The Hitchhiker's Guide' to GEOS

v2022

A Potpourri of Technical Programming Notes

(provided "as is" without support)

April 1988

Heavily Revised for Digital Medium 2020-2022

Preface

This document is very different from the original "The Hitchhiker's Guide to GEOS" in the way it is intended to be used. The original, of course, was designed to be held in your hands in book form. HGG was partially brought to digital form by an OCR scan of Glenn "Cenbe" Holmer's personal copy of the HGG. While this has been an excellent resource for the majority of users that did not have a paper copy, I felt there was much more that could be done with it to make it a very powerful modern tool.

"The Hitchhiker's Guide' to GEOS v2021" was designed from the ground up to live inside a good PDF reader and to be used from that reader. The key features used are Bookmarks, links and search.

Bookmarks

Bookmarks replace the traditional table of contents that is used in book form. They are always present to the reader and provide instant navigation ability to any part of the document at any time. There is a mini TOC located at the start of some chapters that aid in quickly locating an entry in a large chapter. The TOC at the start of **"Ch 20 GEOS Kernal 2.0"** also doubles as an alternative to the bookmark TOC since it provides the API in a different sort order than the TOC does.

A very good pdf reader will also give you the ability to create your own favorites in the document so you can have your own personal set of bookmarks. This makes using an often-used reference point very fast to locate and use.

Links

All API entries, Kernal variables, and macros have been fully indexed and can be clicked on to instantly go to the part of the document that defined them. Other important areas like Examples: are indexed as well. If the text is in bold, it is likely a clickable link. You can then use your readers back up command to return to where you were. This dramatically reduces the navigation "size" of the document. This replaces the traditional Paper Index. **Note**: All internal Links are simply bolded so as not to deter from the "Theme" of having a look and feel of a book from the 80's. Outside links to websites are bolded and underlined.

Search

A design goal from the start was to be able to instantly find the definition of an API entry, Kernal variable, Macro etc... This was achieved by having the name of an area that defines and describes an entry end with a colon.
This allows this to happen.

search for **PutBlock**:	This will take you to the only place in the document where **PutBlock** has a colon after the name, which is the API page that fully defines it.

If, instead, you are looking for the places **PutBlock** is referenced then:

search for **PutBlock**	Searching without the colon will give a result of all the times it appears in the document.

PDF Readers

Sumatra PDF: This is the recommend reader for HG'G v2020 on a PC. It is fast, small and portable. Handles multiple tabs and multiple windows. Perfect handling of Bookmarks and its most powerful feature for HG'G v2020 is its ability to add favorites. Its navigation also mirrors that of web browsers in the way it uses mouse buttons.

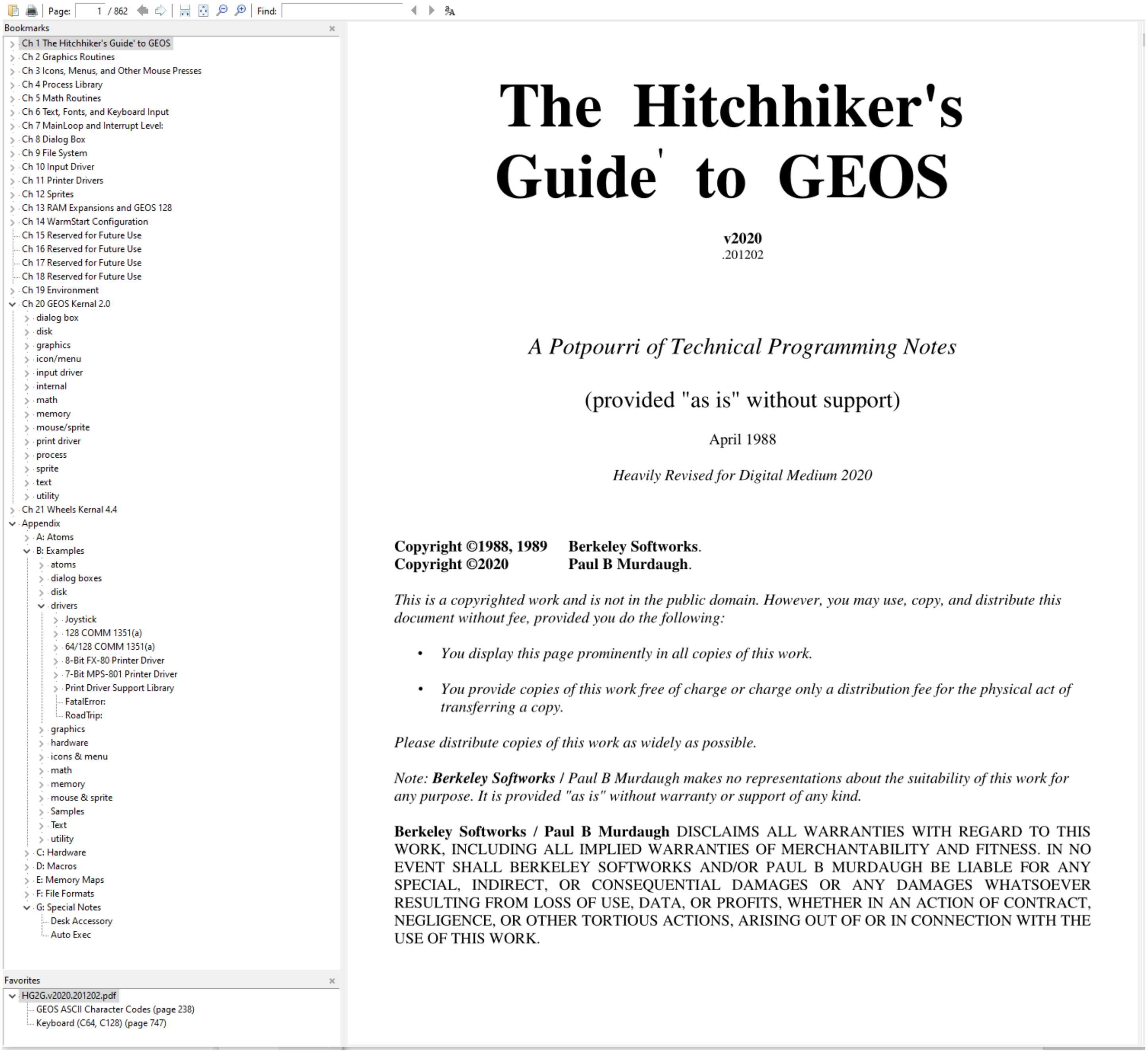
Adobe Acrobat Reader: This reader handles Bookmarks ok. You can trick it into opening the same document in multiple tabs but it does not do multi window. The basic reader also has no ability to create your own favorites into the document. Reader allows going back after using a link by using alt + left arrow.

Chromes built in reader: Better than nothing. Bookmark handling is very poor. This one is not recommended.

This is an example of what your view into this document should look like:

- Bookmarks Pane on the left. (This will be the primary method of getting around).
- Favorites Pane below that for creating your own links into the document. (Feature of Sumatra)
- Find section for being able to search.

If you do not have at least the ability to use the bookmarks this document will be very difficult to use.



The Hitchhiker's Guide' to GEOS

v2020
.201202

A Potpourri of Technical Programming Notes

(provided "as is" without support)

April 1988

Heavily Revised for Digital Medium 2020

Copyright ©1988, 1989 Berkeley Softworks.
Copyright ©2020 Paul B Murdaugh.

This is a copyrighted work and is not in the public domain. However, you may use, copy, and distribute this document without fee, provided you do the following:

- *You display this page prominently in all copies of this work.*
- *You provide copies of this work free of charge or charge only a distribution fee for the physical act of transferring a copy.*

Please distribute copies of this work as widely as possible.

Note: ***Berkeley Softworks*** */ Paul B Murdaugh makes no representations about the suitability of this work for any purpose. It is provided "as is" without warranty or support of any kind.*

Berkeley Softworks / Paul B Murdaugh DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS WORK, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL BERKELEY SOFTWORKS AND/OR PAUL B MURDAUGH BE LIABLE FOR ANY SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE, OR OTHER TORTIOUS ACTIONS, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS WORK.

Introduction to v2020

v2020 has reached its Living Document Stage. This will always be titled "The Hitchhiker's Guide' to GEOS v2020". It will have a changing sub-version in the document that will be a simple date code: YYMMDD. This date code will be applied to the end of the filename as well for easily identifying the version. Anyone hosting this file may have the code on the file or just in the description of the file.

The Goal of this Document is to provide a one stop resource for GEOS programming information. This document is comprised of the following:

1. 100% Converted to Fully Indexed Digital Form:
   The Hitchhiker's Guide to GEOS (HGG)
   by Berkeley Softworks 1988

   ***Note**: all Apple Information has been removed from this conversion. If I get geoAssembler ported into the Apple GEOS, there will be another document made from this one with all the Apple information in it. Until then, the lack of development tools for Apple led to an early death of GEOS in that environment and its inclusion here is of no value to a CBM GEOS developer.*

2. All sections from OGPRG that were not covered by the HGG were assimilated.
   The Official GEOS Programmer's Reference Guide (OGPRG)
   by Berkeley Softworks 1987

3. Information not available from the above sources has been added and noted with superscript from the following sources.
   A. GEOS Programmer's Reference Guide (GPRG)
      by Alexander Donald Boyce 1986
      Revised by Bo Zimmerman 1997

   B. Information now available from the disassembled GEOS Kernal.

   C. Information obtained from my disassembly of GEOS applications.

4. Included API Information for Wheels 4.4.
   This section is still very much a work in progress and will grow and improve over time.
   ***Note**: Thanks to "THE" email chain collected by* Bo Zimmerman, *there is some original author source for documentation. In addition, more information will be extracted from the disassembled sources of both the Wheels Kernal and of Wheels applications.*

TODO for future versions:

1. Add Tutorials for at least the following:
   a. creating Auto-Exec applications. With all of the special restrictions outlined.
   b. creating Desk Accessories. With all of the special restrictions outlined.
   c. creating VLIR applications. With fully functioning Module Management outlined.
2. Continue manually going through Wheels Kernal code to provide documentation on the remaining Kernal additions that have not been documented yet.

Comments, suggestions and error corrections are welcome. They can be emailed to:
Paul B Murdaugh - **paulbmurdaugh@gmail.com**
Writer of Dual Top and the Landmark Series for GEOS.

Sources

	Hitchhiker's Guide to GEOS 1988	Base Source Document
1	GEOS Programmer's Reference Guide 1986/1997	Secondary documentation source. Notes with a superscript 1 (^{1}) are from this document.
2	The Official GEOS™ Programmer's Reference Guide	Adding Content that is not already covered in HGG and GPRG
3	Paul B Murdaugh (author). Personal experience, Information learned from the rewrite / upgrades of geoProgrammer applications combined with discoveries from reverse engineering Berkeley applications.	Additional Content /changes made by me get a (^{3}). Example note^3: Also, any content related to geoProgrammer' 2.1 is my original content.
4	Scott Hutter. Data Mining	Found great extinct resource on additional Wheels documentation from 2002ish era. All of Kernal Group 0 was originally documented from his findings and then greatly expanded upon and corrected as needed. All the information was validated using live transactions in geoDebugger.
5	The Official GEOS™ Programmer's Reference Guide - Italian Version	Contains over 100 new pages that the English version does not have. Much of that content has been processed and assimilated.
	Transactor vol 9 issue 5 (a great Canadian magazine :))	Inspiration for **Quick Reference** at the end of the Interrupt Main chapter. The construction of this was made by the same people that made the HGG. It is just in 6502 pseudo code and fits on one page. What resulted was a blending of information from HGG and other information from the actual Kernal and using the single page layout from Transactor.
	MAPPING THE COMMODORE 128 ISBN 0-87455-060-2	Excellent documentation of the hardware in the C128.
	Additional Sources to be added as used	

Contributors

1	Bruce Thomas	Very extensive proofreading and provided much needed feedback. Provided technical input and layout ideas for cover page of Chapter 20.
2	Glenn "Cenbe" Holmer	Extensive proofreading and provided much needed feedback. Provided record formats for Text and Photo Albums V2.1.
3	Scott Hutter	Data Mining and proofreading. Assisted with Keyboard Special Key testing.
4	Bo Zimmerman	GEOS Programmer's Reference Guide 1997. Wheels documentation collection via email with Wheels author.
5	Dave Lee	Corrected error in StartMouseMode API entry.
6	Facebook group "GEOS - Wheels - GeoWorks - MegaPatch – gateWay"	General Feedback and a place for me to distribute this document.
	Additional Contributors to be added as needed	

Table of Contents

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 only 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 Publishers 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 translate 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 the full the 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 services for placing lines, rectangles, and images on the screen, as well as handling menus, icons, and dialog boxes. 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 use 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 symbols, labels, 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 runnable GEOS application file.

- Accepts a link command file created with geoWrite.
- Creates all GEOS application 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.

geoProgrammer' 2.1

geoProgrammer' is a ground-up upgrade to geoProgrammer, releasing its full potential for GEOS development.

The geoProgrammer' development system consists of three major components and support components:

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

- All known documented bugs from 1.1 have been fixed. Plus dozens of other undocumented bugs.
- Macro capacity increased from 30 to 200.
- No more hidden errors.
- Full four drive support with smart search for .include files.
- Shortcut support for double click from the desktop assembling.
- Command line support for use with scripting.
- Label names can now start with an underscore character. **_**
- Features in development.
    - Completely new file dialog box with multi select / sorting and filtering.
    - Make application with make file support for mass building large projects.
    - 128 only version with even more macro and symbol capacity.

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

- All known documented bugs from 1.1 have been fixed. Plus, dozens of other undocumented bugs.
- Up to 3x faster link times over geoLinker V1.1.
- Symbol capacity for all seq applications has increased from 966 to 2000 / (2039 when in 80 Col mode).
- Symbol capacity for all VLIR overlays increased from 764 to 767 (806 when in 80 Col mode).
- Symbol capacity for all VLIR base modules is now dynamic with the ability to use part of the overlay area as symbols for the base module. Dedicated VLIR base symbols increased from 966 to 1233 with a dynamic capacity up to 2000. This allows for very large VLIR base modules that only use small VLIR overlays.
- Maximum .rel files per module increased from 10 to 20 to further encourage the use of library (.rel) files.
- New report output types: GEOS Symbol file format and Vice debugger label file format.
- Full four drive support with smart search for include files.
- Shortcut support for double click from the desktop linking.

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

- Utility program to set the keyboard repeat delay so the debugger can be used in high clock rate environments.
- Future plans
    - Greatly increased symbol capacity via reservation of an additional REU bank.
    - Greatly increased macro capacity via reservation of an additional REU bank.
    - Last three commands in history instead of just one.
    - 80 Column support on the 128 that will have a full 80 character wide display in the debugger.

geoMake uses a make file created in geoWrite as a build guide for intelligently building projects with a single action.

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, 1571 or 1581 disk drive.
- RAM-Expansion Unit. Commodore 17xx series, GEORAM or CMD RAMLink.
- 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, 1571 or 1581 disk drive.
- RAM-Expansion Unit. Commodore 17xx series, GEORAM or CMD RAMLink.
- 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 a 128 version of geoPaint.
- geoProgrammer for the Commodore 128.

Vice Emulator

With the Vice emulator you can develop applications for the Commodore 64 and Commodore 128. Vice provides many advantages over native development platforms. Vice is very actively being developed with improvements being made all the time. The following is recommended for developing under this environment:

- Linux OS / Windows OS / MacOS for the host OS.
- GEOS 2.0 disk images. Recommended for testing. Can also be used as primary development environment
- Wheels 4.4 disk images. Recommended for primary development environment.
- Reliable drive emulation is limited to 1541/1571/1581 floppy drives and 1541/1571/1581 ram drives.
- Wheels also provides native REU RAM drive support and can create RAM drives up to 16mb.
- Recommended development setup in GEOS 2.0 or Wheels 4.4.
- Drive A 1581. Booting from this drive. Then being the source for moving project files to/from REU
- Drive B RAM 1581. This drive will hold the current project files.
- Drive C RAM 1581. Holds geoAssembler/geoLinker/geoDebugger/ OS includes and applications / fonts.
- Drive D 1581 or 1571 for secondary path for bringing data in and out of REU development area.
- Dual Top for the development desktop.
- geoProgrammer' 2.1 for Commodore 64 and 128.

Basic GEOS

Introduction

Welcome to programming under GEOS. If you are already a Commodore 64 (C64) programmer you will find your transition to GEOS to be smooth. If you are new to programming the C64, you will find that you'll progress quickly because GEOS takes care of many of the difficult details of programming, and lets you concentrate on your design.

This reference guide assumes a knowledge of assembly language programming, and a general familiarity with the C64 computer. A good assembly language programming book on the 6502 chip and a copy of the Commodore 64 Programmer's Reference Guide are good references to have handy.

GEOS stands for Graphic Environment Operating System and, as its name implies, GEOS uses graphic elements to provide a simple user interface and operating system. The philosophy of GEOS is to handle in a simple way much of the dirty work an application might otherwise have to perform: the disk handling, the bit-mapped screen manipulation, the menus, the icons, the dialog boxes, and printer and input device support.

Programmers who take full advantage of the features GEOS has to offer should be able to cut development time significantly and increase the quality of their applications at the same time. Many of these features, such as proportionally spaced fonts, or a disk turbo, would not make sense for programmers to design into each application. With GEOS, these features are provided. In the time it takes to write simple text routines one can be using proportionally spaced fonts, menus, icons, and dialog boxes to provide a sharp, intuitive, and general user interface.

Using GEOS's menus, window, and other graphic features makes applications look better, and easier to use. GEOS makes it easier for the user to switch between applications, since different applications are controlled in more or less the same way.

GEOS also changes what is possible to do with the C64. Having a built-in diskTurbo system makes possible applications which are much more data intensive. Database and other applications may incorporate much larger amounts of data. The scope of programs possible on the C64 increases.

Learning any new system is an investment in time. From the very beginning though, the amount of time and energy put into learning GEOS should pay rewards in the ease of implementing features in your program that would otherwise take much longer. The goals of GEOS are simple: greater utility and performance for the user; greater utility and simplicity for the programmer. This manual is part of our effort in achieving these goals.

Speaking the Same Language

Before we begin, a word about the notations which we'll use is in order. Within this manual we refer to constants, memory locations, variables, and routines by their symbolic names. This makes for much easier reading than trying to remember a thousand different hexadecimal addresses. A jsr **DoMenu** is much more descriptive than a jsr $C151. The actual addresses and values for the symbolic names may be found in chapter 19 "**Environment**" and chapter 20 "**GEOS Kernal 2.0**". As a convention, constants are all in upper case (TRUE, FALSE), variables begin lower case and have every following word part capitalized (**mouseXPos**, **mouseData**) and routine names have every word part capitalized (**DoMenu**). In addition to using symbolic names, we also use some simple assembler macros. For example:

```
      LoadB  variable,#value
```

is a macro for

```
      lda    #value
      sta    variable
```

A complete listing of the macros used in GEOS appears in **Appendix D**: **Macros**.

The Basics

The following features are supported by GEOS and are described in this manual:

Pull-down menus
Icons
Proportionally spaced fonts
String I/O routines using proportionally spaced fonts
Dialog boxes
Complete graphics library
Complete math library
Multitasking within applications
Fast disk access
Paged file system
Complete set of printer interfaces
Input Drivers with samples for Joystick and Mouse

GEOS is a full-fledged operating system, and its central part is the Kernal. The Kernal is a memory resident program, i.e., it is always in the C64 memory and is running all the time. It is the Kernal that contains support for all the windows, menus, icons, fonts and other features of GEOS. The deskTop, on the other hand, is not a part of the GEOS Kernal but is an application just like geoWrite and geoPaint. In fact, one could write an entirely different file manipulation "shell", as such programs are called, and throw away the deskTop altogether.

Much of the programming under GEOS consists of constructing tables to define menus and icons and specifying routines for the Kernal to call when the menus and icons are activated. It works like this:

Note: Any input the user can send to an application running under the GEOS Kernal - pulling open a menu, activating a menu, entering text, moving the mouse - is called an event. The GEOS Kernal provides the support for processing events. The application supplies a table to define the menus, icons, and other events as well as a service routine to be executed when the event is activated by user input. When the GEOS Kernal determines an event has occurred it calls the appropriate service routine. Service routines may then make use of GEOS text, graphics, disk turbo, or other routines to implement the action desired.

Applications may still have direct control over the hardware, but in many cases much of this support can be ceded to the Kernal. As an example, instead of passing a signal to the application like "the mouse was clicked", the GEOS Kernal might conclude from several mouse movements and clicks that a menu event has occurred, i.e., a menu was pulled down and a selection was made. Routines inside the GEOS Kernal called dispatchers react to user actions, whether it be a menu, icon, or other event, and call the proper user defined service routine to handle it.

In the case of our menu event above, the GEOS Kernal would reverse video flash the selected menu box and call the proper service routine provided for the activated menu selection. This type of interaction is known as event driven programming.

An event is defined as:

1. A user-initiated action.
2. A user defined time-based process.

An example of a process would be a routine which is run every second to update a clock. The application programmer provides the routine and tells the GEOS Kernal how often to run it. Every time that amount of time elapses an event is triggered. When there are no user actions taking place only the GEOS Kernal code is running. Most applications can run entirely event driven. The GEOS Kernal supports moving the mouse, and detecting whether the mouse button is clicked over an icon, a menu, or some other area on the screen. The memory location **otherPressVec** contains the address of a routine to call when the user clicks the mouse outside any menu or icon. The memory location **keyVector** contains the address of a routine to call when a key on the keyboard is hit. The application may then call a routine that returns all buffered input. In an application such as an editor, the screen represents part of a page. Clicking the mouse in the screen area has the meaning of selecting a position on the page. This position then becomes the position at which to enter text or draw graphics.

When the user clicks the mouse in the screen area (outside of menus or icons), the routine whose address is stored in **otherPressVec** is called. The routine may look at the variables **mouseXPos** and **mouseYPos** to determine the position of the mouse. When a key, or keys are hit, the routine in **keyVector** is called and the application may then call **GetNextChar** to return the characters entered by the user. **otherPressVec** and **keyVector** are initialized to 0 indicating there are no routines to call. The application's initialization code should set these vectors to the address of appropriate routines or leave them 0 if no service routine is being provided.

Double Clicks through otherPressVec

Double clicking is clicking the mouse button quickly twice in succession. The reader is already familiar with double clicking an application's file icon on the deskTop to cause the application to be run. Here we discuss double clicking through **otherPressVec**. Double clicking on an icon is discussed in the icon chapter.

The GEOS Kernal supports a variable called **dblClickCount**. To support a double click we do the following. The first time the mouse is clicked over the screen area, the **otherPressVec** routine is dispatched. As part of the service routine we check the value of **dblClickCount** and if it is 0, load it with the constant CLICK_COUNT (30). Our service routine then does anything else it needs to do to service a single click, and return. Every interrupt, **dblClickCount** is decremented if it is not already 0. If the screen area is clicked on again before **dblClickCount** has reached 0, then our service routine will know that this is the second of two clicks and may take the appropriate action.

Example: **OPVector**

Together with **otherPressVec** and **keyVector**, the menu and icon service routines provide the tools to design most simple applications. To provide even more flexibility, the GEOS Kernal makes provisions for running non-event routines for applications needing them. These will be described later.

Getting Started

The first thing an application should do when run from the GEOS deskTop is to define its menus, icons, and indicate the service routines to call for keyboard input and mouse presses. It should also clear the screen and draw any graphic shapes it needs to set up the general screen appearance.

Note: When a user double clicks on an application's icon from the deskTop, the GEOS Kernal will initialize the system to a default state, load the application, and perform a jsr to the application's initialization routine. The address of the initialization routine is specified in the application's File Header block, which we'll describe later. The initialization routine contains data tables for defining the menus, icons, and other events, and calls GEOS routines for reading the tables and setting up the events. It also draws the initial screen. Upon completion, the initialization routine returns to the GEOS Kernal. The main program loop in the GEOS Kernal will now be running and will be ready to handle menu selections, icon presses or any other event defined by the application.

When any event is triggered, the GEOS Kernal calls the service routine specified by the application. Just as the initialization routine did, each service routine executes and returns to the GEOS Kernal.

Summary

Several important points have been covered in this section. To summarize, the GEOS Kernal is an operating system which shares the memory space of the C64 with an application and is running all the time. The GEOS Kernal handles much of the low-level hardware interaction. When an event occurs, such as the keyboard being pressed, or a menu being selected, the GEOS Kernal calls the proper application service routine as specified in the application's initialization code. The application service routine processes the event, possibly calling upon GEOS graphics and text support routines, and eventually returns to the GEOS Kernal. The GEOS Kernal is then ready to process the next event and dispatch the proper service routine.

When the application's icon is double clicked by the user, the GEOS Kernal loads the application, initializes the system to a default state, and calls the application's initialization routine. The initialization routine provides the necessary tables and calls the proper GEOS Kernal routines for setting up the application's events. It also draws the initial screen.

In this manual we explain exactly how all this is done and show examples of menus, icons, and text input in a small sample application. Used in this capacity an application may be easily prototyped in a week. To give a more intuitive idea of how the GEOS Kernal works, we describe its overall structure in the next section.

The GEOS Kernal Structure

There are two levels of code running within the GEOS Kernal, **MainLoop** and **InterruptLevel**.

MainLoop

The GEOS Kernal **MainLoop** is just one long loop of code. It checks for events and dispatches the proper application service routine. Each time it goes through its cycle, the **MainLoop** code checks for any user input and determines its significance.

- A mouse button click can signify:
  - an icon being selected,
  - a menu being opened,
  - an item being selected from an open menu,
  - or, outside of any menu or icon, an activation of **otherPressVec**.
- Keyboard input generates:
  - user entered text to be dealt with by an application's **keyVector** service routine, or text for a dialog box to be processed by the GEOS Kernal.
- A process timeout signifies:
  - that an application service routine should run.

Given the input, **MainLoop** decides what to do. In the case of a menu, for example, it will figure out if:

1. A submenu needs to be pulled down, e.g., the edit menu is selected and edit menu choices need to be displayed.
2. An item that triggers a service routine is being selected, e.g., "cut" under the edit submenu, then the application service routine for the menu item "cut" needs to be run.

InterruptLevel

The GEOS Kernal **InterruptLevel** code handles the 6510 IRQ interrupt which is triggered 60 times a second on NTSC systems (50 Times a second for PAL) by a raster interrupt on the C64. Every 60th of a second, the processor is stopped in its execution of **MainLoop**, and the **InterruptLevel** code is run. **InterruptLevel** completes in much less than a 60th of a second. All it does is read the hardware. Thus even if **MainLoop** takes much longer than a 60th of a second (by executing very long application service routines, for example), **InterruptLevel** will maintain a timely interaction with the hardware: Keys pressed on the keyboard or clicks of the mouse button won't be lost.

InterruptLevel saves the state of the machine and goes about interacting directly with the hardware. It buffers keyboard input, decrements the process timers (see the section on processes), moves the sprites and mouse, and detects presses of the mouse button. For example, if the mouse button is pressed, **InterruptLevel** sets a flag that is checked by **MainLoop**. **MainLoop** decides what to do depending on whether the mouse was positioned over a menu, icon, or somewhere else on screen. Thus, the first part of an event sequence always starts in **InterruptLevel**. Processes, the mouse, and the keyboard are watched by **InterruptLevel** and when changes are detected flags are set which **MainLoop** checks at least once each time through its loop. **InterruptLevel** restores the state of the machine when it exits and returns to **MainLoop**. **MainLoop** processes any changes detected in **InterruptLevel** and calls the appropriate application service routines.

Most C64 programmers are used to writing their own **MainLoop** and **InterruptLevel** code. It is important to realize that this is already done by the GEOS Kernal. The GEOS Kernal is akin to a skeleton that the programmer fleshes out. GEOS compatible applications consist of a collection of tables for defining events and service routines to handle the events. The flow of control is structured by the Kernal.

Whenever a service routine returns, it returns to **MainLoop**. Any service routine may redraw the screen, entirely reinitialize all events, new icons, menus and anything else, and safely return to the **MainLoop**. **MainLoop** will then continue where it left off, just after the call to the service routine. A menu item can be defined that causes the application to go to another "screen" with all new functions. The service routine for this menu item may erase the screen and initialize new menus and icons. When the menu item service routine returns to **MainLoop**, **MainLoop** will continue checking for events, but will be checking the newly defined ones. Usually the next event to check for is an icon press. If a menu was selected, however, **MainLoop** will skip the icon check since an icon and a menu could not have both been selected with the same press. The same is true with other event checks. During the next **MainLoop**, the new menus, icons, and other events will be checked.

Letting the GEOS Kernal do much of the dirty work and having the application define and process events, frees the applications programmer from having to reinvent the wheel every time. This approach is sufficient to program even complex applications. geoWrite, geoPaint, and the deskTop were programmed in this fashion. To make programming even easier, the GEOS Kernal provides many utility routines (graphics, text, disk) that aid application development. The following section covers how to call the GEOS Kernal routines.

Calling GEOS Kernal Routines

This section gives a brief description of how the GEOS Kernal routines are used by the programmer. This should make the following programming examples clear. The first convention adopted when we began to develop the GEOS Kernal was to set aside some variable storage in zero page (**zpage**). This was done because 6502 instructions use less space and execute quicker in **zpage**. We also made the convention that the GEOS Kernal routines would use this variable space to accept parameters, perform internal calculations, and return values. Making routines modular like this with specific input and output makes it easier to track how each routine changes memory, and also makes it easier for developers other than Berkeley Softworks to use the GEOS Kernal routines.

To this end, 32 bytes in **zpage** beginning at location 2 are set aside for use as pseudoregisters. These memory locations are divided into 16 word-length variables with the names **r0**, **r1**, **r2**, ..., **r15**. The low-byte of each **pseudoregister** may be referenced as either rN or rNL, where N is the number of the register: e.g., **r0**, **r0L**. The high-bytes may be individually referenced as rNH, e.g., **r0H**, **r1H**.

Typically, arguments to the GEOS Kernal routines are passed and returned in these pseudoregisters. This way all the GEOS Kernal routines may perform all their internal calculations with **zpage** variables. Instead of starting off trying to manage hundreds of the GEOS Kernal locations in your head, the programmer starts off with only sixteen.

The pseudoregisters are not the only way to pass parameters to the GEOS Kernal routines. Sometimes a, x, y, and even the carry flag is used for speed. There is also another way known as an in-line call. An in-line call solves the problem that when a routine is used frequently, a large number of bytes within an application can be taken up simply by the assembly language instructions that load the **pseudoregisters** for the routines with the proper values. Some frequently used routines therefore have an in-line form to save bytes. Whereas normally a routine gets its parameters from pseudoregisters, the in-line version will get its parameters from the bytes immediately following the call to the routine. For example, the in-line call to the routine to draw a rectangle is shown below:

```
jsr    i_Rectangle              ; draw a rectangle in the current system pattern
                                ; (The system patterns can be changed with the
                                ;  routine SetPattern)
.byte  0                        ; top of rectangle. possible range: 0-199
.byte  199                      ; bottom of rectangle. possible range: 0-199
.word  0                        ; left-side. possible range: 0-319
.word  319                      ; right-side. possible range: 0-319
```

Whereas the standard call looks like:

```
LoadB  r2L,#0                   ; top of rectangle. possible range: 0-199
LoadB  r2H,#199                 ; bottom of rectangle. possible range: 0-199
LoadW  r3,#0                    ; left-side. possible range: 0-319
LoadW  r4,#319                  ; right-side. possible range: 0-319
jsr    Rectangle                ; draw it
```

When an in-line routine is called, the first thing it does is to pop a word off the stack. Instead of pointing to the return address though, this word points to the parameters passed in-line after the jsr. The in-line routine picks up its parameters, loads the proper pseudoregisters with them, stuffs the correct return address back on the stack, and then enters the regular routine.

In-line routines make sense when a routine is called a large number of times with fixed values, such as **Rectangle**. A call to **i_Rectangle** to erase or set up part of an application screen within an application works well with an in-line call since the input parameters don't change. It takes fewer bytes to store parameters as .byte and .word immediately following the subroutine call and have the subroutine include the code to pick the values up than it does to include the code to load the proper pseudoregisters before each call to the routine. To be more specific, a "`LoadW r3,#0`" takes up 8 bytes whereas a "`.word 0`" takes up only two. In-line routine names always begin with an i.

Utility routines taking several fixed arguments have in-line entry points. Other routines less frequently called, or requiring only 1 or 2 parameters, do not have an in-line form.

In this section we talked about how applications call GEOS utility routines, and how the GEOS Kernal calls user routines in response to events. We covered **MainLoop**, and Interrupt Level code within the GEOS Kernal and what each is responsible for. In the next section we cover how an application may include its own code directly within **InterruptLevel** or **MainLoop**. Generally, this is not recommended, but in some circumstances, like supporting special external hardware, it may be required. When this is necessary, the application can load special vectors provided in system RAM that allow the addition of code to **InterruptLevel** or **MainLoop**. Most programmers may skip the next paragraph on non-event code. A good rule of thumb is to avoid altering **MainLoop** or **InterruptLevel** code. In particular, an application specific interrupt routine can lead to difficult to fix synchronization bugs between **MainLoop** and **InterruptLevel** code.

Non-Event Code

Most applications will never need non-event driven code. This is code that needs to run every interrupt or every **MainLoop** regardless of what the user is doing and also cannot be set up as a process. The only cause for this is supporting a special hardware device. The programmer who needs to run non-event triggered code may do so by altering certain system vectors provided for that purpose. The vectors for adding interrupt and **MainLoop** code are **intTopVector**, **intBotVector**, and **appMain**. If an application has interrupt code it wants executed before the GEOS Kernal Interrupt Level code, it can alter the address contained in **intTopVector**. An indirect jump is performed through **intTopVector** which normally contains the address of **InterruptMain**.

Putting the address of an application routine here will cause it to be run at the beginning of each interrupt. The end of the application's interrupt routine should contain a jmp to **InterruptMain**. Similarly, to execute code after normal the GEOS Kernal Interrupt Code has run, alter **intBotVector**. At the end of **InterruptMain** code, the GEOS Kernal does a subroutine call to the address contained in **intBotVector** unless it is zero (its default value). Any routine executed through **intBotVector** should perform an rts, not rti upon completion.

Most programming can be accomplished through events. Additional **MainLoop** routines can be added, however, by loading **appMain** with the address of the routine to call. During each **MainLoop** a jmp indirect is made through

appMain unless it is zero (its default value). Performing an rts at the end of the routine called through **appMain** will return properly to the GEOS Kernal **MainLoop**.

Steps in Designing a GEOS Application

We can now breakdown what is involved in programming under GEOS.

Choose the events:
decide what menus, icons, etc. the application is to have. A special kind of event is a time base process which we will cover in a later chapter.

Define the events:
load the vectors or construct the tables which define the events themselves. For example, menu structures are defined with a simple table structure.

Write the routines:
construct the routines which are called by **MainLoop** to service the events you've defined.

To this point, this first section aims to provide an overview of what programming under the GEOS Kernal is like. GEOS allows an application to be very quickly prototyped because it breaks the program up into smaller easier to tackle event definition tables and event service routines. Before we begin coding the events for the application, we present a short discussion of the hardware setup used by GEOS: the graphics mode it uses, its layout in memory, and how the bank-switching registers are set.

It is actually possible to program under GEOS and not know anything about graphics modes or bank switching, so if you are new to the C64, don't worry if this next section seems difficult. It assumes you have read the Commodore 64 Programmer's Reference Guide. It is unlikely that you will need to change the standard GEOS memory map. However, you may on occasion wish to access a favorite routine in the Commodore Kernal ROM, or a floating-point routine in the BASIC ROM and then return to normal execution. The remainder of this chapter is devoted to a "physical" description of GEOS. That is, the graphics mode its programmed in, where it is located in memory, how to tell what version Kernal is running, what the hardware control registers are set to and how to alter the memory map to use Kernal or BASIC ROM routines.

Hi-Resolution Bit-Mapped Mode

GEOS uses the bit-mapped graphics mode of the C64 at a resolution of 320 by 200 pixels. In this mode, 8000 bytes (200 scanlines by 40 bytes per line) are used to display the screen. If you are unfamiliar with this mode you may want to refer to the Commodore 64 Programmer's Reference Guide (see page 121 for a general description of the hi-resolution bit-mapped graphics mode as well as pages 102 - 105 for some useful tables).

To make programming applications under the GEOS Kernal easier, another 8000-byte buffer is kept which is usually used to hold a backup copy of the screen data. Routines are provided which copy the image stored in the background buffer to the screen (foreground buffer) and vice versa. This is helpful when a menu is pulled down over the application's window, or a dialog box appears, and it writes over the data on the foreground screen. To recover what was on the screen previously, the menus and dialog boxes copy the background screen to the foreground screen thus saving the application the trouble of having to recreate the screen itself, something which sometimes is impossible.

These recovery routines are accessible from application routines as well. The geoPaint application uses these routines to "undo" graphics changes which the user decides to discard, the GEOS Kernal routines used to recover from background include, **RecoverAllMenus**, **RecoverLine**, **RecoverMenu**, and **RecoverRectangle**. These routines are explained in the graphics and Menu sections of this manual. Buffering to the background can be disabled if the application's program desires to use the area in the background buffer for some other purpose such as for expanding available code space. This is also described in the graphics section under Display Buffering.

Memory Map

The GEOS Kernal Memory Map table documents the C64 memory used by the GEOS Kernal and that which is left free for use by the application. Applications have about 22k from address $0400 - $5FFF. With special provision, applications may also expand over the background screen buffer. This opens up another 8k bringing the total to about 30k. This may seem like a limited amount of memory at first, but it is important to realize that all the menu, icon, dialog box, disk, file system, and various buffer support is included within the GEOS Kernal. This means much less work for the developer, less expensive development, shorter product cycles and it also means that the 22k to 30k left to the developer will go a lot further. The speed of the disk access routines also makes it practical to swap functional units in and out during program execution. Very large and sophisticated applications can be developed using memory overlay techniques. In fact, the new GEOS VLIR file structure as described in a later chapter is designed to facilitate loading program modules into memory as needed.

The location of application code and RAM is all that most developers will ever need to know about the GEOS Kernal memory map. RAM is provided in three separate places, plus whatever application space the programmer wants to devote to it. First, the pseudoregisters **r0 - rl5** may be used by applications. GEOS routines also use these locations. The registers used by each GEOS routine are well documented. Second, there are 4 bytes from $FB - FE in **zpage** that are unused by either BASIC or the C64 Kernal. These are used as pseudoregisters **a0** and **a1**. By passing values to utility routines in **zpage** locations and having them use these **zpage** pseudoregisters internally, a large number of bytes can be saved because **zpage** locations only generate one byte of addressing. This far outweighs the bytes wasted loading and unloading the pseudoregisters with parameters before and after each routine call.

Another **zpage** area is provided, from $70 - 7F. These are the pseudoregisters **a2 - a9**. Finally, the memory area from $7F40 - 7FFF is available for non-zpage RAM. For a complete variable layout, see the variable listings by address in "**Chapter 19 Environment**", "**Variables by Address**".

GEOS MEMORY MAP

Num. Bytes Decimal	Address Range Hexadecimal	Description
1	0000	6510 Data Direction Register
1	0001	6510 I/O register
110	0002-006F	**zpage** used by GEOS and application
16	0070-007F	**zpage** for only application, regs a2-a9
123	0080-00FA	**zpage** used by C64 Kernal & BASIC
4	00FB-00FE	**zpage** for only application, regs a0-a1
1	00FF	**zpage** Used by Kernal ROM & BASIC routines
256	0100-01FF	6510 stack
512	0200-03FF	RAM used by C64 Kernal ROM routines
23552	0400-5FFF	Application program and data
8000	6000-7F3F	Background screen RAM
192	7F40-7FFF	Application RAM
2560	8000-89FF	GEOS disk buffers and variable RAM
512	8A00-8BFF	Sprite picture data
1000	8C00-8FE7	Video color matrix
16	8FE8-8FF7	GEOS RAM
8	8FF8-8FFF	Sprite pointers
4096	9000-9FFF	Disk driver
8000	A000-BF3F	Foreground screen RAM or BASIC ROM
192	BF40-BFFF	GEOS tables
4288	C000-CFFF	4k GEOS Kernal code, always resident
4096	D000-DFFF	4k GEOS Kernal or 4k C64 I/O space
7808	E000-FE74	8k GEOS Kernal or 8k C64 Kernal ROM
378	FE80-FFF9	Input driver
6	FFFA-FFFF	6510 NMI, IRQ, and reset vectors

All I/O, screen drawing and interrupt control can and should be handled by the GEOS Kernal. The Kernal routines are extremely easy to use and take up memory space whether the application uses them or not. The following section describes in detail the hardware configuration used by the GEOS Kernal and can be skipped by most users. If, for example, you plan on supporting an I/O device which the GEOS Kernal does not (yet) support, or will be writing in BASIC instead of assembler, this material will be relevant.

GEOS Kernal Version Bytes

There are several bytes within the GEOS Kernal that identify what version GEOS is running. At location $C006 we find the string "GEOS BOOT". This string can be used to determine if the application was booted from GEOS. Developers who will not be using the GEOS Kernal routines in their applications can write over all but $C000 to C07F which are used to return the user to the deskTop after quitting the application. These bytes may be copied elsewhere and moved back to reboot GEOS.

Immediately following the "GEOS BOOT" string are two digits containing the version number. Currently these bytes may be $12 or $13 for versions 1.2 or 1.3, respectively. For GEOS Kernals version 1.3 and beyond have additional information bytes just after the version byte. First there is a language byte. Following the language byte are three bytes that are reserved for future expansion and are currently $00. As of this writing, the English, German, and Spanish (v1.2 only) have been implemented, whereas the other languages have not.
This area appears in memory as shown on the following page:

GEOS Kernal Information Bytes

```
                                        ; Kernal code starts at $C000.

BootGEOS:
       jmp o_BootGEOS                   ; Jump vector back into GEOS. If the routine o_BootGEOS
                                        ; moves in future versions of GEOS, doing a jmp to
                                        ; BootGEOS at $C000 will still work. As long as the
                                        ; space $C000 to $C02F is preserved, a jump to $C000
                                        ; will reboot GEOS.

ResetHandle:
       jmp    internal routine          ; This is a jump vector used by the internals of GEOS.


bootName:                               ; This is at $C006. This string can be used to check if an
       .byte "GEOS BOOT"                ; application was booted from GEOS.

version:
       .byte $20                        ; A hex byte containing the GEOS version number.
                                        ; The current version is 2.0.
                                        ; Wheels Current Version is 4.4 (.byte $44).

nationality:
                                        ; L_AMERICAN              = 0
                                        ; L_GERMAN                = 1
                                        ; L_FRENCH                = 2    (not implemented)
                                        ; L_DUTCH                 = 3    (not implemented)
                                        ; L_ITALIAN               = 4    (not implemented)
                                        ; L_SWEDISH               = 5    (not implemented)
                                        ; L_SPANISH               = 6    (not implemented)
                                        ; (Spanish Version Drean is V1.2. 1.2 did not have a
                                        ; nationality byte)
                                        ; L_PORTUGUESE            = 7    (not implemented)
                                        ; L_Finnish (Finland)     = 8    (not implemented)
                                        ; L_UK                    = 9    (not implemented)
                                        ; L_Norwegian (Norway)    = 10   (not implemented)

.if AMERICAN
       .byte L_AMERICAN                 ; ENGLISH
.elif GERMAN
       .byte L_GERMAN                   ; GERMAN
.else
       .byte NULL                       ; reserved for future use
.endif
```

Bank Switching and Configuring

The major part of the GEOS Kernal occupies memory from $BF40 on up. This means that the GEOS Kernal is using RAM in address space which is normally used for other purposes. The address space from D000 to DFFF is normally used as I/O space, but the C64 has RAM which can be swapped in over this area. Similarly, the C64 Kernal ROM and BASIC ROM can be bank switched out and another 8k of RAM opened up. During normal operation, all the GEOS Kernal banks are swapped in and the BASIC, C64 Kernal ROM, and I/O space are mapped out. All I/O processing is handled by the GEOS Kernal during interrupt level and the GEOS Kernal takes care of all the bank switching itself.

The selected bank is determined by the contents of location $0001 and two lines coming from the cartridge and external ROM ports. Since the GEOS Kernal runs without any ROM cartridges, the internal pull up resistors on these two cartridge lines cause them to default to high. The placement of screen RAM and the ROM character set is determined by the contents of address $D018.

Note: If your application needs to access I/O space outside of the GEOS Kernal routines, or access the C64 Kernal or BASIC ROMS, it should make use of two GEOS Kernal routines, **InitForIO** and **DoneWithIO**. These routines will take care of changing and restoring the memory map, and disabling interrupts and sprites as needed.

Memory mapping is described in the Commodore 64 Programmer's Reference Guide (pages 101 through 106 and 260 through 267). The following two tables outline the default settings which the GEOS Kernal uses.

GEOS Control Register Settings

Control Function	Memory Location	Value Stored	Description
Bank Select	**CPU_DATA** (0001)	`xxxxx000`	Selects which ROM banks to appear in the address space. GEOS swaps C64 Kernal, I/O and BASIC out.
VIC Chip Location Select	Bits 0,1 of **cia2pra (**DD00)	`xxxxxx01`	Chooses which 16k address range the Vic chip can address. GEOS selects bank 2 at $8000 - $BFFF
Screen Memory (Character Set in text mode)	Bits 1,2,3 of **grmemptr (**D018)	`xxxx100x`	Set graphic video RAM at A000 - BF3F (When switching to text mode, this is used to store the 2K character set)
Color Matrix (Character Screen in text mode)	Top 4 bits of **grmemptr (**D018)	`0011xxxx`	Together with the VIC chip bank select, determines the location of the video color matrix. GEOS uses 8C00 – 8FE7

Constants for RAM/ROM Bank Switching

```
RAM_64K            = $30              ; 64K RAM system
IO_IN              = $35              ; 60K RAM, 4K I/O space in
KRNL_BAS_IO_IN     = $37              ; both Kernal and BASIC ROMs in
KRNL_IO_IN         = $36              ; ROM Kernal and I/O space mapped in
KRNL_CH_BAS_IN     = $33              ; ROM Kernal + basic + Character ROM
```

Assembler Directives

Our development environment here at Berkeley Softworks may not be similar to yours. The assembler we use is of our own design. In the sample application presented throughout this manual then the reader will be seeing our assembler's directives and our macros. We will then try to keep the usage of macros to a minimum and will try to provide list file outputs when necessary. Below is a table listing the assembler directives or pseudo operations as they are sometimes known.

Assembler Directives Used in Examples

Type	Pseudo Op Directive	Arguments	Description
Start Relocatable Code Section:			
	.psect		Start new relocatable program section. Required after a .zsect and .ramsect section to return to program section.
Start Zero Page Section:			
	.zsect	[VALUE]	*VALUE* is a zero page address. The following zero page declarations are assembled starting at address *VALUE*. If *VALUE* is missing $00 will be used as the start address.
Start RAM Section:			
	.ramsect	[VALUE]	*VALUE* is a system RAM address. The following variable declarations are assembled starting at address *VALUE*. If *VALUE* is missing, starts a relocatable section which the linker will relocate.
Label:			
	NAME:		Assigns the current address to *NAME*.
Constants:			
	NAME=[key]VALUE		Equate *NAME* to *VALUE*, where *VALUE* is a decimal number unless preceded by a *key* character:

key	**type**	***NAME***	***VALUE***
	= decimal	DEC10	= 10
$	= hex	HEX10	= $0A
%	= binary	BIN10	= %1010
?	= octal	OCTAL8	=?10
'	= Character	CHARA	= 'A'

Type	Pseudo Op Directive	Arguments	Description
Data:			
	.byte	val1,val2,...	Allocates value number of sequential bytes.
	.word	val1,val2,...	Store val1, val2, ... in sequential 16-bit words.
	.block	VALUE	Allocates *VALUE* number of sequential bytes. In .psect this assigns $00 to each of the bytes in the block. Almost always the .block directive should only be used in .zsect and .ramsect. Using .block in .psect will needlessly increase the size of the application on disk.
.zsect			
	.block	VALUE	Allocates *VALUE* number of sequential bytes.
.ramsect			
	.block	VALUE	Allocates *VALUE* number of sequential bytes.
Conditional:			
	.if	expression	if expression is true; assemble the enclosed program code.
	.elif		ends an .if block and begins another.
	.else		begins an alternate block
	.endif		terminates .if block.

Note: When testing features such as icons and menus, it is often useful to use dummy service routines that merely execute an rts. This way menu and icon structures can be tested and verified before adding true service routines. After these events, are defined, menus will pull down and icon structures will blink even though they will merely call empty service routines. This allows the structure of the program to be tested and verified before the actual code is written.

What's to Come

In the following sections it will be assumed a basic working knowledge on getting a GEOS application started. If you need help getting to that level start with the "geoProgrammer User's Manual". This manual will get you familiar with geoProgrammer (geoAssembler, geoLinker and geoDebugger). After completing the manual, you will have the ability to build sample applications and be ready to continue on here.

The following sections provide you with all the information needed to build basic or advanced applications under GEOS. Graphics, Icons, Menus, Processes, Math Routines, Text and Keyboard, **MainLoop** and Interrupt level, and Dialog Boxes along with file handling, input and printer drivers, and sprite support are all covered in detail. Each section consists of a general explanation, with examples throughout.

In **Ch 10 Input Driver** we present tutorials on how to write input drivers and cover the various library routines. Fully working source code for both joystick and mouse drivers are located in **Appendix B**.

Compatibility of applications with GEOS 128

Generally, applications created for GEOS 64 that exploit the jump table at $C100 and delegate to the operating system the job of all low-level functions, should not encounter compatibility problems if they are run with GEOS 128. Compatibility is possible since GEOS 128 is an extension of GEOS 64, and as such it fully preserves its characteristics. The global system variables in both GEOS 128 and GEOS 64 are the same; all the Kernal routines perform the same tasks in both systems, with the only difference that GEOS 128 adds several other routines and globals. In particular, GEOS 128 is very close to the structure of GEOS V1.3, since it is able to handle RAM expansions in the same way. However, we will see what could be the reasons for any incompatibilities and how to remedy them. For now, it is sufficient to underline that in principle all the applications created following the directives of this manual, should work correctly even with GEOS 128 in 40-columns mode. At the time of writing, no applications have yet been developed for GEOS 64 that are able to take advantage of the 80-column mode offered by GEOS 128 and the clock frequency of 2MHz. This does not mean, however, that this will not be possible later on.

Any application created for GEOS 64, as we will see, can install the switch 40/80 item in the GEOS menu, thus providing the user with a double horizontal resolution screen (640 x 200). In order to not create unnecessary confusion, throughout the manual we will refer mainly to GEOS 64, and **chapter 13 RAM Expansions and GEOS 128** will discuss topics useful to the programmer who wants to create applications compatible with GEOS 128, plus other topics. In that chapter it will be assumed that the reader has already read the entire manual and is therefore aware of the fundamental characteristics of the operating system. **Chapter 19 Environment**, in addition to describing all the constants and global variables used by GEOS 64, also contains useful information for creating applications that exploit some features of GEOS 128.

GEOS V1.3+ and RAM expansions

Version 1.3 of GEOS, in the eyes of the application and the user, is basically just an extension of version 1.2. Therefore, all the routines of version 1.2, the parameters, the operational structure and the location of the global variables, are fully maintained in version 1.3. But some other capabilities have been added to the primitive structure. First of all, GEOS v1.3 is able to manage RAM expansion units (REU) and contains some routines specifically dedicated to the use of the additional RAM introduced by the expansion modules. These routines are illustrated in **chapter 13 RAM Expansions and GEOS 128** together with compatibility with GEOS 128.

RAM expansions are very useful work tools. They considerably reduce the need for disk access, allowing you to save work time and therefore speed up operations. Depending on the amount of additional memory, GEOS is able to use the expansion module to move large amounts of data in a very short time, to simulate a drive (RAM disk), to install a Shadowed Drive and to quickly reload the system without performing disk accesses, for example after running a Basic file. But the most remarkable aspect of these possible uses is that the applications are not required to know how and in what way GEOS is using the inserted RAM expansion, since its management in the aforementioned cases is entirely entrusted to the Kernal. However, applications can also use RAM expansions to perform completely different tasks, calling the appropriate routines made available by the Kernal.

In GEOS V1.3 the module dealing with disk access has been further improved. In particular, the entry point of the **FreeBlock** routine is now available in the jump table at $C100. Finally, in the new GEOS version it is now able to manage AUTO_EXEC files that are automatically executed when the system is installed.

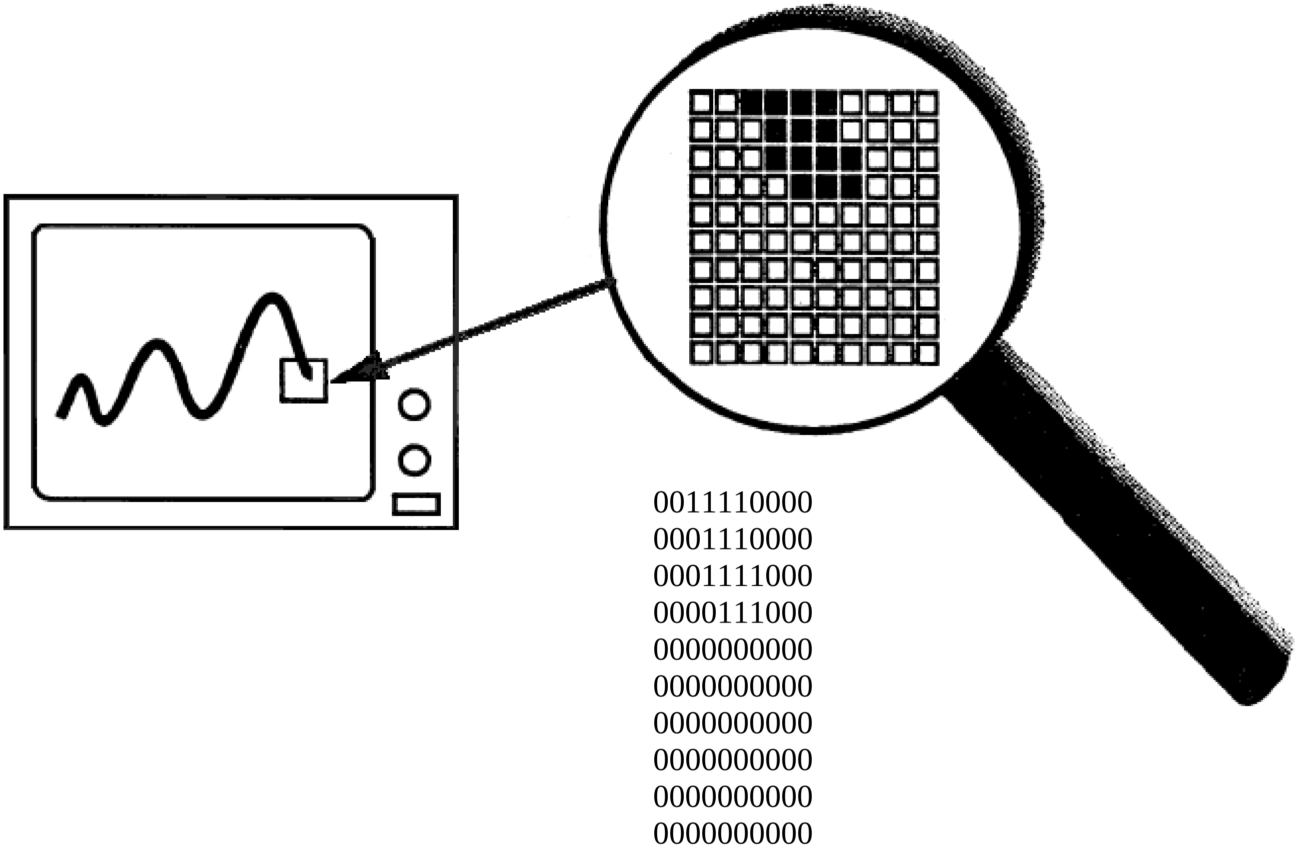
Graphics Routines

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 l'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 referred to as *bitmapped graphics*, and a *bitmap* is a picture or image created in this way.



0011110000
0001110000
0001111000
0000111000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000

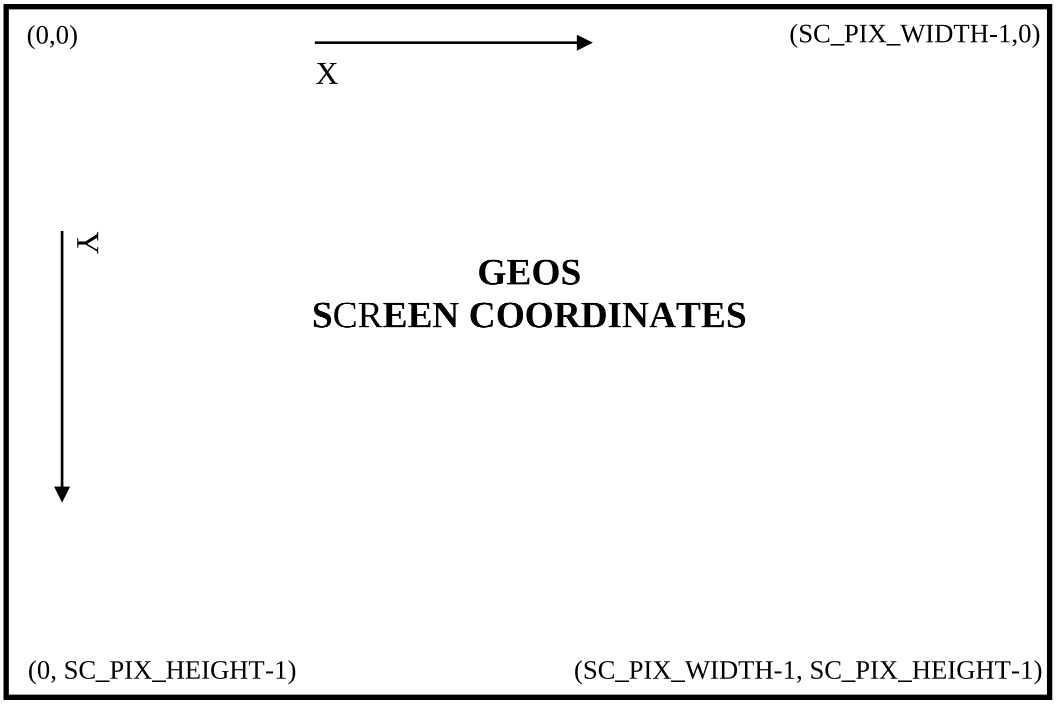
Color

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 a 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	320	Pixel width of 40-column screen.
SC_40_HEIGHT	200	Pixel height of 40-column screen.
SC_80_WIDTH	640	Pixel width of 80-column screen.
SC_80_HEIGHT	200	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	640	Pixel width of 80-column screen.
SC_PIX_HEIGHT	200	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	320	Pixel width of 40-column screen.
SC_PIX_HEIGHT	200	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: **Check128**.

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.
```

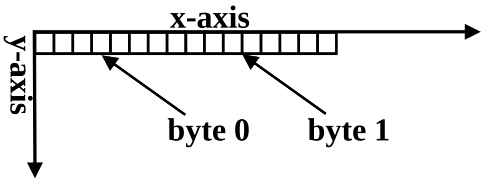
For more information, refer to "**GEOS 128 X-position and Bitmap Doubling**" in this chapter. Also see **NormalizeX** in the Routine Reference Section.

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 of shifting right three times. See **MseToCardPos** in "**Examples / graphics**".

Example: **MseToCardPos**.

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 80-column graphics screen of the Commodore 128, 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.
bit 5:	1	— Limit **GetString** text entry to foreground screen.
	0	— **GetString** text entry will use b7, b6
bit 5-0:	*reserved for future use* — should always be zeros	

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

ST_WR_FORE	use foreground.
ST_WR_BACK	use background.
ST_WRGS_FORE	**GetString** only uses foreground.

and they can be used in the 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
```

Important: 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 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: **StopMenus**

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 foreground") 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 Commodore 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.

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 calculated 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. The application's recover routine will need to compensate for the shadow box. For more information on dialog boxes, refer to Chapter 8: **"Dialog Box"**.

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 opportunity 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 3: "**Icons**, **Menus**, **and Other Mouse presses**"

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 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 Commodore 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 is 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: See **Sta80Fore**, **Lda80Fore** in Examples.

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 12 Sprites**").

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 background buffer (BACK_SCR_BASE to BACK_SCR_BASE+7999) to store the first 100 scanlines of background buffer data and the 8,000-byte foreground screen buffer (SCREEN_BASE+$40 to SCREEN_BASE+$40+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.

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.

GEOS 128 Virtual Sprites

GEOS 128 (in 80-column mode) renders 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, GEOS 128 provides 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 "**Soft Sprites**" in "**Chapter 12 Sprites**"

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.

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: **FilledRect**

Important: 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**

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.

Note[3]: A More efficient method is to build the application with all doubling in place. Then if the program detects it is on a C64 it will remove the doubling bits with a simple and #%00011111. If you are trying to add doubling instead then you have to have additional logic to handle when an ADD1_W gets applied.

Note[3]: The best correct solution has not been created yet as of this writing. If the C64 Kernal was updated to be able to use **NormalizeX** in the same way 40-column GEOS on the 128 does, then all applications could be written with no need for self-modification and would work the same on C64/C128 40/80.

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

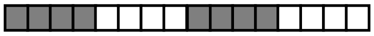
Due to the rectangular nature of bitmapped graphics, horizontal and vertical lines are inherently fast and easy to create and manipulate. GEOS provides four routines for working with horizontal and vertical lines:

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.
RecoverLine	Recover a horizontal line from the background buffer.

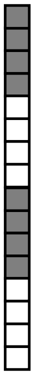
Line Patterns

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 l'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:

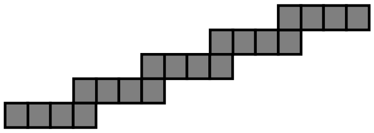


The pattern byte is always drawn as if aligned to an eight-pixel boundary. If the endpoints of a line do not coincide with eight-pixel boundaries, then bits are masked off the appropriate ends. The effect of this is that a pattern is always aligned to specific pixels, regardless of the endpoints and 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

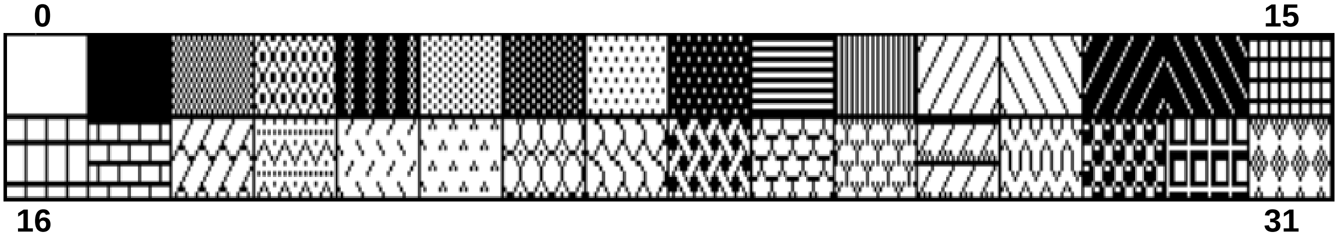
GEOS uses two types of patterns: line patterns and fill patterns. A line pattern is a one-byte repeating pixel pattern used by routines like **HorizontalLine** and **VerticalLine**, and a fill pattern is an 8x8 pixel block represented by eight bytes in memory and used by routines like **Rectangle**.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

Pattern #2 is a 50% fill pattern and is defined by the following:

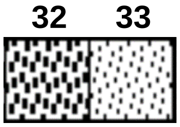
```
.byte %10101010
.byte %01010101
.byte %10101010
.byte %01010101
.byte %10101010
.byte %01010101
.byte %10101010
.byte %01010101
```

This pattern has alternating set and clear pixels. Drawing a filled rectangle in this pattern would produce a medium-dark block. (This is the default background pattern in GEOS).

All versions of the GEOS Kernal contain the following predefined patterns numbered 0-31:



There are also 2 additional patterns 32 and 33 that require extra logic to use:



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

SetPattern	Set the current pattern to a pattern between 0 and 31.

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**. 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.

Patterns 32 and 33 are accessed as an offset to pattern 31. After calling **SetPattern** with 31 in the accumulator, you will need to add the size of a pattern (8) to the **curPattern** address to use pattern 32 or add 16 to use 33.

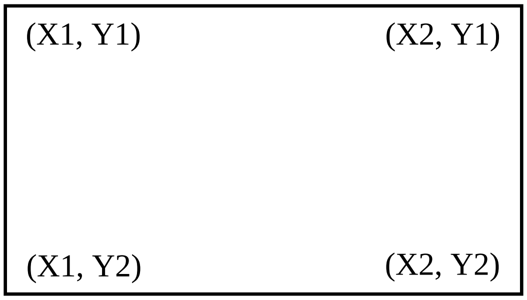
Example:

```
lda    #31
jsr    SetPattern
AddVW  #8,curPattern       ; curPattern is now pointing to Pattern 32
AddVW  #8,curPattern       ; curPattern is now pointing to Pattern 33
```

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 divisible 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.

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 the BIGCOUNT byte. The following *COUNT*-220 bytes comprise packets in ***repeat*** and ***unique*** format that should be repeated *BIGCOUNT* times. The total length of this packet is 2 bytes plus the sum of the ***repeat*** and ***unique*** packet sizes. A ***bigcount*** cannot contain 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:

COUNT	Byte
25,	**0**

repeat: the decompaction routine encounters the *COUNT* value 25, which is in the range 1-127. The following *Byte* (0), is repeated 25 (*COUNT*) times:

Output: 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

COUNT	Stream
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 5 *Stream* bytes (*COUNT*-128) are used once each:

Output: 240, 220, 10, 0, 7

Header	Packets
224, **4**,	**3**, **10**, **5**, **3**

Header: COUNT = 224 ;*Packets* size = COUNT-220
BIGCOUNT= 4 ;Number of times to process *Packets*

bigcount: the final packet begins with a *COUNT* of 224, which is in the range 221-255. *COUNT* starts a two-byte header and the following byte (4) is the *BIGCOUNT* byte. These two bytes are interpreted to mean the 4 *Packets* bytes (*COUNT*-220) are repeated four (*BIGCOUNT*) times. The *Packets* bytes are expected to be in the ***repeat*** and ***unique*** 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 (*BIGCOUNT*) times:

```
Output :   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:

Example: **BitCompact**.

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 about 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.

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.
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 **otherPressVec**, 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_NM_ICNS	byte	Total number of icons in this table. Range: 1-31
+1	OFF_IC_XMOUSE	word	Initial mouse x-position. If $0000, mouse position will not be altered.
+3	OFF_IC_YMOUSE	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 (31) 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_NM_ICNS 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.

Note: *OFF_I_NEXT=8 Offset to next icon in structure if it exists.*

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_WIDTH) 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
IESIZE       = OFF_I_NEXT              ; 8 bytes

;---   The actual icon data structure to pass to DoIcons follows:
;      Icon Table

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

;---   Icon 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 ICONSON_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**

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  NULL                      ; dummy mouse x (don't reposition)
       .byte  NULL                      ; dummy mouse y
       .word  NULL                      ; bitmap pointer to NULL (disabled)
       .byte  NULL                      ; dummy x-position
       .byte  NULL                      ; dummy y-position
       .byte  1,1                       ; dummy width and height
       .word  NULL                      ; dummy event handler
```

Alternative dummy table:

```
DummyIconTable:
       .byte    1                       ; one icon
       .block   5                       ; 3 bytes of null mouse position and a NULL bitmap pointer

       ; Scanning of the table stops once the NULL bitmap pointer is read in.
       ; The six additional table bytes are not needed.
```

Important: The maximum number of icons that can be in an icon table is 31. Attempting to use more then 31 icons will likely cause a system crash.

MainLoop and Icon Event Handlers

When the user clicks the mouse button on an active icon, GEOS **MainLoop** will use **IsMseInRegion** to 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	$00	The icon event handler is immediately called; the icon image is untouched
ST_FLASH	$80	The icon is inverted for **selectionFlash** vblanks and then reverted to its normal state before the event handler is called.
ST_INVERT	$40	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 loading **dispBufferOn** with ST_WR_FORE, and calling **InvertRectangle**. See Example: **InvertIcon**

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 (30). 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 user clicks on the icon again 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-click 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:

Function: Icon double-click handler

Description: 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
       jsr    InvertIcon                ; first push of a double-click
                                        ; so just invert the selection
       bra    90$
10$
       jsr    OpenIcon                  ; double-click detected, go process it
90$
       rts                              ; exit
```

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:

Function: Icon double-click handler.

Description: polled mouse method Open Icon.

```
IconEvent2:
       ;---   User pressed mouse once, start double-click counter going
       LoadB  dblClickCount,#CLICK_COUNT; start delay (30 tics)

10$    ;---   Loop until double-click counter times-out or button is released
       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
20$    ;      button is pressed a second time
       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)
       jmp    DoDoubleClick             ; do double-click stuff

30$    ;---   Single-click detected (no double-click)
       jmp    DoSingleClick             ; do single-click stuff
```

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 **otherPressVec**. When control vectors through **otherPressVec**, the value in **r0H** is meaningless. For more information on **otherPressVec**, refer to "**Other Mouse Presses**" in this chapter.

Other Things to Know About Icons

Icon Releases and otherPressVec

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

Example:

```
;--- otherPressVec routine that ignores releases (high bit of mouseData is set on releases)
MyOtherPress:                          ; control comes here from otherPressVec
             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                       ; exit
```

For more information on **otherPressVec**, 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_I_PIC 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_I_PIC 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).

Inverting an Icon

GEOS uses **InvertRectangle** to invert an icon that has been clicked on while the **iconSelFlag**=ST_INVERT. The registers used by **InvertRectangle** are still loaded with the icon image coordinates when the icon event handler is called.

Example:

```
      ;---   On entry, the foreground screen icon image is inverted
      ;      Pass:        r2L,r2H      top and bottom of icon
      ;                   r3,r4        left and right of icon
      ;      Called by:   IconEvent1
      InvertIcon:
             PushB  dispBufferOn              ; save current display buffer setting
             LoadB  dispBufferOn,#ST_WR_FORE  ; set to foreground only
             jsr    InvertRectangle           ; invert the selected icon using the coordinates
                                              ; passed from the GEOS icon event
             PopB   dispBufferOn              ; restore display buffer setting
             rts
```

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 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 ignored 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 "**Graphics Routines"** for more information.

Example:

```
Function:     Sample GEOS 128 icon table. Uses automatic doubling feature. Using compiler flags for
              conditional assembly between C128 and C64.

Note:         You can build applications that work on both the 128 in 80cols and the 64 at runtime.

C128          = TRUE
C64           = FALSE
.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
OFF_I_NEXT    = 8

   ;--- The actual icon data structure to pass to DoIcons follows
   ;     Icon Table
   I_header:
         .byte  NUMOFICONS
         .word  ((PAINTX*8) + (PAINTW*8/2)) | DOUBLE_W    ; position mouse over paint icon
         .byte  PAINTY + PAINTH/2

   ;---   Icon 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) / OFF_I_NEXT; number of icons in table

   ;--- Dummy icon event routines which do nothing but return
   PaintEvent:
         rts
.endif
```

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

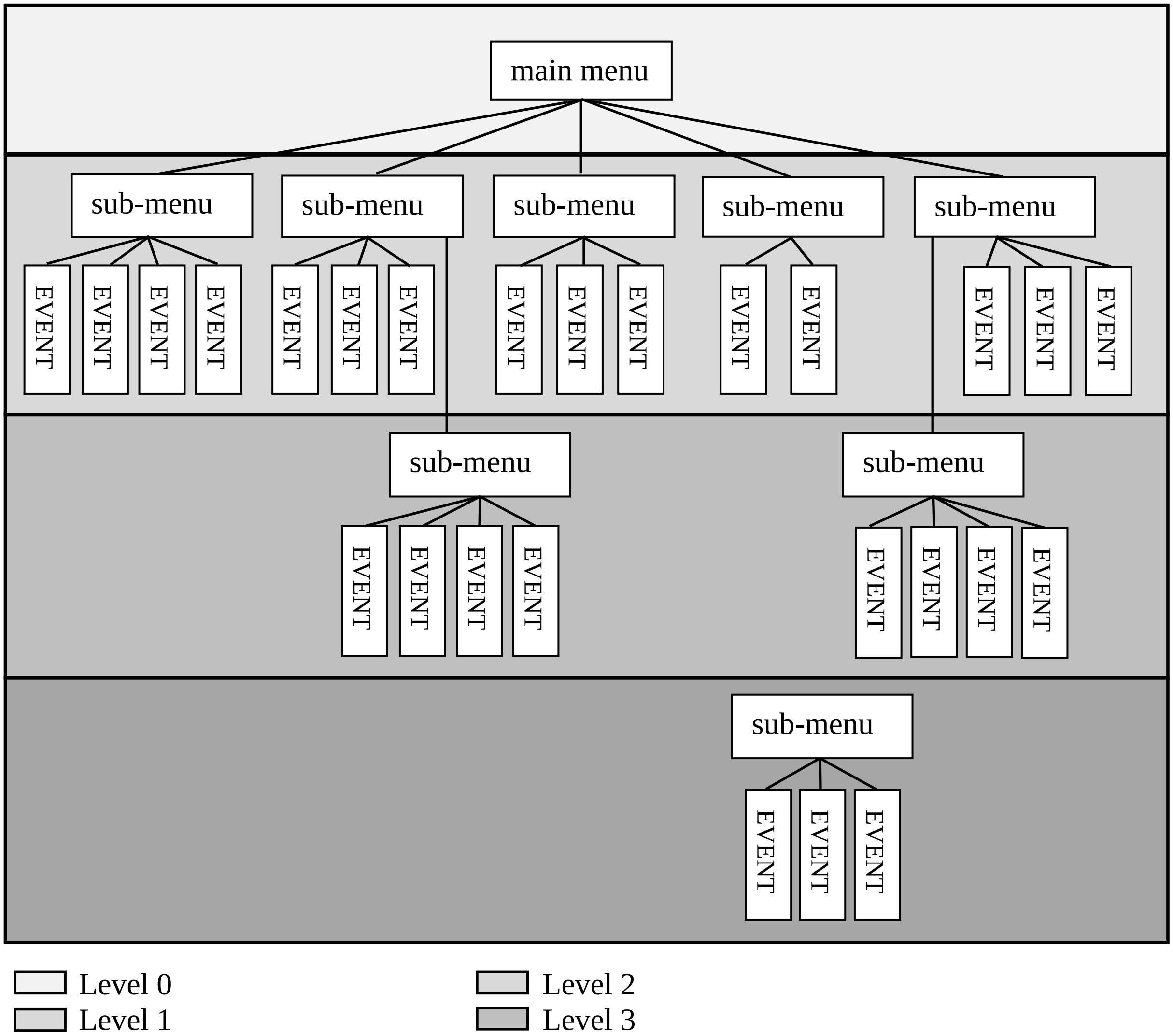
GEOS divides the labor of handling menus between itself and the application. The GEOS Kernal handles all of the user's interaction with the menus. This includes drawing the menu items, opening up necessary sub-menus, and restoring the Screen area from the background buffer when the menus are retracted. **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:

Important: Menu tables in GEOS 128 cannot use DOUBLE_W with x-coordinates. If used, the menu will draw correctly but the mouse will not be able to interact with the menu. This limitation has been corrected in Wheels.

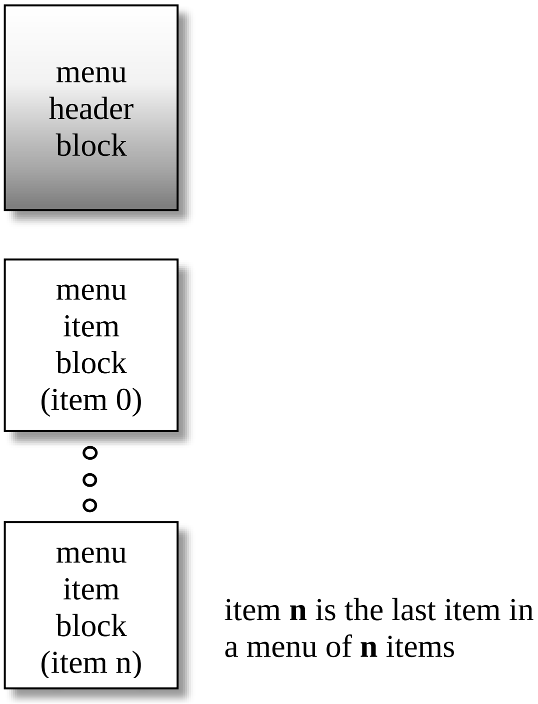


Sample Menu Tree

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 (=4), 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_MY_TOP	byte	Top edge of menu rectangle (y1 pixel position).
+1	OFF_MY_BOT	byte	Bottom edge of menu rectangle (y2 pixel position).
+2	OFF_MX_LEFT	word	Left edge of menu rectangle (x1 pixel position).
+4	OFF_MX_RIGHT	word	Right edge of menu rectangle (x2 pixel position).
+6	OFF_M_ATTRIBUTE	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 before version 2.0 and all versions of GEOS 128 (1.3 and 2.0) do not correctly handle menus that extend beyond an x-position of 255.

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 (15) selectable menu items.

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

Constant	Value	Description
HORIZONTAL	$00	Arrange menu items in this menu/sub-menu horizontally.
VERTICAL	$80	Arrange menu items in this menu/sub-menu vertically.
CONSTRAINED	$40	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	$00	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 (OFF_M_ATTRIBUTE):

7	6	5	4	3	2	1	0
b7	b6	n/a		b3-b0			

b7	orientation:	1 = VERTICAL;	0 = HORIZONTAL.
b6	constrained:	1 = CONSTRAINED;	0 = UN_CONSTRAINED.
b5-b4	not used		
b3-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 of 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 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 arc 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	Value	Description
SUB_MENU	$80	This menu item leads to a sub-menu. The *OFF_POINTER_ITEM* is a pointer to the sub-menu data structure (points to first byte of a menu/sub-menu header).
DYN_SUB_MENU	$40	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 preprocessing and return with **r0** containing a pointer to a sub-menu data structure or $0000 to ignore the selection.
MENU_ACTION	$00	This menu item generates an event. The *OFF_POINTER_ITEM* is a pointer to the event routine to call.

Bitwise Breakdown of the Item Type byte:

b7	b6	b5	b4-b0

b7	sub-menu flag.
b6	dynamic sub-menu flag.
b5-b0	*reserved for future use.*

Example Menu: mainMenu

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 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 set 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:
    1. the mouse moving outside of the rectangle defined by **mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight**.

    2. 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 to use mouse faults for its own purposes, it 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:

Example: **ResetMouse**

Other Mouse Presses

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

otherPressVec 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                            ; or return immediately to ignore releases
10$
      ;--- Handle press here
```

Because **otherPressVec** gets called on each press (and release), any double-click detection must be performed manually by the other-press routine. Handling double-clicks through **otherPressVec** is similar to the polled mouse method used with icons, the major difference being a check for releases on entry.

An alternative double click method is demonstrated in the **OPVector** example. This method does not use UI time to implement the polled mouse method; instead, it sets the **dblClickCount** on a button release and checks the count on a button press. If the count is greater then zero on a button press, then a double click has been detected.

```
      bbpl  mouseData,10$            ; check state of the mouse button
                                     ; branch to handle presses
      ;--- If double click detection is needed, set count to CLICK_COUNT (30) on release events
      LoadB dblClickCount,#CLICK_COUNT
      rts                            ; return immediately to ignore releases
10$
      ;--- Handle press here
      ;    (Check dblClickCount for double click (count > 0))
```

Example: **OPVector**

Process Library

A process 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 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 runnable. 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 (20) 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 the process becomes runnable (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 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 (50) and NTSC (60) machines and the automatic assignment of process constants with (* - procTable)/PSIZE.

Sample process data structure FRAME_RATE NTSC=60 / PAL = 50

```
procTable:
;---   MOUSE CHECK PROCESS
;      Check mouse position and change pointer form as necessary.
       .word  CheckMouse                      ; process event routine
       .word 10                               ; check every 10 vblanks
       MOUSECHECK   = (*-procTable)/PSIZE -1  ; process number

;---   REAL-TIME CLOCK PROCESS
;      Increment a real-time clock counter every second
       .word Tick                             ; process event routine
       .word FRAME_RATE                       ; one second worth of vblanks
       RTCLOCK      = (* - procTable)/PSIZE -1; process number

;---   SCREEN-SAVER PROCESS
;      Save the screen by turning off colors after five minutes.
       .word ScreenSave                       ; process event routine
       .word 5*60*FRAME_RATE                  ; frames in 5 minutes
                                              ; delay = 5 min * 60 sec/min * frames/sec)
       SCRNSAVER    = (*-procTable)/PSIZE -1  ; process number
       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, runnable). 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 to 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 screensaver 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
       ;---   disable interrupts to synchronize freezing
       php                             ; save current interrupt disable status
       sei                             ; disable interrupts
       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 old 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 becomes runnable, 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 runnable. An internal GEOS flag (called the *runnable flag*) is set, indicating to **MainLoop** that an event is pending. The timer is then restarted with its initialization value. **MainLoop** will ignore the runnable flag as long as the process is blocked. When the process is later unblocked, **MainLoop** will see the runnable 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 runnable. 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 runnable. The process may in fact already be marked as runnable and GEOS is only awaiting the next pass through **MainLoop** to generate an event (A process that is marked as runnable 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 runnable, 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.

Parameters:	x	PROCNUM — 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 runnable immediately.

EnableProcess merely sets the runnable 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 privileged status and cannot override a blocked state. However, because it doesn't depend on (or affect) the current timer value, the process can become runnable 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 discussion 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 runnable 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 runnable flag is set and it is not blocked, **MainLoop** clears the runnable flag 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 runnable flag, but since there is only one runnable 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 runnable at that time, the interrupt may occur while **MainLoop** is the midst of handling processes: processes that have already been passed-by may become runnable but not get executed until the next time through **MainLoop**, which could be a fraction of a second later.

For more Information refer to Chapter 7: "**MainLoop and Interrupt Level**".

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 performs 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:

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

Function: 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
```

Function: 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 capabilities: they can only operate directly on eight-bit quantities (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-precision (two byte) shifting, as well as multiplication and division.

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 parameter. Because the parameters are located on zero page, their addresses are one-byte quantities 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 **r0** and **r1**, then call a GEOS math routine, like **Ddiv** (double-precision divide) with the address of **r0** and **r1** in the x and y registers.

Example:

```
ldx   #r0                         ; load up address of first parameter
ldy   #r1                         ; and address of other parameter
jsr   Ddiv                        ; divide the word in r0 by the word in r1
```

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 quantities 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 D**: **Macros"**. 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:

Function: Increment Word.

Args: addr – address of word to increment.

Action: Increment word by 1. If the result is zero, then the zero flag in the status register is set.

```
.macro IncW addr
      inc    addr
      bne    done
      inc    addr+1
      done:
.endm
```

Function: Decrement zero page word.

Args: zaddr – zero page address of word to decrement.

Action: Decrement zero page word. If the result is zero, then the zero flag in the status register is set.

Destroys: a, x.

```
.macro DecZW zaddr
      ldx    #[zaddr                   ; load x with address of zp word for call
      jsr    Ddec                      ; call GEOS routine
                                       ; z flag is set if both high and low become $00
.endm
```

Function: Decrement Word by 1.

Args: addr – address of word to decrement.

Action: Fast Decrement word by 1. Useful for values that will never go to zero. IE address pointers.

Note: on return, z flag is meaningless.

Destroys: a

```
.macro DecW addr
      lda   addr                     ; get low-byte
      bne   z                        ; if zero have to do high-byte
      dec   addr+1                   ; decrement high-byte
z:
      dec   addr                     ; decrement low-byte
.endm
```

Most applications will use **IncW** and **DecZW** to take advantage of the flags which are set when the values reach zero. However, **DecW** 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**

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 a 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.

Example: **DSmult**

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 add divide-by-zero error checking by using these two wrapper routines:

Example: **NewDdiv**, **NewDSdiv**.

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 facilitates the translation of keyboard input to text output by mapping most keypresses 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 typesetter'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.

A complete list of official GEOS Font files appears in "**Appendix F**: **File Formats**" **"Official Fonts"**.

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 l/5th the width of an upper-case "W".

Character Width and Height

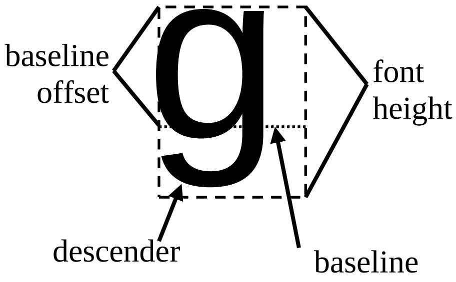
Although some characters are taller than others, all characters in a given font are treated as if they are the same height. This height is the font's point size. A 10-point font has a height of ten pixels. If a character's image is smaller than 10 pixels, it is because its definition includes white pixels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

Since 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 attributes*: 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 **currentMode**:

currentMode Bit Flags

b7		b6	b5	b4	b3	b2	b1	b0

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.	

Note: [†]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 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.

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

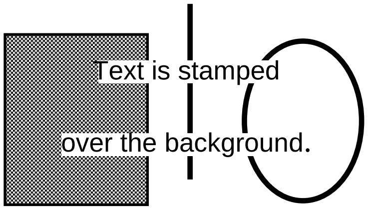
Style attributes temporarily modify the plaintext definition of the character and, in some cases, change the size and ultimate shape of the character:

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

Important: 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 the Size of a Character** " 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 **Graphics Routines**.

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 **Graphics Routines**. There is one notable difference: because **SmallPutChar** will accept negative x-positions (allowing characters to be clipped at the left 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. Character code 127 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 character code 128 that is a visual representation of the shortcut key (a Commodore symbol). 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.

Note: A complete list of GEOS character codes appears in "**Chapter 19 Environment**" **"Structures / Keyboard"**.

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 ever needs 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, it 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.

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.

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

Note: A complete list of GEOS escape codes appears in "**Chapter 19 Environment**" "**Structures / Keyboard"**.

There are many ways to handle 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:

Function: Save character as last printed and print with **PutChar**.

Args: none.

Description: Macro to replace jsr **PutChar** in your code so that lastChar holds the value of the last character printed.

```
.macro PutChar
      sta   lastChar                  ; character is already in the accumulator
      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).

Manual Character Clipping

One of the criticisms of GEOS is the inconsistent and sometimes capricious character clipping capabilities — not all versions of GEOS fully support partial character clipping and the versions that do have inherent idiosyncrasies. 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 calculation and should only be used in special cases where controlled character clipping is needed.

Example: **ClipChar**.

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 no 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 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: **Print**.

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**.*

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** 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, **PutString** 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.

PutStrFault is a partial solution to this problem. **PutStrFault** immediately terminates string printing on any fault (left or right-margin) by moving **r0** forward to point to the null. Install **PutStrFault** into **StringFaultVec** prior to using **PutString**.

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.

Example: **SmartPutString**.

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 the current set of attributes. 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,LF                    ; start in the upper-left corner and
                                         ; 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.",NULL
```

Note: A complete list of GEOS escape codes appears in "**Chapter 19 Environment**" **"Structures / Keyboard"**.

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 commands 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 uninitialized so the first graphics string command should be a MOVEPENTO.

Example:

```
TextGraphics:
       .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
```

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.

Important: 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.

GetString

GetString provides a convenient way for an application to get text input from the user without using a dialog box. **GetString** takes care of intercepting keypresses and echoing the characters to the screen. The beauty of **GetString** is that it builds the string concurrently with the rest of **MainLoop**, allowing menus, icons, and processes to remain functional while the user is typing in the string.

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

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

GetString uses the pointer in **r0** combined with the size of the buffer in **r2L** to provide a working window for the user to be able to enter/edit data in a field. This buffer can start out empty (field does not have any data yet) or it can start out with some data already in it (field has had information previously entered into it). Since **r2L** is the size of the buffer and not the size of the data already in the buffer, the value of **r2L** will be the same in both of these situations.

Example:

```
.ramsect
       rName:       .block 20+1   ; 20 character name buffer plus byte for null terminator
.psect
       tName:
              .byte  "Name: ",NULL

DisplayForm:
       jsr    DispTxtPrompts      ; call routine to PutString all text prompt labels to the form
Field1:
       LoadW  r0,#rName           ; set buffer pointer to our name buffer
       LoadB  r2L,#20                    ; set size of buffer (max characters to enter)
       LoadB  r1L,#0              ; use system fault routine
       LoadW  r11,#nameXPos       ; set x-position of text prompt to application defined value
       LoadB  r1H,#nameYPos       ; set y-position of text prompt to application defined value
       LoadW  keyVector,#Field2   ; set STRINGDONE to point control to next field after CR is entered
       jsr    GetString           ; call GetString. user input starts after we return to the
MainLoop
       ;---   do any additional desired steps prior to user having control of entry
       rts
Field2:
       ... same code structure as in Field1
```

The first time DisplayForm is called, the rName buffer is empty, so the user just has a blank prompt to enter data into. For our example, the user enters "Arthur Dent" into the "Name: " field. When the user causes this form to be displayed a second time, the rName buffer contains "Arthur Dent", NULL. When the Field1 block is executed again the user will see the "Name: " field already populated with "Arthur Dent" ("Name: Arthur Dent") and the text entry prompt will be after the name. The user now has the ability to edit the name in any way needed.

GetString and dispBufferOn

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

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

Example: **NewGetString**.

Note: Original handwritten note about the above paragraphs regarding dispBufferOn: *"Not entirely clear. Make sure people know that this is not really of that much importance"*.

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

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

Forcing End of String Input

Because **GetString** accepts input concurrently with **MainLoop** there might be some user action other than pressing [Return] that the application may want to recognize as the end of input marker. Unfortunately, there is no direct way to terminate **GetString** before the user presses [Return]. The trick of choice in this situation is to simulate a press of the return key by loading **keyData** with a CR and vectoring through **keyVector** as in:

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

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

Note: 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).

Although the word "Load" in **LoadCharSet** is misleading in that it implies it automatically loads the character set from disk into memory, the application must read the font data into memory prior to calling this routine. **LoadCharSet** expects 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**. Some sophisticated GEOS applications use a font-caching 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 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 15). 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 be 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 **GetFHdrInfo** to access the header block for each font file. The font file header block contains information pertinent to the particular font file, such as the font style ID, the available point sizes, and the amount of memory required for each point size. These values can be accessed in the header block by using the following offsets:

Constant	Offset	Field Size	Description
O_GHFONTID	$80	1 word	Font style ID. (0 – 1023)
O_GHPTSIZES	$82	15 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	$61	15 words	Size (in bytes) of each character set from smallest to largest point size. (These numbers have a one-to-one correspondence 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 15-character set ID numbers. A character set ID number is a 16-bit combination of the style ID and a point size identifier. The style ID is stored in the upper 10 bits and the point size is stored in the lower 6 bits:

Character Set ID Word:

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

b6-bl5 font style ID.
b0-b5 point size.

This combination of font style ID 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 Durant style has a style ID of 15, so the Durant 10 font would have a character set ID of:

(15 << 6) | 10 or $03CA
(15 * 64) + 10 = 970 or $03CA ; alternate method to calculate the character set ID

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

Note: GEOS font IDs were meant to be unique; in fact, Berkeley Softworks even had a font registration service to help ensure this. However, GEOS users didn't always use the service, and a complete list wasn't available unless you had Dick Estel's Font Resource Directory, which itself could get out of date.

A **web app** listing all known GEOS fonts and a PDF sample sheet including a sorted list of font IDs and names can be found here: **Lyon Labs GEOS Fonts**.

You can use these resources to explore GEOS fonts and to make sure that if you create one yourself, it will have a unique font ID.

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 2 byte index 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 (in 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 next word.
+6	1 word	Pointer to beginning of character bitstream data (relative to beginning of data structure). Bitstream data typically follows the index table.
+(8)	? 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 zero bits out to the next byte boundary. 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 (9) bytes. Whenever GEOS needs to switch fonts internally (while drawing the BSW 9 text in menus, for example), these bytes are saved off to **saveFontTab**, which is also FONTLEN bytes long. If a GEOS application needs to temporarily change fonts, it can simply duplicate this technique, saving and restoring to **fontTable** and **saveFontTab** as needed.

Keyboard Input

Many keyboard input needs can be accommodated 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 without 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.

If the shortcut key (designated by the Commodore logo on CBM 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 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 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**.

Ignoring Keys While Menus are Down

Because **MainLoop** is still running full-speed when menus are down, **keyVector** will still be 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) 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:

Example: **ShortKey**.

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 current interrupt disable status
       sei                        ; disable interrupts
       jsr    PromptOff           ; prompt = off
       LoadB  alphaFlag,#0        ; clear alpha flag
       plp                        ; restore old interrupt 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:

Example: "**Sample Keyboard Entry Routine**."

Sample Better Get String

With the routines discussed in this chapter it is possible to build a sophisticated word processor. To show how these routines fit together we can build a simple version of **GetString**. For want of a better name, let's call it **OurGetString**. It will read buffered input from the keyboard, display and update the text prompt position so that it moves ahead of the text, and echo the characters back to the screen. When we get this running, we can generalize it by adding support for reading embedded control characters. **OurGetString** can then be used as the basis for a text editor module that reads from a buffer as well as/instead of from the keyboard.

We begin by looking at **keyVector**, and **keyData**. **keyVector** contains the address of the keyboard dispatch routine. **keyData** gets the value of the key that was pressed. The **keyVector** routine gets called every time GEOS detects that a key was hit. Initially **keyVector** is set to 0 by the GEOS Kernal so all characters typed from the keyboard will be ignored. The application should load **keyVector** with the address of a routine to handle character input. In the present case this is the address of **OurGetString**.

When a key is pressed on the keyboard, the Interrupt Level code in GEOS places the ASCII value of that key in the variable **keyData**. Interrupt Level checks this every 60th of a second. During **MainLoop**, GEOS will check a flag left by Interrupt Level and if it indicates that a key has been pressed, **MainLoop** will call **OurGetString**. **OurGetString** can then get the character value out of **keyData**.

MainLoop does a little more than this though. If the application is doing a lot of processing, then it is possible that the user may have had a chance to enter two or three characters since the last call through **keyVector** to **OurGetString**. In this case, GEOS automatically buffers keyboard input. If Interrupt Level finds that another key has been pressed, and **keyVector** hasn't been serviced, it saves the character in its own internal buffer. The routine **GetNextChar** can then be call from within the keyboard dispatch routine to retrieve characters stacked up in the input buffer. Each time **GetNextChar** is called it returns the next character from the input buffer. When there are no more characters to return, **GetNextChar** returns zero.

When **OurGetString** is called, we retrieve the first character from **keyData**. We then call **GetNextChar** in a loop to return the remaining characters. Each time we get a character we store it in our own input buffer, inBuffer. As we retrieve the input characters, we will want to echo them back. This means calling **PutChar** to print it to the screen. You pass **PutChar** the character to print and an x and y-position on screen to print it at. The position can be any legal position on the screen, 0 to 319 for x, and 0 to 199 for y. **PutChar** is the same routine used by **GetString** and **PutString**.

It is also possible to use **StringFaultVec** to handle printing off screen, or outside of margins. **StringFaultVec** will get called when **PutChar** tries to print a character outside of the **leftMargin**, **rightMargin**. **PutChar** will also clip any part of a character that appears outside of **windowTop** and **windowBottom**. Clipping means that any part of a character appearing outside the top and bottom-margins will not be printed. Therefore, on the top and bottom-edges of a text window, chopped off characters may appear. This is useful for implementing scrolling where characters may be of different fonts and sizes on the same line.

StringFaultVec can be used to scroll a text window left or right or to wrap characters from the right-side of the screen to the left. In the first case, if the text window as defined on the screen by **windowTop**, **windowBottom**, **leftMargin** and **rightMargin** is used as a window overlooking a much larger document, then it is natural to want to scroll the document under the window. When a character is entered that lies outside the window, the **StringFaultVec** routine is called and may then erase the text in the window area and redraw it shifted to the left to make room for the new text on the right.

OurStringFault dispatch routine will perform a simple character text wrap. Characters typed past the end of the line will be moved to the beginning of the next. It will look at the height of the current line, add that to the vertical position of the text and use the result as the new vertical position. **leftMargin** is used as the new horizontal position. When the **OurStringFault** handler returns, it returns the same as if **PutChar** had returned. **OurGetString** will not know that **OurStringFault** was ever triggered. All it knows is that it called **PutChar** and a character was printed.

To briefly recap, **OurGetString** will prompt the user for input, display the text prompt, and get keyboard data from reading **keyData** and calling **GetNextChar**. As the characters are entered they will be echoed via **PutChar** and stored in our own internal buffer. If the end of the line is reached before the user hits return, **OurStringFault** handler will perform a character wrap.

The routine begins with the call to **PutString** in order to print the prompt.

```
jsr    i_PutString                      ; call the routine
.word  XPOSPROMPT                       ; the inline x-position, Possible range 0-319
.byte  YPOSPROMPT                       ; the inline y-position, Possible range 0-199
.byte  "Enter something here: ",0       ; the string to print
;---   code resumes here
```

Now we should put up the text prompt. To do this we need to set the size and position. For now we will be printing in the standard GEOS character set which is 9 point and so let's choose 12 for the size of the vertical bar. The x, y-position for the bar is easiest to find by experiment, trying a value and running the program. For now lets define the constants XPOSPROMPT and YPOSPROMPT and guess at their initial values, later.

```
XPOSPROMPT = some x value in range 0 to 319
YPOSPROMPT = some y value in range 0 to 199
```

Next we call **PromptOn** in order to turn on the sprite used for the text prompt and position it. The text prompt uses sprite 1.

```
lda    #9                               ; pass height of text prompt
jsr    InitTextPrompt                   ; init the prompt
LoadW  stringX,#XPOSPROMPT              ; pass the x-position and y-position for prompt
LoadB  stringY,#YPOSPROMPT
jsr    PromptOn                         ; make it visible
```

stringX and **stringY** are the variables used by **PromptOn** to hold the x, y-position of the prompt. The cursor is now visible. **OurGetString** will get a character, print it to the screen, and then move the prompt to the right of the character. Luckily **PutChar** returns **r1** and **r11** updated for the width of the char. All we need to do is transfer the updated x-position to **stringX**. So let's start writing **OurGetString**.

The first thing to do is make sure we get called. Let's load **keyVector** with **OurGetString's** address. While we're at it let's do the same for our string fault vector routine. Add the following line to the prompting code above.

```
LoadW  keyVector,#OurGetString          ; set up keyboard dispatch
LoadW  StringFaultVec,#OurStringFault   ; set up Margin Fault Handler
```

Let's take a close look at **OurGetString**. It gets the first character from **keyVector**, checks for the carriage return the user types to terminate the input string. If the character is not a CR then we echo it with **PutChar**, and store it in the input Buffer. Next, **GetNextChar** is called to return any additional chars until it returns zero. As part of echoing each input character, **OurGetString** will advance the text prompt the width of the character. Since **stringX** and **stringY** are used to pass the x, y-position for the text prompt to **PromptOn**, we also use them to hold the position to print the input characters at as well. The code is as follows.

```
OurGetString:
      ldx    #0                           ; used as index into our buffer
      lda    keyData                      ; get first key
10$
      cmp    #CR                          ; see if user indicates end of string
      beq    90$                          ; if so go terminate the string

      sta    inBuffer,x                   ; add to our input buffer
      pha                                 ; save the char
      inx                                 ; point to next open byte in inBuffer

      MoveW  stringY,r1H                  ; Get position for char from stringX and stringY
      MoveW  stringX,r11                  ;     (the position of the prompt)
      pla                                 ; get the character from stack
      jsr    PutChar                      ; echo the char to the screen
                                          ; PutChar returns new x, y-position
                                          ; in r11 and r1H, use for prompt
      MoveW r11, stringX                  ; Get x-position for next char into
                                          ; stringX. Only x-position changed
      jsr    PromptOn                     ; update the prompt position

      jsr    GetNextChar                  ; see if last character
      ;cmp   #0                           ; (Z flag is set by GetNextChar when buffer is empty)
      bne    10$                          ; Loop again if more characters
      ;--- if zero then exit
90$
      lda    #NULL                        ; terminate the input string in
      sta    inBuffer,x                   ; inBuffer
      rts
```

We can now input and echo characters to the screen. Eventually though, **OurGetString** will try to print a character past **rightMargin**, and **OurStringFault** will get called. We want it to change the x, y-position of the text prompt and the location for drawing upcoming characters to the next line. In order to reset the y-position to the next line, **OurStringFault** has to know how tall the characters on the present line are. The easiest way to do this is to use the routine, **GetRealSize**. **OurStringFault** should save the character passed to it, and call **GetRealSize** to find out the height of the character. It needs to add this height plus a little more to space the lines apart to the present vertical position in **stringY**. **stringX** is set to the left-margin and the character is printed.

```
OurStringFault:
       pha                     ; save the char passed us
       ldx   currentMode       ; style may affect char width
       jsr   GetRealSize       ; we want the height
       txa                     ; height returned in x
       clc
       adc   stringY           ; add height to stringY
       adc   #2                ; add a little line spacing
       sta   stringY           ; new y-position
       LoadW stringX,#leftMargin ; print from left-margin
       pla                     ; restore the char
       jsr   PutChar           ; print the char at beginning of line
       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 l/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 application. 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.

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 then 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: **appMain**, 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 aspects 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 interrupt is an IRQ (Interrupt ReQuest), the 6502 has the option of ignoring 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 needed and then only for short periods of time. If an interrupt 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 interrupt-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 disastrous bug.

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

```
php                             ; save current interrupt disable status
sei                             ; disable interrupts
                                ; (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.

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 Interrupt Level process than its basic functioning. However, there are some unavoidable conflicts between Interrupt 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 Interrupt 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-position
```

Which expands to the following at assembly time:

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

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

```
      php                        ; save current interrupt disable status
      sei                        ; disable interrupts around access
      MoveW  mouseXPos,oldX      ; update our old mouse x-position with current mouse x-position
      MoveB  mouseYPos,oldY      ; Get a consistent y-position at the same time.
      plp                        ; restore old interrupt status
```

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

```
      php                        ; save current interrupt disable status
      sei                        ; disable interrupts around compare
      CmpW   mouseXPos,oldX      ; compare current x-position with Old x-position
      plp                        ; restore old interrupt status
```

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: **IsMseInMargins**

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 interrupt 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 GEOS Interrupt Level. An application that uses **intTopVector** should call the address that was originally in **intTopVector** when it is done. This will ensure that the GEOS **InterruptMain** will be executed properly.

Example:

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

;--- Remove our interrupt routine from intTopVector, replacing it with old.
Removeint:
      php                            ; save current interrupt disable status
      sei                            ; disable interrupts
      MoveW  oldTopVector,intTopVector ; restore old routine
      plp                            ; restore old interrupt status
      rts

;--- My interrupt service routine
MyIntRout:
      ...                            ; interrupt code here
      ...
      ldx    oldTopVector+1
      lda    oldTopVector
      jmp    CallRoutine             ; end with transfer to InterruptMain
```

Guidelines 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 others 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 both systems (GEOS 64, GEOS 128). 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

```
InterruptLevel:
{
        /* 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 in case the pointer is $0000.
        * /
        CallRoutine(intTopVector)           /* call indirectly through intTopVector. On the C64/128, this
                                               points to InterruptMain. */
        CallRoutine(intBotVector)           /* call indirectly through intBotVector. This is usually
                                               $0000, which CallRoutine ignores. */
        / * Context Restore:
           Restore information about the system configuration that we 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

```
/ *
        InterruptMain
        Called through intTopVector under GEOS 64/128.
* /
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();
}
```

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 runnable */
                    ResetTimer();                       /* and reset the counter */
                }
            }
        }
    }
    Return();
}
```

UpdateSleeps

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

            /* Draw the mouse here */
            {
                DrawSprite (mousePicData)              /* copy mouse picture into sprite data table */
                PosSprite (mouseXPos, mouseYPos)       /* position the sprite */
                if (GEOS64)                            / * if GEOS 64... */
                { EnablSprite (MOUSE)                      /* always enable the sprite each time */
                }
            }
        }
    }
    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(MOUSEON_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 both systems (GEOS 64 and GEOS 128). 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 */
    {
        KeyboardService();              /* service keyboard and related MainLoop functions */
        ProcessService();               /* service processes */
        SleepService();                 /* service sleeping routines */
        CBMTimeService();               /* service the Commodore time */
        CallRoutine(appMain);           / * Call any application code that NEEDS to be handled
                                            Every MainLoop */

    } /* endwhile */
}
```

KeyboardService

```
KeyboardService:
{
    if (C128)   /* GEOS 128 handles 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 */
        CallRoutine(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 */
        CallRoutine(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 */
        }
        CallRoutine(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... */
    {
        CallRoutine(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 runnable & not blocked */
            {   Process ==~RUNABLE;                        /* clear runnable flag */
                ProcessEvent();                            /* and generate a process event by calling the
                                                              routine in the table. */
            }
        }
    }
    Return();
}
```

SleepService

```
SleepService:
{
    if (numSleeping > 0)       /* Only do this if there are routines sleeping */
    {
        for (EachSleeping)         /* go through each sleeping routine */
        {
            if (SleepTimer=0)          /* if counter not zero, then time to awake! */
            {
            RemoveSleep();                 /* remove this sleeper from the internal list */
            WakeUp();                      /* and go wake it up */
            }
        }
    }
    Return();
}
```

```
SystemMouseService:
{
    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_FLSH_BIT))     /* flash icon? */
                        { InvertIcon();                        /* invert once */
                          Sleep (selectionFlash);              /* sleep awhile */
                          InvertIcon();                        /* invert back again */
                        }
                        else if (iconSelFlag (ST_INVRT_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
                        { r0H = FALSE;                         / * else, set single click flag */
                        }
```

```
                    /* call the icon event routine */
                    r0L = icon;                          /* tell event routine which icon */
                    CallRoutine (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
  */
  CallRoutine (otherPressVec);         /* it's an "other" press... "other" as in something the
                                          system doesn't really care about */
}
```

SystemFaultService

```
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 == UN_CONSTRAINED */
        {
        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();
}
```

Quick Reference Pseudo-Code

InterruptLevel

*Save the state of the machine. This includes A, X, Y and S plus **r0-rl5** and the memory configuration*

```
        jsr  SaveState
```

Now the I/O area is switched in. GEOS 128 also ensures that bank 1 is the active bank.

```
        jsr  SetIOIn
```

*Now **dblClickCount** is decremented. This variable is used to tell if the user clicks the mouse twice in rapid succession*

```
        jsr  DblClicks

    .if GEOSl28
        GEOS 128 updates the mouse here
        jsr  DoMouse
        jsr  SetMouse
    .endif
```

Now scan the keyboard and if a key is found place it in the keyboard queue.

```
        jsr  DoKeyboard

        jsr  DoAlarmSnd       service alarm tone timer
```

*Normally **intTopVector** points to **InterruptMain**. If you wedge a routine in here the routine must end with jmp **InterruptMain**.*

```
        ldx  #]intTopVector
        lda  #[intTopVector
        jsr  CallRoutine      execute InterruptMain
```

*Normally **intBotVector** is NULL, i.e. $0000.*
A routine wedged in here should end with rts.

```
        ldx  #]intBotVector
        lda  #[intBotVector
        jsr  CallRoutine      normally unused.

        jsr  RestoreState     back the way it was.

        rti
```

InterruptMain

Called during each interrupt via **intTopVector**. This routine performs the bulk of the interrupt's work and must be called or things will freeze up.

```
    .if GEOS64
        GEOS 64 services the mouse here
        jsr  DoMouse
    .endif
        jsr  UpdateProcesses    update process
                                timers
        jsr  UpdateSleeps       update sleep timers
        jsr  UpdatePrompt       flash text prompt
        jmp  GetRandom          get a new random
                                number
```

MainLoop

GEOS 128 will handle soft (80-col) sprites here.

```
        jsr  KeyboardService  service pressFlag,
                              inputVector,
                              mouseVector,
                              keyVector,
                              mouseFaultVec
```

*Menu/Icon mouse presses are handled through **mouseVector**. **mouseVector** is normally set to **SystemMouseService**. When mouse action is not handled, then **SystemMouseService** calls **CallRoutine** (**otherPressVec**)*

Now we check if any processes or sleeping routines should be executed.

```
        jsr  ProcessService
        jsr  SleepService
```

Next, update the time and alarm variables. If it is time for the alarm to sound call **alarmTmtVector**.

```
        jsr  CBMTimeService
```

***appMain** is normally NULL. You can wedge your own MainLoop routines in here*

```
        ldx  #]appMain
        lda  #[appMain
        jsr  CallRoutine

        rmbf 7,grcntrl1

        jmp  MainLoop           loop is never ending
```

Dialog Box

Dialog Boxes (DB) appear as a rectangle in which text, icons, and string manipulation may occur. Dialog Boxes are used by applications to display error conditions, warn the user about possibly unexpected side effects, prompt for a sentence or two of input, present filenames for selection, and perform various other tasks where user participation is desired. Several frequently used Dialog Box functions are built directly into the GEOS Kernal. Along with programmer defined functions, Dialog Boxes provide a simple, compact, yet flexible user interface.

A Dialog Box may be called up on the screen at any time. It is like a small application, running in its own environment. It will not harm the current application, or change any of its data (unless this is intentionally done by a programmer supplied routine). Calling up a Dialog Box causes most of the state of the machine to be saved. All the Kernal variables, vectors, and menu and icon structures are saved. The Dialog Box can therefore be very elaborate, since it need not worry about permanently affecting the state of the machine. The pseudoregisters **r0H-rl5**, however, are not saved, nor is the screen under where the Dialog Box appears. Restoring the screen appearance after a DB is run is described later.

To call up a Dialog Box use the routine **DoDlgBox**. To exit from a Dialog Box and return to the application call **RstrFrmDialog**. All the variability of Dialog Boxes is provided by a powerful yet simple table. The table specifies the dimensions and functionality of the Dialog Box. DB tables are made up of a series of command and data bytes. DB command bytes indicate icons to display or commands (usually for printing text) to execute within the DB. DB data bytes specify information such as location of the DB, its dimensions, and text strings.

DB Structure

The first entry in a DB table is a command byte defining its position. This can either be a byte indicating a default position for the DB, DEF_DB_POS, or a byte indicating a user defined position, SET_DB_POS which must be followed by the position information.

Position Command

The position command byte is or'ed with a system pattern number to be used to fill in a shadow box. The shadow box is a rectangle of the same dimensions as the DB and is filled with one of the system patterns. The shadow box appears underneath the DB one card to the right and one card below. A system pattern of 0 indicates no shadow box. It's easier to look at an example of a DB with a shadow box than it is to describe it. **A picture of one** appears in the **Open Box example** later in this chapter.

The two forms for the position byte, default and user defined, are:

Start of Default Dialog

```
------------------------------------
.byte DEF_DB_POS | pattern
```

Start of Custom Size Dialog

```
-----------------------------------------------
.byte    SET_DB_POS | pattern
.byte    top            ; (0-199)
.byte    bottom         ; (0-199)
.word    leftside       ; (0-319)
.word    right          ; (0-319)
```

Note: Additional information on Dialogs can be found in "**Chapter 19 Environment > Structures > dialog/Icons/Menus/Graphics**"

DB Icons and Commands

The Kernal supports a special set of resident icons for use in DBs. DB Icons provide a simple user response to a question or statement. When the user clicks on one of these icons the DB is erased, the number of the selected icon is returned in **r0L**, and **RstrFrmDialog** is automatically called. The application that called **DoDlgBox** then checks **r0L** and acts accordingly, usually calling a routine it associates with that icon. DB Icons indicating OK, CANCEL, YES, NO, OPEN and DISK are provided.

DB Commands are provided for running any arbitrary routine, printing a text string, prompting for and receiving a text string, putting up a scrolling filename box, putting up a user-defined icon, and providing a routine vector to jump through if the joystick button is pressed when the cursor is not over any icon. DB Commands take the form of one command byte containing the number of the command to execute and any following optional data bytes. After the position byte (or bytes) may appear a number of icon or command bytes.

Icon Commands

Whenever a system DB icon is activated, the DB exits, returning the icon's number in **r0L**. The application can then know which icon was selected and take the appropriate action. A maximum of 8 icons may be defined in a DB.

An Icon byte is followed by two bytes defining the position of the icon as an offset from the upper left corner of the DB. The first is the x-position (icon x-position uses cards, 0-39; text x-position use pixels 0-255); the second is the y-position in pixels, 0-199. The OK icon is the most common icon. The OK icon command would look like the following:

```
.byte   OK                        ; icon to display
.byte   x_card_offset             ; icon x-position uses bytes (cards)    0-39
.byte   y_offset                  ; y-position is always in pixels        0-199
```

Table of icon commands

Icon	Value	Example	Description
OK	1	.byte OK .byte x_card_offset .byte y_offset	Draw OK icon x-offset is in cards (0-39) y-offset in lines (0-199)
CANCEL	2	.byte CANCEL .byte x_card_offset .byte y_offset	Draw CANCEL icon x-offset is in cards (0-39) y-offset in lines (0-199)
YES	3		etc...
NO	4		
OPEN	5		
DISK	6		
	7-10		Marked for future use.

Important: The x-position of text fields is stored in a single byte, not in the normal word. This limits the x-position to a range of 0-255. Since the x-position is an offset from the left-side of the Dialog Box this would only be a limitation if a custom size dialog box is created that is wider than 255 pixels.

Dialog Box Commands

Several commands are defined for use in DBs. Many are used to put up text within the Box. For example, the command DBTXTSTR is followed by two position offset bytes and a word pointing to a text string. When used in a DB, DBTXTSTR will display the text string at a position offset from the upper left corner of the DB. The position offsets are measured in pixels from top of the DB to the baseline of the text string, and in pixels from the left-side of the DB to the left-side of the first character in the string. This means any string may be offset at most 255 pixels from the left-side of the DB. The following table contains the available commands.

Table of DB Commands:

Command	Value	Example	Description
DBTXTSTR	11	.byte DBTXTSTR .byte x_offset .byte y_offset .word textPtr	**PutString** *textPtr* at selected offsets. pixel offset 0-255 pixel offset 0-199 *textPtr* contains address of null terminated string
DBVARSTR	12	.byte DBVARSTR .byte x_offset .byte y_offset .byte zPgPtr	**PutString** @@zPgPtr *zPgPtr* is an address of a zero page ptr to a null terminated string. **Example**: .byte **r15**
DBGETSTRING	13	.byte DBGETSTRING .byte x_offset .byte y_offset .byte zPgPtr .byte *maxChars*	Read a text string typed by user into buffer. *zPgPtr* points to address of a buffer that is *maxChars* bytes. **Example**: .byte **r5** with **r5** containing address of string buffer
DBSYSOPV	14	.byte DBSYSOPV	Enable function that causes a return to the application whenever mouse is pressed any place except over an icon.
DBGRPHSTR	15	.byte DBGRPHSTR .word graphicsStrPtr	**i_GraphicsString** *graphicsStrPtr* *graphicsStrPtr* contains address of a graphics string. ([1]This command will end Dialog Box processing)
DBGETFILES¥	16	.byte DBGETFILES .byte x_offset .byte y_offset	Display the filename box inside the DB. ¥ **r7L** = **FILETYPE** **r5** = buffer **r10** = file class
DBOPVEC	17	.byte DBOPVEC .word msePressVector	sets **otherPressVec** to *msePressVector*. Vector is called when the mouse button is pressed any place except over an icon.
DBUSRICON	18	.byte DBUSRICON .byte x_card_offset .byte y_offset .word userIcon	userIcon table: .word ptrIconData .word NULL .byte width in bytes .byte height in pixels .word ServiceRoutine **Note**: (width | DOUBLE_B for 128)
DB_USR_ROUT	19	.byte DB_USR_ROUT .word userVector	Call *userVector* after the DB is drawn and before the DB icons have been drawn.
NULL	0	.byte NULL	Ends the Dialog Box definition.

¥ See **Example**: **GetWorkFile**.

The registers **r5** through **r10** and **r15** may be used to pass parameters to those commands expecting them. (As well as any other zero page address the application has defined for itself, e.g. **a0**). A couple of the commands deserve further explanation.

DBGETSTRING

DBGETSTRING receives a text string from user input and stores it in the buffer pointed to by the register address specified in *zPgPtr*. The command also echoes the input string onto the screen, at the position indicated by the coordinates *x_offset* and *y_offset*, expressed in pixels. The offsets refer to the upper left corner of the dialog box. The maximum number of characters that can be entered is set with *maxChars*.

DBGRPHSTR

DBGRPHSTR command will always be the last command processed in the table. If you need to draw grahics on the dialog box and you need another command to be the last command, you should use **DB_USR_ROUT** instead and do the call to **GraphicsString** within the user routine.

DBGETFILES

The DBGETFILES DB command is the most powerful. A picture of it appears below:

A box containing the names of files which can be selected is displayed. If there are more files than can be displayed at one time, the up/down arrow icon can be used to scroll the filenames up or down. A maximum of 15 files may be viewed this way. Usually this is enough. Upon execution of the DB, **r7L** is expected to contain the GEOS file type (SYSTEM, DESK_ACC, APPLICATION, APPL_DATA, FONT, PRINTER, INPUT_DEVICE, DISK_DEVICE, AUTO_EXEC, INPUT_128). **r5** should point at a buffer to contain the selected filename. If the caller passes a filename in **r5** and this file is one of the files found by DBGETFILES, then this filename will appear highlighted when the filenames are displayed in the dialog box.

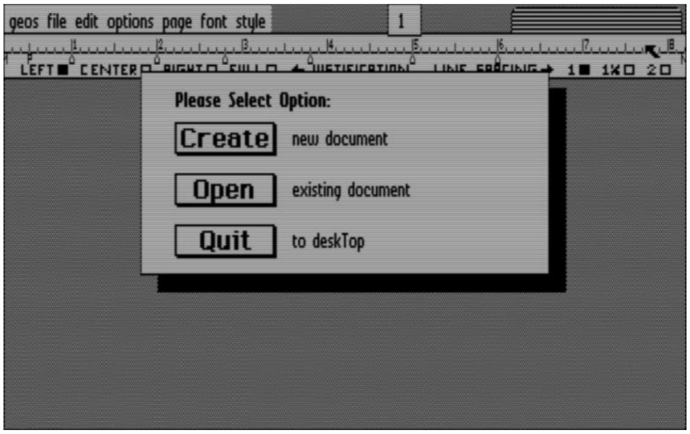
When a file is selected, its name will be null terminated and placed in this buffer. **r10** should be set to null to match all files of the given type, or point to a buffer containing the permanent name string of files to be matched. The permanent name string is contained in the **file header block** for each file. It contains a name that is the same for all files of the same type. For example, geoPaint will only want to open files it created. It points **r10** to the

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

string "Paint Image", when using DBGETFILES. This is useful for displaying only those files of GEOS type APPL_DATA created by a specific program.

The end of a DB definition table is signaled with a .byte NULL as the last entry. As examples speak louder than explanations, we present two DB examples below:

Example: **openBoxDB**, **getFileDB**.



DBOPVEC

DBOPVEC sets up a vector which contains the address of a routine to call whenever the user clicks outside of an icon. This routine will be run and its rts will return to the DB code in **MainLoop**. Other icons or DB commands may then be executed, or icons selected.

If the programmer wants the routine to exit from the DB altogether as DBSYSOPV does, then a jmp **RstrFrmDialog** should be executed from within the routine. Whenever this is done, **sysDBData** should be loaded with a value that **RstrFrmDialog** will then transfer to **r0L** when it exits. In situations where several user responses are possible within a DB, the calling application checks **r0L** to determine the action that caused the DB exit. Your DBOPVEC routine should return **sysDBData** a value that cannot be mistaken for a different icon in the same DB. Since DBs can only handle 8 icons, any number greater than 8 is sufficient.

DBUSRICON

If the programmer wishes to have an icon in a DB that is not one of the Kernal supported DB icons, he may use the DBUSRICON command to define his own. A word following the command byte points to an icon table not unlike the table normally used to define icons within an application. As can be seen in the **Table of DB Commands**, the position bytes for the icon within this table are set to zero as the position offset bytes just following the command byte are used instead. The user routine pointed to from inside this icon table is executed immediately when a press within the icon is detected. Like DBOPVEC, instead of returning to the application like the predefined system icons, this user icon returns to the DB level in **MainLoop**.

To make the user routine return to the application it may execute a jmp **RstrFrmDialog**. A QUIT or OK icon may also be used in the same DB to cause a return to the application. As with DBOPVEC, the DBUSRICON routine should load **sysDBData** with a value that **RstrFrmDialog** will then transfer to **r0L**. This value should be selected so that the application will not mistake it for one of the DB icons.

DB_USR_ROUT

The DB_USR_ROUT command executes a programmer supplied routine when the DB is drawn. This routine may be quite elaborate, setting up processes, menus, edit windows and the like. Since **DoDlgBox** and **RstrFrmDialog**, respectively, save and restore the system state, a DB_USR_ROUT called routine need not worry about trashing the state of the system. However, you may not call **DoIcons** from within a DB_USR_ROUT if you are also using the standard Dialog Box Icons as the two sets of icons will interfere. The DB icon structure is drawn and initialized after the DB_USR_ROUT is called. This way an icon may be placed on top of a graphic drawn by the DB_USR_ROUT.

Note: It is standard practice in Berkeley applications to have the DB_USR_ROUT set **appMain** to point to the routine that will perform the custom dialog box setup. The first step that routine performs is to remove it's hook from **appMain**. This allows the dialog to complete its internal processing before we do our modifications. The first time the **MainLoop** runs after the dialog is done our routine will get called through the **appMain** vector.

Example:

```
ExDB:
      .byte  DEF_DB_POS | 1           ; simple dialog definition table
      .byte  OK, DBI_X_2,DBI_Y_2      ; OK Button
      .byte  DB_USR_ROUT              ; setup for our routine to get hooked
      .word  DBHook                   ; into the MainLoop
      .byte  NULL

DBHook:
      ;---   code here executes BEFORE Dialog Box icons are drawn
      LoadW  appMain,#UsrRoutine      ; set our UsrRoutine to be called at
      rts                             ; the end of the next MainLoop

UsrRoutine:
      ;---   code here executes AFTER all dialog box setup is done.
      LdNull appMain                  ; remove hook into main loop
      LoadW  r0,#myGraphics
      jsr    GraphicsString
      ...
```

Exiting from a DB

The applications screen is recovered in one of two ways. First, if the screen's contents are buffered to the background screen, then all that needs to be done is a **RecoverRectangle** which will copy the background screen to the foreground screen. If the **dispBufferOn** flag is set so that the background is being used for code space and not to buffer the foreground screen, then the application must provide another means to recreate the screen appearance.

When **RstrFrmDialog** is called it will call the routine whose address is in **RecoverVector**. **RecoverVector** normally contains the address of **RecoverRectangle**. To recover the screen when the display is being buffered, two calls through **RecoverVector** are done. First, the **RstrFrmDialog** routine sets up the coordinates of the DB's shadow box and vectors through **RecoverVector**. This will restore the area under the shadow box. Second, it sets up the coordinates of the area under the DB itself and vectors through **RecoverVector** again. In this way the contents of the Background Screen corresponding to the area under the DB and its shadow box are copied to the Foreground Screen.

If the application does not use the Background Screen RAM as a screen buffer then it must provide the address of a different routine to call. The alternate routine address must be stored in **RecoverVector** in order to provide some other means of recreating the screen appearance. **RecoverVector** is called once for the Shadow and then once for the Dialog Box. If there is no Shadow then **RecoverVector** will only be called one time.

The dimensions of the areas to recreate are passed in the regular **RecoverRectangle** registers **r2-r4**. When you have a shadow, it will be more efficient to only recover the screen behind the "Full Dialog Box" one time instead of once for the Shadow rectangle and again for the Dialog Box only rectangle. To do that you will need a flag to show the state of the drawing and variables to save the shadow dimensions. The Following example illustrates a simple recovery setup that uses the default background pattern to replace the removed dialog box.

Example:

```
;--- (ramsect area assumed to be initialized to NULL at program startup)
.ramsect
       rYB:
             .block 1     ; holds the bottom y-coordinate of the shadow and doubles as our flag
       rXR:
             .block 2     ; holds right x-coordinate of the shadow area

RecoverRect:
       lda   rYB          ; if rYB is zero we are in the first call
       bne   50$
       ;---  First call from RecoverVector
       MoveB r2H,rYB      ; save the bottom y-coordinate
       MoveW r4,rXR       ; save the right x-coordinate
       rts                ; exit so the dialog can continue to be removed

50$    ;---  second call from RecoverVector
       sta   r2H          ; set bottom of the recovery rectangle to the bottom of the shadow
       MoveW rXR,r4       ; set right-side of the recovery rectangle to the right of the shadow
       LoadB rYB,#NULL    ; reset flag to NULL so it will be in correct state next use
       lda   #2           ; recover behind the full dialog Box using standard background pattern
       jsr   SetPattern
       jmp   Rectangle

       ;--- sample Setup before call to DoDlgBox
       LoadW RecoverVector,#RecoverRect; activate RecoverVector processing
       LoadW r0,#dlgDB
       jsr   DoDlgBox                  ; activate dialog box
       LoadW RecoverVector,#NULL       ; turn off RecoverVector processing
       lda   r0L
       ...
```

Dialog Box RAM Buffer

This buffer is for variables that are saved when dialog boxes or desk accessories are run. Both of these actions require the system to be able to warmstart GEOS and return to the application state after the action completes. This ability to backup and restore the system state allows for both the Dialog Box / Desk Accessory to startup into a known base startup, just like the application itself always starts up at this same warmstart state.

Limitations

There are 2 rules to Dialog Boxes that have to be followed since there is only 1 buffer and no mechanism for nesting.

1. Never run a Dialog Box from a Dialog Box.
2. Never run a Dialog Box from a Desk Accessory.

Attempting to do either of those actions will cause unpredictable results likely causing a system crash when returning from the Dialog Box. To see why this happens we will need to examine the contents of the **dlgBoxRamBuf**.

Note: See **"Dialog Box RAM Buffer"** in Chapter 19: "**Environment > Structures"** for a detailed breakdown of the contents of this buffer.

Removing Limitations

In order to perform either of the tasks listed above, an application will need to back up the Dialog Box RAM Buffer before either of those actions and then restore it after the action is done.

The **dlgBoxRamBuf** is TOT_SRAM_SAVED bytes so you will need a buffer in ramsect large enough to hold it.

Applications can do their own backup and restore of this buffer to get around the Dialog Box limitations:

Example:

```
	TOT_SRAM_SAVED = 417

.ramsect
	dbrb_back:
		.block	TOT_SRAM_SAVED		; allocate enough RAM to hold a copy of the buffer

.psect
	Bck_dbrb:
		jsr	i_MoveData		; move the contents of the dlgBoxRamBuf to holding
		.word	dlgBoxRamBuf		; buffer
		.word	dbrb_back
		.word	TOT_SRAM_SAVED
		rts
```

```
Rst_dbrb:
      PushB  r0L
      jsr    i_MoveData          ; restore the contents of the dlgBoxRamBuf
      .word  dbrb_back           ; from the holding buffer
      .word  dlgBoxRamBuf
      .word  TOT_SRAM_SAVED
      PopB   r0L
      rts

;--- From an Auto Exec or from inside a dialog.

jsr    Bck_dbrb                  ; backup dialog RAM buffer
LoadW  r0,#dlgBox                ; display the dialog box
jsr    DoDlgBox
jsr    Rst_dbrb                  ; restore the dialog RAM buffer
lda    r0L
;--- process Dialog Box Result
....
```

Note: To allow further nesting of Dialog Box's, an application would need a way of tracking nesting levels and a storage strategy for keeping the nested 417-byte buffers. With nesting logic in place, you could easily allow an Auto Exec to not only use a Dialog Box, but that Dialog Box could also call another Dialog Box.

File System

The GEOS file system is based on the normal C64 DOS file system. A combination of two factors led to an augmentation of the basic structure: first, the C64 was not originally designed to be a disk computer, and second, the addition of the diskTurbo now makes it practical to read and write parts of a file as needed. Previously the slowness of the disk drive often meant that files were read in at the beginning of execution, and not written until exiting the program. If file writes had to be done in the middle of execution, a coffee break was usually warranted.

GEOS supports two different types of files. The first is similar to regular C64 files and is called a SEQUENTIAL[¥] file. This type of file is made up of a chain of sectors on the disk. The first two bytes of each sector contain a track and sector pointer to the next sector on the disk, except for the last sector which contains $00 in the first byte to indicate that it is the last block, and an index to the last valid data byte in the sector in the second byte. The second type of file is a new structure, called a Variable Length Indexed Record, or VLIR for short. An additional block, called a Header Block, is added to both VLIR & SEQUENTIAL files. It contains an icon graphic for the file, as well as other data as discussed later.

To understand GEOS files, one must first understand the Commodore files on which they are based. I refer the reader to any of the several good disk drive books available. I use the Commodore 1541 (or 1571) User's Guide, and The Anatomy of the 1541 Disk Drive (from Abacus Software).

This chapter is divided into three sections. The first, for those already familiar with the 1541, is a brief refresher of the basic Commodore DOS. Second, we present GEOS routines for opening and closing disks and dealing with directories and standard files. The final section is devoted to a detailed look at VLIR files.

The Foundation

A 1541 disk is divided into 35 tracks. Each track is a narrow band around the disk. Track 1 is at the edge of the disk and track 35 is at the center. Each track is divided into sectors, which are also called blocks. The tracks near the outside edge of the disk are longer and therefore can contain more blocks than those near the center. The Block Distribution by Track tables show the number of sectors in each track for each of the GEOS 2.0 supported drives.

1541 Block Distribution by Track

Track Number	Range of Sectors	Total Blocks
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17
		683

1581 Block Distribution by Track

Track Number	Range of Sectors	Total Blocks
1 to 80	0 to 39	3160
		3160

1571 Block Distribution by Track

Track Number	Range of Sectors	Total Blocks
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17
36 to 52	0 to 20	21
53 to 39	0 to 18	19
60 to 35	0 to 17	18
66 to 70	0 to 16	17
		1366

Directory Track

Note: [¥] SEQUENTIAL stands for any non-VLIR file in GEOS, and should not be confused with the SEQ C64 file format. In fact, USR, PRG and SEQ C64 files all qualify as GEOS SEQUENTIAL file types.

Track 18, the 1541/1571 directory track (1581 uses track 40), is used to hold information about the individual files contained on the disk. Sector 0 on this track contains the Block Availability Map (BAM) and the directory header. The BAM contains 1 bit for every available block on the disk. The bits corresponding to blocks already allocated to files are set while the bits corresponding to free blocks are cleared. Before the BAM bits is a pointer to the first directory block, which is described later. The BAM format is unchanged by GEOS.

Directory Header

The Directory Header contains the disk name, an ID word (to tell different disks apart), and three new elements for GEOS, a GEOS ID string, a track/sector pointer to the Off-Page Directory block, and a disk protection byte. The GEOS ID string is contained in an otherwise unused portion of the BAM/directory header block. It identifies the disk as a GEOS disk and identifies the version number, which can be important for data compatibility between present and future versions of GEOS. See the BAM Format/directory header table below. This string should not be confused with the GEOS Kernal ID and version string at $C000 as described in **"GEOS Kernal Information Bytes"** in Chapter **"Basic GEOS"**.

Here is the format of the 1541/71 BAM and directory header:

	BAM Format/Directory Header			
	Byte Offset	**Constant**	**Contents**	**Definition**
	`$00`		18	Track of first directory block. Always 18.
	`$01`		1	Sector of first directory block. Always 1.
	`$02`		'A'	ASCII char A indicating 1541 disk format.
	`$03`			1541 and 1581 not used: $00 1571 double-sided flag: $00 = single-sided format $80 = double-sided format
BAM TRACK 1	`$04`	`OFF_TO_BAM`		Number of free sectors in track 1
	`$05`			Track 1, BAM for sectors 0-7
	`$06`			Track 1, BAM for sectors 8-15
	`$07`			Track 1, BAM for sectors 16-20
BAM TRACK 2	`$08`			Number of free sectors in track 2
	`$09`			Track 2, BAM for sectors 0-7
	`$0A`			Track 2, BAM for sectors 8-16
	`$0B`			Track 2, BAM for sectors 17-20
	... BAM for tracks 3-35			
DIRECTORY HEADER	`$90`	`OFF_DISK_NAME`		16-byte Disk name
	`$A0`		$A0 x 2	2 Shifted spaces
	`$A2`	`OFF_DSK_ID`		Disk ID (word)
	`$A4`		$A0	Shifted space
	`$A5`		'2'	DOS version: 2
	`$A6`		'A'	Format type
	`$A7`		$A0 x 4	4 Shifted spaces
	`$AB`	`OFF_OP_TR_SC`		Tr/Sr of off page directory block
	`$AD`	`OFF_GS_ID`	16 bytes	GEOS ID string. "GEOS format V1.2"
	`$BD`	`OFF_GS_DTYPE`	$00 / 'P'	'B' indicates protected Boot disk 'P' indicates protected Master disk
	`$BE`		$00	Unused

Disk Protection Byte

The disk protection byte is at OFF_GEOS_DTYPE (189) in the Directory Header. This byte is normally 0, but may be set to 'P, to mark a disk as a Master Disk. GEOS Version 1.3 and beyond deskTops will not allow a Master Disk to be formatted, copied over, or have files deleted from the deskTop notePad. Files may still be moved to the border and deleted from there. This saves GEOS developers from having to replace application disks that have been formatted, or otherwise destroyed by user accident.

Off Page Directory Block

The Off-Page directory block is a new GEOS structure but has the same format as regular Commodore directory blocks. Directory blocks hold up to 8 directory entries. Each directory entry (also known as file entry because it describes a file), contains information about one file. When a file is moved off the deskTop notepad onto the border, the file's directory entry is erased from its directory block and is copied to the off-page directory block. A buffer in memory is also reserved to save information about each file on the border.

Important: The off-page feature exists so that a file can be copied between disks on a one drive system. The Icon for an off-page file will remain on the deskTop border when a new disk is opened and the deskTop set to display the contents of the new disk. The file can then be dragged to the notepad from the border, thus copying it to the new disk.

Directory Block

The format of the directory block is shown below. The overall structure of a directory block is unchanged. The following table was derived from the C64 disk drive manual.

Directory Block Structure

Offset	Description
$00	Track and sector of next directory block
$02	Directory entry 1
$20	*Unused*
$32	Directory entry 2
	...
$E0	*Unused*
$E2	Directory entry 8
Directory blocks appear only on the directory track	

Directory Entry

Several unused bytes in each directory entry have been taken for use by GEOS. Bytes 1 and 2 point to the first data block in the file unless the file is a GEOS VLIR file. In this case these bytes point to the VLIR file's index table. Bytes 19 and 20 point to a new GEOS table, the file header block as described below. Bytes 21 and 22 are used to convey the GEOS structure and type of the file. The structure byte indicates how the data is organized on disk: 0 for SEQUENTIAL, or 1 for VLIR. The file type refers to what the file is used for, DATA, BASIC, APPLICATION and other types as listed in the table below. The SYSTEM_BOOT file type should only be used by GEOS Boot and Kernal files themselves.

The TEMPORARY file type is for swap files. All files of type TEMPORARY are automatically deleted from any disk opened by the deskTop. The deskTop assumes they were left there by accident, usually when an application crashes and a swap file is left behind. When creating swap files, use the TEMPORARY file type and start the filename with the character PLAINTEXT.

Example:

```
        swapName:
                .byte PLAINTEXT,"My swap file",NULL
```

This will cause the file to print in plain text on the desk top and will prevent a user file with the same name to be accidentally removed when "My swap file" is created. Finally, bytes 23 through 27 are used to hold a time and day stamp so that files may be dated.

Directory Entry

Offset	Description
$00	Commodore file type
$01	Track and sector of first data block in this file. or VLIR index block
$03	16 Character file name padded with shifted spaces $A0
$13	Track and sector of GEOS file header (new structure)
$15	GEOS file structure type: 0=SEQuential, 1=VLIR
$16	GEOS file type
$17	Date: Year. The year is stored as the last two digits of the actual year. Applications must provide their own century logic. It is safe to assume any year < 86 is in the 21st century. The original spec was for the year to be an offset from the year 1900. This would have been the perfect solution. The GEOS code base may be too large to patch and fix this problem now.
$18	Month/day/hour/minute
$1C	File size expressed as number of blocks in the file. (word)

Note: For a more detailed view of the directory entry see **"Directory Entry"** in Chapter 19 "**Environment > Structures**".

File Header Block

The GEOS file header block was created to hold the icon picture and other information that is handy for GEOS to have around. Something worth bringing attention to is that the file header block is pointed to by bytes $13-14 of the file's directory entry. Thus, any C64 SEQUENTIAL file may have a header block. (Bytes $13-14 was previously used to point to the first side sector in a C64 DOS relative file, so these bytes are unused in a SEQUENTIAL file. This is also why the REL file is not a valid Commodore file type under GEOS). Bytes 0 and 1 in all disk blocks point to the next block in the file, or the offset to the last data byte in the last block of a file. Since the file header block is only a single block associated with a file, bytes 0-1 are always set to $00, $FF. This indicates that no blocks follow and all bytes in the block are used.

We follow the header block diagram below by a complete description of its contents:

GEOS File Header Block

(256 bytes. New GEOS file extension. Pointed to by Directory Entry)

Offset	Constant	Contents	Description
$00		00, FF	00=Indicates this is the last block in the chain. FF=Index to the last valid data byte in the block.
$02	O_GHIC_WIDTH	3	Width of icon in bytes, always 3
$03	O_GHIC_HEIGHT	21	Height of file icon in lines, always 21.
$04	O_GHIC_PIC		Icon data
$44	O_GHCMDR_TYPE		C64 file type
$45	O_GHGEOS_TYPE		GEOS file type
$46	O_GHSTR_TYPE		GEOS structure type
$47	O_GHST_ADDR		Start address in memory for loading the program.
$49	O_GHEND_ADDR		End address in memory for loading a desk accessory, otherwise start address -1.
$4B	O_GHST_VEC		Address of initialization routine to call after loading the program.
$4D	O_GHFNAME O_GHCNAME		Permanent filename. Bytes 0-11 = Filename padded with spaces; Permanent ClassName. (Data Files) Bytes 0-11 = ClassName Padded with spaces 12-15=version string "V1.3"; 16-18=0's.
$60	O_128_FLAGS		OS compatibility flag.
$61	O_GH_AUTHOR	Author Name	If application program, holds name of software designer.
$75	O_GHP_FNAME	Parent Application	If data file, 20-byte parent application's permanent filename. Bytes 0-11=name padded with spaces; 12-15=version string "V1.3"; 16-20=0's
$89	O_GHAPDAT	Application	23 bytes for application use.
$A0	O_GHINFO_TXT	Get Info	Used for the file menu option Info. String must be null terminated.

Fonts use the data area of the file header block from $61 to $9F in a different way.

Offset		Contents	Description
$61	O_GHSETLEN		VLIR Size (word) of each Point Size. 15 words.
$80	O_GHFONTID		**Font style ID** (word).
$82	O_GHPTSIZES		List of **Character Set ID** (word). 15 words.

> **Note**: For a more detailed view of the File Header Block see **"File Header Block"** in Chapter 19 "**Environment > Structures**".

File Header Block In Detail:

Icon data

Bytes at offset O_GHIC_WIDTH contain the width and height of the icon data that follows. File icons are always 3 bytes wide by 21 scan lines high. The two-dimension bytes precede the data because the internal routine used by GEOS to draw icons is a general routine for drawing any size icon and it expects the two bytes to be there. The image bytes at O_GHIC_PIC contain the picture data for the icon in compacted bit-map format. Byte 4 is the bitmap format byte. There are three compacted bit-map formats. The second format as described in **"GEOS Compacted Bitmap Format"** in chapter **Graphics Routines**, is a straight uncompacted bit-map. To indicate this format, the format byte should be within the range 128 to 220. The number of bytes in the bit-map is the value of this format byte minus 128. Since the value of the highest bit is 128, the lower 7 bits, up to a value of 92 indicate the number of bytes that follow.

Commodore File Type

The lowest 3 bits at O_GHCMDR_TYPE is the old C64 file type, PRG, SEQ, USR, or REL.

GEOS file type

The byte at O_GHGEOS_TYPE, is the GEOS file type. Presently there are 15 different GEOS file types. There may be additional file types added later, but these will most likely be application data files and will be lumped together under APPL_DATA.

GEOS file structure type

O_GHSTR_TYPE is the GEOS file structure type. This is either VLIR or SEQUENTIAL. (Remember, a SEQUENTIAL GEOS file is just a linked chain of disk blocks. It does not mean a C64 SEQ file).

Start Address

The word at O_GHST_ADDR is the starting address at which to load the file. Normally, GEOS will load a file starting at the address specified in O_GHST_ADDR. Later we will see how an alternate address can be specified. This is sometimes useful for loading a data file into different places in memory.

End Address

The word at O_GHEND_ADDR contains the address of the end of the file. GEOS uses this address when loading Desk Accessories. This allows GEOS to backup enough application space to allow the desk accessory to be loaded. Other file types besides Desk Accessories should have an End Address = Start Address – 1.

Application Initialization vector

If the file is a BASIC, ASSEMBLY, APPLICATION, or DESK_ACC, then it is an executable file. The deskTop will look at the word at offset O_GHST_VEC for the address to start execution at after the file has been loaded. Usually this is the same as the start address for loading the file, but need not be.

Permanent Filename / Permanent ClassName

A Permanent Filename for a file is necessary since the user can rename files at will. VLIR applications like geoWrite need to be able to find their VLIR records when they first load up. Instead of searching for the name "GEOWRITE" which can be changed by the users, it searches for it's Permanent File Name which will always be the same even if the file is named "Suzy Wong at the Beach".

The 20 bytes at O_GHP_FNAME store the Permanent Filename string for all files except APPL_DATA files. Though there are 20 bytes allocated for this string, the last 4 bytes should always be 3 nulls (0). For applications the last byte is the OS Compatibility Flag at offset O_128_FLAGS, otherwise it is another 0. Bytes 0-11 are used for the file name and padded with spaces if necessary. Bytes 11 to 15 should be the version number of the file. We have developed the convention that Version numbers follow the format: V1.0 where V1 is just a capital ASCII V followed by the major and minor version numbers separated by an ASCII period.

Example Permanent File Name:

```
.byte  "geoWrite    V2.1",NULL,0,0,CF_40
```

APPL_DATA files use a Permanent ClassName at O_GHP_CNAME. This is the same location in the header as O_GHP_FNAME. The 20 byte string is a 12 character ClassName followed by a 4 character Version number and then 4 nulls. The Class Name is used by applications when they are looking for their data files. They will search for all files of a specific class. This also serves the purpose of allowing the Application to know the version of the Data File.

Example Class Name:

```
.byte  "Write Image V2.0",NULL,0,0,0
```

Author

The 20 bytes at O_GH_AUTHOR are for storing Information about the Creator of the application. The string in this field must be NULL terminated.

Example:

```
.byte "Dave & Mike",NULL,0,0,0,0,0,0,0,0
```

Parent Application

When GEOS needs to locate an application it looks at the Parent Application string at O_GHP_FNAME. When a user double clicks on a data file, GEOS will look at the Parent Application string and try to find a file of that name. If it cannot find the file on the current disk, it will ask the user to insert a disk containing an application file of that name, "Please insert a disk with geoWrite". When looking for an application, GEOS will only check the first 12 letters of the name, the filename, and will ignore the Version Number for the time being. GEOS assumes that the user will have inserted the version of the application he wants to use. In making this assumption, GEOS tacitly assumes that applications will be downwardly compatible with data files created by earlier versions of the same application. This need not absolutely be the case as will be seen below.

When the application is loaded and begins executing, it should look at the Permanent ClassName String of the data file. Normally this string will be similar to the Parent Application filename and the version numbers may be different. Thus, if you double click on a datafile and that datafile has a Parent Application of "geoWrite   V2.1" the deskTop, which doesn't compare version numbers, will load and start executing geoWrite 2.1. geoWrite will then look at the version number in the data file's Class Name String and determine if a conversion of data file formats needs to take place. If there were changes between the V1.2 and 2.0 versions of the data files then the data will have to be converted.

It is much more likely for the code of a program to change - to fix bugs - than it is for the data file format to change. Data format version numbers then tend to leapfrog application numbers. For example, application X starts out with V1.0. After a month of beta test V1.1 is released. After 1 week of retail shipping a bug is found and a running production change to V1.2 is made and users with V1.1 are upgraded. Meanwhile the data file format is still V1.0; any version of the application can use it. Six months later V2.0 is released with greatly expanded capabilities and a new data format. The data Version Number should then change to V2.0, leapfrogging V1.1, and V1.2. This will indicate to V1.0 to V1.2 versions of the program that they cannot read the new format. If the user has the newer version of the program than he should be using it and not an older version.

Important: It is up to the application in its initialization code to look at the data file's version number and determine whether or not it can handle it, and if so whether or not the data needs to be converted.

Permanent Name Example

As an example, suppose the user double clicks on a geoWrite 1.0 document. The deskTop will look for a file with the name stored in the Parent Application string. If this program is not found on the current disk the deskTop will ask the user to insert a disk containing it. The deskTop only looks at the first 12 characters and will ignore the version number. After loading geoWrite, control is passed to the application. The deskTop passes a few appropriate flags and a character string containing the name of the data file. The application, in this case geoWrite, will look at the data file's Permanent Class name string, then its version number, and determines if it can read the file, or if it needs to convert it to the more up-to-date version. Similarly, if an older version of an application, e.g. geoWrite 1.0, cannot read a data file created with a newer version of the application, it needs to cancel itself and return to the deskTop or request another disk.

Constants for Accessing Table Values

Constants that are used with the file system and tables described above are included in Chapter 19 "**Environment > Constants**". These constants make code easier to read and support, and therefore are included here. Most of the constants are for indexing to specific elements of the file tables presented above. The constants are broken down into the following sections, GEOS file types, standard Commodore file types, directory header, directory entry, file header, and disk constants.

Disk Variables

When an application first gets called there is already some information waiting for it. Several variables maintained by the deskTop for its own use are still available to the application when it is run. Other variables are set up by the deskTop in the process of loading the application. This subsection covers all the variables an application may expect to be waiting for it when it is first run. This information set up for desk accessories is slightly different. For more details on running desk accessories see the routines **GetFile** and **LdDeskAcc** later in this chapter.

Several variables necessary to talk to the drive are available to the application. The variable **curDrive** contains the number of the drive containing the application's disk, either 8 or 9. When first run, the ID bytes for the disk containing the application are in the drive as one might expect.

Numerous variables are set up during the process of loading an application. The first group of these have to do with how the application was selected by the user. If the user double clicked the mouse pointer on a data file, GEOS will load the application and pass it the name of the data file. The application may then know which data file to use. A bit is set in **r0L** to indicate if a datafile has been specified. If this is the case, **r3** will point to the filename of the data file, and **r2** will point to a string containing the name of the disk which contains the data file. An application may have also been run merely in order to print a data file. Another bit is used in **r0L** to indicate this.

r0L - load option flag

Bit 7 (application files only)
- 0 - no data file specified
- 1 - (constant for this bit is ST_LD_DATA) data file was double-clicked on and this application is its parent.

Bit 6 (application files only)
- 0 - no printing
- 1 - (constant for this bit is ST_PR_DATA) The deskTop sets this bit when the user clicked on a data file and then selected print from the file menu. The application prints the file and exits.

r2 and **r3** are valid only if bits 1 and/or 6 in **r0L** are set.

r2 - Pointer to name of disk containing data file. Points to **dataDiskName**, a buffer containing the name of the disk which in turn contains a data file for use with the application we are loading. The application can then process the data file as indicated by bit 6 of **r0L**.

r3 - Pointer to data filename string. **r3** contains a pointer to a filename buffer, **dataFileName** that holds the filename of the data file to be used with the application.

The directory entry, directory header and the file header block are also available in memory.
dirEntryBuf - Directory entry for file.
curDirHead - The directory header of the disk containing the file.
fileHeader - Contains the GEOS file header block.

There is also a BLOCKSIZE table created as the application file is read.
fileTrScTab - List of track/sector for file. Max file size is 127 blocks (32,258 bytes).

r5L - Offset from the beginning of **fileTrScTab** to the last track/sector entry in **fileTrScTab**

We now turn to discussing the actual routines used to access the disk. The next section presents an overview of how to use the disk routines, and how to use the serial bus with GEOS.

Using GEOS Disk Access Routines

The GEOS Kernal contains a multitude of disk routines. These routines span a range of uses, from general powerful routines, to specific primitive routines. Most applications use only a handful out of the collection, mostly the general high-level routines. Other applications need more exacting level of disk interaction and so an intermediate level of disk access routine is provided. These are routines used by the high-level routines to do what they do, and can be used to create other functions.

Finally, the most primitive routines are interesting only to those who want to access a serial device other than a printer or disk drive, use the C64 DOS disk routines, or create a highly custom disk routine, a nonverified write for example.

Basic Disk Access

When running GEOS, only one device at a time may be selected on the serial bus. Usually this is one of the disk drives, A or B, but it may also be a printer or other device. The routine **SetDevice** is used to change the currently selected drive. You pass **SetDevice** the number of the drive, (8 or 9) for the drive you want to have access to the serial bus.

After selecting the drive with **SetDevice**, call **OpenDisk** to initiate access to the disk. **OpenDisk** initializes both the drive's memory and various GEOS Kernal variables for accessing files on the disk.

Once the disk has been opened, the programmer may call any of the following:

high-level Disk Routines		Page
DeleteFile	Delete file.	20-13
EnterDeskTop	Leave application and return to GEOS deskTop.	20-15
FindFile	Search for a particular file.	20-20
FindFTypes	Find all files of a particular GEOS type.	20-21
GetFile	Load GEOS file.	20-31
GetPtrCurDkNm	Return pointer to current disk name.	20-38
OpenDisk	Open disk in current drive.	20-48
RenameFile	GEOS disk file.	20-58
RstrAppl	Leave desk accessory and return to calling application.	20-59
SaveFile	Save Memory to create a GEOS file.	20-60
SetDevice	Establish communication with a new serial device.	20-62
SetGEOSDisk	Convert normal CBM disk into GEOS format disk.	20-65

For VLIR Routines, see **"VLIR files"** Later in this chapter.

mid-level and low-level Routines

The routines above handle many of the functions required of an operating system, but by themselves are by no means complete. These high-level routines are implemented on top of a functionally complete set of intermediate-level routines that may be used to implement any other function needed. For example, there are no routines for formatting disks, copying disks, or copying files in the GEOS Kernal. Most applications have little need for copying disks or files and so these functions were not included in the Kernal. Instead, these functions are provided by the deskTop. The deskTop is an application like any other such as geoWrite or geoPaint, except that the deskTop is a file manipulation application, and not an editor. The copy and validate functions available in the deskTop are implemented by using the intermediate GEOS Kernal routines.

Care must be taken when using these routines to make sure that all entry requirements are met before calling them. Calling one of these routines without the proper variables and/or tables set up may trash the disk, crash the system, or both. In particular, a block is set aside in the GEOS Kernal to contain a copy of the disk's Directory Header. Some of the routines expect **curDirHead**, to be valid, and if any values were changed by the routine it will be necessary to write the header back to disk afterwards. Below is a list in decreasing order of usefulness of these more primitive routines.

Name	Description	Page
GetBlock	Read single disk block into memory.	20-27
GetBufBlock	Read single disk block into **diskBlkBuf**	20-28
PutBlock	Write single disk block from memory.	20-50
PutBufBlock	Write single disk block from **diskBlkBuf**.	20-51
GetFHdrInfo	Read a GEOS file header into **fileHeader**.	20-30
ReadFile	Read chained list of blocks into memory.	20-55
WriteFile	Write chained list of blocks to disk.	20-71
ReadByte	Read a File 1 byte at a time.	20-54
GetDirHead	Read directory header into memory.	20-29
PutDirHead	Write directory header to disk. (Updates BAM)	20-52
NewDisk	Initialize a drive.	20-45
LdApplic	Load GEOS application.	20-40
LdDeskAcc	Load GEOS desk accessory.	20-42
LdFile	Load GEOS data file.	20-44
GetFreeDirBlk	Find an empty directory slot.	20-34
AllocateBlock	Mark a disk block as in-use.	20-6
BlkAlloc	Allocate space on disk.	20-8
NxtBlkAlloc	Version of **BlkAlloc** that starts at a specific block.	20-46
SetNextFree	Search for nearby free disk block and allocate it.	20-66
FreeBlock	Mark a disk block as not-in-use in BAM.	20-24
SetGDirEntry	Create and save a new GEOS directory entry.	20-63
BldGDirEntry	Build a GEOS directory entry in memory.	20-7
FollowChain	Follow chain of sectors, building track/sector table.	20-23
FastDelFile	Quick file delete (requires full track/sector list).	20-18
FindBAMBit	Get allocation status of particular disk block.	20-19
FreeFile	Free all blocks associated with a file.	20-25
Get1stDirEntry	Get first directory entry.	20-26
CalcBlksFree	Calculate total number of free disk blocks.	20-10
ChkDkGEOS	Check if a disk is GEOS format.	20-12
GetNxtDirEntry	Get directory entry other than first.	20-36
GetOffPageTrSc	Get track and sector of off-page directory.	20-37
StartAppl	Warmstart GEOS and start application in memory.	20-68

Very Low-Level Primitive Routines

An even more primitive level of routines is also available. There are only three reasons one might have for using these routines:

1. To access the standard C64 DOS routines. As mentioned before, the deskTop does this to access the formatting routines.
2. To talk to a device other than the disk drive or printer.
3. To write highly optimized disk routines for moving large numbers of blocks around that are ordered on the disk in some unusual way. The routines in the previous sections for reading and writing a linked chain of blocks on disk are almost always sufficient.

These are all ways you might want to use the serial bus that are outside the realm of what GEOS supports directly. The low-level routines below are provided to allow safe access to the serial bus, and a safe return to GEOS disk usage:

Name	Description	Page
InitForIO	Turn off all interrupts, disable sprites, bank switch the C64 Kernal and I/O space in.	20-39
DoneWithIO	Restore interrupts, enable sprites, and switch in the previous RAM configuration.	20-14
EnterTurbo	Uploads the turbo code to the drive and starts it running.	20-16
ExitTurbo	Deactivate disk turbo on current drive.	20-17
PurgeTurbo	Normally the turbo code is always running. **PurgeTurbo** removes the turbo code resident in the disk drive and returns control of the serial bus to the C64 DOS.	20-49
ReadBlock	Read a block from disk. Turbo code must already be running, and **InitForIO** must have been called.	20-53
WriteBlock	Write a block to disk. No verify is done, the Turbo code must be running, and **InitForIO** must have been called.	20-70
VerWriteBlock	Same as **WriteBlock** except that the block is verified after writing.	20-69
ReadLink	Read track/sector link.	20-57
ChangeDiskDevice	Change disk drive device number.	20-11

Accessing the Serial Bus

Follow the procedure below to use the C64 serial bus:

1. Call **SetDevice** to set up the device you want to use. **SetDevice** will give the serial bus to whatever device you request.
2. If you want to use C64 DOS disk routines, then you will have to turn off the disk turbo code running in the drive. To do this, call **PurgeTurbo**. If not using the C64 DOS routines skip this step.
3. Call **InitForIO** to turn off interrupts, sprites and set the I/O space and C64 Kernal in.
4. Call any of the standard C64 DOS serial bus routines to access the serial device on the bus.
5. When finished with the bus, call **DoneWithIO**. This sets the system configuration back to what it was before you called **InitForIO**. The next GEOS disk routine that you call (except for **ReadBlock**, **WriteBlock**, or **VerWriteBlock**) will automatically restart the diskTurbo.

VLIR Files

File Structure

The VLIR file structure was created to allow applications to grow much larger than the 30k available to them in GEOS. With a faster 1541 disk speed, it becomes practical to break an application up into several different modules, and swap them in as needed. A good way to organize such an application is to keep one module always resident while the others share a common memory area. The resident module is allowed to call subroutines in any of the other swap modules but the other modules may only call routines in the resident module. This keeps the application from getting bogged down with endless swapping. Applications tend to execute out of one module for a while, and then swap modules and execute out of another for a while.

Records

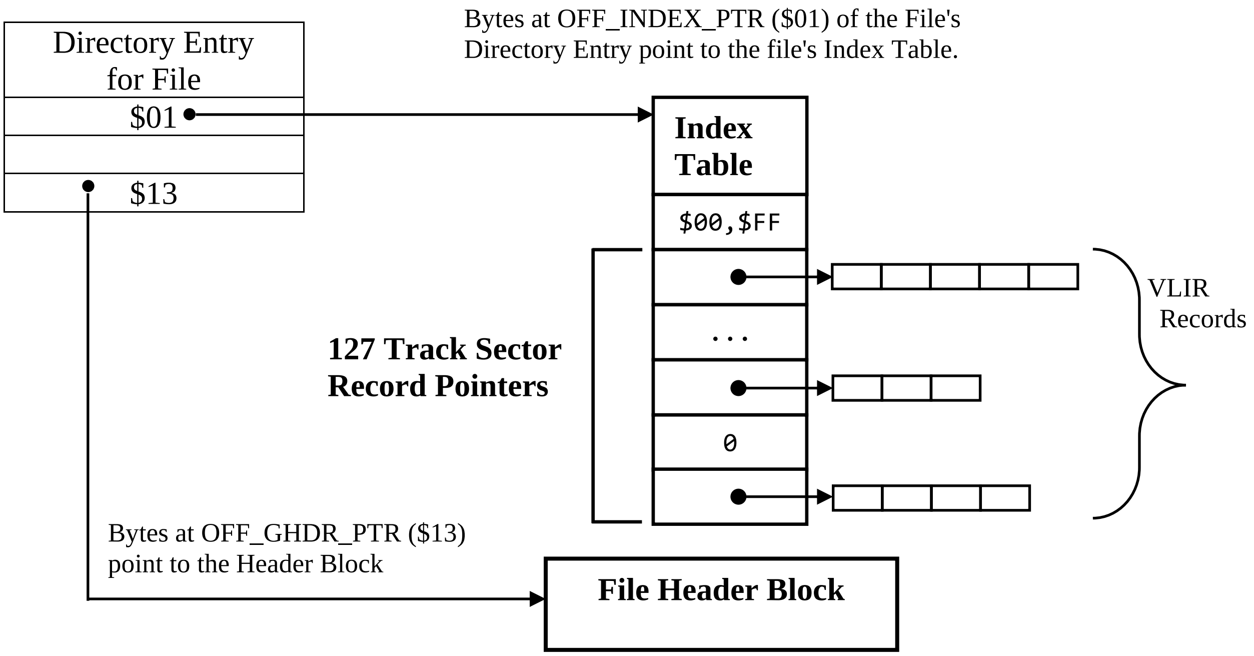
A VLIR file is comprised of several modules referred to as records. Each record, is a chained link of blocks just like a regular Commodore file. Thus, a VLIR file is somewhat like a collection of files. The same routines used to save a regular SEQUENTIAL file to disk may be used to save individual records in a VLIR file. In addition, several VLIR specific routines are provided.

The VLIR file routines allocate sectors on disk for records the same as is done for regular files, using the one block track/sector allocation table, **fileTrScTab**. Each record may therefore be from 0 to 127 blocks long, (just under 32k: 32,258 bytes), the maximum number of track/sector pointers **fileTrScTab** can hold. If the application uses the background screen buffer for program space, it has the use of memory from $400 to $8000 which is also just under 32k. An Index Table, holds the track/sector pointers to the first block in each record. The diagram below shows how the VLIR file uses an Index Table to organize the records in the file.

A VLIR file can be identified by looking at the GEOS Structure type byte in the file's Directory Entry. In addition, the Directory Entry contains a track/sector pointer to the file's Index Table. In a regular SEQUENTIAL file this word points to the first data block in the file. See the beginning of the file system section for more details on the Directory Entry structure. The Index Table consists of 127 entries, numbered 0 to 126, where each entry is a pointer to a record. The rest of the entries in the Directory Entry, such as the pointer to the Header Block, are the same.

Note: VLIR is an acronym for Variable Length Indexed Record. Both applications, and data files may be stored in VLIR format. For example, the font files are divided into several records, one for each point size.

VLIR - Variable Length Indexed Record File Structure



Bytes at OFF_INDEX_PTR ($01) of the File's Directory Entry point to the file's Index Table.

Bytes at OFF_GHDR_PTR ($13) point to the Header Block

VLIR Routines

The routines for reading and writing records, closely resemble those one might expect for manipulating objects in a linked list: **NextRecord**, **PreviousRecord**, and others.

This "linked list" concept makes use of a pointer to the current record. This pointer may be set directly or set to the next or previous record. The current record may be deleted, read from, or written to. At each access, the full record must be dealt with. Thus, the application should provide sufficient RAM at any one time to accommodate the largest possible record it could be processing. New empty records may be inserted before, or appended after the current record. New records are empty and may be written to. Presently there is no way to detach a record and re-attach it somewhere else (This would be a trivial task for an application to handle on its own). **DeleteRecord** is destructive, i.e., frees up the sectors, and **InsertRecord** only works with empty records.

The index table may be stored in memory, often in the **fileHeader** buffer, to make it possible to go directly to a record using **PointRecord** instead of advancing one record at a time with **NextRecord** or **PreviousRecord**.

Description of the routines available specifically for VLIR files:

Name	Description	Page
AppendRecord	Insert a new VLIR record after the current record.	20-73
CloseRecordFile	Close/Save currently open VLIR file.	20-74
DeleteRecord	Delete current VLIR record.	20-75
InsertRecord	Insert new VLIR record in front of current record.	20-76
NextRecord	Make next VLIR the current record.	20-77
OpenRecordFile	Open VLIR file on current disk.	20-78
PointRecord	Make specific VLIR record the current record.	20-79
PreviousRecord	Make previous VLIR record the current record.	20-80
ReadRecord	Read current VLIR record into memory.	20-81
UpdateRecordFile	Update currently open VLIR file without closing.	20-82
WriteRecord	Write current VLIR record to disk.	20-83

An attempt has been made to return meaningful error flags concerning operations on the structure. The following is a list of possible errors as returned in the x register by VLIR Record routines.

Error Messages

UNOPENED_VLIR
This error is returned upon an attempt to Read/Write/Delete/Append a record of a VLIR file before it has been opened with **OpenRecordFile**.

INV_RECORD
This error will appear if an attempt is made to Read/Write/Next/Previous a record what doesn't exist (isn't in the Index Table). This error is not fatal, and may be used to move the Record pointer to the end of the record chain.

OUT_OF_RECORDS
This error occurs when an attempt is made to Insert/Append a record to a file that already contains the maximum number of records allowed (127 currently).

STRUCT_MISMATCH
This error occurs when a routine supporting a function for one type of file structure is called to operate on a file of different type.

Creating a VLIR File

Use the **SaveFile** routine to initially create a VLIR file:

The File Header should contain the following values:

Offset $00.	Pointer to VLIR filename (word).
C64 file type	- USR
GEOS File Structure Type	- VLIR

For Creating an empty VLIR File:

Start address:	0
End address:	FFFF (-1)

For Saving Data into a new VLIR file with Record 0 populated:

Start address:	Start address of data to save.
End address:	End address of data to be saved.

This creates a VLIR file on disk with an Index Table with no records. The current record pointer is set to -1: a null pointer. Before any manipulation of the file is possible, it must be opened with **OpenRecordFile**. This loads certain internal buffers GEOS needs. With a completely empty record file like this, the first record must be created with **AppendRecord**. After that, calls to **InsertRecord**, and **DeleteRecord** are possible.

When through with the file, it is imperative that the programmer close it by calling **CloseRecordFile**. This will update the file's index table, the disk BAM, and the "blocks used" entry in the file's directory entry. Note that only one VLIR file may be opened at time.

Input Driver

The Standard Driver

GEOS currently supports the joystick (the standard driver), a proportional mouse and a graphics tablet. On the screen, the position of the joystick or mouse is shown by an arrow cursor. We shall use the terms, mouse, pointer, and cursor, interchangeably to refer to the arrow cursor on the screen. We shall use the term device to denote the actual hardware.

Each Interrupt, the GEOS Kernal Interrupt Level code calls the input driver. The job of the input driver is to compute the values of the following variables.

```
mouseXPos:
       .block 2      ; word x-position in visible screen pixels of the mouse pointer (0-319)

mouseYPos:
       .block 2      ; byte y-position in visible screen pixels of mouse pointer (0-199)

mouseData:
       .block 1      ; byte set to nonnegative if fire-button pressed, negative if released

pressFlag:
       .block 1      ; byte bit 5 (MOUSE_BIT) set if a change in the button
                     ;      bit 6 (INPUT_BIT) if any change in input device since last interrupt
```

Both the GEOS Kernal and applications may then read and act on these variables. The GEOS Kernal reads bit 5 (MOUSE_BIT) in the **pressFlag** variable to determine if there has been a change in the mouse button. If there has been a change, then the Kernal reads **mouseData** to determine whether the change is a press or release. If the mouse button has been pressed (indicated by **mouseData** changing from negative to nonnegative) then GEOS will check to see whether the mouse position is over a menu, an icon, or screen area. If it is over a menu, then the menu dispatcher is called. If it's over an icon, then the icon dispatcher is called. If it's not a menu or icon then the routine in **otherPressVec** is called.

If the joystick changes from being pressed to being released (**mouseData** has a negative value) then the Kernal will vector through **otherPressVec**. **Note**: all releases are vectored through **otherPressVec**, even if the original press was over a menu or icon. The application's **otherPressVec** routine must be capable of screening out these unwanted releases. The reason that the mouse acts like this is that the ability to detect releases was added relatively late to the GEOS Kernal. The menu and icon modules were already complete. **otherPressVec** is called on all releases including those for menus and icons so that its routine can take special action on those releases as well as its own, if necessary. Usually, the application's **otherPressVec** routine will either ignore releases altogether, or only act on releases following screen area presses.

What an Input Driver Does

It is the job of the input driver to read the hardware bytes it needs to load **mouseData** and **pressFlag** with the proper values. It must determine the change in the position of the mouse and store new values in **mouseXPos** and **mouseYPos**.

Different input drivers compute the mouse x, y-position in entirely different ways. As an example, the joystick driver does this by first reading the joystick port, and then computing an acceleration from the direction the joystick was pressed. From that, a velocity, and finally a position is determined. A proportional mouse is entirely different. The Commodore mouse sends differing voltage levels to the potentiometer inputs in the joystick port and the SID chip in the C64 reads the voltage level and stores an 8-bit number for both x and y. The driver computes a change in position from the voltage level as reflected by the value of the two bytes. No matter how it is done, though, the input driver is responsible for setting the 4 variables mentioned above.

Location and Responsibilities of Input Driver

The code for the joystick input driver takes up the 380 bytes beginning at MOUSE_BASE, the area from $FE80-FFF9. GEOS 128 uses MSE128_BASE, the area from $FD00-FE7F

When an alternate input driver such as a graphics tablet is loaded by the deskTop, it is installed at this location. If you write an input driver, it should be assembled at this address. All GEOS applications will expect three routines, **InitMouse**, **SlowMouse** and **UpdateMouse**, and the four variables mentioned above to be supported by any input driver. These three routines should perform the same function, regardless of the input device. This way the particular application running need know nothing about which input driver the user has chosen. These routines may begin anywhere within the input driver area just so long as a short jump table is provided right at the beginning of the input driver space:

Address	Contents	
`MOUSE_BASE`	`jmp` **`InitMouse`**	
`MOUSE_BASE + 3`	`jmp` **`SlowMouse`**	
`MOUSE_BASE + 6`	`jmp` **`UpdateMouse`**	
`MOUSE_BASE` **`+ 9`**	`jmp` **`SetMouse`**	`; 128 Driver has 1 additional jump table entry`

These are the addresses that the GEOS Kernal and applications will actually call. For example, to call **UpdateMouse**, the Kernal will do a jsr MOUSE_BASE + 6 during Interrupt Level. The first routine the input driver must provide is **InitMouse**. It is called to perform any initialization, and set any variables, the driver needs before the other two routines are called.

Note: **SetMouse** does not exist in GEOS 64 Input Drivers. If a 128 Input Driver does not need to use **SetMouse**, then place an rts at MOUSE_BASE + 9 instead of a jmp entry.

Acceleration, Velocity, and Nonstandard Variables

Some input devices, such as the joystick, need to be adjusted for different sensitivities. For example, sometimes the user will want the joystick to accelerate to its maximum velocity quickly. Other times, such as when opening a menu, the user will want it to move more slowly so as to make it easier to select an item without slipping off the menu altogether.

Other devices such as proportional mice and graphics tablets do not make use of acceleration and velocity. These devices deal more directly with position and distance moved. Still other devices as yet uninvented may need special variables of their own. The question arises how to best support different input devices in a way that the application need not know which device is being used, and yet leave room for new devices. There are three parts to the solution.

First, there is a basic level that every input drive should be able to support. This includes maintaining the position variables **mouseXPos**, and **mouseYPos**, and the mouse button variables, **pressFlag**, and **mouseData**. At the very least, an input driver must generate values for these variables.

Second, additional variables for joystick-like devices, are allocated in the GEOS Kernal RAM space. The joystick is the default driver for GEOS, and needs to keep track of acceleration and velocity variables. These variables include **maxMouseSpeed**, **minMouseSpeed**, and **mouseAccel**. These variables are loaded with default values by the driver's initialization routine, and are located in GEOS Kernal RAM area so that they may be used by the preference manager to adjust the speed of the mouse. There is also a routine, **SlowMouse** that is called by the GEOS Kernal itself to slow the mouse down during menu selection. This routine is presented below. Together this routine and these variables allow a high level of control over joystick behavior. This may seem like a lot of effort to spend on a joystick, but considering that most users will be using a joystick, such effort is appropriate.

Different devices like Commodore's proportional mouse do not require any special treatment. It is not based on velocity, but on distance. Its motion is precise enough to make fine tuning unnecessary. It is possible that some as yet unknown input device may become available that does require special treatment. In this case a third approach may be used.

This approach is to augment the regular position and button variables with four bytes beginning at the label **inputData** in the Kernal RAM. These variables may be used to pass additional values to an application. Any input device that needs to pass parameters to an application other than the position, mouse button, or velocity and acceleration variables, should pass them here. **Note**: Applications which rely on **inputData** become device dependent.

Whenever the input state has changed, the driver must:

1. update the 4 mandatory mouse variables;
2. update **inputData**, if supported;
3. the INPUT_BIT, (bit 6) should be set in **pressFlag**.

In addition, an application that uses **inputData** must load the vector **inputVector** with the address of a routine that retrieves values from **inputData**. When the Kernal sees the INPUT_BIT set, it will vector through **inputVector** if it is nonzero. As an example, the joystick driver loads a value for the direction in the first of these four bytes and the current speed of the mouse in the second. geoPaint uses these values in its routine to scroll the drawing. When in scroll mode, geoPaint sets **inputVector** with the address of a routine used in scrolling. Whenever the direction of the joystick changes, **inputVector** is vectored through and the geoPaint scroll routine stops or changes the direction of the scrolling.

This use of these variables is probably unfortunate because although they are natural to generate for the joystick, they are not so natural to generate for other drivers, such as proportional mice. The drivers for these devices must generate these direction values by hand so that they will completely work with geoPaint.

Note: The only reason for using **inputData** is to support a special input device that communicates in a custom fashion with its own application. As this can cause incompatibility with other input devices and other applications, this approach should be used sparingly. An application can check the variable string **inputDevName** for the name of the current input device. The deskTop loads the null-terminated filename of the input driver file into this 17-byte string.

The general approach then for supporting a new input driver should be clear. First compute the position and button variables. If geoPaint scrolling is to be supported, direction variables will need to be supported. Finally, some custom tailorable driver support is possible. The variables discussed above are presented in more detail below, after the outlines for **SlowMouse** and **UpdateMouse**.

SlowMouse

The **SlowMouse** routine, as outlined below, sets the joystick speed to zero. The joystick is then free to accelerate again. From its name, one might instead expect **SlowMouse** to reduce the **maxMouseSpeed**, but this is not the case.

The reason for having a routine like this is to make using menus easier. When a menu opens, and the user slides down the selections and hits the mouse button when over the desired item, the GEOS Kernal will then open a submenu and put the mouse pointer on the first selection of the submenu. The user may then select one of it's items. It was found that almost all users keep the joystick direction pushed until the submenu comes up. By this time the mouse will have reached maximum velocity, and, when placed on the submenu graphic by the application, will go flying off. **SlowMouse** just zeros out the mouse's speed so that this won't happen. Drivers for mice and graphics tablets which don't use velocity need to include this routine even though in this case it will merely perform an rts.

To make the mouse actually slowdown from within an application, **maxMouseSpeed**, and **mouseAccel** can be lowered. The standard values for these variables may be found in the Mouse Variable and Mouse Constant sections later in this section.

UpdateMouse

UpdateMouse is the main routine in an input driver. Its responsibilities include reading the joystick port in order to determine how the input device has changed, and translating this into a change in **mouseXPos**, **mouseYPos**, **mouseData** and **pressFlag**. If geoPaint scrolling is to be supported, then direction information must be returned in **inputData**. If special input driver information is to be passed to an application then **inputData** should again be used.

Mouse Variables for Input Driver

The following variables are supported by the mouse module. Most of these variables have been described briefly above.

Required Mouse Variables

mouseXPos	word	x-position in visible screen pixels of the mouse pointer (0-319).
mouseYPos	byte	y-position in visible screen pixels of mouse pointer (0-199).
mouseData	byte	Nonnegative if fire-button pressed, negative if released.
pressFlag	byte	Bit 5 (MOUSE_BIT) set by driver if a change in the button; Bit 6 (INPUT_BIT) set if any change in input device since last interrupt.

MOUSE_BIT = %00100000
INPUT_BIT = %01000000

Optional Mouse Variables

maxMouseSpeed	byte	Used to control the maximum speed or motion of the input device. In the case of a joystick, **maxMouseSpeed** controls the maximum velocity the mouse can travel across the screen. This variable is unused for graphics tablets and proportional mice. Best values for this byte depend on how the input driver uses this variable to compute current speed and position. For a joystick, legal values are 0-127. Default value is: MAXIMUM_VELOCITY=127 This is the constant for the default maximum velocity to store in **maxMouseSpeed**.

minMouseSpeed	byte	Used to control the minimum speed or motion of the input device. See **maxMouseSpeed** above. Legal joystick values are; 0-127. Default value is: MINIMUM_VELOCITY=30 Minimum velocity to store in **minMouseSpeed**. Anything slower than this bogs down.
mouseAccel	byte	This byte controls how fast the input device accelerates. In the case of a joystick, it controls how fast the joystick accelerates to its maximum speed. In the case of a graphics pad it might scale the distance moved with the pointer on the pad to the distance moved on the screen. Currently this variable is only used by the joystick driver. Legal values are 0-255. Default value is: MOUSE_ACCELERATION=127 Typical acceleration byte value of mouse.
inputVector	word	Contains the address of a routine called from **MainLoop** to use input driver information supplied by unorthodox input devices. The idea here is that some input drivers may be able to produce more information than the x and y-position data for an application that may want to use this info. If **UpdateMouse** supports such extra info it should store it in **inputData** array and set the INPUT_BIT in **pressFlag**. When GEOS **MainLoop** sees this bit set it will call the routine whose address is stored in **inputVector**.
inputData	4 bytes	Used to store device dependent information. For joysticks: inputData:0-7 joystick directions: 0 = right 1 = up & right 2 = up 3 = up & left 4 = left 5 = left & down 6 = down 7 = down & right -1 = joystick centered **inputData**+1: current mouseSpeed

The Mouse as Seen by the Application

To this point, we have discussed input devices as seen from the perspective of a programmer wanting to write an input driver. The other side of the coin is how an application interacts with the input driver. The regular action of the mouse is as described above. Mouse presses are checked for icon, or menu activation, or a press in the user area of the screen.

To start the mouse functioning like this, the routine **StartMouseMode** is called. Since this is done by the deskTop to get itself running, the application need not call **StartMouseMode** itself. To turn mouse functioning off, one calls **ClearMouseMode**. A bit in the variable **mouseOn** is cleared, the sprite for the mouse is disabled (the sprite data is no longer DMA'd for display, important for RS-232, disk, and other time critical applications) and **UpdateMouse** is no longer called during interrupt level. This is the reason the mouse pointer flickers during disk accesses: **ClearMouseMode** is called by the disk turbo code. To restore mouse functioning after a call to **ClearMouseMode**, call **StartMouseMode**.

To temporarily turn the mouse picture off, but have its position and **inputData** variables still set, call **MouseOff**. **UpdateMouse** in the input driver is still called, just the sprite for the mouse, sprite 0 is disabled. To turn the mouse on again, call **MouseUp**. **MouseUp** reenables the mouse sprite and causes the mouse to be redrawn the next interrupt in case the mouse had been moved since being turned off. To temporarily disable the mouse, call **MouseOff** and then **MouseUp**.

Additional Mouse Control

GEOS allows you to limit the movement of the mouse to a region on screen. The GEOS Kernal will constrain the mouse within a rectangle defined by two word length variables, **mouseLeft**, and **mouseRight**, and two byte length variables, **mouseTop**, and **mouseBottom**. The input driver needs know nothing about these variables. After it updates **mouseXPos**, and **mouseYPos**, the Kernal will check to see if the new position is out of bounds, and if necessary force its position back to the edge of the rectangle. The Kernal will also vector through **mouseFaultVec**. This vector is initialized to zero by the Kernal. The application may load **mouseFaultVec** with the address of a routine to implement, for example, scrolling a document under the screen window. The effect would of the screen scrolling whenever the user drew the mouse pointer off the edge of the screen.

There is also a routine for checking to see if the mouse pointer is within a certain region on screen. This routine is quite useful if clicking inside a box or other region is to have special significance in your application. This routine is called **IsMseInRegion** and you pass it the coordinates of the sides of the rectangular region you want it to check.

A couple of more mouse variables are used. **mousePicData** contains 64 bytes for the sprite picture of the mouse, while **mouseVector** contains the address of the routine **MainLoop** calls to handle all mouse functioning. If the MOUSEON_BIT of **mouseOn** is set, then every time the input driver indicates the mouse button has been pushed, **mouseVector** is vectored through. It is unclear why the programmer might want to change **mouseVector**, as this would disable icon and menu handling. **otherPressVec** is more likely the vector to change.

mouseOn also contains bits for turning menu and icon handling on and off. Unfortunately, a call to the menu handling routine will serve to turn the icon enable bit on upon its exit. This is the reason a dummy icon table is necessary for those programs running without icons.

Mouse Variables for Applications

The following variables are supported by the mouse module in the GEOS Kernal for application use.

mouseOn

byte A flag which contains bits determining the status of the mouse and menus. Also contains bits used by the Menu and Icon modes.

bit 7 Mouse On if set
bit 6 Set if Menus being used (should always be 1)
bit 5 Set if Icons being used (should always be 1)

```
SET_MSE_ON          = %10000000  Bit set in mouseData to turn mouse on
SET_MENUON          = %01000000  Bit set in mouseData to turn Menus on
SET_ICONSON         = %00100000  Bit set in mouseData to turn Icons on
MOUSEON_BIT         = 7          The number of bit used to turn mouse on
MENUON_BIT          = 6          The number of bit used to turn on menus
ICONSON_BIT         = 5          The number of bit used to turn on icons
```

mouseLeft

word mouse cursor not allowed to travel left of this programmer set position. Legal range is 0-319.

mouseRight

word Mouse cursor not allowed to travel right of this pixel position on screen. Legal range is 0-319.

mouseTop

byte Mouse cursor not allowed to travel above this pixel position on screen. Legal range is 0-199.

mouseBottom

byte Mouse cursor not allowed to travel below this pixel position on screen. Legal range is 0-199.

mousePicData

bytes Sprite picture data for mouse cursor picture. This area is copied into the actual sprite data area by the GEOS Kernal.

mouseVector

word Routine called by GEOS Kernal when mouse button pressed.

mouseFaultVec

word Routine to call when mouse tries to go outside of **mouseTop**, **mouseBottom**, **mouseLeft**, and **mouseRight** boundaries. GEOS will not allow the mouse to actually go outside the boundaries.

Sample Joystick Driver

A Complete driver ready to build has been included to show how all the content of this chapter come together. See "**Joystick Driver**" in Examples\Drivers.

Example: **Joystick Driver**

Printer Drivers

This chapter is intended for:

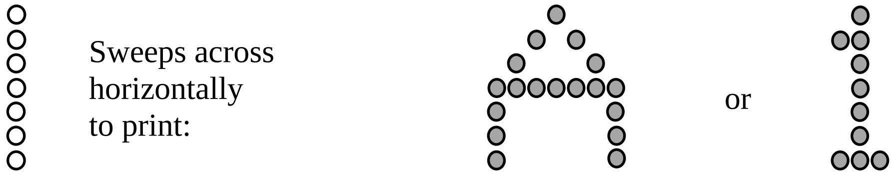
1. programmers who want to use GEOS printer drivers with their applications,
   or
2. programmers who want to write a GEOS printer driver for a previously un-supported printer.

The State of Printers

There is such a multitude of different printer types on the market today that several books could be written about their operation. In fact, several have. To find out about a specific printer or interface card consult the operator's manual or visit the local computer store.

There are two basic categories of printers: "character" (typewriters, daisywheel, band printers, etc.), and dot-matrix printers. Character printers are only capable of printing character shapes that are physically on the print wheel (band, ball, or hammers). In general, this makes them unsuitable for use with GEOS since GEOS stores and prints both character fonts and graphics as a bit map. GEOS does support a near letter-quality print mode for the 1526 Commodore printer, but to use GEOS as it was intended to be used requires a dot-matrix printer. Dot-matrix printers are constructed with vertical lines of pins which can be individually controlled to strike the ribbon (or squirt the ink, in the case of an ink-jet printer, which also falls into the dot-matrix category) onto the paper. The device holding these pins is called the printhead. As the printhead moves across the page, different dot-columns are printed, leaving a two-dimensional pattern (matrix) of dots. Individual characters are patterns of adjoining dots on the page as in the illustration below:

Printhead Character Matrix



ASCII and Graphic Printing

Dot matrix patters usually operate in two modes. In the first, ASCII mode, an application feeds the printer ASCII character codes and the printer prints from its own internal character set. In its own memory it stores the dot pattern for all the letters. In addition to this first mode there is the ability to send the printer the actual dot patterns to print.

The printer's internal character set is used for draft and near-letter quality (NLQ) modes of printing. In draft mode the application passes the printer driver a string of regular ASCII (not Commodore ASCII) characters. The printer prints these out in its fast-single strike draft mode using its internal character set. NLQ mode is just like draft mode except that several overstrikes or other methods are used in order to make the printed output sharper.

GEOS uses the graphics mode of the printer for all graphic and most text printing. This is how it is possible to print different fonts. This mode is variously referred to as Graphics Mode, Bit-Image Mode, or APA (All Points Addressable) Graphics Mode. This mode interprets bytes in the print buffer not as ASCII characters, but as bit patterns (vertically oriented) for the printhead to print. The example below shows how a typical printhead might be addressed in graphics mode. Each pin on the printer is assigned a bit. The "Dot Columns as Printed" columns show the value passed to the printer and the image it produces.

Bit Value	Printhead	Dot Columns as Printed															
		01	02	04	08	10	20	40	80	AA	55	00	3C	42	81	81	42
$01 %00000001	○	●									●				●	●	
$02 %00000010	○		●							●				●			●
$04 %00000100	○			●							●		●				
$08 %00001000	○				●					●			●				
$10 %00010000	○					●					●		●				
$20 %00100000	○						●			●			●				
$40 %01000000	○							●			●			●			●
$80 %10000000	○								●	●					●	●	

Dot Matrix Printer Types

There are two general categories of printheads around today: 9-pin and 24-pin. 9-pin printheads use the top 7 or 8 pins to actually print in graphics mode. The bottom one or two pins are used to print descending characters. These are ASCII characters like "g" and "p" that have tails below the print line. Whether 7 or 8 pins are used to print graphics is also dependent on the printer itself. Bit 0 may be at either the top or the bottom pin, depending on the individual printer. Since 8-bit data is easier for an 8-bit computer to handle than 7-bit data, having to spoon feed a printer 7-bit wide data can be tedious. As a bit of foreshadowing let us mention this will be discussed in more detail later when we discuss the print algorithms. Presently we continue with a general printer description.

Typically, the pins make a 1/72" x 1/72" dot, spaced 1/72" apart vertically. Dot-columns are spaced at 1/60", 1/72", 1/80", or even closer depending on the printer and the mode in which it is running. 24-pin printers work basically the same way the 9-pin printers do, except at a higher resolution (24 pins in the same area as the 9 and a correspondingly higher horizontal resolution).

Printers enter and exit graphics mode one of two ways: some are given a command to enter graphics mode and stay that way until a command is given to exit graphics mode. Others are given the command to enter graphics mode, followed by a byte count. Until the count reaches zero, every byte that the printer sees is printed out in graphics mode.

Once the program is capable of individually firing pins on the printhead, the only thing preventing it from printing a whole page of solid graphics is the control of how far the printer line-feeds when told to do so. Fortunately, every printer that has a graphics mode, also has the ability to be told how far to advance the paper when a LF is encountered. The first step in understanding printing in either ASCII or graphics mode is to learn how to communicate with the printer. Most printing is done through the C64's serial port. An exception to this is geoCable by Berkeley Softworks which allows you to run any Centronics parallel printer from the user parallel port with GEOS. The following section deals with the C64 serial bus interface to the printers.

Talking to Printers

This section describes the way the serial bus works, the routines in the C64 Kernal ROM used to communicate with peripheral devices, and the types of interfaces available for parallel input printers.

The C64 communicates with its peripheral devices (disk drives, printers, etc.) over a serial bus. The serial bus supports up to five devices connected at once in a daisy-chain fashion. There are three basic types of activity on

Note: For more information on the serial bus and how it works, see the Commodore 64 Programmer's Reference Guide (pp 362-366).

the serial bus, "control", "talk", and "listen". The C64 is the controller of the bus, and can tell peripheral devices when to "talk" (to output data onto the bus) or when to "listen" (accept input from the bus). The devices are assigned unique addresses which are output on the bus when a control signal from the C64 is sent out. These "addresses" are single byte numbers based on device type. All serial printers are assigned the number 4. To work with the C64, a printer must recognize a 4 on the serial bus as its "address" and react to the next byte which is one of several possible command bytes. It can be any valid command byte that the device recognizes. This second byte is called the secondary address.

The C64 Kernal ROM has routines resident within it to operate the serial bus. These routines "talk", "un-talk", "listen", "un-listen", send secondary addresses, and receive and send data on the serial bus. These routines are called with device addresses (if needed for the routine) in the accumulator, and return error codes in the accumulator. The Kernal routines set the carry flag to indicate that the value in the accumulator is a valid error code and not just left-over garbage. These primitive routines are used by printer drivers to set up transmission of data over the serial bus to the printer.

Parallel Interface Questions

Since many of the higher quality printers available are not equipped with interfaces for the Commodore serial bus (most have Centronics parallel interfaces), the user must either use the geoCable printer cable and geoCable printer drivers, or use a serial-to-parallel interface that recognizes the Commodore serial bus protocol and the Centronics standard. Fortunately, a few such devices exist, and are readily available to the consumer at major retailers. Some of these are: Cardco G-Whiz, Cardco Super-G, and Telesys Turboprint CG.

Note: For more information on the Kernal ROM routines, see the Commodore 64 Programmer's Reference Guide (pp 270-304).

GEOS Printer Drivers

Now that we have covered the basics of printer operation, we proceed to printer driver operation. In order for all applications to be able to talk to all printer drivers, two things are necessary.

1. All applications must see a single general interface standard.
2. A driver must be written for each functionally different printer that takes the application's output, and tailors it to a specific printer.

The application is responsible for one half of the work and the printer driver for the other half.

The Interface - For Graphic Printing

Printer drivers and applications pass data through a 640-byte buffer. This buffer is sized to hold eight scanlines of 80 bytes per scanline resolution. This is the maximum line width supported by GEOS. (Some applications may not support the entire width of a GEOS page. For example, geoWrite versions prior to 2.1 only support 60 bytes across. In this case the application must put out blank bytes on either end of the buffer line).

What this amounts to is the application assembles a buffer of graphics data in hi-res bitmap mode card format, calls a printer driver routine that reorganizes the data, and sends it over the serial bus. The application's programmer must then know how to format the data, and what routines in the printer driver to call. The printer driver author must implement the standard set of routines to print on a specific printer. This means reordering the bytes significantly since the printer expects bytes that represent vertical columns of pixel data while each byte of data passed in the 640-byte buffer represents eight horizontally aligned pixels. This work is done in four separate callable routines.

GetDimensions:	Return the dimensions in Commodore screen cards of the page the printer can support.
InitForPrint:	Called once per document to initialize the printer. Presently only used to set baud rates.
StartPrint:	Initialize the serial bus at the beginning of every page, and fake an opened logical file in order to use C64 Kernal routines to talk to the printer.
PrintBuffer:	Print the 640-byte buffer just assembled by the application when printing in graphics mode.
StopPrint:	Do end of page handling, a form feed and for 7-bit graphics printing flush the remaining scanlines in the buffer.

The application is in control of the printing process. It calls **InitForPrint** once to initialize the printer. Then **StartPrint** is called to set up the serial bus. After that **GetDimensions** is usually called to find out the width of the printable line and the max number of lines in the page. The application then fills the buffer with bitmap data in card format and calls **PrintBuffer** to print it. As soon as a full page has been printed, **StopPrint** is called to perform the form feed and any other end of page processing necessary. The process begins again on the next page with a **StartPrint**.

ASCII Printing

All ASCII printing is done on a 66 lines per page and 80 character per line basis. The application passes the printer driver a null terminated ASCII string. Any formatting of the document such as adding spaces to approximate tabs should be done by the application. All end-of lines are signaled by passing a carriage return to the driver. The driver will output a CR as well as a linefeed for every CR it receives in order to move the printhead to the beginning of the next line. For some applications, such as geoPaint, a draft or NLQ mode of printing do not make sense. Others, such as geoWrite, will offer draft and NLQ modes of printing for printing text and will skip any embedded graphics in the document.

The procedure for ASCII printing is much the same as for graphic printing. The application calls **InitForPrint** once to initialize the printer. If NLQ mode is desired then **SetNLQ** is called. The application then calls **StartASCII**, instead of **StartPrint** to set up the serial bus. The application may now begin sending lines. It passes a null terminated string of characters, pointed to by **r0**, to **StartASCII**. Spaces used to format the output should be embedded within the string passed to **StartASCII**. A carriage return should be printed at the end of every line.

StartASCII:	same as **StartPrint** except for printing in draft or NLQ modes.
PrintASCII:	Use this routine instead of **PrintBuffer** for draft and NLQ printing. The application passes a null terminated ASCII character string to the driver instead of the 640-byte buffer, and the printer prints in its own charset.
SetNLQ:	Send the printer whatever initialization string necessary to put it into near letter quality mode.

Calling a Driver from an Application

Printer drivers are assembled at PRINTBASE ($7900), and may expand up to $7F3F. Applications must leave this memory space available for the printer driver. In addition, the application must provide space for two 640-byte RAM buffers. The application uses the first buffer to pass the 80 cards (640-bytes) of graphics data to the driver. The driver uses the other internally. These two buffers are pointed at by **r0** and **r1** when a driver routine is called.

At the beginning of each printer driver is a short jump table for the externally callable routines. Once the driver is loaded an application calls printer routines just like any other Kernal routine.

Name	Description	Page
GetDimensions	Get CBM printer page dimensions.	20-176
InitForPrint	Initialize printer (once per document).	20-177
PrintASCII	Send ASCII data to printer.	20-178
PrintBuffer	Send graphics data to printer.	20-179
SetNLQ	Begin near-letter quality printing.	20-180
StartASCII	Begin ASCII mode printing.	20-181
StartPrint	Begin graphics mode printing.	20-182
StopPrint	End page of printer output.	20-183

Using a Printer Driver from an Application

For Graphics Printing:

(A) Call **GetDimensions** to get: (1) the length of the line supported by the printer (constant is CARDSWIDE) usually 80 but sometimes 60, in x, and (2) the number of rows of cards in a page (which is the same as the number of times to call **PrintBuffer**) in y (constant is CARDSDEEP).

(B) Call **InitForPrint** once per document to initialize the printer. Call **StartPrint** once per page to set up the Commodore file to output on the serial bus. Any errors are returned in x and the carry bit is set. If no error was detected, x is returned with $00.

(C) To print out each row of cards (do 1, 2, and 3 for each line) do the following.
  (1) Load a 640-byte buffer with a line of data (80 cards) and load **r0** with the start address of the 640-byte buffer.
  (2) Load **r1** with the start addr of 640-bytes RAM for the print routines to use. Load **r2** with the color to print. Multicolor printers require several passes of the print head. Each in a different color, each with a different set of data. For each line then, **PrintBuffer** is called for each color.
  (3) Call the **PrintBuffer** routine. **Note**: Go to 1 until page is complete.

Note: **r1** must point to the same memory for the whole document, and must be preserved between calls to **PrintBuffer**. **r0** can change each time **PrintBuffer** is called.

(D) Call the **StopPrint** routine after each page to flush the print buffer (if using a 7-bit printer then scanlines left in the buffer pointed to by **r1** need to be printed out rather than combined with the next row of data) and to close the Commodore output file.

Note: CARDSWIDE is the number of Commodore hi-res bit-mapped cards wide.
CARDSDEEP is the number of Commodore hi-res bit-mapped cards deep.

For ASCII Printing:

(A) Call **InitForPrint** once per page to initialize the printer.

(B) Call **SetNLQ** if printing in near letter quality mode is desired.

(C) Call **StartASCII** once per page to set up the Commodore file to output on the serial bus. Any errors are returned in x and the carry bit is set. If no error was detected, x is returned with $00.

(D) To print out each row of cards (do 1, 2, and 3 for each line) do the following.
  (1) Load a buffer with a string of ASCII character data and load **r0** with the start address of the buffer. Append a CR to the end of each line to cause a CR and LF to be output by the printer.
  (2) Load **r1** with the start address of 640-bytes RAM for the print routines to use.
  (3) Call the **PrintASCII** routine. **Note**: Unlike **PrintBuffer**, **r1** need not point to the same memory for the whole document, or be preserved between calls to **PrintASCII**. **r0** can change each time **PrintASCII** is called. Goto 1 until document is complete.

(E) Call the **StopPrint** routine (PRINTBASE + 9) at the end of every page to form feed to the next page, and to close the Commodore output file.

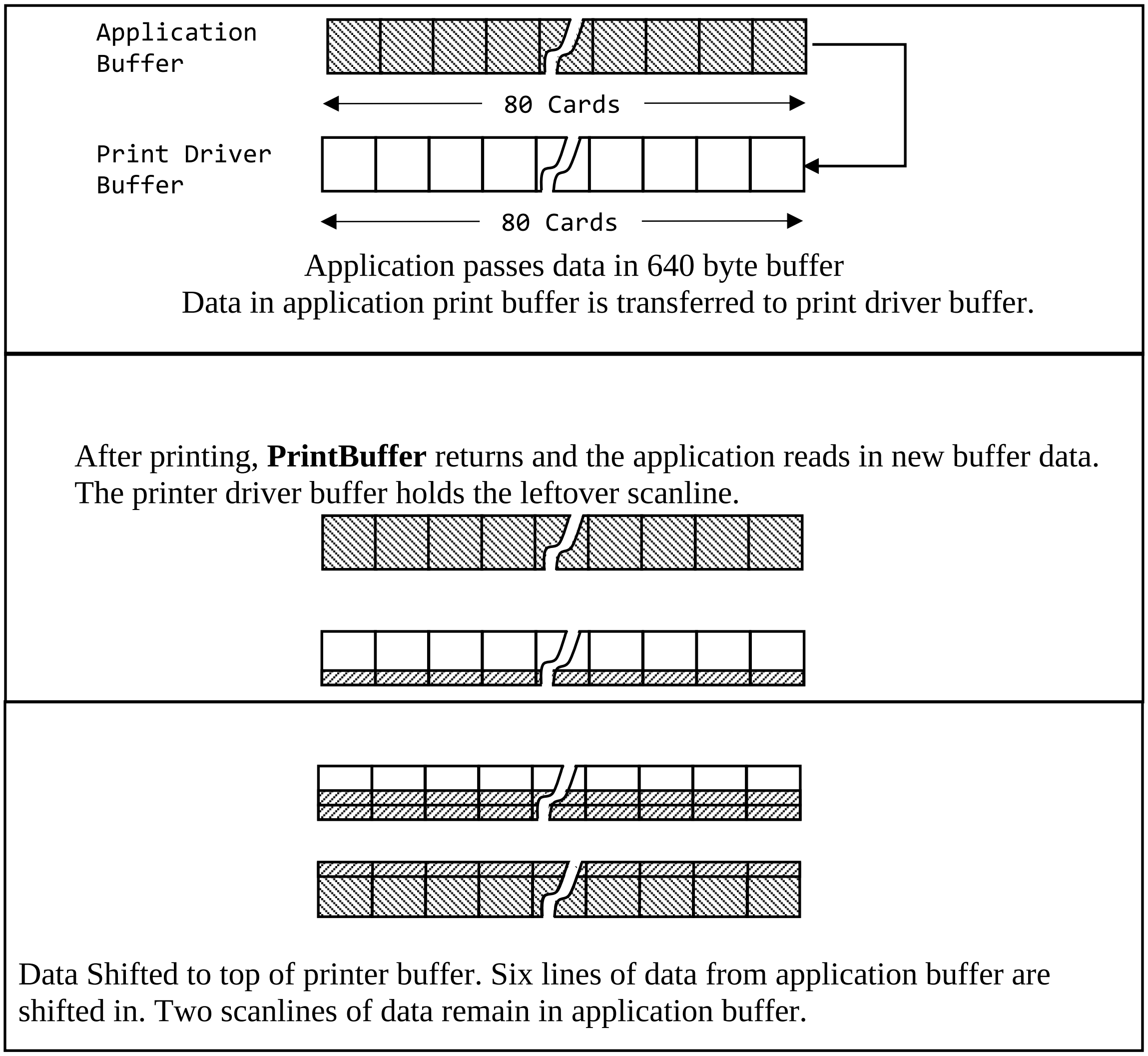
We now describe these routines in greater detail. After this section we present two sample printer drivers. The first is for Commodore compatible printers. This driver is a good model for any 60 dot per inch printer. Following the Commodore driver is the driver for the Epson FX series of printers. This driver is a good model for any 80 dot per inch printer.

SamplePrinterDriver

Introduction to Sample Driver

Two basic printer drivers provide the prototypes for the remainder of drivers in existence, one for 7-bit and one for 8-bit printers. These two types of drivers differ in that the 7-bit high printers can only print out 7 scanlines of data at one time. Since we pass 8-bit data to the printers, one scanline of data must be saved after the first call to **PrintBuffer** and joined with the next set of data. The second time **PrintBuffer** is called it prints the leftover scanline along with six scanlines from the eight just passed. Two scanlines will be left over. By the time 56 scanlines have been passed, **PrintBuffer** will have enough left over to print two scanlines high rows. It will have six left over, print them with one from the newly passed eight and then print the seven left over.

The diagram below shows the first few step in the printing out of a page:



Application passes data in 640 byte buffer
Data in application print buffer is transferred to print driver buffer.

After printing, **PrintBuffer** returns and the application reads in new buffer data. The printer driver buffer holds the leftover scanline.

Data Shifted to top of printer buffer. Six lines of data from application buffer are shifted in. Two scanlines of data remain in application buffer.

Printing with a 7-Bit high Printer

The first panel shows the application has passed a full buffer to the printer driver; the printer driver then copies the data into its buffer for printing. In the second panel the printer driver has printed the top 7 scanlines of its buffer, sent a CRLF to the printer, and left one scanline unprinted. The application has also reloaded its buffer with 8 more scanlines of data. In the third panel, the leftover scanline in the printer driver's buffer has been shifted to the top and 6 scanlines of data have been shifted in from the application's buffer to fill up the lower part of the buffer. The **PrintBuffer** routine is now ready to start printing out the buffer.

It should be clear then that the printer driver needs its own 640-byte buffer to save scanlines between calls from the application so that it may combine the leftover lines with incoming lines.

The 8-bit printers avoid all this shifting around of data. They print the entire buffer of data at each call to **PrintBuffer**. Both types of drivers, however, must take some pains to "rotate" the data, which is to say assemble the horizontal bytes into vertical bytes for transmission over the serial bus. The first byte to be sent to the printhead is made up of the seventh bit from each of the first 8 (or 7 for a 7-bit printer) bytes in the first card. One bit at a time is shifted out from each of the bytes in the first card. Some printers put the bit from the first byte on top and others on the bottom.

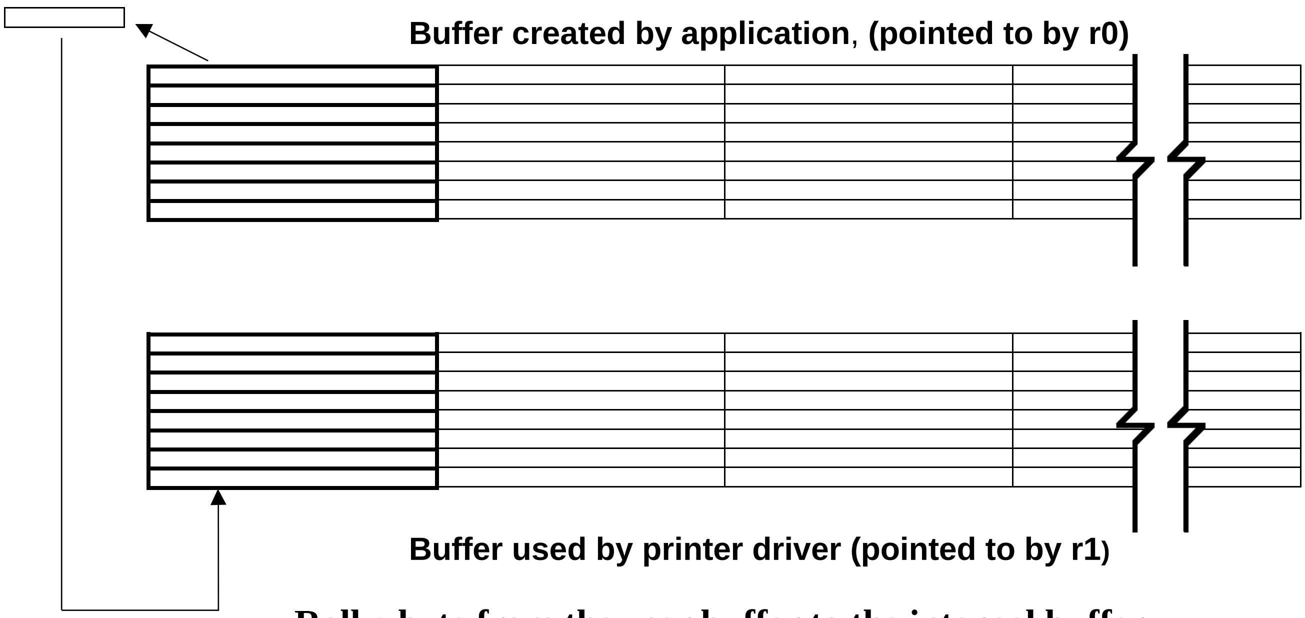
We now turn to a sample printer driver for an 8-bit printer, the Epson FX80. Later we will present the algorithm we use to deal with 7-bit printers such as the Commodore 801.

Sample printer driver for an 8-bit printer:

Sample is located in Appendix B: Examples: **8-Bit FX-80 Printer Driver**.

Sample Printer driver for 7-bit printers:

The Commodore driver is similar in overall structure to the Epson driver presented earlier. The fact that the Commodore printer is a 7-bit printer makes life a bit harder. The 8-byte high card-oriented buffer must itself be buffered into so that it may be printed 7 scanlines at a time. This is done in routines **TopRollBuffer** and **BotRollBuffer**. **TopRollBuffer** calls **RollaCard** to take a byte off the top of a card in the print buffer and shift each byte in the card up one as shown below.



Roll a byte from the user buffer to the internal buffer.

After the line is printed, there will be left over lines in the user buffer that will be printed the next time **PrintBuffer** is called. (Remember that with 7-bit printers, **PrintBuffer** can only print 7 of the 8 scanlines passed from the application in the buffer pointed to by **r0**. This leaves one scanline of data left over after the first call to **PrintBuffer**). **BotRollBuffer** rolls these leftover lines into the internal print buffer. For example, before the first line is printed, **TopRollBuffer** rolls the top 7 lines from the user print buffer to the internal printer driver buffer. These lines are printed and then **BotRollBuffer** is called to shift the remaining scanline from the user buffer to the internal buffer. **PrintBuffer** then returns to the application which is now free to reload the user buffer. **TopRollBuffer** and **BotRollBuffer** read a table to determine how many scanlines to roll each time they are called. The actual rolling of the scanlines is done a card at a time because the bytes in the user print buffer are organized that way. It was decided to have the application pass its output graphics data in card format since it is probable that most of the routines for drawing to the screen could then be reused to create the data for the printer.

Included below is an assembler listing of the driver for Commodore compatible printers.

Sample is located in Appendix B: Examples: **7-Bit MPS-801 Printer Driver**.

Note: GEOS was also designed to communicate with Postscript™-equipped printers which may print via Laser or Ink-jet technology. When using special fonts and software they will produce near typeset-quality output. However, creating drivers for Postscript™ is outside the scope of this document. If you want to get the best possible print output from GEOS, search the internet for the Laser Lovers' Disk and/or the geoPublish Tutorial.

Sprites

Hardware Sprites

The GEOS Kernal provides a simple interface to the hardware sprites supported by the C64. These routines control the sprites by writing to the VIC chip sprite registers as well as writing to the data space from which the VIC reads the sprite picture data. The reader should be familiar with the basic structure of sprite support on the C64 as explained in the Commodore 64 Programmer's Reference Guide.

One of the space/function tradeoffs made in GEOS was to support only basic sprite functions. Applications requiring elaborate sprite manipulation, such as games, will probably not be using many of GEOS's features, whereas business, or text-based applications will benefit from GEOS text, disk, and user interface features, and probably not need complicated sprite support.

The GEOS Kernal provides the following routines for drawing, erasing, and positioning:

Name	Description	Page
DisablSprite	Disable sprite.	20-194
DrawSprite	Define sprite image.	20-195
EnablSprite	Enable sprite.	20-196
PosSprite	Position sprite.	20-197

Plus, additional Sprite related mouse routines:

Name	Description	Page
MouseOff	Disable mouse pointer and GEOS mouse tracking.	20-170
MouseUp	Enable mouse pointer and GEOS mouse tracking.	20-171

Soft Sprites

The C64 contains a VIC chip to handle sprites in hardware. Unfortunately, the VIC is not available on the 128 while in 80-column mode. The functions of the VIC have been simulated in software that is included in the 128 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 128 Kernal provides the following additional routines for Soft Sprites:

Name	Description	Page
HideOnlyMouse	C128 Temporarily remove soft-sprite mouse pointer.	20-168
SetMsePic	C128 Set and preshift new soft-sprite mouse picture.	20-172
TempHideMouse	C128 Hide soft-sprites before direct screen access.	20-174

The major changes include: sprite 0 (the mouse pointer) 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 pointer's size in either x or y. You cannot change the color of the pointer. The size of the pointer image is limited to 16-pixels wide and 8 lines high. One added feature is the ability to add a white outline to the image that is used for the pointer. 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 image definition (previously unused) is now used to provide some size information about the sprite. This is used to optimize the drawing code.

Problem Areas to Watch Out for:

All sprite image data

All image data should be adjusted to include the 64th byte. This byte has size information that is required by the software sprite routines. The format of this byte is: high bit set means that the sprite is no more than 9 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 sprite image definition, or stuff it into the right place with some special code.

Writing directly to the screen

Since the 40-column sprites are handled with hardware, writing directly to the screen memory isn't a problem. If you do write directly to the VDC screen memory (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 **MainLoop**.

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 the 640 pixels screen width. You must use **PosSprite** to set the x-position. (**PosSprite** uses **NormalizeX** on the x-coordinate and then divides the x-coordinate by 2 before storing it into the VIC).

Reading values from the VIC chip

This is also ok for the status values and for the y-position. The x-position is in 40-column format. It will need to be multiplied by 2 to get the 80-column coordinate.

Using VIC chip collision detection

The chip continues to operate, so if you are using the **PosSprite** call (see above) collisions should be detected with some loss of accuracy (the low bit).

Writing to the VIC chip

(or calling **PosSprite**, **EnablSprite**, **DisablSprite**) at interrupt level:

Don't do it. Since the mouse and the sprites are drawn at **MainLoop**, this causes subtle, irreproducible timing bugs that are impossible to get out.

Known bugs in release 1 of GEOS 128 (1.3):

1) If location $1300 in application space is zero, then sprites in 80-column mode go haywire. All of our current applications that run in 80-column mode have put in a patch for this. Bug is in sprite code.

2) Doubling bitmaps through **BitmapClip** doesn't work.

3) **i_BitmapClip** needs call to **TempHideMouse** before being called.

Note: These three bugs were fixed in GEOS 128 V1.4.

RAM Expansions and GEOS 128

Introduction

Starting in version 1.3, GEOS is able to manage memory expansions in various ways (REU, RAM-Expansion Unit). This is one of the features that most differentiate version 1.2 from later versions. In the first part of this chapter we will examine the operations that GEOS performs in a "transparent" way to applications and the application possibilities of additional RAM in tasks parallel to those of the system.

In the second part of the chapter we will instead address the compatibility problem of an application with GEOS 128, and the various measures necessary to take advantage of the 80-columns offered by the C128.

Finally, in the last part of the chapter we will illustrate a whole series of small tricks useful to every programmer. Some are mostly gimmicks to get around the rare bugs present in the GEOS Kernal.

RAM expansions

The C64, by its nature, is unable to access an amount of memory higher than 64K. This limitation is due to the size of the address bus of the 6510 CPU, which, being formed by 8 distinct lines, can address at most 65536 bytes (64K). Faced with this physical limitation, any memory increase just seems impossible. Instead, the obstacle can be overcome. At the expansion port (Expansion Port) of the C64 are several lines, among which are the entire address bus, the data bus and a line that allows you to temporarily disable the CPU.

It is therefore possible that an external processor may temporarily take over the computer and perform operations directly in the memory of the C64. The REU's take advantage of this. They are in fact equipped with an internal processor capable of performing memory operations at very high speed, with large amounts of data. The CPU of the C64 cannot therefore directly access the banks of memory contained in the REU, but can communicate with the external processor, passing it some parameters and ordering it to perform some operations. In the moment the REU receives the command, it disables the 6510 and performs the required operations by interacting with the computer memory and the REU. The banks are all 64K and the size of the expansion determines the number of banks it contains.

To communicate with the REU the CPU must provide some parameters:

1. The REU BANK with which the operation takes place.
2. The address inside the bank.
3. The address inside the C64 where the operation is to begin.
4. The number of bytes needed.

These parameters must be stored in particular REU registers, located from EXP-BASE (DF00) onwards. With the addition of the REU the control registers of the external processor become accessible. When the parameters have been set, the CPU must store the operations in the command register assigned to the external processor. At this point, each time there is a command, the expansion processor executes it by temporarily disabling the 6510. The 6510 resumes control only when the operation is completed, and does not participate in anyway. The operation, therefore, takes place in a completely "transparent" and instant way as far as the C64 CPU can tell.

There are four main operations that can be carried out with the REU. Each requires a different command:

1. The VERIFY command allows you to compare data blocks of the same size, respectively contained in the memory of the C64 and that of the expansion.
2. The STASH command allows the transfer of a block of data from C64 memory to expansion.
3. The FETCH command, vice versa, transfers a block of data from the expansion to the C64 memory.

4. SWAP allows you to simultaneously exchange a block of data in memory with a block of the same size contained in the REU.

Name	Description	Page
VerifyRAM	RAM-Expansion Unit verify.	20-165
StashRAM	Transfer memory to RAM-Expansion Unit.	20-161
FetchRAM	Transfer data from RAM-Expansion Unit.	20-156
SwapRAM	Swap memory with an REU memory block.	20-163

Obviously, the amount of memory involved in each operation cannot exceed the size of the memory bank you are working with. The speed of the data transfer reaches 200K per second, and this makes it convenient to use RAM expansions even to just move large amounts of data from one area of the computer memory to each other. The last important feature for the management of expansions in the GEOS environment is about resetting the computer. Contrary to what one might think, the RAM expansions are not erased when resetting the computer and the information that is stored in them remains unaltered. The REU will only lose its contents by turning off the computer or deleting the contents voluntarily.

Now that we know more about how REUs work, we're able to illustrate how they are used by the GEOS Kernal and in which configurations you can get them. The GEOS Kernal V1.3 is able to "see" expansions up to 512K of memory. To be more precise, it can interact with any size REU up to 512K and organized in 64K banks. The possible quantities are therefore 64K, 128K, 192K, 256K, 320K, 384K, 448K, 512K. The actions that can be performed by the Kernal depend on the amount of external memory available. **Note**: GEOS 2.0 can use up to 2MB of an REU.

The user chooses the type of configuration that best suits his needs through the Configure application, which recognizes the type of expansion inserted and (depending on the amount of additional memory available) offers the user different possible system configurations.

There are two operations that the Kernal can always perform, even with the smallest expansion:

1. Move data areas very quickly from one point to another in the memory.
2. Save the Kernal in the REU for fast reboots that do not require disk access.

Applications that have to move large amounts of data, such as geoPaint when moving the working window to the drawing pad, often employ the **MoveData** routine of the GEOS Kernal. But **MoveData** is very slow when it has to perform large movements, since it must resort to a loop of instructions. If there is an expansion, however, you can delegate this task to the external processor: the Kernal does nothing but transfer the command to the REU, and immediately afterwards the REU transfers control back to the computer with the memory at the new address.

The total time required for the operation is much lower than that required by the traditional **MoveData** loop. When you choose this option, also called **MoveData**, Configure alters the system appropriately so that **MoveData** performs its functions using BANK 0 of the REU. The second thing the Kernal is able to do with an expansion consists in transferring the entire system and the reboot code into REU Bank 0, so that you can reboot without accessing the disk. With this option, when the user orders the Kernal to give control to Basic, the entire Kernal is transferred in the expansion together with a loader. To return to the GEOS environment, the user can press the "restore" button, or do a sys 49152, or finally run the Rboot file; the entire Kernal is then transferred from the expansion into memory in less than a second and control is immediately given back to GEOS. At this point the Kernal loads and runs deskTop.

The option just described, which Configure identifies as RAM Reboot, is particularly useful when you have to run many non-GEOS compatible files, returning to the GEOS environment each time in the shortest possible time. Upon returning, the previous configuration is kept, including the contents of any RAM disk, which we will discuss shortly. Note that if the Kernal is also simulating a RAM disk on the expansion, and a copy of deskTop resides in the RAM disk, when the system is reactivated by the expansion, deskTop is also loaded by the expansion.

MoveData and RAM Reboot can be selected simultaneously and they do not interfere with other possible uses of the REU. If the amount of available external memory exceeds 256K, GEOS is able to exploit it to achieve a Shadowed drive or RAM disk. Of course, these are alternative options to each other. The new "virtual" disk drive 1541 that is created can be either drive A or drive B. (With GEOS 2.0, 1571 and 1581 RAM disks can also be created).

The Shadowed drive is a real 1541 disk drive backed by a RAM Drive the same capacity as the formatted disk. Each time the user loads an application or a data file into memory, the file is transferred to the Shadowed drives RAM drive so that the Kernal can load from it (and not from disk) in a very short time. Each time an application saves a file to disk, the file is also copied to the RAM drive. In this way the loading of all the files read or saved at least one time can happen directly from the REU.

As an alternative to the Shadowed disk, the user can configure the GEOS Kernal to use the RAM expansion as a virtual 1541 disk, i.e. as a standalone RAM disk. The virtual disk is identified as drive B since the real drive is drive A. For applications and for the user, it is as if a second disk is a connected 1541 drive. The difference is that the files saved on the virtual disk are loaded very quickly (in little more than the time to double click the mouse button on the icon), and RAM disk data copying is virtually instant. However, we must remember that the contents of the RAM disk are completely lost if the computer is turned off. Since the two options cannot coexist, the user must decide which one will be most useful to him when making his choices via Configure.

The Configure application is of the AUTO-EXEC type, and therefore during system boot is always executed before deskTop. When it executes, Configure checks the contents of **firstBoot**, and if it is $00 it detects that deskTop has not been loaded yet and therefore the installation should progress. (Configure was not called by the user, but by the system). Configure will automatically configure the system according to the specifications that were saved by the user the previous time, or sets the default ones. However, when Configure is called by the user, it finds the contents of **firstBoot** is different from $00 and therefore decides the user should receive control for setting up a new system configuration, which will be saved on disk. From now on, when CONFIGURE automatically runs at boot time it will use the data saved on disk to configure the system as established by the user.

All the operational possibilities just described, offered by the GEOS V1.3+ Kernal, are completely transparent to applications. The applications are not required to know if drive A is Shadowed, or if drive B is virtual, since the system masks any differences, and not even if the **MoveData** routine uses the expansion processor or not. Applications continue to use the routines of the Kernal as they always have, that is, by checking exclusively if there are two disk drives or just one.

Apps and Expansions

Even though GEOS is able to efficiently and independently manage any RAM expansion, it may happen that an application wishes to use the REU to perform different tasks. For example, store fonts without the expansion necessarily being used as a RAM disk. For this purpose, GEOS makes five system routines available to applications specifically to give commands to the memory expansion. The applications can access the **ramExpSize** variable to determine the number of 64K banks of which the currently inserted expansion is composed. The addresses within each bank are relative to the beginning of the bank itself, and therefore are independent from your order number. Finally, remember that these routines are only available in GEOS version 1.3 and later, and in GEOS 128.

Applications and compatibility with GEOS 128

- Most C64 GEOS software will run under the C128 GEOS in 40-column mode.
- All data files, scraps, fonts, & printer drivers are identical under C64 and C128 GEOS.
- Input drivers are located at different addresses in the two machines, and hence are incompatible. We have added a new file type, INPUT_128, for C128 input drivers.
- As the deskTop is heavily tied into each OS, we've decided to give the 128 its own desktop filename, "128 DESKTOP", so as to avoid confusion with the 64's "DESK TOP" file. (The deskTop is of file type "SYSTEM", and can't be renamed by the user).
- Use the **c128Flag** to determine what OS you are running under. See **Example**: **Check128**.

GEOS 128 can be considered a very close relative of GEOS for the C64. All the routines in GEOS for the C64 jump table are faithfully reported in GEOS 128, and the parameters are all the same. All system variables that are available in GEOS V1.3+ are the same in GEOS 128 1.3+. For these reasons, applications produced by Berkeley Softworks for GEOS 64 can be run in the GEOS 128 environment. The "almost" is necessary because there is always some difference.

Applications that need to access the computer's original Kernal are not compatible with GEOS 128, due to the substantial differences between the Kernal of the C128 and that of the C64. These applications include, for example, the desk accessory calculator and geoCalc, which perform complex mathematical operations by accessing the computer's ROM math routines. If you want the application to be compatible with both systems while accessing ROM routines, it is necessary to create two distinct jump tables into the ROM, one for each Kernal.

GEOS 128, in addition to faithfully reproducing the characteristics of GEOS V1.3+ for the C64, has several new features including 80-column graphics. Let's see what steps are required for applications that were created explicitly for GEOS 64 to use the 80-column screen of the C128.

128 Flags for Applications & Desk Accessories

In order for the 128 DESKTOP & other applications to know what files run in what mode, we've adopted a standard that should be used on ALL application, desk accessories, & auto-execution applications. This flag is located in the file header block of each of these programs. Since permanent filenames are only 16 bytes long, we have 4 leftover bytes that have been unused till now, that we've constantly been setting to all 0's. The last of the bytes (see O_128_FLAGS) now has meaning to the 128 OS & DeskTop.

Bit 7	Bit 6	Description
0	0	The application runs in 40-column mode only.
0	1	The application runs in both 40-column and 80-column modes.
1	0	The application cannot run under GEOS 128.
1	1	The application runs only in 80-column mode.

Note: bits 5 through 0 are unused and should always be 0.

80-column graphics with GEOS 128

If you want the application to be able to enable and manage the 80-column mode offered by GEOS 128, you have to follow some fundamental guidelines.

1. GEOS 128 must be able to determine if the application is compatible with 80-column mode. GEOS 128 needs this information because if 80-column mode is enabled, and the application cannot use the 80-column screen, you must notify the user or automatically return to 40-column mode. The application must

set the value CF_40_80 ($40) in the O_128_FLAGS location of its File Header. This will allow GEOS 128 to use both graphic modes (40 and 80-columns) with the application.

> **Note**: 128 GEOS routines **LdApplic** and **LdDeskAcc** will return the error INCOMPATIBLE if these flags in the file header block do not allow running in the currently active **graphMode**.

2. In 80-column mode it is necessary to enlarge all menus so that they are able to contain the BSW 128 system font, which is wider than the C64 system font. The custom of Berkeley is to set the right limit value in the menu structures based on the value contained in **graphMode** ($80 for 80-columns, $00 for 40-columns). The **graphMode** variable is only present in GEOS 128.

3. Most changes in graphical values needed for compatibility with the 80-column mode can be accomplished by setting bit 15 of all the x-coordinates and all widths that are passed to the system by OR'ing the value with DOUBLE_W. In 40-columns mode the high bit is ignored, while in the 80-column mode it serves double all horizontal dimensions. By doing so, the image always has the same size on the screen. For example, if an x-coordinate = 50 pixels in 40-column mode, it must be passed to GEOS 128 in the form $0032 | DOUBLE_W ($8032), so that in 80-column mode it becomes $0064 (100 pixels).

4. In the application's GEOS menu (or in any case within any menu) the entry "switch 40/80" must be available. The procedure associated with the event must simply perform the logical EOR operation between **graphMode** and the constant $80 (inverts the value of bit 7), store the result in **graphMode** and call the **SetNewMode** routine ($C2DD GEOS 128 only). Later the application must redraw the current screen in the new graphics mode. If the horizontal dimensions already have bit 15 set to 1, the routine that redraws the screen works without any changes. Here is an example of the codes associated with the item "switch 40/80":

   ```
   SwitchDsp:
         lda    graphMode
         eor    #$80
         sta    graphMode
         jsr    SetNewMode                ; (SetNewMode routine is only available in GEOS 128)
         ;--- code to initialize the screen again
   ```

   This same block can be made easier to read and maintain by using the **tmbf** macro.

   ```
   SwitchDsp:
         tmbf   7,graphMode               ; Toggle the MSD bit of graphMode
         jsr    SetNewMode                ; (SetNewMode routine is only available in GEOS 128)
         ;--- code to initialize the screen again
   ```

5. The trick adopted to adapt the horizontal dimensions to 80-columns (bit 15 set with DOUBLE_W) is not always effective. When the value of a horizontal coordinate is doubled in 80-column mode, the 0 bit of the resulting word is always cleared. In some cases, this can be a serious limitation: for example, when you want to fill the screen with a pattern that extends to the right-edge of the screen.

   To solve this problem the ability to add 1 to the x-coordinate was introduced in the graphic routines of GEOS 128: bit 15 of the word continues to have the same meaning (if set to 1 the value is doubled in 80-columns), while bit 13 gives new information, but only in the 80-column mode. Bit 13 becomes bit 0 of the resulting word from the "doubling" operation. If for example you want to locate the side right of the screen, the horizontal coordinate must be $A000 + 319.

   Example:

   ```
         LoadW x-coordinate,#319 | DOUBLE_W | ADD1_W
   ```

Thanks to these five tricks you should be able to easily exploit the 80-column mode of the C128. However, some tweaks may be needed in the testing phase of the application layout (this type of verification is always advisable).

The little tricks of the trade

In this last section we report a series of small tricks of which the programmer should take into account in the implementation of applications. In some, the case is to get around the small bugs still present in the GEOS structure.

1. If the application can run in the GEOS 128 environment, and is capable of managing the second drive, pay particular attention to all calls from the **PutDirHead** routine, and each time insert immediately before it 'jsr **EnterTurbo**'. This is necessary because in the first production of GEOS 128 V1.3 there is a bug in the 1571 disk driver: the call to **EnterTurbo** is missing. The result is that, in certain circumstances, calling **PutDirHead** can also ruin the disk. This trick does not create incompatibility with GEOS 64. The bug is present in GEOS 128 Configure V1.4. It was fixed no later than V2.0 of GEOS 128 with Configure V2.0 9/8/88.

2. If the program can run desk accessories, Blackjack programs with a Date < 10/9/86 alter the content of the word for $4C95 (builds on or after 10/9/86 do not have this issue). This address is not in the area temporarily saved on disk. To remedy this bug, the code responsible for the desk accessories must be preceded by and followed by the instructions **PushW** $4C95 and **PopW** $4C95. Furthermore GEOS 64 V1.0-V1.3 does not save **moby2** while running desk accessories. This means sprites can easily be enlarged in height by the DAs and then modified. Here is a practical example of how to act, both on the application and on the desk accessory:

   Note: These work arounds are not necessary as of GEOS 1.4 and above.
   Note: geoWrite 2.1 still uses these workarounds to protect against older desk accessories and to allow it to run on older versions of GEOS without issue.

```
Applications:
       ldx    CPU_DATA               ; save the state of moby2 on the stack
       LoadB  CPU_DATA,#IO_IN
       PushB  moby2
       stx    CPU_DATA
       PushW  $4C95                  ; save the data that Blackjack would destroy
LOAD AND RUN THE DESK ACCESSORY HERE
       PopW   $4C95                  ; restore the word
       ldx    CPU_DATA
       LoadB  CPU_DATA,#IO_IN
       PopB   moby2                  ; restore moby2
       stx    CPU_DATA

DeskAccessory:                       ; init code
       ldx    CPU_DATA               ; save the state of moby2
       LoadB  CPU_DATA,#IO_IN
       MoveB  moby2,savedmoby2
       LoadB  moby2,#$XX             ; set moby2 as needed
       stx    CPU_DATA

ExitCode:
       ldx    CPU_DATA               ; restore the state of moby2
       LoadB  CPU_DATA,#IO_IN
       MoveB  savedmoby2,moby2
       stx    CPU_DATA
```

Note: Desk Accessories included with GEOS 1.4+ do not have the above code in them.

3. GEOS may not work properly if no icon has been defined. If the application does not use icons, it is better to define a dummy one to avoid problems. You can define it to be one scan line high, one byte wide and with the pointer to the graphic data cleared. (This is true in all versions of GEOS).

4. In GEOS 1.4+, it must never be assumed that the concatenation of directory blocks begins with sector $12 / $01, or that the Directory Header Block is located at T/S $12 / $00, as the format is different for 1581 disks. They must always execute the **GetDirHead**, **PutDirHead**, **Get1stDirEntry** and **GetNxtDirEntry** routines present in the current disk driver.

5. The current device must never be directly changed in **curDrive** or **curDevice**. Instead, you need to call **SetDevice** to address the disk drive desired.

6. In desk accessories: It is possible a desk accessory might detect that it cannot run while it is initializing. e.g. desk accessory requires GEOS 2.0 to run but the current OS version is 1.3. The initialization code cannot jump directly to **RstrAppl**, instead use **LoadW appMain**,**#RstrAppl** and then rts back to the **MainLoop**. At the end of the next **MainLoop** the desk accessory will be terminated and the calling application will be restored.

7. In the dialog boxes: the **DB_USR_ROUT** command is executed before icons have been drawn. If the custom routine needs to draw something over the icons, you must load **appMain** with the address of another routine, and delegate it to display the drawings over the icons.

8. Never use the **MoveData** routine to move the contents of registers **r0 - r15**.

9. The dialog boxes can manage no more than eight icons at the same time. If the box must display more than eight icons, it must manage them autonomously through the vector **otherPressVec**.

10. Remember that the handling of events (processes, routines pointed to by **keyVector** and **appMain**) is active while a menu is open. The routine pointed to by **otherPressVec** is partially active: it analyzes only the button releases. If the application wants to ignore these events when opening a menu (very frequent situation) don't forget to disable them.

11. Calls to **DoMenu** and **DoIcons** move the mouse. Since generally this is not desirable, one must act as follows:

```
PushW  mouseXPos
PushB  mouseYPos
jsr    DoIcons            ; or DoMenu
PopB   mouseYPos
PopW   mouseXPos
```

12. If the application interacts with **RecoverVector** (to restore the covered background from a menu or dialog box) remember that the routine identified by the vector is called twice when restoring the background underneath a dialog box that has a shadow. If the shadow pattern is 0 the recovery routine is only called once.

13. GEOS 1.1 interrupt main does not clear the decimal mode bit in the Processor Status Register (PSR). Since the counts are done with this bit cleared, the interrupt must never occur while decimal mode is activated. ie: You must disable interrupts before performing decimal mode operations and reenable interrupts after decimal mode is off. This problem was fixed in GEOS V1.2.

14. If the application turns off (blanks) the screen or writes to **grcntrl1** ($D011), make sure bit 7 is always at 0. Since accidentally, in the course of several operations, this bit can become 1, the following code can be used to reset it:

```
lda    grcntrl1            ; get the current value
and    #%01111111          ; reset bit 7
sta    grcntrl1            ; store the new value

;---(Macro version).
rmbf   7,grcntrl1          ; get current value of grcntrl1,
                           ; reset bit 7 and store new value
```

15. When an application activates a menu with **DoMenu**, GEOS sets **mouseFaultVec** to point to an internal handler that controls the closing of the current menu when the mouse goes beyond the edges of the menu. This will conflict with the application if it also needs to use **mouseFaultVec** while having an active menu structure. The solution to the problem is obtained with two interventions, one in the application initialization routine and one in the service routine that the application assigns to the **mouseFaultVec** vector.

First intervention. When the application wants to use **mouseFaultVec** and simultaneously a menu structure, the initialization routine must, after the call in **DoMenu**, store the contents of the **mouseFaultVec** vector in an internal vector. Once the pointer to the system handler has been saved, the application can set **mouseFaultVec** with the address of the applications service handler.

Example:

```
Init:
    LoadW  r0,#ourMenu
    jsr    DoMenu
    MoveW  mouseFaultVec,saveMFV
    LoadW  mouseFaultVec,#MFVHandler
    ...
```

Second intervention. When the service routine associated with **mouseFaultVec** receives control, it must check whether its execution was requested by the system as a result of the mouse overstepping one of the limits set by the application. If it was, it can perform its functions and return control to **MainLoop** with an rts instruction. Otherwise, it must hand over control to the routine whose address was stored in the internal vector by the initialization routine.

```
MFVHandler:
    lda    menuNumber          ; check if a menu is active
    beq    10$                 ; if the menuNumber is 0 then menu is closed
    ldx    saveMFV+1           ; menu is active. let the system routine handle this
    lda    saveMFV
    jmp    CallRoutine         ; transfer control and let Kernal return to the Main Loop
    ;---   Change jmp to a jsr if you still need to process
    ;      after menu handling is done.
10$ ...                        ; application mouse fault handler logic starts here
```

WarmStart Configuration

Whenever **FirstInit** is called, such as when GEOS boots, the Commodore hardware is setup. This includes setting up the VIC chip RAM bank, and the CIA chips. The following table summarizes the state of the machine.

Initial Boot Configuration

Address	Value	Size	Comment
C64&128			clear decimal mode with cld
CPU_DDR	$2F	1	init. 6510 data direction reg.
CPU_DATA	$30	1	Set to ALL RAM
OS_VARS	0	$A00	Clear GEOS RAM area, Global & local, to all 0's
CPU_DATA	$36		set memory map to have Kernal & I/O in
128 Only			
scr80polar	$40	1	VDC BG/FG Polarity
scr80colors	$E0	1	(VDC_GRY1<<4) | VDC_BLACK
VDC	defaults		VDC set to 640X200 Monochrome.
vdcClrMode	0	1	
C64&128			
CIA registers			
cia1ddrb	0	1	clear cia1 DDRB Initialize key scan values
cia1crb	0	1	clear **cia1crb** to no keys pressed
cia2crb	0	1	clear **cia2crb**
cia1cra	$80/00	1	set 50/60hz bit PAL/NTSC.
cia2cra	$80/00	1	set 50/60hz bit PAL/NTSC.
cia2pra	(**cia2pra**| #$30|#$04| GRBANK2)	1	Keep old serial bus data. (so we don't screw up fast serial bus) set graphics chip bank select (CIA port A).
cia2ddra	$3F	1	Set DDRA direction.
cia1icr	$7F	1	clear interrupt sources.
cia2icr	$7F	1	
	Init the cia1 time of day clock		
cia1crb	(**cia1crb** & $7F)	1	set to TOD clock reads and writes.
cia1todhr	%10000000|$0C		Noon.
cia1todmin	0		minutes.
cia1todsec	0		seconds.
cia1tod10ths	0		and 1/10 seconds.
VIC registers			
mob0xpos	0	16	initial x, y-position of sprites 0-7.
msbxpos	0	1	most significant bits of all sprites x-position.
mob0clr	BLUE	1	Mouse color.
mob1clr	BLUE	1	String cursor color.
mobprior	$00	1	(object/background priority)0=obj.
mobmcm	$00	1	(object multicolor) 1 = mem.
mobx2	%00000000	1	Disable Sprite x-double-width.
moby2	%00000000	1	Disable Sprite y-double-height.
mobenble	%00000001	1	(object enable) only the mouse.
grcntrl1	ST_DEN|ST_25ROW|ST_BMM|3		(**Note**: need y scroll = 3). (byte)
rasreg	251	1	raster reg. (set for interrupt at bottom)

grcntrl2	ST_40COL	1	Set Graphics Mode.
grmemptr	(((]COLOR__MATRIX)*2)&$F0) | ((]SCREEN_BASE*2)&$0E)		VIC Memory setup (byte)
grirq	%00001111	1	Acknowledge all VIC interrupts.
grirqen	%00000001	1	Enable Raster Interrupt.

Mouse and window variables

pressFlag	0	1	no presses to handle.
dispBufferOn	ST_WR_FORE | ST_WR_BACK	1	Write to both screen and back buffer.
mouseXPos	0	2	
mouseYPos	0	1	
mouseOn	%11100000	1	Enable Mouse/Menus/Icons.
mousePicData	arrow pic	64	copy arrow picture to mouse.
msePicPtr	**mousePicData**	2	
mouseLeft	0	1	Mouse constraints.
mouseTop	0	1	
mouseRight	319	2	639 in C128 80 Col Mode.
mouseBottom	199	1	
maxMouseSpeed	MAXIMUM_VELOCITY	1	Mouse Speed.
minMouseSpeed	MINIMUM_VELOCITY	1	
mouseAccel	MOUSE_ACCELERATION	1	
currentMode	PLAINTEXT	1	Text Mode.
windowTop	0	1	Text constraints.
windowBottom	199	2	
leftMargin	0	2	
rightMargin	319	2	639 in C128 80 Col Mode.
inputData	-1	1	(diskData current joystick direction)
COLOR_MATRIX	DKGREY<<4 | LTGREY	1000	dark grey on light grey screen.
extclr	BLACK	1	Border color
interleave	8	1	Disk interleave.
curDrive	8	1	Initialize disk drive with SetDevice.
curDevice	8	1	reinitialized.
numDrives	8	1	Change # of drives to 1

Time and Date

minutes	0		The following sets up initial Year/Month/Day/Hour, for now, so that the disk date stamps look reasonable.
seconds	0		
year	86		
month	9		
day	20		09/20/1986
hour	12		Noon
alarmSetFlag	0	1	
o_alarmCount	0	1	Internal Counter of active alarms.

Vectors

appMain	NULL	2	
intTopVector	o_InterruptMain	2	Set Vector to Kernal internal handler
intBotVector	NULL	2	
keyVector	NULL	2	
inputVector	NULL	2	
otherPressVec	NULL	2	
RecoverVector	o_RecoverRectangle	2	Kernal handler for recovering background
mouseVector	NULL	2	
mouseFaultVec	NULL	2	
StringFaultVec	NULL	2	
alarmTmtVector	NULL	2	
BRKVector	o_Panic	2	Kernal internal handler for BRK
EnterDeskTop	o_EnterDeskTop	2	Set Vector to Kernal internal handler

selectionFlash	SELECTION_DELAY	1	
alphaFlag	0	1	
iconSelFlag	ST_FLASH	1	set default to flash
faultData	0	1	

Kernal Private Variables

o_nbrProcesses	0	1	reset Kernal Private variables to 0
o_numberAsleep	0	1	
o_curIconIndex	0	1	

Sprite pointers

Initialize Sprite pointers to sprite picture data[†]

spr0pic	[(spr0pic>>6)	1
spr1pic	[(spr1pic>>6)	1
spr2pic	[(spr2pic>>6)	1
spr3pic	[(spr3pic>>6)	1
spr4pic	[(spr4pic>>6)	1
spr5pic	[(spr5pic>>6)	1
spr6pic	[(spr6pic>>6)	1
spr7pic	[(spr7pic>>6)	1

Final Steps

Forcefully exit any running turbo code

Restore ROM Vectors

```
MoveShortBlock $FD30, $0314,32         Restore the C64 vectors in page 3 from ROM

Grey the Screen:                       place a grey pattern all over the screen
                                       A000 to BF3F
```

Note:[†] For more info on how Sprite pointers work see the Commodore 64 Programmer's Reference Manual. The video space is 16K thus needing only 14 bits to address the entire space. The sprite pictures use 63 bytes and must be on 64 byte boundaries, thus the start of each sprite picture has an address with the low 6 bits 0. Thus 14 - 6 = 8, only one byte is needed to specify the start address of a picture anywhere in the 16K graphics memory space.

Dialog Box and Auto Exec Configuration

When a Dialog Box or Auto Execute application is loaded the current system state is saved. (See Chapter 19: "**Environment > Structures > dlgBoxRamBuf"** for more information on what is saved). The following table shows the default values applied before passing control to the Dialog Box or Auto Exec.

Mouse and window variables

currentMode	PLAINTEXT	1	Plain Text Mode.
dispBufferOn	ST_WR_FORE | ST_WR_BACK	1	Write to both screen and back buffer.
mouseOn	%11100000	1	Enable Mouse/Menus/Icons.
mousePicData	arrow pic	64	copy arrow picture to mouse.
windowTop	0	1	Text constraints.
windowBottom	199	2	
leftMargin	0	2	
rightMargin	319	2	639 in C128 80 Col Mode.
pressFlag	0	1	no presses to handle.

Vectors

appMain	NULL	2	
intTopVector	o_InterruptMain	2	Set Vector to Kernal internal handler
intBotVector	NULL	2	
mouseVector	NULL	2	
keyVector	NULL	2	
inputVector	NULL	2	
mouseFaultVec	NULL	2	
otherPressVec	NULL	2	
StringFaultVec	NULL	2	
alarmTmtVector	NULL		
BRKVector	o_Panic	2	Kernal internal handler for BRK
RecoverVector	o_RecoverRectangle	2	Kernal handler for recovering background

selectionFlash	SELECTION_DELAY	1	
alphaFlag	0	1	
iconSelFlag	ST_FLASH	1	set default to flash
faultData	0	1	

Kernal Private Variables

o_nbrProcesses	0	1	No Active Processes
o_numberAsleep	0	1	No Sleepers
o_curIconIndex	0	1	No Icons

Sprite pointers

Initialize Sprite pointers to sprite picture data[†]

spr0pic	[(spr0pic>>6)	1
spr1pic	[(spr1pic>>6)	1
spr2pic	[(spr2pic>>6)	1
spr3pic	[(spr3pic>>6)	1
spr4pic	[(spr4pic>>6)	1
spr5pic	[(spr5pic>>6)	1
spr6pic	[(spr6pic>>6)	1
spr7pic	[(spr7pic>>6)	1

Chapter 15

Reserved for Future Use

Chapter 16

Reserved for Future Use

Chapter 17

Reserved for Future Use

Chapter 18

Reserved for Future Use

Environment

constants

Miscellaneous:

These constants should always appear first in your constants files.

```
TRUE                = -1
FALSE               = 0
```

C128

```
ADD1_W              = $2000
DOUBLE_W            = $8000
DOUBLE_B            = $80
GR_40               = 0                   ; graphMode 40-column active
GR_80               = %10000000           ; 80-column active
ARROW               = $00                 ; arrow pointer
```

Fonts

```
FONTLEN             = $9                  ; size of fontTable
```

Flags

```
CLEAR               = 0
SET                 = 1
```

pressFlag

```
KEYPRESS_BIT        = 7                   ; other keypress
INPUT_BIT           = 6                   ; input device change
MOUSE_BIT           = 5                   ; mouse press

SET_KEYPRESS        = %10000000           ; other keypress
SET_INPUTCHG        = %01000000           ; input device change
SET_MOUSE           = %00100000           ; mouse press
```

faultFlag

```
OFFTOP_BIT          = 7                   ; mouse fault up
OFFBOTTOM_BIT       = 6                   ; mouse fault down
OFFLEFT_BIT         = 5                   ; mouse fault left
OFFRIGHT_BIT        = 4                   ; mouse fault right
OFFMENU_BIT         = 3                   ; menu fault

SET_OFFTOP          = %10000000           ; mouse fault up
SET_OFFBOTTOM       = %01000000           ; mouse fault down
SET_OFFLEFT         = %00100000           ; mouse fault left
SET_OFFRIGHT        = %00010000           ; mouse fault right
SET_OFFMENU         = %00001000           ; menu fault

ANY_FAULT           = %11111000
```

Desk Accessory save foreground bit (Obsolete)

```
FG_SAVE             = %10000000           ; save and restore foreground graphics data
CLR_SAVE            = %01000000           ; save and restore color information
```

Dialog Box:

```
DEF_DB_POS          = $80               ; command for default dialog box position
SET_DB_POS          = 0                 ; command for user-set DB position
```

Descriptor table commands

```
OK                  = 1                 ; put up system icon for "OK", command is
                                        ; followed by 2 byte position indicator, x-position
                                        ; in bytes, y-position in pixels. Note: positions
                                        ; are offsets from the top left corner of the
                                        ; dialog box.
CANCEL              = 2                 ; like OK, system DB icon, position follows
YES                 = 3                 ; like OK, system DB icon, position follows
NO                  = 4                 ; like OK, system DB icon, position follows
OPEN                = 5                 ; like OK, system DB icon, position follows
DISK                = 6                 ; like OK, system DB icon, position follows
;FUTURE1            = 7                 ; reserved for future system icons
;FUTURE2            = 8                 ; reserved for future system icons
;FUTURE3            = 9                 ; reserved for future system icons
;FUTURE4            = 10                ; reserved for future system icons
DBTXTSTR            = 11                ; command to display a text string.
DBVARSTR            = 12                ; used to put out variant strings.
DBGETSTRING         = 13                ; get an ASCII string from the user.
DBSYSOPV            = 14                ; any press not over an icon return to application.
DBGRPHSTR           = 15                ; execute graphics string.
DBGETFILES          = 16                ; get filename from user.
DBOPVEC             = 17                ; user defined other press vector.
DBUSRICON           = 18                ; user defined icon.
DB_USR_ROUT         = 19                ; user defined routine.
```

Offsets into descriptor table

```
OFF_DB_FORM         = 0                 ; box form description, i.e. shadow or not
OFF_DB_TOP          = 1                 ; position for top of dialog box
OFF_DB_BOT          = 2                 ; position for bottom of dialog box
OFF_DB_LEFT         = 3                 ; position for left of dialog box
OFF_DB_RIGHT        = 5                 ; position for right of dialog box
OFF_DB_1STCMD       = 7                 ; 1st command in dialog box
                                        ; descriptor table
```

System Dialog Icon dimensions

```
SYSDBI_WIDTH        = 6                 ; width in bytes
SYSDBI_HEIGHT       = 16                ; height in pixels
MAX_DB_ICONS        = 8                 ; maximum number of dialog icons
                                        ; this includes system icons + user icons
```

These equates define a standard, default, dialog box position and size as well as some standard positions within the box for outputting text and icons.

Default Coordinates

```
DEF_DB_TOP          = 32                ;$20  top y-coordinate of default box
DEF_DB_BOT          = 127               ;$7F  bottom y-coordinate of default box
DEF_DB_LEFT         = 64                ;$40  left-edge of default box
DEF_DB_RIGHT        = 255               ;$FF  right-edge of default box
```

Standard Text Locations

```
TXT_LN_X            = 16                ;$10  standard text x-start
TXT_LN_1_Y          = 16                ;$10  standard text line y-offsets
TXT_LN_2_Y          = 32                ;$20
TXT_LN_3_Y          = 48                ;$30
TXT_LN_4_Y          = 64                ;$40
TXT_LN_5_Y          = 80                ;$50
```

Standard Icon Locations

```
DBI_X_0             = 1                 ;$01  left-side standard icon x-position
DBI_X_1             = 9                 ;$09  center standard icon x-position
DBI_X_2             = 17                ;$11  right-side standard icon x-position
DBI_Y_0             = 8                 ;$08  top standard icon y-position
DBI_Y_1             = 40                ;$28  middle standard icon y-position
DBI_Y_2             = 72                ;$48  bottom standard icon y-position
```

Icon Y Locations for dialogs with 4 Icons on right-side

```
DBGF_Y_0            = 25                ;$19  OPEN
DBGF_Y_1            = 42                ;$2A  DISK
DBGF_Y_2            = 59                ;$3B  DRIVE
DBGF_Y_3            = 76                ;$4C  CANCEL
```

Disk:

```
BLOCKSIZE           = 256               ; total bytes in block
BLKDATSIZE          = 254               ; total data bytes in a block

;---   Equates for variable "driveType". High two bits of driveType have special meaning
;      (only 1 may be set):
;      Bit 7: if 1, then RAM DISK
;      Bit 6: if 1, then Shadowed disk
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
DRV_NETWORK         = 15                ; drive type for GEOS geoNet "drive"

DRIVE_A             = 8
DRIVE_B             = 9
DRIVE_C             = 10
DRIVE_D             = 11
```

Directory

DirHeader: **curDirHead** $8200

```
DK_NM_ID_LEN        = 18                ; # of characters in disk name

;--- Offsets into a directory header structure
;                   = 3                 ; 1571 double sided flag. $80=double sided format.
OFF_TO_BAM          = 4                 ; first BAM entry
OFF_DISK_NAME       = 144               ; disk name string
OFF_DSK_ID          = 162               ; disk ID
OFF_OP_TR_SC        = 171               ; track and sector for off page directory
                                        ; entries. 8 files may be moved off page
OFF_GS_ID           = 173               ; where GEOS ID string is located
OFF_GS_DTYPE        = 189               ; GEOS disk type.
                                        ;      0 for normal disk
                                        ;      'B' for BOOT disk
                                        ;      'P' for Master disk
                                        ; zeroed on destination disk during disk copy
```

DirBlock

```
FRST_FILE_ENTRY     = 2                 ; first dir entry is at byte #2
```

DirEntry: **dirEntryBuf** $8400

```
ENTRY_SIZE          = 16                ; size of filename
DIRENTRY_SIZE       = 30
ST_WR_PR            = $40               ; write protect bit: bit 6 of byte 0 in the
                                        ; directory entry
```

DirEntry Offsets

```
OFF_CFILE_TYPE      = 0                 ; standard Commodore file type indicator
OFF_INDEX_PTR       = 1                 ; index table pointer (VLIR file)
OFF_DE_TR_SC        = 1                 ; track for file's 1st data block
OFF_FNAME           = 3                 ; file name
OFF_GHDR_PTR        = 19                ; track/sector info on where header block is
OFF_GSTRUC_TYPE     = 21                ; GEOS file structure type
OFF_GFILE_TYPE      = 22                ; GEOS file type indicator
OFF_YEAR            = 23                ; year (1st byte of date stamp)
OFF_SIZE            = 28                ; size of the file in blocks
OFF_NXT_FILE        = 32                ; next file entry in directory structure
```

low-level GEOS disk handling routines

```
N_TRACKS           = 35                 ; # of tracks available on the 1541 disk
DIR_TRACK          = 18                 ; track # reserved on disk for directory
DIR_1581_TRACK     = 40                 ; 1581 track # reserved on disk for directory
TOTAL_BLOCKS       = 664                ; number of blocks on 1541 disk, not including directory
                                        ; track
```

Disk access commands

```
MAX_CMND_STR       = 32                 ; maximum length a command string would have
DIR_ACC_CHAN       = 13                 ; default direct access channel
REL_FILE_NUM       = 9                  ; logical file number & channel used for relative files
CMND_FILE_NUM      = 15                 ; logical file number & channel used for command files

;--- Indexes to a command buffer for setting the track and sector number for a direct access
;    command.
TRACK              = 9                  ; offset to low-byte decimal ASCII track number
SECTOR             = 12                 ; offset to low-byte decimal ASCII sector number
```

Disk Errors:

```
NO_ERROR           = $00                ; no error
NO_BLOCKS          = $01                ; not enough blocks
INV_TRACK          = $02                ; invalid track
INSUFF_SPACE       = $03                ; not enough blocks on disk
FULL_DIRECTORY     = $04                ; directory full
FILE_NOT_FOUND     = $05                ; file not found
BAD_BAM            = $06                ; bad allocation map
UNOPENED_VLIR      = $07                ; unopened VLIR file (this is a non fatal error)
INV_RECORD         = $08                ; invalid VLIR record
OUT_OF_RECORDS     = $09                ; cannot insert/append more records
STRUCT_MISMATCH    = $0A                ; file structure mismatch
BFR_OVERFLOW       = $0B                ; buffer overflow during load
CANCEL_ERR         = $0C                ; deliberate cancel error
DEV_NOT_FOUND      = $0D                ; device not found
INCOMPATIBLE       = $0E                ; this error is returned when an attempt is made
                                        ; to load a program that can't be run on the
                                        ; current graphics modes under GEOS 128
HDR_NOT_THERE      = $20                ; cannot find file header block
NO_SYNC            = $21                ; can't find sync mark on disk
DBLK_NOT_THERE     = $22                ; data block not present
DAT_CHKSUM_ERR     = $23                ; data block checksum error
WR_VER_ERR         = $25                ; write verify error
WR_PR_ON           = $26                ; disk is write protected
HDR_CHKSUM_ERR     = $27                ; checksum error in header block
DSK_ID_MISMAT      = $29                ; disk ID mismatch
BYTE_DEC_ERR       = $2E                ; can't decode flux transitions off of disk
DOS_MISMATCH       = $73                ; wrong DOS indicator on the disk
```

File Types:

```
;--- This is the value in the "GEOS file type" byte of a directory entry that is pre-GEOS.

NOT_GEOS            = 0                   ; Old C64 file, without GEOS header
                                          ;     (PRG, SEQ, USR, REL)

;--- The following are GEOS file types reserved for compatibility with old C64 files, that have
;    simply had a GEOS header placed on them. Users should be able to double click on files of
;    type BASIC and ASSEMBLY, whereupon they will be fast-loaded and executed from under BASIC.

BASIC               = 1                   ; C64 BASIC program, with a GEOS header attached.
                                          ; (Commodore file type PRG) to be used on programs that
                                          ; were executed before GEOS with:
                                          ;     LOAD "FILE",8
                                          ;     RUN
ASSEMBLY            = 2                   ; C64 ASSEMBLY program, with a GEOS header attached.
                                          ; (Commodore file type PRG) to be used on programs that
                                          ; were executed before GEOS with:
                                          ;     LOAD "FILE",8,1
                                          ;     SYS(Start Address)
DATA                = 3                   ; non-executable DATA file (PRG, SEQ, or USR)
                                          ; with a GEOS header attached for icon & notes ability.

;--- The following are file types for GEOS applications & system use:
;    ALL files having one of these GEOS file types should be of Commodore file type USR.

SYSTEM              = 4                   ; GEOS system file
DESK_ACC            = 5                   ; GEOS desk accessory file
APPLICATION         = 6                   ; GEOS application file
APPL_DATA           = 7                   ; data file for a GEOS application
FONT                = 8                   ; GEOS font file
PRINTER             = 9                   ; GEOS printer driver
INPUT_DEVICE        = 10                  ; INPUT device (mouse, etc.)
DISK_DEVICE         = 11                  ; DISK device driver
SYSTEM_BOOT         = 12                  ; GEOS system boot file (for GEOS, GEOS BOOT, GEOS KERNAL)
TEMPORARY           = 13                  ; temporary file type, for swap files.
                                          ; the deskTop will automatically delete all
                                          ; files of this type upon opening a disk.
AUTO_EXEC           = 14                  ; application to automatically be loaded & run
                                          ; just after booting, but before deskTop runs
INPUT_128           = 15                  ; 128 Input driver
NUM_FILE_TYPES      = 15                  ; # of file types, including NON_GEOS (=0)
```

GEOS file structure types

```
;--- Each "structure type" specifies the organization of data blocks on the disk,
;    and has nothing to do with the data in the blocks.

SEQUENTIAL          = 0                   ; standard T/S structure (like Commodore SEQ and PRG)
VLIR                = 1                   ; variable-length-indexed-record file (used for fonts,
                                          ; documents & some programs) this is a GEOS only format
```

Standard Commodore file types (supported by the old 1541 DOS)

```
DEL                 = 0                 ; deleted file
SEQ                 = 1                 ; sequential file
PRG                 = 2                 ; program file
USR                 = 3                 ; user file
REL                 = 4                 ; relative file
CBM                 = 5                 ; partition / sub-directory file,
                                        ; (only valid on 1581 drives).¥
```

Note:¥ GEOS only partially supports the CBM file type by handling it correctly during disk validation. See the Commodore 1581 DISK DRIVE User's Guide for more information on using 1581 partitions and sub-directories outside of GEOS.

File Header Block **fileHeader** $8100

Offsets into a GEOS file header block

```
O_GHIC_WIDTH        = $02               ; byte: width in bytes of file icon
O_GHIC_HEIGHT       = $03               ; byte: indicates height of file icon
O_GHIC_PIC          = $04               ; 64 bytes: picture data for file icon
O_GHCMDR_TYPE       = $44               ; byte: Comm. file type
O_GHGEOS_TYPE       = $45               ; byte: GEOS file type
O_GHSTR_TYPE        = $46               ; byte: GEOS file structure type
O_GHST_ADDR         = $47               ; 2 bytes: start address of file in memory
O_GHEND_ADDR        = $49               ; 2 bytes: end address of file in memory
O_GHST_VEC          = $4B               ; 2 bytes: initialization vector if file is application
O_GHFNAME           = $4D               ; 20 bytes, permanent filename
O_GHCNAME           = $4D               ; 20 bytes, data files permanent class name

O_128_FLAGS         = $60               ; 1 byte, flags to indicate if this program
                                        ; will run under the C128 OS in 40-column and
                                        ; in 80-column. These flags are valid for
                                        ; applications, desk accessories, and auto-exec files
                                        ; Bit 7: zero if runs in 40-column
                                        ; Bit 6: one if runs in 80-column
      ;--- Constants for 128 FLAGS
      CF_40         = $00               ; 64/128 40-column mode only
      CF_40_80      = $40               ; 64/128 40/80 and 80-column modes
      CF_64         = $80               ; 64 Only. Does not run under GEOS 128
      CF_128        = $C0               ; 128 80-column mode only

O_GH_AUTHOR         = $61               ; 20 bytes: author's name (only for application's)
O_GHAPDAT           = $89               ; application data
O_GHINFO_TXT        = $A0               ; offset to notes that are stored with the file
                                        ; and edited in the deskTop "get info" box
```

When file is an application's data file

```
O_GHP_FNAME         = $75               ; 20 bytes: permanent filename of parent application
;O_GHP_DISK         = $61               ; 20 bytes: disk name of parent application's disk
                                        ; (parent application's disk name was never implemented
                                        ;  in any GEOS application)
```

Font File Type Offsets *(into File Header Block)*

```
O_GHSETLEN          = $61
O_GHFONTID          = $80
O_GHPTSIZES         = $82
```

GetFile

```
;--- The following equates define file loading options for several of the GEOS file handling
;    routines like GetFile. These bit definitions are used to set the RAM variable r0/loadOpt.

ST_LD_AT_ADDR       = $01               ; "Load At Address": Load file at caller specified address
                                        ; instead of address file was saved from.

ST_LD_DATA          = $80               ; "Load Datafile": Used when application datafile is
                                        ; opened from deskTop. Used to indicate to application
                                        ; that r2 and r3 contain information about where to find
                                        ; the selected datafile.

ST_PR_DATA          = $40               ; "Print Datafile": Used when application datafile is
                                        ; selected for printing from deskTop. Used to indicate to
                                        ; application that r2 and r3 contain information about
                                        ; where to find the selected datafile.
```

VLIR

```
MAX_VLIR_RECS       = 127               ; Maximum number of VLIR records
```

Graphics

Constants for screen size

```
SC_BYTE_WIDTH       = 40                ; width of screen in bytes
SC_PIX_WIDTH        = 320               ; width of screen in pixels
SC_PIX_HEIGHT       = 200               ; height of screen in scanlines
SC_SIZE             = 8000              ; size of screen memory in bytes
```

Bits used to set dispBufferOn flag (controls which screens get written to)

```
ST_WR_FORE          = $80               ; write to foreground
ST_WR_BACK          = $40               ; write to background
ST_WRGS_FORE        = $20               ; limit GetString text entry to foreground screen
                                        ; this bit has no effect on anything outside of GetString
```

Values for graphics strings

```
MOVEPENTO           = 1                 ; move pen to x, y
LINETO              = 2                 ; draw line to x, y
RECTANGLETO         = 3                 ; draw a rectangle to x, y
NEWPATTERN          = 5                 ; set a new pattern
ESC_PUTSTRING       = 6                 ; start PutString interpretation
FRAME_RECTO         = 7                 ; draw frame of rectangle
PEN_X_DELTA         = 8                 ; move pen by signed word delta in x
PEN_Y_DELTA         = 9                 ; move pen by signed word delta in y
PEN_XY_DELTA        = 10                ; move pen by signed word delta in x & y
```

Values for PutDecimal calls

```
SET_LEFTJUST        = %10000000         ; left justified
SET_RIGHTJUST       = %00000000         ; left justified
SET_SUPRESS         = %01000000         ; no leading 0's
SET_NOSUPRESS       = %00000000         ; leading 0's
```

Screen colors

```
BLACK               = 0
WHITE               = 1
RED                 = 2
CYAN                = 3
PURPLE              = 4
GREEN               = 5
BLUE                = 6
YELLOW              = 7
ORANGE              = 8
BROWN               = 9
LTRED               = 10
DKGREY              = 11
GREY                = 12
MEDGREY             = 12
LTGREEN             = 13
LTBLUE              = 14
LTGREY              = 15
```

VDC Screen Colors

```
VDC_BLACK           = $00               ; black
VDC_DKGREY          = $01               ; dark grey
VDC_BLUE            = $02               ; dark blue
VDC_LTBLUE          = $03               ; light blue
VDC_GREEN           = $04               ; dark green
VDC_LGREEN          = $05               ; light green
VDC_CYAN            = $06               ; dark cyan
VDC_LTCYAN          = $07               ; light cyan
VDC_RED             = $08               ; dark red
VDC_LTRED           = $09               ; light red
VDC_PURPLE          = $0A               ; dark purple
VDC_LTPURPLE        = $0B               ; light purple
VDC_YELLOW          = $0C               ; dark yellow
VDC_LTYELLOW        = $0D               ; light yellow
VDC_LTGREY          = $0E               ; light grey
VDC_WHITE           = $0F               ; white
```

Values for SetColorMode

```
VDC_CLR0            = 0                 ; monochrome
VDC_CLR1            = 1                 ; 640x176 8x8 Color Cards, 16K VDC limited to 176 lines
VDC_CLR2            = 2                 ; 640x200 8x8 Color Cards
VDC_CLR3            = 3                 ; 640x200 8x4 Color Cards
VDC_CLR4            = 4                 ; 640x200 8x2 Color Cards
```

Hardware

CPU_DATA

```
;--- The following equates define the numbers written to the CPU_DATA register (location $0001 in
;    C64 and C128). These numbers control the hardware memory map of the C64.
;    Harmless to use on GEOS 128 but has no effect on RAM/ROM configuration.
;    (In GEOS 128 I/O is always mapped in).

IO_IN               = $35               ; 60K RAM, 4K I/O space in
RAM_64K             = $30               ; 64K RAM
KRNL_BAS_IO_IN      = $37               ; both Kernal and basic ROM's mapped into memory
KRNL_IO_IN          = $36               ; Kernal ROM and I/O space mapped in
KRNL_CH_BAS_IN      = $33               ; Kernal + basic + Char ROM
```

128 MMU

```
CIO_IN              = $7E               ; 60K RAM, 4K I/O
CRAM_64K            = $7F               ; 64K RAM
CKRNL_BAS_IO_IN     = $40               ; Kernal, I/O, basic
CKRNL_IO_IN         = $4E               ; Kernal, I/O
CIO_INB0            = $3E               ; Bank 0, 60K RAM, 4K I/O
```

SID

```
;--- Voice part offsets from voice bases sidVoc1,sidVoc2,sidVoc3
;                                       or from sidbase+O_VOICE[2..3]
O_FREQUENCY         = 0                 ; Examples:
O_FRELO             = 0                 ;
O_FREHI             = 1                 ; LoadW sidbase+O_VOICE3+O_PULSEWIDTH,#$800
O_PULSEWIDTH        = 2
O_PWLO              = 2                 ; tabReg: .byte O_SUREL,O_ATDCY,O_VCREG,O_FRELO,O_FREHI
O_PWHI              = 3
O_VCREG             = 4
O_ATDCY             = 5                 ; LoadB sidVoc2+O_ATDCY,#$34
O_SUREL             = 6

;--- control offsets from sidbase
O_FREQCUTOFF        = $15
O_FCLO              = $15
O_FCHI              = $16
O_RESFILT           = $17
O_SIGVOL            = $18
O_OSC3              = $1B
O_ENV3              = $1C

;voice offsets from sidbase
O_VOICE1            = 0
O_VOICE2            = 7
O_VOICE3            = $0E
```

VIC Chip

```
GRBANK0             = %11               ; bits indicate VIC RAM is $0000 - $3FFF, 1st 16K
GRBANK1             = %10               ; bits indicate VIC RAM is $4000 - $7FFF, 2nd 16K
GRBANK2             = %01               ; bits indicate VIC RAM is $8000 - $BFFF, 3rd 16K
GRBANK3             = %00               ; bits indicate VIC RAM is $c000 - $FFFF, 4th 16K

MOUSE_SPRNUM        = 0                 ; sprite number used for mouse
                                        ; (used to set VIC)

VIC_YPOS_OFF        = 50                ; position offset from 0 to position a
                                        ; hardware sprite at the top of the screen
                                        ; used to map from GEOS coordinates to hardware
                                        ; position coordinates.

VIC_XPOS_OFF        = 24                ; As above, offset from hardware 0
                                        ; position to left of screen, used to map GEOS
                                        ; coordinates to VIC

ALARMMASK           = %00000100         ; mask for the alarm bit in the cia chip
                                        ; interrupt control register
```

grcntrl1 graphics control register #1 D011

```
;--- ie msb raster /ECM /BMM /DEN /RSEL /y scroll bits.
ST_ECM              = $40
ST_BMM              = $20
ST_DEN              = $10
ST_25ROW            = $08
```

grcntrl2 graphics control resister #2 D016

```
;--- ie: RES/MCM/CSEL/x scroll bits
ST_MCM              = $10
ST_40COL            = $08               ;
```

grirqen Graphics chip interrupt enable register D01A

```
ST_RASEN            = %01               ; Enable raster interrupts
```

VDC

```
;vdccr              = $D600             ; Control Register
;vdcdr              = $D601             ; Data Register (R/W)

R0_HT               = 0                 ; Horizontal total
R1_HD               = 1                 ; Horizontal displayed
R2_HP               = 2                 ; Horizontal Sync
R3_VHW              = 3                 ; Vertical sync width | Horizontal sync width
R4_VT               = 4                 ; Vertical total
R5_VA               = 5                 ; Vertical total adjust
R6_VD               = 6                 ; Vertical characters displayed
R7_VP               = 7                 ; Vertical sync position
R8_IM               = 8                 ; Interlaced mode control
R9_CTV              = 9                 ; Rasterlines per character row
R10_CMS             = $0A               ; Cursor Mode / Cursor Start scan line
R11_CE              = $0B               ; Cursor end scan line
R12_DSH             = $0C               ; Start address of display memory in VDC RAM (word)
R13_DSL             = $0D               ;       (GEOS default $0000)
R14_CPH             = $0E               ; cursor position in text mode (word)
R15_CPL             = $0F
R16_LPV             = $10               ; Light Pen vertical position
R17_LPH             = $11               ; Light Pen horizontal position
R18_UAH             = $12               ; Update address (word). Location in VDC Memory for
R19_UAL             = $13               ; read/write using R31_DA and destination of block copies
R20_AAH             = $14               ; Attribute start address (word)
R21_AAL             = $15               ;  (GEOS default $3880, but not used in monochrome mode)
R22_CGW             = $16               ; Character Width
R23_CDV             = $17               ; Character Height
R24_VSS             = $18               ; b7:   1=Block copy
                                        ;       0=Block fill
                                        ; b6:   1=reverse video
                                        ; b5:   Text mode blink control
                                        ;       1=slow
                                        ;       0=fast
                                        ; b4-0: vertical smooth scroll
R25_HSS             = $19               ; b7:   1=bitmap mode
                                        ;       0=text mode
                                        ; b6:   Text mode attribute control
                                        ;       1= attributes enabled
                                        ; b5:   Text mode gap fill
                                        ;       1=semigraphic mode
                                        ; b4:   pixel clock
                                        ;       1=Double horizontal pixels
                                        ; b3-0: Horizontal smooth scroll
R26_FBC             = $1A               ; Foreground color / background color
R27_AI              = $1B               ; Address increment per row
R28_CB              = $1C               ; Character base address / RAM-type
R29_UL              = $1D               ; Underline scan line count
R30_WC              = $1E               ; Block copy/fill word count
R31_DA              = $1F               ; Data Register: Data byte pointed to by R18_UA
R32_BAH             = $20               ; Block copy source address. (word)
R33_BAL             = $21               ; (Copies to R18_UA Update address)
R34_DEB             = $22               ; Display enable begin
R35_DEE             = $23               ; Display enable end
R36_DRR             = $24               ; RAM refresh/scan Line
R37_HVS             = $25               ; hsync/vsync polarity. (8568 Only)
```

Keyboard:

```
KEY_QUEUE_SIZE      = 16               ; size of the keyboard queue (buffer)
KEY_REPEAT_COUNT    = 15               ; 1/4 second: auto-repeat time
                                       ; for the keyboard (maximum 254 and not 255)

KEY_F1              = 1
KEY_F2              = 2
KEY_F3              = 3
KEY_F4              = 4
KEY_F5              = 5
KEY_F6              = 6
KEY_LEFT            = 8                ; BACKSPACE
KEY_TAB             = 9
KEY_ENTER           = 13
KEY_F7              = 14
KEY_F8              = 15
KEY_UP              = 16
KEY_DOWN            = 17
KEY_HOME            = 18
KEY_CLEAR           = 19
KEY_LARROW          = 20
KEY_UPARROW         = 21
KEY_STOP            = 22
KEY_RUN             = 23
KEY_BPS             = 24
KEY_INSERT          = 28
KEY_DELETE          = 29
KEY_RIGHT           = 30
KEY_INVALID         = 31
```

128 Keys

```
KEY_NOSCRL          = 7
KEY_LF              = 10
KEY_HELP            = 25
KEY_ALT             = 26
KEY_ESC             = 27
```

Menu and Icon

Icon:

Default delay between flashes for icons and inverted time for menus delay is in vblanks.

```
SELECTION_DELAY     = 10                ; 1/6 of a second

MAX_ICONS           = 31                ;Attempting do use more then 31 will likely
                                        ;cause a system crash
```

iconSelFlag

*These equates are bit values for **iconSelFlag** that determine how an icon selection is indicated to the user. If ST_FLASH is set, ST_INVERT is ineffective.*

```
ST_NOTHING          = $00               ; indicate icon should not be changed
ST_FLASH            = $80               ; bit to indicate icon should flash
ST_INVERT           = $40               ; bit to indicate icon should be inverted

ST_FLSH_BIT         = 7                 ; icon should flash
ST_INVRT_BIT        = 6                 ; icon should invert
```

Offsets into the icon structure

```
OFF_NM_ICNS         = 0                 ; number of icons in structure
OFF_IC_XMOUSE       = 1                 ; mouse start x-position
OFF_IC_YMOUSE       = 3                 ; mouse start y-position
```

Offsets into an icon record in icon structure

Constant declarations from HGG. Adopted for official constants in geoProgrammer 2.x+

```
OFF_I_PIC           = 0                 ; picture pointer for icon
OFF_I_X             = 2                 ; x-position of icon
OFF_I_Y             = 3                 ; y-position of icon
OFF_I_WIDTH         = 4                 ; width of icon
OFF_I_HEIGHT        = 5                 ; height of icon
OFF_I_EVENT         = 6                 ; pointer to service routine for selected icon
OFF_I_NEXT          = 8                 ; Size of Icon Record
```

Constant declarations from geoProgrammer 1.x. Included for backwards compatibility.

```
OFF_PIC_ICON        = 0                 ; picture pointer for icon
OFF_X_ICON_POS      = 2                 ; x-position of icon
OFF_Y_ICON_POS      = 3                 ; y-position of icon
OFF_WDTH_ICON       = 4                 ; width of icon
OFF_HEIGHT_ICON     = 5                 ; height of icon
OFF_SRV_RT_ICON     = 6                 ; pointer to service routine for icon
OFF_NX_ICON         = 8                 ; next icon in icon structure
```

Menu:

```
MAX_M_ITEMS         = 15
MAX_M_NESTING       = 4
M_HEIGHT            = 14
```

Types

```
HORIZONTAL          = %00000000
VERTICAL            = %10000000
CONSTRAINED         = %01000000
UN_CONSTRAINED      = %00000000
```

Offsets

```
OFF_MY_TOP          = 0                 ; offset to y-position of top of menu
OFF_MY_BOT          = 1                 ; offset to y-position of bottom of menu
OFF_MX_LEFT         = 2                 ; offset to x-position of left-side of menu
OFF_MX_RIGHT        = 4                 ; offset to x-position of right-side of menu
OFF_M_ATTRIBUTE     = 6                 ; offset to Alignment | Number of items
OFF_NUM_M_ITEMS     = 6                 ; Offset to last menu byte. (deprecated)
OFF_1ST_M_ITEM      = 7                 ; offset to record for 1st menu item in structure
```

Menu Item Offsets

```
OFF_TEXT_ITEM       = 0                 ; Pointer to null-terminated string for this menu item
OFF_TYPE_ITEM       = 2                 ; Selection type (sub-menu, event, dynamic sub-menu)
OFF_POINTER_ITEM    = 3                 ; Pointer to sub-menu data structure, event routine, or
                                        ; dynamic sub-menu routine, depending on selection type
```

Actions

```
SUB_MENU            = $80               ; for setting byte in menu table that indicates
DYN_SUB_MENU        = $40               ; whether the menu item causes action
MENU_ACTION         = $00               ; or sub-menu
```

Mouse

Bit flags for mouseOn variable

```
SET_MSE_ON          = %10000000
SET_MENUON          = %01000000
SET_ICONSON         = %00100000

MOUSEON_BIT         = 7
MENUON_BIT          = 6
ICONSON_BIT         = 5
```

Default Reset Count for dblClickCount

```
CLICK_COUNT         = 30
```

Memory Map

```
zpage               = $00               ; start of Zero Page
APP_ZPL             = $70               ; application dedicated zero page block. 16 bytes
APP_ZIO             = $80               ; swappable Kernal I/O / application zpage space
APP_ZPH             = $FB               ; application Dedicated zero page block. 4 bytes
APP_LVAR            = $0200             ; application low variable space
KERNAL_VECTORS      = $031A             ; Kernal vectors when ROM is switched in
APP_LRAM¥           = $0334             ; application low space
APP_RAM             = $0400             ; start of application space
BACK_SCR_BASE       = $6000             ; base of background screen
PRINTBASE           = $7900             ; load address for print drivers
APP_VAR             = $7F40             ; application variable space
OS_VARS             = $8000             ; OS variable base
SPRITE_PICS         = $8A00             ; base of sprite pictures
COLOR_MATRIX        = $8C00             ; video color matrix
DISK_BASE           = $9000             ; disk driver base address
SCREEN_BASE         = $A000             ; base of foreground screen
;                     $BF40             ; start of C64 low OS code space
;                     $BF80             ; start of C128 low OS code space
OS_ROM              = $C000             ; start of OS code space
OS_JUMPTAB          = $C100             ; start of GEOS jump table
vicbase             = $D000             ; video interface chip base address
sidbase             = $D400             ; sound interface device base address
ctab                = $D800             ; color table for text mode.
                                        ; Note: GEODEBUGGER and GEOBASIC use text mode
cia1base            = $DC00             ; 1st communications interface adaptor (CIA)
cia2base            = $DD00             ; second CIA chip
EXP_BASE            = $DF00             ; base address of RAM-Expansion unit

MSE128_BASE         = $FD00             ; start of 128 input driver
END_MSE128          = $FE80             ; end of 128 input driver

MOUSE_JMP           = $FE80             ; start of mouse jump table
MOUSE_BASE          = $FE80             ; start of input driver
END_MOUSE           = $FFFA             ; one byte past the end of the input driver
NMI_VECTOR          = $FFFA             ; NMI vector location
RESET_VECTOR        = $FFFC             ; reset vector location
IRQ_VECTOR          = $FFFE             ; interrupt vector location
```

¥**Important**: The APP_LRAM region is used by the debugger. See **Appendix E: Memory Maps** for more information.

Process:

```
MAX_PROCESSES       = 20                ; maximum number of processes
SLEEP_MAX           = 20                ; maximum number of sleeping threads
PSIZE               = 4                 ; size of process table entry
```

Possible values for processFlags

```
SET_RUNABLE         = %10000000         ; runnable flag
SET_BLOCKED         = %01000000         ; process blocked flag
SET_FROZEN          = %00100000         ; process frozen flag
SET_NOTIMER         = %00010000         ; not a timed process flag

RUNABLE_BIT         = 7                 ; runnable flag
BLOCKED_BIT         = 6                 ; process blocked flag
FROZEN_BIT          = 5                 ; process frozen flag
NOTIMER_BIT         = 4                 ; not a timed process flag
```

pseudoregisters

```
r0L                 = $02               ; Descriptive access to low-byte of registers:
r1L                 = $04               ;
r2L                 = $06               ; To conserve symbol names, only the basename and
r3L                 = $08               ; the high byte name should be sent to the debugger.
r4L                 = $0A               ;
r5L                 = $0C               ; Example:
r6L                 = $0E               ; Both of these will work in the debugger
r7L                 = $10               ;   m r0
r8L                 = $12               ;   m r0H
r9L                 = $14
r10L                = $16               ; This will not work when not sending low-byte symbols.
r11L                = $18               ;   m r0L
r12L                = $1A
r13L                = $1C               ; use m r0 instead.
r14L                = $1E
r15L                = $20


a0                  = $FB               ; Default names for user registers:
a1                  = $FD               ; a0-a9 conflict with actual address names in the
a2                  = $70               ; debugger so they are included as constants that do
a3                  = $72               ; not get exported to the .dbg file.
a4                  = $74
a5                  = $76               ; in the debugger refer to these by the low/high names
a6                  = $78
a7                  = $7A               ; Example use in debugger:
a8                  = $7C               ;   m a0L
a9                  = $7E               ;   m a0H

                                        ; Even if a0-a9 were exported as symbols the following
                                        ; command would still fail to do as expected.
                                        ;   m a0
                                        ;
                                        ; Will return the contents of the address $A0 instead
                                        ; of the address referenced by the symbol name a0.
```

Text

Bit flags in mode

```
SET_UNDERLINE       = %10000000
SET_BOLD            = %01000000
SET_REVERSE         = %00100000
SET_ITALIC          = %00010000
SET_OUTLINE         = %00001000
SET_SUPERSCRIPT     = %00000100
SET_SUBSCRIPT       = %00000010
SET_PLAINTEXT       = 0

UNDERLINE_BIT       = 7
BOLD_BIT            = 6
REVERSE_BIT         = 5
ITALIC_BIT          = 4
OUTLINE_BIT         = 3
SUPERSCRIPT_BIT     = 2
SUBSCRIPT_BIT       = 1
```

PutChar constants

```
EOF                 = 0                 ; end of text object
NULL                = 0                 ; end of string
BACKSPACE           = 8                 ; move left a card
TAB                 = 9
FORWARDSPACE        = 9                 ; move right one card
LF                  = 10                ; move down a card row
HOME                = 11                ; move to left top corner of screen
UPLINE              = 12                ; move up a card line
PAGE_BREAK          = 12                ; page break
CR                  = 13                ; move to beginning of next card row
ULINEON             = 14                ; turn on underlining
ULINEOFF            = 15                ; turn off underlining
ESC_GRAPHICS        = 16                ; escape code for graphics string
ESC_RULER           = 17                ; ruler escape
REV_ON              = 18                ; turn on reverse video
REV_OFF             = 19                ; turn off reverse video
GOTOX               = 20                ; use next word as x cursor position
GOTOY               = 21                ; use next byte as y cursor position
GOTOXY              = 22                ; use next word as x and then the next byte as y cursor pos
NEWCARDSET          = 23                ; use next two bytes as new font id, then a style byte
BOLDON              = 24                ; turn on BOLD characters
ITALICON            = 25                ; turn on ITALIC characters
OUTLINEON           = 26                ; turn on OUTLINE characters
PLAINTEXT           = 27                ; plain text mode
USELAST             = 127               ; erase character.  Used with GetRealSize to get the size
                                        ; of the previously printed character.
SHORTCUT            = 128               ; (%10000000) Mask bit for a shortcut character. Any
                                        ; character read of the keyboard buffer with bit 7 set
                                        ; will be a shortcut key.¥
```

¥**Example**: User enters C= + 'A'
Character in **keyData** will be ('A' | SHORTCUT)

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
alarmSetFlag:	851C	851C	1	FALSE	No	TRUE if the alarm is set for GEOS to monitor, else FALSE
alarmTmtVector:	84AD	84AD	2	NULL	Yes	Address of a service routine for the alarm clock time-out (ringing, graphic etc.) that the application can use if necessary.
alphaFlag:	84B4	84B4	1	0	Yes	Flag for alphanumeric string input 0xxx xxxx if not getting text input 11xx xxxx if getting text input bit Description --- -- b7: Flag indicating alphanumeric input is on b6: Flag indicating prompt is visible b5-0: Counter before prompt flashes
a0: a0L/a0H(+1) **a1**: ... **a2**: **a3**: **a4**: **a5**: **a6**: **a7**: **a8**: **a9**:	FB FD 70 72 74 76 78 7A 7C 7E	FB FD 70 72 74 76 78 7A 7C 7E	2 2 2 2 2 2 2 2 2 2	None	No	Place holder pseudoregister names for application use. GEODEBUGGER sees the names **a0-a9** as address's. Because of this, it is recommended to provide descriptive names of the .zsect space as needed on a per application basis. Example: .zsect APP_ZPL ;(APP_ZPL=$70) zSysType: .block 1 ;bit flags for runtime system zDevApp: .block 1 ;device number of application zDevData: .block 1 ;device number of data file z0: .block 2 ;general purpose register. ...
appMain:	849B	849B	2	NULL	Yes	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**.
backBufPtr:	–	131B†	16	None	No	Screen pointer where the back buffer came from. Resides in backRAM of C128.
backXBufNum:	–	132B†	8	None	No	For each sprite, there is one byte here for how many bytes wide the corresponding sprite is. Used by C128 soft sprite routines and resides in backRAM.
backYBufNum:	–	1333†	8	None	No	For each sprite, there is one byte here for how many scanlines high the corresponding sprite is. Used C128 by soft sprite routines and resides in backRAM.
bakclr0: **bakclr1**: **bakclr2**: **bakclr3**:	D021 D022 D023 D024	D021 D022 D023 D024	1 1 1 1	DKGREY WHITE RED CYAN	No	Hardware registers to control text screen background colors. b7-4 = not used. always 1's b0-3 = color for text background. bakclr(1-3) only used in multi color mode. (Not used by GEOS).

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
baselineOffset:	26	26	1	6	Yes	Offset from top line to baseline in character set. i.e. it changes as fonts change. Default $06 - for BSW 9 font.
bkvec:	0316	0316	2	Kernal Def	No	BRK instruction vector when ROMs are switched in.
bootName:	C006	C006	9	->	No	This is the start of the "GEOS BOOT" string. The Gateway version of GEOS has "GATEWAY" at this location.
BRKVector:	84AF	84AF	2	System	Yes	Vector to the routine that is called when a BRK instruction is encountered. The default is to vector to the operating system error dialog box routine.
c128Flag:	C013	C013	1	None	No	Defines current machine type. b7: 0 = C64 1 = C128 b6-0: not used
cardDataPntr:	2C	2C	2	$D2DC	Yes	Pointer to the actual card graphic data for the current font in use. Default address is the location of the BSW 9 system font.
cia1cra:	DC0E	DC0E	1	None	No	Timer control register a.
cia1crb:	DC0F	DC0F	1	None	No	Timer control register b. b7: 1=Setting time sets the alarm.
cia1ddra:	DC02	DC02	1	None	No	Data direction register a. 0=read only, 1=write only
cia1ddrb:	DC03	DC03	1	None	No	Data direction register b. 0=read only, 1=write only
cia1icr:	DC0D	DC0D	1	None	No	Interrupt control register.
cia1pra:	DC00	DC00	1	None	No	Peripheral data register a. 1. b7-b0: keyboard matrix columns. control port 2: b4: joystick fire button. b3-b0: joystick direction. -or- b4: mouse left button. (0=pressed) b0: mouse right button. (0=pressed)
cia1prb:	DC01	DC01	1	None	No	Peripheral data register b. b7-b0: keyboard matrix rows. control port 1: b4: joystick fire button. b3-b0: joystick direction. -or- b4: mouse left button. (0=pressed) b0: mouse right button. (0=pressed)
cia1sdr:	DC0C	DC0C	1	None	No	Serial data register.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
cia1tahi: **cia1talo**:	DC05 DC04	DC05 DC04	1 1	None	No	high byte of counter Timer A. Programmable counter (word)
cia1tbhi: **cia1tblo**:	DC07 DC06	DC07 DC06	1 1	None	No	high byte of counter Timer B. Programmable counter (word)
cia1tod10ths:	DC08	DC08	1	None	No	10ths of second register: (R/W) (GEOS time) b3-b0: range (0-9) **Important**: Writting or reading **cia1tod10ths** starts the time of day timer.
cia1todhr:	DC0B	DC0B	1	None	No	Hours – AM; PM register: (R/W) BCD (GEOS time) b7: 0=AM, 1=PM b6-b5: not used. always 0 b4: tenths place; range (0-1) b3-b0: ones place; range (0-9) **Important**: Writing or reading **cia1todhr** stops the time of day timer.
cia1todmin:	DC0A	DC0A	1	None	No	Minutes register: (R/W) BCD (GEOS time) b6-b4: tenths place; range (0-5) b3-b0: ones place; range (0-9)
cia1todsec:	DC09	DC09	1	None	No	Seconds register. (R/W) BCD (GEOS time) b6-b4: tenths place; range (0-5) b3-b0: ones place; range (0-9)
cia2cra:	DD0E	DD0E	1	None	No	Timer control register a.
cia2crb:	DD0F	DD0F	1	None	No	Timer control register b.
cia2ddra:	DD02	DD02	1	None	No	Data direction register a. 0=read only, 1=write only
cia2ddrb:	DD03	DD03	1	None	No	Data direction register b. 0=read only, 1=write only
cia2icr:	DD0D	DD0D	1	None	No	Interrupt control register.
cia2pra:	DD00	DD00	1	%01	No	Peripheral data register a. VIC Banks b7-3: serial bus `%00` C000-FFFF b2: RS232 TXD `%01` 8000-BFFF (GEOS default) b1-0: VIC bank `%10` 4000-7FFF `%11` 0000-3FFF
cia2prb:	DD01	DD01	1	None	No	Peripheral data register b. Used by RS232 serial routines.
cia2sdr:	DD0C	DD0C	1	None	No	Serial data register.
cia2tahi: **cia2talo**:	DD05 DD04	DD05 DD04	1 1	None	No	Timer a. high byte of word sized counter value. low byte of counter.
cia2tbhi: **cia2tblo**:	DD07 DD06	DD07 DD06	1 1	None	No	Timer b. high byte of word sized counter value. low byte of counter.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
cia2tod10ths:	DD08	DD08	1	None	No	10ths of sec register: (read/write) b3-b0: range (0-9)
cia2todhr:	DD0B	DD0B	1	None	No	Hours – AM; PM register: (read/write BCD) b7: 0=AM, 1=PM b6-b4: tenths place; range (0-5) b3-b0: ones place; range (0-9)
cia2todmin:	DD0A	DD0A	1	None	No	Minutes register: (read/write BCD) b7-b4: tenths place; range (0-5) b3-b0: ones place; range (0-9)
cia2todsec:	DD09	DD09	1	None	No	Seconds register: (read/write BCD) b7-b4: tenths place; range (0-5) b3-b0: ones place; range (0-9)
clkreg:	–	D030	1	%1	No	C128 clock speed register: b3-7: = not used (always 1's) b2: = test bit. Should always be 0. b0: 0 = 1MHz; 1 = 2MHz
config:	–	FF00	1	CIO_IN	No	C128 **MMU configuration register**.
CPU_DATA:	01	01	1	RAM_64K	No	**6510 data register**. Controls the hardware memory map of the C64.
CPU_DDR:	00	00	1	%101111	No	**6510 data direction register**. **Note[3]**: Writing $00 to this address will disable output to **CPU_DATA** register. This may cause unexpected results.
curDevice:	BA	BA	1	8	No	Current serial device number. See **curDrive** for more information.
curDirHead:	8200	8200	256	None	No	Buffer containing header information for the disk in currently selected drive. Structure: **Directory Header**
curDrive:	8489	8489	1	8	No	Device number of the currently active disk drive. Allowed values are 8 – 11.
curEnable:	–	1300†	1	None	No	This is an image of the C64 **mobenble** register.
curHeight:	29	29	1	9	Yes	Card height in pixels of the current font in use.
curIndexTable:	2A	2A	2	$D218	Yes	Pointer to the table of sizes, in bytes, of each card in of the current font.
curmobx2:	–	1302†	1	None	No	Image of the C64 **mobx2** register. Used for C128 soft sprites. In backRAM
curmoby2:	–	1301†	1	None	No	Image of C64 **moby2** register. Used for C128 soft sprites. In backRAM.
curPattern:	22	22	2	$D010	Yes	Pointer to the first byte of the graphics data for the current pattern in use. **Note**: Each pattern is 1 byte wide and 8 bytes high, to give an 8x8 bit pattern.
curRecord:	8496	8496	1	0	No	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	Address (hex)		Size	Default	Saved	Description
	64	128				

currentMode:	2E	2E	1	0	Yes	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 manipulating these bits are as follows.
						
						`Bit   Style            Constant`
						`---   -----            --------`
						`b7:   Underline        SET_UNDERLINE   = %10000000`
						`b6:   Bold             SET_BOLD        = %01000000`
						`b5:   Reverse          SET_REVERSE     = %00100000`
						`b4:   Italics          SET_ITALIC      = %00010000`
						`b3:   Outline          SET_OUTLINE     = %00001000`
						`b2:   Superscript      SET SUPERSCRIPT = %00000100`
						`bl:   Subscript        SET_SUBSCRIPT   = %00000010`
						`b0:   Unused`
						To Clear 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

curSetWidth:	27	27	2	$3C	Yes	Card width in pixels for the current font
curType:	88C6	88C6	1	Drv 8 type	No	Holds the current drive 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
curXpos0:	–	1303[†]	16	None	No	The current x-positions of the C128 soft sprites. In BackRAM
curYpos0:	–	1313[†]	8	None	No	The current y-positions of the C128 soft sprites. In BackRAM
dataDiskName:	8453	8453	18	None	No	Holds the disk name that an application's data file is on.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
dataFileName:	8442	8442	17	None	No	Name of a data file to open. The name is passed to the parent application so the file can be opened. (Null terminated)
dateCopy:	C018	C018	3	YMD	No	Copy of system variables **year**, **month**, and **day**.
day:	8518	8518	1	20	No	Holds the value for current **day**.
dblClickCount:	8515	8515	1	0	No	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.
dir2Head:	8900	8900	256	None	No	Second BAM block. Used by 1571 and 1581 drives.
dir3Head:	9C80	9C80	256	None	No	Third BAM block. Used by 1581 drive.
dirEntryBuf:	8400	8400	30	0	No	Buffer used to build a file's directory entry. Structure: **Directory Entry**
diskBlkBuf:	8000	8000	256	0	No	General disk block buffer. Initialized to all zeros.

diskOpenFlg:	848A	848A	1	0	No	This flag byte is not used by the Kernal. It is initialized to FALSE ($00) when the entire block is cleared at startup. It is never touched again by the Kernal.
						It is used by the DeskTop. The flag follows the status of the currently selected drive. If the disk is open this byte is set to TRUE. When DeskTop closes a disk it sets **diskOpenFlg** to FALSE.
						Note: **diskOpenFlg** could be freely used by applications to perform the same function as the DeskTop (or for any other purpose as well). But it would be up to the application to set and maintain the value of the byte.
						Note: **diskOpenFlg** can be used as a base for indexing into **turboFlags**.
						Example:
						`ldx   curDrive`
						`lda   diskOpenFlg,X`

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
dispBufferOn:	`2F`	`2F`	1	$C0	Yes	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 ST_WR_FORE = %10000000 ; $80 ;b7 ST_WR_BACK = %01000000 ; $40 ;b6 ST_WRGS_FORE = %00100000 ; $20 ;b5 Default is ST_WR_FORE | ST_WR_BACK ; $C0 Use ST_WR_FORE (write to foreground) and ST_WR_BACK (write to background) to access these bits. **Note**: Dialog Boxes use (ST_WR_FORE | ST_WRGS_FORE) **Important**: %00xxxxxxxx is an undefined state and will result in sending most graphic operations to the center of the display area.
dlgBoxRamBuf:	`851F`	`851F`	417	None	n/a	This is the buffer for variables that are saved when desk accessories or dialog boxes are run.
doRestFlag:	–	`1B54`[†]	1	0	No	Flag needed because of overlapping soft sprite problems on C128. 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 BackRAM.
DrACurDkNm:	`841E` `842E`	`841E` `842E`	16 2	None	No	Disk name of the current disk in drive A, 16 characters padded with $A0. 2 character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
DrBCurDkNm:	`8430` `8440`	`8430` `8440`	16 2	None	No	Disk name of the current disk in drive B, 16 characters padded with $A0. 2 character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
DrCCurDkNm:	`88DC` `88EC`	`88DC` `88EC`	16 2	None	No	Disk name of the current disk in drive C, 16 characters padded with $A0. 2 character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
DrDCurDkNm:	`88EE` `88FE`	`88EE` `88FE`	16 2	None	No	Disk name of the current disk in drive D, 16 characters padded with $A0. 2 character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
driveData:	88BF	88BF	4	None	No	One byte is reserved for each disk drive, to be used by the disk driver. Each driver may use it differently.
driveNdxType:	(8486)	(8486)	–	–	–	Label used for indexing **driveType** with drive number. `lda driveNdxType,x`
driveType:	848E	848E	4	Drv 8 type	No	There are 4 bytes at location **driveType**, one for each of four possible drives. 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 Description ---------- ---- ------------------------------ 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
extclr:	D020	D020	1	BLACK	No	Exterior (border) color.

faultData:	84B6	84B6	1	0	Yes	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 signaled when the mouse is outside the current menu area. The bits for signaling 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`

fileHeader:	8100	8100	256	0	No	Header Block buffer for a GEOS file. VLIR routines use this for the files VLIR index table. Structure: **File Header Block**
fileSize:	8499	8499	2	None	No	Current size (in blocks) of a file. It is pulled in from and written to the file's directory entry.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
fileTrScTab:	8300	8300	256	0	No	Track and sector chain for a file of maximum size of 32258 bytes.
fileWritten:	8498	8498	1	None	No	Flag indicating if the currently open file has been written to since the last update of its index table and the BAM.
firstBoot:	88C5	88C5	1	0	No	This flag is changed from $00 to $FF when the deskTop comes up after booting.
fontTable:	26	26	8	Default Font	Yes	**fontTable** is a label for the beginning of variables for the current font in use. These variables are **baselineOffset**, **curSetWidth**. **curHeight**, **curIndexTable**, and **cardDataPntr**, **currentMode** is also saved/restored to **saveFontTab**. For more information, see documentation on these variables.
gatewayFlag:	C007	C007	1	None	No	The gateway version of GEOS there will be a 'A' ($41) at this location. Gateway has a different boot string at $C006. "GATEWAY". **Note**: variable name adopted from cc65 for cross source compatibility.
georambs:	DFFF	DFFF	1	None	No	High 5 bits of GEORAM block select register.
georampg:	DE00	DE00	$100			GEORAM memory page.
georamps:	DFFE	DFFE	1			Low 6 bits of GEORAM page select register.
graphMode:	3F	3F	1	None	No	Current video mode for C128. 40-Column: GR_40=$00 80-Column: GR_80=$80 (%10000000) sample usage **graphMode** `bbsf 7,graphMode, Do80ColStuff`
grcntrl1:	D011	D011	1	$3B	No	Graphics control register #1, ie: msb raster /ECM /BMM /DEN /RSEL /y scroll bits. Constants defined for use with above register: ;b7: bit 9 of **rasreg** ST_ECM = $40 ;b6: 1=extended color text mode ST_BMM = $20 ;b5: 1=bitmap graphics mode ST_DEN = $10 ;b4: 0=blank screen to border color ST_25ROW = $08 ;b3: 1=25 rows, 0=24 rows ;b2-0: vertical fine scroll
grcntrl2:	D016	D016	1	ST_40COL	No	Graphics control resister #2, e.g: RES/MCM/CSEL/x scroll bits defined for use with above register: ST_MCM = $10 ST_40COL = $08
grirq:	D019	D019	1	n/a	No	Graphics chip interrupt register.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
grirqen:	D01A	D01A	1	%01	No	Graphics chip interrupt enable register: b0: enable raster interrupt in grirqen ST_RASEN = %01
grmemptr:	D018	D018	1	$38 Effective $8C00 $A000	No	Graphics memory pointer VM13-VM10|CB13-CB11. ie video matrix and character base. Defines offset within VIC bank address as set in **cia2pra**. b7-4: Offset to video (color) matrix. %0111 $0C00 (1K pages) b3-1: Offset to bitmap/character image memory. %100 $2000 (2K pages)
hour:	8519	8519	1	12	No	Variable for **hour**. 0-23
iconSelFlag:	84B5	84B5	1	0	Yes	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.
inputData:	8506	8506	4	None	No	This is where input drivers pass device specific information to applications that want it.
inputDevName:	88CB	88CB	17	None	No	Name of the current input device, e.g. COMM MOUSE for Commodore mouse.
inputVector:	84A5	84A5	2	NULL	Yes	Pointer to routine to call on input device change.
intBotVector:	849F	849F	2	NULL	Yes	Vector to routine to call after the operating system interrupt code has run. This allows applications to have interrupt level routines.
interleave:	848C	848C	1	8	No	Used by **BlkAlloc** routine as the desired **interleave** when selecting free blocks for a disk chain.
intTopVector:	849D	849D	2	NULL	Yes	Vector to routine to call before operating system interrupt code is run. It allows applications to interrupt level routines.
invertBuffer:	–	1CED[†]	80	None	No	Buffer area used to speed up the 80-column **InvertLine** routine. Resides in backRAM.
irqvec:	0314	0314	2	Kernal Def	No	IRQ vector when ROMs are switched in.
IRQVector:	FFFE	FFFE	2	$FAA2	No	Interrupt request vector.

Name	Address (hex)					
	64	128	Size	Default	Saved	Description
isGEOS:	848B	848B	1	Disk	No	Flag to indicate whether the current disk is a GEOS disk.
kernalVectors:	031A	031A	26	Kernal Def	No	Location of kernal vectors when ROMS are switched in.
keydata:	8504	8504	1	0	No	Holds the ASCII value of the current last key that was pressed. Used by keyboard service routines.
keyreg:	—	D02F	1	None	No	C128 keyboard register for # pad & other keys: b7-b3: = Not Used (always 1's) b2-b0: = Scan Rows 8.9 and 10
keyVector:	84A3	84A3	2	NULL	Yes	Vector to routine to call on keypress. **Note**[3]: The 26A1 default address listed in the original HGG is in the DeskTop application. When an application starts, **keyVector** is NULL.
leftMargin:	35	35	2	0	Yes	Leftmost point for writing characters. Doing a carriage return will return to this point. If an attempt is made to write to the left of **leftMargin**, the routine pointed to by **StringFaultVec** is called.
lpxpos:	D013	D013	1	None	No	Light pen x-position.
lpypos:	D014	D014	1	None	No	Light pen y-position.
maxMouseSpeed:	8501	8501	1	$7F	No	Maximum speed for mouse cursor.
mcmclr0:	D025	D025	1	None	Yes	Multi-color mode color 0.
mcmclr1:	D026	D026	1	None	Yes	Multi-color mode color 1.
menuNumber:	84B7	84B7	1	0	Yes	Number of currently working menu.
minMouseSpeed:	8502	8502	1	$1E	No	Minimum speed for mouse cursor.
minutes:	851A	851A	1	0	No	Holds the **minutes** for time of day clock.
mmucr:	—	D500	1	CIO_IN	No	C128 MMU **Configuration register**. (Mirrored by **config** at FF00)
mmulcra/b/c/d:	—	FF01	1x4	—	No	C128 MMU Load configuration registers a,b,c and d.
mmup0H:	—	D508	1	%xxxxxxx0	No	C128 MMU Zero Page bank pointer.
mmup0L:	—	D507	1	0	No	C128 MMU Zero Page page pointer.
mmup1H:	—	D50A	1	%xxxxxxx0	No	C128 MMU Stack Page bank pointer.
mmup1L:	—	D509	1	1	No	C128 MMU Stack Page page pointer.
mmumcr:	—	D505	1	$37	No	C128 MMU Mode configuration register. (b7 value follows keyboard 40/80 switch. 0=80 Cols: Switch down).
mmurcr:	—	D506	1	$40	No	C128 MMU **RAM Configuration register**.
mmupcra/b/c/d:	—	D501	1x4	—	No	C128 MMU Pre configuration registers a,b,c and d.
mob0clr:	D027	D027	1	None	No	Color of sprite 0.
mob0xpos:	D000	D000	1	None	Yes	x-position of sprite 0.
mob0ypos:	D001	D001	1	None	Yes	y-position of sprite 0.

Name	Address (hex) 64	128	Size	Default	Saved	Description
mob1clr:	D028	D028	1	None	Yes	Color of sprite 1.
mob1xpos:	D002	D002	1	None	Yes	x-position of sprite 1.
mob1ypos:	D003	D003	1	None	Yes	y-position of sprite 1.
mob2clr:	D029	D029	1	None	Yes	Color of sprite 2.
mob2xpos:	D004	D004	1	None	Yes	x-position of sprite 2.
mob2ypos:	D005	D005	1	None	Yes	y-position of sprite 2.
mob3clr:	D02A	D02A	1	None	Yes	Color of sprite 3.
mob3xpos:	D006	D006	1	None	Yes	x-position of sprite 3.
mob3ypos:	D007	D007	1	None	Yes	y-position of sprite 3.
mob4clr:	D02B	D02B	1	None	Yes	Color of sprite 4.
mob4xpos:	D008	D008	1	None	Yes	x-position of sprite 4.
mob4ypos:	D009	D009	1	None	Yes	y-position of sprite 4.
mob5clr:	D02C	D02C	1	None	Yes	Color of sprite 5.
mob5xpos:	D00A	D00A	1	None	Yes	x-position of sprite 5.
mob5ypos:	D00B	D00B	1	None	Yes	y-position of sprite 5.
mob6clr:	D02D	D02D	1	None	Yes	Color of sprite 6.
mob6xpos:	D00C	D00C	1	None	Yes	x-position of sprite 6.
mob6ypos:	D00D	D00D	1	None	Yes	y-position of sprite 6.
mob7clr:	D02E	D02E	1	None	Yes	Color of sprite 7.
mob7xpos:	D00E	D00E	1	None	Yes	x-position of sprite 7.
mob7ypos:	D00F	D00F	1	None	Yes	y-position of sprite 7.
mobbakcol:	D01F	D01F	1	None	No	Sprite to background collision register.
mobenble:	D015	D015	1	None	Yes	Sprite enable bits.
mobmcm:	D01C	D01C	1	0	Yes	Sprite multi-color mode select.
mobmobcol:	D01E	D01E	1	0	No	Object to object collision register.
mobprior:	D01B	D01B	1	0	Yes	Object to background priority.
mobx2:	D01D	D01D	1	0	Yes	Double object size in x.
moby2:	D017	D017	1	0	Yes	Double object size in y.
month:	8517	8517	1	9	No	Holds the **month** for time of day clock.
mouseAccel:	8503	8503	1	$75	No	Acceleration of mouse cursor.
mouseBottom:	84B9	84B9	1	199	Yes	Bottom most position for mouse cursor. Normally set to bottom of the screen.
mouseData:	8505	8505	1	None	No	State of mouse button: high bit set if button is released; clear if pressed.
mouseFaultVec:	84A7	84A7	2	System Handler	Yes	Vector to routine to call when mouse goes outside region defined for mouse position or when mouse goes off of a menu.

Name	Address (hex) 64	128	Size	Default	Saved	Description
mouseLeft:	84BA	84BA	2	0	Yes	Left most position for mouse.

mouseOn:	30	30	1	$E0	Yes	Flag indicating that the mouse/Menu/Icon is on. Bit usage and constants for accessing them are as follows:
						`Bit  Mode              Constant`
						`---  ---------------   ----------------------`
						`b7:  mouse on if set   SET_MSE_ON  =%10000000`
						`b6:  menus on if set   SET_MENUON  =%01000000`
						`b5:  icons on if set   SET_ICONSON =%00100000`
						`b4 – b0: not used`

mousePicData:	84C1	84C1	64	Mouse Pic.	No	64-byte array for the mouse sprite picture.
mouseRight:	84BC	84BC	2	40=319 80=639	Yes	Right most position for mouse.
mouseSave:	-	1B55†	24	None	No	Screen data for what is beneath mouse soft sprite. Resides in backRam.
mouseTop:	84B8	84B8	1	0	Yes	Top most position for mouse.
mouseVector:	84A1	84A1	2	System Handler	Yes	Routine to call on a mouse button press. See also: **otherPressVec**
mouseXPos:	3A	3A	2	None	No	Mouse x-position.
mouseYPos:	3C	3C	1	0	No	Mouse y-position.
msbxpos:	D010	D010	1	None	Yes	Sprite #0-7 Bit 8 of x-coordinates. Enables byte sized x-coordinate to be > 255. b0: -> Sprite 0
msePicPtr:	31	31	2	$84C1	Yes	Pointer to the mouse graphics data. default = **mousePicData**
nationality:	C010	C010	1	0 USA	No	Nationality of Kernal. 0 American 1 German 2 French (France & Belgium)† 3 Dutch† 4 Italian† 5 Swiss (Switzerland)† 6 Spanish† 7 Portuguese† 8 Finnish (Finland)† 9 UK† 10 Norwegian (Norway)† 11 Danish (Denmark)† 12 Swedish† *†Not yet implemented*
nmivec:	0318	0318	2	Kernal Def	No	NMI vector when ROMs are switched in.
NMIVector:	FFFA	FFFA	2	$FB24	No	Non maskable interrupt vector.
numDrives:	848D	848D	1	Actual	No	Number of drives in the system.

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
obj0Pointer:	8FF8	8FF8	1	$28	Yes	Pointer to picture data for mouse cursor. Pointer to picture data for text entry cursor. Pointers to picture data for Sprites 2-7. **Note**: Sprite pointers are always in the last 8 bytes of the video matrix area. Pointers are an index into 64 byte pages starting the base of the current VIC bank. **Note**: GEOS uses VIC bank 2 at address $8000. To get the address of the sprite image, multiply the object pointer by 64 and add it to the VIC bank starting address. $28*64 + $8000 = **spr0pic** ($8A40)
obj1Pointer:	8FF9	8FF9	1	$29		
obj2Pointer:	8FFA	8FFA	1	$2A		
obj3Pointer:	8FFB	8FFB	1	$2B		
obj4Pointer:	8FFC	8FFC	1	$2C		
obj5Pointer:	8FFD	8FFD	1	$2D		
obj6Pointer:	8FFE	8FFE	1	$2E		
obj7Pointer:	8FFF	8FFF	1	$2F		
otherPressVec:	84A9	84A9	2	NULL	Yes	Vector to routine that is called when the mouse button is pressed and it is not on either a menu or an icon.
potX:	D419	D419	1	None	No	Mouse position: bits 1-6 = current x-position.
potY:	D41A	D41A	1	None	No	Mouse position: bits 1-6 = current y-position.
pressFlag:	39	39	1	0	No	Flag to indicate that a new key has been pressed. bit Constant Description ---- --------------------- ---------------- b7: KEYPRESS_BIT keyboard data is new b6: INPUT_BIT input device direction change b5: MOUSE_BIT mouse button data is new
PrntDiskName:	8476	8476	16+2/ 17+1	None	No	Disk name that current printer driver is on. (Null terminated). When disk name is 16 bytes, the null terminator overwrites the first byte of the Disk ID.
PrntFilename:	8465	8465	17	None	No	Name of the current printer driver. (Null terminated)
ramBase:	88C7	88C7	4	None	No	Starting RAM bank for each disk drive to use if the drive type is either a RAM Disk or Shadowed Drive.
ramExpSize:	88C3	88C3	1	Actual	No	Number of 64K RAM banks available in RAM expansion unit.
random:	850A	850A	2	None	No	Calculated each interrupt to generate a random number. **random** = (2*(**random**+1) // 65521) **Note**: // is the modulus operator.
rasreg:	D012	D012	1	None	No	Raster register.
RecoverVector:	84B1	84B1	2	See Desc	Yes	Pointer to routine that is called to recover the background behind menus and dialog boxes. Normally this routine is **RecoverRectangle**, but the user can supply his own routine.
resetVector:	FFFC	FFFC	2	$FB24	No	Reset vector.
returnAddress:	3D	3D	2	None	No	Address to return to from in-line call.
rightMargin:	37	37	2	40=319 80=639	Yes	The rightmost point for writing characters. If an attempt is made to write past **rightMargin**, the routine pointed to by **StringFaultVec** is called.

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
r0:	02	02	2	None	No	Kernal pseudoregisters: Pseudoregisters are used when calling into the GEOS Kernal. Each call will have a list of registers to setup. Registers have common uses across the GEOS API but none are exclusively for only one thing. The pseudoregister **r15** is never used by the kernal **and** pseudoregisters **r12-r14** are very rarely used. These pseudoregisters make for very safe temporary **zpage** use. Always consider using the available options in **r0-r15** that do not conflict with your current Kernal interaction as temporary storage.
r1:	04	04	2			
r2:	06	06	2			
r3:	08	08	2			
r4:	0A	0A	2			
r5:	0C	0C	2			
r6:	0E	0E	2			
r7:	10	10	2			
r8:	12	12	2			
r9:	14	14	2			
r10:	16	16	2			
r11:	18	18	2			
r12:	1A	1A	2			
r13:	1C	1C	2			
r14:	1E	1E	2			
r15:	20	20	2			
savedmoby2:	88BB	88BB	1	None	No	Saved value of **moby2** for context saving done when dialog boxes and desk accessories run. (**moby2** was left out of the original GEOS save code, so it was put here. It remains to be compatible with older desk accessories, etc. that use it)
saveFontTab:	850C	850C	9	None	No	Buffer for saving the user active font table when going into menus.
scr80colors:	–	88BD	1	$E0	No	Screen colors for 80-column mode on the C128. Copy of reg 26 in the VDC.
scr80polar:	–	88BC	1	$40	No	Copy of reg 24 in the VDC for the C128. Controls reversing foreground and background colors. This is used for making the border color match either the background or the foreground as needed.
screencolors:	851E	851E	1	$BF	No	Default 40 Col screen colors.
seconds:	851B	851B	1	0	No	Seconds variable for the time of day clock.
selectionFlash:	84B3	84B3	1	10	Yes	Speed at which menu items and icons are flashed. Value is number of vblanks.
shiftBuf:	–	1B45[†]	7	None	No	Buffer for shifting/doubling sprites. Located in backRAM.
shiftOutBuf:	–	1B4C[†]	7	None	No	Buffer for shifting/doubling/oring sprites. Located in backRAM.
sidAtDcy:	D405	D405	1	None	No	Attack / Decay. b7-4: attack phase duration. (0-15) (.002 to 8 seconds) b3-0: decay phase duration. (0-15) (.006 to 24 seconds)
sidbase:	D400	D400	–	–	–	Sound interface device base address.
sidEnv3:	D41C	D41C	1	None	No	Voice 3 Envelope generator output. (R/O)

Name	Address (hex)					
	64	128	Size	Default	Saved	Description
sidFcHi: **sidFcLo**:	D416 D415	D416 D415	1 1	None	No	Filter cutoff frequency (11 bit word). (W/O) Word is b7-0 of high byte concatenated with b2-0 of low byte.
sidFreHi: **sidFreLo**:	D401 D400	D401 D400	1 1	None	No	Frequency of output tone. (word) frequency = Hz / 0.06095 for NTSC systems frequency = Hz / 0.05873 for PAL systems tunning A_4 = 440^{Hz} / 0.06095 = 7217 = NTSC frequency of $1C31. (word)
sidOsc3:	D41B	D41B	1	None	No	Voice 3 oscillator output. (R/O) **Note**: The output from this register can be used for generating random numbers.
sidPWHi: **sidPWLo**:	D403 D402	D403 D402	1 1	None	No	Pulse width. Word sized pulse width of pulse wave. (b15-12 not used) b11-b0 (0-4095) represents a duty cycle percentage from 0 to 100. a 50% duty cycle of 2048 ($800) produces the richest pulse output.
sidResFilt:	D417	D417	1	None	No	Filter selection and resonance control. b7-4: resonance. (0-15) (none -> full) b3: 1=external output is filtered. b2-0: 1=voice is filtered. (b0=voice 1;b1=voice 2;b2=voice 3)
sidSigVol:	D418	D418	1	None	No	Volume and filter mode control. b7: 1=voice 3 cut off from combined output b6-4: low/high/band/pass filters b3-0: percentage of max volume. (0-15) (0-100%)
sidSuRel:	D406	D406	1	None	No	Sustain / Release. b7-b4: sustain % of peak output (0-15) (0 to 100%). A sustain value of 0 makes the sound end after decay instead of after release. b3-b0: release phase time. (0-15) (.006 to 24 seconds)
sidVCReg:	D404	D404	1	None	No	Voice control. **Note**: Only one wave type can be active at a time (b7-4). b7: 1=noise b6: 1=pulse wave b5: 1=sawtooth wave b4: 1=triangle wave b3: test flag: 1 stops voice output; 0 allows voice output. b2: ring modulation; 1 = active. b1: sync flag; 1 = synchronization active. b0: gate flag: 1 starts attack; 0 starts release (ends the sound)
sidVoc1:	D400	D400	7	None	No	Voice 1 registers base address. Access registers directly with **sidFreLo – sidSuRel** For lookup tables use offsets `[ O_FRELO .. O_SUREL ]`

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
sidVoc2:	D407	D407	7	None	No	Voice 2 registers base address. Access directly with **sidbase**+O_VOICE2+ [O_FRELO..O_SUREL] - or [**sidFreLo..sidSuRel**] + O_VOICE2 For lookup tables use offsets O_VOICE2 + [O_FRELO..O_SUREL]
sidVoc3:	D40E	D40E	7	None	No	Voice 3 registers base address. Use same addressing access as **sidVoc2** but use O_VOICE3 instead. Example: **LoadW sidbase**+O_VOICE3+O_FREQUENCY,#note440
sizeFlags:	–	1B53†	7	None	No	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.
softOnes:	–	1C2D†	192	None	No	Buffer used for putting sprite bitmaps up on screen without disturbing background. Resides in backRAM.
softZeros:	–	1B6D†	192	None	No	Buffer used for putting sprite bitmaps up on screen without disturbing background. Resides in backRAM.
spr0pic:	8A00	8A00	64	Mouse Pic	No	Graphics data for sprite 0. This sprite holds the mouse pointer image.
spr1pic:	8A40	8A40	64	None	No	Used for text prompt. Populated by **InitTextPrompt**. **Note**: If application is not using text prompts then this can be a data area for the application.
spr2pic:	8A80	8A80	64	None	No	Used for application sprite images. **Note**: If the application is not using these sprites then this can be a data area for the application. **Example**: Use all of sprite 1 through 7 area as a ramsect buffer. .ramsect $8A40 highBuf: .block 448.
spr3pic:	8AC0	8AC0	64			
spr4pic:	8B00	8B00	64			
spr5pic:	8B40	8B40	64			
spr6pic:	8B80	8B80	64			
spr7pic:	8BC0	8BC0	64			
sspr1back:	–	133B†	294	None	No	Soft sprite backRAM buffers 1-7. Used to save the screen behind the sprites. Each buffer is 7 bytes wide by 42 scanlines high (292 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. **Note**: If the application is not actively using the sprites this can be an application data area.
sspr2back:	–	1461†	294			
sspr3back:	–	1587†	294			
sspr4back:	–	16AD†	294			
sspr5back:	–	17D3†	294			
sspr6back:	–	18F9†	294			
sspr7back:	–	1A1F†	294			
STATUS:	90	90	1	None	No	Kernal I/O status.
string:	24	24	2	None	Yes	Used by GEOS as a pointer to string destinations for routines such as **GetString**.
StringFaultVec:	84AB	84AB	2	NULL	Yes	Vector called when an attempt is made to write a character past **leftMargin** or **rightMargin**.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
stringX:	84BE	84BE	2	None	Yes	The x-position for string input.
stringY:	84C0	84C0	1	None	Yes	The y-position for string input.
sysDBData:	851D	851D	1	None	n/a	Variable that is used to indicate which icon caused a return to the application (from a dialog box). The actual data is returned to the user in **r0L**. Applications can set a value in this field as a result of a dialog action. The value will then be returned in **r0L** when the dialog closes.
sysFlgCopy:	C012	C012	1	None	No	Copy of the **sysRAMFlg** that is saved here when going into Commodore BASIC. See **sysRAMFlg** for more information.
sysRAMFlg:	88C4	88C4	1	None	No	If RAM expansion is installed, bank 0 is reserved for the Kernal's use. The **sysRAMFlg** byte contains flags designating its usage: b7: if 1 $0000-$78FF is used by C64 **MoveData** routine $0000-$38FF is used by C128 **MoveData** routine b6: if 1 $8300-$B8FF holds disk drivers for drives A through D b5: if 1 $7900-$7DFF is loaded with GEOS RAM area $8400-88FF by **ToBasic** routine when going to BASIC. $7E00-$82FF is loaded with reboot code by CONFIGURE. The reboot code 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 $B900-$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). b4: if 1 C128 only: $D500-DC3F holds the active print driver. See **GetFile** notes for more information. Also see **REU-BANK0**
systemVector:	–	FFF8	2	**Reset**	No	Soft reset vector. Called when reset button is pressed.

turboFlags:	8492	8492	4	None	No	Turbo state flags for drives 8 through 11.
						Flag byte layout:
						b7: set when turbo is loaded
						b6: set when turbo is active
						b5-0: not used, always zero
						diskOpenFlg can be used as a base to index into this table by drive number.
						Example:
						`ldy curDrive`
						`lda diskOpenFlg,y`

usedRecords:	8497	8497	1	0	No	Holds the number of records in an open VLIR file.

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
vdccr:	–	`D600`	1	None	No	VDC Control Register (R/W) Write: b5-b0: register number for data register to use. Read: b7: STATUS 1=data register ready b6: LP 1=light pen latched b5: VBLANK 1=vblank active b4-b3: unused/always 0 b2-b0: H/W version
vdcdr:	–	`D601`	1	None	No	VDC Data Register (R/W). Registers: **constants\Hardware\VDC**
vdcClrMode:	–	`88BE`	1	0	No	Holds the current color mode for C128 color routines. 0 = monochrome 1 = 176x640 8x8 color. (Do not attempt to draw below Col 175) 2 = 200x640 8x8 color. 64K VDC RAM required. 3 = 200x640 8x4 color. 64K required. 4 = 200x640 8x2 color. 64K required.
version:	`C00F`	`C00F`	1	$20	No	Holds byte indicating what version of GEOS is running. Version number is stored in high and low nibbles of version byte. Examples of known versions: $11 = Version 1.1 $12 = Version 1.2 $13 = Version 1.3 $20 = Version 2.0 $44 = Wheels version 4.4
windowBottom:	`34`	`34`	1	199	Yes	Bottom line of window for text clipping.
windowTop:	`33`	`33`	1	0	Yes	Top line of window for text clipping.
year:	`8516`	`8516`	1	86	No	Holds the **year** for the time of day clock.

†Located in backRAM

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
zpage:						Zero page base address.
CPU_DDR	00	00	1	%101111	No	**6510 data direction register**.
CPU_DATA	01	01	1	RAM_64K	No	**6510 data register**. Controls the hardware memory map of the C64.
r0	02	02	2	None	No	Kernal pseudoregisters: Pseudoregisters are used when calling into the GEOS Kernal. Each call will have a list of registers to setup. Registers have common uses across the GEOS API but none are exclusively for only one thing. The pseudoregister **r15** is never used by the kernal and pseudoregisters **r12-r14** are very rarely used. These pseudoregisters make for very safe temporary **zpage** use. Always consider using the available options in **r0-r15** that do not conflict with your current Kernal interaction as temporary storage.
r1	04	04	2			
r2	06	06	2			
r3	08	08	2			
r4	0A	0A	2			
r5	0C	0C	2			
r6	0E	0E	2			
r7	10	10	2			
r8	12	12	2			
r9	14	14	2			
r10	16	16	2			
r11	18	18	2			
r12	1A	1A	2			
r13	1C	1C	2			
r14	1E	1E	2			
r15	20	20	2			
curPattern:	22	22	2	$D010	Yes	Pointer to the first byte of the graphics data for the current pattern in use.
string:	24	24	2	None	Yes	Pointer to string destinations for routines such as **GetString**.
fontTable:			(9)			Beginning of font variables.
baselineOffset:	26	26	1	6	Yes	Offset from top line to baseline in character set.
curSetWidth:	27	27	2	$3C	Yes	Card width in pixels for the current font.
curHeight:	29	29	1	9	Yes	Card height in pixels of the current font in use.
curIndexTable:	2A	2A	2	$D218	Yes	Pointer to the table of sizes, in bytes, of each card in of the current font.
cardDataPntr:	2C	2C	2	$D2DC	Yes	This is a pointer to the actual card graphic data for the current font in use.
currentMode:	2E	2E	1	0	Yes	Current text drawing mode.
dispBufferOn:	2F	2F	1	$C0	Yes	Routes graphic and text operations between foreground and background buffers.
mouseOn:	30	30	1	$E0	Yes	Flag indicating that the mouse/menu/icon is on.
msePicPtr:	31	31	2	$84C1	Yes	Pointer to the mouse graphics data. default = **mousePicData**
windowTop:	33	33	1	0	Yes	Top line of window for text clipping.
windowBottom:	34	34	1	199	Yes	Bottom line of window for text clipping.
leftMargin:	35	35	2	0	Yes	Leftmost point for writing characters.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
rightMargin:	37	37	2	319/639	Yes	The rightmost point for writing characters.
pressFlag:	39	39	1	0	No	Flag to indicate that a new input action has occurred.
mouseXPos:	3A	3A	2	None	No	Mouse x-position.
mouseYPos:	3C	3C	1	0	No	Mouse y-position.
returnAddress:	3D	3D	2	None	No	Address to return to from in-line call.
graphMode:	–	3F	1	None	No	Current video mode for C128. GR_40=$00 / GR_80=$80
APP_ZPL:	70	70	16	None	No	Application private zero page. **a2-a9** when using default naming.
STATUS:	90	90	1	None	No	Kernal I/O status.
curDevice:	BA	BA	1	8	No	Current serial device number.
APP_ZPH:	FB	FB	4	None	No	Application private zero page. **a0-a1** when using default naming.
APP_LVAR:	0200	0200	276	None	No	Application variable space. **Note**: **AppLowVar**
irqvec:	0314	0314	2	Kernal Def	No	IRQ vector when ROMs are switched in.
bkvec:	0316	0316	2	Kernal Def	No	BRK instruction vector when ROMs are switched in.
nmivec:	0318	0318	2	Kernal Def	No	NMI vector when ROMs are switched in.
kernalVectors:	031A	031A	26	Kernal Def	No	Location of kernal vectors when ROMS are switched in.
APP_LRAM:	0334	0334	204	None	No	Optional application space. **Note**: Conflicts with geoDebugger.
APP_RAM:	0400	0400	23K	None	No	Start of application space.
BACK_SCR_BASE:	6000	6000	8000	None	No	Base of background screen. (Lines 0-99 of VDC background screen).
PRINTBASE:	7900	7900	1600	None	No	Load address for print drivers.
APP_VAR:	7F40	7F40	192	None	No	Application variable space.
diskBlkBuf:	8000	8000	256	0	No	General disk block buffer. Initialized to all zeros.
fileHeader:	8100	8100	256	0	No	Header block buffer for a GEOS file.
curDirHead:	8200	8200	256	0	No	Buffer containing header information for the disk in currently selected drive. Structure: **Directory Header**
fileTrScTab:	8300	8300	256	0	No	Track and sector chain for a file of maximum size of 32258 bytes.
dirEntryBuf:	8400	8400	30	0	No	Buffer used to build a file's directory entry. Structure: **Directory Entry**
DrACurDkNm:	841E 842E	841E 842F	16 2	None	No	Disk name of the current disk in drive A, 16 characters padded with $A0. 2-character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
DrBCurDkNm:	8430 8440	8430 8440	16 2	None	No	Disk name of the current disk in drive B, 16 characters padded with $A0. 2-character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
dataFileName:	8442	8442	17	None	No	Name of a data file to open by parent application. (Null terminated)
dataDiskName:	8453	8453	16+2	None	No	Holds the disk name that an application's data file is on.
PrntFilename:	8465	8465	17	None	No	Name of the current printer driver. (Null terminated).

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
PrntDiskName:	8476	8476	16+2/17+1	None	No	Disk name that current printer driver is on. (Null terminated). When disk name is 16 bytes, the null terminator overwrites the first byte of the Disk ID.
driveNdxType:	(8486)	(8486)	–	–	–	Label used for indexing **driveType** with drive number. `lda driveNdxType,x`
–	8488	8488	1	–	–	Not used by GEOS. Free to use by applications.
curDrive:	8489	8489	1	8	No	Device number of the currently active disk drive.
diskOpenFlg:	848A	848A	1	0	No	Not used by the GEOS Kernal. Label can be used as index base into **turboFlags**
isGEOS:	848B	848B	1	Disk	No	Flag to indicate whether the current disk is a GEOS disk.
interleave:	848C	848C	1	8	No	Used by **BlkAlloc** routine as the desired **interleave**.
numDrives:	848D	848D	1	Actual #	No	Number of drives in the system.
driveType:	848E	848E	4	–	No	Drive type of each of the four possible drives.
turboFlags:	8492	8492	4	None	No	Turbo state flags for drives 8 through 11
curRecord:	8496	8496	1	0	No	Current record number for an open VLIR file.
usedRecords:	8497	8497	1	0	No	Holds the number of records in an open VLIR file
fileWritten:	8498	8498	1	None	No	Flag indicating currently open VLIR file has been changed.
fileSize:	8499	8499	2	None	No	Current size (in blocks) of a file.
appMain:	849B	849B	2	NULL	Yes	Main loop service routine vector
intTopVector:	849D	849D	2	NULL	Yes	Interrupt top service routine vector.
intBotVector:	849F	849F	2	NULL	Yes	Interrupt bottom service routine vector.
mouseVector:	84A1	84A1	2	System	Yes	Mouse button press service routine vector. See also: **otherPressVec**
keyVector:	84A3	84A3	2	NULL	Yes	Vector to routine to call on keypress.
inputVector:	84A5	84A5	2	NULL	Yes	Input device direction change.
mouseFaultVec:	84A7	84A7	2	System	Yes	Mouse fault service routine vector.
otherPressVec:	84A9	84A9	2	NULL	Yes	Mouse button press service routine vector. (not on either a menu or an icon)
StringFaultVec:	84AB	84AB	2	NULL	Yes	String margin fault service routine vector.
alarmTmtVector:	84AD	84AD	2	NULL	Yes	Alarm clock time-out service routine vector.
BRKVector:	84AF	84AF	2	System	Yes	BRK instruction service routine vector. Defaults to calling **Panic**.
RecoverVector:	84B1	84B1	2	System	Yes	Background recover service routine vector.
selectionFlash:	84B3	84B3	1	10	Yes	Speed at which menu items and icons are flashed. Value is number of vblanks.
alphaFlag:	84B4	84B4	1	0	Yes	Flag for alphanumeric string input.
iconSelFlag:	84B5	84B5	1	0	Yes	Icon selection flags.
faultData:	84B6	84B6	1	0	Yes	Holds information about mouse faults.
menuNumber:	84B7	84B7	1	0	Yes	Number of currently working menu
mouseTop:	84B8	84B8	1	0	Yes	Top most position for mouse.
mouseBottom:	84B9	84B9	1	199	Yes	Bottom most position for mouse cursor. Normally set to bottom of the screen.

Name	Address (hex)		Size	Default	Saved	Description
	64	128				
mouseLeft:	84BA	84BA	2	0	Yes	Left most position for mouse.
mouseRight:	84BC	84BC	2	319/639	Yes	Right most position for mouse.
stringX:	84BE	84BE	2	None	Yes	The x-position for string input.
stringY:	84C0	84C0	1	None	Yes	The y-position for string input.
mousePicData:	84C1	84C1	64	Mouse Pic	No	64-byte array for the mouse sprite picture.
maxMouseSpeed:	8501	8501	1	$7F	No	Maximum speed for mouse cursor.
minMouseSpeed:	8502	8502	1	$1E	No	Minimum speed for mouse cursor.
mouseAccel:	8503	8503	1	$75	No	Acceleration of mouse cursor.
keyData:	8504	8504	1	0	No	Holds the ASCII value of the last key that was pressed.
mouseData:	8505	8505	1	None	No	State of mouse button: high bit set if button is released; clear if pressed.
inputData:	8506	8506	4	None	No	Input driver device specific information.
random:	850A	850A	2	None	No	Calculated each interrupt to generate a random number.
saveFontTab:	850C	850C	9	None	No	Buffer for saving the user active font table when going into menus.
dblClickCount:	8515	8515	1	0	No	Used to determine when an icon is double clicked on.
year:	8516	8516	1	86	No	Holds the **year** for the time of day clock.
month:	8517	8517	1	9	No	Holds the **month** for time of day clock.
day:	8518	8518	1	20	No	Current **day**.
hour:	8519	8519	1	12	No	Current **hour**.
minutes:	851A	851A	1	0	No	Holds the **minutes** for time of day clock.
seconds:	851B	851B	1	0	No	Current **seconds**.
alarmSetFlag:	851C	851C	1	FALSE	No	TRUE if the alarm is set for GEOS to monitor, else FALSE.
sysDBData:	851D	851D	1	None	No	Icon number that caused a return to the application (from a dialog box).
screencolors:	851E	851E	1	$BF	No	Default 40 column screen colors.
dlgBoxRamBuf:	851F	851F	417	None	n/a	Dialog Box Ram buffer. Used when desk accessories or dialog boxes are run.
savedmoby2:	88BB	88BB	1	None	No	Used by old desk accessories/applications. Prior to version 1.3 of GEOS.
scr80polar:	–	88BC	1	$40	No	Copy of reg 24 in the VDC for the C128.
scr80colors:	–	88BD	1	$E0	No	Screen colors for 80-column mode on the C128. It is a copy of reg 26 in the VDC.
vdcClrMode:	–	88BE	1	0	No	Holds the current color mode for C128 color routines.
driveData:	88BF	88BF	4	None	No	One byte is reserved for each disk drive, to be used by the disk driver.
ramExpSize:	88C3	88C3	1	Actual	No	Number of 64K RAM banks available in RAM expansion unit.
sysRAMFlg:	88C4	88C4	1	None	No	REU Bank 0 control flags.
firstBoot:	88C5	88C5	1	$00/$FF	No	FALSE ($00) While system is booting. TRUE ($FF) after boot is complete.
curType:	88C6	88C6	1	Drv 8 type	No	Holds the current drive type.
ramBase:	88C7	88C7	4	None	No	Starting RAM bank for each disk drive that uses RAM banks.

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
inputDevName:	88CB	88CB	17	None	No	Name of the current input device.
DrCCurDkNm:	88DC	88DC	16	None	No	Disk name of the current disk in drive C, 16 characters padded with $A0.
	88EC	88EC	2			2-character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
DrDCurDkNm:	88EE	88EE	16	None	No	Disk name of the current disk in drive D, 16 characters padded with $A0.
	88FE	88FE	2			2-character DiskID. (ID is always $A0A0 from a bug in all versions of GEOS)
dir2Head:	8900	8900	256	None	No	Second BAM block. Used by 1571 and 1581 drives.
spr0pic:	8A00	8A00	64	Mouse Pic	No	Graphics data for sprite 0. This sprite holds the mouse pointer image.
spr1pic:	8A40	8A40	64	None	No	Used for text prompt.
spr2pic:	8A80	8A80	64	None	No	Used for application sprite images. If the application is not using these sprites then this can be a data area for the application. **Example**: Use all of sprite 1 through 7 area as a ramsect buffer. .ramsect $8A40 highBuf: .block 448.
spr3pic:	8AC0	8AC0	64			
spr4pic:	8B00	8B00	64			
spr5pic:	8B40	8B40	64			
spr6pic:	8B80	8B80	64			
spr7pic:	8BC0	8BC0	64			
COLOR_MATRIX:	8C00	8C00	1000	None	No	Foreground and background color cards for 40 column mode.
obj0Pointer:	8FF8	8FF8	1	$28	Yes	Pointer to the picture data for mouse cursor. ($8000+$28*64=**spr0pic**)
obj1Pointer:	8FF9	8FF9	1	$29		Pointer to picture data for text entry cursor. ($8000+$29*64=**spr1pic**)
obj(2-7)Pointer:	8FFA	8FFA	1x6	$2A-$2F		Pointers to the picture data for sprites 2-7.
DISK_BASE:	9000	9000	4096	None	No	Disk driver for currently active drive.
dir3Head:	9C80	9C80	256	None	No	Third BAM block. Used by 1581 drive.
JmpIndX:		9D80	6	–	–	C128: 2 entry jump table
SCREEN_BASE:	A000	A000	8000	None	No	Base of foreground screen. C64 & C128 in 40 column mode.
	–	A040	8000	None	No	Lower half of VDC back screen buffer. (Lines 100-199)
	BF40	BF80				Start of C64 GEOS Kernal / Start of C128 GEOS Kernal.
bootName:	C006	C006	9	->	No	Start of the "GEOS BOOT" string. "GATEWAY" on Gateway version.
gatewayFlag:	C007	C007	1	None	No	On the gateway version of GEOS there will be a 'A' at this location.
version:	C00F	C00F	1	$20	No	Holds byte indicating what version of GEOS is running.
nationality:	C010	C010	1	0 (USA)	No	Nationality of Kernal.
sysFlgCopy:	C012	C012	1	None	No	Copy of the **sysRAMFlg**.
c128Flag:	C013	C013	1	None	No	Defines current machine type. $80=C128 / $00 = C64.
dateCopy:	C018	C018	3	YMD	No	Copy of system variables **year**, **month**, and **day**.

Name	Address (hex) 64	 128	Size	Default	Saved	Description
vicbase:						Video Interface Chip base address.
mob0xpos:	D000	D000	1	None	Yes	Sprite 0 x-position.
mob0ypos:	D001	D001	1	None	Yes	Sprite 0 y-position.
mob1xpos:	D002	D002	1	None	Yes	Sprite 1 x-position.
mob1ypos:	D003	D003	1	None	Yes	Sprite 1 y-position.
mob2xpos:	D004	D004	1	None	Yes	Sprite 2 x-position.
mob2ypos:	D005	D005	1	None	Yes	Sprite 2 y-position.
mob3xpos:	D006	D006	1	None	Yes	Sprite 3 x-position.
mob3ypos:	D007	D007	1	None	Yes	Sprite 3 y-position.
mob4xpos:	D008	D008	1	None	Yes	Sprite 4 x-position.
mob4ypos:	D009	D009	1	None	Yes	Sprite 4 y-position.
mob5xpos:	D00A	D00A	1	None	Yes	Sprite 5 x-position.
mob5ypos:	D00B	D00B	1	None	Yes	Sprite 5 y-position.
mob6xpos:	D00C	D00C	1	None	Yes	Sprite 6 x-position.
mob6ypos:	D00D	D00D	1	None	Yes	Sprite 6 y-position.
mob7xpos:	D00E	D00E	1	None	Yes	Sprite 7 x-position.
mob7ypos:	D00F	D00F	1	None	Yes	Sprite 7 y-position.
msbxpos:	D010	D010	1	None	Yes	Bit 8 of sprite #0-7 x-coordinates.
grcntrl1:	D011	D011	1	$3B	No	Graphics control register #1.
rasreg:	D012	D012	1	None	No	Raster register.
lpxpos:	D013	D013	1	None	No	Light pen x-position.
lpypos:	D014	D014	1	None	No	Light pen y-position.
mobenble:	D015	D015	1	None	Yes	Sprite enable bits.
grcntrl2:	D016	D016	1	ST_40COL	No	Graphics control resister #2.
moby2:	D017	D017	1	0	Yes	Double object size in y.
grmemptr:	D018	D018	1	%0011100x	No	Graphics memory pointer VM13-VM10|CB13-CB11.
grirq:	D019	D019	1	–	No	Graphics chip interrupt register.
grirqen:	D01A	D01A	1	%01	No	Graphics chip interrupt enable register.
mobprior:	D01B	D01B	1	0	Yes	Object to background priority.
mobmcm:	D01C	D01C	1	0	Yes	Sprite multi-color mode select.
mobx2:	D01D	D01D	1	0	Yes	Double object size in x.
mobmobcol:	D01E	D01E	1	0	No	Object to object collision register.
mobbakcol:	D01F	D01F	1	None	No	Sprite to background collision register.
extclr:	D020	D020	1	BLACK	No	Exterior (border) color.

	Address (hex)					
Name	**64**	**128**	**Size**	**Default**	**Saved**	**Description**
bakclr0:	D021	D021	1	DKGREY	No	Hardware registers to control background colors 0-3.
bakclr1:	D022	D022	1	WHITE		b7-4: = not used. always 1's
bakclr2:	D023	D023	1	RED		b0-3: = color for text background.
bakclr3:	D024	D024	1	CYAN		bakclr(1-3) only used in multi color mode. (Not used by GEOS).
mcmclr0:	D025	D025	1	None	Yes	Multi-color mode color 0.
mcmclr1:	D026	D026	1	None	Yes	Multi-color mode color 1.
mob0clr:	D027	D027	1	None	No	Color of sprites 0 – 7.
mob1clr:	D028	D028	1			
mob2clr:	D029	D029	1			
mob3clr:	D02A	D02A	1			
mob4clr:	D02B	D02B	1			
mob5clr:	D02C	D02C	1			
mob6clr:	D02D	D02D	1			
mob7clr:	D02E	D02E	1			
keyreg:	–	D02F	1	None	No	C128 keyboard register for # pad & other keys: b2-b0 = scan rows 8,9 and 10.
clkreg:	–	D030	1	None	No	C128 clock speed register: b0 0=1MHz; 1=2MHz

sidbase:						Sound interface device base address.	**Alternate addressing methods**
sidVoc1:	D400	D400	7	None	No	Voice 1 registers:	
sidFreLo:	D400	D400	1	None	No	Frequency. Word sized value	**sidVoc1**+O_FREQUENCY
sidFreHi:	D401	D401	1			high byte	**sidVoc1**+O_FREHI
sidPWLo:	D402	D402	1	None	No	Pulse width. Word sized value	**sidVoc1**+O_PULSEWIDTH
sidPWHi:	D403	D403	1			high byte	**sidVoc1**+O_PWHI
sidVCReg:	D404	D404	1	None	No	Voice control.	**sidVoc1**+O_VCREG
sidAtDcy:	D405	D405	1	None	No	Attack / Decay.	**sidVoc1**+O_ATDCY
sidSuRel:	D406	D406	1	None	No	Sustain / Release.	**sidVoc1**+O_ATDCY
sidVoc2:	D407	D407	7	None	No	Voice 2 registers:	**sidVoc2**+O_FREQUENCY... etc.
sidVoc3:	D40E	D40E	7	None	No	Voice 3 registers:	**sidVoc3**+O_FREQUENCY... etc.
sidFcLo:	D415	D415	1	None	No	Filter cutoff frequency. Word sized value	**sidbase**+O_FREQCUTOFF
sidFcHi:	D416	D416	1			high byte	**sidbase**+O_FCHI
sidResFilt:	D417	D417	1	None	No	Filter selection and resonance control.	**sidbase**+O_RESFILT
sidSigVol:	D418	D418	1	None	No	Volume and filter mode control.	**sidbase**+O_SIGVOL
potX:	D419	D419	1	None	No	Mouse position: bits 1-6 = current x-position.	
potY:	D41A	D41A	1	None	No	Mouse position: bits 1-6 = current y-position.	

Name	Address (hex) 64	128	Size	Default	Saved	Description
sidOsc3:	D41B	D41B	1	None	No	Voice 3 oscillator output. (Read Only) **sidbase**+O_OSC3
sidEnv3:	D41C	D41C	1	None	No	Voice 3 envelope generator output. (R/O) **sidbase**+O_ENV3

Name	64	128	Size	Default	Saved	Description
mmucr:	–	D500	1	CIO_IN	No	C128 MMU **Configuration register**. (Mirrored by **config** at FF00)
mmupcra/b/c/d:	–	D501	1x4	None	No	C128 MMU Pre configuration registers a,b,c and d.
mmumcr:	–	D505	1	$37	No	C128 MMU Mode configuration register. b7 = 40/80 key switch. 0=80 columns
mmurcr:	–	D506	1	$40	No	C128 MMU **RAM Configuration register**.
mmup0L:	–	D507	1	0	No	C128 MMU Zero Page page pointer.
mmup0H:	–	D508	1	%xxxxxxx0	No	C128 MMU Zero Page bank pointer.
mmup1L:	–	D509	1	1	No	C128 MMU Stack Page page pointer.
mmup1H:	–	D50A	1	%xxxxxxx0	No	C128 MMU Stack Page bank pointer.

Name	64	128	Size	Default	Saved	Description
vdccr:	–	D600	1	None	No	VDC Control Register (R/W).
vdcdr:	–	D601	1	None	No	VDC data register (R/W). Registers: **constants\Hardware\VDC**

Name	64	128	Size	Default	Saved	Description
ctab:	D800	D800	1000	None	No	Character colors when in text mode. (GEODEBUGGER, GEOBASIC).

cia1base: Complex Interface Adapter 1

Name	64	128	Size	Default	Saved	Description
cia1pra:	DC00	DC00	1	None	No	Peripheral data register a. Keyboard/Joystick/Mouse inputs.
cia1prb:	DC01	DC01	1	None	No	Peripheral data register b. Keyboard/Joystick/Mouse inputs.
cia1ddra:	DC02	DC02	1	None	No	Data direction reg a. 0=read only, 1=write only.
cia1ddrb:	DC03	DC03	1	None	No	Data direction reg a. 0=read only, 1=write only.
cia1talo:	DC04	DC04	1	None	No	Timer A. Programmable counter (word)
cia1tahi:	DC05	DC05	1			high byte of counter.
cia1tblo:	DC06	DC06	1	None	No	Timer B. Programmable counter (word)
cia1tbhi:	DC07	DC07	1			high byte of counter.
cia1tod10ths:	DC08	DC08	1	None	No	10ths of second register. read/write (GEOS time) **Important**: Writting or reading **cia1tod10ths** starts the time of day timer.
cia1todsec:	DC09	DC09	1	None	No	Seconds register. (R/W) BCD (GEOS time)
cia1todmin:	DC0A	DC0A	1	None	No	Minutes register. (R/W) BCD (GEOS time)
cia1todhr:	DC0B	DC0B	1	None	No	Hours – AM; PM register. (R/W) BCD (GEOS time) **Important**: Writing or reading **cia1todhr** stops the time of day timer.
cia1sdr:	DC0C	DC0C	1	None	No	Serial data register.
cia1icr:	DC0D	DC0D	1	None	No	Interrupt control register.

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
cia1cra:	DC0E	DC0E	1	None	No	Timer control register a.
cia1crb:	DC0F	DC0F	1	None	No	Timer control register b.
cia2base:						Complex Interface Adapter 2
cia2pra:	DD00	DD00	1	None	No	Peripheral data register a.
cia2prb:	DD01	DD01	1	None	No	Peripheral data register b. Used by RS232 serial routines.
cia2ddra:	DD02	DD02	1	None	No	Data direction register a. 0=read only, 1=write only
cia2ddrb:	DD03	DD03	1	None	No	Data direction register b. 0=read only, 1=write only
cia2talo:	DD04	DD04	1	None	No	Timer a. Word value
cia2tahi:	DD05	DD05	1			high byte
cia2tblo:	DD06	DD06	1	None	No	Timer b. Word value
cia2tbhi:	DD07	DD07	1			high byte
cia2tod10ths:	DD08	DD08	1	None	No	10ths of sec register. read/write. [b3-b0 range (0-9)]
cia2todsec:	DD09	DD09	1	None	No	Seconds register. (R/W) BCD [b7-b4 Tenths (0-5);b3-b0 Ones (0-9)]
cia2todmin:	DD0A	DD0A	1	None	No	Minutes register. (R/W) BCD
cia2todhr:	DD0B	DD0B	1	None	No	Hours – AM; PM reg. (R/W) BCD
cia2sdr:	DD0C	DD0C	1	None	No	Serial data register.
cia2icr:	DD0D	DD0D	1	None	No	Interrupt control register.
cia2cra:	DD0E	DD0E	1	None	No	Timer control register a.
cia2crb:	DD0F	DD0F	1	None	No	Timer control register b.
georampg:	DE00	DE00	$100	N/A	No	GEORAM: memory page.
georamps:	DFFE	DFFE	1			b5-0: page select register.
georambs:	DFFF	DFFF	1			b4-0: block select register.
EXP_BASE:	DF00	DF00	11	N/A	No	Commodore REU base address.
MSE128_BASE:	FD00	FD00	384	Joystick	No	C128 input driver.
MOUSE_JMP:	FE80	FE80	9/15	–	No	Jump table entries for C64 and C128 input driver.
MOUSE_BASE:	FE80	FE80	378	Joystick	No	C64 input driver.
	–	FE8F	–	–	No	C128 Kernal starts again after input driver jump table.
config:	–	FF00	1	CIO_IN	No	C128 **MMU configuration register**.
mmulcra/b/c/d:	–	FF01	1x4	N/A	No	C128 MMU Load configuration registers a,b,c and d.
END_MOUSE:	FFFA	–	–	–	–	Defined as one byte past the end of the C64 mouse driver.

	Address (hex)					
Name	64	128	Size	Default	Saved	Description
systemVector:	–	FFF8	2	**Reset**	No	Soft reset vector. Called when reset button is pressed.
NMIVector:	FFFA	FFFA	2	FB24/FF25	No	Non maskable interrupt vector. (Default: C64/C128)
resetVector:	FFFC	FFFC	2	FB24/FF25	No	Reset vector. (Default: C64/C128)
IRQVector:	FFFE	FFFE	2	FAA2/FF05	No	Interrupt request vector / BRK instruction handler (Default: C64/C128)

C128 backRAM:

Name	64	128	Size	Default	Saved	Description
curEnable:	–	1300[†]	1	None	No	Image of the C64 **mobenble** register.
curmoby2:	–	1301[†]	1	None	No	Image of C64 **moby2** register. Used for C128 soft sprites.
curmobx2:	–	1302[†]	1	None	No	Image of the C64 **mobx2** register. Used for C128 soft sprites.
curXpos0:	–	1303[†]	16	None	No	The current x-positions of the C128 soft sprites.
curYpos0:	–	1313[†]	8	None	No	The current y-positions of the C128 soft sprites.
backBufPtr:	–	131B[†]	16	None	No	Screen pointer where the back buffer came from.
backXBufNum:	–	132B[†]	8	None	No	For each sprite, 1 byte for how many bytes wide the corresponding sprite is.
backYBufNum:	–	1333[†]	8	None	No	For each sprite, Number of scanlines high of corresponding sprite.
sspr1back:	–	133B[†]	294	None	No	Buffers for soft sprites l – 7 are used for saving the screen behind the sprites. Each buffer is 7 bytes wide by 42 scanlines high (292 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. **Note**: If an application is not actively using sprites, this can be an application data area.
sspr2back:	–	1461[†]	294			
sspr3back:	–	1587[†]	294			
sspr4back:	–	16AD[†]	294			
sspr5back:	–	17D3[†]	294			
sspr6back:	–	18F9[†]	294			
sspr7back:	–	1A1F[†]	294			
shiftBuf:	–	1B45[†]	7	None	No	Buffer for shifting/doubling sprites.
shiftOutBuf:	–	1B4C[†]	7	None	No	Buffer for shifting/doubling/oring sprites.
sizeFlags:	–	1B53[†]	7	None	No	Height of sprite | 9-pixel flag.
doRestFlag:	–	1B54[†]	1	0	No	Flag needed because of overlapping soft sprite problems on C128.
mouseSave:	–	1B55[†]	24	None	No	Screen data for what is beneath mouse soft sprite.
softZeros:	–	1B6D[†]	192	None	No	Buffer used for preserving background behind soft sprites.
softOnes:	–	1C2D[†]	192	None	No	Buffer used for preserving background behind soft sprites.
invertBuffer:	–	1CED[†]	80	None	No	Buffer area used to speed up the 80-column **InvertLine** routine.
deskAccSwap:	–	2000[†]	24K	None	No	Desk Accessory swap buffer. 2000-7FFF
	–	A000[†]		–	No	Start of Back Ram Kernal space
NMIVector:	–	FFFA[†]	2	$FF25	No	Non maskable interrupt vector.
resetVector:	–	FFFC[†]	2	$FF25	No	Reset vector.
IRQVector:	–	FFFE[†]	2	$FF05	No	Interrupt request vector / BRK instruction handler

[†] *Located in backRAM*

dlgBoxRamBuf

Dialog Box RAM buffer

This buffer is for variables that are saved when dialog boxes or desk accessories are run. Both of these actions require the system to be able to warmstart GEOS and return to the application state after the action completes. This ability to backup and restore the system state allows for both the Dialog Box / Desk Accessory to startup into a known base startup, just like the application itself always starts up at this same warmstart state.

Breakdown of Dialog Box RAM buffer

dlgBoxRamBuf Expressed as a ramsect declaration:

```
.ramsect     dlgBoxRamBuf
       dbrb_ZP:            .block 23     ; Zero Page variables
       dbrb_GL:            .block 38     ; Global variables
       dbrb_LC:            .block 278    ; Kernal internal variables
       dbrb_SP:            .block 39     ; Sprite data
       dbrb_FUTURE:        .block 39     ; Filler. Current Kernals do not use all of the buffer's
                                         ; 417 bytes
```

dlgBoxRamBuf Converted to CONSTANTS:

CONSTANT	Size	Description
SRAM_ZPSIZE	23	Zero Page variables
SRAM_GLSIZE	38	Global variables
SRAM_LC	278	Kernal internal local variables
SRAM_SPSIZE	39	Sprite data
SRAM_FT	39	Future filler. Current Kernals do not use all of the buffer.
TOT_SRAM_SAVED	417	

SRAM_ZP Zero Page variables.

SRAM_ZP	$Addr	Size	Start of saved Zero Page area
curPattern	22	2	Pointer to the first byte of the graphics data for the current pattern in use.
string	24	2	Pointer to string destinations for routines such as **GetString**.
baselineOffset	26	1	Offset from top line to baseline in character set.
curSetWidth	27	2	Card width in pixels for the current font.
curHeight	29	1	Card height in pixels of the current font in use.
curIndexTable	2A	2	Pointer to the table of sizes, in bytes, of each card in of the current font.
cardDataPntr	2C	2	Pointer to the card graphic data for the current font in use.
currentMode	2E	1	Current text drawing mode. Each bit is a flag for a drawing style.
dispBufferOn	2F	1	Routes graphic and text between foreground and background buffers.
mouseOn	30	1	Mouse/Icon/Menu active bit flag.
msePicPtr	31	2	Pointer to the mouse graphics data.
windowTop	33	1	Top line of window for text clipping.
windowBottom	34	1	Bottom line of window for text clipping.
leftMargin	35	2	Leftmost point for writing characters.
rightMargin	37	2	The rightmost point for writing characters.
SRAM_ZP_END	38		**End of save Zero Page area (inclusive)**
SRAM_ZPSIZE		23	(SRAM_ZP_END+1) – SRAM_ZP

SRAM_GL Global Variables

Name	Addr	Size	Description
SRAM_GL	849B		**Start of Saved Global RAM Area 849B-84C0**
appMain	849B	2	Vector that allows applications to include their own main loop code.
intTopVector	849D	2	Vector to routine to call before operating system interrupt code is run.
intBotVector	849F	2	Vector to routine to call after the operating system interrupt code has run.
mouseVector	84A1	2	Routine to call on a mouse key press.
keyVector	84A3	2	Vector to routine to call on keypress.
inputVector	84A5	2	Pointer to routine to call on input device change.
mouseFaultVec	84A7	2	Vector too routine to call when mouse goes outside defined region.
otherPressVec	84A9	2	Vector to call when the mouse button is pressed outside of Menu/Icon.
StringFaultVec	84AB	2	String margin fault service routine vector.
alarmTmtVector	84AD	2	Service routine for the alarm clock time-out.
BRKVector	84AF	2	Vector to the routine that is called when a BRK instruction is encountered
RecoverVector	84Bl	2	Vector to recover background behind menus and dialog boxes.
selectionFlash	84B3	1	speed at which menu items and icons are flashed.
alphaFlag	84B4	1	Flag for alphanumeric string input.
iconSelFlag	84B5	1	Flag specify how the system should indicate icon selection to the user.
faultData	84B6	1	Holds information about mouse faults.
menuNumber	84B7	1	Number of currently working menu.
mouseTop	84B8	1	Top most position for mouse.
mouseBottom	84B9	1	Bottom most position for mouse cursor.
mouseLeft	84BA	2	Left most position for mouse.
mouseRight	84BC	2	Right most position for mouse.
stringX	84BE	2	The x-position for string input.
stringY	84C0	1	The y-position for string input.
SRAM_GL_END	84C0		**End of Save Global Ram area (inclusive)**
SRAM_GLSIZE		38	(SRAM_GL_END+1) – SRAM_GL

SRAM_LC Kernal Internal Local Variables

SRAM_LC area is for internal Kernal Local Variables and structures. SRAM_LC is comprised of the following:

CONSTANT	Size	Description
MENU_SPACE	49	Variables and tables containing current Menu (3 * MAX_M_NESTING) + (2 * MAX_M_ITEMS) +7
PROC_SPACE	227	Variables and tables holding processes and sleepers. (MAX_PROCESSES * 7) + (SLEEP_MAX * 4) +7
	2	Internal variable holds pointer to current icon table.
SRAM_LC	278	2 + MENU_SPACE + PROC_SPACE

MENU_SPACE Break Down.

```
        MENU_SPACE = (3 * MAX_M_NESTING) + (2 * MAX_M_ITEMS) +7
.ramsect
        ;--- Menu Variables                     7 bytes
        menuOptNumber:      .block              1
        menuTop:            .block              1
        menuBottom:         .block              1
        menuLeft:           .block              2
        menuRight:          .block              2

        ;--- Nesting Tables                            ; MAX_M_NESTING = 4
                                                       ; 3 Tables, allow for 4 menu nesting
        levels
        menuStackL:         .block              4      ; Each level requires 3 bytes to store
        menuStackH:         .block              4
        menuOptionTab:      .block              4      ; Nest size = MAX_NEST * 3

        ;--- Menu Item Tables                          ; MAX_M_ITEMS = 15
                                                       ; Two tables, allows for 15 menu items
        menuLimitTabL:      .block              15     ; Each menu item requires 2 bytes to
                                                       ; store.
        menuLimitTabH:      .block              15     ; Items size = MAX_ITEMS * 2
```

PROC_SPACE Break Down.

```
        PROC_SPACE = (MAX_PROCESSES * 7) + (SLEEP_MAX * 4) +7
        ;---   Processes
        timersTab:          .block              40     ; MAX_PROCESSES = 20 Processes
        timersCMDs:         .block              20     ;
        timersRtns:         .block              40     ; Each process requires 7 bytes to
        store
        timersVals:         .block              40     ; Process size = MAX_PROCESSES * 7

        ;---   2 Bytes of +7
        numTimers:          .block              1      ; Part of + 7
        delaySP:            .block              1      ;

        ;---   Sleepers
        delayValL:          .block              20     ; SLEEP_MAX = 20 sleepers
        delayValH:          .block              20     ; Each sleep requires 4 bytes to store
        delayRtnsL:         .block              20
        delayRtnsH:         .block              20     ; Sleep size = SLEEP_MAX * 4
```

```
;--- Internal variables falling right after process tables.
;--- Last 5 bytes of the +7
stringLen:          .block 1
stringMaxLen:       .block 1
tmpKeyVector:       .block 2
stringMargCtrl:     .block 1
```

SRAM_SP Sprite Data

Save the current state of all 8 sprites.

Name	Addr	Size	Description
obj0Pointer	`8FF8`	8	sprite byte pointers
mob0xpos	`D000`	16	x, y-positions of sprites
msbxpos	`D010`	1	bit 9 of sprite x-positions.
mobenble	`D015`	1	sprite enable bits
mobprior	`D01B`	1	object to background priority
mobmcm	`D01C`	1	sprite multi-color mode select.
mobx2	`D01D`	1	double object size in x.
mcmclr0	`D025`	1	multi-color mode color 0
mcmclr1	`D026`	1	multi-color mode color 1
mob1clr	`D028`	7	Color of sprites 1-7
moby2	`D017`	1	double object size in y
SRAM_SPSIZE		39	

SRAM_FT Future use filler bytes

DBRBSIZE = 417 ; Hard coded size of Dialog Box ram buffer

SRAM_FT = DBRBSIZE – (**SRAM_ZPSIZE + SRAM_ZPSIZE + SRAM_LC + SRAM_SPSIZE**)

Nothing is actually done with the **SRAM_FT** bytes. They are just a place holder in the formula that leads to all 417 Bytes of the buffer being accounted for.

This is the actual table used to control the population of and restoration from the **dlgBoxRamBuf**.

```
DialogCopyTab:
       .word   curPattern        ; zero page variables
       .byte   23
       .word   appMain           ; vectors
       .byte   38
       .word   $40               ; Internal vector
       .byte   2
       .word   $86C0             ; Internal menu tables menuOptNumber
       .byte   49
       .word   $86F1             ; Internal timer tables
       .byte   227
       .word   obj0Pointer       ; obj0Pointer
       .byte   8
       .word   mob0xpos          ; $D000
       .byte   17
       .word   mobenble          ; $D015
       .byte   1
       .word   mobprior          ; $D01B
       .byte   3
       .word   mcmclr0           ; $D025
       .byte   2
       .word   mob1clr           ; $D028
       .byte   7
       .word   moby2             ; $D017
       .byte   1
       .word   NULL
```

Saved RAM Buffer Variables by Name

Name	Description
alarmTmtVector	Service routine for the alarm clock time-out.
alphaFlag	Flag for alphanumeric string input.
appMain	**MainLoop** service routine vector.
BRKVector	Vector to the routine that is called when a BRK instruction is encountered.
baselineOffset	Offset from top line to baseline in character set.
cardDataPntr	This is a pointer to the actual card graphic data for the current font in use.
curHeight	Card height in pixels of the current font in use.
curIndexTable	Pointer to the table of sizes, in bytes, of each card in of the current font.
curPattern	Pointer to the first byte of the graphics data for the current pattern in use.
currentMode	Current text drawing mode. Each bit is a flag for a drawing style.
curSetWidth	Card width in pixels for the current font.
dispBufferOn	Routes graphic and text operations to either the foreground/background/both.
faultData	Holds Information about mouse faults.
fontTable	Variables for the current font in use.
iconSelFlag	Flag specify how the system should indicate icon selection to the user.
inputVector	Pointer to routine to call on input device change.
intBotVector	Vector to routine to call after the operating system interrupt code has run.
intTopVector	Vector to routine to call before operating system interrupt code is run.
keyVector	Vector to routine to call on keypress.
leftMargin	Leftmost point for writing characters.
menuNumber	Number of currently working menu.
mouseBottom	Bottom most position for mouse cursor. Normally set to bottom of the screen.
mouseFaultVec	Vector too routine to call when mouse goes outside defined region.
mouseLeft	Left most position for mouse.
mouseOn	Mouse/Icon/Menu active bit flag.
mouseRight	Right most position for mouse.
mouseTop	Top most position for mouse.
mouseVector	Routine to call on a mouse key press.
msePicPtr	Pointer to the mouse graphics data.
otherPressVec	Vector to call when the mouse button is pressed outside of Menu/Icon.
RecoverVector	Vector to recover background behind menus and dialog boxes.
rightMargin	The rightmost point for writing characters.
selectionFlash	Speed at which menu items and icons are flashed.
string	Pointer to string destinations for routines such as **GetString**.
StringFaultVec	Vector called when an attempt is made to write a character past **rightMargin**.
stringX	The x-position for string input.
stringY	The y-position for string input.
windowTop	Top line of window for text clipping.

Saved I/O by Address

Start	End	Description
mob0xpos	**msbxpos**	All sprite x and y-positions
mobenble		Sprite enable bits for all sprites.
mobprior	**mobx2**	Sprite background priority/color mode and x-doubling.
mcmclr0	**mcmclr1**	Multi color mode colors.
mob1clr	**mob7clr**	Color of all non pointer sprites. (sprites 1-7)
moby2		Sprite y-doubling.

DIALOG

Note[2]: The first entry in a DB table is a command byte defining its position. This can either be a byte indicating a default position for the DB, DEF_DB_POS **(%10000000)**, or a byte indicating a user defined position, SET_DB_POS **(%00000000)** which must be followed by the position information.

The position command byte is or'ed with a system pattern number to be used to fill in a shadow box. The shadow box is a rectangle of the same dimensions as the DB and is filled with one of the system patterns. The shadow box appears underneath the Dialog Box, Offset 1 card right and 1 card down.

```
Start of Default Dialog              Start of Custom Size Dialog
-----------------------------------  ---------------------------------------------
.byte DEF_DB_POS | pattern           .byte   SET_DB_POS | pattern
                                     .byte   top             ; (0-199)
                                     .byte   bottom          ; (0-199)
                                     .word   left            ; (0-319 or 0-639)
                                     .word   right           ; (0-319 or 0-639)
```

Note[1]: Standard window size:

Standard window size:	columns	64-255
	rows	32-127

Note[1]: If the shadow pattern is zero, then no shadow is drawn.

Note[3]: Maximum # of dialog icons is 8. This can be worked around by drawing your own images and detecting mouse clicks over the images with **otherPressVec** and **IsMseInRegion**.

Note: See: "**Saved RAM Buffer Variables by Name**" on the previous page for a list of saved global variables and saved I/O values.

Position Commands

After the position byte (or bytes) may appear a number of icon or command bytes. Most require position coordinates. The x and y-positions are an offset from the upper left corner of the DB.

Icon:	x-position uses bytes (cards)	0-39	x_card_offset	
Text:	x-position uses pixels	0-255	x_offset	; byte sized field
	y-position is always in pixels	0-199	y_offset	

Doubling for 40/80 mode compatibility

Location	Type	Doubling Required	Note
Custom Size	word x-coordinate	Yes	80 only can use native coordinates
Icon / **DBUSRICON**	x_card_offset	No (Optional)	128 GEOS always doubles
DBTXTSTR	x_offset	**No**	Would add 128 to x-position
DBUSRICON structure	byte width in bytes	Yes	80 only can use native width
DBGETFILES	byte x_offset	**No**	Would add 128 to x-position
DBGETSTRING	byte x_offset	**No**	Would add 128 to x-position
DBVARSTR	byte x_offset	**No**	Would add 128 to x-position

Note: See **Ch 19 constants/Dialog Box/coordinates** for a list of CONSTANTS for the dialog window, text and icon positions.

Dialog Box Icons

Icon	Value	Example	Description	Keyboard Shortcut
OK	1	.byte OK .byte x_card_offset .byte y_offset	Draw OK icon x-offset is in cards (0-39) y-offset in pixels (0-199)	RETURN **Note**: Shortcuts will close the dialog box as if the icon was clicked on.
CANCEL	2		Draw CANCEL icon	c
YES	3		etc...	y
NO	4			n
OPEN	5			o
DISK	6			d
	7-10		Marked for future use.	

Dialog Box Commands

Command	Value	Example	Description
DBTXTSTR	11	.byte DBTXTSTR .byte x_offset .byte y_offset .word textPtr	**PutString** *textPtr* at selected offsets. x pixel offset 0-255 y pixel offset 0-199 *textPtr* contains address of null terminated string.
DBVARSTR	12	.byte DBVARSTR .byte x_offset .byte y_offset .byte zPgPtr	**PutString** @@zPgPtr *zPgPtr* is an address of a zero page ptr to a null terminated string. **Example**: .byte **r15**
DBGETSTRING	13	.byte DBGETSTRING .byte x_offset .byte y_offset .byte zPgPtr .byte *maxChars*	Read a text string typed by user into buffer. *zPgPtr* points to address of a buffer that is *maxChars* bytes. **Example**: .byte **r5** with **r5** containing address of string buffer
DBSYSOPV	14	.byte DBSYSOPV	Closes DB when the mouse is pressed anywhere other then over an icon.
DBGRPHSTR	15	.byte DBGRPHSTR .word graphicsStrPtr	**i_GraphicsString** *graphicsStrPtr* *graphicsStrPtr* contains address of a graphics string. ([1]This command will end Dialog Box processing)
DBGETFILES¥	16	.byte DBGETFILES .byte x_offset .byte y_offset	Display the filename box inside the DB. ¥ **r7L** = **FILETYPE** **r5** = buffer **r10** = File Class
DBOPVEC	17	.byte DBOPVEC .word msePressVector	sets **otherPressVec** to *msePressVector*. Called when mouse button pressed any place except over an icon.
DBUSRICON	18	.byte DBUSRICON .byte x_card_offset .byte y_offset .word userIcon	userIcon table: .word ptrIconData .word NULL .byte width in bytes .byte height in pixels .word ptrIconAction **Note**: (width | DOUBLE_B for 128)
DB_USR_ROUT	19	.byte DB_USR_ROUT .word userVector	Call *userVector* after the DB is drawn and before the dialog box icons have been drawn.
NULL	0	.byte NULL	Ends the Dialog Box definition.

GraphicsString

Available commands for **GraphicsString**:

Command	Value	Example	Description
NULL	0	.byte NULL	Ends the graphics string
MOVEPENTO	1	.byte MOVEPENTO .word xPos .byte yPos	Move the pen position to the absolute coordinates (xPos, yPos)
LINETO	2	.byte LINETO .word xPos .byte yPos	Draw a line from the current pen position to (xPos, yPos), which becomes new pen position.
RECTANGLETO	3	.byte RECTANGLETO .word xPos .byte yPos	Draw a rectangle using the pattern byte; from the current pen position to (xPos, yPos), which becomes new pen position.
	4	unused	
NEWPATTERN	5	.byte NEWPATTERN .byte patternNbr	Load system pattern with new pattern.
ESC_PUTSTRING	6	.byte ESC_PUTSTRING .word xPos .byte yPos .byte "String",NULL	Switch to interpreting the remainder of the string as **i_PutString** inline commands.
FRAME_RECTO	7	.byte FRAME_RECTO .word xPos .byte yPos	Frame a rectangle using a solid line. Start at the current pen position to (xPos, yPos), which becomes the new pen position.
PEN_X_DELTA	8	.byte PEN_X_DELTA .word xOffset	move pen by signed word delta in xOffset
PEN_Y_DELTA	9	.byte PEN_Y_DELTA .word yOffset	move pen by signed word delta in yOffset
PEN_XY_DELTA	10	.byte PEN_XY_DELTA .word xOffSet .word yOffSet	move pen by signed word delta in xOffset & yOffset

Example: **GrphcsStr1**

Icon Table

Header:

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

Icon Entries:

+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.

Menu

```
M_HEIGHT=14
MAX_M_ITEMS=15
```

Menu/Sub-menu Header:

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

Menu Item: (OFF_1ST_M_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.

Types of Menu Items (for use in OFF_TYPE_ITEM byte):

Constant	Value	Description
SUB_MENU	$80	This menu item leads to a sub-menu. The *OFF_POINTER_ITEM* is a pointer to the sub-menu data structure (points to first byte of a menu/sub-menu header).
DYN_SUB_MENU	$40	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 preprocessing and return with **r0** containing a pointer to a sub-menu data structure or $0000 to ignore the selection.
MENU_ACTION	$00	This menu item generates an event. The *OFF_POINTER_ITEM* is a pointer to the event routine to call.

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

Constant	Value	Description
HORIZONTAL	$00	Arrange menu items in this menu/sub-menu horizontally.
VERTICAL	$80	Arrange menu items in this menu/sub-menu vertically.
CONSTRAINED	$40	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	$00	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 (OFF_M_ATTRIBUTE):

7	6	5	4	3	2	1	0
b7	b6	n/a		b3-b0			

b7 orientation: 1 = VERTICAL; 0 = HORIZONTAL
b6 constrained: 1 = CONSTRAINED; 0 = UN_CONSTRAINED
b3-b0 number of items in menu/sub-menu (up to MAX_M_ITEMS).

Two Examples of the attribute byte:

```
.byte (7 | VERTICAL | UN_CONSTRAINED)  ; vertical, unconstrained menu with seven items

.byte (11 | HORIZONTAL | CONSTRAINED)  ; horizontal, constrained menu with eleven items
```

DirHeader: curDirHead $8200

Offset	Constant	Size	Description
$03		1	1571 double sided flag. $80=double sided format.
$04	OFF_TO_BAM	140	first BAM entry
$90	OFF_DISK_NAME	16	disk name string
$A2	OFF_DSK_ID	2	disk ID
$AB	OFF_OP_TR_SC	2	track and sector for off page directory
$AD	OFF_GS_ID	16	GEOS ID string
$BD	OFF_GS_DTYPE	1	GEOS disk type 0 = normal disk 'B' = BOOT disk 'P' = Master disk

Directory Entry: dirEntryBuf $8400

Offset	Constant	Size	Description
$00	OFF_CFILE_TYPE	1	DOS file type Bit 7 1=file closed/normal state Bit 6 write protect bit ST_WR_PR %01000000 Bit 2-0 Commodore file type DEL = 0 deleted SEQ = 1 sequential PRG = 2 program USR = 3 user (GEOS) REL = 4 relative file. invalid in GEOS CBM = 5 1581 Partition. not supported by GEOS
$01	OFF_INDEX_PTR OFF_DE_TR_SC	2	index table pointer (VLIR file T/S) track/Sector for file's 1st data block
$03	OFF_FNAME	16	file name padded with hard spaces $A0
$13	OFF_GHDR_PTR	2	track/sector of GEOS header block
$15	OFF_GSTRUC_TYPE	1	GEOS file structure type SEQUENTIAL=0 VLIR=1
$16	OFF_GFILE_TYPE	1	GEOS file type NOT_GEOS=0 C64 file no header BASIC=1 C64 BASIC w/header ASSEMBLY=2 C64 Assembly w/header DATA=3 C64 DATA File w/header SYSTEM=4 GEOS system file DESK_ACC=5 GEOS desk accessory APPLICATION=6 GEOS application APPL_DATA=7 GEOS data file FONT=8 GEOS font PRINTER=9 GEOS print driver INPUT_DEVICE=10 GEOS mouse / joystick / etc. DISK_DEVICE=11 GEOS disk driver SYSTEM_BOOT=12 GEOS boot file TEMPORARY=13 GEOS swap file (The deskTop will automatically delete all temporary files when opening a disk) AUTO_EXEC=14 application to automatically be ran just after booting, but before deskTop runs. INPUT_128=15 128 input driver
$17	OFF_YEAR	5	Y/M/D/H/M
$1C	OFF_SIZE	2	file size in blocks

File Header Block: **fileHeader** $8100

Offset	Constant	Size	Description
$00		2	$00, $FF When creating a file with **SaveFile** , this location holds a word pointer to a buffer containing the filename
$02	O_GHIC_WIDTH	1	width in bytes of file icon
$03	O_GHIC_HEIGHT	1	height of file icon in pixels
$04	O_GHIC_PIC	64	icon data
$44	O_GHCMDR_TYPE	1	Commodore file type (b7 = 1, b6-b0 = file type) DEL = 0 deleted SEQ = 1 sequential PRG = 2 program USR = 3 user (GEOS) REL = 4 relative file. invalid in GEOS CBM = 5 1581 Partition. Not supported by GEOS
$45	O_GHGEOS_TYPE	1	GEOS file type NOT_GEOS = 0 C64 file no header BASIC = 1 C64 BASIC w/header ASSEMBLY = 2 C64 assembly w/header DATA = 3 C64 data file w/header SYSTEM = 4 GEOS system file DESK_ACC = 5 GEOS desk accessory APPLICATION = 6 GEOS application APPL_DATA = 7 GEOS data file FONT = 8 GEOS font PRINTER = 9 GEOS print driver INPUT_DEVICE = 10 GEOS mouse etc. DISK_DEVICE = 11 GEOS DISK driver SYSTEM_BOOT = 12 GEOS boot file TEMPORARY = 13 GEOS swap file AUTO_EXEC = 14 application ran while booting INPUT_128 = 15 128 input driver
$46	O_GHSTR_TYPE	1	GEOS file structure type (0=SEQUENTIAL, 1=VLIR)
$47	O_GHST_ADDR	2	start address of file
$49	O_GHEND_ADDR	2	end address of file (Only valid for desk accessories)
$4B	O_GHST_VEC	2	application initialization vector
$4D	O_GHFNAME	12	permanent filename (for all but APPL_DATA files)
	O_GHCNAME		permanent class name (for APPL_DATA files)
		4	version string. Example: V1.0 or extended version V1.000
		3	normally 3 zeros. First 2 bytes may be used for extended version.
$60	O_128_FLAGS	1	OS compatibility flag Constant b7 b6 CF_40 = $00 0 0 64/128 40-column mode only CF_40_80 = $40 0 1 64/128 40 and 80-column modes CF_64 = $80 1 0 64 only (does not run under GEOS 128) CF_128 = $C0 1 1 128 80-column mode only (does not run under GEOS 64)
$61	O_GH_AUTHOR	20	application author's name (only if application)
	O_GHP_DISK		disk name of parent application's disk (only if data file) (This was never implemented and included here only for completeness)
$75	O_GHP_FNAME	20	parent application's permanent filename (only if Data File)
$89	O_GHAPDAT	23	data area for application use
$A0	O_GHINFO_TXT	96	notes that are stored with the file and edited in the deskTop "get info" box. null terminated

Note: use **GetFHdrInfo** to load a file's header block into **fileHeader**.

Fonts use the data area of the file header block from $61 to $9F in a different way:

Offset	Constant	Size	Description
`$61`	`O_GHSETLEN`	30	`VLIR size (word) of each point size. 15 words`
`$80`	`O_GHFONTID`	2	**`Font style ID`** `(word)`
`$82`	`O_GHPTSIZES`	30	`list of` **`Character Set IDs.`** `15 words`

Disk Errors:

GEOS I/O Routines return errors in the x-register:

Constant	Dec	Hex	Description
NO_ERROR	0	$00	No error occurred
NO_BLOCKS	1	$01	Not enough blocks on disk
INV_TRACKS	2	$02	Invalid track or sector
INSUFF_SPACE	3	$03	Disk full, insufficient space
FULL_DIRECTORY	4	$04	Directory is full
FILE_NOT_FOUND	5	$05	File not found
BAD_BAM	6	$06	Bad allocation map (attempt to deallocate an unallocated block, or the reverse)
UNOPENED_VLIR	7	$07	VLIR file not open
INV_RECORD	8	$08	VLIR record does not exist (This is a non fatal error)
OUT_OF_RECORDS	9	$09	Out of records: Too many VLIR chains
STRUCT_MISMATCH	10	$0A	GEOS structure mismatch: File is not a VLIR file
BFR_OVERFLOW	11	$0B	Buffer overflow: **ReadRecord** max read size exceeded
CANCEL_ERR	12	$0C	Deliberate cancel error
DEV_NOT_FOUND	13	$0D	Device not found
INCOMPATIBLE	14	$0E	Incompatible 40/80
HDR_NOT_THERE	32	$20	Disk block read error: No header block sync character
NO_SYNC	33	$21	Unformatted or missing disk
DBLK_NOT_THERE	34	$22	No data Block found
DAT_CHKSUM_ERR	35	$23	Data block checksum error
WR_VER_ERR	37	$25	Write verify error
WR_PR_ON	38	$26	Write protect on
HDR_CHKSUM_ERR	39	$27	Disk block write: Header checksum error
DSK_ID_MISMAT	41	$29	Disk ID mismatch
BYTE_DEC_ERR	46	$2E	Drive speed read error
DOS_MISMATCH	115	$73	Wrong DOS indicator

GEOS Input Control Codes **keyData**

Code	Constant	Description / common action
$00	*N/A*	*Cannot be created by a keyboard sequence in GEOS*
$01	KEY_F1	Function key F1
$02	KEY_F2	Function key F2
$03	KEY_F3	Function key F3
$04	KEY_F4	Function key F4
$05	KEY_F5	Function key F5
$06	KEY_F6	Function key F6
$07	KEY_NOSCRL¥ǂ	Pause/resume scrolling
$08	KEY_LEFT or BACKSPACE	Move cursor left one character in geoWrite. Other applications may delete the previous character.
$09	KEY_TAB*	geoWrite uses to represent a tab
$0A	KEY_LF¥ǂ	
$0B	*N/A*ǂ	*unused*
$0C	*N/A*ǂ	*unused*
$0D	KEY_ENTER	Carriage return: move current cursor position down one line and over to the left-margin (value in **leftMargin**). Mirror behavior of OK button in dialogs.
$0E	KEY_F7	Function key F7
$0F	KEY_F8	Function key F8
$10	KEY_UP	Up arrow
$11	KEY_DOWN	Move down a line
$12	KEY_HOMEǂ	Move cursor to top of current page
$13	KEY_CLEARǂ	Clear edit field
$14	KEY_LARROW	Used in geoWrite with C= for previous page.
$15	KEY_UPARROWǂ	This keystroke is always translated to ^ (or |) and will never appear in **keyData** as $15
$16	KEY_STOP	Used by geoProgrammer for interrupting a process
$17	KEY_RUN	
$18	KEY_BPSǂ	British pound symbol
$19	KEY_HELP¥ǂ	Display context relative information to the user
$1A	KEY_ALT¥ǂ	
$1B	KEY_ESC¥ǂ	Mirror behavior of CANCEL button in dialogs
$1C	KEY_INSERT	Delete the previous character. (geoWrite mirrors the function of the delete key)
$1D	KEY_DELETE	Delete the previous character
$1E	KEY_RIGHT	Move cursor right one character in geoWrite. geoPaint mirrors function of KEY_LEFT
$1F	KEY_INVALID	Unexpected scan code lookup result

Special key: **RESTORE** The restore key generates a NMI. The restore key will never generate a keypress event.

C= + keypress combinations have bit 7 set in **keyData**.

Example: User presses C=+'A' and **keyData** will contain 'A' | SHORTCUT ($C1)

Notes: Control codes $01-1A can be created with the keyboard combinations **CONTROL** + A-Z

ǂNot used by Berkeley applications

¥C128 keyboard only

*C64: Tab = **CONTROL** + I
C128: Tab = **TAB** key or Tab = **CONTROL** + I

```
KEY_QUEUE_SIZE      = 16               ; size of the keyboard queue (buffer)
KEY_REPEAT_COUNT    = 15               ; 1/4 second: auto-repeat time
```

GEOS Text Escape Character Codes

Code	Constant	Description
$00	NULL	String termination character
$01	[†]	*unused*
$02	[†]	*unused*
$03	[†]	*unused*
$04	[†]	*unused*
$05	[†]	*unused*
$06	[†]	*unused*
$07	[†]	*unused*
$08	BACKSPACE	Erase the previous character
$09	FORWARDSPACE TAB[*]	Not implemented in GEOS 64 or GEOS 128 geoWrite uses to represent a tab (use TAB constant)
$0A	LF	Line feed: Move current printing position down one line (value in **curHeight**)
$0B	HOME	Move current printing position to upper-left screen corner
$0C	UPLINE PAGE_BREAK	Move current printing position up one line (value in **curHeight**) geoWrite uses for page-break
$0D	CR	Carriage return: move current printing position down one line and over to the left-margin (value in **leftMargin**)
$0E	ULINEON	Begin underlining
$0F	ULINEOFF	End underlining
$10	ESC_GRAPHICS[¥]	Escape code for graphics string. Remainder of this string is treated as input to the **GraphicsString** routine.
$11	ESC_RULER	Unimplemented. This escape code is ignored by GEOS text routines. This escape code is used by geoWrite to represent a ruler escape.
$12	REV_ON	Begin reverse video printing (white on black)
$13	REV_OFF	End reverse video printing
$14	GOTOX[¥]	Change the x-coordinate of the current printing position to the word value stored in the following two bytes
$15	GOTOY[¥]	Change the y-coordinate of the current printing position to the byte value in the following byte
$16	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.
$17	NEWCARDSET [¥]	Unimplemented. This does nothing but skip over the following three bytes geoWrite uses for font changes.
$18	BOLDON	Begin boldface printing
$19	ITALICON	Begin italicized printing
$1A	OUTLINEON	Begin outlined printing
$1B	PLAINTEXT	Begin plain text printing (turns off all type style attributes)
$1C	[†]	*unused*
$1D	[†]	*unused*
$1E	[†]	*unused*
$1F	[†]	*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**.

[*] C64: Tab = **CONTROL** + I.
C128: Tab = **TAB** key or Tab = **CONTROL** + I.

GEOS ASCII Character Codes

Code		Character	Code		Character	Code		Character
$20	32	space	$41	65	A	$61	97	a
$21	33	!	$42	66	B	$62	98	b
$22	34	"	$43	67	C	$63	99	c
$23	35	#	$44	68	D	$64	100	d
$24	36	$	$45	69	E	$65	101	e
$25	37	%	$46	70	F	$66	102	f
$26	38	&	$47	71	G	$67	103	g
$27	39	'	$48	72	H	$68	104	h
$28	40	(	$49	73	I	$69	105	i
$29	41	)	$4A	74	J	$6A	106	j
$2A	42	*	$4B	75	K	$6B	107	k
$2B	43	+	$4C	76	L	$6C	108	l
$2C	44	,	$4D	77	M	$6D	109	m
$2D	45	-	$4E	78	N	$6E	110	n
$2E	46	.	$4F	79	O	$6F	111	o
$2F	47	/	$50	80	P	$70	112	p
$30	48	0	$51	81	Q	$71	113	q
$31	49	1	$52	82	R	$72	114	r
$32	50	2	$53	83	S	$73	115	s
$33	51	3	$54	84	T	$74	116	t
$34	52	4	$55	85	U	$75	117	u
$35	53	5	$56	86	V	$76	118	v
$36	54	6	$57	87	W	$77	119	w
$37	55	7	$58	88	X	$78	120	x
$38	56	8	$59	89	Y	$79	121	y
$39	57	9	$5A	90	Z	$7A	122	z
$3A	58	:	$5B	91	[	$7B	123	{
$3B	59	;	$5C	92	\	$7C	124	|
$3C	60	<	$5D	93	]	$7D	125	}
$3D	61	=	$5E	94	^	$7E	126	~
$3E	62	>	$5F	95	_	$7F	127	USELAST[†]
$3F	63	?	$60	96	`	$80	128	C= [¥]
$40	64	@						

[†] deletion character. Use USELAST with **GetRealSize** to get the size of the last printed character in order to erase it from the screen.

[¥] SHORTCUT symbol included in BSW system fonts. **Note**: Shortcut key codes in **keyData** have bit 7 set.

Special keys:	TAB	CONTROL I	; C128 can also use the TAB key
underline:	_	C= -	; Valid character for use in labels
vertical bar	|	C= ⇧	; Logical OR
tilde	~	C= *	; One's complement / negate
open brace	{	C= [	; Same behavior as (
close brace	}	C=]	; Same behavior as)
backslash	\	C= /	
grave accent	`	C= @	
circumflex	^	⇧	; Bitwise XOR

Memory Map

Address (Hex)	Size	Description
00		**zpage**. See "**Zero Page**" In **Appendix E**: **Memory Maps** for full details 6510 Data direction register
01		**6510 I/O register**
02-21	32	**pseudoregisters r0-r15**
70-7F	16	application .zsect space (placeholder names **a2-a9**)
80-FA	123	Kernal .zsect (applications can use this as .zsect space while using **SwZp**)
FB-FE	4	application .zsect space (placeholder names **a0-a1**)
100-1FF	256	6510 stack
200-313	276	**APP_LVAR**
334-3FF	200	**AppLowRAM**
400-5FFF	$5C00	Application program and data
6000-7F3F (7900)	$1F40 $640	Background screen RAM (Load address for print drivers)
7F40-7FFF	192	Application RAM
8000-80FF	256	**diskBlkBuf** General disk block buffer
8100-81FF	256	**fileHeader** File header block buffer track/sector table for VLIR files
8200-82FF	256	**curDirHead** Disk header
8300-83FF	256	**fileTrScTab** File track and sector chain
8400-841E	30	**dirEntryBuf** Directory entry
841E-849A		**Disk Variables**
849B-84B2		**Vectors** application controlled Kernal vectors
84B3-851C		**Kernal Variables**
88BB-888F		**Upper Kernal Variables**
8900-89FF	256	**dir2Head** 2nd BAM for 1571/1581 drives
8A00-8BFF		Sprite picture data. (See **OsVars** in **Appendix E**: for more details)
8C00-8FE7		Video color matrix
8FF8-8FFF		Sprite pointers
9000-9FFF		Disk driver
A000-BF3F		Foreground screen RAM or second half of background screen in 80 col
BF40-BFFF		GEOS tables
C000-CFFF D000-DFFF E000-XXXX	$1000 $1000	4k GEOS Kernal code, always resident 4k GEOS Kernal code or I/O space 8k GEOS Kernal code, always resident. Stops at start of input driver
D000		**vicbase** Video Interface Chip base address
D400		**sidbase** Sound Interface Device
D600		**VDC Registers** C128 80 column screen control
DC00		**cia1base** Complex Interface Adapter
DD00		**cia2base** Complex Interface Adapter #2
DF00		**REU Area**
FD00-FE7F	384	C128 input driver
FE80-FFF8 FE89-FFF9	9 369	Input driver jump table (C64, C128) C64 input driver
FFFA-FFFF		6510 NMI, IRQ, and reset vectors

GEOS Kernal 2.0

Alphabetical Listings of Routines

Name	Addr	Description	Category	Page
AccessCache	C2EF	C128 Provides a mechanism for disk drivers to cache up to 21 blocks.	**memory**	20-148
AllocateBlock	9048	Mark a disk block as in-use.	**disk mid-level**	20-6
AppendRecord	C289	Insert a new VLIR record after the current record.	**disk VLIR**	20-73
BBMult	C160	Byte by byte (single-precision) unsigned multiply.	**math**	20-136
Bell	n/a	1000 Hz Bell sound.	**utility**	20-214
BitmapClip	C2AA	Display a compacted bitmap, clipping to a sub-window.	**graphics**	20-85
BitmapUp	C142	Display a compacted bitmap without clipping.	**graphics**	20-87
BitOtherClip	C2C5	**BitmapClip** with data coming from elsewhere (e.g., disk).	**graphics**	20-88
BldGDirEntry	C1F3	Build a GEOS directory entry in memory.	**disk mid-level**	20-7
BlkAlloc	C1FC	Allocate space on disk.	**disk mid-level**	20-8
BlockProcess	C10C	Block process from running. Does not freeze timer.	**process**	20-185
Bmult	C163	Byte by word unsigned multiply.	**math**	20-137
BootGEOS	C000	Reboot GEOS. Requires only 128 bytes at $C000.	**internal**	20-127
CalcBlksFree	C1DB	Calculate total number of free disk blocks.	**disk mid-level**	20-10
GetScanLine	C13C	Calculate scanline address.	**graphics**	20-95
CallRoutine	C1D8	pseudo-subroutine call. $0000 aborts call.	**utility**	20-215
ChangeDiskDevice	C2BC	Change disk drive device number.	**disk very low-level**	20-11
ChkDkGEOS	C1DE	Check if a disk is GEOS format.	**disk mid-level**	20-12
ClearMouseMode	C19C	Stop input device monitoring.	**mouse/sprite**	20-167
ClearRam	C178	Clear memory to $00.	**memory**	20-149
CloseRecordFile	C277	Close/Save currently open VLIR file.	**disk VLIR**	20-74
CmpFString	C26E	Compare two fixed-length strings.	**memory**	20-150
CmpString	C26B	Compare two null-terminated strings.	**memory**	20-151
ColorCard	C2F8	C128 Get or Set a Color Card. In 40 or 80-column mode.	**graphics**	20-90
ColorRectangle	C26B	C128 Draw a Color rectangle on the 80-column Screen.	**graphics**	20-91
CopyFString	C268	Copy a fixed-length string.	**memory**	20-152
CopyString	C265	Copy a null-terminated string.	**memory**	20-153
CRC	C20E	Cyclic Redundancy Check calculation.	**utility**	20-216
Dabs	C16F	Double-precision signed absolute value.	**memory**	20-138
Ddec	C175	Double-precision unsigned decrement.	**math**	20-139
Ddiv	C169	Double-precision unsigned division.	**math**	20-140

Name	Addr	Description	Category	Page
DeleteFile	C238	Delete file.	**disk high-level**	20-13
DeleteRecord	C283	Delete current VLIR record.	**disk VLIR**	20-75
DisablSprite	C1D5	Disable sprite.	**sprite**	20-194
DMult	C166	Double-precision unsigned multiply.	**math**	20-142
Dnegate	C172	Double-precision signed negation.	**math**	20-143
DoBOp	C2EC	C128 Back-RAM memory move/swap/verify primitive.	**memory**	20-154
DoDlgBox	C256	Display and begin interaction w/dialog box.	**dialog box**	20-2
DoIcons	C15A	Display and begin interaction with icons.	**icon/menu**	20-113
DoInlineReturn	C2A4	Return from inline subroutine.	**utility**	20-217
DoMenu	C151	Display and begin interaction with menus.	**icon/menu**	20-114
DoneWithIO	C25F	Restore system after serial I/O.	**disk very low-level**	20-14
DoPreviousMenu	C190	Retract sub-menu and reactivate menus up one level.	**icon/menu**	20-116
DoRAMOp	C2D4	Primitive for communicating with REU (RAM-Expansion Unit).	**memory**	20-155
DrawLine	C130	Draw, clear, or recover line between two endpoints.	**graphics**	20-92
DrawPoint	C133	Draw, clear, or recover a single screen point.	**graphics**	20-93
DrawSprite	C1C6	Define sprite image.	**sprite**	20-195
DSdiv	C16C	Double-precision signed division.	**math**	20-144
DShiftLeft	C15D	Double-precision left shift (zeros shifted in).	**math**	20-145
DShiftRight	C262	Double-precision right shift (zeros shifted in).	**math**	20-146
EnableProcess	C109	Make a process runnable immediately.	**process**	20-186
EnablSprite	C1D2	Enable sprite.	**sprite**	20-196
EnterDeskTop	C22C	Leave application and return to GEOS deskTop.	**disk high-level**	20-15
EnterTurbo	C214	Activate disk turbo on current drive.	**disk very low-level**	20-16
ExitTurbo	C232	Deactivate disk turbo on current drive.	**disk very low-level**	20-17
FastDelFile	C244	Quick file delete (requires full track/sector list).	**disk mid-level**	20-18
FetchRAM	C2CB	Transfer data from RAM-Expansion Unit.	**memory**	20-156
FillRam	C17B	Fill memory with a particular byte.	**memory**	20-157
FindBAMBit	C2AD	Get allocation status of particular disk block.	**disk mid-level**	20-19
FindFile	C20B	Search for a particular file.	**disk high-level**	20-20
FindFTypes	C23B	Find all files of a particular GEOS type.	**disk high-level**	20-21
FirstInit	C271	Initialize GEOS variables.	**internal**	20-128
FollowChain	C205	Follow chain of sectors, building track/sector table.	**disk mid-level**	20-23
FrameRectangle	C127	Draw an outline in a pattern.	**graphics**	20-94
FreeBlock	C2B9	Mark a disk block as not-in-use in BAM.	**disk mid-level**	20-24
FreeFile	C226	Free all blocks associated with a file.	**disk mid-level**	20-25

Name	Addr	Description	Category	Page
FreezeProcess	C112	Pause a process countdown timer.	**process**	20-187
Get1stDirEntry	9030	Get first directory entry.	**disk mid-level**	20-26
GetBlock	C1E4	Read single disk block into memory.	**disk low-level**	20-27
GetBufBlock	903C	Read single disk block into **diskBlkBuf**	**disk low-level**	20-28
GetCharWidth	C1C9	Calculate width of char without style attributes.	**text**	20-199
GetDimensions	790C	Get CBM printer page dimensions.	**print driver**	20-176
GetDirHead	C247	Read directory header into memory.	**disk mid-level**	20-29
GetFHdrInfo	C229	Read a GEOS file header into **fileHeader**.	**disk mid-level**	20-30
GetFile	C208	Load GEOS file.	**disk high-level**	20-31
GetFreeDirBlk	C1F6	Find an empty directory slot.	**disk mid-level**	20-34
GetNextChar	C2A7	Get next character from keyboard queue.	**text**	20-200
GetNxtDirEntry	9033	Get directory entry other than first.	**disk mid-level**	20-36
GetOffPageTrSc	9036	Get track and sector of off-page directory.	**disk mid-level**	20-37
GetPtrCurDkNm	C298	Return pointer to current disk name.	**disk mid-level**	20-38
GetRandom	C187	Calculate new **random** number.	**utility**	20-218
GetRealSize	C1B1	Calculate actual character size with attributes.	**text**	20-201
GetScanLine	C13C	Calculate scanline address.	**graphics**	20-95
GetSerialNumber	C196	Return GEOS serial number.	**internal**	20-129
GetString	C1BA	Get string input from user.	**text**	20-202
GotoFirstMenu	C1BD	Retract all sub-menus and reactivate at main level.	**icon/menu**	20-117
GraphicsString	C136	Execute a string of graphics commands.	**graphics**	20-96
HideOnlyMouse	C2F2	Temporarily remove 128 soft-sprite mouse pointer.	**mouse/sprite**	20-168
HorizontalLine	C118	Draw a horizontal line in a pattern.	**graphics**	20-98
i_BitmapUp	C1AB	Inline **BitmapUp**.	**graphics**	20-87
i_FillRam	C1B4	Inline **FillRam**.	**memory**	20-157
i_FrameRectangle	C1A2	Inline **FrameRectangle**.	**graphics**	20-94
i_GraphicsString	C1A8	Inline **GraphicsString**.	**graphics**	20-96
i_ImprintRectangle	C253	Inline **ImprintRectangle**.	**graphics**	20-100
i_MoveData	C1B7	Inline **MoveData**.	**memory**	20-160
i_PutString	C1AE	Inline **PutString**.	**text**	20-210
i_RecoverRectangle	C1A5	Inline **RecoverRectangle**.	**graphics**	20-105
i_Rectangle	C19F	Inline **Rectangle**.	**graphics**	20-106
ImprintRectangle	C250	Imprint rectangular area to background buffer.	**graphics**	20-100
InitForIO	C25C	Prepare system for serial I/O.	**disk very low-level**	20-39
InitForPrint	7900	Initialize printer (once per document).	**print driver**	20-177

Alphabetical Listings of Routines

Name	Addr	Description	Category	Page
InitMouse	`FE80`	Initialize input device.	**input driver**	20-122
InitProcesses	`C103`	Initialize processes.	**process**	20-188
InitRam	`C181`	Initialize memory areas from table.	**memory**	20-158
InitTextPrompt	`C1C0`	Initialize text prompt.	**text**	20-204
InsertRecord	`C286`	Insert new VLIR record in front of current record.	**disk VLIR**	20-76
InterruptMain	`C100`	Main interrupt level processing.	**internal**	20-130
InvertLine	`C11B`	Invert the pixels on a horizontal screen line.	**graphics**	20-99
InvertRectangle	`C12A`	Invert the pixels in a rectangular screen area.	**graphics**	20-101
IsMseInRegion	`C2B3`	Check if mouse is within a screen region.	**mouse/sprite**	20-169
LdApplic	`C21D`	Load GEOS application.	**disk mid-level**	20-40
LdDeskAcc	`C217`	Load GEOS desk accessory.	**disk mid-level**	20-42
LdFile	`C211`	Load GEOS data file.	**disk mid-level**	20-44
LoadCharSet	`C1CC`	Load and activate a new font.	**text**	20-205
MainLoop	`C1C3`	GEOS **MainLoop** processing.	**internal**	20-131
MouseOff	`C18D`	Disable mouse pointer and GEOS mouse tracking.	**mouse/sprite**	20-170
MouseUp	`C18A`	Enable mouse pointer and GEOS mouse tracking.	**mouse/sprite**	20-171
MoveBData	`C2E3`	128 BackRAM memory move routine.	**memory**	20-159
MoveData	`C17E`	Intelligent memory block move.	**memory**	20-160
NewDisk	`C1E1`	Initialize a drive.	**disk mid-level**	20-45
NextRecord	`C27A`	Make next VLIR the current record.	**disk VLIR**	20-77
NormalizeX	`C2E0`	Normalize C128 X-coordinates for 40/80 modes.	**graphics**	20-102
NxtBlkAlloc	`C24D`	Version of **BlkAlloc** that starts at a specific block.	**disk mid-level**	20-46
OpenDisk	`C2A1`	Open disk in current drive.	**disk high-level**	20-48
OpenRecordFile	`C274`	Open VLIR file on current disk.	**disk VLIR**	20-78
Panic	`C2C2`	System-error dialog box.	**internal**	20-132
PointRecord	`C280`	Make specific VLIR record the current record.	**disk VLIR**	20-79
PosSprite	`C1CF`	Position sprite.	**sprite**	20-197
PreviousRecord	`C27D`	Make previous VLIR record the current record.	**disk VLIR**	20-80
PrintASCII	`790F`	Send ASCII data to printer.	**print driver**	20-178
PrintBuffer	`7906`	Send graphics data to printer.	**print driver**	20-179
PromptOff	`C29E`	Turn off text prompt.	**text/keyboard**	20-206
PromptOn	`C29B`	Turn on text prompt.	**text/keyboard**	20-207
PurgeTurbo	`C235`	Remove disk turbo from current drive.	**disk very low-level**	20-49
PutBlock	`C1E7`	Write single disk block from memory.	**disk low-level**	20-50
PutBufBlock	`903F`	Write single disk block from **diskBlkBuf**.	**disk low-level**	20-51

Alphabetical Listings of Routines

Name	Addr	Description	Category	Page
PutChar	C145	Display a single character to screen.	**text**	20-208
PutDecimal	C184	Format and display an unsigned double-precision nbr.	**text**	20-209
PutDirHead	C24A	Write directory header to disk.	**disk mid-level**	20-52
PutString	C148	Print string of characters to screen.	**text**	20-210
ReadBlock	C21A	Get disk block primitive.	**disk very low-level**	20-53
ReadByte	C2B6	Read a File 1 byte at a time.	**disk mid-level**	20-54
ReadFile	C1FF	Read chained list of blocks into memory.	**disk mid-level**	20-55
ReadLink	904B	Read track/sector link.	**disk very low-level**	20-57
ReadRecord	C28C	Read current VLIR record into memory.	**disk VLIR**	20-81
RecoverAllMenus	C157	Recover all menus from background buffer.	**icon/menu**	20-118
RecoverLine	C11E	Recover horizontal screen line from background buffer.	**graphics**	20-104
RecoverMenu	C154	Recover single menu from background buffer.	**icon/menu**	20-119
RecoverRectangle	C12D	Recover rectangular screen area from background buffer.	**graphics**	20-105
Rectangle	C124	Draw a filled rectangle.	**graphics**	20-106
ReDoMenu	C193	Reactivate menus at the current level.	**icon/menu**	20-120
RenameFile	C259	Rename GEOS disk file.	**disk mid-level**	20-58
Reset	03E4	C128 Soft reset handler. Located in BackRAM	**internal**	20-133
ResetHandle	C003	Internal Bootstrap entry point.	**internal**	20-134
RestartProcess	C106	Unblock, unfreeze, and restart process.	**process**	20-189
RstrAppl	C23E	Leave desk accessory and return to calling application.	**disk mid-level**	20-59
RstrFrmDialog	C2BF	Exits from a dialog box.	**dialog box**	20-2
SaveFile	C1ED	Save Memory to create a GEOS file.	**disk high-level**	20-60
SetColorMode	C2F5	Change GEOS 128 80-column Color Mode.	**graphics**	20-107
SetDevice	C2B0	Establish communication with a new serial device.	**disk high-level**	20-62
SetGDirEntry	C1F0	Create and save a new GEOS directory entry.	**disk mid-level**	20-63
SetGEOSDisk	C1EA	Convert normal CBM disk into GEOS format disk.	**disk high-level**	20-65
SetMouse	FD09	C128 Reset input device scanning circuitry.	**input driver**	20-123
SetMsePic	C2DA	Set and preshift new soft-sprite mouse picture.	**mouse/sprite**	20-172
SetNewMode	C2DD	Change GEOS 128 graphics mode (40/80 switch).	**graphics**	20-108
SetNextFree	C292	Search for nearby free disk block and allocate it.	**disk mid-level**	20-66
SetNLQ	7915	Begin near-letter quality printing.	**print driver**	20-180
SetPattern	C139	Set current fill pattern.	**graphics**	20-109
Sleep	C199	Put current subroutine to sleep for a specified time.	**process**	20-190
SlowMouse	FE83	Reset mouse velocity variables.	**input driver**	20-124
SmallPutChar	C202	Fast character print routine.	**text**	20-211

Alphabetical Listings of Routines

Name	Addr	Description	Category	Page
StartAppl	C22F	Warmstart GEOS and start application in memory.	**disk mid-level**	20-68
StartASCII	7912	Begin ASCII mode printing.	**print driver**	20-181
StartMouseMode	C14E	Start monitoring input device.	**mouse/sprite**	20-173
StartPrint	7903	Begin graphics mode printing.	**print driver**	20-182
StashRAM	C2C8	Transfer memory to RAM-Expansion Unit.	**memory**	20-161
StopPrint	7909	End page of printer output.	**print driver**	20-183
SwapBData	C2E6	128 memory swap between front/backRAM.	**memory**	20-162
SwapRAM	C2CE	RAM-Expansion Unit memory swap.	**memory**	20-163
TempHideMouse	C2D7	Hide soft-sprites before direct screen access.	**mouse/sprite**	20-174
TestPoint	C13F	Test status of single screen point (on or off?).	**graphics**	20-110
ToBasic	C241	Pass Control to Commodore BASIC.	**utility**	20-219
UnblockProcess	C10F	Unblock a blocked process, allowing it to run again.	**process**	20-191
UnfreezeProcess	C115	Unpause a frozen process timer.	**process**	20-192
UpdateMouse	FE86	Update mouse variables from input device.	**input driver**	20-125
UpdateRecordFile	C295	Update currently open VLIR file without closing.	**disk VLIR**	20-82
UseSystemFont	C14B	Use default system font (BSW 9).	**text**	20-212
VerifyBData	C2E9	128 BackRAM verify.	**memory**	20-164
VerifyRAM	C2D1	RAM-Expansion Unit verify.	**memory**	20-165
VerticalLine	C121	Draw a vertical line in a pattern.	**graphics**	20-111
VerWriteBlock	C223	Disk block verify primitive.	**disk very low-level**	20-69
WriteBlock	C220	Write disk block primitive.	**disk very low-level**	20-70
WriteFile	C1F9	Write chained list of blocks to disk.	**disk mid-level**	20-71
WriteRecord	C28F	Write current VLIR record to disk.	**disk VLIR**	20-83

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
DoDlgBox	C256	Display and begin interaction w/dialog box.	**dialog box**	20-3
RstrFrmDialog	C2BF	Exits from a dialog box.		20-2
ChangeDiskDevice	C2BC	Change disk drive device number.	**disk very low-level**	20-11
DoneWithIO	C25F	Restore system after serial I/O.		20-14
EnterTurbo	C214	Activate disk turbo on current drive.		20-16
ExitTurbo	C232	Deactivate disk turbo on current drive.		20-17
InitForIO	C25C	Prepare system for serial I/O.		20-39
PurgeTurbo	C235	Remove disk turbo from current drive.		20-49
ReadBlock	C21A	Get disk block primitive.		20-53
ReadLink	904B	Read track/sector link.		20-57
VerWriteBlock	C223	Disk block verify primitive.		20-69
WriteBlock	C220	Write disk block primitive.		20-70
GetBlock	C1E4	Read single disk block into memory.	**disk low-level**	20-27
GetBufBlock	903C	Read single disk block into **diskBlkBuf**.		20-28
PutBlock	C1E7	Write single disk block from memory.		20-50
PutBufBlock	903F	Write single disk block from **diskBlkBuf**.		20-51
AllocateBlock	9048	Mark a disk block as in-use.	**disk mid-level**	20-6
BldGDirEntry	C1F3	Build a GEOS directory entry in memory.		20-7
BlkAlloc	C1FC	Allocate space on disk.		20-8
CalcBlksFree	C1DB	Calculate total number of free disk blocks.		20-10
ChkDkGEOS	C1DE	Check if a disk is GEOS format.		20-12
FastDelFile	C244	Quick file delete (requires full track/sector list).		20-18
FindBAMBit	C2AD	Get allocation status of particular disk block.		20-19
FollowChain	C205	Follow chain of sectors, building track/sector table.		20-23
FreeBlock	C2B9	Mark a disk block as not-in-use in BAM.		20-24
FreeFile	C226	Free all blocks associated with a file.		20-25
Get1stDirEntry	9030	Get first directory entry.		20-26
GetDirHead	C247	Read directory header into memory.		20-29
GetFHdrInfo	C229	Read a GEOS file header into **fileHeader**.		20-30
GetFreeDirBlk	C1F6	Find an empty directory slot.		20-34
GetNxtDirEntry	9033	Get directory entry other than first.		20-36
GetOffPageTrSc	9036	Get track and sector of off-page directory.		20-37

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
LdApplic	C21D	Load GEOS application.		20-40
LdDeskAcc	C217	Load GEOS desk accessory.		20-42
LdFile	C211	Load GEOS data file.		20-44
NewDisk	C1E1	Initialize a drive.		20-45
NxtBlkAlloc	C24D	Version of **BlkAlloc** that starts at a specific block.		20-46
PutDirHead	C24A	Write directory header to disk.		20-52
ReadByte	C2B6	Read a File 1 byte at a time.		20-54
ReadFile	C1FF	Read chained list of blocks into memory.		20-55
SetGDirEntry	C1F0	Create and save a new GEOS directory entry.		20-63
SetNextFree	C292	Search for nearby free disk block and allocate it.		20-66
StartAppl	C22F	Warmstart GEOS and start application in memory.		20-68
WriteFile	C1F9	Write chained list of blocks to disk.		20-71
DeleteFile	C238	Delete file.	**disk high-level**	20-13
EnterDeskTop	C22C	Leave application and return to GEOS deskTop.		20-15
FindFile	C20B	Search for a particular file.		20-20
FindFTypes	C23B	Find all files of a particular GEOS type.		20-21
GetFile	C208	Load GEOS file.		20-31
GetPtrCurDkNm	C298	Return pointer to current disk name.		20-38
OpenDisk	C2A1	Open disk in current drive.		20-48
RenameFile	C259	GEOS disk file.		20-58
RstrAppl	C23E	Leave desk accessory and return to calling application.		20-59
SaveFile	C1ED	Save Memory to create a GEOS file.		20-60
SetDevice	C2B0	Establish communication with a new serial device.		20-62
SetGEOSDisk	C1EA	Convert normal CBM disk into GEOS format disk.		20-65
AppendRecord	C289	Insert a new VLIR record after the current record.	**disk VLIR**	20-73
CloseRecordFile	C277	Close/Save currently open VLIR file.		20-74
DeleteRecord	C283	Delete current VLIR record.		20-75
InsertRecord	C286	Insert new VLIR record in front of current record.		20-76
NextRecord	C27A	Make next VLIR the current record.		20-77
OpenRecordFile	C274	Open VLIR file on current disk.		20-78
PointRecord	C280	Make specific VLIR record the current record.		20-79
PreviousRecord	C27D	Make previous VLIR record the current record.		20-80
ReadRecord	C28C	Read current VLIR record into memory.		20-81

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
UpdateRecordFile	`C295`	Update currently open VLIR file without closing.		20-82
WriteRecord	`C28F`	Write current VLIR record to disk.		20-83
BitmapClip	`C2AA`	Display a compacted bitmap, clipping to a sub-window.	**graphics**	20-85
BitmapUp	`C142`	Display a compacted bitmap without clipping.		20-87
i_BitmapUp	`C1AB`	Inline **BitmapUp**.		20-87
BitOtherClip	`C2C5`	**BitmapClip** with data coming from elsewhere (e.g., disk).		20-88
ColorCard	`C2F8`	C128 Get or Set a Color Card. In 40 or 80-column mode.		20-90
ColorRectangle	`C2FB`	C128 Draw a Color rectangle on the 80-column Screen.		20-91
DrawLine	`C130`	Draw, clear, or recover line between two endpoints.		20-92
DrawPoint	`C133`	Draw, clear, or recover a single screen point.		20-93
FrameRectangle	`C127`	Draw a rectangular frame (outline).		20-94
i_FrameRectangle	`C1A2`	Inline **FrameRectangle**.		20-94
GetScanLine	`C13C`	Calculate scanline address.		20-95
GraphicsString	`C136`	Execute a string of graphics commands.		20-96
i_GraphicsString	`C1A8`	Process a graphic command table / inline.		20-96
HorizontalLine	`C118`	Draw a horizontal line in a pattern.		20-98
InvertLine	`C11B`	Invert the pixels on a horizontal screen line.		20-99
ImprintRectangle	`C250`	Imprint rectangular area to background buffer.		20-100
i_ImprintRectangle	`C253`	Inline **ImprintRectangle**.		20-100
InvertRectangle	`C12A`	Invert the pixels in a rectangular screen area.		20-101
NormalizeX	`C2E0`	Normalize C128 X-coordinates for 40/80 modes.		20-102
RecoverLine	`C11E`	Recover horizontal screen line from background buffer.		20-104
Rectangle	`C124`	Draw a filled rectangle.		20-106
i_Rectangle	`C19F`	Inline **Rectangle**.		20-106
RecoverRectangle	`C12D`	Recover rectangular screen area from background buffer.		20-105
i_RecoverRectangle	`C1A5`	Inline **RecoverRectangle**.		20-105
SetColorMode	`C2F5`	Change GEOS 128 80-column Color Mode.		20-107
SetNewMode	`C2DD`	Change GEOS 128 graphics mode (40/80 switch).		20-108
SetPattern	`C139`	Set current fill pattern.		20-109
TestPoint	`C13F`	Test status of single screen point (on or off?).		20-110
VerticalLine	`C121`	Draw a vertical line in a pattern.		20-111
DoIcons	`C15A`	Display and begin interaction with icons.	**icon/menu**	20-113
DoMenu	`C151`	Display and begin interaction with menus.		20-114
DoPreviousMenu	`C190`	Retract sub-menu and reactivate menus up one level.		20-116

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
GotoFirstMenu	C1BD	Retract all sub-menus and reactivate at main level.		20-117
RecoverAllMenus	C157	Recover all menus from background buffer.		20-118
RecoverMenu	C154	Recover single menu from background buffer.		20-119
ReDoMenu	C193	Reactivate menus at the current level.		20-120
InitMouse	FE80	Initialize input device.	**input driver**	20-122
SetMouse	FD09	C128 Reset input device scanning circuitry.		20-123
SlowMouse	FE83	Reset mouse velocity variables.		20-124
UpdateMouse	FE86	Update mouse variables from input device.		20-125
BootGEOS	C000	Reboot GEOS. Requires only 128 bytes at $C000.	**internal**	20-127
FirstInit	C271	Initialize GEOS variables.		20-128
GetSerialNumber	C196	Return GEOS serial number.		20-129
InterruptMain	C100	Main interrupt level processing.		20-130
MainLoop	C1C3	GEOS **MainLoop** processing.		20-131
Panic	C2C2	System-error dialog box.		20-132
Reset	03E4	C128 Reset handler located in BackRAM		20-133
ResetHandle	C003	internal Bootstrap entry point.		20-134
BBMult	C160	Byte by byte (single-precision) unsigned multiply.	**math**	20-136
Bmult	C163	Byte by word unsigned multiply.		20-137
Dabs	C16F	Double-precision signed absolute value.		20-138
Ddec	C175	Double-precision unsigned decrement.		20-139
Ddiv	C169	Double-precision unsigned division.		20-140
DMult	C166	Double-precision unsigned multiply.		20-142
Dnegate	C172	Double-precision signed negation.		20-143
DSdiv	C16C	Double-precision signed division.		20-144
DShiftLeft	C15D	Double-precision left shift (zeros shifted in).		20-145
DShiftRight	C262	Double-precision right shift (zeros shifted in).		20-146
AccessCache	C2EF	C128 Provides a mechanism for disk drivers to cache up to 21 blocks.	**memory**	20-148
ClearRam	C178	Clear memory to $00.		20-149
CmpFString	C26E	Compare two fixed-length strings.		20-150
CmpString	C26B	Compare two null-terminated strings.		20-151
CopyFString	C268	Copy a fixed-length string.		20-152

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
CopyString	C265	Copy a null-terminated string.		20-153
DoBOp	C2EC	C128 Back-RAM memory move/swap/verify primitive.		20-154
DoRAMOp	C2D4	Primitive for communicating with REU (RAM-Expansion Unit).		20-155
FetchRAM	C2CB	Transfer data from RAM-Expansion Unit.		20-156
FillRam	C17B	Fill memory with a particular byte.		20-157
i_FillRam	C1B4	Inline **FillRam**.		20-157
i_MoveData	C1B7	Inline **MoveData**.		20-160
InitRam	C181	Initialize memory areas from table.		20-158
MoveBData	C2E3	C128 BackRAM memory move routine.		20-159
MoveData	C17E	Intelligent memory block move.		20-160
StashRAM	C2C8	Transfer memory to RAM-Expansion Unit.		20-161
SwapBData	C2E6	128 memory swap between front/backRAM.		20-162
SwapRAM	C2CE	Swap memory with an REU memory block.		20-163
VerifyBData	C2E9	128 BackRAM verify.		20-164
VerifyRAM	C2D1	RAM-Expansion Unit verify.		20-165
ClearMouseMode	C19C	Stop input device monitoring.	**mouse/sprite**	20-167
HideOnlyMouse	C2F2	(128) Temporarily remove soft-sprite mouse pointer.		20-168
IsMseInRegion	C2B3	Check if mouse is inside a window.		20-169
MouseOff	C18D	Disable mouse pointer and GEOS mouse tracking.		20-170
MouseUp	C18A	Enable mouse pointer and GEOS mouse tracking.		20-171
SetMsePic	C2DA	Set and preshift new soft-sprite mouse picture.		20-172
StartMouseMode	C14E	Start monitoring input device.		20-173
TempHideMouse	C2D7	Hide soft-sprites before direct screen access.		20-174
GetDimensions	790C	Get CBM printer page dimensions.	**print driver**	20-176
InitForPrint	7900	Initialize printer (once per document).		20-177
PrintASCII	790F	Send ASCII data to printer.		20-178
PrintBuffer	7906	Send graphics data to printer.		20-179
SetNLQ	7915	Begin near-letter quality printing.		20-180
StartASCII	7912	Begin ASCII mode printing.		20-181
StartPrint	7903	Begin graphics mode printing.		20-182
StopPrint	7909	End page of printer output.		20-183
BlockProcess	C10C	Block process from running. Does not freeze timer.	**process**	20-185

Alphabetical List of Routines by Category

Name	Addr	Description	Category	Page
EnableProcess	C109	Make a process runnable immediately.		20-186
FreezeProcess	C112	Pause a process countdown timer.		20-187
InitProcesses	C103	Initialize processes.		20-188
RestartProcess	C106	Unblock, unfreeze, and restart process.		20-189
Sleep	C199	Put current routine to sleep for a specified time.		20-190
UnblockProcess	C10F	Unblock a blocked process, allowing it to run again.		20-191
UnfreezeProcess	C115	Unpause a frozen process timer.		20-192
DisablSprite	C1D5	Disable sprite.	**sprite**	20-194
DrawSprite	C1C6	Define sprite image.		20-195
EnablSprite	C1D2	Enable sprite.		20-196
PosSprite	C1CF	Position sprite.		20-197
GetCharWidth	C1C9	Calculate width of char without style attributes.	**text**	20-199
GetNextChar	C2A7	Get next character from keyboard queue.		20-200
GetRealSize	C1B1	Calculate actual character size with attributes.		20-201
GetString	C1BA	Get string input from user.		20-202
InitTextPrompt	C1C0	Initialize text prompt.		20-204
LoadCharSet	C1CC	Load and begin using a new font.		20-205
PromptOff	C29E	Turn off text prompt.		20-206
PromptOn	C29B	Turn on text prompt.		20-207
PutChar	C145	Display a single character to screen.		20-208
PutDecimal	C184	Format and display an unsigned double-precision nbr.		20-209
PutString	C148	Print string of characters to screen.		20-210
i_PutString	C1AE	Inline **PutString**.		20-210
SmallPutChar	C202	Fast character print routine.		20-211
UseSystemFont	C14B	Use default system font (BSW 9).		20-212
Bell	n/a	1000 Hz Bell sound.	**utility**	20-214
CallRoutine	C1D8	pseudo-subroutine call. $0000 aborts call.		20-215
CRC	C20E	Cyclic Redundancy Check calculation.		20-216
DoInlineReturn	C2A4	Return from inline subroutine.		20-217
GetRandom	C187	Calculate new **random** number.		20-218
ToBasic	C241	Pass Control to Commodore BASIC.		20-219

dialog box

Name	Addr	Description	Page
DoDlgBox	`C256`	Display and begin interaction w/dialog box.	20-3
RstrFrmDialog	`C2BF`	Exits from a dialog box.	20-2

DoDlgBox: (C64, C128) **C256**

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.

When using **DBGETFILES**
r5 BUFFER ptr to buffer to store returned filename (word).
r7L **FILETYPE** GEOS file type to search for (byte). (NULL for all)
r10 PERMNAME ptr to permanent name to search for (word). (NULL for all)

Wheels: When using DBGETFILES **and bit 7 of r7L is set**.
r5 FILTER ptr to Filter Procedure (word).
called once for every file before adding to the list of files.
r7L **FILETYPE** GEOS file type to search for (byte). (NULL for all)
r10 PERMNAME ptr to permanent name to search for (word). (NULL for all)

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 warm start 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**.

Note: Part of the system context saved within **DoDlgBox** is the current stack pointer. Dialog boxes cannot be nested. **DoDlgBox** is not reentrant. That is, a dialog box should never call **DoDlgBox**†.

Note^3: **dispBufferOn** defaults to (ST_WR_FORE | ST_WRGS_FORE) while in a Dialog Box.

Note^3: †*It is possible to overcome the limitations noted here.* ***See Chapter 8 Dialog Box > Removing Limitations****.*

Structure: **DIALOG**.

Example:

See also: **RstrFrmDialog**.

RstrFrmDialog: (C64, C128) **C2BF**

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.

Destroys: assume a, x, y, **r0H-r15**.

Description: **RstrFrmDialog** allows a custom dialog box routine to exit from the 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 times: first for the patterned shadow rectangle and second for the dialog box rectangle. The application may want to special-case these two recovers when recovering.

Note: **RstrFrmDialog** restores the sp register to the 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.

Structure: **DIALOG**.

Example:

See also: **DoDlgBox**, **RecoverRectangle**.

All **disk** routines by name

Name	Description	Category	Page
AllocateBlock	Mark a disk block as in-use.	**mid-level**	20-6
BldGDirEntry	Build a GEOS directory entry in memory.	**mid-level**	20-7
BlkAlloc	Allocate space on disk.	**mid-level**	20-8
CalcBlksFree	Calculate total number of free disk blocks.	**mid-level**	20-10
ChangeDiskDevice	Change disk drive device number.	**very Low level**	20-11
ChkDkGEOS	Check if a disk is GEOS format.	**mid-level**	20-12
DeleteFile	Delete file.	**high-level**	20-13
DoneWithIO	Restore system after serial I/O.	**very Low level**	20-14
EnterDeskTop	Leave application and return to GEOS deskTop.	**high-level**	20-15
EnterTurbo	Activate disk turbo on current drive.	**very Low level**	20-16
ExitTurbo	Deactivate disk turbo on current drive.	**very Low level**	20-17
FastDelFile	Quick file delete (requires full track/sector list).	**mid-level**	20-18
FindBAMBit	Get allocation status of particular disk block.	**mid-level**	20-19
FindFile	Search for a particular file.	**high-level**	20-20
FindFTypes	Find all files of a particular GEOS type.	**high-level**	20-21
FreeBlock	Mark a disk block as not-in-use in BAM.	**mid-level**	20-24
FreeFile	Free all blocks associated with a file.	**mid-level**	20-25
FollowChain	Follow chain of sectors, building track/sector table.	**mid-level**	20-23
Get1stDirEntry	Get first directory entry.	**mid-level**	20-26
GetBlock	Read single disk block into memory.	**low-level**	20-27
GetBufBlock	Read single disk block into **diskBlkBuf**.	**low-level**	20-28
GetDirHead	Read directory header into memory.	**mid-level**	20-29
GetFHdrInfo	Read a GEOS file header into **fileHeader**.	**mid-level**	20-30
GetFile	Load GEOS file.	**high-level**	20-31
GetFreeDirBlk	Find an empty directory slot.	**mid-level**	20-34
GetNxtDirEntry	Get directory entry other than first.	**mid-level**	20-36
GetOffPageTrSc	Get track and sector of off-page directory.	**mid-level**	20-37
GetPtrCurDkNm	Return pointer to current disk name.	**high-level**	20-38
InitForIO	Prepare system for serial I/O.	**very Low level**	20-39
LdApplic	Load GEOS application.	**mid-level**	20-40
LdDeskAcc	Load GEOS desk accessory.	**mid-level**	20-42
LdFile	Load GEOS data file.	**mid-level**	20-44
NewDisk	Initialize a drive.	**mid-level**	20-45
NxtBlkAlloc	Version of **BlkAlloc** that starts at a specific block.	**mid-level**	20-46
OpenDisk	Open disk in current drive.	**high-level**	20-48
PurgeTurbo	Remove disk turbo from current drive.	**very Low level**	20-49
PutBlock	Write single disk block from memory.	**low-level**	20-50
PutBufBlock	Write single disk block from **diskBlkBuf**.	**low-level**	20-51
PutDirHead	Write directory header to disk.	**mid-level**	20-52
ReadBlock	Get disk block primitive.	**very Low level**	20-53
ReadByte	Read a file 1 byte at a time.	**mid-level**	20-54
ReadFile	Read chained list of blocks into memory.	**mid-level**	20-55
ReadLink	Read track/sector link.	**very Low level**	20-57
RenameFile	GEOS disk file.	**high-level**	20-58
RstrAppl	Leave desk accessory and return to calling application.	**high-level**	20-59
SaveFile	Save memory to create a GEOS file.	**high-level**	20-60
SetDevice	Establish communication with a new serial device.	**high-level**	20-62
SetGDirEntry	Create and save a new GEOS directory entry.	**mid-level**	20-63

All **disk** routines by name

Name	Description	Category	Page
SetGEOSDisk	Convert normal CBM disk into GEOS format disk.	**high-level**	20-65
SetNextFree	Search for nearby free disk block and allocate it.	**mid-level**	20-66
StartAppl	Warmstart GEOS and start application in memory.	**mid-level**	20-68
VerWriteBlock	Disk block verify primitive.	**very Low level**	20-69
WriteBlock	Write disk block primitive.	**very Low level**	20-70
WriteFile	Write chained list of blocks to disk.	**mid-level**	20-71

AllocateBlock: (C64, C128) mid-level **9048**

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

Parameters:

r6L	TRACK	— track number of block (byte).
r6H	SECTOR	— sector number of block (byte).

Uses:

curDrive	device number of the active drive.
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:

x	error ($00 = no error). BAD_BAM
r6	unchanged.

Alters:

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

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

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

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 **NewAllocateBlock** subroutine will properly allocate a block on any device, including the 1541.

Example: **NewAllocateBlock**.

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

BldGDirEntry: (C64, C128) mid-level C1F3

Function: Builds a directory entry in memory for a GEOS file using the information in a file header.

Parameters:
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).

Uses: **curDrive** device number of the active drive.

Returns: **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: a, x, y, **r1H**.

Description: Given a GEOS file header, **BldGDirEntry** will build a system specific directory entry suitable for writing to an empty directory slot.

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

Example: **MySetGDirEntry**.

See also: **SetGDirEntry**.

BlkAlloc: (C64, C128) mid-level **C1FC**

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

Parameters:

r2	BYTES	— number of bytes to allocate space for (word). Commodore version can allocate up to 32,258 bytes (127 Commodore blocks).
r6	TSTABLE	— pointer to buffer for building out track and sector table of allocated blocks, usually points to **fileTrScTab** (word).

Uses:

curDrive	device number of the active drive.
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.

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

Alters:

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

Destroys: a, y, **r4-r8**.

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 accommodate the data, **BlkAlloc** returns an INSUFF_SPACE error without allocating any blocks. Otherwise, **BlkAlloc** calls **SetNextFree** to allocate the proper number of unused blocks.

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.

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

Example: **GrabSomeBlocks**.

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

CalcBlksFree: (C64, C128) mid-level **C1DB**

Function: Calculate total number of free blocks on disk.

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

Uses:

curDrive	device number of the active drive.
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:

r4	number of free blocks.
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 INSUFF_SPACE error, midway through the allocation. (This is why it is usually not necessary to check for sufficient space before saving a file or a VLIR record—the higher-level GEOS disk routines handle this checking automatically).

CalcBlksFree 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
jsr 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 Kernal’s
LoadW  r5,#curDirHead     ; point to the directory header
jsr    CalcBlksFree       ; r3 comes back with total number of blocks
                          ; on this device
```

Example: **CheckDiskSpace**.

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

ChangeDiskDevice: (C64, C128) very low-level **C2BC**

Function: Instruct a drive to change its serial device number.

Parameters: a NEWDEVNUM — new device number to give current drive.

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.

Example:

See also: **SetDevice**.

ChkDkGEOS: (C64, C128) mid-level **C1DE**

Function: Check Commodore disk for GEOS format.

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

Returns:

a TRUE/FALSE matching **isGEOS**.

st set according to value in **isGEOS**.

GEOS Disk	z flag=0	bne GEOSDisk
	n flag=1	bmi GEOSDisk
Non-GEOS Disk	z flag=1	beq nonGEOSDisk
	n flag=0	bpl nonGEOSDisk

Alters: **isGEOS** set to TRUE if disk is a GEOS disk, otherwise set to FALSE.

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 possibility 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’s 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**

Example:

```
ProcDisk:
   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    50$               ; if not a GEOS disk, branch
   ;---   code here to handle GEOS disk
   bra    90$               ; jump to exit
50$
   ;---   code here to handle non-GEOS disk
90$
   clc                      ; success exit
   rts
99$
   sec                      ; error exit
   rts
```

See also: **SetGEOSDisk**.

DeleteFile: (C64, C128) high-level **C238**

Function: Delete a GEOS file by deleting 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 (word).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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

Alters:

diskBlkBuf	used for temporary block storage.
dirEntryBuf	deleted directory entry.
fileHeader	temporary storage of index table when deleting a VLIR file.

Written to Disk:

curDirHead	BAM updated to reflect newly freed blocks.
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: a, y, **r1-r2, r4-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.

The directory entry is deleted by setting its OFF_CFILE_TYPE byte to $00.

DeleteFile final step is to call **PutDirHead** to write the changes in the BAM to disk.

Example:

See also: **FreeFile**, **FreeBlock**.

DoneWithIO: (C64, C128) very low-level **C25F**

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**.

Example: **MyPutBlock**.

See also: **InitForIO**.

EnterDeskTop: (C64, C128) high-level **C22C**

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.

Note: **EnterDeskTop** will first search a RAMdisk for a copy of the deskTop to ensure the fastest loading time.

Example:

See also: **RstrAppl**, **GetFile**.

EnterTurbo: (C64, C128) very low-level C214

Function: Activate disk drive turbo mode.

Parameters: none.

Uses:

curDrive	device number of the active drive.
curType	v1.3+: checks disk type because not all use turbo software.

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

Destroys: a, y.

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 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.

Example: **MyPutBlock**.

See also: **WriteBlock**, **ExitTurbo**, **PurgeTurbo**.

ExitTurbo: (C64, C128) very low-level **C232**

Function: Deactivate disk drive turbo mode.

Parameters: none.

Uses: **curDrive** device number of the active 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.

Example:

See also: **EnterTurbo**, **PurgeTurbo**.

FastDelFile: (C64, C128) mid-level C244

Function: Special 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	device number of the active drive.
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).

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

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

Destroys: a, y, **r1**, **r3-r8**.

Description: **FastDelFile** quickly deletes a sequential file by taking advantage 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: See Example **DeleteDirEntry**.

Since **FastDelFile** deletes a block at a time until a track number of $00 is found, it is capable of deleting files with chains larger than 127 blocks, which is the standard GEOS limit imposed by the size of **fileTrScTab**.

Examples: **DeleteDirEntry**, **ReadAndDelete**.

See also: **FreeFile**, **DeleteFile**.

FindBAMBit: (C64, C128) mid-level **C2AD**

Function: Get disk block allocation status.

Parameters:

r6L TRACK — track number of block (byte).
r6H SECTOR — sector number of block (byte).

Uses:

curDrive	device number of the active drive.
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, x, **r7H**, and **r8H**. For an example of using these extra 1541 return values, refer to **AllocateBlock**.

Examples:

```
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: **AllocateBlock**, **FreeBlock**, **GetDirHead**. **PutDirHead**.

FindFile: (C64, C128) high-level **C20B**

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 (16) bytes (not counting null terminator). (word).
Uses:	**curDrive**	device number of the active drive.
	curType	GEOS 64 v1.3 and later for detecting REU shadowing.
Returns:	x	error ($00 = no error). FILE_NOT_FOUND
	r1	track/sector of directory block containing entry.
	r5	Pointer to directory entry within **diskBlkBuf**.
Alters:	**dirEntryBuf**	directory entry of file if found.
	diskBlkBuf	contains directory block where FILENAME 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.

Since GEOS 2.0 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.

Example: **LoadBASIC**.

See also: **Get1stDirEntry**, **GetNxtDirEntry**, **FindFTypes**.

FindFTypes: (C64, C128) high-level **C23B**

Function: Builds a list of files of a particular GEOS type from the current directory.

Used By: **DBGETFILES** dialog box routine.

Parameters:

r6 BUFFER — pointer to buffer for building-out file list; allow ENTRY_SIZE+1 bytes for each entry in the list (word).
r7L **FILETYPE** — GEOS file type to search for (byte).
r7H MAXFILES — maximum number of filenames to return, usually used to prevent overwriting buffer (byte).
r10 PERMNAME — pointer to permanent name string to match or $0000 to ignore permanent name string (word).

Uses:

curDrive device number of the active drive.
curType GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x error ($00 = no error).
r7H decremented once for each file name.

Alters: **diskBlkBuf** used as temporary buffer for directory blocks.

Destroys: 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 accommodate *MAXFILES* filenames of ENTRY_SIZE+1 bytes each.

FindFTypes first clears enough of the area at *BUFFER* to hold *MAXFILES* filenames then calls **Get1stDirEntry** 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 Vx.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.

Note: Since 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.

Example:

See also: **FindFile**, **Get1stDirEntry**, **GetNxtDirEntry**.

FollowChain: (C64, C128) mid-level **C205**

Function: Follow a chain of Commodore disk blocks, building out a track/sector table.

Parameters:

r1L	START_TR	— track number of starting block (byte).
r1H	START_SC	— 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	device number of the active drive.
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**.

Examples:

```
LoadB   r1L,#START_TR         ; start track
LoadB   r1H,#START_SC         ; and sector
LoadW   r3,#fileTrScTab       ; buffer for table
jsr     FollowChain           ; get allocation status
txa                           ; set status flags
bne     HandleError           ; branch if error
```

See also: **FastDelFile**, **WriteFile**, **ReadLink**.

FreeBlock: (C64, C128) mid-level **C2B9**

Function: Free an allocated disk block.

Parameters:	**r6L**	TRACK	— track number of block to free (byte).
	r6H	SECTOR	— sector number of block to free (byte).

Uses:	**curDrive**	device number of the active drive.
	curDirHead	must contain the current directory header.
	dir2Head†	(BAM for 1571 and 1581 drives only).
	dir3Head†	(BAM for 1581 drive only).

Returns:	x	error ($00 = no error).	
		BAD_BAM	if block already free.
	r6L, **r6H**		unchanged.

Alters:	**curDirHead**	BAM updated to reflect newly allocated block.
	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: 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.

Note: **FreeBlock** was not added to the 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. (See Example: **MyFreeBlock**).

Example: **MyFreeBlock**.

See also: **FreeFile**, **AllocateBlock**.

FreeFile: (C64, C128) mid-level **C226**

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** (word).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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

Alters:

diskBlkBuf	used for temporary block storage.
curDirHead	BAM updated to reflect newly allocated block.
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.

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

Destroys: a, y, **r1-r2, r4-r8**.

Description: Given a valid directory entry, **FreeFile** will delete (free) all blocks associated 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**.

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 = 30 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.

Since **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 **fileTrScTab**.

Example:

See also: **DeleteFile**, **FreeBlock**.

Get1stDirEntry: (C64, C128) mid-level **9030**

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	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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**.

Since 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 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    #$13
    bcc    10$                  ; branch < v1.3
    jmp    Get1stDirEntry       ; direct call
10$
    jmp    o_Get1stDirEntry     ; go through jump table
```

Example:

See also: **GetNxtDirEntry**, **FindFTypes**.

GetBlock: (C64, C128) low-level C1E4

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).
r1L	TRACK	— track number (byte).
r1H	SECTOR	— sector number on track (byte).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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.

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.

Note[3]: The original Hitchhikers Guide to GEOS stated the 1581 driver had a bug that destroyed **r1** and **r4**. GEOS 64 Version 1.5 (First 64 version with 1581 support) and above do not have this problem. GEOS 128 1.3 with CONFIGURE 1.4 (Earliest version locatable at this time) does not have this problem. It is possible a CONFIGURE 1.3 on the 128 exists that does have this problem. This bug warning can be safely ignored.

Example:

See also: **PutBlock, ReadBlock, GetBufBlock**.

GetBufBlock: (C64, C128) low-level **903C**

Function: General purpose routine to get a block from the current disk into **diskBlkBuf**.

Parameters:
r1L TRACK — track number (byte).
r1H SECTOR — sector number on track (byte).

Uses:
curDrive device number of the active drive.
curType GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:
x error ($00 = no error).
r1 unchanged.
r4 address of **diskBlkBuf**.

Alters:
diskBlkBuf contains block read from disk.

Destroys: a, y.

Description: **GetBufBlock** loads **r4** with the address of **diskBlkBuf** and calls **GetBlock** to read a block from the disk into **diskBlkBuf**. **GetBufBlock** is useful for setting **r4** with the common location of **diskBlkBuf** when reading a single block.

See **GetBlock** for more information.

Example:

```
Function:    Delete file from directory page.
Uses:        r1L    TRACK  - track number of directory page.
             r1H    SECTOR - sector number of directory page.
             r5     pointer to current directory entry.
Uses:        diskBlkBuf   contains page from the directory track pointed to by r1.
Returns:     x      error ($00 = no error).
             r1,r5  unchanged.
             r4     address of diskBlkBuf.
Destroys:    a, y, r0,r2-r3,r6-r9.
__________________________________

DelTemp:
      PushW  r5                 ; save r5 and r1
      PushW  r1
      jsr    GetName            ; extract name from directory entry
                                ; returns r0=address of null terminated name
      jsr    DeleteFile         ; delete the file
      bxne   HandleError        ; on error: branch to handler
      PopW   r1                 ; restore r1 and r5
      PopW   r5
                                ; save 8 bytes by not doing LoadW r4,#diskBlkBuf
      jmp    GetBufBlock        ; read updated directory page into diskBlkBuf
```

See also: **GetBlock**, **PutBlock**, **PutBufBlock**, **WriteBlock**, **BlkAlloc**.

GetDirHead: (C64, C128) mid-level **C247**

Function: Read directory header from disk. GEOS also reads in the BAM.

Parameters: none.

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x	error ($00 = no error).
r4	pointer to **curDirHead**.

Alters:

curDirHead	contains 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.

Destroys: a, y, **r1**.

Description: **GetDirHead** reads the full directory header (256 bytes) into the buffer at **curDirHead**. This block also includes the BAM (block allocation map) for the entire disk.

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.

Example:

See also: **PutDirHead**.

GetFHdrInfo: (C64, C128) mid-level **C229**

Function: Loads the GEOS file header for a particular directory entry.

Parameters: **r9** **DIRENTRY**— pointer to directory entry of file, usually points to **dirEntryBuf** (word).

Uses:

curDrive	device number of the active drive.
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.
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).

Alters:

fileHeader	contains 256-byte GEOS file header.
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**, **Get1stDirEntry** and **GetNxtDirEntry**. **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: (C64, C128) high-level **C208**

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 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.
  - 1 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).

When loading a desk accessory:

r10L RECVR_OPTS — no longer used; set to $00 (see below for explanation) (byte).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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:

r0L as originally passed to **GetFile**.

r2 points to **dataDiskName**. (**r2** as originally passed to **GetFile** is used to copy the name.)

r3 points to **dataFileName**. (**r3** as originally passed to **GetFile** is used to copy the name.)

dataDiskName contains name of data disk if bit 7 or 6 of **r0L** is set.

dataFileName contains name of data file if bit 7 or 6 of **r0L** 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 application's 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	$01	Load at address: load application at the address passed in **r7** as opposed to the address in the file header.
ST_LD_DATA	$80	Load data file: application is being passed the name of a data file to load.
ST_PR_DATA	$40	Print data file: application is being passed the name of a data file to print.

Note[3]: C128: When b4 in **sysRAMFlg** is set, the print driver header block and the print driver are cached in an internal Kernal cache.

The following example will let **GetFile** load the print driver from this cache instead of from disk. If the driver is not available in cache (**sysRAMFlg** (b4=0)) then the driver will load normally from the current disk.

Example:
```
bbeq        PrntFilename,99$    ; If printer name is not set, then error out
LoadW       r6,#PrntFilename    ; r6 must point to the PrntFilename variable
LoadB       r0L,#0              ; All load options must be reset.
jsr         GetFile             ; If b4 is set in sysRAMFlg , GetFile loads driver from
                                ; cache, else it loads it from disk.
```

C128 : If the flags in the file header block do not allow running in the currently active **graphMode**; an INCOMPATIBLE error is returned.

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.
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.

The application should always set **r10L** 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).

See **LdDeskAcc Note** for more information.

Example:

See also: **LdFile**, **LdDeskAcc**, **LdApplic**.

GetFreeDirBlk: (C64, C128) mid-level **C1F6**

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 (byte).

Uses:

curDrive	device number of the active drive.
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.

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

Returns:

x	error ($00 = no error). FULL_DIRECTORY.
r10L	page number of empty directory slot.
r1	block (track/sector) number of directory block in **diskBlkBuf**.
y	index to empty directory slot in **diskBlkBuf**.

Alters:

curDirHead	contains directory header.
dir2Head†	(BAM for 1571 and 1581 drives only).
dir3Head†	(BAM for 1581 drive only).

Destroys: a, **r0**, **r3**, **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.

Since GEOS 2.0 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.

Note: **GetFreeDirBlk** does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

Example: **MySetGDirEntry**.

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

GetNxtDirEntry: (C64, C128) mid-level **9033**

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	device number of the active drive.
diskBlkBuf	must be unaltered from previous call to **Get1stDirEntry** or **GetNxtDirEntry**.
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**.
y	non-zero if end of directory reached.

Alters: **diskBlkBuf** directory block.

Destroys: a, **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**.

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 GEOS v1.2 and later:

```
;****************************************************
; MyGetNxtDirEntry — Call 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.

_GetNxtDirEntry = $CA10              ; exact entry point

MyGetNxtDirEntry:
   CmpB   version,#$13               ; check version number
   bcc    10$                        ; branch < v1.3
   jmp    GetNxtDirEntry             ; go through jump table
10$
   jmp    _GetNxtDirEntry            ; direct call
```

Example:

See also: **GetlstDirEntry**, **FindFTypes**.

GetOffPageTrSc: (C64, C128) mid-level **9036**

Function: Get track and sector of off-page directory.

Parameters: none.

Uses:

curDrive	device number of the active drive.
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**.

Destroys: a, y, **r5**.

Description: 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.

Example:

```
;--- Put off-page block into diskBlkBuf
    jsr    GetOffPageTrSc       ; get off-page directory block
    bxne   99$                  ; exit on error
    tya                         ; check for GEOS disk
    tax                         ; put in x in case error
    bne    99$                  ;
    LoadW  r4,#diskBlkBuf       ; get off-page block
    jsr    GetBlock             ; return with error status in x
99$
    rts
```

See also: **PutDirHead**.

GetPtrCurDkNm: (C64, C128) high-level **C298**

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** device number of the active drive.

Returns: x error ($00 = no error).
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 **DrXCurDkNm** 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 **DrXCurDkNm** 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 compatibility with future versions of GEOS that might support more drives.

C64: Versions of GEOS before v1.3 only support 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. **GetPtrCurDkNm** 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: Commodore disk names are always a fixed-length 16-character string. If the name is less than 16 characters, the string is padded with $**A0**.

Example:

See also:

InitForIO: (C64, C128) very low-level **C25C**

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 l MHz, switches in the ROM and I/O banks if necessary, and performs any 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**.

Example: **MyPutBlock**.

See also: **DoneWithIO**, **SetDevice**.

LdApplic: (C64, C128) mid-level **C21D**

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** (word).

r0L LOAD_OPT: (byte).

- 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.
  - 1 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 of *LOAD_OPT* is set: pointer to name of the data file (word).

Uses:

curDrive device number of the active drive.
curType GEOS 64 v1.3 and later for detecting REU shadowing.

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 **r0L** is set.
dataFileName contains name of data file if bit 6 of **r0L** 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 to 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**.

C128 : If the flags in the file header block do not allow running in the currently active **graphMode**; an INCOMPATIBLE error is returned.

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 specific load address. When **LdApplic** returns to the caller, it does so before calling **StartAppl** to warmstart GEOS.

Example:

See also: **GetFile**, **LdDeskAcc**, **StartAppl**.

LdDeskAcc: (C64, C128) mid-level **C217**

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 — always set to $00 (see note below for explanation) (byte).

Uses: **curDrive** device number of the active drive.
curType GEOS 64 v1.3 and later for detecting REU shadowing.

Returns: *returns when desk accessory exits with a call to* **RstrAppl**.
x error ($00 = no error).
BFR_OVERFLOW

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 to 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 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 backRAM. 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**.

C64 : GEOS versions 1.3 and above have a 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.

C128 : If the flags in the file header block do not allow running in the currently active **graphMode**; an INCOMPATIBLE error is returned.

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.
bit 6:	1	force desk accessory to save color memory and restore it on return to application.
	0	not necessary for desk accessory to color memory.

Note: 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 **r10L** to $00 before calling desk accessories just to enforce this standard!).

The application should always set **r10L** 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).

Example:

See also: **GetFile**, **LdApplic**, **RstrAppl**, **RstrFrmDialog**.

LdFile: (C64, C128) mid-level **C211**

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** (word).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x	error ($00 = no error).
r7	pointer to last byte read in plus one.

Alters:

fileHeader	contains 256-byte GEOS file header.
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.

Destroys: Not listed in the source material. **LdFile** is listed as being in an unusable state so this is to be expected. Assume the same as **GetFile**: a, x, y, **r0-r10**.

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 go indirectly through **GetFile**.

All versions of **LdFile** to date under GEOS are unusable because the load variables 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.

Note[3]: There is a situation where **LdFile** is the only option. If you need to load an application without it executing automatically, **LdFile** will perform this function perfectly fine. If you attempt to do this with **GetFile** using "**r0L** LOAD_OPT: 1; load at address in **r7**; application will not be started automatically". The file will be loaded; then instead of returning to the caller; **GetFile** does a jmp **EnterDeskTop** so that the application never regains control.

Note[3]: The **GetFile** routine **Destroys**: a, x, y, **r0-r10**. **LdFile** is the core of **GetFile** so this is a safe assumption for **LdFile** as well.

Example:

See also: **GetFile**, **LdApplic**, **LdDeskAcc**.

NewDisk: (C64, C128) mid-level **C1E1**

Function: Tell the turbo software that a new disk has been inserted into the drive.

Parameters:
r1L TRACK — used to set drive head position (byte).
r1H SECTOR — used to set drive head position (byte).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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 it is first loaded. A call to **NewDisk** during the application's initialization will stop the disk.

Note[1]: **NewDisk** also positions the head over the *TRACK* and *SECTOR*.

Calls[2]: **EnterTurbo**, **InitForIO**, **DoneWithIO**.

Example:

See also: **OpenDisk**, **SetDevice**.

NxtBlkAlloc: (C64, C128) mid-level **C24D**

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. Can allocate up to 32,258 bytes (127 blocks) (word).
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 (word). *usually, a position within **fileTrScTab**.*

Uses:

curDrive	device number of the active drive.
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.

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

Alters:

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

Destroys: a, y, **r4-r8**.

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**.

Use **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.

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

Example:

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

OpenDisk: (C64, C128) high-level **C2A1**

Function: Open the disk in the current drive.

Parameters: none.

Uses:

curDrive	device number of the active drive.
driveType	type of drive to open (for shadowing information).

Calls[2]: **NewDisk**, **GetDirHead**, **ChkDkGEOS**, **GetPtrCurDkNm**.

Returns:

x	error ($00 = no error).
r5	pointer to disk name buffer as returned from **GetPtrCurDkNm**. This is a pointer to one of the **DrXCurDkNm** arrays.

Alters:

Dr*x*CurDkNm	current disk name array contains disk name.
curDirHead	current directory header.
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).

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

Destroys: a, y, **r0-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 if necessary (use **PutDirHead**) before accessing another disk.

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**.

Note: This routine calls **GetDirHead** which loads in the BAM from disk. **PutDirHead** should be called prior to this routine if the BAM has been modified by freeing or allocating blocks.

Example: **KeyTrap**.

See also: **SetDevice**, **NewDisk**.

PurgeTurbo: (C64, C128) very low-level **C235**

Function:	Completely deactivate and remove disk drive turbo code from current drive, returning to standard Commodore DOS mode.
Parameters:	none.
Uses:	**curDrive** device number of the active 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.
Example:	

See also: **EnterTurbo**, **ExitTurbo**.

PutBlock: (C64, C128) low-level **C1E7**

Function: General purpose routine to write a block to disk with verify.

Parameters:

r4	BUFFER	— address of buffer to get block from (word).
r1L	TRACK	— valid track number (byte).
r1H	SECTOR	— valid sector on track (byte).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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.

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.

Note[3]: **PutBlock** does no boundary check on the buffer. If the buffer is less than BLOCKSIZE ($100) bytes, **PutBlock** will write the buffer and the memory contents that are after the buffer. This normally will not cause any problems as the size of data in the data block is stored in offset 1 of the block when the block is not full.

Example:

See also: **GetBlock**, **WriteBlock**, **BlkAlloc**.

PutBufBlock: (C64, C128) low-level **903F**

Function: General purpose routine to write a block from **diskBlkBuf** to disk with verify.

Parameters:

r1L	TRACK	— valid track number (byte).
r1H	SECTOR	— valid sector on track (byte).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x	error ($00 = no error).
r1	unchanged.
r4	address of **diskBlkBuf**.

Destroys: a, y.

Description: **PutBufBlock** loads **r4** with the address of **diskBlkBuf** and calls **PutBlock** to write a block from **diskBlkBuf** to the disk. **PutBufBlock** is useful for setting **r4** with the common location of **diskBlkBuf** when writing a single block.

See **PutBlock** for more information.

Example:

See also: **PutBlock, GetBlock**, **GetBufBlock**, **WriteBlock**, **BlkAlloc**.

PutDirHead: (C64, C128) mid-level **C24A**

Function: Write directory header to disk. GEOS also writes out the BAM.

Parameters: none.

Uses:

curDrive	device number of the active drive.
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).

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

Returns:

x	error ($00 = no error).
r4	pointer to **curDirHead**.

Destroys: a, y, **r1**.

Description: **PutDirHead** writes the directory header to disk from the buffer at **curDirHead**. GEOS writes out the full directory header block, including the BAM (block allocation map).

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 routine's, such as **BlkAlloc**, only update the BAM in memory (for speed and ease of error recovery). When a new file is written to 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. An application that manipulates 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.

Example:

See also: **GetDirHead**.

ReadBlock: (C64, C128) very low-level **C21A**

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**	device number of the active 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.

Example: **MyGetBlock**.

See also: **GetBlock**, **WriteBlock**, **VerWriteBlock**.

ReadByte: (C64, C128) mid-level **C2B6**

Function: Special version of **ReadFile** that allows reading a chained list of blocks a byte at a time.

Parameters: *on initial call only:*

r1	START_TRSC	—	track/sector of first data block (word).
r4	BLOCKBUF	—	pointer to temporary buffer of BLOCKSIZE bytes for use by **ReadByte**, usually a pointer to **diskBlkBuf** (word).
r5L	EOD	—	end of data (EOD) index for *BLOCKBUF*. $00 (byte). must be set to $00.
r5H	NDX	—	index to current byte in *BLOCKBUF* (byte). must be set to $00.

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

a	byte returned
x	error ($00 = no error). BFR_OVERFLOW

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*.

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**

Note: Reading a chain a byte at a time 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.

Example:

```
        MoveW  startTS,r1                 ; set initial track and sector
        LoadW  r4,#diskBlkBuf             ; set location of work buffer
        LdNull r5                         ; initialize r5L and r5H data indexes
10$
        jsr    ReadByte                   ; read next byte. byte is returned in a-register
        cpx    #00                        ; if x != 0 then exit
        bne    $98                        ; (normal exit is x = #BFR_OVERFLOW)
        ;---   process byte in a-register here (must preserve r1, r4 and r5)
        bra 10$                           ; loop back to get next byte
```

See also: **OpenDisk**, **SetDevice**.

ReadFile: (C64, C128) mid-level **C1FF**

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. Can read up to 32,258 bytes (127 blocks) (word).
r1 START_TRSC — track/sector of first data block (word).

Uses:
curDrive device number of the active drive.
curType GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:
x error ($00 = no error).
BFR_OVERFLOW
r7 pointer to last byte read into BUFFER plus one.
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.
r2 unused bytes remaining in *BUFFER*.
r5L byte index into **fileTrScTab** of last entry (last entry = **fileTrScTab** plus value in **r5**).

Alters:
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.
diskBlkBuf Each block is read into **diskBlkBuf** before copying to *BUFFER*.

Destroys: y, (**r1**), **r3-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**.

Note: 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 BLKDATSIZE (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 * BLKDATSIZE (254) = 32,258 bytes of data.

Example:

See also: **GetFile**, **WriteFile**, **ReadRecord**.

ReadLink: (C64, C128) very low-level **904B**

Function: Read link (first two bytes) from a disk block.

Parameters:

r1L	TRACK	— track number (byte).
r1H	SECTOR	— sector on track (byte).
r4	BUFFER	— address of buffer of at least BLOCKSIZE bytes, usually points to **diskBlkBuf** (word).

Uses: **curDrive** device number of the active drive.

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

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 is 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.

Note: Disk drives that do not offer any speed increase through **ReadLink** will simply perform a **ReadBlock**.

Important: Does not work in C64 1541/RAM 1541 drivers prior to GEOS 1.5/CONFIGURE 1.6.
Use **ReadBlock** instead with 1541 drives if C64 OS version is less than 1.5.

Example:

See also: **ReadBlock**, **FollowChain**.

RenameFile: (C64, C128) high-level **C259**

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 (word).
r0 NEWNAME — pointer to new null-terminated name (word).

Uses: **curDrive** device number of the active drive.
driveType type of drive to open (for shadowing information),

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

Alters: **diskBlkBuf** used for temporary block storage.
dirEntryBuf old directory entry.
curDirHead BAM updated to reflect newly freed blocks.
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: a, y, **r4-r6**.

Description: **RenameFile** searches the current directory for *OLDNAME* and changes the name string in the directory entry to *NEWNAME*.

RenameFile first calls **FindFile** to get the directory entry and ensure the *OLDNAME* 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.

When using **Get1stDirEntry** and **GetNxtDirEntry** 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 **RenameFile** the pointer to that buffer. Otherwise, when **FindFile** uses **diskBlkBuf** it will corrupt the file name.

Note^3: This Routine calls **FindFile** which loads the BAM in from disk. It is important to close all VLIR files and update the BAM if necessary (use **PutDirHead**) before using **RenameFile**.

Example:

See also: **FreeFile**, **FreeBlock**.

RstrAppl: (C64, C128) high-level **C23E**

Function: Standard desk accessory return to application.

Parameters: none.

Uses: **curDrive** device number of the active drive.

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 the "SWAP FILE".

Note: If a disk error occurs when reading in the "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**.

SaveFile: (C64, C128) high-level **C1ED**

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	**FILEHDR**	— pointer to GEOS file header for file (word).
r10L	DIRPAGE	— directory page to begin searching for an empty directory slot (byte).

Uses:

curDrive	device number of the active drive.
year, **month**, **day**, **hours**, **minutes**	for date-stamping file.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.
interleave	desired physical sector **interleave** (usually 8).

Returns:

x	error ($00 = no error).
r1	track and sector of last block written.
r9	unchanged.
r6	pointer to **fileTrScTab**.

Alters:

dirEntryBuf	contains newly-built directory entry.
diskBlkBuf	contains contents of last block written.
fileHeader	builds 256-byte GEOS file header.
fileTrScTab	$00-$01 contain T/S of file header. end of table is marked with track=$00.
curDirHead	BAM updated to reflect newly allocated block.
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: a, y, **r1-r8**.

Description: **SaveFile** is the most general-purpose 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 *FILEHDR* 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 *FILEHDR* 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 file, then calls **WriteFile** to put the data into it. After the file is written, the BAM is written to disk using **PutDirHead**.

Note: **SaveFile** sets the byte at **fileHeader**+O_GHINFO_TXT to NULL.

Note^3: If the start and end addresses are equal, no data blocks are written causing an empty SEQUENTIAL file's directory entry to have a start T/S of 00/FF. This is not a normal valid state for a SEQUENTIAL file and should have at least one block added to it.

Note^3: **Required offsets into GEOS File Header to set:**

Offset	**Constant**	**Size**	**Description**
$00		word	Pointer to filename
$44	O_GHCMDR_TYPE	byte	DOS file type
$45	O_GHGEOS_TYPE	byte	GEOS file type
$46	O_GHSTR_TYPE	byte	GEOS file structure type (SEQUENTIAL or VLIR)
$47	O_GHST_ADDR	word	Memory to save; start address ***note**: (Set to $0000 when creating a VLIR)*
$49	O_GHEND_ADDR	word	Memory to save; end address ***note**: (Set to -1 or $FFFF when creating a VLIR)*

Example:

See also: **GetFile**, **OpenRecordFile**.

SetDevice: (C64, C128) high-level **C2B0**

Function: Establish communication with a new peripheral.

Parameters: a DEVNUM — 8,9,10,11 (DRIVE_A through DRIVE_D) for disk drives, PRINTER for serial printer, or any other valid serial device bus address.

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 active 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 by using a *DEVNUM* value of *PRINTER*.

Each I/O device has an associated device number that distinguishes its I/O from 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**.

SetDevice is designed to switch between serial bus devices, *DEVNUM* reflects the architecture of 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.

GEOS up through v1.2 supports two disk devices, DRIVE_A and DRIVE_B. GEOS v1.3 and later supports up to four disk devices, DRIVE_A through DRIVE_D. DeskTop only supports 3 devices.

Note: **SetDevice** calls **ExitTurbo** so that the old device is no longer actively sensing the serial bus, then installs the new device driver 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), GEOS must switch the device drivers, making the driver for the selected device active. GEOS stores inactive device drivers in the Commodore 128 backRAM and in special system areas in an REU. GEOS applications must use **SetDevice** to change the active device. An application should never directly modify **curDrive** or **curDevice**.

Example: **KeyTrap**.

See also: **OpenDisk**, **ChangeDiskDevice**.

SetGDirEntry: (C64, C128) mid-level **C1F0**

Function: Builds a system specific directory entry from a GEOS file header, date-stamps it, and writes it out to the current directory.

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 (byte).
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).

Uses:

curDrive	device number of the active drive.
year, **month**, **day**, **hour**, **minutes**	for date-stamping file.
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.

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

Returns:

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

Alters:

dirEntryBuf	contains newly-built directory entry.
diskBlkBuf	used for temporary storage of the directory block.
curDirHead[2]	updated Directory Header.

Destroys: a, y, **r1**, **r3-r5**, **r7-r8**.

Description: **SetGDirEntry** calls **BldGDirEntry** to build a system specific directory entry from 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.

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

Note[3]: **Required offsets into GEOS File Header to set**:

Offset	**Constant**	**Size**	**Description**
$00		word	Pointer to filename
$44	O_GHCMDR_TYPE	byte	DOS file type
$45	O_GHGEOS_TYPE	byte	GEOS file type
$46	O_GHSTR_TYPE	byte	GEOS file structure type (SEQUENTIAL or VLIR)

Example:

See also: **GetFile**, **OpenRecordFile**.

SetGEOSDisk: (C64, C128) high-level **C1EA**

Function: Convert Commodore disk to GEOS format.

Calls^2: **GetDirHead**, **CalcBlksFree**, **SetNextFree**, **PutDirHead**.

Parameters: none.

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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

Alters:

curDirHead	directory header is read from disk.
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: a, y, **r0L**, **r1**, **r4-r5**.

Description: **SetGEOSDisk** converts a standard Commodore disk into 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 see if there is a block available for the off-page directory (if there isn't, an **INSUFF_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.

Example:

See also: **ChkDkGEOS**.

SetNextFree: (C64, C128) mid-level **C292**

Function: Search for a nearby free block and allocate it.

Parameters:

r3L	START_TR — start searching from this track (byte).
r3H	START_SC — start searching from this sector (byte).

Uses:

curDrive	device number of the active drive.
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.

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

Returns:

x	error ($00 = no error). INSUFF_SPACE
r3L	track of allocated block.
r3H	sector of allocated block.

Alters:

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

Destroys: a, y, **r6-r7**, **r8H**.

Description: Given the current track/sector as passed in **r3L/r3H**, **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 if it is unused. If the block is used, **SetNextFree** picks another nearby sector (jumping to another track if necessary) and 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 **INSUFF_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, **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 Append2Record 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 Append2Record routine would pass 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 **INSUFF_SPACE** error is returned. Since **SetNextFree** only searched from the current block to the end of the disk, the possibility exists that a free block lies somewhere on a previous, still unchecked disk area. The following alternative to **SetNextFree** will circumvent this problem: (See Example: **MySetNextFree)**.

Note: **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	$28	(one track)
1541	DIR_TRACK	$12	(one track)
1571	DIR_TRACK DIR_TRACK+N_TRACKS	$12 $12+$23	(two tracks)

SetNextFree does not automatically write out the BAM. See **PutDirHead** for more information on writing out the BAM.

Example: **MySetNextFree**.

See also: **GetFile**, **OpenRecordFile**.

StartAppl: (C64, C128) mid-level **C22F**

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: (byte).

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.

1 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 **DrXCurDkNm** buffers (word).

r3 DATA_FILE — only valid if bit 7 of *OPTIONS* 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 **r0L** is set.

dataFileName contains name of data file if bit 6 of **r0L** 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, initializing tables, clearing the processor stack, and executing the application's initialization code with a jsr from **MainLoop**.

Example:

See also: **LdApplic**, **GetFile**.

VerWriteBlock: (C64, C128) very low-level **C223**

Function: Very low-level verify block on disk.

Parameters:
r1L TRACK — 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 device number of the active 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.

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.

Example: **MyPutBlock**.

See also: **WriteBlock**, **PutBlock**.

WriteBlock: (C64, C128) very low-level **C220**

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	device number of the active 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 a 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.

Example: **MyPutBlock**.

See also: **PutBlock**, **ReadBlock**, **VerWriteBlock**.

WriteFile: (C64, C128) mid-level **C1F9**

Function: Write data to a chained list of disk blocks.

Parameters: **r7** DATA — pointer to start of data (word).
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 (word).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

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

Destroys: 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.

Note: **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 BLKDATSIZE (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 **diskBlkBuf**+0 and **diskBlkBuf**+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.

Example:

See also: **SaveFile**, **WriteRecord**, **ReadFile**.

VLIR

AppendRecord	`C289`	Insert a new VLIR record after the current record.	20-73
CloseRecordFile	`C277`	Close/Save currently open VLIR file.	20-74
DeleteRecord	`C283`	Delete current VLIR record.	20-75
InsertRecord	`C286`	Insert new VLIR record in front of current record.	20-76
NextRecord	`C27A`	Make next VLIR the current record.	20-77
OpenRecordFile	`C274`	Open VLIR file on current disk.	20-78
PointRecord	`C280`	Make specific VLIR record the current record.	20-79
PreviousRecord	`C27D`	Make previous VLIR record the current record.	20-80
ReadRecord	`C28C`	Read current VLIR record into memory.	20-81
UpdateRecordFile	`C295`	Update currently open VLIR file without closing.	20-82
WriteRecord	`C28F`	Write current VLIR record to disk.	20-83

AppendRecord: (C64, C128) **C289**

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	device number of the active drive.
fileWritten[†]	if **FALSE**, assumes record just opened (or updated) and reads BAM into memory.
curRecord	current record number.
fileHeader	VLIR index table.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.
curDirHead	BAM updated to reflect newly allocated block.
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:

x	error ($00 = no error). OUT_OF_REC0RDS

Alters:

curRecord	new record number.
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.
curDirHead	directory header read in if **fileWritten** is **FALSE** on call.

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 up to **MAX_VLIR_RECS** records (127). 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 ($00 $FF). When a VLIR file is first created by **SaveFile**, all records are marked as unused ($00 $00). 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.

Note: An empty record is marked with $00 $FF in the VLIR index table (stored in the buffer at **fileHeader**). An unused record is marked with $00 $00. Use **PointRecord** to check the status of a particular record (unused, empty, or filled).

Example: **SaveRecord**.

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

CloseRecordFile: (C64, C128) **C277**

Function: Close the current VLIR file (updating it in the process) so that another may be opened.

Parameters: none.

Uses:

curDrive	device number of the active drive.
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.

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).

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

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

Alters:

fileWritten†	set to **FALSE** to indicate the file hasn't been altered since last updated.
diskBlkBuf	used for temporary storage of the directory block.

***Note**: When making manual changes to the VLIR, setting **fileWritten** to **TRUE** will cause **CloseRecordFile** to write the changes to disk.*

Destroys: a, y, **r1**, **r4**, **r5**.

Description: **CloseRecordFile** first calls **UpdateRecordFile** then closes the VLIR file so that another may be opened.

Because 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: **SaveRecord**.

See also: **OpenRecordFile**, **UpdateRecordFile**.

DeleteRecord: (C64, C128) **C283**

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	device number of the active drive.
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM into memory.
curRecord	current record number.
fileHeader	VLIR index table stored in this buffer.
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).

†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	record marked as empty ($00 $FF).
fileSize	decremented to reflect any deleted record blocks.
curDirHead	current directory header/BAM modified to free blocks.
dir2Head†	(BAM for 1571 and 1581 drives only).
dir3Head†	(BAM for 1581 drive only).

Destroys: 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 and VLIR file information on the disk. Call **CloseRecordFile** or **UpdateRecordFile** to update the file when done modifying.

Example:

See also: **AppendRecord**, **InsertRecord**.

InsertRecord: (C64, C128) **C286**

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	device number of the active drive.
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM into memory.
curRecord	current record number.
fileHeader	VLIR index table.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.
curDirHead	BAM updated to reflect newly allocated block.
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:

x	error ($00 = no error) OUT_OF_RECORDS.

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.
usedRecords	number of records in file that are currently in 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** (127) 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 ($00, $FF) .

InsertRecord does not update the VLIR file information on disk. Call **CloseRecordFile** or **UpdateRecordFile** to update the file when done modifying.

Example: **SaveRecord**.

See also: **ReadRecord**, **WriteRecord**, **CloseRecordFile**, **UpdateRecordFile**.

NextRecord: (C64, C128) **C27A**

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) INV_RECORD.
y	track of first data block. If no error, then a value of value of $00 here means the record is allocated but not in use (has no data blocks).
a	new current record number.
r1L	Track of first data block in record.
r1H	Sector of first data block in record.

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: **SaveRecord**.

See also: **PreviousRecord**, **PointRecord**.

OpenRecordFile: (C64, C128) **C274**

Function: Open an existing VLIR file for access.

Parameters: **r0** FILENAME — pointer to null-terminated name of file (word).

Uses:

curDrive	device number of the active drive.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x	error ($00 = no error). STRUCT_MISMATCH
r1L	track of VLIR index block.
r1H	sector of VLIR index block.
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.

†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** initializes 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**.

Note: GEOS up to 2.0 does not support a hierarchical file system, the "current directory" is actually the entire disk.

Note:[3] This routine calls **GetDirHead** which loads in the BAM from disk. **PutDirHead** should be called prior to this routine if the BAM has been modified by freeing or allocating blocks.

Example: **SaveRecord**.

See also: **ReadRecord**, **WriteRecord**, **CloseRecordFile**, **UpdateRecordFile**.

PointRecord: (C64, C128) **C280**

Function: Make a particular record the current record.

Parameters: a RECORD — record number to make current.

Uses:

fileHeader	index table checked to establish whether record exists.
usedRecords	number of currently used records in the VLIR file.

Returns:

x	error ($00 = no error). INV_RECORD
y	track of VLIR record. If no error, then a value of value of $00 here means record is allocated but not in use (has no data blocks).
a	new current record number.
r1L	track of VLIR record.
r1H	sector of VLIR record.

Note: **r1**

$0000	record is not allocated.
$FF00	record is allocated but not in use (has no data blocks); this information is already flagged in y.
other	track/sector of first data block in record.

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 numbered 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: **SaveRecord**.

See also: **NextRecord**, **PreviousRecord**.

PreviousRecord: (C64, C128) **C27D**

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). INV_RECORD
y	track of VLIR record. 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.
r1L	track of VLIR record.
r1H	**s**ector of VLIR record.

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: **SaveRecord**.

See also: **NextRecord**, **PointRecord**.

ReadRecord: (C64, C128) **C28C**

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 blocks) (word).

Uses:

curDrive	device number of the active drive.
curRecord	current record number.
fileHeader	VLIR index table. Table holds track / sector of first block of each record.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.

Returns:

x	error ($00 = no error). BFR_OVERFLOW
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.
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.
r2	unused bytes remaining in *BUFFER*.
r5L	byte index into **fileTrScTab** of last entry (last entry = **fileTrScTab** plus value in **r5**).

Alters:

fileTrScTab	As the chain of 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.
diskBlkBuf	Each block is read into **diskBlkBuf** before copying to *BUFFER.*

Destroys: y, (**r1**), **r3-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**.

UpdateRecordFile: (C64, C128) C295

Function: Update the disk copy of the VLIR index table, BAM 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	device number of the active drive.
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.

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).

†*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.

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) and time/date-stamps the directory entry. If the **fileWritten** flag is **FALSE**, it does nothing.

UpdateRecordFile writes out the index block, adds the time/date-stamp and **fileSize** information to the directory entry, and writes out the new BAM with a call to **PutDirHead**.

Because 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 executed, no conflicts will arise.

Example:

See also: **CloseRecordFile**, **OpenRecordFile**.

WriteRecord: (C64, C128) **C28F**

Function: Write data to the current VLIR record.

Parameters:

r2	BYTES	— data bytes to write to record. Can write up to 32,258 bytes (127 blocks).
r7	RECDATA	— pointer to start of record data (word).

Uses:

curDrive	device number of the active drive.
fileWritten†	if **FALSE**, assumes record just opened (or updated) and reads BAM into memory.
curRecord	current record number.
fileHeader	VLIR index table stored in this buffer.
curType	GEOS 64 v1.3 and later for detecting REU shadowing.
curDirHead	BAM updated to reflect newly allocated block.
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: x error ($00 = no error).

Alters:

fileWritten†	set to **TRUE** to indicate that 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.
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	BAM updated to reflect newly freed and allocated blocks.
dir2Head†	(BAM for 1571 and 1581 drives only).
dir3Head†	(BAM for 1581 drive only).

Destroys: 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. 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.

WriteRecord does not write the BAM 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.

Example:

See also: **ReadRecord**, **WriteFile**.

graphics

Name	Addr	Description	Page
BitmapClip	C2AA	Display a compacted bitmap, clipping to a sub-window.	20-85
BitmapUp	C142	Display a compacted bitmap without clipping.	20-87
i_BitmapUp	C1AB	Inline **BitmapUp**.	20-87
BitOtherClip	C2C5	**BitmapClip** with data coming from elsewhere (e.g., disk)	20-88
ColorCard	C2F8	C128 Get or Set a Color Card. In 40 or 80-column mode.	20-90
ColorRectangle	C2F8	C128 Draw a Color rectangle on the 80-column Screen.	20-91
DrawLine	C130	Draw, clear, or recover line between two endpoints.	20-92
DrawPoint	C133	Draw, clear, or recover a single screen point.	20-93
FrameRectangle	C127	Draw a rectangular frame (outline).	20-94
i_FrameRectangle	C1A2	Inline **FrameRectangle**.	20-94
GetScanLine	C13C	Calculate scanline address.	20-95
GraphicsString	C136	Execute a string of graphics commands.	20-96
i_GraphicsString	C1A8	Process a graphic command table / inline	20-96
HorizontalLine	C118	Draw a horizontal line in a pattern	20-98
InvertLine	C11B	Invert the pixels on a horizontal screen line.	20-99
ImprintRectangle	C250	Imprint rectangular area to background buffer.	20-100
i_ImprintRectangle	C253	Inline **ImprintRectangle**.	20-100
InvertRectangle	C12A	Invert the pixels in a rectangular screen area.	20-101
NormalizeX	C2E0	Normalize C128 X-coordinates for 40/80 modes.	20-102
RecoverLine	C11E	Recover horizontal screen line from background buffer.	20-104
Rectangle	C124	Draw a filled rectangle.	20-106
i_Rectangle	C19F	Inline **Rectangle**.	20-106
RecoverRectangle	C12D	Recover rectangular screen area from background buffer.	20-105
i_RecoverRectangle	C1A5	Inline **RecoverRectangle**.	20-105
SetColorMode	C2F5	Change GEOS 128 80-column Color Mode	20-107
SetNewMode	C2DD	Change GEOS 128 graphics mode (40/80 switch).	20-108
SetPattern	C139	Set current fill pattern.	20-109
TestPoint	C13F	Test status of single screen point (on or off?).	20-110
VerticalLine	C121	Draw a vertical line in a pattern.	20-111

BitmapClip: (C64, C128) **C2AA**

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 coordinate: pixel position / 8 (byte).
r1H	YPOS	— y-coordinate (byte).
r2L	W_WIDTH	— width in cards: pixel width / 8 (byte).
r2H	W_HEIGHT	— height in pixels (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 is placed at (XPOS*8, YPOS). The lower-right corner is at (XPOS*8+W_WIDTH*8, YPOS+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**.

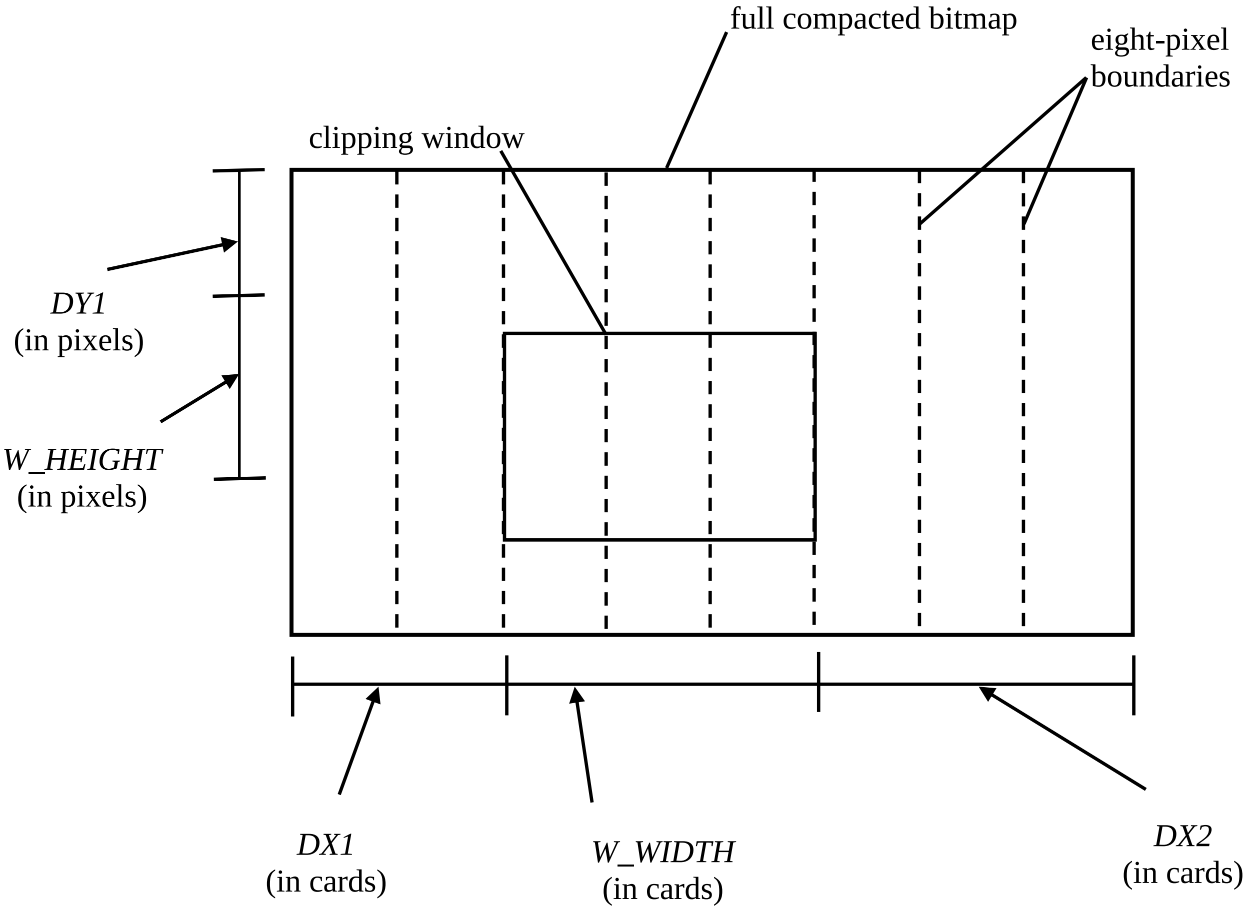
Description: **BitmapClip** uncompacts a rectangular area of a full bitmap, clipping (ignoring) any data that exists outside of the desired area. The rectangular subset is called the *clipping window*.

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 (respectively) 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 Kernal's where the release byte is greater than $01, these problems have been fixed.[†]

Note[3]:[†] There is no supporting documentation or sample code to identify the location of the "release" Kernal variable. **BitmapClip** problems were fixed in the version of GEOS 128 1.3 that included CONFIGURE 1.4.

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



C64, **C128**: 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 initially, 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 cards to skip at the end of each scanline.

Example: **DisplayImage**.

See also: **BitmapUp**, **BitOtherClip**.

BitmapUp:, i_BitmapUp (C64, C128) C142, C1AB

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 YPOS — 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-coordinate: pixel position / 8.
.byte YPOS — 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, YPOS). The lower-right corner is at (XPOS*8+WIDTH*8, YPOS+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-r9L**.

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
```

Example: **ShowBitmap**.

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

BitOtherClip: (C64, C128) **C2C5**

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). ***Note****: Set by SYNC before first byte is retrieved. Does not need to be pre loaded.*
r1L	XPOS	— x-card-coordinate: pixel position /8 (byte).
r1H	YPOS	— y-coordinate (byte).
r2L	WIDTH	— width in cards: pixel width/8 (byte).
r2H	HEIGHT	— height in pixels (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 (word). Called each time a byte from a compacted bitmap is needed; data byte is returned via *BUFFER*. ldy #0 and then sta (r0),y.
r14	SYNC	— pointer to synchronization routine (word). Called before 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 bitmap is placed at (XPOS * 8, YPOS). The lower-right corner is at (XPOS * 8+WIDTH*8, YPOS+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 are 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 set the data byte in the *BUFFER*. A typical *APPINPUT* routine saves any pseudoregisters (**r0-r13)** 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 example: **AppInput**.

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 *WIDTH* parameters will automatically double the x-position and the width (respectively) in 80-column mode.

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

Warning: **BitOtherClip** is unable to handle big count compression. This is from a bug that exists in all versions of the GEOS Kernal. The result of the bug can be either screen corruption or getting stuck in an endless loop.

Example: **BitOtherClip Example**.

See also:

ColorCard: (C128) C2F8

Function: Get or set a color card. In 40 or 80-column mode.

Parameters: **r3** X1 — x-coordinate of screen pixel (word).
r11L Y1 — y-coordinate of screen pixel (byte).
st carry MODE:

C	Operation
1	set color attribute with COLOR value in a.
0	get color attribute and return value in a.

When Setting:
a COLOR — new color to change color card attribute to.

Uses: **graphMode** GRMODE — determines which Screen attributes to use.
When *GRMODE* = **GR_80**:
vdcClrMode contains the value of the current VDC color mode.

Returns: When Getting:
a color card attribute at requested location.

When Setting:
nothing.

Destroys: a, x, y, **r5**.

Description: **ColorCard** will set, or read a single color card. Setting a color card sets its byte value to *COLOR*. Reading an attribute gets its value and returns the color card byte in a.

The color card offset is calculated and added to the attribute base address to get the final address of the color card byte. ***Note****: (X1 must already be normalized prior to calling* ***ColorCard****).*

Example:

color card address = attribute base + (X1 / 8) + (Y1 / (Color Card Height)).
Note*: color card height is determined by the color mode in* ***vdcClrMode****.*

The color card is retrieved from the **COLOR_MATRIX** in 40-column mode, or from the VDC's attributes in 80-column mode.

The carry (c) flag in the processor status register (s) is used to pass *MODE* to **ColorCard**. The following can be used before the call to set or clear this flag appropriately:

- Use sec to set carry (c) flag in order to set a new color card.
- Use clc clear the carry (c) flag in order to get a color card.

Example:

```
.macro SetColorCard color
         lda   #[color
         sec
         jsr   ColorCard
.endm
```

Example:

See also: **ColorRectangle**, **SetColorMode**.

ColorRectangle: (C128) C2FB

Function: Draw a color rectangle on the 80-column screen.

Parameters:

a	FBCOLOR	— foreground and background color to draw. (byte).
	b7-b4:	foreground color.
	b3-b0:	background color.
r2L	Y1 — y-coordinate of top corners (byte).	
r2H	Y2 — y-coordinate of bottom corners (byte).	
r3	X1 — x-coordinate of left corners (word).	
r4	X2 — x-coordinate of right corners (word).	

where (X1, Y1) is the upper-left corner of the rectangle and (X2, Y2) is the lower-right corner.

Calls: **ColorCard**.

Returns: nothing.

Destroys: a, x, y, **r11L**.

Description: **ColorRectangle** draws a color rectangle on the screen as determined by *FBCOLOR* and the coordinates of the upper-left and lower-right corners. (The rectangle is NOT filled with the current fill pattern). **ColorRectangle** draws using *FBCOLOR* to set Foreground and Background color cards in the VDC Attributes.

The color card width is 8 pixels, and are aligned on horizontal byte boundaries. The current **vdcClrMode** determines the height of the color card. In 8x8 mode each color card covers a matrix 8 pixels wide by 8 rows high. 8x4 is 4 rows high and 8x2 is 2 rows.

Since the color cards are at fixed aligned boundaries there is more resolution in the passed coordinates then can actually be used. All x-coordinates are divided by 8 to compute the offset into the attributes. Passing a value of 32 as an x-coordinate and passing 33 will yield the same results on the screen. This works the same way with the y-coordinate but varies by the color card resolution set by **vdcClrMode**.

ColorRectangle operates by calling **ColorCard** in a loop, changing the attribute for every color card for every set of lines that fall in boundaries of the current color card height.

128: **ColorRectangle** does not normalize x-coordinates. Normal use is to call **Rectangle** first to draw the foreground image. Then call **ColorRectangle** using the now normalized x-coordinates. Otherwise you can call **NormalizeX** for *X1* and *X2*.

Example:

See also: **SetColorMode**.

DrawLine: (C64, C128) **C130**

Function: Draw, clear, or recover a line defined by two arbitrary endpoints.

Parameters:

r3	X1 — x-coordinate of pixel (word).
r11L	Y1 — y-coordinate of pixel (byte).
r4	X2 — x-coordinate of second endpoint (word).
r11H	Y2 — y-coordinate of second pixel (byte).
st	MODE:

N	C	Operation
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: <u>when setting or clearing pixels</u> (not recovering):

dispBufferOn:

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

Destroys: a, x, y, **r3-13**.

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 **Graphics Routines** for more information).

Example:

See also: **TestPoint**, **DrawLine**.

DrawPoint: (C64, C128) **C133**

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	Operation
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 (not recovering):
dispBufferOn:
bit 7: write to foreground screen if set.
bit 6: write to background screen if set.

Destroys: a, x, y, **r5-r6**.

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* 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 **Graphics Routines** for more information).

Example:

See also: **TestPoint**, **DrawLine**.

FrameRectangle:, **i_FrameRectangle** (C64, C128) **C127**, **C1A2**

Function: Draw a rectangular frame (one-pixel thickness).

Parameters: Normal:

a	eight-bit line pattern.
r2L	Y1 — y-coordinate of top corners (byte).
r2H	Y2 — y-coordinate of bottom corners (byte).
r3	X1 — x-coordinate of left corners (word).
r4	X2 — x-coordinate of right corners (word).

Inline:

*data appears immediately after the jsr **i_FrameRectangle**.*

.byte	Y1 — y-coordinate of top corners.
.byte	Y2 — y-coordinate of bottom corners.
.word	X1 — x-coordinate of left corners.
.word	X2 — x-coordinate of right corners.
.byte	PATTERN — eight-bit line pattern.

where (X1, Y1) is the upper-left corner of the frame and (X2, Y2) is the lower-right corner.

Uses: **dispBufferOn**:

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

Destroys: a, x, y, **r5-r9**, **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.

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 **Graphics Routines** for more information).

Example:

See also: **Rectangle**, **ImprintRectangle**, **RecoverRectangle**, **InvertRectangle**.

GetScanLine: (C64, C128) **C13C**

Function: Calculate the memory address of a particular screen line.

Parameters: x YCOORD — y-coordinate of line.

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

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 40-column mode specific (see notes below for 128 80-column mode).

```
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.

Example:

See also:

GraphicsString:, **i_GraphicsString** (C64, C128) **C136**, **C1A8**

Function: Execute a string of graphics commands.

Parameters: Normal:
r0 **GRSTRING** — pointer to null-terminated graphic string (word).

Inline:
data appears immediately after the jsr ***i_GraphicsString****.*
.byte **GRSTRING** — null-terminated graphics string data.

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-r13**.

Description: When GEOS was first being developed, it was found that it was common for an application to call a large number of graphic routines to set up various screen displays — clearing the screen, drawing boxes and window borders, etc. A shorthand method for doing this was therefore developed. **GraphicsString** allows the application to create a string of graphic commands to be executed in turn, thereby grouping the calls in a convenient format and saving any space that would have been taken up by parameter loading and jsr's.

GraphicsString introduces the concept of a pen position, an (x, y) coordinate on the screen used as the base for the graphics operation. For example, the **GraphicsString** LINETO command has only two parameters — an x- and a y-position. The line is drawn from the current pen position to the (x, y) point. The ending point of the line then becomes the new pen position. In this way, a series of connected lines can be drawn by supplying the successive endpoints. The pen-position is in an unknown state when **GraphicsString** is called. A MOVEPENTO command should be issued to set the initial pen position.

In the **GraphicsString** commands, an x-position is always a word value and a y-position is always a byte value. However, delta-values — values which specify a change in the current pen x- or y-position — are two's complement signed words. Note that even though the high-byte of the delta y-position is required, it is not used in GEOS 2.0/Wheels 4.4, but it is possible it could be used in future versions.

Code from the Kernal to show this behavior:

```
;--- on entry r0 points to the data after the PEN_Y_DELTA command
Pen_Y_Delta:
      ldy   #0
      lda   (r0),y       ; get low-byte of y-delta
      iny                ; (point y to high-byte of delta word)
      add   penYPos      ; add low-byte of y-delta to current pen position
      sta   penYPos      ; save result
      ;---   (don't do anything with high-byte of delta word)
      iny                ; point y to next command byte
      AddYW r0           ; add y to r0 so r0 now points to the next command byte
90$
      rts                ; exit
```

The available **GraphicsString** commands are:

Command	No.	Example	Description
`NULL`	`0`	`.byte NULL`	Graphics string terminator byte.

`MOVEPENTO`	`1`	`.byte MOVEPENTO`
		`.word x`
		`.byte y`
			Make the current pen position the (x, y) coordinate specified.

`LINETO`	`2`	`.byte LINETO`
		`.word x`
		`.byte y`
			Draw a line from the current pen position to the (x, y) position specified. (x, y) becomes the current pen position.

`RECTANGLETO`	`3`	`.byte RECTANGLETO`
		`.word x`
		`.byte y`
			Draw a rectangle using the pattern byte from the current pen position to opposing corner (x, y) specified.

`PENFILL`	`4`	`n/a`	Not Currently Implemented.
`NEWPATTERN`	`5`	`.byte NEWPATTERN` `.byte ptrn`	Change the current pattern to ptrn; see **SetPattern**

`ESC_PUTSTRING`	`6`	`.byte ESC_PUTSTRING`
		`.word x`
		`.byte y`
		`.byte "String",NULL`
			The remainder of the string is treated as input to the **PutString** command, where (x, y) is the coordinate where the string is placed.

`FRAME_RECTO`	`7`	`.byte FRAME_RECTO`
		`.word x`
		`.byte y`
			Frame a solid rectangle. Start at the current pen position to (x, y), which becomes the new pen position.

`PEN_X_DELTA`	`8`	`.byte PEN_X_DELTA` `.word dx`	add the signed value of dx to the pen's current x-position.
`PEN_Y_DELTA`	`9`	`.byte PEN_Y_DELTA` `.word dy`	add the signed value of dy to the pen's current y-position.

`PEN_XY_DELTA`	`10`	`.byte PEN_XY_DELTA`
		`.word dx`
		`.word dy`
			add the signed values of dx and dy to the pen's current x- and y-position.

Note: Any lines or rectangle frames are drawn with the solid bit-pattern (%11111111).

Note: When using ESC_PUTSTRING, note that **PutString** will not return to **GetString** when it encounters a null. The null actually marks the end of the whole string. To resume graphics string processing, use the **PutString** ESC_GRAPHICS escape.

Example: **GrphcsStr1**.

See also: **PutString**.

HorizontalLine: (C64, C128) **C118**

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.

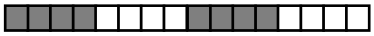
Uses: **dispBufferOn**:
bit 7: write to foreground screen if set.
bit 6: write to background screen if set.

Returns: **r11L** unchanged.

Destroys: a, x, y, **r5-r8**, **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-coordinate.

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 **Graphics Routines** for more information).

Example:

See also: **VerticalLine**, **InvertLine**, **RecoverLine**, **DrawLine**.

InvertLine: (C64, C128) **C11B**

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: write to foreground screen if set.
bit 6: write to background screen if set.

Returns: nothing.

Destroys: a, x, y, **r5-r8**.

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 **Graphics Routines** for more information).

Example:

See also: **VerticalLine**, **HorizontalLine**, **RecoverLine**, **DrawLine**.

ImprintRectangle:, **i_ImprintRectangle** (C64, C128) **C250**, **C253**

Function: Imprints the pixels within a rectangular region from the foreground screen to the background buffer.

Parameters: Normal:

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

Inline:

data appears immediately after the jsr ***i_ImprintRectangle****.*

.byte	Y1 — y-coordinate of top corners.
.byte	Y2 — y-coordinate of bottom corners.
.word	X1 — x-coordinate of left corners.
.word	X2 — x-coordinate of right corners.

where (X1, Y1) is the upper-left corner of the rectangle and (X2, Y2) is the lower-right corner.

Returns: nothing.

Destroys: a, x, y, **r5-r8**, **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 **Graphics Routines** for more information).

Note^3: ImprintLine is an internal Kernal routine. It does not have a jump table entry.

Example:

See also: **RecoverRectangle**, **Rectangle**, **InvertRectangle**.

InvertRectangle: (C64, C128) **C12A**

Function: Inverts the pixels within a rectangular region.

Parameters:

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

where (X1, Y1) is the upper-left corner of the rectangle and (X2, Y2) is the lower-right corner.

Uses: **dispBufferOn**:

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

Returns: nothing.

Destroys: a, x, y, **r5-r8**, **r11**.

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 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 **Graphics Routines** for more information).

Example:

See also: **Rectangle**, **ImprintRectangle**, **RecoverRectangle**, **FrameRectangle**.

NormalizeX: (C128) **C2E0**

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 14 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-a9). 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 address of r9
jsr    NormalizeX          ; normalize the value in r9
```

The following breakdown of the word-length x-coordinate illustrates how the special bits affect the adjustment process.

b15	b14	b13	x-pixel coordinate (b0-b12)

b0-12	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 changed (normal positive).
1	1	n	x value changed (normal negative).
0	1	n	x=x*2-n (double negative).
1	0	n	x=x*2+n (double positive).

Note: For more information, Refer to "**GEOS 128 X-position and Bitmap Doubling**" in chapter **Graphics Routines** for more information.

Example:

See also:

RecoverLine: (C64, C128) **C11E**

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: **r3**, **r4**, and **r11L** unchanged.

Destroys: a, x, y, **r5-r8**.

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 **Graphics Routines** for more information).

Example:

See also: **HorizontalLine**, **InvertLine**, **VerticalLine**, **DrawLine**.

RecoverRectangle:, **i_RecoverRectangle**(C64, C128) **C12D**, **C1A5**

Function: Recovers the pixels within a rectangular region from the background buffer to the foreground screen.

Parameters: Normal:

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

Inline:

data appears immediately after the jsr ***i_RecoverRectangle****.*

.byte	Y1 — y-coordinate of top corners.
.byte	Y2 — y-coordinate of bottom corners.
.word	X1 — x-coordinate of left corners.
.word	X2 — x-coordinate of right corners.

where (X1, Y1) is the upper-left corner of the rectangle and (X2, Y2) is the lower-right corner.

Returns: **r2**, **r3**, and **r4** unchanged.

Destroys: a, x, y, **r5-r8, r11**.

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 **Graphics Routines** for more information).

Example:

See also: **ImprintRectangle**, **Rectangle**, **InvertRectangle**.

Rectangle:, **i_Rectangle** (C64, C128) **C124**, **C19F**

Function: Draw a rectangle in the current fill pattern.

Parameters: Normal:

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

Inline:
data appears immediately after the jsr ***i_Rectangle****.*
.byte Y1 — y-coordinate of top corners.
.byte Y2 — y-coordinate of bottom corners.
.word X1 — x-coordinate of left corners.
.word X2 — x-coordinate of right corners.

where (X1, Y1) is the upper-left corner of the rectangle and (X2, Y2) is the lower-right corner.

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

Destroys: a, x, y, **r5-r8, r11**.

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.

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 **Graphics Routines** for more information).

Example:

See also: **FrameRectangle**, **SetPattern**, **ImprintRectangle**, **RecoverRectangle**, **InvertRectangle**.

SetColorMode: (C128) **C2F5**

Function: Change GEOS 128 80-column Color Mode.

Parameters: a CLRMODE — New Color Mode to change to.

Uses: **graphMode** GRMODE — Must be **GR_80** for the new mode to be set.

Alters: **vdcClrMode** Contains the value of the current Color Mode.

Returns: nothing.

Destroys: a, x, y, **r0**.

Description: **SetColorMode** Sets up the VDC for the desired graphic mode. *CLRMODE* Must be in the following list of valid modes.

Constant	Value	Resolution	Color Mode	Attribute Address	RAM
VDC_CLR0	0	640 x 200	monochrome*	n/a	16K
VDC_CLR1¥	1	640 x 176	8x8 color	v3880	16K
VDC_CLR2†	2	640 x 200	8x8 color	v4000	64K
VDC_CLR3†	3	640 x 200	8x4 color	v4000	64K
VDC_CLR4†	4	640 x 200	8x2 color	v4000	64K

After changing to modes 1-4, the application should set all the color cards for the screen using **ColorRectangle**.

Note:* GEOS default color mode.

Note: ¥ Attempting to draw to rows > 175 will corrupt the attribute area. The application is responsible for maintaining a valid y-coordinate (0-175). Use **mouseBottom** to **windowBottom** to constrain the mouse and text output, setting them both to a value <= 175. **Note**: Graphics routines are not constrained by the reduced resolution of this color mode. They will attempt to draw to any row up to 199 as they normally would.

Note: † These modes require 64K of VDC RAM. GEOS does not check to see if the RAM is actually available. The application is responsible for confirming the amount of VDC RAM available to the system prior to changing modes.

Note: **vdcClrMode** should not be directly set by the application.

Example:

See also:

SetNewMode: (C128) C2DD

Function: Changes GEOS 128 from 40-column mode to 80-column mode, or vice-versa.

Parameters: none.

Uses: **graphMode** GRMODE — new graphics mode to change to:
40-Column: GR_40.
80-Column: GR_80.

Returns: nothing.

Destroys: a, x, y, **r0**.

Description: **SetNewMode** 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 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.

Note[3]: The original guide states **r0-r15** are destroyed when this routine is called. This was not accurate.

Example: **ChangeMode**.

See also:

SetPattern: (C64, C128) **C139**

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: **SetPattern** sets the current fill pattern. There are 34 system patterns (numbered 0-33) in GEOS; 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 **curPattern** in order to access pattern 32, add 16 to access pattern 33, and so on.

Example:

See also:

TestPoint: (C64, C128) C13F

Function: Test and return the value of a single point (pixel).

Parameters: **r3** X1 — x-coordinate of pixel (word).
r11L Y1 — y-coordinate of pixel (byte).

where (X1, Y1) is the coordinate of the point to test.

Uses: **dispBufferOn**:
bit 7: write to foreground screen if set.
bit 6: write to background screen if set.
(if both bit 6 and bit 7 are set, then only the pixel in the background screen is tested).

Returns: **r3L**, **r11L** unchanged.

Destroys: a, x, y, **r5-r6**.

Description: **TestPoint** will test a pixel in cither 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.

C128: 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 2 **Graphics Routines** for more information).

Example:

See also: **DrawPoint**.

VerticalLine: (C64, C128) **C121**

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 bottommost endpoint (byte).

where (X1, Y1) and (X1, Y2) define the endpoints of the vertical line.

Uses: **dispBufferOn**:

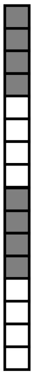
bit 7:	write to foreground screen if set.
bit 6:	write to background screen if set.

Returns: **r3L**, **r3H**, **r4** unchanged.

Destroys: a, x, y, **r5-r8L**.

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.

Example:

See also: **HorizontalLine**.

icon/menu

Name	Addr	Description	Page
DoIcons	`C15A`	Display and begin interaction with icons.	20-113
DoMenu	`C151`	Display and begin interaction with menus.	20-114
DoPreviousMenu	`C190`	Retract sub-menu and reactivate menus up one level.	20-116
GotoFirstMenu	`C1BD`	Retract all sub-menus and reactivate at main level.	20-117
RecoverAllMenus	`C157`	Recover all menus from background buffer.	20-118
RecoverMenu	`C154`	Recover single menu from background buffer.	20-119
ReDoMenu	`C193`	Reactivate menus at the current level.	20-120

DoIcons: (C64, C128) **C15A**

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: a, x, y, **r0-r15**.

Description: **DoIcons** takes an *ICONTABLE*, draws the enabled icons (those whose OFF_I_PIC 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 must 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).

3: **faultData** is cleared to $00, indicating no faults.

4: 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.

5: 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.

Note: The maximum number of icons in *ICONTABLE* is 31 (MAX_ICONS).

Example: **IconsUp**.

See also: **DoMenu**.

DoMenu: (C64, C128) **C151**

Function: Display and activate 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: a, x, y, **r0-r13**.

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 **GotoFirstMenu**, 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. **mouseXPos** and **mouseYPos** are set so that the pointer is centered over the selection number passed in a. 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:
      php                    ; save processor status register
      sei                    ; disable interrupts around call
      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 to their saved state
      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**.

DoPreviousMenu: (C64, C128) **C190**

Function: Retracts the current sub-menu and reactivates menus at the previous level.

Parameters: none.

Destroys: assume a, x, y, **r0-r15**.

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 **mouseXPos** and **mouseYPos** 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 disastrous.

Example:

See also: **DoMenu**, **GotoFirstMenu**, **ReDoMenu**, **RecoverMenu**.

GotoFirstMenu: (C64, C128) C1BD

Function: Retracts the current sub-menu and reactivates menus at the previous level.

Parameters: none.

Destroys: assume a, x, y, **r0-r15**.

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**.

RecoverAllMenus: (C64, C128) C157

Function: Removes all menus (including the main menu) from the foreground screen by recovering from the background buffer.

Parameters: none.

Destroys: assume a, x, y, **r0-r15**.

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**.

RecoverMenu: (C64, C128) **C154**

Function: Removes the current menu from the foreground screen by recovering from the background buffer.

Parameters: none.

Destroys: assume a, x, y, **r0-r15**.

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: **DoMenu**.

ReDoMenu: (C64, C128) C193

Function: Reactivate menus at the current level.

Parameters: none.

Destroys: assume a, x, y, **r0-r15**.

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**.

input driver

Name	Addr	Description	Page
InitMouse	`FE80`	Initialize input device.	20-122
SetMouse	`FD09`	C128 Reset input device scanning circuitry.	20-123
SlowMouse	`FE83`	Reset mouse velocity variables.	20-124
UpdateMouse	`FE86`	Update mouse variables from input device.	20-125

InitMouse: (C64, C128) **FE80**

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**.

Example:

See also: **SlowMouse**, **UpdateMouse**, **SetMouse**, **StartMouseMode**, **MouseUp**.

SetMouse: (C128) **FE89**

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. 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**.

Example:

See also: **SlowMouse**, **UpdateMouse**, **InitMouse**.

SlowMouse: (C64, C128) **FE83**

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.

Example:

See also: **UpdateMouse**, **InitMouse**, **SetMouse**.

UpdateMouse: (C64, C128) **FE86**

Function: Update the mouse variables based on any changes in the state of the input device.

Parameters: none.

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

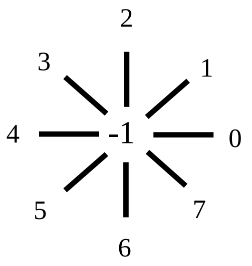
Destroys: assume a, x, y, **r0-r15**.

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 convert 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.

Example:

See also: **SlowMouse**, **InitMouse**, **SetMouse**.

internal

Name	Addr	Description	Page
BootGEOS	`C000`	Reboot GEOS. Requires only 128 bytes at $C000.	20-127
FirstInit	`C271`	Initialize GEOS variables.	20-128
GetSerialNumber	`C196`	Return GEOS serial number.	20-129
InterruptMain	`C100`	Main interrupt level processing.	20-130
MainLoop	`C1C3`	GEOS **MainLoop** processing.	20-131
Panic	`C2C2`	System-error dialog box.	20-132
Reset	`03E4`	C128 Reset handler located in BackRAM	20-133
ResetHandle	`C003`	internal Bootstrap entry point.	20-134

BootGEOS: (C64, C128) **C000**

Function: Restart GEOS from a non-GEOS application.

Parameters: none.

Returns: does not return.

Destroys: n/a.

Description: **BootGEOS** provides a method for a non-GEOS application 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** and **BootGEOS**+$7F. 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 a "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 **bootName** for the ASCII (not CBMASCII) string "GEOSBOOT". If loaded by GEOS, the program can check bit 5 of **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 return to GEOS on a Commodore 64, set **CPU_DATA** to KRNL_BAS_IO_IN ($37) and then jump to **BootGEOS**. To return to GEOS on a Commodore 128 see the following examples:

```
;--- C128 must first setup the system configuration before jumping to BootGEOS
;    Note: Code must reside below $4000 in the low Common ram area.

.psect $1BFE                                ; any valid location in bottom 16K
   GoGEOS:
       rmbf   0,config                      ; map in I/O so mmurcr can be set
       setbit mmurcr,#%00110000,#%01000111  ; set Common ram on for bottom 16K
                                            ; and VIC in bank 1
       LoadB  config,#CIO_IN                ; activate bank 1 memory
       jmp    BootGEOS                      ; reboot GEOS
```

Example: **RoadTrip**.

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

FirstInit: (C64, C128) **C271**

Function: Simulates portions of the GEOS cold start procedure without actually rebooting GEOS or destroying the application in memory.

Parameters: none.

Returns: GEOS variables and system hardware in a cold start state; stack and application space unaffected.

Destroys: a, x, y, **r0-r2**.

Description: **FirstInit** is part of the GEOS cold start 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.

Example:

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

GetSerialNumber: (C64, C128) **C196**

Function: Return the 16-bit serial number or pointer to the serial string for the current GEOS Kernal.

Parameters: none.

Returns: **r0** 16-bit serial number.

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.

Example:

See also:

InterruptMain: (C64, C128) **C100**

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.

Note: **InterruptMain** is a subset of the full interrupt level process. **InterruptMain** is typically called through the **intTopVector**. 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.

Example:

See also: **MainLoop**.

MainLoop: (C64, C128) **C1C3**

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**.

Panic: (C64, C128) **C2C2**

Function: Display "system error" dialog box.

Parameters: 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**.

See also: **InterruptMain**.

Reset: (C128) **03E4**

Function: Internal handler used during the C128 reset process.

Parameters: none.

Returns: does not return.

Description: **Reset** is only used during the C128 reset process. Normally activated by the reset switch. It is not useful to applications and is documented here only because it is part of the GEOS Kernal. **Reset** is located in BackRAM.

```
Reset:
     LoadB     config,#CIO_IN
     jmp       BootGEOS
```

Example:

See also: **BootGEOS**.

ResetHandle: (C64, C128) **C003**

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.

Example:

See also: **BootGEOS**.

math

Name	Addr	Description	Page
BBMult	`C160`	Byte by byte (single-precision) unsigned multiply.	20-136
BMult	`C163`	Byte by word unsigned multiply.	20-137
Dabs	`C16F`	Double-precision signed absolute value.	20-138
Ddec	`C175`	Double-precision unsigned decrement.	20-139
Ddiv	`C169`	Double-precision unsigned division.	20-140
DMult	`C166`	Double-precision unsigned multiply.	20-142
Dnegate	`C172`	Double-precision signed negation.	20-143
DSdiv	`C16C`	Double-precision signed division.	20-144
DShiftLeft	`C15D`	Double-precision left shift (zeros shifted in).	20-145
DShiftRight	`C262`	Double-precision right shift (zeros shifted in).	20-146

BBMult: (C64, C128) **C160**

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, 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 *0PERAND2* and the result is stored as a word back in *OPERAND1*. Note *OPERAND1* starts out as a byte parameter but becomes a word result with the high-byte at *OPERAND1*+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 because the result always fits in a word ($FF*$FF = $FE01).

Example: **8BitMultiply**.

See also: **BMult**, **DMult**, **Ddiv**, **DSdiv**.

BMult: (C64, C128) **C163**

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 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 high-byte of *OPERAND2* and calling **DMult**. The result is stored as a word back in *OPERAND1*.

Note: **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**:* **16x8Multiply**, **ConvToUnits**.

See also: **BMult**, **DMult**, **Ddiv**, **DSdiv**.

Dabs: (C64, C128) C16F

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 returns 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: **DSmult**.

See also: **DNegate**.

Ddec: (C64, C128) **C175**

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. The absolute address of the word is passed in x. If the result of the decrement is zero, then 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.

Note[3]: The macro **DecW** should be used in cases where speed is more important than code size. Inner loops should always use **DecW** if space allows. **Ddec** should be used when space is at a premium as it costs only 5 bytes to use. The Kernal uses **Ddec** in **CRC** because space in the Kernal is more valuable than the speed of the **CRC** procedure that is not normally ever used in an inner loop. See Example: **DdecvsDecW**.

Example: **Kernal_CRC**, **DdecvsDecW**, **DecCounter**, **DecZW**.

See also:

Ddiv: (C64, C128) **C169**

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, 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:

```
zpage = $00
         ;---      Example use of the validated Ddiv wrapper.
         ldx       #r0                 ; point x to dividend
         ldy       #r1                 ; point y to divisor
         jsr       DdivValidated       ; call our validated Ddiv routine
         bmi       99$                 ; branch on divide by zero error
         ...

DdivValidated:
         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
```

There is no possibility of overflow (a result which cannot fit in 16 bits).

Example: **ConvToUnits**, **CheckDiskSpace**, **NewDdiv**.

See also: **DSdiv**, **DMult**, **BBMult**, **BMult**.

DivideBySeven: (Apple)

Function: Divide a byte value by 7.

Parameters: **r0L** OPERAND1 — byte to divide by 7.

Returns: a result.

Destroys: a.

Description: Bonus code page: Commodore GEOS has no **DivideBySeven** in the Kernal like Apple GEOS does, so here is a block to do a similar operation on an 8-bit value.

```
DvBy7:
      lda    r0L
      lsr
      lsr
      lsr
      adc    r0L
      ror
      lsr
      lsr
      adc    r0L
      ror
      lsr
      lsr
      rts
```

Example:

See also:

DMult: (C64, C128) **C166**

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: *results 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.

r7 contains the top 8 bits of a 24bit result. Overflow in the result beyond 24-bits is ignored.

Example: **DSmult**.

See also: **Bmult**, **BBMult**, **Ddiv**, **DSdiv**.

Dnegate: (C64, C128) **C172**

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.

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: **DSmult**, **NewSDSdiv**.

See also: **Dabs**.

DSdiv: (C64, C128) **C16C**

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).

Returns:
x, 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.

Destroys: a, **r9**.

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**.

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 fix this problem:

See Example: **NewSDSdiv**.

Note: Because **r8** and **r9** are used in the division process, they cannot be used as 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:

```
zpage = $00
         ...
         ldx       #r0                 ; point x to dividend
         ldy       #r1                 ; point y to divisor
         lda       zpage,y                   ; get low-byte of divisor
         ora       zpage+1,y           ; get high-byte of divisor
         bne       10$                 ; if either non-zero, go divide
         jmp       DivideByZero        ; else, flag error
10$
         jmp       DSdiv               ; divide
```

Example: **NewDSdiv, NewSDSdiv**.

DShiftLeft: (C64, C128) C15D

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 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 word will not be shifted.

Carry Flag <- High-byte <- Low-byte

Carry Flag <-	High-byte	<- Low-byte	
C	7-6-5-4-3-2-1-0	7-6-5-4-3-2-1-0	<- 0

Example:

See also: **Ddiv**, **DMult**, **BBMult**, **BMult**.

See also: **DShiftRight**.

DShiftRight: (C64, C128) **C262**

Function: Arithmetically right-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 right (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: **DShiftRight** is a double-precision math 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 right-shift will divide by two, two right-shifts will divide by four, three right-shifts will divide by eight, and so on: value = value / 2^{count}.

Note: If a COUNT of $00 is specified, the word will not be shifted.

	High-byte ->	**Low-byte ->**	**Carry Flag**
0 ->	7-6-5-4-3-2-1-0	7-6-5-4-3-2-1-0	C

Example: **MseToCardPos**, **ConvToUnits**.

See also: **DShiftLeft**.

memory

Name	Addr	Description	Page
AccessCache	`C2EF`	C128 Provides a mechanism for disk drivers to cache up to 21 blocks.	20-148
ClearRam	`C178`	Clear memory to $00.	20-149
CmpFString	`C26E`	Compare two fixed-length strings.	20-150
CmpString	`C26B`	Compare two null-terminated strings.	20-151
CopyFString	`C268`	Copy a fixed-length string.	20-152
CopyString	`C265`	Copy a null-terminated string.	20-153
DoBOp	`C2EC`	C128 backRAM memory move/swap/verify primitive.	20-154
DoRAMOp	`C2D4`	Primitive for communicating with REU (RAM-Expansion Unit).	20-155
FetchRAM	`C2CB`	Transfer data from RAM-Expansion Unit.	20-156
FillRam	`C17B`	Fill memory with a particular byte.	20-157
i_FillRam	`C1B4`	Inline **FillRam**.	20-157
i_MoveData	`C1B7`	Inline **MoveData**.	20-160
InitRam	`C181`	Initialize memory areas from table.	20-158
MoveBData	`C2E3`	128 backRAM memory move routine.	20-159
MoveData	`C17E`	Intelligent memory block move.	20-160
StashRAM	`C2C8`	Transfer memory to RAM-Expansion Unit.	20-161
SwapBData	`C2E6`	128 swap memory between front/backRAM.	20-162
SwapRAM	`C2CE`	Swap memory with an REU memory block.	20-163
VerifyBData	`C2E9`	128 backRAM verify.	20-164
VerifyRAM	`C2D1`	RAM-Expansion Unit verify.	20-165

AccessCache: (C128) **C2EF**

Function: Provides a mechanism for disk drivers to cache up to 21 blocks.

Parameters:

r1H	BLOCK	— block number (0-20) (byte).
r4	BUFFER	— address of block buffer; must be at least BLOCKSIZE bytes (word).
y	MODE	— operation mode:

b0	b1	Description
0	0	Save block at *BUFFER* to cache.
0	1	Read block from cache and write to *BUFFER*.
1	0	Swap block at *BUFFER* with block in cache.
1	1	verify (compare) *BUFFER* against contents of cache.
	-1	Erase cache. (*MODE*=$FF)

***Note**: the MODE parameter closely matches the low nibble of the **DoRAMOp** CMD parameter.*

Calls: **DoBOp**.

Returns: *When saving:*
nothing.

When reading:
contents of cache block written to *BUFFER*
Z=1 block data on cache is not valid. (block has not been written to cache since the last erase).
Z=0 block data on cache is valid.

When verifying:
a, y $00 if data matches.
$FF if mismatch.

Destroys: a, x, y.

Description: **AccessCache** provides disk drivers with the ability to maintain a cache of 21 blocks. The cache is large enough to hold a full track on a 1541 or 1571 drive. The 1541, 1571 and 1581 disk drivers use the cache to speed up access to directory blocks. Drivers reset the cache every time a new disk is detected and every time the active device is changed.

Note: **Verify** has an oddity where it returns the error status in the y and a-registers. the x-register is always $00 regardless of the outcome of the verify.

Note: **Erase Cache** erases the first 2 bytes of every block in the cache. The remaining bytes of the blocks are left unchanged.

Note: **AccessCache** appears in the jump table in Wheels 4.4 but performs no function. It immediately does an rts.

Example:

See also:

ClearRam: (C64, C128) **C178**

Function: Clear a region of memory to $00.

Parameters:

r0	COUNT	— number of bytes to clear (0 - 64K) (word).
r1	ADDR	— address of area to clear (word).

Returns: nothing.

Destroys: a, y, **r0**, **r1**, **r2L**.

Description: **ClearRam** clears *COUNT* bytes starting at *ADDR* to *ADDR+COUNT*. It useful for initializing ramsect variables and data sections.

Note: Do not use **ClearRam** to initialize **r0-r2L**. Also, for when space is at a premium, it actually takes fewer bytes to call **i_FillRam** with a fill value of $00.

Note[1]: **ClearRam** sets **r2L** to $00 and calls **FillRam**.

Example: **InitBuffers**.

See also: **FillRam**, **InitRam**.

CmpFString: (C64, C128) **C26E**

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: **Find**.

See also: **CmpString**, **CopyFString**.

CmpString: (C64, C128) **C26B**

Function: Compare two null-terminated strings.

Parameters:

x SOURCE — zero page address of pointer to source null terminated string (byte pointer to a word pointer).

y DEST — zero page address of pointer to destination null terminated 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 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.
bgt	branch if *SOURCE* string is greater than *DEST* string.
bcc	branch if *SOURCE* string is less than *DEST* string.
ble	branch if *SOURCE* string is less than or equal to *DEST* string.

Note: **CmpString** cannot compare strings longer than 256 bytes (including the null). The compare process is aborted after 256 bytes.

Example: **Find2**.

See also: **CmpFString**, **CopyString**.

CopyFString: (C64, C128) **C268**

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 strings (1-255)

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 *DEST* areas overlap, the *DEST* address must be less than the *SOURCE* address for the copy to work properly.

Note: 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: **CopyBuffer**.

See also: **CopyString**, **CmpString**, **MoveData**.

CopyString: (C64, C128) **C265**

Function: Copy a null-terminated string.

Parameters:
x SOURCE — zero page address of pointer to a NULL terminated source string (byte pointer to a word pointer).
y DEST — zero page address of pointer to destination buffer (byte pointer to a word pointer).

Returns: Buffer pointed to by *DEST* contains copy of *SOURCE* string, including the terminating NULL.

Destroys: a, x, y.

Description: **CopyString** copies a null terminated string pointed to by *SOURCE* to the buffer pointed to by *DEST*. All Characters in the string are copied, including the null-terminator. If the *SOURCE* and *DEST* areas overlap, the *DEST* address must be less than the *SOURCE* address for the copy to work properly.

Note: **CopyString** cannot copy more than 256 bytes. The copy process is aborted after 256 bytes.

Example: **CopyBuffer**, **CopyStr**.

; Kernal code for **CopyString** and **CopyFString**

```
_CopyString:
          lda #0                ; load flag for null terminated copy
_CopyFString:
          stx    cpyset+1       ; set zero page source
          sty    cpyset+3       ; set zero page destination
          tax                   ; set copy flag (0=NULL Terminated)
          ldy    #0             ; set index to start of string
cpyset:
10$       MoveB  "(0),Y","(0),Y" ; move byte from SOURCE to DEST
          bne    20$            ; if NULL and X=0 (NULL terminated copy)
          bxeq   90$            ;    then exit.
20$       iny                   ; if y has wrapped around
          beq    90$            ;    then exit
          bxeq   10$            ; if null terminated copy then get next char
          dex                   ; if fixed length not reached
          bne    10$            ;    then get next char
90$       rts
```

See also: **CopyFString**, **CmpString**, **MoveData**.

DoBOp: (C128) C2EC

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 = frontRAM; 1 = backRAM (byte).
r3H	A2BANK	— *ADDR2* bank: 0 = frontRAM; 1 = backRAM (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 nibble of the **DoRAMOp** CMD parameter.*

Returns: **r0-r3** unchanged.

When verifying:

x $00 if data matches.
$FF if mismatch.
DEV_NOT_FOUND if bank or REU not available.

when MODE=1:

Values in **r0** and **r1** are swapped on return. This is so **DoBOp** can put the source addr and dest address in the correct order for its call to **MoveBData**.

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*.

Important[3]: Using *MODE* 1 will cause the address' in **r0** and **r1** to be swapped. If an application is not expecting this behavior, unexpected results may occur as a result of the swapped register contents.

Example:

See also: **MoveBData**, **SwapBData**, **VerifyBData**, **DoRAMOp**.

DoRAMOp: (C64 v1.3+, C128) **C2D4**

Function: Primitive for communicating with REU (RAM-Expansion Unit) devices.

Parameters:

r0	CBMSRC	— address in Main Memory (word).
r1	REUDEST	— address in REU bank (word).
r2	COUNT	— number of bytes to operate with (word).
r3L	REUBANK	— REU bank number to use (byte).
y	CMD	— command to send to REU (byte).

Returns: **r0-r3L** 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 = success).

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-nibble to be %1001 (transfer with current configuration and disable FF00 decoding). The lower nibble can be one of the following:

b1	b0	Description
0	0	Transfer from Commodore to REU.
0	1	Transfer from REU to Commodore.
1	0	Swap.
1	1	Verify.

***Note**: the low nibble of the **DoRAMOp** CMD parameter closely matches the **DoBOp** MODE parameter.*

Note: On a Commodore 128, if the VIC chip is mapped to frontRAM (with the MMU VIC bank pointer), the REU will read/write using frontRAM. Similarly, if the VIC chip is mapped to backRAM, the REU will read/write using backRAM. The REU ignores the standard bank selection controls on the 8510. GEOS 128 defaults with the VIC mapped to frontRAM.

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**, **VerifyRAM**, **DoBOp**.

FetchRAM: (C64 v1.3+, C128) **C2CB**

Function: Primitive for transferring data from an REU.

Parameters:

r0	CBMDEST	— address in Main Memory to start writing (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-r3L	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 = success).

Destroys: y.

Description: **FetchRAM** moves a block of data from a REU BANK into Commodore memory. This routine is a "use at your own risk" 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**: (C64, C128) **C17B**, **C1B4**

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:

*data appears immediately after the jsr **i_FillRam**.*
.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 any desired value.

Note: Do not use **FillRam** to initialize **r0-r2L**.

Example: **InitBuffers**.

See also: **ClearRam**, **InitRam**.

InitRam: (C64, C128) **C181**

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 of 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       NULL                    ; end table
```

where **InitRam** expects the next *address* parameter.

Note: Do not use **InitRam** to initialize **r0-r2L**.

Example:

See also: **FillRam**, **ClearRam**.

MoveBData: (C128) **C2E3**

Function: Special version of **MoveData** that will move data within either frontRAM or backRAM (or from one bank to the other).

Parameters:

r0	SOURCE	— address of source block in memory (word).
r1	DEST	— address of destination block in memory (word).
r2	COUNT	— number of bytes to move (word).
r3L	SRCBANK	— source bank: 0 = backRAM; 1 = frontRAM (byte).
r3H	DSTBANK	— destination bank: 0 = backRAM; 1 = frontRAM (byte).

Returns: **r0-r3** unchanged.

Destroys: a, x, y.

Description: **MoveBData** is a block move routine that allows data to be moved in either frontRAM, backRAM, 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, the *DEST* address must be less than the *SOURCE* address for the copy to work properly.

MoveBData is especially useful for copying data from frontRAM to backRAM or from backRAM to frontRAM.

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 significantly slower than **MoveData** and should be avoided if the move will occur entirely within frontRAM.

Example:

See also: **MoveBData**, **SwapBData**, **VerifyBData**, **DoBOp**.

MoveData:, **i_MoveData**: (C64, C128) **C17E**, **C1B7**

Function: Moves a block of data from one area to another.

Parameters: Normal:

r0	SOURCE	— address of source block in memory (word).
r1	DEST	— address of destination block in memory (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 insufficient 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: **MoveData** should only be used to move data within the standard frontRAM application space. Use **MoveBData** to move memory within backRAM or between frontRAM and backRAM.

GEOS 128 2.0 will use DMA-move only when the following conditions are met:

1. bit 7 of **sysRAMFlg** is set.
2. **r2** is less than $3900 bytes. ($5800 bytes in GEOS 128 1.4).
3. **r0** and **r1** are greater than $1FF.

Because the RAM-Expansion Unit DMA follows the VIC chip bank select, an application that is displaying a 40-column screen from backRAM must either disable DMA-moves or temporarily switch the VIC chip to frontRAM before the **MoveData** call.

Note: Do not use **MoveData** on **r0**-**r6**.

Example:

See also: **MoveBData**, **CopyString**.

StashRAM: (C64 v1.3+, C128) **C2C8**

Function: Primitive for transferring data to an REU.

Parameters:

r0	CBMSRC	— address in Main Memory to start reading (word).
r1	REUDEST	— address in REU bank to stash data (word).
r2	COUNT	— number of bytes to stash (word).
r3L	REUBANK	— REU bank number to stash to (byte).

Returns:

r0-r3L 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 = success).

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: **SwapRAM**, **FetchRAM**, **VerifyRAM**, **DoRAMOp**, **MoveBData**.

SwapBData: (C128) **C2E6**

Function: Swaps two regions of memory within either frontRAM or backRAM (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 = frontRAM; 1 = backRAM (byte).
r3H	A2BANK	— *ADDR2* bank: 0 = frontRAM; 1 = backRAM (byte).

Returns: **r0-r3** unchanged.

Destroys: a, x, y.

Description: **SwapBData** is a block swap routine that allows data to be swapped in either frontRAM, backRAM, 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 frontRAM to backRAM or from backRAM to frontRAM.

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**.

SwapRAM: (C64 v1.3+, C128) C2CE

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 bank 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**.

VerifyBData: (C128) C2E9

Function: Compares (verifies) two regions of memory against each other. The regions may either be in frontRAM or backRAM (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 swap (word).
r3L	A1BANK	— *ADDR1* bank: 0 = frontRAM; 1 = backRAM (byte).
r3H	A2BANK	— *ADDR2* bank: 0 = frontRAM; 1 = backRAM (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 frontRAM, backRAM, or in frontRAM and backRAM. 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) C2D1

Function: Verify (compare) data in main 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 (word).
r3L	REUBANK	— REU bank number to verify with (byte).

Returns:

r0-r3L 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 to Verify the contents match. If bit 5 of the a-register is set, there was a failed comparison during validation. 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: **SwapRAM**, **FetchRAM**, **StashRAM**, **DoRAMOp**, **VerifyBData**.

mouse/sprite

Name	Addr	Description	Page
ClearMouseMode	`C19C`	Stop input device monitoring.	20-167
HideOnlyMouse	`C2F2`	(128) Temporarily remove soft-sprite mouse pointer.	20-168
IsMseInRegion	`C2B3`	Check if mouse is inside a window.	20-169
MouseOff	`C18D`	Disable mouse pointer and GEOS mouse tracking.	20-170
MouseUp	`C18A`	Enable mouse pointer and GEOS mouse tracking.	20-171
SetMsePic	`C2DA`	Set and preshift new soft-sprite mouse picture.	20-172
StartMouseMode	`C14E`	Start monitoring input device.	20-173
TempHideMouse	`C2D7`	Hide soft-sprites before direct screen access.	20-174

ClearMouseMode: (C64, C128) C19C

Function: Stop monitoring the input device.

Parameters: nothing.

Returns: nothing.

Destroys: a, x, y, **r3L**.

Alters:

mouseOn	set to $00, totally disabling all mouse tracking.
mobenble	sprite #0 bit cleared by **DisablSprite**.

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**.

Example:

See also: **StartMouseMode**, **MouseOff**.

HideOnlyMouse: (C128) **C2F2**

Function: Temporarily removes the soft-sprite mouse pointer from the graphics screen.

Parameters: nothing.

Returns: nothing.

Uses: **graphMode** Current video mode for C128

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.

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 useful if using GEOS graphics routines.

Example:

See also: **TempHideMouse**.

IsMseInRegion: (C64, C128) **C2B3**

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:

```
bmi          InRegion            ; branch if mouse was in region
     -or-
bpl          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: See **NewIsMseInRegion**.

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 **Graphics Routines** for more information).

On return from **IsMseInRegion**, *X1* and *X2* are normalized in **r3** and **r4**.
Example:

```
...
LoadW r4,#200 | DOUBLE_W
jsr IsMseInRegion
;--- r4 = 400
```

Example: **NewIsMseInRegion**.

See also: **HorizontalLine**.

MouseOff: (C64, C128) **C18D**

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: **MouseUp**, **ClearMouseMode**.

MouseUp: (C64, C128) **C18A**

Function: Reenables the mouse pointer and GEOS mouse tracking.

Parameters: nothing.

Returns: nothing.

Modifies:

mobenble	sprite #0 bit cleared by **DisablSprite**.
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**.

Note: **StartMouseMode** calls this routine.

Example:

See also: **MouseOff**, **ClearMouseMode**.

SetMsePic: (C128) **C2DA**

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: (word).

ARROW ($0000).

Returns: nothing.

Destroys: a, x, y, **r0-r15**.

Description: The software sprite routines used by GEOS 128 in 80-column mode 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 fastmoving 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**.

Note[3]: ARROW = $0000.

Example: **ArrowUp**.

See also: **TempHideMouse**, **HideOnlyMouse**, **DrawSprite**.

StartMouseMode: (C64, C128) C14E

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.

If the mouse will be repositioned, then disable interrupts around the call to **StartMouseMode**.

Example: **MouseInit**.

See also: **ClearMouseMode**, **MouseUp**, **MouseOff**, **SlowMouse**, **DoMenu**, **DoIcons**, **TempHideMouse**, **HideOnlyMouse**.

TempHideMouse: (C128) C2D7

Function: Temporarily removes soft-sprites and the mouse pointer from the graphics screen.

Parameters: nothing.

Uses: **graphMode**.

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**.

Note: In 40-column mode (bit 7 of **graphMode** is zero), **TempHideMouse** exits immediately without affecting the hardware sprites.

Example:

See also: **HideOnlyMouse**.

print driver

Name	Addr	Description	Page
GetDimensions	`790C`	Get CBM printer page dimensions.	20-176
InitForPrint	`7900`	Initialize printer (once per document).	20-177
PrintASCII	`790F`	Send ASCII data to printer.	20-178
PrintBuffer	`7906`	Send graphics data to printer.	20-179
SetNLQ	`7915`	Begin near-letter quality printing.	20-180
StartASCII	`7912`	Begin ASCII mode printing.	20-181
StartPrint	`7903`	Begin graphics mode printing.	20-182
StopPrint	`7909`	End page of printer output.	20-183

Note: C128 caches the active print driver. See **GetFile** notes to see how to load the driver from cache instead of from disk.

GetDimensions: (C64, C128) **790C**

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 accommodate 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 the 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. GEOS printer drivers always assume 80-columns by 66 lines.

Example:

See also: **StartPrint**, **StartASCII**.

InitForPrint: (C64, C128) **7900**

Function: Initialize printer. Perform once per document.

Parameters: none.

Returns: nothing.

Destroys: assume a, x, 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.

InitForPrint is also used to set the printer baud rate for serial printers.

Example:

See also: **StartPrint**, **StartASCII**.

PrintASCII: (C64, C128) **790F**

Function: Send ASCII string to the printer.

Parameters: **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.

Returns: nothing.

Destroys: assume a, x, y, **r0-r15**.

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 form feed 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.

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**. GEOS printer drivers are set to print 80 characters per line and 66 lines per page.

Example:

See also: **StartPrint**, **StartASCII**.

PrintBuffer: (C64, C128) **7906**

Function: Print one cardrow (eight lines) of graphics data.

Parameters:

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).

Returns: nothing.

Destroys: a, x, y, **r0-r15**.

Description: **PrintBuffer** prints eight lines of graphic data on the printer. The maximum width of each line is determined by the capabilities 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**.

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 form feed when necessary.

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.

Example:

See also: **PrintASCII**, **StartPrint**, **StopPrint**, **InitForPrint**.

SetNLQ: (C64, C128) 7915

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.

Example:

See also: **StartASCII**, **PrintASCII**.

StartASCII: (C64, C128) 7912

Function: Enable ASCII text mode printing.

Parameters: **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.

Returns: x STATUS — printer error code; $00 = no error.

Destroys: assume a, y, **r0-r15**.

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.

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.

Example:

See also: **PrintASCII**, **StopPrint**, **StartPrint**.

StartPrint: (C64, C128) **7903**

Function: Enable graphics-mode printing.

Parameters: **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.

Returns: x STATUS — printer error code; $00 = no error.

Destroys: a, y, **r0-r15**.

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.

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.

Example:

See also: **StopPrint**, **StartASCII**.

StopPrint: (C64, C128) 7909

Function: Flush output buffer and form feed the printer (called at the end of each page).

Parameters:
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).

Returns: x STATUS — printer error code; $00 = no error.

Destroys: assume a, x, y, **r0-r15**.

Description: **StopPrint** instructs the printer driver to flush any internal buffers and end the page.

StopPrint ends both graphic and ASCII printing.

Note: GEOS printer drivers always form feed when **StopPrint** is called.

Example:

See also: **StartPrint**, **StartASCII**.

process

Name	Addr	Description	Page
BlockProcess	`C10C`	Block process from running. Does not freeze timer.	20-185
EnableProcess	`C109`	Make a process runnable immediately.	20-186
FreezeProcess	`C112`	Pause a process countdown timer.	20-187
InitProcesses	`C103`	Initialize processes.	20-188
RestartProcess	`C106`	Unblock, unfreeze, and restart process.	20-189
Sleep	`C199`	Put current routine to sleep for a specified time.	20-190
UnblockProcess	`C10F`	Unblock a blocked process, allowing it to run again.	20-191
UnfreezeProcess	`C115`	Unpause a frozen process timer.	20-192

BlockProcess: (C64, C128) **C10C**

Function: Block a processes event.

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 runnable flag of a particular process so that if a process timer reaches zero (causing the process to become runnable) no process event is generated until the process is subsequently unblocked with a call to **UnblockProcess**. **BlockProcess** stops the process 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 runnable flag is set and the timer is restarted. As long as the process is blocked, though, **MainLoop** ignores this runnable flag and, therefore, never generates an event. When a blocked process is later unblocked, **MainLoop** checks the runnable flag. If the runnable 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 particular process
```

See also: **UnblockProcess**, **FreezeProcess**.

EnableProcess: (C64, C128) **C109**

Function: Makes a process runnable immediately.

Parameters: x PROCESS — process to enable (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 runnable on the next pass through **MainLoop**, independent of its timer value.

EnableProcess merely sets the runnable flag in the 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 privileged status and cannot override a blocked process. However, because it doesn't depend on or affect the current timer value, the process can become runnable 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 creating application-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**.

FreezeProcess: (C64, C128) **C112**

Function: Freeze a process's countdown timer at its current value.

Parameters: x PROCESS — process to freeze (0 to *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 runnable 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** has no effect.

Example:

See also: **UnfreezeProcess**, **BlockProcess**.

InitProcesses: (C64, C128) **C103**

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 arc 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, runnable). 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 referring 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.

Process Table record structure:

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.

Note[3]: MAX_PROCESSES = 20.

Example:

See also: **Sleep**, **RestartProcess**.

RestartProcess: (C64, C128) C106

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 initially start a process after a call to **InitProcesses** or to rewind a process to the beginning of its cycle.

Note: **RestartProcess** clears the runnable 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**.

Sleep: (C64, C128) **C199**

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, putting the current routine "to sleep". At Interrupt Level, the *DELAY* value associated with each sleeping routine is decremented. When the associated *DELAY* value reaches zero, **MainLoop** removes the sleeping routine from the sleep table and performs 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**.

<u>Sleeping in Detail:</u>

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** executes 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 imperative that this is in fact a return address; do not save any values on the stack before calling **Sleep**).

5: At interrupt level GEOS decrements the 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: **BeepThrice**.

See also: **InitProcesses**.

UnblockProcess: (C64, C128) **C10F**

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 runnable flag so that if a process timer reaches zero (causing the process to become runnable) 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 runnable while it is blocked. As long as the process is blocked, however, **MainLoop** will ignore the runnable flag. When the process is subsequently unblocked, **MainLoop** will recognize a set runnable 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: (C64, C128) C115

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**.

sprite

Name	Addr	Description	Page
DisablSprite	`C1D5`	Disable sprite.	20-194
DrawSprite	`C1C6`	Define sprite image.	20-195
EnablSprite	`C1D2`	Enable sprite.	20-196
PosSprite	`C1CF`	Position sprite.	20-197

DisablSprite: (C64, C128) **C1D5**

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**, **PosSprite**.

DrawSprite: (C64, C128) **C1C6**

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: a, y, **r5**.

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).

The 64th byte has size information that is required by the software sprite routines. The format of this byte is:

b7:	1	sprite <= 9 pixels wide
	0	sprite > 9 pixels wide
b6-0:	number of scan lines (1-21)	

Refer to "**Chapter 12 Sprites/Soft Sprites**" for more information.

Example:

See also: **PosSprite**, **EnablSprite**, **DisablSprite**.

EnablSprite: (C64, C128) **C1D2**

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**, **MouseOff**, **PromptOff**, **DrawSprite**, **PosSprite**.

PosSprite: (C64, C128) **C1CF**

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	sprite x-position (lower 8-bits).
msbNxpos	sprite x-position (bit 9).
mobNypos	sprite y-position.

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 x-positions are translated to C64 hardware coordinates (newXPos = **NormalizeX**(*XPOS*) **/** 2)) 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.

Example:

See also: **DrawSprite**, **EnablSprite**, **DisablSprite**.

text

Name	Addr	Description	Page
GetCharWidth	C1C9	Calculate width of char without style attributes.	20-199
GetNextChar	C2A7	Get next character from keyboard queue.	20-200
GetRealSize	C1B1	Calculate actual character size with attributes.	20-201
GetString†	C1BA	Get string input from user.	20-202
InitTextPrompt	C1C0	Initialize text prompt.	20-204
LoadCharSet	C1CC	Load and begin using a new font.	20-205
PromptOff	C29E	Turn off text prompt.	20-206
PromptOn	C29B	Turn on text prompt.	20-207
PutChar†	C145	Display a single character to screen.	20-208
PutDecimal†	C184	Format and display an unsigned double-precision number.	20-209
PutString†	C148	Print string of characters to screen.	20-210
i_PutString†	C1AE	Inline **PutString**.	20-210
SmallPutChar	C202	Fast character print routine.	20-211
UseSystemFont	C14B	Use default system font (BSW 9).	20-212

Note†: Under GEOS 128, OR’ing DOUBLE_W into the *X* parameters of text routines 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 **Graphics Routines** for more information).

GetCharWidth: (C64, C128) **C1C9**

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.

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**.

GetNextChar: (C64, C128) **C2A7**

Function: Retrieve the next character from the keyboard queue.

Parameters: none.

Returns: a keyboard character code of character 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: **KeyHandler**.

See also: **GetString**.

GetRealSize: (C64, C128) **C1B1**

Function: Calculate the printed size of a character based on any style attributes.

Parameters:

	a	CHAR — character code of character.
	x	MODE — style mode (as stored in **currentMode**).

Uses: **curHeight**.
baselineOffset.

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 on 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.

Note: **lastWidth** is local to the GEOS Kernal and therefore inaccessible to applications. **lastWidth** contains the actual width of the most recently printed character.

Note:

Bold:	increases width by 1.
Outline:	increases height and width by 2.

Underline, italic, reverse do not change the size of the character.

Although the size changes are currently predictable, you should always use **GetRealSize** to get the character size to insure compatibility with future versions of the operating system.

Example: **ClipChar**.

```
GetSizeW:
      ;--- 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
```

See also: **GetCharWidth**.

GetString: (C64, C128) **C1BA**

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 data string (if no default data string is used, the first byte of the buffer must be NULL). This buffer must be at least *MAX_CH*+1 bytes long (word).
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). (**r2L** must be >= size of the default data string in **r0**) (byte).
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 (word).
keyVector	STRINGDONE	—routine to call when the string is terminated by the user typing a carriage return. $0000 = no routine provided (word).

Uses:

*at call to **GetString**:*

curHeight	for size of text prompt.
baselineOffset	for positioning default data 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.
string	*BUFR* (pointer to start of string buffer).
keyVector	$0000.
StringFaultVec	$0000.

Destroys:

*at call to **GetString**:*

a, x, y, **r0-r13**.

Description: **GetString** installs a character handling routine into **GetString** 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. Use ST_WRGS_FORE with **dispBufferOn** to limit output to the foreground screen.

The following is a breakdown of what **GetString** does:

1: Variables local to the **GetString** character input routine are initialized. Global string input variables such as **string**, **stringX**, and **stringY** are also initialized.

2: **PutString** is called to output the default data string stored in the character buffer. If no default data 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 initialized by calling **InitTextPrompt** with the value in **curHeight**. **PromptOn** is also called.

6: Control is returned to the application.

Note: 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.

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.

Note^2: If the user manages to type off the end of the screen, specifically past **rightMargin**, **GetString** will stop echoing characters although it will still enter the characters into the buffer.

Example: **NewGetString**.

See also: **PutChar**, **PutString**, **GetNextChar**.

InitTextPrompt: (C64, C128) **C1C0**

Function: Initialize sprite #1 for use as a text prompt.

Parameters: a HEIGHT — pixel height for the prompt.

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.

Example:

See also: **PromptOn**, **PromptOff**.

LoadCharSet: (C64, C128) **C1CC**

Function: Begin using a new font.

Parameters: **r0** **FONTPTR** — address of font header (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 initialize the font variables for the font pointed at by the *FONTPTR* parameter.

Example:

See also: **UseSystemFont**.

PromptOff: (C64, C128) **C29E**

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**.

Example: **KillPrompt**.

See also: **InitTextPrompt**, **PromptOn**.

PromptOn: (C64, C128) **C29B**

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 be 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.

Example: **KillPrompt**.

See also: **InitTextPrompt**, **PromptOff**.

PutChar: (C64, C128) **C145**

Function: Process a single character code (both **escape codes** and **printable characters**).

Parameters:

a	CHAR	— character code.
r11	XPOS	— x-coordinate of left of character (word).
r1H	YPOS	— y-coordinate of character baseline (byte).

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****).*	
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	unchanged.

Destroys: a, x, y, **r1L**, **r2-r10**, **r12**, **r13**.

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 **r1L** and **PutChar** vectors through **StringFaultVec**. If the right-edge of the character 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.

Note: A complete list of GEOS escape codes and character codes appears in "**Chapter 19 Environment**" **"Structures / Keyboard"**.

Example:

See also: **SmallPutChar**, **PutString**, **PutDecimal**.

PutDecimal: (C64, C128) **C184**

Function: Format and print a 16-bit positive integer.

Parameters:	a	FORMAT	— formatting codes — 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 (byte).

Uses: same as **PutChar**.

Returns:	**r11**	x-position for next character.
	r1H	unchanged.

Destroys: a, x, y, **r0**, **r1L**, **r2-r10**, **r12**, **r13**.

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 byte in the a-register as follows:

FORMAT:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
b7	b6	b0-b5					

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_SUPRESS
SET_NOSUPRESS

Note: The maximum 16-bit decimal number is 65535 ($FFFF), so the printed number will never exceed five characters.

Example:

See also: **PutChar**.

PutString:, i_PutString (C64, C128) C148, C1AE

Function: Print a string to the screen.

Parameters: Normal:

r0 STRING — pointer to string data (word).
r11 XPOS — x-coordinate of left of character (word).
r1H YPOS — y-coordinate of character baseline (byte).

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).
r0 points to NULL terminator for *STRING*.

Destroys: a, x, y, **r1L**, **r2-r10**, **r12**, **r13**.

Description: **PutString** passes a full string of data to **PutChar** a character at a time. **PutChar** 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**, **r1L**, 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 **"Text**, **Fonts**, **and Keyboard Input"**.

Basic operation of **PutString**:

```
SudoPutString:
10$
      ldy    #0          ; use zero offset
      lda    (r0),y      ; get character
      beq    90$         ; exit if NULL terminator
      jsr    PutChar     ; otherwise process char
      IncW   r0          ; move to next byte in string
      bra    10$         ; and loop through again
90$
      rts                ; exit
```

Note: 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.

Note: A complete list of GEOS escape codes and character codes appears in "**Chapter 19 Environment**" **"Structures / Keyboard"**.

Example: **Print**, **PutStrFault**.

See also: **PutChar**, **GraphicsString**.

SmallPutChar: (C64, C128) **C202**

Function: Print a single character without the **PutChar** overhead.

Parameters:

a	CHAR	— character code.
r11	XPOS	— x-coordinate of left of character (word).
r1H	YPOS	— y-coordinate of character baseline (byte).

Uses: same as **PutChar**.

Returns:

r11	x-position for next character.
r1H	unchanged.

Destroys: a, x, y, **r1L**, **r2-r10**, **r12**, **r13**.

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 that 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 **leftMargin**, including negative x-position clipping, is not supported by early versions of GEOS 64 (earlier than v1.4) — the entire character is clipped instead. Full **leftMargin** clipping is supported on all other versions of GEOS: GEOS 64 v1.4 and above, GEOS 128 (both in 64 and 128 mode).

Like **PutChar**, 159 is the maximum *CHAR* value that **SmallPutChar** will handle correctly. Most fonts will not have characters for codes beyond 129.

Example: **ClipChar**.

See also: **PutChar**, **PutString**.

UseSystemFont: (C64, C128) C14B

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**.

utility

Name	Addr	Description	Page
Bell	n/a	1000 Hz Bell sound.	20-214
CallRoutine	`C1D8`	pseudo-subroutine call. $0000 aborts call.	20-215
CRC	`C20E`	Cyclic Redundancy Check calculation.	20-216
DoInlineReturn	`C2A4`	Return from inline subroutine.	20-217
GetRandom	`C187`	Calculate new **random** number.	20-218
ToBasic	`C241`	Pass Control to Commodore BASIC.	20-219

Bell: (Apple GEOS) **n/a**

Function: Makes a brief beeping sound.

Parameters: none.

Returns: nothing.

Destroys: a.

Description: **Bell** sounds a 1000^{Hz} signal. The sound lasts approximately 1/10th of a second.

Note: **Bell** does not exist in Commodore GEOS. Use the following code with your GEOS application to provide the behavior of the Apple Bell Kernal routine.

```
; Based on code in the original HGG.
; Author: Dan Kaufman (w Chris Hawley)
; Updated by: Paul B Murdaugh

         PULSE          = %01000001
         NOTE590        = $25DF          ; 590Hz       (original value for frequency)
         NOTE1K         = $4016          ; 1000Hz      (actual value for 1000HZ)
Bell:
         PushB      CPU_DATA             ; switch to I/O space
         LoadB      CPU_DATA,#IO_IN
         LoadB      sidVCReg,#0
         LoadW      sidVoc1+O_PULSEWIDTH,#$800
         LoadB      sidAtDcy,#$06
         LoadB      sidSuRel,#$00
         LoadW      sidVoc1+O_FREQUENCY,#NOTE1K
         LoadB      sidVCReg,#PULSE
         PopB       CPU_DATA             ; return to memory space
         rts
```

Example: **BeepThrice**.

See also:

CallRoutine: (C64, C128) **C1D8**

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 simulating a jsr to the address in the x and a register, 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 to ignore the vector. Other examples of this usage are **intTopVector**, **intBotVector**, 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 **intTopVector** and **intBotVector**). Do not use **CallRoutine** to call inline (i_) routines, as it will not return properly.

Example: **HandleCommand**, **KeyTrap**.

See also:

CRC: (C64, C128) **C20E**

Function: 16-bit cyclic redundancy check (CRC).

Parameters:
r0 DATA — pointer to start of data (word).
r1 LENGTH — of bytes to check (word).

Returns: **r2** CRC value for the specified range (word).

Destroys: a, y, **r0-r3L**.

Description: **CRC** calculates a 16-bit cyclic-redundancy error-checking value on a range of data. This value can be used to check the integrity of the data at a later time. For example, before saving off a data file, an application might perform a **CRC** on the data and save the value along with the rest of the data. Later, when the application reloads the data, it can perform another CRC on it and compare the new value with the old value. If the two are different, the data has unquestionably been corrupted.

Note: Given the same data, **CRC** will produce the same value under all versions of GEOS.

Note[1]: This routine is called by the bootup routines to compute the checksum of GEOS BOOT. This checksum is used to create the interrupt vector address. The reason for this was to prevent piracy.

Example: **Kernal_CRC**

```
MAGIC_VALUE     = $0317              ; CRC value that we’re looking for
DATA_SIZE       = $2434              ; size of data

.ramsect
      buffer:
                .block DATA_SIZE

.psect

Checksum:
      LoadW     r0,#buffer           ; r0 <- data area to checksum
      LoadW     r1,#DATA_SIZE        ; r1 <- bytes in buffer to check
      jsr       CRC                  ; r2 <- CRC value for data area
      CmpWI     r2,#MAGIC_VALUE      ; return status to caller
      rts                            ; if equal (beq), then CRC is good
```

See also:

DoInlineReturn: (C64, C128) **C2A4**

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 **DolnlineReturn**.
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 (a 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 modified 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**.

See also:

GetRandom: (C64, C128) **C187**

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 modulus 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:

See also:

ToBasic: (C64, C128) **C241**

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 (word).

r5 **DIRENTRY** — 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 (word).

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 (word).

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 **BootGEOS-BootGEOS+**$7F are preserved (those bytes may be saved and restored later). To return to GEOS, the called program can execute a jump to **BootGEOS**.

A program in the C64 environment can check to see if it was loaded by GEOS by checking the memory starting at **bootName** for the ASCII (not CBMASCII) string "GEOS BOOT”. If loaded by GEOS, the program can check bit 5 of **sysFlgCopy**, if this bit is reset, ask the user to insert their GEOS boot disk; if this bit is set, GEOS will reboot from the RAM expansion unit to actually return to GEOS, set **CPU_DATA** to KRNL_BAS_IO_IN ($37) and jump to **BootGEOS**.

Note: C128: To return to GEOS:

```
;--- Code must reside below $4000
rmbf   0,config                         ; Map in I/O in current bank
setbit mmurcr,#%00110000,#%01000111     ; Common ram on for bottom 16K
                                        ; VIC in bank 1
LoadB  config,#CIO_IN                   ; Activate bank 1 memory
jmp    BootGEOS                         ; Return to GEOS
```

Example: **LoadBASIC**.

See also: **BootGEOS**.

Wheels Kernal 4.4

Introduction

excerpts from original Wheels Manual
This operating system came about ten years after the release of GEOS 2.0 and about 12 years after the first release of GEOS. If you've been a loyal GEOS user all along, then you've likely grown to appreciate the many nice features of GEOS and the common sense that went into the original development of it. GEOS is a remarkable enhancement to the Commodore computer and has had much to do with keeping this computer platform alive all these years.

Since much of the original GEOS Kernal has been rewritten or changed in Wheels, what's actually left of the GEOS Kernal can be found in versions of GEOS prior to V2.0. Therefore, you can install Wheels 64 as an upgrade to any version of GEOS 64 from V1.3 through V2.0. Likewise, Wheels 128 may be installed as an upgrade to GEOS 128 from V1.4 through V2.0. Just about every beige-colored Commodore PC came with GEOS 64 V1.3. There is still a big advantage to upgrading to Wheels from GEOS 2.0 as opposed to upgrading from an earlier version, though. GEOS 2.0 came with much improved versions of geoWrite and geoPaint. So, you might still want to buy your own copy of GEOS 2.0 just to get the newer versions of those applications.

Throughout the years, we've had new pieces of hardware released and methods employed to be able to use these products with our GEOS systems. But there's nothing like having the support for the hardware built directly into the operating system, rather than patching it up to do the job. That was one of the goals of the Wheels operating system, to better utilize what we have available to us today and to provide better support for the future.

These computers will be around for a while longer yet, and your new Wheels system picks up where GEOS left off.

A Thumber's Guide to Wheels 4.4

There was a stated intention from the author / creator to make a "A Thumber's Guide to GEOS". The following section is an attempt to create that guide from a combination of sources including all the way down to walking through code in the debugger. This section is far from complete but it is the hope that it will grow over time and will someday be "done".

Welcome to the undiscovered country of the internals of Wheels 4.4.

Environment

Terms

REU	RAM-Expansion Unit.
RBAM	REU Bank Allocation Map.
Bank	REU 64K bank.

Constants

```
; Kernal groups
KG0_REU      = 0

;--- run flags for GetNewKernal
NO_RUN       = %01000000    ; $40
RUN_FIRST    = %00000000    ; $00

;--- REU
MAX_RPART    = 8            ; Maximum Number of REU Partitions
```

Equates

```
    kgBase          = $5000
      kgJMPTbl      = $5000
      rBAMCRC       = $5024
      reuHDR        = $5025                  ; REU Header Block
        reuBAM      = $5025                  ; Permanent RBAM
        rBAM        = $5045                  ; RBAM Workspace
        rPART       = $5065                  ; Partition ID's
        rPARTSB     = $506D                  ; Partition Starting Bank table
        rPARTSZ     = $5075                  ; Partition Size Table
        rPARTNM     = $507D                  ; Partition Name table. 16 characters + NULL
```

Internal Equates

```
        ; Only Valid in 4.4
        bitMskTbl   = $522D                  ; Bit Position for each bank/8.
        rCurPart    = $5373                  ; Current Partition Nbr
```

variables

Kernal

```
numDesktops    =  $88a6   ; Current Nesting Level of Desktops
dtDrive        =  $8868   ; DeskTop Drive
dtPartition    =  $8869   ; DeskTop Partition
dtType         =  $886a   ; DeskTop Type?
```

version `$41 - $44 V4.1 to V4.4 for Wheels.`
`$11 - $20 V1.1 to 2.0 for Berkeley GEOS.`

Driver

```
dirHeadTrack   =  $905c   ; the current directory header track.
dirHeadSector  =  $905d   ; the current directory header sector.
cableType      =  $9073   ; With the HD, if bit 7 is set, the parallel cable is being used
                          ; With the RAM1581 drivers, if bit 7 is set, then this is a RamLink
                          ; If cleared, then it's a normal RAMdisk running in an REU

ckdBrdrYet     =  $9074   ; $FF means GetNxtDirEntry is working in the system directory.
                          ; (read only)

driverVersion  =  $904f   ; ($51) driverVersion will be V5.1 ($51) or greater.


openError      =  $9071   ; 1    Set By OpenDisk to show the status of the last disk opened.
dir3Head       =  $9c80   ; to be used by the disk drivers only. Resides within each driver.
                          ; ($9c80-$9d7f)
```

to make the driver behave as if OpenDisk has ran on the drive and was successful

```
LoadB openError,#0        ; FIXME. What other values are there? Would it not normally be 0 anyway?
```

Kernal Jump Table

```
InitMachine        = $c2fe
GEOSOptimize       = $c301
DEFOptimize        = $c304
DoOptimize         = $c307
NFindFTypes        = $c30a
ReadXYPot          = $c30d
MainIRQ            = $c310
ColorRectangle_W   = $c313      ; Original name is ColorRectangle
i_ColorRectangle   = $c316
SaveColor          = $c319
RstrColor          = $c31c
ConvToCards        = $c31f
```

Driver Jump Table

```
ddriveType         = $904e
driverVersion      = $904f
OpenRoot           = $9050      ; OpenRoot-OpenDirectory: This is just like in GateWay for
compatibility
OpenDirectory      = $9053      ; open any directory on a native partition
GetBamBlock        = $9056
PutBamBlock        = $9059
dirHeadTrack       = $905c
dirHeadSector      = $905d
curBamBlock        = $905e
lastBamByte        = $905f
lastBamSector      = $9060
bamAltered         = $9061
highestTrack       = $9062
GetHeadTS          = $9063      ; Get the Track and Sector of the directory header.
PutHeadTS          = $9066
GetLink            = $9069
GetSysDirBlk       = $906c
startBank          = $906f
startPage          = $9070
pagesUsed          = $9071
```

Structures

```
 .ramsect    kgBase
 kgJMPTbl:    .block $21    ; 5000-5020                                  $5000
 rMR          .block 3      ; 5021-5023  "MR#"                           $5021
 rBAMCRC:     .block 1      ; checksum of reuHDR                         $5024
 reuHDR:      .block $E0    ; 5025-5104                                  $5025
```

```
.ramsect     reuHDR
                           ; 5025-5104
 reuBAM:     .block $20    ; Permanent RBAM                              $5025
 rBAM:       .block $20    ; RBAM Workspace                              $5045
 rPART:      .block 8      ; Partition ID's                              $5065
 rPARTSB:    .block 8      ; Partition Starting Bank table #             $506D
 rPARTSZ:    .block 8      ; Partition Size Table                        $5075
 rPARTNM:    .block $88-1  ; Partition Name table                        $507D
                           ; Names are 16 characters + NULL
 reuHDREnd:  .block 1      ; Last byte of rPARTNM                        $5104
                           ; Last byte of reuHDR                         $5104
```

Internal Structures

```
;--- Only Valid in 4.4
.psect       $522D
   bitMskTbl:      byte         $80, $40, $20, $10, $08, $04, $02, $01
```

Memory Maps

Local RAM

Kernal Group load area. Occupied as a result of a call ***GetNewKernal***

$5000	kgWorkspace	$1000	Total Area occupied by a loaded Kernal group
$5000	kgJMPTbl	$24	Kernal Group Jump Table Entries
$5024	reuHDR	$E1	REU Header Block
$5105	?	?	Unknown
$51B6	Start of Kernal Code		
$522D	bitMskTbl	8	$80, $40, $20, $10, $08, $04, $02, $01

All Kernal Groups by Name

KG0_REU	= 0	
AllocAllRAM	$5006	Allocate all available banks in REU
AllocRAMBlock	$5009	Allocate a Bank in the REU.
DelRamDevice	$501B	Remove a partition from REU
FreeRAMBlock	$500C	Release a Bank in the REU
GetRAMBam	$5000	Load Ram Expansion 'BAM'
GetRAMInfo	$500F	Get information on available REU Banks
PutRAMBam	$5003	Update REU BAM
RamBlkAlloc	$5012	Allocate a Block of REU Banks.
RamDevInfo	$501E	Get information on a REU partition.
RemoveDrive	$5015	Remove a RAM drive from the REU
SvRamDevice	$5018	Create a partition in the REU.

KG1_DEVICE	= 1
DevNumChange	$5000
SwapDrives	$5003

KG2_DISK	= 2
DBFormat	$5003
DBEraseDisk	$5009
EraseDisk	$500C
FormatDisk	$5006
NSetGEOSDisk	$5000

KG3_READFILE	= 3
OReadFile	$5000

KG4_WRITEFILE	= 4
OWriteFile	$5000

KG5_DIRECTORY	= 5	
ChDiskDirectory	$5009	This routine may be safely called from within another dialog box. This works just like ChPartition and ChSubdir other than the ability to call it from a dialog box. It will start the user out in the appropriate mode.
ChgParType	$5000	Call this with **r4L** holding either a 1 for a native type or a 4 for a 1581 type, and the appropriate driver will be invoked for this CMD device. It's rare that this routine is ever needed. It's mainly used by the operating system when switching partitions.

ChPartition	$5003	This will call up a system dialog box, allowing the user to select a different partition or subdirectory. This starts out by displaying a list of the currently available partitions. This may not be called from within another dialog box unless the programmer is familiar with how to preserve dialog box variables.
ChPartOnly	$501E	This is just like ChPartition, except that it doesn't allow the user the ability to change subdirectories. Only a partition can be selected. All other aspects are the same as ChPartition.
ChSubdir	$5006	This is similar to ChPartition, except that it starts out by displaying a list of the subdirectories within the current directory. The user is also given the ability to change partitions.
DownDirectory	$5015	This will open a subdirectory within the current directory if it's a native mode partition or native RAMdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory. Call this with dirEntryBuf containing the directory entry of the desired subdirectory.
FindRamLink	$5027	This will search for a RamLink. If found, x will hold the "real" device number of the RamLink, not the drive letter assignment as seen by the user. This allows the programmer to address the RamLink through direct DOS calls if needed. If x=0, then there is no RamLink on the system. This routine works whether the RamLink is configured for use by the operating system or not.
GetFEntries	$500C	
GoPartition	$5018	Select a partition on a CMD device. Call this with x holding the number of the desired partition. The partition must be either a 1581 or native mode type. The correct driver will be installed by this routine and the current directory on the desired partition will be opened. The directory is whichever one is listed by the drive's own DOS as the current directory.
TopDirectory	$500F	Open the root directory of the current drive. If it's a native mode partition or native RAMdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory.

UpDirectory	$5012	open the parent directory of the current drive if it's a native mode partition or native RAMdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory.

KG6MKDIR	= 6	
MakeDirectory		$5000
MakeSysDir		$5003

KG7VALDISK	= 7	
ValDisk		$5000

KG8CPYDISK	= 8	
CopyDisk		$5000
TestCompatibility		$5003

KG9COPY	= 9	
CopyFile		$5000

KG10DESKTOP	= 10		
InstallDriver		$5006	Install Printer Driver / Input Driver

```
;--- Put Driver Directory Entry into dirEntryBuf then the following code installs the driver.
      lda   #(KG10DESKTOP | NO_RUN)
      jsr   GetNewKernal
      jsr   InstallDriver
      jmp   RstrKernal
```

FindAFile		$500C
FindDesktop		$5009
NewDesktop		$5000
OEnterDesktop		$5003

KG11TOBASIC	= 11	
KToBasic		$5000

OpenDirectory: (C64, C128) **9053**

Function: open any directory on a native partition.

Parameters:

r1L	TRACK	— Track of subdir to open.
r1H	SECTOR	— Sector of subdir to open.

Destroys: (unknown).

Returns: nothing.

Description: **OpenDirectory** opens any directory on a native partition. Load **r1L**, **r1H** with the track and sector of the subdir and call **OpenDirectory**. This does basically the same thing as **OpenDisk**.

Example:

See also:

GetHeadTS: (C64, C128) **9063**

Function: Get the track and sector of the directory header.

Parameters: none.

Destroys: (unknown).

Returns:

r1L	track of directory header.
r1H	sector of directory header.
r2L	current partition number.

Description: **GetHeadTS** is contained in every Wheels disk driver.

Example:

See also:

GetNewKernal: (C64, C128) $**9D80**

Function: Load Kernal Group.

Parameters: a GROUPNBR to load | RUNFLAG
RUNFLAG Bit 6 of a.
1 Selected Kernal Group Swapped into memory at 5000-5FFF.
0 First Routine in group executed. (Kernal Group swapped back).

Destroys: (unknown).

Returns: varies depending on RUNFLAG and GROUPNBR.

Description: **GetNewKernal** allows access to the Extended Kernal available in 4.4.

If RUNFLAG is 0 **GetNewKernal** behaves as a far jsr to the first routine in the Kernal Group. Performing the following...

Swap the extended Kernal group into memory.
Execute the first routine in the group.
Swap the Kernal back out of memory.
Control is returned to the caller.

If RUNFLAG is set:

Extended Kernal is swapped into memory at 5000-5FFF.
Control is returned to the caller.

(Kernal will remain in memory until a call to **RstrKernal** to swap it back).

Note: Kernal Groups are loaded from the Last REU bank which is reserved exclusively for the 4.4 Kernal.

Note: Caller cannot be in the Range 5000-5FFF as that address range is swapped out with the Kernal Group

Note: Loading **KG0_REU** also loads in the **reuHDR**.

Example:

```
KG0_REU   = $00
NO_RUN    = %01000000
RUN_FIRST = %00000000

LoadREUGrp:
   lda #KG0_REU|NO_RUN
   jmp GetNewKernal
```

See also: **RstrKernal**

RstrKernal: (C64, C128) $**9D83**

Function: Unload Extended Kernal group.

Parameters: none.

Destroys: a.

Returns: nothing.

Alters: Memory area from 5000-5FFF is restored to its previous contents.

Description: **RstrKernal** is used to restore the memory area 5000-5FFF after using **GetNewKernal** to load in an extended Kernal Group.

Example:

```
KG0_REU   =$00
NO_RUN    =%01000000
RUN_FIRST =%00000000

.ramsect
   freeBanks:
          .block 1

.psect

GetBanksFree:
   lda    #KG0_REU|NO_RUN  ; select REU Group and don’t execute 1st
   jsr    GetNewKernal     ; load in Kernal group
   jsr    GetRAMInfo       ; Call Kernal Group function to get
                           ; number of free REU banks
   MoveB  r4H,freeBanks    ; save the result
   jmp    RstrKernal       ; remove Kernal Group, restoring 5000-5FFF
                           ; to its previous contents
```

See also: **GetNewKernal**

AllocAllRAM: (C64, C128) $**5006**

Function: Allocate all available banks in REU.

Parameters: none.

Uses:	**reuHDR**	REU Header Block.
Alters:	**rBAM**	All bits in **rBAM** reset to mark banks as used.
	rBAMCRC	New CRC generated.

Returns: nothing.

Destroys: a, y.

Description: **AllocAllRAM** allows a program to allocate all banks in the REU for their own use.

Since **AllocAllRAM** returns no information on what blocks got allocated, an application must already have knowledge of what banks were available prior to calling **AllocAllRAM**.

Note: No permanent changes are made. Call to **PutRAMBam** is required to update changes to **reuBAM**. **RstrKernal** must then be called to make the changes permanent.

Note^{4}:

Example:

See also: **PutRAMBam**, **AllocRAMBlock**, **RamBlkAlloc**.

AllocRAMBlock: (C64, C128) $**5009**

Function: Allocate a Bank in the REU.

Parameters: **r6L** RBANK — REU Bank Number.
valid range: 1...ramExpSize-2 (byte).

Uses:

ramExpSize	Number of banks in REU.
rBAM	RBAM Workspace.
rBAMCRC	Check sum of **reuHDR**.

Alters:

rBAMCRC	New CRC generated.
rBAM	*RBANK* BAM bit reset to used.

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

Destroys: a, x, y.

Description: **AllocRAMBlock** allocates a single bank in the REU in **rBAM**

Note: No permanent changes are made. Call to **PutRAMBam** is required to update changes to **reuBAM**. **RstrKernal** must then be called to make the changes permanent.

Note: Bank $00 and the last bank are reserved for the Kernal and are already allocated.

Note: BAD_BAM is returned in x for the following conditions.

1. *RBANK* has already been allocated.
2. *RBANK* is not a valid bank #.

Note[4]:

Example:

See also: **PutRAMBam**, **AllocAllRAM**, **RamBlkAlloc**.

DelRamDevice: (C64, C128) $**501B**

Function: Remove a partition from REU.

Parameters: y PARTITION — Partition Nbr: 1-8 (Max of 8)

Uses:

rBAM — RBAM Workspace.
rPARTSB — Partition Starting Bank table.
rPARTSZ — Partition Size table.
rPARTNM — Partition Name table.

Calls:

GetRAMBam	Reset **rBAM** to match **reuBAM**
PutRAMBam	Save **rBAM** workspace to **reuBAM**

Alters:

rBAM — RBAM Updated to reflect freed Banks.
rPARTSB — Partition Starting Bank entry set to $00
rPARTSZ — Partition Size entry set to $00
rPARTNM — Partition Name entry NULLed.
reuBAM — Permanent RBAM updated with changes in **rBAM**.
rBAMCRC — Check sum of **reuHDR**

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

DEV_NOT_FOUND

1. *PARTITION* = 0
2. *PARTITION* > 8
3. **rPARTSB**,*y* = $00 (Selected Partition is not in use).

Destroys: a, y, **r1**, **r3H**, **r6L**.

Description: **DelRamDevice** removes a partition from the REU.

Note: No permanent changes are made. **RstrKernal** must be called to make the changes permanent.

Example:

See also: **SvRamDevice**, **RamDevInfo**.

FreeRAMBlock: (C64, C128) $**500C**

Function: Release a Bank in the REU.

Parameters: **r6L** RBANK — REU Bank Number.
valid range: 1 ... ramExpSize-2 (byte).

Uses: **ramExpSize**.	Number of banks in REU
rBAM	RBAM Workspace

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

Alters: **rBAM**	— RBAM Updated to reflect freed Banks.
rBAMCRC	— Check sum of **reuHDR**

Destroys: a, y.

Description: **FreeRAMBlock** Release a Bank in the REU.

FreeRAMBlock resets the BAM bit for *RBANK* in **rBAM**, marking it as free.

Note: The Only Error Checking is for BAD_BAM

1. *RBANK* is not zero. To Protect against freeing Kernal bank 0
2. *RBANK* is < ramExpSize. To Protect against freeing Kernal in REU last block.
3. *RBANK* is currently allocated.

There are no checks to see if the Bank is assigned to an active partition.

Note[4]:

Example:

See also: **AllocAllRAM**, **AllocRAMBlock**, **RamBlkAlloc**.

GetRAMBam: (C64, C128) $**5000**

Function:	Reset **rBAM** to match contents of **reuBAM**	
Parameters:	none.	
Uses:	**reuBAM**	Permanent RBAM.
Alters:	**rBAM**	RBAM Updated to reflect freed Banks.
	rBAMCRC	Check sum of **reuHDR**.
Returns:	nothing.	
Destroys:	a, y.	

Description: **GetRAMBam** copies the contents of **reuBAM** to **rBAM** to give a fresh working copy of the RBAM. This should be used as a rollback step if an error occurs while processing changes to the **rBAM**. After calling **GetRAMBam** the **rBAM** workspace will be complete reset and all prior changes are lost.

Note: RBAM is a type of structure. It is synonymous with the BAM on a CBM disk.

Example:

See also: **PutRAMBam**

GetRAMInfo: (C64, C128) $500F

Function: Get information on available Banks.

Parameters: nothing.

Uses: **r6**, **r9L**.

ramExpSize	Number of banks in REU.
reuBAM	Permanent RBAM.
rBAM	RBAM Workspace.

Calls: **GetRAMBam**.
RamBlkAlloc.

Alters:

rBAMCRC	New CRC generated.
rBAM	Reloaded from **reuBAM**

Returns:

r2L	# of consecutive free Banks. 0 = no banks available.
r3L	# of starting bank pointing to the largest free area.
r4H	# of free 64KB banks.

Destroys: a, y, **r2H**, **r3H**, **r6**, **r9L**

Description: **GetRAMInfo** gives a snapshot of available RAM that a program can use the REU. **GetRAMInfo** loads **r2L** with **ramExpSize-1** and loads **r3L** with $00 and calls **RamBlkAlloc**.

The **RamBlkAlloc** routine will allocate the largest available contiguous memory area and pass its parameters upon return. Upon returning from the **RamBlkAlloc** routine, it then calls the **GetRAMBam** routine to undo any changes that **RamBlkAlloc** routine may have made. Next, it recomputes the RAM BAM's checksum value and stores it back onto $5024. The resulting parameters are then returned back to the calling program.

Example:

See also:

PutRAMBam: (C64, C128) $**5003**

Function: Update REU BAM.

Parameters: none.

Uses:	**rBAM**	— RBAM Workspace.
Alters:	**reuBAM**	Permanent RBAM updated with changes in **rBAM**.
	rBAMCRC	Check sum of **reuHDR**.

Returns: nothing.

Destroys: a, y.

Description: **PutRAMBam** applies the changes made to **rBAM** to the RBAM in **reuBAM**.

Note: No permanent changes are made. **RstrKernal** must be called to make the changes permanent.

Example:

See also: **GetRAMBam**

RamBlkAlloc: (C64, C128) $**5012**

Function: Allocate a Block of REU Banks.

Parameters: **r2L** BANKS — Number of contiguous Banks needed (byte).
r3L START — Starting Bank Number (byte).

Uses: **rBAM** — RBAM Workspace.

Calls:	**GetRAMBam**	Reset **rBAM** to match **reuBAM**.
	PutRAMBam	Save **rBAM** workspace to **reuBAM**.
Alters:	**rBAM**	RBAM Updated to reflect allocated blocks.
	rBAMCRC	Check sum of **reuHDR**.

Returns: x error ($00 = no error).
INSUFF_SPACE
Failed to Allocate Requested Banks.

r3L Starting Bank Number of allocated *BANKS*.

Destroys: a, y, **r2H**, **r3H**, **r6**, **r9L**.

Description: **RamBlkAlloc** allocates a *BANKS* sized block of contiguous Banks in the REU. **RamBlkAlloc** searches for the contiguous block of Banks starting at *START* Bank Number.

if *START* = 0 **RamBlkAlloc** will start searching from the first Bank and will search the entire REU for block of Banks large enough fulfill the request.

Note: Wheels will not allow other programs to allocate these banks once they have been allocated. Make sure to free any Banks allocated when the application is done using them.

Example:

See also: **AllocAllRAM**, **AllocRAMBlock**.

RamDevInfo: (C64, C128) $**501E**

Function: Get information on a REU partition.

Parameters: y = Partition Nbr. 1-8 (Max of 8).

Uses:

rPART	Partition ID table.
rPARTSB	Partition Starting Bank table.
rPARTSZ	Partition Size table.
rPARTNM	Partition Name table.

Returns:

r2L	Size of Partition in Banks.
r3L	Starting Bank Number.
r7L	Partition ID.
r1	Pointer to Partition Name.

Destroys: a.

Description: **RamDevInfo** gets stats about a particular partition.

Example:

See also: **SvRamDevice**, **RamDevInfo**.

RemoveDrive: (C64, C128) $**5015**

Function: Remove a RAM drive from the REU.

Parameters: nothing.

Uses

Uses	**driveType**	type of drive to open.
	numDrives	number of drives in the system.
	curDrive	device number of the active drive.
	curType	currently active drive type.
	curDevice	currently active device.
	ramBase	RAM bank for each disk drive to use.

Calls: **SetDevice**.
PurgeTurbo.

Returns: nothing.

Destroys: a, y, **r4L**.

Description: **RemoveDrive** checks **numDrives** to ensure that there are at least two drives running. No sense in deleting the only drive in a system! Using the drive number passed in **r4L**, it calls **SetDevice** & **PurgeTurbo**. Next, it zeroes out the corresponding **driveType** entry and the **ramBase** entry in these two tables. It then zeroes out **curType**, **curDrive**, **curDevice** and finally decreases the value found in **numDrives** by one.

This has the effect of removing a RAM drive from the Wheels OS system. It does not actually remove the RAMdisk in a physical sense. It is just that some pointers indicating the existence of a RAMdrive is simply wiped out.

Note[4]: Interestingly enough, there is no corresponding AddDrive entry. Maybe this routine is contained in the Toolbox instead and is not in the Group 0 section of the Wheels OS Kernal.

Example:

See also:

KG0_REU

SvRamDevice: (C64, C128) $**5018**

Function: Create a Partition in the REU.

Parameters:
r0 NAME — pointer to a 16-byte null-terminated partition name (word).
r2L BANKS — # of contiguous Banks needed (byte).
r3L START — Starting Bank Number (byte).
(0 = Let the Kernal decide which starting Bank to use).
r7L PARTID — ID number, can be any number less than 128 (byte).
(Any number higher than 128 designates a RAMdisk).
y PARTNBR — Partition Nbr. 1-8
0 Let the Kernal decide the partition number.

Uses:
reuBAM — Permanent RBAM.
rBAM — RBAM Workspace.
rPART — Partition ID table.
rPARTSB — Partition Starting Bank table.
rPARTSZ — Partition Size table.
rPARTNM — Partition Name table.

Calls: :
GetRAMBam Reset **rBAM** to match **reuBAM**.
RamBlkAlloc. Allocate *BANKS*
PutRAMBam Save **rBAM** workspace to **reuBAM**

Alters:
rBAM Updated to reflect Allocated Banks.
reuBAM Permanent RBAM updated with changes in **rBAM**.
rPART Partition ID set to *PARTID.*
rPARTSB Partition Starting Bank entry set to *START.*
rPARTSZ Partition Size entry set to *BANKS.*
rPARTNM Partition Name entry set to *NAME.*
rBAMCRC Updated Check sum of **reuHDR**.

Returns: x error ($00 = no error).
FULL_DIRECTORY ($04)
1. *PARTION* > 8
2. *PARTIONNBR* = 0 and all partitions are in use.
3. **rPARTSB**,y != 0

Destroys: a, y, **r0**, **r1**, **r2L**, **r3L**, **r7L**.

Description: **SvRamDevice** sets up a partition in the REU. The partition will be reserved by the Kernal and survive various computing sessions. Once created, a program can simply reuse that partition over and over instead of individually allocating and freeing up expansion RAM memory every time it boots.

REU partitions are preserved for use in future computing sessions.

See also: **DelRamDevice**, **RamDevInfo**.

DevNumChange:

Function: Get information on a REU partition.

Parameters: x =

Uses:

Returns:

Destroys:

Description:

Example:

See also:

CopyFile: **09**:**5000**

Function: Copy File from Current Directory to Destination

Parameters: none.

Uses:

Alters:

Returns: x error ($00 = no error).
$FF= Destination file existed. Copy Aborted.

Destroys: $7900-$7fff

Description:

Note:

Example:

Copy File:

COPYING FILES and CHANGING PARTITIONS

Currently active directory is the source directory.

1. **dirEntryBuf** is loaded with the directory entry of the source file.

2. **r0** DESTNAME — Destination filename.

   This can also be used to duplicate a file. If the source and destination directories are the same and **r0** points to a filename that is different from the source Filename, then a file duplication will take place within the same directory.

   If the names are different and the source and destination directories are also different, then the file will be copied and the copy will receive a new name. Whatever **r0** points to is what the destination file will be named.

3. **r3L** bit 7 = 0 Copy file into the main directory.
   = 1 Copy file to the system directory.
   bit 6 = 0 Use Multi Drive Copier.
   = 1 Use Single Drive Copier.
   User will be prompted to insert the source and destination disks as needed.
   bits 0-5 contains the destination drive number (8-11).

4. **r2L** - If the destination is a partitionable device, this is loaded with the destination partition number. It's safe to set this even on non-partitionable devices such as the 1541. Therefore, it's not necessary to determine the drive type prior to copying a file.

5. **r1L**, **r1H** - track and sector of the destination directory on a native mode partition or RAMdisk. These values are also meaningless on a 1541, 71, or 81 type directory.
   If the destination turns out to be the system directory of the main directory, then a simple directory entry "move" will take place.

6. **r3H** - set bit 7 to force a "move" instead of a copy, if desired,

   provided the destination is within the same partition as the source. If this is not the case, then a copy is performed instead of a move.

7. **r2H**- bit 7 if clear, will replace the file on the destination if one of the same name as what **r0** points to exists.

   If set, then the file will be skipped if one of the same name is found.

   RETURNS x - $00 = no error.
   $FF means a file of the same name existed (bit 7 of **r2H** was set). The copy did not proceed.

Sample copy file use:

```
fNameBuffer:
       .block 17

CopyAFile:
     PushB  curDrive
     jsr    OpenDisk
     PushB  r1L
     PushB  r1H
     jsr    GetHeadTS                 ; get the current partition number into r2L
     PushB  r2L
     lda    #8
     jsr    SetDevice
     jsr    OpenDisk
     LoadW  r6,#fNameBuffer
     jsr    FindFile                  ; load dirEntryBuf
     LoadW  r0,#fNameBuffer           ; destination name
     PopB   r2L                       ; destination partition
     PopB   r1H                       ; destination dir sector
     PopB   r1L                       ; destination dir track
     PopB   r3L                       ; destination drive
     lda    #9                        ; Kernal group 9
     jmp    GetNewKernal              ; run the first routine in group 9
```

Important: There is no error handling in this example to keep the sample focused. In real world code "txa bne" would be after the I/O calls to handle errors.

Miscellaneous

Find a RamLink. Upon return, x can be tested. 0 = No RamLink found. >0 = "real" device number of RamLink

```
      WhereIsRamLink:
            lda   #(5|NO_RUN)
            jsr   GetNewKernal
            jsr   FindRamLink
            jmp   RstrKernal
```

Pop up a dialog box allowing the user to select a partition or subdirectory.

```
thisPartition:
      .block 1
thisTrack:
      .block 1
thisSector:
      .block 1


  GetNewDirectory:
      lda   #(5|NO_RUN)
      jsr   GetNewKernal
      jsr   ChDiskDirectory
      jsr   RstrKernal
      jsr   GetHeadTS
      MoveB r2L,thisPartition   ; save the user's choice of partitions
      MoveB r1L,thisTrack       ; save the header track
      MoveB r1H,thisSector      ; Save the header sector
      rts
```

Note: As sample code blocks are discovered they are added here until a more permanent home is found for them.

Appendix

A: Atoms

Introduction to atoms.

Building Blocks of an Application

The smallest level of GEOS application is the core instruction set of the 6502 processor. This simplistic instruction set can make creating large bug free applications difficult. If not managed in an organized way an application can quickly become larger and more complex than it should be.

There is a hierarchy of building blocks that leads to a solid base for creating applications.

Large vlir application	Collection of selected process and library blocks. Glued together by opcode and macro. UI added for user control.
process / Small App	Collection of libraries. Glued together by opcode and macro. Has data structures. Document Type Handlers etc...+UI elements
library	Collection of common atoms. May contain small data structures. Attached in the linking phase to the application or to a process library.
atom	logic block. Normally only uses CPU registers. May use simple data structures. Fully tested and optimized. (IR/RE)
macro	single statement. Represents a common multi opcode task. Increase readability of source (IR). Reduce opportunities for coding errors. (RE)
opcode	>> 6502 Instructions. virtually infinite ways of combining into an application. << Steps must be taken to reduce the amount of an application that is created from this level.

This appendix focuses on the atom level.

atom

atoms are small reusable blocks of code. Depending on the atom and on situational needs, an atom may be used as a subroutine or as inline code. When used as a subroutine they should be grouped together with other related atoms and kept in a .rel library.

Creating an atom

- atoms should be small, less than a page of code.
- Base level atoms will only use processor registers.
- Mid-level atoms use pseudoregisters and possibly depend on the existence of a named global variable. This is the case with multiple atoms working around the same global. Globals for atoms should be located in zero page when possible.
- High level atoms will call lower level atoms.
- Getting an atom to work is only the first stage in creating a new atom.

Optimizing

Time investment in reducing the size in bytes and/or the execution time in cycles is well spent here. The situation will dictate which of the two is more important between size and speed. Inner loops are all about speed and spending extra bytes for speed is often the correct path. At the top level UI it is all about space and trading speed for size is the correct path.

Some atoms may perform double duty:

1. General purpose routine that saves space in the main body of the application by being a jsr target. With every call saving total space used.

2. Provide faster inner loop logic by using inline:
   Saves: a jsr and an rts. 3 + 6 cycles.
   Costs: The size of the atom in bytes.

This dual personality can be handled in 2 different ways:

1. Copy the body of the atom into the inner loop. This is the easiest and quickest way to deploy.

2. Have an outer shell for the atom in the first atom name file. It only contains the following pseudo code:

```
Atom:                                   ; Name of the Atom routine
       .include  _atomname_i            ; Include the core of the atom
       rts                              ; End the routine with standard rts.
```

   Then the inner loop at the point of insertion will have the following:

```
code here ...
.include _atomname_i
code here ...
```

Libraries

The convention for storing an atom is _atomname with no extension. It is not intended to ever directly assemble this file. The atoms will then be included mostly into the main source files of libraries with .include _atomname The libraries are then assembled which generate .rel files. These are in turn used when linking applications that use them. The more of your codebase that lives in libraries means more pre tested and pre optimized code. This results in much faster application builds.

If you use 1541/71/81 type devices for storage then the disk will be the logical storage unit for a library. For devices that support subdirectories the libraries should be separated by directory. Within each library area will be the _source files for the atoms, the global .include files and the library .include files. These files will then be assembled generating a .rel file which is the final form of the library.

The rest of this section will provide some actual atomic pieces from geoProgrammer applications and from geoWrite as well as other atoms created for geoProgrammer' 2.1.

quick reference

Categories

Identifier	Category
bit	bit operations
br	branching
cmp	Comparisons
flow	Alters flow of logic
math	Math
hw	Hardware
size	Code Base Reduction
text	Text Operations
util	Utility
conv	Conversion

Sources

Identifier	Source
gP1	geoProgrammer1.1
gD	geoDebugger
gW	geoWrite 128
gP'	geoProgrammer' 2.1
	Other sources will be added as used.

by name

BCD2Bin	Convert Binary Coded Decimal to binary value	gP'	conv
Bin2bin	Convert single Binary ASCII character ['0','1'] to a binary value	gP	util
Bin2Bin	Convert Binary ASCII string ['0','1'...] to a binary word value	gP'	util
Bookmark	Create a bookmark of current stack location	gD	flow
DiskName	Get pointer to disk name in **r0**.	gP'	util
DoDlg	Wrapper for **DoDlgBox** to reduce codebase size	gP	size
Hex2Nib	Convert nibble with hex ASCII value to a binary nibble.	gP'	util
Hex2NibF	Convert nibble with hex ASCII value to a binary nibble. Fast Version. Does not convert character to uppercase first.	gP'	util
Lower	Convert character to lowercase.	gP'	text
Nib2Hex	Convert nibble to ASCII hex character.	gP'	util
SwZp	Swap Kernal I/O zp area with buffer area	gP	util
SwpNib	Swap Upper and Lower nibbles in byte	gP'	util
Upper	Convert character to uppercase	gP'	text

BCD2Bin: conv

Function: Convert single BCD (Binary Coded Decimal) value (00-99) to a binary value.

Parameters: accumulator NBR — Number to process

Returns: binary value in a

Destroys: **r15L**.

Description:

```
binary value = (n10*10) + n1.

(n10*8) + (n10*2) = n10*10

(n10*16)/2 + (n10*16)/8 = (n10*8) + (n10*2)
```

Note: Especially useful for time /dates that are stored in BCD format.

Filename: _ BCD2Bin
Source: geoProgrammer'

Example:

```
lda    #$31
jsr    BCD2Bin
                      ; a now = $1F
```

```
BCD2Bin:                                                     ; a = $31
        pha                        ; Save BCD Value
        and     #%11110000         ; Get 10s place and divide by 2
        lsr     a                  ;
        sta     r15L               ; r15L=(n10*16)/2                ; r15L = $18
        lsr     a                  ;
        lsr     a                  ;
        adc     r15L               ;
        sta     r15L               ; r15L=r15L +(n10*16)/8          ; r15L = $1E ($18 + $06)
        pla                        ; Restore BCD Value
        and     #%00001111         ; Add Ones place
        adc     r15L               ; r15L=r15L+n1                   ; r15L = $1F ($1E + $01)
        rts
```

Bin2bin: text

Function: Convert single Binary ASCII character ['0','1'] to a binary value.

Parameters: accumulator CHAR — Character to process

Returns: If *CHAR* is a valid binary character;
binary value of that character
carry flag = 0
otherwise
carry flag = 1

Destroys: nothing.

Description:

Note: Tuned by removing clc.

Filename: _ Bin2bin
Source: geoProgrammer'

Example:

```
Bin2bin:
        cmp     #'0'                ; if char < ASCII zero then invalid
        bcc     98$
        cmp     #'1'+1              ; if character > '1' then invalid
        bcs     99$
        sbc     #'0' – 1            ; Convert to ASCII value†
        clc
        rts
98$     sec
99$     rts
```

†Carry is known to be clear at the sbc. -1 accounts for that.

See also:

Bin2Bin: text

Function: Convert Binary ASCII string ['0','1'...] to a binary word value.

Parameters: **r0** PTR — Pointer to string to process.
x REG — Pointer to zero page register for the result.

Returns: y = index to terminating character.
Carry Flag = 0 No Error.
binary value of binary string in *REG*
Carry Flag = 1 Error.
a = TRUE Value Overflowed the word.
a = FALSE Invalid Character in string.

Destroys: nothing.

Description: Convert a stream of binary digits to a binary value and save the result in *REG*. The string does not have to be null terminated since this routine can handle processing a value that exists inside a stream of characters. This is useful when parsing a file. Each time a % character is encountered you would call this routine to get the value that occurs after the %. Then **AddYW r0** to bump the zp pointer to the next byte in the string.

Filename: _ Bin2Bin
Source: geoProgrammer'

Example:

```
       Bin2Bin:
              ldy    #0
              sty    zpage,x
              sty    zpage,x+1
       10$
              lda    (r0),y
              jsr    Bin2bin             ; convert ASCII to bit value
              bcs    50$                 ; if carry set we had a non-binary character else
              ror    a                   ;    put new bit into the carry flag
              rol    zpage,x             ;    rotate the result to the left with new incoming bit
              rol    zpage,x +1          ;    now rotate high-byte
              bcs    99$                 ;    if carry is set then we overflowed the word
              iny                        ;    else loop
              bcc    10$                 ; (carry always clear here. bcc instead of bra saves
                                         ; one byte and two cycles)
       ;---
       50$    jsr    IsAlphaN            ; if the ending character is alpha numeric then we have
              bcs    98$                 ;    an invalid binary string
       90$    clc                        ; our part of the string is done
              rts                        ; exit

       98$    lda    #FALSE              ; invalid character in stream
clda   99$,          #[TRUE              ; value over flowed word
              sec
              rts
```

See also:

Bookmark: flow

Function: Save last return address and stack position before the return address.

Parameters: nothing.

Returns: nothing.

Alters:

bm_rts	—	Return Address saved as the bookmark
bm_Stack	—	Stack address to restore too when returning to bookmark

Destroys: nothing.

Description: Provides the ability to reset Program Logic and stack to a previous state.

Filename: _Bookmark
Source: geoDebugger

Example:

```
              jsr Bookmark              ; bookmark here as the start of a process
      ;---    do long process
      ;---    If error during process then return to Bookmark using bm_rts and bm_Stack.
```

```
.ramsect
      bm_rts:
            .block 2
      bm_Stack:
            .block 1

Bookmark:
      php                               ; save status register, a and x
      pha
      PushX
      tsx                               ; transfer SP to x
      inx                               ; point X to return address of caller
      inx
      inx
      inx
      lda   $100,x                      ; save return address + 1 to bm_rts
      add   #1
      sta   bm_rts
      inx
      lda   $100,x
      sta   bm_rts+1
      stx   bm_Stack                    ; save stack point before caller return address
      PopX                              ; restore status register, a and x
      pla
      plp
      rts
```

See also:

DoDlg: size

Function: stub to call **DoDlgBox** using a and x to point to dialog box.

Parameters: x DBH — High-byte of dialog box address.
a DBL — Low-byte of dialog box address.

Returns: Same as **DoDlgBox**.

Destroys: Same as **DoDlgBox**.

Description: Reduce foot print of an application that uses multiple dialog boxes.

Note: Normal way to call a dialog box is using:

```
LoadW r0,#dbTable         ; 8 byte Sequence
```

With **DoDlg** you send the dbTable like this:

```
ldx    #]dbTable          ; 2 bytes
lda    #[dbTable          ; 2 bytes
```

It takes 8 bytes to **LoadW r0**,#value but it only takes 4 bytes to lda and ldx with the dbTable address bytes.

It only requires 2 uses of **DoDlg** for it to cut a profit.

DoDlg size = 7 bytes.
Savings per use = 4 bytes.

Filename: _DoDlg
Source: geoProgrammer

Example:

```
...
ldx    #]dbTable                    ; load high-byte of dbTable address
lda    #[dbTable                    ; load low-byte of dbTable address
jsr    DoDlg                        ; put up the dialog box
...
```

```
DoDlg:
       stx    r0H                   ; store high-byte                2 bytes
       sta    r0L                   ; store low-byte of address      2 bytes
       jmp    DoDlgBox              ; transfer control to DoDlgBox   3 bytes
```

See also:

IsAlphaN: text

Function: Check if a character is alpha numeric.

Parameters: a CHAR — Character to check.

Returns: Carry = 0 Not Alpha Numeric
Carry = 1 Alpha Numeric

Destroys: nothing.

Description: test a for
 = Underscore or
(>='0' and <=9) or
(>='A' and <=Z) or
(>='a' and <=z)

Filename: _IsAlphaN
Source: geoProgrammer

Example:

```
  IsAlphaN:
       cmp    #'_'               ; underscores are OK
       beq    99$                ; exit w/carry set
       cmp    #'0'
       bcc    99$                ; if less than '0' then not alpha numeric. exit w/cc
       cmp    #'9'+1
       bcc    98$                ; if <= '9' then we have a number. exit w/cs.
       cmp    #'A'
       bcc    99$                ; if < 'A' then not alpha numeric. exit w/cc
       cmp    #'Z'+1
       bcc    98$                ; of <= 'Z' then alpha numeric. exit w/cs
       cmp    #'a'
       bcc    90$                ; if < 'a' then not alpha numeric. exit w/cc
       cmp    #'z'+1             ; if <= 'z' then alpha numeric. exit w/cs
       bcc    98$
       clc
       rts                       ; exit.
;---   At the atom level. 2 exits are ok from one routine if it saves cycles or bytes
;      in doing so. In this case "bcc 99$" could replace the rts but that would make this
;      exit take 2 extra cycles and 1 extra byte
  98$
       sec
  99$
       rts                       ; exit
```

See also:

Lower: text

Function: Convert character to lowercase.

Parameters: accumulator CHAR — Character to process.

Returns: If *CHAR* is a uppercase letter;
    return lowercase of that letter
otherwise return accumulator unchanged.

Destroys: nothing.

Description: range checking is performed on CHAR. Only valid Uppercase Alpha characters will be altered.

Note: Tuned by removing clc.

Filename: _Lower
Source: geoProgrammer'

Example:

```
Lower:
      cmp    #'A'                ; If character < 'A' then exit.
      bcc    14$
      cmp    #'Z'+1              ; If character > 'Z' then exit.
      bcs    10$
      adc    #('a'-'A')          ; †Convert to Lower Case.
10$
      rts
```

†*Carry is known to be clear at the adc. no need to clc prior to the adc.*

See also: **Upper**

Nib2Hex: text

Function: Convert nibble to ASCII value for a hex character

Parameters: accumulator NIBBLE — Binary value to convert

Returns: a ASCII character that represents the hex value of *NIBBLE*.

Destroys: nothing.

Description: Converts the low *NIBBLE* into an ASCII value. For speed and size there is no error checking to make sure the high nibble is 0.

Note: Tuned by removing clc.

Filename: _ Nib2Hex
Source: geoProgrammer'

Example:

```
;--- Speed Optimized Version. Best for inner loop use.
;    10 bytes: (0-9) 12 Cycles, (A-F) 13 Cycles*
Nib2Hex:
        cmp    #10                ; if nibble is less than 10 then
        bcs    80$
        adc    #'0'               ;    add value of ASCII zero
        rts                       ; exit
80$
        adc    #('A' – 10) -1     ; add offset to 'A' minus the base value of 10
        rts

;--- Size Optimized Version. Saves 1 byte over Speed optimized version.
;    Best for general purpose library use.
;    9 bytes: (0-9) 13 Cycles, (A-F) 14 Cycles*
Nib2Hex:
        cmp    #10                ; if nibble is greater than 9 then
        bcc    90$
        adc    ('A' – ('9'+1)) -1 ;    add offset to 'A' from '9' (7)†
90$
        adc    #'0'               ; add value of ASCII zero
        rts
```

†*Carry is known to be set at the adc. -1 accounts for that.*
**Plus 1 cycle if branch crosses a page boundary.*

```
;---    Smallest and slowest version. Best for byte conservation in non inner loops.
;       8 bytes: (0-9,A-F) 16 Cycles.
Nib2Hex:
        sed                       ; enter BCD mode
        clc                       ;
        adc #$90                  ; produces $90-$99 (C=0) or $00-$05 (C=1)
        adc #$40                  ; produces $30-$39 or $41-$46
        cld                       ; leave BCD mode
        rts
```

See also:

Hex2Nib:, **Hex2NibF**: text

Function: Convert nibble with hex ASCII value to a binary nibble.

Parameters: accumulator NIBBLE — Hex Character to process

Returns: If NIBBLE is a valid Hex character;
return binary value of NIBBLE in the accumulator.
Carry Flag = 0
otherwise return
Carry Flag = 1

Destroys: On error: a

Description: Use Hex2Nib when the case of the hex character is known. Use Hex2NibF when the characters are known to be upper case.

Note: Tuned by removing 2 sec instructions.

Filename: _ Hex2Nib
Source: geoProgrammer'

Example:

```
Hex2Nib:
        jsr    Upper              ; convert character to upper case

;--- enter here if the data is known to be already upper case
Hex2NibF:
        cmp    #'0'               ; if the char is < ASCII zero then invalid character.
        bcc    99$
        cmp    #'9'+1             ; if char is <= '9' then convert number
        bcc    90$
        cmp    #'A'               ; if char is < ASCII A then invalid character
        bcc    99$
        cmp    #'F'+1             ; of char is <= 'F' then convert letter
        bcs    99$
        sbc    #('A'-10) -1       ; subtract offset from 'A' to the base value of 10†
        clc
        rts
90$
        sbc    #'0' -1            ; subtract offset from '0' to the base value of 0†
        clc
        rts
99$
        sec
        rts
```

†Carry is known to be clear at the sbc. -1 accounts for that without having to spend a byte and 2 cycles to use a sec instruction.

See also:

DiskName: util

Function: Get Pointer to Diskname in **r0**.

Parameters: accumulator DRIVE — Device Number of Desired Drive.

Returns: **r0** Contains pointer to DiskName of *DRIVE*.

Destroys: y.

Description: Since the Disk Names are not stored in a contiguous space, they cannot be retrieved by a simple index lookup. This is an efficient size and speed method to get the pointer to the name.

Note: For speed and size there is no error checking on validity of *DRIVE*.

Filename: _ DiskName
Source: geoProgrammer'

Example:

```
        lda    curDrive             ; get current drive number
        jsr    DiskName             ; get ptr to the disk name in r0
        ...
```

```
DiskName:
        pha                         ; preserve a on stack
        and    #%00000111           ; normalize drives to offset to 0. e.g. drive 8 is now 0
        asl    a                    ; set index to drive name table
        tay
        MoveB  "T_DskNm+1,y",r0H    ; save pointer to name in r0
        MoveB  "T_DskNm,y",r0L
        pla                         ; restore a
        rts                         ; exit

;---    table of pointers to disk name buffers
T_DskNm:
        .word  DrACurDkNm
        .word  DrBCurDkNm
        .word  DrCCurDkNm
        .word  DrDCurDkNm
```

See also:

SetSys: util

Function: Set sysType flag for runtime decisions based on current hardware.

Parameters: nothing.

Returns:

a	current value of sysType. z status flag = 0 if GEOS version < 1.3 N=1 C128; N=0 C64 V=1 80 Col Mode; V=0 40 Col Mode
x	current value **version**

Alters sysType. Application variable (byte).

Destroys: nothing.

Description: combines multiple system checks into 1 flag byte.
Examples of logic checks:

```
        bbsf   7,sysType, 128$
        bbrf   7,sysType, 64$
128$
        bbrf   6,sysType, 40$       ; 40 col mode
        bbsf   7,sysType, 80$       ; 80 col mode

        lda    sysType
        beq    Old version < 1.3    ; exit out if your app does not support old Kernal
```

Filename: _SetSys
Source: geoProgrammer'

Example:

```
SYS_OLD        =$0000
SYS_64         =%0001
SYS_128        =%10000000
SYS_VDC        =%11000000
SetSys:
        lda    #0
        ldx    version
        cpx    #$13
        bcc    90$
        bbrf   7,c128Flag, 64$
        lda    graphMode
        and    #%10000000
        lsr    a
        ora    #%10000000
64$
        ora    #%00000001
        cpx    $40
        bcc    90$
        ora    #%00000010
90$
        sta    sysType
        bit    sysType              ; set Flags based on OS and video mode
        rts
```

See also:

SwpNib: util

Function: Swap Upper and Lower nibbles in byte.

Parameters: accumulator TARGET — byte value to nibble swap.

Returns: a Upper and Lower nibbles are swapped.

Destroys: nothing.

Description: Fast and compact way to swap nibbles. One useful case for this is reversing foreground and background colors in a byte.

Note:

Filename: _ SwpNib
Source: geoProgrammer'

Example:

```
        lda    screencolors        ; get current default screen colors
        jsr    SwpNib              ; invert colors
        sta    screencolors        ; set new default colors
        ...
```

```
    SwpNib:
        asl    a                   ; shift a left
        adc    #$80                ; adc %10000000 to a. b7 is now bit 0, b6 now in carry
        rol    a                   ; rotate a left. b7 and b6 are now bits 1 and 0
        asl    a                   ; repeat the process for the other 2 bits
        adc    #$80
        rol    a
        rts

.if 0
    Break down of the logic.
        asl    a                   ; shift a left.
                                   ; carry flag = b7
                                   ; bit 0 = 0

        adc    #$80                ; adc %10000000 to a
                                   ; bit 0 = carry flag
                                   ; carry flag = b6

        rol    a                   ; rotate a left. b7 and b6 are now bits 1 and 0

        asl    a                   ; repeat the process for the other 2 bits
        adc    #$80
        rol    a
        rts
.end if
```

See also:

SwZp: util

Function: Swap Kernal I/O zero page area with buffer area.

Parameters: nothing.

Alters: $80-FA is swapped with buffer at rSwZp.

Returns: I/O zero page area swapped with buffer.

Destroys: nothing.

Description: Allows greatly increasing the number of bytes available in Zero Page space for an application. To use this the application must call this as its first step during initialization and as its last step on shutting down. For the life of the application it must call this routine prior to any API calls that access the serial bus (IE anything related to drives or printers) to put the Kernals I/O zero page space back. After the I/O is done then call this again to put the application zero page back.

Note: This method is used by all of the Berkeley applications. Also note that none of the Berkeley applications treat the application registers a0-a9 as a0-a9. They use the zero page space from $70-7F (APP_ZPL), and $FB-FE (APP_ZPH) as .zsect space with variables declared and used in this space having varying sizes as needed.

Filename: _SwZp
Source: geoProgrammer'

```
;---   Define the size of area to use. FA-80 is the entire I/O zp space. Constant declaration and
;      ramsect is to be provided in the main body of the application.
SWZP_SZ      = APP_ZPH - APP_ZIO       ; ($FB-$80 = 123 bytes)

.ramsect
      rSwZp:
             .block SWZP_SZ
.psect
      SwZp:
             php                       ; save registers
             pha
             PushX
             PushY
             ldx    #SWZP_SZ -1        ; set size of area to swap. get from 80-FA

      10$                              ; while x > 0 loop
             lda    APP_ZIO,X          ;    load a from zp
             ldy    rSwZp,X            ;    load y from buffer
             sty    APP_ZIO,X          ;    store y to zp
             sta    rSwZp,X            ;    store a to buffer
             dex
             bpl    10$                ; end loop

             PopY                      ; restore registers
             PopX
             pla
             plp
             rts                       ; $80-$FA now available to use
```

SwZp[†]

Example:

```
.zsect APP_ZPL                  ; APP_ZPL = $70
       zTS:   .block 2

.ramsect
       rSwZp: .block 123

.psect
       Init:
              jsr    SwZp
              ....   Do Rest of the initialization.

       DoSomeIO:
              LoadW  r0,#diskBlkBuf
              MoveW  zTS,r1
              jsr    SwZp
              jsr    GetBlock
              jsr    SwZp
              txa
              bne    handle_error
              ...

       Shutdown:
              jsr    SwZp
              jmp    EnterDeskTop
```

[†] **Important**: **SwZp** is impleminted in all of the large Berkeley applications. This is by far the most important piece of logic that has been extracted from the Berkeley applications. Having a block of .zsect that spans from $70-FE grants an application a vast improvement in efficiency it would not otherwise have. This method can be further evolved to have the **SwZp** integrated into the calls. e.g. **GetBlock** becomes sGetBlock. This version of **GetBlock** can then also do a txa before returning so z contains the success / failure status. This reduces the footprint of this logic into a size profit.

See also:

Upper: text

Function: Convert character to uppercase.

Parameters: accumulator CHAR — Character to process.

Returns: If *CHAR* is an lowercase letter;
returns uppercase of that letter
otherwise returns accumulator unchanged

Destroys: nothing.

Description: range checking is performed on CHAR. Only valid lower-case alpha characters will be altered.

Filename: _Upper
Source: geoProgrammer'

Example: **KeyTrap**.

```
Upper:
      cmp   #'a'                ; if character < 'a' then exit
      bcc   90$                 ;
      cmp   #'z'+1              ; if character > 'z' then exit
      bcs   90$                 ;
      sbc   #('a'-'A') -1       ; convert to upper-case†
90$
      rts
```

†Carry is known to be clear at the sbc. The -1 is to compensate for the additional +1 subtraction caused by the cleared carry. This uses assembler time to save runtime bytes (1) and cycles (2) by removing the need for the sec instruction.

See also: **Lower**

B: Examples

This section contains all the code examples from the book chapters and from the GEOS 2.0 API.

The examples are organized into the following categories.

atoms	Small reusable blocks of code. They may be used as subroutines or as inline code depending on the atom and on situational needs.
dialog boxes	Everything to do with Dialog Boxes.
disk	Disk I/O.
drivers	Input and Print Drivers. Includes multiple examples of each.
graphics	Covers all graphical output to the screen.
hardware	Code specific to the C64 and/or C128 hardware.
icons & menu	...
keyboard	
math	...
memory	...
mouse & sprite	...
text	Text output to screen and text input from keyboard.
utility	Miscellaneous routines.

Note: This section is for GEOS 2.0. Wheels 4.4 has its own examples section within **Chapter 21 Wheels Kernal 4.4**.

KeyTrap:

```
.psect
.include _upper

       T_Action:
              'A','B','C','D'          ; keyboard commands to act on. case insensitive
       T_ActL:                         ; low pointer table to action handlers
              .byte  [SetDrv8
              .byte  [SetDrv9
              .byte  [SetDrv10
              .byte  [SetDrv11
       T_ActH:                         ; high pointer table to action handlers
              .byte  ]SetDrv8
              .byte  ]SetDrv9
              .byte  ]SetDrv10
              .byte  ]SetDrv11
       T_ACTCNT=*-T_ActH

       Init:
              LoadW  keyVector,#KeyTrap
              rts

       KeyTrap:                        ; routine hooked into keyVector
              lda    menuNumber        ; check current menu level
              bne    90$               ; ignore keys while menus down
              lda    keyData           ; get Keypress and
              jsr    Upper             ; convert it to uppercase
              ldy    #T_ACTCNT-1       ; search action table for a hit
       10$
              cmp    T_Action,y
              beq    20$
              dey
              bpl    10$
       90$
              rts                      ; no action found for press. exit
       20$
              ldx    T_ActH,y
              lda    T_ActL,y          ; action found
              jmp    CallRoutine       ; execute the handler

       SetDrv8:
              lda    #8
clda   SetDrv9,      #9
clda   SetDrv10,     #10
clda   SetDrv11,     #11
              jsr    SetDevice         ; set device to user selected number
              jsr    OpenDisk          ; open the disk
              jmp    ErrHndlr          ; generic Error Handler
                                       ; displays error dialog or does nothing on no error
```

ImpBin:

```
.if 0
```

Function: Convert an ASCII Binary string to a word value.

Parameters: nothing.

Alters: zVal zero page word to hold result

Returns:

r0 Pointer to string.
y index to string terminator.
Carry Flag 0 = No error.
1 = Invalid Binary String.

Description: Simple use of Bin2Bin. Converts a simple null terminated string to a binary value.

```
.endif
```

```
.zsect APP_ZPL                          ; APP_ZPL = $70
      zVal:
             .block 2                   ; zero page variable to hold result

.psect

.include _Bin2Bin

      bstr:
             .byte  "00101101",NULL

   GetOneVal:
      LoadW  r0,#bstr                   ; set pointer to string
      ldx    #zVal                      ; set zp pointer
      jsr    Bin2Bin                    ; convert binary string to word value
      bcs    HandleError
      rts                               ; return with result in zVal
```

getFileDB:

```
.if 0
```

Function: Get geoWrite File name from user using **DBGETFILES**.

Parameters:	**r7L**	**FILETYPE**	— GEOS File type (NULL=Any File Type).
	r5	BUFFER	— pointer to buffer to return filename in.
	r10	PERMNAME	— Pointer to permanent name string (NULL=Any Data Type).

Description: A DB table from geoWrite for putting up a Dialog Box for loading an existing document. If Open was selected from the **openBoxDB**, then the **getFileDB** is displayed to help the user choose from the available files.

```
.endif
.ramsect
      fileName:
            .block 17                 ; buffer to hold filename
      diskName:
            .block 20                 ; disk name

.psect
      gwClass:
            .byte  "Write Image",NULL  ; only want geoWrite files

      tOnDisk:                        ; disk name header text
            .byte  BOLDON,"On disk:",PLAINTEXT,NULL
+++++++++

      getFileDB:
            .byte  DEF_DB_POS | 1     ; standard DB with shadow pattern #1
                                      ; the GetFile's box works well inside a standard DB
            .byte  DBTXTSTR           ; display a text string
            .byte  DBI_X_2 * 8 – 6    ; x-offset in pixels for text. (=17*8-6 = 130)
            .byte  TXT_LN_1_Y – 6     ; y-offset
            .word  tOnDisk            ; "On disk:"

            .byte  DBTXTSTR           ;
            .byte  DB_ICN_X_2 * 8 - 6 ; x-offset in pixels for text. (=17*8-6 = 130)
            .byte  TXT_LN_2_Y – 12    ;
            .word  diskName           ; buffer already loaded with disk name
            .byte  DBGETFILES,4,4     ; get filename box command
            .byte  OPEN,DBI_X_2,DBGF_Y_0             ; [Open] Icon
            .byte  DBUSRICON,DBI_X_2,DBGF_Y_1        ; [Drive] Icon
            .word  uiDrive            ; disabled when only 1 Drive is on the system
                                      ; allows changing drives
            .byte  CANCEL,DBI_X_2,DBGF_Y_3           ; [Cancel] Icon
      dbDisk:                                        ; [Disk] Icon
            .byte  DISK,DBI_X_2,DBGF_Y_2
                                      ; disabled on Drive geoWrite Loads from
                                      ; allows changing disks while dialog is open
            .byte NULL                ; end of DB definition
```

GetWorkFile:

```
.if 0
```

Function: Get geoWrite File name from user using **DBGETFILES**.

Parameters:
r7L **FILETYPE** — GEOS File type. (NULL=Any File Type)
r5 BUFFER — pointer to buffer to return filename in.
r10 PERMNAME — Pointer to permanent name string. (NULL=Any Data Type).

Returns: TRUE	File selected. (With selected Filename saved in fileName buffer).
FALSE	User canceled out of the dialog.

Description: The filename box displays a list of filenames. Any filename can be selected by the user. It is then copied into *BUFFER* pointed to by **r5**. If **r10** is not null, it points to *PERMNAME* which contains the permanent name string, e.g., "Paint Image" taken from the File Header. In this case, only geoPaint documents will be displayed for selection. If there are more files than can be displayed within the box, pressing the scroll arrows that appear under the filename box will scroll the filenames up or down. Max of 16 files supported.

```
                                                                      .endif
.ramsect
      fileName:
            .block 17                 ; buffer to hold filename

.psect
      gwClass:
            .byte  "Write Image",NULL  ; only want GeoWrite files

      gwDB:
            .byte  DEF_DB_POS | 1      ; default Size with a solid shadow
            .byte  OK                  ; display [OK] Icon
            .byte  DBI_X_0             ; left-side
            .byte  DBI_Y_2             ; bottom row
                                       ; display [CANCEL] Icon. right/bottom using compact layout.
            .byte  CANCEL,DBI_X_2,DBI_Y_2
            .byte  DBGETFILES,4,4      ; display file selection box @offset x4,y4
                                       ; 4 pixels in from the left, and down from top
            .byte  NULL                ; end of dialog Box table

;---   Display a dialog box to get the user selected name of a GeoWrite File.

      GetWorkFile:
            LoadW  r5,#fileName        ; buffer to save selected filename
            LoadB  r7L,#APPL_DATA      ; want data files
            LoadW  r10,#gwClass        ; only show GeoWrite files
            LoadW  r0,#gwDB            ; point r0 to our dialog box table
            jsr    DoDlgBox            ; display the dialog box
            CmpBI  r0L,#CANCEL         ; set return value based on user
            beq    99$                 ; icon selection
            lda    #[TRUE              ; user pressed [OK]
clda  99$,         #FALSE              ; user pressed [CANCEL]
            rts
```

openBoxDB:

`.if 0`

Function: Dialog Table to display geoWrite's Open Dialog Box.

Description: A table from geoWrite for putting up Dialog Box for selecting a new document, opening an existing document, or quitting geoWrite altogether. Also includes supporting data structures.

`.endif`

```
ICOLOFF   = 2                         ; number of cards to offset icons from DB.
                                      ; (card is 8 pixels wide and 8 pixels tall)
ICOWDTH   = 6                         ; width of icons in cards
ICOTXTOFF = 7                         ; number of pixels to offset text after icons
ICOTXTXP  = (ICOLOFF + ICOWDTH) * 8 + ICOTXTOFF

       openBoxDB:
              .byte  DEF_DB_POS | 1   ; standard DB with shadow pattern #1

              .byte  DBTXTSTR         ; display a text string
              .byte  TXT_LN_X         ; place it at the standard x-offset (=16 pixels)
              .byte  2*8              ; y-offset in pixels from top of box
              .word  selectOptionTxt  ; pointer to the message "Please Select Option:"

              .byte  DBUSRICON        ; programmer defined icon
              .byte  2                ; x-offset in cards for left-side of icon
              .byte  3*8              ; y-offset in pixels for top of icon
              .word  uiCreate         ; pointer to the icon table for the [Create] icon

              .byte  DBTXTSTR         ;
              .byte  ICOTXTXP         ; place to the right of [Create] icon.
              .byte  3 * 8+10         ; y-offset: 10 below top of the [Create] icon
              .word  tNewDoc          ; pointer to text for "new document"

              .byte  OPEN             ; standard system OPEN icon
              .byte  2, 6*8           ; x-offset in cards
                                      ; y-offset, 3 Cards below [Create] icon

              .byte  DBTXTSTR         ;
              .byte  ICOTXTXP         ;
              .byte  6 * 8+10         ; y-offset: 10 below Top of [Open]
              .word  tExisting        ; pointer to "existing document"

              .byte  DBUSRICON        ;
              .byte  2                ; x-offset in cards
              .byte  9*8              ; y-offset in pixels
              .word  uiQuit           ; pointer to [Quit] icon

              .byte  DBTXTSTR         ;
              .byte  ICOTXTXP         ;
              .byte  9 * 8+10         ; y-offset for text after [Quit]
              .word  tDesktop         ; " to deskTop"
              .byte  NULL             ; end of table
```

```
       uiDrive:
              .word  iDrive,NULL
              .byte  SYSDBI_WIDTH,SYSDBI_HEIGHT
              .word  UADrive

       selectOptionTxt:
       ;--- The select option message with embedded BOLDON and PLAINTEXT bytes
       ;--- to turn boldface on and off
              .byte  BOLDON, "Please Select Option:",PLAINTEXT,NULL

       tNewDoc:
              .byte  "new document",NULL ; note each of these strings are null terminated

       tExisting:
              .byte  "existing document",NULL

       tDesktop:
              .byte  "to deskTop",NULL

       uiCreate:                        ; user icon definition table
              .word  iCreate            ; address of picture data for the [Create] icon
              .word  NULL               ; not used
              .byte  6                  ; icon is 6 cards wide
              .byte  16                 ; 16 pixels tall
              .word  UACreate           ; pointer to the service routine which creates the
                                        ; file, and returns to the application

       uiQuit:                          ; icon definition table
              .word  iQuit              ; address of picture data for the [Quit] icon
              .word  NULL               ; not used
              .byte  SYSDBI_WIDTH       ; icon is 6 Cards wide
              .byte  SYSDBI_HEIGHT      ; 16 pixels tall
              .word  UAQuit             ; pointer to the service routine which quits to the
                                        ; deskTop

       ;--- service routine for the [create] icon
       UACreate:
              lda    #OK                ; indicate icon number as if OK icon was activated.
clda   UAQuit,       #CANCEL            ; return value for [quit]
              sta    sysDBData          ; store icon number before RstrFrmDialog call
              jmp    RstrFrmDialog      ; exit from DB

       iCreate:
```

```
       iQuit:
```

CheckDiskSpace:

```
.if 0
```

Description: Ensures that the current disk has 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.

Parameters: **r2** BYTES — number of bytes we need.

Returns: x = If not enough space, returns an INSUFF_SPACE error.

x = 0 If there is enough space.

z Flag follows value of x.

Destroys: a, y, **r2**, **r3**, **r8**, **r9**.

```
.endif
NO_ERROR      = 0
BLOCK_SIZE    = $100

;--- Number of bytes that can be stored in each block on the disk.
;    accounts for two-byte track/sector link
BLOCK_BYTES   = BLOCK_SIZE – 2

CheckDiskSpace:
    bweq    r2,90$                    ; exit if no BYTES to check
    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: remainder in r8
    bweq    r8,10$                    ; branch if no remainder bytes
    IncW    r2                        ; otherwise 1 more block needed
                                      ; r2 = BLOCKS needed to hold BYTES
10$                                   ; get number of free blocks on disk
    LoadW   r5,#curDirHead            ; point to directory header
    jsr     CalcBlksFree              ; r4 <- free blocks on disk
    CmpW    r2,r4                     ; are there enough free blocks?
    bgt     99$                       ; if not, assume. correct, branch
90$
    ldx     #NO_ERROR                 ; otherwise, no error
    rts
99$
    ldx     #INSUFF_SPACE             ; not enough space
    rts                               ; exit
```

DeleteDirEntry:

```
.if 0
```

Description: Remove a directory entry without actually freeing any blocks in the file.

Parameters: **r0** FILENAME — pointer to FILENAME to delete.

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

Destroys: a, y, **r0**, **r9**.

```
.endif
```

```
.ramsect
    rFileName:
            .block 17
    nullTrScTable:
            .block 256

;--- sample caller
CallDelDir:
    LoadW   r0,#rFileName
    jsr     DeleteDirEntry
    rts

;--- Pass r0 pointer to filename
DeleteDirEntry:
    LoadW   r3,#nullTrScTable          ; pass dummy table
    jmp     FastDelFile                ; will only delete the directory entry
```

;--- 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**.

GrabSomeBlocks:

`.if 0`

Function: **GrabSomeBlocks** — allocate enough disk blocks to hold data in buffer.

Parameters: nothing.

Returns: Carry flag:
1 = Error
0 = success.

x = Error Nbr if Carry is set,
or 0.

`.endif`

```
    K = 1024                            ; one kilobyte

.ramsect
   buffer:
         .block 5*K -1                  ; 5K buffer
   bufferE:
         .block 1                       ; end of 5k Buffer

   BUF_SIZE = (bufferE – buffer)+1      ; size of buffer

.psect

   GrabSomeBlocks:
         LoadW  r2,#BUF_SIZE            ; number of bytes to allocate
         LoadW  r6,#fileTrSecTab        ; buffer to build out table
         jsr    BlkAlloc                ; allocate the blocks
         txa                            ; check status
         bne    99$                     ; and exit on error
         ;--- more code here
   90$
         ldx    #0
         clc                            ; success exit
         rts
   99$
         sec                            ; error exit
         rts
```

MyFreeBlock:

`.if 0`

Function: **MyFreeBlock** — allocate specific block in BAM with any CBM device driver. And any GEOS version.

Parameters: **r6L** track number.
r6H sector number.

Note: **FreeBlock** was not added to the GEOS jump table until v1.3.

`.endif`

```
MyFreeBlock:
    lda      version                ; check GEOS version number
    cmp      #$13                   ; version Less than 1.3?
    bcc      10$                    ;
    jmp      FreeBlock              ; if not, go through jump table

10$
    jsr      FindBAMBit             ; returns r8H = mask for BAM byte
                                    ;      r7H = offset to track
                                    ;      x = offset into bam
                                    ;      a = masked value
    bne      99$                    ; if 1, then not allocated, give error
    txa
    bne      99$
    lda      r8H                    ; get mask
    eor      curDirHead,x           ; flip BAM bit to make available
    sta      curDirHead,x           ;
    ldx      r7H                    ; one more free block
    inc      curDirHead,x           ;
    ldx      #NO_ERROR              ; ($00)
    rts

99$
    ldx      #BAD_BAM
    rts
```

MyPutBlock:

`.if 0`

Function: **MyPutBlock** — Write **diskBlkBuf** to disk.

Parameters:

r1L	track number.	
r1H	sector number.	
r4	address of block to write.	
verify	FALSE (0);	do not verify
	!= FALSE;	verify after write

Note: If you have multiple blocks to write you should write the entire chain and then verify the chain. See **WriteBlock** description for more information.

`.endif`

```
.ramsect
    nextTrack:        .block     1
    nextSector:       .block     1
    outbuffer:        .block     $FE
    track:            .block     1
    sector:           .block     1
    verify:           .block     1
.psect

CallMyPutB:
       LoadW  r4,#outBuffer-2
       MoveB  track,#r1L
       MoveB  sector,#r1H
       LoadB  verify,#[TRUE
       jsr    MyPutBlock
       bcs    99$
       rts                       ; return good status in carry
99$
       ...                       ; error handler or let caller handle error
       rts

MyPutBlock:
       jsr    EnterTurbo         ; go into turbo mode
       txa                       ; check for error in X
       bne    99$                ; branch if error found
       jsr    InitForIO          ; prepare for serial I/O
       jsr    WriteBlock         ; primitive write block
       txa                       ; set status flags
       bne    99$                ; branch if error found
       lda    verify             ; check verify flag
       beq    80$                ; branch if not verifying
       jsr    VerWriteBlock      ; verify block we wrote
       txa                       ; set status flags
       bne    99$                ; branch if error found
80$
       jsr    DoneWithIO         ; restore after I/O done
       clc
       rts                       ; no errors
99$
       jsr    DoneWithIO         ; restore after I/O done
       sec
       rts                       ; error status exit
```

MyReadBlock:

`.if 0`

Function: **MyReadBlock** — Read sector from disk into **diskBlkBuf**.

Parameters:

r1L	track number.
r1H	sector number.
r4	address of block to read into.

Description: Demonstrates use of very-low level disk primitives.

`.endif`

```
.ramsect
         nextTrack:  .block    1
         nextSector: .block    1
         outbuffer:  .block    $FE
         inbuffer:   .block    $100
         track:      .block    1
         sector:     .block    1
         verify:     .block    1

.psect

         CallMyPutB:
            LoadW  r4,#inBuffer
            MoveB  track,r1L
            MoveB  sector,r1H
            jsr    MyReadBlock
            bcs    99$
            rts                       ; return good status in carry
         99$
            ...                       ; error handler or let caller handle error
            rts

         MyReadBlock:
            jsr    EnterTurbo         ; go into turbo mode
            txa                       ; check for error in X
            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
                                      ; (x is preserved in DoneWithIO)
            txa                       ; get error result of ReadBlock
            bne    99$                ; branch if error found
         90$
            clc                       ; carry cleared when flowing through here
         99$
            sec                       ; carry set when branch to 99$ occurs
            rts
```

MySetGDirEntry:

`.if 0`

Function: This routine duplicates the function of the Kernal's **SetGDirEntry** for demonstration purposes. It shows examples of the following routines:

BldGDirEntry
GetFreeDirBlk
PutBlock

Parameters: Same as **SetGDirEntry**.

Destroys: Same as **SetGDirEntry**.

`.endif`

```
DIRCOPYSIZE = 30                          ; size of directory entry for copy
TDSIZE = 5                                ; number of bytes in time/date entry

MySetGDirEntry:
    jsr       BldGDirEntry                ; build directory entry for GEOS file
    jsr       GetFreeDirBlk               ; get block with free directory entry
                                          ; block number of block in r1
    txa                                   ; test for error code
    bne       99$                         ; if error, exit...
    AddYWS    diskBlkBuf,r5               ; add offset into diskBlkBuf for dir entry
                                          ; and put result in r5
    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       99$                         ; if error, exit
    clc
    rts                                   ; success exit
99$
    sec
    rts                                   ; error exit

TimeStampEntry:
    ldy       #(OFF_YEAR+TDSIZE)-1        ; offset to time/date stamp
10$
    lda       dirEntryBuf,y               ; get the year/month/day/hour/minute
    sta       (r5),y                      ; store in dir entry
    dey                                   ;
    bpl       10$                         ; loop until done
    rts
```

MySetNextFree:

`.if 0`

Purpose: Get next free block. If no block found retry from first of disk.

Parameters:

r3L	START_TR	— start allocating from this track (byte).
r3H	START_SC	— start allocating from this sector (byte).

Returns: x error ($00 = no error)
INSUFF_SPACE

Destroys: a, y, **r6-r7**, **r8H**.

Description: Since **SetNextFree** in 1541 and 1571 drivers only searched from the current block to the end of the disk, the possibility exists that a free block lies somewhere on a previous, still unchecked disk area. The following alternative to **SetNextFree** will circumvent this problem.

`.endif`

```
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    #INSUFF_SPACE            ; check for no blocks
      beq    10$                      ; start from beginning if none
90$   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.
.if VER>=2
10$   lda    curType                  ; Current type is already saved for us
      and    #%00111111               ; Mask off RAM and shadow drive flags.
      cmp    DRV_1581
      bge    90$                      ; 1581 and all drivers since, restart the search internaly
      ldx    #1
      stx    r3L                      ; always track 1
      dex                             ;
      stx    r3H                      ; and sector 0
      jmp    SetNextFree
.else
10$   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
      lda    curType
      and    #$0F                     ; searching scheme
      cmp    DRV_1581
      bne    20$
      ldx    #39                      ; 1581 counts down on inner (39-1)
20$   stx    r3L                      ; track number
      jmp    SetNextFree
.endif
```

NewAllocateBlock:

```
.if 0
```

Function: NewAllocateBlock — allocate specific block in BAM with any CBM GEOS device driver.

Parameters: **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**.

```
.endif
NewAllocateBlock:
    ldy      curDrive                ; get current drive
    lda      driveType-8,y           ; get drive type
    and      #%00001111              ; 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      99$                     ; if zero, then it's not free
    lda      r8H                     ; get bit mask for BAM
    eor      #$FF                    ; convert to clearing mask
    and      curDirHead,x            ; and with BAM byte to clear
                                     ; bit and show as allocated
    sta      curDirHead,x            ; and store back.
    ldx      r7H                     ; get base of track's entry
    dec      curDirHead,x            ; dec #free blocks this track
    ldx      #NO_ERROR               ; show no error
    rts                              ; exit
99$
    ldx      #BAD_BAM                ; show error — already in use
    rts                              ; exit

;--- Example Caller Routine:
.ramsect
    diskBlock: .block 2

.psect

CallNewAlloc:
    MoveW    diskBlock,r6            ; block to allocate
    jsr      NewAllocateBlock        ; (see above)
    cpx      #BAD_BAM                ; BAD_BAM means block in use
    beq      95$                     ; branch if block already in use
    txa                              ; check for other error
    bne      99$                     ; branch if error
    ;--- code to handle newly allocated block goes here

95$                                  ; block was not free...
    ;--- code to handle block already allocated goes here

99$
    jmp      MyDiskError             ; call error handler with error in x
```

ReadAndDelete:

.if 0

Function: Read sequential file into memory and then delete it from disk.

Parameters: **r6** pointer to filename
r2 size of buffer (max size of file)

Returns: x error code.

Destroys: a, y, **r0-r9**.

Description: 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).

.endif

```
       ReadAndDelete:
          MoveW  r6,r0                  ; save pointer for FastDelFile
          jsr    FindFile               ; find file on disk
          txa                           ; set status flags
          bne    99$                    ; branch on error
          LoadW  r9,#dirEntryBuf        ; get directory entry
          jsr    GetFHdrInfo            ; get GEOS file header
          txa                           ; set status flags
          bne    99$                    ; branch on error
       ;---      check filetype and branch if VLIR
          CmpBI  fileHeader+OFF_GSTRUCT_TYPE,#VLIR
          beq    98$
          jsr    ReadFile               ; read in file
          txa                           ; else set status flags
          bne    99$                    ; branch on other error
       20$
          LoadW  r3,#fileTrScTab        ; track/sector table
          jsr    FastDelFile            ; file read OK, delete it!
          bne    99$
          clc                           ; carry clear and z=0
          rts                           ; good exit
cldxI  98$,      #STRUCT_MISMATCH       ; can't load VLIR
       99$
          sec                           ; carry set
          rts                           ; error exit with error in x
```

SaveRecord:

`.if 0`

Function: Append new record into an existing VLIR.

Parameters:

appendPoint	already set to the last VLIR record.
Filename	buffer populated with VLIR's filename.

Note: geoProgrammer does not support the * counter in .ramsect. The method below must be used when the assembler needs to calculate the size of a ramsect field.

`.endif`

```
NAME_LENGTH=17

.ramsect
      appendPoint: .block 1            ; record to append to
      filename:    .block NAME_LENGTH  ; hold null-terminated filename
      bufStart:    .block 1023         ; data buffer
      bufEnd:      .block 1            ; length of buffer
      BUFLENGTH = (BufEnd - BufStart)+1

.psect

SaveRecord:
      LoadW  r0,#filename              ; pointer to filename
      jsr    OpenRecordFile            ; open VLIR file
      txa                              ; check open status
      bne                              ; 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
      txa                              ; check point status
      bne    99$                       ; exit on error
90$                                    ; clean exit
      clc                              ; clear carry for all ok
      rts
99$                                    ; error handler
      sec                              ; set carry to show returning with an error
      rts
```

Joystick

```
.if 0
```

Function: Sample Joystick Driver from OGPRG

Files:

app.lnk	Linker file
app.hdr.s	Header file
app.driver.s	Driver source
app.Inc	Master Include. sends all symbols to debugger
app.con	Application constants
app.sym	Application symbols
app.mac	Application macros

Callable Routines:

o_InitMouse
o_SlowMouse
o_UpdateMouse

```
.endif
```

app.lnk

;**Function**: Linker file for Joystick Driver.
;
;**Filename**: app.lnk
;
;**Uses**: app.hdr.rel
; app.driver.rel
;
;**Callable Routines**:
; o_InitMouse
; o_SlowMouse
; o_UpdateMouse

```
.output      JOYSTICK
.header      app.hdr.rel
.seq

.psect       mouse_Base
app.driver.rel
```

app.hdr.s

```
.if 0
```

Function: Define File Header Block.

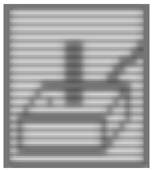
Filename: app.hdr.s Header File

Uses: app.con

Callable Routines:
None.

```
.endif
.if    Pass1
       .noeqin
       .include     app.con
       .eqin
.endif

.header                                 ; start of header section
       .word  0                         ; first two bytes are always zero
       .byte  3                         ; width in bytes
       .byte  21                        ; and height in scanlines of:
```



```
       .byte  $80|USR                   ; Commodore file type assigned to GEOS files
       .byte  INPUT_DEVICE              ; GEOS file type
       .byte  SEQUENTIAL                ; SEQ file structure
       .word  MOUSE_BASE                ; start address for saving file data
       .word  endJoystick               ; end address for saving file data (-1)
       .word  NULL                      ; not actually used (execution start address)

       ;--- 20 byte permanent name
       .byte "Input Drvr  V1.1",0,0,0,0

       ;--- 20 bytes for author name
       .byte  "Dave & Mike & PBM",NULL,0,0

.endh
```

app.driver.s

```
.if 0
```

Function: Main Source file for Joystick Driver.

Filename app.driver.s

Uses: app.inc

Callable Routines:

o_InitMouse
o_SlowMouse
o_UpdateMouse

Hardware: Joystick is read from **cia1prb** (Joystick Port 1 DC01). The bit values returned from this port are naturally set to 1. With the joystick at rest, the low 5 bits will always be %11111.

Only the b4-b0 are for the joystick with the following assignments.

b4 0 = Fire Button Pressed
Joystick Disk Directions
b3 0 = right
b2 0 = left
b1 0 = Down
b0 0 = Up

This port is shared with the keyboard. The keyboard has to be masked off prior to reading the joystick values. To do this write %11111111 to **cia1pra** (DC00) to select no keyboard rows to scan. Then any value read from **cia1prb** (DC01) will be from the joystick. This input from the Joystick is converted into the following output and saved into **inputData**.

inputData: 0-7 for moving, -1 for centered
joystick directions:

0 = right
1 = up & right
2 = up
3 = up & left
4 = left
5 = left & down
6 = down
7 = down & right
-1 = joystick centered

inputData+1: current mouseSpeed

```
                                                                 .endif
.include      app.inc
```

Jump Table

```
                                                                                   .if 0
Jump Table to Mouse Driver Routines
                                                                                  .endif
;--- Input   driver jump table

      jmp    o_InitMouse
      jmp    o_SlowMouse
      jmp    o_UpdateMouse

;--- Local variables:

fracXMouse:                          ; fractional mouse position
      .byte  0

fracYMouse:                          ; fractional mouse position
      .byte  0

fracSpeedMouse:                      ; fractional part of current mouse speed
      .byte  0

velXMouse:                           ; x component of current Speed
      .byte  0

velYMouse:                           ; y component of current Speed
      .byte  0

curMouse:                            ; current value of fire button
      .byte  0

currentDisk:                         ; current value of joystick
      .byte  0

lastKeyRead:                         ; for debouncing joystick
      .byte  0
```

o_InitMouse:

```
.if 0
```

Function:	External routine: This routine initializes the 'mouse'.
Called By:	At initialization EXTERNALLY.
Parameters:	nothing.
Alters:	**mouseXPos** **mouseYPos** starting position for the mouse.
Returns:	none.
Uses:	none.
Destroys:	a, x, y, **r0 - r15** Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
.endif
o_InitMouse:
      jsr   o_SlowMouse              ; do LoadB mouseSpeed,#0
      sta   fracSpeedMouse
      sta   mouseXPos
      sta   mouseXPos+1
      sta   mouseYPos
      LoadB diskData,#[-1            ; pass release
      jmp   ComputeMouseVels         ; store the correct speeds
```

o_SlowMouse:

```
.if 0
```

Function:	External routine: Called when menus are pulled down to slow the mouse.	
Called By:	External and Internal.	
Parameters:	none.	
Returns:	nothing.	
Alters:	nothing.	
Destroys:	a, x, y, **r0 - r15**	Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
                                                                                       .endif
o_SlowMouse:
      LoadB  mouseSpeed,#0            ; zero speed
SM_rts:
      rts
```

o_UpdateMouse:

```
.if 0
```

Function:	External routine:	This routine is called every interrupt to update the position of the pointer on the screen. First, the joystick is read and the mouse velocities are updated. The mouse position is then updated.
Called By:	Interrupt code.	
Parameters:	**mouseXPos** **mouseYPos**	current position for the mouse.
Returns:	**mouseXPos** **mouseYPos**	current position for the mouse.
Alters:		
Destroys:	a, x, y, **r0 - r15**	Even though this does not destroy everything here, another driver may destroy anything or nothing. So have to declare everything destroyed.

```
.endif
o_UpdateMouse:
	jsr	C64Joystick		; scan keyboard and update velocities
	bbrf	MOUSEON_BIT,mouseOn,SM_rts; if mouse off then don't update
	jsr	UpdMouseVels		; update mouse Speed & Velocities
	jsr	UpdXMouse		; update x-position of mouse
	;--- fall through to UpdYMouse
```

UpdYMouse:

```
.if 0
```

Function: Internal routine: Update the y-position of the mouse by adding in the velocity.
Called By: o_UpdateMouse.

Parameters: **mouseYPos** current position for the mouse.

Returns: **mouseYPos** updated.

Alters: none.

Destroys: a, y, **r1H**.

```
                                                                    .endif
UpdYMouse:
      ldy    #0                     ; assume positive velocity
      lda    velYMouse              ; get velocity
      bpl    10$
      dey                           ; if negative then sign extend with -1
10$
      sty    r1H                    ; store high-byte
      asl    a                      ; shift left thrice
      rol    r1H                    ; add fractional position
      asl    a
      rol    r1H
      asl    a
      rol    r1H
      add    fracYMouse             ; add fractional position
      sta    fracYMouse             ; store new fractional position
      lda    r1H                    ; get high-byte of velocity
      adc    mouseYPos              ; add position
      sta    mouseYPos
      rts
```

UpdMouseVels:

`.if 0`

Function:	Internal routine:	Update the velocity of the mouse by adding in the acceleration.
Called By:	o_UpdateMouse.	
Parameters:	none.	
Uses:	mouseSpeed	current mouse speed.
	velXmouse	current velocity.
	velYmouse	
Alters:	mouseSpeed	updated.
	velXMouse	
	velYMouse	
Destroys:	a, x y, **r0-r2**.	

`.endif`

```
UpdMouseVels:
      ldx   diskData                  ; get direction
      bmi   20$                       ; if release then branch
      CmpB  maxMouseSpeed,mouseSpeed  ; check for maximum speed
      blt   15$                       ; if max then do nothing
      AddB  mouseAccel,fracSpeedMouse ; add acceleration to speed
      bcc   30$
      inc   mouseSpeed                ; increment mouse speed if necessary
      bra   30$
15$
      sta   mouseSpeed
20$
                                      ; get Minimum speed and compare to current speed
      CmpB  minMouseSpeed,mouseSpeed  ; don't make less than minimum
      bge   25$                       ; if minimum > Current then branch
      SubB  mouseAccel,fracSpeedMouse ; subtract acceleration from speed
      bcs   30$
      dec   mouseSpeed
      bra   30$                       ; decrement mouse speed if necessary
25$
      sta   mouseSpeed
30$   ;--- fall through to ComputeMouseVels
      ;jmp  ComputeMouseVels          ; finally, based on direction and
                                      ; speed, calculate new mouse X & Y velocities
```

ComputeMouseVels:

`.if 0`

Function: Internal routine: Compute mouse velocity based on joystick direction.

Called By: Internal Only.

Uses:

diskData	– joystick direction.
mouseSpeed	– current mouse speed.

Alters: velXMouse, velYMouse - set depending of passed direction.

Destroys: a, x, y, **r0**–**r2**.

```
                                                                                                    .endif
ComputeMouseVels:
      ldx    diskData
      bmi    10$                         ; if release then handle
      MoveB  mouseSpeed,r0L              ; pass magnitude
      jsr    SineCosine
      MoveB  r1H,velXMouse
      MoveB  r2H,velYMouse
      rts
10$                                      ; released
      LoadB  velXMouse,#0                ; zero x-velocity
      sta    velYMouse                   ; zero y-velocity
      rts
```

UpdXMouse:

```
.if 0
```

Function: Internal routine: Update the x-position of the mouse by adding in the velocity.

Called By: o_UpdateMouse.

Uses: **mouseXPos** - current position for the mouse.

Alters: **mouseXPos** - updated.

Destroys a, x, y, **r11**– **r12L**.

```
.endif
UpdXMouse:
      ldy   #$FF                    ; assume negative
      lda   velXMouse
      bmi   10$                     ; if indeed negative then branch
      iny                           ; else sign extend with zero
10$
      sty   r11H
      sty   r12L
      asl   a                       ; multiply by 8 for permanent speed power of 3
      rol   r11H
      asl   a
      rol   r11H
      asl   a
      rol   r11H
                                    ; add velocity to fractional position
      add   fracXMouse              ; add fractional position
      sta   fracXMouse              ; store new fractional position
      lda   r11H                    ; get high-byte of velocity
      adc   mouseXPos               ; add low-byte of position
      sta   mouseXPos               ; and store
      lda   r12L                    ; this is actually triple precision math
      adc   mouseXPos+1             ; add the high-byte of integer x-position
      sta   mouseXPos+1             ; r11 now has newly calculated x-position
      rts
```

C64Joystick:

```
.if 0
```

Function: Internal routine: Read the joystick and update the appropriate mouse related variables.

Called By: o_UpdateMouse.

Uses: none

Alters:

lastKeyRead	- set to new joystick read.
currentDisk	- set to new joystick direction (only if new).
pressFlag	- MOUSE_BIT set if fire button pressed. - INPUT_BIT set if joystick direction changed.
diskData	- new disk direction, if changed.
mouseData	- new state of fire button, if changed.

Destroys a, x, y.

```
.endif
C64Joystick:
        LoadB   cia1pra,#%11111111          ; scan no rows, so we're sure of stick
        lda     cia1prb                     ; get port data for joystick A (port 1)
        eor     #$FF                        ; complement data for positive logic
        cmp     lastKeyRead                 ; software debounce, must be same twice
        sta     lastKeyRead                 ; store value for debounce
        bne     20$                         ; if not same, don't pass return value

        and     #%1111                      ; isolate stick bits
        cmp     currentDisk                 ; compare to current stick value
        beq     10$                         ; if no change then branch
        sta     currentDisk                 ; set to new stick value
        tay                                 ; put value in y
        lda     directionTable,y            ; get the value to pass from table
        sta     diskData
        smbf    INPUT_BIT,pressFlag         ; mark that input device has changed
        jsr     ComputeMouseVels
10$
        lda     lastKeyRead                 ; get press
        and     #%10000                     ; isolate the fire button
        cmp     curMouse                    ; and compare it to the current value
        beq     20$                         ; if no change then branch
        sta     curMouse                    ; else, set new button value
        asl     a                           ; shift into bit 7
        asl     a
        asl     a
        eor     #%10000000                  ; complement to position logic
        sta     mouseData
        smbf    MOUSE_BIT,pressFlag         ; set changed bit
20$
        rts
```

```
directionTable:
       .byte [-1                       ; pass a -1 if no direction pressed
       .byte 2                         ; see hardware description at start
       .byte 6                         ; of this module to understand the
       .byte DISK_INVALID              ; direction conversions here
       .byte 4                         ; note that DISK_INVALID ($FF) are nonvalid states
       .byte 3                         ; actually they should be impossible
       .byte 5                         ; unless the controller is broken
       .byte DISK_INVALID
       .byte 0
       .byte 1
       .byte 7
       .byte DISK_INVALID
       .byte DISK_INVALID
       .byte DISK_INVALID
       .byte DISK_INVALID
       .byte DISK_INVALID

cosineTable:
       .byte 255                       ; dir 0         0    degree angle
       .byte 181                       ; dir 2        45    degree angle
                                       ; Note: the cosineTable overlaps the sineTable
sineTable:
       .byte 0                         ; dir 0         0    degree angle
       .byte 181                       ; dir 2        45    degree angle
       .byte 255                       ; dir 4        90    degree angle
       .byte 181                       ; dir 6       135    degree angle
       .byte 0                         ; dir 8       180    degree angle
       .byte 181                       ; dir 10     -135    degree angle
       .byte 255                       ; dir 12      -90    degree angle
       .byte 181                       ; dir 14      -45    degree angle

sineCosineTable:
       .byte POSITIVE | (POSITIVE >> 1); dir 0         0    degree angle
       .byte POSITIVE | (NEGATIVE >> 1); dir 2        45    degree angle
       .byte POSITIVE | (NEGATIVE >> 1); dir 4        90    degree angle
       .byte NEGATIVE | (NEGATIVE >> 1); dir 6       135    degree angle
       .byte NEGATIVE | (POSITIVE >> 1); dir 8       180    degree angle
       .byte NEGATIVE | (POSITIVE >> 1); dir 10     -135    degree angle
       .byte POSITIVE | (POSITIVE >> 1); dir 12      -90    degree angle
       .byte POSITIVE | (POSITIVE >> 1); dir 14      -45    degree angle
```

SineCosine:

```
.if 0
```

Function: Internal routine: SineCosine does a sixteen-direction sine and cosine and multiplies this value by a magnitude.

Called By: ComputMouseVels.

Parameters: x, diskData direction (0 to 15).
r0L magnitude of speed.

Returns: **r1L** x-velocity.
r2H y-velocity.

Destroys a, x, y, **r0**, **r6-r8**.

```
                                                                                                   .endif
SineCosine:
      lda    cosineTable,x             ; save cosine value
      sta    r1L
      lda    sineTable,x               ; save sine value
      sta    r2L
      lda    sineCosineTable,x         ; get signs
      pha
      ldx    #r1L                      ; compute x-velocity
      ldy    #r0L                      ; (Could do MultBB manually to avoid call to BBMult)
      jsr    BBMult
      ldx    #r2L                      ; compute y-velocity
      ;ldy   #r0L                      ; y already points to r0L
      jsr    BBMult
      pla
      pha
      bpl    10$                       ; if x-positive then branch
      NegateW r1
10$
      pla
      and    #%1000000
      beq    20$                       ; if y-positive then branch
      NegateW r2
20$
      rts
endJoystick:
```

app.con

```
                                                                                       .if 0
```

Function: Application constants

Filename: app.con

Uses: geo.con

Callable Routines:
none.

```
                                                                                      .endif
.include geo.con                       ; standard GEOS constants

;--- All constants only used by this application go here.

POSITIVE     = 0
NEGATIVE     = %10000000

;--- marks a joystick position that is impossible, short of a hardware fault
DISK_INVALID = $FF
```

app.sym

```
.if 0
```

Function: Application symbols

Filename app.sym

Uses:
geo.sym `standard GEOS symbols (jump table and variables)`
`geo.cia.sym  Includes full detailed symbols for the CIA chip`

Callable Routines:
none.

```
.endif
.include geo.sym
.include geo.cia.sym

;--- All zero page / .zsect declarations created for the application go here.


;--- All symbols created for the application go here.


;--- Global variables:

;--- We Normally don't want to send any constants to the linker.
;--- If we need a constant to go to linker for use in the .lnk file or other linker resolutions,
;--- then need to redefine it here.

mouse_Base   = MOUSE_BASE

diskData     = inputData              ; current disk direction
mouseSpeed   = inputData+ 1           ; current mouse speed
```

app.mac

```
.if 0
```

Function: Application Macros

Filename app.mac

Uses: geo.mac

Callable Routines:
None.

```
.endif
.include geo.mac                        ; standard GEOS macros

;--- All macros created for the application go here.

.macro NegateW zaddr
      ldx   #[zaddr
      jsr   Dnegate
.endm
```

app.Inc

`.if 0`

Function: Application include.

Filename app.Inc

Uses:

app.con	Application constants.
app.mac	Application macros.
app.sym	Application symbols.

Callable Routines:

None.

Note: This can only be used one time as an include per application. Use app.inc for secondary source files.

`.endif`

```
.if Pass1
      .noeqin                         ; never want to send CONSTANTS to linker
      .noglbl
            .include     app.con
            .include     app.mac
      .glbl
      .eqin
      .include     app.sym            ; all symbols will go to linker/debugger
.endif

.psect
```

128 COMM 1351(a)

`.if 0`

Function: Sample mouse driver.

Files:

app.lnk	Link file
app.hdr	Header file
app.driver.s	Driver source
app.Inc	Master Include. Sends all symbols to debugger
app.con	Application constants
app.sym	Application symbols
app.mac	Special macro(s) used for this driver.

Description: This driver source was generated from reverse engineering the 128 COMM 1351(a) driver. The source generates an exact copy.

Callable Routines:

o_InitMouse
o_SlowMouse
o_UpdateMouse
o_SetMouse

`.endif`

app.lnk

Function: Linker file for 1351 Mouse Driver

Filename: app.lnk

Uses:
app.hdr.rel
app.driver.rel

Callable Routines:
o_InitMouse
o_SlowMouse
o_UpdateMouse
o_SetMouse

```
.output      128Comm1351(a)
.header      app.hdr.rel
.seq

.psect       $FD00

app.driver.rel
```

app.hdr.s

```
                                                                                                    .if 0
```

Function: Define File Header Block

Filename: app.hdr.s Header File

Uses: app.con

Callable Routines:
None.

```
                                                                                                  .endif
.if    Pass1
       .noeqin
       .include     app.con
       .eqin
.endif

.header                                ; start of header section
JoyHdr:
       .word  0                        ; first two bytes are always zero
       .byte  3                        ; width in bytes
       .byte  21                       ; and height in scanlines of:
```

```
       .byte $80 | USR                 ; Commodore file type assigned to GEOS files
       ;.byte INPUT_DEVICE             ; GEOS file type
       .byte INPUT_128                 ; GEOS file type
       .byte SEQUENTIAL                ; SEQ file structure
       ;.word MOUSE_BASE               ; start address for saving file data
       .word MSE128_BASE               ; start address for saving file data
       .word endJoystick               ; end address for driver
       .word NULL                      ; not used (execution start address)
       ;--- 20 byte permanent name
       .byte "128 Comm 1351(a)",0,0,0,0

       ;--- 20 bytes for author name
       .byte  "Dave & Mike & PBM",NULL,0,0

.endh
```

app.driver.s

```
.include      app.Inc
```

```
                                                                        .if 0
```

Function: Main Source file for 1351(a) Mouse Driver.

Filename: app.driver.s

Uses: app.Inc

Callable Routines:

o_InitMouse
o_SlowMouse
o_UpdateMouse
o_SetMouse

Hardware: Mouse button is read from **cia1prb** (Joystick Port 1 DC01).
The bit values returned from this port are naturally set to 1. With the left mouse button pressed, b4 will be 0. The right mouse button uses b0, the same bit as the Joystick Up direction.

Only b0 and b4 is for the mouse with the following assignments.

b0	0	= right mouse button pressed
b4	0	= left button pressed

This port is shared with the keyboard. The keyboard has to be masked off prior to reading the mouse values. To do this write %11111111 to **cia1pra** (DC01) to select no keyboard rows to scan. Then any value read from **cia1prb** (DC01) will be from the mouse.

The mouse position is read from **potX** (D419) and **potY** (D41A). (**Note**: Only bits 1-6 are valid and bits 0 and 7 must be masked out). By comparing the values of these ports to the last saved values, a direction and distance can be computed. Acceleration is handled by expanding the distance moved on a sliding scale. Small movements are 1:1 distance. Large movements are up to 3x times the distance. The distance moved in the x and y are used to update the mouse position.

The calculated mouse direction is converted into Joystick directions and saved into **inputData**.

inputData: 0-7 for moving, -1 for centered
joystick directions:
0 = right
1 = up & right
2 = up
3 = up & left
4 = left
5 = left & down
6 = down
7 = down & right
-1 = joystick centered

```
                                                                      .endif
```

Jump Table

```
                                                                                            .if 0
Jump Table to Mouse Driver Routines
                                                                                           .endif
      ;--- input driver jump table

  jmp  o_InitMouse
  jmp  o_SlowMouse
  jmp  o_UpdateMouse
  jmp  o_SetMouse


      ;--- local variables:

lastButton:  .byte 0                    ; current value of mouse button
lastpotX:    .byte 0
lastpotY:    .byte 0
lastSpeed:   .byte 0

dblClkFlg:   .byte 0
```

o_InitMouse:

`.if 0`

Function:	External routine: This routine initializes the 'mouse'.
Called By:	At initialization EXTERNALLY.
Parameters:	nothing.
Alters:	**mouseXPos** starting position for the mouse. **mouseYPos**
Returns:	none.
Uses:	none.
Destroys:	a, x, y, **r0 - r15** Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

`.endif`

```
o_InitMouse:
     LoadW  mouseXPos,#8              ; set initial mouse position
     sta    mouseYPos
     ;---   fall through into o_SlowMouse
```

o_SlowMouse:

```
.if 0
```

Function:	External routine: Called when menus are pulled down to slow the mouse.	
Called By:	External and Internal.	
Parameters:	none.	
Returns:	nothing.	
Alters:	nothing.	
Destroys:	a, x, y, **r0 - r15**	Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
.endif
o_SlowMouse:
     rts
```

o_UpdateMouse:

```
.if 0
```

Function: External routine: This routine is called every interrupt to update the position of the pointer on the screen. First, the mouse is read and the mouse velocities are updated. The mouse position is then updated.

Called By: Interrupt code.

Parameters: nothing.

Uses: **mouseOn**.

Alters:

mouseXPos	current position for the mouse.
mouseYPos	
pressFlag	Set MOUSE_BIT to show input Device changed.

Returns: nothing.

Destroys: a, x, y, **r0 - r15** Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
                                                                                   .endif
o_UpdateMouse:
     bit    mouseOn
     bpl    o_SlowMouse               ; if mouse off then don't update
     PushB  cia1ddra                  ; save data direction reg a
     PushB  cia1ddrb                  ; save data direction reg b
     PushB  cia1pra                   ; save peripheral data reg a
     LoadB  cia1ddra,#0               ; set data direction registers to read
     sta    cia1ddrb
     lda    cia1prb                   ; peripheral data reg b
     and    #$10
     cmp    lastButton
     beq    10$
     sta    lastButton
     asl    a
     asl    a
     asl    a
     sta    mouseData
     smbf   MOUSE_BIT, pressFlag      ; set new mouse button data
10$
     ldx    lastSpeed
     dex
     bpl    20$
     ldx    #3
20$
     stx    lastSpeed
     sec
     lda    maxMouseSpeed
     sbc    speedTable,X
     bmi    90$
     ldx    #0
     stx    r1
     lda    potX
```

```
     ldy   lastpotX
     jsr   AccelDist
     sta   r2
     stx   r2H
     sty   lastpotX
     cmp   #0
     beq   40$
     and   #$80
     bne   30$
     lda   #$40
30$
     ora   r1
     sta   r1
40$
     bit   graphMode                 ; if 80-column mode then
     bpl   50$
     asl   r2                        ;    double move distance
     rol   r2H
50$
     jsr   GetDistance
     add   mouseXPos
     sta   mouseXPos
     txa
     adc   mouseXPos+1
     sta   mouseXPos+1
     lda   potY
     ldy   lastpotY
     jsr   AccelDist                 ; calculate Y distance moved
     sta   r2                        ; save distance
     stx   r2H
     sty   lastpotY                  ; save potY to last potY
     cmp   #$00
     beq   70$
     and   #$80
     lsr   a
     lsr   a
     lsr   a
     bne   60$
     lda   #$20
60$
     ora   r1
     sta   r1
70$
     jsr   GetDistance
     sec
     eor   #$FF                      ; reverse Y direction and add to mouse position
     adc   mouseYPos
     sta   mouseYPos
     txa
     eor   #$FF
     adc   #0
     cmp   #$FF
     bne   80$
     LoadB mouseYPos,#0
80$
     lda   r1
     lsr   a
     lsr   a
     lsr   a
```

```
      lsr   a
      tax
      lda   dirTable,X
      sta   inputData
  90$
      PopB  cia1pra
      PopB  cia1ddrb
      PopB  cia1ddra
      rts

;--- note that DISK_INVALID ($FF) are nonvalid states, actually they should be impossible
;--- unless the controller is broken

  dirTable:
      .byte [-1                    ; pass a -1 if no direction
      .byte 6                      ; see hardware description at start
      .byte 2                      ; of this module to understand the
      .byte DISK_INVALID           ; direction conversions here
      .byte 0
      .byte 7
      .byte 1
      .byte DISK_INVALID
      .byte 4
      .byte 5
      .byte 3
      .byte DISK_INVALID
      .byte DISK_INVALID
      .byte DISK_INVALID
      .byte DISK_INVALID
      .byte DISK_INVALID

  speedTable:
      .byte $3F,$1F,$00,$00
```

o_SetMouse:

```
                                                                                              .if 0
```

Function:	External routine: reset the pot (potentiometer) scanning lines so that they will recharge with the new value.
Called By:	Interrupt code.
Parameters:	nothing.
Uses:	**mouseOn**.
Alters:	**mouseXPos** - Current position for the mouse **mouseYPos**
Returns:	nothing.
Destroys:	a, x, y, **r0 - r15**. Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
                                                                                              .endif
 o_SetMouse:
      LoadB  cia1ddra,#%11111111
      LoadB  cia1pra,#%01000000
      rts
```

AccelDist:

```
.if 0
```

Function: Internal routine: Calculate the distance moved using last pot, current pot and acceleration table.

Called By: o_UpdateMouse.

Parameters:
y LASTPOT — lastpot.
a POT — pot.

Returns:
y pot to save.
r2 move distance.
a low-byte of move distance.
x high-byte of move distance.

Destroys: **r0**.

```
                                                                  .endif
AccelDist:
     sty    r0
     sta    r0H
     ldx    #0
     sub    r0                    ; calculate raw distance moved
     and    #%01111111            ; strip off sign bit
     cmp    #64
     bcs    10$                   ; if distance is < 64 then
     lsr    a                     ;    distance = distance >> 1
     beq    90$                   ;    if distance = 0 then no move. exit
     tay                          ;
     lda    accelTbl -1,Y         ;    get accel Distance
     ldy    r0H                   ;    put current pot in Y to be saved to lastpot
     rts                          ;    exit
10$                               ; end if
     ora    #%11000000            ;
     cmp    #$FF
     beq    90$                   ; if distance = -1 then exit.
     sec                          ;
     ror    a                     ;
     eor    #$FF                  ; distance = ~(distance / 2 | %10000000)
     tay                          ; distance now has a max value of 31. %xxx00000
     lda    accelTbl,Y            ;
     eor    #$FF
     add    #1
     ldx    #$FF
     ldy    r0H
     rts
90$
     lda    #0
     rts

     .byte  $01,$01,$01
accelTbl:
     .byte  $01,$01,$02,$02,$03,$04,$06,$08,$09,$0B,$0D,$0F,$11,$13,$15,$19
     .byte  $1D,$20,$23,$26,$29,$2C,$2F,$32,$35,$38,$3C,$41,$4B,$50,$5A,$64
```

GetDistance:

```
                                                                                 .if 0
```

Function: Internal routine: Return mouse movement distance saved in **r2**.

Called By: o_UpdateMouse.

Parameters: none.

Returns:

x	high-byte of movement distance.
a	low-byte of movement distance.

Alters: none.

Destroys: none.

```
                                                                                .endif

 GetDistance:
      ldx   r2H
      lda   r2L
      rts

endJoystick:                            ; ending Label used in .hdr file

;--- The following was in the disassembly of the mouse driver, commented out now
;--- since it was never used.

;T_posAdj:
;      .byte  $01,$01,$01,$00           ; unused position adjustment table
```

app.con

```
                                                                                                   .if 0
```

Function: Application constants

Filename: app.con

Uses: geo.con

Callable Routines:
None.

```
                                                                                                  .endif
.include geo.con                        ; standard GEOS constants
.include ge8.con                        ; 128 GEOS constants

;--- All constants only used by this application go here.

POSITIVE     = 0
NEGATIVE     = %10000000
cia1pra      = cia1base
cia1prb      = $DC01

;--- Marks a joystick position that is impossible, short of a hardware fault.
DISK_INVALID = $FF
```

app.sym

```
.if 0
```

Function: Application symbols

Filename app.sym

Uses:

geo.sym	standard GEOS symbols (jump table and variables)
geo.cia.sym	Includes full detailed symbols for the CIA chip
ge8.sym	128 GEOS symbols

Callable Routines:
none.

```
.endif
.include geo.sym
.include geo.cia.sym
.include ge8.sym

;--- All zero page declarations created for the application go here.

;--- All symbols created for the application go here.


;--- Global variables:

;--- We normally don't send any constants to the linker.
;--- If we need one to go to linker for use in the .lnk file or other linker resolutions
;--- then need to redefine here.
mouse_Base   = MSE128_BASE

diskData     = inputData              ; current disk direction
mouseSpeed   = inputData+ 1           ; current mouse speed

;potX        == $D419                 ; bits 1-6 = current x-position
;potY        == $D41A                 ; bits 1-6 = current y-position
```

app.mac

```
                                                                                   .if 0
```

Function: Application macros

Filename app.mac

Uses: geo.mac `standard GEOS macros`

Callable Routines:
none.

```
                                                                                  .endif
.include geo.mac

;--- All macros created for the application go here.

.macro NegateW zaddr
      ldx   #[zaddr
      jsr   Dnegate
.endm
```

app.Inc

```
.if 0
```

Function: Application include

Filename app.Inc

Uses:		
	app.con	Application constants
	app.mac	Application macros
	app.sym	Application symbols

Callable Routines:
none.

Note: This can only be used one time as an include per application. Use app.inc for secondary source files.

```
                                                                                                   .endif
.if Pass1
      .noeqin                         ; never want to send CONSTANTS to linker
      .noglbl
            .include     app.con
            .include     app.mac
      .glbl
      .eqin
      .include     app.sym            ; all symbols will go to linker/debugger
.endif

.psect
```

64_128 COMM 1351(a)

`.if 0`

Function: Sample Mouse Driver.

Files:

ReadMe	Build instructions
1351.64.lnk	Link file
1351.128.lnk	Link file
1351.hdr.s	Header file
1351.driver.s	Driver source
1351.Inc	Master Include. Sends all symbols to debugger
1351.cfg	Configuration file.
1351.con	Application constants
1351.sym	Application symbols
1351.mac	Special macro(s) used for this driver

Description: This driver source was created from the 128 COMM 1351(a) as a base. It was then optimized and updated. This version has the following new features:

1. Generates either a 128 or 64 driver.
2. Right mouse button generates a double click action.
3. Variable acceleration at build time. This could evolve into a driver that has its sensitivity change on the fly with a supporting application.

Callable Routines:

InitMouse
SlowMouse
UpdateMouse
SetMouse (128 Version)

`.endif`

ReadMe

Example 1351 Mouse Driver

This driver is configurable to generate a mouse driver for 64 GEOS 1.2+ or 128 GEOS 1.4+.

The driver configuration options are located in 1351.cfg.
Options:

C128	TRUE	= driver is generated for a 128 GEOS.
	FALSE	= 64 GEOS

DRAG Valid values 0 to 2.
0 = Default acceleration matching that of the original 1351(a) driver.
1..2 = Increasingly less acceleration.
Note: Values outside of the listed range will cause the mouse to be useless.
The DRAG value is displayed in the info section of the driver so you will know what value was used when the driver was built.
Recommended value is 1.

1351.cfg

```
;--- Configuration Options

C128   = TRUE                              ; FALSE = Build 64 Driver
                                           ; TRUE  = Build 128

.if C128
       .include ge8.con
       INPTYPE      = INPUT_128
       MAXDRVSIZE   = (END_MSE128-MSE128_BASE)
.else
       INPTYPE      = INPUT_DEVICE
       MAXDRVSIZE   = (END_MOUSE-MOUSE_BASE) + 60
.endif

DRAG   = 0                                 ; 0-2  0=Normal. 1-2 increasingly less acceleration
```

1351.64.lnk

Function: Linker file for C64 1351 Mouse Driver.

Filename: 1351.lnk

Uses:
1351.hdr.rel
1351.driver.rel

Callable Routines:
InitMouse
SlowMouse
UpdateMouse

```
.output      64_1351(a)
.header      1351.hdr.rel
.seq

.psect       BASEMOUSE           ;$FE80
1351.driver.rel
```

1351.128.lnk

Function: Linker file for C128 1351 Mouse Driver.

Filename: 1351.lnk

Uses: 1351.hdr.rel
1351.driver.rel

Callable Routines:
InitMouse
SlowMouse
UpdateMouse
SetMouse

```
.output      128Comm1351a
.header      1351.hdr.rel
.seq

.psect       BASEMOUSE           ;$FD00
1351.driver.rel
```

1351.hdr

```
.if 0
```

Function: Define File Header Block

Filename: 1351.hdr Header File

Uses: 1351.con

Callable Routines:
none.

```
.endif

.if    Pass1
       .noeqin
       .include     1351.con
       .eqin
.endif

.header                                  ; start of header section
       .word  0                          ; first two bytes are always zero
       .byte  3                          ; width in bytes
       .byte  21                         ; and height in scanlines of:
```

```
       .byte $80 | USR                   ; Commodore file type assigned to GEOS files
       .byte INPTYPE                     ; GEOS file type
       .byte SEQUENTIAL                  ; SEQ file structure
       .word BaseMouse                   ; start address for saving file data
       .word endDriver                   ; end address for saving file data (-1)
       .word BaseMouse                   ; not used (execution start address)

       ;--- 20 byte permanent name
.if C128
       .byte "128 Comm 1351(a)",NULL,0,0,0
.else
       .byte "Comm 1351(a)",NULL,0,0,0,0,0,0,0
.endif
       ;--- 20 bytes for author name
       .byte  "Dave & Mike & PBM",NULL,0,0

       .block 43
       ;--- info block
       .byte  "Right Button Double Click",CR
       .byte  "Drag = ",DRAG+'0',NULL

.endh
```

1351.driver.s

```
.include     1351.Inc
                                                                                   .if 0
 Page  Block          Description                                            Notes
    2                 Overview Comments.
    3  BaseMouse      jmp table.
    4  _SetMouse      128 Refresh Pot. (In line with jmp table).
    5                 Local variables.
                      dirTable. Used to translate mouse movements into
                      direction.
    6  _InitMouse     Initialize the mouse.
    7  _UpdateMouse   Header Info for o_UpdateMouse.
    8  UpdBtns        Update status of Left and Right buttons.
    9  UpdateX        Calculate x axis move distance and update mouseXPos.
   10  UpdateY        Calculate y axis move distance and update mouseYPos.
   11  AccelDist      Compute Distance Moved on X and Y planes.
   12  endDriver      Assembler size Check for driver size exceeded.
                                                                                  .endif
```

Note: This page includes an index into the rest of the app.driver.s file, with page numbers, block names, and descriptions. These page numbers represent the actual page numbers in geoWrite. To try to make those page numbers make sense, the geoWrite page numbers are in the headings of each of the app.driver.s pages. The point of this is to teach an organization tool that can be used with source code in geoWrite that makes it very fast to find what you are looking for.

Overview

`.if 0`

Function: Main Source file for 1351(a) Mouse Driver.

Filename 1351.driver.s

Uses: 1351.Inc

Callable Routines:

- _InitMouse
- _SlowMouse
- _UpdateMouse
- _SetMouse (128 Only)

Hardware: Mouse button is read from **cia1prb** (Joystick Port 1 DC01). The bit values returned from this port are naturally set to 1. With the left mouse button pressed, b4 will be 0. The right mouse button uses b0, the same bit as the Joystick Up direction.

Only b0 and b4 is for the mouse with the following assignments.

b0	0	= right mouse button pressed
b4	0	= left button pressed

This port is shared with the keyboard. The keyboard has to be masked off prior to reading the mouse values. To do this write %11111111 to **cia1pra** (DC01) to select no keyboard rows to scan. Then any value read from **cia1prb** (DC01) will be from the mouse.

The mouse position is read from **potX** (D419) and **potY** (D41A). (**Note**: Only bits 1-6 are valid and bits 0 and 7 must be masked out). By comparing the values of these ports to the last saved values, a direction and distance can be computed. Acceleration is handled by expanding the distance moved on a sliding scale. Small movements are 1:1 distance. Large movements are up to 3x times the distance. The distance moved in the x and y are used to update the mouse position.

The calculated mouse direction is converted into Joystick directions and saved into **inputData**.

inputData: 0-7 for moving, -1 for centered.
joystick directions:
0 = right
1 = up & right
2 = up
3 = up & left
4 = left
5 = left & down
6 = down
7 = down & right
-1 = joystick centered

`.endif`

Jump Table

```
                                                                                                  .if 0
Jump Table to Mouse Driver Routines
                                                                                                 .endif
BaseMouse:
      ;--- Input driver jump table
;InitMouse
      jmp   _InitMouse
      ;---
;SlowMouse
      rts                             ; _SlowMouse has nothing to do
      nop                             ; rts instead of jmp to rts
      nop                             ; saving 3 cycles
      ;---
;UpdateMouse
      jmp   _UpdateMouse
      ;---
      ;---   Normally _setMouse jmp entry but we are inlining it.

;--- Fall into o_SetMouse instead of jmp. Saves 3 bytes.
;.if C128
;     jmp   _SetMouse
;.endif
;--- This makes o_SetMouse replace the 4th jump table entry.
```

_SetMouse:

```
.if 0
```

Function:	External routine:	Reset the pot (potentiometer) scanning lines so that they will recharge with the new value.
Called By:	Interrupt code.	
Parameters:	nothing.	
Uses:	**mouseOn**	
Alters:	**mouseXPos** **mouseYPos**	current position for the mouse
Returns:	nothing.	
Destroys:	a, x, y, **r0 - r15**	Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
.endif
.if C128
      _SetMouse:
             LoadB  cia1ddra,#%11111111
             LoadB  cia1pra,#%01000000
             rts
.endif
```

Locals

```
      lastButton:         .byte 0      ; current value of mouse button
      lastpotX:           .byte 0
      lastpotY:           .byte 0
      dblClkFlg:          .byte 0

      dirTable:
            .byte  [-1                 ; pass a -1 if no direction
            .byte  6                   ; down       0001
            .byte  2                   ; up         0010
            ;---   can't do up and down at same time
            .byte  DISK_INVALID        ; n/a        0011
            .byte  0                   ; right      0100
            .byte  7                   ; right/down 0101
            .byte  1                   ; right/up   0110
            ;---   can't do up and down at same time
            .byte  DISK_INVALID        ; n/a        0111
            .byte  4                   ; left       1000
            .byte  5                   ; left/down  1001
            .byte  3                   ; left/up    1010
            ;---   All remaining possibilities are invalid and will never happen.

 accelTbl:
.if DRAG=1
      .byte  $01,$01,$01,$02,$02,$03,$04,$06,$08,$09,$0B,$0D,$0F,$11,$13,$15
      .byte  $19,$1D,$20,$23,$26,$29,$2C,$2F,$32,$35,$38,$3C,$41,$4B,$50,$5A

.elif DRAG=2
      .byte  $01,$01,$01,$01,$02,$02,$03,$04,$06,$08,$09,$0B,$0D,$0F,$11,$13
      .byte  $15,$19,$1D,$20,$23,$26,$29,$2C,$2F,$32,$35,$38,$3C,$41,$4B,$50

.else
      .byte  $01,$01,$02,$02,$03,$04,$06,$08,$09,$0B,$0D,$0F,$11,$13,$15,$19
      .byte  $1D,$20,$23,$26,$29,$2C,$2F,$32,$35,$38,$3C,$41,$4B,$50,$5A,$64

.endif
```

```
.if 0
```

Function:	External routine: This routine initializes the 'mouse'.
Called By:	At initialization EXTERNALLY.
Parameters:	nothing.
Alters:	**mouseXPos** starting position for the mouse. **mouseYPos**
Returns:	none.
Uses:	none.
Destroys:	a, x, y, **r0 - r15** Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
.endif
_InitMouse:
      LoadW  mouseXPos,#8              ; set initial mouse position
      sta    mouseYPos
br_rts                                                                       =*
      rts                              ; rts also used by o_UpdateMouse
```

_UpdateMouse:

```
.if 0
```

Function:	External routine:	This routine is called every interrupt to update the position of the pointer on the screen. First, the mouse is read and the mouse velocities are updated. The mouse position is then updated.
Called By:	Interrupt code.	
Parameters:	nothing.	
Uses:	**mouseOn**.	
Alters:	**mouseXPos** **mouseYPos**	current position for the mouse.
	pressFlag	Set MOUSE_BIT to show input device changed.
Returns:	nothing.	
Destroys:	a, x, y, **r0 - r15**	Even though this does not destroy everything here, another driver may destroy anything or nothing. So, we have to declare everything destroyed.

```
                                                                                         .endif
_UpdateMouse:
      bbrf  7,mouseOn,o_SlowMouse     ; if mouse off then don't update
.if !C128
      PushB CPU_DATA
      LoadB CPU_DATA,#IO_IN
.endif
      PushB cia1ddra                  ; save Data Direction Reg a
      PushB cia1ddrb                  ; save Data Direction Reg b
      PushB cia1pra                   ; save Peripheral Data Reg a
      ldy   #0
      sty   cia1ddra                  ; set Data Direction Registers to read
      sty   cia1ddrb
      ;---  fall through to UpdBtns
```

```
  UpdBtns:
       bbeq   dblClkFlg,10$             ; check internal dblClkFlg
                                        ; if zero then get new buttons

       ;---   when dblClickCount is 29 show button released
       CmpB   dblClickCount,#29
       beq    30$                       ;

       ;---   when dblClickCount >= 27 do nothing and go to movement
       cmp    #27
       bcs    ResetPot

       sty    dblClkFlg                 ; reset internal dblClkFlg
       bcc    20$                       ; always branch to set button pressed
                                        ; to complete automated double-click

  10$  lda    cia1prb                   ; get Left and right button bits
       and    #%00010001                ; strip out bits we don't care about.
       cmp    lastButton                ; if the button status has not changed then
       beq    ResetPot                  ;    exit this section
       sta    lastButton                ; save new button status

       ;---   Reset dblClickCount to 0. On button press, icon handler sets dblClickCount to 30
       ;       to give the user roughly .5 seconds to perform a second click
       sty    dblClickCount
       lsr    a                         ; rotate right button bit into the carry flag
       beq    20$                       ; if z=0 then left button was pressed
       bcs    30$                       ; if carry is set then right button was not pressed
                                        ; and left was released
       inc    dblClkFlg                 ; right button was pressed, bump internal dblClkFlg

  20$  clc                              ; button pressed / clear carry flag for bit 7
       .byte  $B0                       ; (bcs instruction. skip next byte)

  30$  sec                              ; button released / set carry flag for bit 7
       ror    a                         ; put carry into bit 7

  40$  sta    mouseData                 ; set left button status. (b7 = pressed / released)
       smbf   MOUSE_BIT,pressFlag       ; set flag to show the mouse button state has changed

  ResetPot:
.if !C128
       LoadB  cia1ddra,#%11111111       ;reset the pot (potentiometer) scanning lines
       LoadB  cia1pra,#%01000000        ;so that they will recharge with the new value.
       ldx    #102
  10$  nop                              ;Wait for new value to be read
       nop
       nop
       dex
       bne    10$
.endif

;---   fall through to UpdateX
```

```
      UpdateX:
      ;--- Calculate x axis movement and update mouseXPos
             lda    potX                ; get current raw hardware x-position
             ldy    lastpotX            ; get last hardware x-position
             jsr    AccelDist           ; calculate distance and direction
             sty    lastpotX            ; save new last position
             tay                        ; check if movement occurred
             beq    20$                 ; if z=0 then no movement on x axis
             bmi    10$                 ; if negative then moving left
             lda    #%0100              ; moving right
clda  10$,   #%1000 ; moving Left
      20$    sta    r1                  ; save left/right/none direction moved in r1
             bbrf   7,graphMode,30$     ; if 80-column mode then
             asl    r2                  ;    double move distance.
      ;--- r2H is always 0 or FF, and it follows bit 7 of r2, no need to rol it
      30$    AddW   r2, mouseXPos       ; add movement to current x-position

;---  fall through to UpdateY
```

UpdateY

```
      UpdateY:
            ;--- Calculate Y axis movement and update mouseYPos
            lda    potY                 ; get current raw hardware y-position
            ldy    lastpotY             ; get last hardware y-position
            jsr    AccelDist            ; calculate Y distance moved
            sty    lastpotY             ; save potY to last Pot Y
            tay
            beq    80$                  ; no Y motion...
            bmi    10$
            lda    #%10                 ; moving up
clda  10$,         #%01                 ; moving down
            ora    r1                   ; set direction bit flag
            sta    r1                   ; save up/down/none direction moved in r1
            sec
            lda    mouseYPos            ; subtract movement from current mouse y-position
            sbc    r2
            bbsf   7,r2H,20$            ; check if Moving up or down
            bcc    30$                  ; moving up and bcc = above screen top
            clc
      20$   bcs    40$                  ; rolled over byte. >256 result
            cmp    #SC_PIX_HEIGHT       ; rolled over screen bottom
            bcs    40$
clda  30$,         #0                   ; moving up and went under 0. reset to 0
clda  40$,         #SC_PIX_HEIGHT-1     ; moving down and went over 255. reset to 199
            sta    mouseYPos            ; save new y-position

      80$   ldx    r1                   ; translate direction moved into joystick directions
            lda    dirTable,X
            sta    inputData            ; save into inputData
      UMExit:
            PopB   cia1pra              ; restore registers to previous state
            PopB   cia1ddrb
            PopB   cia1ddra
.if !C128
            PopB   CPU_DATA
.endif
            rts                         ; exit
```

AccelDist:

```
.if 0
```

Function: Internal routine:
Calculate the distance moved using last pot, current pot and acceleration table.

Called By: o_UpdateMouse.

Parameters:
y LASTPOT — lastpot
a POT — pot

Returns:
y pot to save.
r2 move distance.
a low-byte of move distance.
x high-byte of move distance.

Destroys: **r0**.

```
                                                                              .endif
      AccelDist:
            sty   r0
            sta   r0H
            ldx   #0
            sub   r0                  ; calculate raw distance moved
            and   #%01111111          ; strip off sign bit
            cmp   #64                 ; if distance is < 64
            bcs   10$
            lsr   a                   ;    distance = distance >> 1
            beq   80$                 ;    if distance = 0 then no move exit
            tay                       ;
            lda   accelTbl-1,Y        ;    get accel distance
            ldy   r0H                 ;    put current pot in Y to be saved to lastpot
            bra   90$                 ; exit

      10$   ora   #%11000000
            cmp   #$FF
            beq   80$                 ; if distance = -1 then exit
            sec
            ror   a
            eor   #$FF                ; distance = ~(distance / 2 | %10000000)
            tay                       ; distance now has a max value of 31 (%xxx00000)
            lda   accelTbl,Y          ; get accelerated distance (AD)
            eor   #$FF
            add   #1                  ; a now has the 2's complement of AD
            ldx   #[-1
            ldy   r0H
clda  80$,        #0                  ; if branched into 80$ then
                                      ;    set move distance to no movement
      90$   sta   r2                  ; save distance
            stx   r2H                 ; save direction
            rts
```

endDriver

```
endDriver:

.if * > MAXDRVSIZE
       .echo Error: Driver has exceeded maximum size.
.endif

.end
```

1351.con

```
                                                                                         .if 0
```

Function: Application constants

Filename: 1351.con

Uses: geo.con
1351.cfg

Callable Routines:
none.

```
                                                                                        .endif
.include geo.con                       ; standard GEOS constants
.include 1351.cfg                      ; multi output build needs cfg file to control output

;--- All constants only used by this application go here.

POSITIVE     = 0
NEGATIVE     = %10000000

;--- Marks a joystick position that is impossible, short of a hardware fault
DISK_INVALID = $FF
```

1351.sym

```
.if 0
```

Function: Application symbols

Filename: 1351.sym

Uses:

geo.sym	standard GEOS symbols (jump table and variables)
geo.cia.sym	Includes full detailed symbols for the CIA chip
ge8.sym	128 GEOS symbols

Callable Routines:

none.

```
                                                                                             .endif
.include geo.sym
.include geo.cia.sym
.include ge8.sym

;--- All zero page declarations created for the application go here.

;--- Driver does not get to use application zero page space.
;.zsect APP_ZPL                         ; 16 bytes dedicated
;.zsect APP_ZIO                         ; 123 bytes of swappable I/O
;.zsect AAP_ZPH                         ; 4 bytes dedicated

;--- All symbols created for the application go here.

;
;      Global variables:

;--- We normally don't send any constants to the linker.
;--- If we need one to go to linker for use in the .lnk file or other linker resolutions
;--- then need to redefine here.

.if C128
             BASEMOUSE=MSE128_BASE
.else
             BASEMOUSE=MOUSE_BASE
.endif

diskData     = inputData                ; current disk direction
mouseSpeed   = inputData+ 1             ; current mouse speed

;potX        == $D419                   ; bits 1-6 = current x-position
;potY        == $D41A                   ; bits 1-6 = current y-position
```

1351.mac

```
                                                                                                  .if 0
```

Function: Application macros.

Filename 1351.mac

Uses: geo.mac

Callable Routines:
none.

```
                                                                                                  .endif
.include geo.mac                            ; standard GEOS macros

;--- All macros created for the application go here.

.macro NegateW zaddr
      ldx    #[zaddr
      jsr    Dnegate
.endm
```

1351.Inc

```
.if 0
```

Function:	Application include	
Filename	1351.Inc	
Uses:	1351.con	Application constants
	1351.mac	Application macros
	1351.sym	Application symbols

Callable Routines:

none.

Note: This can only be used one time as an include per application. Use 1351.inc for secondary source files.

```
                                                                                   .endif
.if Pass1
      .noeqin                          ; never want to send CONSTANTS to linker
      .noglbl
            .include     1351.con
            .include     1351.mac
      .glbl
      .eqin
      .include     1351.sym            ; all symbols will go to linker/debugger
.endif

.noeqin
.psect
```

8-Bit FX-80 Printer Driver

`.if 0`

Function: Sample 8-bit printer driver for FX-80 series.

Files:

app.lnk	Link file
app.hdr	Header file
app.driver.s	Driver source
app.Inc	Master Include. Sends all to symbols debugger
app.con	Application constants
app.sym	Application symbols
app.mac	Application macros

Callable Routines:

InitForPrint	-> _InitForPrint
StartPrint	-> _StartPrint
PrintBuffer	-> _PrintBuffer
StopPrint	-> _StopPrint
GetDimensions	-> _GetDimensions
PrintASCII	-> _PrintASCII
StartASCII	-> _StartASCII
SetNLQ	-> _SetNLQ

For: EPSON FX-80,FX-100,RX-80,RX-100,JX-80
PANASONIC KX-1091,KX-1092,KX-1592,KX-1595

Tested on: EPSON JX-80

`.endif`

app.lnk

;**Function**:	Linker file for FX-80 Driver.
;
;**Filename**:	app.lnk
;
;**Uses**:	app.hdr.rel
;	app.driver.rel

```
.output      FX-80
.header      app.hdr.rel
.seq

.psect       $7900        ; printBase
app.driver.rel
prndrv.lib.rel
```

app.hdr

```
                                                                                                  .if 0
```

Function: Define File Header Block.

Filename: app.hdr.s Linker header file

Uses: app.con

Callable Routines:
none.

```
                                                                                                 .endif
.if    Pass1
       .noeqin
       .include     app.con
       .eqin
.endif

.header                                  ; start of header section
       .word  0                          ; first two bytes are always zero
       .byte  3                          ; width in bytes
       .byte  21                         ; and height in scanlines of:
```

```
       .byte  $80 | USR                  ; Commodore file type assigned to GEOS files
       .byte  PRINTER                    ; GEOS file type
       .byte  SEQUENTIAL                 ; SEQ file structure
       .word  PRINTBASE                  ; start address for saving file data
       .word  PRINTEND                   ; end address of print driver
       .word  NULL                       ; not used (execution start address)
       ;---   20 byte permanent name
       .byte  "FX80-Drvr   V1.1",0,0,0,0

       ;---   20 bytes for author name
       .byte  "OGPRG & PBM",NULL,0,0,0,0,0,0,0,0
.endh
```

app.driver.s

```
                                                                                      .if 0
```

Function: Main Source file for FX-80 Print Driver.

Filename: app.driver.s

Uses: app.Inc

Callable Routines:

InitForPrint	->	_InitForPrint
StartPrint	->	_StartPrint
PrintBuffer	->	_PrintBuffer
StopPrint	->	_StopPrint
GetDimensions	->	_GetDimensions
PrintASCII	->	_PrintASCII
StartASCII	->	_StartASCII
SetNLQ	->	_SetNLQ

```
                                                                                      .endif
.include     app.Inc
```

Jump Table

```
                                                                                   .if 0
```

Jump Table to Print Driver Routines

```
                                                                                   .endif
;--- Input   driver jump table
;InitForPrint:
             rts
             nop
             nop
;StartPrint:
             jmp    _StartPrint
;PrintBuffer:
             jmp    _PrintBuffer
;StopPrint:
             jmp    _StopPrint
;GetDimensions:
             jmp    _GetDimensions
;PrintASCII:
             jmp    _PrintASCII
;StartASCII:
             jmp    _StartASCII
;SetNLQ:
             jmp    _SetNLQ

;--- RAM STORAGE/ UTILITIES

;      Local variables:

printerName:
      ;--- name of printer as it should appear in menu
      .byte "Epson FX-80",NULL
prntblcard:
      .block 8                         ; printable character block
breakcount:
      .byte  0
;reduction:
;     .byte  0
;cardwidth:
;     .byte  0                         ; width of the print buffer line in cards
                                       ; Used for reduction flag in laser drivers
scount:
      .byte  0                         ; string output routine counter
cardcount:
      .byte  0                         ;
modeflag:
      .byte  0                         ; either 0=graphics, or $FF=ASCII
                                       ; for draft or nlq mode
                                       ; utility routines: (see "Print Driver Support Library")
```

_StartPrint:

`.if 0`

Function: Performs initialization necessary before printing each page of a document.

Called By: A GEOS application.

Parameters: nothing.

Returns: nothing.

Destroys a, x, y, **r3**.

Description: This is the **StartPrint** routine as discussed above. It initializes the serial bus to the printer, and sets up the printer to receive graphic data.

`.endif`

```
_StartPrint:
      lda   #0                    ; set for graphic mode
      sta   modeflag
StartIn:
      lda   #PRINTADDR            ; set to channel 4
      jsr   SetDevice
      jsr   InitForIO             ; set I/O space in, disable interrupts
      lda   #0
      sta   status                ; initialize the error byte to no error
      jsr   OpenFile              ; open the file for the printer
      lda   status                ; if problems with the output channel, go to
      bne   20$                   ; error handling routine
      jsr   OpenPrint             ; open channel to printer
      jsr   InitPrinter           ; initialize the printer for graphic mode
      jsr   ClosePrint            ; close the print channel
      jsr   Delay                 ; wait for weird timing problem
      jsr   DoneWithIO            ; set mem map back, and enable ints
      ldx   #0
      rts
20$                               ; save error return from the routines
      pha                         ; bit 0 set: timeout, write
                                  ; bit 7 set: device not present
      jsr   CloseFile             ; close the file anyway
      jsr   DoneWithIO            ; set mem map back, and enable interrupts
      PopX                        ; recover the error return
      rts                         ; pass out in x

Delay:
      ldx   #0
10$
      ldy   #0
20$
      dey
      bne   $20
      dex
      bne   $10
      rts
```

_PrintBuffer:

```
                                                                   .if 0
```

Function: Prints out the indicated 640-byte buffer of graphics data (80 cards) as created by an application.

Called By: A GEOS application.

Parameters:

r0	BUFFER	— address of the 640-bytes (80 cards) to be printed.
r1	PBBUFFER	— address of an additional 640-byte buffer for **PrintBuffer** to use.

Destroys: a, x, y, **r3**.

Description: **PrintBuffer**, as described in more detail above, is the top level routine that dumps data from the GEOS 640-byte buffer maintained in the Commodore C64 to the printer using the serial port.

Note: The *PBBUFFER* may not change between calls to **PrintBuffer**. 7-bit printers use it to store the left-over scanlines between calls. Each time **PrintBuffer** is called it is passed 8 scanlines of data but only 7 may be printed.

```
                                                                   .endif
_PrintBuffer:
      lda    #PRINTADDR                ; set to channel 4
      jsr    SetDevice                 ; set to printer device
      jsr    InitForIO                 ; put I/O space in and disable interrupts
      jsr    OpenPrint                 ; open channel to printer
      MoveW  r0,r3
      jsr    PrnPrintBuffer            ; print the users 8-bit high buffer
      jsr    Greturn                   ; do CR-LF here
      jsr    ClosePrint                ; close the print channel
      jsr    DoneWithIO                ; put RAM back in, enable interrupts
      rts
```

_StopPrint:

```
.if 0
```

Function: Called at end of every page to flush output buffer and tell the printer to form feed.

Parameters: **r0** BUFFER — address of the 640-bytes (80 cards) to be printed.
r1 PBBUFFER — address of an additional 640-byte buffer for **PrintBuffer** to use.

Destroys a, x, y, **r3**.

Description: **StopPrint** is called after all cards for a given page have been sent to the printer. It does a **SetDevice**, **InitForIO**, makes the printer listen, and if the printhead was printing 7-bit high data, flushes out any remaining lines of data in the print buffer. It then does a form-feed and an unlisten, closes the Commodore output file, and does a **DoneWithIO**.

Note: The *PBBUFFER* may not change between calls to **PrintBuffer**. 7-bit printers use it to store the left-over scanlines between calls. Each time **PrintBuffer** is called it is passed 8 scanlines of data but only 7 may be printed.

```
.endif
_StopPrint:
      lda   #PRINTADDR              ; set to channel 4
      jsr   SetDevice               ; set to printer device
      jsr   InitForIO               ; put I/O space in and disable interrupts
      jsr   OpenPrint               ; open channel to printer
      jsr   FormFeed                ; do a form feed
      jsr   ClosePrint              ; close the print channel
      jsr   CloseFile               ; close the print file
      jsr   DoneWithIO              ; put RAM back in, enable interrupts
      rts
```

_GetDimensions:

```
                                                                                .if 0
```

Function:	Return the dimensions in cards of the rectangle that will print in an 8 x 10.5 area of the screen.
Parameters:	nothing.
Returns:	a $00 (printer has graphics and text modes). x width, in cards, that this printer can put out across a page. y height, in cards, that this printer can put down a page.
Uses:	nothing.
Destroys:	nothing.
Description:	**GetDimensions** returns the number of cards wide and high that this printer is capable of printing out on an 8 x 10.5 subset of an 8.5 x 11 inch page.

```
                                                                               .endif
_GetDimensions:
     ldx    #CARDSWIDE                   ; get the number of cards wide
     ldy    #CARDSDEEP                   ; and get the number of cards high
     lda    #0                           ; set for graphics and text driver
     rts
```

_StartASCII:

```
.if 0
```

Function: Initializes the Epson to receive ASCII print streams.

Called By: A GEOS application.

Parameters: nothing.

Returns: nothing.

Destroys: a.

Description: Just sets the mode flag, called by user at beginning of each document.

```
.endif
_StartASCII:
      LoadB  modeflag,#ASCII           ; set mode to ASCII printing
      jmp    StartIn                   ; pick up rest of start print
```

_SetNLQ:

```
.if 0
```

Function: Initializes the Epson to near letter quality mode.

Called By: A GEOS application.

Parameters: nothing.

Returns: nothing.

Destroys: a.

Description: Send the printer driver the correct initialization bytes to put it in NLQ mode.

```
.endif
_SetNLQ:
      lda    #PRINTADDR                ; set to channel 4
      jsr    SetDevice                 ; set device number
      jsr    InitForIO                 ; put I/O space in, disable interrupts
      jsr    OpenPrint                 ; open channel to printer
      LoadW  r3,#nlqTbl                ; table of initialization bytes to send
      lda    #NLQTBLSZ                 ; the length of the table
      jsr    Strout                    ; send the table to the printer
      jsr    ClosePrint                ; close the print channel
      jsr    DoneWithIO                ; put RAM back in, enable interrupts
      rts

nlqTbl:
      ;---   NLQ bytes, transmitted from right to left
      ;---   command the printer to enable NLQ mode
      .byte  $47, ESC, $45, ESC
NLQTBLSZ = * - nlqTbl                  ; end of command string to activate the NLQ mode
```

_PrintASCII:

```
.if 0
```

Function: Prints a null terminated ASCII string to the printer.

Called By: A GEOS application.

Parameters: none.

Uses:

r0	pointer to the ASCII string.
r1	pointer to the 640-bytes buffer for the printer driver to use.

Returns: nothing.

Destroys a, x, y.

Description: Sends a null terminated ASCII string to the printer. All carriage returns and linefeeds must be handled by the application. Carriage returns are mapped into CR-LF.

```
.endif
_PrintASCII:
      lda    #PRINTADDR                ; set to channel 4
      jsr    SetDevice                 ; set to printer device
      jsr    InitForIO                 ; put I/O space in and disable interrupts
      jsr    OpenPrint                 ; open print channel
10$
      ldy    #0                        ; init the index into ASCII string
      lda    (r0),y                    ; get the character
      beq    30$                       ; if at end of string, exit
      cmp    #CR                       ; if carriage return, add LF
      bne    20$                       ; branch if not CR
      jsr    Ciout                     ; output the character
      lda    #LF                       ; load up a linefeed
20$
      jsr    Ciout
      IncW   r0                        ; point to next character
      bra    10$                       ; do again
30$
      jsr    ClosePrint                ; close the print channel
      jsr    DoneWithIO                ; put RAM back in, enable interrupts
      rts
```

PrnPrintBuffer:

`.if 0`

Function: Prints out the print buffer pointed to by **r3**.

Called By: **PrintBuffer**.

Parameters: **r3** address of start of buffer to print.

Returns: **r3** unchanged.

Destroys: a, x, y, **r0-r15**.

Description: Checks to see if the buffer is empty before printing the data. Then for each card in the buffer, rotate the data and send it to the printer.

`.endif`

```
PrnPrintBuffer:
      PushW  r3                        ; save the buffer pointer
      jsr    TestBuffer                ; see if the buffer is all zeros
      bcs    10$                       ; if there is data in the buffer, send it
      PopW   r3                        ; dummy pop
      rts
10$
      jsr    SetGraphics               ; set graphics mode for this line
      PopW   r3                        ; restore the buffer pointer
      lda    #CARDSWIDE                ; load the card count (up to 80)
      sub    cardcount
      tax
20$
      PushX                            ; save x
      jsr    Rotate                    ; rotate the card
      jsr    SendBuff                  ; send the rotated card
      AddVW  #8,r3                     ; update pointer to buffer
      PopX                             ; recover x
      dex
      bne    20$                       ; if not, do another card
      rts
```

TestBuffer:

```
                                                                                .if 0
```

Function: Tests buffer to see if there is anything to print.

Called By: PrnPrintBuffer.

Parameters: **r3** pointer to beginning of print buffer to test.

Returns: carry flag 1 = if data in the buffer;
0 = no data in the buffer.

Destroys: a, x, **r3**.

Description: Check all the bytes in the buffer to see if all are 0.

```
                                                                              .endif
```

TestBuffer:

```
        LoadB   cardcount,#0
        LoadB   scount,#7                   ; assume 8-bit high printhead
        AddVW   #(CARDSWIDE-1)*8,r3
        ldx     #CARDSWIDE                  ; load the cards / line
10$
        ldy     scount
20$
        lda     (r3),y                      ; check a byte
        bne     99$                         ; if zero, skip to check another byte
        dey                                 ; point at next byte in card
        bpl     20$                         ; if not at end, check next byte in this card
        SubVW   #8,r3                       ; point at next card
        inc     cardcount
        dex                                 ; see if all the cards are done
        bne     10$                         ; if not done, do another card
        clc                                 ; if here, then line was clear
        rts
99$
        sec                                 ; set the carry to signal data was found
        rts
```

InitPrinter:

`.if 0`

Function: Initializes the Epson to line-feed 8/72 inch.

Called By: **StartPrint**.

Parameters: nothing.

Returns:		
	r3	#inittbl
	scount	$FF
	y	0

Destroys: a, x, **r3**.

Description: Outputs to the printer a string of characters which initializes it. See the printer's owners manual.

```
                                                                                                  .endif
InitPrinter:
      bit   modeflag                  ; see if printing ASCII or graphic mode
      bmi   10$                       ; branch if ASCII
      LoadW r3,#inittbl               ; table of bytes for initialization
      lda   #INITSZ                   ; length of string
      jmp   Strout                    ; output the string
10$
      LoadW r3,#ainittbl              ; table of bytes for ASCII initialization
      lda   #AINITSZ                  ; length of string
      jmp   Strout                    ; output the string

inittbl:
      .byte 8, 'A',ESC                ; send 8/72" line feed
      .byte '@',ESC                   ; reset totally
INITSZ=*- inittbl

ainittbl:
      .byte 2,ESC                     ; send 6 lines/inch
      .byte '@',ESC                   ; reset totally
AINITSZ=*- ainittbl
```

SetGraphics:

```
.if 0
```

Function: Sets the Epson into 640-column graphics mode.

Called By: PrnPrintBuffer.

Parameters: cardcount the number of the card being processed.

Returns:

r3	# wsdgphtbl, the printer width table.
scount	$FF
y	0

Destroys: a.

Description: Tell printer the graphics mode and how many bytes to expect.

```
                                                                                                   .endif
SetGraphics:
      LoadB  r3H,#0                     ; clear top byte
      MoveB  cardcount,r3L              ; load cardcount into low-byte
      asl    r3L                        ; x 8
      rol    r3H
      asl    r3L
      rol    r3H
      asl    r3L
      rol    r3H
      ;---   set total width for the page(bytes)
      SubVWS #CARDSWIDE*8,r3,wsdgphtbl
      LoadW  r3,#wsdgphtbl              ; table of control bytes for 640 col
                                        ; single density
      lda    #WSDGPHSZ                  ; length of string
      jmp    Strout                     ; output the string

wsdgphtbl:                              ; (Reverse Order String)
      .word  NULL                       ; N1 N2: Number of graphic bytes to output
      .byte  4,"*",ESC                  ; ESC * 4: set screen dump mode
                                        ; (80 dpi in graphics mode)
WSDGPHSZ = *-wsdgphtbl
```

SendBuff:

```
.if 0
```

Function: Sends a printable card out the serial port.

Called By: PrnPrintBuffer.

Uses: prntblcard.

Returns: nothing.

Destroys: a, x.

Description: After a card has been rotated so that the bytes each represent a vertical column of bits to go to the printer, SendBuff sends the card across the serial bus.

```
.endif
SendBuff:
      ldx   #0                         ; initialize the count
10$
      PushX                            ; save count
      lda   prntblcard,x               ; get byte to send
      jsr   Ciout                      ; send this byte
      PopX                             ; recover the count
      inx                              ; point at next byte
      cpx   #8                         ; are we done with all bytes?
      bne   10$                        ; if not, continue with sending
      rts
```

Greturn:

```
.if 0
```

Function: Set carriage / Line feed to printer.

Called By: **PrintBuffer**.

Parameters: nothing.

Returns: nothing.

Destroys: a.

Description: Outputs the CR/LF ($0D/$0A) pair to advance to beginning of next line.

```
                                                                                                  .endif
Greturn:
      lda    #CR
      jsr    Ciout                    ; send carriage return
      lda    #LF
      jsr    Ciout                    ; send linefeed
      rts
```

FormFeed:

```
.if 0
```

Function: Send the form feed command to the printer.

Called By: **PrintBuffer**.

Parameters: nothing.

Returns: nothing.

Destroys: a.

Description: Outputs a form feed ($0C) to advance printer to next page.

```
                                                                                 .endif
FormFeed:
      lda   #FF                     ; form feed
      jsr   Ciout                   ; send it
      rts
```

Rotate:

```
.if 0
```

Function: Rotates a hi-res bit mapped card from the 640-byte print buffer to an 8 byte buffer which is then ready for sending to the printer.

Called By: PrnPrintBuffer.

Parameters:	**r3**	address of the card to be operated on.
Returns:	prntblcard	rotated data placed here.

Destroys: a, x, y.

Description: Create the nth byte in the prntblcard buffer out of the nth bit of each of the bytes in the card pointed to by **r3**. This rotates a hires bit mapped card from the 640-byte print buffer pointed at by **r3** into the prntblcard 8-byte buffer.

```
.endif
Rotate:
      php                             ; save processor status register
      sei                             ; disable any IRQs
      ldy   #7                        ; initialize the index into the card
10$
      lda   (r3),y                    ; get the byte from the card
      ldx   #7                        ; initialize the index into the printable card
20$
      ror   a                         ; get the least significant bit into c
      ror   prntblcard,x              ; shift it into the printable card table
      dex                             ; next bit
      bpl   20$                       ; if not done, store another bit
      dey                             ; next byte
      bpl   10$                       ; if not done, load another byte
      plp                             ; restore interrupts to their saved state
      rts                             ; exit

PRINTEND:                             ; last label in Epson FX printer driver
```

app.Inc

```
.if 0
```

Function:	Application include	
Filename:	app.inc	Master Include. Symbols go to Linker / Debugger
Uses:	app.con	Application constants
	app.mac	Application macros
	app.sym	Application symbols

Callable Routines:
none.

Note: This can only be used one time as an include per application. Use app.inc for secondary source files.

```
                                                                                                   .endif
.if Pass1
      .noeqin                         ; never want to send CONSTANTS to linker
      .noglbl
            .include     app.con
            .include     app.mac
      .glbl
      .eqin
      .include     app.sym            ; all symbols will go to linker/debugger
.endif

.psect
```

app.con

```
                                                                                        .if 0
```

Function: Application constants

Filename: app.con

Uses: geo.con

Callable Routines:
none.

```
                                                                                      .endif
.include geo.con                       ; standard GEOS constants

;--- All constants only used by this application go here.
LOWERCASE    =      7                  ; command that does nothing
TRANSPARENT  =      5
CARDSWIDE    =      80                 ; 80 Commodore cards wide
CARDSDEEP    =      90                 ; 90 Commodore cards deep
CGPX         =      8                  ; graphic mode activation command
ECGPX        =      15                 ; graphic mode deactivation command
ESC          =      $1B
FF           =      $0C

ASCII        =      $FF
GRAPHIC      =      $00
```

app.sym

```
.if 0
```

Function: Application symbols.

Filename: app.sym

Uses: geo.sym

Callable Routines:
none.

```
.endif
.include geo.sym                        ; standard GEOS symbols (jump table and variables)

;--- All zero page declarations created for the application go here.

;--- All Symbols created for the application go here.


;      Global variables:
status       == $90                     ; serial command status
Ciout        == $FFA8                   ; transmit a byte over the serial bus
printBase    =  PRINTBASE

;--- prndrv.lib needs these constants to be global so they are declared here
SECADD       =     TRANSPARENT          ; secondary address
PRINTADDR    =     4                    ; Serial address of the printer
```

app.mac

```
                                                                                                   .if 0
```

Function: Application macros.

Filename: app.mac

Uses: geo.mac

Callable Routines:
none.

```
                                                                                                  .endif
.include geo.mac                             ; standard GEOS macros

;--- All macros created for the application go here.

.macro NegateW zaddr
       ldx    #[zaddr
       jsr    Dnegate
.endm
```

7-Bit MPS-801 Printer Driver

`.if 0`

Function: Sample 7-bit printer driver for MPS-801.

Files.

app.lnk	Link file
app.hdr.s	Header file
app.driver.s	Driver source
app.Inc	Master Include. Sends all to symbols debugger
app.inc	Secondary include. Sends nothing from includes to debugger
app.con	Application constants
app.sym	Application symbols
app.mac	Application macros

Callable Routines:

InitForPrint	->	rts	; not supported
StartPrint	->	_StartPrint	
PrintBuffer	->	_PrintBuffer	
StopPrint	->	_StopPrint	
GetDimensions	->	_GetDimensions	
PrintASCII	->	_PrintASCII	
StartASCII	->	_StartASCII	
SetNLQ	->	rts	; not supported

For: COMMODORE MPS-801, 1525

Tested on: COMMODORE MPS-801

Control Codes:

EnterGraphicMode	8
Line Feed	$0A
CR	$0D
DoubleWidth	$0E
StardardCharMode	$0F
TabSettingPrint Head	$10
Cursor Down Mode	$11
Reverse	$18
RepeatGraphicsSelected	$1A
Dot Address	$1B
CursorUpMode	$91
OffReverse	$92

`.endif`

app.lnk

Function: Linker file for MPS-801 Driver.

File name: app.lnk

Uses:

app.hdr.rel
app.driver.rel

```
.output      MPS-801
.header      app.hdr.rel
.seq

.psect       $7900        ;printBase

app.driver.rel
prndrv.lib.rel
```

app.hdr.s

```
                                                                                        .if 0
```

Function: Define File Header Block.

Filename: app.hdr.s Linker header file

Uses: app.con

Callable Routines:
None.

```
                                                                                       .endif
.if     Pass1
      .noeqin
      .include     app.con
      .eqin
.endif

.header                                 ; start of header section
      .word  0                          ; first two bytes are always zero
      .byte  3                          ; width in bytes
      .byte  21                         ; and height in scanlines of:
```

```
      .byte  $80 | USR                  ; Commodore file type assigned to GEOS files
      .byte  PRINTER                    ; GEOS file type
      .byte  SEQUENTIAL                 ; SEQ file structure
      .word  PRINTBASE                  ; start address for saving file data
      .word  PRINTEND                   ; end address of print driver
      .word  NULL                       ; not used (execution start address)
      ;---   20 byte permanent name
      .byte  "MPS-801-DrvrV1.1",0,0,0,0

      ;---   20 bytes for author name
      .byte  "OGPRG & PBM",NULL,0,0,0,0,0,0,0,0
.endh
```

app.driver.s

```
.if 0
```

Function: Main Source file for MPS-801 Driver

Filename: app.driver.s

Uses: app.Inc

Callable Routines:

InitForPrint	-> rts	;No action
StartPrint	-> _StartPrint	
PrintBuffer	-> _PrintBuffer	
StopPrint	-> _StopPrint	
GetDimensions	-> _GetDimensions	
PrintASCII	-> _PrintASCII	
StartASCII	-> _StartASCII	
SetNLQ	-> rts	;Not supported. No Action

```
                                                        .endif
.include     app.Inc
```

Jump Table

```
                                                                                      .if (0)
Jump Table to Print Driver Routines
                                                                                      .endif
;--- Input driver jump table

;InitForPrint:
             rts
             nop
             nop
;StartPrint:
             jmp    _StartPrint
;PrintBuffer:
             jmp    _PrintBuffer
;StopPrint:
             jmp    _StopPrint
;GetDimensions:
             jmp    _GetDimensions
;PrintASCII:
             jmp    _PrintASCII
;StartASCII:
             jmp    _StartASCII
;SetNLQ:
             rts
             nop
             nop

;--- RAM STORAGE / UTILITIES

;--- Local variables:

printerName:
       ;---   name of printer as it should appear in menu
       .byte "MPS-801",NULL
prntblcard:
       .block 8                        ; printable character block
breakcount:
       .byte  0
scount:
       .byte  0                        ; string output routine counter
cardcount:
       .byte  0                        ;
modeflag:
       .byte  0                        ; either 0=graphics, or $FF=ASCII
                                       ; for draft or nlq mode
linesLeft:
       .byte  0                        ; Blank lines remaining on page
                                       ; utility routines: (see "Print Driver Support Library")
```

_StartPrint:

```
.if 0
```

Function: **StartPrint** initializes the serial bus to the printer, sets up the printer to receive graphic data, and initialize the break count RAM location.

Called By: A GEOS application.

Parameters: nothing.

Returns: nothing.

Destroys a, x, y, **r3**.

```
                                                                              .endif
_StartPrint:
      LoadB  modeflag,#GRAPHIC         ; set for graphic mode
StartIn:
      lda    #PRINTADDR                ; set to channel 4
      jsr    SetDevice
      jsr    InitForIO                 ; set I/O space in, disable interrupts
      lda    #99
      bit    modeflag
      bpl    10$
      lda    #66
10$
      sta    linesLeft
      LoadB  breakcount,#0             ; initialize the counter for the card breaks
      sta    status                    ; initialize the error byte to no error
      jsr    OpenFile                  ; open the file for the printer
      lda    status                    ; if problems with the output channel, go to
      bne    20$                       ; error handling routine
      jsr    Delay                     ; wait for weird timing problem
      jsr    DoneWithIO                ; set mem map back, and enable interrupts
      ldx    #0
      rts
20$                                    ; save error return from the routines
      pha                              ; bit 0 set: timeout, write
                                       ; bit 7 set: device not present
      jsr    CloseFile                 ; close the file anyway
      jsr    DoneWithIO                ; set mem map back, and enable interrupts
      PopX                             ; recover the error return
      rts                              ; pass out in x

Delay:
      ldx    #0
10$
      ldy    #0
20$
      dey
      bne    $20
      dex
      bne    $10
      rts
```

_PrintBuffer:

```
.if 0
```

Function: **PrintBuffer** is the top level routine that dumps data from the GEOS 640-byte buffer maintained in the C64 to the printer using the serial port.

Called By: A GEOS application.

Parameters:

r0	BUFFER	— address of the 640-bytes (80 cards) to be printed.
r1	PBBUFFER	— address of an additional 640-byte buffer for **PrintBuffer** to use.

IMPORTANT: this memory pointed at by **r1** MUST stay intact between calls to the **PrintBuffer** routine. It is used as a storage area for the partial lines for the 7-bit high printers.

Returns: **r0**, **r1** preserved.

Destroys: a, x, y, **r3**.

```
.endif
_PrintBuffer:
      lda   #PRINTADDR                ; set to channel 4
      jsr   SetDevice                 ; set to printer device
      jsr   InitForIO                 ; put I/O space in and disable interrupts
      jsr   OpenPrint                 ; open channel to printer
      jsr   IPrintBuffer              ; print out a line
      jsr   ClosePrint                ; close the print channel
      jsr   DoneWithIO                ; put back the memory map, enable interrupts
      rts                             ; exit
```

IPrintBuffer:

`.if 0`

Function: Flush Buffer of any remaining lines.

Called By: A GEOS application.

Parameters:

r0	BUFFER	— address of the 640-bytes (80 cards) to be printed.
r1	PBBUFFER	— address of an additional 640-byte buffer for **PrintBuffer** to use.

IMPORTANT: This memory pointed at by **r1** MUST stay intact between calls to the **PrintBuffer** routine. It is used as a storage area for the partial lines for the 7-bit high printers.

Returns: **r0**, **r1** preserved.

Destroys: a, x, y, **r3**.

`.endif`

```
IPrintBuffer:
      jsr    TopRollBuffer            ; roll into print buffer
      MoveW  r1,r3
      jsr    PrnPrintBuffer           ; print it
      jsr    BotRollBuffer            ; roll leftover lines into print buffer
      CmpBI  breakcount,#7            ; see if we just print the last line in brktab
      bne    10$                      ; if not, skip
      MoveW  r1,r3                    ; point to print buffer
      jsr    PrnPrintBuffer           ; print it again
      LoadB  breakcount,#0            ; stuff breakcount
10$
      inc    breakcount               ; next index to breaks in 7-bit printing
                                      ; valid values = 1-7
      rts
```

_StopPrint:

```
.if 0
```

Function: StopPrint is called when a page of a document is finished or when the document itself is finished.

Parameters:

r0	BUFFER	— address of the 640-bytes (80 cards) to be printed.
r1	PBBUFFER	— address of an additional 640-byte buffer for **PrintBuffer** to use.

IMPORTANT: this memory pointed at by **r1** MUST stay intact between calls to the **PrintBuffer** routine. It is used as a storage area for the partial lines for the 7-bit high printers.

Returns: **r0**, **r1** unchanged.

Destroys a, x, y, **r3**.

Description: **StopPrint** is called after all cards for a given page have been sent to the printer. It does a **SetDevice**, **InitForIO**, makes the printer listen, and if the printhead was printing 7-bit high data, flushes out any remaining lines of data in the print buffer. It then does a form-feed and an unlisten, closes the Commodore output file, and does a **DoneWithIO**.

```
.endif
_StopPrint:
      lda    #PRINTADDR              ; set to channel 4
      jsr    SetDevice               ; set to printer device
      jsr    InitForIO               ; put I/O space in and disable interrupts
      jsr    OpenPrint               ; open channel to printer
      bit    modeflag                ; if ASCII printing
      bmi    10$                     ;    skip buffer flush
      PushW  r0                      ; save the buffer addresses
      PushW  r1
      MoveW  r0, r1                  ; load the address of RAM to clear
      LoadW  r0,#640                 ; length to clear
      jsr    ClearRam                ; clear it
      PopW   r1                      ; recover the buffer addresses
      PopW   r0
      jsr    IPrintBuffer            ; flush out the buffer data
10$
      jsr    FormFeed                ; do a form feed
      jsr    ClosePrint              ; close the print channel
      jsr    CloseFile               ; close the print file
      jsr    DoneWithIO              ; put RAM back in, enable interrupts
      rts
```

_GetDimensions:

```
.if 0
```

Function:	returns the number of cards wide and high that this printer is capable of printing out on an 8.5 by 11 inch page.
Called By:	A GEOS application.
Parameters:	nothing.
Returns:	a $00 (printer has graphics and text modes). x width, in cards, that this printer can put out across a page. y height, in cards, that this printer can put down a page.
Uses:	none.
Destroys	nothing.
Description:	**GetDimensions** returns the number of cards wide and high that the printer is capable of printing in an 8 x 10.5 inch rectangle. This rectangle is included entirely in the page, which usually measures 8.5 x 11 inches.

```
                                                                                                    .endif
_GetDimensions:
      ldx   #CARDSWIDE                ; get the number of cards wide
      ldy   #CARDSDEEP                ; and get the number of cards high
      lda   #GRAPHIC                  ; set for graphics or text driver
      rts
```

_StartASCII:

```
.if 0
```

Function: Sets the Commodore up to receive ASCII print streams.

Called By: A GEOS application.

Parameters: nothing.

Returns: nothing.

Destroys: a.

```
.endif
_StartASCII:
        LoadB  modeflag,#ASCII          ; set mode to ASCII printing
        jmp    StartIn                  ; pick up rest of start print
```

_PrintASCII:

```
.if 0
```

Function: Prints a null terminated ASCII string passed in the buffer pointed at by **r0**.

Called By: A GEOS application.

Parameters: none.

Uses:

	r0	pointer to the ASCII string.
	r1	pointer to the 640-bytes buffer for the printer driver to use.

Returns: nothing.

Destroys a, x, y.

```
.endif
_PrintASCII:
      lda    #PRINTADDR                ; set to channel 4
      jsr    SetDevice                 ; set to printer device
      jsr    InitForIO                 ; put I/O space in and disable interrupts
      jsr    OpenPrint                 ; open print channel
10$
      ldy    #0                        ; init the index into ASCII string
      lda    (r0),y                    ; get the character
      beq    80$                       ; if at end of string, exit
      cmp    #CR
      bne    20$
      dec    linesLeft                 ; reduce number of blank lines to feed at end of page
20$
      cmp    #'A'                      ; see if alpha char, for CBM ASCII conversion
      bcc    30$                       ; branch if not CR
      cmp    #'Z'+1
      bcs    30$
      eor    #%100000                  ; convert upper to lower and vice-versa
30$
      jsr    Ciout
      IncW   r0                        ; point to next character
      bra    10$                       ; do again
80$
      jsr    ClosePrint                ; close the print channel
      jsr    DoneWithIO                ; put RAM back in, enable interrupts
      rts
```

PrnPrintBuffer:

```
.if 0
```

Function: Prints out the print buffer pointed to by **r3**.

Called By: **PrintBuffer**.

Parameters: **r3** address of start of buffer to print.

Returns: nothing.

Destroys: a, x, y, **r0-r15**.

Description: Checks to see if the buffer is empty before printing the data. Then for each card in the buffer, rotate the data and send it to the printer.

```
.endif
PrnPrintBuffer:
      PushW  r3                         ; save the buffer pointer
      jsr    TestBuffer                 ; see if the buffer is all zeros
      bcs    10$                        ; if there is data in the buffer, send it
      PopW   r3                         ; dummy pop
      jsr    SetGraphics                ; set graphics mode for this line
      bra    80$
10$
      jsr    SetGraphics                ; set graphics mode for this line
      PopW   r3                         ; restore the buffer pointer
      lda    #CARDSWIDE                 ; load the card count (up to 80)
      sub    cardcount
      tax
20$
      PushX                             ; save x
      jsr    Rotate                     ; rotate the card
      jsr    SendBuff                   ; send the rotated card
      AddVW  #8,r3                      ; update pointer to buffer
      PopX                              ; recover x
      dex                               ; done?
      bne    20$                        ; if not, do another card
80$
      jsr    Greturn                    ; do graphics return here
      jsr    UnSetGraphics              ; get out of graphics mode
      rts
```

TopRollBuffer:

```
                                                                                .if 0
```

Function: Rolls the entire print buffer up the correct amount of lines for the previously unprinted lines to be printed over any new lines in the user buffer.

Called By: **PrintBuffer**.

Parameters:
r0 BUFFER — pointer to user buffer.
r1 PBBUFFER — pointer to my print buffer.

Returns: nothing.

Destroys: a, x.

```
                                                                               .endif
TopRollBuffer:
      PushW  r0                       ; save buffer pointers
      PushW  r1
      ldx    #CARDSWIDE-1             ; load the card count
10$
      ldy    breakcount               ; get the count for the break table index
      lda    topbreaktab,y            ; get the number of lines to roll
      jsr    RollaCard                ; rotate the card
      dex                             ; done?
      bpl    10$                      ; if not, do another card
      PopW   r1                       ; recover the pointers
      PopW   r0
      rts

topbreaktab:
      .byte 8,7,6,5,4,3,2,1
```

BotRollBuffer:

```
.if 0
```

Function: Rolls the entire print buffer up the correct amount of lines for the unprinted lines from the user buffer to be rolled into the bottom of the print buffer.

Called By: **PrintBuffer**

Parameters:

r0	BUFFER	—	pointer to user buffer.
r1	PBBUFFER	—	pointer to my print buffer.

Returns: nothing.

Destroys: a, x.

```
                                                                     .endif
BotRollBuffer:
      PushW  r0                          ; save buffer pointers
      PushW  r1
      ldx    #CARDSWIDE-1                ; load the card count
10$
      lda    breakcount                  ; get the count for the number of lines to roll
      jsr    RollaCard                   ; rotate the card
      dex                                ; done?
      bpl    10$                         ; if not, do another card
      PopW   r1                          ; recover the pointers
      PopW   r0
      rts
```

RollaCard:

```
.if 0
```

Function: Rolls a card from the user buffer into the print buffer lines.

Called By: **TopRollBuffer**, **BotRollBuffer**.

Parameters: a number of lines to roll.

Returns:	**r0**	**r0**+8.
	r1	**r1**+8.

Destroys: a, **r3L**.

```
                                                                          .endif
RollaCard:
        sta    r3L                         ; store the loop count
10$
        jsr    Roll8BOut                   ; shift out of the user buffer
        jsr    Roll8BIn                    ; and into the print buffer
        dec    r3L                         ; done?
        bne    10$                         ; if not, do another byte
        AddVW  #8,r0                       ; update pointer to user buffer
        AddVW  #8,r1                       ; update pointer to print buffer
        rts
```

TestBuffer:

```
                                                                   .if 0
```

Function:	Tests buffer to see if there is anything to print.
Called By:	PrnPrintBuffer.
Parameters:	**r3** BUFFER — pointer to beginning of print buffer to test.
Returns:	carry flag 1 = data in the buffer; 2 = no data in the buffer.
Destroys:	a, **r3**.
Description:	Check all the bytes in *BUFFER* to see if all are $00.

```
                                                                   .endif
TestBuffer:
      LoadB  cardcount,#0
      LoadB  scount,#6                   ; assume 7-bit high printhead
      AddVW  #(CARDSWIDE-1)*8,r3
      ldx    #CARDSWIDE-1                ; load the cards / line
10$
      ldy    scount
20$
      lda    (r3),y                      ; check a byte
      bne    99$                         ; if zero then skip to check another byte
      dey                                ; point at next byte in card
      bpl    20$                         ; if not at end, check next byte in this card
      SubVW  #8,r3                       ; point at next card
      inc    cardcount
      dex                                ; see if all the cards are done
      bpl    10$                         ; if not done, do another card
      clc                                ; if here, then line was clear
      rts
99$
      sec                                ; set the carry to signal data was found
      rts
```

Roll8BIn:

```
.if 0
```

Function:	Rotates 8 bytes through a, used in the routines to load the second 640-byte print buffer, the effect is to roll a line of cards up 1 line.	
Called By:	**RollaCard**.	
Parameters:	a	byte to fill in at bottom of card.
	r1	pointer to card to roll up 1 line.
Returns:	nothing.	
Destroys:	a, y.	

```
                                                              .endif
Roll8BIn:
      pha                         ; save the byte to fill in with
      ldy    #0                   ; initialize the index to the card
10$
      iny                         ; point at next line down (top byte is lost)
      lda    (r1),y               ; load a line from the card
      dey                         ; point at next line up
      sta    (r1),y               ; store the byte at the next line up
      iny                         ; point at next line down
      cpy    #7                   ; see if at last line in card
      bmi    10$                  ; if not, do more lines
      PopB   "(r1),y"             ; recover the byte to fill in
                                  ; store the byte at the bottom line
      rts
```

Roll8BOut:

```
.if 0
```

Function: Rotates 8 bytes through a, used in the routines to empty the first 640-byte print buffer, the effect is to roll a card up 1 line and leave the byte pushed out on top in a.

Called By: **RollaCard**.

Parameters: **r0** pointer to card to roll up 1 line.

Returns: a byte from the top of the card.

Destroys: y.

```
.endif
Roll8BOut:
      ldy   #0                    ; initialize the index to the card
      lda   (r0),y                ; load the top line from the card
      pha                         ; save the byte
10$
      iny                         ; point at next line down (top byte is lost)
      lda   (r0),y                ; load a line from the card
      dey                         ; point at next line up
      sta   (r0),y                ; store the byte at the next line up
      iny                         ; point at next line down
      cpy   #7                    ; see if at last line in card
      bmi   10$                   ; if not, do more lines
      pla                         ; recover the byte to fill in
      rts
```

SetGraphics:, **UnSetGraphics**:

`.if 0`

Function: Sets graphics mode.
Unsets Graphics Mode.

Called By: PrnPrintBuffer.

Parameters: nothing.

Returns: nothing.

Destroys: a.

`.endif`

```
      SetGraphics:
             lda       #CGPX           ; send character to set graphics mode
clda  UnSetGraphics,   #ECGPX          ; send character to unset graphics mode

             jsr       Ciout
             rts
```

SendBuff:

```
.if 0
```

Function: sends the prntblcard out to the serial port.

Called By: PrnPrintBuffer.

Uses: prntblcard.

Returns: nothing.

Destroys: a, x.

Description: Synopsis: After a card has been rotated so that the bytes each represent a vertical column of bits to go to the printer, SendBuff sends the card across the serial bus.

```
.endif
SendBuff:
      ldx    #0                        ; initialize the count
10$
      PushX                            ; save count
      lda    prntblcard,x              ; get byte to send
      ora    #$80                      ; add to get out of valid ASCII space
      jsr    Ciout                     ; send this byte
      PopX                             ; recover the count
      inx                              ; point at next byte
      cpx    #8                        ; are we done with all bytes?
      bne    10$                       ; if not, continue with sending
      rts
```

Greturn:

```
.if 0
```

Function: Set carriage / Line feed to printer.

Called By: **PrintBuffer**.

Parameters: nothing.

Returns: nothing.

Destroys: a

Description: Outputs the CR/LF ($0D/$0A) pair to advance to beginning of next line.

```
.endif
Greturn:
      lda   #CR                         ; carriage return
      jsr   Ciout                       ; send it
      rts
```

FormFeed:

```
.if 0
```

Function: Advance the remaining page from the printer.

Called By: **PrintBuffer**.

Parameters: nothing.

Returns: nothing.

Destroys: a

Description: For every blank line left to print advance the printer until the page is complete.

```
.endif
FormFeed:
     bit   modeflag                  ; if in graphics mode then
     bmi   PgFeed                    ;    feed rest of page
                                     ; else
     jsr   SetGraphics               ;    put printer in graphics mode
     jsr   PgFeed                    ;    feed rest of page
     jsr   UnSetGraphics             ;    turn off graphics mode
     rts                             ;    exit
PgFeed:                              ; loop
     lda   #$0D
     jsr   Ciout                     ; send carriage return to advance the page
     dec   linesLeft
     bne   PgFeed                    ; until page complete
     rts
```

Rotate:

```
.if 0
```

Function: Rotates a hi-res bit mapped card from the 640-byte print buffer to an 8 byte buffer which is then ready for sending to the printer.

Called By: PrnPrintBuffer.

Parameters: **r3** address of the card to be operated on.

Returns: prntblcard rotated data placed here.

Destroys: a, x, y.

Description: Create the nth byte in the prntblcard buffer out of the nth bit of each of the bytes in the card pointed to by **r3**. This rotates a hires bit mapped card from the 640-byte print buffer pointed at by **r3** into the prntblcard 8-byte buffer.

```
.endif
Rotate:
        php                                  ; save current interrupt disable status
        sei                                  ; disable interrupts
        ldy     #7                           ; initialize the index into the card
10$
        lda     (r3),y                       ; get the byte from the card
        ldx     #7                           ; initialize the index into the printable card
20$
        ror     a                            ; get the least significant bit into c
        ror     prntblcard,x                 ; shift it into the printable card table
        dex                                  ; next bit
        bpl     20$                          ; if not done, store another bit
        dey                                  ; next byte
        bpl     10$                          ; if not done, load another byte
        plp                                  ; restore old interrupt status
        rts
PRINTEND:                                    ; last label in the MPS-801 Printer driver
```

app.Inc

```
                                                                         .if 0
```

Function: Application Include

Filename:	app.Inc	Master Include. Symbols go to Linker / Debugger
Uses:	app.con	Application constants
	app.mac	Application macros
	app.sym	Application symbols

Callable Routines:

None.

Note: This can only be used one time as an include per application. Use app.inc for secondary source files.

```
                                                                        .endif
.if Pass1
       .noeqin                          ; never want to send CONSTANTS to linker
       .noglbl
              .include     app.con
              .include     app.mac
       .glbl
       .eqin
       .include     app.sym             ; all symbols will go to linker/debugger
.endif

.psect
```

app.con

```
.if 0
```

Function: Application constants

Filename: app.con

Uses: geo.con

Callable Routines:
none.

```
                                                                                   .endif
.include geo.con    ; Standard GEOS constants.

;--- All constants only used by this application go here.
LOWERCASE    =      7                   ; command that does nothing
CARDSWIDE    =      60                  ; 60 Commodore cards wide
CARDSDEEP    =      94                  ; 90 Commodore cards deep
CGPX         =      8                   ; graphic mode activation command
ECGPX        =      15                  ; graphic mode deactivation command
PRINTADDR    =      4                   ; Serial address of the printer
ASCII        =      $FF
GRAPHIC      =      $00
```

app.sym

```
.if 0
```

Function: Application symbols

Filename: app.sym

Uses: geo.sym

Callable Routines:
None.

```
.endif
```

```
.include geo.sym                ; standard GEOS symbols. (jump table and variables)

;--- All zero page declarations created for the application go here.


;--- All Symbols created for the application go here


;      Global variables:

status       == $90                     ; serial command status
Ciout        == $FFA8                   ; transmit a byte over the serial bus
printBase    = PRINTBASE

;--- prndrv.lib needs these constants to be global so they are declared here
SECADD       = LOWERCASE                ; secondary address
PRINTADDR    = 4                        ; Serial address of the printer
```

app.mac

```
                                                                                               .if 0
```

Function: Application macros

Filename: app.mac

Uses: geo.mac

Callable Routines:
None.

```
                                                                                              .endif
.include geo.mac                        ; standard GEOS macros

;--- All macros created for the application go here.

.macro NegateW zaddr
       ldx    #[zaddr
       jsr    Dnegate
.endm
```

Print Driver Support Library

`.if 0`

Function: Support Library for Print Drivers

Files:

prndrv.lib.s	Library source
app.inc	Secondary Include. Sends nothing from includes to debugger
app.con	Application constants
app.sym	Application symbols
app.mac	Application macros

Creates: prndrv.lib.rel

Note: To obtain more information on the serial bus transmission protocol of the C64 and its features, please refer to the official operation guide.

Callable Routines:

OpenFile	opens the Commodore structure of the file
CloseFile	closes the Commodore structure of the file
OpenPrint	prepares the printer for listening on the serial bus
ClosePrint	the communication with the printer on the serial bus ends
Strout:	transmits a string of bytes on the serial bus.

`.endif`

prndrv.lib.s

```
                                                                    .if 0
```

Function: Main Source file for Print Driver Support Library

Filename: prndrv.lib.s

Uses: app.inc

Callable Routines:

OpenFile	opens the Commodore structure of the file
CloseFile	closes the Commodore structure of the file
OpenPrint	prepares the printer for listening on the serial bus
ClosePrint	end communications with the printer on the serial bus
Strout	transmits a string of bytes on the serial bus.

```
                                                                    .endif
.include     app.inc
```

OpenFile:

```
.if 0
```

Function: Internal routine: prepares the file structure for communications with the printer through the serious bus.

Called By: **PrintBuffer**.

Parameters: none.

Returns: none.

Uses: none.

Destroys: a, x, y.

```
.endif
OpenFile:
      lda   #PRINTADDR                ; device number
      jsr   Listen                    ; directs the printer
      lda   #SECADD|$F0               ; load the secondary address for this
      jsr   Second                    ; printer and transmit
      jsr   Unlsn                     ; commands the printer to stop listening on the bus
      rts
```

CloseFile:

`.if 0`

Function: Internal routine: Disables the file structure for communications with the printer.

Called By: **PrintBuffer**.

Parameters: none.

Returns: none.

Uses: none.

Destroys: a, x, y.

`.endif`

```
CloseFile:
      lda   #PRINTADDR                  ; device number
      jsr   Listen                      ; directs the printer
      lda   #SECADD|$E0                 ; load the secondary address for this
      jsr   Second                      ; printer and transmit
      jsr   Unlsn                       ; commands the printer to stop listening on the serial bus
      rts
```

OpenPrint:

`.if 0`

Function: Internal routine: Initializes the printer listening on the serial bus.

Called By: **PrintBuffer**.

Parameters: none.

Returns: none.

Uses: none.

Destroys: a.

`.endif`

```
OpenPrint:
      lda   #PRINTADDR                ; device number
      jsr   Listen                    ; directs the printer
      lda   #SECADD|$60               ; load the secondary address for this printer
      jsr   Second                    ; Transmit
      rts
```

ClosePrint:

```
.if 0
```

Function: Internal routine: Initializes the printer listening on the serial bus.

Called By: **PrintBuffer**.

Parameters: none.

Returns: none.

Uses: none.

Destroys: a, x, y.

```
                                                                        .endif
ClosePrint:
      jsr   Unlsn                    ; commands the printer to stop listening
                                     ; on the serial bus
      rts
```

Strout:

`.if 0`

Function: Strout (string out) transmits the string of characters pointed to by **r3**.

Called By: **PrintBuffer**

Parameters:

r3	STRING — string to be transmitted (data must be arranged in reverse order).
a	number of bytes to be transmitted.

Alters: scount

Return:

scount	$FF
y	$00

Destroys: a.

`.endif`

```
Strout :
      sta   scount                      ; save the index
      dec   scount
10$
      ldy   scount                      ; load the index
      lda   (r3),y                      ; get the byte
      jsr   Ciout                       ; send it
      dec   scount                      ; update the index
      bpl   10$                         ; if the table is not finished, proceed
                                        ; with the next character
      rts
```

app.inc

```
                                                                                               .if 0
```

Function: Application include

File name: app.inc

Uses:		
	app.con	Application constants
	app.mac	Application macros
	app.sym	Application symbols

Callable Routines:
none.

Note: This sends nothing from the includes to the linker. Use this when app.Inc has been used in a main source file, or from the main source file if you don't care about debugger information.

```
                                                                                               .endif
.if Pass1
      .noeqin                        ; never want to send CONSTANTS to linker
      .noglbl
            .include     app.con
            .include     app.mac
            .include     app.sym      ; no symbols will go to linker/debugger
      .glbl
      .eqin
.endif

.psect
```

app.con

```
                                                                                   .if 0
```

Function: Application constants

File name: app.con

Uses: geo.con

Callable Routines:
None.

```
                                                                                  .endif
.include geo.con                        ; standard GEOS constants

;--- All constants only used by this application/Library go here.
```

app.sym

```
                                                                                   .if 0
```

Function: Application symbols.

File name: app.sym

Uses: geo.sym

Callable Routines:
None.

```
                                                                                  .endif
.include geo.sym                      ; standard GEOS symbols (jump table and variables)

;--- All zero page declarations created for the application go here.

;--- All symbols created for the application go here.


;     Global variables:

;--- We normally don't send any constants to the linker.
;--- If we need one to go to linker for use in the .lnk file or other linker resolutions
;--- then need to redefine here.

Acptr        = $FFA5                  ; Input byte from serial port
Ciout        = $FFA8                  ; Transmit a byte over the serial bus
Listen       = $FFB1                  ; Command a device on the serial bus to listen
Second       = $FF93                  ; Send secondary address for listen
Unlsn        = $FFAE                  ; Send an UNLISTEN command
Untlk        = $FFAB                  ; Send an UNTALK command
```

app.mac

```
                                                                                                   .if 0
```

Function: Application macros.

File name: app.mac

Uses: geo.mac

Callable Routines:
None.

```
                                                                                                 .endif
.include geo.mac                       ; Standard GEOS macros.

;--- All macros created for the application go here.
```

BitOtherClip 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
       error:         .block 1          ; temp holder for disk errors
       curPhoto:      .block 1          ; Record to use
       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:

```
.if 0
```

Function:	Read a picture in from a photo album record and draw it clipped to a window. Scroll values allow a specific portion of the bitmap to be shown.	
Parameters:	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.
	picLength	from photo record.
	x	error ($00 = NO_ERROR).
Destroys:		

```
.endif
DrawPhoto:
      jsr   ClearWindow              ; clear the drawing window
      jsr   GetPicSize               ; get the size of the picture
      txa                            ; check for error
      bne   99$                      ; carry comes back set if we can draw
      jsr   SetUpPhoto               ; set up clipping parameters
      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:

```
                                                                                                   .if 0
```

Function: Erase the window areas where we plan to put the bitmap.

Parameters: nothing.

Returns:

Alters: **curPattern** = pattern 0.

Destroys: a, x, y, **r5-r8**.

```
                                                                                                  .endif
ClearWindow:
     lda     #0                        ; use blank fill pattern
     jsr     SetPattern
     jsr     i_Rectangle
     .byte   WIN_Y
     .byte   (WIN_Y+WIN_HEIGHT)
     .word   (WIN_CRDX * 8)
     .word   (WIN_CRDX*8 + WIN_CRDWIDTH*8)
     rts
```

SetUpPhoto:

.if 0

Function:	Set up clipping regions and other parameters.	
Parameters:	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.	

.endif

```
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               ; accumulator <- (picWidth-leftOffset)
                                        ; (difference between width of
        sub      leftOffset             ; picture and offset into picture)
        bcc      99$                    ; if offset exceeds width, then skip
        beq      99$                    ;    over picture draw
        cmp      #WIN_CRDWIDTH          ; if width to display exceeds width of bitmap then
        bcc      10$                    ;    display as much as
        lda      #WIN_CRDWIDTH          ;    will fit in the window.
10$                                     ;
        sta      r2L                    ; r2L <- card width to display
                                        ;
        lda      picLength              ; accumulator <- (picLength-topOffset)
                                        ; (difference between height of
        sub      topOffset              ; picture and offset into picture)
        bcc      99$                    ; if offset exceeds height, then
        beq      99$                    ;    skip over picture draw
        cmp      #WIN_HEIGHT            ; if height to display exceeds height
        bcc      20$                    ;    of bitmap, then display as much as
        lda      #WIN_HEIGHT            ;    will fit in the window.
```

```
20$                                     ;
       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                ;
       sub      r2L                     ;
       sbc      leftOffset              ;
       sta      r11H                    ; r11H <- cards to skip on right
       clc                              ; flag as OK to draw
       rts                              ; exit
99$                                     ;
       sec                              ; flag as not to draw
       rts                              ;
```

PutUpPhoto:

```
.if 0
```

Function: Draw photo from record.

Parameters: nothing.

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

Destroys: a, x, y, **r0-r15**.

```
                                                                     .endif
PutUpPhoto:
     jsr     GetPicSize                 ; reload picture length and width
     txa                                ; check for error or no picture
     bne     99$                        ; leave on error
;    --- No need to preload r0. Sync is called before the first byte of each packet is retrieved.
;    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:

*Part of **BitOtherClip Example** on how to handle APPINPUT.*

```
.if 0
```

Function: Bitmap input routine called by **BitOtherClip**. Returns a single byte of the uncompacted bitmap into buffer pointed to by (**r0**).

Parameters: **r0** BUFFER — Active *BUFFER* being used by **BitOtherClip**.

Uses: saveR1, saveR5, **BitOtherClip** active parameters.

Returns: bitmap byte in **BitOtherClip's** buffer (off of **r0**) any error in error.

Destroys: a, y.

```
.endif
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    ReadByte                    ; get a byte from the file
                                         ; (byte is in A)
      stx    error                       ; save any error
      ldy    #0                          ; null indirection index
      sta    (r0),y                      ; store byte into buffer
      MoveW  r5,#saveR5                  ; r5 -> saveR5
      MoveW  r1,#saveR1                  ; r1 -> saveR1
      PopW   r5                          ; restore r1, r4, and r5
      PopW   r4                          ;
      PopW   r1                          ;
      rts                                ; exit

.if 0
```

Function: Dumb synchronization routine needed by **BitOtherClip**.
Resets **r0** buffer pointer back to start of buffer.

Uses: clipBuffer.

Alters: **r0**.

Returns: **r0** set to start of buffer.

Destroys: a.

```
.endif

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

GetPicSize:

```
                                                                                        .if 0
```

Function: Get picture size and other misc. setup for PutUpPhoto

Returns: x error.

```
                                                                                       .endif
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
     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:

```
.if 0
```

Function: Set up variables and stuff for **ReadByte**

Parameters: **r1L**, **r1H** track/sector of first block in chain

Returns: **r1**, **r4**, **r5** set up for **ReadByte**
x error ($00 = no error)

Destroys: a, y.

```
.endif
SetUpReadByte:
     ldx      #NO_ERROR
     MoveW    r4,r1                     ; r1 <- track/sector of 1st block
     LoadW    r4,#diskBlkBuf            ; r4 <- disk buffer for ReadByte
     LdNull   r5                        ; r5 <- $0000 (for ReadByte)
     rts                                ; exit
```

```
.if 0
```

Function: Set up variables and stuff for **ReadByte**.

Parameters: **r1L**, **r1H** track/sector of first block in chain.

Returns: **r1**, **r4**, **r5** set up for **ReadByte**.
x error ($00 = no error).

Destroys: a, y.

```
.endif
ReadSizeBytes:
     jsr      ReadByte                  ; get photo width
     sta      picWidth
     txa                                ; check for error
     bne      99$
     jsr      ReadByte                  ; get photo length (low-byte)
     sta      picLength
     txa                                ; check for error
     bne      99$
     jsr      ReadByte                  ; get photo length (high-byte)
     sta      picLength+1
99$
     rts                                ; exit error in x
```

Compact Bitmap

`.if 0`

Name: 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 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.

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 grab data from) and...

```
Cnvrt40:
      ldx    yPos               ; get y-coordinate of top of bitmap
      jsr    GetScanLine        ; use it to calculate screen pointers
      lda    xPos               ; get x-coordinate (low-byte)
      and    #%11111000         ; strip off 3 bits for card x-position
                                ; add card offset to
      add    r5L                ; base pointer (low-byte first)
      sta    r5L
      lda    xPos+1             ; (high-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   graphMode                  ; make sure in 80-column mode
      bpl   Cnvrt40                    ; handle 40 like C64
      PushB dispBufferOn               ; save current dispBufferOn
      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 register
      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 address

      ;---  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).

Parameters:	**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 uncompacted 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.
Destroys:	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.

`.endif`

```
MAX_REPEAT   =      127                 ; maximum repeat COUNT value
MAX_UNIQUE   =      191                 ; 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                         ; initialize index into buffers
      sta    (r0),y                     ; store repeat # to destination
      lda    (r1),y                     ; get repeat value
      iny                               ; point to next byte in destination buffer
      sta    (r0),y                     ; store to destination buffer
      AddVW  #2,r0                      ; move up destination pointer
      bra    80$                        ; exit
20$
                                        ; use UNIQUE
      jsr    GetUnique                  ; calc # of unique bytes to use
                                        ; (A = number of unique)
      ldy    #0                         ; initialize index into buffers.
      ora    #$80                       ; convert unique count to packet count value
      sta    (r0),y                     ; store to destination buffer
30$
      lda    (r1),y                     ; get first unique value
      iny                               ; increment 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 destination pointer
      AddBW  r5L,r0                     ; move up destination pointer
      dec    r5L                        ; correct back to # done
                                        ; fall through to exit
80$
      AddBW  r5L,r1                     ; move up source pointer
      SubBW  r5L,r2                     ; subtract off #left in source buffer
      bwne   r2,10$                     ; Loop until r2=0
      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                         ; initialize 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
20$
      cpx   r2L                        ; all of them?
      beq   90$                        ; if yes, then that many
      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
30$                                    ; yes, close this UNIQUE packet
      PopW  r1                         ; retrieve start pointer
      lda   r5L                        ; get # to do unique
      rts
```

ChangeMode:

`.if 0`

Function: Change Video Mode in GEOS 128.

Parameters: nothing.

Returns: nothing.

Destroys: a, x, y, **r0**.

`.endif`

```
GREYPAT=2

ChangeMode:
       tmbf   7,graphMode                ; toggle 40/80 bit
       jsr    SetNewMode                 ; Set new video mode
       jsr    GreyScreen                 ; grey out new screen
       rts                               ; exit


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_PIX_WIDTH-1) | DOUBLE_W | ADD1_W
     .byte    SC_PIX_HEIGHT-1
     .byte    NULL
     rts
```

Check128:

.if 0

Function: Check for GEOS 128.

Parameters: nothing.

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

.endif

```
Check128:
    lda     #$12                    ; c128Flag not valid until version 1.3
    cmp     version                 ; first see if version <= 1.2
    bpl     10$                     ; if so; branch and say C64
    lda     c128Flag                ; else set minus based on high bit c128Flag
10$
    rts

Example usage:
    jsr     Check128
    bpl     10$                     ; ignore if under GEOS 64
    jsr     DoDeDoubling            ; else, patch x-coordinates to remove doubling bits
10$
    .
    .
    .
```

DblDemo1:

`.if 0`

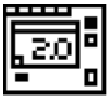
Function: Will assemble differently depending on the status of the C64 and C128 assembly constants. If assembling for GEOS 64, doubling constants will be set to zero so that they will not affect the x-positions. If assembling for GEOS 128, doubling constants will be set according to GEOS Constants file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

`.endif`

```
.if !(C128 ^^ C64)                     ; C64/C128 flags must be mutually exclusive!
       .echo DblDemo not designed to assemble for both GEOS 64 and GEOS 128!
.else
       .if    !C128                    ; if not assembling for GEOS 128, force
                                       ; doubling constants to harmless values so
                                       ; GEOS 64 graphics routines
                                       ; don't get confused.
             DBLE_B = 0                ; Note³: geoAssembler.x cannot do reassignment
             DBLE_W = 0                ; need a new equate to hold the conditional
             AD1_W  = 0                ; value.
       .else                           ;
             DBLE_B = DOUBLE_B         ; if this logic block was in the CONSTANTS
             DBLE_W = DOUBLE_W         ; file it could set DOUBLE_B, DOUBLE_W, ADD1_W as
             AD1_W  = ADD1             ; needed and then all of the code base would
       .endif                          ; use those values.

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

Bitmap:
```



```
BM_WIDTH = picW                        ; byte bitmap width (40-col)
BM_HEIGHT = picH                       ; byte bitmap width (40-col)

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|DBLE_B)          ; xPos
       .byte (BM_YPOS                  ; yPos
       .byte (BM_WIDTH|DBLE_B)         ; width
       .byte BM_HEIGHT                 ; height
90$
       rts                             ; exit
       ;--- (both C128 & C64 constants were both TRUE or both FALSE)
.endif
```

DisplayImage:

`.if 0`

Function: General purpose routine to display a portion of compacted bitmap image in a window.

Parameters:

pixBuf	compacted bitmap image in pseudo-photo scrap 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**.

`.endif`

```
.ramsect
       xoffset:
             .block 1                  ; card x index into bitmap (byte)
       yoffset:
             .block 2                  ; pixel y index into bitmap (word)

       ;--- 2K picture buffer
       pixWidth:
             .block 1                  ; width of picture in cards (byte)
       pixHeight:
             .block 2                  ; height of picture in pixels (word)
       pixImage:
             .block $800-3             ; start of bitmap image

.psect

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 x to window
       lda      pixWidth               ; get bitmap width
       sec                             ;
       sbc      #WINDOW_WIDTH          ;
       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$     ;---      clip y to window
        ;         subtract window height from bitmap height
        ;         store intermediate result in r3
        SubVWS    #WINDOW_HEIGHT,pixHeight,r3
        SubW      r12,r3                 ; now subtract y-index into bitmap
        bpl       20$                    ; branch if no underflow
        lda       r3L
        adc       #WINDOW_HEIGHT         ; correct for underflow
        sta       r2H
20$     jsr       BitmapClip             ; display the bitmap with clipping
        rts                              ; exit
```

FilledRect:

```
.if 0
```

Function: Draw a filled rectangle using the current pattern.

Parameters: none.

Returns: none.

Destroys: a, x, y, **r2-r9**, **r11**.

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

FilledRect:
       jsr     i_Rectangle             ; inline call
       .byte   Y1,Y2                   ; y1, 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
       .byte   Y1,Y2                   ; y1, y2
       .word   (X1|DOUBLE_W)           ; x1 with doubled width
       .word   (X2|DOUBLE_W|ADD1_W)    ; x2 with doubled width + offset for frame
       .byte   $FF
       rts


;---   size optimized Version
;      saves 7 bytes over the original version of FilledRect
;      while achieving the same result

FilledRect:
       jsr     i_Rectangle             ; inline call
       .byte   Y1,Y2                   ; y1, y2
       .word   (X1|DOUBLE_W)           ; fill full size of final rectangle
       .word   (X2|DOUBLE_W)           ;
                                       ; X (r3, r4) and Y (r2L, r2H) are set and returned
                                       ; unchanged by i_Rectangle

       lda     #$FF                    ; set line pattern
       jmp     FrameRectangle          ; frame full size of rectangle
```

GrphcsStr:

```
.if 0
```

Function: Draw a simple rectangle with pattern 0.

Uses: upper left corner at (xUL, yUL)
and lower right at (xLR, xLB).

```
.endif
GrphcsStr:
        jsr i_GraphicsString
        .byte   NEWPATTERN,0
        .byte   MOVEPENTO
        .word   xUL
        .byte   yUL
        .byte   RECTANGLETO
        .word   xLR
        .byte   xLB
        .byte   NULL
        rts

;---   Draw Berkeley Softworks plaque and display copyright text inside it.
;      Doubling information is in the x-coordinates.
;      The application this came from is compatible with all modes of GEOS at runtime.

  BSW_Sig:
        jsr     i_GraphicsString
        .byte   NEWPATTERN,1            ; draw shadow
        .byte   MOVEPENTO
        .word   DOUBLE_W | 48
        .byte   148
        .byte   RECTANGLETO
        .word   DOUBLE_W | 288
        .byte   196
        .byte   NEWPATTERN,0            ; draw background of plaque
        .byte   MOVEPENTO
        .word   DOUBLE_W | 40
        .byte   140
        .byte   RECTANGLETO
        .word   DOUBLE_W | 280
        .byte   188
        .byte   FRAME_RECTO             ; frame top section
        .word   DOUBLE_W | 40
        .byte   140
        .byte   FRAME_RECTO             ; frame bottom section
        .word   DOUBLE_W | 280
        .byte   170
        .byte   ESC_PUTSTRING           ; now put application name using PutString
        .word   DOUBLE_W | 136
        .byte   152
        .byte   BOLDON
        .byte   "geoAssembler'"
        .byte   GOTOXY                  ; go to new XY for copyright
        .word   DOUBLE_W | 108
        .byte   164
        .byte   PLAINTEXT               ; and print it
        .byte   "Copyright 1987 Berkeley Softworks",NULL
        rts
```

MseToCardPos:

```
.if 0
```

Function: converts current mouse positions to card position.

Parameters: nothing.

Uses: **mouseXPos**, **mouseYPos**.

Returns:

r0L	mouse card x-position (byte).
r0H	mouse card y-position (byte).

Destroys: a, x, y.

```
.endif
```

```
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 (r0)
    lda       mouseYPos         ; get mouse y-position while interrupts are disabled
    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
    lsr       a                 ; shift y-position in accumulator 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 card y-position
    rts                         ; exit
```

Note: If you do not disable interrupts prior to getting the value of **mouseXPos** you could get **r0H** with lda/sta and before getting **r0L** an interrupt occurs and the mouse position is updated during the interrupt. Now, when you do lda/sta for **r0L**, it is for a different **mouseXPos** reading giving unpredictable results.

Note^3: By also getting the Y value while interrupts are disabled, you are guaranteed to get a consistent reading for all three parts of the mouse position.

The three mouse position parts that have to be read, and the order that they are normally read:

mouseXPos+1	(byte)	high-byte of x-position word.
mouseXPos	(byte)	low-byte of x-position word.
mouseYPos	(byte)	y-position.

If interrupts are enabled while reading these three values the interrupt could occur between the read of **mouseXPos**+1 and **mouseXPos**, making **mouseXPos**+1 a completely unrelated value to **mouseXPos** and **mouseYPos**. The same is true if the interrupt occurs after **mouseXPos** is read. In this case the **mouseXPos** word will be from one sampling of the mouse and the **mouseYPos** byte will be from a different and unrelated later sampling.

ShowBitmap:

`.if 0`

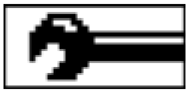
Function: ShowBitmap.

Note: For C64 and C128:
Showing compile time handling of C64/C128 differences with x-position.

`.endif`

```
.if C128
    DOUBLE_B = %10000000
.else
    DOUBLE_B = NULL
.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:
    LoadB    dispBufferOn,#(ST_WR_FORE | ST_WR_BACK)

;---bug fix for 128 release 1. (Not needed for 2.0+)
.if (C128)
    jsr      TempHideMouse              ; remove sprites
.endif

    jsr      i_BitmapUp                 ; inline bitmap call
    .word    Bitmap                     ;   *bitmap address
    .byte    BM_XPOS | DOUBLE_B         ;   *x-position
    .byte    BM_YPOS                    ;   *y-position
    .byte    BM_WIDTH | DOUBLE_B        ;   *width
    .byte    BM_HEIGHT                  ;   *height

90$
     rts                                ; exit
```

StopMenus:

```
.if 0
```

Function:	Example of how to temporarily disable menus and then restart them at a later time.		
Note:	jsr	StopMenus	will stop menu processing.
	jsr	RestartMenus	will return menu processing to its prior state.

```
.endif
oldMouseOn:
    .byte    0                          ; temp save area for mouseOn variable

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

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
```

i_VerticalLine:

```
.if 0
```

Function: Inline version of **VerticalLine**.

Parameters: Inline:

```
.word  x1
.word  x2
.byte  y1
```

```
                                                                                                        .endif
V_BYTES = 5                         ; number of inline bytes in call

i_VerticalLine:
      ;---   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
      php                              ; save st register to return
      lda    #V_BYTES +1               ; # of bytes + 1
      jmp    DoInlineReturn            ; jump to inline return. DO NOT jsr!
```

GetFPS:

`.if 0`

Author: PBM.
Parameters: nothing.
Returns: a = fps.
minus flag set if known model was not found.

Note: minus return should never happen without a bug in C64Model.

`.endif`

```
models:
       .byte %00,%01,%10,%11
NBR_MODELS=*-models

frates:
       .byte 50,60,60,50

GetFPS:
       jsr    C64Model
10$
       ldx    #NBR_MODELS-1
       cmp    models,x
       beq    90$
       dex
       bpl    10$
       lda    #[TRUE
       rts
90$
       lda    frates,x
       rts
```

C64Model:

`.if 0`

Function: Detect PAL/NTSC.

Original Name:
DetectC64Model.

Author: TWW.

Description:
312 rasterlines -> 63 cycles per line PAL
=> 312 * 63 = 19656 Cycles / VSYNC => #>76 %00
262 rasterlines -> 64 cycles per line NTSC V1
=> 262 * 64 = 16768 Cycles / VSYNC => #>65 %01
263 rasterlines -> 65 cycles per line NTSC V2
=> 263 * 65 = 17095 Cycles / VSYNC => #>66 %10
312 rasterlines -> 65 cycles per line PAL DREAN
=> 312 * 65 = 20280 Cycles / VSYNC => #>79 %11

`.endif`

```
C64Model:
      ;---   Use CIA #1 timer B to count cycles in a frame
      lda    #$FF
      sta    cia1tblo
      sta    cia1tbhi                ; latch #$FFFF to timer B
10$
      bbrf   7, grcntrl1, 10$        ; wait until raster > 256

20$
      bbsf   7, grcntrl1, 20$        ; wait until raster = 0

      ldx    #%00011001
      stx    cia1crb                 ; start timer B (one shot mode
                                     ; (timer stops automatically when underflow))
30$
      bbrf   7, grcntrl1, 30$        ; wait until raster > 256

40$
      bbsf   7, grcntrl1, 40$        ; wait until raster = 0

      sub    cia1tbhi                ; high-byte number of cycles used
      and    #%00000011
      rts
```

DetectC64Model source from CodeBase64:
https://codebase64.org/doku.php?id=base:detect_pal_ntsc

Sta80Fore:

```
.if 0
```

Function: Stores byte to 128 80-column foreground screen.

Parameters: **r5** address in foreground memory.
a data value (for Sta80Fore).

Returns: a data value (for Lda80Fore).

Destroys x.

Note: Call **TempHideMouse** to disable software sprites before writing foreground screen directly.

```
                                                                                            .endif
R18_UAH      = $12              ; update high-byte of VDC pointer
R19_UAL      = $13              ; update low-byte of VDC pointer
R31_DA       = $1F              ; data byte at current VDC pointer
vdccr        = $D600
vdcdr        = $D601

Sta80Fore:
;---    Send data byte to the VDC chip
        jsr     NewVDCAddress   ; update VDC address with foreground screen pointer (r5)
        ldx     #R31_DA         ; request VDC data register
        stx     vdccr           ;
10$     bit     vdccr           ; test VDC status
        bpl     10$             ; loop till VDC ready for data byte
        sta     vdcdr           ; store data byte
        rts                     ; exit
```

Lda80Fore:

`.if 0`

Function: loads byte from 128 80-column foreground screen.

Parameters: **r5** address in foreground memory.

Returns: a data value from VDC screen memory.

Destroys x.

`.endif`

```
Lda80Fore:
       jsr      NewVDCAddress   ; update VDC address with foreground screen pointer (r5)
       ldx      #R31_DA         ; request VDC data register
       stx      vdccr           ;
10$    bit      vdccr           ; test VDC status
       bpl      10$             ; loop till VDC ready for data byte
       lda      vdcdr           ; get data byte
       rts                      ; exit
```

NewVDCAddress:

`.if 0`

Function: Set VDC Memory pointer to address in **r5**.

Parameters: **r5** address in foreground memory.

Returns: nothing.

Destroys: x.

Description: Transfer value in **r5** to VDC internal hi/lo address register.

Note: Call **TempHideMouse** to disable software sprites before writing foreground screen directly.

`.endif`

```
NewVDCAddress:
        ldx     #R18_UAH
        stx     vdccr           ; ask VDC for high-byte
10$     bit     vdccr           ; check VDC status
        bpl     10$             ; and loop till VDC ready
        ldx     r5H             ; store high-byte of address
        stx     vdcdr           ; to VDC chip

        ldx     #R18_UAL        ; ask VDC for low-byte
        stx     vdccr           ;
20$     bit     vdccr           ; check VDC status
        bpl     20$             ; and loop till VDC ready
        ldx     r5L             ; store low-byte of address
        stx     vdcdr           ; to VDC chip
        rts                     ; exit
```

IconsUp:

```
                                                                                    .if 0
```

Function: Install an icon table.

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 example **NoIcons**).

```
                                                                                    .endif
IconsUp:
    ;---     draw to both buffers
    LoadB    dispBufferOn,#(ST_WR_FORE | ST_WR_BACK)
    LoadW    r0,#IconTable             ; put pointer to table in r0
    jmp      DoIcons                   ; activate the icons and exit
```

mainMenu:

```
.if 0
```

Function: Sample Menu Table.

Description: Define 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).

```
.endif

;--- Menu bounding rectangle
MAINX1      = 0                    ; left-edge
MAINY1      = 0                    ; top-edge
MAINX2      = 72                   ; right-edge
MAINY2      = MAINY1 + M_HEIGHT    ; bottom-edge
M_ITEMS     = 3

;*******************************************************
; 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

;--- 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
```

NoIcons:

```
.if 0
```

Function: 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**.

```
.endif
```

```
DummyIconTable:
       .byte 1                        ; one icon
       .word NULL                     ; dummy mouse x (don't reposition)
       .byte NULL                     ; dummy mouse y
       .word NULL                     ; bitmap pointer to $0000 (disabled)
       .byte NULL                     ; dummy x-position
       .byte NULL                     ; dummy y-position
       .byte 1,1                      ; dummy width and height
       .word NULL                     ; dummy event handler

NoIcons:
      LoadW r0,#DummyIconTable        ; point to dummy icon table
      jmp DoIcons                     ; install. let DoIcons rts
```

Keyboard Entry Routine

Constants and Variables

```
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

SPACE        = 32                       ; first printable character code

.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 the indirect jump vector straddles a page boundary, fix it to compensate for a bug
;      in the 6502 architecture.
;      To use this logic we must know the base address that the .rel file will use.
;      PSB (PSect Base address) is where we KNOW the .rel will be linked.
;      * is the current psect offset.
;      PSB+* = current real memory pointer of final code.
;      (warning: if the link address does not match PSB, it will break this logic).
;      if PSB is not a known value then this logic block cannot be used.
;      Use a fixed location in .zsect or .ramsect that does not span a page boundary instead.
PSB    = $400                           ; psect address this code will be linked at
.if ((PSB+* & $FF) == $FF)              ; if the real memory address is on the edge of a boundary
             .block 1                   ;    allocate 1 byte to move the vector to the next page
.endif

       bufFaultVec:
             .block 2                   ; vector cannot span a page boundary!

       tempDisp:
             .block 1                   ; temporary hold for dispBufferOn

       sysKeySave:
             .block 2                   ; holds address of system key routine

.psect
```

Table of control keys

```
	;--- Keys and their corresponding routines
ctrlKeys :
	.byte  CR                        ; 1 Carriage return
	.byte  BACKSPACE                 ; 2 backspace
	.byte  KEY_DELETE                ; 3 ditto
	.byte  KEY_INSERT                ; 4 ditto    ; Insert is a shifted Delete.
	.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
	.byte  ]DoBackSpace
	.byte  ]DoBackSpace
	.byte  ]DoBackSpace
	.byte  ]DoBackSpace
```

StartText:

```
.if 0
```

Name: StartText.

Function: Initialize 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.

Parameters: nothing.

Returns: text input routine in **keyVector**.

Destroys: assume a, x, y, **r0-r15**.

```
.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

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

	LoadW	leftMargin,#TXT_LEFT      ; set the left and right-margins
	LoadW	rightMargin,#TXT_RIGHT
	LoadW	windowTop,#TXT_TOP        ; set the top and bottom-margins
	LoadW	windowBottom,#TXT_BOT

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

	lda	curHeight                 ; initialize the prompt
	jsr	InitTextPrompt
	jsr	PromptOn

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

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

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

	LoadB	textOn,#[TRUE             ; turn text on
	rts                               ; exit
```

KeyIn:

```
                                                                .if 0
```

Function: **keyVector** handler. Control comes here off of **MainLoop** when a key is pressed.

Uses: **keyData**, **menuNumber**.

```
                                                                .endif
```

```
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:

```
.if 0
```

Function: Process Shortcut Keypresses.

Parameters: a.

Description: Control comes here when shortcut keys are pressed.

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

NormalKey:

```
.if 0
```

Function: Process Non-Shortcut Keypresses.

Parameters: a.

Uses:

Returns:

Description: Control comes here when non-shortcut keys are pressed.

```
                                                                  .endif
NormalKey:
      ;---  Return immediately if text is off
      lda   textOn
      bne   10$                       ; branch if text on
      rts
10$
      jsr   KillPrompt                ; turn the prompt off
      PushB dispBufferOn              ; save the current value of dispBufferOn
      MoveB textDispBufOn,dispBufferOn; load correct value for text output.

      ;---  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            ; y printing position
      sta   r1H

      ;---  Process the character
      lda   keyData                   ; get the keypress again
      cmp   #SPACE                    ; compare 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-register 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
        ldx     bufFaultVec+1
        jsr     CallRoutine             ; call buffer overflow routine

88$
        ;---    Clean up
        lda     textOn                  ; only re-enable the prompt if text
        beq     90$                     ; is still on (might have changed!)
        jsr     PromptOn                ; turn the prompt back on

90$
        PopB    dispBufferOn            ; restore dispBufferOn
        rts                             ; exit
```

KillPrompt:

```
                                                                       .if 0
```

Function: Proper way to use **PromptOff**.

Description: Disable interrupts and clears **alphaFlag**.

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

DoReturn:

```
.if 0
```

Function: Process a carriage return.

Description: No real carriage return handler yet. Just shut text off.

```
.endif
DoReturn:
        LoadB  textOn,#FALSE
        rts
```

DoBackspace

`.if 0`

Function: Process a backspace.

Description:

`.endif`

```
DoBackspace:
      ldy    txtBufIndex               ; get ptr into current text buffer
      beq    90$                       ; 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
      lda    #NULL                     ; we just deleted and make it the
      sta    (txtBuf),y                ; null-terminator
      MoveW  r11,stringX               ; update the cursor's x-position
90$
      rts                              ; exit
```

EraseCharacter:

```
                                                                               .if 0
```

Function: Physically remove a character from the screen
Description:

```
                                                                               .endif
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
        SubBS  r3L,r1H,r2L                 ; calc top of character by subtracting
                                           ; baseline offset from y-position
                                           ; and making top-edge of rectangle
        txa                                ; add char height to top-edge
                                           ; to calc bottom-edge
        add    r2L
        sta    r2H                         ; and make bottom of rectangle
        sty    r3L                         ; set down width so we can subtract it
                                           ; from the current x-position to
        sub    r11L                        ; find the character's starting
        sta    r3L                         ; position
        ldy    r11H
        bcs    10$                         ; subtract one from high-byte if borrow
        dey
10$
        sty    r3H                         ; make left-edge of rectangle
        jsr    Rectangle                   ; erase in current pattern
        rts                                ; exit

                                                                               .if 0
```

Function: Handle Buffer Overflow.
Description: What to do if the buffer hits its maximum.

```
                                                                               .endif
BufOverflow:
        LoadB  textOn,#FALSE
        rts

                                                                               .if 0
```

Function: text fault handler.
Description: String faults come here.

```
                                                                               .endif
TextFault:
        LoadB  textOn,#FALSE               ; no real text fault handler, yet, just shut text off
        rts
```

KeyHandler:

```
                                                                                    .if 0
```

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

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

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

      ;---  lock out interrupts for a moment
      ;     so we don't get any new keypresses
      php                             ;save current interrupt disable status
      sei                             ;disable interrupts
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 old interrupt 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
```

DoNewKeys:

```
                                                                                   .if 0
 A do-nothing routine that just pretends to empty our own keyboard buffer.
                                                                                  .endif
DoNewKeys:
      ldx    #0                        ; 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
99$
      rts                              ; exit
```

KillPrompt:

```
.if 0
```

Function: Safely turn off text prompt.

Parameters: nothing.

Returns: nothing.

Alters: **alphaFlag**.

Destroys: a, x, **r3L**.

Description: Disables interrupts and then turns text prompt off.

```
                                                                .endif
KillPrompt:
        php                             ; save current interrupt disable status
        sei                             ; disable interrupts
        jsr     PromptOff               ; prompt - off
        LoadB   alphaFlag,#0            ; clear alpha flag
        plp                             ; restore old interrupt status
        rts
```

NewGetString

```
.if 0
```

Function: New front-end to **GetString** to guarantee a consistent state of **dispBufferOn** during the entire entry.

Parameters: same as **GetString**.
Returns: same as **GetString**.
Destroys: same as **GetString**.

Description: Wedges into **keyVector** before SystemStringService gets control. This routine uses StringPatch to adjust **dispBufferOn** so that it holds the value that it contained when NewGetString was first called, making every character print consistently. It otherwise acts just like **GetString**.

Note: It is very unlikely that **dispBufferOn** will be getting changed during **MainLoop** processing during **GetString**. The primary purpose of this example is to show how to hook into the **GetString** processing.

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

.psect

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

      jsr    GetString                ; Call GetString as normal

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

StringPatch:

```
.if 0
```

Function: When a key is pressed during a **GetString**, control comes here.

Description: We load up the correct value of **dispBufferOn**, link through to the correct SystemStringService, and restore **dispBufferOn** when control comes back. When the string is terminated with [Return], SystemStringService will take care of removing us.

```
                                                                                    .endif
StringPatch:

      PushB  dispBufferOn             ; Save the current value of dispBufferOn

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

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

      ;---   we will eventually get control again. Restore the old value of dispBufferOn before
      ;      going back to MainLoop
      PopB   dispBufferOn
      rts                             ; Exit
```

ShortKey:

```
.if 0
```

Function: Shortcut key handler.

Parameters: keycode in accumulator.

Description: Short cut key dispatcher. Call From **keyVector** handler.

```
                                                                        .endif
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$                      ; it's lowercase: convert to upper
      ;--- Carry will always be clear here.
      ;sec
      ;--- Subtract 1 extra and save a byte and 2 cycles by not doing the sec.
      sbc    #('a'-'A') -1            ; 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                ; 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
shortcuts:
       .byte '0'                        ; 1 undo
       .byte 'T'                        ; 2 text
       .byte 'P'                        ; 3 print
       .byte 'Q'                        ; 4 quit
       .byte 'N'                        ; 5 new document
       .byte 'G'                        ; 6 go to 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 [Doltalic                  ; 9
       .byte [DoUnderline               ; 10
       .byte [DoDelete                  ; 11
       .byte [DoCopy                    ; 12
       .byte [DoScroll                  ; 13
       .byte [DoLoad                    ; 14

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 ]Doltalic                  ; 9
       .byte ]DoUnderline               ; 10
       .byte ]DoDelete                  ; 11
       .byte ]DoCopy                    ; 12
       .byte ]DoScroll                  ; 13
       .byte ]DoLoad                    ; 14
```

8BitMultiply:

`.if 0`

Function: 8 Bit unsigned multiply.

Parameters:

r1L	multiplicand.
r1H	multiplier.

Returns: unsigned product in **r2**.

Destroys: a, x, y, **r7L**, **r8**.

Description: Multiply **r1L** by **r1H** and store the word product in **r2**.

`.endif`

```
8BitMultiply:
    MoveB     r1L,r2L                    ; r2L <- r1L copy of multiplicand
    ldx       #r2                        ; x <- multiplicand address
    ldy       #r1H                       ; y <- multiplier address
    jsr       BBMult                     ; r2 <- r2L * r1H do multiplication
    rts
```

16x8Multiply:

```
.if 0
```

Function: 16x8 Bit unsigned multiply.

Parameters:

x	**zpage** address of multiplicand.
y	**zpage** address of multiplier.

Returns: unsigned result in address pointed to by x.

x, y	unchanged.

Description: Multiply the value in **r9** by 87 and store the result back in **r9** (**r1** is destroyed).

```
                                                                        .endif
16x8Multiply:
    ldx       #r9                       ; point to multiplicand in r9
    LoadB     r1L,#87                   ; r1L <- 87 (multiplier)
    ldy       #r1L                      ; point to multiplier in r1L
    jsr       Bmult                     ; r9 <- r9 * r1L
    rts
```

ConvToUnits:

`.if 0`

Function: 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.

Parameters: **r0** number to convert (in pixels).

Returns:

r0 inches / centimeters.
r1L tenths of an inch / millimeters.

Destroys: a, x, y, **r0-r1**, **r8-r9**.

Description: Assembler time decision on whether inches or centimeters is to be used.

`.endif`

```
.if AMERICAN
    INCHES = TRUE
.else
    INCHES = FALSE                     ; metric
.endif

ConvToUnits:                           ; first, convert r0 to length in 1/20 of
                                       ; standard units

.if INCHES
                                       ; for inches, need to multiply by
                                       ;      20          1
                                       ; ------------- = ---
                                       ; 80 dots/inch     4
    ldx       #r0                      ; which amounts to a divide by four
    ldy       #2
    jsr       DShiftRight
.else
                                       ; For Centimeters, need to multiply by
                                       ;      20          1
                                       ; ------------- = ---
                                       ; 31.5 dots/cm     63
                                       ;
    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 l/20ths
    IncW      r0                       ; add in one more l/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 - l/20ths
    asl       r1L                      ; and convert to l/10ths (rounded)
    rts                                ; exit
```

DdecvsDecW:

```
.if 0
```

Function: Size in bytes vs speed in cycles of **Ddec** and **DecW**.

Ddec Represents a maximum of 7 byte savings over **DecW** every time it is used in your code. If Not needing a zero result after **DecW** then only a 3 byte savings.

DecW Takes roughly ½ the time to execute. In an inner loop executed 1 Million times, **DecW** will save roughly 20 seconds off the time vs **Ddec**.

```
.endif
zCounter=$70
.macro DecW   dest
    lda       dest
    bne       dolow
    dec       dest+1
dolow:
    dec       dest
.endm

Ddec code block.
Machine Code    Opcode                 Bytes     Cycles
------------    ---------------        ------    --------
A2 70           ldx #zCounter          2         2
20 0E C2        jsr Ddec               3         6
                (Kernal Routine)       0         27 - 32
-------------------------------------------------------------------------------------------------
                Total                  5         35 - 40

DecW macro code block.
Machine Code    Opcode                 Bytes     Cycles
------------    ---------------        ------    --------
A9 70           lda zCounter           2         3
D0 02           bne 10$                2         2 or 3 or 4
C6 71           dec zCounter+1         2         5
             10$
C6 70           dec zCounter           2         5
-------------------------------------------------------------------------------------------------
                Total                  8         11  Worst Case 15
                if branch crosses page 12

    ;-- When using DecW on a counter, Add check for word=0 after the DecW macro
A9 70           lda zCounter           2         2
05 70           ora zCounter+1         2         3
-------------------------------------------------------------------------------------------------
                Total                  12        16 - 20

Kernal Ddec     ;Actual Kernal Code for Ddec
Machine Code    Opcode                 Bytes     Cycles
------------    ---------------        ------    --------
B5 00           lda zpage,X                      4
D0 02           bne 10$                (1/256ish chance 2) or 3 or Worst case:4
D6 01           dec zpage+1,X                    6
             10$
D6 01           dec zpage,X                      6
B5 00           lda zpage,X                      4
D6 01           ora zpage+1,X                    4
60              rts                              6
                                       =====================
                    Total           Best Case:  27                Worst Case:32
             if branch crosses Page             28                (1/256 chance)
```

DecCounter:

`.if 0`

Description: Example use for **Ddec**.

Parameters: nothing.

Alters: zCounter.

Destroys: a, anything destroyed in DoSomething.

`.endif`

```
.ramsect      APP_ZPL                         ; $70
    zCounter:
              .block  1

COUNT         = $FFF0

    DecCounter:
              LoadW   zCounter,#COUNT
    10$
              jsr     DoSomething
              ldx     #zCounter
              jsr     Ddec
              bne     10$
              rts
```

Divide By Zero:

```
.if 0
```

Function:	**NewDdiv**	Wrapper for **Ddiv** with divide-by-zero error checking.
	NewDSdiv	Wrapper for **DSdiv** with divide-by-zero error checking.

Parameters:	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**.

Example:

```
;---   Example use of the validated Ddiv wrapper.
ldx    #r0          ; point x to dividend
ldy    #r1          ; point y to divisor
jsr    NewDdiv      ; call our validated Ddiv routine
bmi    99$          ; branch on divide by zero error
...
```

```
                                                                                   .endif
DIVIDE_BY_ZERO        = $FF
NO_ERROR              = $00

      NewDdiv:
            lda    zpage,y             ; get low-byte of divisor
            ora    zpage+1,y           ; and high-byte of divisor
            beq    99$                 ; if both are zero, raise error
            jsr    Ddiv                ; divide
            lda    #NO_ERROR           ; and return no error
clda  99$,         #DIVIDE_BY_ZERO
            rts

      NewDSdiv:
            lda    zpage,y             ; get low-byte of divisor
            ora    zpage+1,y           ; and high-byte of divisor
            beq    99$                 ; if both are zero, raise error
            jsr    DSdiv               ; divide
            lda    #NO_ERROR           ; and return no error
clda  99$,         #DIVIDE_BY_ZERO
            rts
```

DSmult:

`.if 0`

Function: DSMult double-precision signed multiply.

Parameters:
x **zpage** address of multiplicand.
y **zpage** address of multiplier.

Returns: signed product 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**

`.endif`

```
DSmult:
      lda   zpage+1,x                ; get sign of multiplicand (high-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
```

Kernal_CRC:

```
.if 0
```

Function: This is the actual Kernal Code for **CRC**.

Parameters: **r0** pointer to start of data.
r1 # of bytes to check.

Returns: **r2** CRC Checksum.

Destroys: a, x, y, **r0**, **r1**, **r3L**.

```
.endif
```

```
       Kernal_CRC:
             ldy   #$FF
             sty   r2L
             sty   r2H
             iny
       10$
             lda   #$80
             sta   r3L
       20$
             asl   r2L
             rol   r2H
             lda   (r0),y
             and   r3L
             bcc   30$
             eor   r3L
       30$
             beq   40$
             tmbf  5,r2L
             tmbf  4,r2H
       40$
             lsr   r3L
             bcc   20$
             iny
             bne   50$
             inc   r0H
       50$
             ldx   #r1
             jsr   Ddec                ; Ddec returns with z flag following the value of r1
;            bwne  r1,10$              ; No need to recheck for zero.
             bne   10$
             rts
```

NewSDSdiv:

```
.if 0
```

Function: Wrapper for **DSdiv**. Call as you would call **DSdiv**.

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).

Returns:
x, y unchanged.
r8 the fractional remainder (word) with a matching sign of the dividend.
word pointed to by OPERAND2 equals its absolute value.
word pointed to by OPERAND1 contains the word result.

Destroys: a, **r9**.

Decscription: 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 fix this problem

Example:

```
;---   Example use of the validated Ddiv wrapper.
ldx    #r0          ; point x to dividend
ldy    #r1          ; point y to divisor
jsr    NewSDSdiv    ; call our validated Ddiv routine
bmi    99$          ; branch on divide by zero error
...
```

```
                                                                   .endif
NewSDSdiv:
    lda       zpage+1,x          ; save sign of dividend
    php
    jsr       DSdiv              ; divide as normal
    plp                          ; then get sign of dividend back
    bpl       90$                ; ignore if positive
    PushX                        ; save x-register
    ldx       #r8                ; else, negate remainder
    jsr       Dnegate
    PopX                         ; restore x-register
90$
    rts
```

CopyBuffer:

.if 0

Function: Examples for **CopyFString and CopyString**.

.endif

```
srcBuff:
       .byte "Any Values can be in the buffer",NULL,CR
       .byte $0C, ”NULLS are just zeros for CopyFString”,CR

LENBUFF = (*-srcBuff)

.ramsect
       destBuff:
              .block LENSTRING

.psect

CopyBuffer:
       LoadW  r5,#srcBuff                ; point to start of source buffer
       LoadW  r1L,#destBuff              ; point to start of destination buffer
       ldx    #r5                        ; x <- source register address
       ldy    #r1L                       ; y <- destination register address
       lda    #LENBUFF                   ; a <- length of buffer
       jsr    CopyFString                ; destBuff <- srcBuff (copy)
       rts

srcStr:
       .byte "Any values but null can be in the string",NULL
LENSTRING = (*-srcStr)

.ramsect
       DestBuff:
              .block LENSTRING

.psect

CopyStr:
       LoadW  r0,#srcStr                 ; point to start of source String
       LoadW  r1,#destBuff               ; point to start of destination buffer
       ldx    #r0                        ; x <- source register address
       ldy    #r1                        ; y <- destination register address
       jsr    CopyString                 ; destBuff <- srcStr (copy)
       rts
```

Find:

```
.if 0
```

Function: Examples for Find.

```
.endif
```

```
REC_SIZE = 5                        ; size of each record
.ramsect
       Data:
              .block 1024           ; table of zip code locations

.psect
       Key:
              .byte  "94704"        ; zip code to find

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 <- destination string – list
       lda    #REC_SIZE             ;      a <- length of each record
       jsr    CmpFString            ;      compare key with current record
       beq    20$                   ;      if they match, branch to handler
       AddVW  #REC_SIZE,r1          ;      otherwise point to the next record
       DecW   r2                    ;      r2 — (decrement counter)
       bne    10$                   ; While (r2 > 0)
       ;---
       jmp    NotMatched            ; jmp to no match handler
20$
       jmp    Matched               ; jmp to match handler
```

Find2:

```
.if 0
```

Function: Another example for find.

```
.endif
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  <- destination string - list
        jsr    CmpString           ;
        beq    20$
        jmp    NotMatched          ; jmp to no match handler
20$
        jmp    Matched             ; jmp to match handler


original:
        .byte  "Mark Charles Heartless",NULL

Copy:
        .byte "Mark Charlie Heartless",NULL
```

InitBuffers:

`.if 0`

Function: Clear RAM examples.

`.endif`

```
;--- initialize buffers and variables to zero

InitBuffers:
    LoadW     r0,#varStart              ; clear variable space
    LoadW     r1,#(varEnd-varStart)
    jsr       ClearRam
    LoadW     r0,#heapStart             ; clear heap
    LoadW     r1,#(heapEnd-heapStart)
    jmp       ClearRam


;--- Alternate version. Using more space efficient i_FillRam

InitBuffers:
    jsr       i_FillRam                 ; clear variable space
    .word     varStart
    .word     varEnd-varStart
    .byte     $AA                       ; with any value you choose

    jsr       i_FillRam                 ; clear heap
    .word     heapStart
    .word     heapEnd-heapStart
    .byte     $00                       ; heap set to zero’s
    rts
```

ArrowUp:

`.if 0`

Function: Put up a new mouse picture.

`.endif`

```
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:
    HILO     %1111111110000000         ; mask
    HILO     %1111111001111110
    HILO     %0001100111111001
    HILO     %0110011111100111
    HILO     %0111111110011111
    HILO     %0111111110011111
    HILO     %0111111111101111
    HILO     %0000000000001111

    HILO     %0000000000000000         ; image
    HILO     %0000000001111110
    HILO     %0000000111111000
    HILO     %0110011111100000
    HILO     %0111111110000000
    HILO     %0111111110000000
    HILO     %0111111111100000
    HILO     %0000000000000000
```

MouseInit:

`.if 0`

Purpose: Initialize the mouse and start it at screen center.

Parameters: nothing.

Returns: nothing.

Alters: **alphaFlag**.

Destroys: a, x, y, **r0-r15**.

Description: Disable interrupts and then setup mouse at screen center.

`.endif`

```
MouseInit:
        LoadW  r11,#(SC_PIX_WIDTH/2)     ; screen center
        ldy    #(SC_PIX_HEIGHT/2)
        sec                              ; set carry to move mouse
        php                              ; save current interrupt disable status
        sei                              ; disable interrupts
        jsr    StartMouseMode
        plp                              ; restore old interrupt status
        rts
```

NewIsMseInRegion:

`.if 0`

Function: Replacement for **IsMseInRegion**.

Description: Handles the disabling of interrupts so return status registers are not effected by plp.

`.endif`

```
NewIsMseInRegion:
       ;---   disable interrupts around coordinate checks
       ;      so it doesn't change while we're looking
       php                             ; save current interrupt disable status
       sei                             ; disable interrupts
       CmpB   mouseYPos,r2L            ; compare mouse y-position to top-edge
       blt    10$                      ; branch if outside
       cmp    r2H                      ; compare to bottom-edge
       bgt    10$                      ; branch if outside
       CmpW   mouseXPos,r3             ; compare mouseX with left-edge
       blt    10$                      ; branch if outside
       CmpW   mouseXPos,r4             ; compare mouseX with right-edge
       bgt    10$                      ; branch if outside
       plp                             ; restore old interrupt status (before setting st reg)
       lda    #[TRUE                   ; return inside region status
       rts                             ; exit
10$    plp                             ; restore old interrupt status (before setting st reg)
       lda    #FALSE                   ; return outside region status
       rts                             ; exit

NewIsMseInRegion:                      ;-- Alternative version compatible with 128 GEOS.
       php
       sei
       jsr    IsMseInRegion            ;IsMseInRegion handles DOUBLE_W coordinates.
       bpl    10$
       plp
       lda    #[TRUE
       rts
10$    plp
       lda    #FALSE
       rts

NewIsMseInRegion:                      ; Much smaller version but the y-register is destroyed.
       php
       sei
       jsr    IsMseInRegion
       plp
       tay                             ; transfer result to y-register to reset status flags.
;      ora    #0                       ; or use 'ora' instead of tay and maintain the y-register
       rts                             ; at the cost of one more byte used.

SampleUse:
       LoadW  r3,#windowX1             ; get coordinates of window's rectangle
       LoadW  r2L,#windowY1
       LoadW  r4,#windowX2
       LoadW  r2H,#windowY2
       jsr    NewIsMseInRegion         ; check for mouse inside region
       bpl    MouseOutsideWindow       ; branch if outside window area
       ...
```

IsMseInMargins:

`.if 0`

Function: Check if mouse is within the left and right text margin

`.endif`

```
      IsMseInMargins:
            ;---   disable interrupts around mouseXPos access
            php                     ; save current interrupt disable status
            sei                     ; disable interrupts
            MoveW  mouseXPos,r0     ; and copy current position to a working location
            plp                     ; restore old interrupt status

            CmpW   r0,leftMargin    ; check left-margin
            bcc    99$              ; fault out if less than left
      10$
            CmpW   r0,rightMargin   ; check right-margin
            ble    20$              ; branch if inside right
            bcs    99$              ; fault out
      20$
            lda    #[TRUE           ; no fault (inside text margins)
clda  99$,         #FALSE           ; fault outside of margins
            rts
```

OPVector:

`.if 0`

Function: Sample **otherPressVec** handler.

Description: gets called on each press (and release) of input button.
demonstrates double click detection.

```
                                                                        .endif
OPVector:
      ;---  Ignore releases on entry
      lda   mouseData                  ; check state of the mouse button
      bpl   05$                        ; branch to handle presses
      rts                              ; but return immediately to ignore releases

05$   ;--- User pressed mouse once, start double-click counter going
      LoadB dblClickCount,#CLICK_COUNT; start delay
10$
      ;--- Loop until double-click counter times-out or button is released
      lda   dblClickCount              ; check double-click timer
      beq   30$                        ; if timed-out, no double-click
      lda   mouseData                  ; else, check for second press
      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)
      jmp   DoDoubleClick              ; do double-click stuff

      ;--- Single-click detected (no double-click)
30$   jmp   DoSingleClick              ; do single-click stuff
```

Alternative method:

This method does not use UI time waiting in a loop too see of the user is going to click again.

```
.ramsect
      rDblCnt:    .block 1
.psect

OPVector:
      bbrf  7,mouseData,10$            ; Do work on button press
      LoadB dblClickCount,#CLICK_COUNT; On release set double click count to 30
      rts                              ; and return immediately to otherwise ignore releases.

10$   MoveB dblClickCount,rDblCnt      ; save current count
      ;--- Do work to determine what is being effected.
      ...
      ;--- In area that responds to a dbl click, check the saved count.
50$   bbne  rDblCnt,70$                ; if rDblCnt >= 0 then we have a double click
      ;--- do single click work here

70$   ;--- do double click work here
90$   rts
```

ResetMouse:

```
.if 0
```

Function: 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.

```
.endif
ResetMouse:
      ;---   (Following line changed to save bytes)
      LdNull mouseLeft                ; reset mouse left to left screen edge
      sta    mouseTop                 ; and mouse top to top screen edge

.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
      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           ; exit
      rts
```

ClipChar:

```
                                                                        .if 0
```

Function: Draw a character, clipping it EXACTLY to **leftMargin**, **rightMargin**, **windowTop** and **windowBottom**.

Parameters:
a character to print.
r1L x-position.
r1H y-position.

Returns:
r11 x-position for next char.
r1H y-position for next char.

Destroys: a, x, y, **r2-r10L**.

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

Note: **SmallPutChar** already does character clipping at the margins. The **ClipChar** example should not be used simply for the purpose of printing a partial character at the margin. The following code will perform the same function:

```
strFaultHandler:
      ldy    #0
      lda    (r0),y              ; get the character that caused the fault
      jsr    SmallPutChar        ; use SmallPutChar to draw clipped character
      LoadW  r0,#null-1          ; set r0 to point to a null to end
      rts                        ; PutString processing.

null:  byte NULL
```

```
                                                                       .endif

.ramsect
      savedWidths:
            .block 4                  ; values from index table stored here

.psect

ClipChar:
      sta    r1L                      ; store character
      ldx    currentMode              ; get width of character
      jsr    GetRealSize              ;
      dey                             ; use width - 1 to calc last position
      AddYWS r11,r2                   ; r2 = last pixel that char covers
      CmpW   r2,leftMargin            ; check for char entirely off window
      blt    10$                      ; if so then exit
      CmpW   rightMargin,r11
      bge    20$
10$   AddVWS #1,r2,r11                ; r11 = one pixel beyond where char would have gone
      rts                             ; exit

20$   SubBS  #32,r1L,r3L              ; push old width table values
      asl    a                        ; get card #
      tay
      ldx    #0
```

```
30$
       lda    (curIndexTable),y         ; store this char's index values
       sta    savedWidths,x
       iny
       inx
       cpx    #4
       bne    30$                       ; loop to copy values

       CmpW   leftMargin,r11
       blt    40$
       lda    r3L
       asl    a
       tay
       lda    leftMargin                ; check for clipping on left
       sub    r11L
       add    (curIndexTable),y
       sta    (curIndexTable),y
       iny
       lda    #0
       adc    (curIndexTable),y
       sta    (curIndexTable),y
       MoveW  leftMargin,r11
40$
       CmpW   r2,rightMargin
       blt    50$                       ; check for clipping on right
       SubBS  rightMargin,r2L,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$
       PushB  r1L                       ; save it for later
       jsr    SmallPutChar              ; draw the character!!
       pla
       sub    #' '                      ; $20
       asl    a                         ; recover old widths
       tay
       ldx    #0
60$
       lda    savedWidths,x
       sta    (curIndexTable),y
       iny
       inx
       cpx    #4
       bne    60$
       rts
```

Print:

```
                                                                                                .if 0
```

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

```
                                                                                               .endif
```

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

Print:
      LoadB   dispBufferOn,#(ST_WR_FORE | ST_WR_BACK) ; both buffers!
      LoadW   r11,#STR_X                ; string x-position
      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
```

PutStrFault:

`.if 0`

Function: Modify default GEOS string fault handling with **PutString**.

Note: Activate this handler with:
`LoadW  StringFaultVec,#PutStrFault`

Description: String fault routine to immediately terminate string printing when any fault (left or right-margin) is generated by setting **r0** to point to the end of the string.

`.endif`

```
PutStrFault:
      ;--- go through the string looking for the null
      ldy   #0                       ; load index to character pointed to by (r0)
10$
      lda   (r0),y                   ; get character
      beq   90$                      ; if null then exit
      IncW  r0                       ; bump pointer to check next character
      bne   10$                      ; loop until we find null
90$
      ;--- return to PutString pointing at a null
      rts
```

SmartPutString:

```
.if 0
```

Description: 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.

Parameters: Same as **PutString**.

Returns: **r15** points to the offending character in the string that caused the fault. (NULL if no fault).

Destroys:

```
                                                                .endif
SmartPutString:
       PushW  StringFaultVec            ; saving Fault Vector for restore on exit
       LoadW  StringFaultVec,#FaultFix  ; install new fault routine
       LdNull r15                       ; clear r15 to $0000
       jsr    PutString                 ; call PutString with our string fault routine in place
90$
       PopW   StringFaultVec            ; restore the old string fault routine
       rts                              ; return
                                        ; caller can now check if r15 has a value
```

An alternate implementation.

During application init, set the **StringFaultVec** to the FaultFix handler. Then leave it for the life of the application. GEOS will reset the vector on application close.

```
       LoadW  StringFaultVec,#FaultFix  ; set it and forget it
```

You can now use **PutString** or **i_PutString** as you always have with the new ability to check for margin faults after the call to either one.

Example:

```
       ...
       jsr    PutString
       CmpWI  r15,#0
       bne    HandleFault
       ...
```

If you impose a restriction that strings cannot be in zero page then you can check this way.

```
       ...
       jsr    PutString
       lda    r15H
       bne    HandleFault
       ...
```

FaultFix:

```
.if 0
```

Function: New **StringFaultVec** Handler.

Parameters: Called by **PutString** when margin fault occurs. Normal **PutString** registers will be set.

Returns: **r15** points to the offending character in the string that caused the fault. (null if no fault).

Destroys: same as **PutString**.

Description: Fixes the handling of right margin fault by:

1. All attempts to continue printing, stop immediately.
2. Pointer to the offending character position that caused the fault is returned in **r15**.

Note: left-margin fault behavior is not changed.

Note: GEOS 128 x-coordinates are already in a normalized state at time of handler call.

```
.endif

fakeNull:
      .byte  NULL                        ; null for FaultFix

FaultFix:
      CmpW   rightMargin,r11             ; check x-coordinate with right-edge
      ble    90$                         ; exit if right not exceeded;
                                         ; the character was outside the left-edge

      MoveW  r0,r15                      ; save the pointer to the offending character in r15
      LoadW  r0,#(fakeNull-1)            ; -1 since PutString will check the "next" char on return
90$
      rts
```

BeepThrice:

```
.if 0
```

Function: Beep three times.

Description: Runs off the **MainLoop** by using **Sleep**.

```
.endif
.if TARGET_NTSC
    FRAME_RATE=60
.else
    FRAME_RATE=50
.endif

BELL_INTERVAL = (FRAME_RATE/10)        ; approximately. 1/10 second.

BeepThrice:
      jsr     Bell                     ; sound the bell
      LoadW   r0,#BELL_INTERVAL        ;
      jsr     Sleep                    ; pause a bit
      jsr     Bell                     ; sound the bell again
      LoadW   r0,#BELL_INTERVAL
      jsr     Sleep                    ; pause a bit
      jmp     Bell                     ; sound the bell again and let bell rts
```

Note[3]: see **GetFPS** for detecting frame rate for portability between hardware.

FatalError:

`.if 0`

Function: use **Panic** to send a fatal error message to the user.

Parameters: **r0**

`.endif`

```
.ramsect
    GEOS_save:
          .block BYTESTOSAVE           ; save area for GEOS restart block

.psect

    FatalError:
          IncW  r0                     ; add 2 to error number
          IncW  r0                     ; to compensate for Panic
.if C64
          PushW r0                     ; push error number onto stack
.else
   ;---   128, expects all kinds of internal
   ;      machine-state information (10 bytes total) on the stack.
   ;      it ignores all but the bottom-most word.
          ldx   #5                     ; 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



   ;---   Alternate Version with live detection of 64/128
   ;      and a more efficient setting of the stack pointer.

   FatalError:
          IncW  r0                     ; add 2 to error number
          IncW  r0                     ; to compensate for Panic
          bbrf  7,c128Flag,10$         ; if C64. just push once.
          ;---  128, expects all kinds of internal
          ;     machine-state information (10 bytes total) on the
          ;     stack. It ignores all but the bottom-most word.
          tsx                          ; set stack pointer down 8 bytes to prepare for r0
          txa                          ; push for the last word
          sub   #8
          txs                          ; save the new stack pointer
   10$                                 ; now put final word onto stack
          PushW r0                     ; push error number onto stack
          jmp   Panic                  ; go put up the Panic dialog box
```

HandleCommand:

`.if 0`

Function: Given a command number this routine handles dispatching control to the appropriate routine.

Parameters: y command number.

Returns: depends on command.

Destroys: depends on command.

`.endif`

```
UNIMPLEMENTED = $0000

HandleCommand:
    cpy       #TOT_CMDS                 ; check command # against last cmd#
    bcs       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 high/low-bytes of the routine
;--- associated with each command number. If a command is not yet implemented
;--- use the UNIMPLEMENTED constant

CMDtabH:                                ; high-bytes
    .byte     ]UNIMPLEMENTED            ; high-byte of command 0
    .byte     ]Cmdl                     ; high-byte of command 1
    .byte     ]Cmd2                     ; etc...
    .byte     ]Cmd3
    .byte     ]Cmd4
CMDtabL:                                ; low-bytes
    .byte     [UNIMPLEMENTED            ; low-byte of command 0
    .byte     [Cmdl                     ; low-byte of command 1
    .byte     [Cmd2                     ; etc...
    .byte     [Cmd3
    .byte     [Cmd4

TOT_CMDS = (CMDtabL-CMDtabH)            ; total Number of commands

Cmd1:
    ;--- Perform some action here.
    rts
Cmd2:
    ;--- Perform some action here.
    rts
Cmd3:
    ;--- Perform some action here.
    rts
Cmd4:
    ;--- Perform some action here.
    rts
```

LoadBASIC:

```
.if 0
```

Function: 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.

Parameters: nothing.

```
.endif
basicProg:
    .byte     "GodZilla",NULL

runCommand:
    .byte     "RUN",NULL

LoadBASIC:
    LoadW     r6,#basicProg            ; find Basic Program to run
    jsr       FindFile                 ; r5 will now point to programs DIR entry
    txa
    bne       99$                      ; if FILE_NOT_FOUND or other disk errors exit
    LoadW     r0,#runCommand           ; point at command string
    LoadW     r7,#$801                 ; assume standard address
    jmp       ToBasic
99$
    sec
    rts
```

RoadTrip:

```
.if 0
```

Function: Demonstrate leaving GEOS to use all of the resources of the machine and returning again via rebooting by either REU or disk. Note: 128 Code for reboot must `reside below $4000`.

```
.endif
BYTESTOSAVE     = 128                  ; # of bytes to save at BootGEOS.
RBOOT_BIT       = 5                    ; bit in sysFlgCopy to check
CIO_IN          = $7E
config          = $FF00

.ramsect
    GEOS_save:
          .block BYTESTOSAVE           ; save area for GEOS restart block

.psect

RoadTrip:
    jsr   OnEntry                      ; save Kernal Boot strap
    jsr   HaveAFunTrip                 ; do anything... use all of Kernal RAM
                                       ; just no GEOS Kernal calls while you are gone
    jmp   OnExit                       ; reboot the Kernal


OnEntry:
    ldx   #BYTESTOSAVE-1               ; save bytes GEOS needs so we can use area
                                       ; STARTLOOP
10$ MoveB "BootGEOS,x","GEOS_save,x"   ; copy a byte
    dex                                ;    count = count -1
    bpl   10$                          ;    if (count > 0), then loop
    rts                                ; ENDLOOP


OnExit:
    bbsf  RBOOT_BIT,sysFlgCopy, 10$    ; if rboot flag is not set
    jsr   AskForBootDisk               ;    get user to insert boot disk
10$ CmpB  version,#$13                 ; get version of GEOS
    bcc   64$                          ; if version < 1.3, then branch
    bbrf  7,c128Flag,64$               ; else, test for GEOS 128 and branch if GEOS64
;--- 128
    rmbf  0,config                     ; Map in I/O in current bank
    setbit mmurcr,#%00110000,#%01000111; Common ram on for bottom 16K / VIC in bank 1
    LoadB config,CIO_IN                ; load 128 memory mapping, activate bank 1 memory
    bne   20$                          ; (always branch)

64$ LoadB CPU_DATA,#KRNL_BAS_IO_IN     ; load 64 memory mapping
20$ ldx   #BYTESTOSAVE-1               ; restore bytes GEOS needs to restart
                                       ; STARTLOOP
30$ MoveB "GEOS_save,x","BootGEOS,x"   ; copy a byte
    dex                                ;    count = count -1
    bpl   30$                          ;    if (count > 0), then loop
                                       ; ENDLOOP
    bbsf  RBOOT_BIT,sysFlgCopy, 90$    ; if rboot flag is set, branch to rboot
    jsr   AskForBootDisk               ; else, get user to insert boot disk
90$ jmp   BootGEOS
```

6510 data register

C64

CPU_DDR = $00

Data Direction Register.	
Power on default	$2F
GEOS default	$2F

Bit	Description
b7:	unused
b0-b6:	Sets Data Direction of **CPU_DATA** port. 0 = Bit is read only 1 = Bit is write only

CPU_DATA = $01

Machine power on default	KRNL_BAS_IO_IN
GEOS default	RAM_64K
GEOS during serial I/O	IO_IN

RAM_64K	= $30	; %11 0000	64K RAM
KRNL_CH_BAS_IN	= $33	; %11 0011	Kernal + basic + Char ROM
IO_IN	= $35	; %11 0101	60K RAM, 4K I/O space in
KRNL_IO_IN	= $36	; %11 0110	Kernal + I/O
KRNL_BAS_IO_IN	= $37	; %11 0111	Kernal + basic + I/O

FFFF	RAM_64K	KRNL_CH_BAS_IN	IO_IN	KRNL_IO_IN	KRNL_BAS_IO_IN
E000	8K RAM	8k KERNAL ROM	8K RAM	8k KERNAL ROM	8k KERNAL ROM
D000	4K RAM	CHAR ROM	I/O	I/O	I/O
C000	4K RAM	4K RAM	4K RAM	4K RAM	4K RAM
A000	8K RAM	8K BASIC	8K RAM	8K RAM	8K BASIC
0100	24K RAM	24K RAM	24K RAM	24K RAM	24K RAM
	Zero Page	Zero Page	Zero Page	Zero Page	Zero Page

> **Note**: In GEOS 128, I/O is always mapped in. **CPU_DATA** does not control RAM/ROM on the 128. It is safe to use **CPU_DATA** in the same way as on the C64 before using I/O, so no code changes around it are neccessary. See "Mapping the Commodore 128" for more information on **CPU_DATA**.

Keyboard (C64, C128)

cia1pra (DC00)	**cia1prb** (DC01)							
	b7 %01111111	b6 %10111111	b5 %11011111	b4 %11101111	b3 %11110111	b2 %11111011	b1 %11111101	b0 %11111110
b0 %11111110	KEY_UP KEY_DOWN	KEY_F6 KEY_F5	KEY_F4 KEY_F3	KEY_F2 KEY_F1	KEY_F8 KEY_F7	KEY_LEFT KEY_RIGHT	KEY_ENTER	KEY_INSERT KEY_DELETE
b1 %11111101	Left SHIFT (LOCK)	E	S	Z	$ 4	A	W	# 3
b2 %11111011	X	T	F	C	& 6	D	R	% 5
b3 %11110111	V	U	H	B	(8	G	Y	' 7
b4 %11101111 [CONTROL]	N	O	K	M	0	J	I [TAB]	) 9
b5 %11011111 C=	< ,	@ `	[: {	> .	- _	L	P	+
b6 %10111111 C=	? / \	^ (UpArrow) |	=	Right SHIFT	KEY_CLEAR KEY_HOME	] ; }	* ~	KEY_BPS (£)
b7 %01111111	KEY_RUN KEY_STOP	Q	C=	SPACE	" 2	CTRL	KEY_LARROW	! 1

Sample code to check for C= Commodore key pressed:

```
.if 0
```

Parameters: nothing.

Returns: Z=0 beq to key is pressed.
Z=1 bne to key not pressed.

Destroys: a, x.

```
.endif
cia1pra     = $DC00
cia1prb     = $DC01

      IsCKeyPressed:
            php                       ; save processor status
            sei                       ; disable interrupts
            ldx    CPU_DATA           ; save current memory map
            LoadB  CPU_DATA,#IO_IN    ; bring I/O space into memory
            LoadB  cia1pra,#%01111111 ; scan for row 7
            lda    cia1prb            ; get row 7
            stx    CPU_DATA           ; restore memory map
            plp                       ; restore processor/interrupt status
            and    #%00100000         ; Mask out bit 5, if bit 5 is reset (0)
                                      ;      then the C= key was pressed
            rts                       ; exit
```

128 Keyboard - additional Keys

keyreg (D02F)	**cia1prb** (DC01)							
	b7 %01111111	b6 %10111111	b5 %11011111	b4 %11101111	b3 %11110111	b2 %11111011	b1 %11111101	b0 %11111110
b0 %11111110	1	7	4	2	KEY_TAB	5	8	KEY_HELP
b1 %11111101	3	9	6	KEY_ENTER	KEY_LF	-	+	KEY_ESC
b2 %11111011	KEY_NOSCRL	KEY_RIGHT	KEY_LEFT	KEY_DOWN	KEY_UP	.	0	KEY_ALT

The 128 can use the same logic block as C64 GEOS for reading the base keyboard. If the application is designed for 128 only then the saving/setting of **CPU_DATA** can be removed from the code block.

Sample 128 only code to check for TAB key pressed from the 128's additional keys:

```
.if 0
```

Parameters: nothing.
Returns: N=1 bmi to key is pressed.
N=0 bpl to key not pressed.
Destroys: a, x.

```
.endif
```

```
keyreg      = $DC2F
cia1pra     = $DC00
cia1prb     = $DC01

      IsTabKeyPressed:
            php                      ; save processor status
            sei                      ; disable interrupts
            LoadB cia1pra,#%11111111 ; don't scan for any of the standard keyboard rows
            LoadB keyreg,#%11111110  ; scan for row 0 in number pad area
            bbrf  4,cia1prb,10$      ; if bit 5 is reset (0) then the tab key was pressed
            ldx   #FALSE             ; tab key was not pressed
cldx  10$,        #[TRUE             ; tab key was pressed
            plp                      ; restore processor/interrupt status
            txa                      ; set N flag
            rts                      ; exit
```

MMU: Configuration Register D500, FF00

FF00 is a Mirror of D500. FF00 is always visible to the CPU.

	Configuration Register	**config**=FF00 **mmucr**=D500
Bits	**Description**	**Constant**
7-6	Bank select	
	00 Bank 0	MBANK0 =%00000000
	01 Bank 1†	MBANK1 =%01000000
	10 Bank 2	MBANK2 =%10000000
	11 Bank 3	MBANK3 =%11000000
5-4	C000-CFFF, E000-EFFF	Zone 4
	00 Kernal ROM	MHKERNAL =%000000
	01 Internal Function ROM	MHIROM =%010000
	10 External Function ROM	MHEROM =%100000
	11 RAM†	MHERAM =%110000
3-2	8000-BFFF	Zone 3
	00 Basic ROM	MUBASIC =%0000
	01 Internal Function ROM	MUIROM =%0100
	10 External Function ROM	MUEROM =%1000
	11 RAM†	MURAM =%1100
1	4000-7FFF	Zone 2
	0 BASIC ROM low	MBASIC =%00
	1 RAM†	MEXTROM =%10
0	D000-DFFF	Zone 5
	0 I/O†	MIO =%0
	1 1 RAM or Character ROM	MCROM =%1

†GEOS defaults

	RAM Configuration Register	**mmurcr**=D506
Bits	**Description**	**Constant**
7-6	Bank select for VIC video bank	
	00 Bank 0	MBANK0 =%00000000
	01 Bank 1†	MBANK1 =%01000000
	10 Bank 2	MBANK2 =%10000000
	11 Bank 3	MBANK3 =%11000000
5-4	Not used	
3-2	Common Ram Location	
	00 Disabled†	CRL_OFF =%0000
	01 Bottom	CRL_BOT =%0100
	10 Top	CRL_TOP =%1000
	11 Both	CRL_BOTH =%1100
0-1	Size of Common Ram	
	00 1k †	CRS_1K =%00
	01 4k	CRS_4K =%01
	10 8k	CRS_8K =%10
	11 16k	CRS_16K =%11

†GEOS defaults

Bank Configurations

config (D500/FF00)

GEOS configurations

```
CIO_IN                 = %01111110     $7E    ; 60K RAM, 4K I/O   GEOS default
CRAM_64K               = %01111111     $7F    ; 64K RAM
CKRNL_BAS_IO_IN        = %01000000     $40    ; Kernal, I/O, basic
CKRNL_IO_IN            = %01001110     $4E    ; Kernal, I/O
CIO_INB0               = %00111110     $3E    ; BANK 0, 60K RAM, 4K I/O BACKRAM DEBUGGER default
```

Commodore standard configurations

```
BANK_0    = MBANK0|MHERAM|MURAM|MEXTROM|MCROM ; No ROMs, RAM 0

BANK_0    = %00111111     ; No ROMs, RAM 0
BANK_1    = %01111111     ; No ROMs, RAM 1
BANK_2    = %10111111     ; No ROMs, RAM 2    ; requires 512k expanded 128
                                              ; otherwise same as bank 0
BANK_3    = %11111111     ; No ROMs, RAM 3    ; requires 512k expanded 128
                                              ; otherwise same as bank 1

BANK_4    = MBANK0|MHIROM|MUIROM|MEXTROM|MIO
BANK_5    = MBANK1|MHIROM|MUIROM|MEXTROM|MIO
BANK_6    = MBANK2|MHIROM|MUIROM|MEXTROM|MIO
BANK_7    = MBANK3|MHIROM|MUIROM|MEXTROM|MIO

BANK_8    = MBANK0|MHEROM|MUEROM|MEXTROM|MIO
BANK_9    = MBANK1|MHEROM|MUEROM|MEXTROM|MIO
BANK_10   = MBANK2|MHEROM|MUEROM|MEXTROM|MIO
BANK_11   = MBANK3|MHEROM|MUEROM|MEXTROM|MIO

BANK_12   = %00000110                         ; int function ROM, Kernal and I/O, RAM 0
BANK_13   = %00001010                         ;
BANK_14   = %00000001                         ; all ROMs, char ROM RAM 0
BANK_15   = %00000000                         ; all ROMs, RAM 0 power on default

BANK_99   = $00001110                         ; I/O, KERNAL, RAM 0 48K
```

Miscellaneous

```
;--- Set shared RAM size to 16K
lda    mmurcr
and    #%11111100
ora    CRS_16K
sta    mmurcr


.macro SetVICBank bank
       lda    cia2pra
       and    #%11111100
       ora    #(3 – bank)
       sta    cia2pra
.endm
```

17XX RAM Expansion:

EXP_BASE:
DF00: Status Register – Read Only
b7: Interrupt Pending: 1 = interrupt waiting to be served
b6: End of Block: 1 = transfer complete
b5: Fault: 1 = block verify error
b4: Size: 1 = 256 KB on 1764 and 512K on a 1750
0 = 128 KB on 1700.
b3...0: Version 0

Note: Bits 7-5 are cleared when this register is read
Note: REU can be expanded in size beyond the original shown by Bit 4. Testing the RAM is the only way to find the actual size. CONFIGURE does this and puts the result in **ramExpSize**.

DF01: Command Register – Read/Write Write to this register to start operation.
b7: Execute: 1 = Transfer per current configuration (GEOS default = 1)
b6: Reserved:
b5: Load: 1 = enable AUTOLOAD option (GEOS default = 0)
With autoload enabled the address and length registers (see below) will be unchanged after a command execution. Otherwise the address registers will be counted up to the address of the last accessed byte of a DMA + 1, and the length register will be changed (normally to 1).
b4: FF00 1=Disable FF00 decode (GEOS default = 1)
If this bit is set command execution starts immediately after setting the command register. Otherwise command execution is delayed until write access to memory position **config** ($FF00)
b3-2: Reserved:
b1-0: Transfer type: 00 = transfer C64 –> REU
01 = transfer C64 <– REU
10 = swap C64 <-> REU
11 = compare C64 – REU
DF02: .word C64 base address
DF04: .word REU base address
DF06: .byte bank **Note**: When read, bits b7-b3 are always set
DF07: .word transfer size
DF09: Interrupt mask register – Read/Write
b7-5: Interrupt flags 000 = Interrupts disabled (GEOS default = 0)
b4-0: unused
DF0A: Address control register – Read/Write
b7-6: 00 = Increment both addresses (GEOS default = 0)
01 = Fix expansion address
10 = Fix C64 address
11 = Fix both addresses
b5..0: unused

Note[3]: By using a fixed address in the REU as a source you can very quickly initialize large blocks of RAM.

References:
1764 Ram Expansion Module Users Guide / 1700 1750 Ram Expansion Module Users Guide
http://www.zimmers.net/anonftp/pub/cbm/documents/chipdata/programming.reu Richard Hable

GEORAM

GEOS 2.0 requires version 2.0r to use a GEORAM.

An application will normally use the GEOS REU API to work with the GEORAM. Using the API will keep the application portable between systems with different REU types installed.

The GEORAM Unit has 512k bytes of RAM which appear to the system Unit as 2048 256-byte pages. The device has two page select registers (at $DFFE and $DFFF) to set up which page can be accessed by the processor.

The page select register is 6 bits wide at $DFFE. Each block of pages is 16K.
The block select register is 5 bits wide at $DFFF. (512 REU, each size upgrade gets another active bit).

Both registers are write-only locations, so an image must be kept of their current state if needed later. The memory itself appears as one 256-byte page at $DE00 to $DEFF.

$DE00 256-byte directly accessible page of RAM $DEFF		
$DF00 Do not write to this area		
$DFFE	b5-0 page select register	(256 byte pages)
$DFFF	b4-0 block select register	(16K blocks)

Size	Block Range (DFFE)	Total Number of Blocks (DFFF)
512K	$00 - 1F	32
1MB	$00 - 3F	64
2MB	$00 - 7F	128
4MB	$00 - FF	256

Example:

```
georampg=$DE00
georamps=$DFFE
georambs=$DFFF
GRB_SIZE=$4000                                             ; 16K page size
GRPG_SIZE=$100                                             ; 256 byte block size
REU_BANK=0                                                 ; rboot code is 128 bytes in bank 0
REU_ADDR=$BC40                                             ; and it is at address  $BC00+$40
GRAM_BLK=REU_BANK*4+REU_ADDR/GRB_SIZE                      ; 2    (2*GRB_SIZE    = $8000)
GRAM_PG=(REU_ADDR-((REU_ADDR/GRB_SIZE)*GRB_SIZE))/GRPG_SIZE; $3C  ($3C*GRPG_SIZE = $3C00)
   ;--- Restore reboot code from GEORAM for rboot.
          LoadB  georamps, #GRAM_PG     ; 3C   ($3C*GRPG_SIZE      = $3C00)
          LoadB  georambs, #GRAM_BLK    ; 02   (  2*GRB_SIZE       = $8000)
                                        ; Address in bank 0        = $BC00+$40
   ;--- Boot code is now visible at georampg
          ldx    #$7F
   10$    MoveB  "georambs+$40,x","BootGEOS,x"
          dex
          bpl    10$
```

6502 Instruction Set

Legend		
rel	Relative offset	signed value -128 to 127
zp	Zero Page address	$00 - $FF
abs	Absolute address	$0000 - $FFFF

Hi	Low Nibble											
	0	1	2	4	5	6	8	9	A	C	D	E
00	brk	ora (zp,X)			ora zp	asl zp	php	ora #	asl a		ora abs	asl abs
10	bpl rel	ora (zp),y			ora zp,x	asl zp,x	clc	ora abs,y			ora abs,x	asl abs,x
20	jsr abs	and (zp,x)		bit zp	and zp	rol zp	plp	and #	rol a	bit abs	and abs	rol abs
30	bmi rel	and (zp),y			and zp,x	rol zp,x	sec	and abs,y			and abs,x	rol abs,x
40	rti	eor (zp,x)			eor zp	lsr zp	pha	eor #	lsr a	jmp abs	eor abs	lsr abs
50	bvc rel	eor (zp),y			eor zp,x	lsr zp,x	cli	eor abs,y			eor abs,x	lsr abs,x
60	rts	adc (zp,x)			adc zp	ror zp	pla	adc #	ror a	jmp (abs)	adc abs	ror abs
70	bvs rel	adc (zp),y			adc zp,x	ror zp,x	sei	adc abs,y			adc abs,x	ror abs,x
80		sta (zp,x)		sty zp	sta zp	stx zp	dey		txa	sty abs	sta abs	stx abs
90	bcc rel	sta (zp),y		sty zp,x	sta zp,x	stx zp,y	tya	sta abs,y	txs		sta abs,x	
A0	ldy #	lda (zp,x)	ldx #	ldy zp	lda zp	ldx zp	tay	lda #	tax	ldy abs	lda abs	ldx abs
B0	bcs rel	lda (zp),y		ldy zp,x	lda zp,x	ldx zp,y	clv	lda abs,y	tsx	ldy abs,x	lda abs,x	ldx abs,y
C0	cpy #	cmp (zp,x)		cpy zp	cmp zp	dec zp	iny	cmp #	dex	cpy abs	cmp abs	dec abs
D0	bne rel	cmp (zp),y			cmp zp,x	dec zp,x	cld	cmp abs,y			cmp abs,x	dec abs,x
E0	cpx #	sbc (zp,x)		cpx zp	sbc zp	inc zp	inx	sbc #	nop	cpx abs	sbc abs	inc abs
F0	beq rel	sbc (zp),y			sbc zp,x	inc zp,x	sed	sbc abs,y			sbc abs,x	inc abs,x

D: Macros

Terms

Term	Description
addend	A number which is added to another.
addr	Target for a relative branch. Target of Macro Action
augend	The number to which an addend is added.
bitNumber	Index for bit position. example %10000000 / bitNumber 7 is set.
difference	Result of subtraction.
dest	An address to store a macro result.
immed	A Constant number.
minuend	A number from which another is to be subtracted.
result	The Sum of addition. New value after BIT operation.
source	An address to load from. Address or Immediate value in byte macros.
subtrahend	A number to be subtracted from another.
value	A Constant number.
zaddr	Zero Page Address.

Categories

Identifier	Category
bit	Bit operations.
br	Branching.
cmp	Comparisons.
flow	Alters flow of logic.
math	Math.
hw	Hardware.
util	Utility.

Sources

Identifier	Source
gP1	geoProgrammer1.1
gP'	geoProgrammer' 2.1
HGG	Created by PBM to perform actions for HGG Macros that were not defined in geoProgrammer1.1. **Example**: macro **bgt** is used in HGG but is not in geoProgrammer1.1. Macro logic was obvious so it was created here for use in the examples.
GPG	Official GEOS Programmer's Reference Guide
	Other sources will be added as used

bit operations

rmb		resets bit in destination byte.	gP1
	bitNumber	bit number in byte to reset.	
	dest	address of byte which contains bit to reset.	
		Destroys: nothing.	
rmbf		reset bit in byte.	gP1
	bitNumber	bit number in byte to reset.	
	dest	address of byte which contains bit to reset.	
		Destroys: a.	
setbit		Set bits in byte.	gP'
	source	address of byte which contains the bits to be set.	
	mask	address of bit mask to logical AND with *source*. (or immediate value)	
	bits	address of bits to logical OR with *source*. (or immediate value)	
		Destroys: a	
smb		Set bit in byte.	gP1
	bitNumber	bit number in byte to set (7 for MSD).	
	dest	address of byte which contains bit to set.	
		Destroys: nothing.	
smbf		Set bit in byte.	gP1
	bitNumber	bit number in byte to set.	
	result	address of byte which contains bit to set.	
		Destroys: a.	
tmb		Toggle bit in byte.	gP'
	bitNumber	bit number in byte to toggle.	
	result	address of byte which contains bit to toggle.	
		Destroys: nothing.	
tmbf		Toggle bit in byte.	gP'
	bitNumber	bit number in byte to toggle.	
	result	address of byte which contains bit to toggle.	
		Destroys: a.	

branching

bbeq		Branch if (source = 0).	gP'
	source	address of byte to test for zero.	
	addr	where to branch to if byte is zero.	
		Returns: a = value @*source*.	
bbmi		Branch if (source < 0). (bit 7 is set)	gP'
	source	address of signed byte to test for negative.	
	addr	where to branch to if byte is negative.	
		Returns: a = value @*source*.	
bbne		Branch if (source != 0).	gP'
	source	address of byte to test for not zero.	
	addr	where to branch to if byte is not zero.	
		Returns: a = value @*source*.	
bbpl		Branch if (source != 0).	gP'
	source	address of signed byte to test for positive.	
	addr	where to branch to if byte is positive.	
		Returns: a = value @*source*.	
bbr		tests bit in source byte, branches if reset.	gP1

	bitNumber source addr	bit number in byte to test (7 for MSD). address of byte which contains bit to test. where to branch to if bit is reset.	
bbrf	bitNumber source addr	Branch if bit reset. bit number in byte to test (7 for MSD). address of byte which contains bit to test. where to branch to if bit is reset. Destroys: a if bitNumber is < 6.	gP1
bbs	bitNumber source addr	Branch if bit set. bit number in byte to test (7 for MSD). address of byte which contains bit to test. where to branch to if bit is set. Destroys: nothing.	gP1
bbsf	bitNumber source addr	Branch if bit set. bit number in byte to test (7 for MSD). address of byte which contains bit to test. where to branch to if bit is set. Destroys: a if bitNumber is < 6.	gP1
bge	addr	Branch if (a >= b).	gP1
bgt	addr	Branch if (a > b).	HGG
ble	addr	Branch if (a <= b).	HGG
blt	addr	Branch if (a < b).	HGG
bra	addr	Unconditional branch to relative addr.	gP1
bweq	source addr	Branch if ([source |]source = 0) address of word to test for zero. where to branch to if source is zero.	gP'
bwne	source addr	Branch if ([source |]source != 0) address of word to test for zero. where to branch to if source is not zero.	gP'
bxeq	addr	Branch if (x-register = 0). where to branch to. Returns: a-register = x-register.	gP'
bxne	addr	Branch if (x-register != 0). where to branch to. Returns: a-register = x-register.	gP'

comparisons

CmpB	source dest	test (s == d). address of first byte (or #immediate value). address of second byte (or #immediate value).	gP1
CmpBI	source immed	test (s == #i). address of first byte. value to compare to.	gP1
CmpW	source dest	test (S == D). address of first byte. address of second byte.	gP1
CmpWI	source immed	test (S == #I). address of first word. constant value to compare to.	gP1

flow

clda		load accumulator on branch to label.	gP'
	label	Label for branch target.	
	addr	address load accumulator from on branch.	
cldxI		load x register on branch to label.	gP'
	label	Label for branch target.	
	value	#immediate value to load into x register on branch.	
cldyI		load y register on branch to label.	gP'
	label	Label for branch target.	
	value	#immediate value to load into y register on branch.	

math

add	addend	a = a + add.	gP1
AddAW		AU = AU + a-register.	gP'
	augend	address of word to add to.	
		Destroys: a.	
AddB		au = au + add.	gP1
	addend	address of byte to add, or #immediate value.	
	augend	address of byte to add to.	
		Destroys: a.	
AddBS		s = au + add.	gP'
	addend	address of byte to add, or #immediate value.	
	augend	address of byte to add to.	
	sum	address of byte to save result to.	
		Destroys: a.	
AddBSW		S = au + add.	gP'
	addend	address of byte to add, or #immediate value.	
	augend	address of byte to add to.	
	sum	address of word to save result to.	
		Destroys: a.	
AddBW		AU = AU + add.	gP'
	addend	address of byte to add, or #immediate value.	
	augend	address of word to add to.	
		Destroys: a.	
AddBWS		S = AU + add.	gP'
	addend	address of byte to add to augend.	
	augend	address of word to add to.	
	sum	address of word to save result to.	
		Destroys: a.	
AddCB		au = au + carry + add.	gP'
	addend	address of byte to add, or #immediate value.	
	augend	address of byte to add to.	
		Destroys: a.	
AddRW		AU = AU + #R.	gP'
	value	#Relocatable address (or #immediate value) to add to augend.	
		address of word to add to.	
	augend	Destroys: a.	

AddVB	value augend	au = au + #v. #immediate byte value to add to augend. address of byte to add to. Destroys: a.	gP1
AddVW	value augend	AU = AU + #V. #immediate byte or word value to add to augend. address of word to add to. Destroys: a.	gP1
AddVWS	addend augend sum	S = #AU + ADD. #immediate byte or word value to add to augend. address of word to add to. address of word to save result to. Destroys: a.	gP'
AddW	addend augend	AU = ADD + AU. address of word to add. address of word to add to. Destroys: a.	gP1
AddWS	addend augend sum	S = AU + ADD. address of word to add to augend. address of word to add to. address of word to save result to. Destroys: a.	gP'
AddYW	augend	AU = AU + y. address of word to add to. Destroys: a.	gP'
AddYWS	augend sum	S = AU + y. address of word to add to. address of word to save result to. Destroys: a.	gP'
DecW	addr	A = A -1. address of word to decrement. Destroys a.	gP'
sub	subtrahend	accumulator = accumulator – s. address of byte to subtract, or #immediate value. Destroys: a.	gP1
SubB	subtrahend minuend	m = m – s. address of byte to subtract, or #immediate value. address of byte to subtract from and store result to. Destroys: a.	gP1
SubBS	subtrahend minuend difference	m = m – s. address of byte to subtract, or #immediate value. address of byte to subtract from and store result to. address of byte to store the result. Destroys: a.	gP1
SubBW	subtrahend minuend	M = M – s. address of byte to subtract. address of word to subtract from. Destroys: a.	gP'

SubBWS	subtrahend minuend difference	M = M – s. address of byte to subtract. address of word to subtract from. address of word to store the result. Destroys: a.	gP'
SubVW	value minuend	M = M – #V. value of subtrahend. address of word to subtract from. Destroys: a.	gP'
SubVWS	subtrahend minuend difference	D = M – #S. value to subtract. address of word to subtract from. address of word to store the result. Destroys: a.	gP'
SubW	subtrahend minuend	M = M – S. address of word to subtract. address of word to subtract from. Destroys: a.	gP1
SubWS	subtrahend minuend difference	D = M – S. address of word to subtract. address of word to subtract from. address of word to store result. Destroys: a.	gP'
SubWVS	subtrahend minuend difference	D = #M – S. address of word to subtract. #immediate value to subtract from. address of word to store the result. Destroys: a.	gP'

utility

Dialog	dbBox	Call DoDlgBox address of dialog box structure to display	gP'
IncW	addr	A = A + 1. address of word to increment.	gP'
jsr_a	procedure param	a=param; jsr procedure. address of routine to call. address of byte to load, or #immediate value.	gP'
jsr_x	procedure param	x=param; jsr procedure. address of routine to call. address of byte to load into x, or #immediate value.	gP'
LdNull	addr	D = #$0000. address of word to load with null. *(accumulator is only loaded once).* Destroys: a.	gP'
LdWW	dest dest2 value	D,D2 = #V. address of word to load with value. address of second word to load with value. #immediate value to load. (constant or relocatable address) Destroys: a.	gP'

LoadB		d = #v.	gP1
	dest	address of byte to load with value.	
	value	#immediate value to load.	
		Destroys: a.	
LoadW		D = #V.	gP1
	dest	address of byte to load with value.	
	value	#immediate value to load.	
		Destroys: a.	
MoveB		d = s.	gP1
	source	source address.	
	dest	destination address.	
		Destroys: a.	
MoveW		D = S.	gP1
	source	source address.	
	dest	destination address.	
		Destroys: a.	
MvWW		D,D2 = S	gp'
	source	source address.	
	dest	destination address.	
	dest2	second destination address.	
		Destroys: a.	
PopB		Pull a byte from the stack.	gP1
	dest	where to store byte value.	
		Destroys: a.	
PopW		Pull a word from the stack.	gP1
	dest	where to store word value.	
		Destroys: a.	
PopX	-	Pull X from Stack.	gP'
		Destroys: a.	
PopY	-	Pull Y from Stack.	gP'
		Destroys: a.	
PushB		Push byte to stack.	gP1
	source	address of the byte to push (or #immediate value).	
PushW		Push the word at source onto the stack.	gP1
	source	address of the word to push.	
		Destroys: a.	
PushX	-	Push X to Stack.	gP'
		Destroys: a.	
PushY	-	Push Y to Stack.	gP'
		Destroys: a.	

By Name

add	addend	a = a + add.	gP1	math
AddAW		AU = AU + a-register.	gP'	math
	augend	address of word to add to.		
		Destroys: a.		
AddB		au = au + add.	gP1	math
	addend	address of byte to add, or #immediate value.		
	augend	address of byte to add to.		
		Destroys: a.		
AddBS		s = au + add.	gP'	math
	addend	address of byte to add, or #immediate value.		
	augend	address of byte to add to.		
	sum	address of byte to save result to.		
		Destroys: a.		
AddBSW		S = au + add.	gP'	math
	addend	address of byte to add, or #immediate value.		
	augend	address of byte to add to.		
	sum	address of word to save result to.		
		Destroys: a.		
AddBW		AU = AU + add.	gP'	math
	addend	address of byte to add, or #immediate value.		
	augend	address of word to add to.		
		Destroys: a.		
AddBWS		S = AU + add.	gP'	math
	addend	address of byte to add to augend.		
	augend	address of word to add to.		
	sum	address of word to save result to.		
		Destroys: a.		
AddCB		au = au + carry + add.	gP'	math
	addend	address of byte to add, or #immediate value.		
	augend	address of byte to add to.		
		Destroys: a.		
AddRW		AU = AU + #R.	gP'	math
	value	#Relocatable address (or #immediate value) to add to augend.		
		address of word to add to.		
	augend	Destroys: a.		
AddVB		au = au + #v.	gP1	math
	value	#immediate byte value to add to augend.		
	augend	address of byte to add to.		
		Destroys: a.		
AddVW		AU = AU + #V.	gP1	math
	value	#immediate byte or word value to add to augend.		
	augend	address of word to add to.		
		Destroys: a.		
AddVWS		S = AU + #ADD.	gP'	math
	addend	#immediate byte or word value to add to augend.		
	augend	address of word to add to.		
	sum	address of word to save result to.		
		Destroys: a.		

AddW		AU = ADD + AU.	gP1	math
	addend	address of word to add.		
	augend	address of word to add to.		
		Destroys: a.		
AddWS		S = AU + ADD.	gP'	math
	addend	address of word to add to augend.		
	augend	address of word to add to.		
	sum	address of word to save result to.		
		Destroys: a.		
AddYW		AU = AU + y.	gP'	math
	augend	address of word to add to.		
		Destroys: a.		
AddYWS		S = AU + y.	gP'	math
	augend	address of word to add to.		
	sum	address of word to save result to.		
		Destroys: a.		
bbeq		Branch if (source = 0).	gP'	br
	source	address of byte to test for zero.		
	addr	where to branch to if byte is zero.		
		Returns: a = value @*source*.		
bbmi		Branch if (source < 0). (bit 7 is set)	gP'	br
	source	address of signed byte to test for negative.		
	addr	where to branch to if byte is negative.		
		Returns: a = value @*source*.		
bbne		Branch if (source != 0).	gP'	br
	source	address of byte to test for not zero.		
	addr	where to branch to if byte is not zero.		
		Returns: a = value @*source*.		
bbpl		Branch if (source != 0).	gP'	br
	source	address of signed byte to test for positive.		
	addr	where to branch to if byte is positive.		
		Returns: a = value @*source*.		
bbr		tests bit in source byte, branches if reset.	gP1	br
	bitNumber	bit number in byte to test (7 for MSD).		
	source	address of byte which contains bit to test.		
	addr	where to branch to if bit is reset.		
bbrf		Branch if bit reset.	gP1	br
	bitNumber	bit number in byte to test (7 for MSD).		
	source	address of byte which contains bit to test.		
	addr	where to branch to if bit is reset.		
		Destroys: a if bitNumber is < 6.		
bbs		Branch if bit set.	gP1	br
	bitNumber	bit number in byte to test (7 for MSD).		
	source	address of byte which contains bit to test.		
	addr	where to branch to if bit is set.		
		Destroys: nothing.		

bbsf	bitNumber source addr	Branch if bit set. bit number in byte to test (7 for MSD). address of byte which contains bit to test. where to branch to if bit is set. Destroys: accumulator if bitNumber is < 6.	gP1	br
bge	addr	Branch if (a >= b).	gP1	br
bgt	addr	Branch if (a > b).	HGG	br
ble	addr	Branch if (a <= b).	HGG	br
blt	addr	Branch if (a < b).	HGG	br
bra	addr	Unconditional branch to relative addr.	gP1	br
bweq	source addr	Branch if (source | (source +1) = 0). address of word to test for zero. where to branch to if source is zero.	gP'	br
bwne	source addr	Branch if (source | (source +1) != 0). address of word to test for zero. where to branch to if source is not zero.	gP'	br
bxeq	addr	Branch if (x-register = 0). where to branch to. Returns: a-register = x-register.	gP'	br
bxne	addr	Branch if (x-register != 0). where to branch to. Returns: a-register = x-register.	gP'	br
clda	label addr	load accumulator on branch to label. Label for branch target. address load accumulator from on branch.	gP'	flow
cldxI	label value	load x register on branch to label. Label for branch target. #immediate value to load into x register on branch.	gP'	flow
cldyI	label value	load y register on branch to label. Label for branch target. #immediate value to load into y register on branch.	gP'	flow
CmpB	source dest	test (s == d). address of first byte (or #immediate value). address of second byte (or #immediate value).	gP1	cmp
CmpBI	source immed	test (s == #i). address of first byte. value to compare to.	gP1	cmp
CmpW	source dest	test (S == D). address of first byte. address of second byte.	gP1	cmp
CmpWI	source immed	test (S == #I). address of first word. constant value to compare to.	gP1	cmp
DecW	addr	A = A -1. address of word to decrement. Destroys a.	gP'	math
Dialog	dbBox	Call DoDlgBox. address of dialog box structure to display	gP'	util

IncW		A = A + 1.	gP'	util
	addr	address of word to increment.		
jsr_a		a=param; jsr procedure.	gP'	util
	procedure	address of routine to call.		
	param	address of byte to load into a, or #immediate value.		
jsr_x		x=param; jsr procedure.	gP'	util
	procedure	address of routine to call.		
	param	address of byte to load into x, or #immediate value.		
LdNull		D = #$0000.	gP'	util
	addr	address of word to load with null.		
		(accumulator is only loaded once).		
		Destroys: a.		
LdWW		D,D2 = #V.	gP'	util
	dest	address of word to load with value.		
	dest2	address of second word to load with value.		
	value	#immediate value to load. (constant or relocatable address)		
		Destroys: a.		
LoadB		d = #v.	gP1	util
	dest	address of byte to load with value.		
	value	#immediate value to load.		
		Destroys: a.		
LoadW		D = #V.	gP1	util
	dest	address of byte to load with value.		
	value	#immediate value to load.		
		Destroys: a.		
MoveB		d = s.	gP1	util
	source	source address.		
	dest	destination address.		
		Destroys: a.		
MoveW		D = S.	gP1	util
	source	source address.		
	dest	destination address.		
		Destroys: a.		
MvWW		D,D2 = S	gp'	util
	source	source address.		
	dest	destination address.		
	dest2	second destination address.		
		Destroys: a.		
PopB		Pull a byte from the stack.	gP1	util
	dest	where to store byte value.		
		Destroys: a.		
PopW		Pull a word from the stack.	gP1	util
	dest	where to store word value.		
		Destroys: a.		
PopX	-	Pull X from Stack.	gP'	util
		Destroys: a.		
PopY	-	Pull Y from Stack.	gP'	util
		Destroys: a.		

PushB	source	Push byte to stack. address of the byte to push (or #immediate value).	gP1	util
PushW	source	Push the word at source onto the stack. address of the word to push.	gP1	util
PushX	-	Push X to Stack. Destroys: a.	gP'	util
PushY	-	Push Y to Stack. Destroys: a.	gP'	util
rmb	bitNumber dest	resets bit in destination byte. bit number in byte to reset. address of byte which contains bit to reset. Destroys: nothing.	gP1	bit
rmbf	bitNumber dest	reset bit in byte. bit number in byte to reset. address of byte which contains bit to reset. Destroys: a.	gP1	bit
setbit	source mask bits	Set bits in byte. address of byte which contains the bits to be set. address of bit mask to logical AND with *source*. (or immediate value) address of bits to logical OR with *source*. (or immediate value) Destroys: a	gP'	bit
smb	bitNumber dest	Set bit in byte. bit number in byte to set (7 for MSD). address of byte which contains bit to set. Destroys: nothing.	gP1	bit
smbf	bitNumber result	Set bit in byte. bit number in byte to set. address of byte which contains bit to set. Destroys: a.	gP1	bit
sub	subtrahend	accumulator = accumulator – s. address of byte to subtract, or #immediate value. Destroys: a.	gP1	math
SubB	subtrahend minuend	m = m – s. address of byte to subtract, or #immediate value. address of byte to subtract from and store result to. Destroys: a.	gP1	math
SubBS	subtrahend minuend difference	m = m – s. address of byte to subtract, or #immediate value. address of byte to subtract from and store result to. address of byte to store the result. Destroys: a.	gP1	math
SubBW	subtrahend minuend	M = M – s. address of byte to subtract. address of word to subtract from. Destroys: a.	gP'	math

SubBWS		M = M – s.	gP'	math
	subtrahend	address of byte to subtract.		
	minuend	address of word to subtract from.		
	difference	address of word to store the result.		
		Destroys: a.		
SubVW		M = M – #V.	gP'	math
	value	value of subtrahend.		
	minuend	address of word to subtract from.		
		Destroys: a.		
SubVWS		D = M – #S.	gP'	math
	subtrahend	value to subtract.		
	minuend	address of word to subtract from.		
	difference	address of word to store the result.		
		Destroys: a.		
SubW		M = M – S.	gP1	math
	subtrahend	address of word to subtract.		
	minuend	address of word to subtract from.		
		Destroys: a.		
SubWS		D = M – S.	gP'	math
	subtrahend	address of word to subtract.		
	minuend	address of word to subtract from.		
	difference	address of word to store result.		
		Destroys: a.		
SubWVS		D = #M – S.	gP'	math
	subtrahend	address of word to subtract.		
	minuend	#immediate value to subtract from.		
	difference	address of word to store the result.		
		Destroys: a.		
tmb		Toggle bit in byte.	gP'	bit
	bitNumber	bit number in byte to toggle.		
	result	address of byte which contains bit to toggle.		
		Destroys: nothing.		
tmbf		Toggle bit in byte.	gP'	bit
	bitNumber	bit number in byte to toggle.		
	result	address of byte which contains bit to toggle.		
		Destroys: a.		

add: math

Form: **add** addend gP1

Function: a = a + add.

Parameters: addend address of byte to add, or #immediate value.

Returns: sum in accumulator.

Destroys: nothing.

Description: Add the *addend* to the accumulator. *addend* is either an address or an immediate byte value. If it is an address, the byte at the address is added to the value in the a-register. If it is an immediate value (preceded by a # sign), the actual value is added to the a-register. The result is returned in the a-register. The sole purpose of the add macro is to combine the adc with its mandatory clc instruction.

Note: Result is not stored.

Example:

```
      add    #12
      ...
      add    mouseYPos
```

```
.macro add   addend
      clc                          ; clear carry to start an addition
      adc    addend                ; add addend to the accumulator
.endm
```

```
; Sample of how a macro is stored in GEOASSEMBLER.

macro body:
.byte  "clc",CR                 ; 4 bytes:  mnemonic 3 bytes
                                ;           line terminator 1 byte
.byte  "adc ",$01,PAGE_BREAK    ; 6 bytes:  mnemonic 3 bytes
                                ;           [SPACE] 1 byte
                                ;           parameter number 1 byte
                                ;           macro terminator 1 byte (PAGE_BREAK)
                                ===============
                                ;Total: 10 bytes
```

See also: **AddB**, **AddW**.

AddAW: math

Form: **AddAW** augend gP'

Function: AU = AU + a-register.

Parameters: augend address of word to add the a-register to.

Destroys: a.

Description: Add a-register to word at location of *augend*.

Note:

Example:

```
      ;---   Filter geoWrite page of esc objects
      10$    bbeq   "(r0),y",90$ ; exit when end of file found
             cmp    #PAGE_BREAK
             beq    90$          ; exit when end of page found
             cmp    #NEWCARDSET
             beq    17$
             cmp    #ESC_RULER
             beq    18$
             cmp    #ESC_GRAPHICS
             beq    19$
             sta    (r1),y
             IncW   r0
             IncW   r1
             bra    10$

      17$    lda    #4           ; size of NEWCARDSET
clda  18$,          zGWRulerSize ; size of Ruler in buffer. (V1.1 and 2.x have different sizes)
clda  19$,          #5           ; size of ESC_GRAPHICS
             AddAW  r0           ; add size of current object to buffer pointer
             bra    10$          ; loop back to get next character
```

```
.macro AddAW augend
      clc                                  ; clear carry to start an addition
      adc    augend                        ; add a-register to low-byte of augend
      sta    augend                        ; store updated low-byte of augend
      bcc    z                             ; if carry is not set then done
      inc    augend+1                      ; else increment high-byte of augend
z:
.endm

.macro AddAW augend                        ; Compact next form
      add    augend                        ; add a-register to low-byte of augend
      sta    augend                        ; store updated low-byte of augend
      bcc    z                             ; if carry is not set then done
      inc    augend+1                      ; else increment high-byte of augend
z:
.endm
```

See also: **AddB**, **AddW**.

AddB: math

Form:	**AddB** addend, augend	gP1
Function:	au = au + add.	

Parameters:

addend	address of byte to add, or #immediate value.
augend	address of byte to add to.

Destroys: a.

Returns:

C=1	addition overflowed the result byte.
C=0	no overflow.

Description: Adds the byte at one address (*addend*) to the byte at another address (*augend*) and stores the result in *augend*.

Example:

```
;--- Move input prompt by amount in r1L lines.
AddB   r1L,stringY
...
;--- Move input prompt down 10 scan lines.
AddB   #$0A,stringY
```

```
.macro AddB addend, augend
      clc                              ; must start a new add with carry cleared
      lda    addend                    ; get value to add
      adc    augend                    ; add to value to add too
      sta    augend                    ; store result
.endm
```

See also: **AddCB**, **AddBS, AddBW**, **AddW**.

AddBS: math

Form:	**AddBS** addend, augend, sum	gP'

Function: s = au + add.

Parameters:

addend	address of byte to add, or #immediate value.
augend	address of byte to add to.
sum	address of byte to save result to.

Destroys: a.

Returns:

C=1	Addition overflowed the result byte.
C=0	No overflow.

Description: Add *addend* to *augend* and save result in *sum*.

Note: Any overflow is lost and will be reflected by C=1 on return.

Example:

```
;--- Move input prompt by amount in r1L lines from offset defined in zCurOffSet
AddBS  r1L,zCurOffSet,stringY
...
;--- Move input prompt down 10 scan lines from offset defined in zCurOffSet
AddBS  #$0A,zCurOffSet,stringY
```

```
.macro AddBS addend, augend, sum
      lda    augend                      ; get augend to add to
      clc                                ; must start a new add with carry cleared
      adc    addend                      ; add the addend byte
      sta    sum                         ; store result in sum byte
.endm
```

See also: **AddBSW**.

AddBSW: math

Form: **AddBSW** addend, augend, sum gP'

Function: S = au + add.

Parameters:

addend	address of byte to add, or #immediate value.
augend	address of byte to add to.
sum	address of word to save result to.

Destroys: a.

Description: Add *addend* to *augend* and save word sized result in *sum*.

Note:

Example:

```
;---   Add value in r1L to current Platform line to get new reach from platform
AddBSW r1L,zCurOffSet,zReach
...
;---   Add 10 to current Platform line to get new reach from platform
AddBSW #$0A,zPlatform,zReach
```

```
.macro AddBSW addend, augend, sum
      lda    augend                     ; get the value to add too
      clc                               ; must start a new add with carry cleared
      adc    addend                     ; add the addend byte to the augend
      sta    sum                        ; store result
      lda    #0                         ; set high-byte of sum to zero
      adc    #0                         ; add overflow to high byte of sum
      sta    sum+1
.endm
```

See also: **AddW**.

AddBW: math

Form:	**AddBW** addend, augend	gP'

Function: AU = AU + add.

Parameters:

addend	address of byte to add, or #immediate value.
augend	address of word to add to.

Destroys: a.

Description: Add *addend* (byte) to word at location of *augend* and save the result in *augend*.

Note:

Example:

```
;--- Calculate new file size by the value in nbrBlks
AddBW  nbrBlks,fileSize
...
;--- Calculate pointer to next icon using size of icon structure
AddBW  #OFF_NX_ICON,r0
```

```
.macro AddBW addend, augend
      lda    addend                     ; load addend low-byte
      clc                               ; clear carry to start an addition
      adc    augend                     ; add to low-byte of augend
      sta    augend                     ; store updated augend
      bcc    z                          ; if carry is not set then done
      inc    augend+1                   ; else increment high-byte of augend
z:
.endm
```

See also: **AddB**, **AddW**.

AddBWS: math

Form: **AddBWS** addend, augend, sum gP'

Function: SUM = AU + add.

Parameters:	addend	address of byte to add, or #immediate value.
	augend	address of word to add to.
	sum	address of word to add to save result.

Destroys: a.

Description: Add *addend* (byte) to word at location of *augend* and save the result to byte pointed to by *sum*.

Note:

Example:

```
;--- Calculate temporary file size to test if new addition will fit on disk
AddBWS nbrBlks,fileSize,sizeCheck
...
;--- Calculate pointer to next icon from reference pointer in r14
;--- using size of icon structure
AddBWS #OFF_NX_ICON,r14,r0
```

```
.macro AddBWS addend, augend, sum
      lda   augend                     ; load augend low-byte
      clc                              ; clear carry to start an addition
      adc   addend                     ; add addend byte
      sta   sum                        ; store result in sum
      lda   augend+1                   ; add carry to the
      adc   #0                         ; high-byte of augend
      sta   sum+1                      ; and save in sum
.endm
```

See also: **AddB**, **AddW**.

AddCB: **math**

Form: **AddCB** addend, augend gP'

Function: au = au + carry + add.

Parameters:

addend	address of byte to add, or #immediate value.
augend	address of byte to add to.

Destroys: a.

Returns:

C=1	addition overflowed the result byte.
C=0	no overflow.

Description: Adds the carry and the byte at one address (*addend*) to the byte at another address (*augend*) and stores the result in *augend*.

Note:

Example:

```
;--- Add word to an indexed word.
AddB   r1L,"rValuesL,X"          ; AddB clears the carry and adds the values
AddCB  r1H,"rValuesH,X"          ; AddCB includes the carry in the addition

;--- Add word to an indirect indexed word.
AddB   r1L,"(zValues),Y"
iny                              ; advance index to next byte
AddCB  r1H,"(zValues),Y"

;--- Add word to an indirect indexed word.
AddB   r1L,"(zValues),Y"
IncW   zValues                   ; advance pointer to next byte
AddCB  r1H,"(zValues),Y"
```

```
.macro AddCB addend, augend
      lda    addend                  ; get value to add
      adc    augend                  ; add carry + addend
      sta    augend                  ; store result
.endm
```

See also: **AddB**.

AddRW: utility

Form:	**AddRW** value, augend	gP'
Function:	**Function**: AU = AU + #R.	
Parameters:	value #Relocatable address (or #immediate value) to add to augend. augend address of word to add to.	
Destroys	a.	
Description:	Adds a relocatable address or (#immediate value) (*value)* to the word at *augend* and stores the result in *augend*.	
Note:		
Example:		

```
...
AddRW        rBuffer, pointer ; add start of buffer address to pointer
```

```
.macro AddRW value, augend
      lda    #[(value)                  ; load low-byte of value
      clc                               ; clear carry to start an addition
      adc    augend                     ; add to low-byte of augend
      sta    augend                     ; store updated augend
      lda    #](value)                  ; carry was set if adc above overflowed
      adc    augend+1                   ; add carry + value to high-byte of address
      sta    augend+1                   ; store result
.endm
```

See also:

AddVB: math

Form:	**AddVB** value, augend	gP1

Function: au = au + #v.

Parameters:

value	#immediate value to add to augend.
augend	address of byte to add to.

Destroys: a.

Description: Adds an immediate byte value (*value)* to the byte at *augend* and stores the result in *augend*.

Note: This macro is redundant with **AddB**. **AddB** can do immediate values as well. **AddVB** was left in geoProgrammer' 2.1 for backwards compatibility with existing source.

Note: Use **AddBs**, or **AddBS** to add a value to a byte and store into a different address.

Example:

```
;--- Move input prompt down 10 scan lines.
AddVB     #$0A,stringY    ; Macro adds the #. Redundant to use it again here.
```

```
.macro AddVB value, augend
      lda   augend                    ; load low-byte of augend
      clc                             ; clear carry to start an addition
      adc   #value                    ; add #immediate value
      sta   augend                    ; store result
.endm
```

See also: **AddW**.

AddVW: math

Form: **AddVW** value, augend gP1

Function: AU = AU + #V.

Parameters:

value	#immediate byte or word value to add to augend.
augend	address of word to add to.

Destroys: a.

Description: Adds an immediate byte or word value (*value)* to the word at *augend* and stores the result in *augend*.

Note:

Example: **Find**.

```
;--- Move input prompt to the right by 12 pixels
AddVW     #12,stringX
```

```
.macro AddVW value, augend
      clc                               ; clear carry to start an addition
      lda    #[(value)                  ; load low-byte of value
      adc    augend                     ; add to low-byte of augend
      sta    augend                     ; store updated augend
.if   (value >= 0) && (value <= 255)
      bcc    z                          ; carry was set if adc above overflowed
      inc    augend+1                   ; increment high-byte of word
z:
.else
      lda    #](value)                  ; carry was set if adc above overflowed
      adc    augend+1                   ; add carry + value to high-byte of address
      sta    augend+1                   ; store result
.endif
.endm
```

See also: **AddB**.

AddVWS: math

Form:	**AddVWS**	addend, augend, sum	gP'

Function: S = AU + #ADD.

Parameters:

addend	#immediate value to add to augend.
augend	address of word to add to.
sum	address of word to save the result.

Destroys: a.

Description: Add *addend* to *augend* and store in *sum.*

Note:

Example: **ClipChar**

```
AddVWS       #$400,r1,r0  ; Add $400 to value in r1 and save result in r0.
```

```
.macro AddVWS addend, augend, sum
      lda    augend                     ; load low-byte of word being added to
      clc                               ; clear carry to start an addition
      adc    #[(addend)                 ; add low-byte of addend
      sta    sum                        ; save result in sum
      lda    augend+1                   ; now add the high-byte and save it
      adc    #](addend)
      sta    sum+1
.endm
```

See also: **AddB**.

AddW: math

Form:	**AddW**	addend, augend	gP1

Function: AU = ADD + AU.

Parameters:

addend	address of word to add to augend.
augend	address of word to add to.

Destroys: a.

Description: Adds the word at *addend* to the word at *augend* and stores the result in *augend*.

Note:

Example:

```
.macro AddW addend, augend
      lda    addend                  ; load addend low-byte
      clc                            ; clear carry to start an addition
      adc    augend                  ; add to destination low-byte
      sta    augend                  ; store result, sec carry with overflow
      lda    addend+1                ; load source high-byte
      adc    augend+1                ; add with carry to high-byte dest
      sta    augend+1                ; store result
.endm
```

See also: **AddB**.

AddWS: math

Form: **AddWS** addend, augend, sum gP'

Function: AU = ADD + AU.

Parameters:

addend	address of word to add to augend.
augend	address of word to add to.
sum	address of word to save the result.

Destroys: a.

Description: Add *addend* to *augend* and store in *augend*.

Note:

Example:

```
.macro AddWS addend, augend, sum
      lda    addend                     ; load addend low-byte
      clc                               ; clear carry to start an addition
      adc    augend                     ; add to destination low-byte
      sta    sum                        ; store result, sec carry with overflow
      lda    addend+1                   ; load source high-byte
      adc    augend+1                   ; add with carry to high-byte dest
      sta    sum+1                      ; store result
.endm
```

See also: **AddB**.

AddYW: math

Form:	**AddYW** augend	gP'
Function:	AU = AU + y.	
Parameters:	y ADDEND — value in y to add to augend. augend address of word to add to.	
Destroys:	a.	
Description:	Add *ADDEND* to *augend* and store sum in *augend*.	
Note:		
Example:		

```
.macro AddYW augend
      tya                                 ; put addend in a
      clc                                 ; reset carry flag
      adc    augend                       ; add addend to low-byte of augend
      sta    augend
      bcc    z                            ; if carry is set then increment high-byte of augend
      inc    augend+1
z:
.endm
```

See also: **AddYWS**.

AddYWS: math

Form:	**AddYWS** augend, sum	gP'

Function: S = AU + y.

Parameters:

y	ADDEND — value in y to add to augend.
augend	address of word to add to.
sum	address of word to save the result.

Destroys: a.

Description: Add *ADDEND* to *augend* and store result in *sum*.

Note:

Example: **`ClipChar`**, **`MySetGDirEntry`**.

```
.macro AddYWS augend, sum
      tya                             ; put addend in a
      clc                             ; reset carry flag
      adc    augend                   ; add addend to low-byte of augend
      sta    sum                      ; save low-byte to sum
      lda    #0                       ;
      adc    augend+1                 ; add carry to the high-byte
      sta    sum+1                    ; save high-byte of the result
.endm
```

See also: **AddYW**.

bbeq: branch

Form:	**bbeq** source, addr	gP'

Function: Branch if (source = 0).

Parameters:

source	address of byte to test for zero.
addr	where to branch to if byte is zero.

Returns: a = value @*source.*

Description: Branch to addr if *source* byte is zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note:

Example:

```
bbeq            yPos, 60$       ;if y position is zero then branch to handle edge of screen action.
```

```
.macro bbeq source, addr
      lda    source                    ; load source byte
      beq    addr                      ; branch if zero
.endm
```

See also: **bbne**.

bbmi: branch

Form:	**bbmi** source, addr	HGG

Function: Branch if (source < 0). (bit 7 is set)

Parameters:

source	address of signed byte to test for negative.
addr	where to branch to if byte is negative.

Returns: a = value @*source.*

Description: Branch to addr if *source* byte is less than zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note: Use "**bbsf** 7,source, addr" to branch when negative while not altering the accumulator.

Example:

```
.macro bbmi source, addr
      lda    source                     ; load source byte
      bmi    addr                       ; branch if negative (bit 7 is set)
.endm
```

See also: **bbpl**, **bbsf**

bbne: branch

Form: **bbne** source, addr gP'

Function: Branch if (source != 0).

Parameters:

source	address of byte to test for not zero.
dest	where to branch to if byte is not zero.

Returns: a = = value @*source.*

Description: Branch to addr if *source* byte is not zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note:

Example:

```
.macro bbne source, addr
      lda    source                     ; load source byte
      bne    addr                       ; branch if not zero
.endm
```

See also: **bbeq**.

bbpl: branch

Form:	**bbpl** source, addr	gP'

Function: Branch if (source >= 0). (bit 7 is not set)

Parameters:

source	address of signed byte to test for positive.
addr	where to branch to if byte is positive.

Returns: a = value @*source*.

Description: Branch to addr if *source* byte is greater than or equal to zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note: Use "7,**bbrf** source,addr" to branch on positive while not altering the accumulator.

Example:

```
.macro bbpl source, addr
      lda   source                  ; load source byte
      bpl   addr                    ; branch if positive (bit 7 is not set)
.endm
```

See also: **bbmi**.

bbr: branch

Form:	**bbr** bitNumber, source, addr	gP1

Function: test bit in source byte, branch on reset.

Parameters:

bitNumber	bit number in byte to test (7 for MSD, 0 for LSD).
source	address of byte which contains bit to test.
addr	where to branch to if bit is reset.

Destroys: nothing.

Description: Tests a bit in the byte at *source*. *bitNumber* is the bit to test; it is a value which ranges from zero to seven, with zero being the LSB and seven being the MSB of the byte. If the bit is reset, a relative branch to *addr* is taken. Otherwise, it falls through to the next instruction. Does not affect any registers.

Note: No status registers will change as a result of the test.

Example:

```
bbr    MOUSEON_BIT,mouseOn,SM_rts
```

```
.macro bbr bitNumber, source, addr
      php                               ; save processor status register
      pha                               ; save a
      lda    source                     ; load byte to be tested
      and    #(1 << bitNumber)          ; mask out the bit to test
      bne    z                          ; if bit set then done
      pla                               ; else
      plp                               ;    restore a and process status registers
      bra    addr                       ;    branch to target
z:
      pla                               ; restore a
      plp                               ; restore processor status register
.endm
```

See also: **bbrf**.

bbrf: branch

Form:	**bbrf** bitNumber, source, addr	gP1

Function: Branch if bit reset.

Parameters:

bitNumber	bit number in byte to test (7 for MSD, 0 for LSD).
source	address of byte which contains bit to test.
addr	where to branch to if bit is set.

Destroys: if bitNumber is < 6:

a.

if bitNumber is 6 or 7:

nothing.

Description: Tests a bit in the byte at *source*. *bitNumber* is the bit to test; it is a value which ranges from zero to seven, with zero being the LSB and seven being the MSB of the byte. If the bit is reset, a relative branch to *addr* is taken. Otherwise, it falls through to the next instruction. Identical to **bbr**, except it is faster and affects the ST and a-register.

Note: Fast version that destroys the accumulator. Use **bbs** to preserve a. (The a-register is only destroyed when testing bits 0-5).

Example: **o_UpdateMouse**.

```
bbrf   MOUSEON_BIT,mouseOn,SM_rts
```

```
.macro bbrf bitNumber, source, addr
.if    (bitNumber = 7)                   ; bits 7 and 6 have fast checks for bit set
       bit    source
       bpl    addr
.elif  (bitNumber = 6)
       bit    source
       bvc    addr
.else
       lda    source                     ; other bits require a load and a test
       and    #(1 << bitNumber)
       beq    addr
.endif
.endm
```

See also: **bbr**.

bbs: branch

Form:	**bbs** bitNumber, source, addr	gP1

Function: Branch if bit set.

Parameters:

bitNumber	bit number in byte to test (7 for MSD, 0 for LSD).
source	address of byte which contains bit to test.
addr	where to branch to if bit is set.

Destroys: nothing.

Description: Tests a bit in the byte at *source*. *bitNumber* is the bit to test; it is a value which ranges from zero to seven, with zero being the LSB and seven being the MSB of the byte. If the bit is set, a relative branch to *addr* is taken. Otherwise, it falls through to the next instruction. Does not affect any registers.

Note: Process status register is preserved and does not reflect the results of the bit test.

Note: **bbs** should only be used instead of **bbsf** if the accumulator needs to be preserved.

Example:

```
bbsf   KEYPRESS_BIT,pressFlag,KbdChg
```

```
.macro bbs bitNumber, source, addr
      php                            ; save processor status register
      pha                            ; save a
      lda    source                  ; load byte to be tested
      and    #(1 << bitNumber)       ; mask out the bit to test
      beq    z                       ; if reset then done
      pla                            ; else
      plp                            ;    restore a and process status registers
      bra    addr                    ;    branch to target
z:
      pla                            ; restore a
      plp                            ; restore processor status register
.endm
```

See also: **bbr**.

bbsf: branch

Form:	**bbsf** bitNumber, source, addr	gP1

Function: Branch if bit set.

Parameters:

bitNumber	bit number in byte to test (7 for MSD, 0 for LSD).
source	address of byte which contains bit to test.
addr	where to branch to if bit is set.

Description: Tests a bit in the byte at *source*. *bitNumber* is the bit to test; it is a value which ranges from zero to seven, with zero being the LSB and seven being the MSB of the byte. If the bit is set, a relative branch to *addr* is taken. Otherwise, it falls through to the next instruction. Identical to **bbs**, except it is faster and affects the ST and a-register.

Destroys: if *BITPOS* is < 6
    a.
if *BITPOS* >= *6*
    nothing.

Note: Fast version that destroys the accumulator. Use **bbs** to preserve a. (The a-register is only destroyed when testing bits 0-5).

Example:

```
bbsf   MOUSE_BIT,pressFlag,MseChg
```

```
.macro bbsf bitNumber, source, addr
.if    (bitNumber = 7)                 ; bits 7 and 6 have fast checks for bit set
       bit    source
       bmi    addr
.elif  (bitNumber = 6)
       bit    source
       bvs    addr
.else
       lda    source                   ; other bits require a load and a test
       and    #(1 << bitNumber)
       bne    addr
.endif
.endm
```

See also: **bbr**.

bge: branch

Form:	**bge** addr	gP1
Function:	Branch if (a >= b).	
Parameters:	addr where to branch to.	
Destroys:	nothing.	
Description:	If carry flag is set, then branch to *addr*.	
Note:		
Example:	**RoadTrip**.	

```
.macro bge addr
      bcs    addr                          ; if carry set then branch to addr
.endm
```

See also: **bge**, **bgt**, **blt**, **ble**.

bgt: branch

Form:	**bgt** addr	HGG
Function:	Branch if (a > b).	
Parameters:	addr where to branch to.	
Destroys:	nothing.	
Description:	If carry flag is set and if zero flag is not set, then branch to *addr*.	
Note:		
Example:	**NewIsMseInRegion**.	

```
.macro bgt addr
      beq    z                         ; if zero flag set then done
      bcs    addr                      ; if carry set then branch to addr
z:
.endm
```

See also: **bge**, **bgt**, **blt**, **ble**.

ble: branch

Form:	**ble** addr	HGG
Function:	Branch if (a <= b).	
Parameters:	addr where to branch to.	
Destroys:	nothing.	
Description:	If carry flag is clear or if zero flag is set, then branch to *addr*.	
Note:		
Example:		

```
CmpB        mouseYPos,#10      ; check position of the mouse
ble         MseAtTop           ; branch if mouse is less than or equal to our top
```

```
.macro ble addr
      bcc    addr                     ; branch if carry clear
      beq    addr                     ; branch if zero flag set
.endm
```

See also: **bge**, **bgt**, **blt**, **ble**.

blt: branch

Form:	**blt** addr	HGG
Function:	Branch if (a < b) —> addr.	
Parameters:	addr where to branch to.	
Destroys:	nothing.	
Description:	If carry flag is reset, then branch to *addr*.	
Note:		
Example:	**NewIsMseInRegion**.	

```
.macro blt addr
      bcc    addr                          ; branch if carry clear
.endm
```

See also: **bge**, **bgt**, **ble**, **bra**.

bra: branch

Form:	**bra** addr	gP1
Function:	Unconditional relative branch to addr.	
Parameters:	addr where to branch to.	
Destroys:	nothing.	
Description:	Generates an unconditional relative branch. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is valid address or label; it can be a local label.	
Note:		
Example:	**RoadTrip**.	

```
.macro bra addr
      clv                                ; clear overflow flag
      bvc    addr                        ; branch on overflow clear to addr
.endm
```

See also: **bge**, **bgt**, **blt**, **ble**.

bweq: branch

Form:	**bweq** source, addr gP'
Function:	Branch if (source | (source +1) = 0).
Parameters:	source address of word to test for zero. addr where to branch to.
Destroys:	a.
Description:	Branch to addr if *source* word is zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).
Note:	
Example:	**RoadTrip**.

```
.macro bweq source, addr
      lda    source                     ; load low-byte of source
      ora    source+1                   ; or with high-byte of source
      beq    addr                       ; branch if zero flag is set
.endm
```

See also: **bwne**.

bwne: branch

Form:	**bwne** source, addr	gP'

Function: Branch if (source | (source +1) != 0).

Parameters:

source	address of word to test for zero.
addr	where to branch to.

Destroys: a.

Description: Branch to addr if *source* word is not zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note:

Example: **RoadTrip**.

```
.macro bwne source, addr
      lda    source                     ; load low-byte of source
      ora    source+1                   ; or with high-byte of source
      bne    addr                       ; branch if zero flag is not set
.endm
```

See also: **bweq**.

bxeq: branch

Form: **bxeq** addr gP'

Function: Branch if (x-register != 0).

Parameters: addr where to branch to.

Returns: a = x.

Description: Branch to addr if ***x-register*** is not zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note: Commonly used after disk access to branch if there were no errors.

Example:

```
.macro bxeq addr
      txa                                ; transfer x-register to a-register to set flags
      beq    addr                        ; branch if x is not zero
.endm
```

See also: **bxne**.

bxne: branch

Form: **bxne** addr gP'

Function: Branch if (x-register != 0).

Parameters: addr where to branch to.

Returns: a = x.

Description: Branch to addr if ***x-register*** is not zero. Allows relative branching forward and backward with the same limitations as normal 6502 branch instructions (+127 or -128 bytes), *addr* is a valid label, local label or offset (127 thru -128).

Note: Commonly used after disk access to branch to an error handler.

Example:

```
jsr    GetBufBlock         ; load block into the diskBlkBuf
bxne   99$                 ; is x<>0 then go to the error handler
```

```
.macro bxne addr
       txa                                ; transfer x-register to a-register to set flags
       bne    addr                        ; branch if x is not zero
.endm
```

See also: **bxeq**.

clda: flow

Form: **clda** label, data gP'

Function: Load accumulator on branch to label.

Parameters:

label	Label for branch targeting.
data	memory address to load accumulator from (or #immediate value) if branch target is used.

Description: Conditionally load accumulator with *data* if *label* is used as the destination of a branch instruction.

Note:

Example: **IsMseInMargins**.

```
       DAApp:                    ; branch (jmp/jsr/br) to here loads accumulator
             lda    zDevApp      ; from the application drive
clda   DAData,      zDevData     ; branch to here loads accumulator from data drive
clda   DAOutput,    zDevOutput   ; branch to here loads accumulator from output drive
       SafeSetD:
             cmp    curDrive     ; only set new device if selected device is a change
             bne    SfSetDev     ; from the current drive
             rts
       SfSetDev:
                    ...

       ; Example using constants.
             lda    #4                  ; if flow gets here a=4 when PointRecord called
clda   40$,         #3                  ; local labels are ok. a=3 if branch to 40$
clda   Rec2,        #2                  ; if branch or jmp/jsr to Rec2 a = 2
clda   Rec1,        #1                  ; if jmp/jsr to Rec1 a = 1
       jsr   PointRecord
       ...
```

```
.macro clda label, data
       .byte $2C                        ; $2C is opcode for an absolute bit instruction
label:                                  ; if flow goes through the bit instruction then
       lda   data                       ;    the lda command will never happen
.endm
```

See also: **cldx, cldy**.

cldx: flow

Form:	**cldx**	label, data	gP'

Function: Load x-register on branch to label.

Parameters:

label	Label for branch targeting.
data	memory address to load the x-register from (or #immediate value) if branch target is used.

Description: Conditionally load *data* into x register if *label* is used as the destination of a branching instruction.

Note:

Example:

```
             ldx    #6                  ; if flow gets here y=6 when lda diskBlkBuf,x executes
cldxI  40$,         #4                  ; local labels are ok. y=4 if branch to 40$
cldxI  Rec2,        #2                  ; if branch or jmp/jsr to Rec2 y = 2
cldxI  Rec1,        #0                  ; if jmp/jsr to Rec1 y = 1
             lda    diskBlkBuf,y
       ...
```

```
.macro cldx label, data
       .byte $2C                        ; $2C is opcode for an absolute bit instruction
label:                                  ; if flow goes through the bit instruction then
       ldx    data                      ;    the ldx command will never happen
.endm
```

See also: **clda, cldy**.

cldy: flow

Form:	**cldy**	label, data	gP'

Function: Load y-register on branch to label.

Parameters:

label	Label for branch targeting.
data	memory address to load the y-register from (or #immediate value) if branch target is used.

Description: Conditionally load *value* into y register if *label* is used as the destination of a branching instruction.

Note:

Example:

```
             ldy    #6                   ; if flow gets here y = 6 when lda (r0),y executes
cldyI  40$,         #4                   ; local labels are ok. y = 4 if branch to 40$
cldyI  Rec2,        #2                   ; if branch or jmp/jsr to Rec2 y = 2
cldyI  Rec1,        #0                   ; if jmp/jsr to Rec1 y = 1
             lda    (r0),y
       ...
```

```
.macro cldy label, data
       .byte $2C                         ; $2C is opcode for an absolute bit instruction
label:                                   ; if flow goes through the bit instruction then
       ldy    data                       ;    the ldy command will never happen
.endm
```

See also: **clda, cldx**.

CmpB: cmp

Form:	**CmpB**	source, dest	gP1

Function: test (s == d).

Parameters:

source	address of first byte to compare, or #immediate value.
dest	address of byte to compare to, or #immediate value.

Destroys: a.

Description: Compares the byte at *source* to the byte at *dest*.

Note:

Example:

```
          CmpB      #20,myVar        ; compare constant with variable
          CmpB      myVar,count      ; compare two variables
          CmpB      count,#40        ; compare variable with constant
```

```
.macro CmpB source, dest
      lda    source                    ; get source byte
      cmp    dest                      ; compare source to dest
.endm
```

See also: **CmpBI**.

CmpBI: cmp

Form:	**CmpBI** source, immed	gP1

Function: test (s == #i).

Parameters:

source	address of byte to compare.
immed	#immediate value to compare to.

Destroys: a.

Description: Compares the byte at *source* with the immediate byte *immed*.

Note: This macro is redundant with **CmpB** since **CmpB** can do immediate values too. Left in geoProgrammer' 2.1 for backwards compatibility with existing source.

Example: **ReadAndDelete**.

```
.macro CmpBI source, immed
       lda    source                    ; load source byte
       cmp    #immed                    ; compare to #immediate value
.endm
```

See also: **CmpB**.

CmpW: cmp

Form: **CmpW** source, dest gP1

Function: test (S == D).

Parameters:

source	address of first word to compare.
dest	address of word to compare to.

Destroys: a.

Description: Compares the word at *source* with the word at *dest*. Note: the high-bytes are compared first, so the condition codes (and therefore subsequent branches) are the same as for one-byte comparisons.

Note:

Example: **IsMseInMargins**.

```
.macro CmpW source, dest
      lda   source+1                 ; get high-byte of source
      cmp   dest+1                   ; compare source to dest
      bne   z                        ; if bytes are not equal then
                                     ;    done
      lda   source                   ; load low-byte
      cmp   dest                     ; compare to low-byte of #immediate value
z:
.endm
```

See also: **CmpWI**.

CmpWI: cmp

Form:	**CmpWI** source, immed	gP1

Function: test (S == #I).

Parameters:

source	address of word to compare.
immed	#immediate value to compare to.

Destroys: a.

Description: Compares the word value at *source* to the immediate word *immed*. As with **CmpW**, the condition codes (and therefore subsequent branches) are the same as for one-byte comparisons.

Note:

Example:

```
.macro CmpWI source, immed
      lda    source+1                  ; load high-byte of source
      cmp    #](immed)                 ; compare to high-byte of #immediate value
      bne    z                         ; if bytes are not equal then done
      lda    source                    ; load low-byte
      cmp    #[(immed)                 ; compare to low-byte of #immediate value
z:
.endm
```

See also: **CmpW**.

DecW: Math

Form:	**DecW** addr	gP'

Function: A = A -1.

Parameters: addr address of word to decrement.

Destroys: a.

Description: Decrement word by 1.

Note: Zero flag does not follow the result value at *addr*.

Example: **Find**.

```
.macro DecW addr
      lda    addr                      ; load low-byte
      bne    z                         ; if low-byte is zero then
      dec    addr+1                    ;    decrement high-byte
z:
      dec    addr                      ; decrement low-byte
.endm
```

See also: **IncW**.

Dialog: util

Form:	**Dialog** dbBox	gP'
Function:	Call DoDlgBox.	
Parameters:	dbBox address of dialog box to display.	
Destroys:	a, x.	
Description:	Companion macro to the atom **DoDlg**. Loads the x-register with the high-byte of *dbBox* address and the a-register with the low-byte of *dbBox* address. It then calls the **DoDlg** routine to display the dialog box.	
Note:		
Example:		

```
Dialog #dbMyDlg            ; display dialog box
lda    r0L                 ; get dialog box result
```

```
.macro Dialog dbBox
      ldx    #]dbBox                    ; load x with high-byte address of zero page pointer
      lda    #[dbBox                    ; load a with low-byte address of zero page pointer
      jsr    DoDlg                      ; activate dialog
.endm
```

See also: **DecW**.

IncW: utility

Form:	**IncW** addr	gP'
Function:	A = A + 1.	
Parameters:	addr address of word to increment.	
Destroys	.	
Description:	Increment *addr*.	
Note:	If the result is zero, then the zero flag in the status register is set.	
Example:		

```
.macro IncW addr
      inc    addr                     ; increment addr
      bne    z                        ; if result of increment is not zero then done
      inc    addr+1                   ; else increment high-byte of address
z:
.endm
```

See also: **DecW**.

jsr_a: utility

Form: **jsr_a** procedure, param gp'

Function: a=param; jsr procedure.

Parameters:

procedure	address of routine to call.
param	address of byte to load into a, or #immediate value.

Destroys: nothing.

Description: Loads the a-register with *param* and then calls *procedure*.

Note:

Example:

```
      jsr_a  SetPattern , #2

      ...

      jsr_a  SetDevice, DrvData
```

```
.macro jsr_a procedure, param
       lda    param                     ; load a-register with param to pass to procedure
       jsr    procedure                 ; call the procedure
.endm
```

See also: **jsr_x**

jsr_x: utility

Form:	**jsr_x** procedure, param	gp'

Function: x=param; jsr procedure.

Parameters:

procedure	address of routine to call.
param	address of byte to load into x, or #immediate value.

Destroys:

Description: Loads the x-register with *param* and then calls *procedure*.

Note:

Example:

```
      jsr_x  Ddec, #r11          ; Decrement zero page word in r11

      ...

      jsr_x  Dnegate, #z70       ; perform two's complement on word in z70
```

```
.macro jsr_x procedure, param
      ldx    param                    ; load x-register with param to pass to procedure
      jsr    procedure                ; call the procedure
.endm
```

See also: **jsr_a**.

LdNull: utility

Form:	**LdNull** addr	gP'

Function: D = #$0000.

Parameters: dest address of word to load with null value.

Destroys a.

Description: Load a word at *dest* with a null value ($0000).

Note:

Example: **SmartPutString**

```
      VectorAppMain:
             LdNull  appMain    ; Disable hook to appMain
             rts
```

```
.macro LdNull addr
      lda    #0                        ; load null value
      sta    addr                      ; store low-byte of dest
      sta    addr+1                    ; store it in high-byte of dest
.endm
```

See also: **LoadW**, **LdWW**

LdWW: utility

Form:	**LdWW** dest, dest2, value	gP'
Function:	D,D2 = #V.	

Parameters:

dest	address of word to load with an immediate value.
dest2	address of second word to load with an immediate value.
value	immediate word to load (constant or relocatable address).

Destroys a.

Description: Load a word at *dest* and at *dest2* with an immediate *value* or relocatable address.

Note:

Example:

```
LdWW   r0, appMain,#NULL
```

```
.macro LdWW dest, dest2, value
      lda    #[(value)                 ; load low-byte of value
      sta    dest                      ; store low-byte of dest
      sta    dest2                     ; store low-byte of dest2
      lda    #](value)                 ; load high-byte of value
      sta    dest+1                    ; store it in high-byte of dest
      sta    dest2+1                   ; store it in high-byte of dest2
.endm
```

See also: **LoadW**, **LdNull**

LoadB: utility

Form:	**LoadB** dest, value	gP1

Function: d = #v.

Parameters:

dest	address of byte to load with immediate value.
value	byte to load.

Destroys a.

Description: Loads a memory address (*dest*) with an immediate byte (*value*).

Note:

Example: **ShowBitmap**.

```
.macro LoadB dest, value
      lda   #value                     ; load value
      sta   dest                       ; store byte to dest
.endm
```

See also: **LoadW**.

LoadW: utility

Form: **LoadW** dest, value gP1

Function: D = #V.

Parameters:

dest	address of word to load with immediate value.
value	#immediate value to load.

Destroys a.

Description: Loads a memory address (*dest*) with an immediate word (*value*). A word is two bytes in length and is placed at dest and dest+1 in low-byte, high-byte order.

Note:

Example: **DisplayImage**.

```
.macro LoadW dest, value
      lda   #](value)                   ; load high-byte of value
      sta   dest+1                      ; store byte to high-byte of dest
      lda   #[(value)                   ; load low-byte of value
      sta   dest                        ; store byte to low-byte of dest
.endm
```

See also: **LoadB**.

MoveB: utility

Form:	**MoveB** source, dest	gP1

Function: d = s.

Parameters:

source	source address.
dest	destination address.

Destroys a.

Description: Moves a byte from one address (*source*) to another address (*dest*). The byte at the source address is not destroyed.

Note:

Example: **StopMenus**.

```
.macro MoveB source, dest
      lda   source                     ; load source byte
      sta   dest                       ; store it in dest
.endm
```

See also: **MoveW**.

MoveW: utility

Form:	**MoveW** source, dest	gP1

Function: D = S.

Parameters:

source	source address of word to move.
dest	destination address of word to set.

Destroys a.

Description: Moves a word (two bytes) from one address (*source*) to another address (*dest*). The word at the *source* address is not destroyed.

Note:

Example: **MseToCardPos**.

```
.macro MoveW source, dest
      lda   source+1                  ; load high-byte of source
      sta   dest+1                    ; store it to high-byte of dest
      lda   source                    ; load low-byte of source
      sta   dest                      ; store it to low-byte of dest
.endm
```

See also: **MoveXW**.

MvWW: utility

Form:	**MvWW** source, dest, dest2	gP'

Function: D,D2 = S.

Parameters:

source	source address of word to move.
dest	destination address of word to set.
dest2	destination address of second word to set.

Destroys a.

Description: Moves a word (two bytes) from one address (*source*) to address (*dest*) and address (*dest2*). The word at the *source* address is not destroyed.

Note:

Example:

```
.macro MvWW source, dest, dest2
      lda   source                    ; load low-byte of source
      sta   dest                      ; store it to low-byte of dest
      sta   dest2                     ; store it to high-byte of dest2
      lda   source+1                  ; load high-byte of source
      sta   dest+1                    ; store it to high-byte of dest
      sta   dest2+1                   ; store it to high-byte of dest2
.endm
```

See also:

PopB: utility

Form:	**PopB** dest	gP1
Function:	Pull dest byte from stack.	
Parameters:	dest where to store byte value.	
Destroys	a.	
Description:	The opposite of **PushB**; pops a byte from the stack and stores it at *dest*.	
Note:		
Example:		

```
.macro PopB dest
      pla                           ; load byte from stack
      sta   dest                    ; save byte to dest
.endm
```

See also: **PushB**.

PopW: utility

Form: **PopW** dest gP'

Function: Pull dest word from stack.

Parameters: dest where to store word value.

Destroys a.

Description: The opposite of **PushW**; pops a word (two-bytes) from the stack and stores it at *dest*. The first byte popped is the low-byte and is stored at *dest*; the second byte is the high-byte and is stored at *dest* +1.

Note:

Example:

```
      PopW   r3
```

```
.macro PopW dest
      pla                               ; load byte from stack
      sta    dest                       ; save byte to low-byte of dest
      pla                               ; load byte from stack
      sta    dest+1                     ; save it to high-byte of dest
.endm
```

See also: **PushW**.

PopX: utility

Form:	**PopX**	gP'
Function:	Pull x register from stack.	
Parameters:	none.	
Destroys	a.	
Description:	Pull accumulator from stack and store in x-register.	
Note:		
Example:		

```
.macro PopX
      pla                              ; load byte from stack
      tax                              ; transfer a into x-register
.endm
```

See also: **PushX**.

PopY: utility

Form:	**PopY**	gP'
Function:	Pull y register from stack.	
Parameters:	none.	
Description:	Pull accumulator from stack and store in y register.	
Destroys:	a.	
Note:		
Example:		

```
.macro PopY
      pla                              ; load byte from stack
      tay                              ; transfer a into y
.endm
```

See also: **PushY**.

PushB: utility

Form:	**PushB**	source	gP1

Function: Push source byte to stack.

Parameters: source address of the byte to push, or #immediate value.

Destroys: a.

Description: Pushes the byte at *source* onto the stack. *source* can be an immediate value preceded by a #-sign if desired.

Note:

Example:

```
PushB    "zpage,y"
...
PushB    #32
```

```
.macro PushB source
       lda   source                    ; load byte into a
       pha                             ; push a onto the stack
.endm
```

See also: **PushW**.

PushW: utility

Form:	**PushW** source	gP1

Function: Push word to stack.

Parameters: source address of the word to push.

Destroys: a.

Description: Pushes the word (two-bytes) at *source* onto the stack. The high-byte at *source*+1 is pushed first, followed by the low-byte at *source*.

Note:

Example:

```
.macro PushW source
      lda   source+1                ; load high-byte of word
      pha                           ; push a onto the stack
      lda   source                  ; load low-byte of word
      pha                           ; push a onto the stack
.endm
```

See also: **PopW**.

PushX: utility

Form:	**PushX**	gP'
Function:	Push x-register to stack.	
Parameters:	none.	
Destroys:	a.	
Description:	Push x-register onto the stack.	
Note:		
Example:		

```
.macro PushX
      txa                             ; transfer x-register to a-register
      pha                             ; push a-register onto the stack
.endm
```

See also: **PopX**.

PushY: utility

Form:	**PushY**	gP'
Function:	Push y register to stack.	
Parameters:	none.	
Destroys	a.	
Description:	Push y-register onto the stack.	
Note:		
Example:	**SwZp**.	

```
.macro PushY
      tya                              ; transfer y-register to a-register
      pha                              ; push a-register onto the stack
.endm
```

See also: **PopX**.

rmb: bit

Form:	**rmb** bitNumber, dest	gP1

Function: Reset bit in byte.

Parameters:

bitNumber	bit number in byte to reset (7 for MSD, 0 for LSD).
dest	address of byte which contains bit to reset.

Destroys: a.

Description: Resets (clears to 0) a bit in the byte at *dest*. *bitNumber* is a value from zero to seven, with zero being the LSB and seven being the MSB of the byte.

Note: **rmb** should only be used instead of **rmbf** if the accumulator needs to be preserved.

Example:

```
rmb    MENU_ON_BIT,mouseOn
```

```
.macro rmb bitNumber, dest
      pha                             ; save the accumulator
      lda    #[~(1 << bitNumber)      ; load bit mask
      and    dest                     ; reset selected bit
      sta    dest                     ; save modified byte
      pla                             ; restore accumulator from stack
.endm
```

See also: **rmbf**.

rmbf: bit

Form:	**rmbf** bitNumber, dest gP1
Function:	Reset bit in byte.
Parameters:	bitNumber bit number in byte to set (7 for MSD, 0 for LSD). dest address of byte which contains bit to be reset.
Destroys:	a.
Description:	Resets (clears to 0) a bit in the byte at *dest*. *bitNumber* is a value from zero to seven, with zero being the LSB, and seven being the MSB of the byte. Identical to **rmb**, except that it is faster, smaller, and destroys the a-register.
Note:	Fast version that destroys the accumulator. Use **rmb** to preserve the a-register.
Example:	**StopMenus**. `rmbf   MENU_ON_BIT,mouseOn`

```
.macro rmbf bitNumber, dest
       lda    #[~(1 << bitNumber)       ; load bit mask
       and    dest                      ; reset selected bit
       sta    dest                      ; save modified byte
.endm
```

See also:

setbit: bit

Form:	**setbit** source, mask, bits	gP'

Function: Set bits in byte.

Parameters:

source	address of byte which contains the bits to be set.
mask	address of bit mask to logical AND with *source*. (or immediate value)
bits	address of bits to logical OR with *source*. (or immediate value)

Destroys: nothing.

Description: Sets *bits* in the byte at *source* while retaining settings of other bits. Use *mask* to isolate out and retain the bit settings of bits not being affected by *bits*. The *bits* are then applied using a logical OR and are saved back to the *source*.

Example:

```
setbit  CPU_DATA, #%11111001, rCPU_DATA      ; restore bits b2-b1 with saved value

setbit  cia2pra, #%11111100, #%01            ; Put VIC bank at $8000
```

```
.macro setbit source, mask, bits
      lda   source                    ; load the source byte
      and   mask                      ; apply the mask
      ora   bits                      ; set selected bits
      sta   source                    ; save modified byte
.endm
```

See also: **smbf**.

smb: bit

Form:	**smb** bitNumber, result	gP1
Function:	Set bit in byte.	
Parameters:	bitNumber result	bit number in byte to set (7 for MSD, 0 for LSD). address of byte which contains the bit to be set.
Destroys:	nothing.	
Description:	Sets a bit in the byte at *result*. *bitNumber* is a value from zero to seven, with zero being the LSB, and seven being the MSB of the byte.	
Note:	**smb** should only be used instead of **smbf** if the accumulator needs to be preserved.	
Example:		

```
smb    MENU_ON_BIT,mouseOn
```

```
.macro smb bitNumber, result
      pha                           ; save the accumulator
      lda    #(1 << bitNumber)      ; load mask
      ora    result                 ; set selected bit
      sta    result                 ; save modified byte
      pla                           ; restore the accumulator from the stack
.endm
```

See also: **smbf**.

smbf: bit

Form:	**smbf** bitNumber, result	gP1

Function: Set bit in byte.

Parameters:

bitNumber	bit number in byte to set (7 for MSD, 0 for LSD).
result	address of byte which contains the bit to be set.

Destroys: a.

Description: Sets a bit in the byte at *result*, *bitNumber* is a value from zero to seven, with zero being the LSB, and seven being the MSB of the byte. Identical to **smb**, except that it is faster, smaller, and destroys the a-register.

Note: Fast version that destroys the accumulator. Use **smb** to preserve the accumulator.

Example: **C64Joystick**.

```
smbf   MOUSE_ON_BIT,mouseOn
```

```
.macro smbf bitNumber, result
      lda    #(1 << bitNumber)         ; load mask
      ora    result                    ; set selected bit
      sta    result                    ; save modified byte
.endm
```

See also: **smb**.

sub: math

Form:	**sub** subtrahend	gP1

Function: accumulator = accumulator – s.

Parameters: subtrahend address of byte to subtract, or #immediate value.

Destroys: a.

Description: *subtrahend* is either an address or an immediate byte value. If it is an address, the byte at the address is subtracted from the value in the `accumulator`. If it is an immediate value (preceded by a # sign), the actual value is subtracted from the `accumulator`. The result is returned in the `accumulator`. The sole purpose of the sub macro is to combine the sbc instruction with its mandatory sec instruction.

Note:

Example:

```
sub    #11
...
sub    r7L
```

```
.macro sub subtrahend
      sec                              ; set carry before starting a new subtraction
      sbc    subtrahend                ; subtract the subtrahend from the accumulator
.endm
```

See also:

SubB: math

Form:	**SubB** subtrahend, minuend.	gP1

Function: m = m – s.

Parameters:

subtrahend	address of byte to subtract, or #immediate value.
minuend	address of byte to subtract from.

Destroys: a.

Description: *subtrahend* is either an address or an immediate byte value. Subtracts the *subtrahend* from the byte at address (*minuend*) and stores the result in *minuend*.

Note:

Example:

```
SubB         r2L,r15
...
SubB         #$20,lastKey
```

```
.macro SubB subtrahend, minuend
      sec                             ; set carry before starting a new subtraction
      lda    minuend                  ; get minuend byte
      sbc    subtrahend               ; subtract the subtrahend from the minuend
      sta    minuend                  ; store difference in minuend
.endm
```

See also:

SubBS: math

Form: **SubBS** subtrahend, minuend, difference gP'

Function: d = m – s.

Parameters:

subtrahend	address of byte to subtract, or #immediate value.
minuend	address of byte to subtract from.
difference	address of byte to store the result.

Destroys: a.

Description: Subtract *subtrahend* from *minuend* and store result in *difference*.

Note:

Example:

```
SubBS  r2L,r15L,r14L       ; subtract value at r2L from r15L and save result in r14L
...
SubBS  #$20,r15L,r14L      ; subtract $20 from r15L and save result in r14L
```

```
.macro SubBS subtrahend, minuend, difference
       sec                              ; set carry before starting a new subtraction
       lda    minuend                   ; get minuend byte
       sbc    subtrahend                ; subtract the subtrahend from the minuend
       sta    difference                ; store result in difference
.endm
```

See also:

SubBW: math

Form:	**SubBW** subtrahend, minuend	gP'

Function: M = M – s.

Parameters:

subtrahend	address of byte to subtract, or #immediate value.
minuend	address of word to subtract from.

Destroys: a.

Description: Subtract *subtrahend* from *minuend* and store result in *minuend*.

Note:

Example:

```
SubBW  r0L,r1              ; subtract byte value at r0L from word value at r1
SubBW  #7,r1               ; subtract 7 from word at r1
```

```
.macro SubBW subtrahend, minuend
      lda   minuend                    ; get minuend low-byte
      sec                              ; set carry before starting a new subtraction
      sbc   subtrahend                 ; subtract the subtrahend from the minuend
      sta   minuend                    ; store result back into minuend
      bcs   z                          ; exit if no carry
      dec   minuend+1                  ; subtract 1 from high-byte
z:
.endm
```

See also:

SubBWS: math

Form: **SubBWS** subtrahend, minuend, difference gP'

Function: D = M – s.

Parameters:

subtrahend	address of byte to subtract, or #immediate value.
minuend	address of word to subtract from.
difference	address of word to store the result.

Destroys: a.

Description: Subtract *subtrahend* from *minuend* and store result in *difference*.

Note:

Example:

```
SubBWS    #7,r1L,r15L
```

```
.macro SubBWS subtrahend, minuend, difference
       sec                              ; set carry before starting a new subtraction
       lda    minuend                   ; get minuend low-byte
       sbc    subtrahend                ; subtract the subtrahend from the minuend
       sta    difference                ; store result back into minuend
       lda    minuend+1                 ; get minuend high-byte
       sbc    #0                        ; subtract with carry from minuend
       sta    difference+1              ; store result back into high-byte of minuend
.endm
```

See also:

SubVW: math

Form: **SubVW** value, minuend gP'

Function: M = M – #V.

Parameters:

value	#immediate value to subtract.
minuend	address of word to subtract from.

Destroys: a.

Description: Subtracts an immediate byte or word (*value)* from the word at *minuend* and stores the result in *minuend*.

Note: When *value* is the address of a relocatable label, use **SubVWS**.

Example:

```
SubVW     #20,rightMargin
```

```
.macro SubVW value, minuend
      sec                                ; set carry before starting a new subtraction
      lda    minuend                     ; get minuend low-byte
      sbc    #[(value)                   ; subtract the subtrahend from the minuend
      sta    minuend                     ; store result back into minuend
.if   (value >= 0) && (value <= 255)
      bcs    z                           ; exit if no carry
      dec    minuend+1                   ; subtract 1 from high-byte
z:
.else
      lda    minuend+1                   ; get minuend high-byte
      sbc    #](value)                   ; subtract subtrahend high-byte with carry from minuend
      sta    minuend +1                  ; store result in minuend
.endif
.endm
```

See also: **SubVWS**

SubVWS: math

Form:	**SubVWS** subtrahend, minuend, difference.	gP'

Function: D = M – #S.

Parameters:

subtrahend	#immediate value to subtract.
minuend	address of word to subtract from.
difference	address of word to store the result.

Destroys: a.

Description: Subtract *subtrahend* from *minuend* and store result in *difference*.

Note:

Example:

```
SubVWS    #RECSIZE,bufSize,bufLeft
```

```
.macro SubVWS subtrahend, minuend, difference
      sec                             ; set carry before starting a new subtraction
      lda    minuend                  ; get source low-byte
      sbc    #[(subtrahend)           ; subtract the subtrahend from the minuend
      sta    difference               ; store into difference
      lda    minuend+1                ; get minuend high-byte
      sbc    #](subtrahend)           ; subtract subtrahend high-byte with carry from minuend
      sta    difference+1             ; store result in difference
.endm
```

See also: **SubVW**

SubW: math

Form:	**SubW** subtrahend, minuend.	gP1
Function:	M = M – S.	
Parameters:	subtrahend address of word to subtract. minuend address of word to subtract from.	
Destroys:	a.	
Description:	Subtracts the word at *subtrahend* from the word at *minuend* and stores the result in *minuend*.	
Note:		
Example:	`SubW strSize,bufFree`	

```
.macro SubW subtrahend, minuend
      lda    minuend                    ; get low-byte of the minuend
      sec                               ; set carry before starting a new subtraction
      sbc    subtrahend                 ; subtract the subtrahend from the minuend
      sta    minuend                    ; store result back into minuend
      lda    minuend+1                  ; get minuend high-byte
      sbc    subtrahend+1               ; subtract the high-byte with carry from the subtrahend
      sta    minuend+1                  ; store result back into high-byte of minuend
.endm
```

See also:

SubWS: math

Form:	**SubWS** subtrahend, minuend, difference	gP'
Function:	D = M – S.	
Parameters:	subtrahend address of word to subtract. minuend address of word to subtract from. difference address of word to save result too.	
Destroys:	a.	
Description:	Subtract *subtrahend* from *minuend* and store result in *difference*.	
Note:		
Example:	`SubW strSize,bufSize,bufFree`	

```
.macro SubWS subtrahend, minuend, difference
      lda   minuend                   ; get low-byte of the minuend
      sec                             ; set carry before starting a new subtraction
      sbc   subtrahend                ; subtract the subtrahend from the minuend
      sta   difference                ; store result
      lda   minuend+1                 ; get minuend high-byte
      sbc   subtrahend+1              ; subtract the high-byte with carry from the subtrahend
      sta   difference+1              ; store high-byte of result
.endm
```

See also:

SubWVS: math

Form:	**SubWVS** subtrahend, minuend, difference.	gP'

Function D = #M – S.

Parameters:

subtrahend	address of word to subtract.
minuend	#immediate value to subtract from.
difference	address of word to hold result.

Destroys: a.

Description: Subtract *subtrahend* from *minuend* and store the result in *difference*.

Note:

Example:

```
SubWVS mouseXPos,#SC_PIX_WIDTH,distToEdge
```

```
.macro SubWVS subtrahend, minuend, difference
      lda    #[(minuend)              ; get low-byte of the minuend
      sec                             ; set carry before starting a new subtraction
      sbc    subtrahend               ; subtract the subtrahend from the minuend
      sta    difference               ; store result in difference
      lda    #](minuend)              ; now do the high-byte with the carry from the result
      sbc    subtrahend+1             ; of the first subtract
      sta    difference+1             ; store result back into high-byte of difference
.endm
```

See also:

tmb: bit

Form:	**tmb** bitNumber, result.	gP'
Function:	Toggle bit in byte.	
Parameters:	bitNumber	bit number in byte to set (7 for MSD, 0 for LSD).
	result	address of byte which contains the bit to toggle.
Destroys:	nothing.	
Description:	Toggle bit position *bitNumber* in *result* byte.	
Note:	**tmb** should only be used instead of **tmbf** if the accumulator needs to be preserved.	
Example:		

```
tmb          6,menuOpt
```

```
.macro tmb bitNumber, result
      pha                           ; save the accumulator
      lda    result                 ; load byte to modify
      eor    #(1 << bitNumber)      ; toggle selected bit
      sta    result                 ; save modified byte
      pla                           ; restore the accumulator
.endm
```

See also: **tmbf**.

tmbf: bit

Form:	**tmbf** bitNumber, result.	gP'
Function:	Toggle bit in byte.	
Parameters:	bitNumber	bit number in byte to set (7 for MSD, 0 for LSD).
	result	address of byte which contains bit to toggle.
Destroys:	a.	
Description:	Toggle bit position *bitNumber* in *result* byte.	
Note:	Fast version that destroys the accumulator. Use **tmb** to preserve a.	
Example:		

```
tmbf         7,myFlag
```

```
.macro tmbf bitNumber, result
      lda    result                   ; load byte to modify
      eor    #(1 << bitNumber)        ; toggle selected bit
      sta    result                   ; save modified byte
.endm
```

See also: **tmb**.

E: Memory Maps

GEOS Memory Region Map

Address	Region		Equate	Description	App Usable
`00`	**ZeroPage**	†¥		Zero Page	144
`100`	**StackPage**	†¥		6510 Stack	Var
`200`	**AppLowVar**	¥	`APP_LVAR`	low application variable space	All
`314`	Vectors			ROM Vectors when ROM is switched in	-
`334`	**AppLowRAM**	†	`APP_LRAM`	Used by GEODEBUGGER	All
`400`	AppRAM	†	`APP_RAM`	start of application space	All
`6000`	**Backscreen**	†¥	`BACK_SCR_BASE`	base of background screen	All
`7900`	PRINTBASE	†¥		load address for print drivers	All
`7F40`	AppVar	†	`APP_VAR`	application variable space	All
`8000`	**OsVars**	†¥	`OS_VARS`	OS variable base	384
`8C00`	**ColorMatrix**	†¥	**`COLOR_MATRIX`**	video color matrix	All
`9000`	DiskDrivers		`DISK_BASE`	disk driver base address	-
`A000`	**Forescreen**	†¥	`SCREEN_BASE`	base of foreground screen	7960
`BF40`	Kernal Low				-
`D000`	I/O / Kernal	†¥	**`vicbase`**	video interface chip base address	1024
`E000`	Kernal High / ROM				-

†Contains areas that are usable as application RAM.
¥Requires special consideration to use. See Memory Region Maps for more details on locations and conditions.

Zero Page

00	**CPU_DDR**	6510 data direction register.
01	**CPU_DATA**	Built-in 6510 I/O port, bit oriented.
02	**r0-r15**	GEOS Kernal zero page pseudoregisters.
22	**curPattern**	Pointer to fill pattern data.
24	**string**	Pointer to input buffer.
26	**fontTable**	Label for start of current font settings.
26	**baselineOffset**	Number of pixels from top of font to baseline.
27	**curSetWidth**	Pixel width of font bitstream in bytes.
29	**curHeight**	Card height in pixels (point size[1]) of font.
2A	**curIndexTable**	Pointer to font index table.
2C	**cardDataPntr**	Pointer to font image data.
2E	**currentMode**	Current text drawing mode.
	;--- fontTable End	
2F	**dispBufferOn**	Controls the screen to draw too. Fore/back or both.
30	**mouseOn**	Mouse/Menu/Icon control flag.
31	**msePicPtr**	Pointer to the mouse graphics data.
	;--- Text Clipping	
33	**windowTop**	Top line of window for text clipping.
34	**windowBottom**	Bottom margin, usually 199.
35	**leftMargin**	Leftmost point for writing characters.
37	**rightMargin**	The rightmost point for writing characters.
39	**pressFlag**	Input control flags.
3A	**mouseXPos**	Mouse's x-position.
3C	**mouseYPos**	Mouse's y-position.
3D	**returnAddress**	Address to return to from in-line call.
3F	**graphMode**	40 / 80-column mode flag (only in GEOS 128).
40		GEOS Kernal internal use.
70	APP_ZPL	Generically named. Application **zpage** area (A2-A9). 16-bytes.
80-FA	APP_ZIO	Swappable Kernal I/O/application **zpage** space.
(BA)	**curDevice**	Current serial device number.
FB	APP_ZPH	Generically named. Application **zpage** area (A0-A1). 4-bytes.
FF		Used by BASIC to convert floating point number to string.

[*] **Note**: 80-FA is only used by the Kernal during I/O. See **SwZp** for how to make safe use of this area in your applications.

Application Memory Available in Zero Page

70-7F FB-FE	Dedicated application space. ...	16 4
FF	This byte is only used by BASIC and is free to use from within GEOS	1
	Total bytes with no application effort.	21
80-FA	Conditionally available space. This space is used only by Kernal I/O routines. To safely use this area as application RAM, use **SwZp** to swap the area with an application buffer as needed.	123
	Total Zero Page space with logic added.	144

Stack Page

0100-01FF 6510 Hardware Stack Area.

The depth of stack usage is largely under the control of the application. It can be managed so that x% of the stack will never be used. This remaining bottom of the stack area can then be used as application space. An example of this practice is GEODEBUGGER that uses a data area starting at 0100. Knowing that the Debugger uses this area is also an important consideration if you want the application to remain compatible with GEODEBUGGER for debugging that application.

	Dedicated application Space.	0
	Total bytes with no application effort.	0
100-x	Conditionally available space. Depends on applications stack needs. Half of the stack as a data area could be safely used under normal circumstances. Careful monitoring of stack usage during design time would be required to fine tune the number to get the maximum safe amount.	0-127
	Total potential Zero Page space with logic added.	127

AppLowVar APP_LVAR

200-313

This area is unused by the C128 Kernal or the DEBUGGER and is safe for the application to use with some restrictions. The C64 CMD Kernal uses 02A1 during serial I/O and will freeze if this byte is changed to a non-zero value.

	Dedicated application space.	276
02A1	C64 ENABL. This byte cannot be altered by a C64 application	-1
	C64 Total bytes with no application effort.	275
	C128 Total bytes with no application effort.	276

Example: ramsect definition that is compatible with C64 and C128 GEOS.

```
.ramsect APP_LVAR                   ;200-2A0
                    .block 161      ;Break up the block statement with individual assignments.
                                    ;Make sure the total .block usage puts rENABL at $02A1.
        rENABL:     .block 1        ;Byte at $02A1 cannot be changed from 0 on C64 GEOS without
                                    ;freezing I/O.
                                    ;Label name is used for verifying that this byte is correctly set aside.
.ramsect APP_LVAR+$A2               ;2A2-313
                    .block 114      ;
```

AppLowRAM APP_LRAM

334-3FF

This area is completely unused by the Kernal. DEBUGGER uses this location and would not be compatible with an application that alters this area in anyway.

	Dedicated application space.	204
	Total bytes with no application effort.	0
	Total bytes with loss of ability to use the DEBUGGER.	204

BackScreen

6000-7F3F
In order to use the BackScreen as an application space you must:

1. **LoadB dispBufferOn**, #ST_WR_FORE.
2. Provide a mechanism for recovering the background behind dialog boxes. This can be either redrawing the area where the dialog was or by saving the part of the Foreground screen that the dialog uses to an application buffer. See Chapter **"Graphics Routines"**, **"Using the Background Buffer as Extra Memory"** for more information and **"Exiting from a DB"** in chapter **"Dialog Box"** for sample code.

7900-7F3F PRINTBASE

This part of the Backscreen region doubles as the load location for print driver when printing. If the application is going to be printing this area would be a temporary use only while printing is not in progress.

	Dedicated application space.	0
	Total bytes with no application effort.	0
	Total application space with added logic.	8000

OsVars

If the application is not using sprites, then the sprite images can be a data area for the application. Never use **spr0pic** as this is the mouse pointer. **spr1pic** is for the text prompt. The **spr1pic** image is created every time **InitTextPrompt** is called. So **spr1pic** is safe to use as long as the application is not using text input or is only using the **spr1pic** area as temporary space between uses of text input.

	Dedicated application Space.	0
848A	**diskOpenFlg**. This variable is only used by the desktop and can be freely used by any application for any purpose during the life of the application.	1
8A40	**spr1pic**	64
8A80	**spr2pic**	64
8AC0	**spr3pic**	64
8B00	**spr4pic**	64
8B40	**spr5pic**	64
8B80	**spr6pic**	64
8BC0	**spr7pic**	64
	Total bytes with no application effort.	449
	Total application space with added logic.	-

Example: Use all of sprite 1 through 7 area as a ramsect buffer.

```
.ramsect spr1pic          ; $8A40
       highBuf:
              .block 448
```

ColorMatrix

8C00-8FE7

C64 and C128 in 40 Col mode

There will be a visual penalty for using this area as it directly affects what the user is seeing. geoPublish uses this area during processing and accepts the visual penalties. If space is tight this can be the only last option for more room to work with. You would normally not want to use the last screen line so that a readable status line can be maintained. Post processing, the color matrix should be set back to the current FG/BG color in **screencolors**.

C128 80 Col mode

This area can be freely used but should be reset prior to exiting the application by setting the entire color matrix to the current FG/BG color in **screencolors**.

C64 & C128

	Dedicated application space.	0
8C00	**COLOR_MATRIX**	1000
	Total bytes with no application effort .	0
	C128 Total application space with added logic.	1000
	C64 Total application space with added logic.	960

Forescreen

C64 and C128 in 40 Col mode

A000-BF3F

The foreground screen can be used during processing. To hide its use, you can set the **COLOR_MATRIX** to have the same FG/BG colors for the screen area that is being used for data. Normally you would not want to use the last Card Row of the foreground screen so that a readable status line can be maintained. This approach is used by geoAssembler. Post processing, the color matrix should be set back to the current FG/BG color in **screencolors**.

C64 & C128 40-column Mode

	Dedicated application space.	0
A000	Foreground screen.	8000
	Total bytes with no application effort .	0
	Total application space with added logic.	7680

C128 80 Col mode

A040-BF7F

This area is part of the background screen. The same considerations must be made as were for the **BackScreen** region.

C128 80-column Mode

	Dedicated application space.	0
A000	Unused. Free to use by the application.	64
A040	Bottom half of background screen (Top half is at 6000).	8000
	Total bytes with no application effort .	64
	Total application space with added logic.	8064

I/O

D800-D9FF

C64

This area holds the Color Table for video modes not used by the GEOS Kernal. This area is free to be used by the application as a data area. Considerations for this region:

1. In this area only the lower nibble (b3-0) of every byte are writeable.
2. When read, the top nibble will be random values and must be masked off.
3. This area is also used by the DEBUGGER as it runs in text mode and text mode uses this color table. Note that the DEBUGGER will not allow changes to this area in interactive mode.

How useful this region may be to an application would be very application dependent.

C64

	Dedicated application space.	0
D800	Color Table for unused video modes.	1000
	Total bytes with no application effort.	0
	Total nibbles with added logic.	1000

C128

D800-D9FF

This area holds the Color Table for video modes not used by the GEOS Kernal. This area is free to be used by the application as a data area. On the 128 this area has 2 Pages that can be swapped out using the register at **CPU_DATA** ($01). Bit 0 Controls the block that is mapped into memory. 0 selects block 0 and 1 selects block 1. Bit 1 Controls which of the 2 blocks the VIC chip uses.

The DEBUGGER uses block 1 for its text colors. block 0 can therefore be used without worrying about conflicting with the DEBUGGER.

C128

	Dedicated application space.	0
D800	Color Table for unused video modes.	2000
	Total bytes with no application effort.	0
	Total nibbles with added logic and be compatible with GEODEBUGGER	1000
	Total nibbles with added logic and not be compatible with GEODEBUGGER	2000

How useful this region may be to an application would be very application dependent.

128 BackRAM:

GEOS Primary Bank is Bank 1.
BackRAM is bank 0. This allows common RAM to be turned on and have parts of bank 0 then appear into the memory space of bank 1 as shared RAM is always Bank 0 RAM and is always visible to the CPU when active.

Bank 0:
0000-00FF: Common RAM zero page ;GEOS always uses zero page from Bank 0.
0100-01FF: Common RAM stack ;GEOS always uses stack page from Bank 0.
0200-03FF: Common code area

03E4-03EB: Soft reset handler
0400-1FFF: Soft Sprites
2000-7FFF: Swap area for Desk Accessories
If your application does not use Desk Accessories this may be used as an application data area.
8000-9FFF: Unused
A000-ABFF: GEOS Kernal
AC00-C0FF Access Cache
C100-FFFF: GEOS Kernal

Bank 0 backRAM

$0000	$400	$FF00	$FF05
	BANK 0	MMU	ROM

Bank 1 GEOS Address Space

$0000	$400	$FF00	$FF05
	BANK 1 GEOS APPLICATION SPACE	MMU	ROM

Bank 2

$0000	$400	$FF00	$FF05
	BANK 2 (bank 0 if 128 is not expanded)	MMU	ROM

Bank 3

$0000	$400	$FF00	$FF05
	BANK 3 (bank 1 if 128 is not expanded)	MMU	ROM

***Note**: BANK-4 thru BANK 13 Not used by GEOS.*

Bank 14

$0000	$400	$4000	$D000	$E000	$FF00	$FF05
	RAM 0	Basic ROM	Char Rom	Kernal ROM	MMU	ROM

Bank 15

$0000	$400	$4000	$D000	$E000	$FF00	$FF05
Common RAM	RAM 0	Basic ROM	I/O	Kernal ROM	MMU	ROM

REU-BANK0

REU Address

`0000- 38FF` `0000- 78FF`	C128 **MoveData** routine C64 **MoveData** routine	C128 has a smaller area free for use with **MoveData**. If not using **MoveData** or DMA Move Data is disabled, then an application can use this area as desired. **Note**: GEODEBUGGER disables DMA moves and uses this area when the REU debugger is loaded. Using this area will make the application incompatible with the REU debugger.
`3900- 78FF`	C128 BACKRAM Kernal	REU geoDebugger will destroy this area.
`7900- 7DFF`	$8400-88FF GEOS Data	
`7E00- 82FF`	reboot code	

`8300`
`9080`
`9E00`
`AB80`
	disk driver for drive A
	disk driver for drive B
	disk driver for drive C
	disk driver for drive D
		Each disk driver is in 2 parts:
		Driver code: $C80 (3456) bytes
		dir3Head: $100 (256) bytes

`B900- BB7F`	9D80-9FFF JmpIndX+	Kernal area
`BB80- BC3F`	BF40-BFFF	Kernal tables
`BC40- BCBF`	C000-C07F	Kernal
`BCC0- CC3F`	C080-CFFF	Kernal
`CC40- FC3F`	Kernal D000-FFFF	Configuration changes can be made to the REU backup of the Kernal so they will persist through a reboot.
`FC40- FFFF`	Unused by GEOS	
	C128 only	**Note**: When the 128 DeskTop installs a print driver, it gets saved in the RAM behind I/O.
`D500- D5FF`	Print driver header block D8C0-D0BF	256 bytes
`D600- DC3F`	Print driver D9C0-DFFF	1600 bytes

Special Locations

REU Addr	Description	**Note**:
`0000-0007`	Ram check area "RAMCheck" written and read back. Used by GEOS 1.3+ and Wheels.	If any byte in bank 0:$00-$07 matches by position with "RAMCheck" a failure is raised and detection stops. This is a warm start bug that can cause bank 1 detection to fail and GEOS/Wheels will not be able to startup with only bank 0.

F: File Formats

Overview

This chapter describes the output file data formats of the Text Scrap, Photo Scrap, Notepad, geoWrite and geoPaint files. The Photo Scrap and Text Scrap files are designed so that text and graphics data can be shared between applications. This is the format used by the Photo Manager and Text manager desk accessories. Both the Text and Photo Scraps are stored as sequential system files on disk. When the user performs a cut or copy operation from inside an application, a Photo Scrap or Text Scrap file is created on the application disk. The user can then quit the present application, start up a new one and paste the contents of the Scrap file into the new application's document. Scraps can also be collected into Albums using the Photo Manager or Text Manager desk accessories. The geoWrite output format is important for programmers desiring to output geoWrite format from their programs or read geoWrite documents into their documents.

The following file formats are covered:

1. **Photo Scrap**
2. **Text Scrap**
3. **geoWrite**
4. **geoPaint**
5. **notepad**
6. **Text Album**
7. **Photo Album**

There is also a section in this Appendix detailing the **Official Fonts** that Berkeley Softworks supplied with GEOS and various other ancillary packages.

Future Releases

The Photo and Text Scrap formats have been expanded in the past to include new features and may be expanded in the future. To avoid problems, applications should check the version string in the File Header block of the Text or Photo Scrap files before using the data. Checking version strings is described in "**Chapter 9 File System**". Bytes 89 - 92 (decimal) of the File Header contain the ASCII string, V1.1, or a later version of it. Version 1.1 was the first general release format contained in any data file. If the scrap file is an older format than your application supports, it will have to be converted, something the application will probably want to provide. If the format is newer than the application, then the application should refrain from using it.

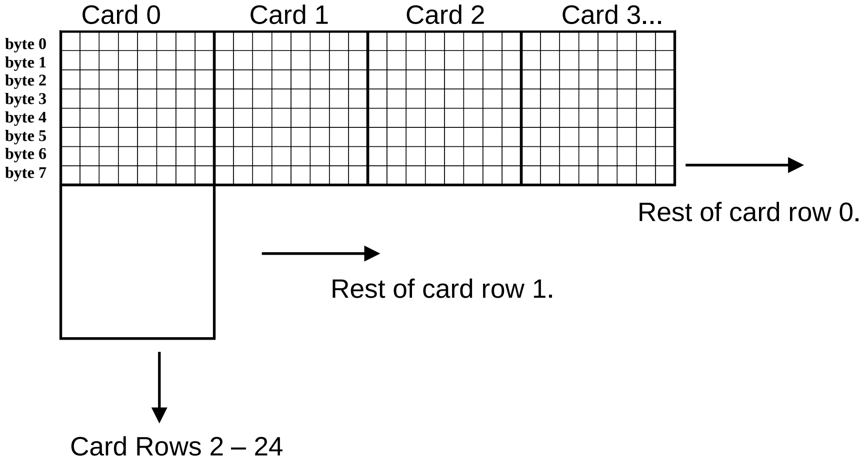
Photo Scrap

The Photo Scrap presently supports a single Bit-Mapped Object. A Bit-Mapped Object is a GEOS object for storing compacted bit-mapped data. Compacted data and Bit-Mapped Objects are described in detail in the Graphics chapter. Photo Scraps consist of a Bit-Mapped Object which may be followed by compacted Color Table for the bit-mapped area described by the Bit-Mapped Object.

In uncompacted form, the Color Table contains one byte of color information for each card generated by uncompacting the Bit-Mapped Object. The Color Table bytes are taken from the one-thousand-byte color table that normally determines the colors of each of the cards on the screen in standard high-resolution bit-mapped mode. A card, as referred to here, is the same as a Programmable Character as described in the C64 manual. The reader is referred to the description of bit-mapped graphics, cards, and color bytes starting on page 121 of Commodore 64 Programmer's Reference Guide.

In C64 hi-res bit-mapped mode, a card takes up eight bytes and defines an eight-pixel wide by eight pixel high square on the screen. Each card is associated with a byte which determines its color. For example, the first color byte in the Color Table controls the color of the upper left most card on the screen. The second color byte determines the color of the second 8x8 card which appears just to the right of the first card and so on.

A diagram of the organization of bytes in the bit-mapped mode screen is:



Byte Organization in Bit-Map Screen

Photo Scraps are not limited to the size of the screen. While most applications create scraps, which are smaller than the full screen, there will eventually be those which will construct a Scrap from an object larger than the screen size. The Color Table and bit-mapped data may be greater or less than full screen size for hi-res bit-mapped mode.

Consequently, three bytes containing the dimensions of the Bit-Mapped Object appear before the first COUNT/Bit-map pair. The first byte contains the width of the bitmap in bytes and is followed by two bytes containing the height in scanlines. Multiplying the two together gives the total number of graphic bytes to be generated by the following COUNT/Bit-map pairs. The height must always be divisible by 8 as only complete card rows are cut or copied to the Photo Scrap. The width of the scrap is always in complete cards. These restrictions are necessary because each color byte represents the color of a complete 8-byte card.

The color table is compacted using the same compaction schemes used to compact the Bitmap Object into COUNT/Bit-map pairs. Thus, even the color information appears in the Photo Scrap as a series of COUNT/Bit-mapped pairs. The Color Table COUNT/Bit-map starts just after the last graphics COUNT/Bit-map. After the proper number of graphics data bytes have been uncompacted, the next COUNT/Bit-map pair begins the compacted ColorTable. The number of data bytes divided by 8 gives you the number of ColorTable bytes to be uncompacted. The Figure below shows the structure of the Photo Scrap.

Photo Scrap Data Format

Byte Number	Contents	Purpose
0	Width	The width in bytes of the bitmap picture.
1-2	Height	The height in scanline of the bitmap picture.
3	Count	Three modes for storing bitmap data depending on Count: 0-127: use next byte COUNT times (repeat count) 128-220: use next (COUNT-128) bytes once each (straight bitmap) 221-255: use next byte as BIGCOUNT (a repeat count), repeat the following (COUNT -220) bytes BIGCOUNT times
4-end of bitmap	Bitmap Data	The bitmap data in one of the three COUNT modes
--	Count	New mode byte
-	Bitmap Data	The bitmap data in one of the three COUNT modes
-		More Count/Bitmap Pairs
-	Color Table	Color Table stored compacted. (optional) One color byte generated for each uncompacted card.

To summarize, the Photo Scrap is made up of three-dimension bytes, followed by one large compacted Bit-Mapped Object, and may be followed by a Color Table. Both the Bit-Mapped Object and the Color Table are a collection of COUNT/Bit-map pairs in different compaction formats. A COUNT/Bit-map pair consists of a format byte followed by a series of data bytes in the indicated compaction format. As described in the graphics chapter in this manual, uncompacted Bit-Mapped Object data must be reordered from scanlines to cards. The Color Table contains, in compacted form the Bit-Mapped Mode color bytes for each 8 by 8 card defined by the uncompacted Bit-Mapped Object.

Text Scrap V1.2

This section describes the V1.2 Text Scrap. The V2.0 Text Scrap is a superset of the V1.2 Text Scrap. The only addition to Text Scraps for V2.0 is a ruler escape that contains positioning information. The ruler escape is described in the next section.

The Text Scrap is an ASCII string with embedded escape characters. The escape characters are requisitioned from the nonprintable ASCII chars, sometimes called control chars¥. There are two escape chars found in Text Scraps. First is TAB (char $9). It is up to the application to support or not to support tabs as it wishes. The second escape character is given the constant name NEWCARDSET ($17). It signals the beginning of a 4-byte font/style escape string. The first two bytes after NEWCARDSET are the font ID of the font to be used to display the following text. The final byte in the string indicates the style of the following text: plain, bold, italic underline and/or outline. Each style is controlled by a bit in the style byte. Setting the bold bit, for example turns on bold face. The significance of each bit is shown below.

A complete NEWCARDSET escape string will appear whenever there is a change in either font or style. The Text Manager desk accessory will not display tabs, font and style changes but they are stored within the Text Scrap nonetheless. Applications must expect these special characters, in addition to regular ASCII characters within the Text Scrap file. The structure of the Text Scrap is shown immediately below.

Note: In ASCII the normal printable character set starts with the character '0' which has a number $20. The first 32 ($20) ASCII characters ($0 - $1F), are unprintable as they don't correspond to any letter or number like 'a' or '0'. These characters are often used to embed command strings in text.

Text Scrap

The Text Scrap file, as it appears in memory, begins with two bytes which contain the total number of bytes to follow. (Note that these bytes don't count themselves in the total). After these two count bytes follows a mandatory NEWCARDSET escape string.

The escape string is four bytes long and begins with NEWCARDSET. The next two bytes are the font ID number. The low 6 bits of this word contain the point size of the font. The upper 10 bits contain a unique number for the font. The font word is followed by a style byte in which each bit signifies a style, as shown in the table below. Setting a bit in the style byte will turn its associated function on. Clearing the bit turns the function off. All style bits reset to 0 indicates plain text printing.

Text Scrap File Format 1.2

Byte Number	Contents	Purpose		
0-1	Length	Number of bytes to follow in file.		
2	NEWCARDSET	NEWCARDSET ($17). Start of Font/Style command string.		
3-4	Font ID	The low 6 bits of font ID is the point size of the font. The upper 10 bits is the unique number of the font in which the following text should appear.		
5	Style byte	**Constant**	**Value**	**Function**
		SET_UNDERLINE	10000000	Bit 7=1: turn on underlining
		SET_BOLD	01000000	Bit 6=1: turn on bold face
		SET REVERSE	00100000	Bit 5=1: turn on reverse video
		SET ITALIC	00010000	Bit 4=1: turn on italics
		SET OUTLINE	00001000	Bit 3=1: turn on outline
	V2.0+	SET_SUPERSCRIPT	00000100	Bit 2=1: turn on superscript
	V2.0+	SET_SUBSCRIPT	00000010	Bit 1=1: turn on subscript
		SET_PLAINTEXT	00000000	All bits=0, indicates plain text
6-end	Text string	The ASCII text with embedded tabs, font/style, and if V2.0 ruler escapes.		

The remainder of the string is composed of text with embedded tabs and possibly more NEWCARDSET escape strings. There is no special character appearing as the last character in the scrap so the application must compare the number of bytes read with a total as computed from the first two bytes of the file.

To summarize, the Text Scrap begins with a length word, followed by a mandatory Font/Style change command string, and followed by ASCII chars, tabs, and possibly more Font/Style change strings. This is the V1.2 text scrap.

Version 2.0 Ruler Escape

A ruler escape was added to the V2.0 Text Scrap to maintain compatibility with geoWrite files when justification and multiple "rulers" (formatting changes) within the page were added. A ruler escape need not appear anywhere in the text scrap, but if it appears, it will appear at the beginning of the file, or at the beginning of a paragraph. Paragraphs are defined as ending with a CR, so a ruler escape will always be preceded by a CR. Ruler escapes are 27 bytes long. They contain information about the document's margins, paragraph justification, and color, if supported. The format of the V2.0 ruler escape is shown below.

Format of Ruler Escape

Byte Number	Content	Description
0	ESC RULER	ESC RULER=$11
1-2	Left Margin	Left Margin in pixel positions. Range 0-479 (639 with V2.1 data file)
3-4	Right Margin	Right Margin in pixel positions. Range: Left Margin < Right Margin <=479/639
5-6 7-8 9-10 11-12 13-14 15-16 17-18 19-20	8 Tabs tab 2 tab 3 tab 4 tab 5 tab 6 tab 7 tab 8	Each tab is one word: Bit 15: 0 for normal text tab 1 for decimal tab, decimal points aligned Bit 14-0: Tab position. Range: (> Left Margin) Tab (< Right Margin)
21-22	Paragraph Marker	How far to indent paragraphs. Range is 0 – (< Right Margin)
23	Justification	Bits for justification and line spacing Bits 7-4: 0 = Internal use. (should always be %0001) Bits 3-2: 0 = single spaced text 1 = one and a half spaced text 2 = double spaced text Bits 1-0: 0 = left justified text 1 = centered text 2 = right justified text 3 = left and right (fully) justified text
24	Text Color	The color of the text. Currently no GEOS application uses this byte
25-26	Reserved	Reserved for future use

Note: Tabs are not displayed in the Text Manager even though they appear in the ruler data in the file. In applications that use tabs, the tab character causes spacing to the position of the next tab, if set. A wrap to the beginning of the next line is done if no tab is defined in the currently active ruler to the right of the position of the embedded tab character.

geoWrite

There are currently 2 generations of geoWrite. 1.x and 2.x. 2.x added the following abilities.

1. Superscript and subscript
2. Headers and footers
3. Ruler changes:
    - a. Paragraph marker
    - b. Decimal tabs
    - **c. Justification**
    - d. Multiple rulers per page
4. V2.x File header block added the following:
    - a. Starting page number
    - b. Title page
    - c. Variable page height

Output File Formats

Like the Text Scrap, there is a V1.1 and a V2.0/2.1 geoWrite output format. The version numbers are different for the output file formats and the program releases. You will find geoWrite with version strings of V1.2, V1.3, and V2.0 for the Writer's Workshop, while the output file formats are either V1.1 or V2.0. V2.1 of geoWrite arrived with GEOS 2.0.

In both formats, documents are stored in VLIR files. In general, each record in the VLIR file stores one page of text. Some records are used to store pictures and, in the case of V2.0 files, header and footer information. This arrangement is show below.

VLIR Format for geoWrite Files

Record #	V1.1 Format Files	V2.0/2.1 Format Files
0-60	Text pages	Text pages
61	Text page	Header, empty for none
62	Text page	Footer, empty for none
63	Text page	Reserved
64-127	Pictures in **BitmapUp** format	Pictures in **BitmapUp** format

The major difference between the V1.1 and V2.0 formats is that the Writer's Workshop V2.0 version supports headers and footers. Pages 61-63 may be used to store text pages with the earlier releases of geoWrite, but these will not be carried over when editing with the geoWrite V2.0. This is probably not a problem since no one has ever gotten close to actually being able to store a 64-page document on a 1541 disk. When double sided support for the 1571 becomes available this may become possible.

In geoWrite, each document is broken up into separate pages and each page stored in its own VLIR record. A page consists of ruler information followed by text. For a V1.1 geoWrite file the ruler data consists of right and left-margin and tab data.

The text that follows is stored as ASCII. Escape strings are used for font/style changes and for including pictures. The data for each picture is stored in a separate record. All non-empty pages must start with a font/style escape. A font/style escape cannot be followed immediately by another font/style escape, geoWrite files may also include pictures with an ESC_GRAPHICS. The data for the picture is stored in its own record as a bit-mapped object. See the graphics section for the format of a bit-mapped object.

Graphics Escape String

Byte	Function	Description
0	ESC_GRAPHICS	The escape to graphics control char = $10
1	Width	Picture's width in cards
2 - 3	Height	Picture's height in scanlines
5	Record Number	Number of the record containing the picture data The picture data is a photo scrap

geoWrite V1.x

Early Versions of GeoWrite have a fixed ruler that only appears at the start of every page. **Note**: 1.x does not have any information stored in its file header block.

geoWrite V1.x Page Layout

Byte	Description
0-19	Ruler
0-1 2-3 4-20	Left Margin Range 0 – 479 Right Margin in pixel positions. Range: (> Left Margin) and (<= 479) 8 Tabs. Range (> Left Margin) Tab (< Right Margin)
21-24	NEWCARDSET = ($17) font/style escape
25- ...	Text of document, may contain ruler, font/style, graphics, or page break escapes PAGE_BREAK = $0C, causes geoWrite to begin a new page ESC_GRAPHICS = $10, includes a picture
Last byte	EOF = 0 appears as last byte of document.

Sample Ruler in geoProgrammer format.

```
T_RulerV1:
       .word  0                   ; left margin
       .word  479                 ; right margin
       .word  72                  ; tabs 1-8
       .word  112
       .word  184
       .word  224
       .word  296
       .word  336
       .word  479                 ; Unused tab
       .word  479                 ; Unused tab

T_CardSet:
       .byte  NEWCARDSET          ; font set
       .word  BSW                 ; $0009 font ID 0. 9 point font
       .byte  SET_PLAINTEXT
```

geoWrite V2.0

Version 2.0 is similar to V1.2 but includes a more extensive ruler escape. This is the same format as found in Text Scrap files. The file format for V2.0+ data files is as follows.

geoWrite V2.0+ Page Layout

Offset	Description
0 - 26	Ruler escape string
27 – 30	NEWCARDSET = ($17) font/style escape
31 - ...	Text of document, may contain ruler, font/style, graphics, or page break escapes. PAGE_BREAK = $0C, Causes geoWrite to begin a new page. ESC_GRAPHICS = $10, includes a picture
Last byte	EOF = 0 appears as last byte of document.

Further information is also stored in the file header of V2.0 files. This information includes the height of the footer and header, the page height the document was formatted with (different depending on the selected printer driver), and flags for NLQ and title page modes.

geoWrite V2.0+ File Header Information

Offset	Contents	Description
$89	Page Number	Page number to print on first page of this file, need not be 1.
$8B	Title and NLQ	Bit 7 set = make title page (no header, footer on first page) Bit 6 set = turn NLQ fixed width spacing on.
$8C	Header Height	The height in pixels reserved on each page for the header.
$8E	Footer Height	The height in pixels reserved on each page for the footer.
$90	Page Height	Different printers support different vertical resolutions. If the height of the page as stored here does not match what the printer is capable of, then geoWrite 2.0 reformats the file to match the printer.

Sample V2.0/2.1 Ruler

```
T_RulerV2.0:
       .byte  ESC_RULER          ; $11
       .word  0                  ; left margin
       .word  480                ; right margin
       .word  40                 ; tab 1
       .word  96
       .word  152                ; tab 3
       .word  208
       .word  264
       .word  320
       .word  376
       .word  432                ; tab 8
       .word  8                  ; paragraph marker
       .byte  %00010000          ; justification
       .byte  NULL               ; text color (not implemented)
       .word  NULL               ; reserved

T_CardSet:
       .byte  NEWCARDSET
       .word  BSW                ; $0009 font ID 0. 9 point font
       .byte  SET_PLAINTEXT
```

geoWrite Tab Stops

Tab stops in geoWrite are set at .1" resolution. Each .1" translates to 8 dots when printed, and 8 pixels on the display. The V1.1 Ruler starts at 1.2" and ends at 7.2" giving a print area of 7" using 560 dots. Margins, paragraph and tab stops are all offsets of 1.2". V2.1 starts at .2" and ends at 8.2" with all stops being an offset from .2"

V1.1/2.0 Ruler on a 40 column screen

Stop Type	Inches	offset	in hex	
Left Margin	1.2"	0	$0000	First available tab position is at 1.3" (Tabs must be inside the margin markers)
Paragraph	1.5"	24	$0018	(Paragraph marker not available in V1.1)
Tab1	2.0"	64	$0040	
Tab2	3.0"	144	$0090	Each tab is 80 dots/pixels apart
...				
Right Margin	7.2"	560	$0230	Last possible tab is at 7.1" which is .1" left of the right margin

V2.1 Ruler on an 80 column screen

Stop Type	Inches	offset	in hex	
Left Margin	.02"	0	$0000	First available tab position is at 1.3" (Cannot have a tab before or on the left margin marker)
Paragraph	.5"	24	$0018	Each .1" is 8 dots/pixels apart
Tab1	1.0"	64	$0040	
Tab2	2.0"	144	$0090	
...				
Tab8	7.9"	616	$0268	Tabs must be inside the margins
Right Margin	8.0"	624	$0270	Right margin set .2" in from the hard right margin of 8.2"

geoWrite Summary

geoWrite files are divided into pages stored in different records of a VLIR file. These records may also contain bitmap data for pictures included in the document. In addition the V2.0 format includes header, footer, and page height as well as justification, NLQ and title page flags. In V1.1 files, there is only one small ruler at the top of the page. A different ruler may control each paragraph in V2.0 files.

The above information should be sufficient to enable programmers to read and to create files in any of the formats. It is important to note that each of the earlier versions of output file formats are subsets of the later versions. Thus the V1.1 Text Scrap is a subset of the V2.0 and can be read by the later version Text Manager. The only possible incompatibility between formats is the ability of V1.1 geoWrite to store text pages in the header, footer, and reserved records. As mentioned above, it is unlikely that a 64-page document will fit on one disk.

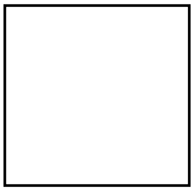
Text Scraps and geoWrite files differ in that Text Scraps are meant to be only one page or less. The Text Scrap is designed to be a more generic object, enabling a common ground between word processors.

geoPaint

As of the latest version of geoPaint V2.0 there is only one version of geoPaint data files: V1.1. Each geoPaint file is comprised of an image that is 640x720 pixels. This image is organized in 8x8 cards, which forms a matrix of 80x90 cards. With one foreground / background color card for each 8x8 image card.

Sample image card

```
T_ImageCard:
       .byte  %11111111
       .byte  %10000001
       .byte  %10000001
       .byte  %10000001
       .byte  %10000001
       .byte  %10000001
       .byte  %10000001
       .byte  %11111111
```



Sample Color card

```
T_ColorCard:               ; dark grey foreground, light grey background
       .byte  (DKGREY <<4) | LTGREY
```

Output File Format

Like geoWrite documents, geoPaint images are stored in VLIR files. The geoPaint image is divided up into 45 different VLIR records. Each record in the VLIR file stores two card rows of image data and two rows of color cards. It takes 45 records to store the entire 90 card rows of the image. This simple arrangement is shown below.

VLIR Format for geoPaint Files

Record #	V1.1 Format Files
0-44	Card Row Sets

VLIR Records

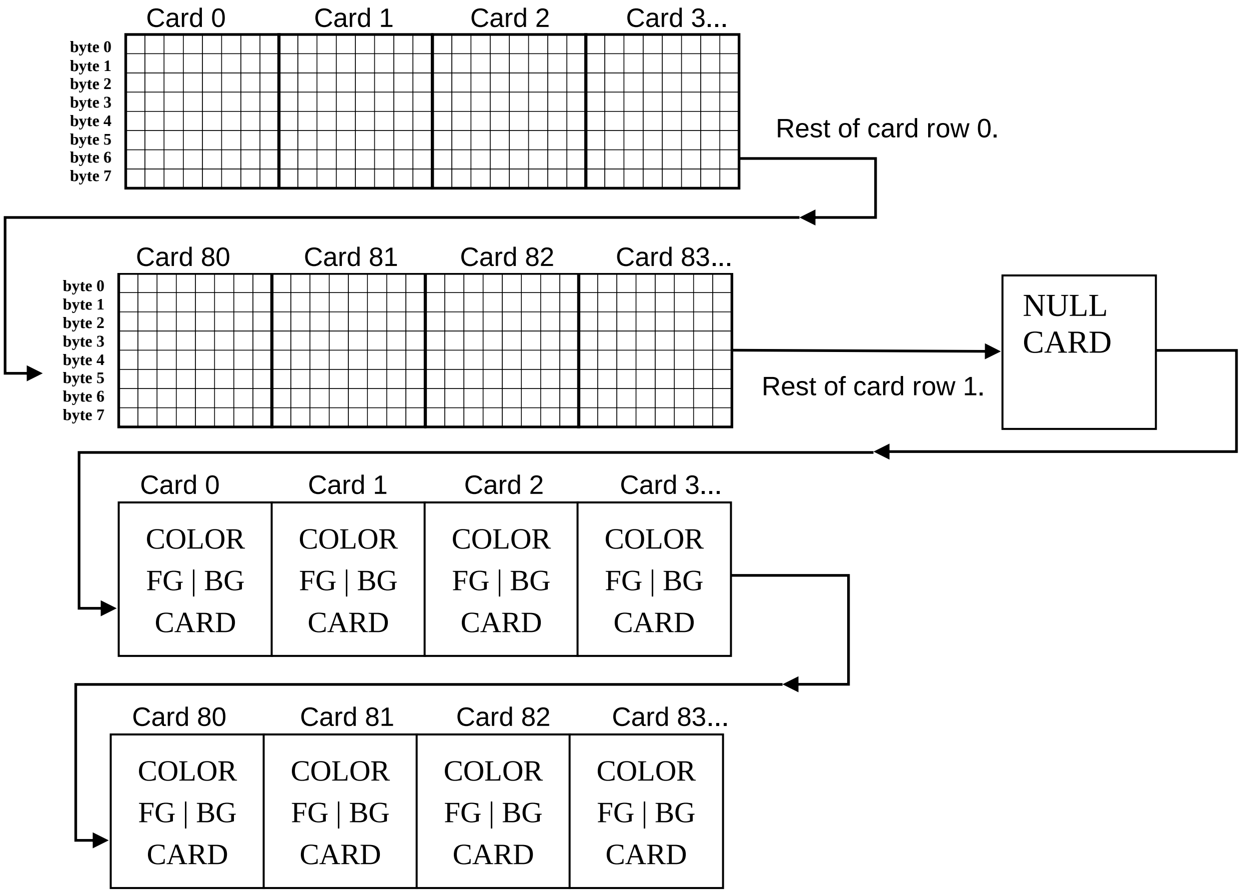
Each VLIR record contains a card row set that contains two rows of image cards, one NULL card and the color cards for the two rows of image cards. A NULL is saved after the card row set in the VLIR record.

Card row set

Count	Contents	Size in Bytes
2	80-column wide set of image cards. (80-columns * 8 *card height*) * 2 *rows*.	1280
1	Null terminating Card (1 * 8 *card height*)	8
2	80-column wide sets of Color cards. (80-columns * 1 *color card size* * 2 *rows*)	160
	Total bytes to be compressed	1448

Card Row Set

The card row set is processed as one continuous stream of bytes. Example byte stream from VLIR Record 0:

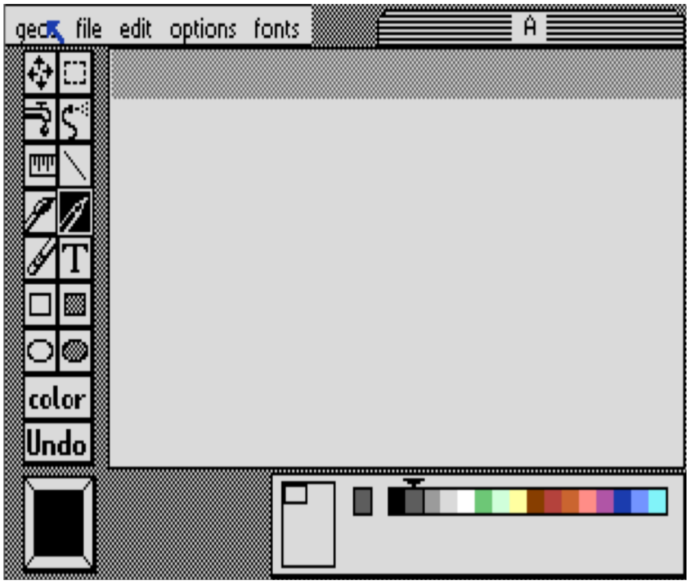


Byte stream Compression

geoPaint Card Row Set Format

Offset	Contents	Purpose
0	CMD	Compression Command
		Three modes for storing bitmap data depending on count:
	CMD = 0-63:	COUNT=CMD. Use next COUNT bytes. (Uncompressed Data)
	CMD = 65-127:	COUNT=CMD-64 Repeat the next card COUNT times.
	CMD = 129-255:	COUNT=CMD-128 Repeat the next byte COUNT times.
0-end of stream	Bitmap data	The bitmap data in one of the three COUNT modes
--	Count	New mode byte
-	Bitmap data	The bitmap data in one of the three COUNT modes
-		More count/bitmap Pairs
-	Color table	Color table stored compacted. One color byte generated for each uncompacted card.

Sample Compression



VLIR Record 0 contains the two card rows of pattern 2 that were drawn on the image above. This pattern started at column 1 and continued until the right-edge of the image.

```
   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
0 00 3F 07 2A 55 2A 55 2A  55 2A 7F 55 AA 55 AA 55
1 AA 55 AA 50 55 AA 55 AA  55 AA 55 AA 08 55 2A 55
2 2A 55 2A 55 2A 7F 55 AA  55 AA 55 AA 55 AA 50 55
3 AA 55 AA 55 AA 55 AA 01  55 88 00 FF BF A1 BF 00
```

Record 0 Decompression

CMD	Description	Count	Data	Byte Count
7	String	7	2A:55:2A:55:2A:55:2A	7
127	Repeat Card	63	[55:AA:55:AA:55:AA:55:AA]	504
80	Repeat Card	16	[55:AA:55:AA:55:AA:55:AA]	128
8	String	8	55:AA:55:AA:55:AA:55:2A	8
127	Repeat Card	63	[55:AA:55:AA:55:AA:55:AA]	504
80	Repeat Card	16	[55:AA:55:AA:55:AA:55:AA]	128
1	String	1	55	1
136	Repeat byte	8	0	8
255	Repeat byte	127	BF	127
161	Repeat byte	33	BF	33
			Total Decompressed bytes	1448

geoPaint Summary

geoPaint files contain a single 640x720 image that is spread across 45 records of a VLIR file. The format is used across all versions of geoPaint for both 64 and 128 GEOS.

notepad

The Notes data file created by notepad only has 1 version: "Notes   V1.0". The data file is a very simple VLIR file. Each page of the Notes file is stored in its own VLIR record. This limits the total number of pages to the standard VLIR limit of 127 records.

Each page of a Notes file contains a simple NULL terminated string with the CR being the only supported control character. There is no support for fonts / tabs / styles etc...

A page is limited in size to 1 disk block, which gives the page a max data size of BLKDATSIZE (254) including the null terminator.

Text Album

1.0

The 1.0 file format is used by all versions of text manager prior to V2.1. This format is a simple VLIR structure with every page in the Album being a VLIR Record with a v1.2 Text Scrap. The Album can have a maximum of 60 Text Scraps.

2.1

The 2.1 file format adds two new features over 1.0.

1. It now can contain v2.0 Text Scraps so it now supports ruler escapes.
2. The ability to name each page in the Album.

The page name table is stored in the last VLIR record. Every time a page is added or removed from the album, or a page name changes, this record is deleted and rewritten with the new contents. Note that the VLIR records are always kept together without gaps. If you have a 2-page album the pages will be stored in record 0 and record 1, with the page name table stored in record 2. If you add a new page now, it would be stored in record 2 and the page name table record will become the new last record at record 3.

The page name table has the following format:

Page Name Table

Offset	Contents	Size in Bytes
0	Number of pages in the Album.	2
1	Page 1 Name. 16-character null terminate.	17
†18	Page 2 Name (If present).	17
†35	Page 3 Name (If present).	17
	...	
†**xx**	NULL record table terminator	17

Example:

Byte Stream in Record 2, in an album with two pages:

This album has 2 pages the first page is not named and the second page is named. For every page there is a 16-character field to hold the name plus a NULL-terminator. This list of page names is terminated with a terminating 17-character field of all NULLs.

```
02                                                    number of pages
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    (Page Was not named)
4D 79 20 50 61 67 65 3A 00 00 00 00 00 00 00 00 00    My Page:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    NULL Table Terminator
```

Note: † The page name table always has one 17-byte name field for each page in the album. After all of the page name fields there is another 17 bytes of NULL to end the table.

Photo Album

1.0

The 1.0 file format is used by all versions of photo manager prior to V2.1 of the photo manager. This format is a simple VLIR structure with every page in the album being a VLIR record containing a photo Scrap.

2.1

The 2.1 file format adds the ability to name each page in the Album. The page names are stored in the last VLIR record. Every time a page is added or removed from the album, or a page name changes, this record is deleted and rewritten with the new contents. VLIR Records are always kept together without gaps. If you have a two page album the pages will be stored in record 0 and record 1 with the page names stored in record 2. If you add a new page, that page would be stored in record 2 and the page name record will become the new last record at record 3.

The page name record has the following format:

Page Name Table

Offset	Contents	Size in Bytes
0	Number of pages in the Album.	2
1	Page 1 Name. 16-character null terminate.	17
†18	Page 2 Name. (If present).	17
†35	Page 3 Name. (If present).	17
	...	
†**xx**	NULL record table terminator	17

Example:
Byte stream in Record 3, in an album with three pages, and the second and third pages are named:

This record starts with the number of pages, followed by three 17-character fields of page names and 1 terminating 17-character field of all NULLs.

```
03                                                  number of pages
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (page Was not named)
4D 79 20 50 61 67 65 3A 00 00 00 00 00 00 00 00 00  My Page:
49 63 6F 6E 73 20 66 6F 72 20 44 6C 67 00 00 00 00  Icons for Dlg
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  NULL table terminator
```

Note: † The page name table always has one 17-byte name field for each page in the album. After all of the page name fields there is another 17 bytes of NULL to end the table.

Official Fonts

The table on the next page contains the presently supported GEOS fonts. The geoLaser fonts are designed to look as closely as possible to the fonts inside an Apple LaserWriter®. When preparing documents to be laser printed, these fonts should be used.

GEOS Fonts

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
BSW	0	9	0009			40-Col System Font
BSW128	128	9	2009			80-Col System Font
University	1	†6 10 12 14 18 24	0046 004A 004C 004E 0052 0058	4/7/86 12:00 PM 3/7/86 3:00 PM	University V1.1 University V1.0	GEOS 1.1+ GEOS 1.0
California	2	10 12 *13 14 18	008A 008C 008D 008E 0092	9/5/88 7:23 PM 4/7/86 12:00 PM 3/7/86 3:00 PM	GeoFont 1.4 California V1.1 California V1.0	GEOS 1.5+ (Adds 13 Point) GEOS 1.1+ GEOS 1.0 *Only in newest version.
Roma	3	9 12 18 24	00C9 00CC 00D2 00D6	4/7/86 12:00 PM 3/7/86 3:00 PM	Roma V1.1 Roma V1.0	GEOS 1.1+ GEOS 1.0
Dwinelle	4	18	0112	4/7/86 12:00 PM 3/7/86 3:00 PM	Dwinelle V1.1 Dwinelle V1.0	GEOS 1.1+ GEOS 1.0
Cory	5	12 13	014C 014D	4/7/86 12:00 PM 3/7/86 3:00 PM	Cory V1.1 Cory V1.0	GEOS 1.1+ GEOS 1.0

†*deskTop uses a copy of University 6 point for displaying filenames.*

FONTPACK1

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Tolman	6	12 24	018C 0198	4/7/86 12:00 PM	Tolman V1.1	¥
Bubble	7	24	01D8	4/7/86 12:00 PM	Bubble V1.1	¥
FontKnox	8	24	0218	4/7/86 12:00 PM	FontKnox V1.1	†¥
Harmon	9	10 20	024A 0254	4/7/86 12:00 PM	Harmon V1.1	¥
Mykonos	10	12 24	028c 0298	4/7/86 12:00 PM	Mykonos V1.1	†¥
Boalt	11	12 24	02CC 02D8	4/7/86 12:00 PM	Boalt V1.1	†¥
Stadium	12	12	0230	4/7/86 12:00 PM	Stadium V1.1	†¥
Tilden	13	12 24	030C 034C	4/7/86 12:00 PM	Tilden V1.1	¥
Evans	14	18	0392	4/7/86 12:00 PM	Evans V1.1	¥
Durant	15	10 12 18 24	03CA 03CC 03D2 03D8	4/7/86 12:00 PM	Durant V1.1	†¥
Telegraph	16	18	0412	4/7/86 12:00 PM	Telegraph V1.1	†¥
Superb	17	24	0458	4/7/86 12:00 PM	Superb V1.1	†¥
Bowditch	18	12 24	048C 0498	4/7/86 12:00 PM	Bowditch V1.1	¥
Ormond	19	12 24	04CC 04D8	4/7/86 12:00 PM	Ormand V1.1	The file on the FONTPACK1 disk is misspelled as Ormand. †¥
Elmwood	20	18 36	0512 0524	4/7/86 12:00 PM	Elmwood V1.1	¥
Hearst	21	10 12 18 24	054A 054C 0552 0558	4/7/86 12:00 PM	Hearst V1.1	¥

Brennens	22	18	0592	9/5/86 3:11 PM 4/7/86 12:00 PM	Brennens V1.1 Brennens V1.1	Updated in Plus FONTPACK1 [†¥]
Channing	23	14 16 24	05CE 05DO 05D8	4/7/86 12:00 PM	Channing V1.1	[¥]
Putnam	24	12 24	060C 0618	4/7/86 12:00 PM	Putnam V1.1	[¥]
LeConte	25	12 18	064C 0652	4/7/86 12:00 PM	LeConte V1.1	[†¥]

[†] *font also appears in FONTPACK Plus*

[¥] *font also appears in International FONTPACK*

geoLaser Fonts

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Commodore	26	10	068A	7/1/86 10:17 AM	Commodore V1.1	Used to Represent NLQ mode output
LW_Roma	27	9 10 12 14 18 24	06C9 06CA 06CC 06CE 06D2 06D8	9/9/87 8:14 AM	GeoFont 1.4	Times Roman geoPublish / GEOS OS DISK 3
LW_Cal	28	9 10 12 14 18 24	0709 070A 070C 070E 0712 0718	9/9/87 8:15 AM	GeoFont 1.4	Helvetica geoPublish / GEOS OS DISK 3
LW_Greek	29	9 10 12 14 18 24	0749 074A 074C 074E 0752 0758	9/27/87 7:00 PM 9/9/87 18:16 AM	LW_Greek V1.1 GeoFont 1.4	Symbol geoPublish / GEOS OS DISK 3
LW_Barrows	30	9 10 12 14 18 24	0789 078A 07BC 078E 0792 0798	9/27/87 7:00 PM 9/22/87 7:00 PM 4/7/86 12:00 PM	LW_Barrows V1.1	Courier geoPublish / GEOS OS DISK 3

GEOS LaserWriter *Plus* Fonts

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
LW_Giannini	31	10 12 14 18 24	07CA 07CC 07CE 07D2 07D8	11/19/87 8:07 AM	LW_PlusA V1.1	ITC Avant Garde
LW_Bacon	32	10 12 14 18 24	0810 0812 0814 0818 081E	11/19/87 8:08 AM	LW_PlusB V1.1	ITC Bookman
LW_Haviland	33	10 12 14 18 24	084A 084C 084E 0852 0858	11/19/87 8:08 AM	LW_PlusC V1.1	Helvetica Narrow

GEOS LaserWriter *Plus* Fonts

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
LW_Piedmont	34	10 12 14 18 24	088A 088C 088E 0892 0898	11/19/87 8:08 AM	LW_PlusD V1.1	New Century Schoolbook
LW_Cowell	35	10 12 14 18 24	08CA 08CC 08CE 08D2 08D8	11/19/87 8:09 AM	LW_PlusE V1.1	Palatino
LW_Galey	36	10 12 14 18 24	090A 090C 090E 0912 0918	11/19/87 8:09 AM	LW_PlusF V1.1	Zapf Chancery GeoWorld's †**LW_Zapf** font can be substituted for better onscreen representation.
LW_Shattuck	37	10 12 14 18 24	094A 094C 094E 0952 0958	11/19/87 8:09 AM	LW_PlusG V1.1	Zapf Dingbats

†*See* ***Laser Printing Note*** *for more information*

International FONTPACK

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Roma_SP	41	9 12 18 24	0A49 0A4C 0A52 0A58	11/12/87 8:44 AM	Roma_SP V1.1	
University_FR	43	6 10 12 14 18 24	0AC6 0ACA 0ACC 0ACE 0AD2 0AD8	11/6/87 10:14 AM	University_FV1.1	
Roma_FR	45	9 12 18 24	0B49 0B4C 0B52 0B58	11/6/87 10:14 AM	Roma_FR V1.1	
University_IT	53	6 10 12 14 18 24	0D46 0D4A 0D4C 0D4E 0D52 0D58	11/11/87 4:04 PM	GeoFont 1.4	
Roma_IT	55	9 12 18 24	0DC9 0DCC 0DD2 0DD8	11/6/87 10:21 AM	Roma_IT V1.1	
University_GE	56	6 10 12 14 18 24	0E06 0E0A 0E0C 0E0E 0E12 0E18	10/21/86 1:22 PM	GeoFont 1.4	
California_GE	*57*					*In GERMAN GEOS 2.5*
FONTPACK 1	*International FONTPACK includes all fonts from FONTPACK1 (with Ormond being spelled correctly)*					

International FONTPACK

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
University_SW	58	6	E86	11/12/87 1:52 PM	University_SV1.1	
		10	E8A			
		12	E8C			
		14	E8E			
		18	E92			
		24	E98			
Roma_SW	60	9	F09	11/12/87 1:51 PM	Roma_SW V1.1	
		12	F0C			
		18	F12			
		24	F18			
University_SP	63	6	FC6	11/12/87 8:42 AM	GeoFont 1.4	
		10	FCA			
		12	FCC			
		14	FCE			
		18	FD2			
		24	FD8			
Roma_GE	64	9	1009	10/28/87 10:27 AM	GeoFont 1.4	
		12	100C			
		18	1012			
		24	1018			
Dwinelle_GE	*65*		*see*	*GERMAN GEOS 2.5*		*In GERMAN GEOS 2.5*
Cory_GE	*66*			*for details*		*In GERMAN GEOS 2.5*
University_UK	68	6	1106	10/12/87 5:43 PM	University_UV1.1	
		10	110A			
		12	110C			
		14	110E			
		18	1112			
		24	1118			
California_UK	69	10	114A	10/28/87 3:05 PM	GeoFont 1.4	
		12	114C			
		14	114E			
		18	1152			
Roma_UK	70	9	1189	10/28/87 2:49 PM	GeoFont 1.4	
		12	118C			
		18	1192			
		24	1198			
Dwinelle_UK	71	18	11D2	10/12/87 2:13 PM	Dwinelle_UK V1.1	
Cory_UK	72	12	120C	10/12/87 2:13 PM	Cory_UK V1.1	
		24	1218			
University_DA	78	6	1386	10/21/86 1:22 PM	GeoFont 2.0	
		10	138A			
		12	138C			
		14	138E			
		18	1392			
		24	1398			
Roma_DA	80	9	1409	3/23/92 1:00 AM	GeoFont 2.0	
		12	140C			
		18	1412			
		24	1418			
University_SE	83	6	14C6	10/21/86 1:10 PM	GeoFont 2.0	
		10	14CA			
		12	14CC			
		14	14CE			
		18	14D2			
		24	14D8			

International FONTPACK

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Roma_SE	85	9 12 18 24	1549 154C 1552 1558	11/6/87 10:24 AM	Roma_SE V1.1	

German GEOS 2.5

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
University_GE	56	6 10 12 14 18 24	E06 E0A E0C E0E E12 E18	10/21/86 1:22 PM	GeoFont 1.4	Also in International FONTPACK
California_GE	57	10 12 14 18	E4A E4C E4E E52	11/3/87 11:28 AM	GeoFont 1.4	
Roma_GE	64	9 12 18 24	1009 100C 1012 1018	10/28/87 10:27 AM	GeoFont 1.4	Also in International FONTPACK
Dwinelle_GE	65	18	1052	10/21/86 1:23 PM	GeoFont 1.4	
Cory_GE	66	12	108C	10/21/86 1:23 PM	GeoFont 1.4	
Commodore_GE	129	10	204A	11/3/87 11:28 AM	GeoFont 1.4	

Mega Fonts

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Mega Roma	155	48	068A	9/27/87 7:00 PM 9/22/87 7:00 PM	GeoFont 1.4	geoPublish
Mega Cal	156	48	06C9	9/27/87 7:00 PM 9/22/87 7:00 PM	GeoFont 1.4	geoPublish
Mega Barrows	158	48	0709	9/27/87 7:00 PM 9/22/87 7:00 PM	GeoFont 1.4	geoPublish

FONTPACK *Plus*

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Stern	200	24	3218	10/20/87 8:15 AM	GeoFont 1.4	
North Gate	201	18	3252	10/20/87 8:16 AM	GeoFont 1.4	
Haste	202	12 18 24	328C 3292 3298	10/20/87 8:17 AM	GeoFont 1.4	
Bancroft	203	12 18 24	32CC 32D2 32D8	10/20/87 8:17 AM	GeoFont 1.4	
Solano	204	24	3318	10/20/87 8:17 AM	GeoFont 1.4	
Barrington	205	16 32	3350 3360	10/27/87 9:56 AM	GeoFont 1.4	
Braille	206	12 24	338C 3398	10/20/87 8:18 AM	GeoFont 1.4	
Eshlemen	207	28	33DC	10/20/87 8:19 AM	GeoFont 1.4	
Ashby	208	23	3417	10/20/87 8:20 AM	GeoFont 1.4	
Spats	209	24	3458	10/20/87 8:21 AM	GeoFont 1.4	
Kensington	210	27	349B	10/20/87 9:07 AM	GeoFont 1.4	
Flints	211	12 24	34CC 34D8	10/20/87 9:15 AM	GeoFont 1.4	

FONTPACK *Plus*

Font Name	Number	Point Sizes	ID	Date	Permanent Name	Notes
Derby	212	9	3509	10/20/87 9:16 AM	GeoFont 1.4	
		18	3512			
Oxford	213	14	354E	10/27/87 11:37 AM	GeoFont 1.4	
		28	355C			
Lewis	214	18	3592	10/20/87 9:18 AM	GeoFont 1.4	
		26	359A			
		34	35A2			
Fog	215	24	35D8	10/20/87 9:16 AM	GeoFont 1.4	
Latimer	216	18	3612	10/20/87 9:19 AM	GeoFont 1.4	
McLaughlin	217	18	3652	10/20/87 9:20 AM	GeoFont 1.4	
Cursive	218	17	3691	10/27/87 10:19 AM	GeoFont 1.4	
		29	369D			
Sather	219	20	36D4	10/20/87 10:24 AM	GeoFont 1.4	
Zellerbach	220	13	370D	10/20/87 10:25 AM	GeoFont 1.4	
		18	3712			
Wurster	221	25	3759	10/20/87 10:25 AM	GeoFont 1.4	
Sproul	222	24	3798	10/20/87 10:26 AM	GeoFont 1.4	
Birge	223	18	37D2	10/20/87 10:26 AM	GeoFont 1.4	
Dana	224	13	380D	8/22/88 1:02 AM	GeoFont 1.4	
		24	3818			
Dwight	225	18	3852	10/20/87 10:26 AM	GeoFont 1.4	
Venetian	226	24	3898	10/20/87 10:29 AM	GeoFont 1.4	
Mulford	227	12	38CC	10/20/87 10:29 AM	GeoFont 1.4	
Bowles	228	24	3918	10/20/87 10:48 AM	GeoFont 1.4	
Euclid	229	36	3964	10/20/87 10:31 AM	GeoFont 1.4	
Grizzly	230	18	3992	10/20/87 10:31 AM	GeoFont 1.4	
Kips	231	22	39D6	10/27/87 10:32 AM	GeoFont 1.4	
Callaghan	232	12	3A0C	10/20/87 11:43 AM	GeoFont 1.4	
		20	3A14			
		25	3A19			
America	233	24	3A58	10/20/87 11:45 AM	GeoFont 1.4	
Etcheverry	234	24	3A98	10/20/87 11:46 AM	GeoFont 1.4	
Ellsworth	235	12	3ACC	10/20/87 11:46 AM	GeoFont 1.4	
		24	3AD8			
Spook	236	26	3B1A	10/20/87 11:47 AM	GeoFont 1.4	
Lawrence	237	12	3B4C	10/20/87 11:14 AM	GeoFont 1.4	
Spruce	238	12	3B8C	10/20/87 11:48 AM	GeoFont 1.4	
		24	3B98			
Moffit	239	12	3BCC	10/20/87 11:48 AM	GeoFont 1.4	
		24	3BD8			
Hilgard 1	240	28	3C1C	10/20/87 11:49 AM	GeoFont 1.4	
Hilgard 2	241	28	3C5C	10/20/87 11:49 AM	GeoFont 1.4	
Wheeler	242	9	3C89	10/20/87 11:51 AM	GeoFont 1.4	
		12	3C8C			
		14	3C8E			
		18	3C92			
		24	3C98			
FONTPACK1 fonts	*Includes FontKnox, Mykonos, Boalt, Stadium, Durant, Telegraph, Superb, Ormond, Brennens, LeConte from FONTPACK1. Only the Brennens font is an updated font.*					
Brennens	*22*	*18*	*0592*	*9/5/86 3:11 PM*	*Brennens V1.1*	

Note: If you really want to get the best print output from GEOS you need to use a PostScript equipped Laser or Inkjet printer. In order to maximize your time and efforts the best resource for Laser Printing is **Dale Sidebottom's Laser Lovers Disk** available from the LyonLabs web site. Download and digest the PostScript Printing from the Commodore PDF and then grab the two D81 images. Modified versions of all of the LW Fonts are on the Laser Lovers Disk.

G: Special Notes

Desk Accessory

Responsibilities:

1. It is the job of the DA to ensure that if the current drive (**curDrive**) is changed, that it be returned to its original value so that **RstrAppl** can find the SWAP FILE. (C64 GEOS)
2. Must fill its' screen section with the appropriate screen color.
3. Must not use the top 16 scanlines of the screen.
4. Must set its' own sprite picture data, colors, positions, and X/Y doubling information.
5. Must only use a specific, contiguous area of application memory space as defined in the DA's header block. Any other memory used by the DA must be manually backed up and restored.

Note: Applications are responsible for backing up and restoring sprite data if they are using that area. DA's may freely use the area the from **spr1pic** thru **spr7pic** without needing to backup/restore the data there.

Restrictions:

1. Since Desk Accessories and Dialogs both save the system state to **dlgBoxRamBuf**, a DA cannot use Dialog Boxes unless it does a backup of **dlgBoxRamBuf** (417 bytes @851F) and restores it before the DA closes. Without backing that area up, calling a dialog box will trash the system state of the calling application and the calling application can no longer be restored. †

2. Desk accessories larger than 24K cannot be used under GEOS 128. This is the amount of space available in backRAM for desk accessories.

Note: **r10L** RECVR_OPTS is obsolete and should always be assumed to be $00. Applications must always handle the saving and restoring of the foreground screen and color memory. See **LdDeskAcc Note** for more information.

†*For workarounds to these limitations see **Chapter 8 Dialog Box > Removing Limitations**.*

Auto Exec

Responsibilities:

Always check **firstBoot** at startup and behave accordingly based on the result:

- When **firstBoot** == FALSE; perform boot time logic.
- When **firstBoot** == TRUE; perform application mode logic. Normally this will be some form of user setup.

Restrictions:

1. The only available input driver is the joystick unless you load one in yourself.

2. Cannot modify RAM from $5000-5FFF when running during first boot. Kernal boot code is still active in this area during boot time when the auto exec is running.

3. If you need full drive support you must run *after* CONFIGURE.

4. Kernal patches should run *before* CONFIGURE so that CONFIGURE will stash the changes with the rest of the Kernal into REU bank 0 for rboot.