been selected with the **SetMode** command. The page width and height can be determined with **GetMode**. Because a new line is not started until a **CR** character is encountered an **NULL** can be inserted mid-string. When **PrintASCII** encounters it and returns, the application can call **SetMode** to change the modes mid-line.

Example:

See also: **PrintASCII, StartPrint, StopPrint, InitForPrint.**

PrintBuffer (Apple, C64, C128) printer driver

Function: Print one cardrow (eight lines) of graphics data.

Parameters:

Commodore

r0 PRINTDATA — pointer to 640 bytes of graphic data in Commodore card format (8x8 pixel blocks). This is one row of 80 cards, which amounts to eight lines of printer data (word).

r1 WORKBUF — pointer to the 1,920-byte work buffer established with **StartPrint** (word).

r2 COLRDATA — pointer to 80 bytes of Commodore card color data (40-column screen format) for the cardrow; pass $0000 for normal black and white printing (word).

Apple

r0 PRINTDATA — pointer to 640 bytes of graphic data in linear bitmap format. This is eight 640-bit rows of printer data (word).

r1L LF_SUPPRESS — set to **TRUE** to suppress automatic linefeed after printing buffer (for overlayed printing); normally set to **FALSE**.

Uses:

Apple

RWbank the memory bank that the *PRINTDATA* buffer is in. Valid settings are **MAIN** and **AUX**.

Returns:

Commodore

nothing.

Apple

x STATUS — printer error code; $00 = no error.

Destroys:

Commodore

assume **a, x, y, r0–r15**.

Apple

assume **a, y, r0–r4**.

Description: **PrintBuffer** prints eight lines of graphic data on the printer. The maximum width of each line is determined by the capabilites of the printer and its driver. 640 dots per line is standard, but some printers and drivers handle less. The application can determine the capabilities of the printer with a call to **GetDimensions** (Commodore) or **GetMode** (Apple). If the printer cannot handle the full 640 dots, **PrintBuffer** will ignore any pixels at the end of each line.

The application must call **StartPrint** before sending graphic data to **PrintBuffer**. It is also the job of the application to keep track of the number of possible cardrows per page and call **StopPrint** to formfeed when necessary.

C64 & C128: The data passed in *PRINTDATA* is in Commodore card format, where data is stacked into 8x8-pixel blocks. Graphic printer data can be built-up directly on the 40-column graphics screen using GEOS routines and sent directly to the printer (calculating the address using **GetScanLine**). Because one printer cardrow is

equivalent to two screen cardrows the full 640-dot printer cardrow can be created using two sequential screen cardrows.The sequential memory organization of the 40-column screen wraps the end of one screen cardrow around to the beginning of the next screen cardrow. In the 80-column mode of GEOS 128, one screen line is equivalent to one printer line. However, the data must first be converted from linear bitmap format into card format (a simple operation). Also, since the foreground screen can only be accessed indirectly through the VDC chip, the printer data is usually built-up in the background screen buffer.

Apple: The data passed in *PRINTDATA* is in linear bitmap format, where the first 80 bytes represent the 640 bits that are printed on the first line, the next 80 bytes represent the 640 bits on the next line, and so on to fill out eight lines. Printer data can be created using the GEOS graphics routines, then converted from Apple internal screen format to linear bitmap format using **GetLine** and **GetBackLine.**

With the proper printer and driver, color output can be created by setting making three passes per line, each time printing in a different color (cyan, yellow, magenta). The first color is set with **SetMode** and **PrintBuffer** is called with the *LF_SUPPRESS* flag set to **TRUE**, which will return the print carriage without performing an linefeed. The next color can then be printed on top of what is already there. When calling **PrintBuffer** for the third time (the last color), set the *LF_SUPPRESS* flag to **FALSE**, which will carriage return and linefeed.

Example:

See also: **PrintASCII, StartPrint, StopPrint, InitForPrint.**

PromptOff (Apple, C64, C128) text/keyboard

Function: Turn off the prompt (remove the text cursor from the screen).

Parameters: none.

Alters: **alphaFlag** (($c0 & (**alphaFlag** & $40) | PROMPT_DELAY),*where PROMPT_DELAY = 60.*

Destroys: **a, x, r3L**

Description: **PromptOff** removes the text prompt from the screen. To ensure the prompt will remain invisible until a subsequent call to **PromptOn**, interrupts must be disabled before calling **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
```

Example:

See also: **InitTextPrompt, PromptOn.**

PromptOn
~~PromptOff~~

PromptOn (Apple, C64, C128) text/keyboard

Function: Turn on the prompt (show the text cursor on the screen).

Parameters: none.

Uses:

stringX	cursor x-position (word).
stringY	cursor y-position (byte).

Alters:

alphaFlag	(($c0 & (**alphaFlag** | $40) | PROMPT_DELAY),*where PROMPT_DELAY = 60.*

Destroys: **a, x, r3L.**

Description: **PromptOn** makes the text prompt visible and active at the position specified by **stringX** and **stringY**. The prompt will flash once every second (*PROMPT_DELAY*). If **stringX** or **StringY** are changed, the cursor will repositioned automatically the next time the cursor flashes. To make the update immediate, call **PromptOn**. Before **PromptOn** is called for the first time, **InitTextPrompt** should be called.

See also: **InitTextPrompt, PromptOff.**

Example:

PurgeTurbo (C64, C128) — very low-level disk

Function: Completely deactivate and remove disk drive turbo code from current drive, returning to standard Commodore DOS mode.

Parameters: none.

Uses: **curDrive** currently active disk drive.

Returns: **x** error ($00 = no error).

Destroys: **a, y, r0–r3.**

Description: **PurgeTurbo** deactivates and removes the turbo software from the current drive, returning control of the device to the disk drive's internal ROM software. This allows access to normal Commodore DOS routines. An application may want to access the Commodore DOS to perform disk functions not offered by the GEOS Kernal such as formatting.

Apple: Apple GEOS has no **PurgeTurbo** equivalent.

See also: **EnterTurbo, ExitTurbo.**

PutBlock (Apple, C64, C128) — low-level disk

Function: General purpose routine to write a block to disk with verify.

Parameters:

r4 BUFFER — address of buffer to get block from; must be at least **BLOCKSIZE** bytes (word).

Commodore:
r1L TRACK — valid track number (byte).
r1H SECTOR — valid sector on track (byte).

Apple:
r1 BLOCK — ProDOS block number (word).

Uses:

curDrive currently active disk drive to write to.

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank bank *BUFFER* is in (**MAIN** or **AUX**).
numDiskRetries number of times to attempt rewrite if verify fails. If $00, no verify will be performed.

Returns:

x error ($00 = no error).
r1, r4 unchanged

Destroys: **a, y.**

Description: **PutBlock** writes a block from *BUFFER* to the disk. **PutBlock** is useful for implementing disk utility programs and new file structures.

C64 & C128: **PutBlock** is a higher-level version of **WriteBlock**. It calls **InitForIO**, **EnterTurbo**, **ReadBlock**, and **DoneWithIO**. If an application needs to write many blocks at once, **WriteBlock** may offer a faster solution. If the disk is shadowed, **PutBlock** will also write the data to the shadow memory.

Apple: **PutBlock** provides the lowest-level block access to a ProDOS compatible device. It uses the ProDOS device driver **WRITE** block command directly. Apple GEOS, for this reason, does not have a **WriteBlock** equivalent.

Example:

See also: **GetBlock, WriteBlock, BlkAlloc.**

PutChar (Apple, C64, C128) text

Function: Process a single character code (both escape codes and printable characters).

Parameters:
- **a** CHAR — character code (byte).
- **r11** XPOS — x-coordinate of left of character (word).
- **r1H** YPOS — y-coordinate of character baseline (word).

Uses:
- **dispBufferOn** display buffers to direct output to.
- **currentMode** character style.
- **leftMargin** left margin to contain character.
- **rightMargin** right margin to contain characters.

*(following set by **LoadCharSet** and **LoadAuxSet**).*

- **curHeight** height of current font.
- **baselineOffset** number of pixels from top of font to baseline.
- **cardDataPntr** pointer to current font image data.
- **curIndexTable** pointer to current font index table data.
- **curSetWidth** pixel width of font bitstream in bytes.

Returns:
- **r11** x-position for next character.
- **r1H** y-position for next character (usually unchanged).

Destroys:

Commodore:
a, x, y, r1L, r2–r10, r12, r13

Apple:
a, x, y, r1L, r2

Description: **PutChar** is the basic character handling routine. If the character code is less than 32, **PutChar** will look-up a routine address in an internal jump table to process the escape code. Only send implemented escaped codes to **PutChar**.

If the character code is 32 or greater, **PutChar** treats it as a printable character. First it establishes the printed size of the character with any style attributes (**currentMode**) then checks the character position against the bounds in **leftMargin** and **rightMargin**. If the left edge of the character will fall to the left of **leftMargin**, then the width of the character is added to the x-position in **r11** and **PutChar** vectors through **StringFaultVec**. If the right edge of the characer will fall to the right of **rightmargin**, then **PutChar** vectors through **stringFaultVec** without altering the x-position. The character is not printed in either case.

Assuming no margin fault, **PutChar** will print the character to the screen at the desired position. Any portion of the character that lies above **windowTop** or below **windowBottom** will not be drawn.

PutChar cannot be used to directly process multi-byte character codes such as **GOTOX** or **ESC_GRAPHICS** unless **r0** is maintained as a string pointer when **PutChar** is called (as it is in **PutString**). See **PutString** for more information.

See also: **SmallPutChar, PutString, PutDecimal.**

PutDecimal (Apple, C64, C128) text

Function: Format and print a 16-bit positive integer.

Parameters:
- **a** FORMAT — formatting codes (byte) — see below.
- **r0** NUM — 16-bit integer to convert and print (word).
- **r11** XPOS — x-coordinate of leftmost digit (word).
- **r1H** YPOS — y-coordinate of baseline (word).

Uses: same as **PutChar.**

Returns:
- **r11** x-position for next character.
- **r1H** unchanged.

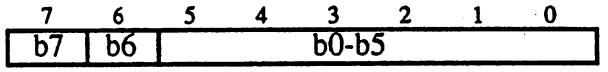
Destroys:

Commodore:
a, x, y, r0, r1L, r2–r10, r12, r13

Apple:
a, x, y, r0, r1L, r2

Description: **PutDecimal** converts a 16-bit positive binary integer to ASCII and sends the result to **PutChar.** The number is formatted based on the FORMAT parameter bytes in the **a**-registers as follows:

FORMAT:



- b7 justification: 1 = left; 0 = right.
- b6 leading zeros: 1 = suppress; 0 = print.
- b5-b0 field width in pixels (only used if right justifying).

The following constants may be used:

SET_LEFTJUST
SET_RIGHTJUST
SET_SUPPRESS
SET_NOSUPPRESS

NOTE: The maximum 16-bit decimal number is 65535 ($ffff), so the printed number will never exceed five characters.

Example:

See also: **PutChar.**

PutDirHead (Apple, C64, C128) mid-level disk

Function: Write directory header to disk. Commodore GEOS also writes out the BAM.

Parameters: none.

Uses:
curDrive
curDirHead current directory header.

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.
dir2Head† (BAM for 1571 and 1581 drives only)
dir3Head† (BAM for 1581 drive only)

†*used internally by GEOS disk routines; applications generally don't use.*

Apple:
curKBlkno current key block to write directory header to.

Returns: **x** error ($00 = no error).

Commodore:
r4 pointer to **curDirHead.**

Destroys: Commodore:
a, y, r1.

Apple:
a, y, r1, r4.

Description: **PutDirHead** writes the directory header to disk from the buffer at **curDirHead.** Because of differences in the Commodore and Apple file systems, this can mean different things. Commodore GEOS writes out the full directory header block, including the BAM (block allocation map). Apple GEOS only writes out the 39-byte ProDOS directory header for the current directory's key block.

C64 & C128: GEOS disks, like the standard Commodore disks upon which they are based, have one directory header. The directory header occupies one full block on the disk. The Commodore directory header contains information about the disk, such as the location of the directory blocks, the disk name, and the GEOS version string (if a GEOS disk). The Commodore directory header also contains the disk BAM, which flags particular sectors as used or unused.

PutDirHead calls **PutBlock** to write out the directory header block from the buffer at **curDirHead.** The directory header block contains the directory header and the disk BAM (block allocation map). Applications that are working with the mid- and low-level GEOS disk routines may need to call **PutDirHead** to update the BAM on the disk with the BAM in memory. Many useful, mid-level GEOS routies, such as **BlkAlloc,** only update the BAM in memory (for speed and ease of error recovery). When a new file is written disk, GEOS allocates the blocks in the in-memory BAM, writes the blocks out using the track sector table, then, as the last operation, calls **PutDirHead** to write the new BAM to the disk. An

application that uses the mid-level GEOS routines to build its own specialized disk file functions will need to keep track of the status of the BAM in memory, writing it to disk as necessary.

It is important that the BAM in memory not get overwritten by an outdated BAM on the disk. Applications that manipulate the BAM in memory (or calls GEOS routines that do so), must be careful to write out the new BAM before calling a routine that might overwrite it. Routines that call **GetDirHead** include **OpenDisk, SetGEOSDisk,** and **OpenRecordFile.**

GEOS VLIR routines set the global variable **fileWritten** to **TRUE** to signal that the VLIR file has been written to and that the BAM in memory is more recent than the BAM on the disk. **CloseRecordFile** checks this flag. If **fileWritten** is **TRUE, CloseRecordFile** calls **PutDirHead** to write out the new BAM.

Apple: Apple GEOS disks, like the ProDOS disks upon which they are based, have a directory header for each directory. The header for a root directory is called a *volume directory header* and the header for a subdirectory is called a *subdirectory header*. These directory headers are 39-byte structures defined by ProDOS. They contain such information as the directory (or volume) name, the date stamp, and any read/write access flags. The directory header *does not* contain the VBM (volumen bit map, the Apple equivalent of a BAM).

PutDirHead first reads in the key block of the current directory (pointed at by **curKBlkno**) and copies the 39-byte directory header information from **curDirHead** to the key block, then rewrites the key block to disk. Only these 39-bytes in the key block are changed.

Since Apple GEOS does not store the allocation map (VBM) in the directory header like Commodore GEOS, it not necessary to be as careful about rereading the directory header. **PutDirHead** only needs to be called when an application explicitly wants to change the directory header, which few applications (outside of the deskTop) will ever do. For information on writing the Apple VBM, refer to **PutVBM.**

Example:

See also: **GetDirHead, GetVBM, PutVBM.**

PutScreenLine (Apple) graphics

Function: Copies data from the application's buffer, in internal format, to a byte-aligned horizontal line on the screen.

Parameters:

r0	DATA — address of buffer to copy data from (word).
r1H	XINDX — byte in line to begin with (seven-bit Apple screen byte) (byte).
r2L	XWIDTH — width in bytes of line (seven-bit Apple screen bytes) (byte).
r1L	Y — y-coordinate of line (byte).

*where (XINDX*7,Y) and (XINDX*7+XWIDTH,Y) define the endpoints of the line.*

Uses: **dispBufferOn:**

bit 7 — get data from foreground screen if set.
bit 6 — get data from background buffer if set.
If both bits are set, foreground screen is used.

Returns: **r0** address of byte following last byte in the buffer.

Destroys: **a, x, y, r0, r1H, r2L, r5–r6**

Description: **PutScreenLine** copies bytes directly from the specified buffer to the screen memory. The screen is treated as a contiguous block of bytes even though, in actuality, alternate bytes lie in different memory banks.

Bytes are copied from the buffer pointed to by **r0** and stored in screen memory beginning at the byte-index into the line stored in **r1H**.

Note: No clipping at the screen edge is performed; the values passed are assumed to lie entirely on one screen line.

Example:

See also: **GetScreenLine.**

PutString, i_PutString (Apple, C64, C128) text

Function: Print a string to the screen.

Parameters: Normal:
r0 STRING — pointer to string data (word).
r11 XPOS — x-coordinate of left of first character (word).
r1H YPOS — y-coordinate for character baselines (word).

Inline:
data appears immediately after the **jsr i_PutString**

.word	XPOS	x-coordinate.
.byte	YPOS	y-coordinate.
.byte	STRINGDATA	null terminated string (no length limit)

Uses: same as **PutChar.**

Returns: **r11** x-position for next character.
r1H y-position for next character (usually unchanged).

Destroys: Commodore
a, x, y, r1L, r2–r10, r12, r13

Apple
a, x, y, r1L, r2

Calls: **PutChar.**

Description: **PutString** passes a full string of data to **PutChar** a character at a time. It maintains **r0** as a running pointer into the string and so supports multi-byte escape codes such as **GOTOXY.**

If a character exceeds one of the margins, **PutChar** will vector through **stringFaultVec** as appropriate. **r0, r11,** and **r1H** will all contain useful values (current string pointer, x-position, and y-position, respectively). For more information, refer to "String Faults (Left or Right Margin Exceeded)" in Chapter XX.

Basic operation of **PutString:**

```
PutString:
5$:
     ldy     #0                ;use zero offset
     lda     (r0),y            ;get character
     beq     10$               ;exit if NULL terminator
     jsr     PutChar           ;otherwise process char.
     IncW    r0                ;move to next byte in string
     bra     5$                ;and loop through again
10$:
     rts                       ;exit
```

C64 & C128: Unless a special string fault routine is placed in **stringFaultVec** prior to calling **PutString,** a margin fault will be ignored and **PutString** will attempt to print the next character.

Apple: If **stringFaultVec** contains $0000 when **PutString** is called, Apple GEOS installs a temporary string fault routine that fast-forwards through the string to the null-terminator whenever a fault is generated. **stringFaultVec** is restored with a $0000 when **PutString** exits.

Example:

See also: **PutChar, GraphicsString.**

PutVBM (Apple) — mid-level disk

Function: Flush the currently cached VBM block, writing it out to disk.

Parameters: none.

Uses:
curDrive
curVBlkno† block to write cache to.

Returns: **x** error ($00 = no error).

Alters: **VBMchanged**† set to **FALSE**; indicates current cache matches disk.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4.**

Description: **PutVBM** writes the currently cached VBM block to its proper spot on the disk. This cache is inaccessible to applications but is used indirectly by routines such as **SetNextFree**. **PutVBM** is called by high-level GEOS routines to automatically flush the cache, bringing the copy of the VBM on the disk up-to-date with the copy in memory. Specialized applications that call mid- and low-level disk routines may need to flush the cache manually.

Apple GEOS disk routines cache a single VBM block in an internal memory buffer. This speeds up disk operations by allowing the VBM to be accessed quickly during multiple-sector disk operations. This parallels the way Commodore GEOS buffers the BAM in the directory header. In fact, wherever a Commodore GEOS application calls **PutDirHead** to update the BAM on the disk, an Apple GEOS application can usually substitute a call to **PutVBM**.

But there are differences between the Commodore buffering and the Apple buffering. Because higher-density storage devices (a hard disk, for example) will have more than one VBM block, the entire VBM cannot be stored in memory. The Apple GEOS routines that manage the disk cache will automatically flush the currently cached VBM block before loading in another. But if another VBM block is never loaded, the cache may not necessarily get flushed. For example, some mid-level routines, such as **BlkAlloc** and **AllocateBlock**, do not explicitly flush the cache (just as their Commodore counterparts do not call **PutDirHead**), which may mean that the block in the cache may be more up-to-date than the block on the disk. An application that uses these mid-level routines should flush the cache manually with a calls to **PutVBM**.

The less often the cache is flushed, the faster a multiple-sector disk operation will run. GEOS VLIR routines, for example, do not flush the cache until **CloseRecord** is called. At the very least, the application must be careful to flush the VBM cache before calling any other routine that might call **GetVBM** because **GetVBM** will overwrite the cached block before it is used to update the disk copy. **GetVBM** is called by routines such as **OpenDisk** and **BlkAlloc**.

Note: **PutVBM** will write out the cached VBM block even if **VBMchanged** is **FALSE**.

Example:

See also: **GetVBM, GetDirHead, PutDirHead.**

ReadBackLine (Apple) graphics

Function: Translates a screen line in the background buffer from internal format to linear bitmap format.

Parameters:
r0 DATA — address of buffer to copy data to (word).
r1L Y — y-coordinate of line (byte).

Returns: **r0** address of byte following last byte in the buffer.

Destroys: **a, x, y, r0**

Description: **ReadBackLine** reads bytes directly from the background buffer and converts them into linear bitmap format in the application's buffer. The buffer pointed to by *DATA* must be at least **SC_PIX_WIDTH/7** = 80 bytes long.

Example:

ReadBlock (C64, C128) — very low-level disk

Function: Very low-level read block from disk.

Parameters:
- **r1L** TRACK — valid track number (byte).
- **r1H** SECTOR — valid sector on track (byte).
- **r4** BUFFER — address of buffer of **BLOCKSIZE** bytes to read block into (word).

Uses:
- **curDrive** currently active disk drive.
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns: **x** error ($00 = no error).

Destroys: **a, y.**

Description: **ReadBlock** reads the block at the specified *TRACK* and *SECTOR* into *BUFFER*. If the disk is shadowed, **ReadBlock** will read from the shadow memory. **ReadBlock** is a pared down version of **GetBlock**. It expects the application to have already called **EnterTurbo** and **InitForIO**. By removing this overhead from **GetBlock**, multiple sector reads can be accomplished without the redundant initialization. This is exactly what happens in many of the higher-level disk routines that read multiple blocks at once, such as **ReadFile**.

ReadBlock is useful for multiple-sector disk operations where speed is an issue and the standard GEOS routines don't offer a decent solution. **ReadBlock** can function as the foundation of specialized, high-speed disk routines.

Apple: Apple GEOS has no **ReadBlock** equivalent. Use **GetBlock** instead.

Example:

```
;Read sector from disk into diskBlkBuf. Demonstrates use of
;very-low level disk primitives.
;
;Pass:
;     track    track number
;     sector   sector on track
;
;Returns:
;     x             error code
;
MyGetBlock:
     LoadW    r4,#diskBlkBuf      ;where to get data from
     MoveB    r1L,track           ;track number
     MoveB    r1H,sector          ;sector number
     jsr      EnterTurbo          ;go into turbo mode
     txa                          ;set status flags
     bne      99$                 ;branch if error found
     jsr      InitForIO           ;prepare for serial I/O
     jsr      ReadBlock           ;primitive read block
     jsr      DoneWithIO          ;restore after I/O done
99$:
     rts                          ;exit
```

See also: **GetBlock, WriteBlock, VerWriteBlock.**

ReadByte (Apple, C64, C128) — mid-level disk

Function: Special version of **ReadFile** that allows reading a chained list of blocks a byte at a time.

Parameters: *on initial call only:*

r4 BLOCKBUF — pointer to temporary buffer of **BLOCKSIZE** bytes for use by **ReadByte**, usually a pointer to **diskBlkBuf** (Apple GEOS: must be in main memory) (word).

r5 $0000 (word).

Commodore:

r1 START_TRSC — track/sector of first data block (word).

Apple:

r1 INDXBLKPTR — pointer to index buffer that contains index block of chain to read (must be in main memory) (word).

Uses: **curDrive**

Commodore:

curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:

RWbank bank *BUFFER* is in (**MAIN** or **AUX**).

Returns:

a byte returned

x error ($00 = no error).

r1, r4, r5 contain internal values that must be preserved between calls to **ReadByte**.

Destroys: **y**.

Description: **ReadByte** allows a chain of blocks on the disk to be read a byte at a time. The first time **ReadByte** is called, **r1**, **r4**, and **r5** must contain the proper parameters. When **ReadByte** returns without an error, the **a** register will contain a single byte of data from the chain. To read another byte, call **ReadByte** again. Between calls to **ReadByte**, the application must preserve **r1**, **r4**, **r5**, and the data area pointed to by *BLOCKBUF*. Apple GEOS applications must also preserve the 512-byte index block pointed to by *INDXBLKPTR* and the status of **RWbank**.

ReadByte loads a block into *BLOCKBUF* and returns a single byte from the buffer at each call. After returning the last byte in the buffer, **ReadByte** loads in the next block in the chain and starts again from the beginning of *BLOCKBUF*. This process continues until there are no more bytes in the file. A **BFR_OVERFLOW** error is then returned.

ReadByte is especially useful for displaying very large bitmaps with **BitOtherClip** and **NewBitOtherClip**.

C64 & C128: Reading a chain a byte at a time under Commodore GEOS involves finding the first data block and passing its track/sector to **ReadFile**. The track/sector of the first data block in a sequential file is returned in **r1** by **GetFHdrInfo**. The first data block of a VLIR record is contained in the VLIR's index table.

Apple: Because the ProDOS filing system is different from the Commodore filing system, different steps are involved in using **ReadByte**. The Apple GEOS version of GEOS expects an entire index block for the chain, not merely an initial track/sector as the Commodore version does. The index block for a sequential file is returned in **r1** by **GetFHdrInfo**, but it must be loaded into memory with **GetBlock**. The index block for a VLIR record is contained in the VLIR's index table.

Example:

See also: **GetFile, WriteFile, ReadRecord.**

ReadClock (Apple) — clock driver

Function: Causes the clock driver to update the global GEOS clock variables with the current time. Called during MainLoop Level by GEOS.

Parameters: none.

Returns: nothing.

Alters:

year	updated from clock device
month	updated from clock device
day	updated from clock device
hour	updated from clock device
minutes	updated from clock device
seconds	updated from clock device
alarmOn	if bit 7 set and time for alarm to trigger, then bit 7 is cleared and bit 6 is set (byte is shifted right once).

Destroys: assume **x, y, r0–r15.**

Description: GEOS calls **ReadClock** during MainLoop Level. **ReadClock** is expected to update the global clock variables with the current time. It also checks for an alarm trigger and sets **alarmOn** appropriately. (Depending on the clock, an alarm trigger may be flagged automatically in hardware or may have be generated in software by comparing the current time with an internal alarm time.)

See also: **ClockInt, SetTimeDate, SetAlarm, ResetAlarm.**

ReadFile
~~ReadBlock~~

ReadFile (Apple, C64, C128) — mid-level disk

Function: Read a chained list of blocks into memory.

Parameters:
r7 BUFFER — pointer to buffer where data will be read into (word).
r2 BUFSIZE — size of buffer Commodore version can read up to 32,258 bytes (127 Commodore blocks); Apple version can read up to the maximum two-byte number that can be passed in **r2**: 65,535 ($ffff) bytes (word).

Commodore:
r1 START_TRSC — track/sector of first data block (word).

Apple:
r1 INDXBLOCK — block number of index block for chain (word).

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank bank *BUFFER* is in (**MAIN** or **AUX**).

Returns:
x error ($00 = no error).
r7 pointer to last byte read into *BUFFER* plus one.

Commodore:
r1 if **BFR_OVERFLOW** error returned, contains the track/sector of the block that, had it been copied from **diskBlkBuf** to the application's buffer space, would have exceeded the size of *BUFFER*. The process of copying any extra data from **diskBlkBuf** to the end of *BUFFER* is left to the application. The data starts at **diskBlkBuf+2**. If no error, then **r1** is destroyed.

r5L byte index into **fileTrScTab** of last entry (last entry = **fileTrScTab** plus value in **r5**).

Apple:
r1 if **BFR_OVERFLOW** error returned, contains the number of the block that would have overwritten the end of *BUFFER* had the entire block been read directly into memory. *BUFFER* is filled with as much data from the block as will fit. If no error, then **r1** is destroyed.

Alters:
Commodore:
fileTrScTab As the chain is followed, the track/sector pointer to each block is added to the file track/sector table. The track and sector of the first data block is added at **fileTrScTab+2** and **fileTrScTab+3**, respectively, because the first two bytes (**fileTrScTab+0** and **fileTrScTab+1**) are reserved for the GEOS file header track/sector.

Apple:
INDEXBLOCKBUF index block for chain (in auxiliary memory; see below for information on accessing this buffer).

Destroys: Apple:
y, (**r1**), **r3–r4** (see above for **r1**).

Commodore:
y, (**r1**), **r2–r4** (see above for **r1**).

Description: **ReadFile** reads a chain of blocks from the disk into memory at *BUFFER*. Although the name implies that it reads "files" into memory, it actually reads a chain of blocks and doesn't care whether this chain is a sequential file or a VLIR record — **ReadFile** merely reads blocks until it encounters the end of the chain or overflows the memory buffer.

ReadFile can be used to load VLIR records from an unopened VLIR file. **geoWrite**, for example, loads different fonts while another VLIR file is open by looking at all the font file index tables and remembering the index information for records that contain font data. When a VLIR document file is open, **geoWrite** can load a different font by passing one of these saved values in **r1** to **ReadFile**. **ReadFile** will load the font into memory without disturbing the opened VLIR file.

For reading a file when only the filename is known, use the high-level **GetFile**.

C64 & C128: The Commodore filing system links blocks together with track/sector links: each block has a two-byte track/sector forward-pointer to the next sector in the chain (or $00/$ff to signal the end). Reading a chain involves passing the first track/sector to **ReadFile**. The first block contains a pointer to the next block, and so on. The whole chain can be followed by reading successive blocks.

ReadFile reads each 256-byte block into **diskBlkBuf** and copies the 254 data bytes (possibly less in the last block of the chain) to the *BUFFER* area and copies the two-byte track/sector pointer to **fileTrScTab**. This process is repeated until the last block is copied into the buffer or when there is more data in **diskBlkBuf** than there is room left in *BUFFER*.

when there is more data in **diskBlkBuf** than there is room left in *BUFFER*, **ReadFile** returns with a **BFR_OVERFLOW** error without copying any data into *BUFFER*. The application can copy data, starting at **diskBlkBuf+2**, to fill the remainder of *BUFFER* manually.

Because of the limited size of **fileTrScTab** (256 bytes), **ReadFile** cannot load more than 127 blocks of data. (256 total bytes divided by two bytes per track/sector minus two bytes for the GEOS file header equals 127.) 127 blocks can hold 127 * 254 = 32,258 bytes of data.

Apple: Unlike the Commodore filing system, the ProDOS filing system links blocks together with entries in an index block: each entry in the index block holds a two-byte block number. Going through each entry constitutes walking the chain. For

this reason, reading a chain involves passing, not the first data block, but the block number of the index block.

Apple GEOS first reads the index block into the internal buffer **INDEXBLOCKBUF** (in auxiliary memory), then reads in the blocks specified in the index.

Applications cannot directly access the auxiliary buffer **INDEXBLOCKBUF**. However, **MoveAuxData** can be used to copy the block into the appliciation's memory space if access to it is necessary:

```
AUXtoMAIN       == %10000000
MAINtoAUX       == %01000000

      LoadW     r0, #INDEXBLOCKBUF          ;copy from index blk
      LoadW     r1, #diskBlkBuf             ;to temp buffer
      LoadW     r2, #BLOCKSIZE              ;move a full block
      lda       #AUXtoMAIN                  ;copy aux to main
      jsr       MoveAuxData
```

Apple GEOS **ReadFile** could conceivably load up to 65,536 bytes of data into memory. This is of little use, however, because the absence of large, usable memory blocks functionally limits reads to 32K or 64 blocks.

Example:

See also: **GetFile, WriteFile, ReadRecord.**

ReadLink
~~ReadFile~~

ReadLink (C64, C128) — very low-level disk

Function: Read link (first two bytes) from a Commodore disk block

Parameters:
- **r1L** TRACK — valid track number (byte).
- **r1H** SECTOR — valid sector on track (byte).
- **r4** BUFFER — address of buffer of at least **BLOCKSIZE** bytes, usually points to **diskBlkBuf** (word).

Uses: **curDrive** currently active disk drive.

Returns: **x** error ($00 = no error).

Alters: **diskBlkBuf**

Destroys: **a, y.**

Description: **ReadLink** returns the track/sector link from a disk block as the first two bytes in *BUFFER*. The remainder of *BUFFER* (**BLOCKSIZE**-2 bytes) may or may not be altered.

ReadLink is useful for following a multiple-sector chain in order to build a track/sector table. It mainly of use on 1581 disk drives, which walk through a chain significantly faster when only the links are read. Routines such as **DeleteFile** and **FollowChain** will automatically take advantage of this capability of 1581 drives.

Disk drives that do not offer any speed increase through **ReadLink** will simply perform a **ReadBlock.**

Apple: Apple GEOS has no **ReadLink** equivalent; ProDOS blocks are linked by the index block.

Example:

See also: **ReadBlock, FollowChain.**

Note: Does not work in 1541 drivers; use ReadBlock

ReadRecord (Apple, C64, C128) VLIR disk

Function: Read in the current VLIR record.

Parameters:
r7 BUFFER — pointer to start buffer where data will be read into (word).
r2 BUFSIZE — size of buffer: Commodore version can read up to 32,258 bytes (127 Commodore blocks); Apple version can read up to the maximum two-byte number that can be passed in **r2**: 65,535 ($ffff) bytes (word).

Uses:
curDrive
curRecord current record pointer
fileHeader index table holds first block of record.

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank bank *RECDATA* is read into (**MAIN** or **AUX**).

Returns:
x error ($00 = no error).
a **$00** = empty record, no data read.
$ff = record contained data.
r7 pointer to last byte read into *BUFFER* plus one if not an empty record, otherwise unchanged.

Commodore:
r1 if **BFR_OVERFLOW** error returned, contains the track/sector of the block that, had it been copied from **diskBlkBuf** to the application's buffer space, would have exceeded the size of *BUFFER*. The process of copying any extra data from **diskBlkBuf** to the end of *BUFFER* is left to the application. The data starts at **diskBlkBuf+2**. If no error, then **r1** is destroyed.

Apple:
r1 if **BFR_OVERFLOW** error returned, contains the number of the block that would have overwritten the end of *BUFFER* had the entire block been read directly into memory. *BUFFER* is filled with as much data from the block as will fit. If no error, then **r1** is destroyed.

Alters:
Commodore:
fileTrScTab As the chain blocks in the record is followed, the track/sector pointer of each block is added to the file track/sector table. The track and sector of the first block in the record is added at **fileTrScTab+2** and **fileTrScTab+3**. Refer to **ReadFile** for more information.

Destroys:
Apple:
y, (r1),r3–r4 (see above for **r1**).

Commodore:

y, (r1), r2–r4 (see above for **r1**).

Description: **ReadRecord** reads the current record into memory at *BUFFER*. If the record contains more than *BUFSIZE* bytes of data, then a **BFR_OVERFLOW** error is returned.

ReadRecord calls **ReadFile** to load the chain of blocks into memory.

Example:

See also: **WriteRecord, ReadFile.**

ReadScanLine (Apple) graphics

Function: Translates a screen line on the foregorund screen from internal format to linear bitmap format.

Parameters:
r0 DATA — address of buffer to copy data to (word).
r1L Y — y-coordinate of line (byte).

Returns: **r0** address of byte following last byte in the buffer.

Destroys: **a, x, y, r0**

Description: **ReadScanLine** reads bytes directly from the foreground screen and converts them into linear bitmap format in the application's buffer. The buffer pointed to by DATA must be at least **SC_PIX_WIDTH/7** = 80 bytes long.

Example:

RecoverAllMenus (Apple, C64, C128) icon/menu

Function: Removes all menus (including the main menu) from the foreground screen by recovering from the background buffer.

Parameters: none.

Destroys: assume **r0–r15, a, x, y**

Description: **RecoverAllMenus** is a very low-level menu routine which recovers the area obscured by the opened menus from the background buffer. Usually this routine is only called internally by the higher-level menu routines. It is of little use in most applications and is included in the jump table mainly for historical reasons.

RecoverAllMenus operates by loading the proper GEOS registers with the coordinates of the menu rectangles and calling the routine whose address is in **recoverVector** (normally **RecoverRectangle**) repeatedly.

Example:

See Also: **DoPreviousMenu, ReDoMenu, GotoFirstMenu, RecoverMenu.**

RecoverFG (Apple) graphics/utility

Function: Restores a portion of the foreground screen from data saved with **SaveFG**.

Parameters:
r0 FGSTRUCT — pointer to an **FG** data structure (word).
r1 FGDATA — pointer to **SaveFG** data to restore (word).

Returns: Foreground screen restored.

Destroys: **a, x, y, r1–r6.**

Description: **RecoverFG** restores a rectangular area of the foreground screen saved with **SaveFG**.

FGSTRUCT points to an **FG** data structure, which is in the following format:

```
FGSTRUCT1:
     .word    X1          ;pixel x-position of left edge
     .byte    Y1          ;pixel y-position of top edge
     .word    X2          ;pixel x-position of right edge (X1 + WIDTH)
     .byte    HEIGHT      ;pixel height (Y2 - Y1)
```

The **FG** data structure used to restore an area should be the same as the one used to save the area.

Example:

See also: **SaveFG**.

RecoverLine (Apple, C64, C128) graphics

Function: Recovers a horizontal line from the background buffer to the foreground screen.

Parameters:
r3 X1 — x-coordinate of leftmost endpoint (word).
r4 X2 — x-coordinate of rightmost endpoint (word).
r11L Y1 — y-coordinate of line (byte).

where (X1,Y1) and (X2,Y1) define the endpoints of the line to recover.

Returns: nothing

Destroys:
Commodore
a, x, y, r5–r8

Apple
a, x, y

Description: **RecoverLine** recovers the pixels which fall on the horizontal line whose coordinates are passed in the GEOS registers. The pixel values are copied from the background buffer to the foreground screen.

Note: The flags in **dispBufferOn** are ignored; the pixels are always copied to the foreground screen regardless of the value in this variable.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **ImprintLine, HorizontalLine, InvertLine,VerticalLine , DrawLine.**

RecoverMenu (Apple, C64, C128) icon/menu

Function: Removes the current menu from the foreground screen by recovering from the background buffer.

Parameters: none.

Destroys: assume **r0–r15, a, x, y**

Description: **RecoverMenu** is a very low-level menu routine which recovers the rectangular area obscured by the current menu. Usually this routine is only called internally by the higher-level menu routines such as **DoPreviousMenu**. It is of little use in most applications and is included in the jump table mainly for historical reasons.

RecoverMenu operates by loading the proper GEOS registers with the coordinates of the current menu's rectangle and calling the routine pointed to by **recoverVector** (normally **RecoverRectangle**).

Example:

See Also: **DoPreviousMenu, ReDoMenu, GotoFirstMenu, RecoverAllMenus.**

RecoverRectangle, i_RecoverRectangle (Apple, C64, C128) graphics

Function: Recovers the pixels within a rectangular region from the background buffer to the foreground screen.

Parameters: Normal:
r3 X1 — x-coordinate of upper-left (word).
r2L Y1 — y-coordinate of upper-left (byte).
r4 X2 — x-coordinate of lower-right (word).
r2H Y2 — y-coordinate of lower-right (byte).

Inline:
data appears immediately after the **jsr i_RecoverRectangle**
.byte Y1 y-coordinate of upper-left.
.byte Y2 y-coordinate of lower-right.
.word X1 x-coordinate of upper-left.
.word X2 x-coordinate of lower-right.

where (X1,Y1) is the upper-left corner of the rectangular area and (X2,Y2) is the lower-right corner.

Returns: **r2**, **r3**, and **r4** unchanged.

Destroys: Commodore:
a, **x**, **y**, **r5–r8**, **r11L**

Apple:
a, **x**, **y**, **r11L**

Description: **RecoverRectangle** copies the pixels within a rectangular region from the background buffer to the foreground screen by calling **RecoverLine** in a loop.

Note: The flags in **dispBufferOn** are ignored; the pixels are always copied to the foreground screen regardless of the value in this variable.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **ImprintRectangle**, **Rectangle**, **InvertRectangle**.

RecoverSysRam (Apple) internal

Function: Restores the system state after a call to **InitForDialog.**

Parameters: none.

Returns: system restored to its prior state.

Destroys: **a,x, y, r1, r2, r3L, r4.**

Description: **RecoverSysRam** restores the state of GEOS that was saved with **InitForDialog.**

See also: **InitForDialog.**

Rectangle, i_Rectangle (Apple, C64, C128) graphics

Function: Draw a rectangle in the current fill pattern.

Parameters: Normal:
r3 X1 — x-coordinate of upper-left (word).
r2L Y1 — y-coordinate of upper-left (byte).
r4 X2 — x-coordinate of lower-right (word).
r2H Y2 — y-coordinate of lower-right (byte).

where (X1,Y1) is the upper-left corner of the rectangle and (X2,Y2) is the lower-right corner.

Inline:
data appears immediately after the **jsr i_Rectangle**
.byte Y1 y-coordinate of upper-left.
.byte Y2 y-coordinate of lower-right.
.word X1 x-coordinate of upper-left.
.word X2 x-coordinate of lower-right.

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: nothing

Destroys: Commodore:
a, x, y, r5–r8

Apple:
a, x, y

Description: **Rectangle** draws a filled rectangle on the screen as determined by the coordinates of the upper-left and lower-right corners. The rectangle is filled with the current 8x8 (card-sized) fill pattern.

The 8x8 pattern within the rectangle is drawn as if it were aligned to a card boundary: that is, the bit-pattern is synchronized with (0,0), and, since the patterns are 8x8, they are aligned with every eighth pixel thereafter. This allows the patterns in adjacent or overlapping rectangles to line-up regardless of the actual pixel positions.

Rectangle operates by calling **HorizontalLine** in a loop, changing the bit-pattern byte after every line based on the current 8x8 fill pattern.

Note: Because all GEOS coordinates are inclusive, framing a filled rectangle requires either calling **FrameRectangle** *after* calling **Rectangle** (and thereby overwriting the perimeter of the filled area) or calling **FrameRectangle** with (*X1*–1,*Y1*–1) and (*X2*+1,*Y2*+1) as the corner points.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* and*X2* parameters will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will

automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

See also: **FrameRectangle, SetPattern, ImprintRectangle, RecoverRectangle, InvertRectangle.**

ReDoMenu (Apple, C64, C128) icon/menu

Function: Reactivate menus at the current level.

Parameters: none.

Destroys: assume **r0–r15, a, x, y**

Description: **ReDoMenu** is used by the application's menu event handler to instruct GEOS to leave all menus (including the current menu) open when control is returned to **MainLoop**. **menuNumber** is unchanged. Keeping the current menu open allows another selection to be made immediately.

ReDoMenu will redraw the current menu. If menu event routine changes the text in the menu (adding a selection asterisk, for example), a call to **ReDoMenu** will redraw the menu with the new text while leaving the menu open for another selection.

Example:

See Also: **DoMenu, GotoFirstMenu, DoPreviousMenu.**

RenameFile (Apple, C64, C128) — high-level disk

Function: Renames a file that is in the current directory.

Parameters:
- **r6** OLDNAME — pointer to null-terminated name of file as it appears on the disk (Apple GEOS: must be in main memory) (word).
- **r0** NEWNAME — pointer to new null-terminated name (Apple GEOS: must be in main memory) (word).

Uses: **curDrive**

Commodore:
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
- **curVBlkno**† used by VBM cacheing routines.
- **VBMchanged**† used by VBM cacheing routines.
- **numVBMBlks**† used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Alters:
- **diskBlkBuf** used for temporary block storage.
- **dirEntryBuf** old directory entry.

Commodore:
- **curDirHead** BAM updated to reflect newly freed blocks.
- **dir2Head**† (BAM for 1571 and 1581 drives only)
- **dir3Head**† (BAM for 1581 drive only)

Apple:
- **curVBlkno**† used by VBM cacheing routines.
- **VBMchanged**† set to **FALSE** by VBM cacheing routines to indicate cached VBM block has already been flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4–r6.**

Description: **RenameFile** searches the current directory for *OLDFILE* and changes the name string in the directory entry to *NEWFILE*.

RenameFile first calls **FindFile** to get the directory entry and ensure the *OLDFILE* does in fact exist. (If it doesn't exist, a **FILE_NOT_FOUND** error is returned.)

The directory entry is read in, the new file name is copied over the old file name, and the directory entry is rewritten. The date stamp of the file is not changed, but the modification date stamp of the directory (Apple GEOS only) is updated.

When using **Get1stDirEntry** and **GetNextDirEntry** to establish the old file name, do not pass **RenameFile** a pointer into **diskBlkBuf**. Copy the file name from **diskBlkBuf** to another buffer (such as **dirEntryBuf**) and pass **FreeFile** the pointer to that buffer. Otherwise when **FreeFile** uses **diskBlkBuf** it will corrupt the file name.

C64 & C128: Since Commodore GEOS does not support a hierarchical file system, the "current directory" is actually the entire disk. The call to **FindFile** also reads the BAM in from disk.

Apple: Will only rename a file in the current directory. To rename a file from another directory, the application must change directories (refer to **GoDirectory** for moving to another directory).

Example:

See also: **DeleteDir, FreeDir, FreeFile, FreeBlock.**

ReOpenDisk (Apple) — high-level disk

Function: Reopen a disk to the most recent directory.

Parameters: none

Uses:

curDrive	drive that disk is in.
curDirTabLo	last current directory.
curDirTabHi	last current directory.

Returns:

x	error ($00 = no error).
r5	pointer to disk name buffer as returned from **GetPtrCurDkNm**. This is a pointer to one of the **Drx CurDkNm** arrays.

Alters:

DrxCurDkNm	current disk name array contains disk name.
curDirHead	current directory header.
curKBlkno	block number of current directory's key block.
sysDirBlkno	block number of /SYSTEM directory. If system directory not found on disk, then this number is block of the root directory.
totNumBlks†	total number of blocks in volume.
pathnameBuf†	set to last logged directory.
curVBlkno†	block currently cached.
numVBMBlks†	number of VBM blocks on disk.
VBMchanged†	set to **FALSE**.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r0–r2, r4.**

Description: **ReOpenDisk** reopens a disk to its current directory (for each drive, GEOS keeps track of the block of the current directory).. **ReOpenDisk** allows an application to quickly switch back and forth between files on different drives, as is often the case when an application and its data are on different disks. It saves the application from calling **GoDirectory** every time drives are changed. It otherwise performs the same initialization as **OpenDisk.**

Note: **ReOpenDisk** assumes the user has not switched disks since the last time the drive was accessed. If a new disk is placed in the drive, the current directory block number associated with the drive will no longer be valid The application may want remember the volume name when first opening the disk and then compare the volume name after each call to **ReOpenDisk** with the original. If they match, its a fairly safe assumption that the disk has not changed.

Because GEOS uses the same allocation and file buffers for each drive, it is important to close all files and update the VBM (calling **PutVBM** if necessary) before accessing another disk.

Example:

See also: **OpenDisk, SetDevice.**

ResetAlarm (Apple) — clock driver

Function: Clock driver routine to disable the alarm.

Parameters: none.

Returns: nothing.

Alters: **alarmOn** $00

Destroys: assume **a, x, y, r0–r15.**

Description: **ResetSetAlarm** is a clock driver routine for disabling an alarm.

See also: **SetAlarm, SetTimeDate, ClockInt, ReadClock.**

ResetHandle (C64, C128) internal

Function: Internal routine used during the GEOS boot process.

Parameters: none.

Returns: does not return.

Description: **ResetHandle** is only used during the GEOS boot process. It is not useful to applications and is documented here only because it exists in the jump table.

See also: **BootGEOS.**

This page intentionally left blank to maintain right/left (verso/recto) page ordering. Final version will correct this.

RestartProcess (Apple, C64, C128) process

Function: Reset a process's timer to its starting value then unblock and unfreeze the process.

Parameters: **x** PROCESS — process to restart (0 – *n*–1, where *n* is the number of processes in the table) (byte).

Returns: **x** unchanged.

Destroys: **a**

Description: **RestartProcess** sets a process's countdown timer to its initialization value then unblocks and unfreezes it. Use **RestartProcess** to initally start a process after a call to **InitProcesses** or to rewind a process to the beginning of its cycle.

Note: **RestartProcess** clears the runable flag associated with the process, thereby losing any pending call to the process.

RestartProcess should always be used to start a process for the first time because **InitProcesses** leaves the value of the countdown timer in an unknown state.

Example:

See also: **InitProcesses, EnableProcess, UnfreezeProcess, UnblockProcess.**

RestoreFontData (Apple) text

Function: Restore internal font data from **saveFontTab.**

Parameters: none.

Returns: **curHeight**
baselineOffset
cardDataPntr
curIndexTable
curSetWidth

Destroys: **a, x.**

Description: **RestoreFontData** reverses the effect of **SaveFontData** by restoring the internal font variables from **saveFontTab.**

Example:

See also: **SaveFontData.**

RstrAppl (Apple, C64, C128) high-level disk

Function: Standard desk accessory return to application.

Parameters: none.

Uses: **curDrive.**

Returns: *never returns to desk accessory.*

Description: A desk accessory calls **RstrAppl** when it wants to return control to the application that called it. **RstrAppl** loads the swapped area of memory from the **SWAP FILE**, restores the saved state of the system from the internal buffer, resets the stack pointer to its original position, and returns control to the application.

It is the job of the desk accessory to ensure that if the current drive (**curDrive**) is changed that it be returned to its original value so that **RstrAppl** can find **SWAP FILE**. Under Apple GEOS it is not necessary to save the current directory.

Note: If a disk error occurs when reading in **SWAP FILE**, the remainder of the context switch (restoring the state of the system, etc.) is bypassed and control is immediately returned to the caller of the desk accessory. The application will have only a moderate chance to recover, however, because the area of memory that the desk accessory overlayed may very well include the area where the jsr to **GetFile** or **LdDeskAcc** resides. The return, therefore, may end up in the middle of desk accessory code.

Example:

See also: **StartAppl, GetFile.**

RstrFrmDialog (Apple, C64, C128) dialog box

Function: Exits from a dialog box, restoring the system to the state prior to the call to **DoDlgBox**.

Parameters: none.

Returns: Returns to point where **DoDlgBox** was called. System context is restored. **R0L** contains **sysDBData** return value.

Uses: **sysDBData.**

Destroys: assume **a, x, y, r0H–r15**

Description: **RstrFrmDialog** allows a custom dialog box routine to exit from the a dialog box. **RstrFrmDialog** is typically called internally by the GEOS system icon dialog box routines. However, it may be called by any dialog box routine to force an immediate exit.

RstrFrmDialog first restores the GEOS system state (context restore) and then calls indirectly through **recoverVector** to remove the dialog box rectangle from the screen. The routine in **recoverVector** is called with the **r2–r4** loaded for a call to **RecoverRectangle**. By default **recoverVector** points to **RecoverRectangle**, which will automatically recover the foreground screen from the background buffer. However, if the application is using background buffer for data, it will need to intercept the recover by placing the address of its own recover routine in **recoverVector**. If there is no shadow on the dialog box, then **recoverVector** is only called through once with **r2–r4** holding the coordinates of the dialog box rectangle. However, if the dialog box has a shadow, then **recoverVector** will be called through two 'imes: first for the patterned shadow rectangel and second for the dialog box rectangle. The application may want to special-case these two recovers when recovering.

Apple: If **recoverOnce** is set to **TRUE, RstrFrmDialog** will only call through **recoverVector** once as if there were no shadow box (even if there is one). **RstrFrmDialog** automatically calls **RestoreSysRam.**

Note: **RstrFrmDialog** restores the **sp** register to value it contained at the call to **DoDlgBox** just before returning. This allows **RstrFrmDialog** to be called with an arbitrary amount of data on top of the stack (as would be the case if called from within a subroutine). GEOS will restore the stack pointer properly.

Example:

See also: **DoDlgBox, InitForDialog, RestoreSysRam, RecoverRectangle, RestoreFG.**

SaveFG (Apple) graphics/utility

Function: Saves a portion of the foreground screen to a buffer.

Parameters:
r0 FGSTRUCT — pointer to an FG data structure (word).
r1 FGBUF — pointer to buffer to place foreground data (word).

Returns: Buffer pointed to by FGBUF contains foreground data in a form that RestoreFG can restore.

Destroys: a, x, y, r1–r6.

Description: SaveFG saves a rectangular area of the foreground screen to a temporary buffer. This buffer can later be restored to the foreground screen with RestoreFG. The SaveFG and RestoreFG duo allow the application to save and restore areas of the foreground screen without imprinting them to the background buffer. This capability lets the application use the background buffer for data while still using dialog boxes and menus, which depend on recovering from the background buffer. The application need only saveFG the proper areas of the screen prior to putting up a dialog box or letting a menu drop, patch into recoverVector, and RestoreFG as necessary.

FGSTRUCT points to an FG data structure, which is in the following format:

```
.word   X1      ;pixel x-position of left edge
.byte   Y1      ;pixel y-position of top edge
.word   X2      ;pixel x-position of right edge (X1 + WIDTH)
.byte   HEIGHT  ;pixel height (Y2 – Y1)
```

The amount of data stored into the buffer at *FGBUF* depends on the size of the saved region. The following relationship applies:

bufBytes = HEIGHT * (X1/7 – X2/7 +1)

Most GEOS applications will reserve 5,712 bytes for the save buffer, which will hold the largest standard dialog box or about three levels of submenus.

Note: It is useful to set recoverOnce to TRUE when using SaveFG, which will cause dialog boxes with shadows to only call through recoverVector once, rather than the normal twice.

Example:

See also: RecoverFG.

SaveFile (Apple, C64, C128) — high-level disk

Function: General purpose save file routine that will create a GEOS sequential file and save a region of memory to it or create an empty GEOS VLIR file.

Parameters:

r9 HEADER — pointer to GEOS file header for file. The first two bytes of the file header point to the filename (word).

r10L DIRPAGE — GEOS directory page to begin searching for free directory slot; each directory page holds eight files and corresponds to one notepad page on the GEOS deskTop. The first page is page one.

Uses:

curDrive

year, month, day, hours, minutes for date-stamping file.

Commodore:

curType GEOS 64 v1.3 and later: for detecting REU shadowing.

interleave† desired physical sector interleave (usually 8);. applications need not set this explicitly — will be set automatically by internal GEOS routines.

Apple:

RWbank source bank for sequential data and *HEADER* (**MAIN** or **AUX**).

curKBlkno current directory.

curVBlkno† used by VBM cacheing routines.

VBMchanged† used by VBM cacheing routines.

numVBMBlks† used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x error ($00 = no error).

r9 pointer to **fileHeader**, which contains file header block as written to disk.

Commodore:

r6 pointer to **fileTrScTab**, which contains track/sector table for the file.

Alters:

dirEntryBuf contains newly-built directory entry.

diskBlkBuf used for temporary storage.

curDirHead this buffer contains the current directory header.

Commodore:

fileTrScTab Contains track/sector table for file as returned from **BlkAlloc**. The track and sector of the file header block is at **fileTrScTab**+0 and **fileTrScTab**+1. The end of the table is marked with a track value of $00.

dir2Head† (BAM for 1571 and 1581 drives only)

dir3Head† (BAM for 1581 drive only)

Apple:

INDEXBLOCKBUF† index block for chain as created by **BlkAlloc** (in auxiliary memory; see **BlkAlloc** for information on accessing it).

†used internally by GEOS disk routines; applications generally don't use.

Destroys: Commodore:
a, y, r0–r8.

Apple:
a, y, r1–r8.

Description: **SaveFile** is the most general -pupose write data type routine in GEOS. It creates a new file, either sequential or VLIR. If the file is a sequential file, it will write out the range of memory specified in the header to disk. If the file is a VLIR file, it will create an empty file (just a file header and an index table; all records in the index table are marked as unused).

Not only does the file header pointed to by *HEADER* act as a prototype for the file, it also holds all the information needed to create the file. This includes the file type (**SEQ** or **VLIR**) and other pertinent information, such as the start and end address, which are used when creating a sequential file. The file header pointed to by *HEADER* has one element, however, that *is* changed before it is written to disk: the first word of the fileheader points to a null-terminated filename string. **SaveFile** patches this word in its own copy in **fileHeader** before it is written to disk.

SaveFile calls **SetGDirEntry** and **BlkAlloc** to construct the basic elements of the file, then calls **WriteFile** to put the data into it. After the file is written, the BAM/VBM is written to disk (**PutDirHead** under Commodore GEOS; **PutVBM** under Apple GEOS.)

Example:

See also: **GetFile, OpenRecordFile.**

SaveFontData (Apple) text

Function: Save internal font data to **saveFontTab.**

Parameters: none.

Returns: nothing.

Uses: **curHeight**
baselineOffset
cardDataPntr
curIndexTable
curSetWidth

Destroys: **a, x.**

Description: **SaveFontData** saves the internal font variables to the area at **saveFontTab.** This allows a font to be temporarily changed and then restored. **saveFontTab** is **FONTLEN** bytes long.

See also: **RestoreFontData.**

Example:

SetAlarm (Apple) — clock driver

Function: Clock driver routine to set the clock's alarm time.

Parameters:
r0L HOUR — hour (0–23) (byte).
r0H MINUTE — minute (0–59) (byte).

Returns: nothing.

Alters: **alarmOn** %10000000 (alarm enabled but not triggered).

Destroys: assume **a, x, y, r0–r15.**

Description: **SetAlarm** is a clock driver routine for setting a new alarm time. Most clock drivers do no error checking on the parameters and expect valid values.

Note: To be sure the alarm is triggered correctly, disable interrupts around the call to **SetAlarm.**

See also: **ResetAlarm, SetTimeDate, ClockInt, ReadClock.**

SetDevice (C64, C128) — high-level disk

Function: Establish communication with a new peripheral (disk or printer).

Parameters:
- **a** — DEVNUM — **DRIVE_A** through **DRIVE_D** for disk drives, **PRINTER** for serial printer, or any other valid serial device bus address (byte).

Uses:
- **curDevice** — currently active device.

Returns:
- **x** — error ($00 = no error).

Alters:
- **curDevice** — new current device number.
- **curDrive** — new current drive.number if device is a disk drive.
- **curType** — GEOS v1.3 and later: current drive type (copied from **driveType** table).

Destroys: a, y.

Description: SetDevice changes the currently active Commodore device and is used primarily to switch from one disk drive to another. **SetDevice** also allows a printer driver to gain access to the serial bus.

Each I/O device has an associated *device number* that distinguishes its I/O from the I/O of other devices. At any given time only one device is active. The active device is called the *current device* and to change the current device an application calls **SetDevice.**

Because **SetDevice** was originally designed to switch between serial bus devices, *DEVNUM* reflects the architecture of Commodore serial bus: disk drives are numbered 8 through 11 and the printer is numbered 4. However, not all I/O devices are actual serial bus peripherals. A RAMdisk, for example, uses a special device driver to make a cartridge port RAM-expansion unit emulate a Commodore disk drive. **SetDevice** switches between these devices just as if they were daisy-chained off of the serial bus.

Commodore GEOS up through v1.2 supports two disk devices, **DRIVE_A** and **DRIVE_B**. Commodore GEOS v1.3 and later supports up to four disk devices, **DRIVE_A** through **DRIVE_D**, within the Kernal disk routines but not from the deskTop. (The current version of the deskTop will never pass **numDrives** with a value greater than two.) GEOS always addresses the first drive as **DRIVE_A**, the second as **DRIVE_B**, and so on regardless of the physical device settings.

Because most printers attach to the Commodore through the serial port, printer drivers will call **SetDevice** with a *DEVNUM* value of **PRINTER** to make the printer the active device.

Note: **SetDevice** calls **ExitTurbo** to ensure that the old device is no longer actively sensing the serial bus then rearranges the device drivers as necessary to make the new device (*DEVNUM*) the current device. With more than one type of device attached (e.g., a 1541 and a 1571), Commodore GEOS must switch the internal device drivers, making the the driver for the selected device active. GEOS stores

inactive device drivers in the Commodore 128 back RAM and in special system areas in an REU. For these reasons it is important that Commodore GEOS applications use **SetDevice** to change the active device. An application should never directly modify **curDrive** or **curDevice**.

Apple: **SetDevice** does not exist in Apple GEOS because the task of switching between devices is not complicated by daisy-chained I/O devices and inactive device drivers (as it is on the Commodore computers). To specify a new disk drive, store the new drive number directly into **curDrive**.

To make porting disk routines easier, the following Apple equivalent of **SetDevice** can be used:

```
.if   (APPLE)        ;Apple GEOS SetDevice equivalent

SetDevice:
      sta      curDrive     ;make the requested drive current
      ldx      #NO_ERROR    ;and return a good status
      rts

.endif
```

See also: **OpenDisk, ChangeDiskDevice.**

SetGDirEntry (Apple, C64, C128) — mid-level disk

Function: Builds a system specific directory entry from a GEOS file header, date-stamps it, and writes it out to the current directory. Apple GEOS version will also create and write out subdirectories.

Parameters: **r10L** DIRPAGE — directory page to begin searching for free slot; each directory page holds eight files and corresponds to one notepad page on the GEOS deskTop. The first page is page one.

Commodore:
- **r2** NUMBLOCKS — number of blocks in file (word).
- **r6** TSTABLE — pointer to a track/sector list of unused blocks (unused but allocated in the BAM), usually a pointer to **fileTrScTab**; **BlkAlloc** can be used to build such a list (word).
- **r9** FILEHDR — pointer to GEOS file header (word).

Apple:
- **r2** NUMBLOCKS — number of blocks in file or $0000 to create a subdirectory (word).
- **r6** INDXBLK — If creating a file: block number of sequential-file index block or VLIR master index block. If creating a subdirectory: key block of subdirectory (word).
- **r7** FILEBYTES — number of bytes in file (word).
- **r8** HDBLKNUM — block number of this file's header block (word).
- **r9** FILEHDR — if creating a file, pointer to GEOS file header; if creating a subdirectory, pointer to 16-character max. null-terminated name for subdirectory (16 characters *including* **NULL**) (must be in main memory) (word).

Uses: **curDrive**
year, month, day, hours, minutes for date-stamping file.

Commodore:
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.
- **curDirHead** this buffer must contain the current directory header.
- **dir2Head**† (BAM for 1571 and 1581 drives only)
- **dir3Head**† (BAM for 1581 drive only)
- **interleave**† desired physical sector interleave (usually 8);. applications need not set this explicitly — will be set automatically by internal GEOS routines. Only used when new directory block is allocated

Apple:
- **curKBlkno** current directory.
- **curVBlkno**† used by VBM cacheing routines.
- **VBMchanged**† used by VBM cacheing routines.
- **numVBMBlks**† used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns: **x** error ($00 = no error).

Commodore:
r6 pointer to first non-reserved block in track/sector table (**SetGDirEntry** reserves.one block for the file header and a second block for the index table if the file is a VLIR file).

Apple:
r1 block number that directory entry was added to.
r10L directory page actually used.
r10H unused directory entry within block in **diskBlkBuf**. There are 13 directory entries per ProDOS block, numbered 1–13.

Alters:
dirEntryBuf contains newly-built directory entry.
diskBlkBuf used for temporary storage of the directory block.

Destroys:
Commodore:
a, y, r0–r5, r7–r8.

Commodore:
a, y, r1, r3–r5, r7–r8.

Description: **SetGDirEntry** calls **BldGDirEntry** to build a system specific directory entry form the GEOS file header, date-stamps the directory entry, calls **GetFreeDirBlk** to find an empty directory slot, and writes the new directory entry out to disk. The Apple version of **SetGDirEntry** can also be used to create subdirectory entries by passing a *NUMBLOCKS* value of $0000.

Most applications will create new files by calling **SaveFile** and new subdirectories by calling **MakeSubDir**. Both these routines call **SetGDirEntry** as part of their normal processing.

Example:

See also: **BldGDirEntry .**

SetGEOSDisk (C64, C128) high-level disk

Function: Convert Commodore disk to GEOS format.

Parameters: none.

Uses:

curDrive	
curType	GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns: **x** error ($00 = no error).

Alters:

curDirHead	directory header is read from disk.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Destroys: **a, y, r0L, r1, r4–r5.**

Description: **SetGEOSDisk** converts a standard Commodore disk tinto GEOS format by writing the GEOS ID string to the directory header (at **OFF_GEOS_ID**) and creating an off-page directory block. An application can call **SetGEOSDisk** after **OpenDisk** returns the **isGEOS** flag set to **FALSE**. Typically the user is prompted before the conversion.

SetGEOSDisk expects the disk to have been previously opened with **OpenDisk**. It first calls **GetDirHead** to read the directory header into memory then calls **CalcBlksFree** to to see if there is block available for the off-page directory (if there isn't, an **INSUFFICIENT_SPACE** error is returned). **SetNextFree** is then called to allocate the off-page directory block. The off-page directory block is written with empty directory entries and a pointer to it is placed in the directory header (at **OFF_OP_TR_SC**). Finally **PutDirHead** is called to write out the new BAM and directory header.

Apple: All ProDOS disks are already GEOS compatible.

Example:

See also: **ChkDkGEOS.**

SetLdVars (Apple) — mid-level disk

Function: Sets the internal "Ld" variables (those used by **LdAppl, LdDeskAcc,** and **LdFile**).

Parameters:

r0L LOAD_OPT:
- bit 0: 0 load at address specified in file header; application will be started autmatically
  1 load at address in **r7**; application will not be started automatically.
- bit 7: 0 not passing a data file.
  1 **r2** and **r3** contain pointers to disk and data file names.
- bit 6: 0 not printing data file.
  printing data file; application should print file and exit.

r7 LOAD_ADDR — optional load address. only used if bit 0 of *LOAD_OPT* is set (word).

r2 DATA_DISK — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the disk that contains the data file, usually a pointer to one of the **DrxCurDkNm** buffers (word).

r3 DATA_FILE — only valid if bit 7 or bit 6 of *LOAD_OPT* is set: pointer to name of the data file (word).

r10L RECVR_OPTS (byte).

Returns: nothing.

Destroys: **a**.

Description: **SetLdVars** sets the internal load variables from GEOS pseudoregisters. **GetFile, LdApplic, LdDeskAcc,** and **LdFile** call **SetLdVars** as necessary. Most applications will not need to call it directly.

Note: **LdDeskAcc** does not properly set the internal "Ld" variables when it is called directly (as opposed to indirectly through **GetFile**, in which case the variables are set properly). **SetLdVars** may be used to properly set the internal "Ld" variables prior to calling **LdDeskAcc**.

See also: **GetLdVars.**

SetMode (Apple) — printer driver

Function: Set printer mode.

Parameters: **r0–r2L** MODE — flags for possible printer modes and capabilities (five bytes).

Returns: **x** STATUS — printer error code; $00 = no error (byte)

Destroys: assume **r0–r4**.

Description: **SetMode** allows an application to access special printer capabilities. Features are selected through five *MODE* bytes. The bits in the *MODE* bytes correspond to the list of available features returned by **GetMode**.

SetMode operates intelligently. It first compares the requested *MODE* setting against the current state of the printer. If a bit setting is different and the driver supports the associated feature, then the proper codes are sent to the printer to establish the new mode.

The *MODE* Values:

r0L: (matches **currentMode** text variable)

bit	description
b7	†underline.
b6	†bold.
b5	reverse.
b4	†italic.
b3	outline.
b2	†superscript.
b1	†subscript.
b0	*reserved for future use.*

r0H: (text density selection)

bit	description
b7	†pica type (10 cpi).
b6	†elite type (12 cpi).
b5	†condensed type (16 cpi).
b4	†proportional type.
b3	†double height.
b2	†half-height.
b1	†eight lines per inch vertical density.
b0	†six lines per inch vertical density.

r1L: (miscellaneous features)

bit	description

b7	print red or magenta.
b6	print yellow.
b5	print blue or cyan.
b4	†expanded (double-width) type.
b3	†NLQ (near letter quality) type.
b2	*reserved for future use.*
b1	*reserved for future use.*
b0	*reserved for future use.*

r1H: (internal font selection)

bit	description
b7	font 7.
b6	font 6.
b5	font 5.
b4	font 4.
b3	font 3.
b2	font 2 (Roman?).
b1	font 1 (Helvetica?).
b0	font 0 (Courier?).

r2L: (horizontal graphics density selection)

bit	description
b7	60 dpi.
b6	†80 dpi.
b5	90 dpi.
b4	120 dpi.
b3	180 dpi.
b2	300 dpi.
b1	360 dpi.
b0	*reserved for future use.*

†*This feature implemented in most Berkeley Softworks printer drivers.*

Note: Most printer drivers only support a small subset of these features even if the printer they are driving supports more.

Example:

See also: **GetMode, SetNLQ.**

SetMouse (C128) — input driver

Function: Input device scan reset.

Parameters: none.

Returns: nothing

Destroys: assume **a, x, y, r0–r15**

Description: GEOS 128 calls **SetMouse** during Interrupt Level, immediately after the keyboard is scanned for a new key, to reset the pot (potentiometer) scanning lines so that they will recharge with the new value of. It is primarily of use with the Commodore 1351 mouse, which requires having the pot lines reset regularly. Other input drivers will have a **SetMouse** routine that merely performs an rts. An application should never need to call **SetMouse**.

See also: **SlowMouse, UpdateMouse, InitMouse, KeyFilter.**

SetMsePic (Apple, C128) sprite/mouse

Function: Uploads and pre-shifts a new mouse picture for the software sprite handler.

Parameters: **r0** MSEPIC — pointer to 32 bytes of mouse sprite image data or one of the following special codes:

ARROW	Arrow pointer (Apple and C128)
HOURGLASS	Busy (Apple only)

Returns: nothing.

Destroys: **a, x, y, r0–r15**

Description: The software sprite routines used by GEOS 128 in 80-column mode and Apple GEOS treat the mouse sprite (sprite #0) differently than the other sprites. Sprite #0 is optimized and hardcoded to provide reasonable mouse-response while minimizing the flicker typically associated with erasing and redrawing a fast-moving object. The mouse sprite is limited to a 16x8 pixel image. The image includes a mask of the same size and both are stored in a pre-shifted form within internal GEOS buffers. For these reasons, a new mouse picture must be installed with **SetMsePic** (as opposed to a normal **DrawSprite**). **SetMsePic** pre-shifts the image data and lets the soft-sprite mouse routine know of the new image.

SetMsePic accepts one parameter: a pointer to the mask and image data or a constant value for one of the predefined shapes. If a user-defined shape is used, the data that *MSEPIC* points to is in the following format:

16 bytes	16x8 "cookie cutter" mask. Before drawing the software mouse sprite, GEOS and's this mask onto the foreground screen. Any zero bits in the mask, clear the corresponding pixels. One bits do not affect the screen.
16 bytes	16x8 sprite image. After clearing pixels with the mask data, the sprite image is or'ed into the area. Any one bits in the sprite image set the corresponding pixels. Zero bits do not affect the screen.

GEOS treats the each 16-byte field as 8 rows of 16 bits (two bytes per row).

Note: **SetMsePic** calls **HideOnlyMouse.**

Example:

```
;*******************************************************
;Put up a new mouse picture
;*******************************************************

ArrowUp:
     LoadW    r0, #DnArrow        ;point at new image
     jsr      SetMsePic           ;install it
     rts

;macro to store a word value in high/low order
.macro        HILO word
     .byte    ]word,[word
.endm

;Mouse picture definition for down-pointing arrow
DnArrow:
```

```
;mask

     HILO     %1111111110000000
     HILO     %1111111001111110
     HILO     %0001100111111001
     HILO     %0110011111100111
     HILO     %0111111110011111
     HILO     %0111111110011111
     HILO     %0111111111101111
     HILO     %0000000000001111


;image
     HILO     %0000000000000000
     HILO     %0000000001111110
     HILO     %0000000111111000
     HILO     %0110011111100000
     HILO     %0111111110000000
     HILO     %0111111110000000
     HILO     %0111111111100000
     HILO     %0000000000000000
```

See also: **TempHideMouse, DrawSprite.**

SetNewMode (C128) graphics

Function: Changes GEOS 128 from 40-column mode to 80-column mode, or vice-versa.

Parameters: **graphMode** GRMODE — new graphics mode to change to:
40-column: **GR_40**
80-column: **GR_80**

Returns: nothing.

Destroys: **a, x, y, r0-r15.**

Description: **SetNewMode** changes the operating mode of the Commodore 128.

40-column mode (graphMode == GR_40)

1: 8510 clock speed is slowed down to 1Mhz because VIC chip cannot operate at 2Mhz.

2: **rightMargin** is set to 319.

3: **UseSystemFont** is called to begin using the 40-column font.

4: 40-column VIC screen is enabled.

5: 80-column VDC screen is set to black on black, effectively disabling it.

80-column mode (graphMode == GR_80)

1: 8510 clock speed is raised to 2Mhz.

2: **rightMargin** is set to 639.

3: **UseSystemFont** is called to begin using the 80-column font.

4: 40-column VIC screen is disabled.

5: 80-column VDC screen is enabled.

Example:

```
ChangeMode:
     jsr     GreyScreen          ;grey out old screen
     lda     graphMode           ;switch mode by flipping
     eor     #%10000000          ;40/80 bit
     jsr     SetNewMode          ;and calling SetNewMode
     jsr     GreyScreen          ;grey out new screen
     rts                         ;exit

GREYPAT      = 2                 ;grey pattern to use
GreyScreen:
     jsr     i_GraphicsString
     .byte   NEWPATTERN,GREYPAT  ;set to grey pattern
     .byte   MOVEPENTO           ;put pen in upper left
```

```
.word   0                       ;x
.byte   0                       ;y
.byte   RECTANGLETO             ;grey out entire screen
.word   (SC_40_WIDTH-1)|DOUBLE_W|ADD1_W
.byte   SC_PIX_HEIGHT
.byte   NULL
rts
```

SetNextFree (Apple, C64, C128) — mid-level disk

Function: Search for a nearby free block and allocate it.

Parameters: **r3** block (Commodore track/sector) to begin search (word).

Uses: **curDrive**

Commodore:

curDirHead	this buffer must contain the current directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)
interleave†	desired physical sector interleave (usually 8);. applications need not set this explicitly — will be set automatically by internal GEOS routines.

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.
totNumBlocks†	for detecting last block on disk.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).
r3	block (Commodore track/sector) allocated.

Alters:

Commodore:

curDirHead	BAM updated to reflect newly allocated block.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

curVBlkno†	used by VBM cacheing routines.
VBMchanged†	set to **TRUE** by VBM cacheing routines to indicate cached VBM block has changed and needs to be flushed

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys:

Commodore:
a, y, r6–r7, r8H.

Apple:
a, y, r6, r7, r8H.

Description: Given the current block as passed in **r3**, **SetNextFree** searches for the next free block on the disk. The "next" free block is not necessarily adjacent to the previous block because **SetNextFree** may interleave the blocks. Proper interleaving allows the drive to read and write data as fast as possible because it minimizes the time the drive spends waiting for a block to spin under the read/write head. It means, however, that sequential data blocks may not occupy adjacent blocks on

the disk. As long as an application is using the standard GEOS file structures, this interleaving should not be apparent.

After determining the ideal sector from any interleave calculations, SetNextFree tries to allocate the block it if it is unused. If the block is used, SetNextFree picks another nearby sector (jumping to another track if necessary) and calls tries again. This process continues until a block is actually allocated or the end of the disk is reached, whichever comes first. If the end of the disk is reached, an INSUFFICIENT_SPACE error is returned.

Notice that SetNextFree only searches for free blocks starting with the current block and searching towards the end of the disk. It does not backup to check other areas of the disk because it assumes they have already been filled. (Actually, under Commodore GEOS,.SetNextFree will backtrack as far back as beginning of the current track but will not go to any previous tracks.). Usually this is a safe assumption because SetNextFree is called by BlkAlloc, which always begins searching for free blocks from the beginning of the disk.

It is conceivable, however, that an application might want to implement an AppendRecord function (or something of that sort), which would append a block of data to an already existing VLIR record without deleting, reallocating, and then rewriting the record like WriteRecord.

In order to maintain any interleave from the last block in the record to the new block, the AppendRecord routine passes the track and sector of the last block in the record to SetNextFree. SetNextFree will start searching from this block. If a free block cannot be found, an INSUFFICIENT_SPACE error is returned. Since SetNextFree only searched from the current block to the end of the disk, the possiblility exists that a free block lies somewhere on a previous, still unchecked disk area. The following alternative to SetNextFree will circumvent this problem:

```
MySetNextFree:
;Look for a free block starting at the current block
;so that we continue the interleave if possible
     jsr      SetNextFree ;       look for block to allocate
     cpx      #INSUFFICENT_SPACE  ;check for no blocks
     beq      10$                 ;start from beginning if none
     rts                          ;exit on any other error or
                                  ;valid block found.

;We got an insufficient space error. Start the search
;again from the beginning of the disk.
10$:
.if          APPLE
     LoadW   r3,#0                ;start at block 0

.else        ;CBM
     LoadB   r3H,#0               ;always sector 0
     ldx     #1                   ;assume track 1
     ldy     curDrive             ;but special case 1581
     lda     driveType-8,y        ;because of outer/inner track
     and     #$0f                 ;searching scheme
     cmp     #DRV_1581
     bne     20$                  ;branch if not 1581
     ldx     #39                  ;1581 counts down on inner (39-1)
                                  ;then up on outer (41-80)
20$:
```

```
        stx     r3L                 ;track number
.endif
        jmp     SetNextFree         ;go search. let it return
```

C64 & C128: **SetNextFree** uses the value in **interleave** to establish the ideal next sector. A good interleave will arrange successive sectors so as to minimize the time the drive spends stepping the read/write head and waiting for the desired sector to spin around. The value in **interleave** is usually set by the Configure program and internally by GEOS disk routines. The application will usually not need to worry about the value in **interleave.**

Because Commodore disks store the directory on special tracks, **SetNextFree** will automatically skip over these special tracks unless **r3L** is started on one of these tracks, in which case **SetNextFree** assumes that this was intentional and a block on the directory track is allocated. (This is exactly how **GetFreeDirBlk** operates.) The directory blocks for various drives can be determined by the following constants:

1581	DIR_1581_TRACK	(one track)
1541	DIR_TRACK	(one track)
1571	DIR_TRACK DIR_TRACK+N_TRACKS	(two tracks)

SetNextFree does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

Apple: The Apple version of **SetNextFree** does no interleave calculations because ProDOS devices handle physical interleaving when they map block numbers to physical sectors. **SetNextFree** on the Apple merely increments the block number.

SetNextFree does not automatically flush the VBM cache. See **PutVBM** for more information on flushing the cache.

Example:

See also: **AllocateBlock, FreeBlock, BlkAlloc.**

SetNLQ (C64, C128) — printer driver

Function: Enter high-quality printing mode.

Parameters: **r1** WORKBUF — pointer to a 640-byte work buffer for use by the printer driver (word).

Returns: nothing.

Destroys: assume **a, x, y, r0–r15.**

Description: SetNLQ sends the appropriate control codes to place the printer into high-quality print mode (as opposed to the default draft mode). **SetNLQ** is called after **StartASCII** has been called to enable text output.

See also: **StartASCII, PrintASCII, SetMode.**

SetPattern (Apple, C64, C128) graphics

Function: Set the current fill pattern.

Parameters: **a** GEOS system pattern number (must be between 0 and 31) (byte).

Returns: nothing.

Alters: **curPattern** Contains an address pointing to the eight-byte pattern.

Destroys: **a**

Description: Sets the current fill pattern. There are 34 system patterns (numbered 0-33) in GEOS 64 and GEOS 128; Apple GEOS adds a 35th, user-defined pattern (number 34). Unfortunately, **SetPattern** will only work correctly with patterns numbered 0–31. To access higher number patterns, call **SetPattern** with a value of 31 and add 8 to **curPatttern** to access pattern 32, add 16 to access pattern 33, and so on.

Apple: Although **curPattern** will contain a valid address, it is an address in auxiliary-high memory, and this memory bank is all but inaccessible to user applications. Use **GetPattern** and **SetUserPattern** to access the pattern data.

Example:

See also: **GetPattern, SetUserPattern.**

SetTimeDate (Apple) — clock driver

Function: Clock driver routine to set the clock's time and date.

Parameters:

r0L	YEAR — current year (calendar year -1900) (byte).
r0H	MONTH — current month (0–12) (byte).
r1L	DAY — day of month (1–28, 29, 30, 31) (byte).
r1H	HOUR — hour (0–23) (byte).
r2L	MINUTE — minute (0–59) (byte).
r2H	SECOND — seconds (0-59) (byte).

Returns: nothing.

Destroys: assume **a, x, y, r0–r15.**

Description: **SetTimeDate** is a clock driver routine for setting a new time and date. Most clock drivers do no error checking on the parameters and expect valid values (passing a nonexistent day in a month is a no-no).

Note: To be sure the time is set correctly, disable interrupts around the call to **SetTimeDate.**

See also: **ReadClock, ClockInt, SetAlarm, ResetAlarm.**

SetUserPattern (Apple) graphics

Function: Upload an eight-byte pattern definition to a GEOS system pattern.

Parameters:
- **a** GEOS system pattern number to alter.
- **r0** pointer to eight-byte pattern definition (word).

returns: nothing.

Alters:
- **curPattern** altered so that the new pattern becomes the current pattern. **r0** is unchanged.

Destroys: **a, y**

Description: **SetUserPattern** uploads an eight-byte pattern definition to the system pattern space in auxiliary high memory. This is the only convenient way to modify the system pattern definitions under Apple GEOS.

Example:

See also: **GetPattern, SetPattern.**

Sleep (Apple, C64, C128) process

Function: Pause execution of a subroutine ("go to sleep") for a given time interval.

Parameters: **r0** DELAY — number of vblanks to sleep (word).

Returns: nothing: does not return directly to caller (see description below).

Destroys: **a, x, y**

Description: **Sleep** stops executing the current subroutine, forcing an early **rts** to the routine one level lower, essentially putting the current routine "to sleep." At Interrupt Level, the *DELAY* value assocated with each sleeping routine is decremented. When the the *DELAY* value reaches zero, **MainLoop** removes the sleeping routine from the sleep table and perfomrs a **jsr** to the instruction following the original **jsr Sleep**, expecting a subsequent **rts** to return control back to **MainLoop**. For example, in the normal course of events, **MainLoop** might call an icon event service routine (after an icon is clicked on). This service routine can perform a **jsr Sleep**. **Sleep** will force an early **rts**, which, in this case, happens to return control to **MainLoop**. When the routine awakes (after *DELAY* vblanks have occurred), **MainLoop** performs a **jsr** to the instruction that follows the original **jsr Sleep**. When this wake-up **jsr** occurs, it occurs at some later time the contents of the processor registers and GEOS pseudoregisters are uninitialized. A subsequent **rts** will return to **MainLoop**.

Sleeping in Detail:

1: The application calls **Sleep** with a **jsr Sleep**. The **jsr** places a return address on the stack and transfers the processor to the **Sleep** routine.

2: **Sleep** pulls the return address (top two bytes) from the stack and places those values along with the *DELAY* parameter in an internal sleep table.

4: **Sleep** exectues an **rts**. Since the original caller's return address has been pulled from the stack and saved in the sleep table, this **rts** uses the next two bytes on the stack, which it assumes comprise a valid return address. (Note: it is imperitive that this is in fact a return address; do not save any values on the stack before calling **Sleep**.)

5: At Interrupt Level GEOS decrement sthe sleep timer until it reaches zero.

6: On every pass, **MainLoop** checks the sleep timers. If one is zero, then it removes that sleeping routine from the table, adds one to the return address it pulled from the stack (so it points to the instruction following the **jsr Sleep**), and **jsr**'s to this address. Because no context information is saved along with the sleep address, the awaking routine cannot depend on any values on the stack, in the GEOS pseudoregisters, or in the processor's registers.

Note: A *DELAY* value of $0000 will cause the routine to sleep only until the next pass through **MainLoop**.

When debugging an application, be aware that **Sleep** alters the normal flow of control.

Example:

See also: **InitProcesses.**

SlowMouse (Apple, C64, C128) — input driver

Function: Kills any accumulated speed in a non-proportional input device.

Parameters: none.

Returns: nothing

Alters: internal input-driver speed variables, if any.

Destroys: assume **a, x, y, r0–r15**

Description: Input drivers for non-proportional input devices, such as a joystick, will often internally associate a speed and velocity with movement. This way the pointer can speed up when the user is trying to move large distances. **SlowMouse** will tell the input driver to kill any accumulated speed, effectively stopping the pointer at a specific location and forcing it to regain momentum. Depending on the input driver, **SlowMouse** may or may not have an effect on the pointer's movement. The standard mouse driver, for example, simply performs an **rts** but some other input driver may actually copy the value in **minMouseSpeed** to its own internal speed variable.

GEOS calls **SlowMouse** when it drops menus down. A driver that has velocity variables should adjust the current speed so that the pointer does not immediately jump off the menu. An application may want to call **SlowMouse** when the user is required to make precise movements.

See also: **UpdateMouse, InitMouse, SetMouse, KeyFilter.**

SmallPutChar (Apple, C64, C128) text

Function: Print a single character without the **PutChar** overhead.

Parameters:
a CHAR — character code (byte).
r11 XPOS — x-coordinate of left of character (word).
r1H YPOS — y-coordinate of character baseline (word).

Uses: same as **PutChar.**

Returns:
r11 x-position for next character.
r1H unchanged.

Destroys:
Commodore:
a, x, y, r1L, r2–r10, r12, r13

Apple:
a, x, y, r1L, r2

Description: **SmallPutChar** is a bare bones version of **PutChar. SmallPutChar** will not handle escape codes, does no margin faulting, and does not normalize the x-coordinates on GEOS 128.

SmallPutChar will assume the character code is a valid and printable character Any portion of the character that lies above **windowTop** or below **windowBottom** will not be drawn. If a character lies partially outside of **leftMargin** or **rightMargin, SmallPutChar** will only print the portion of the character lies within the margins. **SmallPutChar** will also accept small negative values for the character x-position, allowing characters to be clipped at the left screen edge.

Note: Partial character clipping at the left margin, including negative x-postion clipping, is not supported by early versions of GEOS 64 (earlier than v1.4) — the entire character is clipped instead. Full left-margin clipping is supported on all other versions of GEOS: GEOS 64 v1.4 and above, GEOS 128 (both in 64 and 128 mode), and Apple GEOS.

Like **PutChar,** 159 is the maximum *CHAR* value that **SmallPutChar** will handle correctly. Most fonts will not have characters for codes beyond 129.

Example:

See also: **PutChar, PutString.**

SoftSprHandler (Apple) sprite

Function: Software sprite handler.

Parameters: none.

Uses: **mobenble**
moby2
mobx2
req1xpos–req7xpos
mob1ypos–mob7ypos

Returns: nothing.

Alters: **offFlag** $40 Flag to Redraw mouse at next interrupt.

Destroys: **a, x, y, r0–r15**

Description: **SoftSprHandler** is the routine **MainLoop** calls to update the software sprites. This involves redrawing them if they have been temporarily removed (by a call to **TempHideMouse**, perhaps) or erasing and redrawing them if they have moved. An application will normally have no need to call this routine directly, assuming **MainLoop** is being returned to normally.

This is where sprites are physically repositioned. A call to **PosSprite**, for example, only updates the **reqXpos***N* and **mob***N***Ypos** request position variables associated with the sprite. The actual physical position does not change until **SoftSprHandler** is called (usually at **MainLoop**) to redraw the sprites.

SoftSprHandler draws higher numbered sprites first so that sprites with lower numbers will appear on top of sprites with higher numbers. Sprite #5, for example, will appear on top of sprite #4 when they are drawn together.

Note: **SoftSprHandler** does not use **reqXpos0** and **mob0ypos** because the mouse sprite is redrawn at interrupt level.

See also: **TempHideMouse, HideOnlyMouse.**

SortAlpha (Apple) utility

Function: Case-insensitive alphanumeric sort.

Parameters:
r0 LIST — pointer to unsorted list (word).
r1 NUMRECS — number of 16-byte records in the list

Returns: list sorted.

Destroys: **a, x, y, r1–r4.**

Description: **SortAlpha** sorts an arbitrarily large list of 16-byte ASCII records. The record size is currently fixed at 16-bytes (the number of characters in a GEOS filename), although future versions of Apple GEOS will probably make the record size variable.

SortAlpha uses a selection sort algorithm, which works well with small records and has a running time proportional to $NUMRECS^2$. **SortAlpha** ignores letter case in its comparisons.

Note: **SortAlpha** correctly handles the trivial cases where *NUMRECS* is either zero or one.

Example:

```
REC_SIZE == 16              ;always 16 in Apple GEOS v1.0

     LoadW    r0,#Data             ;point to record list
     LoadW    r1,#NUM_RECS         ;number of records to sort
     jsr      SortAlpha            ;go sort them
     rts                           ;exit

Data:
     .byte    "ZETA            "
     .byte    "0123456789ABcDeF"
     .byte    "gAMma over beta "
     .byte    "stewardesses    "
     .byte    "123ABC123abc    "
     .byte    "abcdefghijklmnop"
     .byte    "qrstuvwxyz012345"
     .byte    "stewardess      "
     .byte    "beta            "
     .byte    "alpha           "
     .byte    "delta           "
     .byte    "steward         "

NUM_RECS      = ((*-Data)/REC_SIZE)
.if   ( (Data + NUM_RECS*REC_SIZE) != *)
     .echo    Something is wrong with sort data
.endif
```

StartAppl (Apple, C64, C128) mid-level disk

Function: Warmstart GEOS and start an application that is already loaded into memory.

Parameters: *These are all passed on to the application being started.*
- **r7** START_ADDR — start address of application (word).
- **r0L** OPTIONS :
  - bit 7: 0 not passing a data file.
  - 1 **r2** and **r3** contain pointers to disk and data file names.
  - bit 6: 0 not printing data file.
  - printing data file; application should print file and exit.
- **r2** DATA_DISK — only valid if bit 7 or bit 6 of *OPTIONS* is set: pointer to name of the disk that contains the data file, usually a pointer to one of the **Drx CurDkNm** buffers (word).
- **r3** DATA_FILE — only valid if bit 7 or bit 6 of *OPTION* is set: pointer to name of the data file (word).

Returns: *never returns.*

Passes: *warmstarts GEOS and passes the following to the application at START_ADDR:*
- **r0** as originally passed to **StartAppl.**
- **r2** as originally passed to **StartAppl** (use **dataDiskName**).
- **r3** as originally passed to **StartAppl.**(use **dataFileName**).
- **dataDiskName** contains name of data disk if bit 7 of **r0** is set.
- **dataFileName** contains name of data file if bit 6 of **r0** is set.

Alters: GEOS brought to a warmstart state.

Destroys: n/a

Description: **StartAppl** warmstarts GEOS and **jsr**'s to *START_ADDR* as if the application had been loaded from the deskTop. **GetFile** and **LdApplic** call **StartAppl** automatically when loading an application.

StartAppl is useful for bringing an application back to its startup state. It completely warmstarts GEOS, resetting variables, intializing tables, clearing the processor stack, and executing the application's initialization code with a **jsr** from **MainLoop**.

Example:

See also: **LdApplic, GetFile, WarmStart.**

StartASCII (Apple, C64, C128) printer driver

Function: Enable ASCII text mode printing.

Parameters:

Commodore

r1 WORKBUF — pointer to a 640-byte work buffer for use by the printer driver.(word). **PrintASCII** uses this work area as an intermediate buffer; the buffer must stay intact throughout the entire page.

Apple

none.

Returns: **x** STATUS — printer error code; $00 = no error.

Destroys:

Commodore

assume **a, y, r0–r15.**

Apple

assume **a, y, r0–r4.**

Description: **StartASCII** enables ASCII text mode printing. An application calls **StartASCII** at the beginning of each page. It assumes that **InitForPrint** has already been called to initialize the printer.

C64 & C128: **StartASCII** takes control of the serial bus by opening a fake Commodore file structure and requests the printer (device 4) to enter listen mode. It then sends the proper control sequences to place the printer into text mode.

Apple: **StartASCII** sends the proper control sequences to place the printer into ASCII mode. It reestablishes the proper text mode as set by **SetMode**, but it does not reset the printer. This allows an application to mix graphics and ASCII text on the same page.

Example:

See also: **PrintASCII, StopPrint, StartPrint**

StartMouseMode (Apple, C64, C128) mouse/sprite

Function: Instructs GEOS to start or restart its monitoring of the input device (usually a mouse but depending on the input driver may be a joystick or other device).

Parameters:

r11	MOUSEX — x-position to start mouse at (word) If this parameter is $0000, then the mouse position is not changed and the mouse velocity is not altered.	
y	MOUSEY — y-position to start mouse at (byte).	
st	carry flag:	0 = same as setting MOUSEX to $0000. 1 = no effect.

Alters:

mouseVector	loaded with address of **SystemMouseService.**
mouseFaultVec	loaded with address of **SystemFaultService.**
faultData	$00
mouseXPos	
mouseYPos	
mouseOn	**MOUSEON_BIT** set by **MouseUp.**
mobenble	sprite #0 bit set by **MouseUp.**

Destroys: **a, x, y, r0–r15**

Description: **StartMouseMode** Instructs GEOS to start or restart its monitoring of the input device. Most normal GEOS applications will not need to call this routine because it is called internally by both **DoMenu** and **DoIcons.** If an application is not using icons nor menus, it should call **StartMouseMode** during its initialization.

StartMouseMode does the following:

1: If the carry flag is set and the *MOUSEX* parameter is non-zero, then *MOUSEX* is copied into **mouseXPos**, *MOUSEY* is copied into **mouseYPos**, and the input driver **SlowMouse** routine is called. If running under GEOS 128, *MOUSEX* is first passed through **NormalizeX** before getting loaded into **mouseXPos.**

2: The address of the internal **SystemMouseService** routine is loaded into **mouseVector** and the address of the internal **SystemFaultService** routine is loaded into **mouseFaultVec.**

3: A $00 is stored into **faultData**, clearing any mouse faults.

4: **MouseUp** is called to enable the mouse.

64 & 128: If the mouse will be repositioned, then disable interrupts around the call to **StartMouseMode.** It is not necessary to disable interrupts under Apple GEOS.

Example:

```
;Intialize the mouse and start it at screen center
MouseInit:
     LoadW    r11,#(SC_PIX_WIDTH/2)      ;screen center
     ldy      #(SC_PIX_HEIGHT/2)
     sec                                 ;set to move mouse
.if  (C64 || C128)
```

```
        php
        sei
        jsr     StartMouseMode
        plp
.else ;(APPLE)
        jsr     StartMouseMode
.endif
        rts
```

See also: ClearMouseMode, MouseUp, MouseOff, SlowMouse, DoMenu, DoIcons, TempHideMouse, HideOnlyMouse.

StartPrint (Apple, C64, C128) — printer driver

Function: Enable graphics-mode printing.

Parameters:

Commodore

r1 WORKBUF — pointer to a 1,920-byte work buffer for use by the printer driver.(word). **PrintBuffer** uses this work area as an intermediate buffer; this buffer must stay intact throughout the entire page.

Apple

none.

Returns: **x** STATUS — printer error code; $00 = no error.

Destroys:

Commodore

assume **a, y, r0–r15.**

Apple

assume **a, y, r0–r4.**

Description: **StartPrint** enables graphic printing. An application calls **StartPrint** at the beginning of each page. It assumes that **InitForPrint** has already been called to initialize the printer.

C64 & C128: **StartPrint** takes control of the serial bus by opening a fake Commodore file structure and requests the printer (device 4) to enter listen mode. It then sends the proper control sequences to place the printer into graphics mode.

Apple: **StartPrint** sends the proper control sequences to place the printer into graphics mode. It reestablishes the proper graphics mode as set by **SetMode**, but it does not reset the printer. This allows an application to mix ASCII text and graphics on the same page.

Example:

See also: **StopPrint, StartASCII.**

OUT OF ORDER → PUT AFTER StartPrint

StashRAM (C64 v1.3 & C128) memory

Function: Primitive for transferring data to an REU.

Parameters:
- **r0** CBMSRC — address in Commodore to start reading (word).
- **r1** REUDST — address in REU bank to put data (word).
- **r2** COUNT — number of bytes to stash (word).
- **r3L** REUBANK — REU bank number to stash to (byte).

Returns:
- **r0–r3** unchanged.
- **x** error code: $00 (no error) or **DEV_NOT_FOUND** if bank or REU not available.
- **a** REU status byte and'ed with $60 ($40 = successful stash).

Destroys: y

Description: **StashRAM** moves a block of data from Commodore memory into an REU bank. This routine is a "use at your own risk" low-level GEOS primitive

StashRAM uses the **DoRAMOp** primitive by calling it with a *CMD* parameter of %10010000.

Note: Refer to **DoRAMOp** for notes and warnings.

Example:

See also: **FetchRAM, SwapRAM, VerifyRAM, DoRAMOp, MoveBData.**

This page intentionally left blank to maintain right/left (verso/recto) page ordering. Final version will correct this.

StatusCard (Apple) — card driver

Function: Get the current input/output status of the printer card.

Parameters: none.

Returns:

x	STATUS — card error code; $00 = no error (byte)
st	sign flag: set if card ready to accept output. carry flag: set if card has input ready.

Destroys: **a, y.**

Description: **StatusCard** returns information about the current state of the card. The input-ready and ready-for-output flags are only valid if the card is capable of returning this type of information. This capability can be checked with a call to **InfoCard.**

Note: **StatusCard** must be called after an **OpenCard** and before a **CloseCard.**

Example:

See also: **InfoCard.**

StopPrint (Apple, C64, C128) — printer driver

Function: Flush output buffer and formfeed the printer (called at the end of each page).

Parameters:

Commodore

r0 TEMPBUF — pointer to a 640-byte area of memory that can be set to $00 (word).

r1 WORKBUF — pointer to a 1,920-byte work buffer used by **PrintBuffer** (word).

Apple

r1L FF_SUPPRESS — set to **TRUE** to suppress automatic formfeed after flushing the buffer; normally set to **FALSE**.

Returns:

Apple

x STATUS — printer error code; $00 = no error.

Destroys:

Commodore

assume **a, x, y, r0–r15**.

Apple

assume **a, y, r0–r4**.

Description: **StopPrint** instructs the printer driver to flush any internal buffers and end the page.

StopPrint ends both graphic and ASCII printing.

C64 & C128: Commodore GEOS printer drivers always formfeed when **StopPrint** is called.

Apple: Apple GEOS printer drivers can suppress the normal **StopPrint** formfeed, thereby flushing the print buffers but allowing the application to resume printing at the current line. This way ASCII text and graphics can be mixed on a page with alternating calls to **StartASCII**, **StopPrint**, and **StartPrint**.

Example:

See also: **StartPrint, StartASCII.**

SwapBData (C128) memory

Function: Swaps two regions of memory within either front RAM or back RAM (or between one bank and the other).

Parameters:

r0	ADDR1 — address of first block in application memory (word).
r1	ADDR2 — address of second block in application memory (word).
r2	COUNT — number of bytes to swap (word).
r3L	A1BANK — *ADDR1* bank: 0 = front RAM; 1 = back RAM (byte).
r3H	A2BANK — *ADDR2* bank: 0 = front RAM; 1 = back RAM (byte).

Returns: **r0–r3** unchanged.

Destroys: **a, x, y**

Description: **SwapBData** is a block swap routine that allows data to be swapped in either front RAM, back RAM, or between front and back. If the *ADDR1* and *ADDR2* areas are in the same bank and overlap, *ADDR2*. must be less than *ADDR1*.

SwapBData is especially useful for swapping data from front RAM to back RAM or from back RAM to front RAM.

SwapBData uses the **DoBOp** primitive by calling it with a *MODE* parameter of $02.

Note: **SwapBData** should only be used to swap data within the designated application areas of memory.

Example:

See also: **MoveBData, VerifyBData, DoBOp.**

SwapMainAndAux (Apple) memory

Function: Swaps a block of memory on a byte-by-byte basis between application main memory and application aux memory.

Parameters:
- **r0** SOURCE — address of block to swap (word).
- **r2** COUNT — number of bytes to swap (0 – 48K).

Destroys: **a, y, r0, r1L, r2**

Description: **SwapMainAux** swaps a block of bytes that lie at the same address in both main and auxiliary application memory. This is really only useful in the area between $6000 and $8000, where application main and application aux memory occupy the same address space.

Example:

See also: **MoveAuxData, MoveBData, SwapBData, SwapRAM.**

SwapRAM (C64 v1.3 & C128) memory

Function: Primitive for swapping data between Commodore memory and an REU.

Parameters:
- **r0** CBMADDR — address in Commodore to swap (word).
- **r1** REUADDR — address in REU to swap (word).
- **r2** COUNT — number of bytes to swap (word).
- **r3L** REUBANK — REU bank number to fetch from (byte).

Returns:
- **r0–r3** unchanged.
- **x** error code: $00 (no error) or **DEV_NOT_FOUND** if *REUBANK* or REU not available.
- **a** REU status byte and'ed with $60 ($40 = successful swap).

Destroys: y

Description: **SwapRAM** swaps a block of data in an REU bank with a block of data in Commodore memory. This routine is a "use at your own risk" low-level GEOS primitive

SwapRAM uses the **DoRAMOp** primitive by calling it with a *CMD* parameter of %10010010.

Note: Refer to **DoRAMOp** for notes and warnings.

Example:

See also: **StashRAM, FetchRAM, VerifyRAM, DoRAMOp, SwapBData.**

TempHideMouse (Apple, C128) mouse/sprite

Function: Temporarily removes soft-sprites and the mouse pointer from the graphics screen.

Parameters: nothing.

Returns: nothing.

Uses:

graphMode	(128 only).
offFlag	(Apple only).
noEraseSprites	(Apple only).

Alters:

offFlag	set to **TRUE** (Apple only).

Destroys: **a, x**

Description: **TempHideMouse** temporarily removes all soft-sprites (mouse pointer *and* sprites 2–7) unless they are already removed. This routine is called by all GEOS graphics routines prior to drawing to the graphics screen so that software sprites don't interfere with the graphic operations. An application that needs to do direct screen access should call this routine prior to modifying screen memory.

The sprites will remain hidden until the next pass through **MainLoop**.

128: In 40-column mode (bit 7 of **graphMode** is zero), **TempHideMouse** exits immediately without affecting the hardware sprites.

Apple: If bit 7 of **offFlag** is set, **offFlag** is set to **TRUE** (reaffirming the fact that the sprites and mouse are marked as hidden) and exits without affecting any sprites. If **noEraseSprites** is set to **TRUE**, sprites 2–7 are not erased from the screen. This is useful for keeping sprites on the screen (and, therefore, avoiding any flicker) if the application is confident that the graphic operation will not be in the area of a sprite.

Example:

See also: **HideOnlyMouse**.

TestPoint (Apple, C64, C128) graphics

Function: Test and return the value of a single point (pixel).

Parameters:
r3 X1 — x-coóridnate of pixel (word).
r11L Y1 — y-coordinate of pixel (byte).

where (X1,Y1) is the coordinate of the point to test.

Uses: **dispBufferOn**
bit 6 — if set, test pixel in foreground screen.
bit 7 — if set, test pixel in background buffer.
(If both bit 6 and bit 7 are set, then only the pixel in the background screen is tested.).

Returns: **r3, r11L** unchanged.
st carry flag set if point set; cleared if point clear.

Destroys: **a, x, y, r5–r6**

Description: **TestPoint** will test a pixel in either the foreground screen or the background buffer (or both simultaneously) and return the pixel's status by either setting or clearing the carry (x) flag accordingly. The **jsr TestPoint** is usually followed immediately by a **bcc** or **bcs** so that a set or clear pixel may be handled appropriately.

128: Under GEOS 128, or'ing **DOUBLE_W** into the *X1* will automatically double the x-position in 80-column mode. Or'ing in **ADD1_W** will automatically add 1 to a doubled x-position. (Refer to "GEOS 128 X-position and Bitmap Doubling" in Chapter.@gr@ for more information.)

Example:

See also: **DrawPoint.**

ToBasic (C64, C128) utility

Function: Removes GEOS and passes control to Commodore BASIC with the option of loading a non-GEOS program file (BASIC or assembly-language) and/or executing a BASIC command.

Parameters:

- **r0** CMDSTRING — pointer to null-terminated command string to send to BASIC interpreter.
- **r5** DIR_ENTRY — pointer to the directory entry of a standard Commodore file (PRG file type), which itself can be either a BASIC or ASSEMBLY GEOS-type file. If this parameter is $0000, then no file will be loaded.
- **r7** LOADADDR — if **r5** is non-zero, then this is the file load address. For a BASIC program, this is typically $801. If **r5** is zero and a tokenized BASIC program is already in memory, then this value should point just past the last byte in the program. If **r5** is zero and no program is in memory, this value should be $803, and the three bytes at $800–$802 should be $00.

Returns: n/a

Destroys: n/a

Description: **ToBasic** gives a GEOS application the ability to run a standard Commodore assembly-language or BASIC program. It removes GEOS, switches in the BASIC ROM and I/O bank, loads an optional file, and sends an optional command to the BASIC interpreter.

Once **ToBasic** has executed, there is no way to return directly to the GEOS environment unless the RAM areas from $c000 through $c07f are preserved (those bytes may be saved and restored later). To return to GEOS, the called program can execute a jump to $c000 (**BootGEOS**).

A program in the C64 environment can check to see if it was loaded by GEOS by checking the memory starting at $c006 for the ASCII (not CBMASCII) string "GEOS BOOT". If loaded by GEOS, the program can check bit 5 of $c012: if this bit is set, ask the user to insert their GEOS boot disk; if this bit is clear, GEOS will reboot from the RAM expansion unit. To actually return to GEOS, set **CPU_DATA** to $37 (**KRNL_BAS_IO_IN**) and jump to $c000 (**BootGEOS**).

Example:

```
;*******************************************************
;LoadBASIC:
;     Loads a Commodore BASIC program and starts it
;running. Assumes that the program is a standard BASIC
;file that loads at $801. This example does little
;error checking.
;
;Pass:          r5   =      pointer to BASIC program's directory
;                           entry.
;
;*******************************************************
LoadBASIC:
     LoadW     r0,#RunCommand       ;point at command string
     LoadW     r7,#$801             ;assume standard address
```

```
        jsr     ToBasic             ;expect to not return
        brk                         ;should never get here!

;Text command to send to BASIC interpeter so BASIC
;program is started up properly.
RunCommand:
        .byte   "RUN",NULL
```

UnblockProcess (Apple, C64, C128) process

Function: Allow a process's events to go through.

Parameters: **x** PROCESS — number of process (0 – *n*–1, where *n* is the number of processes in the table) (byte).

Returns: **x** unchanged.

Destroys: **a**

Description: **UnblockProcess** causes **MainLoop** to again recognize a process's runable flag so that if a process timer reaches zero (causing the process to become runable) an event will be generated.

Because the GEOS Interrupt Level continues to decrement the countdown timer as long as the process is not frozen, a process may become runable while it is blocked. As long as the process is blocked, however, **MainLoop** will ignore the runable flag. When the process is subsequently unblocked, **MainLoop** will recognize a set runable flag as a pending event and call the appropriate service routine. Multiple pending events are ignored: if a blocked process's timer reaches zero more than once, only one event will be generated when it is unblocked. To prevent a pending event from happening, use **RestartProcess** to unblock the process.

Note: If a process is not blocked, an unnecessary call to **UnblockProcess** will have no effect.

Example:

See also: **BlockProcess, UnfreezeProcess, EnableProcess, RestartProcess.**

UnfreezeProcess (Apple, C64, C128) process

Function: Resume (unfreeze) a process's countdown timer.

Parameters: **x** PROCESS — number of process (0 – *n*–1, where *n* is the number of processes in the table) (byte).

Returns: **x** unchanged.

Destroys: **a**

Description: **UnfreezeProcess** causes a frozen process's countdown timer to resume decrementing. The value of the timer is unchanged; it begins decrementing again from the point where it was frozen. If a process is not frozen, a call to **UnfreezeProcess** will have no effect.

Note: If a process is not frozen, a call to **UnfreezeProcess** will have no effect.

Example:

See also: **FreezeProcess, BlockProcess.**

UpdateMouse (Apple, C64, C128) — input driver

Function: Update the mouse variables based on any changes in the state of the input device.

Parameters: Commodore:
none.

Apple:
st carry flag: 0 interrupt was caused by mouse card.
1 interrrupt was not caused by mouse card.

Returns: nothing

Alters:

mouseXPos	mouse x-position.
mouseYPos	mouse y-position.
mouseData	state of mouse button: high bit set if button is released; clear if pressed.
pressFlag	MOUSE_BIT and INPUT_BIT set appropriately.
inputData	depends on device.

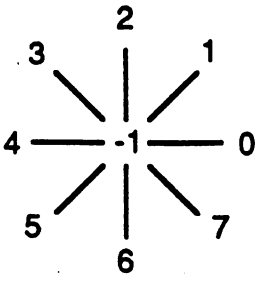
Destroys: assume **a, x, y, r0–r15**

Description: GEOS calls **UpdateMouse** at Interrupt Level to update the GEOS mouse variables with the actual state of the input device. An application should never need to call **UpdateMouse**.

A typical input driver's **UpdateMouse** routine will scan the device hardware and update **MouseXPos** and **MouseYPos** with new positions if the coordinates have changed. It will also update **mouseData** with the current state of the input button (high-bit set if released; cleared if pressed) and set **MOUSE_BIT** in **pressFlag** if the button state has changed since the last call to **UpdateMouse**.

The four byte **inputData** field, which was originally for device-dependent information, has adopted the following standard offsets:

inputData+0 (byte) 8-position device direction (joystick direction; mouse drivers convet a moving mouse to an appropriate direction):



inputData+1 (byte) current speed (Commodore joystick drivers only).

Standard GEOS input drivers should set the **INPUT_BIT** of **pressFlag** if **inputData+0** has changed since the last time **UpdateMouse** was called.

Because most GEOS applications leave **inputVector** set to its default $0000 value, setting this bit will usually have no effect.

Apple: The Apple GEOS version of **UpdateMouse** will treat **mouseYPos** as a two-byte value, propagating any underflow into the high-byte at **mouseYPos+1**. This gives the subsequent mouse-fault check enough precision to detect whether the user moved the mouse off the top or the bottom of the screen (a negative number = off the top edge). The GEOS mouse fault routine will always clear this high-byte after constraining the mouse to the screen edges.

See also: **SlowMouse, InitMouse, SetMouse, KeyFilter.**

UpdateParent (Apple) mid-level disk

Function: Update data om current directory header.

Parameters: **a** UPDATE_FLAGS — operations to perform (word).

DEC_NUM_FILES	— decrement file count.
INC_NUM_FILES	— increment file count.
MOD_DATE	— update modification date.
CREATE_DATE	— set creation.
DEC_DIR_BLKS	— decrement directory block count.
DEC_DIR_BLKS	— increment directory block count.

multiple changes can be made in one call by bitwise or'ing these constants together.

Uses: **curDirHead** should contain current directory header.

Returns: **x** error ($00 = no error).
r10L directory page actually used.

Alters: **curDirHead** contains newly-changed current directory header.
diskBlkBuf contains newly-changed parent directory header unless current directory is volume directory (in which case there is no parent header).

Destroys: **a, y, r1, r4, r5.**

Description: **UpdateParent** upates the current and parent directory headers according to the flags passed in *UPDATE_FLAGS*. Whenever a the contents of a subdirectory are altered — whether adding or deleteing files or other subdirectores — the current and parent directory headers need to be updated to reflect their new contents. The high-level GEOS routines will automatically call **UpdateParent** when they alter the contents of a subdirectory.

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

UpdateRecordFile (Apple, C64, C128) VLIR disk

Function: Update the disk copy of the VLIR index table, BAM or VBM, and other VLIR information such as the file's time/date-stamp. This update only takes place if the file has changed since opened or last updated..

Parameters: none.

Uses:

curDrive	
fileWritten†	if **FALSE**, no updating occurs because file has not been written to.
fileHeader	VLIR index table stored in this buffer.
fileSize	total number of disk blocks used in file (includes index block, GEOS file header, and all records).
dirEntryBuf	directory entry of VLIR file.
year, month, day, hours, minutes	for date-stamping file.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	this buffer must contain the current directory header.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

fileBytes	total number of bytes in file (written to bytes 254, 255, 511, and 512 of the master index block).
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).

Alters:

fileWritten†	set to **FALSE** to indicate that file hasn't been altered since last updated.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, y, r1, r4, r5.**

Description: **UpdateRecordFile** checks the **fileWritten** flag. If the flag is **TRUE**, which indicates the file has been altered since it was last updated, **UpdateRecordFile** writes the various tables kept in memory out to disk (e.g., index table, BAM/VBM) and time/date-stamps the directory entry. If the **fileWritten** flag is **FALSE**, it does nothing.

UpdateRecord writes out the index block, adds the time/date-stamp and **fileSize** information to the directory entry, and writes out the new BAM/VBM with a call to **PutDirHead** or **PutVBM** (Commodore GEOS and Apple GEOS, respectively).

C64 & C128: Because Commodore GEOS stores the BAM in global memory, the application must be careful not to corrupt it before the VLIR file is updated. If the **fileWritten** flag is **TRUE** and the BAM is reread from disk, the old copy (on disk) will overwrite the current copy in memory. In the normal use of VLIR disk routines, where a file is opened, altered, then closed before any other disk routines are executued, no conflicts will arise.

Example:

See also: **CloseRecordFile, OpenRecordFile.**

UpDirectory (Apple) — high-level disk

Function: Makes the parent directory the current directory.

Parameters: none.

Uses:
curDrive
curKBlkno current directory.

Returns:
x error ($00 = no error).
y pathname status ($00 = OK; **BFR_OVERFLOW** = pathname longer than **pathnameBuf**).

Alters:
curKBlkno new current directory.
curDirHead header of new directory.
pathnameBuf† system pathname buffer updated to reflect new path.
curDirTabLo†
curDirTabHi†

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: **a, r0L, r1, r2, r4.**

Description: **UpDirectory** moves up one level in the hierarchical file system, making the parent directory the current working directory. If **UpDirectory** is called when the root directory is active, an **AT_ROOT_DIR** error is returned.

UpDirectory first changes the current key block number, then updates the pathname in **pathnameBuf** calling **GetPathname** if necessary. The current directory header is read in with a call to **GetDirHead**

C64 & C128: Commodore GEOS does not support a hierarchical file system.

Example:

See also: **SetGDirEntry, DeleteDir.**

UseSystemFont (Apple, C64, C128) text

Function: Begin using default system font (BSW 9)

Parameters: none.

Returns: nothing.

Alters:

curHeight	height of font.
baselineOffset	number of pixels from top of font to baseline.
cardDataPntr	pointer to current font image data.
curIndexTable	pointer to current font index table.
curSetWidth	pixel width of font bitstream in bytes.

Destroys: **a, x, y, r0.**

Description: **UseSystemFont** calls **LoadCharSet** with the address of the always-resident BSW 9 font.

128: In 80-column mode a double-width BSW 9 font is substituted.

Example:

See also: **LoadCharSet, LoadAuxSet.**

VerifyBData (C128) memory

Function: Compares (verifies) two regions of memory against each other. The regions may either be in front RAM or back RAM (or one in front and the other in back).

Parameters:

r0	ADDR1 — address of first block in application memory (word).
r1	ADDR2 — address of second block in application memory (word).
r2	COUNT — number of bytes to compare/verify (word).
r3L	A1BANK — *ADDR1* bank: 0 = front RAM; 1 = back RAM (byte).
r3H	A2BANK — *ADDR2* bank: 0 = front RAM; 1 = back RAM (byte).

Returns:

r0–r3	unchanged.
x	$00 if data matches; $ff if mismatch.

Destroys: **a, y**

Description: **VerifyBData** is a block verify routine that allows the data in one region of memory to be compared to the data in another region in memory. The regions may be in either front RAM, back RAM, or in front and back. The *ADDR1* and *ADDR2* areas may overlap even if they are in the same bank.

VerifyBData uses the **DoBOp** primitive by calling it with a *MODE* parameter of $03.

Note: **VerifyBData** should only be used to compare data within the designated application areas of memory.

Example:

See also: **MoveBData, SwapBData, DoBOp.**

VerifyRAM (C64 v1.3 & C128) memory

Function: Primitive for verifying (comparing) data in Commodore memory with data in an REU.

Parameters:

r0	CBMADDR — address in Commodore to start (word).	
r1	REUADDR — address in REU bank to start (word).	
r2	COUNT — number of bytes to verify/compare (word).	
r3L	REUBANK — REU bank number to compare with (byte).	

Returns:

r0–r3	unchanged.		
x	error code:	$00 (no error) or **DEV_NOT_FOUND** if bank or REU not available.	
a	REU status byte and'ed with $60:	$40	data match
		$20	data mismatch

Destroys: y

Description: **VerifyRAM** compares a block of data in Commodore memory with a block of data in an REU bank. This routine is a "use at your own risk" low-level GEOS primitive

VerifyRAM uses the **DoRAMOp** primitive by calling it with a *CMD* parameter of %10010011.

Note: Refer to **DoRAMOp** for notes and warnings.

Example:

See also: **StashRAM, FetchRAM, SwapRAM, DoRAMOp, VerifyBData.**

VerticalLine (Apple, C64, C128) graphics

Function: Draw a vertical line with a repeating bit-pattern.

Parameters:
a eight-bit repeating pattern to use (*not* a GEOS pattern number).
r4 X1 — x-coordinate of line (word).
r3L Y1 — y-coordinate of topmost endpoint (byte).
r3H Y2 — y-coordinate of bottomost endpoint (byte).

where (X1,Y1) and (X1,Y2) define the endpoints of the vertical line.

Uses: **dispBufferOn:**
bit 7 — write to foreground screen if set.
bit 6 — write to background screen if set.

Returns: **r3L, r3H, r4** unchanged.

Destroys:
Commodore:
a, x, y, r5L–r8L

Apple:
a, x, y

Description: **VerticalLine** sets and clears pixels on a single vertical line according to the eight-bit repeating pattern. Wherever a 1-bit occurs in the pattern byte, a pixel is set, and wherever a 0-bit occurs, a pixel is cleared.

Bits in the pattern byte are used top-to-bottom, where bit 7 is at the top.A bit pattern of %11110000 would create a vertical line like:

The pattern byte is always drawn as if aligned to a card boundary. If the endpoints of a line do not coincide with card boundaries, then bits are masked off the appropriate ends. The effect of this is that a pattern is always aligned to specific pixels, regardless of the endpoints, and that adjacent lines drawn in the same pattern align.

Note: To draw patterned vertical lines using the 8x8 GEOS patterns, draw rectangles of one-pixel width by calling the GEOS **Rectangle** routine with identical x-coordinates.

VerWriteBlock (C64, C128)

very low-level disk

Function: Very low-level verify block on disk.

Parameters:
- **r1L** TRACK — valid track number (byte).
- **r1H** SECTOR — valid sector on track (byte).
- **r4** BUFFER — address of buffer of **BLOCKSIZE** bytes that contains data that should be on this sector (word).

Uses:
- **curDrive** currently active disk drive.
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns: **x** error ($00 = no error).

Destroys: **a, y.**

Description: **VerWriteBlock** verifies the validity of a recently written block. If the block does not verify, the block is rewritten by calling **WriteBlock**. **VerWriteBlock** is a low-level disk routine and expects the application to have already called **EnterTurbo** and **InitForIO**.

VerWriteBlock can be used to accelerate the verifies that accompany multiple-sector writes by first writing all the sectors and then verifying them. This is often faster than verifying a sector immediately after writing it because when writing sequential sectors, the GEOS turbo code will catch the sector interleave. If a sector is written and then immediately verified, the turbo code will need to wait for the disk to make one complete revolution before the newly-written sector will again pass under the read/write head. By writing all the sectors first and catching the interleave, then verifying all the sectors (again, catching the interleave), the dead time when the turbo code is waiting for the disk to spin around is minimized. Many of the higher-level disk routines that write multiple blocks do just this.

VerWriteBlock is useful for multiple-sector disk operations where speed is an issue and the standard GEOS routines don't offer a decent solution. **VerWriteBlock** can function as the foundation of specialized, high-speed disk routines.

Note: **VerWriteBlock** does not always do a byte-by-byte compare with the data in *BUFFER*. Some devices, such as the Commodore 1541, can do a cyclic redundancy check on the data in the block, and this internal checksum is sufficient evidence of a good write. Other devices, such as RAM-expansion units, have built-in byte-by-byte verifies.

Apple: **VerWriteBlock** does not exist in Apple GEOS. To verify a block, read it into a general purpose buffer (**diskBlkBuf**, for example). If the block reads without an error, it verified. The ProDos device driver does its own checksum on data blocks, so a byte-by-byte comparison of the data isn't necessary to determine if the block was properly written — rereading the block without an error is sufficient evidence of a good write. See **PutBlock** for more information.

Example: See **WriteBlock**.

See also: **WriteBlock, PutBlock.**

WarmStart (Apple) internal

Function: Executes portions of the GEOS warmstart procedure (the same warmstart executed before a new application, desk accessory, or dialog box is started up).

Parameters: none

Returns: GEOS variables in a warmstart state; stack and application space unaffected.

Destroys: **a, x, y, r0–r2**

Description: **WarmStart** is part of the GEOS warmstart procedure. It resets some GEOS variables and data structures (both global and local) to their default state, the state they are in when an application, desk accessory, or dialog box is started up. GEOS calls **WarmStart** internally, and is of little use to an application. An application that needs to place GEOS in a warmstart state (for restarting, for example), is better off using **StartAppl.**

See also: **FirstInit, StartAppl.**

WriteBlock (C64, C128) — very low-level disk

Function: Very low-level write block to disk.

Parameters:
- **r1L** TRACK — valid track number (byte).
- **r1H** SECTOR — valid sector on track (byte).
- **r4** BUFFER — address of buffer of **BLOCKSIZE** bytes that contains data to write out (word).

Uses:
- **curDrive** currently active disk drive.
- **curType** GEOS 64 v1.3 and later: for detecting REU shadowing.

Returns: **x** error ($00 = no error).

Destroys: **a, y.**

Description: **WriteBlock** writes the block at *BUFFER* to the specified *TRACK* and *SECTOR*. If the disk is shadowed, **WriteBlock** will also write the data to the shadow memory. **WriteBlock** is pared down version of **PutBlock**. It expects the application to have already called **EnterTurbo** and **InitForIO**, and it does not verify the data after writing it.

WriteBlock can be used to accelerate multiple-sector writes and their accompanying verifies by writing all the sectors first and then verifying them. This is often faster than verifying a sector immediately after writing it because when writing sequential sectors, the GEOS turbo code will catch the sector interleave. If a sector is written and then immediately verified, the turbo code will need to wait for the disk to make one complete revolution before the newly-written sector will again pass under the read/write head. By writing all the sectors first and catching the interleave, then verifying all the sectors (again, catching the interleave), the dead time when the turbo code is waiting for the disk to spin around is minimized. Many of the higher-level disk routines that write multiple blocks do just this.

WriteBlock is useful for multiple-sector disk operations where speed is an issue and the standard GEOS routines don't offer a decent solution. **WriteBlock** can function as the foundation of specialized, high-speed disk routines.

Apple: Apple GEOS has no **WriteBlock** equivalent. Use **PutBlock** instead. To write a block without verifying the data, set **numDiskRetries** to $00 before calling **PutBlock**. See **PutBlock** for more information.

Example:

```
;Write sector from diskBlkBuf to disk and then verify
;if necessary.
;
;Pass:
;     track     track number
;     sector    sector on track
;     verify    verify data? (TRUE == YES)
;
;Returns:
;     x              error code
;
```

```
MyPutBlock:
     LoadW    r4,#diskBlkBuf      ;where to get data from
     MoveB    r1L,track           ;track number
     MoveB    r1H,sector          ;sector number
     jsr      EnterTurbo          ;go into turbo mode
     txa                          ;set status flags
     bne      99$                 ;branch if error found
     jsr      InitForIO           ;prepare for serial I/O
     jsr      WriteBlock          ;primitive write block
     txa                          ;set status flags
     bne      80$                 ;branch if error found
     lda      verify              ;check verify flag
     beq      80$                 ;branch if not verifying
     jsr      VerWriteBlock       ;verify block we wrote
80$:
     jsr      DoneWithIO          ;restore after I/O done
99$:
     rts                          ;exit
```

See also: **PutBlock, ReadBlock, VerWriteBlock.**

WriteFile (Apple, C64, C128) — mid-level disk

Function: Write data to a chained list of disk blocks.

Parameters: **r7** DATA — pointer to start of data (word).

Commodore:
r6 TSTABLE — pointer to a track/sector list of blocks to write data to (unused but allocated in the BAM), usually a pointer to **fileTrScTab+2**; **BlkAlloc** can be used to build such a list. Apple GEOS uses the table in **INDEXBLOCKBUF** (word).

Uses: **curDrive**

Commodore:
curType GEOS 64 v1.3 and later: for detecting REU shadowing.

Apple:
RWbank bank *BUFFER* is in (**MAIN** or **AUX**).
INDEXBLOCKBUF index block for chain (in auxiliary memory; see below for information on accessing this buffer).

Returns: **x** error ($00 = no error).

Destroys: Apple:
a, y, r1, r4.

Commodore:
a, y, r1–r2, r4, r6–r7.

Description: **WriteFile** writes data from memory to disk. The disk blocks are verified, and any blocks that don't verify are rewritten.

Although the name "WriteFile" implies that it writes "files," it actually writes a chain of blocks and doesn't care if this chain is an entire sequential file or merely a VLIR record.

C64 & C128: **WriteFile** uses the track/sector table at *TSTABLE* as a list of linked blocks that comprise the chain. The end of the chain is marked with a track/sector pointer of $00,$ff. **WriteFile** copies the next 254 bytes from the data area to **diskBlkBuf+2**, looks two-bytes ahead in the *TSTABLE* for the pointer to the *next* track/sector, and copies those two-bytes to **dskBlkBuf+0** and **dskBlkBuf+1**. **WriteFile** then writes this block to disk. This is repeated until the end of the chain is reached.

WriteFile does not flush the BAM (it does not alter it either — it assumes the blocks in the track/sector table have already been allocated). See **BlkAlloc**, **SetNextFree**, and **AllocateBlock** for information on allocating blocks. See **PutDirHead** for more information on writing out the BAM.

Apple: Apple GEOS uses the internal index block buffer **INDEXBLOCKBUF** (in auxiliary memory) as a list of blocks that comprise the chain. Applications cannot

directly access this buffer, but **MoveAuxData** can be used to copy the index block from the appliciation's memory space to **INDEXBLOCKBUF**:

```
AUXtoMAIN       == %10000000
MAINtoAUX       == %01000000

        LoadW   r0, #myIndexBlock       ;copy from app. main
        LoadW   r1, #INDEXBLOCKBUF      ;to temp high aux
        LoadW   r2, #BLOCKSIZE          ;move a full block
        lda     #MAINtoAUX              ;copy main to aux
        jsr     MoveAuxData
```

WriteFile *does not* write the index block out to disk.

WriteFile does not flush the VBM cache (it does not alter it either — it assumes the blocks in the index block have already been allocated). See **BlkAlloc**, **SetNextFree**, and **AllocateBlock** for information on allocating blocks. See **PutVBM** for more information on flushing the VBM cache.

WriteFile first writes the blocks out in a tight loop, then makes a second pass, rereading each into **diskBlkBuf** to verify the data. If a verify error occurs, the *entire* file is rewritten.

Example:

See also: **SaveFile, WriteRecord, ReadFile.**

WriteRecord (Apple, C64, C128) VLIR disk

Function: Write data to the current VLIR record.

Parameters:

r2	BYTES — data bytes to write to record. Commodore version can write up to 32,258 bytes (127 Commodore blocks); Apple version can write up to 64,512 (126 ProDOS blocks) (word).
r7	RECDATA — pointer to start of record data (word).

Uses:

curDrive	
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM/VBM into memory.
curRecord	current record pointer.
fileHeader	VLIR index table stored in this buffer.

Commodore:

curType	GEOS 64 v1.3 and later: for detecting REU shadowing.
curDirHead	current directory header/BAM.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

RWbank	bank *RECDATA* is written from (**MAIN** or **AUX**).
curVBlkno†	used by VBM cacheing routines.
VBMchanged†	used by VBM cacheing routines.
numVBMBlks†	used by VBM cacheing routines.

†*used internally by GEOS disk routines; applications generally don't use.*

Returns:

x	error ($00 = no error).

Alters:

fileWritten†	set to **TRUE** to indicate the file has been altered since last updated.
fileHeader	index table adjusted to point to new chain of blocks for current record.
fileSize	adjusted to reflect new size of file.

Commodore:

fileTrScTab	Contains track/sector table for record as returned from **BlkAlloc**. The track and sector of the first block in the record is at **fileTrScTab**+0 and **fileTrScTab**+1. The end of the table is marked with a track value of $00.
curDirHead	current directory header/BAM modified by write operation.
dir2Head†	(BAM for 1571 and 1581 drives only)
dir3Head†	(BAM for 1581 drive only)

Apple:

fileBytes	adjusted to reflect new size of file.

†*used internally by GEOS disk routines; applications generally don't use.*

Destroys: Apple:
a, y, r1–r4, r6, r7, r8H.

Commodore:
a, y, r0–r9.

Description: **WriteRecord** writes data to the current record. All blocks previously associated with the record are freed. **BlkAlloc** is then used to allocate enough new blocks to hold *BYTES* amount of data (Apple GEOS will allocate one ProDOS index block in addition to the data blocks). The data is then written to the chain of sectors by calling **WriteFile**. The **fileSize** variable is updated to reflect the new size of the file. Apple GEOS also updates **fileBytes**.

WriteRecord does does not write the BAM/VBM and internal VLIR file information to disk. Call **CloseRecordFile** or **UpdateRecordFile** when done to update the disk with this information.

Note: **WriteRecord** correctly handles the case where the number of bytes to write (*BYTES*, **R2**) is zero. The record is freed and marked as allocated but not in use.

Apple: When **WriteRecord** returns, the index block of the record just written is in the aux. memory buffer **INDEXBLOCKBUF**. Applications cannot access this buffer directly. **MoveAuxData** can be used, however, to copy the block into the appliciation's memory space if access to it is necessary:

```
AUXtoMAIN       == %10000000
MAINtoAUX       == %01000000

      LoadW     r0, #INDEXBLOCKBUF          ;copy from index blk
      LoadW     r1, #diskBlkBuf             ;to temp buffer
      LoadW     r2, #BLOCKSIZE              ;move a full block
      lda       #AUXtoMAIN                  ;copy aux to main
      jsr       MoveAuxData
```

Example:

See also: **ReadRecord, WriteFile.**

Name	C64	C128	Apple	Description
AllocateBlock	$9048	$9048	$0300	Mark a disk block as in-use.
AppendRecord	$c289	$c289	$0330	Insert a new VLIR record after the current record.
AuxDExit	n/a	n/a	$efa9	Aux-driver deinstall and exit routine.
AuxDInt	n/a	n/a	$efa3	Aux-driver interrupt level routine.
AuxDKeyFilter	n/a	n/a	$efa6	Aux-driver keypress filter.
AuxDMain	n/a	n/a	$efa0	Aux-driver MainLoop level routine.
BBMult	$c160	$c160	$fecc	Byte by byte (single-precision) unsigned multiply.
Bell	n/a	n/a	$ff86	1000 Hz Bell sound.
BitmapClip	$c2aa	$c2aa	$fe5a	Display a compacted bitmap, clipping to a sub-window.
BitmapUp	$c142	$c142	$fe54	Display a compacted bitmap without clipping.
BitOtherClip	$c2c5	$c2c5	$fe5d	BitmapClip with data coming from elsewhere (e.g., disk)
BldGDirEntry	$c1f3	$c1f3	$036f	Build a GEOS directory entry in memory.
BlkAlloc	$c1fc	$c1fc	$0351	Allocate space on disk.
BlockProcess	$c10c	$c10c	$feb4	Block process from running. Does not freeze timer.
BMult	$c163	$c163	$fecf	Byte by word unsigned multiply.
BootGEOS	$c000	$c000	n/a	Reboot GEOS. Requires only 128 bytes at $c000.
CalcBlksFree	$c1db	$c1db	$0324	Calculate total number of free disk blocks.
CallRoutine	$c1d8	$c1d8	$ff08	pseudo-subroutine call. $0000 aborts call.
CancelPrint	n/a	n/a	$6018	Cancel printing, clearing printer and I/O card buffers.
ChangeDiskDevice	$c2bc	$c2bc	n/a	Instruct CBM drive to change its serial device address.
ChkDkGEOS	$c1de	$c1de	n/a	Check CBM disk for GEOS format.
ClearCard	n/a	n/a	$6715	Clear any buffered I/O operations.
ClearMouseMode	$c19c	$c19c	$fe9f	Stop input device monitoring.
ClearRam	$c178	$c178	$fefc	Clear memory to $00.
ClockInt	n/a	n/a	$0803	Clock driver interrupt level routine.
CloseCard	n/a	n/a	$670c	Close access to I/O card.
CloseRecordFile	$c277	$c277	$0333	Close currently open VLIR file.
CmpFString	$c26e	$c26e	$fef0	Compare two fixed-length strings.
CmpString	$c26b	$c26b	$feed	Compare two null-terminated strings.
CopyFString	$c268	$c268	$feea	Copy a fixed-length string.
CopyFullScreen	n/a	n/a	$ff92	Fast vertical screen copy.
CopyLine	n/a	n/a	$ff5f	Bit-boundary horizontal line copy.
CopyScreenBlock	n/a	n/a	$ff5c	Bit-boundary rectangle copy.
CopyString	$c256	$c256	$fee7	Copy a null-terminated string.
CRC	$c20e	$c20e	$ff14	Cyclic Reduncancy Check calculation.
Dabs	$c16f	$c16f	$fedb	Double-precision signed absolute value.
Ddec	$c175	$c175	$fee1	Double-precison unsigned decrement.
Ddiv	$c169	$c169	$fed5	Double-precision unsigned division.
DeleteDir	n/a	n/a	$037e	Delete directory.
DeleteFile	$c238	$c238	$0357	Delete file.
DeleteRecord	$c283	$c283	$0336	Delete current VLIR record.
DisablSprite	$c1d5	$c1d5	$feab	Disable sprite.
DivideBySeven	n/a	n/a	$ff68	Quick division by seven for direct screen access.
DMult	$c166	$c166	$fed2	Double-precision unsigned multiply.
Dnegate	$c172	$c172	$fede	Double-precision signed negation.
DoBOp	n/a	$c2ec	n/a	C128-backram memory primitive.
DoDlgBox	$c256	$c256	$ff17	Display and begin interaction with dialog box.
DoIcons	$c15a	$c15a	$fe0c	display and begin interaction with icons.
DoInlineReturn	$c2a4	$c2a4	$fe09	Return from inline subroutine.
DoMenu	$c151	$c151	$fe0f	Display and begin interaction with menus.
DoneWithIO	$c25f	$c25f	n/a	Restore system after I/O across CBM serial bus.
DoPreviousMenu	$c190	$c190	$fe15	Retract sub-menu and reactivate menus up one level.
DoRAMOp	$c2d4	$c2d4	n/a	CBM RAM-expansion unit access primitive.
DownDirectory	n/a	n/a	$0381	Open subdirectory.
DrawLine	$c130	$c130	$fe33	Draw, clear, or recover line between two endpoints.
DrawPoint	$c133	$c133	$fe21	Draw, clear, or recover a single screen point.
DrawSprite	$c1c6	$c1c6	$fea2	Define sprite image.

Name	C64	C128	Apple	Description
DSdiv	$c16c	$c16c	$fed8	Double-precision signed division.
DShiftLeft	$c15d	$c15d	$fec6	Double-precision left shift (zeros shifted in).
DShiftRight	$c262	$c262	$fec9	Double-precision right shift (zeros shifted in).
EnableProcess	$c109	$c109	$fec3	Make a process runable immediately.
EnablSprite	$c1d2	$c1d2	$fea8	Enable sprite.
EnterDeskTop	$c22c	$c22c	$ff59	Leave application and return to GEOS deskTop.
EnterTurbo	$c214	$c214	n/a	Activate CBM disk turbo on current drive.
EraseCharacter	n/a	n/a	$ff3e	Erase text character from screen.
ExitTurbo	$c232	$c232	n/a	Deactivate CBM disk turbo on current drive.
FastDelFile	$c244	$c244	n/a	Quick file delete (requires full track/sector list).
FdFTypesInDir	n/a	n/a	$0363	Find files in directory other than current directory.
FetchRAM	$c2cb	$c2cb	n/a	Transfer data from CBM RAM-expansion unit.
FillRam	$c17b	$c17b	$feff	Fill memory with a particular byte.
FindBAMBit	$c2ad	$c2ad	n/a	Get allocation status of particular CBM disk block.
FindFile	$c20b	$c20b	$0369	Search for a particular file.
FindFTypes	$c23b	$c23b	$0366	Find all files of a particular GEOS type.
FindVBMBit	n/a	n/a	$036c	Get allocation status of particular ProDOS disk block.
FirstInit	$c271	$c271	$ff11	GEOS startup entry point.
FndFilinDir	n/a	n/a	$036c	Find a file in a directory other than current directory.
FollowChain	$c205	$c205	n/a	Follow chain of CBM sectors, building track/sector table.
FrameRectangle	$c127	$c127	$fe3f	Draw a rectangular frame (outline).
FreeBlock	$c2b9	$c2b9	$032a	Mark a disk block as not-in-use.
FreeDir	n/a	n/a	$03ba	Free all blocks associated with a subdirectory.
FreeFile	$c226	$c226	$035a	Free all blocks assocatiated with a file.
FreezeProcess	$c112	$c112	$feba	Pause a process countdown timer.
Get1stDirEntry	$9030	$9030	$039c	Get first directory entry.
GetBlock	$c1e4	$c1e4	$0300	Read single disk block into memory.
GetCharWidth	$c1c9	$c1c9	$fe87	Calculate width of character without style attributes.
GetDimensions	$790c	$790c	n/a	Get CBM printer page dimensions.
GetDirHead	$c247	$c247	$0372	Read directory header into memory.
GetFHdrInfo	$c229	$c229	$0390	Read a GEOS file header into memory.
GetFile	$c208	$c208	$030f	Load GEOS file.
GetFreeDirBlk	$c1f6	$c1f6	$0375	Find an empty directory slot.
GetLdVars	n/a	n/a	$ff44	Transfer internal Ld variables to GEOS pseudoregisters.
GetMode	n/a	n/a	$6012	Return current printer resolution and setable attributes.
GetNextChar	$c2a7	$c2a7	$fe75	Get next character from character queue.
GetNxtDirEntry	$9033	$9033	$039c	Get directory entry other than first.
GetPathname	n/a	n/a	$0384	Return current path string.
GetPattern	n/a	n/a	$ff7d	Get eight-byte GEOS pattern definition.
GetPtrCurDkNm	$c298	$c298	$ff56	Return pointer to current disk name.
GetRandom	$c187	$c187	$fee4	Calculate new random number.
GetRealSize	$c1b1	$c1b1	$fe84	Calculate actual character size with attributes.
GetScanLine	$c13c	$c13c	$fe66	Calculate scanline address.
GetScreenLine	n/a	n/a	$ff62	Copy Apple screen data to buffer.
GetSerialNumber	$c196	$c196	$4b38	Return GEOS serial number or pointer to name string.
GetSpriteData	n/a	n/a	$ff6e	Get sprite image data.
GetString	$c1ba	$c1ba	$fe72	Get string input from user.
GetVBM	n/a	n/a	$031e	Read first VBM block into VBM cache.
GoDirectory	n/a	n/a	$039f	Go directory to a specific subdirectory.
GotoFirstMenu	$c1bd	$c1bd	$fe18	Retract all sub-menus and reactivate at main level.
GraphicsString	$c136	$c136	$fe60	Execute a string of graphics commands.
HideOnlyMouse	n/a	$c2f2	$ff7a	Temporarily remove soft-sprite mouse pointer.
HorizontalLine	$c118	$c118	$fe27	Draw a patterned horizontal line.
i_BitmapUp	$c1ab	$c1ab	$fe57	Inline BitmapUp.
i_FillRam	$c1b4	$c1b4	$ff02	Inline FillRam.
i_FrameRectangle	$c1a2	$c1a2	$fe42	Inline FrameRectangle.
i_GraphicsString	$c1a8	$c1a8	$fe63	Inline GraphicsString.
i_ImprintRectangle	$c253	$c253	$fe51	Inline ImprintRectangle.

Name	C64	C128	Apple	Description
i_MoveData	$c1b7	$c1b7	$fef9	Inline MoveData.
i_NewBitUp	n/a	n/a	$ff9e	Inline NewBitUp.
i_PutString	$c1ae	$c1ae	$fe6c	Inline PutString.
i_RecoverRectangle	$c1a5	$c1a5	$fe4b	Inline RecoverRectangle.
i_Rectangle	$c19f	$c19f	$fe3c	Inline Rectangle.
ImprintLine	n/a	n/a	$ff8f	Imprint horizontal line to background buffer.
ImprintRectangle	$c250	$c250	$fe4e	Imprint rectangular area to background buffer.
InfoCard	n/a	n/a	$670f	Get I/O card attributes.
InitCard	n/a	n/a	$6700	Intialize I/O card.
InitForDialog	n/a	n/a	$ff4a	Internal pre-dialog box intialization.
InitForIO	$c25c	$c25c	n/a	Prepare CBM system for I/O across serial bus.
InitForPrint	$7900	$7900	$6000	Initialize printer (once per document).
InitMouse	$fe80	$fe80	$f000	Initialize input device.
InitProcesses	$c103	$c103	$feae	Intialize processes.
InitRam	$c181	$c181	$ff05	Initialize memory areas from table.
InitSprite	n/a	n/a	$ff32	Initialize software sprites.
InitTextPrompt	$c1c0	$c1c0	$fe78	Initialize text prompt.
InputByte	n/a	n/a	$6709	Read byte from I/O card.
InsertRecord	$c286	$c286	$0339	Insert new VLIR record in front of current record.
InterruptMain	$c100	$c100	$fe03	Main interrupt level processing.
InvertLine	$c11b	$c11b	$fe2d	Invert the pixels on a horizontal screen line.
InvertRectangle	$c12a	$c12a	$fe45	Invert the pixels in a rectangular screen area.
IrqMiddle	n/a	n/a	$ff20	Software interrupts entry point.
IsMseInRegion	$c2b3	$c2b3	$fe9c	Check if mouse is within a screen region.
JsrToAux	n/a	n/a	$03b4	Call aux-memory subroutine.
KeyFilter	n/a	n/a	$f009	Input driver keypress filter.
LdApplic	$c21d	$c21d	$038d	Load GEOS application.
LdDeskAcc	$c217	$c217	$035d	Load GEOS desk accessory.
LdFile	$c211	$c211	$0312	Load GEOS data file.
LoadAuxSet	n/a	n/a	$ff77	Use aux-memory character set.
LoadCharSet	$c1cc	$c1cc	$fe8a	Use main-memory character set.
MainLoop	$c1c3	$c1c3	$fe00	GEOS MainLoop processing.
MakeSubDir	n/a	n/a	$0387	Create ProDOS subdirectory.
MouseOff	$c18d	$c18d	$fe96	Disable mouse pointer and GEOS mouse tracking.
MouseUp	$c18a	$c18a	$fe99	Enable mouse pointer and GEOS mouse tracking.
MoveAuxData	n/a	n/a	$ff6b	Apple aux-memory memory move routine.
MoveBData	n/a	$c2e3	n/a	128 backram memory move routine.
MoveData	$c17e	$c17e	$fef6	Memory move routine.
NewBitClip	n/a	n/a	$ff95	Bit-boundary BitmapClip.
NewBitOtherClip	n/a	n/a	$ff0e	Bit-boundary BitOtherClip.
NewBitUp	n/a	n/a	$ff1d	Bit-boundary BitmapUp.
NewDisk	$c1e1	$c1e1	n/a	Tell CBM disk turbo software that a new disk is in drive.
NextRecord	$c27a	$c27a	$0336	Make next VLIR the current record.
NormalizeX	n/a	$c2e0	n/a	Normalize C128 X-coordinates for 40/80 modes.
NxtBlkAlloc	$c24d	$c24d	$0354	Version of BlkAlloc that starts at a specific block.
OpenCard	n/a	n/a	$6703	Open I/O card for access.
OpenDisk	$c2a1	$c2a1	$0399	Open disk in current drive.
OpenRecordFile	$c274	$c274	$033f	Open VLIR file on current disk.
OutputByte	n/a	n/a	$6706	Write byte to I/O card.
Panic	$c2c2	$c2c2	$fef3	System-error dialog box.
PointRecord	$c280	$c280	$0342	Make specific VLIR record the current record.
PosSprite	$c1cf	$c1cf	$fea5	Position sprite.
PreviousRecord	$c27d	$c27d	$0345	Make previous VLIR record the current record.
PrintASCII	$790f	$790f	$600c	Send ASCII data to printer.
PrintBuffer	$7906	$7906	$6009	Send graphics data to printer.
PromptOff	$c29e	$c29e	$fe7e	Turn off text prompt.
PromptOn	$c29b	$c29b	$fe7b	Turn on text prompt.
PurgeTurbo	$c235	$c235	n/a	Remove CBM disk turbo software from drive.

Name	C64	C128	Apple	Description
PutBlock	$c1e7	$c1e7	$0303	Write singel disk block from memory.
PutChar	$c145	$c145	$fe81	Display a single character to screen.
PutDecimal	$c184	$c184	$fe6f	Format and display an unsigned double-precison number.
PutDirHead	$c24a	$c24a	$037b	Write directory header to disk.
PutScreenLine	n/a	n/a	$ff65	Copy buffer data to Apple screen memory.
PutString	$c148	$c148	$fe69	Print string of characters to screen.
PutVBM	n/a	n/a	$0321	Flush VBM cache; write currently cached block to disk.
ReadBackLine	n/a	n/a	$ff74	Get data from background screen in linear bitmap format.
ReadBlock	$c21a	$c21a	n/a	CBM get disk block primitive.
ReadByte	$c2b6	$c2b6	$ff41	Read disk file a byte at a time.
ReadClock	n/a	n/a	$0800	Update GEOS time/date variables from clock hardware.
ReadFile	$c1ff	$c1ff	$0315	Read chained list of blocks into memory.
ReadLink	$904b	$904b	n/a	Read CBM track/sector link.
ReadRecord	$c28c	$c28c	$0348	Read current VLIR record into memory.
ReadScanLine	n/a	n/a	$ff98	Get data from foreground screen in linear bitmap format.
RecoverAllMenus	$c157	$c157	$fe1e	Recover all menus from background buffer.
RecoverFG	n/a	n/a	$03ae	Restores foreground screen from data created with SaveFG.
RecoverLine	$c11e	$c11e	$fe30	Recover horizontal screen line from background buffer.
RecoverMenu	$c154	$c154	$fe1b	Recover single menu from background buffer.
RecoverRectangle	$c12d	$c12d	$fe48	Recover rectangular screen area from background buffer.
RecoverSysRam	n/a	n/a	$ff53	Restore system after dialog box or desk accessory.
Rectangle	$c124	$c124	$fe39	Draw a filled rectangle.
ReDoMenu	$c193	$c193	$fe12	Reactivate menus at the current level.
RenameFile	$c259	$c259	$0393	Rename GEOS disk file.
ReOpenDisk	n/a	n/a	$03a2	Reopen disk to previous directory.
ResetAlarm	n/a	n/a	$080c	Disable clock driver alarm.
ResetHandle	$c003	$c003	n/a	Internal bootstrap entry point.
RestartProcess	$c106	$c106	$feb1	Unblock, unfreeze, and restart process.
RestoreFontData	n/a	n/a	$ff38	Restore font variables from saveFontTab.
RstrAppl	$c23e	$c23e	$0360	Leave desk accessory and return to calling application.
RstrFrmDialog	$c2bf	$c2bf	$ff1a	Exit dialog box.
SaveFG	n/a	n/a	$03ab	Save foreground screen data for RestoreFG.
SaveFile	$c1ed	$c1ed	$0318	Save/create a GEOS file.
SaveFontData	n/a	n/a	$ff35	Copy font variables to saveFontTab.
SetAlarm	n/a	n/a	$0809	Set clock driver alarm.
SetDevice	$c2b0	$c2b0	n/a	Establish communication with a new serial device.
SetGDirEntry	$c1f0	$c1f0	$037b	Create and save a new GEOS directory entry.
SetGEOSDisk	$c1ea	$c1ea	n/a	Convert normal CBM disk into GEOS format disk.
SetLdVars	n/a	n/a	$ff47	Set internal Ld variables from GEOS pseudoregisters.
SetMode	n/a	n/a	$6015	Set print mode.
SetMouse	n/a	$fe89	n/a	Reset input device scanning circuitry.
SetMsePic	n/a	$c2da	$ff2f	Set and preshift new soft-sprite mouse picture.
SetNewMode	n/a	$c2dd	n/a	Change GEOS 128 graphics mode (40/80 switch).
SetNextFree	$c292	$c292	$032d	Search for nearby free disk block and allocate it.
SetNLQ	$7915	$7915	n/a	Begin near-letter quality printing.
SetPattern	$c139	$c139	$fe36	Set current fill pattern.
SetTimeDate	n/a	n/a	$0806	Set clock driver time and date.
SetUserPattern	n/a	n/a	$ff80	Define new GEOS pattern image.
Sleep	$c199	$c199	$fec0	Put current subroutine to sleep for a specified time.
SlowMouse	$fe83	$fe83	$f003	Reset mouse velocity variables.
SmallPutChar	$c202	$c202	$fe90	Fast character print routine.
SoftSprHandler	n/a	n/a	$ff2c	Apple soft-sprite drawing routine.
SortAlpha	n/a	n/a	$03a5	Insertion sort.
StartAppl	$c22f	$c22f	$fe06	Warmstart GEOS and start application in memory.
StartASCII	$7912	$7912	$6006	Begin ASCII mode printing.
StartMouseMode	$c14e	$c14e	$fe93	Start monitoring input device.
StartPrint	$7903	$7903	$6003	Begin graphics mode printing.
StashRAM	$c2c8	$c2c8	n/a	Transfer memory to CBM RAM-expansion unit.

Name	C64	C128	Apple	Description
StatusCard	n/a	n/a	$6712	Get current status of I/O card.
StopPrint	$7909	$7909	$600f	End page of printer output.
SwapBData	n/a	$c2e6	n/a	C128 memory swap between front/back ram.
SwapMainAndAux	n/a	n/a	$ff71	Apple memory swap between main and aux.
SwapRAM	$c2ce	$c2ce	n/a	CBM RAM-expansion unit memory swap.
TempHideMouse	n/a	$c2d7	$ff29	Hide soft-sprites before direct screen access.
TestPoint	$c13f	$c13f	$fe24	Test status of single screen point (on or off?).
ToBasic	$c241	$c241	n/a	Call Commodore BASIC.
UnblockProcess	$c10f	$c10f	$feb7	Unblock a blocked process, allowing it to run again.
UnfreezeProcess	$c115	$c115	$febd	Unpause a frozen process timer.
UpdateMouse	$fe86	$fe86	$f006	Update mouse variables from input device.
UpdateParent	n/a	n/a	$03b7	Update parent directory to reflect any changes.
UpdateRecordFile	$c295	$c295	$034b	Update currently open VLIR file without closing it.
UpDirectory	n/a	n/a	$038a	Close current directory and move up one level.
UseSystemFont	$c14b	$c14b	$fe8d	Use default system font (BSW 9).
VerifyBData	n/a	$c2e9	n/a	C128 backram verfiy.
VerifyRAM	$c2d1	$c2d1	n/a	CBM RAM-expansion unit verify.
VerticalLine	$c121	$c121	$fe2a	Draw a patterned vertical line.
VerWriteBlock	$c223	$c223	n/a	CBM disk block verify primitive.
WarmStart	n/a	n/a	$ff4d	Bring GEOS to a warmstart state.
WriteBlock	$c220	$c220	n/a	CBM write disk block primitive.
WriteFile	$c1f9	$c1f9	$031b	Write chained list of blocks to disk.
WriteRecord	$c28f	$c28f	$034e	Write current VLIR record to disk.

NOTES: The following routines have had there names changed to avoid confusion and/or make them unique in the first eight characters:

Current Name	Formerly	Current Name	Formerly
ClockInt	ReadClockInt	NewBitClip	NewBitmapClip
CopyLine	CopyScreenLine	NewBitUp	NewBitmapUp
i_NewBitUp	i_NewBitmapUp	PutChar	Putchar
LoadAuxSet	LoadAuxCharSet	RstrFrmDialog	RstrFrmDialogue
MoveAuxData	MoveBData	SetMsePic	SetMousePicture

Apple: The following routines were not added to the jump table until version 2.0.3 (version 2.0, release 3). To call these routines in v2.0.2 (the initial public release) requires a patch to the jump tables.

NewBitOtherClip, i_NewBitUp, AllocateBlock, Get1stDirEntry, GetNxtDirEntry

CBM The following routines do not have jump table entries in the 1541 device driver. The device type must be checked before calling any of these routines. See the actual routine reference for more information and 1541 alternatives.

AllocateBlock, ReadLink

The following routines do not exist in version 1.2 and below of CBM GEOS:

Get1stDirEntry, GetNxtDirEntry, FreeBlock .

They require calling directly into the GEOS Kernal. See the actual routine reference for more information.

APPLE GEOS V2.0.2 KERNAL PATCH (revision 3)

Matt Loveless
Robert Boudidge
April 6, 1988

Revision History
================

03/22/88	ml/rb	First hacked up
03/23/88	ml/rb	Fixed aux/main snafu
04/06/88	ml/rb	Added i_NewBitmapUp for 'ol vargas.
04/08/88	mav	Used patch and made misc bug fixes. Called routine i_NewBitmapUp and it seemed to work fine.
4/18/88	mav	Changed references to GetNextDirEntry to GetNxtDirEntry.

Note: This patch information has not yet been thoroughly tested. If anybody has problems, please direct them to Robert and/or Matt.

The following routines were inadvertently left out of the Apple GEOS jump table in the first production release (v2.0, rel. 2):

NewBitOtherClip
AllocateBlock
GetFstDirEntry
GetNxtDirEntry
i_NewBitmapUp

Applications that wish to use these routines must patch the Kernal jump tables in order to run under v2.0, rel. 2. Starting with v2.0, release 3, these routines are guaranteed to be in the jump table. Applications MUST check the version number before patching in order to be compatible with future versions of Apple GEOS.

The following unused jump table slots will be used:

NewBitOtherClip	$ff0e	(old ToBasic)
AllocateBlock	$030c	(old ReadChain)
GetFstDirEntry	$0396	(old ChkDkGEOS)
GetNxtDirEntry	$039c	(old SetGEOSDisk)
i_NewBitmapUp	$ff9e	(old ageos RamDD)

These equates will not change and will ensure an application is upward compatible with future versions of Apple GEOS.

Patch info for v2.0, release 2:

Routine	Address	In aux	In main	
NewBitOtherClip	$ff0e	jmp $df9a	jsr $fc16	(aux = altzp)
AllocateBlock	$030c	jmp $45c1	jsr $0900	(aux = ramwrt)
GetFstDirEntry	$0396	jmp $4e66	jsr $0900	(aux = ramwrt)
GetNxtDirEntry	$039c	jmp $4e78	jsr $0900	(aux = ramwrt)
i_NetBitmapUp	$ff9e	jmp Panic	jsr $ecb3	(aux = ramwrt)

(Note that i_NewBitmapUp cannot be run while the alternate zero page (ALTZP_ON) is switched in. The jump table entry jumps to the Panic routine. This routine must be called when the main zero page is in.)

And in code:

```
;*******************************************************************************************************************
;Patch          Patch GEOS v2.0.2
;
;*******************************************************************************************************************

;*** EQUATES ***

;Locations in jump table
NewBitOtherClip         = $ff0e         ;pixel boundary version of BitOtherClip
AllocateBlock           = $030c         ;allocate block in VBM (opposite of FreeBlock)
GetFstDirEntry          = $0396         ;get 1st directory entry in current directory
GetNxtDirEntry          = $039c         ;get next directory entry
i_NewBitmapUp           = $ff9e         ;Do an inline NewBitmapUp.  Mike likes this call.

;Direct entry points into v2.0.2 kernal
j_NewBitOtherClip       = $df9a
o_AllocateBlock         = $45c1
o_GetFstDirEntry        = $4e66
o_GetNxtDirEntry        = $4e78
iT_NewBitmapUp          = $ecb3

SwitchAllRegs           = $fc16         ;for aux-high bank graphics switch
```

```
LowSwitch               = $0900         ;for aux bank (low jump table)

GOOD_VERSION            = $20           ;version 2.0.3 does not need to be patched
GOOD_RELEASE            = $03

.macro  PatchJmpTable   dest, source    ;copies three bytes from source to dest
        lda     source
        sta     dest
        lda     source+1
        sta     dest+1
        lda     source+2
        sta     dest+2
.endm

;Note:  Can be munged to use less bytes by table driving the patch. Done this way here for
;       demonstration purposes only.

Patch:
;       Check version to ensure that we don't patch Kernals that are already
;       fixed.  We check for version 2.0.3 (good version) explicitly because
;       2.0.2 is the first release of the Apple Kernal and 2.0.3 has the fixes
;       in it.

        lda             version
        cmp             #GOOD_VERSION
        bne             10$             ;branch if current version >2.0.  If
                                        ; not then it is 3.xx which is fixed.
        lda             release
        cmp             #GOOD_RELEASE
        bge             10$             ;branch if this release >= good release

        bra             20$             ;bra to do the patch

10$:
        rts

;This is a version that needs to be patched. Go patch it.
;first patch addresses in Main
20$:
        PatchJmpTbl     NewBitOtherClip, M_NewBitPatch
        PatchJmpTbl     AllocateBlock, M_AllocBlkPatch
        PatchJmpTbl     Get1stDirEntry, M_Get1stPatch
        PatchJmpTbl     GetNxtDirEntry; M_GetNxtPatch
        PatchJmpTbl     i_NewBitmapUp, M_iNewBitmapUp

;patch graphics in high auxiliary
        sta             ALTZP_ON                                ; switch in high aux
        PatchJmpTbl     NewBitOtherClip, A_NewBitPatch
        PatchJmpTbl     i_NewBitmapUp, A_iNewBitmapUp
        sta             ALTZP_OFF                               ; switch out high aux

;patch others in normal aux
        sta             RAMWRT_ON                               ; send writes to aux bank
        PatchJmpTbl     AllocateBlock, A_AllocBlkPatch          ; read from main, write to aux
        PatchJmpTbl     Get1stDirEntry, A _Get1stPatch          ; read from main, write to aux
        PatchJmpTbl     GetNxtDirEntry, A_GetNxtPatch           ; read from main, write to aux
        sta             RAMWRT_OFF                              ; send writes back to main

99$:
        rts

;Main memory patches
M_NewBitPatch:
        jsr     SwitchAllRegs

M_AllocBlkPatch:
M_Get1stPatch:
M_GetNxtPatch:
        jmp     LowSwitch
M_iNewBitmapUp:
        jmp     ii_NewBitmapUp

;Aux memory patches
A_NewBitPatch:
        jmp     j_NewBitOtherClip
A_AllocBlkPatch:
        jmp     o_AllocateBlock
A_Get1stPatch:
```

```
        jmp     o_GetIstDirEntry
A_GetNxtPatch:
        jmp     o_GetNxtDirEntry
A_iNewBitmapUp:
        jmp     Panic                   ;can't call this routine from aux memory.
```

This page intentionally left blank to maintain right/left (verso/recto) page ordering. Final version will correct this.

GEOS Text Escape Character Codes

Code	Constant	Description
0	NULL¥	String termination character.
1	†	*unused*
2	†	*unused*
3	†	*unused*
4	†	*unused*
5	†	*unused*
6	†	*unused*
7	†	*unused*
8	BACKSPACE	Erase the previous character (Apple GEOS uses the width in **lastWidth**).
9	FORWARDSPACE	*Not implemented in GEOS 64 or GEOS 128.* Move current printing position rightward the width of a space character. This escape code is used by geoWrite to represent a tab (use **TAB** constant).
10	LF	Linefeed: move current printing position down one line (value in **currentHeight**).
11	HOME	Move current printing position to upper-left screen corner.
12	UPLINE	Move current printing position up one line (value in **currentHeight**). This escape code is used by geoWrite to represent a page-break (use **PAGE BREAK** constant).
13	CR	Carriage return: move current printing position down one line and over to the left margin (value in **leftMargin**).
14	ULINEON	Begin underlining.
15	ULINEOFF	End underlining.
16	ESC_GRAPHICS ¥	Escape code for graphics string: remainder of this string is treated as input to the **GraphicsString** routine.
17		*unimplemented* This escape code is ignored by GEOS text routines. This escape code is used by geoWrite to represent a ruler escape (use **ESC RULER** constant).
18	REVON	Begin reverse video printing (white on black).
19	REVOFF	End reverse video printing.
20	GOTOX ¥	Change the x-coordinate of the current printing position to the word value stored in the following two bytes.
21	GOTOY ¥	Change the y-coordinate of the current printing position to the byte value in the following byte.
22	GOTOXY ¥	Change the x-coordinate of the current printing position to the word value stored in the following two bytes and change the y-coordinate to the value in the third byte.
23	NEWCARDSET ¥	*unimplemented.* This does nothing but skip over the following two bytes.
24	BOLDON	Begin boldface printing.
25	ITALICON	Begin italicized printing.
26	OUTLINEON	Begin outlined printing.
27	PLAINTEXT	Begin plain text printing (turns off all type style attributes).
28	†	*unused*
29	†	*unused*
30	†	*unused*
31	†	*unused*

†Should never be sent to a GEOS text routine unless the application is running under a future version of GEOS that explicitly supports this character code.

¥For use with **PutString**; not directly supported by **PutChar**.

GEOS ASCII Character Codes

Code	Character
32	space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@

Code	Character
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`

Code	Character
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	deletion character: **USELAST**
128	short-cut key: **SHORTCUT**

```
*****************************************************************************
**** UPDATE:    PREVIOUS PATCH INFORMATION HAD BANK INFORMATION SWITCHED ****
****            USE THIS VERSION TO PATCH KERNAL AND DISREGARD EARLIER   ****
*****************************************************************************
```

APPLE GEOS V2.0.2 KERNAL PATCH (revision 2)

Matt Loveless
March 23, 1988

Note: This patch information has not yet been thoroughly tested. If anybody has problems, please direct them to Robert and/or Matt.

The following routines were inadvertently left out of the Apple GEOS jump table in the first production release (v2.0, rel. 2):

NewBitOtherClip
AllocateBlock
GetFirstDirEntry
GetNextDirEntry

Applications that wish to use these routines must patch the Kernal jump tables in order to run under v2.0, rel. 2. Starting with v2.0, release 3, these routines are guaranteed to be in the jump table. Applications MUST check the version number before patching in order to be compatible with future versions of Apple GEOS.

The following unused jump table slots will be used:

NewBitOtherClip	$ff0e	(old ToBasic)
AllocateBlock	$030c	(old ReadChain)
GetFirstDirEntry	$0396	(old ChkDkGEOS)
GetNextDirEntry	$039c	(old SetGEOSDisk)

These equates will not change and will ensure an application is upward compatible with future versions of Apple GEOS.

Patch info for v2.0, release 2:

Routine	Address	In aux	In main	
NewBitOtherClip	$ff0e	jmp $df9a	jsr $fc16	(aux = altzp)
AllocateBlock	$030c	jmp $45c1	jsr $0900	(aux = ramwrt)
GetFirstDirEntry	$0396	jmp $4e66	jsr $0900	(aux = ramwrt)
GetNextDirEntry	$039c	jmp $4e78	jsr $0900	(aux = ramwrt)

And in code:

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

NewBitOtherClip         = $ff0e         ;pixel boundary version of BitOtherClip
AllocateBlock           = $030c         ;allocate block in VBM (opposite of FreeBlock)
GetFirstDirEntry        = $0396         ;get 1st directory entry in current directory
GetNextDirEntry         = $039c         ;get next directory entry

;Direct entry points into v2.0.2 kernal
j_NewBitOtherClip       = $df9a
o_AllocateBlock         = $45c1
o_GetFirstDirEntry      = $4e66
o_GetNextDirEntry       = $4e78

SwitchAllRegs           = $fc16         ;for aux-high bank graphics switch
LowSwitch               = $0900         ;for aux bank (low jump table)

GOOD_VERSION            = $20           ;version 2.0.3 does not need to be patched
GOOD_RELEASE            = $03

.macro  PatchJmpTable   dest, source    ;copies three bytes from source to dest
        lda     source
        sta     dest
        lda     source+1
        sta     dest+1
        lda     source+2
        sta     dest+2
.endm

;Note:  Can be munged to use less bytes by table driving the patch. Done this way here for
;       demonstration purposes only.

Patch:
;Check version to ensure that we don't patch Kernals that are already fixed
        lda             version
```

```
        cmp             #GOOD_VERSION
        bge             99$             ;branch if current version >= good version
        lda             release
        cmp             #GOOD_RELEASE
        bge             99$             ;branch if current release >= good release

;This is a version that needs to be patched. Go patch it.
;first patch addresses in Main
        PatchJmpTbl     NewBitOtherClip, M_NewBitPatch
        PatchJmpTbl     AllocateBlock, M_AllocBlkPatch
        PatchJmpTbl     GetFirstDirEntry, M_GetFirstPatch
        PatchJmpTbl     GetNextDirEntry, M_GetNextPatch

;patch graphics in high auxiliary
        sta             ALTZP_ON                        ; switch in high aux
        PatchJmpTbl     NewBitOtherClip, A_NewBitPatch
        sta             ALTZP_OFF                       ; switch out high aux

;patch others in normal aux
        sta             RAMWRT_ON                       ; send writes to aux bank
        PatchJmpTbl     AllocateBlock, A_AllocBlkPatch  ; read from main, write to aux
        PatchJmpTbl     GetFirstDirEntry, A_GetFirstPatch ; read from main, write to aux
        PatchJmpTbl     GetNextDirEntry, A_GetNextPatch ; read from main, write to aux
        sta             RAMWRT_OFF                      ; send writes back to main

99$:
        rts

;Main memory patches
M_NewBitPatch:
        jsr     SwitchAllRegs

M_AllocBlkPatch:
M_GetFirstPatch:
M_GetNextPatch:
        jmp     LowSwitch

;Aux memory patches
A_NewBitPatch:
        jmp     j_NewBitOtherClip
A_AllocBlkPatch:
        jmp     o_AllocateBlock
A_GetFirstPatch:
        jmp     o_GetFirstDirEntry
A_GetNextPatch:
        jmp     o_GetNextDirEntry
```

Summary of desired disk read/write operations and what routines to call
---

Read a block from disk into MAIN memory, given block number
ReadBlock or GetBlock with RWbank set to #MAIN (default)

Read a block from disk into AUX memory, given block number
ReadBlock or GetBlock with RWbank set to #AUX (set, call, reset)

Write a block from MAIN memory to disk, given block number
WriteBlock or PutBlock with RWbank set to #MAIN (default)

Write a block from AUX memory to disk, given block number
WriteBlock or PutBlock with RWbank set to #AUX (set, call, reset)

Read (and execute, if possible) a SEQUENTIAL file into MAIN memory, given its filename
GetFile with RWbank set to #MAIN (default)

Read a SEQUENTIAL data file into AUX memory, given its filename (filename must be in MAIN) (note: datafiles only; cannot read Desk Accessories or Applications into AUX and have them execute)
GetFile with RWbank set to #AUX (set, call, reset)

Read a SEQUENTIAL file into MAIN memory, given its directory entry
LdFile with RWbank set to #MAIN (default)

Read a SEQUENTIAL file into AUX memory, given its directory entry (directory entry must be in MAIN)
LdFile with RWbank set to #AUX (set, call, reset)

Open a VLIR file and read (and execute, if possible) it's first record into MAIN memory, given its filename
GetFile with RWbank set to #MAIN (default)

Open a VLIR data file and read it's first record into AUX memory, given its filename (filename must be in MAIN) (note: cannot run DA's or Applic's in AUX)
GetFile with RWbank set to #AUX (set, call, reset)

Read current VLIR record into MAIN memory
ReadRecord with RWbank set to #MAIN (default)

Read current VLIR record into AUX memory, given record number
ReadRecord with RWbank set to #AUX (set, call, reset)

Read a single VLIR record into MAIN memory, given record number
PointRecord then ReadRecord with RWbank set to #MAIN (default)

Read a single VLIR record into AUX memory, given record number
PointRecord then ReadRecord with RWbank set to #AUX (set, call, reset)

Read a single VLIR record into MAIN memory, given block number of record's index block (must handle VLIR variables yourself)
ReadFile with RWbank set to #MAIN (default)

Read a single VLIR record into AUX memory, given block number of record's index block (must handle VLIR variables yourself)
ReadFile with RWbank set to #AUX (default)

Write an indexed "chain" from MAIN memory, given a pre-allocated index table (in internal indexBlock buffer)
WriteFile with RWbank set to #MAIN (default)

Write an indexed "chain" from AUX memory, given a pre-allocated index table (in internal indexBlock buffer)
WriteFile with RWbank set to #AUX (set, call, reset)

Write a file from MAIN memory, given the header block for it
SaveFile with RWbank set to #MAIN (default)

Write a file from AUX memory, given the header block for it
SaveFile with RWbank set to #AUX (set, call, reset)

Write current VLIR record from MAIN memory
WriteRecord with RWbank set to #MAIN (default)

Write current VLIR record from AUX memory, given record number
WriteRecord with RWbank set to #AUX (set, call, reset)

Write a single VLIR record from MAIN memory, given record number
PointRecord then WriteRecord with RWbank set to #MAIN (default)

Write a single VLIR record from AUX memory, given record number
PointRecord then WriteRecord with RWbank set to #AUX (set, call, reset)

Specifications proposal for implementation of /SYSTEM directory 10/27/87

All disks from which the AppleGEOS DeskTop runs an application will either already have a /SYSTEM directory or have one created. The /SYSTEM directory (a subdirectory named SYSTEM of the root directory) is a special GEOS subdirectory where system data files (files of type SYSTEM, scraps, and fonts), desk accessories, and applications can reside. In addition to the /SYSTEM directory, these kinds of files can also reside anywhere else on any other directory on the disk.

Kernal:

The kernal support planned for the /SYSTEM directory is in the routine OpenDisk. In addition to its normal function, OpenDisk will also find the /SYSTEM direcotry and store the block number of it's key block in the global variable sysDirBlkno. If there is no /SYSTEM directory found, a 0 will be returned in this word.

Another place where kernal support might be needed is with Desk Accessories. When a desk accessory is run, it will be assumed that the current directory will be the directory with the desk accessory (i.e. that the Desktop and applications will do a GoDirectory before LdDeskAcc or GetFile). If no change are made to the desk accessory routines, the DA's Swap File will be saved to the current directory. If the desk accessory were to change the current directory, then when RstrAppl is called, RstrAppl will not know where to find the Swap File. The solution to this is to always save and load the Swap File to/from a fixed location, either the /SYSTEM directory or the root directory. This will require that both LdDeskAcc and RstrAppl make calls to GoDirectory to make to the root directory or the /SYSTEM the current directory. Unfortunately, this will increase overall desk accessory access time.

Applications:

Application handling of the /SYSTEM directory is flexible. It is up to each application to determine how much support for the /SYSTEM directory is necessary. For example, if scraps are supported, then they should be saved and loaded from the /SYSTEM directory. This is not mandatory, as the saving and directory manipulation must be done by the application, but this allows other applications to find them. To actually save something to the /SYSTEM directory will be very simple. Merely use the following code:

```
        :
        :
PushW   curKBlkno               ;save current directory
MoveW   sysDirBlkno,r0          ;goto to /SYSTEM directory
jsr     GoDirectory
txa                             ;check for errors!
bne     ErrorHandle
        :
(set up registers for SaveFile)
        :
jsr     SaveFile                ;save file to /SYSTEM directory
txa                             ;check for errors!
bne     ErrorHandle
PopW    r0                      ;pop saved curKBlkno into r0
jsr     GoDirectory             ;return to original directory
txa                             ;check for errors!
bne     ErrorHandle
        :
        :
```

For fonts and desk accessories, an application also has a choice of using any directory or group of directories it wants. However, to be consistent with other applications it is suggested that only the current directory and the /SYSTEM directory be used. Then, when a font or desk accessory is to be loaded, the above code could be used by replacing SaveFile with GetFile. GetFile will automatically determine whether to use LdDeskAcc or LdFile.

When an application wants to gather a list of desk accessories (for the GEOS menu) or fonts available, it will have to do a FindFTypes with both the current directory and the /SYSTEM directory. Since the /SYSTEM directory is known from the global variable sysDirBlkno, this is an ideal situation to use the new Kernal routine FdFTypesInDir which searches in a given directory. The application will have to keep track of which of the files it finds are from the /SYSTEM directory and which are from the current directory so it can switch to the correct directory when it wants to load an arbitrary file from the list. The sample application will handle this for the desk accessories in the GEOS menu. This will probably be all that most applications will need.

An application's data files can, of course, be loaded and saved to any directory that the application desires, but again, for consistency, only the current directory should be used. When opening a data file at the start of an application, the standard NEWDBGETFILES dialog box routine can be used. The desktop will maintain this interface by going to the directory of a data file that is double-clicked on when the parent application is launched.

Desktop:

The Desktop will deal with the /SYSTEM directory to a greater extent than either the Kernal or applications.

In addition to having to manage the list of Desk Accessories in the GEOS menu and setting of the current directory for double-clicked data files mentioned above, the desktop will also have to create the /SYSTEM directory if none exists, manage a list of applications for a new menu item, handle dialog boxes with application and desk accessory selection boxes for new view-by-text modes, and getting printer drivers, input drivers, preference, and configuration files from the /SYSTEM directory.

Interleaving issues

The main issue with reading and writing files is whether we will be able to "catch the interleave". A ProDOS disk's physical sectors are contiguous, but a ProDOS block, which is made up of two sectors has both inter-block interleaving and intra-block interleaving. Inter-block interleaving seperates the two sectors of each block by a single sector. Intra-block interleaving seperates the two sectors of a block (in addition to the inter-block interleaving sector in between them) by a single sector.

The disk spins at 300 rpm and the Apple 6502 runs a 1 Mhz so there are 200,000 cycles per revolution. At 16 sectors per track, this works out to about 12,500 cycles per sector. This means that we have about 12,500 cycles to process both between fetching the two sectors of a block (while we're in the device driver) and between fetching contiguous blocks (while in the read and write file routines).

The device driver is such that there will be very little time to process data if the interleave is to be taken advantage of. This means there must be very little overhead between calls to ReadBlock (or one of the related routines) during ReadFile or between calls to WriteBlock during WriteFile.

The obvious solution is read or write blocks directly to the destination address or directly from the source address. The problem with this is that if complete blocks are read into a destination buffer, bytes after the end of the buffer will be trashed if the buffer is not a multiple of the block and the size of the valid data in the file being read is the same size as the buffer.

The method used by appleGEOS ReadFile eliminates this problem and still can read blocks directly into their destination. Since a complete list of the blocks in the file is available beforehand, all blocks but the last can be read directly into the destination while the final block can be read into some intermediate buffer for partial copying to the destination. This is done by scanning ahead in the list of file blocks. This allows us to catch the interleave if the blocks of a file are contiguous. For the last block, there will be no more blocks to read, so catching the interleave is not important. This works with reading into both the MAIN and AUX banks of memory.

For WriteFile, such a scheme is not possible because the disk routines can only write to disk from AUX memory. Therefore, every 512-byte block of MAIN memory that is to be written to disk must first be copied to HIGH memory or AUX memory. Writing from AUX memory can be done without missing the interleave because no copying is necessary. But writing from MAIN memory requires copying of 512 bytes, which along with the fact that the device driver has to do work before writing a block eliminates the possibility of catching the interleave.

Memory features of file reading and writing routines

With the 128K Apple comes the picture of that 128K memory as two individual 64K banks. This limits the use of the memory because switching between the two banks in an organized manner is non-trivial.

To alleviate this problem and still provide the flexiblilty that more memory allows, the appleGEOS disk routines allowing reading and writing files from both the MAIN and AUXILIARY banks. A flag, RWbank, controls which bank files will be read/written to/from. Generally, this flag will be set to MAIN for normal applications work.

The GEOS/Commodore Block Allocation Map (BAM) is replaced with the ProDOS Volume Bit Map (VBM) in the Apple version of GEOS. The VBM has the same function as the BAM, showing which blocks on a disk are allocated and which are free to be allocated. There are, however, several changes in the allocation map format and in the routines that deal with it.

The VBM is configured as one or more blocks on the disk it maps. Each byte of each block represents the status of 8 blocks on the disk. Within a byte, the bits are mapped MSB->LSB to blocks X->X+7 where X is some multiple of 8. So, for example, the first byte of the VBM represents disk blocks 0 to 7. Each additional byte represents the next 8 blocks on the disk.

The VBM is allotted at least one block on the disk. So even if the VBM requires only 35 bytes (as it does on a 143K 5.25" floppy), it use and entire 512-byte block. Since blocks contain 512 bytes, this allows a single VBM block to represent (8*512=) 4096 disk blocks. This is the case for 143K 5.25" floppies and 800K 3.5" microfloppies. Hard disks, with upwards of 5 megabytes, will require more than on VBM block.

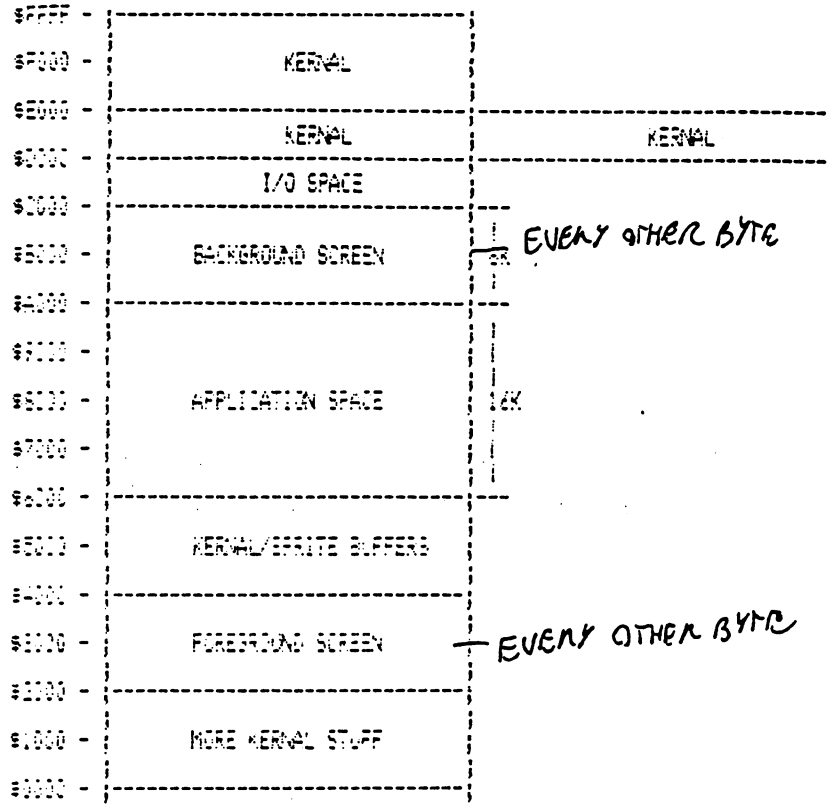
In a normal ProDOS formatted disk, the VBM usually begins at block 6. For multiple block VBMs, the other blocks occupy blocks sequentially after block 6. These blocks are allocated at format time, and the size of a disk is fixed, so there can be no problem with ever needing to allocate more VBM blocks.

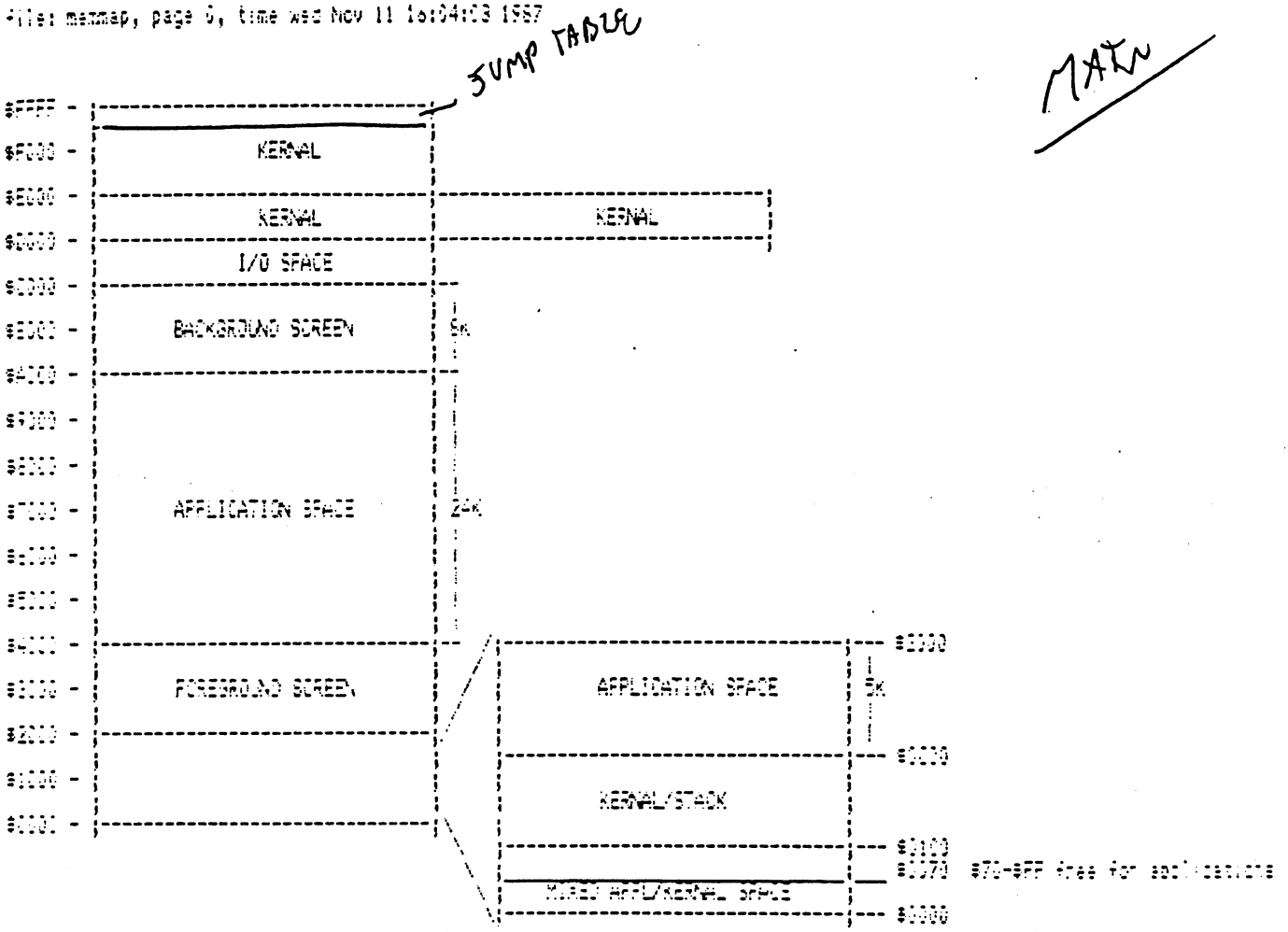
Therefore, the differences between the ProDOS VBM and the Commodore BAM are its location, its possible size, and its format. The location of the VBM is not in the directory header. The size of the VBM, although fixed for a given drive type, is different for different capacity drives. The VBM does not contain a count of free blocks.

These differences affect the GEOS disk routines that deal with the VBM/BAM. These routines are: CalcBlksFree, FreeBlock, SetNextFree, SetIfFree, and FindVBMBit (FindBAMBit). For example, CalcBlksFree can no longer just add up the number of free blocks per track (which is stored in the Commodore BAM). In the Apple version, CalcBlksFree has to explicitly check each bit of the VBM, and do the appropriate calculations to determine the number of free blocks on a disk. It might even have to read in another disk block.

The routines, as a group have changed because of a combination of two of the differences, the location and the size. The routines now employ a type of cache for the VBM. The routines have a single block buffer in memory that it uses to hold the "current" VBM block. This makes sense in multiple-block VBMs, but with 143K 5.25" floppies and 800K 3.5" microfloppies, this buffer always holds the one and only VBM block. The routines always maintain a correct copy of the current VBM block in the buffer. Whenever access is needed to a different VBM block, the current block, if it has changed, is written out to disk before reading in the new block.

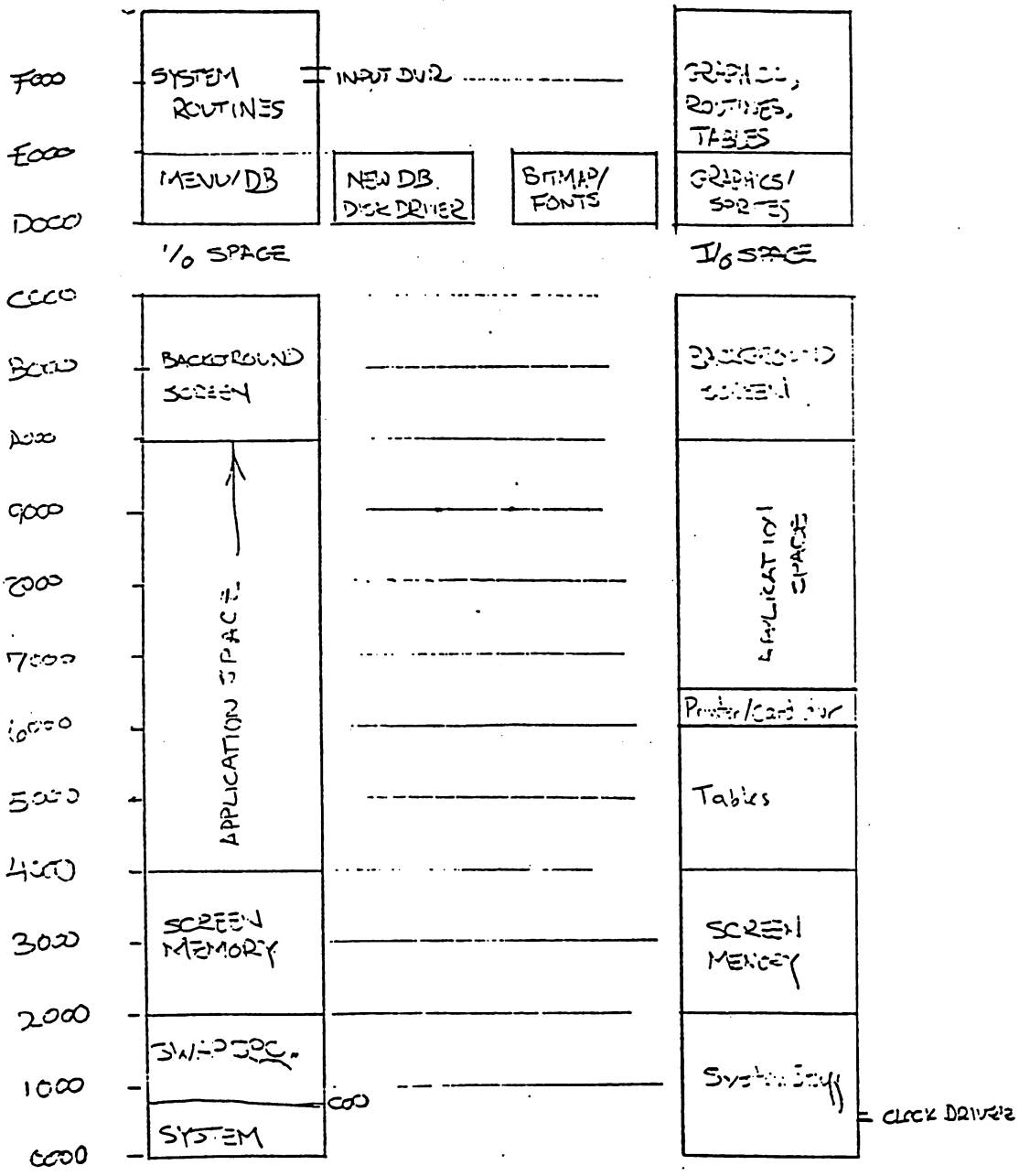
APPLE MEMORY MAP



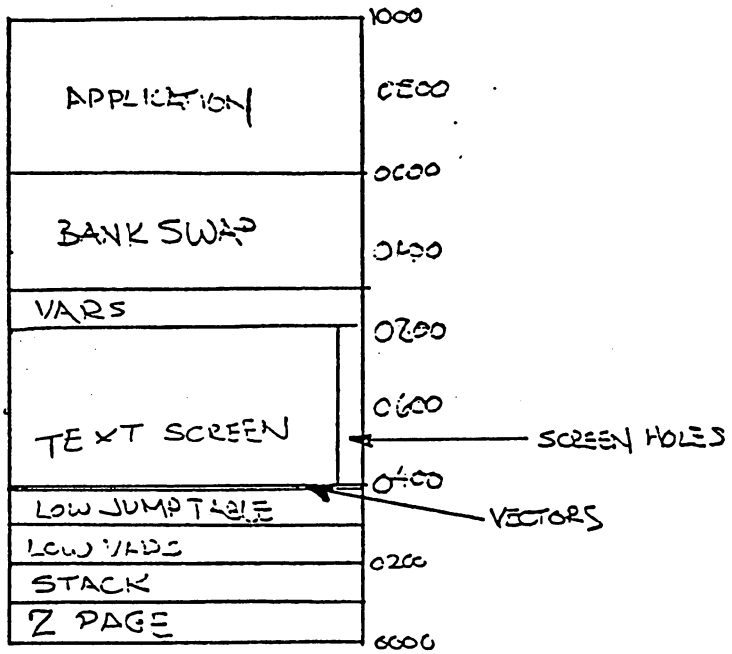


JUMP TABLES EXIST IN BOTH BANKS BUT W/ DIFFERENT INTERNAL ROUTINE TO HANDLE BANK SWITCHING.

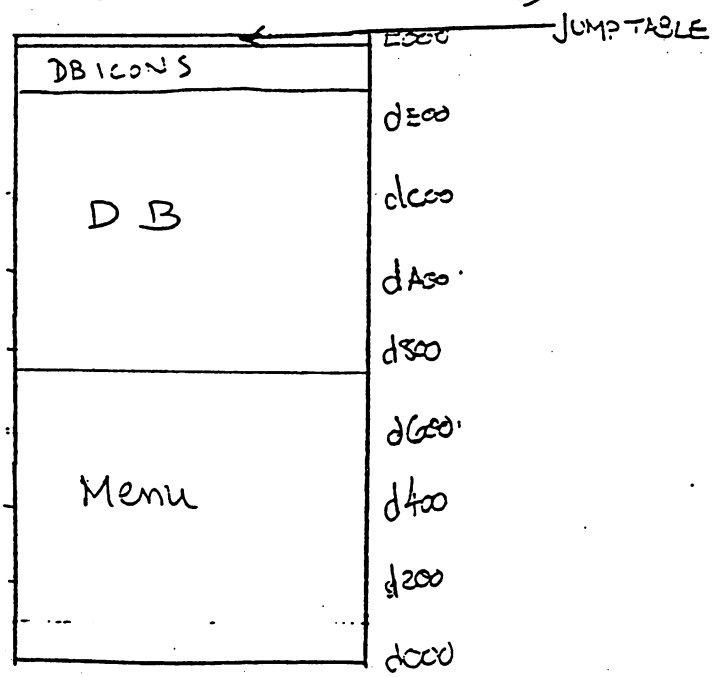
APPLE GEOS


F000
E000
D000
C000
B000
A000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0000
SYSTEM ROUTINES
INPUT DVR
MENU/DB
NEW DB DISK DRIVER
BITMAP/ FONTS
GRAPHICS/ SPRITES
I/O SPACE
BACKGROUND SCREEN
APPLICATION SPACE
SCREEN MEMORY
SWAP SPACE
SYSTEM
GRAPHICS, ROUTINES, TABLES
I/O SPACE
BACKGROUND SCREEN
APPLICATION SPACE
Printer/Card Drv
Tables
SCREEN MEMORY
System Stuff
= CLOCK DRIVERS

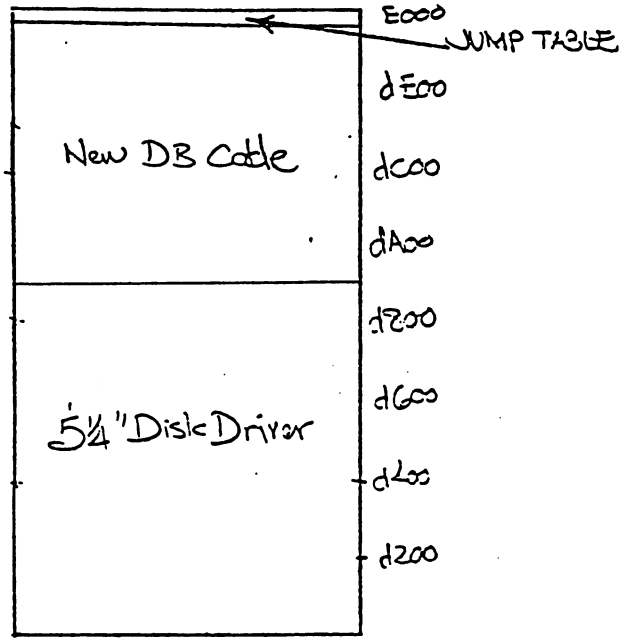
SYSTEM (0000-1000 MAIN)



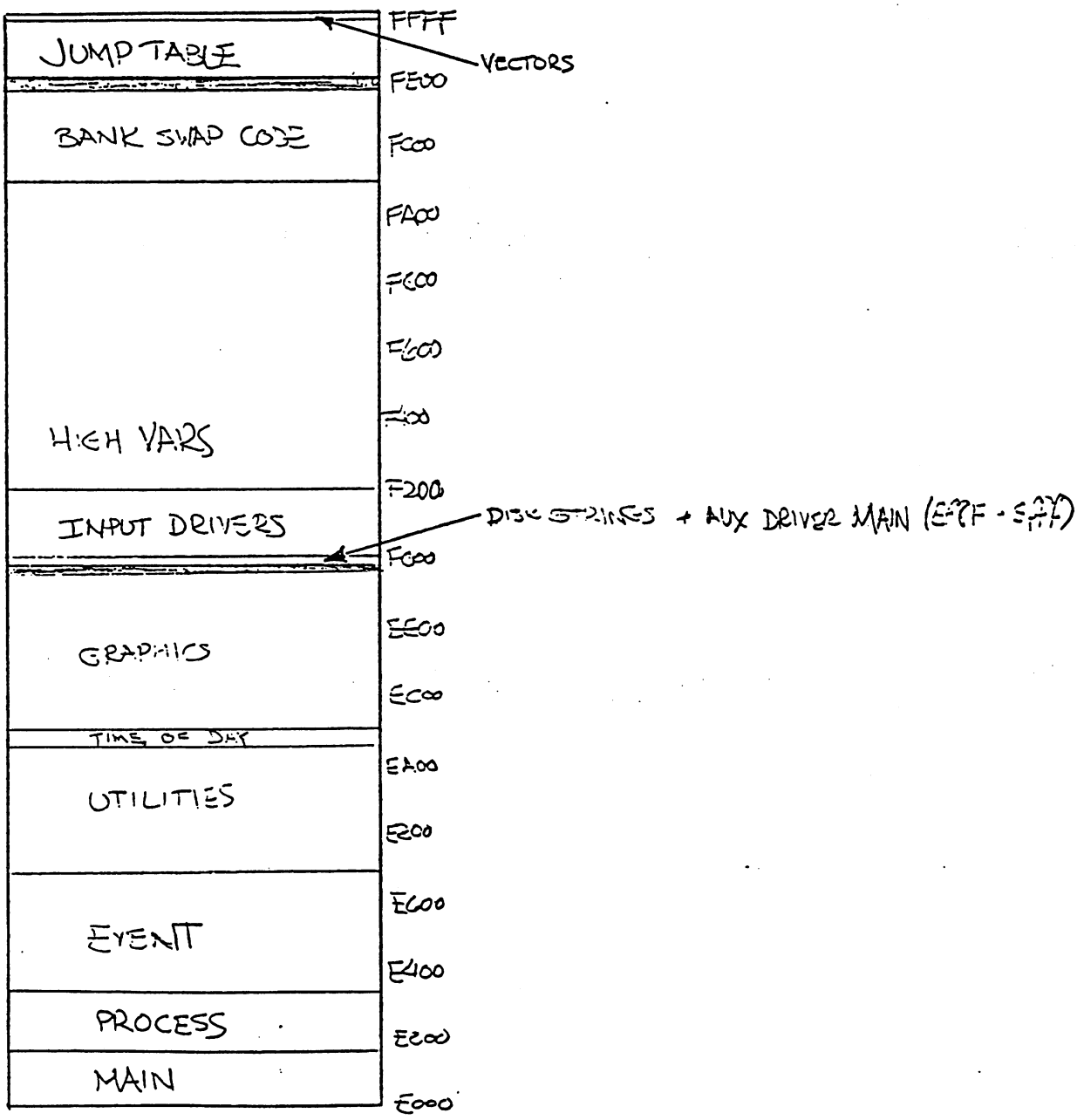
MENU/DB (d000 - dfff)



NEW DB / DISK DRIVER (d000-dfff alt. main

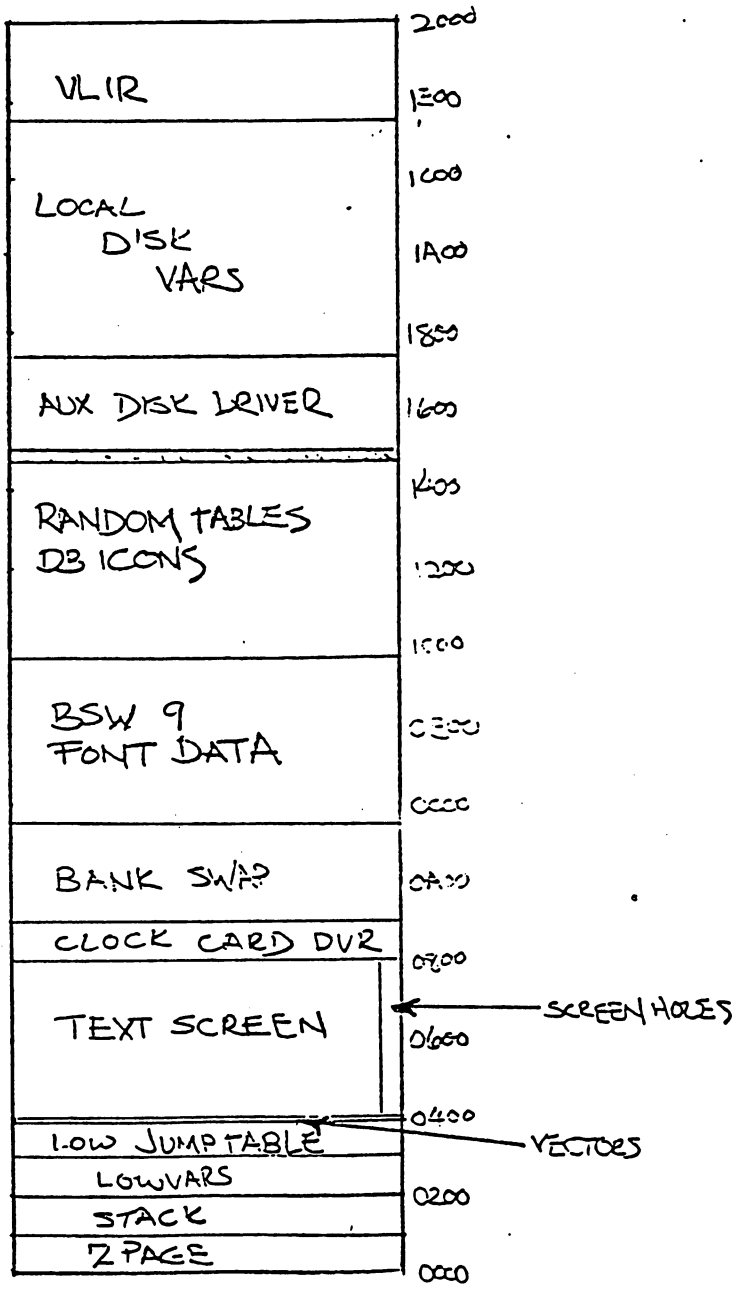


SYSTEM ROUTINES (E000 - FFFF MAIN)

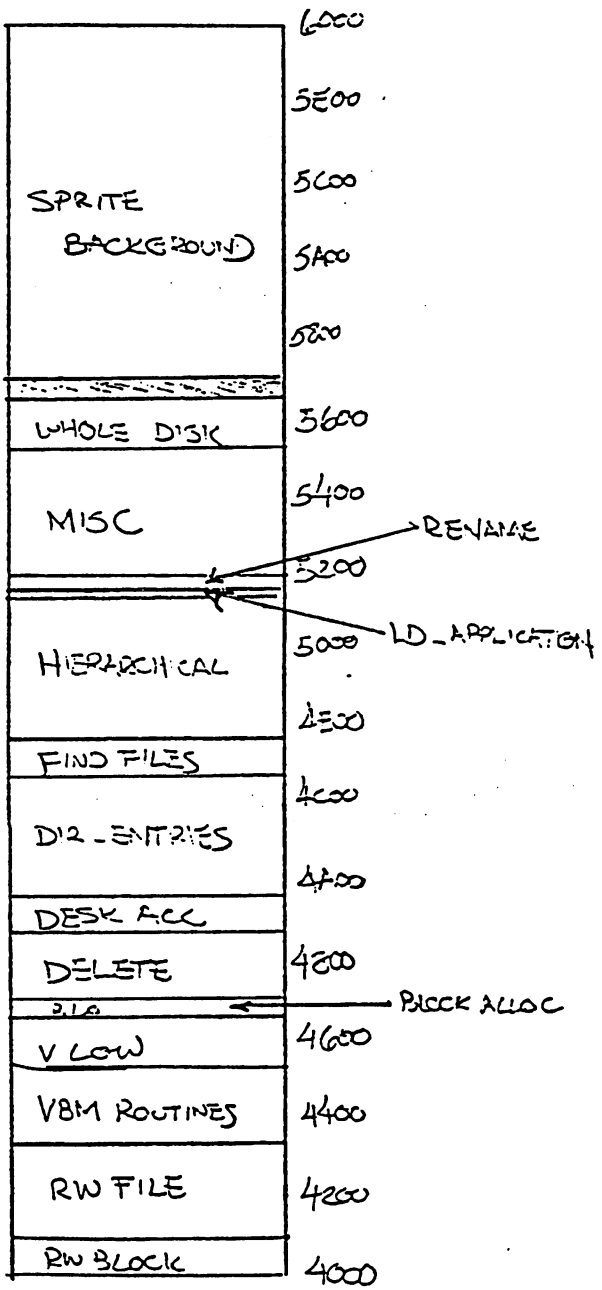


12/22 RLL

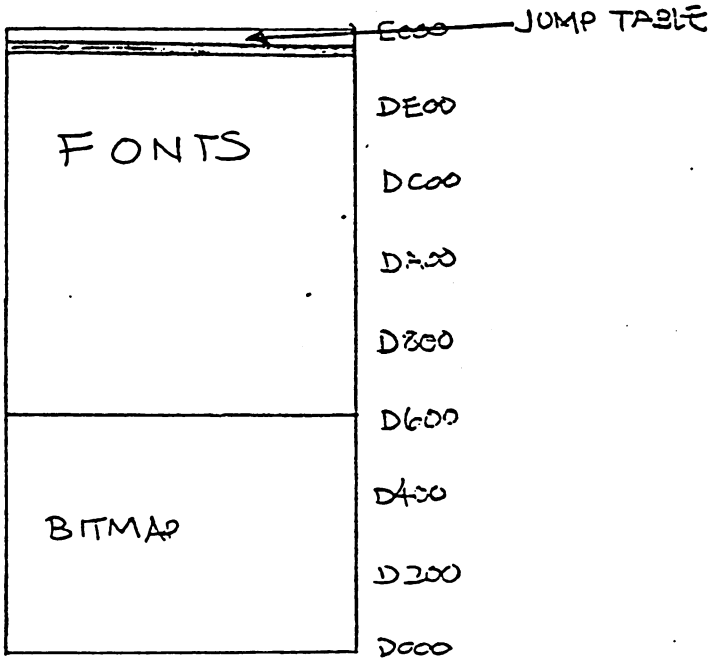
SYSTEM STUFF (0000-2000 AUX



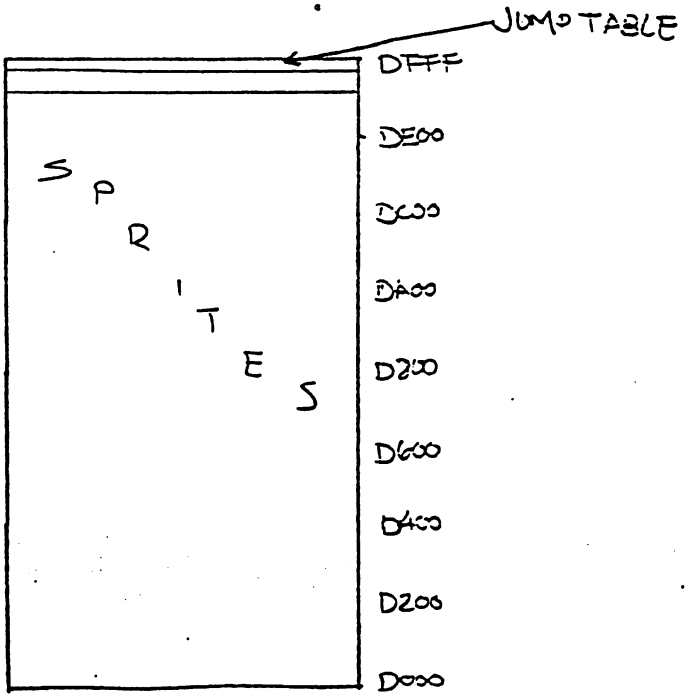
TABLES (4000 - 6000 AUX)



BITMAP/FONTS (D000–DFFF AUX)



SPRITES (D000-DFFF ALT AUX)



File: AppleSpriteGuide (128spriteGuide)
Author: Jim DeFrisco
Date: 14 January 1987

Purpose: This file explains what needs to be done to convert the portions of C64 GEOS applications that deal with sprites to run under Apple GEOS.

Background Info:

The C64 contains a chip to handle sprites in hardware. Unfortunately, this chip is not available on the Apple, so the functions of that chip have been simulated in software that is included in the kernal. Most of the capabilities of the VIC chip have been taken care of, and if you are not doing exotic things with sprites your code may work with one or two changes.

The major changes include: sprite 0 (the cursor) is treated differently than any other sprite. The code for this beast has been optimized to get reasonably fast mouse response, with a resulting loss in functionality. You cannot double the cursor's size in either x or y. You cannot change the color of the cursor. The size of the cursor is limited to 16-pixels wide and 8 lines high. One added feature is the ability to add a white outline to the picture that is used for the cursor. This allows it to be seen while moving over a black background.

For the other 7 sprites, all the capabilities have been emulated except for color and collision detection. In addition, the 64th byte of the sprite picture definition (previously unused) is now used to provide some size info about the sprite. This is used to optimize the drawing code.

To change your code over, look over the following list of possible problem areas. This list will be updated as more problems are found, and you should add an entry when you come up with a new one.

Sprite Problem Areas:

To make sure your sprite code will run under AppleGEOS, read over the following list. If you see a problem that applies to your application, take the appropriate corrective action. If you encounter a problem that is not listed below, add a solution when you come up with one (or state that a solution is not possible).

PROBLEM	SOLUTION
Writing directly to the screen	Since the old sprite were handled with hardware, writing to the screen wasn't a problem. If you do it (system calls NOT included), then call "TempHideMouse" before the write. This will erase the cursor and any sprites you have enabled. You don't have to do anything to get them back, this is done automatically during the next main loop.
Change the cursor picture (sprite 0)	This should be done with the new call "SetMousePicture". The source for this routine is in /staff/jim/applegeos/Sprite/softSprites. There is a limit to the size of the cursor to 16 pixels wide and 8 lines high, and you cannot use "double in x or y".
Use a large picture for the cursor.	You should discontinue using sprite 0 for this. When you need to change the arrow to something large, disable sprite 0 and use another sprite that tracks the mousePosition variables. Right now, the only way to disable sprite 0 is to give it a null picture or clear the low bit of mobenble. A "disable mouse" call should be added soon.
All sprite picture data	All picture data should be adjusted to include the 64th byte. This byte has size information that is read by the software sprite routines, even if they are garbage values. The format of this byte is: high bit set means that the sprite is no more that 8 pixels wide (this means it can be shifted 7 times and still be contained in 2 bytes). The rest of the byte is a count of the scan lines in the sprite. You can either include this info as part of the picture definition, or stuff it into the right place with some special code.

PROBLEM	SOLUTION
Writing directly to the VIC chip	This is generally ok, since the sprite emulation routines take the position and doubling info from the registers on the VIC chip, with the exception of the x position. The VIC chip allows 9 bits for x positions, which is not enough for a 560-wide screen. You should write the x position to the global variables "reqXpos0, reqXpos1..." (request x pos). These are full words, in consecutive locations. Better yet, use the "PosSprite" call in the kernel.
Reading values from the VIC chip	This is also ok for the status values and for the y position. The x position should be read from the reqXpos0 regs.
Using VIC chip collision detection	There is none.
Writing to the VIC chip (or calling PosSprite, EnablSprite, DisablSprite) at interrupt level	Don't do it. Since the sprites are drawn at main loop, this causes subtle, irreproducable timing bugs that are impossible to get out. It is all right to change the Mouse's position here, since the mouse is drawn at interrupt level also.
Turning the mouse On/Off	The mouse cursor is the only "sprite" that is drawn at interrupt level. There is now a byte (called offFlag) that is used to tell the interrupt code what the status of the mouse currently is. When TempHideMouse (or any graphics routine) is called, the mouse is erased from the screen. The value $80 is stored in offFlag to tell the interrupt code not to draw the mouse. While the value in offFlag is $80, the mouse will NOT be redrawn or erased. When we wish to have the mouse redrawn, we store the value $40 in offFlag, which tells the interrupt code to resume drawing the mouse. Main Loop stores this value in offFlag when the application returns to it. A value of $00 in offFlag allows the interrupt code to function normally (erase and redraw the mouse whenever necessary) and is stored there by the interrupt code.

Font changes:

Now the system font is located in auxilliary memory. There is plenty of space on the aux side, so the applications will probably want to use it for holding the font data. Certain flags need to be set to tell the font routines that the fonts are in auxilliary memory. If the font is in main memory, the application should call LoadCharSet as usual. But, if the font is in auxilliary memory, it should call LoadAuxCharSet, which sets a bunch of flags and vars for the font routines.

Most of the font variables still exist in AppleGEOS. Three variables lie in main memory where the application can easily access them:

baselineOffset
currentSetWidth
currentHeight

Two of the variables lie in the alternate zero page:

currentIndexTable
cardDataPointer

If the application needs to access these, It must first do a 'sta ALTZP_ON' and then access them, and then do a 'sta ALTZP_OFF'. Remember that ALL of the main (application) zero page and stack is swapped out here, so it cannot be accessed until the 'sta ALTZP_OFF' is executed.

This file contains the many changes made in the dialog box routines:

DBUSRROUTINE: We now assume that only ONE DBUSRROUTINE call is made per dialog box. This call is now made AFTER the icons are displayed. This makes it possible to use DBUSRROUTINE to put stuff up over the icons...

DBGETFILES: Holding the button down on the scroll icon makes the scrolling continue. Ho Hum... Also, the buffer that holds the filenames is in auxilliary memory, and will hold 32 filenames.

DBGETNAMES: This is exactly like DBGETFILES except that the user passes the address of a routine to load a buffer with the names to be displayed. This routine should:

Clear out the buffer in auxilliary memory from $7B00 - $7B50

Clear the high bit of r7H and decrement r7H for each name stuck in the buffer.

Put up to 32 file names in the buffer (starting at $7B00 in aux mem) The names may be up to 15 characters long, + a null terminator.

A buffer can be cleared in auxilliary memory by doing a "sta RAMWRT_ON" before the call to ClearRam and a "sta RAMWRT_OFF" afterwards.

The Dialog Box table entry for this routine is as follows:

```
.byte   DBGETNAMES
.word   Address of routine to get names
.byte   xpos_offset of file box from corner of DB
.byte   ypos_offset of file box from corner of DB
```

NEWDBGETFILES: This routine has a whole slew of fun new features... You can see the spec file for a description. Here is what you need to know to use it:

The NEWDBGETFILES dialog box puts up its own OPEN icon. But the application can treat it as if it were its own OPEN icon.

The NEWDBGETFILES box returns the selected file name in the same manner as DBGETFILES does... It also returns the directory block in which the file lies in the global variable curKBlkno.

The NEWDBGETFILES table entry is of the following format:

```
.byte   NEWDBGETFILES
.byte   file box x offset
.byte   file box y offset
.byte   open icon x offset (bytes)
.byte   open icon y offset
.byte   directory icon/pathname text x offset
.byte   directory icon/pathname text y offset
```

ICONS: There were many new system icons added also. The complete list is now:

CANCEL,OK,YES,NO,OPEN,DISK,DRIVE,QUIT,CREATE and IGNORE

These can be used by applications outside of the dialog boxes. They lie in auxilliary memory, and their addresses are equated to the constants: CANCEL_ICN_ADDRESS,OK_ICN_ADDRESS, etc...

Also, the dialog boxes now allow 16 icons to be displayed!

For PutString:

A change has been made in PutString to fix a small 'bug'. In C64 GEOS, PutString does not handle any margin faults (i.e. If we attempt to print out 'WWWiii', and only 'WW' fits, PutString will keep trying to output characters until the end of the string is reached). Now, PutString will test to see if there is a user specified StringFaultVector. If not (StringFaultVector = $0000), then it will put its own StringFaultVector in, which will merely exit the PutString routine when the margin is reached.

A change has been made in GetString. SystemStringService now always null-terminates the string in the buffer (i.e. even while the user is inputting the string, it is null-terminated).

This file contains the changes made in the icon and menu routines.

Icons:

The icon pictures can now be "double width" and can also lie in aux memory. Also, the icon routines currently move the mouse when they are run. This option can be disabled. Within the table, flags are set to show that these options are active. These flags are:

```
.word   XPositionOfMouse        ;Set high bit to disable mouse
                                ; move

.byte   X_POS_ICON              ;Set high bit to signify that
                                ;the icon picture lies in
                                ;auxilliary memory

.byte   ByteWidthOfIcon         ;Set high bit to signify that
                                ;we want the icon data to be
                                ;double-width
```

Menus:

DoMenu currently places the mouse over the menu item passed in the accumulator... Now, if the value in the accumulator is negative, the DoMenu routine does not move the mouse.

For bitmaps:

There are now two entry points to the bitmap routines. The old ones still work (BitmapClip, BitmapUp, etc). There are now also new routines with a new format (NewBitmapClip, NewBitmapUp). This format allows routines to place bitmaps on a pixel boundary and to let a bitmap have a pixel width. The pixel width convention is as follows:

Assuming a bitmap with a pixel width of 11, the first line is represented by X's and the second line is represented by Y's. The bitmaps must lie in memory as follows:

```
byte #0:    XXXXXXXX
byte #1:    XXXUUUUU        ;The pixel boundary lies within a byte
byte #2:    YYYYYYYY        ; so the remainder of the byte will be
byte #3:    YYYUUUUU        ; ignored (U stands for unused bits)
```

Also, the BitmapUp routines allow the icons to be doubled in width. To use this option, set the high bit (bit 7) in the high byte of the pixel width (for NewBitmapUp) or the high bit in the byte width (for BitmapUp). The routine will double the width of the bitmap, but not the x coordinate of the bitmap .Also, the routines allow the bitmap to be in auxilliary ram. To enable this option, set the high bit in the x coordinate.

The parameters that are passed to the routines have changed. These changes are described here:

****

Parameters for BitmapUp:

Pass:

```
r0 - pointer to the bit mapped data
r1L - x byte position for the bit mapped data (0-69)
        NOTE:  if bit 7 is set in r1L, the bitmap is in aux. ram
r1H - y pixel position for the bit mapped data (0-191)
r2L - width in bytes of the bit mapped data (0-70)
        NOTE:  if bit 7 is set in r2L, the icon is doubled in X
                 (bitmap width is doubled)
r2H - height in pixels of the bit mapped data (1-192)
```

Pass: (inline)

```
.word   dataPointer
.byte   xCoord
.byte   yCoord
.byte   width
.byte   height
```

Parameters for NewBitmapUp:

Pass:

```
r0 - pointer to the bit mapped data
r1L - height in pixels of the bit mapped data (1-192)
r1H - y pixel position for the bit mapped data (0-191)
r2 - width in pixels of the bit mapped data (0-560)
        NOTE:  if bit 7 is set in r2H, the icon is doubled in X
                 (bitmap width is doubled)
r3 - x pixel position for the bit mapped data (0-559)
        NOTE:  if bit 7 is set in r3H, the bitmap is in aux. ram
```

Pass: (inline)

```
.word   dataPointer
.word   xCoord
.byte   yCoord
.word   width
.byte   height
```

```
*******************************************************************************

Parameters for BitmapClip:

Pass:
        r0 - pointer to the bit mapped data
        r1L - x byte position for the bit mapped data (0-69)
                        NOTE:  if bit 7 is set in r1L, the bitmap is in aux. ram
        r1H - y pixel position for the bit mapped data (0-192)
        r2L - width in bytes of the bit mapped data (to print) (0-70)
                        NOTE:  if bit 7 is set in r2L, the icon is doubled in X
                                (bitmap width is doubled)
        r2H - height in pixels of the bit mapped data (to print) (1-192)

        r11L - number of bytes to skip each row
        r11H - number of bytes to skip after printing each row
        r12 - number of lines to skip before printing

Parameters for NewBitmapClip:

Pass:
        r0 - pointer to the bit mapped data
        r1L - height in pixels of the bit mapped data (to print) (1-192)
        r1H - y pixel position for the bit mapped data (0-192)
        r2 - width in pixels of the bit mapped data (to print) (0-560)
                        NOTE:  if bit 7 is set in r2H, the icon is doubled in X
                                (bitmap width is doubled)
        r3 - x pixel position for the bit mapped data (0-559)
                        NOTE:  if bit 7 is set in r3H, the bitmap is in aux. ram

        r11L - number of bytes to skip each row
        r11H - number of bytes to skip after printing each row
        r12 - number of lines to skip before printing

*******************************************************************************
```

```
;*************************************************************************
;                               ReadBlock
;       synopsis
;               provides the Apple equivalent of the Commodore's lou level
;               routine
;       Author:
;               Brian Chin              February 1988
;       Called by:
;       Pass:
;               r1 -- block number
;               r4 -- buffer address
;               RWbank -- bank to use
;       Accesses:
;               curDrive -- drive to read block from
;       Returns:
;               x -- disk error
;       Alters:
;               a, y
;       Calls:
;               GetBlock
;*************************************************************************

ReadBlock:
        jmp     GetBlock                ;these are equaivalent routines
```

```
;*************************************************************************
;                               WriteBlock
;       synopsis
;               provides the Apple equivalent of the Commodore's low level
;               routine
;       Author:
;               Brian Chin              February 1988
;       Called by:
;       Pass:
;               r1 -- block number
;               r4 -- buffer address
;               RWbank -- bank to use
;       Accesses:
;               curDrive -- drive to read block from
;       Returns:
;               x -- disk error
;       Alters:
;               a, y
;       Calls:
;               PutBlock
;*************************************************************************

WriteBlock:
        PushB   numDiskRetries          ;save global retry count
        LoadB   numDiskRetries,#0       ;set to zero for no retries
        jsr     PutBlock                ;put block to disk w/o retries
        PopB    numDiskRetries          ;restore number of retries
        rts                             ;return with error from PutBlock
```

```
;*********************************************************************
;                           VerWriteBlock
;       synopsis
;               provides the equivalent of the Commodore GEOS Kernal
;               routine
;       Author:
;               Brian Chin              February 1988
;       Called by:
;       Pass:
;               r1 -- block number
;               r4 -- buffer address
;               RWbank -- bank to use
;       Accesses:
;               curDrive -- drive to write
;               numDiskRetries -- global retry count
;       Returns:
;               x -- disk error status
;       Alters:
;               a, y
;       Calls:
;               ReadBlock (Kernal routine)
;               WriteBlock (routine above)
;*********************************************************************

generalBuf      =       $1767   ;address of Kernal's internal diks buffer in
                                ;       AUXILIARY memory

VerWriteBlock:

        PushB   numDiskRetries  ;save global retry count
        ldx     #0              ;assume no error in case of no retries
10$:
        lda     numDiskRetries  ;get global retry count
        beq     90$             ;if no retries, do nothing

        PushW   r4              ;save buffer address
        PushB   RWbank          ;save bank status
        LoadW   r4,#generalBuf  ;use Kernal internal block buffer
        LoadB   RWbank,#AUX
        jsr     ReadBlock       ;read block to verify it
        PopB    RWbank          ;restore bank status
        PopW    r4              ;restore buffer address
        txa
        beq     90$             ;if no error, exit

        dec     numDiskRetries  ;adjust retry count to indicate upcoming attempt

        jsr     WriteBlock      ;call our write block routine (above) that
                                ;       does no retries
        txa
        beq     10$             ;if no error, loop back to verify it
                        ;if error, exit immediately

90$:
        PopB    numDiskRetries
        rts
```

GEOS Variable documentation

Preliminary release 3/23/88

lastWidth
tegOlpos, etc.

lastMenu/RecoverOnce

Name: alarmSetFlag

Formerly:

Address: C64: $851c C128: $851c Apple: NA

Size: Byte

Default: FALSE

Saved?: No

Description: TRUE if the alarm is set for geos to monitor, else FALSE.

Note:

Name: alarmTmtVector

Formerly:

Address: C64: $84ad C128: $84ad Apple: $0212

Size: Word

Default: 0

Saved?: Yes

Description: address of a service routine for the alarm clock time-out (ringing, graphic etc.) that the application can use if necessary.

Note:

Name: alphaFlag

Formerly:

Address: C64: $84b4 C128: $84b4 Apple: $0225

Size: Byte

Default: 0 if not getting text input
11xxxxxx if getting text input, where xxxxx are counter bits

Saved?: Yes

Description: Flag for alphanumeric string input
bit 0-5 - Counter before prompt flashes
bit 6 - Flag indicating prompt is visible
bit 7 - Flag indicating alphanumeric input is on

Note:

Name: appMain

Formerly: applicationMain

Address: C64: $849b C128: $849b Apple: $0200

Size: Word

Default: $00

Saved?: No

Description: Vector that allows applications to include their own main loop code. The code pointed to by appMain will run at the end of every GEOS mainloop.

Note:

Name: backBufPtr

Formerly:

Address: C64: NA C128: $131b Apple: $9d7c

Size: 16 bytes

Default: None

Saved?: No

Description: Screen pointer where the back buffer came from.
Resides in back ram of C128 and Apple.

Note:

Name: bakclr0, bakclr1, bakclr2, bakclr3

Formerly:

Address: C64: $do21 - $do24 respectively
C128: $do21 - $do24 respectively
Apple: NA

Size: 1 Byte each

Default: ???????????

Saved?: No

Description: Background colors 0 - 3

Note:

Name: backXBufNum

Formerly:

Address: C64: NA C128: $132b Apple: $d98c

Size: 8 bytes

Default: None

Saved?: No

Description: For each sprite, there is one byte here for how many bytes wide the corresponding sprite is. Used by C128 and Apple soft sprite routines and resides in back ram.

Note:

Name: backYBufNum

Formerly:

Address: C64: NA C128: $1333 Apple: $d994

Size: 8 bytes

Default: None

Saved?: No

Description: For each sprite, there is one byte here for how many scanlines high the corresponding sprite. Used by C128 and Apple soft sprite routines and resides in back ram.

Note:

Name: bootName

Formerly:

Address: C64: $C006 C128: $C006 Apple: $E003

Size: 9 Bytes

Default: GEOS BOOT

Saved?: No

Description: This is the start of the "GEOS BOOT" string.

Note:

Name: BRKVector

Formerly:

Address: C64: $84af C128: $84af Apple: $0214

Size: Word

Default: Commodore - $Cf85
Apple - $FEF3

Saved?: Yes

Description: Vector to the routine that is called when a BRK instruction is encountered. The default is to the operating system "System Error" dialog box routine.

Note:

Name: bkvec

Formerly:

Address: C64: $0316 C128: $316 Apple: NA

Size: Word

Default: ????????

Saved?: No

Description: BRK instruction vector for when ROMs are switched in.

Note:

Name: baselineOffset

Formerly:

Address: C64: $26 C128: $26 Apple: $0218

Size: Byte

Default: $06 - for BSW 9 font

Saved?: Yes

Description: Offset from top line to baseline in character set. i.e. it changes as fonts change.

Note:

Name: CPU_DATA

Formerly:

Address: C64: $0001 C128: NA Apple: NA

Size: Word

Default: RAM_64K ($30)

Saved?: No

Description: Address of 6510 data register that controls the harduare memory map of the C64. The following constants are used with CPU_DATA.

Value	Mapping
IO_IN	60K RAM, 4K I/O space
RAM_64K	64K RAM
KRNL_BAS_IO_IN	both Kernal and basic ROMs mapped into memory
KRNL_IO_IN	Kernal ROM and I/O space mapped into memory

Note:

Name: CPU_DDR

Formerly:

Address: C64: $0000 C128: NA Apple: NA

Size: Byte

Default: $00

Saved?: No

Description: address of 6510 data direction register

Note:

Name: cardDataPntr

Formerly: cardDataPointer

Address: C64: $2c C128: $2c Apple: $60

Size: Word

Default: Commodore: $D2DC - for BSW 9 font
Apple: $0150 - for BSW 9 font

Saved?: Yes

Description: This is a pointer to the actual card graphics data for the current font in use

Note:

Name: curDirHead

Formerly:

Address: C64: $8200 C128: $8200 Apple: $FA80

Size: Commodore - 256 bytes
Apple - 39 bytes

Default: $00

Saved?: No

Description: For Commodore, it is the buffer containing header information for the disk in currently selected drive.
On the Apple, curDirHead contains the header of the current directory. Initialized to all zeros.

Note:

Name: curDevice

Formerly:

Address: C64: $BA C128: $ba Apple: NA

Size: Byte

Default: Commodore = $08

Saved?: No

Description: This holds the current serial device number. See curDrive for more information

Note:

Name: curDrive

Formerly:

Address: C64: $8489 C128: $8489 Apple: $F60d

Size: Byte

Default: Commodore = $08
Apple = $00

Saved?: No

Description: Holds the device number of the currently active disk drive. For Commodore, allowed values are 8 - 11. For Apple, drives are numbered 0 - 3.

Note:

Name: curEnable

Formerly:

Address: C64: NA C128: $1300 Apple: $0951

Size: Byte

Default: None

Saved?: No

Description: This is an image of the C64 mobenble register. Used for C128 and Apple soft sprites.

Note:

Name: curHeight

Formerly: currentHeight

Address: C64: $29 C128: $29 Apple: $021B

Size: Byte

Default: $09 for BSW 9 font

Saved?: Yes

Description: Used to hold the card height in pixels of the current font in use.

Note:

Name: curIndexTable

Formerly: currentIndexTable

Address: C64: $2a C128: $2a Apple: $005E

Size: Word

Default: Commodore: $D218 for BSW 9 font
Apple: $0C08 for BSW 9 font

Saved?: Yes

Description: curIndexTable points to the table of sizes, in bytes, of each card in of the current font. On the Apple, this variable only exists in the back ram zero page. Applications must switch zero pages before accessing it directly.

Note:

Name: curmobx2

Formerly:

Address: C64: NA C128: $1302 Apple: $D953

Size: Byte

Default: None

Saved?: No

Description: Image of the C64 mobx2 register. Used for C128 and Apple soft sprites. Resides in back ram.

Note:

Name: curmoby2

Formerly:

Address: C64: NA C128: $1301 Apple: $D952

Size: Byte

Default: None

Saved?: No

Description: Image of of C64 moby2 register. Used for C128 and Apple soft sprites. Resides in back ram.

Note:

Name: curPattern

Formerly: currentPattern

Address: C64: $22 C128: $22 Apple: $022c

Size: Word

Default: $D010

Saved?: Yes

Description: curPattern points to the first byte of the graphics data for the current pattern in use.

Note: Each pattern is 1 byte wide and 8 bytes bytes high, to give an 8 by 8 bit pattern.

Apple: The location of patterns pointed to bu curPattern is in back ram and should not be accessed directly. See SetPattern and GetPattern.

Name: curRecord

Formerly:

Address: C64: $8496 C128: $8496 Apple: $f618

Size: Byte

Default: $00

Saved?: No

Description: Holds the current record number for an open VLIR file.

Note: When a VLIR file is opened, using OpenRecordFile, curRecord is set to 0 if there is at least 1 record in the file, or -1 if their are no records.

Name: currentMode

Formerly:

Address: C64: $2e C128: $2e Apple: $021C

Size: Byte

Default: $00

Saved?: Yes

Description: Holds the current text drawing mode. Each bit is a flag for a drawing style. If set, that style is active, if clear it is inactive. The bit usage and constants for maninuplating these bits are as follows.

Bit	Style	Constant
b7:	Underline flag	SET_UNDERLINE = %10000000
b6:	Bold flag	SET_BOLD = %01000000
b5:	Reverse flag	SET_REVERSE = %00100000
b4:	Italics flag	SET_ITALIC = %00010000
b3:	Outline flag	SET_OUTLINE = %00001000
b2:	Superscript flag	SET_SUPERSCRIPT=%00000100
b1:	Subscript flag	SET_SUBSCRIPT = %00000010
b0:	Unused	
Clears all flags (plain text)		SET_PLAINTEXT = %00000000

Any combination of flags can be set or clear. If current mode is plaintext, all flags are clear.

Constants that can be used within text strings themselves that affect currentMode are:
UNDERLINEON, UNDERLINEOFF, REVERSEON, REVERSEOFF, BOLDON, ITALICON, OUTLINEON, PLAINTEXT

Note:

Name: curSetWidth

Formerly: currentSetWidth

Address: C64: $27 C128: $27 Apple: $219

Size: Word

Default: Commodore: $003c
Apple: $0051

Saved?: Yes

Description: Holds the card width in pixels for the current font

Note:

Name: curType

Formerly:

Address: C64: $88c6 C128: $88c6 Apple: NA

Size: Byte

Default: Disk type of drive 8 for Commodore

Saved?: No

Description: Holds the current disk type. This value is copied from driveType for quicker access to the current drive
b7: Set if the disk is a RAM disk
b6: Set if using disk shadowing

Only one of bit 6 or 7 may be set. Other constants used with curType are
DRV_NULL = 0 No drive present at this device address
DRV_1541 = 1 Drive type Commodore 1541
DRV_1571 = 2 Drive type Commodore 1571
DRV_1581 = 3 Drive type Commodore 1581

Note:

Name: curXpos0 LOWMEM

Formerly:

Address: C64: NA C128: $1303 Apple: $D954

Size: 16 bytes LOCAL. 7

Default: None

Saved?: No

Description: The current X positions of the C128 and Apple soft sprites. Resides in back ram.

Note:

Name: curYpos0 LOWMEM

Formerly:

Address: C64: NA C128: $1313 Apple: $D964

Size: 8 bytes LOCAL.

Default: None

Saved?: No

Description: The current Y positions for the C128 and Apple soft sprites. Resides in back ram.

Note:

Name: devUnitTab

Formerly:

Address: C64: NA C128: NA Apple: $FAEF

Size: 4 Bytes

Default: $60 in each byte

Saved?: No

Description: The ProDos unit numbers of the four possible devices are kept here. Used for communicating with the device drivers.

Note:

Name: devTabHi

Formerly:

Address: C64: NA C128: NA
Apple: devTabHi - $FAE7, devTabLo - $FAEB

Size: 4 Bytes each

Default: devTabHi - $D0 in each byte, devTabLo - $00 in each byte

Saved?: No

Description: For the Apple, these are the high and low bytes of the four possible device drivers.

Note:

Name: dirBlkno

Formerly:

Address: C64: NA C128: NA Apple: $F620

Size: Word

Default: $02

Saved?: No

Description: Block number of the key block of the directory containing this file's entry.

Note:

Name: dirPtr

Formerly:

Address: C64: NA C128: NA Apple: $F622

Size: Word

Default: $00

Saved?: No

Description: Pointer into diskBlkBuf for this file's entry.

Note:

Name: diskBlkBuf

Formerly:

Address: C64: $8000 C128: $8000 Apple: $F659

Size: Commodore - 256 bytes
Apple - 512 bytes

Default: $00

Saved?: No

Description: General disk block buffer. Initialized to all zeros.

Note:

Name: doRestFlag

Formerly:

Address: C64: NA C128: $1b54 Apple: $DB1d

Size: Byte

Default: FALSE = 0

Saved?: No

Description: Flag needed because of overlapping soft sprite problems on C128 and Apple. Set to TRUE if we see a sprite that needs to be redrawn and therefore all higher numbered sprites need to be redrawn as well. Resides in back Ram.

Note:

Name: driveType

Formerly:

Address: C64: $848e C128: $848e Apple: $FAF3

Size: 4 bytes.

Default: Set to type of drive 8 on Commodore or drive 0 on Apple.

Saved?: No

Description: There are 4 bytes at location driveType, one for each of four possible drives.

For Commodore, each byte has the following format:
b7: Set if drive is RAM DISK
b6: Set if Shadowed disk
Only 1 of bit 7 or bit 6 may be set

Constants and values used for drive types are

Constant	Value	Type
DRV_NULL	0	No drive present at this device address
DRV_1541	1	Drive type Commodore 1541
DRV_1571	2	Drive type Commodore 1571
DRV_1581	3	Drive type Commodore 1581

For Apple, the only differences between drive types is whether the drive media is removable (e.g. floppy) or non-removable (e.g. hard drive). The bit usage is as follows:

b7: set if disk medium is removable
b6: set if device is interruptable
b5,4: number of drives on the device (0-3)
b3: device driver supports format
b2: device driver supports write
b1: device driver supports read
b0: device driver supports status call

This bit usage conforms to the information ProDos keeps about disk drives. GEOS, however, only makes use of bits 7 and 3.

Note:

Name: drSizeLo, drSizeHi

Formerly: driveSizeLo, driveSizeHi

Address: C64: NA C128: NA
Apple: drSizeLo - $ FAF7, driveSizeHi - $FAFB

Size: 4 Bytes each

Default: None

Saved?: No

Description: The low and high bytes of the sizes of the four possible device sizes.

Note:

Name: dis g

Formerly:

Address: C64: $848a C128: $848a Apple: $F617

Size: Byte

Default: TRUE ($FF)

Saved?: No

Description: Set to TRUE or FALSE to indicate whether a disk is currently open.

Note:

Name: DrACurDkNm

Formerly:

Address: C64: $841e C128: $841e Apple: $FAA7

Size: Commodore - 18 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: This is the disk name of the current disk in drive A.
Commodore - padded with $A0
Apple - padded with $00

Note:

Name: DrBCurDkNm

Formerly:

Address: C64: $8430 C128: $8430 Apple: $FAB7

Size: Commodore - 18 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: This is the disk name of the current disk in drive B.
Commodore - padded with $A0
Apple - padded with $00

Note:

Name: dataFileName

Formerly:

Address: C64: $8442 C128: $8442 Apple: $02A4

Size: Commodore - 17 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: This is the name of a data to open. The name is passed to the parent application so the file can be openned.

Note:

Name: dataDiskName

Formerly:

Address: C64: $8453 C128: $8453 Apple: $02B4

Size: Commodore - 18 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: Holds the disk name that an application's data file is on.

Note:

Name: dispBufferOn

Formerly: displayBufferOn

Address: C64: $002f C128: $002f Apple: $021d

Size: byte

Default: (ST_WR_FORE | ST_WR_BACK) = $c0

Saved?: yes

Description: Routes graphic and text operations to either the foreground screen, background buffer, or both simultaneously.

b7: 1 = draw to foreground screen buffer
b6: 1 = draw to background buffer
b5: 1 = limit GetString text entry to foreground screen.
0 = GetString text entry will use b7,b6
b4-b0: reserved for future use; should always be 0

Use ST_WR_FORE (write to foreground) and ST_WR_BACK (write to background) to access these bits.

Note: %00xxxxxxxx is an undefined state and will result in sending most graphic operations to the center of the dispaly area.

Name: day

Formerly:

Address: C64: $8518 C128: $8518 Apple: $F202

Size: Byte

Default: 20

Saved?: No

Description: Holds the value for current day.

Note:

Name: dlgBoxRamBuf

Formerly:

Address: C64: $851f C128: $851f Apple: $F381

Size: Commodore - 417 Bytes
Apple - 649 Bytes

Default: None

Saved?: Yes

Description: This is the buffer for variables that are saved when desk accessories or dialog boxes are run.

Note:

Name: driveData

Formerly:

Address: C64: $88bf C128: $88bf Apple: NA

Size: 4 bytes

Default: None

Saved?: No

Description: One byte is reserved for each disk drive, to be used by the the disk driver. Each driver may use it differently.

Note:

Name: dblClickCount

Formerly:

Address: C64: $8515 C128: $8515 Apple: $0258

Size: Byte

Default: $00

Saved?: No

Description: Used to determine when an icon is double clicked on. When an icon is selected, dblClickCount is loaded with a value of CLICK_COUNT (30). dblClickCount is then decremented each interrupt. If the value is non-zero when the icon is again selected, then the double click flag (r0H) is passed to the service routine with a value of TRUE. If the dblClickCount variable is zero when the icon is clicked on, then the flag is passed with a value of FALSE.

Note:

Name: DrCCurDkNm

Formerly:

Address: C64: $88dc C128: $88dc Apple: $FAC7

Size: Commodore - 18 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: This is the disk name of the current disk in drive C.
Commodore - padded with $A0
Apple - padded with $00

Note:

Name: DrDCurDkNm

Formerly:

Address: C64: $88ee C128: $88ee Apple: $FAD7

Size: Commodore - 18 bytes
Apple - 16 bytes

Default: None

Saved?: No

Description: This is the disk name of the current disk in drive 0.
Commodore - padded with $A0
Apple - padded with $00

Note:

Name: dirEntryBuf

Formerly:

Address: C64: $8400 C128: $8400 Apple: $FA59

Size: Commodore - 256
Apple - 39 bytes

Default: $00

Saved?: No

Description: Buffer used to build a file's directory entry. Initialized to all zeros.

Note:

Name: dir2Head

Formerly:

Address: C64: $8900 C128: $8900 Apple: NA

Size: Commodore - 256 bytes

Default: None

Saved?: No

Description: This is the 2nd directory header block used for larger capacity disk drives (e.g. Commodore 1571)

Note:

Name: dateCopy

Formerly:

Address: C64: $C018 C128: $C018 Apple: NA

Size: 3 bytes

Default: Same as variables year, month, day

Saved?: No

Description: This is a copy of the system variables for year, month, and day.

Note:

Name: extclr

Formerly:

Address: C64: $D020 C128: $D020 Apple: NA

Size: Byte

Default: $FB

Saved?: No

Description: Holds value for exterior (border) color.

Note:

Name: faultData

Formerly:

Address: C64: $84b6 C128: $84b6 Apple: $0227

Size: Byte

Default: $00

Saved?: Yes

Description: Holds information about mouse faults. Mouse faults occur when the mouse attempts to move outside the bounds set by mouseLeft, mouseRight, mouseTop, and mouseBottom. A fault is also signalled when the mouse is outside the current menu area. The bits for signalling are used as follows:

Bit	Fault	Constant for bit access
b7:	mouse fault up	OFFTOP_BIT
b6:	mouse fault down	OFFBOTTOM_BIT
b5:	mouse fault left	OFFLEFT_BIT
b4:	mouse fault right	OFFRIGHT_BIT
b3:	menu fault	OFFMENU_BIT

Note:

Name: fileHeader

Formerly:

Address: C64: $8100 C128: $8100 Apple: $F859

Size: Commodore - 256 bytes
Apple - 512 bytes

Default: $00

Saved?: No

Description: Buffer used to hold the header block for a GEOS file.

Note:

Name: fileSize

Formerly:

Address: C64: $8499 C128: $8499 Apple: $F61B

Size: Word

Default: None

Saved?: No

Description: This is the current size (in blocks) of a file. It is pulled in from and written to the file's directory entry.

Note:

Name: fileTrScTab

Formerly:

Address: C64: $8300 C128: $8300 Apple: NA

Size: 256 Bytes

Default: $00

Saved?: No

Description: For Commodore, it is the buffer used to hold the track and

sector chain for a file of maximum size of 32258 bytes.

Note:

Name: fileWritten

Formerly:

Address: C64: $8498 C128: $8498 Apple: $F61A

Size: Byte

Default: None

Saved?: No

Description: Flag indicating if a if the currently open file has been written to since the last update of its index table and the BAM.

Note:

Name: firstBoot

Formerly:

Address: C64: $88c5 C128: $88c5 Apple: $0281

Size: Byte

Default: See below

Saved?: No

Description: This flag is changed from 0 to $FF when the deskTop comes up after booting.

Note:

Name: fontData

Formerly: saveFontTab

Address: C64: $850C C128: $850c Apple: NA

Size: 9 bytes

Default: None

Saved?: No

Description: Buffer for saving the user active font table when going into menus.

Note:

Name: fontTable

Formerly: cardData

Address: C64: $26 C128: $26 Apple: NA

Size: 8 bytes

Default: Default font information

Saved?: Yes

Description: fontTable is a label for the beginning of variables for the current font in use. These variables are baselineOffset, curSetWidth, curHeight, curIndexTable, and cardDataPntr. For more information, see documentation on these variables.

Note:

Name: grcntrl1

Formerly:

Address: C64: $D011 C128: $D011 Apple: NA

Size: Byte

Default: $55

Saved?: No

Description: graphics control register #1, ie msb raster/ECM/BMM/DEN/RSEL/y scroll bits defined for use with above reg.
st_ecm = $40
st_bmm = $20
st_den = $10
st_25row = $08

Note:

Name: grcntrl2

Formerly:

Address: C64: $D016 C128: $D016 Apple: NA

Size: Byte

Default: $AA

Saved?: No

Description: graphics control register #2, ie: RES/MCM/CSEL/x scroll bits defined for use with above reg.
st_mcm = $10
st_40col = $08

Note:

Name: grirq

Formerly:

Address: C64: $D019 C128: $D019 Apple: NA

Size: Byte

Default: $42

Saved?: No

Description: graphics chip interrupt register

Note:

Name: grirqen

Formerly:

Address: C64: $d01a C128: $d01a Apple: NA

Size: Byte

Default: $24

Saved?: No

Description: graphics chip interrupt enable register
bit to enable raster interrupt in grirqen is bit 0
st_rasen = $01

Note:

Name: grmemptr

Formerly:

Address: C64: $d018 C128: $d018 Apple: NA

Size: Byte

Default: $99

Saved?: No

Description: graphics memory pointer VM13-VM10|CB13-CB11. ie video matrix and character base.

Note:

Name: hour

Formerly:

Address: C64: $8519 C128: $8519 Apple: $F203

Size: Byte

Default: 12

Saved?: No

Description: Variable for hour

Note:

Name: iconSelFlag

Formerly:

Address: C64: $84b5 C128: $84b5 Apple: $0226

Size: Byte

Default: $00

Saved?: Yes

Description: This RAM variable contains flag bits in b7 and b6 to specify how the system should indicate icon selection to the user. If no bits are set, then the system does nothing to indicate icon selection, and the service routine is simply called.
The possible flags are:

```
ST_FLASH  = $80 ; flash the icon
ST_INVERT = $40 ; invert the selected icon
```

If ST_FLASH is set, the ST_INVERT flag is ignored and the icon flashes but is not inverted when the programmer's routine is called. If ST_INVERT is set, and ST_FLASH is CLEAR, then the icon will be inverted when the programmer's routine is called.

Note:

Name: indexBlkno

Formerly:

Address: C64: NA C128: NA Apple: $F624

Size: Word

Default: $00

Saved?: No

Description: Block number of the VLIR index table (ProDos master index

block).

Note:

Name: inputData

Formerly:

Address: C64: $8506 C128: $8506 Apple: $0247

Size: 4 bytes

Default: None

Saved?: No

Description: This is where input drivers pass device specific information to applications that want it.

Note:

Name: inputDevName

Formerly:

Address: C64: $88cb C128: $88cb Apple: $08CC

Size: Commodore - 17 bytes
Apple - 16 Bytes

Default: None

Saved?: No

Description: String that holds the name of the current input device.
e.g. COMM MOUSE for commodore mouse.

Note:

Name: inputVector

Formerly:

Address: C64: $84a5 C128: $84a5 Apple: $020A

Size: Word

Default: $00

Saved?: No

Description: Pointer to routine to call on input device change.

Note:

Name: intBotVector

Formerly: interruptBottomVector

Address: C64: $849f C128: $849f Apple: $0204

Size: Word

Default: $00

Saved?: No

Description: Vector to routine to call after the operating system interrupt code has run. This allows applications to have interrupt level routines.

Note:

Name: intTopVector

Formerly: interruptTopVector

Address: C64: $849d C128: $849d Apple: $0202

Size: Word

Default: $00

Saved?: No

Description: Vector to routine to call before operating system interrupt code is run. It allows applications to interrupt level routines.

Note:

Name: intSource

Formerly: interruptSource

Address: C64: NA C128: NA Apple: $02C6

Size: Byte

Default: None

Saved?: No

Description: Byte to indicate where interrupts are coming from on the Apple.
$80 indicates mouse card
$40 indicates interrupt management card
$00 indicates software interrupts

Note:

Name: interleave

Formerly:

Address: C64: $848c C128: $848c Apple: NA

Size: Byte

Default: $08

Saved?: No

Description: Variable used by BlkAlloc routine as the desired interleave when selecting free blocks for a disk chain.

Note:

Name: invertBuffer

Formerly:

Address: C64: NA C128: $1ced Apple: NA

Size: 80 Bytes

Default: None

Saved?: No

Description: Buffer area used to speed up the 80 column InvertLine routine. Resides in back Ram.

Note:

Name: irqvec

Formerly:

Address: C64: $0314 C128: $314 Apple: NA

Size: Word

Default: $95fd

Saved?: No

Description: irq vector.

Note:

Name: isGEOS

Formerly:

Address: C64: $848b C128: $849b Apple: NA

Size: Byte

Default: Disk dependent

Saved?: No

Description: Flag to indicate whether the current disk is a GEOS disk.

Note:

Name: keyData

Formerly:

Address: C64: $8504 C128: $8504 Apple: $0245

Size: Byte

Default: $00

Saved?: No

Description: Holds the ASCII value of the current last key that was pressed.
Used by keyboard service routines.

Note:

Name: keyVector

Formerly:

Address: C64: $84a3 C128: $84a3 Apple: $0208

Size: Word

Default: Commodore - $26A1
Apple - $41BD

Saved?: Yes

Description: Vector to routine to call on keypress

Note:

Name: leftMargin

Formerly:

Address: C64: $0035 C128: $0035 Apple: $0220

Size: Word

Default: $00

Saved?: Yes

Description: Leftmost point for writing characters. Doing a carriage return will return to this point.

Note:

Name: machineType

Formerly:

Address: C64: NA C128: NA Apple: $02C4

Size: Byte

Default: None

Saved?: No

Description: Type of Apple machine. Values are:

$00: IIe
$80: IIc or something else

Note:

Name: maxMouseSpeed

Formerly: maximumMouseSpeed

Address: C64: $8501 C128: $8501 Apple: $027D

Size: Byte

Default: $7f (127)

Saved?: No

Description: Maximum speed for mouse cursor.

Note:

Name: mcmclr0, mcmclr1

Formerly:

Address: C64: $d025, $d026
C128: $d025, $d026
Apple: NA

Size: 1 Byte each

Default: None

Saved?: No

Description: These are the variables for multi-color mode colors 0 and 1 respectively on the Commodore.

Note:

Name: menuNumber

Formerly:

Address: C64: $84b7 C128: $84b7 Apple: $0228

Size: Byte

Default: $00

Saved?: No

Description: Number of currently working menu

Note:

Name: minMouseSpeed

Formerly: minimumMouseSpeed

Address: C64: $8502 C128: $8502 Apple: $027E

Size: Byte

Default: $1e (30)

Saved?: No

Description: Minimum speed for mouse cursor.

Note:

Name: minutes

Formerly:

Address: C64: $851a C128: $851a Apple: $F204

Size: Byte

Default: $00

Saved?: No

Description: Variable for minutes for time of day clock.

Note:

Name: mob0clr, mob1clr, mob2clr, mob3clr, mob4clr, mob5clr, mob6clr, mob7clr

Formerly:

Address: C64: $d027 - $d02e
C128: $d027 - $d02e
Apple: NA

Size: 1 Byte each

Default: None

Saved?: No

Description: These are the colors of the 8 objects (sprites) on Commodore.

Note:

Name: mob0xpos, mob0ypos, mob1xpos, mob1ypos, ..., mob7xpos, mob7ypos

Formerly:

Address: C64: $D000 - $D00F
C128: $D000 - $D00F
Apple: NA ——— NOT TRUE

Size: 1 Byte each

Default: None

Saved?: No

Description: These are the x any y positions of sprites #0 to #7 respectively.

Note:

Name: mobbakcol

Formerly:
Address: C64: $d01f C128: $d01f Apple: NA
Size: Byte
Default: None
Saved?: No
Description: sprite to background collision register for Commodore.
Note:

Name: mobenble
Formerly:
Address: C64: $do15 C128: $d015 Apple: $0818
Size: Byte
Default: $01
Saved?: No
Description: sprite enable bits
Note:

Name: mobmcm
Formerly:
Address: C64: $d01c C128: $d01c Apple: NA
Size: Byte
Default: $00
Saved?: No
Description: sprite multi-color mode select
Note:

Name: mobmobcol
Formerly:
Address: C64: $d01e C128: $d01e Apple: NA
Size: Byte
Default: $00
Saved?: No
Description: object to object collision register
Note:

Name: mobprior
Formerly:
Address: C64: $d01b C128: $d01b Apple: NA
Size: Byte
Default: $00
Saved?: No

Description: object to background priority

Note:

Name: mobx2, moby2

Formerly:

Address: C64: $d01d, d017 respectively
C128: $d01d, d017 respectively
Apple: $0819, $081A

Size: 1 Byte each

Default: $00

Saved?: No

Description: Double object size in x and y respectively

Note:

Name: month

Formerly:

Address: C64: $8517 C128: $8517 Apple: $F201

Size: Byte

Default: $09

Saved?: No

Description: Holds month for time of day clock

Note:

Name: mouseAccel

Formerly: mouseAcceleration

Address: C64: $8503 C128: $8503 Apple: $027F

Size: Byte

Default: $75 (127)

Saved?: No

Description: Acceleration of mouse cursor

Note:

Name: mouseBottom

Formerly:

Address: C64: $84b9 C128: $84b9 Apple: $0058

Size: Byte

Default: C64 - 199
Apple - 191

Saved?: Yes

Description: Bottom most position for mouse cursor. Normally set to bottom of the screen.

Note:

Name: mouseFaultVec

Formerly: mouseFaultVector

Address: C64: $84a7 C128: $84a7 Apple: $020C

Size: Word

Default: System handling routine

Saved?: Yes

Description: Vector to routine to call when mouse goes outside region defined for mouse position or when mouse goes off of a menu.

Note:

Name: mouseLeft

Formerly:

Address: C64: $84ba C128: $84ba Apple: $0059

Size: Word

Default: $00

Saved?: Yes

Description: Left most position for mouse

Note:

Name: mouseOn

Formerly:

Address: C64: $30 C128: $30 Apple: $005D

Size: Byte

Default: $E0

Saved?: Yes

Description: Flag indicating that the mouse is mode is on. Bit usage and constants for accessing them are as follows

Bit	Mode	Constant
b7:	mouse on if set	SET_MOUSEON = %10000000
b6:	menus on if set	SET_MENUON = %01000000
b5:	icons on if set	SET_ICONSON = %00100000

b4 - b0 not used

Note:

Name: mousePicData

Formerly:

Address: C64: $84c1 C128: $84c1 Apple: NA

Size: 64 bytes

Default: mouse pointer picture

Saved?: No

Description: 64 byte array for the mouse sprite picture on Commodore

Note:

Name: mouseRight

Formerly:

Address: C64: $84bc C128: $84bc Apple: $005B

Size: Word

Default: C64 - 319
C128 - 639
Apple - 559

Saved?: Yes

Description: Right most position for mouse.

Note:

Name: mouseSave

Formerly:

Address: C64: NA C128: $1b55 Apple: $EE53

Size: 24 Bytes

Default: None

Saved?: No

Description: Screen data for what is beneath mouse soft sprite. Resides in back Ram.

Note:

Name: mouseTop

Formerly:

Address: C64: $84b8 C128: $84b8 Apple: $0057

Size: Byte

Default: $00

Saved?: Yes

Description: Top most position for mouse

Note:

Name: mouseVector

Formerly:

Address: C64: $84a1 C128: $84a1 Apple: $0206

Size: Word

Default: System routine for icons, menus, etc.

Saved?: Yes

Description: Routine to call on a mouse key press

Note:

Name: mouseXPos

Formerly: mouseXPosition

Address: C64: $3a C128: $3a Apple: $0241

Size: Word

Default: None

Saved?: No

Description: Mouse X position

Note:

Name: mouseYPos

Formerly: mouseYPosition

Address: C64: $3c C128: $3c Apple: $0243

Size: Byte

Default: None

Saved?: No

Description: Mouse Y position

Note:

Name: mouseXOffset

Formerly:

Address: C64: NA C128: NA Apple: $0229

Size: Word

Default: $00

Saved?: No

Description: Offset from mouseXPos to draw mouse sprite. From -$8000 to $7fff

Note:

Name: mouseYOffset

Formerly:

Address: C64: $ C128: $ Apple: $022B

Size: Byte

Default: $00

Saved?: No

Description: Offset from mouseYPos to draw the mouse sprite. From -128 to 127

Note:

Name: msbxpos

Formerly:

Address: C64: $d010 C128: $d010 Apple: NA

Size: Byte

Default:

Saved?:

Description:

Note:

Name: msePicPtr

Formerly: mousePicture

Address: C64: $31 C128: $31 Apple: NA

Size: Word

Default: $84c1

Saved?: Yes

Description: pointer to the mouse graphics data

Note:

Name: nationality

Formerly:

Address: C64: $c010 C128: $c010 Apple: $E00D

Size: Byte

Default: 0 - American

Saved?: No

Description: Byte to hold nationality of Kernal. Values are:
0 American
1 German
2 French (France & Belgium)
3 Dutch
4 Italian
5 Swiss (Switzerland)
6 Spanish
7 Portugese
8 Finnish (Finland)
9 UK
0 Norwegian (Norway)
11 Danish (Denmark)
12 for Swedish

Note:

Name: nmivec

Formerly:

Address: C64: $0318 C128: $318 Apple: NA

Size: Word

Default: $90c9

Saved?: No

Description: NMI vector

Note:

Name: noEraseSprites

Formerly:

Address: C64: NA C128: NA Apple: $0240

Size: Byte

Default: FALSE = $00

Saved?: No

Description: Flag to stop routine TempHideMouse from erasing sprites #2 -

#7. Set bit #7 (set to $80) to signal not to erase sprites.

Note:

Name: numDrives

Formerly:

Address: C64: $848d C128: $848d Apple: $F60E

Size: Byte

Default: Drive dependent

Saved?: No

Description: Set to number of drives on the system

Note:

Name: obj0Pointer, obj1Pointer, obj2Pointer, obj3Pointer, obj4Pointer, obj5Pointer, obj6Pointer, obj7Pointer

Formerly:

Address: C64: $8ff8 - $8fff
C128: $8ff8 - $8fff
Apple: NA

Size: 1 Byte each

Default: obj0Pointer is set to mouse cursor picture

Saved?: No

Description: Pointers to the picture data for sprites

Note:

Name: offFlag

Formerly:

Address: C64: NA C128: NA Apple: $027C

Size: Byte

Default: FALSE = $00

Saved?: No

Description: Flag telling whether the mouse needs to be redrawn. It is used as follows:

Value	
$80	mouse is erased; do not redraw it during interrupt
$40	mouse is erased; OK to redraw it next interrupt
$00	normal: mouse is displayed. (Must be erased before drawn at a new position.)
$ff	(negative/set to TRUE anytime TempHideMouse is called)

Note:

Name: otherPressVec

Formerly: otherPressVector

Address: C64: $84a9 C128: $84a9 Apple: $020E

Size: Word

Default: $00

Saved?: Yes

Description: Vector to routine that is called when the mouse button is pressed and it is not on either a menu or an icon.

Note:

Name: pressFlag

Formerly:

Address: C64: $39 C128: $39 Apple: $08FE

Size: Byte

Default: $00

Saved?: No

Description: Flag to indicate that a new key has been pressed. Bit usage:

b7: key data is new
b6: disk data is new
b5: mouse data is new

Note:

Name: PrntFilename

Formerly:

Address: C64: $8465 C128: $8465 Apple: $08AC

Size: Commodore - 17 bytes

Default: None

Saved?: No

Description: Name of the current printer driver

Note:

Name: PrntDiskName

Formerly:

Address: C64: $8476 C128: $8476 Apple: NA

Size: Commodore - 18 bytes

Default: None

Saved?: No

Description: Disk name that current printer driver is on

Note:

Name: ramBase

Formerly:

Address: C64: $88c7 C128: $88c7 Apple: NA

Size: 4 bytes

Default: None

Saved?: No

Description: RAM bank for each disk drive to use if the drive type is either a RAM Disk or Shadowed Drive

Note:

Name: ramExpSize

Formerly:

Address: C64: $88c3 C128: $88c3 Apple: NA

Size: Byte

Default: RAM drive dependent

Saved?: No

Description: Byte for number or ram banks available in Ram expansion unit.

Note:

Name: random

Formerly:

Address: C64: $850a C128: $850a Apple: $024C

Size: Word

Default: None

Saved?: No

Description: Variable incremented each interrupt to generate a random number

Note:

Name: rasreg

Formerly:

Address: C64: $d012 C128: $d012 Apple: NA

Size: Byte

Default: None

Saved?: No

Description: raster register

Note:

Name: RecoverVector

Formerly:

Address: C64: $84b1 C128: $84b1 Apple: $0216

Size: Word

Default: Points to RecoverRectangle routine

Saved?: Yes

Description: Pointer to routine that is called to recover the background behind menus and dialogue boxes. Normally this routine is RecoverRectangle, but the user can supply his own routine.

Note:

Name: returnAddress

Formerly:

Address: C64: $3d C128: $3d Apple: $0064

Size: Word

Default: None

Saved?: No

Description: Address to return to from in-line call

Note:

Name: reqXpos0, reqXpos1, reqXpos2, reqXpos3, reqXpos4, reqXpos5, reqXpos6, reqXpos7

Formerly:

Address: C64: NA C128: NA Apple: See below

Size: Two Bytes each

Default: None

Saved?: No

Description: These variables correspond to the Commodore VIC chip registers for sprite X positions. They reside at:

reqXpos0	$0800
reqXpos1	$0802
reqXpos2	$0804
reqXpos3	$0806
reqXpos4	$0808
reqXpos5	$080a
reqXpos6	$080c
reqXpos7	$080e

Note:

Name: rightMargin

Formerly:

Address: C64: $37 C128: $37 Apple: $0222

Size: Word

Default: C64 - 319
C128 - 639
Apple - 559

Saved?: Yes

Description: The rightmost point for writing characters. If an attempt is made to write past rightMargin, the routine pointed to by StringFaultVec is called.

Note:

Name: saveFontTab

Formerly:

Address: C64: $850c C128: $850c Apple: $024E

Size: Commodore - 9 bytes
Apple - 10 Bytes

Default: None

Saved?: No

Description: When a menu is selected, the users active font table is saved in this buffer.

Note:

Name: savedmoby2

Formerly:

Address: C64: $88bb C128: $88bb Apple: NA

Size: Byte

Default: None

Saved?: Yes

Description: Saved value of moby2 for context saving done when dialogue boxes and desk accessories run. Because this was left out of the original GEOS save code, it was put here so it remains compatible with desk accessories, etc. that use the size of TOT_SRAM_SAVED for how what gets saved.

Note:

Name: scr80colors

Formerly:

Address: C64: NA C128: $88bd Apple: NA

Size: Byte

Default: None ??

Saved?: No

Description: Screen colors for 80 column mode on the C128. It is a copy of reg 26 in the VDC.

Note:

Name: scr80polar

Formerly:

Address: C64: NA C128: $88bc Apple: NA

Size: Byte

Default: None ??

Saved?: No

Description: Copy of reg 24 in the VDC for the C128

Note:

Name: screencolors

Formerly:

Address: C64: $851e C128: $851e Apple: NA

Size: Byte

Default: $BF (191)

Saved?: No

Description: Default screen colors

Note:

Name: seconds

Formerly:

Address: C64: $851b C128: $851b Apple: $F205

Size: Byte

Default: $00

Saved?: No

Description: Seconds variable for the time of day clock.

Note:

Name: selectionFlash

Formerly:

Address: C64: $84b3 C128: $84b3 Apple: $0224

Size: Byte

Default: SLECTION_DELAY = $0A (10)

Saved?: Yes

Description: Variable for the speed at which menu items and icons are flashed.

Note:

Name: shiftBuf

Formerly:

Address: C64: NA C128: $1b45 Apple: $0070

Size: 7 bytes

Default: None

Saved?: No

Description: Buffer for shifting/doubling sprites. Located in back Ram.

Note:

Name: shiftOutBuf

Formerly:

Address: C64: NA C128: $1b4c Apple: $0078

Size: 7 Bytes

Default: None

Saved?: No

Description: Buffer for shifting/doubling/oring sprites. Located in back Ram.

Note:

Name: sizeFlags

Formerly:

Address: C64: NA C128: $1b53 Apple: $0B1C

Size: Byte

Default: None

Saved?: No

Description: height of sprite | 9-pixel flag this is grabbed from the 64th byte of the sprite definition. The high bit is set if the sprite is only 9 pixels wide. The rest of the byte is a count of scan lines.

Note:

Name: softZeros

Formerly:

Address: C64: NA C128: $1b6d Apple: $D0E0

Size: 192 Bytes

Default: None

Saved?: No

Description: Buffer used for putting sprite bitmaps up on screen without disturbing background. Resides in back Ram.

Note:

Name: softOnes

Formerly:

Address: C64: NA C128: $1c2d Apple: $D000

Size: 192 Bytes

Default: None

Saved?: No

Description: Buffer used for putting sprite bitmaps up on screen without disturbing background. Resides in back Ram.

Note:

Name: spr0pic, spr1pic, spr2pic, spr3pic, spr4pic, spr5pic, spr6pic, spr7pic

Formerly:

Address: C64: $8a00 - 8bc0 respectively
C128: $8a00 - 8bc0 respectively
Apple: NA

Size: 64 bytes each

Default: spr0pic is used for mouse picture

Saved?: No

Description: This is where the graphics data for sprites 0 - 7 are kept on the C64.

Note:

Name: sspr1back, sspr2back, sspr3back, sspr4back, sspr5back, sspr6back, sspr7back

Formerly:

Address: C64: NA C128: See below Apple: $56D0

Size: Commodore - 294 bytes each
Apple - 336 bytes each

Default: None

Saved?: No

Description: For each of the soft sprites #1 - #7, there is a buffer in the back Ram for saving the screen behind the sprites. The mouse (sprite #0) is handled seperately. Locations for the buffers are:

Commodore:

sspr1back:	$133b
sspr2back:	$1461
sspr3back:	$1587
sspr4back:	$16ad
sspr5back:	$17d3
sspr6back:	$18f9
sspr7back:	$1a1f

Apple:

sspr1back:	$56d0
sspr2back:	$5820
sspr3back:	$5970
sspr4back:	$5ac0
sspr5back:	$5c10
sspr6back:	$5d60
sspr7back:	$5eb0

Each buffer is 7 bytes wide by 42 scanlines high (292 bytes) on the Commodore and 8 bytes wide by 42 scanlines high for Apple soft sprites. The extra byte is needed because of the Apples 7 bit screen bytes. These buffers are large enough to hold the largest possible sprite size (doubled in both x and y) and include an extra byte in width to save stuff on byte boundaries.

Name: string

Formerly:

Address: C64: $24 C128: $24 Apple: $0053

Size: Word

Default: None

Saved?: Yes

Description: Used by GEOS as a pointer to string destinations for routines such as GetString.

Note:

Name: StringFaultVec

Formerly: StringFaultVector

Address: C64: $84ab C128: $84ab Apple: $0210

Size: Word

Default: $00

Saved?: Yes

Description: Vector called when an attempt is made to write a character past rightMargin.

Note:

Name: stringX

Formerly:

Address: C64: $84be C128: $84be Apple: $022E

Size: Word

Default: None

Saved?: Yes

Description: The X position for string input

Note:

Name: stringY

Formerly:

Address: C64: $84c0 C128: $84c0 Apple: $0230

Size: Byte

Default: None

Saved?: Yes

Description: The Y position for string input

Note:

Name: sysDBData

Formerly:

Address: C64: $851d C128: $851d Apple: $0259

Size: Byte

Default: None

Saved?: NA

Description: Variable that is used internally to indicate which icon caused a return to the application (from a dialogue box). The actual data is returned to the user in r0L.

Note:

Name: sysFlgCopy

Formerly:

Address: C64: $c012 C128: $c012 Apple: NA

Size: Byte

Default: None

Saved?: No

Description: This is a copy of the sysRAMFlg that is saved here when going into basic on Commodore. See sysRAMFlg for more information.

Note:

Name: sysRAMFlg

Formerly:

Address: C64: $88c4 C128: $88c4 Apple: NA

Size: Byte

Default: None

Saved?: No

Description: If RAM expansion in, Bank 0 is reserved for the kernal's use. This byte contains flags designating its usage:

Bit 7: if 1, $0000-$78FF used by MoveData routine

Bit 6: if 1, $8300-$88FF holds disk drivers for drives A through C
Bit 5: if 1, $7900-$7DFF is loaded with GEOS ram area $8400-$88FF by ToBasic routine when going to BASIC.
Bit 4: if 1, $7E00-$82FF is loaded with reboot code by a setup AUTO-EXEC file, which is loaded by the restart code in GEOS at $C000 if this flag is set, at $6000, instead of loading GEOS_BOOT. Also, the area $8900-$FC3F is saved for the kernal for fast re-boot without system disk (depending on setup file). This area should be updated when input devices are changed (implemented in V1.3 deskTop).

Note: Commodore only

Name: totNumBlks

Formerly:

Address: C64: NA C128: NA Apple: $F608

Size: Word

Default: $118

Saved?: No

Description: Total number of blocks in current volume.

Note:

Name: turboFlags

Formerly:

Address: C64: $8492 C128: $8492 Apple: NA

Size: 4 bytes

Default: None

Saved?: No

Description: The turbo state flags for drives 8 through 11 on Commodore

Note:

Name: usedRecords

Formerly:

Address: C64: $8497 C128: $8497 Apple: $F619

Size: Byte

Default: $00

Saved?: No

Description: Holds the number of records in an open VLIR file

Note:

Name: vdcClrMode

Formerly:

Address: C64: NA C128: $88be Apple: NA

Size: Byte

Default: None ??

Saved?: No

Description: Holds the current color mode for C128 color routines.
Note:

Name: version
Formerly:
Address: C64: $c00f C128: $c00f Apple: $E00C
Size: Byte
Default: None
Saved?: No
Description: Holds byte indicating what version of GEOS is running.
Note:

Name: windowTop
Formerly:
Address: C64: $33 C128: $33 Apple: $021E
Size: Byte
Default: $00
Saved?: Yes
Description: Top line of window for text clipping
Note:

Name: year
Formerly:
Address: C64: $8516 C128: $8516 Apple: $F200
Size: Byte
Default: 86
Saved?: No
Description: Holds the year for the time of day clock.
Note: